Neuronale Netze

Prof. Dr. Rudolf Kruse

Computational Intelligence
Institut für Wissens- und Sprachverarbeitung
Fakultät für Informatik
kruse@iws.cs.uni-magdeburg.de

Vorlesungsinhalte

- **Einleitung**
- **Schwellenwertelemente**
- **Allgemeine Neuronale Netze**
- **Mehrschichtige Perzeptren**
- **Radiale-Basis-Funktions-Netze**
- **Selbstorganisierende Karten**
- **Hopfield-Netze**
- **Rekurrente Neuronale Netze**
- **Support Vector Machines**
- **Neuro-Fuzzy-Systeme**

Motivation: Warum (künstliche) neuronale Netze?

- **(Neuro-)Biologie / (Neuro-)Physiologie / Psychologie:**
  - Ausnutzung der Ähnlichkeit zu echten (biologischen) neuronalen Netzen
  - Modellierung zum Verständnis Arbeitsweise von Nerven und Gehirn durch Simulation
- **Informatik / Ingenieurwissenschaften / Wirtschaft**
  - Nachahmen der menschlichen Wahrnehmung und Verarbeitung
  - Lösen von Lern-/Anpassungsproblemen sowie Vorhersage- und Optimierungsproblemen
- **Physik / Chemie**
  - Nutzung neuronaler Netze, um physikalische Phänomene zu beschreiben
  - Spezialfall: Spin-Glas (Legierungen von magnetischen und nicht-magnetischen Metallen)

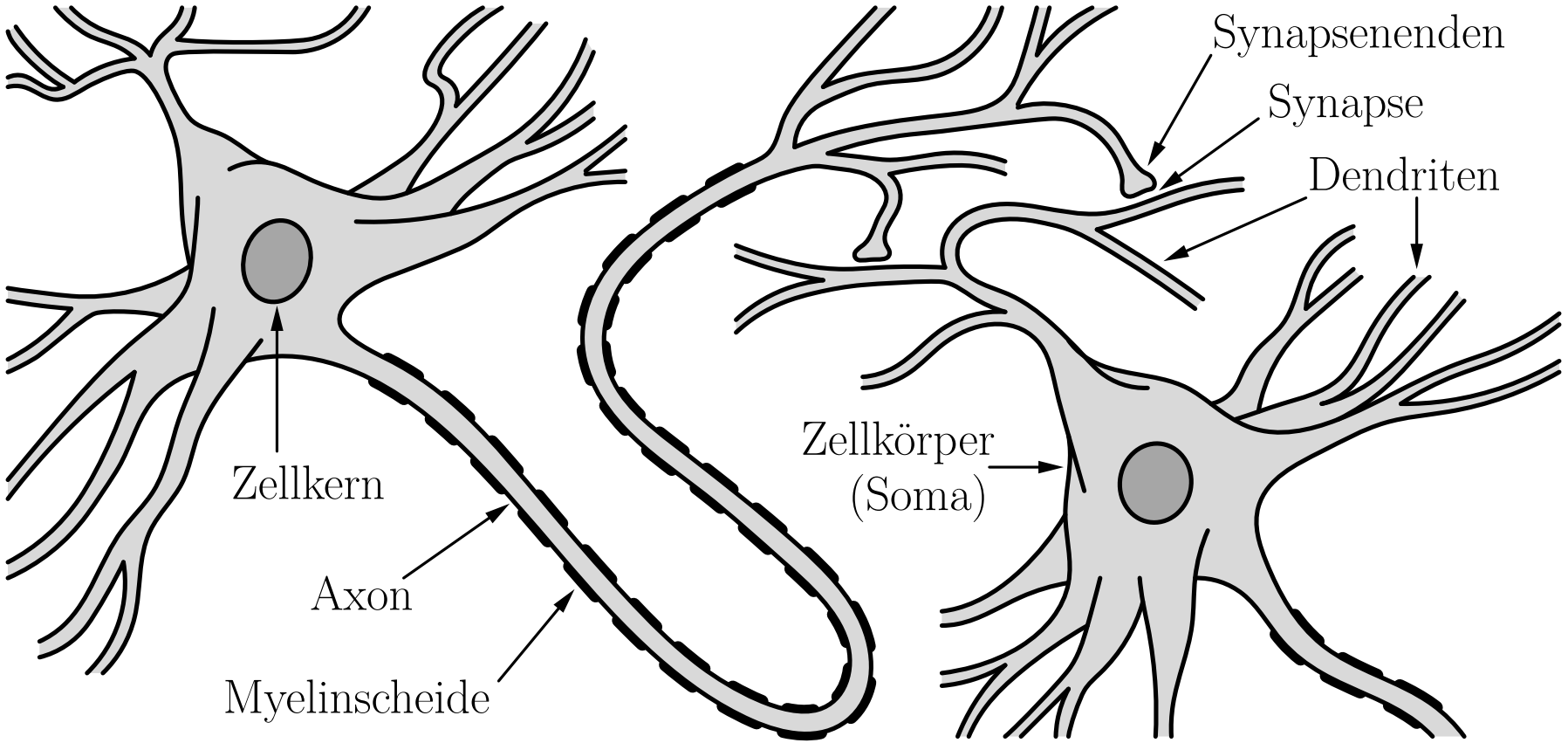
Konventionelle Rechner vs. Gehirn

	Computer	Gehirn
Verarbeitungseinheiten	1 CPU, 10^9 Transistoren	10^{11} Neuronen
Speicherkapazität	10^9 Bytes RAM, 10^{10} Bytes Festspeicher	10^{11} Neuronen, 10^{14} Synapsen
Verarbeitungsgeschwindigkeit	10^{-8} sec.	10^{-3} sec.
Bandbreite	$10^9 \frac{bits}{s}$	$10^{14} \frac{bits}{s}$
Neuronale Updates pro sec.	10^5	10^{14}

Konventionelle Rechner vs. Gehirn

- Beachte: die Hirnschaltzeit ist mit 10^{-3} s recht langsam, aber Updates erfolgen parallel. Dagegen braucht die serielle Simulation auf einem Rechner mehrere hundert Zyklen für ein Update.
- Vorteile neuronaler Netze:
  - Hohe Verarbeitungsgeschwindigkeit durch massive Parallelität
  - Funktionstüchtigkeit selbst bei Ausfall von Teilen des Netzes (Fehlertoleranz)
  - Langsamer Funktionsausfall bei fortschreitenden Ausfällen von Neuronen (*graceful degradation*)
  - Gut geeignet für induktives Lernen
- Es erscheint daher sinnvoll, diese Vorteile natürlicher neuronaler Netze künstlich nachzuahmen.

Struktur eines prototypischen biologischen Neurons



(Stark) vereinfachte Beschreibung neuronaler Informationsverarbeitung

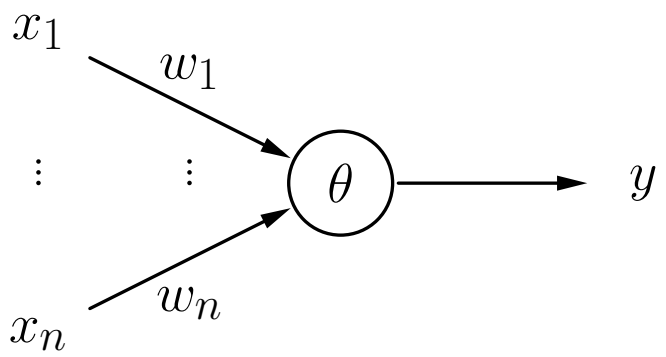
- Das Axonende gibt Chemikalien ab, **Neurotransmitter** genannt.
- Diese bewirken an der Membran des Empfängerdendriten die Veränderung der Polarisierung.
  (Das Innere ist typischerweise 70mV negativer als die Außenseite.)
- Abnahme in der Potentialdifferenz: **anregende** Synapse
  Zunahme in der Potentialdifferenz: **hemmende** Synapse
- Wenn genügend anregende Information vorhanden ist, wird das Axon depolarisiert.
- Das resultierende **Aktionspotential** pflanzt sich entlang des Axons fort.
  (Die Geschwindigkeit hängt von der Bedeckung mit Myelin ab.)
- Wenn das Aktionspotential die Synapsenenden erreicht,
  löst es die Abgabe von Neurotransmittern aus.

Schwellenwertelemente

Schwellenwertelemente

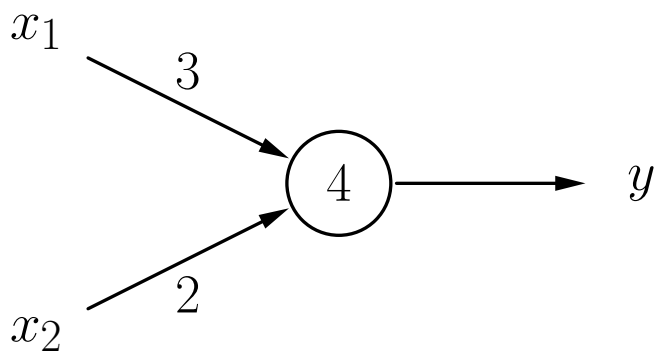
Ein **Schwellenwertelement** (Threshold Logic Unit, TLU) ist eine Verarbeitungseinheit für Zahlen mit n Eingängen $x_1, \ldots, x_n$ und einem Ausgang y. Das Element hat einen **Schwellenwert** θ und jeder Eingang x_i ist mit einem **Gewicht** w_i versehen. Ein Schwellenwertelement berechnet die Funktion

$$y = \begin{cases} 1, & \text{falls} \quad \boldsymbol{xw} = \sum_{i=1}^{n} w_i x_i \geq \theta, \\ 0, & \text{sonst.} \end{cases}$$



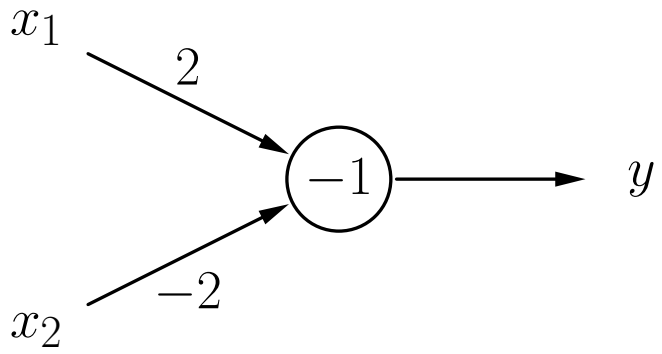
Schwellenwertelemente: Beispiele

Schwellenwertelement für die Konjunktion $x_1 \wedge x_2$.



x_1	x_2	$3x_1 + 2x_2$	y
0	0	0	0
1	0	3	0
0	1	2	0
1	1	5	1

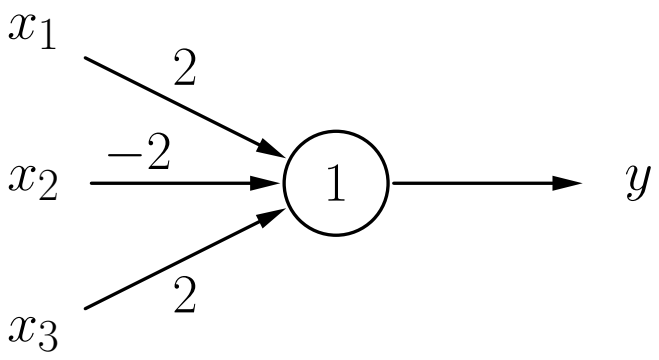
Schwellenwertelement für die Implikation $x_2 \rightarrow x_1$.



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

Schwellenwertelemente: Beispiele

Schwellenwertelement für $(x_1 \wedge \overline{x_2}) \vee (x_1 \wedge x_3) \vee (\overline{x_2} \wedge x_3)$**.**



x_1	x_2	x_3	$\Sigma_i\, w_i x_i$	y
0	0	0	0	0
1	0	0	2	1
0	1	0	−2	0
1	1	0	0	0
0	0	1	2	1
1	0	1	4	1
0	1	1	0	0
1	1	1	2	1

Rückblick: Geradendarstellungen

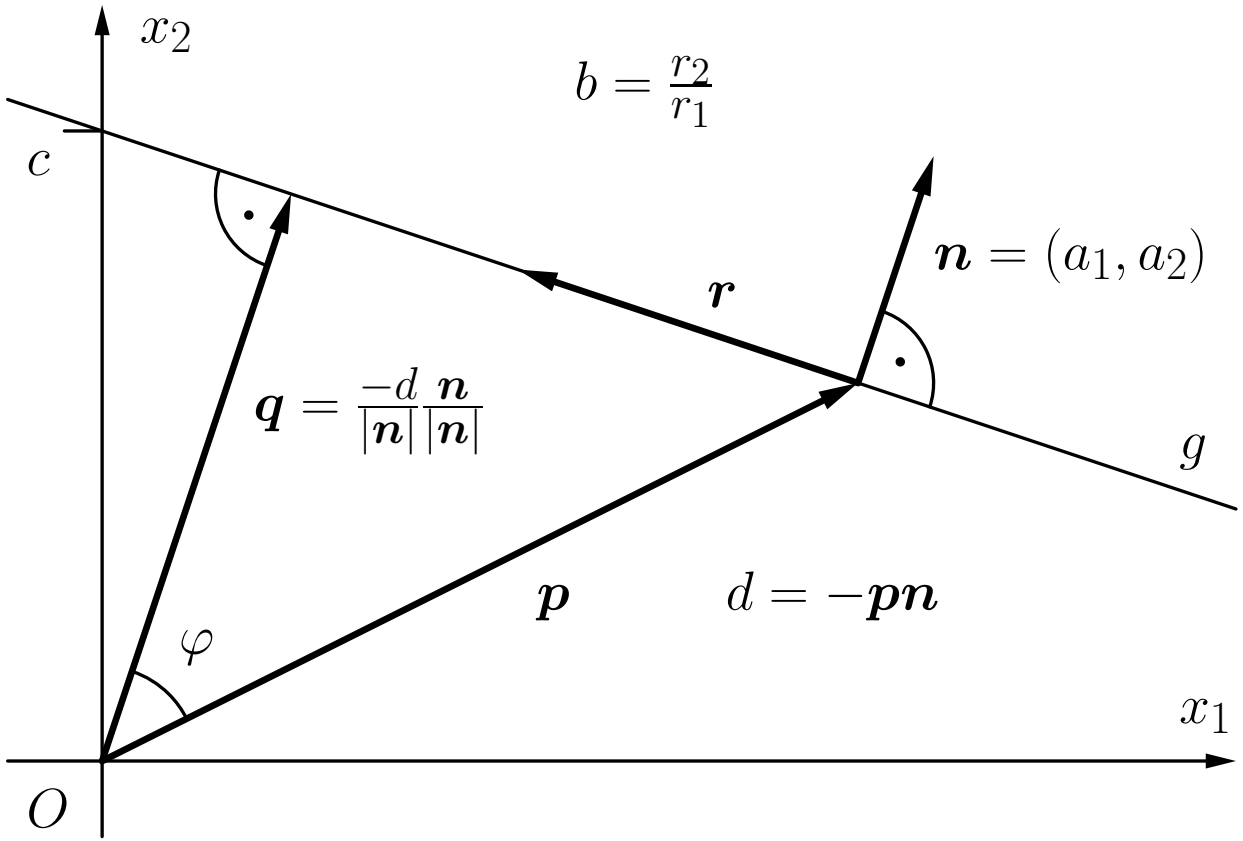
Geraden werden typischerweise in einer der folgenden Formen dargestellt:

$$
\begin{array}{lcl}
\text{Explizite Form:} & g & \equiv \quad x_2 = bx_1 + c \\
\text{Implizite Form:} & g & \equiv \quad a_1 x_1 + a_2 x_2 + d = 0 \\
\text{Punkt-Richtungs-Form:} & g & \equiv \quad \boldsymbol{x} = \boldsymbol{p} + k\boldsymbol{r} \\
\text{Normalform} & g & \equiv \quad (\boldsymbol{x} - \boldsymbol{p})\boldsymbol{n} = 0
\end{array}
$$

mit den Parametern

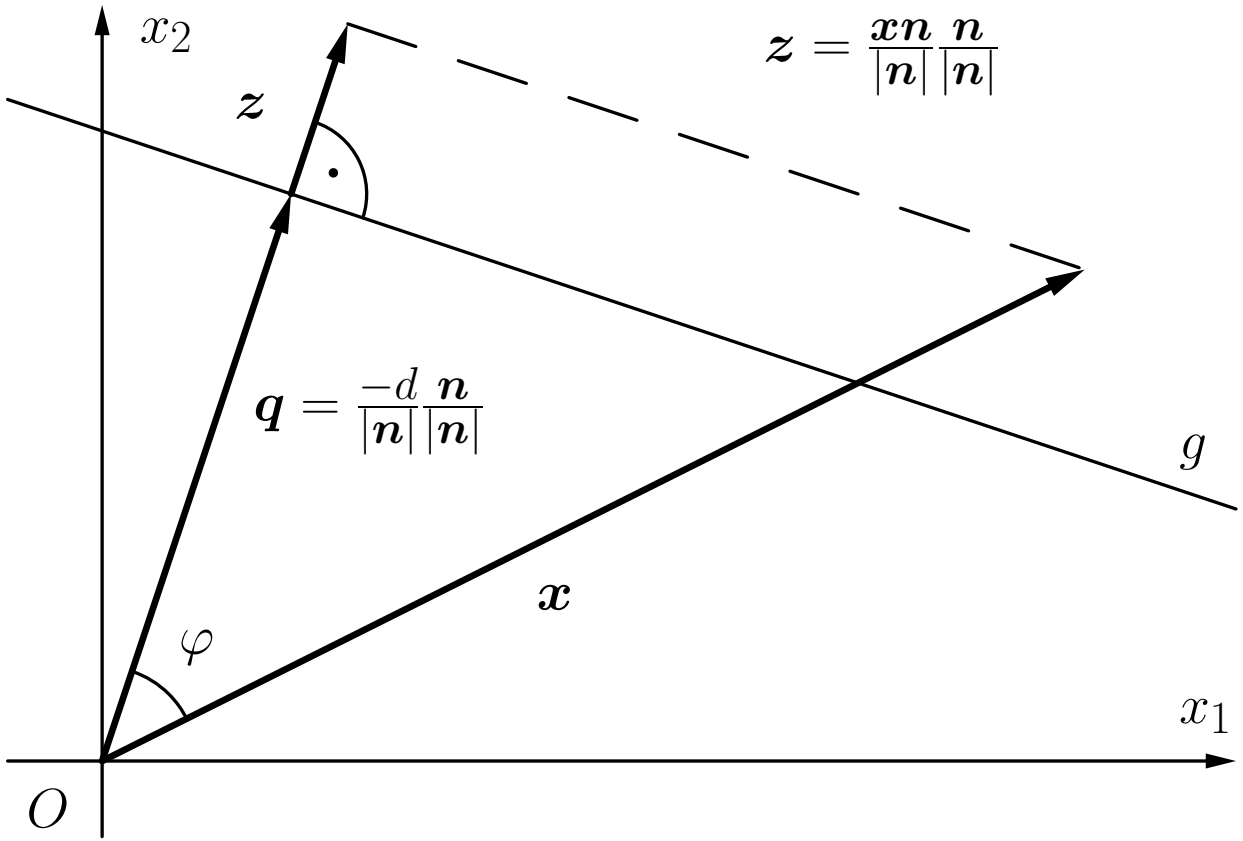
b : Anstieg der Geraden
c : Abschnitt der x_2-Achse
$\boldsymbol{p}$: Vektor zu einem Punkt auf der Gerade (Ortsvektor)
$\boldsymbol{r}$: Richtungsvektor der Gerade
$\boldsymbol{n}$: Normalenvektor der Gerade

Eine Gerade und ihre definierenden Eigenschaften.



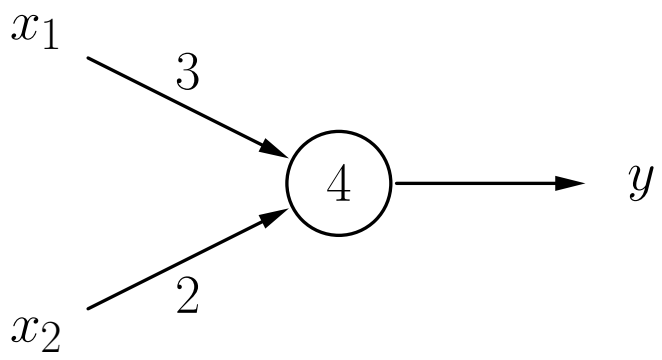
Schwellenwertelemente: Geometrische Interpretation

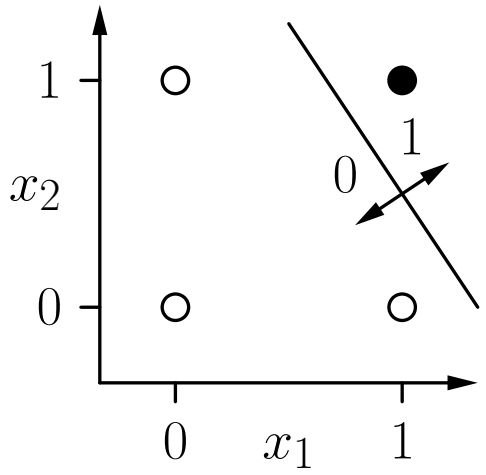
Bestimmung, auf welcher Seite ein Punkt $\boldsymbol{x}$ liegt.



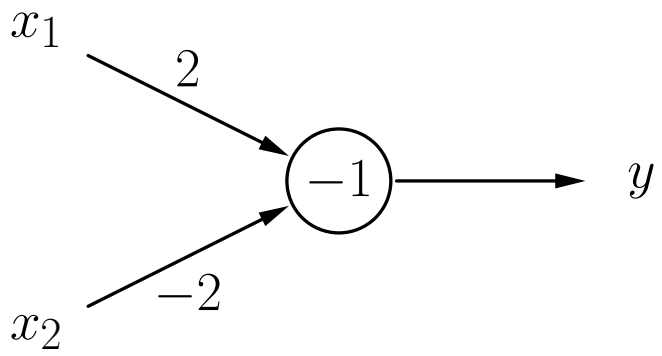
Schwellenwertelemente: Geometrische Interpretation

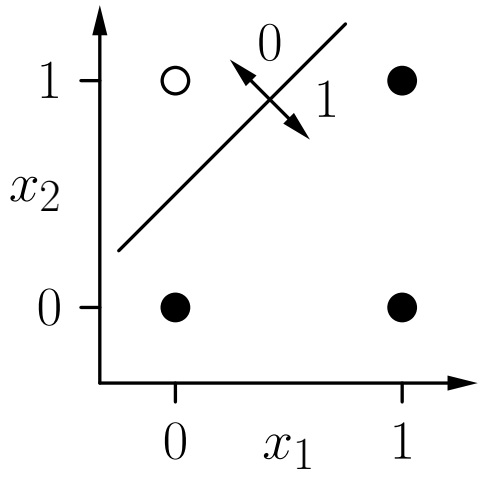
Schwellenwertelement für $x_1 \wedge x_2$.





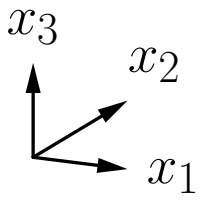
Ein Schwellenwertelement für $x_2 \rightarrow x_1$.

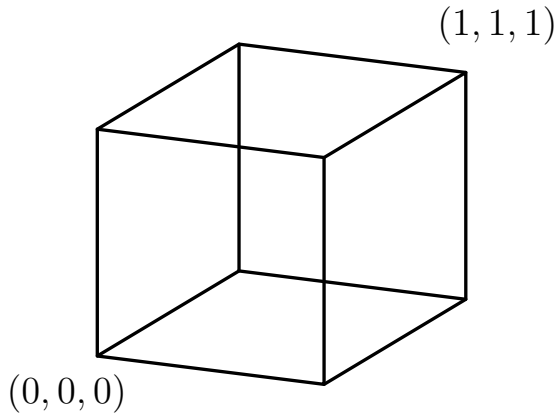




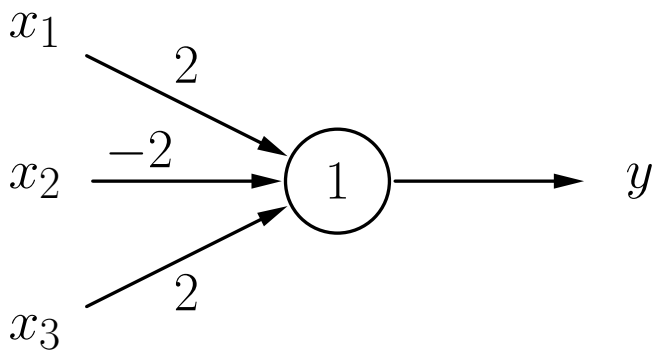
Schwellenwertelemente: Geometrische Interpretation

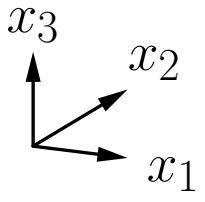
Darstellung 3-dimensionaler Boolescher Funktionen:

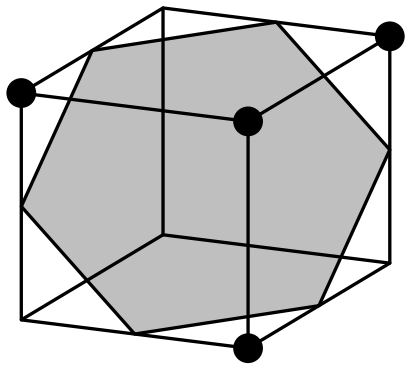




Schwellenwertelement für $(x_1 \wedge \overline{x_2}) \vee (x_1 \wedge x_3) \vee (\overline{x_2} \wedge x_3)$**.**







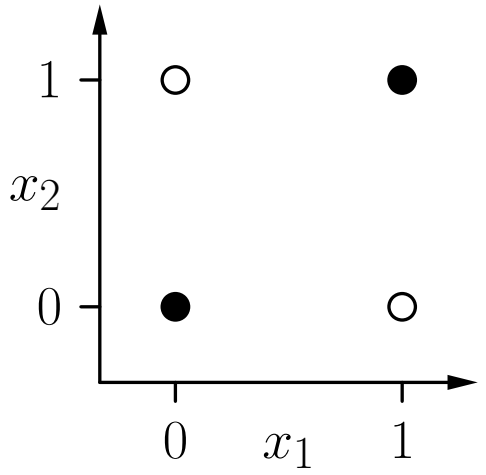
Schwellenwertelemente: lineare Separabilität

- Zwei Punktmengen in einem n-dimensionalen Raum heißen linear separabel, wenn sie durch eine (n-1)-dimensionale Hyperebene getrennt werden können. Die Punkte der einen Menge dürfen dabei auch auf der Hyperebene liegen.

- Eine Boolesche Funktion heißt linear separabel, falls die Menge der Urbilder von 0 und die Menge der Urbilder von 1 linear separabel sind.

Das Biimplikationsproblem $x_1 \leftrightarrow x_2$: Es gibt keine Trenngerade.

x_1	x_2	y
0	0	1
1	0	0
0	1	0
1	1	1



Formaler Beweis durch *reductio ad absurdum*:

$$\begin{array}{lll} \text{da } (0,0) \mapsto 1: & 0 & \geq \theta, \quad (1) \\ \text{da } (1,0) \mapsto 0: & w_1 & < \theta, \quad (2) \\ \text{da } (0,1) \mapsto 0: & w_2 & < \theta, \quad (3) \\ \text{da } (1,1) \mapsto 1: & w_1 + w_2 & \geq \theta. \quad (4) \end{array}$$

(2) und (3): $w_1 + w_2 < 2\theta$. Mit (4): $2\theta > \theta$, oder $\theta > 0$. Widerspruch zu (1).

Schwellenwertelemente: Grenzen

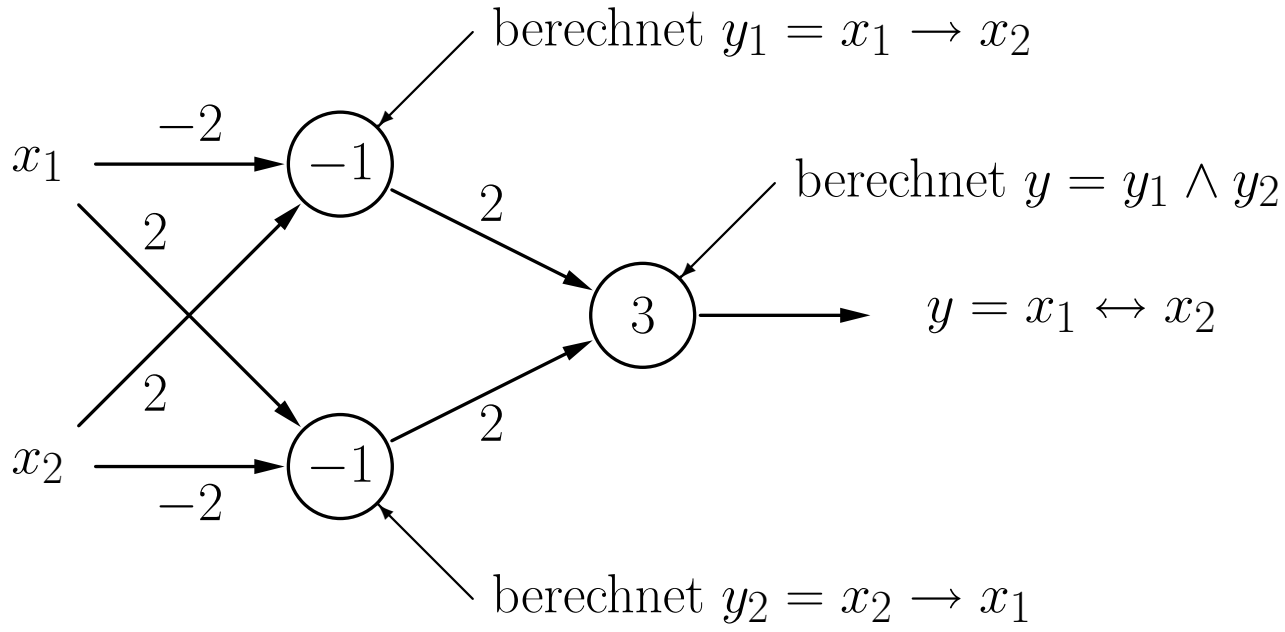
Vergleich zwischen absoluter Anzahl und der Anzahl linear separabler Boolescher Funktionen.
([Widner 1960] zitiert in [Zell 1994])

Eingaben	Boolesche Funktionen	linear separable Funktionen
1	4	4
2	16	14
3	256	104
4	65536	1774
5	$4.3 \cdot 10^9$	94572
6	$1.8 \cdot 10^{19}$	$5.0 \cdot 10^6$

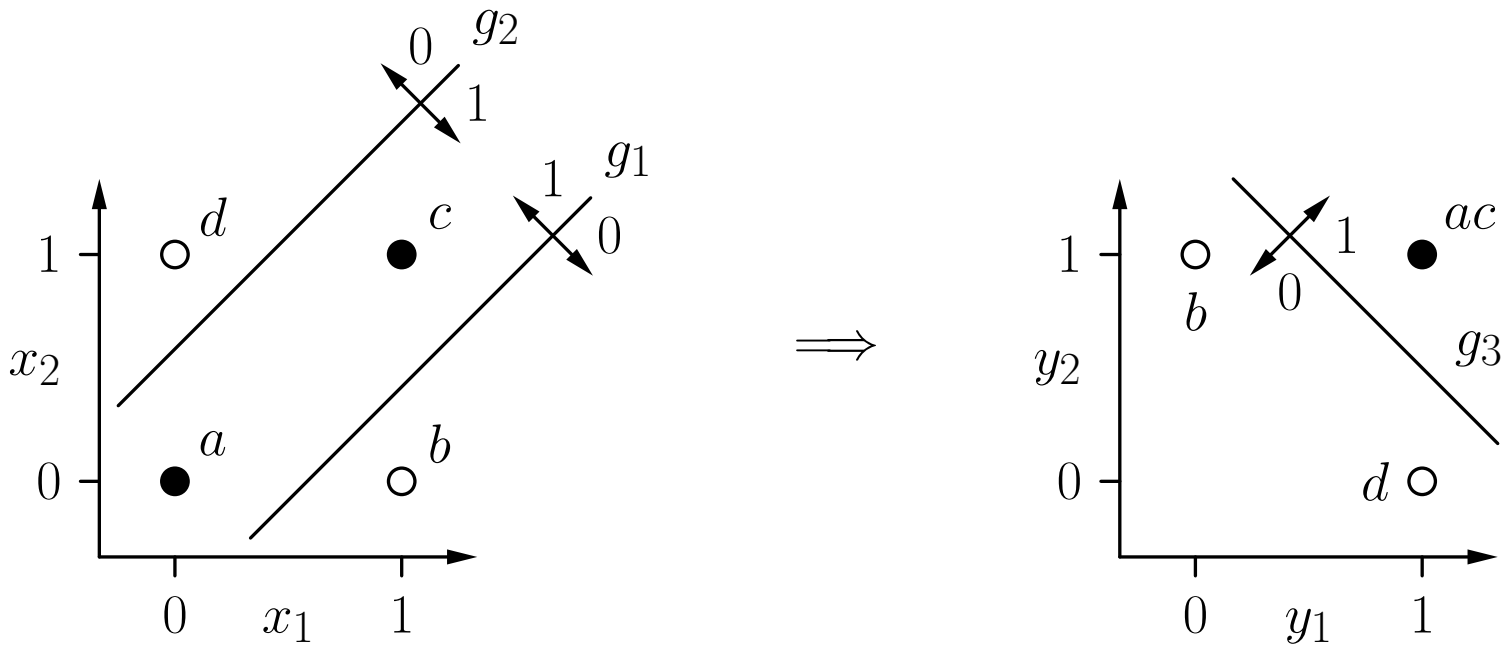
- Für viele Eingaben kann ein SWE fast keine Funktion berechnen.
- Netze aus Schwellenwertelementen sind notwendig, um die Berechnungsfähigkeiten zu erweitern.

Biimplikationsproblem, Lösung durch ein Netzwerk.

Idee: logische Zerlegung $x_1 \leftrightarrow x_2 \quad \equiv \quad (x_1 \rightarrow x_2) \wedge (x_2 \rightarrow x_1)$



Lösung des Biimplikationsproblems: Geometrische Interpretation



- Die erste Schicht berechnet neue Boolesche Koordinaten für die Punkte.
- Nach der Koordinatentransformation ist das Problem linear separabel.

Darstellung beliebiger Boolescher Funktionen

Sei $y = f(x_1, \ldots, x_n)$ eine Boolesche Funktion mit n Variablen.

(i) Stelle $f(x_1, \ldots, x_n)$ in disjunktiver Normalform dar. D.h. bestimme $D_f = K_1 \vee \ldots \vee K_m$, wobei alle K_j Konjunktionen von n Literalen sind, d.h., $K_j = l_{j1} \wedge \ldots \wedge l_{jn}$ mit $l_{ji} = x_i$ (positives Literal) oder $l_{ji} = \neg x_i$ (negatives Literal).

(ii) Lege ein Neuron für jede Konjunktion K_j der disjunktiven Normalform an (mit n Eingängen — ein Eingang pro Variable), wobei

$$w_{ji} = \begin{cases} 2, & \text{falls } l_{ji} = x_i, \\ -2, & \text{falls } l_{ji} = \neg x_i, \end{cases} \qquad \text{und} \qquad \theta_j = n - 1 + \frac{1}{2} \sum_{i=1}^{n} w_{ji}.$$

(iii) Lege ein Ausgabeneuron an (mit m Eingängen — ein Eingang für jedes Neuron, das in Schritt (ii) angelegt wurde), wobei

$$w_{(n+1)k} = 2, \quad k = 1, \ldots, m, \qquad \text{und} \qquad \theta_{n+1} = 1.$$

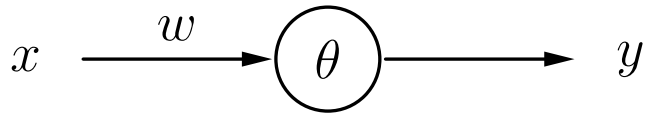
Trainieren von Schwellenwertelementen

Trainieren von Schwellenwertelementen

- Die geometrische Interpretation bietet eine Möglichkeit, SWE mit 2 und 3 Eingängen zu konstruieren, aber:
  - Es ist keine automatische Methode (Visualisierung und Begutachtung ist nötig).
  - Nicht möglich für mehr als drei Eingabevariablen.
- **Grundlegende Idee des automatischen Trainings:**
  - Beginne mit zufälligen Werten für Gewichte und Schwellenwert.
  - Bestimme den Ausgabefehler für eine Menge von Trainingsbeispielen.
  - Der Fehler ist eine Funktion der Gewichte und des Schwellenwerts: $e = e(w_1, \ldots, w_n, \theta)$.
  - Passe Gewichte und Schwellenwert so an, dass der Fehler kleiner wird.
  - Wiederhole diese Anpassung, bis der Fehler verschwindet.

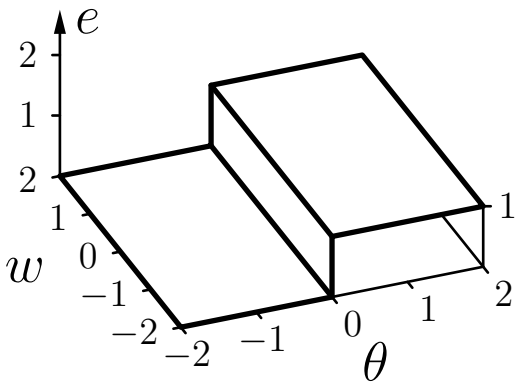
Trainieren von Schwellenwertelementen

Schwellenwertelement mit einer Eingabe für die Negation $\neg x$.

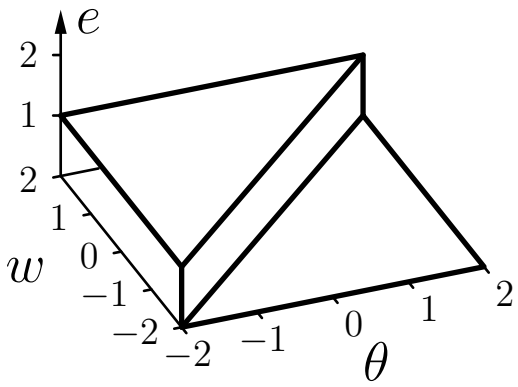


x	y
0	1
1	0

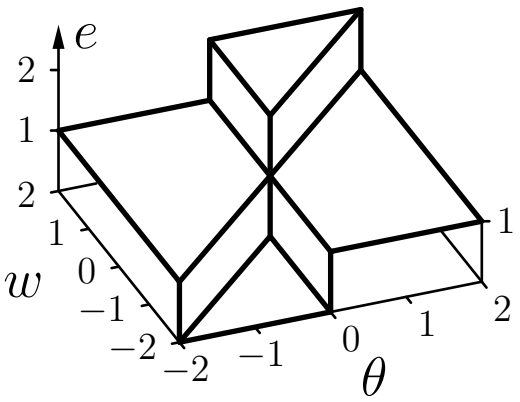
Ausgabefehler als eine Funktion von Gewicht und Schwellenwert.



Fehler für $x = 0$



Fehler für $x = 1$

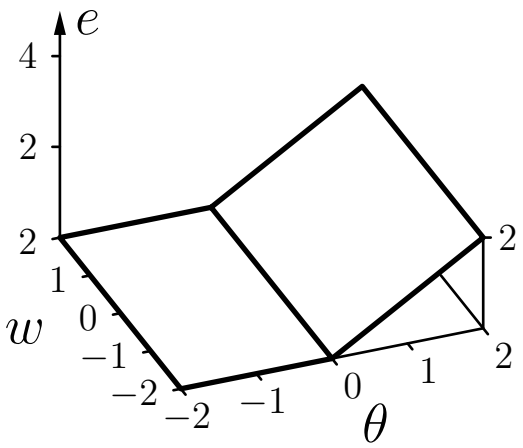


Summe der Fehler

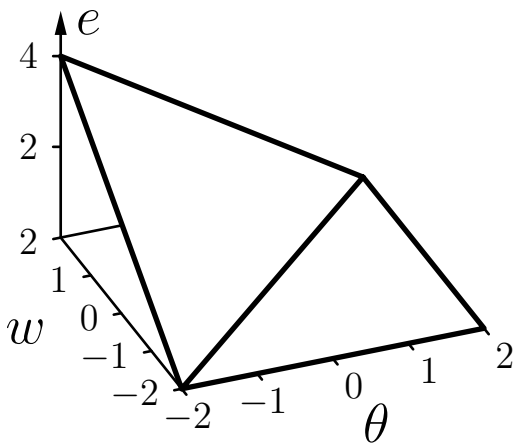
Trainieren von Schwellenwertelementen

- Die Fehlerfunktion kann nicht direkt verwendet werden, da sie aus Plateaus besteht.
- Lösung: Falls die berechnete Ausgabe falsch ist,
  berücksichtige, wie weit die gewichtete Summe vom Schwellenwert entfernt ist.

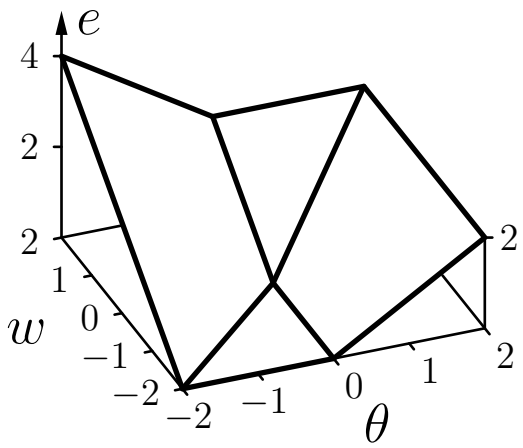
Modifizierter Ausgabefehler als Funktion von Gewichten und Schwellenwert.



Fehler für $x = 0$



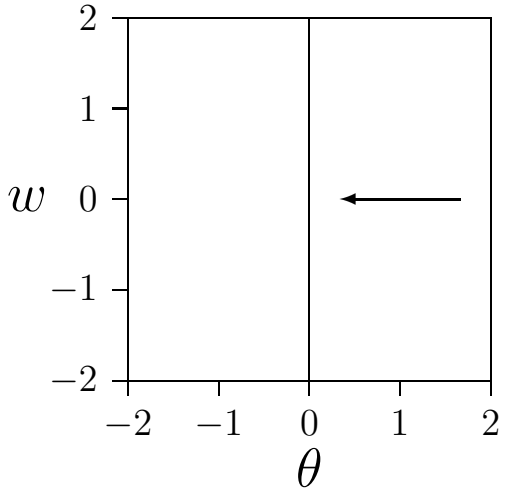
Fehler für $x = 1$



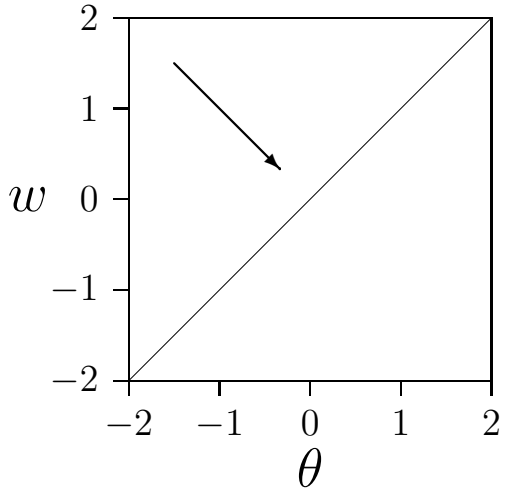
Summe der Fehler

Trainieren von Schwellenwertelementen

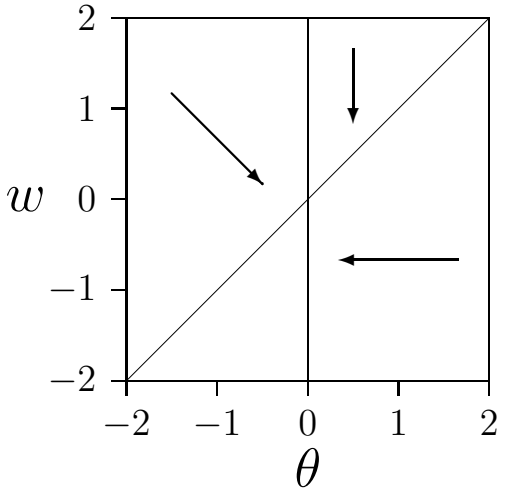
Schema der resultierenden Richtungen der Parameteränderungen.



Änderungen für $x = 0$



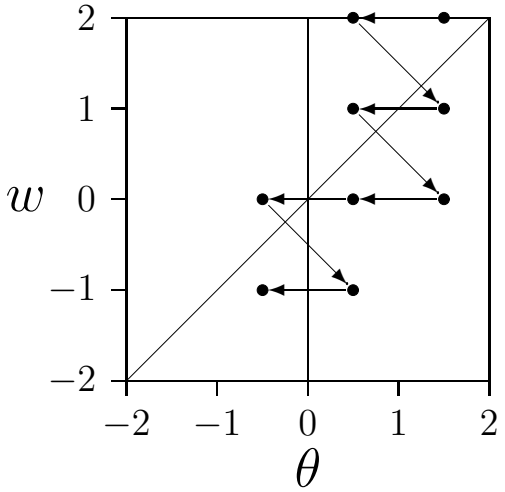
Änderungen für $x = 1$



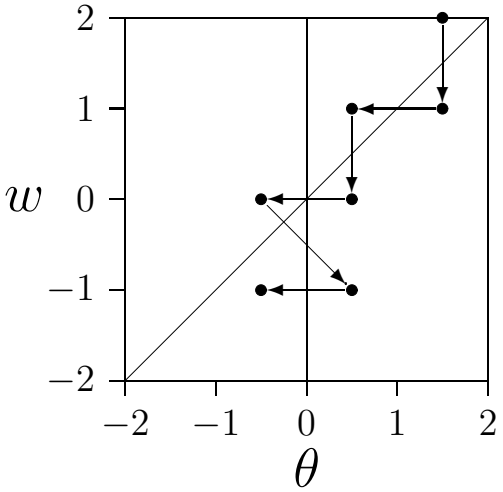
Summe der Änderungen

- Beginne an zufälligem Punkt.
- Passe Parameter iterativ an,
  entsprechend der zugehörigen Richtung am aktuellen Punkt.

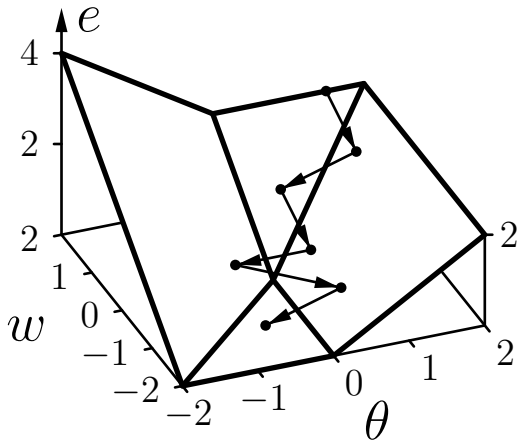
Beispieltrainingsprozedur: Online- und Batch-Training.



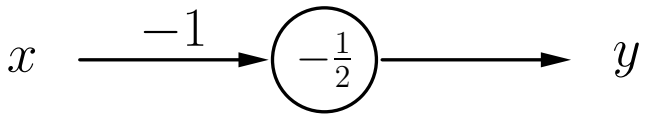
Online-Lernen

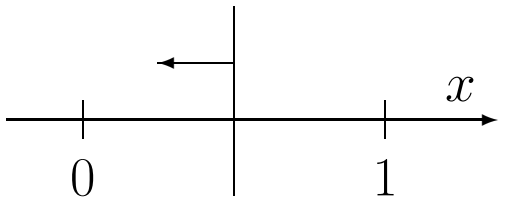


Batch-Lernen



Batch-Lernen





Trainieren von Schwellenwertelementen: Delta-Regel

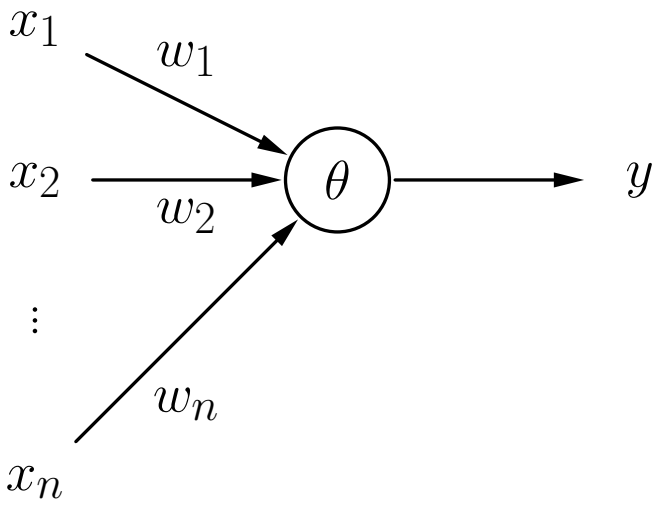
Formale Trainingsregel: Sei $\boldsymbol{x} = (x_1, \ldots, x_n)$ ein Eingabevektor eines Schwellenwertelements, o die gewünschte Ausgabe für diesen Eingabevektor, und y die momentane Ausgabe des Schwellenwertelements. Wenn $y \neq o$, dann werden Schwellenwert θ und Gewichtsvektor $\boldsymbol{w} = (w_1, \ldots, w_n)$ wie folgt angepasst, um den Fehler zu reduzieren:

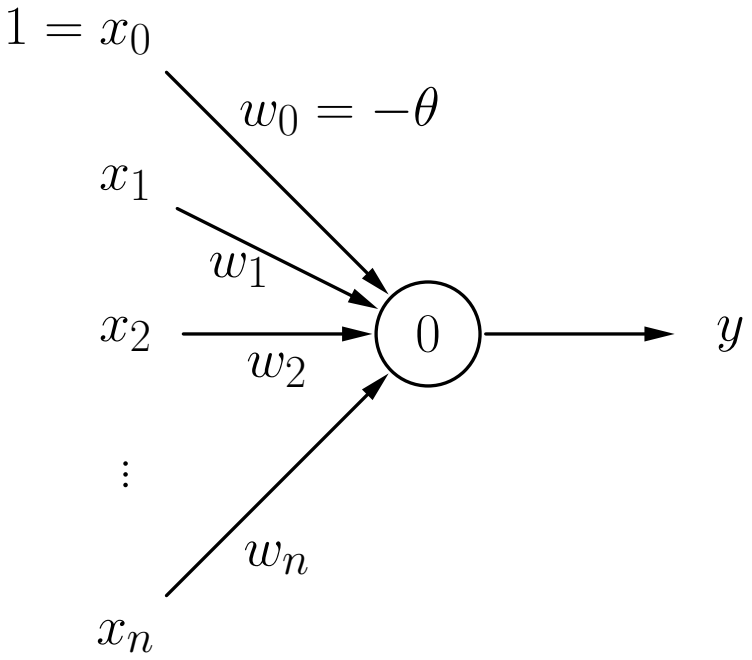
$$\begin{aligned} \theta^{(\text{neu})} &= \theta^{(\text{alt})} + \Delta\theta \quad \text{wobei} \quad \Delta\theta &&= -\eta(o - y), \\ \forall i \in \{1, \ldots, n\}: \quad w_i^{(\text{neu})} &= w_i^{(\text{alt})} + \Delta w_i \quad \text{wobei} \quad \Delta w_i &&= \eta(o - y)x_i, \end{aligned}$$

wobei η ein Parameter ist, der **Lernrate** genannt wird. Er bestimmt die Größenordnung der Gewichtsänderungen. Diese Vorgehensweise nennt sich **Delta-Regel** oder **Widrow–Hoff–Procedure** [Widrow and Hoff 1960].

- **Online-Training:** Passe Parameter nach jedem Trainingsmuster an.
- **Batch-Training:** Passe Parameter am Ende jeder **Epoche** an, d.h. nach dem Durchlaufen aller Trainingsbeispiele.

Ändern des Schwellenwerts in ein Gewicht:





$$\sum_{i=1}^{n} w_i x_i \geq \theta$$

$$\sum_{i=1}^{n} w_i x_i - \theta \geq 0$$

Trainieren von Schwellenwertelementen: Delta-Regel

```
procedure online_training (var w, var θ, L, η);
var y, e;                                   (* Ausgabe, Fehlersumme *)
begin
  repeat
    e := 0;                                 (* initialisiere Fehlersumme *)
    for all (x, o) ∈ L do begin             (* durchlaufe Trainingsmuster*)
      if (wx ≥ θ) then  y := 1;             (* berechne Ausgabe*)
                  else  y := 0;             (* des Schwellenwertelements *)
      if (y ≠ o) then begin                 (* Falls Ausgabe falsch *)
        θ := θ − η(o − y);                  (* passe Schwellenwert *)
        w := w + η(o − y)x;                 (* und Gewichte an *)
        e := e + |o − y|;                   (* summiere die Fehler*)
      end;
    end;
  until (e ≤ 0);                            (* wiederhole die Berechnungen*)
end;                                        (* bis der Fehler verschwindet*)
```

Trainieren von Schwellenwertelementen: Delta-Regel

```
procedure batch_training (var w, var θ, L, η);
var y, e,                                   (* Ausgabe, Fehlersumme *)
    θ_c, w_c;                               (* summierte Änderungen *)
begin
  repeat
    e := 0; θ_c := 0; w_c := 0;             (* Initialisierungen *)
    for all (x, o) ∈ L do begin             (* durchlaufe Trainingsbeispiele*)
      if (wx ≥ θ) then  y := 1;             (* berechne Ausgabe *)
                  else  y := 0;             (* des Schwellenwertelements *)
      if (y ≠ o) then begin                 (* Falls Ausgabe falsch*)
        θ_c := θ_c - η(o - y);              (* summiere die Änderungen von*)
        w_c := w_c + η(o - y)x;             (* Schwellenwert und Gewichten *)
        e   := e   +  |o - y|;              (* summiere Fehler*)
      end;
    end;
    θ := θ + θ_c;                           (* passe Schwellenwert*)
    w := w + w_c;                           (* und Gewichte an *)
  until (e ≤ 0);                            (* wiederhole Berechnungen *)
end;                                        (* bis der Fehler verschwindet*)
```

Trainieren von Schwellenwertelementen: Online

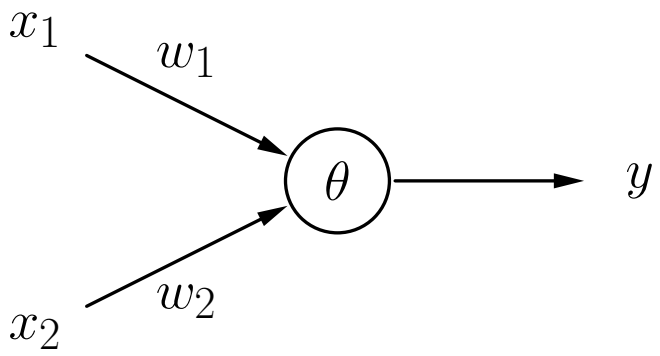
Epoche	x	o	$\boldsymbol{xw}$	y	e	$\Delta\theta$	Δw	θ	w
								1.5	2
1	0	1	−1.5	0	1	−1	0	0.5	2
	1	0	1.5	1	−1	1	−1	1.5	1
2	0	1	−1.5	0	1	−1	0	0.5	1
	1	0	0.5	1	−1	1	−1	1.5	0
3	0	1	−1.5	0	1	−1	0	0.5	0
	1	0	0.5	0	0	0	0	0.5	0
4	0	1	−0.5	0	1	−1	0	−0.5	0
	1	0	0.5	1	−1	1	−1	0.5	−1
5	0	1	−0.5	0	1	−1	0	−0.5	−1
	1	0	−0.5	0	0	0	0	−0.5	−1
6	0	1	0.5	1	0	0	0	−0.5	−1
	1	0	−0.5	0	0	0	0	−0.5	−1

Trainieren von Schwellenwertelementen: Batch

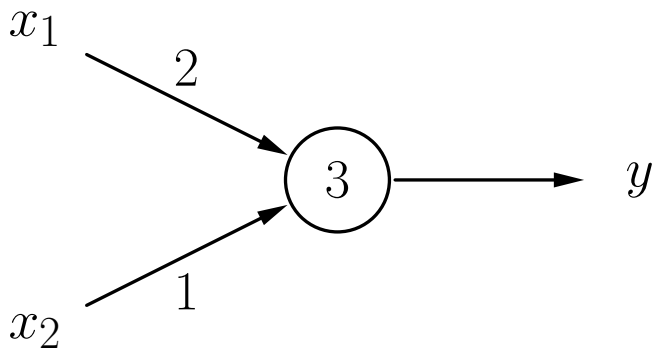
Epoche	x	o	$\boldsymbol{xw}$	y	e	$\Delta\theta$	Δw	θ	w
								1.5	2
1	0	1	−1.5	0	1	−1	0		
	1	0	0.5	1	−1	1	−1	1.5	1
2	0	1	−1.5	0	1	−1	0		
	1	0	−0.5	0	0	0	0	0.5	1
3	0	1	−0.5	0	1	−1	0		
	1	0	0.5	1	−1	1	−1	0.5	0
4	0	1	−0.5	0	1	−1	0		
	1	0	−0.5	0	0	0	0	−0.5	0
5	0	1	0.5	1	0	0	0		
	1	0	0.5	1	−1	1	−1	0.5	−1
6	0	1	−0.5	0	1	−1	0		
	1	0	−1.5	0	0	0	0	−0.5	−1
7	0	1	0.5	1	0	0	0		
	1	0	−0.5	0	0	0	0	−0.5	−1

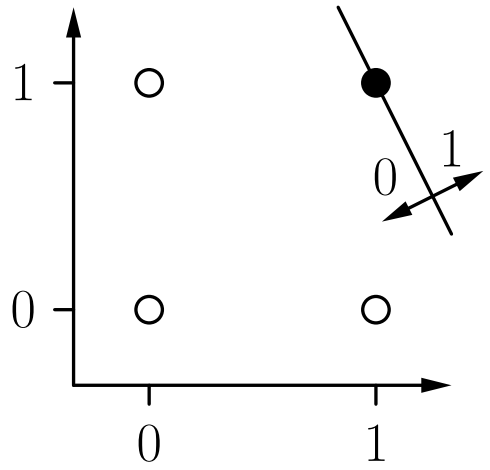
Trainieren von Schwellenwertelementen: Konjunktion

Schwellenwertelement mit zwei Eingängen für die Konjunktion.



x_1	x_2	y
0	0	0
1	0	0
0	1	0
1	1	1





Trainieren von Schwellenwertelementen: Konjunktion

Epoche	x_1	x_2	o	$\boldsymbol{xw}$	y	e	$\Delta\theta$	Δw_1	Δw_2	θ	w_1	w_2
										0	0	0
1	0	0	0	0	1	−1	1	0	0	1	0	0
	0	1	0	−1	0	0	0	0	0	1	0	0
	1	0	0	−1	0	0	0	0	0	1	0	0
	1	1	1	−1	0	1	−1	1	1	0	1	1
2	0	0	0	0	1	−1	1	0	0	1	1	1
	0	1	0	0	1	−1	1	0	−1	2	1	0
	1	0	0	−1	0	0	0	0	0	2	1	0
	1	1	1	−1	0	1	−1	1	1	1	2	1
3	0	0	0	−1	0	0	0	0	0	1	2	1
	0	1	0	0	1	−1	1	0	−1	2	2	0
	1	0	0	0	1	−1	1	−1	0	3	1	0
	1	1	1	−2	0	1	−1	1	1	2	2	1
4	0	0	0	−2	0	0	0	0	0	2	2	1
	0	1	0	−1	0	0	0	0	0	2	2	1
	1	0	0	0	1	−1	1	−1	0	3	1	1
	1	1	1	−1	0	1	−1	1	1	2	2	2
5	0	0	0	−2	0	0	0	0	0	2	2	2
	0	1	0	0	1	−1	1	0	−1	3	2	1
	1	0	0	−1	0	0	0	0	0	3	2	1
	1	1	1	0	1	0	0	0	0	3	2	1
6	0	0	0	−3	0	0	0	0	0	3	2	1
	0	1	0	−2	0	0	0	0	0	3	2	1
	1	0	0	−1	0	0	0	0	0	3	2	1
	1	1	1	0	1	0	0	0	0	3	2	1

Trainieren von Schwellenwertelementen: Biimplikation

Epoch	x_1	x_2	o	$\boldsymbol{xw}$	y	e	$\Delta\theta$	Δw_1	Δw_2	θ	w_1	w_2
										0	0	0
1	0	0	1	0	1	0	0	0	0	0	0	0
	0	1	0	0	1	−1	1	0	−1	1	0	−1
	1	0	0	−1	0	0	0	0	0	1	0	−1
	1	1	1	−2	0	1	−1	1	1	0	1	0
2	0	0	1	0	1	0	0	0	0	0	1	0
	0	1	0	0	1	−1	1	0	−1	1	1	−1
	1	0	0	0	1	−1	1	−1	0	2	0	−1
	1	1	1	−3	0	1	−1	1	1	1	1	0
3	0	0	1	0	1	0	0	0	0	0	1	0
	0	1	0	0	1	−1	1	0	−1	1	1	−1
	1	0	0	0	1	−1	1	−1	0	2	0	−1
	1	1	1	−3	0	1	−1	1	1	1	1	0

Trainieren von Schwellenwertelementen: Konvergenz

Konvergenztheorem: Sei $L = \{(\boldsymbol{x}_1, o_1), \ldots (\boldsymbol{x}_m, o_m)\}$ eine Menge von Trainingsmustern, jedes bestehend aus einem Eingabevektor $\boldsymbol{x}_i \in \mathbb{R}^n$ und einer gewünschten Ausgabe $o_i \in \{0, 1\}$. Sei weiterhin $L_0 = \{(\boldsymbol{x}, o) \in L \mid o = 0\}$ und $L_1 = \{(\boldsymbol{x}, o) \in L \mid o = 1\}$. Falls L_0 und L_1 linear separabel sind, d.h., falls $\boldsymbol{w} \in \mathbb{R}^n$ und $\theta \in \mathbb{R}$ existieren, so dass

$$\begin{aligned} \forall (\boldsymbol{x}, 0) \in L_0 : &\quad \boldsymbol{w}\boldsymbol{x} < \theta \quad \text{und} \\ \forall (\boldsymbol{x}, 1) \in L_1 : &\quad \boldsymbol{w}\boldsymbol{x} \geq \theta, \end{aligned}$$

dann terminieren sowohl Online- als auch Batch-Training.

- Für nicht linear separable Probleme terminiert der Algorithmus nicht.

Trainieren von Netzwerken aus Schwellenwertelementen

- Einzelne Schwellenwertelemente haben starke Einschränkungen:
  Sie können nur linear separable Funktionen berechnen.
- Netzwerke aus Schwellenwertelemente können beliebige Boolesche Funktionen berechnen.
- Das Trainieren einzelner Schwellenwertelemente mit der Delta-Regel ist schnell und findet garantiert eine Lösung, falls eine existiert.
- Netzwerke aus Schwellenwertelementen können nicht mit der Delta-Regel trainiert werden.

Allgemeine (Künstliche) Neuronale Netze

Graphentheoretische Grundlagen

Ein (gerichteter) **Graph** ist ein Tupel $G = (V, E)$, bestehend aus einer (endlichen) Menge V von **Knoten** oder **Ecken** und einer (endlichen) Menge $E \subseteq V \times V$ von **Kanten**.

Wir nennen eine Kante $e = (u, v) \in E$ **gerichtet** von Knoten u zu Knoten v.

Sei $G = (V, E)$ ein (gerichteter) Graph und $u \in V$ ein Knoten. Dann werden die Knoten der Menge

$$\text{pred}(u) = \{v \in V \mid (v, u) \in E\}$$

die **Vorgänger** des Knotens u
und die Knoten der Menge

$$\text{succ}(u) = \{v \in V \mid (u, v) \in E\}$$

die **Nachfolger** des Knotens u genannt.

Allgemeine Neuronale Netze

Allgemeine Definition eines neuronalen Netzes Ein (künstliches) **neuronales Netz** ist ein (gerichteter) Graph $G = (U, C)$,
dessen Knoten $u \in U$ **Neuronen** oder **Einheiten** und
dessen Kanten $c \in C$ **Verbindungen** genannt werden.
Die Menge U der Knoten wird partitioniert in

- die Menge U_{in} der **Eingabeneuronen**,
- die Menge U_{out} der **Ausgabeneuronen**, und
- die Menge U_{hidden} der **versteckten Neuronen**.

Es gilt:

$$U = U_{\text{in}} \cup U_{\text{out}} \cup U_{\text{hidden}},$$

$$U_{\text{in}} \neq \emptyset, \qquad U_{\text{out}} \neq \emptyset, \qquad U_{\text{hidden}} \cap (U_{\text{in}} \cup U_{\text{out}}) = \emptyset.$$

Allgemeine Neuronale Netze

Jede Verbindung $(v, u) \in C$ besitzt ein **Gewicht** w_{uv} und
jedes Neuron $u \in U$ besitzt drei (reellwertige) Zustandsvariablen:

- die **Netzwerkeingabe** net_u,
- die **Aktivierung** act_u, und
- die **Ausgabe** out_u.

Jedes Eingabeneuron $u \in U_{\mathrm{in}}$ besitzt weiterhin eine vierte reellwertige Zustandsvariable,

- die **externe Eingabe** ex_u.

Weiterhin besitzt jedes Neuron $u \in U$ drei Funktionen:

- die **Netzwerkeingabefunktion** $f_{\mathrm{net}}^{(u)} : \mathbb{R}^{2|\,\mathrm{pred}(u)|+\kappa_1(u)} \to \mathbb{R}$,
- die **Aktivierungsfunktion** $f_{\mathrm{act}}^{(u)} : \mathbb{R}^{\kappa_2(u)} \to \mathbb{R}$, und
- die **Ausgabefunktion** $f_{\mathrm{out}}^{(u)} : \mathbb{R} \to \mathbb{R}$,

die benutzt werden, um die Werte der Zustandsvariablen zu berechnen.

Typen (künstlicher) neuronaler Netze

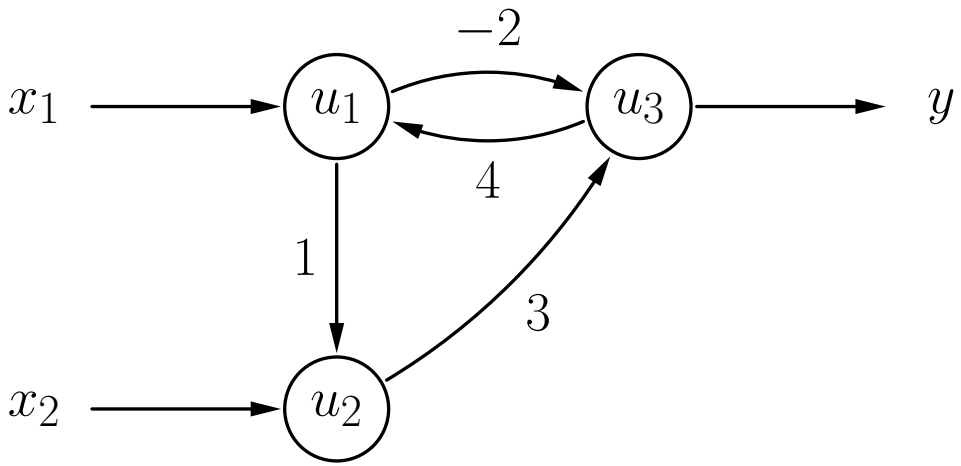
- Falls der Graph eines neuronalen Netzes **azyklisch** ist, wird das Netz **Feed-Forward-Netzwerk** genannt.
- Falls der Graph eines neuronalen Netzes **Zyklen** enthält, (rückwärtige Verbindungen), wird es **rekurrentes Netzwerk** genannt.

Darstellung der Verbindungsgewichte als Matrix

$$\begin{array}{c} \begin{array}{cccc} u_1 & u_2 & \dots & u_r \end{array} \\ \begin{pmatrix} w_{u_1u_1} & w_{u_1u_2} & \dots & w_{u_1u_r} \\ w_{u_2u_1} & w_{u_2u_2} & & w_{u_2u_r} \\ \vdots & & & \vdots \\ w_{u_ru_1} & w_{u_ru_2} & \dots & w_{u_ru_r} \end{pmatrix} \end{array} \begin{array}{c} u_1 \\ u_2 \\ \vdots \\ u_r \end{array}$$

Allgemeine Neuronale Netze: Beispiel

Ein einfaches rekurrentes neuronales Netz:

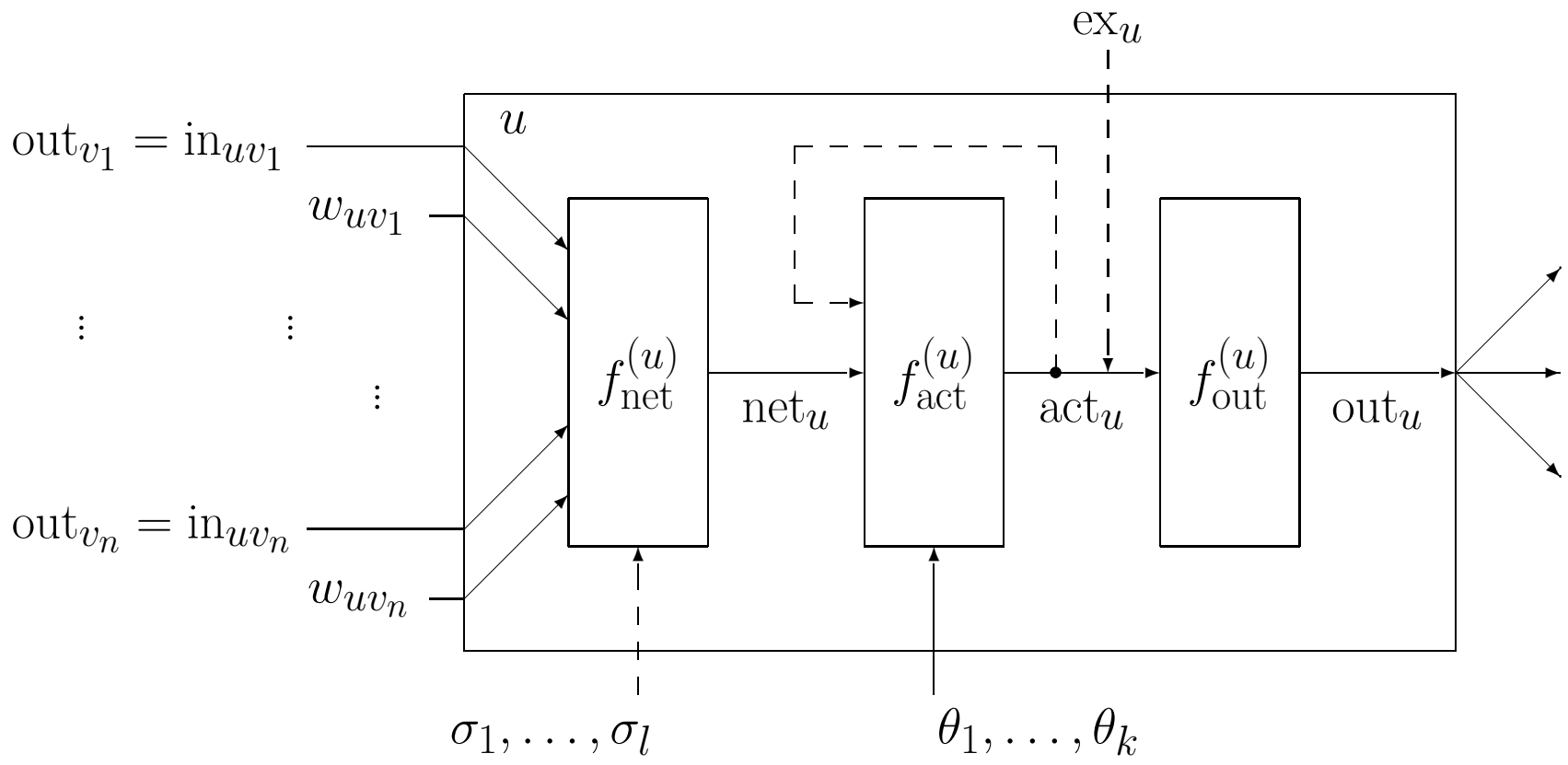


Gewichtsmatrix dieses Netzes:

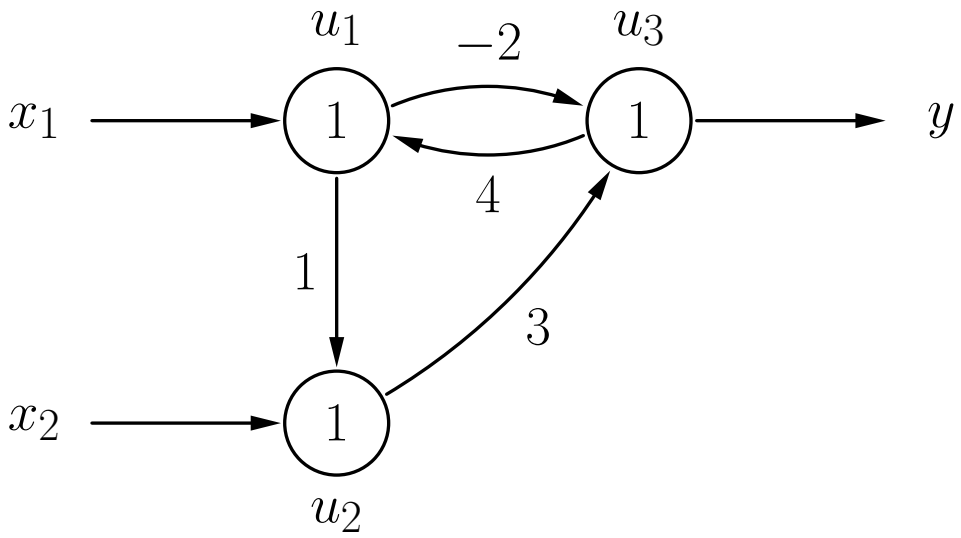
$$\begin{array}{c} \\ \begin{pmatrix} 0 & 0 & 4 \\ 1 & 0 & 0 \\ -2 & 3 & 0 \end{pmatrix} \end{array} \begin{array}{c} \begin{matrix} u_1 & u_2 & u_3 \end{matrix} \\ \begin{matrix} u_1 \\ u_2 \\ u_3 \end{matrix} \end{array}$$

Structure of a Generalized Neuron

Ein verallgemeinertes Neuron verarbeitet numerische Werte



Allgemeine Neuronale Netze: Beispiel



$$f_{\text{net}}^{(u)}(\boldsymbol{w}_u, \mathbf{in}_u) = \Sigma_{v \in \text{pred}(u)} w_{uv} \text{in}_{uv} = \Sigma_{v \in \text{pred}(u)} w_{uv} \, \text{out}_v$$

$$f_{\text{act}}^{(u)}(\text{net}_u, \theta) = \begin{cases} 1, & \text{falls} \quad \text{net}_u \geq \theta, \\ 0, & \text{sonst.} \end{cases}$$

$$f_{\text{out}}^{(u)}(\text{act}_u) = \text{act}_u$$

Aktualisierung der Neuronenaktivierungen

	u_1	u_2	u_3	
Eingabephase	**1**	**0**	**0**	
Arbeitsphase	1	0	**0**	$\text{net}_{u_3} = -2$
	0	0	0	$\text{net}_{u_1} = 0$
	0	**0**	0	$\text{net}_{u_2} = 0$
	0	0	**0**	$\text{net}_{u_3} = 0$
	0	0	0	$\text{net}_{u_1} = 0$

- Aktualisierungsreihenfolge:
  $u_3, u_1, u_2, u_3, u_1, u_2, u_3, \ldots$
- Eingabephase: Aktivierungen/Ausgaben im Startzustand (erste Zeile)
- Die Aktivierung des gerade zu aktualisierenden Neurons (fettgedruckt) wird mit Hilfe der Ausgaben der anderen Neuronen und der Gewichte neu berechnet.
- Ein stabiler Zustand mit eindeutiger Ausgabe wird erreicht.

Aktualisierung der Neuronenaktivierungen

	u_1	u_2	u_3	
Eingabephase	**1**	**0**	**0**	
Arbeitsphase	1	0	**0**	$\text{net}_{u_3} = -2$
	1	**1**	0	$\text{net}_{u_2} = 1$
	0	1	0	$\text{net}_{u_1} = 0$
	0	1	**1**	$\text{net}_{u_3} = 3$
	0	**0**	1	$\text{net}_{u_2} = 0$
	1	0	1	$\text{net}_{u_1} = 4$
	1	0	**0**	$\text{net}_{u_3} = -2$

- Aktualisierungsreihenfolge:
  $u_3, u_2, u_1, u_3, u_2, u_1, u_3, \ldots$
- Es wird kein stabiler Zustand erreicht (Oszillation der Ausgabe).

Definition von Lernaufgaben für ein neuronales Netz

Eine **feste Lernaufgabe** L_{fixed} für ein neuronales Netz mit

- n Eingabeneuronen, d.h. $U_{\text{in}} = \{u_1, \ldots, u_n\}$, and
- m Ausgabeneuronen, d.h. $U_{\text{out}} = \{v_1, \ldots, v_m\}$,

ist eine Menge von **Trainingsbeispielen** $l = (\boldsymbol{\imath}^{(l)}, \boldsymbol{o}^{(l)})$, bestehend aus

- einem **Eingabevektor** $\boldsymbol{\imath}^{(l)} = (\text{ex}_{u_1}^{(l)}, \ldots, \text{ex}_{u_n}^{(l)})$ and
- einem **Ausgabevektor** $\boldsymbol{o}^{(l)} = (o_{v_1}^{(l)}, \ldots, o_{v_m}^{(l)})$.

Eine feste Lernaufgabe gilt als gelöst, wenn das NN für alle Trainingsbeispiele $l \in L_{\text{fixed}}$ aus den externen Eingaben im Eingabevektor $\boldsymbol{\imath}^{(l)}$ eines Trainingsmusters l die Ausgaben berechnet, die im entsprechenden Ausgabevektor $\boldsymbol{o}^{(l)}$ enthalten sind.

Lösen einer festen Lernaufgabe: Fehlerbestimmung

- Bestimme, wie gut ein neuronales Netz eine feste Lernaufgabe löst.
- Berechne Differenzen zwischen gewünschten und berechneten Ausgaben.
- Summiere Differenzen nicht einfach, da sich die Fehler gegenseitig aufheben könnten.
- Quadrieren liefert passende Eigenschaften, um die Anpassungsregeln abzuleiten.

$$e = \sum_{l \in L_{\text{fixed}}} e^{(l)} = \sum_{v \in U_{\text{out}}} e_v = \sum_{l \in L_{\text{fixed}}} \sum_{v \in U_{\text{out}}} e_v^{(l)},$$

$$\text{wobei} \qquad e_v^{(l)} = \left(o_v^{(l)} - \text{out}_v^{(l)}\right)^2$$

Definition von Lernaufgaben für ein neuronales Netz

Eine **freie Lernaufgabe** L_{free} für ein neuronales Netz mit

- n Eingabeneuronen, d.h. $U_{\text{in}} = \{u_1, \ldots, u_n\}$,

ist eine Menge von **Trainingsbeispielen** $l = \left(\boldsymbol{\imath}^{(l)}\right)$, wobei jedes aus

- einem **Eingabevektor** $\boldsymbol{\imath}^{(l)} = \left(\text{ex}_{u_1}^{(l)}, \ldots, \text{ex}_{u_n}^{(l)}\right)$ besteht.

Eigenschaften:

- Es gibt keine gewünschte Ausgabe für die Trainingsbeispiele.
- Ausgaben können von der Trainingsmethode frei gewählt werden.
- Lösungsidee: **Ähnliche Eingaben sollten zu ähnlichen Ausgaben führen.** (Clustering der Eingabevektoren)

Normalisierung der Eingabevektoren

- Berechne Erwartungswert und Standardabweichung jeder Eingabe:

$$\mu_k = \frac{1}{|L|} \sum_{l \in L} \mathrm{ex}_{u_k}^{(l)} \qquad \text{and} \qquad \sigma_k = \sqrt{\frac{1}{|L|} \sum_{l \in L} \left(\mathrm{ex}_{u_k}^{(l)} - \mu_k\right)^2},$$

- Normalisiere die Eingabevektoren auf Erwartungswert 0 und Standardabweichung 1:

$$\mathrm{ex}_{u_k}^{(l)(\mathrm{neu})} = \frac{\mathrm{ex}_{u_k}^{(l)(\mathrm{alt})} - \mu_k}{\sigma_k}$$

- Vermeidet Skalierungsprobleme.

Mehrschichtige Perzeptren

Multilayer Perceptrons (MLPs)

Mehrschichtige Perzeptren

Ein **r-schichtiges Perzeptron** ist ein neuronales Netz mit einem Graph $G = (U, C)$ das die folgenden Bedingungen erfüllt:

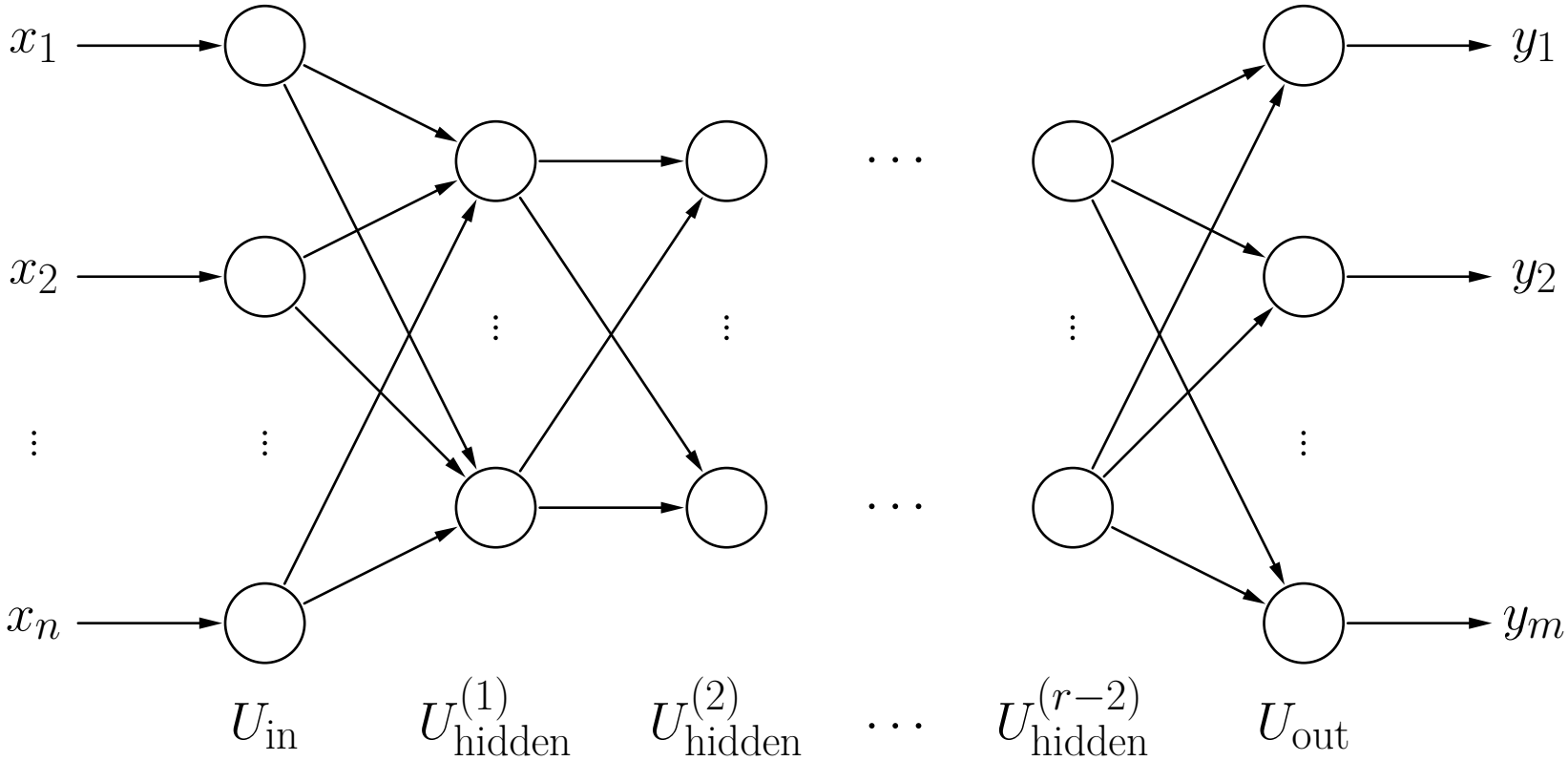
(i) $U_{\text{in}} \cap U_{\text{out}} = \emptyset$,

(ii) $U_{\text{hidden}} = U_{\text{hidden}}^{(1)} \cup \cdots \cup U_{\text{hidden}}^{(r-2)}$,
$\forall 1 \leq i < j \leq r-2: \quad U_{\text{hidden}}^{(i)} \cap U_{\text{hidden}}^{(j)} = \emptyset$,

(iii) $C \subseteq \left(U_{\text{in}} \times U_{\text{hidden}}^{(1)}\right) \cup \left(\cup_{i=1}^{r-3} U_{\text{hidden}}^{(i)} \times U_{\text{hidden}}^{(i+1)}\right) \cup \left(U_{\text{hidden}}^{(r-2)} \times U_{\text{out}}\right)$
oder, falls keine versteckten Neuronen vorhanden ($r = 2, U_{\text{hidden}} = \emptyset$),
$C \subseteq U_{\text{in}} \times U_{\text{out}}$.

- Vorwärtsgerichtetes Netz mit streng geschichteter Struktur.

Allgemeine Struktur eines mehrschichtigen Perzeptrons



Mehrschichtige Perzeptren

- Die Netzwerkeingabe jedes versteckten Neurons und jedes Ausgabeneurons ist die **gewichtete Summe** seiner Eingaben, d.h.

$$\forall u \in U_{\text{hidden}} \cup U_{\text{out}} : \qquad f_{\text{net}}^{(u)}(\boldsymbol{w}_u, \mathbf{in}_u) = \boldsymbol{w}_u \mathbf{in}_u = \sum_{v \in \text{pred}\,(u)} w_{uv} \,\text{out}_v \,.$$

- Die Aktivierungsfunktion jedes versteckten Neurons ist eine sogenannte **Sigmoide**, d.h. eine monoton steigende Funktion

$$f : \mathbb{R} \to [0,1] \quad \text{with} \quad \lim_{x \to -\infty} f(x) = 0 \quad \text{and} \quad \lim_{x \to \infty} f(x) = 1.$$

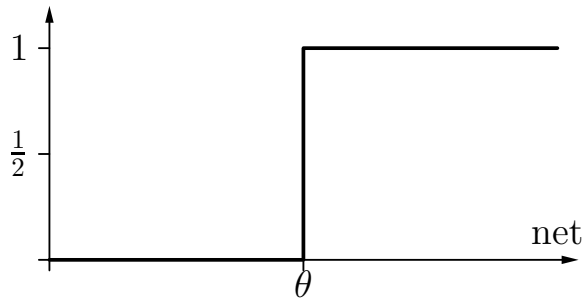
- Die Aktivierungsfunktion jedes Ausgabeneurons ist ebenfalls entweder eine Sigmoide oder eine **lineare Funktion**, d.h.

$$f_{\text{act}}(\text{net}, \theta) = \alpha \,\text{net} - \theta.$$

Sigmoide als Aktivierungsfunktionen

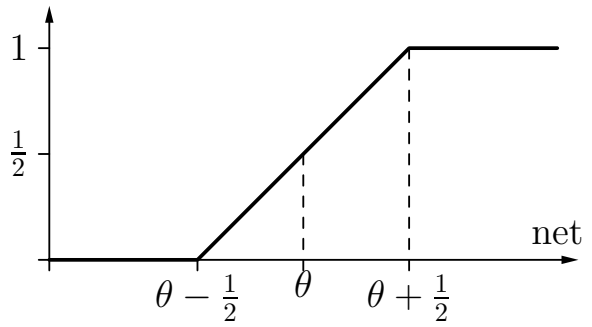
Stufenfunktion:

$$f_{\text{act}}(\text{net}, \theta) = \begin{cases} 1, & \text{falls net} \geq \theta, \\ 0, & \text{sonst.} \end{cases}$$



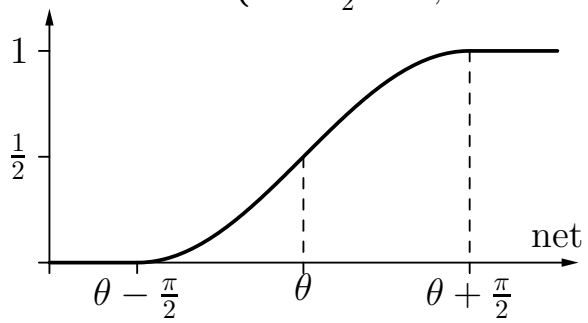
Semilineare Funktion:

$$f_{\text{act}}(\text{net}, \theta) = \begin{cases} 1, & \text{falls net} > \theta + \frac{1}{2}, \\ 0, & \text{falls net} < \theta - \frac{1}{2}, \\ (\text{net} - \theta) + \frac{1}{2}, & \text{sonst.} \end{cases}$$



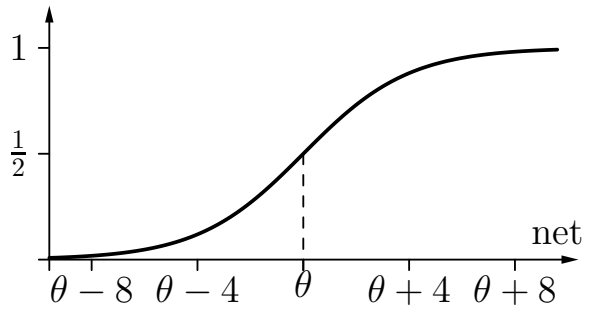
Sinus bis Sättigung:

$$f_{\text{act}}(\text{net}, \theta) = \begin{cases} 1, & \text{falls net} > \theta + \frac{\pi}{2}, \\ 0, & \text{falls net} < \theta - \frac{\pi}{2}, \\ \frac{\sin(\text{net} - \theta) + 1}{2}, & \text{sonst.} \end{cases}$$



Logistische Funktion:

$$f_{\text{act}}(\text{net}, \theta) = \frac{1}{1 + e^{-(\text{net} - \theta)}}$$

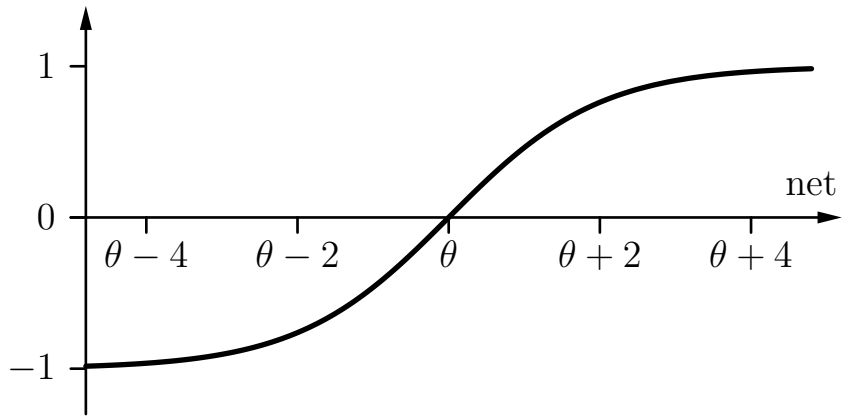


Sigmoide als Aktivierungsfunktionen

- Alle Sigmoiden auf der vorherigen Folie sind **unipolar**,
  d.h. sie reichen von 0 bis 1.
- Manchmal werden **bipolare** sigmoidale Funktionen verwendet,
  wie beispielsweise der *tangens hyperbolicus*.

tangens hyperbolicus:

$$f_{\text{act}}(\text{net}, \theta) = \tanh(\text{net} - \theta)$$
$$= \frac{2}{1 + e^{-2(\text{net} - \theta)}} - 1$$



Mehrschichtige Perzeptren: Gewichtsmatrizen

Seien $U_1 = \{v_1, \ldots, v_m\}$ and $U_2 = \{u_1, \ldots, u_n\}$ die Neuronen zwei aufeinanderfolgender Schichten eines MLP.

Ihre Verbindungsgewichte werden dargestellt als eine $n \times m$-Matrix

$$\mathbf{W} = \begin{pmatrix} w_{u_1 v_1} & w_{u_1 v_2} & \ldots & w_{u_1 v_m} \\ w_{u_2 v_1} & w_{u_2 v_2} & \ldots & w_{u_2 v_m} \\ \vdots & \vdots & & \vdots \\ w_{u_n v_1} & w_{u_n v_2} & \ldots & w_{u_n v_m} \end{pmatrix},$$

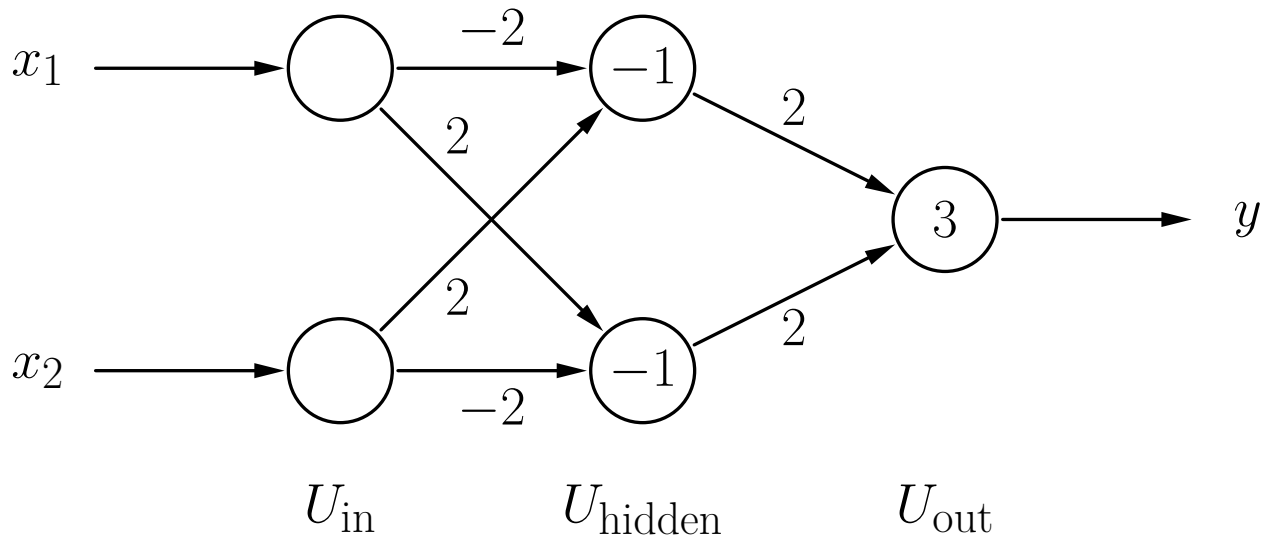
wobei $w_{u_i v_j} = 0$, falls es keine Verbindung von Neuron v_j zu Neuron u_i gibt.

Vorteil: Die Berechnung der Netzwerkeingabe kann geschrieben werden als

$$\mathbf{net}_{U_2} = \mathbf{W} \cdot \mathbf{in}_{U_2} = \mathbf{W} \cdot \mathbf{out}_{U_1}$$

wobei $\mathbf{net}_{U_2} = (\mathrm{net}_{u_1}, \ldots, \mathrm{net}_{u_n})^\top$ und $\mathbf{in}_{U_2} = \mathbf{out}_{U_1} = (\mathrm{out}_{v_1}, \ldots, \mathrm{out}_{v_m})^\top$.

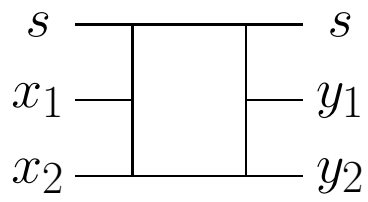
Lösen des Biimplikationsproblems mit einem MLP.

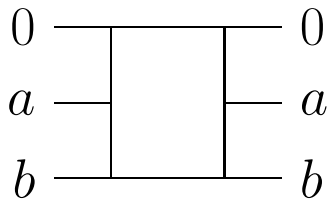


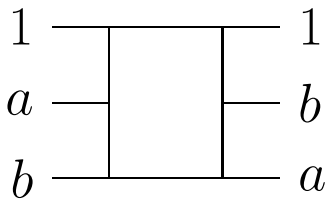
Man beachte die zusätzlichen Eingabeneuronen im Vergleich zur Lösung mit Schwellenwertelementen.

$$\mathbf{W}_1 = \begin{pmatrix} -2 & 2 \\ 2 & -2 \end{pmatrix} \quad \text{und} \quad \mathbf{W}_2 = \begin{pmatrix} 2 & 2 \end{pmatrix}$$

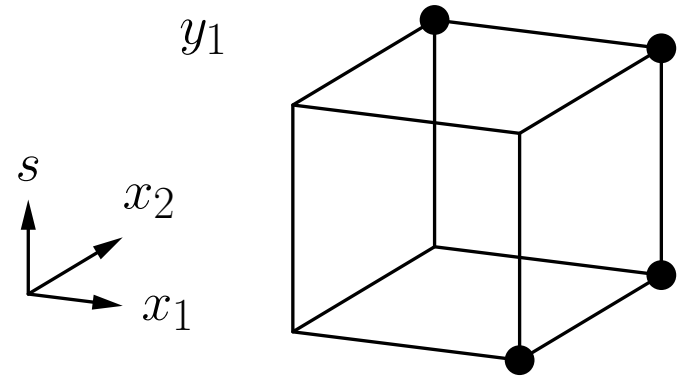
Mehrschichtige Perzeptren: Fredkin-Gatter

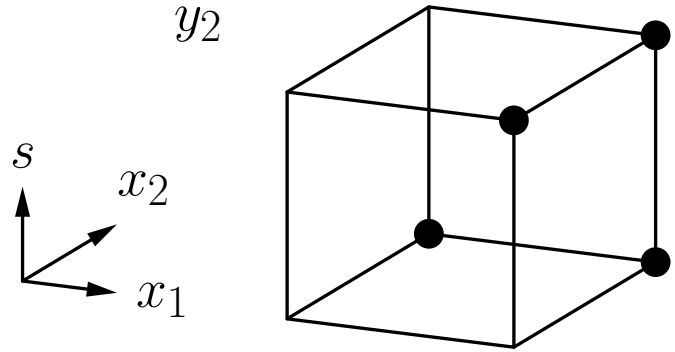




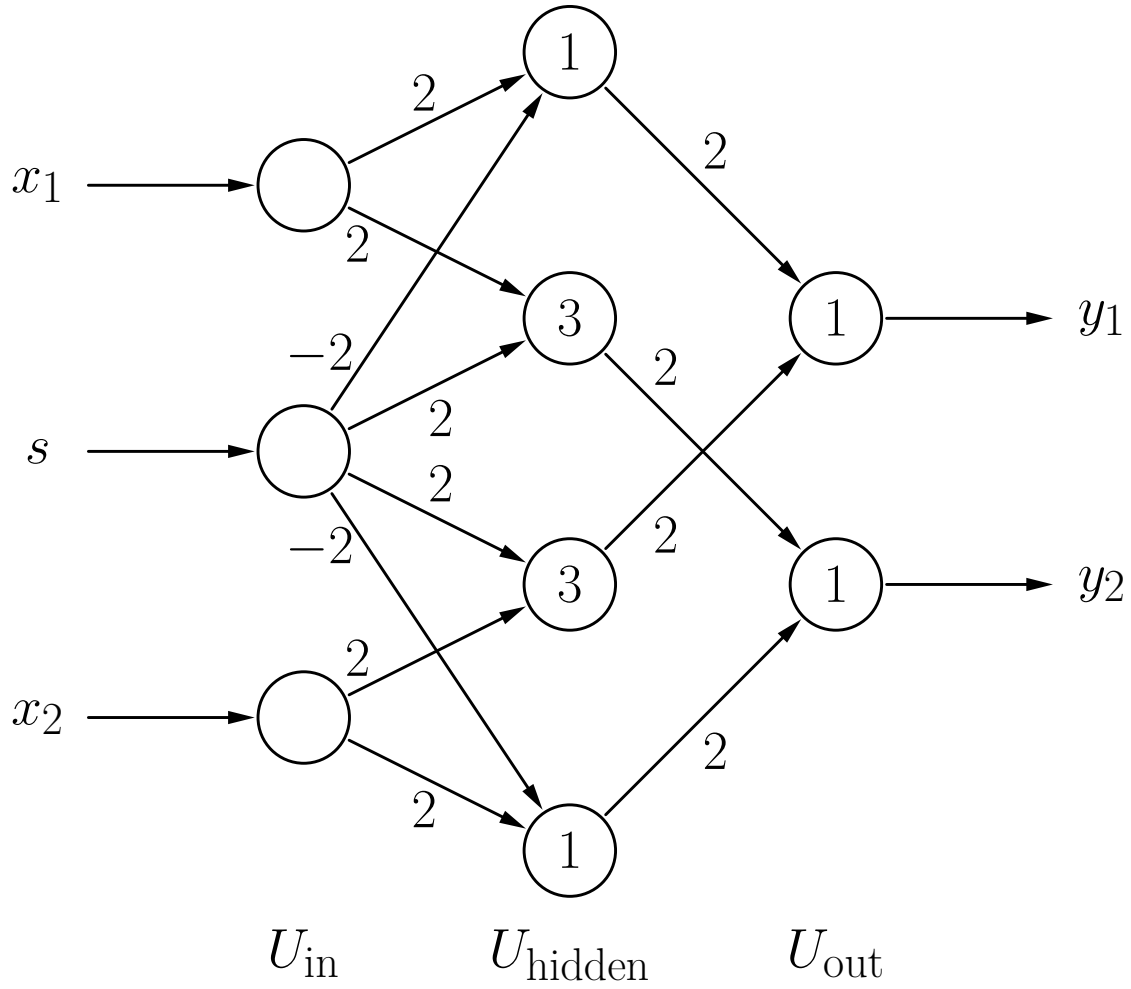


s	0	0	0	0	1	1	1	1
x_1	0	0	1	1	0	0	1	1
x_2	0	1	0	1	0	1	0	1
y_1	0	0	1	1	0	1	0	1
y_2	0	1	0	1	0	0	1	1





Mehrschichtige Perzeptren: Fredkin-Gatter



$$\mathbf{W}_1 = \begin{pmatrix} 2 & -2 & 0 \\ 2 & 2 & 0 \\ 0 & 2 & 2 \\ 0 & -2 & 2 \end{pmatrix}$$

$$\mathbf{W}_2 = \begin{pmatrix} 2 & 0 & 2 & 0 \\ 0 & 2 & 0 & 2 \end{pmatrix}$$

Warum nicht-lineare Aktivierungsfunktionen?

In Matrixschreibweise ergibt sich für zwei aufeinanderfolgende Schichten U_1 und U_2

$$\mathbf{net}_{U_2} = \mathbf{W} \cdot \mathbf{in}_{U_2} = \mathbf{W} \cdot \mathbf{out}_{U_1}.$$

Wenn die Aktivierungsfunktionen linear sind, d.h.

$$f_{\text{act}}(\text{net}, \theta) = \alpha \, \text{net} - \theta.$$

können die Aktivierungen der Neuronen in der Schicht U_2 wie folgt berechnet werden:

$$\mathbf{act}_{U_2} = \mathbf{D}_{\text{act}} \cdot \mathbf{net}_{U_2} - \boldsymbol{\theta},$$

wobei

- $\mathbf{act}_{U_2} = (\text{act}_{u_1}, \ldots, \text{act}_{u_n})^\top$ der Aktivierungsvektor ist,
- $\mathbf{D}_{\text{act}}$ eine $n \times n$ Diagonalmatrix aus den Faktorenα_{u_i}, $i = 1, \ldots, n$, ist und
- $\boldsymbol{\theta} = (\theta_{u_1}, \ldots, \theta_{u_n})^\top$ der Bias-Vektor ist.

Warum nicht-lineare Aktivierungsfunktionen?

Falls die Ausgabefunktion auch linear ist, gilt analog

$$\mathbf{out}_{U_2} = \mathbf{D}_{\text{out}} \cdot \mathbf{act}_{U_2} - \boldsymbol{\xi},$$

wobei

- $\mathbf{out}_{U_2} = (\text{out}_{u_1}, \ldots, \text{out}_{u_n})^\top$ der Ausgabevektor ist,
- $\mathbf{D}_{\text{out}}$ wiederum eine $n \times n$ Diagonalmatrix aus Faktoren ist und
- $\boldsymbol{\xi} = (\xi_{u_1}, \ldots, \xi_{u_n})^\top$ ein Biasvektor ist.

In Kombination ergibt sich

$$\mathbf{out}_{U_2} = \mathbf{D}_{\text{out}} \cdot \big(\mathbf{D}_{\text{act}} \cdot \big(\mathbf{W} \cdot \mathbf{out}_{U_1}\big) - \boldsymbol{\theta}\,\big) - \boldsymbol{\xi}$$

und daher

$$\mathbf{out}_{U_2} = \mathbf{A}_{12} \cdot \mathbf{out}_{U_1} + \boldsymbol{b}_{12}$$

mit einer $n \times m$-Matrix $\mathbf{A}_{12}$ und einem n-dimensionalen Vektor $\boldsymbol{b}_{12}$.

Warum nicht-lineare Aktivierungsfunktionen?

Daher ergibt sich

$$\mathbf{out}_{U_2} = \mathbf{A}_{12} \cdot \mathbf{out}_{U_1} + \boldsymbol{b}_{12}$$

und

$$\mathbf{out}_{U_3} = \mathbf{A}_{23} \cdot \mathbf{out}_{U_2} + \boldsymbol{b}_{23}$$

für die Berechnungen zwei aufeinanderfolgender Schichten U_2 und U_3.

Diese beiden Berechnungen können kombiniert werden zu

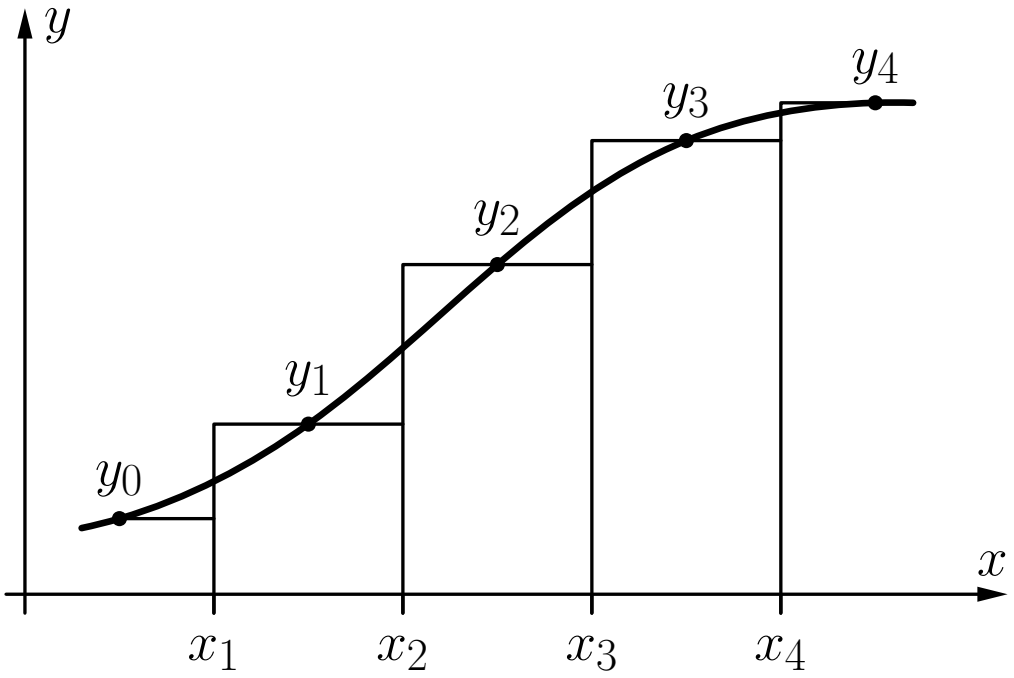
$$\mathbf{out}_{U_3} = \mathbf{A}_{13} \cdot \mathbf{out}_{U_1} + \boldsymbol{b}_{13},$$

wobei $\mathbf{A}_{13} = \mathbf{A}_{23} \cdot \mathbf{A}_{12}$ und $\boldsymbol{b}_{13} = \mathbf{A}_{23} \cdot \boldsymbol{b}_{12} + \boldsymbol{b}_{23}$.

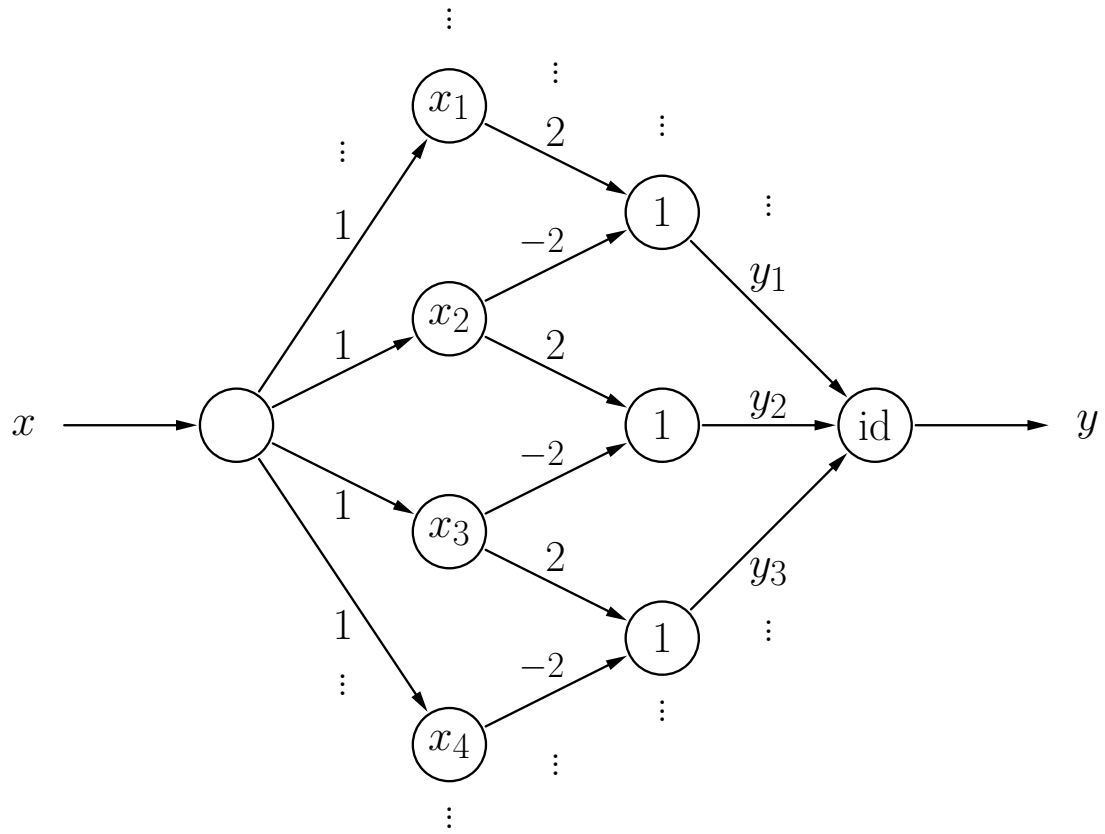
Ergebnis: Mit linearen Aktivierungs- und Ausgabefunktionen können beliebige mehrschichtige Perzeptren auf zweischichtige Perzeptren reduziert werden.

Idee der Funktionsapproximation

- Nähere eine gegebene Funktion durch eine Stufenfunktion an.
- Konstruiere ein neuronales Netz, das die Stufenfunktion berechnet.



Mehrschichtige Perzeptren: Funktionsapproximation

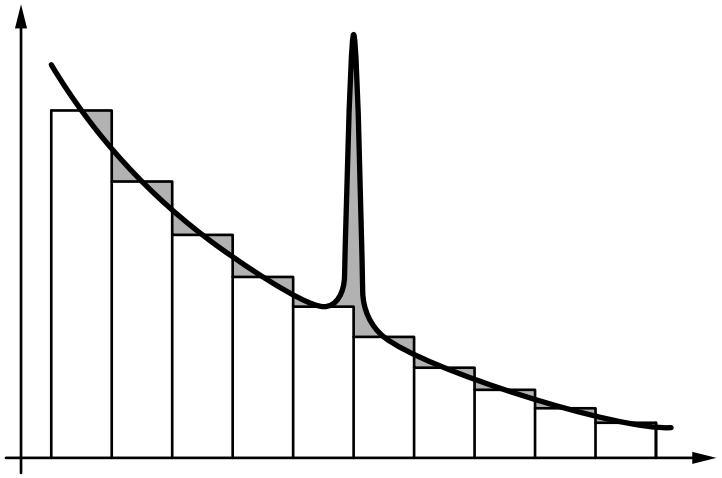


Ein neuronales Netz, das die Treppenfunktion von der vorherigen Folie berechnet. Es ist immer nur eine Stufe passend zum Eingabewert aktiv und die Stufenhöhe wird ausgegeben.

Mehrschichtige Perzeptren: Funktionsapproximation

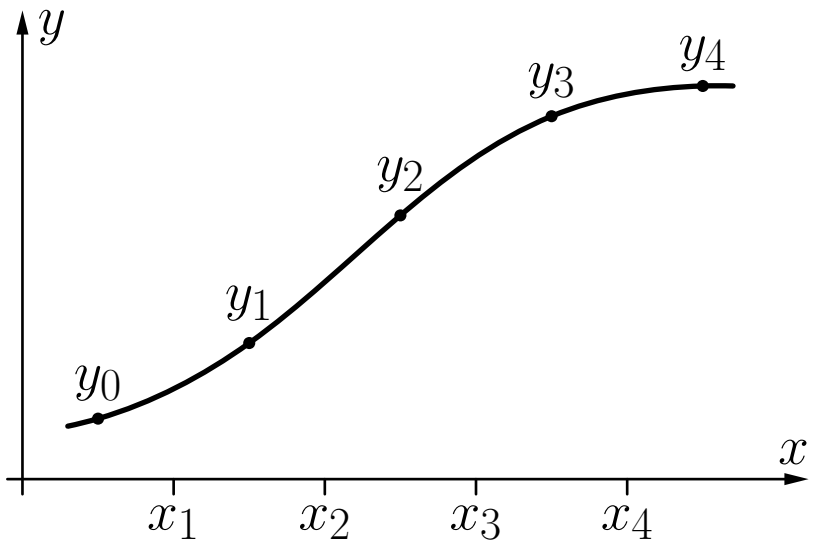
Theorem: Jede Riemann-integrierbare Funktion kann mit beliebiger Genauigkeit durch ein vierschichtiges MLP berechnet werden.

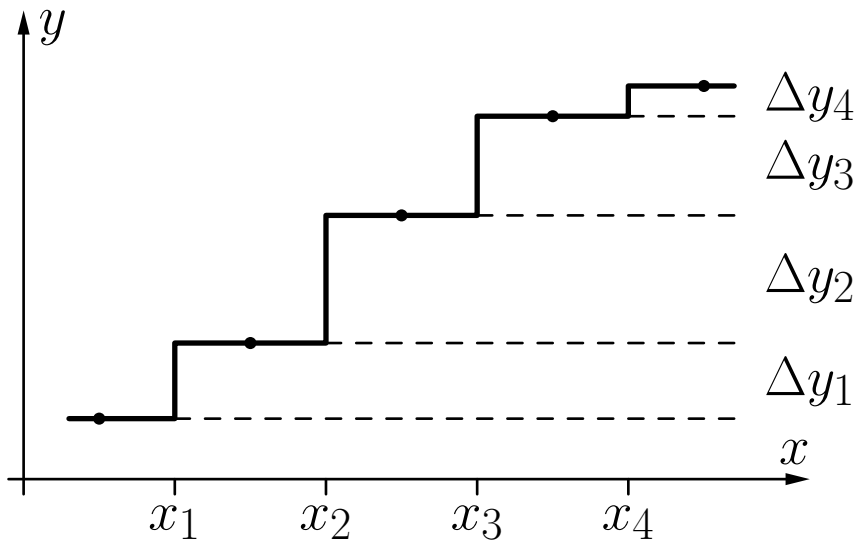
- Aber: Fehler wird bestimmt als die **Fläche** zwischen Funktionen.

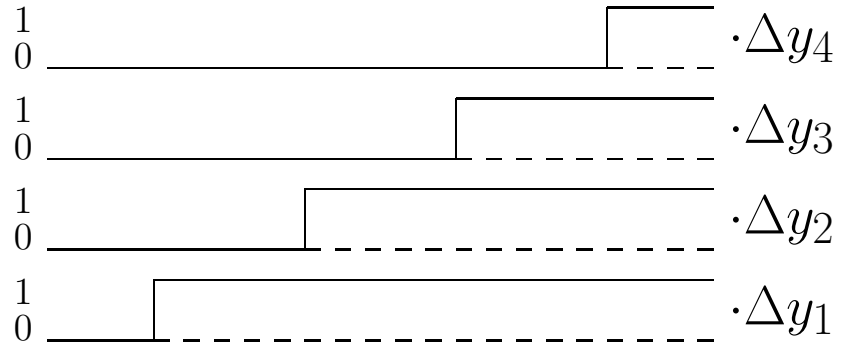


- Weitere mathematische Untersuchungen zeigen, dass sogar gilt: Mit einem dreischichtigen Perzeptron kann jede stetige Funktion mit beliebiger Genauigkeit angenähert werden (Fehlerbestimmung: maximale Differenz der Funktionswerte).

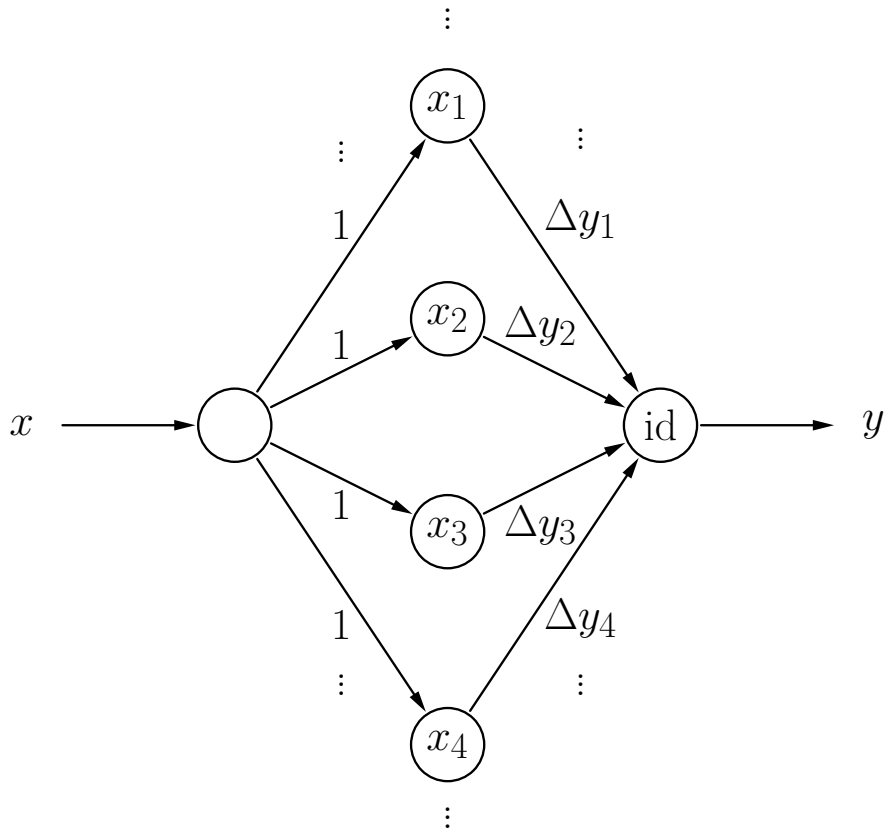
Mehrschichtige Perzeptren: Funktionsapproximation





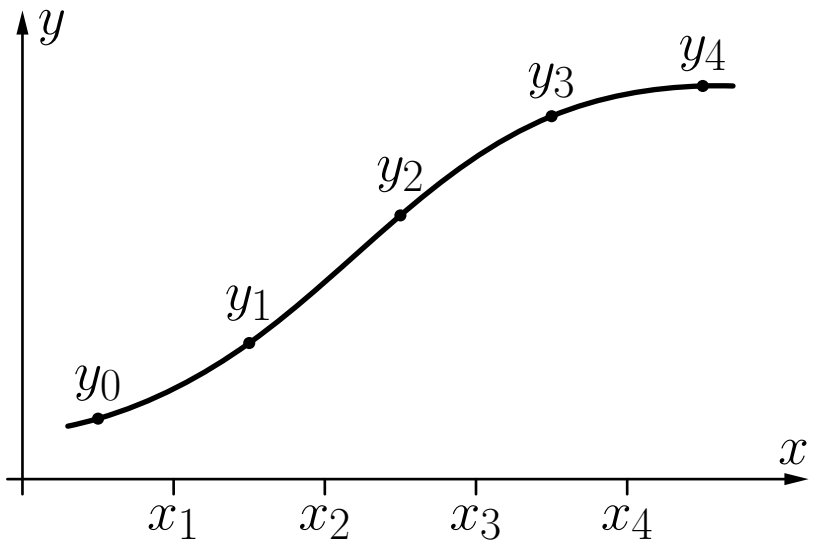


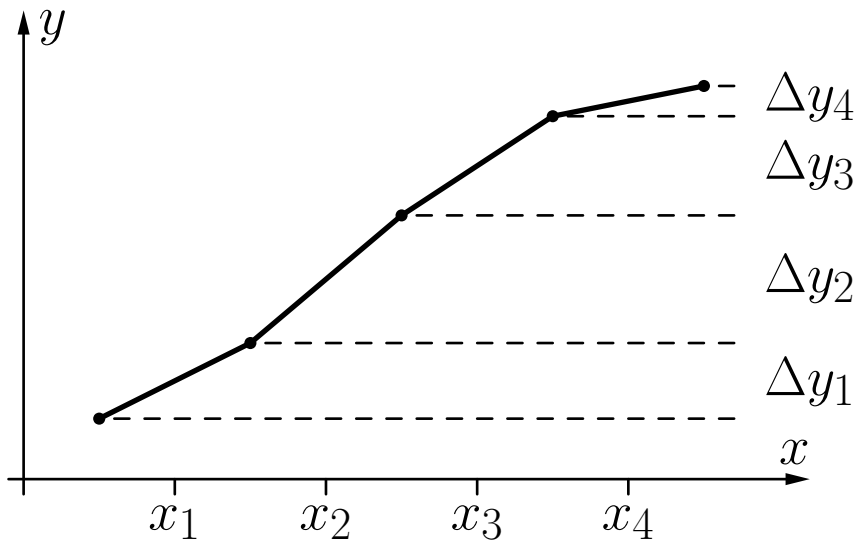
Mehrschichtige Perzeptren: Funktionsapproximation

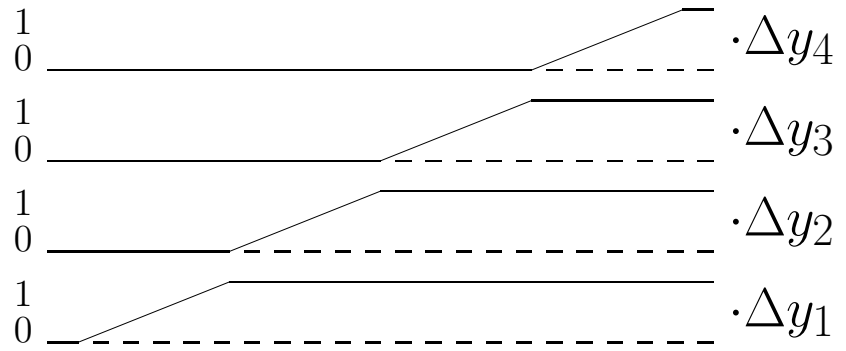


Ein neuronales Netz, das die Treppenfunktion von der vorherigen Folie als gewichtete Summe von Sprungfunktionen berechnet.

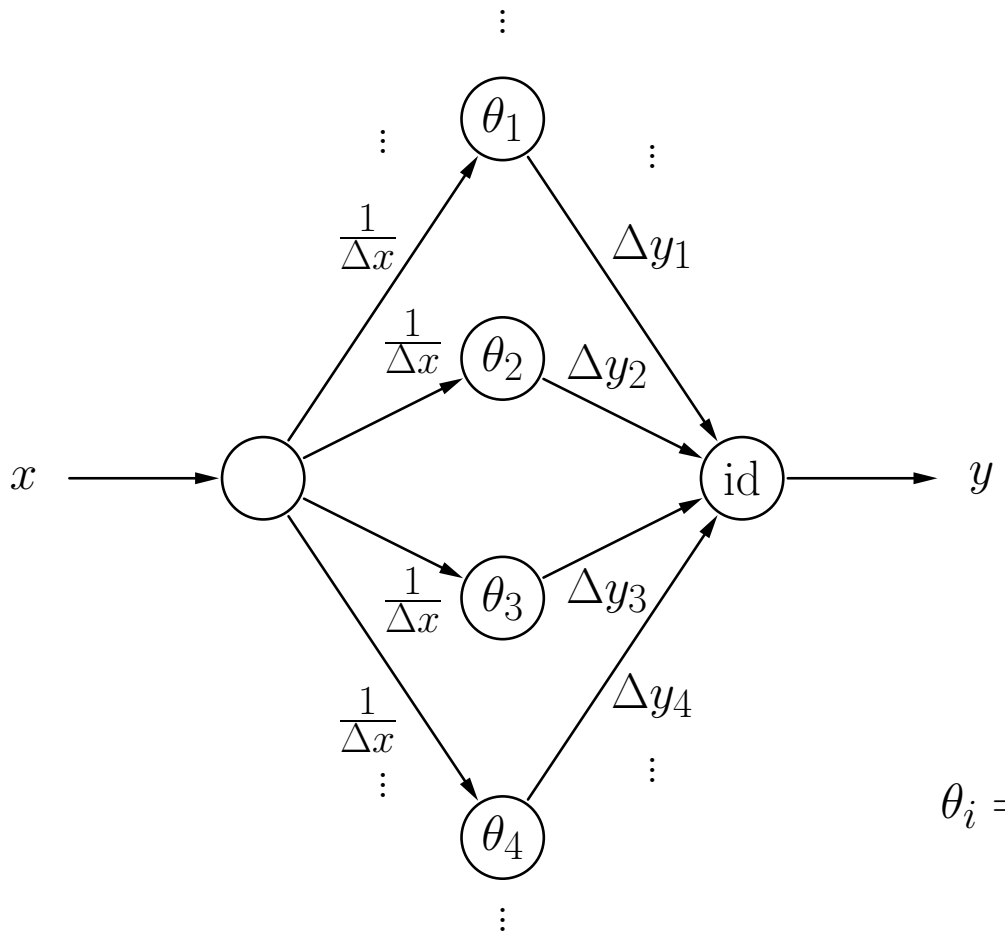
Mehrschichtige Perzeptren: Funktionsapproximation







Mehrschichtige Perzeptren: Funktionsapproximation



$$\theta_i = \frac{x_i}{\Delta x}$$

Ein neuronales Netz, das die stückweise lineare Funktion von der vorherigen Folie durch eine gewichtete Summe von semi-linearen Funktionen berechnet, wobei $\Delta x = x_{i+1} - x_i$.

Mathematischer Hintergrund: Regression

Das Trainieren von NN ist stark verwandt mit Regression

Geg:
- Ein Datensatz $((x_1, y_1), \ldots, (x_n, y_n))$ aus n Daten-Tupeln und
- Hypothese über den funktionellen Zusammenhang, also z.B. $y = g(x) = a + bx$.

Idee: Minimiere die Summe der quadrierten Fehler, d.h.

$$F(a, b) = \sum_{i=1}^{n} (g(x_i) - y_i)^2 = \sum_{i=1}^{n} (a + bx_i - y_i)^2.$$

Notwendige Bedingungen für ein Minimum:

$$\frac{\partial F}{\partial a} = \sum_{i=1}^{n} 2(a + bx_i - y_i) = 0 \quad \text{und}$$

$$\frac{\partial F}{\partial b} = \sum_{i=1}^{n} 2(a + bx_i - y_i)x_i = 0$$

Mathematischer Hintergrund: Lineare Regression

Resultat der notwendigen Bedingungen: System sogenannter **Normalgleichungen**, d.h.

$$na + \left(\sum_{i=1}^{n} x_i\right) b = \sum_{i=1}^{n} y_i,$$

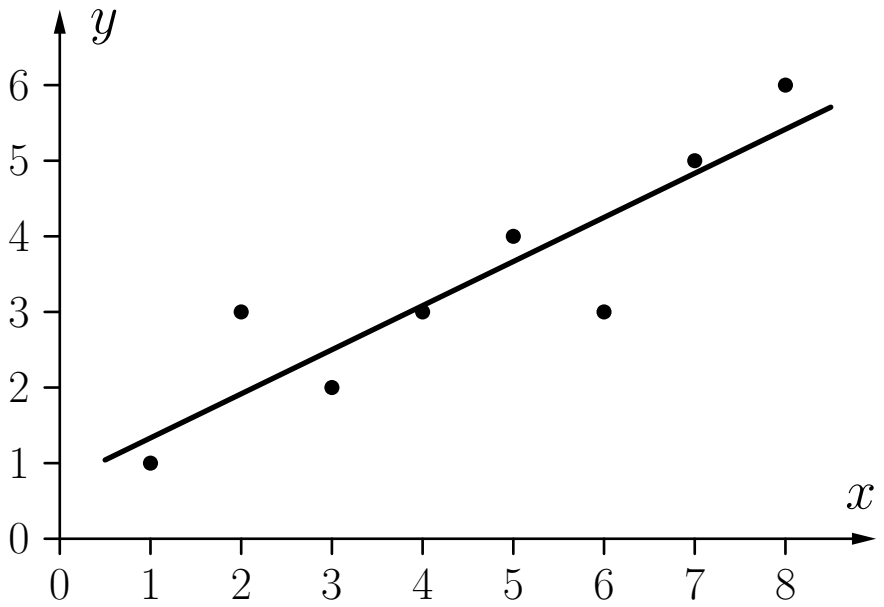
$$\left(\sum_{i=1}^{n} x_i\right) a + \left(\sum_{i=1}^{n} x_i^2\right) b = \sum_{i=1}^{n} x_i y_i.$$

- Zwei lineare Gleichungen für zwei Unbekannte a und b.
- System kann mit Standardmethoden der linearen Algebra gelöst werden.
- Die Lösung ist eindeutig, falls nicht alle x-Werte identisch sind.
- Die errechnete Gerade nennt man **Regressionsgerade**.

Lineare Regression: Beispiel

x	1	2	3	4	5	6	7	8
y	1	3	2	3	4	3	5	6

$$y = \frac{3}{4} + \frac{7}{12}x.$$



Generalisierung auf Polynome

$$y = p(x) = a_0 + a_1 x + \ldots + a_m x^m$$

Idee: Minimiere die Summe der quadrierten Fehler, d.h.

$$F(a_0, a_1, \ldots, a_m) \;=\; \sum_{i=1}^{n} (p(x_i) - y_i)^2 \;=\; \sum_{i=1}^{n} (a_0 + a_1 x_i + \ldots + a_m x_i^m - y_i)^2$$

Notwendige Bedingungen für ein Minimum: Alle partiellen Ableitungen verschwinden, d.h.

$$\frac{\partial F}{\partial a_0} = 0, \quad \frac{\partial F}{\partial a_1} = 0, \quad \ldots \quad , \frac{\partial F}{\partial a_m} = 0.$$

System von Normalgleichungen für Polynome

$$
\begin{aligned}
n a_0 + \left(\sum_{i=1}^{n} x_i\right) a_1 + \ldots + \left(\sum_{i=1}^{n} x_i^m\right) a_m &= \sum_{i=1}^{n} y_i \\
\left(\sum_{i=1}^{n} x_i\right) a_0 + \left(\sum_{i=1}^{n} x_i^2\right) a_1 + \ldots + \left(\sum_{i=1}^{n} x_i^{m+1}\right) a_m &= \sum_{i=1}^{n} x_i y_i \\
\vdots \qquad\qquad \vdots \qquad\qquad & \qquad\qquad \vdots \\
\left(\sum_{i=1}^{n} x_i^m\right) a_0 + \left(\sum_{i=1}^{n} x_i^{m+1}\right) a_1 + \ldots + \left(\sum_{i=1}^{n} x_i^{2m}\right) a_m &= \sum_{i=1}^{n} x_i^m y_i,
\end{aligned}
$$

- $m + 1$ lineare Gleichungen für $m + 1$ Unbekannte $a_0, \ldots, a_m$.
- System kann mit Standardmethoden der linearen Algebra gelöst werden.
- Die Lösung ist eindeutig, falls nicht alle x-Werte identisch sind.

Generalisierung auf mehr als ein Argument

$$z = f(x, y) = a + bx + cy$$

Idee: Minimiere die Summe der quadrierten Fehler, d.h.

$$F(a, b, c) \;=\; \sum_{i=1}^{n} (f(x_i, y_i) - z_i)^2 \;=\; \sum_{i=1}^{n} (a + bx_i + cy_i - z_i)^2$$

Notwendige Bedingungen für ein Minimum: Alle partiellen Ableitungen verschwinden, d.h.

$$\begin{aligned}
\frac{\partial F}{\partial a} &= \sum_{i=1}^{n} 2(a + bx_i + cy_i - z_i) &&= 0, \\
\frac{\partial F}{\partial b} &= \sum_{i=1}^{n} 2(a + bx_i + cy_i - z_i)x_i &&= 0, \\
\frac{\partial F}{\partial c} &= \sum_{i=1}^{n} 2(a + bx_i + cy_i - z_i)y_i &&= 0.
\end{aligned}$$

System von Normalgleichungen für mehrere Argumente

$$
\begin{aligned}
n a + \left(\sum_{i=1}^{n} x_i\right) b + \left(\sum_{i=1}^{n} y_i\right) c &= \sum_{i=1}^{n} z_i \\
\left(\sum_{i=1}^{n} x_i\right) a + \left(\sum_{i=1}^{n} x_i^2\right) b + \left(\sum_{i=1}^{n} x_i y_i\right) c &= \sum_{i=1}^{n} z_i x_i \\
\left(\sum_{i=1}^{n} y_i\right) a + \left(\sum_{i=1}^{n} x_i y_i\right) b + \left(\sum_{i=1}^{n} y_i^2\right) c &= \sum_{i=1}^{n} z_i y_i
\end{aligned}
$$

- 3 lineare Gleichungen für 3 Unbekannte a, b und c.
- System kann mit Standardmethoden der linearen Algebra gelöst werden.
- Die Lösung ist eindeutig, falls nicht alle x-Werte oder alle y-Werte identisch sind.

Allgemeiner multilinearer Fall:

$$y = f(x_1, \ldots, x_m) = a_0 + \sum_{k=1}^{m} a_k x_k$$

Idee: Minimiere die Summe der quadrierten Fehler, d.h.

$$F(\boldsymbol{a}) = (\mathbf{X}\boldsymbol{a} - \boldsymbol{y})^\top (\mathbf{X}\boldsymbol{a} - \boldsymbol{y}),$$

wobei

$$\mathbf{X} = \begin{pmatrix} 1 & x_{11} & \ldots & x_{m1} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{1n} & \ldots & x_{mn} \end{pmatrix}, \qquad \boldsymbol{y} = \begin{pmatrix} y_1 \\ \vdots \\ y_n \end{pmatrix}, \qquad \text{und} \qquad \boldsymbol{a} = \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_m \end{pmatrix}$$

Notwendige Bedingungen für ein Minimum:

$$\nabla_{\boldsymbol{a}} F(\boldsymbol{a}) = \nabla_{\boldsymbol{a}} (\mathbf{X}\boldsymbol{a} - \boldsymbol{y})^\top (\mathbf{X}\boldsymbol{a} - \boldsymbol{y}) = \mathbf{0}$$

- $\nabla_{\boldsymbol{a}}\ F(\boldsymbol{a})$ kann einfach berechnet werden mit der Überlegung, dass der Nabla-Operator

$$\nabla_{\boldsymbol{a}} = \left(\frac{\partial}{\partial a_0}, \ldots, \frac{\partial}{\partial a_m}\right)$$

  sich formell wie ein Vektor verhält, der mit der Summe der quadrierten Fehler "multipliziert" wird.

- Alternativ kann man die Differentiation komponentenweise beschreiben.

Mit der vorherigen Methode bekommen wir für die Ableitung:

$$\begin{aligned}
&\nabla_{\boldsymbol{a}} \left(\mathbf{X}\boldsymbol{a} - \boldsymbol{y}\right)^\top (\mathbf{X}\boldsymbol{a} - \boldsymbol{y}) \\
&= \left(\nabla_{\boldsymbol{a}} \left(\mathbf{X}\boldsymbol{a} - \boldsymbol{y}\right)\right)^\top (\mathbf{X}\boldsymbol{a} - \boldsymbol{y}) + \left(\left(\mathbf{X}\boldsymbol{a} - \boldsymbol{y}\right)^\top \left(\nabla_{\boldsymbol{a}} \left(\mathbf{X}\boldsymbol{a} - \boldsymbol{y}\right)\right)\right)^\top \\
&= \left(\nabla_{\boldsymbol{a}} \left(\mathbf{X}\boldsymbol{a} - \boldsymbol{y}\right)\right)^\top (\mathbf{X}\boldsymbol{a} - \boldsymbol{y}) + \left(\nabla_{\boldsymbol{a}} \left(\mathbf{X}\boldsymbol{a} - \boldsymbol{y}\right)\right)^\top (\mathbf{X}\boldsymbol{a} - \boldsymbol{y}) \\
&= 2\mathbf{X}^\top (\mathbf{X}\boldsymbol{a} - \boldsymbol{y}) \\
&= 2\mathbf{X}^\top \mathbf{X}\boldsymbol{a} - 2\mathbf{X}^\top \boldsymbol{y} \quad = \quad \mathbf{0}
\end{aligned}$$

Einige Regeln für Vektor-/Matrixberechnung und Ableitungen:

$$
\begin{aligned}
(\mathbf{A}+\mathbf{B})^\top &= \mathbf{A}^\top + \mathbf{B}^\top \\
(\mathbf{AB})^\top &= \mathbf{B}^\top \mathbf{A}^\top \\
\nabla_{\boldsymbol{z}} \mathbf{A}\boldsymbol{z} &= \mathbf{A}
\end{aligned}
\qquad
\begin{aligned}
\nabla_{\boldsymbol{z}} f(\boldsymbol{z})\mathbf{A} &= (\nabla_{\boldsymbol{z}} f(\boldsymbol{z}))\mathbf{A} \\
\nabla_{\boldsymbol{z}} (f(\boldsymbol{z}))^\top &= (\nabla_{\boldsymbol{z}} f(\boldsymbol{z}))^\top \\
\nabla_{\boldsymbol{z}} f(\boldsymbol{z})g(\boldsymbol{z}) &= (\nabla_{\boldsymbol{z}} f(\boldsymbol{z}))g(\boldsymbol{z}) + f(\boldsymbol{z})(\nabla_{\boldsymbol{z}} g(\boldsymbol{z}))^\top
\end{aligned}
$$

Ableitung der zu minimierenden Funktion:

$$
\begin{aligned}
\nabla_{\boldsymbol{a}} F(\boldsymbol{a}) &= \nabla_{\boldsymbol{a}} (\mathbf{X}\boldsymbol{a} - \boldsymbol{y})^\top (\mathbf{X}\boldsymbol{a} - \boldsymbol{y}) \\
&= \nabla_{\boldsymbol{a}} ((\mathbf{X}\boldsymbol{a})^\top - \boldsymbol{y}^\top)(\mathbf{X}\boldsymbol{a} - \boldsymbol{y}) \\
&= \nabla_{\boldsymbol{a}} ((\mathbf{X}\boldsymbol{a})^\top \mathbf{X}\boldsymbol{a} - (\mathbf{X}\boldsymbol{a})^\top \boldsymbol{y} - \boldsymbol{y}^\top \mathbf{X}\boldsymbol{a} + \boldsymbol{y}^\top \boldsymbol{y}) \\
&= \nabla_{\boldsymbol{a}} (\mathbf{X}\boldsymbol{a})^\top \mathbf{X}\boldsymbol{a} - \nabla_{\boldsymbol{a}} (\mathbf{X}\boldsymbol{a})^\top \boldsymbol{y} - \nabla_{\boldsymbol{a}} \boldsymbol{y}^\top \mathbf{X}\boldsymbol{a} + \nabla_{\boldsymbol{a}} \boldsymbol{y}^\top \boldsymbol{y} \\
&= (\nabla_{\boldsymbol{a}} (\mathbf{X}\boldsymbol{a})^\top)\mathbf{X}\boldsymbol{a} + ((\mathbf{X}\boldsymbol{a})^\top (\nabla_{\boldsymbol{a}} \mathbf{X}\boldsymbol{a}))^\top - 2\nabla_{\boldsymbol{a}} (\mathbf{X}\boldsymbol{a})^\top \boldsymbol{y} \\
&= ((\nabla_{\boldsymbol{a}} \mathbf{X}\boldsymbol{a})^\top)\mathbf{X}\boldsymbol{a} + (\nabla_{\boldsymbol{a}} \mathbf{X}\boldsymbol{a})^\top \mathbf{X}\boldsymbol{a} - 2(\nabla_{\boldsymbol{a}} (\mathbf{X}\boldsymbol{a})^\top)\boldsymbol{y} \\
&= 2(\nabla_{\boldsymbol{a}} \mathbf{X}\boldsymbol{a})^\top \mathbf{X}\boldsymbol{a} - 2(\nabla_{\boldsymbol{a}} \mathbf{X}\boldsymbol{a})^\top \boldsymbol{y} \\
&= 2\mathbf{X}^\top \mathbf{X}\boldsymbol{a} - 2\mathbf{X}^\top \boldsymbol{y}
\end{aligned}
$$

Multilineare Regression

Notwendige Bedingungen für ein Minimum also:

$$\begin{aligned} \nabla_{\boldsymbol{a}} F(\boldsymbol{a}) &= \nabla_{\boldsymbol{a}} (\mathbf{X}\boldsymbol{a} - \boldsymbol{y})^\top (\mathbf{X}\boldsymbol{a} - \boldsymbol{y}) \\ &= 2\mathbf{X}^\top \mathbf{X}\boldsymbol{a} - 2\mathbf{X}^\top \boldsymbol{y} \stackrel{!}{=} \mathbf{0} \end{aligned}$$

Als Ergebnis bekommen wir das System von **Normalgleichungen**:

$$\mathbf{X}^\top \mathbf{X}\boldsymbol{a} = \mathbf{X}^\top \boldsymbol{y}$$

Dieses System hat eine Lösung, falls $\mathbf{X}^\top \mathbf{X}$ nicht singulär ist. Dann ergibt sich

$$\boldsymbol{a} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \boldsymbol{y}.$$

$(\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top$ heißt die (Moore-Penrose-)**Pseudoinverse** der Matrix $\mathbf{X}$.

Mit der Matrix-Vektor-Repräsentation des Regressionsproblems ist die Erweiterung auf **Multipolynomiale Regression** naheliegend:
Addiere die gewünschten Produkte zur Matrix $\mathbf{X}$.

Generalisierung auf nicht-polynomiale Funktionen

$$\text{Einfaches Beispiel:} \qquad y = ax^b$$

Idee: Finde Transformation zum linearen/polynomiellen Fall.

$$\text{Transformation z.B.:} \qquad \ln y = \ln a + b \cdot \ln x.$$

Spezialfall: **logistische Funktion**

$$y = \frac{Y}{1 + e^{a+bx}} \qquad \Leftrightarrow \qquad \frac{1}{y} = \frac{1 + e^{a+bx}}{Y} \qquad \Leftrightarrow \qquad \frac{Y - y}{y} = e^{a+bx}.$$

Ergebnis: Wende sogenannte **Logit-Transformation** an:

$$\ln\left(\frac{Y - y}{y}\right) = a + bx.$$

Logistische Regression: Beispiel

x	1	2	3	4	5
y	0.4	1.0	3.0	5.0	5.6

Transformiere die Daten mit

$$z = \ln\left(\frac{Y - y}{y}\right), \qquad Y = 6.$$

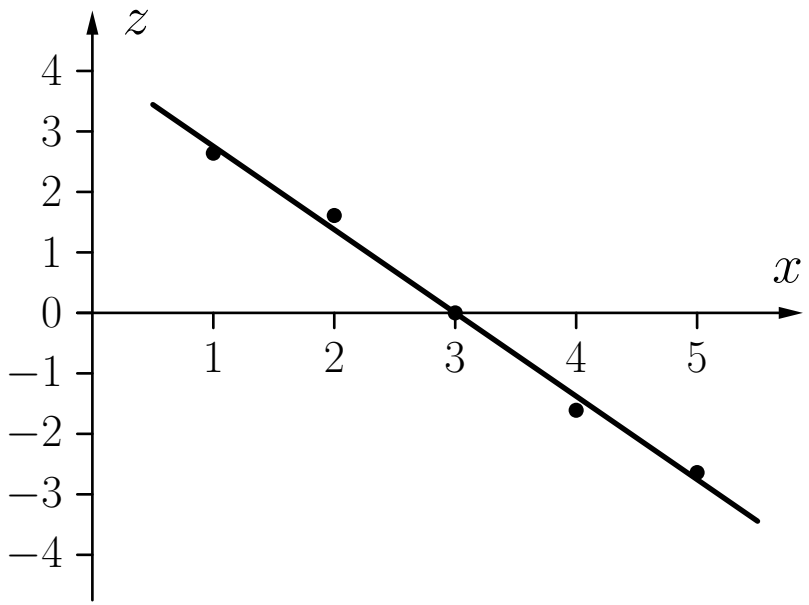
Die transformierten Datenpunkte sind

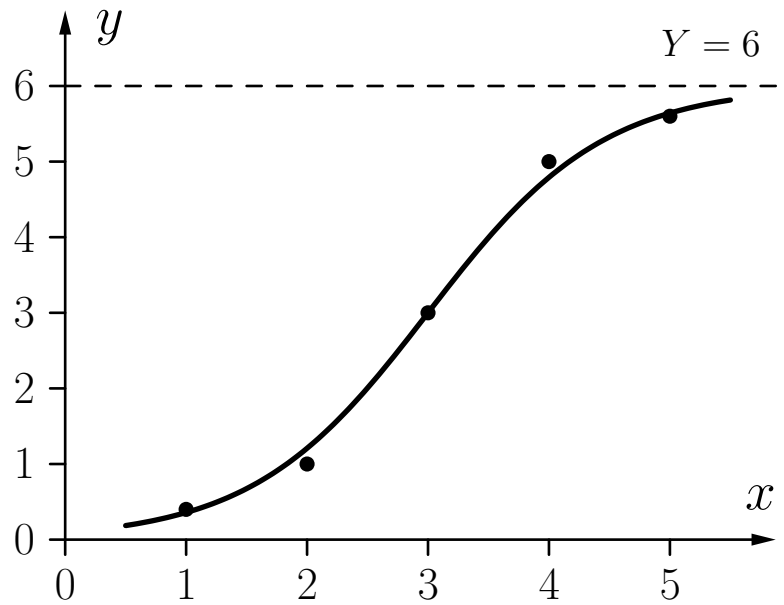
x	1	2	3	4	5
z	2.64	1.61	0.00	-1.61	-2.64

Die sich ergebende Regressionsgerade ist

$$z \approx -1.3775x + 4.133.$$

Logistische Regression: Beispiel





Die logistische Regressionsfunktion kann von einem einzelnen Neuron mit

- Netzeingabefunktion $f_{\text{net}}(x) \equiv wx$ mit $w \approx -1.3775$,
- Aktivierungsfunktion $f_{\text{act}}(\text{net}, \theta) \equiv (1 + e^{-(\text{net} - \theta)})^{-1}$ mit $\theta \approx 4.133$ und
- Ausgabefunktion $f_{\text{out}}(\text{act}) \equiv 6\,\text{act}$

berechnet werden.

Training von MLPs

Training von MLPs: Gradientenabstieg

- Problem der logistischen Regression: Funktioniert nur für zweischichtige Perzeptren.
- Allgemeinerer Ansatz: **Gradientenabstieg**.
- Notwendige Bedingung: **differenzierbare Aktivierungs- und Ausgabefunktionen**.

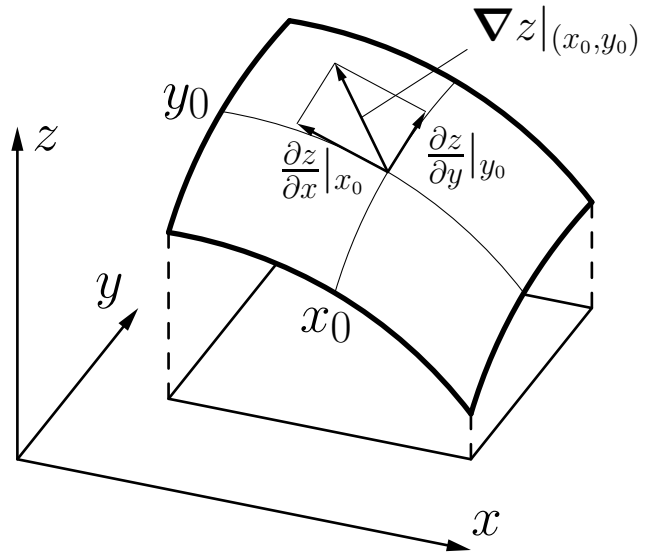


Illustration des Gradienten einer reellwertigen Funktion $z = f(x, y)$ am Punkt (x_0, y_0). Dabei ist $\nabla z|_{(x_0,y_0)} = \left(\frac{\partial z}{\partial x}|x_0, \frac{\partial z}{\partial y}|y_0\right)$.

Gradientenabstieg: Formaler Ansatz

Grundidee: Erreiche das Minimum der Fehlerfunktion in kleinen Schritten.

Fehlerfunktion:

$$e = \sum_{l \in L_{\text{fixed}}} e^{(l)} = \sum_{v \in U_{\text{out}}} e_v = \sum_{l \in L_{\text{fixed}}} \sum_{v \in U_{\text{out}}} e_v^{(l)},$$

Erhalte den Gradienten zur Schrittrichtungsbestimmung:

$$\nabla_{\boldsymbol{w}_u} e = \frac{\partial e}{\partial \boldsymbol{w}_u} = \left(-\frac{\partial e}{\partial \theta_u}, \frac{\partial e}{\partial w_{up_1}}, \ldots, \frac{\partial e}{\partial w_{up_n}} \right).$$

Nutze die Summe über die Trainingsmuster aus:

$$\nabla_{\boldsymbol{w}_u} e = \frac{\partial e}{\partial \boldsymbol{w}_u} = \frac{\partial}{\partial \boldsymbol{w}_u} \sum_{l \in L_{\text{fixed}}} e^{(l)} = \sum_{l \in L_{\text{fixed}}} \frac{\partial e^{(l)}}{\partial \boldsymbol{w}_u}.$$

Gradientenabstieg: Formaler Ansatz

Einzelmusterfehler hängt nur von Gewichten durch die Netzeingabe ab:

$$\nabla_{\boldsymbol{w}_u} e^{(l)} = \frac{\partial e^{(l)}}{\partial \boldsymbol{w}_u} = \frac{\partial e^{(l)}}{\partial \operatorname{net}_u^{(l)}} \frac{\partial \operatorname{net}_u^{(l)}}{\partial \boldsymbol{w}_u}.$$

Da $\operatorname{net}_u^{(l)} = \boldsymbol{w}_u \mathbf{in}_u^{(l)}$, bekommen wir für den zweiten Faktor

$$\frac{\partial \operatorname{net}_u^{(l)}}{\partial \boldsymbol{w}_u} = \mathbf{in}_u^{(l)}.$$

Für den ersten Faktor betrachten wir den Fehler $e^{(l)}$ für das Trainingsmuster $l = \left(\boldsymbol{\imath}^{(l)}, \boldsymbol{o}^{(l)}\right)$:

$$e^{(l)} = \sum_{v \in U_{\text{out}}} e_u^{(l)} = \sum_{v \in U_{\text{out}}} \left(o_v^{(l)} - \operatorname{out}_v^{(l)}\right)^2,$$

d.h. die Summe der Fehler über alle Ausgabeneuronen.

Daher haben wir

$$\frac{\partial e^{(l)}}{\partial \operatorname{net}_u^{(l)}} = \frac{\partial \sum_{v \in U_{\text{out}}} \left(o_v^{(l)} - \operatorname{out}_v^{(l)}\right)^2}{\partial \operatorname{net}_u^{(l)}} = \sum_{v \in U_{\text{out}}} \frac{\partial \left(o_v^{(l)} - \operatorname{out}_v^{(l)}\right)^2}{\partial \operatorname{net}_u^{(l)}}.$$

Da nur die eigentliche Ausgabe $\operatorname{out}_v^{(l)}$ eines Ausgabeneurons v von der Netzeingabe $\operatorname{net}_u^{(l)}$ des Neurons u abhängt, das wir betrachten, ist

$$\frac{\partial e^{(l)}}{\partial \operatorname{net}_u^{(l)}} = -2 \underbrace{\sum_{v \in U_{\text{out}}} \left(o_v^{(l)} - \operatorname{out}_v^{(l)}\right) \frac{\partial \operatorname{out}_v^{(l)}}{\partial \operatorname{net}_u^{(l)}}}_{\delta_u^{(l)}},$$

womit zugleich die Abkürzung $\delta_u^{(l)}$ für die im Folgenden wichtige Summe eingeführt wird.

Gradientenabstieg: Formaler Ansatz

Unterscheide zwei Fälle:

- Das Neuron u ist ein **Ausgabeneuron**.
- Das Neuron u ist ein **verstecktes Neuron**.

Im ersten Fall haben wir

$$\forall u \in U_{\text{out}}: \qquad \delta_u^{(l)} = \left(o_u^{(l)} - \text{out}_u^{(l)}\right) \frac{\partial \, \text{out}_u^{(l)}}{\partial \, \text{net}_u^{(l)}}$$

Damit ergibt sich für den Gradienten

$$\forall u \in U_{\text{out}}: \qquad \nabla_{\boldsymbol{w}_u} e_u^{(l)} = \frac{\partial e_u^{(l)}}{\partial \boldsymbol{w}_u} = -2 \left(o_u^{(l)} - \text{out}_u^{(l)}\right) \frac{\partial \, \text{out}_u^{(l)}}{\partial \, \text{net}_u^{(l)}} \, \mathbf{in}_u^{(l)}$$

und damit für die Gewichtsänderung

$$\forall u \in U_{\text{out}}: \qquad \Delta \boldsymbol{w}_u^{(l)} = -\frac{\eta}{2} \nabla_{\boldsymbol{w}_u} e_u^{(l)} = \eta \left(o_u^{(l)} - \text{out}_u^{(l)}\right) \frac{\partial \, \text{out}_u^{(l)}}{\partial \, \text{net}_u^{(l)}} \, \mathbf{in}_u^{(l)}.$$

Gradientenabstieg: Formaler Ansatz

Genaue Formel hängt von der Wahl der Aktivierungs- und Ausgabefunktion ab, da gilt

$$\mathrm{out}_u^{(l)} = f_{\mathrm{out}}(\,\mathrm{act}_u^{(l)}) = f_{\mathrm{out}}(f_{\mathrm{act}}(\,\mathrm{net}_u^{(l)})).$$

Betrachte Spezialfall mit

- Ausgabefunktion ist die Identität,
- Aktivierungsfunktion ist logistisch, d.h. $f_{\mathrm{act}}(x) = \frac{1}{1+e^{-x}}$.

Die erste Annahme ergibt

$$\frac{\partial\,\mathrm{out}_u^{(l)}}{\partial\,\mathrm{net}_u^{(l)}} = \frac{\partial\,\mathrm{act}_u^{(l)}}{\partial\,\mathrm{net}_u^{(l)}} = f'_{\mathrm{act}}(\,\mathrm{net}_u^{(l)}).$$

Gradientenabstieg: Formaler Ansatz

Für eine logistische Aktivierungsfunktion ergibt sich

$$\begin{aligned} f'_{\text{act}}(x) &= \frac{\mathrm{d}}{\mathrm{d}x}\left(1+e^{-x}\right)^{-1} = -\left(1+e^{-x}\right)^{-2}\left(-e^{-x}\right) \\ &= \frac{1+e^{-x}-1}{(1+e^{-x})^2} = \frac{1}{1+e^{-x}}\left(1-\frac{1}{1+e^{-x}}\right) \\ &= f_{\text{act}}(x)\cdot(1-f_{\text{act}}(x)), \end{aligned}$$

und daher

$$f'_{\text{act}}\left(\text{net}_u^{(l)}\right) = f_{\text{act}}\left(\text{net}_u^{(l)}\right)\cdot\left(1-f_{\text{act}}\left(\text{net}_u^{(l)}\right)\right) = \text{out}_u^{(l)}\left(1-\text{out}_u^{(l)}\right).$$

Die sich ergebende Gewichtsänderung ist daher

$$\Delta\boldsymbol{w}_u^{(l)} = \eta\left(o_u^{(l)}-\text{out}_u^{(l)}\right)\text{out}_u^{(l)}\left(1-\text{out}_u^{(l)}\right)\ \mathbf{in}_u^{(l)},$$

womit die Berechnungen sehr einfach werden.

Jetzt: Das Neuron u ist ein **verstecktes Neuron**, d.h. $u \in U_k$, $0 < k < r-1$.

Die Ausgabe $\mathrm{out}_v^{(l)}$ eines Ausgabeneurons v hängt von der Netzeingabe $\mathrm{net}_u^{(l)}$ nur indirekt durch seine Nachfolgeneuronen $\mathrm{succ}(u) = \{s \in U \mid (u,s) \in C\} = \{s_1, \ldots, s_m\} \subseteq U_{k+1}$ ab, insbesondere durch deren Netzeingaben $\mathrm{net}_s^{(l)}$.

Wir wenden die Kettenregel an und erhalten

$$\delta_u^{(l)} = \sum_{v \in U_{\mathrm{out}}} \sum_{s \in \mathrm{succ}(u)} (o_v^{(l)} - \mathrm{out}_v^{(l)}) \frac{\partial \mathrm{out}_v^{(l)}}{\partial \mathrm{net}_s^{(l)}} \frac{\partial \mathrm{net}_s^{(l)}}{\partial \mathrm{net}_u^{(l)}}.$$

Summentausch ergibt

$$\delta_u^{(l)} = \sum_{s \in \mathrm{succ}(u)} \left(\sum_{v \in U_{\mathrm{out}}} (o_v^{(l)} - \mathrm{out}_v^{(l)}) \frac{\partial \mathrm{out}_v^{(l)}}{\partial \mathrm{net}_s^{(l)}} \right) \frac{\partial \mathrm{net}_s^{(l)}}{\partial \mathrm{net}_u^{(l)}} = \sum_{s \in \mathrm{succ}(u)} \delta_s^{(l)} \frac{\partial \mathrm{net}_s^{(l)}}{\partial \mathrm{net}_u^{(l)}}.$$

Fehlerrückpropagation

Betrachte die Netzeingabe

$$\mathrm{net}_s^{(l)} = \boldsymbol{w}_s \mathbf{in}_s^{(l)} = \left(\sum_{p \in \mathrm{pred}(s)} w_{sp} \, \mathrm{out}_p^{(l)} \right) - \theta_s,$$

wobei ein Element von $\mathbf{in}_s^{(l)}$ die Ausgabe $\mathrm{out}_u^{(l)}$ des Neurons u ist. Daher ist

$$\frac{\partial \, \mathrm{net}_s^{(l)}}{\partial \, \mathrm{net}_u^{(l)}} = \left(\sum_{p \in \mathrm{pred}(s)} w_{sp} \frac{\partial \, \mathrm{out}_p^{(l)}}{\partial \, \mathrm{net}_u^{(l)}} \right) - \frac{\partial \theta_s}{\partial \, \mathrm{net}_u^{(l)}} = w_{su} \frac{\partial \, \mathrm{out}_u^{(l)}}{\partial \, \mathrm{net}_u^{(l)}},$$

Das Ergebnis ist die rekursive Gleichung (Fehlerrückpropagation)

$$\delta_u^{(l)} = \left(\sum_{s \in \mathrm{succ}(u)} \delta_s^{(l)} w_{su} \right) \frac{\partial \, \mathrm{out}_u^{(l)}}{\partial \, \mathrm{net}_u^{(l)}}.$$

Fehlerrückpropagation

Die sich ergebende Formel für die Gewichtsänderung ist

$$\Delta \boldsymbol{w}_u^{(l)} = -\frac{\eta}{2} \boldsymbol{\nabla}_{\boldsymbol{w}_u} e^{(l)} = \eta \, \delta_u^{(l)} \, \mathbf{in}_u^{(l)} = \eta \left(\sum_{s \in \mathrm{succ}(u)} \delta_s^{(l)} w_{su} \right) \frac{\partial \, \mathrm{out}_u^{(l)}}{\partial \, \mathrm{net}_u^{(l)}} \, \mathbf{in}_u^{(l)}.$$

Betrachte erneut den Spezialfall mit

- Ausgabefunktion: Identität,
- Aktivierungsfunktion: logistisch.

Die sich ergebende Formel für die Gewichtsänderung ist damit

$$\Delta \boldsymbol{w}_u^{(l)} = \eta \left(\sum_{s \in \mathrm{succ}(u)} \delta_s^{(l)} w_{su} \right) \mathrm{out}_u^{(l)} \left(1 - \mathrm{out}_u^{(l)}\right) \mathbf{in}_u^{(l)}.$$

Fehlerrückpropagation: Vorgehensweise

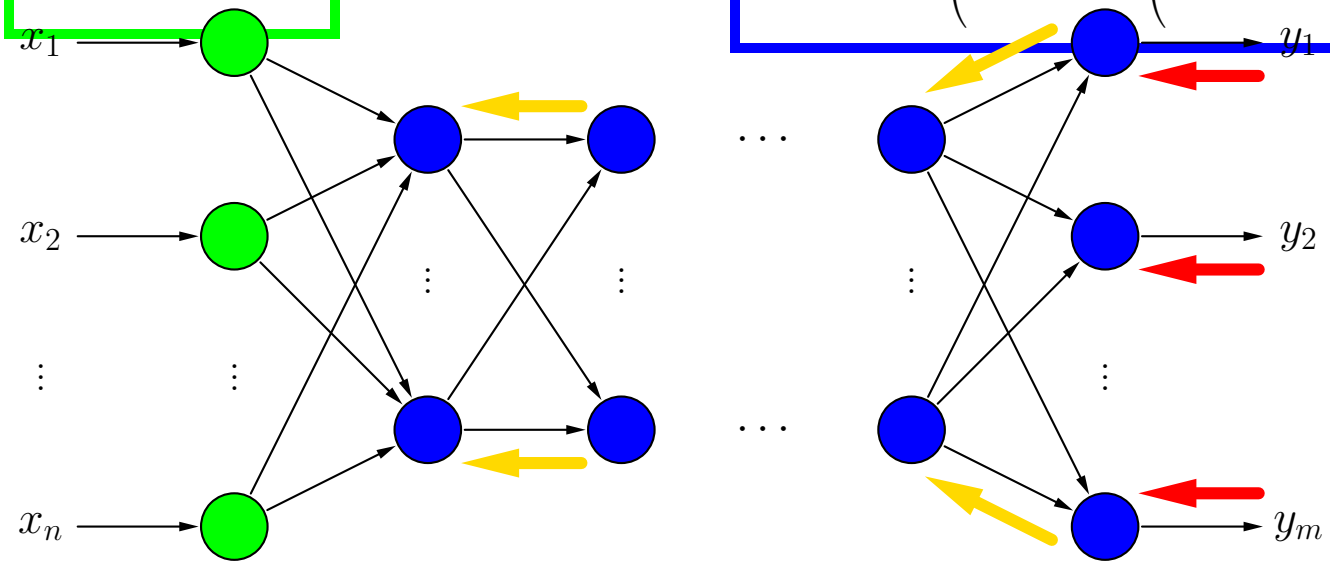
$\forall u \in U_{\text{in}}:$

$\text{out}_u^{(l)} = \text{ex}_u^{(l)}$

Vorwärts-propagation:

$\forall u \in U_{\text{hidden}} \cup U_{\text{out}}:$

$$\text{out}_u^{(l)} = \left(1 + \exp\left(-\Sigma_{p \in \text{pred}(u)}\, w_{up}\, \text{out}_p^{(l)}\right)\right)^{-1}$$



- logistische Aktivierungs-funktion
- impliziter Biaswert

Rückwärts-propagation:

$\forall u \in U_{\text{hidden}}:$

$$\delta_u^{(l)} = \left(\Sigma \ldots \delta_w^{(l)} w_{uw}\right) \lambda_u^{(l)}$$

Aktivierungs-ableitung:

$$\lambda_u^{(l)} = \text{out}_u^{(l)}\left(1 - \text{out}_u^{(l)}\right)$$

Fehlerfaktor:

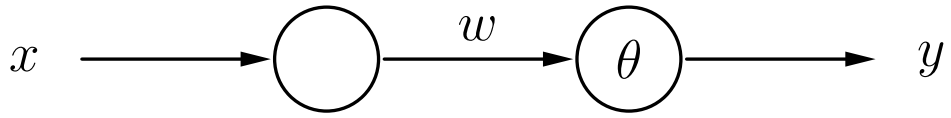
$\forall u \in U_{\text{out}}:$

$$\delta_u^{(l)} = \left(o_u^{(l)} - \text{out}_u^{(l)}\right) \lambda_u^{(l)}$$

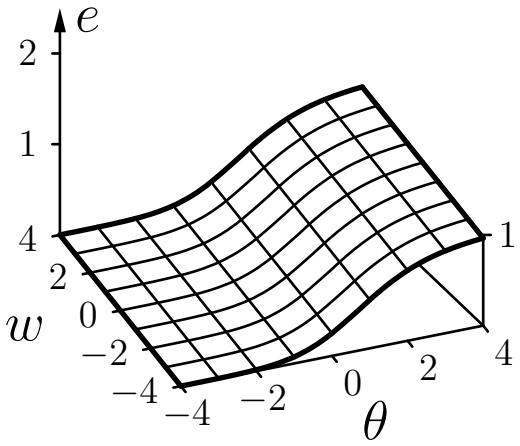
Gewichts-änderung:

$$\Delta w_{up}^{(l)} = \eta\, \delta_u^{(l)}\, \text{out}_p^{(l)}$$

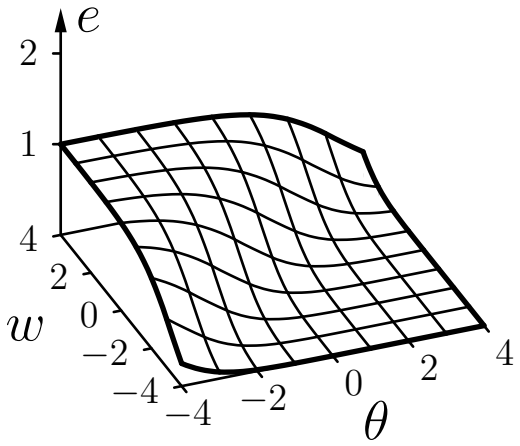
Gradientenabstieg für die Negation $\neg x$



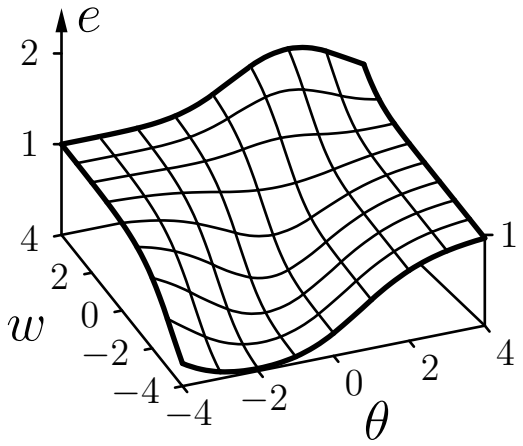
x	y
0	1
1	0



Fehler für $x = 0$



Fehler für $x = 1$



Summe der Fehler

Gradientenabstieg: Beispiele

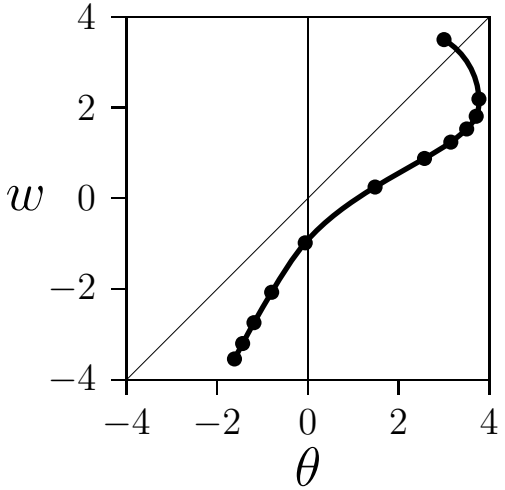
Epoche	θ	w	Fehler
0	3.00	3.50	1.307
20	3.77	2.19	0.986
40	3.71	1.81	0.970
60	3.50	1.53	0.958
80	3.15	1.24	0.937
100	2.57	0.88	0.890
120	1.48	0.25	0.725
140	−0.06	−0.98	0.331
160	−0.80	−2.07	0.149
180	−1.19	−2.74	0.087
200	−1.44	−3.20	0.059
220	−1.62	−3.54	0.044

Online-Training

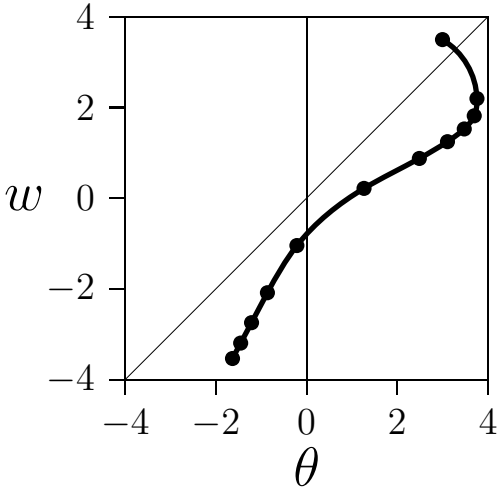
Epoche	θ	w	Fehler
0	3.00	3.50	1.295
20	3.76	2.20	0.985
40	3.70	1.82	0.970
60	3.48	1.53	0.957
80	3.11	1.25	0.934
100	2.49	0.88	0.880
120	1.27	0.22	0.676
140	−0.21	−1.04	0.292
160	−0.86	−2.08	0.140
180	−1.21	−2.74	0.084
200	−1.45	−3.19	0.058
220	−1.63	−3.53	0.044

Batch-Training

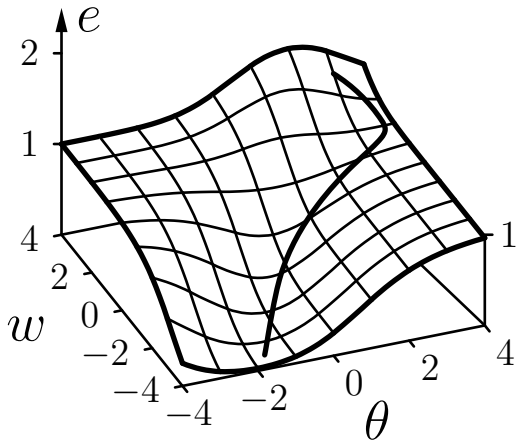
Visualisierung des Gradientenabstiegs für die Negation $\neg x$



Online-Training



Batch-Training



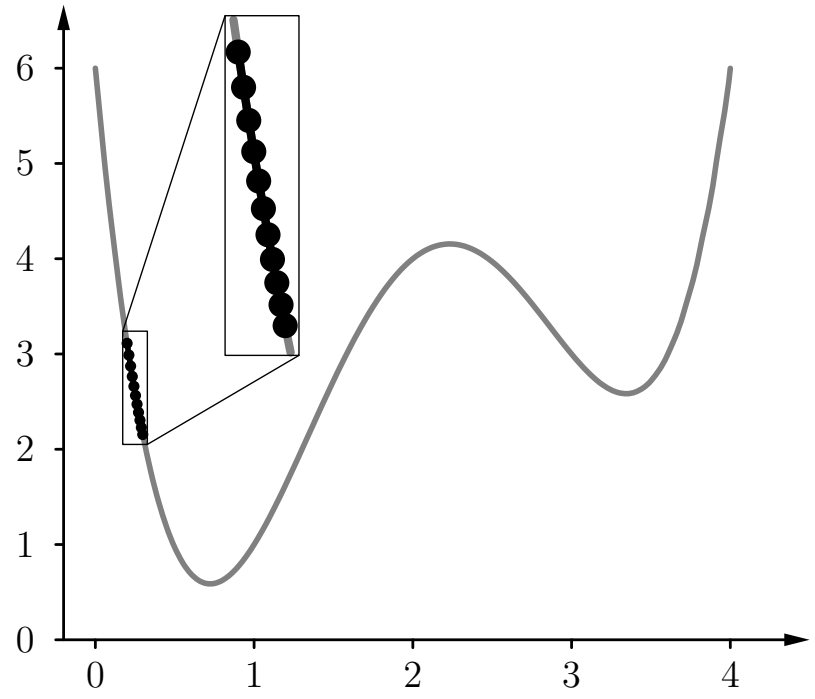
Batch-Training

- Das Training ist offensichtlich erfolgreich.
- Der Fehler kann nicht vollständig verschwinden, bedingt durch die Eigenschaften der logistischen Funktion.

Gradientenabstieg: Beispiele

Beispielfunktion:
$$f(x) = \frac{5}{6}x^4 - 7x^3 + \frac{115}{6}x^2 - 18x + 6,$$

i	x_i	$f(x_i)$	$f'(x_i)$	Δx_i
0	0.200	3.112	−11.147	0.011
1	0.211	2.990	−10.811	0.011
2	0.222	2.874	−10.490	0.010
3	0.232	2.766	−10.182	0.010
4	0.243	2.664	−9.888	0.010
5	0.253	2.568	−9.606	0.010
6	0.262	2.477	−9.335	0.009
7	0.271	2.391	−9.075	0.009
8	0.281	2.309	−8.825	0.009
9	0.289	2.233	−8.585	0.009
10	0.298	2.160		

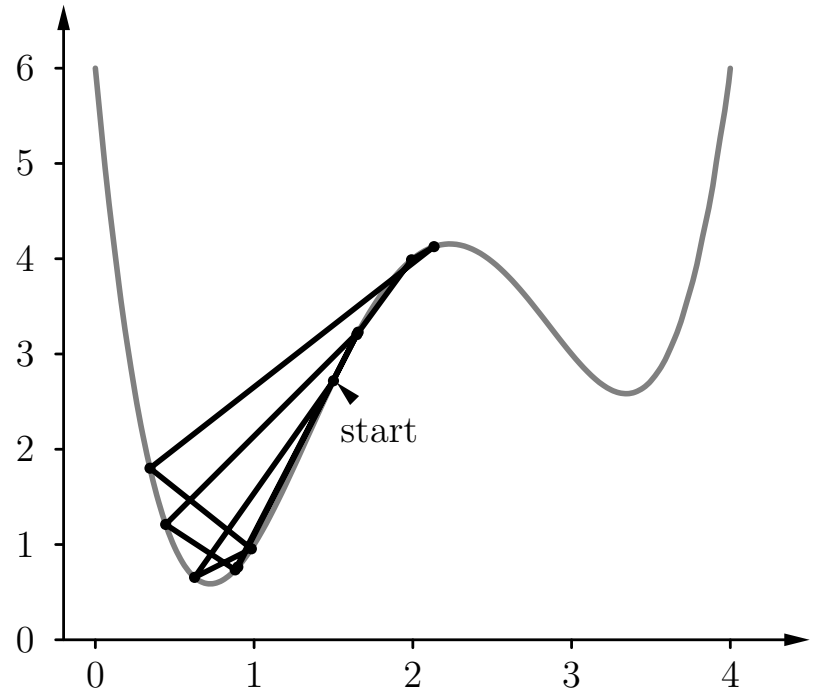


Gradientenabstieg mit Startwert 0.2 und Lernrate 0.001.

Gradientenabstieg: Beispiele

Beispielfunktion: $$f(x) = \frac{5}{6}x^4 - 7x^3 + \frac{115}{6}x^2 - 18x + 6,$$

i	x_i	$f(x_i)$	$f'(x_i)$	Δx_i
0	1.500	2.719	3.500	−0.875
1	0.625	0.655	−1.431	0.358
2	0.983	0.955	2.554	−0.639
3	0.344	1.801	−7.157	1.789
4	2.134	4.127	0.567	−0.142
5	1.992	3.989	1.380	−0.345
6	1.647	3.203	3.063	−0.766
7	0.881	0.734	1.753	−0.438
8	0.443	1.211	−4.851	1.213
9	1.656	3.231	3.029	−0.757
10	0.898	0.766		

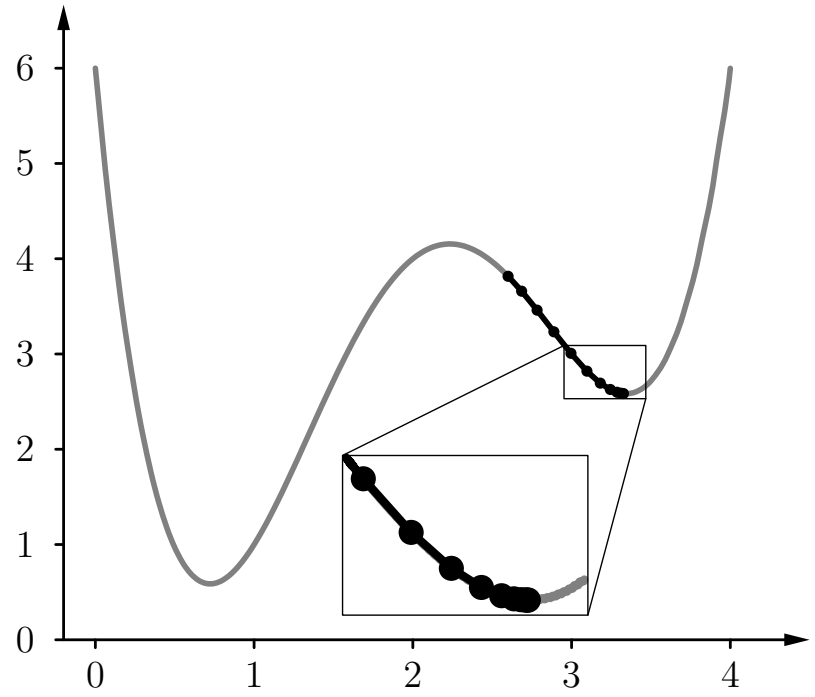


Gradientenabstieg mit Startwert 1.5 und Lernrate 0.25.

Gradientenabstieg: Beispiele

Beispielfunktion: $$f(x) = \frac{5}{6}x^4 - 7x^3 + \frac{115}{6}x^2 - 18x + 6,$$

i	x_i	$f(x_i)$	$f'(x_i)$	Δx_i
0	2.600	3.816	−1.707	0.085
1	2.685	3.660	−1.947	0.097
2	2.783	3.461	−2.116	0.106
3	2.888	3.233	−2.153	0.108
4	2.996	3.008	−2.009	0.100
5	3.097	2.820	−1.688	0.084
6	3.181	2.695	−1.263	0.063
7	3.244	2.628	−0.845	0.042
8	3.286	2.599	−0.515	0.026
9	3.312	2.589	−0.293	0.015
10	3.327	2.585		



Gradientenabstieg mit Startwert 2.6 und Lernrate 0.05.

Gradientenabstieg: Varianten

Gewichts-Updateregel:

$$w(t+1) = w(t) + \Delta w(t)$$

Standard-Backpropagation:

$$\Delta w(t) = -\frac{\eta}{2} \nabla_w e(t)$$

Manhattan-Training:

$$\Delta w(t) = -\eta \, \mathrm{sgn}(\nabla_w e(t))$$

d.h. es wird nur die Richtung (Vorzeichen) der Änderung beachtet und eine feste Schrittweite gewählt

Moment-Term:

$$\Delta w(t) = -\frac{\eta}{2} \nabla_w e(t) + \beta \, \Delta w(t-1),$$

d.h. bei jedem Schritt wird noch ein gewisser Anteil des vorherigen Änderungsschritts mit berücksichtigt, was zu einer Beschleunigung führen kann

Selbstadaptive Fehlerrückpropagation:

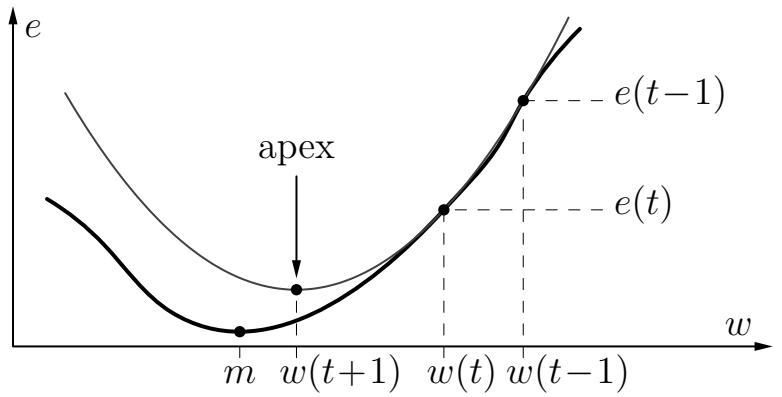
$$\eta_w(t) = \begin{cases} c^- \cdot \eta_w(t-1), & \text{falls } \nabla_w e(t) \cdot \nabla_w e(t-1) < 0, \\ c^+ \cdot \eta_w(t-1), & \text{falls } \nabla_w e(t) \cdot \nabla_w e(t-1) > 0 \\ & \quad \wedge \nabla_w e(t-1) \cdot \nabla_w e(t-2) \geq 0, \\ \eta_w(t-1), & \text{sonst.} \end{cases}$$

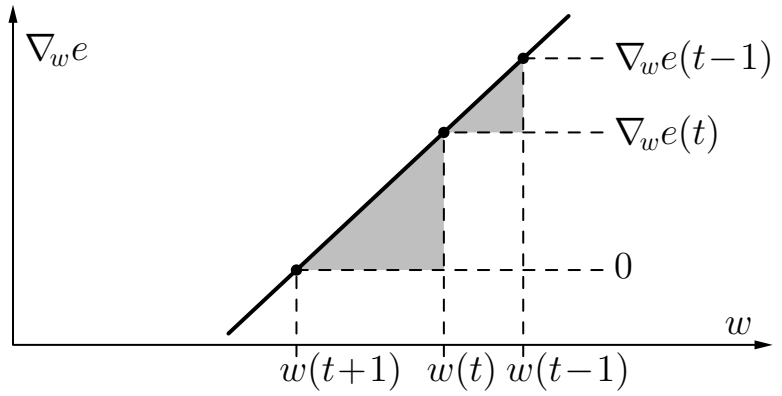
Elastische Fehlerrückpropagation:

$$\Delta w(t) = \begin{cases} c^- \cdot \Delta w(t-1), & \text{falls } \nabla_w e(t) \cdot \nabla_w e(t-1) < 0, \\ c^+ \cdot \Delta w(t-1), & \text{falls } \nabla_w e(t) \cdot \nabla_w e(t-1) > 0 \\ & \quad \wedge \nabla_w e(t-1) \cdot \nabla_w e(t-2) \geq 0, \\ \Delta w(t-1), & \text{sonst.} \end{cases}$$

Typische Werte: $c^- \in [0.5, 0.7]$ und $c^+ \in [1.05, 1.2]$.

Quickpropagation





Die Gewichts-Updateregel kann aus den Dreiecken abgeleitet werden:

$$\Delta w(t) = \frac{\nabla_w e(t)}{\nabla_w e(t-1) - \nabla_w e(t)} \cdot \Delta w(t-1).$$

Gradientenabstieg: Beispiele

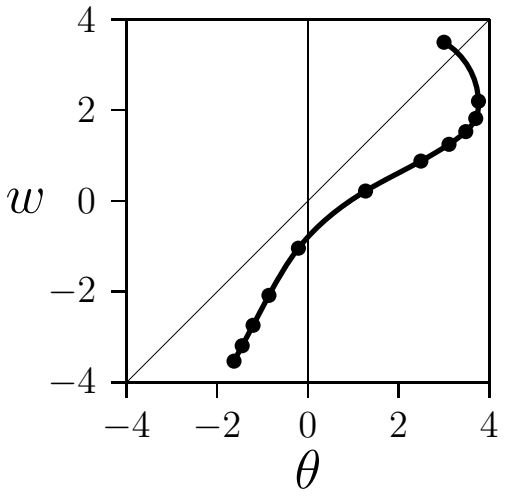
Epoche	θ	w	Fehler
0	3.00	3.50	1.295
20	3.76	2.20	0.985
40	3.70	1.82	0.970
60	3.48	1.53	0.957
80	3.11	1.25	0.934
100	2.49	0.88	0.880
120	1.27	0.22	0.676
140	−0.21	−1.04	0.292
160	−0.86	−2.08	0.140
180	−1.21	−2.74	0.084
200	−1.45	−3.19	0.058
220	−1.63	−3.53	0.044

ohne Momentterm

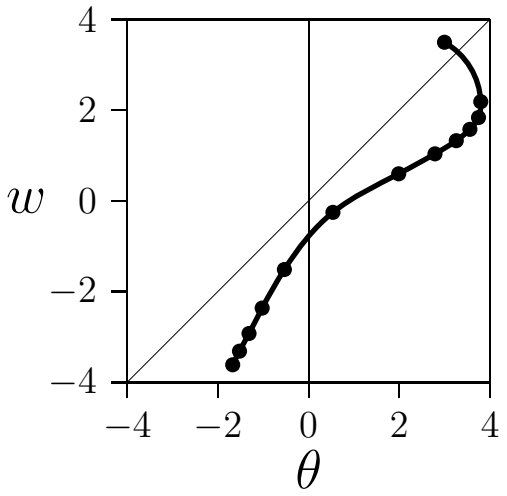
Epoche	θ	w	Fehler
0	3.00	3.50	1.295
10	3.80	2.19	0.984
20	3.75	1.84	0.971
30	3.56	1.58	0.960
40	3.26	1.33	0.943
50	2.79	1.04	0.910
60	1.99	0.60	0.814
70	0.54	−0.25	0.497
80	−0.53	−1.51	0.211
90	−1.02	−2.36	0.113
100	−1.31	−2.92	0.073
110	−1.52	−3.31	0.053
120	−1.67	−3.61	0.041

mit Momentterm

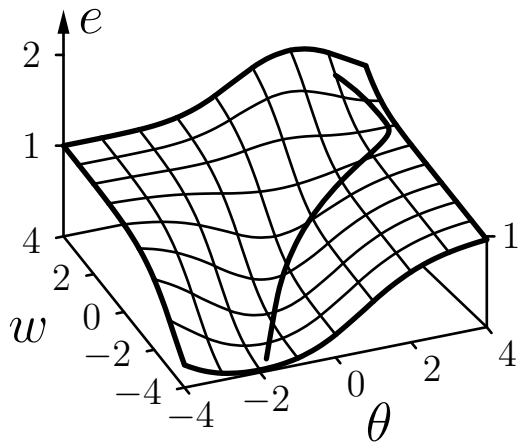
Gradientenabstieg: Beispiele



ohne Momentterm



mit Momentterm



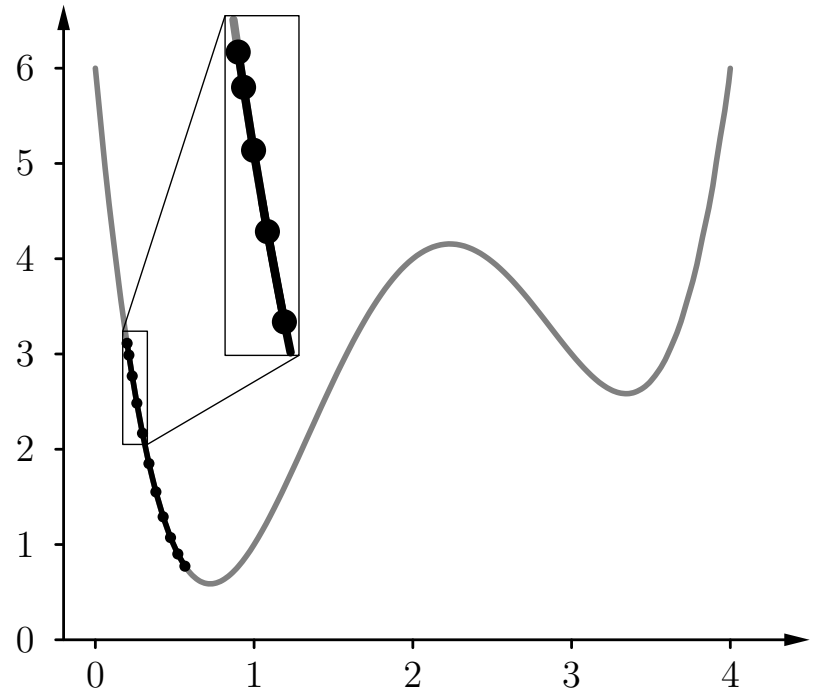
mit Momentterm

- Punkte zeigen die Position alle 20 (ohne Momentterm) oder alle zehn Epochen (mit Momentterm).
- Lernen mit Momentterm ist ungefähr doppelt so schnell.

Gradientenabstieg: Beispiele

Beispielfunktion: $$f(x) = \frac{5}{6}x^4 - 7x^3 + \frac{115}{6}x^2 - 18x + 6,$$

i	x_i	$f(x_i)$	$f'(x_i)$	Δx_i
0	0.200	3.112	−11.147	0.011
1	0.211	2.990	−10.811	0.021
2	0.232	2.771	−10.196	0.029
3	0.261	2.488	−9.368	0.035
4	0.296	2.173	−8.397	0.040
5	0.337	1.856	−7.348	0.044
6	0.380	1.559	−6.277	0.046
7	0.426	1.298	−5.228	0.046
8	0.472	1.079	−4.235	0.046
9	0.518	0.907	−3.319	0.045
10	0.562	0.777		

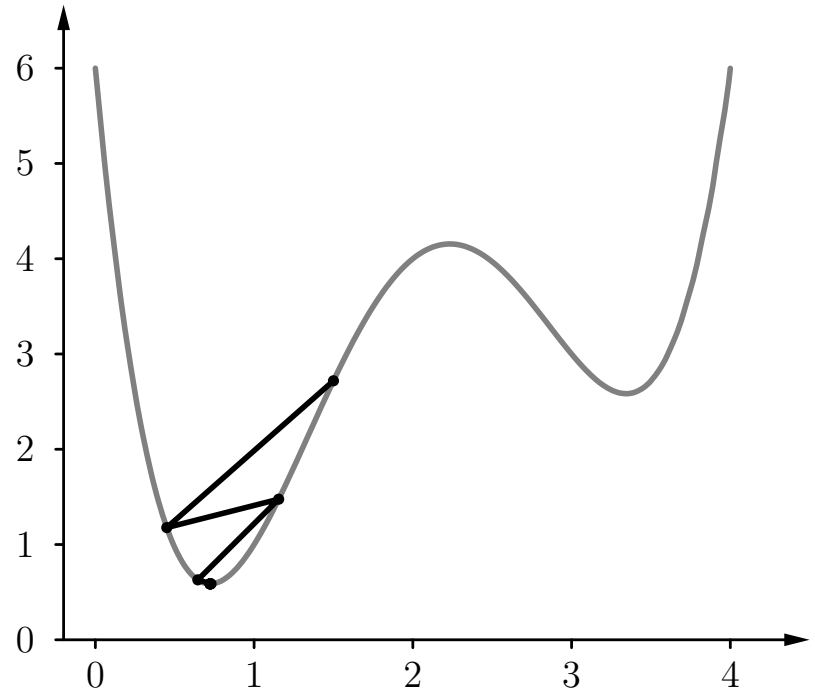


Gradientenabstieg mit Momentterm($\beta = 0.9$)

Gradientenabstieg: Beispiele

Beispielfunktion: $$f(x) = \frac{5}{6}x^4 - 7x^3 + \frac{115}{6}x^2 - 18x + 6,$$

i	x_i	$f(x_i)$	$f'(x_i)$	Δx_i
0	1.500	2.719	3.500	−1.050
1	0.450	1.178	−4.699	0.705
2	1.155	1.476	3.396	−0.509
3	0.645	0.629	−1.110	0.083
4	0.729	0.587	0.072	−0.005
5	0.723	0.587	0.001	0.000
6	0.723	0.587	0.000	0.000
7	0.723	0.587	0.000	0.000
8	0.723	0.587	0.000	0.000
9	0.723	0.587	0.000	0.000
10	0.723	0.587		



Gradientenabstieg mit selbstadaptierender Lernrate ($c^+ = 1.2$, $c^- = 0.5$).

Andere Erweiterungen der Fehlerrückpropagation

Flat Spot Elimination:

$$\Delta w(t) = -\frac{\eta}{2}\nabla_w e(t) + \zeta$$

- Eliminiert langsames Lernen in der Sättigungsregion der logistischen Funktion.
- Wirkt dem Verfall der Fehlersignale über die Schichten entgegen.

Gewichtsverfall: (engl. weight decay)

$$\Delta w(t) = -\frac{\eta}{2}\nabla_w e(t) - \xi\, w(t),$$

- Kann helfen, die Robustheit der Trainingsergebnisse zu verbessern.
- Kann aus einer erweiterten Fehlerfunktion abgeleitet werden, die große Gewichte bestraft:

$$e^* = e + \frac{\xi}{2} \sum_{u \in U_{\text{out}} \cup U_{\text{hidden}}} \left(\theta_u^2 + \sum_{p \in \text{pred}(u)} w_{up}^2 \right).$$

Sensitivitätsanalyse

Sensitivitätsanalyse

Problem: schwer verständliches Wissen, das in einem gelernten neuronalen Netz gespeichert ist:

- Geometrische oder anderweitig anschauliche Deutung gelingt nur bei einfachen Netzen, versagt aber bei komplexen praktischen Problemen
- Versagen des Vorstellungsvermögens insbesondere bei hochdimensionalen Räumen
- Das neuronale Netz wird zu einer *black box*, die auf unergründliche Weise aus den Eingaben die Ausgaben berechnet.

Idee: Bestimmung des Einflusses einzelner Eingaben auf die Ausgabe des Netzes.

$\rightarrow$ Sensitivitätsanalyse

Sensitivitätsanalyse

Frage: Wie wichtig sind einzelne Eingaben für das Netzwerk?

Idee: Bestimme die Änderung der Ausgabe relativ zur Änderung der Eingabe.

$$\forall u \in U_{\text{in}}: \qquad s(u) = \frac{1}{|L_{\text{fixed}}|} \sum_{l \in L_{\text{fixed}}} \sum_{v \in U_{\text{out}}} \frac{\partial \operatorname{out}_v^{(l)}}{\partial \operatorname{ex}_u^{(l)}}.$$

Formale Herleitung: Wende Kettenregel an.

$$\frac{\partial \operatorname{out}_v}{\partial \operatorname{ex}_u} = \frac{\partial \operatorname{out}_v}{\partial \operatorname{out}_u} \frac{\partial \operatorname{out}_u}{\partial \operatorname{ex}_u} = \frac{\partial \operatorname{out}_v}{\partial \operatorname{net}_v} \frac{\partial \operatorname{net}_v}{\partial \operatorname{out}_u} \frac{\partial \operatorname{out}_u}{\partial \operatorname{ex}_u}.$$

Vereinfachung: Nimm an, dass die Ausgabefunktion die Identität ist.

$$\frac{\partial \operatorname{out}_u}{\partial \operatorname{ex}_u} = 1.$$

Für den zweiten Faktor bekommen wir das allgemeine Ergebnis:

$$\frac{\partial \operatorname{net}_v}{\partial \operatorname{out}_u} = \frac{\partial}{\partial \operatorname{out}_u} \sum_{p \in \operatorname{pred}(v)} w_{vp} \operatorname{out}_p = \sum_{p \in \operatorname{pred}(v)} w_{vp} \frac{\partial \operatorname{out}_p}{\partial \operatorname{out}_u}.$$

Das führt zur Rekursionsformel

$$\frac{\partial \operatorname{out}_v}{\partial \operatorname{out}_u} = \frac{\partial \operatorname{out}_v}{\partial \operatorname{net}_v} \frac{\partial \operatorname{net}_v}{\partial \operatorname{out}_u} = \frac{\partial \operatorname{out}_v}{\partial \operatorname{net}_v} \sum_{p \in \operatorname{pred}(v)} w_{vp} \frac{\partial \operatorname{out}_p}{\partial \operatorname{out}_u}.$$

Aber für die erste versteckte Schicht bekommen wir

$$\frac{\partial \operatorname{net}_v}{\partial \operatorname{out}_u} = w_{vu}, \qquad \text{therefore} \qquad \frac{\partial \operatorname{out}_v}{\partial \operatorname{out}_u} = \frac{\partial \operatorname{out}_v}{\partial \operatorname{net}_v} w_{vu}.$$

Diese Formel stellt den Beginn der Rekursion dar.

Sensitivitätsanalyse

Betrachte (wie üblich) den Spezialfall, bei dem

- die Ausgabefunktion die Identität ist
- und die Aktivierungsfunktion logistisch ist.

In diesem Fall lautet die Rekursionsformel

$$\frac{\partial \operatorname{out}_v}{\partial \operatorname{out}_u} = \operatorname{out}_v(1 - \operatorname{out}_v) \sum_{p \in \operatorname{pred}(v)} w_{vp} \frac{\partial \operatorname{out}_p}{\partial \operatorname{out}_u}$$

und der Anker der Rekursion ist

$$\frac{\partial \operatorname{out}_v}{\partial \operatorname{out}_u} = \operatorname{out}_v(1 - \operatorname{out}_v) w_{vu}.$$

Radiale-Basisfunktionen-Netze

Eigenschaften von Radiale-Basisfunktionen-Netzen (RBF-Netzen)

- RBF-Netze sind streng geschichtete, vorwärtsbetriebene neuronale Netze mit genau einer versteckten Schicht.
- Als Netzeingabe- und Aktivierungsfunktion werden radiale Basisfunktionen verwendet.
- Jedes Neuron erhält eine Art "Einzugsgebiet".
- Die Gewichte der Verbindungen von der Eingabeschicht zu einem Neuron geben das Zentrum an.

Radiale-Basisfunktionen-Netze

Ein **radiale-Basisfunktionen-Netz (RBF-Netz)** ist ein neuronales Netz mit einem Graph $G = (U, C)$, das die folgenden Bedingungen erfüllt:

(i) $U_{\text{in}} \cap U_{\text{out}} = \emptyset$,

(ii) $C = (U_{\text{in}} \times U_{\text{hidden}}) \cup C', \quad C' \subseteq (U_{\text{hidden}} \times U_{\text{out}})$

Die Netzeingabefunktion jedes versteckten Neurons ist eine **Abstandsfunktion** zwischen dem Eingabevektor und dem Gewichtsvektor, d.h.

$$\forall u \in U_{\text{hidden}}: \qquad f_{\text{net}}^{(u)}(\boldsymbol{w}_u, \mathbf{in}_u) = d(\boldsymbol{w}_u, \mathbf{in}_u),$$

wobei $d : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}_0^+$ eine Funktion ist, die $\forall \boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z} \in \mathbb{R}^n$: erfüllt:

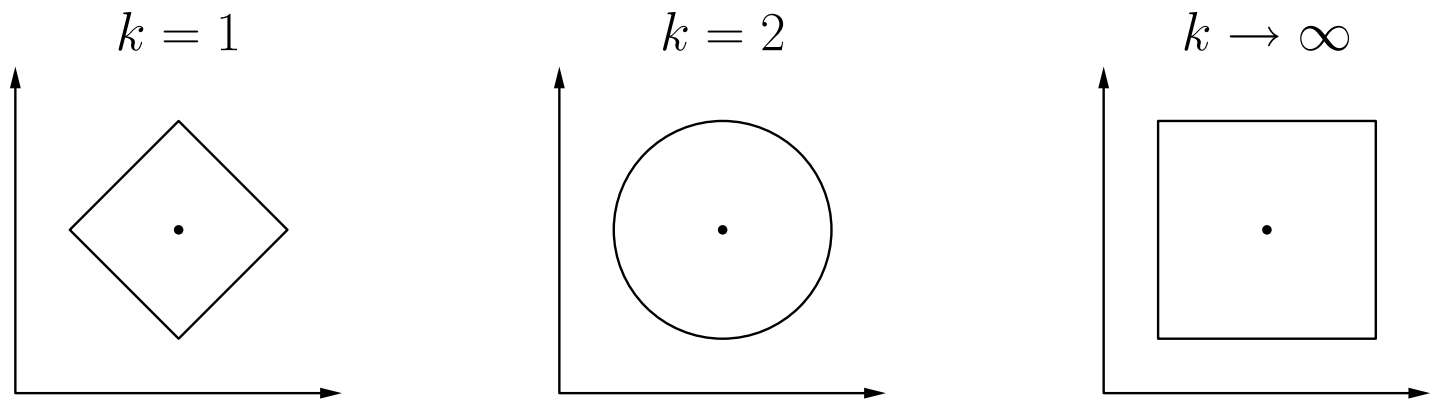
$$\begin{array}{rll} (i) & d(\boldsymbol{x}, \boldsymbol{y}) = 0 \quad \Leftrightarrow \quad \boldsymbol{x} = \boldsymbol{y}, & \\ (ii) & d(\boldsymbol{x}, \boldsymbol{y}) = d(\boldsymbol{y}, \boldsymbol{x}) & \text{(Symmetrie)}, \\ (iii) & d(\boldsymbol{x}, \boldsymbol{z}) \leq d(\boldsymbol{x}, \boldsymbol{y}) + d(\boldsymbol{y}, \boldsymbol{z}) & \text{(Dreiecksungleichung)}. \end{array}$$

Veranschaulichung von Abstandsfunktionen

$$d_k(\boldsymbol{x}, \boldsymbol{y}) = \left(\sum_{i=1}^{n} (x_i - y_i)^k \right)^{\frac{1}{k}}$$

Bekannte Spezialfälle dieser Familie sind:

$k = 1$: Manhattan-Abstand ,
$k = 2$: Euklidischer Abstand,
$k \to \infty$: Maximum-Abstand, d.h. $d_\infty(\boldsymbol{x}, \boldsymbol{y}) = \max_{i=1}^{n} |x_i - y_i|$.



(alle Punkte auf dem Kreis bzw. den Vierecken haben denselben Abstand zum Mittelpunkt, entsprechend der jeweiligen Abstandsfunktion)

Radiale-Basisfunktionen-Netze

Die Netzeingabefunktion der Ausgabeneuronen ist die gewichtete Summe ihrer Eingaben, d.h.

$$\forall u \in U_{\text{out}}: \qquad f_{\text{net}}^{(u)}(\boldsymbol{w}_u, \mathbf{in}_u) = \boldsymbol{w}_u \mathbf{in}_u = \sum_{v \in \text{pred}\,(u)} w_{uv}\, \text{out}_v\,.$$

Die Aktivierungsfunktion jedes versteckten Neurons ist eine sogenannte **radiale Funktion**, d.h. eine monoton fallende Funktion

$$f : \mathbb{R}_0^+ \to [0,1] \quad \text{with} \quad f(0) = 1 \quad \text{and} \quad \lim_{x \to \infty} f(x) = 0.$$

Die Aktivierungsfunktion jedes Ausgabeneurons ist eine lineare Funktion

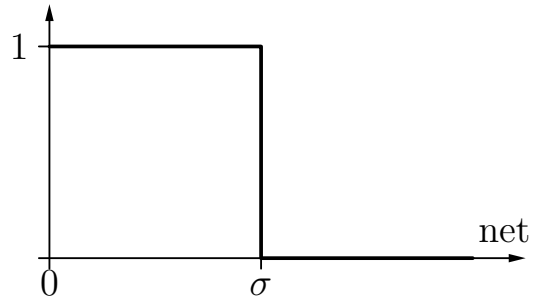
$$f_{\text{act}}^{(u)}(\text{net}_u, \theta_u) = \text{net}_u - \theta_u.$$

(Die lineare Aktivierungsfunktion ist wichtig für die Initialisierung.)

Radiale Aktivierungsfunktionen

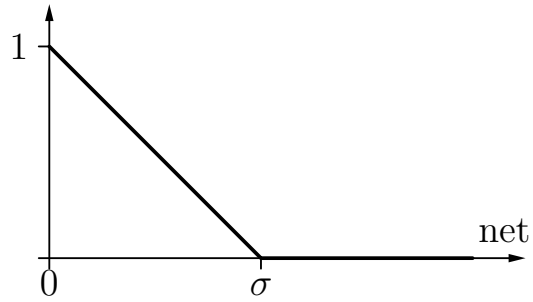
Rechteckfunktion:

$$f_{\text{act}}(\text{net}, \sigma) = \begin{cases} 0, & \text{falls net} > \sigma, \\ 1, & \text{sonst.} \end{cases}$$



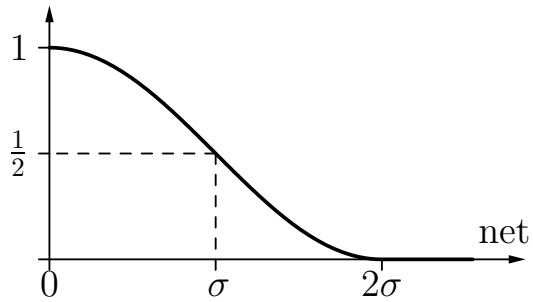
Dreiecksfunktion:

$$f_{\text{act}}(\text{net}, \sigma) = \begin{cases} 0, & \text{falls net} > \sigma, \\ 1 - \frac{\text{net}}{\sigma}, & \text{sonst.} \end{cases}$$



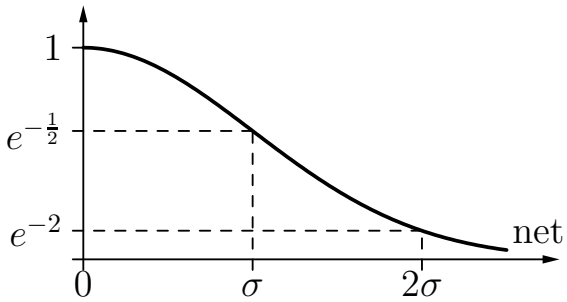
Kosinus bis Null:

$$f_{\text{act}}(\text{net}, \sigma) = \begin{cases} 0, & \text{falls net} > 2\sigma, \\ \frac{\cos\left(\frac{\pi}{2\sigma}\,\text{net}\right)+1}{2}, & \text{sonst.} \end{cases}$$

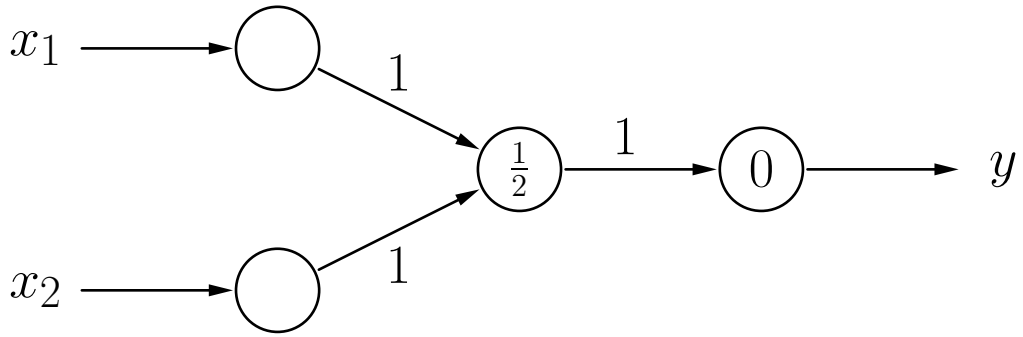


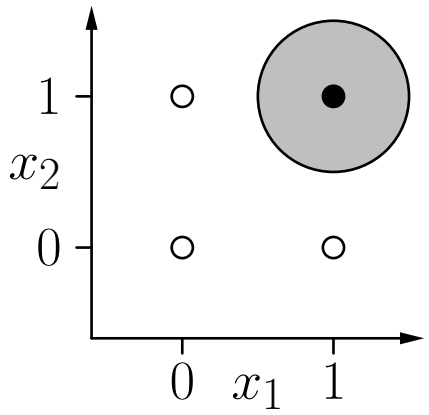
Gaußsche Funktion:

$$f_{\text{act}}(\text{net}, \sigma) = e^{-\frac{\text{net}^2}{2\sigma^2}}$$



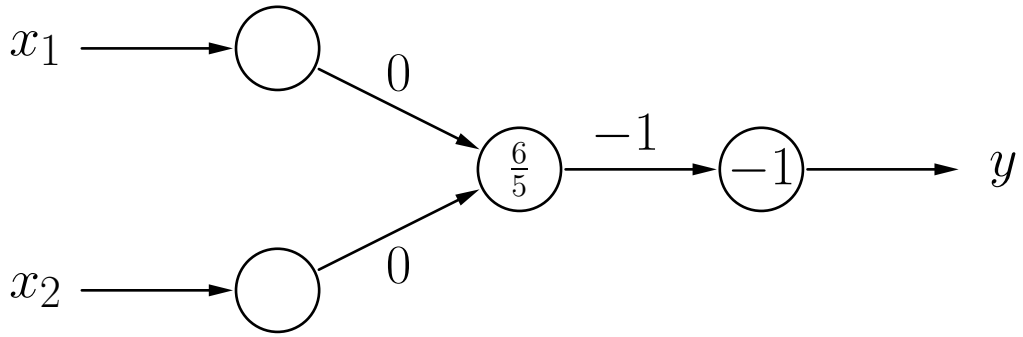
Radiale-Basisfunktionen-Netz für die Konjunktion $x_1 \wedge x_2$

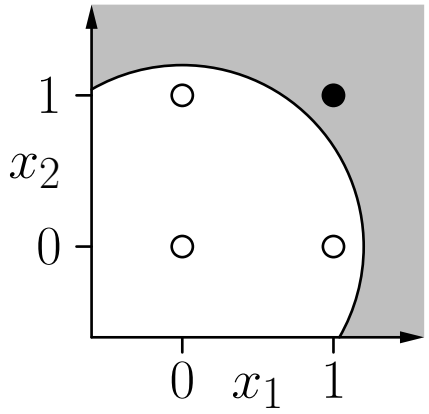




- (1,1) ist Zentrum
- Referenzradius ist $\frac{1}{2}$
- Euklidischer Abstand
- Rechteckfunktion als Aktivierung
- Biaswert 0 im Ausgabeneuron

Radiale-Basisfunktionen-Netz für die Konjunktion $x_1 \wedge x_2$



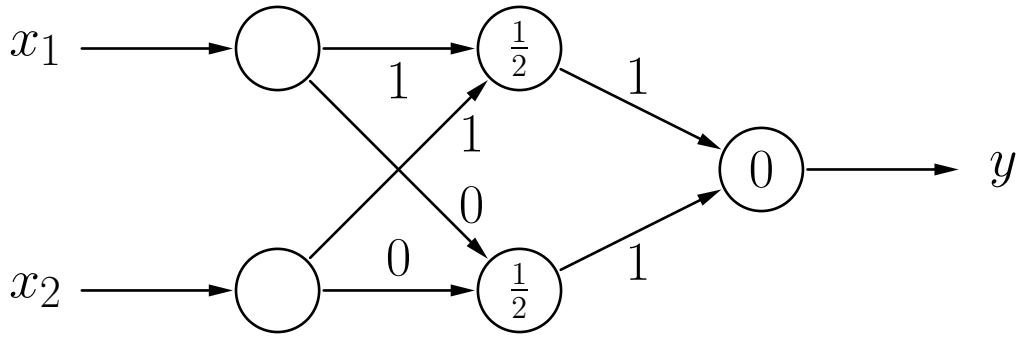


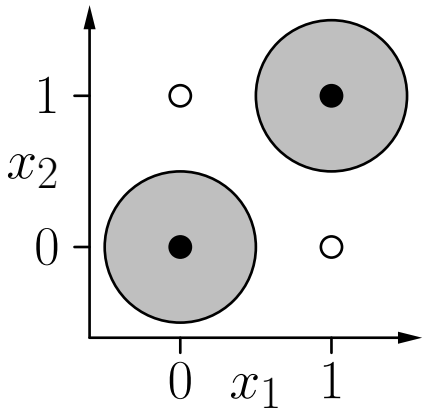
- (0,0) ist Zentrum
- Referenzradius ist $\frac{6}{5}$
- Euklidischer Abstand
- Rechteckfunktion als Aktivierung
- Biaswert -1 im Ausgabeneuron

Radiale-Basisfunktionen-Netz für die Biimplikation $x_1 \leftrightarrow x_2$

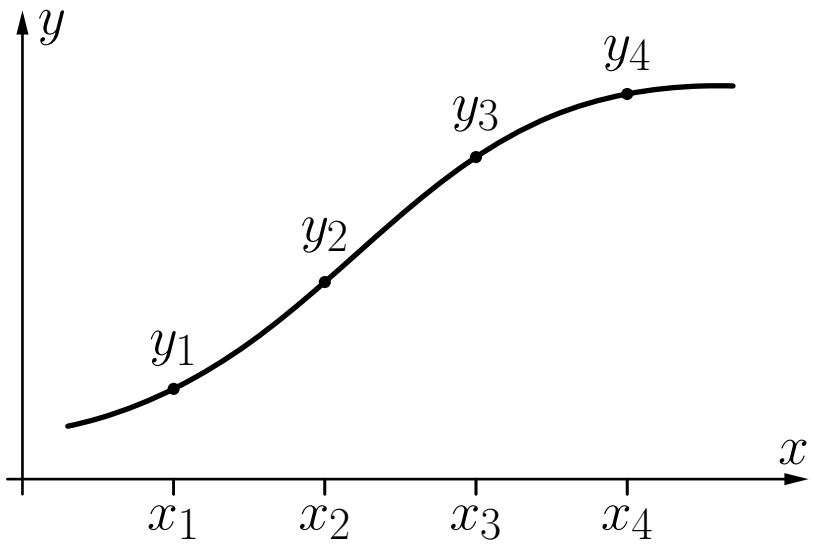
Idee: logische Zerlegung

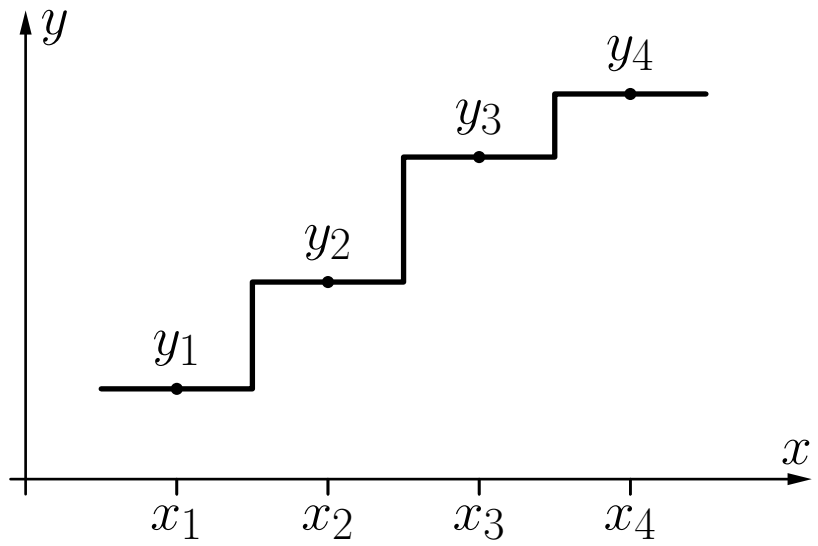
$$x_1 \leftrightarrow x_2 \quad \equiv \quad (x_1 \wedge x_2) \vee \neg(x_1 \vee x_2)$$

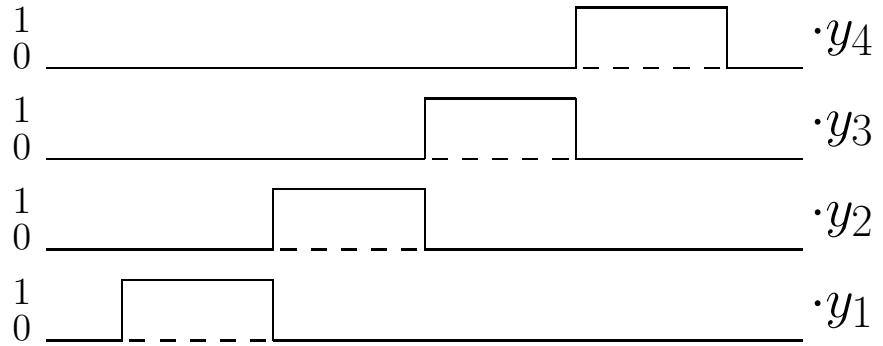




Radiale-Basisfunktionen-Netze: Funktionsapproximation

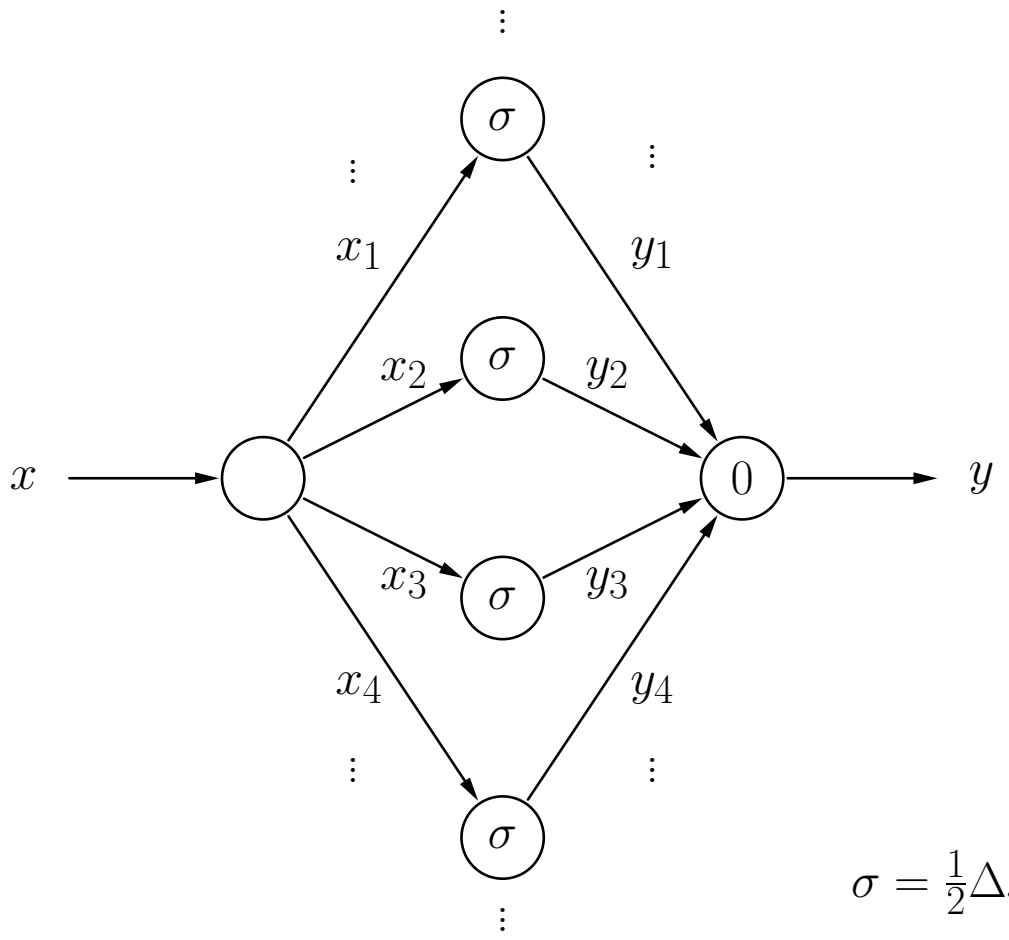






Annäherung der Originalfunktion durch Stufenfunktionen, deren Stufen durch einzelne Neuronen eines RBF-Netzes dargestellt werden können (vgl. MLPs).

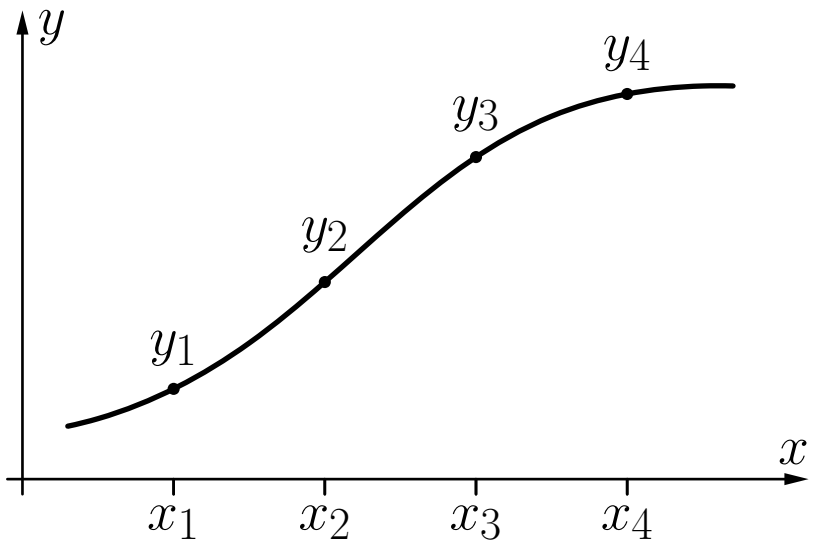
Radiale-Basisfunktionen-Netze: Funktionsapproximation

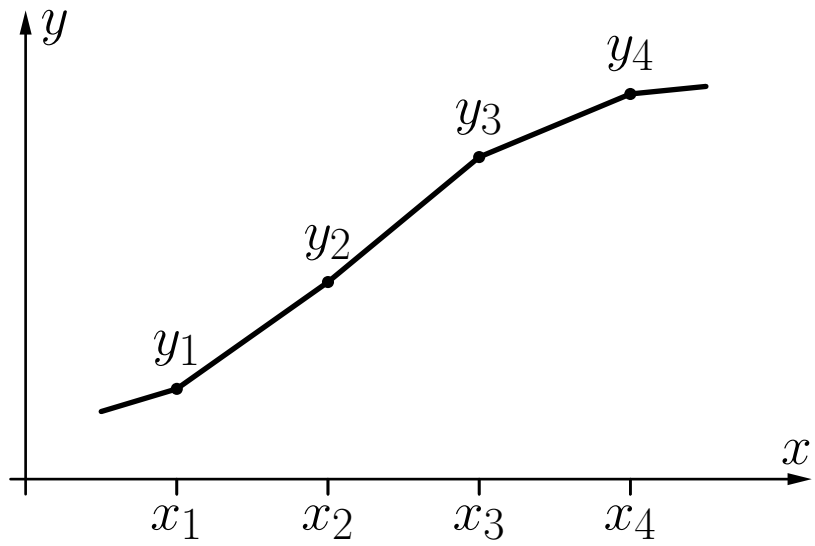


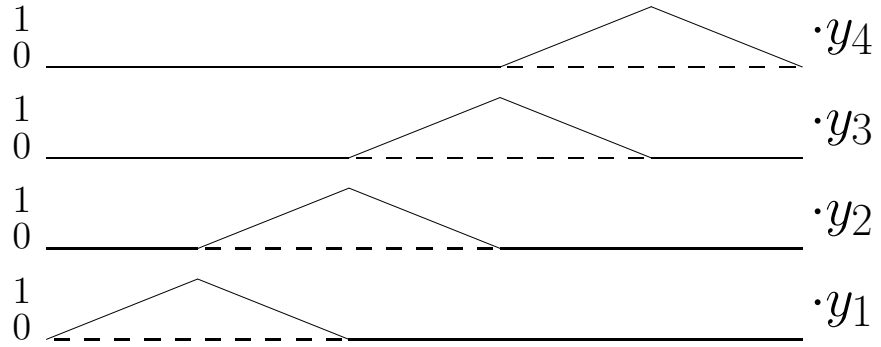
$$\sigma = \frac{1}{2}\Delta x = \frac{1}{2}(x_{i+1} - x_i)$$

Ein RBF-Netz, das die Treppenfunktion von der vorherigen Folie bzw. die stückweise lineare Funktion der folgenden Folie berechnet (dazu muss nur die Aktivierungsfunktion der versteckten Neuronen geändert werden).

Radiale-Basisfunktionen-Netze: Funktionsapproximation

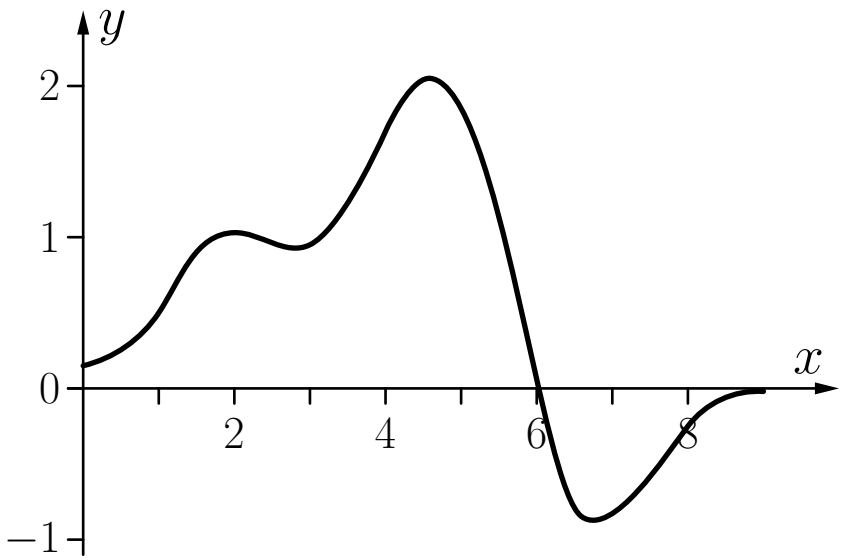


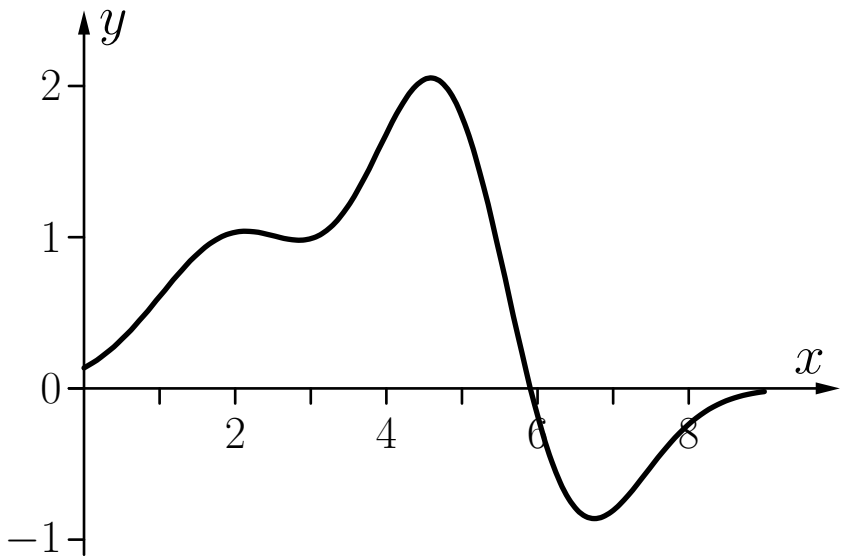


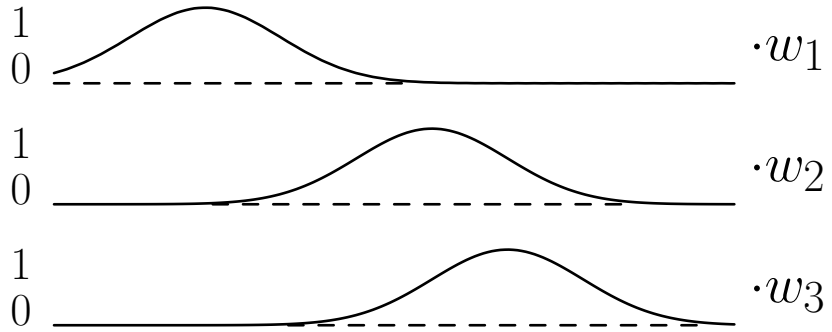


Darstellung einer stückweise linearen Funktion durch eine gewichtete Summe von Dreiecksfunktionen mit Zentren x_i.

Radiale-Basisfunktionen-Netze: Funktionsapproximation

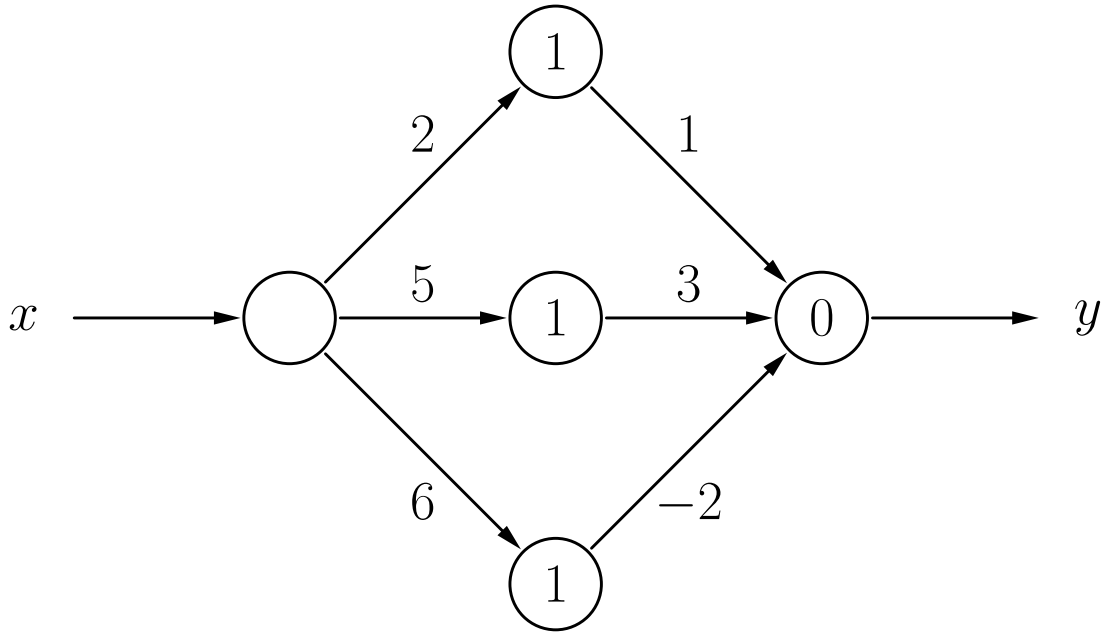






Annäherung einer Funktion durch eine Summe von Gaußkurven mit Radius $\sigma = 1$.
Es ist $w_1 = 2$, $w_2 = 3$ und $w_3 = -2$.

RBF-Netz für eine Summe dreier Gaußfunktionen



Training von RBF-Netzen

Radiale-Basisfunktionen-Netze: Initialisierung

Sei $L_{\text{fixed}} = \{l_1, \ldots, l_m\}$ eine feste Lernaufgabe,
bestehend aus m Trainingsbeispielen $l = (\boldsymbol{\imath}^{(l)}, \boldsymbol{o}^{(l)})$.

Einfaches RBF-Netz:
Ein verstecktes Neuron v_k, $k = 1, \ldots, m$, für jedes Trainingsbeispiel

$$\forall k \in \{1, \ldots, m\}: \qquad \boldsymbol{w}_{v_k} = \boldsymbol{\imath}^{(l_k)}.$$

Falls die Aktivierungsfunktion die Gaußfunktion ist,
werden die Radien σ_k nach einer Heuristik gewählt

$$\forall k \in \{1, \ldots, m\}: \qquad \sigma_k = \frac{d_{\max}}{\sqrt{2m}},$$

wobei

$$d_{\max} = \max_{l_j, l_k \in L_{\text{fixed}}} d\left(\boldsymbol{\imath}^{(l_j)}, \boldsymbol{\imath}^{(l_k)}\right).$$

Initialisieren der Verbindungen von den versteckten zu den Ausgabeneuronen

$$\forall u : \sum_{k=1}^{m} w_{uv_m} \operatorname{out}_{v_m}^{(l)} - \theta_u = o_u^{(l)} \qquad \text{oder (abgekürzt)} \qquad \mathbf{A} \cdot \boldsymbol{w}_u = \boldsymbol{o}_u,$$

wobei $\boldsymbol{o}_u = \left(o_u^{(l_1)}, \ldots, o_u^{(l_m)}\right)^\top$ der Vektor der gewünschten Ausgaben ist, $\theta_u = 0$, und

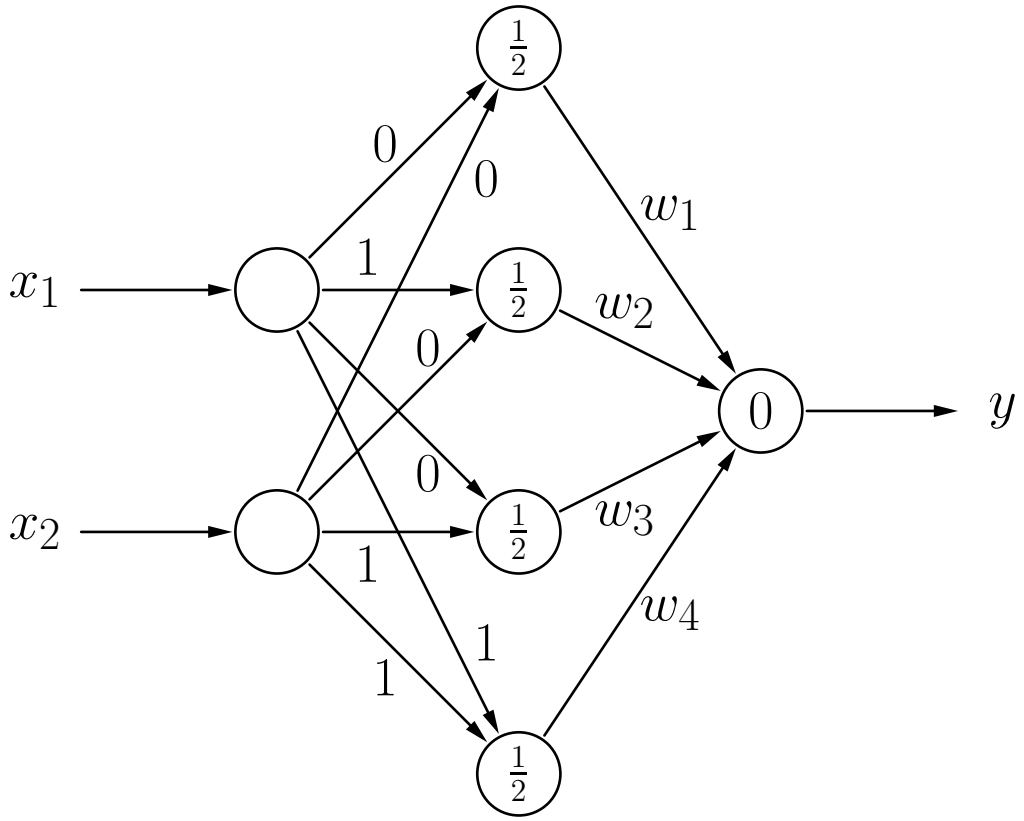
$$\mathbf{A} = \begin{pmatrix} \operatorname{out}_{v_1}^{(l_1)} & \operatorname{out}_{v_2}^{(l_1)} & \ldots & \operatorname{out}_{v_m}^{(l_1)} \\ \operatorname{out}_{v_1}^{(l_2)} & \operatorname{out}_{v_2}^{(l_2)} & \ldots & \operatorname{out}_{v_m}^{(l_2)} \\ \vdots & \vdots & & \vdots \\ \operatorname{out}_{v_1}^{(l_m)} & \operatorname{out}_{v_2}^{(l_m)} & \ldots & \operatorname{out}_{v_m}^{(l_m)} \end{pmatrix}.$$

Ergebnis: Lineares Gleichungssystem, das durch Invertieren der Matrix $\mathbf{A}$ gelöst werden kann:

$$\boldsymbol{w}_u = \mathbf{A}^{-1} \cdot \boldsymbol{o}_u.$$

Einfaches RBF-Netz für die Biimplikation $x_1 \leftrightarrow x_2$

x_1	x_2	y
0	0	1
1	0	0
0	1	0
1	1	1



Einfaches RBF-Netz für die Biimplikation $x_1 \leftrightarrow x_2$

$$\mathbf{A} = \begin{pmatrix} 1 & e^{-2} & e^{-2} & e^{-4} \\ e^{-2} & 1 & e^{-4} & e^{-2} \\ e^{-2} & e^{-4} & 1 & e^{-2} \\ e^{-4} & e^{-2} & e^{-2} & 1 \end{pmatrix} \qquad \mathbf{A}^{-1} = \begin{pmatrix} \frac{a}{D} & \frac{b}{D} & \frac{b}{D} & \frac{c}{D} \\ \frac{b}{D} & \frac{a}{D} & \frac{c}{D} & \frac{b}{D} \\ \frac{b}{D} & \frac{c}{D} & \frac{a}{D} & \frac{b}{D} \\ \frac{c}{D} & \frac{b}{D} & \frac{b}{D} & \frac{a}{D} \end{pmatrix}$$

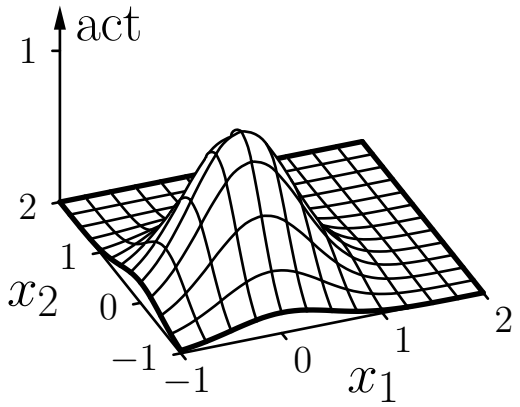
wobei

$$\begin{aligned} D &= 1 - 4e^{-4} + 6e^{-8} - 4e^{-12} + e^{-16} &&\approx \quad 0.9287 \\ a &= \quad 1 \quad - 2e^{-4} + e^{-8} &&\approx \quad 0.9637 \\ b &= -e^{-2} + 2e^{-6} - e^{-10} &&\approx -0.1304 \\ c &= \quad e^{-4} - 2e^{-8} + e^{-12} &&\approx \quad 0.0177 \end{aligned}$$

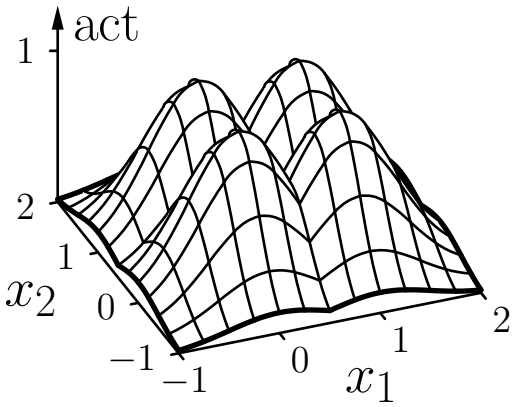
$$\boldsymbol{w}_u \;=\; \mathbf{A}^{-1} \cdot \boldsymbol{o}_u \;=\; \frac{1}{D} \begin{pmatrix} a+c \\ 2b \\ 2b \\ a+c \end{pmatrix} \approx \begin{pmatrix} 1.0567 \\ -0.2809 \\ -0.2809 \\ 1.0567 \end{pmatrix}$$

RBF-Netz-Initialisierung: Beispiel

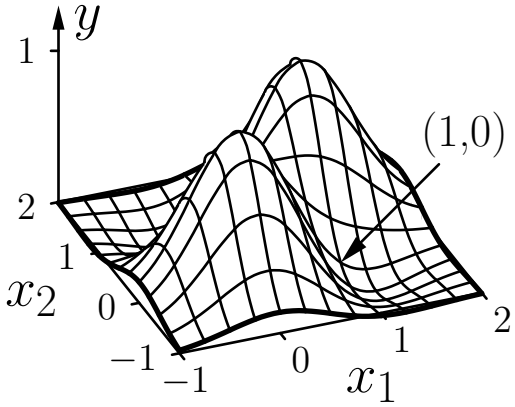
Einfaches RBF-Netz für die Biimplikation $x_1 \leftrightarrow x_2$



einzelne Basisfunktion



alle Basisfunktionen



Ausgabe

- Die Initialisierung führt bereits zu einer perfekten Lösung der Lernaufgabe.
- Weiteres Trainieren ist nicht notwendig.

Radiale-Basisfunktionen-Netze: Initialisierung

Normale Radiale-Basisfunktionen-Netze:
Wähle Teilmenge von k Trainingsbeispielen als Zentren aus.

$$\mathbf{A} = \begin{pmatrix} 1 & \mathrm{out}_{v_1}^{(l_1)} & \mathrm{out}_{v_2}^{(l_1)} & \ldots & \mathrm{out}_{v_k}^{(l_1)} \\ 1 & \mathrm{out}_{v_1}^{(l_2)} & \mathrm{out}_{v_2}^{(l_2)} & \ldots & \mathrm{out}_{v_k}^{(l_2)} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & \mathrm{out}_{v_1}^{(l_m)} & \mathrm{out}_{v_2}^{(l_m)} & \ldots & \mathrm{out}_{v_k}^{(l_m)} \end{pmatrix} \qquad \mathbf{A} \cdot \boldsymbol{w}_u = \boldsymbol{o}_u$$

Berechne (Moore–Penrose)-Pseudoinverse:

$$\mathbf{A}^+ = (\mathbf{A}^\top \mathbf{A})^{-1} \mathbf{A}^\top.$$

Die Gewichte können dann durch

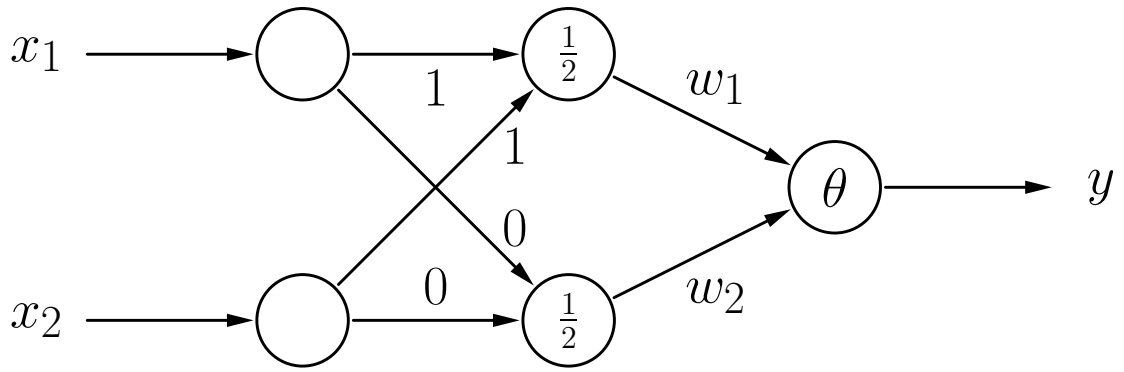
$$\boldsymbol{w}_u = \mathbf{A}^+ \cdot \boldsymbol{o}_u = (\mathbf{A}^\top \mathbf{A})^{-1} \mathbf{A}^\top \cdot \boldsymbol{o}_u$$

berechnet werden.

Normales RBF-Netz für die Biimplikation $x_1 \leftrightarrow x_2$

Wähle zwei Trainingsbeispiele aus:

- $l_1 = (\boldsymbol{\imath}^{(l_1)}, \boldsymbol{o}^{(l_1)}) = ((0,0),(1))$
- $l_4 = (\boldsymbol{\imath}^{(l_4)}, \boldsymbol{o}^{(l_4)}) = ((1,1),(1))$



Normales RBF-Netz für die Biimplikation $x_1 \leftrightarrow x_2$

$$\mathbf{A} = \begin{pmatrix} 1 & 1 & e^{-4} \\ 1 & e^{-2} & e^{-2} \\ 1 & e^{-2} & e^{-2} \\ 1 & e^{-4} & 1 \end{pmatrix} \qquad \mathbf{A}^+ = (\mathbf{A}^\top \mathbf{A})^{-1} \mathbf{A}^\top = \begin{pmatrix} a & b & b & a \\ c & d & d & e \\ e & d & d & c \end{pmatrix}$$

wobei

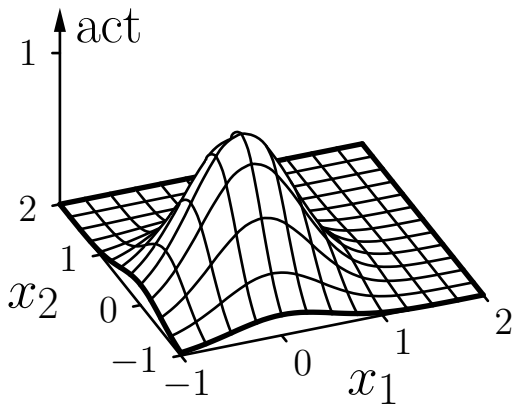
$$\begin{aligned} a &\approx -0.1810, & b &\approx 0.6810, \\ c &\approx 1.1781, & d &\approx -0.6688, & e &\approx 0.1594. \end{aligned}$$

Gewichte:

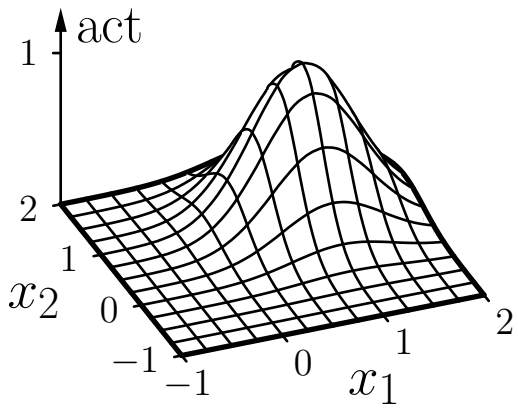
$$\boldsymbol{w}_u = \begin{pmatrix} -\theta \\ w_1 \\ w_2 \end{pmatrix} = \mathbf{A}^+ \cdot \boldsymbol{o}_u \approx \begin{pmatrix} -0.3620 \\ 1.3375 \\ 1.3375 \end{pmatrix}.$$

RBF-Netz-Initialisierung: Beispiel

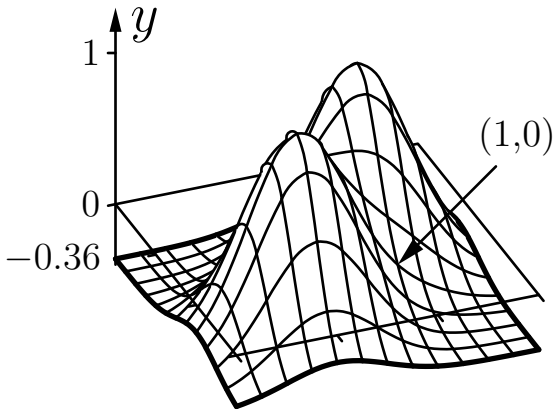
Normales RBF-Netz für die Biimplikation $x_1 \leftrightarrow x_2$



Basisfunktion (0,0)



Basisfunktion (1,1)



Ausgabe

- Die Initialisierung führt bereits zu einer perfekten Lösung der Lernaufgabe.
- Dies ist Zufall, da das lineare Gleichungssystem wegen linear abhängiger Gleichungen nicht überbestimmt ist.

Bestimmung passender Zentren für die RBFs

Ein Ansatz: **k-means-Clustering**

- Wähle k zufällig ausgewählte Trainingsbeispiele als Zentren.
- Weise jedem Zentrum die am nächsten liegenden Trainingsbeispiele zu.
- Berechne neue Zentren als Schwerpunkt der dem Zentrum zugewiesenen Trainingsbeispiele.
- Wiederhole diese zwei Schritte bis zur Konvergenz,
  d.h. bis sich die Zentren nicht mehr ändern.
- Nutze die sich ergebenden Zentren für die Gewichtsvektoren der versteckten Neuronen.

Alternativer Ansatz: **Lernende Vektorquantisierung**

Training von RBF-Netzen:
Herleitung der Update-Regeln ist analog zu der für MLPs.

Gewichte von den versteckten zu den Ausgabeneuronen.

Gradient:

$$\nabla_{\boldsymbol{w}_u} e_u^{(l)} = \frac{\partial e_u^{(l)}}{\partial \boldsymbol{w}_u} = -2\big(o_u^{(l)} - \mathrm{out}_u^{(l)}\big)\,\mathbf{in}_u^{(l)},$$

Gewichtsänderungsregel:

$$\Delta \boldsymbol{w}_u^{(l)} = -\frac{\eta_3}{2} \nabla_{\boldsymbol{w}_u} e_u^{(l)} = \eta_3\big(o_u^{(l)} - \mathrm{out}_u^{(l)}\big)\,\mathbf{in}_u^{(l)}$$

(Zwei weitere Lernraten sind notwendig für die Positionen der Zentren und der Radien.)

Training von RBF-Netzen:
Zentren: (Gewichte von Eingabe- zu versteckten Neuronen).

Gradient:

$$\nabla_{\boldsymbol{w}_v} e^{(l)} = \frac{\partial e^{(l)}}{\partial \boldsymbol{w}_v} = -2 \sum_{s \in \mathrm{succ}(v)} \left(o_s^{(l)} - \mathrm{out}_s^{(l)}\right) w_{su} \frac{\partial \mathrm{out}_v^{(l)}}{\partial \mathrm{net}_v^{(l)}} \frac{\partial \mathrm{net}_v^{(l)}}{\partial \boldsymbol{w}_v}$$

Gewichtsänderungsregel:

$$\Delta \boldsymbol{w}_v^{(l)} = -\frac{\eta_1}{2} \nabla_{\boldsymbol{w}_v} e^{(l)} = \eta_1 \sum_{s \in \mathrm{succ}(v)} \left(o_s^{(l)} - \mathrm{out}_s^{(l)}\right) w_{sv} \frac{\partial \mathrm{out}_v^{(l)}}{\partial \mathrm{net}_v^{(l)}} \frac{\partial \mathrm{net}_v^{(l)}}{\partial \boldsymbol{w}_v}$$

Training von RBF-Netzen:
Zentren: (Gewichte von Eingabe- zu versteckten Neuronen).

Spezialfall: **Euklidischer Abstand**

$$\frac{\partial \operatorname{net}_v^{(l)}}{\partial \boldsymbol{w}_v} = \left(\sum_{i=1}^{n} \left(w_{vp_i} - \operatorname{out}_{p_i}^{(l)} \right)^2 \right)^{-\frac{1}{2}} \left(\boldsymbol{w}_v - \mathbf{in}_v^{(l)} \right).$$

Spezialfall: **Gaußsche Aktivierungsfunktion**

$$\frac{\partial \operatorname{out}_v^{(l)}}{\partial \operatorname{net}_v^{(l)}} = \frac{\partial f_{\text{act}}\left(\operatorname{net}_v^{(l)}, \sigma_v \right)}{\partial \operatorname{net}_v^{(l)}} = \frac{\partial}{\partial \operatorname{net}_v^{(l)}} \, e^{-\frac{\left(\operatorname{net}_v^{(l)} \right)^2}{2\sigma_v^2}} = -\frac{\operatorname{net}_v^{(l)}}{\sigma_v^2} \, e^{-\frac{\left(\operatorname{net}_v^{(l)} \right)^2}{2\sigma_v^2}}.$$

Training von RBF-Netzen:
Radien der radialen Basisfunktionen.

Gradient:

$$\frac{\partial e^{(l)}}{\partial \sigma_v} = -2 \sum_{s \in \mathrm{succ}(v)} \left(o_s^{(l)} - \mathrm{out}_s^{(l)}\right) w_{su} \frac{\partial \, \mathrm{out}_v^{(l)}}{\partial \sigma_v}.$$

Gewichtsänderungsregel:

$$\Delta \sigma_v^{(l)} = -\frac{\eta_2}{2} \frac{\partial e^{(l)}}{\partial \sigma_v} = \eta_2 \sum_{s \in \mathrm{succ}(v)} \left(o_s^{(l)} - \mathrm{out}_s^{(l)}\right) w_{sv} \frac{\partial \, \mathrm{out}_v^{(l)}}{\partial \sigma_v}.$$

Spezialfall: **Gaußsche Aktivierungsfunktion**

$$\frac{\partial \, \mathrm{out}_v^{(l)}}{\partial \sigma_v} = \frac{\partial}{\partial \sigma_v} \, e^{-\frac{\left(\mathrm{net}_v^{(l)}\right)^2}{2\sigma_v^2}} = \frac{\left(\mathrm{net}_v^{(l)}\right)^2}{\sigma_v^3} \, e^{-\frac{\left(\mathrm{net}_v^{(l)}\right)^2}{2\sigma_v^2}}.$$

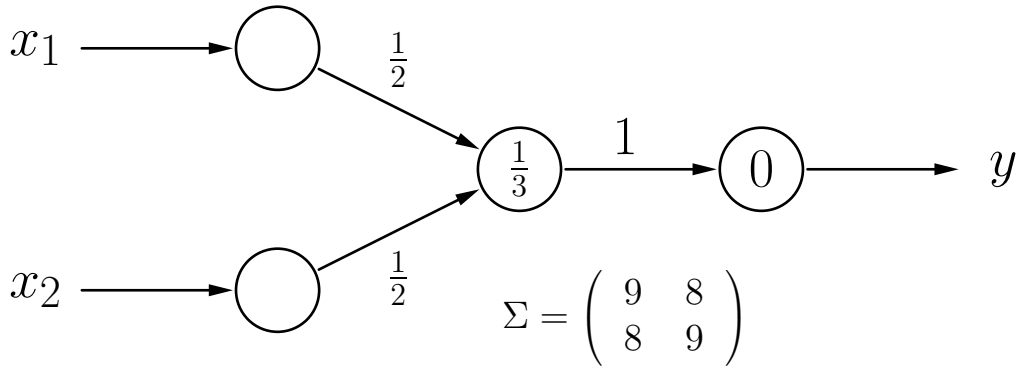
Verallgemeinerung der Abstandsfunktion

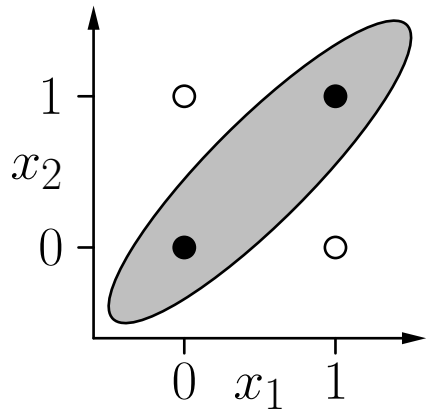
Idee: Benutze anisotrope (richtungsabhängige) Abstandsfunktion.

Beispiel: **Mahalanobis-Abstand**

$$d(\boldsymbol{x}, \boldsymbol{y}) = \sqrt{(\boldsymbol{x} - \boldsymbol{y})^\top \Sigma^{-1} (\boldsymbol{x} - \boldsymbol{y})}.$$

Beispiel: **Biimplikation**





Radiale-Basisfunktionen-Netze: Anwendung

Vorteile

- einfache Feedforward-Architektur
- leichte Anpassbarkeit
- daher schnelle Optimierung und Berechnung

Anwendung

- kontinuierlich laufende Prozesse, die schnelle Anpassung erfordern
- Approximierung
- Mustererkennung
- Regelungstechnik

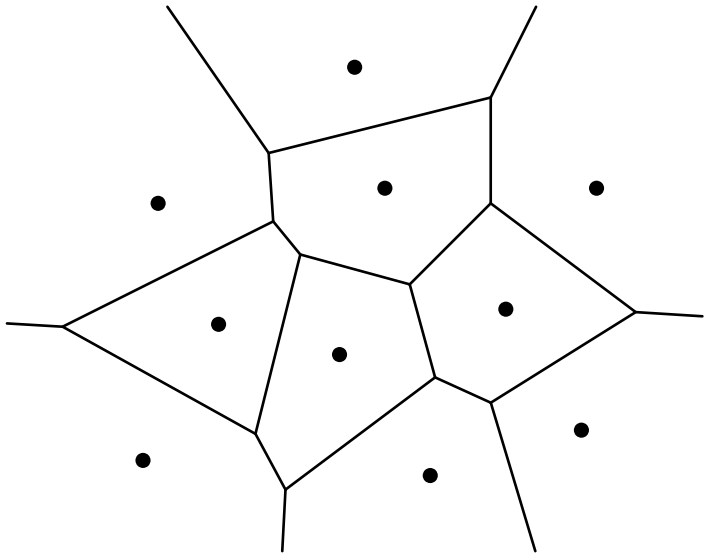
Lernende Vektorquantisierung

(engl. Learning Vector Quantization)

Motivation

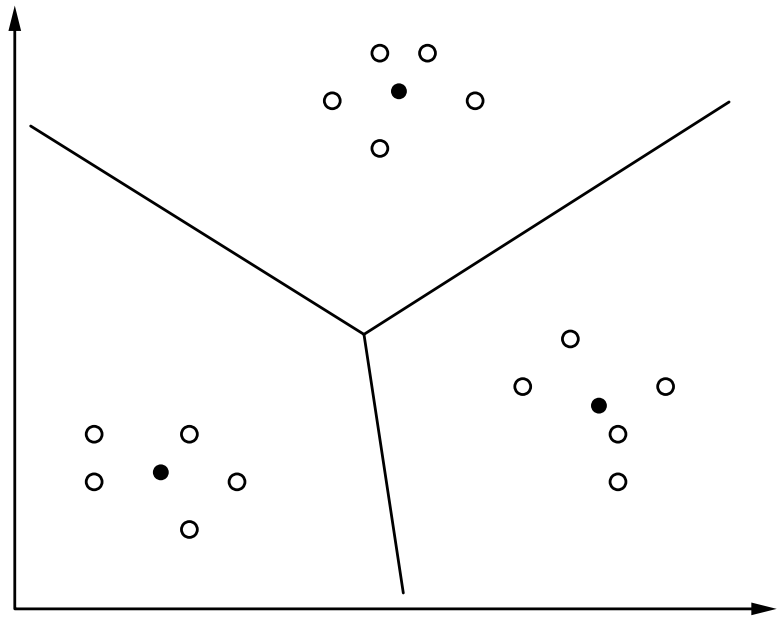
- Bisher: festes Lernen, jetzt freies Lernen, d.h. es existieren keine festgelegten Klassenlabels oder Zielwerte mehr für jedes Lernbeispiel
- Grundidee: ähnliche Eingaben führen zu ähnlichen Ausgaben
- Ähnlichkeit zum Clustering: benachbarte (ähnliche) Datenpunkte im Eingaberaum liegen auch im Ausgaberaum benachbart

Voronoidiagramm einer Vektorquantisierung



- Punkte repräsentieren Vektoren, die zur Quantisierung der Fläche genutzt werden.
- Linien sind die Grenzen der Regionen, deren Punkte am nächsten zu dem dargestellten Vektor liegen.

Finden von Clustern in einer gegebenen Menge von Punkten



- Datenpunkte werden durch leere Kreise dargestellt (○).
- Clusterzentren werden durch gefüllte Kreise dargestellt (●).

Lernende Vektorquantisierung, Netzwerk

Ein **Lernendes Vektorquantisierungsnetzwerk (LVQ)** ist ein neuronales Netz mit einem Graphen $G = (U, C)$ das die folgenden Bedingungen erfüllt:

(i) $U_{\text{in}} \cap U_{\text{out}} = \emptyset$, $U_{\text{hidden}} = \emptyset$

(ii) $C = U_{\text{in}} \times U_{\text{out}}$

Die Netzeingabefunktion jedes Ausgabeneurons ist eine **Abstandsfunktion** zwischen Eingabe- und Gewichtsvektor, d.h.

$$\forall u \in U_{\text{out}}: \qquad f_{\text{net}}^{(u)}(\boldsymbol{w}_u, \mathbf{in}_u) = d(\boldsymbol{w}_u, \mathbf{in}_u),$$

wobei $d : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}_0^+$ eine Funktion ist, die $\forall \boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z} \in \mathbb{R}^n$:

$$\begin{array}{rll} (i) & d(\boldsymbol{x}, \boldsymbol{y}) = 0 \quad \Leftrightarrow \quad \boldsymbol{x} = \boldsymbol{y}, & \\ (ii) & d(\boldsymbol{x}, \boldsymbol{y}) = d(\boldsymbol{y}, \boldsymbol{x}) & \text{(Symmetrie)}, \\ (iii) & d(\boldsymbol{x}, \boldsymbol{z}) \leq d(\boldsymbol{x}, \boldsymbol{y}) + d(\boldsymbol{y}, \boldsymbol{z}) & \text{(Dreiecksungleichung)} \end{array}$$

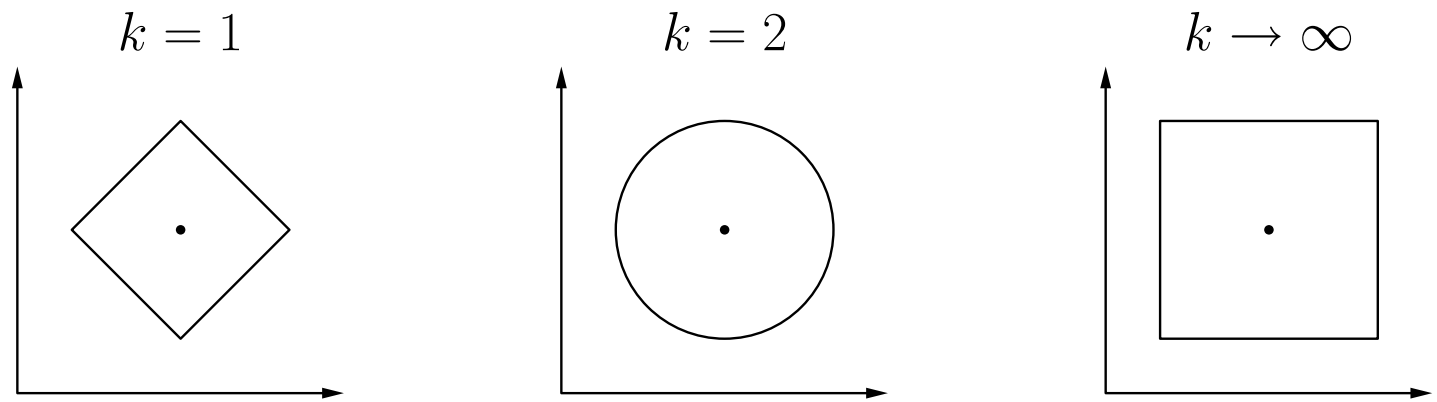
erfüllt.

Veranschaulichung von Abstandsfunktionen

$$d_k(\boldsymbol{x}, \boldsymbol{y}) = \left(\sum_{i=1}^{n} (x_i - y_i)^k \right)^{\frac{1}{k}}$$

Bekannte Spezialfälle:

$k = 1:$	Manhattan- oder City-Block-Abstand,
$k = 2:$	Euklidischer Abstand,
$k \to \infty:$	Maximum-Abstand, d.h. $d_\infty(\boldsymbol{x}, \boldsymbol{y}) = \max_{i=1}^{n} \lvert x_i - y_i \rvert$.



(alle Punkte auf dem Kreis bzw. den Vierecken haben denselben Abstand zum Mittelpunkt, entsprechend der jeweiligen Abstandsfunktion)

Die Aktivierungsfunktion jedes Ausgabeneurons ist eine sogenannte **radiale Funktion**, d.h. eine monoton fallende Funktion

$$f : \mathbb{R}_0^+ \to [0, \infty] \quad \text{with} \quad f(0) = 1 \quad \text{and} \quad \lim_{x \to \infty} f(x) = 0.$$

Manchmal wird der Wertebereich auf das Intervall $[0, 1]$ beschränkt.
Durch die spezielle Ausgabefunktion ist das allerdings unerheblich.

Die Ausgabefunktion jedes Ausgabeneurons ist keine einfache Funktion der Aktivierung des Neurons. Sie zieht stattdessen alle Aktivierungen aller Ausgabeneuronen in Betracht:

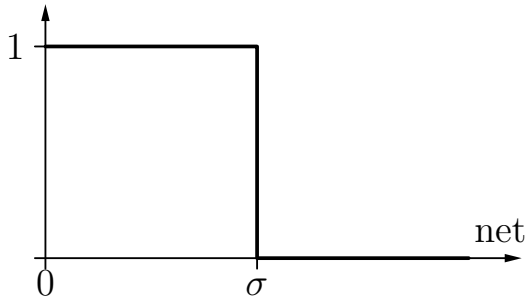
$$f_{\text{out}}^{(u)}(\text{act}_u) = \begin{cases} 1, & \text{falls } \text{act}_u = \max\limits_{v \in U_{\text{out}}} \text{act}_v, \\ 0, & \text{sonst.} \end{cases}$$

Sollte mehr als ein Neuron die maximale Aktivierung haben, wird ein zufällig gewähltes Neuron auf die Ausgabe 1 gesetzt, alle anderen auf Ausgabe 0: **Winner-Takes-All-Prinzip**.

Radiale Aktivierungfunktionen

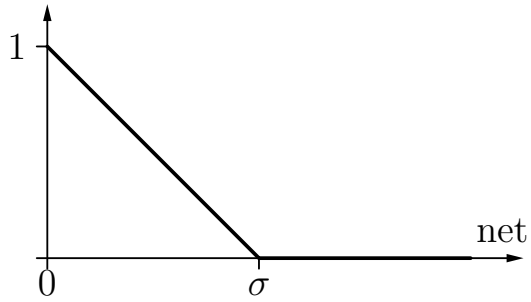
Rechteckfunktion:

$$f_{\text{act}}(\text{net}, \sigma) = \begin{cases} 0, & \text{falls net} > \sigma, \\ 1, & \text{sonst.} \end{cases}$$



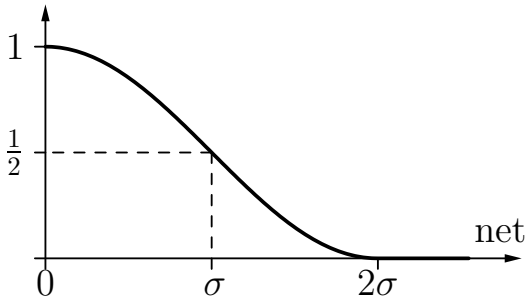
Dreiecksfunktion:

$$f_{\text{act}}(\text{net}, \sigma) = \begin{cases} 0, & \text{falls net} > \sigma, \\ 1 - \frac{\text{net}}{\sigma}, & \text{sonst.} \end{cases}$$



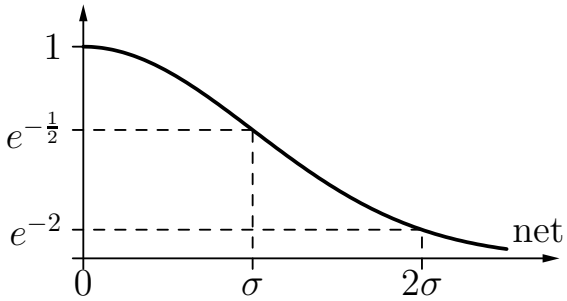
Kosinus bis Null:

$$f_{\text{act}}(\text{net}, \sigma) = \begin{cases} 0, & \text{falls net} > 2\sigma, \\ \frac{\cos\left(\frac{\pi}{2\sigma}\,\text{net}\right)+1}{2}, & \text{sonst.} \end{cases}$$



Gauß-Funktion:

$$f_{\text{act}}(\text{net}, \sigma) = e^{-\frac{\text{net}^2}{2\sigma^2}}$$



Anpassung der Referenzvektoren (Codebuch-Vektoren)

- Bestimme zu jedem Trainingsbeispiel den nächsten Referenzvektor.
- Passe nur diesen Referenzvektor an (Gewinnerneuron).
- Für Klassifikationsprobleme kann die Klasse genutzt werden:
  Jeder Referenzvektor wird einer Klasse zugeordnet.

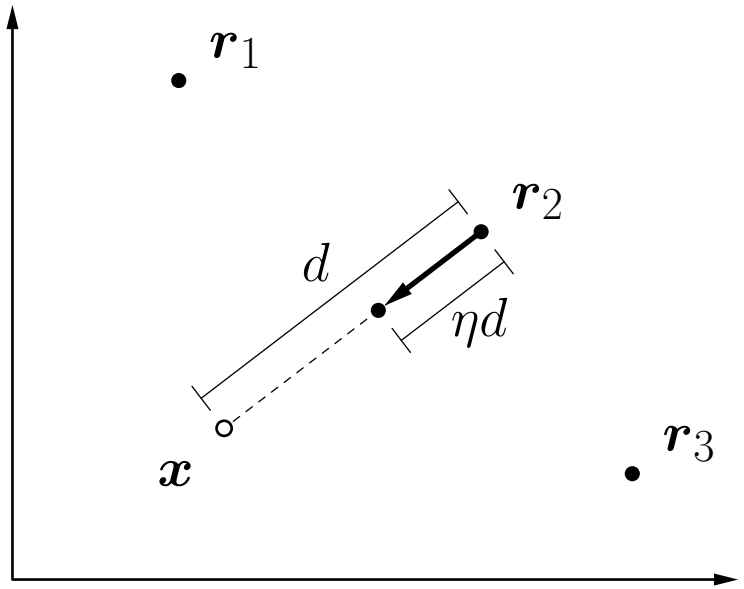
Anziehungsregel (Datenpunkt und Referenzvektor haben dieselbe Klasse)

$$\boldsymbol{r}^{\,(\text{new})} = \boldsymbol{r}^{\,(\text{old})} + \eta\big(\boldsymbol{x} - \boldsymbol{r}^{\,(\text{old})}\big),$$

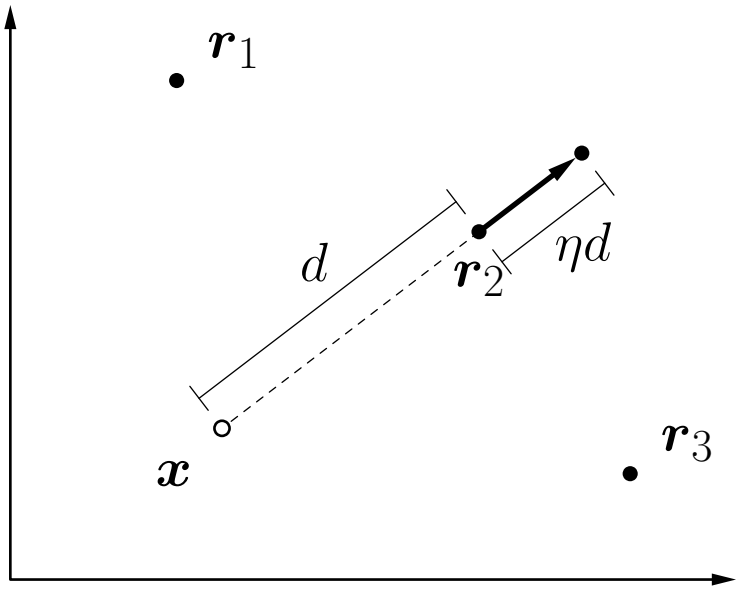
Abstoßungsregel (Datenpunkt und Referenzvektor haben verschiedene Klassen)

$$\boldsymbol{r}^{\,(\text{new})} = \boldsymbol{r}^{\,(\text{old})} - \eta\big(\boldsymbol{x} - \boldsymbol{r}^{\,(\text{old})}\big).$$

Anpassung der Referenzvektoren



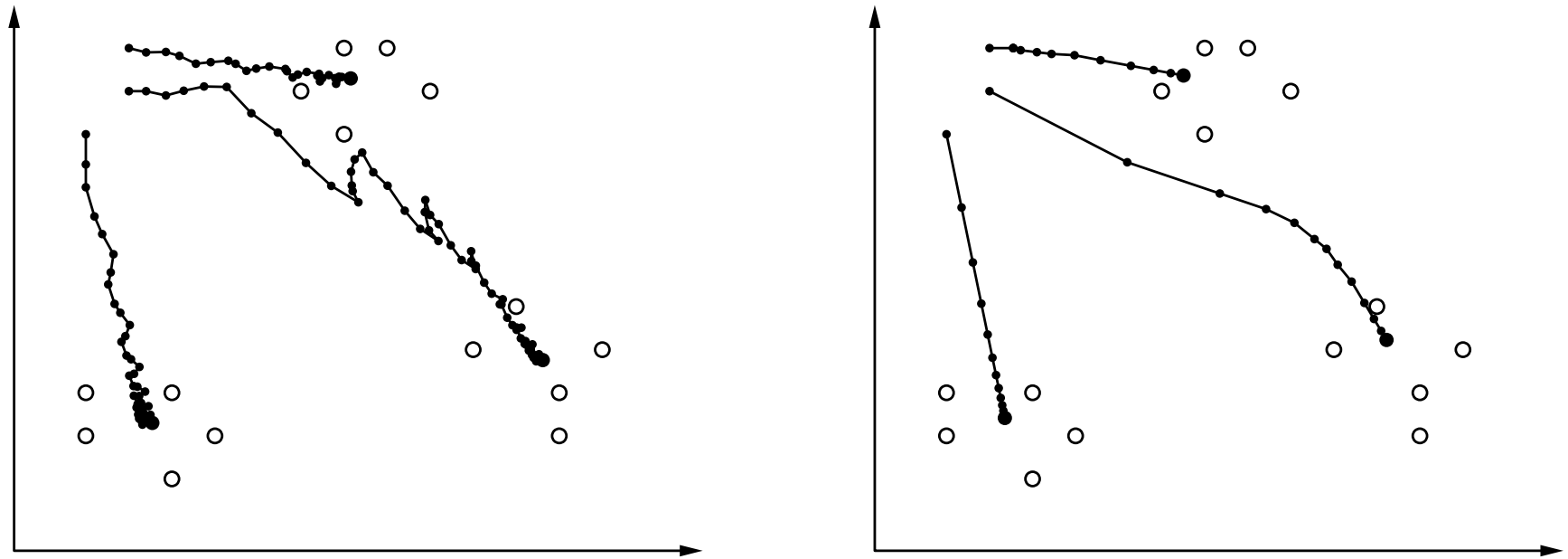
Anziehungsregel



Abstoßungsregel

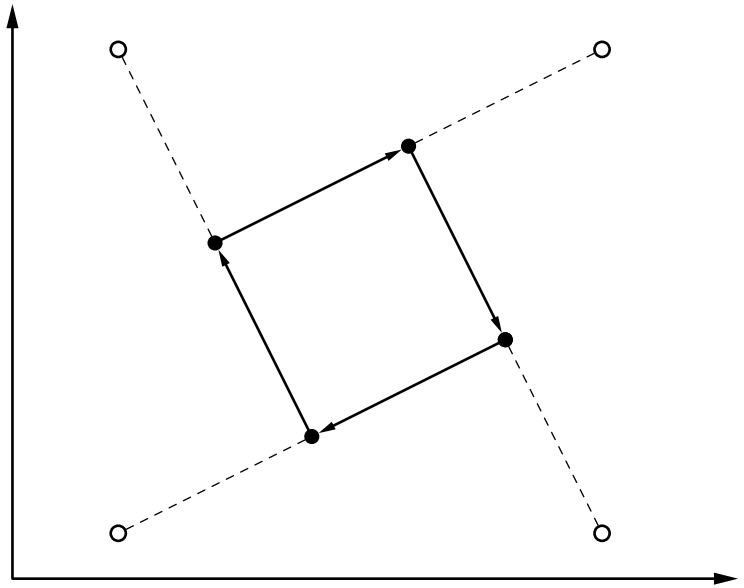
- $\boldsymbol{x}$: Datenpunkt, $\boldsymbol{r}_i$: Referenzvektor
- $\eta = 0.4$ (Lernrate)

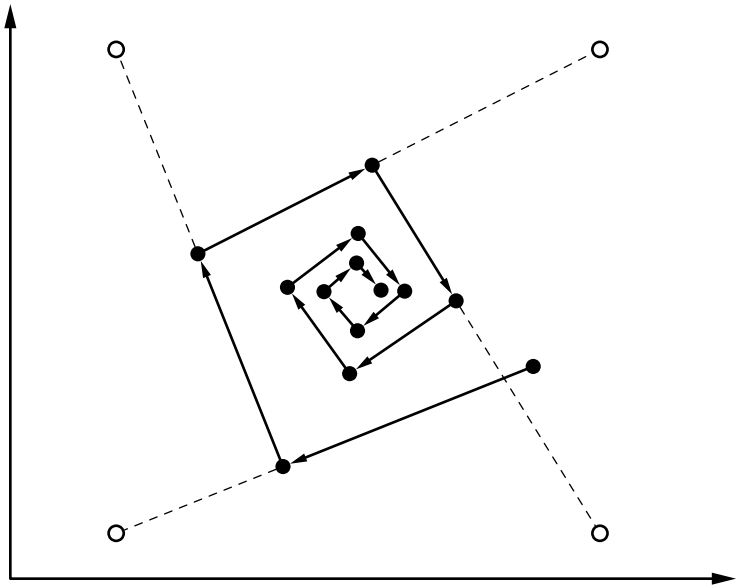
Anpassung der Referenzvektoren



- Links: Online-Training mit Lernrate $\eta = 0.1$,
- Rechts: Batch-Training mit Lernrate $\eta = 0.05$.

Problem: feste Lernrate kann zu Oszillationen führen





Lösung: **zeitabhängige Lernrate**

$$\eta(t) = \eta_0 \alpha^t, \quad 0 < \alpha < 1, \qquad \text{oder} \qquad \eta(t) = \eta_0 t^\kappa, \quad \kappa > 0.$$

Verbesserte Anpassungsregel für klassifizierte Daten

- **Idee:** Passe nicht nur den Referenzvektor an, der am nächsten zum Datenpunkt liegt (das Gewinnerneuron), sondern passe **die zwei nächstliegenden Referenzvektoren**.

- Sei $\boldsymbol{x}$ der momentan bearbeitete Datenpunkt und c seine Klasse.
  Seien $\boldsymbol{r}_j$ und $\boldsymbol{r}_k$ die zwei nächstliegenden Referenzvektoren und z_j sowie z_k ihre Klassen.

- Referenzvektoren werden nur angepasst, wenn $z_j \neq z_k$ und entweder $c = z_j$ oder $c = z_k$. (o.B.d.A. nehmen wir an: $c = z_j$.)

  Die **Anpassungsregeln** für die zwei nächstgelegenen Referenzvektoren sind:

$$\begin{aligned} \boldsymbol{r}_j^{(\text{new})} &= \boldsymbol{r}_j^{(\text{old})} + \eta\big(\boldsymbol{x} - \boldsymbol{r}_j^{(\text{old})}\big) \qquad \text{and} \\ \boldsymbol{r}_k^{(\text{new})} &= \boldsymbol{r}_k^{(\text{old})} - \eta\big(\boldsymbol{x} - \boldsymbol{r}_k^{(\text{old})}\big), \end{aligned}$$

  wobei alle anderen Referenzvektoren unverändert bleiben.

Lernende Vektorquantisierung: "Window Rule"

- In praktischen Experimenten wurde beobachtet, dass LVQ in der Standardausführung die Referenzvektoren immer weiter voneinander wegtreibt.
- Um diesem Verhalten entgegenzuwirken, wurde die **window rule** eingeführt: passe nur dann an, wenn der Datenpunkt $\boldsymbol{x}$ in der Nähe der Klassifikationsgrenze liegt.
- "In der Nähe der Grenze" wird formalisiert durch folgende Bedingung:

$$\min\left(\frac{d(\boldsymbol{x},\boldsymbol{r}_j)}{d(\boldsymbol{x},\boldsymbol{r}_k)},\frac{d(\boldsymbol{x},\boldsymbol{r}_k)}{d(\boldsymbol{x},\boldsymbol{r}_j)}\right) > \theta, \qquad \text{wobei} \qquad \theta = \frac{1-\xi}{1+\xi}.$$

  ξ ist ein Parameter, der vom Benutzer eingestellt werden muss.
- Intuitiv beschreibt ξ die "Größe" des Fensters um die Klassifikationsgrenze, in dem der Datenpunkt liegen muss, um zu einer Anpassung zu führen.
- Damit wird die Divergenz vermieden, da die Anpassung eines Referenzvektors nicht mehr durchgeführt wird, wenn die Klassifikationsgrenze weit genug weg ist.

Soft LVQ

Idee: Benutze weiche Zuordnungen anstelle von *winner-takes-all*.

Annahme: Die Daten wurden aus einer Mischung von Normalverteilungen gezogen. Jeder Referenzvektor beschreibt eine Normalverteilung.

Ziel: Maximiere das Log-Wahrscheinlichkeitsverhältnis der Daten, also

$$\begin{aligned}\ln L_{\text{ratio}} = &\sum_{j=1}^{n} \ln \sum_{\boldsymbol{r} \in R(c_j)} \exp\left(-\frac{(\boldsymbol{x}_j - \boldsymbol{r})^\top (\boldsymbol{x}_j - \boldsymbol{r})}{2\sigma^2}\right) \\ &- \sum_{j=1}^{n} \ln \sum_{\boldsymbol{r} \in Q(c_j)} \exp\left(-\frac{(\boldsymbol{x}_j - \boldsymbol{r})^\top (\boldsymbol{x}_j - \boldsymbol{r})}{2\sigma^2}\right).\end{aligned}$$

Hierbei ist σ ein Parameter, der die "Größe" jeder Normalverteilung angibt.
$R(c)$ ist die Menge der Referenzvektoren der Klasse c und $Q(c)$ deren Komplement.

Intuitiv: für jeden Datenpunkt sollte die Wahrscheinlichkeitsdichte für seine Klasse so groß wie möglich sein, während die Dichte für alle anderen Klassen so klein wie möglich sein sollte.

Anpassungsregel abgeleitet aus Maximum-Log-Likelihood-Ansatz:

$$\boldsymbol{r}_i^{(\text{new})} = \boldsymbol{r}_i^{(\text{old})} + \eta \cdot \begin{cases} u_{ij}^{\oplus} \cdot \left(\boldsymbol{x}_j - \boldsymbol{r}_i^{(\text{old})}\right), & \text{if } c_j = z_i, \\ -u_{ij}^{\ominus} \cdot \left(\boldsymbol{x}_j - \boldsymbol{r}_i^{(\text{old})}\right), & \text{if } c_j \neq z_i, \end{cases}$$

wobei z_i die dem Referenzvektor $\boldsymbol{r}_i$ zugehörige Klasse ist und

$$u_{ij}^{\oplus} = \frac{\exp\left(-\frac{1}{2\sigma^2}\left(\boldsymbol{x}_j - \boldsymbol{r}_i^{(\text{old})}\right)^\top \left(\boldsymbol{x}_j - \boldsymbol{r}_i^{(\text{old})}\right)\right)}{\sum\limits_{\boldsymbol{r} \in R(c_j)} \exp\left(-\frac{1}{2\sigma^2}\left(\boldsymbol{x}_j - \boldsymbol{r}^{(\text{old})}\right)^\top \left(\boldsymbol{x}_j - \boldsymbol{r}^{(\text{old})}\right)\right)} \quad \text{and}$$

$$u_{ij}^{\ominus} = \frac{\exp\left(-\frac{1}{2\sigma^2}\left(\boldsymbol{x}_j - \boldsymbol{r}_i^{(\text{old})}\right)^\top \left(\boldsymbol{x}_j - \boldsymbol{r}_i^{(\text{old})}\right)\right)}{\sum\limits_{\boldsymbol{r} \in Q(c_j)} \exp\left(-\frac{1}{2\sigma^2}\left(\boldsymbol{x}_j - \boldsymbol{r}^{(\text{old})}\right)^\top \left(\boldsymbol{x}_j - \boldsymbol{r}^{(\text{old})}\right)\right)}.$$

$R(c)$ ist die Menge der Referenzvektoren, die zu Klasse c gehören und $Q(c)$ ist deren Komplement.

Hard LVQ

Idee: Leite ein Schema mit scharfen Zuordnungen aus der unscharfen Version ab.

Ansatz: Lasse den Größenparameter σ der Gaußfunktion gegen Null streben.

Die sich ergebende Anpassungsregel ist somit:

$$\boldsymbol{r}_i^{(\text{new})} = \boldsymbol{r}_i^{(\text{old})} + \eta \cdot \begin{cases} u_{ij}^{\oplus} \cdot (\boldsymbol{x}_j - \boldsymbol{r}_i^{(\text{old})}), & \text{if } c_j = z_i, \\ -u_{ij}^{\ominus} \cdot (\boldsymbol{x}_j - \boldsymbol{r}_i^{(\text{old})}), & \text{if } c_j \neq z_i, \end{cases}$$

wobei

$$u_{ij}^{\oplus} = \begin{cases} 1, & \text{falls } \boldsymbol{r}_i = \underset{\boldsymbol{r} \in R(c_j)}{\operatorname{argmin}} d(\boldsymbol{x}_j, \boldsymbol{r}), \\ 0, & \text{sonst,} \end{cases} \qquad u_{ij}^{\ominus} = \begin{cases} 1, & \text{falls } \boldsymbol{r}_i = \underset{\boldsymbol{r} \in Q(c_j)}{\operatorname{argmin}} d(\boldsymbol{x}_j, \boldsymbol{r}), \\ 0, & \text{sonst.} \end{cases}$$

$\boldsymbol{r}_i$ ist der nächstgelegene Vektor derselben Klasse

$\boldsymbol{r}_i$ ist der nächstgelegene Vektor einer anderen Klasse

Diese Anpassungsregel ist stabil, ohne dass eine *window rule* die Anpassung beschränken müsste.

Lernende Vektorquantisierung: Erweiterungen

- **Frequency Sensitive Competitive Learning**
  - Der Abstand zu einem Referenzvektor wird modifiziert, indem berücksichtigt wird, wieviele Datenpunkte diesem Referenzvektor zugewiesen sind.
- **Fuzzy LVQ**
  - Nutzt die enge Verwandschaft zum Fuzzy-Clustering aus.
  - Kann als Online-Version des Fuzzy-Clustering angesehen werden.
  - Führt zu schnellerem Clustering.
- **Größen- und Formparameter**
  - Weise jedem Referenzvektor einen Clusterradius zu.
    Passe diesen Radius in Abhängigkeit von der Nähe der Datenpunkte an.
  - Weise jedem Referenzvektor eine Kovarianzmatrix zu.
    Passe diese Matrix abhängig von der Verteilung der Datenpunkte an.

Selbstorganisierende Karten

(engl. Self-Organizing Maps (SOMs))

Eine **selbstorganisierende Karte** oder **Kohonen-Merkmalskarte** ist ein neuronales Netz mit einem Graphen $G = (U, C)$ das folgende Bedingungen erfüllt:

(i) $U_{\text{hidden}} = \emptyset$, $U_{\text{in}} \cap U_{\text{out}} = \emptyset$,

(ii) $C = U_{\text{in}} \times U_{\text{out}}$.

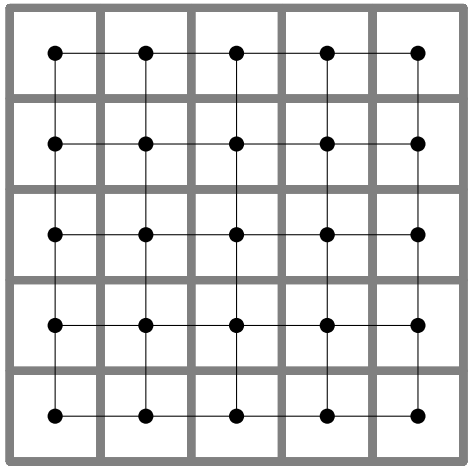
Die Netzeingabefunktion jedes Ausgabeneurons ist eine **Abstandsfunktion** zwischen Eingabe- und Gewichtsvektor. Die Aktivierungsfunktion jedes Ausgabeneurons ist eine **radiale Funktion**, d.h. eine monoton fallende Funktion

$$f : \mathbb{R}_0^+ \to [0, 1] \quad \text{with} \quad f(0) = 1 \quad \text{and} \quad \lim_{x \to \infty} f(x) = 0.$$

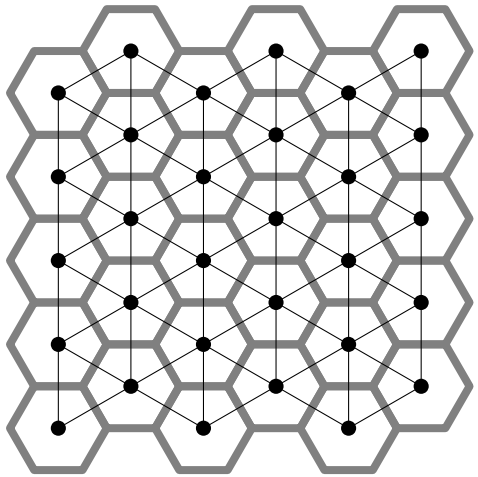
Die Ausgabefunktion jedes Ausgabeneurons ist die Identität.
Die Ausgabe wird oft per "**winner takes all**"-Prinzip diskretisiert.
Auf den Ausgabeneuronen ist eine **Nachbarschaftsbeziehung** definiert:

$$d_{\text{neurons}} : U_{\text{out}} \times U_{\text{out}} \to \mathbb{R}_0^+ .$$

Nachbarschaft der Ausgabeneuronen: Neuronen bilden ein Gitter



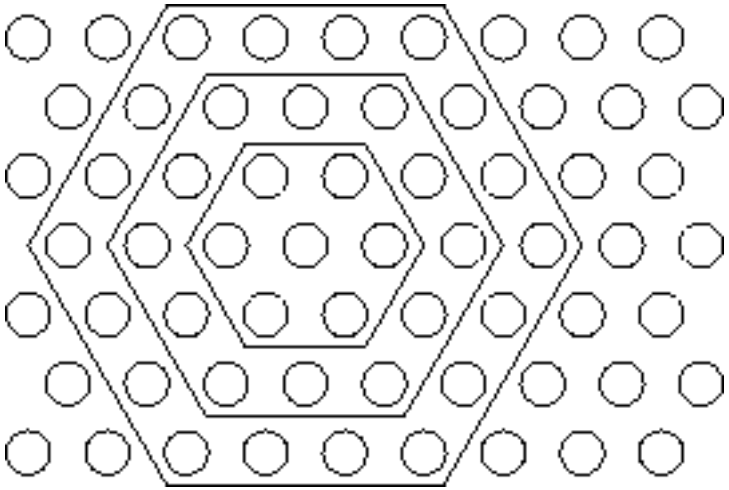
quadratisches Gitter

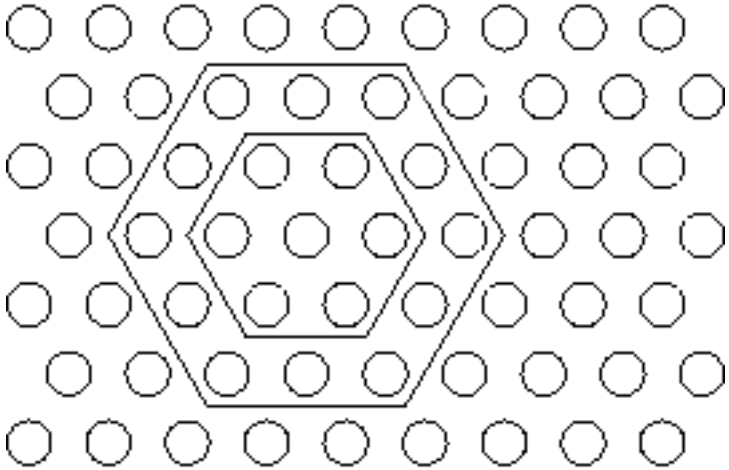


hexagonales Gitter

- Dünne schwarze Linien: Zeigen nächste Nachbarn eines Neurons.
- Dicke graue Linien: Zeigen Regionen, die einem Neuron zugewiesen sind.

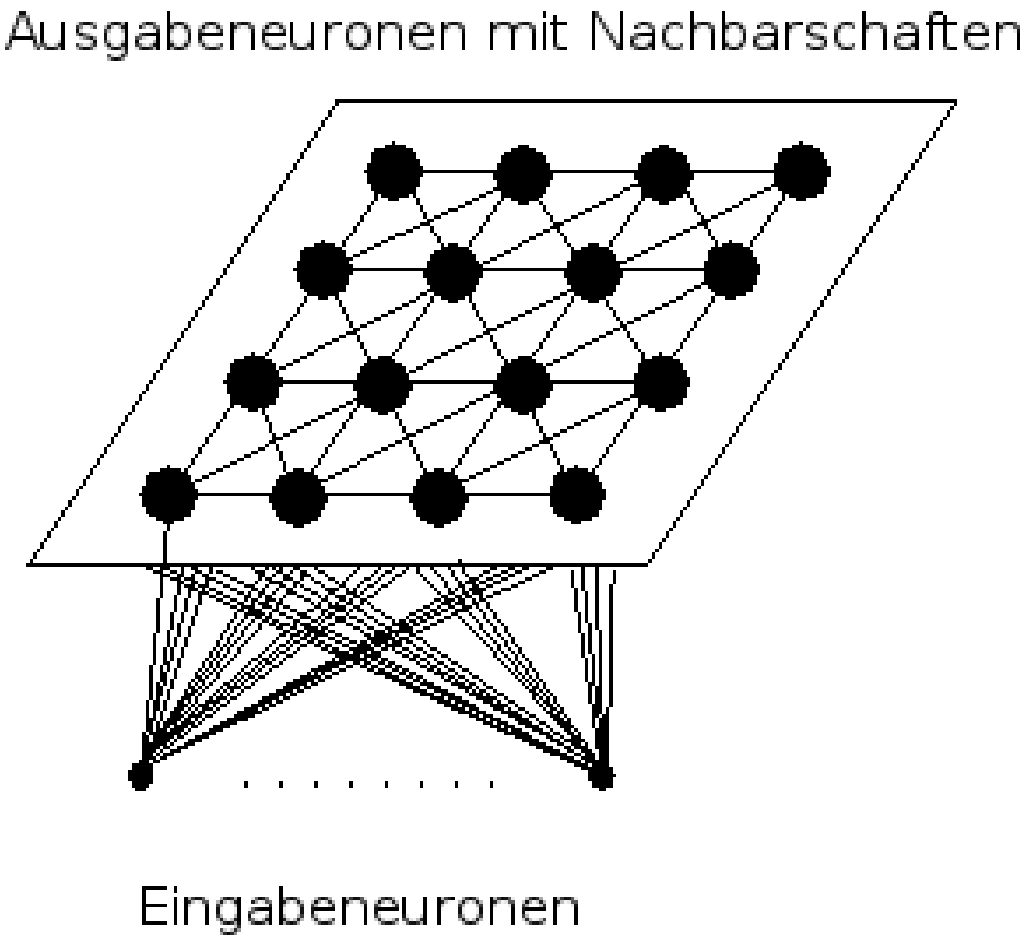
Nachbarschaft des Gewinnerneurons





Der Nachbarschaftsradius wird im Laufe des Lernens kleiner.

Die “Karte” stellt die Ausgabeneuronen mit deren Nachbarschaften dar.



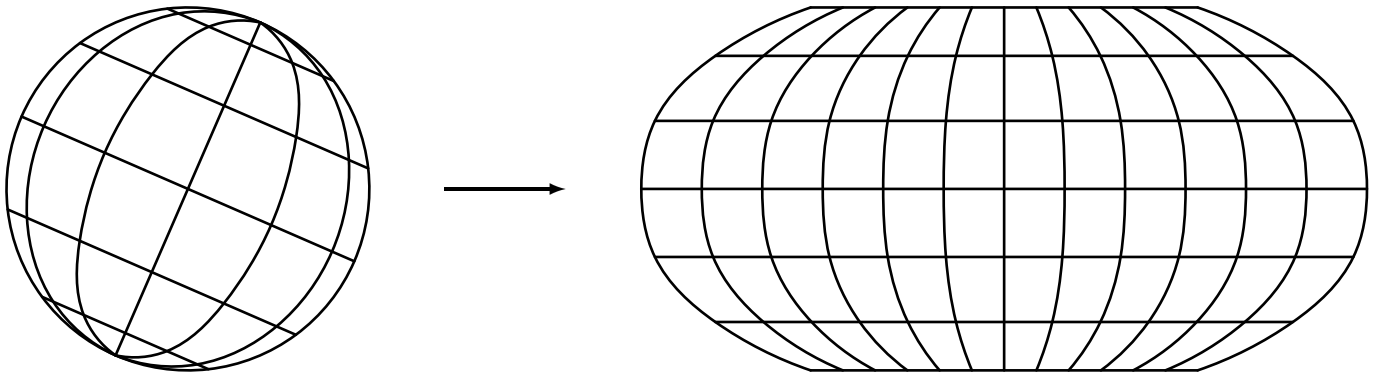
Ablauf des SOM-Lernens

1. Initialisierung der Gewichtsvektoren der Karte
2. zufällige Wahl des Eingabevektors aus der Trainingsmenge
3. Bestimmung des Gewinnerneurons über Abstandsfunktion
4. Bestimmung des zeitabhängigen Radius und der im Radius liegenden Nachbarschaftsneuronen des Gewinners
5. Zeitabhängige Anpassung dieser Nachbarschaftsneuronen, weiter bei 2.

Topologieerhaltende Abbildung

Abbildungen von Punkten, die im Originalraum nah beieinander sind, sollen im Bildraum ebenfalls nah beieinander sein.

Beispiel: **Robinson-Projektion** der Oberfläche einer Kugel



- Die Robinson-Projektion wird häufig für Weltkarten genutzt.
- $\rightarrow$ eine SOM realisiert eine topologieerhaltende Abbildung.

Finde topologieerhaltende Abbildung durch Beachtung der Nachbarschaft

Anpassungsregel für Referenzvektor:

$$\boldsymbol{r}_u^{(\text{new})} = \boldsymbol{r}_u^{(\text{old})} + \eta(t) \cdot f_{\text{nb}}\big(d_{\text{neurons}}(u, u_*), \varrho(t)\big) \cdot \big(\boldsymbol{x} - \boldsymbol{r}_u^{(\text{old})}\big),$$

- u_* ist das Gewinnerneuron (Referenzvektor am nächsten zum Datenpunkt).
- Die Funktion f_{nb} ist eine radiale Funktion.

Zeitabhängige Lernrate

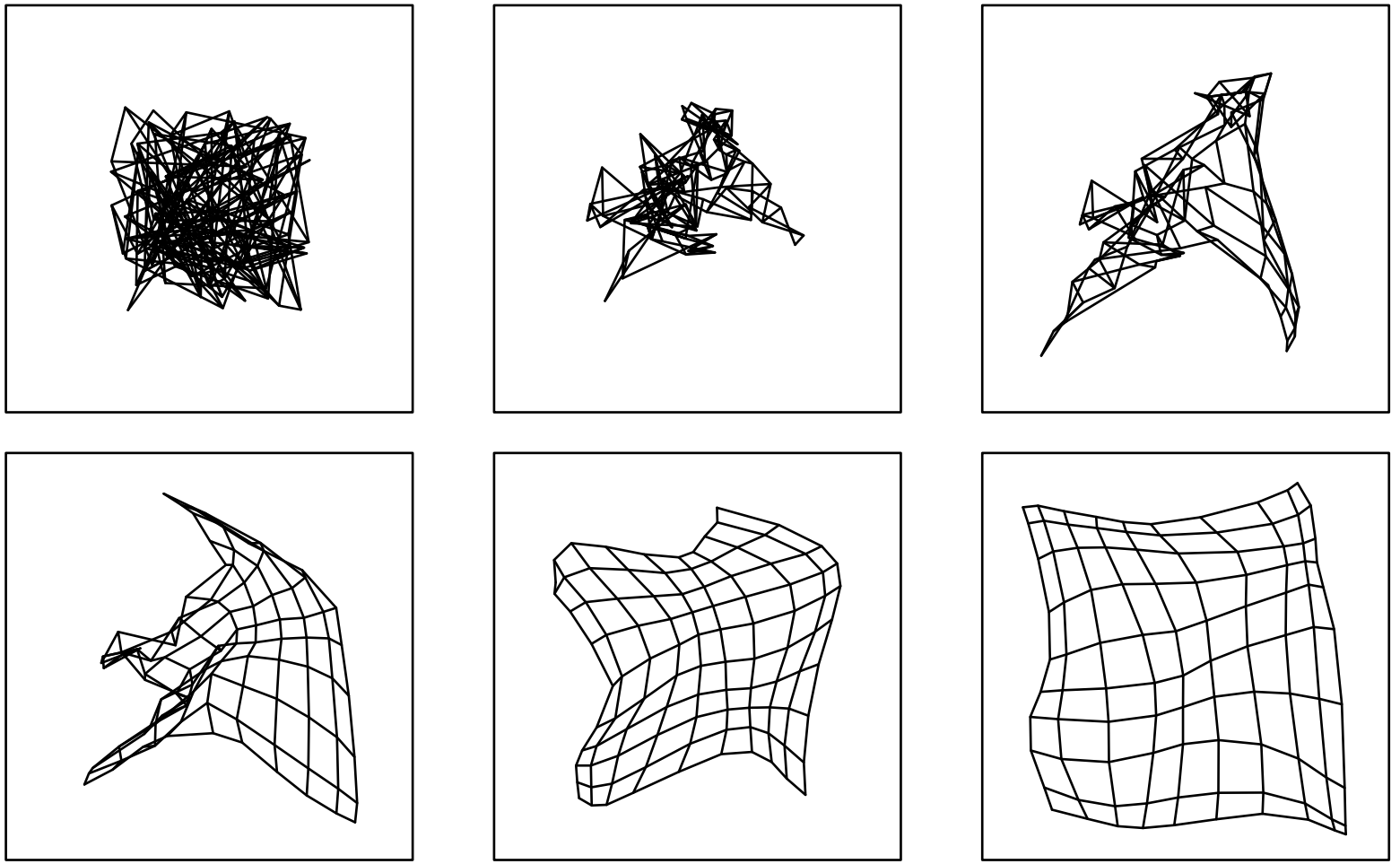
$$\eta(t) = \eta_0 \alpha_\eta^t, \quad 0 < \alpha_\eta < 1, \qquad \text{or} \qquad \eta(t) = \eta_0 t^{\kappa_\eta}, \quad \kappa_\eta > 0.$$

Zeitabhängiger Nachbarschaftsradius

$$\varrho(t) = \varrho_0 \alpha_\varrho^t, \quad 0 < \alpha_\varrho < 1, \qquad \text{or} \qquad \varrho(t) = \varrho_0 t^{\kappa_\varrho}, \quad \kappa_\varrho > 0.$$

Selbstorganisierende Karten: Beispiele

Beispiel: Entfalten einer zweidimensionalen SOM



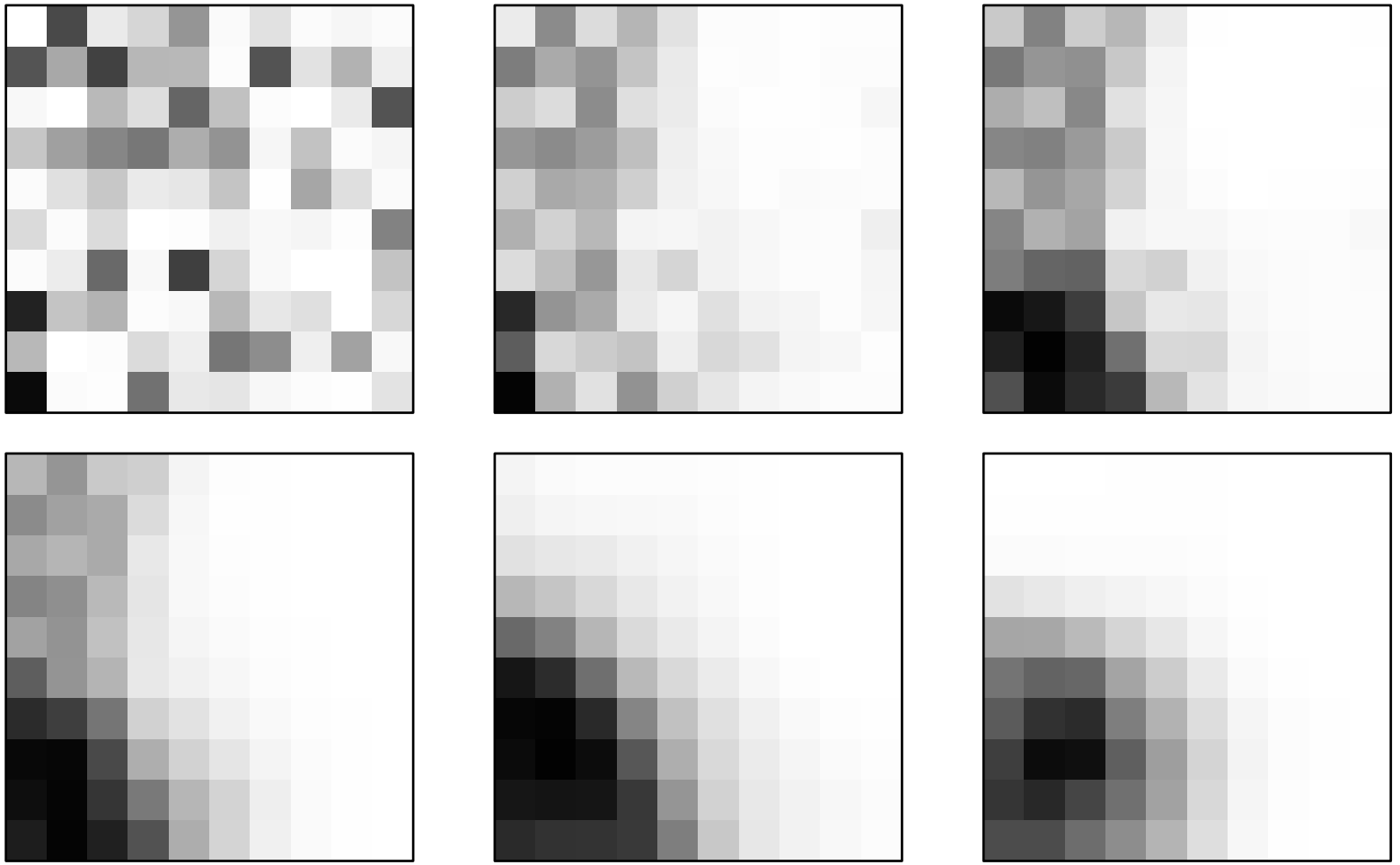
Selbstorganisierende Karten: Beispiele

Beispiel: Entfalten einer zweidimensionalen SOM (Erläuterungen)

- Entfaltung einer 10x10-Karte, die mit zufälligen Mustern aus $[-1, 1] \times [-1, 1]$ trainiert wird
- Initialisierung mit Referenzvektoren aus $[-0.5, 0.5]$
- Linien verbinden direkte Nachbarn (Gitter/Grid)
- Lernrate $\eta(t) = 0.6 * t$
- Gaußsche Nachbarschaftsfunktion f_{nb}
- Radius $\rho(t) = 2.5 * t^{-0.1}$

Selbstorganisierende Karten: Beispiele

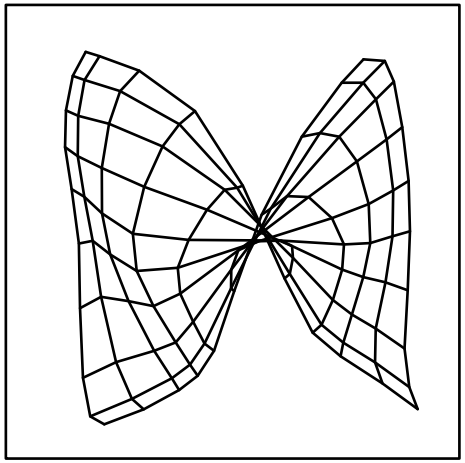
Beispiel: Entfalten einer zweidimensionalen SOM



Einfärbungen der Trainingsstufen der SOM von der vorherigen Folie für das Eingabemuster $(-0.5, -0.5)$ unter Verwendung einer Gaußschen Aktivierungsfunktion.

Selbstorganisierende Karten: Beispiele

Beispiel: Entfalten einer zweidimensionalen SOM

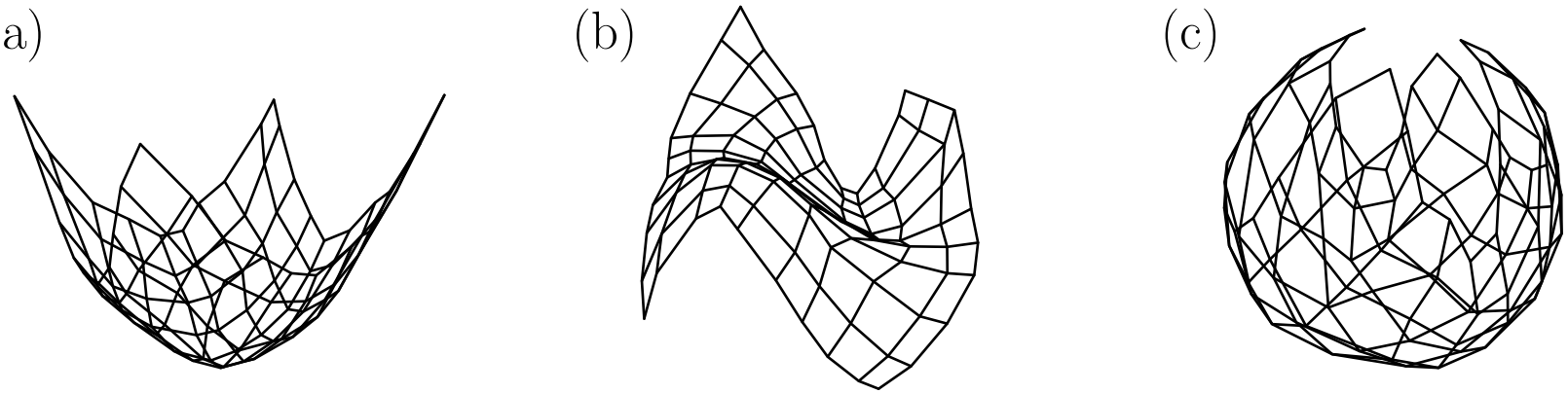


Das Trainieren einer SOM kann u.a. fehlschlagen, falls

- die Initialisierung ungünstig ist oder
- die (anfängliche) Lernrate zu klein gewählt ist oder
- die (anfängliche) Nachbarschaft zu klein gewählt ist.

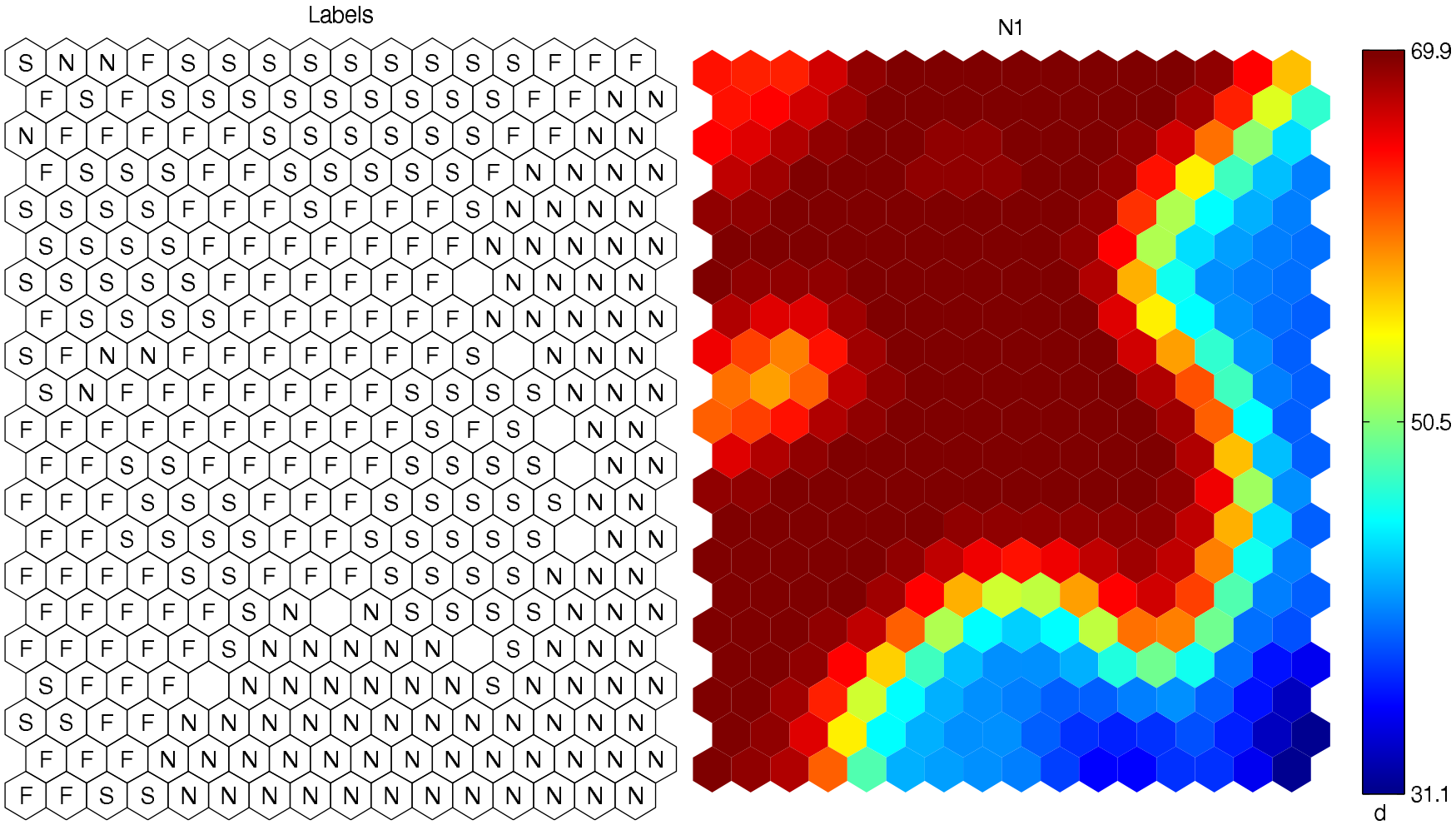
Selbstorganisierende Karten: Beispiele

Beispiel: Entfalten einer zweidimensionalen SOM, Dimensionsreduktion



- Als Lernstichprobe werden zufällige Punkte der Oberfläche einer Rotationsparabel (bzw. kubische Funktion, Kugel) gewählt, also drei Eingabeneuronen (x,y,z-Koordinaten).
- Eine Karte mit 10×10 Ausgabeneuronen wird trainiert.
- Die 3D-Referenzvektoren der Ausgabeneuronen (mit Gitter) werden dargestellt.
- Wegen 2D-Fläche (gekrümmt) klappt die Anpassung sehr gut.
- In diesen Fällen haben Originalraum und Bildraum unterschiedliche Dimensionen.
- Selbstorganisierende Karten können zur Dimensionsreduktion genutzt werden.

SOM, Beispiel Clustering von Feldbearbeitungsstrategien



Links: selbstorganisierende Karte mit eingezeichneten Klassenlabels
Rechts: eine der zum Lernen der Karte verwendeten Variablen mit Farbskala, dargestellt auf der gelernten Karte

SOM, Phonemkarte des Finnischen

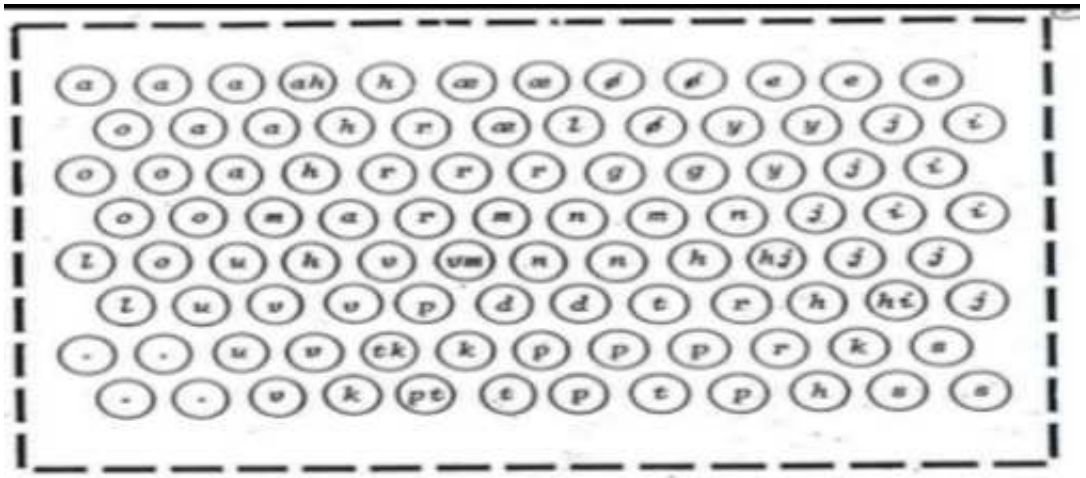


Abb. 2.6.7 Phonemkarte des Finnischen (nach [KOH88])

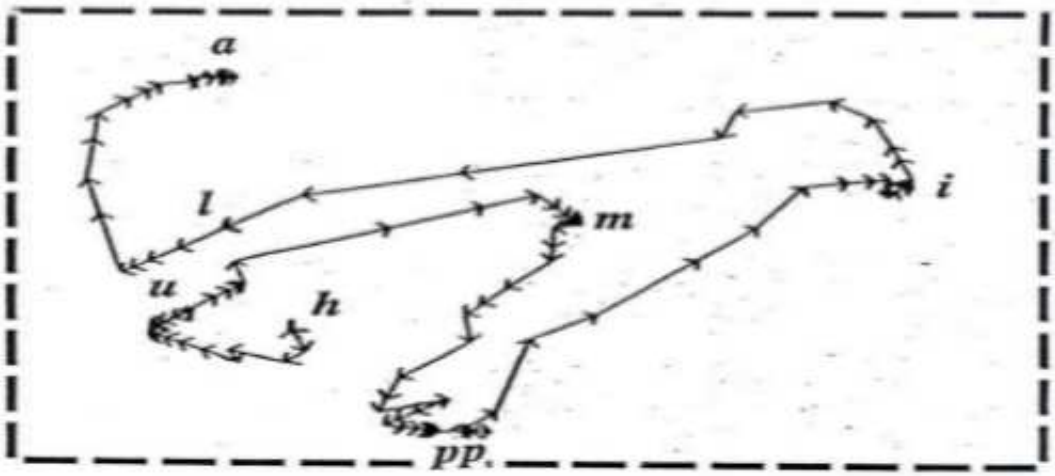
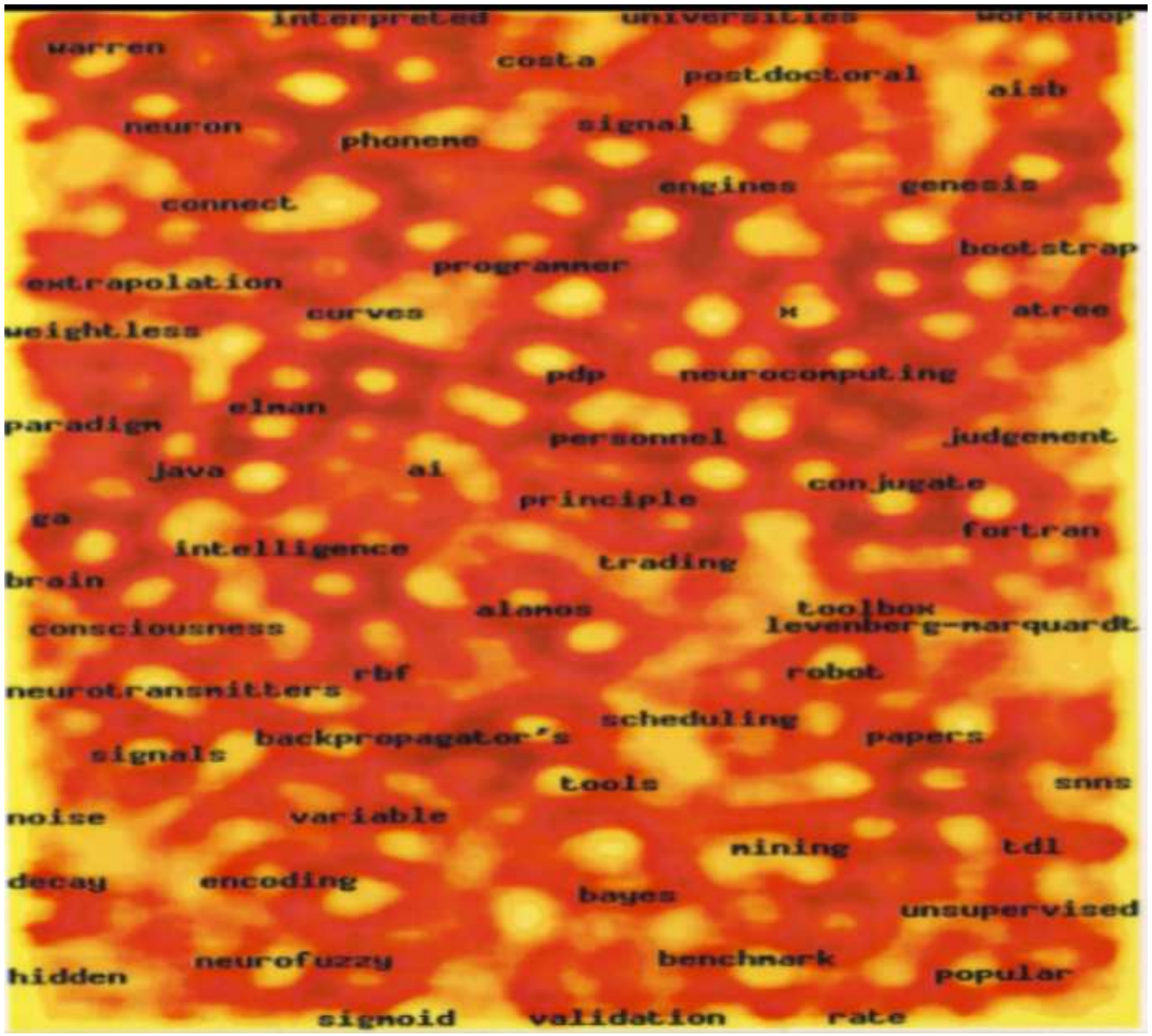
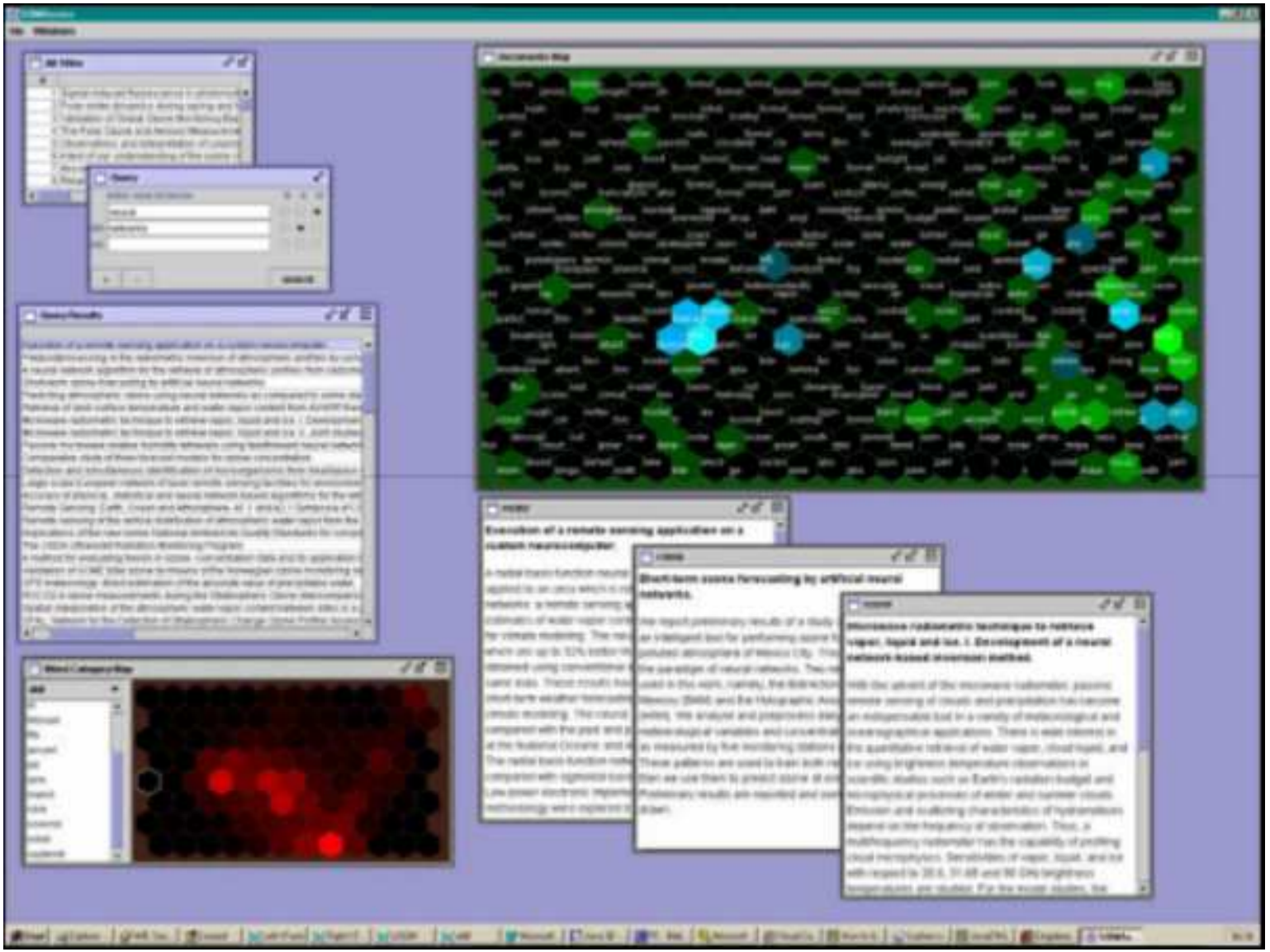


Abb. 2.6.8 Phonemsequenz für */humppila/* (nach [KOH88])

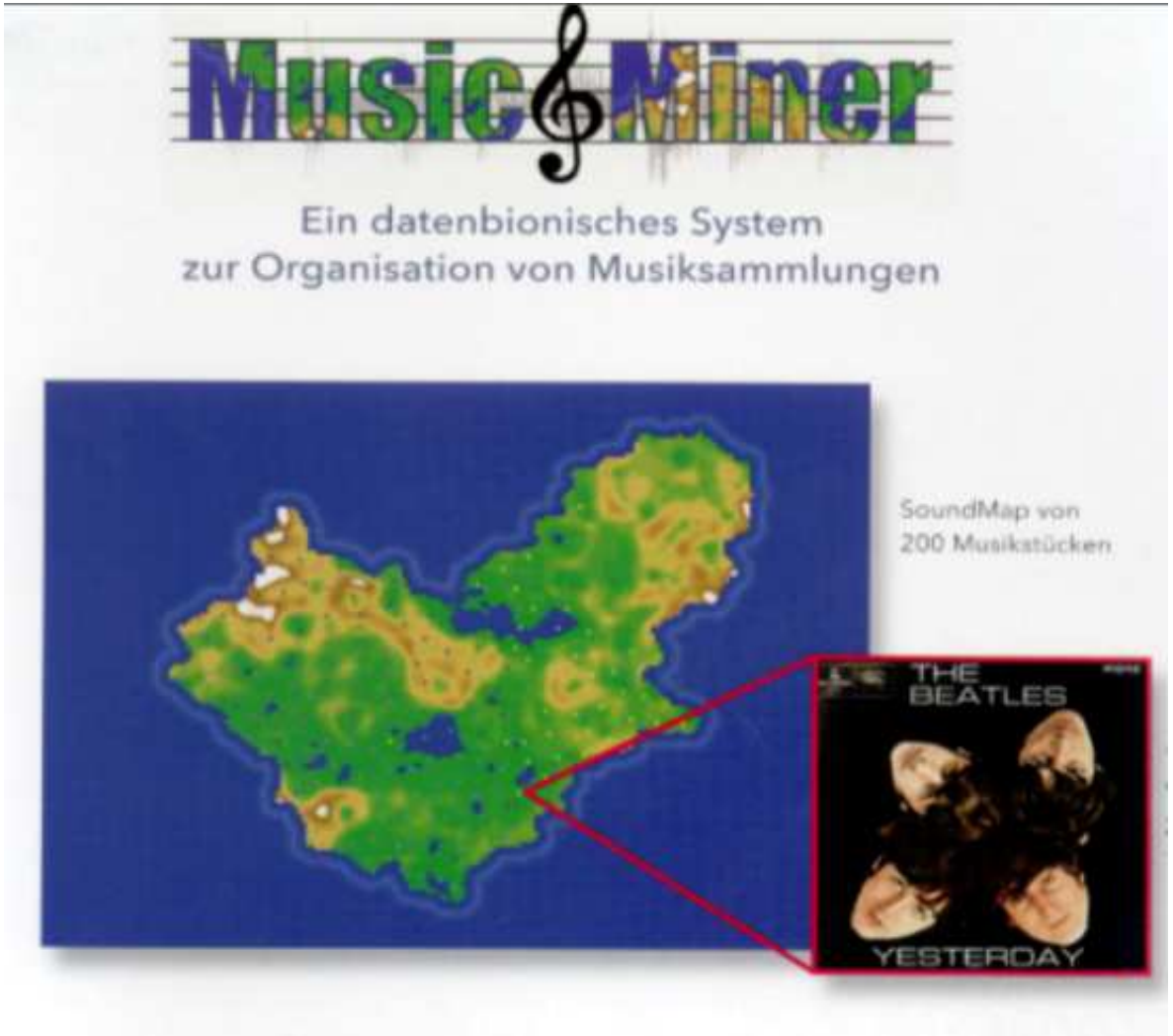
SOM, Websom


interpreted
universities
workshop
warren
costa
postdoctoral
aisb
neuron
phonene
signal
engines
genesis
connect
bootstrap
progranner
extrapolation
curves
atree
weightless
pdp
neuroconputing
elman
paradigm
personnel
judgement
java
ai
conjugate
principle
ga
fortran
intelligence
trading
brain
alanos
toolbox
consciousness
levenberg-marquardt
rbf
robot
neurotransmitters
scheduling
backpropagator's
papers
signals
tools
snns
noise
variable
nining
tdl
decay
encoding
bayes
unsupervised
neurofuzzy
benchmark
hidden
popular
signoid
validation
rate

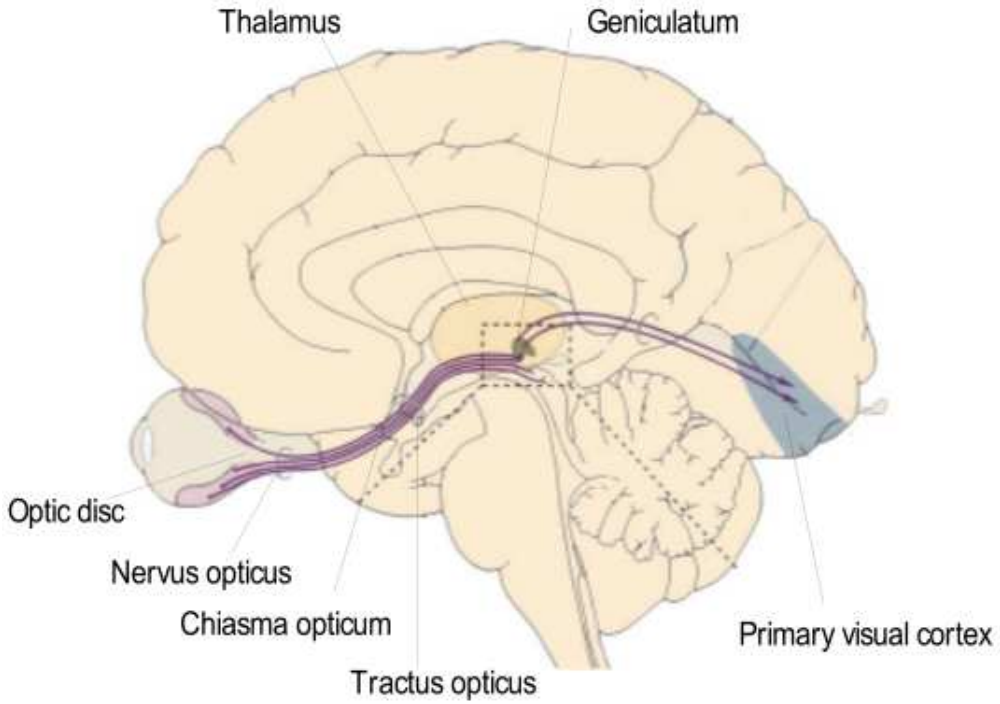
SOM, Websom



SOM, MusicMiner



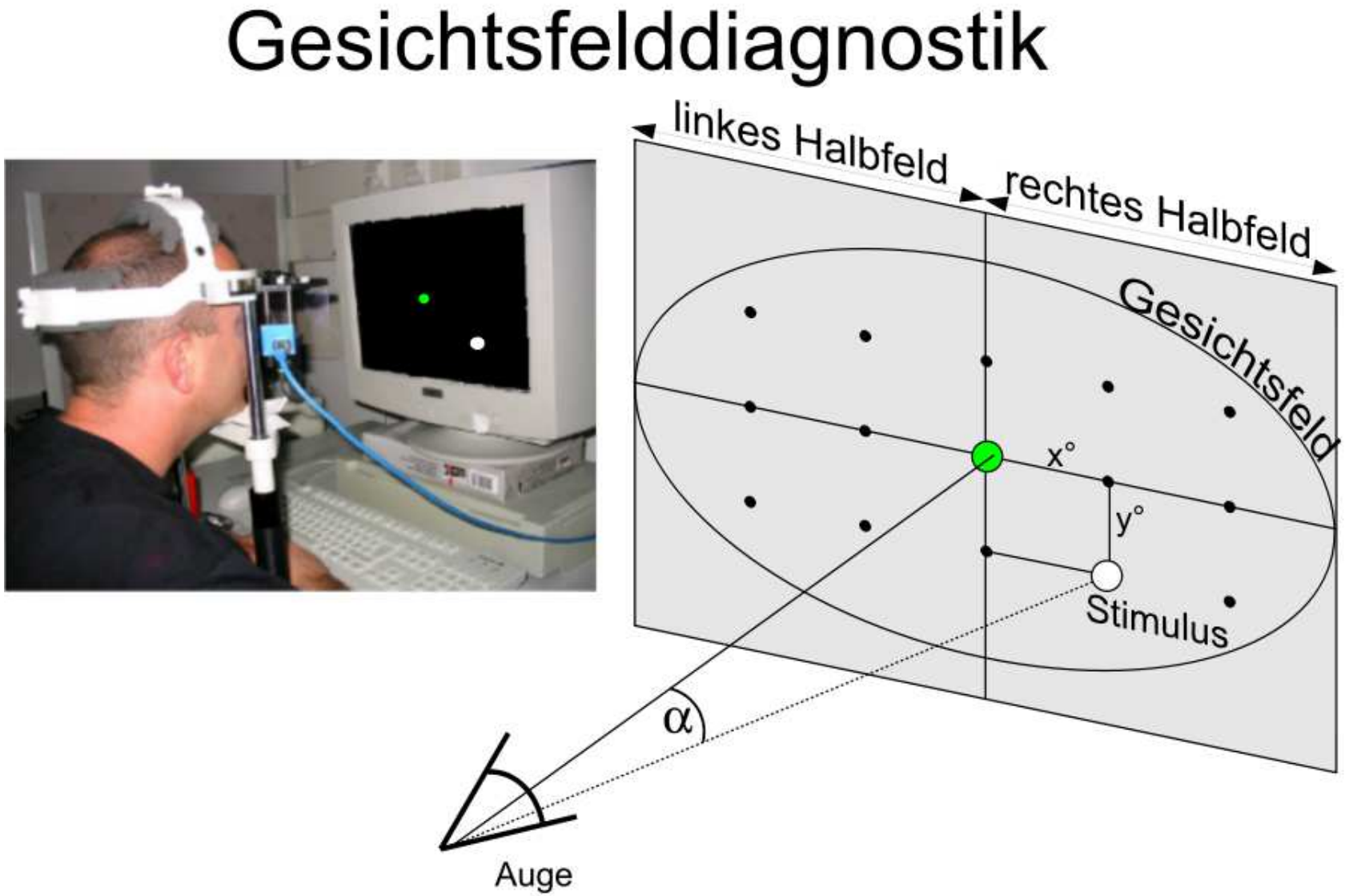
Gesichtsfeldausfall

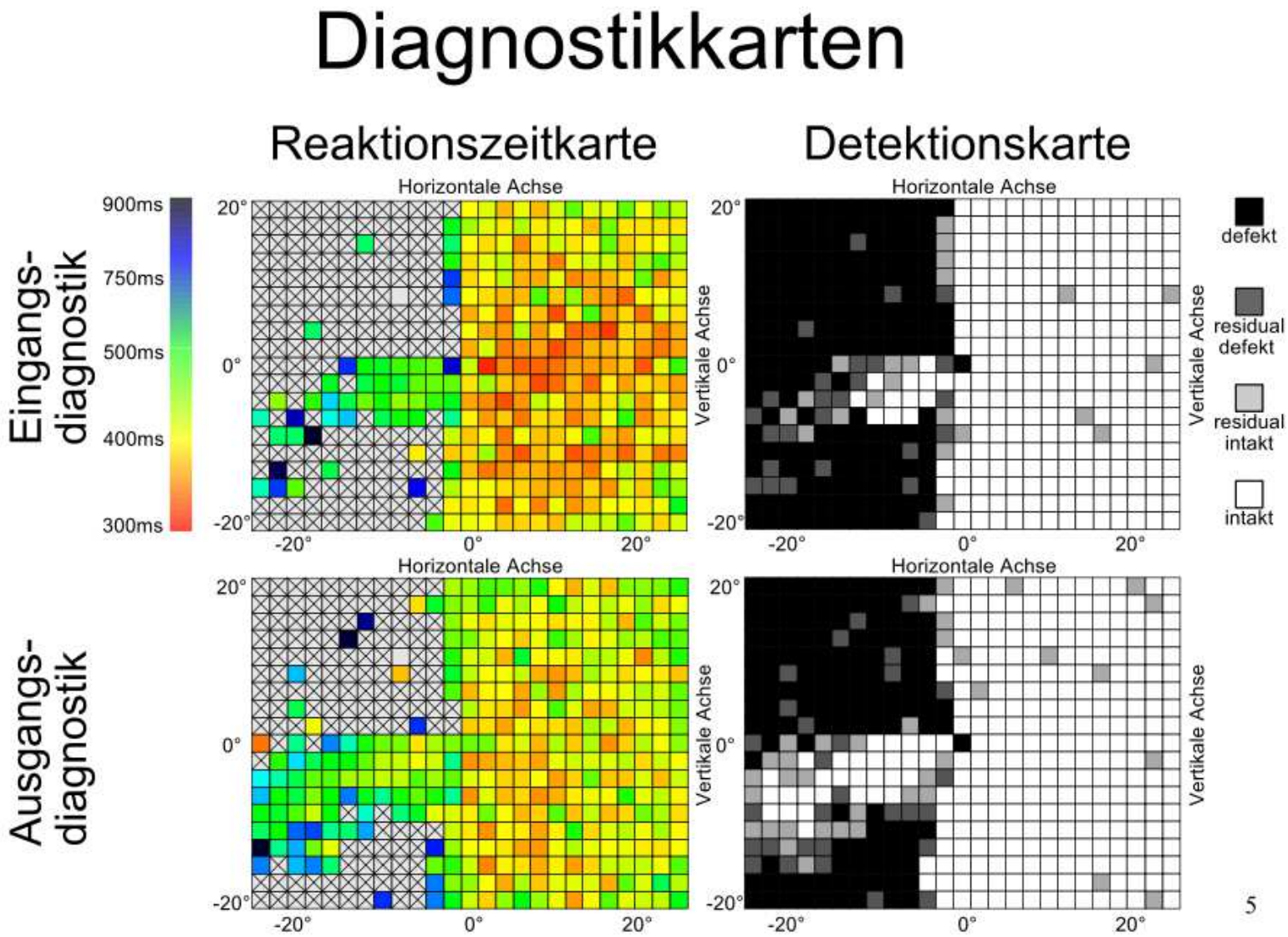


E.Kandel, J. Schwartz, T. Jessell: Neurowissenschaften, 1996.

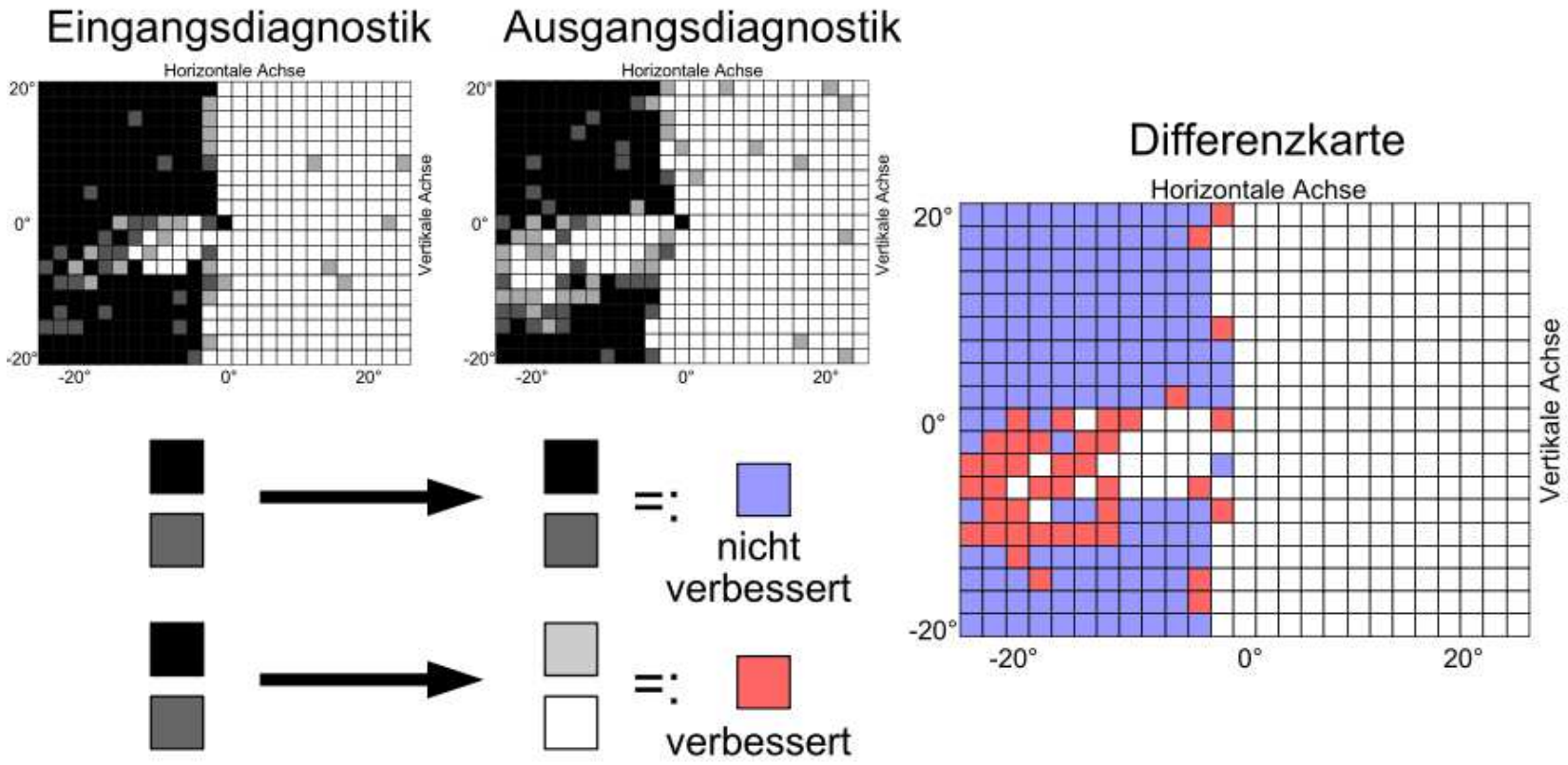
Visuelle Restitutionstherapie
(6 Monate à 1h pro Tag)

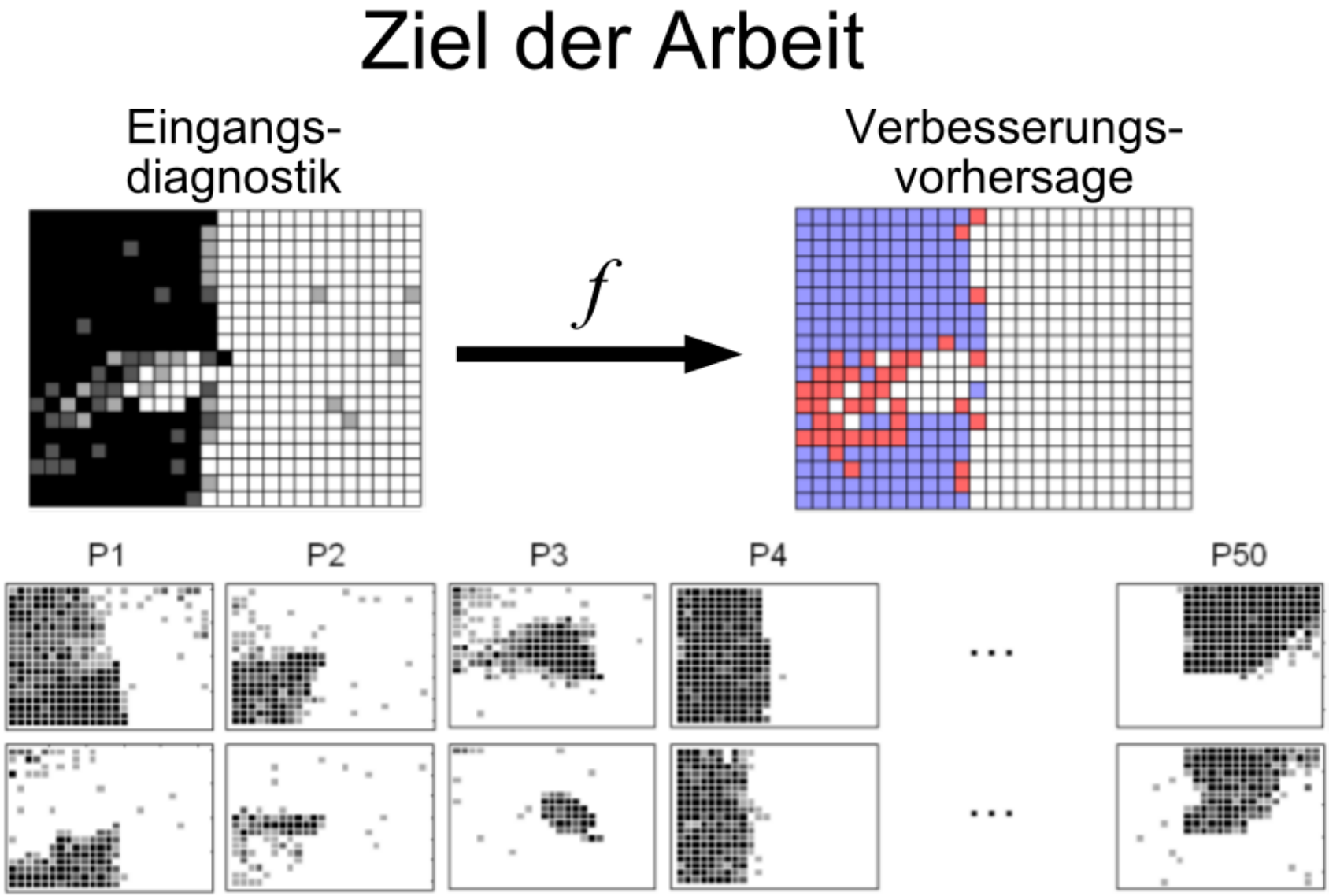
E. Kasten, S. Wuest, W. Behrens-Baumann, and B. A. Sabel.
Computerbased training for the treatment of partial blindness.
Nat Med, 4(9):1083–7,1998.


Gesichtsfelddiagnostik
linkes Halbfeld
rechtes Halbfeld
Gesichtsfeld
x°
y°
Stimulus
α
Auge


Diagnostikkarten
Reaktionszeitkarte
Detektionskarte
Eingangs-diagnostik
Ausgangs-diagnostik
Horizontale Achse
Vertikale Achse
900ms
750ms
500ms
400ms
300ms
20°
0°
-20°
defekt
residual defekt
residual intakt
intakt
5

Verbesserte und nicht verbesserte Positionen




Ziel der Arbeit
Eingangs-
diagnostik
f
Verbesserungs-
vorhersage
P1
P2
P3
P4
P50

Hopfield-Netze

Hopfield-Netze

Ein **Hopfield-Netz** ist ein neuronales Netz mit einem Graphen $G = (U, C)$, das die folgenden Bedingungen erfüllt:

(i) $U_{\text{hidden}} = \emptyset$, $U_{\text{in}} = U_{\text{out}} = U$,

(ii) $C = U \times U - \{(u, u) \mid u \in U\}$.

- In einem Hopfield-Netz sind alle Neuronen sowohl Eingabe- als auch Ausgabeneuronen.
- Es gibt keine versteckten Neuronen.
- Jedes Neuron erhält seine Eingaben von allen anderen Neuronen.
- Ein Neuron ist nicht mit sich selbst verbunden.

Die Verbindungsgewichte sind symmetrisch, d.h.

$$\forall u, v \in U, u \neq v : \qquad w_{uv} = w_{vu}.$$

Hopfield-Netze

Die Netzeingabefunktion jedes Neurons ist die gewichtete Summe der Ausgaben aller anderen Neuronen, d.h.

$$\forall u \in U: \quad f_{\text{net}}^{(u)}(\boldsymbol{w}_u, \mathbf{in}_u) = \boldsymbol{w}_u \mathbf{in}_u = \sum_{v \in U - \{u\}} w_{uv} \, \text{out}_v \, .$$

Die Aktivierungsfunktion jedes Neurons ist eine Sprungfunktion, d.h.

$$\forall u \in U: \quad f_{\text{act}}^{(u)}(\text{net}_u, \theta_u) = \begin{cases} 1, & \text{falls} \quad \text{net}_u \geq \theta_u, \\ -1, & \text{sonst.} \end{cases}$$

Die Ausgabefunktion jedes Neurons ist die Identität, d.h.

$$\forall u \in U: \quad f_{\text{out}}^{(u)}(\text{act}_u) = \text{act}_u \, .$$

Alternative Aktivierungsfunktion

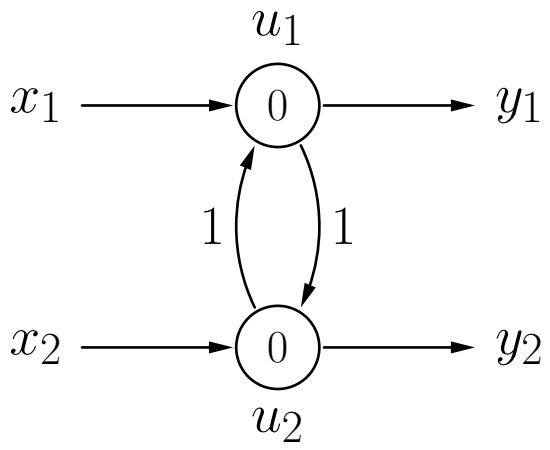
$$\forall u \in U: \quad f_{\text{act}}^{(u)}(\text{net}_u, \theta_u, \text{act}_u) = \begin{cases} 1, & \text{falls} \quad \text{net}_u > \theta, \\ -1, & \text{falls} \quad \text{net}_u < \theta, \\ \text{act}_u, & \text{falls} \quad \text{net}_u = \theta. \end{cases}$$

Diese Aktivierungsfunktion bietet einige Vorteile bei der späteren physikalischen Interpretation eines Hopfield-Netzes. Diese wird allerdings in der Vorlesung nicht weiter genutzt.

Allgemeine Gewichtsmatrix eines Hopfield-Netzes

$$\mathbf{W} = \begin{pmatrix} 0 & w_{u_1 u_2} & \dots & w_{u_1 u_n} \\ w_{u_1 u_2} & 0 & \dots & w_{u_2 u_n} \\ \vdots & \vdots & & \vdots \\ w_{u_1 u_n} & w_{u_1 u_n} & \dots & 0 \end{pmatrix}$$

Sehr einfaches Hopfield-Netz



$$\mathbf{W} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

Das Verhalten eines Hopfield-Netzes kann von der Update-Reihenfolge abhängen.

- Die Berechnungen können oszillieren, wenn Neuronen synchron aktualisiert werden.
- Die Berechnung konvergiert immer, wenn die Neuronen asynchron in fester Reihenfolge aktualisiert werden.

Parallele Aktualisierung der Neuronenaktivierungen

	u_1	u_2
Eingabephase	$\mathbf{-1}$	$\mathbf{1}$
Arbeitsphase	$\mathbf{1}$	$\mathbf{-1}$
	$\mathbf{-1}$	$\mathbf{1}$
	$\mathbf{1}$	$\mathbf{-1}$
	$\mathbf{-1}$	$\mathbf{1}$
	$\mathbf{1}$	$\mathbf{-1}$
	$\mathbf{-1}$	$\mathbf{1}$

- Die Berechnungen oszillieren, kein stabiler Zustand wird erreicht.
- Die Ausgabe hängt davon ab, wann die Berechnung abgebrochen wird.

Hopfield-Netze: Beispiele

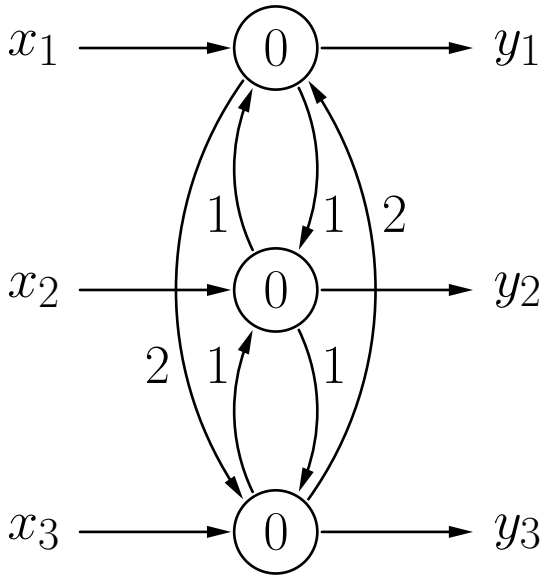
Sequentielle Aktualisierung der Neuronenaktivierungen

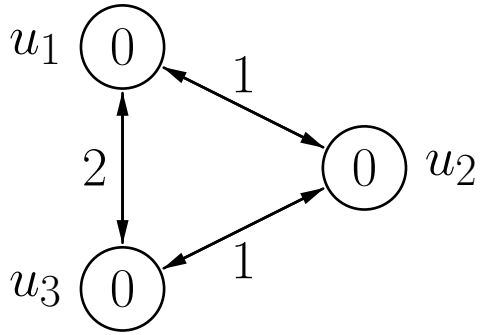
	u_1	u_2
Eingabephase	$\mathbf{-1}$	$\mathbf{1}$
Arbeitsphase	$\mathbf{1}$	1
	1	$\mathbf{1}$
	$\mathbf{1}$	1
	1	$\mathbf{1}$

	u_1	u_2
Eingabephase	$\mathbf{-1}$	$\mathbf{1}$
Arbeitsphase	-1	$\mathbf{-1}$
	$\mathbf{-1}$	-1
	-1	$\mathbf{-1}$
	$\mathbf{-1}$	-1

- Aktivierungsreihenfolge u_1, u_2, u_1, ... bzw. u_2, u_1, u_1, ...
- Unabhängig von der Reihenfolge wird ein stabiler Zustand erreicht.
- Welcher Zustand stabil ist, hängt von der Reihenfolge ab.

Vereinfachte Darstellung eines Hopfield-Netzes

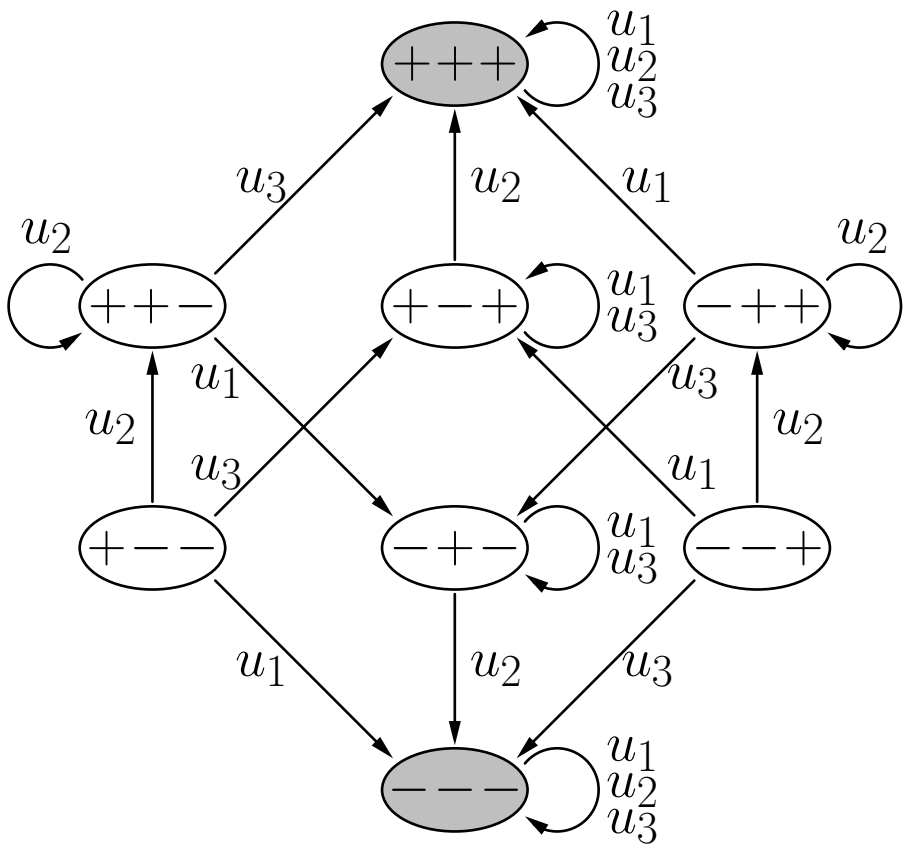




$$\mathbf{W} = \begin{pmatrix} 0 & 1 & 2 \\ 1 & 0 & 1 \\ 2 & 1 & 0 \end{pmatrix}$$

- Symmetrische Verbindungen zwischen Neuronen werden kombiniert.
- Eingaben und Ausgaben werden nicht explizit dargestellt.

Graph der Aktivierungen und Übergänge



(Zustandsgraph zum Netz auf der vorherigen Folie, Erläuterung nächste Folie)

Graph der Aktivierungen und Übergänge

- +/-: Aktivierung der Neuronen (+ entspricht +1, − entspricht −1)
- Pfeile: geben die Neuronen an, deren Aktualisierung zu dem jeweiligen Zustandsübergang führt
- grau unterlegte Zustände: stabile Zustände
- beliebige Aktualisierungsreihenfolgen ablesbar

Hopfield-Netze: Konvergenz der Berechnungen

Konvergenztheorem: Wenn die Aktivierungen der Neuronen eines Hopfield-Netzes asynchron (sequentiell) durchgeführt werden, wird ein stabiler Zustand nach einer endlichen Anzahl von Schritten erreicht.

Wenn die Neuronen zyklisch in einer beliebigen, aber festen Reihenfolge durchlaufen werden, sind höchstens $n \cdot 2^n$ Schritte (Aktualisierungen einzelner Neuronen) notwendig, wobei n die Anzahl der Neuronen im Netz ist.

Der Beweis erfolgt mit Hilfe einer **Energiefunktion**.
Die Energiefunktion eines Hopfield-Netzes mit n Neuronen $u_1, \ldots, u_n$ ist

$$\begin{aligned} E &= -\frac{1}{2}\,\mathbf{act}^\top \mathbf{W}\mathbf{act} + \boldsymbol{\theta}^T\mathbf{act} \\ &= -\frac{1}{2} \sum_{u,v \in U, u \neq v} w_{uv}\,\mathrm{act}_u\,\mathrm{act}_v + \sum_{u \in U} \theta_u\,\mathrm{act}_u\,. \end{aligned}$$

Hopfield-Netze: Konvergenz

Man betrachte die Energieänderung die aus einer aktivierungsändernden Aktualisierung entsteht:

$$\begin{aligned}
\Delta E = E^{(\text{new})} - E^{(\text{old})} &= \Big(- \sum_{v \in U - \{u\}} w_{uv} \, \text{act}_u^{(\text{new})} \, \text{act}_v + \theta_u \, \text{act}_u^{(\text{new})} \Big) \\
&- \Big(- \sum_{v \in U - \{u\}} w_{uv} \, \text{act}_u^{(\text{old})} \, \text{act}_v + \theta_u \, \text{act}_u^{(\text{old})} \Big) \\
&= \Big(\text{act}_u^{(\text{old})} - \text{act}_u^{(\text{new})} \Big) \Big(\underbrace{\sum_{v \in U - \{u\}} w_{uv} \, \text{act}_v}_{= \text{net}_u} - \theta_u \Big).
\end{aligned}$$

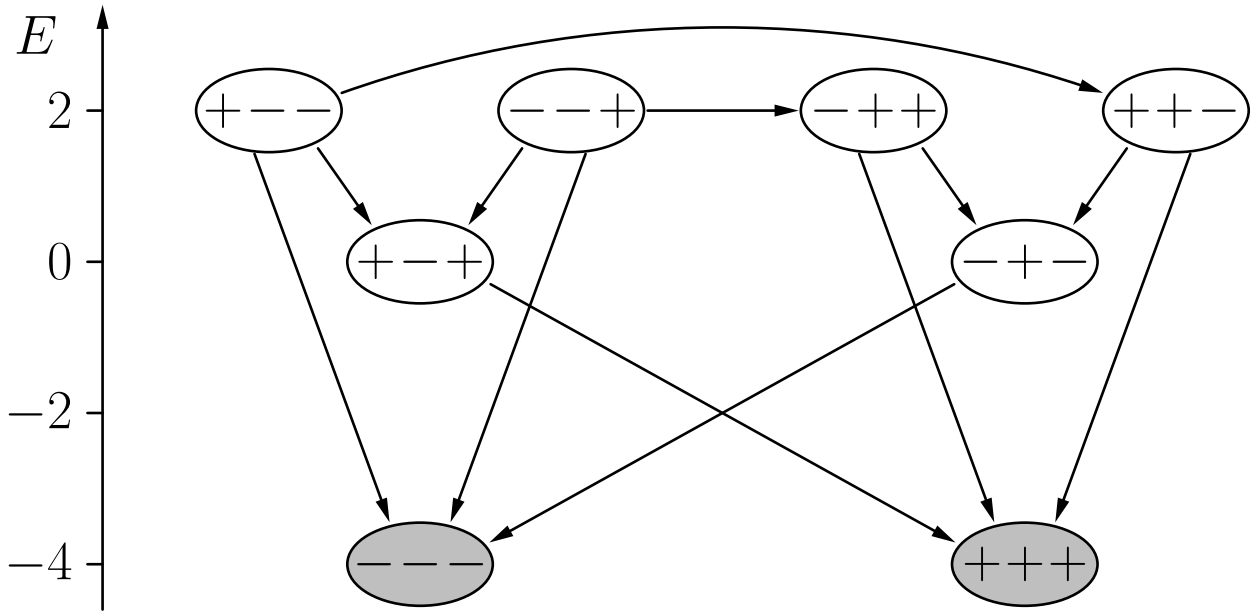
- $\text{net}_u < \theta_u$: Zweiter Faktor kleiner als 0.
  $\text{act}_u^{(\text{new})} = -1$ and $\text{act}_u^{(\text{old})} = 1$, daher erster Faktor größer als 0.
  Ergebnis: $\Delta E < 0$.
- $\text{net}_u \geq \theta_u$: Zweiter Faktor größer als oder gleich 0.
  $\text{act}_u^{(\text{new})} = 1$ und $\text{act}_u^{(\text{old})} = -1$, daher erster Faktor kleiner als 0.
  Ergebnis: $\Delta E \leq 0$.

Hopfield-Netze: Konvergenz

Höchstens $n \cdot 2^n$ Schritte bis zur Konvergenz:

- die beliebige, aber feste Reihenfolge sorgt dafür, dass alle Neuronen zyklisch durchlaufen und neuberechnet werden
  - a) es ändert sich keine Aktivierung – ein stabiler Zustand wurde erreicht
  - b) es ändert sich mindestens eine Aktivierung: dann wurde damit mindestens einer der 2^n möglichen Aktivierungszustände ausgeschlossen.
- Ein einmal verlassener Zustand kann nicht wieder erreicht werden (siehe vorherige Folien).
- D.h. nach spätestens 2^n Durchläufen durch die n Neuronen ist ein stabiler Zustand erreicht.

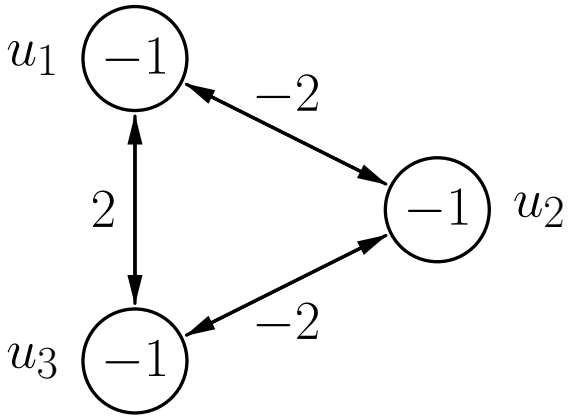
Ordne die Zustände im Graphen entsprechend ihrer Energie

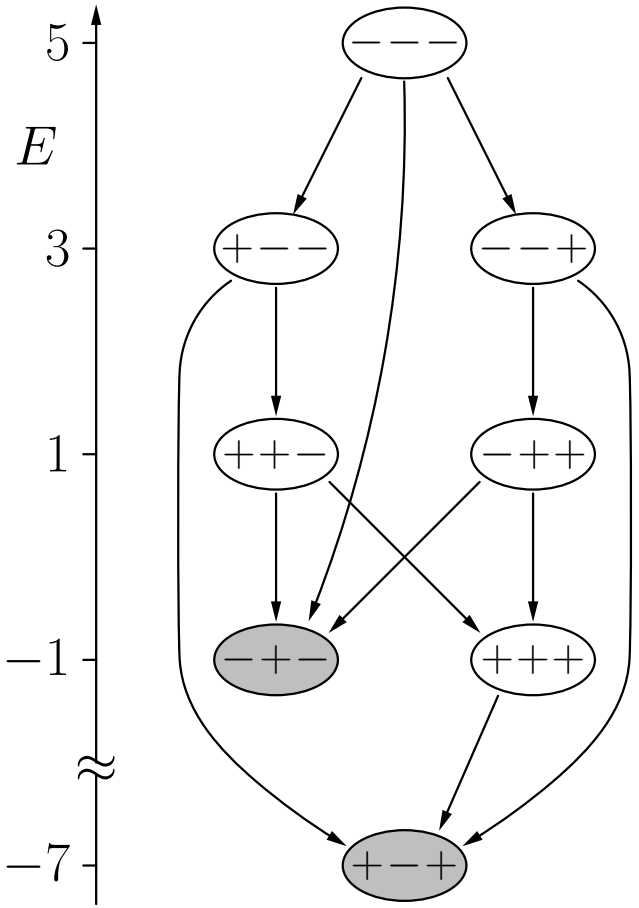


Energiefunktion für das Beispiel-Hopfield-Netz:

$$E = -\operatorname{act}_{u_1}\operatorname{act}_{u_2} - 2\operatorname{act}_{u_1}\operatorname{act}_{u_3} - \operatorname{act}_{u_2}\operatorname{act}_{u_3}\,.$$

Der Zustandsgraph muss nicht symmetrisch sein





Physikalische Interpretation: Magnetismus

Ein Hopfield-Netz kann als (mikroskopisches) Modell von Magnetismus gesehen werden (sogenanntes Ising-Modell, [Ising 1925]).

physikalisch	neuronal
Atom	Neuron
Magnetisches Moment (Spin)	Aktivierungszustand
Stärke des äußeren Magnetfeldes	Schwellenwert
Magnetische Kopplung der Atome	Verbindungsgewichte
Hamilton-Operator des Magnetfeldes	Energiefunktion

Idee: Nutze stabile Zustände, um Muster zu speichern

Zuerst: Speichere nur ein Muster $\boldsymbol{x} = (\mathrm{act}_{u_1}^{(l)}, \ldots, \mathrm{act}_{u_n}^{(l)})^\top \in \{-1, 1\}^n$, $n \geq 2$,
d.h. finde Gewichte, so dass der Zustand ein stabiler Zustand wird.

Notwendige und hinreichende Bedingung:

$$S(\mathbf{W}\boldsymbol{x} - \boldsymbol{\theta}\,) = \boldsymbol{x},$$

wobei

$$\begin{array}{rcl} S : \mathbb{R}^n & \to & \{-1, 1\}^n, \\ \boldsymbol{x} & \mapsto & \boldsymbol{y} \end{array}$$

mit

$$\forall i \in \{1, \ldots, n\}: \quad y_i = \begin{cases} 1, & \text{falls } x_i \geq 0, \\ -1, & \text{sonst.} \end{cases}$$

Falls $\boldsymbol{\theta} = \mathbf{0}$, dann kann eine passende Matrix $\mathbf{W}$ leicht berechnet werden. Es reicht eine Matrix $\mathbf{W}$ zu finden mit

$$\mathbf{W}\boldsymbol{x} = c\boldsymbol{x} \qquad \text{mit } c \in \mathbb{R}^+.$$

Algebraisch: Finde eine Matrix $\mathbf{W}$ die einen positiven Eigenwert in Bezug auf $\boldsymbol{x}$ hat.

Wähle

$$\mathbf{W} = \boldsymbol{x}\boldsymbol{x}^T - \mathbf{E}$$

wobei $\boldsymbol{x}\boldsymbol{x}^T$ das sogenannte **äußere Produkt** von $\boldsymbol{x}$ mit sich selbst ist.

Mit dieser Matrix erhalten wir

$$\begin{aligned} \mathbf{W}\boldsymbol{x} &= (\boldsymbol{x}\boldsymbol{x}^T)\boldsymbol{x} - \underbrace{\mathbf{E}\boldsymbol{x}}_{=\boldsymbol{x}} \overset{(*)}{=} \boldsymbol{x}\,\underbrace{(\boldsymbol{x}^T\boldsymbol{x})}_{=|\boldsymbol{x}|^2=n} - \boldsymbol{x} \\ &= n\boldsymbol{x} - \boldsymbol{x} = (n-1)\boldsymbol{x}. \end{aligned}$$

Hebb'sche Lernregel [Hebb 1949]

In einzelnen Gewichten aufgeschrieben lautet die Berechnung der Gewichtsmatrix wie folgt:

$$w_{uv} = \begin{cases} 0, & \text{falls } u = v, \\ 1, & \text{falls } u \neq v, \text{act}_u^{(p)} = \text{act}_u^{(v)}, \\ -1, & \text{sonst.} \end{cases}$$

- Ursprünglich von biologischer Analogie abgeleitet.
- Verstärkt Verbindungen zwischen Neuronen, die zur selben Zeit aktiv sind.

Diese Lernregel speichert auch das Komplement des Musters:

Mit $\mathbf{W}\boldsymbol{x} = (n-1)\boldsymbol{x}$ ist daher auch $\mathbf{W}(-\boldsymbol{x}) = (n-1)(-\boldsymbol{x})$.

Speichern mehrerer Muster

Wähle

$$\begin{aligned}\mathbf{W}\boldsymbol{x}_j &= \sum_{i=1}^{m} \mathbf{W}_i \boldsymbol{x}_j &&= \left(\sum_{i=1}^{m} (\boldsymbol{x}_i {\boldsymbol{x}_i}^T)\boldsymbol{x}_j\right) - m\underbrace{\mathbf{E}\boldsymbol{x}_j}_{=\boldsymbol{x}_j} \\ &&&= \left(\sum_{i=1}^{m} \boldsymbol{x}_i({\boldsymbol{x}_i}^T\boldsymbol{x}_j)\right) - m\boldsymbol{x}_j\end{aligned}$$

Wenn die Muster orthogonal sind, gilt

$${\boldsymbol{x}_i}^T\boldsymbol{x}_j = \begin{cases} 0, & \text{falls} \quad i \neq j, \\ n, & \text{falls} \quad i = j, \end{cases}$$

und daher

$$\mathbf{W}\boldsymbol{x}_j = (n-m)\boldsymbol{x}_j.$$

Speichern mehrerer Muster

Ergebnis: So lange $m < n$, ist $\boldsymbol{x}_j$ ein stabiler Zustand des Hopfield-Netzes.

Man beachte, dass die Komplemente der Muster ebenfalls gespeichert werden.

Mit $\mathbf{W}\boldsymbol{x}_j = (n-m)\boldsymbol{x}_j$ ist daher auch $\mathbf{W}(-\boldsymbol{x}_j) = (n-m)(-\boldsymbol{x}_j)$.

Aber: die Speicherkapazität ist verglichen mit der Anzahl möglicher Zustände sehr klein (2^n).

Nicht-orthogonale Muster:

$$\mathbf{W}\boldsymbol{x}_j = (n-m)\boldsymbol{x}_j + \underbrace{\sum_{\substack{i=1 \\ i \neq j}}^{m} \boldsymbol{x}_i(\boldsymbol{x}_i{}^T \boldsymbol{x}_j)}_{\text{“Störterm”}} .$$

- p_j kann trotzdem stabil sein, wenn $n-m \geq 0$ gilt und der “Störterm” hinreichend klein ist.
- Dieser Fall tritt ein, wenn die Muster “annähernd” orthogonal sind.
- Je größer die Zahl der zu speichernden Muster ist, desto kleiner muß der Störterm sein.
- Die theoretische Maximalkapazität eines Hopfield-Netzes wird praktisch nie erreicht.

Assoziativspeicher: Beispiel

Beispiel: Speichere Muster $\boldsymbol{x}_1 = (+1, +1, -1, -1)^\top$ und $\boldsymbol{x}_2 = (-1, +1, -1, +1)^\top$.

$$\mathbf{W} = \mathbf{W}_1 + \mathbf{W}_2 = \boldsymbol{x}_1 \boldsymbol{x}_1^T + \boldsymbol{x}_2 \boldsymbol{x}_2^T - 2\mathbf{E}$$

wobei

$$\mathbf{W}_1 = \begin{pmatrix} 0 & 1 & -1 & -1 \\ 1 & 0 & -1 & -1 \\ -1 & -1 & 0 & 1 \\ -1 & -1 & 1 & 0 \end{pmatrix}, \qquad \mathbf{W}_2 = \begin{pmatrix} 0 & -1 & 1 & -1 \\ -1 & 0 & -1 & 1 \\ 1 & -1 & 0 & -1 \\ -1 & 1 & -1 & 0 \end{pmatrix}.$$

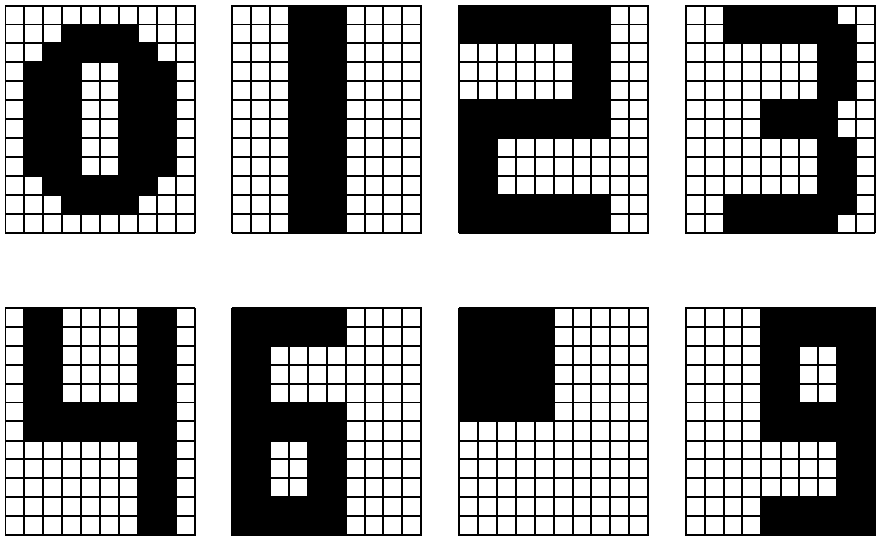
Die vollständige Gewichtsmatrix ist:

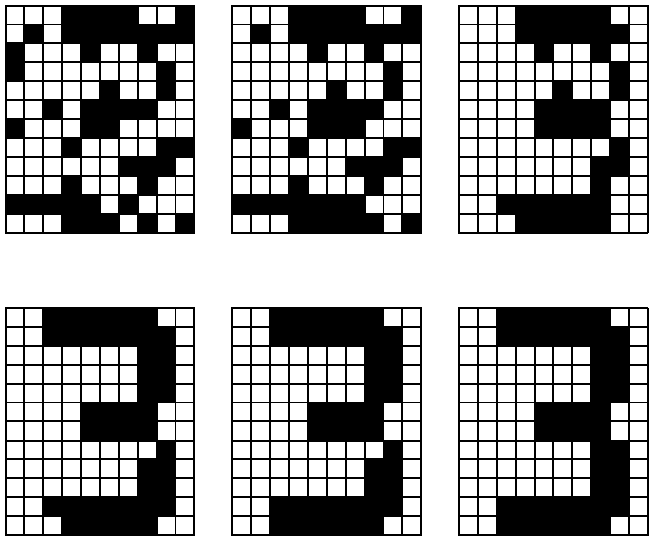
$$\mathbf{W} = \begin{pmatrix} 0 & 0 & 0 & -2 \\ 0 & 0 & -2 & 0 \\ 0 & -2 & 0 & 0 \\ -2 & 0 & 0 & 0 \end{pmatrix}.$$

Daher ist

$$\mathbf{W}\boldsymbol{x}_1 = (+2, +2, -2, -2)^\top \qquad \text{und} \qquad \mathbf{W}\boldsymbol{x}_1 = (-2, +2, -2, +2)^\top.$$

Beispiel: Speichere Bitmaps von Zahlen





- Links: Bitmaps, die im Hopfield-Netz gespeichert sind.
- Rechts: Rekonstruktion eines Musters aus einer zufälligen Eingabe.

Trainieren eines Hopfield-Netzes mit der Delta-Regel

Notwendige Bedingung, dass ein Muster $\boldsymbol{x}$ einem stabilen Zustand entspricht:

$$\begin{array}{llllll}
s(0 & + w_{u_1u_2}\,\mathrm{act}_{u_2}^{(p)} & + \ldots + w_{u_1u_n}\,\mathrm{act}_{u_n}^{(p)} & - \theta_{u_1}) = \mathrm{act}_{u_1}^{(p)}, \\
s(w_{u_2u_1}\,\mathrm{act}_{u_1}^{(p)} & + 0 & + \ldots + w_{u_2u_n}\,\mathrm{act}_{u_n}^{(p)} & - \theta_{u_2}) = \mathrm{act}_{u_2}^{(p)}, \\
\vdots & \vdots & \vdots & \vdots \quad \vdots \\
s(w_{u_nu_1}\,\mathrm{act}_{u_1}^{(p)} & + w_{u_nu_2}\,\mathrm{act}_{u_2}^{(p)} & + \ldots + 0 & - \theta_{u_n}) = \mathrm{act}_{u_n}^{(p)}.
\end{array}$$

mit der standardmäßigen Schwellenwertfunktion

$$s(x) = \begin{cases} 1, & \text{falls} \quad x \geq 0, \\ -1, & \text{sonst.} \end{cases}$$

Trainieren eines Hopfield-Netzes mit der Delta-Regel

Überführe Gewichtsmatrix in einen Gewichtsvektor:

$$\boldsymbol{w} = (\begin{array}{llll} w_{u_1u_2}, & w_{u_1u_3}, & \ldots, & w_{u_1u_n}, \\ & w_{u_2u_3}, & \ldots, & w_{u_2u_n}, \\ & & \ddots & \vdots \\ & & & w_{u_{n-1}u_n}, \\ -\theta_{u_1}, & -\theta_{u_2}, & \ldots, & -\theta_{u_n} \end{array}).$$

Konstruiere Eingabevektoren für ein Schwellenwertelement

$$\boldsymbol{z}_2 = (\mathrm{act}_{u_1}^{(p)}, \underbrace{0, \ldots, 0,}_{n-2 \text{ Nullen}} \mathrm{act}_{u_3}^{(p)}, \ldots, \mathrm{act}_{u_n}^{(p)}, \ldots 0, 1, \underbrace{0, \ldots, 0}_{n-2 \text{ Nullen}}).$$

Wende die Deltaregel auf diesen Gewichtsvektor und die Eingabevektoren an, bis sich Konvergenz einstellt.

Nutze Energieminimierung, um Optimierungsprobleme zu lösen

Allgemeine Vorgehensweise:

- Transformiere die zu optimierende Funktion in eine zu minimierende.
- Transformiere Funktion in die Form einer Energiefunktion eines Hopfield-Netzes.
- Lies die Gewichte und Schwellenwerte der Energiefunktion ab.
- Konstruiere das zugehörige Hopfield-Netz.
- Initialisiere das Hopfield-Netz zufällig und aktualisiere es solange, bis sich Konvergenz einstellt.
- Lies die Lösung aus dem erreichten stabilen Zustand ab.
- Wiederhole mehrmals und nutze die beste gefundene Lösung.

Hopfield-Netze: Aktivierungstransformation

Ein Hopfield-Netz kann entweder mit Aktivierungen -1 und 1 oder mit Aktivierungen 0 and 1 definiert werden. Die Netze können ineinander umgewandelt werden.

Von $\mathrm{act}_u \in \{-1, 1\}$ in $\mathrm{act}_u \in \{0, 1\}$:

$$\begin{aligned} w^0_{uv} &= 2w^-_{uv} && \text{und} \\ \theta^0_u &= \theta^-_u + \sum_{v \in U - \{u\}} w^-_{uv} \end{aligned}$$

Von $\mathrm{act}_u \in \{0, 1\}$ in $\mathrm{act}_u \in \{-1, 1\}$:

$$\begin{aligned} w^-_{uv} &= \frac{1}{2} w^0_{uv} && \text{und} \\ \theta^-_u &= \theta^0_u - \frac{1}{2} \sum_{v \in U - \{u\}} w^0_{uv}. \end{aligned}$$

Hopfield-Netze: Lösen von Optimierungsproblemen

Kombinationslemma: Gegeben seien zwei Hopfield-Netze auf derselben Menge U Neuronen mit Gewichten $w_{uv}^{(i)}$, Schwellenwerten $\theta_u^{(i)}$ und Energiefunktionen

$$E_i = -\frac{1}{2} \sum_{u \in U} \sum_{v \in U - \{u\}} w_{uv}^{(i)} \operatorname{act}_u \operatorname{act}_v + \sum_{u \in U} \theta_u^{(i)} \operatorname{act}_u,$$

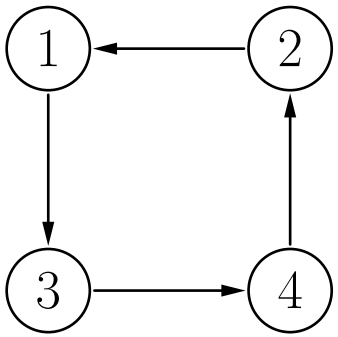
$i = 1, 2$. Weiterhin sei $a, b \in \mathbb{R}$. Dann ist $E = aE_1 + bE_2$ die Energiefunktion des Hopfield-Netzes auf den Neuronen in U das die Gewichte $w_{uv} = aw_{uv}^{(1)} + bw_{uv}^{(2)}$ und die Schwellenwerte $\theta_u = a\theta_u^{(1)} + b\theta_u^{(2)}$ hat.

Beweis: Einfach Berechnungen durchführen.

Idee: Zusätzliche Bedingungen können separat formuliert und später mit einbezogen werden.

Beispiel: Problem des Handlungsreisenden (TSP – Traveling Salesman Problem)

Idee: Stelle Tour durch Matrix dar.



$$\begin{array}{c} \\ \\ \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix} \end{array} \quad \begin{array}{c} \text{Stadt} \\ 1\ \ 2\ \ 3\ \ 4 \\ 1. \\ 2. \\ 3. \\ 4. \end{array} \quad \text{Schritt}$$

Ein Element m_{ij} der Matrix ist 1 wenn die i-te Stadt im j-ten Schritt besucht wird und 0 sonst.

Jeder Matrixeintrag wird durch ein Neuron repräsentiert.

Minimierung der Tourlänge

$$E_1 = \sum_{j_1=1}^{n} \sum_{j_2=1}^{n} \sum_{i=1}^{n} d_{j_1 j_2} \cdot m_{i j_1} \cdot m_{(i \bmod n)+1, j_2}.$$

Doppelsumme über die benötigten Schritte (Index i):

$$E_1 = \sum_{(i_1,j_1) \in \{1,\ldots,n\}^2} \sum_{(i_2,j_2) \in \{1,\ldots,n\}^2} d_{j_1 j_2} \cdot \delta_{(i_1 \bmod n)+1, i_2} \cdot m_{i_1 j_1} \cdot m_{i_2 j_2},$$

wobei

$$\delta_{ab} = \begin{cases} 1, & \text{falls} \quad a = b, \\ 0, & \text{sonst.} \end{cases}$$

Symmetrische Version der Energiefunktion:

$$E_1 = -\frac{1}{2} \sum_{\substack{(i_1,j_1) \in \{1,\ldots,n\}^2 \\ (i_2,j_2) \in \{1,\ldots,n\}^2}} -d_{j_1 j_2} \cdot (\delta_{(i_1 \bmod n)+1, i_2} + \delta_{i_1, (i_2 \bmod n)+1}) \cdot m_{i_1 j_1} \cdot m_{i_2 j_2}$$

Hopfield-Netze: Lösen von Optimierungsproblemen

Zusätzliche Bedingungen, die erfüllt werden müssen:

- Jede Stadt wird in genau einem Schritt der Tour besucht:

$$\forall j \in \{1, \ldots, n\}: \qquad \sum_{i=1}^{n} m_{ij} = 1,$$

  d.h. jede Spalte der Matrix enthält genau eine 1.

- In jedem Schritt der Tour wird genau eine Stadt besucht:

$$\forall i \in \{1, \ldots, n\}: \qquad \sum_{j=1}^{n} m_{ij} = 1,$$

  d.h. jede Zeile der Matrix enthält genau eine 1.

Diese Bedingungen werden erfüllt durch zusätzlich zu optimierende Funktionen.

Hopfield-Netze: Lösen von Optimierungsproblemen

Formalisierung der ersten Bedingung als Minimierungsproblem:

$$\begin{aligned} E_2^* &= \sum_{j=1}^{n} \left(\left(\sum_{i=1}^{n} m_{ij} \right)^2 - 2 \sum_{i=1}^{n} m_{ij} + 1 \right) \\ &= \sum_{j=1}^{n} \left(\left(\sum_{i_1=1}^{n} m_{i_1 j} \right) \left(\sum_{i_2=1}^{n} m_{i_2 j} \right) - 2 \sum_{i=1}^{n} m_{ij} + 1 \right) \\ &= \sum_{j=1}^{n} \sum_{i_1=1}^{n} \sum_{i_2=1}^{n} m_{i_1 j} m_{i_2 j} - 2 \sum_{j=1}^{n} \sum_{i=1}^{n} m_{ij} + n. \end{aligned}$$

Doppelsumme über benötigte Städte (Index i):

$$E_2 = \sum_{(i_1,j_1) \in \{1,\ldots,n\}^2} \sum_{(i_2,j_2) \in \{1,\ldots,n\}^2} \delta_{j_1 j_2} \cdot m_{i_1 j_1} \cdot m_{i_2 j_2} - 2 \sum_{(i,j) \in \{1,\ldots,n\}^2} m_{ij}.$$

Sich ergebende Energiefunktion:

$$E_2 = -\frac{1}{2} \sum_{\substack{(i_1,j_1)\in\{1,\ldots,n\}^2 \\ (i_2,j_2)\in\{1,\ldots,n\}^2}} -2\delta_{j_1 j_2} \cdot m_{i_1 j_1} \cdot m_{i_2 j_2} + \sum_{(i,j)\in\{1,\ldots,n\}^2} -2m_{ij}$$

Die zweite zusätzliche Bedingung wird analog gehandhabt:

$$E_3 = -\frac{1}{2} \sum_{\substack{(i_1,j_1)\in\{1,\ldots,n\}^2 \\ (i_2,j_2)\in\{1,\ldots,n\}^2}} -2\delta_{i_1 i_2} \cdot m_{i_1 j_1} \cdot m_{i_2 j_2} + \sum_{(i,j)\in\{1,\ldots,n\}^2} -2m_{ij}.$$

Kombinieren der Energiefunktionen:

$$E = aE_1 + bE_2 + cE_3 \qquad \text{wobei} \qquad \frac{b}{a} = \frac{c}{a} > 2 \max_{(j_1,j_2)\in\{1,\ldots,n\}^2} d_{j_1 j_2}.$$

Hopfield-Netze: Lösen von Optimierungsproblemen

Aus der resultierenden Energiefunktionen können wir die Gewichte

$$w_{(i_1,j_1)(i_2,j_2)} = \underbrace{-ad_{j_1 j_2} \cdot (\delta_{(i_1 \bmod n)+1,i_2} + \delta_{i_1,(i_2 \bmod n)+1})}_{\text{von } E_1} \underbrace{-2b\delta_{j_1 j_2}}_{\text{von } E_2} \underbrace{-2c\delta_{i_1 i_2}}_{\text{von } E_3}$$

und die Schwellenwerte:

$$\theta_{(i,j)} = \underbrace{0a}_{\text{von } E_1} \quad \underbrace{-2b}_{\text{von } E_2} \quad \underbrace{-2c}_{\text{von } E_3} = -2(b+c)$$

ablesen.

Problem: die zufällige Initialisierung und die Aktualisierung bis zur Konvergenz führen nicht immer zu einer Matrix, die tatsächlich eine Tour repräsentiert, geschweige denn eine optimale Tour.

Rekurrente Neuronale Netze

Rekurrente Netze: Abkühlungsgesetz

Ein Körper der Temperatur ϑ_0 wird in eine Umgebung der Temperatur ϑ_A eingebracht.

Die Abkühlung/Aufheizung des Körpers kann beschrieben werden durch das **Newtonsche Abkühlungsgesetz**:

$$\frac{\mathrm{d}\vartheta}{\mathrm{d}t} = \dot{\vartheta} = -k(\vartheta - \vartheta_A).$$

Exakte analytische Lösung:

$$\vartheta(t) = \vartheta_A + (\vartheta_0 - \vartheta_A)e^{-k(t-t_0)}$$

Ungefähre Lösung mit Hilfe des **Euler-Cauchyschen Polygonzuges**:

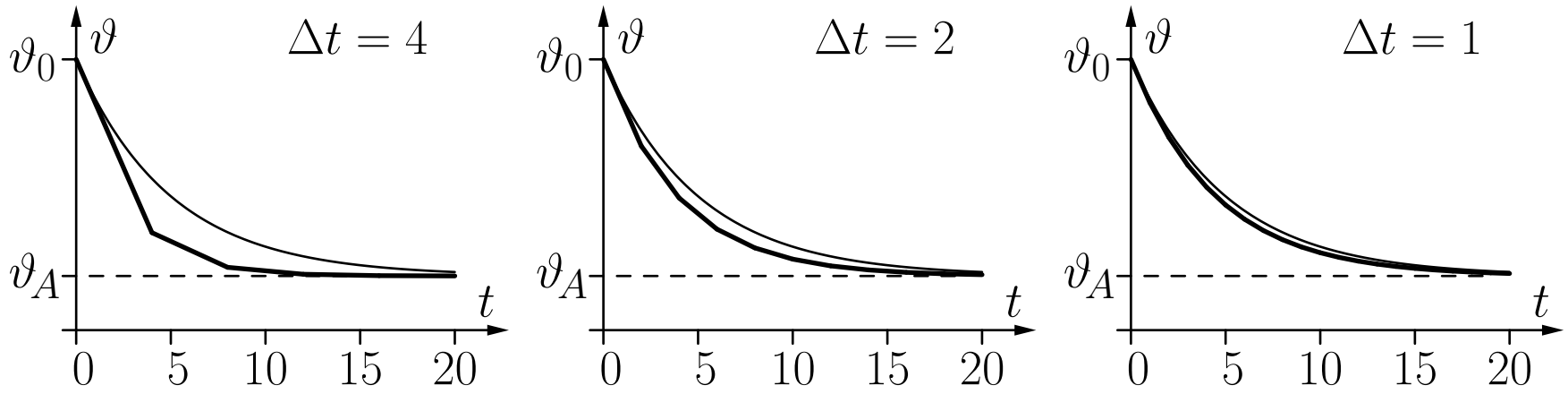
$$\vartheta_1 = \vartheta(t_1) = \vartheta(t_0) + \dot{\vartheta}(t_0)\Delta t = \vartheta_0 - k(\vartheta_0 - \vartheta_A)\Delta t.$$

$$\vartheta_2 = \vartheta(t_2) = \vartheta(t_1) + \dot{\vartheta}(t_1)\Delta t = \vartheta_1 - k(\vartheta_1 - \vartheta_A)\Delta t.$$

Allgemeine rekursive Gleichung:

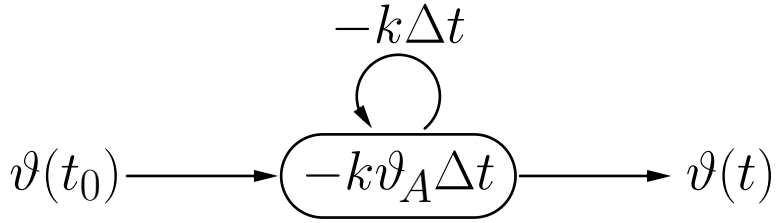
$$\vartheta_i = \vartheta(t_i) = \vartheta(t_{i-1}) + \dot{\vartheta}(t_{i-1})\Delta t = \vartheta_{i-1} - k(\vartheta_{i-1} - \vartheta_A)\Delta t$$

Euler–Cauchy-Polygonzüge für verschiedene Schrittweiten:



Die dünne Kurve ist die genaue analytische Lösung.

Rekurrentes neuronales Netz:



Rekurrente Netze: Abkühlungsgesetz

Formale Herleitung der rekursiven Gleichung:

Ersetze Differentialquotient durch **Differenzenquotient**

$$\frac{\mathrm{d}\vartheta(t)}{\mathrm{d}t} \approx \frac{\Delta\vartheta(t)}{\Delta t} = \frac{\vartheta(t+\Delta t) - \vartheta(t)}{\Delta t}$$

mit hinreichend kleinem Δt. Dann ist

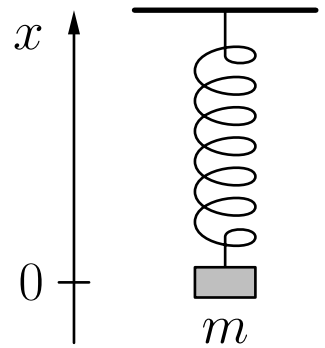
$$\vartheta(t+\Delta t) - \vartheta(t) = \Delta\vartheta(t) \approx -k(\vartheta(t) - \vartheta_A)\Delta t,$$

$$\vartheta(t+\Delta t) - \vartheta(t) = \Delta\vartheta(t) \approx -k\Delta t\vartheta(t) + k\vartheta_A\Delta t$$

und daher

$$\vartheta_i \approx \vartheta_{i-1} - k\Delta t\vartheta_{i-1} + k\vartheta_A\Delta t.$$

Rekurrente Netze: Masse an einer Feder



Zugrundeliegende physikalische Gesetze:

- **Hooke'sches Gesetz**: $F = c\Delta l = -cx$ (c ist eine federabhängige Konstante)
- **Zweites Newton'sches Gesetz**: $F = ma = m\ddot{x}$ (Kraft bewirkt eine Beschleunigung)

Resultierende Differentialgleichung:

$$m\ddot{x} = -cx \qquad \text{oder} \qquad \ddot{x} = -\frac{c}{m}x.$$

Rekurrente Netze: Masse an einer Feder

Allgemeine analytische Lösung der Differentialgleichung:

$$x(t) = a\sin(\omega t) + b\cos(\omega t)$$

mit den Parametern

$$\omega = \sqrt{\frac{c}{m}}, \qquad \begin{aligned} a &= x(t_0)\sin(\omega t_0) + v(t_0)\cos(\omega t_0), \\ b &= x(t_0)\cos(\omega t_0) - v(t_0)\sin(\omega t_0). \end{aligned}$$

Mit gegebenen Initialwerten $x(t_0) = x_0$ und $v(t_0) = 0$ und
der zusätzlichen Annahme $t_0 = 0$ bekommen wir den einfachen Ausdruck

$$x(t) = x_0 \cos\left(\sqrt{\frac{c}{m}}\, t\right).$$

Rekurrente Netze: Masse an einer Feder

Wandle Differentialgleichung in zwei gekoppelte Gleichungen um:

$$\dot{x} = v \qquad \text{and} \qquad \dot{v} = -\frac{c}{m}x.$$

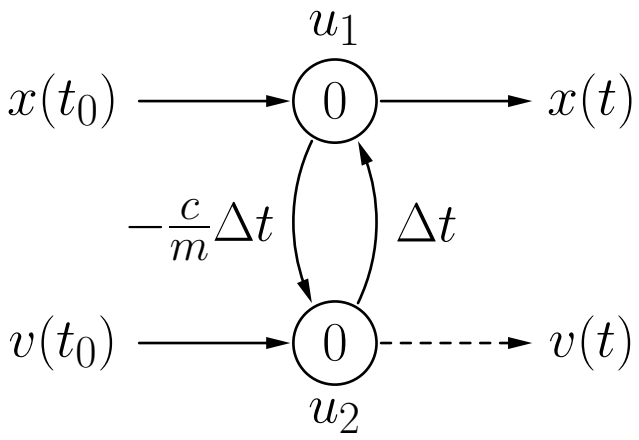
Approximiere Differentialquotient durch Differenzenquotient:

$$\frac{\Delta x}{\Delta t} = \frac{x(t+\Delta t) - x(t)}{\Delta t} = v \qquad \text{and} \qquad \frac{\Delta v}{\Delta t} = \frac{v(t+\Delta t) - v(t)}{\Delta t} = -\frac{c}{m}x$$

Resultierende rekursive Gleichungen:

$$\begin{aligned} x(t_i) &= x(t_{i-1}) + \Delta x(t_{i-1}) &&= x(t_{i-1}) + \Delta t \cdot v(t_{i-1}) \qquad \text{und} \\ v(t_i) &= v(t_{i-1}) + \Delta v(t_{i-1}) &&= v(t_{i-1}) - \frac{c}{m}\Delta t \cdot x(t_{i-1}). \end{aligned}$$

Rekurrente Netze: Masse an einer Feder



Neuron u_1: $f_{\text{net}}^{(u_1)}(v, w_{u_1u_2}) = w_{u_1u_2}v = -\dfrac{c}{m}\Delta t\ v$ und

$f_{\text{act}}^{(u_1)}(\text{act}_{u_1}, \text{net}_{u_1}, \theta_{u_1}) = \text{act}_{u_1} + \text{net}_{u_1} - \theta_{u_1},$

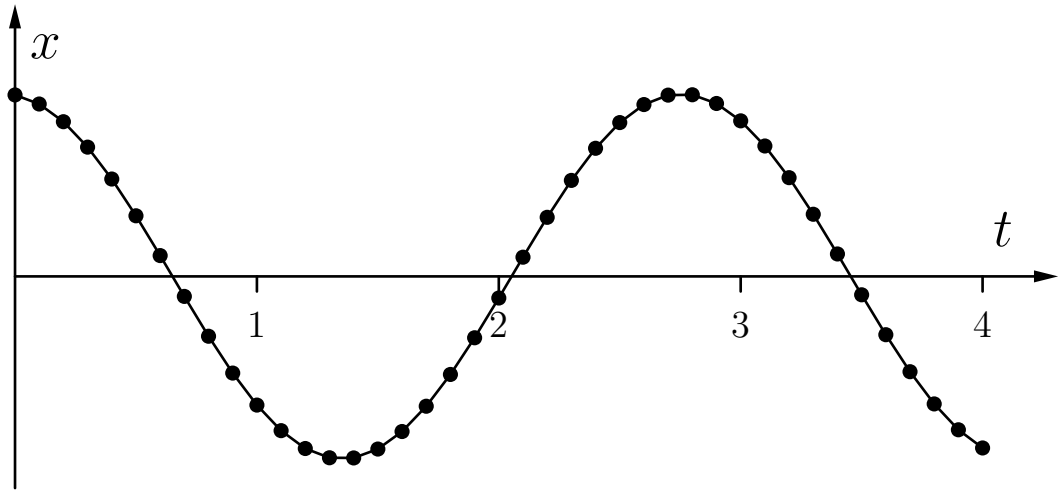
Neuron u_2: $f_{\text{net}}^{(u_2)}(x, w_{u_2u_1}) = w_{u_2u_1}x = \Delta t\ x$ und

$f_{\text{act}}^{(u_2)}(\text{act}_{u_2}, \text{net}_{u_2}, \theta_{u_2}) = \text{act}_{u_2} + \text{net}_{u_2} - \theta_{u_2}.$

Rekurrente Netze: Masse an einer Feder

Einige Berechnungsschritte des neuronalen Netzes:

t	v	x
0.0	0.0000	1.0000
0.1	−0.5000	0.9500
0.2	−0.9750	0.8525
0.3	−1.4012	0.7124
0.4	−1.7574	0.5366
0.5	−2.0258	0.3341
0.6	−2.1928	0.1148



- Die resultierende Kurve ist nah an der analytischen Lösung.
- Die Annäherung wird mit kleinerer Schrittweite besser.

Allgemeine Darstellung expliziter Differentialgleichungen n-ten Grades:

$$x^{(n)} = f(t, x, \dot{x}, \ddot{x}, \ldots, x^{(n-1)})$$

Einführung von $n - 1$ Zwischengrößen

$$y_1 = \dot{x}, \qquad y_2 = \ddot{x}, \qquad \ldots \qquad y_{n-1} = x^{(n-1)}$$

Gleichungssystem

$$\begin{aligned} \dot{x} &= y_1, \\ \dot{y}_1 &= y_2, \\ &\vdots \\ \dot{y}_{n-2} &= y_{n-1}, \\ \dot{y}_{n-1} &= f(t, x, y_1, y_2, \ldots, y_{n-1}) \end{aligned}$$

von n gekoppelten Differentialgleichungen ersten Grades.

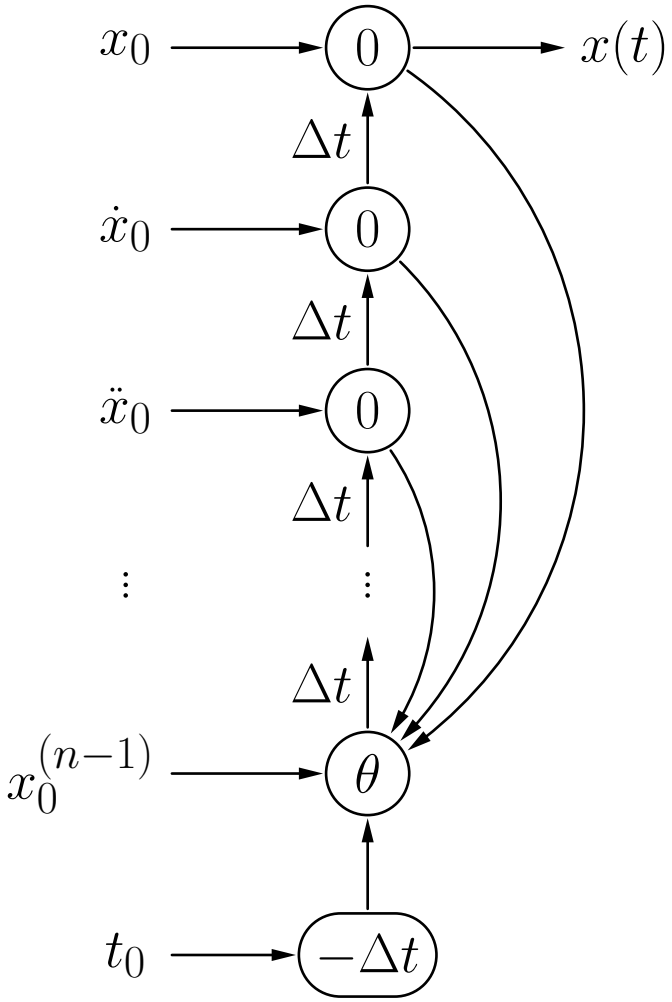
Rekurrente Netze: Differential Equations

Ersetze Differentialquotient durch Differenzenquotient, um die folgenden rekursiven Gleichungen zu erhalten:

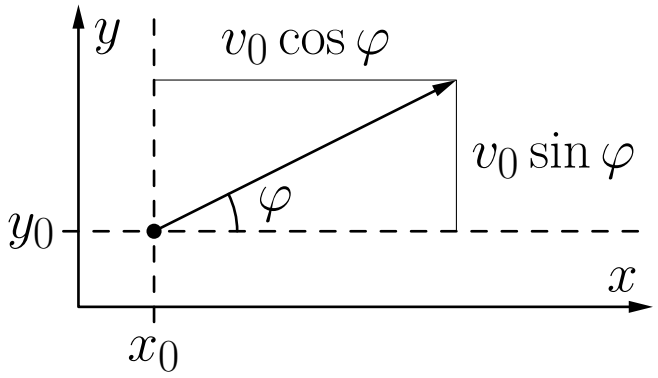
$$
\begin{aligned}
x(t_i) &= x(t_{i-1}) + \Delta t \cdot y_1(t_{i-1}), \\
y_1(t_i) &= y_1(t_{i-1}) + \Delta t \cdot y_2(t_{i-1}), \\
&\vdots \\
y_{n-2}(t_i) &= y_{n-2}(t_{i-1}) + \Delta t \cdot y_{n-3}(t_{i-1}), \\
y_{n-1}(t_i) &= y_{n-1}(t_{i-1}) + f(t_{i-1}, x(t_{i-1}), y_1(t_{i-1}), \ldots, y_{n-1}(t_{i-1}))
\end{aligned}
$$

- Jede dieser Gleichungen beschreibt die Aktualisierung eines Neurons.
- Das letzte Neuron benötigt eine spezielle Aktivierungsfunktion.

Rekurrente Netze: Differentialgleichungen



Rekurrente Netze: Schräger Wurf



Schräger Wurf eines Körpers.

Zwei Differentialgleichungen (eine für jede Koordinatenrichtung):

$$\ddot{x} = 0 \qquad \text{und} \qquad \ddot{y} = -g,$$

wobei $g = 9.81\,\mathrm{ms}^{-2}$.

Anfangsbedingungen $x(t_0) = x_0$, $y(t_0) = y_0$, $\dot{x}(t_0) = v_0 \cos \varphi$ und $\dot{y}(t_0) = v_0 \sin \varphi$.

Rekurrente Netze: Schräger Wurf

Führe Zwischenbedingungen ein:

$$v_x = \dot{x} \qquad \text{und} \qquad v_y = \dot{y}$$

um das System der folgenden Differentialgleichungen zu erhalten:

$$\dot{x} = v_x, \qquad \dot{v}_x = 0,$$

$$\dot{y} = v_y, \qquad \dot{v}_y = -g,$$

aus dem wir das System rekursiver Anpassungsformeln erhalten

$$x(t_i) = x(t_{i-1}) + \Delta t\ v_x(t_{i-1}), \qquad v_x(t_i) = v_x(t_{i-1}),$$

$$y(t_i) = y(t_{i-1}) + \Delta t\ v_y(t_{i-1}), \qquad v_y(t_i) = v_y(t_{i-1}) - \Delta t\ g.$$

Rekurrente Netze: Schräger Wurf

Bessere Beschreibung: Benutze **Vektoren** als Eingaben und Ausgaben

$$\ddot{\boldsymbol{r}} = -g\boldsymbol{e}_y,$$

wobei $\boldsymbol{e}_y = (0, 1)$.

Anfangsbedingungen sind $\boldsymbol{r}(t_0) = \boldsymbol{r}_0 = (x_0, y_0)$ und $\dot{\boldsymbol{r}}(t_0) = \boldsymbol{v}_0 = (v_0 \cos\varphi, v_0 \sin\varphi)$.

Führe eine **vektorielle** Zwischengröße $\boldsymbol{v} = \dot{\boldsymbol{r}}$ ein, um

$$\dot{\boldsymbol{r}} = \boldsymbol{v}, \qquad\qquad \dot{\boldsymbol{v}} = -g\boldsymbol{e}_y$$

zu erhalten.

Das führt zu den rekursiven Anpassungsregeln

$$\begin{aligned} \boldsymbol{r}(t_i) &= \boldsymbol{r}(t_{i-1}) + \Delta t\, \boldsymbol{v}(t_{i-1}), \\ \boldsymbol{v}(t_i) &= \boldsymbol{v}(t_{i-1}) - \Delta t\, g\boldsymbol{e}_y \end{aligned}$$

Rekurrente Netze: Schräger Wurf

Die Vorteile vektorieller Netze werden offensichtlich, wenn Reibung mit in Betracht gezogen wird:

$$\boldsymbol{a} = -\beta \boldsymbol{v} = -\beta \dot{\boldsymbol{r}}$$

β ist eine Konstante, die von Größe und Form des Körpers abhängt.
Dies führt zur Differentialgleichung

$$\ddot{\boldsymbol{r}} = -\beta \dot{\boldsymbol{r}} - g\boldsymbol{e}_y.$$

Führe die Zwischengröße $\boldsymbol{v} = \dot{\boldsymbol{r}}$ ein, um

$$\dot{\boldsymbol{r}} = \boldsymbol{v}, \qquad\qquad \dot{\boldsymbol{v}} = -\beta \boldsymbol{v} - g\boldsymbol{e}_y,$$

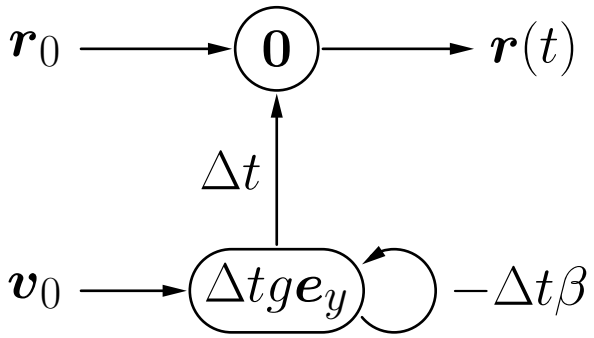
zu erhalten,

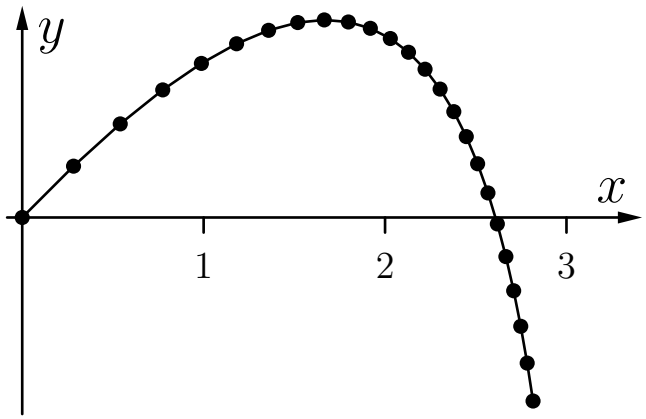
woraus wir die folgenden rekursiven Anpassungsformeln bekommen:

$$\begin{aligned} \boldsymbol{r}(t_i) &= \boldsymbol{r}(t_{i-1}) + \Delta t\ \boldsymbol{v}(t_{i-1}), \\ \boldsymbol{v}(t_i) &= \boldsymbol{v}(t_{i-1}) - \Delta t\ \beta\ \boldsymbol{v}(t_{i-1}) - \Delta t\ g\boldsymbol{e}_y. \end{aligned}$$

Rekurrente Netze: Schräger Wurf

Sich ergebendes rekurrentes neuronales Netz:





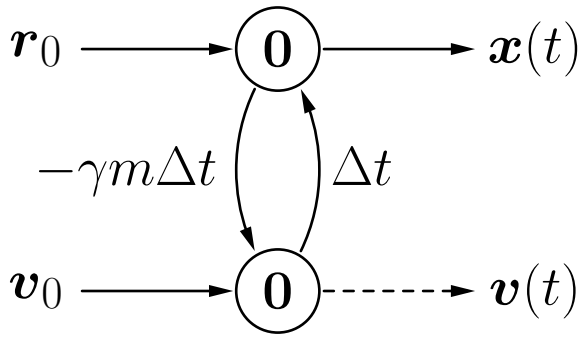
- Es gibt keine "seltsamen" Kopplungen wie in einem nicht-vektoriellen Netz.
- Man beachte die Abweichung von der Parabel, die durch die Reibung bewirkt wird.

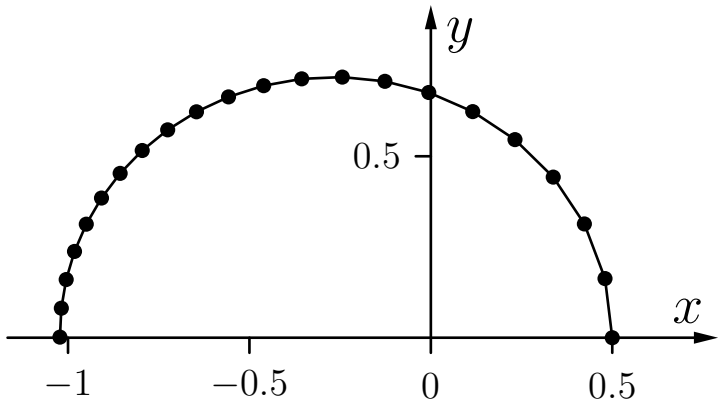
Rekurrente Netze: Umlaufbahnen der Planeten

$$\ddot{\boldsymbol{r}} = -\gamma m \frac{\boldsymbol{r}}{|\boldsymbol{r}|^3} \qquad \Rightarrow \qquad \dot{\boldsymbol{r}} = \boldsymbol{v} \qquad \dot{\boldsymbol{v}} = -\gamma m \frac{\boldsymbol{r}}{|\boldsymbol{r}|^3}$$

Rekursive Anpassungsregeln:

$$\begin{aligned} \boldsymbol{r}(t_i) &= \boldsymbol{r}(t_{i-1}) + \Delta t\, \boldsymbol{v}(t_{i-1}) \\ \boldsymbol{v}(t_i) &= \boldsymbol{v}(t_{i-1}) - \Delta t\, \gamma m \frac{\boldsymbol{r}(t_{i-1})}{|\boldsymbol{r}(t_{i-1})|^3} \end{aligned}$$

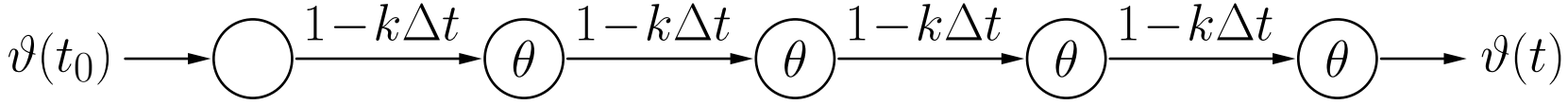




Rekurrente Netze: Backpropagation über die Zeit

Idee: Entfalte das Netzwerk zwischen Trainingsmustern,
d.h. lege ein Neuron für jeden Zeitpunkt an.

Beispiel: **Newton'sches Abkühlungsgesetz**



Entfalten in vier Schritten. Es ist $\theta = -k\vartheta_A\Delta t$.

- Training: Standard-Backpropagation im entfalteten Netzwerk.
- Alle Anpassungen beziehen sich auf dasselbe Gewicht.
- Anpassungen werden ausgeführt, wenn das erste Neuron erreicht wird.

Überwachtes Lernen / Support Vector Machines

Überwachtes Lernen, Diagnosesystem für Krankheiten

Trainingsdaten: Expressionsprofile von Patienten mit bekannter Diagnose

Durch die bekannte Diagnose ist eine Struktur in den Daten vorgegeben, die wir auf zukünftige Daten verallgemeinern wollen.

Lernen/Trainieren: Leite aus den Trainingsdaten eine Entscheidungsregel ab, die die beiden Klassen voneinander trennt.

Generalisierungsfähigkeit: wie gut ist die Entscheidungsregel darin, zukünftige Patienten zu diagnostizieren?

Ziel: finde eine Entscheidungsregel mit hoher Generalisierungsfähigkeit!

Lernen von Beispielen

Gegeben: $X = \{x_i, y_i\}_{i=1}^{n}$, Trainingsdaten von Patienten mit bekannter Diagnose

bestehend aus:

$$x_i \in \mathbb{R}^g \text{ (Punkte, Expressionsprofile)}$$

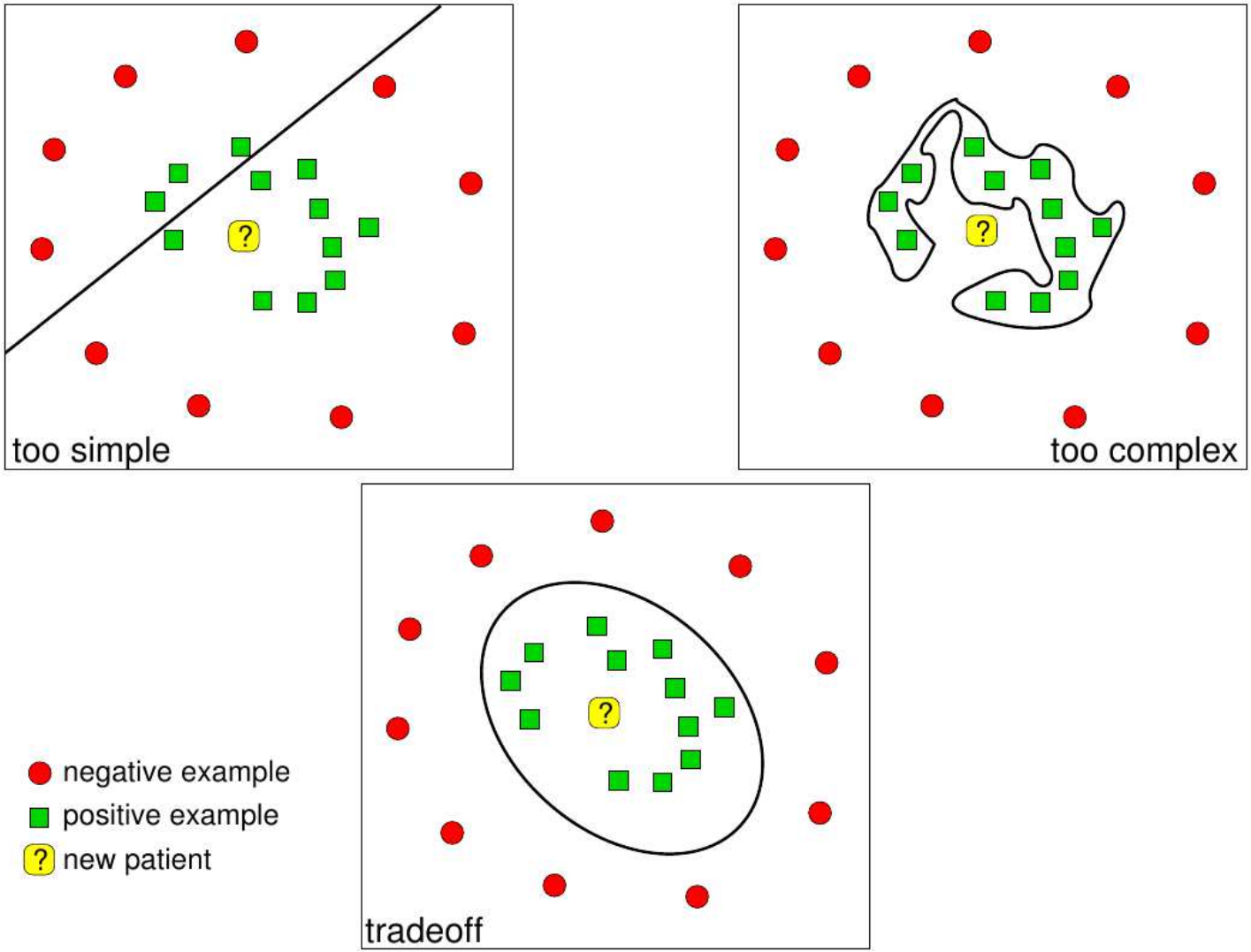
$$y_i \in \{+1, -1\} \text{ (Klassen, 2 Arten von Krebs)}$$

Entscheidungsfunktion:

$$f_X : \mathbb{R}^g \to \{+1, -1\}$$

$$\text{Diagnose} = f_X(\text{neuer Patient})$$

Underfitting / Overfitting



Lineare Trennung der Trainingsdaten

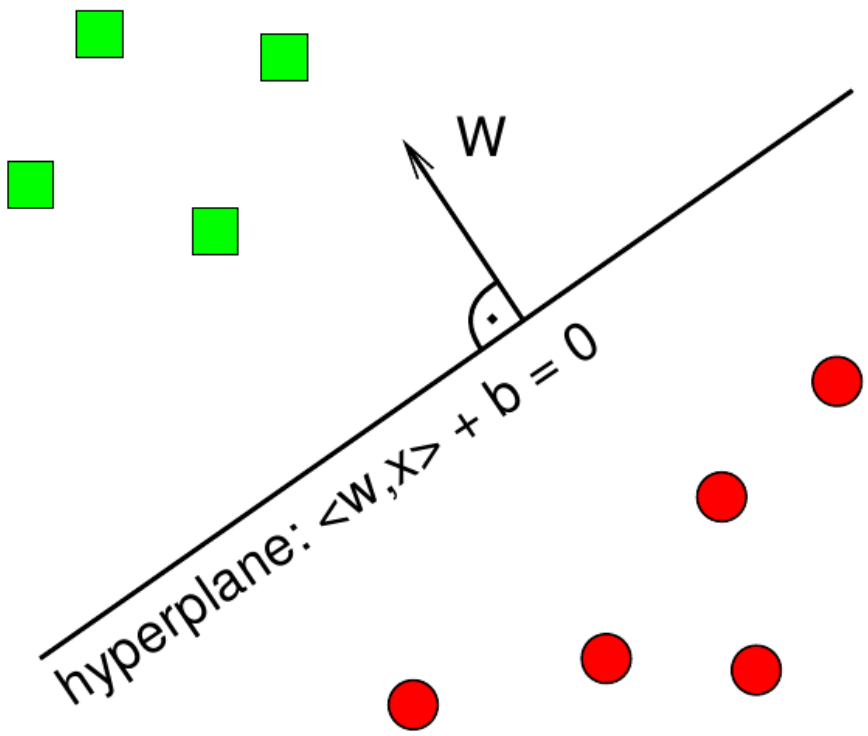
Wir fangen mit linearer Trennung an und vergrößern die Komplexität in einem zweiten Schritt durch Kernel-Funktionen.

Eine trennende Hyperebene ist definiert durch

- den Normalenvektor w und
- die Verschiebung b:

Hyperebene $\mathcal{H} = \{x | \langle w, x \rangle + b = 0\}$

$\langle \cdot, \cdot \rangle$ nennt man inneres Produkt
oder Skalarprodukt



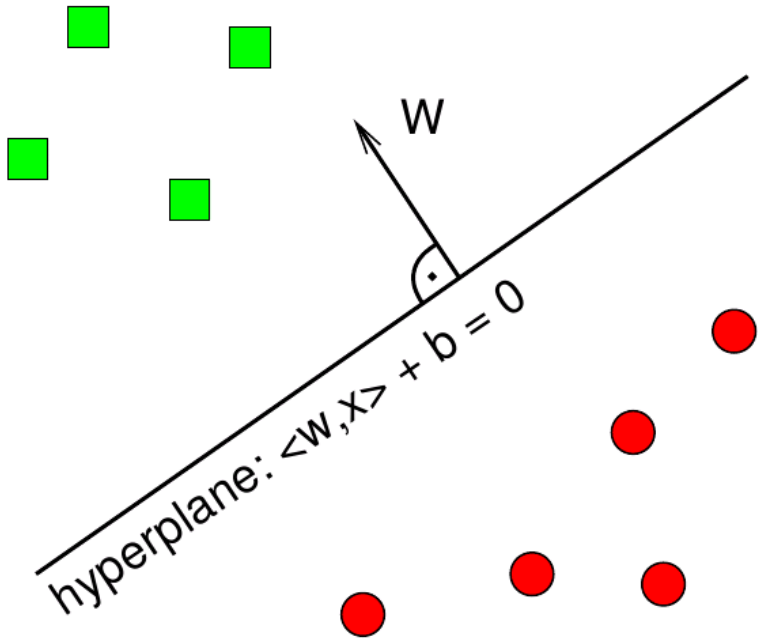
Vorhersage der Klasse eines neuen Punktes

Training: Wähle w und b so, daß die Hyperebene die Trainingsdaten trennt.

Vorhersage: Auf welcher Seite der Hyperebene liegt der neue Punkt?

Punkte in Richtung des Normalenvektors diagnostizieren wir als POSITIV.

Punkte auf der anderen Seite diagnostizieren wir als NEGATIV.



Motivation

Ursprung in statistischer Lerntheorie; Klasse optimaler Klassifikatoren

Zentrales Problem der statistischen Lerntheorie: Generalisierungsfähigkeit: Wann führt ein niedriger Trainingsfehler zu einem niedrigen echten Fehler?

Zweiklassenproblem:
Klassifikationsvorgang $\equiv$ Zuordnungsfunktion $f(x, u) : \; x \rightarrow y \in \{+1, -1\}$
x: Muster aus einer der beiden Klassen
u: Parametervektor des Klassifikators

Lernstichprobe mit l Beobachtungen $x_1, x_2, \ldots, x_l$
mit entsprechender Klassenzugehörigkeit $y_1, y_2, \ldots, y_l$
$\rightarrow$ das empirische Risiko (Fehlerrate) für gegebenen Trainingsdatensatz:

$$R_{emp}(u) \; = \; \frac{1}{2l} \sum_{i=1}^{l} |y_i - f(x_i, u)| \; \in [0, 1]$$

Viele Klassifikatoren, z.B. neuronale Netze, minimieren das empirische Risiko

Erwartungswert des Zuordnungsfehlers (expected risk):

$$R(u) = E\{R_{test}(u)\} = E\{\frac{1}{2}|y - f(x,u)|\} = \int \frac{1}{2}|y - f(x,u)|p(x,y)\, dxdy$$

$p(x,y)$: Verteilungsdichte aller möglichen Samples x mit entsprechender Klassenzugehörigkeit y (Dieser Ausdruck nicht direkt auswertbar, da $p(x,y)$ nicht zur Verfügung steht)

Optimale Musterklassifikation:
Deterministische Zuordnungsfunktion $f(x,u): \; x \rightarrow y \in \{+1,-1\}$ gesucht, so dass das Expected Risk minimiert wird

Zentrale Frage der Musterklassifikation:

> Wie nah ist man nach l Trainingsbeispielen am *echten* Fehler? Wie gut kann aus dem empirischen Risiko $R_{emp}(u)$ das echte Risiko $R(u)$ abgeschätzt werden? (Structural Risk Minimization statt Empirical Risk Minimization)

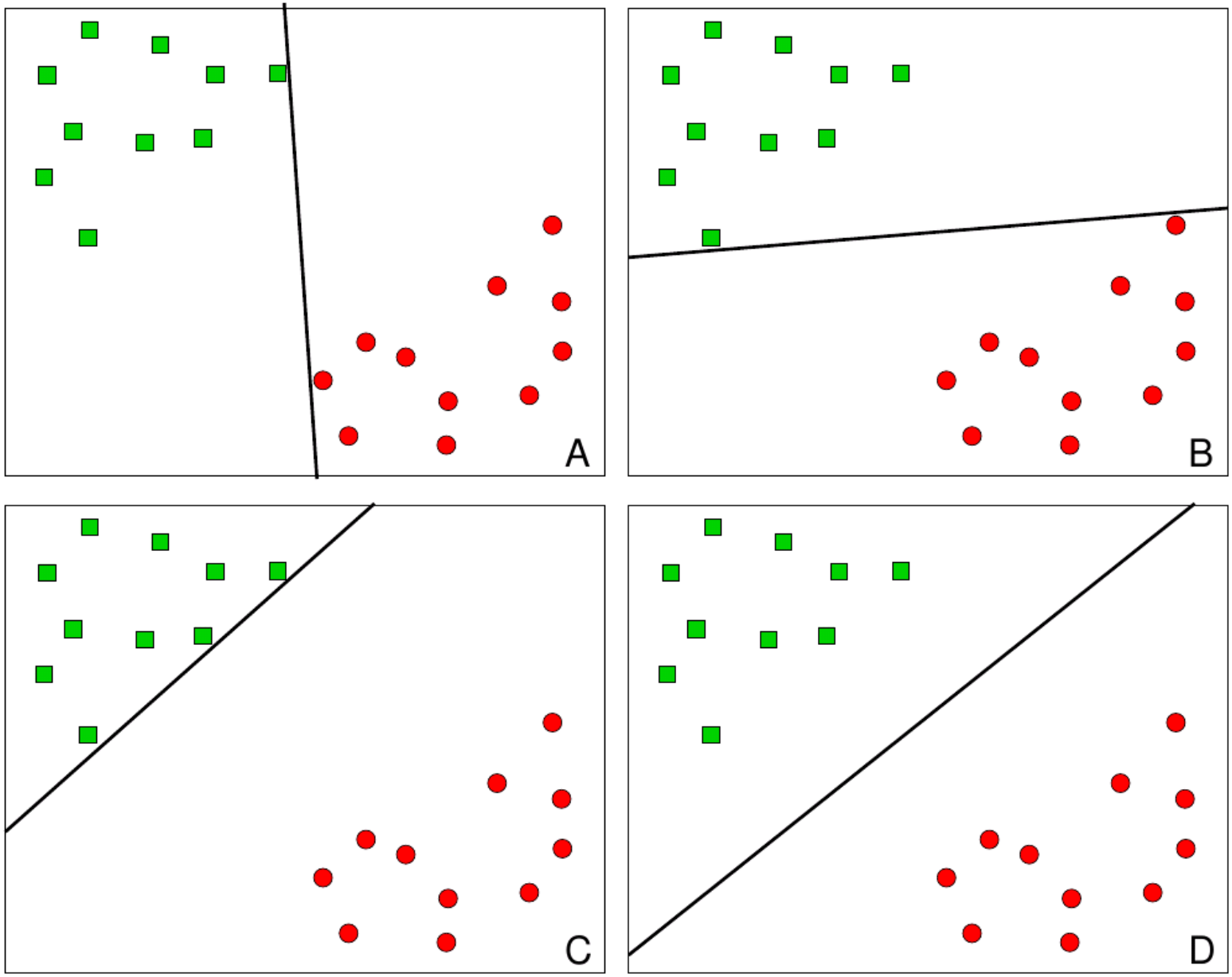
Antwort durch Lerntheorie von Vapnik-Chervonenkis $\rightarrow$ SVM

SVM für linear trennbare Klassen

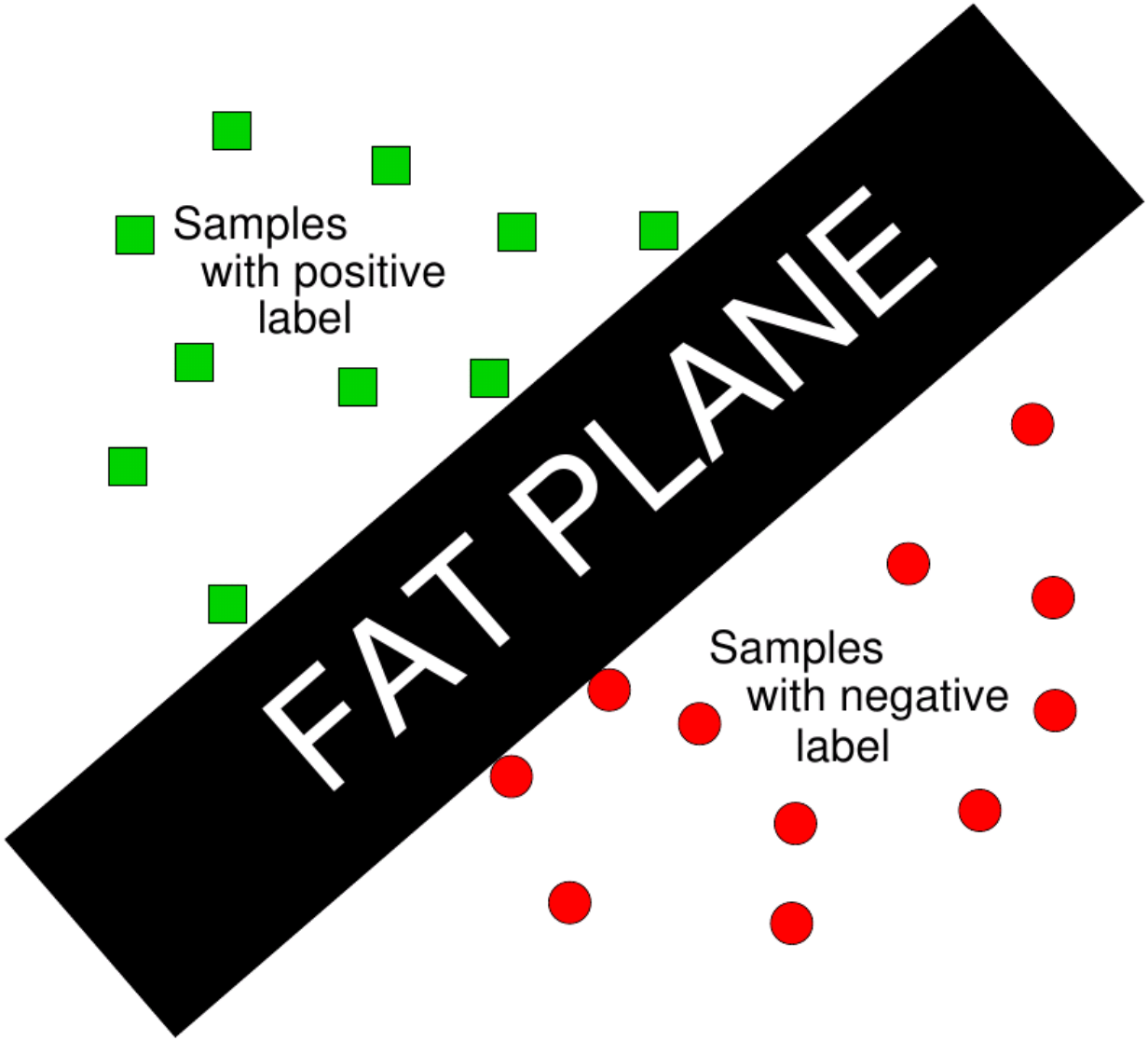
Bisherige Lösung:

- Allgemeine Hyperebene: $wx + b = 0$
- Klassifikation: $\text{sgn}(wx + b)$
- Training z.B. mittels Perzeptron-Algorithmus (iteratives Lernen, Korrektur nach jeder Fehlklassifikation; keine Eindeuigkeit der Lösung)

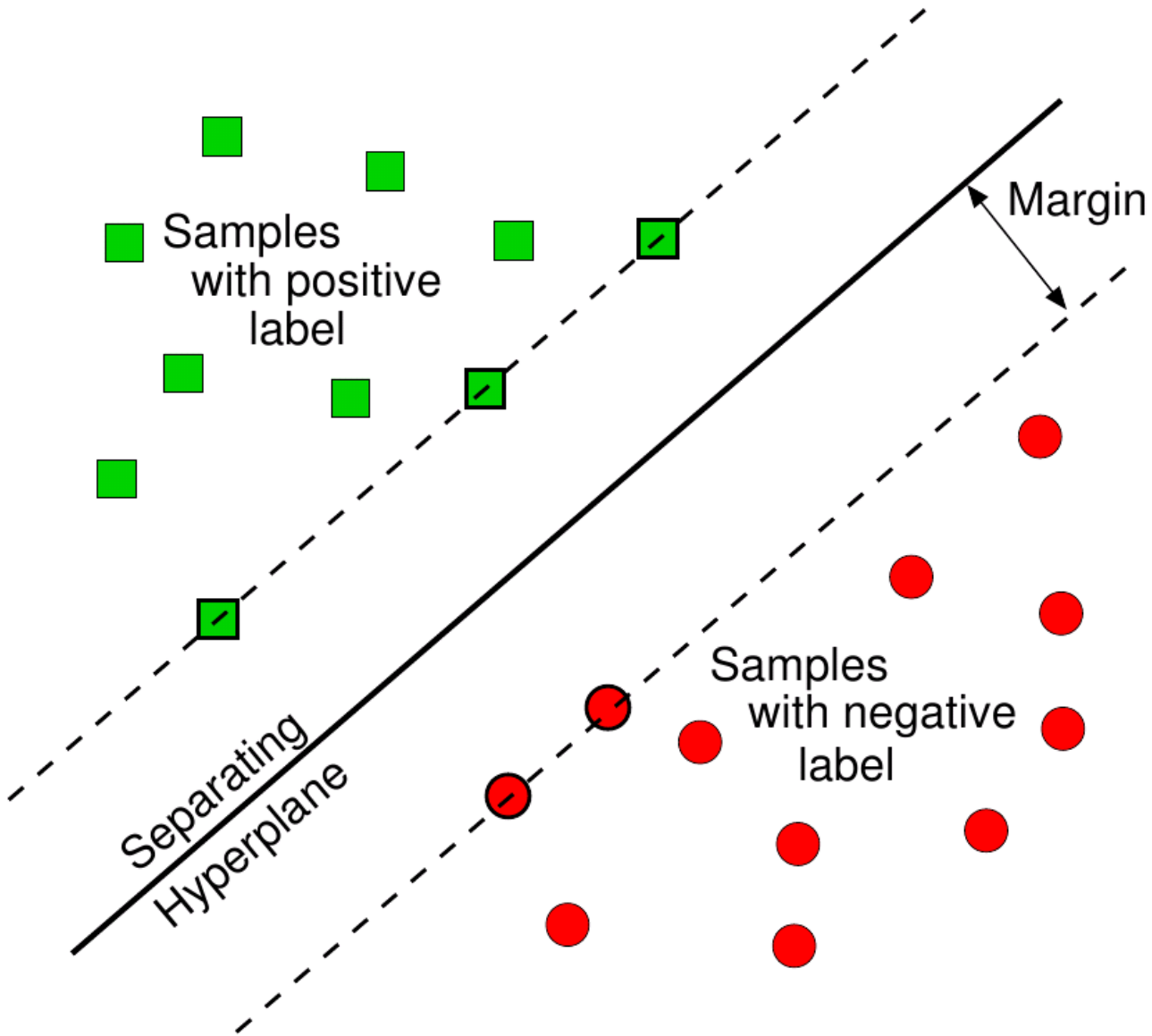
Welche Hyperebene ist die beste? Und warum?



Keine scharfe Trennung, sondern eine . . .

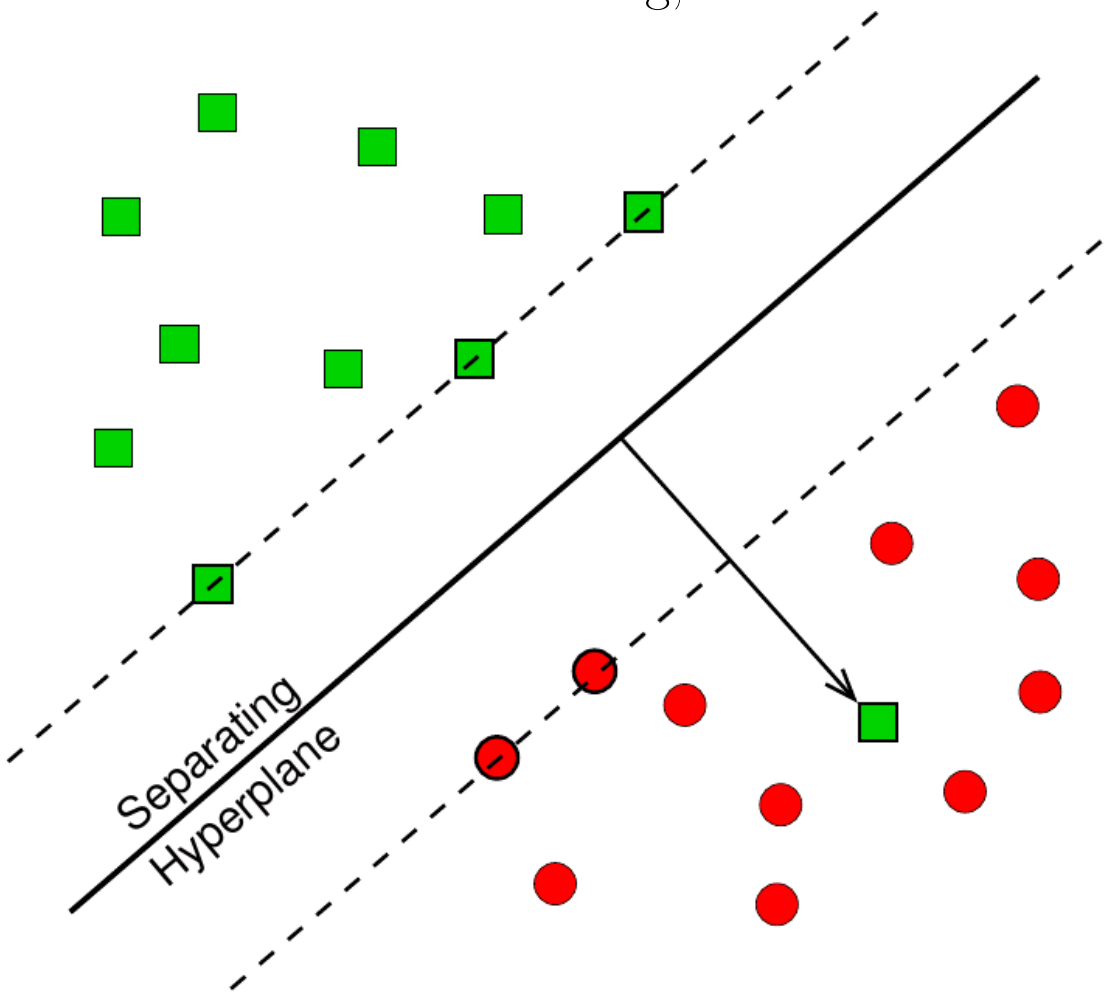


Trenne die Trainingsdaten mit maximaler Trennspanne



Trenne die Trainingsdaten mit maximaler Trennspanne

Versuche eine lineare Trennung, aber lasse Fehler zu:



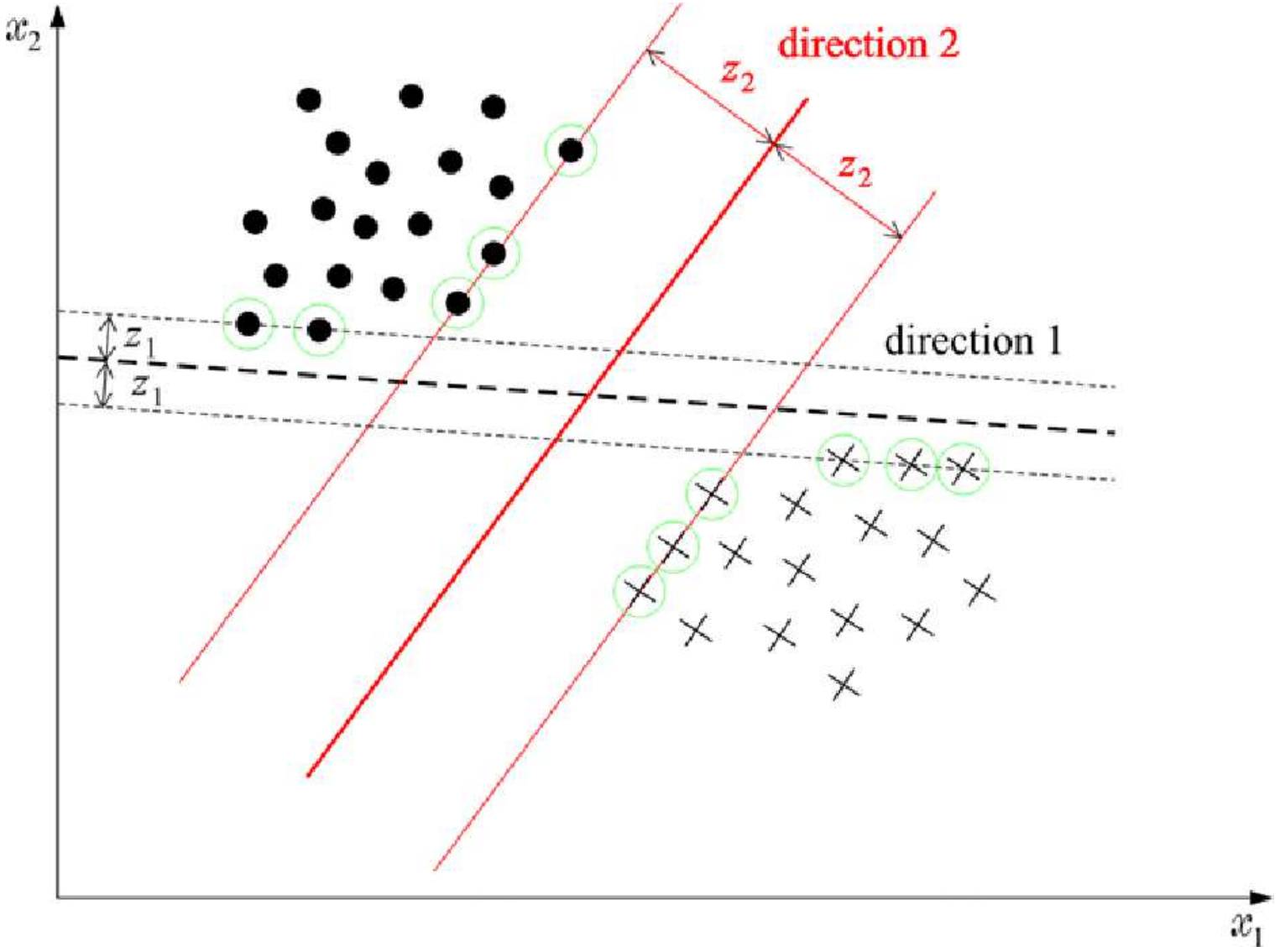
Strafe für Fehler: Abstand zur Hyperebene multipliziert mit Fehlergewicht C

SVM für linear trennbare Klassen

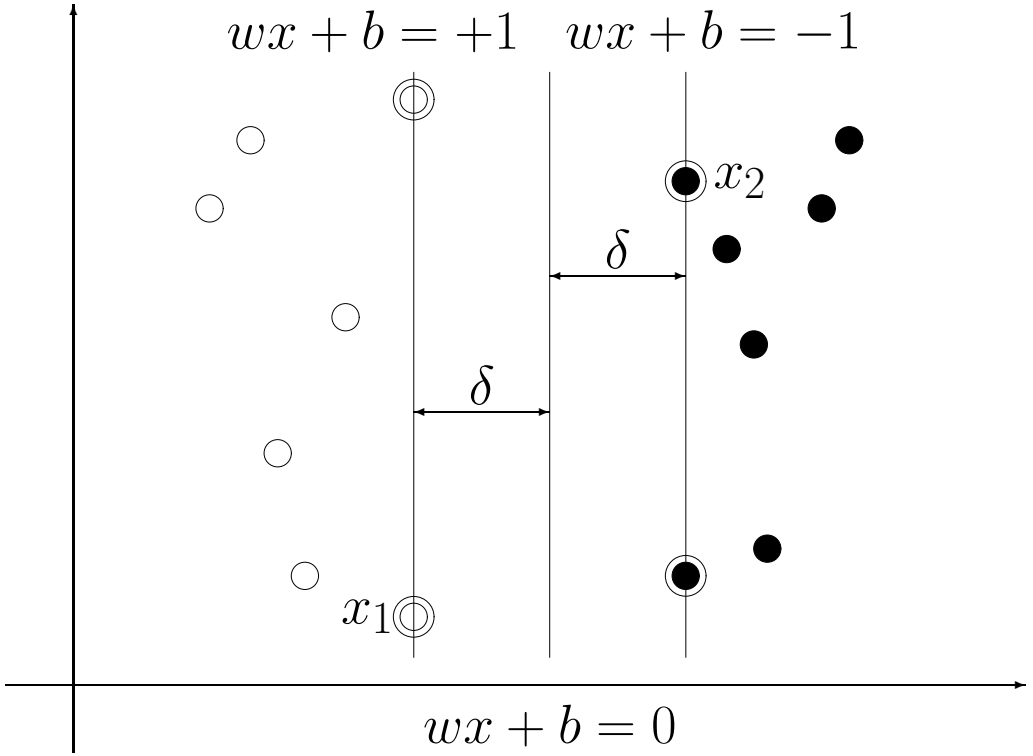
- Bei SVM wird eine trennende Hyperebene mit maximalem Rand gesucht. *Optimal:* diejenige mit größtem 2δ von allen möglichen Trennebenen.
- Anschaulich sinnvoll (Bei konstanter Intraklassenstreuung wächst Klassifikationssicherheit mit wachsendem Interklassenabstand)
- Theoretisch sind SVM durch statistische Lerntheorie begründet

SVM für linear trennbare Klassen

Large-Margin Klassifikator: Trennlinie 2 besser als Trennlinie 1



SVM für linear trennbare Klassen



Trainingsdaten werden korrekt klassifiziert, falls:

$$y_i(wx_i + b) > 0$$

Invarianz dieses Ausdrucks gegenüber einer positiven Skalierung führt zu:

$$y_i(wx_i + b) \geq 1$$

mit den kanonischen Hyperebenen:

$$\begin{cases} wx_i + b = +1; \text{(Klasse mit } y_i = +1) \\ wx_i + b = -1; \text{(Klasse mit } y_i = -1) \end{cases}$$

Der Abstand zwischen den kanonischen Hyperebenen ergibt sich durch Projektion von $x_1 - x_2$ auf den Einheitsnormalenvektor $\frac{w}{||w||}$:

$$2\delta = \frac{2}{||w||}; \quad \text{d.h. } \delta = \frac{1}{||w||}$$

$\rightarrow$ Maximierung von $\delta \equiv$ Minimierung von $||w||^2$

SVM für linear trennbare Klassen

Optimale Trennebene durch Minimierung einer quadratischen Funktion unter linearen Nebenbedingungen:

Primales Optimierungsproblem:
minimiere: $J(w,b) = \frac{1}{2}||w||^2$
unter Nebenbedingungen $\forall i \; [y_i(wx_i + b) \geq 1], \; i = 1, 2, \ldots, l$

Einführung einer Lagrange-Funktion:

$$L(w,b,\alpha) = \frac{1}{2}||w||^2 - \sum_{i=1}^{l} \alpha_i [y_i(wx_i + b) - 1]; \quad \alpha_i \geq 0$$

führt zum *dualen Problem*: maximiere $L(w,b,\alpha)$ bezüglich α, unter den Nebenbedingungen:

$$\frac{\partial L(w,b,\alpha)}{\partial w} = 0 \quad \Longrightarrow \quad w = \sum_{i=1}^{l} \alpha_i y_i x_i$$

$$\frac{\partial L(w,b,\alpha)}{\partial b} = 0 \quad \Longrightarrow \quad \sum_{i=1}^{l} \alpha_i y_i = 0$$

SVM für linear trennbare Klassen

Einsetzen dieser Terme in $L(w, b, \alpha)$:

$$\begin{aligned}
L(w,b,\alpha) &= \frac{1}{2}||w||^2 - \sum_{i=1}^{l} \alpha_i [y_i(wx_i + b) - 1] \\
&= \frac{1}{2} w \cdot w - w \cdot \sum_{i=1}^{l} \alpha_i y_i x_i - b \cdot \sum_{i=1}^{l} \alpha_i y_i + \sum_{i=1}^{l} \alpha_i \\
&= \frac{1}{2} w \cdot w - w \cdot w + \sum_{i=1}^{l} \alpha_i \\
&= -\frac{1}{2} w \cdot w + \sum_{i=1}^{l} \alpha_i \\
&= -\frac{1}{2} \sum_{i=1}^{l} \sum_{j=1}^{l} y_i y_j \alpha_i \alpha_j x_i x_j + \sum_{i=1}^{l} \alpha_i
\end{aligned}$$

SVM für linear trennbare Klassen

Duales Optimierungsproblem:

maximiere: $L'(\alpha) = \sum_{i=1}^{l} \alpha_i - \frac{1}{2} \sum_{i=1}^{l} \sum_{j=1}^{l} y_i y_j \alpha_i \alpha_j x_i x_j$

unter Nebenbedingungen $\alpha_i \geq 0$ und $\sum_{i=1}^{l} y_i \alpha_i = 0$

Dieses Optimierungsproblem kann mithilfe der konvexen quadratischen Programmierung numerisch gelöst werden

SVM für linear trennbare Klassen

Lösung des Optimierungsproblems:

$$\begin{aligned} w^* &= \sum_{i=1}^{l} \alpha_i y_i x_i = \sum_{x_i \in SV} \alpha_i y_i x_i \\ b^* &= -\frac{1}{2} \cdot w^* \cdot (x_p + x_m) \\ &\quad \text{für beliebiges } x_p \in SV,\ y_p = +1,\ \text{und } x_m \in SV,\ y_m = -1 \end{aligned}$$

wobei

$$SV = \{x_i \mid \alpha_i > 0,\ i = 1, 2, \ldots, l\}$$

die Menge aller Support-Vektoren darstellt

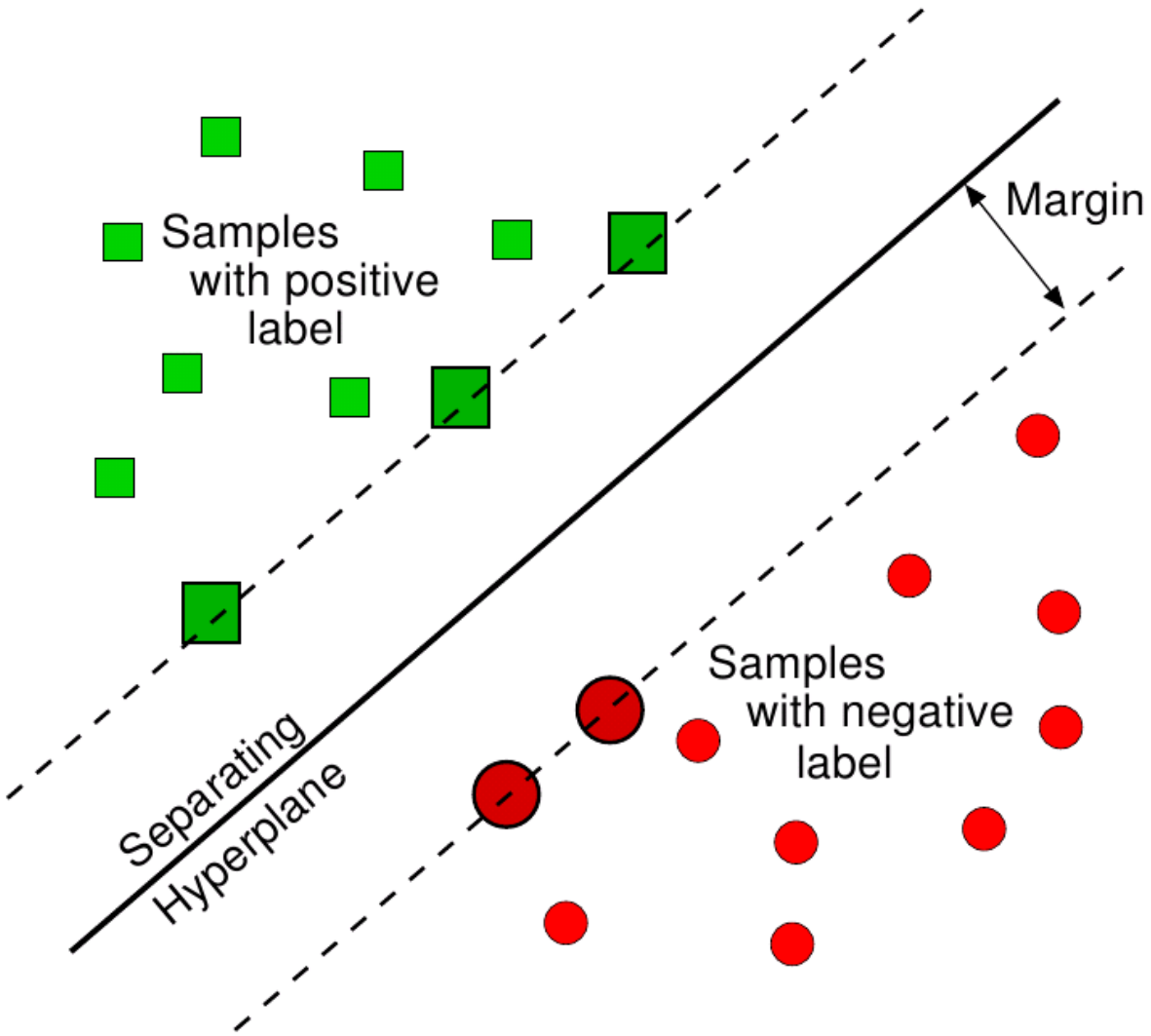
Klassifikationsregel:

$$\operatorname{sgn}(w^* x + b^*) = \operatorname{sgn}[(\sum_{x_i \in SV} \alpha_i y_i x_i) x + b^*]$$

Die Klassifikation hängt nur von den Support-Vektoren ab!

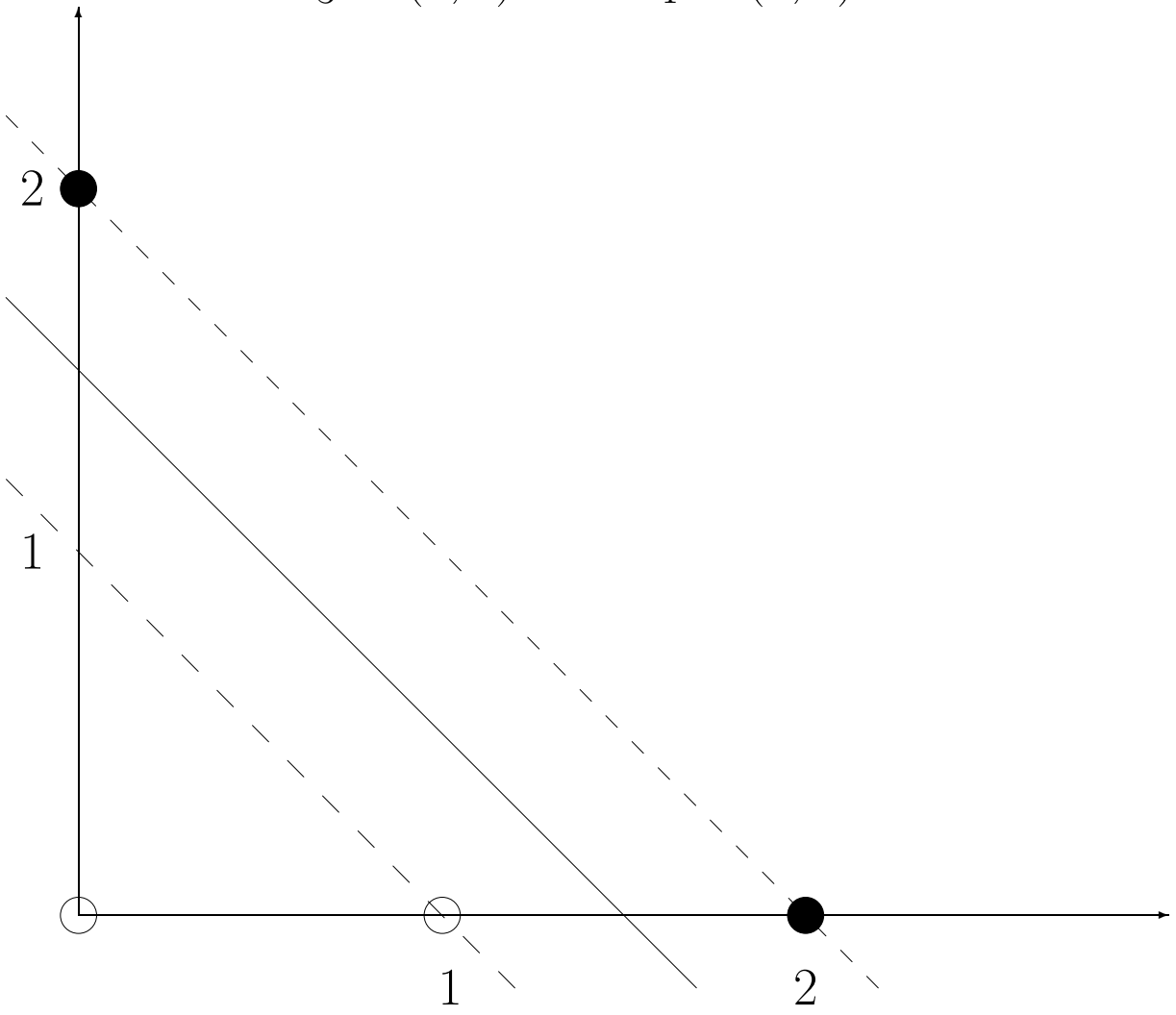
SVM für linear trennbare Klassen

Beispiel: Support-Vektoren



SVM für linear trennbare Klassen

Beispiel: Klasse +1 enthält $x_1 = (0, 0)$ und $x_2 = (1, 0)$;
Klasse -1 enthält $x_3 = (2, 0)$ und $x_4 = (0, 2)$



SVM für linear trennbare Klassen

Das duale Optimierungsproblem lautet:

maximiere: $L'(\alpha) = (\alpha_1 + \alpha_2 + \alpha_3 + \alpha_4) - \frac{1}{2}(\alpha_2^2 - 4\alpha_2\alpha_3 + 4\alpha_3^2 + 4\alpha_4^2)$
unter Nebenbedingungen $\alpha_i \geq 0$ und $\alpha_1 + \alpha_2 - \alpha_3 - \alpha_4 = 0$

Lösung:

$$\alpha_1 = 0, \quad \alpha_2 = 1, \quad \alpha_3 = \tfrac{3}{4}, \quad \alpha_4 = \tfrac{1}{4}$$
$$SV = \{(1,0),\ (2,0),\ (0,2)\}$$
$$w^* = 1 \cdot (1,0) - \tfrac{3}{4} \cdot (2,0) - \tfrac{1}{4} \cdot (0,2) = (-\tfrac{1}{2}, -\tfrac{1}{2})$$
$$b^* = -\tfrac{1}{2} \cdot (-\tfrac{1}{2}, -\tfrac{1}{2}) \cdot ((1,0) + (2,0)) = \tfrac{3}{4}$$

Optimale Trennlinie: $x + y = \frac{3}{2}$

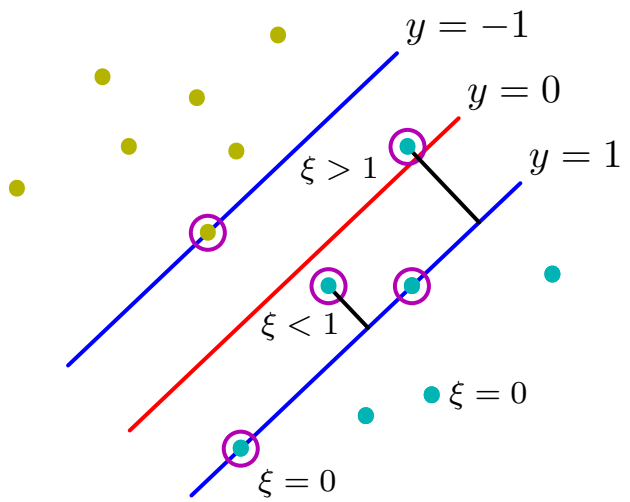
SVM für linear trennbare Klassen

Beobachtungen:

- Für die Support-Vektoren gilt: $\alpha_i > 0$
- Für alle Trainingsdaten außerhalb des Randes gilt: $\alpha_i = 0$
- Support-Vektoren bilden eine "sparse" Darstellung der Stichprobe und sind ausreichend für die Klassifikation
- Die Lösung entspricht dem globalen Optimum und ist eindeutig
- Der Optimierungsvorgang benötigt nur Skalarprodukte $x_i x_j$

SVM für nicht linear separierbare Klassen

In diesem Beispiel existiert keine Trennlinie so dass $\forall i \; [y_i(wx_i + b) \geq 1]$



Alle drei Fälle können interpretiert werden als: $y_i(wx_i + b) \; \geq \; 1 - \xi_i$

A) $\xi_i = 0$
B) $0 < \xi_i \leq 1$
C) $\xi_i > 1$

Drei mögliche Fälle:

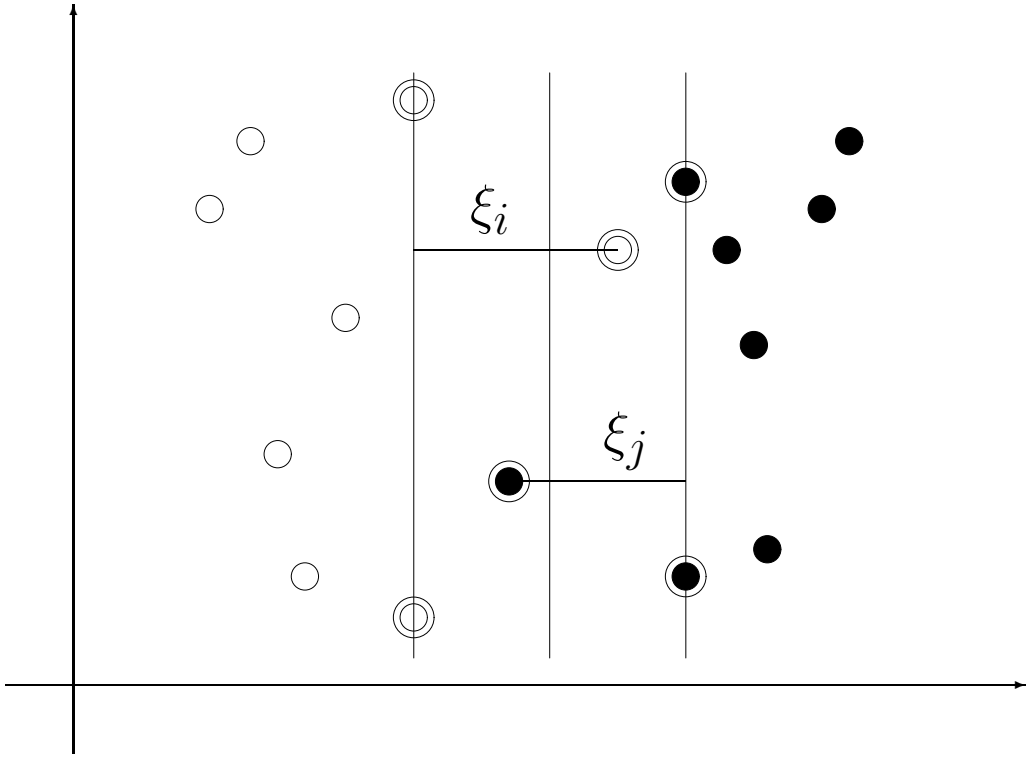
A) Vektoren außerhalb des Bandes, die korrekt klassifiziert werden, d.h.
$$y_i(wx_i + b) \; \geq \; 1$$
B) Vektoren innerhalb des Bandes, die korrekt klassifiziert werden, d.h.
$$0 \; \leq \; y_i(wx_i + b) \; < \; 1$$
C) Vektoren, die falsch klassifiziert werden, d.h.
$$y_i(wx_i + b) \; < \; 0$$

SVM für nicht linear separierbare Klassen

Motivation für Verallgemeinerung:

- Keine Lösung mit bisherigem Ansatz für nicht trennbare Klassen
- Verbesserung der Generalisierung bei Ausreißern in der Randzone

Soft-Margin SVM: Einführung von "slack"-Variablen



SVM für nicht linear separierbare Klassen

Bestrafen von Randverletzungen via "slack"-Variablen

Primales Optimierungsproblem:

minimiere: $J(w, b, \xi) = \frac{1}{2}||w||^2 + C \sum_{i=1}^{l} \xi_i$

unter Nebenbedingungen $\forall i \; [y_i(wx_i + b) \geq 1 - \xi_i, \; \xi_i \geq 0]$

Duales Optimierungsproblem:

maximiere: $L'(\alpha) = \sum_{i=1}^{l} \alpha_i - \frac{1}{2} \sum_{i=1}^{l} \sum_{j=1}^{l} y_i y_j \alpha_i \alpha_j x_i x_j$

unter Nebenbedingungen $0 \leq \alpha_i \leq C$ und $\sum_{i=1}^{l} y_i \alpha_i = 0$

(Weder die slack-Variablen noch deren Lagrange-Multiplier tauchen im dualen Optimierungsproblem auf!)

Einziger Unterschied zum linear trennbaren Fall: Konstante C in den Nebenbedingungen

SVM für nicht linear separierbare Klassen

Lösung des Optimierungsproblems:

$$\begin{aligned} w^* &= \sum_{i=1}^{l} \alpha_i y_i x_i = \sum_{x_i \in SV} \alpha_i y_i x_i \\ b^* &= y_k(1 - \xi_k) - w^* x_k; \quad k = \arg\max_i \alpha_i \end{aligned}$$

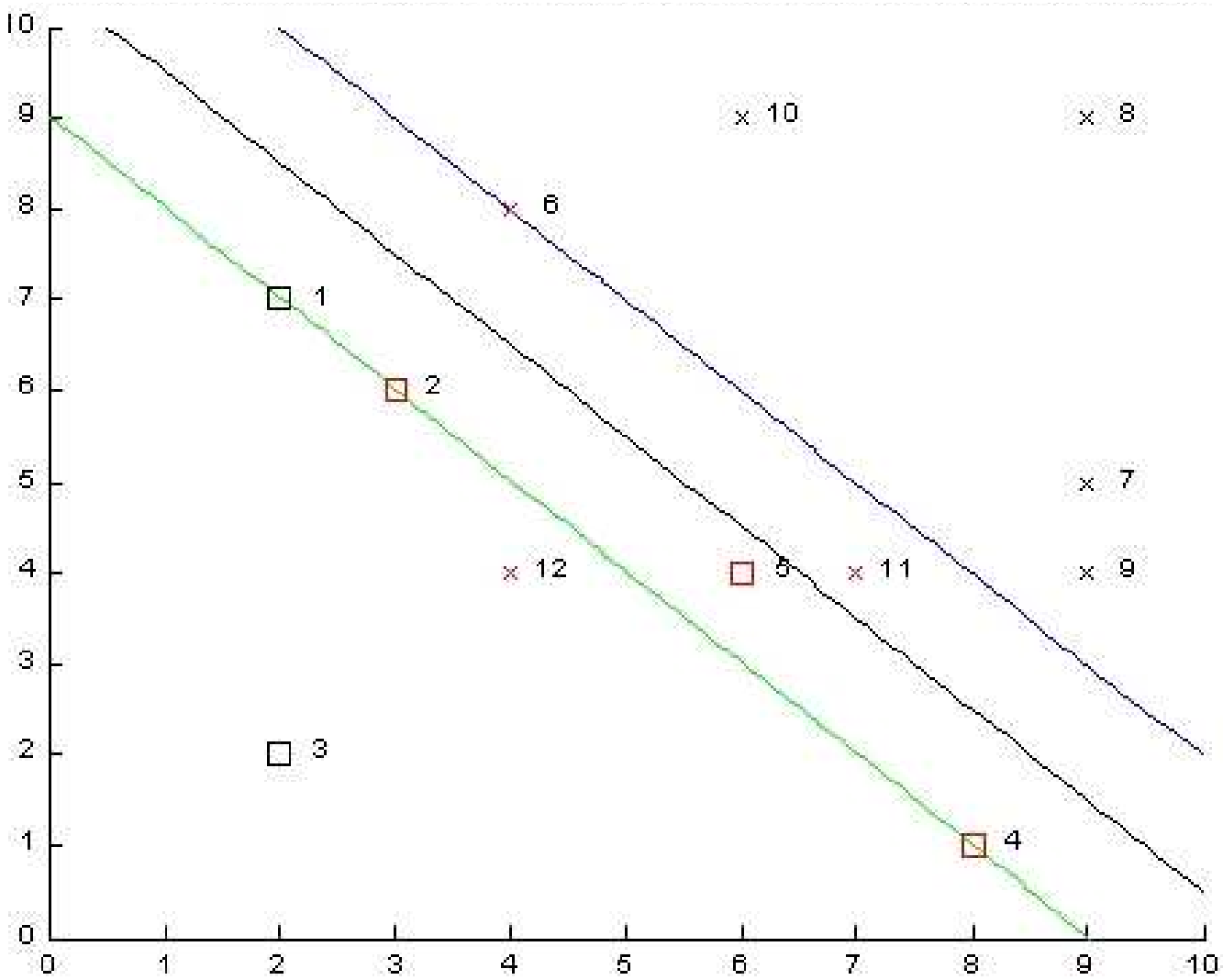
wobei

$$SV = \{x_i \mid \alpha_i > 0,\ i = 1, 2, \ldots, l\}$$

die Menge aller Support-Vektoren darstellt

SVM für nicht linear separierbare Klassen

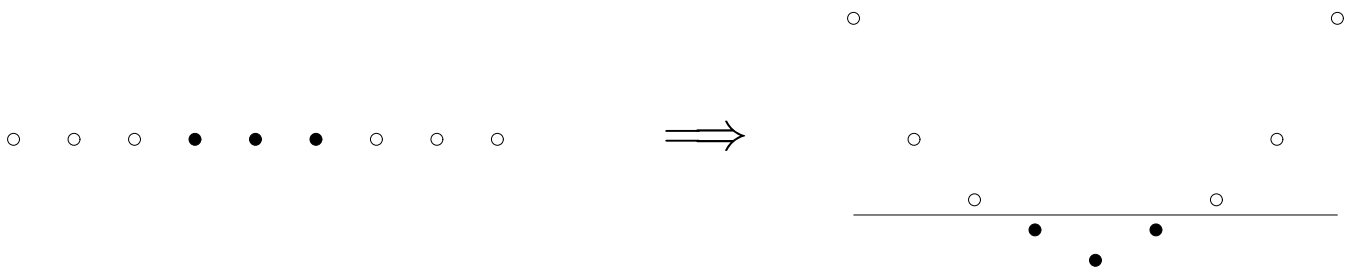
Beispiel: nicht linear trennbare Klassen



Nichtlineare SVM

Nichtlineare Klassengrenzen: Hyperebene → keine hohe Genauigkeit

Beispiel: Transformation $\Psi(x) = (x, x^2) \rightarrow C_1$ und C_2 linear trennbar



Idee: Merkmale $x \in \Re^n$ durch

$$\Psi : \quad \Re^n \longrightarrow \Re^m$$

in einen höherdimensionalen Raum $\Re^m$, $m > n$, transformieren und in $\Re^m$ eine optimale lineare Trennebene finden

Transformation Ψ steigert die lineare Trennbarkeit!

Trennende Hyperebene in $\Re^m \equiv$ nichtlineare Trennfläche in $\Re^n$

Nichtlineare SVM

Problem: Sehr hohe Dimension des Merkmalsraums $\Re^m$
Z.B. Polynome p-ten Grades über $\Re^n \rightarrow \Re^m$, $m = O(n^p)$

Trick mit Kernelfunktionen:

Ursprünglich in $\Re^n$: nur Skalarprodukte $x_i x_j$ erforderlich
neu in $\Re^m$: nur Skalarprodukte $\Psi(x_i)\Psi(x_j)$ erforderlich

Lösung:
$\Psi(x_i)\Psi(x_j)$ müssen nicht explizit ausgerechnet werden, sondern können mit reduzierter Komplexität mit Kernelfunktionen ausgedrückt werden

$$K(x_i, x_j) \;=\; \Psi(x_i)\Psi(x_j)$$

Nichtlineare SVM

Beispiel: Für die Transformation $\Psi : \Re^2 \longrightarrow \Re^6$

$$\Psi((y_1, y_2)) = (y_1^2,\ y_2^2,\ \sqrt{2}y_1,\ \sqrt{2}y_2,\ \sqrt{2}y_1y_2,\ 1)$$

berechnet die Kernelfunktion

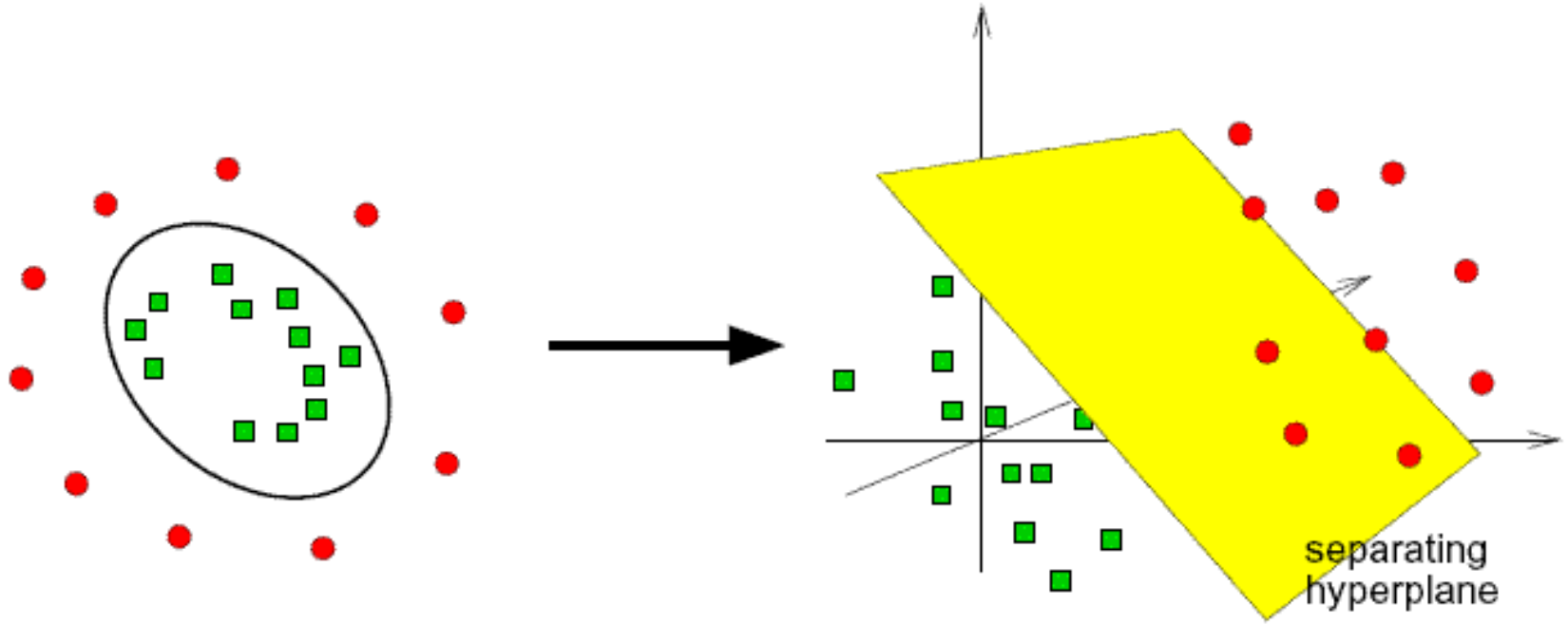
$$\begin{aligned} K(x_i, x_j) &= (x_i x_j + 1)^2 \\ &= ((y_{i1}, y_{i2}) \cdot (y_{j1}, y_{j2}) + 1)^2 \\ &= (y_{i1}y_{j1} + y_{i2}y_{j2} + 1)^2 \\ &= (y_{i1}^2, y_{i2}^2, \sqrt{2}y_{i1}, \sqrt{2}y_{i2}, \sqrt{2}y_{i1}y_{i2}, 1) \\ &\quad \cdot (y_{j1}^2, y_{j2}^2, \sqrt{2}y_{j1}, \sqrt{2}y_{j2}, \sqrt{2}y_{j1}y_{j2}, 1) \\ &= \Psi(x_i)\Psi(x_j) \end{aligned}$$

das Skalarprodukt im neuen Merkmalsraum $\Re^6$

Nichtlineare SVM

Beispiel: $\Psi : \Re^2 \longrightarrow \Re^3$

$$\Psi((y_1, y_2)) = (y_1^2, \sqrt{2}y_1y_2, y_2^2)$$



Die Kernelfunktion

$$K(x_i, x_j) = (x_i x_j)^2 = \Psi(x_i)\Psi(x_j)$$

berechnet das Skalarprodukt im neuen Merkmalsraum $\Re^3$. Wir können also das Skalarprodukt zwischen $\Psi(x_i)$ und $\Psi(x_j)$ ausrechnen, ohne die Funktion Ψ anzuwenden.

Nichtlineare SVM

Häufig verwendete Kernfunktionen:

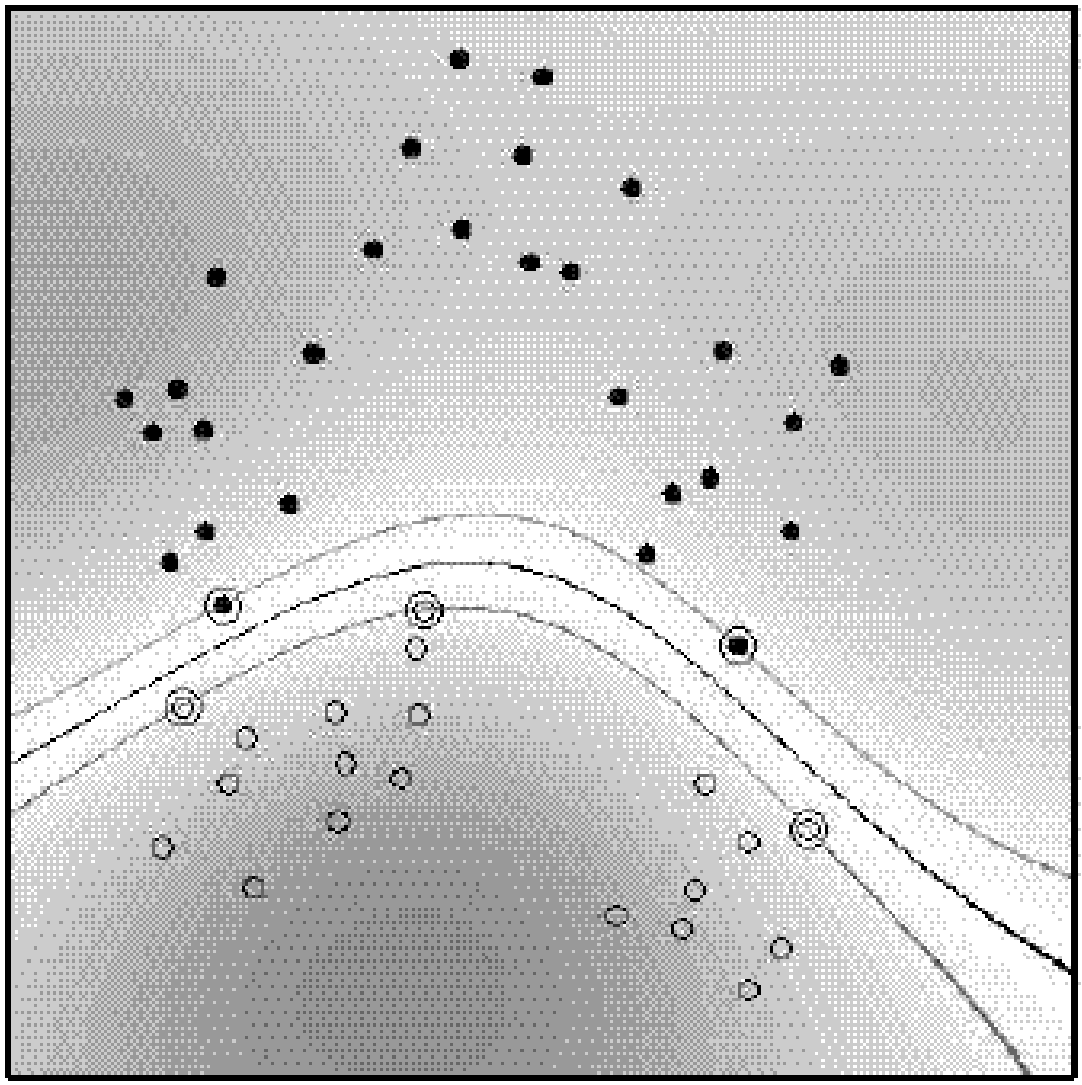
$$\begin{array}{lrcl} \text{Polynom-Kernel:} & K(x_i, x_j) & = & (x_i x_j)^d \\ \text{Gauß-Kernel:} & K(x_i, x_j) & = & e^{-\frac{||x_i - x_j||^2}{c}} \\ \text{Sigmoid-Kernel:} & K(x_i, x_j) & = & \tanh(\beta_1 x_i x_j + \beta_2) \end{array}$$

Lineare Kombinationen von gültigen Kernels $\rightarrow$ neue Kernelfunktionen

Wir müssen nicht wissen, wie der neue Merkmalsraum $\Re^m$ aussieht. Wir brauchen nur die Kernel-Funktion als ein Maß der Ähnlichkeit.

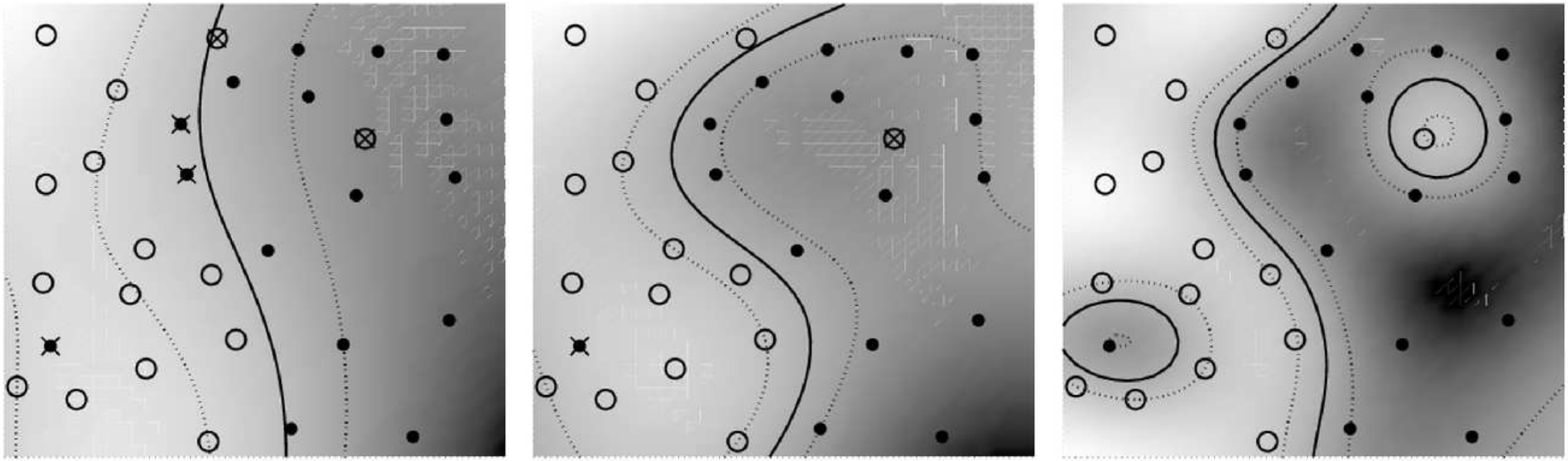
Nichtlineare SVM

Beispiel: Gauß-Kernel ($c = 1$). Die Support-Vektoren sind durch einen extra Kreis gekennzeichnet.



Nichtlineare SVM

Beispiel: Gauß-Kernel ($c = 1$) für Soft-Margin SVM.



Schlußbemerkungen

Stärken von SVM:

- SVM liefert nach derzeitigen Erkenntnissen sehr gute Klassifikationsergebnisse; bei einer Reihe von Aufgaben gilt sie als der Top-Performer
- Sparse-Darstellung der Lösung über Support-Vektoren
- Leicht anwendbar: wenig Parameter, kein a-priori-Wissen erforderlich
- Geometrisch anschauliche Funktionsweise
- Theoretische Aussagen über Ergebnisse: globales Optimum, Generalisierungsfähigkeit

Schwächen von SVM:

- Langsames speicherintensives Lernen
- "Tuning SVMs remains a black art: selecting a specific kernel and parameters is usually done in a try-and-see manner"

Schlußbemerkungen

- Liste von SVM-Implementierungen unter
  `http://www.kernel-machines.org/software`
- LIBSVM ist die gängigste: `http://www.csie.ntu.edu.tw/~cjlin/libsvm/`

Neuro-Fuzzy-Systeme

Neuro-Fuzzy-Systeme

- Nachteil neuronaler Netze:
  - Ergebnisse sind schlecht interpretierbar (black box)
  - Vorwissen kann schlecht dargestellt werden
- Ausweg:
  - hybrides System, bei dem NN mit einem regelbasierten System gekoppelt sind
  - ein mögliches Verfahren: Neuro-Fuzzy-Systeme

Kurzeinführung in die Fuzzy-Theorie

- **Klassische Logik:** nur Wahrheitswerte *wahr* und *falsch*
- **Klassische Mengenlehre:** entweder *ist Element* oder *nicht*
- Zweiwertigkeit dieser Theorien: oft unangemessen
- Beispiel: **Sorites-Paradoxon** (griech. *sorites*: Haufen)
  wahr: "Eine Milliarde Sandkörner sind ein Sandhaufen."
  wahr: "Wenn man von einem Sandhaufen ein Sandkorn entfernt, bleibt ein Sandhaufen übrig."
- *wahr:* 999 999 999 "Sandkörner sind ein Sandhaufen."
- mehrfache Wiederholung des gleichen Schlusses:
  falsch: "1 Sandkorn ist ein Sandhaufen."
- Frage: Bei welcher Anzahl Sandkörner ist Schluss nicht wahrheitsbewahrend?

Kurzeinführung in die Fuzzy-Theorie

- Offenbar: keine genau bestimmte Anzahl Sandkörner, bei der der Schluss auf nächstkleinere Anzahl falsch ist
- Problem: Begriffe der natürlichen Sprache (z.B. “Sandhaufen”, “kahlköpfig”, “warm”, “schnell”, “hoher Druck”, “leicht” etc.) sind **vage**
- beachte: vage Begriffe sind *unexakt*, aber nicht *unbrauchbar*
  - auch für vage Begriffe: Situationen/Objekte, auf die sie *sicher anwendbar* sind und solche, auf die sie *sicher nicht anwendbar* sind
  - dazwischen: **Penumbra** (lat. für *Halbschatten*) von Situationen, in denen es unklar ist, ob die Begriffe anwendbar sind, oder in denen sie nur mit Einschränkungen anwendbar sind (“kleiner Sandhaufen”).
  - Fuzzy-Theorie: mathematische Modellierung der Penumbra

Fuzzy-Logik

- Erweiterung der klassischen Logik um Zwischenwerte zwischen *wahr* und *falsch*
- Wahrheitswert: jeder Wert aus $[0, 1]$, wobei $0 \mathrel{\widehat{=}} \textit{falsch}$ und $1 \mathrel{\widehat{=}} \textit{wahr}$
- **Erweiterung der logischen Operatoren**

Klassische Logik		Fuzzy-Logik		
Operation	Notation	Operation	Notation	Beispiel
Negation	$\neg a$	Fuzzy-Negation	$\sim a$	$1 - a$
Konjunktion	$a \wedge b$	t-Norm	$\top(a, b)$	$\min(a, b)$
Disjunktion	$a \vee b$	t-Konorm	$\bot(a, b)$	$\max(a, b)$

- **Prinzipien** der Erweiterung:
  - für Extremwerte 0 und 1 sollen sich Operationen genauso verhalten wie ihre klassischen Vorbilder (Rand-/Eckbedingungen)
  - für Zwischenwerte soll ihr Verhalten monoton sein
  - Gesetze der klassischen Logik sollen (fast alle) erhalten werden

Fuzzy-Mengenlehre

- klassische Mengenlehre basiert auf Begriff "*ist Element von*" ($\in$)
- alternativ: Zugehörigkeit zu Menge mit *Indikatorfunktion* beschreibbar: sei X eine Menge, dann heißt

$$I_M : X \to \{0, 1\}, \qquad I_M(x) = \begin{cases} 1, & \text{falls } x \in X, \\ 0, & \text{sonst,} \end{cases}$$

  Indikatorfunktion der Menge M bzgl. Grundmenge X
- in Fuzzy-Mengenlehre: ersetze Indikatorfunktion durch *Zugehörigkeitsfunktion*: sei X (klassische/scharfe) Menge, dann heißt

$$\mu_M : X \to [0, 1], \qquad \mu_M(x) \widehat{=} \text{ Zugehörigkeitsgrad von } x \text{ zu } M,$$

  Zugehörigkeitsfunktion (membership function) der **Fuzzy-Menge** M bzgl. der *Grundmenge* X
- Fuzzy-Menge: definiert über ihre Zugehörigkeitsfunktion

Formale Definition einer Fuzzy-Menge

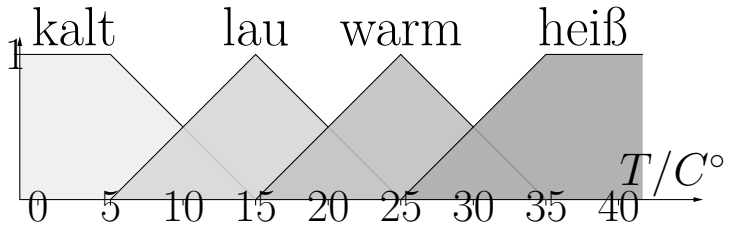
1. Eine Fuzzy-Menge $\mu \in X \neq \emptyset$ ist eine Funktion, die aus der Referenzmenge X in das Einheitsintervall abbildet, d.h. $\mu : X \to [0, 1]$.

2. $\mathcal{F}(X)$ stellt die Menge aller Fuzzy-Mengen von X dar,
d.h. $\mathcal{F}(X) \stackrel{\text{def}}{=} \{\mu \mid \mu : X \to [0, 1]\}$.

Fuzzy-Partitionen und Linguistische Variablen

- um Wertebereich durch sprachliche (linguistische) Ausdrücke beschreiben zu können, wird er mithilfe von Fuzzy-Mengen fuzzy-partitioniert
- jeder Fuzzy-Menge der Partitionierung ist ein linguistischer Term zugeordnet
- übliche Bedingung: an jedem Punkt müssen sich Zugehörigkeitsgrade aller Fuzzy-Mengen zu 1 addieren

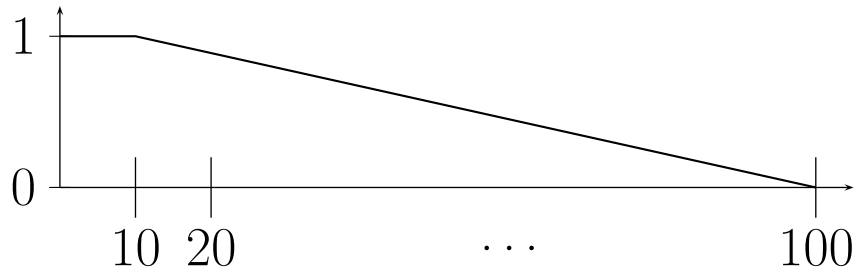
Beispiel: Fuzzy-Partitionierung für Temperaturen
Linguistische Variable mit den Werten *kalt*, *lau*, *warm* und *heiß*.



Subjektive Definition einer Fuzzymenge – Beispiel

- X Menge der Magdeburger Einwohner im Alter zwischen 10 und 100 Jahren
- $Y = \{1, \ldots, 100\}$
- $j(y)$ Anzahl der Einwohner die y alt sind, die sich als "jung" bezeichnen
- $n(y)$ Gesamtzahl der Einwohner im Alter y

$$\mu : Y \to [0, 1], \quad \mu(y) = \begin{cases} \frac{j(y)}{n(y)} & \text{if } y > 10 \\ 1 & \text{if } y \leq 10 \end{cases}$$

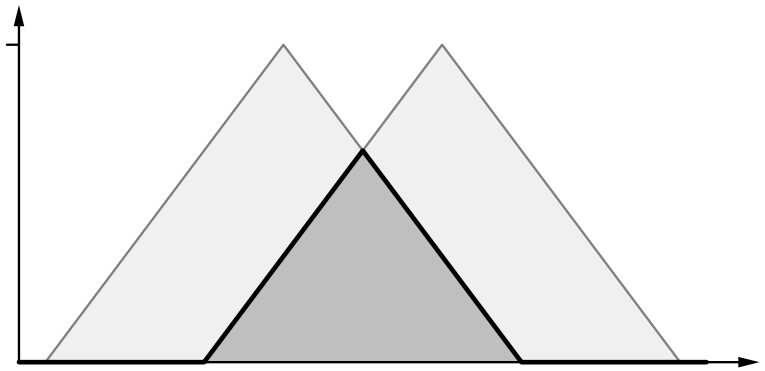


Operationen

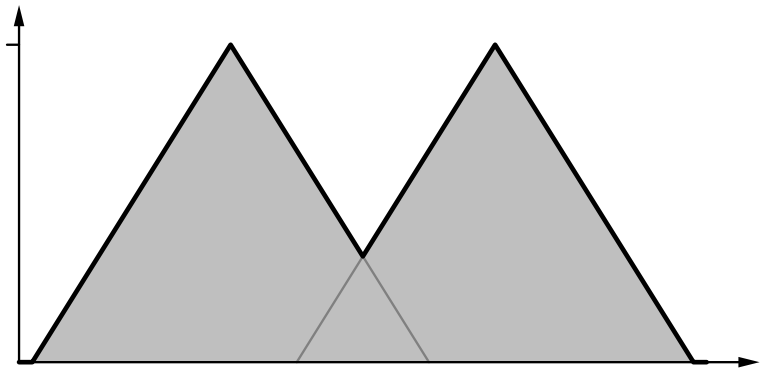
- wie beim Übergang von klassischer Logik zur Fuzzy-Logik: hier auch Erweiterung der Operationen nötig
- **dieser Erweiterung**:
  greife auf logische Definition der Operationen zurück
- elementweise Anwendung der logischen Operatoren
- (Fuzzy-)Mengen A und B über Grundmenge X

Komplement	klassisch	$\overline{A} = \{x \in X \mid x \notin A\}$
	fuzzy	$\forall x \in X: \quad \mu_{\overline{A}}(x) = \sim\mu_A(x)$
Schnitt	klassisch	$A \cap B = \{x \in X \mid x \in A \wedge x \in B\}$
	fuzzy	$\forall x \in X: \quad \mu_{A\cap B}(x) = \top(\mu_A(x), \mu_B(x))$
Vereinigung	klassisch	$A \cup B = \{x \in X \mid x \in A \vee x \in B\}$
	fuzzy	$\forall x \in X: \quad \mu_{A\cup B}(x) = \bot(\mu_A(x), \mu_B(x))$

Beispiele für Schnitt und Vereinigung

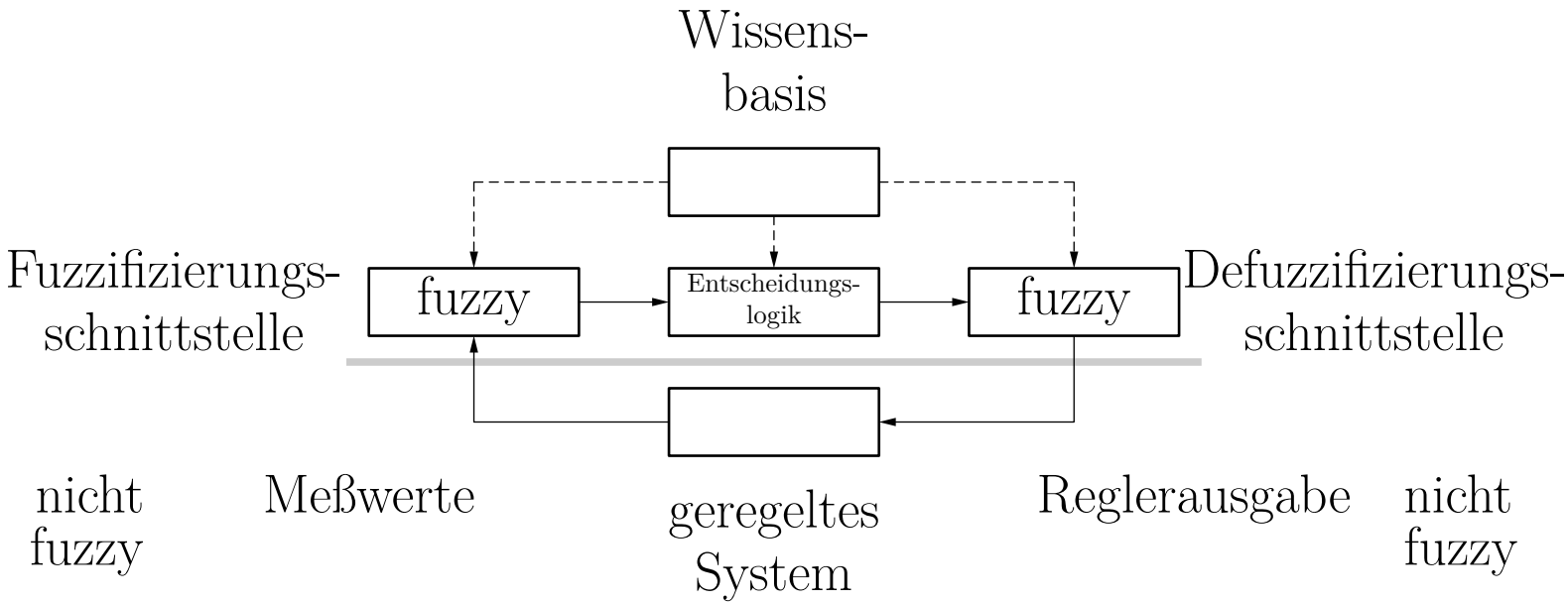


Fuzzy-Schnitt (min)



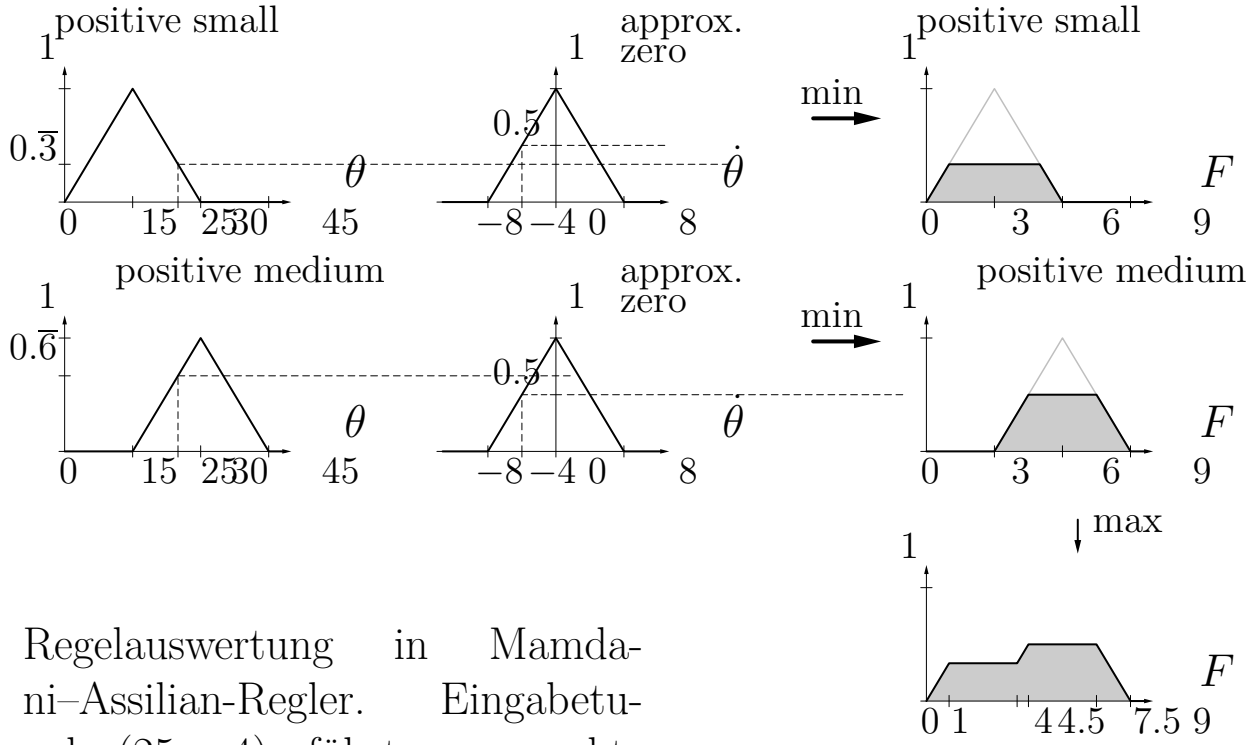
Fuzzy-Vereinigung (max)

Architektur eines Fuzzy-Reglers



- Wissensbasis enthält Fuzzy-Regeln für Steuerung und Fuzzy-Partitionen der Wertebereiche der Variablen
- Fuzzy-Regel: **if** X_1 **is** $A_{i_1}^{(1)}$ **and** $\ldots$ **and** X_n **is** $A_{i_n}^{(n)}$ **then** Y **is** B.
  $X_1, \ldots, X_n$ sind Messgrößen und Y ist Stellgröße
  $A_{i_k}^{(k)}, B$: linguistische Terme (mit Fuzzy-Mengen assoziiert)

Fuzzy-Regelung nach Mamdani-Assilian



Regelauswertung in Mamdani–Assilian-Regler. Eingabetupel $(25, -4)$ führt zur rechts gezeigten unscharfen Ausgabe. Aus dieser Fuzzy-Menge wird entsprechender Ausgabewert durch Defuzzifizierung bestimmt, z.B. durch die Schwerpunktmethode (COG).

Neuro-Fuzzy-Systeme

Der Aufbau eines Fuzzy-Systems benötigt:

- Vorwissen (Fuzzy-Regeln, Fuzzy-Mengen)
- Manuelle Anpassungen, die zeitaufwendig und fehlerträchtig sind

$\Rightarrow$ Unterstütze diesen Prozess durch Lernverfahren:

- Erlernen von Fuzzy-Regeln (Struktur-Lernen)
- Erlernen von Fuzzy-Mengen (Parameter-Lernen)

Ansatz mit künstlichen neuronalen Netzen kann genutzt werden

Fallstudie: Aktienkursvorhersage

Prognose der täglichen relativen Änderungen des DAX, aufbauend auf Zeitreihen von Börsen-Indizes im Zeitraum von 1986 bis 1997

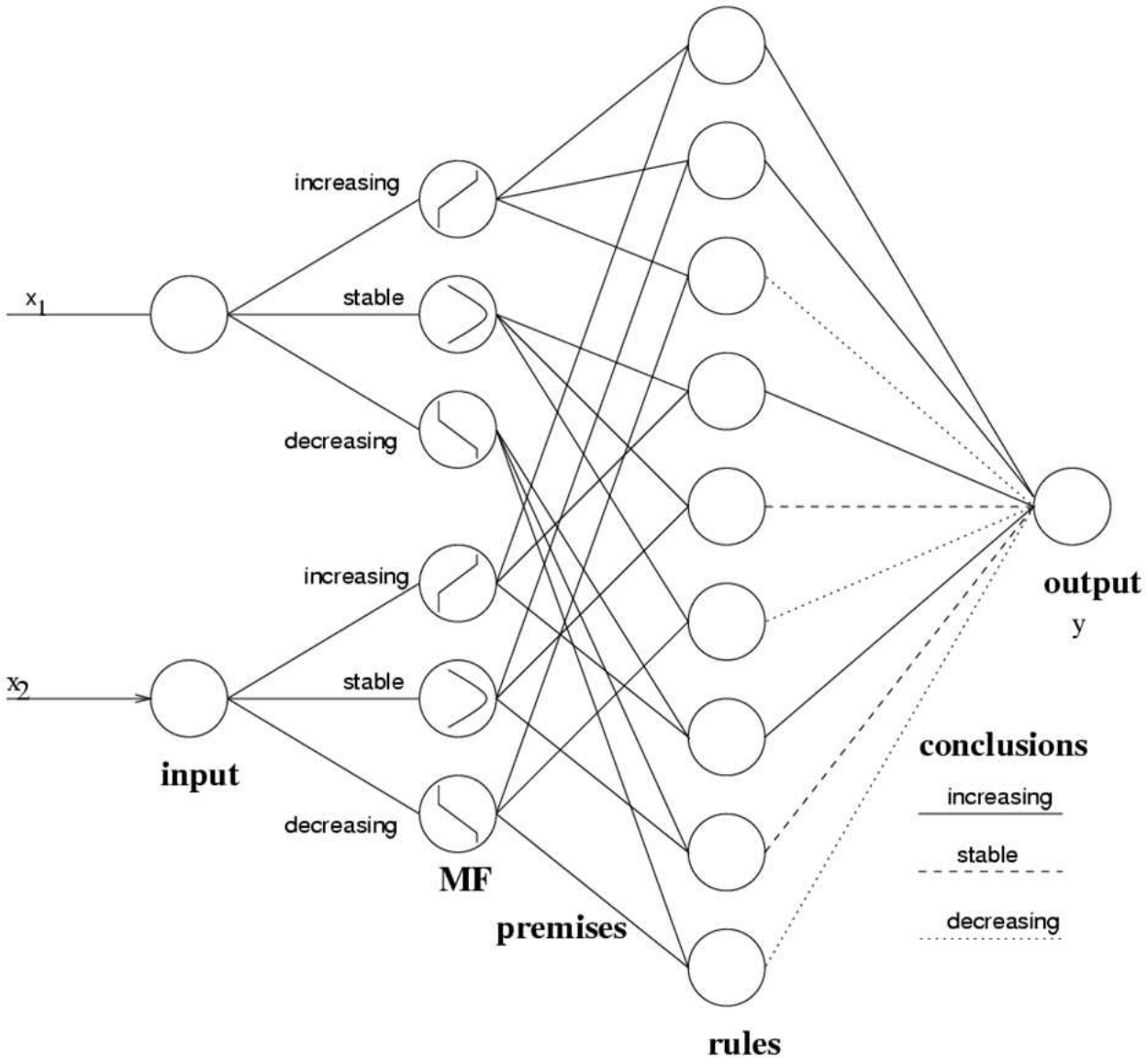
DAX	Composite-DAX
German 3 months interest rate	Return Germany
German Morgan-Stanley index	Dow Jones industrial index
DM / US-$	US treasure bonds
gold price	Japanese Nikkei-Index
European Morgan-Stanley-Index	Price earning ratio

Fuzzy-Regeln im Finanzbereich

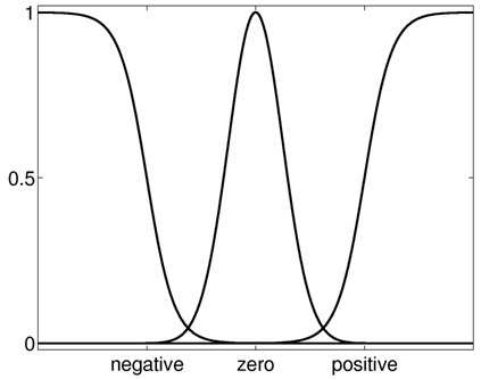
- trend rule
  IF DAX = decreasing AND US-$ = decreasing
  THEN DAX prediction = decreasing
  WITH high certainty
- turning point rule
  IF DAX = decreasing AND US-$ = increasing
  THEN DAX prediction = increasing
  WITH low certainty
- delay rule
  IF DAX = stable AND US-$ = decreasing
  THEN DAX prognosis = decreasing
  WITH very high certainty
- in general
  IF x_1 is μ_1 AND x_2 is μ_2 AND $\ldots$ AND x_n is μ_n
  THEN $y = \eta$
  WITH weight k

Neuro-Fuzzy-Architektur

Ausschnitt eines NF-Systems.



Von Regeln zu Neuronalen Netzen



1. Bewertung von Zugehörigkeitsgraden
2. Bewertung von Regeln (Regelaktivität)

$$\mu_l = \mathbb{R}^n \to [0,1]^r, \quad \underline{x} \Rightarrow \prod_{j=1}^{D_l} \mu_{c,s}^{(j)}(x_i)$$

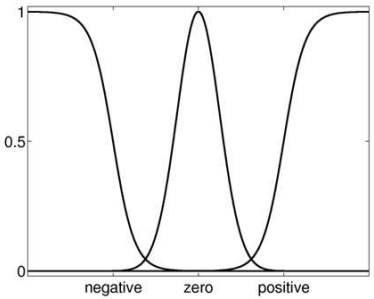
3. Akkumulation von Regeleingaben, Normalisierung

$$\mathrm{NF} : \mathbb{R}^n \to \mathbb{R}, \quad \underline{x} \Rightarrow \sum_{l=1}^{r} w_l \frac{k_l \mu_l(\underline{x})}{\sum_{j=1}^{r} k_j \mu_j(\underline{x})}$$

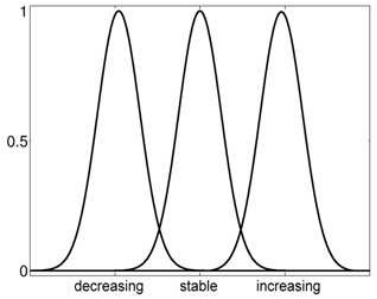
Dimensionsreduktion des Gewichtsraums

1. Zugehörigkeitsfunktionen verschiedener Eingaben teilen ihre Parameter untereinander, z.B.
$$\mu^{\text{stabil}}_{\text{DAX}} = \mu^{\text{stabil}}_{\text{C-DAX}}$$

2. Zugehörigkeitsfunktionen derselben Eingabevariable dürfen nicht einander passieren, sondern müssen ihre Originalreihenfolge beibehalten, d.h.



$$\mu_{\text{decreasing}} < \mu_{\text{stable}} < \mu_{\text{increasing}}$$



Vorteile:

- Die optimierte Regelbasis ist immer noch interpretierbar.
- Die Anzahl freier Parameter wurde reduziert.

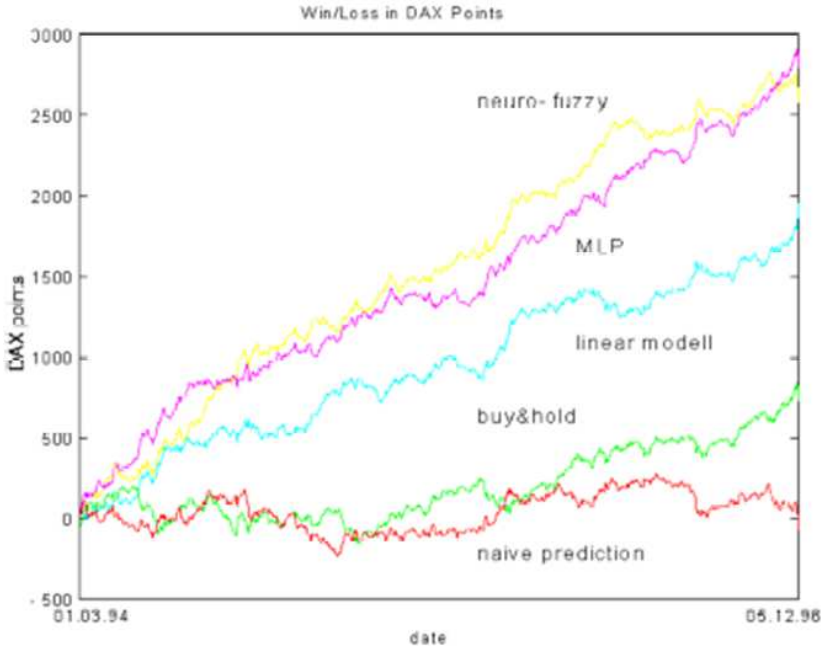
Lernen des NFS

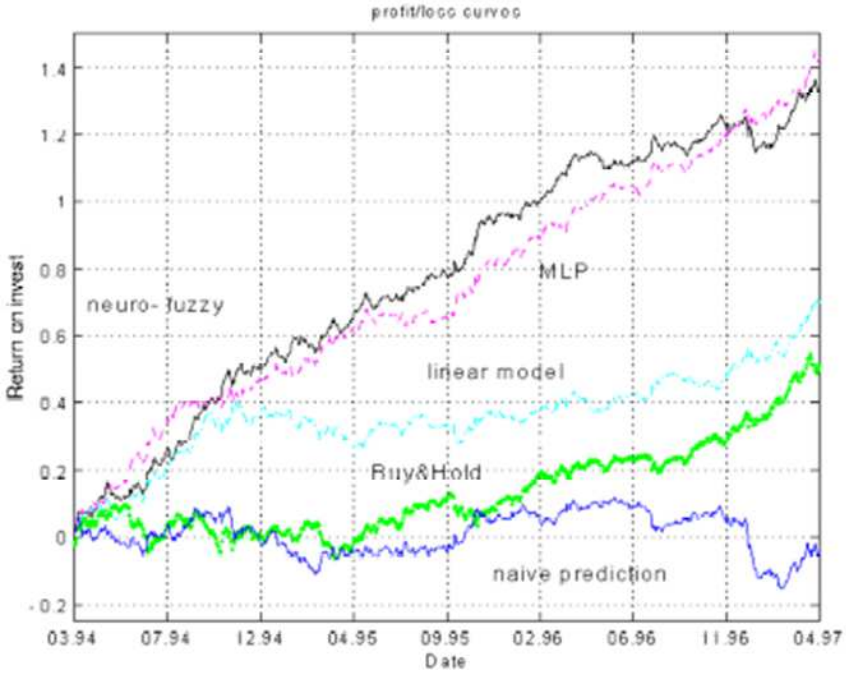
- Die Parameter der Fuzzy-Mengen,
- die Gewichte
- und die Regelwichtigkeiten

werden durch ein Backpropagation-Verfahren ermittelt. Es werden Pruning-Verfahren genutzt.

Gewinnkurven

- verschiedene Modelle
- Validierungsdaten: März 1994 bis April 1997





Fallstudie: medizinische Diagnose nach NEFCLASS-Einführung

- Ergebnisse von Patienten, die auf Brustkrebs getestet wurden (Wisconsin Breast Cancer Data)
- Entscheidungsunterstützung: liegt ein gutartiger oder bösartiger Fall vor?
- Ein Chirurg muß die Klassifikation auf ihre Plausibilität hin überprüfen können.
- Es wird nach einem einfachen und interpretierbaren Klassifikator gesucht

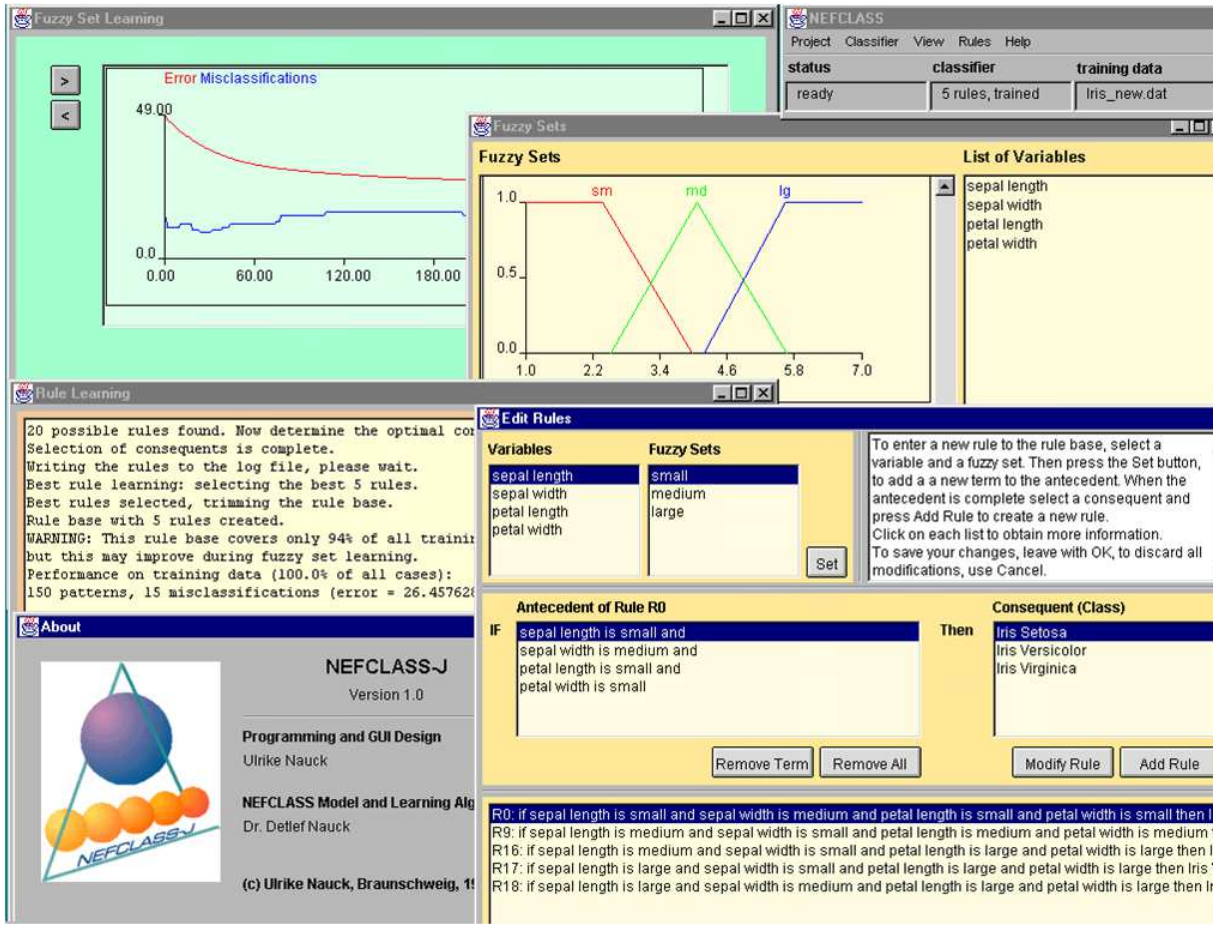
$$\Rightarrow \text{Wissensentdeckung}$$

Fallstudie: WBC-Daten

- 699 Fälle (16 Fälle mit fehlenden Werten)
- 2 Klassen: gutartig(458), bösartig (241)
- 9 Attribute mit Werten in $\{1, \ldots, 10\}$ (ordinale Skala, aber normalerweise numerisch interpretiert)
- Experiment: x_3 und x_6 werden als nominale Attribute interpretiert
- x_3 und x_6 werden oft als "wichtige" Attribute angesehen

Anwendung von NEFCLASS-J

- Werkzeug zur Entwicklung von NF-Klassifikatoren
- Java-Implementierung
- frei verfügbar zu Forschungszwecken
- Projekt in unserer Gruppe gestartet



`http://fuzzy.cs.ovgu.de/nefclass/nefclass-j/`

NEFCLASS: Neuro-Fuzzy-Klassifikator

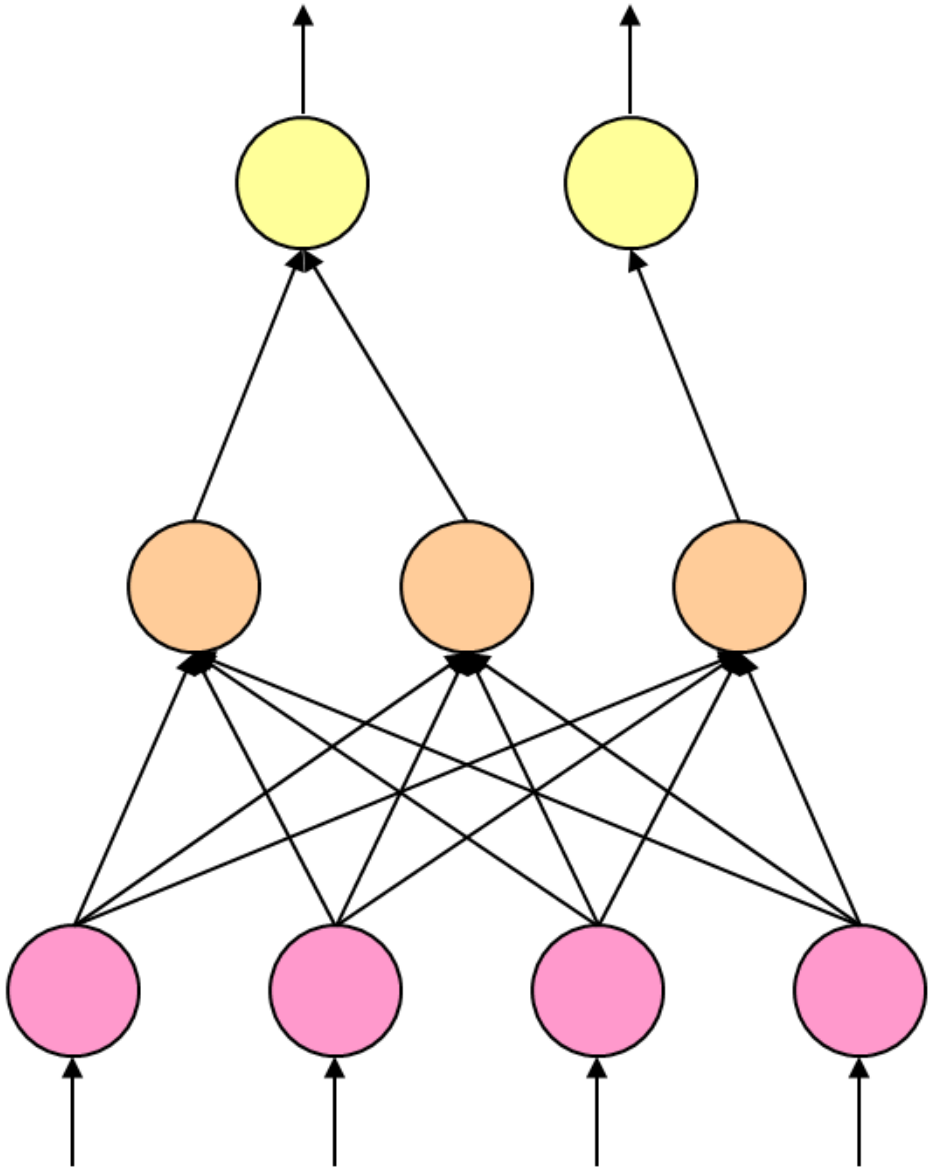
Ausgabevariablen

ungewichtete Verbindungen

Fuzzy-Regeln

Fuzzy-Mengen (Antezedens)

Eingabeattribute (Variablen)



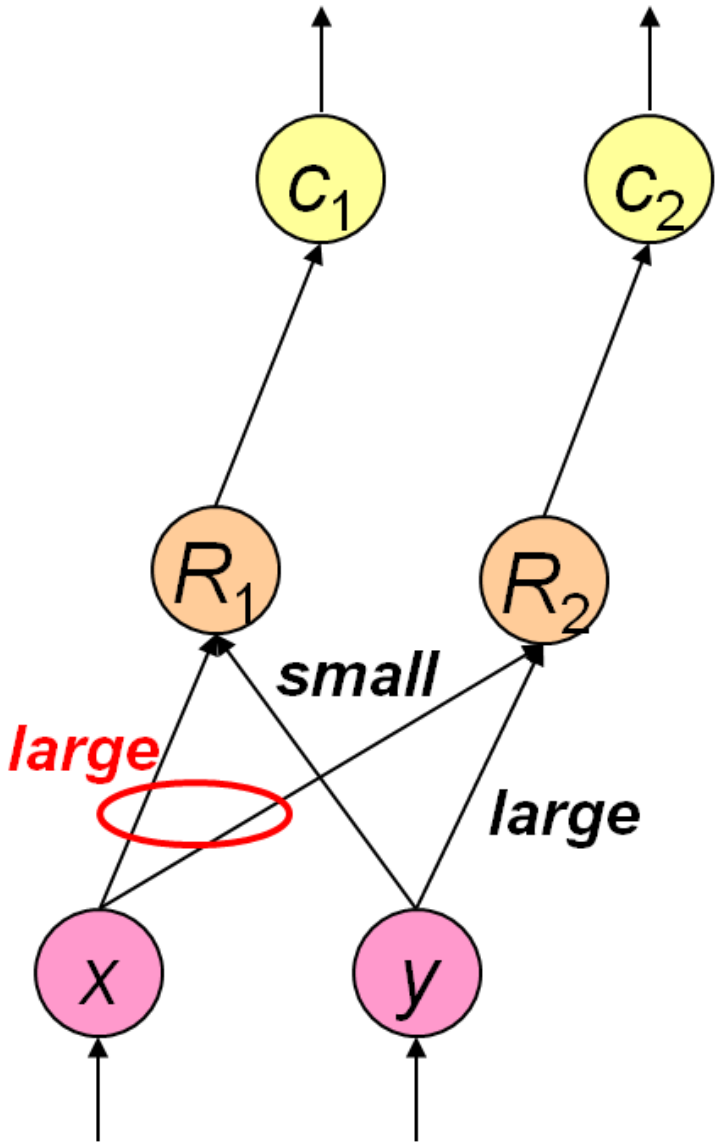
NEFCLASS: Merkmale

- automatische Erstellung der Fuzzy-Regelbasis aus Daten
- Trainieren verschiedener Formen von Fuzzy-Mengen
- Verarbeiten von numerischen und symbolischen Attributen
- Behandlung von fehlenden Werten (kein Entfernen)
- automatische Beschneidungsstrategien
- Verschmelzen von Expertenwissen und Daten

Darstellung von Fuzzy-Regeln

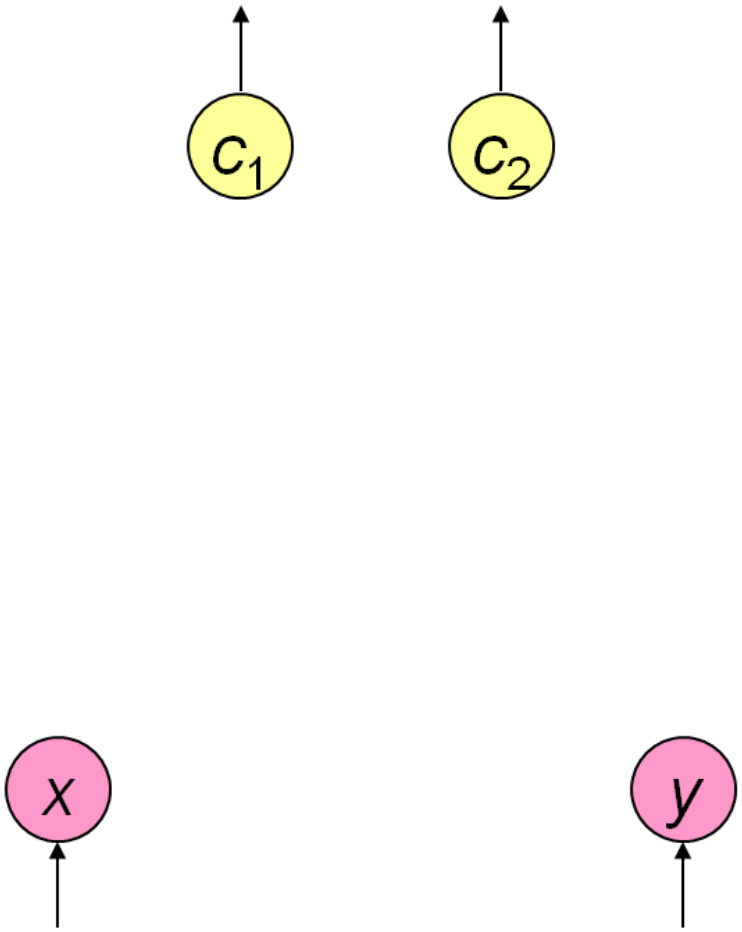
Beispiel: 2 Regeln
R_1 : if x is *large* and y is *small*, then class is c_1
R_2 : if x is *large* and y is *large*, then class is c_2

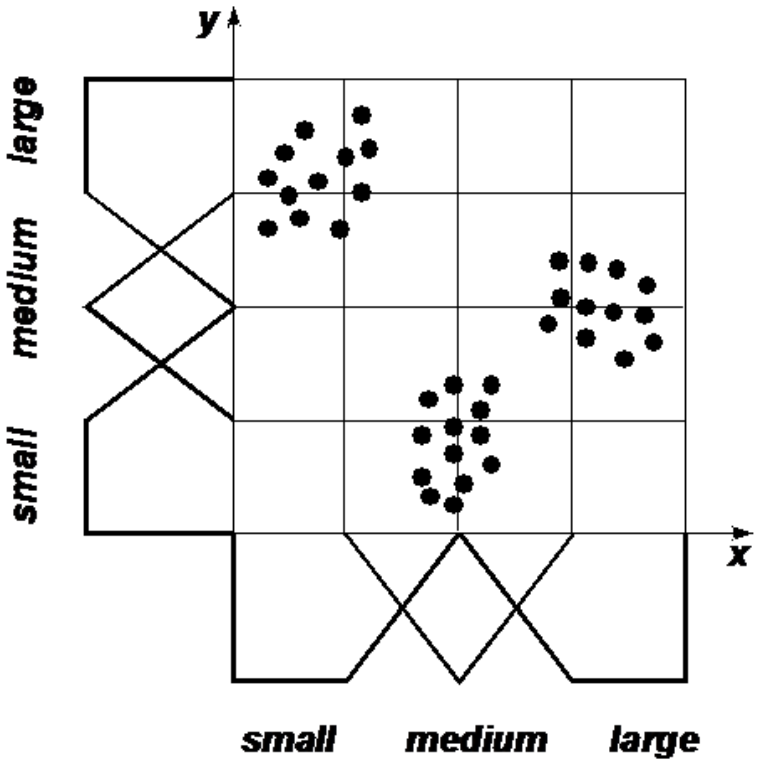
- Verbindungen $x \rightarrow R_1$ und $x \rightarrow R_2$ sind verbu
- Fuzzymenge *large* teilt Gewicht auf
- d.h. *large* hat immer
  dieselbe Bedeutung in beiden Regeln



1. Trainieren: Initialisierung

Spezifiziere anfängliche Fuzzy-Partitionen für alle Eingabevariablen





1. Trainieren: Regelbasis

for each pattern p {
  find antecedent A *s.t.* $A(p)$ is maximal
  if $A \notin L$ {
    add A to L
  }
}
for each antecedent $A \in L$ {
  find best consequent C for A
  create rule base candidate $R = (A, C)$
  determine performance of R
  add R to B
}
return one rule base from B

Fuzzy-Regel-Basen können auch aus Vorwissen, Fuzzy-Cluster-Analyse, Fuzzy-Entscheidungsbäumen, Evolutionären Algorithmen etc. gewonnen werden

Auswahl einer Regelbasis

Effizienz einer Regel:

$$P_r = \frac{1}{N} \sum_{i=1} N(-1)^c R_r(\boldsymbol{x}_i)$$

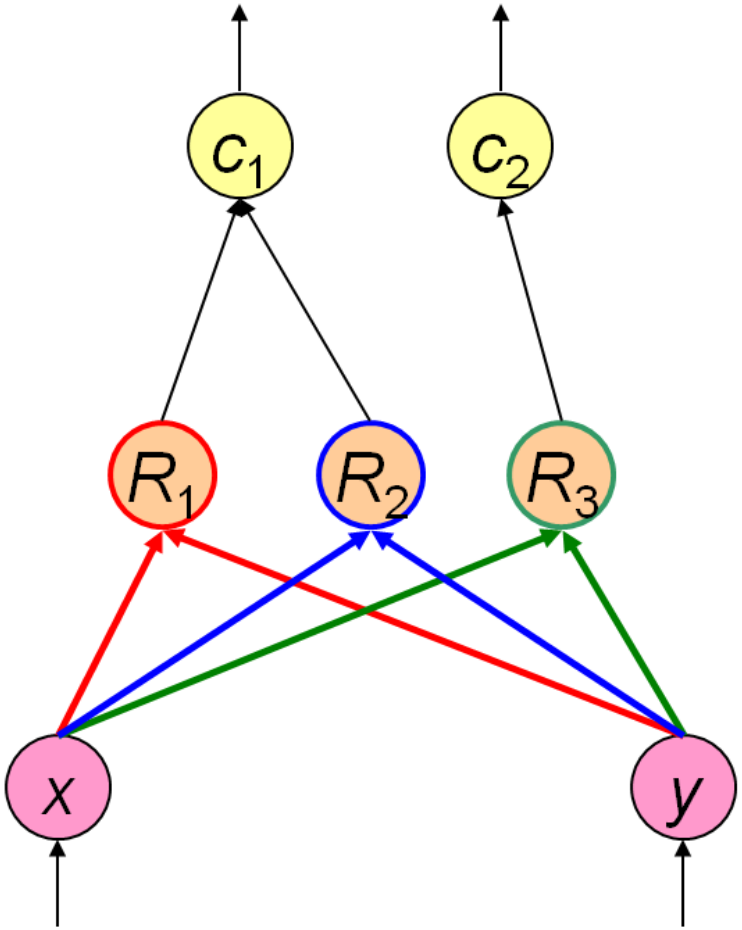
mit

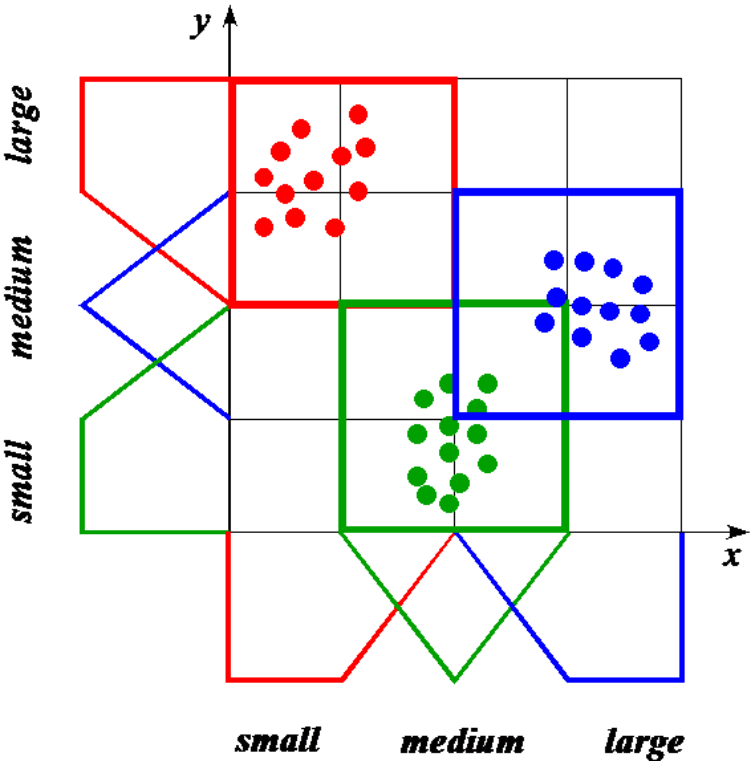
$$c = \begin{cases} 0 & \text{falls } \mathrm{class}(x_i) = \mathrm{con}(R_r), \\ 1 & \text{sonst} \end{cases}$$

- sortiere Regeln nach Effizienz
- wähle entweder die besten r Regeln oder die besten r/m Regeln pro Klasse aus
- r ist entweder gegeben oder wird automatisch so bestimmt, dass alle Muster abgedeckt werden

Induktion der Regelbasis

NEFCLASS benutzt eine angepasste Wang-Mendel-Prozedur





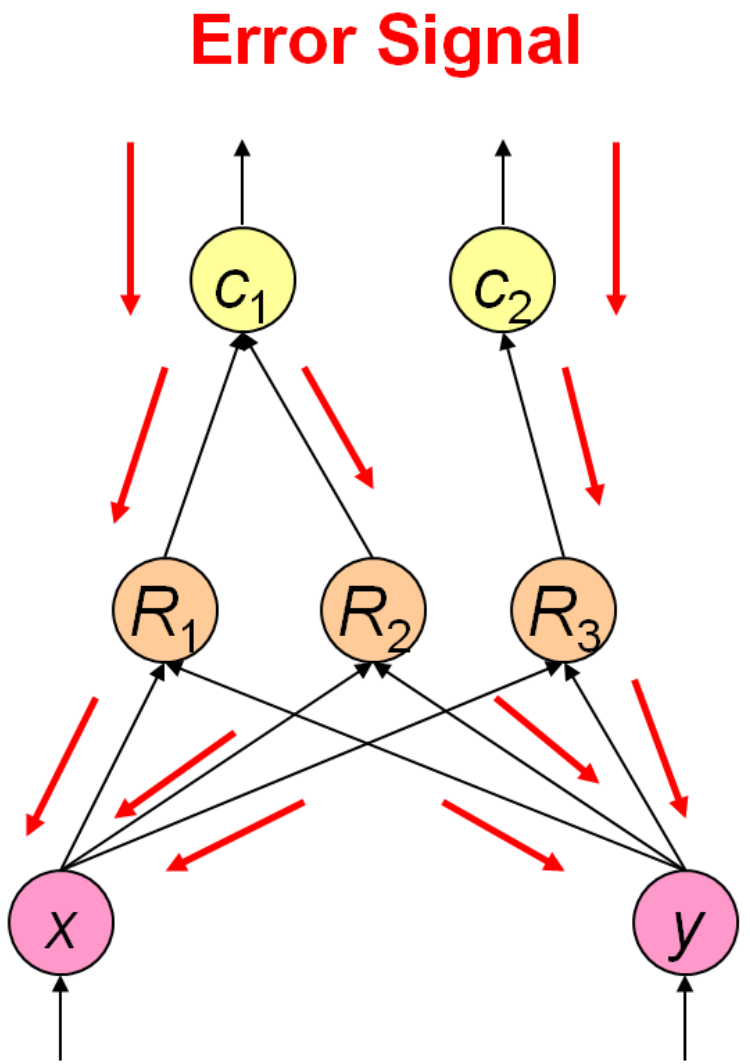
Berechnung des Fehlersignals

Fuzzy-Fehler (j-te Ausgabe):
$E_j = \operatorname{sgn}(d)(1 - \gamma(d))$

mit $d = t_j - o_j$ und
$\gamma : \mathbb{R} \to [0, 1], \gamma(d) = \exp - \left(\frac{a \cdot d}{d_{\max}}\right)^2$

(t: korrekte Ausgabe, o: aktuelle Ausgabe)

Regel-Fehler:
$E_r = (\tau_r(1 - \tau_r) + \varepsilon)E_{\mathrm{con}(R_r)}$
mit $0 < \varepsilon \ll 1$



3. Trainingsschritt: Fuzzy-Mengen

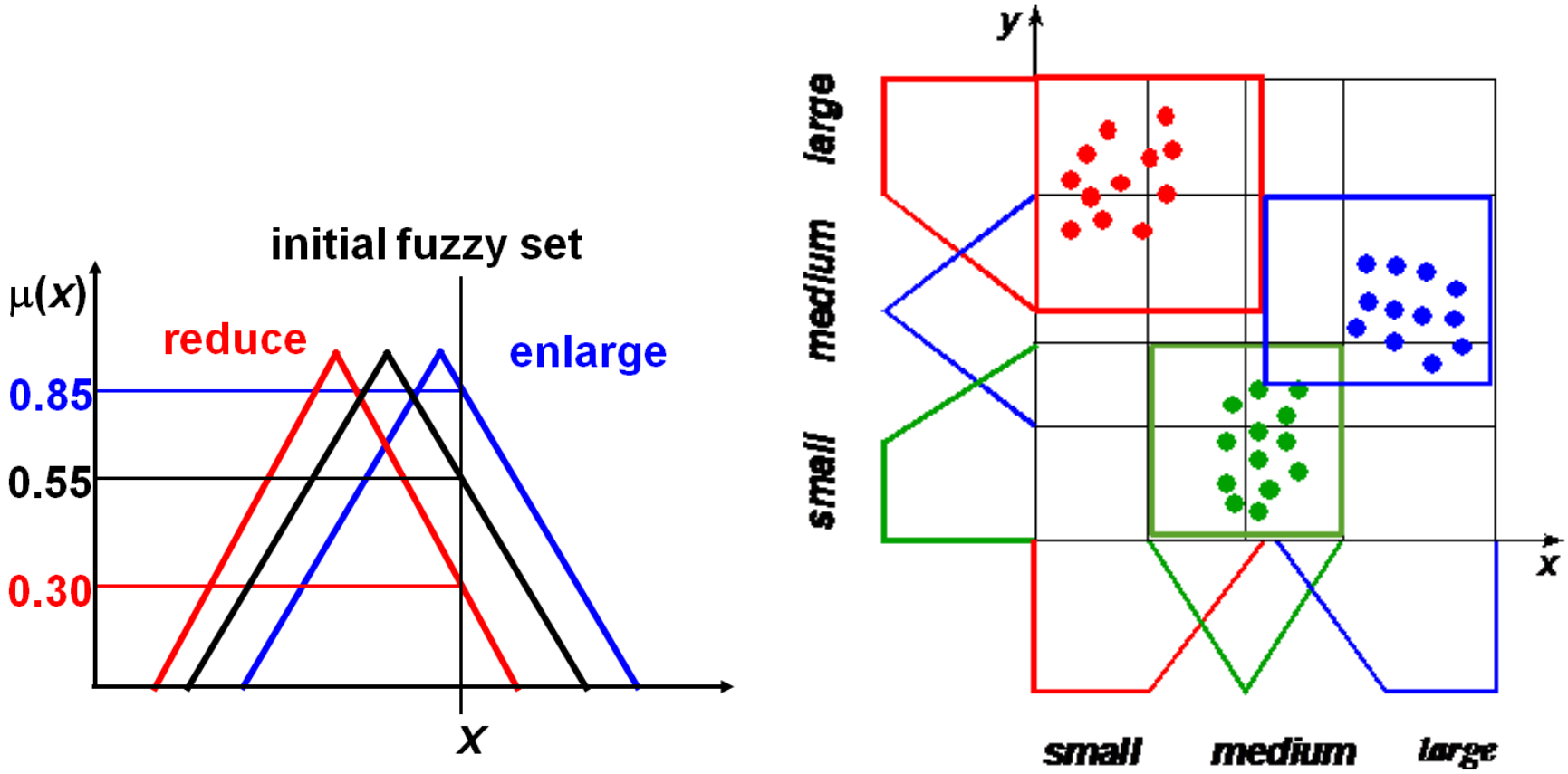
z.B. dreieckige Zugehörigkeitsfunktion

$$\mu_{a,b,c} : \mathbb{R} \to [0,1], \quad \mu_{a,b,c}(x) = \begin{cases} \frac{x-a}{b-a} & \text{if } x \in [a,b), \\ \frac{c-x}{c-b} & \text{falls } x \in [b,c], \\ 0 & \text{sonst} \end{cases}$$

Parameteranpassungen für eine Antezedens-Fuzzymenge:

$$f = \begin{cases} \sigma\mu(x) & \text{falls } E < 0, \\ \sigma(1-\mu(x)) & \text{sonst} \end{cases}$$
$$\Delta b = f \cdot E \cdot (c-a) \operatorname{sgn}(x-b)$$
$$\Delta a = -f \cdot E \cdot (b-a) + \Delta b$$
$$\Delta c = f \cdot E \cdot (c-b) + \Delta b$$

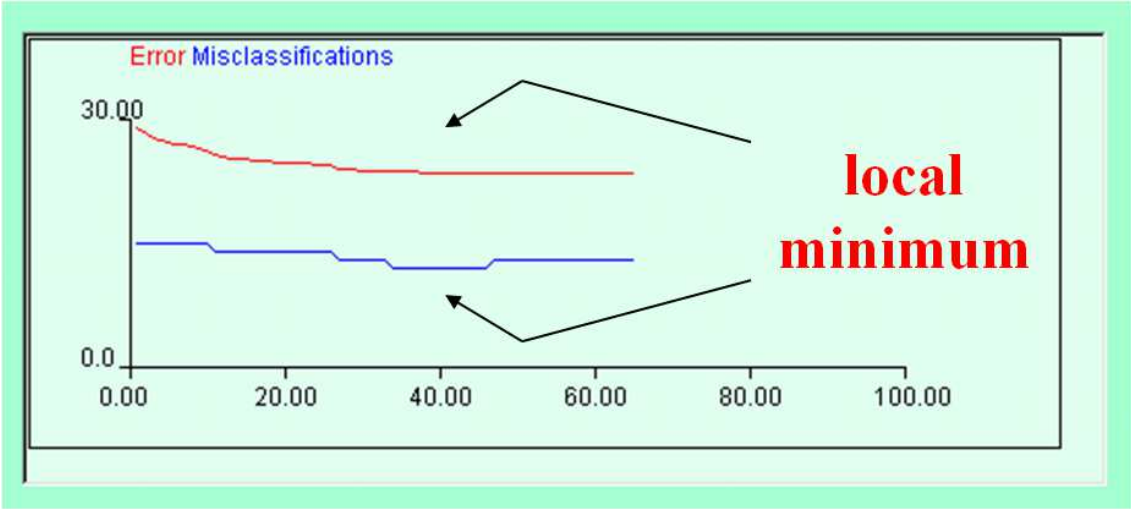
Trainieren von Fuzzy-Mengen



Heuristik: die Fuzzy-Menge wird von x weg (auf x zu) bewegt und ihr *support* wird reduziert (vergrößert) um den Zugehörigkeitsgrad von x zu reduzieren (erhöhen)

Trainieren von Fuzzy-Mengen

do {
  for each pattern {
    accumulate parameter updates
    accumulate error
  }
  modify parameters
} **while** change in error

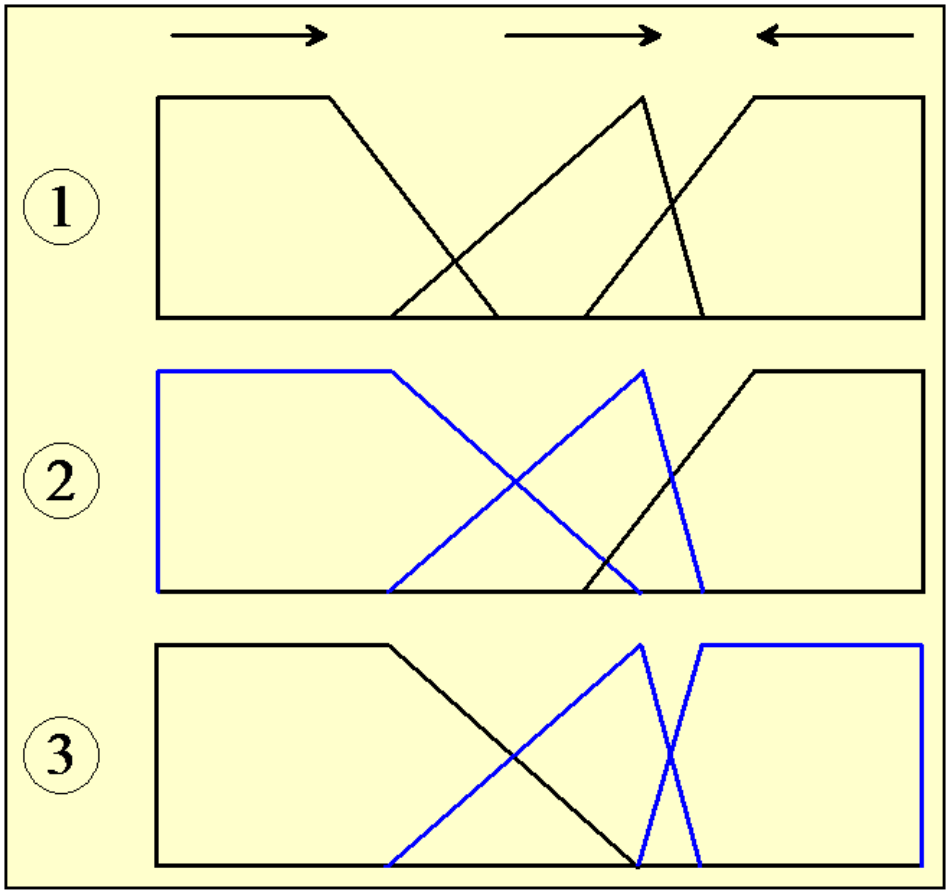


Varianten:

- Adaptive Lernrate
- Online-/Batch-Lernen
- Optimistisches Lernen (n Schritte in die Zukunft blickend)

Einschränkungen beim Trainieren von Fuzzy-Mengen

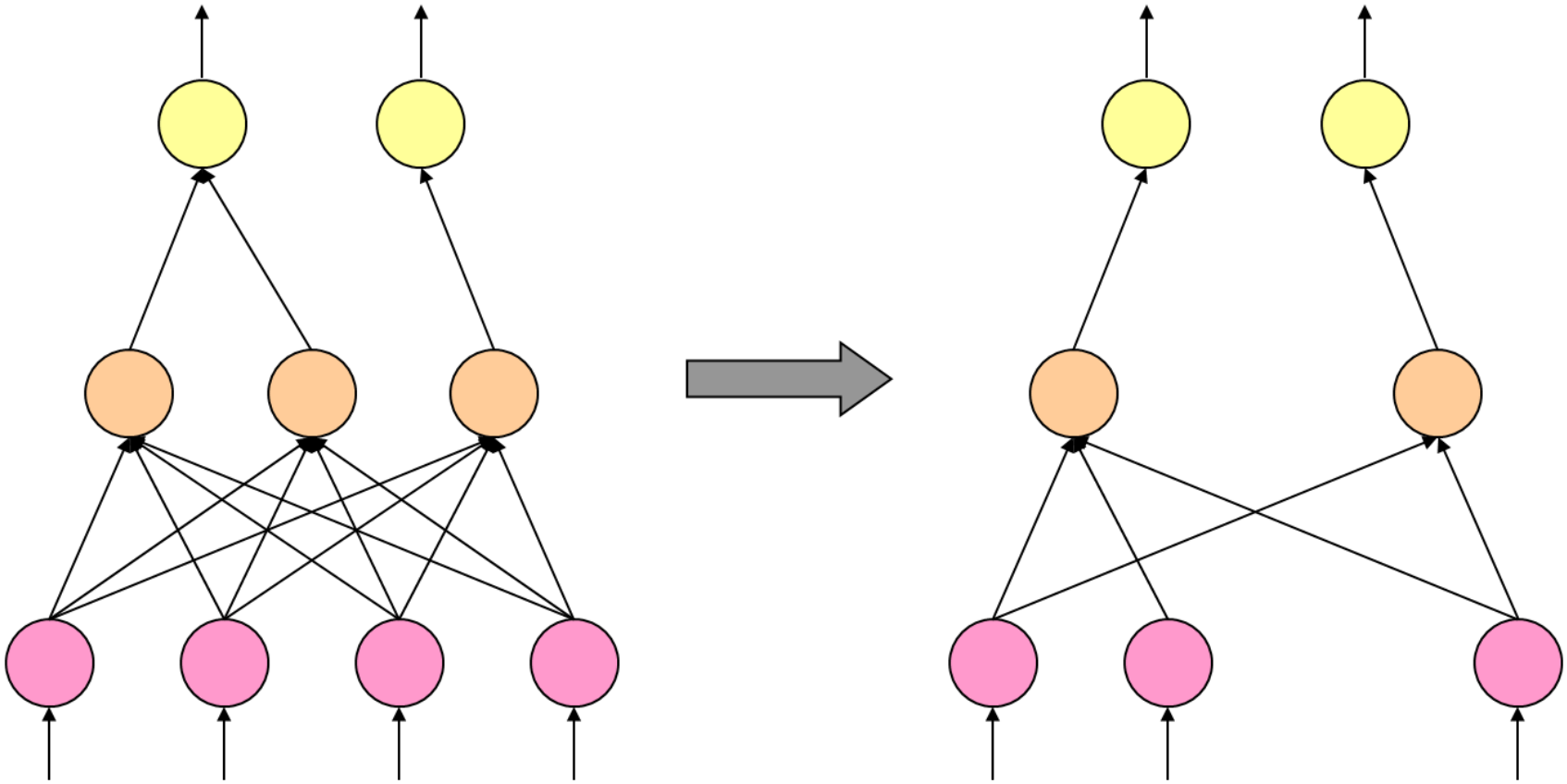
- gültige Parameterwerte
- nicht-leere Schnitte benachbarter Fuzzy-Mengen
- Beibehalten relativer Positionen
- Erhalt der Symmetrie
- Komplette Abdeckung (Zugehörigkeitsgrade für jedes Element summieren sich zu 1)



Correcting a partition after modifying the parameters

4. Trainingsschritt: Stutzen

Ziel: Entferne Variablen, Regeln und Fuzzy-Mengen, um die Interpretierbarkeit und Generalisierungsfähigkeit zu verbessern



Stutzen: Methoden

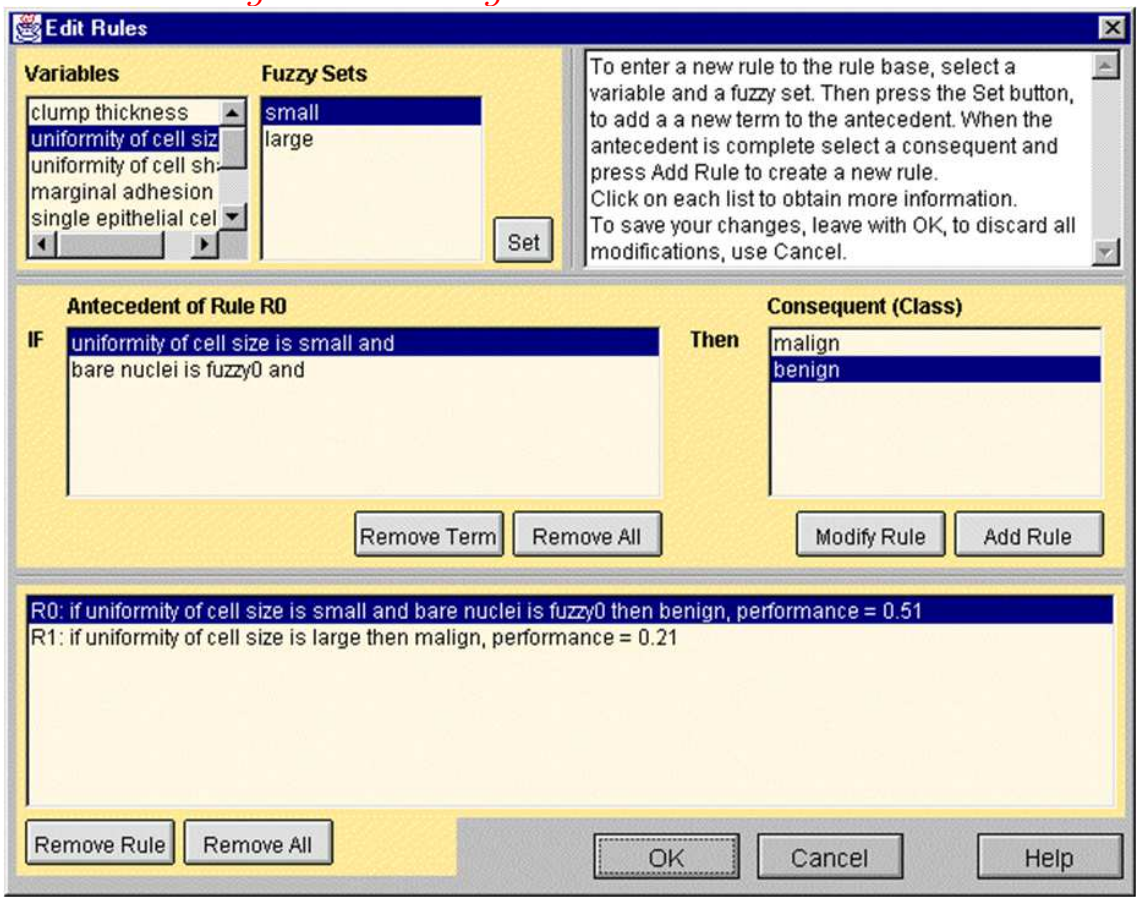
do {
  select pruning method
  do {
    execute pruning step
    train fuzzy sets
    if no improvement {
      undo step
    }
  } **while** there is improvement
} **while** there is further method

1. Entferne Variablen (Korrelationen, Information Gain, etc.)
2. Entferne Regeln (Effizienz einer Regel)
3. Entferne Terme (Erfüllungsgrad einer Regel)
4. Entferne Fuzzy-Mengen

WBC- Ergebnisse: Fuzzy-Regeln

R_1: if uniformity of cell size is *small* and bare nuclei is fuzzy0 then *benign*

R_2: if uniformity of cell size is *large* then *malignant*



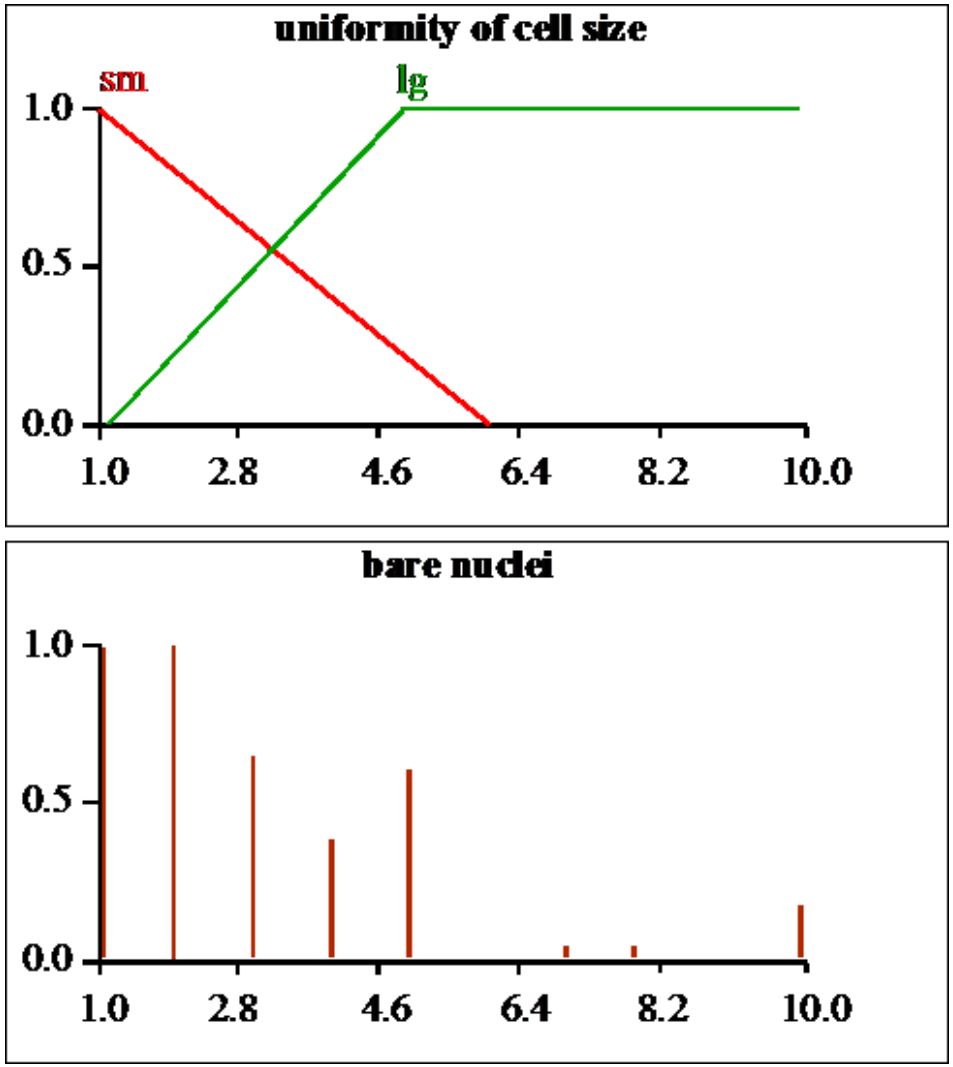
WBC-Ergebnisse: Klassifikation

	vorhergesagte Klasse					
	bösartig		gutartig		Σ	
bösartig	228	(32.62%)	13	(1.86%)	241	(34.99%)
gutartig	15	(2.15%)	443	(63.38%)	458	(65.01%)
Σ	243	(34.76)	456	(65.24)	699	(100.00%)

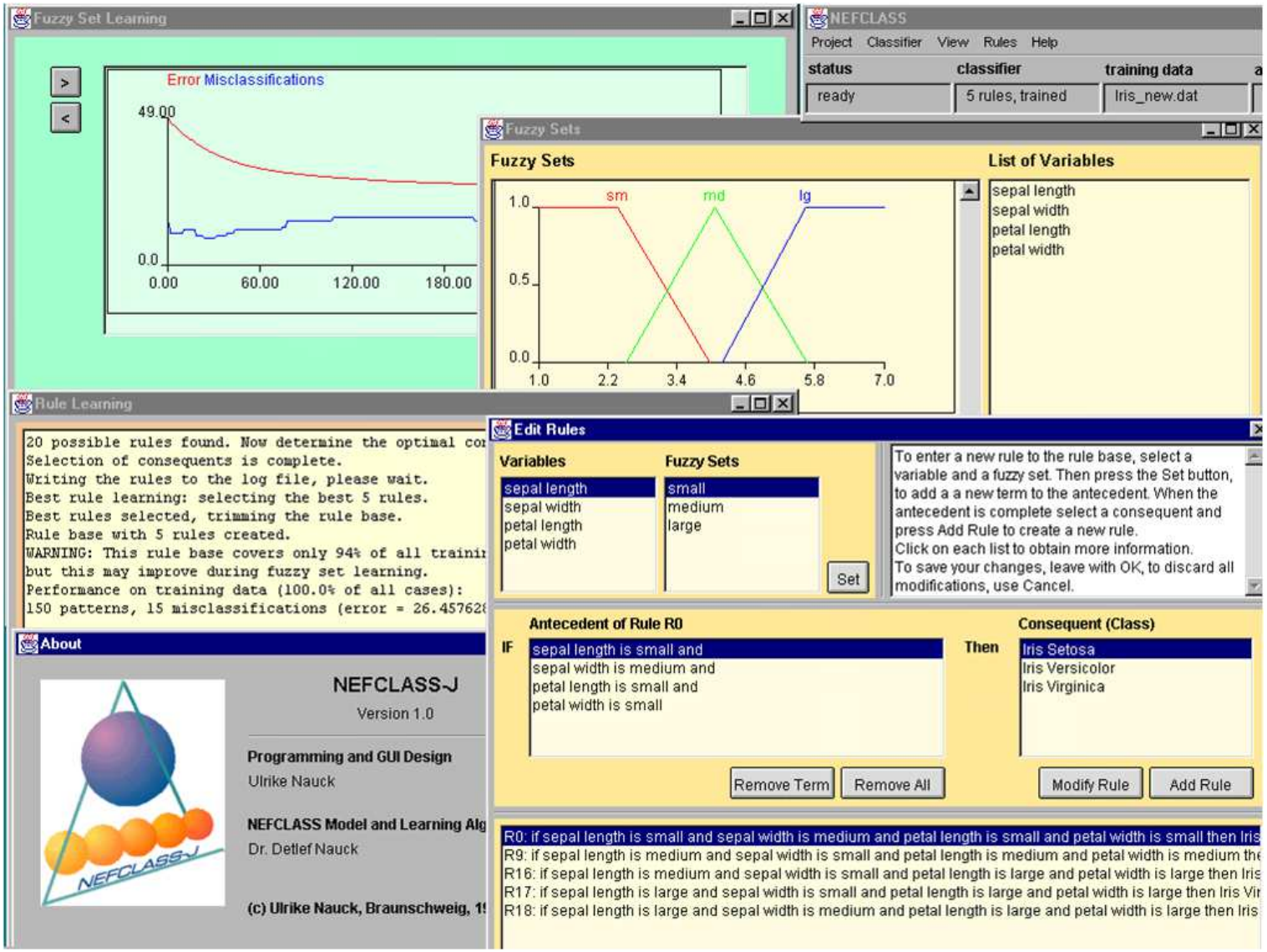
Geschätzte Vorhersageleistung auf unbekannten Daten (Kreuzvalidierung):

NEFCLASS-J:	95.42%	NEFCLASS-J (numerisch):	94.14%
Discriminant Analysis:	96.05%	Multilayer Perceptron:	94.82%
C 4.5:	95.10%	C 4.5 Rules:	95.40%

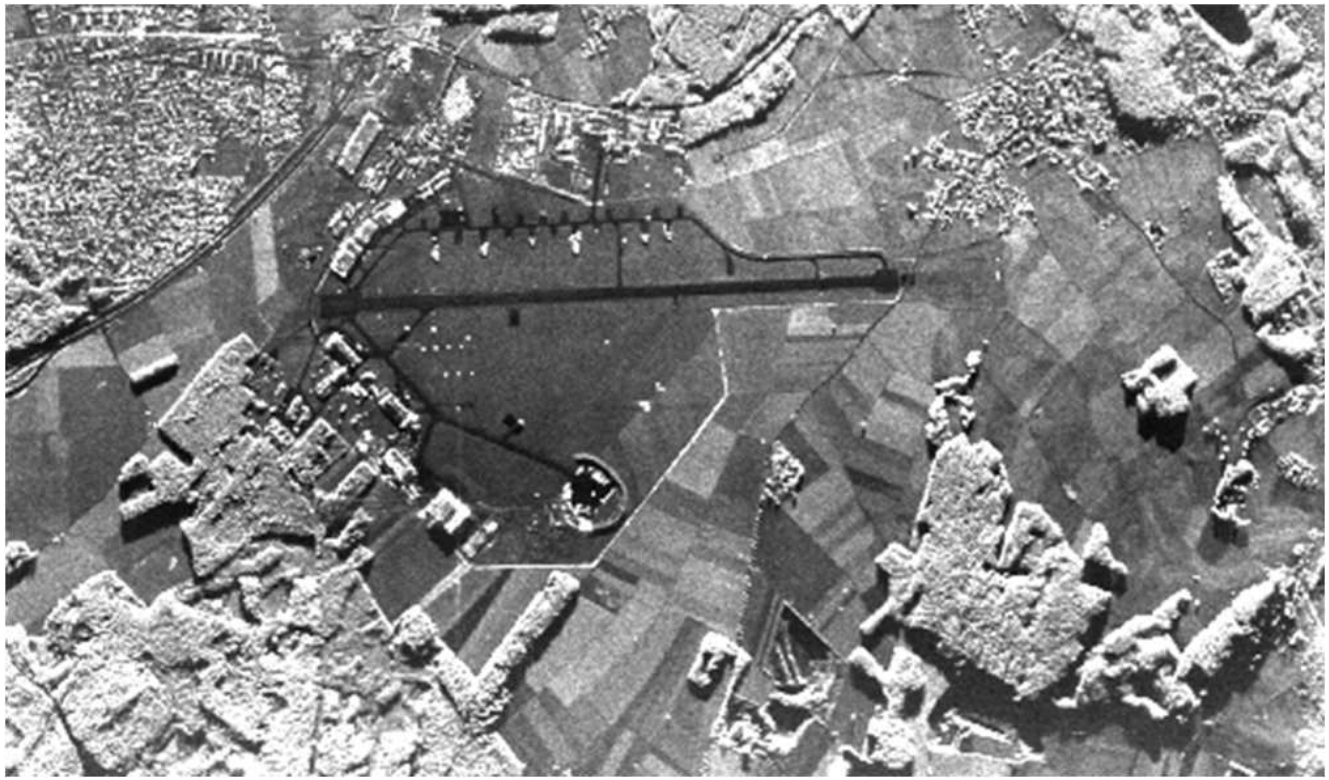
WBC-Ergebnisse: Fuzzy-Mengen



NEFCLASS-J

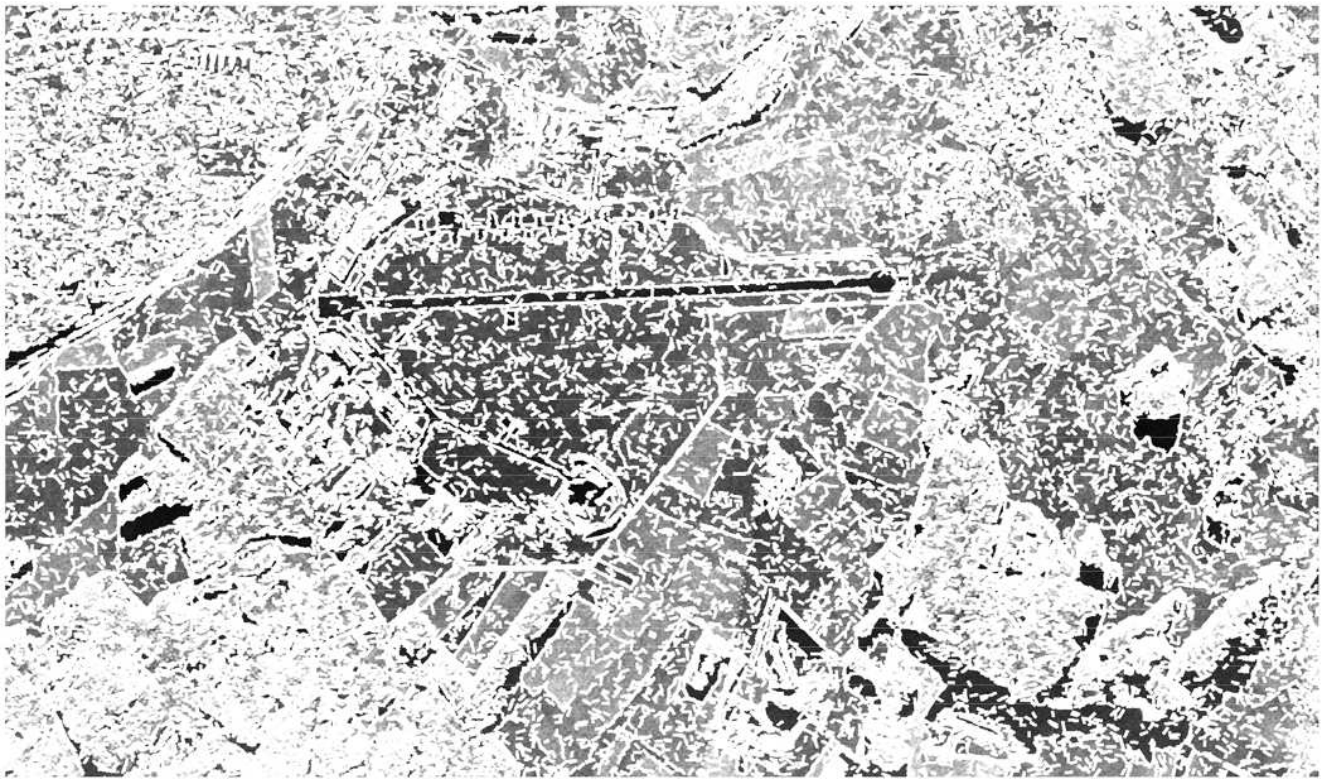

Fuzzy Set Learning
Error Misclassifications
49.00
0.0
0.00
60.00
120.00
180.00
NEFCLASS
Project Classifier View Rules Help
status
classifier
training data
ready
5 rules, trained
Iris_new.dat
Fuzzy Sets
Fuzzy Sets
List of Variables
sm
md
lg
1.0
0.5
0.0
1.0
2.2
3.4
4.6
5.8
7.0
sepal length
sepal width
petal length
petal width
Rule Learning
20 possible rules found. Now determine the optimal cor
Selection of consequents is complete.
Writing the rules to the log file, please wait.
Best rule learning: selecting the best 5 rules.
Best rules selected, trimming the rule base.
Rule base with 5 rules created.
WARNING: This rule base covers only 94% of all trainir
but this may improve during fuzzy set learning.
Performance on training data (100.0% of all cases):
150 patterns, 15 misclassifications (error = 26.457628
Edit Rules
Variables
Fuzzy Sets
sepal length
sepal width
petal length
petal width
small
medium
large
Set
To enter a new rule to the rule base, select a variable and a fuzzy set. Then press the Set button, to add a a new term to the antecedent. When the antecedent is complete select a consequent and press Add Rule to create a new rule.
Click on each list to obtain more information.
To save your changes, leave with OK, to discard all modifications, use Cancel.
About
NEFCLASS-J
Version 1.0
Programming and GUI Design
Ulrike Nauck
NEFCLASS Model and Learning Alg
Dr. Detlef Nauck
(c) Ulrike Nauck, Braunschweig, 19
Antecedent of Rule R0
IF
sepal length is small and
sepal width is medium and
petal length is small and
petal width is small
Then
Consequent (Class)
Iris Setosa
Iris Versicolor
Iris Virginica
Remove Term
Remove All
Modify Rule
Add Rule
R0: if sepal length is small and sepal width is medium and petal length is small and petal width is small then Iris
R9: if sepal length is medium and sepal width is small and petal length is medium and petal width is medium the
R16: if sepal length is medium and sepal width is small and petal length is large and petal width is large then Iris
R17: if sepal length is large and sepal width is small and petal length is large and petal width is large then Iris Vir
R18: if sepal length is large and sepal width is medium and petal length is large and petal width is large then Iris

Fallstudie: Linienerkennung



- Extraktion von Kantensegmenten (Burns' operator)
- weitere Schritte:
  Kanten → Linien → lange Linien → parallele Linien → Landebahnen

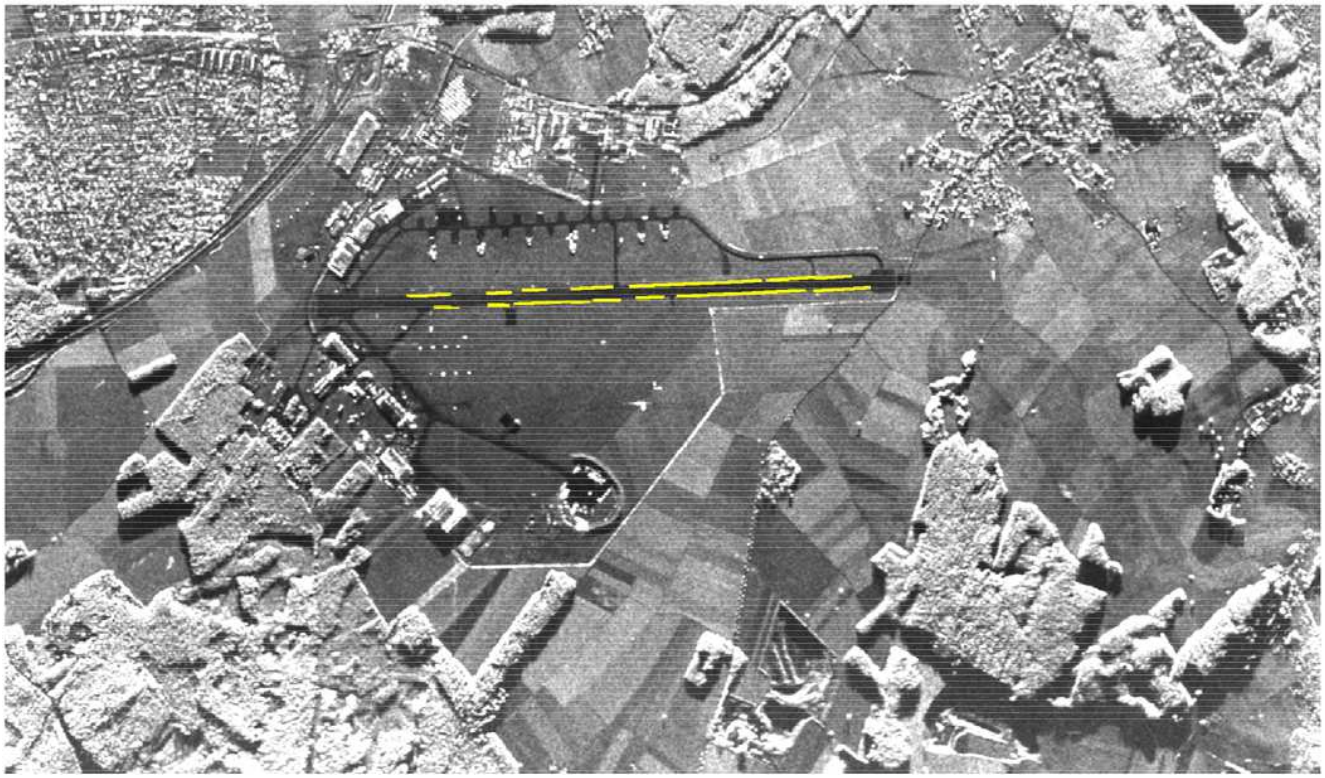
Fallstudie: Linienerkennung



Probleme:

- sehr viele Linien wegen verzerrter Bilder
- Lange Ausführungszeiten der Erstellungs-Schritte (bis Landebahnen)

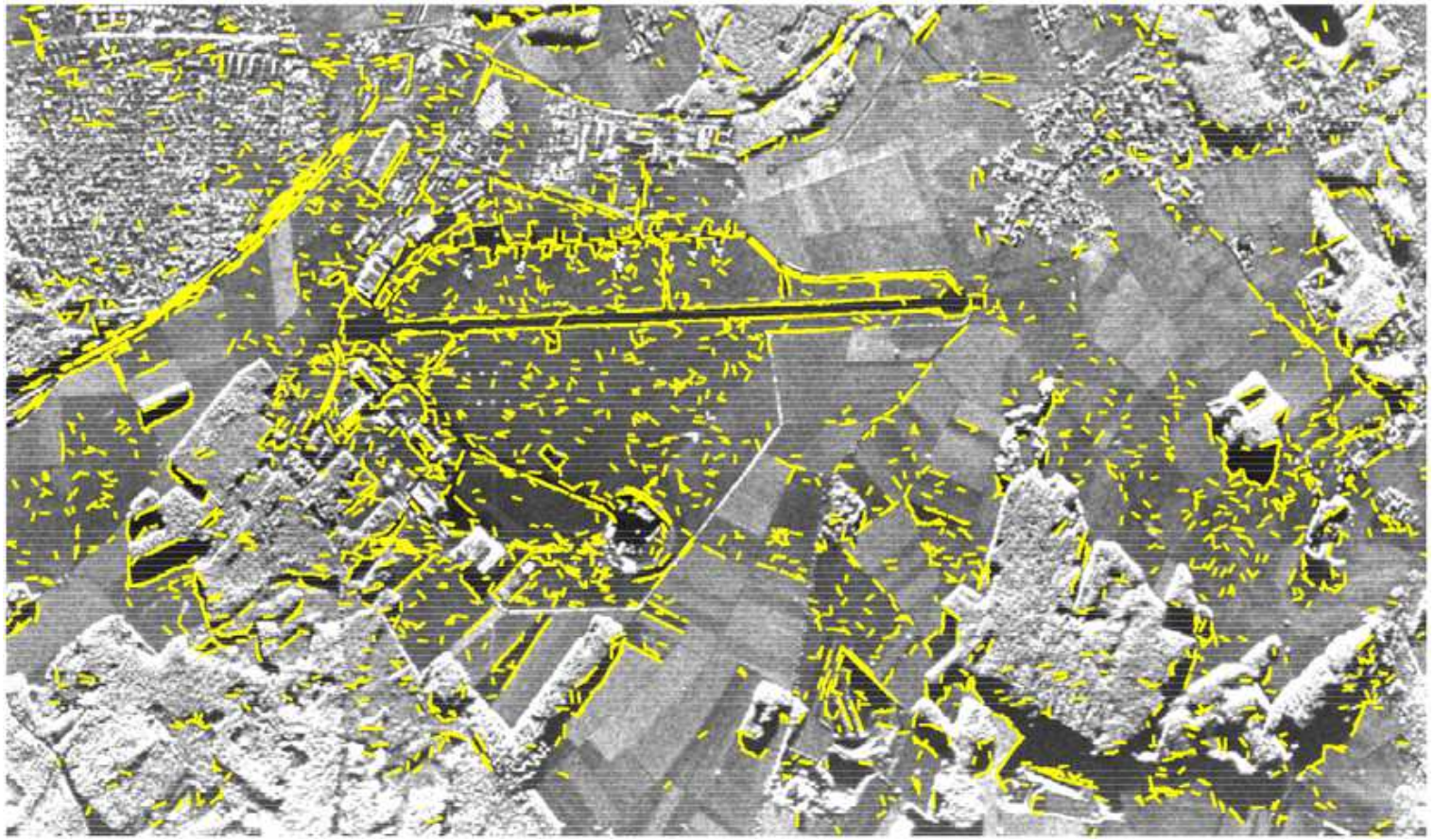
Fallstudie: Linienerkennung



nur wenige Linien für Landebahnen genutzt
Ansatz:

- extrahiere Textureigenschaften der Linien
- identifiziere und verwirf überflüssige Linien

Fallstudie: Linienerkennung



mehrere Klassifikatoren: minimum distance, k-NN, C 4.5, NEFCLASS
Probleme: Klassen überlappen und sind extrem unbalanciert
obiges Ergebnis durch modifiziertes NEFCLASS erhalten:

- alle Landebahnlinien gefunden, Reduktion auf 8.7% der Kantensegmente

Zusammenfassung

- Neuro-Fuzzy-Systeme können nützlich zur Wissensentdeckung sein
- Interpretierbarkeit ermöglicht die Plausibilitätskontrolle und erhöht die Akzeptanz
- NFS nutzen Toleranzen aus, um zu beinahe optimalen Lösungen zu kommen
- NFS-Lernalgorithmen müssen mit Einschränkungen umgehen können, um die Semantik des ursprünglichen Modells nicht zu verletzen
- keine automatische Modellerstellung $\Rightarrow$ Benutzer muß mit dem Werkzeug **umgehen**
- Einfache Lerntechniken unterstützen die explorative Datenanalyse.

Fallstudie: Qualitätskontrolle

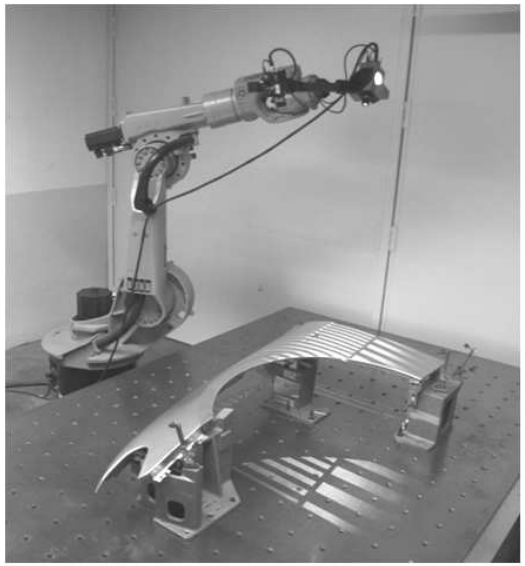
Heutiges Verfahren

- Oberflächenkontrolle:
  manuell durchgeführt
- erfahrener Arbeiter bearbeitet
  Oberfläche mit Schleifstein
- Experten klassifizieren Abweichungen
  durch sprachliche Beschreibungen
- umständlich, subjektiv,
  fehleranfällig, zeitaufwendig

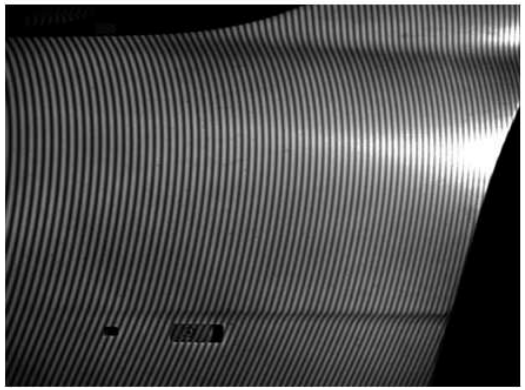
vorgeschlagener Ansatz:

- Digitalisierung der Oberfläche mit optischen Mess-Systemen
- Charakterisierung der Formabweichungen durch mathematische Eigenschaften (nahe der subjektiven Merkmale)

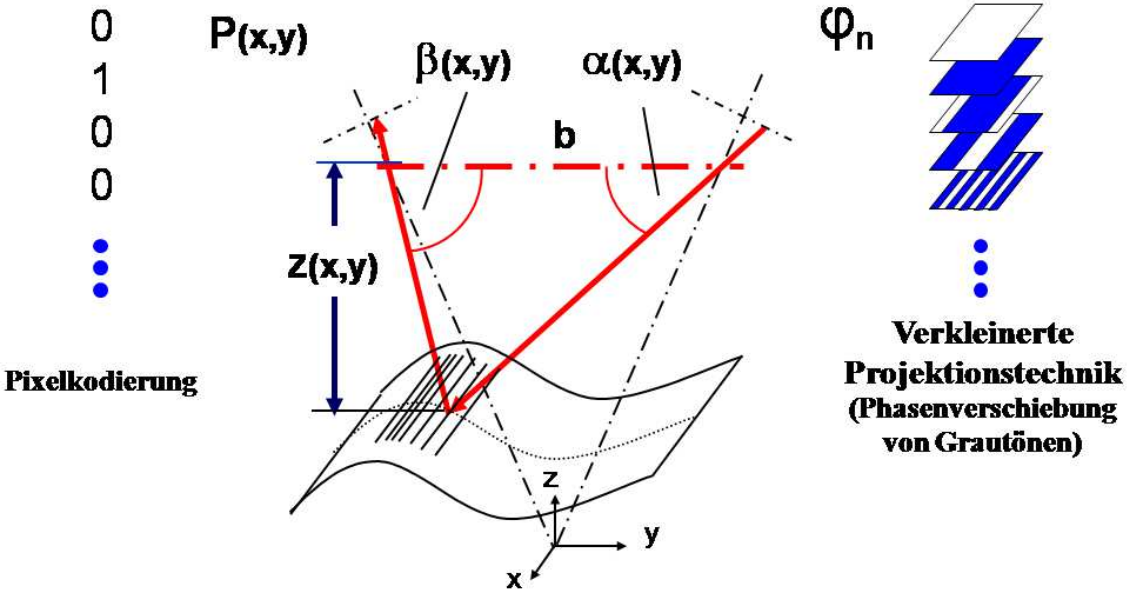
Topometrisches 3D Mess-System



breuckmann

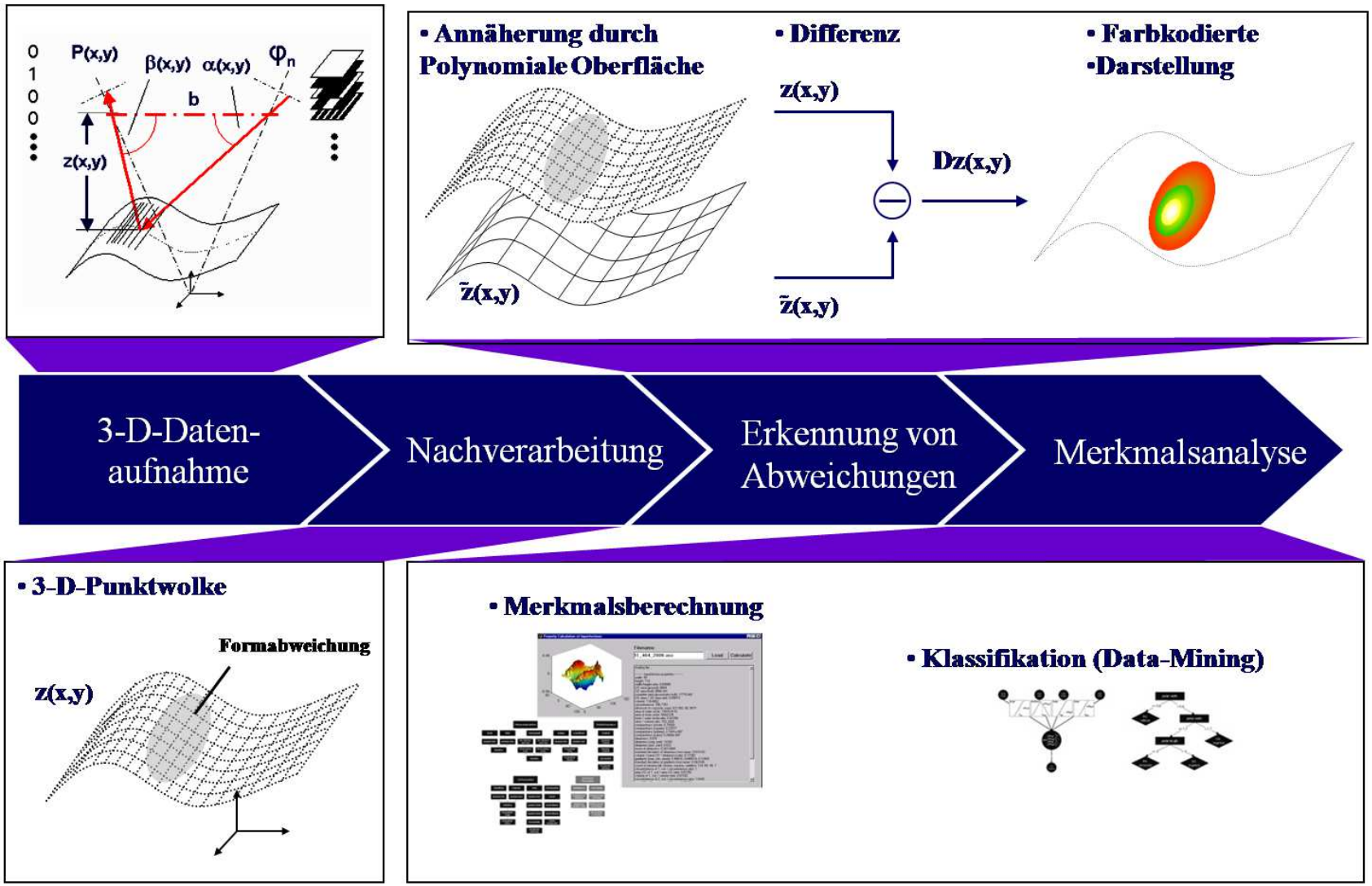


Triangulation und Gitterprojektion

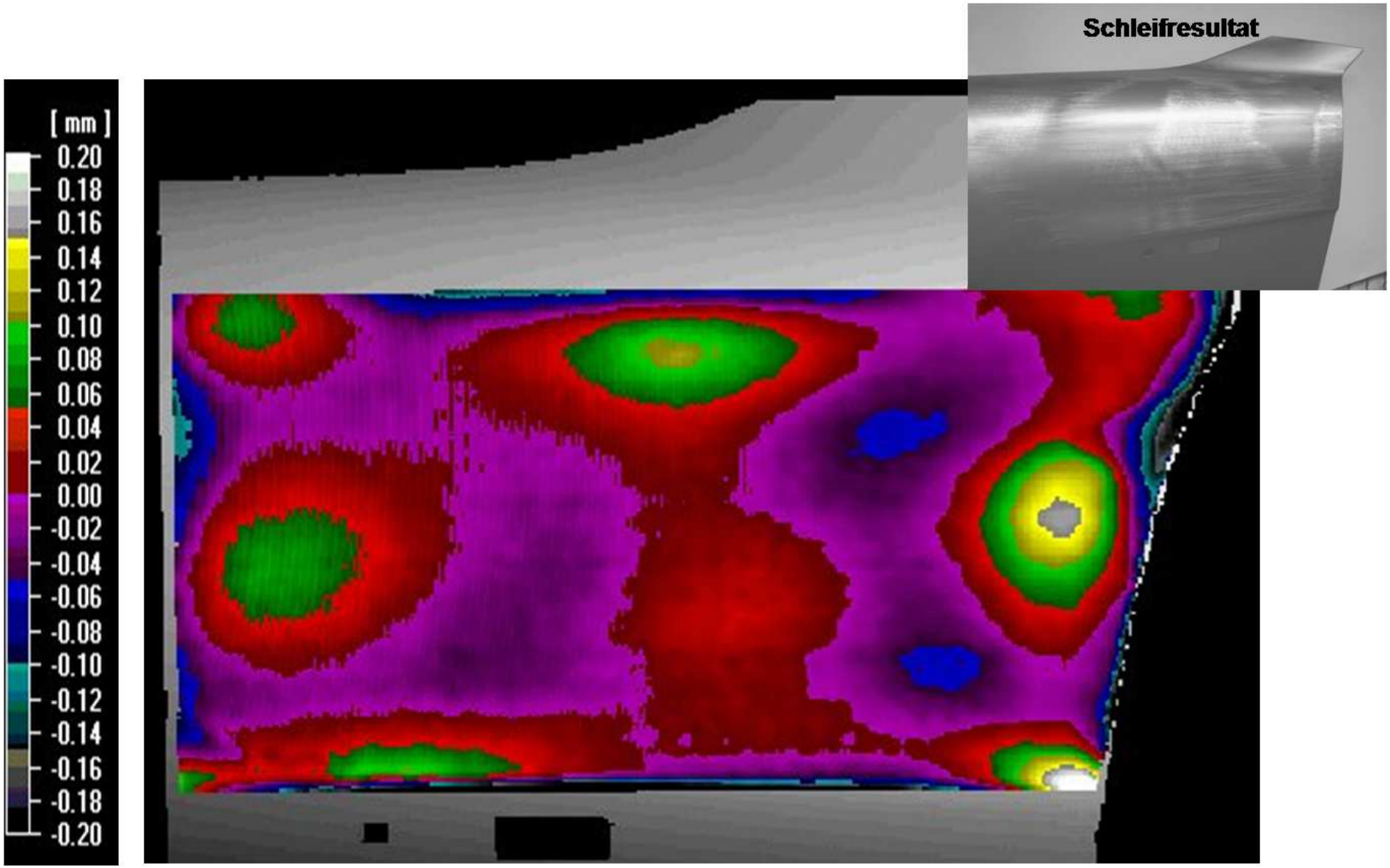


- hohe Punktdichte
- schnelle Datenansammlung
- genaue Messung
- kontakt- und harmlos

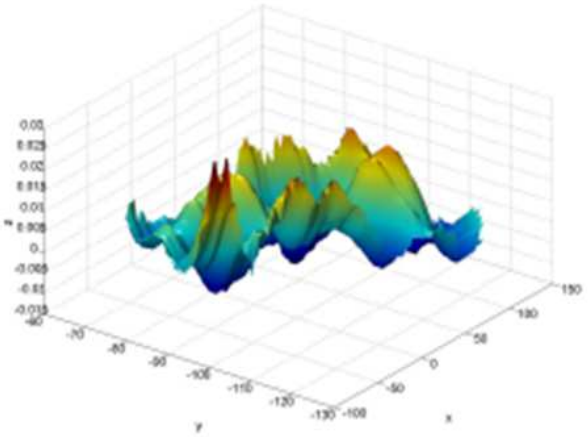
Datenverarbeitung



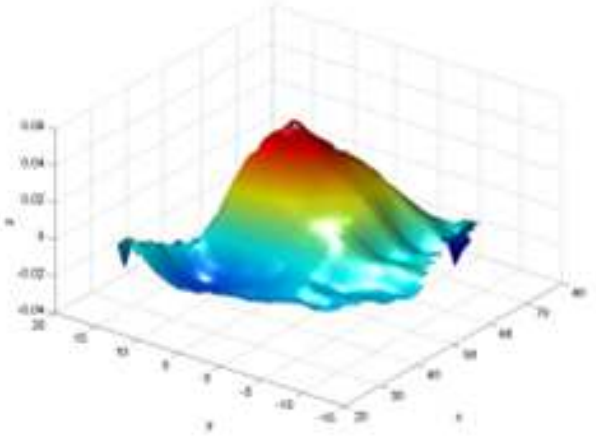
Farbkodierte Darstellung



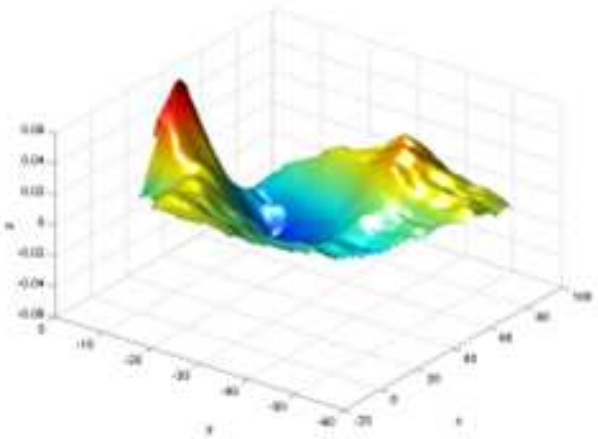
3D-Darstellung lokaler Oberflächendefekte



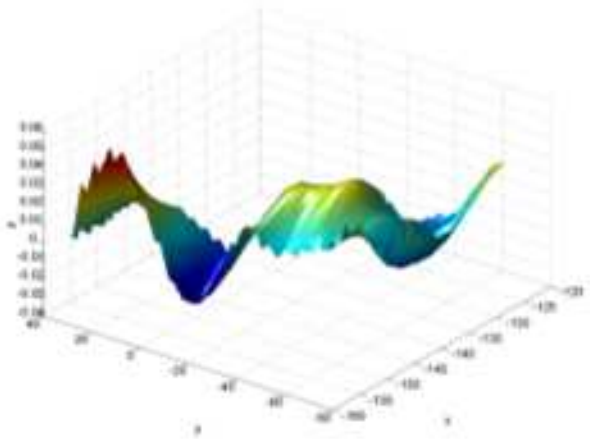
unebene Oberfläche
mehrere Einfallsstellen in Serie/benachbart



Walzenmarkierung
lokale Glättung der Oberfläche



Einfallstelle
leichte flach basierte Senke einwärts



Wellplatte
mehrere schwerere Faltungen in Serie

Charakteristik der Daten

- 9 Meisterstücke mit insgesamt 99 Defekten analysiert
- für jeden Defekt, 42 Merkmale berechnet
- Typen sind eher unbalanciert
- seltene Klassen verworfen
- einige extrem korrelierte Merkmale verworfen (31 übrig)
- Rangfolge der 31 Merkmale nach Wichtigkeit
- geschichtete 4-fache Kreuzvalidierung fürs Experiment

Regelbasis für NEFCLASS:

Rule base
- Rule 1: IF (max_distance_to_cog IS fun 2 AND min_extrema IS fun 1 AND max_extrema IS fun 1) THEN type IS press_mark
- Rule 2: IF (max_distance_to_cog IS fun 2 AND all_extrema IS fun 1 AND max_extrema IS fun 2) THEN type IS sink_mark
- Rule 3: IF (max_distance_to_cog IS fun 3 AND min_extrema IS fun 2 AND max_extrema IS fun 2) THEN type IS uneven_surface
- Rule 4: IF (max_distance_to_cog IS fun 2 AND min_extrema IS fun 2 AND max_extrema IS fun 2) THEN type IS uneven_surface
- Rule 5: IF (max_distance_to_cog IS fun 2 AND all_extrema IS fun 1 AND min_extrema IS fun 2) THEN type IS press_mark
- Rule 6: IF (max_distance_to_cog IS fun 3 AND all_extrema IS fun 2 AND max_extrema IS fun 3) THEN type IS uneven_surface
- Rule 7: IF (max_distance_to_cog IS fun 3 AND min_extrema IS fun 3) THEN type IS uneven_surface

Klassifikationsgenauigkeit:

	NBC	DTree	NN	NEFCLASS	DC
Trainingsmenge	89.0%	94.7%	90%	81.6%	46.8%
Testmenge	75.6%	75.6%	85.5%	79.9%	46.8%