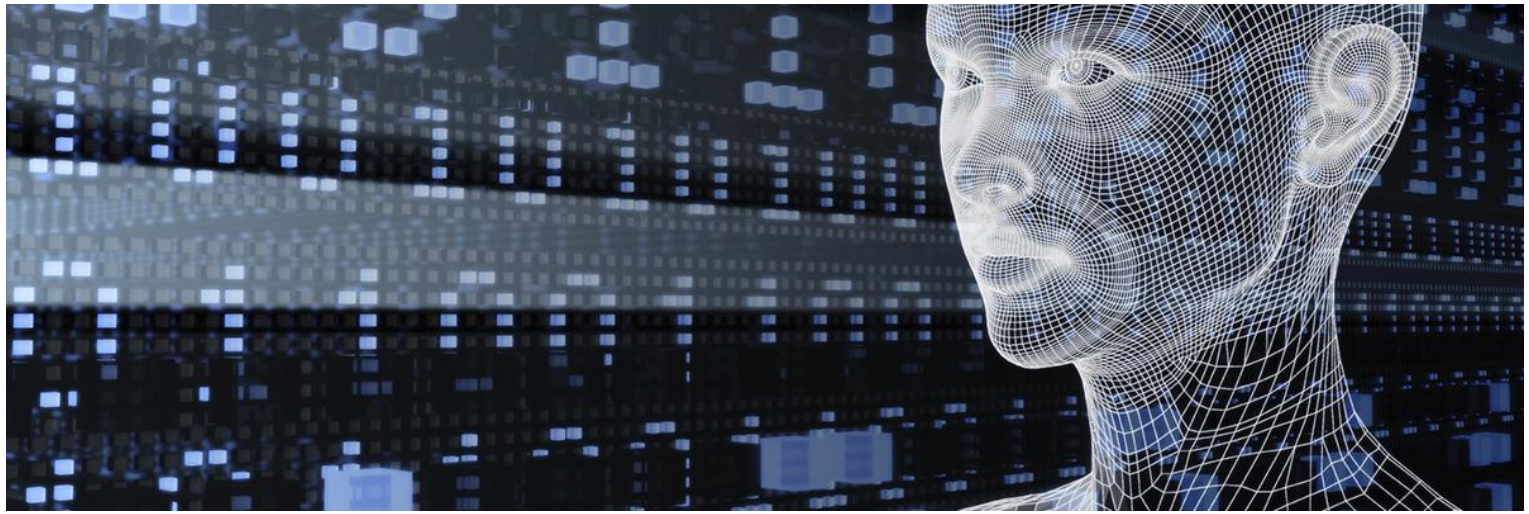
Künstliche Intelligenz

Vorlesung 11&12: Neuronale Netze



Neuronale Netze: Motivation

- **(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

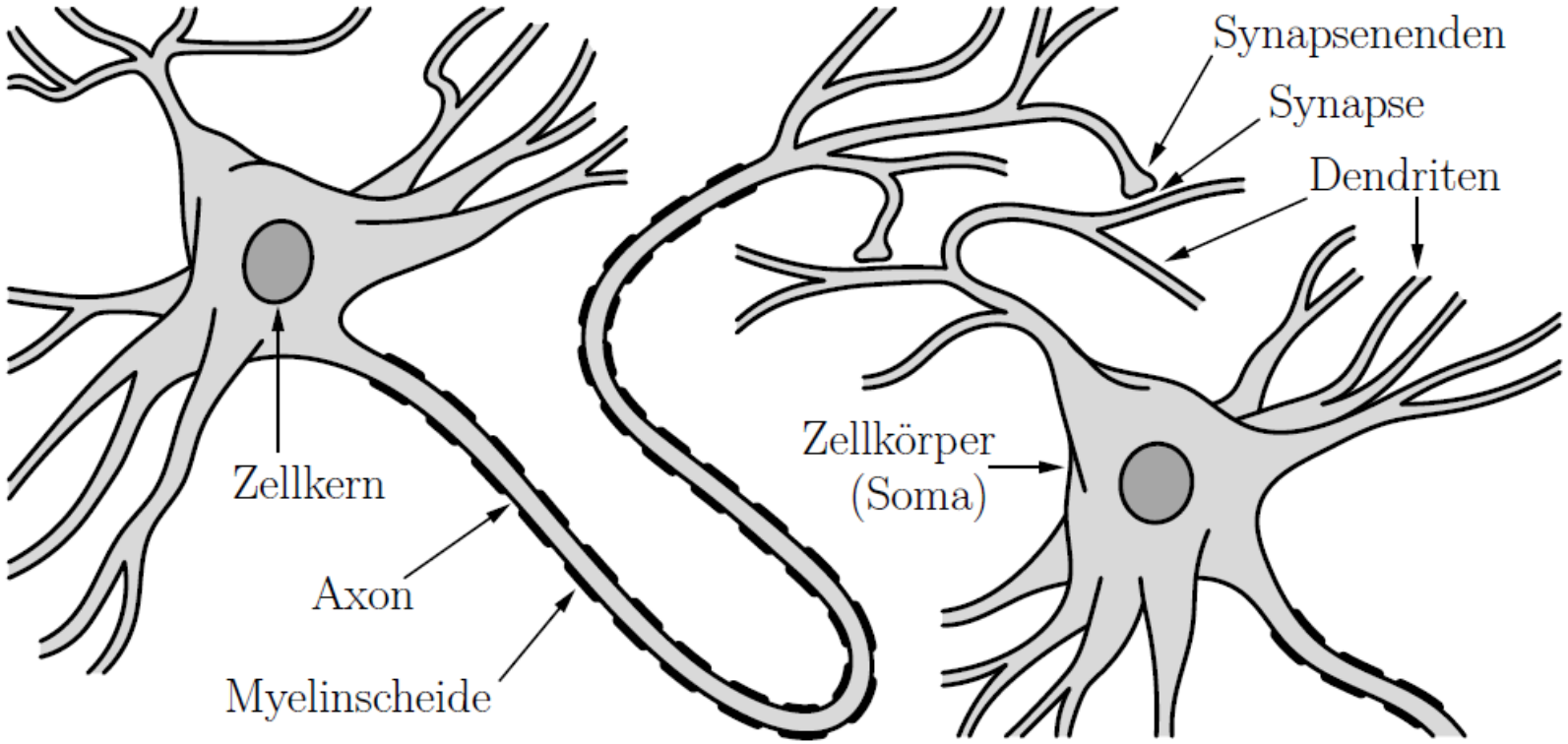
	Computer	Gehirn
Verarbeitungseinheiten	1 CPU, 10^9 Transistoren	10^{11} Neuronen
Speicherkapazität	10^9 Bytes RAM, 10^{10} Bytes Fest-speicher	10^{11} Neuronen, 10^{14} Synapsen
Verarbeitungs-geschwindigkeit	10^{-8} sec.	10^{-3} sec.
Bandbreite	$10^9 \frac{bits}{s}$	$10^{14} \frac{bits}{s}$
Neuronale 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.

Biologischer Hintergrund

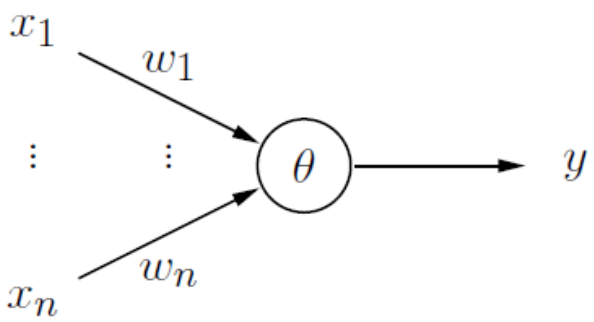
Struktur eines prototypischen biologischen Neurons



Schwellenwertelemente

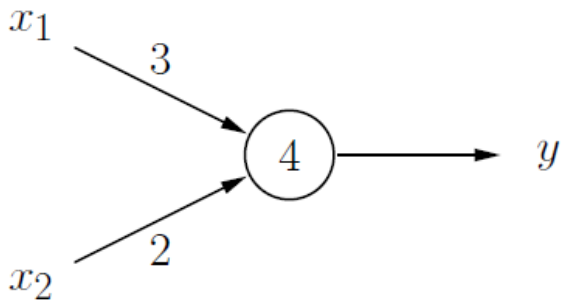
Ein **Schwellenwertelement** (Threshold Logic Unit, TLU) ist eine Verarbeitungseinheit für Zahlen mit n Eingängen $x_1, \ldots, x_n$ und einem Ausgang y. Das Element hat einen **Schwellenwert** θ und jeder Eingang x_i ist mit einem **Gewicht** w_i versehen. Ein Schwellenwertelement berechnet die Funktion

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



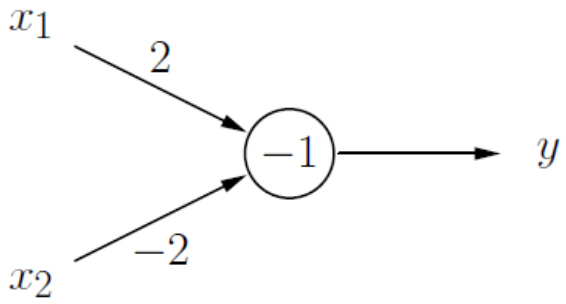
Schwellenwertelemente: Beispiele

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

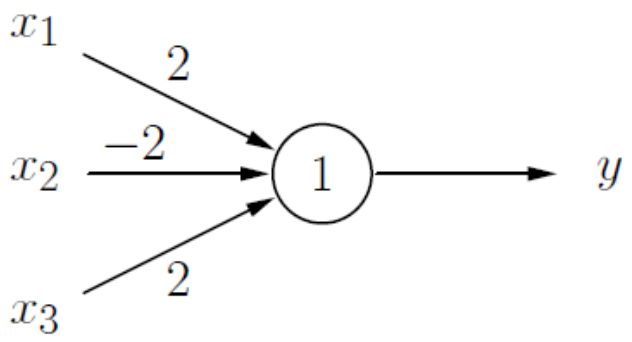
Schwellenwertelement für die Implikation $x_2 \rightarrow x_1$.



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

Schwellenwertelemente: Geometrische Interpretation

Rückblick: Geradendarstellungen

Geraden werden typischerweise in einer der folgenden Formen dargestellt:

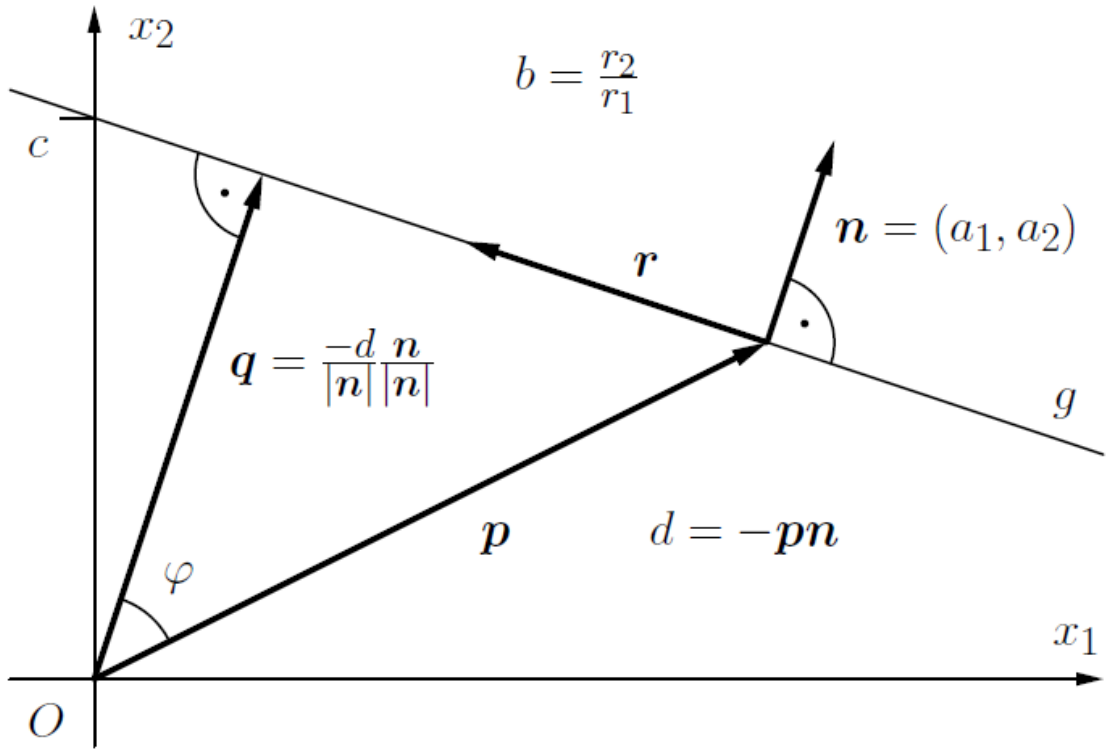
Explizite Form:	$g \equiv$	$x_2 = bx_1 + c$
Implizite Form:	$g \equiv$	$a_1x_1 + a_2x_2 + d = 0$
Punkt-Richtungs-Form:	$g \equiv$	$\boldsymbol{x} = \boldsymbol{p} + k\boldsymbol{r}$
Normalform	$g \equiv$	$(\boldsymbol{x} - \boldsymbol{p})\boldsymbol{n} = 0$

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

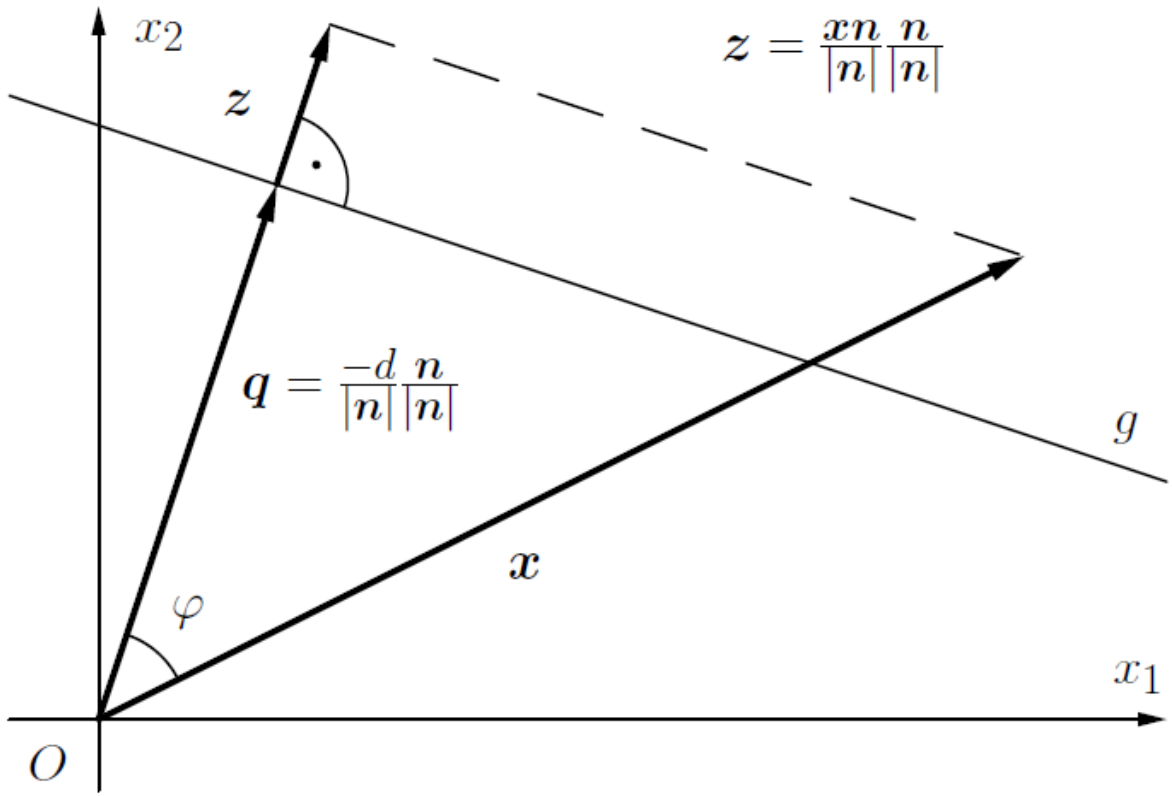
Schwellenwertelemente: Geometrische Interpretation

Eine Gerade und ihre definierenden Eigenschaften.



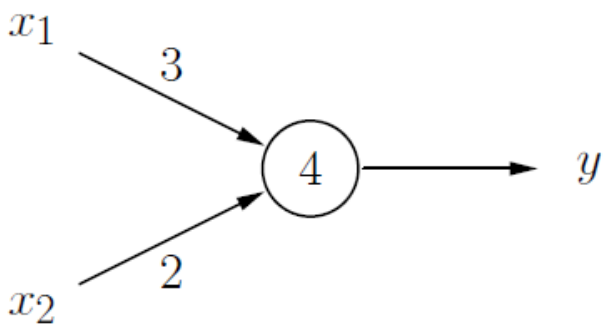
Schwellenwertelemente: Geometrische Interpretation

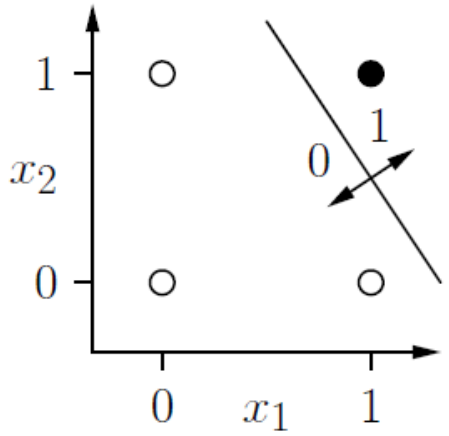
Bestimmung, auf welcher Seite ein Punkt x liegt.



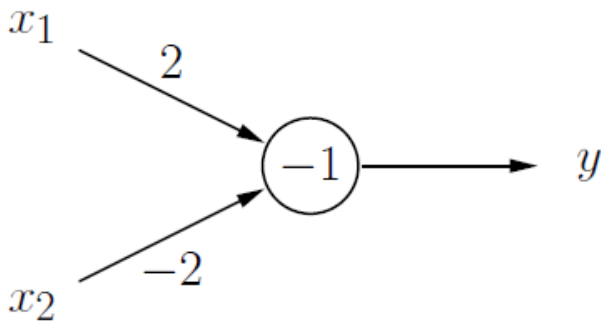
Schwellenwertelemente: Geometrische Interpretation

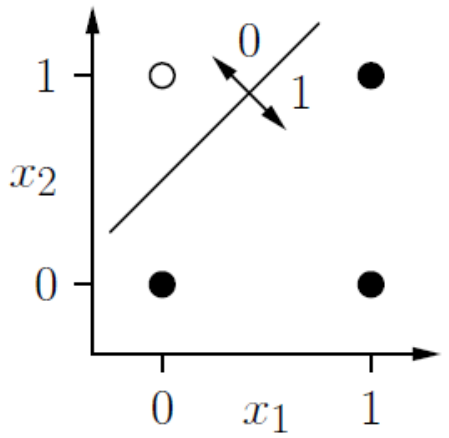
Schwellenwertelement für $x_1 \wedge x_2$.





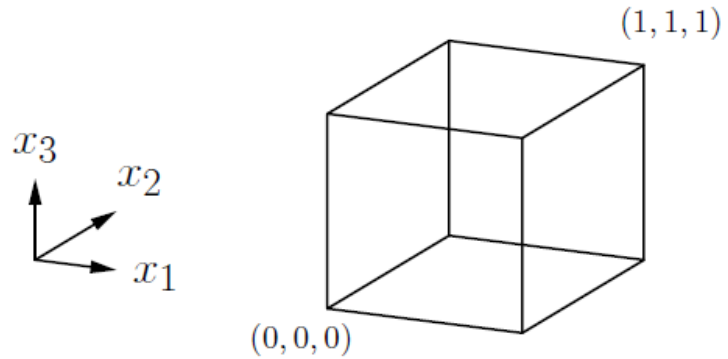
Ein Schwellenwertelement für $x_2 \rightarrow x_1$.



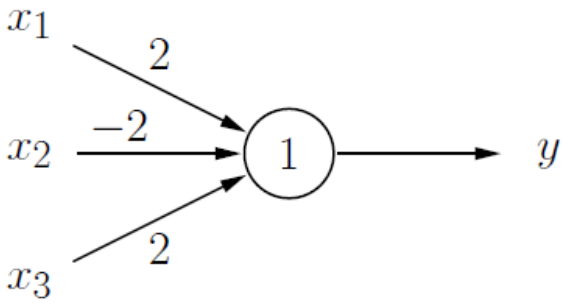


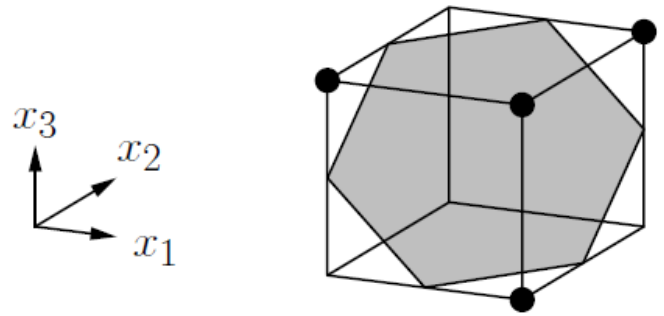
Schwellenwertelemente: Geometrische Interpretation

Darstellung 3-dimensionaler Boolescher Funktionen:



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





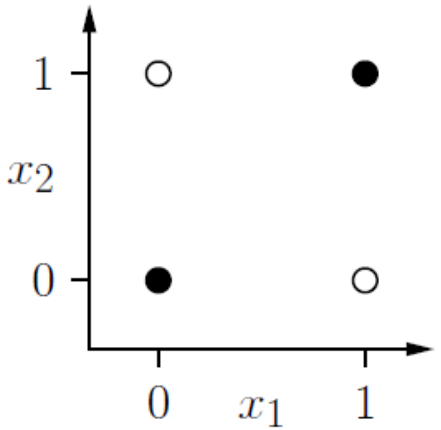
Schwellenwertelemente: lineare Separabilität

- 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.

Schwellenwertelemente: Grenzen

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*:

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

(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])

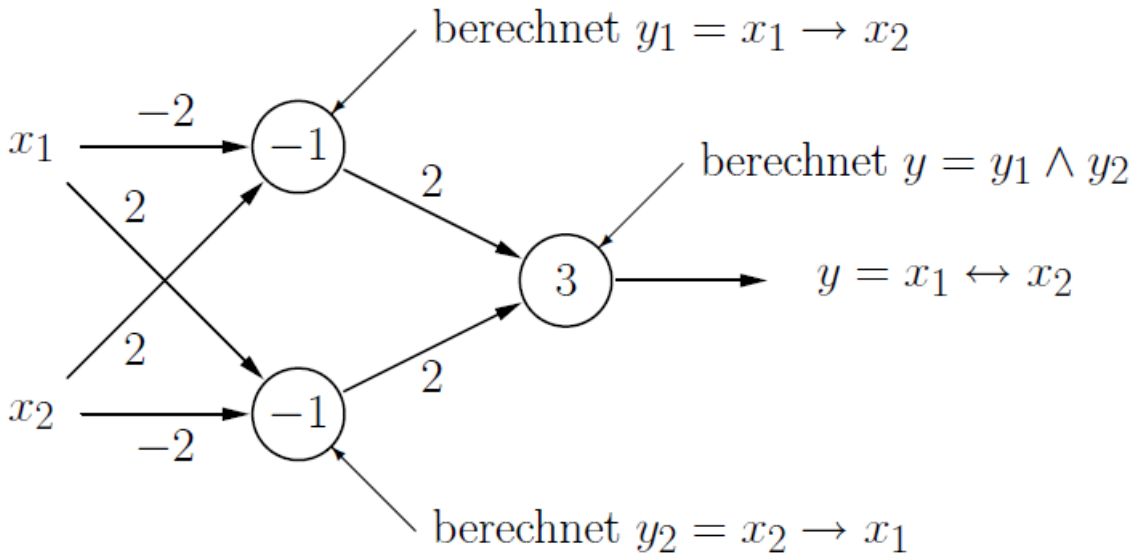
Eingaben	Boolesche Funktionen	linear separable Funktionen
1	4	4
2	16	14
3	256	104
4	65536	1774
5	$4.3 \cdot 10^9$	94572
6	$1.8 \cdot 10^{19}$	$5.0 \cdot 10^6$

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

Netze aus Schwellenwertelementen

Biimplikationsproblem, Lösung durch ein Netzwerk.

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



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

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

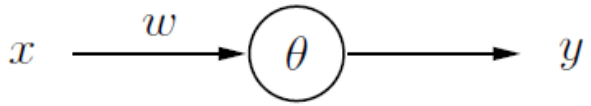
Trainieren von Schwellenwertelementen

- 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.

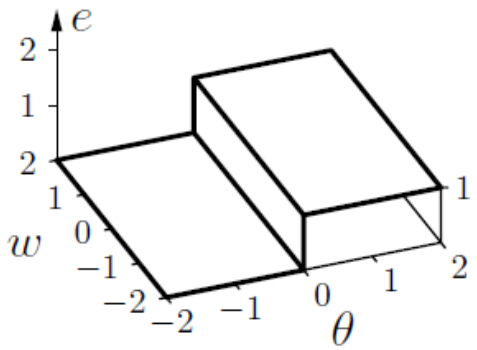
Trainieren von Schwellenwertelementen

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

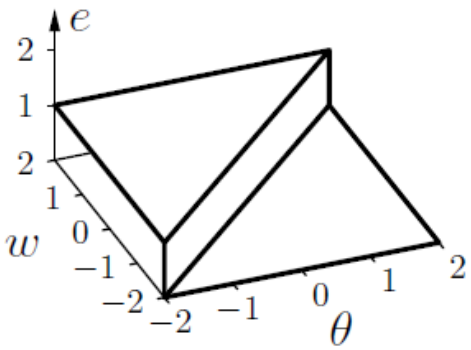


x	y
0	1
1	0

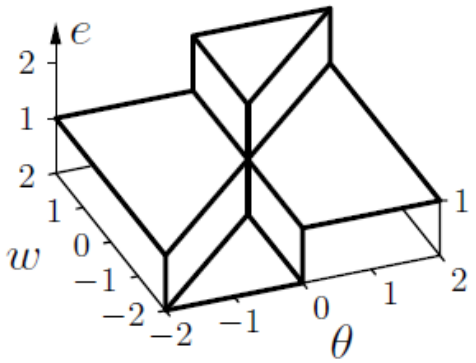
Ausgabefehler als eine Funktion von Gewicht und Schwellenwert.



Fehler für $x = 0$



Fehler für $x = 1$

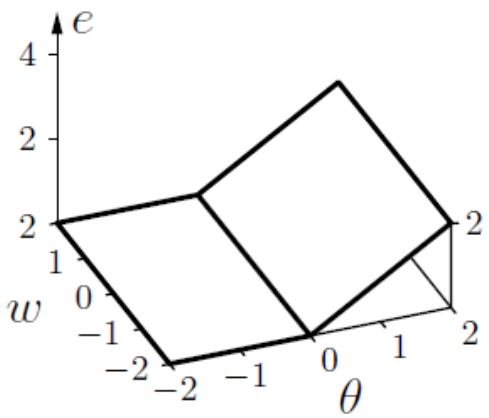


Summe der Fehler

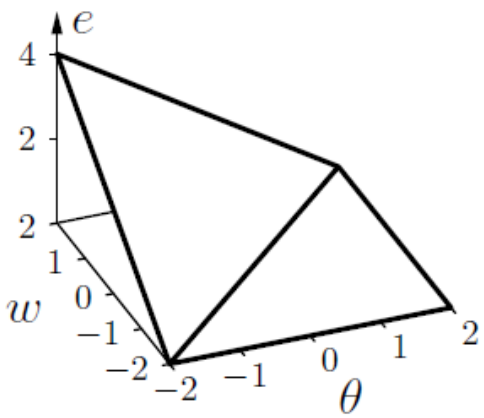
Trainieren von Schwellenwertelementen

- 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.

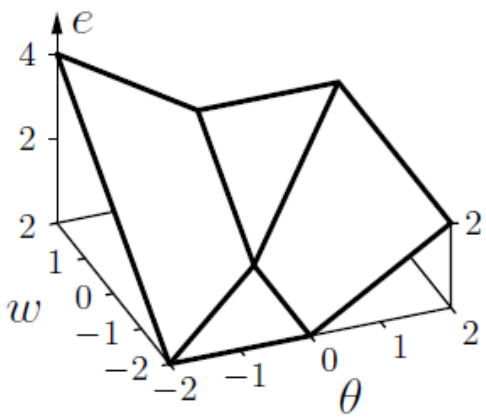
Modifizierter Ausgabefehler als Funktion von Gewichten und Schwellenwert.



Fehler für $x = 0$



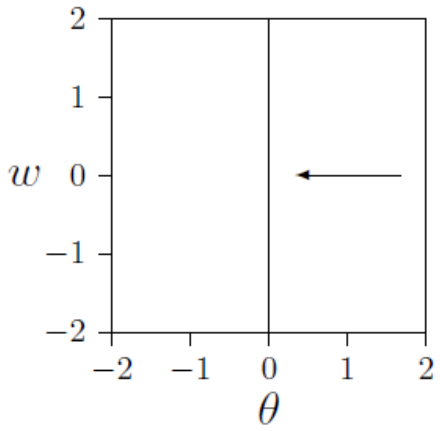
Fehler für $x = 1$



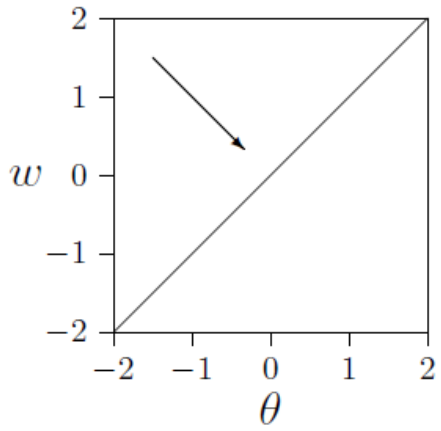
Summe der Fehler

Trainieren von Schwellenwertelementen

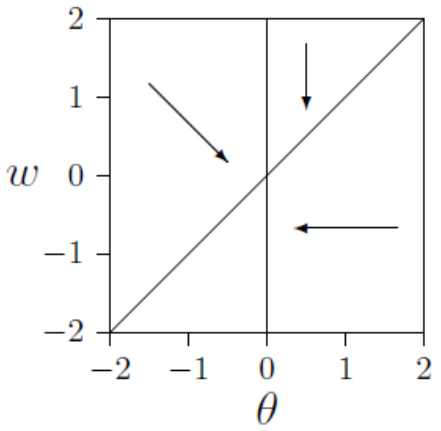
Schema der resultierenden Richtungen der Parameteränderungen.



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.

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:

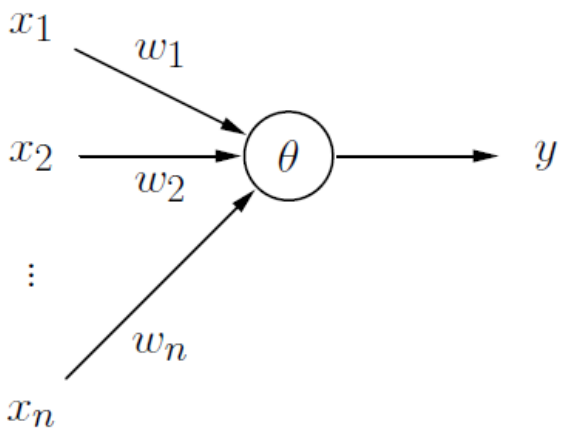
$$\begin{aligned} \theta^{(\text{neu})} &= \theta^{(\text{alt})} + \Delta\theta \quad \text{wobei} \quad \Delta\theta &&= -\eta(o - y), \\ \forall i \in \{1, \ldots, n\}: \quad w_i^{(\text{neu})} &= w_i^{(\text{alt})} + \Delta w_i \quad \text{wobei} \quad \Delta w_i &&= \eta(o - y)x_i, \end{aligned}$$

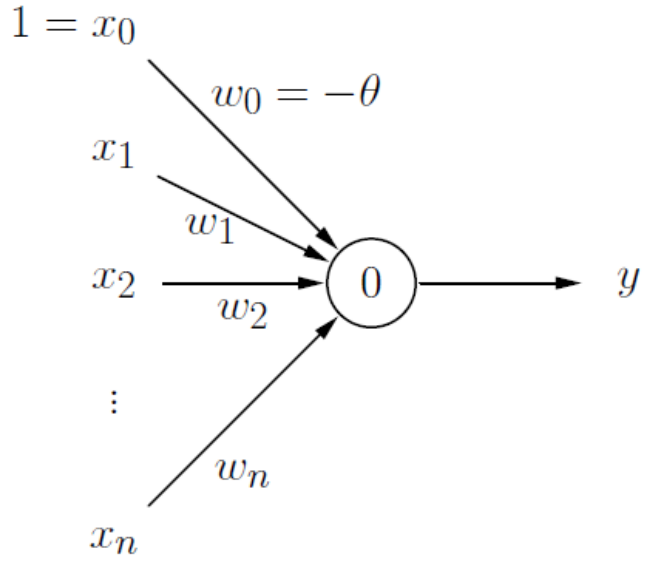
wobei η ein Parameter ist, der **Lernrate** genannt wird. Er bestimmt die Größenordnung der Gewichtsänderungen. Diese Vorgehensweise nennt sich **Delta-Regel** oder **Widrow–Hoff–Procedure** [Widrow and Hoff 1960].

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

Trainieren von Schwellenwertelementen: Delta-Regel

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

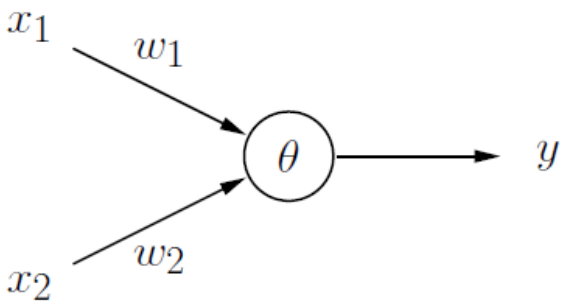
Epoche	x	o	$\boldsymbol{xw}$	y	e	$\Delta\theta$	Δw	θ	w
								1.5	2
1	0	1	−1.5	0	1	−1	0	0.5	2
	1	0	1.5	1	−1	1	−1	1.5	1
2	0	1	−1.5	0	1	−1	0	0.5	1
	1	0	0.5	1	−1	1	−1	1.5	0
3	0	1	−1.5	0	1	−1	0	0.5	0
	1	0	0.5	0	0	0	0	0.5	0
4	0	1	−0.5	0	1	−1	0	−0.5	0
	1	0	0.5	1	−1	1	−1	0.5	−1
5	0	1	−0.5	0	1	−1	0	−0.5	−1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

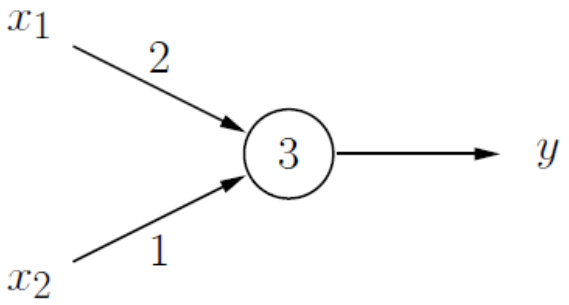
Epoche	x	o	$\boldsymbol{xw}$	y	e	$\Delta\theta$	Δw	θ	w
								1.5	2
1	0	1	−1.5	0	1	−1	0		
	1	0	0.5	1	−1	1	−1	1.5	1
2	0	1	−1.5	0	1	−1	0		
	1	0	−0.5	0	0	0	0	0.5	1
3	0	1	−0.5	0	1	−1	0		
	1	0	0.5	1	−1	1	−1	0.5	0
4	0	1	−0.5	0	1	−1	0		
	1	0	−0.5	0	0	0	0	−0.5	0
5	0	1	0.5	1	0	0	0		
	1	0	0.5	1	−1	1	−1	0.5	−1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

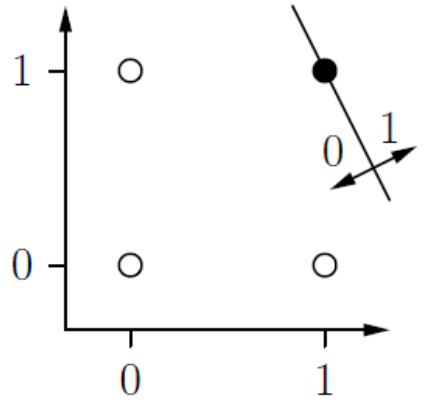
Trainieren von Schwellenwertelementen: Konjunktion

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 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 $u \in U_{\text{in}}$ besitzt weiterhin eine vierte reellwertige Zustandsvariable,

- die **externe Eingabe** ex_u.

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

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

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

Allgemeine Neuronale Netze

Typen (künstlicher) neuronaler Netze

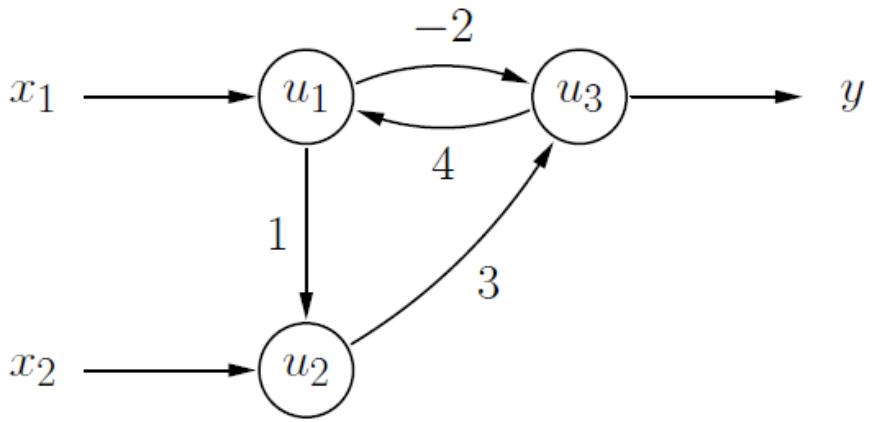
- Falls der Graph eines neuronalen Netzes **azyklisch** ist, wird das Netz **Feed-Forward-Netzwerk** genannt.
- Falls der Graph eines neuronalen Netzes **Zyklen** enthält, (rückwärtige Verbindungen), wird es **rekurrentes Netzwerk** genannt.

Darstellung der Verbindungsgewichte als Matrix

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

Allgemeine Neuronale Netze: Beispiel

Ein einfaches rekurrentes neuronales Netz:

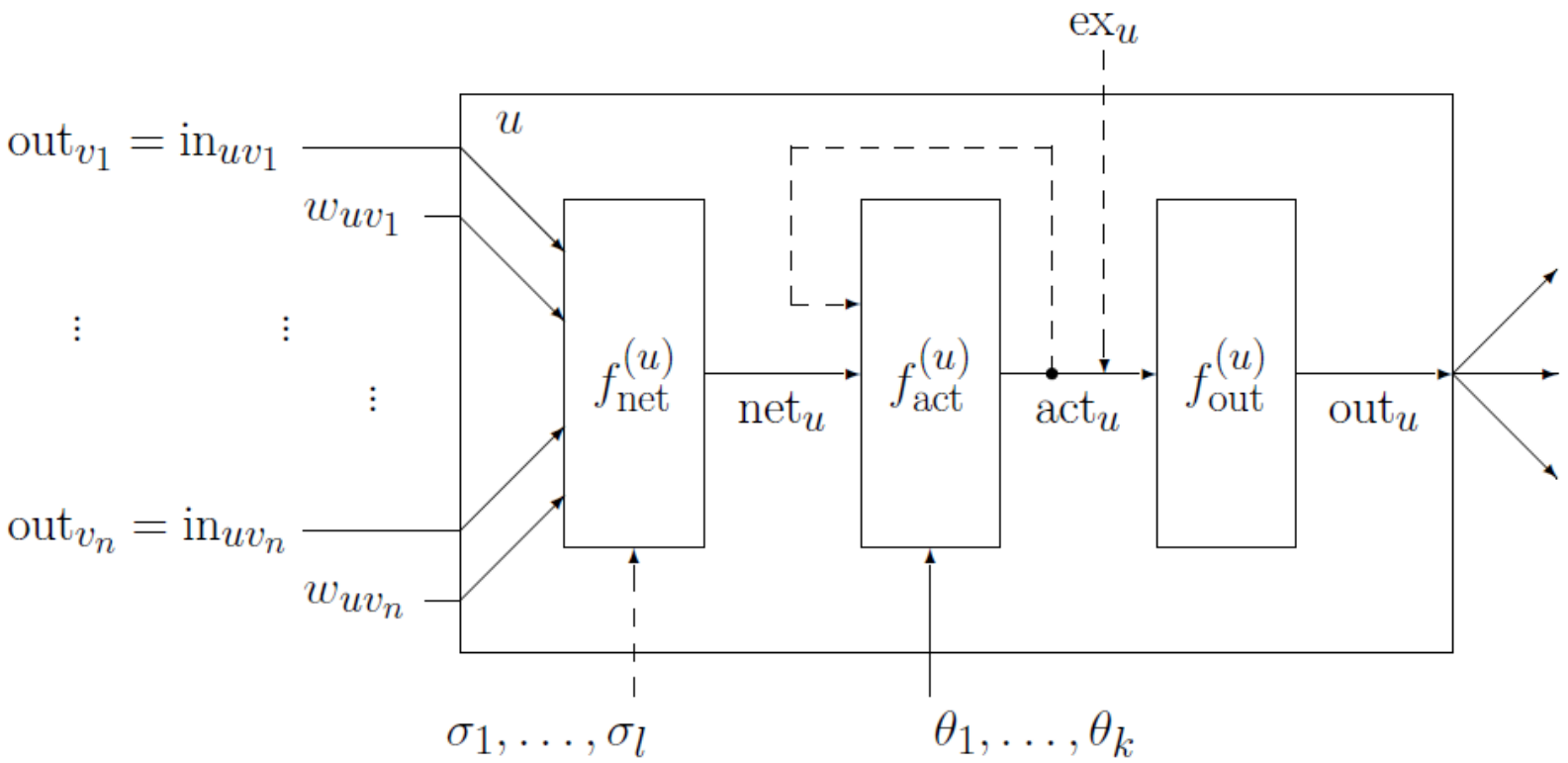


Gewichtsmatrix dieses Netzes:

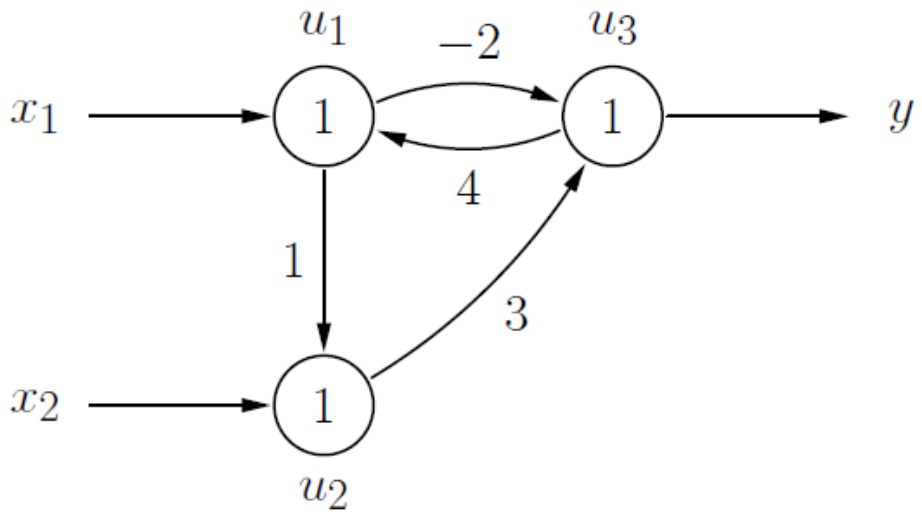
$$\begin{array}{c} \\ u_1 \\ u_2 \\ u_3 \end{array} \begin{array}{c} \begin{array}{ccc} u_1 & u_2 & u_3 \end{array} \\ \begin{pmatrix} 0 & 0 & 4 \\ 1 & 0 & 0 \\ -2 & 3 & 0 \end{pmatrix} \end{array}$$

Struktur eines künstlichen Neurons

Ein verallgemeinertes Neuron verarbeitet numerische Werte



Allgemeine Neuronale Netze: Beispiel



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

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

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

Allgemeine Neuronale Netze: Beispiel

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.

Allgemeine Neuronale Netze: Beispiel

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).

Allgemeine Neuronale Netze: Training

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.

Allgemeine Neuronale Netze: Training

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.

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

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

Allgemeine Neuronale Netze: Training

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** $l = (\boldsymbol{\imath}^{(l)})$, wobei jedes aus

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

Allgemeine Neuronale Netze: Vorverarbeitung

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)

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

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

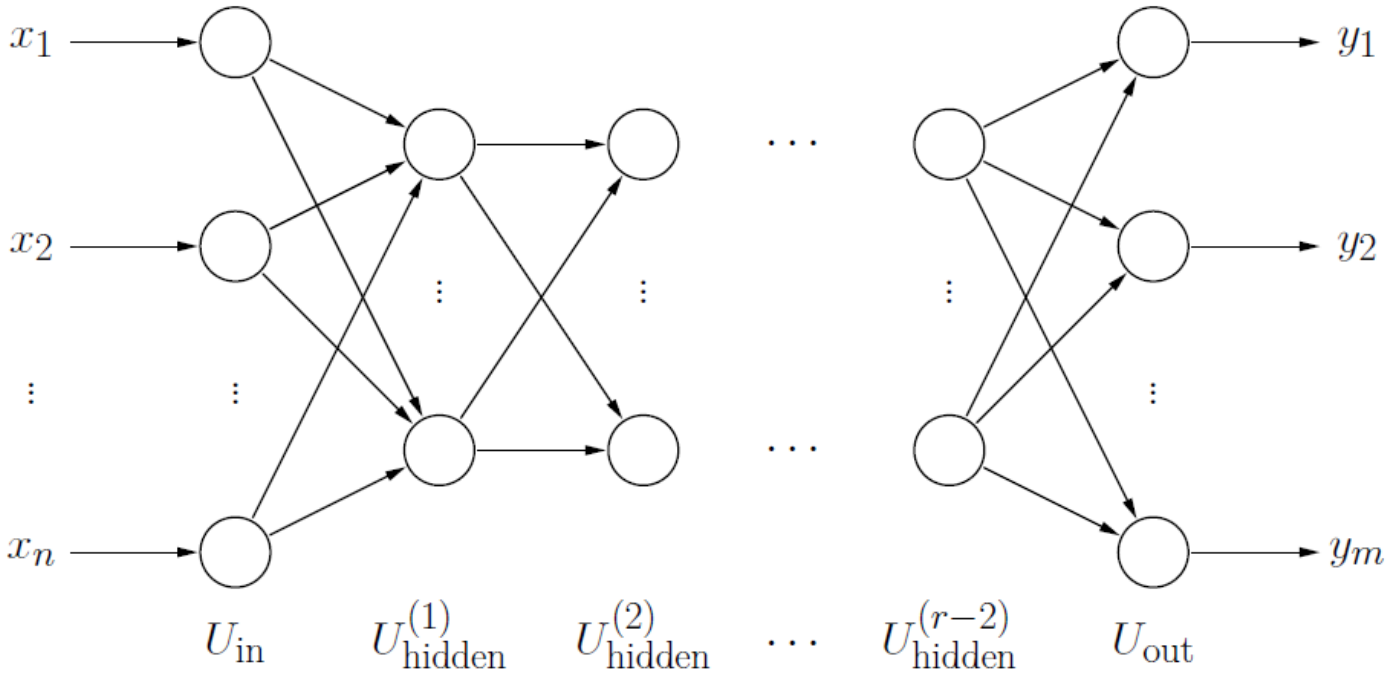
(ii) $U_{\text{hidden}} = U_{\text{hidden}}^{(1)} \cup \cdots \cup U_{\text{hidden}}^{(r-2)}$,
$\forall 1 \leq i < j \leq r-2: \quad U_{\text{hidden}}^{(i)} \cap U_{\text{hidden}}^{(j)} = \emptyset$,

(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.

Mehrschichtige Perzeptren
Multilayer Perceptrons (MLPs)

Allgemeine Struktur eines mehrschichtigen Perzeptrons



Mehrschichtige Perzeptren
Multilayer Perceptrons (MLPs)

- 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

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

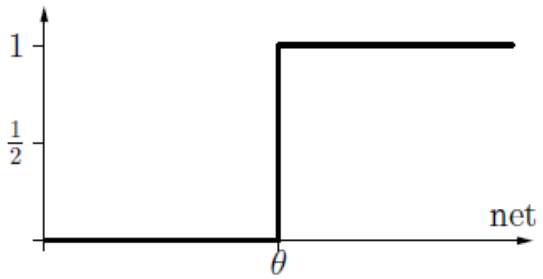
- Die Aktivierungsfunktion jedes Ausgabeneurons ist ebenfalls entweder eine Sigmoide oder eine **lineare Funktion**, d.h.

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

Sigmoide als Aktivierungsfunktionen

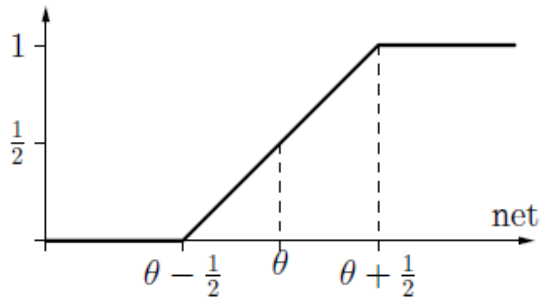
Stufenfunktion:

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



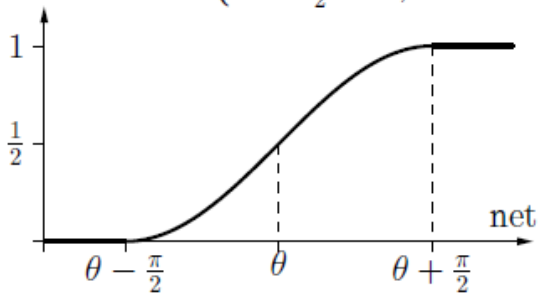
Semilineare Funktion:

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



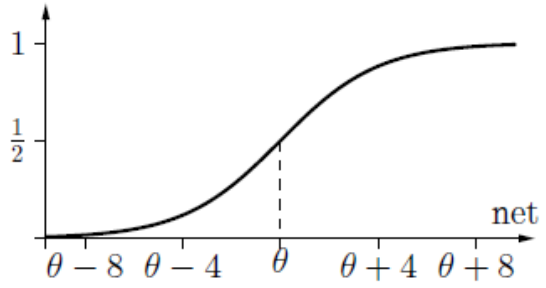
Sinus bis Sättigung:

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



Logistische Funktion:

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



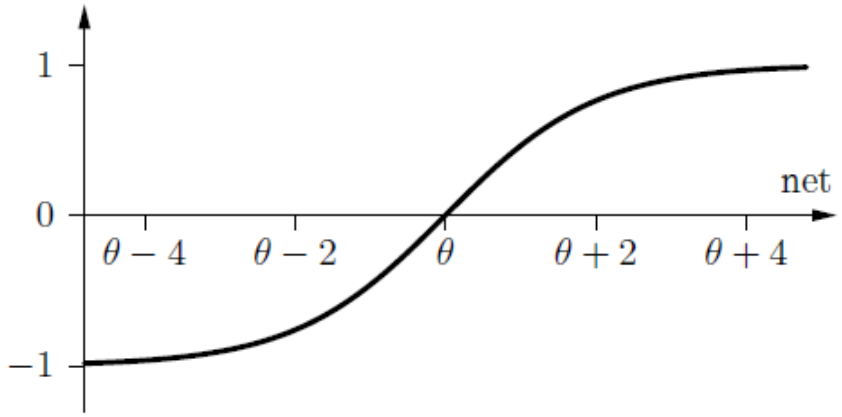
Sigmoide als Aktivierungsfunktionen

- 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.

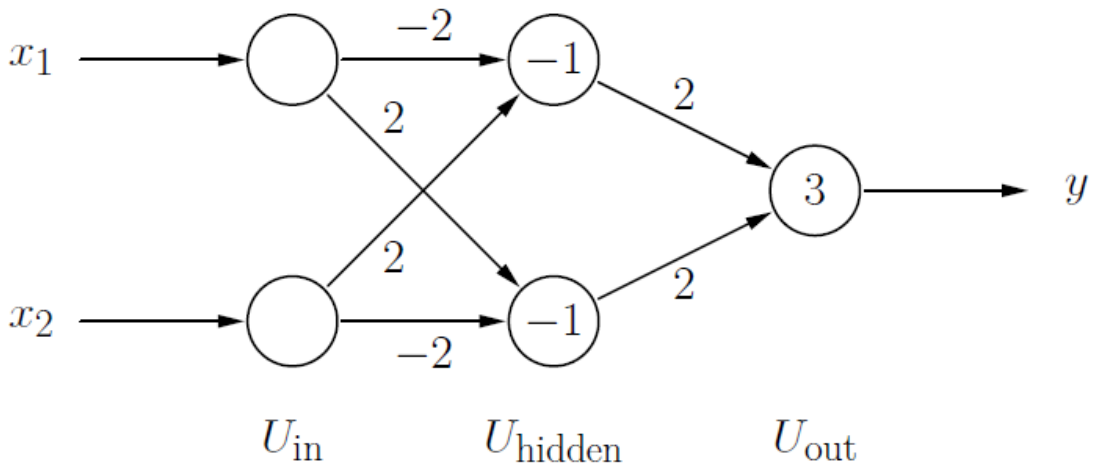
Vorteil: Die Berechnung der Netzwerkeingabe kann geschrieben werden als

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

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

Mehrschichtige Perzeptren: Biimplication

Lösen des Biimplikationsproblems mit einem MLP.



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

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

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

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

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

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

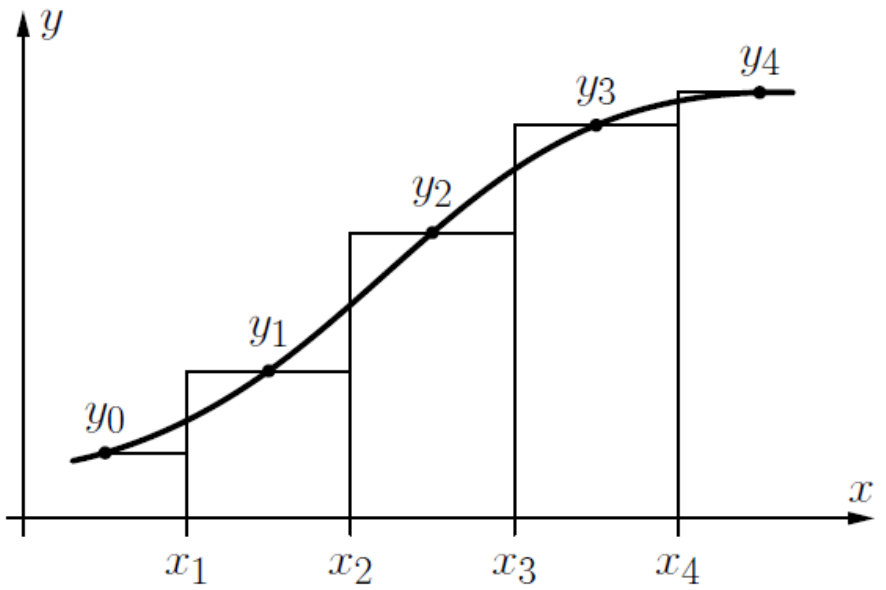
wobei $\mathbf{A}_{13} = \mathbf{A}_{23} \cdot \mathbf{A}_{12}$ und $\boldsymbol{b}_{13} = \mathbf{A}_{23} \cdot \boldsymbol{b}_{12} + \boldsymbol{b}_{23}$.

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

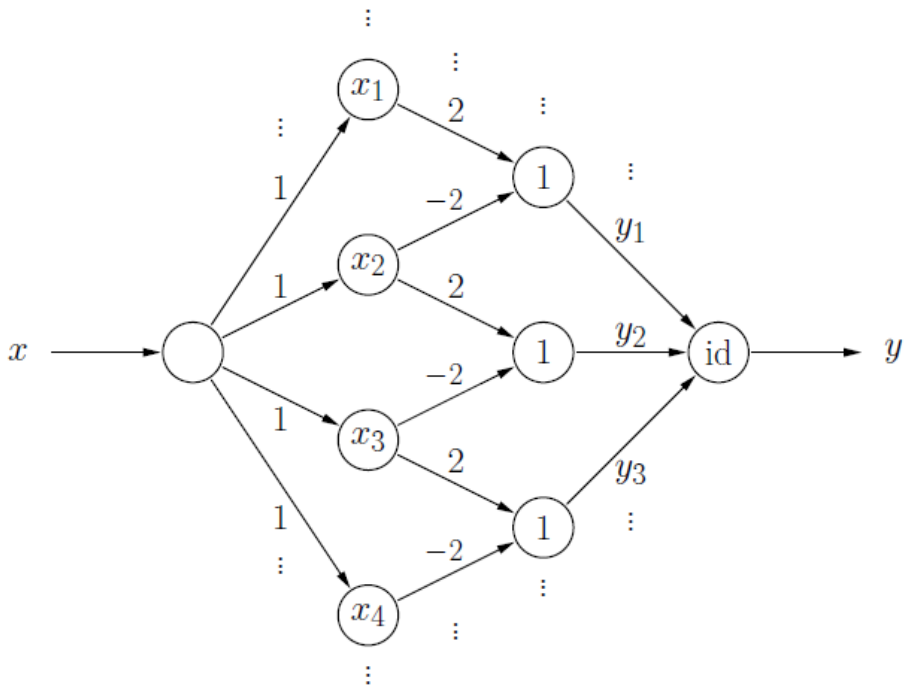
Mehrschichtige Perzeptren: Funktionsapproximation

Idee der Funktionsapproximation

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



Mehrschichtige Perzeptren: Funktionsapproximation

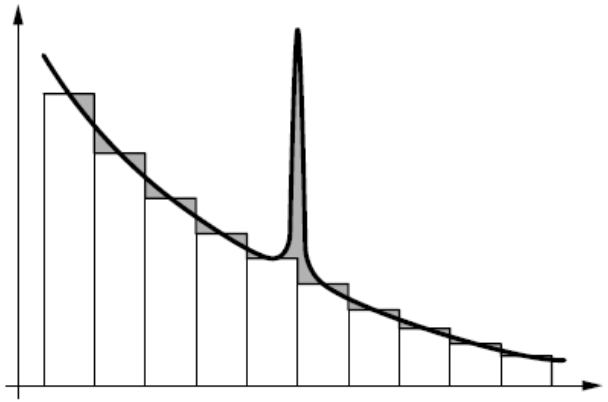


Ein neuronales Netz, das die Treppenfunktion von der vorherigen Folie berechnet. Es ist immer nur eine Stufe passend zum Eingabewert aktiv und die Stufenhöhe wird ausgegeben.

Mehrschichtige Perzeptren: Funktionsapproximation

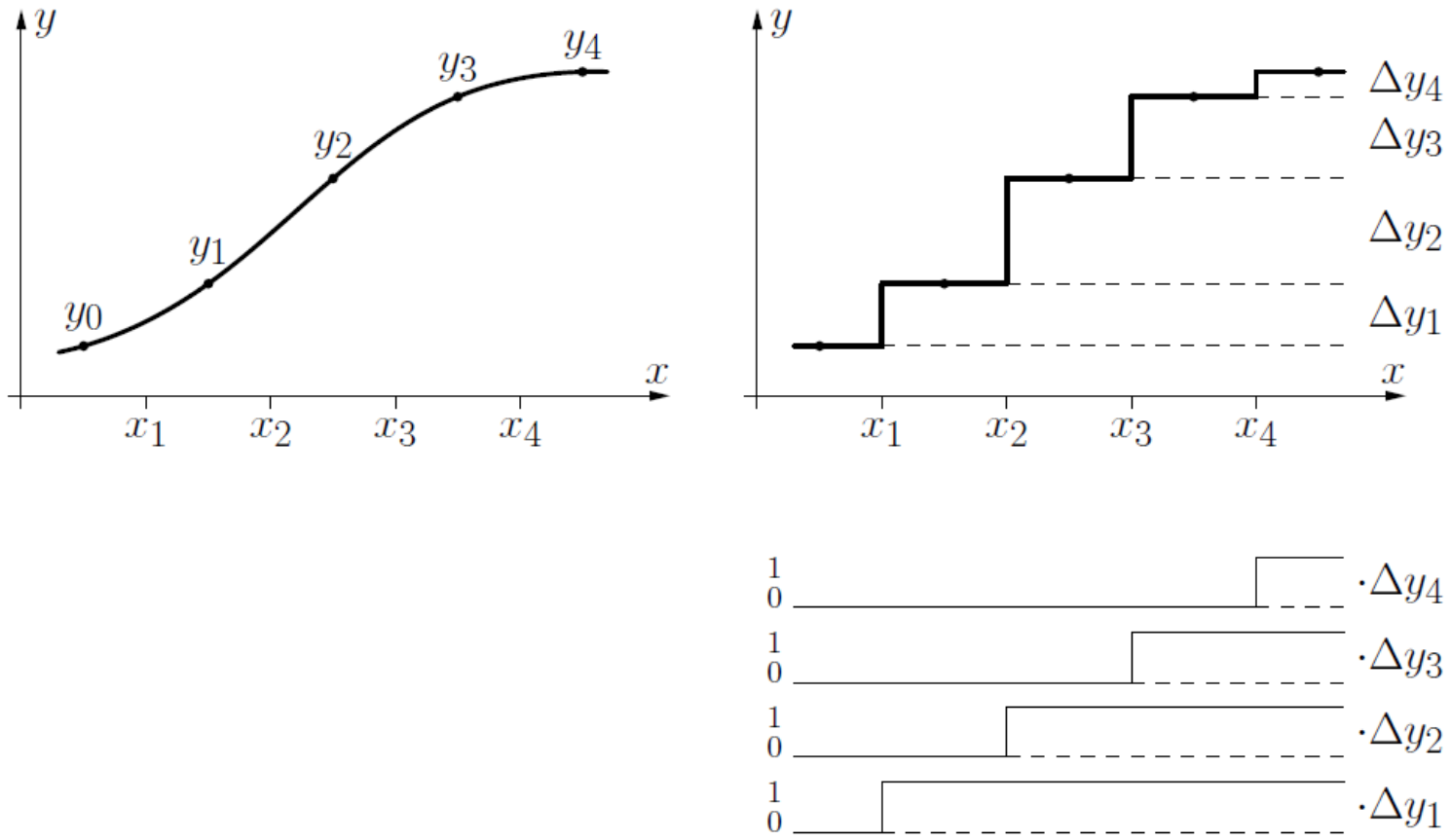
Theorem: Jede Riemann-integrierbare Funktion kann mit beliebiger Genauigkeit durch ein vierschichtiges MLP berechnet werden.

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

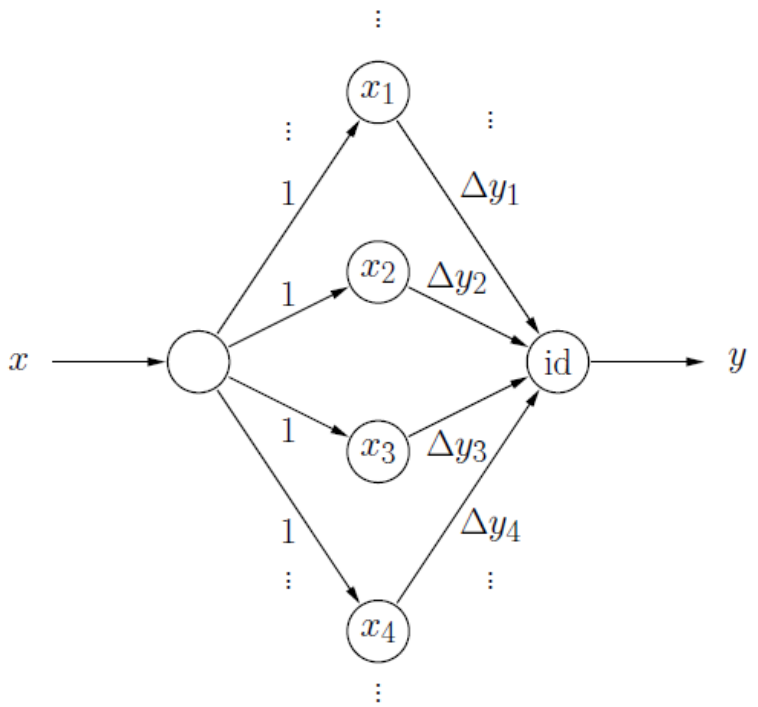


- Weitere mathematische Untersuchungen zeigen, dass sogar gilt: Mit einem dreischichtigen Perzeptron kann jede stetige Funktion mit beliebiger Genauigkeit angenähert werden (Fehlerbestimmung: maximale Differenz der Funktionswerte).

Mehrschichtige Perzeptren: Funktionsapproximation

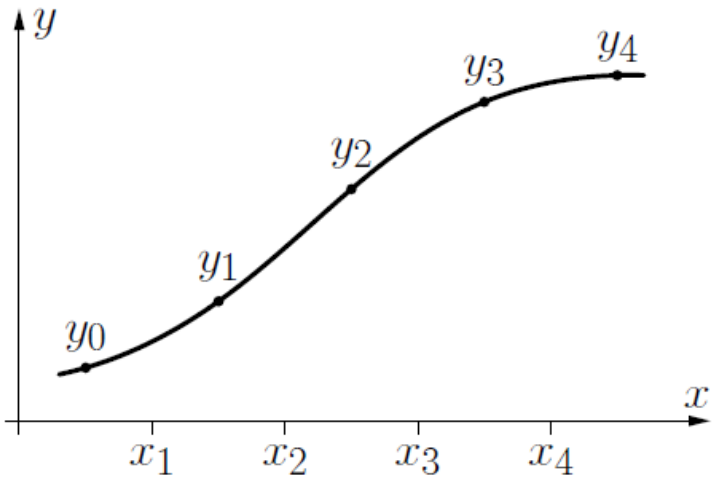


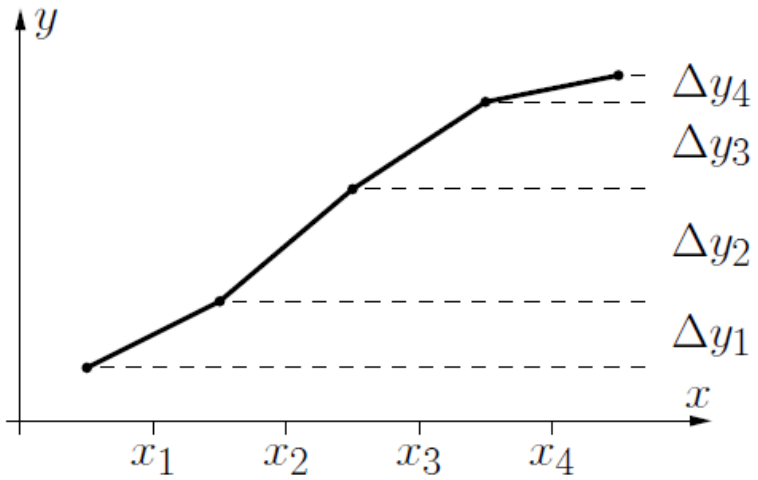
Mehrschichtige Perzeptren: Funktionsapproximation

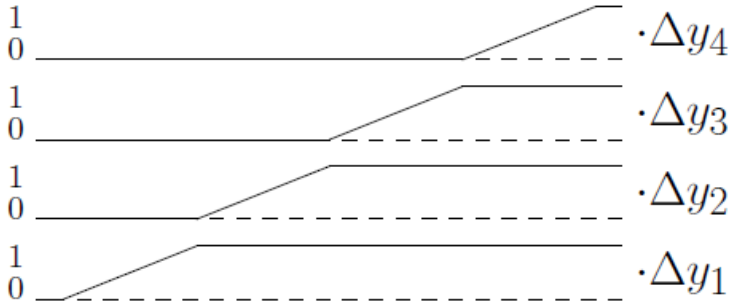


Ein neuronales Netz, das die Treppenfunktion von der vorherigen Folie als gewichtete Summe von Sprungfunktionen berechnet.

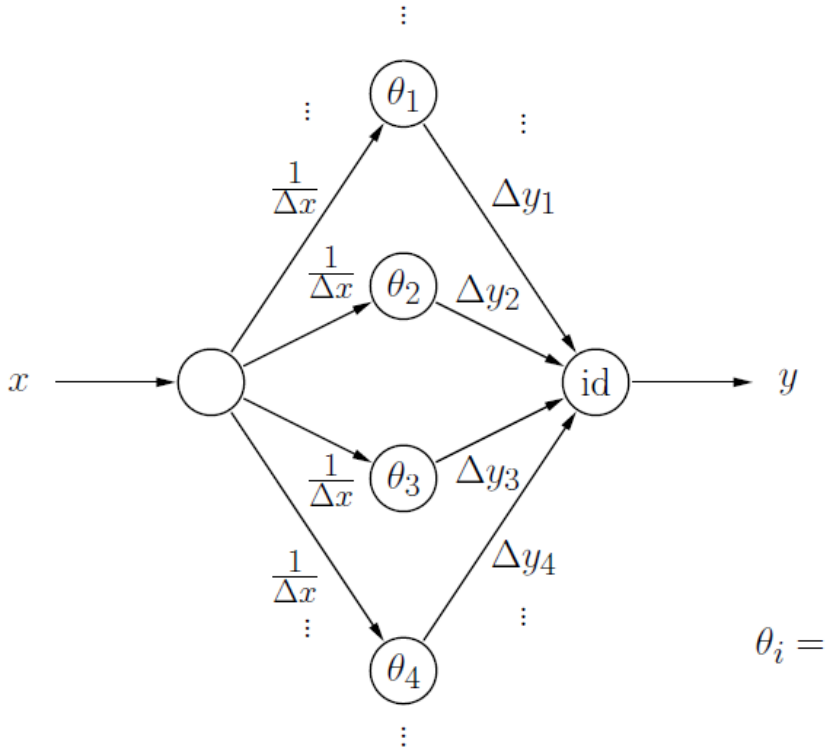
Mehrschichtige Perzeptren: Funktionsapproximation







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: Lineare 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.

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

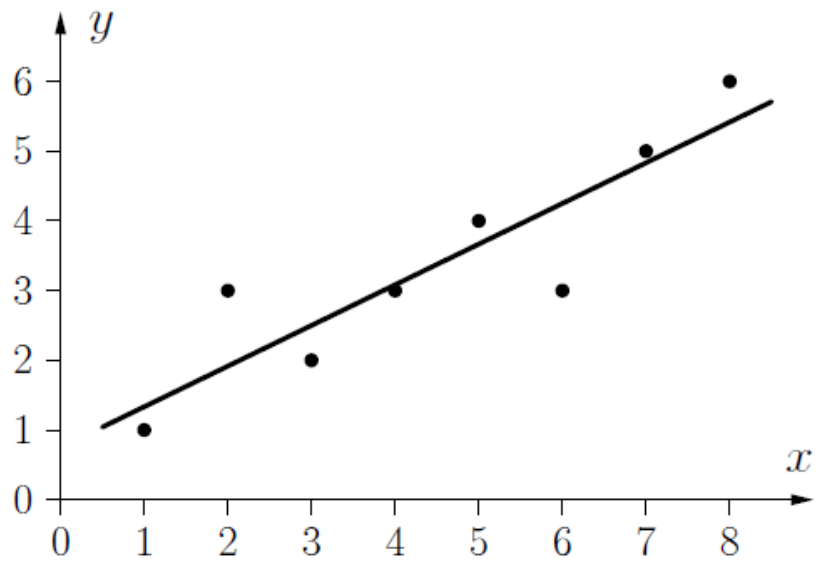
$$\left(\sum_{i=1}^{n} x_i\right) a + \left(\sum_{i=1}^{n} x_i^2\right) b = \sum_{i=1}^{n} x_i y_i.$$

- 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

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



Mathematischer Hintergrund: Polynomiale Regression

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.

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

Mathematischer Hintergrund: Polynomiale Regression

System von Normalgleichungen für Polynome

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

- $m+1$ 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.

Mathematischer Hintergrund: Multilineare Regression

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.

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

Mathematischer Hintergrund: Multilineare Regression

System von Normalgleichungen für mehrere Argumente

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

- 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.

Multilineare Regression

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:

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

Multilineare Regression

- $\nabla_{\boldsymbol{a}}\ F(\boldsymbol{a})$ 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:

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

Multilineare Regression

Einige Regeln für Vektor-/Matrixberechnung und Ableitungen:

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

Ableitung der zu minimierenden Funktion:

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

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 $\mathbf{X}$.

Mathematischer Hintergrund: Logistische Regression

Generalisierung auf nicht-polynomiale Funktionen

Einfaches Beispiel: $y = ax^b$

Idee: Finde Transformation zum linearen/polynomiellen Fall.

Transformation z.B.: $\ln y = \ln a + b \cdot \ln x.$

Spezialfall: **logistische Funktion**

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

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

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

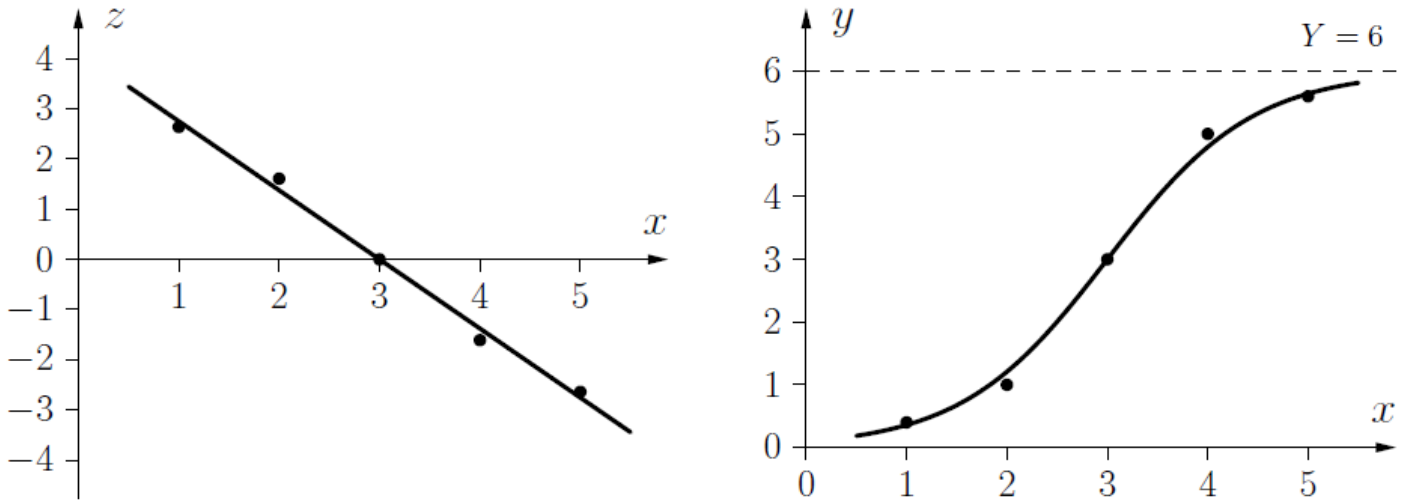
Die transformierten Datenpunkte sind

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: Gradientenabstieg

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

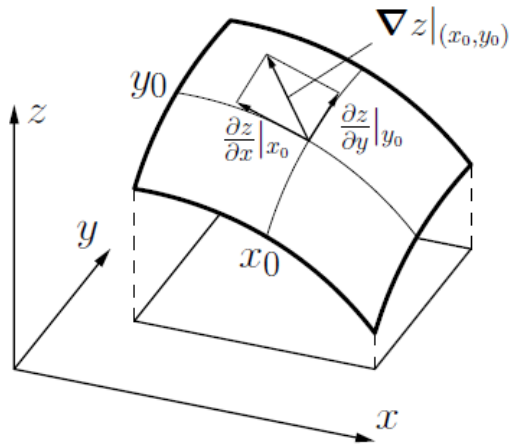


Illustration des Gradienten einer reellwertigen Funktion $z = f(x, y)$ am Punkt (x_0, y_0). Dabei ist $\nabla z|_{(x_0,y_0)} = \left(\frac{\partial z}{\partial x}|_{x_0}, \frac{\partial z}{\partial y}|_{y_0}\right)$.

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 $l = (\boldsymbol{\imath}^{(l)}, \boldsymbol{o}^{(l)})$:

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

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

Gradientenabstieg: Formaler Ansatz

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.

Fehlerrückpropagation engl: error backpropagation

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

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

Wir wenden die Kettenregel an und erhalten

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

Summentausch ergibt

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

Fehlerrückpropagation

Betrachte die Netzeingabe

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

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

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

Das Ergebnis ist die rekursive Gleichung (Fehlerrückpropagation)

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

Fehlerrückpropagation

Die sich ergebende Formel für die Gewichtsänderung ist

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

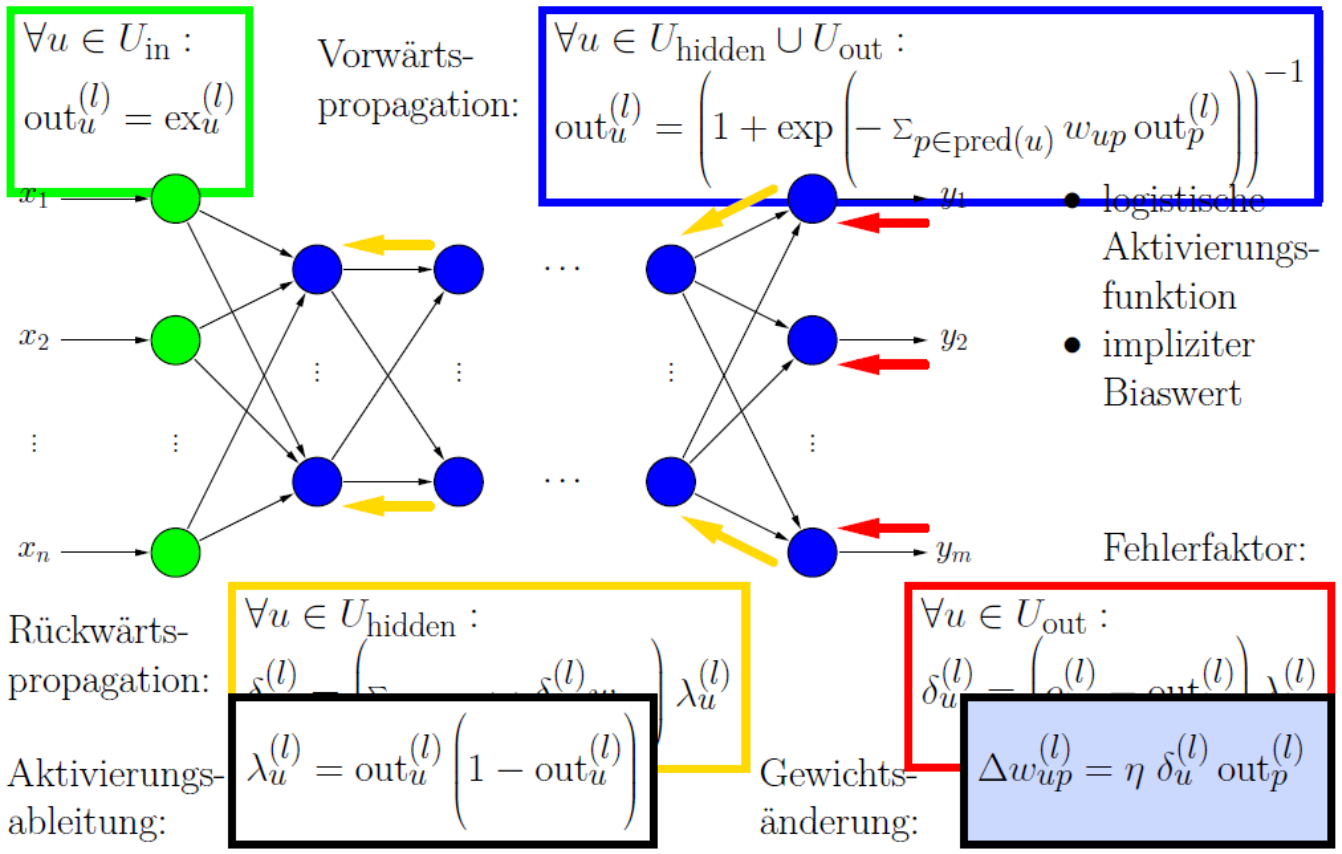
Betrachte erneut den Spezialfall mit

- Ausgabefunktion: Identität,
- Aktivierungsfunktion: logistisch.

Die sich ergebende Formel für die Gewichtsänderung ist damit

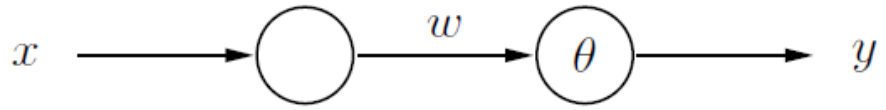
$$\Delta \boldsymbol{w}_u^{(l)} = \eta \left(\sum_{s \in \mathrm{succ}(u)} \delta_s^{(l)} w_{su} \right) \mathrm{out}_u^{(l)} \left(1 - \mathrm{out}_u^{(l)}\right) \mathbf{in}_u^{(l)}.$$

Fehlerrückpropagation: Vorgehensweise

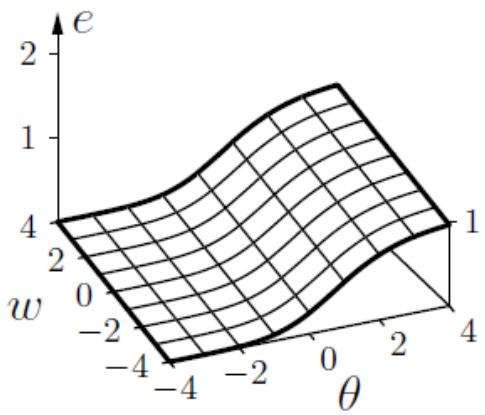


Gradientenabstieg: Beispiele

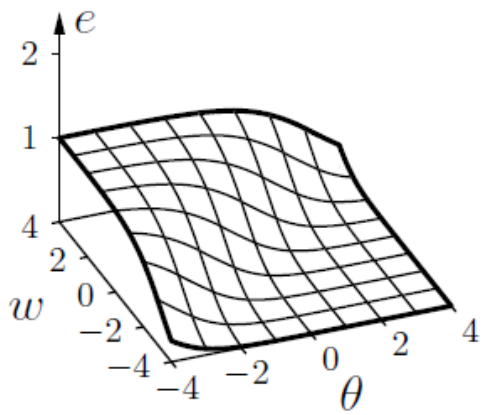
Gradientenabstieg für die Negation $\neg x$



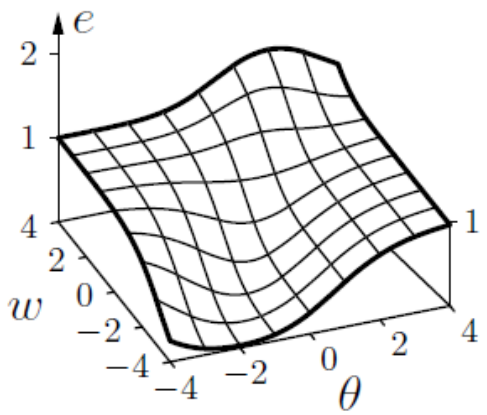
x	y
0	1
1	0



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

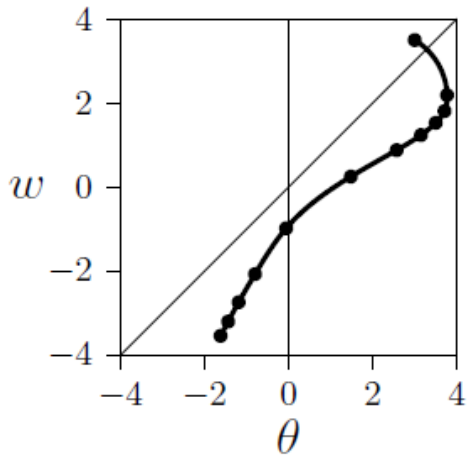
Online-Training

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

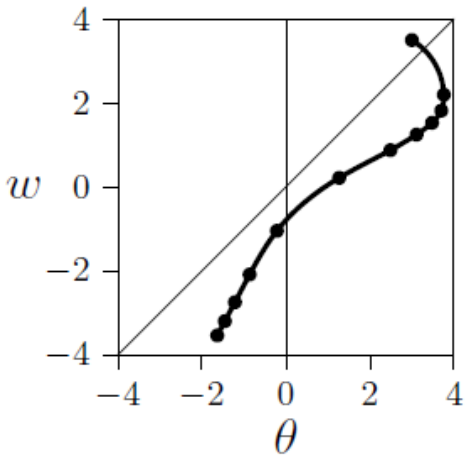
Batch-Training

Gradientenabstieg: Beispiele

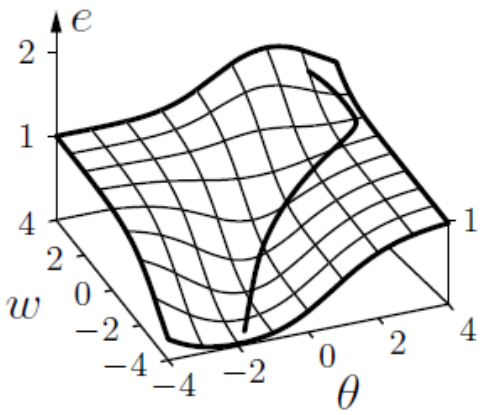
Visualisierung des Gradientenabstiegs für die Negation $\neg x$



Online-Training



Batch-Training



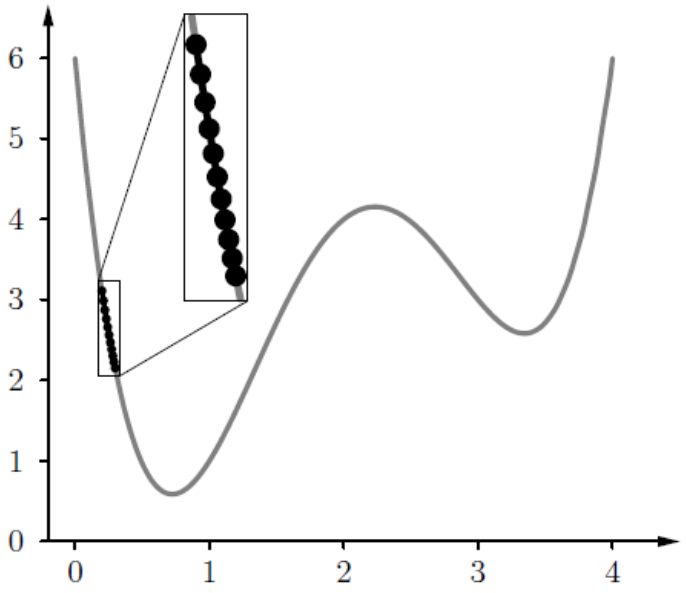
Batch-Training

- 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		

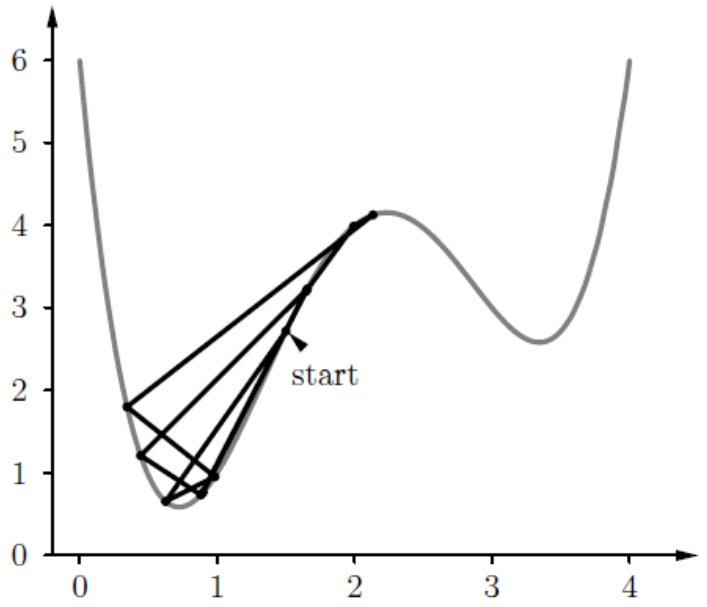


Gradientenabstieg mit Startwert 0.2 und Lernrate 0.001.

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		

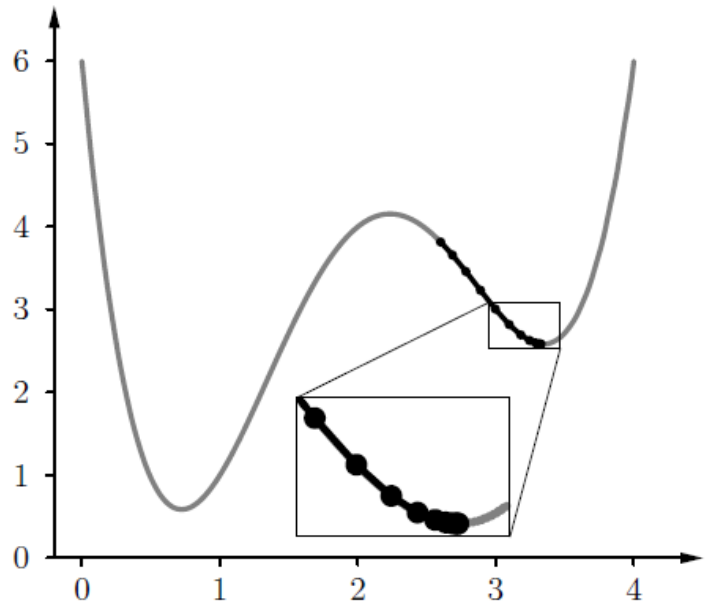


Gradientenabstieg mit Startwert 1.5 und Lernrate 0.25.

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

Gradientenabstieg: Varianten

Selbstadaptive Fehlerrückpropagation:

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

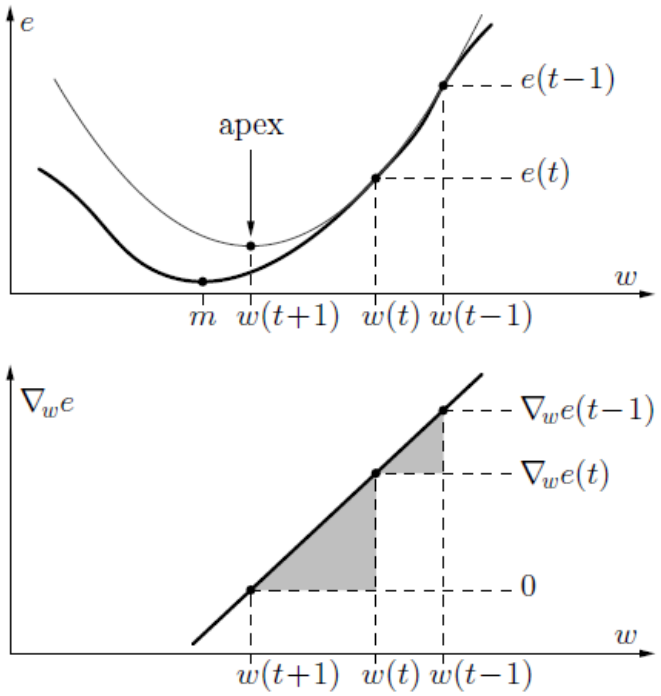
Elastische Fehlerrückpropagation:

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

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

Gradientenabstieg: Varianten

Quickpropagation



Die Gewichts-Updateregel kann aus den Dreiecken abgeleitet werden:

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

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:

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

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:

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

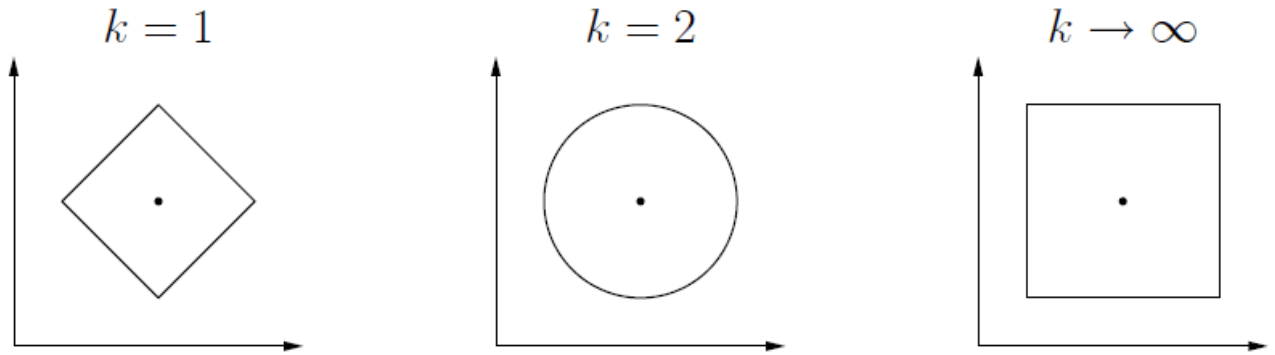
Abstandsfunktionen

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

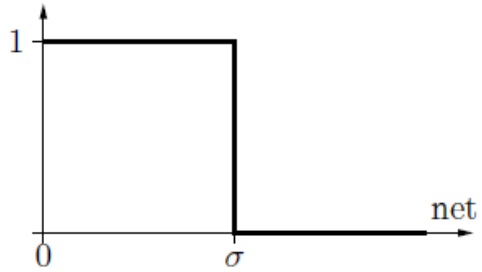
$$f_{\text{act}}^{(u)}(\text{net}_u, \theta_u) = \text{net}_u - \theta_u.$$

(Die lineare Aktivierungsfunktion ist wichtig für die Initialisierung.)

Radiale Aktivierungsfunktionen

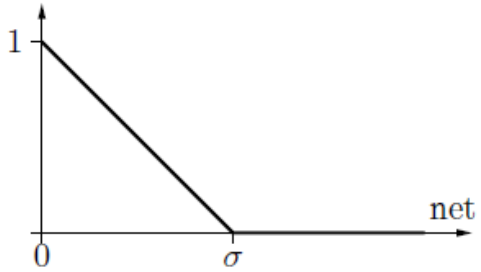
Rechteckfunktion:

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



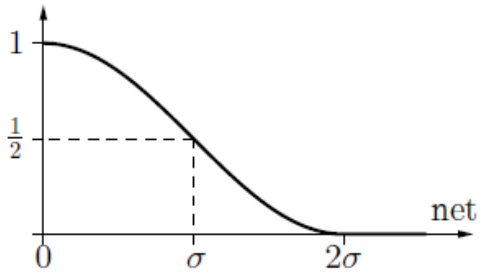
Dreiecksfunktion:

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



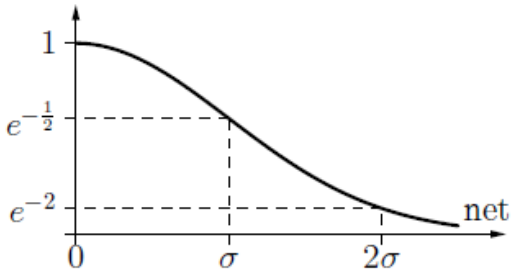
Kosinus bis Null:

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



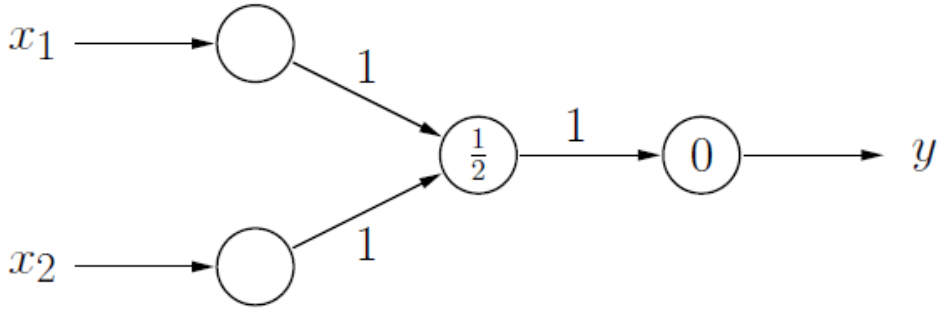
Gaußsche Funktion:

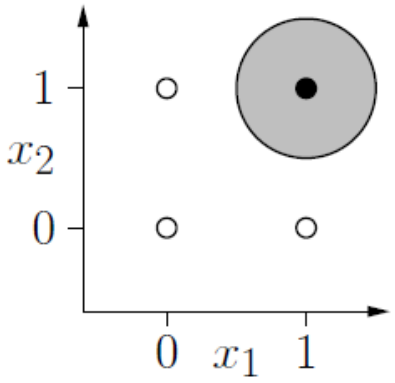
$$f_{\text{act}}(\text{net}, \sigma) = e^{-\frac{\text{net}^2}{2\sigma^2}}$$



Radiale-Basisfunktionen-Netze: Beispiele

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

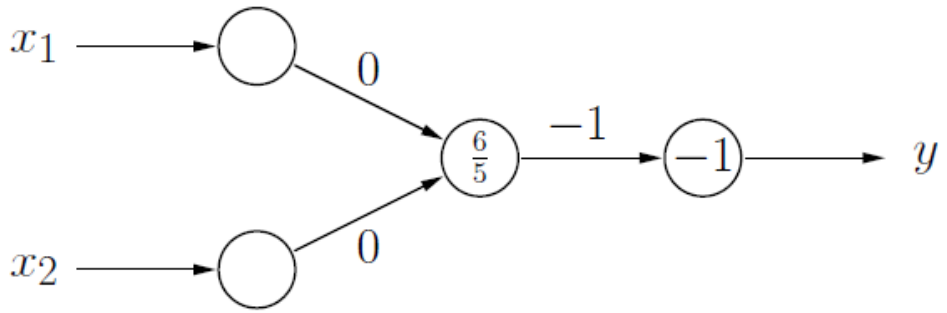


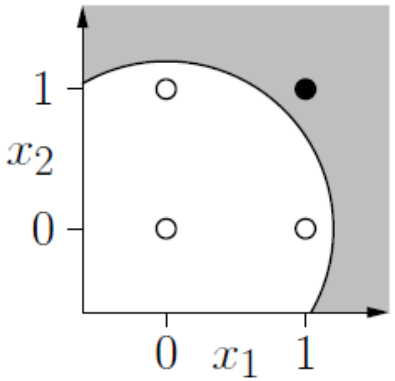


- (1,1) ist Zentrum
- Referenzradius ist $\frac{1}{2}$
- Euklidischer Abstand
- Rechteckfunktion als Aktivierung
- Biaswert 0 im Ausgabeneuron

Radiale-Basisfunktionen-Netze: Beispiele

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





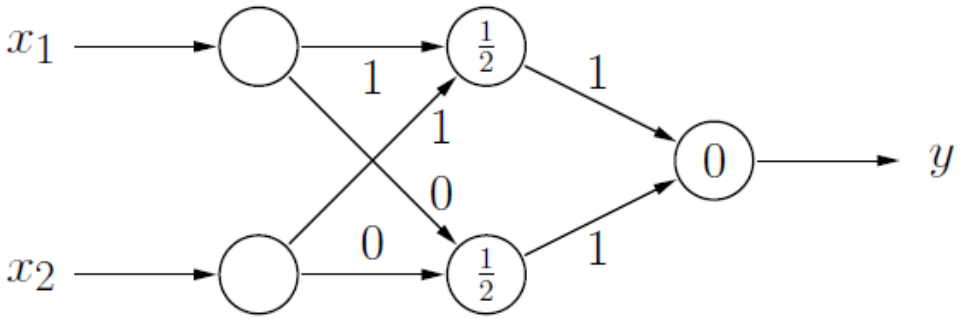
- (0,0) ist Zentrum
- Referenzradius ist $\frac{6}{5}$
- Euklidischer Abstand
- Rechteckfunktion als Aktivierung
- Biaswert -1 im Ausgabeneuron

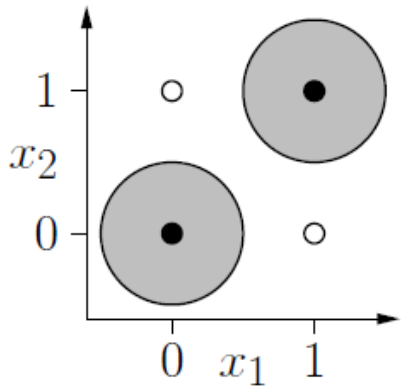
Radiale-Basisfunktionen-Netze: Beispiele

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

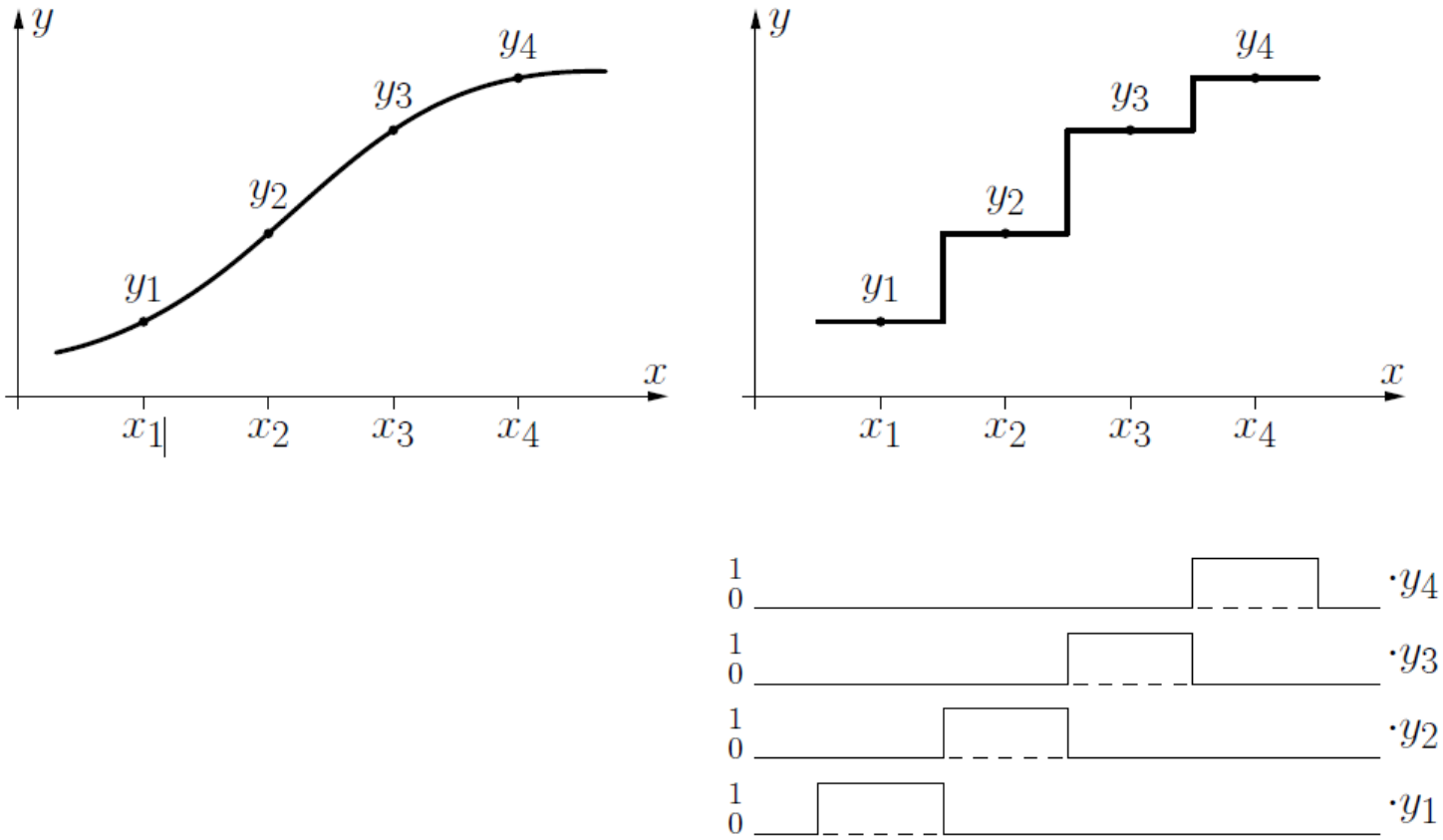
Idee: logische Zerlegung

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



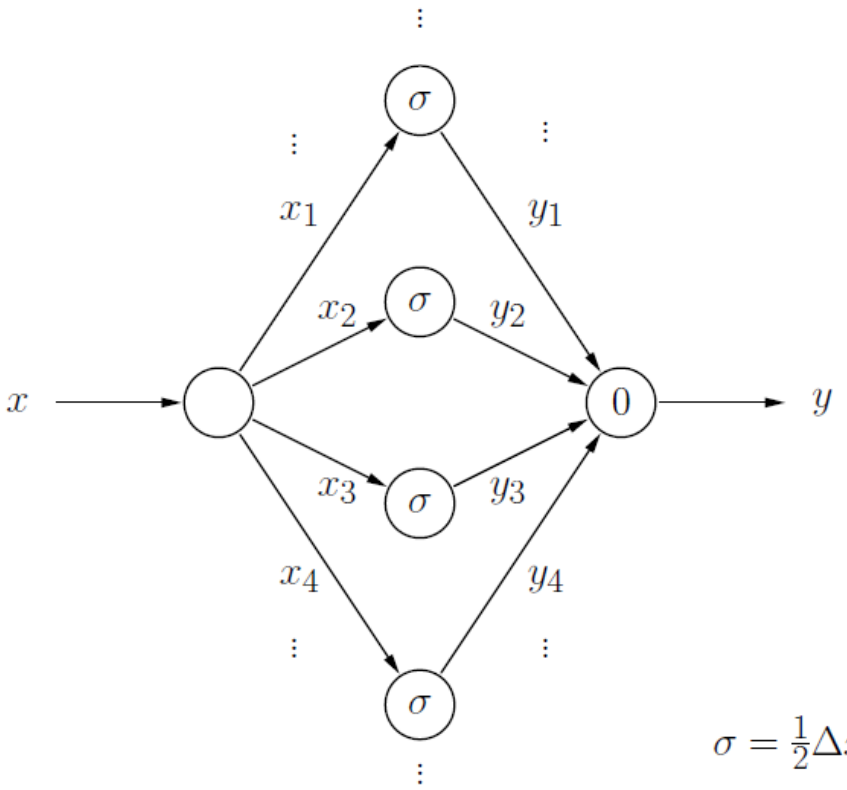


Radiale-Basisfunktionen-Netze: Funktionsapproximation



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

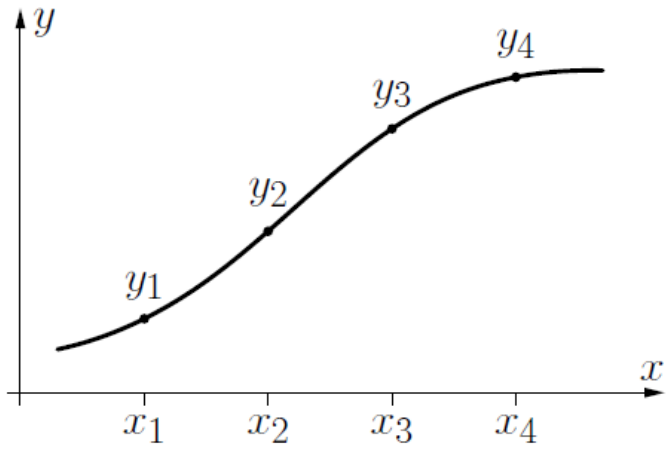
Radiale-Basisfunktionen-Netze: Funktionsapproximation

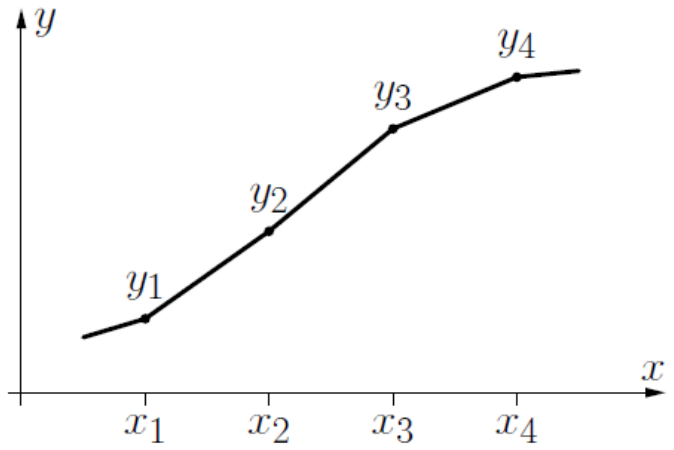


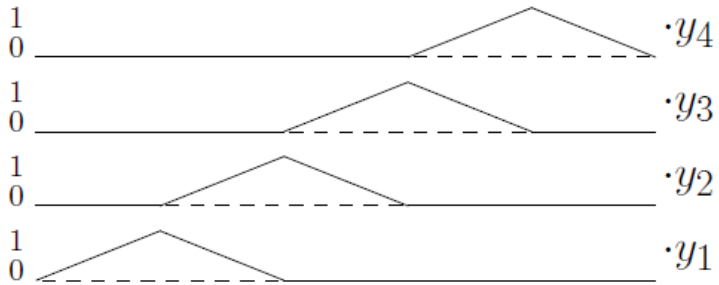
$$\sigma = \frac{1}{2}\Delta x = \frac{1}{2}(x_{i+1} - x_i)$$

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

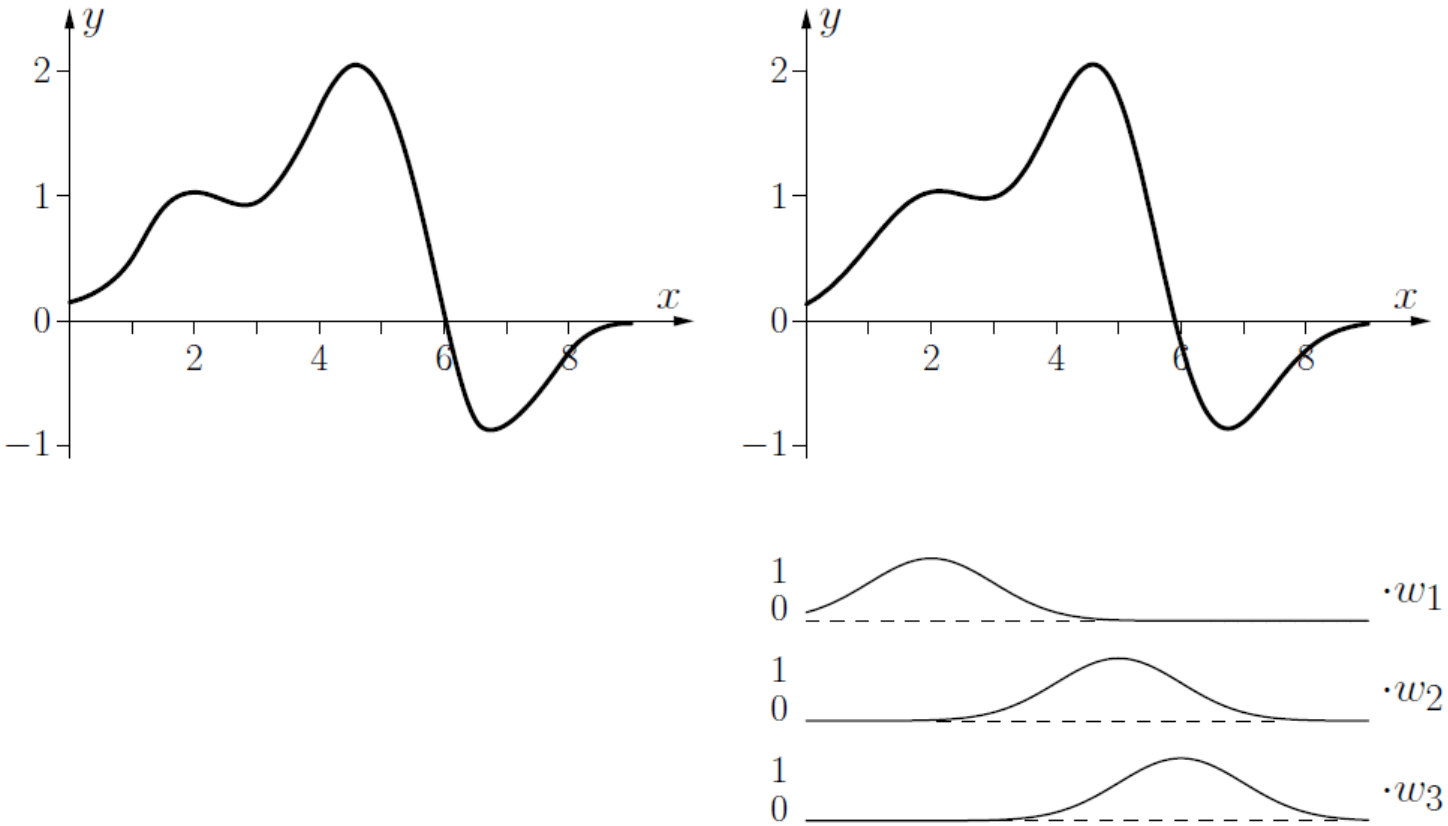






Darstellung einer stückweise linearen Funktion durch eine gewichtete Summe von Dreiecksfunktionen mit Zentren x_i.

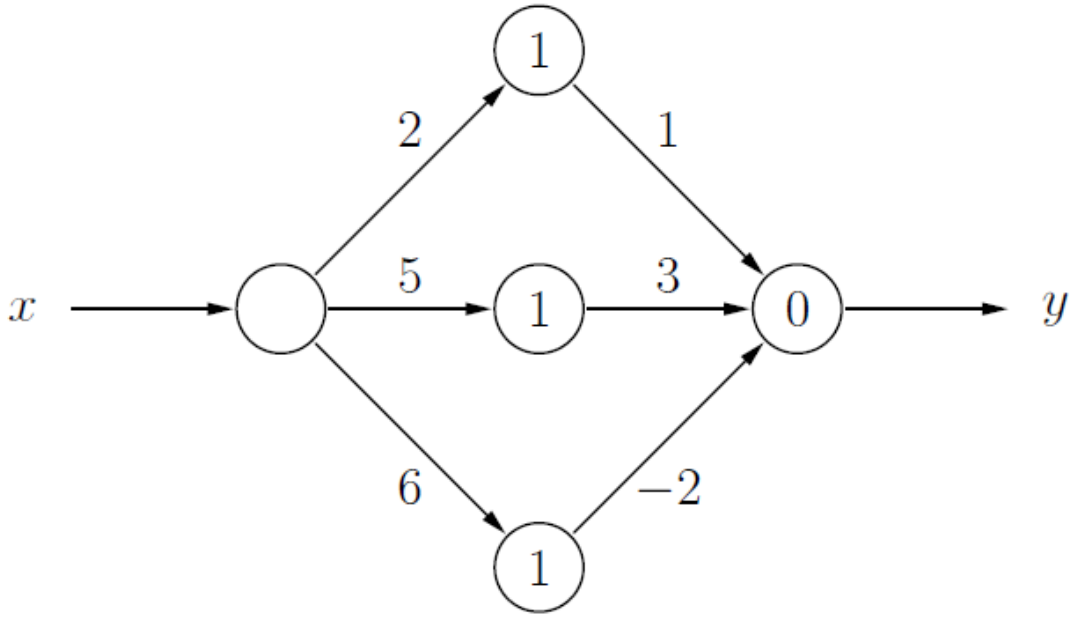
Radiale-Basisfunktionen-Netze: Funktionsapproximation



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

Radiale-Basisfunktionen-Netze: Funktionsapproximation

RBF-Netz für eine Summe dreier Gaußfunktionen



Training von RBF-Netzen

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

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

Radiale-Basisfunktionen-Netze: Initialisierung

Initialisieren der Verbindungen von den versteckten zu den Ausgabeneuronen

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

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

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

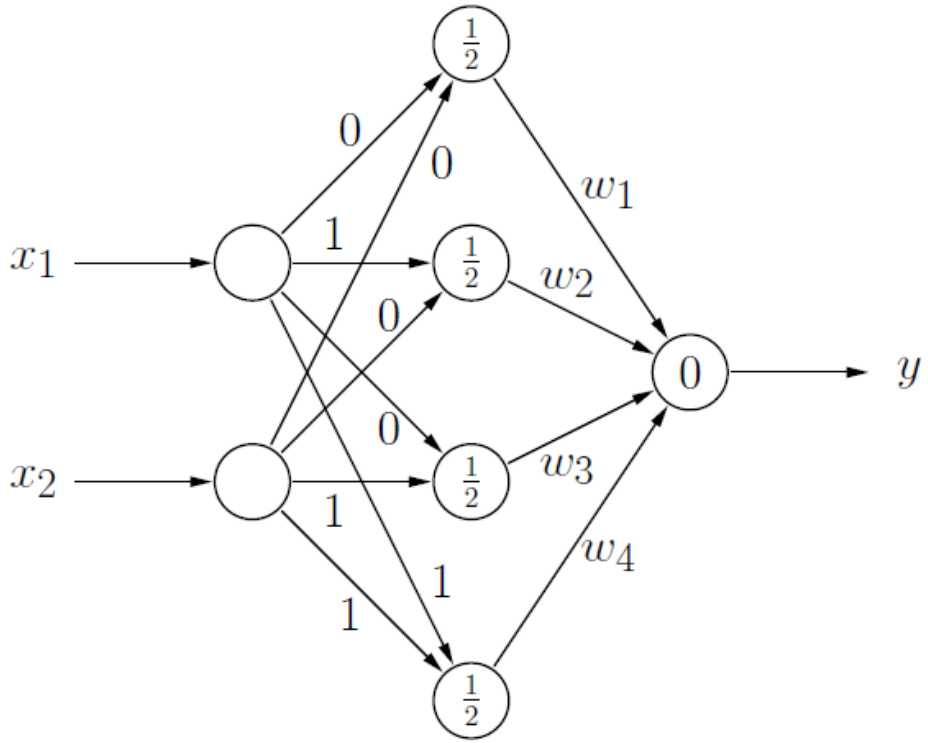
Ergebnis: Lineares Gleichungssystem, das durch Invertieren der Matrix $\mathbf{A}$ gelöst werden kann:

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

RBF-Netz-Initialisierung: Beispiel

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



RBF-Netz-Initialisierung: Beispiel

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

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

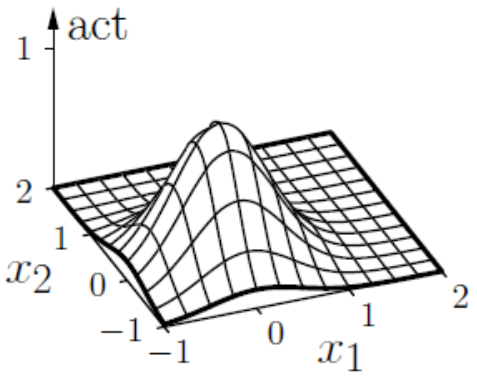
wobei

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

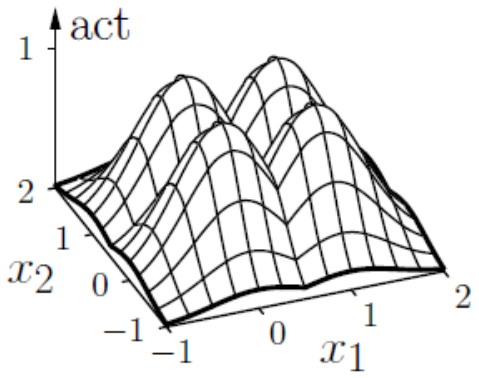
$$\boldsymbol{w}_u = \mathbf{A}^{-1} \cdot \boldsymbol{o}_u = \frac{1}{D} \begin{pmatrix} a+c \\ 2b \\ 2b \\ a+c \end{pmatrix} \approx \begin{pmatrix} 1.0567 \\ -0.2809 \\ -0.2809 \\ 1.0567 \end{pmatrix}$$

RBF-Netz-Initialisierung: Beispiel

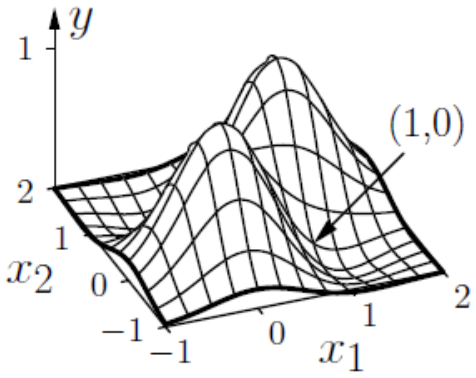
Einfaches RBF-Netz für die Biimplikation $x_1 \leftrightarrow x_2$



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.

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

Berechne (Moore–Penrose)-Pseudoinverse:

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

Die Gewichte können dann durch

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

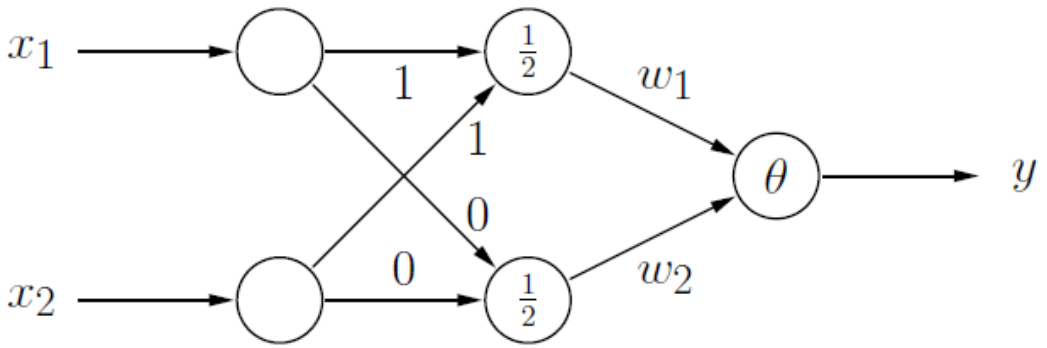
berechnet werden.

RBF-Netz-Initialisierung: Beispiel

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

Wähle zwei Trainingsbeispiele aus:

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



RBF-Netz-Initialisierung: Beispiel

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

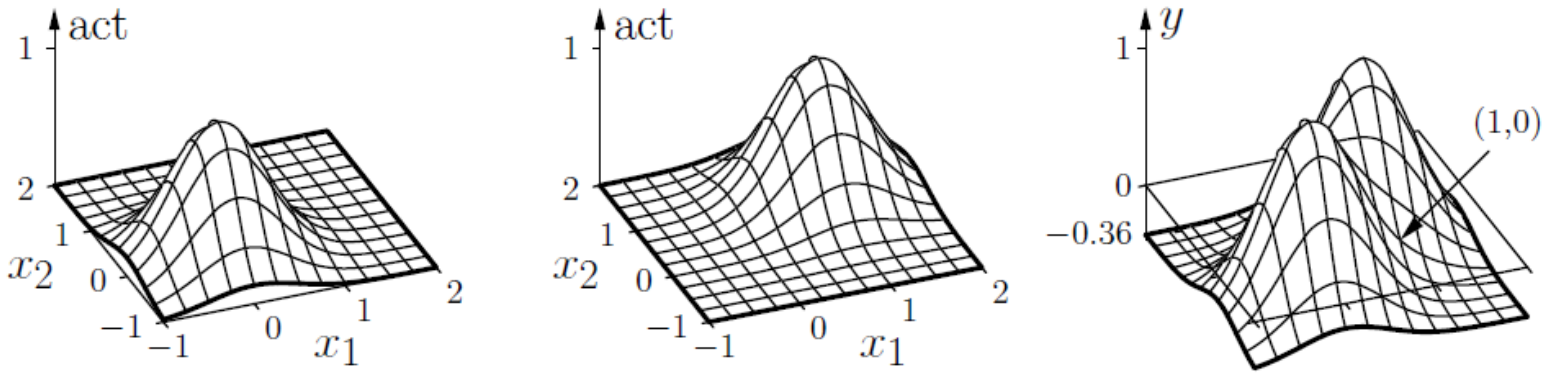
$$\begin{aligned} a &\approx -0.1810, & b &\approx 0.6810, \\ c &\approx 1.1781, & d &\approx -0.6688, & e &\approx 0.1594. \end{aligned}$$

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.

Radiale-Basisfunktionen-Netze: Initialisierung

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**

Radiale-Basisfunktionen-Netze: Training

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

Gewichte von den versteckten zu den Ausgabeneuronen.

Gradient:

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

Gewichtsänderungsregel:

$$\Delta \boldsymbol{w}_u^{(l)} = -\frac{\eta 3}{2} \nabla_{\boldsymbol{w}_u} e_u^{(l)} = \eta 3 \left(o_u^{(l)} - \mathrm{out}_u^{(l)}\right) \mathbf{in}_u^{(l)}$$

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

Radiale-Basisfunktionen-Netze: Training

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:

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

Radiale-Basisfunktionen-Netze: Training

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**

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

Radiale-Basisfunktionen-Netze: Training

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**

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

Radiale-Basisfunktionen-Netze: Verallgemeinerung

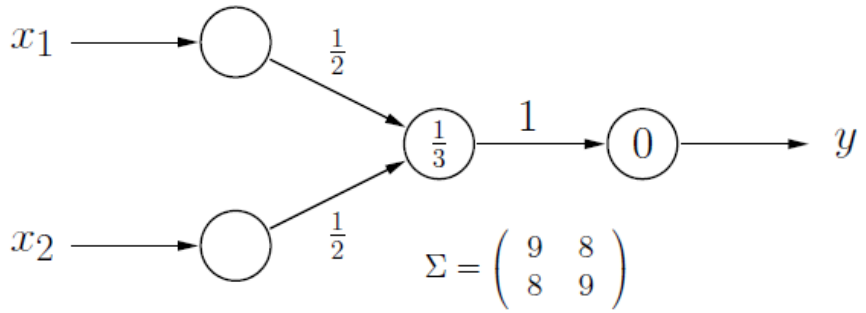
Verallgemeinerung der Abstandsfunktion

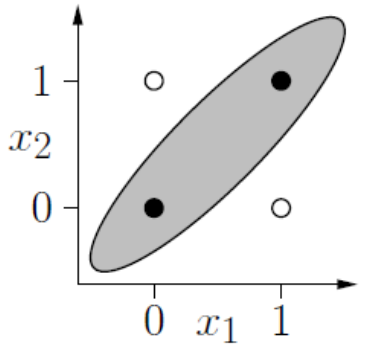
Idee: Benutze anisotrope (richtungsabhängige) Abstandsfunktion.

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

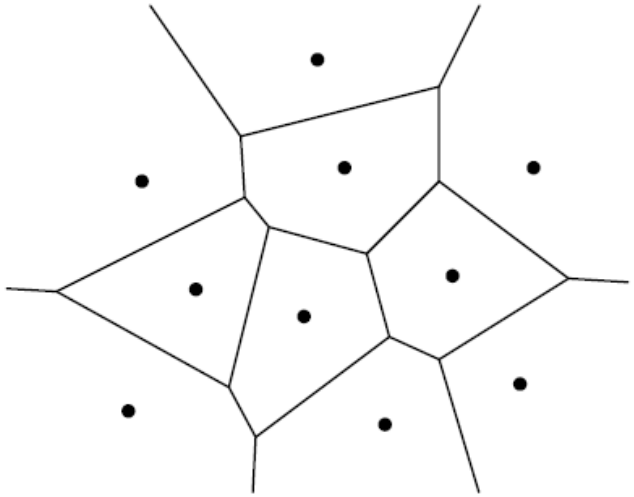
- kontinuierlich laufende Prozesse, die schnelle Anpassung erfordern
- Approximierung
- Mustererkennung
- Regelungstechnik

Lernende Vektorquantisierung
(engl. Learning Vector Quantization)

- 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

Vektorquantisierung

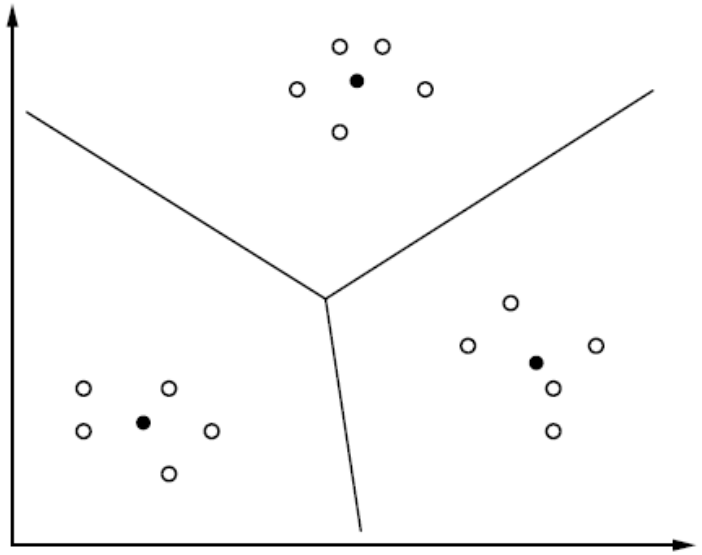
Voronoidiagramm einer Vektorquantisierung



- 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.

Lernende Vektorquantisierung

Finden von Clustern in einer gegebenen Menge von Punkten



- Datenpunkte werden durch leere Kreise dargestellt ($\circ$).
- Clusterzentren werden durch gefüllte Kreise dargestellt ($\bullet$).

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 $\forall \boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z} \in \mathbb{R}^n$:

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

erfüllt.

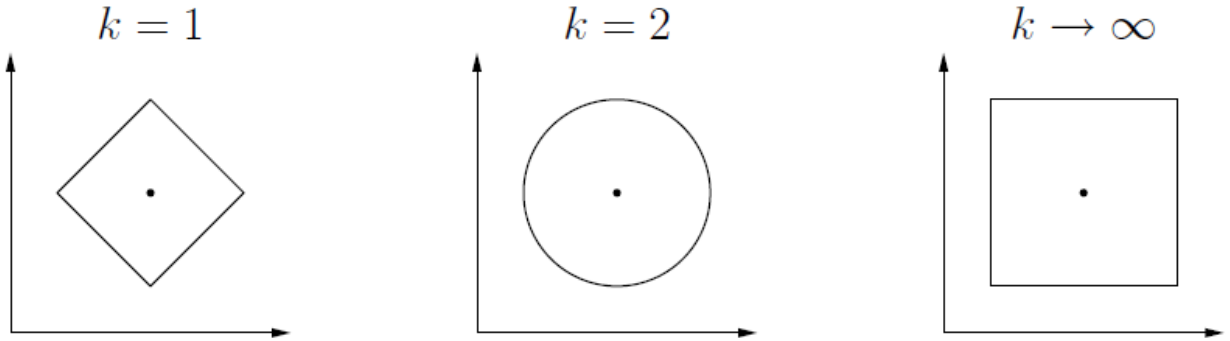
Abstandsfunktionen

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. $d_\infty(\boldsymbol{x}, \boldsymbol{y}) = \max_{i=1}^{n} |x_i - y_i|$.



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

Lernende Vektorquantisierung

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:

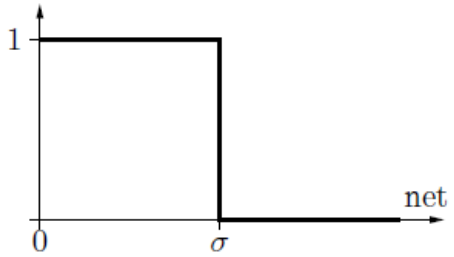
$$f_{\text{out}}^{(u)}(\text{act}_u) = \begin{cases} 1, & \text{falls } \text{act}_u = \max\limits_{v \in U_{\text{out}}} \text{act}_v, \\ 0, & \text{sonst.} \end{cases}$$

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

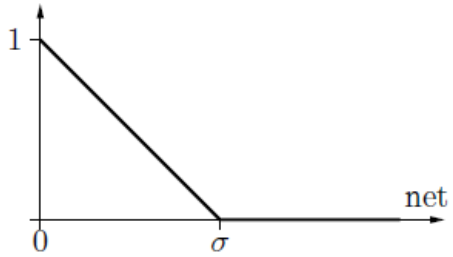
Rechteckfunktion:

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



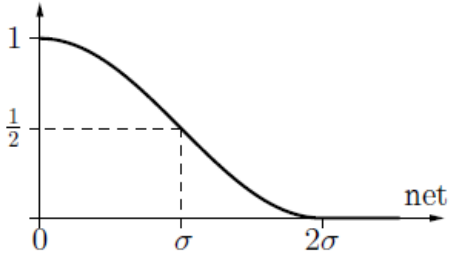
Dreiecksfunktion:

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



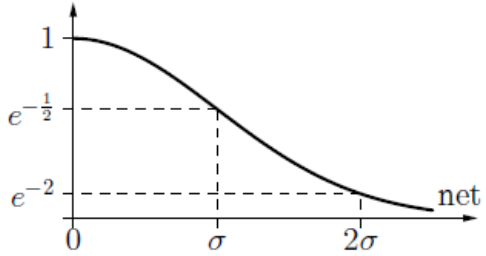
Kosinus bis Null:

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



Gauß-Funktion:

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



Lernende Vektorquantisierung

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)

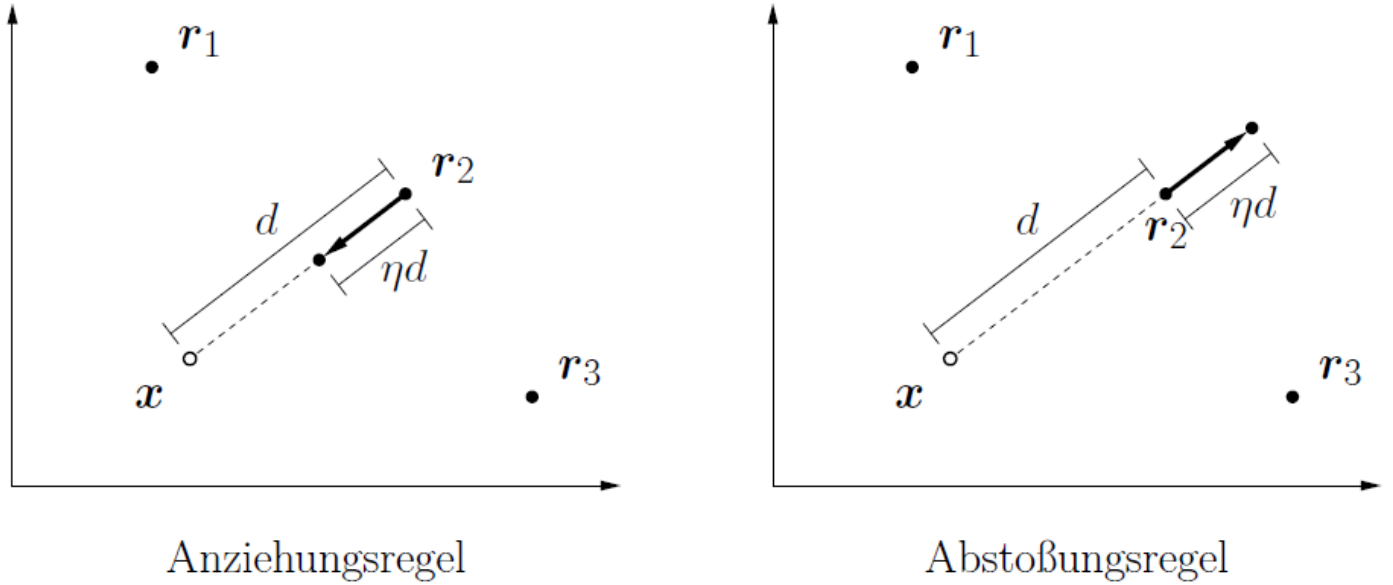
$$\boldsymbol{r}^{(\text{new})} = \boldsymbol{r}^{(\text{old})} + \eta\big(\boldsymbol{x} - \boldsymbol{r}^{(\text{old})}\big),$$

Abstoßungsregel (Datenpunkt und Referenzvektor haben verschiedene Klassen)

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

Lernende Vektorquantisierung

Anpassung der Referenzvektoren



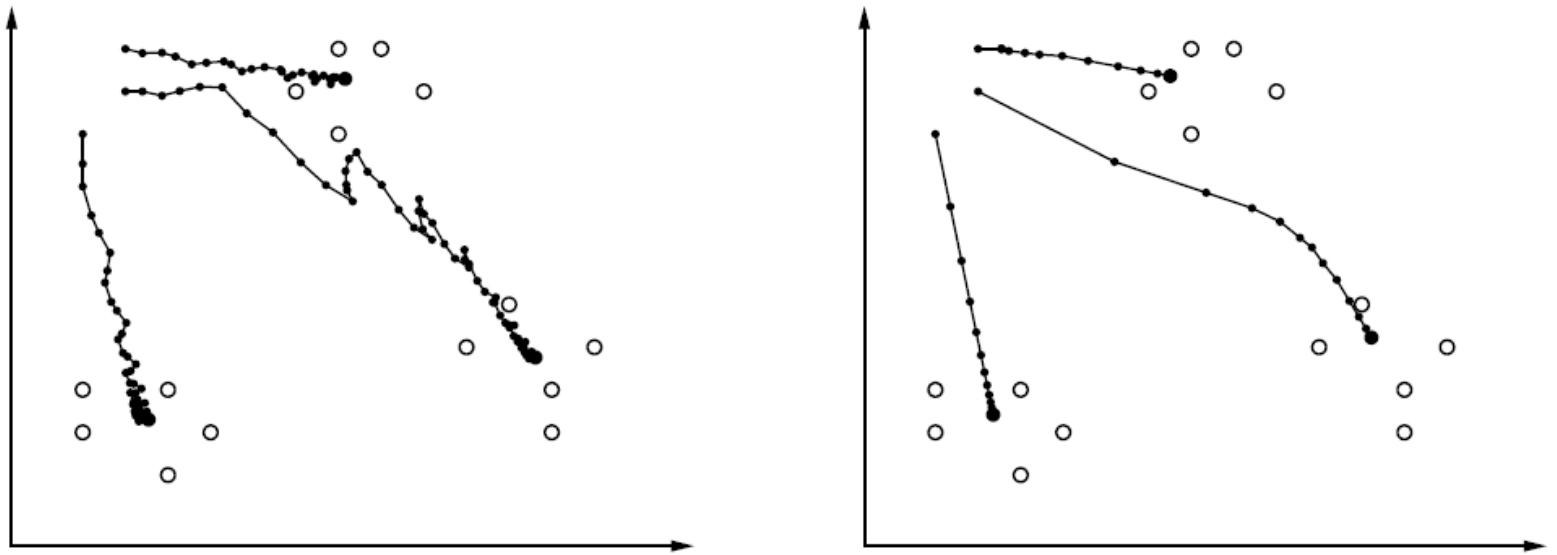
Anziehungsregel

Abstoßungsregel

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

Lernende Vektorquantisierung: Beispiel

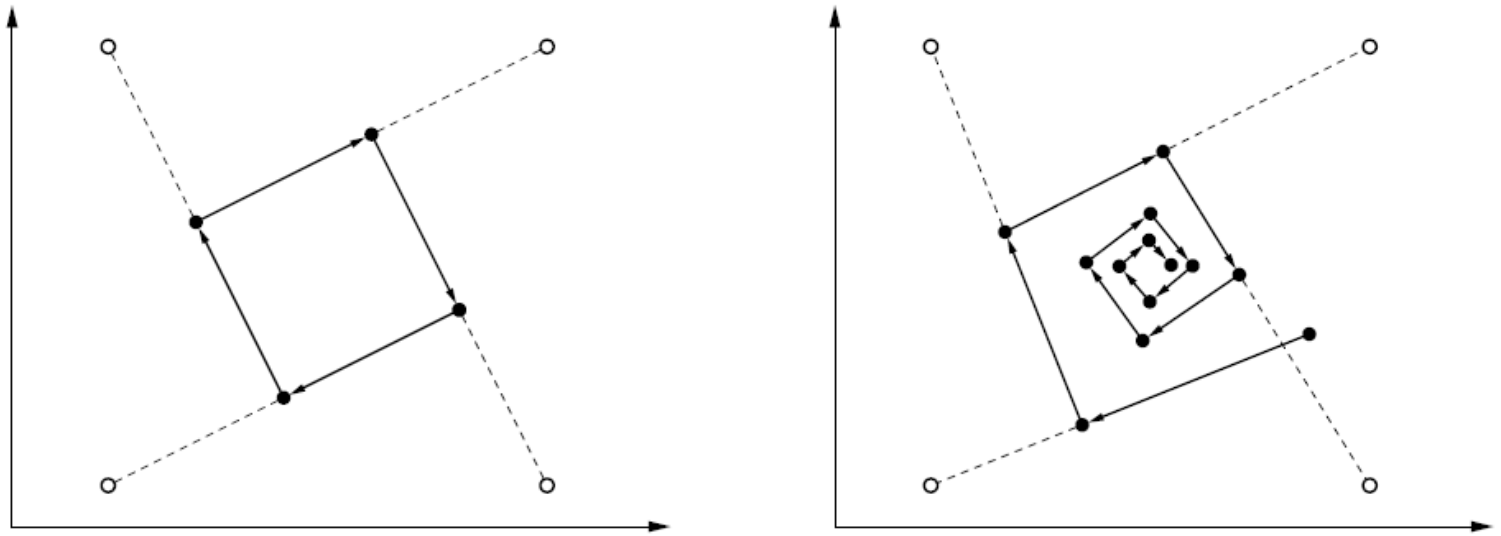
Anpassung der Referenzvektoren



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

Lernende Vektorquantisierung: Verfall der Lernrate

Problem: feste Lernrate kann zu Oszillationen führen



Lösung: **zeitabhängige Lernrate**

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

Lernende Vektorquantisierung: Klassifikation

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:

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

  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.

Selbstorganisierende Karten

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

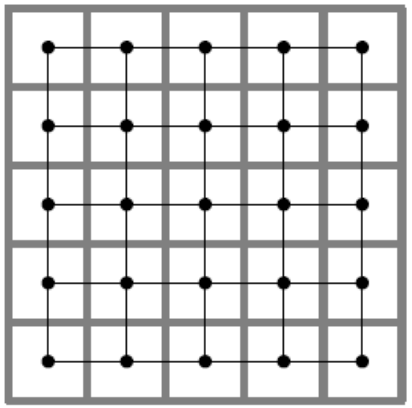
$$f : \mathbb{R}_0^+ \to [0, 1] \quad \text{with} \quad f(0) = 1 \quad \text{and} \quad \lim_{x \to \infty} f(x) = 0.$$

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

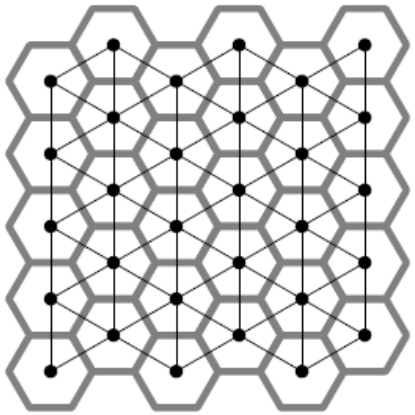
$$d_{\text{neurons}} : U_{\text{out}} \times U_{\text{out}} \to \mathbb{R}_0^+ .$$

Selbstorganisierende Karten: Nachbarschaft

Nachbarschaft der Ausgabeneuronen: Neuronen bilden ein Gitter



quadratisches Gitter

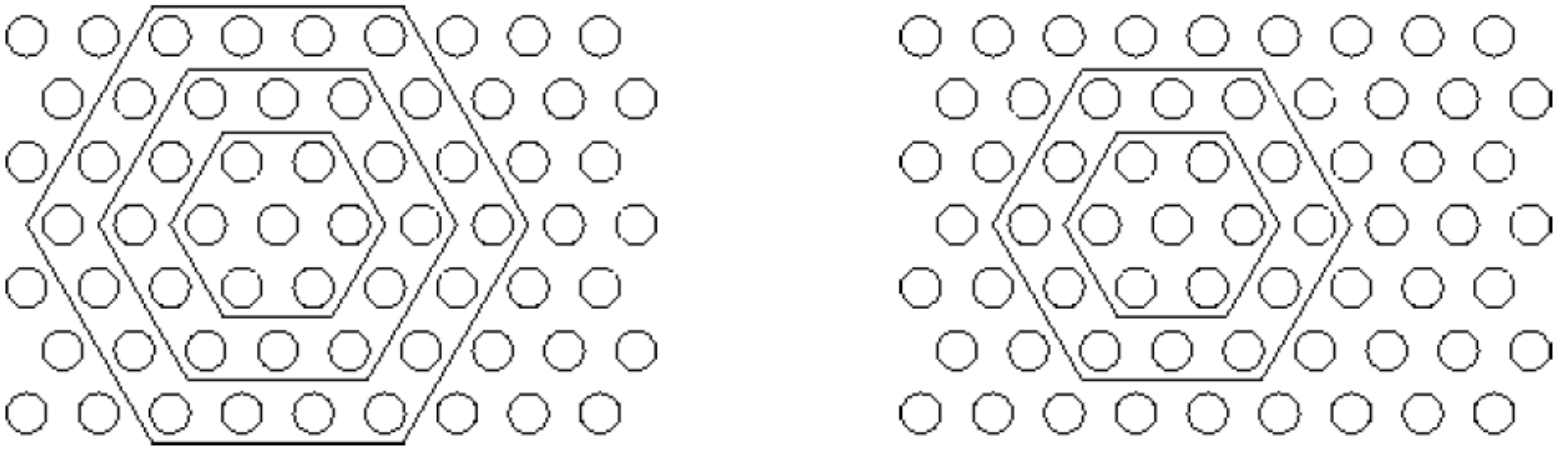


hexagonales Gitter

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

Selbstorganisierende Karten: Nachbarschaft

Nachbarschaft des Gewinnerneurons



Der Nachbarschaftsradius wird im Laufe des Lernens kleiner.

Selbstorganisierende Karten: Struktur

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.