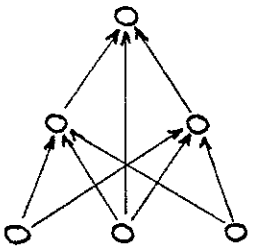
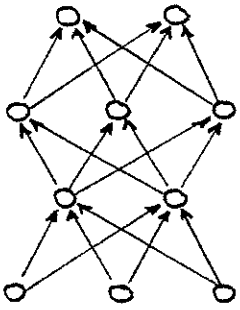
# VI. Multilayer Perceptrons



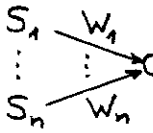




→ *Feedforward-Netzwerke mit einer oder mehreren Schichten von Hidden Units*

→ *Beliebig komplizierte Klassifizierungsgebiete realisierbar*

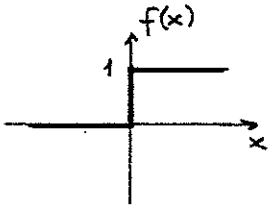
→ *Lernverfahren ?*

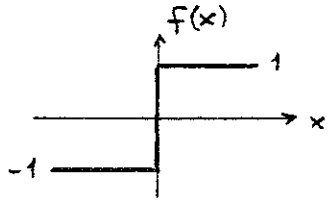
Aktivierungsfunktionen :



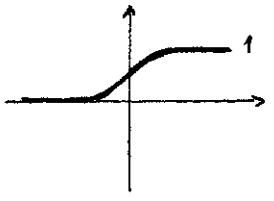

$$S = f\left(\sum_i W_i S_i\right)$$

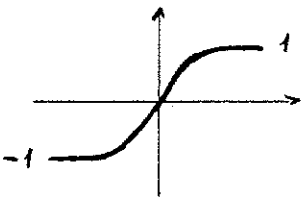
- Stufenfunktionen :




- Sigmoide Funktionen:




*meistens:* $f(x) = \dfrac{1}{1+e^{-x}}$ $\qquad$ $f(x) = \dfrac{1-e^{-x}}{1+e^{-x}}$

→ $f'(x) = f(x)\,[1-f(x)]$ $\qquad$ $f'(x) = \frac{1}{2}\,[1-f(x)^2]$

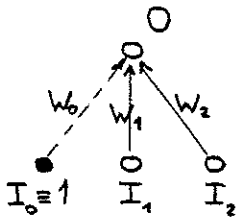
# VI.1 Rolle der Hidden Units

Die "hidden units" (inneren Neuronen) konstruieren eine interne Darstellung der Inputmuster, welche es erlaubt, die gewünschte Input/Output-Abbildung zu realisieren.

Beispiel: XOR-Problem :

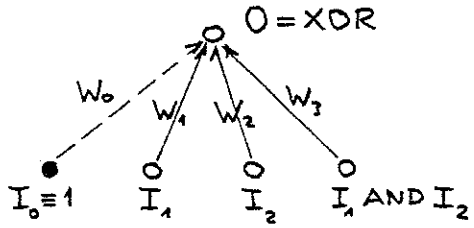
| $I_1$ | $I_2$ | XOR | AND |
|---|---|---|---|
| + | + | − | + |
| + | − | + | − |
| − | + | + | − |
| − | − | − | − |

- *Perceptron*



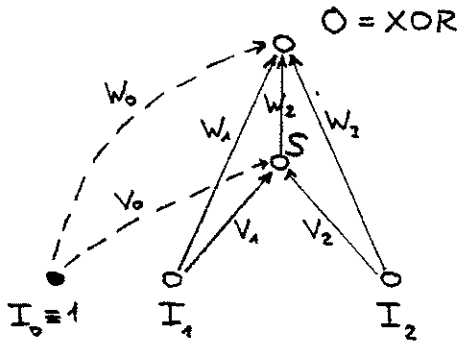

*kann XOR-Abbildung nicht implementieren !*

- *Aber erweiterte Input-Darstellung macht Perceptron-Lösung möglich:*




z.B. $W_0 = -1$, $W_1 = +1$, $W_2 = +1$, $W_3 = -2$

- *Realisierung mit 1 Hidden Unit:*




Das "Hidden" Neuron S konstruiert eine geeignete interne Darstellung (z.B. $S = I_1$ AND $I_2$) automatisch während dem Lernprozess!

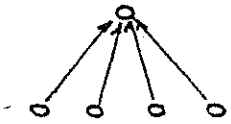
# VI.2 Lernvermögen

## 1) Realisierbare Klassifizierungsgebiete:

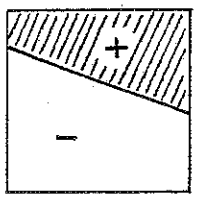
- Inputs reell $(-\infty < I_i < +\infty)$
- 1 Output mit $f(x) = sign(x)$ [d.h. $0 = \pm 1$]

Inputraum:

Perceptron:

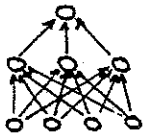


→

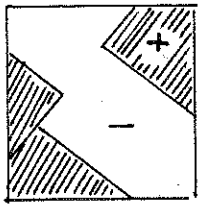



Durch Hyperebene begrenzter Halbraum

1 hidden Layer:

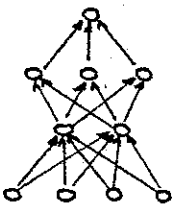


→

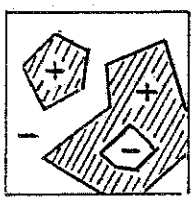



Nichtüberlappende konvexe Polygone (offen oder abgeschlossen)

2 hidden layers:



→




Beliebige Gebiete

(Komplexität nur durch Zahl der Hidden Units begrenzt)

→ **2 Schichten von Hidden Units genügen, um jede Klassifizierung realisieren zu können!**

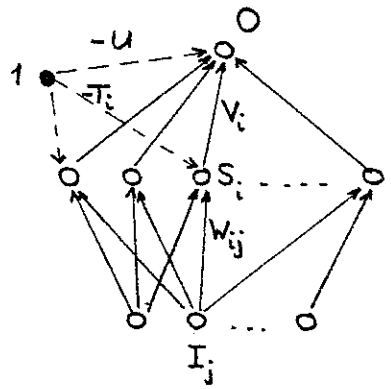
## 2) Boole'sche Funktionen:

- Inputs $I_1, \ldots, I_N$ , $I_j = \pm 1$
- Output $O = \pm 1$

$\longrightarrow$ Boole'sche Funktion $O = f_B(I_1, \ldots, I_N)$

$\longrightarrow$ **Ein Netzwerk mit 1 Schicht hidden Units kann jede beliebige Boole'sche Funktion realisieren!**

Beweis konstruktiv:




$i = 0, \ldots, 2^N - 1$ ($2^N$ hidden units)

$j = 0, \ldots, N-1$ (N Input units)

$$S_i = \text{sign}\left(\sum_j W_{ij} I_j - T_i\right) \quad , \quad O = \text{sign}\left(\sum_i V_i S_i - U\right)$$

wähle: $W_{ij} = \begin{cases} +b & \text{falls "j-tes bit der Binärdarst. von i" = 1} \\ -b & \text{sonst} \end{cases}$

$T_i = (N-1)\, b$

$\longrightarrow$ Für i-tes Inputmuster ($i = 0, \ldots, 2^N - 1$) ist $S_i = +1$ und $S_k = -1$ , $k \neq i$ .

wähle weiter:

$V_i = p D_i$ , $D_i$ = vorgegebener Output für i-tes Inputmuster

$U = -\sum_i V_i$

$\longrightarrow$ $0 = D_i$ für i-tes Inputmuster !

*Bemerkung:* Der Beweis zeigt nur, dass es mit $2^N$ Hidden Units geht, aber nicht wieviele Hidden Units es mindestens braucht.

3) Stetige Funktionen:

$\longrightarrow$ Jede stetige Funktion $\mathbb{R}^n \to \mathbb{R}^m$ kann durch ein Feedforward-Netzwerk [n Input units, m (lineare) Output units] mit nur 1 Schicht Hidden Units [$\sqcup$ oder $\sqcap$] beliebig genau approximiert werden!

[K. Hornik et al, Neural Networks 2, 359 (1989)]

ZUSAMMENFASSUNG:

- Jedes Klassifizierungs- oder Modellierungsproblem lässt sich durch ein neuronales Netzwerk mit höchstens 2 Schichten von Hidden Units lösen!

  [Das bedeutet aber nicht, dass die Beschränkung auf 1 oder 2 hidden layers immer optimal ist, z.B. bezüglich Lernverhalten!]

## VI.3 Lernen durch Fehlerminimierung

- Lernbeispiele: $\underline{I}^\nu / \underline{D}^\nu$, $\nu = 1, \ldots, N$
- $\underline{D}^\nu$ = vorgegebener Output
- Output des Netzwerks: $\underline{O}^\nu = \underline{O}(\underline{I}^\nu, \{W_{ij}\})$
- Quadratischer Outputfehler:

$$F_\nu = \frac{1}{2} \sum_i (D_i^\nu - O_i^\nu)^2 \qquad \text{[Fehler für Beispiel } \nu\text{]}$$

$$F = \sum_\nu F_\nu \qquad \text{[totaler Outputfehler]}$$

→ **LERNEN = Bestimmen der $W_{ij}$, sodass $F = \sum_\nu F_\nu = \min$**

GRADIENTENVERFAHREN:

→ *Iterative Minimierung von $F$: $W_{ij} \longrightarrow W_{ij} + \Delta W_{ij}$*

a) Example-by-Example Learning:

$$\Delta W_{ij} = -\eta \frac{\partial F_\nu}{\partial W_{ij}}$$

→ Gewichtsänderung direkt nach jeder Präsentation eines Lernbeispiels.

(Lernbeispiele in zufälliger Reihenfolge!)

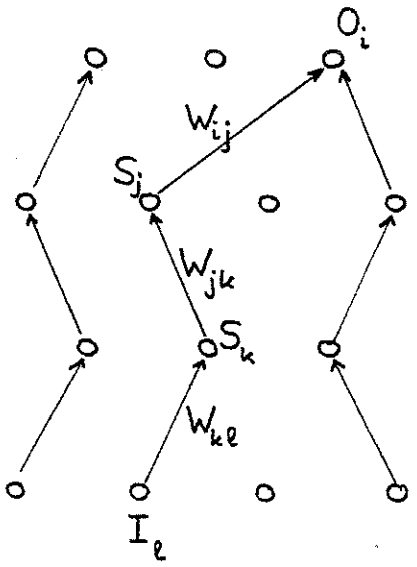
b) Batch-Learning:

$$\Delta W_{ij} = -\eta \frac{\partial F}{\partial W_{ij}}, \quad F = \sum_\nu F_\nu$$

→ Gewichtsänderung erst nach Präsentation aller Lernbeispiele.

*Bemerkung:* Bei Gradientenverfahren müssen Fehlerfunktion $F$ und Aktivierungsfunktion $f$ differenzierbar sein!

## VI.4 Der "Error-Backpropagation" Algorithmus




$$O_i = f\left(\sum_j W_{ij} S_j\right) \equiv f(V_i)$$

$$\uparrow$$

$$S_j = f\left(\sum_k W_{jk} S_k\right) \equiv f(V_j)$$

$$\uparrow$$

$$S_k = f\left(\sum_l W_{kl} I_l\right) \equiv f(V_k)$$

$$\uparrow$$

$$I_l$$

$$F = \frac{1}{2} \sum_i (D_i - O_i)^2 = \min \quad : \quad \Delta W_{ij} = -\eta \frac{\partial F}{\partial W_{ij}}$$

$$\longrightarrow \quad \Delta W_{ij} = \eta (D_i - O_i) f'(V_i) S_j \equiv \delta_i \cdot S_j$$

$$\Delta W_{jk} = \eta \sum_i (D_i - O_i) f'(V_i) W_{ij} f'(V_j) S_k$$

$$= \sum_i \delta_i W_{ij} f'(V_j) S_k \equiv \delta_j \cdot S_k$$

$$\Delta W_{kl} = \sum_j \delta_j W_{jk} f'(V_k) I_l \equiv \delta_k \cdot I_l$$

$\longrightarrow$ <u>Backpropagation:</u>

$$\delta_i = \eta (D_i - O_i) f'(V_i) \longrightarrow \Delta W_{ij} = \delta_i S_j$$

$$\downarrow$$

$$\delta_j = \sum_i \delta_i W_{ij} \cdot f'(V_j) \longrightarrow \Delta W_{jk} = \delta_j S_k$$

$$\downarrow$$

$$\delta_k = \sum_j \delta_j W_{jk} \cdot f'(V_k) \longrightarrow \Delta W_{kl} = \delta_k I_l$$

→ *Der Backpropagation-Algorithmus ist eine effiziente Implementierung der Gradientenmethode für Feedforward-Netzwerke.*

Probleme beim Backpropagation-Lernverfahren :

Fehlerfunktion $F(\underline{W})$ ist oft sehr kompliziert und kann viele lokale Minima und lange, flache Täler enthalten :

→ *schlechte Lösungen (lokale Minima)*

→ *Lernprozess sehr langsam (flache Täler, $\eta$ klein)*

→ *Oszillationen, keine Konvergenz ($\eta$ zu gross)*

## VI.5 Varianten, "Tricks"

1) *Überrelaxation ("momentum term"):*

$$\Delta W_{ij}(k+1) = -\eta \frac{\partial F}{\partial W_{ij}} + \alpha \Delta W_{ij}(k)$$

[$\Delta W_{ij}(k)$ = Gewichtsänderung nach k-ter Lernbeispielpräsentation]

2) *Akkumulation der Gewichtsänderungen über gewisse Anzahl n von Lernbeispielen.*

3) *Gewichtszerfall :*

$$\Delta W_{ij} = -\eta \frac{\partial F}{\partial W_{ij}} - \beta W_{ij}$$

4) Von Zeit zu Zeit zufällige Gewichtsänderungen vornehmen.

5) Alternative Fehlerfunktionen benutzen:

Statt $F^\nu = \frac{1}{2} \sum_i (D_i^\nu - O_i^\nu)^2$

(Summe der quadrierten Outputfehler)

z.B.:

$$F^\nu = \sum_i \left[ D_i^\nu \log \frac{D_i^\nu}{O_i^\nu} + (1-D_i^\nu) \log \frac{1-D_i^\nu}{1-O_i^\nu} \right] \qquad \text{falls } 0 < D,O < 1$$

$$F^\nu = \sum_i \left[ \frac{1}{2}(1+D_i^\nu) \log \frac{1+D_i^\nu}{1+O_i^\nu} + \frac{1}{2}(1-D_i^\nu) \log \frac{1-D_i^\nu}{1-O_i^\nu} \right]$$

falls $-1 < D,O < +1$

("Informationsdifferenz" !)

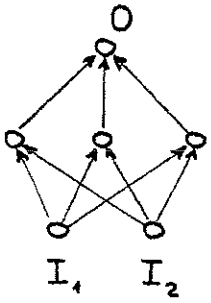
6) Überhaupt kein Gradientenverfahren zur Minimierung von F verwenden, sondern z.B. ein stochastisches Optimierungsverfahren:

- Simulated Annealing
- Evolutionsalgorithmen
- etc

(siehe Kapitel VIII)

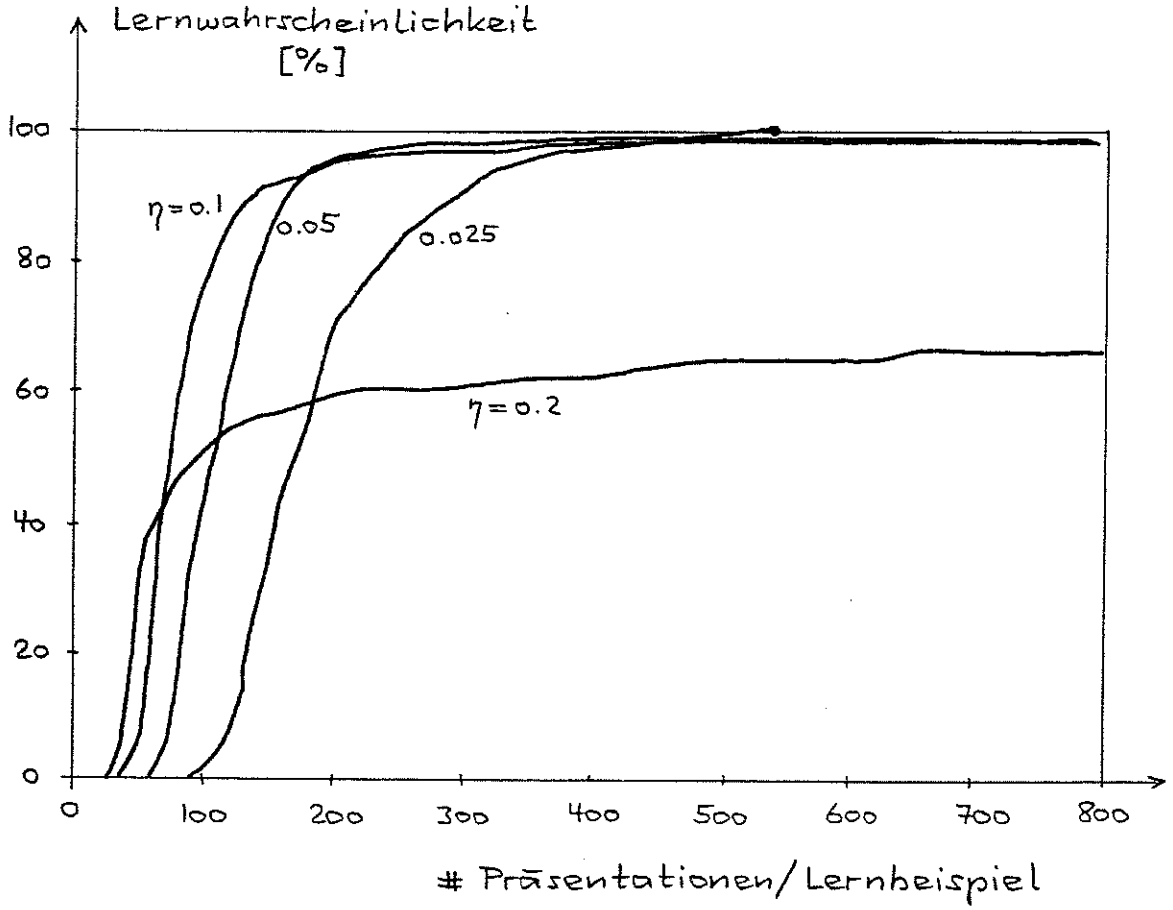
# VI.6 Beispiele

## A) XOR-Problem:




| $I_1$ | $I_2$ | O |
|---|---|---|
| + | + | − |
| + | − | + |
| − | + | + |
| − | − | − |

- 3 hidden units : → 13 Gewichte (inkl. Schwellen)
- Backpropagation $\left[ f(x) = \frac{1-e^{-x}}{1+e^{-x}} \right.$ , $\left. O > 0.8 \text{ bzw. } < -0.8 \right]$
- $\eta = 0.025, 0.05, 0.1, 0.2$ ; $\alpha = 0.9$
- Gewichtsänderung nach jeder Präsentation eines Lernbeispiels



## B) Erkennung von handgeschriebenen Ziffern

(Urs Müller, 1995)

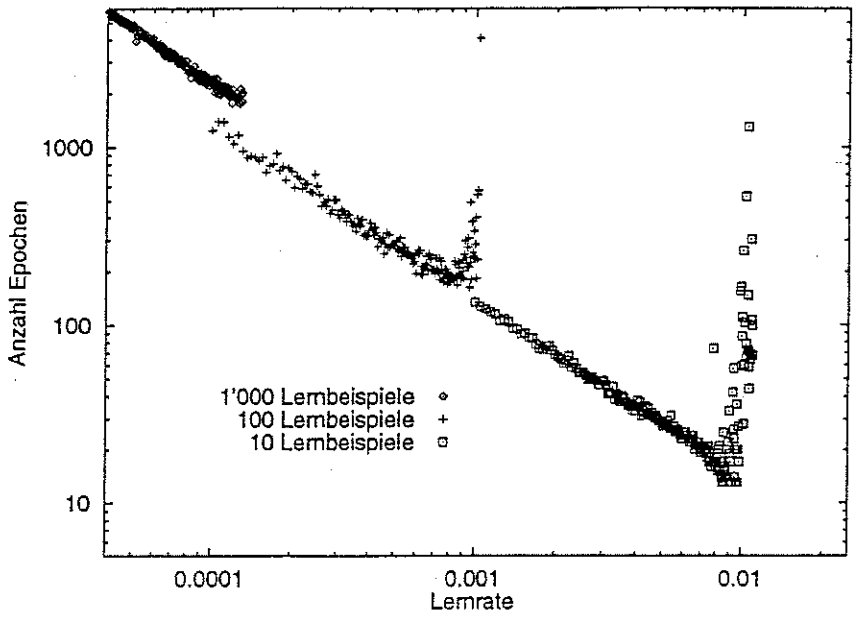
- Gegeben: Handgeschriebene Ziffern (0 bis 9), dargestellt als 16 x 16 Pixelbilder
  [Pixel = -1 (weiss) oder +1 (schwarz)

- Multilayer-Perceptron mit 1 Schicht von Hidden Units:
  256 Inputneuronen
  100 Hidden Units
  10 Outputneuronen (je 1 für jede Ziffer)

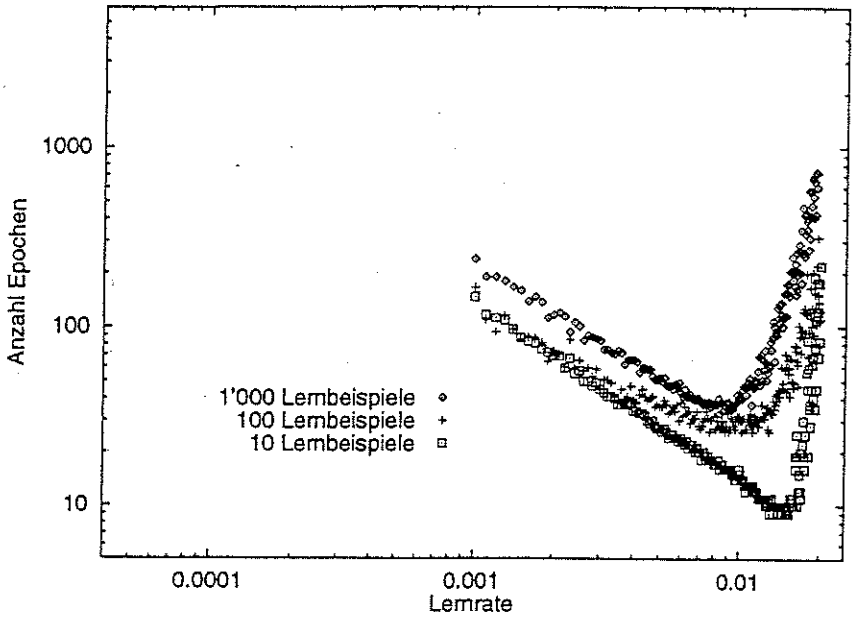
  Aktivierungsfunktion:
  $f(x) = 1.1 \cdot \tanh(1.5\,x)$

- Es wird trainiert, bis alle Lernbeispiele richtig erkannt werden. (Error-Backpropagation)

- Figuren B1 und B2 zeigen Anzahl notwendige Epochen (Epoche = Präsentation aller Lernbeispiele) als Funktion der Lernrate $\eta$ für verschieden grosse Lernsets.




Figur B1 : Batch-Learning




Figur B2 : Example-by-Example Learning

BEMERKUNGEN :

- Konvergenzgeschwindigkeit nimmt mit zunehmender Lernrate zuerst zu (Anzahl notwendiger Epochen sinkt) und dann wieder ab.
  Bei noch grösseren Lernraten als in den Figuren eingezeichnet : → keine Konvergenz mehr.

- Für kleine Lernsets (10 Lernbeispiele) ist die Konvergenzgeschwindigkeit für beide Verfahren (Batch und Example-by-Example) etwa gleich. Auch die optimalen Lernraten sind ähnlich.

- Für grosse Lernsets ist Example-by-Example Learning wesentlich schneller als Batch Learning.

- Batch Learning kann aber auch Vorteile haben:
  z.B. werden bei der Approximation von kontinuierlichen Input/Output-Zusammenhängen mit Batch Learning grössere Genauigkeiten erzielt als mit Example-by-Example Learning!

# GRADIENTEN - LERNVERFAHREN

(ERGÄNZUNG)

NOTATION: $\underline{W} = (W_1, W_2, \ldots, W_n)$ , $O^\mu = O(\underline{I}^\mu, \underline{W})$

$F(\underline{W}) = \min$

[ z.B. $F(\underline{W}) = \frac{1}{2} \sum_{\mu=1}^{N} (D^\mu - O^\mu)^2$ ]

GRADIENTENVERFAHREN:

Iterationsschritte: $k = 1, 2, \ldots$

$$\longrightarrow \quad \underline{W}^{k+1} = \underline{W}^k + \Delta \underline{W}^k \quad , \quad \Delta \underline{W}^k = \Delta \underline{W}(\underline{g}^k)$$

$$\underline{g} = \left( \frac{\partial F}{\partial W_1}, \ldots, \frac{\partial F}{\partial W_n} \right) \quad , \quad \underline{g}^k = \underline{g}(\underline{W}^k)$$

EINFACHE GRADIENTENVERFAHREN:

[ z.B. Backpropagation (Batch-Mode!) ]

$\underline{W}^0$ zufällig

$\underline{W}^{k+1} = \underline{W}^k - \eta \, \underline{g}^k$

$\eta = \text{const.}$ (genügend klein!)

→ *Probleme bei langen flachen Tälern, etc.*

# METHODE DER KONJUGIERTEN GRADIENTEN

[siehe z.B.:
R. Battiti, Neural Computation 4, 141-166 (1992)]

$\underline{W}^0$ zufällig, $\underline{p}^0 = \underline{g}^0$

$\rightarrow \underline{W}^{k+1} = \underline{W}^k - \alpha_k \underline{p}^k$, $\alpha_k$ so, dass F entlang $\underline{p}^k$ minimal wird

$$\underline{p}^{k+1} = -\underline{g}^{k+1} + \beta_k \underline{p}^k, \quad \beta_k = \frac{(\underline{g}^{k+1} - \underline{g}^k) \cdot \underline{g}^{k+1}}{(\underline{g}^{k+1} - \underline{g}^k) \cdot \underline{p}^k}$$

- Falls F = quadr. Funktion der $W_{ij}$ :

  → Auf der Geraden durch $\underline{W}^{k+1}$ in Richtung $\underline{p}^{k+1}$ ist der Gradient von F in Richtung $\underline{p}^k$ immer gleich Null [k-te Minimierung wird nicht zerstört!].

  → Verfahren konvergiert in n Schritten! (n = Anzahl Gewichte)

- Neuronale Netzwerke:

  → F ist keine quadr. Funktion der $W_{ij}$.
  → Numerische Probleme
  → Varianten von KG-Verfahren

  Für viele Lernprobleme sind KG-Methoden wesentlich schneller als Backpropagation!