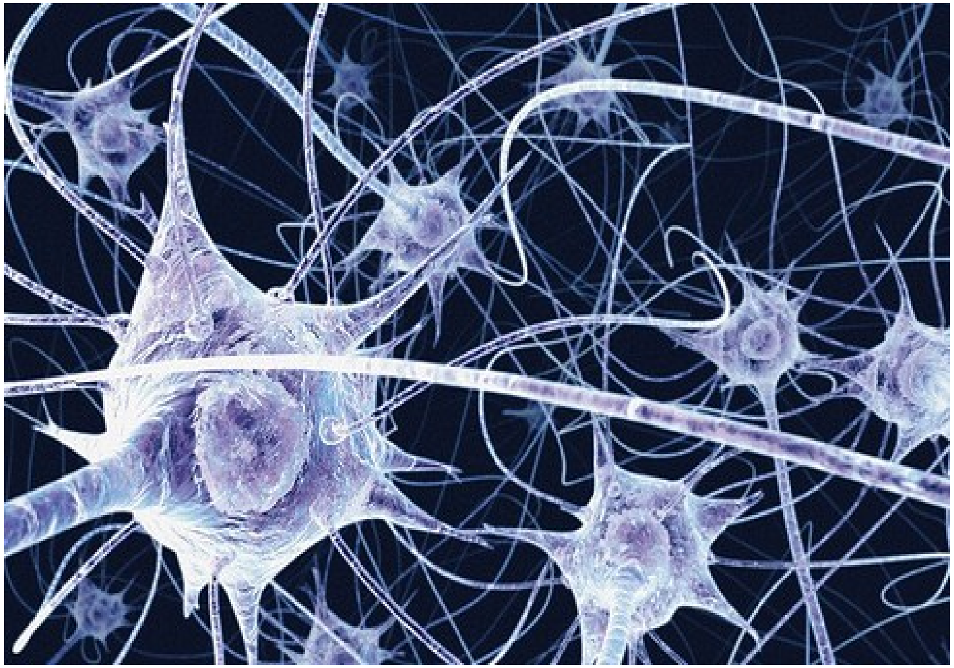
6 Neuronale Netze

6.1 Multilayer Perzeptron (MLP)

6.1.1 Einführung

Stark vereinfachter Versuch einer technischen Realisierung biologischer Neuronaler Netze.



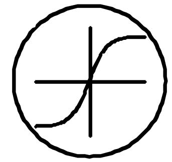
s. www.noows.de

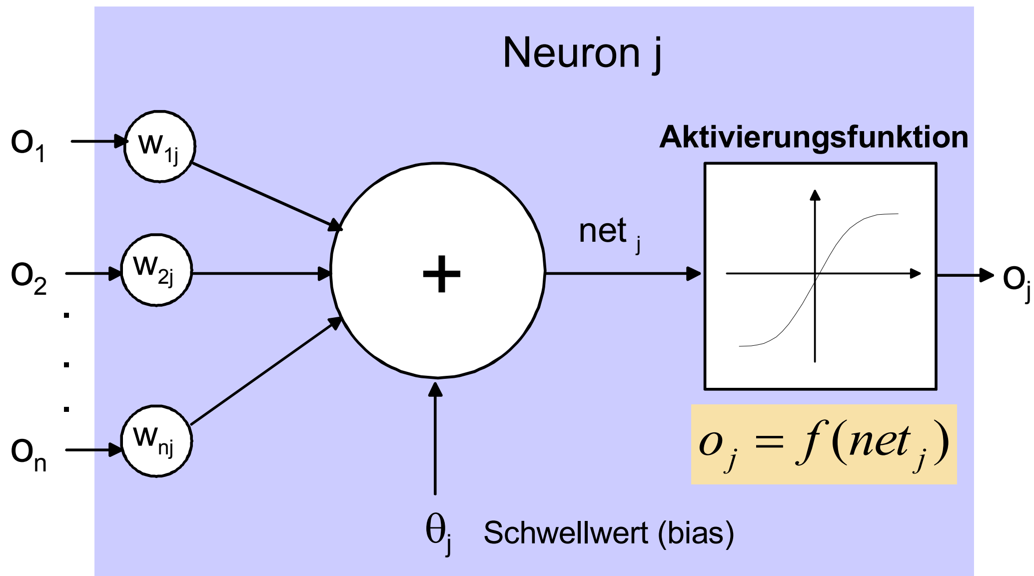
Zweck: - Funktionsapproximation
- Klassifikation

Lernstrategie:
- überwachtes Lernen

6.1.2 Aufbau

6.1.2.1 Das Neuron





Anm. zur Gewichtsnotation:

$w_{\text{Eingangsnummer (i), Nr. des Neurons (j)}}$

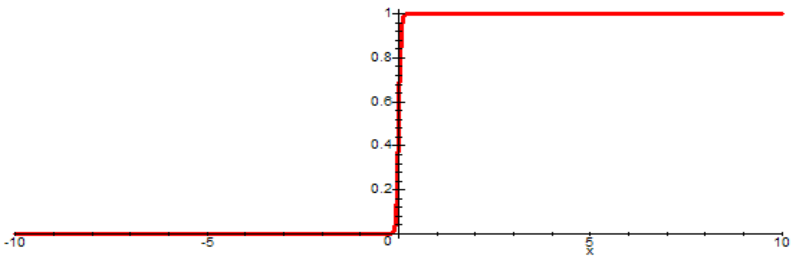
$$net_j = \sum_i o_i w_{ij} + \theta_j$$

6.1.2.2 Verschiedene Aktivierungsfunktionen $o_j = f(net_j)$

a) <u>Schrittfunktion</u>

Vorteil: sehr einfach berechenbar

Nachteil: nicht differenzierbar

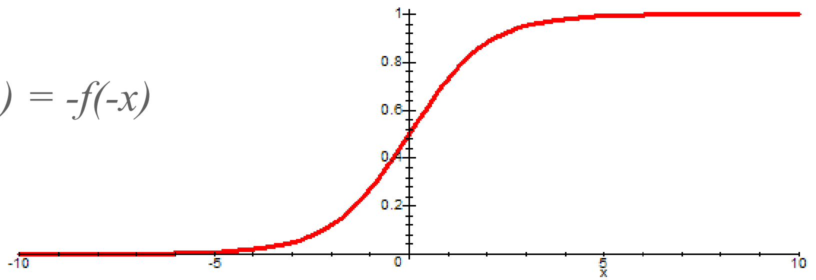


b) <u>Logistische Funktion (sigmoid)</u>

Vorteil: differenzierbar

Nachteil: keine ungerade Funktion $f(x) = -f(-x)$

$$f_{\log} = \frac{1}{1+e^{-x}}$$

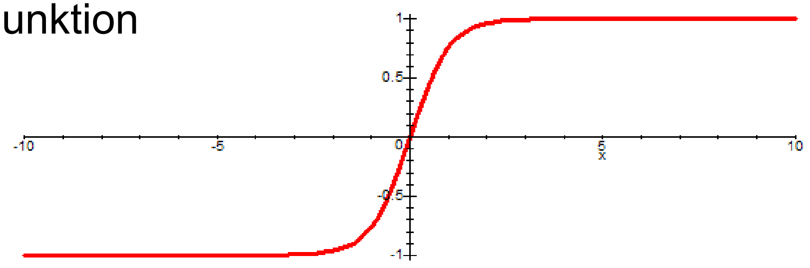


c) <u>Tangens hyperbolicus (sigmoid)</u>

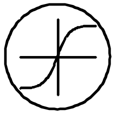
Vorteile: differenzierbar, ungerade Funktion

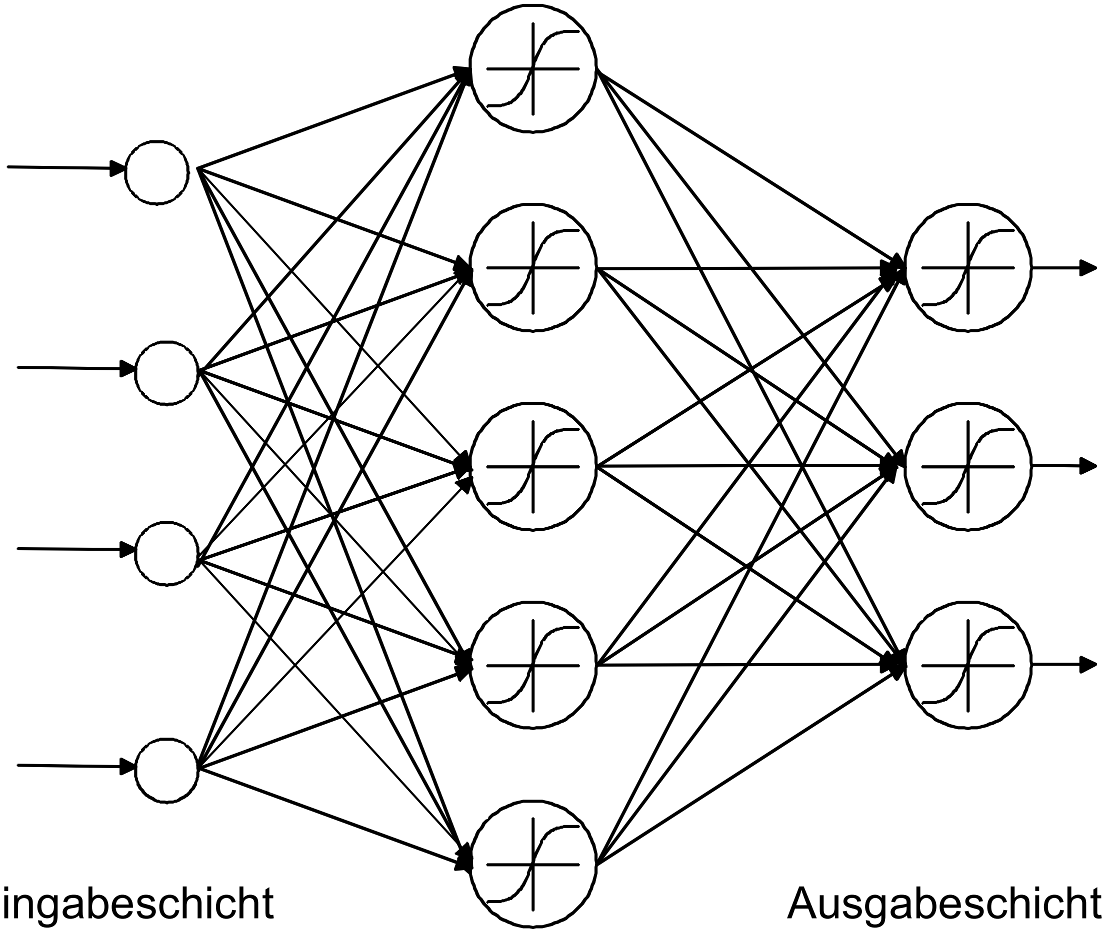
Nachteil: -

$$f_{\log} = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$



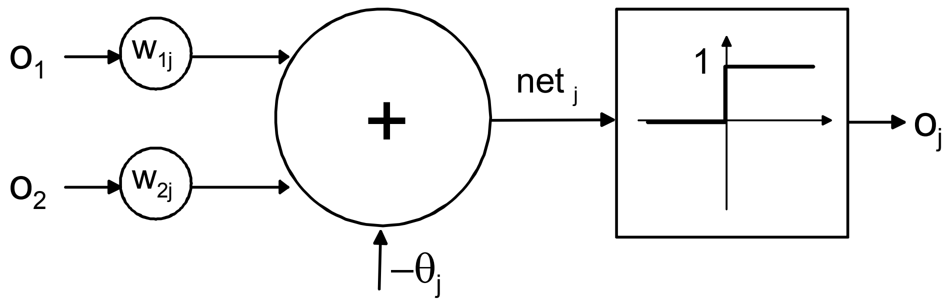
6.1.2.3 Verbindungsnetzwerk (feed forward NN)

Anm.:  = Neuron



6.1.3 Gedankenspiel zur Funktionsweise

6.1.3.1 Welche Funktionen kann ein einzelnes Neuron j repräsentieren ?



$$
\begin{array}{llll}
o_1 \cdot w_{1j} + o_2 \cdot w_{2j} - \theta_j & \geq & 0 & o_j = 1 \\
 & & \text{sonst} & o_j = 0
\end{array}
$$

o_j hat also dann den Wert 1, wenn gilt:

$$o_1 \cdot w_{1j} + o_2 \cdot w_{2j} \;\geq\; \theta_j$$

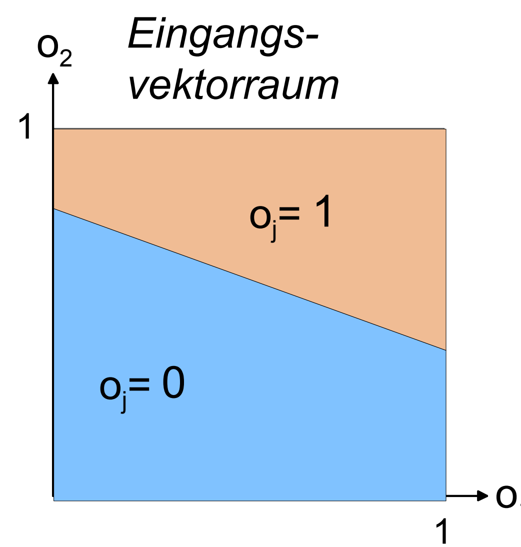
Die Gleichung der Trennlinie lautet somit

$$o_1 \cdot w_{1j} + o_2 \cdot w_{2j} \;=\; \theta_j$$

oder umgeformt nach o_2:

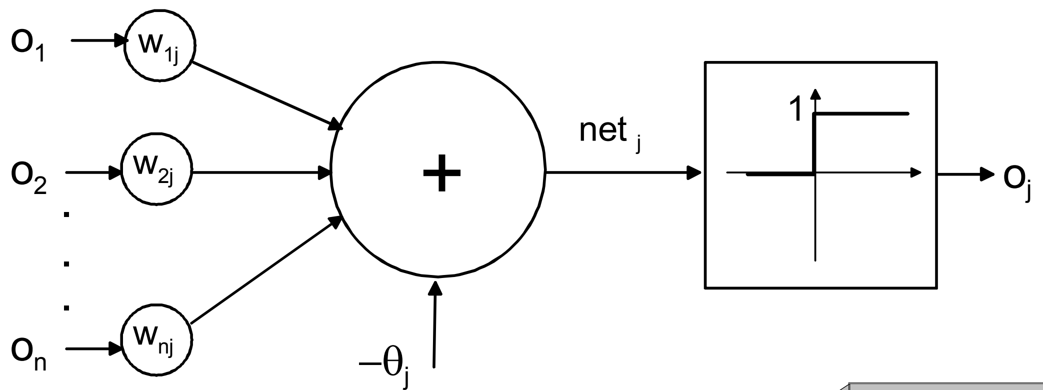
$$o_2 = -\frac{w_{1j}}{w_{2j}} \cdot o_1 + \frac{1}{w_{2j}} \cdot \theta_j$$

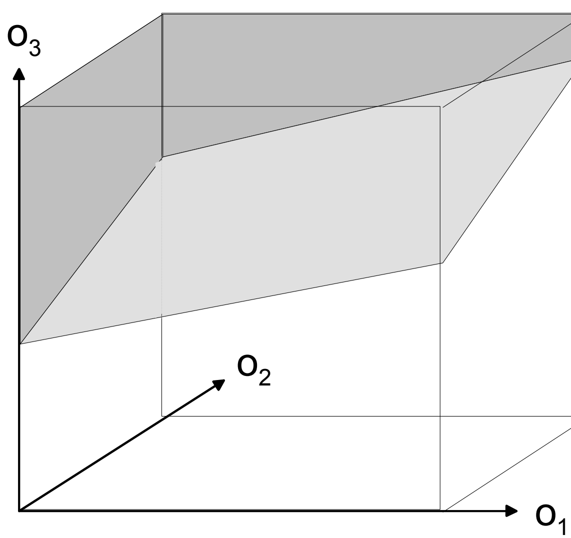
= Geradengleichung vom Typ y=mx+b



6.1.3.2 Verallgemeinerung auf n Eingänge

Ein Neuron mit n-Eingängen teilt den n-dimensionalen *Eingangsvektorraum* mit Hilfe einer *Hyperebene* in zwei Teile (*2 Halbräume*).





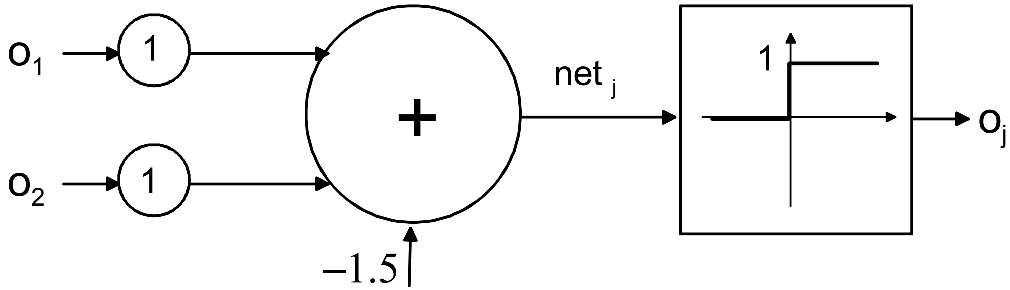
Beispiel:

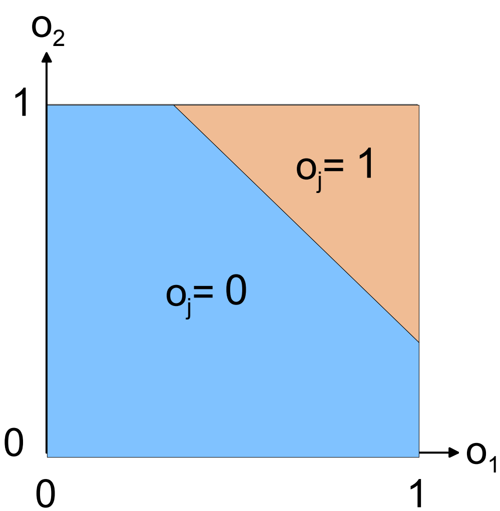
3 Eingänge

Fazit: Ein Neuron kann nur linear separierbare Mengen klassifizieren.

6.1.3.3 Gedankenexperiment: Realisierung logischer Funktionen durch ein Neuron

Beispiel: AND





o_j ist hat also dann den Wert 1, wenn gilt:

$$o_2 + o_1 - 1.5 \geq 0$$

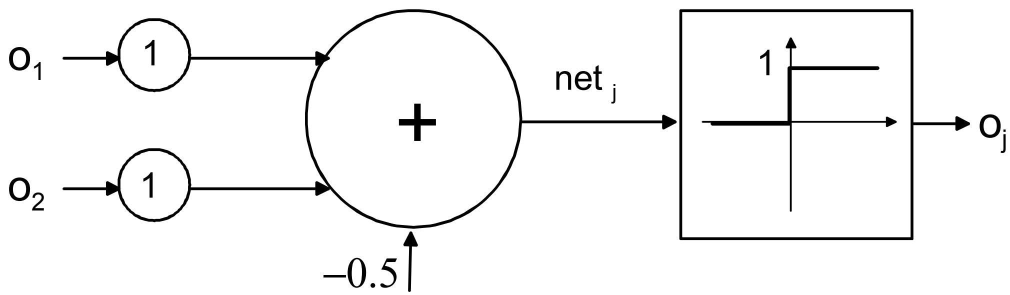
o_1	o_2	o_j
0	0	0
0	1	0
1	0	0
1	1	1

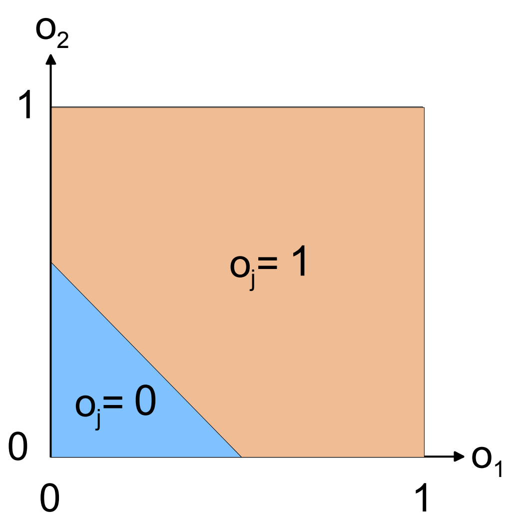
Gleichung der Trennlinie:

$$o_2 = -o_1 + 1.5$$

Frage: Was muss man tun, um ein Neuron mit NAND-Charakteristik zu erhalten ?

Beispiel: OR





o_j ist hat also dann den Wert 1, wenn gilt:

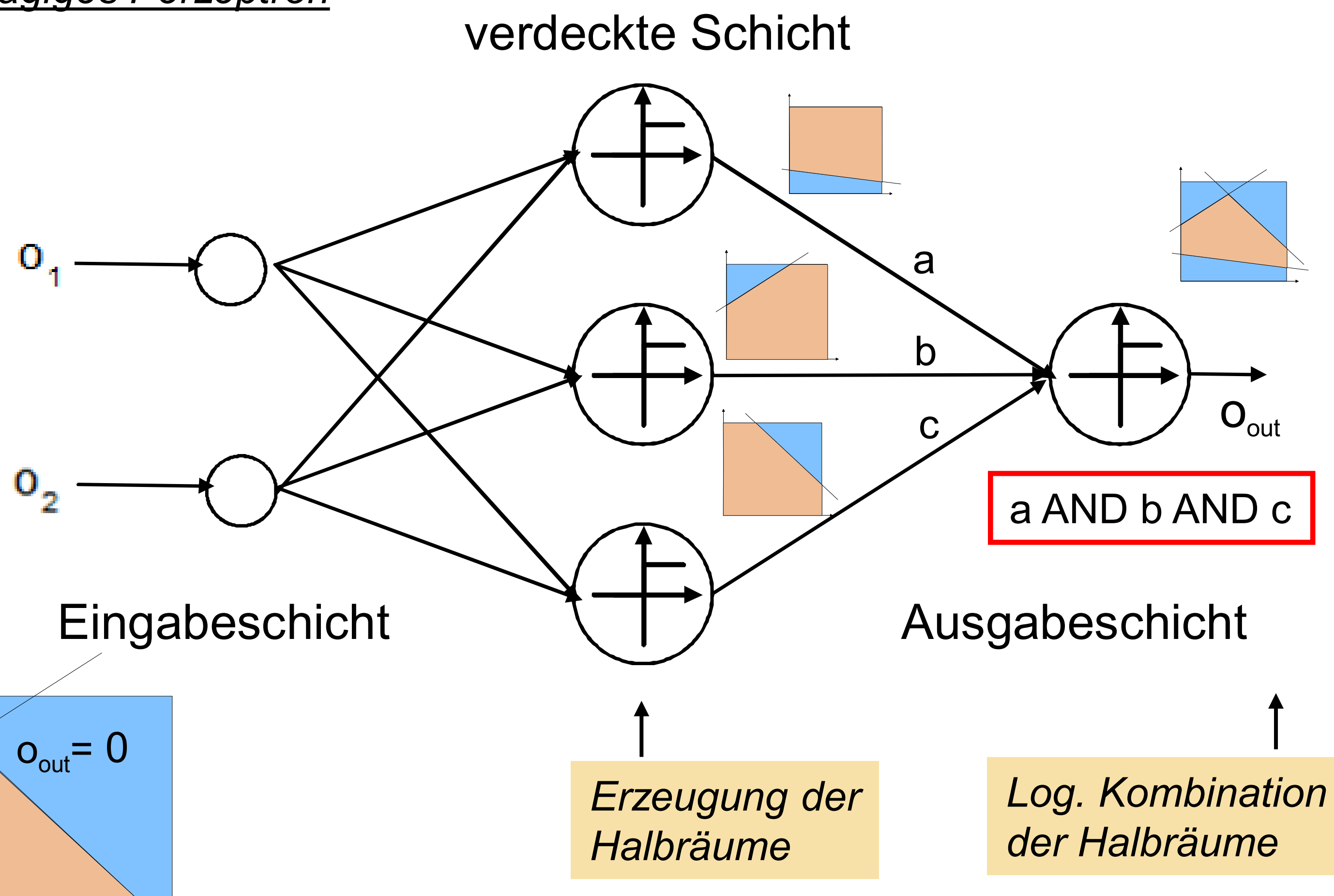
$$o_2 + o_1 - 0.5 \geq 0$$

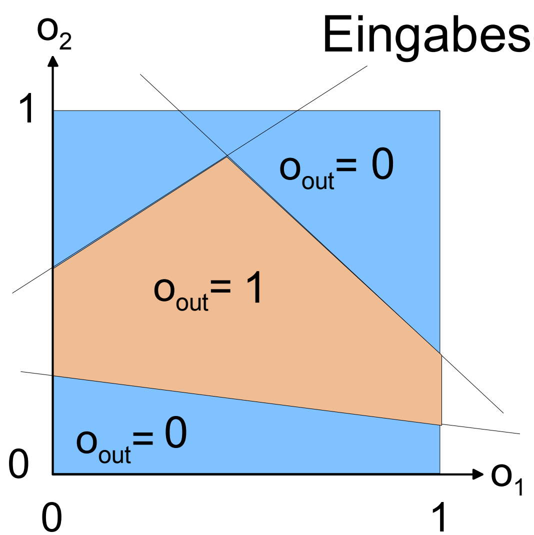
o_1	o_2	o_j
0	0	0
0	1	1
1	0	1
1	1	1

Gleichung der Trennlinie:

$$o_2 = -o_1 + 0.5$$

6.1.3.4 Zweilagiges Perzeptron



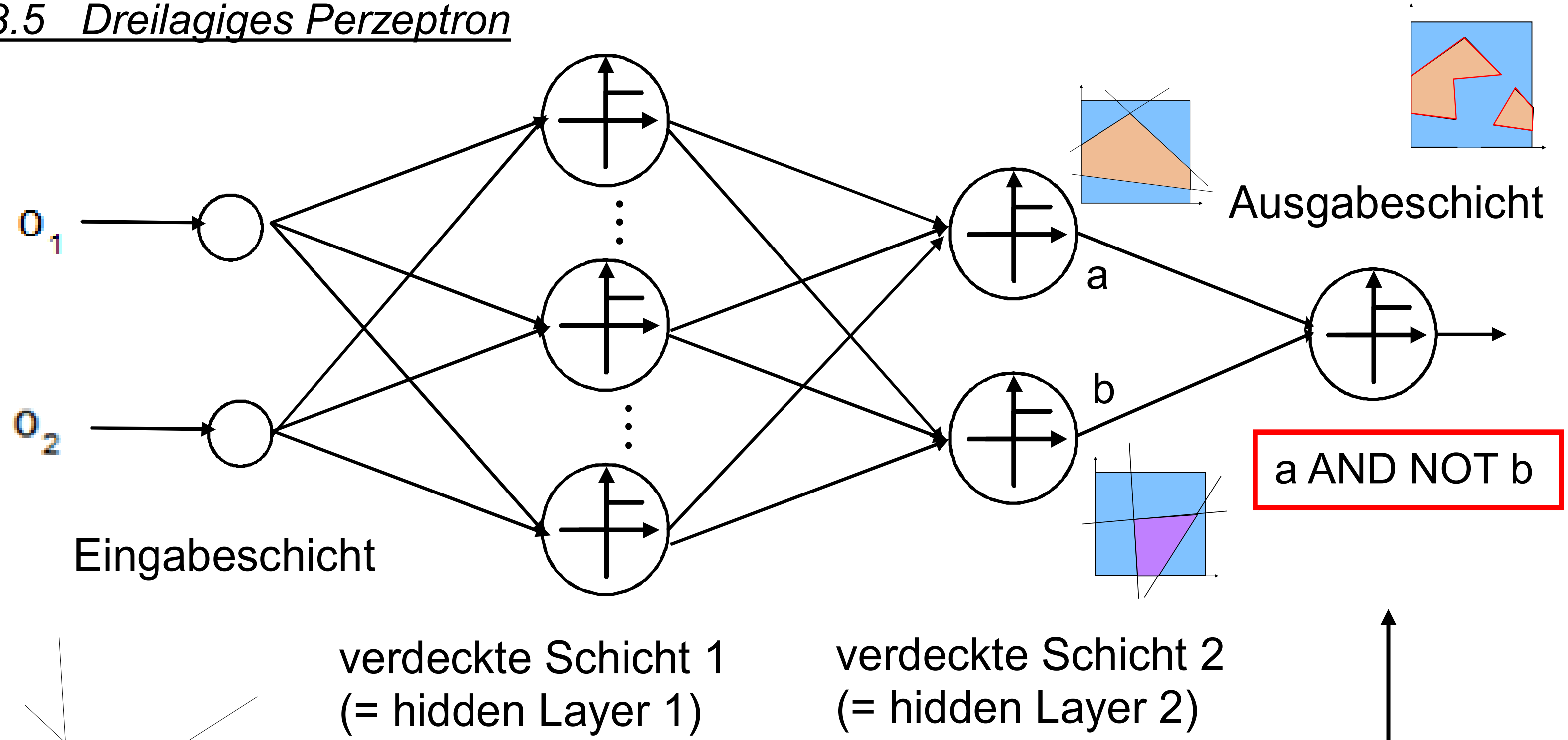


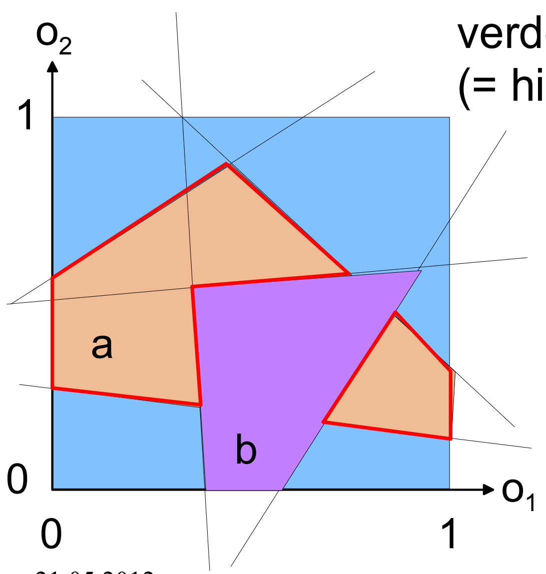
Durch log. Kombination von Halbräumen können konvexe Polygone erzeugt werden.

ÜBUNG: Zweilagiges Perzeptron

Geben Sie ein zweilagiges Perzeptron an, welches XOR-Verhalten besitzt.

6.1.3.5 Dreilagiges Perzeptron





Erzeugung der Halbräume

Log. Kombination der Halbräume

Log. Kombination der Polygone

Durch log. Kombination von <u>konvexen Polygonen</u> können <u>konkave Polygone</u> erzeugt werden.

aber Achtung:

Die gerade verwendete Argumentation zeigt nur, dass man mit einem 2HL-MLP auf jeden Fall jede Funktion approximieren kann (bzw. beliebig geformte 0-1-Bereiche).

Die Argumentation zeigt nicht, dass man mindestens 2 hidden Layer benötigt, um jede Funktion approximieren zu können (bzw. beliebig geformte 0-1-Bereiche).

Abkürzung:
2HL-MLP : Multilayer-Pezeptron mit 2 hidden Layern

6.1.3.6 Universal-approximation-Theorem (UAT)

Nach dem UAT gilt, dass ein Multilayer-Perzeptron (MLP) mit einem *hidden Layer* (*1HL-MLP*) grundsätzlich jede Funktion approximieren kann. (Cybenko 1988)

Anm.: eine genügende Anzahl von hidden Neuronen vorausgesetzt

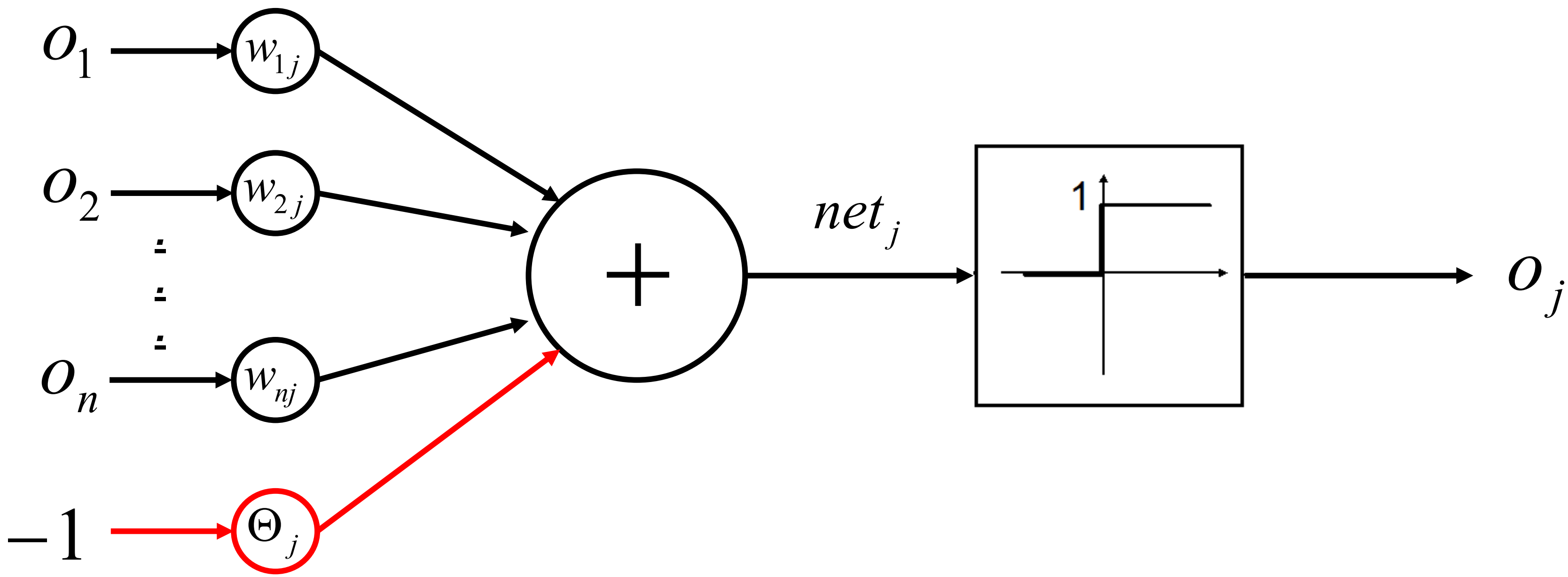
aber Achtung:

Das UAT ist ein reines Existenztheorem. Es sagt nichts darüber aus,

a) ob ein 1HL-MLP ein besseres Lernverhalten aufweist als ein nHL-MLP,

b) ob ein 1HL-MLP ein besseres Generalisierungsverhalten hat als ein nHL-MLP

6.1.3.7 Praktische Realisierung des Bias



Das Neuron erhält einen zusätzlichen Eingang, an dem konstant -1 anliegt.
Der Biaswert wird dann wie ein normales Gewicht verwendet (und trainiert).

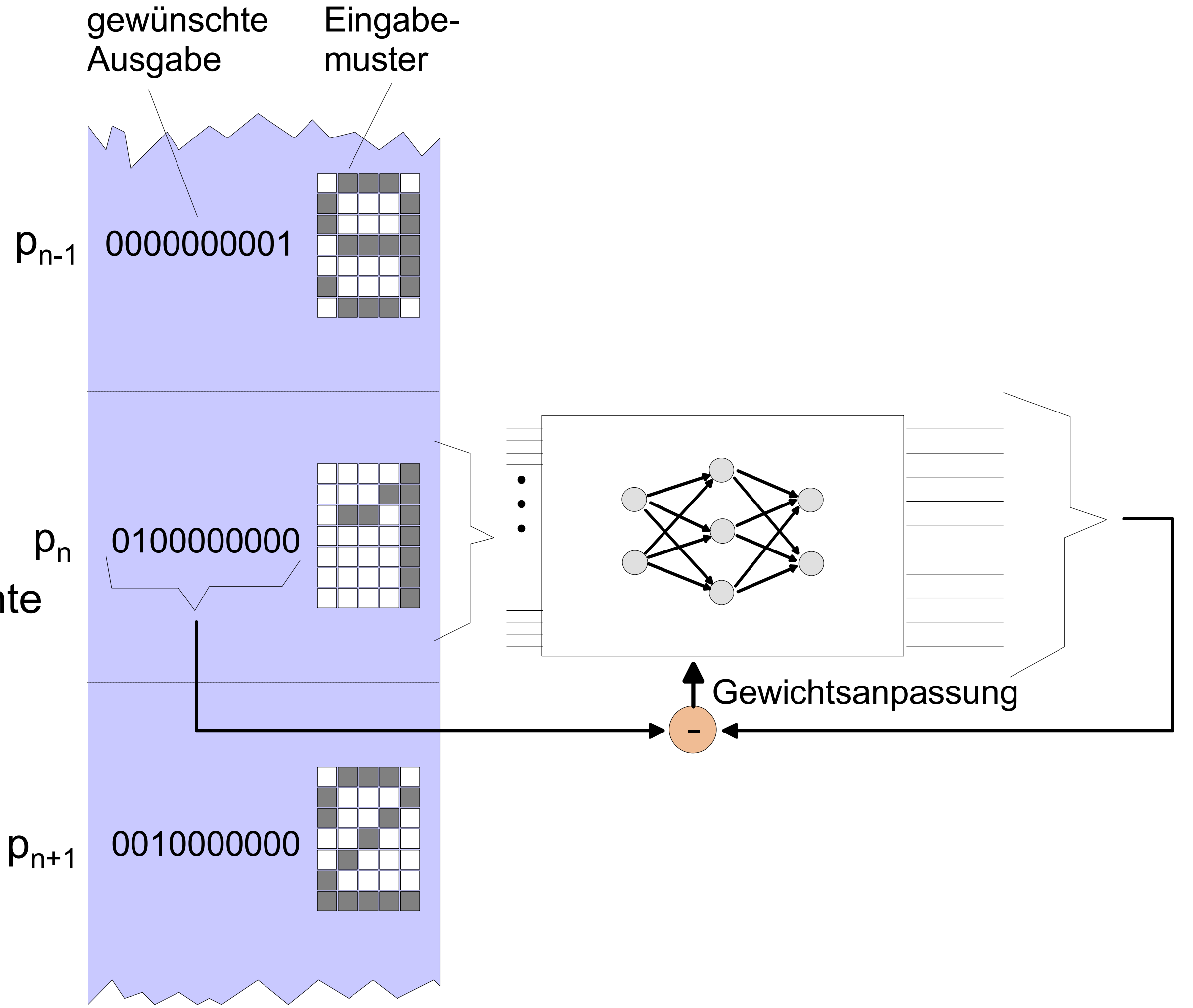
6.1.4 Einstellung der Gewichte durch Training

6.1.4.1 Vorüberlegungen

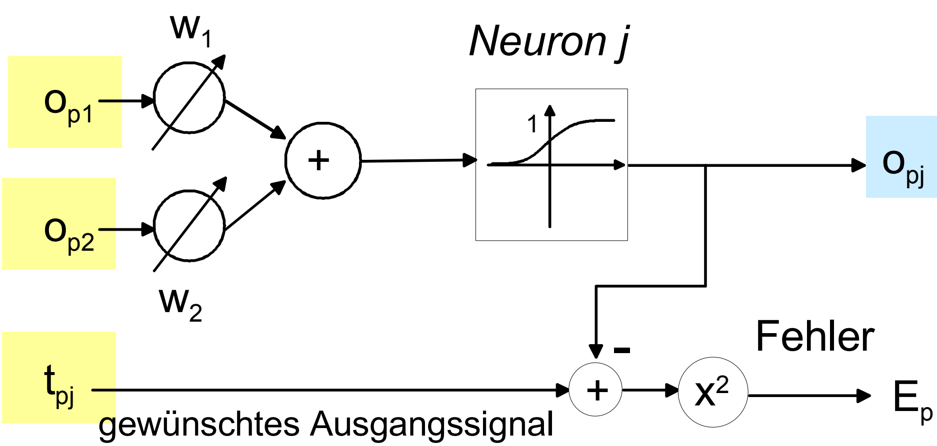
Wie können die Verbindungsgewichte eingestellt werden ?

→ *Training*

= Beeinflussung der Gewichte, so dass sich das gewünschte Verhalten einstellt.



6.1.4.2 Gedankenexperiment: Trainieren eines Neurons

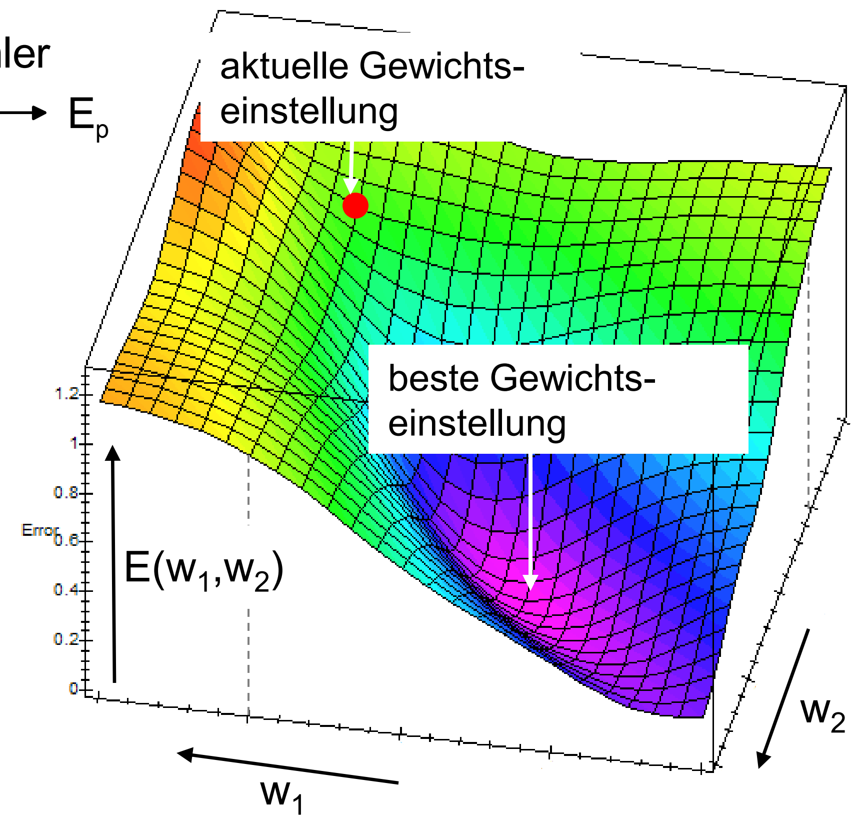


Ziel: Die Gewichte w_1 und w_2 so einstellen, dass die Differenz zwischen gewünschtem Ausgangssignal t_{pj} und tatsächlichem Ausgangssignal o_{pj} über alle p minimal wird.

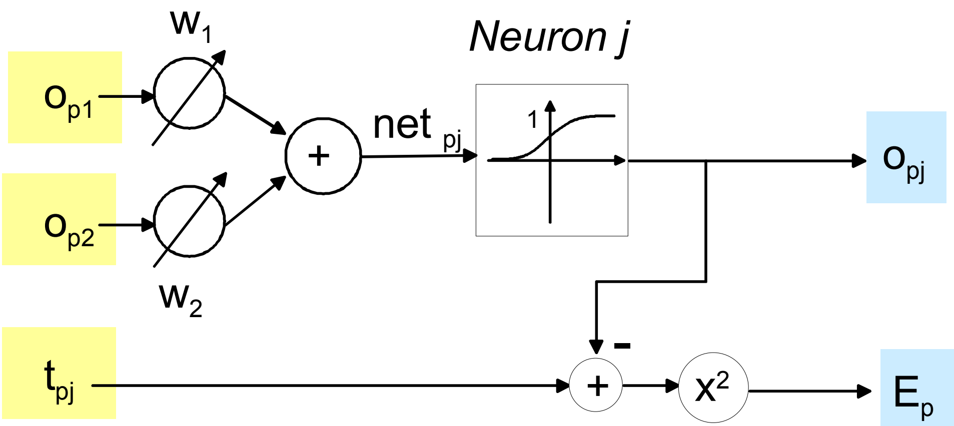
Soll Ist

	vorgegeben			gemessen	
p	o_{p1}	o_{p2}	t_{pj}	o_{pj}	E_p
1	1.0	1.0	0.5	0.5	0.0
2	0.0	1.0	0.3	0.9	0.36
3	1.0	0.0	0.7	0.1	0.36
4	1.0	0.7	0.4	0.65	0.06

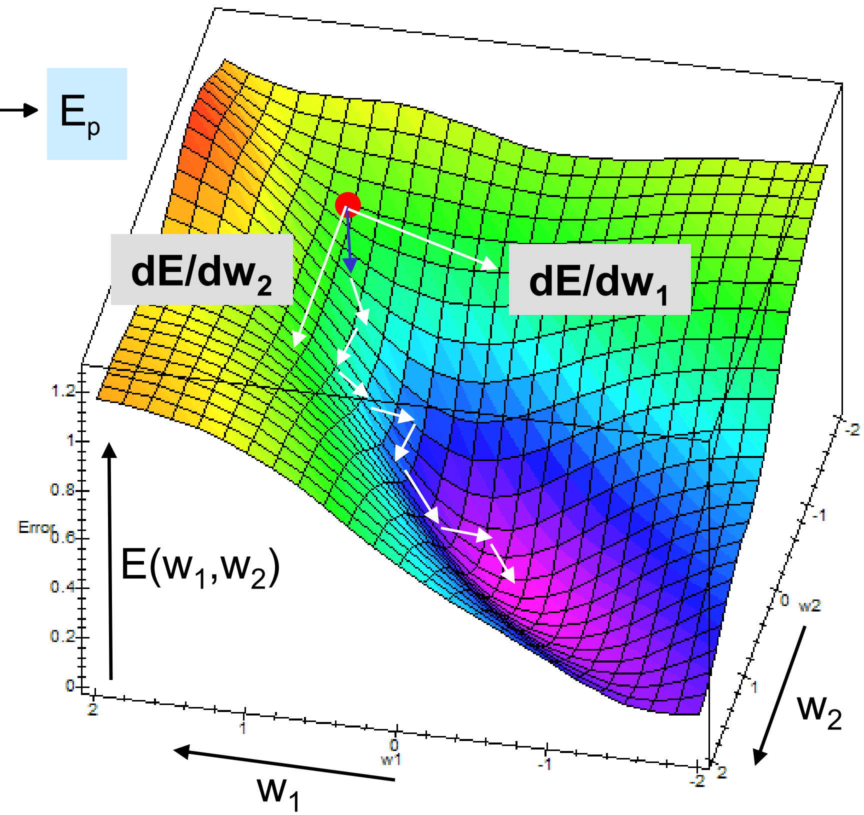
$$E = \sum_p (t_{pj} - o_{pj})^2 = 0.78$$



6.1.4.3 Lösungsidee



Ansatz: Ausgehend von den aktuellen Werten für w_1 und w_2 wird pro Trainingsmuster t_p ein kleiner Schritt in Richtung des steilsten Abstiegs (neg. Gradient) in der Fehlerfunktion E_p vorgenommen.



Gradient des Fehlergebirges :

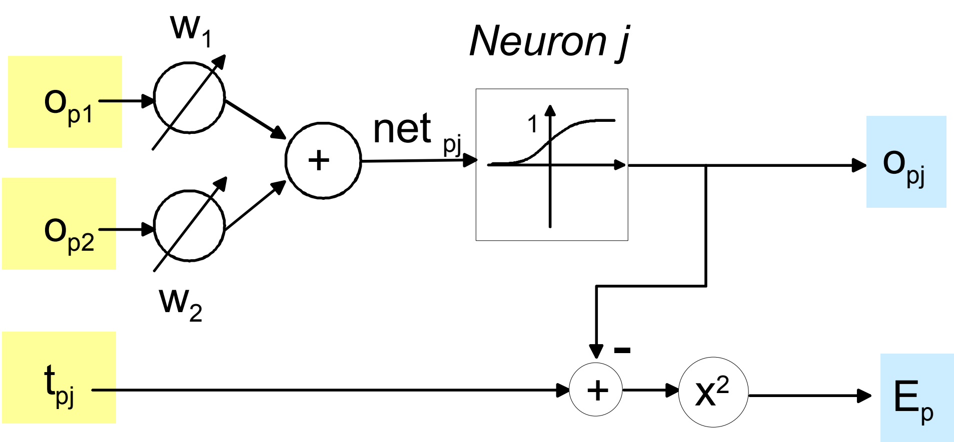
$$\vec{\nabla} E_p(w_1, w_2) = \left(\frac{\partial E_p}{\partial w_1}, \quad \frac{\partial E_p}{\partial w_2} \right)^T$$

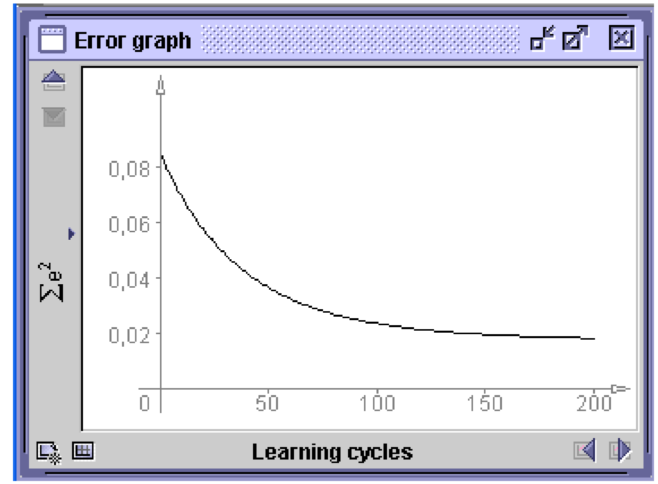
Gewichtsverbesserung:

$$\vec{w}_{n+1} = \vec{w}_n - \eta \cdot \vec{\nabla} E$$

η : Schrittweitenfaktor

6.1.4.4 Backpropagation-Lernalgorithmus





mit den Zusammenhängen :

$$E_p = E_p\left(o_{pj}\right) = (t_{pj} - o_{pj})^2$$

$$o_{pj} = o_{pj}\left(net_{pj}\right) = f_{act}\left(net_{pj}\right) \qquad net_{pj} = o_{p1}w_1 + o_{p2}w_2$$

erhält man mit der Kettenregel die Komponenten des Gradienten

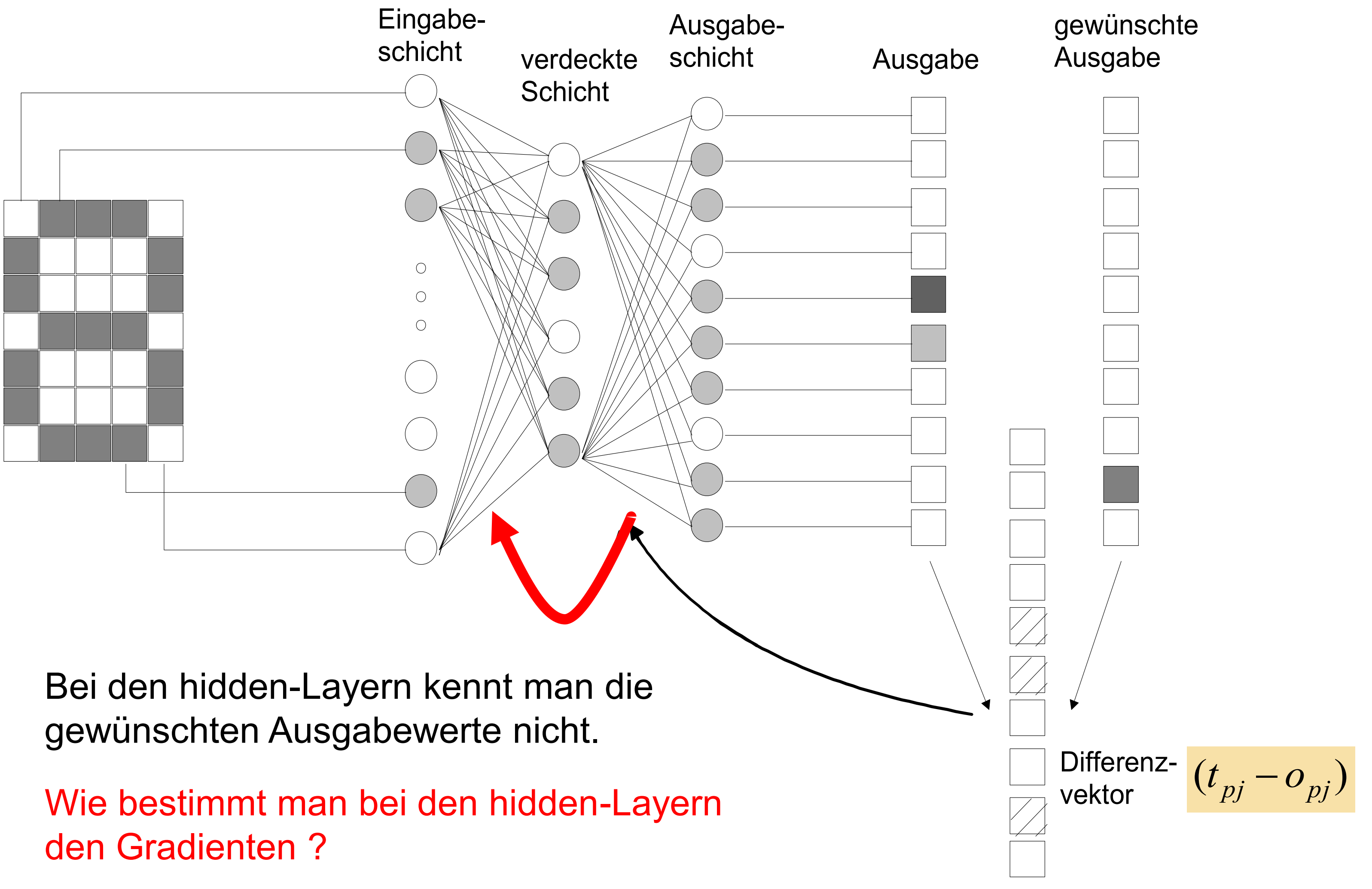
$$\frac{\partial E_p}{\partial w_i} = \frac{\partial E_p}{\partial o_{pj}} \cdot \frac{\partial o_{pj}}{\partial net_{pj}} \cdot \frac{\partial net_{pj}}{\partial w_i} = -2(t_{pj} - o_{pj}) \cdot f'(net_{pj}) \cdot o_{pi}$$

Zusammengefasst:
Backpropagation-Algorithmus für einen Ausgang der Ausgabeschicht:

$$\begin{pmatrix} w_1 \\ w_2 \\ \dots \\ w_m \end{pmatrix}_{n+1} = \begin{pmatrix} w_1 \\ w_2 \\ \dots \\ w_m \end{pmatrix}_{n} + \eta \cdot 2(t_{pj} - o_{pj}) \cdot f'(net_{pj}) \cdot \begin{pmatrix} o_{p1} \\ o_{p2} \\ \dots \\ o_{pm} \end{pmatrix}$$

mit $net_{pj} = o_{p1} \cdot w_1 + o_{p2} \cdot w_2 + \dots\dots\dots + o_{pm} \cdot w_m$

6.1.4.5 Backpropagation für das Multilayer-Perzeptron (MLP)

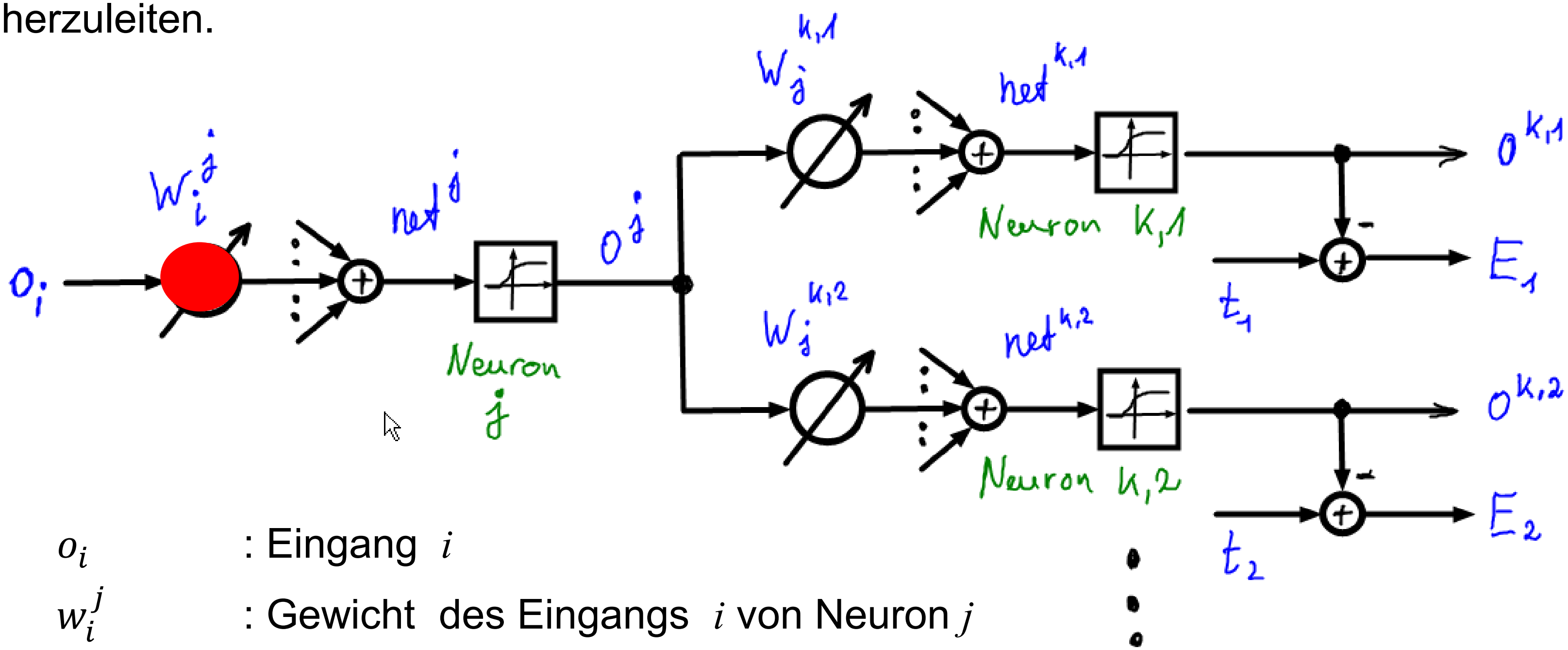


Bei den hidden-Layern kennt man die gewünschten Ausgabewerte nicht.

Wie bestimmt man bei den hidden-Layern den Gradienten ?

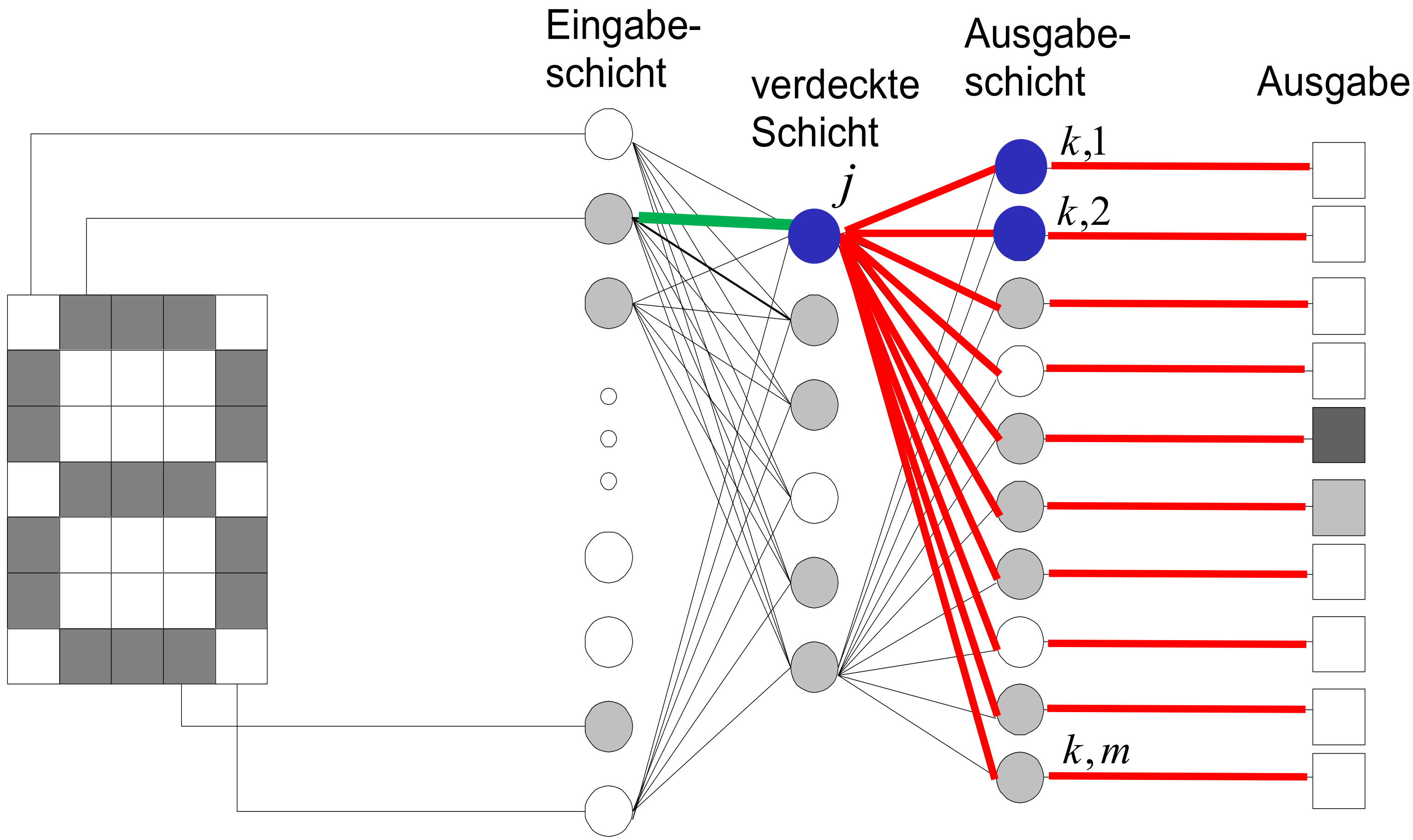
ÜBUNG: Training des hidden Layers des zweilagigen Perzeptrons

Der Backpropagation-Trainingsalgorithmus für ein zweilagiges Perzeptron ist herzuleiten.



o_i	: Eingang i
w_i^j	: Gewicht des Eingangs i von Neuron j
o^j	: Ausgang des Neuron j
$w_j^{k,1}$	: Gewicht des mit Neuron j verbundenen Eingangs von Neuron $k,1$
t_1	: gewünschter Output von Ausgang 1
$o^{k,1}$	: tatsächlicher Output des Neuron $k,1$
E_1	: Fehlerquadrat Ausgang 1

ÜBUNG: ...... Fortsetzung



Wie hängt der Fehler der Ausgabe von **diesem** Gewicht ab?

6.1.4.6 Backpropagation-Lernalgorithmus : *incrementelles Verfahren*

Initialisiere alle Gewichte mit Zufallswerten, Fehler E = sehr großer Wert
while (Fehler E > vorgegebene Fehlergrenze)
    Lege ein zufällig gewähltes Eingabemuster an
    for each Ausgabeneuron
        Modifiziere die Gewichte mit "*BP-Regel für Ausgabeneuronen*"
    endfor
    for each Verdecktes Neuron
        Modifiziere die Gewichte mit "*BP-Regel für verdeckte Neuronen*"
    endfor
    Berechne den Gesamtfehler E
endwhile

Alternative.: ***Batch-Verfahren***

1. Es wird der Fehler über alle Trainingsmuster bestimmt.
2. Erst dann wird die Modifikation der Gewichte vorgenommen.

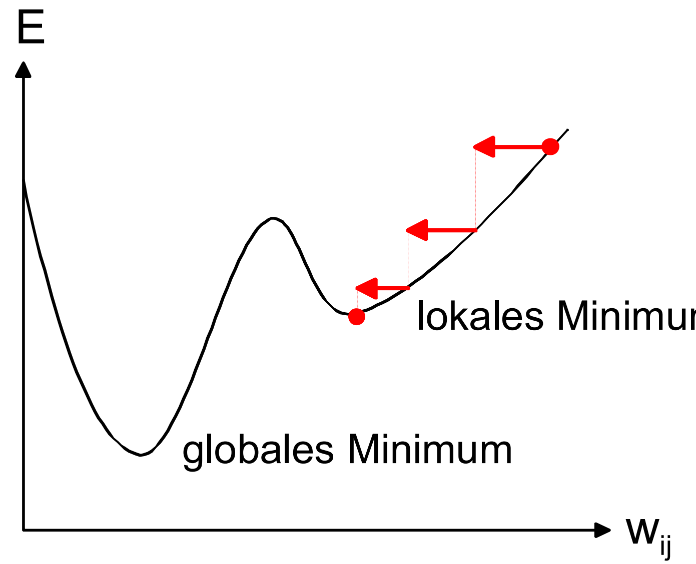
6.1.4.7. Probleme des Backpropagation

Lokale Minima in der Fehlerfunktion

Schrittweite ist abhängig von:

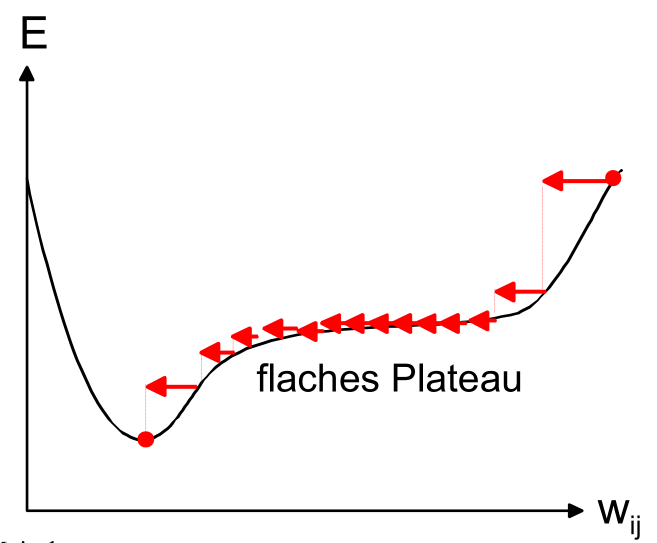
- Schrittweitenfaktor η (const.)
- lokalem Gradient

→ in Minima ist die Schrittweite = 0

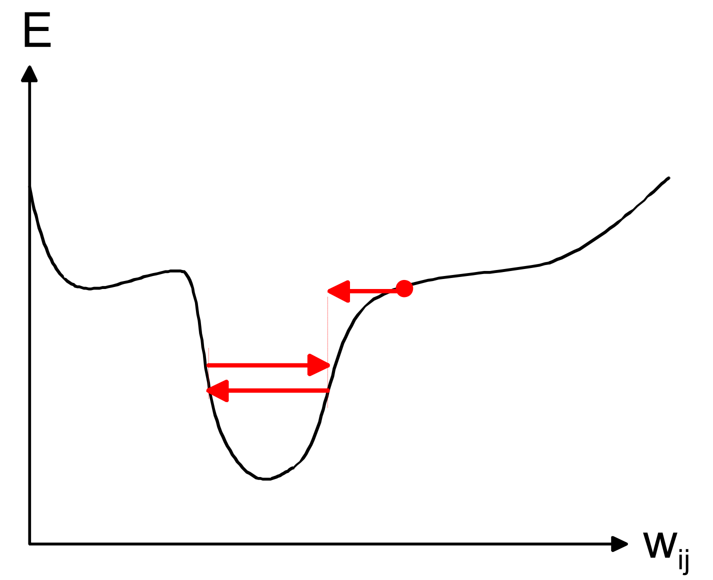


Plateaus in der Fehlerfunktion

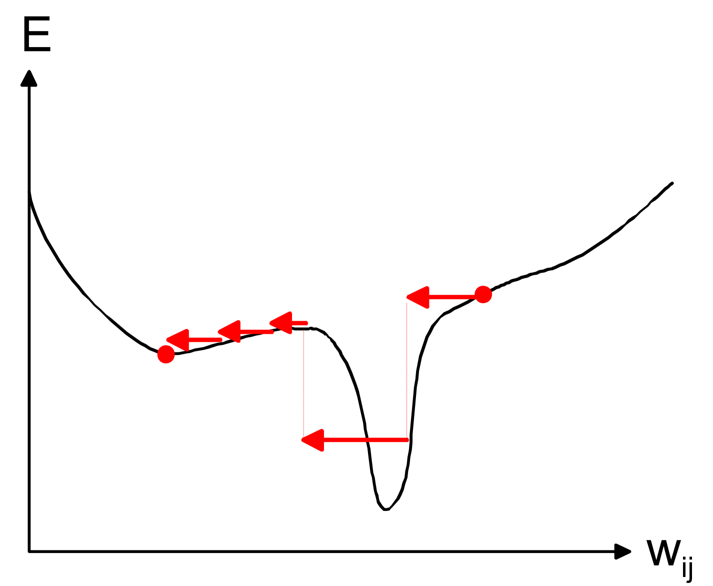
→ in flachen Funktionsbereichen ist die Schrittweite sehr klein



Oszillation in steilen Bereichen



Aussprung aus steilen Minima



6.1.4.8 Modifikationen des Backpropagation-Lernverfahrens

Momentum-Term

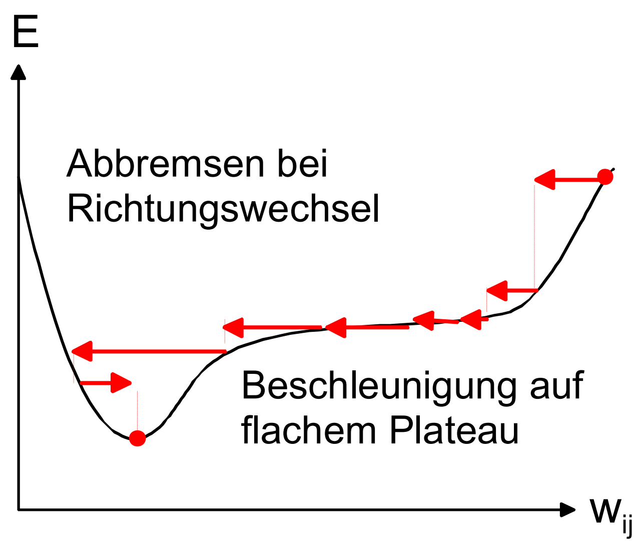
Zweck: Problemminderung bei
- Plateaus,
- und steilen Schluchten (Oszillation).

Ansatz: gewichtete Berücksichtigung (α) der vorherigen Schrittweite

$$\Delta w_{ij}(t+1) = \eta \cdot \delta_j \cdot o_i + \alpha \cdot \Delta w_{ij}(t)$$

Momentum-Term

Wirkung:
- Beschleunigung auf Plateaus
- Abbremsen bei Wechsel der Gradientenrichtung

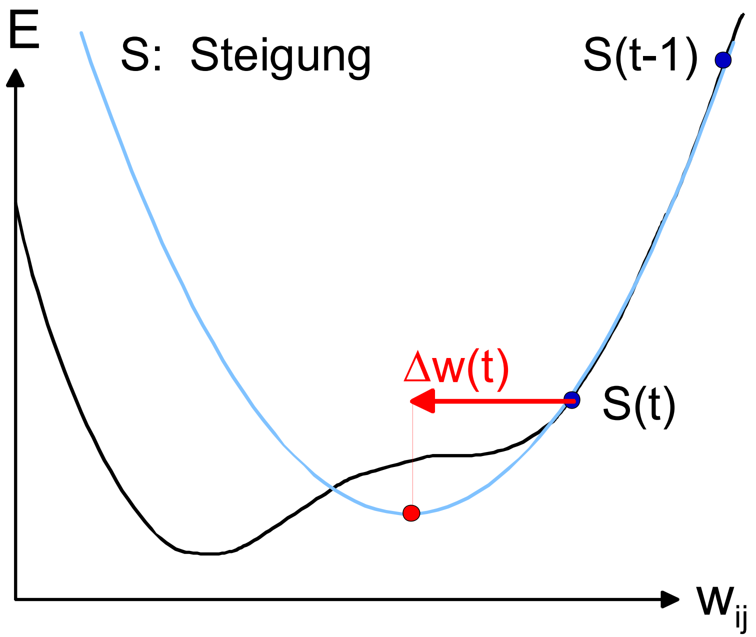


Quickprop

Zweck: Schnelleres Training

Ansatz: E(w) wird näherungsweise als Parabel angenommen.

1. Die Steigungen zweier aufeinanderfolgender Schritte werden berechnet.
2. In die beiden Steigungen wird eine Parabel eingepasst (Interpolation).
3. Der Scheitelpunkt der Parabel ist das Ergebnis dieses Iterationsschritts.



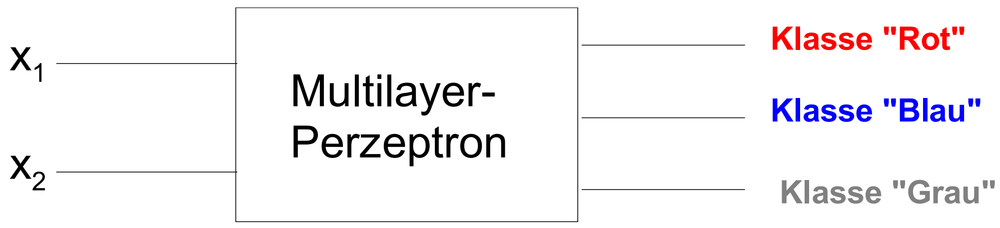
SuperSAB: (*Fast Adaptive Backpropagation*)

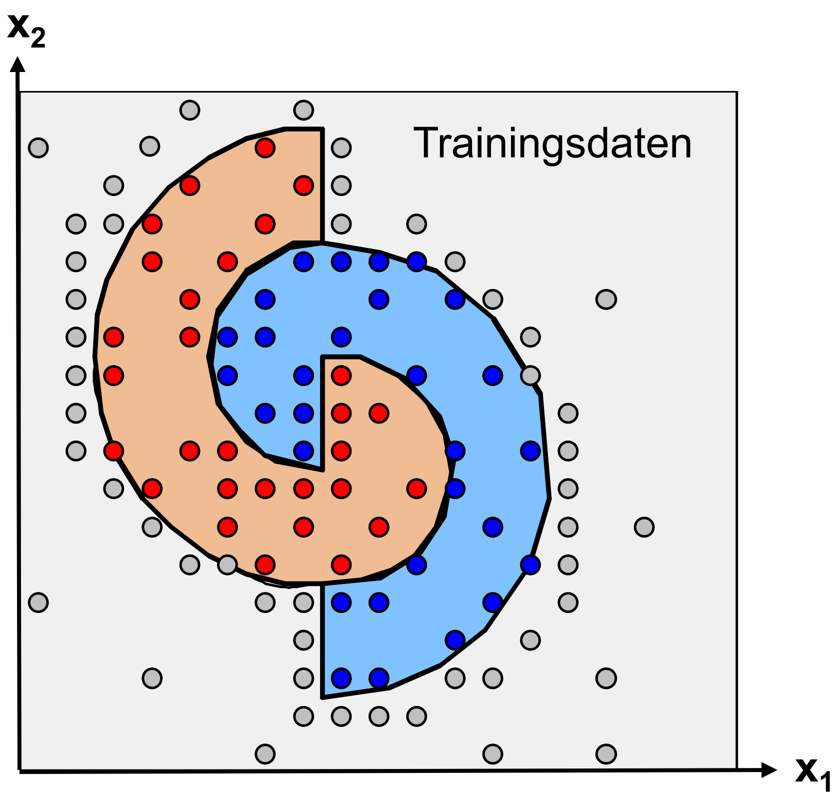
Die Schrittweitenfaktoren η_{ik} werden für jedes Gewicht individuell berechnet.

- η_{ik} wird größer, wenn sich die Richtung über mehrere Schritte nicht ändert.
- η_{ik} wird kleiner, wenn sich die Richtung ändert.

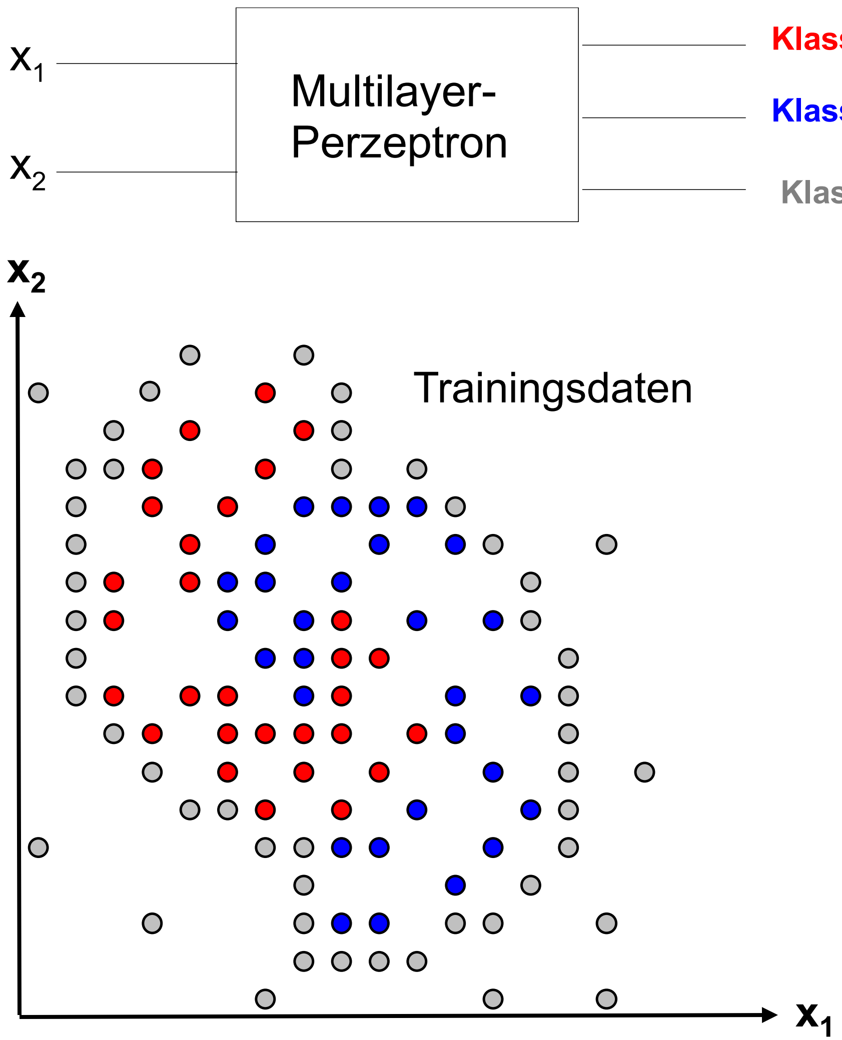
Resilient Propagation: Kombination aus SuperSAB und Quickprop.

6.1.4.9 Beispiel: Spiralproblem

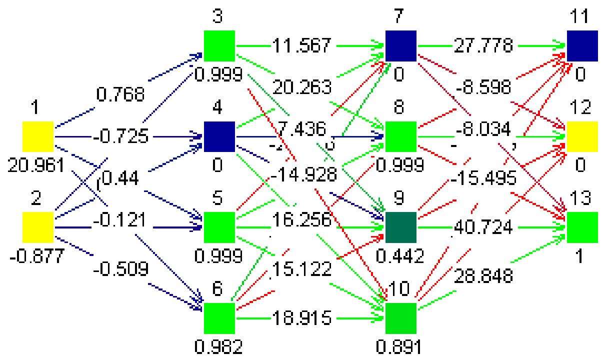


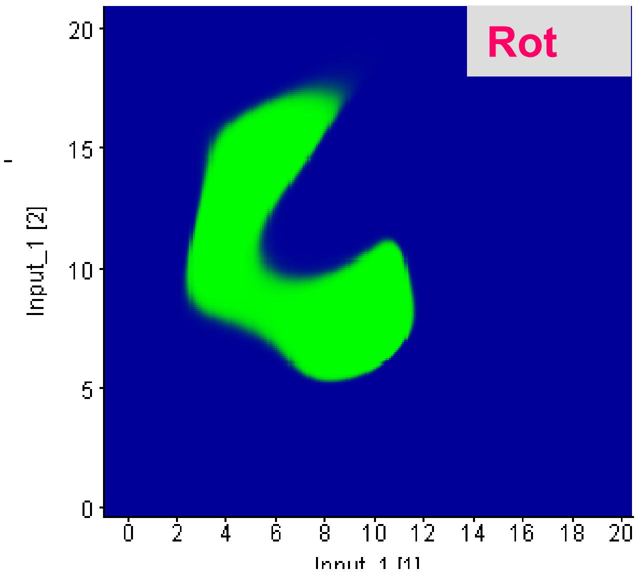



```
No. of patterns : 106
No. of input units : 2
No. of output units : 3

# Input pattern 1:
7 17
# Output pattern 1:
1 0 0
# Input pattern 2:
9 3
# Output pattern 2:
0 1 0
# Input pattern 3:
3 12
# Output pattern 3:
1 0 0
# Input pattern 4:
14 9
# Output pattern 4:
0 1 0
........
```

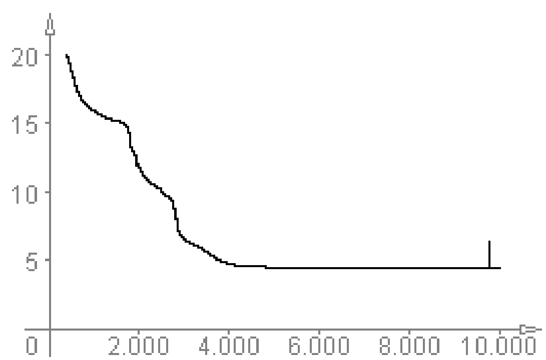

x1
x2
Multilayer-
Perzeptron
Klasse "Rot"
Klasse "Blau"
Klasse "Grau"
x2
Trainingsdaten
x1

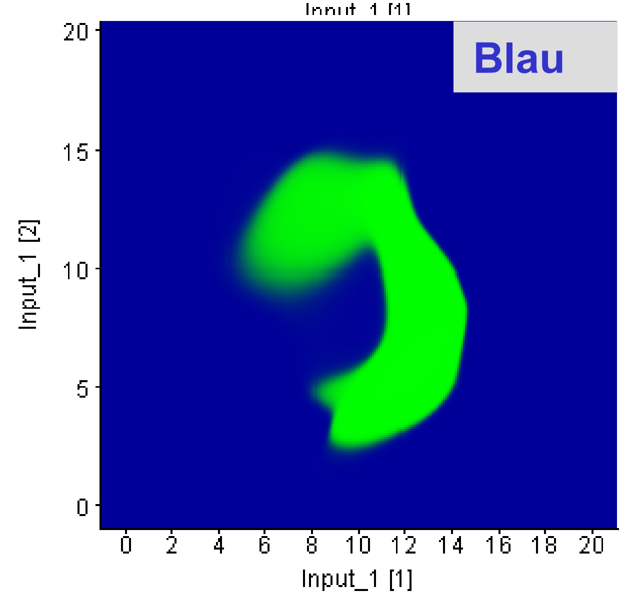
Netz a: 2 – 4 – 4 – 3, Resilient-Propagation



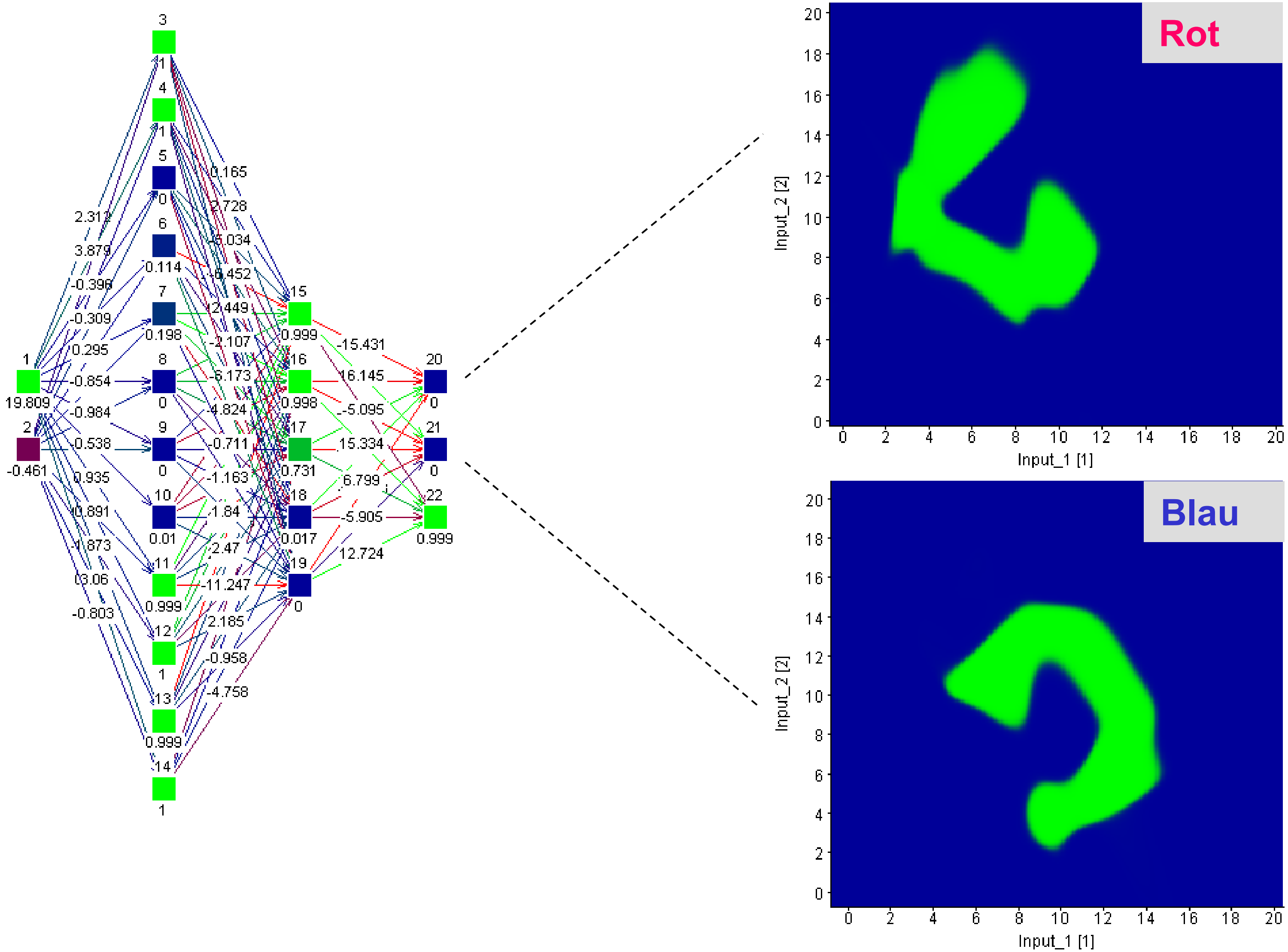


Fehlerverlauf beim Training

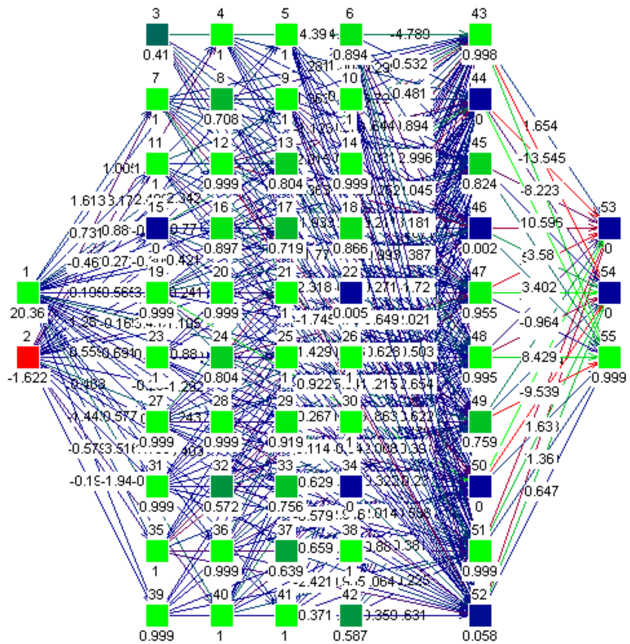


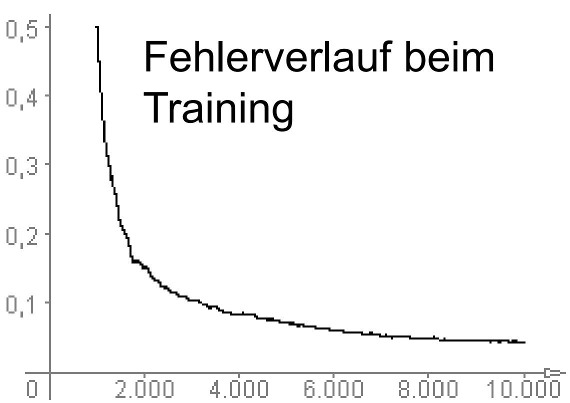


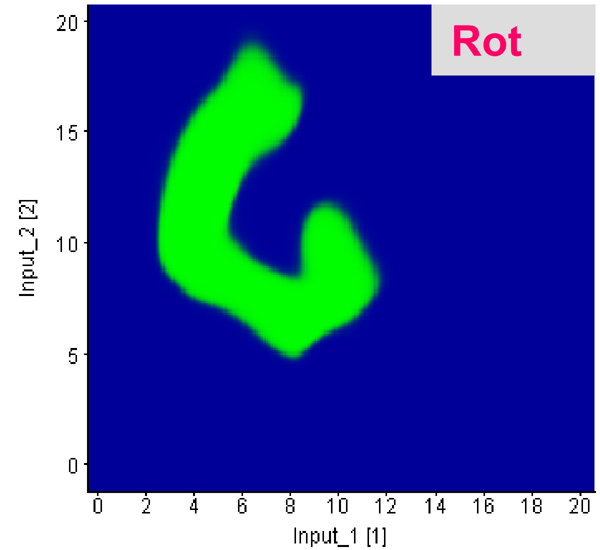
Netz b: 2 – 12 – 5 – 3 , Resilient-Propagation

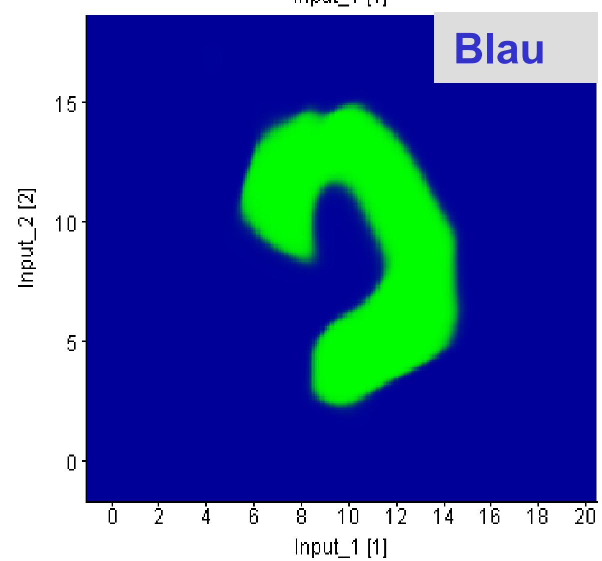


Netz c: 2 – 20 – 10 – 3, Resilient-Propagation

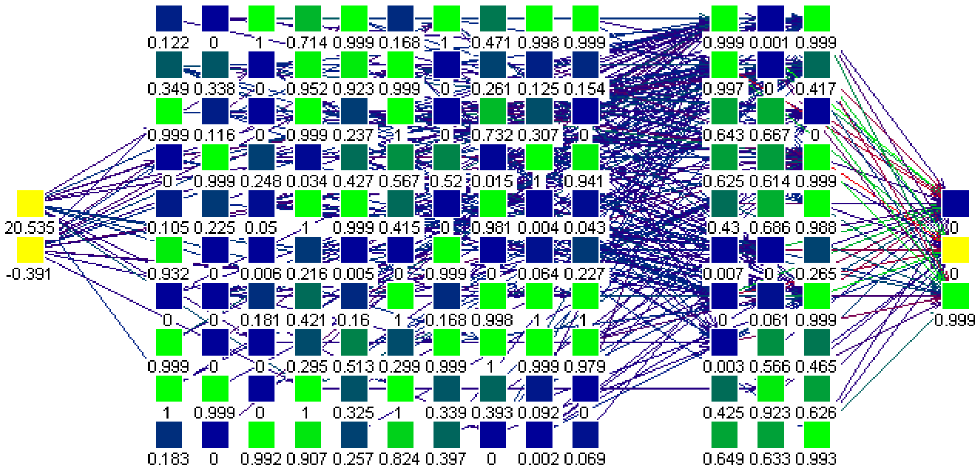


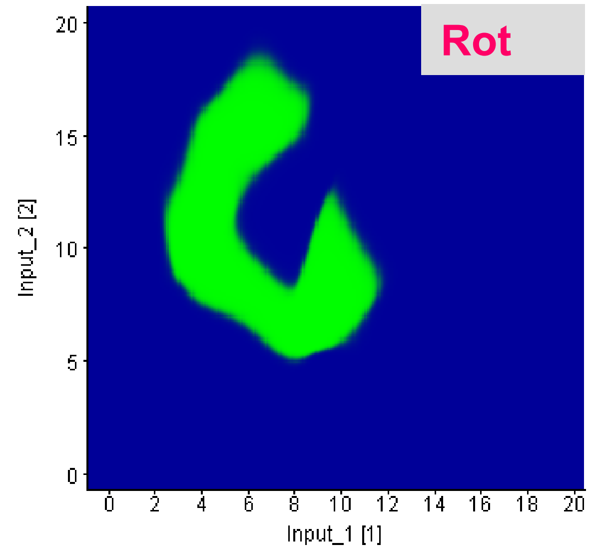




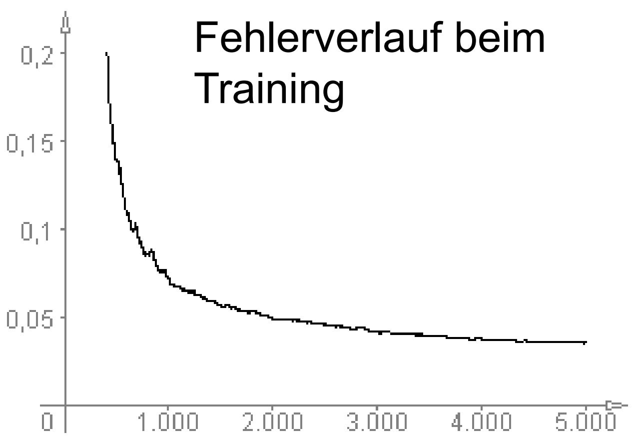


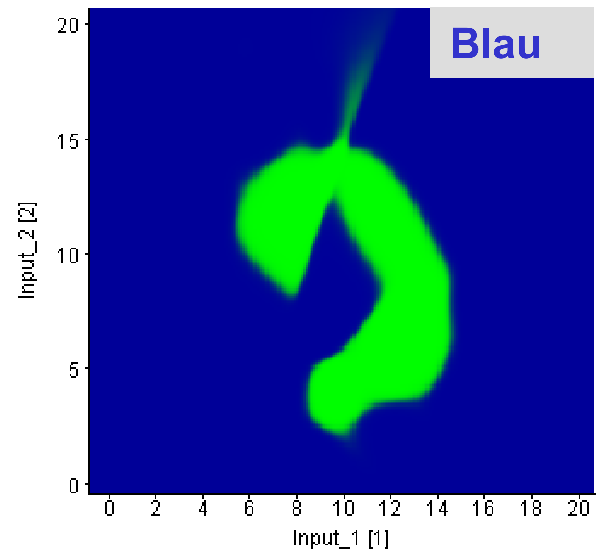
Netz c: 2 – 100 – 30 – 3, Resilient-Propagation





Fehlerverlauf beim Training

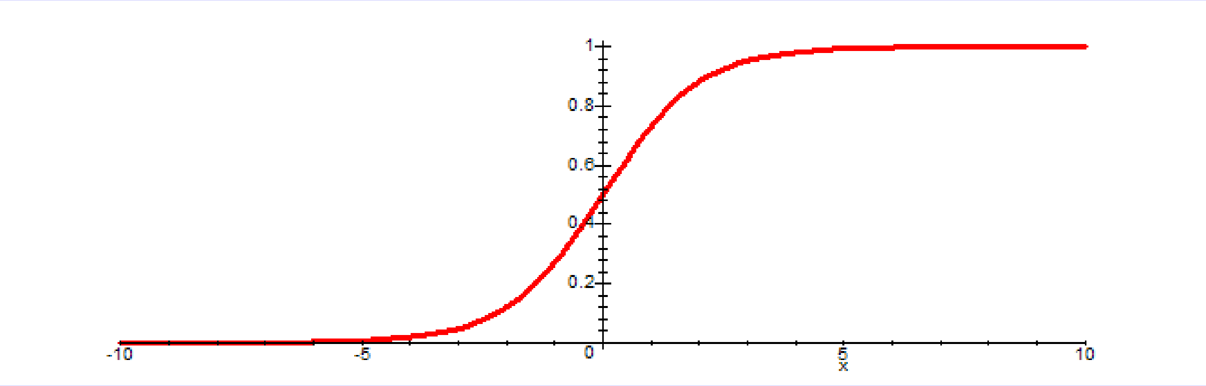




6.1.5 Maßnahmen für ein schnelleres Training

1. **Die gewünschten Ausgangswerte (Trainingswerte) müssen im Wertebereich der Ausgangsaktivierungsfunktion liegen !**

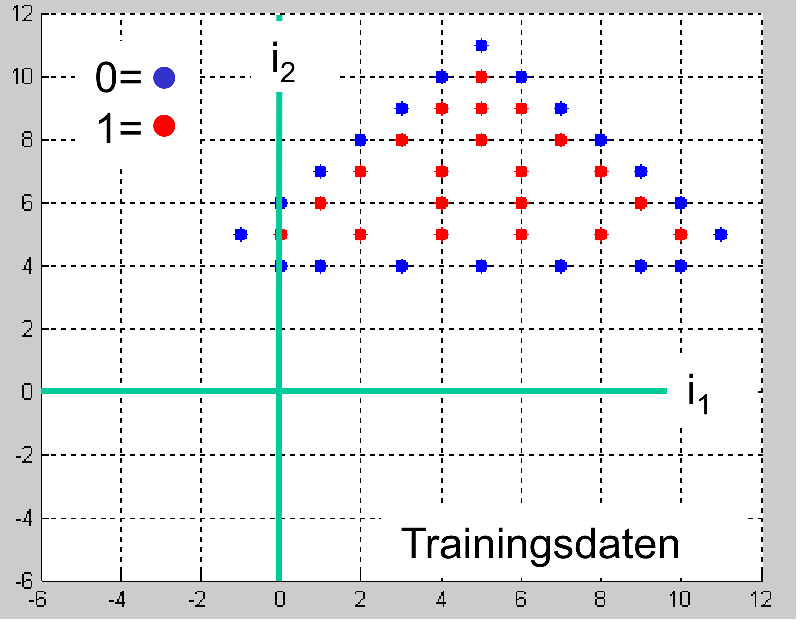
Beispiel: Aktivierungsfunktion der Ausgangsneuronen sei die logistische Funktion

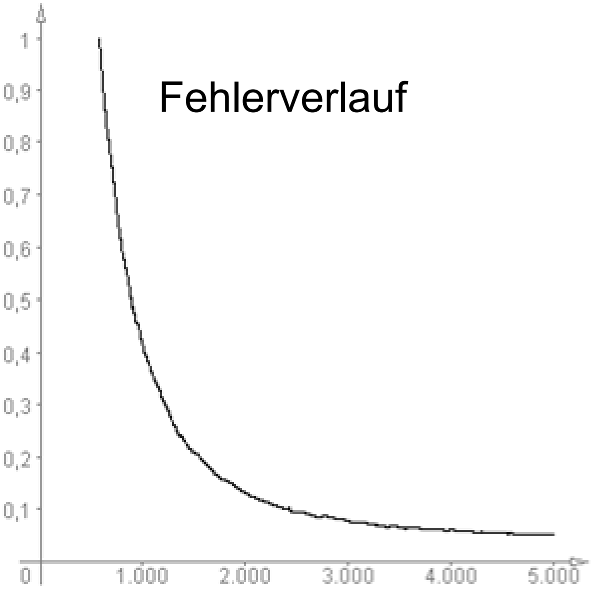


→ Die Trainingswerte müssen im Intervall (0, 1) liegen, z.B. 0.05 (false) und 0.95 (true).

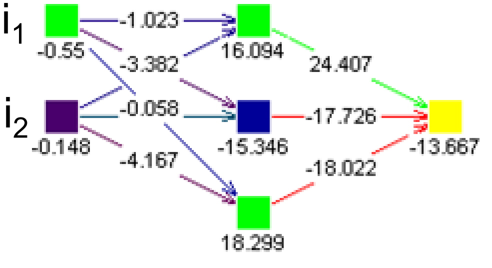
2. **Der Mittelwert der Trainingswerte sollte möglichst Null sein !**

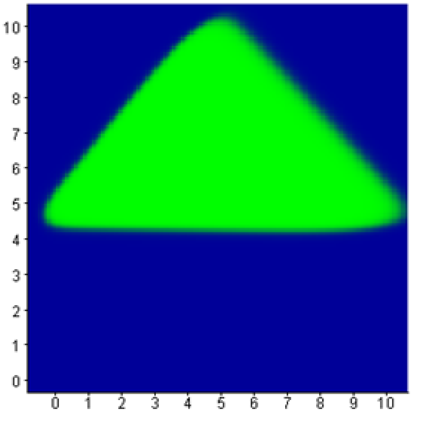
Fall 1 : Eingangswerte sind nicht mittelwertfrei





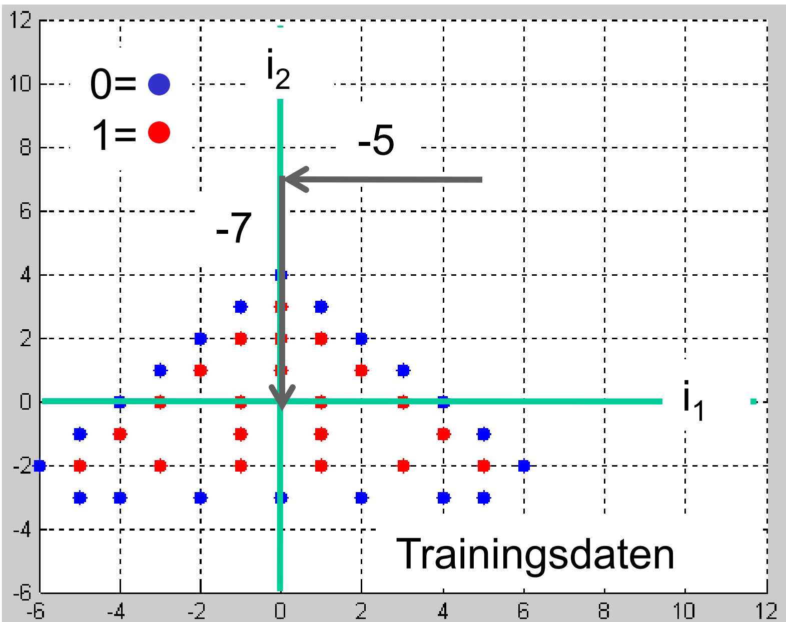
Netz-
topologie

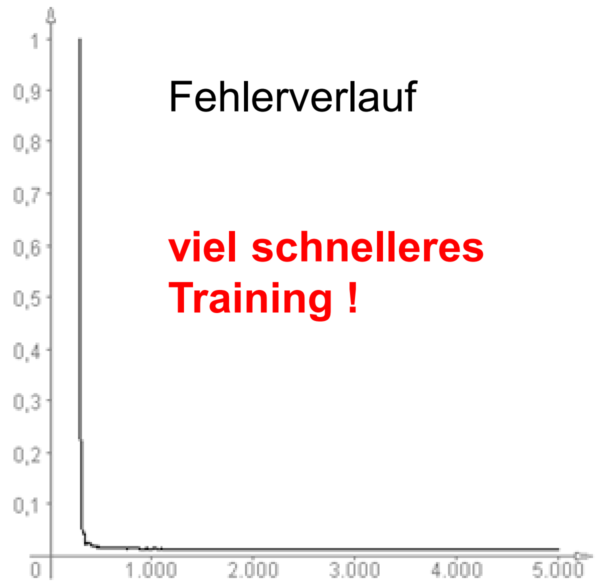




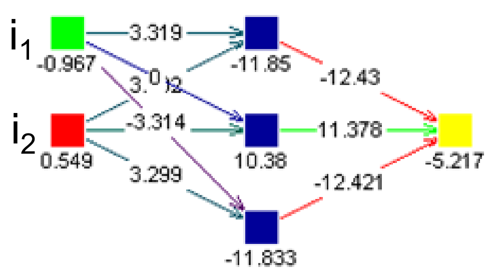
Netz-
ausgabe

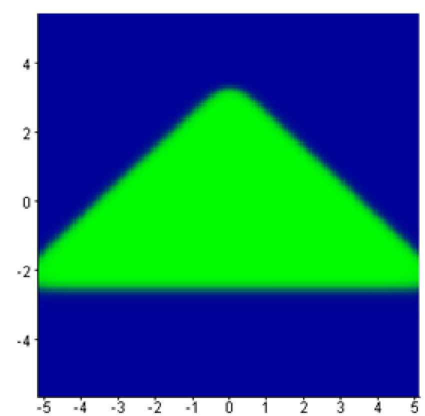
Fall 2 : Eingangswerte sind <u>nahezu</u> mittelwertfrei





Netz-
topologie





Netz-
ausgabe

6.1.6 Praktische Vorgehensweise beim Netzentwurf

Festlegung der Hidden-Neuronen-Anzahl und anderer Parameter

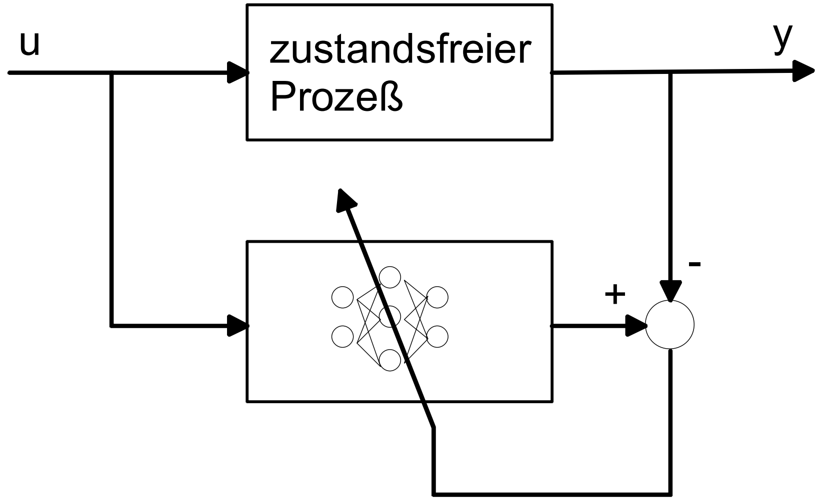
- "Der Entwurf eines Neuronalen Netzes ist mehr *Kunst* als Wissenschaft."
- Die Zahl der Trainingsmuster sollte mind. 10x der Anzahl der Gewichte betragen.
- Zu viele Trainingszyklen können zu einer Überanpassung ("*overfitting*") des Netzes an die Trainingsdaten führen (schlechte Generalisierung).
- Die Suche nach den optimalen Trainings- und Netzparametern erfolgt am besten durch Vergleich mehrerer Varianten und Verfeinerung.

Überprüfung der korrekten Funktionsweise

- Ein kleiner Restfehler bedeutet nur, dass die Trainingsdatenmenge gut erkannt wird, aber nicht zwangsläufig, dass das NN gute Generalisierungseigenschaften hat.
- Die Generalisierungsfähigkeit des trainierten NN sollte mit einer unabhängigen Testdatenmenge (ca. 10-30% der Trainingsdatenmenge) überprüft werden.

6.1.7 Andere Trainingssituationen

Beispiel: Erlernen von Prozeßkennfeldern y=f(u)



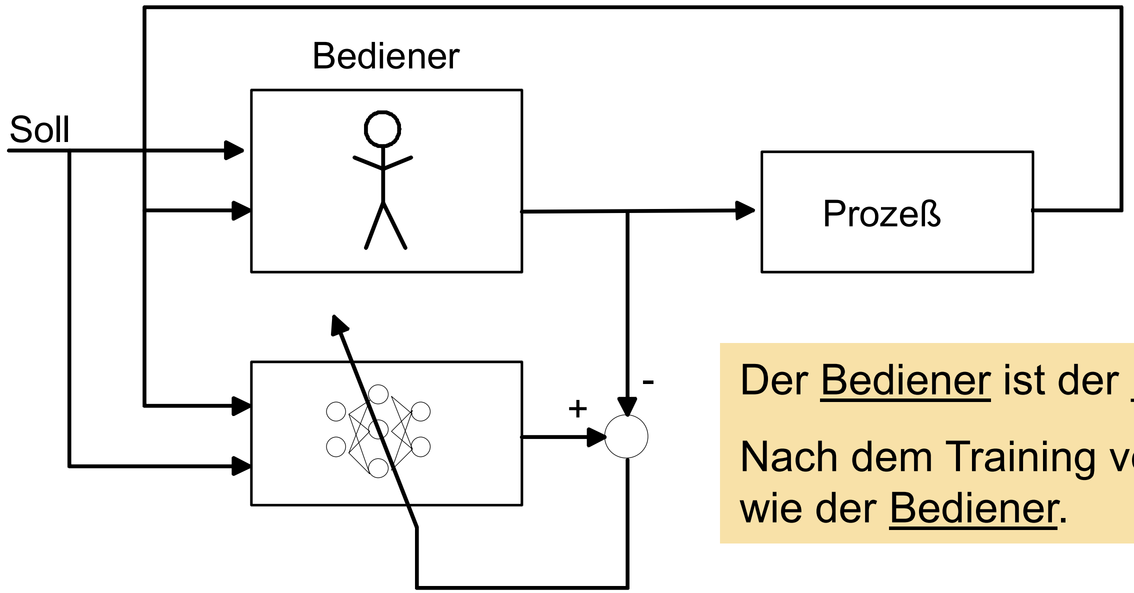
Der Prozeß selbst ist der Lehrer.

Nach dem Training verhält sich das NN wie der Prozess.

Anwendungen:

- Prozeßsimulation technischer Systeme (z.B. Kunststoff-Spritzgussanlage)
- Kennfeldermittlung (z.B. für Verbrennungsmotor)
- Verhaltenssimulation komplexer Systeme (z.B. Konsumentenverhalten)
- Prognosesysteme
- prädiktive Regelungen

Beispiel: Erlernen von Bedienerverhalten

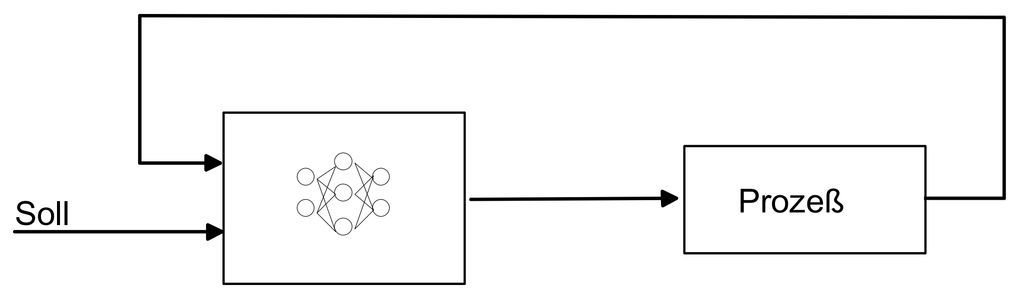


Der Bediener ist der Lehrer/Trainer.

Nach dem Training verhält sich das NN wie der Bediener.

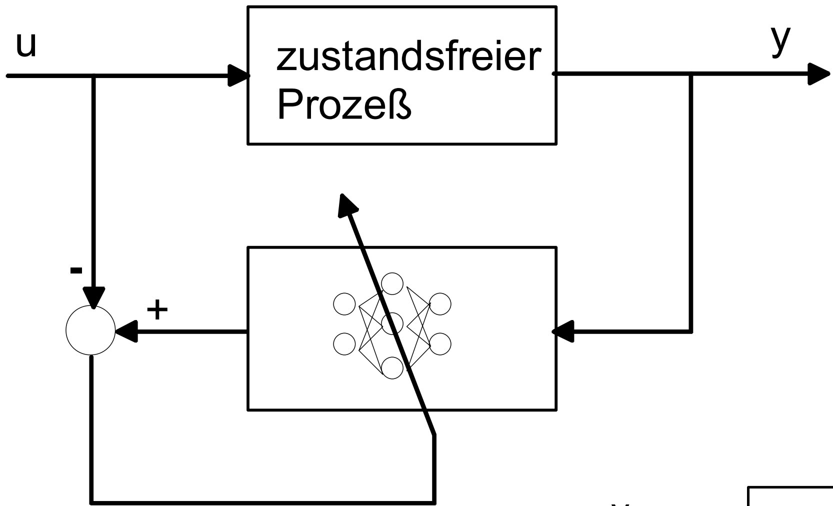
Anwendungen:

- adaptive Regelungen
- Assistenzsysteme (z.B. Einparkassistent)
- Verhaltenstraining künstl. Wesen in Computerspielen
- Agentenmodellierung in Simulationssystemen



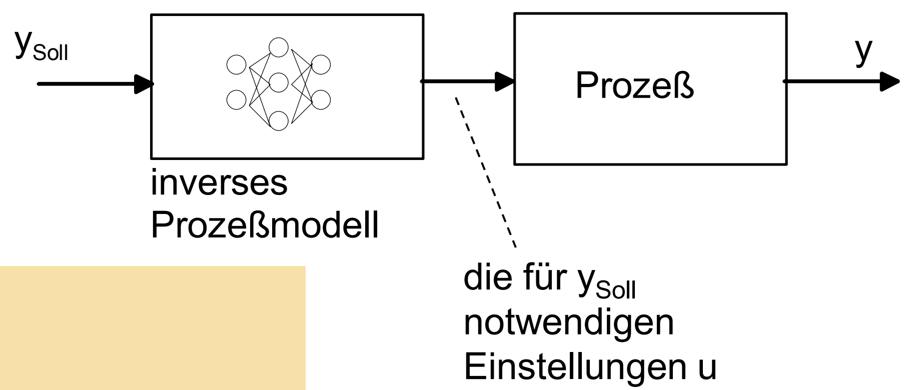
NN mit Bediener-verhalten

Beispiel: *Erlernen des inversen Prozeßkennfeldes u=f'(y)*



Der Prozeß selbst ist der Lehrer.

Nach dem Training verhält sich das NN wie der <u>inverse Prozess</u>.

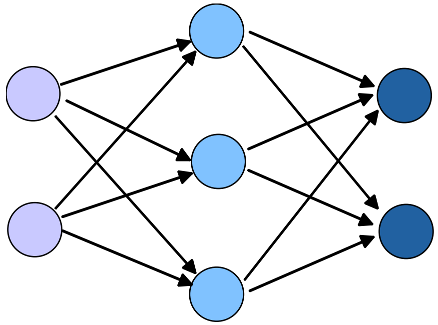


Anwendungen:

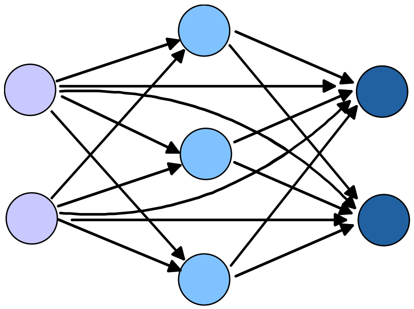
- Steuerungen
- inverse Roboterkinematik (kart. Koordinaten → Gelenkwinkel)

6.2 Andere Netztopologien und Lernstrategien

6.2.1 Netze ohne Rückkopplung



"Feedforward"



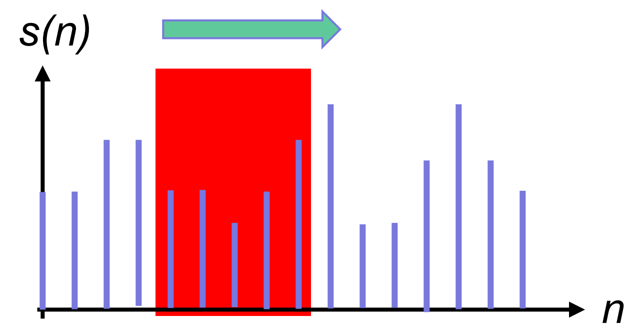
"Feedforward, shortcut connections"

Netze ohne Rückkopplung (= zustandsfreie Netze)

Die Ausgangssignale sind nur von den Eingangssignalen abhängig.

$$\boxed{\vec{y} = \vec{f}\,(\vec{x})}$$

6.2.2 Netze für zeitveränderliche Muster



<u>*6.2.2.1 Sliding-Window-Verfahren*</u>

Zweck:

Erkennung von Systematiken in Impulsfolgen

Voraussetzung:

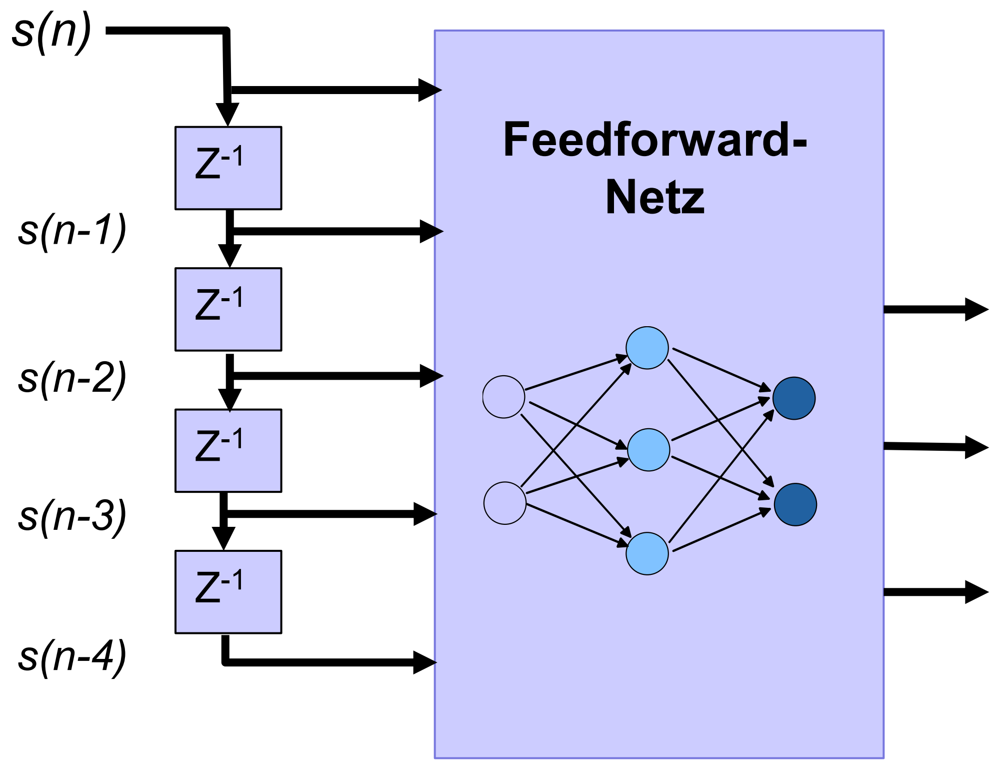
Getaktetes Eingangssignal

Beispiele:

- Prognosesysteme
- Aktien kaufen/halten/verkaufen
- Morsedecoder

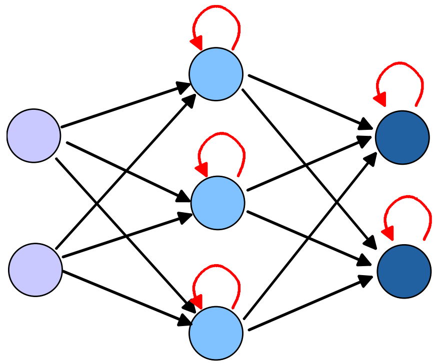
Typ. Eigenschaft:

2 gleiche Teilfolgen im Fenster erzeugen exakt dieselbe Ausgabe. (vergl. FIR-Filter)

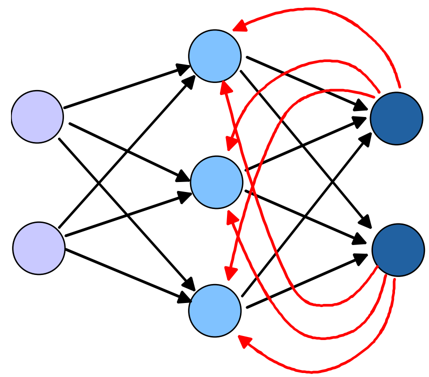


Anm.: z^{-1} = Verzögerung um einen Takt

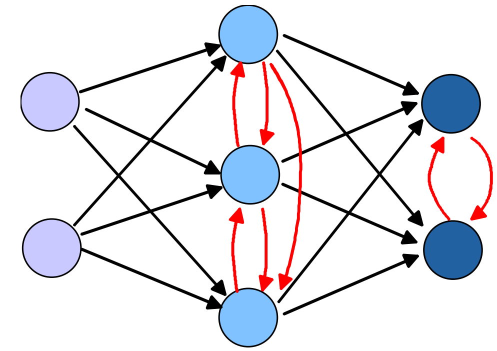
6.2.2.2 Getaktete Netze mit Rückkopplung (zustandsbehaftete Netze)



direkte Rückkopplungen



indirekte Rückkopplungen



laterale Rückkopplungen

Anm.: Die roten Verbindungen sind jeweils um einen Takt verzögert (beinhalten z^{-1}).

Netze mit Rückkopplung (= zustandsbehaftete Netze)

Die Ausgangssignale sind abhängig

- von den Eingangssignalen
- von der zeitlichen Vorgeschichte (gespeichert als innerer Zustand $\vec{z}(n)$)

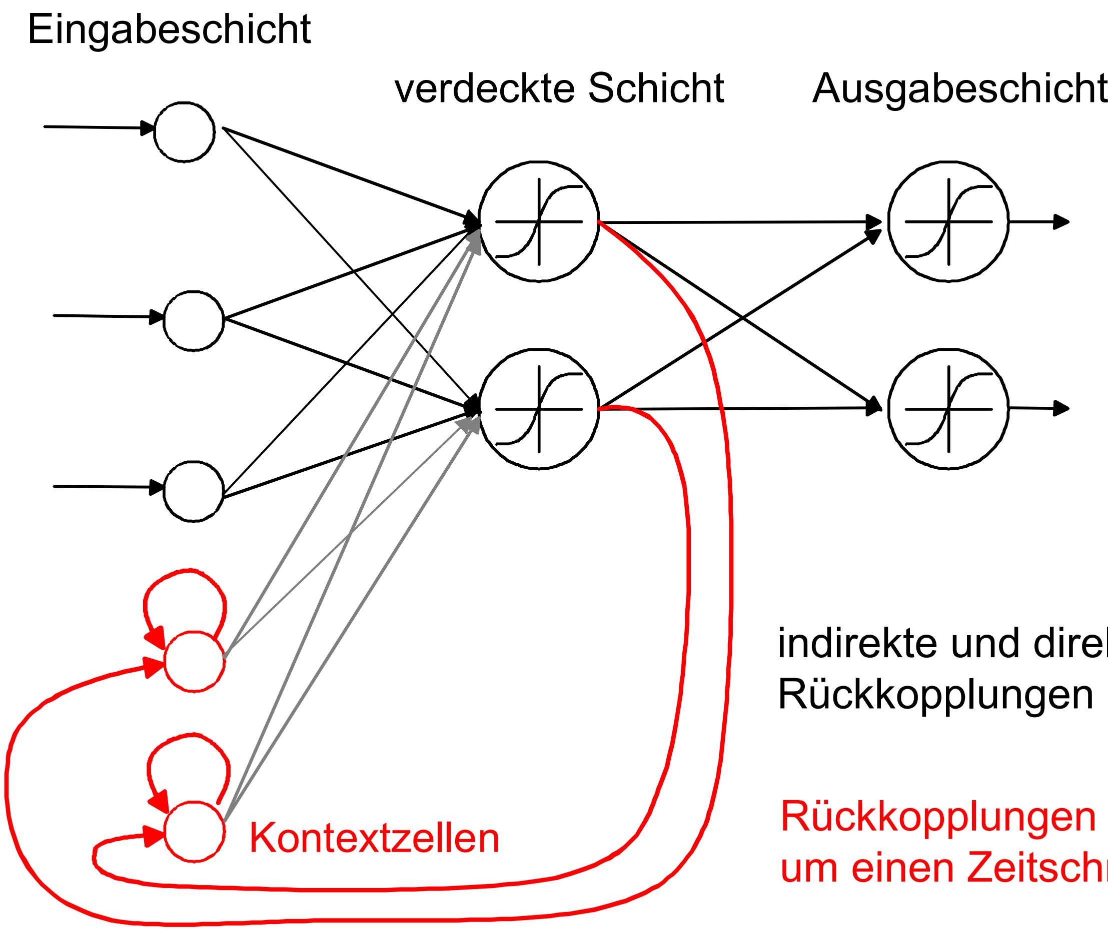
$$\text{Zustandsübergangsfunktion:} \quad \vec{z}(n+1) = \vec{f}\left[\vec{z}(n), \vec{x}(n)\right]$$

$$\text{Ausgabefunktion:} \quad \vec{y}(n) = \vec{g}\left[\vec{z}(n), \vec{x}(n)\right]$$

6.3 Beispiel für ein zustandsbehaftetes NN: Jordan-Elman-Netz

Zweck: Erkennung von Systematiken in Zeitfunktionen

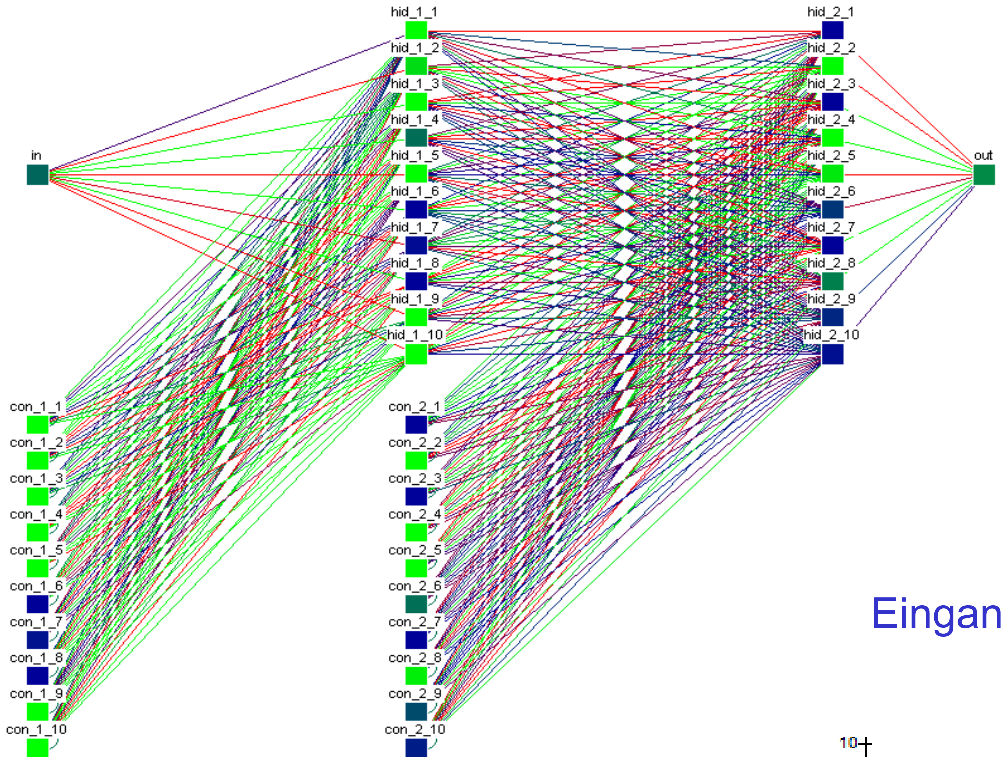
Lernstrategie: "*supervised learning*", ähnlich zu Backpropagation



indirekte und direkte Rückkopplungen

Rückkopplungen sind jeweils um einen Zeitschritt verzögert

Prognose eines zeitlichen Verlaufes

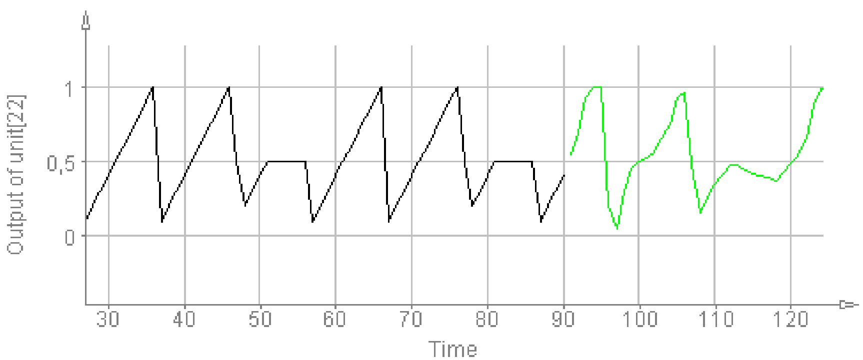


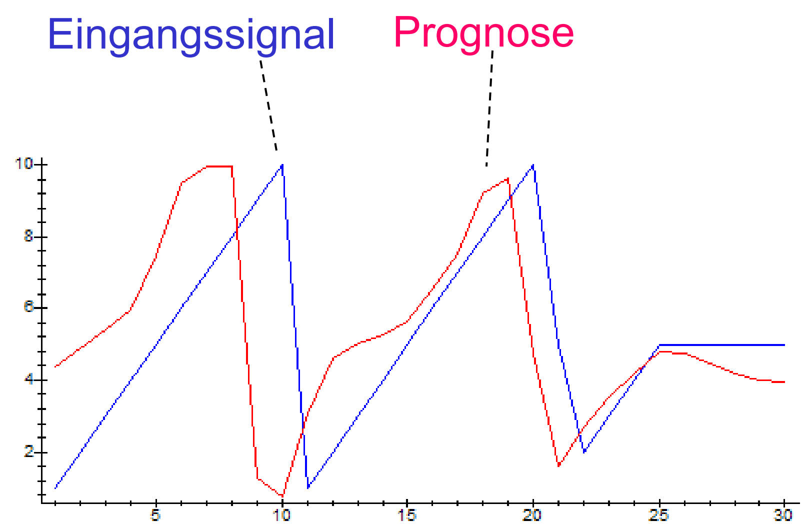
<u>Signale beim Training:</u>

- Eingangssignal x(n)
- Trainingsvorgabe x(n+1)

<u>Trainiertes Netz:</u>

- Eingangssignal x(n)
- Voraussage für x(n+1)





6.4 Kohonen-Netze ("Self-Organizing-Maps„ = SOM)

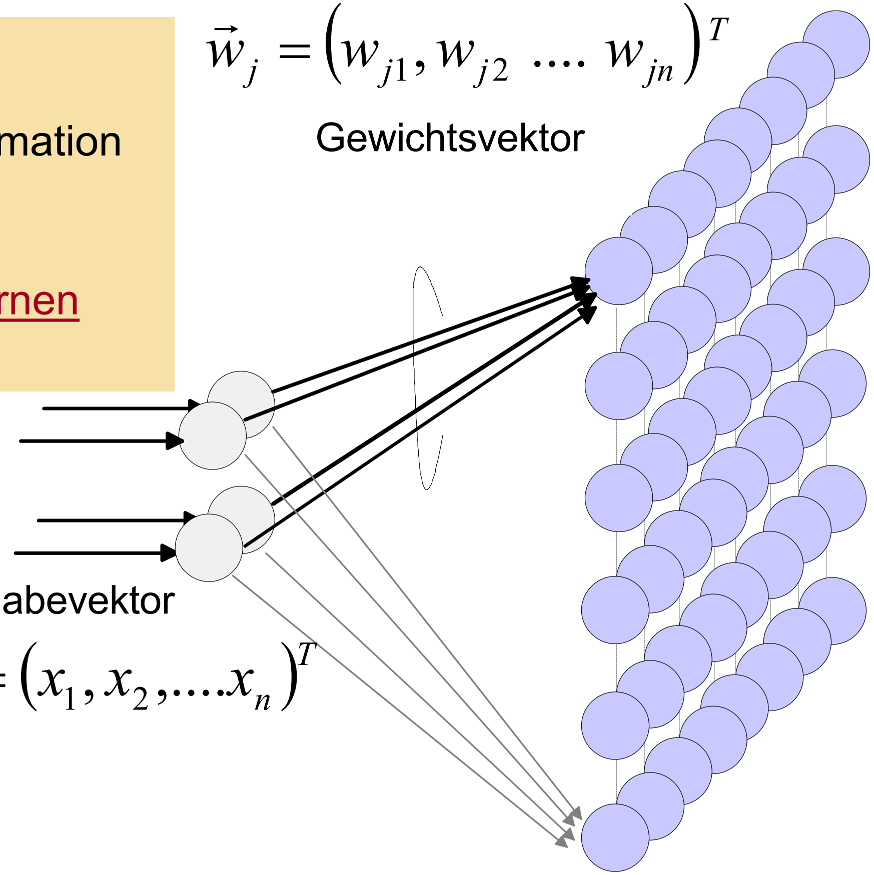
6.4.1 Einordnung und Definitionen

Zweck:
- Optimierung
- Clustering
- Funktionsapproximation

Lernstrategie:
- <u>unüberwachtes Lernen</u>

$$\vec{w}_j = \left(w_{j1}, w_{j2} \; .... \; w_{jn}\right)^T$$

Gewichtsvektor

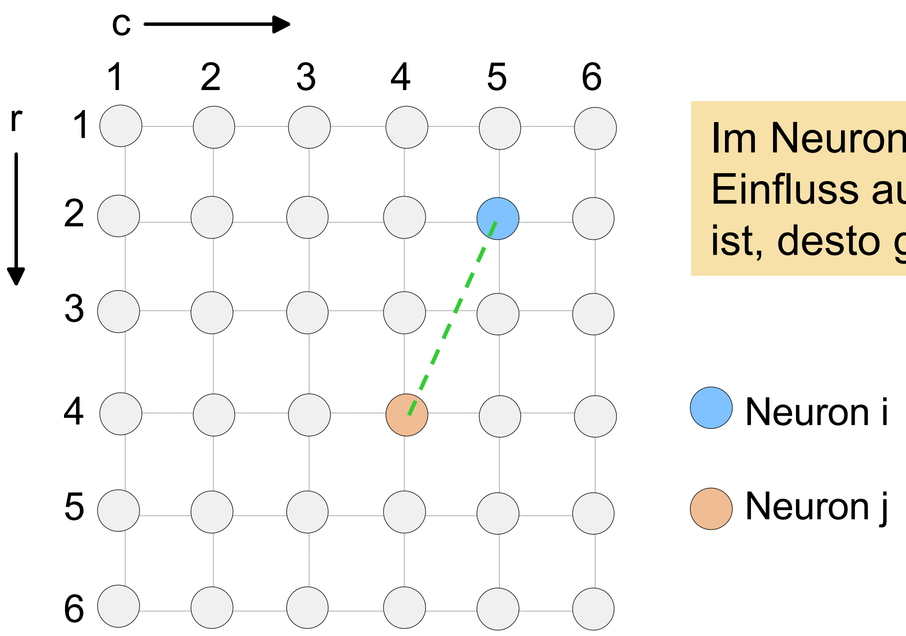


Eingabevektor

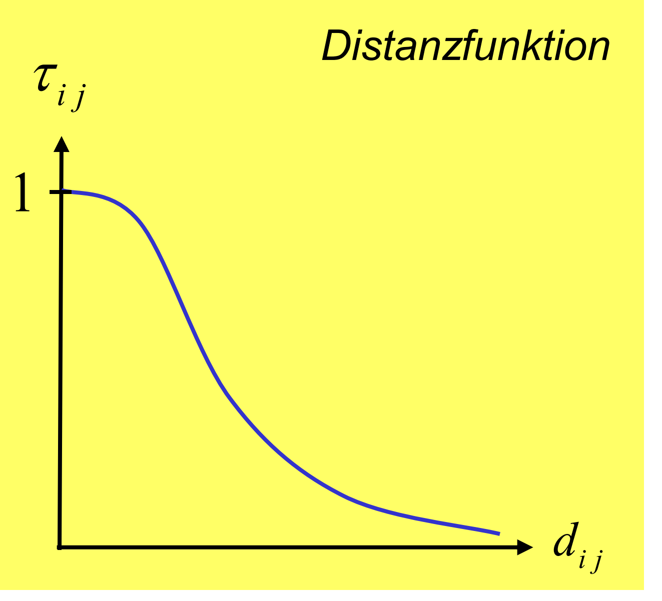
$$\vec{x} = \left(x_1, x_2, ...x_n\right)^T$$

Kohonen-Neuronen mit *Nachbarschafts-gitter* (<u>keine</u> Lateralverbindungen)

Definition: *Gitterdistanz u. Distanzfunktion* ***zwischen 2 Neuronen***



Im Neuronengitter benachbarte Neuronen üben Einfluss aufeinander aus. Je näher ein Nachbar ist, desto größer ist der Einfluss.



Topologische Distanz (Gitterdistanz)

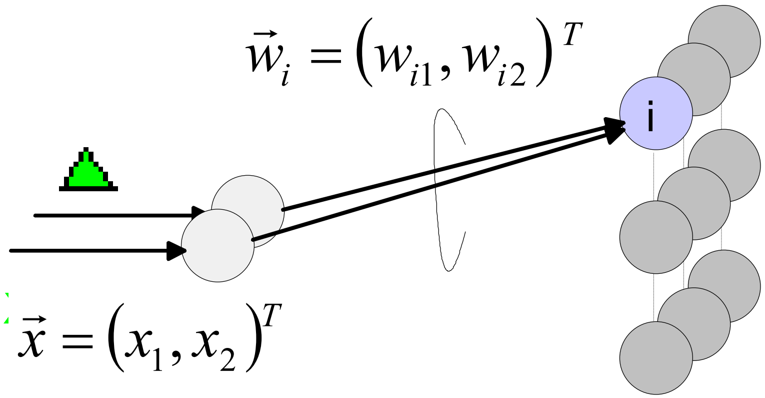
$$d_{ij} = \sqrt{(c_i - c_j)^2 + (r_i - r_j)^2}$$

Topologische Distanzfunktion (Wirkung)

$$\tau_{ij} = e^{-\frac{d_{ij}^2}{2\sigma^2}}$$

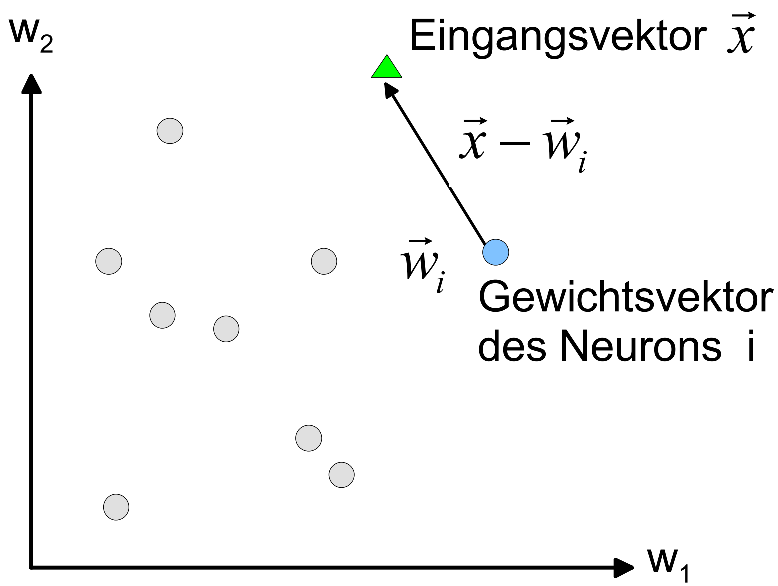
Die Distanzfunktion ist ein Maß für den Einfluß, den ein Neuron i auf ein Neuron j hat.

Definition: *Gewichtsdistanz* (X-W) **zwischen Neurongewicht und Eingangsvektor**



Der Eingangsvektor $\vec{x}$ übt Einfluss auf die Neuronengewichte aus.

Je geringer der Unterschied zwischen dem aktuellen Neuronengewicht $\vec{w}_i$ eines Neurons i und dem Eingangsvektor $\vec{x}$ ist, desto größer ist der Einfluss.

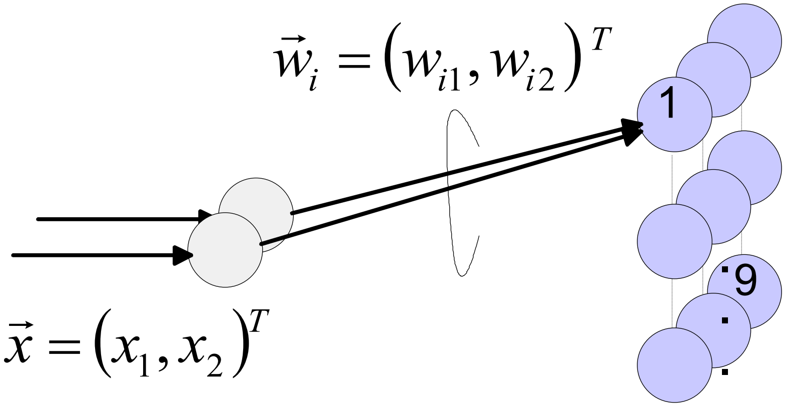


Interpretation der Darstellung

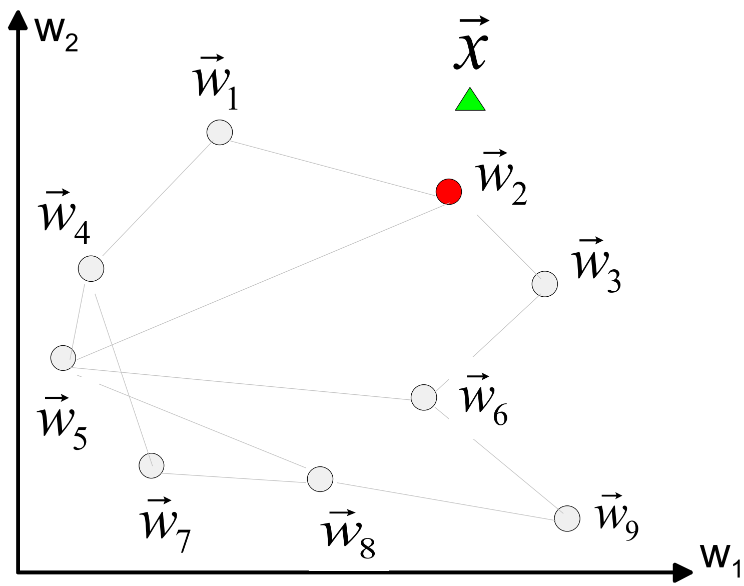
Die Position eines Punktes im Vektorraum stellt den Gewichtsvektor $\vec{w}_i$ des Neurons i dar.

Das grüne Dreieck kennzeichnet den Ort des Eingangsvektors im Vektorraum.

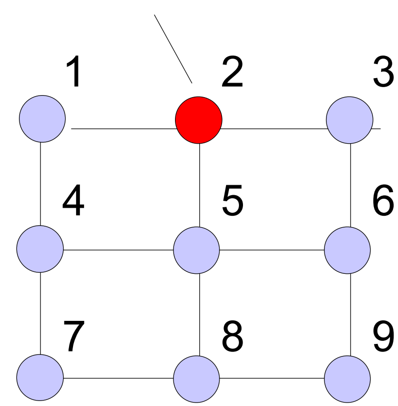
Definition: Gewinnerneuron



Das Neuron mit der kleinsten euklidischen Distanz zwischen Eingangsvektor $\vec{x}$ und Gewichtsvektor $\vec{w}_i$ ist das <u>Gewinnerneuron</u>.



Gewinnerneuron



Neuronengewichte mit eingezeichneter Neuronentopologie

6.4.2 Lernalgorithmus

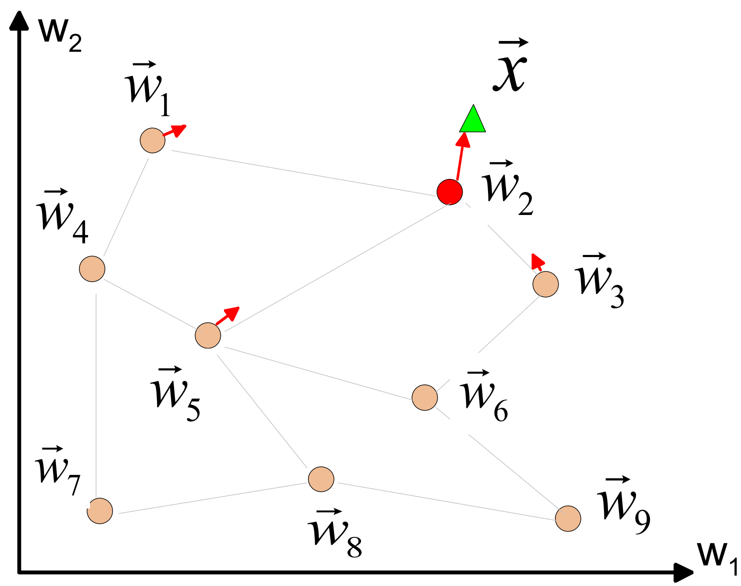
Der Gewichtsvektor $\vec{w}_i$ eines Neurons j wird umso mehr in Richtung des Eingangsvektors $\vec{x}$ verändert,

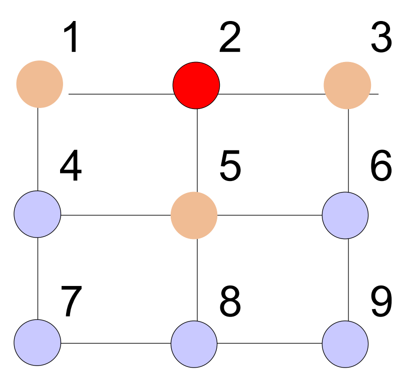
je kleiner die topologische Distanz des Neurons j vom Gewinnerneuron i ist.

$$\vec{w}_j(t+1) = \vec{w}_j(t) \; + \; \eta \cdot \tau_{ij} \cdot \left(\vec{x} - \vec{w}_j(t) \right)$$

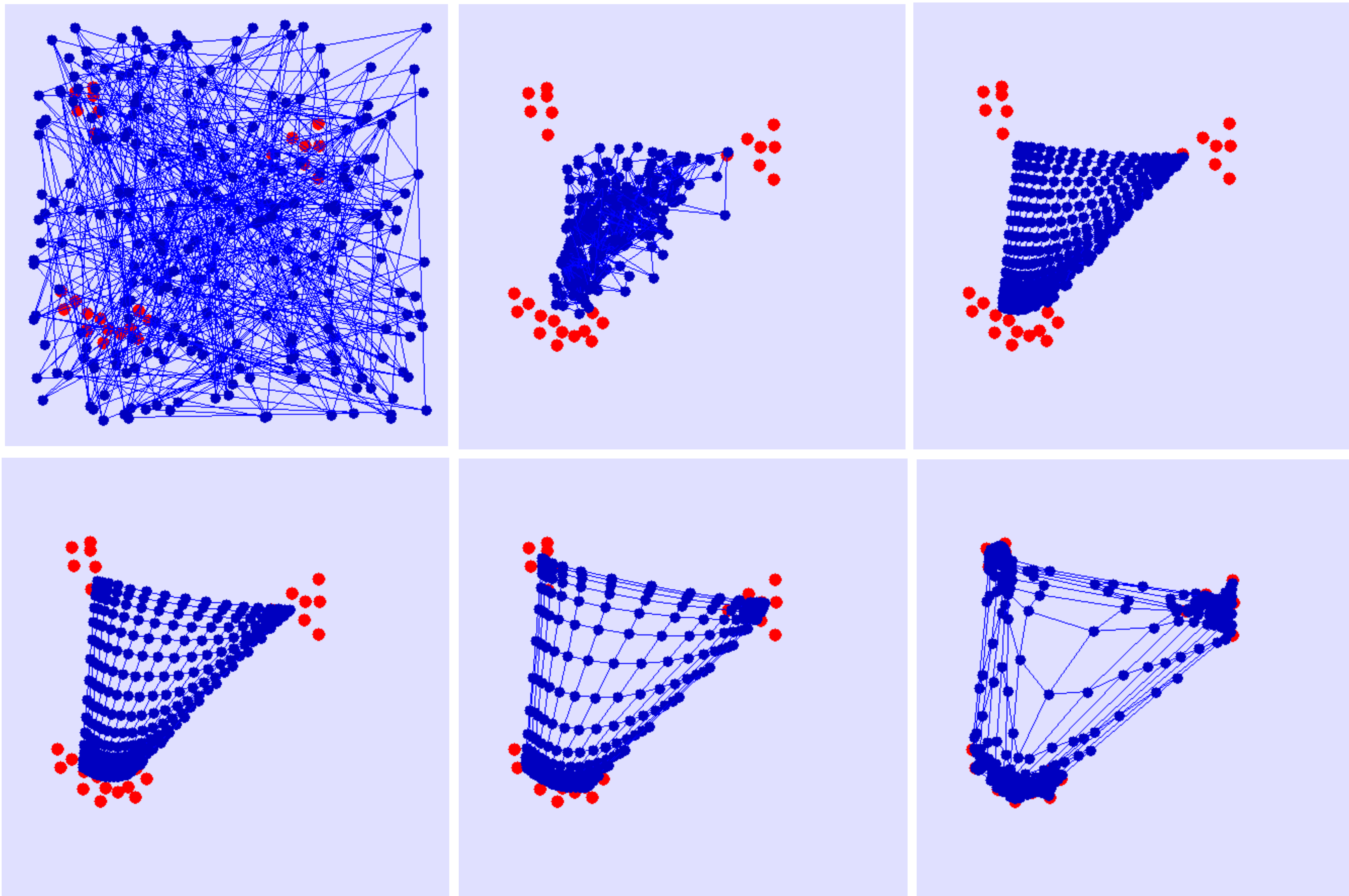
η Schrittweitenfaktor

τ_{ij} Distanzfunktionswert zwischen Neuron j und dem Gewinnerneuron i





Beispiel: Selbstorganisation und Clustering (2D-Netz)



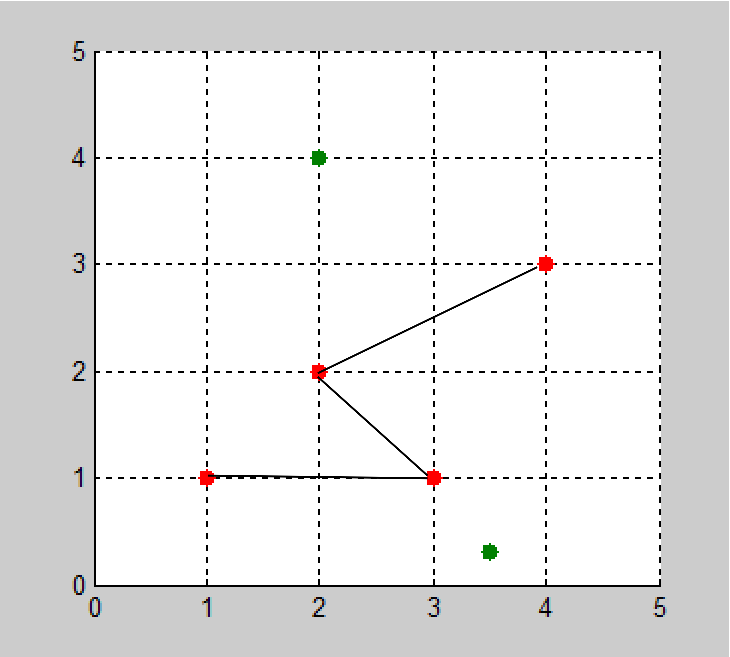
ÜBUNG: Kohonen-Netz

Gegeben sind zwei Eingabewerte:

$\vec{x}_1$ = (3.5, 0.3)
$\vec{x}_2$ = (2.0 4.0)

und ein Nachbarschaftsnetz (eindimensional) mit den initialen Gewichten:

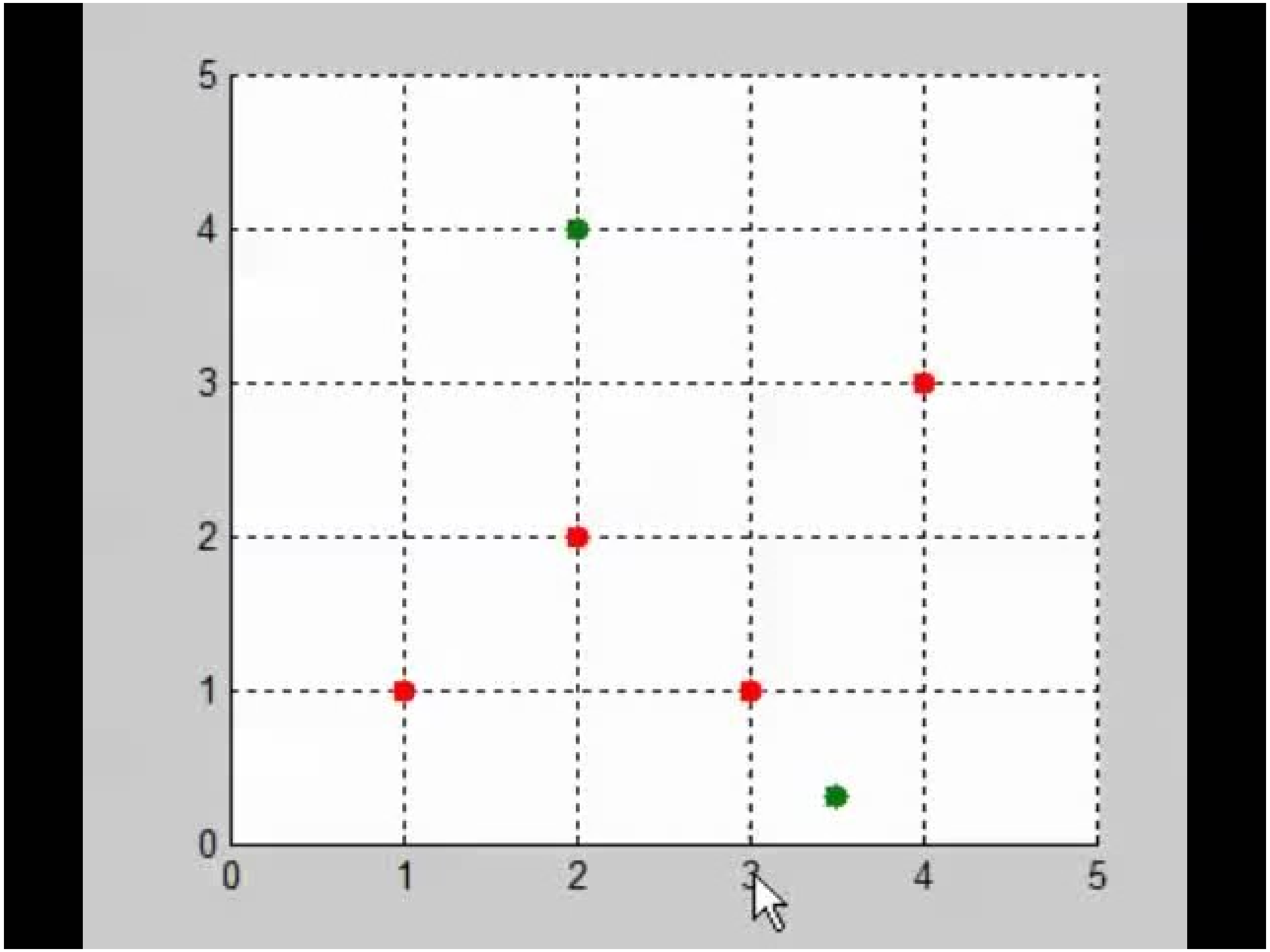
$\vec{w}_1$ = (1, 1)
$\vec{w}_2$ = (3, 1)
$\vec{w}_3$ = (2, 1)
$\vec{w}_4$ = (4, 3)



$\sigma = 2, \quad \eta = 0.5$

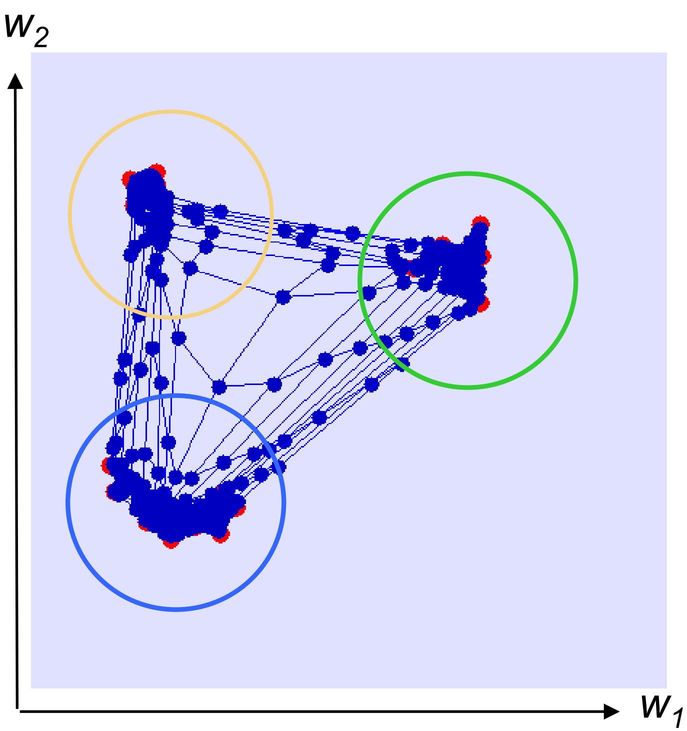
a) Geben Sie die Neuronengewichte nach Anwendung des ersten Eingabewertes an.
b) Geben Sie die Neuronengewichte nach Anwendung des zweiten Eingabewertes an.

```
for i = 1:NumberOfIterations
    for nx = 1:NumberOfInputs % hier: 2

        % finde Gewinnerneuron zu aktuellem Input x
        winner = -1;          currMinDist  = 1e12;
        for nw = 1:NumberOfNeurons    % hier: 4
            dist = norm(w(:,nw)-x(:,nx));
            if dist < currMinDist
                currMinDist  = dist;  winner = nw;
            end
        end

        % Gewichte w für aktuellen Input x korrigieren
        for nw = 1:NumberOfNeurons    % hier: 4
            d = abs(winner - nw);     % topol. Distanz
            Tau = exp(-d^2/(2*sigma^2));
            w(:,nw) = w(:,nw) + eta * Tau * (x(:,nx)-w(:,nw));
        end
    end
    % Ergebnis anzeigen .....
end
```

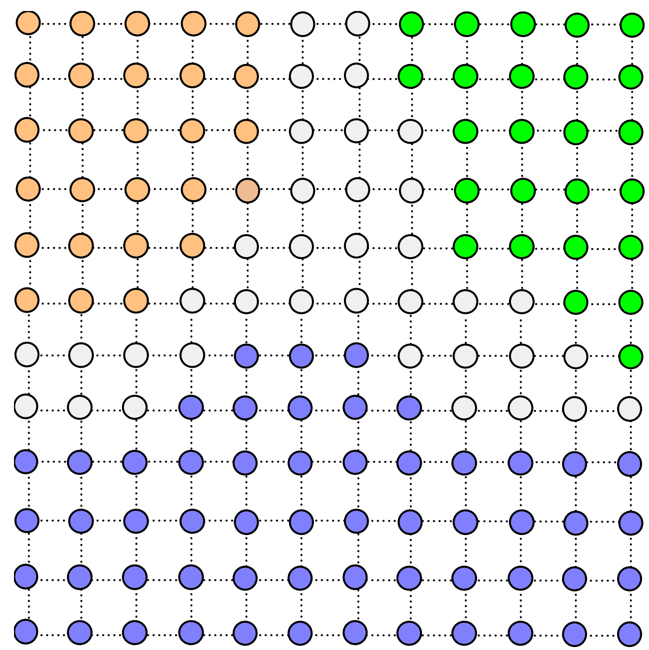

0
1
2
3
4
5
0
1
2
3
4
5

6.4.3 Anwendungen

Clustering



Gewichte der Neuronen



Neuronen mit etwa gleichen Gewichts-vektoren haben die gleiche Farbe (Klasse)

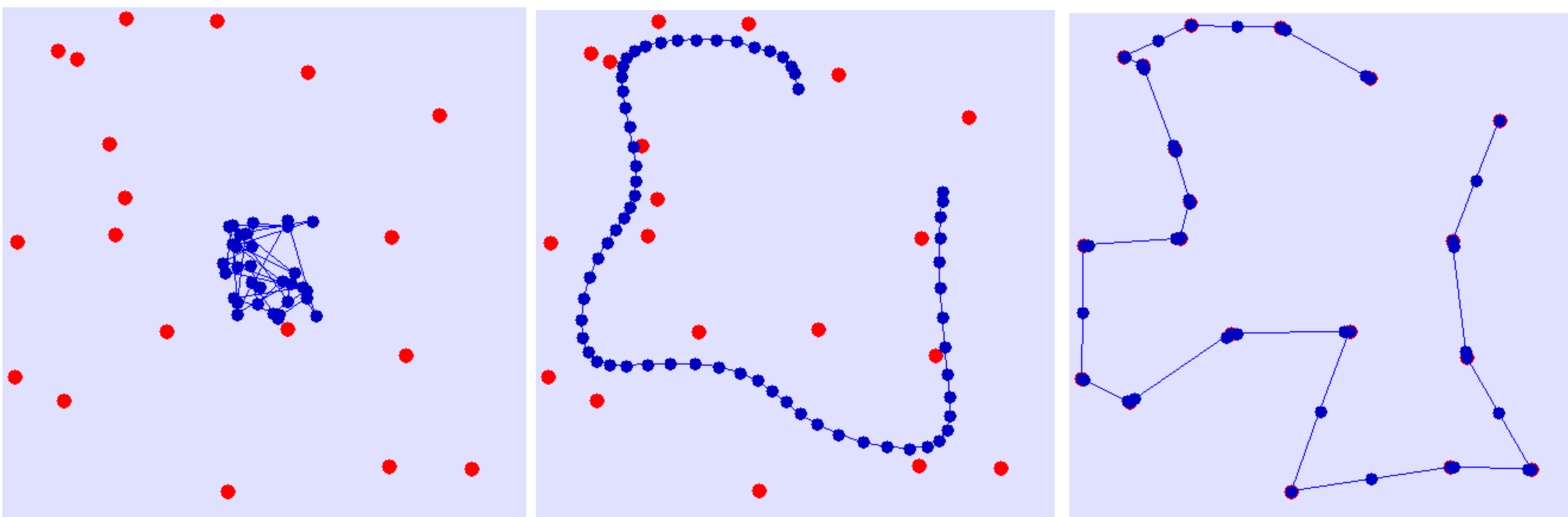
Optimierung

Beispiel : Travelling-Salesman-Problem (TSP)

Aufgabe: n Städte so besuchen, daß der Weg möglichst kurz ist.
→ Rechenaufwand: n!/2 Reisevarianten .

Problem: Auch bei relativ wenigen Städten in endlicher Zeit kaum noch lösbar.

Bsp.:	n=10 Städte	→	$n!/2 = 1.8*10^{6}$
	n=20 Städte	→	$n!/2 = 1.2*10^{18}$
	n=50 Städte	→	$n!/2 = 1.5*10^{64}$



Funktionsapproximation

Beispiel: y=f(x)

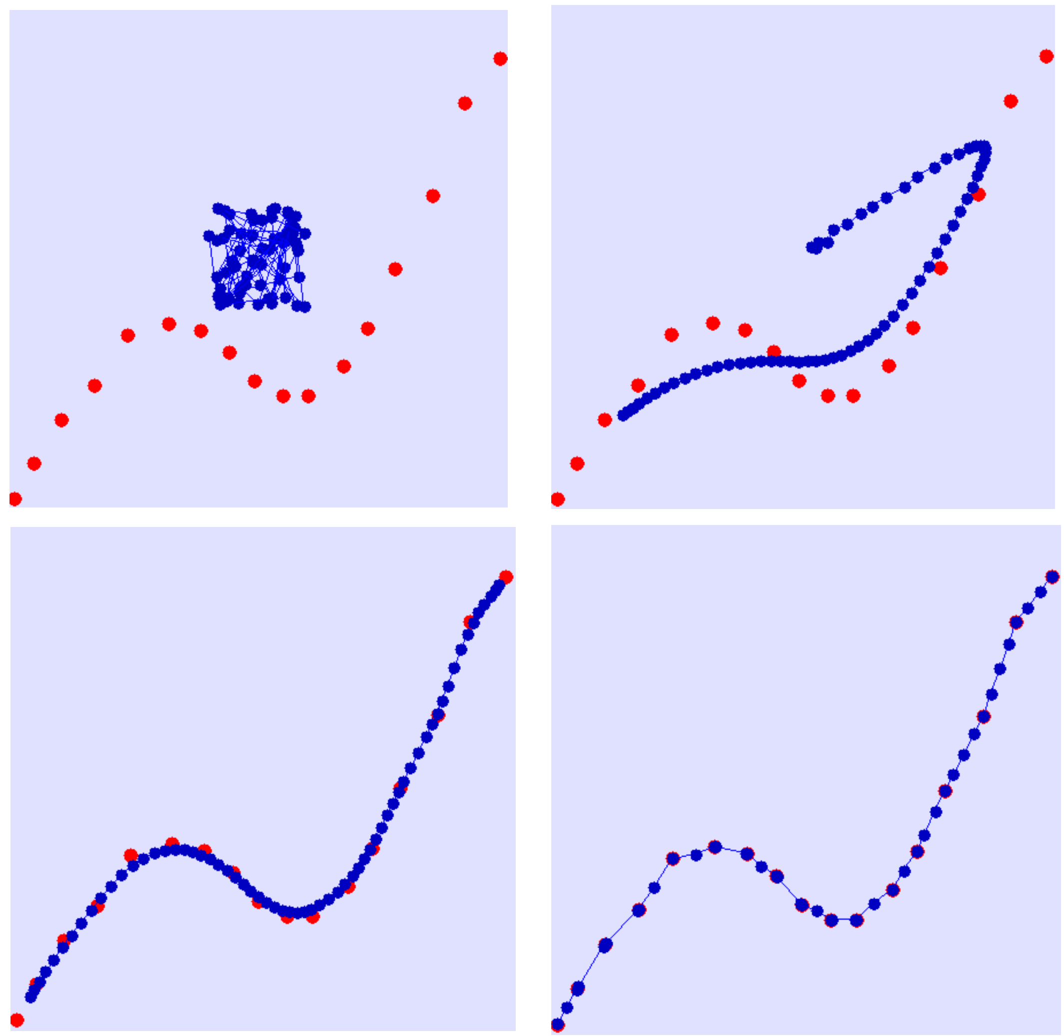
Training:

2 Eingänge → (x,y)

Betriebsphase:

Es wird das Neuronenpaar gesucht, deren x-Gewichte den gesuchten x-Wert einschließen.

Den gesuchten y-Wert erhält man durch lineare Interpolation zwischen den zugehörigen y-Gewichten.



6.5 Kurzübersicht über weitere Netztypen und Lernverfahren

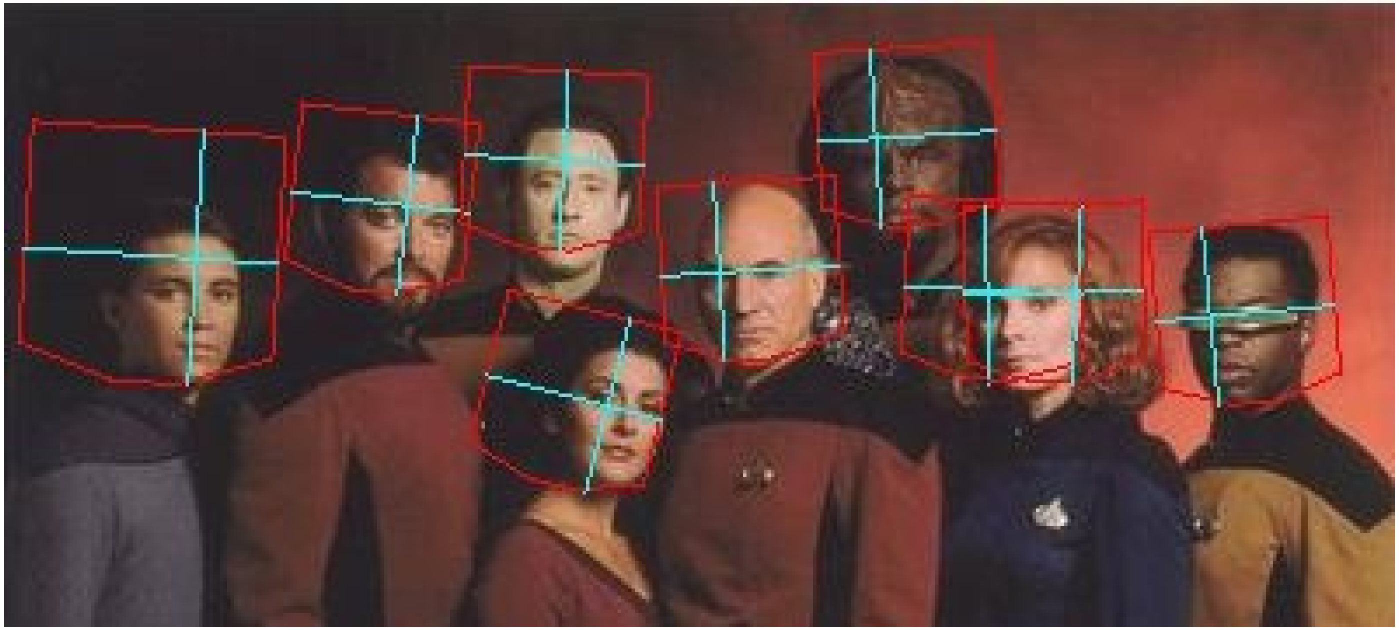
6.5.1 Gründe für andere Netztypen

- schnelleres Training
- Translations- und skalierungsinvariante Klassifikation von Bildmustern (z.B. *Neocognitron, Faltungsnetzwerke*)
- Nachtrainierbarkeit oder Online-Trainierbarkeit
  - Lernen neuer Assoziationen ohne das bereits Gelernte zu vergessen ("*Stabilitäts-Plastizitäts-Dilemma*")
  - Veränderlichkeit über die Zeit (hohe Plastizität)
- Erkennung von Systematiken in Zeitreihen
- Erlernen von dyn. Verhalten
- Verarbeitung binärer Signale
- Parallelisierbarkeit (Lern- und Abrufphase)

6.5.2 Kleine Auswahl weiterer Netztypen und Lernverfahren

<u>Convolutional Networks</u> (s. Le Cun)

- robuste Zeichenerkennung (unempfindlich gegen Rotation, Translation, Skalierung, Strichdicke, ...
- robuste Gesichtserkennung u.v.m.



s. http://yann.lecun.com/exdb/publis/index.html

Adaptive Resonance Theory (ART)

- Modellierung zeitveränderlicher Systeme (nachträgliches dazulernen)
- nachtrainierbar
- wenn nötig, werden neue Klassen erzeugt (Plastizität)
- alte Klassen werden nur wenig verändert (Stabilität)
- verschiedene Varianten für binäre/reellwertige Muster sowie für Zeitsignale

Cascade-Correlation Learning Architecture

- passt beim Lernen nicht nur die Gewichte an, sondern auch die Netztopologie
- alte Muster bleiben beim Lernen erhalten (Stabilität),
  neue Muster führen zu einer automatischen Erweiterung des Netzes (Plastizität)
- nachtrainierbar

6.6 Einsatzbeispiele

Mustererkennung

- Texturanalyse (z.B. Granitfliesen, Möbelholz)
- Klanganalyse
  - QS bei Keramiken
  - Motor,- Lager- und Getriebediagnose
- Gesichtserkennung, Ident. von Fingerabdrücken, Gestenerkennung
- Spracherkennung, Sprechererkennung
- Abfallsortierung, Sortierung von Naturprodukten nach Güteklassen
- Image-Retrievel-Systeme (Suche in Bilddatenbanken)
- OCR-Systeme, Lesen von Handschriften, Unterschriftkontrolle

Steuerungs- und Regelungstechnik

- Inverse Kinematiken
- Fluglageregelungen für Flugzeuge und Helikopter
- Regelung verfahrenstechnischer Prozesse, Dampferzeugung, Wasseraufbereitung
- Destillationsprozesse

Datenanalyse und Prognose

- Kennfeldoptimierung
- Aktienanalyse
- Kreditwürdigkeitsanalyse und Insolvenzprüfung
- Lastprognosen in der Energiewirtschaft oder Flugzeugen
- Wettervorhersage

Optimierung

- Tourenplanung ("Travelling Salesman Problem")
- Maschinenbelegungspläne (Planung von Reihenfolgen)
- Packprobleme ("*Rucksackproblem*")
- Stundenplanproblem

6.7. Entwicklung neuronaler Systeme

6.7.1 Entwurf, Aufbau und Training

6.7.1.1 Wann ist der Einsatz Neuronaler Netze sinnvoll ?

Gibt es für die zu lösende Aufgabe analytische Verfahren ?

Beispiel "Regelungstechnik":

Lassen sich PID- ,Zustandsregler oder Adaptive Regler einsetzen ?
Wenn ja, sind diese i.allg. zu bevorzugen, da deren korrekte Funktion anhand von Stabilitätskriterien beweisbar ist.

Beispiel "Bildverarbeitung":

Ist das zu lösende Problem algorithmisch lösbar ?
Welche Gründe sprechen für eine neuronale Lösung.

Neuronale Netze sind kein Allheilmittel gegen Unwissenheit im Bereich der analytischen Verfahren.

Es gibt meist gute Gründe die entweder für analytische Verfahren oder für neuronale Verfahren sprechen.

6.7.1.2 Festlegung der Vorverarbeitung

Man darf von NN keine allzu hohen Abstraktionsleistungen erwarten.

- Die Zahl der Eingangsmerkmale sollte so gering wie möglich gehalten werden.

- Der Informationsgehalt der Merkmale bezüglich der zu lösenden Aufgabe sollte so hoch wie möglich sein.

- Teilaufgaben sollten isoliert gelöst werden.

6.7.1.3 Zusammenstellen der Trainings- und Testdaten

- nicht zu unterschätzender Aufwand

- eine ausreichende Trainingsdatenmenge ist oft schwer oder gar nicht beschaffbar

- Die Trainingsdaten müssen gesichtet und klassifiziert werden
  (für "supervised learning"). → ggf. Entwicklung geeigneter Werkzeuge.

- Die Trainingsdaten müssen in ein geeignetes Datenformat überführt werden.
  → ggf. Entwicklung geeigneter Konvertierungsprogramme.

6.7.1.4 Wahl der Trainingsparameter und des Lernalgorithmus

Der Entwurfserfolg ist vor allem abhängig von der Intuition des Netzdesigners.

Trainingsparameter sowie die Anzahl der Layer und "hidden Neuronen" werden am besten experimentell ermittelt (Batchjobs).

6.7.1.5 Training und Test

Die Qualität des Netzes kann nur durch Test mit einem unabhängigen Testdatensatz verifiziert werden