Grundlagen der Künstlichen Intelligenz

15 Lernen in Künstlich Neuronalen Netzen

Grundlagen, Netzwerkstrukturen, Perzeptron, Backpropagation

Volker Steinhage

Inhalt

- Motivation
- Grundlegende Elemente neuronaler Netze
- Netzwerkstrukturen
- Lernen in neuronalen Netzen
- Perzeptron
- Backpropagation
- Beispielanwendung
- Zusammenfassung & Ausblick

Einordnung (1)

Vom mathematischen Standpunkt:

Künstlich Neuronale Netze (*KNNs*) als *Methode*, Funktionen zu repräsentieren

- durch Netzwerke von einfachen Berechnungselementen

  (vergleichbar mit logischen Schaltkreisen)

- die aus Beispielen gelernt werden können

⇝ in dieser Vorlesung!

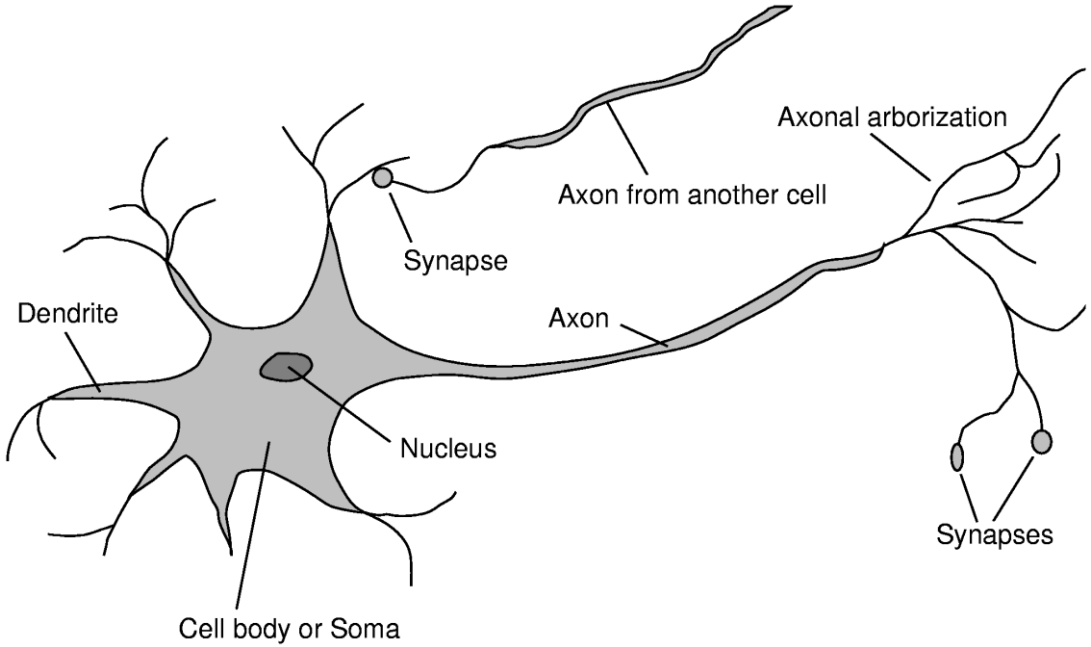
Vom biologischen Standpunkt:

Künstlich Neuronale Netze als *Modell* des Gehirns und seiner Funktionsweise

⇝ nicht in dieser Vorlesung!

Motivation

Ausgangspunkt: Abstrahierende Nachbildung von Struktur und Verarbeitungsmechanismen des Gehirns mit ca. 10^9 Neuronen und ca. 10^{18} verbindenden Synapsen



Ziel: Viele Prozessoren (Neuronen) und Verbindungen (Synapsen), die parallel und verteilt Informationen verarbeiten.

Forschung auf dem Gebiet der KNNs (1)

1943: McCulloch und Pitts führen die Idee eines binären Neurons ein

1943-1969: Forschung an KNNs meist an einschichtigen KNNs (*Perzeptrons*)

1969: Minsky und Papert zeigen, dass *Perzeptrons* sehr beschränkt sind

1969-1980: Kaum noch Arbeiten auf dem Gebiet

Seit 1981: Erste Renaissance neuronaler Netze

- in der KI als *Werkzeug* benutzt zum Approximieren von Funktionen
- insbesondere für sensomotorische Aufgaben gut geeignet
- in der Biologie und Physiologie als Modell

Forschung auf dem Gebiet der KNNs (2)

Seit ca. 2009: Zweite Renaissance neuronaler Netze

- KNNs liefern bei herausfordernden Anwendungen wie intern. Mustererkennungswettbewerben bessere Ergebnisse als konkurrierende Lernverfahren
- Zwischen 2009 und 2012 gewannen tiefe KNNs aus dem Schweizer KI Labor IDSIA gleich acht internat. Wettbewerbe in den Bereichen Mustererkennung und masch. Lernen(1)
- Tiefe KNNs auf der Basis effizienter GPU-Implementierungen und in Verbindung mit „Big Data“ erzielen die bisher besten Ergebnisse auf dem *ImageNet* Benchmark (2),(3),(4)

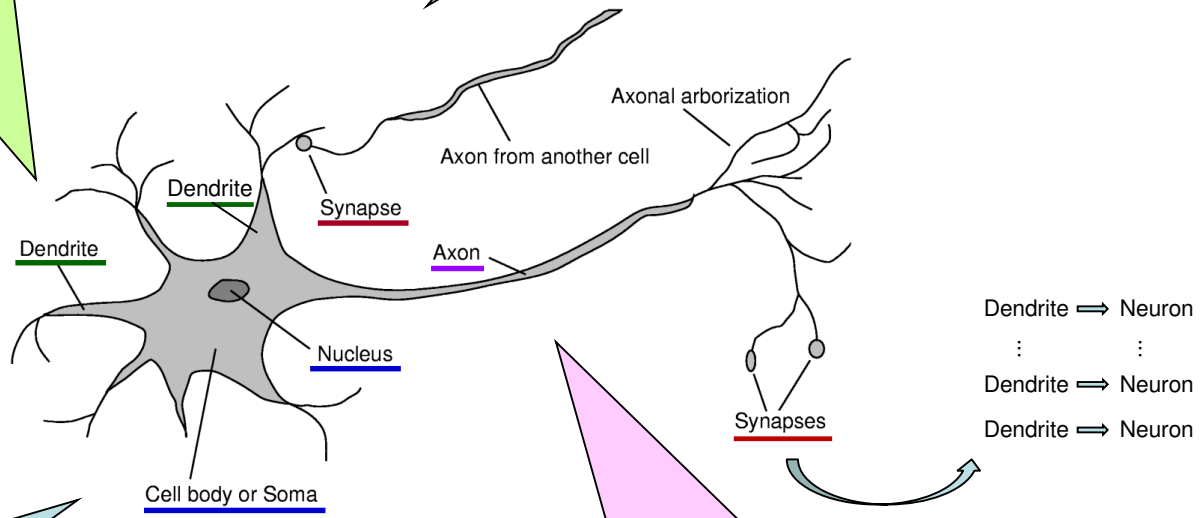
---

(1) http://www.kurzweilai.net/how-bio-inspired-deep-learning-keeps-winning-competitions 2012 Kurzweil AI Interview mit Jürgen Schmidhuber zu den acht Wettbewerben, die sein Deep Learning Team zwischen 2009 und 2012 gewann.
(2) A. Krizhevsky, I. Sutskever, G. E. Hinton: *ImageNet Classification with Deep Convolutional Neural Networks.* NIPS 25, MIT Press, 2012. http://www.cs.toronto.edu/~hinton/absps/imagenet.pdf
(3) M. D. Zeiler, R. Fergus: *Visualizing and Understanding Convolutional Networks.* TR arXiv:1311.2901 [cs.CV], 2013. http://arxiv-web3.library.cornell.edu/abs/1311.2901.
(4) https://en.wikipedia.org/wiki/ImageNet (13.06.18)

Grundbegriffe natürlicher neuronaler Netze *

Dendriten: Stachelartige Fortsätze eines Neurons, auf denen mehrere Synapsen enden

Synapsen: Verbindung zwischen Dendrit eines Neurons und Axon eines anderen Neurons



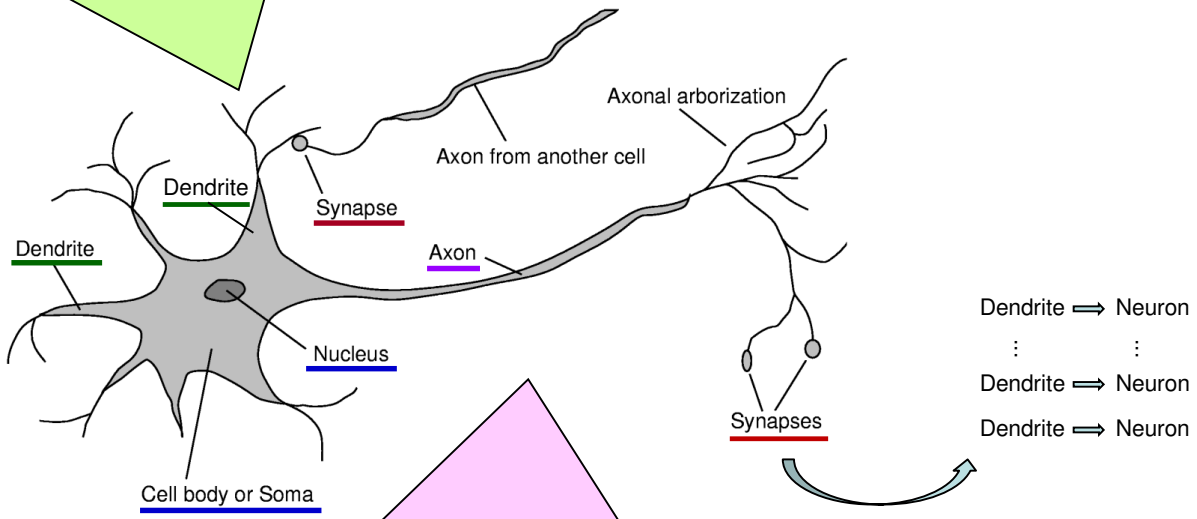
Neuron: Nervenzelle, bestehend aus Zellkörper (Soma) und Zellkern (Nucleus)

Axon: Verbindungsfaser eines Neurons, die am Ende oft verzweigt in Vielzahl synaptischer Endungen an anderen Neuronen

* Diese Darstellung dient nicht einer hinreichenden Erklärung neurophysiologischer Zusammenhänge, sondern der Erläuterung einiger der wichtigsten Begrifflichkeiten und Zusammenhänge aus der Neurophysiologie, auf deren Grundlage Begriffe und Funktionalitäten von künstliche Neuronalen Netze (KNN) abgeleitet werden.

Grundbegriffe natürlicher neuronaler Netze (2)

Signalübertragung 1: Chem. Neurotransmittersubstanzen (Natrium- und Kaliumionen) werden von **Synapsen** auf **Dendriten** übertragen und verändern das elekrostatische Potential des **Neurons**

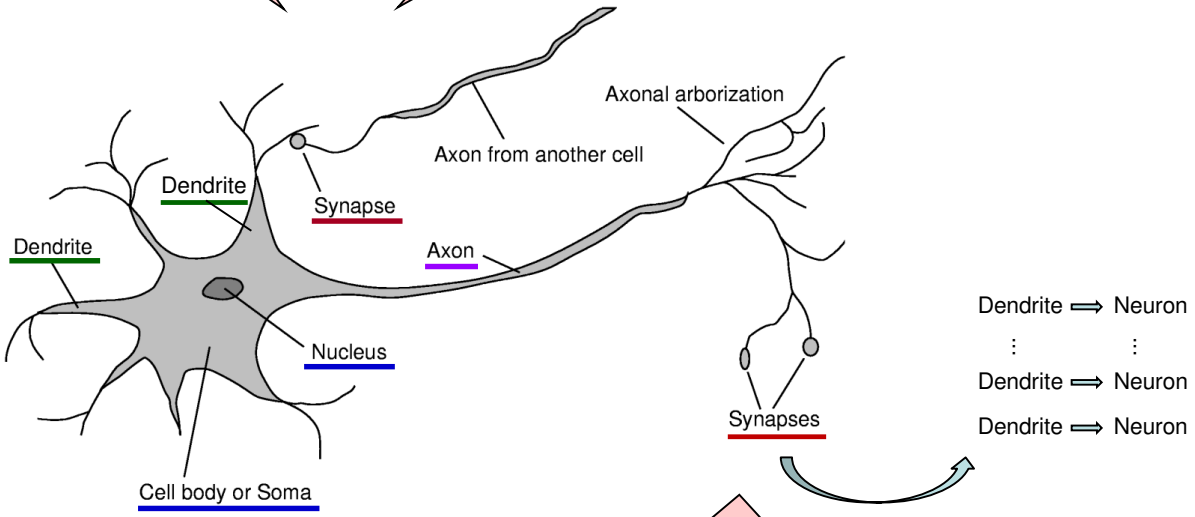


Signalübertragung 2: Ab bestimmten Wert des sog. Aktionspotentials des Neurons wird dieses Potential als Nervensignal entlang des **Axons** fortgepflanzt bis zu den **Synapsen** anderer Neuronen

Grundbegriffe natürlicher neuronaler Netze *

Erregende Synapsen (*Excitatory Synapses*): Erhöhen das Potential des Neurons und animieren damit das Aussenden von Nervensignalen

Hemmende Synapsen (*Inhibitory Synapses*): Verringern das Potential des Neurons und behindern damit das Aussenden von Nervensignalen

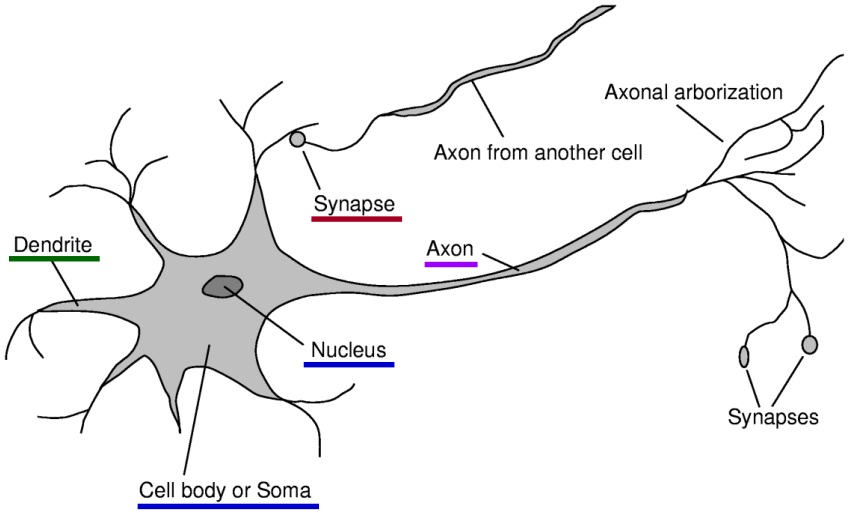


Plastizität von Synapsen: Die Stärke von Erregung bzw. Hemmung hängt von der Transmitterausschüttung der Synapse ab. Diese ändert sich über die Zeit aufgrund chemischer Veränderungen in der Synapse, die wiederum von der Geschichte der empfangenen Nervensignale abhängen ⇝ Speichern und Lernen von Information

Grundbegriffe künstlicher neuronaler Netze

Einheiten (*Units*): Stellen die Knoten (Neuronen) im Netz dar.

Verbindungen (*Links*): Kanten zwischen den Knoten des Netzes. Jeder Knoten hat Ein- und Ausgabekanten.



Gewichte (*Weights*): Jede Kante hat eine Gewichtung, in der Regel eine reelle Zahl.

Ein-/Ausgabeknoten (*Input and Output Units*): Besonders gekennzeichnete Knoten, die mit der Außenwelt verbunden sind.

Aktivierungsniveau (*Activation Level*): Der von einem Knoten aus seinen Eingabekanten zu jedem Zeitpunkt berechnete Wert. Dieser Wert wird über Ausgabekanten an die Nachbarknoten weitergeleitet.

Funktionsweise einer Einheit

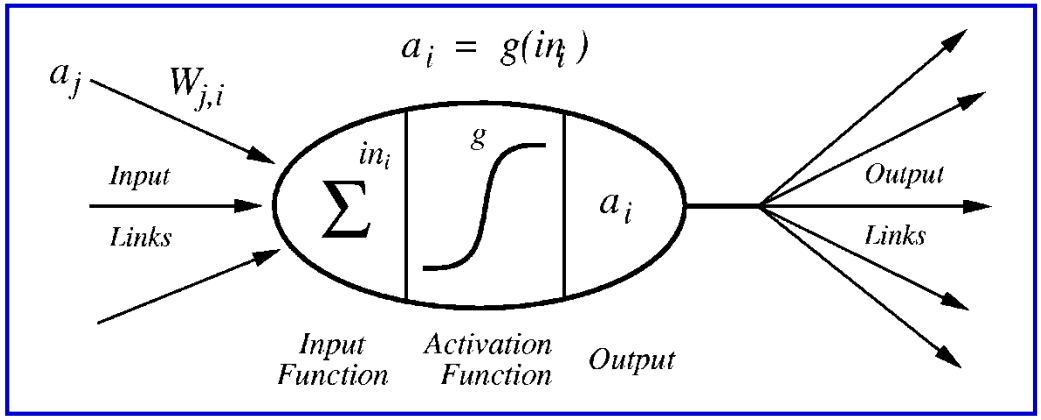
Eingabefunktion in_i berechnet die Stärke der Eingabe für Einheit i als *Linear-kombination* von Eingabeaktivierung a_j und Gewichten $w_{j,i}$ über alle Knoten j, die direkt mit i verbunden sind:

$$in_i = \sum_{j=0,\ldots,n} w_{j,i} \cdot a_j = \mathbf{w}_i \cdot \mathbf{a}_i$$

mit Vektoren $\mathbf{a}_i$ und $\mathbf{w}_i$ der eingehenden Aktivierungswerte bzw. entspr. Gewichte.

Eine i.A. nichtlineare Aktivierungsfunktion g berechnet das Aktivierungsniveau a_i:

$$a_i = g(in_i) = g\left(\sum_{j=0,\ldots,n} w_{j,i} \cdot a_j\right).$$

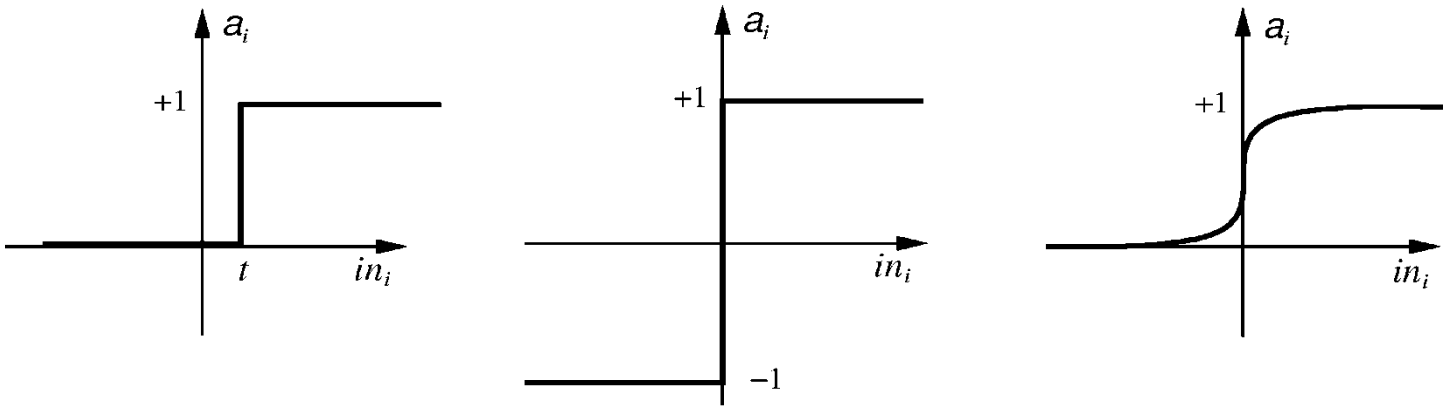


Aktivierungsfunktion

$$step_t(x) = \begin{cases} 1 & \text{if } x \geq t \\ 0 & \text{if } x < t \end{cases}$$

$$sign(x) = \begin{cases} +1 & \text{if } x \geq t \\ -1 & \text{if } x < t \end{cases}$$

$$sigmoid(x) = \frac{1}{1+e^{-x}}$$



(a) Step function **(b) Sign function** **(c) Sigmoid function**

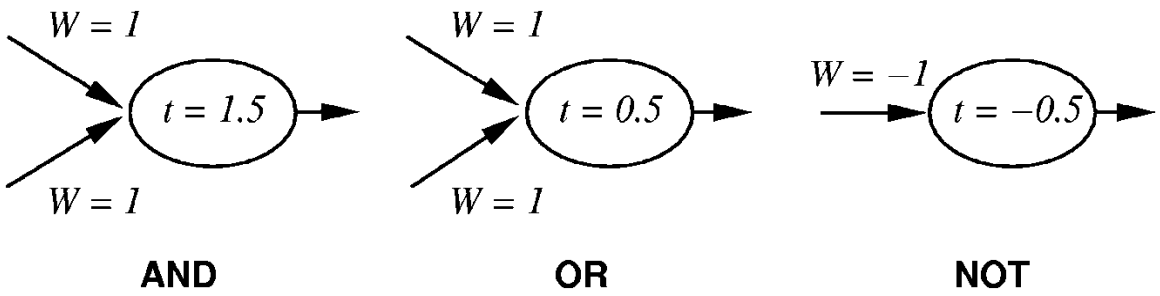
Beachte: *t* in $step_t$ stellt einen Schwellwert (*threshold*) dar – in Analogie zum Aktionspotential beim natürl. Nervensystem (s. Signalübertragung 2 auf Folie 8).

Statt des Schwellwertes *t* wird ein on-Neuron (Bias), also ein konstanter Eingang $a_0 = 1$, hinzugefügt. Der Schwellwert wird durch die Gewichtung $w_{0,i} = -t$ angegeben bzw. gelernt:

$$a_i = step_t\left(\sum_{j=1,\ldots,n} w_{j,i} \cdot a_j\right) = step_0\left(\sum_{j=0,\ldots,n} w_{j,i} \cdot a_j\right).$$

Was kann man mit neuronalen Netzen darstellen?

- z.B. *Boolesche Funktionen* (mit Hilfe der $step_t$ Funktion):



… und durch Verschaltung vieler Einheiten beliebige Boolesche Funktionen

⇝ Schaltkreistheorie

⇝ Aber neuronale Netze können noch viel mehr …

Netzwerktopologien

- Rekurrente Netze oder zyklische Netze zeigen i.A. bidirektionale Verbindungen, aus denen beliebige Topologien gebildet werden können.

  nicht in dieser Vorlesung

  – Ausgaben des Netzes können dabei als Eingaben zurückgegeben werden.

  – Die Netzantwort für eine bestimmte Eingabe kann so vom Ausgangszustand abhängig sein, der wiederum von vorherigen Eingaben abhängig sein kann (↝ Art von Kurzzeitgedächtnis ↝ interessant für Gehirnmodellierung)

  Gegenstand dieser Vorlesung

- Feed-forward-Netze oder azyklische Netze zeigen unidirektionale Verbindungen.

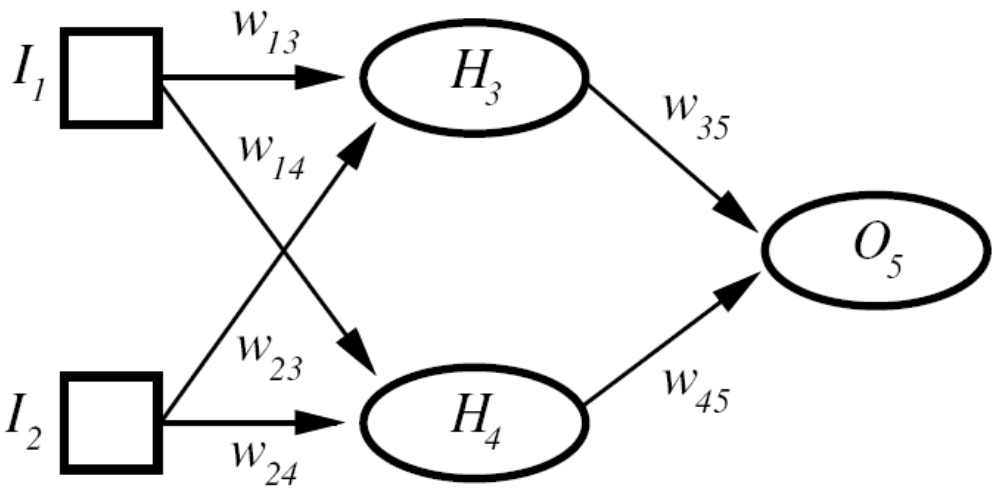
- Hybrid: Modular aus azyklischen und rückgekoppelten Komponenten aufgebaut. Beispiel: *Winner-takes-all-Netze* als unidirektionale Feed-forward-Netze mit lateraler bidirektionaler Inhibition in Wettbewerbsschichten.

nicht in dieser Vorlesung

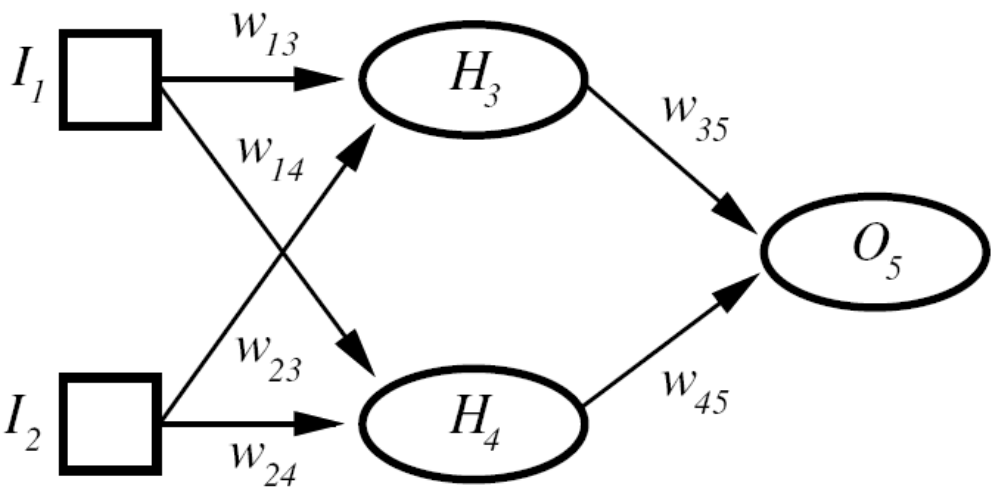
Feed-forward-Netze (FF-Netze)

Feed-forward-Netze werden i. A. in Schichten (Layer) konstruiert (→ GFFs). Dabei existieren keine Verbindungen zwischen Einheiten einer Schicht, sondern immer nur zur jeweils nächsten Schicht (gerichtete Netze)

Beispiel:

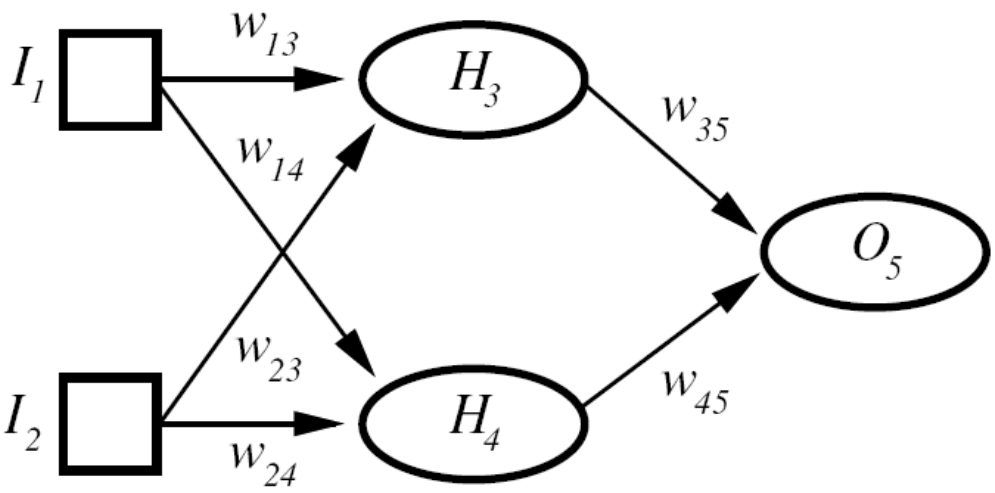


Geschichtete Feed-forward-Netze (GFFs) (1)



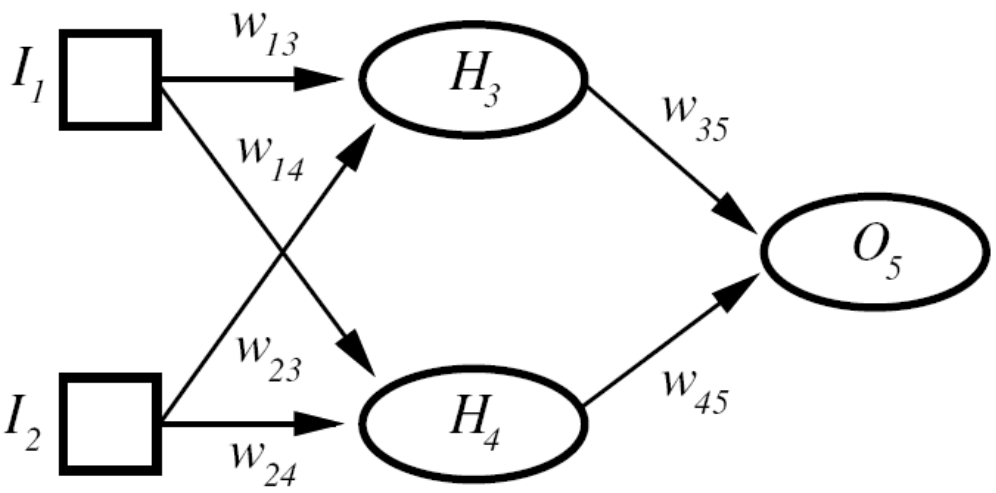
- **Eingabeeinheiten** (hier: I_1, I_2) erhalten ihr Aktivierungsniveau von der Umwelt
- **Ausgabeeinheiten** (hier: O_5) teilen ihr Ergebnis der Umwelt mit
- **Verborgene Einheiten** (*Hidden Units*) (hier: H_3, H_4)
  - sind Einheiten ohne direkte Verbindungen zur Umwelt
  - sind in verborgenen Schichten (*Hidden Layers*) angeordnet

Geschichtete Feed-forward-Netze (GFFs) (2)



- **Anzahl der Schichten**: wird angegeben *unter Ignorierung der Eingabeschicht*
- **Perzeptron**: einschichtiges FF-Netz ohne verborgene Schicht
- **Mehrschichtige Netzwerke** (*Multilayer Networks*):
  mindestens eine verborgene Schicht

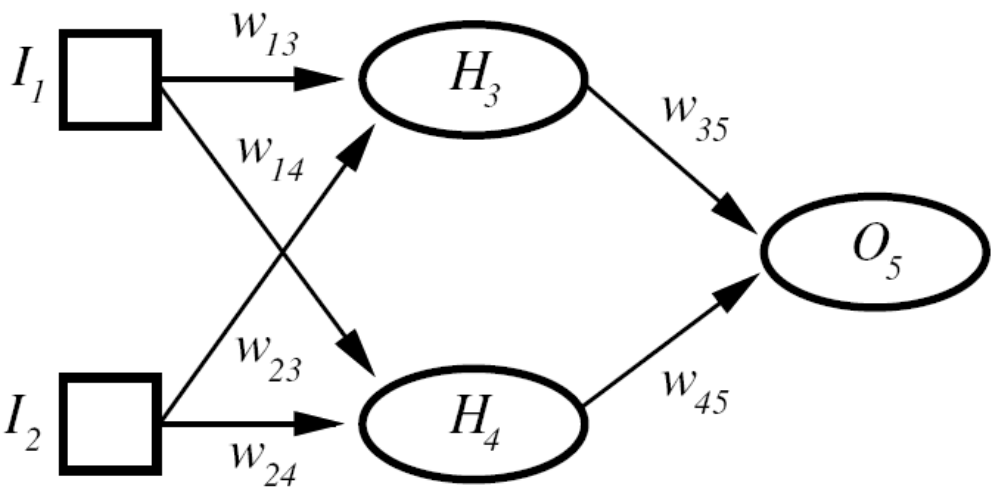
Darstellungskraft von GFF-Netzen



- GFF-Netze *mit einer verborgenen Schicht* können beliebige *stetige Funktionen* darstellen.

- GFF-Netze *mit zwei verborgenen Schichten* können zusätzlich auch *diskontinuierliche Funktionen* darstellen.

Beispiel



Bei fester Topologie und Aktivierungsfunktion g sind darstellbare Funktionen charakterisierbar in parametrisierter Form (*Parameter = Gewichte*):

$$\begin{aligned} a_5 &= g(w_{35} \cdot a_3 + \cdot w_{45} \cdot a_4) \\ &= g(w_{35} \cdot g(w_{13} \cdot I_1 + w_{23} \cdot I_2) + w_{45} \cdot g(w_{14} \cdot I_1 + w_{24} \cdot I_2)) \end{aligned}$$

$\Rightarrow$ Lernen in NNs = Suche nach richtigen Parameter- bzw. Gewichtswerten

$\Rightarrow$ aus statistischer Sicht: nichtlineare Regression

Lernen mit neuronalen Netzen

- Gegeben sei eine Menge von Beispielen $E = \{e_1, \ldots, e_n\}$, wobei jedes Beispiel aus *Eingabewerten* und *Ausgabewerten* besteht

- Ziel: Netzwerk soll die Funktion lernen, die die Beispiele erzeugt hat – dabei ist die Netzwerktopologie vorgegeben und die Gewichte sollen angepasst werden

- Lernen in Epochen: man legt dem Netz die Beispiele mehrfach (in Epochen) vor

```
function NEURAL-NETWORK-LEARNING(examples) returns network

    network ← a network with randomly assigned weights
    repeat
        for each e in examples do
            O ← NEURAL-NETWORK-OUTPUT(network, e)
            T ← the observed output values from e
            update the weights in network based on e, O, and T
        end
    until all examples correctly predicted or stopping criterion is reached
    return network
```

Optimale Netzwerkstruktur?

1) Die Wahl des richtigen Netzes ist ein schwieriges Problem!

2) Zudem kann das optimale Netz exponentiell groß relativ zur Eingabe werden*

Probleme:

- Netz zu klein: gewünschte Funktion nicht darstellbar
- Netz zu groß: beim Trainieren lernt das Netzwerk die Beispiele „auswendig“ ohne zu generalisieren ⇝ *Overfitting*

Es gibt keine gute Theorie zur Wahl des richtigen Netzes!

---

* Z.B. Netze für Boolesche Funktionen: Ein Ergebnis aus der Schaltkreistheorie besagt, dass die *meisten* Booleschen Funktionen mit $O(2^n/n)$ Gattern dargestellt werden müssen.

Heuristik des *Optimal brain damage*

Optimal Brain Damage*:

1. Wähle Netz mit sehr großer Zahl an Verbindungen und trainiere das Netz
2. Reduziere Zahl der Verbindungen mit Hilfe von Informationstheorie und trainiere das Netz erneut
3. Wenn Performanz gleich oder besser, dann gehe zu 2

Beispiel: Netz zum Erkennen von handgeschriebenen Postleitzahlen

→ 3/4 der anfänglichen Verbindungen wurden eingespart

Es gibt auch Verfahren, um von einem kleinen Netz zu einem besseren, größeren Netz zu kommen: Füge sukzessive Knoten für alle nicht erlernten Beispiele ein bis Trainingsmenge korrekt klassifiziert

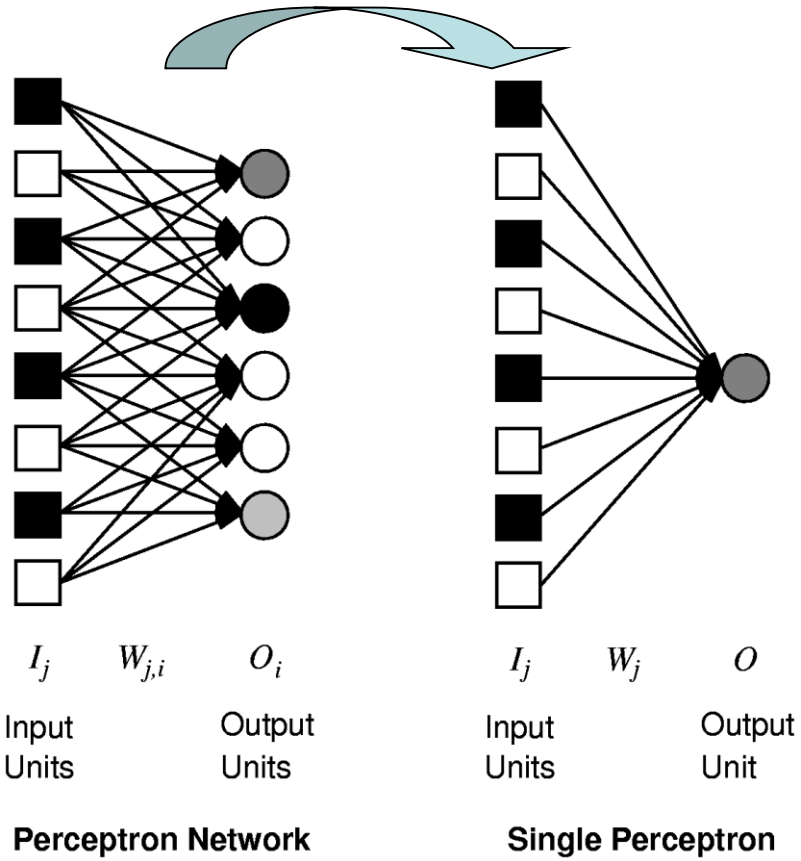
---

* Yann LeCun, John Denker, Sara Solla. Optimal Brain Damage. Advances in Neural Information Processing Systems 2 (NIPS 1989)

Perzeptron

Einschichten-FF-Netze oder *Perzeptrons* wurden insbes. in den 1950er Jahren studiert. Sie waren die einzigen GFFs mit bekannten *effektiven* Lernverfahren.

Da alle Ausgabeknoten unabhängig voneinander sind, ist Betrachtung auf einen einzigen Ausgabeknoten *O* reduzierbar:

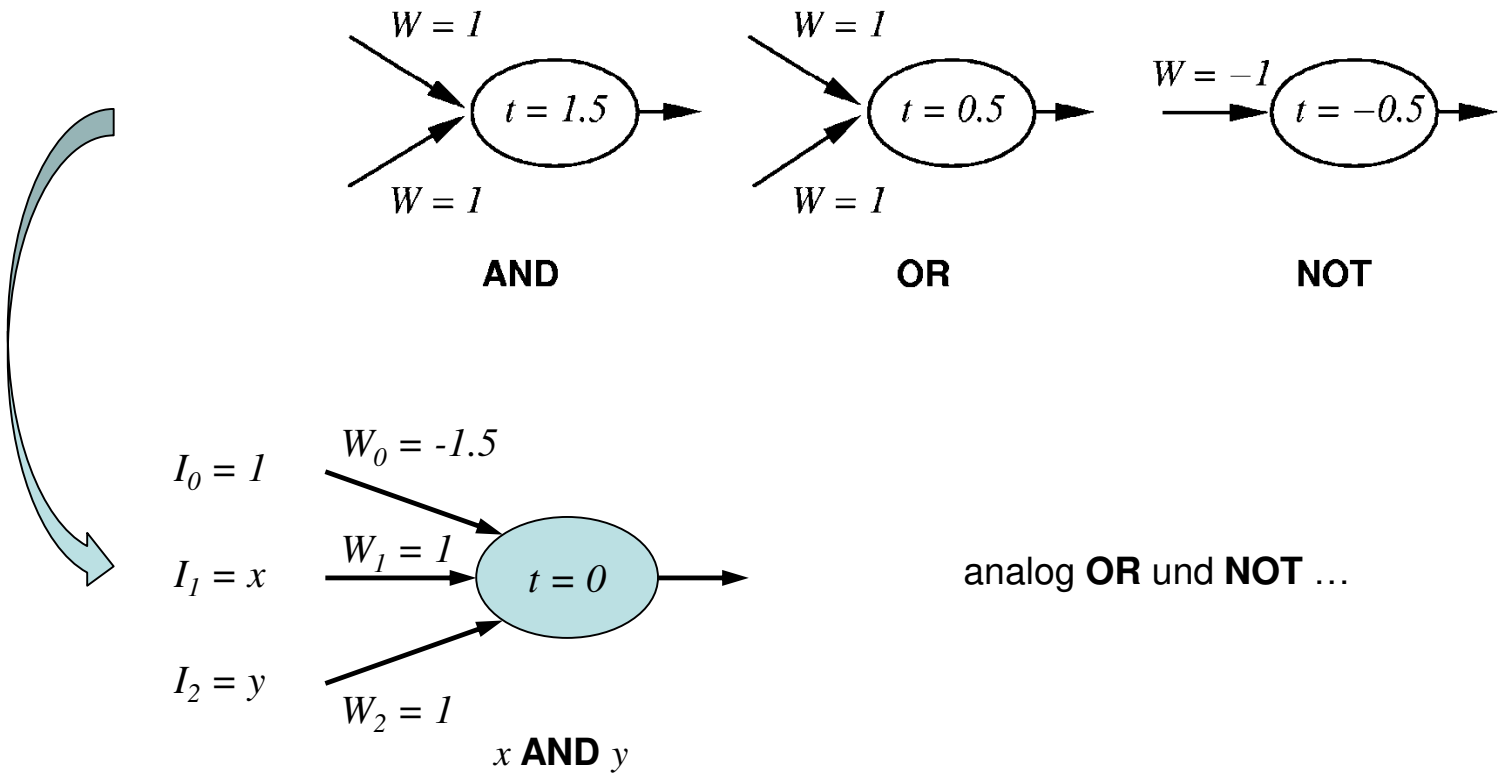


Was können Perzeptrons darstellen?

Mit Schwellwert-Aktivierungsfunktionen sind z.B. Boolesche Funktionen darstellbar:

$$O = step_0\left(\sum_{j=0,\ldots,n} W_j \cdot I_j\right) = \boldsymbol{W} \cdot \boldsymbol{I}$$

für n-stellige Boolesche Funktionen mit $I_0 = 1$ als on-Neuron (Bias)



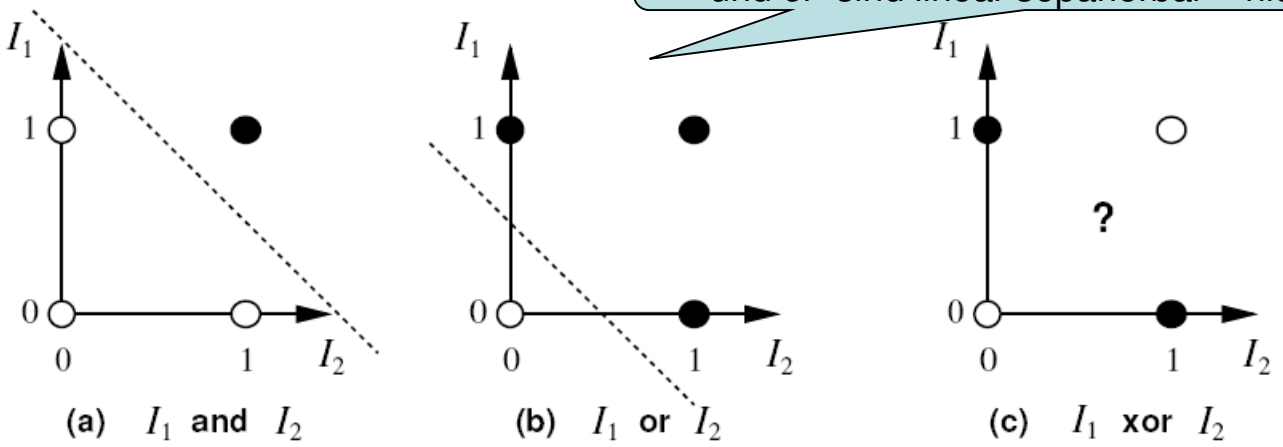
(Vgl. Folie 13)

analog **OR** und **NOT** …

Was können Perzeptrons darstellen?

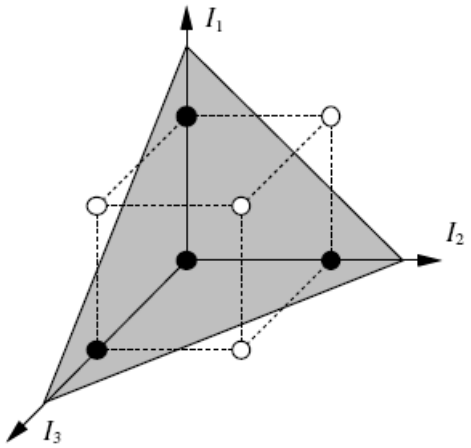
Aber: Perzeptrons können nur die Klasse aller *linear separierbaren Funktionen* darstellen:

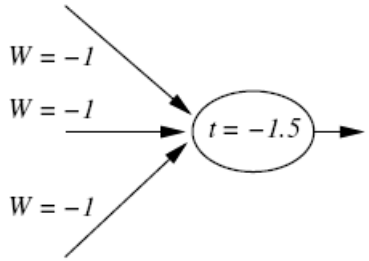
Urbilder von 0 und 1 der Booleschen Funktionen *and* und *or* sind linear separierbar – nicht so bei *xor*



Grund: Die Gleichung $\boldsymbol{W} \cdot \boldsymbol{I} = 0$ teilt den gesamten Raum entlang einer Geraden (2D), einer Ebene (3D), einer Hyperebene (allg.) genau in zwei Bereiche:

- Ausgabe 1, wenn $\boldsymbol{W} \cdot \boldsymbol{I} \geq 0$
- Ausgabe 0, wenn $\boldsymbol{W} \cdot \boldsymbol{I} < 0$





Wie können Perzeptrons lernen?

Die meisten Lernverfahren folgen dem Ansatz der ***Current Best Hypothesis***: es wird immer nur eine Hypothese betrachtet, die optimiert wird.

- Dabei gilt: aktuelle Hypothese = aktuelles Netz
  = aktuelle Werte der Gewichte

- Start: Das Netzwerk werde mit zufällig gewählten Gewichten (i.A. aus dem Intervall [-0.5, 0.5]) initialisiert

- Iterative Optimierung: durch Ändern der Gewichte wird die Konsistenz mit den Trainingsbeispielen hergestellt
  - durch Reduzieren der Differenz zwischen aktuell berechnetem und vorgegebenem Wert
  - definiert in der sog. *Perzeptron-Lernregel*

Die Perzeptron-Lernregel

Sei T (für *target*) die vorgegebene korrekte Ausgabe und sei O die berechnete Ausgabe des aktuellen Netzes, dann ist der Fehler $Error = T - O$

Lernziel:

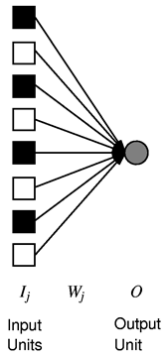
- ↗ wenn *Error* positiv ist, sollte O erhöht werden
- ↗ wenn *Error* negativ ist, sollte O erniedrigt werden

Da jede Eingabeeinheit I_j den Beitrag $W_j \cdot I_j$ zu O liefert, sollte zur Erhöhung von O das Gewicht W_j bei positivem I_j größer und bei negativem I_j kleiner werden.

Dies führt zur Perzeptron-Lernregel:

$$W_j \leftarrow W_j + \alpha \cdot I_j \cdot Error$$

mit positiver Konstante α, die Lernrate genannt wird.

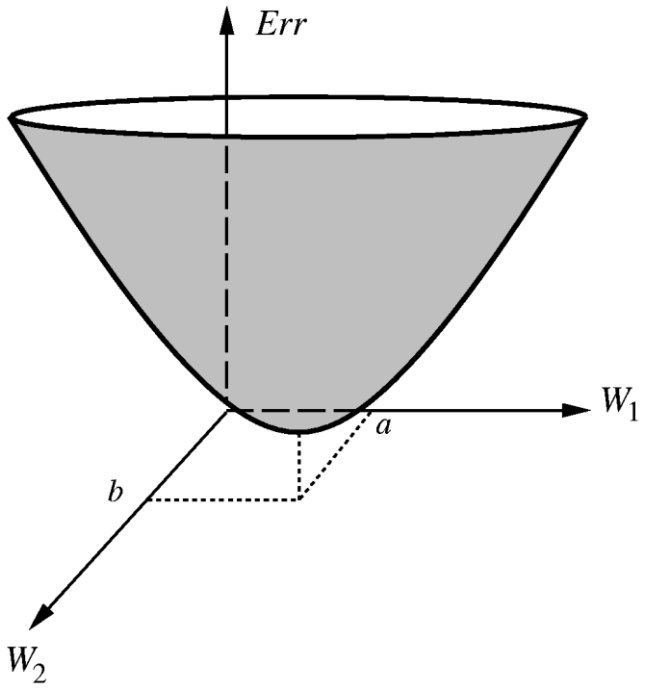


Satz [Rosenblatt 1960]: Bei linear separierbaren Funktionen konvergiert das Lernen mit der Perzeptron-Lernregel immer zu korrekten Werten.

Wieso funktioniert die Perzeptron-Lernregel?

- Perzeptron-Lernen entspricht *Gradientenabstieg* im Raum aller Gewichte, wobei der Gradient durch die *Fehlerfläche* bestimmt wird.
- Jede Gewichtskonfiguration bestimmt einen Punkt auf der Fläche.

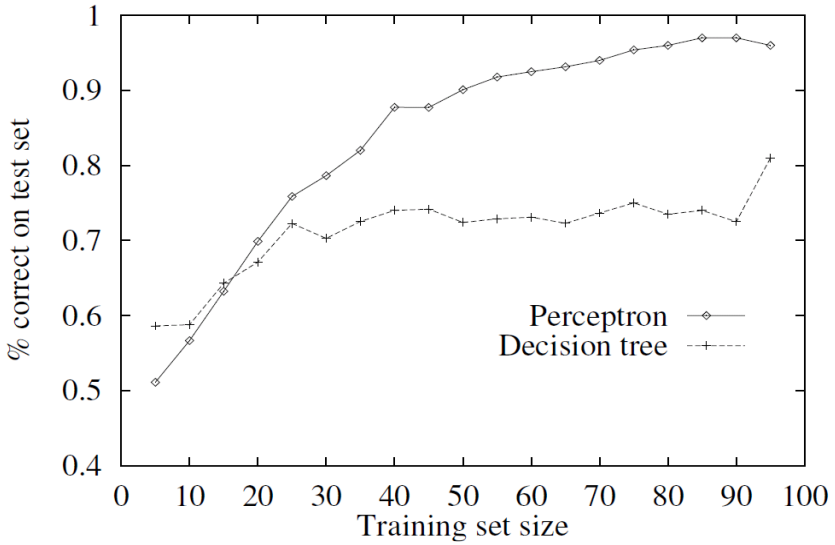
$\rightsquigarrow$ *Partielle Ableitung* bzgl. der einzelnen Gewichte.



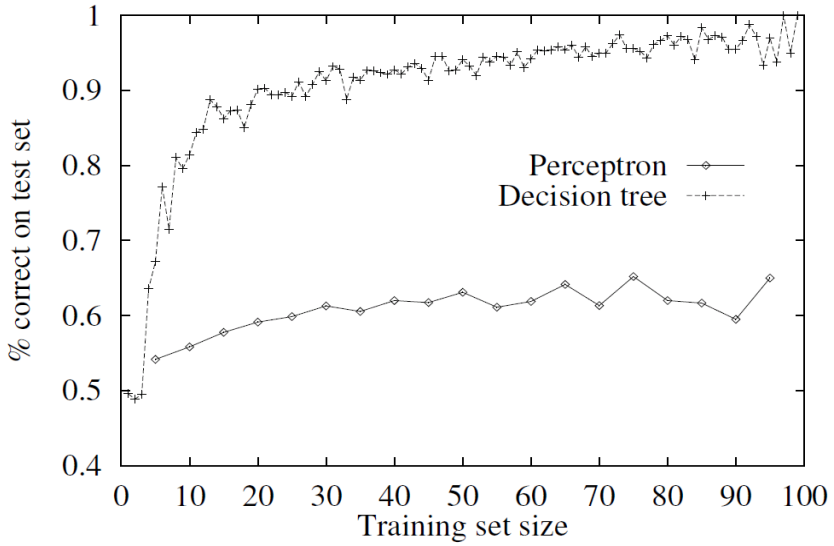
Da es bei linear separierbaren Funktionen keine lokalen Minima gibt, folgt der Perzeptron-Konvergenzsatz.

Lernkurven: Perzeptron vs. Entscheidungsbaum

Majoritätsproblem



Restaurantproblem



Majoritätsproblem (links): $O = 1$, wenn mind. $n/2$ von n Booleschen Eingaben wahr sind

↝ kompakte Darstellung bei Perzeptron mit $t = n/2$ und $w_j = 1$ für $j = 1, \ldots, n$.

↝ ungünstige Darstellung bei DT: Knotenzahl exponentiell in der Zahl der Attribute

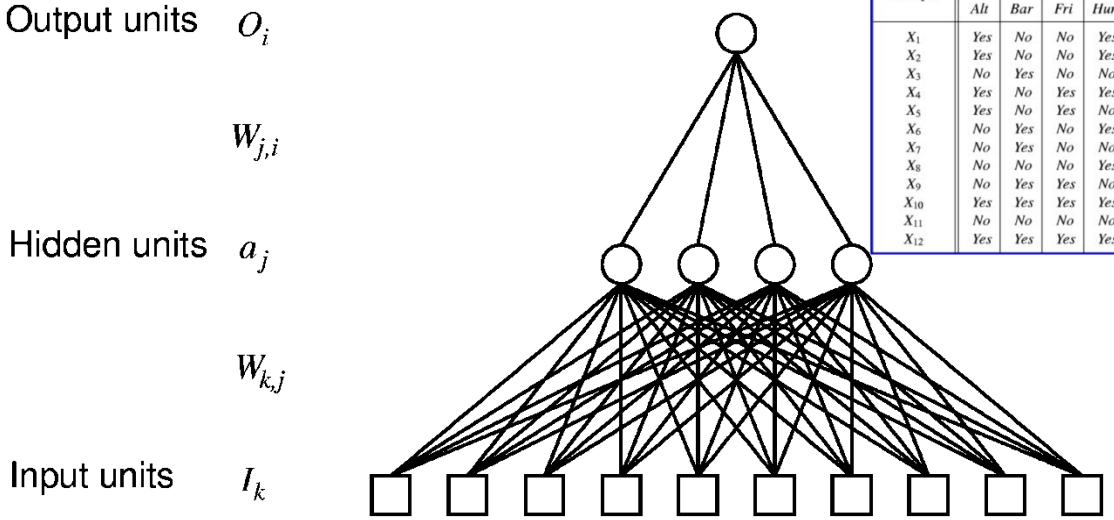
Restaurantproblem (rechts): nicht linear separierbar

↝ beste Trennungsebene bei Perzeptron mit maximaler Korrektheit von 65%.

↝ Multilayer-Feed-Forward-Netze als Ausweg?

Mehrschichten-FF-Netze

Beispiel:



Example	Attributes										Goal
	Alt	*Bar*	*Fri*	*Hun*	*Pat*	*Price*	*Rain*	*Res*	*Type*	*Est*	*WillWait*
X_1	*Yes*	*No*	*No*	*Yes*	*Some*	*$$$*	*No*	*Yes*	*French*	*0–10*	*Yes*
X_2	*Yes*	*No*	*No*	*Yes*	*Full*	*$*	*No*	*No*	*Thai*	*30–60*	*No*
X_3	*No*	*Yes*	*No*	*No*	*Some*	*$*	*No*	*No*	*Burger*	*0–10*	*Yes*
X_4	*Yes*	*No*	*Yes*	*Yes*	*Full*	*$*	*No*	*No*	*Thai*	*10–30*	*Yes*
X_5	*Yes*	*No*	*Yes*	*No*	*Full*	*$$$*	*No*	*Yes*	*French*	*>60*	*No*
X_6	*No*	*Yes*	*No*	*Yes*	*Some*	*$$*	*Yes*	*Yes*	*Italian*	*0–10*	*Yes*
X_7	*No*	*Yes*	*No*	*No*	*None*	*$*	*Yes*	*No*	*Burger*	*0–10*	*No*
X_8	*No*	*No*	*No*	*Yes*	*Some*	*$$*	*Yes*	*Yes*	*Thai*	*0–10*	*Yes*
X_9	*No*	*Yes*	*Yes*	*No*	*Full*	*$*	*Yes*	*No*	*Burger*	*>60*	*No*
X_{10}	*Yes*	*Yes*	*Yes*	*Yes*	*Full*	*$$$*	*No*	*Yes*	*Italian*	*10–30*	*No*
X_{11}	*No*	*No*	*No*	*No*	*None*	*$*	*No*	*No*	*Thai*	*0–10*	*No*
X_{12}	*Yes*	*Yes*	*Yes*	*Yes*	*Full*	*$*	*No*	*No*	*Burger*	*30–60*	*Yes*

Das Restaurantproblem als 2-Schichten-FF-Netz:

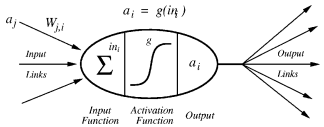
- 10 Eingaben I_k für die 10 Attribute
- 4 innere Knoten mit Aktivierungsniveaus a_i
- 1 Ausgabe O_i für das „*will wait*“ - Zielprädikat

- Bei Perzeptron: Ausgabeabweichung führt zu Korrektur des „verantwortlichen Gewichtes“ zwischen entsprechenden Eingabeknoten und dem Ausgabeknoten
- Bei Mehrschichten-Netz: Verteilte Fehlerverantwortlichkeit ⇝ Fehlerpropagation

Herleitung

Ziel: Minimierung von der Summe der Fehlerquadrate.

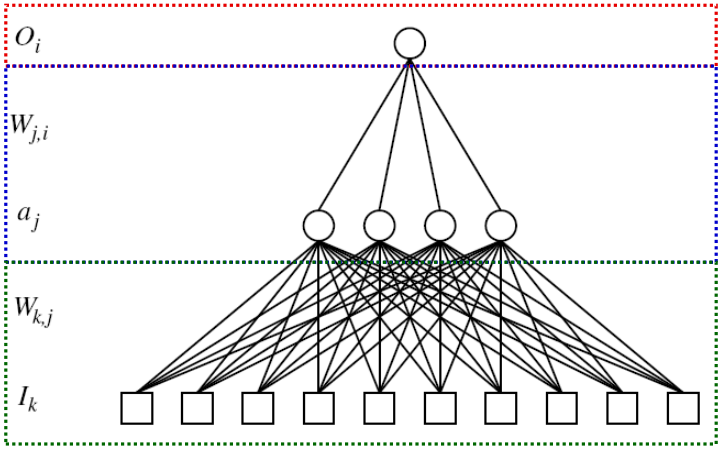
Mit Gewichtsvektor $\boldsymbol{W}$ ist die Fehlerfunktion:



$$E(\boldsymbol{W}) = \tfrac{1}{2} \sum_i (T_i - O_i)^2$$

$$= \tfrac{1}{2} \sum_i (T_i - g(\sum_j (W_{j,i} \cdot a_j)))^2 \quad (*)$$

$$= \tfrac{1}{2} \sum_i (T_i - g(\sum_j (W_{j,i} \cdot g(\sum_k (W_{k,j} \cdot I_k)))))^2$$



„Verantwortlichkeit der 1. Stufe“ (*): partielle Ableitung von E nach $W_{j,i}$: $\partial E / \partial W_{j,i}$

↝ Abzuleiten also: $\tfrac{1}{2} \sum_i (T_i - g(\sum_j (W_{j,i} \cdot a_j)))^2$

Nur jeweils ein Summand in den Summen über j ist von einem bestimmten W_{ji} abhängig. Die anderen Summanden verschwinden also als Konstante beim Differenzieren nach W_{ji}.

$$\partial E / \partial W_{j,i} = 2 \cdot \tfrac{1}{2} (T_i - g(\sum_j (W_{j,i} \cdot a_j))) \cdot - g'(\sum_j (W_{j,i} \cdot a_j)) \cdot a_j$$

$$= - (T_i - O_i) \cdot g'(in_i) \cdot a_j$$

$$= - a_j \cdot \Delta_i \qquad \text{mit } \Delta_i = (T_i - O_i) \cdot g'(in_i)$$

Analog kann man für $W_{k,j}$ zeigen: $\partial E / \partial W_{k,j} = - I_k \cdot \Delta_j$ mit $\Delta_j = g'(in_j) \cdot \sum_i W_{j,i} \cdot \Delta_i$

Lernen in Mehrschichten-Netzen: *Backpropagation* (1)

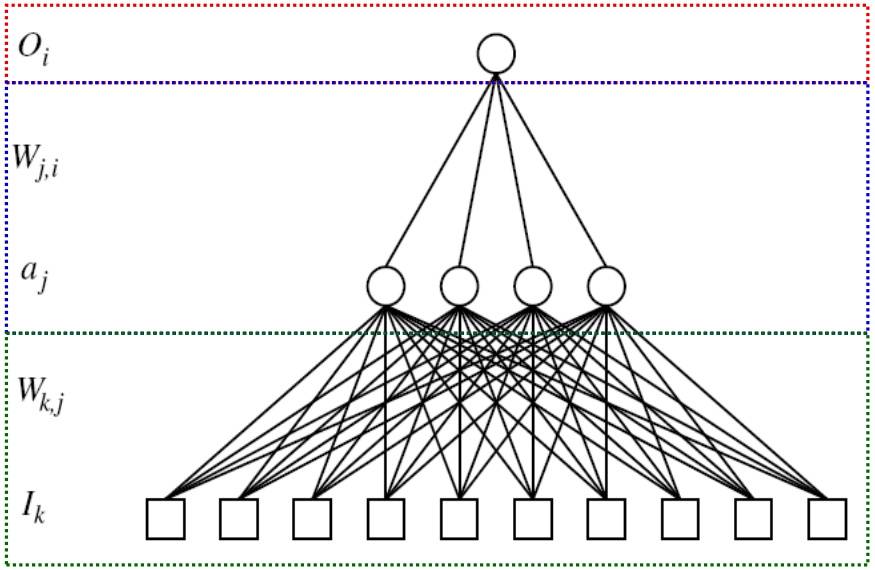
Für alle Gewichte wird ihr Anteil am Fehler bestimmt und die Gewichte entsprechend geändert, d.h. der Fehler wird rückwärts durchs Netz propagiert.

Für Gewichte $W_{j,i}$ zu Ausgabeknoten i:

$$W_{j,i} \leftarrow W_{j,i} + \alpha \cdot a_j \cdot \Delta_i$$

mit $\Delta_i = Error_i \cdot g'(in_i)$

und $Error_i = T_i - O_i$



Für Gewichte $W_{k,j}$ zu inneren Knoten j:

$$W_{k,j} \leftarrow W_{k,j} + \alpha \cdot I_k \cdot \Delta_j \quad \text{mit } \Delta_j = g'(in_j) \cdot \Sigma_i W_{j,i} \cdot \Delta_i$$

Da g differenzierbar sein muss, wählt man für g zum Bspl. die *Sigmoid*-Funktion.

Der Backpropagation-Algorithmus

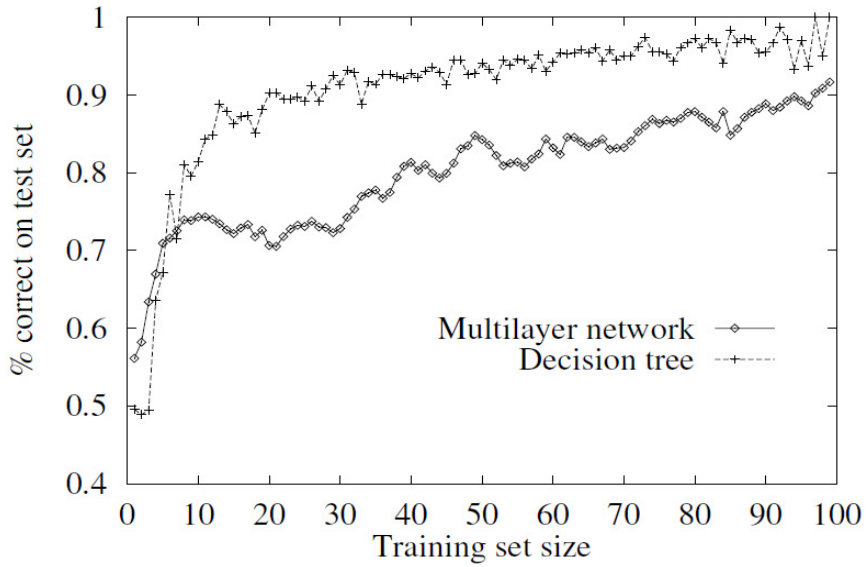
function BACK-PROP-LEARNING($examples$, $network$) **returns** a neural network
inputs: $examples$, a set of examples, each with input vector **x** and output vector **y**
$network$, a multilayer network with L layers, weights $w_{i,j}$, activation function g
local variables: Δ, a vector of errors, indexed by network node

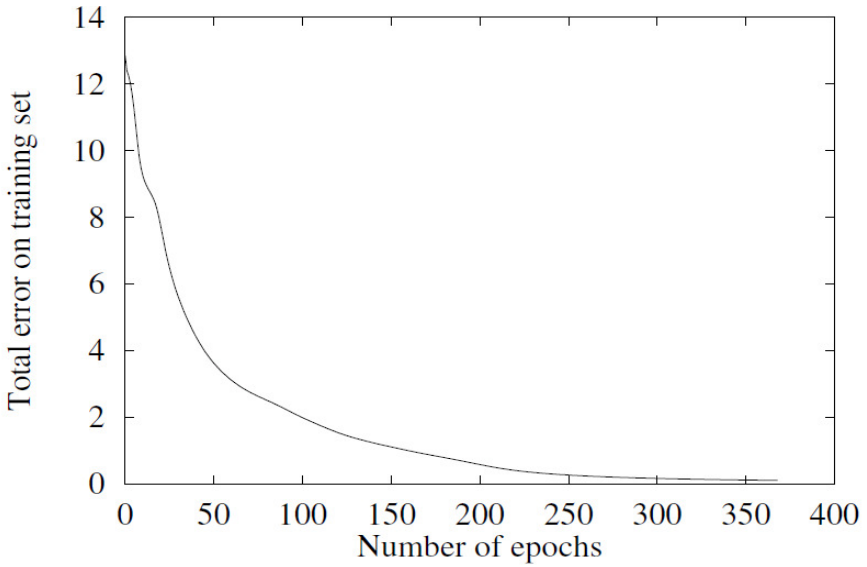
repeat
for each example (**x**, **y**) **in** $examples$ **do**
/* *Propagate the inputs forward to compute the outputs* */
for each node i in the input layer **do**
$a_i \leftarrow x_i$
for $\ell = 2$ **to** L **do**
for each node j in layer ℓ **do**
$in_j \leftarrow \sum_i w_{i,j} \; a_i$
$a_j \leftarrow g(in_j)$
/* *Propagate deltas backward from output layer to input layer* */
for each node j in the output layer **do**
$\Delta[j] \leftarrow g'(in_j) \times (y_j - a_j)$
for $\ell = L - 1$ **to** 1 **do**
for each node i in layer ℓ **do**
$\Delta[i] \leftarrow g'(in_i) \sum_j w_{i,j} \; \Delta[j]$
/* *Update every weight in network using deltas* */
for each weight $w_{i,j}$ in $network$ **do**
$w_{i,j} \leftarrow w_{i,j} + \alpha \times a_i \times \Delta[j]$
until some stopping criterion is satisfied
return $network$

Lernkurven

Lernkurve für das Restaurantbeispiel mit dem 2-Schichtennetz:

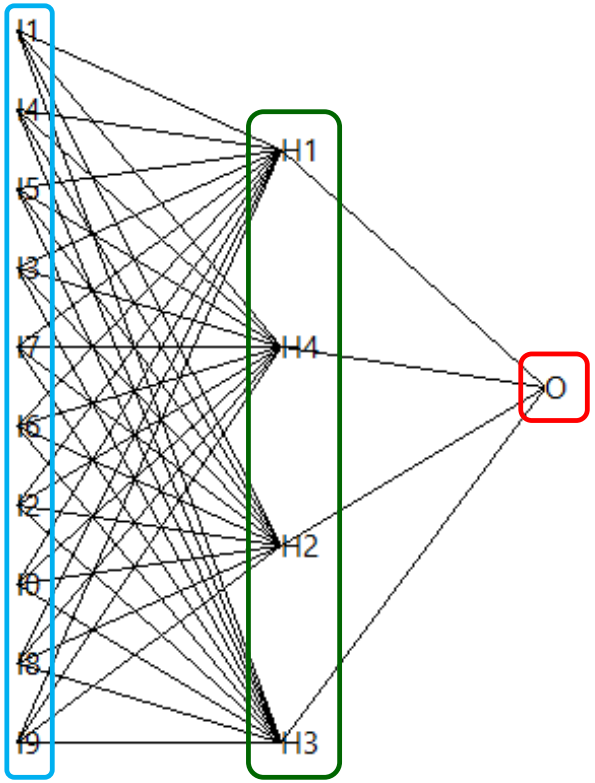
- links: im Vergleich mit der Lernkurve für das Entscheidungsbaum-Lernen,
- rechts: Fehler in Abhängigkeit von der Anzahl der Lernepochen bei Trainingsmenge mit 100 Beispielen.

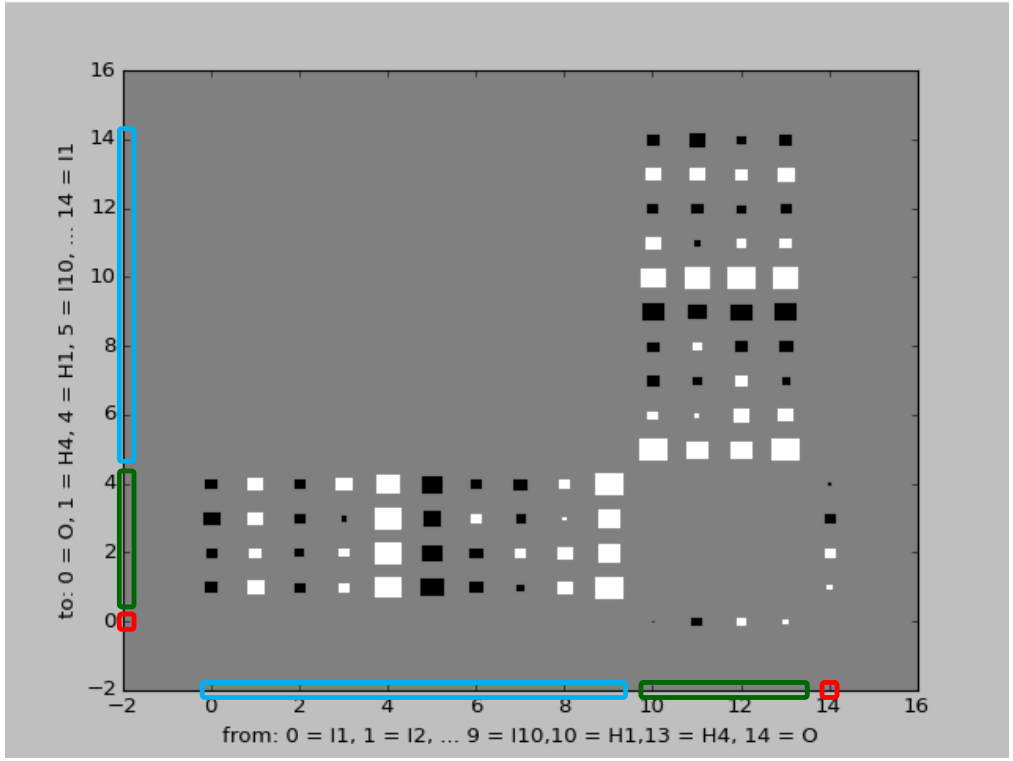




Hinton-Diagramme

Hinton-Diagramme dienen der Wertevisualisierung der Kantengewichte eines KNNs. Jedes Feld steht für ein Kantengewicht. Seine Größe ist proportional zum Wert des Kantengewichts. Seine Farbe (hier schwarz oder weiß) kodiert das Vorzeichen (weiß = positiv, schwarz = negativ).





Screenshots von AIMA-Py, s. Manual zu AIMA-Py, Institut für Informatik 4, Univ. Bonn, 2015.

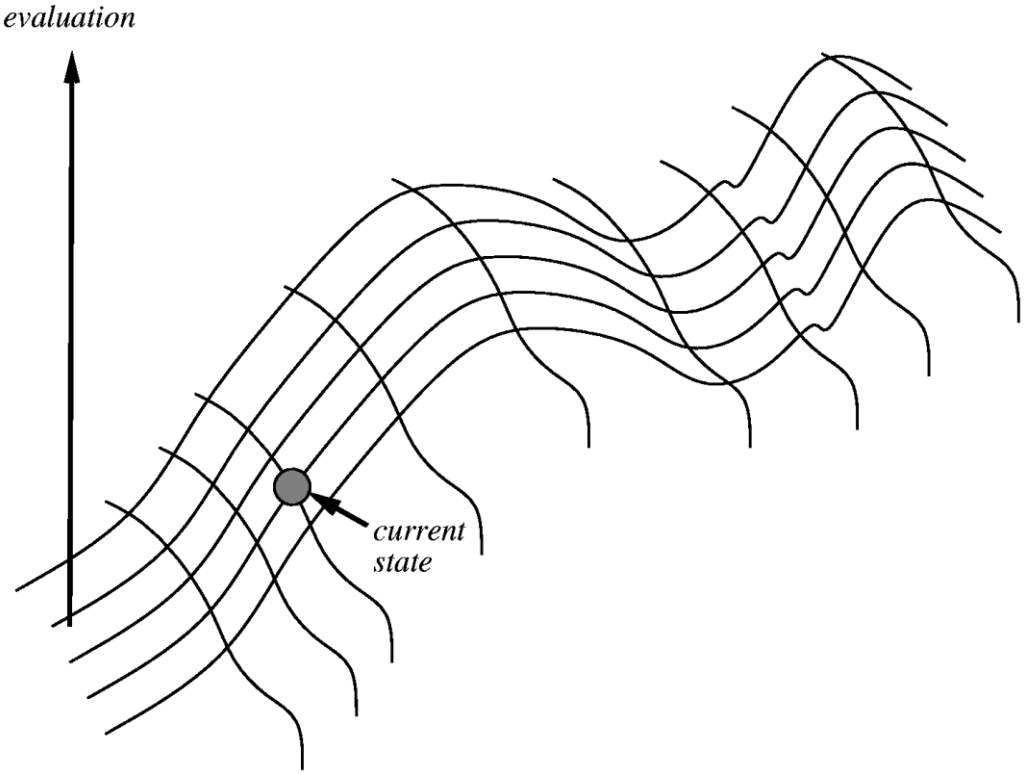
Backpropagation = Gradientenabstieg (1)

Backpropagation kann wieder als *Gradientenabstieg* im Raum aller Gewichte aufgefasst werden, wobei der Gradient durch die *Fehlerfläche* bestimmt wird.

Allerdings treten nun nicht mehr rein konvexe Fehlerflächen auf, sondern der Lernalgorithmus ist prinzipiell mit dem Problem *lokaler Minima* konfrontiert.

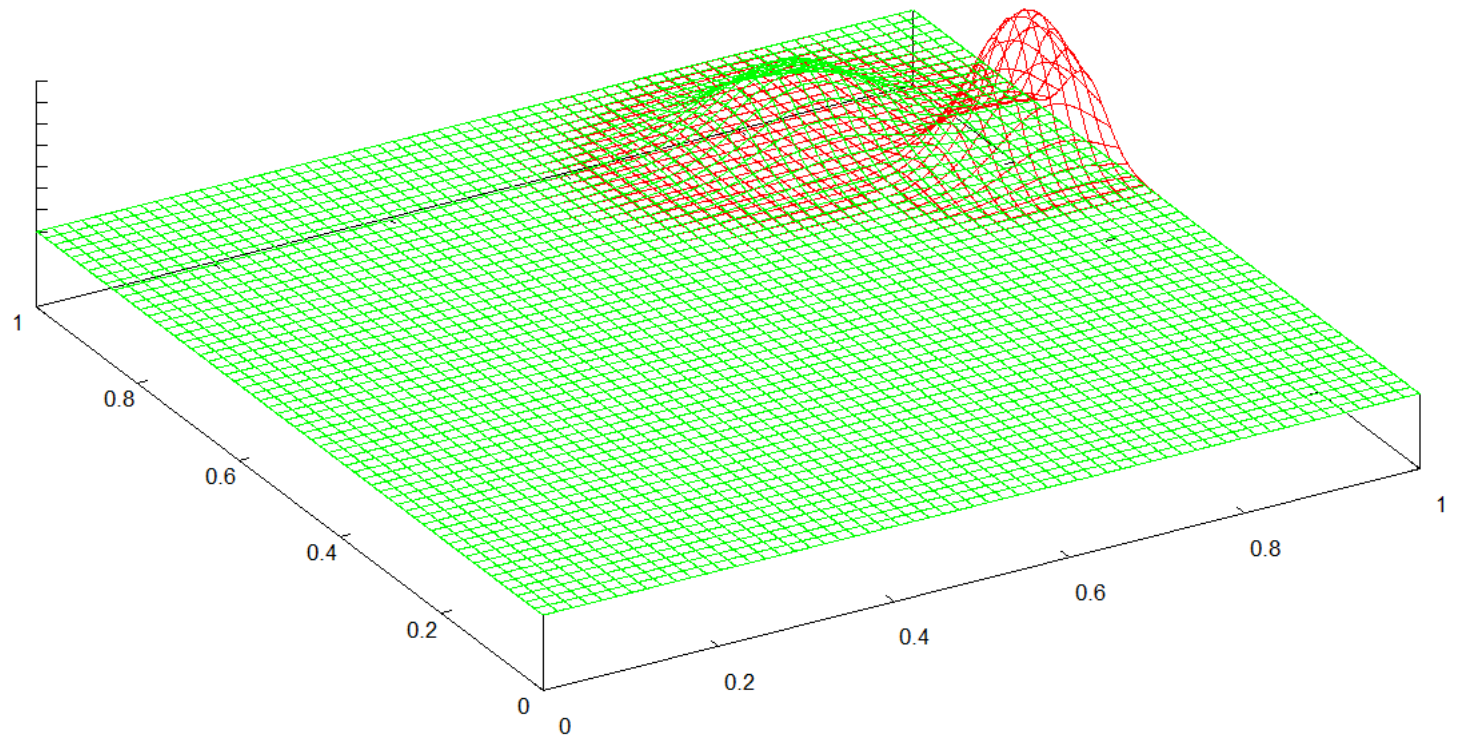
Prinzipielle Lösungen:

- ⇝ Neustarten
- ⇝ Tabusuche
- ⇝ Simulated Annealing



Backpropagation = Gradientenabstieg (2)

Untersuchungen (s.u.) zeigen aber auch, dass die zugrundeliegende Struktur der Daten (Verteilungen, Transformationen) für viele Anwendungen zu wohlgestellten Klassifikationsproblemen führen.



- Lin, H.W., Tegmark, M., Rolnick, D. (2016). Why does deep and cheap learning work so well? URL http://arxiv.org/abs/1608.08225 (13.06.2018).
- Swirszcz, G., Czarnecki, W. M., Pascanu, R. (2017). Local minima in training of neural networks. URL https://arxiv.org/abs/1611.06310 (13.06.2018).

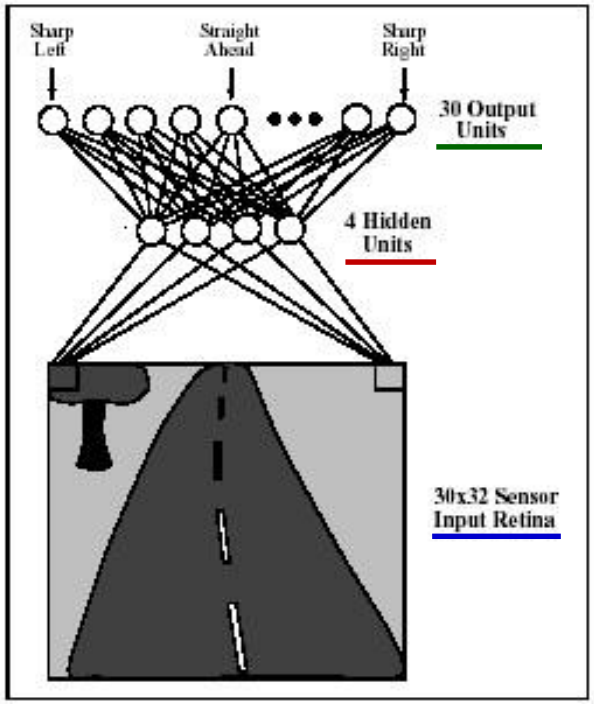
Komplexität, Generalisierung, Robustheit und Transparenz

- Wofür sind sie gut: Neuronale Netze eignen sich zum Lernen von Funktionen über Attributen, die auch *kontinuierlich* sein können
- Größe des Netzes: bis zu $2^n/n$ verborgene Einheiten mit 2^n Gewichten, um beliebige Boolesche Funktionen darzustellen, meist aber weniger.
- Effizienz des Lernens: Zeitkomplexität pro Epoche: $O(m \cdot |\mathbf{W}|)$ für m Beispiele und $|\mathbf{W}|$ Gewichte
- Generalisierung: gut, wenn man das richtige Netz gewählt hat … dafür aber keine Theorie … aber Heuristiken wie *Optimal Brain Damage*
- Rauschen: nichtlineare Regression ist sehr tolerant gegenüber Rauschen
- Transparenz: schlecht – oft weiß man nicht, warum das Netz gut funktioniert
  $\rightsquigarrow$ Black-Box-Ansatz

Beispielanwendung: Auto fahren (1)

- ALVINN [Pomerleau 93] steuert ein Fahrzeug auf einer Straße
- Eingabe von Videokamerabild, das zu einem 30x32-Bild für 960 Eingabeeinheiten transformiert wird
- Vier verborgene Einheiten sowie 30 Ausgabeeinheiten für die Steuerungskommandos

→ ca. 1000 Neuronen

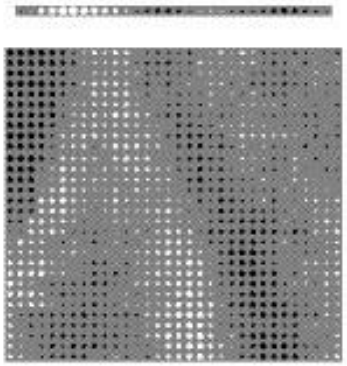


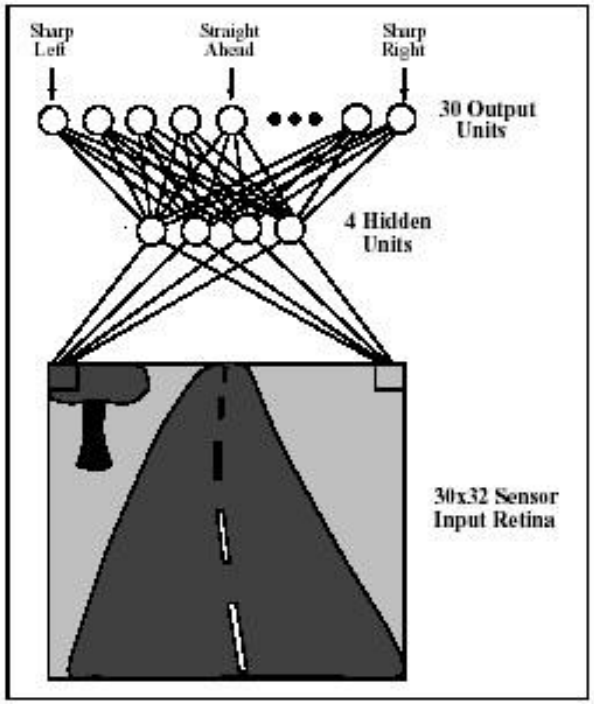
Dean Pomerleau: Knowledge-based Training of Artificial Neural Networks for Autonomous Robot Driving. Robot Learning, J. Connell and S. Mahadevan, ed., 1993.

Beispielanwendung: Auto fahren (2)

- Trainingsdaten werden durch menschlichen Fahrer gewonnen.
- Nach fünf Minuten Fahrt als Trainingsdaten kann ALVINN auf den trainierten Straßen bis zu 70 mph schnell fahren.

- Probleme: andere Straßentypen und andere Beleuchtungen.
- MANIAC [Jochem et al 93] ist ein Metasystem, welches das „richtige“ ALVINN-Netz für die aktuelle Straße auswählt.





Todd M. Jochem , Dean A. Pomerleau , Charles E. Thorpe: MANIAC: A Next Generation Neurally Based Autonomous Road Follower. Proc. Intern. Conf. on Intelligent Autonomous Systems (IAS), 1993.

Zusammenfassung & Ausblick

- Neuronale Netze sind ein Berechnungsmodell, das einige Aspekte des Gehirns nachbildet.
- Feed-forward-Netze sind die am einfachsten zu analysierenden Netze.
- Ein Perzeptron ist ein einschichtiges FF-Netz, mit dem man aber nur *linear separierbare* Funktionen repräsentieren kann. Mit Hilfe der Perzeptron-Lernregel können solche Funktionen erlernt werden.
- Mehrschichtige Netze können beliebige Funktionen darstellen.
- Backpropagation ist ein Verfahren, um Funktionen in solchen Netzen zu lernen, und das auf Gradientenabstieg beruht. In der Praxis kann man damit viele Probleme lösen, allerdings gibt es keine Garantien.