Neuronale Netzwerke

Anke Rieger

LS-8 Report 2

Universität Dortmund
Fachbereich Informatik
Lehrstuhl VIII - Künstliche Intelligenz
Postfach 500500
4600 Dortmund

Inhaltsverzeichnis

1 Einführung

Neuronale Netzwerke sind Modelle, die auf statistischen Verfahren basieren und zum Problemlösen und Lernen benutzt werden können.
Einige der ersten Resultate dieses Forschungsgebiets waren die Arbeiten von McCulloch und Pitts: '*A Logical Calculus of the Ideas Immanent in Neural Activity*" [McCulloch und Pitts, 1943], Minskys Dissertation: '*Neural Nets and the Brain Model Problem*' [Minsky, 1954] , und Rosenblatts Arbeiten, u.a. sein Buch '*Principles of Neurodynamics*' [Rosenblatt, 1962] . Die Euphorie dieser Jahre entsprang dem Glauben, ein Modell gefunden zu haben, mit dem sich das Gehirn und damit menschliche Intelligenz simulieren lasse. 1969 zeigten Minsky und Papert jedoch in ihrem Buch '*Perceptrons*' [Minsky und Papert, 1990] an welchen, oft relativ einfachen, Problemen das Perceptron scheitert. Dies hatte zur Folge, daß bis Anfang der 80-iger Jahre die Forschung auf diesem Gebiet praktisch zum Stillstand kam. Seither wurden neue Netzwerk-Modelle und Algorithmen entwickelt. Es seien hier nur einige Arbeiten genannt, wie die von Hopfield [Hopfield, 1982, 1984], von Kohonen [Kohonen, 1984] und die PDP-Bände (*Parallel Distributed Processing*) von Rumelhart und McClelland [Rumelhart und McClelland, 1986]. Seit 1987 findet jährlich eine von IEEE organisierte Konferenz '*International Conference on Neural Networks*' statt.
Neuronale Netze sind dynamische Systeme, die aus einer großen Anzahl einfacher Verarbeitungselemente bestehen, die parallell und unabhängig voneinander arbeiten können. Es sind fehler-tolerante Systeme mit verteilter Repräsentation und Kontrolle. In den folgenden Abschnitten wird eine kurze Einführung in die Terminologie gegeben, die im Zusammenhang mit neuronalen Netzen verwendet wird (siehe auch [Kemke, 1988]). Danach werden einige Netzwerkmodelle mit ihren Anwendungen und Lernalgorithmen vorgestellt.

2 Terminologie

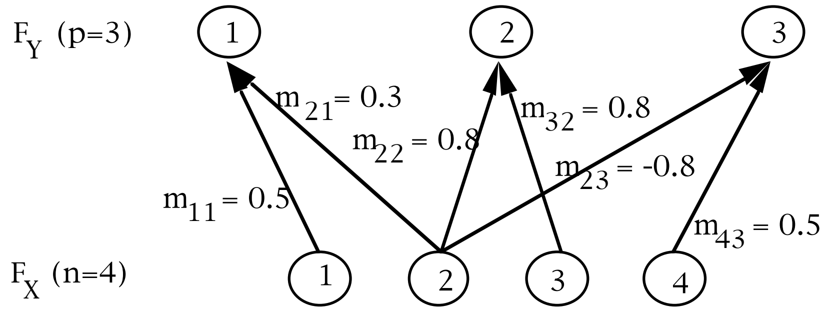
Neuronale Netzwerke bestehen aus
- einer bzw. mehreren Gruppen von **Einheiten** (*units*) und
- gewichteten **Verbindungen**, die die Neuronen miteinander verbinden.

Einheiten werden auch als Neuronen, Elemente, Knoten oder Verarbeitungs-elemente des Netzwerks bezeichnet, Verbindungen auch als Kanten.

Die Aufgabe einer Einheit i besteht darin, die Eingabe, die sie von ihren Nachbarn erhält, zu verarbeiten, entsprechend ihren Zustand zu ändern und eine Ausgabe über die von ihr ausgehenden Verbindungen an benachbarte Einheiten weiterzugeben. Dies geschieht unabhängig von allen anderen Einheiten, die somit parallel arbeiten können.
Die **Eingabe** einer Einheit i ist eine reelle Zahl, die mit $net_i \in \Re$ ($\Re$: Menge der reellen Zahlen) bezeichnet wird.
Der (Aktivierungs-) **Zustand** eines Neurons i zum Zeitpunkt t ist ebenfalls eine reelle Zahl und wird mit $x_i(t)$ bezeichnet. Wenn nichts anderes gesagt wird, nehmen wir im folgenden an, daß $x_i(t)=net_i(t)$.
Die **Ausgabefunktion** $S(x_i(t))$ bestimmt, welche Ausgabe eine Einheit i in Abhängigkeit von ihrem Zustand an ihre Nachbarn weitergibt. Das Verhalten von Neuronen kann also auch funktional beschrieben werden: Eine Eingabe wird in ein Ausgabesignal $S(x(t))$ überführt.

Die Einheiten sind durch gerichtete Verbindungen miteinander verbunden, die mit Gewichten versehen sind. Das **Gewicht** einer Verbindung von Einheit i zu Einheit j ist eine reelle Zahl und wird mit $m_{ij} \in \Re$ bezeichnet.

Betrachten wir folgendes Netzwerk, bestehend aus zwei Ebenen:



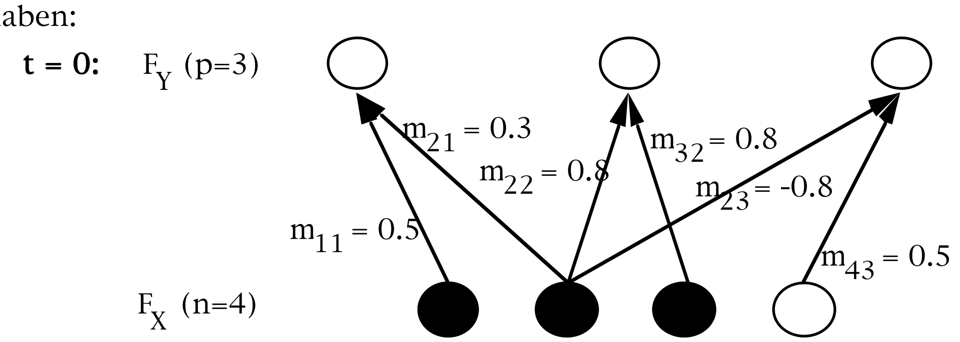
Die untere Ebene wird Eingabeebene genannt. Ihre n Einheiten werden in einer Gruppe zusammengefaßt, die mit F_X bezeichnet wird. Die obere Ebene wird Ausgabeebene genannt. Ihre p Einheiten werden in der Gruppe F_Y zusammengefaßt. Die Einheiten aus F_X sind durch gerichtete, mit Gewichten versehenen Verbindungen mit den Einheiten aus F_Y verbunden. So gehen z.B. von der zweiten Einheit aus F_X drei Verbindungen aus zu jeder der drei Ausgabeeinheiten: das Gewicht m_{21} der Verbindung von der zweiten Eingabeeineheit zur ersten Ausgabeeinheit hat den Wert 0.3, das Gewicht m_{22} der Verbindung zur zweiten Ausgabeeinheit hat den Wert 0.8 und das Gewicht m_{23} der Verbindung zur dritten Ausgabeeinheit hat den Wert -0.8.
Nehmen wir an, daß die Einheiten von F_X ihre Ausgabe nach der Funktion $S(x_i)=x_i$, i=1,...,n, und die Ausgabeeinheiten ihre Ausgabe nach der Schwellwertfunktion

$$S(y_j^t) = \begin{cases} 1, & \text{wenn } y_j^t > 0 \\ 0, & \text{wenn } y_j^t \leq 0 \end{cases}, \qquad j=1,\ldots,p.$$

berechnen, d.h. die Ausgabe einer Ausgabeeinheit ist Eins, wenn der Zustand der Einheit j zum Zeitpunkt t größer als Null war, sonst ist sie Null (Die Ausdrücke x_i^t und y_j^t sind gleichbedeutend mit $x_i(t)$ und $y_j(t)$).
Nehmen wir weiterhin an, daß die Einheiten zum Zeitpunkt t=0 den Zustand

$$X(0) = (\ x_1(0)=1\ \ x_2(0)=1\ \ x_3(0)=1\ \ x_4(0)=0\ )$$

haben:



Zum Zeitpunkt t=1 berechnen alle Eingabeeinheiten ihre Ausgabe

$$S(X(0))=(\ S(x_1(0))=1\ \ S(x_2(0))=1\ \ S(x_3(0))=1\ \ S(x_4(0))=0\ )$$

und propagieren sie über die gewichteten Verbindungen zu den Einheiten der Ausgabeebene. Diese erhalten als Eingabe net_j, j=1,...,p, die gewichtete Summe der Ausgaben der vorgeschalteten Eingabeeinheiten:

$$net_j = \sum_{i=1}^{n} S(x_i)m_{ij}, \quad j=1,\ldots,m.$$

Alle Ausgabeeinheiten ändern gleichzeitig, d.h. synchron, ihren Zustand nach der Regel $y_j(t)=net_j(t)$:

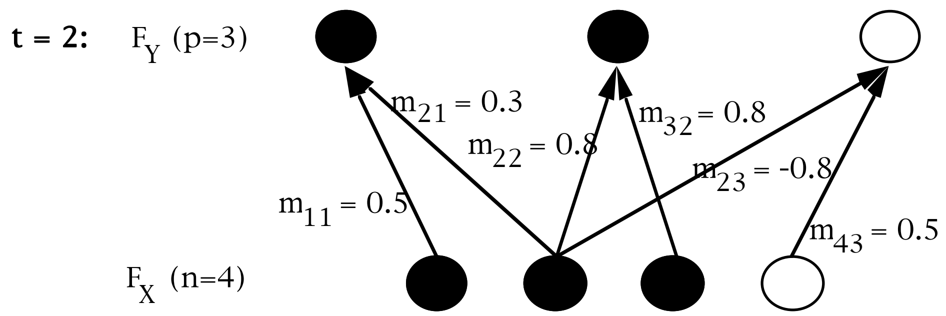
$$y_1 = S(x_1)m_{11} + S(x_2)m_{21} = 0.5*1 + 0.3*1 = 0.8$$
$$y_2 = S(x_2)m_{22} + S(x_3)m_{32} = 0.8*1 + 0.2*1 = 1.0$$
$$y_3 = S(x_2)m_{23} + S(x_4)m_{43} = -0.8*1 + 0.5*0 = -0.8.$$

Zum Zeitpunkt t=1 haben die Ausgabeeinheiten also den Zustand

$$Y(1) = (y_1(1)=0.8 \;\; y_2(1)=1.0 \;\; y_3(1)=-0.8 \;)$$

Zum Zeitpunkt t=2 berechnen sie ihre Ausgabe S(Y) nach der Schwellwertfunktion:

$$S(Y(1)) = (\; S(y_1(1))=1 \; S(y_2(1))=1 \;\; S(y_3(1))=0 \;)$$



Da von ihnen keine Verbindungen ausgehen, endet der Propagierungsprozeß hier.

Fassen wir die anhand des Beispiels erläuterte Notation zusammen:

F_X:	Gruppe der n Einheiten der Eingabeebene
F_Y:	Gruppe der p Einheiten der Ausgabeebene
$x_i(t) \in \Re$:	Zustand der Einheit i aus F_X zum Zeitpunkt t mit $i \in \{1,...,n\}$
$y_j(t) \in \Re$:	Zustand der Einheit j aus F_Y zum Zeitpunkt t mit $j \in \{1,...,p\}$
$X(t)=(x_1(t) \;...\; x_n(t))$:	Zustandsvektor für F_X
$Y(t)=(y_1(t) \;...\; y_p(t))$:	Zustandsvektor für F_Y
$S(x_i)$, $S(y_j) \in \Re$:	Ausgabe/Signal der Einheit i bzw j
S(X), S(Y):	Ausgabe-/Signalvektoren für F_X, F_Y
$m_{ij} \in \Re$:	Gewicht der Verbindung von Einheit i aus F_X zu Einheit j aus F_Y.

(Vektoren werden als Zeilenvektoren aufgefaßt.)

<u>Funktionales Verhalten eines Netzwerks:</u>
Das Verhalten des gesamten Netzwerks kann auch durch eine Funktion

$$f: \Re^n \rightarrow \Re^p$$

beschrieben werden, die einen n-dimensionalen Eingabevektor auf einen p-dimensionalen Ausgabevektor abbildet. Im Beispiel haben wir f: $\Re^4 \rightarrow \Re^3$. Welche Form hat nun diese Funktion? Wir können die Gewichte der Verbindungen des Netzwerks in einer Matrix M zusammenfassen:

$$
\begin{array}{c|c|ccc}
 & i \backslash j & 1 & 2 & 3 \\
\hline
 & 1 & 0.5 & 0 & 0 \\
F_X & 2 & 0.3 & 0.8 & -0.8 \\
 & 3 & 0 & 0.2 & 0 \\
 & 4 & 0 & 0 & 0.5
\end{array}
\quad .
$$

(Spalten 1–3: F_Y)

Die Indizes für die Zeilen repräsentieren die Eingabeeinheiten von F_X, die Spaltenindizes die Ausgabeeinheiten von F_Y, d.h. das Element m_{ij} repräsentiert, wie oben definiert, das Gewicht der Verbindung von Einheit $i\in\{1,...,n\}$ zu Einheit $j\in\{1,...,p\}$.
Die Ausgabe der Eingabeeinheiten zum Zeitpunkt t=1 war

$$S(X) = (1\ 1\ 1\ 0).$$

Daraus ergab sich für die Ausgabeeinheiten der Zustand

$$
\begin{aligned}
Y &= \left(y_1=\sum_{i=1}^{n} S(x_i)m_{i1} \quad y_2=\sum_{i=1}^{n} S(x_i)m_{i2} \quad y_3=\sum_{i=1}^{n} S(x_i)m_{i3} \quad y_4=\sum_{i=1}^{n} S(x_i)m_{i4} \right) \\
&= S(X)\ M.
\end{aligned}
$$

Der Zustandsvektor der Ausgabeeinheiten zum Zeitpunkt t=1 ist damit das Ergebnis der Matrixmultiplikation des Ausgabevektors von F_X mit der 4x3-Matrix M.
Für unser Beispiel erhalten wir also

$$
\begin{aligned}
f: \quad & \Re^4 \rightarrow \Re^3 \\
& X_{1x4} \rightarrow Y_{1x3} = S(X)_{1x4}\ M_{4x3}
\end{aligned}
$$

bzw. für den allgemeinen Fall:

$$
\begin{aligned}
f: \quad & \Re^n \rightarrow \Re^p \\
& X_{1xn} \rightarrow Y_{1xp} = S(X)_{1xn}\ M_{nxp}.
\end{aligned}
$$

Im einfachsten Fall betrachten wir Netzwerke bestehend aus 2 Ebenen. Wir fassen die n Einheiten der unteren Ebene in einer Gruppe F_X zusammen und bezeichnen sie als Eingabeeinheiten. Die p Einheiten der oberen Ebene fassen wir in der Gruppe F_Y zusammen und bezeichnen sie als Ausgabeeinheiten.
Der Zustand eines Netzwerks zum Zeitpunkt t wird durch Zustandsvektoren für die Felder F_X und F_Y dargestellt: $X(t)=(x_1(t),...,x_n(t))$ und $Y(t)=(y_1(t),...,y_p(t))$ mit $x_i(t),y_j(t)\in\Re$. Der **Zustandsraum** des Feldes F_X ist der n-dimensionale Raum $\Re^n$. Entsprechend ist $\Re^p$ der Zustandsraum für F_Y. Ein neuronales Netzwerk assoziiert bzw. bildet Eingabevektoren **x** auf Ausgabevektoren **y** ab, d.h. das Verhalten des ganzen Netzwerks kann, wie das Verhalten einzelner Neuronen, funktional beschrieben werden: $f:\Re^n\rightarrow\Re^p$.
Werden die Felder F_X und F_Y konkateniert, so entsteht ein einziges Feld $F_Z=[F_X|F_Y]$ mit dem Zustandsraum $\Re^{n+p}$. Diese Konkatenation konvertiert ein **heteroassoziatives** Netzwerk in ein **autoassoziatives** Netzwerk.

Der **Signalraum** eines Feldes F_X mit n Einheiten besteht aus allen möglichen n-dimensionalen Ausgabe-/Signalvektoren $S(X(t))=(S_1(x_1(t))...S_n(x_n(t)))$, wobei $S_i(x_i(t))$ die Ausgabefunktion des i-ten Neurons darstellt. Wenn nichts anderes gesagt wird, wird im folgenden angenommen, daß die Ausgabefunktion für alle Einheiten einer Gruppe/Ebene die gleiche ist. Damit die Ausgabe nicht unendlich groß wird, wird in der Regel gefordert, daß die Ausgabefunktion nach oben hin beschränkt ist. Weiterhin wird gefordert, daß Ausgabe- bzw. Signalfunktionen monoton steigend sind, d.h. $S' \geq 0$. Größere Aktivierungen x können die Ausgabe also entweder vergrößern oder lassen sie unverändert.
Beispiele für Ausgabefunktion sind die logistische Funktion

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

mit der positiven, reellen Skalierungskonstante $c>0$ und $S'(x)=cS(1-S)>0$ und die Schwellwertfunktion

$$S(x^{t+1}) = \begin{cases} 1, & \text{wenn } x^{t+1}>T \\ S(x^t), & \text{wenn } x^{t+1}=T \\ 0, & \text{wenn } x^{t+1}<T \end{cases}$$

mit dem Schwellwert $T \in \Re$. Schwellwertfunktion und logistische Funktion werden identisch für $c \rightarrow \infty$. In der Praxis werden oft binäre Ausgabefunktionen mit $S(x) \in \{0,1\}$ oder bipolare Ausgabefunktionen mit $S(x) \in \{-1,1\}$ verwendet.
Wenn die Ausgabefunktionen beschränkt sind, ist der Signalraum einer Gruppe von Einheiten ein n- bzw. p-dimensionaler Würfel.
Im Zusammenhang mit Boltzmann-Maschinen werden probabilistische Ausgabefunktionen verwendet, die die Form

$$p(S(x)=1) = \frac{1}{1 + \exp(\frac{-\Delta E}{T})}$$

mit

ΔE: Differenz der "Energie" des Netzwerks zwischen zwei aufeinanderfolgenden Zeitpunkten;
$T \in \Re$: Temperatur

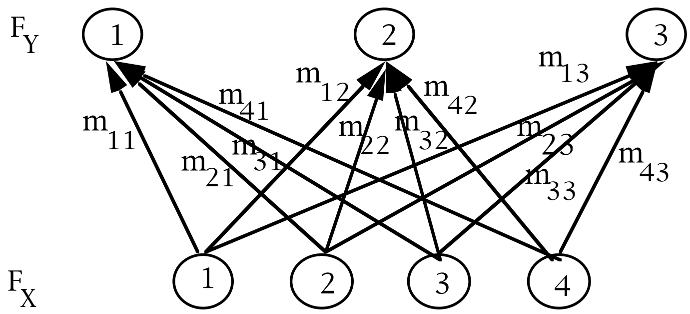
haben. Die Verwendung dieser Ausgabefunktion in Boltzmann-Maschinen wird im Zusammenhang von Simulated Annealing zur Lösung von Optimierungsproblemen beschrieben.

Die Neuronen eines Netzwerks sind durch gewichtete Kanten miteinander verbunden. Das Gewicht der gerichteten Verbindung von Einheit i nach Einheit j wird mit m_{ij} bezeichnet und hat als Wert eine reelle Zahl: $m_{ij} \in \Re$. Wenn $m_{ij}>0$, dann wird die Verbindung als **exitatorisch**, wenn $m_{ij}<0$ als **inhibitorisch** bezeichnet. Im folgenden bezeichnen wir mit **M** die Matrix der Gewichte der Verbindungen von Einheiten aus F_X zu Einheiten aus F_Y, mit **N** die Matrix der Gewichte der Verbindungen von Einheiten aus F_Y zu Einheiten aus F_X, mit **P** die Matrix der Gewichte der Verbindungen von Einheiten aus F_X untereinander und mit **Q** die Matrix der Gewichte der Verbindungen von Einheiten aus F_Y untereinander. Die Verbindungsmatrix für das konkatenierte Netzwerk $F_Z=[F_X|F_Y]$ sieht dann so aus:

$$\begin{array}{c|cccc|cccc}
 & x_1 & x_2 & .. & x_n & y_1 & y_2 & .. & y_p \\
\hline
\begin{matrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{matrix} & & \mathbf{P} & & & & \mathbf{M} & & \\
\hline
\begin{matrix} y_1 \\ y_2 \\ \vdots \\ y_p \end{matrix} & & \mathbf{N} & & & & \mathbf{Q} & &
\end{array}$$

Wenn die Matrizen P, N und Q Nullmatrizen sind, sprechen wir von Netzwerken mit ausschließlich vorwärtsggerichteten Verbindungen (***feedforward* -Netz**).

Feedforward-Netz mit n=4 Eingabeeinheiten und p=3 Ausgabeeinheiten:



Falls $N \neq 0$, haben wir es mit ***feedback*-Netzen** zu tun. **Bidirektionale Netzwerke** sind die einfachste Form von *feedback*-Netzwerken. Für sie gilt: Die Matrizen P und Q sind Nullmatrizen und $M=N^T$ und $N=M^T$, wobei M^T und N^T die transponierten Matrizen von M und N sind. Ein Spezialfall von bidirektionalen Netzwerken sind unidirektionale Netze mit $F_X=F_Y$ und der quadratischen nxn-Matrix M. Bidirektionale und unidirektionale Netzwerke sind Spezialfälle von heteroassoziativen bzw. autoassoziativen Netzen.

Die Zustandsvektoren X und Y zu einem bestimmten Zeitpunkt t werden als **Kurzzeitgedächtnis** des Netzwerks bezeichnet. Dagegen werden die Gewichte der Verbindungen als **Langzeitgedächtnis** des Netzwerks betrachtet.

Neuronale Netzwerke sind **dynamische Systeme**, d.h. ihre Zustände und Gewichte ändern sich mit der Zeit und in Abhängigkeit von Parametern des Netzwerks. Sie können somit durch eine Menge von Differentialgleichungen beschrieben werden:

$$\begin{aligned}
\dot{x}_i &= e_i(X,Y,\ldots),\ i=1,\ldots,n \\
\dot{y}_j &= g_j(X,Y,\ldots),\ j=1,\ldots,p \\
\dot{m}_{ij} &= h_{ij}(X,Y,\ldots),\ i=1,\ldots n;\ j=1,\ldots,p,
\end{aligned}$$

wobei $\dot{x}_i$, $\dot{y}_j$ und $\dot{m}_{ij}$ die zeitlichen Ableitungen von x_i, y_j und m_{ij} sind.

Wir betrachten zuerst den Fall, daß $\dot{m}_{ij}=0$ ist, d.h. die Gewichte bleiben konstant. Später werden wird Lernalgorithmen für Netzwerke kennenlernen, in denen die Gewichte modifiziert werden, d.h. $\dot{m}_{ij} \neq 0$.
Additive Modelle für die Zustandsänderungen von Einheiten von Netzwerken mit 2 Ebenen, F_X und F_Y, mit den Verbindungsmatrizen M und N haben die folgende Form:

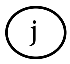
$$\dot{x}_i = -A_i x_i + \sum_{j=1}^{p} S_j(y_j) n_{ji} + I_i$$

$$\dot{y}_j = -A_j y_j + \sum_{i=1}^{n} S_i(x_i) m_{ij} + J_j$$

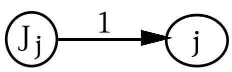
mit

- $A_i, A_j \in \Re$: konstanter reeller Faktor > 0
- $I_i, J_j \in \Re$: konstante Eingabe der Eingabeeinheit $i \in \{1,\ldots,n\}$ bzw. der Ausgabeeinheit $j \in \{1,\ldots,p\}$
- $S_i(x_i)$, $S_j(y_j)$: Ausgabefunkion der i-ten Eingabe bzw. j-ten Ausgabeeinheit
- $m_{ij} \in \Re$: Gewicht der Verbindung von Eingabeeinheit i aus F_X zu Ausgabeeinheit j aus F_Y (Element der i-ten Zeile und j-ten Spalte der Matrix M)
- $n_{ji} \in \Re$: Gewicht der Verbindung von Ausgabeeinheit j aus F_Y zu Eingabeeinheit i aus F_X (Element der j-ten Zeile und i-ten Spalte der Matrix N)

Beispiele:
Nehmen wir eine isolierte Einheit j, die zu einem Zeitpunkt t=0 einen Impuls $y_j(0)$ bekommt und dann mit der Zeit ein Ruhepotential erreicht. Eine solche Einheit wird durch den ersten Summand der obigen Differentialgleichungen beschrieben. Der Faktor A_j/A_i bewirkt eine Verschiebung der e-Funktion parallel zur y-Achse:

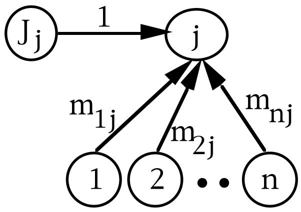
Netz	Differentialgleichung	Lösung
	$\dot{y}_j = - A_j y_j$	$y_j(t) = A_j\, y_j(0)\, e^{-t}$

Nehmen wir nun an, daß die Einheit j neben dem initialen Impuls eine konstante Eingabe J_j erhält. Dann wird das dynamische Verhalten dieser Einheit zusätzlich durch den zweiten Summanden beschrieben:

Netz	Differentialgleichung	Lösung
	$\dot{y}_j = - A_j y_j + J_j$	$y_j(t)= y_j(0)\, e^{-t} + J_j(1-e^{-t})$ mit $A_j=1$

Betrachten wir nun noch den Fall, daß die Einheit j von n benachbarten Einheiten die Eingabe $net_j = \sum_{i=1}^{n} S_i(x_i) m_{ij}$ erhält:

Netz



Differentialgleichung

$$\dot{y}_j = -A_j y_j + \sum_{i=1}^{n} S_i(x_i) m_{ij} + J_j$$

Der Zustand eines Verarbeitungselements ändert sich also im allgemeinen Fall in Abhängigkeit von dem aktuellen Zustand, der Eingabe (Summe der gewichteten Ausgaben der direkt mit ihm verbundenen Neuronen) und einer konstanten Eingabe.

Spezialfälle sind das additive Modell für bidirektionale Netze ($M=N^T$, $N=M^T$) mit

$$\dot{x}_i = -A_i x_i + \sum_{j=1}^{p} S_j(y_j) m_{ij} + I_i \qquad \text{und} \qquad \dot{y}_j = -A_j y_j + \sum_{i=1}^{n} S_i(x_i) m_{ij} + J_j$$

und das additive Modell für autoassoziative, unidirektionale ($F_X=F_Y$) Netze mit n Differentialgleichungen erster Ordnung

$$\dot{x}_i = -A_i x_i + \sum_{j=1}^{n} S_j(y_j) m_{ji} + I_i \quad .$$

Ist die Ausgabefunktion S eine Schwellwertfunktion, haben die Neuronen nicht kontinuierliche, sondern diskrete Ausgaben, die mit AN oder AUS bezeichnet werden und im binären Fall durch die Werte 0 und 1, im bipolaren Fall durch die Werte -1 und 1 dargestellt werden. Dieser Typ von Einheit wird auch **McCulloch-Pitts Neuron** [McCulloch und Pitts, 1943] genannt.
Diskrete bidirektionale Netzwerke mit den Schwellwertfunktionen

$$S_i(x_i^{t+1}) = \begin{cases} 1, & \text{wenn } x_i^{t+1} > U_i \\ S_i(x_i^t), & \text{wenn } x_i^{t+1} = U_i \\ 0, & \text{wenn } x_i^{t+1} < U_i \end{cases} \qquad \text{und} \qquad S_j(y_j^{t+1}) = \begin{cases} 1, & \text{wenn } y_j^{t+1} > V_j \\ S_j(y_j^t), & \text{wenn } y_j^{t+1} = V_j \\ 0, & \text{wenn } y_j^{t+1} < V_j \end{cases}$$

mit den reellwertigen Schwellwerten $U=(U_1 \ldots U_n)$ und $V=(V_1 \ldots V_p)$ haben folgende Lösung für die oben genannten Differentialgleichungen mit $A_i=A_j=0$:

$$x_i^{t+1} = \sum_{j=1}^{p} S_j(y_j^t) m_{ij} + I_i \qquad \text{und} \qquad y_j^{t+1} = \sum_{j=1}^{n} S_i(x_i^t) m_{ij} + J_j \quad .$$

Je nachdem, ob alle Einheiten gleichzeitig oder immer nur eine Teilmenge der Einheiten zu einem Zeitpunkt t ihren Zustand ändert, sprechen wir von Netzwerken mit **synchronen** bzw. **asynchronen** Verarbeitungsmodus.

Neuronale Netzwerke können somit anhand von vier Kriterien charakterisiert werden:

1. Einheiten
   - McCulloch-Pitts-Neuron (diskret)
   - Logistische Ausgabefunktion (kontinuierlich)
   - Probabilitische Ausgabefunktion
2. Verarbeitungsmodus der Einheiten
   - synchron
   - asynchron
3. Verbindungsstruktur
   - *feedforward* (N=P=Q=0)
   - *feedback* (M≠0, N≠0)
4. Anzahl der Ebenen
   - autoassoziativ (eine Ebene)
   - heteroassoziativ (mehr als eine Ebene)

3 Anwendungen von neuronalen Netzwerken

Wir werden zwei Anwendungen von neuronalen Netzwerken betrachten: Mustererkennung und die Lösung von kombinatorischen Optimierungsproblemen. Für diese beiden Anwendungen gibt es Verfahren, mit deren Hilfe die Gewichte der Netzwerke bestimmt bzw. erlernt werden können.

3.1 Mustererkennung

Netzwerke, die Mustererkennungsaufgaben lösen, werden assoziative Netzwerke genannt. Ihre Aufgabe besteht in der Speicherung von Paaren bestimmter Ein- und Ausgabemuster/-vektoren $(\mathbf{x}_1,\mathbf{y}_1),\ldots,(\mathbf{x}_m,\mathbf{y}_m)$. Wird ein bestimmtes Eingabemuster $\mathbf{x}_i \in \Re^n$, $i \in \{1,\ldots,m\}$ an die n Eingabeeinheiten von F_X angelegt, soll das zugeordnete Muster $f(\mathbf{x}_i)=\mathbf{y}_i \in \Re^p$ an den p Ausgabeeinheiten von F_Y erzeugt werden, d.h. gemäß der Funktion $f:\Re^n \rightarrow \Re^p$ bildet f den Eingabevektor $\mathbf{x}_i$ auf den zugehörigen Ausgabevektor $f(\mathbf{x}_i)=\mathbf{y}_i \in \Re^p$ ab. Neuronale Netzwerke realisieren hier **assoziative Speicher mit inhaltsorientiertem Zugriff**.

Wir betrachten diskrete Netzwerke, in denen Informationen durch 0/1-Vektoren (binäre Darstellung) bzw. durch -1/1-Vektoren (bipolare Darstellung) dargestellt werden. Diese Vektoren können als Merkmalsvektoren aufgefaßt werden, deren Komponenten den Wert 1 oder 0/-1 annehmen, je nachdem, ob ein bestimmtes Merkmal vorhanden ist oder nicht.
(Bipolare Darstellung (x_i,y_i): $x_i \in \{-1,1\}^n$, $y_i \in \{-1,1\}^p$. Binäre Darstellung (a_i,b_i): $a_i \in \{0,1\}^n$, $b_i \in \{0,1\}^p$)

Beispiel 1: Autoassoziative Speicherung von 6 Worten der englischen Sprache {*able*,*trip*,*trap*,*take*,*time*,*cart*}, die aus vier Buchstaben der Menge {*a*,*b*,*c*,*e*,*i*,,*k*,*l*,*m*,*p*,*r*,*t*} bestehen. Wir müssen unterscheiden, an welcher Stelle die einzelnen Buchstaben erscheinen. Daraus ergibt sich folgender allgemeiner Merkmalsvektor mit 44 Komponenten der Art *Buchstabe/Stelle*, mit dessen Hilfe sich die Worte darstellen lassen:

$$\mathbf{x}_i=(x_{a/1} \ldots x_{t/1}\, x_{a/2} \ldots x_{t/2}\, x_{a/3} \ldots x_{t/3}\, x_{a/4} \ldots x_{t/4}).$$

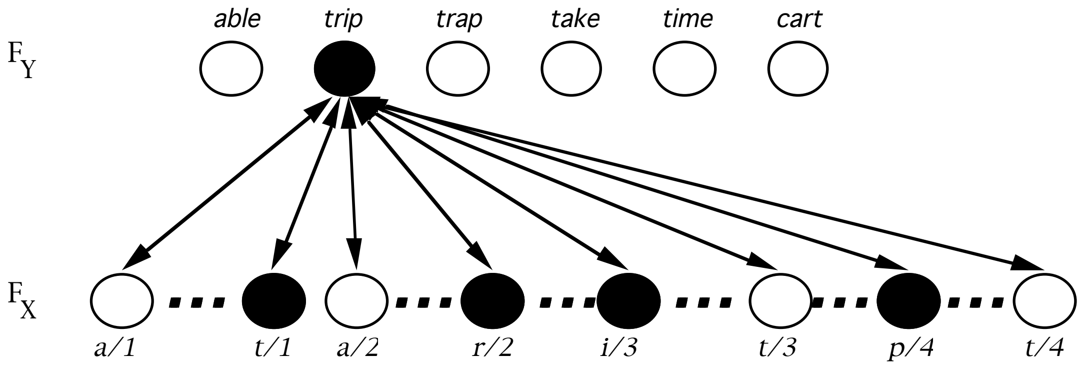
Die erste Komponente nimmt den Wert 1 an, wenn der Buchstabe *a* an erster Stelle erscheint, etc. Das Wort *able* wird dann also durch den Vektor $\mathbf{x}_{able}$

dargestellt, dessen Komponenten $x_{a/1}$, $x_{b/2}$, $x_{l/3}$ und $x_{e/4}$ den Wert 1 haben; alle anderen haben den Wert -1.

Beispiel 2: Heteroassoziative Speicherung der oben genannten Worte. In diesem Fall werden Paare von Ein- und Ausgabevektoren gespeichert. Die Eingabevektoren werden gewählt wie in Beispiel 1, die Ausgabevektoren haben 6 Komponenten, eine für jedes Wort $\mathbf{y}_i=(x_{able}\ x_{trip}\ x_{trap}\ x_{take}\ x_{time}\ x_{card})$:

$$\mathbf{x}_{able}=(x_{a/1}=1 \dots x_{b/2}=1 \dots x_{l/3}=1 \dots x_{e/4}=1) \quad \mathbf{y}_{able}=(1\ 0\ 0\ 0\ 0\ 0)$$
$$\mathbf{x}_{trip}=(x_{t/1}=1 \dots x_{r/2}=1 \dots x_{i/3}=1 \dots x_{p/4}=1) \quad \mathbf{y}_{trip}=(0\ 1\ 0\ 0\ 0\ 0)$$
$$\mathbf{x}_{trap}=(x_{t/1}=1 \dots x_{r/2}=1 \dots x_{a/3}=1 \dots x_{p/4}=1) \quad \mathbf{y}_{trap}=(0\ 0\ 1\ 0\ 0\ 0)$$
$$\mathbf{x}_{take}=(x_{t/1}=1 \dots x_{a/2}=1 \dots x_{a/3}=1 \dots x_{e/4}=1) \quad \mathbf{y}_{take}=(0\ 0\ 0\ 1\ 0\ 0)$$
$$\mathbf{x}_{time}=(x_{t/1}=1 \dots x_{i/2}=1 \dots x_{m/3}=1 \dots x_{e/4}=1) \quad \mathbf{y}_{time}=(0\ 0\ 0\ 0\ 1\ 0)$$
$$\mathbf{x}_{cart}=(x_{c/1}=1 \dots x_{a/2}=1 \dots x_{r/3}=1 \dots x_{t/4}=1) \quad \mathbf{y}_{cart}=(0\ 0\ 0\ 0\ 0\ 1).$$

Die folgende Abbildung zeigt das heteroassoziative Netzwerk für Beispiel 2. Aktive Einheiten sind schwarz gezeichnet. Es ist das Ein-/Ausgabepaar für das Wort *trip* dargestellt. Um übersichtlich zu bleiben, sind nur die Verbindungen zwischen den Eingabeeinheiten und der Ausgabeeinheit y_{trip} dargestellt.



Das heteroassoziative Netz überführt die **verteilte** Repräsentation der Konzepte in eine **lokale** Repräsentation. Im ersten Fall wird ein Konzept durch ein Aktivitätsmuster über mehreren Neuronen dargestellt. Das Konzept, hier ein Wort, liegt genau dann vor, wenn die Aktivitäten der Eingabeeinheiten eine bestimmte AN/AUS-Konstellation annehmen. Weitere Konzepte/Worte werden als alternative Aktivitätsmuster über denselben Neuronen repräsentiert. Überlappen sich Aktivitätsmuster unterschiedlicher Informationen, sodaß man nicht mehr unterscheiden kann, welche Information dargestellt wird, spricht man von ***Crosstalk***. Im Falle der lokalen Repräsentation (*one-unit-one-concept*) wird ein Wort durch eine einzige Einheit dargestellt, d.h. das Konzept liegt genau dann vor, wenn die Einheit aktiv ist. Ein alternatives Konzept wird durch die Aktivität einer anderen Einheit repräsentiert.

Die Fragen, die sich nun natürlich stellen sind: Wie können die Gewichte des assoziativen Netzwerks bestimmt werden, damit beim Anlegen eines der gespeicherten Muster das entsprechende Ausgabemuster produziert wird? Eine Eigenschaft dieser assoziativen Netze ist es, daß sie aus unvollständigen Eingabevektoren, die zum Teil mit den gelernten übereinstimmen bzw. in gewissem Grade von ihnen abweichen, die vollständigen, korrekten Ausgabemuster rekonstruieren und vervollständigen. Wie kann aber garantiert werden, daß ein *feedback*-Netzwerk beim Anlegen und Propagieren eines beliebigen Musters konvergiert, d.h. ein globales

Gleichgewicht erreicht. Darüber hinaus soll dieser Gleichgewichtszustand einem der gespeicherten Muster entsprechen, der dem Eingabemuster am meisten ähnelt.

Das Problem kann formal so beschrieben werden:

Gegeben:

m Vektorpaare $(\mathbf{x}_1,\mathbf{y}_1),\ldots,(\mathbf{x}_m,\mathbf{y}_m)$ (heteroassoziativer Speicher) der m Vektoren $\mathbf{x}_1,\ldots,\mathbf{x}_m$ (autoassoziativer Speicher)

Gesucht:

M_{nxp}, d.h. die Gewichte des Netzwerks bzw. die Parameter der Funktion

$$f:\ \Re^n \rightarrow \Re^p$$
$$\mathbf{x}_{1xn} \rightarrow S(X_{1xn})M_{nxp},$$

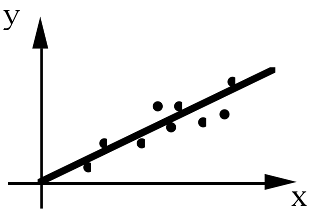
so daß $S(\mathbf{y})=\mathbf{y}_i$ mit $\mathbf{y}=f(\mathbf{x})$ bzw. $S(\mathbf{x}_{new})=\mathbf{x}_i$ mit $f(\mathbf{x})=\mathbf{x}_{new}$, $i\in\{1,\ldots,m\}$.

Die m Vektorpaare bzw. Vektoren können als "Stützpunkte" der Funktion f, die approximiert werden soll, aufgefaßt werden. Verdeutlichen wir dies anhand des eindimensionalen Falls, d.h. n=p=1:

$$f:\ \Re \rightarrow \Re$$
$$x \rightarrow mx+b.$$

Gegeben:

Punktepaare $(x_1,y_1),\ldots,(x_m,y_m)$



Gesucht:

Gerade, die durch die Punkte verläuft bzw. Parameter *m* und *b* der Geradengleichung $f(x)=mx+b$, sodaß $f(x_i)=y_i$.

Im folgenden werden drei Regeln für verschiedene Netzwerkmodelle zur Bestimmung der Gewichte vorgestellt: Die Hebb'sche Regel für diskrete, bidirektionale Netzwerke, die Regel für Kohonen-Netze und die Regel für Hopfield-Netze.

Hebb'sche Regel:

Bei gegebenen m Vektorpaaren $(\mathbf{x}_i,\mathbf{y}_i)$ können mit Hilfe der Hebb'schen Lernregel [Hebb, 1949] die Gewichte eines diskreten, bidirektionalen, synchronen Netzwerks in einem Schritt bestimmt werden. Wir nehmen an, daß die m Vektorassoziationen $(\mathbf{x}_i,\mathbf{y}_i)$, i=1,...,m in bipolarer Darstellung gegeben sind, d.h. $\mathbf{x}_i\in\{-1,1\}^n$, $\mathbf{y}_i\in\{-1,1\}^p$. Liegen die Vektoren in binärer Darstellung - $(\mathbf{a}_i,\mathbf{b}_i)$, $\mathbf{a}_i\in\{0,1\}^n$, $\mathbf{b}_i\in\{0,1\}^p$ - vor, können sie nach folgender Regel konvertiert werden: $\mathbf{x}_i=2\mathbf{a}_i-\mathbf{1}$ und $\mathbf{y}_i=2\mathbf{b}_i-\mathbf{1}$, wobei $\mathbf{1}$ der Vektor bestehend aus n bzw. p Einsen ist. Die Gewichte der Matrix werden dann nach der Regel

$$M=\sum_{k=1}^{m} w_k\ \mathbf{x}_k^T\ \mathbf{y}_k$$

bestimmt, wobei $\mathbf{x}_k^T$ der transponierte Vektor des Zeilenvektors $\mathbf{x}_k$ ist. Die Faktoren w_k ermöglichen es, bestimmten Paaren mehr Einfluß als anderen Assoziationen zuzuordnen. Durch die Hebb'sche Regel werden die gewichteten Korrelationsmatrizen der Vektorpaare summiert. Der Zeilenvektor $\mathbf{x}_k$ kann als k-te Reihe der mxn-Matrix $\mathbf{X}$, $\mathbf{y}_k$ als k-te Reihe der mxp-Matrix $\mathbf{Y}$ aufgefaßt werden. Somit können wir kürzer schreiben

$$M = \mathbf{X}^T \mathbf{W} \mathbf{Y},$$

wobei $\mathbf{W}$ die Diagonalmatrix mit den Gewichten $w_1,\ldots,w_m$ ist. Im autoassoziativen Fall haben wir

$$M = \mathbf{X}^T \mathbf{W} \mathbf{X}.$$

Kohonen-Netze:

Kohonen entwickelte eine alternative Regel für synchrone, *feedforward*-Netzwerke, bestehend aus zwei Ebenen, mit linearen Ausgabefunktionen $S_i(x_i)=x_i$ und $S_j(y_j)=y_j$ [Kohonen, 1984]. Diese Netzwerke propagieren in einem Schritt die Eingabemuster zu den Ausgabeeinheiten, d.h. hier stellt sich im Gegensatz zu den bidirektionalen Netzwerken die Frage nach der Konvergenz nicht. Weiterhin können die Assoziationen $(\mathbf{x}_k,\mathbf{y}_k)$ beliebige reellwertige Vektoren der Dimension n bzw. p sein. Die Regel für die Bestimmung der Verbindungsmatrix lautet

$$M = \mathbf{X}^{-1} \mathbf{Y}$$

falls die Inverse X^{-1} der Matrix X existiert, ansonsten

$$M = \mathbf{X}^* \mathbf{Y}$$

wobei die nxm-Matrix X^* die Pseudo-Inverse von X ist, für die gilt:

$$\mathbf{X}\,\mathbf{X}^*\,\mathbf{X} = \mathbf{X}$$
$$\mathbf{X}^*\,\mathbf{X}\,\mathbf{X}^* = \mathbf{X}^*$$
$$\mathbf{X}^*\,\mathbf{X} = (\mathbf{X}^*\mathbf{X})^T,\ \mathbf{X}\,\mathbf{X}^* = (\mathbf{X}\,\mathbf{X}^*)^T.$$

Im autoassoziativen Fall haben wir $M = \mathbf{X}^*\,\mathbf{X}$.

Wenn die inverse Matrix $(\mathbf{X}\,\mathbf{X}^T)^{-1}$ existiert gilt: $\mathbf{X}^*=\mathbf{X}^T(\mathbf{X}\,\mathbf{X}^T)^{-1}$. Die Matrix $M = \mathbf{X}^*\,\mathbf{Y}$ minimiert den mittleren quadratischen Fehler für die Ausgabe, d.h. der Erwartungswert $E(\mathbf{y}_k-\hat{\mathbf{y}}_k)^2$ der Differenz zwischen erwünschter Ausgabe $\mathbf{y}_k$ und tatsächlicher Ausgabe $\hat{\mathbf{y}}_k$ zum Quadrat wird minimal.

Wenn die Eingabevektoren $\mathbf{x}_1,\ldots,\mathbf{x}_m$ orthonormal (und damit auch linear unabhängig) sind, d.h.

$$\mathbf{x}_i\,\mathbf{x}_j^T = \begin{cases} 1, & \text{wenn } i=j \\ 0, & \text{wenn } i\neq j \end{cases}$$

gilt

$$M = \mathbf{X}^T \mathbf{Y}.$$

Aus der Orthonormalität der Vektoren folgt, daß $(\mathbf{X}\,\mathbf{X}^T)^{-1}$ existiert und die Identitätsmatrix I ist, d.h $M = \mathbf{X}^*\mathbf{Y} = \mathbf{X}^T(\mathbf{X}\,\mathbf{X}^T)^{-1}\mathbf{Y} = \mathbf{X}^T\mathbf{Y}$. Diese Gleichung gleicht der Hebb'schen Regel, wobei jedoch hier die Vektoren reellwertig sein können. In der Praxis ist die Bedingung der Orthonormalität bzw. der linearen Unabhängigkeit der Eingabevektoren selten erfüllt. Dies hat zur Konsequenz, daß das Netzwerk empfindlich gegenüber Störungen ist. In diesem Falle erkennt es Muster, die nur in geringem Maße von den Eingabemustern abweichen, nicht mehr.

Hopfield-Netze:

Die nach Hopfield [Hopfield, 1982,1984] benannten Netze sind autoassoziativ, bidirektional, diskret und asynchron. Mit ihnen könnte also auch Beispiel 1 gelöst werden. Die Gewichte werden nach folgender Regel bestimmt:

$$M = \sum_{k=1}^{m} \mathbf{x}_k{}^T \mathbf{x}_k - mI$$

mit der nxn Identitätsmatrix I.

Konvergenz von *feedback*-Netzwerken:
Die drei vorgestellten Regeln wurden für verschiedene Netzwerktypen definiert, die sich unter anderem durch ihre Verbindungsstruktur (*feedforward* oder *feedback*) unterscheiden. Hier soll noch einmal auf das oben angesprochene Konvergenzproblem eingegangen werden. Was bedeutet **Konvergenz** bei *feedback*-Netzen? (Bei *feedforward*-Netzen stellt sich die Frage der Konvergenz natürlich nicht, da hier ein Eingabemuster nur in eine Richtung propagiert werden und damit der Propagierungsprozeß bei den Ausgabeeinheiten automatisch endet.) Nehmen wir den einfachsten Fall des bidirektionalen Netzwerks, d.h. wir haben zwei Ebenen, F_X und F_Y, und für die Verbindungsstruktur gilt: $\mathbf{M}=\mathbf{N}^T$, $\mathbf{N}=\mathbf{M}^T$ und $\mathbf{P}=\mathbf{Q}=0$. Es werden abwechselnd Muster von den Ein- zu den Ausgabeeinheiten und dann von Einheiten aus F_Y zu Einheiten aus F_X propagiert:

t=0:	S(X)	$\rightarrow$	M	$\rightarrow$	S(Y)
t=1:	S(X')	$\leftarrow$	M^T	$\leftarrow$	S(Y)
t=2:	S(X')	$\rightarrow$	M	$\rightarrow$	S(Y')
t=3:	S(X'')	$\leftarrow$	M^T	$\leftarrow$	S(Y')
t=4:	S(X'')	$\rightarrow$	M	$\rightarrow$	S(Y'')
			...		

Das Netzwerk befindet sich im **Gleichgewicht**, wenn gilt:

t:	$S(X_f)$	$\rightarrow$	M	$\rightarrow$	$S(Y_f)$
t+1:	$S(X_f)$	$\leftarrow$	M^T	$\leftarrow$	$S(Y_f)$,

d.h. es wird an den Ein- und Ausgabeeinheiten immer wieder dasselbe Ausgabemuster erzeugt. Das Paar $(S(X_f),S(Y_f))$ wird als ein **Fixpunkt** des Netzwerks bezeichnet.
Ein bidirektionales Netzwerk wird **bidirektional stabil** genannt, wenn das Netzwerk beim Anlegen beliebiger Eingabemuster einen Fixpunkt erreicht.

Wie kann nun gezeigt werden, daß ein diskretes, bidirektionales Netzwerk konvergiert bzw. stabil ist?

In [Kosko, 1992] wird erläutert, daß der **globale Zustand des Netzwerks** durch eine Funktion charakterisiert werden kann, die im Zusammenhang mit Hopfield-Netzen und Boltzmann-Maschinen **Energiefunktion** genannt wird. Diese Funktion ist so definiert:

$$E = -S(X)MS(Y)^T - S(X)[I-U]^T - S(Y)[J-V]^T$$

mit den konstanten Eingabevektoren $I=[I_1,\ldots,I_n]$ und $J=[J_1,\ldots,J_p]$ und den Schwellwerten $U=[U_1,\ldots,U_n]$ und $V=[V_1,\ldots,V_p]$, I_i, J_j, U_i, $V_j \in \Re$, i=1,...,n, j=1,...,p.

Wie Kosko in [Kosko, 1992] anhand des folgenden Theorems gezeigt hat, garantiert die Existenz dieser Energiefunktion, für die gilt, daß $\dot{E} \leq 0$ und E nach unten hin beschränkt ist, daß ein diskretes, bidirektionales Netzwerk für beliebige Matrizen M stabil ist.

Theorem:
Diskrete, bidirektionale Netzwerke mit synchronem oder asynchronem Verarbeitungsmodus sind für beliebige Verbindungsmatrizen M bidirektional stabil.
Beweis:
Betrachtet wird die Änderung der Ausgabevektoren zwischen Zeitpunkt t und t+1. Die Änderung der Ausgabe zwischen zwei Zeitpunkten wird folgendermaßen definiert:

$$\Delta S(X) = S(X^{t+1}) - S(X^t) = (\ \Delta S(x_1)\ \Delta S(x_2)\ \dots\ \Delta S(x_n))$$

und

$$\Delta S(Y) = S(Y^{t+1}) - S(Y^t) = (\ \Delta S(y_1)\ \Delta S(y_2)\ \dots\ \Delta S(y_p))$$

mit

$$\Delta S(x_i) = S(x_i^{t+1}) - S(x_i^t),\ i=1,\dots,n \text{ und } \Delta S(y_j) = S(y_j^{t+1}) - S(y_j^t),\ j=1,\dots,p.$$

Wir nehmen an, daß zwischen zwei Zeitpunkten t und t+1 immer mindestens eine Einheit ihre Ausgabe ändert, aber immer nur Einheiten aus einer Gruppe (F_X oder F_Y). Auf diese Weise ist es möglich, synchrone und asynchrone Netze zu modellieren.
Für binäre Einheiten, die ihre Ausgabe nach einer Schwellwertfunktion berechnen, gilt: Wenn $S(x_i)\neq 0$ dann $\Delta S(x_i)=1-0=1$ oder $\Delta S(x_i)=0-1=-1$. Für bipolare Einheiten gilt: Wenn $S(x_i)\neq 0$ dann $\Delta S(x_i)=1--1=2$ oder $\Delta S(x_i)=-1-1=-2$. (Entsprechendes gilt für die Ausgabeeinheiten.)

Die Änderung der "Energie" wird definert als

$$\Delta E = E^{t+1} - E^t \neq 0$$

und ist ungleich Null, da immer eine Einheit entweder aus F_X oder F_Y ihre Ausgabe ändert.
Nehmen wir zuerst an, daß eine Einheit aus F_X die Änderung verursacht. Dann erhalten wir

$$\begin{aligned}
\Delta E &= E^{t+1} - E^t \\
&= -\Delta S(X)\ M\ S(Y)^T - \Delta S(X)\ [I - U]^T \\
&= -\Delta S(X)[S(Y)M^T + I - U]^T \\
&= -\sum_{i=1}^{n}\sum_{j=1}^{p} \Delta S(x_i)S(y_j^t)m_{ij} - \sum_{i=1}^{n} \Delta S(x_i)I_i - \sum_{i=1}^{n} \Delta S(x_i)U_i \\
&= -\sum_{i=1}^{n} \Delta S(x_i) \sum_{j=1}^{p} S(y_j^t)m_{ij} - \sum_{i=1}^{n} \Delta S(x_i)I_i - \sum_{i=1}^{n} \Delta S(x_i)U_i \\
&= -\sum_{i=1}^{n} \Delta S(x_i)\ [\sum_{j=1}^{p} S(y_j^t)m_{ij} + I_i - U_i] \\
&= -\sum_{i=1}^{n} \Delta S(x_i)\ [x_i^{t+1} - U_i]
\end{aligned}$$

Zu zeigen ist nun, daß der letzte Ausdruck kleiner Null ist, d.h.

$$- \sum_{i=1}^{n} \Delta S(x_i) \; [x_i^{t+1} - U_i] < 0.$$

Wenn $\Delta S(x_i) \neq 0$ für eine Einheit $i \in \{1,...,n\}$ folgt daraus, daß $x_i^{t+1} \neq U_i$. Dann müssen wir zwei Fälle betrachten:

Fall 1: $\Delta S(x_i) > 0$

Dann gilt $\Delta S(x_i) = S(x_i^{t+1}) - S(x_i^t) = 1-0$. Da die Einheit ihre Ausgabe nach der Schwellwertfunktion

$$S(x_i^{t+1}) = \begin{cases} 1, & \text{wenn } x_i^{t+1} > U_i \\ S(x_i^t), & \text{wenn } x_i^{t+1} = U_i \\ 0, & \text{wenn } x_i^{t+1} < U_i \end{cases}$$

berechnet, folgt $x_i^{t+1} > U_i$. Das heißt, daß das Produkt $\Delta S(x_i)[x_i^{t+1} - U_i]$ positiv ist und damit $\Delta E<0$.

Fall 2: $\Delta S(x_i) < 0$

Dann gilt $\Delta S(x_i) = 0-1$. Hier folgt $x_i^{t+1} < U_i$. Das Produkt $\Delta S(x_i)[x_i^{t+1} - U_i]$ wird also wieder positiv und damit auch $\Delta E<0$.

Das gleiche gilt für jede der 2^n-1 möglichen Teilmengen der Menge F_X der Eingabeeinheiten. Ebenso kann man für die Ausgabeeinheiten aus F_Y zeigen, daß jede Zustandsänderung des Netzes eine Verminderung der Energiefunktion E verursacht, d.h. zusammenfassend: $E^{t+1}-E^t<0$ für jede Zustandsänderung. Da nichts über die Gewichtsmatrix M gesagt wurde, gilt dieser Satz für beliebige Gewichte.

Der Prozeß des Einschwingens in einen Gleichgewichtszustand, der einem Zustand minimaler Energie entspricht, wird **Relaxation** genannt.

Beispiel für das Speichern von Vektorpaaren in einem diskreten, bidirektionalen, synchronen Netzwerk mit Hilfe der Hebb'schen Lernregel:

Das folgende Beispiel stammt aus [Kosko, 1992]: Gegeben seien die m=2 ungewichteten ($w_1=w_2=1$) Vektorpaare $(\mathbf{a}_1,\mathbf{b}_1)$ und $(\mathbf{a}_2,\mathbf{b}_2)$ in binärer Darstellung:

$$\mathbf{a}_1 = (1\ 0\ 1\ 0\ 1\ 0),\ \mathbf{b}_1 = (1\ 1\ 0\ 0)$$
$$\mathbf{a}_2 = (1\ 1\ 1\ 0\ 0\ 0),\ \mathbf{b}_2 = (1\ 0\ 1\ 0).$$

Die entsprechenden bipolaren Vektoren sind:

$$\mathbf{x}_1 = (1\ {-1}\ 1\ {-1}\ 1\ {-1}),\ \mathbf{y}_1 = (1\ {-1}\ {-1}\ {-1})$$
$$\mathbf{x}_2 = (1\ 1\ 1\ {-1}\ {-1}\ {-1}),\ \mathbf{y}_2 = (1\ {-1}\ 1\ {-1}).$$

Werden die Muster $\mathbf{x}_i$ an die Eingabeeinheiten angelegt, d.h. $\mathbf{x}_i$=X, soll an den Ausgabeeinheiten die Ausgabe $\mathbf{y}_i$=S(Y) ausgegeben werden. Die Gewichtsmatrix M wird dann nach der Hebb'schen Regel so bestimmt:

$$M = \mathbf{x}_1^T\mathbf{y}_1 + \mathbf{x}_2^T\mathbf{y}_2$$

$$= \begin{pmatrix} 1 \\ -1 \\ 1 \\ -1 \\ 1 \\ -1 \end{pmatrix} \begin{pmatrix} 1 & 1 & -1 & -1 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ 1 \\ -1 \\ -1 \\ -1 \end{pmatrix} \begin{pmatrix} 1 & -1 & 1 & -1 \end{pmatrix}$$

$$= \begin{pmatrix} 1 & 1 & -1 & -1 \\ -1 & -1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ -1 & -1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ -1 & -1 & 1 & 1 \end{pmatrix} + \begin{pmatrix} 1 & -1 & 1 & -1 \\ 1 & -1 & 1 & -1 \\ 1 & -1 & 1 & -1 \\ -1 & 1 & -1 & 1 \\ -1 & 1 & -1 & 1 \\ -1 & 1 & -1 & 1 \end{pmatrix}$$

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

Für bidirektionale Netzwerke gilt $N = M^T$. Betrachten wir nun die binären Vektorpaare $(\mathbf{a}_1,\mathbf{b}_1)$ und $(\mathbf{a}_2,\mathbf{b}_2)$. Alle Schwellwerte U_i,V_j und konstanten Eingaben I_i,J_i, $i=1,\ldots,n$, $j=1,\ldots,p$ werden (willkürlich) auf den Wert Null gesetzt. A_i und B_j bezeichnen die Zustandsvektoren der Ein- und Ausgabeeinheiten. Legen wir den Vektor $\mathbf{a}_1$ an die Eingabeeinheiten, so erhalten wir $A_1=\mathbf{a}_1$ und $S(A_1)=\mathbf{a}_1$. Dieses Muster $S(A_1)$ wird vorwärts propagiert:

$$S(A_1)M = (4\ 2\ -2\ -4) = B_1 \quad \rightarrow \quad (1\ 1\ 0\ 0) = S(B_1) = \mathbf{b}_1.$$

Der Pfeil $\rightarrow$ steht für die Anwendung der jeweiligen Schwellwertfunktion. Das Ausgabemuster $S(B_1)$ wird zurück zu den Eingabeeinheiten propagiert:

$$S(B_1)M^T = (2\ -2\ \ 2\ -2\ \ 2\ -2) = A_1 \rightarrow \quad (1\ 0\ 1\ 0\ 1\ 0) = S(A_1) = \mathbf{a}_1.$$

Das Paar $(\mathbf{a}_1,\mathbf{b}_1)$ ist also ein Fixpunkt des bidirektionalen, assoziativen Speichers. Die Energie des Zustands ist $E(S(A_1),S(B_1)) = -S(A_1)\ M\ S(B_1)^T = -6$.
Propagieren wir entsprechend $\mathbf{a}_2$ durch das Netzwerk erhalten wir mit $S(A_2)=\mathbf{a}_2$:

$$S(A_2)M = (4\ -2\ 2\ -4) = B_1 \quad \rightarrow \quad (1\ 0\ 1\ 0) = S(B_2) = \mathbf{b}_2$$

und

$$S(B_2)M^T = (2\ 2\ 2\ -2\ \ -2\ -2) = A_2 \rightarrow \quad (1\ 1\ 1\ 0\ 0\ 0) = S(A_2) = \mathbf{a}_2.$$

Das Paar $(\mathbf{a}_2,\mathbf{b}_2)$ ist also ebenfalls ein Fixpunkt des Netzwerks mit der Energie $E(S(A_2),S(B_2)) = -S(A_2)\ M\ S(B_2)^T = -6$. Die beiden bewußt codierten Fixpunkte haben also die gleiche Energie.
Nehmen wir nun den Eingabevektor $\mathbf{a} = (0\ 1\ 1\ 0\ 0\ 0)$. Die Hamming-Distanzen zu den Vektoren $\mathbf{a}_1$ und $\mathbf{a}_2$ sind $H(\mathbf{a},\mathbf{a}_1)=3$ und $H(\mathbf{a},\mathbf{a}_2)=1$. Wird der Vektor durch das Netz propagiert, geschieht folgendes:

$$S(A) = \mathbf{a}$$

$$S(A)M = (2\ -2\ 2\ -2) = B \quad \rightarrow \quad (1\ 0\ 1\ 0) = S(B_2) = \mathbf{b}_2$$

und

$$S(B_2)M^T = (2\ 2\ 2\ -2\ \ -2\ -2) = A_2 \rightarrow \quad (1\ 1\ 1\ 0\ 0\ 0) = S(A_2) = \mathbf{a}_2.$$

Das Netz rekonstruiert aus dem Eingabevektor $\mathbf{a}$ das Fixpunktpaar, dessen Eingabemuster ihm am meisten ähnelt.

Betrachten wir den Vektor $\mathbf{a}$ = (0 0 1 1 0 0). Dieser Vektor ähnelt $\mathbf{a}_1$ mehr als $\mathbf{a}_2$: $H(\mathbf{a},\mathbf{a}_1)=3<5=H(\mathbf{a},\mathbf{a}_2)$. Es ist also wahrscheinlicher, daß das Netz den Gleichgewichtszustand $(\mathbf{a}_1,\mathbf{b}_1)$: Wir sehen aber, daß

$$S(A)M = (-2\ 2\ -2\ 2) = B \quad \rightarrow \quad (1\ 0\ 1\ 0) = S(B) = \mathbf{b}_2^c$$

mit dem Vektor $\mathbf{b}_2^c$, der das Komplement von $\mathbf{b}_2$ ist. Wird dieser Vektor zurückpropagiert,

$$S(B_2^c)M^T = (-2\ -2\ -2\ 2\ \ 2\ 2) = A_2^c \quad \rightarrow \quad (0\ 0\ 0\ 1\ 1\ 1) = S(A_2^c) = \mathbf{a}_2^c$$

sehen wir, daß das Netzwerk in einen anderen Fixpunkt $(\mathbf{a}_2^c,\mathbf{b}_2^c)$ konvergiert. Diese unbeabsichtigten, "unechten" Fixpunkte entstehen, je mehr Neuronen das Netz hat, d.h. je größer die Dimension des Netzwerks ist.

Der Vollständigkeit wegen, gehen wir das Beispiel für die bipolaren Vektorpaare $(\mathbf{x}_1,\mathbf{y}_1)$ und $(\mathbf{x}_2,\mathbf{y}_2)$ durch:
Da

$$S(X_1)=\mathbf{x}_1,$$
$$S(X_1)M = (8\ 4\ -4\ -8) = Y_1 \quad \rightarrow \quad (1\ 1\ -1\ -1) = S(Y_1) = \mathbf{y}_1$$
$$S(Y_1)M^T = (4\ -4\ 4\ -4\ \ 4\ -4) = X_1 \quad \rightarrow \quad (1\ -1\ 1\ -1\ 1\ -1) = S(X_1) = \mathbf{x}_1$$

und

$$S(X_2)=\mathbf{x}_2,$$
$$S(X_2)M = (8\ -4\ 4\ -8) = Y_2 \quad \rightarrow \quad (1\ -1\ 1\ -1) = S(Y_2) = \mathbf{y}_2$$
$$S(Y_2)M^T = (4\ 4\ 4\ -4\ \ -4\ -4) = X_2 \quad \rightarrow \quad (1\ 1\ 1\ -1\ -1\ -1) = S(X_2) = \mathbf{x}_2$$

sind $(\mathbf{x}_1,\mathbf{y}_1)$ und $(\mathbf{x}_2,\mathbf{y}_2)$ Fixpunkte des Netzwerks mit der Energie E=-24.
Der "verrauschte" Eingabevektor $\mathbf{x}$=(-1 1 1 -1 -1 -1) produziert den Fixpunkt $(\mathbf{x}_2,\mathbf{y}_2)$, da $S(X) = \mathbf{x}$ und

$$S(X)M = (4\ -4\ -4\ -4) = Y \quad \rightarrow \quad (1\ -1\ 1\ -1) = S(Y) = \mathbf{y}_2.$$

Der Eingabevektor $\mathbf{x}$=(-1 -1 -1 1 1 -1), produziert den komplementären Fixpunkt $(\mathbf{x}_2^c,\mathbf{y}_2^c)$, da $S(X) = \mathbf{x}$ und

$$S(X)M = (-4\ 4\ -4\ 4) = Y \quad \rightarrow \quad (-1\ 1\ -1\ 11) = S(Y^c{}_2) = -\mathbf{y}_2 = \mathbf{y}_2^c$$
$$S(Y_2^c)M^T = (-4\ -4\ -4\ 4\ \ 4\ 4) = X_2^c \quad \rightarrow \quad (1\ 1\ 1\ -1\ -1\ -1) = S(X_2^c) = \mathbf{x}_2^c.$$

3.2 Kombinatorische Optimierungsprobleme

Kombinatorische Optimierungsprobleme bestehen darin, **Minima bzw. Maxima einer Kostenfunktion E** zu finden, die von sehr vielen Variablen abhängt. Die allgemeine Problemstellung kann folgendermaßen definiert werden:

Gegeben:

n Variablen $X_1,\ldots,X_n$,

Wertebereiche $W_1,\ldots,W_n$ für jede der Variablen $X_1,\ldots,X_n$,

Randbedingungen (lokal und global), die für eine Zuordnung erfüllt sein müssen,

Kostenfunktion E, die eine Zuordnung bewertet.

Gesucht:

Zuordnung $z: X_i \rightarrow w_i \in W_i$, i=1,…,n, für die die Kostenfunktion minimal wird.

Ein typische Optimierungsaufgabe ist das Problem des Handlungsreisenden: Gegeben sind n Städte $S_1,\ldots,S_n$ und die Entfernungen zwischen Paaren von

Städten: distanz(S_1,S_2), distanz(S_1,S_3), ..., distanz(S_{n-1},S_n). Gesucht ist eine Strecke, auf der jede Stadt einmal besucht wird und deren Länge minimal ist.

Gegeben:

n Städte $S_1,\ldots,S_n$,

n(n-1) Distanzen zwischen Paaren von Städten: distanz(S_1,S_2), distanz(S_1,S_3), …, distanz(S_{n-1},S_n).

Gesucht:

Strecke, auf der jede Stadt einmal besucht wird und deren Länge minimal ist.

Randbedingungen:

1.) Jede Stadt darf nur einmal besucht werden.
2.) Während eines Stops kann sich der Handlungsreisende immer nur in einer Stadt aufhalten.
3.) Die Länge der Strecke soll minimal werden.

Nehmen wir nun eine nxn-Matrix **X** mit $x_{ij} \in \{0,1\}$, deren Zeilenindizes die Städte sind und deren Spaltenindizes die Stationen der Reise sind. Wenn $x_{ij}=1$, ist die Stadt S_i die j-te Station der Reise.

	1	2	3	...	n
1	x_{11}	x_{12}	x_{13}	...	x_{1n}
2	x_{21}	x_{22}	x_{23}	...	x_{2n}
3	x_{31}	x_{32}	x_{33}	...	x_{3n}
:	...	...	...	...	...
:	...	...	...	...	...
n	x_{n1}	...	...	...	x_{nn}

mit

$$x_{ij} = \begin{cases} 1, & \text{wenn die Stadt i an j-ter Stelle besucht wird} \\ 0, & \text{sonst} \end{cases} .$$

Hätten wir zum Beispiel 4 Städte S_1, S_2, S_3, S_4 stellt die folgende Matrix eine Lösung des Problems dar, in der zuerst S_2, dann S_3 und S_4 und zuletzt S_1 besucht wird:

	1	2	3	4
S1	0	0	0	1
S2	1	0	0	0
S3	0	1	0	0
S4	0	0	1	0

Bezogen auf die Matrixdarstellung der Lösung, erfüllt eine Lösung die erste Randbedingung, wenn, wie von Hopfield und Tank in [Hopfield und Tank, 1985] gezeigt , jede Zeile genau eine Eins enthält:

$$\sum_{j=1}^{n} x_{ij}=1 \qquad i=1,\ldots,n.$$

Die zweite Randbedingung ist erfüllt, wenn jede Spalte genau eine Eins enthält:

$$\sum_{i=1}^{n} x_{ij}=1 \qquad j=1,\ldots,n.$$

Sind die ersten beiden Randbedingungen erfüllt, haben wir eine konsistente Lösung, die dann optimal ist, wenn die Länge der Strecke minimal ist, d.h. wenn der Ausdruck

$$\sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{n} \text{distanz}(S_i,S_j) \; * \; x_{ik} \; * \; (x_{j,k+1} \; + \; x_{j,k-1})$$

minimal ist. Dabei sind die Indizes modulo n definiert, i und j beziehen sich auf Städte/Zeilen der Matrix M und k bezieht sich auf Stationen/Spalten der Matrix M. Daraus ergibt sich folgende Kostenfunktion E, deren Minimum gesucht wird:

$$\begin{aligned} E \quad = \quad & \frac{A}{2} \mid (\sum_{i=1}^{n} \; \sum_{j=1}^{n} x_{ij}) - n \mid \\ + \quad & \frac{B}{2} \mid (\sum_{j=1}^{n} \; \sum_{1=1}^{n} x_{ij}) - n \mid \\ + \quad & \frac{C}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{n} \text{distanz}(S_i,S_j) \; * \; x_{ik} \; * \; (x_{j,k+1} \; + \; x_{j,k-1}) \end{aligned}$$

mit

A, B, $C \in \Re$: Konstanten, mit denen die Randbedingungen gewichtet werden können.

Die ersten beiden Summanden nehmen den Wert Null an, wenn die Lösung konsistent ist. Die Betragszeichen verhindern, daß eine inkonsistente Lösung, bei der z.B. keine Stadt besucht wird, besser bewertet wird, als eine konsistente Lösung. Die Kostenfunktion ist nach unten hin beschränkt durch die Länge der Strecke, die eine optimale Lösung darstellt.

Basierend auf der Darstellung einer Lösung durch die Matrix X und der Kostenfunktion E kann nun mit Hilfe des **Gradientenverfahrens** eine lokal optimale Lösung gefunden werden. Der Algorithmus für dieses Iterationsverfahren lautet:

1. Initialisiere die Elemente der Matrix X zufällig mit Werten $\in \{0,1\}$; lege eine maximale Iterationszahl k_{max} fest; $\Delta E=\infty$; k=0.
2. Wiederhole solange bis $\Delta E=0$ oder $k=k_{max}$:

   2a. Wähle zufällig ein Element x^k_{ij} der Matrix X^k und ändere seinen Wert

   $$x^{neu}_{ij} = \begin{cases} 1, \text{ wenn } x^k_{ij}=0 \\ 0, \text{ wenn } x^k_{ij}=1 \end{cases}$$

   $$\Rightarrow \qquad X^{new}$$

   2b. Berechne die Änderung der Kostenfunktion $\Delta E = E(X^{new}) - E(X^k)$:
   wenn $\Delta E<0$, dann akzeptiere die Lösung: $X^k=X^{new}$;
   k=k+1;

Das Gradientenverfahren generiert iterativ eine Folge von Lösungen, wobei in Schritt 2b nur solche akzeptiert werden, die besser als die vorherige Lösung sind. Das numerische Gradientenverfahren ist ein Suchverfahren (Suche nach dem lokalen Minimum), das nach dem gleichen Prinzip funktioniert, wie das *hill-climbing* Verfahren. Auch hier haben wir das Problem, daß das gefundene Minimum ein lokales ist, aber kein globales Minimum der Kostenfunktion sein muß. In der Regel wird das Verfahren mehrmals, ausgehend von verschiedenen initialen Lösungen, gestartet, um dann unter den gelieferten lokalen Minima das kleinste auszusuchen.

Alternativ dazu kann mit einer anderen Methode, genannt '***Simulated Annealing***', nach dem global optimalen Minimum gesucht werden [Kirkpatrick et al., 1983]. *Annealing* - "Abkühlung" - ist eine Methode, die in der Physik benutzt wird und mit deren Hilfe Eigenschaften von Substanzen im Aggregat-Zustand bestimmt und analysiert werden. Man möchte z.B. sehen, ob eine Substanz flüssig bleibt oder ein Kristall oder Glas bildet. Aggregat-Zustände entsprechen Zuständen minimaler Energie. Um diese zu erreichen werden die Substanzen erst einmal bei extrem hohen Temperaturen erhitzt ($\rightarrow$ hohe Energie). Dann wird die Temperatur schrittweise erniedrigt. Bei jeder Temperatur wird solange gewartet, bis die Substanz bzw. deren Atome/Moleküle einen Gleichgewichtszustand erreicht haben, dessen Energie minimal ist. Wenn die Substanz langsam genug abgekühlt wird, erreicht die Substanz einen Gleichgewichtszustand, der global minimal ist. Der Zustand X einer Substanz wird durch eine Wahrscheinlichkeitsverteilung beschrieben:

$$P(X) = \exp\left(- \frac{E(X)}{k_B T} \right)$$

mit

$E(X) \in \Re$:	Energie des Zustandes X der Substanz
$k_B \in \Re$:	Boltzmann-Konstante
$T \in \Re$:	Temperatur.

Während des Abkühlens wird der Metropolis-Algorithmus [Metropolis et al., 1953] benutzt, um bei einer bestimmten Temperatur T das Verhalten der Atome/Moleküle der Substanz zu simulieren. Hier wird bei jedem Schritt der Zustand eines zufällig ausgewählten Atoms/Moleküls geändert. Dann wird die Differenz ΔE zwischen der Energie der Substanz vor und nach der Änderung berechnet. Ist $\Delta E \leq 0$ wird die Änderung akzeptiert. Ist $\Delta E > 0$ wird eine Zufallszahl r aus dem Intervall [0,1] generiert und die Wahrscheinlichkeit $P(\Delta E) = \exp(-\Delta E / k_B T)$ berechnet. Wenn $P(\Delta E)$ größer ist als r, wird der neue Zustand akzeptiert, obwohl seine Energie höher ist als die des vorigen Zustandes. Der Metropolis-Algorithmus wird solange wiederholt, bis sich die Energie nicht mehr ändert. Dann wird die Temperatur nach dem Abkühlungsschema erniedrigt und der Prozeß wird wiederholt.

Betrachtet man nun die folgende Analogie, so wird verständlich, wie *Simulated Annealing* zur Lösung von Optimierungsproblemen verwendet werden kann: Der Zustand einer Substanz, beschrieben durch die Zustände der einzelnen Atome/Moleküle, wird als Lösung eines Optimierungsproblems, beschrieben durch die einzelnen Variablen aufgefaßt. Die Energie der Substanz entspricht dann der Kostenfunktion: beide sollen minimiert werden.

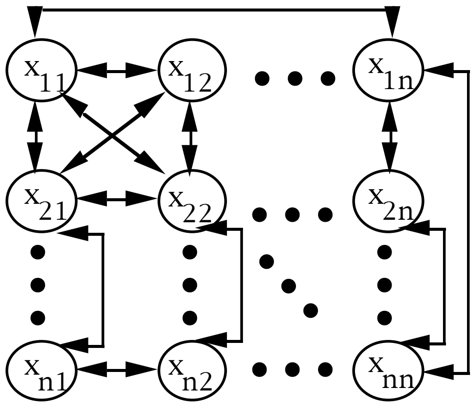
Der Algorithmus für das *Simulated Annealing* lautet:

1. Initialisierung
   1a. Bestimme ein Annealing-Schema: T_{max}, T_{min}, ΔT
   1b. Bestimme eine maximale Zahl von Wiederholungen: k_{max}
   1c. Belege die Variablen x^0_{ij} der Matrix X^0 zufällig mit Werten$\in \{0,1\}$
   1d. Zähler für Iterationen: k=0
   Temperaur: $T=T_{max}$.
2. Wiederhole solange bis $T=T_{min}$:
   2a. $T=T-\Delta T$
   2b. k=0
   2c. Wiederhole solange bis $\Delta E=0$ oder $k=k_{max}$:
      c1. Wähle zufällig ein Element x^k_{ij} der Matrix X^k und ändere seinen Wert $\Rightarrow X^{new}$
      c2. Berechne die Änderung ΔE: $\Delta E = E(X^{new}) - E(X^k)$
      c3. Metropolis-Algorithmus:
      wenn $\Delta E \le 0$: $X^k = X^{new}$
      wenn $\Delta E > 0$: $p = \exp \frac{-\Delta E}{T}$;
      generiere eine Zufallszahl $r\in [0,1]$;
      wenn $p>r$: $X^k = X^{new}$.
      c4. k=k+1

Simulated Annealing unterscheidet sich vom Gradientenverfahren durch das Abkühlungsschema und den Metropolis-Algorithmus: Es werden nicht nur Änderungen akzeptiert, die zu einer Verbesserung (Minimierung der Energie/Kostenfunktion) führen, sondern nach einem stochastischen Prinzip auch solche, die zu Verschlechterungen führen. Dies ermöglicht im Gegensatz zum Gradientenverfahren das Verlassen lokaler Minima.
Simulated Annealing ist also, wie A* [Nilsson, 80], ein Suchverfahren, daß zu optimalen Lösungen führt. Die Frage ist natürlich, ob und wann ein solches Verfahren terminiert. Aarts und Korst bewiesen in [Aarts, Korst, 1989] die asymptotische Konvergenz des Verfahrens.

Wir haben nun also zwei numerische (Such-)Verfahren kennengelernt, mit denen Optimierungsprobleme gelöst werden können. Diese beide Methoden können auch mit neuronalen Netzwerken realisiert werden: Das Gradientenverfahren mit Hopfield-Netzen, das *Simulated Annealing* mit Boltzmann-Maschinen.
Betrachten wir noch einmal die Matrixdarstellung der Lösungen für das Problem des Handlungsreisenden: Die Elemente x_{ij} der Matrix waren bei n Städten n^2 Variablen, die den Wert 0 oder 1 annahmen, je nachdem, ob die i-te Stadt an j-ter Stelle besucht wurde oder nicht. Diese Variablen können als diskrete Einheiten eines autoassoziativen Netzes dargestellt werden.

Autoassoziatives Netz mit n•n Einheiten



Matrixdarstellung einer Lösung des Problems des Handlungsreisenden

	1	2	•••	n
1	x_{11}	x_{12}	•••	x_{1n}
2	x_{21}	x_{22}	•••	x_{2n}
⋮	⋮	⋮	⋱	⋮
n	x_{n1}	x_{n2}	•••	x_{nn}

Die Ausgabe $S(x_i)$ einer Einheit i, entspricht dann dem Wert der Variablen x_i des Optimierungsproblems. Das globale Verhalten des Netzwerks wird durch die oben eingeführte Energiefunktion

$$E_{Netz} = -S(X)MS(Y)^T - S(X)[I-U]^T - S(Y)[J-V]^T$$

die sich im autoassoziativen Fall auf

$$E_{Netz} = -S(X)MS(X)^T - 2S(X)[I-U]^T$$

reduziert, beschrieben. Die Kostenfunktion für das Problem des Handlungsreisenden war

$$E_{Opt.} = \frac{A}{2} \mid (\sum_{i=1}^{n} \sum_{j=1}^{n} x_{ij}) - n \mid + \frac{B}{2} \mid (\sum_{j=1}^{n} \sum_{1=1}^{n} x_{ij}) - n \mid$$
$$+ \frac{C}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{n} \mathrm{distanz}(S_i,S_j) * x_{ik} * (x_{j,k+1} + x_{j,k-1}).$$

Durch Gleichsetzen der Energiefunktion und der Kostenfunktion erhalten wir die Gewichte des Netzwerks und können damit das Optimierungsproblem durch ein autoassoziatives, diskretes neuronales Netzwerk lösen.

Hopfield-Netze und Boltzmann-Maschinen sind autoassoziative, asynchrone, *feedback*-Netze mit symmetrischer Verbindungsmatrix M für die gilt: $m_{ij}=m_{ji}$, $i \neq j$, $i,j=1,\ldots,n$ und $m_{ii}=0$. Sie unterscheiden sich durch die Funktion zur Bestimmung der Ausgabe einer Einheit. Beim Hopfield-Netz wird eine Einheit zufällig ausgewählt, die dann ihre Eingabe berechnet, ihren Zustand entsprechend ändert und im diskreten Fall ihre Ausgabe nach einer Schwellwertfunktion berechnet. Gemäß dem oben dargestellten Theorem gilt für Hopfield-Netze, daß beliebige Zustandsänderungen die Energiefunktion E immer vermindern, d.h. es werden immer nur Zustände akzeptiert, deren Energie kleiner ist als die des vorherigen Zustands. Wird das Netz zur Lösung von Optimierungsproblemen verwendet, heißt das, daß immer nur Lösungen akzeptiert werden, die besser sind als die vorherige: Das Hopfield-Netz realisiert das Gradientenverfahren. Wir haben hier also neben der Anwendung als inhaltsorientierter Speicher eine weitere Anwendung des Hopfield-Netzes.

Boltzmann-Maschinen ([Aarts, Korst, 1989], Ackley et al., 1985] , [Hinton und Sejnowski, 1986]) ändern ihre Zustände nach der Methode des *Simulated Annealing* mit der probabilistischen Funktion:

$$p(x_i) = \frac{1}{1+\exp(-\Delta E/T)} ,$$

d.h. die Einheit i nimmt den Zustand 1 mit der Wahrscheinlichkeit $p(x_i=1)$ an. Der *Annealing*-Algorithmus für Boltzmann-Maschinen hat folgende Form:

1. Initialisierung
   - 1a. Bestimme ein Annealing-Schema: T_{max}, T_{min}, ΔT
   - 1b. Bestimme eine maximale Zahl von Wiederholungen: k_{max}
   - 1c. Bestimme die Zustände der Einheiten aus F_X zufällig: X^0
   - 1d. Zähler für Iterationen: k=0
     Temperaur: $T=T_{max}$.
2. Wiederhole solange bis $T=T_{min}$:
   - 2a. $T=T-\Delta T$
   - 2b. k=0
   - 2c. Wiederhole solange bis $\Delta E=0$ oder $k=k_{max}$:
     - c1. Wähle zufällig eine Einheit x^k_{ij} und ändere ihren Zustand $\Rightarrow X^{new}$
     - c2. Berechne die Änderung ΔE: $\Delta E = E(X^{new}) - E(X^k)$
     - c3. Metropolis-Algorithmus:
       wenn $\Delta E \leq 0$: $X^k = X^{new}$
       wenn $\Delta E > 0$: $p(x_i) = \frac{1}{1+\exp(-\Delta E/T)}$;
       generiere eine Zufallszahl $r \in [0,1]$;
       wenn p>r: $X^k = X^{new}$.
     - c4. k=k+1

4 Lernen mit neuronalen Netzwerken

Wir unterscheiden zwischen Methoden zum Lernen aus Beispielen und zum Lernen aus Beobachtungen. Im ersten Fall ist bekannt, zu welchen Klassen die Beispiele gehören, d.h. wir haben es mit einer Form von überwachtem Lernen zu tun. Im zweiten Fall sind die Kategorien (Art und Anzahl), zu denen die Beobachtungen gehören, unbekannt. Diese Art von Lernen wird auch unüberwachtes Lernen genannt.

Im Gegensatz zu logik-basierten Lernverfahren [Morik, 1993], haben wir es hier mit numerischen, statistischen Lernverfahren zu tun. Ihr Ziel ist es, eine Schätzfunktion f für die unbekannte Funktion $f^*: X \to Y$, $X \in \Re^n$, $Y \in \Re^p$ mit Hilfe von m Beispielen $(\mathbf{x}_1,\mathbf{y}_1),\ldots,(\mathbf{x}_m,\mathbf{y}_m)$ bzw. von m Beobachtungen $(\mathbf{x}_1,\bullet),\ldots,(\mathbf{x}_m,\bullet)$ zu finden. Die Parameter dieser linearen Funktion $f(X)=XM$ sind die Gewichte des Netzwerks, repräsentiert durch die Verbindungsmatrix M. Die $\mathbf{x}_i$ und $\mathbf{y}_i$ werden als Realisationen von Zufallsvariablen aufgefaßt, d.h. die gesuchte Schätzfunktion f basiert auf der unbekannten Wahrscheinlichkeitsdichte $p(\mathbf{x},\mathbf{y})$ bzw. $p(\mathbf{x})$.

4.1 Lernen aus Beispielen - Überwachtes Lernen

Das Lernproblem kann folgendermaßen formuliert werden:
Gegeben:

Beispiele $(\mathbf{x}_1,\mathbf{t}_1),\ldots,(\mathbf{x}_m,\mathbf{t}_m)$:

Die Eingabemuster $\mathbf{x}_b$, b=1,…,m repräsentieren die Beispiele, die bestimmten Kategorien, dargestellt durch die Ausgabevektoren $\mathbf{t}_b$, zugeordnet sind. Wir bezeichnen die Ausgabevektoren der Realisationen für das Training mit $\mathbf{t}$, um sie von der tatsächlichen Ausgabe $S(\mathbf{y})$ des Netzwerks zu unterscheiden.

Ziel:

Bestimmung der Schätzfunktion $f:\Re^n \to \Re^p$ bzw. die Bestimmung der Gewichte des Netzes, so daß $\mathbf{t}_b=S(\mathbf{y}_b)$ mit $\mathbf{y}_b=f(\mathbf{x}_b)=\mathbf{x}_b M$, b=1,…,m.

Während der Lernphase sollen also die Parameter der Funktion f, d.h. die Gewichte des Netzwerks so bestimmt werden, daß für alle Realisationen $(\mathbf{x}_b,\mathbf{t}_b)$, b=1,...,m die Ausgabe $S(\mathbf{y}_b)$ des Netzwerks mit der erwünschten Ausgabe $\mathbf{t}_b$ übereinstimmt. Ein weiterer erwünschter Lerneffekt ist der der Generalisierung: Vektoren $\mathbf{x}_l$, die einem $\mathbf{x}_b$, $b\in\{1,...,m\}$ ähneln, aber nicht exakt mit ihm übereinstimmen, sollen dem zugehörigen $\mathbf{t}_b$ zugeordnet werden.

Wir beschränken uns im folgenden erst einmal auf Lernverfahren für *feedforward*-Netze mit zwei Ebenen. Gegeben sei eine Trainingsmenge bestehend aus m Ein-/Ausgabepaaren $(\mathbf{x}_b,\mathbf{t}_b)$, b=1,...,m. Für eine Verbindungsmatrix M und ein Beispiel $(\mathbf{x}_b,\mathbf{t}_b)$ spiegelt der **Fehler $D_b(f)$** der Funktion f den Unterschied zwischen erwünschter Ausgabe $\mathbf{t}_b$ und tatsächlicher Ausgabe $S(\mathbf{y}_b)$ mit $\mathbf{y}_b=f(\mathbf{x}_b)=\mathbf{x}_b M$ wider. Dieser Fehler kann z.B. als Differenz der Vektoren $\mathbf{t}_b$ und $S(\mathbf{y}_b)$ definiert werden:

$$D_b(f) = \mathbf{t}_b - S(\mathbf{y}_b) \quad \text{mit} \quad D_b(f)\in\Re^p.$$

Der **Gesamtfehler $D(f)$** der Funktion f kann dann als Summe der Fehler für die einzelnen Beispiele definiert werden:

$$D(f) = \sum_{b=1}^{m} D_b(f) = \sum_{b=1}^{m} \mathbf{t}_b - S(\mathbf{y}_b) \quad \text{mit} \quad D(f)\in\Re^p.$$

Ziel ist es, die Parameter der Funktion f, d.h. die m_{ij}, i=1,...,n, j=1,...,p, so zu schätzen, daß der Fehler den Wert 0 annimmt, sodaß erwünschte und tatsächliche Ausgabe übereinstimmen. In obigem Fall bedeutet dies, daß D(f) zum Nullvektor wird. Dieser Fehler wird als Beurteilungskriterium für die Güte der gefundenen Schätzfunktion benutzt: Für sie sollte der Fehler minimal, möglichst gleich Null sein.

Das Lernproblem kann also als Minimierungsproblem aufgefaßt werden: Bestimme die Schätzfunktion f so, daß der Fehler D(f) minimal wird.
Eine Methode zur Minimierung von Funktionen haben wir schon kennengelernt, das Gradientenverfahren. Lernen wird also wieder als Suche aufgefaßt, nämlich als Suche nach der optimalen Gewichtsmatrix M: Ausgehend von einer Startmatrix $M^{(0)}$, wird iterativ eine Folge von Matrizen $M^{(k)}$, k>0, erzeugt, die die Matrix M^* approximieren, für die D(f) minimal wird:

$$M^{(k+1)} = M^{(k)} + c^{(k)}\, \nabla D^{(k)}$$

bzw.

$$\mathbf{m}_i^{(k+1)} = \mathbf{m}_i^{(k)} + c^{(k)}\, \nabla D_i^{(k)}, \qquad i=1,\dots,n$$

wenn die Zeilenvektoren der Matrix M betrachtet werden. Die **Lernrate** $c^{(k)}$ ist Element einer Folge von abnehmenden, positiven, reelen Zahlen. Je langsamer die $c^{(k)}$, $k=1,2,\dots,\infty$ abnehmen, desto schneller wird gelernt. Je schneller sie abnehmen, desto weniger Information wird durch die Gewichte, das Langzeitgedächtnis, vergessen. Die $c^{(k)}$, $k=1,2,\dots,\infty$ können z.B. die Elemente der Folge $c^{(k)} = 1/k$ sein. $\nabla D^{(k)}$ ist der Gradient des Gesamtfehlers bei der k-ten Iteration. Der **Gradient** $\nabla D_i^{(k)} \in \Re^p$ der k-ten Iteration ist ein Vektor, dessen Komponenten die partiellen Ableitungen des Gesamtfehlers $D^{(k)}$ nach dem jeweiligen Gewicht m_{ij} sind:

$$\nabla D_i = \left(\frac{\partial D}{\partial m_{i1}} \; \frac{\partial D}{\partial m_{i2}} \cdots \frac{\partial D}{\partial m_{ip}} \right), \qquad i=1,\dots,n.$$

Dieser Vektor zeigt in die Richtung, in die man den Gewichtsvektor $\mathbf{m}_i$ drehen muß, um den Fehler maximal zu verkleinern.

Die Regel zur Modifikation einzelner Gewichte hat dann die Form

$$m_{ij}^{(k+1)} = m_{ij}^{(k)} + \Delta m_{ij}^{(k)}$$

mit

$$\Delta m_{ij}^{(k)} = c^{(k)} * \frac{\partial D}{\partial m_{ij}}, \qquad i=1,\dots,n \text{ und } j=1,\dots,p.$$

Eine spezielle Form der Gleichung für $\Delta m_{ij}^{(k)}$ ist die sogenannte **Delta-Regel**:

$$\Delta m_{ij}^{(k)} = c^{(k)} * (t_j - S(y_j)) * S(x_i).$$

Das Gewicht m_{ij} der Verbindung von Einheit i zu Einheit j wird in Abhängigkeit von der Differenz zwischen erwünschter und tatsächlicher Ausgabe der Einheit j und in Abhängigkeit von der Ausgabe der vorgeschalteten Einheit i geändert.

In den nächsten beiden Abschnitten werden nun zwei Lernverfahren vorgestellt: Das erste für diskrete *feedforward*-Netze mit zwei Ebenen (**Perceptrons**), das zweite für kontinuierliche *feedforward*-Netze bestehend aus mehr als zwei Ebenen. Beide Verfahren sind Gradientenverfahren mit unterschiedlich definiertem Fehler.

4.1.1 Lernen mit dem Perceptron

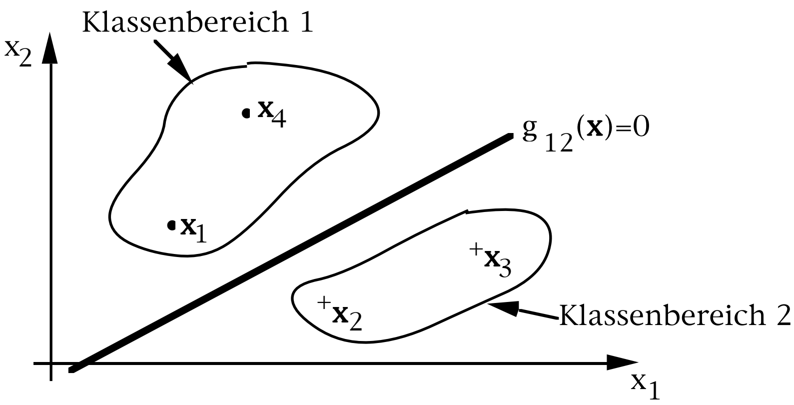
Mit Hilfe des Perceptrons (und anderer Netzwerke) können typische Mustererkennungsaufgaben mit folgender Problemstellung gelöst werden:

Gegeben:

m Beispiele $(\mathbf{x}_b, \mathbf{t}_b)$, b=1,…,m

Anzahl der Klassen: z

Nehmen wir z.B. an, daß die Beispiele durch 2 Merkmale, x_1 und x_2 beschrieben werden, d.h. $\mathbf{x}_b = (x_{b_1}\ x_{b_2})$, $b=1,\dots,m$. Gegeben seien m=4 Beispiele für z=2 Klassen. Wie in der Graphik dargestellt, gehören $\mathbf{x}_1$ und $\mathbf{x}_4$ zu Klasse 1 und $\mathbf{x}_2$ und $\mathbf{x}_3$ zu Klasse 2.



Wir haben zwar Beispiele für die Klassen, der vollständige **Klassenbereich** (*decision region*) für jede Klasse ist jedoch unbekannt.

Gesucht:

Unterscheidungsfunktionen $g_1(\mathbf{x}),\ldots,g_z(\mathbf{x})$ für jede der z Klassen

Trennflächen/-ebenen $g_{kl}(\mathbf{x})$ für je zwei Klassen k und l, $k \neq l$, $k,l=1,\ldots, z$

Bei logik-basierten Lernverfahren [Morik, 1993] haben wir für jede Klasse/Kategorie einen logischen Ausdruck, der für Beispiele der Klasse wahr und für alle Beispiel anderer Klassen falsch wird. Hier haben wir es mit numerischen **Unterscheidungsfunktionen** (*discriminant functions*) zu tun, die auf Merkmalsvektoren **x** angewendet werden, durch die Objekte beschrieben werden. Diese Funktionen dienen dazu, ein Objekt der (unbekannten) Klasse, der es angehört, zuzuordnen: Für den gegebenen Merkmalsvektor **x** des Objekts wird der Wert aller Unterscheidungsfunktionen $g_1(\mathbf{x}),\ldots,g_z(\mathbf{x})$ bestimmt und das Objekt wird der Klasse zugeordnet, deren Funktion den maximalen Wert hat. Daraus ergibt sich die Forderung an eine Unterscheidungsfunktion g_k, $k \in \{1,\ldots,z\}$: Sei **x** ein Element der Klasse k. Ermittelt man die Werte der Unterscheidungsfunktionen $g_1(\mathbf{x}),\ldots,g_z(\mathbf{x})$, so soll $g_k(\mathbf{x})$ den maximalen Wert haben, d.h.

$$\forall\ \mathbf{x} \in \text{Klasse } k: \quad g_k(\mathbf{x}) > g_l(\mathbf{x}), \qquad l \neq k,\ k,l=1,\ldots,z.$$

Trennflächen/-ebenen (*decision surfaces*) $g_{kl}(\mathbf{x})$ für je zwei Klassen k und l, $l \neq k$, $k,l=1,\ldots,z$, sind definiert als die Differenz der beiden Unterscheidungsfunktionen für die Klassen l und k:

$$g_{kl}(\mathbf{x}) = g_k(\mathbf{x}) - g_l(\mathbf{x}).$$

Sie dienen der Zuordnung eines Objekts zu einer der beiden Klassen. Wenn $g_k(\mathbf{x}) > g_l(\mathbf{x})$ ist, ist $g_{kl}(\mathbf{x})>0$ und **x** wird der Klasse k zugeordnet. Wenn $g_k(\mathbf{x}) < g_l(\mathbf{x})$ ist, ist $g_{kl}(\mathbf{x})<0$ und **x** wird der Klasse l zugeordnet. Hier werden nur die Werte von zwei und nicht, wenn $z>2$, von allen Unterscheidungsfunktionen verglichen, d.h. daß im Falle $z>2$ **x** nicht unbedingt Klasse k angehört, wenn $g_{kl}(\mathbf{x})>0$. Es bedeutet nur, daß das Objekt den Elementen der Klasse k mehr ähnelt, als denen der Klasse l.

Wir beschränken uns im folgenden auf lineare Unterscheidungsfunktionen der Form

$$g_k(\mathbf{x}) = m_1x_1 + m_2x_2 + \ldots + m_nx_n + m_{n+1}, \qquad k \in \{1,\ldots,z\}.$$

Die Trennebenen sind dann ebenfalls linear. Weiterhin betrachten wir erst einmal nur z=2 Klassen. Gesucht wird dann die Trennebene

$$
\begin{aligned}
g(\mathbf{x}) &= g_1(\mathbf{x}) - g_2(\mathbf{x}) \\
&= (m^1{}_1 - m^2{}_1)x_1 + (m^1{}_2 - m^2{}_2)x_2 + \ldots + (m^1_n - m^2{}_n)x_n + (m^1{}_{n+1} - m^2{}_{n+1}) \\
&= m_1x_1 + m_2x_2 + \ldots + m_nx_n + m_{n+1}
\end{aligned}
$$

mit

$m_i = m^1{}_i - m^2{}_i, \ i=1,\ldots,n+1$

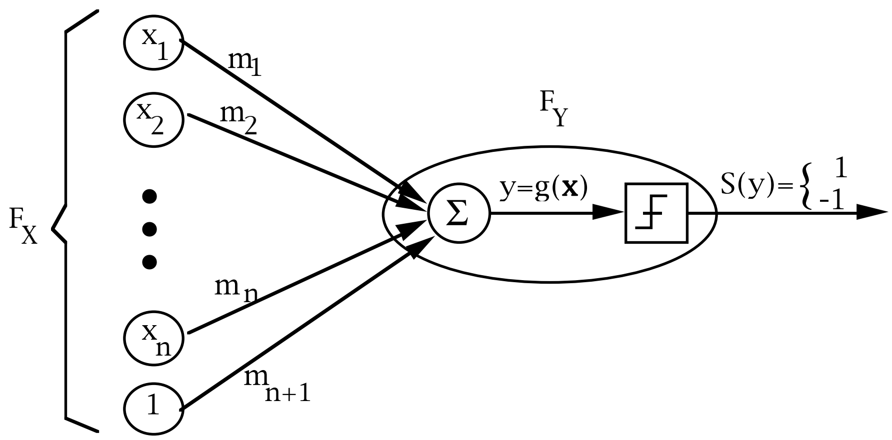
$m^j{}_i$: i-ter Koeffizient der j-ten Unterscheidungsfunktion $g_j(\mathbf{x})$, $j \in \{1,2\}$.

Für diese Trennebene gilt:

$$
\begin{aligned}
g(\mathbf{x}) > 0 &\quad \Rightarrow \quad \mathbf{x} \in \text{Klasse 1} \\
g(\mathbf{x}) < 0 &\quad \Rightarrow \quad \mathbf{x} \in \text{Klasse 2} .
\end{aligned}
$$

Wenn $g(\mathbf{x})=0$ ist, liegt $\mathbf{x}$ genau auf der Geraden und es kann nicht entschieden werden, welcher Klasse $\mathbf{x}$ zugeordnet werden soll.

Die Trennebene kann durch ein **Perceptron** implementiert werden:



Das Perceptron ist ein *feedforward*-Netz mit zwei Ebenen. Die Gruppe F_X besteht aus n+1 Einheiten mit linearer Ausgabefunktion $S(x_i)=x_i$. Die Einheit n+1 wird konstant im Zustand $x_{n+1}=1$ gehalten. Die Gruppe F_Y enthält für z=2 nur eine Einheit, deren Zustand die gewichtete Summe der Ausgaben der Eingabeeinheiten ist, $y=net=\sum_{i=1}^{n+1} S(x_i)m_i$, und die ihre Ausgabe nach der Schwellwertfunktion

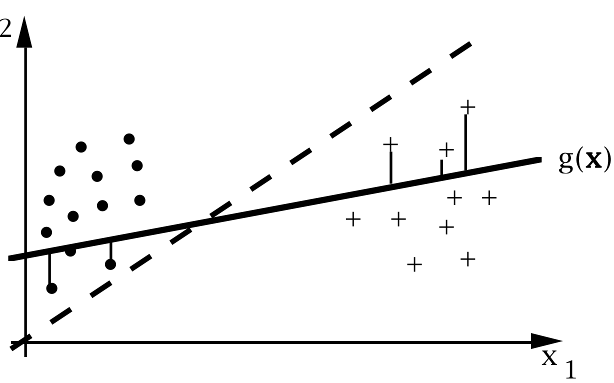
$$
S(y^t) = \begin{cases} 1, & \text{wenn } y^t > 0 \\ -1, & \text{wenn } y^t < 0 \end{cases}
$$

berechnet.

Durch das Lernverfahren sollen die Gewichte des Netzwerks, d.h. die Parameter der Trennfunktion $m_1,\ldots,m_{n+1}$ der Trennebene

$$
g(\mathbf{x}) = m_1x_1 + m_2x_2 + \ldots + m_nx_n + m_{n+1}
$$

ermittelt werden. Dies geschieht mit Hilfe des Gradientenverfahrens. Die folgende Skizze verdeutlicht, wie der Fehler, der als Beurteilungskriterium für die Güte der gefundenen Trennebene verwendet wird, definiert ist.



Dargestellt sind Beispiele für zwei Klassen: Die Beispiele für Klasse 1 sind durch Punkte, die Beispiele für Klasse 2 durch '+'-Zeichen dargestellt. Gegeben sei die Trennebene g(**x**), dargestellt durch die durchgezogene Gerade, für die gelten soll:

$$g(\mathbf{x}) > 0 \quad \Rightarrow \quad \mathbf{x} \in \text{Klasse 1}$$
$$g(\mathbf{x}) < 0 \quad \Rightarrow \quad \mathbf{x} \in \text{Klasse 2} \ .$$

Wenn $g(\mathbf{x})>0$, liegt **x** oberhalb der Geraden, wenn $g(\mathbf{x})<0$, liegt **x** unterhalb der Geraden. Mehrere Beispiele sowohl von Klasse 1 als auch von Klasse 2 werden durch g(**x**) falsch klassifiziert. Die Trennebene kann jedoch dadurch verbessert werden, daß sie gegen den Uhrzeigersinn gedreht wird, wie durch die gestrichelete Linie dargestellt. Für einen Merkmalsvektor **x** ist $|g(\mathbf{x})|$, der Betrag von g(**x**), genau der Abstand des Vektorendpunktes von der Trennebene. In obiger Skizze sind die Abstände für die falsch klassifizierten Beispiele durch senkrechte Linien zwischen Endpunkt und Geraden eingezeichnet. Mit ihnen wird der Fehler definiert, der beim Lernverfahren für das Perceptron verwendet wird:

$$D(\mathbf{m}) = \sum_{\substack{\mathbf{x} \in \text{Menge der falsch} \\ \text{klassifizierten Beispiele}}} |g(\mathbf{x})|$$

$$= \sum_{\substack{\mathbf{x} \in \text{Menge der falsch} \\ \text{klassifizierten Beispiele}}} \left| \sum_{i=1}^{n+1} m_i x_i \right|$$

$$= \sum_{\substack{\mathbf{x} \in \text{Menge der falsch} \\ \text{klassifizierten Beispiele}}} \mathbf{m} \begin{cases} -\mathbf{x}^T, & \text{wenn } \mathbf{x} \text{ falsch klassifiziert wurde und } \mathbf{x} \in \text{Klasse 1} \\ \mathbf{x}^T, & \text{wenn } \mathbf{x} \text{ falsch klassifiziert wurde und } \mathbf{x} \in \text{Klasse 2} \end{cases}$$

Der Gradient $\nabla D(\mathbf{m})$ des Fehlers ist

$$\nabla D(\mathbf{m}) = \left(\frac{\partial D}{\partial m_1} \ \frac{\partial D}{\partial m_2} \cdots \frac{\partial D}{\partial m_n} \right)$$

$$= \sum_{\substack{\mathbf{x} \in \text{Menge der falsch} \\ \text{klassifizierten Beispiele}}} \begin{cases} -\mathbf{x}^T, & \text{wenn } \mathbf{x} \text{ falsch klassifiziert wurde und } \mathbf{x} \in \text{Klasse 1} \\ \mathbf{x}^T, & \text{wenn } \mathbf{x} \text{ falsch klassifiziert wurde und } \mathbf{x} \in \text{Klasse 2} \end{cases}$$

Jetzt ist die Anwendung des Gradientenverfahrens mit folgender Regel (**fixed-increment Regel**) zur Modifikation der Gewichte möglich:

$$\mathbf{m}^{(k+1)} = \mathbf{m}^{(k)} + c^{(k)} * \nabla D(\mathbf{m})$$

$$= \begin{cases} \mathbf{m}^{(k)} - c^{(k)}\,\mathbf{x}, & \text{wenn } \mathbf{x} \text{ falsch klassifiziert wurde und } \mathbf{x} \in \text{Klasse 1} \\ \mathbf{m}^{(k)} + c^{(k)}\,\mathbf{x}, & \text{wenn } \mathbf{x} \text{ falsch klassifiziert wurde und } \mathbf{x} \in \text{Klasse 2} \end{cases}$$

Dies ist eine vereinfachte Form der oben erwähnten Delta-Regel

$$\Delta m_{ij}{}^{(k)} = c^{(k)} * (t_j - S(y_j)) * S(x_i),$$

mit $S(x_i)=x_i$ und bei der t_j-$S(y_j)$ nicht berücksichtigt bzw. $c^{(k)}*x_i$ nur berechnet wird, wenn $t_j - S(y_j) \neq 0$.

Der Lernalgorithmus für das Perceptron (***Fixed-Increment Perceptron Learning***; Rosenblatt,62) lautet dann:

1. Bilde ein Perceptron mit n+1 Eingabeeinheiten und 1 Ausgabeeinheit.
2. Initialisiere die Gewichte $m_1,\ldots,m_{n+1}$ zufällig.
3. Wiederhole solange bis alle Beispiele korrekt klassifiziert werden:
   4. Überprüfe für jedes einzelne Beispiel i=1,…,m, ob es korrekt klassifiziert wird; wenn nicht: Modifiziere die Gewichte nach der *fixed-increment* Regel.
   5. Gehe zu Schritt 3.

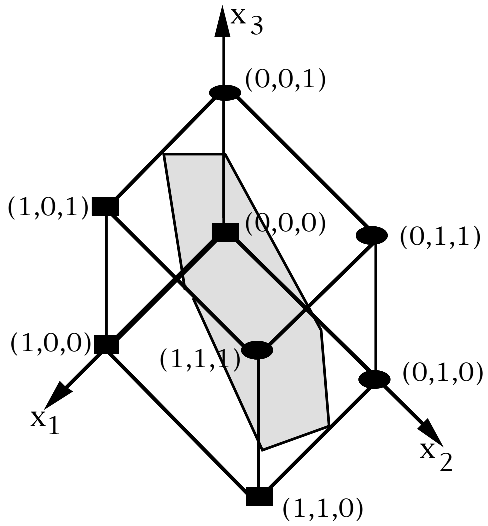
Das Perceptron kann für p>1 Ausgabeeinheiten verallgemeinert werden. Wir haben dann ein *feedforward*-Netz mit zwei Ebenen: Jede Einheit aus F_X ist mit jeder Einheit aus F_Y verbunden. Jede Ausgabeeinheit $j \in \{1,\ldots,p\}$ bildet mit den n Eingabeeinheiten ein Perceptron, daß für jedes Trainingsbeispiel $(\mathbf{x}_l,\mathbf{t}_l)$ l=1,...,m unabhängig von den p-1 anderen Perceptrons mit obigem Algorithmus trainiert werden kann.

Beispiel Perceptron [Nilsson, 1990]:

Gegeben seien m=8 Beispiele für z=2 Klassen, die durch n=4 Merkmale beschrieben werden:

	x_1	x_2	x_3	x_4	t
1	0	1	1	1	-1
2	0	0	0	1	1
3	1	0	0	1	1
4	1	0	1	1	1
5	0	0	1	1	-1
6	1	1	0	1	1
7	1	1	1	1	-1
8	0	1	0	1	-1

Das vierte Merkmal ist bei allen Beispielen gleich. Die folgende Abbildung zeigt, daß eine Ebene existiert, die die beiden Klassen voneinander trennt. Sie ist schraffiert eingezeichnet. Die Enpunkte der Merkmalsvektoren der Beispiele für Klasse 1 (t=-1) sind durch Kreise, die für Klasse 2 (t=1) durch Vierecke markiert.



$$g(\mathbf{x}) = \mathbf{m} * \mathbf{x}^T \begin{cases} > 0, & \text{wenn } \mathbf{x} \in \text{Klasse } 1 \\ = 0, & \text{wenn Klasse undefiniert } (*) \\ < 0, & \text{wenn } \mathbf{x} \in \text{Klasse } 2 \end{cases}$$

$$\mathbf{m}^{(k+1)} = \begin{cases} \mathbf{m}^{(k)} - c^{(k)}\,\mathbf{x}, & \text{wenn } \mathbf{x} \text{ falsch klassifiziert ist und } \mathbf{x} \in \text{Klasse } 1 \\ \mathbf{m}^{(k)} + c^{(k)}\,\mathbf{x}, & \text{wenn } \mathbf{x} \text{ falsch klassifiziert ist und } \mathbf{x} \in \text{Klasse } 2 \end{cases}$$

Im folgenden ist $c^{(k)} = 1$, für $k=1,\ldots,\infty$

Beispiel $\mathbf{x_i}$				Gewichtsvektor								Modifizierter Gewichtsvektor			
x_1	x_2	x_3	x_4	m_1	m_2	m_3	m_4	$g(\mathbf{x})$	$S(\mathbf{y})$	**t**	L	m_1	m_2	m_3	m_4
Wiederholung 1															
0	1	1	1	0	0	0	0	0	*	-1	j	0	-1	-1	-1
0	0	0	1	0	-1	-1	-1	-1	-1	1	j	0	-1	-1	0
1	0	0	1	0	-1	-1	0	0	*	1	j	1	-1	-1	1
1	0	1	1	1	-1	-1	1	1	1	1	n	1	-1	-1	1
0	0	1	1	1	-1	-1	1	0	*	-1	j	1	-1	-2	0
1	1	0	1	1	-1	-2	0	0	*	1	j	2	0	-2	1
1	1	1	1	2	0	-2	1	1	1	-1	j	1	-1	-3	0
0	1	0	1	1	-1	-3	0	-1	-1	-1	n	1	-1	-3	0
Wiederholung 2															
0	1	1	1	1	-1	-3	0	-4	-1	-1	n	1	-1	-3	0
0	0	0	1	1	-1	-3	0	0	*	1	j	1	-1	-3	1
1	0	0	1	1	-1	-3	1	2	1	1	n	1	-1	-3	1
1	0	1	1	1	-1	-3	1	-1	-1	1	j	2	-1	-2	2
0	0	1	1	2	-1	-2	2	0	*	-1	j	2	-1	-3	1
1	1	0	1	2	-1	-3	1	2	1	1	n	2	-1	-3	1
1	1	1	1	2	-1	-3	1	-1	-1	-1	n	2	-1	-3	1
0	1	0	1	2	-1	-3	1	0	*	-1	j	2	-2	-3	0

Wiederholung 3

0	1	1	1	2	-2	-3	0	-5	-1	-1	n	2	-2	-3	0
0	0	0	1	2	-2	-3	0	0	*	1	j	2	-2	-3	1
1	0	0	1	2	-2	-3	1	3	1	1	n	2	-2	-3	1
1	0	1	1	2	-2	-3	1	0	*	1	j	3	-2	-2	2
0	0	1	1	3	-2	-2	2	0	*	-1	j	3	-2	-3	1
1	1	0	1	3	-2	-3	1	2	1	1	n	3	-2	-3	1
1	1	1	1	3	-2	-3	1	-1	-1	-1	n	3	-2	-3	1
0	1	0	1	3	-2	-3	1	-1	-1	-1	n	3	-2	-3	1

Wiederholung 4

0	1	1	1	3	-2	-3	1	-4	-1	-1	n	3	-2	-3	1
0	0	0	1	3	-2	-3	1	1	1	1	n	3	-2	-3	1
1	0	0	1	3	-2	-3	1	4	1	1	n	3	-2	-3	1
1	0	1	1	3	-2	-3	1	1	1	1	n	3	-2	-3	1
0	0	1	1	3	-2	-3	1	-2	-1	-1	n	3	-2	-3	1

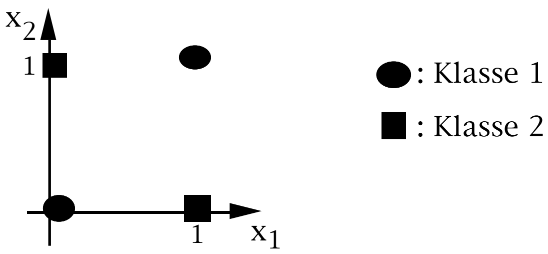
Pereceptron Konvergenz Theorem:

Rosenblatt zeigte in [Rosenblatt, 1959], daß alle Trennebenen g(**x**), die mit Hilfe eines Perceptrons berechnet werden können, mit Hilfe der *fixed-increment* Regel gelernt werden können. Das entsprechende Theorem wird **Perceptron Konvergenz Theorem** genannt. In einem der vorigen Abschnitte wurde erklärt, daß das Gradientenverfahren lokale Minima einer Funktion liefert, die nicht unbedingt globale Minima sein müssen. Rosenblatt zeigte jedoch, daß für das Perceptron Lernen die lokalen Minima von D mit den globalen Minima übereinstimmen.

Das Problem ist jedoch, daß ein Perceptron nur Trennebenen für Klassen berechnen kann, die **linear separabel** sind. (Zwei Mengen von Punkten sind linear separabel, wenn ihre konvexen Hüllen disjunkt sind.)
Dies bedeutet, daß z.B. die XOR-Funktion nicht berechnet und gelernt werden kann:

x_1	x_2	x_1 XOR x_2
0	0	0
0	1	1
1	0	1
1	1	0

Dies kann auch anhand der folgenden Abbildung verdeutlicht werden:



Wir können uns auch anders klar machen, warum die XOR-Funkion weder gelernt noch berechnet werden kann: Wir haben zwei Klassen, die durch zwei Merkmale, x_1 und x_2, beschrieben werden. Die Trennfunktion hat also die Form $g(\mathbf{x}) = m_0 + m_1 x_1 + m_2 x_2$. Setzt man $g(\mathbf{x})=0$, erhält man die Geradengleichung

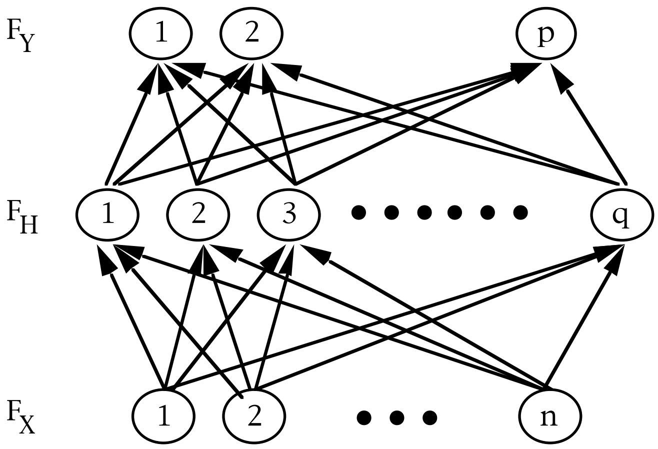
$$x_2 = \frac{m_1}{m_2} x_1 + \frac{m_0}{m_2},$$

die gelernt werden soll. Es gibt aber keine Gerade, die so durch die Punkte gelegt werden kann, daß die Punkte der einen Klasse auf einer Seite und alle anderen Punkte auf der anderen Seite liegen.

Minsky und Papert legten in ihrem Buch 'Perceptrons' [Minsky und Papert, 1990] dar, daß zahlreiche Probleme, wie das XOR-Problem, das *parity*-Problem (bestimme ob ein Bitvektor der Länge n eine gerade oder ungerade Anzahl von Einsen enthält) oder die Bestimmung konvexer Hüllen mit Hilfe des Perceptrons nicht gelöst werden können. Erst in den 80-er Jahren wurde mit dem Backpropagation-Algorithmus ein Lernverfahren für neuronale Netze entwickelt, daß diese Probleme lösen kann.

4.1.2 Backpropagation

Betrachten wir erst einmal das Netzwerkmodell, für das das *Backpropagation*-Verfahren definiert ist. Es ist ein *feedforward*-Netz mit mehr als zwei Ebenen. In der folgenden Skizze ist ein Netz mit drei Ebenen dargestellt.



Die Einheiten einer Ebene sind mit Einheiten der direkt darüber liegenden Ebene verbunden. Wir unterscheiden nun zwischen **externen** und **internen Einheiten** (*hidden units*). Externe Einheiten sind die Ein- und Ausgabeeinheiten in F_X bzw. F_Y. Interne Einheiten sind Einheiten, die weder von der Umgebung Signale erhalten noch Signale an die Umgebung weiterleiten. In der Skizze haben wir eine Ebene mit q internen Einheiten (F_H).

Die Ausgabefunktionen der Eingabeeinheiten sind linear $S(x_i)=x_i$, $i=1,\ldots,n$. Die Ausgabefunktionen für interne Einheiten und Ausgabeeinheiten müssen kontinuierlich und differenzierbar sein. Im Gegensatz zum Perceptron mit seinen diskreten Ausgabeeinheiten, die ihre Ausgabe nach einer Schwellwertfunktion berechnen (nicht differenzierbar), haben wir hier also kontinuierliche Einheiten. Als Ausgabefunktion wird z.B. die logistische Funktion

$$S(h_l) = \frac{1}{1+\exp(-net_l)} \qquad \text{mit} \qquad net_l = \sum_{i=1}^{n} S(x_i) m_{il} + W_l,$$

W_l: konstante Eingabe der internen Einheit $l \in \{1,\ldots,q\}$

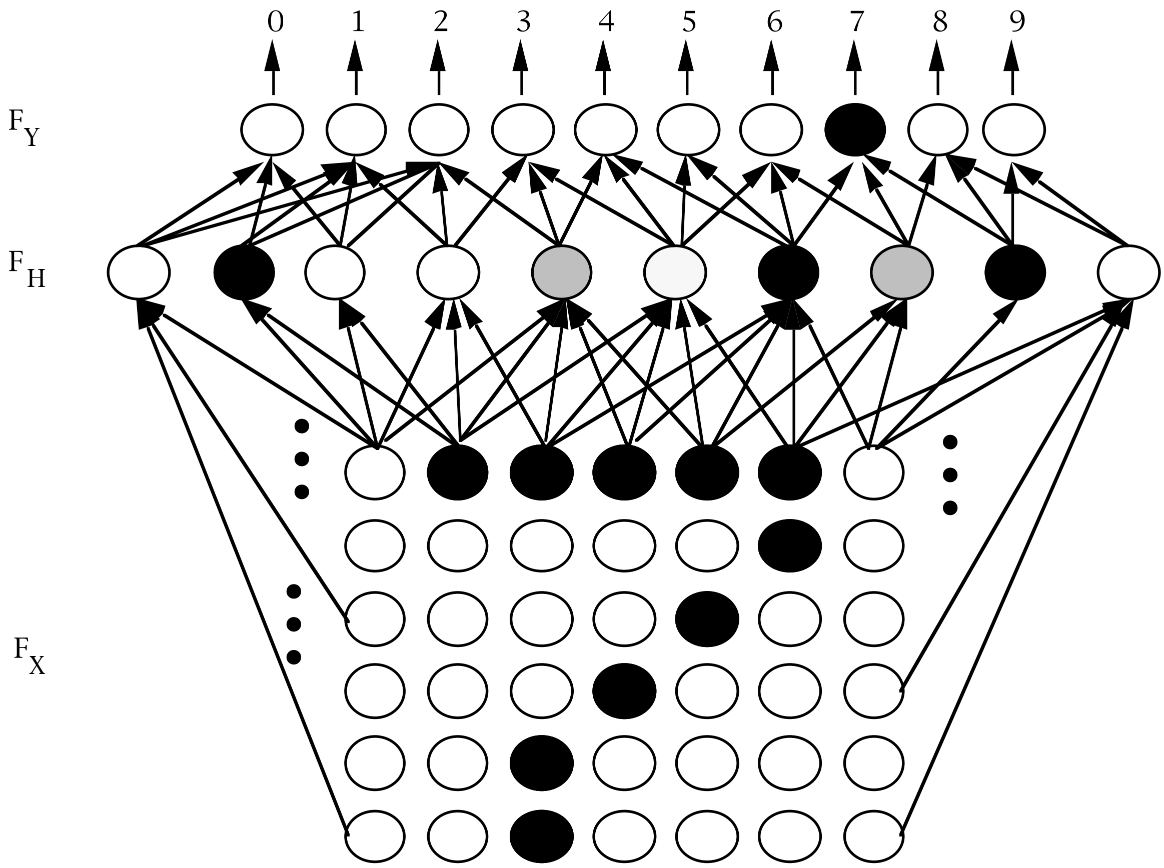
für interne Einheiten h_l und

$$S(y_j) = \frac{1}{1+\exp(-net_j)} \qquad \text{mit} \qquad net_j = \sum_{l=1}^{q} S(h_l)m_{lj} + J_j,$$

J_j: konstante Eingabe der Ausgabeeinheit $j\in\{1,\ldots,p\}$

für Ausgabeeinheiten y_j verwendet.

Die internen Einheiten dienen der Repräsentation komplexer Merkmale, die nicht direkt durch die Eingabemuster **x** dargestellt werden.
Nehmen wir z.B. das in der folgenden Abbildung dargestellte Netzwerk, daß der Erkennung der Ziffern '0' bis '9' dient. Es ist der Übersichtlichkeit wegen nur ein Teil der Verbindungen dargestellt. Prinzipiell ist jede Einheit einer Ebene mit jeder Einheit der direkt darüber liegenden Ebene verbunden. Ein Eingabemuster **x** repräsentiert eine Ziffer (im Beispiel die '7') als Schwarz-Weiß-Bild. Es ist ein Bitmuster, dessen Komponenten angeben, ob das entsprechende Pixel das Bildes schwarz oder weiß ist. Entsprechend gibt die die '7' repräsentierende Ausgabeeinheit die Ausgabe '1' an die Umgebung weiter. Eine interne Einheit kann z.B auf das Vorhandensein einer diagonalen oder horizontalen Linie im Bildmuster reagieren. (Diese speziellen Merkmale sind translationsinvariant, sodaß die Ziffer '7' unabhängig von der Position im zweidimensionalen Raum erkannt werden kann.)



Durch das *Backpropagation*-Verfahren wird diese interne Repräsentation komplexer Merkmale nicht vorprogrammiert, sondern selbständig gelernt.

Vergleichen wir dies kurz mit induktiven Lernverfahren [Morik, 93]. Dort wurde zwischen einer Beschreibungssprache L_E für Beispiele und einer Hypothesensprache L_C für einen Begriff einer Kategorie unterschieden. Gesucht wurde nach einem Ausdruck $c \in L_C$ der alle positiven und keine negativen Beispiele der Kategorie abdeckt. Werden Beispiele durch die Attribute 'Länge' und 'Breite' beschrieben und enthält die Hypothesensprache L_C eine Regel, wie das Attribut 'Fläche' mit Hilfe von 'Länge' und 'Breite' abgeleitet werden kann, können Kategorien prinzipiell durch das Attribut 'Fläche' beschrieben werden bzw. anhand dieses Attributs voneinander abgegrenzt werden. Wenn L_C und L_E also nicht identisch sind, kann eine Kategorie durch Attribute beschrieben werden, die nicht Elemente der Beschreibungssprache für die Beispiele sind. L_C ist Hintergrundwissen, daß beim induktiven Schluß als Theorie bezeichnet wird. In beiden Fällen gilt, daß alle komplexeren Merkmale, die das Lernverfahren berücksichtigen soll bzw. die Regeln zu ihrer Ableitung explizit in L_C oder T dargestellt werden müssen. Entsprechend wächst dann aber der Suchraum für das Lernverfahren.
Beim *Backpropagation*-Verfahren ist die explizite Darstellung/Auflistung der "ableitbaren" Merkmale, dargestellt durch interne Einheiten, nicht nötig. Das Verfahren findet sie selbständig. Durch die Anzahl der internen Einheiten bzw. Ebenen wird lediglich die Anzahl der Merkmale, die entdeckt werden kann, nach oben hin begrenzt. Das Problem ist dann jedoch, daß wir nicht wissen, welche Merkmale die internen Einheiten repräsentieren. Um diese herauszufinden müssen die einzelnen Beispiele mit den gelernten "abgeleiteten" Merkmalen erneut analysiert werden.

Das ***Backpropagation***-Lernverfahren [Rumelhart et al., 1986] löst die folgende, schon bekannte Problemstellung:

Gegeben:

Beispiele $(\mathbf{x}_1,\mathbf{t}_1),\ldots,(\mathbf{x}_m,\mathbf{t}_m)$

Gesucht:

Parameter der Funktion $f:\Re^n \rightarrow \Re^p$ bzw. die Bestimmung der Gewichte des Netzes, so daß $\mathbf{t}_b=S(\mathbf{y}_b)$ mit $\mathbf{y}_b=f(\mathbf{x}_b)=\mathbf{x}_b M$, $b=1,\ldots m$.

Der Lernzyklus besteht aus zwei Schritten:

- **Vorwärtspropagierung** eines Eingabemusters $\mathbf{x}_b$, $b \in \{1,\ldots,m\}$
- **Rückwärtspropagierung** des Fehlers während der die Gewichte modifiziert werden.

Die Frage ist nun, nach welcher Regel die Gewichte modifiziert werden. Bis jetzt haben wir die Delta-Regel

$$\Delta m_{ij}^{(k)} = c^{(k)} * (t_j - S(y_j)) * S(x_i)$$

mit ihrer speziellen Form für das Perceptron kennengelernt. Diese kann jedoch hier nicht ohne weiteres angewendet werden, da die erwünschte Ausgabe t für die internen Einheiten nicht bekannt ist. Statt dessen wird die **verallgemeinerte Delta-Regel** verwendet, die folgende Form hat

$$m_{ij}^{(k+1)} = m_{ij}^{(k)} + \Delta m_{ij}^{(k)}$$

mit

$$\Delta m_{ij} = c^{(k)} * \delta_j * S(x_i)$$

mit

$c^{(k)}$: Lernrate

δ_j: Unterschied zwischen erwünschter und tatsächlicher Ausgabe der Einheit j
$S(x_i)$: Ausgabe der vorgeschalteten Einheit i

Der Faktor δ_j für Gewichte von Verbindungen zu den Ausgabeeinheiten $j \in \{1,...,p\}$ ist so definiert

$$\delta_j = (t_j - S(y_j)) * S'(y_j)$$

Haben wird die logistische Ausgabefunktion für die Ausgabeeinheiten

$$S(y_j) = \frac{1}{1 + e^{-net_j}} \quad \text{mit} \quad S'(y_j) = S(y_j) * (1-S(y_j))$$

erhalten wir

$$\delta_j = (t_j - S(y_j)) * S(y_j) * (1-S(y_j)).$$

Der Faktor δ_l für Gewichte von Verbindungen zu den internen Einheiten $l \in \{1,...,q\}$ ist so definiert:

$$\delta_l = \sum_{j=1}^{p} \delta_j m_{lj} * S'(h_l)$$

Der Faktor δ_l hängt also von den "Fehlern" δ_j der Ausgabeeinheiten ab, die mit der internen Einheit l direkt verbunden sind. Nehmen wir wieder die logistische Ausgabefunktion

$$S(h_l) = \frac{1}{1 + e^{-net_l}} \quad \text{mit} \quad S'(h_l) = S(h_l) * (1-S(h_l))$$

erhalten wir

$$\delta_l = \sum_{j=1}^{p} \delta_j m_{lj} * S(h_l) * (1-S(h_l)).$$

Diese Ausdrücke, δ_j bzw. δ_l, werden in $\Delta m_{ij}^{(k)}$ bzw. $\Delta m_{lj}^{(k)}$ eingesetzt, um $m_{ij}^{(k+1)}$ bzw. $m_{lj}^{(k+1)}$ zu berechnen. Bei den Regeln zur Modifikation der Gewichte werden die Ableitungen der Ausgabefunktionen verwendet. Hier wird klar, warum zu Beginn verlangt wurde, daß die Ausgabefunktionen stetig differenzierbar sein sollen.

Der Algorithmus für das Lernverfahren lautet dann:

1. Initialisiere die Gewichte m_{il} und m_{lj} $i=1,\ldots,n$, $l=1,\ldots,q$, $j=1,\ldots,p$ zwischen Eingabeeinheiten und internen Einheiten bzw. zwischen internen Einheiten und Ausgabeeinheiten zufällig.
2. Wiederhole für jedes Paar $(\mathbf{x}_r,\mathbf{t}_r)$, $r=1,\ldots,m$:

Vorwärtspropagierung:

3. Berechne die Ausgabe der internen Einheiten, z.B. mit

$$S(h_l) = \frac{1}{1 + e^{-net_l}}$$

4. Berechne die Ausgabe der Ausgabeeinheiten z.B. mit

$$S(y_j) = \frac{1}{1 + e^{-net_j}}$$

Rückwärtspropagierung:

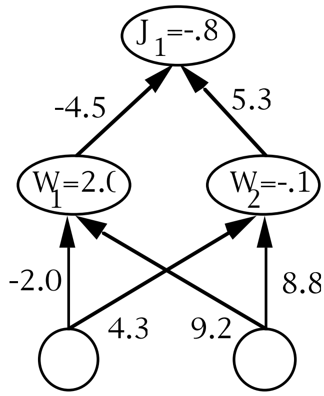
5. Berechne den Fehler δ_j für Ausgabeeinheiten
6. Berechen den Fehler δ_l für die internen Einheiten.
7. Modifiziere die Gewichte der Verbindungen zu den Ausgabeeinheiten.
8. Modifiziere die Gewichte der Verbindungen zu den internen Einheiten.
9. Wiederhole Schritte 2.-8. solange, bis die Gewichte konvergieren oder eine maximale Anzahl von Iterationen durchlaufen wurde.

Hat das Netzwerk mehr als eine Ebene interner Einheiten, müssen für jede der Ebenen die Schritte 3., 6. und 8. durchgeführt werden.

In [Rumelhart et al., 1986] und [Kosko, 1992] wird ausführlich gezeigt, daß auch *Backpropagation* nichts anderes ist, als ein Gradientenverfahren, durch das die Parameter (→Gewichte) der Funktion $f:\Re^n \rightarrow \Re^p$ iterativ approximiert werden, sodaß ein Fehler D minimiert wird. Dieser ist hier als **mittlerer quadratischer Fehler** (MSE: *mean squared error*) von f(**x**) definiert:

$$D = \frac{1}{2} \sum_{l=1}^{m} [\ \mathbf{t}_l - S(\mathbf{y}_l)].$$

Mit Hilfe dieses Verfahrens kann nun unter anderem die Funktion für das XOR-Problem gelernt werden. Es ist klar, daß das Ergebnis von der Anzahl der internen Ebenen und von der Anzahl der internen Einheiten pro Ebene abhängt. Die folgende Abbildung zeigt das Ergebnis für ein Netz mit einer internen Ebene, n=2 Eingabeeinheiten, q=2 internen Einheiten und p=1 Ausgabeeinheit:

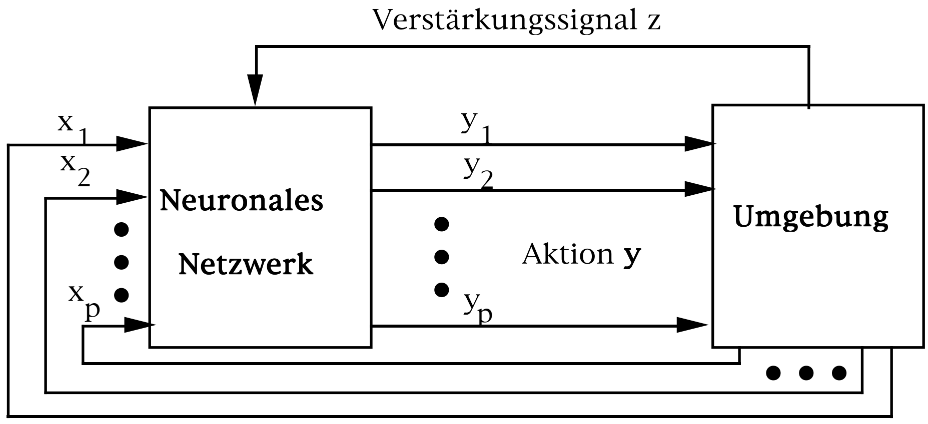


.

4.1.3 Weitere überwachte Lernverfahren

Bis jetzt haben wir Lernverfahren für *feedforward*-Netze kennengelernt. Ein Lernverfahren für autoassoziative *feedback*-Netze ist das von Ackley, Hinton und Sejnowski entwickelte Verfahren für Boltzmann-Maschinen [Ackley et al., 1985].

Ein weiters überwachtes Lernverfahren, daß eine von den bis jetzt besprochenen Verfahren leicht unterschiedliche Zielsetzung hat, ist das von Barto und Sutton entwickelte *Reinforcement Learning* (**Verstärkungs-Lernen**) [Barto et al., 1981,1983]. Ziel ist hier nicht die Speicherung von Ein- und Ausgabepaaren, sondern die Lösung von *Credit-Assignment* Problemen: Wie soll ein System seine Umgebung beeinflussen, um einen gewissen Effekt zu produzieren.
Es wird mit folgendem Modell gearbeitet:



Eine Umgebung liefert einen Beschreibungsvektor $\mathbf{x}$, der den Zustand der Umgebung auf irgendeine Art und Weise widerspiegelt. Dieser dient als Eingabevektor für ein neuronales Netz (Perceptron mit n Ein- und p Ausgabeeinheiten). Das Netz berechnet einen Ausgabevektor $\mathbf{y}$, der eine Aktion darstellt, die in der Umgebung ausgeführt wird. Die Umgebung evaluiert den aus der Aktion resultierenden neuen Zustand $\mathbf{x}_{\mathbf{new}}$ mit einer **Bewertungsfunktion** z, deren Wert positiv ist, wenn die Aktion den Zustand einem Zielzustand näher brachte bzw. negativ, wenn sie den Abstand zum Zielzustand vergrößerte. Dieses Verstärkungssignal wird zusammen mit dem neuen Beschreibungsvektor $\mathbf{x}_{\mathbf{new}}$ zum Netz geleitet. Während der Lernphase modifiziert das Netz seine Gewichte in Abhängigkeit von $\mathbf{x}$, $\mathbf{y}$ und z.

Das Lernproblem ist hier:
Gegeben:

Beispiele $\mathbf{x}_1$, $\mathbf{x}_2$, $\mathbf{x}_3$ …

Bewertungsfunktion z

Gesucht:

Parameter der Funktion $f:\Re^n \rightarrow \Re^p$, für die z einen maximalen Wert annimmt.

Wir haben also keine Ein-/Ausgabepaare gegeben, sondern Eingabemuster $\mathbf{x}$ und eine Bewertungsfunktion z.

Der Lernalgorithmus basiert wieder auf dem Gradientenverfahren:

Wiederhole solange bis ein Maximum der **Bewertungsfunktion** z erreicht worden ist:

1. Umgebung liefert einen Beschreibungsvektor $\mathbf{x}$
2. Netz berechnet einen Ausgabevektor $\mathbf{y}=f(\mathbf{x})$
3. Umgebung wertet den Effekt der Aktion aus $z(\mathbf{y})$
4. Netz ändert die Gewichte entsprechend der Regel

$$m_{ij}^{t+1} = m_{ij}^{t} + c\,(z^{t}-z^{t-1})\,(y_i^{t-1}-y_j^{t-2})\,x_i^{t-1}$$

Das Bemerkenswerte ist, daß hier sowohl bei positiven als auch bei negativen Lernergebnissen gelernt wird. Bei den bisherigen Lernverfahren wurde nur bei negativen Lernergebnissen gelernt, d.h. dann wenn Trainingsbeispiele falsch klassifiziert wurden.

Anwendungsszenarien sind Roboter oder computergestützte Fahrzeuge, die sich in einer eventuell fremden, sich ändernden Umgebung bewegen müssen, ohne mit anderen Gegenständen zu kollidieren, um möglichst schnell zu einem Ziel zu kommen (Navigationsproblem). Es ist nahezu unmöglich, die Umgebung vollständig zu spezifizieren, d.h. alle Beschreibungsvektoren **x** mit den entsprechenden **y** vorzugeben. Statt dessen kennt man Bedingungen, die erfüllt sein müssen (z. B. keine Kollision) und den zu erreichenden Zielzustand. Beides wird mit Hilfe der Bewertungsfunktion ausgedrückt.
Wir haben also neben inhaltsorientierter Speicherung, komb. Optimierungsproblemen und Klassifikationsproblem eine vierte Anwendung von Netzwerken kennengelernt.

4.2 Lernen aus Beobachtungen - Unüberwachtes Lernen

In diesem Abschnitt betrachten wir Methoden zum Lernen aus Beobachtungen. Das Lernproblem kann folgendermaßen formuliert werden:

Gegeben:

m Beobachtungen $(\mathbf{x}_1,\bullet),\ldots,(\mathbf{x}_m,\bullet)$

Gesucht:

Bestimmung der Parameter der Funktion $f:\mathfrak{R}^n \rightarrow \mathfrak{R}^p$ bzw. die Bestimmung der Gewichte des Netzes, so daß sie Cluster der Beobachtungen widerspiegelt. Das Verfahren muß diese Cluster selbständig entdecken und die Parameter so bestimmen, daß gilt:

$f(\mathbf{x}_i)=f(\mathbf{x}_j)$, $i\neq j$, wenn $\mathbf{x}_i$ und $\mathbf{x}_j$ dem gleichen Cluster angehören

und

$f(\mathbf{x}_i)\neq f(\mathbf{x}_j)$, $i\neq j$, wenn $\mathbf{x}_i$ und $\mathbf{x}_j$ verschiedenen Clustern angehören.

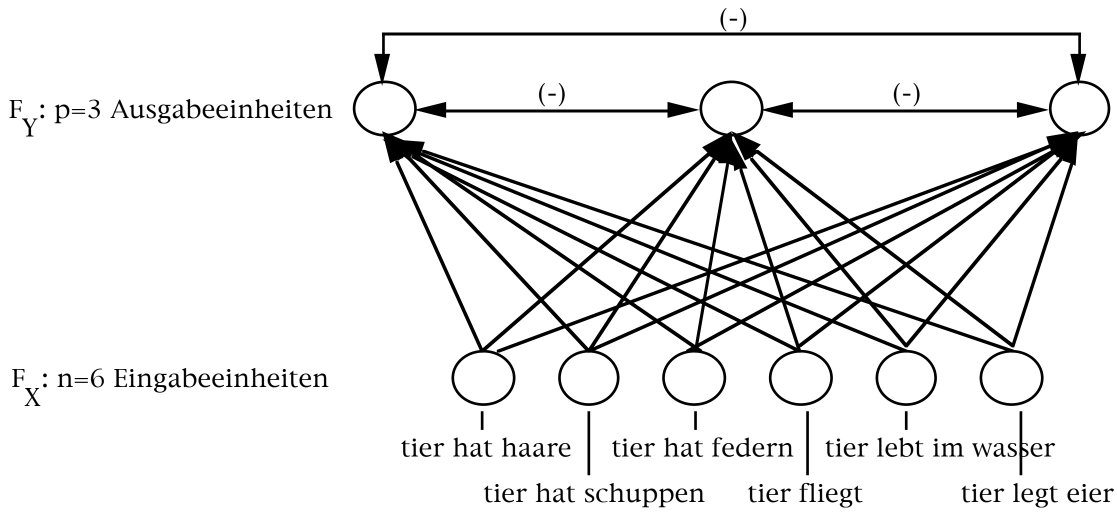
Nehmen wir die folgenden Beobachtungen:

	tier hat haare	tier hat schuppen	tier hat federn	tier fliegt	tier lebt im wasser	tier legt eier
Hund	1	0	0	0	0	0
Fleder-maus	1	0	0	1	0	0
Wal	1	0	0	0	1	0
Kanarien-vogel	0	0	1	1	0	1
Schlange	0	1	0	0	0	1
Alligator	0	1	0	0	1	1

Die Beobachtungen können in drei Kategorien eingeteilt werden: Säugetiere, Reptilien und Vögel. Ziel des unüberwachten Lernens ist es, ohne a priori Informationen über die Klassenzugehörigkeit der Beobachtungen und die Anzahl und Art der Klassen die oben genannten drei Kategorien zu entdecken und zu generalisieren, sodaß Tiere identifiziert werden können, die während der Trainigsphase noch nicht gesehen wurden.

4.2.1 Wettbewerbs-Lernen

Betrachten wir zuerst wieder das Netzwerkmodell, für das das Verfahren definiert ist. Die folgende Abbildung zeigt es an einem Beispiel, bestehend aus n=6 Eingabeeinheiten und p=3 Ausgabeeinheiten.



Die gesuchten Klassen sollen durch Einheitsvektoren dargestellt werden, d.h. für unser Beispiel durch die Ausgabevektoren (1 0 0), (0 1 0) und (0 0 1).
Die Ausgabeeinheiten sind inhibitorisch miteinander verbunden. Wir haben also zum ersten Mal den Fall, daß $\mathbf{Q}\neq 0$. Die Gewichte sind negativ und werden im allgemeinen nach dem folgenden Prinzip festgelegt: Je weiter die Einheiten voneinander entfernt sind, desto kleiner, je näher, desto größer ist der absolute Wert des Gewichts. Durch diese Struktur verhalten sich die Ausgabeeinheiten nach dem **Wettbewerbsprinzip** (*winner-take-all* Prinzip): Nehmen wir an, daß ein bestimmtes Eingabemuster angelegt und zu den Ausgabeeinheiten propagiert wird. Die Ausgabeeinheiten versuchen dann sich gegenseitig zu deaktivieren: Die Ausgabeeinheit mit der höchsten Aktivierung wird seine konkurrierenden Nachbarn in größerem Maße deaktivieren, als diese Konkurrenten diese Ausgabeeinheit. Die Folge ist, daß die schwächer aktivierten Einheiten noch schwächer werden und damit auch ihr inhibierender Effekt auf die Ausgabeeinheit mit dem maximalen Aktivierungswert. Zum Schluß ist diese die einzige, die noch aktiv ist. Das *winner-take-all* Prinzip bewirkt also, daß ein Einheitsvektor als Ausgabemuster produziert wird.

Auf diesem Prinzip basiert das **Wettbewerbs-Lernen** (*Competitive Learning*; [Rumelhart und Zipser, 1986]) zur Bestimmung der Gewichte zwischen Ein- und Ausgabeeinheiten: Es wird ein Eingabevektor angelegt und zu den Ausgabeeinheiten propagiert. Die Ausgabeeinheiten konkurrieren dann miteinander, solange bis eine einzige Einheit aktiv ist und die restlichen inaktiv. Dann werden die Gewichte zwischen den aktiven Eingabeeinheiten und der aktiven Ausgabeeinheit erhöht. Dadurch wird es wahrscheinlicher, daß die aktive Ausgabeeinheit wieder aktiv wird, wenn das Eingabemuster erneut angelegt wird. Dieser Zyklus wird für alle Beobachtungen wiederholt, solange bis die Gewichte konvergieren. Um zu vermeiden, daß eine einzige Ausgabeeinheit immer maximal aktiviert wird, werden in dem Maße in dem Gewichte vergrößert werden andere Gewichte verkleinert. In dem folgenden von Rumelhart und Zipser (86) in den PDP-Bänden vorgestellten Algorithmus für das Wettbewerbs-Lernen wird verlangt, daß die Summe der Gewichte der Verbindungen zu einer Ausgabeeinheit den Wert Eins hat:

$$\sum_{i=1}^{n} m_{ij} = 1, \; j \in \{1,\dots,p\}$$

Wird das Gewicht einer dieser Verbindungen erhöht, müssen andere verkleinert werden, damit die Summe konstant bleibt.

Wettbewerbs-Lernen (*competitive learning*):

1. Lege einen Eingabevektor $\mathbf{x}_l=(x_{l_1},\dots,x_{l_n})$, $l \in \{1,\dots,m\}$ an.
2. Propagiere das Eingabemuster zu den Ausgabeeinheiten und berechne ihre Aktivität.
3. Lasse die Ausgabeeinheiten miteinander konkurrieren, solange bis nur noch eine aktiv ist.
4. Modifiziere die Gewichte der Verbindungen von den Eingabeeinheiten $i=1,\dots,n$ zu der aktivierten Ausgabeeinheit y_j, $j \in \{1,\dots,p\}$ nach folgender Regel:

   $$m_{ij}^{(k+1)} = m_{ij}^{(k)} + \Delta m_{ij}$$

   mit

   $$\Delta m_{ij} = c\,\frac{x_i}{r} - c\, m_{ij}\,, \quad i=1,\dots,n.$$

   Die Konstante c ist eine Lernkonstante≤ 1, r ist die Anzahl der aktiven Eingabeeinheiten des Eingabemusters $\mathbf{x}_l$, $l \in \{1,\dots,m\}$.
5. Wiederhole Schritte 1 bis 4 mehrfach für jedes Eingabemuster $\mathbf{x}_l$, $l \in \{1,\dots,m\}$, solange bis die Gewichte konvergieren oder sich kaum mehr ändern.

Mit diesem Verfahren wurden zahlreiche Probleme erfolgreich gelöst. Es kann für Netzwerke mit mehreren Ebenen verallgemeinert werden. Es gibt für dieses Verfahren jedoch keine genaueren Angaben, die über empirische Experimente hinausgehen und das Konvergenzverhalten bzw. die Bedingungen für die Konvergenz betreffen.

Literatur

[Aarts, Korst, 1989] Aarts E., Korst J. (1989). Simulated Annealing and Boltzmann Machines. John Wiley & Sons, New York.

[Ackley et al., 1985] Ackley D.H., Hinton G.E., Sejnowski T.J. (1985). A Learning Algorithm for Boltzmann Machines. *Cognitive Science* Vol. 9, 147-169.

[Barto et al., 1981] Barto A.G., Sutton R.S., Brouwer P.S. (1981). Associative Search Network: A Reinforcemnet Learning Associative Memory. *Biol. Cybern.* Vol. 40, 201-211.

[Barto et al., 1983] Barto A.G., Sutton R.S., Anderson C.W., (1983). Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problems. *IEEE Trans. on Systems, Man, and Cybernetics* Vol. 13, 834-846, 1983.

[Hebb, 1949] Hebb D.O., (1949). The Organization of Behavior. New York, Wiley.

[Hinton und Sejnowski, 1986] Hinton G.E., Sejnowski, T.J. (1986). Learning and Relearning in Boltzmann machines. In Rumelhart D.E., McClelland J.L., Hrsg., PDP, Vol. 1, Kapitel 7.

[Hopfield, 1982] Hopfield J.J. (1982). Neural Networks and Physical Systems with Emergent Collective Computational Abilities. In *Proc. Natl. Acad. Sci. USA,* Vol. 79, 2554-2558.

[Hopfield, 1984] Hopfield J.J. (1984). Neurons with Graded Response have Collective Computational Properties like those of Two-state Neurons. In *Proc. Natl. Acad. Sci. USA*, Vol. 81, 3088-3092.

[Hopfield und Tank, 1985] Hopfield J.J., Tank D.W. (1985). Neural Computation of Decisions in Optimization Problems. *Biol. Cyber.* Vol 52, 141-152.

[Kemke, 1988] Kemke C. (1988). Der Neuere Konnektionismus: Ein Überblick. Informatik Spektrum, Vol. 11, 143-162.

[Kirkpatrick et al., 1983] Kirkpatrick S., Gelatt C.D., Vecchi M.P. (1983). Optimization by Simulated Annealing. *Science* Vol. 220, 671-680.

[Kohonen, 1984] Kohonen Teuvo (1984). Self-Organization and Associative Memory. Springer Verlag, Berlin.

[Kosko, 1992] Kosko B. (1992). Neural Networks and Fuzzy Systems, A Dynamical Systems Approach to Machine Intelligence. Prentice Hall.

[McCulloch und Pitts, 1943] McCulloch W.S., Pitts W. (1943). A Logical Calculus of the Ideas Immanent in Neural Activity. *Bull. Math. Biophys.* Vol. 5, 115-133.

[Metropolis et al., 1953] Metropolis N., Rosenbluth A., Rosenbluth M., Teller E., Teller A. (1953). Equation of state calculations by fast computing machines. *Journal of Chemical Physics* Vol. 21, 1087-1092.

[Minsky, 1954] Minsky M. (1954). Neural Nets and the Brain Model Problem. Doctoral Dissertation, Princeton University.

[Minsky und Papert, 1990] Minsky M., Papert S. (1990). Perceptrons. MIT Press (expanded edition of the 1969 edition) .

[Morik, 1993] Morik K. (1993): Maschinelles Lernen. LS-8 Report 1, Universität Dortmund.

[Nilsson, 1980] Nilsson N.J. (1980). Principles of Artificial Intelligence. Tioga Publishing Co.

[Nilsson, 1990] Nilsson N.J. (1990). The Mathematical Foundations of Learning Machines. Morgan Kaufmann (reprint of the 1965 edition published as Learning Machines).

[Rosenblatt, 1959] Rosenblatt F. (1959). Two Theorems of Statistical Separability in the Perceptron. In Mechanisation of Thought Processes, Proc. Symp. at Nat. Phys. Lab., London.

[Rosenblatt, 1962] Rosenblatt F.(1962). Principles of Neurodynamics. Spartan Books, New York.

[Rumelhart et al., 1986] Rumelhart D.E., Hinton G.E., Williams R.J. (1986). Learning Internal Representations by Error Representation. In Rumelhart D.E., McClelland J.L., Hrsg., PDP, Vol. 1, Kapitel 8.

[Rumelhart und McClelland, 1986] Rumelhart D.E., McClelland J.L., Hrsg., (1986). Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. I and II, MIT Press.

[Rumelhart und Zipser, 1986] Rumelhart D.E., Zipser D. (1986). Feature Discovery by Competitive Learning. In Rumelhart D.E., McClelland J.L., Hrsg., PDP, Vol. 1, Kapitel 6.

Index