Online-Ergänzung zum Artikel

Computer lernen nach dem Vorbild des Gehirns - Entwicklung eines künstlichen neuronalen Netzwerkes

F. G. Hüske, W. Baumgartner, I. Heil, J. Bohrmann

6. Programmierung eines einschichtigen *feed-forward*-Netzes

Ziel der folgenden Anleitung ist es, ein Perzeptron oder Adaline sowie den dazu gehörigen Lernalgorithmus zu entwerfen und in ein Computerprogramm (z. B. zur Mustererkennung) zu übertragen. Die Erarbeitung kann beispielsweise im Rahmen eines Bionik- oder Neurobiologie-Projekts zur selbstständigen Entwicklung eines KNN in der gymnasialen Oberstufe erfolgen oder auch - in ausgewählten Teilen - in den regulären Unterricht Eingang finden [8].

6.1 Die Software *Octave*

Die Übertragung des Gedankenmodells in ein Computerprogramm soll mit Hilfe der Software *Octave*, einem numerischen Berechnungs- und Simulationswerkzeug, erfolgen. Man kann auch die Version *QtOctave*, eine benutzerfreundliche Oberfläche für *Octave* [9], verwenden. *Octave* ist eine frei erhältliche Software zur Lösung mathematischer Probleme, die eine sogenannte „Skriptsprache“ [10] verwendet. Hierbei handelt es sich um eine Computersprache, wie sie häufig für kleinere, überschaubare Programmieraufgaben benutzt wird [11]. Skriptsprachen ermöglichen einen leichteren Einstieg in das Programmieren als andere, zumeist kompliziertere Programmiersprachen. Im Fall der Erstellung eines KNN zur Mustererkennung können sowohl die Einzelteile des KNN als auch die zugehörige Lernregel durch mathematische Formeln beschrieben werden, die sich mit *Octave* lösen lassen.

Eine detaillierte Einführung in die sehr umfangreiche Skriptsprache ist hier nicht erforderlich, da man sich darüber - bei Bedarf - auf diversen Internetseiten in Tutorials informieren kann (z. B. [12], [13]). Die für die Umsetzung des KNN und der Lernregel benötigten Befehle werden im Folgenden jeweils an den Stellen erklärt, an denen sie zum ersten Mal gebraucht werden.

Die beiden wichtigsten Elemente der Bedienoberfläche *Octave* sind die Elemente „Octave Terminal“ und „Editor“. Im „Octave Terminal“ können in der „Command line“ die Befehle der Skriptsprache eingegeben werden. Es lassen sich beispielsweise beliebige Matrizen oder Vektoren erstellen und innerhalb der Benutzeroberfläche berechnen. Im „Editor“ können Programme geschrieben werden, welche im „Octave Terminal“ aufgerufen werden können, um Berechnungen durchzuführen.

6.2 Implementierung eines einschichtigen *feed-forward*-Netzes

Ein einschichtiges digitales *feed-forward*-Netz ist in *Octave* sehr einfach zu implementieren. Die entsprechende Funktion (das Programm in *Octave*) lautet folgendermaßen:

```
function [o,a]=feedforw(x,W,lambda)
a=W*x;                    % Berechnung des Aktivitätsvektors
o=a>lambda;               % Berechnung des Outputvektors
```

Texte hinter „%“ sind Kommentare, die in *Octave* nicht ausgeführt werden. Jede Funktion in *Octave* beginnt mit dem Befehl „function“. Hinter diesem Befehl werden die Output-Variablen der Funktion in eckigen Klammern, hier Outputvektor *o* und Aktivitätsvektor *a* angegeben, in welche die Ergebnisse der Funktion ausgegeben werden sollen. Auf diese Variablen folgen, durch ein Gleichheitszeichen getrennt, der Name der Funktion, in unserem Fall „feedforw“, und dahinter, in runden Klammern, die Input-Variablen der Funktion, die man dem Programm übergeben muss, der Inputvektor *x*, die Gewichtsmatrix *W* und die Schwelle *lambda*. Zunächst wird in der zweiten Programmzeile die Aktivität berechnet indem die Matrix *W* mit dem Vektor *x* multipliziert wird. Der Stern * steht in *Octave* für eine Matrizenmultiplikation. In der dritten Zeile wird eine „Bool'sche Operation“ durchgeführt, wobei für jeden Eintrag in den Aktivitätsvektor *a* geprüft wird, ob der Eintrag größer als *lambda* ist. Ist dies der Fall, so ist das Ergebnis 1, ansonsten 0. Das Ergebnis ist der Outputvektor, in dem Nullen oder Einsen stehen, je nach dem, ob die Aktivität unter- oder überkritisch ist.

Da es sich bei der Eingabe um einen Spaltenvektor und bei der Gewichtung um eine Matrix handelt, muss darauf geachtet werden, dass die Gewichtsmatrix so gewählt ist, dass die Gesetze der Matrizenmultiplikation nicht verletzt werden. Aus diesem Grund muss die Spaltenanzahl der Gewichtsmatrix gleich der Zeilenanzahl des Eingabevektors sein. Das beschriebene Programm ist ein allgemeines Perzeptron, d. h., ein einschichtiges KNN, mit dem man einfache Logik-Funktionen (wie in Abschnitt 5.1 beschrieben) implementieren kann, oder auch eine Maschinensteuerung, eine Bilderkennung oder eine Spracherkennung. Das große „Geheimnis“ dabei ist die Matrix der Gewichte *W*.

6.3 Der Lernalgorithmus für ein Perzeptron

Um die Gewichtsmatrix *W* zu erhalten, benötigt man das unten angegebene *Octave*-Programm „train“, eine Matrix „Mein“, in deren Spalten die zu trainierenden Eingabemuster stehen, und eine Matrix „Maus“, in deren Spalten die dazu gehörigen zu trainierenden Ausgabemuster stehen. Dies können beliebige Muster sein. Das Programm trainiert dann das Perzeptron und liefert die zugehörige Gewichtsmatrix, sofern das Problem durch ein Perzeptron prinzipiell lösbar ist (siehe unten).

```
function W=train(Mein,Maus,lambda,sigma)

    SMein=size(Mein);                                  %Größe der Eingabematrix ermitteln
    SMaus=size(Maus);                                  %Größe der Ausgabematrix ermitteln

    W=zeros(SMaus(1),SMein(1));                        %Gewichtsmatrix mit Nullmatrix initialisieren

    Anz=size (Mein,2);                                 %Anzahl der Muster bestimmen

    Abweichung=1;                                      % Hilfsvariable, ob Muster noch nicht fehlerfrei
    Durchlauf=0;                                       % Hilfsvariable: Zahl der Lernschritte
    MaxDurchlauf=10000;                                % Maximal erlaubte Zahl der Lernschritte

    while (Abweichung==1) & (Durchlauf<=MaxDurchlauf)  % solange noch nicht alle Muster fehlerfrei
            Abweichung=0;                              % Hilfsvariable auf 0 setzen
            Durchlauf=durchlauf+1;

            for i=1:Anz                                % Alle Muster durchgehen
                    [o,a]=feedforw(Mein(:,i),W,lambda); % Neuronales Netz aufrufen

                            delta=Maus(:,i)-o;                 % Fehler delta bestimmen

                    if norm(delta,inf)>2*sigma         % Falls noch irgendwo ein Fehler
                                    Abweichung=1;              % Hilfsvariable auf 1 setzen
                    end

                    w=w+sigma*delta*Mein(:,i)';                % Lernen gemäß delta-Lernregel

            end
    end
```

Das Programm „train“ kann leicht zum Training eines Adaline modifiziert werden, indem der Fehler *delta* nicht als Maus(:,i)-o berechnet wird, sondern als Maus(:,i)-a. Es ist empfehlenswert, dafür ein eigenes Programm mit einem anderen Namen zu schreiben. Für die Generierung von Matrizen und Vektoren, die Befehle sowie die Indizierung der Vektoren und Matrizen sei auf entsprechende Internet-Tutorials zur Software *Octave* verwiesen (z. B. [10], [12]).

6.4 Einige Beispiele

Betrachten wir zunächst das in Abschnitt 5.1 dargestellte Beispiel eines KNN mit zwei Eingängen und zwei Ausgängen, wobei auf dem ersten Ausgang ein logisches AND und auf dem zweiten Ausgang ein logisches NOR der beiden Eingänge realisiert sein soll. Es wird ein zusätzlicher Eingang, der BIAS benötigt, um das NOR realisieren zu können. Mit den oben beschriebenen Programmen können das Training und die Anwendung unter *Octave* wie folgt aussehen:

```
> Musterein=[0,0,1,1;0,1,0,1;1,1,1,1]
Musterein =

   0   0   1   1
   0   1   0   1
   1   1   1   1
```

```
> Musteraus=[0,0,0,1;1,0,0,0]
Musteraus =

  0  0  0  1
  1  0  0  0

> W=train(Musterein,Musteraus,0.2,1)
W =

  0.40000   0.40000   0.40000
 -0.20000  -0.20000   1.20000

> knn([1;1;1],W,1)
ans =

  1
  0
```

Hinter den Zeichen „>“ sind die Eingaben aufgeführt. Die Ausgaben, die *Octave* generiert, sind in Blau geschrieben. Man kann die Ausgaben der Ergebnisse eines Befehls durch Beendigung der Zeile mit einem Semikolon („;“) unterdrücken. Dies wurde hier nicht gemacht, um die Zwischenergebnisse (blau) sehen zu können. Zuerst wurde die Matrix der Eingabemuster „Musterein“ eingegeben. Die einzelnen Spalten enthalten die zu trainierenden Eingangsvektoren, wobei die letzte Zeile immer 1 ist (der BIAS). Danach wurden die Soll-Ausgabemuster „Musteraus“ gemäß der Wahrheitstabelle in Abschnitt 5.1 eingegeben. Durch Aufrufen des Trainingsprogramms „train“ wird die Gewichtsmatrix ermittelt. Hier wurde mit der Perzeptron-Lernregel gearbeitet, bei der der Fehler als Differenz zwischen Soll- und Ist-Output bestimmt wird. Dem Programm „train“ werden die zu trainierenden Eingabe- und Ausgabemuster sowie die Lernrate σ=0,2 und die Schwelle λ =1 übergeben. Die entsprechende Gewichtsmatrix wird in die Variable *W* geschrieben. Mit dieser Gewichtsmatrix wird dann „knn“ für die eigentliche Anwendung aufgerufen. In diesem Beispiel mit x_1=1, x_2=1 liefert das KNN, wie nicht anders zu erwarten, o_1=1, o_2=0, was ja dem trainierten Muster entspricht.

Dieses Beispiel war relativ trivial und noch in allen einzelnen Schritten „per Hand“ nachvollziehbar. Betrachten wir nun als etwas komplizierteres Beispiel ein Problem der Mustererkennung. In einem kleinen Pixelbild von 5x5 Pixeln soll geprüft werden, ob als Symbol ein „x“ oder ein „o“ dargestellt ist. Die Eingabemuster sind die Pixel des Bildes, die in einen Vektor geschrieben werden müssen, um die Daten in eine für das KNN verwertbare Form zu bringen: ein gesetzter Pixel ist 1, ein nicht gesetzter Pixel 0. Wir wollen zwei Outputs erzielen: der erste soll 1 sein, wenn ein „x“ im Bild dargestellt ist und ansonsten 0, der zweite soll 1 sein wenn ein „o“ dargestellt ist und ansonsten 0. Die entsprechende Eingabe in *Octave* kann folgendermaßen aussehen:

```
> cross=[1,0,0,0,1;0,1,0,1,0;0,0,1,0,0;0,1,0,1,0;1,0,0,0,1]
cross =

  1  0  0  0  1
  0  1  0  1  0
  0  0  1  0  0
  0  1  0  1  0
  1  0  0  0  1
```

```
> circle=[0,1,1,1,0;1,0,0,0,1;1,0,0,0,1;1,0,0,0,1;0,1,1,1,0]
circle =

  0  1  1  1  0
  1  0  0  0  1
  1  0  0  0  1
  1  0  0  0  1
  0  1  1  1  0

> Musterein=[cross(:),circle(:)] ;
> Musteraus=[1,0;0,1];
> W=train(Musterein,Musteraus,0.2,1);
```

Es werden zunächst die beiden zu lernenden Symbole eingegeben, was der Übersichtlichkeit halber getrennt erfolgt. Das erste Symbol ist „x", hier in die Variable „cross" geschrieben. Danach wird das Symbol „o" in die Variable "circle" geschrieben. Die Matrix der zu trainierenden Eingabemuster „Musterein" wird dann aus „cross" und „circle" gebildet, indem die einzelnen Spalten dieser Muster untereinander in jeweils eine Spalte von „Musterein" geschrieben werden, was durch „cross(:)" und „circle(:)" erfolgt. Auf die Ausgabe wurde hier aus Platzgründen verzichtet, indem sie durch Beendigung des Befehls mit „;" unterdrückt wurde: „Musterein" hätte zwei Spalten mit 25 Zeilen ergeben. Danach wird die Matrix der korrespondierenden Ausgabemuster eingegeben, und die Matrix der Gewichte *W* wird berechnet. Nun ist unser Programm aus dem ersten Beispiel in der Lage, eine Mustererkennung durchzuführen. In Abbildung 4 sind die Ergebnisse dargestellt. (Hinweis: Mit dem Befehl „reshape" kann man aus den Vektoren, die das KNN ausgibt, wieder 5x5-Matrizen generieren, die sich dann graphisch darstellen lassen.)

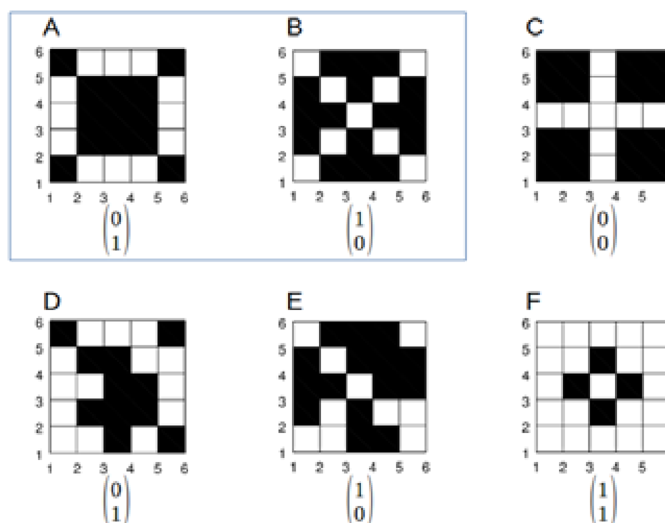


Abb. 4: Ein Beispiel zur Mustererkennung. Die umrahmten Muster „o" (A) und „x" (B) wurden gelernt. Die entsprechenden Outputvektoren sind unter den Mustern dargestellt. Der erste Output zeigt, ob im Eingabemuster ein „x" auftritt, der zweite Output gibt an, ob ein „o" im Input zu finden ist. Ein Muster wie in C, das definitiv weder ein „o" noch ein „x" enthält, führt zu $o_1=o_2=0$. Gestörte Muster wie in D und E, bei welchen einzelne Pixel zufällig verändert wurden, werden korrekt erkannt. Ein Muster, das die Symbole „o" und „x" überlagert enthält (F), liefert $o_1=o_2=1$.

Es folgt ein weiteres Beispiel, in dem das Perzeptron bzw. Adaline als „Autoassoziativspeicher" verwendet werden soll, was bedeutet, dass die Soll-Ausgabe gleich der ungestörten Eingabe ist. In 10x10-Pixel-Bildern wurden die Ziffern 0 bis 9 dargestellt (0= nicht gesetzter Pixel, 1= gesetzter Pixel). Auf die Eingabe eines solchen Musters soll das KNN dasselbe Muster als Ausgabe liefern. Die Muster wurden, wie oben beschrieben, gelernt, indem „knn" aufgerufen wurde, wobei Musteraus=Musterein. In den Abbildung 5 und 6 ist dargestellt, welche Ergebnisse man erhält, wenn man einem trainierten KNN gestörte Daten - hier mithilfe von Zufallszahlen verrauschte Daten - anbietet. Im ersten Beispiel (Abb. 5) wurde die Ziffer 9 mit einem Signal-zu-Rausch-Verhältnis von 3 „verrauscht". Dieses verrauschte Bild wurde dem Perzeptron und dem Adaline übergeben. Die Ergebnisse sind in beiden Fällen saubere Bilder der Ziffer 9. Es ist der Zweck eines Autoassoziativspeichers, gestörte oder unvollständige Daten korrekt zu erkennen.

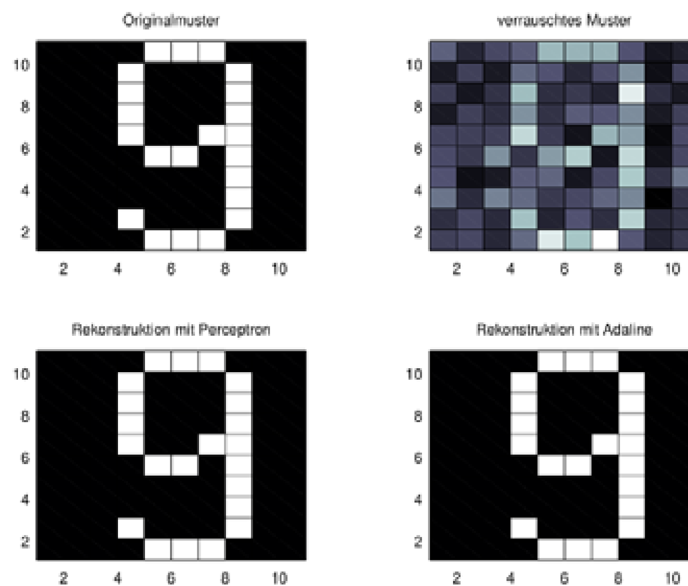


Abb. 5: Ein Beispiel für ein KNN als Autoassoziativspeicher. Die Ziffern 0 bis 9 wurden gelernt. Auf die Eingabe eines ungestörten Trainingsmusters gibt das KNN dieses Muster als Output aus. Im Beispiel wurde das Muster der Ziffer 9 leicht verrauscht, und das verrauschte Muster wurde einem trainierten Perzeptron und einem trainierten Adaline als Input gegeben. Die Ausgaben zeigen in beiden Fällen das korrekt rekonstruierte Muster.

In einem weiteren Beispiel (Abb. 6) wurde die Ziffer 3 mit einem Signal-zu-Rausch-Verhältnis von 2 verrauscht. Es ist hier schwierig, das ursprüngliche Muster in dem verrauschten Bild zu erkennen.

Die Ergebnisse von Perzeptron und Adaline zeigen hier deutliche Unterschiede. Während das Perzeptron offensichtlich Fehler macht, erkennt Adaline das Muster völlig korrekt. Dies ist nicht unerwartet und kann mit vielen Beispielen quantifiziert werden. Adaline hat „exakter" gelernt und nicht nur soweit, dass die Trainingsmuster „gerade so" erkannt werden. Dies hat jedoch seinen Preis, denn der Lernaufwand ist deutlich höher. Mit Hilfe der *Octave*-Befehle „tic" und „toc" zum Starten und Stoppen eines Timers kann der Unterschied gemessen

werden. Auf einem *dual-core-* Pentium-Prozessor mit 2 GHz benötigte das Trainieren eines Perzeptrons etwa 0,02 Sekunden, während das Adaline etwa 40 Sekunden trainiert wurde.

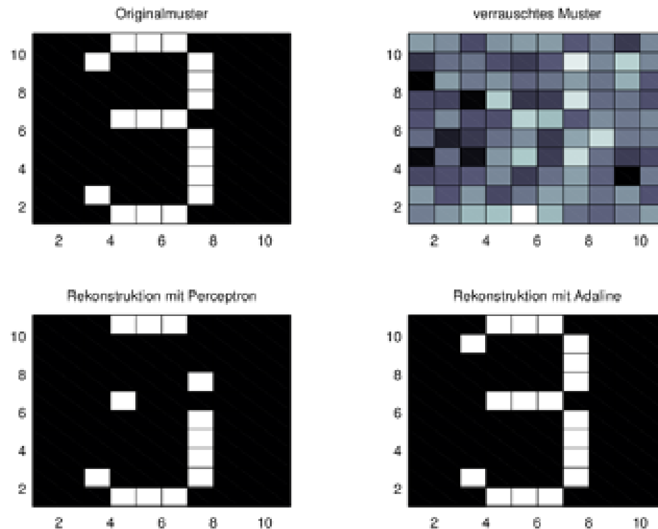


Abb. 6: Beispiel für ein KNN als Autoassoziativspeicher, dessen Zweck es ist, gestörte oder unvollständige Daten korrekt zu erkennen. Das stark verrauschte Muster der Ziffer 3 wurde einem (kurz) trainierten Perzeptron und einem (lang) trainierten Adaline als Input gegeben. Während Adaline das Muster korrekt rekonstruiert, finden sich in der Rekonstruktion des Perzeptrons einige gravierende Fehler.

Analogie: *Es ist offensichtlich, dass es sehr viel schneller geht, wenn man nur für ein „Ausreichend" in der Prüfung lernt, im Vergleich zum Lernen für ein „Sehr gut". Dabei ist zu betonen, dass nur das Training – nicht die Lösung der Aufgabe - entsprechend länger dauert. Denn die Verarbeitung, also die eigentliche Mustererkennung im KNN, ist exakt gleich schnell.*

Die Entscheidung, ob ein Perzeptron oder ein Adaline für ein praktisches Problem Verwendung findet, ist im Einzelfall zu klären. Beide KNN haben ihre Stärken und Schwächen.

Zuletzt sei noch ein kleines, aber sehr lehrreiches Beispiel vorgestellt. Es soll ein Perzeptron oder Adaline darauf trainiert werden, ein logisches XOR (exklusives ODER, ENTWEDER-ODER) zu realisieren. Die Wahrheitstabelle sieht wie folgt aus:

x_1	x_2	o
0	0	0
1	0	1
0	1	1
1	1	0

Gibt man die aus dieser Wahrheitstabelle generierten Muster in ein Trainingsprogramm ein, so kommt es zu keiner „Konvergenz", d. h., das Programm bricht ab, wenn die maximal erlaubte Anzahl an Lernschritten erreicht ist. Würde man die Anzahl der erlaubten Schritte ins Unendliche erhöhen, so würde das Programm unendlich lange laufen, ohne ein Ergebnis zu

liefern. Man kann auch von Hand versuchen das XOR-Problem mit einem Neuron zu lösen und wird dabei scheitern. Dieses Beispiel soll einerseits die Grenzen und andererseits die erweiterten Möglichkeiten der KNN aufzeigen. Einschichtige *feed-forward*-Netze, wie das einlagige Perzeptron oder Adaline, können das XOR nicht realisieren, was jedoch ein einfaches zweilagiges Netz sehr wohl kann. Ein Beispiel für solch ein Netz ist in Abbildung 7 dargestellt.

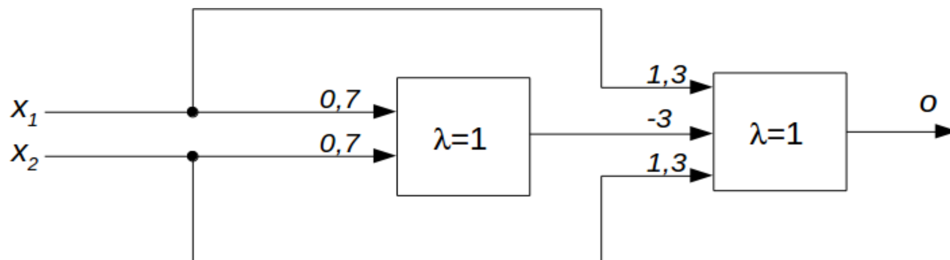


Abb. 7: Ein einfaches zweischichtiges KNN, welches das XOR (exklusives ODER, ENTWEDER-ODER) realisiert.

Ein einschichtiges KNN kann nur „linear separable“ Probleme lösen, d. h. Probleme, bei welchen die Eingabevektoren, welche 0 liefern sollen, und die Eingabevektoren, welche 1 liefern sollen, durch eine „Hyperebene“ des entsprechend dimensionalen Raumes getrennt werden können. Um diese Einschränkung zu überwinden, muss man mehrlagige KNN verwenden. Zweilagige Netze können „konvexe Separationen“ durchführen, während ab drei Lagen prinzipiell jedes Problem gelöst werden kann. Manchmal ist eine vierte Lage für eine schnellere Konvergenz des Lernalgorithmus sinnvoll. Allerdings muss der Lernalgorithmus für mehrlagige Netzwerke adaptiert werden. Man verwendet meist das „Backpropagation“ (verallgemeinerte Delta-Lernregel), ein Verfahren, bei welchem die Fehler einer Schicht an die davor liegende Schicht von Neuronen zurückgegeben werden. Eine genaue Beschreibung dieser Methode würde zwar den Rahmen dieses Artikels sprengen, doch sollte hier aufgezeigt werden, dass es viele Ansätze für eine Weiterführung sowie für Diskussionen und Vergleiche mit den natürlichen Vorbildern gibt.

Weitere Beispiele finden sich in der Literatur (z. B. [3], [4], [5]) und *online* zuhauf. So kann beispielsweise eine Zeitreihenanalyse von Aktienkurs-Verläufen mit KNN erfolgen oder die Steuerung autonomer Fahrzeuge. Auch eine Erweiterung auf mehrschichtigen Netze oder auf analoge Neuronen ist prinzipiell möglich.

Literatur

[1] Brunak, S., Lautrup, B. (1993) Neuronale Netze: Die nächste Computer-Revolution. München u.a.: Hanser

[2] http://www.golem.de/news/maschinenlernen-google-x-simuliert-gehirn-mit-16-000-prozessorkernen-1206-92793.html (19.05.2014)

[3] Hamilton, P. (1993) Künstliche neuronale Netze. Grundprinzipien, Hintergründe, Anwendungen. Berlin: vde-Verlag

[4] Schöneburg, E., Hansen, N., Gawelczyk, A. (1990) Neuronale Netzwerke: Einführung, Überblick und Anwendungsmöglichkeiten. Burgthann: Markt- und Technik-Verlag

[5] Zell, A. (1997) Simulation neuronaler Netze. München: Oldenbourg-Verlag

[6] Kandel, E.R., Schwartz, J.H., Jessell, T.M. (1995) Neurowissenschaften: eine Einführung. Heidelberg u.a.: Spektrum Akademischer Verlag

[7] Rosenblatt, F. (1958) The Perzeptron: a probabilistic model for information storage and organization in the brain. Psychological Review, 65(6), p. 386-408

[8] Hüske, F. (2012) Bionik im Unterricht: Entwicklung eines Schulexperiments zur Funktion künstlicher neuronaler Netze. Staatsexamensarbeit, Fakultät für Mathematik, Informatik und naturwissenschaften, RWTH Aachen

[9] http://wwwmath.uni-muenster.de/num/Vorlesungen/MATLAB-Kurs_SS12/Skript/matlab-einfuehrung.pdf (05.08.2014)

[10] http://math.jacobs-university.de/oliver/teaching/tuebingen/octave/octave/octave.html#SECTION00021000000000000000 (19.05.2014)

[11] http://www.uni-protokolle.de/Lexikon/Skriptsprache.html (19.05.2014)

[12] http://www.christianherta.de/octaveMatlabTutorial.html (19.05.2014)

[13] http://www.uni-koblenz.de/~agas/lehre/ss03/sfm/octave/octave-tutorial.pdf (19.05.2014)

[14] http://www.oszkim.de/materi/edemi/grdglied.html (05.08.2014)

Online-Ergänzung zum Artikel

Computer lernen nach dem Vorbild des Gehirns - Entwicklung eines künstlichen neuronalen Netzwerkes

F. G. Hüske, W. Baumgartner, I. Heil, J. Bohrmann

Arbeitsmaterial 2

1. Anwendung der Delta-Lernregel bei einer UND-Schaltung (Perzeptron):

UND-Schaltung			Schwellenwert: λ=1		Lernrate: σ=0,3							
Input		Soll-Output	Alte Gewichte		gewichteter Input		Summe	Ist-Output	Fehler	Korrektur	Neue Gewichte	
x1	x2	z(j)	w1	w2	x1*w1	x2*w2	a	o(j)	δ(j)	σ*δ(j)	w1	w2
								wenn a>λ dann 1, sonst 0	z(j)-o(j)			
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	0	0	0
1	1	1										
0	0	0										
1	0	0										
0	1	0										
1	1	1										
0	0	0										
1	0	0										
0	1	0										
1	1	1										
0	0	0										
1	0	0										
0	1	0										
1	1	1										

w = Gewicht,

x = Input,

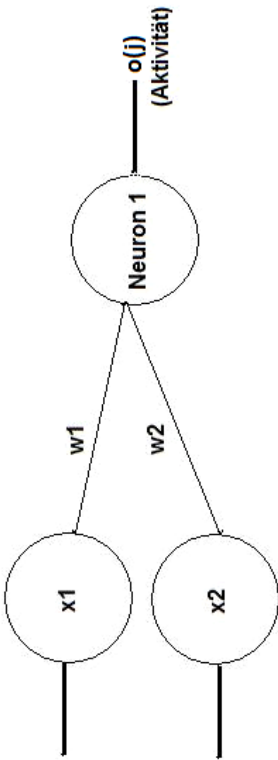
σ = Lernrate (Sigma),

z = gewünschter Output,

o = tatsächlicher Output,

δ = Fehler (Delta)

$$w_{ij}\,(neu) = w_{ij}\,(alt) + x_j\,\sigma\,(z_i - o_i) = w_{ij} + x_j\,\sigma\,\delta_i$$



Arbeitsmaterial 2 (MUSTERLÖSUNG)

1. Anwendung der Delta-Lernregel bei einer UND-Schaltung (Perzeptron):

UND-Schaltung			Schwellenwert: λ=1		Lernrate: σ=0,3							
Input		Soll-Output	Alte Gewichte		gewichteter Input		Summe	Ist-Output	Fehler	Korrektur	Neue Gewichte	
x1	x2	z(j)	w1	w2	x1*w1	x2*w2	a	o(j)	δ(j)	σ*δ(j)	w1	w2
								wenn a>λ dann 1, sonst 0	z(j)-o(j)			
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	0	0	0
1	1	1	0	0	0	0	0	0	1	0,3	0,3	0,3
0	0	0	0,3	0,3	0	0	0	0	0	0	0,3	0,3
1	0	0	0,3	0,3	0,3	0	0,3	0	0	0	0,3	0,3
0	1	0	0,3	0,3	0	0,3	0,3	0	0	0	0,3	0,3
1	1	1	0,3	0,3	0,3	0,3	0,6	0	1	0,3	0,6	0,6
0	0	0	0,6	0,6	0	0	0	0	0	0	0,6	0,6
1	0	0	0,6	0,6	0,6	0	0,6	0	0	0	0,6	0,6
0	1	0	0,6	0,6	0	0,6	0,6	0	0	0	0,6	0,6
1	1	1	0,6	0,6	0,6	0,6	1,2	1	0	0	0,6	0,6
0	0	0	0,6	0,6	0	0	0	0	0	0	0,6	0,6
1	0	0	0,6	0,6	0,6	0	0,6	0	0	0	0,6	0,6
0	1	0	0,6	0,6	0	0,6	0,6	0	0	0	0,6	0,6
1	1	1	0,6	0,6	0,6	0,6	1,2	1	0	0	0,6	0,6

$$w_{ij}\,(neu) = w_{ij}\,(alt) + x_j\,\sigma\,(z_i - o_i) = w_{ij} + x_j\,\sigma\,\delta_i$$

w = Gewicht,

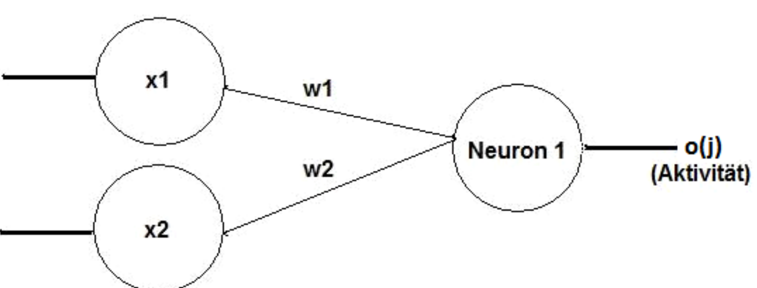
x = Input,

σ = Lernrate (Sigma),

z = gewünschter Output,

o = tatsächlicher Output,

δ = Fehler (Delta)

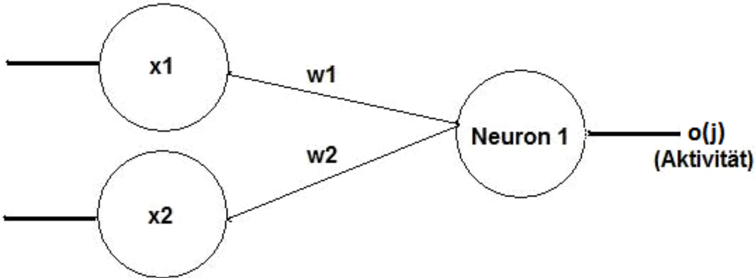


Arbeitsmaterial 2

2. Anwendung der Delta-Lernregel bei einer ODER-Schaltung (Perzeptron):

ODER-Schaltung			Schwellenwert: λ=1		Lernrate: σ=0,3							
Input		Soll-Output	Alte Gewichte		gewichteter Input		Summe	Ist-Output	Fehler	Korrektur	Neue Gewichte	
x1	x2	z(j)	w1	w2	x1*w1	x2*w2	a	o(j)	δ(j)	σ*δ(j)	w1	w2
								wenn a>λ dann 1, sonst 0	z(j)-o(j)			
0	0	0	0	0								
1	0	1										
0	1	1										
1	1	1										
0	0	0										
1	0	1										
0	1	1										
1	1	1										
0	0	0										
1	0	1										
0	1	1										
1	1	1										
0	0	0										
1	0	1										
0	1	1										
1	1	1										

$$w_{ij}\,(neu) = w_{ij}\,(alt) + x_j\,\sigma\,(z_i - o_i) = w_{ij} + x_j\,\sigma\,\delta_i$$



w = Gewicht,

x = Input,

σ = Lernrate (Sigma),

z = gewünschter Output,

o = tatsächlicher Output,

δ = Fehler (Delta)

Arbeitsmaterial 2 (MUSTERLÖSUNG)

2. Anwendung der Delta-Lernregel bei einer ODER-Schaltung (Perzeptron):

ODER-Schaltung			Schwellenwert: λ=1		Lernrate: σ=0,3							
Input		Soll-Output	Alte Gewichte		gewichteter Input		Summe	Ist-Output	Fehler	Korrektur	Neue Gewichte	
x1	x2	z(j)	w1	w2	x1*w1	x2*w2	a	o(j)	δ(j)	σ*δ(j)	w1	w2
								wenn a>λ dann 1, sonst 0	z(j)-o(j)			
0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	1	0,3	0,3	0
0	1	1	0,3	0	0	0	0	0	1	0,3	0,3	0,3
1	1	1	0,3	0,3	0,3	0,3	0,6	0	1	0,3	0,6	0,6
0	0	0	0,6	0,6	0	0	0	0	0	0	0,6	0,6
1	0	1	0,6	0,6	0,6	0	0,6	0	1	0,3	0,9	0,6
0	1	1	0,9	0,3	0	0,3	0,3	0	1	0,3	0,9	0,9
1	1	1	0,9	0,9	0,9	0,9	1,8	1	0	0	0,9	0,9
0	0	0	0,9	0,9	0	0	0	0	0	0	0,9	0,9
1	0	1	0,9	0,9	0,9	0	0,9	0	1	0,3	1,2	0,9
0	1	1	1,2	0,9	0	0,9	0,9	0	1	0,3	1,2	1,2
1	1	1	1,2	1,2	1,2	1,2	2,4	1	0	0	1,2	1,2
0	0	0	1,2	1,2	0	0		0	0	0	1,2	1,2
1	0	1	1,2	1,2	0,6	0	1,2	1	0	0	1,2	1,2
0	1	1	1,2	1,2	0	0,3	1,2	1	0	0	1,2	1,2
1	1	1	1,2	1,2	0,6	0,3	2,4	1	0	0	1,2	1,2

$$w_{ij}\,(neu) = w_{ij}\,(alt) + x_j\,\sigma\,(z_i - o_i) = w_{ij} + x_j\,\sigma\,\delta_i$$

w = Gewicht,

x = Input,

σ = Lernrate (Sigma),

z = gewünschter Output,

o = tatsächlicher Output,

δ = Fehler (Delta)