Ausarbeitung zum Hauptseminar Machine Learning

Matthias Seidl

8. Januar 2004

Zusammenfassung

single-layer networks, linear separability, least-squares techniques

Inhaltsverzeichnis

1 Einführung

Als Motivation der Neuronalen Netze dient das zentrale Nervensystem bzw. Gehirn biologischer Lebewesen. Es bietet im Gegensatz zu konventionellen Rechnern, bei denen Erinnerungen bzw. Daten an bestimmten Adressen abgespeichert werden, einige Vorteile. Dazu gehören:

Assoziativität: Das Gehirn speichert Erinnerungen assoziativ ab. Hört man z.B. den Namen einer bekannten Person, so erinnert man sich gleich an deren Gesicht bzw. Aussehen. Bei Rechnern hat man den Namen unter einer Adresse abgespeichert. Ruft man den Wert der Adresse ab, so wird dann nur der Name zurückgegeben ohne Verweis auf das Gesicht der Person.

Fehlertoleranz: Im Gehirn sterben täglich Nervenzellen ab, sei es durch natürliche Alterungsprozesse, Alkoholkonsum, das Ausüben gefährlicher Sportarten wie z.B. Boxen oder Unfälle. Beim Abspeichern von Erinnerungen sind beim Gehirn eine große Anzahl von Zellen beteiligt, so dass die Ausfälle durch das Absterben von Nervenzellen kompensiert werden können und manche Erinnerungen trotzdem erhalten bleiben können. Bei Rechnern, die auf Adressen basieren, ist das anders. Geht bei einer solchen Adresse nur ein Bit verloren oder wird es falsch übermittelt, so wird gleich eine ganz falsche Information zurückgegeben.

Parallelität: Wie schon erwähnt sind am Abspeichern und Auslesen von Erinnerungen eine große Anzahl von Neuronen beteiligt. Durch diese Parallelität kann das Gehirn sehr effizient arbeiten. Bei PC's sind beim Abfragen einer Adresse nur eine sehr geringe Anzahl aller Teile des Rechners beteiligt. Diese Geräte machen in dieser Zeit gar nichts und warten nur, bis die Abfrage ausgeführt wurde. Hier werden große Rechenkapazitäten verschwendet.

Um die Vorteile natürlicher Nervensysteme auch mit Rechnern zu nutzen, wurde das Modell der künstlichen neuronalen Netze entwickelt. Dieses orientiert sich dabei sehr stark am biologischen Neuron.

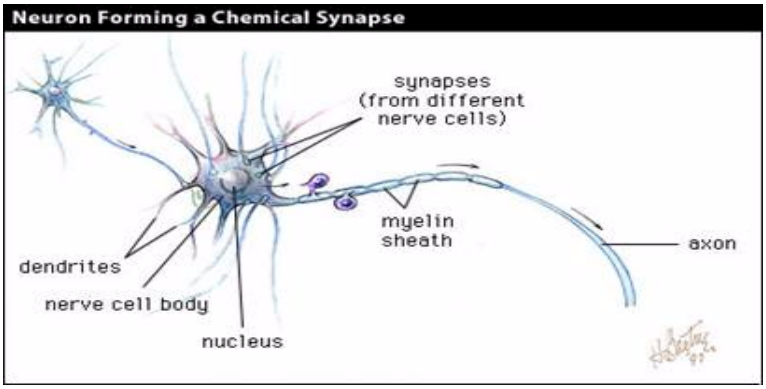

Abbildung 1: Biologisches Neuron

Wichtige Bestandteile sind die Dentriten, welche Signale anderer Neuronen empfangen können, der Zellkörper, das Soma, welches die Signale verarbeitet und „aufsummiert“ , sowie das Axon, welches die Aufgabe übernimmt, Signale an andere Neuronen weiterzuleiten.
Alle diese Merkmale wurden in das Modell eines künstlichen neuronalen Netzes übernommen.

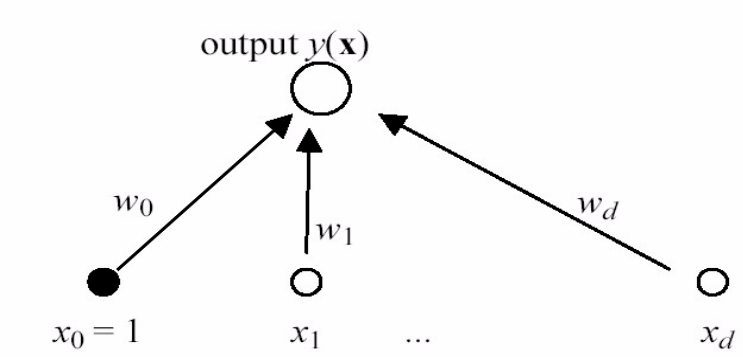

Abbildung 2: Künstliches Neuron

Es besteht aus einer Eingabeschicht mit d Eingabeknoten. Dort werden die Werte $x_1, \ldots, x_d$ eingegeben. Die Ausgabe y(x) des neuronalen Netzes berechnet sich aus der gewichteten Summe ihrer Eingaben, das heisst $y(x) = \sum_{i=1}^{d} w_i x_i + w_0$. Zu erwähnen sei noch das Gewicht w_0, welches Bias benannt wird (- w_0 wird Threshold genannt). Dessen Funktion wird später genauer erklärt.

1.1 Anwendungen neuronaler Netze

Anwendungen neuronaler Netze finden sich in allen erdenklichen Anwendungsgebieten. Folgende stellen nur einen kleinen Ausschnitt dar:

- Zeichen- und Spracherkennung
- Musik Komposition (das Ergebnis ist jedoch etwas gewöhnungsbedürftig)
- Computerspiele wie z.B. Black and White
- Aktienkursvorhersage, Überprüfen der Kreditwürdigkeit
- Maschinensteuerung(z.B. Waschmaschinensteuerung)

Das Beispiel der Waschmaschinensteuerung soll etwas genauer erläutert werden. Dazu werden eine Anzahl von Sensoren an unterschiedlichen Stellen in die Waschmaschine eingebaut. Diese messen die Verschmutzung des Wassers, indem sie die Trübung an Hand der Lichtdurchlässigkeit des Wassers bestimmen. Je verschmutzter das Wasser ist, umso weniger Photonen können das Wasser durchdringen. Die Messwerte der Sensoren werden nun aufsummiert, wobei jeder Wert mit dem Gewicht $w_i = \frac{1}{d}$ multipliziert wird. Es wird also einfach der Mittelwert aller Messwerte gebildet. Übersteigt dieser eine zuvor eingestellte Schwelle, so wird angenommen, daß das Wasser stark verschmutzt ist. Nun wird neues Wasser in die Waschemaschine eingefüllt und die Dauer des Waschvorgangs erhöht.

1.2 Netzwerktypen

Im folgenden sollten einige Typen neuronaler Netze erläutert werden:

feed-forward vs. rekurrente Netze: Bei feedforward Netzen handelt es sich um kreisfreie bzw. azyklische Netze. In Folge haben rekurrente Netze Kreise, d.h. eine Ausgabe eines Ausgabeknotens kann wieder als neue Eingabe verwendet

werden. Dabei handelt es sich eigentlich um endliche Automaten. Rekurrente Netze sind wesentlich schwieriger zu verstehen, da hier z.B. chaotisches Verhalten auftreten kann.

single-layer vs multilayer Netze: Erstere bestehen aus einer Eingabeschicht sowie einer Ausgabeschicht. Letztere können noch eine beliebige Anzahl von sogenannten „versteckten“ Knoten besitzen, die sich zwischen Eingabe- und Ausgabeschicht befinden.

supervised vs. unsupervised Lernen: Beim überwachten Lernen werden die Netze anhand eines Testdatensatzes trainiert. Beim unüberwachten Lernen muss das Netz anhand der Eingabedaten die Muster extrahieren, was natürlich viel schwieriger ist.

continous vs. binary: Man kann für die Eingabewerte reelle Zahlen zulassen, oder aber nur die Werte 0 und 1 verwenden.

2 Grundeigenschaften

Neuronale Netze werden häufig zur Klassifikation benutzt. Die Daten können sowohl in zwei, als auch in mehrere Klassen eingeteilt werden. Die Eigenschaften dieser zwei Typen werden im folgenden etwas genauer beschrieben.

2.1 Zwei Klassen

Um die Daten in zwei Klassen einzuordnen benutzt man folgende, weiter oben schon erwähnte lineare Diskriminante:

$$y(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + w_0$$

Bei $y(\mathbf{x}) = 0$ existiert eine *(d-1)*-dimensionale „decision-boundary“ im *d*-dimensionalen Raum. Der Vektor $\mathbf{x}$ gehört zur Klasse C_1, wenn $y(\mathbf{x}) > 0$. Wenn $y(\mathbf{x}) < 0$ ist, dann wird $\mathbf{x}$ der Klasse C_2 zugewiesen. Der Gewichtsvektor $\mathbf{w}$ steht senkrecht auf der „decision-boundary“ und bestimmt so deren Orientierung im Raum. Seien $\mathbf{x^A}$ und $\mathbf{x^B}$ zwei Punkte auf der „decision-boundary“, so folgt daraus, dass $y(\mathbf{x^A}) = 0 = y(\mathbf{x^B})$. Setzt man nun das ganze in die lineare Diskriminante ein, so erhält man $\mathbf{w^T}(\mathbf{x^B} - \mathbf{x^A}) = 0$. Daraus folgt, dass $\mathbf{w}$ senktrecht zu jedem Vektor auf der „decision-boundary“ sein muss, und damit orthogonal zur „decision-boundary“ selbst ist.

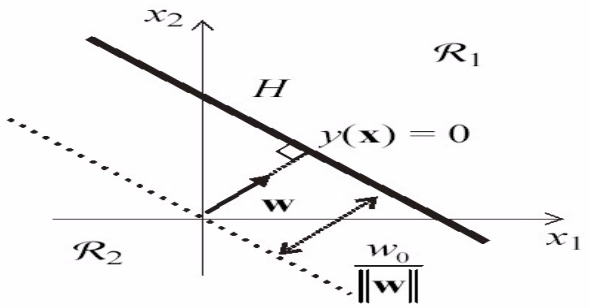

Abbildung 3: Decision-boundary im 2-dimensionalen Raum

Diese Eigenschaft kann nun benutzt werden, um die Gewichte so anzupassen, dass die „decision-boundary“ die Daten in zwei Klassen einteilt, oder anders ausgedrückt, um das Neuron zu trainieren. Das Bias w_0 bestimmt den Normalenabstand l des Ursprungs zur Hyperebene.

$$l = \frac{\mathbf{w^T x}}{|\,\mathbf{x}\,|}$$

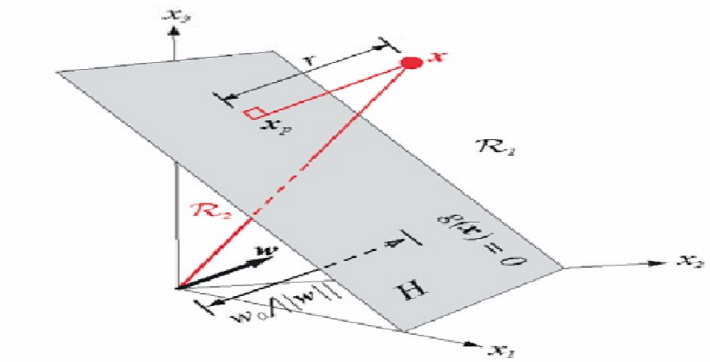

Abbildung 4: Decision-boundary im 3-dimensionalen Raum

2.2 Mehrere Klassen

Das einfache Neuron lässt sich nun leicht erweitern, so dass nun c Klassen klassifiziert werden können. Dazu fügt man einfach für jede Klasse einen neuen Outputknoten in das neuronale Netz ein. Die Ausgaben der einzelnen Outputknoten werden mittels folgender erweiteter linearen Diskriminante berechnet:

$$y_k(\mathbf{x}) = \mathbf{w_k^T x} + w_{k0}$$

Ein Vektor $\mathbf{x}$ gehört zur Klasse C_k, wenn $y_k(\mathbf{x}) > y_j(\mathbf{x})$ für alle j $\neq$ k. Die „desicion-boundary“, welche die Klasse C_k von der Klasse C_j trennt, ist gegeben durch $y_k(\mathbf{x})$ = $y_j(\mathbf{x})$. Sie gehört zu folgender Hyperebene

$$(\mathbf{w_k} - \mathbf{w_j}^T)\mathbf{x} + (w_{k0} - w_{j0}) = 0$$

Analog zum Fall mit den zwei Klassen, ist die Normale zur „decision-boundary“ durch die Differenz zweier Gewichtsvektoren gegeben und der Abstand der „decision boundary“ zum Ursprung berechnet sich nach folgender Formel:

$$l = \frac{(w_{k0} - w_{j0})}{|\,\mathbf{w_k} - \mathbf{w_j}\,|}$$

Durch die Existenz mehrerer Klassen ergeben sich sogenannte „decision-regions“, die alle einfach verknüpft und konvex sind. Dazu weist man einfach nach, dass alle Punkte, die sich auf der Strecke zwischen den Punkten $\mathbf{x^A}$ und $\mathbf{x^B}$ befinden, ebenfalls in der Region R_k sind. Daraus ergeben sich dann die zuvor genannten Eigenschaften.

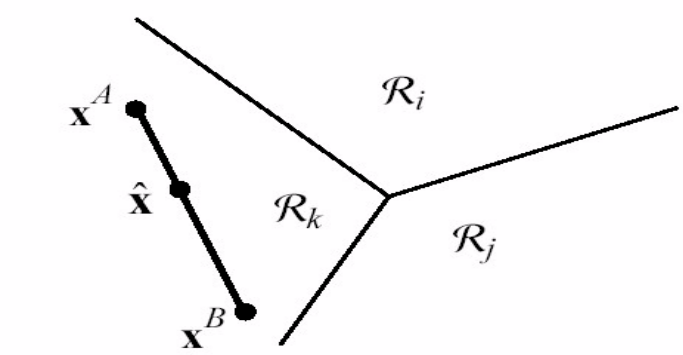

Abbildung 5: Beipiel für die „decision-regions“, die von einer linearen Diskriminante für mehrere Klassen erzeugt werden.

2.3 Aktivierungsfunktion

Um die Approximationsfähigkeit eines neuronalen Netzes zu erweitern, wird häufig eine sogenannte Aktivierungsfunktion verwendet. Diese nichtlineare, meist monotone Funktion wird auf das Ergebnis der gewichteten Summe angewendet und berechnet sich wie folgt:

$$y(\mathbf{x}) = g(\mathbf{w}^{\mathbf{T}}\mathbf{x} + w_0)$$

Folgende Aktivierungsfunktionen werden häufig verwendet.

- lineare Funktion (keine Verallgemeinerung der Diskriminante!)
- Stepfunktion oder Thresholdfunktion

$$g(a) = \begin{cases} -1 & \text{falls } a > 0 \\ 1 & \text{falls } a < 0 \end{cases}$$

- Logistischer Sigmoid

$$g(a) = \frac{1}{1 - e^{-a}}$$

  Der Name Sigmoid bedeutet, dass die Funktion eine S-förmige Gestalt hat. Die Funktion ist monoton und verändert deshalb die Linearität der Diskriminante nicht. Die Ableitung kann sehr einfach bestimmt werden. Dies ist von Vorteil für einen Lernalgorithmus, auf den später etwas genauer eingegangen wird. Wählt man a sehr klein, so kann der logistische Sigmoid durch eine lineare Funktion approximiert werden.

3 Lineare Separabilität

Definition: Wenn alle Datenpunkte einer Trainingsmenge perfekt von einer decision-boundary klassifiziert werden können, dann nennt man die Punkte linear separabel.

Beispiele für linear separable Probleme sind die booleschen Funktionen OR oder AND. Das XOR-Problem, das ebenfalls wie OR/AND aus vier Datenpunkten im 2-dimensionalen Raum besteht, ist leider nicht linear separabel. Leider ist die Menge der nicht linear separablen Punkte viel größer als die der lösbaren Probleme. Nun

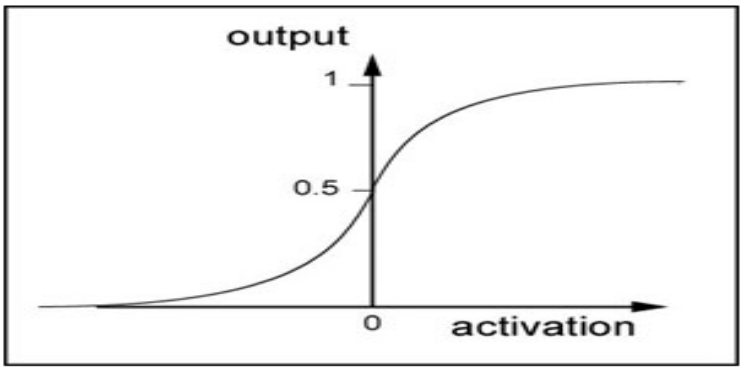

Abbildung 6: Logistischer Sigmoid

stellt sich natürlich die Frage, welche Probleme man mit einfachen einschichtigen neuronalen Netzen überhaupt lösen kann. Dies führt zu folgendem Gedankenexperiment. Man verteilt N Datenpunkte zufällig in d Dimensionen. Die Punkte müssen sich in „general“ position befinden, d.h. keine Teilmenge von d oder weniger Punkten ist linear unabhängig. Im 3-dimensionalen Fall bedeutet dies, das keine 3 Punkte kolinear und keine 4 Punkte auf einer Ebene liegen. Nun teilt man jeden der Datenpunkte zufällig in eine der beiden Klassen C_1 oder C_2 ein. Jede Zuordnung aller Punkte nennt man Dichtomie und es gibt für N Punkte 2^N verschiedene Dichtomieen. Nun kann man eine Rekursionsgleichung herleiten und zeigen, dass die Anzahl der Dichtomieen, die linear separabel sind, durch folgende Formel gegeben ist

$$T(N,d) = \begin{cases} 2^N & \text{falls } d \geq N \\ 2\sum_{k=0}^{d-1} \binom{N-1}{d} & \text{falls } d < N \end{cases}$$

Im Falle von binären Eingabevektoren gibt es bei d Dimensionen 2^d Eingabepatterns und somit 2^{2^d} viele verschiedene Dichtomieen. Leider sind nur weniger als $\frac{2^{2^d}}{d!}$ linear separabel. Um die Mächtigkeit neuronaler Netze zu erweitern, kann man jetzt entweder nichtlineare „decision-boundaries“ zulassen, was zu den „multilayer“ neuronalen Netzen führt, oder aber man transformiert die Inputpatterns erst mit in Hilfe sognannter Basisfunktionen $\Phi(\mathbf{x})$ in einen linear separablen Bereich, der dann von „single-layer networks“ gelernt werden kann.

4 Least-squares techniques

Im folgenden werden zwei Verfahren genauer erläutert, mit deren Hilfe man neuronale Netze trainieren kann. Dazu gibt man für jedes Inputpattern ein sogenanntes Zielpattern vor. Die Lernalgorithmen versuchen nun den Fehler, d.h. die Differenz zwischen Output und Zielwert zu minimieren. Dazu verwendet man häufig die sum-of-squares Fehlerfunktion:

$$E(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}\sum_{k=1}^{c}(y(\mathbf{x^n};\mathbf{w}) - t_k^n)^2$$

$y(\mathbf{x^n})$ repräsentiert die Ausgabe des k-ten Ausgabeknotens, t_k^n den Zielwert für den k-ten Ausgabeknoten, N stellt die Anzahl der Trainingspattern dar und c ist die Anzahl der Ausgabeknoten.

4.1 Lösung mittels Pseudo-Inverse

Natürlich will man die Gewichte so bestimmen, dass der Gesamtfehler minimal wird. Dazu bildet man einfach die Ableitung der Fehlerfunktion und setzt das Ergebnis gleich 0. Es gilt also

$$E(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}\sum_{k=1}^{c}(\sum_{j=0}^{M} w_{kj}\Phi_j^k - t_k^n)^2$$

Die Ableitung davon ergibt

$$\sum_{n=1}^{N}(\sum_{j'=0}^{M} w_{kj'}\Phi_{j'}^k - t_k^n)\Phi_j^n = 0$$

In Vektorschreibweise ergibt das

$$(\mathbf{\Phi^T\Phi})\mathbf{W^T} = \mathbf{\Phi^T T}$$

Als Lösung für unsere Gewichte erhält man also

$$\mathbf{W^T} = \mathbf{\Phi^\dagger T} \qquad mit \qquad \mathbf{\Phi^\dagger} = (\mathbf{\Phi^T\Phi})^{-1}\mathbf{\Phi^T}$$

$\Phi^\dagger$ heisst Pseudoinverse von Φ und deshalb nennt man das ganze Verfahren auch Pseudo-Inverse Lösung des least-squares Problems. Der Vorteil dieses Verfahrens ist, das es relativ schnell sowie die erhaltene Lösung exakt ist. Leider kann man das Verfahren nicht immer anwenden, wenn man zum Beispiel eine nichtlineare Aktivierungsfunktion verwendet, da man hierzu obige Ableitung nicht berechnen kann. Ausserdem kann es zu Problemen kommen, wenn $\Phi^T\Phi$ fast singulär ist. In diesen Fällen setzt man das numerische iterative Verfahren Gradienten-Abstieg ein.

4.2 Gradienten-Abstieg

Dazu wählt man sich einen Gewichtsstartvektor w, indem man ihn z.B. mit zufälligen Werten initialisiert. Im 3-dimensionalen Fall ergibt die Fehlerfunktion, die man ja minimieren will, einen Paraboloiden, auf dem man sich einen zufälligen Startpunkt wählt.

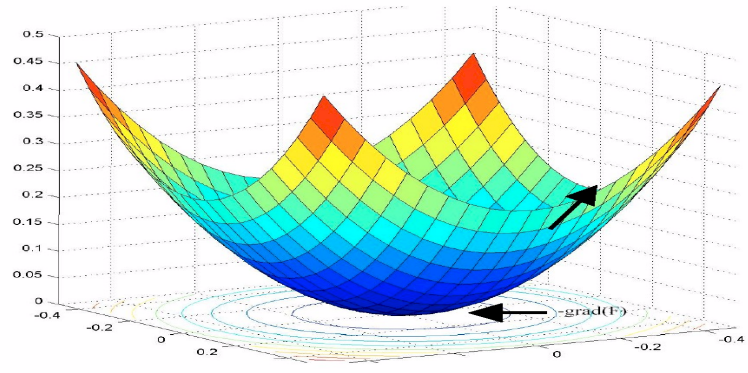

Abbildung 7: Fehlerfunktion E ergibt einen Paraboloid im 3-dimensionalen Raum

Um den Fehler zu minieren, geht man jetzt „stückchenweise“ nach unten. Nun weiß man aber, dass der Gradient in die Richtung des stärksten Anstiegs zeigt.

Deshalb bewegt man sich genau in die entgegengesetzte Richtung, also in die Richtung, in die der negative Gradient zeigt. Die Gewichte werden also folgendermaßen angepasst:

$$\mathbf{w^{t+1}} = \mathbf{w^t} + \eta \nabla \mathbf{w^T} \qquad mit \qquad \nabla \mathbf{w^T} = (\frac{\partial E(\mathbf{w})}{\partial w_1}, \frac{\partial E(\mathbf{w})}{\partial w_2}, \ldots, \frac{\partial E(\mathbf{w})}{\partial w_N})$$

η bezeichnet die Lernrate, die aus dem Intervall [0,1] gewählt wird. Wird sie zu klein gewählt, so terminiert der Algorithmus zu langsam, ist sie zu groß so kann es zu Oszillationen kommen.

Für eine lineare Diskriminante mit Basisfunktionen ergeben sich die partiellen Ableitungen zu

$$\frac{\partial E^n}{\partial w_{kj}} = [y_k \mathbf{x^n} - t_k^n]\Phi_j(\mathbf{x^n}) = \delta_k^n \Phi_j^n$$

Betrachtet man zwei benachbarte Gewichte w^t und w^{t+1} , so kann man damit die Deltaregel herleiten, die als Abbruchkriterium für den Algorithmus dienen kann.

$$\Delta w_{kj} = -\eta \delta_k^n \Phi_j^n$$

Betrachtet man eine lineare Diskriminante mit dem logistischen Sigmoid, so ergibt die partielle Ableitung der Fehlerfunktion:

$$\frac{\partial E^n}{\partial w_{kj}} = g'(a_k)\delta_k^n \Phi_j^n \qquad mit \qquad \delta_k^n = g'(a_k)(y_k(x^n) - t_k^n)$$

Hier bezeichnet g'(a) die Ableitung des logistischen Sigmoid, die sich einfach wie folgt berechnet: g'(a) = g(a)(1-g(a)).

Algorithmus: Gradienten-Abstieg

1. Initialisiere Gewichte mit Zufallswerten
2. Iteriere durch eine Anzahl von Epochen [1]. In jeder Epoche mache
   (a) Berechene die Ausgabe des Netzwerks
   (b) Berechne die Differenz zwischen Output und Zielwert(delta)
   (c) Passe mit Hilfe der Gradientabstiegsregel $w_{kj}^{t+1} = w_{kj}^t - \eta \delta_k^n \Phi_j^n$ die Gewichte an
   (d) Brich die Schleife ab, wenn der Ergebnis der Delta-regel einen gewissen Fehlerwert unterschreitet

5 Das Perzeptron

Das Perzeptron ist ein neuronales Netz mit einer Threshold-Aktivierungsfunktion. Ausserdem werden die Inputs zuerst mit Hilfe von Basisfunktionen Φ_j vorverarbeitet. Die Outputs eines Perzeptrons berechnen sich also wie folgt:

$$y = g(\sum_{j=0}^{M} w_j \Phi_j(\mathbf{x})) = g((\mathbf{w})^T \Phi)$$

[1] Eine Epoche ist ein Durchlauf durch die gesamte Trainingsmenge

Leider kann man hier nicht einfach den Gradientenabstieg verwenden, da die Thresholdfunktion nicht vernünftig abgeleitet werden kann. Deshalb führt man eine andere Fehlerfunktion ein, die leicht ableitbar ist. Dies ist das Perzeptronkriterium

$$E^{perc}(\mathbf{w}) = -\sum_{\Phi^n \in M} (\mathbf{w})^T (\Phi^n t^n)$$

Nun lässt sich der Gradientenabstieg wieder verwenden und nach Ableitung des Perzeptronkriteriums erhält folgende einfache Regel für die Gewichte:

$$w_j^{r+1} = w_j^r + \eta \Phi_j^n t^n$$

Daraus ergibt sich ein einfacher Lernalgorithmus.

1. Überprüfe jedes Pattern in der Trainingsmenge mit Hilfe der aktuellen Gewichte
2. Wurde das Pattern richtig klassifiziert, so fahre zu nächsten Pattern fort.
3. Falls nicht, addiere den Patternvektor multipliziert mit η zum Gewichtsvektor, wenn das Pattern eigentlich zur Klasse C_1 gehört, ansonsten subtrahiere das Eingabepattern vom Gewichtsvektor.
4. Fahre fort, bis alle Patterns richtig klassifiziert sind.

Nach dem Perzeptronkonvergenz-Theorem terminiert der Algorithmus für linear separable Probleme nach einer endlichen Anzahl von Schritten.

6 Vor- und Nachteile

Vorteile:

- es gibt einfache Lernalgorithmen
- man kann Funktionen relativ schnell lernen
- Unempfindlich gegenüber Rauschen
- Daten können in mehrere Klassen eingeteilt werden

Nachteile:

- nur linear separable Funktionen können gelernt werden
- Black-Box Sicht; das Modell kann schlecht validiert werden

Literatur

[1] Christopher M. Bishop: *Neural Networks for Pattern Recognition* Chapter 3.1.-3.5. , Clarendon Press - Oxford, (1995)

[2] Stuart Russell, Peter Norvig *Artificial Intelligence A modern approach* Chapter 20.5, Prentice Hall, (2003)

[3] David J.C. MacKay *Information Theory, Inference, and Learning Algorithms* Chapter 38-41, Cambridge University Press

[4] ftp://ftp.sas.com/pub/neural/FAQ.html

[5] http://home.cc.umanitoba.ca/ umcorbe9/neuron.html

[6] http://www.ai-junkie.com/nnt1.html

[7] http://neuralnetworks.ai-depot.com/