Seminar *Statistische Lerntheorie und ihre Anwendungen*

Ausarbeitung zum Vortrag vom 15. Mai 2007

Neuronale Netze

SS 2007

Universität Ulm

Anna Wallner

Betreuer: Malte Spiess

Inhaltsverzeichnis

1 Motivation

Ein Computer ist dem menschlichen Gehirn in vielerlei Hinsicht überlegen. So rechnet er im Allgemeinen viel schneller und präziser als ein Mensch. Gleichzeitig kann er Daten äußerst zuverlässig speichern und anschließend zu jedem beliebigen Zeitpunkt wieder abrufen. Die Erinnerung eines Menschen dagegen ist bruchstückhaft, Informationen werden meist selbst nach wiederholtem Lernen unvollständig abgespeichert. Auch beim Abrufen der gespeicherten Informationen kann es zu Problemen kommen, da wir Gelerntes oder Erlebtes oft (z.T. auch nur zweitweise) wieder vergessen. Allerdings arbeiten Computersysteme vor allem sequenziell, d.h. Informationen werden nacheinander bearbeitet. Bei komplexen Aufgaben führt dies oftmals zu langen Rechenzeiten.

Die Informationsverarbeitung im Gehirn dagegen läuft in hohem Maße parallel ab. Ein Mensch besitzt ungefähr 100 Milliarden Nervenzellen (Neuronen), die untereinander stark vernetzt sind und über elektrische Impulse interagieren. Dabei „feuert“ ein Neuron, d.h. es leitet einen Reiz weiter, sobald die Summe der empfangenen Reize, die es von anderen Neuronen empfangen hat, einen bestimmten Schwellenwert übersteigt. Neuronen können gleichzeitig untereinander Informationen austauschen, weshalb das Gehirn komplexe Aufgaben wie das Erkennen von Mustern oder das Verallgemeinern von Beispielen gut und schnell lösen kann. Selbst verrauschte oder unvollständige Daten kann ein Mensch im Gegensatz zu Rechenmaschinen bis zu einem gewissen Grad mühelos wieder rekonstruieren.

Um es einem Computersystem zu ermöglichen, komplexe Aufgaben anhand von Trainingsbeispielen zu erlernen, sowie eine hohe Parallelität bei der Informationsverarbeitung und eine höhere Fehlertoleranz zu erreichen, wurden künstliche neuronale Netze (KNN) entwickelt. Die künstlichen Neuronen eines KNNs aktivieren sich untereinander - ähnlich ihrem biologischen Vorbild - mit Hilfe von gerichteten Verbindungen.

Künstliche neuronale Netze eignen sich dazu, beliebig komplexe Funktionen zu approximieren, Aufgaben zu erlernen (z.B. Klassifikation von Daten) und Probleme zu lösen, bei denen eine explizite Modellierung schwierig oder nicht durchführbar ist. Damit finden KNN Anwendung bei Frühwarnsystemen, in der Optimierung, bei Zeitreihenanalysen, sowie in der Bildverarbeitung und Mustererkennung (beispielsweise Schrifterkennung, Spracherkennung oder Data-Mining).

2 Grundlagen

2.1 Künstliche Neuronen

Ein künstliches Neuron ist im Grunde nichts anderes als eine parametrisierte, beschränkte Funktion $f(X_1, X_2, ..., X_p, \omega_1, \omega_2, ..., \omega_p)$, die von Eingaben X_i und Gewichten $\omega_i,\ i = 1, ..., p$, abhängt. Diese Funktion, die auch Aktivierungsfunktion genannt wird,

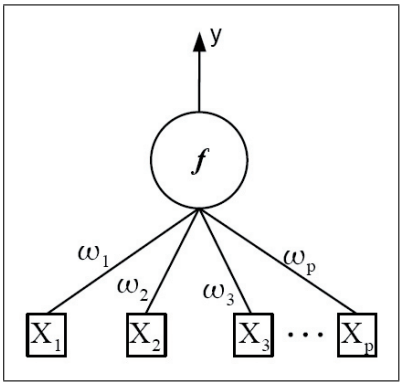

Abbildung 1: Künstliches Neuron

stellt die Ausgabe y des Neurons dar.

Die Eingabe besteht entweder aus der Ausgabe anderer Neuronen oder aus den Werten des beobachteten Prozesses.

Die Gewichte regulieren den Einfluss, den die Eingabewerte auf die Aktivierungsfunktion f haben. Eine erregende Wirkung wird durch positive Gewichte erreicht; Dementsprechend haben negative Gewichte eine hemmende Wirkung. Ist das Gewicht gleich 0, so hat der zugehörige Eingabewert keinen Einfluss auf die Ausgabe des Neurons, d.h. es besteht keine Verbindung zwischen Eingabewert und Neuron.

2.2 Parametrisierung der Aktivierungsfunktion

Es gibt viele Möglichkeiten, die Funktion f zu parametrisieren. Zwei der gängigsten Arten sind die folgenden:

1.) Die Parameter werden den Eingaben direkt zugeordnet:
In diesem Fall wird häufig ein Potenzial verwendet, welches aus der gewichteten Summe der Eingabewerte X_1,..., X_p und einem zusätzlichen Bias-Parameter ω_0 besteht. Man verknüpft das Neuron zu diesem Zweck mit einem Bias-Neuron X_0, dessen Ausgabe konstant eins ist.

$$\Rightarrow v \;=\; \sum_{i=0}^{p} \omega_i X_i \;=\; \omega_0 + \sum_{i=1}^{p} \omega_i X_i$$

Als Aktivierungsfunktion $f(v)$ wird eine s-förmige (sigmoide) Funktion, wie z.B. der Tangens Hyperbolicus oder eine logistische Funktion gewählt, so dass die Ausgabe einer nichtlinearen Kombination der Eingabewerte entspricht. Eine andere Möglichkeit stellt die Stufenfunktion $f(v) = \mathbb{1}_{\{v \geq \theta\}}$ dar, wobei θ den Schwellenwert repräsentiert, bei dessen Überschreitung das Neuron „feuert“.
Die Stufenfunktion hat jedoch den Nachteil, dass sie an der Stelle θ nicht differenzierbar ist, und sich deswegen nicht für das Lernverfahren eignet, das in Abschnitt 3 vorgestellt wird (Backpropagation).

2.) Die Parameter gehören zur Definition der Aktivierungsfunktion:
Ein Beispiel dafür ist die Gauß'sche radiale Basisfunktion:

$$f(X_1, X_2, ..., X_p, \omega_1, \omega_2, ..., \omega_p, \omega_{p+1}) = \exp\left(-\sum_{i=1}^{p} \frac{(X_i - \omega_i)^2}{2\omega_{p+1}^2}\right),$$

wobei der Vektor $(\omega_1, ..., \omega_p)$ die Position des Mittelpunktes der Gaußglocke und ω_{p+1} die zugehörige Standardabweichung darstellt.

2.3 Feedforward-Netze

Unter einem künstlichen neuronalen Netz versteht man die Verknüpfung der nichtlinearen Ausgabefunktionen von Neuronen. Im Folgenden werden wir Feedforward-Netze betrachten, deren Namen daher stammt, dass sie zyklenfrei sind und die Informationen daher stets nur in eine Richtung fließen. Das Gegenstück dazu bilden rekurrente Netze, auf die in Abschnitt 6 kurz eingegangen wird.

Man verwendet in der Regel KNN mit mindestens einer verdeckten Schicht, welche nicht direkt beobachtet wird. Abbildung 2 zeigt ein Netz mit einer verdeckten Schicht, deren Neuronen vollständig verknüpft sind. X_0 und Z_0 sind Bias-Neuronen, deren Ausgabe konstant eins ist. Somit stellen sie, wie auch die Eingabeneuronen X_1,..., X_p, keine Neuronen im eigentlichen Sinne dar, da ihr Ausgabewert nicht durch eine Aktivierungsfunktion bestimmt wird.

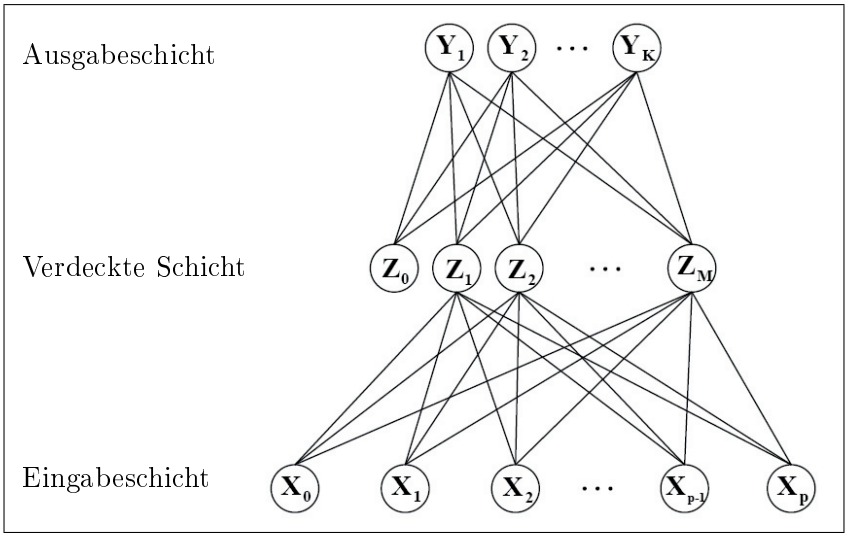

Abbildung 2: Netzwerkdiagramm eines Feedforward-Netzes mit einer verdeckten Schicht, deren Neuronen vollständig verknüpft sind

2.4 Beispiel: XOR-Verknüpfung

Um zu sehen, wie Informationen in einem neuronalen Netz verarbeitet werden, betrachten wir zunächst das einfachste Netz, mit dem eine exklusiv-ODER-Verknüpfung (XOR) dargestellt werden kann. Diese Verknüpfung lässt sich im Gegensatz zum AND- und OR-Operator nicht duch ein KNN ohne verdeckte Schicht ausdrücken. Die XOR-Funktion beschreibt die logische Verknüpfung zweier Operanden, welche die Werte 0 ($\widehat{=}$ false) und 1 ($\widehat{=}$ true) annehmen können. Das Ergebnis lautet genau dann „true“, wenn genau einer der Operanden den Wert 1 besitzt.

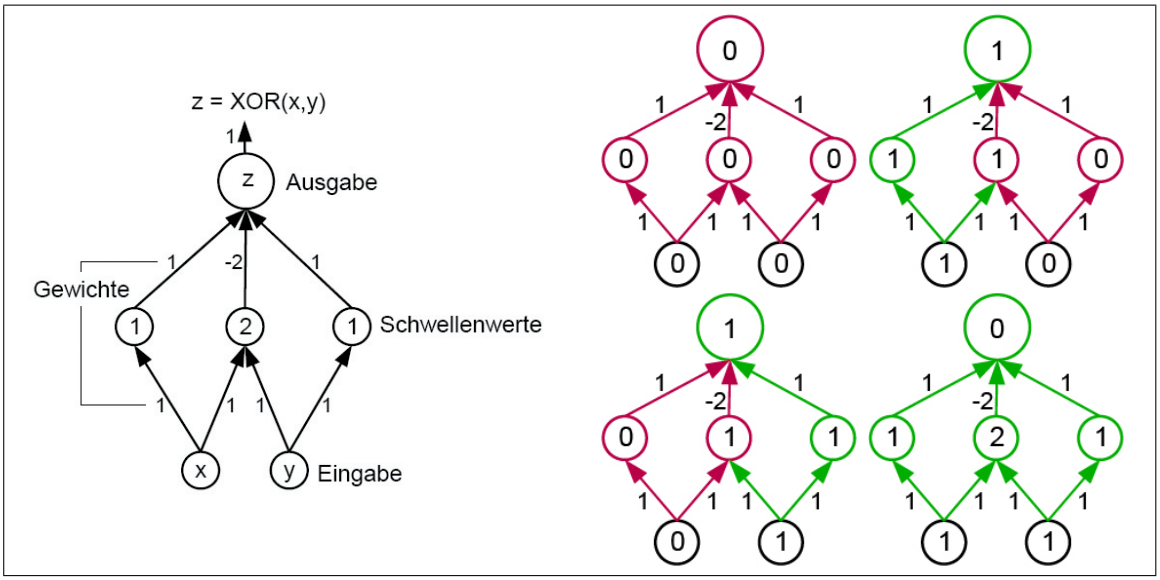

Abbildung 3: Aufbau eines XOR-Netzes und die vier möglichen Kombinationen der Eingabewerte

Die Eingabewerte x und y entsprechen jeweils einem Operaranden. Jede ihrer Verbindungen zu den Neuronen in der verdeckten Schicht hat das Gewicht 1. Die verdeckten Neuronen besitzen jeweils einen Schwellenwert von 1 oder 2. Somit sind ihre Aktivierungsfunktionen durch $f_1(x) = \mathbb{1}_{\{1 \cdot x \geq 1\}}$, $f_2(x,y) = \mathbb{1}_{\{1 \cdot x + 1 \cdot y \geq 2\}}$ und $f_3(y) = \mathbb{1}_{\{1 \cdot y \geq 1\}}$ gegeben. Die gewichtete Summe der Ausgabewerte der verdeckten Neuronen bildet die Ausgabe des gesamten Netzes, die durch folgende Formel beschrieben werden kann:

$$\begin{aligned} XOR(x,y) &= 1 \cdot f_1(x) - 2 \cdot f_2(x,y) + 1 \cdot f_3(y) \\ &= \mathbb{1}_{\{x \geq 1\}} - 2 \cdot \mathbb{1}_{\{x+y \geq 2\}} + \mathbb{1}_{\{y \geq 1\}} \end{aligned}$$

Damit können die folgenden vier Fälle unterschieden werden:

1. $x = y = 0$:
   $XOR(x,y) = 1 \cdot 0 - 2 \cdot 0 + 1 \cdot 0 = 0$
2. $x = 1, y = 0$:
   $XOR(x,y) = 1 \cdot 1 - 2 \cdot 0 + 1 \cdot 0 = 1$
3. $x = 0, y = 1$:
   $XOR(x,y) = 1 \cdot 0 - 2 \cdot 0 + 1 \cdot 1 = 1$
4. $x = y = 1$:
   $XOR(x,y) = 1 \cdot 1 - 2 \cdot 1 + 1 \cdot 1 = 0$

2.5 Netze mit einer verdeckten Schicht

Verallgemeinern wir obiges Beispiel, indem wir ein KNN mit einer verdeckten Schicht wie in Abbildung 2 betrachten.
Wir haben also ein neuronales Netz vorliegen, welches aus 3 Schichten besteht:

- einer Eingabeschicht mit den Eingabewerten $X = (X_1,...,X_p)$ und zusätlichem Bias-Parameter $X_0 := 1$
- einer verdeckten Schicht, bestehend aus den Neuronen $Z = (Z_1,...,Z_M)$ und zusätzlichem Bias-Parameter $Z_0 := 1$
- einer Ausgabeschicht, welche die Werte $T_1,...,T_K$ berechnet

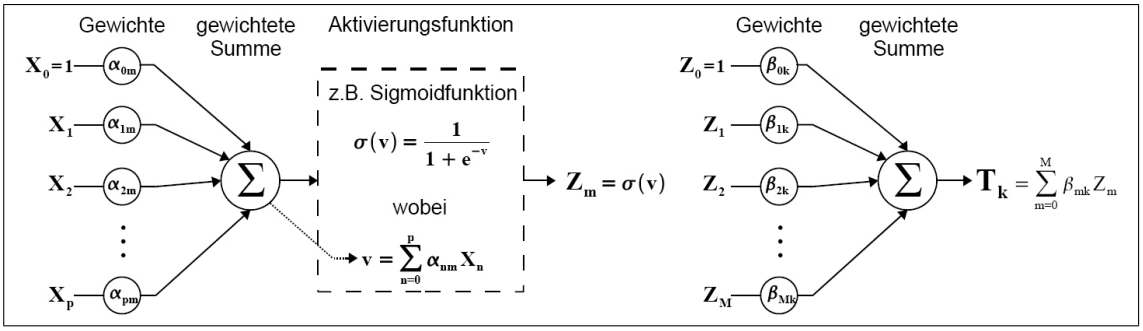

Abbildung 4: Berechnung der Merkmale Z_m und der gewichteten Summen T_k

Gewichtet wird die Verknüpfung zwischen X_n und Z_m durch den Parameter $\alpha_{nm},\ n = 0,...,p,\ m = 1,...,M$, und zwischen Z_m und T_k durch den Parameter $\beta_{mk},\ m = 0,...,M,\ k = 1,...,K$.

Wie berechnet dieses neuronales Netz seine Ausgabewerte?
Zunächst werden aus einer Linearkombination der Eingabewerte die Merkmale Z_m berechnet:

$$Z_m = \sigma\left(\sum_{n=0}^{p} \alpha_{nm} X_n\right) = \sigma\left(\alpha_{0m} + \alpha_m^T X\right),\ m = 1,...,M,$$

wobei $\sigma(v)$ die Aktivierungsfunktion der verdeckten Neuronen darstellt. Wir wählen hierfür die sigmoide logistische Funktion

$$\sigma(v) = \frac{1}{1+e^{-v}}.$$

$$\Rightarrow Z_m = \frac{1}{1+\exp(-\alpha_{0m} - \alpha_m^T X)},\ m = 1,...,M$$

Damit berechnet man die gewichteten Summen

$$T_k = \sum_{m=0}^{M} \beta_{mk} Z_m = \beta_{0k} + \beta_k^T Z,\ k = 1,...,K.$$

Durch die Ausgabefunktionen $g_k(T)$ lässt sich der Ausgabevektor $T = (T_1,...,T_K)$ anschließend zu einer endgültigen Ausgabe $\hat{f}_k$ des Netzes transformieren:

$$\hat{f}_k(X) = g_k(T),\ T = (T_1, ..., T_K),\ k = 1, ..., K,$$

wobei bei Regressionen meist die Identität

$$g_k(T) = T_k$$

und bei Klassifikationen oft die Softmax-Funktion

$$g_k(T) = \frac{e^{T_k}}{\sum_{l=1}^{K} e^{T_l}}$$

verwendet wird. Da bei Anwendung der Softmax-Funktion die Summe über alle $g_k(T)$ den Wert 1 ergibt, kann man bei Klassifikationen die Ausgabewerte $g_k(T)$ als Wahrscheinlichkeiten betrachten, mit der die eingegebenen Daten zu Klasse k gehören. Die zu klassifizierenden Daten werden dann derjenigen Klasse zugeordnet, für die diese Wahrscheinlichkeit am größten ist.

3 Anpassung des Netzes mit Backpropagation

Das „Wissen“ eines künstlichen neuronalen Netzes steckt in seinen Gewichten. Dieses Wissen muss - wie beim Menschen auch - zuvor erlernt werden. Daher werden neuronale Netze zuerst z.B. mit Hilfe des Backpropagation-Algorithmus trainiert.

Sei θ die Menge aller Gewichte eines KNNs:

$$\theta = \left\{(\alpha_{0m}, \alpha_m, \beta_{0k}, \beta_k);\ \alpha_{0m}, \beta_{0k} \in \mathbb{R},\ \alpha_m \in \mathbb{R}^p,\ \beta_k \in \mathbb{R}^M,\ m = 1, .., M,\ k = 1, .., K\right\}$$

Und sei $(X, Y),\ X = (X_1, ..., X_p),\ Y = (y_1, ..., y_K)$, ein Trainingsdatensatz, wobei X die Eingabewerte und Y die gewünschten Ausgabewerte repräsentiert.

Um die Anpassung der Gewichte an das Modell bewerten zu können, betrachten wir die Differenz zwischen der gewünschten Ausgabe y_k und der tatsächlichen Ausgabe $\hat{f}_k(X)$ und bilden daraus die Summe der Fehlerquadrate:

$$R(\theta) = \sum_{k=1}^{K} (y_k - \hat{f}_k(X))^2.$$

Wir wollen nun den Fehler $R(\theta)$ mit Hilfe des Gradientenabstiegs minimieren, um eine gute Anpassung an die Trainingsdaten zu erhalten. Dafür verwenden wir den Backpropagation-Algorithmus, der aus den folgenden Schritten besteht:

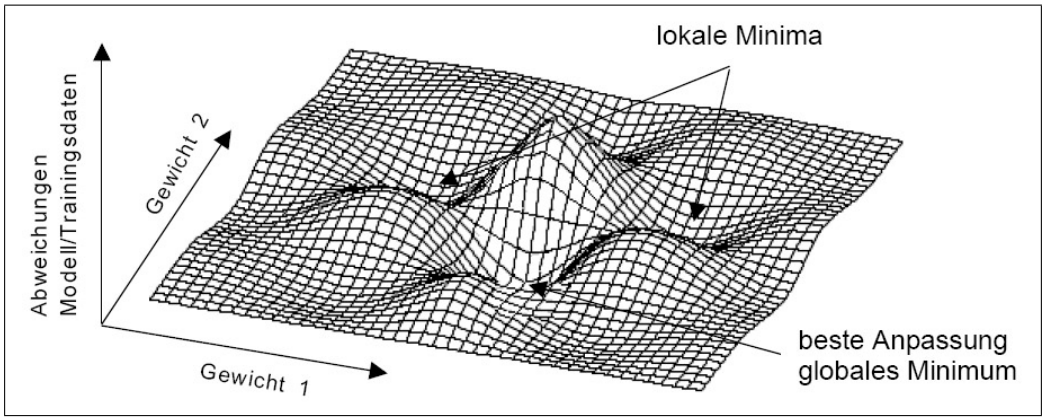

Abbildung 5: $R(\theta)$ lässt sich als Fehleroberfläche darstellen.

1. Schritt: "forward pass"
   Zunächst werden Startwerte für alle Gewichte festgelegt. Dass dies nicht trivial ist, werden wir später sehen. Desweiteren wird ein Eingabemuster angelegt, welches anschließend vorwärts durch das Netz propagiert wird. Daraus erhält man die Ausgabe des Netzes $\hat{f}_k(X),\ k = 1, ..., K$.

2. Schritt:
   Die Ausgabe wird mit den gewünschten Werten verglichen und die Differenz als Fehler des Netzes erachtet. Daraus erhält man $R(\theta)$, die Summe der Fehlerquadrate.

3. Schritt: „backward pass“
   Dieser Schritt gab diesem Algorithmus seinen Namen, da hierbei der Fehler über die Ausgabeschicht zurück zur Eingabeschicht propagiert wird. Dabei werden die Gewichtungen der Verbindungen zwischen den Neuronen, abhängig von ihrem Einfluss auf den Fehler, geändert.

Betrachten wir dieses Verfahren nun genauer für ein KNN mit einer verdeckten Schicht. In Abschnitt 2.5 haben wir gesehen, dass für die Merkmale $Z_m = \sigma(\alpha_0 m + \alpha_m^T X)$ gilt. Dann ist die tatsächliche Ausgabe gegeben durch

$$\begin{aligned}\hat{f}_k(X) &= g_k(T_k) \\ &= g_k\left(\beta_{0k} + \beta_k^T Z\right) \\ &= g_k(\beta_{0k} + \sum_{m=1}^{M} \beta_{mk}\sigma(\alpha_{0m} + \alpha_m^T X)).\end{aligned}$$

Damit kann man den Fehler des neuronalen Netzes auch folgendermaßen formulieren:

$$\begin{aligned} R(\theta) &= \sum_{k=1}^{K} \left(y_k - \hat{f}_k(X)\right)^2 \\ &= \sum_{k=1}^{K} \left(y_k - g_k(\beta_{0k} + \sum_{m=1}^{M} \beta_{mk}\sigma(\alpha_{0m} + \alpha_m^T X))\right)^2. \end{aligned}$$

Die partiellen Ableitungen von $R(\theta)$ berechnen sich damit zu

$$\frac{\partial R(\theta)}{\partial \beta_{mk}} = \underbrace{-2 \cdot (y_k - \hat{f}_k(X)) \cdot \frac{\partial}{\partial \beta_{mk}} g_k(\beta_{0k} + \beta_k^T Z)}_{=: \delta_k} \cdot Z_m$$

$$\frac{\partial R(\theta)}{\partial \alpha_{nm}} = \underbrace{\sum_{k=1}^{K} \beta_{km} \cdot \delta_k \cdot \frac{\partial}{\partial \alpha_{nm}} \sigma(\alpha_{0m} + \alpha_m^T X)}_{=: s_m} \cdot X_n$$

Mit dem Gradientenverfahren kann man nun die Anpassung der Gewichte bestimmen:

$$\Delta\beta_{mk} \;=\; -\gamma \cdot \frac{\partial R(\theta)}{\partial \beta_{mk}} \;=\; -\gamma \cdot \delta_k \cdot Z_m, \quad m = 0, 1, ..., M, \;\; k = 1, ..., K$$

$$\Delta\alpha_{nm} \;=\; -\gamma \cdot \frac{\partial R(\theta)}{\partial \alpha_{nm}} \;=\; -\gamma \cdot s_m \cdot X_n, \quad n = 0, 1, ..., p, \;\; m = 1, ..., M,$$

wobei γ die Lernrate bezeichnet. Diese bestimmt, wie stark die Gewichte verändert werden sollen. Wählt man einen hohen Wert für γ, führt dies zu einer großen Änderung der Gewichte, wodurch das Minimum von $R(\theta)$ leicht verfehlt werden kann. Bei einem sehr niedrigen Wert wird dagegen das Einlernen verlangsamt. Ein vernünftiges Intervall für die Lernrate ist durch [0,01; 0,5] gegeben, wobei oft $\gamma = 0,1$ verwendet wird.

Zum Schluss erfolgt die Aktualisierung der Gewichte:

$$\begin{aligned} \beta_{mk}^{neu} &= \beta_{mk}^{alt} + \Delta\beta_{mk} \\ \alpha_{nm}^{neu} &= \alpha_{nm}^{alt} + \Delta\alpha_{nm} \end{aligned}$$

4 Probleme

4.1 Wahl der Startwerte für die Gewichte

Die Gewichte, die vor dem Lernprozess festgelegt werden, sind entscheidend dafür, ob das Trainieren eines neuronalen Netzes zum Erfolg führt.

Setzt man alle Gewichte gleich Null, erhält man für die partiellen Ableitungen von $R(\theta)$ nach β_{mk} bzw. α_{nm} wiederum Null. Das führt dazu, dass die Gewichte durch den Backpropagation-Algorithmus nicht verändert werden können und das KNN somit nichts lernt.

Im Gegensatz dazu produzieren hohe Startwerte oftmals schlechte Ergebnisse, da die Fehleroberfläche in diesem Fall stark zerklüftet ist. Das hat zur Folge, dass die Anwendung der Methode des Gradientenabstiegs leicht zu suboptimalen Ergebnissen führen kann.

Empfehlenswert ist es daher, Zufallswerte nahe 0 zu wählen. Somit hat man zu Beginn ein fast lineares neuronales Netz gegeben, da $\sigma(v) = (1 + \exp(v))^{-1} \approx 0,5$ für $v \approx 0$. Anschließend wird durch den Backpropagation-Algorithmus an den Stellen, wo es nötig ist, Nichtlinearität eingeführt, indem die Gewichte an den betreffenden Verbindungen vergrößert werden.

4.2 Anzahl der verdeckten Neuronen und Schichten

Je mehr verdeckte Neuronen ein neuronales Netz besitzt, desto flexibler ist es. Zu viel Flexibilität hat jedoch zur Folge, dass das Modell zu Überanpassung neigt (siehe Abschnitt 4.3). Ein Beispiel hierzu zeigt Abbildung 6. Zwei KNN mit jeweils vier bzw. acht verdeckten Neuronen werden mit denselben Daten trainiert (dargestellt durch rote Markierungen). Der zweite Graph approximiert die Trainingsdaten offensichtlich besser als der erste. Wenn man als Testdaten jedoch Punkte wählen würde, die zwischen den Trainingspunkten liegen, so würde das Netz mit den vier verdeckten Neuronen sehr wahrscheinlich besser abschneiden.

Wer zu wenige verdeckte Neuronen wählt, riskiert jedoch, dass das KNN aufgrund der geringen Anzahl an Parametern nicht komplex genug für die Problemstellung ist und daher nicht richtig trainiert werden kann. Man wählt meist zwischen 5 und 100 verdeckte Neuronen, wobei die Anzahl mit der Anzahl der Eingabewerte und der Trainingseinheiten steigt. Allgemein gilt es, ein möglichst sparsames KNN zu finden, das dennoch genug Raum für Komplexität lässt.

Auch die ideale Anzahl an verdeckten Schichten ist nicht leicht zu bestimmen. Einen Ansatz für eine vernünftige Wahl bieten Experimente, wobei auch das Hintergrundwissen berücksichtigt werden sollte.

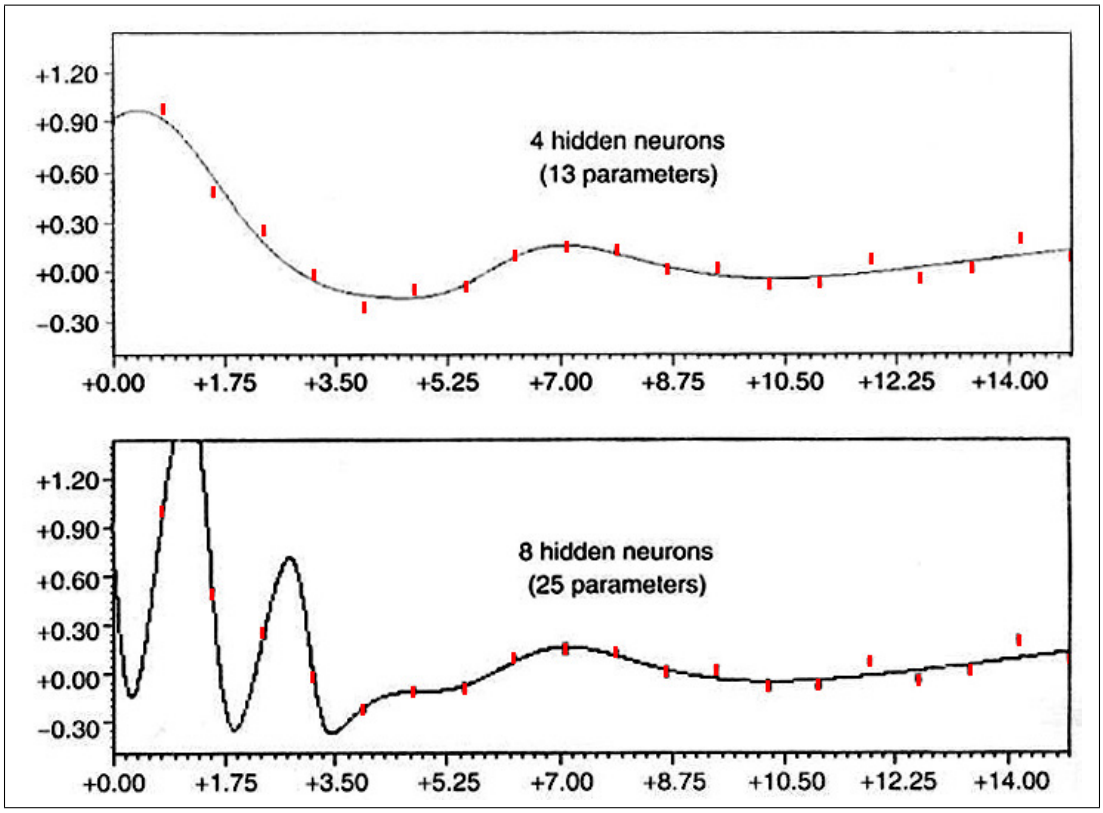

Abbildung 6: Regression mit 4 und mit 8 verdeckten Neuronen

4.3 Überanpassung

Unser Ziel ist es, die Fehlerfunktion $R(\theta)$ zu minimieren. Allerdings ist das Modell im globalen Minimum oft überangepasst. Das hat zur Folge, dass das Modell zwar hervorragend an die Trainingsdaten angepasst ist, bei neuen Daten jedoch schlechte Ergebnisse liefert.

Eine Möglichkeit, diese Situation zu vermeiden, ist der rechtzeitige Abbruch des Verfahrens. Eine bessere Alternative ist durch die Weight-Decay-Methode gegeben, deren Ziel es ist, große Gewichte zu eliminieren und dadurch die Zerklüftung der Fehleroberfläche zu minimieren. Dadurch können Fehler, die bei der Initialisierung der Startgewichte gemacht wurden, wieder ausgeglichen werden. Um dies zu erreichen, wird ein Strafterm zur Fehlerfunktion $R(\theta)$ hinzuaddiert:

$$R(\theta) + \lambda \cdot J(\theta), \text{ mit } J(\theta) = \frac{1}{2}\left(\sum_{m,k} \beta_{mk}^2 + \sum_{n,m} \alpha_{nm}^2\right) \text{ und } \lambda \geq 0.$$

λ hat die Funktion eines Tuningparameters und bestimmt, wie stark die Gewichte schrumpfen sollen. Durch den Strafterm verändert sich die Anpassung der Gewichte nun

folgendermaßen:

$$\Delta\beta_{mk} = -\gamma \cdot \frac{\partial R(\theta) + \lambda J(\theta)}{\partial \beta_{mk}} = -\gamma \cdot \left(\frac{\partial R(\theta)}{\partial \beta_{mk}} + \lambda \beta_{mk} \right)$$

$$\Delta\alpha_{nm} = -\gamma \cdot \frac{\partial R(\theta) + \lambda J(\theta)}{\partial \alpha_{nm}} = -\gamma \cdot \left(\frac{\partial R(\theta)}{\partial \alpha_{nm}} + \lambda \alpha_{nm} \right)$$

Man sieht, dass bei diesem Verfahren große Gewichte stärker bestraft werden als kleine. Außerdem ist zu erkennen, dass zu große Werte für λ ein starkes Schrumpfen der Gewichte gegen 0 bewirken. Man wählt für diesen Parameter daher meist einen Wert aus dem Intervall [0,005; 0,03].
Mit der Weight-Decay-Methode werden Gewichte, die durch das Training nicht verstärkt werden und die somit für das Problem nicht von Bedeutung sind, kontinuierlich gegen 0 verändert. Wenn ein Parameter einen bestimmten Wert unterschreitet, kann er bei Bedarf auch ganz entfernt werden, indem man ihn auf 0 setzt.

4.4 Skalierung der Eingabewerte

Da die Skalierung der Eingabewerte die effektive Skalierung der Gewichte in der untersten Schicht bestimmt, kann sie einen großen Einfluss auf die Qualität des Endergebnisses haben. Aus diesem Grund werden die Eingabedaten derart genormt, dass der empirische Mittelwert eines jeden Trainingsdatensatzes z.B. 0 und die empirische Standardabweichung 1 beträgt. Damit erreicht man eine Gleichbehandlung der Eingabewerte im Regulierungsprozess. Dies wiederum ermöglicht eine sinnvolle Wahl eines Intervalls (in diesem Fall [-0.7; 0,7]), aus dem die Startwerte der Gewichte zufällig ausgewählt werden.

4.5 Multiple Minima

Das Endergebnis hängt maßgeblich von der Wahl der Anfangswerte der Gewichte ab, da die Fehlerfunktion $R(\theta)$ nicht konvex ist und somit viele lokale Minima besitzt. Daher muss zu Beginn experimentiert werden um eine gute Wahl für die Startwerte der Gewichte zu erhalten. Anschließend wählt man dasjenige neuronale Netz, das den kleinsten Fehler verspricht. Alternativ kann als endgültige Vorhersage der Durchschnitt der Vorhersagen aller Netze verwendet werden. Eine weitere Alternative stellt das „Bagging“ dar. Hierbei verfährt man genauso wie bei der ersten Alternative. Der Unterschied besteht jedoch darin, dass die Netze mit zufällig gestörten Trainingsdaten trainiert werden.

5 Beispiel: Klassifikation handgeschriebener Zahlen

Wie bereits erwähnt, ist ein Anwendungsgebiet neuronaler Netze das Erkennen von Mustern. KNN können beispielsweise zur Erkennung von Postleitzahlen auf Briefumschlägen verwendet werden. Im folgenden Beispiel werden mögliche Vorgehensweisen bei der Klassifikation handgeschriebener Zahlen betrachtet. Der Aufbau und die Ergebnisse dieses Beispiels sind aus [1] entnommen.

Zu diesem Zweck werden zunächst geeignete Eingabemuster erzeugt. Dies geschieht durch Einscannen von handgeschriebenen Ziffern mit einer Auflösung von 16x16 Pixel. Mit 320 Ziffern sollen fünf verschiedene neuronale Netze trainiert werden. Anschließend wird jedes davon mit 160 anderen Ziffern getestet.

Jedes der fünf Netze erhält pro Ziffer 256 Eingabewerte, welche die Graustufen der einzelnen Pixel beschreiben. Die Ausgabeschicht eines jeden Netzes besteht aus 10 Neuronen, die die Ziffern 0-9 repräsentieren. $\hat{f}_k(X)$ stellt dabei die Wahrscheinlichkeit dar, dass Bild X zu Klasse k gehört.

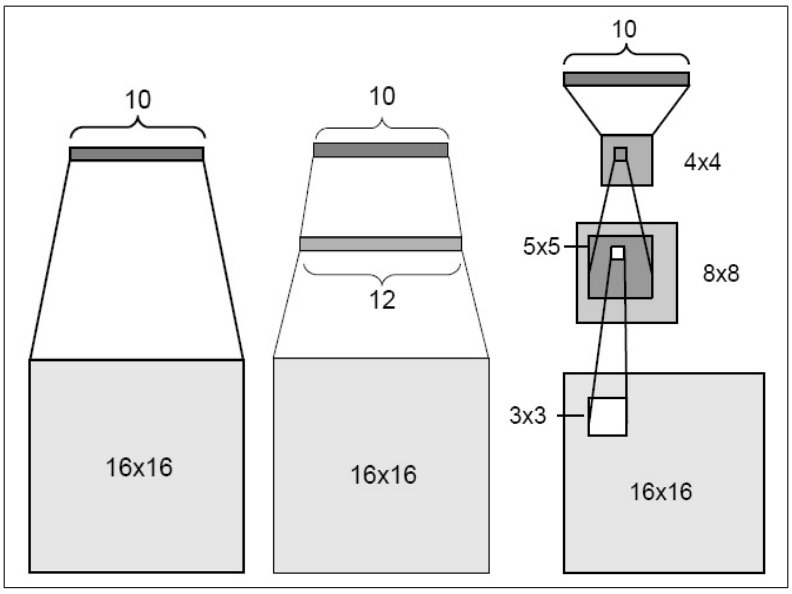

Abbildung 7: Net-1, Net-2 und Net-3

Net-1: KNN ohne verdeckte Schicht

Die 10 Neuronen der Ausgabeschicht sind vollständig mit der Eingabeschicht, die aus 256 Werten besteht, verknüpft, d.h. es werden 2560 Gewichte benötigt. Hinzu kommen noch die Verknüpfungen eines jeden Ausgabeneurons mit einem eigenen Bias-Neuron, so dass dieses lineare Netz insgesamt *2570* Verknüpfungen und Gewichte besitzt. Die Erfolgsquote dieses KNNs lag beim Testen laut [1] bei *80%*.

Net-2: KNN mit einer verdeckten Schicht, deren 12 Neuronen vollständig verknüpft sind

Die Verknüpfung der 12 verdeckten Neuronen mit den 256 Eingabewerten und jeweils

einem Bias-Parameter erfordert $12 \cdot 256 + 12 = 3084$ Gewichte. Jedes der 10 Neuronen in der Ausgabeschicht ist mit den 12 verdeckten Neuronen verknüpft und erhält ebenfalls einen eigenen Biaswert, so dass dieses Netz insgesamt $3084 + 10 \cdot 12 + 10 =$ *3214* Verknüpfungen und Gewichte besitzt. Das sind deutlich mehr Parameter als bei Net-1, doch die Erfolgsquote wird dadurch auf immerhin *87%* angehoben.

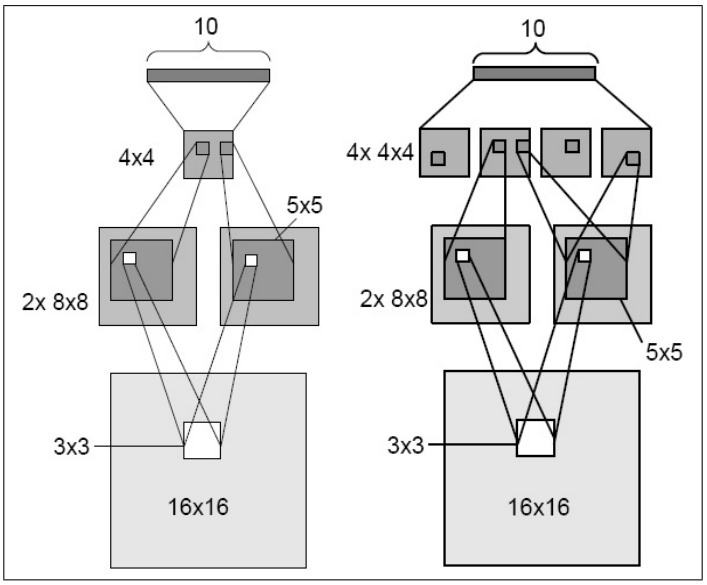

Abbildung 8: Net-4 und Net-5

Net-3: KNN mit 2 verdeckten Schichten, die lokal verknüpft sind

In diesem neuronalen Netz sind die Schichten nicht mehr allesamt vollständig verknüpft. Jedes der 64 Neuronen der ersten verdeckten Schicht ist mit jeweils einem Bias-Neuron und einem Feld von 3x3 Eingabewerten verknüpft. Dieses Feld wird pro Neuron um zwei Pixel verschoben. Auf diese Weise kommen $64 \cdot (9+1) = 640$ Verknüpfungen zustande. Die 16 Neuronen der zweiten verdeckten Schicht sind mit jeweils einem Biaswert und einem 5x5-Feld der ersten verdeckten Schicht verbunden, welches pro Neuron erneut um zwei Pixel verschoben wird. Daraus resultieren demnach weitere $16 \cdot (25+1) = 416$ Verknüpfungen. Die 10 Ausgabeneuronen sind vollständig mit der zweiten verdeckten Schicht und jeweils einem Biasterm verknüpft, so dass man insgesamt $640 + 416 + 10 \cdot (16+1) =$ *1226* Verknüpfungen und Gewichte erhält. Die Erfolgsquote liegt hier beim Testen mit *88,5%* nah an der des KNNs Net-2, allerdings kommt dieses Netz mit weitaus weniger Parametern aus.

Net-4: KNN mit zwei verdeckten Schichten, die lokal verknüpft sind und Weight Sharing auf einer Ebene

Bei diesem Netz kommt zum ersten Mal die Methode des Weight Sharing zum Einsatz: Mehrere Neuronen teilen sich die Parameter, mit welchen die Verbindungen zu einer anderen Schicht gewichtet werden. Den Verknüpfungen dieser Neuronen mit einem Bias-

Neuron werden jedoch weiterhin eigene Gewichte zugewiesen. Weight Sharing führt dazu, dass die Anzahl der Gewichte von der Anzahl der Verknüpfungen abweicht.

Die erste verdeckte Schicht dieses KNNs setzt sich aus zwei Teilen zusammen, die beide aus 64 Neuronen bestehen, welche wie in Net-3 verknüpft sind. Die Neuronen eines jeden Teils teilen sich jeweils die 9 Parameter, welche die Verknüpfungen mit den Eingabewerten gewichten. Außerdem ist jedes Neuron mit einem Bias-Neuron verbunden, was $2 \cdot (64 \cdot (9+1)) = 1280$ Verknüpfungen und $2 \cdot (9+64 \cdot 1) = 146$ Gewichte ergibt. Die 16 Neuronen der zweiten verdeckten Schicht sind mit jeweils 5x5 Neuronen der beiden Teile der ersten verdeckten Schicht verknüpft. Zusammen mit den 16 Biaswerten kommt man auf $16 \cdot (2 \cdot 25 + 1) = 816$ weitere Verknüpfungen und Gewichte. Wie beim KNN Net-3 benötigt die Verbindung der zweiten verdeckten Schicht mit der Ausgabeschicht 170 Gewichte. Somit besitzt dieses Netz *2266* Verknüpfungen, kommt aber mit nur *1132* Gewichten aus. Damit erhalten wir beim Testen eine Erfolgsquote von *94%*, wobei die Anzahl der Parameter im Gegensatz zu Net-3 weiter verringert wurde.

Net-5: KNN mit zwei verdeckten Schichten, die lokal verknüpft sind und Weight Sharing auf zwei Ebenen

Den unteren Teil des Aufbaus dieses KNNs übernehmen wir von Net-4. Die zweite verdeckte Schicht wird jedoch im Gegensatz zu obigem Netz vervierfacht. Jedes Neuron dieser Schicht ist mit jeweils 2·25 Neuronen der ersten verdeckten Schicht verknüpft, wobei die Neuronen eines jeden der vier Teile sich jeweils die 2·25 Gewichte teilen. Berücksichtigt man außerdem die Bias-Parameter, so zählt man auf dieser Ebene $4 \cdot (16 \cdot (2 \cdot 25 + 1)) = 3264$ Verknüpfungen und $4 \cdot (2 \cdot 25 + 16) = 264$ Gewichte. Hinzu kommen noch die $10 \cdot (4 \cdot 16 + 1) = 650$ Verbindungen zwischen der oberen verdeckten Schicht und der Ausgabe, so dass das Netz auf insgesamt *5194* Verknüpfungen, aber auf nur *1060* Gewichte kommt. Die Anzahl der Parameter konnte damit weiter verringert werden. Mit einer Erfolgsquote von *98,4%* schneidet Net-5 zudem am besten ab.

6 Rekurrente Netze

Eine Erweiterung der bisher behandelten KNN bilden die rekurrenten Netze. Wie der Name bereits vermuten lässt, kommt es beim Informationsfluss innerhalb derartiger Netze zu Zyklen. Das bedeutet, dass auch Verbindungen von Neuronen einer Schicht zu Neuronen derselben oder einer vorangegangenen Schicht erlaubt sind. Man versucht mit Hilfe dieser Rückkopplungen zeitlich codierte Informationen in den Daten zu entdecken. Dadurch eignen sich rekurrente Netze vor allem zur Erstellung von Prognosen über die Zukunft oder zur Simulation von menschlichen Verhaltenssequenzen wie der Steuerung der Motorik. Damit ein solches Netz jedoch funktionieren kann, muss eine weitere Komponente explizit miteinbezogen werden: die Zeit.

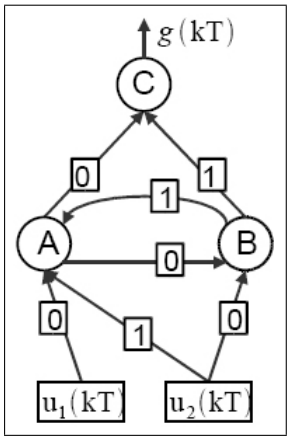

Abbildung 9: Bsp. eines rekurrenten Netzes

Jeder Verknüpfung wird zusätzlich zur Gewichtung eine Zeitverzögerung zugewiesen. Diese Zeitverzögerung ist ein ganzzahliges Vielfaches einer festgelegten Zeitspanne T und kann daher auch gleich 0 ($= 0 \cdot T$) sein. Es ist zu beachten, dass die Summe der Verzögerungen innerhalb eines Zyklus größer 0 zu sein hat, da die Ausgabe eines Neurons zu einem Zeitpunkt kT nicht von der eigenen Ausgabe zum selben Zeitpunkt abhängen soll.

Wie ein derartiges KNN aussehen kann, zeigt Abbildung 9. Zu jedem Zeitpunkt kT, $k \in \mathbb{N}$, erhält das Netz zwei Eingabewerte, $u_1(kT)$ und $u_2(kT)$, und produziert die Ausgabe $g(kT)$. Die Ein- und Ausgabewerte der einzelnen Neuronen zum Zeitpunkt kT kann man der folgenden Tabelle entnehmen:

	A	B	C
Inputs	$u_1(kT)$ $u_2((k-1)T)$ $y_B((k-1)T)$	$u_2(kT)$ $y_A(kT)$	$y_B((k-1)T)$ $y_A(kT)$
Output	$y_A(kT)$	$y_B(kT)$	$g(kT)$

Die Werte $y_A(kT)$ und $y_B(kT)$ beschreiben die Ausgabe des Neurons A bzw. B zum Zeitbunkt kT. Zum Zeitpunkt $k = 1$ produziert das Netz die Ausgabe $g(T)$, die nur vom Eingabewert $u_1(T)$ abhängt. Die Eingabe $u_2(T)$ wirkt sich erst eine Zeiteinheit später auf die Netzausgabe aus.

Wie man auch an diesem Beispiel sehen kann, hängt die Ausgabe eines rekurrenten Netzes nicht immer nur von den aktuellen Eingabewerten ab, wie es bei Feedforward-Netzen der Fall ist. Auch die vorherigen Eingabewerte wirken sich auf Grund der Zyklen weiterhin auf die Ausgabe aus.

7 Literatur

[1] T. Hastie, R. Tibshirani, J. Friedman: *The elements of statistical learning*, Springer, 2001
[2] G. Dreyfus: *Neural Networks - Methodology and Applications*, Springer, 2004
[3] http://www.wikipedia.org
[4] http://www.neuronales-netz.de
[5] http://www.pze.at/linux2/tutor-bu/hege.htm