Zusammenfassung

Zur Lösung vieler Problemstellungen auf dem Gebiet neuronaler Netze reichen einstufige Netze nicht aus, so daß eine verdeckte Schicht Neuronen eingeführt werden muß. Die Schwierigkeit besteht nun darin, diese Schicht optimal zu dimensionieren. Wählt man sie zu groß, d.h. mit zu vielen Hidden-Units, so lernt ein Netz die gegebenen Trainingsmuster nur auswendig, was zur Folge hat, daß das Netz nur schlecht generalisieret. Überdies wird das Lernen unnötig verlangsamt. Wählt man die verdeckte Schicht dagegen zu klein, dann ist das Netz nicht in der Lage, das gestellte Problem zu lösen. Bisher wird eine optimale oder geeignete Topologie von Netzen für praktische Anwendungen durch eine Vielzahl von Versuchen empirisch bestimmt. In dieser Arbeit werden nun für drei unterschiedliche Netzwerktypen, feedforward-Netze, rekurrente Netze und winner-takes-all-Netze, Algorithmen vorgestellt, die ihre Netzwerktopologie selbstständig bestimmen. Bei den Algorithmen handelt es sich im einzelnen um den Cascade-Correlation-Algorithmus, um den Recurrent Cascade-Correlation-Algorithmus und um einen eigenen Algorithmus für winner-takes-all-Netze. Die Algorithmen werden ausführlich anhand vier verschiedener Anwendungen getestet. Der aufwendigste Test für den Cascade-Correlation-Algorithmus und für den Algorithmus für winner-takes-all-Netze besteht in einer Klassifikationsaufgabe, bei der es darum geht, 10000 handgeschriebene Ziffern korrekt zu erkennen. Die Generalisierungsfähigkeit dieser beiden Algorithmen wird dann anhand einer zweiten Menge handgeschriebener Zeichen, wieder aus 10000 Ziffern bestehend, untersucht. Weiterhin wird das Verhalten des Cascade-Correlation-Algorithmus noch an dem Spiralenproblem, einem bekannten Benchmarktest, und an einer anderen Klassifikationsaufgabe untersucht. Der rekurrente Cascade-Correlation-Algorithmus bekommt die Aufgabe, Morsekode zu lernen.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Ziel und Inhalt der Arbeit

Das Ziel dieser Arbeit ist es, Modelle neuronaler Netze zu untersuchen bzw. zu entwickeln, die während des Lernens automatisch eine optimale oder gut geeignete Topologie ihrer Netze bestimmen. Die Fragestellung der automatischen Bestimmung der Topologie neuronaler Netze ist derzeit ein sehr aktueller Forschungsgegenstand neuronaler Netze. Bisher wird eine optimale oder geeignete Topologie solcher Netze für praktische Anwendungen durch eine Vielzahl von Versuchen empirisch bestimmt. Die erfolgreichsten getesteten Lernverfahren sollten in den SNNS, den Stuttgarter Neuronale Netze Simulator, integriert werden.

Untersucht und in den SNNS eingebunden wurden drei neue Lernverfahren. Jedes Lernverfahren ist für das Training eines bestimmten Netzwerktyps geeignet:

1. Cascade-Correlation (CC) [Fahlman et al. 90]:
   Lernalgorithmus zum Trainieren von feedforward-Netzen.
2. Recurrent Cascade-Correlation (RCC) [Fahlman 91]:
   Lernalgorithmus zum Trainieren rekurrenter Netze.
3. Dynamic-Minimum-Distance-Classifier: (DMDC):
   Lernalgorithmus zum Trainieren von winner-takes-all-Netzen.

Der Aufbau und das Vorgehen bei dieser Arbeit ist grob folgender:
Im nächsten Kapitel werden die mathematischen Grundlagen für die Algorithmen CC und RCC gelegt. Im dritten Kapitel gehe ich auf die Realisierung und die Implementierung des CC-Algorithmus ein. Ferner beschreibe ich in diesem Kapitel drei verschiedene Problemstellungen, anhand derer die Leistungsfähigkeit des Cascade-Correlation-Algorithmus getestet wird, um anschließend die Ergebnisse zu diskutieren. Zwei der gestellten Aufgaben kommen aus dem Bereich der Mustererkennung. Es geht hierbei einmal um die Klassifikation gesprochener englischer Vokale und einmal um die Erkennung von 10000 handgeschriebenen Ziffern. Bei der dritten Anwendung handelt es sich um das Spiralenproblem, einem bekannten Benchmarktest für neuronale Netze. Im vierten Kapitel wird aufgezeigt, wie der

RCC-Algorithmus realisiert ist. Dieser Algorithmus bekommt die Aufgabe, Morsekode zu lernen. Im fünften Kapitel wird beschrieben, wie die beiden oben genannten Algorithmen in den SNNS eingebunden sind und wie sie aufgerufen werden. Im sechsten und letzten Kapitel wird ein eigener, auf der Idee eines Minimum-Distance-Classifier basierender Algorithmus hergleitet. Dieser Algorithmus wird ebenfalls mit 10000 hangeschriebenen Ziffern trainiert. Seine Generalisierungsfähigkeit wird mit weiteren 10000 handgeschriebenen Ziffern überprüft.

Kapitel 2

Mathematische Grundlagen der Algorithmen CC und RCC

Dieses Kapitel legt die mathematischen Grundlagen (vgl. [Kreyszig 75] [Bosch 86] [Bosch 87]) für das Verständnis der Algorithmen CC und RCC. Dazu werden zuerst die Begriffe Mittelwert, Varianz und Kovarianz für ein- und zweidimensionale Stichproben definiert. Darauf aufbauend wird anschließend die Bedeutung der Begriffe Regressions- und Korrelationskoeffizient erklärt und danach aufgezeigt, wie diese miteinander zusammenhängen. Es wird sich hierbei herausstellen, daß der Korrelationskoeffizient als ein Maß für den Abstand zweier Zufallsvektoren aufgefaßt werden kann, woraus sich eine sehr bildliche und anschauliche Interpretation der Cascade-Algorithmen ergibt.

2.1 Mittelwert und Varianz einer eindimensionalen Stichprobe

Der empirische Mittelwert $\bar{x}$ einer Stichprobe $x = (x_1, \ldots, x_n)$ vom Umfang n ist definiert durch

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i. \tag{2.1}$$

Die empirische Varianz dieser Stichprobe ist gegeben durch

$$s_x^2 = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})^2. \tag{2.2}$$

An dieser Stelle soll eine oft benutzte andere Form der Varianz hergeleitet werden, die das Arbeiten mit ihr erleichtern kann.

$$\begin{aligned} \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})^2 &= \frac{1}{n-1} \left(\sum_{i=1}^{n} x_i^2 - 2\bar{x}x_i + \bar{x}^2 \right) \\ &= \frac{1}{n-1} \left(\sum_{i=1}^{n} x_i^2 - 2\bar{x} \sum_{i=1}^{n} x_i + n\bar{x}^2 \right) \end{aligned}$$

$$
\begin{aligned}
&= \frac{1}{n-1}\left(\sum_{i=1}^{n} x_i^2 - 2n\bar{x}\underbrace{\frac{1}{n}\sum_{i=1}^{n} x_i}_{\bar{x}} + n\bar{x}^2\right) \\
&= \frac{1}{n-1}\sum_{i=1}^{n} x_i^2 - n\bar{x}^2. \qquad (2.3)
\end{aligned}
$$

2.2 Kovarianz einer zweidimensionalen Stichprobe

Ist $(x, y) = ((x_1, y_1), \ldots, (x_n, y_n))$ eine zweidimensionale Stichprobe, so definiert man

$$
\begin{aligned}
s_{xy} &= \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y}) \\
&= \frac{1}{n-1}\left(\sum_{i=1}^{n} x_i y_i - n\bar{x}\bar{y}\right) \qquad (2.4)
\end{aligned}
$$

als die Kovarianz zwischen zwei eindimensionalen Stichproben x und y. Die Umformung von 2.4 erfolgt analog zu 2.3.

2.3 Regression und Korrelation

Die Regression und die Korrelation sind ganz allgemein ein Maß für die Beziehung zwischen zwei Zufallsvariablen X und Y.

So kann man z.B. nach der Abhängigkeit oder der Beziehung fragen zwischen der Anzahl X von Lernzyklen, die ein neuronales Netz bereits gelernt hat und dem verbleibenden Restfehler Y. Die Zufallsvariable Y soll als abhängig von der Variablen X angesehen werden. X muß keine Zufallsvariable sein. In der Statistik spricht man dann von einer **Regression** von Y bezüglich X, statt wie in der Analysis von einer funktionalen Abhängigkeit zwischen X und Y.

Fragt man nur nach dem Zusammenhang zwischen zwei Zufallsvariablen X, Y und möchte man dabei keine der beiden Variablen als abhängig von der anderen ansehen, wie zum Beispiel bei der Aktivierung X einer Unit in der versteckten Schicht eines neuronalen Netzes und dem Fehler Y einer Ausgabezelle, so spricht man von einer **Korrelation** zwischen X und Y.

2.4 Regressionskoeffizient einer Stichprobe

Die Regressionsanalyse behandelt das Problem, wie aufgrund der Realisierung einer Zufallsvariablen X wahrscheinlichkeitstheoretische Aussagen oder Vorhersagen über die Werte einer zweiten Zufallsvariable Y gemacht werden können. Für Vorhersagen ist es wichtig, daß die beiden Zufallsvariablen nicht stochastisch unabhängig voneinander sind. Je größer

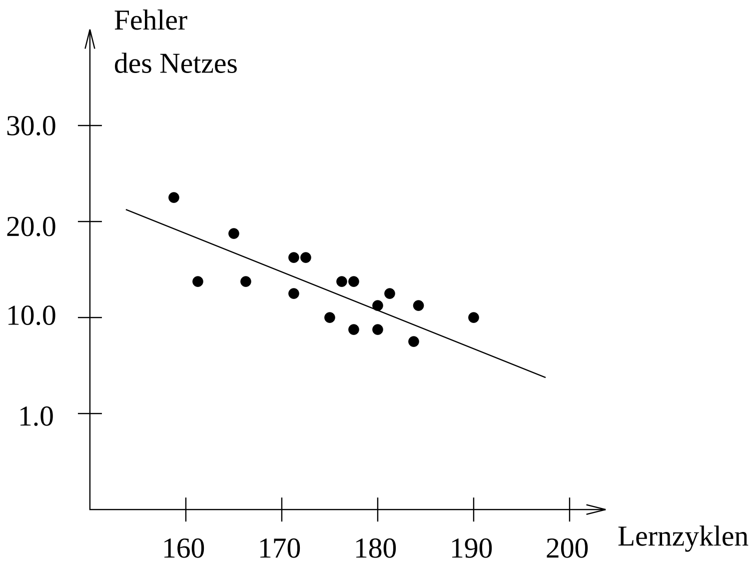

Abbildung 2.1: Regression zwischen dem Fehler eines neuronalen Netzes und der Zahl der Lernzyklen

die Abhängigkeit zwischen den Variablen ist, desto größer die Wahrscheinlichkeit dafür, daß eine Vorhersage eintrifft.

Abb. 2.1 zeigt eine hypothetische Fehlerentwicklung eines neuronalen Netzes in Abhängigkeit von der Anzahl der Lernzyklen. Legt man nach Augenmaß durch die Punktwolke eine sogenannte Ausgleichs- oder Regressionsgerade derart, daß die Stichprobe möglichst gut angepaßt wird, so kann man aufgrund dieser Geraden grobe Vorhersagen über die weitere Fehlerentwicklung des Netzes machen. Da es schwer ist objektiv zu sagen, wann eine Gerade eine Stichprobe möglichst gut anpaßt, benutzt man bei der Regressionsanalyse das Gaußsche **Prinzip der kleinsten Quadrate**, um die Gerade zu finden. Dieses Prinzip besagt:

> Die Gerade ist so zu legen, daß die Summe der Quadrate aller Abstände, die die Punkte von der Geraden haben, möglichst klein wird.

Als Abstandsmaß wird der vertikale Abstand eines Punktes von der Geraden und nicht das Lot auf die Gerade verwendet (vgl. Abb 2.2). Statt einer Geraden könnte auch eine andere Funktion benutzt werden, um eine Stichprobe möglichst gut anzupassen.

Es soll nun allgemein für die Stichprobe $((x_1, y_1), \ldots, (x_n, y_n))$ die Regressionsgerade

$$y = bx + k \tag{2.5}$$

ermittelt werden. Dazu ist die Steigung b und der Achsenabschnitt k zu berechnen. Das vertikale Abstandsquadrat eines Punktes (x_i, y_i) von der Geraden 2.5 beträgt somit

$$(y_i - bx_i - k)^2. \tag{2.6}$$

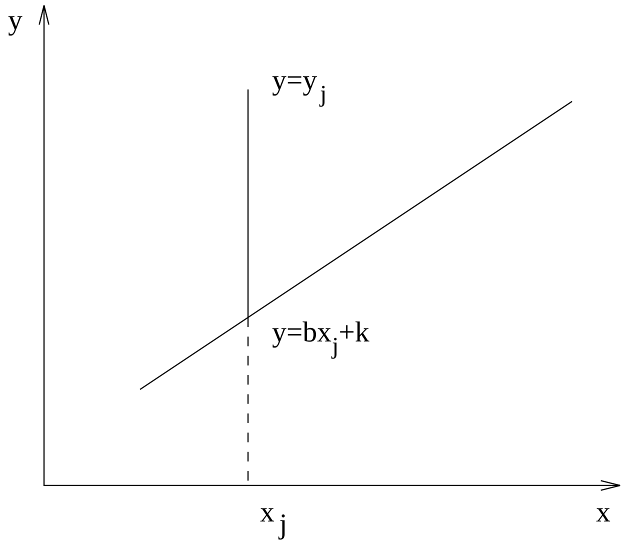

Abbildung 2.2: Berechnung des vertikalen Abstandes

Die Summe a der Abstandsquadrate ist also

$$a = \sum_{i=1}^{n} (y_i - bx_i - k)^2. \tag{2.7}$$

Als notwendige Bedingung für ein Minimum von 2.7 müssen die partiellen Ableitungen $\frac{\partial a}{\partial b}$ und $\frac{\partial a}{\partial k}$ Null werden. Aus 2.7 folgt durch partielle Differentiation nach b und k:

$$\begin{aligned} \frac{\partial a}{\partial b} &= -2 \sum_{i=1}^{n} x_i (y_i - bx_i - k) \\ \frac{\partial a}{\partial k} &= -2 \sum_{i=1}^{n} (y_i - bx_i - k). \end{aligned} \tag{2.8}$$

Setzt man diese Ableitungen gleich Null und definiert zusätzlich

$$\bar{x} = \frac{1}{n}(x_1, \ldots, x_n) \qquad \text{und} \qquad \bar{y} = \frac{1}{n}(y_1, \ldots, y_n), \tag{2.9}$$

so erhält man für die beiden Gleichungen aus 2.8:

$$\begin{aligned} b \sum_{i=1}^{n} x_i^2 + nk\bar{x} &= \sum_{i=1}^{n} x_i y_i, \\ b\bar{x} + k &= \bar{y}. \end{aligned} \tag{2.10}$$

Sofern nicht alle x-Werte identisch sind, hat das lineare Gleichungssystem 2.10 die Lösung

$$b = \frac{\sum_{i=1}^{n} x_i y_i - n\bar{x}\bar{y}}{\sum_{i=1}^{n} x_i^2 - n\bar{x}^2} \qquad \text{und} \qquad k = \bar{y} - b\bar{x}. \tag{2.11}$$

Mit Hilfe der Gleichungen 2.3 und 2.4 kann b aus 2.11 einfacher dargestellt werden. Man erhält für b:

$$b = \frac{(n-1)s_{xy}}{(n-1)s_x^2} = \frac{s_{xy}}{s_x^2}. \tag{2.12}$$

Setzt man die Ergebnisse für b und k aus 2.11 und 2.12 in Gleichung 2.5 ein, so erhält man für die gesuchte Regressionsgerade von y bezüglich x die Darstellung

$$\begin{aligned} y &= \frac{s_{xy}}{s_x^2}x + \bar{y} - \frac{s_{xy}}{s_x^2}\bar{x} \\ &= bx + \bar{y} - b\bar{x}. \end{aligned} \tag{2.13}$$

2.5 Korrelationskoeffizient einer Stichprobe

Als Korrelationskoeffizienten einer Stichprobe $((x_1, y_1) \ldots, (x_n, y_n))$ bezeichnet man den Quotienten

$$\begin{aligned} r &= \frac{\sum_{i=1}^{n} x_i y_i - \frac{1}{n}\sum_{i=1}^{n} x_i \sum_{i=1}^{n} y_i}{\sqrt{\left(\sum_{i=1}^{n} x_i^2 - \frac{1}{n}\left(\sum_{i=1}^{n} x_i\right)^2\right)\left(\sum_{i=1}^{n} y_i^2 - \frac{1}{n}\left(\sum_{i=1}^{n} y_i\right)^2\right)}} \\ &= \frac{s_{xy}}{s_x s_y} \\ &= b\frac{s_x}{s_y}, \end{aligned} \tag{2.14}$$

dabei handelt es sich bei b um den oben definierten Regressionskoeffizienten und bei s_x und s_y um die Standardabweichungen zweier Stichproben x und y. Sowohl s_x als auch s_y sollen existieren. s_{xy} ist die Kovarianz zweier Stichproben x und y.

Wie leicht gezeigt werden kann, liegt der Korrelationskoeffizient r in dem Bereich $-1 \leq r \leq 1$. Damit alle Punkte einer Stichprobe auf der Regressionsgeraden liegen, muß die Summe der Abstandsquadrate (vgl. Gl. 2.7) Null werden. Da es sich um eine Quadratsumme handelt, kann dies nur dann der Fall sein, wenn alle Summanden der Summe Null sind. Setzt man die in Gleichung 2.11 und 2.12 gefundenen Werte für b und k in 2.7 ein, so erhält man

$$\begin{aligned} a &= \sum_{i=1}^{n} \left((y_i - \bar{y}) - b(x_i - \bar{x})\right)^2 \\ &= \sum_{i=1}^{n}(y_i - \bar{y})^2 - 2b\sum_{i=1}^{n}(y_i - \bar{y})(x_i - \bar{x}) + b^2\sum_{i=1}^{n}(x_i - \bar{x})^2 \\ &= (n-1)(s_y^2 - 2bs_{xy} + b^2 s_x^2) \\ &= (n-1)(s_y^2 - b^2 s_x^2) \\ &= (n-1)(1-r^2)s_y^2. \end{aligned} \tag{2.15}$$

Die Umformung von der drittletzten Zeile zur vorletzten Zeile geschieht mit Hilfe der Gl. 2.12. Die letzte Umformung von 2.15 erfolgt mit Hilfe der Gleichung 2.3.

Betrachtet man 2.15 und berücksichtigt, daß es sich bei a um eine positive Quadratsumme handelt, so wird klar, daß r nur Werte zwischen $+1$ und -1 annehmen kann.

Es gibt noch eine weitere Möglichkeit dies zu zeigen. Schreibt man den Korrelationskoeffizienten r (vgl. 2.14) mit Hilfe von 2.3 und 2.4 um, so erhält man für r

$$r = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}. \tag{2.16}$$

Definiert man $(\vec{x}, \vec{y})$ als das skalare Produkt zwischen zwei Vektoren $\vec{x}$ und $\vec{y}$ und $||\vec{x}||$ als die euklidische Norm von $\vec{x}$, wobei

$$\vec{x} = \begin{pmatrix} x_1 - \bar{x} \\ \vdots \\ x_n - \bar{x} \end{pmatrix} \qquad \text{und} \qquad \vec{y} = \begin{pmatrix} y_1 - \bar{y} \\ \vdots \\ y_n - \bar{y} \end{pmatrix}$$

ist, so erhält man für r

$$r = \cos(\alpha) = \frac{(\vec{x}, \vec{y})}{||\vec{x}||\ ||\vec{y}||}. \tag{2.17}$$

Der Korrelationskoeffizient kann somit auch als ein Maß für die Ähnlichkeit zweier Vektoren aufgefaßt werden. Die einzelnen Parameter der Vektoren enthalten die Abweichungen der Realisierungen einer Stichprobe von dem Erwartungswert dieser Stichprobe. Ich bezeichne daher die beiden Vektoren $\vec{x}$ und $\vec{y}$ als Abweichungsvektoren.

Betrachtet man die vorletzte Zeile von 2.15, dann sieht man, daß alle Punkte einer Stichprobe genau dann auf einer Geraden liegen, wenn $s_{xy}^2 = s_x^2 s_y^2$ gilt. Wenn a Null wird, gilt $r = 1$ und wir haben maximale Korrelation zwischen den Werten zweier Stichproben. Abschließend sei noch angemerkt, daß r kein Maß für die Abhängigkeit schlechthin, sondern lediglich ein Maß für die lineare Abhängigkeit zweier Stichproben ist. Ist $r = 0$, so bezeichnet man zwei Stichproben als unkorreliert oder als stochastisch unabhängig.

Kapitel 3

Cascade-Correlation

3.1 Probleme des Backpropagation-Algorithmus

Cascade-Correlation [Fahlman et al. 90], [Hoefeld et al. 91] ist eine Weiterentwicklung des Backpropagation Algorithmus. Der Cascade-Correlation-Algorithmus zeichnet sich gegenüber dem Backpropagation-Algorithmus vor allem durch zwei Vorteile aus:

1. Er konvergiert schneller als Backpropagation.
2. Er bestimmt seine Netzwerktopologie selbstständig.

Bei seinen Untersuchungen, warum Backpropagation so langsam lernt, ist Fahlman auf zwei Hauptgründe gestoßen:

1. Das Schrittweiten-Problem.
2. Das Problem des bewegten Ziels.

Was Fahlman unter diesen beiden Problemen versteht wird nachfolgend beschrieben.

3.1.1 Das Schrittweiten-Problem

Beim herkömmlichen Backpropagation-Algorithmus bewegt man sich mit Hilfe eines Gradienten-Abstiegsverfahren entlang einer Fehlerfläche talwärts. Es kann sehr leicht gezeigt werden, daß bei einer sehr kleinen[1] Schrittweite immer eine Talsohle erreicht wird. Bei der erreichten Talsohle kann es sich dann sowohl um ein lokales als auch um ein globales Minimum handeln, wobei auch das lokale Minimum eine hinreichend gute Lösung für ein Problem darstellen kann. Eine sehr kleine Schrittweite bringt allerdings das Problem mit sich, daß es unter Unständen sehr lange dauern kann, bis ein Minimum erreicht wird. Um dem entgegen zu wirken, wird man versuchen, die Schrittweite so groß wie möglich zu wählen. Wählt man allerdings die Schrittweite zu groß, so birgt das die Gefahr, daß man

[1] Eigentlich muß die Schrittweite unendlich klein sein.

beim Lernen über Minima hinwegspringt, oder daß das Netz zu schwingen beginnt. Eine Möglichkeit diesem Dilemma zu entkommen, stellt der Versuch dar, die Schrittweite der Fehlerfläche mit Hilfe eines Momentum-Terms anzupassen. Der Momentum-Term bewirkt, daß in flachen Gebieten die Schrittweite vergrößert und nach dem Überspringen eines Tals die Schrittweite verkleinert wird. Eine bessere Methode, die Schrittweite der Fehlerfläche anzupassen, stellen Verfahren der zweiten Ordnung wie Quickprop oder Rprop dar. Diese Algorithmen bilden die zweite Ableitung der Fehlerfläche und bekommen so Informationen über das Aussehen der Fehlerfläche selbst. Mit diesem zusätzlichen Wissen wird dann die Schrittweite der Fehlerkurve angepaßt.

3.1.2 Das Problem des bewegten Ziels

Das Problem des sich bewegenden Ziels besagt folgendes: Während des Lernens versucht jede Hidden-Unit, sich in einen Feature-Detector zu entwickeln. Diese Aufgabe wird aber dadurch erschwert, daß sich gleichzeitig alle Hidden-Units verändern und die Units einer Schicht nicht miteinander kommunizieren können. Jede Unit sieht also nur ihre eigene Eingabe und das zurückkommende Fehlersignal, das das zu lösende Problem darstellt. Da sich dieses Signal aber ständig verändert, kann es unter Umständen sehr lange dauern, bis die Aufgabenverteilung der Zellen untereinander klar ist. Folgendes Beispiel verdeutlicht das Problem:

> Verursacht ein Eingabemuster einen besonders großen Fehler, so neigen alle Zellen der verdeckten Schicht dazu, sich auf dieses Problem zu stürzen. Dies kann dann dazu führen, daß sich die Erkennungsrate bereits richtig klassifizierter Muster wieder verschlechtert.

Es muß also zu einer gewissen Aufgabenverteilung der Hidden-Units untereinander kommen. Bis sich diese jedoch vollzogen hat, kann sehr viel Zeit vergehen. Eine Lösung für diese Schwierigkeit stellt der Cascade-Correlation-Algorithmus dar, der immer nur eine Hidden-Unit trainiert. Diese Hidden-Unit versucht einen möglichst großen Teil des verbleibenden Restfehlers zu eliminieren. Bereits trainierte Zellen können sich nicht mehr verändern, da die zu ihnen führenden Verbindungen „eingefroren“ werden. Durch diese Vorgehensweise kann es zu keinen Verteilungskämpfen der Units untereinander kommen.

3.2 Der Cascade-Correlation-Algorithmus

Verbal beschrieben funktioniert der Cascade-Correlation-Algorithmus wie folgt:

1. Beim Start des Verfahrens gibt es nur eine Eingabe- und eine Ausgabeschicht, aber keine versteckte Schicht. Die Eingabeschicht und die Ausgabeschicht werden vollständig miteinander verknüpft.

2. Nun werden alle Verbindungen, die zu einer Ausgabezelle hinführen, mit einem bekannten Lernverfahren, wie z.B. Backprop [Rummelhart et al. 86], Quickprop [Fahlman 88] oder Rprop [Riedmiller et al. 92], so lange trainiert, bis sich der Fehler nicht mehr wesentlich ändert.

3. Als nächstes wird dann eine sogenannte Kandidatenzelle eingeführt, die mit allen Eingabezellen und sofern vorhanden, mit allen Hidden-Units verbunden wird. Zwischen der Kandidatenzelle und den Ausgabezellen gibt es noch keine Verbindungen.

4. Nun wird versucht die Korrelation zwischen der Aktivierung der Kandidatenzelle und dem Fehler der Ausgabezelle zu maximieren. Dazu werden die zu der Kandidatenzelle führenden Verbindungen mit Hilfe eines herkömmlichen Lernverfahrens solange adaptiert, bis sich der Korrelationskoeffizient nicht mehr wesentlich erhöhen läßt.

5. Im letzten Schritt wird die Kandidatenzelle in eine neue Hidden-Unit verwandelt, indem man sie mit allen Ausgabezellen verbindet. Die zu der neuen Hidden-Unit führenden Gewichte werden „eingefroren". Dadurch wird die Hidden-Unit zu einem neuen permanenten Feature-Detector. Jetzt kehrt man zu Schritt 2 zurück.

Dieser Algorithmus wird solange wiederholt, bis der Gesamtfehler des Netzes einen bestimmten vorgegebenen Wert unterschreitet. Statt einer einzelnen Kandidatenzelle kann man auch eine Menge von mehreren Kandidatenzellen verwenden, die dann parallel trainiert werden. Nach Beendigung des Trainings wird die Kandidatenzelle aus der Menge ausgewählt, die die größte Korrelation mit den Ausgabezellen aufweist. Da jede neue Hidden-Unit mit allen vorherigen Hidden-Units verbunden wird, entspricht jede dem Netz neu hinzugefügte Hidden-Unit einem neuen Hidden-Layer. Man bekommt also sehr tiefe Netze. Abbildung 3.3 zeigt, wie ein derartiges Netz nach drei Schritten aussieht.

Abbildungen 3.1 und 3.2 veranschaulichen graphisch wie CC funktioniert. Betrachten wir dazu einmal nur die Ausgabezelle o_1 und die Kandidatenzelle k. CC maximiert die Korrelation zwischen diesen beiden Units. Da aber, wie wir gesehen haben, die Korrelation dem Cosinus zwischen zwei Vektoren entspricht, versucht CC also den Winkel α_1 zwischen den Abweichungsvektoren $\vec{k}$ und $\vec{\delta_1}$ zu minimieren. Nehmen wir einmal an, α_1 könnte auf Null minimiert werden, d.h. $\vec{k}$ und $\vec{\delta_1}$ zeigen exakt in die gleiche Richtung, dann unterscheiden sich beide Vektoren nur noch um einen konstanten Faktor und der verbleibende Restfehler, egal wie groß dieser ist, kann in einem Schritt eliminiert werden. In der Regel wird dies jedoch nicht gelingen. Hat das Netz mehrere Ausgabezellen, so wird CC die Winkel zwischen dem Abweichungsvektor der Kandidatenzelle und den Abweichungsvektoren der Ausgabezellen minimieren. Der Abweichungsvektor der Kandidatenzelle wird sich dann zwischen den Fehlervektoren der Ausgabezellen einpendeln, ähnlich wie es in Abbildung 3.1 dargestellt ist.

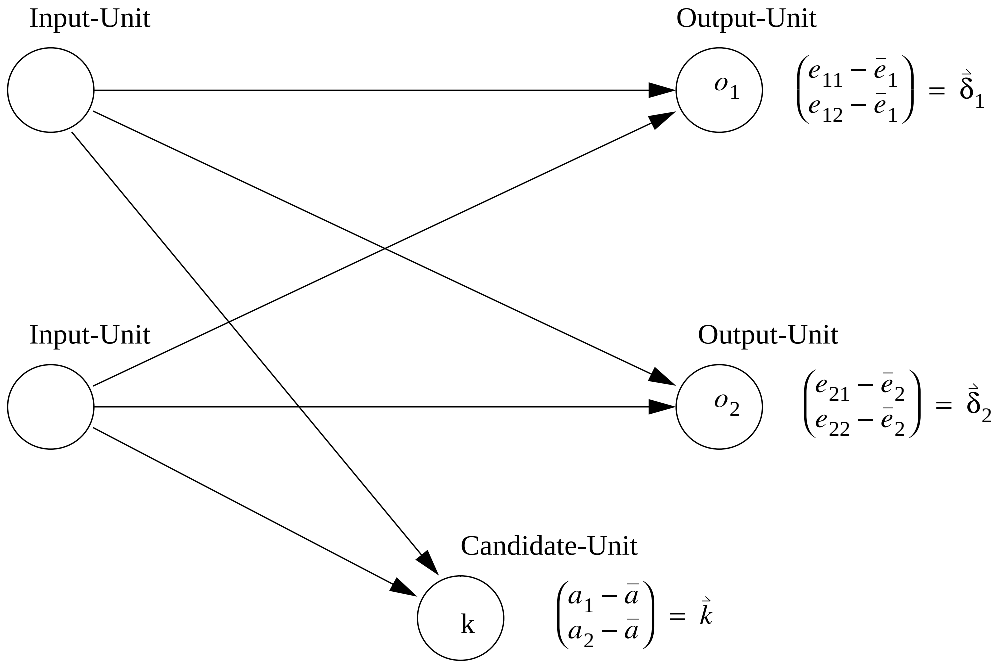

Abbildung 3.1: Ein Netz mit seinen Abweichungsvektoren. Dabei steht a_j für die Aktivierung der Kandidatenzelle bei Eingabemuster j und e_{ij} für den Fehler der Ausgabezelle i bei Eingabemuster j. Die durchschnittliche Aktivierung bzw. Fehler wird mit $\bar{a}$ bzw. mit $\bar{e}_i$ bezeichnet.

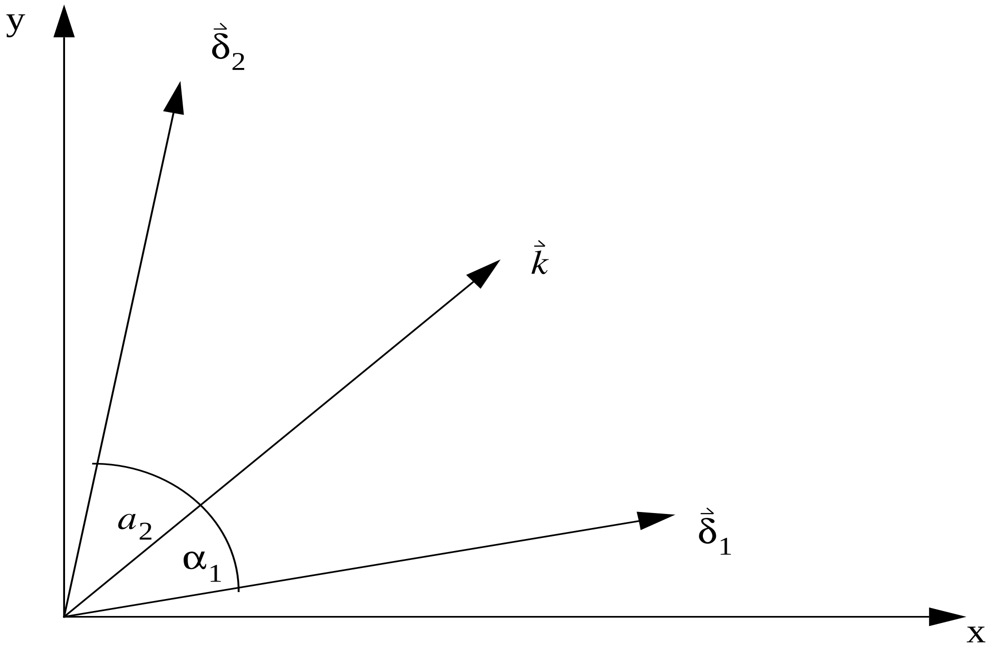

Abbildung 3.2: Die zu obigem Bild gehörenden hypothetischen Abweichungsvektoren.

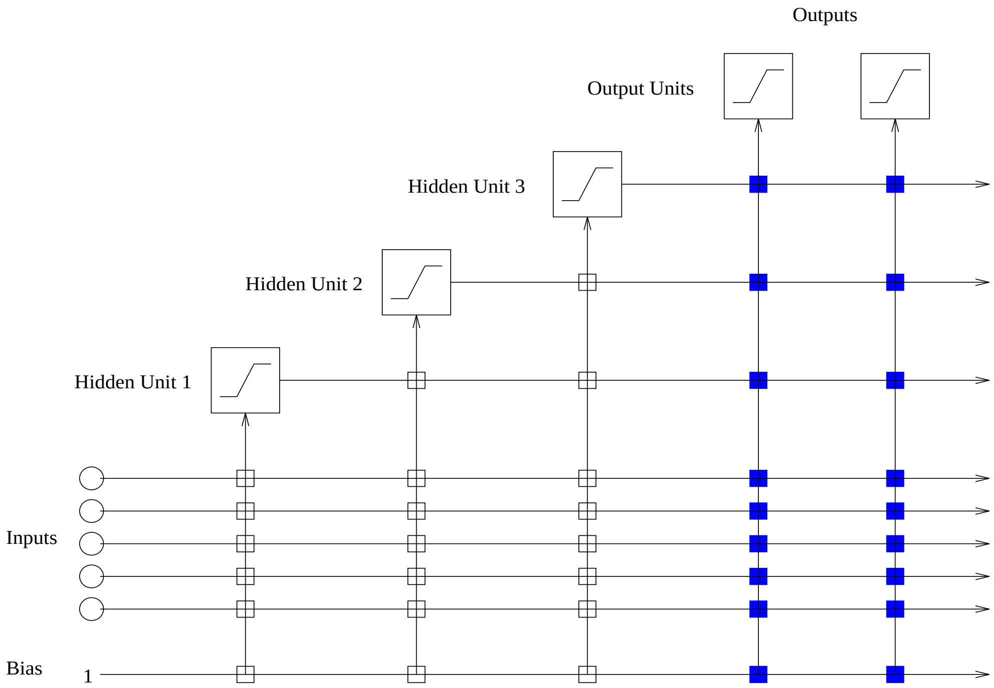

Abbildung 3.3: Ein Netz in der Cascade-Architektur, nachdem drei Hidden-Units eingefügt wurden. Die durch leere Qudrate dargestellten Verbindungen, sind eingefroren. Die Verbindungen, die durch ausgefüllte Quadrate dargestellt sind, werden immer wieder trainiert

3.3 Mathematische Realisierung von Cascade-Correlation

Während des Trainings der Ausgabezellen wird versucht die bekannte Quadratsumme des Fehlers E mit

$$E = \sum_p \frac{1}{2} \sum_o (y_{po} - t_{po})^2 \tag{3.1}$$

zu minimieren, wobei t_{po} der gewünschte Output der Ausgabezelle o bei Muster p und y_{po} die tatsächliche Ausgabe dieser Unit bei dem anliegenden Muster p ist. Da bei CC keine Hidden-Units trainiert werden müssen, berechnet sich der Gradient der Fehlerfläche nach der Regel

$$\begin{aligned} e_{po} &= (y_{po} - t_{po}) f'_p(net_o) \\ \frac{\partial E}{\partial w_{io}} &= \sum_p e_{po} I_{ip}, \end{aligned} \tag{3.2}$$

wobei f'_p die Ableitung der Aktivierungsfunktion der Ausgabezelle o und I_{ip} die Ausgabe einer Input-Unit bzw. einer Hidden-Unit i bei einem anliegenden Muster p ist. Mit w_{io} wird die Verbindung zwischen einer Input bzw. einer Hidden-Unit i und einer Ausgabezelle o bezeichnet.

Nach dem Training der Ausgabezellen werden die Kandidatenzellen so adaptiert, daß die Korrelation C zwischen der Ausgabe y_{po} der Kandidatenzelle und dem Restfehler e_{po} einer

Ausgabezelle maximal wird. Die Korrelation ist nach Fahlman gegeben durch:

$$\begin{aligned} C &= \sum_o \left| \sum_p (y_{po} - \bar{y_o})(e_{po} - \bar{e_o}) \right| \\ &= \sum_o \left| \sum_p y_{po} e_{po} - \bar{e_o} \sum_p y_{po} \right| \\ &= \sum_o \left| \sum_p y_{po}(e_{po} - \bar{e_o}) \right| . \end{aligned} \tag{3.3}$$

Hierbei ist $\bar{y_o}$ die durchschnittliche Aktivierung einer Kandidatenzelle und $\bar{e_o}$ der durchschnittliche Restfehler der Ausgabezelle o über alle Muster p. Die Umformung von der ersten zu der zweiten Zeile folgt unmittelbar aus 2.4. Die letzte Zeile erhält man durch Umsortieren der Summe in der zweiten Zeile. Damit man die Korrelation der Kandidatenzelle mit der gesamten Ausgabeschicht erhält, wird die Korrelation über alle Ausgabezellen betragsmäßig addiert. Die partielle Ableitung[2] von C nach dem Gewicht w_i ergibt sich nun zu

$$\begin{aligned} \delta_p &= \sum_o \sigma_o (e_{po} - \bar{e_j}) f'_p \\ \frac{\partial C}{\partial w_i} &= \sum_p \delta_p I_{pi} , \end{aligned} \tag{3.4}$$

wobei σ_o das Vorzeichen der Korrelation zwischen einer Kandidatenzelle y und der Ausgabezelle o ist.

Wie man sieht, maximiert Fahlman also nicht die Korrelation, sondern die Kovarianz zwischen Kandidaten- und Ausgabezellen. Fahlman schreibt in [Fahlman et al. 90], daß eine der ersten Versionen von CC mit der exakten Ableitung des Korrelationskoeffizienten gearbeitet habe. Nach Fahlman und Lebiere hat es sich jedoch gezeigt, daß CC in den **meisten** Fällen ohne Normierung der Kovarianz besser funktioniert. Warum dies so ist, kann man sich graphisch leicht verdeutlichen (vgl. Abb 3.4 und Abb. 3.5).

Die Länge des Fehlervektores kann als ein Maß für die Größe des Gesamtfehlers einer Unit interpretiert werden. Bildet man die euklidsche Differenz zwischen dem Aktivierungsvektor $\vec{k}$ einer Kandidatenzelle und dem Fehlervektor $\vec{e_1}$ einer Ausgabezelle, so sieht man, daß der Differenzvektor $\vec{d}$ im unnormierten Fall (Abb. 3.5) länger ist als im normierten Fall (Abb. 3.4). Der Kandidatenzelle wird auf diese Weise ein Fehler der Ausgabezelle vorgegaukelt, der größer als der tatsächlich vorhandene Fehler ist. Diese Verzerrung führt dazu, daß sich eine Kandidatenzelle zu einer Ausgabezelle mit einem großen Fehlervektor stärker hingezogen fühlt, als zu einer Ausgabezelle mit einem kleinen Fehler.

Abschließend sei noch angemerkt, daß es für die Implementieung von CC sehr wichtig ist, daß die Ableitung normiert wird. Ohne diese Normierung arbeitet das Verfahren nicht

[2] Bei der Berechnung der Ableitung müssen nur die zu der Kandidatenzelle führenden Gewichte berücksichtigt werden. Alle anderen Gewichte bleiben während der Korrelationsberechnung unverändert.

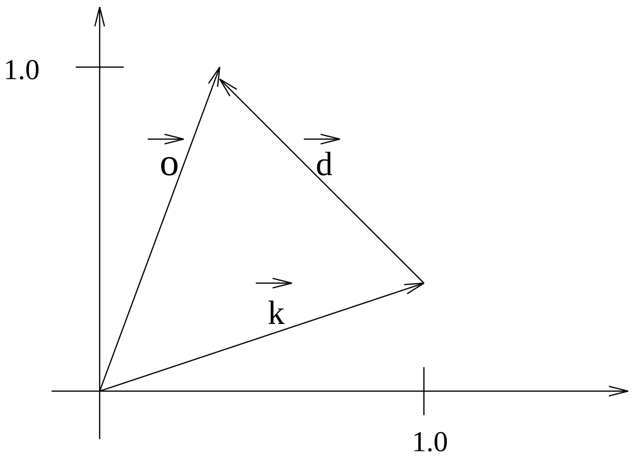

Abbildung 3.4: Vektordifferenz zwischen dem normierten Fehlervektor $\vec{e}$ und dem normierten Abweichungsvektor $\vec{k}$.

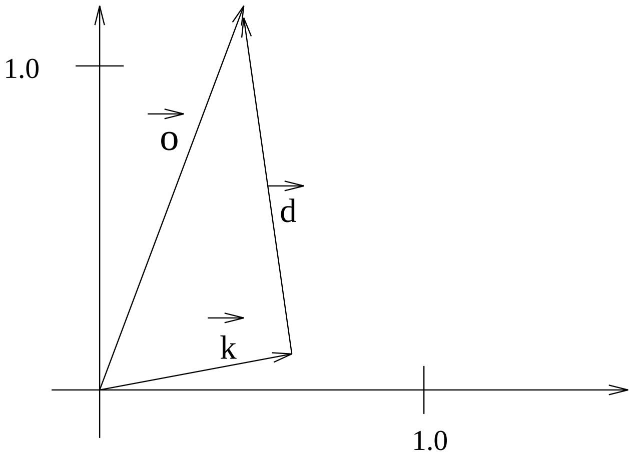

Abbildung 3.5: Vektordifferenz zwischen dem unnormierten Fehlervektor $\vec{e}$ und dem unnormierten Abweichungsvektor $\vec{k}$.

zufriedenstellend. Zu diesem Zweck dividiere ich, wie von R. Scott Crowder[3] in seinem Programm vorgeschlagen, die Ableitung noch durch den konstanten Faktor

$$\sum_p \sum_o (y_{po} - t_{po}) f'_p. \tag{3.5}$$

Hierbei ist y_{po} die Aktivierung der Ausgabezelle o bei Muster p.

[3]Scott Crowder hat das ursprüngliche Cascade-Correlation Programm, das Fahlman in Lisp implementiert hat in ein C-Programm umgeschrieben, so daß ich davon ausgehe, daß der verwendete Normierungsfaktor von Fahlman selbst stammt.

3.4 Details der Implementierung von CC im SNNS

Bevor ich auf die Implementierung von CC im SNNS und der damit verbundenen Probleme eingehe, möchte ich in wenigen Worten den Aufbau und die Philosophie des SNNS [Zell et al. 92] beschreiben.

SNNS ist die Abkürzung für *Stuttgarter Neuronale Netze Simulator*. Der SNNS ist ein Simulator für neuronale Netze, der am Institut für Parallele und Verteilte Höchstleistungsrechner (IPVR) der Universität Stuttgart seit Ende 1989 im Rahmen von mehreren studentischen Arbeiten entwickelt wurde und noch weiterentwickelt wird. Der SNNS besteht aus den drei Teilen:

1. Simulator-Kern:
   Er verwaltet die neuronalen Netze und stellt darauf alle Operationen zur Verfügung. Das Trainieren der Netze findet ausschließlich im Kern statt.

2. X Graphical User Interface (XGUI):
   XGUI ist ein Netzwerkeditor, der es erlaubt, neuronale Netze zu editieren. Des weiteren visulaisiert er alle Netze.

3. Netzwerkspezifikationssprache der Universität Stuttgart (Nessus):
   Nessus gestattet es, mittelgroße bis große strukturierte neuronale Netze mit einer eigens entwickelten Programmiersprache zu erzeugen.

Die strenge Trennung zwischen Kern und Oberfläche bringt mehrere Vorteile mit sich. So kann der Simulator auch ohne die Oberfläche verwendet werden, was z.B. beim Batch-Betrieb sehr wichtig ist. Weitere Vorteile dieser Trennung sind, daß das Lernen nicht durch irgendwelche Oberflächenoperationen verlangsamt wird und daß neue Lernverfahren auch ohne Kenntnisse der X-Oberfläche leicht hinzugefügt werden können.

Der Datenaustausch zwischen Kern und Oberfläche erfolgt über eine Schnittstelle. XGUI kann über diese Schnittstelle auf Daten des Kerns zugreifen. Die umgekehrte Richtung des Datenaustausches ist nicht vorgesehen.

Es hat sich gezeigt, daß diese strenge Aufgabenverteilung zwischen Kern und Oberfläche speziell für CC und RCC eine Vielzahl von Problemen mit sich brachte. Eine der Hauptschwierigkeiten bei der Implementierung von CC auf dem SNNS bestand darin, daß CC kein eigentliches Lernverfahren darstellt, sondern ein Meta-Algorithmus ist, der auf verschiedene Lernverfahren aufgesetzt werden kann. Dadurch war die vom SNNS im XGUI zur Verfügung gestellte Hauptschleife zum Trainieren von Netzen unbrauchbar. Nachfolgend ist das Zusammenspiel von XGUI und Kern beim Trainieren von Netzen abgebildet:

```
/* Hauptschleife im XGUI */

For learnCycle = 1 to MaxLearnCycleNo do begin
  LEARN_beliebigesLernverfahren(...)
end

/* Hauptschleife im Kern */

LEARN_beliebigesLernverfahren(...)
begin
  propagateNetForward(...);
  propagateNetBackward(...);
  updateNet(...);
end
```

Um CC in den SNNS einzubinden, wurde obiges Programmschema wie folgt abgeändert:

```
/* Hauptschleife im XGUI */

For hiddenUnit = 1 to MaxHiddenUnitNo do begin
  LEARN_CascadeCorrelation(...)
end

/* Hauptschleife im Kern */

LEARN_CascadeCorrelation(...)
begin
  if(status <> STOP) then begin
    if(allButtonWasPressed) begin
      allocateStorage(...);
      allButtonWasPressed = FALSE;
    end;
    trainAllOutputUnits(...);
    if((status = testError(...)) == CONTINUE) begin
      trainAllSpecialUnits(...);
      generateHiddenUnit(...);
    end
    trainAllOutputUnits(...);
  end
end
```

Die Funktion *testError(..)* überprüft, ob das Training eines Netzes beendet oder fortgesetzt werden soll und gibt je nach dem den Wert CONTINUE oder STOP zurück. Betrachtet man dieses Programmfragment genauer, so werden eine Reihe von Schwierigkeiten und Veränderungen in der Bedienung des SNNS ersichtlich, die diese Implementierung mit sich bringt. Darauf werde ich in den nächsten Abschnitten genauer eingehen.

3.4.1 Änderungen im Remote-Panel

Im Remote-Panel wurde unter dem Menüpunkt CYCLE bisher immer die Anzahl der maximal zu durchlaufenden Lernzyklen angegeben. Im ersten Programm wird diese Zahl in der Variable *MaxLearnCycleNo* gehalten. Bei CC wird unter diesem Menüpunkt jetzt die maximale Anzahl zu generierender Hidden-Units angegeben. Jedesmal wenn in CC vom Kern aus in die Lernfunktion *LEARN_CascadeCorrelation(..)* gesprungen wird und der Fehler noch klein genug ist (d.h. *testError(...)* liefert *CONTINUE* zurück) wird mit dem Befehl *generateHiddenUnit* eine weitere Hidden-Unit eingeführt.

3.4.2 Speicherverwaltung

Ein weiteres Problem entsteht dadurch, daß das Lernverfahren CC sehr große Mengen an Speicherplatz für sich reservieren muß. Nachdem der ALL Button gedrückt wurde, wird als erste Aktion im Kern der vom Lernverfahren benötigte Speicherplatz reserviert (vgl. *allocStorage*). Der Anstoß, diesen Speicherplatz wieder frei zu geben, muß von der Oberfläche kommen, z.B. beim Wechsel des Lernverfahrens, oder nach dem Laden eines neuen Netzes. Geht der Anstoß, den belegten Speicherplatz wieder frei zu geben von der Oberfläche aus, so bringt dies allerdings zwei entscheidende Nachteile mit sich:

1. Über die gesamte Oberfläche verteilt treten Programmaufrufe und Abfragen auf, die nur den CC-Algorihmus betreffen. Dies führt langfristig zu einer Zerstückelung der Oberfläche und erschwert damit Veränderungen an ihr.

2. Um die den CC-Algorithmus betreffenden Aktionen abzufangen, müßte an der Oberfläche eine Abfrage der Lernfunktion stattfinden. Dies ist zwar prinzipiell möglich, würde aber die strenge Trennung zwischen Kern und Oberfläche untergraben: Die Namen der Lernfunktionen werden vom Kern lokal in einer Funktionstabelle verwaltet. Dies bringt den Vorteil mit sich, daß alle Namensänderungen nur lokal in dieser Tabelle durchgeführt werden müssen. Eine Abfrage der Lernfunktion an der Oberfläche hätte aber zur Folge, daß der Namen des Lernverfahrens auch an der Oberfläche abgeändert werden müßte.

Die oben angeführten Nachteile führen dazu, daß die Wartung des SNNS sehr erschwert und bei häufigem Abweichen durch andere SNNS-Entwickler sogar unmöglich würde.

Um die den CC-Algorithmus betreffenden Daten möglichst lokal zu halten, wird jedesmal nur noch vor dem Allokieren von neuem Speicher im Kern der alte Speicherplatz wieder frei gegeben, sofern dies natürlich notwendig ist. Auch diese Vorgehensweise hat einen Nachteil: Einmal belegter Speicher wird beim Wechsel des Lernverfahrens nicht mehr freigegeben. Dies kann beim Lernen mit einem anderen Lernverfahren unter Umständen zu einem verstärkten Auslagern von Daten auf die Platte führen, wodurch sich das Lernen unnötig verlangsamt. Möchte man den vollen Speicherplatz wieder zur Verfügung haben, so kann dies nur mit einem Neustart des SNNS geschehen. Trotz dieses Nachteils wurde diese Methode zur Speicherverwaltung implementiert.

3.4.3 Ausgaben im Kern

Eine Konvention der Schnittstelle zwischen Kern und Oberfläche besagt, daß vom Kern des SNNS aus keinerlei Ausgaben gemacht werden, sondern daß alle Ausgaben wie z.B. die Fehlerausgabe eines Lernverfahrens oder auch Fehlermeldungen, nur über die Oberfläche zu geschehen haben. Die Idee dieser Vorgabe ist, daß alle Ausgaben an wenigen Stellen abgefangen und weiterverarbeitet werden können. Dies ist z.B. genau dann wichtig, wenn der Kern ohne Oberfläche betrieben werden soll. Betrachtet man sich nochmals obigen Programmkode, so ist offensichtlich, daß sich dieses Prinzip beim CC-Algorithmus nicht durchhalten läßt. Soll ein Benutzer über die Entwicklung der Kovarianz informiert werden, so geht dies nur vom Kern aus. Als Kompromiß wurde eine Variable eingeführt, mit der sich alle vom Kern ausgehenden Meldungen abschalten lassen. Dies ist auch von der Oberfläche aus möglich, indem man im Cascade-Window den Menüpunkt „Print covariance and error" auf OFF stellt.

3.5 Ergebnisse

Im folgenden Abschnitt wird der CC-Algorithmus anhand von drei Anwendungen getestet und die Ergebnisse werden diskutiert. Das erste zu lösende Problem ist das Spiralenproblem. Das Spiralenproblem stellt einen bekannten Benchmarktest für neuronale Netze dar. In der zweiten Anwendung geht es um die Erkennung von gesprochenen englischen Vokalen. Dies ist eine typische Aufgabenstellung aus dem Bereich der Mustererkennung. Die letzte Anwendung behandelt die Klassifikation von handgeschriebenen Ziffern.

3.5.1 Das Spiralenproblem

Zunächst soll kurz der Versuchsaufbau beschrieben werden: Den Ausgangspunkt des Spiralenproblems stellen zwei ineinander gewundene gegenläufige Spiralen (vgl. 3.6) dar. Ein neuronales Netz soll lernen die Punkte dieser beiden Spiralen zu unterscheiden. Jeder Punkt wird durch seine $< x, y >$-Koordinaten dargestellt. Als Trainingsmenge dienen 194 Punkte. Je nach dem, ob ein Punkt der einen oder der anderen Spirale angehört, ist die Ausgabe des Netzes 0 oder 1. Das Ausgangsnetz besteht somit aus zwei Eingabe- und einem Ausgabeneuron (vgl. 5.2).

Da der Cascade-Algorithmus eine Art Meta-Algorithmus ist, der auf verschiedene Lernverfahren aufgesetzt werden kann, war ein Gesichtspunkt der Untersuchungen, wie sich die Auswahl eines der drei Lernverfahren, Backpropagation, Quickprop oder Rprop[4] zum Minimieren des Fehlers bzw. zum Maximieren der Kovarianz, auf das Konvergenzverhalten eines Netzes auswirkt. Zu diesem Zweck wurde ein Netz mit jedem Lernalgorithmus fünfmal trainiert und die so erhaltenen Werte dann gemittelt. Die durch vorhergehende Versuche als geeignet herausgefundenen Lernparameter wurden während dieser Testreihen nicht mehr variiert.

Bei diesen Versuchen hat der Backpropagation-Algorithmus am schlechtesten abgeschnit-

[4] Diese drei Lernverfahren wurden unter CC realisiert.

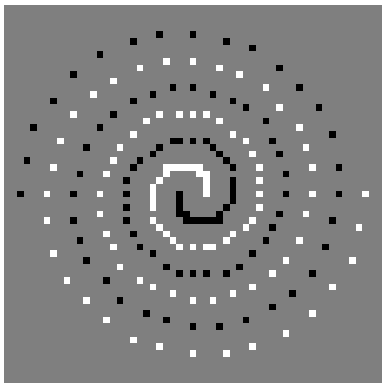

Abbildung 3.6: Trainingsmuster für die Spirale

	Aktivierungsfunktion: Act_LogSym	
	Quickprop	Rprop
	eta_1 = 0.0001 μ_1 = 2.0 ν=0.000001 eta_2 = 0.0009 μ_2 = 2.0	$\eta_1^- = 0.5$ $\eta_1^+ = 1.2$ $\eta_2^- = 0.5$ $\eta_2^+ = 1.2$
Durchschnittliche Anzahl von erzeugten Hidden-Units	14,8	13.6
Durchschnittliche Anzahl von Lernzyklen	4500	4200

Tabelle 3.1: Ergebnisse für das Training des Spiralenproblems mit der symmetrischen Aktivierungsfunktion Act_LogSym über fünf Trainingsläufe gemittelt.

ten. Mit viel Aufwand ist es einmal gelungen, das Spiralenproblem zu lösen. Die Anfangsinitialisierung der Lernparameter[5] war dabei wie folgt: $\eta_1 = 0.2$, $c = 0.1$ und $\eta_2 = 0.8$. Die Momentumterme μ_1 und μ_2 waren auf Null gesetzt.

Nachdem jeweils 10, 15, bzw. 21 Hidden-Units generiert waren, mußten die Lernparameter verkleinert werden, da das Netz nicht mehr weiterlernte. Das fertig trainierte Netz bestand am Ende aus 27 versteckten Zellen. Die durchschnittliche Zyklenzahl, die für das Minimieren des Fehlers, bzw. für das Maximieren der Kovarianz benötigt wurde, betrug in beiden Fällen ungefähr 500 Zyklen, so daß für das Gesamttraining des Netzes rund 27000 Lernschritte nötig waren. Aufgrund dieses schlechten Lernverhaltens, das sich auch bei anderen Aufgabenstellungen bestätigte, wurde Backprop von den weiteren Untersuchungen ausgeschlossen.

Die mit Quickprop, Rprop und einer noch zusätzlich variierten Aktivierungsfunktion erzielten Ergebnisse sind in Tabelle 3.1 und 3.2 festgehalten. Das kleinste Netz, das das Spiralenproblem löste, bestand aus 12 Hidden-Units. Wie man sieht, hängt die Konver-

[5] Die Bedeutung der einzelnen Lernparameter wird in Kapitel 5 beschrieben.

	Aktivierungsfunktion: Act_Logistic	
	Quickprop	Rprop
	$eta_1 = 0.0001$ $\mu_1 = 2.0$ ν=0.000001 $eta_2 = 0.0009$ $\mu_2 = 2.0$	$\eta_1^- = 0.5$ $\eta_1^+ = 1.2$ $\eta_2^- = 0.5$ $\eta_2^+ = 1.2$
Durchschnittliche Anzahl von erzeugten Hidden-Units	25.9	22.3
Durchschnittliche Anzahl von Lernzyklen	8000	6800

Tabelle 3.2: Ergebnisse für das Training des Spiralenproblems mit der gewöhnlichen Aktivierungsfunktion Act_Logistic über fünf Trainingsläufe gemittelt.

genzgeschwindigkeit des Netzes weniger von der verwendeten Lernfunktion als vielmehr von der benutzten Aktivierungsfunktion ab. Die mit Abstand besten Ergebnisse wurden mit der um 0.5 nach unten verschobenen logistischen Aktivierungsfunktion erzielt. Mit anderen Aktivierungsfunktionen konnten keine besseren Ergebnisse erzielt werden. Warum mit der Aktivierungsfunktion Act_LogSym die besten Ergebnisse erzielt wurden, hilft Abbildung 3.7 und 3.8 zu veranschaulichen:

Betrachtet man diese beiden Abbildungen, so erkennt man, daß die Aktivierung der Kandidatenzelle in Abb. 3.7 zwischen 0 und 1 und in Abb. 3.8 zwischen -0.5 und +0.5 liegt. Im ersten Fall wurde die herkömmliche logistische Aktivierungsfunktion verwendet, im zweiten Fall die um 0.5 nach unten verschobene logistische Funktion. Da in die Gewichtsänderung nach der Formel

$$\Delta w_{ij} = \eta \delta_j o_i \tag{3.6}$$

das Produkt aus dem Fehlersignal δ_j einer Zelle j und der Aktivierung o_i ihrer Vorgängerzelle eingeht, muß es also das Ziel sein, dieses Produkt für kleine Fehler klein und für große Fehler groß zu machen. Es ist offensichtlich, daß die symmetrische Aktivierungsfunktion dieser Forderung am besten nachkommt. In Abb. 3.7 erkennt man bei einem Fehlersignal von $+/-0.15$ zwei Bereiche in denen die Aktivierung der Kandidatenzelle Null ist. Das Netz lernt hier also nicht mehr. Bei der Verwendung der symmetrischen Aktivierungsfunktion entfallen diese Gebiete.

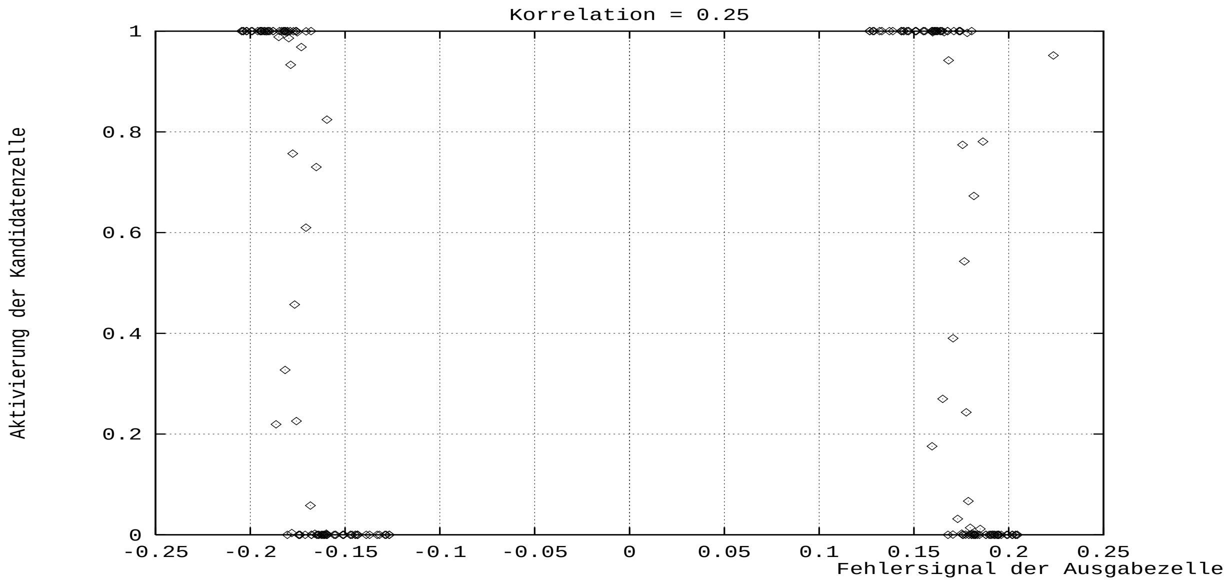

Abbildung 3.7: Abhängigkeit zwischen der Aktivierung einer Kandidatenzelle und dem Fehlersignal der Ausgabezelle nach Berechnung der Korrelation. Es wurde die asymmetrische Aktivierungsfunktion Act_Logistic verwendet.

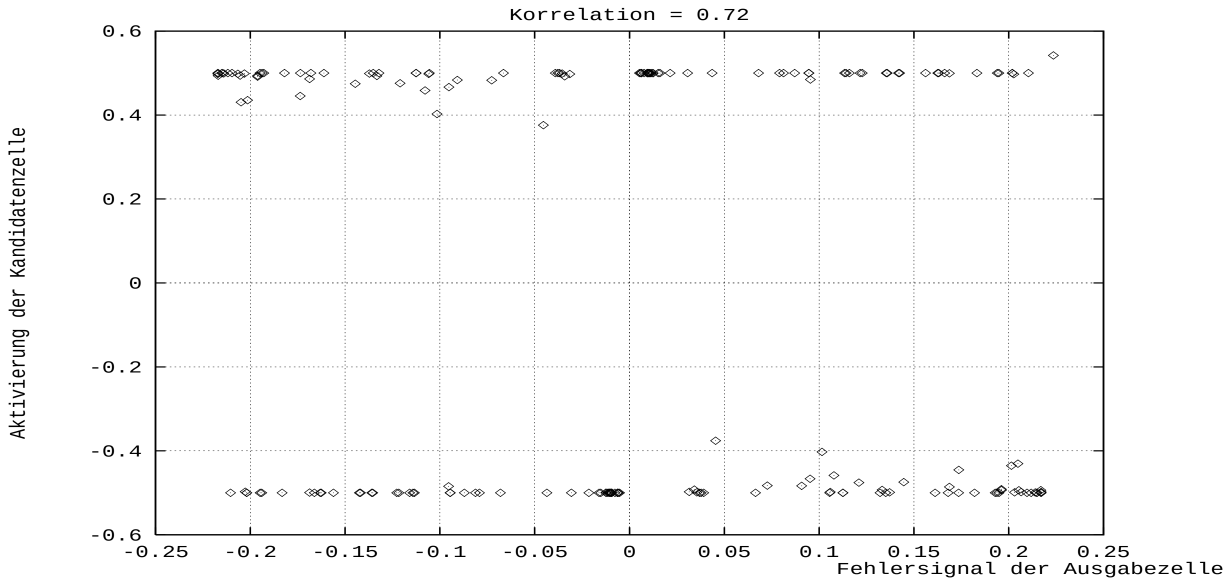

Abbildung 3.8: Abhängigkeit zwischen der Aktivierung einer Kandidatenzelle und dem Fehlersignal der Ausgabezelle nach Berechnung der Korrelation. Es wurde die symmetrische Aktivierungsfunktion Act_LogSym verwendet.

3.5.2 Breite Netze vs. tiefe Netze

Ein weiterer Schwerpunkt der Untersuchungen bestand darin, zu testen, wie sich das Lernverhalten ändert, wenn die neu eingeführten Hidden-Units nicht mehr mit allen vorhergehenden Zellen verbunden sind. Abbildung 3.9 zeigt den Aufbau der untersuchten Netze. Um mit dem CC-Algorithmus das jeweils gewünschte Netz zu generieren, mußte das Programm jedesmal abgeändert werden. Der erste Netzwerktyp (vgl. 3.9) entspricht einem herkömmlichen zweistufigen Netz. Der letzte Netzwerktyp, bei dem jede Hidden-Unit mit allen Vorgängerzellen verbunden ist, stellt ein sehr tiefes Netz dar, bei dem jede versteckte Zelle einen Layer bildet. Aus Zeitgründen war es nicht möglich alle zwischen diesen zwei Netztypen liegenden Netze zu untersuchen. Ich konzentrierte mich daher bei meinen Untersuchungen auf sehr breite bzw. sehr tiefe Netze. Trainiert wurden die Netze für das Spiralenproblem.

Es stellte sich hierbei heraus, daß die tiefen Netze weit besser abschnitten als die breiten Netze. So gelang es für Netze mit nur einer, zwei und drei Verbindungen zwischen den Hidden-Units nicht, die gestellte Aufgabe zu lösen. Mit der halben Datenmenge, d.h. mit wesentlich kleineren Spiralen konnte auch für diese sehr breiten Netze das Problem gelöst werden. Dabei wurden durchschnittlich zwischen 16 und 20 Hidden-Units generiert. Als Aktivierungsfunktion wurde Act_LogSym verwendet. Eine Erklärung für das schlechtere Verhalten der breiten Netze könnte sein, daß sich jede Hidden-Unit im Laufe des Trainings zu einem Feature-Detector ausbildet. Bestehen nun zwischen den einzelnen Zellen der verdeckten Schicht keine oder nur sehr wenige Verbindungen, so gehen diesen Zellen wichtige zusätzliche Informationen über das Eingabemuster verloren, die im Verlauf des Trainings bereits herausgearbeitet wurden. Sind die Zellen der verdeckten Schicht dagegen vollständig miteinander verknüpft, so können die Neuronen auf diese Informationen zugreifen. Wird dieses Wissen nich benötigt, hat das Netz theoretisch die Möglichkeit, die überflüssige Verbindungen auf Null zu trainieren. Aufgrund dieser Ergebnisse wurde darauf verzichtet, dem Benutzer die Möglichkeit zu geben zwischen verschieden tiefen Netzwerktypen zu wählen. Die von CC generierten Netze haben alle die maximale Tiefe.

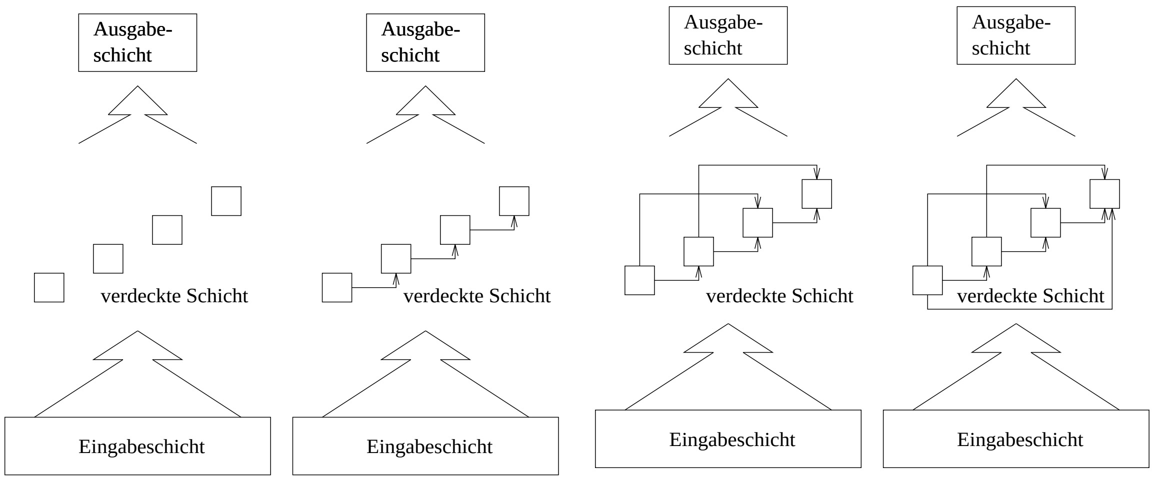

Abbildung 3.9: Netze mit unterschiedlich tiefem Hidden-Layer

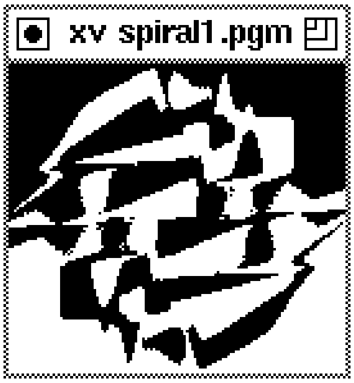

Abbildung 3.10: Generalisierungsverhalten eines Netzes, das unter CC mit Rprop trainiert wurde.

3.5.3 Generalisierungsverhalten von CC

Als letzter Punkt wurde noch das Generalisierungsverhalten der vom CC Algorithmus generierten minimalen Netze untersucht. Hierzu wurden einem fertig trainierten Netz alle Punkte gegeben, die in einem Quadrat um die Spiralen liegen. Je nachdem welcher Spirale ein Punkt zugeordnet wurde (vgl. Abb. 3.10) ist dieser weiß oder schwarz eingefärbt. Punkte, die nicht zugeordnet werden konnten, sind grau eingezeichnet. Abb. 3.10 zeigt die Fläche eines Netzes mit 14 Hidden-Units, das mit dem Rprop-Algorithmus unter CC trainiert wurde. Es ist hier bereits einige Phantasie nötig, um zwei ineinander gewundene Spiralen zu erkennen. Um zu überprüfen, ob dieses schlechte Verhalten durch das sehr kleine Netz verursacht wird, habe ich ein zweites Netz erzeugt, das dann **nicht** mit CC, sondern nur mit Rprop trainiert wurde. Das Netz bestand aus fünf versteckten Schichten, mit jeweils 5, 6, ..., 9 Neuronen pro Schicht. Zwischen den Schichten bestanden Shortcut-Connections. Das Netze hatte also insgesamt 35 Hidden-Units. Abb. 3.11 zeigt das Generalisierungsverhalten dieses Netzes. Es scheint noch schlechter zu sein als bei dem vorherigen Netz. Auffallend sind vor allem die Spitzen, die darauf hindeuten, daß einzelne Gewichte zu hohe Werte angenommen haben. Deshalb habe ich einen weiteren Versuch unternommen, bei dem das Spiralenproblem mit dem Quickprop-Algorithmus unter CC gelöst wurde. Dabei wurde der *weight-decay* Wert in drei Schritten nacheinander von 0.000001 auf 0.0001 angehoben. Abb.3.12 zeigt die Ergebnisse dieser Versuchsreihe. Es ist sehr schön zu beobachten, wie die Spiralen von links nach rechts mit zunehmendem *weight-decay* Wert nach und nach glatter werden. Die erzeugten Netze hatten zweimal 14 und einmal 15 Hidden-Units.

3.5.4 Erkennung von gesprochenen englischen Vokalen

Bei diesem Problem handelt es sich um eine typische Fragestellung aus dem Gebiet der Mustererkennung. Es geht hierbei um die Erkennung von gesprochenen Vokalen aus der englischen Sprache. Die Daten stammen von D.H. Deterding (vgl. [Deterding 89].) Zu den Daten kam man auf folgende Weise: Von 15 Personen wurden 11 englische Wörter jeweils sechsmal ausgesprochen. Die 11 Wörter enthielten alle in der englischen Sprache phonetisch unterscheidbaren Vokale. Nach einigen Vorverarbeitungsschritten wird dann jedes Phonem

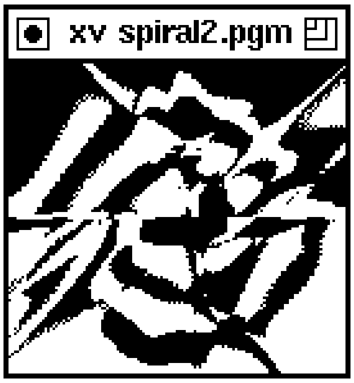

Abbildung 3.11: Generalisierungsverhalten eines Netzes mit 35 Hidden-Units, das nur mit Rprop trainiert wurde.

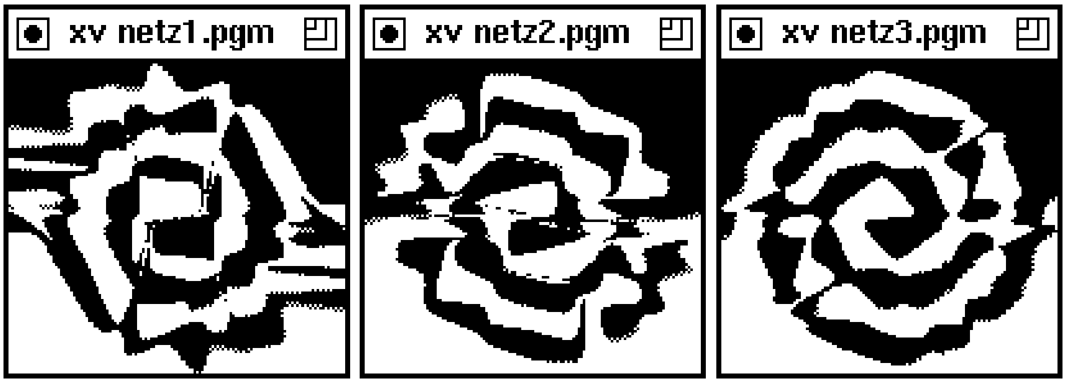

Abbildung 3.12: Einfluß des *Weight-Decay-Lernparameters* auf das Generalisierungsverhalten von Netzen, die unter CC mit Quickprop trainiert wurden.

durch 10 Parameter repräsentiert. Auf diese Weise wurden insgesamt 990 Muster generiert, von denen 528 als Trainigsmuster und die restlichen 462 als Testmuster dienten. Robinson (vgl. [Robinson et al. 88]) hat umfangreiche Tests durchgeführt und ist dabei zu dem Ergebnis gekommen, daß für die Testmuster nur eine bestmögliche Erkennungsrate von 50% erreicht werden kann und stufte daraufhin dieses Problem als mit einem konnektionistischen Ansatz nicht lösbar ein. Robinson hat seine Untersuchungen mit zweistufigen Netzen durchgeführt, die einmal 88 und einmal 527 Neuronen in der mittleren Schicht aufwiesen. Die Eingabeschicht dieser Netze besteht aus 10 und die Ausgabeschicht aus 11 Neuronen.

Interessant an diesem Problem ist zum einen die Anzahl von Hidden-Units, die der Cascade-Algorithmus erzeugt, zum anderen die Frage, wie gut die erzeugten Netze generalisieren. Tabelle 3.3 zeigt die Ergebnisse für verschiedene Aktivierungsfunktionen. Bei dieser Versuchsreihe wurde darauf verzichtet die Ergebnisse über mehrere Versuchsreihen zu mitteln. Die Testergebnisse zeigen zwei Dinge:

1. Die generierten Netze sind ohne Ausnahme alle deutlich kleiner als die von Robinson verwendeten Netze.

2. Das Generalisierungsverhalten der erzeugten Netze ist sehr schlecht und unterstreicht Robinsons These, daß das Problem nicht mit einem konnektionistischen Ansatz lösbar ist.

3.5.5 Erkennung handgeschriebener Ziffern

Bei der letzten und umfangreichsten Anwendung des CC-Algorithmus geht es um die Klassifikation handgeschriebener Ziffern. Die Ziffern sind bereits in Höhe und Breite normiert. Die Auflösung beträgt 16 x 16 Pixel. Tabelle 3.4 zeigt die Ergebnisse zu verschiedenen Zeitpunkten des Trainings. Um das Netz bestehend aus 15 Hidden-Units zu generieren, waren 2550 Lernzyklen notwendig. Getestet wurde das Generalisierungsverhalten mit weiteren 10000 Ziffern. Von diesen Testziffern wurden 96.3% korrekt und 0.8% falsch zugeordnet. Weitere 2.9% konnten überhaupt nicht eingeordnet werden.

3.6 Schlußbemerkungen

Um herauszubekommen, ob die von CC erzeugten Netze auch wirklich minimal sind, habe ich zahlreiche Versuche unternommen, bei denen mit dem CC-Algorithmus erzeugte Netze initialisiert wurden, um dann von einem herkömmlichen Lernalgorithmus trainiert zu werden. Bei keinem Versuch ist es gelungen, die vom CC-Algorithmus erzeugten Netze erneut zu trainieren. In der Regel mußten die Netze mehr als doppelt so groß gewählt werden, damit sie erfolgreich lernen konnten. Dabei hat sich des weiteren herausgestellt, daß Netze, die die gleiche Anzahl von Hidden-Units haben, auch mit einem herkömmlichen Algorithmus besser trainiert werden konnten, wenn die Art und Weise der Verknüpfungen der von CC entsprach.

	Aktivierungsfunktion: Act_LogSym		Aktivierungsfunktion: Act_Logistic	
	Quickprop	Rprop	Quickprop	Rprop
Anzahl der generierten Units	21	20	19	19
Anzahl der benötigten Lernzyklen	7350	7000	6600	6500
Erkennungs-rate auf den Testdaten	33%	32%	35%	36%

Tabelle 3.3: Die Ergebnisse für das Training der Sprachdaten mit einer symmetrischen und einer asymmetrischen Aktivierungsfunktion. Die Lernparameter für Quickprop waren: $eta_1 = 0.0001$ $\mu_1 = 2.0$ ν=0.0001 $eta_2 = 0.0009$ $\mu_2 = 2.0$. Rrop lernte mit den Parametern: $\eta_1^- = 0.5$ $\eta_1^+ = 1.2$ $\eta_2^- = 0.5$ $\eta_2^+ = 1.2$

$eta_1 = 0.0001$ $\mu_1 = 2.0$ ν=0.0001 $eta_2 = 0.0009$ $\mu_1 = 2.0$	
Zahl der generierten Hidden-Units	Prozentsatz der bereits korrekt klassifizierten Trainingsmuster
1	92.3%
3	95.4%
6	97.2%
10	98.9%
15	100.0%

Tabelle 3.4: Prozentsatz der bereits korrekt klassifizierten Trainingsmuster beim Trainieren von 10000 handgeschriebenen Ziffern in Abhängigkeit von der Zahl der generierten Hidden-Units.

Kapitel 4

Rekurrent Cascade-Correlation

Recurrent Cascade-Correlation (RCC) [Fahlman 91] ist eine rekurrente Version von Cascade-Correlation. RCC ermöglicht es, eine Folge von Eingabemustern auf eine Folge von Ausgabemustern abzubilden. Wie bei CC wird immer dann eine neue Hidden-Unit eingeführt, wenn das Training stagniert. Auf diese Weise entstehen sehr kleine und kompakte Netze. Ein fertig trainiertes Netz stellt einen endlichen Automaten dar. Da die mathematische Idee dieselbe ist wie bei CC, werde ich hierauf nur kurz eingehen, dafür aber die Einsatzgebiete rekurrenter Netze aufzeigen (vgl. [Elman 89]).

4.1 Neuronale Netze und die Zeit

Die Zeit spielt im Leben der Menschen bei den vielfältigsten Aufgenstellungen eine sehr wichtige Rolle. So ist die Zeit z.B. untrennbar mit Sprache und Sprachverstehen verbunden. Sprache kann als zeitliche Abfolge von Informationen aufgefaßt werden. In der heutigen Linguistik wird diesem Unstand allerdings nur ungenügend Rechnung getragen. So gehen die Linguisten davon aus, daß ihnen die gesamte Information eines Satzes zu einem bestimmten Zeitpunkt in Form eines Syntaxbaumes zur Verfügung steht. Man spricht in diesem Zusammenhang auch von einer Parallelisierung der Zeit.

Allgemein kann man sagen, daß es prinzipell zwei Möglichkeiten gibt, wie mit Mustern verfahren werden kann, die eine zeitliche Ausdehnung haben:

1. Die zeitliche Abfolge eines Musters wird mit der Dimension des Mustervektors assoziiert. Das heißt konkret: das erste zeitliche Ereignis wird durch das erste Element des Vektors repräsentiert, das zweite Ereignis wird mit dem zweiten Element des Vektors verknüpft usw. . Der gesamte Vektor wird dann parallel verarbeitet.
2. Die Zeit wird implizit durch ihre Auswirkungen auf das Gesamtsystem repräsentiert.

Wählt man den ersten Ansatz zur Repräsentation der Zeit, so entstehen eine Vielzahl von Problemen:

1. Die Eingabe muß solange gepuffert werden, bis das gesamte Muster anliegt, um dann parallel verarbeitet zu werden. Dies könnte über eine Art Schieberegister geschehen.

Es ist allerdings sehr fraglich ob biologische Systeme derartige Schieberegister haben, wie folgende Überlegung verdeutlicht: In einer normalen Gesprächssituation ist es oft der Fall, daß der Hörer weiß, was ihm der Sprecher mitteilen will, bevor dieser seinen Satz beendet hat. Solche Fähigkeiten wären nicht zu beobachten, wenn nur vollständig geäußerte Sätze verarbeitet würden.

2. Ein derartiges Schieberegister müßte eine starre Länge haben, da es für den längsten vorkommenden Vektor passen muß, mehr noch, alle Vektoren hätten damit dieselbe Dimension.

3. Das Hauptproblem dieses Ansatzes ist aber, daß nicht zwischen relativer und absoluter Zeit unterschieden werden kann. Betrachtet man einmal die beiden Vekoren $(0,1,1,1,0,0,0)$ und $(0,0,0,0,1,1,1)$, so sind diese beiden Vektoren Instanzen desselben Musters zu unterschiedlichen Zeitpunkten. Nimmt man jedoch eine geometrische Interpretation der beiden Vektoren vor, so sind sie auf einmal gar nicht mehr ähnlich. Ein konnektionistisches Modell kann zwar trainiert werden diese beiden Vektoren als ähnlich anzusehen, dies ist dann aber das Ergebnis eines externen Lehrers und nicht die Folge der Ähnlichkeit der Muster selbst.

4.2 Modell von Elman

Stellt man die Zeit, wie oben bereits erwähnt, implizit durch ihre Auswirkungen auf den Prozeßablauf dar, so muß man das neuronale Netz mit einer Art Gedächtnis ausstatten, das es befähigt, sich seinen letzten Zustand zu merken. Eine Möglichkeit ein derartiges Kurzzeitgedächtnis zu realisieren stellen rekurrente Netze dar. Eines der zur Zeit am weitesten verbreiteten Modelle ist das Elman-Modell (vgl. Abb. 4.1), das mit diskreten Zeitschritten arbeitet. Die Ausgabe der Hidden-Units zum Zeitpunkt t wird auf alle Hidden-Units zurückgekoppelt und dient zum Zeitpunkt $t+1$ wieder als Eingabe. Die Neuronen, in denen die alte Aktivierung der Hidden-Units abgespeichert ist, stellen eine Art Zustandsvariable dar. Elman bezeichnet diese Neuronen als Kontextzellen. Da ein derartiges Netz einen endlichen Automaten darstellt, kann es einfache reguläre Grammatiken lernen. Rekurrente Netze können somit als ein wichtiges Modell für den Spracherwerb, insbesondere für das Lernen der einer Sprache zugrunde liegenden Grammatik, angesehen werden. Da die Fehlertoleranz eine der Fähigkeiten ist, die die neuronalen Netze besonders auszeichnet, können rekurrente neuronale Netze uns einen Hinweis darauf geben, warum wir auch noch syntaktisch stark verstümmelte Sätze richtig verstehen können.

4.3 Die RCC-Architektur

Das Modell von Elman läßt sich nicht ohne weiteres in die Cascade-Architektur einbinden und muß daher dieser Architektur angepaßt werden: So widerspricht es z.B. dem Konzept der Cascade-Architektur, daß die Ausgabe neuer Hidden-Units als Eingabe bereits existierender Hidden-Units dient. Von einer einmal eingeführten Hidden-Unit können neue Verbindungen nur noch wegführen. Die zu der Zelle hinführenden Gewichte sind eingefroren

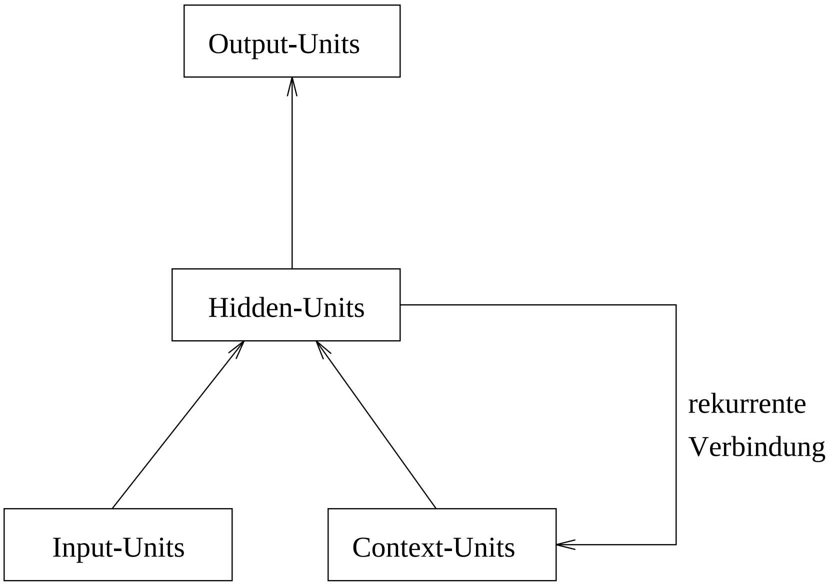

Abbildung 4.1: Schematische Darstellung eines Netzes in der Elman-Architektur

und ihre Anzahl ist unveränderlich. Um wieder in Einklang mit der Cascade-Architektur zu gelangen, wird jede Kandidatenzelle mit nur einer selbstrekurrenten Verbindung versehen (vgl Abb. 4.2). Diese selbstrekurrente Verbindung wird mit allen anderen Eingabeverbindungen so mittrainiert, daß die Korrelation zwischen der Aktivierung der Kandidatenzelle und dem Restfehler der Ausgabezellen maximal wird. Beim Hinzufügen der Kandidatenzelle zum Netz wird die rekurrente Verbindung mit allen anderen zu der neuen Zelle hinführenden Gewichten eingefroren.

4.4 Mathematische Realisierung von RCC

Die Ausgabe $V(t)$ einer selbstrekurrenten Unit r ist gegeben durch:

$$V(t) \quad = \quad f_{act}\left(\sum_i I_i(t)w_{ir} + V(t-1)w_s\right), \tag{4.1}$$

mit den Bezeichnungen:
f_{act}: eine beliebige Aktivierungsfunktion
w_{ir}: das Gewicht der Verbindung zwischen der Eingabeunit i und der Unit r
w_s: das Gewicht der selbstrekurrenten Verbindung
I_i: Ausgabe der Eingabezelle i

Die Ableitung von V(t) bezüglich w_{ir} und w_s ergibt sich zu:

$$\frac{\partial V}{\partial w_{ir}} \quad = f'_{act}(t)(I_i(t) + w_s \tfrac{\partial V(t-1)}{\partial w_{ir}}) \tag{4.2}$$

und

$$\frac{\partial V}{\partial w_s} \quad = f'_{act}(t)(V(t-1) + w_s \tfrac{\partial V(t-1)}{\partial w_s}). \tag{4.3}$$

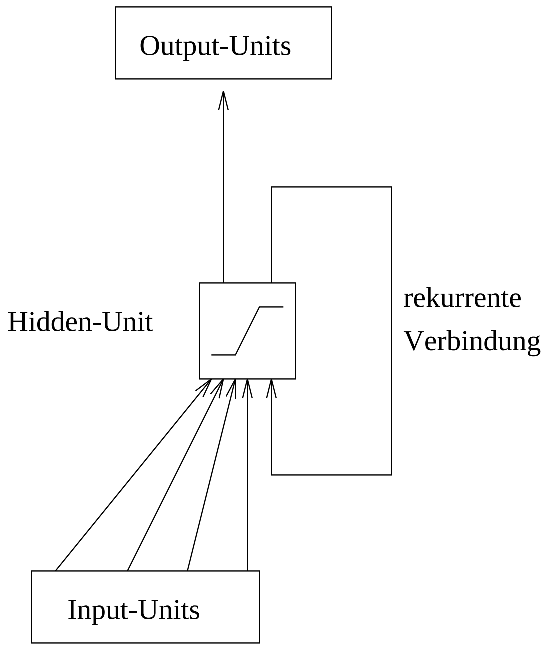

Abbildung 4.2: Schematische Darstellung eines RCC-Netzes.

Der letzte Summand zeigt den Einfluß der rekurrenten Verbindung auf das Ergebnis der Ableitung. Zum Zeitpunkt $t = 0$ geht man davon aus, daß sowohl die vorherige Aktivierung als auch die vorherige Ableitung Null ist. Die Maximierung der Korrelation erfolgt dann auf dieselbe Art und Weise wie bei CC.

4.5 Details der Implementierung im SNNS

Zu den im vorherigen Kapitel bereits beschriebenen Problemen bei der Implementierung von CC im SNNS sollen nachfolgend noch die RCC-spezifischen Probleme und ihre Lösung aufgelistet werden.

4.5.1 Neue Aktivierungsfunktionen

Für RCC mußten folgende Aktivierungsfunktionen neu eingeführt werden:
Act_RCC_Logistic, Act_RCC_Tanh, Act_RCC_LogSym, Act_RCC_Identity. Diese Aktivierungsfunktionen unterscheiden sich von den bereits implementierten Aktivierungsfunktionen nur dadurch, daß sie zu dem Zeitpunkt $t = 0$, bei der Berechnung der Aktivierung einer Unit, deren rekurrenente Verbindung ignoriert. Den neuen Aktivierungsfunktionen wird eine zusätzliche Variable übergeben, die den diskreten Zeitpunkt bei der Abarbeitung eines Musters enthält. Da im SNNS der Aktivierungsfunktion ein Pointer auf die Unit übergeben wird, deren Aktivierung berechnet werden soll, gibt es keine andere Möglichkeit zur Berechnung der Aktivierung, als die Einführung der oben beschriebenen Aktivierungsfunktionen. Würde man der Aktivierungsfunktion dagegen einen Pointer auf den erste Verbindung

übergeben, so hätten obige Funktionen eingespart werden können.

4.5.2 Position der rekurrenten Gewichte in der Verbindungsliste

Im SNNS werden die zu einer Unit führenden Links in einer Liste verwaltet. Um die rekurrente Verbindung eindeutig zu markieren, muß diese Verbindung in der Link-Liste **immer** an erster Stelle stehen. Sollte dies einmal nicht der Fall sein, so wird dies bereits bei der topologischen Überprüfung des Netzes bemerkt, und es erscheint eine Fehlermeldung. Dies kann eventuell bei von Hand erzeugten Netzen der Fall sein, oder aber wenn versehentlich ein falsches Netz geladen wurde. Die von RCC erzeugten Netze haben alle automatisch die richtige Topologie. Die feste Positionierung der rekurrenten Gewichte in der Verbindungsliste bringt einen Geschwindigkeitszuwachs mit sich, da die Aktivierung der rekurrenten Zellen ohne zusätzliche Abfrage berechnet werden kann, wie folgender Programmausschnitt verdeutlicht:

```
/* Berechnung der Aktivierung einer Rekurrenten Unit */
    Act_RCC_beliebigeActFunktion(...,Zeitpunkt)
    {
     if(Zeitpunkt == 0) then begin
       for(link = secondLink; link != NULL; link = link.next) do begin
         net += link.weight * VorgaengerUnit.out;
       end
     end
     else begin
       for(link = firstLink; link != NULL; link = link.next) do begin
         net += link.weight * VorgaengerUnit.out
       end
     end
     return(beliebigeActFunktion(net + bias));
    }
```

4.5.3 Berechnung der Ausgabe einer rekurrenten Zelle

Betrachtet man sich die mathematische Gleichung 4.1 zur Berechnung der Aktivierung einer rekurrenten Zelle, so sieht man, daß es bei der Berechnung der Ausgabe einer rekurrenten Zelle ein weiteres Problem auftaucht: Die Ausgabe einer solchen Zelle muß zum Zeitpunkt t abgespeichert werden, damit man zum Zeitpunkt $t+1$ auf diese Ausgabe zurückgreifen kann. Da im SNNS zwischen Aktivierung und Ausgabe eines Neurons unterschieden wird, ist die Lösung dieses Problems trivial, wie folgender Programmausschnitt zeigt:

```
/* Berechnung der Ausgabe einer rekurrenten Unit */
   Unit.act = Act_RCC_beliebigeActFunktion(...);
   Unit.out = OUT_Func(Unit.act);
```

4.5.4 Das *resetArray*

Wie in der Einleitung bereits erwähnt, ist ein rekurrentes Netz in der Lage, eine Folge von Eingabevektoren auf eine Folge von Ausgabevektoren abzubilden. Die Zahl der Eingabevektoren, die eine bestimmte Ausgabe kodieren, kann schwanken, so daß nicht mehr wie bisher jeder Ausgabe eine Eingabe einer bestimmten Länge zugeordnet werden kann. Ein Beispiel hierfür stellt der Morsekode (vgl. Tabelle 4.1) dar. Wie man sieht schwankt die Zahl und die Abfolge langer und kurzer Piepstöne, die einen Buchstaben darstellen. Ein kurzer Piepston ist durch einen Punkt und ein langer durch einen Strich dargestellt. Die Kodierung des Morsekodes entspricht der von Scott Crowder[1] vorgenommen Verschlüsselung. Wie die Trainingsmuster dem SNNS-Format angepaßt werden, wird ausführlich im nächsten Abschnitt besprochen.

Im vorherigen Abschnitt haben wir gesehen, daß es für die Berechnung der Aktivierung einer Unit wichtig ist, zu wissen, ob es sich bei einem Eingabevektor, um den ersten zu diesem Muster gehörenden Vektor handelt oder nicht. Beim ersten Vektor werden die rekurrenten Verbindungen ignoriert. Um also dem Netz in der Lernphase eindeutig mitteilen zu können, wann ein neues Muster anliegt, wurde ein sogenanntes *resetArray* eingeführt. Dieses Array enthält soviele Parameter wie es Eingabevektoren gibt. An all den Positionen, an denen ein Eingabevektor steht, der den Beginn eines neuen Musters markiert, steht in dem *resetArray* eine 1, an allen anderen Positionen steht eine 0. Bei der Berechnung der Aktivierung einer Unit, wird der Aktivierungsfunktion dann jeweils der einem Eingabevektor zugeordnete Wert aus dem *resetArray* übergeben.

4.5.5 Das *LinkArray*

Etwas vereinfacht hat im SNNS jeder Link folgende Darstellung:

```
struct LINK {
  double weight;
  double value_a,value_b,value_c;
  struct Link *sourceUnit;
}
```

In der Variablen *weight* ist das Gewicht der Verbindung zwischen der Quellzelle und der Zielzelle abgespeichert. Die drei Variablen *value_a, value_b* und *value_c* sind nicht belegt, und können für beliebige Daten als Zwischenspeicher dienen. Wird der RCC-Algorithmus in Verbindung mit dem Quickprop-Algorithmus verwendet, so braucht man zur Datenzwischenspeicherung vier Parameter, so daß die drei zur Verfügung gestellten Variablen nicht mehr ausreichen.

Die einfachste Möglichkeit diese Schwierigkeit zu lösen ist es, die Struktur *LINK* um die Variable *value_d* zu erweitern. Da aber der von einem Netz belegte Speicherplatz in der Hauptsache durch die Zahl seiner Verbindungen zwischen den einzelnen Zellen festgelegt ist, würde bei dieser Lösung der Speicherbedarf aller Netze sprunghaft ansteigen. Weil,

[1] Scott Crowder hat das ursprüngliche RCC-Programm, das Fahlman in Lisp implementiert hat, in ein C-Programm übersetzt.

Morsefolge	Zeichen	Morsefolge	Zeichen
.-	A	-.	N
-...	B	- - -	O
-.-.	C	.- -.	P
-..	D	- -.-	Q
.	E	.-.	R
..-.	F	...	S
- -.	G	-	T
....	H	..-	U
..	I	...-	V
.- - -	J	.- -	W
-.-	K	-..-	X
.-..	L	-.- -	Y
- -	M	- -..	Z

Tabelle 4.1: Kodierung des Alphabets

Zeichenkodierung	Zeichen
ON-OFF-ON-ON-OFF	A
ON-ON-OFF-ON-OFF-ON-OFF-ON-OFF	B
...	...

Tabelle 4.2: Kodierung des Morsekodes

nur der RCC-Algorithmus in Verbindung mit Quickprop diese zusätzliche Speicherstelle braucht, habe ich das Feld *LinkArray* eingeführt, in dem der zusätzliche Wert abgespeichert wird.

4.5.6 Format der Pattern Files unter RCC

Anhand eines konkreten Beispiels soll aufgezeigt werden, wie ein Patternfile auszusehen hat, damit es vom RCC-Algorithmus verwendet werden kann. Tabelle 4.1 zeigt die verwendete Kodierung für das Alphabet mit Punkten und Strichen. Ein jeder Punkt wiederum, wird durch die Folge ON-OFF und ein jeder Strich durch die Folge ON-ON-OFF kodiert. Die endgültige Zeichenkodierung ist in Tabelle 4.2 dargestellt. Nun muß nur noch eine Möglichkeit gefunden werden, Anfang und Ende eines zu lernenden Musters eindeutig zu kennzeichnen. Dies realisiert man durch ein zusätzliches OFF am Ende einer jeden Zeichenfolge.

Der RCC-Algorithmus assoziiert also mit dem letzten Element des Ausgabvektors die Reset-Unit. Hat diese den Wert 1, so interpretiert RCC den nachfolgenden Vektor als Anfangsvektor des nächsten Musters. Beachtet man diese Konvention, so können beliebige Muster für das RCC-Lernverfahren generiert werden.

Eingabeunit	26 Ausgabeunits	Reset-Unit	Codiertes Zeichen
1	0 0 0 ... 0 0 0	0	
0	0 0 0 ... 0 0 0	0	
1	0 0 0 ... 0 0 0	0	
1	0 0 0 ... 0 0 0	0	
0	0 0 0 ... 0 0 0	0	
0	1 0 0 ... 0 0 0	1	A
...	...	...	...

Tabelle 4.3: Representation des Morsekodes im SNNS, wobei die Werte ON und OFF aus der vorherigen Tabelle durch 1 und 0 ersetzt wurden.

Speziell für den Morsecode erhält man dann ein Netz mit einer Eingabeunit und mit 27 Ausgabeunits (vgl. Tbl. 4.3). Das gesamte Pattern-File besteht damit aus 204 Paaren von Eingabe- und Ausgabevektoren.

4.6 Ergebnisse

Getestet wurde der RCC Algoritmus an dem nun schon bekannten Morsekode. Kodiert waren die Testdaten wie oben beschrieben. Zum Minimieren des Fehlers bzw. zum Maximieren der Kovarianz wurden in RCC wieder die drei Lernverfahren Backprop, Quickprop und Rprop verwendet. Der Backpropagation-Algorithmus schnitt auch diesmal wieder am schlechtesten ab. Es ist mir nicht gelungen, mit ihm den Morsekode zu lernen. Mit den anderen beiden Algorithmen gelang es, das Netz zu trainieren, wobei mit dem Quickprop-Algorithmus die mit Abstand kleinsten Netze generiert wurden. Über 5 Testreihen gemittelt, wurden von dem Quickprop-Algorithmus während des Trainings durchschnittlich 14.7 Hidden-Units generiert. Das kleinste Netz bestand aus 9 Hidden-Units. Alle fünf Versuche verliefen erfolgreich, ganz im Gegensatz zu den Versuchen, mit Rprop das Problem zu lösen. Von den fünf Versuchen habe ich 3 abgebrochen, nachdem rund 30 Hidden-Units generiert waren und der Fehler sich praktisch nicht mehr veränderte. Bei den beiden erfolgreichen Versuche generierte der Algorithmus einmal 25 und einmal 27 Hidden-Units. Die Ergebnisse sind in Tabelle 4.4 nochmals zusammengefaßt.

Einige abschließende Bemerkungen zu RCC:

- Um zu überprüfen, ob RCC auch wirklich eine Musterabfolge gelernt hat, oder ob vielleicht doch irgendwie nur eine Zuordnung zwischen einem einzelnen Eingabektor und einem Ausgabevektor gelernt wurde, habe ich einem fertig trainierten Netz nur unvollständige Abfolgen von Eingabevektoren präsentiert. Dabei war zu beobachten, daß die Neuronen blind und ohne Zusammenhang feuerten. Der RCC-Algorithmus lernt also wirklich die zeitliche Abfolge von Mustern!
- Damit es dem Benutzer beim Testen leichter fällt, zu entscheiden, wann eine neue Musterfolge beginnt, wird jeder erste Eingabevektor eines neuen Musters angekündigt.

	Aktivierungsfunktion: Act_Logistic	
	Quickprop	Rprop
	$eta_1 = 0.0001$ $\mu_1 = 2.0$ ν=0.00001 $eta_2 = 0.0009$ $\mu_2 = 2.0$	$\eta_1^- = 0.5$ $\eta_1^+ = 1.2$ $\eta_2^- = 0.5$ $\eta_2^+ = 1.2$
Anzahl der generierten Units	14.7	26
Anzahl der benötigten Lernzyklen	3100	6500

Tabelle 4.4: Ergebnisse für das Training des Morsekodes mit der gewöhnlichen Aktivierungsfunktion Act_Logistic

- Da das Reset-Signal mitgelernt wurde, kann dieses Signal nach dem Training dazu verwendet werden das Netz zurückzusetzen. Das heißt, für einen realen Einsatz brauchte man dann das *reset Array* nicht mehr.

Kapitel 5

Integration von CC und RCC im SNNS

In diesem Kapitel wird beschrieben, wie die beiden Lernverfahren CC und RCC in den SNNS eingebunden sind und wie sie aufgerufen werden. Aus den vorherigen Kapiteln wird bereits ersichtlich, daß die Bedienung der beiden Lernverfahren von der bisher gewohnten Bedienung teilweise stark abweicht.

5.1 Aufruf der Lernverfahren

Zuerst wird beschrieben wie die beiden Lernverfahren aufgerufen werden und was dabei zu beachten ist.

5.1.1 Aufruf von Cascade-Correlation

Das Lernverfahren Cascade-Correlation wird aktiviert, indem man die Lernfunktion CC auswählt. Eine Liste aller zur Verfügung stehenden Lernverfahren erhält man durch Anklicken des Menüpunktes Select learning function.

Vor Beginn des Trainings muß man noch zusätzlich die Update Funktion CC_Order und die Initialisierungs Funktion CC_RandomWeights ausgewählt werden. Vergißt man, eine der beiden zuletzt aufgezählten Funktionen zu aktivieren, so erscheint eine Fehlermeldung und es wird nicht gelernt. Diese Fehlermeldung ist zwar prinzipiell überflüssig, da das Lernen auch ohne Auswahl dieser Funktionen funktioniert, aber die eigene Erfahrung hat gezeigt, daß es zu einem späteren Zeitpunkt leicht zu Fehlern und Mißinterpretationen kommen kann, wenn die falsche Initialisierungs- oder Update-Funktion aktiv ist. Um dies zu vermeiden, müssen vor Beginn des Trainings bereits alle Funktionen korrekt ausgewählt werden.

5.1.2 Aufruf von Recurrent Cascade-Correlation

Bevor RCC gestartet werden kann, sind die gleichen Schritte wie bei CC notwendig. Nur daß diesmal die auszuwählende Lernfunktion $\boxed{\text{RCC}}$ heißt. Entsprechend muß die Update-Funktion $\boxed{\text{RCC\_Order}}$ und die Initialisierungsfunktion $\boxed{\text{RCC\_RandomWeights}}$ aufgerufen werden.

5.2 Aufbau und Belegung des Cascade-Windows

Da Cascade-Correlation eine Art Meta-Algorithmus darstellt, von dem aus wieder eigene Lernverfahren aufgerufen werden, wurde ein zusätzliches Window eingeführt, über das alle Cascade-spezifischen Parameter eingestellt werden können. Dieses Window erscheint, wenn im Hauptmenü der Auswahlpunkt $\boxed{\text{CASCADE}}$ gedrückt wird. Das Hauptmenü wiederum erhält man nach Drücken des $\boxed{\text{XGUI}}$-Buttoms im XGUI-Fenster. Sowohl für den CC-Algorithmus als auch für den RCC-Algorithmus werden alle für das Lernverfahren spezifischen Parameter über dieses Fenster eingestellt. Abbildung 5.1 zeigt das Cascade-Window.

Das Cascade-Fenster ist dreigeteilt und zwar in die Bereiche *Global parameters*, *Candidate parameters* und *Output parameters*. Wie die Namen bereits verdeutlichen, befinden sich unter dem Bereich *Global parameters* die Parameterwerte, die das gesamte Lernverfahren betreffen. Unter dem Bereich *Candidate parameters* bzw. *Output parameters* befinden sich die Werte, die für das Trainieren der Kandidatenzellen bzw. der Ausgabezellen wichtig sind.

Die Auswahl der Aktivierungsfunktion für die Kandidatenzellen bzw. die Hidden-Units kann nicht über die Oberfläche des SNNS geschehen, da diese Neuronen erst während des Trainings dynamisch generiert werden. Die Wahl der gewünschten Aktivierungsfunktion für diese Zellen erfolgt deshalb über das Cascade-Window. Im Gegensatz dazu wird die Auswahl der Aktivierungsfunktion der Ausgabezellen wie üblich über die SNNS-Oberfläche vorgenommen.

Damit es nicht zu ungewollten Inkonsistenzen zwischen der Aktivierungsfunktion der Kandidaten- und Ausgabezellen kommt, wird vor Beginn des Lernens eine Plausibilitätskontrolle durchgeführt, d.h. es wird getestet, ob beide Schichten mit derselben Aktivierungsfunktion initialisiert sind. Ist dies nicht der Fall, so erscheint eine Warnung auf dem Bildschirm. Der Benutzer hat dann die Wahl, das Verfahren abzubrechen oder auch fortzufahren. Auf diese Weise kommt es zu keinerlei Einschränkungen der Funktionalität. Die Erfahrung hat gezeigt, daß durch diese Warnung sehr viele Fehler von vornherein vermieden werden können. Selbstverständlich kann diese Warnung, wie alle anderen vom Kern ausgehenden Meldungen, abgeschaltet werden, so daß der Kern auch weiterhin ohne Oberfläche betrieben werden kann.

5.2.1 Belegung des Cascade-Windows

Die Bedeutung der Parameter des Cascade-Windowss im einzelnen:

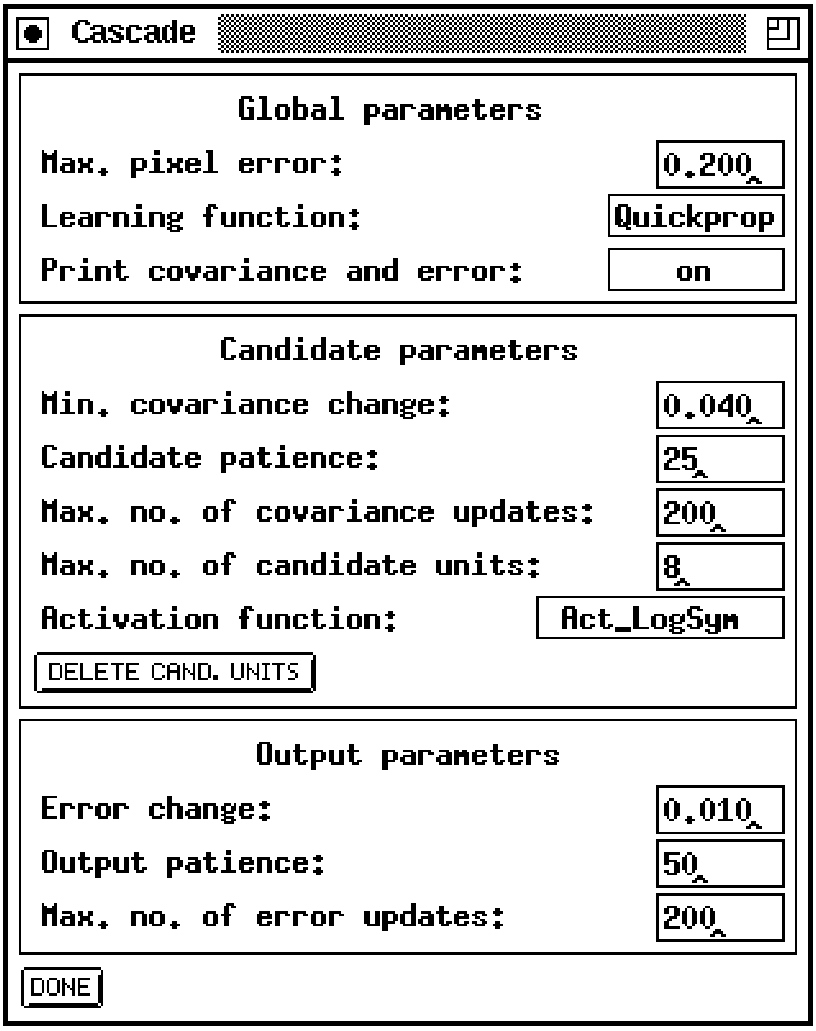

Abbildung 5.1: Das Cascade-Window

- Global parameters:
  - `Max. pixel error:`
    Dieser Wert dient als Abbruchkriterium für das Lernverfahren. Ist der Fehler jeder einzelnen Ausgabezelle kleiner oder gleich dem angegebenen Wert, so endet das Lernen.
  - `Learning function:`
    Über diesen Menüpunkt kann das Lernverfahren ausgewählt werden, mit dem die Kovarianz der Kandidatenzellen maximiert und der Fehler der Ausgabezellen minimiert werden soll. Standardmäßig ist der Quickprop-Algorithmus aktiviert. Klickt man diesen Menüpunkt an, so erscheint ein kleines Pulldown-Menue, über das dann eines der drei Lernverfahren Quickprop, Backprop oder Rprop ausgewählt werden kann.

- `Print covariance and error:`
  Steht dieser Auswahlpunkt auf „on“, so wird nach jedem Update-Schritt der Kandidatenzellen bzw. nach jedem Update-Schritt der Ausgabezellen die Kovarianz bzw. der Fehler angezeigt. Durch das Anklicken von „on“ kann über ein Pulldown-Menue der Wert auf „off“ geschaltet werden. Damit sind alle vom Kern kommenden Meldungen unterbunden. Als einzige Meldung erscheint jetzt noch der verbleibende Restfehler immer dann, wenn das Lernverfahren in die Hauptschleife des SNNS zurückkehrt.

• Candidate parameters:

  - `Min. covariance change:`
    Dieser Wert gibt an, um wieviel Prozent die Kovarianz nach einer gewissen Anzahl von Zyklen mindestens angewachsen sein muß, damit ein weiteres Trainieren der Kandidatenzellen lohnend ist. Wird dieser Prozentsatz nicht erreicht, so geht das Verfahren davon aus, daß ein Maximum erreicht ist, und daß ein weiteres Lernen nichts mehr bringt. Die Kandidatenzelle mit der höchsten Kovarianz wird in ein Hidden Neuron umgewandelt.
  - `Candidate patience:`
    Hier wird die Zyklenzahl eingetragen, nach der überprüft werden soll, ob die oben beschriebene „Min. covariance change“ erreicht wurde. Diese beiden Parameter gehören also eng zusammen. Eine geschickte Wahl der Werte kann das Lernen erheblich beschleunigen. Die Wahl der Parameterwerte ist problemspezifisch.
  - `Max. no. of covariance updates:`
    Die maximale Anzahl von Lernschritten für die Berechnung der Kovarianz. Ist diese Anzahl erreicht, so wird nicht versucht, die Kovarianz weiter zu maximieren, sondern es wird die Kandidatenzelle, die die größte Kovarianz mit der Ausgabeschicht aufweist ausgewählt und in eine Hidden-Unit umgewandelt.
  - `Max no. of candidate units:`
    Hier wird die Anzahl der Kandidatenzellen eingetragen, die erzeugt werden soll, um sie dann parallel zu trainieren. Die Voreinstellung ist hier auf 8 Kandidatenzellen festgelegt. Ein größerer Pool an Kandidatenzellen bringt in der Regel keine signifikante Verbesserung mit sich, verlangsamt aber das Lernen.
  - `Activation function:`
    Dieser Auswahlpunkt ermöglicht die Auswahl verschiedener Aktivierungsfunktionen für die Kandidatenzellen. Zur Auswahl stehen: Act_Logistic, Act_LogSym, Act_Tanh, Act_Identity und Random.

• Output parameters

  - `Min. error change:`
    vgl. „Min. covariance change“
  - `Output patience:`
    vgl. „Covariance patience“
  - `Max. no. of update error updates:`
    vgl. „Max. no. of covariance updates“

Zum Abschluß bleibt noch der Button $\boxed{\text{DELETE ALL CAND. UNITS}}$. Mit diesem Buttom können, z.B. nach dem Lernen, alle Kandidatenzellen gelöscht werden.

5.2.2 Einfluß der Parameter auf das Lernverhalten

Großen Einfluß auf das Lernverhalten von CC haben die drei Parameter, die die Zyklenzahl für die Maximierung der Kovarianz und für die Minimierung des Fehlers festlegen. Bricht man die Maximierung der Kovarianz zu früh ab, so wirkt sich dies negativ auf die anschließend folgende Minimierung des Fehlers aus, da dann die Aktivierung der aus der Kandidatenzelle generierten Hidden-Unit bei einem anliegenden Muster relativ zufällig ausfällt und auch nicht mehr verbessert werden kann. Alle zu ihr führenden Verbindungen werden ja eingefroren. Umgekehrt sollte natürlich auch das Training der Ausgabeschicht nicht zu kurz kommen, da dies wiederum negative Auswirkungen auf die anschließend folgende Maximierung der Kovarianz hat.

Wie diese Parameter am günstigsten zu wählen sind, kann nicht allgemein gesagt werden, da dies von Problem zu Problem verschieden ist. Zusammenfassend kann man aber feststellen, daß es ratsam ist, lieber etwas länger die Kovarianz zu maximieren bzw. den Fehler zu minimieren, da damit die Wahrscheinlichkeit steigt, ein sehr kleines Netz zu generieren, wodurch die Gesamtzeit für das Training des neuronalen Netzes sinkt.

Wie wir gesehen haben, hat die Wahl der Aktivierungsfunktion ebenfalls einen großen Einfluß auf das Konvergenzverhalten eines Netzes. Es kann zwischen den Funktionen Act_Logistic, Act_TanH, Act_Identity, Act_LogSym und Random gewählt werden. Die ersten drei Funktionen sind bereits im SNNS vorhanden und dokumentiert. Neu ist die Aktivierungsfunktion Act_LogSym und Random. Bei Act_LogSym handelt es sich um die bekannte logistische Funktion, die um -0.5 entlang der y-Achse verschoben ist. Der Wertebereich dieser Funktion liegt also zwischen -0.5 und 0.5. Als letzte neue Aktivierungsfunktion bleibt noch Random. Es sei hier aber ausdrücklich darauf hingewiesen, daß es sich hierbei **nicht** um eine Funktion handelt. Die Auswahl dieses Punktes hat zur Folge, daß jede Kandidatenzelle mit einer zufälligen Aktivierungsfunktion aus der Menge Act_Logistic, Act_TanH, Act_Identity oder Act_LogSym initialisiert wird. Weiter sei darauf hingewiesen, daß bei der Auswahl im Cascade-Window nicht zwischen der rekurrenten und der ursprünglichen Form der Aktivierunsfunktionen unterschieden wird. CC und RCC wählen jeweils die für sie korrekte Version aus.

5.3 Die Belegung des Remote-Panels

Die wichtigste Änderung im Remote-Panel zuerst: Die Zahl der Lernzyklen wird bei CC und bei RCC **nicht** über das Remote-Panel eingegeben, sondern über das Cascade-Window (siehe oben). An der Stelle *cycles* wird die maximale Anzahl von verdeckten Zellen eingetragen, die generiert werden soll, bevor das Verfahren stoppt.

Wird im Remote-Panel
die Initialisierungsfunktion CC_RandomWeights oder RCC_RandomWeights ausgewählt, so werden bei der Initialisierung alle Hidden-Units gelöscht und die verbleibenden Ver-

bindungen zwischen der Eingabeschicht und der Ausgabeschicht zufällig initialisiert. Die Kandidatenzellen bleiben unverändert.

Die Eingabe der Lernparameter für die in CC eingebetteten Lernverfahren Quickprop, Backprop und Rprop geschieht wie gewöhnlich über das Remote-Panel. Der ausgewählten Lernfunktion dienen dabei die ersten drei Werte als Lernparameter für das Training der Output Units und die letzten zwei Parameter als Lernparameter für das Training der Kandidatenzellen.

5.3.1 Bedeutung der Lernparameter

Von links nach rechts gelesen haben die Lernparameter folgende Bedeutung:

- Backpropagation:

  1. η_1: Lernparameter, spezifiziert die Schrittweite des Gradientenabstiegs beim Minimieren des Fehlers.
  2. μ_1: Momentumterm, gibt den Anteil der alten Gewichtsänderung an, der zu der neuen Änderung addiert werden soll.
  3. c: Flat Spot Elimination, ein konstanter Wert, der zu der Ableitung der Aktivierungsfunktion hinzuaddiert wird.
  4. η_2: Lernparameter, spezifiziert die Schrittweite des Gradientenanstiegs beim Maximieren der Kovarianz.
  5. μ_2: Momentumterm, gibt den Anteil, der alten Gewichtsänderung an, der zu der neuen Änderung hinzuaddiert werden soll.

  Die verwendete Lernregel hat allgemein folgendes Aussehen:

  $$\Delta w_{ij}(t+1) = \eta S(t) + \mu \Delta w_{ij}(t-1) \tag{5.1}$$

  Die Steigungen $\partial E / \partial w_{ij}$ und $-\partial C / \partial w_{ij}$ werden durch S abgekürzt. Diese abkürzende Schreibweise gilt auch für die noch folgenden Formeln. Es wird nicht zwischen einer Lernfunktion für Ausgabezellen und für Kandidatenzellen unterschieden, da durch einfaches Umdrehen des Vorzeichens der Ableitung sowohl für die Fehlerminimierung als auch für die Kovarianzmaximierung dieselbe Lernfunktion verwendet werden kann.

- Rprop

  1. η_1^-: Schrumpfungsparameter, gibt an, um welchen Faktor der Update-Parameter Δ_{ij} bei der Fehlerminimierung verkleinert werden soll.
  2. η_1^+: Wachstumsparameter, gibt an, um welchen Faktor der Update-Parameter Δ_{ij} bei der Fehlerminimierung anwachsen soll.
  3. Nicht belegt.
  4. η_2^-: Schrumpfungsparameter für Kovarianzmaximierung (vgl. oben)
  5. η_2^+: Wachstumsparameter für Kovarianzmaximierung (vgl. oben)

Bei Rprop werden die Gewichte nach folgender Formel verändert:

$$\Delta w_{ij}(t) = \begin{cases} -\Delta_{ij}(t-1)\eta^{-}, & \text{wenn } S(t)S(t-1) < 0 \\ -\Delta_{ij}(t-1)\eta^{+}, & \text{wenn } S(t) > 0 \text{ und } S(t-1) > 0 \\ \Delta_{ij}(t-1)\eta^{+}, & \text{wenn } S(t) < 0 \text{ und } S(t-1) < 0 \\ 0, & \text{sonst} \end{cases} \tag{5.2}$$

wobei weiter gilt: $\Delta_{ij}(t) = \Delta_{ij}(t-1)\eta^{+/-}$.

- Quickprop

  1. η_1: Lernparameter, spezifiziert die Schrittweite des Gradientenabstiegs beim Minimieren des Fehlers.
  2. μ_1: Maximaler Wachstumsparameter, realisiert einen dynamischen Momentumterm (vgl. Backprop).
  3. ν: Verzögerungsparameter, reduziert alle Gewichte um denselben Faktor und verhindert so ein übermäßiges Anwachsen der Gewichte.
  4. η_2: Lernparameter für Kovarianzmaximierung (vgl. oben).
  5. μ_2: Momentumterm für Kovarianzmaximierung (vgl. oben).

Auch hier sei die verwendete Formel angegeben:

$$\Delta w_{ij}(t) = \begin{cases} \eta S(t), & \text{wenn } \Delta w_{ij}(t-1) = 0 \\ \frac{S(t)}{S(t-1)-S(t)}\Delta w_{ij}(t-1), & \text{wenn } \Delta w_{ij}(t-1) \neq 0 \text{ und } \frac{S(t)}{S(t-1)-S(t)} < \mu \\ \mu\Delta w_{ij}(t-1), & \text{sonst} \end{cases}$$

5.4 Anordnung der Zellen und der Ebenen

Von links nach rechts werden die Schichten wie folgt angeordnet: Input Layer, Hidden Layer, Output Layer, Candidate Layer. Sollte diese Anordnung einmal nicht stimmen, z.B. wenn mit dem BIGNET-Tool Eingabeschicht und Ausgabeschicht untereinander positioniert worden sind, so wird vom Programm die gewünschte Anordnung selbständig hergestellt. Diese starre Anordnung bringt den Vorteil mit sich, daß die Positionen, an denen die neuen Hidden-Units eingeordnet werden sollen, leicht festgestellt werden können. Die Hidden-Units werden immer zwischen Eingabe- und Ausgabeschicht eingefügt. Den benötigten Platz verschafft sich das Programm selbständig. Momentan werden immer fünf Hidden-Neuronen untereinander angeordnet. Die sechste Zelle wird dann wieder neben dem ersten Hidden-Neuron eingefügt. Auf diese Weise bleiben die erzeugten Netze sehr übersichtlich. Die Zahl der verdeckten Zellen, die untereinander angeordnet werden soll, ist in dem File *cc_mac.h* in der Konsten *HIDDEN_LAYER_HEIGHT* abgespeichert. Der Update des Netzes findet erst nach dem Lernen statt. Abb. 5.2 und Abb. 5.3 zeigen das Aussehen eines Netzes vor und nach dem Training.

Es gibt eine Situation, in der das Netz nach einer Veränderung nicht neu gezeichnet wird: Ist das Display-Window des SNNS iconisiert und wird jetzt das Netz verändert, dann wird nach dem enticonisieren das alte und nicht das neue Netz dargestellt. In diesem Fall kann ein Refresh von Hand ausgelöst werden, indem man im Display-Window den SETUP-Button drückt.

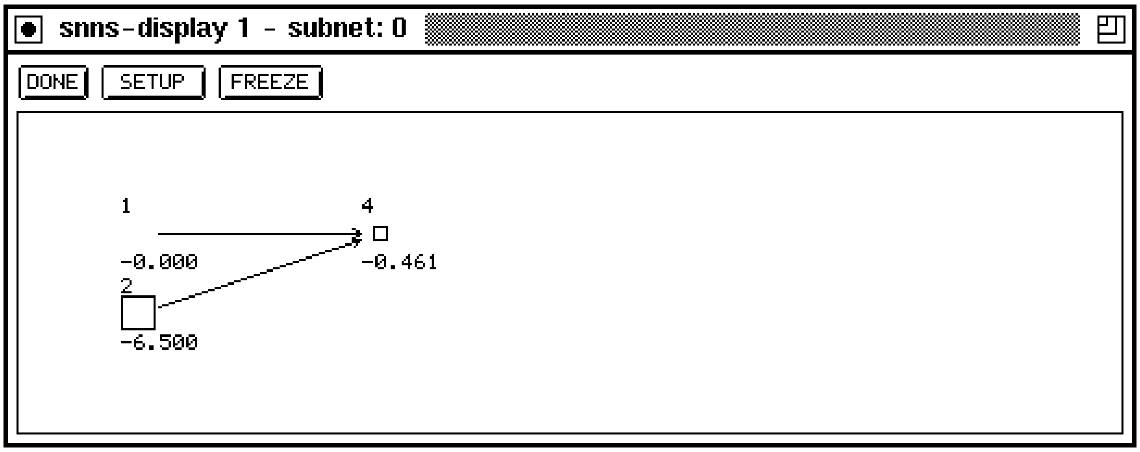

Abbildung 5.2: Ein untrainiertes Netz. Das Netz besteht aus zwei Eingabeneuron und einem Ausgabeneuron

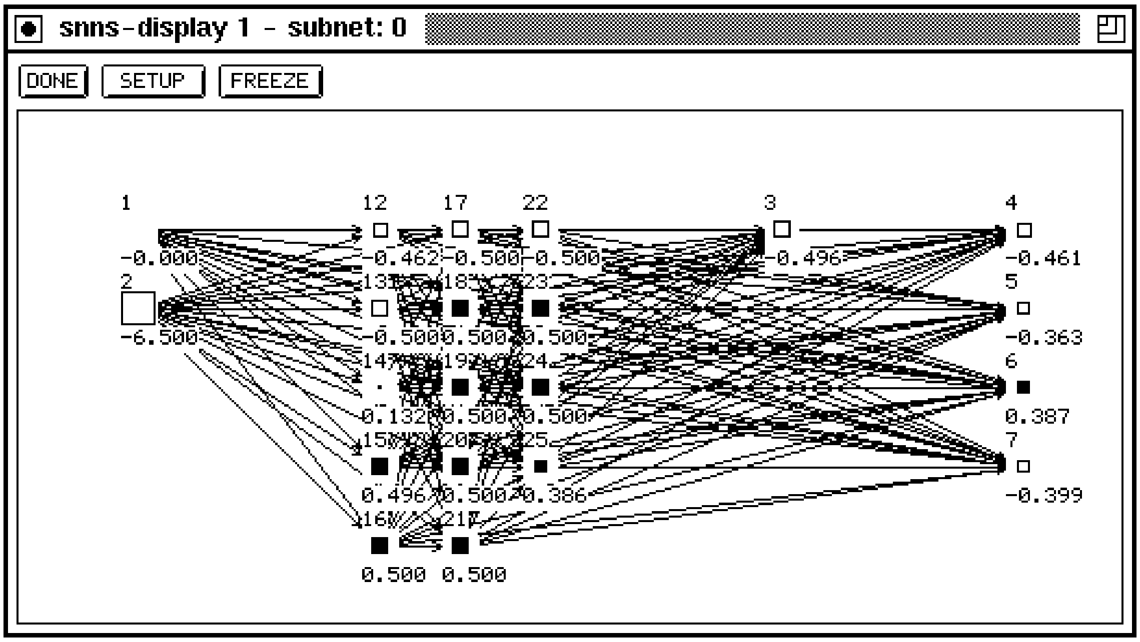

Abbildung 5.3: Das Netz, nachdem das Spiralenproblem gelöst ist. Es besteht nun aus zwei Eingabeneuronen, 14 Hidden-Units einer Ausgabezelle und 4 Kandidatenzellen

Kapitel 6

Dynamischer Nächster-Nachbar-Klassifikator

Das Ziel eines Nächsten-Nachbar-Klassifikators ist es, eine natürliche Aufteilung einer Datenmenge in Gruppen zu finden. Jeder Datenvektor wird hierbei als ein Punkt in einem d-dimensionalen Merkmalsraum aufgefaßt. Geometrisch gesprochen geht man weiter davon aus, daß alle einer bestimmten Klasse angehörenden Daten Puntwolken im Merkmalsraum bilden. Damit sich derartige Punktwolken ausbilden können, müssen alle Muster, die derselben Klasse zugeordnet werden sollen, gewisse Ähnlichkeiten aufweisen. Was man also benötigt, ist ein Ähnlichkeits– bzw. ein Abstandsmaß. In der Regel wird das euklidische Abstandsquadrat $||\vec{x} - \vec{y}||^2$ verwendet.

In einfachen Fällen kann es reichen, wenn eine Klasse durch einen prototypischen Vertreter aus ihr repräsentiert wird. Um ein neues Muster einzuordnen, vergleicht man dieses Muster mit den Repräsentanten aller Klassen. Das neue Muster wird dann der Klasse zugeordnet, zu deren prototypischen Vertreter es am ähnlichsten ist.

Ein auf dem Nächsten-Nachbar-Klassifikator aufbauendes Verfahren funktioniert solange, wie die beobachteten Klassen tatsächlich gewisse Gemeinsamkeiten haben. Ein derartiger Klassifikator ist also ungeeignet, um zum Beispiel das Spiralenproblem zu lösen, bei dem irgendwelche Eingabekoordinaten einer von zwei Spiralen bzw. Klassen zugeordnet werden sollen. Dafür eignen sich diese Verfahren aber sehr gut, wenn es darum geht Buchstaben oder Zahlen zu klassifizieren.

In den nachfolgend Abschnitten werden zunächst einige Verfahren beschrieben, die auf dem Prinzip des Nächsten-Nachbar-Klassifikators aufbauen und bei denen die Zahl der Referenzvektoren fest ist [Schuermann et al. 92], [Duda 73] [Andrews 72]. Anschließend werde ich eine Variante herleiten, bei der sich die Zahl der Referenzvektoren während des Trainings dynamisch verändert.

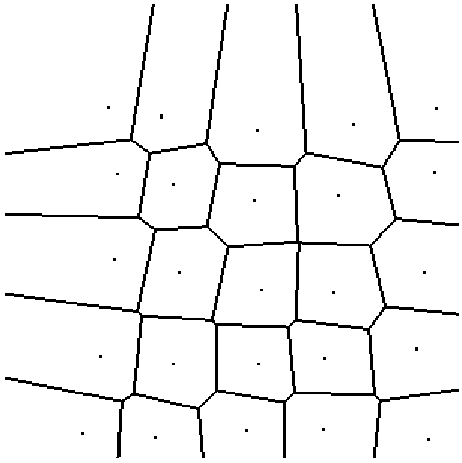

Abbildung 6.1: Aufteilung der Ebene in Cluster durch einen Nächsten-Nachbar-Klassifikator

6.1 Nächster-Nachbar-Klassifikator

Bei diesem ersten Verfahren dient die gesamte Trainingsmenge als Referenzmenge. Jedes Element $\vec{x}_r$ der Trainingsmenge wird mit seiner zugehörigen Klasseninformation k_r als Tupel $(\vec{x}_r, k_r)$ abgespeichert. Um ein unbekanntes Muster $\vec{x}$ zu klassifizieren, durchläuft man einfach die Menge von Referenzvektoren und ordnet $\vec{x}$ der Klasse k_r zu, bei der der Abstand $||\vec{x} - \vec{x}_r||^2$ minimal ist.

Nach Abspeichern der Trainingsvektoren mit der zugehörigen Klasseninformation ist dieses Verfahren sofort einsatzbereit. Dieser Algorithmus funktioniert umso besser, je größer die Trainingsmenge ist. Ein Vorteil dieses Verfahrens ist es, daß die Lernstichprobe durch Hinzufügen eines neuen Tupels jederzeit ergänzt und erweitert werden kann. Der Nächste-Nachbar-Klassifikator teilt wie in Abb. 6.1 aufgezeigt den Merkmalsraum in Parzellen, deren Grenzen durch Ausschnitte aus der Mittelebene zwischen jeweils benachbarten Referenzvektoren gebildet werden. Jede Parzelle wird also durch genau einen Referenzvektor gebildet. Die Klassenzugehörigkeit einer Parzelle wird durch den Referenzvektor bestimmt. Der Merkmalsraum wird auf diese Weise vollständig aufgeteilt.

Ein auf diesem Verfahren aufbauender Lernalgorithmus, der seine Netzwerktopologie selbstständig bestimmt, könnte folgendes Aussehen haben:

1. Führe für jede Klasse aus der Menge der Trainingsvektoren einen beliebigen Vektor als Referenzvektor ein. Speichere diese Vektoren in einer Liste ab.
2. Vergleiche der Reihe nach alle Trainingsvektoren mit den Referenzvektoren aus der Liste.
   (a) Wird ein Trainingsvektor der richtigen Klasse zugeordnet, dann gehe wieder zu Punkt 2, ansonsten füge diesen nicht erkannten Trainingsvektor als neuen Referenzvektor der Vektorliste hinzu und gehe anschließend wieder zu Punkt zwei.

Wiederhole diesen Vorgang solange, bis alle Vektoren klassifiziert sind.

Die Nachteile dieses Verfahrens sind offensichtlich: Zum einen wird eine sehr große Menge an Speicherplatz benötigt – und zwar sowohl für das Training als auch danach für das fertig trainierte Netz –, zum anderen wird ein derartiger Algorithmus relativ langsam sein, da er jedesmal die gesamte Referenzmenge durchlaufen muß. Ein weiterer Nachteil ist, daß das Netz nachtrainiert werden muß, da nicht garantiert werden kann, daß durch neu eingeführte Vektoren bereits richtig klassifizierte Muster nicht evtl. der falschen Klasse zugeordnet werden.

Dieses soeben beschriebene Vorgehen gehört eigentlich bereits in die Klasse der Verfahren, die versuchen den Merkmalsraum mit Hilfe einer Stützwert-Approximation anzunähern. Was genau unter der Stützwert-Approximation zu verstehen ist wird im nächsten Abschnitt beschrieben.

6.2 Stützwert-Approximation

Bei der Stützwert-Approximation versucht man eine Funktion über dem Merkmalsraum, deren Verlauf nur an wenigen Stellen, – den Stützstellen – bekannt ist, durch eine Funktionsapproximation anzunähern. Dabei macht man sich folgenden Satz aus der Numerik zu Hilfe:

> Jede Funktion $f(x)$ läßt sich mit Hilfe der Funktionen $f_n(x)$ darstellen, wenn diese ein Orthogonal- oder Orthonormalsystem bilden.

Die Stützstellen der zu approximierenden Funktion sind die Referenzvektoren. Dem Funktionswert an dieser Stelle entspricht die Klassenbezeichnung, der der Vektor angehört. Man verwendet häufig drei Arten von Stützwertfunktionen:

1. Polyeder-Prisma:
   Innerhalb einer Parzelle hat man einen festen Funktionswert.
2. Zylinder:
   Innerhalb eines kreisförmigen Nachbarschaftsbereich um den Referenzvektor hat man einen festen Funktionswert.
3. Gaußglocken:
   Abhängig vom Abstand eines Musters vom Referenzvektor nimmt der Funktionswert ab.

6.2.1 Polyeder-Prisma als Approximationsfunktion.

Das Polyeder-Prisma stellt den einfachsten Fall dar. Diese Funktion ist gegeben durch:

$$f_r(\vec{x}) = \begin{cases} k_r & \vec{x} \text{ liegt innerhalb der zu } \vec{x}_r \text{ gehörenden Parzelle} \\ 0 & \text{sonst.} \end{cases} \tag{6.1}$$

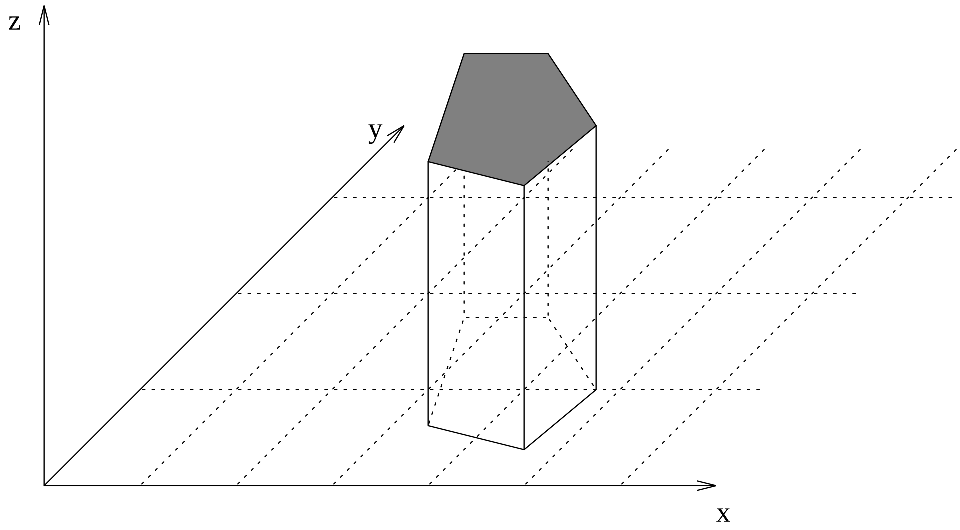

Abbildung 6.2: Ein Polyeder-Prisma als Approximationsfunktion

Da die Funktionen f_r alle orthogonal zueinander sind, kann durch Addition dieser Funktionen die zu approximierende Funktion f gebildet werden: $f = \sum_r f_r(\vec{x})$. Abhängig von der Verteilung der Referenzvektoren können die einzelnen Parzellen unterschiedlich groß sein. Bild 6.2 veranschaulicht das Aussehen dieser Funktionen graphisch.

6.2.2 Zylinder als Approximationsfunktion

Bei diesem Verfahren werden nicht mehr wie oben beschrieben Parzellen gebildet, in denen der Stützwert k_r konstant ist, sondern es wird um den Referenzvektor eine Kugel mit Radius R gelegt, innerhalb der der Stützwert konstant ist (vgl. 6.3):

$$f_r(\vec{x}) = \begin{cases} k_r & \vec{x} \text{ liegt innerhalb der Nachbarschaftskugel mitRadius } R. \\ 0 & \text{sonst.} \end{cases} \tag{6.2}$$

Das bekannteste auf diesem Prinzip aufbauende Verfahren ist das Restricted-Coulomb-Energy Netzwerk. Die Bildung der Referenzvektoren funktioniert hier nach dem Clusterverfahren. Es wird nicht mehr die gesamte Lernstichprobe als Referenzmenge übernommen, sondern nur noch ein kleiner Teil. Tritt der Konfliktfall auf, daß ein Muster einem falschen Vektor zugeordnet wird, so werden die Radien nach bestimmten Regeln verändert. Ein neuer Referenzvektor wird immer dann eingeführt, wenn ein Muster keinem der bereits vorhandenen Referenzvektoren zugeordnet werden kann.

Dieses Verfahren unterscheidet sich von dem vorherigen Verfahren vor allem dadurch, daß die einzelnen Referenzvektoren keinen unendlichen Einzugsbereich mehr haben, sondern daß sich ihr Definitionsbereich auf eine um den Referenzvektor zentrierte Kugel beschränkt. Dies ist auch der Grund dafür, daß ein derartiges Netz nicht immer in der Lage ist, alle Vektoren eindeutig zuzuordnen. Dieser Fall wird beim Training dazu ausgenutzt einen neuen Referenzvektor einzuführen.

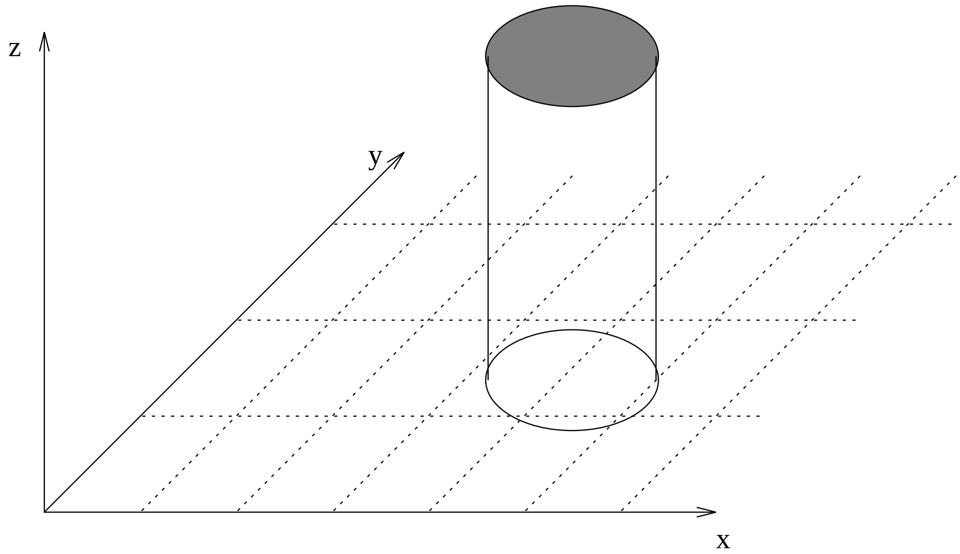

Abbildung 6.3: Ein Zylinder als Approximationsfunktion

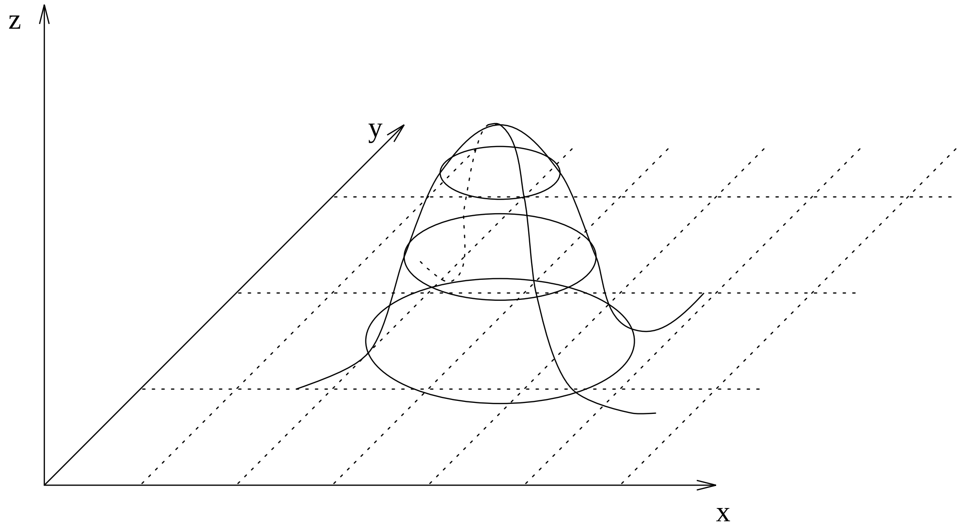

Abbildung 6.4: Eine Gausglocke als Approximationsfunktion

6.2.3 Gauß-Glocken als Approximationsfunktionen

Bei den beiden oben beschriebenen Verfahren bekommt man eine unstetige Approximation der im Merkmalsraum anzunähernden Funktion. Dies liegt ganz offensichtlich an der Art der verwendeten Funktionen. Eine glatte Approximation erhält man dagegen, wenn man zur Annäherung stetige Funktionen verwendet. Eine Möglichkeit hiezu sind die Gaußglocken (vgl. 6.4). Ein hierauf aufsetzendes Verfahren sind die sogenannten Radial-Basis-Funktionen. Dieser Ansatz ist bereits im Rahmen einer Diplomarbeit im SNNS verwirklicht worden. Ich verweise hierzu auf [Vogt 92].

6.3 Dynamischer Nächster-Nachbar-Klassifikator

Bei meinen Untersuchungen von Algorithmen, die auf dem Prinzip des Nächsten-Nachbar-Klassifikator aufbauen, habe ich mich auf die Stützwertapproximation mit Hilfe von Polyeder-Prismen beschränkt. Der Vorteil dieser Stützwertfunktion gegenüber einem Zylinder-Prisma ist der, daß die Parzellen ein beliebiges Aussehen annehmen können. Dies kann z.B. dann von Vorteil sein, wenn die Elemente einer Klasse innerhalb einer Ellipse liegen. Mit Kreisen hat man nun die Möglichkeit, entweder einen die Ellipse einschließenden Kreis zu bilden, was sehr leicht zu Konflikten mit benachbarten Klassen führen kann, oder aber man versucht die Ellipse mit mehreren kleinen Kreisen anzunähern. Prinziell hatte ich daher die Hoffnung, daß eine Approximation mit Polyeder-Prismen besser gelingt als mit Zylinder-Prismen. Einen großen Nachteil bringt aber auch die Zerlegung einer Fläche in Prismen mit sich: Wenn sich im Grenzbereich zwischen zwei Klassen der Einzugsbereich von Referenzvektoren überschneidet, so führt dies hier zu einer Anhäufung von Vektoren, die die Klassengrenze markieren. Im schlimmsten Fall wird für jedes Trainingsmuster ein eigener Referenzvektor eingeführt.

Die ersten Fragen bei diesem Verfahren waren, welche Vektoren zu Beginn des Lernens als Referenzvektoren ausgewählt werden und wieviele Vektoren gewählt werden. Als besonders gut geeignete Startvektoren stellten sich hierbei die Erwartungswertvektoren der einzelnen Klassen heraus, wodurch auch gleichzeitig die Anzahl der Startvektoren festgelegt war. Anschaulich gesprochen zeigen die Erwartungswertvektoren auf den Schwerpunkt der Punktwolken, die die verschiedenen Klassen repräsentieren.

Als Ähnlichkeitsmaß wird das euklidische Abstandsquadrat verwendet:

$$||\vec{x} - \vec{y}||^2 = ||\vec{x}||^2 + ||\vec{y}||^2 - 2(\vec{x}, \vec{y}) \tag{6.3}$$

Sind die Vektoren $\vec{x}$ und $\vec{y}$ normiert, so beschränkt sich die Bestimmung des Abstands zwischen diesen zwei Vektoren auf die Bildung des skalaren Produkts $(\vec{x}, \vec{y})$.

Der erste getestete Algorithmus hatte dann folgende Form:

1. Lade die Trainingsdaten, normiere diese und berechne anschließend für jede Klasse den Mittelwertvektor.
2. Ordne alle Trainingsdaten einem der Referenzvektoren zu.
3. Bilde aus allen falsch zugeordneten Trainingsdaten für jede Klasse erneut den Mittelwert und füge diese neuen Vektoren dem Netz hinzu. Gehe wieder zu Punkt 2.

Der Erfolg dieser Vorgehensweise war sehr bescheiden. Ein Grund dafür wird in Abb. 6.5 graphisch veranschaulicht. Die graue, die weiße und die schwarze Fläche sind Bereiche, in denen Vektoren ein und derselben Klasse A liegen sollen. Bildet man über all diese Vektoren den Mittelwert, so wird dieser Mittelwertvektor wie eingezeichnet liegen. Werden nun die in der weißen Fläche liegenden Vektoren richtig zugeordnet, dann wird aus den in der grauen und in der schwarzen Fläche liegenden Vektoren erneut der Mittelwert gebildet. Wie man aber leicht sieht, wird dieser neue Vektor wieder ganz in der Nähe des alten Vektors liegen und dem Netz somit keine neuen Informationen liefern.

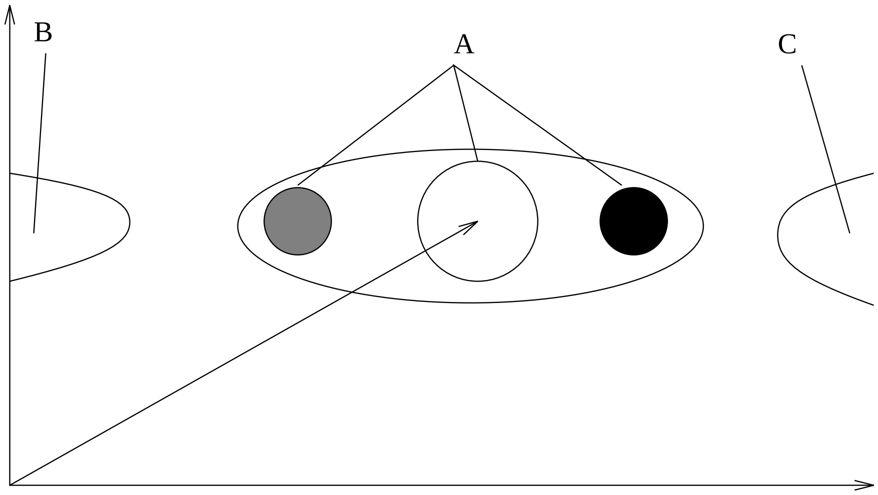

Abbildung 6.5: Mögliche Klassenbildung

Was man also bräuchte wären Gemeinsamkeiten und Unterschiede nicht erkannter Vektoren einer Klasse, um so ein neues Cluster bilden zu können. Abbildung 6.5 gibt auch einen Hinweis hierauf, worin diese Gemeinsamkeiten und Unterschiede bestehen könnten: Den in dem grauen Bereich liegenden Daten ist allen gemeinsam, daß sie fälschlicherweise der Klasse B zugeordnet wurden. Es ist also naheliegend über alle Vektoren einer Klasse A, die der verkehrten Klasse B zugeordnet wurden, den Mittelwertvektor zu bilden. Diese neuen Vektoren werden dann als neue Referenzvektoren eingeführt.

Getestet wurde diese Methode anhand einer Klassifikationsaufgabe, bei der es darum ging handgeschriebene Ziffern zu unterscheiden. Die Ziffern haben eine Größe von 16×16 Pixel. Außerdem sind sie in Breite und Höhe normiert. Die Zahl der zu erkennenden Ziffern wurde bei 100 beginnend bis auf 500 Ziffern erhöht. Dies stellte die obere Grenze der mit diesem Verfahren noch korrekt erkannten Ziffern dar. Es gelang nicht größere Trainingmengen vollständig zu klassifizieren. Außerdem war die Zahl der eingeführten Referenzvektoren für die 500 richtig zugeordnete Ziffern mit über hundert sehr hoch. Die sehr hohe Zahl von Referenzvektoren ist unter anderem damit zu erklären, daß es in den Randbereichen zwischen zwei Klassen zu Überschneidungen kommt. Jeder neu eingeführte Vektor zieht bereits korrekt zugeordnete Muster an, so daß es hier zu einer sehr starken Anhäufung von Vektoren kommt. Geschickt wäre es also, wenn die neuen Vektoren sich vom Zentrum einer Klasse langsam den Randbereichen dieser Klasse annäherten.

Die einzige Möglichkeit, um aus diesem Dilemma herauszukommen war es, die Referenzvektoren nicht mehr länger als starre und nicht bewegbare Stützpunkte anzusehen. Für jedes falsch klassifizierte Muster werden nun zwei Vektoren trainiert bzw. bewegt:

1. Der Referenzvektor, aus der Klasse des falsch klassifizierten Musters, der diesem am ähnlichsten ist, wird ein Stückchen auf dieses Muster zubewegt.

2. Der Referenzvektor, der ein Muster falsch klassifiziert, wird ein Stückchen von diesem wegbewegt.

Das Training der Referenzvektoren erfolgt auf dieselbe Weise wie das Training der Kohonenschicht beim Counterpropagation-Netzwerk. Die Lernregel lautet:

$$w_{ij} = w_{ij} + \eta(o_i - w_{ij}), \tag{6.4}$$

wobei w_{ij} das Gewicht der Verbindung zwischen der Ausgabe o_i einer Eingabezelle i und einer Ausgabezelle j ist. η ist der Lernparameter. Je nachdem, ob sich ein Vektor auf ein Muster zubewegen oder von ihm entfernen soll, muß η größer oder kleiner Null gewählt werden.

Nach mehreren erfolglosen Versuchen hat sich dann folgender Algorithmus als gut geeignet herausgestellt:

1. Lade die Trainingsdaten, normiere diese und berechne anschließend für jede Klasse den Mittelwertvektor. Initialisiere mit diesen Vektoren das Netz.
2. Ordne alle Trainingsdaten der Reihe nach einem der Referenzvektoren zu. Wird dabei ein Trainingsvektor $\vec{t}$ aus Klasse A der Klasse B zugeordnet, so mache folgendes:
   (a) Bewege den Referenzvektor aus der Klasse A, der dem Trainingsvektor am ähnlichsten ist ein Stückchen auf diesen zu.
   (b) Entferne den Referenzvektor, der den Trainingsvektor fälschlicherweise angezogen hat um einen gewissen Betrag von diesem.

   Wiederhole diesen Vorgang solange, bis sich die Zahl der korrekt erkannten Vektoren nicht mehr erhöhen läßt.
3. Bilde nun über alle Trainingsvektoren einer Klasse A, die einer falschen Klasse B zugeordnet wurden, den Mittelwert. Wähle für jede Klasse aus diesen Mittelwertvektoren einen neuen Vektor als Referenzpunkt aus und füge diesen dem Netz hinzu. Gehe nun wieder zu Punkt 2.

6.4 Abbildung des Algorithmus auf ein neuronales Netz

Die Abbildung dieses Algorithmus auf ein neuronales Netz ist sehr einfach: Das Netz besteht aus zwei Schichten, einer Eingabeschicht und einer Ausgabeschicht. Die Eingabeschicht unterscheidet sich durch nichts von der von den feedforward Netzen her bekannten Eingabeschicht. Die Ausgabeschicht dagegen stellt eine *winner-takes-all*-Schicht dar. Ein Eingabemuster wird der am stärksten aktivierten Ausgabezelle zugeordnet. Da jede Ausgabezelle wiederum eindeutig einer bestimmten Klasse zugeordnet werden kann, ist die Klassifizierung eindeutig.

6.4.1 Was berechnen die Neuronen?

Bei herkömmlichen feedforward Netzen berechnet ein Neuronen j das skalares Produkt net_j aus der Ausgabe der Vorgängerzellen i und dem von diesen Zellen zu ihm führenden Gewichten w_{ij}. Danach wird auf dieses skalare Produkt eine Aktivierungsfunktion angewendet. In einer winner-takes-all-Schicht entfällt genau dieser letzte Schritt. Dadurch erreicht man, daß jedes Neuron der Ausgabeschicht stellvertretend für einen Referenzvektor steht und den Abstand dieses Vektors zu einem momentan anliegenden Muster anzeigt. In den Gewichten zwischen der Eingabeschicht und der Ausgabeschicht werden die Referenzvektoren abgespeichert. Die einzigste Vorraussetzung die erfüllt sein muß ist die, daß alle Vektoren normiert sind, da es sonst zu Verzerrungen beim Messen der Abstände kommt. Als Abstandsmaß bekommt man so den Cosinus zwischen den beiden Vektoren:

$$\cos\alpha = net_j = \sum_i w_{ij} o_i \tag{6.5}$$

net_j ist umso größer, je ähnlicher sich Eingabevektor und Referenzvektor sind.

6.5 Integration im SNNS

Die Integration dieses Lernverfahrens in den SNNS war, abgesehen von den bereits bei CC auftretenden Schwierigkeiten, problemlos. Das Ausgangsnetz, d.h. das Netz vor dem Training, besteht aus einer Eingabe- und einer Ausgabeschicht. Die Eingabeschicht enthält wie üblich das zu klassifizierende Muster. In der Ausgabeschicht ist die Klasse abgespeichert, der das momentan anliegende Muster zugeordnet werden soll. Die Ausgabezelle ist lediglich ein Platzhalter, damit das Patternfile korrekt geladen werden kann, da in diesem File die Länge des Ausgabevektors mit eins angegeben ist. Für das Lernen wird diese Zelle nicht mehr benötigt. Das schematische Aussehen eines trainierten Netzes und die Anordnung der Hidden-Units mit ihrer entsprechenden Klassenzugehörigkeit ist in Abb. 6.6 dargestellt.

Da die einzelnen Eingabevektoren auf die Länge eins normiert sind, ist beim normalen Durchlaufen der Muster im SNNS nicht viel zu sehen. Der SNNS bietet im Setup-Window die Möglichkeit die Größe des Rechtecks, das die Aktivität einer Unit darstellt, zu skalieren. Dazu muß bei dem Menüpunkt `units scale:` der Schiebeschalter ganz nach links bewegt werden.

Aus der Anordnung der einzelnen Hidden-Units läßt sich ihre Klassenanordnung eindeutig bestimmen (vgl. 6.6). Es gibt aber noch eine zweite Möglichkeit um festzustellen welche Unit welcher Klasse angehört: Nach Anklicken der gewünschten Hidden-Unit mit der linken Maustaste erscheint an der Stelle, an der normalerweise der Bias abgespeichert ist, ihre Klasse. Ferner zeigt die Ausgabezelle die Klasse an, der ein anliegendes Muster angehört.

6.5.1 Die Belegung des Remote Panels

Ähnlich wie bei den beiden vorherigen Lernverfahren wird an der Stelle `CYCLE` eingetragen, wieviele Hidden-Units oder auch Klassifikationszellen pro Klasse maximal generiert werden sollen, bevor das Verfahren stoppt.

Die fünf Lernparameter sind wie folgt belegt:

1. Lernparameter η^+:
   Gibt an, wie schnell sich ein Referenzvektor einem nicht erkannten Eingabevektor

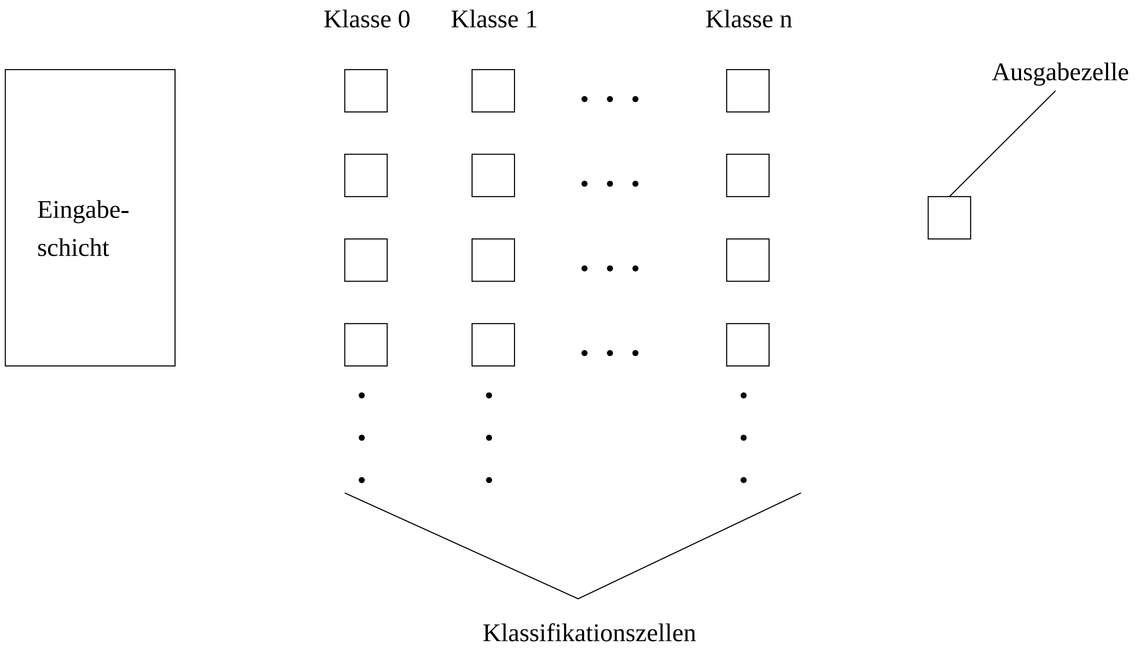

Abbildung 6.6: Schematische Darstellung eines mit DMDC trainierten Netzes.

annähern soll. Bewegt wird derjenige Referenzvektor, der aus der selben Klasse wie der Eingabevektor ist und der diesem am ähnlichsten ist.

2. Lernparameter η^-:
   Gibt an, wie schnell sich derjenige Referenzvektor, dem ein anliegendes Muster irrtümlicher Weise zugeordnet wurde, von diesem entfernen soll.

3. Lernparameter:
   Gibt die maximale Anzahl von Schleifendurchläufen an, bevor das Training der Referenzvektoren abgebrochen wird, und für die einzelnen Klassen neue Units generiert werden.

4. Lernparameter:
   Gibt die Zahl an, um die die falsch klassifizierten Units abnehmen müssen, damit sich ein weiteres Training der Referenzvektoren lohnt.

5. Lernparameter:
   Anzahl der Schleifendurchläufe, nach der jeweils überprüft wird, ob die Menge der falsch klassifizierten Zellen um die im Lernparameter 4 angegebene Zahl, gesunken ist.

Die erste und der zweite Lernparameter entsprechen η aus der Formel 6.4. Die letzten drei Parameter dienen dazu die Trainingsdauer festzulegen.

Klasse	Vek_1	Vek_2	Vek_3	Vek_4	Vek_5	Vek_6	Summe
0	861	76	7	4	52		1000
1	882	76	4	38			1000
2	592	318	25	39	3	23	1000
3	644	291	7	16	42		1000
4	400	399	80	5	109	7	1000
5	373	556	44	27			1000
6	760	66	114	60			1000
7	933	26	18	23			1000
8	531	415	5	10	37	2	1000
9	412	442	118	13	15		1000

Tabelle 6.1: Zuordnung der 10000 Trainingsdaten, zu den 49 Referenzvektoren.

6.6 Ergebnisse

Trainiert und getestet wurde der zuletzt beschriebene Algorithmus mit den nun schon bekannten 10000 Trainingsziffern. Nachdem diese Ziffern vollständig klassifiziert waren, wurde die Generalisierungsfähigkeit des generierten Netzes mit weiteren 10000 Testdaten untersucht.

Um alle 10000 Ziffern der korrekten Klasse zuzuordnen, benötigte dieser Algorithmus 142 Epochen. Unter einer Epoche verstehe ich einen einmaligen Durchlauf über alle Trainingsdaten. Im Verlauf des Trainings wurden 49 Vektoren generiert. Tabelle 6.1 zeigt, wieviele Trainingsziffern nach dem Lernen jedem einzelnen Vektor zugeordnet werden.

Von den 10000 dem Netz zum Testen dargebotenen Ziffern ordnete es 9578 Ziffern der korrekten Klasse zu. Das entspricht 95,78%. Die Erkennungsrate dieses Netzes ist also etwas schlechter als bei dem mit Cascade-Correlation trainierten Netz. Dafür ist das Training um den Faktor 15 schneller. Die Zuordnung der Trainingsvektoren zu den richtigen und den falschen Referenzvektoren ist in Tabelle 6.2 und in Tabelle 6.3 aufgezeigt. Die erste Tabelle enthält die richtig und die zweite Tabelle die falsch zugeordneten Vektoren.

Den Tabellen kann man entnehmen, daß die Trainingsdaten und die Testdaten ungefähr gleich verteilt sind. Weiter erkennt man in Tabelle 6.3, daß zwei Vektoren überhaupt kein Muster zugeordnet wird.

In Abbildung 6.7 und 6.8 sind die Referenzvektoren, die im Verlauf des Trainings generiert wurden, abgebildet. Die eigentlich normierten Vektoren sind mit einem konstanten Faktor multipliziert worden, damit sie besser sichtbar sind.

Klasse	Vek_1	Vek_2	Vek_3	Vek_4	Vek_5	Vek_6	Summe
0	811	74	6	0	60		951
1	855	82	3	39			979
2	579	288	23	34	0	26	950
3	627	273	8	14	41		963
4	369	366	78	6	131	15	965
5	363	526	38	24			951
6	749	52	121	54			976
7	923	18	7	16			964
8	491	408	3	8	35	2	947
9	368	436	98	15	15		932
							$\sum = 9578$

Tabelle 6.2: Verteilung der korrekt zugeordneten Testdaten zu den 49 Referenzvektoren.

Klasse	Vek_1	Vek_2	Vek_3	Vek_4	Vek_5	Vek_6	Summe
0	31	1	1	4	3		40
1	34	1	3	2			40
2	15	9	1	1	1	3	30
3	34	9	6	7	0		56
4	28	8	1	3	3	4	47
5	12	23	3	1			39
6	13	2	4	5			24
7	16	1	3	3			23
8	25	29	1	1	3	2	61
9	24	22	5	6	5		62
							$\sum = 422$

Tabelle 6.3: Verteilung der falsch zugeordneten Testdaten zu den 49 Referenzvektoren.

6.7 Learning Vector Quantization (LVQ)

Nach der Implementierung meines Algorithmus habe ich erfahren, daß es bereits ein Lernverfahren gibt, das auf denselben Ideen aufbaut. Die Bezeichnung dieses Algorithmus ist LVQ [Kohonen et al. 91]. Es gibt drei Versionen LVQ1, LVQ2 und LVQ3 dieses Verfahrens. Nachfolgend beschreibe den Trainingsablauf des Verfahrens LVQ2. Dieser Algorithmus kommt dem von mir implementierten Verfahren am nächsten:

1. Führe eine feste Zahl n von Referenzvektoren ein. Initialisiere diese Vektoren z.B. mit den ersten n Vektoren aus der Menge der Trainingsdaten.
2. Ordne alle Trainingsdaten der Reihe nach einem der Referenzvektoren zu. Wird dabei ein Trainingsvektor $\vec{t}$ aus Klasse A der Klasse B zugeordnet, und liegt $\vec{t}$ in einem „Fenster " zwischen den Referenzvektoren der Klasse A und B, so mache folgendes:
   (a) Bewege den Referenzvektor aus der Klasse A, der dem Trainingsvektor am ähnlichsten ist ein Stückchen auf diesen zu.
   (b) Entferne den Referenzvektor, der den Trainingsvektor fälschlicherweise angezogen hat um einen gewissen Betrag von diesem.

   Gehe nun wieder zu Punkt 2.

Das Training der Referenzvektoren erfolgt nach der Formel 6.4. Wie man sieht, realisiert das DMDC-Verfahren eine Art dynamisches LVQ. Beide Verfahren unterscheiden sich in zwei Punkten:

1. Bei LVQ steht die Zahl der Referenzvektoren von Anfang an fest. Bei DMDC wird ihre Anzahl im Verlauf des Lernens dynamisch vergrößert.
2. Damit in LVQ2 die Referenzvektoren trainiert werden, muß der falsch zugeordnete Trainingsvektor in einem Fenster zwischen den beiden nächsten Referenzvektoren liegen. Ein derartiges Fenster existiert bei DMDC nicht.

6.8 Ausblick

Um die Fähigkeiten dieses Verfahrens objektiver abschätzen zu können, müßte es noch mit anderen Aufgabenstellungen aus dem Gebiebt der Mustererkennung konfrontiert werden. Es ist immerhin denkbar, daß die Verteilung der dargebotenen Ziffern im Merkmalsraum zufällig so war, daß sie diesem Verfahren sehr zugute kam. Das könnte z.B. dann der Fall sein, wenn die Randbereiche zwischen den verschiedenen Klassen nur sehr dünn besetzt sind, so daß die offensichtlichen Nachteile dieses Verfahrens in den Klassenrandbereichen nicht voll zum tragen kommen. Weiterhin ist zu untersuchen, wie sich das Generalisierungsverhalten des Netzes mit der Zahl der Referenzvektoren verändert. Auch Vergleiche mit ähnlichen Lernverfahren wie z.B. dem *Restricted-Coulomb-Energy*-Netzen oder LVQ-Netzen wären sehr interessant.

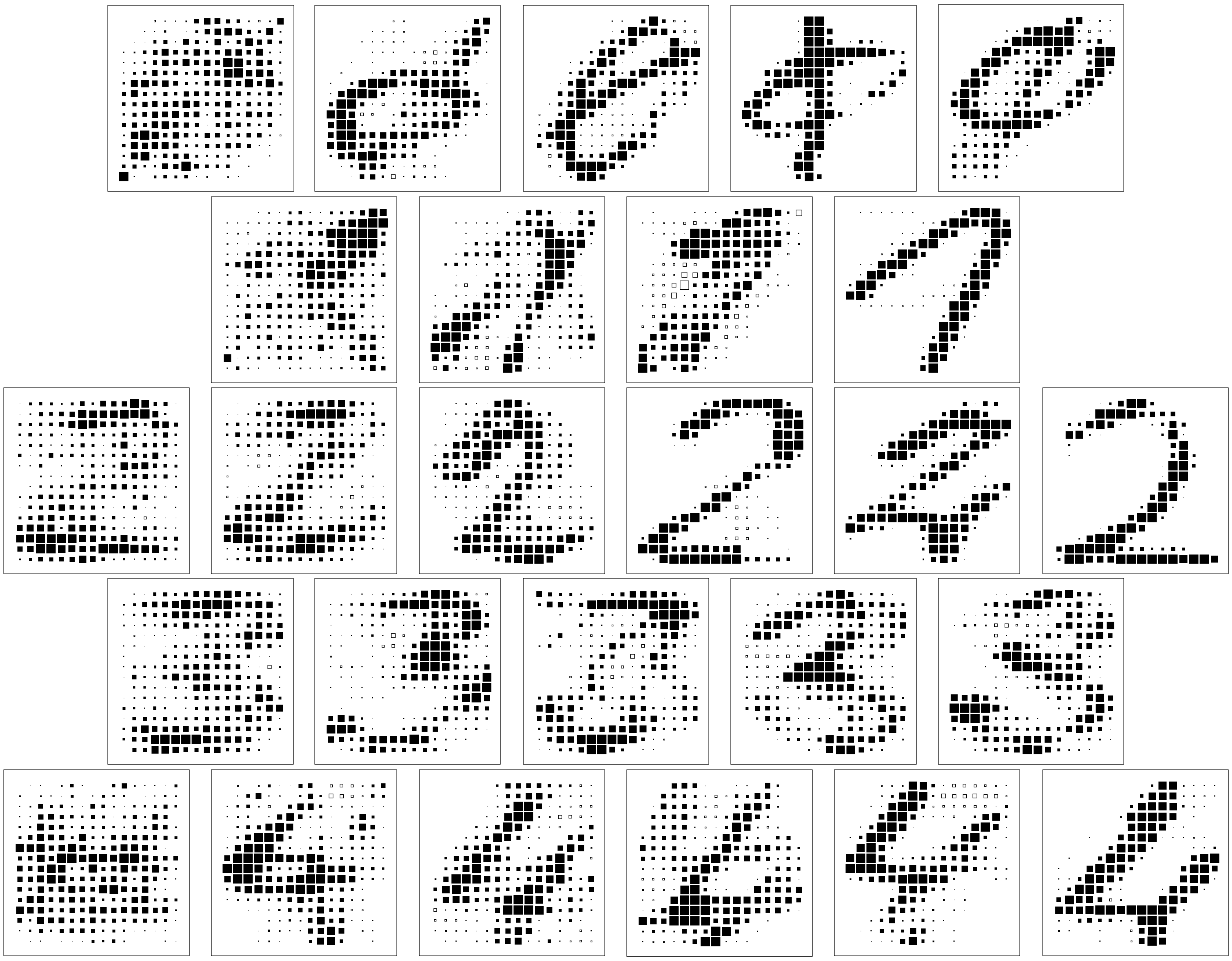

Abbildung 6.7: Referenzvektoren für die Ziffern Null bis vier

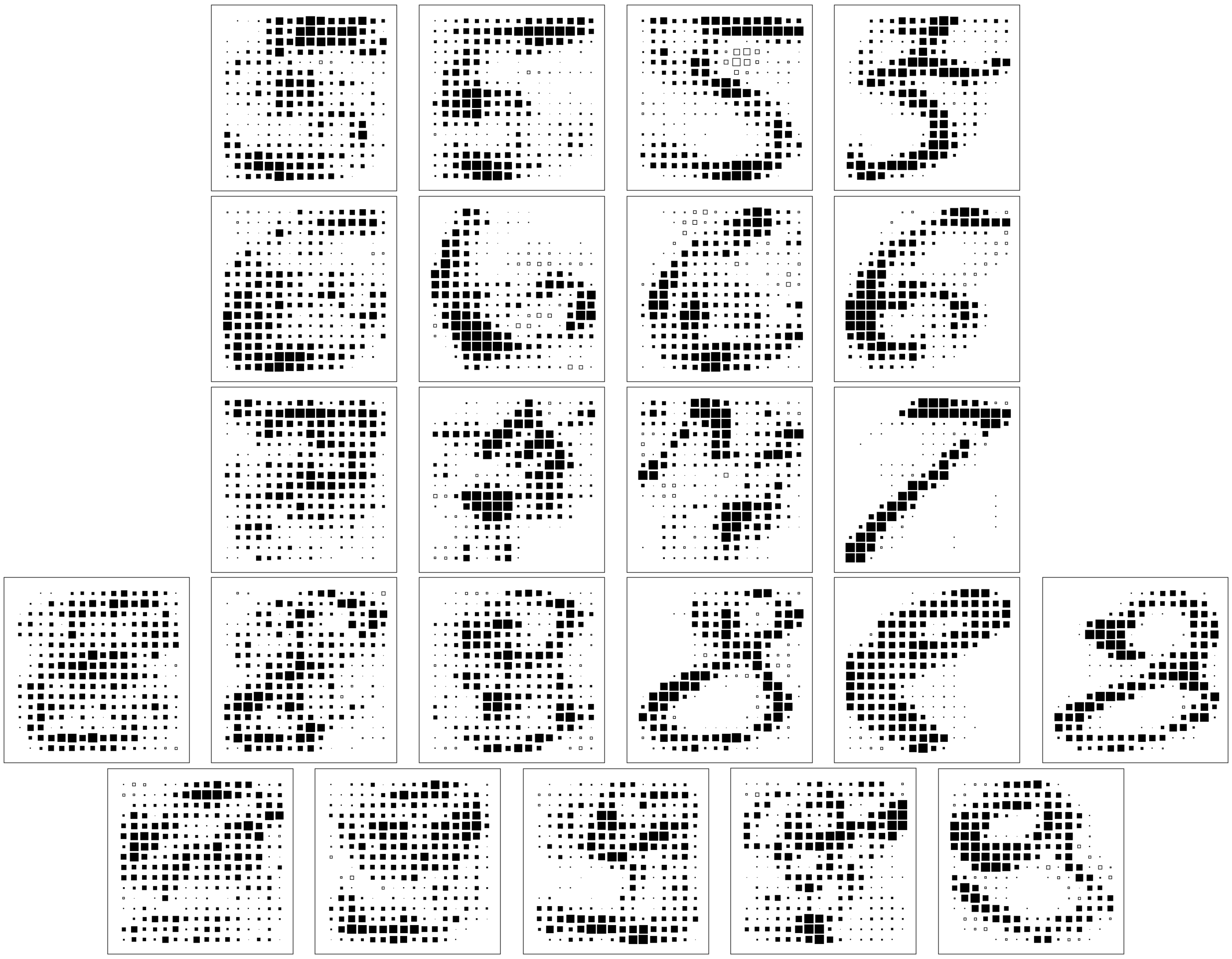

Abbildung 6.8: Die Referenzvektoren für die Ziffern fünf bis neun

Literaturverzeichnis

[Andrews 72] H. Andrews (1972): *Introduction to Mathematical Techniques in Pattern Recognition.* A Wiley-Interscience Publication Wiley & Sons, Inc

[Bosch 86] K. Bosch (1986): *Elementare Einführung in die Wahrscheinlichkeitsrechnung.* Friedr. Vieweg & Sohn, Braunschweig/Wiesbaden

[Bosch 87] K. Bosch (1987): Elementare Einführung in die angewandte Statistik. Friedr. Vieweg & Sohn, Braunschweig/Wiesbaden

[Deterding 89] D. Deterding: *Speaker Normalisation for Automatic Speech Recognition.* University of Cambridge, 1989.

[Duda 73] R. Duda and P. Hart (1973): *Pattern Classification and Scene Analysis.* A Wiley-Interscience Publication Wiley & Sons, Inc

[Elman 89] J. Elman (1988): *Finding Structure in Time.* CRL Tech Report 8801, Univerity of California at San Diego, Center for Research in Language.

[Fahlman 88] S. Fahlman (1988): *Faster-Learning Variations on Back-Propagation: An Empirical Study.* In D.S. Touretzky et al. (eds.) Proceedings of the 1988 Connectionist Models Summer School, Morgan Kaufmann.

[Fahlman et al. 90] S. Fahlman and Lebiere (1990): *The Cascade-Correlation Learning Architecture.* In D.S. Touretzky (ed), Advances in Neural Information Processing Systems 2, Morgan-Kaufman

[Fahlman 91] S. Fahlman (1991): *The Recurrent Cascade-Correlation Architecture.* CMU-CS-91-100, School of Computer Science, Carnegie Mellon Univerity Pittsburg, PA 15213

[Hoefeld et al. 91] M. Hoefeld and S. Fahlman (1990): *Learning with Limited Numerical Precision Using the Cascade-Correlation Algorithm.* CMU-CS-91-130

[Horne et al. 90] B. Horne and D. Hush: *On the optimality of the sigmoid perceptron.* Proc. . of International Joint Conference on Neural Networks, 1:269-272, 1990

[Kohonen et al. 91] T. Kohonen, J. Kangas, J. Laaksoonen and K. Torkkola (1991-1992) : *LVQ_PAK Learning Vector Quantization Program Package.* Laboratory of Computer and Information Science Rakentajanaukio 2 C, SF-02150 Espoo, Finland

[Kreyszig 75] E. Kreyszig (1975): *Statistische Methoden und ihre Anwendungen.* Vandenhoeck & Ruprecht in Göttingen

[Lee et al. 91] D. Lee, Sargur N Srihari and Roger Gaborski: *Bayesian and neural network pattern recognition: a theoretical connection and empirical result with handwritten characters.* Arrtifical Neural Networks and Statistical Pattern Recognition Old and New Connections, I.K. Sethi and A.K. Jain (Editors), 1991 Elsevier Science Publishers B.V.

[Littmann et al. 92] E. Littmann and H. Ritter: *Cascade LLM Networks.* Department of Information Science, Bielefeld University.

[Riedmiller et al. 92] M. Riedmiller & H. Braun (1992): *PROP - A Fast Adaptive Learning Algorithm.* Institut für Logik Komplexität und Deduktionssysteme, University of Karlsruhe, FRG, 1992

[Robinson et al. 88] A. Robinson and F. Fallside: *A Dynamic Connectionist Model for Phoneme Recognition.* In Proceedings of nEuro '88, June 1988.

[Rummelhart et al. 86] D. Rummelhart and J. McClelland: *Parallel Distributed Processing.* Volume 1, MIT Press, 1986.

[Schuermann et al. 92] J. Schürmann, und U.Kreßel: *Mustererkennung mit statistischen Methoden.* Daimler-Benz AG, Forschungszentrum Ulm, Institut für Informationstechnik.

[Simon et al. 92] N. Simon, H. Corporaal and E. Kerckhoffs: *Variations on the Cascade-Correrlation Learning Architekture for Fast Convergence in Robot Control.* Delft Univerity of Technologie, Electrical Engineering Department.

[Vogt 92] M. Vogt (1992): *Implementierung und Anwendung von "Generalized Radial Basis Functions" in einem Simulator neuronaler Netze.* Universität Stuttgart, Institut für Parallele und verteilte Höchstleistungsrechner.

[Zell et al. 92] A. Zell, N. Mache, R. Hübner, M. Schmalzl, T. Sommer, T. Korb: *SNNS User Manual, Version 2.0.* Report No. 3/92, Universität Stuttgart, IPVR, 1992.