

Hierarchische Neuronale Netze: Entwicklung eines  
Programmsystems zur Simulation einer neuen Klasse  
von künstlichen Neuronalen Netzen.

Diplomarbeit

Fachbereich 06

Psychologie und Sportwissenschaft

Justus–Liebig–Universität Gießen

Vorgelegt von:

Ferenc Acs

Eichendorffring 76

35394 Gießen

Betreuer:

Prof.Dr. Gert Haubensak

Gießen, November 2000

In der vorliegenden Arbeit ging es darum, ein Programmsystem zu entwickeln, welches Psychologen, einen Zugang zu einer neuen Art von neuronalen Netzen ermöglicht. Diese neue Klasse von neuronalen Netzen nennt sich „hierarchische neuronale Netze“. Es sollte ein einfach zu bedienendes Programmsystem implementiert werden, welches hinreichend flexibel ist, um auf verschiedene psychologische Fragestellungen angewendet werden zu können.

## Gliederung

1	Einleitung.....	1
2	Was sind neuronale Netze ?.....	3
2.1	Das Neuron.....	3
2.2	Gewichte & Aktivitätsfunktion.....	4
3	Selbstorganisierende neuronale Netze.....	7
3.1	Selbstorganisierende Karten.....	7
3.2	Topologieerhaltende Karten nach Kohonen.....	9
3.3	Hierarchische neuronale Netze.....	12
3.4	Vergleich zwischen den Architekturen.....	14
3.5	Ausgangslage und Fragestellung der Arbeit.....	15
4	Methodik.....	18
4.1	Einleitung.....	18
4.2	Die Lernalgorithmen.....	19
4.3	Zusammenfassung.....	25
5	Programmtechnische Implementation.....	27
5.1	Struktur der Unterprogramme.....	28
5.2	Unit Main.....	29
5.3	Unit Neuron.....	31
5.4	Unit Param.....	35
5.5	Unit Polaru.....	38
5.6	Unit Trainer.....	40
5.7	Unit Vecmat.....	43
5.8	Unit Datau.....	47
5.9	Unit Neuron_e.....	49
5.10	Unit Graph.....	50
5.11	Unit Inspektr.....	56
5.12	Unit FileM.....	58
5.13	Zusammenfassung.....	58
6	Benutzeranleitung.....	60
6.1	Eine Datei mit Trainingsmustern öffnen.....	61

6.2 Festlegen der Netzwerkparameter.....	63
6.3 Trainieren des hierarchischen neuronalen Netzes .....	67
6.4 Die Diagrammfunktion.....	68
7 Tutorial.....	75
7.1 Ordered Search Subtree Update (OSSU).....	76
7.2 Ordered Search Winner Update (OSWU).....	81
7.3 Exhaustive Search Path Update (ESPU).....	86
7.4 Zusammenfassung.....	89
8 Diskussion und Ausblick.....	91
8.1 Diskussion.....	91
8.2 Ausblick: Implementationen zum impliziten Lernen.....	94
8.3 Ausblick: künstliche neuronale Netze in der Psychologie.....	96
9 Literaturverzeichnis.....	98

## Inhaltsverzeichnis

1	Einleitung.....	1
2	Was sind neuronale Netze ?.....	3
2.1	Das Neuron.....	3
2.2	Gewichte & Aktivitätsfunktion.....	4
3	Selbstorganisierende neuronale Netze.....	7
3.1	Selbstorganisierende Karten.....	7
3.2	Topologieerhaltende Karten nach Kohonen.....	9
3.3	Hierarchische neuronale Netze.....	12
3.4	Vergleich zwischen den Architekturen.....	14
3.5	Ausgangslage und Fragestellung der Arbeit.....	15
4	Methodik.....	18
4.1	Einleitung.....	18
4.2	Die Lernalgorithmen.....	19
4.2.1	Ordered Search Subtree Update (OSSU).....	20
4.2.2	Ordered Search Winner Update (OSWU).....	22
4.2.3	Exhaustive Search Path Update (ESPU).....	23
4.3	Zusammenfassung.....	25
5	Programmtechnische Implementation.....	27
5.1	Struktur der Unterprogramme.....	28
5.2	Unit Main.....	29
5.2.1	Wichtige Konstanten und Variablen.....	29
5.2.2	TForm1.Diagramm_Drucken1Click(Sender: TObject); (1,50) .....	30
5.2.3	TForm1.FormCreate (Sender: TObject); (1,68).....	30
5.2.4	TForm1.FormDestroy(Sender: TObject); (2,81).....	30
5.3	Unit Neuron.....	31
5.3.1	Wichtige Konstanten und Variablen.....	31
5.3.2	TNeuron.Create(Gewichtsdimensionen,Zeichendimensionen,Nummer : Longint); (3,70).....	33
5.3.3	TNeuron.Done; (4,95) .....	33
5.3.4	TNeuron.Writekoords(k:array of TWeigt); (4,140).....	33

5.3.5 TNeuron.readkoords(var k:array of TWeight); (5,151).....	34
5.3.6 TNeuron.RandomNumber(min, max: TWeight) : TWeight; (5,178).....	34
5.3.7 TNeuron.getlev : Longint; (5,201).....	34
5.3.8 TNeuron.error(vec : TVector) : double; (6,216).....	34
5.3.9 TNeuron.update(vec : TVector); (6,232).....	35
5.3.10 TNeuron.ischild(n : TNeuron) : boolean; (6,247).....	35
5.4 Unit Param.....	35
5.4.1 Wichtige Variablen.....	36
5.4.2 TParamForm.FormCreate (Sender:TObject); (8,67).....	36
5.4.3 TParamform.Button1Click(Sender:TObject); (9,79).....	37
5.4.4 TParamForm.AlphaEditExit(Sender:TObject); (9,129).....	37
5.4.5 TParamForm.Button2Click(Sender:TObject); (10,157).....	37
5.4.6 TParamForm.spaceonlevel(level:Longint) : LongInt; (11,263).....	38
5.4.7 TParamForm.Button3Click(Sender:TObject); (11,275).....	38
5.5 Unit Polaru.....	38
5.5.1 Wichtige Variablen.....	39
5.5.2 TPolar.phidegread : TDataFormat; (13,39).....	39
5.5.3 TPolar.xread : TDataFormat; (13,51).....	39
5.5.4 TPolar.yread:TDataFormat; (13,56).....	39
5.5.5 Tpolar.xypointread : TRPoint; (13,61).....	39
5.5.6 TPolar.xypointwrite (p:TRPoint); (13,67).....	40
5.5.7 TPolar.xyset(x,y:TDataFormat); (14,88).....	40
5.6 Unit Trainer.....	40
5.6.1 Wichtige Deklarationen und Variablen.....	40
5.6.2 TTrain.OSSU_OSWU_Epoche; (15,70).....	41
5.6.3 TTrain.ESPU_Epoche (17,162).....	42
5.6.4 TTrain.RecordButtonClick(Sender:TObject); (18,256).....	42
5.6.5 TTrain.FormCreate(Sender:TObject); (18,269).....	43
5.6.6 Sonstiges.....	43
5.7 Unit Vecmat.....	43
5.7.1 Wichtige Deklarationen.....	44
5.7.2 TVector.create; (20,56).....	44

5.7.3 TVector.done; (20,62).....	44
5.7.4 TVector.GetDim : LongInt; (21,73).....	44
5.7.5 TVector.Clean; (21,78).....	44
5.7.6 TVector.read(index:LongInt) : TWeight; (21,83).....	44
5.7.7 TVector.read_all (var weights : array of TWeight); (21,95).....	45
5.7.8 TVector.write(index : LongInt; weight:TWeight); (21,116).....	45
5.7.9 TVector.write_all(weights : array of TWeight); (21,131).....	45
5.7.10 TVector.setdim(dim:LongInt); (22,152).....	45
5.7.11 TVector.plus(b:TVector; var erg : TVector); (22,180).....	45
5.7.12 TVector.minus(b:TVector; var erg : TVector); (22,195).....	46
5.7.13 TVector.skalar(b:TVector) : TWeight; (22,208).....	46
5.7.14 Tvektor.mult(b:TWeight; erg : TVector); (23,224).....	46
5.8 Unit Datau.....	47
5.8.1 Wichtige Variablen.....	47
5.8.2 Tdata.Einlesen1Click(Sender:TObject); (24,44).....	48
5.8.3 Tdata.done; (24,70).....	48
5.8.4 Tdata.FormCreate(Sender:TObject); (25,63).....	48
5.8.5 Tdata.DatenInVektoren; (25,90).....	48
5.9 Unit Neuron_e.....	49
5.9.1 Wichtige Deklarationen.....	49
5.10 Unit Graph.....	50
5.10.1 Wichtige Deklarationen und Variablen.....	51
5.10.2 TDiagramm.PaintBoxPaint (Sender:TObject); (28,98).....	52
5.10.3 TDiagramm.FormCreate(Sender:TObject); (28,115).....	52
5.10.4 TDiagramm,PaintBoxMouseMove(Sender:TObject; Shift:TShiftState); (28,136).....	52
5.10.5 TDiagramm.PaintBoxMouseDown(Sender:TObject; Button:TMouseButton; Shift:TShiftState; X,Y:Integer); (29,205).....	53
5.10.6 TDiagramm.PaintBoxMouseUp(Sender:TObject; Button:TMouseButton; Shift:TShiftState; X,Y:Integer); (30,262).....	53
5.10.7 TDiagramm.transform_x (val:TWeight) : Longint; (31,315).....	54
5.10.8 TDiagramm.transform_y(val:TWeight) : Longint; (31,329).....	54

5.10.9 TDiagramm.transformxf(val:Integer) : TWeight; (31,343).....	54
5.10.10 TDiagramm.transformyf(val:Integer) : TWeight; (31,353).....	54
5.10.11 TDiagramm.circlediagramm; (32,418).....	55
5.10.12 TDiagramm.draw_diagramm; (34,544).....	55
5.11 Unit Inspektor.....	56
5.11.1 Wichtige Variablen.....	56
5.11.2 TNeuroninspektor.GewListBoxEnter(Sender:TObject); (38,55).....	57
5.11.3 TNeuroninspektor.GewListBoxClick(Sender:TObject); (39,95).....	57
5.11.4 TNeuroninspektor.GewEditExit(Sender:TObject); (39,115).....	57
5.11.5 TNeuroninspektor.AktNeuronBoxChange(Sender:TObject); (39,126).....	57
5.12 Unit FileM.....	58
5.12.1 TFileManager.save(Dateiname : string):Boolean; (41,27).....	58
5.13 Zusammenfassung.....	58
6 Benutzeranleitung.....	60
6.1 Eine Datei mit Trainingsmustern öffnen.....	61
6.2 Festlegen der Netzwerkparameter.....	63
6.3 Trainieren des hierarchischen neuronalen Netzes .....	67
6.4 Die Diagrammfunktion.....	68
6.4.1 Interaktion mit der Zeichenebene.....	71
6.4.2 Informationen über ein Neuron aufrufen.....	72
6.4.3 Verschieben der Zeichenebene.....	73
6.4.4 Ein Neuron ausschneiden.....	73
6.4.5 Hinein– und Herauszoomen.....	73
6.4.6 Darstellung in Kartesischen oder Polarkoordinaten.....	74
7 Tutorial.....	75
7.1 Ordered Search Subtree Update (OSSU).....	76
7.2 Ordered Search Winner Update (OSWU).....	81
7.3 Exhaustive Search Path Update (ESPU).....	86
7.4 Zusammenfassung.....	89
8 Diskussion und Ausblick.....	91
8.1 Diskussion.....	91
8.2 Ausblick: Implementationen zum impliziten Lernen.....	94



8.3 Ausblick: künstliche neuronale Netze in der Psychologie.....	96
9 Literaturverzeichnis.....	98

## 1 Einleitung

Neuronale Netze sind ein nützliches Werkzeug für die psychologische Forschung. Im Bereich der kognitiven Psychologie liefern sie brauchbare Modelle zur numerischen Modellierung von kognitiven Vorgängen. So konnten bereits Anfang der 80er Jahre wertvolle Erkenntnisse zu Kontexteffekten bei der Buchstabenerkennung gewonnen werden (McClelland & Rumelhart, 1981). Im Bereich des Spracherwerbs gelang es mit einem einfachen neuronalen Netz, Effekte nachzubilden die Kinder beim Lernen von Vergangenheitsformen bei unregelmäßigen Verben zeigen (McClelland & Rumelhart, 1986b). Auch das aus der Gestaltpsychologie bekannte Problem der Figur–Hintergrund–trennung wurde mithilfe von künstlichen neuronalen Netzen erfolgreich angegangen. Zumindest im Gebiet der Tonwahrnehmung liefern neuronale Netze einen Lösungsansatz (Acs, 1995). Hierarchische neuronale Netze scheinen vielversprechende Ansätze aufgrund ihrer Architektur zu zeigen. Diese Untergruppe der neuronalen Netze ist von ihrer Architektur her dazu geeignet, verschiedenste Eingabedaten hierarchisch zu gruppieren und zu klassifizieren (Li et al., 1993). Gerade im Bereich des Impliziten Lernens künstlicher Grammatiken (Servan–Schreiber & Anderson, 1990) scheint dies ein Ansatz zu sein, dessen weitere Verfolgung lohnenswert ist. Die Aufgabe dieser Diplomarbeit bestand darin, ein solches System zu implementieren, welches eine weitere Untersuchung in diesem und in anderen psychologischen Forschungsbereichen ermöglichen soll. Dabei galt es verschiedene Randbedingungen zu berücksichtigen.

1. Das System soll die verschiedensten Eingabedaten verarbeiten können.
2. Es muss verschiedenste Architekturen der zu untersuchenden hierarchischen neuronalen

Netze implementieren können.

3. Verschiedene Lernalgorithmen müssen wählbar sein, da es je nach Struktur der Eingabedaten unterschiedlich gute Algorithmen gibt (Li et al., 1993).
4. Nicht zuletzt sollte das Programm ausbaufähig sein, so daß es ohne größere Umstände an verschiedenste Fragestellungen angepaßt werden kann.

## 2 Was sind neuronale Netze ?

Hier soll eine kurze Einführung zu den wichtigsten Begriffen zu Künstlichen neuronalen Netzen gegeben werden. Diese Einführung ist bewußt knapp gehalten, da es nicht die Aufgabe dieser Diplomarbeit ist, eine allgemeine Einführung zu geben. Um jedoch die wichtigsten Begriffe wie „Neuron“, „Aktivierungsfunktion“ und „Gewichte“ zu erläutern, ist es sinnvoll diese Begriffe hier kurz zu definieren. Eine sehr gute und ausführliche Einführung in diese Materie, besonders mit den Implikationen für die Psychologie wurde von McClelland & Rumelhart vorgelegt (McClelland & Rumelhart, 1986a). Auf dieses Buch sei an dieser Stelle verwiesen.

### 2.1 Das Neuron

Das grundlegendste Element in einem neuronalen Netz ist das Neuron selbst. Ursprünglich ist dieser Begriff schon durch die Physiologie belegt, die mit einem Neuron eine Nervenzelle biologischer Organismen bezeichnet. Wenn hier von einem Neuron gesprochen wird, ist jedoch eine abstrakte Programmeinheit gemeint, die bestimmte, später noch zu erläuternde Berechnungen durchführt. Es können zwar Parallelen zwischen der Physiologie von Organismen und der Theorie Künstlicher neuronaler Netze gezogen werden (Ritter, Martinez & Schulten; Kapitel 1, 1991), jedoch ist diese Sichtweise vom heutigen Standpunkt aus nur als Analogie zu sehen. Es gibt noch nicht genügend Befunde aus der Physiologie, die ein aussagekräftiges Urteil über die Verwandtschaft zwischen physiologischen Neuronen und Neuronen in Künstlichen neuronalen Netzen erlauben würden.

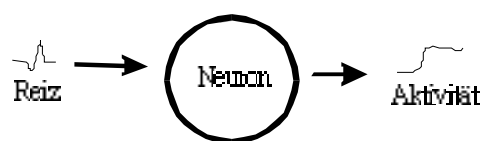


Abbildung 1 Ein einfaches Neuron

Ein Neuron ist eine einfache Ein–Ausgabeeinheit. Diese Einheit erhält bestimmte Werte von außen<sup>1</sup>. Diese Werte, die von außen kommen, nennen wir einen Reiz, dieser Reiz wird innerhalb des Neurons verrechnet und erzeugt eine Ausgabe, die wir Aktivität nennen.

## **2.2 Gewichte & Aktivitätsfunktion**

Neurone sind untereinander verbunden und interagieren miteinander. Diese Interaktion wird durch Gewichte vermittelt. Diese Gewichte beeinflussen die Aktivität eines Neurons, zum einen dadurch, daß der Reiz, der in ein Neuron eingelesen wird, mit einem Gewicht versehen wird, das den Einfluss dieses Reizes auf das Neuron bestimmt, zum anderen dadurch, daß die Aktivität anderer Neuronen selbst einen Reiz darstellen kann, der in das Neuron einfließt. Ein Neuron hat ebenso viele Gewichte, wie Verbindungen zu anderen Neuronen, hinzu kommen noch die Verbindungen nach außen, zu den Eingabereizen. Gewichte können sowohl aktivierend als auch hemmend wirken. In vielen Architekturen wird dies dadurch realisiert, daß die Gewichte Werte zwischen +1 und –1 annehmen können. Hat ein Neuron beispielsweise ein Gewicht von +1 zum Reiz, so bedeutet dies, daß dieser Reiz mit seinem vollen Betrag aktivierend in das Neuron eingeht. Hat das Gewicht den Wert 0, so bedeutet dies: der Reiz hat keinen Einfluss auf die Aktivität des Neurons. Ist das Gewicht im negativen Bereich, so spricht man von einer hemmenden Verbindung, d.h. der negativ gewichtete Reiz wird den Betrag der Aktivität des Neurons senken.

---

<sup>1</sup> Gemeint sind Eingabedaten die dem neuronalen Netz präsentiert werden und in einer bestimmten Weise von diesem Netz verrechnet werden.

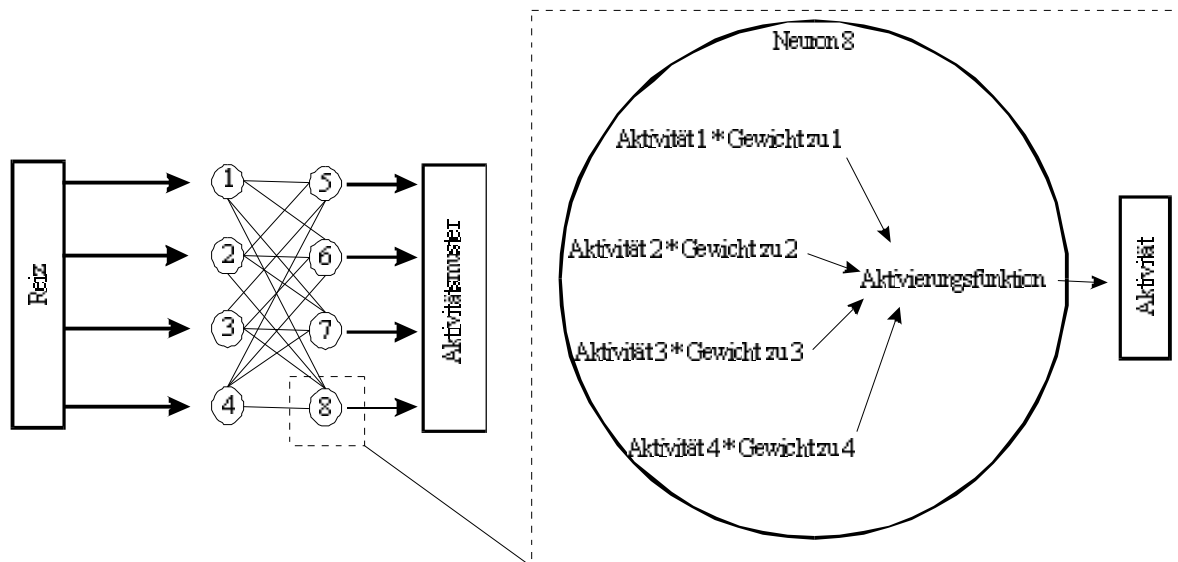


Abbildung 2 Ein einfaches neuronales Netz

Die Aktivitätsfunktion eines Neurons bestimmt die numerische Größe der Aktivität eines Neurons. Am allgemeinsten lässt sich die Aktivitätsfunktion folgendermaßen ausdrücken:

$$A = \text{Aktivitätsfunktion} \left( \sum_j^N g_j(R_j) \right) \quad (1)$$

wobei  $g_j$  das Gewicht des Neurons zum Reiz  $R_j$  darstellt.  $N$  sei die Anzahl der Verbindungen, die ein Neuron besitzt. Die Aktivitätsfunktion eines Neurons ist gewöhnlich eine sigmoid verlaufende Funktion, d.h. ist die Summe der gewichteten Reize ein großer negativer Wert, so sinkt die Aktivität des Neurons auf einen Wert nahe Null, jedoch unterschreitet die Aktivität des Neurons nie eine gewisse Grenze; bei den meisten Architekturen wird sie per Konvention nie kleiner als 0. Analoges gilt für positive Summen der gewichteten Reize, die Aktivität eines Neuron wird per Konvention nie größer als 1.

Ein Beispiel für eine sigmoide Funktion wäre:

$$\sigma(x) = \frac{1}{1 + \exp(x)} \quad (2)$$

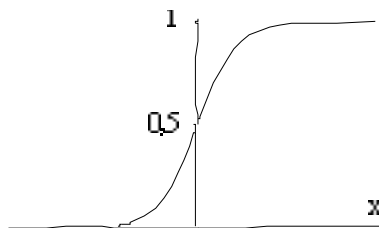


Abbildung 3 Verlauf der sig-  
moiden Funktion aus (2). (Fermi  
Funktion)

Das Besondere neuronaler Netze stellen die Gewichte und deren Veränderungen dar. neuronale Netze versuchen ihre Gewichtungskonfiguration derart anzupassen, daß das gewünschte Resultat nach vielen Anpassungsschritten erreicht wird. In diesen Anpassungsschritten werden dem neuronalen Netz Eingabereize präsentiert, das Ergebnis in Form von Aktivitäten ausgelesen und die Gewichte werden in kleinen Schritten, d.h. iterativ, so angepaßt, daß das gewünschte Ergebnis ausgegeben wird. Diese schrittweise Anpassung an einen bestimmten Zielzustand nennt man das Training eines neuronalen Netzes. Konkretere Einführungen zu dem Gebiet der Gewichte, dem Training und Aktivitätsfunktionen lassen sich am besten an praktischen Programmen nachvollziehen (McClelland & Rumelhart, 1988).

## **3 Selbstorganisierende neuronale Netze**

Eine eigene Klasse von neuronalen Netzen stellen selbstorganisierende neuronale Netze dar. Normalerweise passen neuronale Netze ihre Gewichte iterativ den Eingabereizen an, indem eine Rückmeldung an das neuronale Netz erfolgt, ob das Muster der ausgelesenen Aktivitäten nun das gewünschte Resultat erbracht hat oder nicht. In der Trainingsphase ist es also immer erforderlich, dem neuronalen Netz eine Rückmeldung über die Güte seiner Aktivitätsmuster zu geben. Selbstorganisierende neuronale Netze beschreiten da einen anderen Weg. Sie versuchen sich den Eingabereizen in der Form anzupassen, daß diese nach einem erfolgreichen Training gruppiert werden können. Selbstorganisierende neuronale Netze versuchen also Gemeinsamkeiten, in der Gesamtheit der Eingabereize zu finden und diese nach Ähnlichkeit zu gruppieren. Die folgenden Modelle neuronaler Netze sind nur noch mit den Werkzeugen der Linearen Algebra und der Vektorrechnung zu beschreiben, es gibt jedoch gute Einführungsliteratur zu dem Gebiet der Vektorrechnung (Jänich, 1991).

### **3.1 Selbstorganisierende Karten**

Bei dem bisher vorgestellten Ansatz zu neuronalen Netzen kam den Gewichten der Neurone eine zentrale Rolle zu. Bei der Gruppe der selbstorganisierenden Karten (selbstorganisierende neuronale Netze) kommt der Aspekt hinzu, daß die räumliche Lage der Neurone zueinander eine wichtige Rolle spielt. Am einfachsten kann man sich die Neurone in einer zweidimensionalen Schicht angeordnet vorstellen. Es geht um die Frage, wie Neuronen ihre Verbindungen zueinander organisieren, damit die räumliche Verteilung der Aktivitäten in der Schicht, Ähnlichkeiten der Eingabereize wiedergeben. In einem einfa-



chen Fall kann man sich vorstellen, daß die Eingabereize von einer Schicht Rezeptorneuronen geliefert werden. Als Analogie kann die Schicht der Stäbchenzellen im menschl-

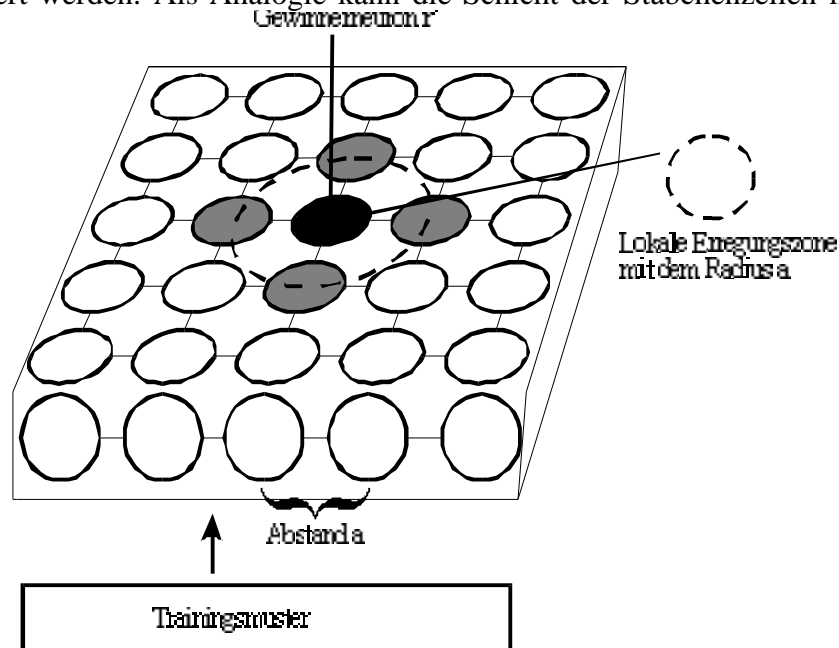


Abbildung 4 Schematische Darstellung eines Kohonen Netzwerkes.

chen Auge dienen. Diese Eingabeschicht gibt nun ihre Aktivitäten an die Schicht von Neuronen weiter, die topologieerhaltend arbeiten soll. Erste Vorschläge zu dieser Sichtweise und deren biologischer Plausibilität kamen von Wilshaw und v.d. Mahlsburg (Wilshaw & v.d. Mahlsburg, 1976; v.d. Mahlsburg & Wilshaw, 1977). Topologieerhaltend meint nun, daß benachbarte Reize in der Schicht der Rezeptorneuronen auch in der darunterliegenden Schicht benachbart repräsentiert werden, jedoch nicht als simple Abbildung, vielmehr soll die Topologie erhalten bleiben, d.h. benachbarte Reize in der Eingabschicht rufen Aktivitätsmuster hervor, die auch benachbart sind. Frühe Modelle hatten eine gewisse biologische Detailtreue zum Anliegen, diese Zielsetzung wurde in späteren Modellen aus Gründen der Zweckmäßigkeit und Effizienz fallengelassen.

Das Modell von Kohonen (1982) bildet einen wesentlich abstrakteren und allgemeineren Ansatz zur Modellierung der selbstorganisierenden neuronalen Netze. Es soll im Fol-

genden genauer erläutert werden, da hierarchische neuronale Netze wiederum eine Abstraktion von Kohonens Modell darstellen.

### **3.2 Topologieerhaltende Karten nach Kohonen**

Das Modell von Kohonen geht von einer Schicht Neurone aus, die zweidimensional angeordnet sind. Der Einfachheit halber stellt man sich diese Neurone wohlgeordnet in Spalten und Zeilen vor, sie sind also schachbrettartig angeordnet. Der Reiz, der in dieses neuronale Netz eingespeist wird, wird an alle Neurone weitervermittelt. Die Architektur nach Kohonen strebt nun Topologieerhaltung in der Form an, daß sich benachbarte Neurone beim Lernen gegenseitig beeinflussen.

Man geht davon aus, daß ein Neuron  $r'$  durch den eingehenden Reiz am stärksten gereizt wird, d.h. die Aktivierung dieses Neurons wird höher als die anderer Neurone sein. Dieses Neuron  $r'$  ist also sozusagen der Gewinner bei einem bestimmten eingehenden Reiz. Wenn das Netz trainiert wird heisst dies, daß die Gewichte des Gewinnerneurons  $r'$  noch nicht unbedingt optimal an den Eingabereiz angepaßt sind. Das Gewinnerneuron  $r'$  soll nun seine Gewichtungskonfiguration iterativ derart anpassen, daß seine Aktivierung bei diesem Reiz maximiert wird. Ferner beeinflußt das Gewinnerneuron die benachbarten Neurone im Abstand  $a$ , ihre Gewichtungskonfiguration ebenfalls so anzupassen, daß ihre Aktivierung zu diesem Reiz maximiert wird, jedoch in einem geringeren Maße als dies bei  $r'$  der Fall ist. Neurone, deren Abstand zu  $r'$  größer als  $a$  ist, werden dagegen gehemmt, ihre Gewichtungskonfiguration zu optimieren. Dies nennt man laterale Inhibition.

Mathematisch lässt sich dies folgendermaßen formulieren: Die Aktivierungsfunktion eines Neurons  $r$ , bei einem Reiz  $X$  sei

$$A_r(X) = \sigma \left( \sum_l w_{rl} x_l - \theta \right) \quad (3)$$

$\sigma(x)$  ist eine monoton verlaufende sigmoide Funktion. Gegen minus unendlich strebt diese Funktion dem Wert 0 zu, gegen plus unendlich strebt sie dem Wert 1 zu.  $\theta$  verschiebt die Erregungsschwelle des Neurons. Daraus ergibt sich, daß ein Neuron nur noch schwach reagiert, wenn sich ein Reiz zu sehr von seiner Gewichtungskonfiguration unterscheidet. Da sich die Neurone jedoch gegenseitig beeinflussen, müssen wir noch eine Erweiterung vornehmen:

$$A_r(X) = \sigma \left( \sum_l w_{rl} x_l + \sum_{r'} g_{rr'} A_{r'} - \theta \right) \quad (4)$$

$g_{rr'}$  modelliert eine Rückkoppelung zwischen dem Neuron  $r$  und dem Gewinnerneuron  $r'$ , die derart definiert ist, daß sich mit wachsendem Abstand  $a$  zwischen  $r$  und  $r'$  der Wert der Funktion  $g_{rr'}$  immer mehr in den negativen Bereich bewegt und dadurch die Aktivität von Neuronen, deren Abstand zu  $r'$  größer als  $a$  ist, auf Null sinken lässt. Die Funktion (4) ist jedoch ein nichtlineares Gleichungssystem und als solches schwer zu lösen. Dies zwang Kohonen dazu, an seinem Modell einige Näherungsannahmen zu machen. Für Kohonens Modell ist es wichtig, den Gewinner  $r'$  zu finden. Kohonen (1982) konnte zeigen, daß es reicht, den Gewinner nur aufgrund des Eingabereizes  $X$  zu ermitteln

$$\|W_{r'} - X\| = \min_r \|W_r - X\| \quad (5)$$

Das Gewinnerneuron  $r'$  ist also das Neuron, dessen Gewichtungskonfiguration,  $W$  am nächsten zum Eingabereiz  $X$  liegt. Dies ist einfacher zu verstehen, wenn man  $W_r$  und  $X$  als Vektoren sieht. Dann ist das Gewinnerneuron  $r'$  dasjenige, dessen Vektor  $W_{r'}$  die geringste euklidische Distanz zu  $X$  hat. Daher soll nun im folgenden die Gewichtungskonfiguration  $W$

als Gewichtsvektor  $W$  und der Eingabereiz  $X$  als Trainingsmuster  $X$  bezeichnet werden. Da die quantitative Höhe der Aktivierungsfunktion bei Kohonen keinen entscheidenden Einfluss auf das qualitative Verhalten eines Kohonen Netzes hat, ist es zulässig, die genaue quantitative Bestimmung der Aktivierungsfunktion zu vernachlässigen und anzunehmen, daß dasjenige Neuron  $r'$  eine maximale Aktivierung besitzt, welches nach (5) ermittelt wird.

Die Veränderung des Gewichtsvektors  $W$  beim Training des Kohonen Netzes mit einem Trainingsmuster  $X_i$  bestimmt sich aus

$$\delta W_{rl} = \epsilon h_{rr'} (X_l - W_{rl}) \quad (6)$$

$\epsilon$  ist hierbei ein Wert zwischen Null und Eins, der die Größe des Adaptationsschrittes bestimmt. Er sollte über die Anzahl der Iterationen abnehmen, damit nach einer anfänglich relativ groben Anpassung der Gewichtsvektoren, im Verlauf der Iterationen eine Fein Anpassung stattfinden kann. Die Neuronenschicht soll sich also asymptotisch stabilisieren, damit gegen Ende der Adaptationsschritte eine Oszillation der Gewichtsvektoren vermieden wird. Der Term  $h_{rr'}$  nähert nun das Konstrukt der lateralen Inhibition an. Es soll vermieden werden, daß Neurone, deren Abstand zu  $r'$  größer als  $a$  ist, ihre Gewichtsvektoren anpassen können. Dies kann man durch die Modellierung von  $h_{rr'}$  gemäß einer Gaußglockenfunktion erreichen, die ihr Maximum bei  $r=r'$  hat

$$h_{rr'} = \exp(-(r-r')^2/2a^2) \quad (7)$$

Dies ist ausreichend, um nun den Trainingsalgorithmus nach Kohonen zu beschreiben (Kohonen, 1984)

Wir verwenden dazu eine sogenannten Pseudoprogrammiersprache. Sie beschreibt den Algorithmus unabhängig von einer bestimmten Programmiersprache in formaler Weise.

1. Initialisiere den Gewichtsvektor jedes Neurons auf einen Zufallswert.
2. **FOR** jedes Trainingsmuster  $X_i = \langle x_0^i, x_1^i, \dots, x_{n-1}^i \rangle$  **DO**
  - a) Bestimme das Gewinnerneuron gemäß Bedingung (5)
  - b) Aktualisiere die Gewichte der Neurone nach Formel (6)
3. **GO TO** Schritt 2

Der Algorithmus nach Kohonen ist also recht einfach zu implementieren, hat jedoch den klaren Nachteil, daß er sehr rechenaufwendig ist und eine hierarchische Gruppierung der Eingabedaten nicht möglich ist.

### **3.3 Hierarchische neuronale Netze**

Hierarchische neuronale Netze sind, wie der Name schon sagt, hierarchisch angeordnet. Ausgehend von einem sogenannten Wurzelneuron sind die Neurone in einer baumartigen Struktur angeordnet. Das Wurzelneuron hat eine bestimmten Anzahl Neurone, die direkt mit ihm verbunden sind, beispielsweise drei. Diese drei Neurone bezeichnet man als Kinder des Wurzelneurons. Das Wurzelneuron ist für die ersten drei Neurone auf der ersten Ebene das Elternneuron. Die Kinder des Wurzelneurons können ihrerseits Eltern sein. Wenn diese jeweils drei Nachkommen haben, haben wir es nun mit insgesamt 13 Neuronen zu tun. Das Wurzelneuron auf Ebene 1, die drei Nachkommen des Wurzelneurons auf Ebene 2 und die 9 Nachkommen der Kinder des Wurzelneurons auf Ebene 3.

Die Neurone sind folgendermaßen untereinander verbunden: jedes Neuron bis auf das Wurzelneuron hat ein Elternneuron, mit dem es verbunden ist. Seinerseits ist jedes Neuron

auch mit seinen Kindneuronen verbunden, bis auf die Neurone auf der untersten Ebene. Im aktuellen Beispiel hätten also die Neurone auf der Ebene 3 keine Kinder, die Neurone auf Ebene 2 haben Kinder, sowie ein Elternteil und das Wurzelneuron hat per Definition kein Elternteil.

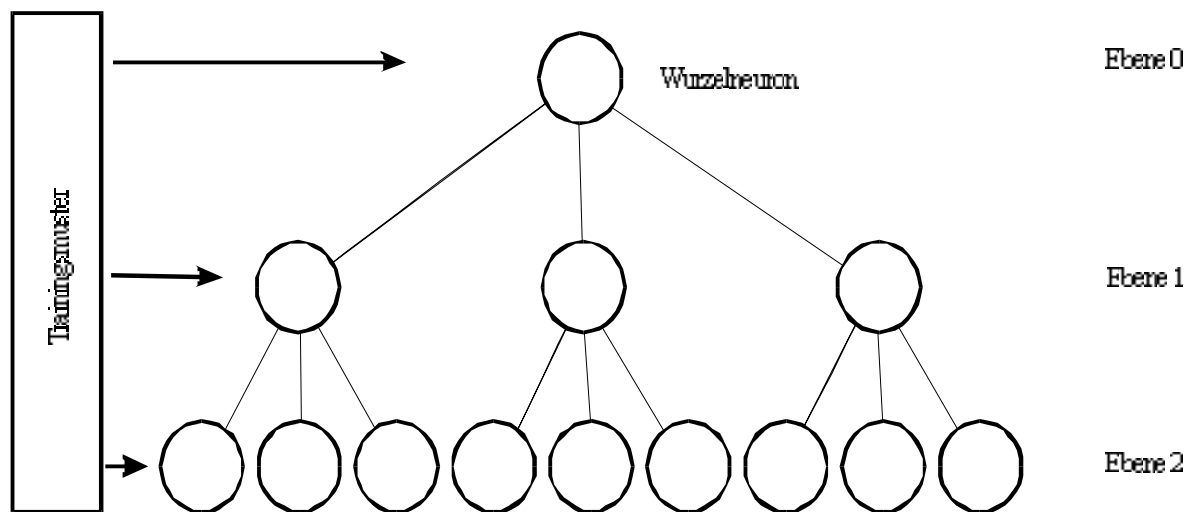


Abbildung 5 Ein einfaches hierarchisches neuronales Netz

Diese Verbindungen zwischen den Neuronen sind nun für die Lernalgorithmen von ausschlaggebender Bedeutung. Prinzipiell wird immer ein Gewinnerneuron auf einer Ebene gesucht, dessen Gewichtsvektor die geringste euklidische Distanz zu dem aktuellen Trainingsmuster aufweist (analog zu Formel (5)). Ist dieser Gewinner nun gefunden, so findet eine Aktualisierung der Gewichte gemäß der Verwandtschaftsbeziehungen statt. Ist ein Gewinnerneuron ermittelt, so werden auch die Gewichte seiner Kinder und deren Kinder usw., aktualisiert.

Eine Ausnahme bildet der OSWU Algorithmus, bei dem nur das Gewicht des Gewinnerneurons aktualisiert wird. Die Lernalgorithmen werden jedoch an späterer Stelle noch detaillierter dargestellt.

Dies führt dazu, daß sich ein hierarchisches neuronales Netz hervorragend dazu eignet, Trainingsmuster in Hierarchien einzuteilen. Hat man Trainingsmuster, die sich sehr ähnlich sind, so ist die Wahrscheinlichkeit sehr hoch, daß die jeweiligen Gewinnerneurone auf der untersten Ebene das gleiche Elternneuron haben. Ein hierarchisches neuronales Netz gruppiert also die Trainingsmuster gemäß ihrer Ähnlichkeit und ist durch den hierarchischen Aufbau dazu in der Lage, prototypische Muster zu generieren, die die Gemeinsamkeiten einer ganzen Gruppe von ähnlichen Mustern abbilden. In unserem Beispiel wäre ein prototypisches Muster der Gewichtsvektor eines Neurons auf Ebene 2. Da sich die Gewichtsvektoren den Trainingsmustern anpassen, kann man nach erfolgtem Training auch den umgekehrten Weg gehen und aus einem Gewichtsvektor ein Trainingsmuster rekonstruieren. Interessant wird es nun, wenn man ein Muster aus einem Neuron rekonstruiert, das höher in der Hierarchie liegt. Dieses Muster ist dann der besagte Prototyp einer Gruppe von Trainingsmustern.

Ein hierarchisches neuronales Netz kann also ausgehend von Trainingsmustern Gemeinsamkeiten in ihnen Entdecken und Hierarchien beliebiger Abstraktionstiefe bilden.

### **3.4 Vergleich zwischen den Architekturen**

Kohonen strebte mit der Gestaltung seines Netzwerktypus an, eine Architektur zu schaffen, die neurophysiologisch begründbar ist. Aus dem Bereich der Wahrnehmungspsychologie gibt es Arbeiten, die diesen Zusammenhang belegen. So konnten Toivaiainen et al. (1998) Zusammenhänge zwischen der Reaktionsweise des menschlichen Gehirns auf Töne verschiedener Klangfarben und der Reaktion eines Kohonen Netzwerkes auf diese Töne aufzeigen. Eine frühe Arbeit von Saarinen (1985) legte die Vermutung nahe, daß es

Ähnlichkeiten zwischen der Neuronenaktivität in einem Kohonen Netzwerk und der Neuronenaktivität im visuellen Kortex gibt. Solche Befunde stehen für hierarchische neuronale Netze noch aus.

Beide Netzwerkarchitekturen gehören zu den sogenannten Selbstorganisierenden neuronalen Netzen. Diese versuchen Trainingsmuster gemäß ihrer Ähnlichkeit zu gruppieren. Generiert man zwei Gruppen von Trainingsmustern, die sich hinreichend unterscheiden, so kann man in einer neuronalen Karte nach Kohonen sehen, daß sich zwei Regionen von benachbarten Neuronen herausbilden, die jeweils auf die Reize einer Gruppe ansprechen. Hier liegt auch der Hauptunterschied zu Kohonen Netzen. Während man anhand der Aktivitätsmuster eines Kohonen Netzwerkes nur entscheiden kann, ob ein Trainingsmuster nun der einen oder der anderen Gruppe angehört, so kann man aus den Gewichtsvektoren eines Elternneurons in einem hierarchischen neuronalen Netz ein prototypisches Muster generieren, welches die Gemeinsamkeiten aller Trainingsmuster einer Gruppe enthält.

Ein hierarchisches neuronales Netz kann also nicht nur über Gruppenzugehörigkeiten von Trainingsmustern entscheiden, sondern ist darüberhinaus in der Lage, Hierarchien aus diesen Mustern zu bilden.

### ***3.5 Ausgangslage und Fragestellung der Arbeit***

Hierarchische neuronale Netze stellen in der psychologischen Forschung einen fast unbekannten Netzwerktypus dar. Während es zu anderen Modellen neuronaler Netze schon eine Reihe von Untersuchungen gibt (Übersichten in: McClelland & Rumelhart 1986b, Bechtel & Abrahamsen 1991), sind hierarchische neuronale Netze und deren Anwendung in der Psychologie noch weitgehend unbekannt. Diese Arbeit hatte zum Ziel, ein Programm–



system zu entwickeln, welches die Untersuchung von hierarchischen neuronalen Netzen ermöglichen soll.

- Das System sollte einfach bedienbar sein, alle Grundfunktionen des Programmes sollten über einen Mausklick erreichbar sein. Es mußte also eine stark graphisch orientierte Anwenderoberfläche geschaffen werden, die dem Anwender der Simulation alle wichtigen Daten auf einen Blick präsentiert.
- Das System sollte die Struktur des hierarchischen neuronalen Netzes dem Anwender visualisieren. Es mußte also ein Weg gefunden werden, die Architektur des Netzes und die Lage der Gewichtsvektoren darzustellen. Dies stellte besonders bei hochdimensionalen Trainingsmustern eine Herausforderung dar, da ein Weg gefunden werden mußte, wie man das Verhalten des Netzes im  $n$ -dimensionalen Raum zweidimensional darstellen kann. Denn letztlich steht den meisten Anwendern nur ein zweidimensional darstellendes Gerät in Form eines Computermonitors zur Verfügung. Die meisten Trainingsmuster, die bei der Untersuchung von psychologischen Fragestellungen verwendet werden, liegen in der Regel in einer wesentlich höheren Dimensionalität vor.
- Die Trainingsmuster, die in das System eingespeist werden, sollten nicht in einem proprietären Dateiformat vorliegen, da es extrem aufwendig ist, Daten die schon digital vorliegen, in ein unbekanntes Dateiformat zu konvertieren. Es mußte also eine Schnittstelle gefunden werden, mit der der Import von Daten ohne großen Konvertierungsaufwand möglich ist. Als Schnittstelle boten sich gängige Datenbankformate an, die den meisten anderen datenverarbeitenden Programmen bekannt sind.
- Das System mußte hochflexibel sein, was einerseits die Dimensionalität als auch den Umfang der Eingabedaten angeht. Andererseits sollten unterschiedlichste Netzwerkar-

chitekturen und Größen simuliert werden können. Das System muß also in der Lage sein, einerseits kleine hierarchische neuronale Netze trainieren zu können, andererseits aber auch sehr umfangreiche Netze, die beispielsweise mit mehreren tausend Neuronen arbeiten um hochkomplexe Eingabedaten korrekt klassifizieren zu können.

- Die Programmierung des Systems sollte in einer Programmiersprache erfolgen, die in psychologischen Anwenderkreisen einen hohen Bekanntheitsgrad aufweist. Das System muß erweiterungsfähig sein. Es soll in der wissenschaftlichen Anwendung ein Grundgerüst bieten, das leicht auf seine spezielle Fragestellungen angepaßt werden kann. Die Dokumentation dieser Schnittstellen und der Arbeitsweise des Systems ist der Schwerpunkt dieser Arbeit.
- Es sollte also ein System geschaffen werden, das weiterführende Untersuchungen zu hierarchischen neuronalen Netzen in der psychologischen Forschung ermöglicht, ohne daß sich ein wissenschaftlicher Anwender mit den nichttrivialen Hürden der programmtechnischen Implementation von Hierarchischen neuronalen Netzen auseinander zu setzen hat.

## 4 Methodik

### 4.1 Einleitung

Wie schon im Kapitel vorher gezeigt, besteht ein hierarchisches neuronales Netz aus einer baumartigen Struktur, in der die Neurone angeordnet sind. In Analogie zu Kohonen Netzwerken ist auch hier die euklidische Distanz zwischen dem Trainingsmuster und den Gewichtsvektoren der Neurone ausschlaggebend um ein Gewinnerneuron ermitteln zu können. Diese Funktion nennt man die Fehlerfunktion eines hierarchischen neuronalen Netzes, sie ist definiert durch

$$fehler(n,i) = frequenz(n) \sum_{l=0}^{n-1} [x_l^i - w_l^n]^2 \quad (8)$$

Ein neuer Parameter der in (8) erscheint, ist die Frequenz  $frequenz(n)$ . Diese gibt an, wie oft ein Neuron im Laufe des Trainings als Gewinner ermittelt wurde. Wird also ein Neuron öfter als Gewinner ermittelt, so steigt der Wert seiner Fehlerfunktion an, dies gibt anderen Neuronen, die mit ihm verbunden sind, die Möglichkeit ihrerseits ihre Gewichtsvektoren anzupassen, indem ihre Wahrscheinlichkeit steigt, als Gewinner ermittelt zu werden. Dies dient dazu, ein Übertrainieren einzelner Neurone zu vermeiden, wenn ihr Gewichtsvektor schon hinreichend gut an den Eingabereiz angepaßt ist. Wenn es den Parameter  $frequenz(n)$  nicht geben würde, so würde in zyklischer Abfolge immer wieder dasselbe Neuron als Gewinnerneuron selektiert werden. Dadurch würde die Anpassung der Gewichtungskonfiguration der anderen Neurone behindert werden. Die hier vorgestellten Algorithmen greifen im Wesentlichen auf den Artikel von Li et al. (1993) zurück.

## **4.2 Die Lernalgorithmen**

Das Lernen in einem hierarchischen neuronalen Netz läßt sich in einer sehr einfachen Regel ausdrücken: verringere die euklidische Distanz des Gewichtsvektors eines Neurons zu dem Trainingsmuster. Dies muß allerdings in einer iterativen Weise geschehen, d.h. die Anpassung des Gewichtsvektors eines Neurons darf nicht ad hoc erfolgen, da es ein neuronales Netz in der Regel mit einer Vielzahl von Trainingsmustern zu tun hat. Eine zu plötzliche Anpassung würde zu einer Fluktuation der Gewichtsvektoren führen, es könnte sich kein stabiler Zustand in den Gewichtsvektoren der Neurone bilden. Sie würden mit jedem Trainingsmuster völlig neu angepaßt werden. Diese Fluktuation gilt es zu vermeiden, damit sich im hierarchischen neuronalen Netz eine stabile Konfiguration der Gewichtsvektoren seiner Neurone bilden kann.

Der Gewichtsvektor des Neurons  $n$  wird durch folgende Funktion aktualisiert:

$$W_l^n(t+1) = W_l^n(t) + \alpha \times [x_l^i - W_l^n(t)] \quad (9)$$

Der Parameter Alpha gibt die Lernrate an, das bedeutet die Geschwindigkeit pro Zeiteinheit, mit der sich die Gewichtsvektoren der Neuronen an die Eingabevektoren anpassen. Der Parameter Alpha ist also dazu da, die oben erwähnte Fluktuation der Gewichtsvektoren zu verhindern.

Bei den nun vorzustellenden Lernalgorithmen besteht der Hauptunterschied darin, welche Gewichtsvektoren von welchen Neuronen aktualisiert werden. Der Gewichtsvektor des Neurons wird selbstverständlich aktualisiert, jedoch unterscheiden sich die Algorithmen in der Art, welche Neurone in Abhängigkeit vom Gewinnerneuron und der Struktur des Netzes noch aktualisiert werden.

Im folgenden sollen die einzelnen Algorithmen (OSSU, OSWU, ESPU) dargestellt werden.

#### 4.2.1 Ordered Search Subtree Update (OSSU)

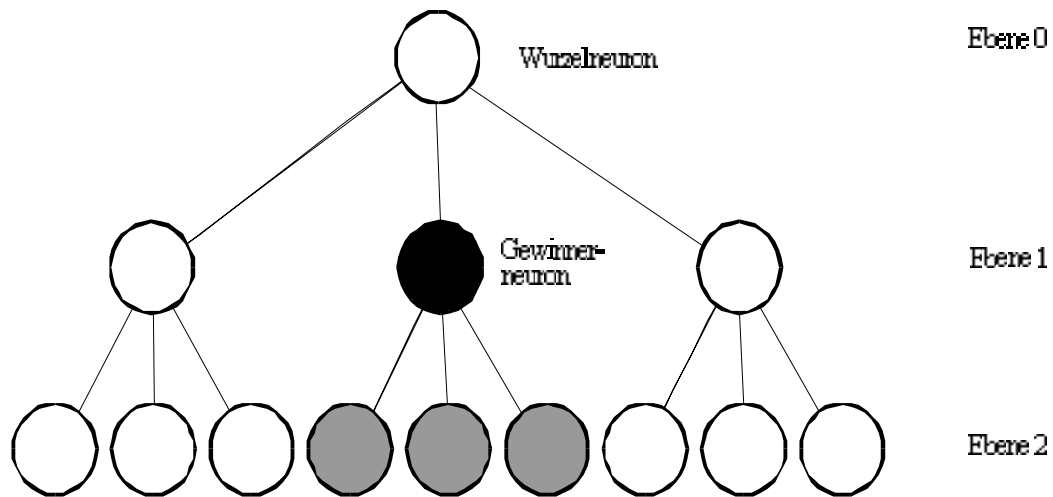


Abbildung 6 Schematische Darstellung der Aktualisierung beim OSSU Algorithmus

Bei diesem Algorithmus wird der Gewinner ausgehend von der baumartigen Struktur des Netzes gesucht. Die Suche nach dem Gewinnerneuron fängt effektiv bei den Kindern des Wurzelneurons an. Das Kind  $n$  mit dem geringsten Fehler  $fehler(n,i)$  zum Trainingsmuster  $i$  ist das Aktuelle Gewinnerneuron. Unter seinen Kinderneuronen wird nun wiederum dasjenige mit dem geringsten Fehler ermittelt. Dieser Prozeß wird fortgesetzt bis man durch sukzessive Suche durch alle Ebenen des Netzes den endgültigen Gewinner ermittelt hat (Ordered Search). Ausgehend von diesem werden die Gewichte seiner Kinder und Kindeskindern bis zur letzten Ebene des hierarchischen neuronalen Netzes aktualisiert (Subtree Update).

Dieser Vorgang wird für alle Trainingsmuster wiederholt.

Ausgehend von der nächsthöheren Ebene wird nun erneut ein Gewinner ermittelt und alle Neurone in dessen Unterzweig aktualisiert, bis alle Ebenen durchlaufen sind.

Nun beginnt eine neue Epoche<sup>2</sup> des Trainings.

---

2 Epoche: alle Trainingsmuster wurden dem Netz einmal präsentiert und jedesmal hat eine Aktualisie-

1. Initialisiere den Gewichtsvektor jedes Neurons auf einen Zufallswert.
2. Gewinner = Wurzelneuron;  
Ebene = 1;
3. **FOR** jedes Trainingsmuster  $X_i = \langle x_0^i, x_1^i, \dots, x_{n-1}^i \rangle$  **DO**
  - a) AktGewinner = Wurzelneuron.
  - b) **FOR** AktEbene = 2 **TO** Ebenen **DO**  
AktGewinner = Das Kind von AktGewinner, das den geringsten Fehler hat;
  - c) Gewinner = AktGewinner;
  - d) Aktualisiere das Gewicht von jedem Neuron im Unterzweig von Gewinner;
  - e) Aktualisiere das Gewicht von Gewinner;  
Erhöhe die Frequenz von Gewinner um 1;
4. **IF** Ebene < DP **THEN** ( Ebene = Ebene+1; **GO TO** Schritt 3;);  
**ELSE GO TO** Schritt 2.

DP ist hierbei die Anzahl der Ebenen des hierarchischen neuronalen Netzes.

#### 4.2.2 Ordered Search Winner Update (OSWU)

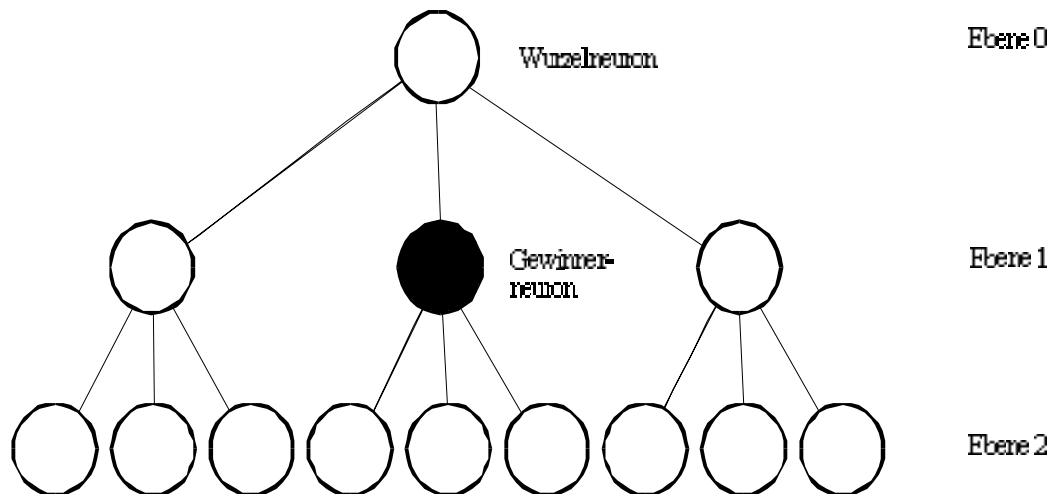


Abbildung 7 Schematische Darstellung der Aktualisierung beim OSWU Algorithmus.

Dieser Algorithmus unterscheidet sich nur minimal vom OSSU Algorithmus. Er differenziert sich nur durch das Weglassen des Schritts, in welchem die Gewichte der Kinderneuronen aktualisiert werden. Hier wird nur der Gewichtsvektor des ermittelten Gewinnerneurons aktualisiert. Der Schritt, in dem die Neurone im Unterzweig vom Gewinnerneuron aktualisiert werden (Subtree Update), fällt hier weg. Der Algorithmus ist zwar nicht so schnell wie der OSSU Algorithmus, da hier jeweils nur der Gewichtsvektor des Gewinnerneurons angepaßt wird, dafür ist er aber genauer, was die Passung an die Trainingsmuster angeht (Li et al., 1993). Die hierarchische Strukturierung der Trainingsmuster bleibt durch die geordnete Suche nach dem Gewinner dennoch erhalten, da die Struktur Ähnlichkeiten in den Gewichtsvektoren der Neurone eines Unterzweiges impliziert.

1. Initialisiere den Gewichtsvektor jedes Neurons auf einen Zufallswert.

2. Gewinner = Wurzelneuron;

Ebene = 1;

3. **FOR** jedes Trainingsmuster  $X_i = \langle x_0^i, x_1^i, \dots, x_{n-1}^i \rangle$  **DO**

a) AktGewinner = Wurzelneuron.

b) **FOR** AktEbene = 2 **TO** Ebenen **DO**

AktGewinner = Das Kind von AktGewinner das den geringsten Fehler hat;

c) Gewinner = AktGewinner;

d) Aktualisiere das Gewicht von Gewinner;

Erhöhe die Frequenz von Gewinner um 1;

4. **IF** Ebene < DP **THEN** ( Ebene = Ebene+1; **GO TO** Schritt 3;);

**ELSE GO TO** Schritt 2.

#### 4.2.3 Exhaustive Search Path Update (ESPU)

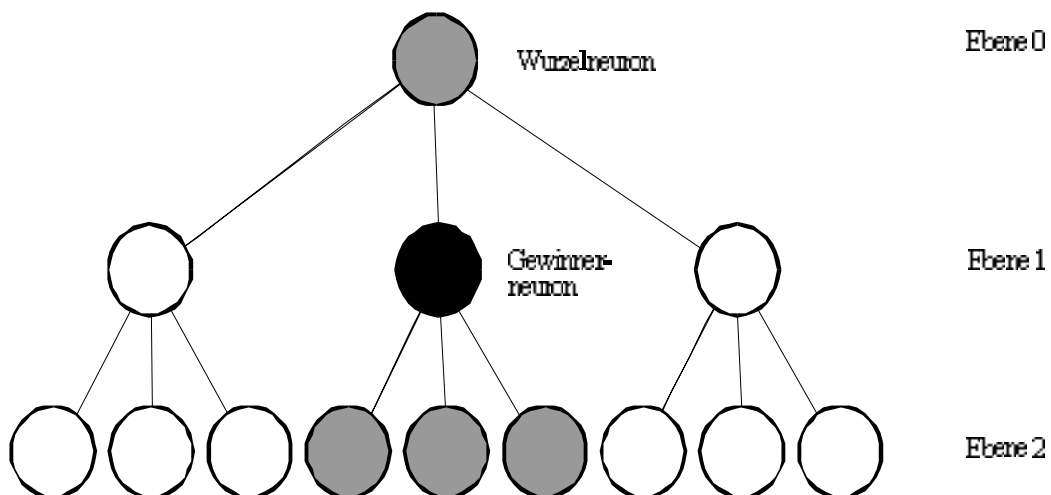


Abbildung 8 Aktualisierungsvorgang beim ESPU Algorithmus



Beim Exhaustive Search Path Update Algorithmus wird der gesamte Pool an Neuronen auf einer Ebene nach dem Neuron mit dem Gewichtsvektor durchsucht, dessen euklidische Distanz am minimalsten zum Eingabevektor ist. Daraufhin werden die Gewichtsvektoren des Gewinnerneurons und der darunterliegenden Neurone aktualisiert (Subtree Update). Außerdem werden auch die Gewichte sämtlicher Neurone zwischen dem Gewinnerneuron und dem Wurzelneuron aktualisiert, also jeweils die Gewichte des Elternneurons, bis man beim Wurzelneuron angelangt ist (Path Update).

1. Initialisiere den Gewichtsvektor jedes Neurons auf einen Zufallswert
2. Gewinner = Wurzelneuron;  
Ebene = 1;
3. **FOR** jedes Trainingsmuster  $X_i = \langle x_0^i, x_1^i, \dots, x_{n-1}^i \rangle$  **DO**
  - a) **FOR** jedes Neuron auf der Ebene AktEbene **DO** berechne fehler(n,i)
  - b) Gewinner = Das Neuron auf AktEbene mit dem geringsten Fehlerwert;
  - c) Aktualisiere das Gewicht jedes Neurons im Unterbaum von Gewinner;
  - d) Aktualisiere das Gewicht von Gewinner;  
Erhöhe die Frequenz von Gewinner um 1;
  - e) Aktualisiere die Gewichte jedes Neurons zwischen Wurzelneuron und Gewinner;
4. **IF** Ebene < DP **THEN** (Ebene = Ebene + 1; **GO TO** Schritt 3);  
**ELSE GO TO** Schritt 2.

### **4.3 Zusammenfassung**

An den oben dargestellten Algorithmen kann man sehr gut nachvollziehen, warum die Architektur eines hierarchischen neuronalen Netzes baumartig ist. Ähnliche Trainingsmuster werden auf denselben Unterzweigen des hierarchischen neuronalen Netzes abgebildet. Das heißt, in den Algorithmen selbst ist die Tendenz enthalten, die Trainingsmuster hierarchisch zu gruppieren. Dies geschieht nicht in der Art und Weise wie die Gewichte aktualisiert werden. Vielmehr ist es entscheidend, wie das Gewinnerneuron ermittelt wird und welche Neurone aktualisiert werden. Am offensichtlichsten ist dieser Zusammenhang beim OSSU Algorithmus. Es findet eine geordnete Suche entlang der Hierarchie des Netzes nach dem Gewinnerneuron statt und entsprechend der Hierarchie werden auch die Gewichtsvektoren angepaßt.

Beim OSWU Algorithmus ist dieser Zusammenhang weniger offensichtlich, da nur jeweils der Gewichtsvektor des Gewinnerneurons angepasst wird. Hier ist die Hierarchie in der geordneten Suche nach dem Gewinner enthalten. Hierbei ist es besonders wichtig, noch einmal die Rolle des Parameters  $\text{frequenz}(n.i)$  herauszustreichen, der verhindert, daß immer wieder dasselbe Neuron als Gewinner selektiert wird, sodaß die Suche entlang der Hierarchie zwangsläufig auch eine Aktualisierung der Gewichtsvektoren der Neurone in seinem Unterbaum nach sich ziehen wird.

Beim ESPU Algorithmus ist nun die Ermittlung des Gewinnerneurons für eine Hierarchiebildung weniger wichtig, da die Hierarchie des Netzes bei der Suche nach dem Gewinner hier keine Rolle spielt. Dafür wird eine Hierarchiebildung dadurch gefördert, daß jeweils der gesamte Unterzweig des Gewinners aktualisiert wird. Darüberhinaus werden noch die in der Hierarchie darüberliegenden Neurone bis zum Wurzelneuron

aktualisiert.

Welcher Algorithmus jedoch am sinnvollsten für einzelne Fragestellungen ist, läßt sich nur empirisch untersuchen, daher war es nötig, dem Anwender des Programms die Möglichkeit zu geben, zwischen allen Algorithmen entscheiden zu können.

## 5 Programmtechnische Implementation

Das Programm zur Simulation hierarchischer neuronaler Netze wurde in der Programmiersprache Delphi der Firma Borland implementiert. Die verwendete Version war die Version 3.0 des Entwicklungspaketes Delphi Professional. Dieses Entwicklungssystem wird am Fachbereich Psychologie der Justus-Liebig-Universität Gießen am häufigsten benutzt und baut auf der Programmiersprache Pascal auf, die in Forschungskreisen recht verbreitet ist. Die Entwicklung dieser RAD<sup>3</sup> Entwicklungsumgebung seitens des Herstellers scheint noch auf Jahre gesichert zu sein. Wegen der Verbreitung dieses Systems und der Zukunftssicherheit seitens der Marktverfügbarkeit fiel die Wahl auf diese Entwicklungsumgebung. Die verwendeten Einführungsbücher können auch dem interessierten Leser empfohlen werden, wenn noch eine Einarbeitung in das Entwicklungssystem erforderlich ist, um das vorliegende Programm speziellen Fragestellungen anzupassen (Cantu 1997; Matcho et al. 1996). Dieser Abschnitt richtet sich also die Entwickler, die sich mit der genauen Funktionsweise dieses Programmes auseinandersetzen möchten. Die Leser, die von einer Modifikation oder Weiterentwicklung dieses Systems absehen möchten, können gleich zu der Lektüre der Benutzeranleitung übergehen.

Es wurde der Versuch unternommen, dieses Programm hochmodular zu entwickeln. Dies wurde einerseits durch die Gruppierung des Programms in Funktionseinheiten, sogenannte Units, realisiert. Eine Unit ist ein Abschnitt des Programmcodes, der eine gewisse Funktion innerhalb des Programms erfüllen soll. So gibt es hier beispielsweise eine Unit, die die programmtechnische Formulierung eines Neurons enthält, die Unit Neuron. Die Unit Graph andererseits löst sich um das nicht unerhebliche Problem der Visualisierung des

---

3 RAD: Rapid Application Developement; Entwicklungswerkzeuge zum Erstellen von Anwendungen, die über eine graphische Schnittstelle mit dem Anwender interagieren.

hierarchischen neuronalen Netzes. Die Units und deren Funktionsweise werden im folgenden noch detailliert beschrieben.

Andererseits wurden objektorientierte Programmiertechniken angewandt, um die Weiterentwicklung zu vereinfachen. Die Unit Neuron enthält beispielsweise die Definition der logischen Einheit Neuron. Möchte ein Programmierer nun den Funktionsumfang dieser Einheit erweitern, so muß er sich nicht mit dem konkreten Programmcode auseinandersetzen, sondern definiert eine andere Logische Einheit, beispielsweise mit dem Namen ErweitertesNeuron und definiert, die Einheit ErweitertesNeuron soll alle Eigenschaften von der logischen Einheit Neuron erben. Diesen Vorgang nennt man Vererbung und er ist eine Eigenschaft von objektorientierten Programmiersprachen, die die Weiterentwicklung des Systems erheblich vereinfacht.

## **5.1 Struktur der Unterprogramme**

Um die nun folgenden Ausführungen verstehen zu können, ist es notwendig, Kenntnisse in der Programmiersprache Delphi<sup>4</sup> zu besitzen. Außerdem sollten dem Leser die Begriffe Objektorientierten Programmierung und Dynamische Speicherverwaltung bekannt sein. Im folgenden wird nur auf die Schlüsselfunktionen der einzelnen Units eingegangen, Funktionen und Prozeduren die selbsterklärend sind oder nur intern von anderen Prozeduren des Programms aufgerufen werden, ohne einen Bezug zur direkten Problematik der Implementation eines hierarchischen neuronalen Netzes zu haben bleiben hier unerwähnt. Sie sind jedoch im Anhang nachzulesen.

Das elementarste Objekt ist natürlich das Neuron, welches in der Unit Neuron deklariert wird. Die Neurone selbst sind nicht statisch definiert, sondern sie werden dynamisch zur

---

<sup>4</sup> Eingetragenes Warenzeichen der Firma Borland

Laufzeit im Speicher erzeugt. Instanzen dieses Objektes werden mittels dynamischer Pointerlisten vom Typ Tlist verwaltet. Dies dient dazu, die Größe des neuronalen Netzes nach oben offen zu halten. Die Größe des neuronalen Netzes ist praktisch nur vom zur Verfügung stehenden Arbeitsspeicher abhängig. Auf einem Computersystem mit 128 MB Arbeitsspeicher ist es ohne weiteres möglich, über 10.000 Neurone zu verwalten.

Die Trainingsmuster werden in der Unit datau verwaltet. Sie stehen als Vektoren zeilenweise in einem Datensatz, der entweder im DBase oder Paradox Datenbankformat vorliegen muß. Auch diese Trainingsmuster werden dynamisch verwaltet, um möglichst flexibel bezüglich der Dimensionalität oder der Anzahl der Trainingsmuster zu sein.

Das Training der Neurone wird durch die Unit Trainer erledigt, welche die Trainingsmuster beim Training und die Aktualisierung der Gewichtsvektoren der Neuronen verwaltet. In dieser Unit sind die Algorithmen OSSU, OSWU, ESPU implementiert.

Dies zur Grobstruktur des Programms. Nun folgt die Dokumentation zu den einzelnen Units. Die Reihenfolge besprochener Funktionen und Prozeduren hält sich an die Reihenfolge der Deklaration im Sourcecode, der im Anhang zu finden ist. Jede Funktion oder Prozedur ist mit einer Seitenangabe und einer Zeilenangabe, in der Form (Seite,Zeile) versehen. Dies ist dazu gedacht, die korrespondierenden Stellen im Anhang zu bezeichnen.

## **5.2 Unit Main**

Diese Unit definiert das Hauptprogrammformular und ist im wesentlichen zur Initialisierung eines neuronalen Netzes beim Programmstart gedacht.

### **5.2.1 Wichtige Konstanten und Variablen**

**NUMNEURONS:** legt die Anzahl der Neurone fest, die in dem Netz enthalten sind,

welches beim Programmstart initialisiert wird. Es ist nicht notwendig, diese Konstante zu ändern, da das am Anfang initialisierte Netz nur dazu dient, ein elementares neuronales Netz zu definieren. Dies geschieht, damit sofort alle Units funktionsfähig und ansprechbar sind. Sonst könnte es zu Speicherfehlern kommen, wenn z.B. Routinen aus der Unit Graph unmittelbar nach dem Programmstart versuchen, auf ein Netz zuzugreifen, welches noch gar nicht initialisiert wurde. Dieses anfängliche Netz wird gelöscht, nachdem der Anwender eine neue Architektur definiert hat.

### **5.2.2 TForm1.Diagramm\_Drucken1Click(Sender: TObject); (1,50)**

Diese Prozedur gibt den aktuellen Inhalt des Graphikfensters, in dem die Neurone dargestellt werden, auf einem angeschlossenen Drucker aus. Dazu wird die Prozedur `diagramm.drawdiagramm` aufgerufen. Hierzu wird jedoch nicht das neuronale Netz auf den Bildschirmcanvas gezeichnet, sondern auf ein Canvas<sup>5</sup> das dem aktuellen Drucker zugeordnet ist.

### **5.2.3 TForm1.FormCreate (Sender: TObject); (1,68)**

Beim Programmstart wird diese Prozedur automatisch aufgerufen und initialisiert das Tlist Objekt `neurons`, welches letztendlich die Liste zur Verwaltung aller Neurone darstellt. Der Zufallszahlengenerator wird neu initialisiert, um Wiederholungseffekte bei der Initialisierung der Gewichtsvektoren der Neurone zu vermeiden.

### **5.2.4 TForm1.FormDestroy(Sender: TObject); (2,81)**

Diese Prozedur wird immer am Programmende aufgerufen und entfernt alle Neurone, die während der Laufzeit des Programmes erzeugt worden sind, vollständig aus dem Speicher.

---

5 In der Programmiersprache Delphi verwendeter Ausdruck für eine Fläche auf der man Zeichenoperationen ausführen kann.

## 5.3 Unit Neuron

Diese Unit bildet eines der Kernstücke des Programms. Hier wird das grundlegende Objekt Neuron definiert. Innerhalb des Objektes Neuron werden die Verwandtschaftbeziehungen der Neurone untereinander verwaltet. Ein Neuron besitzt einen Gewichtsvektor mit dem Namen Weight und Koordinaten mit der Bezeichnung Koords. Es ist wichtig, zwischen beiden Konzepten zu unterscheiden. Der Gewichtsvektor stimmt nicht notwendigerweise mit den Koordinatenangaben für das Neuron überein. Koords ist der Vektor für die graphische Darstellung der Lage des Gewichtsvektors des Neurons.

### 5.3.1 Wichtige Konstanten und Variablen

**NEURON\_DURCHMESSER:** definiert wie groß die Darstellung des Gewichtsvektors des Neurons in der Graphik sein soll. Die Angabe erfolgt in Prozent der zur Verfügung stehenden Zeichenfläche.

**DEFAULT\_NAME:** definiert den Standardnamen eines Neurons. Der Name eines Neurons ist ein String und für verschiedene Anwendungsfelder kann es sinnvoll sein, den Neuronen unterschiedliche Namen zu geben.

**Weight:** ist der Gewichtsvektor eines Neurons, er kann eine beliebige Dimensionalität haben. Die Dimensionalität von Weight wird während der Laufzeit, ausgehend von der Dimensionalität der Trainingsmuster angepasst.

**Err:** ist der akkumulierte durchschnittliche Fehler des Neurons während des Trainings.

**Kinder:** ist eine Liste der Kinder dieses Neurons. Da auch hier ein Tlist Objekt zugrunde liegt, kann ein Neuron beliebig viele Kinder haben.

**Eltern:** ist die Liste der Eltern eines Neurons. Bis jetzt kann ein Neuron allerdings nur ein



Elternteil haben. Möchte man eine Architektur mit mehreren Eltern definieren, müßten umfangreiche Anpassungen am Sourcecode vorgenommen werden und es wäre fraglich, ob die Algorithmen OSSU, OSWU, ESPU korrekt mit einem solchen Netz arbeiten würden.

**Koords:** sind die Koordinaten des Gewichtsvektors eines Neurons. Sie können bis zu drei Dimensionen enthalten. Da aber zur Zeit nur eine zweidimensionale graphische Visualisierung implementiert ist, werden nur die ersten beiden Koordinaten von der Unit Graph zu Visualisierung verwendet. Die Koordinaten liegen in Form von X und Y im euklidischen Raum vor.

**Name:** ist der Name des Neurons als String.

**Number:** ist die Nummer des Neurons. Die Neurone werden fortlaufend, vom Wurzelneuron ausgehend numeriert.

**Level:** bezeichnet die Ebene, auf der das Neuron liegt. Die Anzahl der zur Verfügung stehenden Ebenen ist in der aktuellen Implementation auf 2.147.483.648 Ebenen beschränkt, was jedoch die Anzahl sinnvoll zu verwaltender Neurone übersteigt, sodaß diese Begrenzung rein theoretischer Natur ist.

**Rwinkel:** bei höherdimensionalen Trainingsmustern ist es notwendig, die graphische Präsentation der Neurone als Diagramm im Polarkoordinatensystem (Unit Polar) darzustellen. Rwinkel definiert hierbei den Raumwinkel, den ein Neuron in Anspruch nehmen darf.

**Frequency:** gibt die Frequenz des Neurons an, d.h. wie oft das Neuron während des Trainings als Gewinner selektiert wurde. Diese Variable fließt unmittelbar in die Gewichtsberechnung nach Formel (8) ein.

**Neurons:** Die Variable Neurons, enthält die Liste aller Neuronen im Hierarchischen neuronalen Netz. Auf diese Liste wird während der Laufzeit immer wieder zugegriffen.

### **5.3.2 TNeuron.Create(Gewichtsdimensionen, Zeichendimensionen, Nummer : Longint); (3,70)**

Diese Prozedur initialisiert das Neuron. Die Eltern- und Kindlisten werden im Speicher generiert und die wichtigsten Variablen des Neurons werden initialisiert. Wichtig ist die Übergabe des Parameters Gewichtsdimensionen, da hiermit die Dimensionalität des Gewichtsvektors eines Neurons festgelegt wird. Der Parameter Zeichendimensionen ist zum jetzigen Zeitpunkt auf 2 zu setzen. Der Parameter Nummer ist identisch mit der Indexnummer des Neurons in der Liste neurons.

### **5.3.3 TNeuron.Done; (4,95)**

Diese Prozedur wird aufgerufen, wenn man ein Neuron aus dem Speicher löschen möchte. Die Prozedur gibt nicht nur den Speicher frei den das Neuron belegt hat, sondern die Hierarchie des neuronalen Netzes wird dadurch gewahrt, daß die Kinder dieses Neurons ein neues Elternneuron bekommen. Da das zu löschende Neuron nicht mehr existiert, werden dem Elternneuron des zu löschenden Neurons die Kinder desselben zugeordnet. Dies ist sinnvoll, wenn man Neurone aus der Architektur zwecks Optimierung löschen möchte (Li et al, 1993).

### **5.3.4 TNeuron.Writekoords(k:array of TWeigt); (4,140)**

Hier hat der Programmierer die Möglichkeit externe Koordinatenangaben im euklidischen System dem Neuron bekanntzugeben. Dies wird überwiegend von der Unit Graph in Anspruch genommen, wenn einer Projektion von höherdimensionalen Gewichtsvektoren auf den zweidimensionalen Raum erforderlich ist.

### **5.3.5 TNeuron.readkoords(var k:array of TWeight); (5,151)**

Hier hat der Programmierer die Möglichkeit, sich die aktuellen Koordinaten des Neurons in eine Variable, die beim Prozeduraufruf übergeben wird, ausgeben zu lassen.

### **5.3.6 TNeuron.RandomNumber(min, max: TWeight) : TWeight; (5,178)**

Diese Funktion ist zum Initialisieren eines Gewichtsvektors auf einen Zufallswert gedacht. Die Ausgabe der Funktion kann direkt an die Variable weight des Neurons übergeben werden, um einen zufälligen Gewichtsvektor zu bekommen. Dies ist notwendig, um das neuronale Netz für den ersten Lernschritt zu initialisieren (siehe Abschnitt Methodik). Die Parameter min bzw. max sind hierbei die Grenzen, in denen Zufallswerte zurückgeliefert werden sollen. In der aktuellen Implementation ist min jeweils das kleinste und max das größte in den Trainingsmustern vorkommende Element.

### **5.3.7 TNeuron.getlev : Longint; (5,201)**

Diese Funktion liefert die Ebene zurück, auf der sich das Neuron befindet. Die Ebene wird jeweils zur Laufzeit ermittelt. Dies geschieht, indem durchgezählt wird, wieviele Neurone in der Hierarchie oberhalb des aktuellen Neurons liegen.

### **5.3.8 TNeuron.error(vec : TVector) : double; (6,216)**

Diese Funktion ist zur Bestimmung des Fehlerwertes des Gewichtsvektors eines Neurons zu einem Trainingsmuster vec notwendig. Die Bestimmung des Fehlerwertes erfolgt nach Formel (8). Der Fehlerwert ist die euklidische Distanz zwischen dem Trainingsmuster und dem Gewichtsvektor des Neurons, multipliziert mit der Frequenz des Neurons.

### **5.3.9 TNeuron.update(vec : TVector); (6,232)**

Hier wird die Lernfunktion eines Neuron nach Formel (9) implementiert. Der Lernparameter Alpha wird direkt aus der Unit Param übernommen. Eingegeben wird ein Trainingsmuster vec, zu dem das Neuron seinen Gewichtsvektor aktualisieren soll.

### **5.3.10 TNeuron.ischild(n : TNeuron) : boolean; (6,247)**

Diese Funktion liefert eine Antwort auf die Frage, ob das aktuelle Neuron das Kind von Neuron n ist. Dies ist sinnvoll, wenn man nach einer Umorganisation des Netzes, z.B. durch Ausschneiden diverser Neurone (TNeuron.Done) sichergehen möchte, daß die hierarchische Struktur des Netzes gewahrt geblieben ist.

## **5.4 Unit Param**

Die Unit Param reagiert auf Benutzerinteraktionen. Über diese Unit wird die Architektur eines hierarchischen neuronalen Netzes vom Anwender festgelegt. Die Unit löscht dann das vorhandene Netz aus dem Speicher und erstellt ein neues. Mit der vom Anwender gewünschten Architektur. Zur Konstruktion eines hierarchischen neuronalen Netzes sind zwei Parameter entscheidend: die Anzahl der Kinder pro Neuron und die Anzahl der Ebenen im Netz. Ausgehend von diesen beiden Parametern wird ein entsprechendes hierarchisches neuronales Netz generiert. Es wird Speicher für Neurone reserviert und die Neurone werden entsprechend der vom Anwender gewünschten Architektur miteinander verbunden und die Gewichtsvektoren werden initialisiert. Weiterhin verwaltet diese Unit den Parameter Alpha, also die Lernrate des neuronalen Netzes.

Die zentralen Prozeduren sind Button1Click, Button2Click und Button3Click. Button3Click berechnet die Anzahl der benötigten Neurone für eine Architektur, die sich aus

den Eingabefeldern des Formulars dieser Unit herleiten läßt. `Button1Click` löscht das alte Netz und reserviert entsprechend Speicher für ein neues Netz. `Button2Click` ermittelt die Verwandtschaftsbeziehungen unter den Neuronen und initialisiert die Neurone entsprechend.

#### 5.4.1 Wichtige Variablen

**Gewdim:** ist die Dimensionalität der Gewichtsvektoren. Dieser Parameter ist für die initialisierung neuer Neurone wichtig, da die Dimensionalität der Gewichtsvektoren aus der Dimensionalität der Trainingsmuster ermittelt werden muss.

**DP:** die Tiefe des hierarchischen neuronalen Netzes, also die Anzahl der Ebenen.

**AnzNeurone:** die Anzahl der Neurone im gesamten hierarchischen neuronalen Netz.

**KinderZahl:** die Anzahl der Kinder pro Neuron.

**ZeichDimensionen:** die Dimensionalität der graphischen Darstellungsebene, bisher muß `ZeichDimensionen` immer den Wert zwei haben !

**Alpha:** die Lernrate Alpha des Netzes, dieser Parameter fließt in die Gewichtsaktualisierung nach Formel (9) ein.

**Arity:** eine andere Variable für die Tiefe des hierarchischen neuronalen Netzes. `Arity` ist feststehend, während `DP` programmintern von einigen Routinen beeinflusst wird.

**GewDimensionen:** eine sogenannte Property. Dies ist eine Variable, die die Variable vom `GewDim` kapselt, da `gewdim` eine objektinterne Variable ist.

#### 5.4.2 `TParamForm.FormCreate (Sender:TObject); (8,67)`

Diese Prozedur wird automatisch beim Programmstart ausgeführt. Die für den Programmstart vorgesehenen Werte, die in den Eingabefeldern des Formulars stehen, werden in die korrespondierenden Variablen eingelesen.

#### **5.4.3 TParamForm.Button1Click(Sender:TObject); (9,79)**

Diese Prozedur ist dafür zuständig die alte Neuronenliste zu löschen und deren Speicher freizugeben. Danach wird eine neue Liste von Neuronen angelegt und Speicher für diese reserviert, die Gewichtsvektoren der Neurone werden initialisiert. Dies geschieht, dadurch daß die Dimensionalität und die Extremwerte der Trainingsmuster abgefragt werden und die Gewichtsvektoren entsprechend mit Zufallswerten gefüllt werden.

#### **5.4.4 TParamForm.AlphaEditExit(Sender:TObject); (9,129)**

Wird der Wert des Parameters Alpha vom Benutzer im korrespondierenden Eingabfeld des Formulars geändert, so findet zunächst eine Bereichsüberprüfung statt, dann wird die interne Variable alpha auf diesen Wert gesetzt. Die Bereichsüberprüfung ist sinnvoll, da für alpha gilt  $0 < \alpha < 1$ .

#### **5.4.5 TParamForm.Button2Click(Sender:TObject); (10,157)**

Während die Prozedur 5.4.3 rein die Speicherverwaltung der Neuronenliste implementiert, ist diese Prozedur nun dazu da, die Hierarchien unter den Neuronen festzulegen. Das heißt, jedes Neuron aus der Liste aufzurufen und ihm mitzuteilen, welches Neuron sein Elternteil ist und welchen Neurone seine Kinder sind. Dies geschieht folgendermaßen:, es ist eine Liste mit Neuronen vorhanden und es ist bekannt, wieviele Kinder jedes Neuron haben soll. Ausgehend vom Wurzelneuron wird nun ein hierarchisches neuronales Netz aufgebaut, indem ein Neuron nach dem anderen aus der Liste genommen wird, bis man die Zahl der Kinder des Wurzelneurons erreicht hat. Auf der nächsten Ebene wird analog verfahren. Das hierarchische neuronale Netz wird also vom Wurzelneuron ausgehend Ebene für Ebene aufgebaut.

#### **5.4.6 TParamForm.spaceonlevel(level:Longint) : LongInt; (11,263)**

Diese Funktion wird besonders von der Prozedur 5.4.5 gebraucht. Es ist bekannt, wieviele Kinder jedes Neuron haben soll. Nun stellt sich die Frage, wieviel Neurone auf einer bestimmten Ebene Platz haben. Diese Funktion rechnet dies aus und liefert den entsprechenden Wert zurück.

#### **5.4.7 TParamForm.Button3Click(Sender:TObject); (11,275)**

Diese Prozedur ist eine Schlüsselprozedur zur Erstellung einer flexiblen Hierarchie. Ausgehend von der gewünschten Tiefe des hierarchischen neuronalen Netzes und der gewünschten Anzahl von Kindern pro Neuron, wird der Gesamtbedarf an Neuronen für diese Architektur ermittelt. Daraufhin wird die Prozedur Button1Click (5.4.3) aufgerufen, die den entsprechenden Speicher reserviert und die Prozedur Button2Click (5.4.5) baut nun die Hierarchie des gewünschten Netzes auf.

### **5.5 Unit Polaru**

Polaru wurde als Bibliothek von Prozeduren und Funktionen konzipiert, die dazu dienen, Koordinatensysteme ineinander abzubilden. Die Unit kann nur mit zweidimensionalen Koordinatensystemen arbeiten. Es geht hier speziell um das kartesische Koordinatensystem und das Polarkoordinatensystem. Beim Kartesischen Koordinatensystem wird ein Punkt dadurch definiert, daß man seine Position als Abweichung vom Ursprungspunkt in vertikaler bzw. horizontaler Richtung angibt. Die Vorgehensweise entspricht dem, von der Computergraphik her bekannten Prinzip, der Adressierung von Punkten über eine X und eine Y Achse. Das Polarkoordinatensystem hingegen beschreibt die Lage von Punkten in der Ebene anders. Vom Ursprungspunkt ausgehend wird der Abstand und der Winkel eines

Punktes bestimmt, den er zu dem Ursprungspunkt aufweist. Diese Darstellung der Koordinaten eines Punktes ist für die Unit Graph wichtig, wenn es darum geht, höherdimensionale Gewichtsvektoren auf die beiden Dimensionen der Zeichenebene zu projizieren. Die Bibliothek beschreibt ein Objekt, einen Punkt im polaren Koordinatensystem, der durch die Parameter Phi (Winkel) und Rho (Abstand) spezifiziert wird.

### 5.5.1 Wichtige Variablen

**Phi:** der Winkel, den der zu beschreibende Punkt mit dem Ursprungspunkt und der Ursprungsachse des Koordinatensystems bildet.

**Rho:** der Abstand, den der zu beschreibende Punkt von dem Ursprungspunkt hat.

### 5.5.2 TPolar.phidegread : TDataFormat; (13,39)

Diese Funktion liefert den Parameter phi in der Darstellung des 360°-Winkelsystems zurück. Intern wird phi im Winkelsystem  $0 < x < 2\pi$  repräsentiert, zurückgegeben wird ein Wert im Winkelsystem  $0^\circ < x < 360^\circ$ .

### 5.5.3 TPolar.xread : TDataFormat; (13,51)

Diese Funktion liefert den X-Wert des Punktes im kartesischen Koordinatensystem.

### 5.5.4 TPolar.yread:TDataFormat; (13,56)

Diese Funktion arbeitet analog zu der Funktion xread, mit dem Unterschied daß hier der Y-Wert des Punktes im kartesischen Koordinatensystem zurückgeliefert wird.

### 5.5.5 Tpolar.xypointread : TRPoint; (13,61)

Hat man ein Objekt vom Typ TRPoint, welches ein Punkt im kartesischen Koordinatensystem beschreibt und will ihm die Koordinaten des Punktes im Polarkoordinatensystem übergeben, so reicht es, diese Funktion aufzurufen, welche dem Objekt vom Typ TRPoint die entsprechenden Werte im kartesischen Koordinatensystem zuweist.



### **5.5.6 TPolar.xypointwrite (p:TRPoint); (13,67)**

Hat man ein Objekt p, welches einen Punkt im kartesischen Koordinatensystem darstellt, so kann man ihn an diese Prozedur übergeben und seine Koordinatenangaben werden in das Polarkoordinatensystem umgerechnet und dem aufgerufenen Objekt übergeben.

### **5.5.7 TPolar.xyset(x,y:TDataFormat); (14,88)**

Sind dem Programmierer Koordinatenangaben im kartesischen Koordinatensystem bekannt, und er möchte ein Objekt vom Typ TPolar damit initialisieren, so ruft er diese Prozedur auf und das Objekt wird mit den entsprechenden kartesischen Koordinaten initialisiert, die natürlich intern in das Polarkoordinatensystem umgerechnet werden.

## **5.6 Unit Trainer**

Die Unit Trainer implementiert die eigentlichen Lernalgorithmen (siehe Abschnitt: Methodik). Die Algorithmen OSSU (Ordered Search Subtree Update) und OSWU (Ordered Search Winner Update) sind in einer Routine zusammengefaßt, da der Algorithmus OSWU sich nur im Weglassen der Aktualisierung des jeweiligen Unterzweiges von OSSU unterscheidet. Diese Unit verfügt auch über ein Formular, welches Benutzereinteraktionen zuläßt, beispielsweise einen Button um das Training zu starten und einen Button um es zu stoppen.

### **5.6.1 Wichtige Deklarationen und Variablen**

**TrainMode = (OSSU,OSWU,ESPU);**

Diese Deklaration vom Typ enum definiert drei Zustände einer Variablen, die anzeigen soll auf welchen Trainingsmodus das Programm gerade eingestellt ist.

**Epc:** diese Variable ist der Epochenzähler, an ihr kann man die Anzahl der bisher durch-

laufenen Epochen ablesen. Dies wird dem Anwender des Programms ohnehin in dem Trainer Formular angezeigt.

**Tmode:** ist eine Variable des Typs TrainMode, welche den aktuellen zu rechnenden Algorithmus (OSSU,OSWU,ESPU) anzeigt.

**Learn:** ist eine Boolesche Variable, die anzeigt, ob das Netz trainiert werden soll, d.h. die Gewichtsvektoren angepaßt werden sollen oder ob nur die Reaktion des Netzes auf Eingabemuster getestet werden soll. Dies ist dann die Ermittlung der Gewinner ohne daß eine Aktualisierung der Gewichtsvektoren stattfindet.

**Zeichnen:** repräsentiert den Wert einer Checkbox im Formular. Diese Variable teilt der Unit Graph mit, ob nach jeder Epoche der Status des Netzes graphisch dargestellt werden soll oder ob ein Training ohne graphische Ausgabe stattfinden soll.

**Winlist:** ist eine Liste mit Pointern auf Neurone, die die Gewinner der aktuellen Epoche sind. Die Unit Graph greift auf diese Liste zu, um die Gewinner graphisch darzustellen.

### 5.6.2 TTrain.OSSU\_OSWU\_Epoche; (15,70)

Diese Prozedur führt das Training einer OSSU oder OSWU Epoche durch. Die Ermittlung der Gewinner für beide Algorithmen verläuft nach demselben Schema und unterscheidet sich nur durch die Aktualisierung gewisser Neurone. Die Implementation der Algorithmen richtet sich nach dem schon im Methodikteil vorgestellten Schema. Es gibt nur einen pascaltypischen Unterschied, der darin besteht, daß die IF ... GO TO Anweisung durch eine REPEAT UNTIL Schleife nachmodelliert wurde. Die Prozedur arbeitet intern mit zwei Listen von Neuronen, um die Aktualisierung der Gewichtsvektoren im Unterzweig (Subtree Update) durchzuführen. Eine der Listen (childlist) hat die Funktion, die Kinder des Gewinnerneurons zu verwalten. Die zweite Liste (bufferlist) enthält die Liste der

Kinder der Neurone von childlist. Die Neurone der childlist werden aktualisiert, danach wird die bufferlist in die childlist kopiert und eine neue bufferlist anhand der nun veränderten childlist angelegt. Wieder erfolgt eine Aktualisierung der Neurone in childlist. Dieser Zyklus wird solange durchlaufen, bis man auf der letzten Ebene des hierarchischen neuronalen Netzes angekommen ist. Dieser Schritt entfällt natürlich im Falle des OSWU Algorithmus, da hier nur das Gewinnerneuron aktualisiert wird.

### **5.6.3 TTrain.ESPU\_Epoche (17,162)**

Diese Prozedur leitet das Training nach dem ESPU-Algorithmus ein (siehe Abschnitt 4). Auch hier wurde das IF ... GO TO Konstrukt durch eine REPEAT UNTIL Schleife nachmodelliert. Diese Prozedur arbeitet wie Prozedur 5.6.2 mit zwei Listen childlist und bufferlist, um nach dem in 5.6.2 beschriebenen Algorithmus die Aktualisierung der Neurone im Unterzweig von Gewinner abzuarbeiten. Das Path Update, also die Aktualisierung der Neurone auf dem direkten Pfad vom Gewinnerneuron zum Wurzelneuron ist recht einfach implementiert. Vom Gewinnerneuron ausgehend wird das Elternneuron ermittelt und dessen Gewichtsvektor aktualisiert. Dann findet der Algorithmus das Elternneuron des Elternneurons des Gewinnerneurons und aktualisiert dessen Gewichtsvektor. Diese Schleife wird so oft durchlaufen, bis der Algorithmus bei einem Neuron ohne Elternneuron angekommen ist. Der impliziten Logik der Architektur von hierarchischen neuronalen Netzen folgend, muß dies das Wurzelneuron sein und die Schleife wird beendet.

### **5.6.4 TTrain.RecordButtonClick(Sender:TObject); (18,256)**

Diese Prozedur reagiert auf den Anwenderwunsch, das Training zu starten. Es wird abgefragt, nach welchem Algorithmus trainiert werden soll, um dann in die entsprechende Trainingsunterroutine (5.6.2 oder 5.6.3) zu verzweigen. Außerdem werden nach jeder

Trainingsepoche die Anwendereingaben verarbeitet, um das Programm nicht in einer Endlosschleife einzufrieren und auf Änderungswünsche des Anwenders reagieren zu können.

#### **5.6.5 TTrain.FormCreate(Sender:TObject); (18,269)**

Beim Programmstart wird diese Prozedur automatisch aufgerufen, um die internen Variablen der Unit zu initialisieren.

#### **5.6.6 Sonstiges**

Die anderen Prozeduren dieser Unit dienen nur noch dazu, auf Benutzereingaben auf der Formularebene zu reagieren und interne Variablen des TTrain Objektes zu verändern. Diese Prozeduren sind so einfach zu verstehen, daß eine nähere Erläuterung hier trivial wäre.

### **5.7 Unit Vecmat**

Die Unit Vecmat deklariert ein Objekt des Typs Tvector. Es handelt sich hierbei um die Implementation eines n-dimensionalen Vektors und der zugehörigen wichtigsten Rechenoperationen aus der linearen Algebra. Der Parameter n, also die Dimensionalität des Vektors, ist auf 2.147.483.647 Komponenten beschränkt, was aber für alle praktischen Problemstellungen mehr als ausreichend sein dürfte. Die Vektoren dürften also jede Größe annehmen können, die in einer praktischen Fragestellung noch relevant wäre. Die Implementation dieser Unit wurde durch die Anpassung einer C++ Unit auf Pascal vorgenommen (Blum, 1992). Die Algorithmen lehnen sich stark an die an, welche von Adam Blum (1992) beschrieben wurden.

### 5.7.1 Wichtige Deklarationen

**TWeight = double;**

TWeight ist die Deklaration einer Komponente eines Vektors. Es erschien zweckmäßig, diese Komponenten mit doppelter Fließkommagenauigkeit zu berechnen, um auch Trainingsmuster die sehr nahe beieinander liegen, numerisch sinnvoll trennen zu können.

### 5.7.2 TVector.create; (20,56)

Dies ist der Konstruktor eines TVector Objektes, der jedesmal aufgerufen werden muß, wenn ein neues Objekt vom Typ TVector im Speicher erstellt werden soll.

### 5.7.3 TVector.done; (20,62)

Der Destruktor eines TVector Objektes. Im Gegensatz zum Konstruktor muß dieser nicht explizit aufgerufen werden. Diese Prozedur wird automatisch von Delphi aufgerufen, sobald man ein Objekt vom Typ TVector aus dem Speicher löscht. Die Prozedur löscht ihrerseits die Komponenten des TVector Objektes aus dem Speicher.

### 5.7.4 TVector.GetDim : LongInt; (21,73)

Diese Funktion liefert die Dimensionalität (Anzahl der Komponenten) des TVector Objektes zurück.

### 5.7.5 TVector.Clean; (21,78)

Diese Prozedur löscht alle Komponenten des aktuellen TVector Objektes und gibt deren Speicher frei.

### 5.7.6 TVector.read(index:LongInt) : TWeight; (21,83)

Will man den Wert einer Komponente eines TVector Objektes auslesen, so ruft man diese Funktion auf und gibt über die Variable index an, welche Komponente die Funktion zurückliefern soll.

#### **5.7.7 TVector.read\_all (var weights : array of TWeight); (21,95)**

Will man alle Gewichte eines TVector Objektes auf einmal auslesen, so übergibt man dieser Prozedur ein Array, hier weights genannt, und alle Komponenten werden in das Array kopiert.

#### **5.7.8 TVector.write(index : LongInt; weight:TWeight); (21,116)**

Um eine einzelne Komponente eines TVector Objektes zu verändern, übergibt der Programmierer deren Index (index) und den Wert (weight), den die Komponente nun besitzen soll.

#### **5.7.9 TVector.write\_all(weights : array of TWeight); (21,131)**

Der Programmierer übergibt dieser Prozedur ein Array von Komponenten, die nun zu den Komponenten des TVector Objektes werden sollen. Die Prozedur kopiert diese Komponenten in das Objekt und paßt die Dimensionalität des Objektes an die Dimensionalität des Arrays an.

#### **5.7.10 TVector.setdim(dim:LongInt); (22,152)**

Diese Prozedur ändert die Dimensionalität eines TVector Objektes. Wird die Dimensionalität verringert, so werden die letzten Komponenten des Objektes gelöscht. Wird die Dimensionalität hingegen erhöht, so werden neue Komponenten angefügt und mit dem Wert 0.0 initialisiert.

#### **5.7.11 TVector.plus(b:TVector; var erg : TVector); (22,180)**

Um eine Vektoraddition durchzuführen ruft man diese Prozedur auf.

Der Syntax lautet:

var

a, b, sum : TVector;

begin

a.plus(b,sum);

#### **5.7.12 TVector.minus(b:TVector; var erg : TVector); (22,195)**

Um einen Vektor von einem anderen zu subtrahieren, wird diese Funktion benötigt.

Syntax:

var

a, b, sub : TVector;

begin

a.minus(b,sub);

#### **5.7.13 TVector.skalar(b:TVector) : TWeight; (22,208)**

Diese Funktion bildet das Skalarprodukt aus zwei Vektoren.

Syntax:

var

a, b : TVector;

erg : TWeight;

begin

erg := a.skalar(b);

#### **5.7.14 TVektor.mult(b:TWeight; erg : TVector); (23,224)**

Diese Prozedur multipliziert einen Vektor mit einem Skalar.

Syntax:

var

a, erg : TVector;

b : TWeight;

begin

a.mult(b,erg);

## **5.8 Unit Datau**

Diese Unit ist für das Einlesen und die Konvertierung von Trainingsmustern geschrieben. Alle Routinen dieser Unit verwenden Delphi interne Routinen zum Datenbankzugriff. Es werden Eingabedateien in den Datenbankformaten Paradox und DBase unterstützt. Diese Dateiformate werden auch von den meisten anderen Anwendungen, wie z.B. Excel<sup>6</sup>, unterstützt. Eine Konvertierung von Trainingsmustern in diese Formate dürfte daher kein Problem sein. Das Format der Trainingsmuster ist sehr einfach, das erste Feld der Datensätze enthält ein Label (String) zu jedem Trainingsmuster, gefolgt von den Feldern, die die einzelnen Komponenten der Vektoren, die die Trainingsmuster beschreiben, enthalten. Die Unit ist auch für die Festlegung der unteren bzw. oberen Grenzen zur Initialisierung der Gewichtsvektoren der Neurone zuständig.

### **5.8.1 Wichtige Variablen**

**vec\_min:** enthält die kleinste Komponente aus den eingelesenen Trainingsmustern. Dieser Wert stellt die Untergrenze bei der initialisierung von Gewichtsvektoren mit Zufallswerten dar.

**vec\_max:** verhält sich analog zu vec\_min, mit dem Unterschied, daß hier die Obergrenze der Initialisierung von Trainingsmuster enthalten ist.

**Vectors:** die Liste der Trainingsmuster, diese sind Objekte vom Typ TVector.

---

6 Eingetragenes Warenzeichen der Firma Microsoft



### **5.8.2 Tdata.Einlesen1Click(Sender:TObject); (24,44)**

Diese Prozedur reagiert auf die Benutzeranforderung eine Eingabedatei zu öffnen. Der Dateiname wird ermittelt, die Datei wird geöffnet und die Konvertierung der Datenbankwerte in Objekte des Typs TVektor wird eingeleitet.

### **5.8.3 Tdata.done; (24,70)**

Dies ist der Destruktor eines Tdata Objektes. Es mußte ein spezieller Destruktor implementiert werden, um sicherzustellen, daß die Liste der Trainingsmuster auch aus dem Arbeitsspeicher gelöscht wird.

### **5.8.4 Tdata.FormCreate(Sender:TObject); (25,63)**

Diese Prozedur wird automatisch beim Programmstart aufgerufen. Es werden die Variablen vec\_min und vec\_max initialisiert und die Liste der Trainingsmuster wird angelegt.

### **5.8.5 Tdata.DatenInVektoren; (25,90)**

Dies ist die eigentliche Konvertierungsprozedur. Die die Umwandlung von Datenbankwerten in die Liste der Trainingsmuster implementiert. Die Umwandlungsroutine selbst ist so implementiert, daß die Datenbank Datensatz für Datensatz ausgelesen wird, bis das Ende der Datei erreicht ist. Das erste Feld wird übersprungen, da es nur die Labels enthält. Die folgenden Felder eines Datensatzes werden in die Komponenten eines Objektes vom Typ TVektor umgewandelt. Ist das TVektor Objekt komplett, so wird es in die Liste der Trainingsmuster übertragen.

Während dieser Schleife werden auch die Werte für vec\_min und vec\_max ermittelt. Ist eine Komponente kleiner als der aktuelle Wert von vec\_min, so wird dies zu dem neuen Wert von vec\_min, bis das Dateiende erreicht ist. Analog dazu wird auch die Obergrenze vec\_max ermittelt.

## **5.9 Unit Neuron\_e**

Die Unit Neuron\_e enthält einige wichtige Deklarationen zu Exceptions die während des Programmablaufs auftreten können. Diese Unit wurde konzipiert, um ein Debugging zu erleichtern. Der Programmierer sieht bei einer Fehlermeldung, welche Art von Exception ausgelöst wurde. Das Programm selbst arbeitet mit einigen Routinen, die die hier definierten Exceptions auslösen.

### **5.9.1 Wichtige Deklarationen**

#### **EGewichte\_Bereichsverletzung**

Diese Exception wird ausgelöst, wenn das Programm versucht, auf eine Komponente des Gewichtsvektors eines Neurons zuzugreifen, die nicht existiert. Wenn beispielsweise ein zwanzigdimensionalen Gewichtsvektor vorliegt und der Versuch stattfindet, auf Komponente 21 zuzugreifen, so sollte diese Exception ausgelöst werden.

#### **EVector\_Bereichsverletzung**

Dies ist eine Exception für Trainingsmuster. Es gelten hier die gleichen Regeln wie für EGewichte\_Bereichsverletzung. Sie sollte also ausgelöst werden, wenn auf eine Komponente eines Trainingsmusters zugegriffen wird, die nicht existiert.

#### **EVector\_Rechenfehler**

Tritt bei der Operation mit einem Vector ein Rechenfehler auf, so wird durch diese Exception, dies dem Programmierer gemeldet. Diese Exception wird jedoch von dem vorliegenden Programm in der aktuellen Version nicht unterstützt und ist zu Zwecken der Weiterentwicklung vorgesehen.

#### **ENeuron\_defekt**

Diese Exception wird ausgelöst, wenn auf ein Neuron zugegriffen wird, welches nicht existiert. Dies kann z.B. geschehen, wenn Speicherfehler auftreten. Dieser Fall ist jedoch sehr unwahrscheinlich.

### **5.10 Unit Graph**

Die Unit Graph ist eine der umfangreichsten Units dieses Programms. Sie dient der Visualisierung der Gewichtsvektoren der Neurone auf einer zweidimensionalen Zeichenebene. Die hierarchische Struktur des Netzes wird durch Verbindungslinien zwischen den Projektionen der Gewichtsvektoren dargestellt. Diese Verbindungslinien korrespondieren mit der hierarchischen Struktur der Neurone untereinander. Abhängig von der Dimensionalität der Gewichtsvektoren stellt diese Unit die Gewichtsvektoren entweder direkt auf der Zeichenebene dar (Dimensionalität = 2). In den meisten Fällen arbeitet der Anwender jedoch mit Trainingsmustern, die mehr als zwei Dimensionen haben. In diesem Falle können die Gewichtsvektoren in einem Polarkoordinatensystem abgebildet werden. Sie werden kreisförmig um den Gewichtsvektor des Wurzelneurons angeordnet und die Abstände der Gewichtsvektoren zueinander entsprechen den euklidischen Distanzen im  $n$ -dimensionalen Raum.

Diese Unit verwaltet auch die Benutzereingaben, die über die Zeichenebene stattfinden. Der Mauscursor dient in Interaktion mit der Zeichenebene für verschiedene Aufgaben. Die Position des Mauscursors wird in Bildschirmpixeln angegeben. Diese Positionsangaben müssen in das jeweil verwendete Koordinatensystem transformiert werden. Was ebenfalls Aufgabe der Unit Graph ist.

### 5.10.1 Wichtige Deklarationen und Variablen

**TMode = (Info,Drag,Zoom,Cut);**

Dies ist eine Deklaration des Typs enum. Eine Variable dieses Typs kann unterschiedliche, fest definierte Zustände annehmen. In diesem Fall sind es vier, die der Unit dazu dienen, die aktuelle Bedeutung des Mauscursors zu definieren. Der Wert Info bedeutet, der Mauscursor dient als Instrument, um ein Neuron auszuwählen, über das man mehr erfahren möchte. Drag gibt an, daß der Mauscursor nun die Aufgabe hat, die Zeichenebene zu verschieben. Zoom gibt Auskunft darüber, daß der Mauscursor nun als Fixpunkt einer Ausschnittvergrößerung oder einer Ausschnittverkleinerung dient. Cut verleiht dem Mauscursor die Fähigkeit, einzelne Neurone in der zweidimensionalen Ansicht auszuscheiden.

**PBwidthF, PBheightF:** die Breite und Höhe der Zeichenebene in absoluten Werten, d.h. im gleichen Wertebereich, in dem auch die Trainingsmuster und die Gewichtsvektoren operieren.

**PBwidthval, PBheightval:** geben die Breite und die Höhe der Zeichenebene in Bildschirmpixeln an.

**Buffer:** diese Variable ist eine Kopie der Zeichenebene. Es wird immer zuerst auf den Buffer gezeichnet. Wenn die Zeichenebene buffer komplett ist, wird sie in den sichtbaren Bereich des Bildschirms hineinkopiert, um Flimmereffekte beim Zeichnen zu vermeiden. Diese Technik wird auch Page-Flipping genannt.

**InteractionMode:** gibt an, welche Funktion der Mauscursor gerade hat. Dies ist ein Objekt des Typs TMode.

**ZoomRate:** der vom Anwender eingestellte Vergrößerungs- bzw. Verkleinerungsfaktor der

Zoomfunktion.

**Dataoriginx, Dataoriginy:** diese Variablen sind Public und repräsentieren den linken unteren Punkt der Zeichenebene in der Maßeinheit der Gewichtsvektoren.

**Datawidth, Dataheight:** public Variablen, die die Breite und Höhe der Zeichenebene in den Maßeinheiten der Gewichtsvektoren wiedergeben.

### **5.10.2 TDiagramm.PaintBoxPaint (Sender:TObject); (28,98)**

Die Prozedur PaintBoxPaint zeichnet ein neues Diagramm in den buffer und kopiert den Inhalt des Buffers anschließend auf die Speicherposition der sichtbaren Zeichenebene. Diesem Vorgang ist eine Abfrage vorgeschaltet, die verhindert, daß bei einem laufenden Ausdruck der Zeichenebene diese neu gezeichnet wird, da sonst das Druckergebnis eine zeilenweise Überlagerung mehrerer verschiedener Diagramme ergeben würde.

### **5.10.3 TDiagramm.FormCreate(Sender:TObject); (28,115)**

Bei jedem Programmstart wird diese Prozedur aufgerufen und initialisiert die Variablen der Unit Graph mit sinnvollen Werten.

### **5.10.4 TDiagramm.PaintBoxMouseMove(Sender:TObject; Shift:TShiftState); (28,136)**

Immer wenn die Maus über der Zeichenebene bewegt wird wird diese Prozedur aufgerufen. Sie ist lediglich für den InteractionMode=Info oder Cut interessant, da die Prozedur ständig ermittelt, auf welchen Gewichtsvektor der Mauscursor gerade zeigt.

Beim InteractionMode=Info wird der Name des zugehörigen Neurons und seine Nummer dem Anwender im Formular angezeigt.

Ist InteractionMode=Cut, so führt diese Prozedur darüber Protokoll, über welchem Gewichtsvektor sich die Maus gerade befindet, um das korrespondierende Neuron aus-

schneiden zu können, sobald auf die Maustaste geklickt wird.

#### **5.10.5 TDiagramm.PaintBoxMouseDown(Sender:TObject; Button:TMouse-Button; Shift:TShiftState; X,Y:Integer); (29,205)**

Diese Prozedur reagiert, wenn eine der beiden Maustasten gedrückt wird und sich der Cursor über der Zeichenebene befindet. Im wesentlichen ist diese Prozedur für zwei Modi von Bedeutung:

Hat die Variable InteractionMode den Wert zoom, so wird ein neuer Bildausschnitt definiert. Dies geschieht dadurch, daß die Variablen Dataoriginx, Dataoriginy, Datawidth, Dataheight so manipuliert werden, daß der vom Anwender gewünschte Zoomfaktor in diesen Variablen berechnet wird. Wird die linke Maustaste gedrückt, wird eine Ausschnittsvergrößerung vorgenommen. Wird die rechte Maustaste gedrückt, so wird der Bildausschnitt verkleinert.

Ist InteractionMode =Drag, so protokolliert diese Prozedur den Punkt, an dem die Maustaste gedrückt wurde, um den Ursprungspunkt der Verschiebung festzuhalten. Dieser Punkt wird in der Prozedur 5.10.6 zur Vervollendung der Verschiebeoperation gebraucht.

#### **5.10.6 TDiagramm.PaintBoxMouseUp(Sender:TObject; Button:TMouse-Button; Shift:TShiftState; X,Y:Integer); (30,262)**

Läßt der Anwender die Maustaste über der Zeichenebene los, so gibt es auch hier zwei Modi, bei denen die Prozedur Operationen einleitet:

Ist InteractionMode=info, so wird das Formular der Unit Neuroninspektor geöffnet und es werden weitergehende Informationen über das, über den korrespondierenden Gewichtsvektor selektierte, Neuron angezeigt.

Ist InteractionMode=drag, wird der Punkt ermittelt, an dem der Mauszeiger sich im Moment befindet. Die Prozedur 5.10.5 hat bereits die Position gespeichert, bei der die

Maustaste gedrückt wurde. Diese beiden Koordinatenangaben werden nun verrechnet, um eine Verschiebung der Zeichenebene über die gewünschte Distanz einzuleiten.

Ist InteractionMode=cut, wird das Neuron, über dem sich der Mauszeiger gerade befindet, aus dem hierarchischen neuronalen Netz gelöscht. Die entsprechende Prozedur aus der Unit Neuron (Tneuron.delete) wird aufgerufen und sorgt dafür, daß die Hierarchie des Netzes erhalten bleibt.

#### **5.10.7 TDiagramm.transform\_x (val:TWeight) : Longint; (31,315)**

Die Funktion transformiert eine Positionsangabe im Koordinatensystem der Gewichtsvektoren in eine Koordinatenangabe in Bildschirmpixeln. Dies ist notwendig, wenn der Programmierer herausfinden möchte, ob sich ein Gewichtsvektor in einem bestimmten Abschnitt der Zeichenfläche befindet.

#### **5.10.8 TDiagramm.transform\_y(val:TWeight) : Longint; (31,329)**

Diese Funktion arbeitet analog zu Funktion 5.10.7. Hierbei wird jedoch der y-Wert der Koordinatenangabe in Pixel umgerechnet.

#### **5.10.9 TDiagramm.transformxf(val:Integer) : TWeight; (31,343)**

Diese Funktion rechnet eine Koordinatenangabe (x-Richtung), die in Pixeln vorliegt, in eine Koordinatenangabe um, die den Gewichtsvektoren entspricht. Dies ist notwendig, will der Programmierer wissen, auf welchen Punkt des projizierten Raumes der Mauscursor nun zeigt.

#### **5.10.10 TDiagramm.transformyf(val:Integer) : TWeight; (31,353)**

Diese Funktion arbeitet analog zu 5.10.9. Eine Koordinatenangabe in Bildschirmpixeln wird in das Koordinatensystem der Gewichtsvektoren umgerechnet, in diesem Falle für die y-Richtung.

#### **5.10.11 TDiagramm.circlediagramm; (32,418)**

Die Prozedur `circlediagramm` erzeugt Projektionsansichten höherdimensionaler Gewichtsvektoren in den zweidimensionalen Raum. Dies geschieht durch die Darstellung der Gewichtsvektoren mittels Polarkoordinaten. Der Gewichtsvektor des Wurzelneurons ist der Ursprungspunkt dieses Koordinatensystems. Die Prozedur erstellt zunächst eine nach Ebenen sortierte Liste der Neurone. Dann ermittelt sie die Anzahl der Neurone auf einer Ebene. Aus dieser Anzahl berechnet die Funktion den jeweiligen Raumwinkel, den ein Gewichtsvektor einnehmen darf. Diese Berechnung führt direkt zu der Koordinatenangabe  $\Phi$  für jeden Gewichtsvektor. Der Abstand zum Wurzelneuron  $\rho$  wird entweder vom Anwender festgelegt, was zu einer statischen Darstellung des neuronalen Netzes führt, oder  $\rho$  entspricht der euklidischen Distanz zwischen dem Gewichtsvektor des Kindneurons und dem Gewichtsvektor des Elternneurons. Auf diese Weise wird das Kreisdiagramm ringförmig vom Wurzelneuron ausgehend aufgebaut. Diese Prozedur berechnet nur die Koordinatenwerte der Gewichtsvektoren, während die eigentlichen Zeichenoperationen von der Prozedur 5.10.12 ausgeführt werden.

#### **5.10.12 TDiagramm.draw\_diagramm; (34,544)**

Diese Prozedur übernimmt die eigentlichen Zeichenoperationen auf der Zeichenebene. Sie verwendet zwei einfache Zeichenoperationen. Es werden nur Kreise oder Linien gezeichnet, komplexere Zeichenroutinen werden hier nicht aufgerufen. Zunächst ermittelt die Prozedur, ob ein Polardiagramm gezeichnet werden soll oder ein einfaches zweidimensionales Diagramm. Falls ein Polardiagramm gezeichnet werden soll, wird Prozedur 5.10.11 aufgerufen, welche die Polarkoordinatenwerte der Gewichtsvektoren ermittelt und die Umrechnungsergebnisse in die Neurone schreibt.



Nun beginnt die eigentliche Zeichenoperation. Im Falle einer zweidimensionalen Darstellung werden auch die Trainingsmuster auf der Zeichenebene dargestellt. Der Hierarchie folgend werden auch die Verbindungslinien zwischen den Gewichtsvektoren der Elternneuronen und der jeweiligen Kindneurone eingezeichnet. Schließlich ermittelt die Prozedur aus der Variablen winlist der Unit Train (5.6.1) die Gewinnerneurone. Sofern der Anwender dies wünscht, werden diese durch einen kleinen gelben Kreis markiert.

## **5.11 Unit Inspektr**

Die Unit Inspektr dient zum jetzigen Entwicklungsstand ausschließlich dazu, die Gewichtsvektoren der Neurone numerisch anzuzeigen. Der Anwender kann die Gewichtsvektoren jedes einzelnen Neurons durch das Formular dieser Unit ändern. Die Komponenten werden mittels eines TListBox Objektes angezeigt.

### **5.11.1 Wichtige Variablen**

**Aktneuron:** enthält die Indexnummer des gerade angezeigten Neurons.

**Aktwghtstr:** enthält den Wert der vom Anwender selektierten Komponente des Gewichtsvektors des angezeigten Neurons. Dieser Wert wird als String abgebildet, um Benutzereingaben in dieser Variable abbilden zu können. Die jeweilige Konvertierung zu Fließkommawerten findet in den Prozeduren dieser Unit statt.

**Aktdimstr:** enthält die Indexnummer der gerade selektierten Komponente des Gewichtsvektors des angezeigten Neurons. Auch dieser Wert wird zwecks leichter Interaktion mit dem Anwender als String repräsentiert. Hier findet ebenfalls eine Konvertierung in Fließkommawerte in den Prozeduren der Unit statt.

#### **5.11.2 TNeuroninspektor.GewListBoxEnter(Sender:TObject); (38,55)**

Diese Prozedur zeigt die Komponenten des Gewichtsvektors von aktneuron in einer Listbox an. Eine Überprüfung der Validität von dem Listeneintrag auf das aktneuron zeigt, verhindert einen Speicherzugriffsfehler, falls in aktneuron ungültige Werte enthalten sind.

#### **5.11.3 TNeuroninspektor.GewListBoxClick(Sender:TObject); (39,95)**

Klickt der Anwender in die Listbox, so markiert er damit eine bestimmte Komponente. Diese Prozedur übernimmt die Indexnummer der angewählten Komponente in die Variable Aktdimstr. Der Wert der selektierten Komponente wird in die Variable Aktwghtstr übertragen.

#### **5.11.4 TNeuroninspektor.GewEditExit(Sender:TObject); (39,115)**

Verändert der Anwender den angezeigten Wert einer Komponente des Gewichtsvektors, so bearbeitet diese Prozedur den Änderungswunsch und schreibt den neuen Wert der Komponente in das angezeigte Neuron.

#### **5.11.5 TNeuroninspektor.AktNeuronBoxChange(Sender:TObject); (39,126)**

Wünscht der Anwender den Gewichtsvektor eines anderen Neurons zu sehen, so gibt er in ein Feld des Formulars dieser Unit die gewünschte Neuronnummer an. Diese Prozedur bearbeitet die Benutzereingabe und führt Sicherheitsüberprüfungen durch, die einen Programmabsturz vermeiden sollen. Wird vom Anwender eine ungültige Indexnummer eines Neurons oder ein nichtnumerisches Zeichen eingegeben, so werden die Werte des Wurzelneurons angezeigt. Dieses Neuron ist immer in einem hierarchischen neuronalen Netz vorhanden.

## **5.12 Unit FileM**

Die Unit FileM soll zum jetzigen Entwicklungsstand ausschließlich als Anregung dienen. Sie definiert eine Funktion, aus deren Aufbau man ein Format zum Abspeichern der neuronalen Netze ableiten kann. Diese Unit soll eine Grundstruktur zur Verfügung stellen, mit deren Hilfe der Entwickler Netze abspeichern kann.

### **5.12.1 TFileManager.save(Dateiname : string):Boolean; (41,27)**

Diese Funktion öffnet ein TFileStream Objekt, der Dateiname, der zu sichernden Datei wird an das TFileStream Objekt übergeben. Sukzessive können nun die Speicher Routinen der zu sichernden Objekte aufgerufen werden. An diese Routinen wird die Referenz auf das TFileStream Objekt übergeben. Diese Routinen können nun die Parameter ihrer Objekte in den File-Stream schreiben.

## **5.13 Zusammenfassung**

Wie man sehen konnte, ist das Programm hochmodular aufgebaut. Es wurde darauf verzichtet, globale Variablen zu definieren, weil diese die Übersichtlichkeit des Programmcodes wesentlich einschränken. Fast alle wichtigen Variablen sind in Objekte gekapselt, die Informationen untereinander austauschen. Die Verwaltung eines neuen Netzes übernimmt im wesentlichen die Unit Param bzw. ein Objekt des Typs TParamform, in welchem die globalen Parameter des Netzes gespeichert werden. Es initialisiert ein neues Netz im Speicher und etabliert die hierarchischen Verbindungen der Neurone untereinander.

Die Trainingsmuster werden von einem Objekt des Typs Tdata verwaltet, welches in der Unit Datau definiert ist. Dieses Objekt kapselt sämtliche dateispezifischen Operationen und generiert ein fertiges Set an Trainingsmustern, welches das Objekt TTrain (Unit

Trainer) direkt verwenden kann, um das Netz zu trainieren. Das Ttrain Objekt holt sich die Parameter bezüglich des Lernens (Lernrate Alpha) seinerseits aus einem Objekt des Typs TParamForm, welches auch die Architektur des hierarchischen neuronalen Netzes festlegt.

Das Objekt des Typs TDiagramm (Unit Graph) ist zur Visualisierung von Trainingsmustern und Gewichtsvektoren implementiert worden. Es interagiert im wesentlichen mit dem Objekt des Typs TTrain, von dem es Anweisungen bekommt, wann ein neues Diagramm zu zeichnen ist.

Das zentrale Objekt wird in der Unit Neuron definiert. Objekte des Typs TNeuron sind die eigentliche Essenz dieses Programms. Das Konzept eines Neurons in einem hierarchischen neuronalen Netz wird hier implementiert.

Hinzu kommen Hilfsunits, die ihrerseits Objekte definieren, die in der vorliegenden Form nicht im Sprachumfang der Programmiersprache enthalten waren. Unit Vecmat definiert Vektorrechnungsoperationen und entsprechende Objekte. Die Unit Polaru ist für die Umrechnung von Polarkoordinaten in Kartesische Koordinaten und die umgekehrte Umrechnung geschrieben worden.

Dieses Konzept erscheint am Anfang etwas unübersichtlich, doch erleichtert es die Weiterentwicklung des Sourcecodes, da auf eine strenge Kapselung von zusammengehörigen Konzepten, Methoden und Variablen geachtet wurde. Eine Weiterentwicklung und Anpassung des Systems dürfte daher kein größeres Problem darstellen.

## 6 Benutzeranleitung

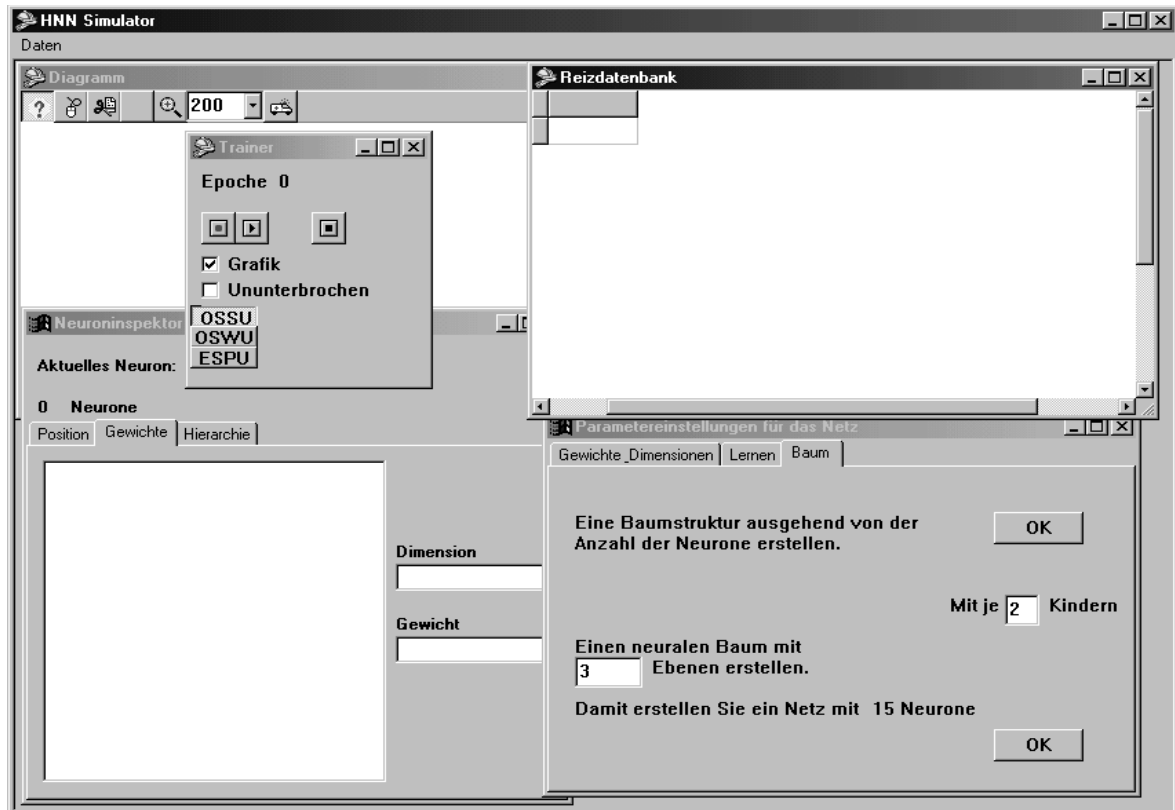


Abbildung 9 Eine Übersicht über alle Fenster der Programmoberfläche

Dieses Kapitel wurde stilistisch so formuliert, daß es als Benutzeranleitung gelesen werden kann. Daher wird der Leser in diesem Kapitel persönlich angesprochen.

Das Programmsystem nennt sich HNN Simulator, dies steht für: hierarchische neuronale Netze Simulator. In Abbildung 9 ist eine Übersicht der Programmoberfläche zu sehen. Sämtliche Programmfunktionen können über diese graphische Benutzeoberfläche angesprochen werden. Das Programmsystem kann ein hierarchisches neuronales Netz und einen Satz von Trainingsmustern verwalten. Alle Fenster dieser Programmoberfläche beziehen sich auf die Bedienung des Netzes und die Verwaltung der Trainingsmuster. Die einzelnen Unterfenster tragen die Bezeichnungen Diagramm, Reizdatenbank, Parameter-

einstellungen für das Netz, Neuroninspektor und Trainer. Dieser Abschnitt soll nun die Funktion der einzelnen Fenster und deren Wechselwirkungen dokumentieren, damit sich der Anwender mit der Programmoberfläche schnell zurechtfinden kann.

### **6.1 Eine Datei mit Trainingsmustern öffnen**

Zunächst muß ein Set von Trainingsmustern in das Programm geladen werden, bevor man sinnvoll anfangen kann, mit diesem Programm zu arbeiten. Dazu klickt man das Fenster mit dem Titel Reizdatenbank an. Daraufhin klicken Sie auf den Punkt Datei der Menüleiste. Ein Untermenü wird sich öffnen, klicken Sie nun auf den Unterpunkt öffnen.

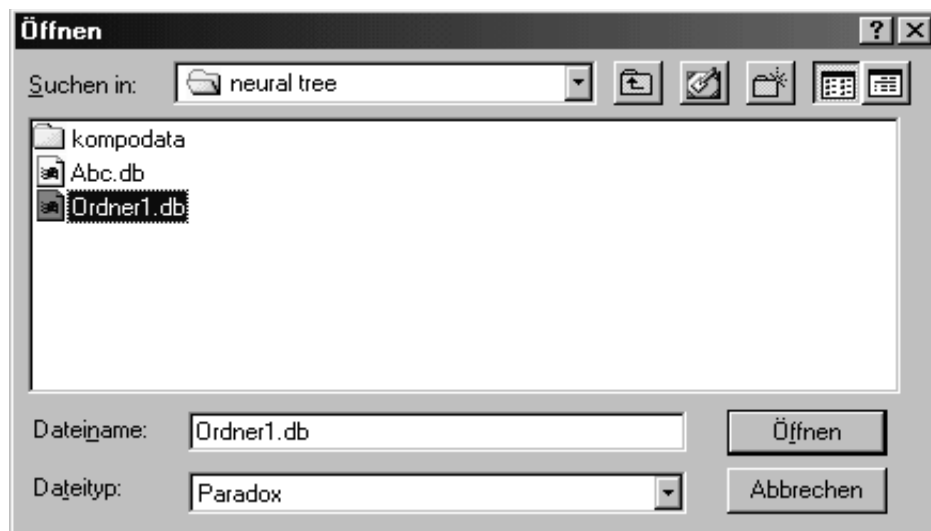
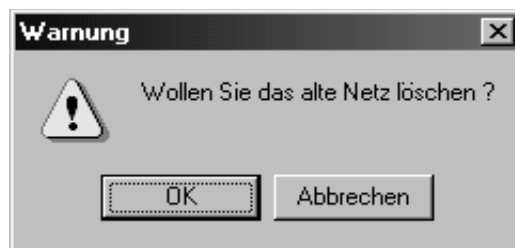


Abbildung 10 Das Dialogfenster zum öffnen einer Datei mit Trainingsmustern

Abbildung 10 zeigt nun das Fenster, welches daraufhin sichtbar wird. Die Trainingsmuster müssen als Dateien im Borland-Paradox oder im dBase Format vorliegen. Es ist ohne weiteres möglich, diese Dateiformate mit gängigen Tabellenkalkulationsprogrammen zu exportieren. Beachten Sie bitte, daß das Format folgende Konvention erfüllen muss: wenn

Sie die Trainingsmuster mit einer Tabellenkalkulation erstellen, muß die erste Spalte entweder leer sein oder ein Label enthalten, welches z.B. zur Markierung von Gruppen der Trainingsmuster dienen kann.

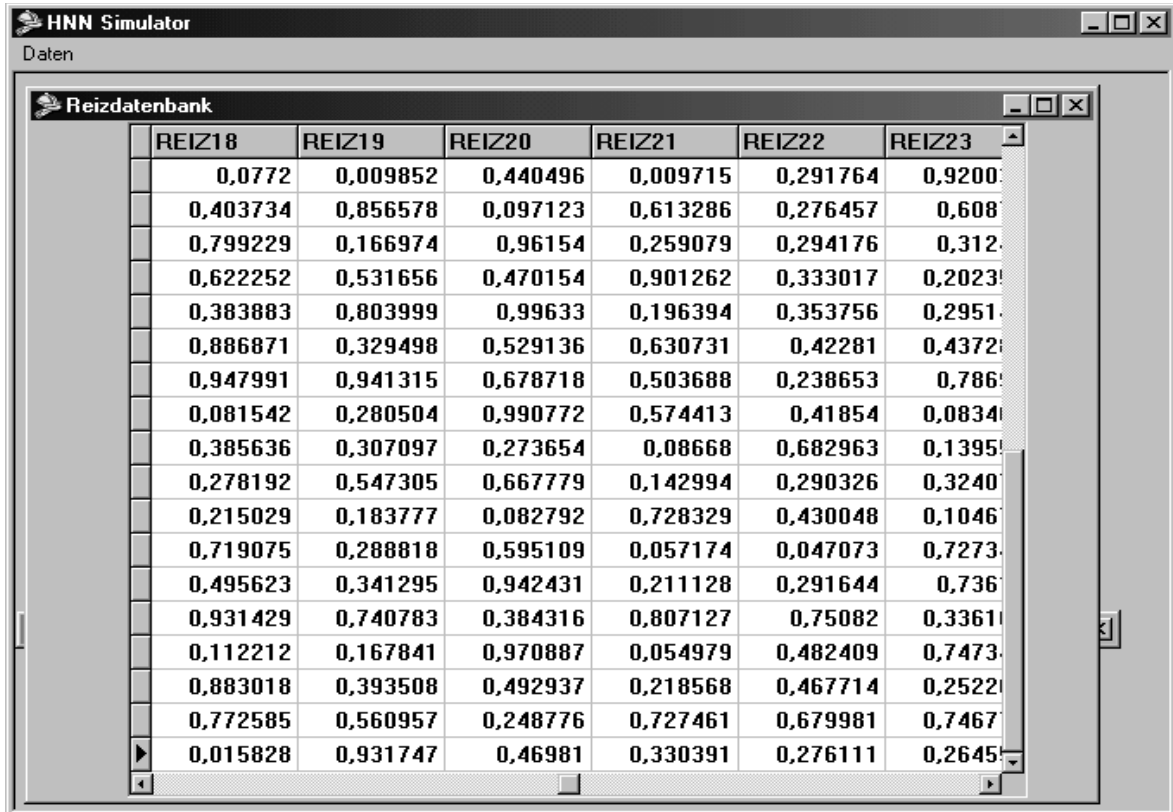
Wählen Sie nun im Pulldown-Fenster Dateityp, das Format Ihrer Trainingsmuster aus, also entweder Paradox oder dBase. Das Fenster entspricht den Konventionen eines Standarddateidialogs von Windows Programmen, welches Sie beispielsweise aus Ihrer Textverarbeitung kennen. Wählen Sie nun das entsprechende Arbeitsverzeichnis und die Datei aus, in der Ihre Trainingsmuster stehen. Nachdem Sie auf das Button „Öffnen“ gedrückt haben, erscheint zunächst ein Warnhinweis.



*Abbildung 11 Dialogabfrage nach Änderungen am Netzwerk*

Diese Warnung können Sie nun mit dem Klick auf das „OK“ Button übergehen, da zum jetzigen Zeitpunkt ja noch kein neuronales Netz definiert ist. Dieses Dialogfenster wird im Laufe des Arbeitens mit dem Programm häufiger auftauchen. Und zwar immer dann, wenn ein neues Netz generiert werden soll. Sämtliche Daten des alten Netzes gehen verloren, wenn Sie auf „OK“ klicken !

Nun werden die Trainingsmuster in das Programm eingelesen und ein erstes einfaches Netz initialisiert. Das Fenster Reizdatenbank enthält nun Ihre Trainingsmuster und müßte in etwa so aussehen (Abbildung 12):



The screenshot shows a window titled 'HNN Simulator' with a sub-window 'Reizdatenbank'. The 'Reizdatenbank' window contains a table with 7 columns labeled REIZ18 through REIZ23. The table lists 20 rows of numerical data, representing training patterns. The values are as follows:

	REIZ18	REIZ19	REIZ20	REIZ21	REIZ22	REIZ23
	0,0772	0,009852	0,440496	0,009715	0,291764	0,9200
	0,403734	0,856578	0,097123	0,613286	0,276457	0,608
	0,799229	0,166974	0,96154	0,259079	0,294176	0,312
	0,622252	0,531656	0,470154	0,901262	0,333017	0,2023
	0,383883	0,803999	0,99633	0,196394	0,353756	0,2951
	0,886871	0,329498	0,529136	0,630731	0,42281	0,4372
	0,947991	0,941315	0,678718	0,503688	0,238653	0,786
	0,081542	0,280504	0,990772	0,574413	0,41854	0,0834
	0,385636	0,307097	0,273654	0,08668	0,682963	0,1395
	0,278192	0,547305	0,667779	0,142994	0,290326	0,3240
	0,215029	0,183777	0,082792	0,728329	0,430048	0,1046
	0,719075	0,288818	0,595109	0,057174	0,047073	0,7273
	0,495623	0,341295	0,942431	0,211128	0,291644	0,736
	0,931429	0,740783	0,384316	0,807127	0,75082	0,3361
	0,112212	0,167841	0,970887	0,054979	0,482409	0,7473
	0,883018	0,393508	0,492937	0,218568	0,467714	0,2522
	0,772585	0,560957	0,248776	0,727461	0,679981	0,7467
	0,015828	0,931747	0,46981	0,330391	0,276111	0,2645

Abbildung 12 Das Fenster Reizdatenbank nach dem Öffnen einer Datei mit Trainingsmustern

Hier können Sie noch einmal überprüfen, ob die Trainingsmuster auch so in das Programm übernommen worden sind, wie Sie es sich gedacht haben. Die einzelnen Trainingsmuster sind hier zeilenweise angeordnet, ein Trainingsmuster ist immer eine Zeile in diesem Fenster.

## 6.2 Festlegen der Netzwerkparameter

Nun ist es notwendig dem Programm mitzuteilen, welche Architektur das neuronale Netz haben soll, das die Trainingsmuster bearbeiten soll. Dies können Sie in dem Fenster



„Parametereinstellungen für das Netz“ vornehmen.

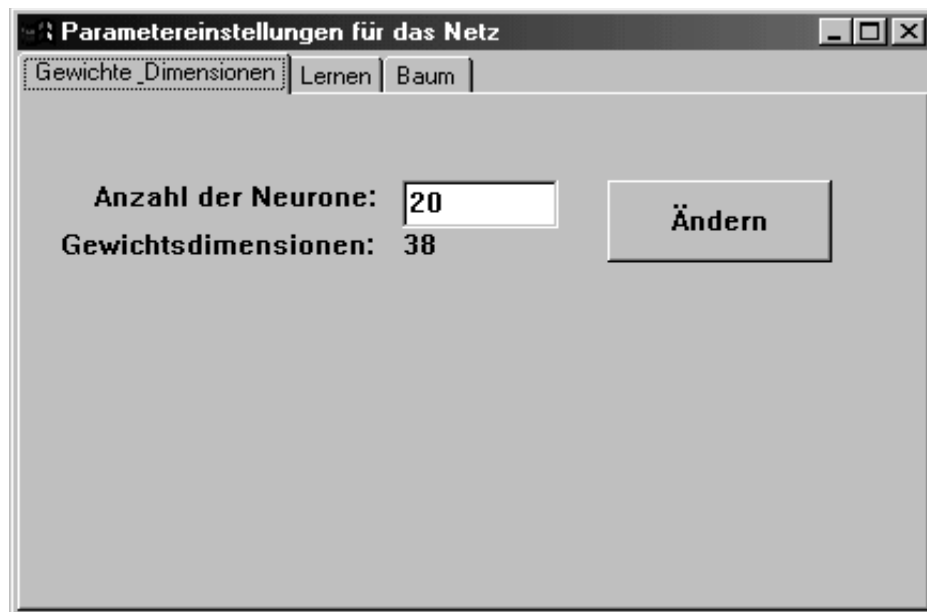


Abbildung 13 Anzahl der Neurone und Dimensionalität der Trainingsmuster

Dieses Fenster besteht aus drei Karteikarten, in denen Sie unterschiedliche Einstellungen vornehmen können. Abbildung 13 zeigt das Fenster „Parametereinstellungen für das Netz“ mit der ersten Karteikarte „Gewichte\_Dimensionen“. Hier können Sie hauptsächlich überprüfen, ob die Dimensionalität der Gewichtsvektoren mit der Dimensionalität Ihrer Trainingsmuster übereinstimmt. Die Neurone des neuronalen Netzes werden automatisch so initialisiert, daß Sie Gewichtsvektoren haben, die in ihrer Dimensionalität den Trainingsmustern entsprechen. Dies ist auch der Grund dafür, warum ein neuronales Netz immer gelöscht wird, wenn Sie ein neues Set von Trainingsmustern öffnen.

Sie können nun auch die Anzahl der Neurone im Netz festlegen, wenn Sie dies wünschen und auf das Button Ändern klicken. Beachten Sie bitte, daß danach noch keine Hierarchie

unter den Neuronen gebildet wird, dies müssen Sie dann erst noch in der Karteikarte Baum einleiten. Dies wird aber ein Weg sein, den Sie nicht beschreiten müssen um ein neuronales Netz zu initialisieren. Wie wir gleich sehen werden, gibt es dazu eine wesentlich bequemere Methode.

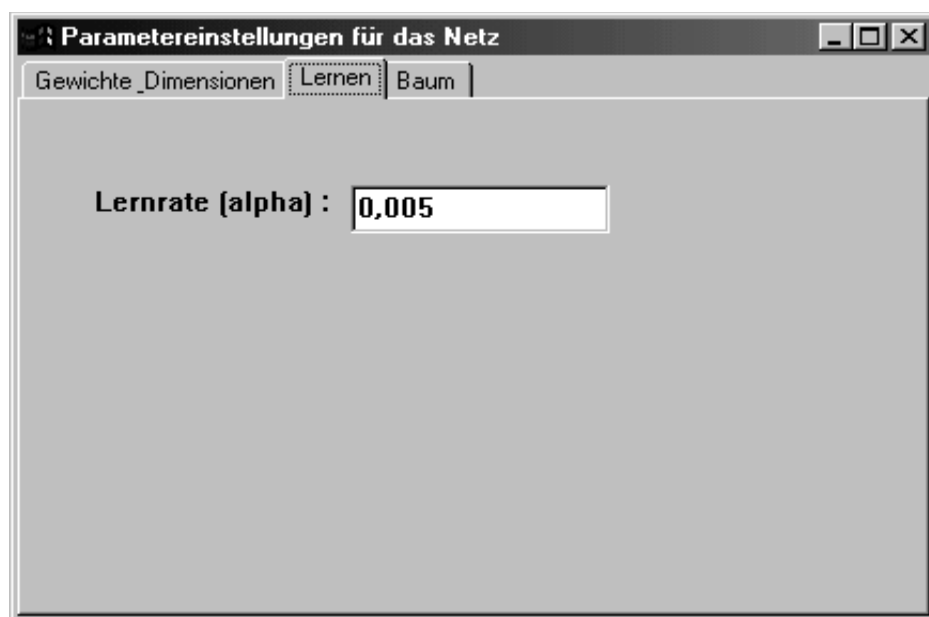


Abbildung 14 Einstellen der Lernrate

In Abbildung 14 sehen Sie nun was passiert, nachdem Sie auf die Karteikarte Lernen geklickt haben. Hier können Sie die Lernrate Alpha des hierarchischen neuronalen Netzes einstellen. Sinnvolle Werte der Lernrate liegen zwischen 0,05 und 0,001. Je größer die Lernrate ist, desto schneller wird sich das Netz Ihren Trainingsmustern anpassen, aber desto ungenauer wird auch das Ergebnis. Experimentieren Sie mit der Lernrate ruhig herum, Sie können diese auch während des Programmablaufs noch ändern.

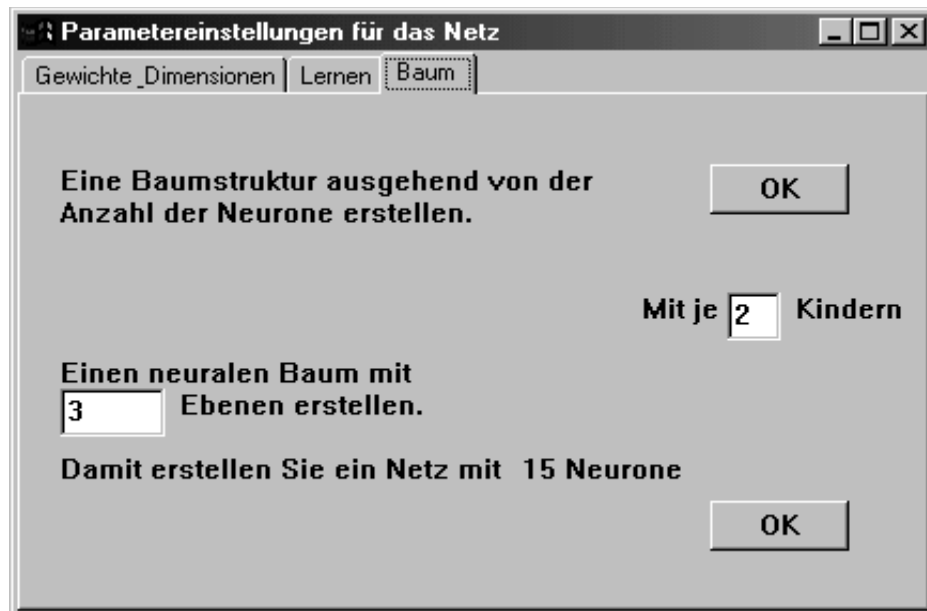


Abbildung 15 Festlegen der Architektur des Netzes

In Abbildung 15 sehen Sie nun die Karteikarte „Baum“. Hier können Sie die baumartige Struktur Ihres hierarchischen neuronalen Netzes bestimmen. Haben Sie in der Karteikarte „Gewichte\_Dimensionen“ eine feste Anzahl von Neuronen eingestellt, so stellen Sie nun bitte ein, wieviele Kinder jedes Neuron haben soll und klicken Sie auf das „OK“ Button oben rechts.

Ist es Ihnen egal, wieviele Neurone das Netz enthalten soll und Sie wollen lieber die Architektur festlegen, d.h. Sie wollen festlegen, wieviele Ebenen das Netz und wieviele Kinder jedes Neuron haben soll, so geht dies in der unteren Hälfte dieser Karteikarte. Nachdem Sie die Anzahl der Kinder pro Neuron festgelegt haben, tragen Sie in das zweite Eingabefeld noch die Anzahl der Ebenen ein, die Sie wünschen. In der letzten Zeile sehen Sie dann augenblicklich, wieviele Neurone generiert werden. Sind Sie mit der Anzahl der Neurone und der Architektur zufrieden, klicken Sie auf das „OK“ Button unten rechts.

### **6.3 Trainieren des hierarchischen neuronalen Netzes**

Nachdem Sie eine Architektur des Netzes festgelegt haben, wird ein neuronales Netz im Arbeitsspeicher ihres Computers generiert. Die Gewichtsvektoren der Neurone werden auf Zufallswerte initialisiert, die in ihrer Streuung dem kleinsten und dem größten Wert entsprechen, der in Ihrem Trainingsmuster vorkommt. Das Training selbst leiten Sie mit dem Fenster „Trainer“ ein.



Abbildung 16

Das Fenster "Trainer"

Ganz oben sehen Sie die Anzahl der bisher berechneten Epochen. In der nächsten Zeile sind nun Bedienelemente zu sehen, die denen eines Videorecorders ähneln. Sie beginnen das Training durch das Anklicken des „Record“ Buttons, mit dem kleinen roten Kreis in der Mitte. Ein laufendes Training können sie mit dem „Stop“ Button (schwarzes Rechteck in der Mitte) unterbrechen. Das „Play“ Button hat bei diesem Stand des Programms keine Funktion, es dient seiner Erweiterungsfähigkeit.

Steht vor der Zeile „Grafik“ ein Häkchen, so können Sie das Training im Graphikfenster beobachten. Beachten Sie bitte, daß dies die Rechengeschwindigkeit herabsetzt, da nach

jeder Epoche eine neue Graphik gezeichnet wird.

Steht vor der Zeile „Ununterbrochen“ ein Häkchen, so wird das Netz ohne Unterbrechung trainiert. Wenn Sie nun das Training stoppen möchten, so müssen Sie auf das „Stopp“ Button klicken.

Die einzelnen Trainingsalgorithmen (siehe Kaptitel 4) können Sie durch die unteren drei Buttons festlegen. Ist das „OSSU“ Button gedrückt, so lernt das Netz nach dem „Ordered Search Subtree Update“ Algorithmus. Zum Umschalten auf den „Ordered Search Winner Update“ Algorithmus drücken Sie auf das Button „OSWU“. Der letzte mögliche Lernalgorithmus heißt nun „Exhaustive Search Path Update“, um diesen zu aktivieren, klicken Sie auf das Button „ESPU“. Sie können die Algorithmen auch während des Trainings wechseln.

## **6.4 Die Diagrammfunktion**

Das Fenster, in dem die eigentlichen Ergebnisse der Simulation zu sehen sind, ist das Fenster „Diagramm“. Dieses Fenster erfüllt mehrere Funktionen. Zunächst stellt es die Postitionen der Trainingsmuster und der Gewichtsvektoren dar, sofern diese zweidimensional vorliegen. Haben die Trainingsmuster mehr als zwei Dimensionen, empfiehlt es sich, auf die Darstellung der Gewichtsvektoren im Polarkoordinatensystem umzuschalten.

Abbildung 17 zeigt die Darstellung eines hierarchischen neuronalen Netzes im Polarkoordinatensystem. Hier sehen Sie, daß nicht nur die Position der Gewichtsvektoren angezeigt wird, sondern daß Sie mit dieser Darstellung auch eine Vorstellung vom hierarchischen Aufbau des Netzes bekommen können. Die Gewichtsvektoren sind gemäß dem hierarchischen Aufbau des neuronalen Netzes untereinander verbunden. Der Gewichts-

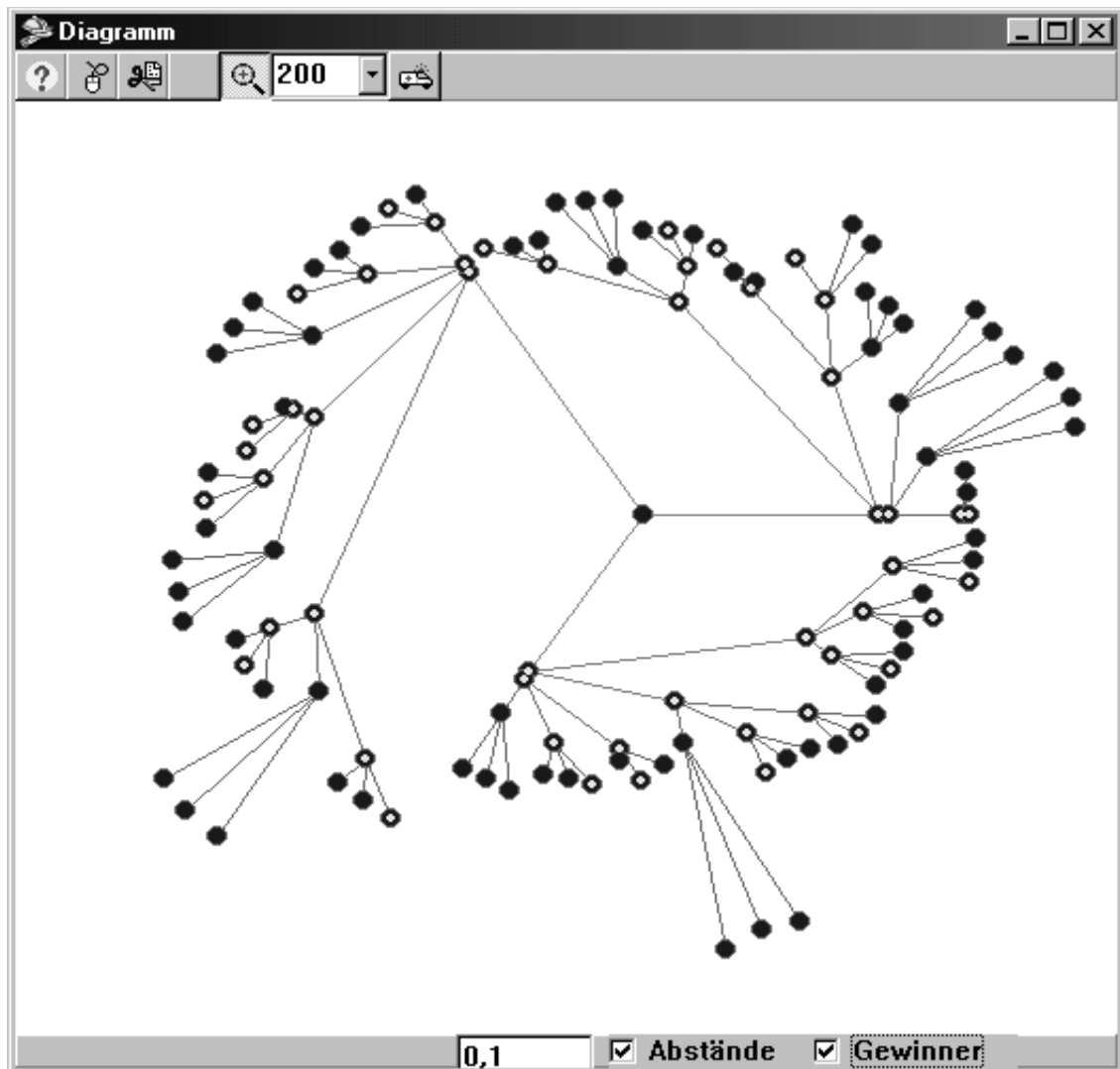


Abbildung 17 Darstellung der Gewichtsvektoren im Polarkoordinatensystem

vektor des Wurzelneurons findet sich in der Mitte der Anordnung. In der Abbildung ist ein hierarchisches neuronales Netz mit je drei Kindern und vier Ebenen zu sehen. Die Abstände der Gewichtsvektoren zum jeweiligen Elternteil entsprechen den euklidischen Distanzen im  $n$ -dimensionalen Raum, wobei  $n$  die Dimensionalität der Trainingsmuster bzw. der Gewichtsvektoren sei.

Bei dieser Darstellung ist es sinnvoll die Checkbox „Abstände“ zu markieren, da hier noch ein fetstgelegter Abstandswert zu den Distanzen hinzuaddiert wird, was die Darstellung

übersichtlicher macht. Ist diese Checkbox nicht markiert 'kollabiert' die Darstellung des Diagramms in Polarkoordinaten meist und es ist nur noch ein Kreis von Neuronen zu sehen. Haben Sie die Checkbox „Gewinner“ markiert, sehen Sie, welche Neurone in der letzten Epoche die Gewinnerneurone waren.

Die zweidimensionale Darstellung empfiehlt sich jedoch um erst einmal das grundlegende Verhalten des hierarchischen neuronalen Netzes zu studieren und die Unterschiede zwischen den einzelnen Algorithmen kennenzulernen.

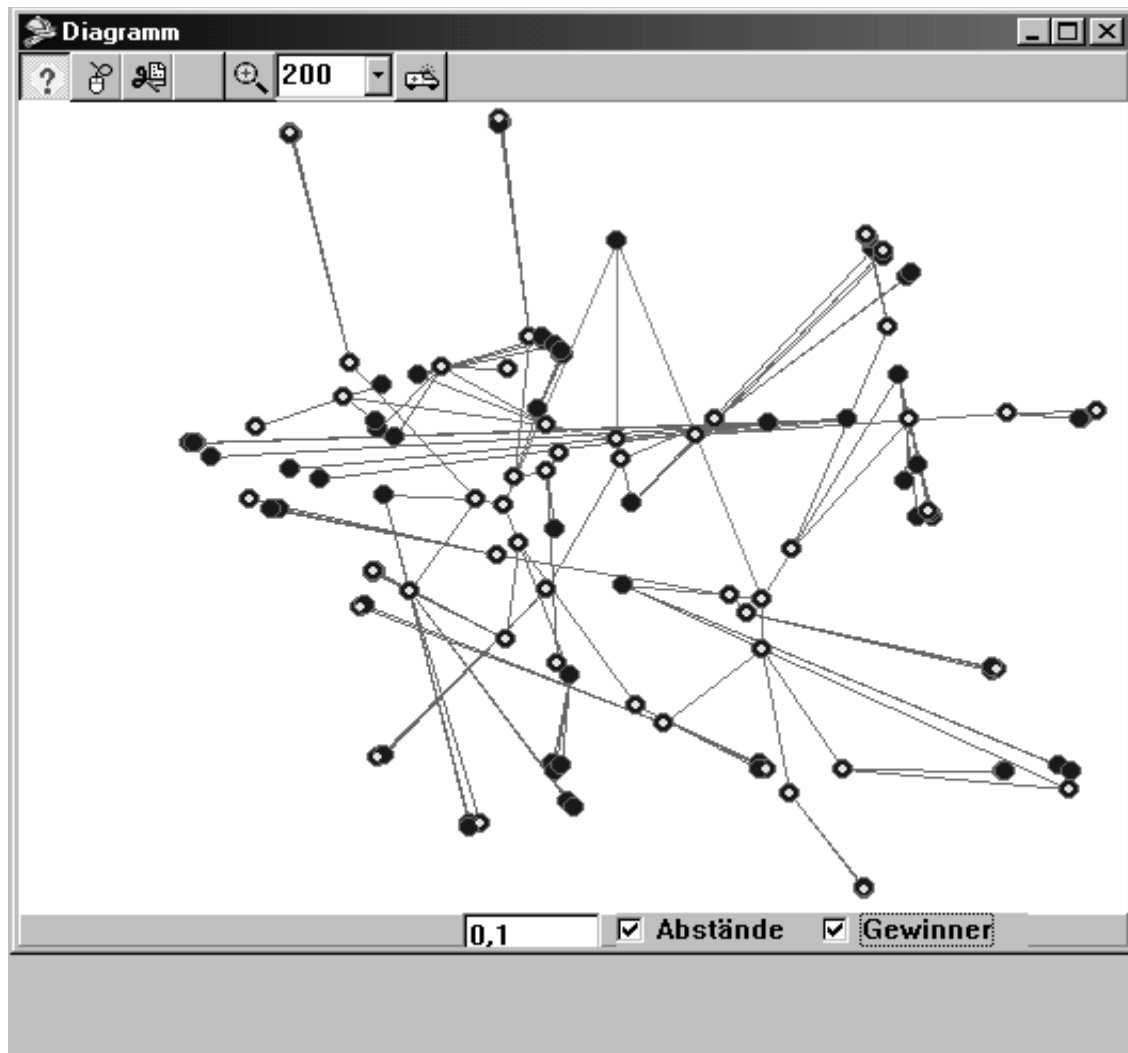


Abbildung 18 Darstellung der Gewichtsvektoren im zweidimensionalen kartesischen Koordinatensystem

Abbildung 18 zeigt die Darstellung der Gewichtsvektoren im zweidimensionalen Kartesischen Koordinatensystem. Diese Darstellung ist zu Beginn sehr unübersichtlich, da die Gewichtsvektoren noch mit Zufallswerten initialisiert sind, Sie müssen das Programm erst einige Epochen rechnen lassen, bis sich erkennbare Strukturen herausbilden.

#### 6.4.1 Interaktion mit der Zeichenebene

Sie haben die Möglichkeit, den Mauscursor zu benutzen um mehrerer Interaktionen mit der Zeichenebene vornehmen zu können. Hierzu sind die Buttons im oberen Teil des Gra-



phikfensters gedacht. Sie können diese Buttons in den Abbildungen 17 und 18 sehen.

#### 6.4.2 Informationen über ein Neuron aufrufen

Haben Sie das Button mit dem Fragezeichen angewählt, so dient der Mauscursor als Zeiger, der Ihnen die Informationen über ein Neuron präsentieren kann. Halten Sie die Maustaste gedrückt und bewegen Sie den Mauscursor zu einem Gewichtsvektor, der Sie interessiert. Links unten im Graphikfenster sehen Sie dann den aktuellen Namen des Neurons, voreingestellt ist der Name „Nobody“, und Sie sehen seine Nummer rechts daneben.

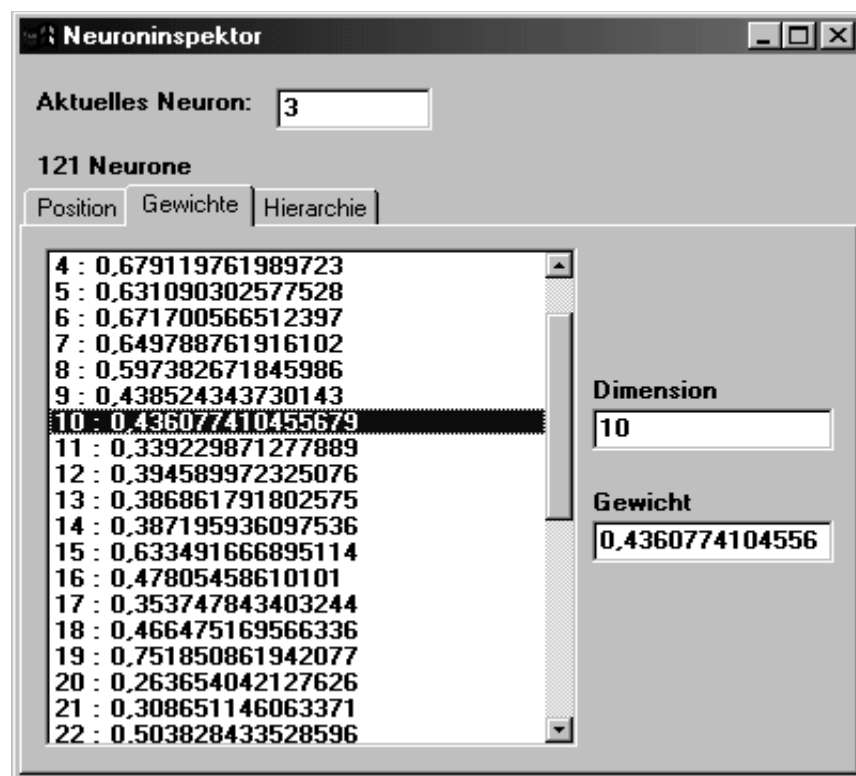


Abbildung 19 Das Fenster Neuroninspektor mit den Komponenten des Gewichtsvektors von Neuron 3

Lassen Sie nun die linke Maustaste los, so wird dieses Neuron in das Fenster „Neuronin–

spektor“ (Abbildung 19) übernommen und Sie können sich die Gewichtsvektoren dieses Neurons ansehen. Wenn Sie eine Komponente des Gewichtsvektors ändern möchten, markieren Sie die gewünschte Komponente mit der Maus. In Abbildung 19 ist gerade die Komponente 10 des Neurons 3 selektiert. Nun können Sie in der Eingabox „Gewicht“ die Komponente von Hand ändern.

#### **6.4.3 Verschieben der Zeichenebene**

Drücken Sie hierzu das Button mit der Maus (2. von links). Sie können nun die Zeichenebene verschieben. Bewegen Sie hierzu die Maus zu dem Punkt auf der Zeichenebene, von dem die Verschiebeoperation ausgehen soll. Halten Sie die linke Maustaste gedrückt und ziehen Sie die Maus zu dem Punkt an dem die Verschiebeoperation beendet werden soll. Lassen Sie nun die linke Maustaste los. Das Diagramm wird nun vom ersten Punkt zum zweiten hin verschoben und Sie sehen einen anderen Bildausschnitt des Diagramms.

#### **6.4.4 Ein Neuron ausschneiden**

Wählen Sie hierzu das Button mit der Schere (3. von links)

Diese Funktion ist zum jetzigen Entwicklungsstand des Systems als experimentell anzusehen. Wenn Sie mit dem Netz experimentieren und testen möchten, wie es darauf reagiert, daß einzelne Neurone ausgeschnitten werden, bewegen Sie die Maus über einen Gewichtsvektor bzw. ein Neuron das ausgeschnitten werden soll, und klicken Sie darauf. Das Neuron verschwindet daraufhin aus der Diagrammdarstellung. Sein Elternneuron wird automatisch mit seinen Kindern verbunden, um die hierarchische Struktur des Netzes zu wahren.

#### **6.4.5 Hinein- und Herauszoomen**

Wählen Sie hierzu das Button mit der Lupe (4. von links)

Wollen Sie Ausschnitte des Diagramms vergrössern oder verkleinern, bewegen Sie den Mauscursor über den gewünschten Bereich des Diagramms. Mit der linken Maustaste zoomen Sie in das Diagramm hinein, mit der rechten Maustaste zoomen Sie aus dem Diagramm heraus. Die Zoomrate können Sie im Pulldown Menü rechts neben dem Zoombutton einstellen. Voreingestellt ist eine Zoomrate von 200%.

#### **6.4.6 Darstellung in Kartesischen oder Polarkoordinaten**

Um zwischen den schon besprochenen Darstellungen in Kartesischen und Polarkoordinaten umschalten zu können, drücken Sie einfach das Button mit dem Auto (ganz rechts). Das Programm schaltet dann zwischen beiden Ansichten hin und her.

## 7 Tutorial

Nachdem die Programmfunktionen im vorigen Kapitel erläutert worden sind, beschäftigt sich dieses Kapitel mit der praktischen Durchführung von Trainings des hierarchischen neuronalen Netzes. Der Schwerpunkt wird hierbei auf die Interpretation der Diagramme gelegt. Hier werden die Unterschiede der drei Trainingsalgorithmen in der praktischen Anwendung aufgezeigt. Bei den hier verwendeten Trainingsmustern handelt es sich um einen Datensatz, welcher drei Cluster repräsentiert. Zwecks guter Darstellbarkeit der Polardiagramme und der Diagramme im kartesischen Koordinatensystem, liegen die Daten als zweidimensionale Vektoren vor.

Dieses Kapitel soll eine Einführung in die Interpretation der vom Programm erzeugten Diagramme geben. Dies geschieht rein visueller Ebene, da es den Anwendern dieses Programms überlassen ist, die für den jeweiligen Verwendungszweck nützlich erscheinenden empirischen Parameter selbst zu implementieren.

## 7.1 Ordered Search Subtree Update (OSSU)

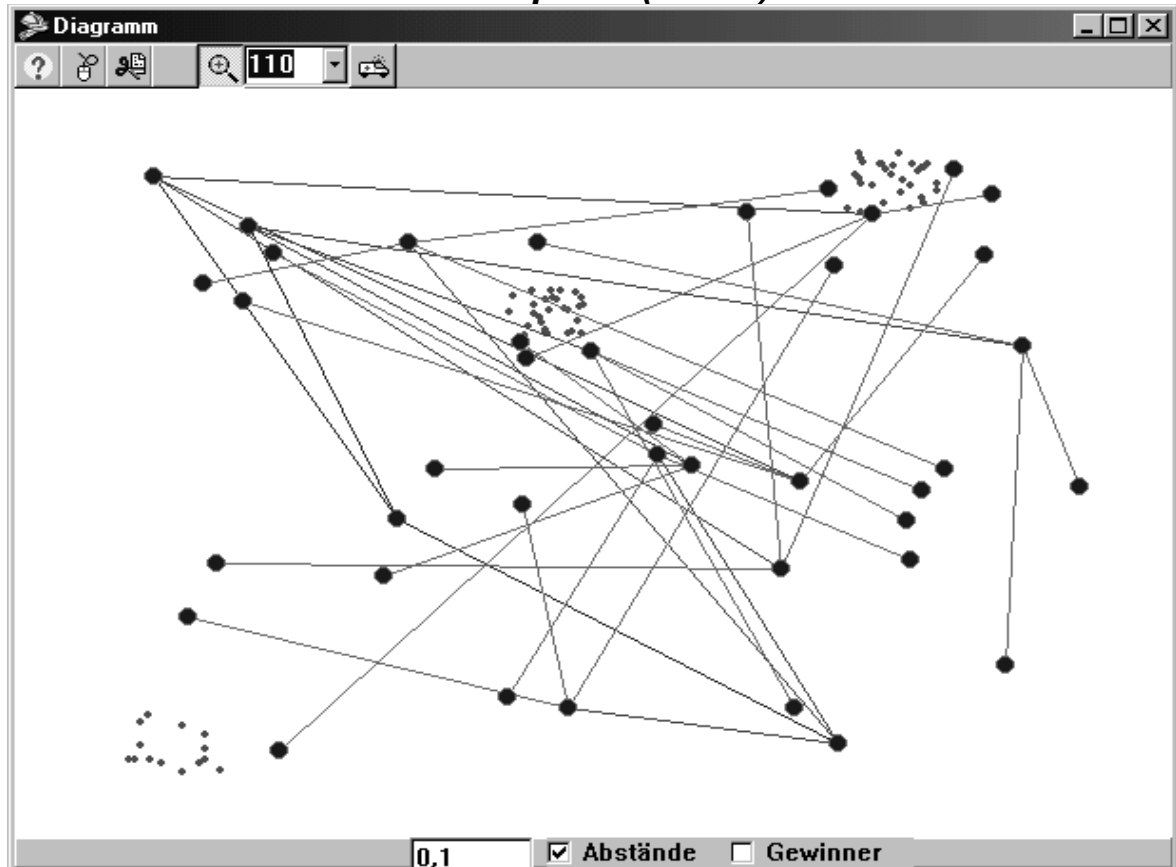


Abbildung 20 Hierarchisches neuronales Netz im initialisierten Anfangszustand

Abbildung 20 zeigt ein hierarchisches neuronales Netz mit drei Ebenen unter dem Wurzelneuron und jeweils drei Kindern pro Neuron. Insgesamt sind es also 40 Neurone. Die Gewichtsvektoren der Neurone sind bisher auf Zufallswerte eingestellt, daher wirkt die Struktur noch sehr unübersichtlich. Sehr schön sind jedoch die Datenpunkte des Satzes von Trainingsmuster zu sehen. Diese bilden drei Cluster, zwei Cluster findet man im rechten oberen Abschnitt des Diagramms, der dritte Cluster ist im linken unteren Bildrand zu sehen.

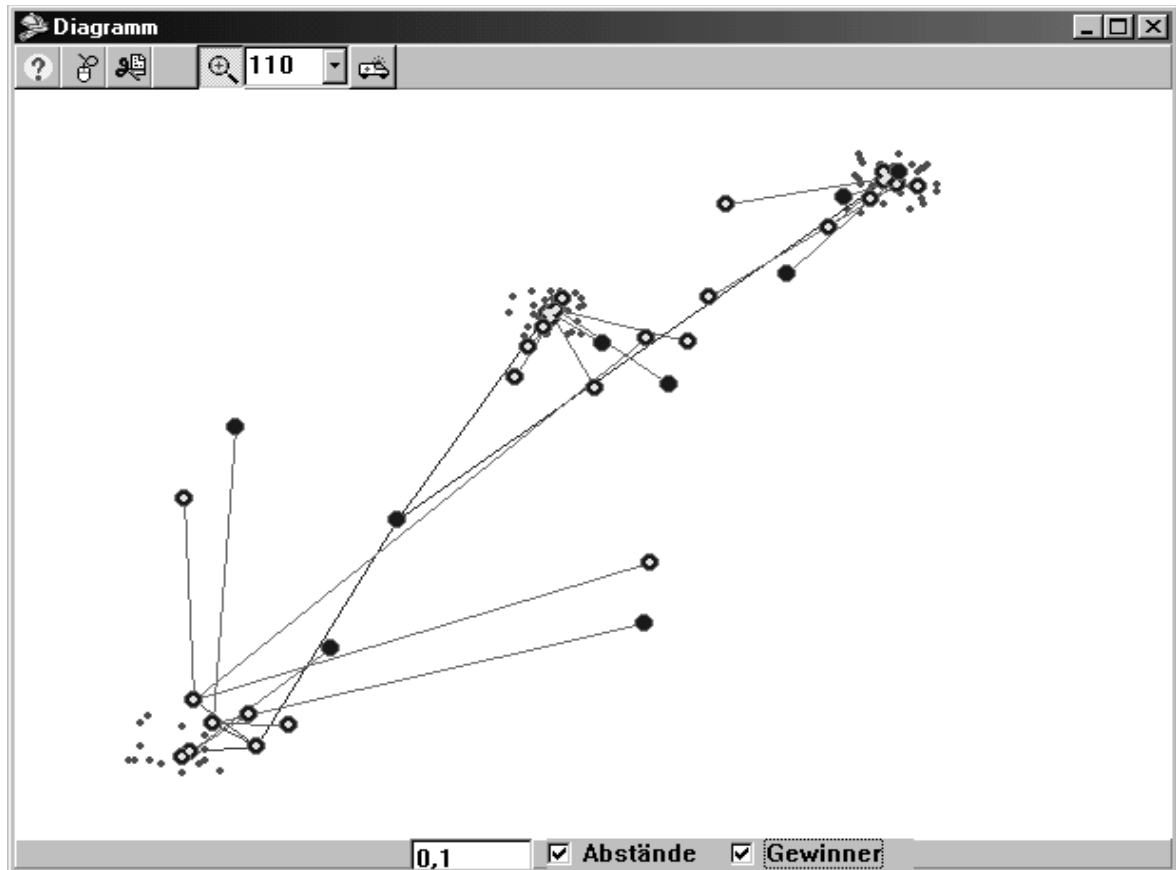


Abbildung 21 Das OSSU Training nach 30 Epochen

Abbildung 21 zeigt nun den Zustand des Netzes nach 30 Trainingsepochen. Man sieht, daß sich die Gewichtsvektoren schon nach dieser verhältnismäßig geringen Zahl von Lernschritten an die Verteilung der Trainingsmuster anpassen. Man beachte, daß der Algorithmus vom Programm so umgesetzt wird, wie im Methodikteil (Kapitel 4) beschrieben. Die Neurone, die in der untersten Ebene liegen, werden als letztes zu ihren Eltern hingezogen, was daran liegt, daß das Gewinnerneuron beim OSSU Training vom Wurzelneuron ausgehend ermittelt wird. Die ersten Ebenen des Netzes werden also zuerst an die Trainingsmuster angepaßt.

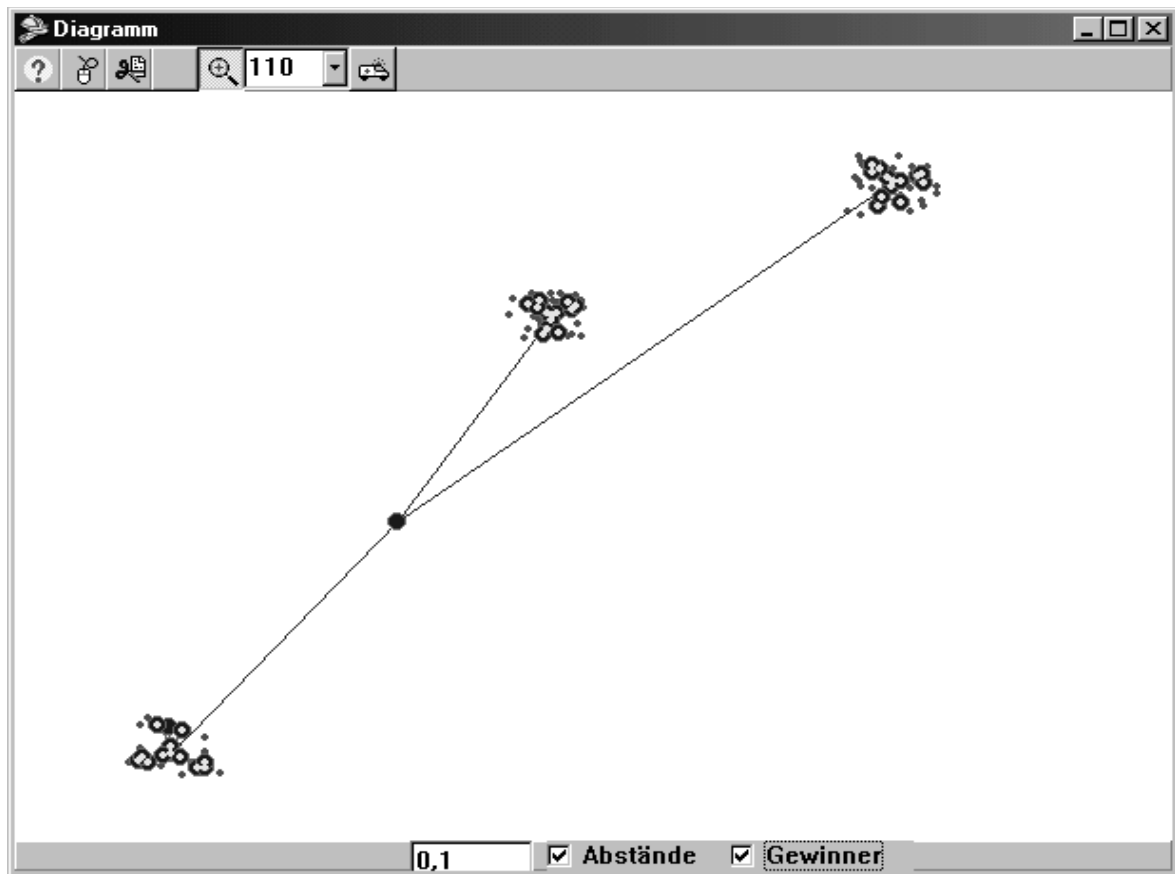


Abbildung 22 Das OSSU Training ist nach 400 Epochen beendet

Nach 400 Epochen ist ein stabiler Zustand eingetreten (Abbildung 22) und das Training konnte beendet werden. Hier ist sehr schön der Gewichtsvektor des Wurzelneurons zu sehen, von dem ausgehend die Gewichtsvektoren seiner Kinder die drei Cluster abbilden, die die Trainingsmuster repräsentieren. Der Anwender kann das Training als beendet ansehen, wenn sich die Gewichtsvektoren nicht mehr merklich anpassen. Dies lässt sich am besten überwachen, wenn das Training läuft und man sich gleichzeitig das Diagramm ansieht. Erscheint es nach einer Anzahl von Epochen statisch und ist keine Bewegung der Gewichtsvektoren zwischen den Epochen mehr zu erkennen, kann man das Training als

beendet erachten.

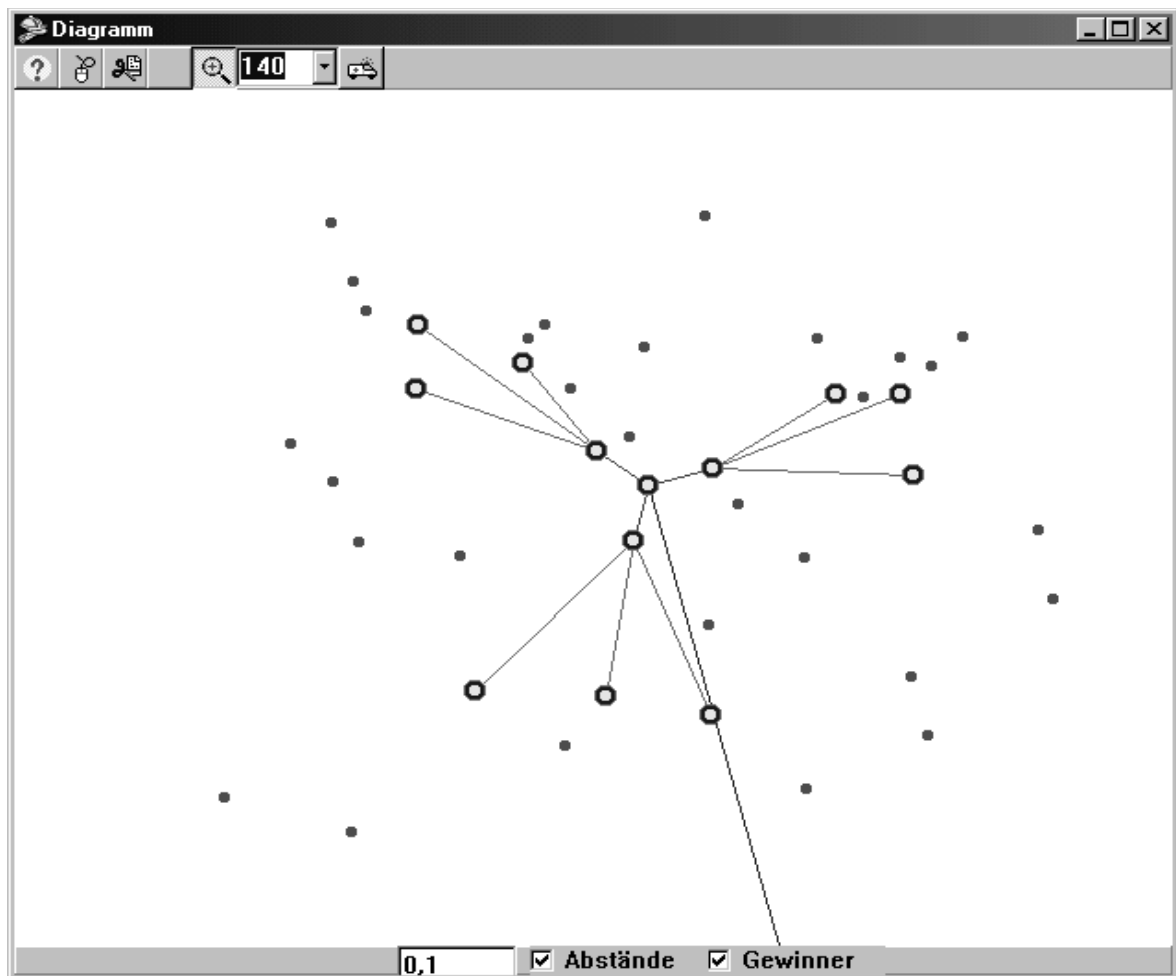


Abbildung 23 Zoom auf den rechten oberen Cluster.

Schaut man sich nun einen Cluster genauer an (Abbildung 23), so sieht man, daß die Gewichtsvektoren die Tendenz aufweisen, sich zum Mittelpunkt des Clusters hin anzuordnen. Damit erhält man eine gute allgemeine Annäherung der Daten. Die Generierung eines prototypischen Musters durch das Neuron auf Ebene 1 (es ist mit dem Wurzelneuron verbunden) ist damit wahrscheinlich. Die Passung an die Trainingsmuster ist jedoch nicht sehr gut, da die Ausreißerwerte des Clusters schlecht abgedeckt werden.



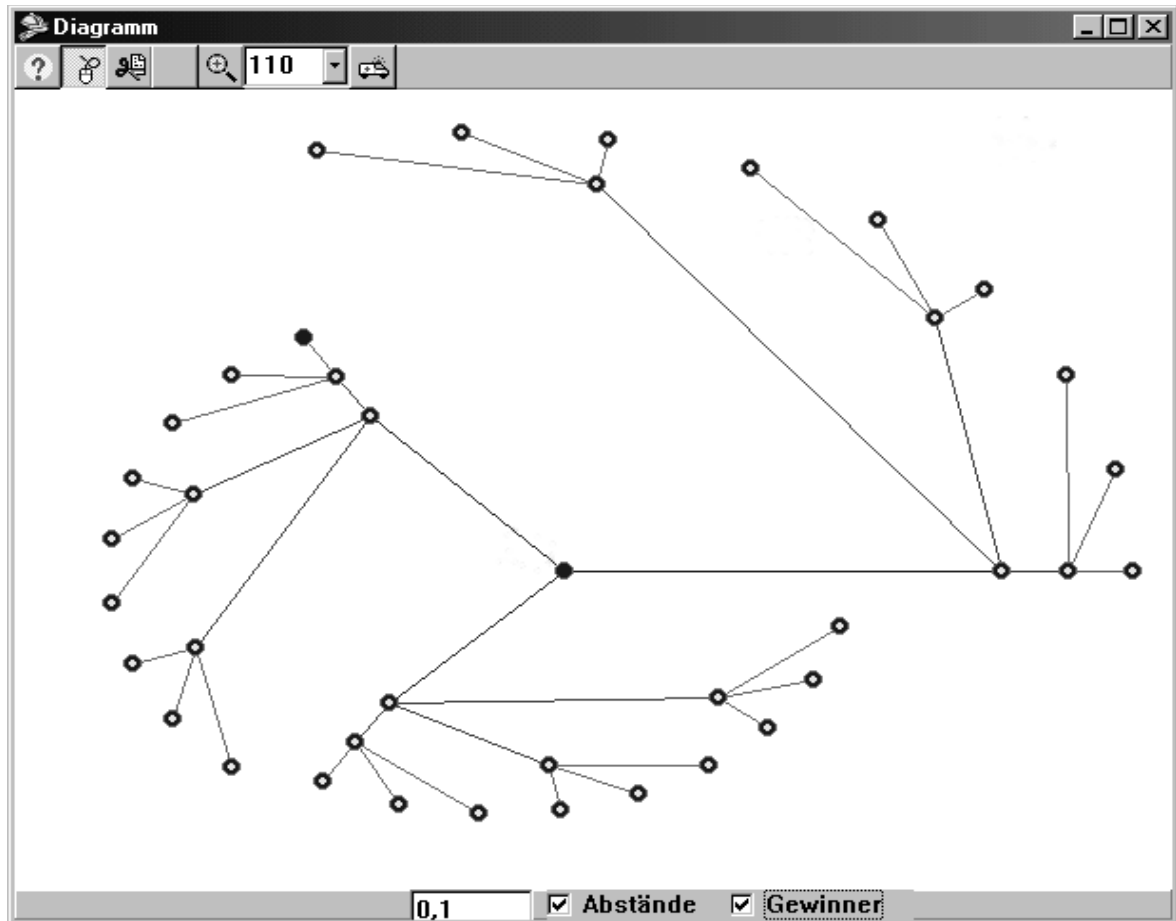


Abbildung 24 Das mit OSSU trainierte Netz nach 300 Epochen in der Polarkoordinatendarstellung

Schaut man sich die Polarkoordinatendarstellung des hierarchischen neuronalen Netzes an (Abbildung 24), so lassen sich Parallelen mit der Darstellung im kartesischen Koordinatensystem (Abbildung 22) ziehen. In der Polarkoordinatendarstellung sieht man, daß es eine Gruppe von Neuronen gibt, deren Abstand zum Wurzelneuron signifikant höher ist, als bei den anderen beiden Gruppen. Dies ist auch in dem kartesischen Diagramm zu sehen. Dabei ist es der rechte obere Cluster, der im Polardiagramm auch rechts oben zu sehen ist. An diesem Beispiel läßt sich gut erkennen, was mit der Repräsentation der

Abstände im euklidischen Sinne gemeint ist. Stellt man sich Kreise vor, die um das Wurzelneuron herum gezogen sind und ein Kindneuron schneiden, so repräsentiert der Kreisdurchmesser die euklidische Distanz im  $n$ -dimensionalen Raum vom Gewichtsvektor des Kindneurons zum Gewichtsvektor des Wurzelneurons.

## ***7.2 Ordered Search Winner Update (OSWU)***

Da der OSWU Algorithmus immer nur den Gewichtsvektor eines Neurons aktualisiert und die Aktualisierung von Gewichtsvektoren, deren Neurone in anderen Ebenen der Hierarchie liegen, vernachlässigt wird, ist mit einer höheren Anzahl von Epochen zu rechnen, bis sich der Zustand des Netzes stabilisiert hat.

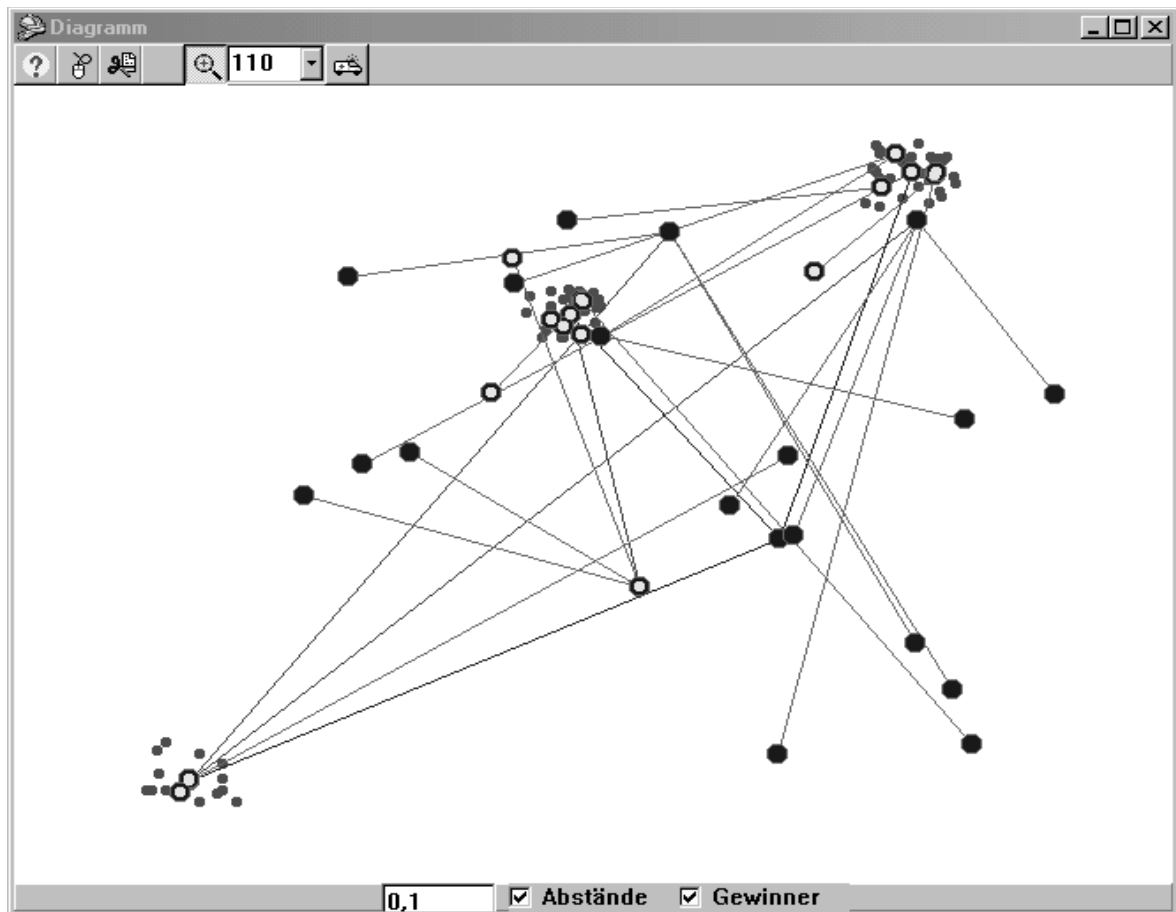


Abbildung 25 OSWU Training nach 400 Epochen

Sieht man sich nach 400 Epochen das kartesische Diagramm des Netzes an, so sieht man eine noch sehr schlechte Passung an die Trainingsmuster. Nach 400 Epochen erreichte das Netz mit dem OSSU Algorithmus bereits einen stabilen Zustand. Im Diagramm sind es vor allem die Neurone der untersten Ebene, deren Gewichtsvektoren noch nicht aktualisiert worden sind. Dies wird erst der Fall sein, wenn der Parameter Frequenz (siehe Kapitel 4) der Neurone, die schon eine recht gute Passung erreicht haben, hoch genug wird um bei der geordneten Suche einen höheren Fehlerwert zu produzieren als bei den noch nicht aktualisierten Neuronen.

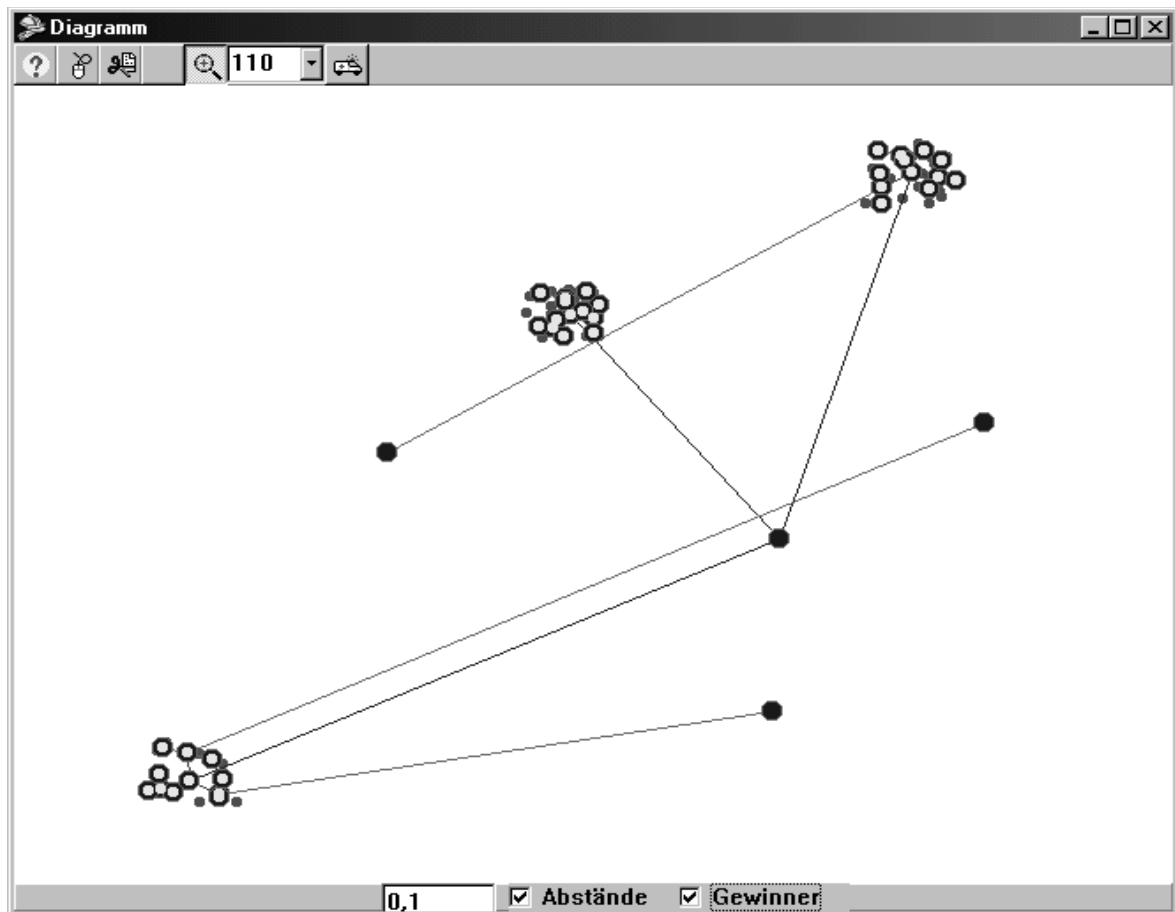


Abbildung 26 OSWU Training nach 6000 Epochen

Läßt man das Training bis zu Epoche 6000 durchlaufen, so sieht man eine gute Anpassung der Gewichtsvektoren an die Trainingsmuster (Abbildung 26). Doch auch zu diesem Zeitpunkt gibt es noch drei Neurone der untersten Ebene, die sich nicht hinreichend an die Trainingsmuster angepaßt haben. Das Training mußte noch bis zu Epoche 10.000 fortgesetzt werden, bis sich zumindest zwei der drei Gewichtsvektoren an die Trainingsmuster angepaßt haben. Auf diese Darstellung wird hier verzichtet, da sie sich nur noch marginal von Abbildung 26 unterscheidet.

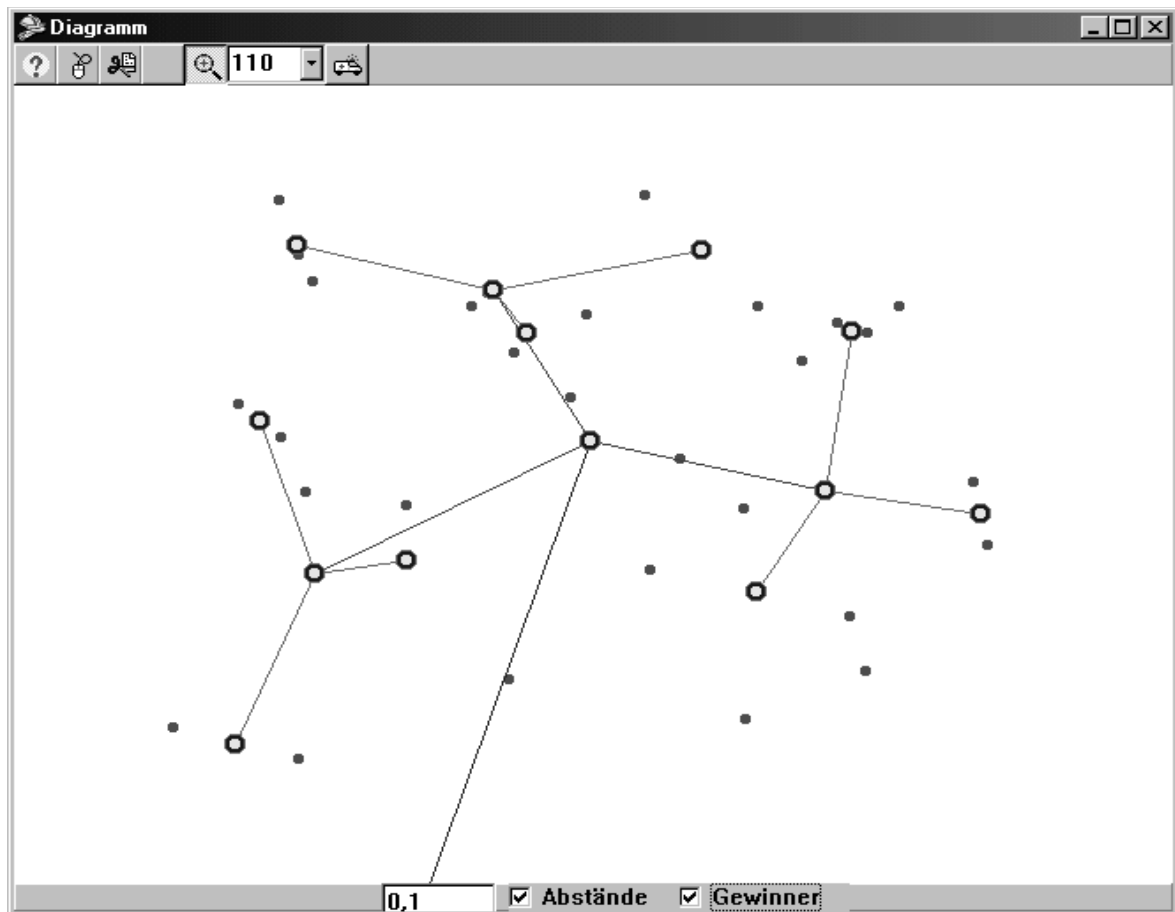


Abbildung 27 OSWU Training nach 10.000 Epochen, Zoom auf den rechten oberen Cluster

Betrachtet man sich nun den rechten oberen Cluster genauer (Abbildung 27), so sieht man eine bessere Passung an die Trainingsmuster als mittels des OSWU Algorithmus erzielt werden konnte. Auch die Randbereiche des Clusters werden hier recht gut von den Gewichtsvektoren abgedeckt. Die Generierung eines prototypischen Musters durch den Gewichtsvektor des Neurons auf Ebene 1 scheint hier genauso gut gewährleistet zu sein, wie beim OSSU Algorithmus. Natürlich erkaufte man sich diese bessere Passung an die Trainingsmuster durch einen erheblich höheren Rechenaufwand. Es waren 10.000 Epochen Training nötig, im Gegensatz zu 400 Epochen im Fall des OSSU Algorithmus.

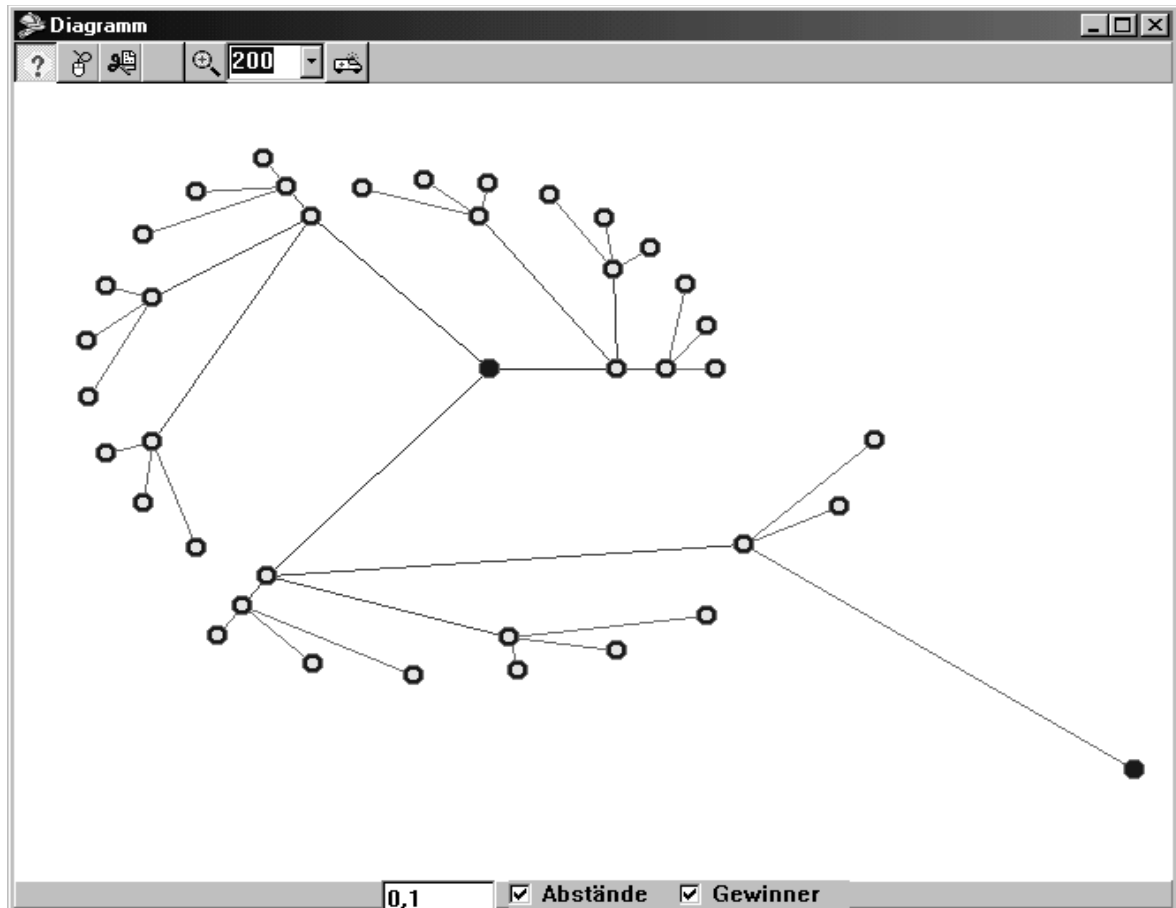


Abbildung 28 Polarkoordinatendarstellung nach Epoche 10.000 des OSWU Trainings

Selbst nach 10.000 Epochen gibt es immer noch einen Gewichtsvektor, der sich nicht gut an die Trainingsmuster angepaßt hat. In dem Polardiagramm (Abbildung 28) ist dieser Effekt sehr schön zu sehen. Während die Geschwister des Ausreißer-neurons und alle anderen Neurone der untersten Ebene in etwa den gleichen Abstand zu ihren Eltern haben, ist die euklidische Distanz des Ausreißers zu seinem Elternneuron merklich höher. Dies wäre ein Fall für die von Li et al. (1993) vorgeschlagene Optimierung des hierarchischen neuronalen Netzes durch das Ausschneiden von Neuronen, deren Frequenz nach einigen hundert Epochen noch nahe Null liegt.

### ***7.3 Exhaustive Search Path Update (ESPU)***

Dieser Trainingsalgorithmus müßte seiner Definition nach mit weniger Trainingsepochen auskommen als der OSWU Algorithmus, da hier wie beim OSSU Algorithmus die Gewichtsvektoren mehrerer Neurone pro Trainingsmuster aktualisiert werden. Zum einen werden, wie beim OSSU Algorithmus, die Gewichtsvektoren der Neurone des Unterzweigs vom Gewinnerneuron aktualisiert (Subtree Update) als auch die Gewichtsvektoren der in der Hierarchie darüberliegenden Neurone, zwischen Gewinnerneuron und Wurzelneuron.

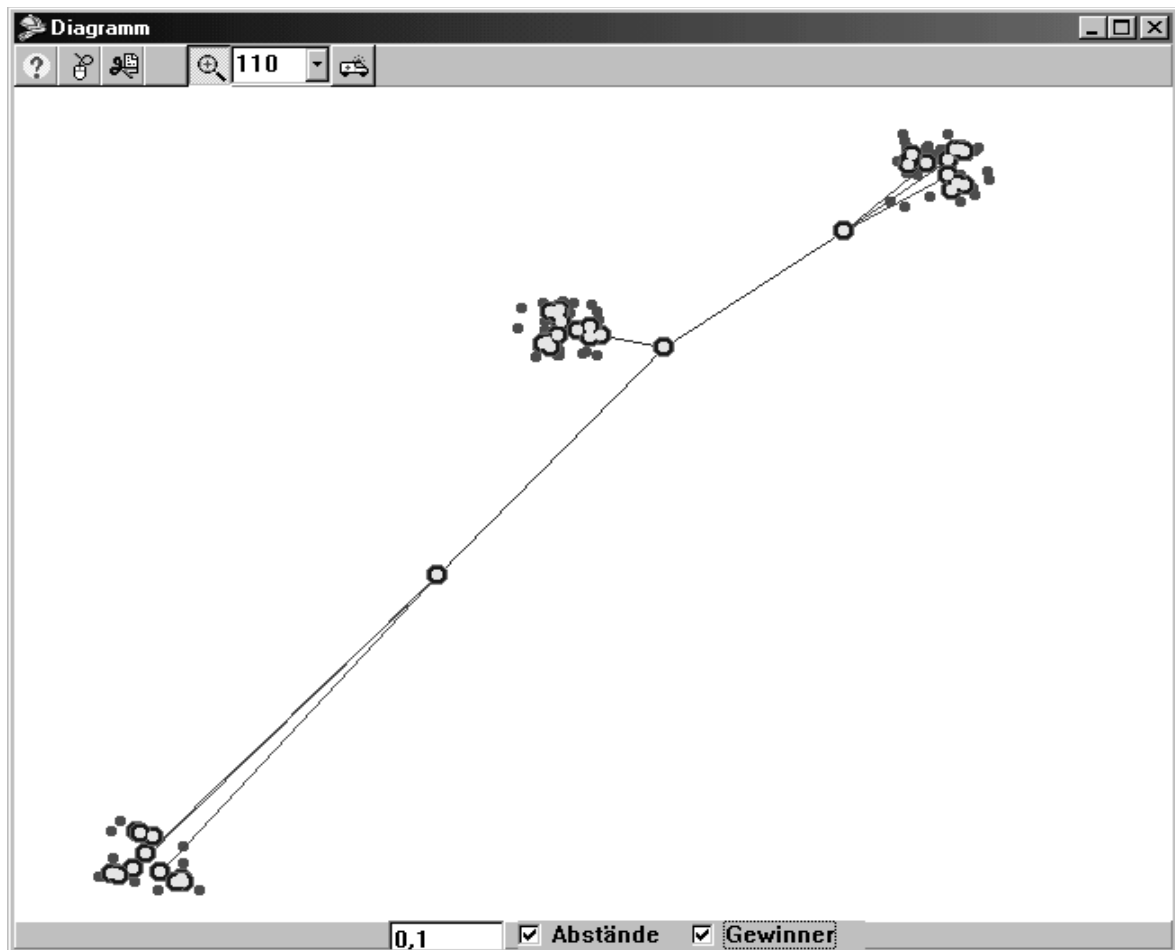


Abbildung 29 ESPU Training nach 400 Epochen

Nach 400 Epochen ist bereits die Endkonfiguration des Netzes erreicht (Abbildung 29). Im Unterschied zu den beiden anderen vorgestellten Algorithmen kann man sehen, daß die Neurone der ersten Ebene dazu tendieren, sich oft nicht optimal anzupassen. Dieses Phänomen ist dem ESPU Algorithmus zuzuschreiben. Daraus kann man ableiten, dass sich dieser Algorithmus nicht gut zur Bildung von Prototypen anhand der Gewichtsvektoren der Neuronen in den oberen Ebenen der Hierarchie eignet.



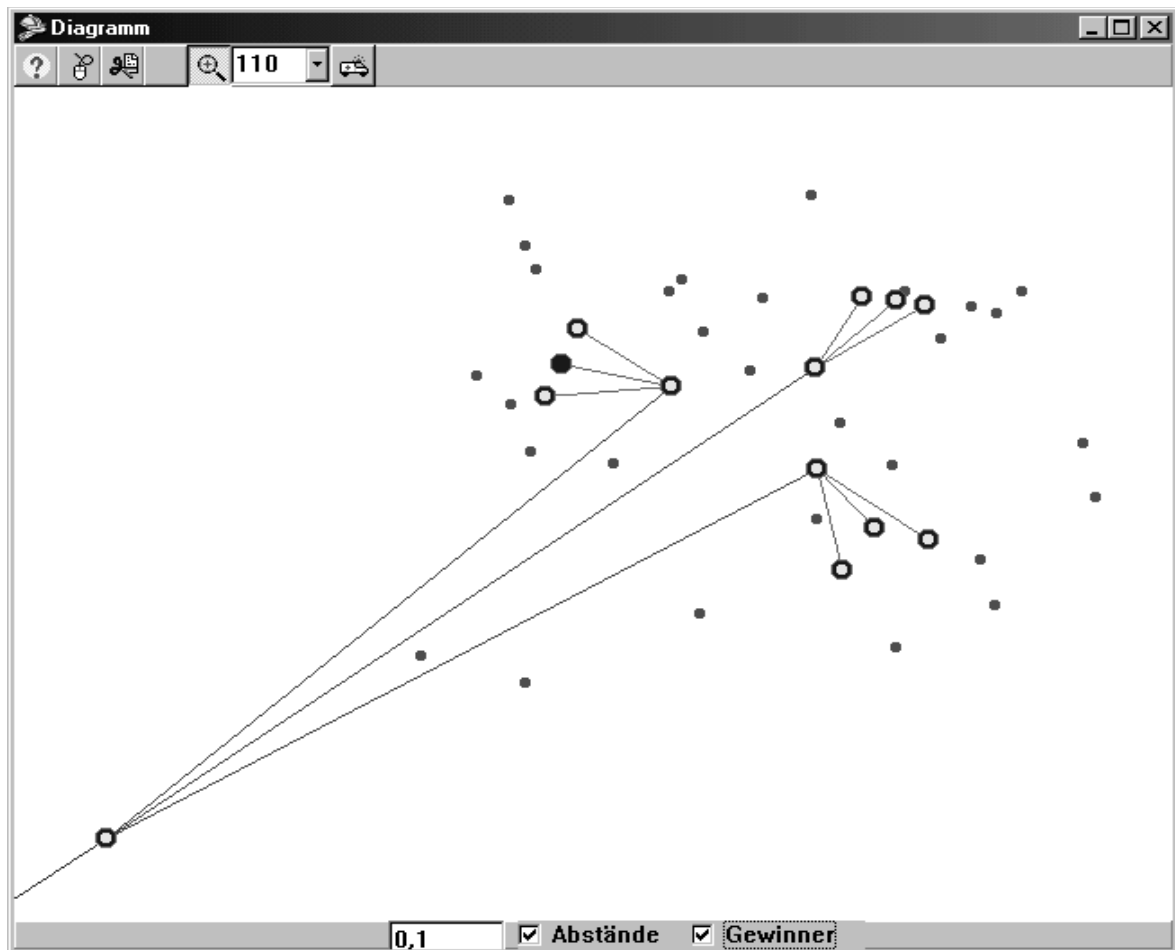


Abbildung 30 Zoom auf den rechten oberen Cluster nach 400 Epochen des ESPU Trainings

Sieht man sich den rechten oberen Cluster der Trainingsmuster an (Abbildung 30), fällt die Tatsache der schlechten Eignung zu Prototypengenerierung besonders auf. Das Neuron der Ebene 1 (Kind vom Wurzelneuron) liegt außerhalb des Clusters. Sein Gewichtsvektor dient somit als schlechter Ausgangspunkt, um ein Trainingsmuster zu ermitteln, welches repräsentativ für den gesamten Cluster sein sollte. Die Passung an die Daten durch die Neurone der untersten Ebene ist jedoch mit der Leistung des OSSU Algorithmus vergleichbar.

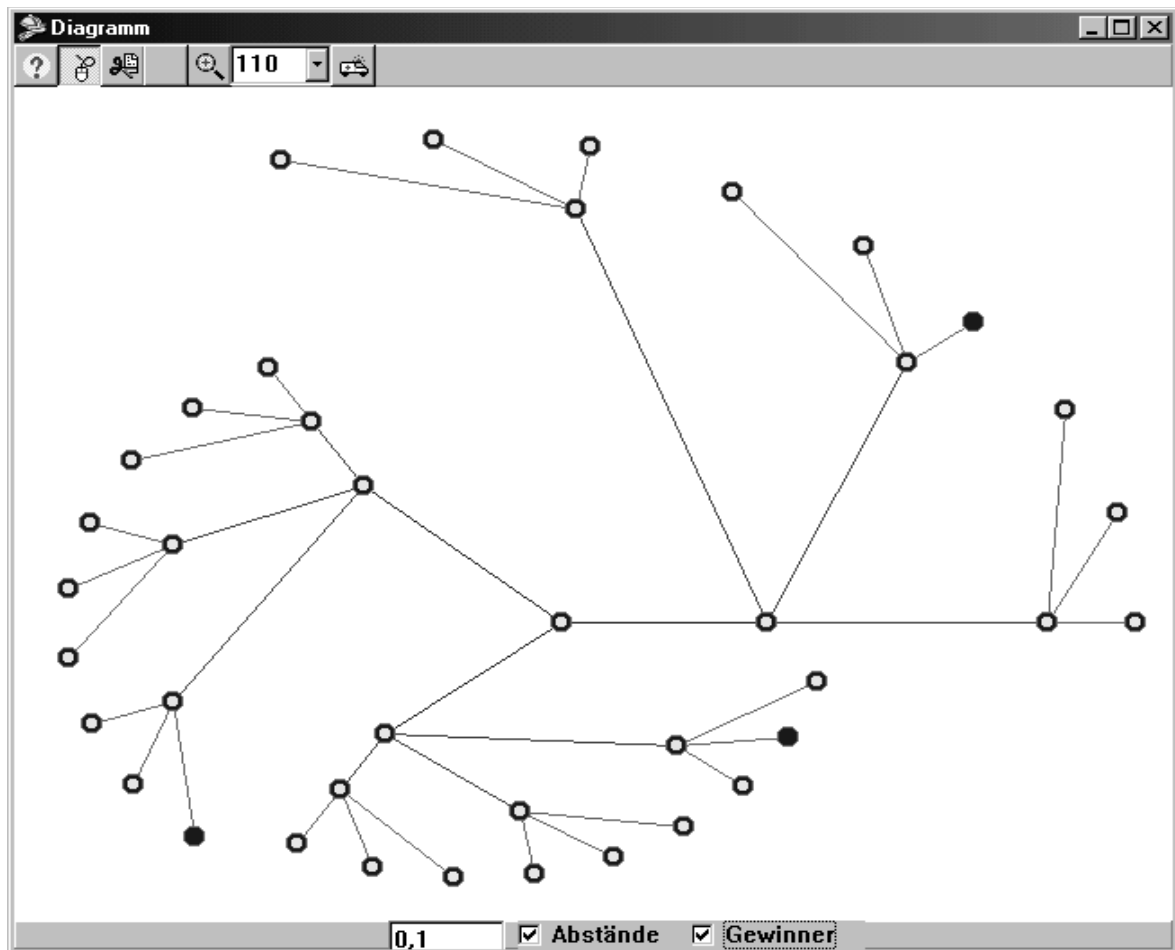


Abbildung 31 Polardiagramm nach 400 Epochen des ESPU Trainings

Schaut man sich nach 400 Epochen das Polardiagramm des Netzes an (Abbildung 31), so sieht man, daß die Neurone der ersten Ebene näher an dem Wurzelneuron liegen als bei den anderen beiden Polardiagrammen. Dies ist ein weiterer Beleg dafür, daß sich die Gewichtsvektoren dieser Neurone schlecht für die Generierung von prototypischen Mustern eignen. Für diesen Anwendungsfall ist es günstiger, wenn die Neurone der oberen Ebenen dicht an ihren Kindern liegen.

## 7.4 Zusammenfassung

Alle Algorithmen passen sich recht gut an die Trainingsmuster an. Die drei Cluster werden

auch jeweils von den drei Kindern des Wurzelneurons klassifiziert. Dies gelingt bei den Algorithmen OSWU und OSSU am besten, da sich hier die Gewichtsvektoren der Neurone der Ebene 1 am besten zur Generierung prototypischer Muster aus deren Gewichtsvektoren eignen. Die Passung an die Daten wird vom OSWU Algorithmus am besten geleistet, wenn auch unter dem Nachteil einer sehr hohen Anzahl an Trainingsepochen, die das hierarchische neuronale Netz durchlaufen muß. Für diesen Anwendungsfall ist der OSSU Algorithmus am besten geeignet, will man einen Kompromiss zwischen Schnelligkeit und guter Passung an die Trainingsmuster erreichen. Der ESPU Algorithmus fällt hier trotz seiner schnellen Ergebnisse zurück. Dies steht in Einklang mit den von Li et al (1993) dargestellten Leistungswerten des ESPU Algorithmus. Der optimale Weg bei diesem Datensatz scheint ein anfängliches Training mit dem OSSU Algorithmus zu sein. Nach der recht schnellen Stabilisierung des Netzes kann man auf den OSWU Algorithmus umschalten, um eine bessere Passung an die Daten zu erreichen ohne eine lange Wartezeit in Kauf nehmen zu müssen, bis sich die Gewichtsvektoren von Ausreißern an die Trainingsmuster angepaßt haben. Ein anderer denkbarer Weg ist die Kombination des OSWU Algorithmus mit dem Ausschneiden von Neuronen, deren Frequenz nach mehreren hundert Epochen immer noch nahe Null liegt. Jedoch muß man für diesen Fall ein Netz hinreichender Größe konstruieren, damit noch genügend Neurone diese Wegoptimierungsoperation überleben.

## 8 Diskussion und Ausblick

### 8.1 Diskussion

Mit dieser Arbeit sollte ein System zur numerischen Modellierung einer neuen Klasse von künstlichen neuronalen Netzen geschaffen werden. Im Gegensatz zu den frühen Modellen neuronaler Netze stellt dieses System eine höhere Abstraktion von dem an die Physiologie angelehnten Grundgedanken dar. Kohonen (1982) zeigte erstmals auf, daß man das Verhalten von künstlichen neuronalen Netzen mit den Werkzeugen der Vektoralgebra so vereinfachen kann, daß die numerische Simulation auf dem Computer wesentlich leichter zu implementieren ist. Li et al. (1993) modifizierten Kohonens Modell, so daß nun eine noch höhere Abstraktionsstufe erreicht ist. Sie taten dies jedoch in Hinblick auf mathematisch-ingenieurwissenschaftliche Anwendungsfelder. Psychologische Anwendungsfelder zu hierarchischen neuronalen Netzen sind hingegen noch unbekannt. Da diese Netze zu der Klasse der selbstorganisierenden und topologieerhaltenden Netze gehören, ist anzunehmen daß es psychologische Anwendungsfelder gibt. Mit selbstorganisierenden, topologieerhaltenden Netzen wurden in der Vergangenheit psychologische Problemstellungen untersucht. Diese neue Klasse von neuronalen Netzen erlaubt es, neben topologieerhaltenden Abbildungen auch eine hierarchische Klassifizierung von Trainingsmustern vorzunehmen. Diese neue Eigenschaft ermöglicht die Konstruktion von psychologischen Modellen mittels einer neuen Klasse von neuronalen Netzen auf der bisher höchsten Abstraktionsebene.

Die hierarchische Gruppierung von Gedächtnisinhalten ist ein zentraler Bestandteil von Servan-Schreibers (1990) Theorie zum Competitive Chunking. Servan-Schreibers Theorie macht Aussagen über die hierarchische Gruppierung von einfachen Gedächtnis-

einheiten, sogenannte Chunks, zu Einheiten auf der nächsthöheren Abstraktionsstufe in der Hierarchie der Chunks. Würde man ein hierarchisches Neuronales Netz mit den von Servan–Schreiber beschriebenen einfachen Chunks als Trainingsmuster trainieren, so stellt sich die interessante Frage, ob diese numerische Simulation zu ähnlichen Ergebnissen kommt, wie sie von Servan–Schreiber vorhergesagt werden. Hiermit würden Prozesse des Competitive Chunking transparenter und einem numerischen Modell zugänglich gemacht. Dieses Modell könnte dann mit Humandaten auf seine Validität geprüft werden.

Der Zweck dieser Arbeit war es, auch Psychologen einen Zugang zu dieser neuen Klasse von selbstorganisierenden neuronalen Netzwerken zu geben. Dieser Zugang wurde hier durch eine einfach zu bedienende Benutzeroberfläche und einer stark visuell orientierten Darstellung des Verhalten eines hierarchischen neuronalen Netzes erreicht. Diese Arbeit zeigt dem Anwender wie diese Diagramme zu interpretieren sind und wie man unterschiedliche Rückschlüsse aus der Anordnung der Gewichtsvektoren untereinander oder zu den Trainingsmustern ziehen kann. Den Trainingsmustern kommt eine zentrale Bedeutung zu. Bei Untersuchungen von Fragestellungen aus der Wahrnehmungspsychologie ist es vergleichsweise einfach sich über die Konstruktion der Trainingsmuster klar zu werden. Untersucht man beispielsweise die akustische Wahrnehmung mit neuronalen Netzen, so sind adäquate Trainingsmuster recht einfach abzuleiten. Acs (1995) hat hierzu Trainingsmuster verwendet, die den Aktivierungen einzelner Neurone in der Cochlea bei der Schallwahrnehmung entsprechen. Bei Fragestellungen, die sich mit dem Wahrnehmungsapparat des Menschen beschäftigen, versucht man, die Trainingsmuster von der Neuronenaktivität der akustischen, visuellen, taktilen, olfaktorischen, thermischen Rezeptoren abzuleiten. Beschäftigt man sich mit anderen Fragestellungen, bei denen es nicht eindeutig

ist, welche Trainingsmuster nun genau in das zu untersuchende System einwirken, ist eine sorgfältige Konstruktion der Trainingsmuster, basierend auf bisherigen Erkenntnissen, des zu untersuchenden Gegenstandes, absolut unerlässlich.

Die Adaptation an psychologische Projekte, die hierarchische neuronale Netze zur Modellbildung heranziehen wollen, wurde ermöglicht. Dies geschah durch die Implementierung der Algorithmen in einer bekannten und einfach zu bedienenden Programmieroberfläche. Die Dokumentation dieser Schnittstellen für den interessierten psychologischen Anwendungsentwickler war ein anderer Hauptaspekt dieser Arbeit. Durch den modularen Programmaufbau und die Kapselung der Schlüsselfunktionen in sinnvolle Funktionseinheiten sollte die Modifikation dieses Systems, um an unterschiedliche psychologische Fragestellungen angepaßt werden zu können, ermöglicht werden. Zu den Erweiterungen könnte beispielsweise eine Parameterausgabe mit verschiedenen statistischen Kennwerten zählen. Da das vorliegende System objektorientiert ist, ist es gut möglich, beispielsweise die Objekte Trainer (5.6) und Neuron (5.3) so zu erweitern, daß man aus Ihnen Objekte ableitet die solche Parameter protokollieren. Einer der wichtigsten Parameter, aus dem sich die Passung des Netzes an Trainingsmuster ableiten läßt (Funktion TNeuron.error in 5.3.8), ist schon enthalten.

Eine andere Frage ist die nach einem geeigneten Abbruchkriterium des Trainings. Zum jetzigen Entwicklungsstand wird das Training des hierarchischen neuronalen Netzes solange fortgesetzt, bis es vom Anwender abgebrochen wird. Es ist denkbar, die Summe der Fehler jedes Neurons zu den Trainingsmustern zu bilden, bei denen sie als Gewinner ermittelt wurden. Dies wäre der Gesamtfehler des Systems. Man könnte nun einen Schwellenwert definieren, ab dem man die Passung des Netzes als hinreichend annimmt.

Das Training kann in diesem Fall durch ein modifiziertes Trainer Objekt (5.6) dann abgebrochen werden, wenn dieser Schwellenwert unterschritten wird. Die Protokollierung der Fehlerwerte könnte während des Trainings durch ein modifiziertes Neuron Objekt (5.3) erfolgen. Die Auswertung, Normierung und Darstellung des Gesamtfehlers läßt sich idealerweise in einer modifizierten Fassung des NeuronInspektor Objekts (5.11) realisieren.

## ***8.2 Ausblick: Implementationen zum impliziten Lernen***

Ein mögliches Anwendungsfeld hierarchischer neuronaler Netze soll hier nur kurz angeschnitten werden. Beim impliziten Lernen geht es darum, daß Versuchspersonen Regelmäßigkeiten in ihrem Verhalten erlernen, ohne daß ihnen bewußt ist, daß es sich hierbei um eine Lernaufgabe handelt. Dies kann beispielsweise dadurch geschehen, daß Versuchspersonen Bedienoperationen auf einem am Computer sichtbaren Feld von Knöpfen vornehmen müssen (Müller 1996). Die Bedienoperationen sind in Sequenzen angeordnet, die einer gewissen Regelmäßigkeit entsprechen. Diese Regelmäßigkeit entsteht aus einer künstlich generierten Grammatik. Werden nun die Versuchspersonen nach mehreren Trainingsläufen darauf hingewiesen, daß die Bediensequenzen einer Regelmäßigkeit entsprachen, beginnt die eigentliche Testphase. Den Versuchspersonen werden nun Bediensequenzen präsentiert, die teilweise der Regelmäßigkeit entsprechen und teilweise nicht. Wie Müller (1996) zeigen konnte, waren die Versuchspersonen in einem signifikanten Maße dazu in der Lage, Sequenzen zu differenzieren, und zwar nach dem Kriterium ob sie der künstlich generierten Grammatik entsprachen oder nicht. Dies ist natürlich auch mit anderen Reizen, wie Buchstabenfolgen, die durch eine künstliche Grammatik generiert

werden, denkbar. Das zugrundeliegende Gedächtnismodell wurde von Servan-Schreiber (1990) vorgestellt. Es wird dabei angenommen, daß das implizite Lernen durch competitive chunking stattfindet. Eingabereize werden in unserem Gedächtnis nach Ähnlichkeit gruppiert und in eine hierarchische Ordnung gebracht. Nachdem Cleeremans (1991) zeigen konnte, daß man implizites Lernen von Versuchspersonen mit einem einfachen künstlichen neuronalen Netz simulieren kann, stellt sich nun die Frage, ob sich diese Resultate nicht mit einem hierarchischen neuronalen Netz replizieren lassen. Man erhielte durch den hierarchischen Aufbau des Neuronalen Netzes die Struktur der von Servan-Schreiber beschriebenen Gedächtniseinheiten als Nebenprodukt. Hierbei ist die Frage nach der Strukturierung der Trainingsmuster einmal mehr wichtig. Im einfachsten Falle würde man Buchstabenfolgen untersuchen, denen eine künstliche Grammatik zugrunde liegt. Man kann recht plausibel argumentieren, daß einzelne Buchstaben von unserem kognitiven Apparat wahrgenommen werden und daß kein Buchstabe eine größere Bedeutung als ein anderer hat. Daher kann man eine solche Buchstabensequenz als einen Vektor codieren. Bestünde die Buchstabensequenz aus sechs Buchstaben, so hätten wir es mit sechsdimensionalen Vektoren zu tun, die unsere Trainingsmuster bilden. Die Werte der einzelnen Komponenten der Trainingsmuster würden die Buchstaben A bis Z codieren, was im einfachsten Falle mit den Werten 1 bis 26 geschehen könnte, d.h. jedem Buchstaben wird eine Zahl zugeordnet. Die so erzeugten Trainingsmuster verwendet man nun zum Training des hierarchischen Neuronalen Netzes. Anhand der daraus entstehenden Konfiguration kann man nun überprüfen, ob sich die einzelnen Neurone in einer ähnlichen wie der von Servan-Schreiber beschriebenen Konfiguration anordnen.

Ferner ist es möglich, einen Vergleich mit Humandaten durchzuführen. Im einfachsten



Fall würde man dem Netz Sequenzen präsentieren, die der Grammatik entsprechen und andere, die ihr nicht entsprechen. Die Trennungsleistung des Netzes hinsichtlich der Gruppierung in grammatische und nicht-grammatische Sequenzen könnte man mit der Diskriminationsleistung von menschlichen Versuchspersonen vergleichen.

Die Untersuchung des impliziten Lernens ist nur eines von vielen möglichen Anwendungsmöglichkeiten hierarchischer Neuronaler Netze in der psychologischen Forschung.

### ***8.3 Ausblick: künstliche neuronale Netze in der Psychologie***

Die zur Verfügung stehende Rechenkapazität moderner Personal-Computer macht es heute möglich, neuronale Netze großer Neuronenzahl und komplexe Sätze an Trainingsmustern zu verwalten. Noch vor zehn Jahren wäre es nicht denkbar gewesen, derart umfangreiche Berechnungen an einem durchschnittlichen EDV-Arbeitsplatz durchzuführen. Zum einen hätte das Training eines hierarchischen neuronalen Netzes mehrere Stunden bis Tage in Anspruch genommen, zum anderen war damals der Arbeitsspeicher dieser EDV-Arbeitsplätze nicht ausreichend, um umfangreiche Datensätze an Trainingsmustern zu verwalten. Es ist anzunehmen, daß sich diese Entwicklung in der Zukunft fortsetzen wird. Dadurch wird es möglich, immer komplexere psychologische Fragestellungen mithilfe von neuronalen Netzen zu untersuchen. Denkbar wäre eine Entwicklung, die der Geschichte der Meteorologie entspricht. Bis in die 50er Jahre hinein waren Meteorologen auf viele einzelne Theorien und Einzelbeobachtungen des Wettergeschehens angewiesen. Mit der Verfügbarkeit immer leistungsfähigerer Computer konnten mehr Messdaten gesammelt und ausgewertet werden. Dies trug zur Entwicklung numerischer Modelle des Wettergeschehens bei. Die numerischen Wettermodelle waren ihrerseits

nichts weiter als eine Simulation des Beobachteten, mit dem Versuch der Extrapolation. Mißglückte Extrapolationen haben zu einer Verbesserung der Wettermodelle geführt. Diese besseren Modelle ermöglichten wiederum die Generierung neuer Hypothesen und Modelle zur Meteorologie. Neuronale Netze sind nichts anderes als die numerische Simulation kognitiver Prozesse. Da auf den menschlichen Geist eine Vielzahl von Faktoren einwirken und er einer Eigendynamik unterworfen ist, ist zu erwarten, daß numerische Simulationen von menschlichen Verhaltensweisen auch extrem komplex werden. Die Verbesserung der numerischen Modelle aufgrund fehlerhafter Extrapolationen menschlichen Verhaltens könnte ihrerseits wieder zur psychologischen Theoriebildung auf der nicht-numerischen Ebene beitragen. Ansätze dazu sind beispielsweise in der numerischen Modellierung der Gesetze aus der Gestaltpsychologie zu sehen (Acs, 1995). Das Hintergrund Problem der Gestaltpsychologie ist mit numerischen Verfahren faßbar geworden, eventuell werden neuronale Netze eines Tages die Antwort auf die Frage liefern, wie die Funktionsweise der Wahrnehmungsprinzipien aus der Gestaltpsychologie zu verstehen ist. Es ist anzunehmen, daß zunächst Phänomene aus dem Bereich der Wahrnehmungspsychologie erklärbar werden, da hier die Frage nach der Konstruktion der Trainingsmuster recht einfach zu beantworten ist. Weiterhin muß lediglich die Frage beantwortet werden, welche Faktoren von außen auf unseren kognitiven Apparat einwirken. Diese Größen sind einer Messung leicht zugänglich. Die numerische Modellierung kognitiver Vorgänge ist, nach Ansicht des Verfassers, eine nicht zu vernachlässigende Größe in der psychologischen Forschung. Sie könnte dazu beitragen, neue und allgemeinere Theorien des kognitiven Apparates zu formulieren.

## 9 Literaturverzeichnis

Acs, –F. (1995). „Zeitliche synchronisierung von Neuronenverbänden als Lösungsansatz zum Bindungsproblem.“, Semesterarbeit an der Justus–Liebig–Universität Gießen. Fachbereich Psychologie

Bechtel, –W. & Abrahamsen, –A. (1991). „Connectionism and the mind: an introduction to parallel processing in networks“, Cambridge Mass. Basil Blackwell.

Blum, –A. (1992). „Neural networks in C++: An object–oriented framework for building connectionist systems“, New York, John Wiley & Sons inc.

Cantu, –M. (1997). „Mastering Delphi 3: Second edition“, San Francisco, Sybex.

Cleeremans, –A., McClelland, –L., –J. (1991). „Learning the structure of event sequences“, Journal of Experimental Psychology: General, Vol. 120, No. 3: 235–253

Jänich, –K. (1991). „Lineare Algebra“, 4. Auflage, Berlin, Heidelberg, Springer Verlag

Kohonen, –T. (1982). „Self–organized formation of topologically correct feature maps.“, Biological Cybernetics, 43: 59–69

Kohonen, –T.(1984). „Self–organization and associative memory.“, Heidelberg, Springer Series in Information Sciences

Li, –T., Fang, –L. & Li, –K. (1993). „Hierarchical classification and vector quantization with neural trees“, Neurocomputing, 5: 119–139

von der Mahlsburg, –C. & Wilshaw, –D., –J. (1977) „How to label nerve cells so that they can interconnect in an ordered fashion“, Proceedings of the National Academy of Science, USA, 74: 5176–5187

Matcho, –J., Salmanowitz, –B., Strool, –S., Biely, –B., Jurkouich, –S., –T., Berry, –S., Sleeper, –L., Dumbrill, –D., Uber, –E. (1996). „Using Delphi 2: Special Edition“, Indianapolis, Que Corporation

McClelland, –J., –L., & Rumelhart, –D., –E. (1981). „An interactive activation model of context effects in letter perception: Part I. An account of basic findings“, *Psychological Review*, 88: 375–407

McClelland, –J., –L., & Rumelhart, –D., –E. (1986a). „Parallel Distributed Processing. Explorations in the microstructure of cognition. Volume 1: Foundations“, MIT Press, Bradford Books.

McClelland, –J., –L., & Rumelhart, –D., –E. (1986b). „On learning the past tenses of english verbs. In: Parallel distributed processing. Explorations in the microstructure of cognition. Volume 2: psychological and biological models“, MIT Press, Bradford Books.

McClelland, –J., –L., & Rumelhart, –D., –E. (1988). „Explorations in parallel distributed processing. A handbook of models, programs and exercises“, MIT Press, Bradford Books.

Müller, –B. (1996). „Zulässigkeitsurteile und Bediensequenzen: Der Einfluß zeitlicher Sukzessivität auf die Kompositionsbildung“, *Zeitschrift für Psychologie*, 204: 281–303

Ritter, –H., Martinetz, –T. & Schulten, –K. (1991). „Neuronale Netze: eine Einführung in die Neuroinformatik selbstorganisierender Netzwerke“, Bonn, München, Addison Wesley.

Saarinen, –J. & Kohonen, –T. (1985). „Self–organized formation of colour maps in a model cortex.“, *Perception*, Vol 14 (6): 711–719

Servan–Schreiber, –E. & Anderson, –J., –R. (1990). „Learning artificial grammars with competitive chunking“, *Journal of Experimental Psychology: Learning, Memory and*

Cognition, Vol. 16, No. 4: 592–608

Toiviainen, –P., Tervaniemi, –M., Louhivouri, –J., Saher, –M., Houtilainen, –M. & Naeaetaenen, –R. (1998). „Timbre similarity: Convergence of neural, behavioral and computational approaches.“, Music Perception, Vol 16 (2): 223–241

Wilshaw, –D.J., von der Mahlsburg, –C. (1979). „A marker induction mechanism for the establishment of ordered neural mappings: Its application to the retinotectal problem.“  
Proceedings of the Royal Society, London, B 287: 203–243