Prüfer:	Prof. Dr. A. Reuter
Betreuer:	Dr. A. Zell

begonnen am:	1. August 1991
beendet am:	16. Dezember 1991
CR-Klassifikation:	I.5.1, I.5.3, I.6, C.1.2, C.1.3

Massiv parallele Simulation der selbstorganisierenden Karten von Kohonen auf einem SIMD-Rechner

Harald Bayer
Diplomarbeit Nr. 832

Fakultät Informatik
Institut für Parallele und
Verteilte Höchstleistungsrechner
Universität Stuttgart
Breitwiesenstraße 20–22
D 7000 Stuttgart 80

Inhaltsverzeichnis

1 Einführung

Das Ziel dieser Arbeit war der Entwurf und die Implementierung eines parallelen Simulators für die selbstorganisierenden Karten von Teuvo Kohonen[1], die dieser in [1]-[4] und [23] beschrieben hat. Selbstorganisierende Karten sind neuronale Netze, die eine topologieerhaltende Abbildung eines Raumes von Eingabevektoren auf die Gewichtsvektoren einer geordneten Menge von Neuronen bilden und dadurch eine Art Klassifikator darstellen und ein Ähnlichkeitsmaß zwischen den Eingabevektoren induzieren. Nach der Lernphase werden ähnliche Eingabevektoren auf benachbarte Neuronen abgebildet. Haben die Eingabevektoren eine höhere Dimensionalität als die selbstorganisierende Karte, die als n-dimensionaler Hyperkubus strukturiert ist, so reduziert der Algorithmus die Eingabevektoren auf die „wichtigen" Komponenten und klassifiziert den Eingaberaum nach diesen.

Da neuronale Netze und insbesondere selbstorganisierende Karten eine hochparallele Architektur sind, erfolgte die Implementierung auf der MasPar MP-1216A, einem SIMD-Parallelrechner mit $2^{14} = 16384$ Prozessoren. Die Steuerung sowie die Darstellung der Zwischenergebnisse erfolgt graphisch mittels X-Windows auf einer UNIX-Workstation. Implementierungssprachen sind MPL (**M**asPar **P**arallel Application **L**anguage, ein um parallele Konstrukte erweitertes C) auf der MasPar-Seite und C und X-Windows auf Seiten der Workstation.

Die MasPar besteht logisch aus zwei Komponenten: dem Frontend (FE) und der data parallel unit (DPU), die über einen VME-Bus miteinander verbunden sind. Das Frontend ist eine DEC-Station 5000/200 während die DPU aus einer array control unit (ACU) und dem Prozessorarray besteht. Die ACU hält alle skalaren („singulären") Daten, das Prozessorarray ist für alle parallelen Operationen zuständig und das Frontend bietet alle Ein-/Ausgabemöglichkeiten, auf ihm läuft auch das Betriebssystem.

Das folgende Kapitel beschreibt den Algorithmus der selbstorganisierenden Karten und deren mathematische Grundlagen.
Dessen Umsetzung sowie andere Implementierungsaspekte werden in Kapitel 3 beschrieben.
Kapitel 4 gibt eine Einführung in den Simulator und beschreibt die Oberfläche und deren Menüs, während Kapitel 5 die Möglichkeit diskutiert, die Semantik der Eingabevektoren zu ändern, um den Simulator dem speziellen Eingaberaum eines Problems anzupassen.
Kapitel 6 nennt die Software- und Hardwarevoraussetzungen zum Betrieb des Simulators sowie die nötigen Dateien und gibt an, wie die Programme übersetzt werden.
In Kapitel 7 werden mit dem Simulator erreichte Ergebnisse und Beispiele präsentiert und Eigenschaften selbstorganisierender Karten aufgezeigt.
Kapitel 8 stellt eine gleichwertige *serielle* Version des Simulators vor und Kapitel 9 enthält Leistungsmessungen des Simulators und Vergleiche mit der seriellen Version sowie in der Literatur erwähnten parallelen Simulatoren.
Kapitel 10 enthält Vorschläge für Weiterführungsmöglichkeiten dieser Arbeit.
Den Abschluß bilden verschiedene Anhänge.

[1]In der Literatur auch unter den Stichworten *feature maps*, *self-organization*, *neural mapping* und *topology-preserving neural maps* zu finden.

2 Selbstorganisierende Karten

Selbstorganisierende Karten sind neuronale Netze mit nur einer Schicht von Neuronen. Zwischen den Neuronen gibt es eine Nachbarschaftsbeziehung, d.h. sie sind in einer n-dimensionalen Struktur angeordnet. Dies ist ein wesentlicher Unterschied zwischen selbstorganisierenden Karten und anderen neuronalen Architekturen (z.B. feed-forward Netzen wie dem multilayer perceptron), bei denen die Neuronen meist als Schichten ungeordneter Mengen strukturiert sind. In der Literatur werden hauptsächlich eindimensionale (Ketten oder Ringe), zweidimensionale (rechteckige) oder dreidimensionale Netzstrukturen benutzt. Höherdimensionale Netzstrukturen sind denkbar und mathematisch problemlos. Da sie aber mit der üblichen und in dieser Arbeit verwendeten Visualisierungsmethode nicht darstellbar sind, werden diese im Folgenden nicht weiter diskutiert, die mathematischen Grundlagen werden trotzdem so allgemein dargestellt, daß eine Übertragung auf höherdimensionale Netzstrukturen kein Problem ist.

Die Eingaben in die Kohonenschicht erfolgen von außen über gewichtete Eingabeverbindungen. Alternativ kann man sich auch vorstellen, daß die Eingabe von einer (virtuellen) Schicht geordneter Neuronen erfolgt. Die Eingabevektoren haben eine beliebige Dimensionalität, die nicht mit der Netzdimension übereinstimmen muß, aber größer sein sollte als diese.

Alle Neuronen erhalten dieselben Eingaben, die aber noch durch die individuellen Gewichte eines jeden Neurons modifiziert werden. Dies ist die bei neuronalen Netzen übliche Sichtweise. Bei selbstorganisierenden Karten gibt es aber eine noch wesentlich anschaulichere Interpretation, die im Folgenden ausschließlich benutzt werden wird: Sowohl die Eingaben als auch die Gewichte werden als Ortsvektoren interpretiert. Das maximal aktivierte Neuron ist dann das Neuron, dessen Gewichtsvektor den geringsten Abstand zum Eingabevektor hat. Ein Lernschritt rückt ein Neuron, das durch seinen Gewichtsvektor repräsentiert wird, näher an den Eingabevektor, der es aktiviert hat, heran. Aber nicht nur das selektierte Neuron lernt, sondern in geringerem Maße auch die Neuronen in seiner Nachbarschaft. Hier kommt also die oben erwähnte Ordnung ins Spiel. Durch diese Nachbarschaftskopplung entsteht eine topologieerhaltende Abbildung des Raumes der Eingaben auf die Neuronenstruktur. Ähnliche Eingabevektoren aktivieren benachbarte Neuronen. Außerdem akquirieren häufige Eingaben mehr Neuronen als seltene Eingaben, wodurch sich eine Verteilung des Eingaberaumes auf die Neuronenstruktur ergibt, die auch von der Häufigkeitsverteilung der Eingabevektoren abhängt. Kohonen nennt diese Eigenschaft selbstorganisierender Karten in [1] den *magnification factor*.

Abbildung 1 zeigt das Schema einer selbstorganisierenden Karte. Alle Neuronen erhalten dieselbe Eingabe, aber nur das Neuron, dessen Gewichtsvektor dem Eingabevektor am „ähnlichsten" ist, reagiert mit einer Ausgabe. Abbildung 2 skizziert, wie dann die Gewichte des selektierten Neurons und dessen Nachbarn dem Eingabevektor angenähert werden.

Die aktuelle Konfiguration eines Kohonen-Netzes kann dargestellt werden, indem jedes Neuron an dem Ort gezeichnet wird, der durch seinen als Ortsvektor interpretierten Gewichtsvektor bestimmt wird. Direkt benachbarte Neuronen werden zusätzlich durch Linien

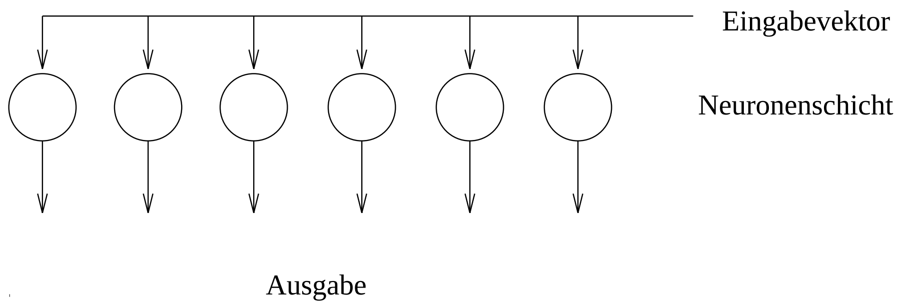

Abbildung 1: Vereinfachtes Schema einer selbstorganisierenden Karte. Die Ausgabe des selektierten Neurons ist 1, alle anderen geben 0 aus.

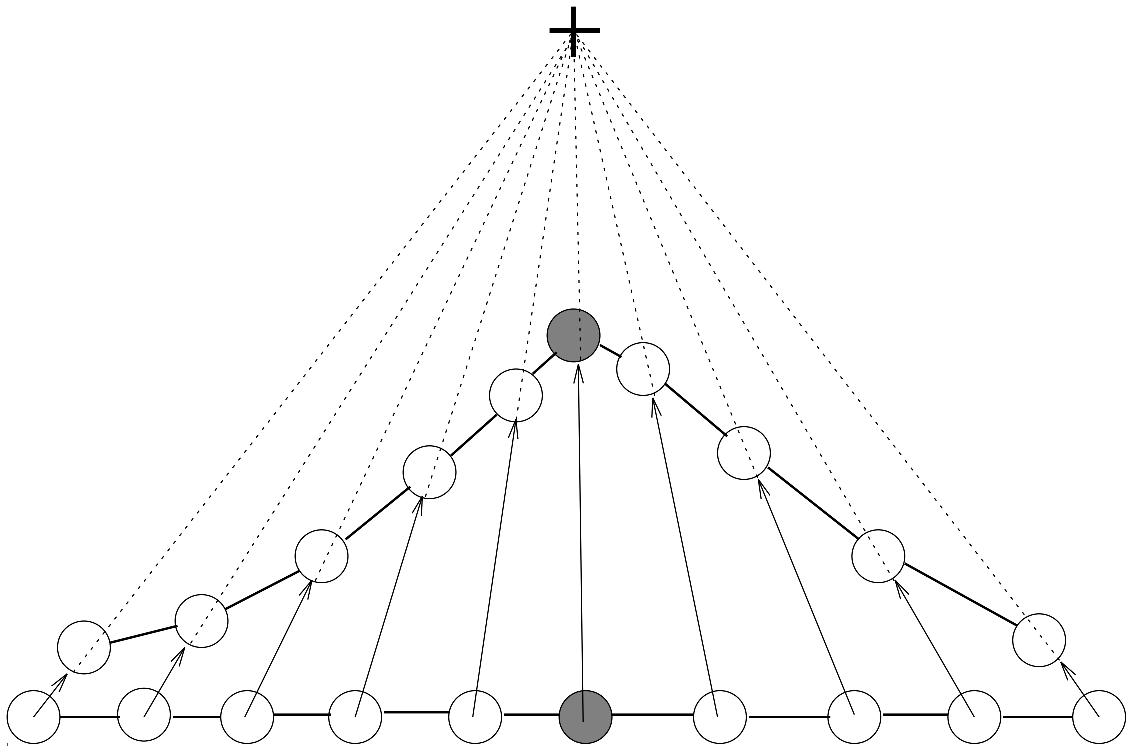

Abbildung 2: Skizze eines Lernschrittes. Der gefüllte Kreis stellt das selektierte Neuron und das Kreuz den Eingabevektor dar. Die untere Kette sind die Ortsvektoren der Neuronen vor dem Lernschritt, die obere Kette die danach. Die Pfeile stellen den Lernschritt dar: Jedes Neuron bewegt sich auf den Eingabevektor zu, die jeweilige Schrittweite ist abhängig vom Abstand zum selektierten Neuron.

verbunden. Diese Darstellungsweise funktioniert nur dann, wenn die Netzdimension gleich der Eingabedimension ist und nur bei ein- bis dreidimensionalen Vektoren, da sich höherdimensionale Strukturen nicht anschaulich darstellen lassen.

Dieses Problem läßt sich durch einen Mechanismus überwinden, den ich *Transformation*[2] nennen möchte. Hierbei wird davon ausgegangen, daß alle Eingabevektoren letztendlich aus der realen (maximal dreidimensionalen) Welt oder zumindest aus einem gedachten vorstellbaren Eingaberaum stammen, der dann ebenfalls maximal dreidimensional ist. Dieser Raum sei $\mathcal{W}$. Eingabereize aus $\mathcal{W}$ werden durch ein Sensorsystem transformiert. Diese Transformation muß nicht dimensionserhaltend sein, sie kann einen hochdimensionalen Vektor von Sensorwerten als Ergebnis haben. Dieser Raum $\mathcal{S}$ von Sensorvektoren wird nun als Eingaberaum für die selbstorganisierten Karten benutzt. Dadurch haben die Gewichtsvektoren der Neuronen dieselbe Dimensionalität wie $\mathcal{S}$ und nicht mehr die Dimensionalität der Neuronenstruktur. Die Umsetzung von $\mathcal{W}$ auf $\mathcal{S}$ geschieht durch die Transformationsfunktion t. Zur Visualisierung der Netzkonfiguration ist es aber nötig, den Neuronen einen Ort in $\mathcal{W}$ zuzuordnen, während ihre Gewichtsvektoren einen Ort in $\mathcal{S}$ repräsentieren. Durch eine Umkehrung[3] der Transformationsfunktion t, der Retransformationsfunktion r, kann jedem Gewichtsvektor aus $\mathcal{S}$ wieder ein eindeutiger Ort in $\mathcal{W}$ zugeordnet werden, der dann visualisiert werden kann. (Wie schon in [19], Kapitel 7.4 bemerkt, könnte t, aber ebensogut r, auch als neuronales Netz statt als mathematische Funktion ausgelegt werden. Hierfür würde sich besonders ein feed-forward–Netz eignen. Aufgrund des allgemeinen Ansatzes der Transformation ist das auch mit dem Simulator möglich.)

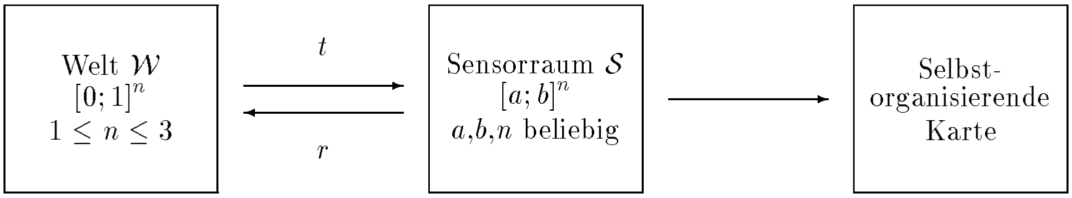

Abbildung 3: Der Transformationsmechanismus

Das biologische Vorbild der selbstorganisierenden Karten sind Bereiche im Gehirn, die eine meist zweidimensionale Abbildung eines Sensorraumes darstellen. So wurde z.B. eine somatotopische Karte entdeckt, die Tastreize der Hautoberfläche abbildet. Dabei werden nicht nur von benachbarten Hautregionen kommende Reize auf benachbarte Neuronen abgebildet, es werden auch „wichtige“ (mit vielen Sensorzellen versehene) Regionen mit einer höheren Auflösung abgebildet, d.h. es werden mehr Neuronen benutzt. Dadurch ergibt sich eine verzerrte Abbildung, in der Bereiche mit häufiger Reizung (Finger, Zunge usw.) überproportional groß repräsentiert werden. Ein weiteres Beispiel sind tonotopische Karten, die akustische Reize abbilden. Hierbei hat man herausgefunden, daß es sich nicht um eine lineare Abbildung der Frequenz des Reizes handelt, sondern um eine annähernd logarithmische Abbildung. Der Grund dafür besteht in der Tatsache, daß zur Erkennung natürlicher

[2] analog zu dem in [1], beschriebenen „relaying network“

[3] keine Umkehrung im strengen mathematischen Sinne, siehe Kapitel 5

Sprache kleine Unterschiede im unteren Frequenzbereich (< 2kHz) wesentlich wichtiger sind als um eine Größenordnung größere Unterschiede im oberen Frequenzbereich (> 4kHz).

Im Gehirn beeinflussen sich benachbarte Neuronen durch gewichtete Verbindungen gegenseitig. Auf neuronale Netze übertragen hieße das, daß alle Neuronen innerhalb der Kohonenschicht voll bidirektional verbunden sein müßten. Diese Verbindungen, die benachbarte Neuronen aktivieren und weiter entfernte hemmen, dienen aber lediglich dazu, einen aktivierten Bereich innerhalb der ansonsten inaktiven Neuronenschicht zu erzeugen. Diese Bereiche sind im Gehirn von unregelmäßiger, in der Regel aber geschlossener Form. Nach Untersuchungen von Teuvo Kohonen (siehe z.B. [4]) reicht es nun, daß ein Bereich beliebiger Form aktiviert wird[4], die konkrete Form dieses Bereiches ist aber unerheblich. Dies macht die Verbindungen zwischen den Neuronen unnötig, sie können durch einen vom Abstand zum maximal aktivierten Neuron abhängigen Distanzfaktor ersetzt werden. Dadurch vereinfacht sich der Algorithmus erheblich, insbesondere entfällt in einer parallelen Implementierung jegliche Kommunikation zwischen den Neuronen bzw. Prozessoren.

Die mathematischen Grundlagen dieses vereinfachten Algorithmus stellen den Rest dieses Kapitels dar. Der allgemeine Algorithmus, der normalerweise nicht zur Implementierung benutzt wird, da er zu aufwendig ist und keine besseren Ergebnisse liefert, ist z.B. in [1] beschrieben. Die konkreten in dieser Arbeit vom Simulationsprogramm benutzten Gleichungen sind im Anhang A angegeben.

Jedes Neuron wird durch zwei Vektoren repräsentiert: statisch durch seine Position $\vec{n}$ in der Neuronenstruktur und dynamisch durch seinen Gewichtsvektor $\vec{w_{\vec{n}}}$. Die Dimensionalität von $\vec{n}$ ist identisch mit der Anzahl der Dimensionen des Netzes. Bei einem zweidimensionalen Netz denotiert $\vec{n} = \begin{bmatrix} x \\ y \end{bmatrix}$ z.B. das Neuron mit diesen x- und y-Koordinaten im Neuronengitter. Die Dimensionalität der Gewichtsvektoren dagegen ist die des Sensorraumes $\mathcal{S}$.

Bei jedem Lernschritt wird der Karte ein Eingabevektor $\vec{\imath}$ aus dem Eingaberaum $\mathcal{W}$ präsentiert. Dann wird das Neuron bestimmt, welches durch den transformierten Eingabevektor $t(\vec{\imath})$ am stärksten aktiviert wird. Dazu wird zunächst für jedes Neuron die Differenz zwischen dessen Gewichtsvektor $\vec{w_{\vec{n}}}$ und $t(\vec{\imath})$ berechnet:

$$\vec{\delta_{\vec{n}}} = t(\vec{\imath}) - \vec{w_{\vec{n}}} \tag{1}$$

Um diese Differenzvektoren vergleichen zu können, werden sie normiert.

$$\mathrm{norm}(\vec{\delta_{\vec{n}}}) = \|\vec{\delta_{\vec{n}}}\| \tag{2}$$

Üblicherweise wird zwar die euklid'sche Norm $\|\vec{x}\|_{\mathrm{euklid}} = \sqrt{\sum x_i^2}$ benutzt, es können aber

[4] Kohonen nennt diesen Bereich *active bubble*

alle Funktionen benutzt werden, die die in den Gleichungen 3 bis 5 beschriebenen Eigenschaften haben.

$$\|\vec{x}\| = 0 \Leftrightarrow \vec{x} = \vec{0} \tag{3}$$

$$\|\mathrm{a} \cdot \vec{x}\| = |\mathrm{a}| \cdot \|\vec{x}\| \qquad \forall a \in \mathrm{R} \tag{4}$$

$$\|\vec{x} + \vec{y}\| \leq \|\vec{x}\| + \|\vec{y}\| \qquad \forall \vec{x}, \vec{y} \tag{5}$$

Das selektierte Neuron $\vec{c}$, wobei $\vec{c}$ seine Position in der Neuronenstruktur ist, ist nun das Neuron, dessen Norm minimal ist:

$$\|\vec{\delta_{\vec{c}}}\| = \min_{\vec{n}} \|\vec{\delta_{\vec{n}}}\| \tag{6}$$

Bei der Benutzung einer selbstorganisierenden Karte endet der Algorithmus hier; die Antwort auf den Eingabevektor $\vec{\imath}$ ist die Position des selektierten Neurons, $\vec{c}$. Dessen Aktivierung wird auf 1 gesetzt, die aller anderen Neuronen auf 0. Zwischenwerte und verteilte Aktivierungsmuster erlaubt der Algorithmus nicht.

Während der Lernphase schließt sich noch die Änderung der Gewichte an. Zunächst wird nach Gleichung 7 der Faktor errechnet, um den jedes Neuron seine Gewichte ändert. Dieser Faktor ϕ hängt ab von der Position des Neurons $\vec{n}$ und der des selektierten Neurons $\vec{c}$. λ ist die Lernrate und α ist eine Funktion, die die Charakteristik der Fernwirkung beschreibt. Sie bestimmt, wie stark Neuronen lernen, die nicht selektiert sind. Hier kommt die Ordnung der Neuronen und deren Nachbarschaftbeziehung zum Tragen: Neuronen in der unmittelbaren Umgebung des selektierten Neurons lernen stärker als weiter entfernte Neuronen. α sollte ein globales Maximum bei $(0;1)$ haben, ist aber ansonsten beliebig. Allerdings sollte eine Anhängigkeit von einem Distanzfaktor vorgesehen sein, der streng monoton sinkt und dabei die Fernwirkung verringert. Der Sinn davon ist, daß zunächst durch eine weitreichende Fernwirkung die globale Ordnung der Karte gelernt wird und dann bei sich verringernder Fernwirkung zunehmend mehr Details gelernt werden.

$$\phi(\vec{n}, \vec{c}) = \lambda \cdot \alpha(\|\vec{c} - \vec{n}\|_{\mathrm{euklid}}) \tag{7}$$

Das eigentliche Lernen ist eine Änderung der als Ortsvektoren interpretierten Gewichte $\vec{w}_{\vec{n}}$ entlang dem Differenzvektor $\vec{\delta_{\vec{n}}}$ und damit in Richtung des transformierten Eingabevektors $t(\vec{\imath})$ (siehe Abbildung 2):

$$\vec{w}_{\vec{n},\mathrm{neu}} = \vec{w}_{\vec{n},\mathrm{alt}} + \phi(\vec{c}, \vec{n}) \cdot \vec{\delta_{\vec{n}}} \tag{8}$$

Damit ist ein Lernschritt, auch Zyklus genannt, beendet und der nächste Eingabevektor kann präsentiert werden.

3 Implementierung des parallelen Simulators

Der implementierte Simulator für selbstorganisierende Karten ist unterteilt in zwei Programme: die X-Oberfläche und den Kern. Zunächst zur Begründung dieser Zweiteilung. Die Implementierung sollte auf der MasPar erfolgen, daher muß von vornherein ein Prozeß auf deren DPU (**D**ata **P**arallel **U**nit) laufen. Diese ist aber nicht in der Lage, selbst eine X-Windows–Oberfläche zu erzeugen, da sie (wie bei Großcomputern üblich) keinen direkten Zugang zu Ein-/Ausgabegeräten hat. Um diesen Zugang zu schaffen, gibt es zwei Möglichkeiten. Erstens kann eine heterogene Programmstruktur benutzt werden, d.h. ein Programm, das teilweise auf der DPU und teilweise auf dem Frontend läuft. Diese Architektur kollidiert mit der grundlegenden Eigenschaft von X-Windows[5], daß Events nur dann behandelt werden, wenn das Programm nicht wartet. Da beim Training des Netzes aber ständig auf das Ende der angeforderten Anzahl von Lernzyklen und auf das Ende der Datenübertragung gewartet werden muß, hätte das zur Folge, daß die Oberfläche für diese Zeit nicht reagiert und den Benutzer von jeder Steuerungsmöglichkeit ausschließt. Auch ein asynchroner Aufruf der Kernroutinen, der von dieser heterogenen Programmarchitektur unterstützt wird, hätte an dem Problem nichts geändert. Zusammen mit der Langsamkeit des Frontends, einer VAX-Station 3520, war dies Grund genug, diese Alternative zu verwerfen. Dieses Argument ist jetzt nach dem kurz vor Abschluß der Arbeit erfolgten Austausch der VAX gegen eine DEC-Station 5000/200 hinfällig (siehe Kapitel 9).

Die zweite Möglichkeit, die auch die implementierte ist, besteht darin, zwei Programme zu benutzen, von denen eines, der Kern, auf der DPU läuft, während das andere, die X-Oberfläche, auf einer (beliebigen) anderen CPU laufen kann. Dadurch erhält man zwei unabhängige miteinander kommunizierende parallele Prozesse. Diese Architektur kommt der Verwendung von X-Windows sehr entgegen und eröffnet zusätzlich eine weitere Möglichkeit zur Parallelität. Bei der Kommunikation zwischen beiden Programmen fiel die Wahl auf eine einfache und sehr robuste Methode: der Kommunikation über Dateien[6]. Jedes Teilprogramm schreibt seine Nachrichten in eine Datei, die dann vom anderen Teilprogramm gelesen wird. Zur Synchronisierung der Prozesse wird eine Datei der Länge Null benutzt, die wie ein Semaphor wirkt. Sobald ein Teilprozeß seine Nachrichten vollständig in eine Datei geschrieben hat, kreiert er die Semaphordatei. Der andere Teilprozeß wartet auf die Existenz dieser Datei und liest und verarbeitet dann die Nachricht.

3.1 Implementierung der X-Oberfläche

Das Programm, das u.a. die X-Oberfläche erzeugt, ist in C unter Verwendung von X-Windows und dem Athena widget set implementiert. Es dient zur Visualisierung der aktuellen Netzkonfiguration und stellt alle Schnittstellen zum Benutzer und zu anderen Pro-

[5]Die Benutzung von X-Windows bestimmt nicht nur wesentlich die Programmstruktur, sondern auch andere Programmparameter.

[6]Diese Methode hat sich bei den leider häufigen Ausfallzeiten der MasPar sehr bewährt, da sie, anders als z.B. TCPIP, nicht auf das Funktionieren beider Teilprogramme angewiesen ist, sondern ein Teilprogramm sozusagen „von Hand“ (durch Bereitstellen der geeigneten Dateien) simuliert werden kann.

grammen zur Verfügung. Im einzelnen sind folgende Schnittstellen implementiert:

- graphische Ausgabe der aktuellen Netzkonfiguration
- Ausgabe der aktuellen Lernrate
- Ausgabe des aktuellen Distanzfaktors
- Ausgabe der für die letzten Lernschritte gebrauchten Zeit
- Ausgabe der gesamten bisher für das Lernen gebrauchten Zeit
- Ausgabe der gemittelten CUPS (**C**onnection **U**pdates **P**er **S**econd, ein bei neuronalen Netzen weitverbreitetes Geschwindigkeitsmaß)
- Ausgabe der aktuellen Netzkonfiguration in eine Postscriptdatei. Das erzeugte Bild ist identisch mit der graphischen Ausgabe auf dem Bildschirm, es wird zusätzlich der Name der Datei und das Erstellungsdatum gespeichert. Wie jede Postscriptdatei kann auch diese mit Ghostscript am Bildschirm oder auf einem Postscript-fähigen Drucker ausgedruckt werden.
- Ausgabe der aktuellen Netzkonfiguration als reine Daten, die in einfacher Weise in einem Anwendungsprogramm benutzt werden können. Somit können selbstorganisierende Karten nicht nur trainiert, sondern auch in realen Anwendungen eingesetzt werden.
- Lesen und Schreiben einer Netzkonfigurationsdatei, die zu SNNS kompatibel ist (näheres hierzu siehe Anhang D.1)
- sowie eine mausgestützte Steuerung der Simulation durch den Benutzer

Näheres zur X-Oberfläche siehe Kapitel 4.

3.2 Implementierung des parallelen Simulatorkerns

Die MasPar ist eine single-intruction-multiple-data-stream–Maschine mit 2k, 4k, 8k oder 16k Prozessorelementen (PEs). Diese sind gleichzeitig in zwei verschiedenen Arten angeordnet. Eine mögliche Sichtweise ist eine Kette, in der alle PEs linear angeordnet sind. Die zweite Sichtweise ist die eines zweidimensionalen Feldes von PEs, im Falle der für diese Arbeit benutzten MasPar 1216A z.B. Quadrat mit einer Kantenlänge von 128 PEs. Der Kern des Simulators ist allerdings so implementiert, daß er sich an die Anzahl der verfügbaren PEs automatisch anpaßt. Auf die verschiedenen Arten der Kommunikation zwischen den PEs wird hier nicht eingegangen, da der implementierte Algorithmus ohne explizite Kommunikation auskommt[7]. Zu diesem Thema sei auf [29] und [30] verwiesen.

[7] Gleichwohl wird die Kommunikation intern von den `reduce()`-Funktionen benutzt (s.u.), dies ist für den Benutzer aber transparent.

Ein wesentlicher Aspekt beim Entwurf eines Programms für einen Parallelrechner sind die Datenstrukturen, da sie darüber entscheiden, wie gut und inwieweit sich der Algorithmus parallelisieren läßt. Bei selbstorganisierenden Karten ist es naheliegend und sinnvoll, ein Neuron mit einem Prozessorelement zu identifizieren. Dies funktioniert natürlich nur, solange die Anzahl der Neuronen nicht größer als die der Prozessorelemente ist. Bei mehr Neuronen muß auf virtuelle Prozessoren ausgewichen werden, d.h. auf einem physischen Prozessor werden die Datenstrukturen für mehr als einen virtuellen Prozessor allokiert. Im Gegensatz zu manchen Parallelrechnern, die von sich aus virtuelle Prozessoren unterstützen (wie z.B. die Connection Machine), kennt weder die Hardware der MasPar noch MPL das Konzept der virtuellen Prozessoren. Dies hat zwar den Nachteil, daß man etwas mehr Arbeit bei der Implementierung hat, bietet aber auch eine Chance: Man kann die virtuellen Prozessoren in der für das Problem geeignetsten Art und Weise auf die physischen Prozessoren verteilen[8].

Die Verteilung der Neuronen wurde nun je nach Dimensionalität der selbstorganisierenden Karte geeignet gewählt. Bei eindimensionalen Karten wird auf die lineare Anordnung der PEs zurückgegriffen, damit sind benachbarte Neuronen auch auf benachbarten PEs lokalisiert. Im dreidimensionalen Fall wird die Sichtweise des Prozessorarrays als zweidimensionales Feld benutzt. Die x- und y-Richtungen werden auf das Prozessorarray abgebildet, während die z-Richtung auf Schichten von virtuellen Prozessorelementen abgebildet wird. Dadurch ergibt sich die Beschränkung, daß das Netz in x- bzw. y-Richtung höchstens so viele Neuronen enthalten darf wie physische Prozessoren vorhanden sind. Wenn die Fernwirkung auf einen endlichen Teilbereich des Netzes beschränkt ist (mittels des Parameters `maxDist`, siehe Kapitel 4.2), ergibt sich so eine einfache Möglichkeit der Optimierung, indem zum Lernen nur die betroffenen Schichten des Netzes herangezogen werden müssen.

Um bei den (am häufigsten benutzten) zweidimensionalen Karten so wenige Beschränkungen wie möglich zu erhalten, wurde ein etwas komplizierterer Algorithmus zur Abbildung der Neuronen auf die PEs gewählt (siehe Abbildung 4): Gegeben sei eine selbstorganisierende Karte (A) einer beabsichtigten Größe und ein zur Verfügung stehendes Prozessorarray (B). Zunächst wird festgestellt, wie oft das Prozessorarray in die Karte ganz hineinpaßt (C). (Paßt umgekehrt die Karte ganz in das Prozessorarray, so ist die Abbildung trivial.) Dann wird die Karte unterteilt in Bereiche in der Größe des Prozessorarrays sowie einen rechten und einen unteren Rest (D). Die ersteren werden zu Schichten von virtuellen Prozessoren (E), mit den zwei Resten werden möglichst wenige weitere Schichten gefüllt. So ergibt sich die bestmögliche Ausnutzung der MasPar. Durch diesen Algorithmus kann das Netz in beiden Dimensionen eine beliebige Größe haben[9], die Anzahl der möglichen Neuronen wird nicht durch die Anzahl der physischen PEs beeinflußt. Nur der auf der DPU verfügbare Speicher beschränkt die maximale Größe eines Netzes.

[8] Zu dieser Diskussion siehe [27].

[9] anders als z.B. in [9], wo vorausgesetzt wird, dass die Zahl der Neuronen in jeder Dimension ein ganzzahliges Vielfaches der Anzahl der PEs ist

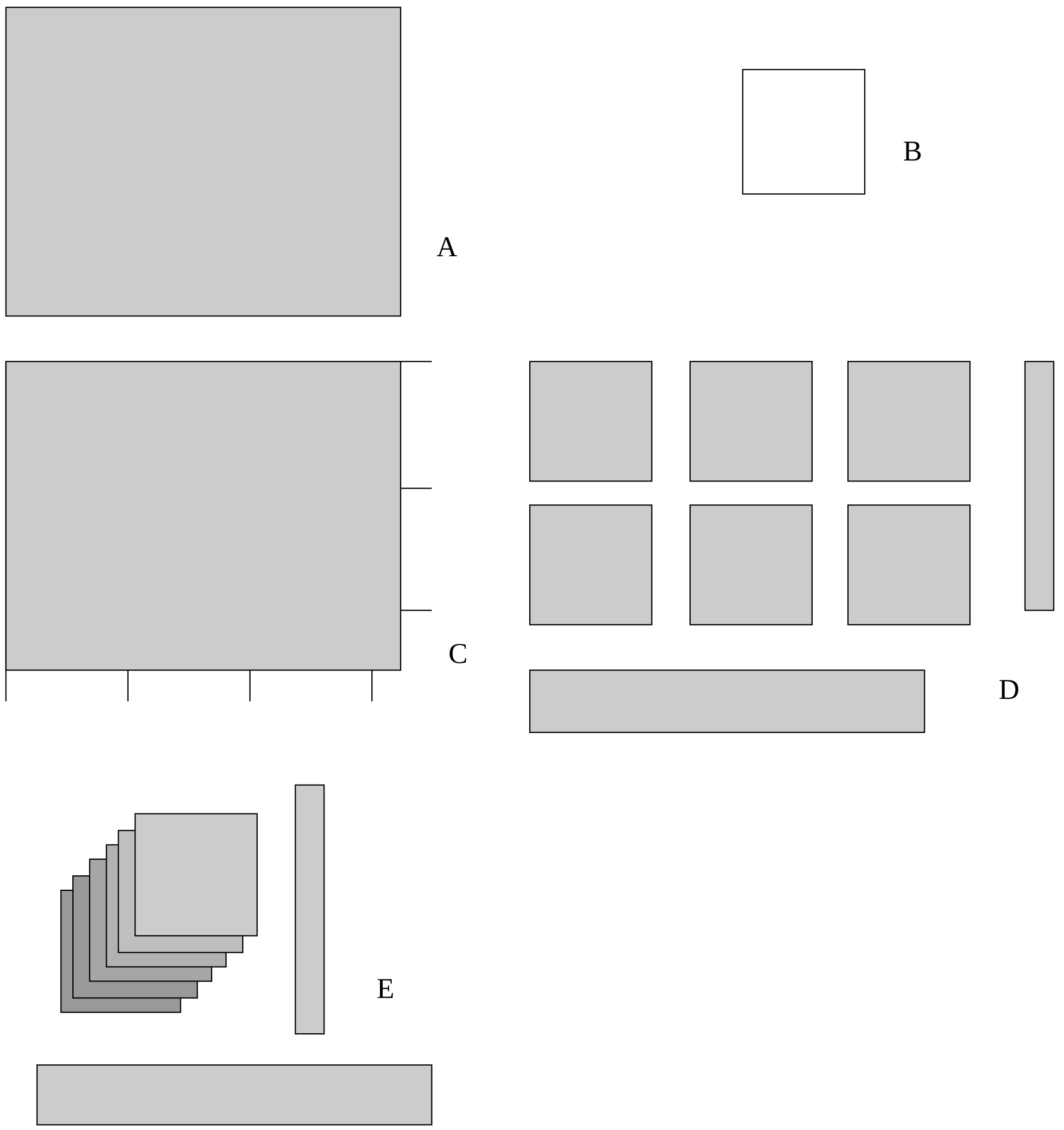

Abbildung 4: Aufteilung einer zweidimensionalen Netzstruktur

Die zentrale Funktion des Kerns ist `DoNSteps()`. Sie führt n Lernschritte durch, wobei n die vom Benutzer spezifizierte und in `StepsToDo` gespeicherte Anzahl von Zyklen ist. Auf diese Funktion soll hier im Detail eingegangen werden. (Stellt man die im Folgenden wiedergegebenen Codestücke zusammen, erhält man exakt den Code der Funktion `DoNSteps`.)

Zunächst wird eine lokale Variable auf die Anzahl der angeforderten Lernschritte gesetzt. Durch `dpuTimerStart()` wird der Timer der MasPar, der zum Bestimmen der verbrauchten Zeit benutzt wird, auf Null gesetzt. Dann wird die Hauptschleife so oft durchgeführt, wie Lernschritte angefordert wurden.

```
n = StepsToDo;
dpuTimerStart();
while (n-- > 0L) {
```

Die erste Aktion in der Schleife ist die Vorbelegung eines Eingabevektors der richtigen Dimensionalität mit gleichverteilten Zufallszahlen und dessen sofortige Transformation. Tritt in der Transformationsfunktion ein Fehler auf, so wird das Lernen sofort beendet und der Kern gibt eine benutzerdefinierte Fehlermeldung aus.

```
  if (!GetRandomVector(&input)) return False;
```

Dann wird für jedes Neuron die Differenz seiner Gewichte zum Eingabevektor und deren Norm berechnet. Diese Schleife wird so selten wie möglich durchlaufen (s.o.), da Neuronen so gut wie möglich auf physische PEs verteilt werden, um die minimale Anzahl von Schichten virtueller Prozessorelemente (`vpe`'s) zu erhalten. Die Anzahl solcher Schichten ist in `nvprocs` gespeichert. Da diese Schichten nicht vollständig durch virtuelle Prozessoren belegt sein müssen, wird das Feld `.pos.dim` benutzt, um physische Prozessoren dann von der Berechnung auszuschließen, wenn sie in der jeweiligen Schicht keinen aktiven virtuellen Prozessor enthalten. Man beachte, daß diese Schleife von der Netzdimension völlig unabhängig ist. Tatsächlich trifft das für fast alle Funktionen des Kerns zu. Um eine Fallunterscheidung zwischen verschiedenen Normierungsfunktionen zu vermeiden, wird hier mit einem indirekten Funktionsaufruf über Zeiger gearbeitet.

```
  for (i = 0; i < nvprocs; i++)
  {
    if (vpe[i].pos.dim)
    {
      for (j = 0; j < myInputDimension; j++)
      {
        vpe[i].difference[j] = input.x[j] - vpe[i].weight[j];
      }
      (*norm)(&(vpe[i]));
    }
  }
```

Der nächste Schritt ist die Suche nach dem Neuron, dessen Norm minimal ist. Dieses Neuron ist dann selektiert; es ist das am stärksten aktivierte. Dazu werden **minNorm** und **nvpeMinNorm**, die während der Suche die minimale Norm aller virtuellen Prozessoren eines physischen Prozessors und den Index (Schichtnummer) dieses virtuellen Prozessors enthalten, benutzt. Dann wird eine `reduce`-Funktion aufgerufen, die in logarithmischer Zeit das globale Minimum **isMin** aller Minima aller aktiven Prozessorelemente bestimmt. Danach kennt jeder Prozessor die minimale Norm, es wird aber die *Position* im Netz von dem Neuron mit der minimalen Norm gesucht. Eine direkte Zuweisung der Art

```
if (isMin == minNorm) selected.pos = vpe[iproc].pos   /* illegal */
```

ist nicht möglich da dies zum einen eine Zuweisung von pluralen auf singuläre Daten wäre, was in MPL nicht erlaubt ist, und zum anderen `iproc`, die interne Nummer des Prozessorelements, nicht unbedingt eindeutig ist, da das Minimum nicht eindeutig sein muß. Darum wird ein neues `reduce` nötig, um zunächst die `iproc`-Nummer des selektierten Neurons herauszufinden. Dann erst kann über `proc`-Aufrufe auf dessen Positionskoordinaten zugegriffen werden. Näheres zu `reduce`, `iproc` und `proc` siehe [29] und [30].

```
minNorm = (plural float)1e10;
nvpeMinNorm = (plural int)0;
for (i = 0; i < nvprocs; i++)
{
  if ((minNorm > vpe[i].normedDifference) && vpe[i].pos.dim)
  {
    minNorm = vpe[i].normedDifference;
    nvpeMinNorm = i;
  }
}
isMin = reduceMind(minNorm);
if (((plural float)isMin == minNorm) && vpe[nvpeMinNorm].pos.dim)
{
  npe = reduceMin32(iproc);
}
selected.dim = proc[npe].vpe[nvpeMinNorm].pos.dim;
for (i = 0; i < NetDimension; i++)
{
  selected.x[i] = proc[npe].vpe[nvpeMinNorm].pos.x[i];
}
```

Als nächstes schließt sich der eigentliche Lernteil an. Wieder läuft eine Schleife über alle Schichten von virtuellen Prozessoren. Darin wird zunächst der euklid'sche Abstand eines Neurons vom selektierten Neuron berechnet. (Genauer gesagt wird das Quadrat des Abstandes berechnet, da dadurch das teure Ziehen der Wurzel eingespart wird.) Dann wird darauf die Funktion α angewandt, die die Art der Fernwirkung bestimmt. Auch hier wird zum Vermeiden einer Fallunterscheidung zwischen den verschiedenen zur Auswahl stehenden Fernwirkungsfunktionen mit einem indirekten Funktionsaufruf gearbeitet. Schließlich wird der resultierende Wert noch mit der Lernrate λ multipliziert. Das Ergebnis ist der Faktor, um den die Gewichte eines Neurons beim aktuellen Lernschritt auf den aktuellen Eingabevektor zubewegt werden. (Je nach α kann dieser Faktor auch negativ sein und sich das Neuron damit vom Eingabevektor wegbewegen.)

```
    for (i = 0; i < nvprocs; i++)
    {
      if (vpe[i].pos.dim)
      {
        currentFactor = vpe[i].pos.x[0] - selected.x[0];
        currentFactor = currentFactor * currentFactor;
        for (j = 1; j < NetDimension; j++)
        {
          currentFactor += (vpe[i].pos.x[j] - selected.x[j])
                          * (vpe[i].pos.x[j] - selected.x[j]);
        }
        if ((currentFactor < (MaxDist * MaxDist)) || (MaxDist == 0.0))
        {
          currentFactor = (*Alpha)(currentFactor) * LRate;
          for (j = 0; j < myInputDimension; j++)
          {
            vpe[i].weight[j] += currentFactor * vpe[i].difference[j];
          }
        }
      }
    }
```

Den Abschluß der Funktion `DoNSteps` bilden einige Aufräumungsarbeiten: Noch in der Schleife werden die Lernrate und der Distanzfaktor wie vom Benutzer spezifiziert dekrementiert. Nach Ende der `while`-Schleife wird die verbrauchte Zeit gestoppt und die Anzahl der gerade gemachten Schritte der Gesamtzahl hinzuaddiert.

```
    (*DecrementLRate)();
    (*DecrementDistFactor)();
  } /*while*/
  NStepsTime = dpuTimerElapsed();
  NumberOfSteps += StepsToDo;
  return True;
```

4 Benutzung des Simulators

4.1 Ein Beispiellauf

Zur Veranschaulichung der Benutzung des Simulators sei hier die Behandlung eines Problems ausführlich beschrieben. Als Problem wird das einfachste und zugleich anschaulichste gewählt, nämlich das Einpassen eines Netzes mit 10 mal 10 Neuronen in das Einheitsquadrat, in dem die Reize (Eingabevektoren) gleichverteilt sind. Dazu reicht als Transformation die Identität aus (siehe Kapitel 5).

Der erste Schritt ist der Start des Simulators. Hierzu muß zuerst der Kern auf der MasPar gestartet werden. Dieser gibt als erstes die Meldung `I got the DPU` aus. Jeder Prozeß, der auf der MasPar ausgeführt wird, kommt zuerst in eine Warteschlange, bis auf der DPU genügend Platz für ihn zur Verfügung steht. Je nach schon laufenden Prozessen kann das beliebig lange dauern, theoretisch kann ein einziger Prozeß den ganzen Speicher für sich beanspruchen und so alle anderen Prozesse so lange blockieren bis er terminiert. Der Benutzer muß allerdings nicht auf diese Meldung warten, er kann sofort die X-Oberfläche aufrufen. Diese füllt einen DEC-Bildschirm ganz aus. Es gibt folgende Bereiche:

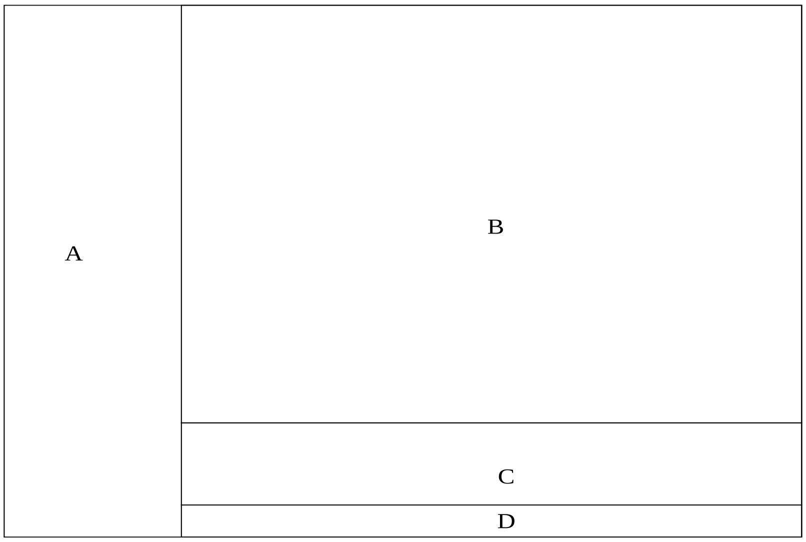

Abbildung 5: Bildschirmaufteilung der X-Oberfläche

A: Der Menübereich. Hier erscheinen alle Menüs und werden alle Parameter eingegeben. Beim Programmstart gibt es nur den Button `Exit`, der es dem Benutzer erlaubt, den Simulator sofort wieder zu verlassen.

B: Der Visualisierungsbereich. Hier werden die aktuellen Netzkonfigurationen angezeigt. Beim Programmstart ist noch kein Netz definiert, darum ist dieser Bereich noch leer.

C: Der Statusbereich. Hier werden später angezeigt:

- die Anzahl der ausgeführten Lernzyklen
- die aktuelle Lernrate
- der aktuelle Distanzfaktor
- die für die letzten n Lernschritte verbrauchte Zeit
- die insgesamt verbrauchte Zeit
- die sich daraus ergebenden connection updates per second (CUPS)
- die hochgerechneten MCUPS, wenn das Prozessorarray der MasPar vollständig genutzt wäre

D: Der Meldungsbereich, in dem Fehlermeldungen und andere Meldungen, die für den Benutzer hilfreich sein können, ausgegeben werden. Es empfiehlt sich, dort ab und zu nachzuschauen. Jetzt nach dem Programmstart steht dort z.B. die Meldung `Waiting for MasPar...`, die darauf hinweist, daß die X-Oberfläche noch versucht, Kontakt zu einem auf der MasPar laufenden Kern zu bekommen.

Sobald der Kontakt zum Kern hergestellt ist, erscheint das STARTUP MENU. Hier werden nur die benutzten Menüeinträge angesprochen, eine genaue Beschreibung aller Menüeinträge erfolgt in Kapitel 4.2.

Um eine Netzbeschreibung zu erzeugen, wird der entsprechende Menüpunkt „Create Network“ angeklickt. Dadurch wechselt der Benutzer ins CREATE MENU. Hier wie in allen Menüs gibt es zwei Typen von Einträgen, nämlich Buttons und Textfelder. Buttons wechseln beim Anklicken ihren Wert, während in Textfelder ein Wert eingetragen werden kann, wenn man den Mauszeiger darauf positioniert. Um mehrere Textfelder zu ändern, kann man entweder jedesmal die Maus auf das nächste bewegen oder man drückt einfach die Eingabetaste und kommt damit in das nächste Textfeld ohne die Maus benutzen zu müssen.

Die Vorgaben im CREATE MENU können vorläufig ohne Änderung übernommen werden, da die Vorgaben für Nx, Ny und Nz (10, 10 und 0) schon die beabsichtigte Netzgröße spezifizieren. Außer „RandomStartConfig“, das die Anfangsbelegung der Gewichte der Neuronen steuert, können alle anderen Parameter auch noch später während des Trainings geändert werden.

Nach dem Bestätigen mit „Ok“ wird das MAIN MENU sichtbar. Dieses ist, wie der Name schon sagt, das zentrale Menü des Simulators. Im Menüfeld „#Steps“ kann der Benutzer eine Zahl eintragen, die für zwei Zwecke benutzt wird. Wenn der Benutzer mit dem Button „Run #Steps“ n Lernschritte anfordert, ist n die dort eingegebene Anzahl. Wenn er „Run“ anklickt, schaltet der Simulator in den kontinuierlichen Lernmodus. Hier wird ununterbrochen[10] gelernt und die aktuelle Netzkonfiguration wird jeweils nach so vielen Schritten neu angezeigt, wie in „#Steps“ angegeben ist.

Versucht man jetzt, das Netz zu trainieren, so wird man feststellen, daß es sich schlecht entfaltet. Der Grund ist, daß der Distanzfaktor, der für die Fernwirkung zuständig ist, mit einem Wert von unter eins zu klein ist. Klickt man „Change Parameters“ an, so bekommt

[10]nur durch die Kommunikation unterbrochen

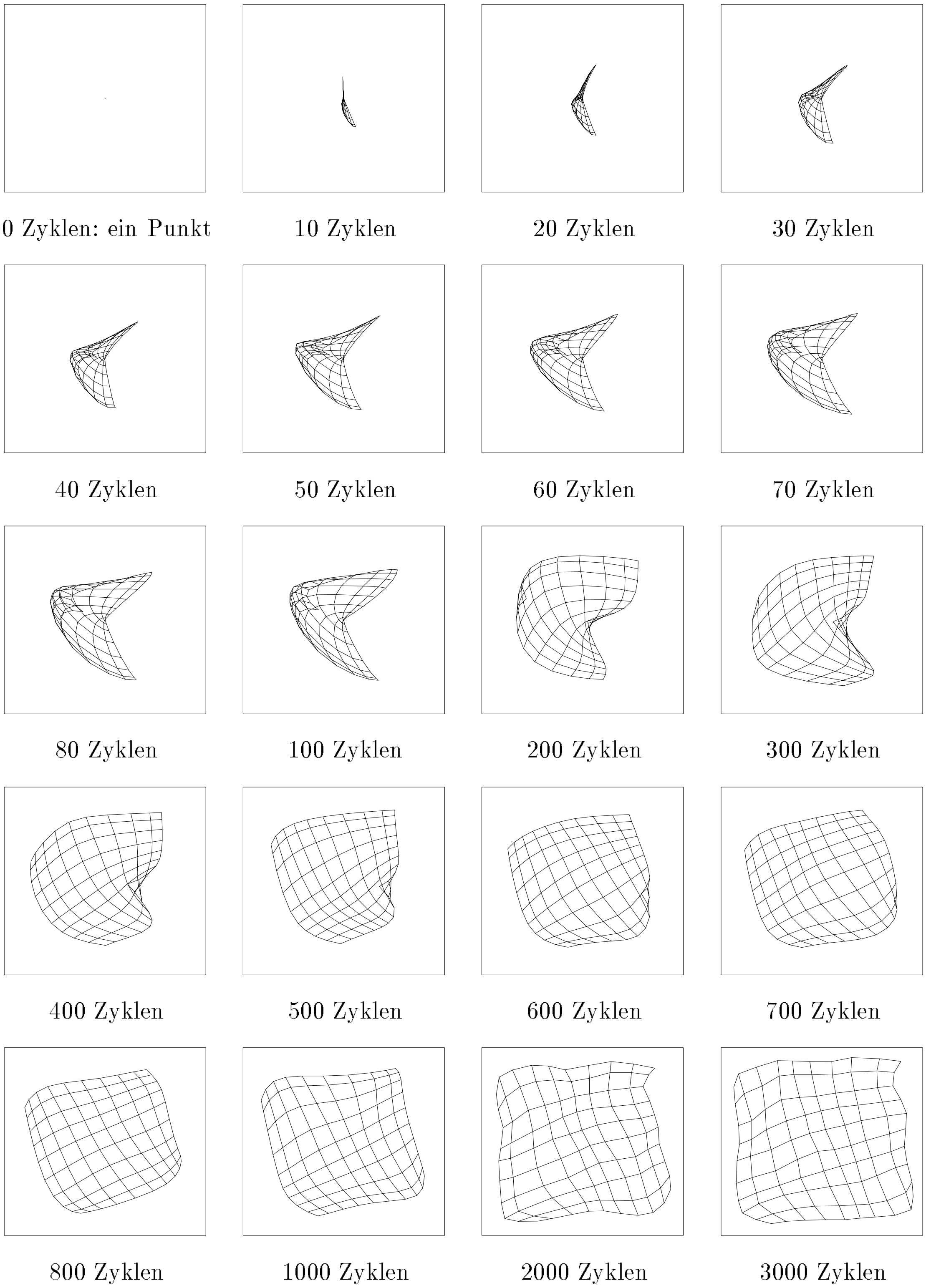

Abbildung 6: Beispiel der Entfaltung einer zweidimensionalen Karte

man ein Menü, in dem man alle während einer laufenden Simulation änderbaren Parameter ändern kann. Eine gute Wahl wäre z.B. eine konstante Lernrate von 0.1 (LRateStart = LRateEnd = 0.1) und ein linear von 10 auf 1 sinkender Distanzfaktor mit einer Halbwertzeit von 5000 Zyklen, d.h. nach 5000 Schritten ist der Distanzfaktor auf 10/2 = 5 gesunken (DistFactorStart = 10, DistFactorEnd = 1, DistFactorHL = 5000, DecDistFactor = linear). Das Ergebnis wäre ähnlich wie in Abbildung 4.1, nur wurde dort als Startkonfiguration `dot` benutzt, d.h. alle Gewichte wurden auf 0.5 initialisiert.

Wenn man jetzt versucht, eine erreichte Netzkonfiguration oder die Netzbeschreibung zu speichern, erhält man eine Fehlermeldung, die den Benutzer darauf hinweist, daß er noch keinen Filenamen angegeben hat. Dieser wird ohne Extension eingegeben, da der Simulator selbständig die richtige Namenserweiterung vornimmt.

Mit „Restart“ kommt man zurück ins STARTUP MENU und kann eine neue Netzbeschreibung eingeben oder von Platte laden. „Exit“ beendet den Simulator, d.h. sowohl die X-Oberfläche als auch der Kern wird beendet. Die X-Oberfläche verschwindet dann vom Bildschirm, der Kern gibt zur Bestätigung zusätzlich eine Ende-Meldung aus.

4.2 Die Menüs

Dieses Kapitel erklärt jeden Menüpunkt, den es im Simulator gibt, seinen Sinn und seine Wirkung.

Das STARTUP MENU ist das erste Menü, das ein Benutzer sieht. Von hier werden Netzbeschreibungen geladen oder man wechselt ins CREATE MENU, wo alle Parameter einer selbstorganisierenden Karte bestimmt werden können. Das MAIN MENU bietet die Möglichkeit, die Netzkonfiguration in verschiedenen Formaten zu speichern und das Training zu steuern, das RUN MENU dient lediglich dazu, ein kontinuierlich lernendes Netz zu unterbrechen. Im CHANGE MENU können alle Parameter, die nur das Training und nicht das Netz als solches betreffen, während des Trainings geändert werden und im SAVE MENU wird das Format der zu speichernden Daten gewählt. Abbildung 7 zeigt alle möglichen Übergänge zwischen den Menüs.

4.2.1 STARTUP MENU

Load Network
Eine Netzbeschreibung aus einer bestehenden Datei lesen. Das Format der Netzbeschreibungsdatei entspricht dem von SNNS Version 1.4–3D, erweitert um Kohonen-spezifische Teile. Für eine genaue Beschreibung des Formats siehe Appendix D. Die Datei enthält schon alle nötigen 3D-Informationen, um das Netz durch die Erweiterungen von Ralf Hübner mit `xgui` dreidimensional darzustellen. Dieses reagiert im Moment noch mit einem Fehler auf die Kohonen-spezifischen Erweiterungen. Um ein mit `xgui` verträgliches File zu erhalten,

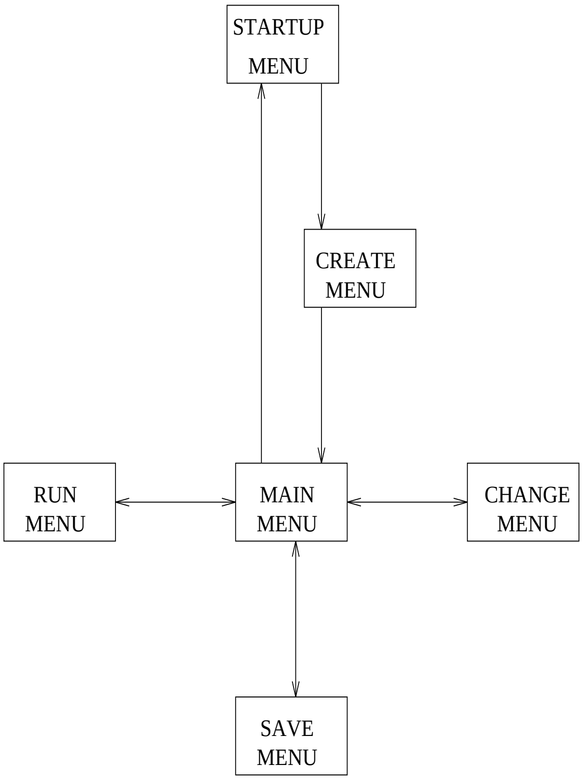

Abbildung 7: Diese Übergänge gibt es zwischen den Menüs. Diese Abbildung schließt auch Übergänge ein, die es nur im Fehlerfall gibt. Welcher Übergang wodurch ausgelöst wird, ist Teil der folgenden Beschreibung.

müssen die Angabe `kohonen` als Lernregel und die `kohonen section` entfernt werden. `xgui` kann das Netz dann zwar in zwei- und dreidimensionaler Darstellung anzeigen, trainiert werden können selbstorganisierende Karten aber damit natürlich (noch) nicht.

FILE:
Hier kann der momentan benutzte Filename eingetragen werden. Die X-Oberfläche hängt selbständig alle benötigten Extensionen an. Das spart zum einen ein paar Tastendrücke, zum anderen wird so sichergestellt, daß jedes File eine seinem Typ entsprechenden Extension bekommt. Gibt der Benutzer z.B. `foo` an, so wird automatisch an Netzbeschreibungsdateien `.net`, an Datenfiles `.wts` und an Bilder `.ps` angehängt. Der Filename kann auch in der Kommandozeile beim Aufruf der X-Oberfläche angegeben werden.

Create Network
wechselt in das CREATE MENU, wo alle Parameter für ein neues Netz eingegeben werden können.

Exit
verläßt die X-Oberfläche und verursacht auch ein Terminieren des Kerns. Dadurch ist die MasPar wieder frei für andere Prozesse. Soll der Simulator später wieder gestartet werden,

muß der Benutzer auch den Kern wieder neu starten.

4.2.2 CREATE MENU

Nx
Anzahl der Neuronen in Richtung der ersten Dimension des Netzes. Bei eindimensionalen Netzen wird hier die Anzahl der Neuronen in der Kette eingetragen.

Ny
Anzahl der Neuronen in Richtung der zweiten Dimension des Netzes. Ist ein eindimensionales Netz beabsichtigt, dann hier 0 oder nichts eintragen.

Nz
Anzahl der Neuronen in Richtung der dritten Dimension des Netzes. Ist ein zweidimensionales Netz beabsichtigt, dann hier 0 oder nichts eintragen.

Dimension
Wird die Dimensionszahl angeklickt, werden die Werte für Nx, Ny und Nz getestet und eventuell korrigiert (z.B. sind nur ganze positive Zahlen erlaubt). Dann wird die aktuelle Anzahl der Dimensionen angezeigt.

Norm (2=Euklid)
Hier kann der Benutzer die gewünschte Norm eingeben, die für das Unterschiedsmaß zwischen einem Eingabevektor und den Gewichten eines Neurons benutzt wird. Dabei bedeutet 2 die euklid'sche Norm, die üblicherweise benutzt wird (euklid'sche Norm von x: $\|\vec{x}\|_{\mathrm{euklid}} = \sqrt{\sum x_i^2}$). 0 steht für die Maximumnorm $\|\vec{x}\|_{\max} = \max|x_i|$. Andere Zahlen werden als Exponenten einer allgemeinen Norm benutzt: $\|\vec{x}\|_n = \sqrt[n]{\sum |x_i|^n}$

LRateStart
Der Anfangswert der Lernrate.

LRateEnd
Die Lernrate sinkt bis auf diesen Wert und bleibt dann konstant. Wird LRateEnd auf LRateStart gesetzt, so bleibt die Lernrate konstant.

LRateHL
Die Halbwertzeit (engl. **h**alf-**l**ife) der Lernrate. Diese sinkt in so vielen Schritten wie hier angegeben auf die Hälfte (nicht aber unter LRateEnd). Diese Spezifikation wurde gewählt, da so ein einheitlicher Parameter benutzt werden kann ohne den Unterschied zwischen linearem und exponentiellem Absinken berücksichtigen zu müssen. Sinkt die Lernrate linear, so ist sie nach LRateHL Schritten auf die Hälfte und nach $2 \cdot$ LRateHL auf 0 gesunken. Sinkt sie aber exponentiell, so ist zwar nach LRateHL Schritten ebenfalls die halbe Lernrate erreicht, nach $2 \cdot$ LRateHL aber erst ein Viertel und nach $3 \cdot$ LRateHL ein Achtel.

DecLRate
Hier wird eingestellt (durch Anklicken) ob die Lernrate linear oder exponentiell sinkt.

Alpha
bestimmt, nach welcher Funktion die Fernwirkung einer Gewichtsänderung mit der Entfernung vom aktivierten Neuron abnimmt. Hier werden acht verschiedene Funktionen zur Auswahl angeboten: **gauss** (die Gaußglocke), **gauss2**, **gauss3** (eine mexican-hat–Funktion), **linear**, **linear2**, **linear3**, **next4** und **next8**. Die Funktionen sind in Abbildung 4.2.2 dargestellt, dabei bedeutet l die Lernrate und d den Distanzfaktor. Die zugehörigen Gleichungen sind die Folgenden:

next4: $\alpha_{\mathrm{next4}}(x) = \begin{cases} 1 & \text{wenn } x^2 < 0.1 \\ 1/10 & \text{wenn } x^2 < 1.8 \\ 0 & \text{sonst} \end{cases}$

next8: $\alpha_{\mathrm{next8}}(x) = \begin{cases} 1 & \text{wenn } x^2 < 0.1 \\ 1/10 & \text{wenn } x^2 < 2.2 \\ 0 & \text{sonst} \end{cases}$

gauss: $\alpha_{\mathrm{gauss}}(x) = \mathrm{e}^{-x^2/\mathrm{DistFactor}}$

gauss2: $\alpha_{\mathrm{gauss2}}(x) = (1 - (\frac{x}{\mathrm{DistFactor}})^2) \cdot \mathrm{e}^{-(x/\mathrm{DistFactor})^2}$

gauss3: $\alpha_{\mathrm{gauss3}}(x) = (1 - 3(\frac{x}{\mathrm{DistFactor}})^2 + (\frac{x}{\mathrm{DistFactor}})^4) \cdot \mathrm{e}^{-(x/\mathrm{DistFactor})^2}$

linear: $\alpha_{\mathrm{linear}}(x) = \begin{cases} 1 & \text{wenn } x < \mathrm{DistFactor} \\ 0 & \text{sonst} \end{cases}$

linear2: $\alpha_{\mathrm{linear2}}(x) = \begin{cases} 1 & \text{wenn } x < \mathrm{DistFactor} \\ -1/3 & \text{wenn } x < 3 \cdot \mathrm{DistFactor} \\ 0 & \text{sonst} \end{cases}$

linear3: $\alpha_{\mathrm{linear3}}(x) = \begin{cases} 1 & \text{wenn } x < \mathrm{DistFactor} \\ -1/10 & \text{wenn } x < 3 \cdot \mathrm{DistFactor} \\ 1/100 & \text{wenn } x < 10 \cdot \mathrm{DistFactor} \\ 0 & \text{sonst} \end{cases}$

DistFactorStart
Der Anfangswert für den bei der Fernwirkung benutzten Faktor. Bei den Alpha-Werten next4 und next8 wird dieser Faktor nicht benutzt. Sonst ist die konkrete Bedeutung dieses Faktors abhängig von `Alpha` (s.o.).

DistFactorEnd
Der Endwert des Faktors für Fernwirkung. Ist DistFactorEnd gleich DistFactorStart, so bleibt der Faktor konstant.

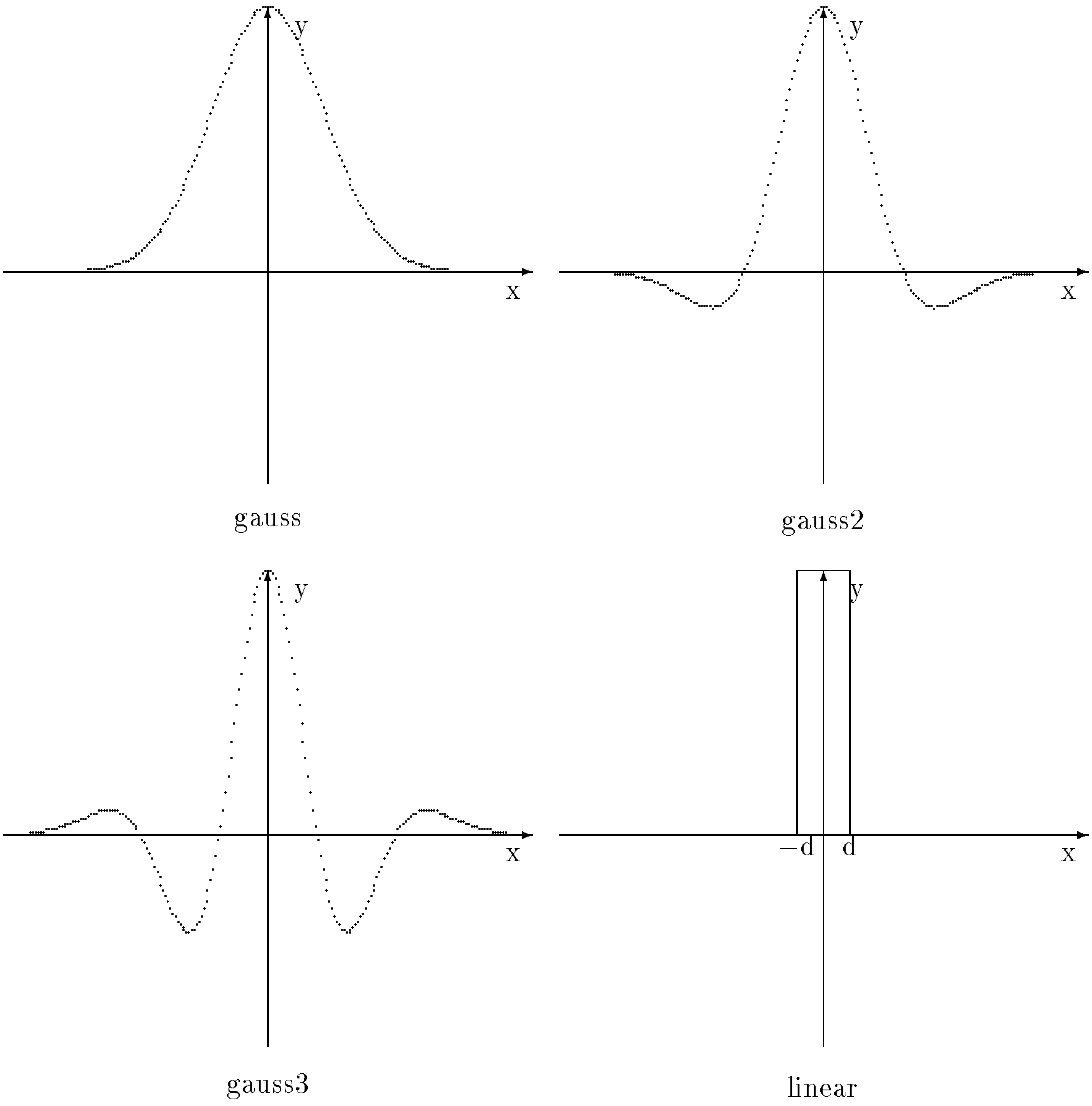

Abbildung 8: Distanzfunktionen

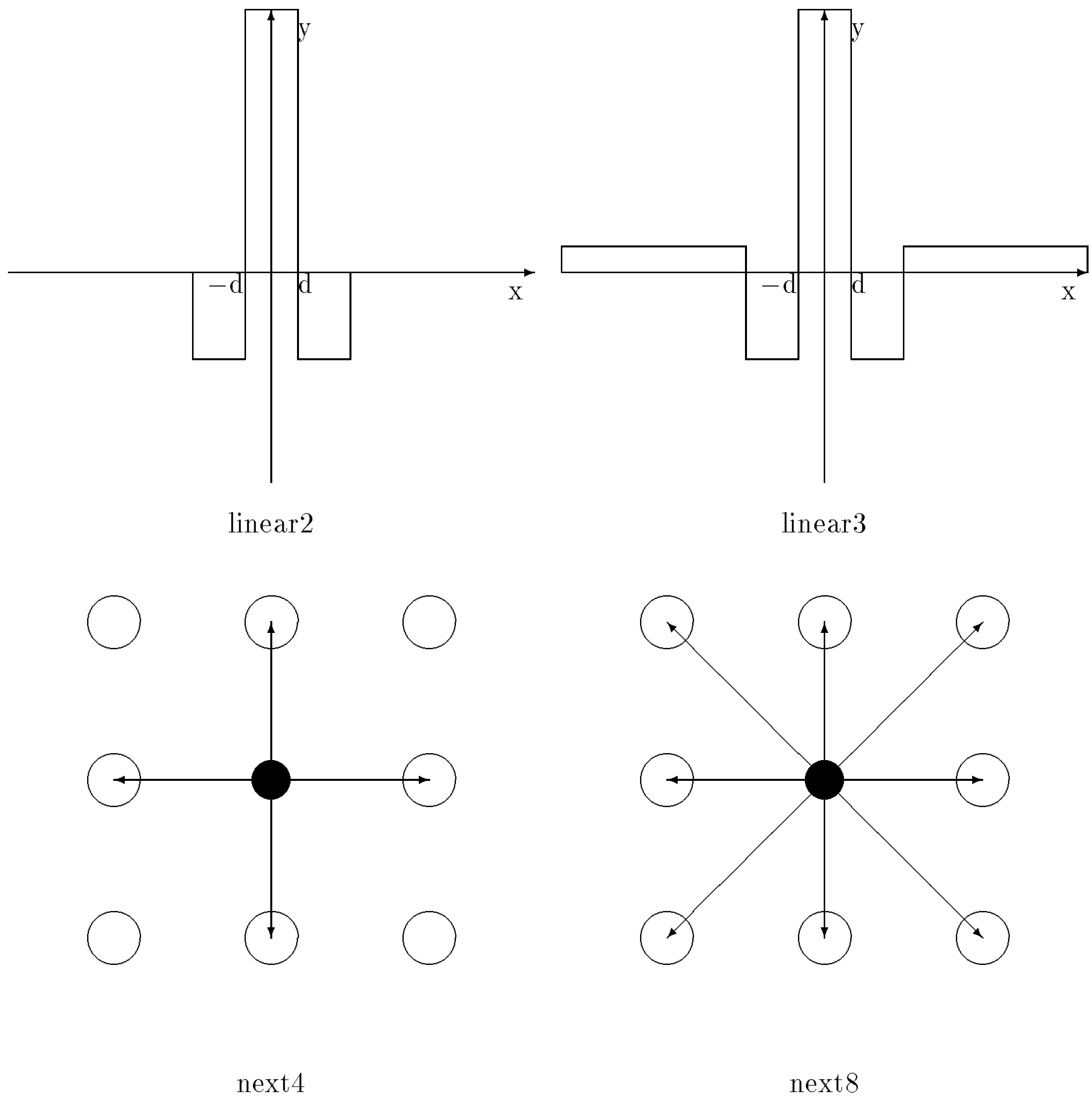

Abbildung 9: Distanzfunktionen (Fortsetzung)

DistFactorHL
Innerhalb der hier angegebenen Anzahl von Schritten sinkt der Fernwirkungsfaktor auf die Hälfte. Damit halbiert sich auch die Fernwirkung selbst. Siehe auch LRateHL.

MaxDist
Die maximale Entfernung, in der die Fernwirkung eine Änderung der Gewichte hervorruft. Ist MaxDist 0, so reicht die Fernwirkung abhängig von Alpha unendlich weit; bei Werten größer 0 reicht die Fernwirkung bis zu dieser Entfernung und springt bei größeren Entfernungen auf 0.

RandomStartConfig
Die Gewichte eines jeden neu konfigurierten Netzes erhalten abhängig von RandomStartConfig zufällige Anfangswerte. Bei `dot` hat jedes Gewicht den Anfangswert 0.5, bei `large` sind es Zufallswerte im gesamten Einheitsintervall, bei `small` liegen die Zufallswerte zwischen 0.45 und 0.55. `corner` ist identisch mit `small`, nur daß die Ausbreitung der Karte dadurch beschleunigt wird, daß alle Eckneuronen schon die Eckwerte als Anfangsgewichte zugewiesen bekommen. Im zweidimensionalen Fall bedeutet dies, daß den Gewichten der vier Eckneuronen die Koordinaten der Ecken (0;0), (0;1), (1;0) und (1;1) zugewiesen werden.

Test Configuration
Solange der Benutzer Parameter eingibt, werden diese nicht auf Plausibilität überprüft. Diese Prüfung kann hier explizit angestoßen werden, damit der Benutzer eventuelle Korrekturen sieht. Sie wird auch implizit angestoßen sobald der Benutzer die neue Konfiguration akzeptiert.

OK
Der Benutzer akzepiert die Netzkonfiguration, die dann an die MasPar übergeben wird.

Cancel
geht zurück zum STARTUP MENU.

4.2.3 MAIN MENU

File:
siehe STARTUP MENU

#Steps
spezifiziert, nach wievielen Schritten die aktuellen Gewichte neu dargestellt werden.

Run #Steps
stößt n Lernschritte an, wobei n die Zahl ist, die unter Anzahl Schritte eingegeben wurde. Sind die Lernschritte durchgeführt, so werden die neuen Gewichte und der neue Status (s.o.) dargestellt.

Run
schaltet in den kontinuierlichen Lernmodus. Solange dieser Modus nicht gestoppt ist, wird ständig gelernt und nach jeweils so vielen Schritten, wie in Anzahl Schritte angegeben wurde, werden die aktuellen Gewichte von der MasPar zur X-Oberfläche übertragen und dort dargestellt.

Change Parameters
geht zum CHANGE MENU. Dort können Parameter des aktuellen Netzes geändert werden, z.B. um ein besseres Lernverhalten zu erreichen.

Save Picture
speichert die aktuellen Gewichte in der identischen Art und Weise wie sie im Graphikfenster dargestellt werden als PostScript-File auf Platte. An den angegebenen Filenamen wird die Extension `.ps` angehängt. Existiert diese Datei schon, so wird sie nicht überschrieben, sondern es werden so lange Extensionen der Form `.`n`.ps` generiert und an den Filenamen angehängt bis ein File dieses Namens nicht existiert.

Save Network
wechselt zum SAVE MENU, wo eine Netzbeschreibungsdatei oder eine Datei mit reinen Daten gespeichert werden kann.

Restart
Die aktuelle Konfiguration wird verworfen; der Benutzer kommt wieder ins STARTUP MENU, wo er eine andere Netzbeschreibungsdatei laden oder eine neue Konfiguration erstellen kann.

Exit
beendet den Simulator. Siehe Exit im STARTUP MENU.

4.2.4 RUN MENU

Wenn der Benutzer dieses Menü sieht, läuft der Simulator im kontinuierlichen Lernmodus. Es enthält nur die Möglichkeit STOP, um diesen Modus wieder abzuschalten. Die Meldung `Stopped` wird ausgegeben, der gerade laufende Lernschritt wird aber noch beendet.

4.2.5 CHANGE MENU

Die Felder in diesem Menü haben dieselbe Bedeutung wie die entsprechenden Felder im CREATE MENU, dienen hier aber dazu, die Parameter eines bestehenden Netzes zu ändern, statt wie dort die Anfangsparameter eines neuen Netzes festzulegen. Hier kann also zwischen zwei Lernschritten die Lernrate, der Fernwirkungsfaktor usw. geändert werden. Beim nächsten Lernschritt werden dann diese neuen Parameter benutzt. Wenn der Benutzer etwas an einem logischen Parameter (z.B. der Lernrate) ändert, sollte er immer alle zu diesem Parameter gehörigen Felder ändern (hier also LRateStart, LRateStop, LRateHL und DecLRate), muß es aber nicht. Wird allerdings DecLRate oder DecDistFactor geändert, so *muß* LRateHL bzw. DistFactorHL geändert werden. Das kommt daher, daß die lineare und die exponentielle Sinkkurve sich nur im Startwert und dann nach HL Schritten in Startwert/2 schneiden und keine sinnvolle Annahme über die neue beabsichtigte Sinkgeschwindigkeit gemacht werden kann. Die Änderungen werden mit **OK** akzepiert und ab dem nächsten Lernschritt benutzt oder mit **Cancel** verworfen. In diesem Fall bleiben die alten Parameter gültig.

4.2.6 SAVE MENU

Von diesem Menü aus kann der Benutzer eine Netzkonfiguration speichern. Der Filename kann noch angegeben oder geändert werden. Entweder wird eine Netzbeschreibungsdatei geschrieben, deren Format eine Erweiterung des SNNS Version 1.4-3D Formates ist (Extension `.net`), oder es werden nur Daten geschrieben (Extension `.wts`). Diese Möglichkeit dient dazu, ein trainiertes Netz in einfacher Weise einem Anwendungsprogramm zugänglich zu machen. Die genauen Formate sind im Anhang beschrieben. Falls der Benutzer es sich anders überlegt hat, kommt er mit **Quit** zum MAIN MENU zurück.

5 Ändern der Transformation

Dieses Kapitel beschreibt, wie ein Benutzer die Transformation ändern kann. Da die Transformation (siehe Kapitel 2) so allgemein wie möglich sein sollte, ist es nicht möglich, einen einfachen, leicht bedienbaren und gleichzeitig mächtigen Mechanismus zu implementieren. Stattdessen wurde eine Programmschnittstelle implementiert, die es dem Benutzer ermöglicht, beliebige Transformationen durch C- bzw. MPL-Funktionen zu spezifizieren. Jede andere Lösung hätte trotz wesentlich höherem Aufwand zu viel unflexibleren und langsameren Transformationsmöglichkeiten geführt.

Eine Transformation setzt sich zusammen aus einer Transformationsfunktion und einer Retransformationsfunktion. Beide folgen einem ähnlichem Schema. Die eine transformiert die ein- bis dreidimensionalen Weltkoordinaten aus dem Welt-Raum $\mathcal{W}$ in den Sensorraum $\mathcal{S}$ und bietet damit eine einfache Möglichkeit, der selbstorganisierenden Karte beliebig verteilte Eingabevektoren beliebiger Dimensionalität zu präsentieren. Die andere transformiert die Gewichtsvektoren, die dieselbe Dimensionalität und Semantik haben wie die Eingabevektoren, zurück zu Weltkoordinaten, damit sie graphisch dargestellt werden können, da die gewählte Darstellungsart nur ein- bis dreidimensionale Strukturen unterstützt.

Statt einer formalen Darstellung sei der Transformationsmechanismus anhand einiger Beispiele erklärt. Die meisten davon benutzen nur die InputDimension 2, sollten aber als Anleitung für weitere Entwicklungen ausreichend sein.

5.1 Beispiel: Identitätstransformation

Die Identitätstransformation hat die minimalen Transformations- und Retransformationsfunktionen. Diese sind:

```
/* Transformationsfunktion */
#include "Kohonen.h"
char ID[] = "identity";
int InputDimension = 0;
int transform(vp)
sVector *vp;
{ return accept;
}
```

```
/* Retransformationsfunktion */
char ID[] = "identity";
int InputDimension = 0;
void retransform(v)
float *v;
{}
```

Erklärung: Die Transformationsfunktion benutzt `Kohonen.h`, um die Typen `sVector` und die möglichen Rückgabewerte `accept`, `reject` und `error` bekannt zu machen. Beide Files enthalten eine `ID`, anhand derer die X-Oberfläche und der Kern bestimmen, ob sie mit derselben Transformation arbeiten. Der Integer `InputDimension` muß ebenfalls in beiden Files übereinstimmen. Hier werden zwei Fälle unterschieden: Erstens kann der Wert 0 sein. Dann wird angenommen, daß die Transformation keine Dimensionsänderung umfaßt und bei allen Netzdimensionen funktioniert. Ist einer der `InputDimension`-Werte gleich 0, so kann der andere beliebig sein. Zweitens, bei Werten ungleich 0, wird angenommen, daß alle Weltkoordinaten zu einer Dimensionalität transformiert werden, die InputDimension entspricht. Dies ist aber nicht die einzige Möglichkeit, um eine Dimensionsreduktion zu erhalten, siehe Kapitel 7.6.

Die Transformationsfunktion `transform()` hat einen `sVector` als Argument. Dieser Typ ist definiert als `struct { int dim; float x[]; } sVector`. Das `x`-Array enthält `dim` Zufallswerte je zwischen 0 und 1, wobei `dim` die Netzdimension ist und die Zufallswerte gleichverteilt sind[11]. Daher rechnet auch die X-Oberfläche damit, daß die Elemente der darzustellenden Vektoren nach der Retransformation ebenfalls zwischen 0 und 1 liegen. Andere Werte führen zu keinem Fehler, wenn sie aber außerhalb des Graphikfensters liegen, werden sie nicht mehr dargestellt. Der `x`-Vektor kann entweder (auf beliebige Art unter dem Einsatz aller geeigneten C-Funktionen) manipuliert werden oder man wählt nur die für den jeweiligen Zweck geeigneten `x`-Vektoren aus, indem man bei akzeptierten (und eventuell vorher transformierten) `x`-Vektoren die Funktion mittels `return accept;` verläßt, während man sonst `reject` zurückgibt. In diesem Fall wird die `transform()`-Funktion so lange mit anderen Argumenten aufgerufen bis sie `accept` liefert. Es gibt noch einen dritten Wert, den `transform()` zurückgeben kann, nämlich `error`. Der Effekt davon ist, daß das Lernen abgebrochen und eine von Benutzer zu definierende Fehlermeldung ausgegeben wird. In Kapitel 5.5 sieht man ein Beispiel, wie `error` benutzt werden kann.

Durch diese Rückgabeparameter muß die Transformationsfunktion keine totale Funktion sein, denn Werte, für die sie nicht definiert ist, können einfach mit `return reject` zurückgewiesen werden. Diesen Mechanismus könnte man Selektion nennen, da hier Teile des Eingaberaumes selektiert und andere ignoriert werden können.

Andererseits *muß* aber die Retransformationsfunktion total sein, da nicht a priori bestimmt werden kann, welche Gewichte sich im Laufe des Lernens ergeben. Sie kann die Umkehrfunktion der Transformationsfunktion sein, muß es aber nicht. Wenn die Transformationsfunktion z.B. kartesische Koordinaten in Polarkoordinaten umwandelt, sollte die Retransformationsfunktion die Umwandlung rückgängig machen. Wenn aber nur die Wahrscheinlichkeitsverteilung der Eingabevektoren geändert wurde, darf dies nicht rückgängig gemacht werden, da sonst der Effekt wieder verlorenginge. Und da obige Transformationsfunktion nichts tut, reicht als Retransformationsfunktion die leere Funktion.

[11] Genauer gesagt sind die Werte nur so gleichverteilt, wie die C-Funktion `random()` sie liefert.

5.2 Beispiel: Eingaben nur über der Halbdiagonale

Angenommen, man will simulieren, daß Eingaben nur in dem Dreieck über der ersten Halbdiagonalen vorkommen. Dann gibt es zwei Möglichkeiten, eine benutzt den Selektionsmechanismus, die andere eine Spiegelung:

```
/* mit Selektion */
#include "Kohonen.h"
char ID[] = "diag1";
int InputDimension = 2;
int transform(vp)
sVector *vp;
{ if (vp->x[0] > vp->x[1]) return reject; else return accept;
}
```

```
/* mit Spiegelung */
#include "Kohonen.h"
char ID[] = "diag2";
int InputDimension = 2;
int transform(vp)
sVector *vp;
{ float h;
  if (vp->x[0] > vp->x[1])
  { h = vp->x[0];
    vp->x[0] = vp->x[1];
    vp->x[1] = h;
  }
  return accept;
}
```

Die beiden Versionen unterscheiden sich in der Effizienz nicht wesentlich, man sollte allerdings darauf achten, daß man bei der Selektion nicht zu enge Grenzen setzt, da sonst zu viele Aufrufe der Funktion nötig werden und der Zufallsgenerator nicht garantiert, daß wirklich jede Zufallszahl erzeugt wird. Man sollte immer versuchen, ein möglichst effizientes Gleichgewicht zwischen Selektion und umständlichen Berechnungen zu finden. Als Retransformationsfunktion kann in diesem Fall obige leere Funktion beibehalten werden, nur die `ID` muß entsprechend geändert werden. Zu beachten ist hier, daß die Funktionen nur für die InputDimension 2 funktionieren und deshalb dies angegeben werden muß.

5.3 Beispiel: Polarkoordinaten

Solange der Benutzer durch eigene Transformationen dies nicht ändert, werden kartesische Koordinaten benutzt. Möchte man Polarkoordinaten, so kann man dies durch die folgenden Funktionen ausdrücken:

```
/* Transformations: kartesisch -> polar */
#include "Kohonen.h"
#include <math.h>
char ID[] = "polar";
int InputDimension = 2;
int transform(vp)
sVector *vp;
{ float r,w;
  r = sqrt((vp->x[0]*vp->x[0])+(vp->x[1]*vp->x[1]));
  w = (vp->x[0] > 0.0) ? atan(vp->x[1] / vp->x[0]) : 0.0;
  vp->x[0] = r;
  vp->x[1] = w;
  return accept;
}

/* Retransformation: polar -> kartesisch */
char ID[] = "polar";
int InputDimension = 2;
void retransform(v)
float *v;
{ float x,y;
  x = v[0] * cos(v[1]);
  y = v[0] * sin(v[1]);
  v[0] = x;
  v[1] = y;
}
```

Hier sind zwei Dinge zu bemerken: Erstens, daß das Argument von `retransform()` ein `float`-Array ist und nicht von Typ `sVector` wie bei `transform()`. Zweitens ist immer zu beachten, daß die transformierten Vektorelemente nicht zu große Werte annehmen dürfen, da sie noch bei der Normierung während der Abstandsberechnung potenziert werden. Der Radius ist begrenzt durch $\sqrt{2}$, hier besteht also keine Gefahr. Anders beim Winkel. Der Arcustangens ist nicht nach oben begrenzt. Die Division durch 0 ist zwar abgefangen, trotzdem kann er, spätestens wenn er quadriert wird, den `float`-Zahlenbereich überschreiten und damit zu einem fatalen (den Kern beendenden) Fehler führen. Hier muß man bei der Implementierung der Transformationsfunktion weitere Maßnahmen ergreifen.

5.4 Beispiel: Geänderte Häufigkeitsverteilung

Die Komponenten der in `vp` an `transform()` übergebenen Eingabevektoren sind gleichverteilt. Möchte man im zweidimensionalen Fall die Wahrscheinlichkeitsverteilung so ändern, daß die Wahrscheinlichkeit eines Eingabevektors von dessen x-Wert abhängt, so geht das am einfachsten folgendermaßen:

```
#include "Kohonen.h"
char ID[] = "identity";
int InputDimension = 2;
int transform(vp)
sVector *vp;
{ if (drand() > vp->x[0]) return accept; else return reject;
}
```

Hier wird die Funktion `drand()`, die ebenfalls in `Kohonen.h` definiert ist, benutzt. Sie liefert einen zwischen 0 und 1 gleichverteilten float-Wert. Außerdem wird die `ID` so benutzt, wie sie gedacht ist: Sie bestimmt, welche Retransformationsfunktion benutzt werden kann, und stellt keinen Namen der Transformationsfunktion dar.

5.5 Beispiel: Werte aus einer Datei

Manchmal möchte man die Eingabevektoren nicht von einem Zufallsgenerator erstellen lassen, sondern Vektoren aus einer anderen Quelle benutzen. In diesem Falle liegen sie z.B. als Datei vor. Auch dazu kann der Transformationsmechanismus benutzt werden:

```
/* Vektoren von Datei "vec.dat" */
#include "Kohonen.h"
#include <stdio.h>
#include <unistd.h>
char ID[] = "identity";
int InputDimension = 2;
FILE vecfile = NULL;
int transform(vp)
sVector *vp;
{ char buffer[100];
  if (vecfile)
  { if (!fgets(buffer,99,vecfile))
    { fclose(vecfile);
      vecfile = NULL;
```

```
      return reject;
    }
    sscanf(buffer,"%g %g",&(vp->x[0]),&(vp->x[1]));
    return accept;
  } else
  { vecfile = fopen("vec.dat","r");
    if (vecfile) return reject; else
    { strcpy(msg,"Cannot read \'vec.dat\'");
      return error;
    }
  }
}
```

Diese Transformationsfunktion liest Werte von der Datei `vec.dat`. Ist die Datei zu Ende, so wird sie vom Anfang wiederholt. (Existiert die Datei nicht, so wird eine Fehlermeldung in der globalen Variablen `msg` übergeben und `error` zurückgeliefert. Um das Ausgeben der Meldung braucht man sich bei der Implementation der Transformationsfunktion nicht zu kümmern. Ebenso wird jegliches Lernen so lange verhindert, bis die Datei `vec.dat` existiert.

5.6 Einbinden einer neuen Transformation

Wenn man eine neue Transformationsfunktion oder eine neue Retransformationsfunktion geschrieben hat, muß man zunächst sicherstellen, daß die Transformationsfunktion die Extension `.m` und die Retransformationsfunktion die Extension `.c` hat. Außerdem werden folgende Dateien benötigt: `Kohonen.m` oder `Kohonen.o`, `KNet.c` oder `KNet.o` sowie `Kohonen.h` und `KNet.h`.

Dann muß man die Programme auf den Maschinen neu kompilieren und zusammenbinden, auf denen sie später laufen sollen, also den Kern auf der MasPar und die X-Oberfläche auf einer Workstation. Die Maschinen müssen nur eine Bedingung erfüllen: Sie müssen über eine gemeinsame Platte verfügen, da die Kommunikation zwischen den beiden Programmen momentan über Dateien läuft.

Dazu sind folgende Befehle nötig:
Zum Erstellen einer neuen X-Oberfläche namens `KNet1` mit der Retransformationsfunktion `retrans1.c`:
`cc -o KNet1 KNet.c retrans1.c -lXaw -lXmu -lXt -lXext -lX11`

Zum Erstellen eines neuen Kerns namens `Kern1` mit der Transformationsfunktion `trans1.m`:
`mpl -o Kern1 Kohonen.m trans1.m`

Näheres zum Kompilieren und zu dazu benötigten Ressourcen steht im nächsten Kapitel.

6 Portierungsaspekte

Dieses Kapitel enthält Angaben darüber, welche Hard- und Software zur Benutzung des Simulators nötig ist, welche Dateien dazugehören und wie er kompiliert wird.

Der Simulatorkern läuft auf einer MasPar mit beliebig großem Prozessorarray. Er paßt sich automatisch während der Laufzeit an die jeweiligen Gegebenheiten an.

Die X-Oberfläche läuft auf einer beliebigen UNIX-Workstation, die folgende Bedingungen erfüllen muß:

- X-Windows Version X11 Release 4 muß installiert sein.
- Das Athena widget set muß installiert sein.
- Ein C-Compiler muß verfügbar sein.
- Die include-Dateien und Bibliotheken für UNIX, X und das Athena widget set müssen im Suchpfad des Compilers sein oder der richtige Suchpfad muß beim Übersetzen angegeben werden.
- Es muß eine Platte geben, auf die sowohl die Workstation als auch die MasPar schreibend und lesend zugreifen können.

Spezielle Angaben im Ressource-File von X-Windows werden nicht vorausgesetzt, alles hier nötige ist in die Sourcefiles hineincodiert.

Zum Simulator gehören die Dateien

`Kohonen.m` und `Kohonen.h`:	der Kern des Simulators
`KNet.c` und `KNet.h`:	die X-Oberfläche des Simulators
`sko.c` und `sko.h`:	der alternative serielle Kern
`transform.m` und `retransform.c`:	Dateien mit der Identitätstransformation
`transform.c`:	Identitätstransformation für seriellen Kern

Sie liegen im dem Verzeichnis `/az3/bayer/diplom/src`, die LaTeX- und PostScript-Quellen für diesen Text sowie eine druckfertige PostScript-Version liegen unter `/az3/bayer/diplom/doc`.

Ein Simulator mit der Identitätstransformation kann, wenn alle obigen Bedingungen erfüllt sind, einfach erzeugt werden:

Erzeugen des Kerns:
```
mpl -o Kern Kohonen.m transform.m
```

Erzeugen der X-Oberfläche:
```
cc -o oberflaeche KNet.c retransform.c -lXaw -lXmu -lXt -lXext -lX11
```

Erzeugen des seriellen Kerns:
```
cc -o serKern sko.c transform.c -lm
```

7 Ergebnisse und Beispiele

Dieses Kapitel zeigt einige Beispiele von trainierten Netzen und schildert bei der Benutzung des Simulators erlangte Erfahrungen mit selbstorganisierenden Karten. Alle Abbildungen stammen aus dem Simulator, sie wurden von diesem als PostScript-File gespeichert.

Zunächst zur Darstellungsweise. Bei eindimensionalen Netzen wird auf der x-Achse die Nummer des Neurons und auf der y-Achse dessen Gewicht aufgetragen. Zusätzlich wird das Gewicht durch eine vertikale Linie über dem Neuron dargestellt und die Endpunkte dieser Linien werden verbunden, wodurch sich eine linear interpolierte Kurve über der Neuronenkette ergibt.

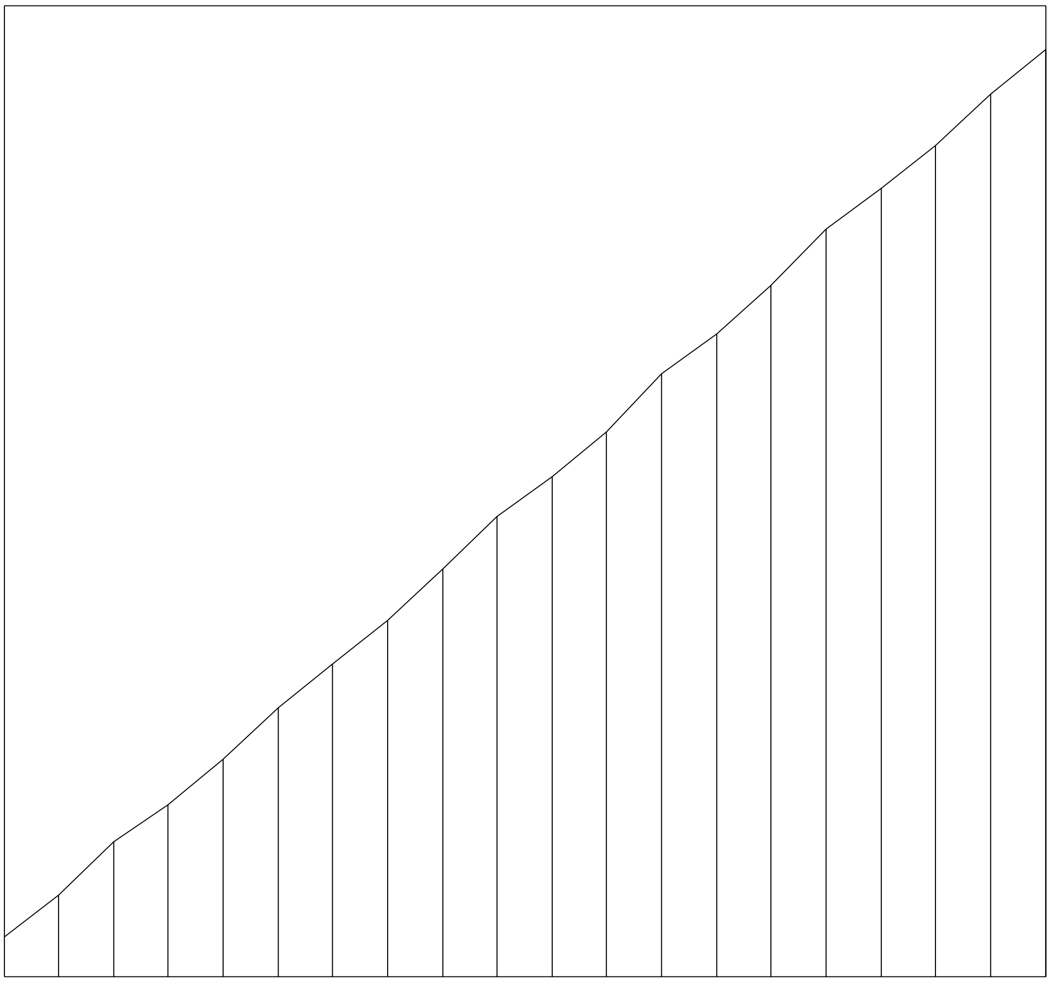

Abbildung 10: Beispiel für die Darstellung einer eindimensionalen Karte.

Bei zwei- und dreidimensionalen Netzen werden die retransformierten Gewichtsvektoren als Ortsvektoren interpretiert und so dargestellt. Im dreidimensionalen Fall wird das Netz zusätzlich noch mittels Parallelprojektion in die Darstellungsebene projeziert. Der Ursprung der Abbildungen ist unten links; zur besseren Orientierung ist im ein- und zweidimensionalen Fall ein Quadrat, im dreidimensionalen Fall das Drahtmodell eines Würfels überlagert; die Kantenlänge ist immer eins.

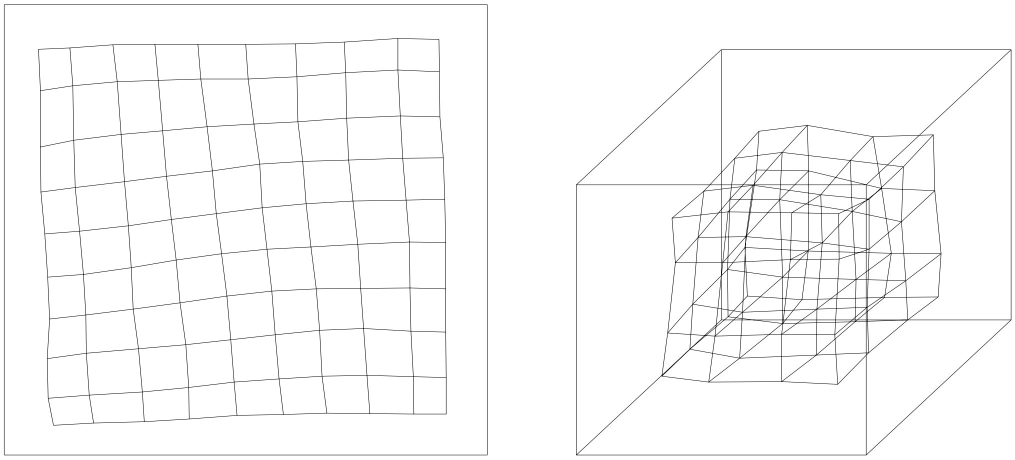

Abbildung 11: Darstellung von zwei- und dreidimensionalen Karten.

Obwohl die Reize im gesamten Eingaberaum gleichverteilt sind, füllen die Karten diesen nicht ganz aus, sondern lassen an den Rändern einen Bereich frei. Genauer: Die inneren Neuronen sind (bei einer ideal trainierten Karte) äquidistant, während die äußersten Neuronen um einen Betrag, der zu der Anzahl der Neuronen in dieser Richtung umgekehrt proportional ist, nach innen verschoben sind. Kohonen ([1]) nennt dies den *boundary effect*. Für seine Erklärung und die Herleitung dieser Zahl sowie weiterer Theoreme betreffs selbstorganisierender Karten siehe [2].

7.1 Beispiel: Defekte Topologie

Wenn die Fernwirkung zu klein ist oder zu schnell gesunken ist, können sich weit voneinander entfernt liegende Neuronen gegenseitig nicht stark genug beeinflussen, um eine globale Ordnung zu erreichen. Teilbereiche des Netzes ordnen sich zwar korrekt an, ihre gegenseitige Ordnung ist aber gestört und wird sich auch nicht mehr korrigieren. Nur durch eine Erhöhung der Fernwirkung läßt sich eine globale Ordnung doch noch herstellen. (Hodges und Wu schlagen in [7] vor, den Radius der Fernwirkung anfangs zwischen 0 und einem Maximum oszilieren zu lassen, um die Neuronen in eine topologisch richtige Lage zu „schütteln".) Ein guter Startwert bei der Gauß'schen Distanzfunktion ist das Maximum von Nx, Ny und Nz. Thomas Martinetz nennt ein Viertel der Netzgröße als günstigen Startwert[12].

Solche defekten Topologien sind häufig, wenn man als Distanzfunktion α next4 oder next8 verwendet, da hier die Fernwirkung auf die unmittelbare Umgebung des selektierten Neurons beschränkt ist. Diese sind hauptsächlich zum Feintuning von Karten geeignet, deren globale Struktur schon gelernt ist.

In der folgenden Abbildung wurde die Topologie in der rechten und der linken Seite der Karte korrekt gelernt, die Beziehung zwischen den Hälften ist aber defekt.

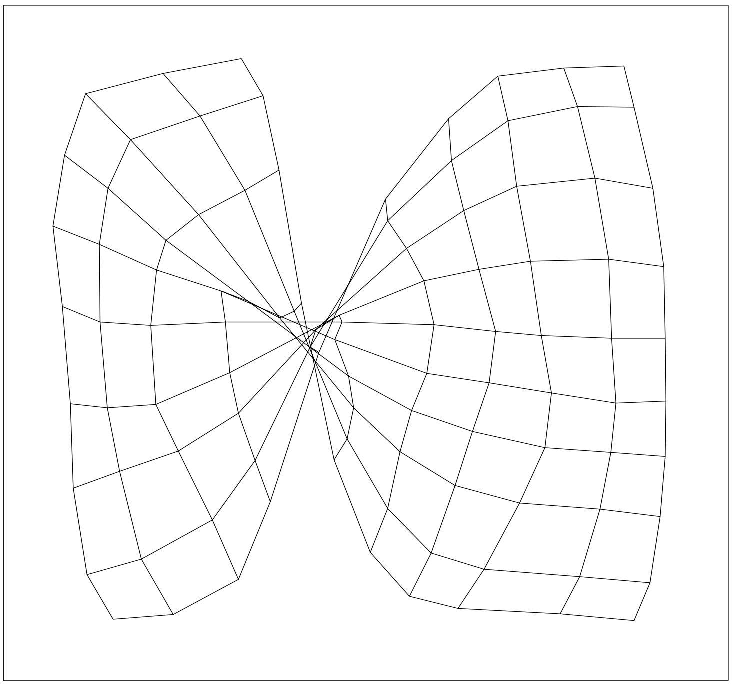

Abbildung 12: Karte mit topologischem Defekt

[12] 1991, personal communication

7.2 Beispiel: Explodierte Karte

Bei einigen Distanzfunktionen (namentlich gauss2, gauss3, linear2 und linear3) gibt es Entfernungen vom selektierten Neuron, in denen sich die Neuronen beim Lernen nicht auf den Reiz zu, sondern von ihm weg bewegen. Das hat den Sinn, ein Ausbreiten der Karte zu beschleunigen. Dies birgt aber die Gefahr in sich, daß Neuronen immer weiter von den Reizen wegbewegt werden: die Karte explodiert. (Manche Autoren setzen normierte Gewichte voraus, was diesem Effekt möglicherweise entgegenwirkt. Der Simulator normiert seine Gewichte aus Aufwands- und Visualisierungsgründen nicht.) Ein ähnlicher Effekt kann sich auch bei Lernraten größer 1 ergeben.

Bei dieser Abbildung wurde der Bereich außerhalb des Einheitsquadrats nachträglich abgeschnitten, damit die Graphik auf einen begrenzten Bereich der Seite beschränkt bleibt. Normalerweise werden mit dem Simulator erzeugte Bilder erst an der Seitengrenze abgeschnitten.

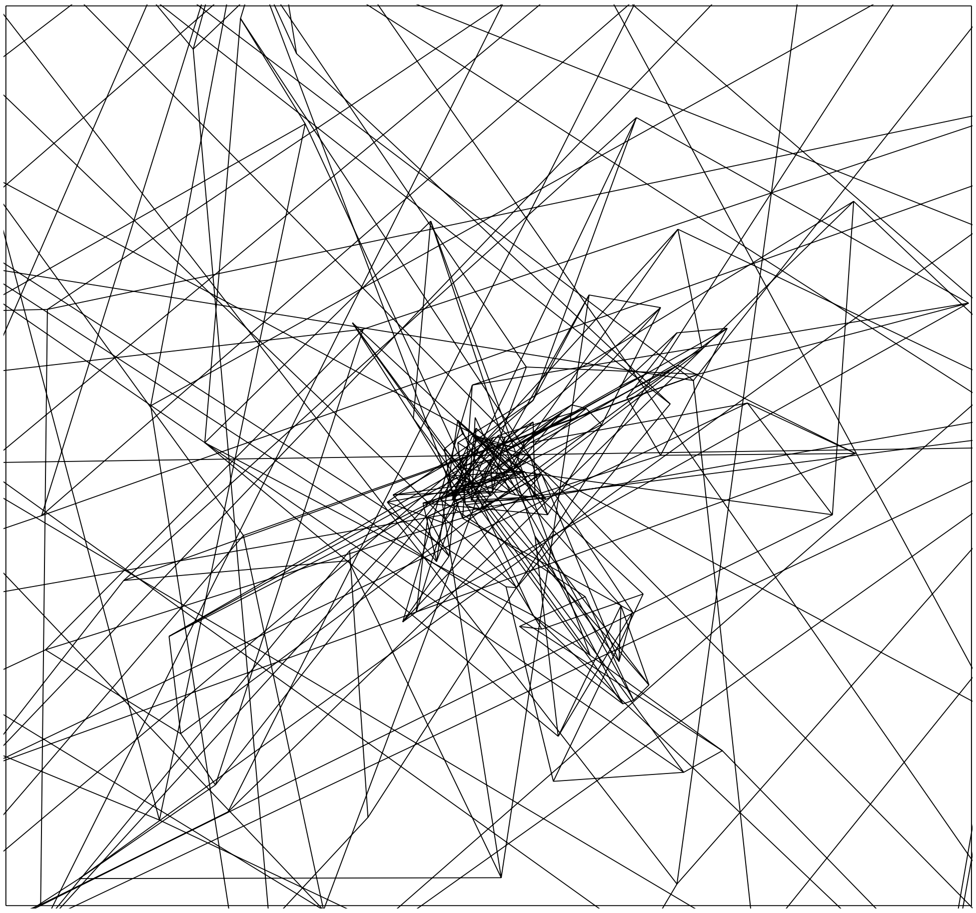

Abbildung 13: Die Karte divergiert.

7.3 Beispiel: Halbdiagonale

Hier wird ein einfaches Beispiel für eine geänderte Wahrscheinlichkeitsverteilung dargestellt. Reize kommen ausschließlich aus dem Dreieck über der ersten Halbdiagonalen, während unter ihr nie Reize auftreten. Selbstorganisierende Karten ziehen sich dann aus Gebieten mit wenigen Reizen zurück und konzentrieren sich in Gebieten mit vielen Reizen.

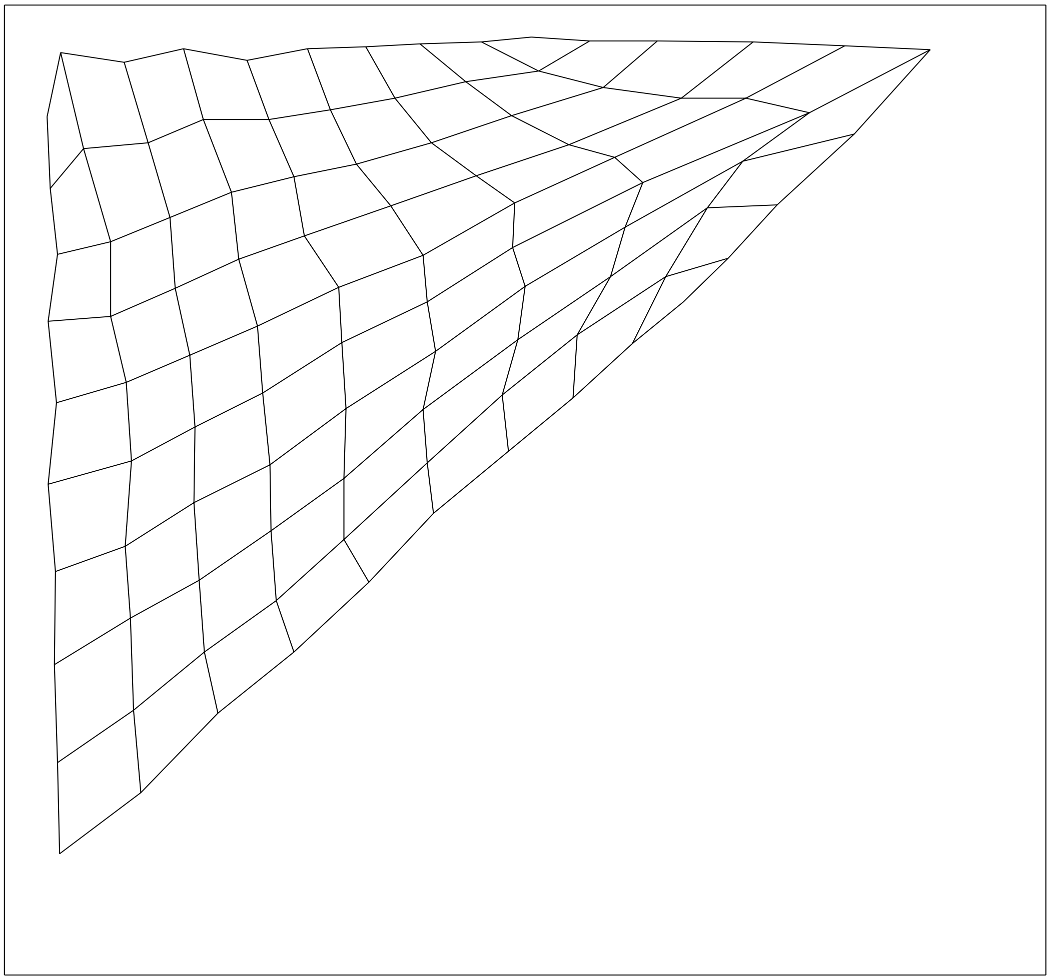

Abbildung 14: Rückzug der Karte aus Bereichen, aus denen keine Eingabereize kommen.

7.4 Beispiel: Diskrete Reize

Ähnlich wie im letzten Beispiel zieht sich ein eindimensionales Netz auf drei Stufen zurück, wenn es nur drei diskrete Eingaben bekommt. Die ständige Wiederholung der Reize 0.25, 0.5 und 0.75 auf ein Netz mit 100 Neuronen führt zu der folgenden Abbildung. Man beachte, daß sich in den Gebieten zwischen den drei Reizzentren keine Neuronen befinden. Diese haben sich auf den Höhen der Reize ungefähr gleichmäßig verteilt.

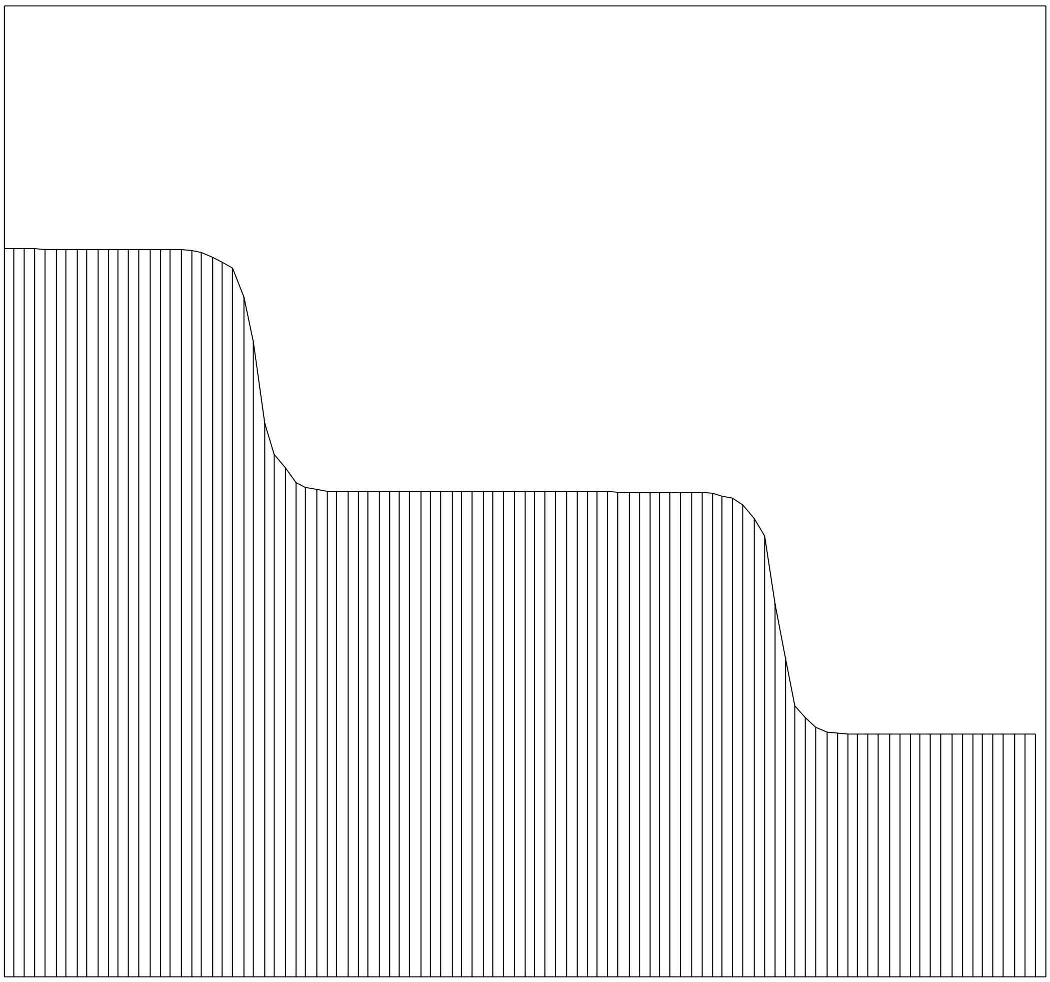

Abbildung 15: Die Gewichte konzentrieren sich auf die Reizzentren.

7.5 Beispiel: Dreieck

Dieses Beispiel stammt aus [4], Seite 135, Abbildung 5.14, ähnelt aber sehr dem in Kapitel 7.3 gezeigten Resultat. Alle Reize stammen hier aus einem gleichschenkligen Dreieck.

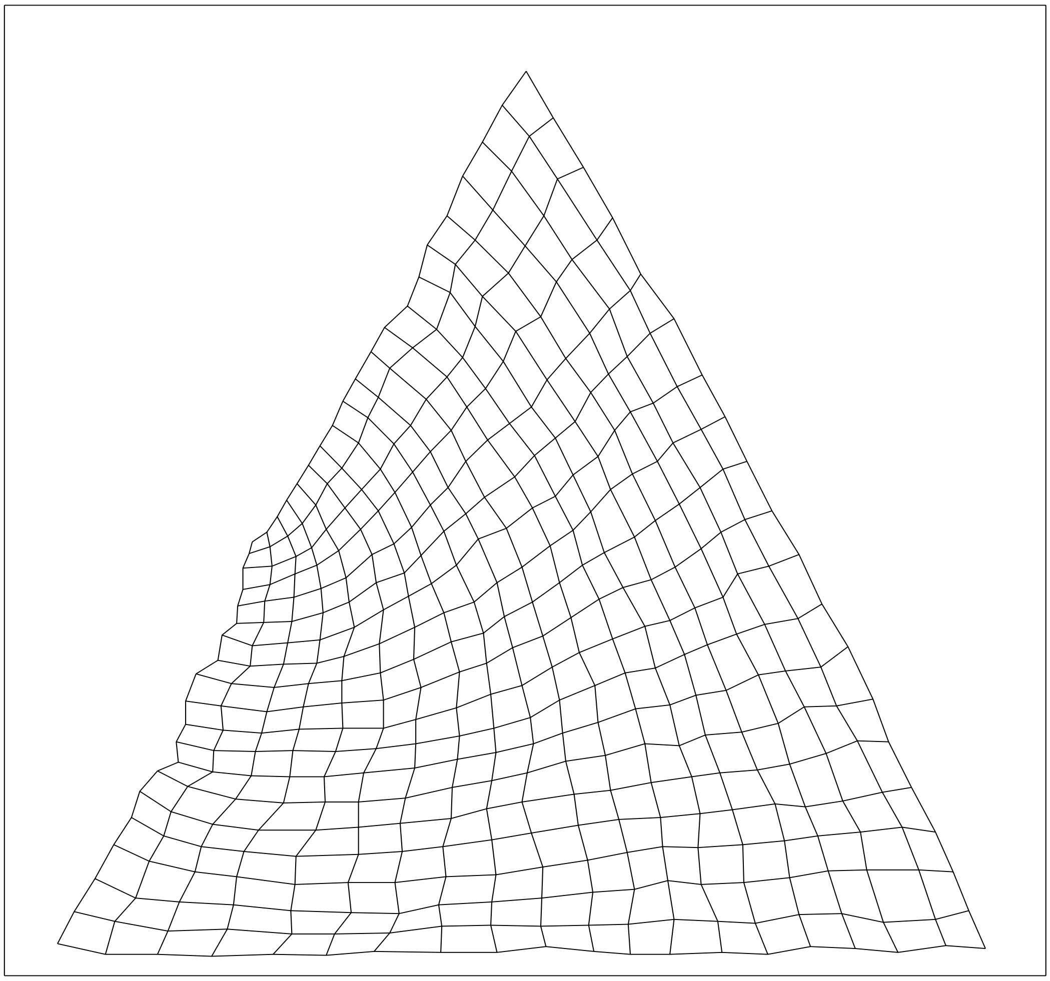

Abbildung 16: Einpassen einer quadratischen Karte in ein Dreieck.

7.6 Beispiel: Dimensionsreduktion

Wenn das Netz eine geringere Dimensionalität als der abzubildende Raum hat, spricht man von Dimensionsreduktion. Dies kann schon ohne Änderung der Transformation dadurch erreicht werden, daß man für eine Netzdimension 1 angibt. Bei der vorgegebenen Identitätsfunktion als Transformation wird ein Netz mit [Nx=10,Ny=10,Nz=0] so interpretiert, daß ein zweidimensionaler Eingaberaum auf ein zweidimensionales Netz abgebildet werden soll. Bei [Nx=10,Ny=0,Nz=0] wird ein eindimensionaler Eingaberaum auf ein eindimensionales Netz abgebildet. Wird aber [Nx=10,Ny=**1**,Nz=0] angegeben, so ist das Netz theoretisch zweidimensional, darum wird ein zweidimensionaler Eingaberaum gewählt. Da es aber in der y-Richtung nur ein Neuron gibt, ist es praktisch ein eindimensionales Netz und kann auch als ein solches benutzt werden. Daraus ergeben sich Abbildungen folgender Art (vgl. auch [13], Seite 65, Abbildung 4.9).

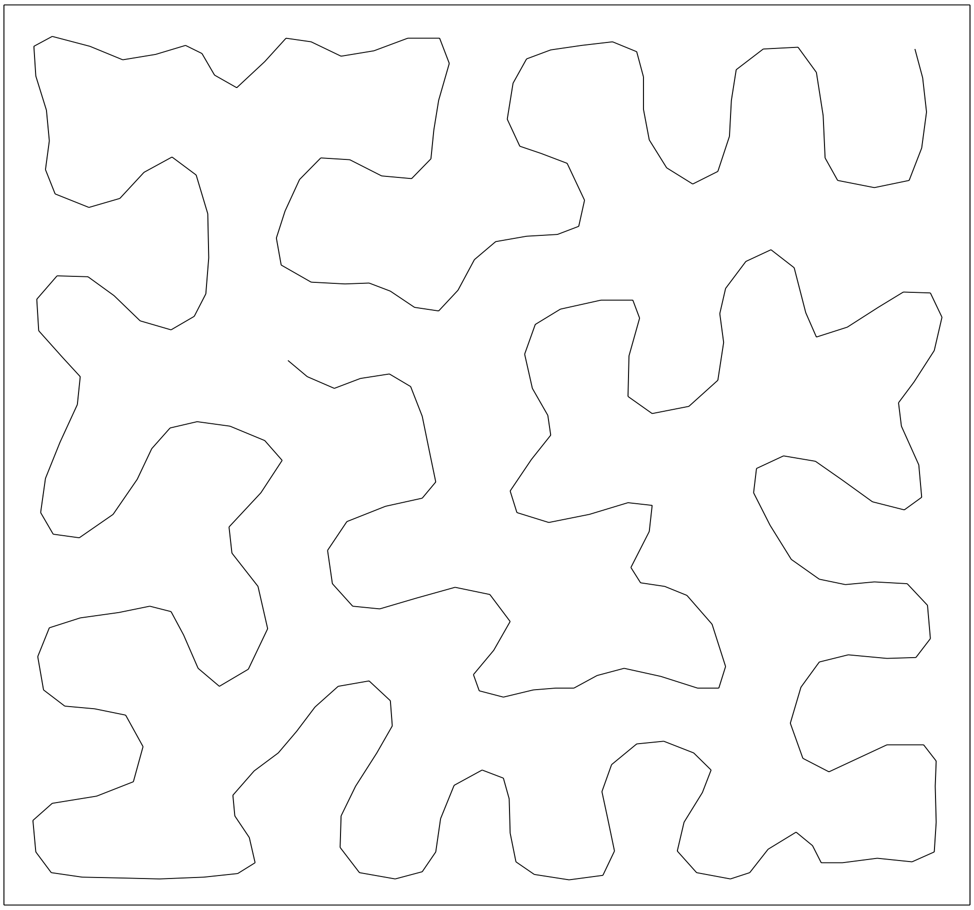

Abbildung 17: Bei der Dimensionsreduktion ergibt sich eine Ähnlichkeit zu Peano-Kurven.

7.7 Beispiel: Travelling Salesman

In [13] und [26] werden selbstorganisierende Karten auch dazu benutzt, das traveling-salesman–Problem (TSP) zu approximieren. Da dazu ringförmige Netzstrukturen nötig sind, die der Simulator nicht unterstützt, kann dieses Problem nicht behandelt werden. Läßt man den Pfad aber offen, ergibt sich ein ähnliches Problem. In diesem Beispiel sind 81 Punkte zu erreichen, die äquidistant in einer 9x9-Struktur angeordnet sind. Läßt man eine 200 Neuronen lange Kette sich frei auf diese Reizzentren verteilen, so wird bei genügend großem anfänglichen Distanzfaktor und genügend kleiner Lernrate diese Variante des TSP approximiert. Dieses Beispiel ist übrigens auch eine Dimensionsreduktion.

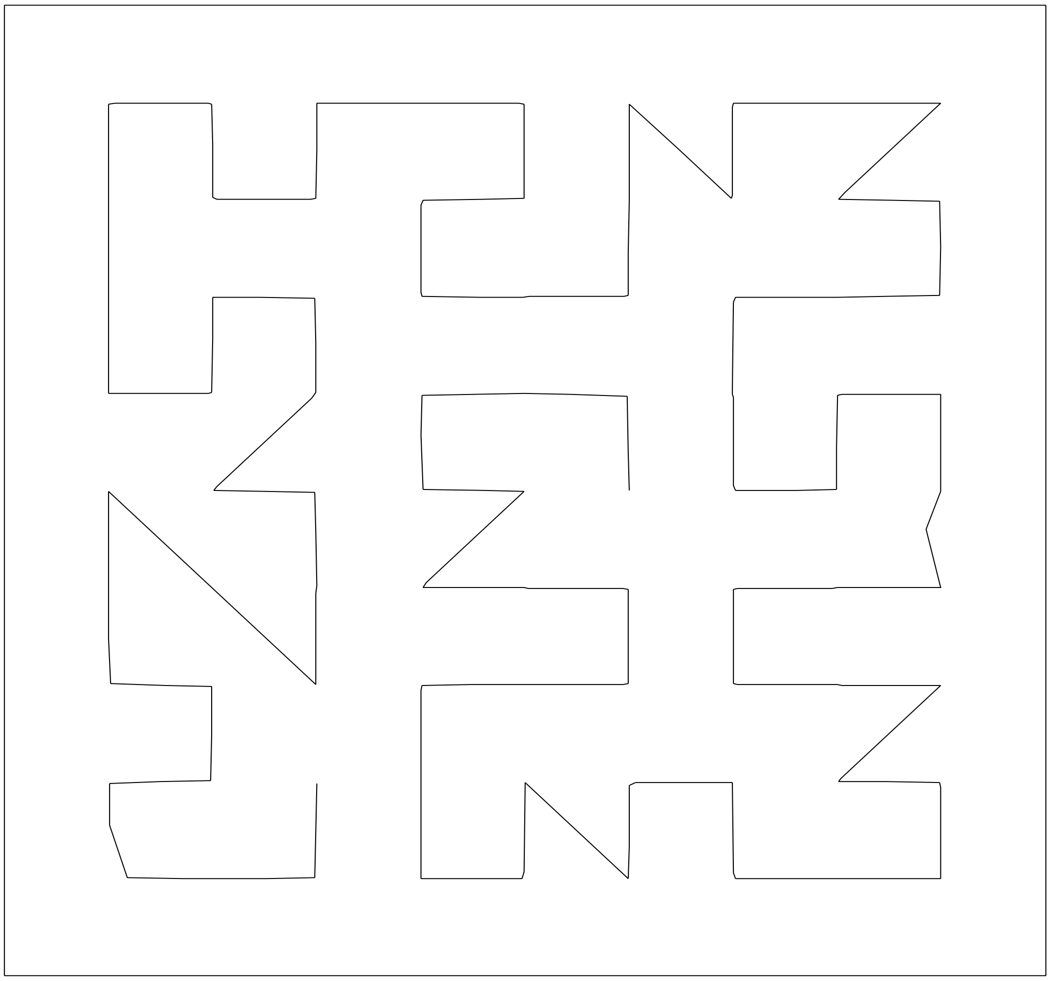

Abbildung 18: Approximation eines minimalen Weges

8 Eine serielle Version des Simulatorkerns

Zusätzlich zur parallelen Version des Simulators wurde aus dieser eine serielle Version entwickelt, die sich auf einer beliebigen UNIX-Workstation übersetzen läßt und somit nicht auf die MasPar angewiesen ist. Die Transformation von parallel nach seriell hat sich – anders als die Gegenrichtung und nur für diesen Fall – als relativ einfach herausgestellt. Das entstandene Programm namens `sko.c` (serial kohonen) unterscheidet sich nur in wenigen Punkten von `Kohonen.m`:

1. Die Größen der ein- und dreidimensionalen Kohonenschichten sind nicht mehr durch die Größe des Prozessorarrays begrenzt (da eine serielle Workstation natürlich kein Prozessorarray hat), sondern nur durch den verfügbaren Speicher.
2. Meldungen, die bei der parallelen Version mit `MPL:` beginnen, fangen hier mit `SKO:` an.
3. Die Meldung `I got the DPU` ist hier bedeutungslos, da es eine Queue wie bei der MasPar bei einer UNIX-Workstation nicht gibt. Sie wird darum auch nicht ausgegeben.

In allen anderen Punkten verhalten sich die parallele und die serielle Version identisch, auch die X-Oberfläche kann die Versionen nicht unterscheiden. Ein wesentlicher Unterschied ist allerdings, daß das Schreiben von einem Megabyte Daten auf einer schnellen Workstation nur 15 Sekunden dauert und nicht 55 Sekunden wie im Moment auf der MasPar. Dadurch hat bei der gegenwärtigen Konfiguration der MasPar die serielle Version bei Netzen, die auf der MasPar durch den Ausgabeengpaß gerade noch animiert werden können, trotz serieller Verarbeitung das bessere Laufzeitverhalten. Näheres über die Geschwindigkeit der Simulatoren siehe Kapitel 9.

Auch in Bezug auf die Transformationsfunktion verhält sich `sko.c` wie `Kohonen.m`, außer daß sie ein `.c`-File statt einem `.m`-File sein muß, da der normale C-Compiler statt dem MPL-Compiler benutzt wird. Außerdem muß `sko.h` statt `Kohonen.h` benutzt werden. Zum Erstellen eines neuen seriellen Kerns namens `sKern1` mit der Transformationsfunktion `trans1.c` kann die Zeile

```
cc -o sKern1 sko.c trans1.c -lm
```

verwendet werden.

9 Leistungsmessungen

Dieses Kapitel enthält Geschwindigkeitsmessungen des Simulators. Alle Messungen wurden auf der MasPar MP-1216A des IPVR durchgeführt. Der Kern war während den Messungen der einzige auf der DPU laufende Prozeß. Die X-Oberfläche lief auf einer sich im normalen multiuser-Betrieb befindlichen SUN ELC, die vom Kern zum Lernen benötigte Zeit ist aber unabhängig davon, auf welcher Maschine sie läuft. Nur die Gesamtzeiten, die auch die Kommunikation zwischen Kern und X-Oberfläche einschließen, hängen von der Geschwindigkeit und Auslaastung der Workstation ab. Deren Geschwindigkeit hat aber einen relativ geringen Einfluß auf die Leistung; die sich aus der wechselnden Belastung der Workstation im normalen Betrieb ergebenden Schwankungen haben mindestens dieselbe Größenordnung.

Tabelle 1 zeigt die Abhängigkeit der Lernzeit von der Anzahl der Neuronen in der selbstorganisierenden Karte und von der Anzahl der Lernschritte. Wie zu erwarten bleibt die Zeit für einen Lernschritt bei geänderter Anzahl der Neuronen fast konstant, da bis auf die Suche des Minimums nach Gleichung 6, die logarithmisch von der Anzahl der Neuronen abhängt, alle Operationen parallel durchgeführt werden solange die Zahl der Neuronen kleiner ist als die Zahl der Prozessorelemente.

#Neuronen	#Lernschritte				
	1	10	100	1000	10000
1	0.002	0.02	0.19	1.99	19.95
10	0.002	0.02	0.20	2.06	20.66
100	0.002	0.02	0.20	2.07	20.68
1000	0.002	0.02	0.21	2.07	20.67

Tabelle 1: Abhängigkeit der Lernzeit von der Anzahl der Neuronen und der Anzahl der Lernschritte. (Parameter: Eindimensionales Netz, Lernrate konstant 0.1, Distanzfaktor konstant 1.0, α ist `next4`, Werte in Sekunden)

Erst wenn die Anzahl der Neuronen größer wird als die Anzahl der PEs steigt die benötigte Zeit an (siehe Tabelle 2). Ein Netz mit 128 mal 128 Neuronen nutzt das Prozessorarray einer MasPar mit 16k PEs gerade voll aus, 129 mal 129 Neuronen sind in jeder Richtung ein Neuron zu größ für das Prozessorarray, dadurch werden zwei Schichten virtueller Prozessoren benötigt. Es ist zu erwarten, daß sich der Zeitverbrauch immer dann erhöht, wenn eine weitere Schicht virtuelle Prozessoren benötigt wird. Aber auch dann steigt der Zeitverbrauch langsamer als linear, wie man in Tabelle 3 sehen kann. Dort wurde eine dreidimensionale Karte benutzt und nur die Anzahl der Neuronen in der z-Dimension erhöht, die auf verschiedene Schichten virtueller Prozessoren abgebildet wird. Dadurch wurde in der ersten Zeile der Tabelle die Lernschleife einmal, in der zweiten Zeile zehnmal und in der dritten Zeile einhundertmal durchlaufen.

Leider sind diese Lernzeiten nur ein Bruchteil der gesamten benötigten Zeit. Tabelle 4 enthält einige Gesamtzeiten, die mit den reinen Lernzeiten in Tabelle 2 verglichen werden können. Gesamtzeiten enthalten außer der Lernzeit auch die Zeiten zur Darstellung und zur Kommunikation zwischen dem Kern und der X-Oberfläche, bei der unter anderem alle Ge-

#Neuronen	#Lernschritte	
	1	1000
1x1	0.003	2.95
10x10	0.003	3.04
100x100	0.003	3.04
128x128	0.003	3.04
129x129	0.005	5.17

Tabelle 2: Konstanter Zeitbedarf solange nicht weitere Schichten virtueller Prozessoren hinzukommen. (Parameter: Zweidimensionales Netz, Lernrate konstant 0.1, Distanzfaktor konstant 1.0, α ist `next4`, Werte in Sekunden)

#Neuronen	#Lernschritte	
	1	1000
1x1x1	0.003	3.91
1x1x10	0.029	28.85
1x1x100	0.277	276.92

Tabelle 3: Abhängigkeit der Lernzeit von der Anzahl der virtuellen Neuronen pro PE. (Parameter: Dreidimensionales Netz, Lernrate konstant 0.1, Distanzfaktor konstant 1.0, α ist `next4`, Werte in Sekunden)

wichte aller Neuronen zur Darstellung durch die X-Oberfläche übertragen werden müssen. Die dort angegebenen Zeiten sind handgestoppte reale Zeiten, also keine reine CPU-Zeit. Man erkennt, daß das Verhältnis zwischen reiner Lernzeit und Gesamtzeit um so besser wird, je mehr Lernzyklen angefordert werden, da der Aufwand für die Kommunikation unabhängig von der Anzahl der Lernzyklen konstant bleibt.

#Neuronen	#Lernschritte	
	1	1000
1x1	2 (667)	5 (1.7)
10x10	5 (1667)	6 (2.0)
100x100	104 (34667)	105 (34)
128x128	160 (53333)	166 (55)
129x129	168 (33600)	175 (34)

Tabelle 4: Gesamtzeiten und in Klammern das Verhältnis zur reinen Lernzeit aus Tabelle 2. (Parameter wie dort, Werte in Sekunden)

Das übliche Maß für die Geschwindigkeit eines Simulators für neuronale Netze ist die Anzahl der Gewichtsänderungen pro Sekunde (connection updates per second, CUPS). Die Tabellen 5 bis 7 entsprechen den Tabellen 1 bis 3, enthalten aber die CUPS statt der Lernzeiten. Eine möglichst volle Ausnutzung jeder Schicht virtueller Prozessoren ist also essentiell für eine gute Leistungsausbeute des Simulators.

Tabelle 8 ist die letzte Tabelle mit Leistungsdaten der MasPar. Sie zeigt die Abhängigkeit

#Neuronen	#Lernschritte				
	1	10	100	1000	10000
1	500	500	526	503	501
10	5000	5000	5000	4854	4840
100	50000	50000	50000	48309	48356
1000	500000	500000	476190	483092	483793

Tabelle 5: CUPS für die Lernzeiten aus Tabelle 1.

#Neuronen	#Lernschritte	
	1	1000
1x1	667	678
10x10	66667	65789
100x100	6666667	6578947
128x128	10922667	10778947
129x129	3328200	3218762

Tabelle 6: CUPS für die Lernzeiten aus Tabelle 2. Diese Tabelle enthält auch die momentane maximale Leistung des Simulators von knapp 11 MCUPS.

#Neuronen	#Lernschritte	
	1	1000
1x1x1	1000	767
1x1x10	1034	1040
1x1x100	1083	1083

Tabelle 7: CUPS für die Lernzeiten aus Tabelle 3.

der Geschwindigkeit von der gewählten Distanzfunktion α. Erstaunlicherweise ergeben sich nicht so große Unterschiede, wie man erwarten könnte, wenn man bedenkt, daß next4, next8, linear, linear2 und linear4 durch wenige Vergleiche implementiert sind, während gauss, gauss2 und gauss3 eine e-Funktion berechnen müssen und die MasPar keine Hardwareunterstützung für Fließkommaoperationen hat. Die letzten zwei Zeilen der Tabelle zeigen, daß es einen Geschwindigkeitsvorteil bringt, wenn man gegen Ende der Lernphase MaxDist benutzt, um die Fernwirkung auf einen kleinen Bereich zu beschränken. (Dies ist am Anfang der Lernphase nicht sinnvoll, da dann die globale Ordnung der Karte nicht gelernt werden kann. Das Ergebnis wäre ähnlich wie in Kapitel 7.2.)

Distanzfunktion α	Lernzeit	CUPS
gauss	3.593	55663
gauss2	3.798	52659
gauss3	3.937	50800
linear	3.081	64914
linear2	3.323	60187
linear3	3.547	56386
next4	3.042	65746
next8	3.042	65746
gauss + MaxDist = 1	3.233	61862
gauss + MaxDist = 5	3.602	55525

Tabelle 8: Vergleich zwischen Distanzfunktionen. (Parameter: Zweidimensionales Netz, 10 mal 10 Neuronen, Lernrate 0.1, Distanzfaktor 1.0, 1000 Lernschritte)

In der Literatur gibt es leider noch recht wenige Leistungsmessungen von Simulatoren für selbstorganisierende Karten auf Parallelrechner, die zum Vergleich herangezogen werden könnten. Die meisten veröffentlichten Zahlen sind nicht zu einem Vergleich geeignet, da andere Netzarchitekturen und andere Lernregeln benutzt werden.

Hodges, Wu und Wang beschreiben in [8] und [9] zwar ausführlich eine Methode zur Parallelisierung, nennen aber keine Zahlen.

Nur Grajski beschreibt in [18] eine anscheinend ausschließlich für Zeitmessungszwecke implementierte selbstorganisierende Karte, die auf einer MasPar mit nur 2048 PEs implementiert wurde. Sie unterstützt nur zweidimensionale Netze und nur die für diese Maschine ideale Netzgröße von 2048 Neuronen. Dadurch entspricht jedem Prozessor genau ein Neuron und es kann ein einfacherer und schnellerer Algorithmus benutzt werden. Welche weiteren Parameter benutzt wurden und inwieweit das Programm optimiert wurde, wird leider nicht erwähnt. Für zweidimensionale Eingabevektoren nennt Grajski eine Geschwindigkeit von 6.6 MCUPS. Der in dieser Arbeit beschriebene Simulator bietet, obwohl er wesentlich flexibler ist, bei einer vollständigen Ausnutzung des Prozessorarrays eine Geschwindigkeit von 11 MCUPS (siehe Tabelle 6).

Zum Schluß noch ein kurzer Vergleich mit der seriellen Version des Simulators (siehe Tabelle 9). Sie ist bei sehr kleinen Netzen (bis ca. 100 Neuronen) schneller als der parallele Simulator, ab ca. 400 Neuronen ist gewinnt die Parallelität der MasPar die Oberhand gegenüber die schnellere, aber serielle Verarbeitung einer Workstation.

#Neuronen	#Lernzeit		CUPS		Gesamtzeit	
	1	1000	1	1000	1	1000
1x1	nicht meßbar		nicht meßbar		nicht meßbar	
10x10	0.03 - 1.8	7	111 - 6667	28571	ca. 2.5	9
100x100	0.03 - 1.8	738	11111 - 666667	27100	ca. 40	756

Tabelle 9: Messungen für die serielle Version des Simulators bei einem bzw. 1000 Lernzyklen. Die Zahlen sind auf einem multiuser-System während des normalen Betriebs gemessen, wodurch sich bei kleinen Zeiten teilweise erhebliche Schwankungen ergaben. Die Aufgabe des Netzes und die verwendeten Parameter waren dieselbe wie in Tabelle 2.

10 Ausblick

Welche Arbeiten könnten sich anschließen? Zunächst müßte festgestellt werden, warum die MasPar beim Schreiben auf Platte um den Faktor vier langsamer ist als eine Workstation und wie dieser Engpaß gemindert oder ganz umgangen werden kann. Typische Meßwerte für das Schreiben von einem Megabyte Daten sind: 150 Sekunden auf der MasPar mit VAX-Frontend, 55 Sekunden seit dem Austausch des VAX-Frontends gegen eine DEC-Station 5000/200 und 15 Sekunden, wenn dieselben Daten von einer SUN4 geschrieben werden. Damit ist die Ein-/Ausgabe auf Platte durch den Austausch des MasPar-Frontends um den Faktor 2.72 schneller geworden. Möglicherweise kommt die volle Leistungsfähigkeit der MasPar erst mit dem für sie entworfenen Disk-Array zur Entfaltung.

Wenn dieser Engpaß gemindert ist, könnte man sich Gedanken über weitere Optimierungen des Lernalgorithmus machen. Da im Moment bei wenigen Lernschritten das Übertragen der aktuellen Netzkonfiguration um Größenordnungen länger dauert als die eigentliche Berechnung[13], wurde dies noch nicht als vordringliches Problem betrachtet. Es sind allerdings noch einige Optimierunges des Algorithmus möglich.

Außerdem wäre es interessant, ein größeres Problem mit dem Simulator anzugeben. Hier würde sich beispielsweise die Ziffernklassifikation von Dr. Ulrich Kreßel vom Daimler-Benz-Forschungszentrum Ulm anbieten, da diese Daten schon in computerlesbarer Form vorliegen, hiermit schon ausführliche Versuche mit anderen Klassifikationsalgorithmen gemacht worden sind und die 256-dimensionalen Eingabevektoren dieses Problems eine extreme Dimensionsreduktion (und eine große Herausforderung) darstellen würden.

Des weiteren wäre der Versuch interessant, die X-Oberfläche mit dem Kern zu einem Programm zu verschmelzen, das dann nur noch die interne Kommunikation zwischen Frontend und Backend (`blockin()/blockout()` und `copyin()/copyout()`) benutzt. Dabei ergeben sich zwar voraussichtlich Probleme mit X-Windows und es wird eine teilweise Reimplementierung nötig, der zu erwartende Geschwindigkeitsvorteil dürfte das aber wettmachen.

[13]siehe Kapitel 9, Tabelle 9

A Mathematische Grundlagen des Simulators

Die mathematischen Grundlagen selbstorganisierender Karten wurden in Kapitel 2 besprochen. Die konkreten im Simulator benutzten Funktionen sind im Folgenden zusammengefaßt.

Zufälliger Eingabevektor: $\vec{\imath}$

Transformierter Eingabevektor: $t(\vec{\imath})$

Differenz eines Neurons: $\vec{\delta_{\vec{n}}} = t(\vec{\imath}) - \vec{w_{\vec{n}}}$

Norm der Differenz: $\mathrm{norm}(\vec{\delta}) = \|\vec{\delta}\|$

Maximumnorm: $\|\vec{x}\|_{\max} = \max\limits_i |x_i|$

Euklid'sche Norm: $\|\vec{x}\|_{\mathrm{euklid}} = \sqrt{\sum x_i^2}$

allgemeine Norm: $\|\vec{x}\|_{\mathrm{n}} = \sqrt[\mathrm{n}]{\sum |x_i|^{\mathrm{n}}}$

maximal aktiviertes Neuron $\vec{c}$: $\|t(\vec{\imath}) - \vec{w_{\vec{c}}}\| = \min\limits_{\vec{n}} \|t(\vec{\imath}) - \vec{w_{\vec{n}}}\|$

currentFactor: $\phi(\vec{n}, \vec{c}) = \begin{cases} \lambda \cdot \alpha(\|\vec{c} - \vec{n}\|_{\mathrm{euklid}}) & \text{wenn } \|\vec{c} - \vec{n}\|_{\mathrm{euklid}} < MaxDist \\ 0 & \text{sonst} \end{cases}$

AlphaNext4: $\alpha_{\mathrm{next4}}(x) = \begin{cases} 1 & \text{wenn } x^2 < 0.1 \\ 1/10 & \text{wenn } x^2 < 1.8 \\ 0 & \text{sonst} \end{cases}$

AlphaNext8: $\alpha_{\mathrm{next8}}(x) = \begin{cases} 1 & \text{wenn } x^2 < 0.1 \\ 1/10 & \text{wenn } x^2 < 2.2 \\ 0 & \text{sonst} \end{cases}$

AlphaGauss: $\alpha_{\mathrm{gauss}}(x) = \mathrm{e}^{-x^2/\mathrm{DistFactor}}$

AlphaGauss2: $\alpha_{\mathrm{gauss2}}(x) = (1 - (\frac{x}{\mathrm{DistFactor}})^2) \cdot \mathrm{e}^{-(x/\mathrm{DistFactor})^2}$

AlphaGauss3: $\alpha_{\mathrm{gauss3}}(x) = (1 - 3(\frac{x}{\mathrm{DistFactor}})^2 + (\frac{x}{\mathrm{DistFactor}})^4) \cdot \mathrm{e}^{-(x/\mathrm{DistFactor})^2}$

AlphaLinear: $\alpha_{\mathrm{linear}}(x) = \begin{cases} 1 & \text{wenn } x < \mathrm{DistFactor} \\ 0 & \text{sonst} \end{cases}$

AlphaLinear2: $\alpha_{\mathrm{linear2}}(x) = \begin{cases} 1 & \text{wenn } x < \mathrm{DistFactor} \\ -1/3 & \text{wenn } x < 3 \cdot \mathrm{DistFactor} \\ 0 & \text{sonst} \end{cases}$

AlphaLinear3: $\alpha_{\text{linear3}}(x) = \begin{cases} 1 & \text{wenn } x < \text{DistFactor} \\ -1/10 & \text{wenn } x < 3 \cdot \text{DistFactor} \\ 1/100 & \text{wenn } x < 10 \cdot \text{DistFactor} \\ 0 & \text{sonst} \end{cases}$

Gewichtsänderung: $\vec{w}_{\vec{n},\text{neu}} = \vec{w}_{\vec{n},\text{alt}} + \phi(\vec{n}, \vec{c}) \cdot \vec{\delta_{\vec{n}}}$

B Meldungen der X-Oberfläche

Dieser Anhang erklärt alle Meldungen, die im Nachrichtenfenster ausgegeben werden. Manche stellen lediglich eine Information für den Benutzer dar, andere sind Fehlermeldungen, die entweder auf eine inkonsistente Bedienung durch den Benutzer oder meist auf fehlenden Plattenplatz zurückzuführen sind. (Auf schnellen Rechnern werden unwichtige Nachrichten manchmal so schnell durch neue ersetzt, daß der Benutzer nicht einmal bemerkt, daß sie da waren. Dies passiert nicht mit Fehlermeldungen.) Meldungen, die mit drei Punkten (...) enden, fordern den Benutzer zum Warten auf.

`Initializing...`
Die Netzspezifikation wird an den Kern geschickt, die X-Oberfläche und die PEs allokieren den nötigen Speicher, die Gewichte werden initialisiert und die Anfangssituation des Netzes dargestellt.

`Stopped.`
Im kontinuierlichen Simulationsmodus wurde das Stop-Kommando akzeptiert, der Benutzer muß aber noch warten bis der laufende Zyklus beendet ist.

`Reading network description file...`
Eine Netzbeschreibungsdatei, d.h eine Datei mit der Endung `.net` wird gelesen; es ist noch kein Fehler aufgetreten.

`File read.`
Die Netzbeschreibungsdatei wurde erfolgreich geladen, die verschiedenen Parameter (wie z.B. LRateHL) sind aber noch nicht auf Plausibilität geprüft.

`Writing network description file...`
Die Netzbeschreibungsdatei oder Daten werden geschrieben; es ist noch kein Fehler aufgetreten.

`File written.`
Die Datendatei (Endung `.wts`) ist erfolgreich geschrieben.

`Running`
Kontinuierliches Lernen wurde gestartet.

`Waiting for MasPar...`
Warte im Einzelschrittmodus darauf, daß der Kern die spezifizierte Anzahl von Lernschritten gemacht und die Daten auf Platte geschrieben hat.

`Reading data...`
Die neuesten Gewichte werden von der (auf einer Workstation laufenden) X-Oberfläche gelesen, um dargestellt zu werden. (Im Gegensatz zum langwierigen Schreibvorgang der MasPar geht das so schnell, daß der Benutzer die Meldung kaum jemals sehen wird.)

`Picture written.`
Die aktuellen Gewichte sind in derselben graphischen Darstellungsweise wie sie auf dem Bildschirm benutzt als PostScriptdatei (Extension `.ps`) auf Platte gespeichert worden.

`Initialization ok, but no data.`
Für eine neue Netzkonfiguration wurde erfolgreich Speicher allokiert, es wurde aber keine Datei mit den Anfangsgewichten gefunden. Dies liegt meist daran, daß sie nicht (oder nicht komplett) auf Platte geschrieben werden konnten, da der Plattenplatz erschöpft ist.

`Waiting for MasPar to respond...`
Um zu verhindern, daß der Benutzer zuerst Arbeit in das Erstellen einer neuen Netzkonfiguration investiert und dann feststellen muß, daß er die Simulation nicht starten kann, wird sofort nach dem Start der X-Oberfläche getestet, ob ein Kern läuft. Solange das nicht der Fall ist, wird diese Meldung dargestellt; der Benutzer hat vorübergehend nur die Möglichkeit, das Programm zu verlassen.

`ERROR: No filename.`
Der Benutzer möchte eine Datei (Netzbeschreibung, Daten oder Bild) schreiben, hat aber noch keinen Dateinamen angegeben. Dateinamen angeben und noch einmal versuchen.

`ERROR: %s.net does not exist.`
Eine Netzbeschreibungsdatei dieses Namens gibt es nicht. Bitte beachten, daß die Extension `.net` automatisch angehängt wird und nicht angegeben werden soll.

`ERROR: %s.net cannot be opened.`
Eine Netzbeschreibungsdatei dieses Namens existiert zwar, kann aber nicht geöffnet werden. Möglicherweise hat der Benutzer kein Leserecht.

`ERROR: Cannot read or no SNNS format.`
Die Netzbeschreibungsdatei enthält nicht den erwarteten Text `SNNS network definition file`.

`ERROR: Learning function must be 'kohonen'.`
Die Netzbeschreibungsdatei enthält nicht den erwarteten Text:
`learning function: kohonen`.

`ERROR: Cannot read or 'kohonen section' not found.`
In der Netzbeschreibungsdatei muß es eine kohonen section geben, da dort relevanten Informationen für die Konfiguration des Kohonen-Netzes enthalten sind.

`ERROR: This version uses '%s' as its transformation, the file uses '%s'.`
Wenn eine Netzbeschreibungsdatei geladen wird, müssen sie und die X-Oberfläche dieselbe Transformation benutzen.

`ERROR: This version uses a %d-dimensional network, the file uses a %d-dimensional network.`
Wenn eine Netzbeschreibungsdatei geladen wird, müssen sie und die X-Oberfläche dieselbe Netzdimensionalität benutzen.

`ERROR: The line above cannot be processed.`
Die Netzbeschreibungsdatei enthält eine Zeile, die nicht mit einem definierten Schlüsselwort beginnt. Die Zeile, die diesen Fehler ausgelöst hat, wird im Statusfenster über dem Nachrichtenfenster angezeigt.

`ERROR: Cannot write the file.`
Die Netzbeschreibungsdatei bzw. die Daten konnten nicht oder nicht vollständig geschrieben werden. Wahrscheinlich ist kein Platz mehr auf der Platte.

`ERROR: Cannot open the file.`
Die Netzbeschreibungsdatei bzw. die Datei für die Daten konnte nicht geöffnet werden. Möglicherweise ist der angegebene Dateiname kein erlaubter Dateiname in UNIX oder es gibt schon eine schreibgeschützte Datei dieses Namens.

`ERROR: Network not allocated.`
Das Netz kann nicht trainiert werden, da die Datenstrukturen nicht allokiert sind. Dies wäre ein Programm- oder Systemfehler.

`ERROR: Cannot execute 'ReInit'.`
Die zum Anstoßen des Lernens nötigen Daten wie Anzahl der Schritte und eventuell geänderte Parameter konnten nicht an den Kern übertragen werden. Grund: Wahrscheinlich volle Platte.

`ERROR: 'ReInit' not accepted.`
Eine Parameteränderung beim Lernen wurde vom Kern zurückgewiesen.

`ERROR: Cannot read data.`
Die Datei, die die aktuellen Gewichte enthält kann nicht gelesen werden oder hat ein falsches Format.

`ERROR: No data.`
Das Bild kann nicht gespeichert werden, da die Datenstrukturen nicht allokiert sind. Dies wäre ein Programm- oder Systemfehler.

`ERROR: Cannot write the picture.`
Beim Speichern der Postscriptdatei ist ein Fehler aufgetreten. Wahrscheinlich ist kein Platz mehr auf der Platte.

`ERROR: Cannot write 'init'.`
Die Netzkonfiguration kann nicht an das Netz übertragen werden. Grund: möglicherweise volle Platte.

`ERROR: Network description rejected.`
Der Kern kommt mit der Netzbeschreibung nicht zurecht. Entweder war sie wegen voller Platte unvollständig oder sie hat gegen andere Bedingungen verstoßen. Nähere Informationen werden direkt vom Kern nach `stdout` ausgegeben.

`ERROR: Cannot allocate data.`
Der Platz für die internen Datenstrukturen für ein so großes Netz steht nicht zur Verfügung. Eigentlich ein Euphemismus für `out of memory`.

C Meldungen des Kerns

Wenn die X-Oberfläche einen Fehler meldet, der vom Kern verursacht wurde, gibt dieser eine zusätzliche Meldung auf `stdout` aus, die den Fehler genauer beschreibt. Darüberhinaus werden auch noch andere Kommentare vom Kern ausgegeben.

`I got the DPU.`
Beim Aufruf des Kerns kommt der Prozeß zunächst in eine Warteschlange. Sobald der Kern die DPU bekommt (und damit läuft), wird diese Meldung ausgegeben. Je nach Anzahl und Konfiguration anderer laufender Prozesse kann das aufgrund der Architektur der MasPar so lange dauern, bis einer der laufenden Prozesse terminiert. Die Warteschlange kann mit dem Befehl `mpq` inspiziert werden.

`Terminated normally.`
Diese Meldung wird bei jedem regulären Ende des Kerns ausgegeben. Dies ist hilfreich, wenn er als Hintergrundprozeß gestartet wurde.

`For a one-dimensional network a maximum of %d neurons are supported.`
Die Anzahl der PEs begrenzt bei eindimensionalen Netzen die Anzahl der Neuronen.

`The base area of the block has to be smaller than the processor array (%dx%d).`
Die Grundfläche des durch das dreidimensionale Neuronennetz gebildeten Quaders, die von `Nx` und `Ny` aufgespannt wird, muß in das verfügbare Prozessorarray passen. Nötigenfalls kann man die Werte von `Nx`, `Ny` und `Nz` so vertauschen, daß diese Bedingung erfüllt ist. Dann muß man allerdings daran denken, die Transformation anzupassen.

`%d network dimensions are not supported.`
Es werden nur ein-, zwei- und dreidimensionale Netze unterstützt, da höhere Netzdimensionen sich auf diese Weise nicht darstellen lassen.

`Can't read 'init'.`
Die Datei `init` ist Teil der Kommunikation. Wenn sie nicht gelesen werden kann, ist wahrscheinlich eine volle Platte (oder etwas Schlimmeres) schuld.

`ID mismatch (transform:%s;retransform:%s).`
Die Transformations-ID und die Retransformations-ID stimmen nicht überein. Wahrscheinlich wurde eines der Programme verwechselt. Näheres siehe Kapitel 5.

`InputDimension mismatch (transform:%d;retransform:%d).`
Die InputDimension der Transformation und die der Retransformation widersprechen sich. Näheres siehe Kapitel 5.

`Out of memory.`
Zusammen mit `syntax error` wohl einer der „Lieblingsfehler“ eines jeden Programmierers. Hier bedeutet er, daß der Speicher der PEs nicht ausreicht, um das konfigurierte Netz zu allokieren. Der Speicherverbrauch hängt von der Anzahl der Neuronen und von der

Dimensionalität der Eingabevektoren ab.

Weitere Meldungen des Kerns wie `touch: cannot stat` (Kollision in der Kommunikation) können ignoriert werden.

D Fileformate

Eine kurze Erklärung zur Syntax der BNF-Grammatiken:

- Nichtterminale stehen in spitzen Klammern, z.B. `DATA`
- `EOL` steht für End-Of-Line (Zeilenende)
- `EOF` steht für Dateiende
- `<DATE>` steht für ein Datum in dem von der C-Funktion `ctime()` gelieferten Format. Man beachte, daß diese schon ein EOL enthält.
- `<INT>` steht für einen Integer, alle Numerierungen gehen ab 0
- `<FLOAT>` steht für einen Fließkommawert
- `<NAME>` steht für einen Namen, der auch als UNIX-Filename benutzbar ist.
- Das Zeichen `|` steht nicht für das Metazeichen *oder*, sondern ist ein Terminal.

D.1 Das Format der Netzbeschreibungsdatei

```
<FILE>         -->  <SNNS> <GEN> <NETNAME> <SOURCEFILES> <SECTIONS> EOF
<SNNS>         -->  SNNS network definitionfile V1.4-3D EOL
<GEN>          -->  generated at <DATE> EOL
<NETNAME>      -->  network name : <NAME> EOL
<SOURCEFILES>  -->  source files : EOL
<SECTIONS>     -->  <BASESECTION> <UNITSECTION> <3DSECTION> <KOSECTION>
<BASESECTION>  -->  <UNITS> <CONNS> <TYPES> <SITES> <LFUNCTION> <UF> EOL
<UNITS>        -->  no. of units : <INT> EOL
<CONNS>        -->  no. of connections : <INT> EOL
<TYPES>        -->  no. of unit types : 0 EOL
<SITES>        -->  no. of site types : 0 EOL
<LFUNCTION>    -->  learning function : kohonen EOL
<UF>           -->  update function : EOL
<UNITSECTION>  -->  unit definition section : EOL EOL <UNITTABLE> EOL
<UNITTABLE>    -->  <UTHEAD> <UTSEP> <UTCONT> <UTSEP> EOL
<UTHEAD>       -->
     no.  |typeName|unitName|act|bias|st |position           |act func|out func|sites EOL
<UTSEP>        -->
   -------|--------|--------|---|----|---|-------------------|--------|--------|----- EOL
<UTCONT>       -->
    <INT> |        |        |   |    | i | <INT>,<INT>,<INT> |        |        | EOL
<3DSECTION>    -->  <3DHEAD> <3DSEP> <3DCONT> EOL
<3DHEAD>       -->   delta x | delta y |    z EOL
<3DSEP>        -->  ---------|---------|--------- EOL
<3DCONT>       -->  <INT> | <INT> | <INT> EOL
<KOSECTION>    -->  <KOHEAD> <TRANS> <NX> <NY> <NZ> <NETDIM> <INPUTDIM>
                    <NPOWER> <LRATE> <ALPHA> <DECFACTOR> <RSC> <INI>
<KOHEAD>       -->  kohonen section : EOL EOL
```

```
<TRANS>        -->  Transformation: <NAME> EOL
<NX>           -->  Nx: <INT> EOL
<NY>           -->  Ny: <INT> EOL
<NZ>           -->  Nz: <INT> EOL
<NETDIM>       -->  NetDimension: <INT> EOL
<INPUTDIM>     -->  InputDimension: <INT> EOL
<NPOWER>       -->  NormPower: <INT> EOL
<LRATE>        -->  <LRSTART> <LREND> <LRHL> <LRDEC>
<LRSTART>      -->  LRateStart: <FLOAT> EOL
<LREND>        -->  LRateEnd: <FLOAT> EOL
<LRHL>         -->  LRateHL: <INT> EOL
<LRDEC>        -->  DecrementLRate: <DEC> EOL
<DEC>          -->  linear
<DEC>          -->  exponential
<ALPHA>        -->  Alpha: <AMODE> EOL
<AMODE>        -->  gauss
<AMODE>        -->  gauss2
<AMODE>        -->  gauss3
<AMODE>        -->  linear
<AMODE>        -->  linear2
<AMODE>        -->  linear3
<AMODE>        -->  next4
<AMODE>        -->  next8
<DECFACTOR>    -->  <DFSTART> <DFEND> <DFHL> <DFDEC> <MAXDIST>
<DFSTART>      -->  DistFactorStart: <FLOAT> EOL
<DFEND>        -->  DistFactorEnd: <FLOAT> EOL
<DFHL>         -->  DistFactorHL: <INT> EOL
<DFDEC>        -->  DecrementDistFactor: <DEC> EOL
<MAXDIST>      -->  MaxDist: <FLOAT> EOL
<RSC>          -->  RandomStartConfig: <RSCMODE> EOL
<RSCMODE>      -->  dot
<RSCMODE>      -->  small
<RSCMODE>      -->  large
<RSCMODE>      -->  corner
<INI>          -->  Initialized: <YN> EOL
<YN>           -->  no
```

Dieses Format kann von der gegenwärtigen SNNS-Version gelesen werden, wenn man von Hand (per Editor) folgende Änderungen macht:

1. `<LFUNCTION>` muß geändert werden auf `learning function: EOL`
2. Die komplette `<KOSECTION>` muß gelöscht werden.

D.2 Das Format der Daten

```
<FILE>          -->  <NX> <NY> <NZ> <NETDIM> <INPUTDIM> <DATASECTION> EOF
<NX>            -->  Nx: <INT> EOL
<NY>            -->  Ny: <INT> EOL
<NZ>            -->  Nz: <INT> EOL
<NETDIM>        -->  NetDimension: <INT> EOL
<INPUTDIM>      -->  InputDimension: <INT> EOL
<DATASECTION>   -->  Data: EOL <DATALIST>
<DATALIST>      -->  <DATA>
<DATALIST>      -->  <DATA> <DATALIST>
<DATA>          -->  <INT> <INT> <INT> <INT> <FLOAT> EOL
```

Bei `<DATA>` sind die ersten drei Integers die Position eines Neurons im Netz, der vierte Integer ist die Nummer eines Gewichtsvektorelements und die Fließkommazahl ist der Wert des Gewichts mit dieser Nummer dieses Neurons. Beispiel: Die Zeile
`6 3 0 1 0.83`
bedeutet, daß das Gewicht mit Nummer 1 des Neurons mit den Koordinaten (6;3;0) den Wert 0.83 hat. Darüberhinaus kann das File beliebige mit `#` beginnende Zeilen enthalten, die als Kommentare behandelt werden.

Literatur

[1] T. Kohonen: *Self-Organized Formation of Topologically Correct Feature Maps*, Biological Cybernetics 43, pp. 59–69, 1982

[2] T. Kohonen: *Analysis of a Simple Self-Organizing Process*, Biological Cybernetics 44, pp. 135–140, 1982

[3] T. Kohonen: *Self-Organization and Associative Memory*, Springer Verlag, 2nd Edition, 1984

[4] T. Kohonen: *Self-Organization and Associative Memory*, Springer Verlag, 3rd Edition, 1989

[5] H. Ritter, T. Kohonen: *Learning "Semantotopic Maps" from Context*, Proceedings of the IJCNN, pp. I-23–I-26, 1990

[6] P. G. Schyns: *Expertise acquisition through concepts refinements in a self-organizing architecture*, ebd., pp. I-236–I-239

[7] R. E. Hodges, Ch.-H. Wu: *A Method to Establish an Autonomous Self-Organizing Feature Map*, ebd., pp. I-517–I-520

[8] R. E. Hodges, Ch.-H. Wu, C.-J. Wang: *Parallelizing the Self-Organizing Feature Map on Multiprocessor Systems*, ebd., pp. II-141–II-144

[9] Ch.-H. Wu, R. E. Hodges, C.-J. Wang: *Parallelizing the Self-Organizing Feature Map on Multiprocessor Systems*, Parallel Computing 17, pp. 821-832, 1991 (Dies ist eine Umarbeitung und Erweiterung von [8])

[10] T. Kohonen: *Some Practical Aspects of the Self-Organizing Map*, ebd., pp. II-253–II-256

[11] J. K. Samarabandu, O. G. Jakubowicz: *Prinziples of Sequential Feature Maps in Multilevel Problems*, ebd., pp. II-683–II-686

[12] R. Hecht-Nielsen: *Neurocomputing*, Addison-Wesley, 1990

[13] H. Ritter, T. Martinetz, K. Schulten: *Neuronale Netze*, Addison–Wesley, 1990

[14] K. Obermayer, H. Ritter, K. Schulten: *Large-Scale Simulation of a Self-Organizing Neural Network: Formation of a Somatotopic Map*, In: R. Eckmiller, G. Hartmann, G. Hauske (eds.): *Parallel Processing in Neural Systems and Computers*, North-Holland, pp. 71–74, 1990

[15] T. Kohonen: *Speech recognition based on topology-preserving neural maps*, In: I. Aleksander (ed.): *Neural Computing Architectures: The Design of Brain-Like Machines*, North Oxford Academic, pp. 26–40, 1989

[16] G. Tattersall: *Neural Map Applications*, ebd., pp. 41–73

[17] A. Singer: *Implementations of Artificial Neural Networks on the Connection Machine*, Thinking Machines Corporation Technical Report RL90-2, 1990

[18] K. A. Grajski: *Neurocomputing Using the MasPar MP-1*, Ford Aerospace Corporation, Advanced Development Department, Technical Report No. 90-010, 1990

[19] Y.-H. Pao: *Adaptive Pattern Recognition and Neural Networks*, Addison-Wesley, 1989

[20] H. Ritter, T. Martinetz, K. Schulten: *Topology-Conserving Maps for Learning Visuo-Motor-Coordination*, Neural Networks Vol. 2, pp. 159–168, 1989

[21] H. Mallot, W. v. Seelen, F. Giannakopoulos: *Neural Mapping and Space-Variant Image Processing*, Neural Networks Vol. 3, pp. 245–263, 1990

[22] V. Cherkassky, H. Lari-Najafi: *Constrained Topological Mapping for Nonparametric Regression Analysis*, Neural Networks Vol. 4, pp. 27–40, 1991

[23] T. Kohonen: *A Simple Paradigm for the Self-Organized Formation of Structured Feature Maps*, In: S. Amari, M.A. Arbib: *Competition and Cooperation in Neural Nets*, Lecture Notes in Biomathematics Nr. 45, Springer, pp. 248–266, 1982

[24] D. E. Rumelhart, D. Zipser: *Feature Discovery by Competitive Learning*, In: D.E. Rumelhart, J.L. McClelland: *Parallel Distributed Processing*, Volume 1, A Bradford Book, MIT Press, pp. 151–193, 1986

[25] T. J. Sejnowski: *Open Questions About Computation in Cerebral Cortex*, In: J.L. McClelland, D.E. Rumelhart: *Parallel Distributed Processing*, Volume 2, A Bradford Book, MIT Press, pp. 372–389, 1986

[26] B. Fritzke, P. Wilke: *Flexmap - A Neural Network for the Traveling Salesman Problem with Linear Time and Space Complexity*, eingereicht für IJCNN 91 (momentan erhältlich über anonymous ftp von cheops.cis.ohio-state.edu)

[27] P. Christy: *Virtual Processors Considered Harmfull*, eingereicht für: The 6th Distributed Memory Computing Conference 1991

[28] M. Witbrock, M. Zagha: *An Implementation of Back-Propagation Learning on GF11, a Large SIMD Parallel Computer*, Carnegie Mellon University, School of Computer Science, report CMU-CS-89-208, 1989 (momentan erhältlich über anonymous ftp von cheops.cis.ohio-state.edu)

[29] *MasPar Parallel Application Language*, User Guide, MasPar Computer Corporation, 1990

[30] *MasPar Parallel Application Language*, Reference Guide, MasPar Computer Corporation, 1990

[31] *MasPar Programming Environment*, User Guide, MasPar Computer Corporation, 1990

[32] *MasPar Commands*, Reference Manual, MasPar Computer Corporation, 1990