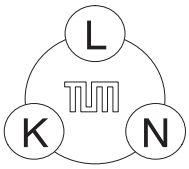
Technische Universität München



Lehrstuhl für Kommunikationsnetze

Prof. Dr.-Ing. Jörg Eberspächer

Lehrstuhl für Theoretische Informatik und Künstliche Intelligenz

Prof. Dr. Dr. h.c. Wilfried Brauer

Diplomarbeit

Parallelimplementierung von rekurrenten selbstorganisierenden Karten

Diplomand:	Holger Arndt
Matrikelnummer:	2025424
Anschrift:	Gabelsbergerstr. 34 80333 München
Betreuer:	Christian Schwingenschlögl Volker Baier
Beginn:	01. Dezember 2002
Abgabe:	31. Mai 2003

Danksagung

Für die hervorragende Unterstützung bei der Erstellung meiner Diplomarbeit bedanke ich mich bei meinen Betreuern Volker Baier und Christian Schwingenschlögl.

Des Weiteren danke ich Herrn Prof. Dr. Dr. h.c. Brauer und Herrn Prof. Dr.-Ing. Eberspächer für die Begutachtung der Arbeit.

Mein besonderer Dank gilt Herrn Dr.-Ing. Maier, der mich bei der Suche nach einem Co-Betreuer am Lehrstuhl für Kommunikationsnetze unterstützte.

Zusammenfassung

Selbstorganisierende Karten sind Modelle neuronaler Netzwerke, die die Fähigkeit des Gehirns zur Selbstorganisation nachahmen und ihre räumliche Struktur selbst bestimmen. Allerdings erfordern große Karten in der Regel sehr viel Rechenleistung. In einem Rechnercluster lässt sich die Aufgabe auf mehrere Workstations verteilen und parallel bearbeiten, was in einer Verkürzung der Verarbeitungszeit resultiert. Die vorliegende Arbeit beschreibt eine parallele Implementierung selbstorganisierender rekurrenter Karten, bei der die einzelnen Prozesse mittels Message-Passing-Software kommunizieren, wodurch eine dem Gehirn ähnliche verteilte Struktur realisiert wird.

Abstract

Self-Organizing Maps are a type of Neural Network to model the brain's ability to self-organize and preserve the topological relationship of the provided data. However, as large models result in high computational complexity, it makes sense to distribute the task among several computers in a cluster. This thesis proposes a parallel implementation of Recurrent Self-Organizing Maps using Message Passing Software for inter-process communication. As a result, computing time can be reduced and a brain-like distributed structure is obtained.

Inhaltsverzeichnis

Kapitel 1

Einführung

Künstliche neuronale Netze (KNN) sind eine Modellarchitektur, die dem menschlichen Gehirn nachempfunden ist. Sie bestehen typischerweise aus einer Vielzahl einfacher Rechenelemente (*Neuronen*), die zu einem komplexen Netzwerk verbunden sind. Die Stärke dieser Verbindungen kann dabei variiert werden, so dass sich das Netz an unterschiedliche Eingabesignale anpassen und darauf adäquat reagieren kann. Künstliche neuronale Netze stellen also einen völlig anderen Ansatz dar als herkömmliche *von Neumann* Rechner, bei denen Programmbefehle von einer zentralen CPU nacheinander abgearbeitet werden müssen. Obwohl sich die Rechenleistung und Speicherkapazität heute verwendeter Computer vervielfacht hat, ist die Struktur jedoch im Wesentlichen gleich geblieben. Im Unterschied dazu gibt es bei den künstlichen neuronalen Netzen keine zentrale Steuereinheit. Die einzelnen Neuronen arbeiten vielmehr in einem hochgradig parallelen Netzwerk zusammen. Die "Intelligenz" liegt dabei in der Vernetzung und gegenseitigen Beeinflussung der einzelnen Neuronen untereinander. Vermutlich ist auch gerade dieser Unterschied und die Ähnlichkeit zu biologischen Nervenzellen dafür verantwortlich, dass an diese Technik einst sehr große Erwartungen geknüpft wurden, die jedoch nur teilweise erfüllt werden konnten. Dennoch stellen künstliche neuronale Netze ein interessantes Forschungsgebiet dar, das bereits eine Vielzahl nützlicher Anwendungen hervorgebracht hat.

Im Rahmen dieser Arbeit werden *rekurrenten selbstorganisierenden Karten* (engl. *Recurrent Self-Organizing Maps*, RSOM) untersucht. Dazu wird eine Simulationsumgebung entwickelt, die auf einem Rechnercluster aus handelsüblichen PCs oder Workstations lauffähig ist. Dadurch dass unabhängige Rechenschritte parallel von mehreren Rechnern ausgeführt werden können, lässt sich die Verarbeitungsgeschwindigkeit erhöhen.

In Kapitel 2 wird zunächst die historische Entwicklung künstlicher neuronaler Netze beschrieben. Danach werden die wichtigsten Modelle und Lernparadigmen vorgestellt. Der Lernalgorithmus für rekurrente selbstorganisierende Karten wird ausführlich dargelegt, da der weitere Teil der Arbeit darauf aufbaut.

Kapitel 3 beschäftigt sich mit der Entwicklung einer Software, die eine rekurrente selbstorganisierende Karte parallel auf einem Rechnercluster simulieren kann. Für die Kommunikation der Prozesse untereinander wird dabei eine *Message Passing Software* verwendet, für die bereits fertige Implementierungen existieren. Um die Gesamtleistung des Systems auch bei Verwendung unterschiedlich leistungsstarker Rechner oder variierender Auslastung zu maximieren, wird ein Verfahren vorgestellt, das die Teilaufgaben entsprechend der Verfügbarkeit von Ressourcen dynamisch verteilt.

Die fertige Software wird in Kapitel 4 anhand unterschiedlicher Testdatensätze evaluiert. Zunächst wird die Leistungsfähigkeit der (rekurrenten) selbstorganiserenden Karte bei einer klassischen Klassifizierungsaufgabe bewertet. Im Verlauf des Kapitels werden auch die Fähigkeiten zur Prädiktion und Klassifikation temporaler Daten untersucht. Neben einer qualitativen Bewertung der Leistung wird außerdem auf den durch die Verwendung eines Rechnerclusters erreichbaren Geschwindigkeitvorteil eingegangen.

Das letzte Kapitel fasst die Ergebnisse zusammen und diskutiert bleibende Probleme, die einer weiteren Untersuchung bedürfen. Im Ausblick werden Lösungsmöglichkeiten aufgezeigt, die sich in zukünftigen Projekten erproben ließen.

Verzeichnisse der verwendeten mathematischen Symbole, Abkürzungen, Tabellen und Abbildungen sind im Anhang enthalten.

Kapitel 2

Künstliche neuronale Netze

2.1 Historischer Überblick

Die Anfänge *künstlicher neuronaler Netze* (KNN) liegen im Jahr 1943 als der Neurophysiologe Warren McCulloch und der Mathematiker Walter Pitts ein Paper über die vermutliche Arbeitsweise von biologischen Nervenzellen veröffentlichten. Ein Neuron sollte sich ihnen zufolge wie ein Schwellwertelement verhalten, das zwei mögliche Zustände kennt. [RMS91, Koh01]

Mit seinem Buch "*The Organization of Behavior*" knüpfte Donald Hebb im Jahr 1949 an diese Arbeit an. Seine wesentliche Erkenntnis bestand darin, dass neuronale Verkünpfungen umso stärker sind, je häufiger sie aktiv werden. Wenn also zwei Neuronen gleichzeitig *feuern*, wird die Verbindung zwischen ihnen verstärkt. Dieses Konzept wurde nach ihm die *hebb'sche Regel* gennant. [RMS91]

1958 veröffentlichte der Neurobiologe Frank Rosenblatt seine Arbeit über das *Perceptron*. Als Vorbild dienten ihm Nervenzellen aus dem Auge. Er erkannte, dass bereits in der Retina eine Vorverarbeitung stattfindet, mit der sich sogar einzelne Merkmale unterscheiden lassen. Sein Netzwerkmodell enthielt ursprünglich lediglich eine Schicht, wurde aber später als *Multilayer-Perceptron* (MLP) mit mehreren Schichten verwendet. Allerdings ließ sich dieses mehrschichtige Modell nur schwer trainieren, da ein passender Lernalgorithmus fehlte. [RMS91, Hay94]

Einen herben Rückschlag erlitt die Forschung im Jahr 1969 als Marvin Minsky und Seymour Papert demonstrierten, dass das ursprüngliche Perceptron einfache Aufgaben wie das XOR-Problem nicht lösen konnte und damit zur Klassifizierung von komplexen Mustern ungeeignet war. So lange lediglich lineare Übertragungsfunktionen verwendet wurden, konnte auch mit mehreren Schichten keine Verbesserung erreicht werden. [RMS91, Hay94]

1972 stellte Teuvo Kohonen einen Algorithmus zur Vektorquantisierung vor. Bei seiner *Learning Vector Quantization* (LVQ) wird eine Trainingsmenge auf eine geringere Anzahl an Referenzvektoren abgebildet, so dass sich der benötigte Speicherplatz reduzieren lässt. [Koh01, Fri98]

In seiner Doktorarbeit entwickelte P. Werbos 1974 einen Algorithmus, der das Trainieren mehrschichtiger Perceptron-Netze ermöglichte. Der Durchbruch folgte jedoch erst später als ein identisches Lernverfahren als *Backpropagation Algorithmus* bekannt wurde. [RMS91, Koh01]

1982 greift Teuvo Kohonen seinen LVQ-Algorithmus in abgewandelter Form wieder auf und veröffentlicht ein Paper über *Self-Organizing Maps* (SOM). Diese Merkmalskarten können ihre Topologie selbständig anpassen und werden oft auch Kohonen-Karten oder Feature Maps genannt. [Koh01, Fri98, Kan94, Koh90]

1986 veröffentlichten David E. Rumelhart und James L. McClelland den *Backpropagation* (BP) Algorithmus zum Trainieren von Multilayer-Perceptrons. Im Nachhinein stellte sich heraus, dass derselbe Algorithmus bereits viel früher von P. Werbos publiziert wurde. Es war jedoch ihre Arbeit, die dem Multilayer-Perceptron und neuronalen Netzen im Allgemeinen zum Durchbruch verhalf. [RMS91, Hay94]

2.2 Trainingsdaten

Nachdem die wichtigsten Modelle bereits kurz vorgestellt wurden, soll nun ihre Arbeitsweise genauer erläutert werden. Künstliche neuronale Netze werden häufig dann eingesetzt, wenn sich ein bestimmter Sachverhalt mathematisch nicht einwandfrei formulieren lässt, weil er entweder zu komplex ist, oder weil kein genaues Wissen über ihn existiert. Ihre Stärke liegt darin, dass sie die Aufgabe anhand von Beispielen *erlernen* können ohne dass dem Netz dabei explizite Regeln vorgeben werden müssten. Einzige Voraussetzung ist, dass genügend Trainingsdaten vorhanden sind.

Die Trainingsdaten bestehen in der Regel aus einzelnen *Merkmalen* $\xi_1, \xi_2, ..., \xi_n \in \mathbb{R}$, die in einem Vektor $\boldsymbol{\xi} = [\xi_1, \xi_2, ...\xi_n]^T \in \mathbb{R}^n$ zusammengefasst und dem Netz präsentiert werden. Solch ein *Merkmalsvektor* (auch *Muster* genannt) für eine Wettervorhersage würde beispielsweise die aktuellen Messwerte für Temperatur, Luftdruck und Windgeschwindigkeit enthalten. Aber erst eine Wetterbeobachtung über einen längeren Zeitraum aus vielen Einzelmessungen $\boldsymbol{\xi}_{(i)}$ macht eine Wettervorhersage möglich. Man spricht von einer *Trainingsmenge* $\mathcal{T} = \{\boldsymbol{\xi}_{(1)}, \boldsymbol{\xi}_{(2)}, ..., \boldsymbol{\xi}_{(z)}\}$ mit $\boldsymbol{\xi}_{(i)} \in \mathbb{R}^n$, die in der Regel begrenzt ist. Anders sieht es aus, wenn eine kontinuierliche Wahrscheinlichkeitsdichtefunktion $p(\boldsymbol{\xi}),\ \boldsymbol{\xi} \in \mathbb{R}^n$ bekannt ist, der die Daten gehorchen. In diesem Fall können beliebig viele Trainingsvektoren $\boldsymbol{\xi}_{(i)}$ errechnet und dem Netz präsentiert werden.

Oft ist zu jedem Eingabevektor $\boldsymbol{\xi}_{(i)}$ auch eine zugehörige Bezeichnung, ein sogenanntes

Label, bekannt. Man spricht in diesem Fall von *gelabelten Daten*. Tragen mehrere Trainingsvektoren dasselbe Label, so gehören sie zu derselben *Klasse* K. Um das Beispiel von oben wieder aufzugreifen, könnte man die Wetterdaten labeln, indem man jeden Datensatz der Klasse K_1 : "schönes Wetter" oder K_2 : "schlechtes Wetter" zuordnet.

Soll ein Netzwerk trainiert werden, sind dazu in der Regel mehrere 100.000 Lernschritte notwendig. Wenn nicht so viele Trainingsbeispiele vorhanden sind, müssen die verfügbaren iterativ wiederverwendet werden. [Koh01, Fri98]

2.3 Vorverarbeitung

In der Regel ist es nicht sinnvoll, die zur Verfügung stehenden Daten unverändert für das Training eines künstlichen neuronalen Netzes zu verwenden. Vielmehr ist es oft erforderlich, eine geeignete *Vorverarbeitung* durchzuführen. Diese richtet sich jedoch sehr stark nach der Beschaffenheit der Daten und es nicht immer einfach, ein geeignetes Verfahren auszuwählen. Grundsätzlich stehen verschiedene Instrumente zur Verfügung.

2.3.1 Skalierung

Da oft unterschiedliche Maßeinheiten in einem Vektor zusammengefasst werden, muss gewährleistet sein, dass nicht die Komponente mit den größten Zahlen dominiert. Abhilfe schafft eine *Skalierung* der einzelnen Komponenten:

$$\xi_i' = \frac{\xi_i - \mu_i}{\sigma_i}, \tag{2.1}$$

wobei μ_i der Mittelwert und σ_i die Standardabweichung der i-ten Komponente aller Vektoren ist. Vektorbasierte neuronale Netze sind sehr empfindlich bezüglich der Skalierung der Daten, da sie Distanzen in einem vorgegebenen Vektorraum messen. Andere Netzwerkarchitekturen wie mehrschichtige Perceptrons besitzen jedoch die Fähigkeit, durch Wahl geeigneter Gewichte selbst eine Skalierung vorzunehmen.

2.3.2 Ausblenden von Komponenten

Häufig sind nicht alle Komponenten für die spätere Verarbeitung wichtig. Da sich der Berechnungsaufwand verringern lässt, wenn weniger Komponenten berücksichtigt werden müssen, ist daher eine Dimensionsreduktion von $\mathcal{T}_{orig} = \{\boldsymbol{\xi}_{(1)}, \boldsymbol{\xi}_{(2)}, ..., \boldsymbol{\xi}_{(z)}\}$ mit $\boldsymbol{\xi}_{(i)} \in \mathbb{R}^n$ auf $\mathcal{T}_{neu} = \{\boldsymbol{\xi}_{(1),neu}, \boldsymbol{\xi}_{(2),neu}, ..., \boldsymbol{\xi}_{(z),neu}\}$, $\boldsymbol{\xi}_{(i),neu} \in \mathbb{R}^m$ mit $m < n$ dringend anzustreben. Ziel ist es, unwichtige Informationen auszublenden ohne dabei wesentliche Daten zu verlieren. Allerdings lässt sich nicht immer auf den ersten Blick erkennen, welche Komponenten eines Vektors informationstragend sind und welche nicht.

2.3.3 Merkmalsextraktion

Auch durch *Merkmalsextraktion* lässt sich eine Reduktion der Dimension erzielen. Dazu werden lineare oder nichtlineare Kombinationen einzelner Komponenten gebildet, indem z.B. Gradientenfilter eingesetzt werden. Häufig wird auch eine *Hauptkomponentenanalyse* (engl. *Principal Component Analysis*, PCA) durchgeführt, bei der die orthogonalen Raumrichtungen berechnet werden, in denen die Daten die größten Varianzen aufweisen. Diese Richtungen werden dann als *Hauptkomponenten* bezeichnet. Allerdings werden bei dieser Methode evtl. vorhandene Label ignoriert, da nur die reinen Daten selbst verarbeitet werden können.

Die Vorverarbeitung ist entscheidend für den Erfolg der weiteren Verarbeitungsschritte. Hier gemachte Fehler können oft nicht mehr ausgeglichen werden. [Koh01, Fri98]

2.4 Lernparadigmen

Bei künstlichen neuronalen Netzen unterscheidet man eine Vielzahl von Netztypen und Lernverfahren. Sie variieren in der Art und Weise, wie sie lernen bzw. trainiert werden.

2.4.1 Überwachtes Lernen

Von *überwachtem Lernen* spricht man, wenn ein "Lehrer" vorhanden ist; d.h. zu jedem Eingabevektor $\boldsymbol{\xi}_{(i)}$ ist ein zugehöriger Ausgabewert oder *Zielvektor* $\boldsymbol{\zeta}_{(i)}$ bekannt. Dieser kennzeichnet meistens die Klasse K, zu der ein Trainingsvektor gehört. Sollen also die Wetterdaten $\mathcal{T} = \{\boldsymbol{\xi}_{(1)}, \boldsymbol{\xi}_{(2)}, ..., \boldsymbol{\xi}_{(z)}\},\ \boldsymbol{\xi}_{(i)} \in \mathbb{R}^n$ den Klassen $\mathcal{K} = \{K_1, K_2\}$ zugeordnet werden, könnte man beispielsweise der ersten Klasse den Ausgabevektor $\boldsymbol{\zeta}_{(1)} = [1, 0]^T$ und der zweiten $\boldsymbol{\zeta}_{(2)} = [0, 1]^T$ zuordnen. Es wird also eine diskrete Abbildung der Eingabevektoren auf eine endliche Zahl von Zielvektoren durchgeführt, man spricht von *Musterklassifikation*.

Eine andere Aufgabe besteht darin, eine nicht-diskrete Abbildung $f(\boldsymbol{\xi})$, die den Trainingsdaten zugrunde liegt, zu erlernen. Dies wird *Funktionsapproximation* genannt. Die Schwierigkeit besteht darin, dass die Daten häufig mit statistischen Fehlern (*Rauschen*) behaftet sind, die selbstverständlich nicht mitgelernt werden sollen. Es stehen also lediglich die Datenvektoren $\boldsymbol{\xi}_{(i)} = \boldsymbol{\xi}_{(i),orig} + \boldsymbol{\varepsilon}(t)$ zum Training zur Verfügung, wobei $\boldsymbol{\varepsilon}(t)$ ein mehrdimensionales zeitlich veränderliches Rauschsignal bezeichnet. Hat das System das Rauschen nicht herausgefiltert sondern mitgelernt, spricht man von *Überanpassung* (engl. *overfitting*).

2.4.2 Unüberwachtes Lernen

Anders sieht es aus, wenn kein Wissen über die zu unterscheidenden Klassen vorhanden ist bzw. kein Lehrer zur Verfügung steht. Auch beim *unüberwachten Lernen* lassen sich mehrere Arten unterscheiden:

- *Clustering*: Die Daten sollen in geeigneten Gruppen oder Klassen K_i entsprechend ihrer Ähnlichkeit zusammengefasst werden. Als Ähnlichkeitmaß zweier Datenvektoren wird häufig der *euklid'sche Abstand*

$$d_E(\boldsymbol{\xi}_{(i)}, \boldsymbol{\xi}_{(j)}) = \|\boldsymbol{\xi}_{(i)} - \boldsymbol{\xi}_{(j)}\| = \sqrt{\sum_{k=1}^{n} (\xi_{(i),k} - \xi_{(j),k})^2} \tag{2.2}$$

  mit $\boldsymbol{\xi}_{(i)}, \boldsymbol{\xi}_{(j)} \in \mathbb{R}^n$ herangezogen [BSMM99]. Im Unterschied zur Klassifizierung ist beim Clustering in der Regel nicht bekannt, wie die Klassen aussehen sollen.

- *Visualisierung*: Hochdimensionale Daten $\boldsymbol{\xi}_{(i)} \in \mathbb{R}^n$ sollen übersichtlich dargestellt werden, so dass die Topologie möglichst gut erhalten bleibt. Es wird also eine diskrete Abbildung von $\boldsymbol{\xi}_{(i)} \in \mathbb{R}^n$ auf $\boldsymbol{\psi}_{(i)} \in \mathbb{R}^m$ durchgeführt. Üblicherweise wird für die Visualisierung eine zweidimensionale Ausgabe $m = 2$, also $\boldsymbol{\psi}_{(i)} \in \mathbb{R}^2$ gewählt; pinzipiell stehen jedoch dieselben Methoden wie bei der Merkmalsextraktion zur Verfügung (vgl. 2.3).

- *Vektorquantisierung*: Aus den Datenvektoren $\mathcal{T} = \{\boldsymbol{\xi}_{(1)}, \boldsymbol{\xi}_{(2)}, ..., \boldsymbol{\xi}_{(w)}\},\ \boldsymbol{\xi}_{(i)} \in \mathbb{R}^n$ wird eine wesentlich kleinere Anzahl an *Referenzvektoren* $\boldsymbol{\varrho}_{(i)}$, ein sogenanntes *Codebook* $\mathcal{C} = \{\boldsymbol{\varrho}_{(1)}, \boldsymbol{\varrho}_{(2)}, ..., \boldsymbol{\varrho}_{(v)}\},\ \boldsymbol{\varrho}_{(i)} \in \mathbb{R}^n$ mit $v \ll w$ bestimmt, durch das die Daten möglichst gut repräsentiert werden. Zu jedem Datenvektor $\boldsymbol{\xi}_{(i)}$ existiert also ein optimaler Codebookvektor $\boldsymbol{\varrho}_{(i \leftarrow opt)}$ mit

$$\|\boldsymbol{\xi}_{(i)} - \boldsymbol{\varrho}_{(i \leftarrow opt)}\| = \min_k \{\|\boldsymbol{\xi}_{(i)} - \boldsymbol{\varrho}_{(k)}\|\}. \tag{2.3}$$

  Dadurch wird der zwangsläufig auftretende Quantisierungsfehler

$$E_q = \frac{1}{2} \sum_i (\boldsymbol{\xi}_{(i)} - \boldsymbol{\varrho}_{(i \leftarrow opt)})^2 \tag{2.4}$$

  minimiert.

2.4.3 Online-Lernen

Auch die Art, wie der eigentliche Lernvorgang - also das Anpassen irgendwelcher Parameter - abläuft, kann sich unterscheiden. Man spricht von *Online-Lernen*, wenn die Trainingsdaten nacheinander präsentiert werden und das System dadurch schrittweise

optimiert wird. Ein einzelner Trainingsvektor bewirkt dabei lediglich eine kleine Veränderung der Parameter. Sollten die Daten mit der Zeit z.B. durch alternde Sensoren driften, kann sich das System daran anpassen ohne dass der gesamte Lernvorgang erneut von vorne begonnen werden müsste. Ein weiterer Vorteil besteht darin, dass immer nur einzelne Trainingsvektoren bearbeitet werden müssen und Rechenaufwand und Speicherbedarf dadurch klein gehalten werden. Das Online-Lernen eignet sich daher vor allem für Anwendungen mit veränderlichen Eingabedaten oder große Datenmengen.

2.4.4 Offline-Lernen

Anders sieht es aus, wenn alle Trainingsvektoren auf einmal verarbeitet werden sollen (*Offline-Lernen*). Eine Anpassung an neue Daten ist dann nicht ohne Weiteres möglich. Auch ist der Rechenaufwand wesentlich größer als beim Online-Lernen, da alle Datensätze zugleich berücksichtigt werden müssen. Allerdings lassen sich dadurch auch oft bessere Ergebnisse erzielen. [Koh01, Fri98]

2.5 Netzwerkmodelle

2.5.1 Perceptron

Das von Frank Rosenblatt entwickelte Perceptron ist das erste künstliche neuronale Netzwerk. Als Vorbild dienten Rosenblatt Nervenzellen aus der Retina. Er erkannte, dass bereits im Auge eine Vorverarbeitung mit Mustererkennung stattfindet. In seinem Modell verwendet er künstliche Neuronen, die einfache mathematische Berechnungen durchführen können.

2.5.1.1 Aufbau

Ein Neuron j summiert die an seinen Eingängen ξ_i anliegenden Signale auf (s. Abb. 2.1). Überschreitet die Summe (genannt *Nettoinput*) $net_j = \sum \xi_i$ einen bestimmten Schwellwert θ_j (*Bias*), so wird der Ausgabewert, die *Aktivierung* ψ_j, von 0 auf 1 gesetzt. Als Übertragungsfunktion wird also die *Schwellwertfunktion* verwendet, die in Abb. 2.2 dargestellt ist.

Für die Aktivierung ψ_j eines Neurons gilt damit entsprechend:

$$\psi_j = f_S(net_j) \tag{2.5}$$

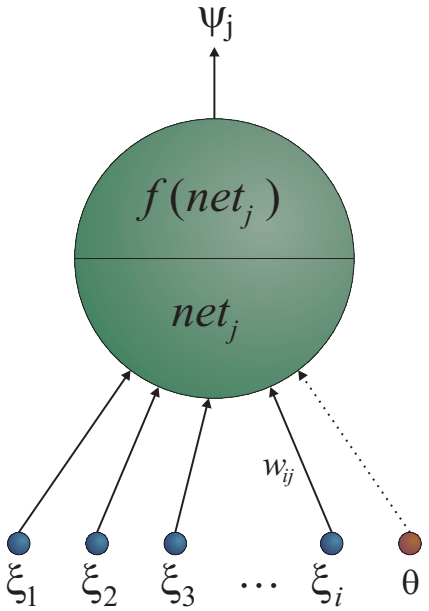


Abbildung 2.1: Neuron eines Perceptrons

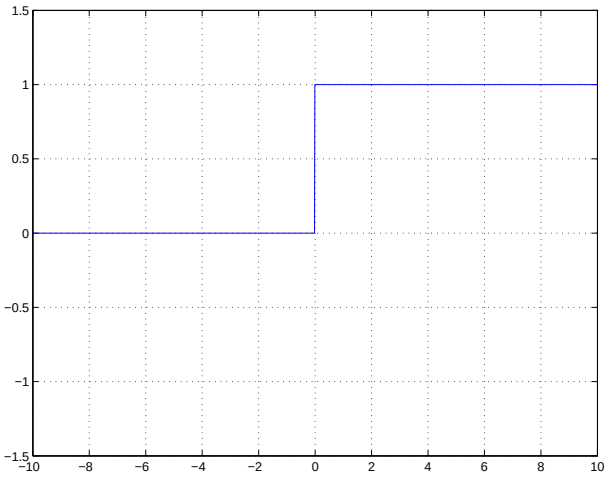


Abbildung 2.2: Schwellwertfunktion

Ein Perceptron enthält mehrere solcher Neuronen, die nebeneinander angeordnet sind. Jedes dieser Neuronen j ist dabei einer Klasse $\mathcal{K}_j$ zugeordnet, die das Netz unterscheiden soll. Das Neuron mit der größten Aktivierung repräsentiert die erkannte Klasse. Diese Neuronenschicht wird daher als Ausgabeschicht bezeichnet (Abb. 2.3 oben). Schreibt man die Aktivierungen aller Neuronen der Ausgabeschicht hintereinander, so erhält man den Ausgebevektor $\boldsymbol{\psi}_{(k)} = [\psi_1, \psi_2, ...\psi_n]^T$.

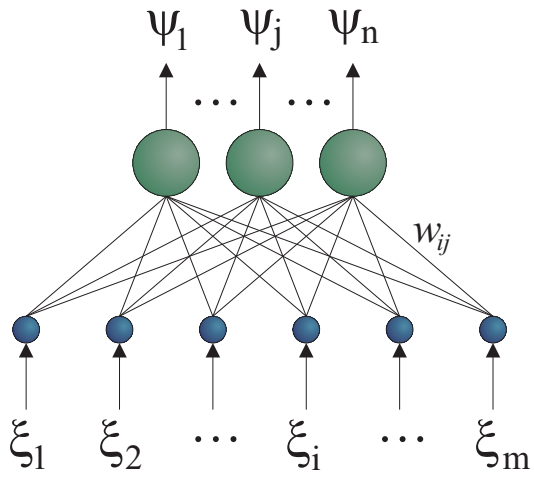


Abbildung 2.3: Aufbau eines Perceptrons

Perceptrons können ausschließlich für überwachtes Lernen eingesetzt werden, d.h. sie setzen die Existenz einer Trainingsdatenmenge $\mathcal{T} = \{(\boldsymbol{\xi}_{(1)}, \boldsymbol{\zeta}_{(1)}), (\boldsymbol{\xi}_{(2)}, \boldsymbol{\zeta}_{(2)}), ..., (\boldsymbol{\xi}_{(z)}, \boldsymbol{\zeta}_{(z)})\}$ voraus, die aus Paaren von Eingabevektoren $\boldsymbol{\xi}_{(k)} \in \mathbb{R}^m$ und zugehörigen Zielvektoren $\boldsymbol{\zeta}_{(k)} \in \mathbb{R}^n$ besteht. Die Eingabevektoren $\boldsymbol{\xi}_{(k)}$ werden dem Netz in der Eingabeschicht (Abb. 2.3 unten) präsentiert, in der sich nur "gedachte" Neuronen i befinden; in ihnen findet keine Verarbeitung statt und für den Ausgang des i-ten Eingabeneurons gilt: $\psi_i = \xi_i$. Ein Perceptron muss daher genauso viele Eingabeneuronen haben, wie der Trainingsvektor $\boldsymbol{\xi}_{(k)}$ Komponenten hat. Da durch die Anzahl der zu erkennenden Klassen auch die Anzahl der Ausgabeneuronen festgelegt ist, wird die Topologie eines Perceptrons vollständig durch die zu verarbeitenden Daten bestimmt.

Ein- und Ausgabeneuronen sind über positive oder negative Gewichte $w_{ij} \in \mathbb{R}$ miteinander verbunden, wobei i den Index des Eingabeneurons und j den des Ausgabeneurons bezeichnet. Die Aktivierung ψ_i der Eingabeneuronen wird also an die Eingänge ξ_j der Ausgabeneuronen weitergereicht und dabei je nach Gewicht verstärkt oder abgeschwächt. Das Verhalten jedes Neurons ist also durch die Gewichte an seinen Eingängen veränderbar. Da die Informationsleitung nur in eine Richtung - von der Eingabeschicht zur Ausgabeschicht - stattfindet, spricht man von einem *feed forward* Netz.

Zur Vereinfachung der Notation kann der Datenvektor $\boldsymbol{\xi}_{(k)} = (\xi_1, \xi_2, ..., \xi_n)^T$ gedanklich um eine Komponente $\xi_0 = 1$ erweitert werden, so dass das Bias über das zugehörige

Gewicht w_0 eingestellt werden kann (vgl. Abb. 2.1). Auf das Symbol θ kann so verzichtet werden.

2.5.1.2 Training

Der Lernvorgang für ein Perceptron-Netz sieht nun wie folgt aus:

- **Schitt 1**: Die Gewichte w_{ij} zwischen allen Neuronen i und j werden mit zufälligen Werten, üblicherweise $-1 \leq w_{ij} \leq 1$, initialisiert.
- **Schritt 2**: Über die Eingabeneuronen i wird dem Netz ein zufällig ausgewählter Trainingsvektor $\boldsymbol{\xi}_{(k)} = [\xi_1, \xi_2, ...\xi_m]^T$ präsentiert: $\psi_i = \xi_{(k),i}$
- **Schritt 3**: Die Ausgabeneuronen j errechnen ihren Nettoinput net_j aus den von vorgeschalteten Neuronen i empfangenen Signalen ψ_i gewichtet mit den Verbindungsstärken w_{ij}:
$$net_j = \sum_i w_{ij} \cdot \psi_i. \tag{2.6}$$
- **Schritt 4**: Der Ausgabewert ψ_j der Ausgabeneuronen wird berechnet
$$\psi_j = f_S(net_j) = f_S\left(\sum_i w_{ij} \cdot \psi_i\right) \tag{2.7}$$
und der so erhaltene Ausgabevektor $\boldsymbol{\psi}_{(k)} = [\psi_1, \psi_2, ...\psi_n]^T$ mit dem gewünschten Zielvektor $\boldsymbol{\zeta}_{(k)} = [\zeta_1, \zeta_2, ..., \zeta_n]^T$ verglichen.
- **Schritt 5**: Die Gewichte werden angepasst, so dass der Trainingsvektor $\boldsymbol{\xi}_{(k)}$ bei der nächsten Präsentation besser abgebildet werden kann, der Abstand von Ausgangsvektor $\boldsymbol{\psi}_{(k)}$ zu Zielvektor $\boldsymbol{\zeta}_{(k)}$ gegeben durch
$$d_E = \sum_i \sqrt{(\psi_{(k),i} - \zeta_{(k),i})^2} \tag{2.8}$$
also geringer wird. Dies geschieht durch folgende Formel:
$$w_{ij,neu} = w_{ij} + \eta \cdot (\zeta_j - \psi_j) \cdot \xi_i \tag{2.9}$$
Dabei kann die positive *Lernrate* η zwischen 0 und 1 eingestellt und so die Stärke der Anpassung beinflusst werden.

Die Schritte 2 bis 5 werden mit allen Trainingsvektoren mehrfach durchgeführt, bis das Netz die Eingabevektoren fehlerfrei auf die Zielvektoren abbilden kann. Die Lernrate wird dabei üblicherweise immer weiter verringert[1]. Ist das Netz erst einmal trainiert, lassen sich auch unbekannte Signale (die den bekannten Vektoren ähnlich sind) den gewünschten Klassen zuordnen.

[1] in 2.5.2.3 wird genauer auf die Lernrate eingegangen

2.5.1.3 Bewertung

Bei genauer Untersuchung des Perceptron-Modells durch Marvin Minsky und Seymour Papert stellte sich heraus, dass das klassische Perceptron lediglich linear trennbare Klassifikationen durchführen kann. Sogar einfache Aufgaben wie das XOR-Problem lassen sich also nicht lösen. Auch Netzwerke mit mehreren Schichten verhalten sich ähnlich, da sie sich wegen ihrer linearen Übertragungsfunktion stets auf einschichtige Netze reduzieren lassen. [RMS91, Hay94]

2.5.2 Multilayer-Perceptron

Abhilfe schafft das *Multilayer-Perceptron* (MLP): Es besteht im Vergleich zum Perceptron aus mehreren Schichten und anstatt einer Schwellwertfunktion wird eine nichtlineare, üblicherweise s-förmige (*sigmoide*) Übertragungsfunktion verwendet, wie sie in Abb. 2.4 dargestellt ist:

$$f_{sig}(x) = \frac{1}{1 + e^{-x}} \tag{2.10}$$

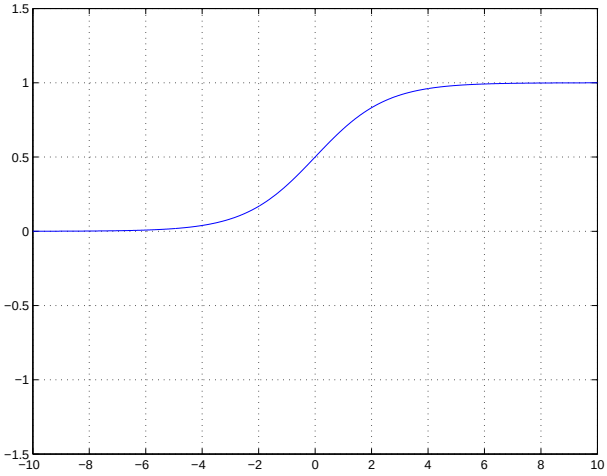


Abbildung 2.4: sigmoide Übertragungsfunktion

Über das Bias lässt sie sich nach oben oder unten verschieben.

2.5.2.1 Aufbau

Wie beim Perceptron dient die erste Neuronenschicht zur Aufnahme von externen Eingabewerten $\xi_{(k)}$ und die letzte zur Wiedergabe der Ausgabewerte $\psi_{(k)}$. Zusätzlich sind

aber noch eine oder mehrere Schichten dazwischen vorhanden, die von außen nicht zugänglich sind und deshalb als *verborgene Schichten* (engl. *hidden layers*) bezeichnet werden (s. Abb. 2.5).

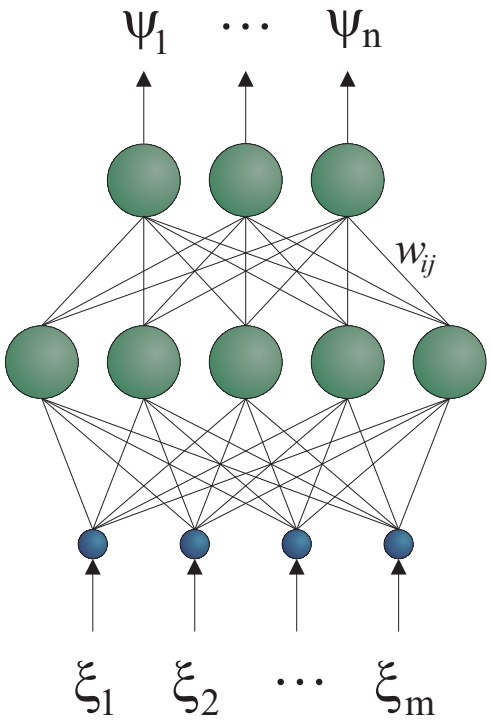


Abbildung 2.5: Aufbau eines zweischichtigen Multilayer-Perceptrons

Die Signale werden von Schicht zu Schicht weitergereicht; der Informationsfluss findet jedoch nur in eine Richtung - von Neuronen der Eingabeschicht über Neuronen der inneren Schichten zu den Neuronen der Ausgebeschicht - statt, weshalb MLPs ebenfalls in die Klasse der feed-forward-Netzwerke einzuordnen sind. Für Neuronen j, die Signale von den Neuronen i empfangen, gilt:

$$net_j = \sum_i w_{ij} \cdot \psi_i \tag{2.11}$$

und

$$\psi_j = f_S(net_j) \tag{2.12}$$

Auch beim MLP ist die Anzahl der Neuronen in Ein- und Ausgabeschicht durch die Daten vorgegeben, die Anzahl der verborgenen Schichten und Neuronen darin bleibt jedoch dem Anwender überlassen. Da in der Eingabeschicht keine Verarbeitung stattfindet, wird diese nicht mitgezählt, wenn die Anzahl der Schichten eines Netzwerkes spezifiziert wird.

2.5.2.2 Backpropagation Algorithmus

Das Training eines MLPs gestaltet sich schwieriger als beim einschichtigen Perceptron, da die einzustellenden Parameter (die Gewichte w_{ij}) über mehrere Schichten verteilt sind. Eine weitere Schwierigkeit stellt die nichtlineare Übertragungsfunktion dar, weshalb sich eine analytische Lösung für große Netzwerke nachweislich nicht angeben lässt. Erst der sogenannte *Backpropagation Algorithmus*, der den Fehler an der Ausgabeschicht schichtweise zurück durch das Netz propagiert und dabei von Schicht zu Schicht korrigiert, konnte dieser Netzwerkarchitektur zum Durchbruch verhelfen.

Ein trainiertes Netzwerk sollte die Eingabevektoren $\boldsymbol{\xi}_{(k)}$ durch seine Ausgabe $\boldsymbol{\psi}_{(k)}$ möglichst gut auf die Zielvektoren $\boldsymbol{\zeta}_{(k)}$ abbilden. Für den gesamten Trainingsdatensatz sollte sich also ein minimaler quadratischer Fehler ergeben:

$$E_q = \frac{1}{2} \sum_{k=1}^{z} \|\boldsymbol{\psi}_{(k)} - \boldsymbol{\zeta}_{(k)}\|^2 \tag{2.13}$$

Genau dann nämlich hat das Netz die unbekannten Relation $\boldsymbol{\zeta} = f(\boldsymbol{\xi})$ optimal modelliert. Ziel des Backpropagation-Lernverfahrens ist es, eine Kombination von Verbindungsgewichten w_{ij} zu finden, mit denen sich die Eingabemuster möglichst fehlerfrei auf die Zielvektoren abbilden lassen.

Ist für einen bestimmten Trainingsvektor $\boldsymbol{\xi}_{(k)}$ der Fehler

$$E = \frac{1}{2} \sum_{i} (\psi_{(k),i} - \zeta_{(k),i})^2 \tag{2.14}$$

aufgetreten, so hat jedes Gewicht, das mit einem Neuron der Ausgabeschicht verbunden ist, einen kleinen Teil dazu beigetragen. Wird es um einen kleinen Wert Δw_{ij} in Richtung des negativen Fehlergradienten verschoben, so lässt sich der Fehler E, der ja von der Aktivierung ψ_j des Neurons j, seinem Nettoinput net_j und damit letztendlich auch von dem Gewicht selbst abhängt, verringern.

Auch für die Gewichte w_{hi} der darunterliegenden Schicht lässt sich ein Δw_{hi} angeben, das zu ihnen hinzu addiert werden muss, damit der Fehler weiter abnimmt. Dieser Term enthält einen Ausdruck, der bereits bei den Gewichten w_{ij} aufgetaucht ist. Es ergibt sich eine rekursive Formel für die Anpassung aller Gewichte, die sich von der obersten Schicht bis hinunter zur untersten abarbeiten lässt. Der Fehler am Ausgang wird also zurück durch das gesamte Netzwerk propagiert, weshalb das Verfahren als Backpropagation Algorithmus berühmt wurde[2].

Zusammenfassend lässt sich das Training eines MLPs folgendermaßen beschreiben:

[2] eine detaillierte Beschreibung des Backpropagation Algorithmus findet sich in [Hay94]

- **Schritt 1**: Die Gewichte aller Neuronen werden mit zufälligen Werten initialisiert.
- **Schritt 2**: Dem Netz wird über die Eingabeschicht ein zufällig ausgewählter Trainingsvektor präsentiert.
- **Schritt 3**: Das Signal wird schichtweise von der Eingangsschicht bis hin zu Ausgangschicht propagiert.
- **Schritt 4**: Der Fehler am Ausgang wird berechnet.
- **Schritt 5**: Unter Verwendung des Backpropagation Algorithmus werden die Gewichte angefangen an der obersten Schicht bis hin zur untersten Schicht rekursiv angepasst, so dass sich der Gesamtfehler reduziert.

Die Schritte 2 bis 5 werden wiederholt, bis das Netz die Daten sicher klassifizieren kann. Die Lernrate ist dabei geeignet einzustellen.

2.5.2.3 Lernrate

Bei der Auswahl der Lernrate muss besondere Sorgfalt angewandt werden. Ziel ist es, dass das Netz nach dem Training einen stationären Zustand erreicht und die Gewichte nicht mehr verändert werden müssen. Die Lernrate zu Trainingsende muss dazu möglichst klein sein. Damit sich das Netz zu Trainingsbeginn schnell an die geforderte Aufgabe anpassen kann, sollte anfangs eine möglichst große Lernrate gewählt werden. Während des Trainings könnte man die Lernrate beispielsweise linear verringern:

$$\eta(t) = \left(1 - \frac{t}{t_{max}}\right)(\eta_{max} - \eta_{min}) \tag{2.15}$$

wobei t_{max} die gesamte Trainingsdauer und η_{min} bzw. η_{max} den Start- und Endwert der Lernrate kennzeichnen.

Auch eine exponentiell abnehmende Lernrate wäre denkbar:

$$\eta(t) = \eta_{max}\left(\frac{\eta_{min}}{\eta_{max}}\right)^{\frac{t}{t_{max}}} \tag{2.16}$$

Der Vorteil liegt darin, dass die Lernrate sehr schnell absinkt und sich das Netz nach anfänglich großen Anpassungen über einen längeren Zeitraum gut an die Trainigsdaten adaptieren kann.

Eine konstante Lernrate könnte verwendet werden, wenn sich das Netz ständig auf neue Eingaben einstellen muss; allerdings konvergiert es dann in der Regel nicht oder nur sehr schwer.

Es wäre auch denkbar eine Lernrate zu definieren, die kurzzeitig größere Werte annehmen kann. Dadurch werden weitreichende Anpassungen im Netz möglich, so dass auch ein lokales Fehlerminimum wieder verlassen werden kann. Die Lernrate könnte dazu beispielsweise an den Klassifizierungsfehler gekoppelt werden.

Eine stetig steigende Lernrate macht jedoch keinen Sinn und wird auch nicht verwendet.

2.5.2.4 Bewertung

Neben der Einstellung der Lernrate ist es ein großes Problem, die richtige Netzwerkgröße zu finden, so dass die Trainingsdaten nicht auswendig gelernt und auch unbekannte Daten gut klassifiziert werden. Um dies zu garantieren, teilt man üblicherweise die zur Verfügung stehenden Daten in ein Trainings- und ein Testset ein. Das Testset dient dabei lediglich dazu, die *Generalisierungsleistung* des Netzes bei unbekannten Daten zu beurteilen, eine Anpassung der Gewichte erfolgt nur beim Trainingsset.

Da es sich beim Backpropagation Algorithmus um ein Gradientenabstiegsverfahren handelt, besteht die Gefahr, dass das Netz lediglich ein lokales Fehlerminimum erreicht. Das Ergebnis wird also durch die anfängliche Wahl der Gewichte wesentlich beeinflusst. Es werden daher oft mehrere Netze mit unterschiedlichen Startwerten gleichzeitig trainiert, von denen dann das beste für die weitere Optimierung ausgewählt wird. [RMS91, Hay94]

2.5.3 Vektorquantisierung

Ein gänzlich anderes Ziel verfolgt die Vektorquantisierung (VQ). Eine Menge von Datenvektoren $\boldsymbol{\xi}_{(i)} \in \mathbb{R}^n$ soll dabei durch eine geringere Anzahl an Codebuch-Vektoren $\boldsymbol{\varrho}_{(k)} \in \mathbb{R}^n$ ersetzt werden. Die Schwierigkeit besteht darin, ein geeignetes Codebuch $\mathcal{C} = \{\boldsymbol{\varrho}_{(1)}, \boldsymbol{\varrho}_{(2)}, ..., \boldsymbol{\varrho}_{(z)}\},\ \boldsymbol{\varrho}_{(i)} \in \mathbb{R}^n$ zu finden. Ist dieses erst einmal erstellt, kann ein gegebener Datenvektor $\boldsymbol{\xi}_{(k)}$ durch den am nächsten gelegenen Codebuch-Vektor $\boldsymbol{\varrho}_{(k \leftarrow opt)}$ beschrieben werden. Als Abstandsmaß wird dafür üblicherweise der euklid'sche Abstand d_E verwendet:

$$d_E = \|\boldsymbol{\xi}_{(k)} - \boldsymbol{\varrho}_{(k \leftarrow opt)}\| = \min_i \{\|\boldsymbol{\xi}_{(k)} - \boldsymbol{\varrho}_{(i)}\|\} \tag{2.17}$$

Werden die Codebuch-Vektoren $\boldsymbol{\varrho}_{(k \leftarrow opt)}$ optimal ausgewählt, so minimiert sich der durchnittlich erwartete Quantisierungsfehler, der gegeben ist durch die Formel

$$E = \int \|\boldsymbol{\xi}_{(k)} - \boldsymbol{\varrho}_{(k \leftarrow opt)}\|^2 p(\boldsymbol{\xi}) d\boldsymbol{\xi} \tag{2.18}$$

wobei $p(\boldsymbol{\xi})$ die Wahrscheinlichkeitsdichtefunktion von $\boldsymbol{\xi}$ darstellt.

Um ein geeignetes Codebuch zu finden gibt es eine Reihe von Verfahren. Populär ist die von Teuvo Kohonen entwickelte *Learning Vector Quantization* (LVQ) mit den Algorithmen LVQ1, LVQ2, LVQ3 und OLVQ1. Das Codebuch wird dabei durch Training erstellt. Im Wesentlichen wird immer derjenige Codebuchvektor ausgewäht und angepasst, der am besten zum Trainingsdatensatz passt. Ist bisher noch kein "guter" Referenzvektor vorhanden, wird er anhand des Trainingsvektors erstellt. Eine detaillierte Beschreibung der unterschiedlichen Verfahren ist in [Koh01] enthalten. [Koh01, Fri98]

2.5.4 Selbstorganisierende Karte

Die *selbstorganisierende Karte* (engl. *Self-Organizing Map*, SOM), die ebenfalls von Teuvo Kohonen entwickelt wurde und gelegentlich auch als Kohonen-Feature-Map bezeichnet wird, stellt eine Kombination aus Vektorquantisierung und Perceptron-Netzen dar. Bei den in Kapitel 2.5.2 vorgestellten Backpropagation-Netzen wird die Funktion eines Neurons im Wesentlichen durch seine Zugehörigkeit zu einer bestimmten Schicht bestimmt; seine Position in dieser ist jedoch nicht von Bedeutung. Dieses Prinzip ist dem biologischen Vorbild nur bedingt ähnlich, denn im Gehirn spielt die räumliche Lage eines Neurons sehr wohl eine Rolle.

2.5.4.1 Aufbau

Kohonen hat sein SOM-Modell aufgebaut wie ein zweischichtiges Perceptron (s. Abb. 2.6). Von einer Eingabeschicht wird die Aktivierung an eine zweidimensionale Ausgangsschicht weitergeleitet, die die eigentliche selbstorganisierende Karte darstellt. Die Neuronen in dieser sind entweder quadratisch oder hexagonal angeordnet. Über die Gewichte w_{ij} zwischen Ein- und Ausgabeschicht ist jedem dieser Neuronen j ein Referenzvektor $\boldsymbol{\varrho}_{(j)}(t) = [w_{1j}, w_{2j}, ..., w_{nj}]^T \in \mathbb{R}^n$ zugeordnet, der zeitlich variabel ist. Alle Neuronen erhalten von der Eingabeschicht identische Datenwerte, um die sie *konkurrieren*. Gewinner für den Datenvektor $\boldsymbol{\xi}(t)$ ist genau das Neuron b mit der kleinsten Aktivierung, bei dem also der euklid'sche Abstand $d_E(\boldsymbol{\xi}(t), \boldsymbol{\varrho}_{(b)}(t))$ seines Referenzvektors $\boldsymbol{\varrho}_{(b)}(t)$ zum Eingabevektor $\boldsymbol{\xi}(t)$ am geringsten ist[3]. Für den Index b gilt dabei:

$$b = \arg\min_i \{d_E(\boldsymbol{\xi}(t), \boldsymbol{\varrho}_{(i)}(t))\} \tag{2.19}$$

Es handelt sich bei der selbstorganisierenden Karte also um ein *vektorbasiertes Netz* mit einer Form von *Wettbewerbslernen* (engl. *Competetive Learning*).

[3]Man beachte, dass bei der SOM ein möglichst kleiner Ausgabewert angestrebt wird, bei den Perceptron-Netzen war hingegen das Neuron mit der größten Aktivierung der "Gewinner".

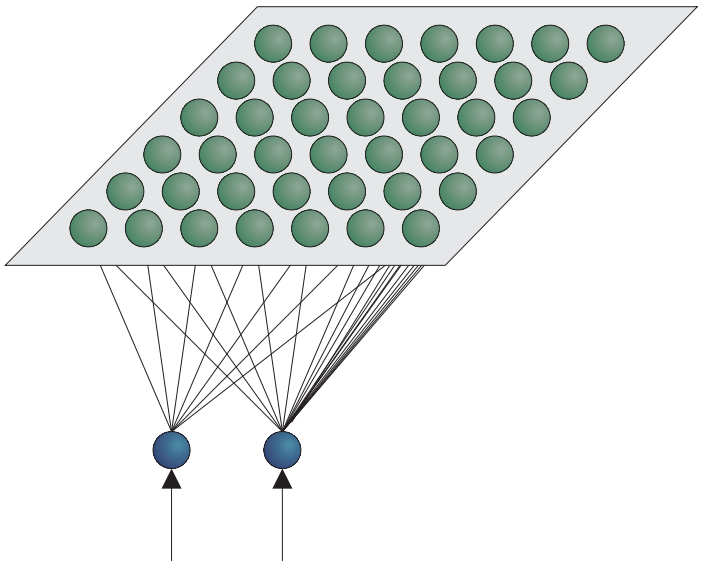


Abbildung 2.6: Aufbau einer selbstorganisierenden Karte

2.5.4.2 Nachbarschaftsfunktionen

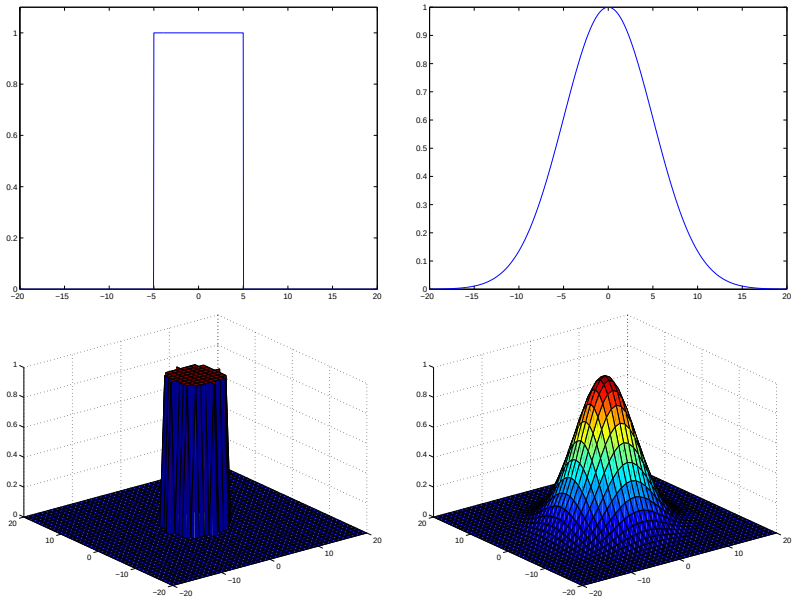
Eine SOM ist lernfähig, indem die Gewichte des Gewinners b in der Weise angepasst werden, dass der euklid'sche Abstand d_E des Referenzvektors $\boldsymbol{\varrho}_{(b)}(t)$ zum Trainingsvektor $\boldsymbol{\xi}(t)$ verringert wird. Außer dem Gewinner b selbst werden allerdings auch Neuronen i in der Nachbarschaft in diese Anpassung einbezogen. Wie stark die Beeinflussung ist, hängt von einer *Nachbarschaftsfunktion* $h_{(i,b)}$ (engl. *neighborhood function*) ab. In Abb. 2.7 sind die am häufigsten verwendeten Funktionen in zwei- und dreidimensionaler Darstellung abgebildet.

Die Bubble-Funktion (Abb. 2.7 a) ist eine Schwellwertfunktion[4], so dass alle Neuronen i in einem definierten Abstand $\sigma(t)$ um das Gewinnerneuron b herum gleich stark angepasst werden. Weiter entfernte Neuronen werden nicht beeinflusst. Als Abstandsmaß zwischen zwei Neuronen (i, b) wird ebenfalls der euklid'sche Abstand d_E verwendet, der sich diesmal über ihre Position in der Kartenebene definiert:

$$d_E(i,b) = \sqrt{(i_x - b_x)^2 - (i_y - b_y)^2} \tag{2.20}$$

wobei i_x die Position eines Neurons i auf der x-Achse und i_y die Lage auf der y-Achse bezeichnet.

[4] vgl. dazu 2.5.1



(a) Bubble (b) Gauß

Abbildung 2.7: Nachbarschaftsfunktionen

Wesentlich rechenintensiver als die Bubble-Funktion, bei der lediglich Abstandswerte verglichen werden müssen, ist die Gauß-Funktion (Abb. 2.7 b):

$$f_{Gauss}(x) = \exp^{-\left(\frac{x^2}{\sigma(t)^2}\right)}, \tag{2.21}$$

Der Radius lässt sich über die Standardabweichung $\sigma(t)$ einstellen. Die Neuronen werden dabei umso schwächer angepasst, je weiter sie vom Gewinnerneuron im Zentrum entfernt sind.

2.5.4.3 Training

Die Gewichtsanpassung wird nur um den ersten Gewinner herum durchgeführt, der "zweite Sieger" wird nicht berücksichtigt, weshalb man vom *winner take all* Prinzip (WTA) spricht.

Auch wie bei den Perceptron-Netzen gibt es eine Lernrate, die man in der Regel gegen 0 konvergieren lässt[5]. Als zweiter Parameter wird der Nachbarschaftsradius ebenfalls stetig fallend gewählt, so dass gegen Ende des Trainings nur noch die Gewichte in unmittelbarer Nähe des Gewinnerneurons optimiert werden. Der neue Referenzvektor $\boldsymbol{\varrho}_{(i)}(t)$ eines Neurons i errechnet sich dabei wie folgt:

[5] vgl. 2.5.2.3

$$\boldsymbol{\varrho}_{(i)}(t+1) = \boldsymbol{\varrho}_{(i)}(t) + \eta(t) \cdot h_{(i,b)}(t)(\boldsymbol{\xi}(t) - \boldsymbol{\varrho}_{(i)}(t)), \tag{2.22}$$

wobei $h_{(i,b)}(t)$ als Nachbarschaftsfunktion die Stärke der Gewichteanpassung eines Neurons i bestimmt, das einen bestimmten Abstand d_E vom Gewinnerneuron b hat.

Mit diesem Lernverfahren ist die SOM fähig, eine Abbildung eines kontinuierlichen mehrdimensionalen Raumes $\mathbb{R}^n$ auf einen diskreten zwei-dimensionalen Raum $\mathbb{R}^2$ durchzuführen. Durch Verwendung des euklid'schen Abstandsmaßes erhält man eine Abbildung, bei der die topologischen Eigenschaften der Datenvektoren auch in der zweidimensionalen Darstellung erhalten bleiben. Die SOM führt dabei selbständig ein Clustering der Daten durch und zählt damit zu den unüberwachten Lernverfahren.

Zusammenfassend soll noch einmal der komplette Lernalgorithmus beschrieben werden:

- **Schritt 1**: Die Referenzvektoren $\boldsymbol{\varrho}_{(i)}(t)$ aller Neuronen i werden mit zufälligen Gewichten, üblicherweise $w_{ij} \in [-1; 1]$, initialisiert.
- **Schritt 2**: Dem Netz wird ein zufällig ausgewählter Trainingsvektor $\boldsymbol{\xi}(t)$ präsentiert.
- **Schritt 3**: Das Neuron b mit dem am besten daran angepassten Referenzvektor $\boldsymbol{\varrho}_{(b)}(t)$ wird bestimmt und als Gewinner ausgewählt:
$$b = \arg\min_i \{d_E(\boldsymbol{\xi}(t), \boldsymbol{\varrho}_{(i)}(t))\} \tag{2.23}$$
- **Schritt 4**: Die Gewichte des Gewinnerneurons und der Neuronen i aus der Nachbarschaft werden angepasst:
$$\boldsymbol{\varrho}_{(i)}(t+1) = \boldsymbol{\varrho}_{(i)}(t) + \eta(t) \cdot h_{(i,b)}(t)(\boldsymbol{\xi}(t) - \boldsymbol{\varrho}_{(i)}(t)) \tag{2.24}$$

Die Schritte 2 bis 4 werden so lange ausgeführt, bis das Training beentet ist und das Netz die Trainingsvektoren sicher abbilden kann. Lernrate und Nachbarschaftsradius werden dabei stetig vermindert.

2.5.4.4 Bewertung

Nach dem Training ist jedes Neuron ein "Spezialist" für einen bestimmten Datenvektor geworden. Neuronen in seiner Umgebung sind dann besonders gut an ähnliche Daten angepasst. Es entsteht eine Merkmalskarte, auf der die topologischen Beziehungen der (hochdimensionalen) Trainingsdaten erhalten bleiben: Ähnliche Muster in den Trainingsdaten werden auf nahe beieinander liegende Gebiete der Kartenfläche abgebildet. Die selbstorganisierende Karte eignet sich daher dazu, Zusammenhänge in den Daten aufzuspüren und diese übersichtlich in zweidimensionaler Form darzustellen. [Koh01, Fri98]

2.5.5 Rekurrente selbstorganisierende Karte

Bei allen bisher vorgestellten Modellen werden zeitliche Beziehungen der Datenvektoren nicht berücksichtigt. Jedoch zeichnen sich viele Probleme wie Wettervorhersage oder Spracherkennung gerade durch die temporale Abhängigkeit der Daten aus. Es besteht also die Notwendigkeit, neuronale Netze auch für zeitliche Problemstellungen (engl. *Temporal Sequence Processing*, TSP) verwendbar zu machen.

Als einfachste Möglichkeit können n zeitlich aufeinander folgende Datenvektoren $\boldsymbol{\xi}(t), \boldsymbol{\xi}(t+1), ..., \boldsymbol{\xi}(t+n)$ gesammelt und dem Netz als ein Vektor $\boldsymbol{\xi}_{(neu)} = [\boldsymbol{\xi}(t), \boldsymbol{\xi}(t+1), ..., \boldsymbol{\xi}(t+n)]^T$ präsentiert werden. Dieses Modell wird als *Time Delay* Netz bezeichnet, da vor jedem Lernschritt gewartet werden muss, bis alle n Datensätze eingegangen sind. Ist die Anzahl der so aneinander gehängten Vektoren allerdings einmal gewählt, lässt sie sich während der Laufzeit nicht mehr verändern. Auch stellt sich das Problem, welche Datensätze in einem Vektor zusammengefasst werden sollen und welche in einen neuen gehören. Dies ist insbesondere bei der Spracherkennung ein Problem, wenn einzelne Phoneme oder Silben in getrennten Vektoren dargestellt werden sollen, da es häufig Überschneidungen zu zeitlich benachbarten Datensätzen gibt. [dABA00][VHdRM97]

2.5.5.1 Aufbau

Bei der *Recurrent Self-Organizing Map* (RSOM) wird ein anderer Ansatz verwendet: Hier werden die Neuronen selbst modifiziert, so dass sie ein "Gedächtnis" erhalten. Jedes Neuron j wird dazu um einen Integrator erweitert. Sein (mehrdimensionales) Ausgangssignal $\boldsymbol{\psi}_{(j)}(t)$ zum Zeitpunkt t errechnet sich unter Berücksichtigung der Aktivierung $\boldsymbol{\psi}_{(j)}(t-1)$ im vorangegangenen Zeitschritt:

$$\boldsymbol{\psi}_{(j)}(t) = (1-\alpha)\boldsymbol{\psi}_{(j)}(t-1) + \alpha(\boldsymbol{\xi}(t) - \boldsymbol{\varrho}_{(j)}(t)), \tag{2.25}$$

wobei $\boldsymbol{\varrho}_{(j)}(t)$ den Gewichtsvektor des Neurons und $\boldsymbol{\xi}(t)$ den aktuellen Trainingsvektor darstellt. Mit dem Parameter $\alpha \in [0;1]$ lässt sich die Stärke der "Erinnerung" an frühere Vektoren einstellen: Bei Werten von $\alpha \approx 1$ wird ein Kurzzeitspeicher realisiert, der bevorzugt aktuelle Eingabedaten berücksichtigt. Je kleiner α gewählt wird, umso weiter zurückliegende Eingangssignale werden in die Berechnung des Ausgangswertes einbezogen. Bei $\alpha = 1$ verhält sich die rekurrente selbstorganisierende Karte jedoch wie die in 2.5.4 beschriebene "normale" SOM. Das Ausgangssignal hängt dann lediglich von den aktuellen Eingabewerten ab, vergangene Aktivierungszustände werden nicht mehr berücksichtigt.

Für die Formel (2.25) kann man eine allgemeine Lösung folgendermaßen angeben:

$$\boldsymbol{\psi}_{(j)}(t) = \alpha \sum_{k=0}^{n-1} (1-\alpha)^k (\boldsymbol{\xi}(t-k)) - \boldsymbol{\varrho}_{(j)}(t-k)) + (1-\alpha)^n \boldsymbol{\psi}_{(j)}(t-n) \tag{2.26}$$

Wird nun $\boldsymbol{\phi}_{(j)}(t) := \boldsymbol{\xi}(t) - \boldsymbol{\varrho}_{(j)}(t)$ gesetzt, lässt sie sich in der Form

$$\boldsymbol{\psi}_{(j)}(t) = (1-\alpha)\boldsymbol{\psi}_{(j)}(t-1) + \alpha\boldsymbol{\phi}_{(j)}(t) \tag{2.27}$$

schreiben, was einem linearen *IIR-Filter*[6] mit der Impulsantwort $I(k) = \alpha(1-\alpha)^k$ für $k \geq 0$ entspricht. Der Aufbau eines IIR-Filters wird in Abb. 2.8 verdeutlicht.

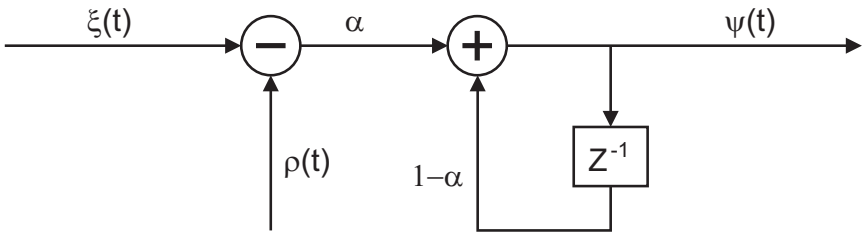


Abbildung 2.8: Aufbau eines IIR-Filters

Man kann sich das IIR-Filter als Gedächtniselement vorstellen, das vergangene Werte speichern kann, so dass sie bei der Berechnung eines neuen Ausgangssignales berücksichtigt werden können. Hierbei werden Werte aus der nahen Vergangenheit stärker angerechnet als weiter zurückliegende. Am besten wird dieses Verhalten aus der Impulsantwort deutlich, wie sie für $\alpha = 0.4$ und $\boldsymbol{\varrho}(t) = 0$ in Abbildung 2.9 dargestellt ist. Einmal mit dem Dirac-Impuls angeregt, klingt das Filter exponentiell gegen null ab. Werte unter 5% des Anfangswertes können in der Regel vernachlässigt werden. Für $\alpha = 0.4$ hieße das, dass die vergangenen 5 Trainingsvektoren für die aktuelle Berechnung eine Rolle spielen.

2.5.5.2 Training

Der Trainingsalgorithmus muss auf Grund des erweiterten Neuronenmodells im Vergleich zu 2.5.4 leicht abgewandelt werden:

- **Schritt 1**: Die Referenzvektoren $\boldsymbol{\varrho}_{(j)}$ aller Neuronen j werden mit zufälligen Werten initialisiert.
- **Schritt 2**: Dem Netz wird ein zufällig ausgewählter Trainingsvektor $\boldsymbol{\xi}(t)$ präsentiert.

[6] Infinite Impulse Response

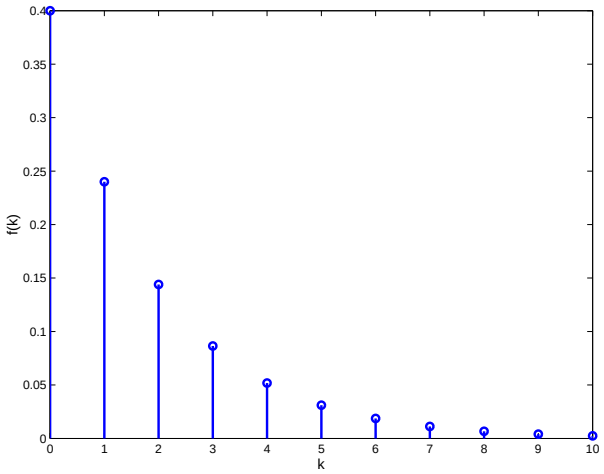


Abbildung 2.9: Impulsantwort des IIR-Filters

- **Schritt 3**: Die Aktivierung aller Neuronen wird berechnet:

$$\boldsymbol{\psi}_{(j)}(t) = (1-\alpha)\boldsymbol{\psi}_{(j)}(t-1) + \alpha(\boldsymbol{\xi}(t) - \boldsymbol{\varrho}_{(j)}(t)) \tag{2.28}$$

- **Schritt 4**: Das Neuron b mit dem am besten angepassten Referenzvektor $\boldsymbol{\varrho}_{(j)}(t)$ wird bestimmt und als Gewinner ausgewählt:

$$\psi_b = min_j\{\|\boldsymbol{\psi}_{(j)}(t)\|\} \tag{2.29}$$

- **Schritt 5**: Die Gewichte des Gewinnerneurons und der Neuronen aus der Nachbarschaft werden angepasst: Die Anpassung der Gewichtsvektoren basiert auf dem Differenzvektor $\boldsymbol{\psi}_{(j)}(t)$:

$$\boldsymbol{\varrho}_{(j)}(t+1) = \boldsymbol{\varrho}_{(j)}(t) + \eta(t)h_{(j,b)}(t)\boldsymbol{\psi}_{(j)}(t) \tag{2.30}$$

Die Schritte 2 bis 4 werden so lange ausgeführt, bis das Training beentet ist und das Netz die Trainingsvektoren sicher abbilden kann. Lernrate $\eta(t)$ und Nachbarschaftsradius $\sigma(t)$ in $h_{(i,b)}(t)$ werden stetig vermindert. Der Faktor α bleibt jedoch über das gesamte Training konstant.

2.5.5.3 Bewertung

Da es sich bei der RSOM um ein relativ neues Modell handelt, werden in der Literatur vergleichweise wenig Anwendungen beschrieben. In [VHdRM97] wurde eine RSOM erfolgreich zur Prädiktion von temporalen Daten eingesetzt.

Der Berechnungsaufwand ist im Vergleich zur SOM höher, da das Ausgabesignal des letzten Zeitschrittes gewichtet mit dem Faktor α zum aktuellen Wert hinzu addiert werden muss. [dABA00]

Kapitel 3

Parallele selbstorganisierende Karten

Den im letzten Kapitel vorgestellen RSOM-Algorithmus auf einem *Rechnercluster* zu verwenden, ist nicht ohne Weiteres möglich. Ein wesentliches Problem ist der Datenaustausch unter Prozessen, die auf mehreren Rechnern *parallel* - also gleichzeitig - ausgeführt werden. Anstatt dafür eine eigene Kommunikationsschnittstelle in das Programm zu implementieren, kann eine fertige *Message Passing Software* verwendet werden.

3.1 Rechnercluster

Rechnercluster bestehen aus mehreren einzelnen Rechnern genannt *Knoten* oder *Nodes* (s. Abb. 3.1). Jeder dieser Nodes kann über mehrere Prozessoren verfügen, man spricht dann von einem *Symmetric Multiprocessor* (SMP) Node. Wird das Cluster aus handelsüblichen PCs zusammengestellt, nennt man dieses ein *Beowulf Cluster*. Der Name stammt aus einem Projekt, das Donald Becker und Thomas Sterling am Center of Excellence in Space Data and Information Sciences (CESDIS) Mitte der 90er Jahre durchführten. Üblicherweise wird ein Prozess pro Prozessor gestartet. Werden auf den Nodes unterschiedliche Programme ausgeführt, spricht man von *Multiple Program Multiple Data* (MPMD). Meistens wird jedoch das *Single Program Multiple Data* (SPMD) Modell verwendet, wo dasselbe Programm auf mehreren Nodes ausgeführt wird. Die Rechner in einem Cluster müssen nicht zwangsläufig alle über Bildschirm und Tastatur verfügen. Sie können auch über das Netzwerk von einem *Master* aus administriert werden.

3.2 Parallele Verarbeitung

Ein Programm lässt sich nicht ohne Weiteres auf mehreren Rechnern gleichzeitig ausführen. Es ist vielmehr notwendig, dass sich die Aufgabe in mehrere Teilschritte zerle-

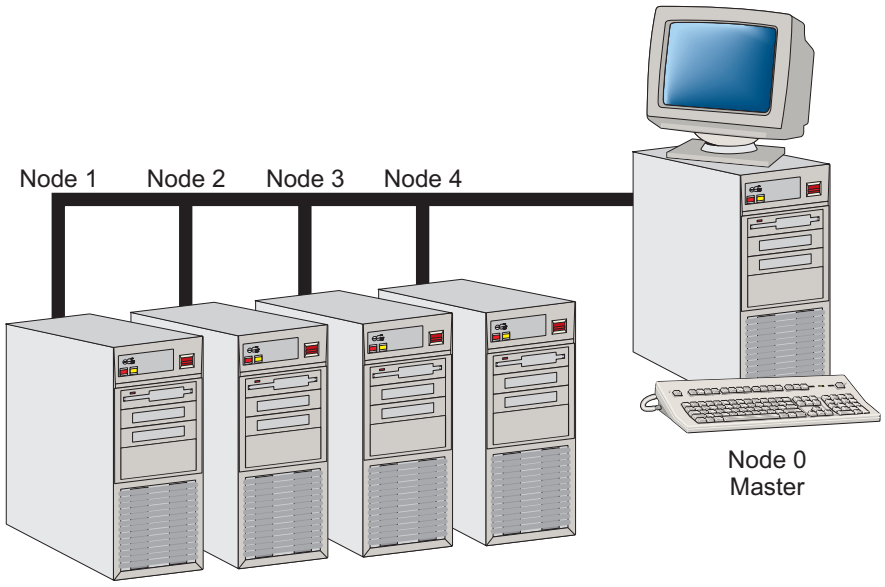


Abbildung 3.1: Rechnercluster (Illustration)

gen lässt, die sich dann unabhängig von einander auf mehreren Nodes bearbeiten lassen. Sollen z.B. zwei sehr lange Vektoren addiert werden, so können sich zwei Nodes die Arbeit teilen, indem ein Rechner nur die erste Hälfte der beiden Vektoren addiert und der andere die zweite (s. Abb. 3.2). Dadurch sollte sich theoretisch die Verarbeitungszeit für das Gesamtproblem halbieren lassen. Allerdings wird der Geschwindigkeitsgewinn durch zusätzliche Datenübertragungen erkauft: Die Teilvektoren müssen zunächst auf beide Rechner verteilt und das Endergebnis dann wieder zusammengesetzt werden. Das erzeugt Kommunikations-Overhead, so dass der mögliche Geschwindigkeitsgewinn reduziert wird. Oft bestehen außerdem Abhängigkeiten zwischen den Teilproblemen (engl. *task*s), so dass ein Prozess auf einen anderen warten muss, bis er dessen Daten weiterverarbeiten kann. Um ein optimales Ergebnis zu erhalten, muss man also stets einen Kompromiss zwischen Rechenleistung und zu übertragender Datenmenge finden und die Tasks möglichst geschickt verteilen.

3.3 Message Passing Software

Um den Datenaustausch in einem parallelen Rechnercluster zu ermöglichen, kann man auf eine fertige Software zurückgreifen. MPI (*Message Passing Interface*) und PVM (*Parallel Virtual Machine*) haben sich dabei als Quasi-Standard etabliert. Beide stehen für eine große Anzahl an Computer-Platformen zur Verfügung und können in C, C++ oder FORTRAN programmiert werden. Die bekanntesten MPI Implementierungen sind MPICH (*MPI Chameleon*) und LAM (*Local Area Multicomputer*). Obwohl PVM Nachrich-

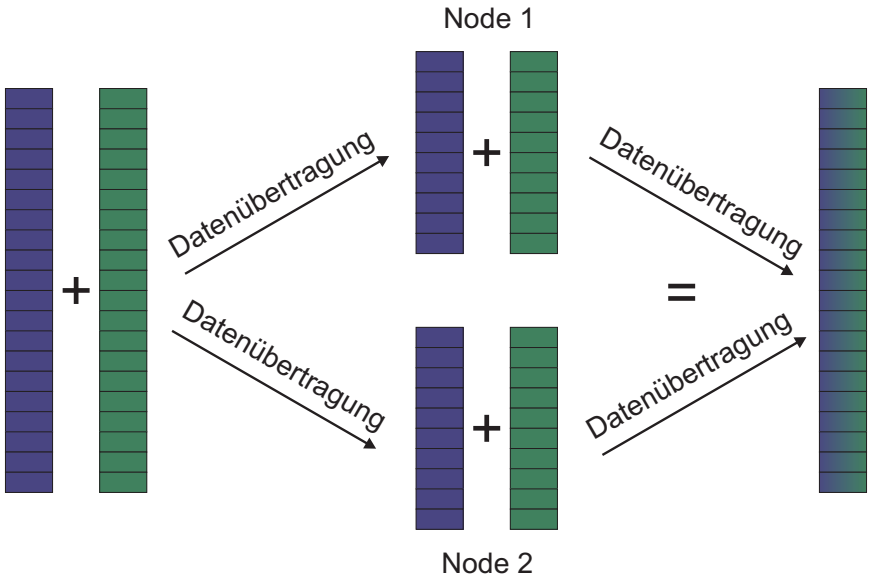


Abbildung 3.2: Parallele Addition von zwei Vektoren

ten Scheduling und auch noch andere zusätzliche Funktionen unterstützt, ist MPI weiterhin auf dem Vormarsch, da es einen einheitlich spezifizierten Standard bietet, der auf vielen Platformen konsequent umgesetzt ist. So läßt sich der für ein Beowulf Cluster generierte MPI-Programmcode ohne Änderungen auf einem Supercomputer ausführen.

Da auch die Rechnercluster am *Leibniz Rechenzentrum*[1] über eine MPI-Schnittstelle verfügen, wurde dieser Implementierung der Vozug gegeben. So können rechenintensive Simulationen ggf. dort durchgeführt werden.

Als Programmiersprache wurde C++ mit der Bibliothek von MPICH ausgewählt.

3.3.1 Kompilieren und Starten

Das Kompilieren der Software gestaltet sich bei MPI relativ einfach. Sowohl MPICH als auch LAM enthalten ein vorbereitetes Skript `mpiCC`, das den Compiler mit den nötigen Parametern aufruft. Es muss lediglich die Bibliothek `mpi.h` in das Programm eingebunden werden, damit die Funktionen von MPI zur Verfügung stehen. Sollen verschiedene Rechnerplattformen in einem Cluster kombiniert werden, so muss das Programm für jede davon einzeln übersetzt werden. Dann ist es sogar möglich, Sun Workstations unter Solaris und Intel Rechner unter Linux zu kombinieren.

Aufgerufen wird das Programm `[name]` mit

[1] http://www.lrz-muenchen.de

```
mpirun -np [n] [name]
```

wobei `[n]` die Anzahl der Prozesse spezifiziert, die gestartet werden sollen. Dazu wird eine Konfigurationsdatei `machines` ausgelesen, die alle im Cluster zur Verfügung stehenden Nodes enthält. Diese wird von oben nach unten abgearbeitet: Der erste Prozess wird auf dem Rechner gestartet, auf dem `mpirun` aufgerufen wurde (man nennt diesen dann *Master*), danach erhält der oberste Rechner in der Liste den zweiten Prozess, der nächste den dritten, usw. Werden mehr Prozesse gestartet als Rechner in der Liste stehen, wird die Liste mehrfach durchlaufen. Zusätzlich kann angegeben werden, wenn ein Rechner über mehr als einen Prozessor verfügt; er erhält dann entsprechend mehr Prozesse zugewiesen.

3.3.2 Senden und Empfangen von Daten

Beim Starten öffnet der Master (der Node, auf dem `mpirun` aufgerufen wurde) eine SSH-Verbindung[2] zu den anderen Rechnern. Diese werden dadurch angewiesen, das gewünschte Programm zu starten. Jedem Prozess ist dabei ein eindeutiger *Rang* r im Cluster zugeordnet, wobei der Master den Rang $r = 0$ erhält. Mit Hilfe der Funktion `MPI_Comm_rank()` kann ein Node festellen, an welcher Stelle er im Cluster steht.

Dies ist wichtig, wenn Daten nur zwischen zwei bestimmten Rechnern ausgetauscht werden sollen. Der sendende Node muss dann angeben, welchen Rang der Zielrechner haben muss. Dieser kann angewiesen werden, Daten nur von einem ganz bestimmten Rechner zu empfangen. Die wichtigsten Funktionen zum Senden und Empfangen werden nachfolgend beschrieben:

- `MPI_Send()`, `MPI_Receive()`: Diese Funktionen werden verwendet, wenn zwei einzelne Rechner über eine Point-to-Point Verbindung Daten austauschen sollen. Absender und Empfänger müssen über den Rang spezifiziert werden. Damit unterschiedliche Datensätze unterschieden werden können, kann der Funktion außerdem ein *Tag*, also eine Kennzeichnung[3], übergeben werden. Nur wenn der Tag beim Senden und Empfangen übereinstimmt, findet eine Kommunikation statt.
- `MPI_Bcast()`: Sind die zu übermittelnden Daten an alle Rechner im Cluster gerichtet, kann mittels `MPI_Bcast()` ein Rundruf (engl. *Broadcast*) durchgeführt werden, so dass die Daten danach auf jedem Node zur Verfügung stehen.
- `MPI_Reduce()`: Auch einfache Berechnungen können mit MPI ausgeführt werden. Soll zum Beispiel das Maximum eines Vektors bestimmt werden, dessen

[2] es könnte auch die wesentlich schnellere RSH-Verbindung verwendet werden, was jedoch aus Sicherheitsgründen nicht möglich war.

[3] der Tag besteht aus aus einer `integer`-Zahl.

Komponenten auf alle Nodes verteilt sind, kann dies mittels `MPI_Reduce()` geschehen. Die Berechnung wird dann in einem Baumverfahren zeitgleich mit der Datenübertragung durchgeführt: Zwei Rechner vergleichen dabei paarweise ihre Daten und bestimmen das jeweilige Maximum, das dann beiden zur Verfügung steht. Danach synchronisieren diese ihre Ergebnisse mit anderen Nodes, usw. So wird schrittweise das Maximum im gesamten Cluster bestimmt, wobei bei vielen Rechnern mehrere Datenübertragungen zugleich ablaufen können. Außerdem muss bei dem genannten Baumverfahren nicht jeder Node mit jedem anderen kommunizieren, so dass das Endergebnis bereits nach wenigen Schritten vorliegt[4]. Neben dem Maximum können auch Summe, Produkt, Minimum und auch bitweise Operationen wie UND, ODER, bzw. XOR durchgeführt werden.

- `MPI_Allreduce()`: Während das Endergebnis bei `MPI_Reduce()` nur einem Zielrechner zur Verfügung steht, wird es durch `MPI_Allreduce()` auf alle Nodes verteilt. Der Befehl fasst also `MPI_Reduce()` und `MPI_Bcast()` zusammen und spart durch die Verwendung des Baumverfahrens Übertragungszeit ein.

MPI bietet noch eine Vielzahl weiterer Funktionen, die jedoch nicht erläutert werden sollen, da sie für die vorliegende Arbeit nicht von Bedeutung sind. [WL]

3.4 Die parallele RSOM

Im folgenden soll gezeigt werden, wie eine (rekurrente) selbstorganisierende Karte auf mehrere Nodes verteilt werden kann[5]. Es wird dabei davon ausgegangen, dass auf jedem Node der Einfachheit halber nur ein Prozess gestartet wird.

Eine RSOM besteht aus mehreren Neuronen, die auf einer zweidimensionalen Karte der Größe $xs \cdot ys$ angeordnet sind. Jedes Neuron ist durch seine Position (xp, yp) in der Karte eindeutig bestimmt, wobei für die Position nur ganzzahlige Werte zugelassen werden sollen. Es ergibt sich also eine rechteckige Anordnung (s. Abb. 3.3). Jedem Neuron soll zusätzlich ein Index nr zugeordnet werden, der sich aus den Koordinaten errechnen lässt:

$$nr = yp \cdot xs + xp \tag{3.1}$$

Zu jedem Neuron ist ein zugehöriger Referenzvektor $\boldsymbol{\varrho}_{nr}$ gespeichert, der sich aus den Gewichten w_i zusammensetzt. Die Aufgabe eines Neurons besteht darin, den eu-

[4] Es wäre interessant gewesen, dieses Baumverfahren näher zu untersuchen, jedoch konnte dazu keine Funktionsbeschreibung gefunden werden.

[5] wird der Parameter $\alpha = 1$ gesetzt, sind SOM und RSOM identisch. Die nachfolgende Beschreibung bezieht sich auf eine rekurrente selbtorganisierende Karte auch wenn dies nicht immer explizit angegeben wird.

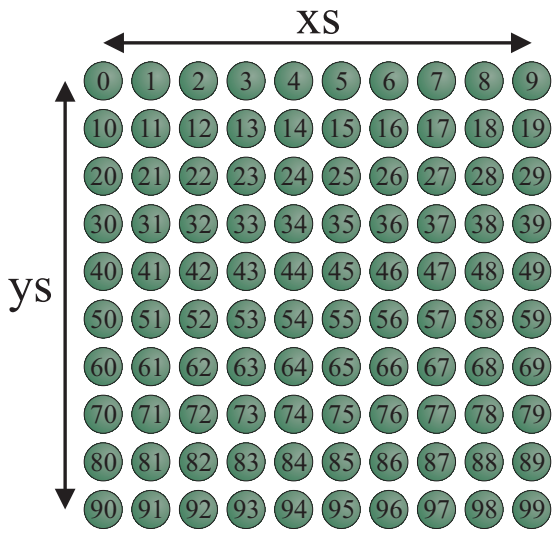


Abbildung 3.3: RSOM mit 100 Neuronen

klid'schen Abstand d_E seines eigenen Referenzvektors zu einem gegebenen Trainingsvektor $\boldsymbol{\xi}(t)$ zu bilden. Der Abstand ist ein Maß für die Aktivierung ψ_{nr} des Neurons nr. Diese Rechenoperation muss von allen Neuronen durchgeführt geführt werden, wobei aber die Reihenfolge unwichtig ist. Es liegt also nahe, die Aufgabe der Abstandsberechung auf mehrere Nodes zu verteilen, so dass jeder Rechner nur einen Teilbereich der RSOM verwalten muss. Hat eine RSOM $n_{ges} = xs \cdot ys$ Neuronen, so erhält jeder Prozess (gekennzeichnet durch seinen Rang r)[6] zunächst

$$n_r = \frac{xs \cdot ys}{np} \tag{3.2}$$

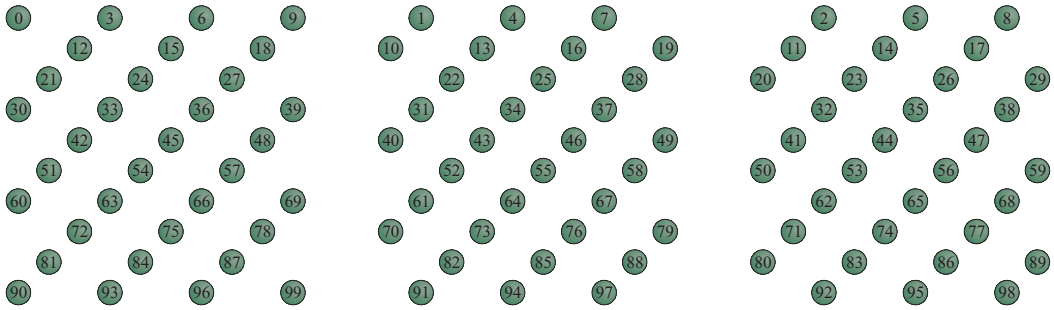
Neuronen, wobei np die Anzahl der gestarteten Prozesse bedeutet. Einem Node r werden genau die Neuronen i zugewiesen, für die gilt:

$$i \mod np = r, \tag{3.3}$$

wobei r den Rang des Rechners im Cluster angibt und $\mod$ eine Modulo-Division durchführt. Dadurch ist gewährleistet, dass jeder Node etwa gleich viele Neuronen erhält; der Unterschied beträgt dann maximal ein Neuron, wenn die Gesamtzahl der Neuronen nicht ohne Rest durch np teilbar ist. Ein Beispiel für $np = 3$ ist in Abbildung 3.4 dargestellt.

Die einzelnen Neuronen sind dabei als Objekte realisiert, die durch ihre Eigenschaften charakterisiert sind:

[6] vgl. 3.3.2



(a) Master $r = 0$ (b) Slave $r = 1$ (c) Slave $r = 2$

Abbildung 3.4: Verteilung der Neuronen auf drei Nodes

- `id`: eindeutiger Index des Neurons in der Karte
- `wcount`: Anzahl der Dimensionen im Referenzvektor
- `alpha`: "Gedächtnisfaktor"
- `xpos`: Position auf der x-Achse
- `ypos`: Position auf der y-Achse
- `activation[wcount]`: Komponenten des Ausgangsvektors ψ
- `euclidactivation`: euklid'scher Abstand zum aktuellen Trainingsvektor
- `weights[wcount]`: Gewichte des Referenzvektors

Ein Neuron implementiert außerdem Funktionen, um die Aktivierung für einen Trainingsvektor zu berechnen und die eigenen Gewichte anzupassen.

3.4.1 Programmablauf

Da jeder Node also nur einen Teil der Neuronen aus der RSOM enthält, müssen Datenpakete zwischen ihnen übertragen werden. Am leichtesten ist dies aus dem Nachrichtenflussdiagramm in Abbildung 3.5 zu entnehmen. Im Beispiel ist ein Cluster mit lediglich drei Nodes dargestellt, um die Übersicht zu wahren.

- **Schritt 1**: Das Programm wird vom Master $r = 0$ aus auf allen Nodes gestartet.
- **Schritt 2**: Jeder Node $0 \leq r < np$ reserviert Speicher für Referenzvektoren und Neuronen. Anhand seines Ranges kann er feststellen, welche Neuronen er erhalten soll.
- **Schritt 3**: Jeder Node $0 \leq r < np$ erstellt die Objekte für "seine" Neuronen und initialisiert ihre Gewichte mit zufälligen Werten.

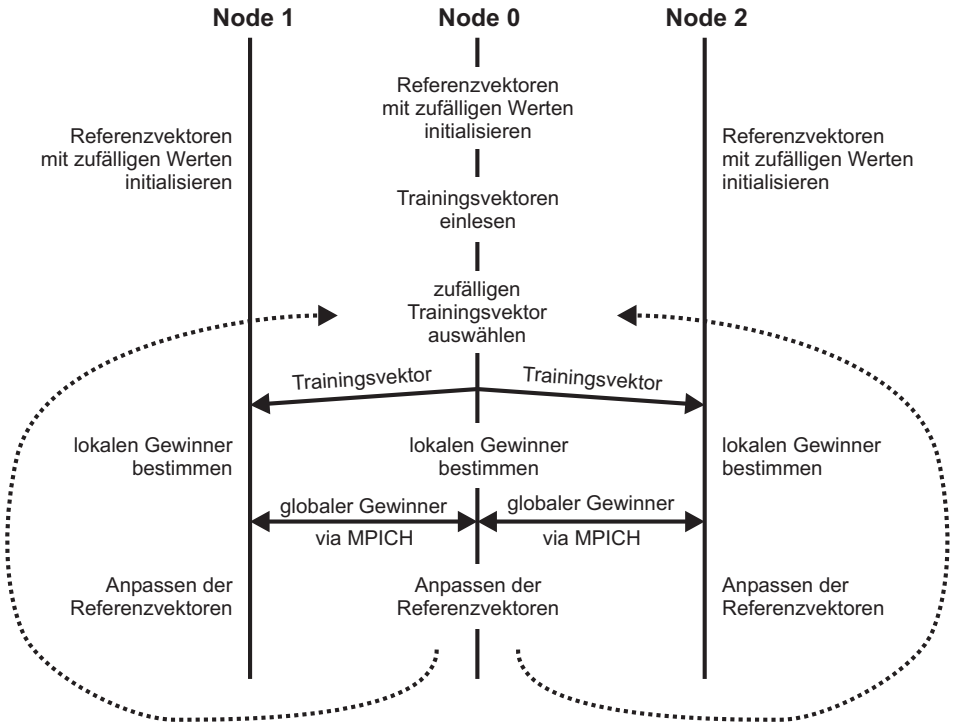


Abbildung 3.5: Nachrichtenflussdiagramm

- **Schritt 4**: Die Trainingsvektoren werden vom Master aus einer Datei eingelesen.
- **Schritt 5**: Der Master $r = 0$ wählt einen zufälligen Trainingsvektor $\boldsymbol{\xi}(t)$ aus und sendet ihn mit `MPI_Bcast()` an alle Slaves $0 < r < np$.
- **Schritt 6**: Jeder Node $0 \leq r < np$ bestimmt den Gewinner unter seinen eigenen Neuronen. Dazu werden die Aktivierungswerte aller Neuronen auf dem Node berechnet, miteinander verglichen und das Minimum ausgewählt.
- **Schritt 7**: Mit `MPI_Allreduce()` wird der globale Gewinner im gesamten Cluster $0 \leq r < np$ ermittelt. Mit dem Parameter `MINLOC` berechnet MPI den minimalen Wert und liefert außerdem den zugehörigen Index zurück. Jeder Node r stellt der Funktion also den Index b_r und den zugehörigen Wert v_r seines Gewinners als $G_r = (i_r, v_r)$ zur Verfügung. MPI bestimmt dann den globalen Gewinner $G_g = \min\{G_1, G_2, ..., G_{np}\}$. Dieser steht danach allen Rechnern $0 \leq r < np$ zur Verfügung ohne dass er mit einem Broadcast erneut versendet werden müsste.
- **Schritt 8**: Aus dem Index i_b des Gewinners b lässt sich auch dessen absolute Position berechnen:

$$xp_b = i_b \mod xs \tag{3.4}$$

  und

$$yp_b = i_b/xs, \tag{3.5}$$

  wobei / eine Ganzahldivision ohne Rest ausführt.
- **Schritt 9**: Da nun die Position des Gewinners b bekannt ist, kann für jedes Neuron i der Abstand zu diesem berechnet werden:

$$d_E(i,b) = \sqrt{(xp_i - xp_b)^2 + (yp_i - yp_b)^2} \tag{3.6}$$

  Dieser lässt sich in die Nachbarschaftsfunktion einsetzen und man erhält:

$$h_{(i,b)} = \exp\left(\frac{-d_E(i,b)^2}{\sigma(t)^2}\right) \tag{3.7}$$

- **Schritt 10**: Nun können die Referenzvektoren aller Neuronen auf den Nodes $0 \leq r < np$ angepasst werden:

$$\boldsymbol{\varrho}_i(t+1) = \boldsymbol{\varrho}_i(t) + \eta(t) h_{(i,b)}(t)(\boldsymbol{\xi}(t) - \boldsymbol{\varrho}_i(t)) \tag{3.8}$$

Die Schritte 5 bis 10 werden iterativ bis zum Ende des Trainigs ausgeführt. Lernrate und Nachbarschaftsradius werden dabei stetig vermindert.

3.4.2 Verteilung der Neuronen

Jeder Rechner erhält zu Beginn eine bestimmte Anzahl von Neuronen aus der RSOM zugewiesen, so dass dann auf allen Nodes etwa gleich viele Neuronen vorhanden sind. Sind alle Rechner gleich leistungsfähig, lässt sich eine RSOM ohne Probleme simulieren. Anders sieht es aus, wenn unterschiedlich schnelle Nodes in einem Cluster kombiniert werden. Der schnellste Rechner wird dann sehr wahrscheinlich als erster mit der Bestimmung seines lokalen Gewinners bzw. mit dem Gewichteupdate fertig sein. Da die Datenübertragung nur gleichzeitig auf allen Rechnern stattfinden kann, muss jeder Node entsprechend seiner Geschwindigkeit unterschiedlich lange auf das Cluster warten, bis er den nächsten Schritt fortsetzen kann (s. Abb. 3.6).

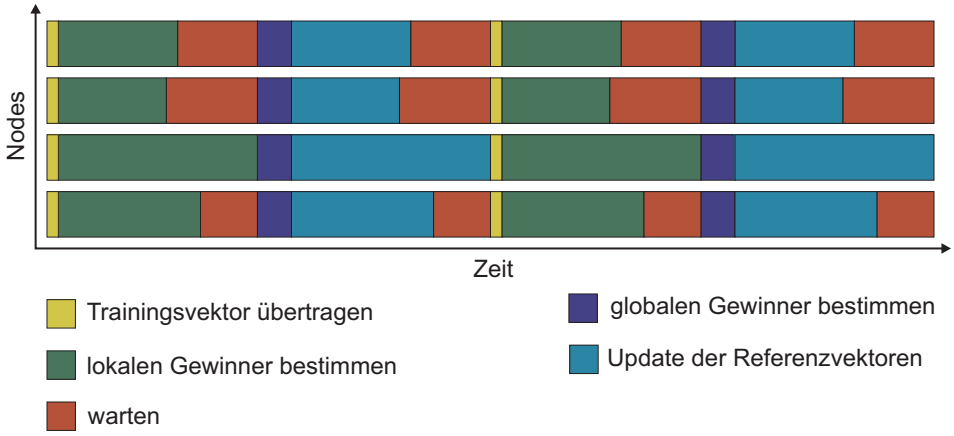


Abbildung 3.6: Parallele Verarbeitung bei schlechter Verteilung der Neuronen

Um diese Wartezeit zu minimieren, soll es zugelassen werden, dass einzelne Neuronen von einem langsamen Rechner auf einen schnelleren übertragen werden. Die Rechnerauslastung lässt sich dadurch aktiv beeinflussen (engl. *load balancing*).

Jeder Rechner berechnet für seine Neuronen den euklid'schen Abstand vom ermittelten Gewinnner und führt anschließend die Gewichtsanpassung unter Verwendung der aktuellen Lernrate und Nachbarschaftsfunktion durch. Über mehrere solcher Schritte lässt sich die Verarbeitungszeit mit der Funktion `MPI_Wtime()` messen und zwar aufgeteilt in Zeit zum Auffinden des lokalen Gewinners t_l, Zeit zur Bestimmung des globalen Gewinners t_g und die Dauer des Gewichteupdates t_u. Die beiden Werte t_l und t_u repräsentieren also die tatsächlich benötigte Rechenzeit auf dem Rechner, wohingegen t_g die Übertragungszeit misst, die wesentlich durch die Netzwerkanbindung beeinlusst wird. Auch wenn ein Node auf einen anderen warten muss, geht diese Zeit in t_g ein, da die Datenübertragung erst beendet wird, wenn jeder Rechner auf die Anfrage reagiert hat.

Auf jedem Node werden zur Gewinnerbestimmung bzw. zum Gewichteupdate alle Neuronen nacheinander abgearbeitet. Setzt man voraus, dass die für jedes Neuron auf-

gewendete Zeit konstant ist, hängt also die benötigte Verarbeitungszeit direkt mit der Anzahl der Neuronen auf dem Rechner zusammen. Wird ein Neuron von einem langsamen Rechner an einen schnellen abgegeben, sinkt die Wartezeit. Es muss also eine optimale Anzahl an Neuronen für jeden Node geben, so dass die Bearbeitungszeit auf allen Rechnern näherungsweise gleich groß und die Gesamtzeit demzufolge minimal ist (s. Abb. 3.7). In diesem Fall treten quasi keine Wartezeiten auf und es gilt:

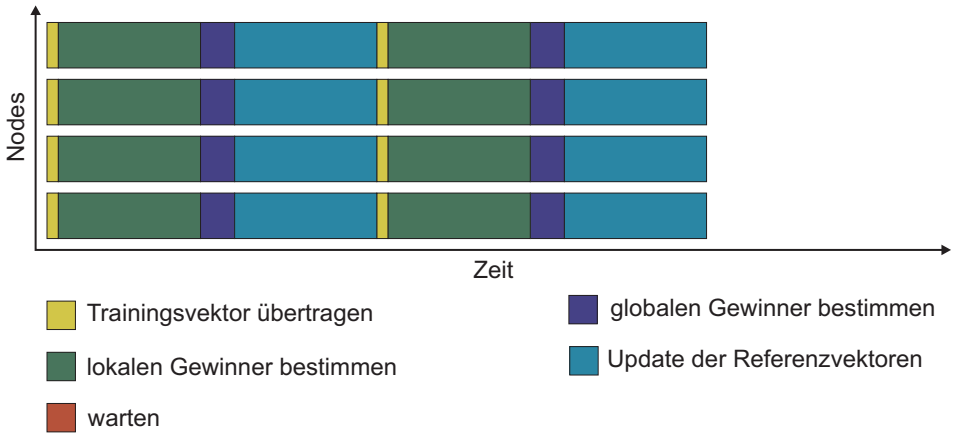


Abbildung 3.7: Parallele Verarbeitung bei optimaler Verteilung der Neuronen

$$t_{l,1} + t_{u,1} = t_{l,2} + t_{u,2} = ... = t_{l,np} + t_{u,np} \tag{3.9}$$

Um die entsprechende Anzahl an Neuronen zu bestimmen und entsprechend zu verteilen wurde ein iterativer Algorithmus verwendet:

- Sind auf einem Rechner n_r Neuronen vorhanden und benötigt er die Bearbeitungszeit $t_l + t_u$, so kann die Zeit für ein einzelnes Neuron angegeben werden durch
$$\Delta t = (t_l + t_u)/n_r \tag{3.10}$$
- Der Node r_{max} mit der längsten und der mit kürzesten Arbeitszeit r_{min} werden bestimmt.
- Ein einzelnes Neuron wird von r_{max} an r_{min} übertragen.
- Die neue Bearbeitungszeit wird geschätzt: Auf r_{max} ist sie gegeben durch
$$t_{r_{max,neu}} = (k-1) \cdot \Delta t \tag{3.11}$$
Für r_{min} gilt entsprechend:
$$t_{r_{min,neu}} = (k+1) \cdot \Delta t \tag{3.12}$$

- Es werden erneut die beiden längsten bzw. kürzesten Zeiten bestimmt - nun jedoch mit den geschätzten Werten.
- Erneut wird ein Neuron übertragen.

Die letzten drei Schritte werden wiederholt, bis die Geschwindigkeitsdifferenz zwischen langsamstem und schnellstem Node $t_{r_{max}} - t_{r_{min}}$ eine bestimmte Schwelle unterschritten hat.

Alternativ hätte man die Geschwindigkeiten der einzelnen Nodes durch einen Testlauf bestimmen und speichern können. Dann könnte jeder Rechner schon zu Beginn des Trainings die richtige Anzahl an Neuronen erhalten und es müsste kein Austausch mehr stattfinden. Der dynamischen Lösung wurde jedoch der Vorzug gegeben, da diese auch bei variierender Systemlast zu optimalen Ergebnissen führt.

3.4.3 Visualisierung

Wie aus 3.4.2 bekannt ist, kann eine Zeitmessung verwendet werden, um jedem Rechner die optimale Neuronenzahl zuzuweisen. Es ist möglich, diese Messungen zu verwenden, um auf dem Master eine Visualisierung zu veranlassen, die beispielsweise jede Sekunde aktualisiert wird. Dazu muss lediglich die Zeit gemessen werden, die für einen einzelnen Trainingsvektor benötigt wird. Daraus lässt sich errechnen, nach wie vielen Lernschritten die Visualisierung aktualisiert werden muss.

9926	+91	elapsed: 137	remain: 1	total: 138			
np	neurons	local	global	update	winner	value	vec
0	98	0.0015	0.0069	0.0030	887	0.7036	11
1	256	0.0014	0.0069	0.0031			
2	83	0.0015	0.0064	0.0033			
3	83	0.0014	0.0063	0.0033			
4	80	0.0015	0.0067	0.0031			
5	82	0.0014	0.0066	0.0031			
6	39	0.0016	0.0060	0.0032			
7	41	0.0015	0.0059	0.0032			
8	38	0.0016	0.0058	0.0031			
9	41	0.0015	0.0057	0.0032			
10	59	0.0014	0.0058	0.0032			
		162	72	346			

Abbildung 3.8: Visualisierung

Ein Beispiel für 11 Nodes ist in Abbildung 3.8 zu sehen. In der obersten Zeile werden links oben der aktuelle Lernschritt und daneben die Anzahl der Schritte bis zur Aktualisierung der Anzeige dargestellt. Danach folgen Angaben für vergangene, verbleibende

und die geschätzte Gesamtdauer in Sekunden. Die folgenden Zeilen enthalten die Daten der beteiligten Nodes: In der ersten Spalte ist der Rang, in der zweiten die Anzahl der Neuronen auf dem entsprechenden Rechner aufgetragen. So ist auch zu erkennen, welcher Rechner die größte Rechenleistung bietet (dieser hat die meisten Neuronen erhalten). Die drei folgenden Spalten zeigen die Messwerte für lokale Gewinnersuche t_l, globale Gewinnersuche t_g und Gewichteupdate t_u. Wie man sieht, sind die Zeiten t_l und t_u auf allen Rechnern nahezu konstant, wohingegen die Zeit zur globalen Gewinnersuche t_g stark variieren kann. Danach folgen die Anzeige des aktuellen Gewinnerneurons mit zugehörigem Wert und der Nummer des präsentierten Vektors. Die letzte Zeile enthält die Zeitsummen aller Nodes über die vergangen Lernschritte: $\sum t_l$, $\sum \max t_g$ und $\sum t_u$.

3.4.4 Zufallszahlen

Anstatt die Vektoren auf dem Master einzulesen und einzeln an die Slaves zu versenden, kann man - ausreichende Speicherkapazität vorausgesetz - auch alle Vektoren auf den Nodes speichern, um die Datenübertragung zu vermeiden. Dies würde einen weiteren Geschwindigkeitsgewinn bedeuten. Es stellt sich allerdings die Frage, wie gewährleistet werden kann, dass auf allen Nodes derselbe Trainingsvektor zufällig ausgewählt wird. Um auf einem Rechner Zufallszahlen zu erzeugen, gibt es prinzipiell mehrere Möglichkeiten:

- Man kann eine natürliche Quelle von "Chaos" verwenden. Hierfür kann z.B. das weiße Rauschen dienen, das aus einer Soundkarte gewonnen wird.
- Die Zufallszahlen werden mit einem Algorithmus berechnet. Dies ist problematisch, da ja Zufallszahlen nicht vorhersehbar sein sollen. Diese Eigenschaft kann bei einem Algorithmus selbstverständlich nicht gegeben sein.
- Es kann eine Datenbasis aus wirklichen Zufallszahlen verwendet werden. Voraussetzung hierfür ist eine große Speicherkapazität, damit eine genügend große Periode erreicht werden kann.

In der Regel wird als Kompromiss ein Algorithmus, ein sogenannter *Zufallsgenerator*, verwendet, um zufällige Zahlenfolgen zu erzeugen. In der Programmiersprache C wird dieser mit der Funktion `rand()` aufgerufen. Allerdings muss er zunächst mittels `srand()` mit einem Startwert initialisiert werden. Üblicherweise wird dazu die Zeit in Sekunden seit dem 1. Januar 1970 verwendet, was durch die Funktion `time(0)` geschieht. Da das Programm auf allen Rechnern gleichzeitig gestartet wird, ist dieser Wert überall identisch, so dass auch identische Zufallszahlen erzeugt werden. Für die Initialisierung der Referenzvektoren wäre es jedoch fatal, die gleichen Startwerte zu verwenden. In diesem Fall kann der modifizierter Aufruf

```
srand(time(0)+getpid());
```

verwendet werden, der die Prozessnummer des Programmes auf dem Node berücksichtigt. Diese ist in der Regel überall unterschiedlich. [Str93]

Ein weiteres Problem zeigte sich, als das Programm für mehrere unterschiedliche Rechnerarchitekturen übersetzt wurde. Es stellte sich heraus, dass die Funktion `rand()` auf unterschiedlichen Architekturen auch unterschiedlich realisiert ist, so dass verschiedene Zufallszahlen generiert wurden auch wenn dies nicht erwünscht war. Das Problem ließ sich jedoch relativ einfach lösen, indem die Funktion zur Generierung der Zufallszahlen explizit in den Programmcode eingebunden wurde. Es wurde dabei die Funktion aus [PTVF92] verwendet.

Kapitel 4

Ergebnisse und Diskussion

4.1 Rechnercluster am Lehrstuhl für TIKI

Für das Projekt standen am *Lehrstuhl für Theoretische Informatik und künstliche Intelligenz* (TIKI) einige ausgemusterte Rechner[1] zur Verfügung (s. Abb. 4.1). Diese wurden im Serverraum aufgestellt und über einen 3Com Switch an das Lehrstuhlnetz angebunden. Außerhalb der Arbeitszeit konnten außerdem die Rechner aus einem Praktikumsraum und Workstations einiger Mitarbeiter verwendet werden. Diese waren über eine Glasfaserstrecke vernetzt. Wie aus Tabelle 4.1 zu entnehmen ist, wurden also die

Rechnertyp	Prozessor	RAM	Betriebssystem
SPARCstation 20 Model 512	2x Model 50 SuperSPARC SPARCmodule	64 MB	SunOS 5.7
SPARCstation 20 Model 512	2x Model 50 SuperSPARC SPARCmodule	64 MB	SunOS 5.7
SPARCstation 20 Model 512	2x Model 50 SuperSPARC SPARCmodule	64 MB	SunOS 5.7
SPARCstation 20 Model 512	2x Model 50 SuperSPARC SPARCmodule	160 MB	SunOS 5.7
SPARCstation 20 Model 702	2x Model 71 SuperSPARC-II SPARCmodule	160 MB	SunOS 5.7
SPARCstation 20 Model 702	2x Model 71 SuperSPARC-II SPARCmodule	96 MB	SunOS 5.7
Sun Ultra 1	1x Model 140 UltraSPARC	128 MB	SunOS 5.7
Sun Ultra 1	1x Model 140 UltraSPARC	64 MB	SunOS 5.7
Sun Ultra 1	1x Model 140 UltraSPARC	96 MB	SunOS 5.7
Sun Ultra 1	1x Model 140 UltraSPARC	64 MB	SunOS 5.7
Sun Ultra 2 Model 2168	2x 168 MHz UltraSPARC	192 MB	SunOS 5.9
Sun Ultra 5/10	1x 440 MHz SUNW, UltraSPARC-IIi	256 MB	SunOS 5.7

Tabelle 4.1: Rechnerübersicht

unterschiedlichsten Leistungsklassen zu einem Cluster kombiniert. Da auch die Netz-

[1] mehrere Sun SPARCstation 20 und eine Sun Ultra 1

Abbildung 4.1: Rechnercluster am Lehrstuhl für TIKI

werkanbindung nicht einheitlich war[2], war es äußerst schwierig, die Leistungsfähigkeit des Clusters objektiv zu beurteilen; auch die momentane Netzwerkauslastung spielte dabei eine Rolle.

4.2 Funktionstest der Software

Zunächst soll allerdings die Funktionsfähigkeit der Software untersucht werden, ohne dass dabei der Geschwindigkeitsaspekt im Vordergrund steht. Zusätzlich zur in Kapitel 3.4.3 beschriebenen Visualisierung wurde vom Master-Node jede Sekunde die Aktivierung ψ_i aller Neuronen i bestimmt. Diese wurde als Matrix in einer tabulatorseparierten Datei abgespeichert, so dass die Antwort des Netzes auf einen gegebenen Trainingsvektor $\boldsymbol{\xi}(t)$ beobachtet werden konnte. Mit der Software Matlab® der Firma MathWorks[3] ließ sich die Netzaktivierung übersichtlich in einem dreidimensionalen Koordinatensystem darstellen. Dadurch konnte das Lernverhalten des Netzes für eine bestimmte Aufgabenstellung quasi in Echtzeit beobachtet werden.

Bei Trainingsbeginn sind bekanntlich alle Gewichte mit zufälligen Werten initialisiert. Die Aktivierung des gesamten Netzes für den ersten Merkmalsvektor $\boldsymbol{\xi}(t = 0)$ ist damit auch ungeordnet, wie in Abb. 4.2 deutlich zu erkennen ist.

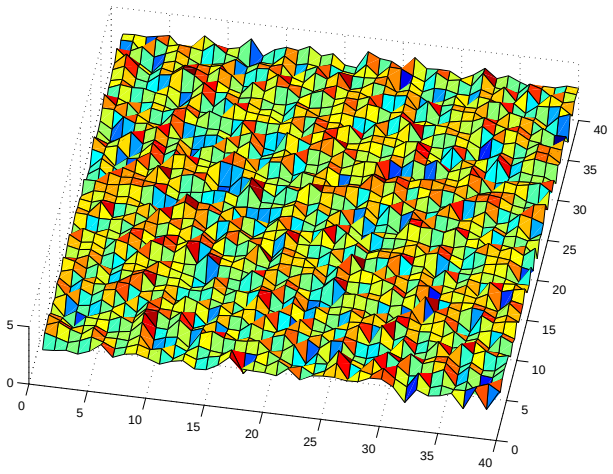


Abbildung 4.2: Ungeordnete Aktivierung

Dargestellt ist ein 40x40 Netz mit 1600 Neuronen, die in der Ebene x-y angeordnet sind. Die Aktivierung der einzelnen Neuronen wurde auf der z-Achse aufgetragen und entsprechend ihrem Wert in unterschiedlichen Farben gekennzeichnet. Blaue Flächen be-

[2] die SPARCstation 20 verfügen über eine 10 MBit Netzwerkkarte, alle anderen Rechner sind mit 100 MBit angebunden.

[3] http://www.matlab.com

deuten niedrige Abstandswerte vom Referenzvektor $\boldsymbol{\varrho}_{(i)}(t)$ zum Referenzvektor $\boldsymbol{\xi}(t)$. Ist die Differenz größer, wird dies durch eine rote oder gelbe Färbung verdeutlicht.

Nach der Präsentation des ersten Trainingsvektors werden die Gewichte der Neuronen adaptiert. Da die Lernrate zu Beginn einen hohen Wert hat[4], sind nun viele Neuronen in der Nähe des Gewinners sehr gut an diesen Vektor angepasst[5]. In Abb. 4.3 ist dies deutlich als großes Minimum (blaue Farbe) zu erkennen. Wird nun ein anderer Trai-

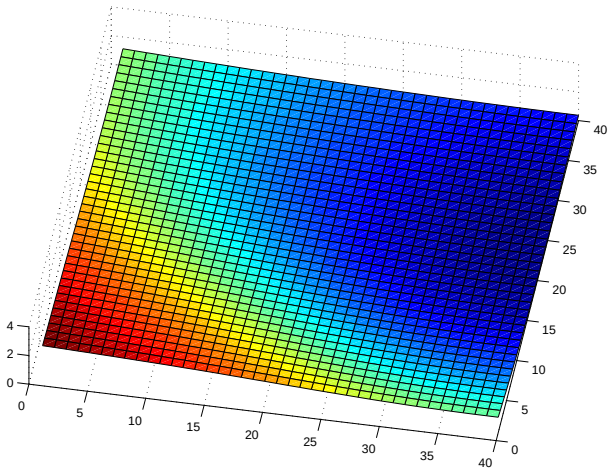


Abbildung 4.3: Netzaktivierung nach nur wenigen Lernschritten

ningsvektor präsentiert, so liegt das Minumum sehr wahrscheinlich an einer anderen Position. Es fällt auf, dass die Minima zu Beginn der Trainingsphase eher am Rand der SOM angeordnet sind. Bei großer Lernrate und weit gefasster Nachbarschaftsfunktion liegen die Gewinnerneuronen dadurch möglichst weit auseinander. Erst wenn Lernrate und Radius der Nachbarschaftsfunktion zu Trainingsende kleinere Werte annehmen, rücken die Minima auch in die Mitte der SOM vor, so dass am Ende tatsächlich die topologischen Beziehungen der einzelnen Merkmalsvektoren untereinander zu erkennen sind.

4.3 Speicherbedarf

Es sei erwähnt, dass sich durch die Verwendung eines Rechnerclusters nicht nur die Trainingszeit für eine selbstorganisierende Karte verkürzen lässt, sondern dass sich auch wesentlich größere Karten simulieren lassen, als dies auf einem einzelnen Rechner möglich wäre. Dies liegt daran, dass die Neuronen auf mehrere Rechner verteilt werden

[4] zu Trainingsbeginn wurde eine Lernrate von $\eta(t=0) = \eta_{max} = 0.9$ gewählt.

[5] auch der Radius für die Nachbarschaftsfunktion ist anfangs sehr groß und erstreckt sich mit $\sigma(t = 0) = \sigma_{max} = 20.0$ über die halbe Karte.

und so jeder Node nur einen Teilbereich der SOM speichern muss. Gerade bei Nodes mit wenig Arbeitsspeicher und großen Karten lässt sich so ein entscheidender Vorteil gegenüber einem einzelnen Rechner erzielen. Der Speicherbedarf wird dabei wesentlich von der Anzahl der Neuronen bestimmt, die auf einem Node gespeichert sind. Dieser hängt hauptsächlich von der Größe eines Trainingsvektors $\boldsymbol{\xi}(t)$ ab. Für jede Komponente ξ_i muss ein Gewicht w_i im Referenzvektor $\boldsymbol{\varrho}_{(j)}(t)$ des Neurons j gespeichert werden, das sich durch einen `double` Wert (8 Byte) darstellen lässt. Für die RSOM muss außerdem die Aktivierung im vorangegangenen Lernschritt zwischengespeichert bleiben, was für jede Komponente ebenfalls jeweils 8 Byte benötigt. Zusätzlich werden für jedes Neuron der Index (8 Byte[6]), der Faktor α (8 Byte), die Anzahl der Komponenten im Gewichtsvektor (4 Byte), die Position (x und y, jeweils 8 Byte) und der euklid'sche Abstand zum letzten Referenzvektor (8 Byte) benötigt. Damit ergibt sich für den gesamten Speicherbedarf S folgende Formel:

$$S = (44 + m \cdot 16) \cdot n_{ges} \text{ Byte} \tag{4.1}$$

wobei n_{ges} für die Anzahl der Neuronen steht und m die Dimension des Trainingsvektors angibt. Eine 100x100 SOM würde also bei Trainingsvektoren $\boldsymbol{\xi}(t) \in \mathbb{R}^{100}$ auf einem einzelnen Rechner mehr als 16 MByte in Anspruch nehmen; wird ein Cluster mit 10 Rechnern verwendet, werden auf jedem Rechner nur noch unter 2 MByte benötigt[7].

Da auf den Testrechnern ein beträchtlicher Teil des Speichers für das Betriebssystem verwendet wurde, ließen sich auf den Sun SPARCstation 20 nur jeweils ca. 10 MByte im Hauptspeicher ablegen, ohne dass Speicherbereiche auf die Fesplatte ausgelagert werden mussten. Für die nachfolgenden Anwendungen war dies jedoch ausreichend.

4.4 Tests mit Beispieldaten

4.4.1 Tiermerkmale

Ein wesentliches Merkmal der SOM ist die Fähigkeit, zweidimensionale Karten zu erzeugen, die semantische Beziehung zwischen Objektbegriffen widerspiegeln. Ein gutes und einfaches Beispiel, um dies zu testen, stellt das aus [Koh01] bekannte Modell von Tiermerkmalen dar. Zu jedem Tier wird dabei ein Merkmalsvektor gebildet, dessen

[6] Der Speicherbedarf kann je nach verwendeter Rechnerarchitektur schwanken. Es sind jeweils die Maximalwerte angegeben, die für ein 64-bit System Geltung haben, vgl. [Str93].

[7] Es lassen sich nur Schätzungen über den tatsächlichen Speicherbedarf anstellen, da auch der Programmcode selbst in den Hauptspeicher geladen werden muss. Dies ist jedoch sehr stark abhängig vom verwendeten Betriebssystem bzw. vom Compiler und dessen Einstellungen. Die oben genannte Formel kann also lediglich die Größenordnung des erforderlichen Speicherplatzes angeben.

Komponenten entweder aus einer 0 (Merkmal nicht vorhanden) oder einer 1 (Merkmal vorhanden) bestehen. In Tabelle 4.2 sind 16 Tiere mit ihren Eigenschaften aufgelistet.

	klein	mittel	groß	2 Beine	4 Beine	Haare	Hufe	Mähne	Federn	jagt	rennt	fliegt	schwimmt
Taube	1	0	0	1	0	0	0	0	1	0	0	1	0
Henne	1	0	0	1	0	0	0	0	1	0	0	0	0
Ente	1	0	0	1	0	0	0	0	1	0	0	0	1
Gans	1	0	0	1	0	0	0	0	1	0	0	1	1
Eule	1	0	0	1	0	0	0	0	1	1	0	1	0
Falke	1	0	0	1	0	0	0	0	1	1	0	1	0
Adler	0	1	0	1	0	0	0	0	1	1	0	1	0
Fuchs	0	1	0	0	1	1	0	0	0	1	0	0	0
Hund	0	1	0	0	1	1	0	0	0	0	1	0	0
Wolf	0	1	0	0	1	1	0	1	0	1	1	0	0
Katze	1	0	0	0	1	1	0	0	0	1	0	0	0
Tiger	0	0	1	0	1	1	0	0	0	1	1	0	0
Löwe	0	0	1	0	1	1	0	1	0	1	1	0	0
Pferd	0	0	1	0	1	1	1	1	0	0	1	0	0
Zebra	0	0	1	0	1	1	1	1	0	0	1	0	0
Kuh	0	0	1	0	1	1	1	0	0	0	0	0	0

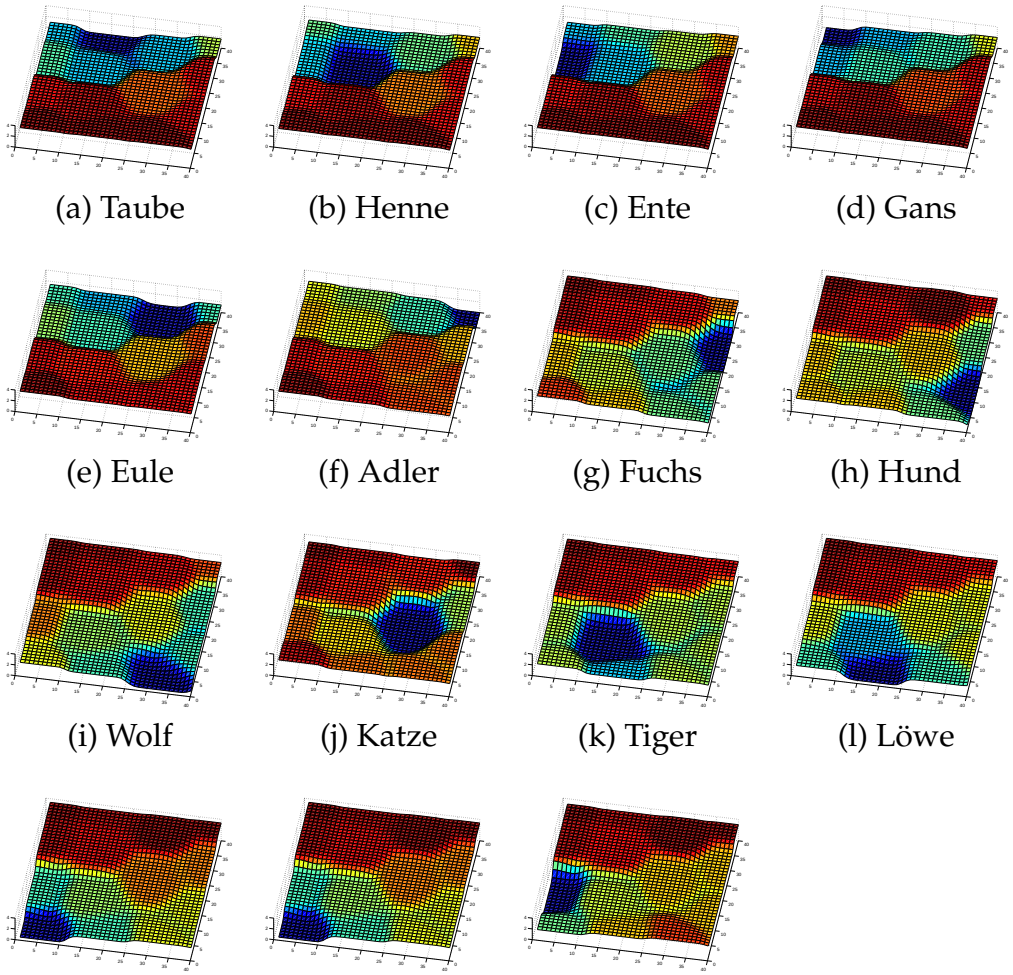
Tabelle 4.2: Matrix der Tiermerkmale

Ein Trainingsvektor ist mit seinen 13 Komponenten vergleichsweise klein und daher gut für einen Funktionstest der Software geeignet. Es wurden quadratische Netze unterschiedlicher Größe[8] mit diesem Trainingsset aus 16 Vektoren trainiert. Der Faktor α wurde dabei auf den Wert 1 gesetzt, da es sich um Daten ohne temporale Zusammenhänge handelte. Dazu wurden die Vektoren dem Netz in zufälliger Reihenfolge mehrere tausend Male präsentiert. Die Lernrate $\eta(t)$ wurde dabei von 0.9 auf 0.1 linear abgesenkt[9]. Als Nachbarschaftsradius $\sigma(t)$ wurde zu Trainingsbeginn die halbe Seitenlänge des Netzes angesetzt und dann ebenfalls linear bis auf 1 vermindert[10]. Als Nachbarschaftsfunktion wurde sowohl die Gauß-Funktion als auch die Bubble-Funktion verwendet.

Bereits nach einigen tausend Trainingsvektoren war die SOM fähig, diese auf unterschiedliche Positionen in der Karte abzubilden, die entsprechend der Ähnlichkeit der Tiere in benachbarten Regionen zu liegen kamen. Abb. 4.4 zeigt die Aktivierung des Netzes für jeden Trainingsvektor nach 30000 Lernschritten.

Wie man erkennen kann, lässt sich die Karte in deutliche Bereiche einteilen, die sich mit Bezeichnungen wie “Huftiere” (unten links) oder “Vögel” (oben) identifizieren lassen.

[8] 10x10 bis 80x80
[9] vgl. 2.5.2.3
[10] vgl. 2.5.4.2



(a) Taube (b) Henne (c) Ente (d) Gans

(e) Eule (f) Adler (g) Fuchs (h) Hund

(i) Wolf (j) Katze (k) Tiger (l) Löwe

(m) Pferd (n) Zebra (o) Kuh

Abbildung 4.4: Aktivierung des Netzes für Vektoren aus Tiermerkmalen

Das Netz hat also die Ähnlichkeiten der Vektoren im 13-dimensionalen Raum beibehalten und diese auf eine zweidimensionale Karte abgebildet.

4.4.1.1 Bewertung

Der Lernerfolg ließ sich anhand dieses einfachen Beispiels leicht visuell überprüfen und qualitativ beurteilen. Eine quantitative Bewertung war über den Quantisierungsfehler E_q möglich. Bei einer perfekten Abbildung der Trainingsvektoren ist der Quantisierungsfehler zu Trainingsende 0. Der Referenzvektor $\boldsymbol{\varrho}_{(b)}(t_{max})$ eines jeden Gewinnerneurons b ist dann identisch zu einem Trainingsvektor $\boldsymbol{\xi} \in \mathcal{T}$. Beim Training mit unterschiedlichen Parametern[11] konnten folgende Beobachtungen gemacht werden:

- Die Minima für unterschiedliche Trainingsvektoren liegen zu Trainingsbeginn bevorzugt am Rand der Karte und rücken erst später in die Mitte vor.
- Die Minima sind zu Trainingsende gleichmäßig über die gesamte Kartenfläche verteilt und haben zueinander den maximal möglichen Abstand.
- Offensichtlich ist die Gauß-Funktion besser geeignet als die Bubble-Funktion, da sich in der Karte keine harten Kanten, sondern weiche Übergänge ausbilden. Dies lässt vermuten, dass die Generalisierungsleistung bei unbekannten Daten besser ist.
- Ein wesentlicher Unterschied zwischen linearer oder exponentieller Absenkung von Lernrate und Nachbarschaftsradius konnte nicht beobachtet werden.
- Damit das Netz gut trainiert ist (Quantisierungsfehler $E_q = 0$), sollte jeder Trainingsvektor ca. 1000 Mal präsentiert werden.
- Je langsamer Lernrate und Nachbarschaftsradius vermindert werden, je länger also das Training andauert, desto besser ist das Endresultat. Allerdings gibt es für das verwendete Beispiel eine obere Grenze von 30000 Lernschritten. Bei noch mehr Trainingsschritten ist keine Verbesserung mehr zu erkennen.
- Die Karte sieht nach dem Training nicht immer gleich aus. Der Endzustand hängt wesentlich von der zufälligen Initialisierung der Gewichtsvektoren ab und ist daher nicht vorhersehbar. Bei den Tiermerkmalen gab es allerdings einige bevorzugte Topologien, die jedoch bei einer quadratischen Karte auch um 90, 180 oder 270 Grad gedreht auftreten konnten. Auch eine horizontale bzw. vertikale Spiegelung entspricht derselben Topologie, da der euklid'sche Abstand gegenüber diesen Transformationen invariant ist.

[11] Als Nachbarschaftsfunktion $h_{(i,b)}$ wurde abwechselnd die Gauß- und Bubble-Funktion verwendet. Lernrate $\eta(t)$ und Nachbarschaftsradius $\sigma(t)$ wurden linear bzw. exponentiell mit variierenden Start- und Endwerten abgesenkt.

Damit war gezeigt, dass die Software einwandfrei arbeitete und einfache Aufgaben zufriedenstellend lösen konnte.

4.4.2 Handschrifterkennung

Ein Trainingsset bestehend aus 16 Vektoren $\in \mathbb{R}^{13}$ stellt eine sehr unrealistische Anwendung für eine selbstorganisierende Karte dar. Wesentlich besser geeignet sind Daten einer realen Anwendung. Aus der Diplomarbeit meines Betreuers standen mir Testdaten für eine Handschrifterkennung zur Verfügung[12]. Der Datensatz bestand aus insgesamt 20000 von Hand geschriebenen Ziffern, die jeweils als 16x16 Bitmap vorlagen. Ein Pixel konnte dabei diskrete Werte zwischen 0 (weiß) und 255 (schwarz) annehmen. Die Ziffern waren bereits bezüglich Größe, Stiftdicke und Winkel normiert. Eine weitere Vorverarbeitung hatte jedoch nicht stattgefunden (s. Abb. 4.5). [Bai98]

4.4.2.1 Skalierung der Daten

Um die Daten für die SOM verwenden zu können, wurde jede Ziffer in einem 256-dimensionalen Vektor zusammengefasst. Die Daten wurden in ein Trainings- und Testset aufgeteilt, das je 10000 Datensätze enthielt. Weiterhin musste eine Skalierung der Grauwerte vorgenommen werden. Da jedoch der Speicherbedarf für die skalierten Werte (`double`) wesentlich höher ist, als für die ursprünglichen (`char`), wurde die Software um eine Skalierungsfunktion erweitert. So war es möglich, alle Vektoren auf einmal einzulesen, und auf allen Rechnern permanent zu speichern. Zur Speicherung eines `char`-Wertes reicht ein einzelnes Byte aus und der erforderliche Speicherplatz beträgt dann:

$$1\,\text{Byte} \cdot 256 \cdot 10000 = 2560000\,\text{Byte} = 2.44\,\text{MByte} \tag{4.2}$$

Zur Speicherung von `double`-Werten wäre der 8-fache Speicher erforderlich gewesen, was insbesondere auf den älteren SPARCstation 20 zu Problemen geführt hätte. Alternativ wäre es auch möglich gewesen, alle Vektoren als `double`-Werte auf einem Master-Rechner mit viel Hauptspeicher einzulesen und dann für jeden Lernschritt lediglich einen Vektor an alle Nodes zu übertragen. Die Übertragung hätte jedoch wesentlich länger gedauert, als jeden Vektor bei Bedarf zu skalieren[13]. Es musste lediglich sicher gestellt werden, dass auf allen Rechnern die gleichen Vektoren zum Training ausgewählt wurden[14].

[12] Diese Daten stammen vom Daimler-Benz Forschungszentrum und wurden mir freundlicherweise von meinem Betreuer Volker Baier zur Verfügung gestellt.

[13] einige μs im Vergleich zu einigen ns

[14] vgl. dazu 3.4.4

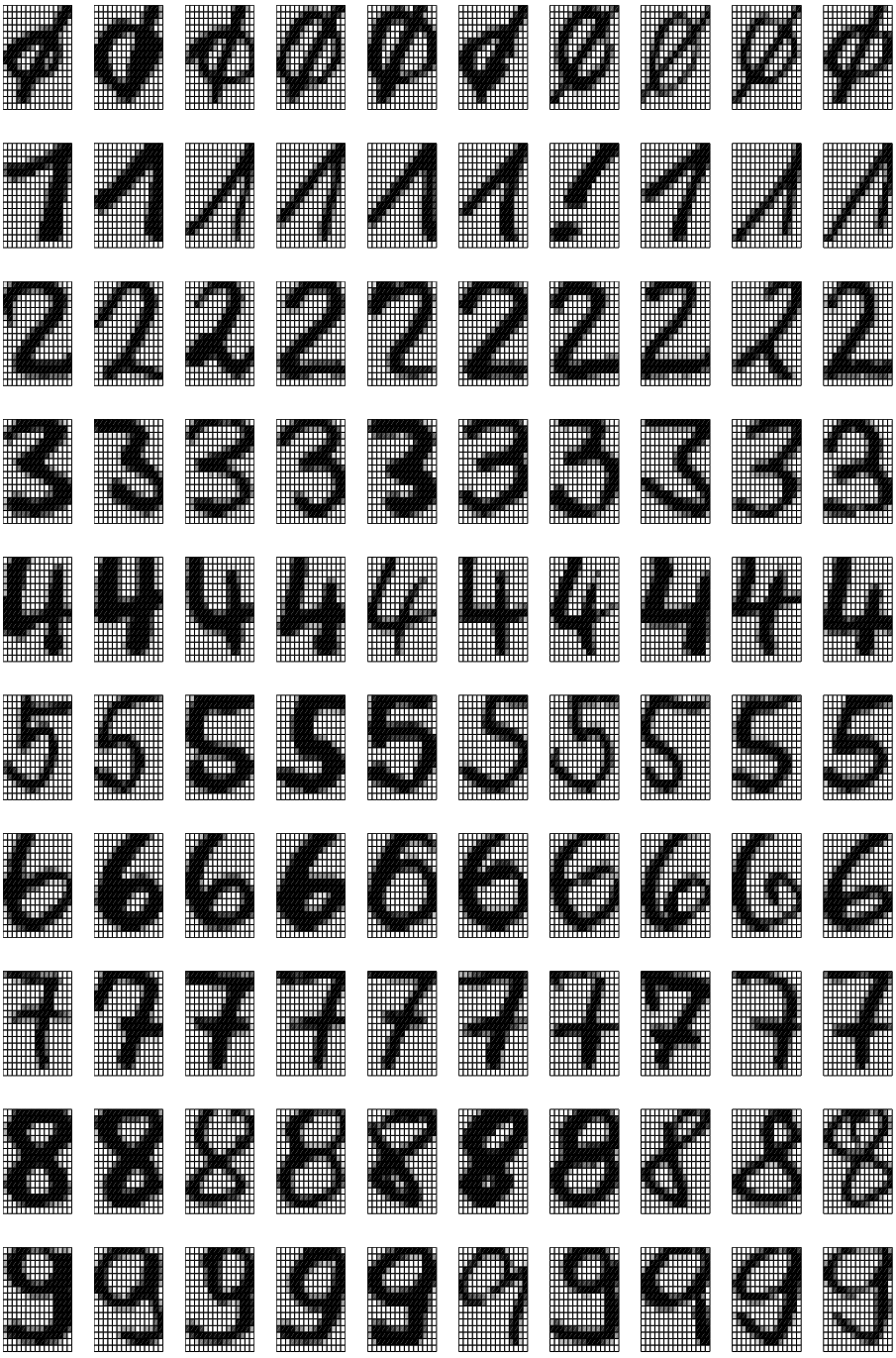


Abbildung 4.5: Beispiele aus dem Zifferndatensatz

4.4.2.2 Training

Einer SOM mit 10x10 Neuronen wurden nun zufällige Vektoren aus dem Trainingsset präsentiert und der Lernfortschritt beobachtet. Dazu wurde eine 10x10 Matrix erstellt, in die der jeweilige Gewinner für einen bestimmten Trainingsvektor eingetragen wurde. War also das Neuron in Spalte xp und Zeile yp Gewinner für die Ziffer 3, so wurde der entsprechende Matrixeintrag auf 3 gesetzt, usw. (s. Abb 4.6). Aus der Abbildung

2	2	3	3	3	3	9	9	7	7
2	2	3	3	3	9	9	7	7	7
2	2	3	3	2	9	7	9	7	4
2	2	8	8	9	4	9	9	4	4
2	2	9	8	8	9	4	4	4	4
2	8	8	8	8	9	9	9	4	5
1	1	8	8	8	8	6	6	5	5
8	1	0	0	8	6	6	6	6	5
1	1	1	0	0	0	6	6	5	5
1	1	0	0	0	0	6	5	5	5

Abbildung 4.6: Gewinnerneuronen für alle Ziffern

ist zu entnehmen, dass ähnlich aussehende Zahlen tatsächlich auch nebeneinander auf der Karte zu liegen kommen[15]. Diese Matrix wurde jede Sekunde vom Master-Node aktualisiert und der Trainingserfolg konnte so direkt beobachet werden.

Zusätzlich ließ sich der Klassifizierungsfehler bestimmen: Es muss dazu lediglich vor jedem Eintrag in die Matrix überprüft werden, welcher Wert vorher an derselben Stelle vorhanden war. Ist dieser identisch mit dem einzutragenden Wert, hat die selbstorganisierende Karte die Ziffer korrekt erkannt. Bei einem unterschiedlichen Wert wurde ein Klassifizierungsfehler begangen. Beobachtet man das Verhältnis zwischen korrekt und falsch klassifizierten Ziffern über einen längeren Zeitraum, kann man eine Aussage über den Lernfortschritt des Netzes treffen.

Das Ergebnis war ernüchternd: Eine 10x10 SOM konnte die Trainingsvektoren nach mehreren 10000 Lernschritten nicht zuverlässig unterscheiden[16]. Bedenkt man die Erkenntnisse aus dem Experiment mit den Tiermerkmalen, so müsste sowohl die Netzgröße als auch die Anzahl der Lernschritte wesentlich größer gewählt werden, um bei diesem weitaus komplexeren Datensatz eine ähnlich gute Erkennungsleistung zu erreichen. Dies war jedoch nicht praktikabel, da das Training dieses vergleichweise kleinen

[15] die Ähnlichkeit zwischen 0 und 1 rührt daher, dass alle Nullen mit einem diagonalen Strich (amerikanische Schreibweise) versehen wurden.

[16] Der Klassifizierungsfehler sank zwar langsam ab, lag aber bei Trainingsende immer noch bei über 30%

Netzes bereits eine immense Zeit benötigte[17]. Eine ausführliche Beurteilung zur Geschwindigkeit folgt in Kapitel 4.5.

Um die Komplexität zu reduzieren, wurden nur noch 100 Vektoren für das Training herangezogen. Es ist dann natürlich nicht möglich, das Netz auch mit unbekannten Vektoren zu testen, da eine Generalisierung bei so wenigen Trainingsvektoren nicht zu erwarten ist. Einem Netz mit 100 Neuronen sollte es allederdings zumindest möglich sein, diese Beispiele fehlerfrei "auswendig zu lernen". Doch auch dieses Ziel konnte nicht erreicht werden. Zwar konnte der Erkennungsfehler auf unter 10% gesenkt werden, von fehlerfreier Klassifikation konnte jedoch nicht die Rede sein.

4.4.2.3 Inkrementelles Training

Um dennoch ein besseres Ergebnis bei derselben Datenmenge zu erhalten, wurde das Lernverfahren modifiziert: Anstatt alle 100 Trainingsvektoren zufällig abwechselnd zu präsentieren, wurde dem Netz lediglich ein Vektor pro Ziffer (also insgesamt 10 Vektoren) präsentiert. Erwartungsgemäß war das Netz nun sehr wohl in der Lage, diese auswendig zu lernen und fehlerfrei zu erkennen. Nach einer bestimmten Anzahl von Lernschritten wurden jedoch dem Trainingsset 10 weitere Ziffern hinzugefügt, so dass dem Netz aus den 20 Vektoren nun entweder bekannte oder neue präsentiert wurden. Gleichzeitig wurden auch Lernrate und Nachbarschaftsradius wieder *heraufgesetzt*, damit sich das Netz an die neuen Daten anpassen konnte. Beide Parameter wurden aber geringer als zu Trainingsbeginn gewählt, so dass sich das Netz nicht mehr komplett umstrukturieren, sondern lediglich die bisherige Topologie optimieren konnte. Nach und nach wurden auch die restlichen Trainingsvektoren in das Training einbezogen und dabei Lernrate und Nachbarschaftsradius jedes Mal erneut hinaufgesetzt - allerdings um immer kleinere Werte.

Am Ende war das Netz zwar immer noch nicht fähig, alle 100 Beispiele fehlerfrei zu klassifizieren, allerdings fiel das Endergebnis offensichtlich ein wenig besser aus als im letzten Test. Inwieweit "*inkrementelles Training*" einen Vorteil bringt, muss jedoch in weiteren Experimenten untersucht werden. Durch geschickte Wahl von Lernrate und Nachbarschaftsradius - sinnvoll gekoppelt an den Erkennungsfehler - sollte es allerdings möglich sein, die Trainingszeit zu verkürzen: Wenn das Netz die bekannten Vektoren bereits gut erkennt, wird es auch einige unbekannte Beispiele korrekt klassifizieren. Nur wenn ein neuer Datensatz falsch erkannt wird, ist eine Anpassung erforderlich, die sich dann jedoch nur auf einen kleinen Teilbereich der SOM beziehen wird.

Eine weitere interessante Methode wäre, ein Netz bei Bedarf zu vergrößern[18], wenn nicht mehr alle Trainingsdaten fehlerfrei erkannt werden können. In Kombination mit inkrementellem Training und bedarfsweiser Anpassung der Lernparameter sollte sich

[17] ein Trainingslauf dauerte mehrere Stunden.
[18] dies hat B. Fritzke in [Fri96] vorgeschlagen.

so ein Netzwerk generieren lassen, das nicht nur eine minimale (und damit optimale) Größe hätte, sondern sich auch in möglichst geringer Zeit trainieren lässt.

4.4.3 Laser Time Series

In den bisherigen beiden Experimenten wurde ein Faktor $\alpha = 1$ verwendet, was also einer "normalen" SOM ohne temporalen Charakter entspricht. Eine wichtige Aufgabe aus dem Bereich der Temporal Sequence Processing ist die Prädiktion, also die Vorhersage zukünftiger Daten. Das Netz wird dabei nicht mit einzelnen Trainingsvektoren $\boldsymbol{\xi}(t)$, sondern einer ganzen Folge $\boldsymbol{\xi}(t), \boldsymbol{\xi}(t+1), ..., \boldsymbol{\xi}(t+n)$ trainiert. Der Faktor α wird dazu entsprechend eingestellt. Sollen beispielsweise die vergangenen 5 Vektoren berücksichtigt werden, wird α auf den Wert 0.4 gesetzt. Nach fünf Eingaben ist die Impulsantwort des IIR-Filters auf unter 5% abgesunken, so dass noch frühere Vektoren eine zu vernachlässigende Rolle spielen.

In [KVHK97] wurde eine rekurrente selbstorganisierende Karte verwendet, um die Leistung eines Lasers in chaotischem Zustand vorherzusagen. Der Trainingsdatensatz "Sante Fe Laser Time Series" wurde bereits für mehrere Experimente verwendet, um die Qualität verschiedener Prädiktionsverfahren zu vergleichen. Er besteht aus insgesamt 10000 (eindimensionalen) Datenwerten. In Abb. 4.7 sind 5000 Werte[19] grafisch dargestellt.

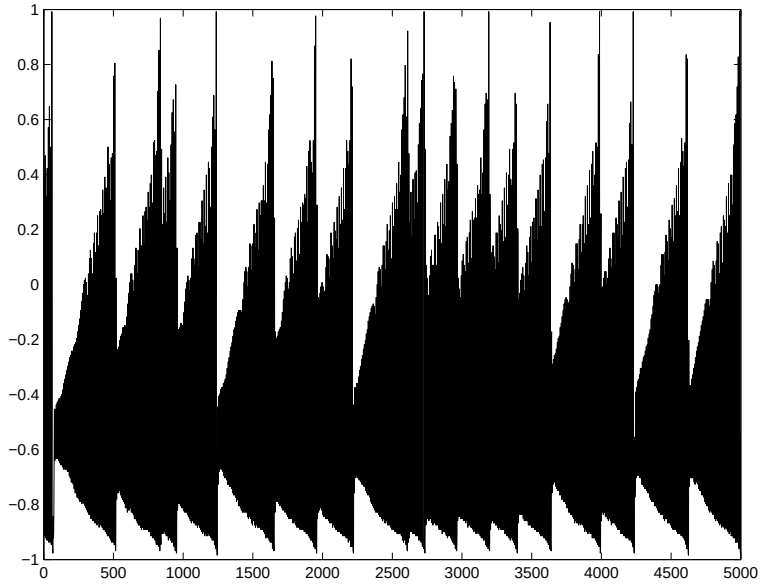


Abbildung 4.7: Sante Fe Laser Time Series

[19] normiert auf das Intervall von -1 bis 1

Soll nun eine RSOM mit diesem Datensatz trainiert werden, so werden dem Netz nacheinander mehrere aufeinanderfolgende Datenwerte $\boldsymbol{\xi}(t), \boldsymbol{\xi}(t+1), ..., \boldsymbol{\xi}(t+n)$ dargeboten. Es wird also mit einer *Datenfolge* trainiert, die jeweis zufällig aus dem Datensatz herausgegriffen wird. Die Gewichte werden auch nicht nach jedem Wert adaptiert, sondern erst nachdem alle n Werte einer Folge präsentiert wurden. Dadurch wird jedes Neuron Repräsentant für eine bestimmte Datenfolge und nicht mehr nur für einen einzelnen Datenvektor. Ist das Netz einmal trainiert, reicht es aus, nur den Anfang einer Folge zu präsentieren; Gewinner ist dann das Neuron mit der (aus Sicht des Netzes) wahrscheinlichsten Folge.

Das bedeutet allerdings, dass man für jede Folge von Datenwerten ein eigenes Neuron benötigt. Soll also der letzte Wert in einer Folge aus 3 Datenvektoren von -1 bis 1 mit einer Genauigkeit von 0.1 vorhergesagt werden, müsste das Netz bereits 8000 Neuronen[20] enthalten! Das ist für komplexere Aufgaben weder praktikabel, noch scheint es biologisch plausibel.

4.4.3.1 Training mit der Mexican-Hat-Funktion

Wahrscheinlicher ist es, dass für jeden Vektor der Folge ein *eigenes* Neuron aktiviert wird und für den nächsten ein anderes[21]. Werden mehrere Datensätze hintereinander präsentiert, so bilden die jeweiligen Gewinnerneuronen auf der RSOM-Oberfläche eine *Trajektorie*, die für jede Datenfolge eine eigene charakteristische Form hat (s. Abb. 4.8).

Damit eine Prädiktion möglich ist, muss die RSOM fähig sein, eine angefangene Trajektorie zu *vervollständigen*. Der beschriebene RSOM-Algorithmus kann dies jedoch nicht leisten. Daher entstand die Idee, das bisherige Modell zu erweitern, so dass auch eine Prädiktion von Trajektorien möglich werden sollte.

Der Grundgedanke bestand darin, die Nachbarschaftsfunktion nicht nur beim Gewichteupdate, sondern auch bei der Bestimmung des Gewinnerneurons zu verwenden: War ein Neuron Gewinner im ersten Schritt einer Folge, so sollte durch Verwendung einer geeigneten Nachbarschaftsfunktion gewährleistet werden, dass im nächsten Schritt ein Neuron *in der Nähe* des jetzigen Gewinners die geringste Aktivierung erhält. Anstatt der Gauß-Funktion soll eine *Mexican-Hat-Funktion* verwendet werden. Wie die Gauß-Funktion hat diese ihr Maximum im Ursprung, sinkt danach aber kurzzeitig in den negativen Bereich ab und konvergiert erst dann gegen null (s. Abb. 4.9):

$$f_{Mexi,(i,b)} = 1 - \frac{d_E(i,b)^2}{\sigma(t)^2} \cdot \exp^{(\frac{d_E(i,b)^2}{\sigma(t)^2})} \tag{4.3}$$

[20] für jeden der drei Werte gibt es 20 Möglichkeiten, also $20 \cdot 20 \cdot 20 = 8000$

[21] Es konnte gezeigt werden, dass ein (biologisches) Neuron direkt nach seiner Aktivierung nicht ein zweites Mal aktiv werden kann, auch wenn der Reiz noch so groß ist. Diese Zeit wird als ??? bezeichnet.

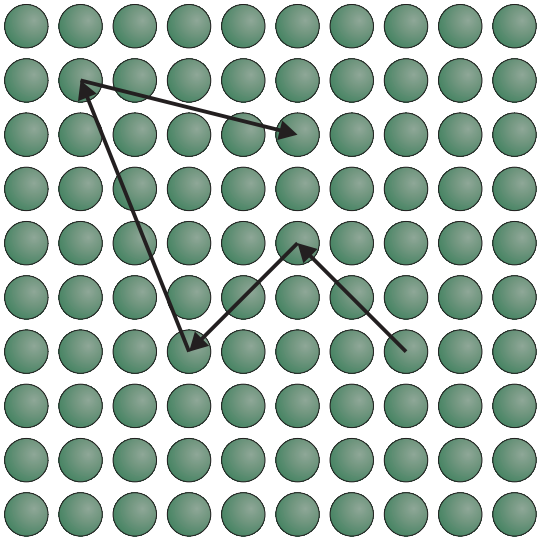


Abbildung 4.8: Trajektorie auf einer RSOM

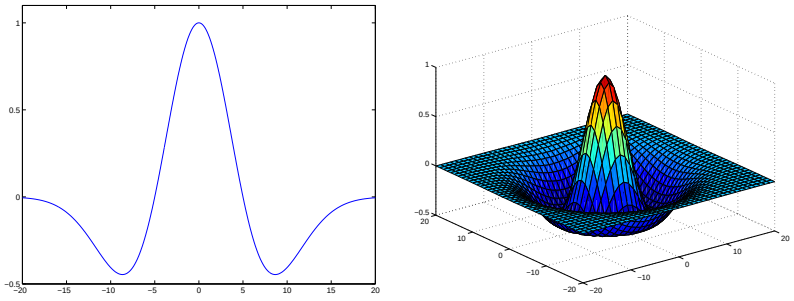


Abbildung 4.9: Mexican-Hat-Funktion

Wird die Umgebung des Gewinners mit der *negativen* Mexican-Hat-Funktion *überlagert*, ist die Chance geringer, dass ein und dasselbe Neuron in zwei Schritten hintereinander die geringste Aktivierung hat. Es wird vielmehr eines in der näheren Umgebung bevorzugt, und zwar an einer Stelle, wo die Mexican-Hat-Funktion unter null absinkt (s. Abb. 4.10).

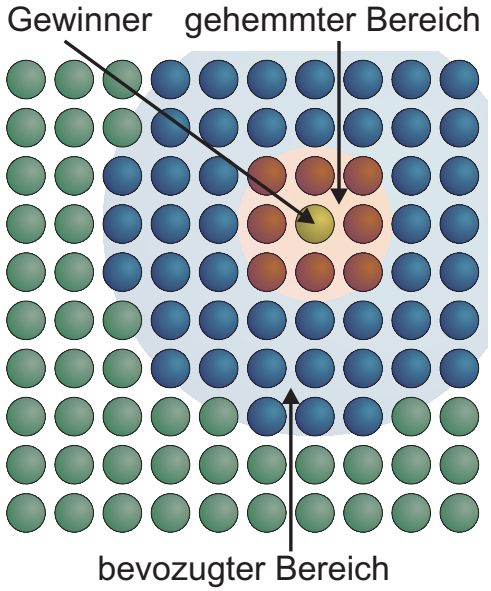


Abbildung 4.10: Biaswirkung durch die Mexican-Hat-Funktion

Das bisherige RSOM-Lernverfahren wird dadurch sozusagen um ein selektives Bias erweitert, das sich nach der Position des aktuellen Gewinners b richtet:

$$\boldsymbol{\psi}_{(i)}(t) = -f_{Mexi,(i,b)}(t) \cdot (1-\alpha)\boldsymbol{\psi}_{(i)}(t-1) + \alpha(\boldsymbol{\xi}(t) - \boldsymbol{\varrho}_{(i)}(t)) \tag{4.4}$$

Beim Gewichteupdate findet jedoch wie bisher die Gauß-Funktion ihre Anwendung:

$$\boldsymbol{\varrho}_{(i)}(t+1) = \boldsymbol{\varrho}_{(i)}(t) + \eta(t)h_{(i,b)}(t)\boldsymbol{\psi}_{(i)}(t) \tag{4.5}$$

Wird das Netz nun mit Vektorfolgen trainiert, sollte es nach einiger Zeit in der Lage sein, diese selbständig zu vervollständigen, wenn lediglich die ersten Daten der Folge und danach Vektoren der Form $\boldsymbol{\xi}_{(0)} = [0, 0, ..., 0]^T$ präsentiert werden. Es wäre dabei unerheblich, wie lang die Folgen gewählt werden. Auch sollte die Vorhersage nicht auf einen Schritt beschränkt sein, da das Netz auch ohne Eingabesignale permanent neue Trajektorien bilden würde.

4.4.3.2 Ergebnis

Leider stellte sich heraus, dass der vorgeschlagene Algorithmus in dieser Form nicht funktionsfähig war. Das Netz ließ sich wie erwartet trainieren; als beim Prädiktionstest jedoch keine weiteren Eingabevektoren mehr folgten, wurde ein Neuron Gewinner, das bisher keinen Datenvektor repräsentiert hatte[22]. Bei näherer Betrachtung ließ sich das Verhalten des Netzes erklären und ein Defizit im Trainingsalgorithmus erkennen:

Werden dem Netz keine Datenvektoren mehr, sondern stattdessen lediglich Nullvektoren präsentiert, wird das Neuron mit dem geringsten euklid'schen Abstand zu diesen ausgewählt. Ist der Nullvektor während des Trainings nicht vorgekommen, wird ein Neuron Gewinner, das bis jetzt noch keinen Trainingsvektor repräsentiert hatte. Dieses Verhalten rührt daher, dass die Mexican-Hat-Funktion lediglich bei der Berechnung der Aktivierung verwendet wurde und nicht bei der Anpassung der Gewichte selbst. Versuche, die Gauß-Funktion auch beim Gewichteupdate durch die Mexican-Hat-Funktion zu ersetzen, scheiterten jedoch daran, dass einige Gewichte dann extrem hohe Werte annahmen und so die komplette RSOM unbrauchbar wurde. In diesem Fall werden die Referenzvektoren nämlich nicht nur zu den Trainingsvektoren *hingezogen*, sondern entfernen sich auch von ihnen (an den Stellen, wo die Mexican-Hat-Funktion unter null liegt).

Das vorgeschlagene Lernverfahren stellt jedoch eine interessante Idee dar, die sich evtl. durch Einführung einer oberen Schranke für die Gewichte bzw. duch weiter modifizierte Lernregeln doch noch realisieren lässt.

4.4.4 RoboCup

Eine andere Aufgabe ist die Klassifikation von temporalen Daten. Dazu wurden mir vom *Lehrstuhl für Bildverstehen und wissensbasierte Systeme* Daten aus dem Roboterfußball (*RoboCup*) zur Verfügung gestellt[23]. Beim RoboCup versuchen zwei konkurrierende Teams von Robotern, einen Ball in das jeweils gegnerische Tor zu schießen - genau wie beim richtigen Fußball. Um seine eigenen Aktionen zu berechnen, muss jeder Roboter zunächst seine Position, die der Mitspieler und die des Balles aus Kameradaten bestimmen. Neben Kameras, die auf den Robotern selbst montiert waren, existiert außerdem eine Deckenkamera, die das Spielfeld von oben aufnimmt. Aus diesen Daten werden mehrmals pro Sekunde die Positionen aller relevanten Objekte bestimmt.

Für das Training der RSOM standen mehrere Mitschnitte realer RoboCup-Spiele zur Verfügung, die zeitlich veränderliche Postitionsdaten enthielten. Isolierte man daraus nur die Positionen des Balles, ergeben sich Trajektorien, die der Laufbahn des Balles

[22] dieses hatte den geringsten euklid'schen Abstand zum Nullvektor

[23] mein besonderer Dank gilt hierbei Thorsten Schmitt

über die gesamte Spielzeit entsprechen. Eine RSOM sollte fähig sein, diese in die Klassen "Ball in Ruhe", "Ball wurde soeben angestoßen" und "Ball in Bewegung" (mit entsprechender Richtungskomponente) einzuteilen.

Zunächst galt es, die relevanten Balldaten aus einer ganzen Reihe zusätzlichen Information (Position der Spieler, des Tors, der Spielfeldmitte, etc.) herauszufiltern. Über ein Perl-Skript ließ sich dies relativ schnell erreichen. Es standen nun zweidimensionale Positionsdaten mit Zeitindex zur Verfügung, also Trainingsvektoren $\boldsymbol{\xi}(t) \in \mathbb{R}^2$, die auf Werte zwischen -1 und 1 skaliert wurden.

4.4.4.1 Untersuchung der Datenqualität

Um zu untersuchen, wie gut diese Daten für das Training geeignet sind, wurde zunächst der Zeitindex untersucht. Optimal wäre es, wenn pro Sekunde etwa gleich viele Messwerte zur Verfügung stehen würden ohne dass größere Lücken (engl. *gaps*) in den Daten auftreten. Dies lässt sich am einfachsten aus einem Diagramm entnehmen: Werden die Datenwerte äquidistant auf der x-Ache und der zugehörige Zeitindex auf der y-Achse aufgetragen, erkennt man sofort Qualität der Daten (s. Abb. 4.11).

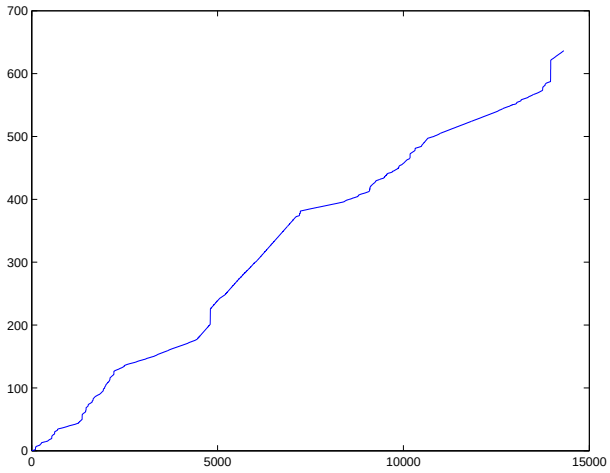


Abbildung 4.11: Werte und zugehöriger Zeitindex

Wünschenswert wäre es, eine Gerade zu erhalten; ihre Steigung ist identisch mit dem Zeitintervall zwischen zwei aufeinanderfolgenden Messwerten. Wie aus Diagramm 4.11 zu entnehmen ist, dauerte der dargestellte Spielabschnitt 636 Sekunden. In dieser Zeit wurden 14315 Postionswerte für den Ball erfasst. Durschnittlich standen also etwa 22.5 Messwerte pro Sekunde zur Verfügung. Allerdings variiert die Anzahl der Messwerte pro Zeiteinheit und es treten auch einige gaps auf (zu erkennen an Abschnitten mit sehr großer Steigung). Dennoch gibt es einige Phasen mit relativ konstanter Datenrate, die sich gut für ein Training eignen würden.

Als nächstes sollen auch die Positionsdaten selbst untersucht werden. In Abbildung 4.12 wurden alle Messwerte (Ballpositionen) in einem x-y-Koordinatensystem aufgetragen. Man kann sich also vorstellen, von oben auf das Spielfeld zu blicken. Auffällig ist, dass

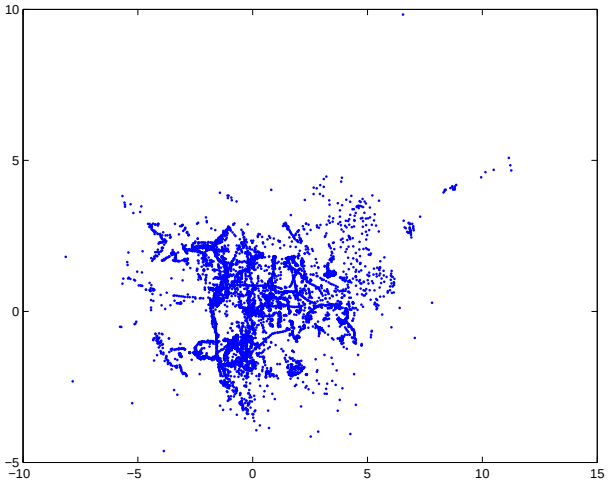


Abbildung 4.12: Ballpositionen

einige Messwerte sehr stark gestreut sind und auch die Ballbewegung lässt sich am gezeigten Diagramm nicht nachvollziehen. Bei näherer Untersuchung stellte sich heraus, dass es die Daten eine große Menge an Messfehlern enthielten - der Ball wurde also an einer ganz anderen Stelle vermutet, als er sich tatsächlich befand.

Angesichts dieser Messfehler schien es nicht sinnvoll, die RoboCup-Daten ohne weitere Vorverarbeitung zum Training der RSOM zu verwenden. Durch eine geeignete Filterung könnte man unrealistische Messwerte aus den Daten entfernen. Anschließend würde eine Mittelwertsbildung über einen bestimmten Zeitabschnitt auch die Anzahl der Messwerte pro Sekunde auf einen einheitlichen Wert bringen (beispielweise nur noch 5 Messwerte pro Sekunde). Aus Zeitgründen wurde darauf jedoch im Rahmen dieser Arbeit verzichtet.

4.5 Performance

Interessant ist auch zu analysieren, welcher Geschwindigkeitszuwachs durch die Verwendung mehrerer Rechner erzielt werden kann. Wie aus der Tabelle 4.1 zu entnehmen ist, standen am Lehrstuhl für Theoretische Informatik und Künstliche Intelligenz sehr unterschiedliche Rechner zur Verfügung. Da alle Rechner ans Netzwerk der Universität angeschlossen waren, konnte die Leistungsfähigkeit auch durch Internetzugriffe oder Datenübertragungen der Mitarbeiter beeinträchtigt werden, was eine objektive Bewertung schwierig macht. Auch die unterschiedlich schnelle Netzwerkanbindung mit 10

MBit für die SPARCstation 20 und 100 MBit für die übrigen Rechner stellt ein Problem dar.

4.5.1 Bandbreite

Zunächst wurde die vorhandene Bandbreite zwischen den Rechnern untersucht, da die Kommunikation der Nodes untereinander für die RSOM eine wesentliche Rolle spielt. Testweise wurden daher mehrere Datenpakete unterschiedlicher Größe zwischen zwei Nodes über MPICH ausgetauscht und die Übertragungsbandbreite gemessen. Aus Tabelle 4.3 ist zu entnehmen, dass die Kommunikation der SPARCstation 20 erwartungsgemäß etwas langsamer ist als bei den Rechnern mit schnellerer Netzwerkanbindung. Da alle SPARCstation 20 direkt an einen gemeinsamen Switch angebunden sind, die übrigen Rechner jedoch oft in unterschiedlichen Räumen stehen, fällt der Unterschied insbesondere bei größeren Paketen nicht so groß aus. Generell ist es ganz offensichtlich günstiger, möglichst wenige große Datenpakete zu verschicken als die Daten auf viele kleinere zu verteilen.

Paketlänge [Bytes]	Netwerkbandbreite [MBytes/s]	
	Sun SPARCstation 20	Sun Ultra 1
1	0.000710	0.001514
2	0.001638	0.003309
4	0.003388	0.007096
8	0.006248	0.011864
16	0.013462	0.027555
32	0.025029	0.048905
64	0.050374	0.092352
128	0.090941	0.153171
256	0.154870	0.259547
512	0.246806	0.370388
1024	0.342876	0.579732
2048	0.457041	0.746537
4096	0.674239	0.939162
8192	0.806379	1.032822
16384	0.958269	1.080549
32768	0.944663	1.098614
65536	0.969510	1.100342
131072	0.960643	1.095245
262144	0.982236	1.086846
524288	0.996725	1.102931

Tabelle 4.3: Netzwerkbandbreite

4.5.2 Rechenleistung

Wesentlich aussagekräftiger als Übertragungszeiten für bestimmte Datenpakete sind jedoch Programmlaufzeiten für gegebene Problemstellungen. Während des Trainings werden bekanntlich für jeden Node die Zeiten gemessen, die der Rechner für die Bearbeitung bestimmter Aufgaben benötigt. Es stehen die Messwerte t_l, t_g und t_u für die Suche nach dem lokalen bzw. globalen Gewinner und das Ändern der Gewichte zur Verfügung. Außerdem wird vom Master die Gesamtlaufzeit des Programmes bestimmt. Addiert man die Messwerte t_l und t_u aller Nodes und stellt sie der Gesamtlaufzeit gegenüber, so erhält man ein Maß für die Rechenleistung des gesamten Clusters.

Der Zeit für die Kommunikation der Rechner untereinander lässt sich ebenfalls bestimmen. Für die Suche des globalen Gewinners wird bekanntlich die MPI-Funktion `MPI_AllReduce()` verwendet. Dabei vergleichen jeweils zwei Nodes ihre Ergebnisse miteinander, jedoch müssen zur Bestimmung des Endergebnisses nicht alle Nodes paarweise miteinander kommunizieren. Die Berechnung findet vielmehr unter Verwendung eines baumartigen Verfahrens statt, so dass das Endergebnis Stück für Stück zusammengesetzt wird. Die Übertragungszeiten können dabei für jeden Node stark variieren. Unter Umständen liegt das Endergebnis bereits in einem Node vor (die Zeitmessung dort wird gestoppt), muss aber noch an andere Nodes übertragen werden (diese warten immer noch auf das Endergebnis). Jeder Node misst also die Zeit, bis *ihm selbst* das Endergebnis zur Verfügung steht. Ist er schneller als die anderen, beginnt er zwar sofort mit dem Gewichteupdate, muss aber spätestens beim nächsten Trainingsvektor auf die anderen Rechner warten. Es bestimmt also der Rechner mit der größten Zeit t_g die Geschwindigkeit des gesamten Clusters. Für die Kommunikationszeit wird daher die Summe der maximalen Werte $\sum \max(t_g)$ zu Grunde gelegt.

4.5.2.1 Sun SPARCstation 20

Zunächst wurde das aus Kapitel 4.4.1 bekannte Beispiel mit den Tiermerkmalen auf mehreren Sun SPARCstation 20 Model 512 durchgeführt[24]. Dem Netz wurden dazu jeweils 10000 zufällig ausgewählte Trainingsvektoren präsentiert und die für das Training benötigten Zeiten in Sekunden gemessen[25]. Netzwerkgröße und Rechnerzahl wurden dabei variiert. Die Ergebnisse von zwei Messungen sind in Tabelle 4.4 dargestellt.

Enthalten sind jeweils die Gesamtlaufzeiten t (gemessen auf dem Master) und die Summen der Messungen aller Nodes für die Suche nach lokalem bzw. globalem Gewinner (t_l und t_g) und für das Gewichteupdate (t_u).

[24] Damit das Programm lauffähig ist, wird mindestens ein Slave benötigt. Die minimale Nodeanzahl beträgt also 2.

[25] Der Faktor α wurde auf einen Wert von 1.0 gesetzt, was einer nicht-rekurrenten SOM entspricht. Die Wahl des Parameters hat jedoch keinen Einfluss auf die Messwerte, da die Rechenoperationen für unterschiedliche α zeitlich äquivalent sind.

Rechnerzahl	Neuronen	20x20		30x30		40x40	
	Messung	1	2	1	2	1	2
2	t	287	289	624	626	1134	1136
	$\sum t_l$	122	122	272	273	484	483
	$\sum \max(t_g)$	30	33	44	43	53	48
	$\sum t_u$	350	350	794	795	1426	1423
3	t	217	217	442	439	788	785
	$\sum t_l$	123	123	273	273	484	484
	$\sum \max(t_g)$	40	38	40	37	49	49
	$\sum t_u$	351	350	795	796	1431	1429
4	t	201	197	370	367	630	627
	$\sum t_l$	124	124	276	275	488	488
	$\sum \max(t_g)$	55	54	63	60	68	64
	$\sum t_u$	351	351	796	794	1426	1423

Tabelle 4.4: Messwerte für Sun SPARCstation 20

Wie deutlich zu erkennen ist, können die einzelnen Messungen um mehrere Sekunden von einander abweichen. Es sei daher angemerkt, dass die Messwerte lediglich eine Näherung darstellen und prinzipbedingt nicht exakt sein können:

- Es treten Schwankungen in der zur Verfügung stehenden Netzwerkbandbreite auf (Internetzugriffe und Datenübertragungen durch andere Rechner).
- Es wurde kein Echtzeitbetriebssystem verwendet und es bleibt daher dem Kernel überlassen, wie viel Rechenleistung er einem Prozess zur Verfügung stellt.
- Jeder Node führt die Zeitmessung selbständig durch, was ebenfalls eine gewisse - wenn auch geringe - Zeit in Anspruch nimmt.
- Werden Neuronen von einem Rechner auf einen anderen Node übertragen, so geht auch dies in die Gesamtzeit t ein. Die Zeiten t_l, t_g und t_u bleiben jedoch unbeeinflusst.

Hätte man eine genaue Performancemessung durchführen wollen, wäre ein spezieller Messaufbau erforderlich gewesen. Die in Tabelle 4.4 gemessenen Zeiten sind jedoch Messwerte für ein konkrete Problemstellung auf einem realen Rechnercluster, dessen Leistung natürlichen Schwankungen unterworfen ist, und sind daher aussagekräftiger als eine exakte Messung unter idealisierten Bedingunen.

Der Tabelle ist außerdem zu entnehmen, dass die benötigte Gesamtzeit bei größeren Netzwerken ansteigt, sich aber durch Verwendung mehrerer Nodes reduzieren lässt. Weiterhin erkennt man, dass die "Gesamtrechenzeit" (gegeben durch $\sum t_l$ und $\sum t_u$) bei gleicher Netzwerkgröße konstant ist, auch wenn mehrere Nodes verwendet werden. Dies liegt daran, dass im durchgeführten Experiment alle Rechner etwa die gleiche

Geschwindigkeit haben. Es ist also unerheblich auf wie viele Nodes ein Problem verteilt wird, seine Komplexität lässt sich natürlich nicht reduzieren. Da die Zeiten $\sum t_l$ und $\sum t_u$ bei gleicher Netzwerkgröße auch nicht ansteigen, erkennt man, dass sich eine (rekurrente) selbstorganisierende Karte offensichtlich sehr gut parallelisieren lässt. Der Kommunikationsaufwand steigt allerdings sowohl bei größeren Karten an, als auch wenn bei gleicher Kartengröße mehrere Nodes verwendet werden. Es fällt außerdem auf, dass die Verkürzung der Gesamtzeit bei einem Sprung von 2 auf 3 Nodes wesentlich ist, bei einem weiteren Node mehr jedoch viel kleiner ausfällt. Da der Kommunikationsaufwand mit wachsender Node-Zahl immer weiter ansteigt, der Gewinn an Gesamtzeit jedoch kleiner wird, ist zu vermuten, dass es je nach Kartengröße eine optimale Zahl an Nodes gibt, mit denen sich das Problem am schnellsten lösen lässt. Wird dann ein weiterer Node hinzugefügt, steigt die Verarbeitungszeit wieder an.

Leider standen am Lehrstuhl maximal vier identische Rechner zur Verfügung, weshalb dieser Effekt nur in einem Cluster aus unterschiedlich leistungsstarken Rechnern beobachtet werden konnte.

4.5.2.2 Sun Ultra 1

Zunächst soll allerdings derselbe Versuch mit mehreren Sun Ultra 1 durchgeführt werden. Aus 4.5 erkennt man sofort, dass diese Rechner mehr als doppelt so schnell sind

Rechnerzahl	Neuronen	20x20		30x30		40x40	
	Messung	1	2	1	2	1	2
2	t	124	124	264	267	473	468
	$\sum t_l$	53	53	122	121	215	214
	$\sum \max(t_g)$	15	15	17	24	17	17
	$\sum t_u$	148	147	338	339	609	608
3	t	92	93	187	187	329	328
	$\sum t_l$	55	55	121	121	216	215
	$\sum \max(t_g)$	16	16	21	18	21	18
	$\sum t_u$	149	149	338	339	610	608
4	t	87	87	157	157	268	266
	$\sum t_l$	56	56	123	123	218	217
	$\sum \max(t_g)$	28	28	28	28	33	31
	$\sum t_u$	149	149	338	338	613	612

Tabelle 4.5: Messwerte für Sun Ultra 1

und auch die Kommunikation über das Netzwerk nur noch einen Bruchteil der Zeit wie bei den SPARCstation 20 in Anspruch nimmt. Qualitativ ergeben sich jedoch dieselben Ergebnisse:

- Die Zeiten können um mehrere Sekunden variieren.
- Die erforderliche Gesamtrechenzeit bleibt bei gleicher Netzwerkgröße konstant.
- Der Kommunikationsaufwand steigt sowohl mit steigender Rechnerzahl als auch bei größeren Karten.
- Der Geschwindigkeitszuwachs durch einen zusätzlichen Node fällt bei wenigen Rechnern größer aus.

4.5.2.3 Unterschiedliche Rechner

Als nächstes soll lediglich die benötigte Gesamtzeit der Dauer für den Datenaustausch gegenübergestellt werden. Um die optimale Anzahl an Nodes auszuloten, müssen nun auch Rechner unterschiedlicher Leistungsklassen kombiniert werden. Allerdings wird durch die Software gewährleistet, dass die Neuronen automatisch so verteilt werden, dass alle Rechner gleich stark ausgelastet sind und das Cluster so mit der maximal möglichen Geschwindigkeit arbeitet.

Neuronen	10x10		
Rechnerzahl	Rechner	$\sum \max(t_g)$	t
1	Ultra 1	-	-
2	Sparc 20/512	26	67
3	Ultra 1	28	55
4	Sparc 20/512	37	62
5	Ultra 1	38	59
6	Sparc 20/512	39	59
7	Ultra 1	41	59
8	Sparc 20/512	49	67
9	Sparc 20/702	53	71
10	Sparc 20/702	51	69
11	Ultra 2	52	70
12	Ultra 10	52	68

Tabelle 4.6: Messwerte für ein gemischtes Cluster (10x10 SOM)

In Tabelle 4.6 sind die Messwerte für eine 10x10 SOM bei 10000 Lernschritten dargestellt. Die aus den beiden vorangegangenen Experimenten bekannten Sun SPARCstation 20 Model 512 und Sun Ultra 1 werden zunächst abwechselnd dem Cluster hinzugefügt. Dabei ist bereits zu erkennen, dass das Hinzufügen eines Nodes nicht zwangläufig zu einem Geschwindigkeitszuwachs führen muss. Vielmehr ist die optimale Clustergröße bereits bei 3 Nodes erreicht. Auffällig ist, dass die Bestimmung des globalen Gewinners für bestimmte Anzahlen an Nodes etwa die gleiche Zeit benötigt[26]. Dies ist auf

[26] ca. 27 Sekunden für 2 und 3, ca. 39 Sekunden für 4,5,6,7 und ca. 51 Sekunden für 8,9,10,11,12

das bereits erwähnte Baumverfahren in MPI zurückzuführen, das bei der Bestimmung des globalen Gewinners Anwendung findet. Wie allerdings aus dem Experiment deutlich wird, kann die Kommunikation unter Umständen für einen Großteil der gesamten Programmlaufzeit verantwortlich sein. Bei 12 Nodes beträgt die eigentliche Rechenzeit nur noch ca. 15 Sekunden[27]. Dies ist zwar wesentlich kürzer als bei zwei Nodes, jedoch bleibt die benötigte Gesamtzeit nahezu gleich. Selbstverständlich ist man bestrebt mit einem möglichst kleinen Cluster gute Ergebnisse zu erzielen, da der nötige Administrationsaufwand, die Wartung und natürlich die Hardware selber einen Kostenfaktor darstellen.

Müssen jedoch größere Aufgabenstellungen gelöst werden (erhöht sich also der Bedarf an Rechenleistung, wobei der Kommunikationsaufwand nur geringfügig ansteigt), kann durch die Verwendung von zusätzlichen Nodes ein deutlicher Geschwindigkeitsgewinn erzielt werden.

Neuronen	30x30		
Rechnerzahl	Rechner	$\sum \max(t_g)$	t
1	Ultra 1	-	-
2	Sparc 20/512	41	395
3	Ultra 1	38	255
4	Sparc 20/512	44	231
5	Ultra 1	43	189
6	Sparc 20/512	45	177
7	Ultra 1	45	155
8	Sparc 20/512	54	159
9	Sparc 20/702	54	151
10	Sparc 20/702	57	146
11	Ultra 2	58	139
12	Ultra 10	58	133

Tabelle 4.7: Messwerte für ein gemischtes Cluster (30x30 SOM)

Tabelle 4.7 zeigt die Zeitdauer für 10000 Lernschritte bei einer 30x30 SOM, was etwa dem neunfachen Rechenaufwand entspricht. Es ist deutlich zu erkennen, dass im Vergleich zum oben beschriebenen Experiment der Kommunikationsaufwand bei zwei Nodes fast auf das Doppelte angestiegen ist, die benötigte Gesamtdauer jedoch beinahe um den Faktor 6 größer ist. Es wird also nur noch etwa ein Zehntel der Zeit für den Datenaustausch verwendet, den Rest verbringen die Nodes allein mit Rechnen. Wie aus der Tabelle zu erkennen ist, lässt sich die Gesamtzeit auf ein Drittel reduzieren, wenn das Cluster von anfänglich zwei Nodes auf 12 aufgestockt wird.

[27] wie bereits oben erwähnt, gehen einige nicht messbare Faktoren in die Gesamtzeit ein. Der Wert $68s - 52s \approx 15s$ ist also nur als Schätzwert zu betrachten.

4.5.2.4 Optimale Kombination

Bleibt die Frage, ob ein ähnliches (oder besseres Ergebnis) nicht auch mit weniger Nodes erzielt werden könnte. Bisher wurden ja langsame und schnelle Nodes abwechselnd dem Cluster hinzugefügt. Für die Messungen in Tabelle 4.8 wurden zunächst die schnellen Rechner verwendet und dann nach und nach langsamere Nodes hinzugefügt.

Neuronen	30x30		
Rechnerzahl	Rechner	$\sum \max(t_g)$	t
1	Ultra 10	-	-
2	Ultra 2	13	137
3	Ultra 1	20	124
4	Ultra 1	33	122
5	Ultra 1	33	111
6	Ultra 1	36	106
7	Sparc 20/702	42	112
8	Sparc 20/702	71	143
9	Sparc 20/512	72	141
10	Sparc 20/512	73	140
11	Sparc 20/512	72	138
12	Sparc 20/512	73	136

Tabelle 4.8: Messwerte für ein gemischtes Cluster bei optimaler Reihenfolge

Bereits mit zwei schnellen Rechnern wird annähernd dieselbe Zeit erreicht, die im letzten Experiment lediglich mit 12 Nodes möglich war. Fügt man dem Cluster weitere (immer noch vergleichsweise schnelle) Nodes hinzu, lässt sich diese Zeitdauer weiter optimieren. Das Minimum wird in diesem Test bei 6 Nodes erreicht, und zwar genau bevor eine (wesentlich langsamere) Sun SPARCstation 20 hinzugefügt wurde. Eine weitere Sun Ultra 1 hätte womöglich noch eine Verbesserung bewirkt. Ein großer Sprung ergibt sich von 7 auf 8 Nodes, da hier der Kommunikationsaufwand beträchtlich ansteigt. Ganz offensichtlich kommt an dieser Stelle ein weiterer "Ast" bei dem von MPI verwendeten Baumverfahren hinzu. Es ist anzunehmen, dass der Kommunikationsaufwand daher bis 15 Nodes etwa gleich bleibt und dann bei 16 Rechnern erneut einen Sprung macht[28]. Dies konnte jedoch leider nicht überprüft werden, da zusätzliche Rechner nur selten zur Verfügung standen.

Es lassen sich also einige einfache Regeln für die Verwendung einer SOM in einem Rechnercluster angeben:

- Es gibt eine optimale Anzahl von Nodes. Werden mehr Nodes verwendet, steigt die Gesamtdauer an.

[28] Leider war nicht bekannt, welches Verfahren dabei genau verwendet wird.

- Je größer die Karte, desto mehr Gewinn bringt ein größeres Cluster.
- Stehen in einem Cluster unterschiedlich schnelle Rechner zur Verfügung, sollten zuerst die schnellsten verwendet werden.
- Wird zu einem Cluster aus schnellen Rechnern ein langsamerer hinzugefügt, so ist nur in seltenen Fällen zu erwarten, dass sich die Verarbeitungszeit reduzieren lässt.
- Leider lässt sich keine Vorhersage treffen, welche Rechnerzahl für ein bestimmtes Problem optimal ist. Um die Anzahl der benötigten Testläufe zu reduzieren, sollte man lediglich Versuche mit $3, 7, 15, ..., 2^n - 1$ Nodes machen, um den Baumalgorithmus von MPI optimal auszunutzen.
- Will man ein bestehendes Cluster aufrüsten, muss es nicht zwangsläufig vergrößert werden, sondern es reicht aus, den langsamsten Rechner durch einen schnelleren zu ersetzen.

Kapitel 5

Schlussfolgerungen und Ausblick

In der vorliegenden Arbeit wurde eine parallele Simulationsumgebung für rekurrente selbstorganisierende Karten implementiert. Anhand von Beispieldaten konnte die Funktionsfähigkeit der Software und die Eignung der RSOM für bestimmte Problemstellungen evaluiert werden.

Es wurde festgestellt, dass eine SOM einfache Klassifizierungsaufgaben zufriedenstellend lösen kann. Durch die von der Karte durchgeführte topologieerhaltende Transformation lassen sich Zusammenhänge in den Daten leichter erkennen. Die SOM eignet sich daher ausgezeichnet für Visualisierungsaufgaben.

Sollen jedoch größere Datenmengen klassifiziert werden, erweist sich die Topologieerhaltung als Nachteil: Dadurch dass auch vollkommen unterschiedliche Daten auf der Karte in eine Ähnlichkeitsbeziehung gesetzt werden müssen, können einzelne Klassen nicht mehr zuverlässig unterschieden werden. Dies ist aus dem Experiment mit dem Zifferndatensatz deutlich geworden. Allerdings lässt sich die Leistung der SOM offensichtlich verbessern, wenn statt einer monoton sinkenden Lernrate bzw. Nachbarschaftsfunktion "intelligente" Parameter verwendet werden, die bei Bedarf verändert werden können. Auch scheint es besser zu sein, das Training mit einem kleinen Teil der Datenmenge zu beginnen und erst nach und nach zu erweitern.

Die Klassifikation temporaler Daten konnte im Rahmen dieser Arbeit leider nicht untersucht werden, da die zur Verfügung stehenden Positionsdaten aus dem Roboter-Fußball für ein Training ungeeignet waren. Eventuell könnte dieser Datensatz nach einer aufwendigen Vorverarbeitung doch noch verwendet werden. Womöglich ist es jedoch sinnvoller, auf andere Trainingsdaten zurückzugreifen.

Ein vielversprechender Ansatz scheint das "selektive Bias" zu sein, das im Experiment mit der Santa Fe Laser Time Series vorgestellt wurde. Allerdings konnte der Lernalgorithmus in der derzeitigen Form noch nicht erfolgreich eingesetzt werden. Hier bleibt zu untersuchen, welche weiteren Modifikationen am Lernverfahren notwendig sind,

um auch Trajektorien auf der RSOM-Oberfläche vorhersagen zu können.

Die erstellte Software selbst konnte erfolgreich für alle genannten Simulationen verwendet werden. Mit nur wenigen Programmzeilen ließ sie sich an die jeweilige Aufgabenstellung anpassen. Insbesondere die Funktion zum Einlesen der Trainigsvektoren musste dabei den Daten angepasst werden.

Das Kompilieren und Starten des Programmes in einem Rechnercluster bereitete keinerlei Probleme. Für die Simulation wurde die zur Verfügung stehende Rechenleistung stets optimal ausgenutzt, da die einzelnen Neuronen den Ressourcen entsprechend dynamisch verteilt wurden.

In zukünftigen Projekten könnte die Software für weitere Simulationen verwendet werden, um beispielsweise das Verhalten kaskadierter selbstorganisierender Karten zu untersuchen. Da eine objektorientierte Programmiersprache verwendet wurde, sollte sich auch dies mit relativ geringem Aufwand realisieren lassen.

Der Geschwindigkeitsgewinn durch parallele Verarbeitung konnte in 4.5.2 nachgewiesen werden. Wünschenswert wäre es, zukünftige Simulationen auch in leistungsfähigeren Rechnerclustern wie am Leibniz-Rechenzentrum durchzuführen. Da sich insbesondere die Netzwerkanbindung als bremsender Faktor herausgestellt hat, sollte sich die Verarbeitungsgeschwindigkeit bei Verwendung geeigneter Hardware noch drastisch steigern lassen.

Anhang A

Notation

Es wurde versucht, in dieser Arbeit auf eine konsistente Notation zurückzugreifen. Mengen werden dabei stets mit kalligrafischen Buchstaben bezeichnet. Vektoren werden durch fettgedruckte griechische Buchstaben symbolisiert.

$\mathcal{C} = \{\boldsymbol{\varrho}_1, \boldsymbol{\varrho}_2, ..., \boldsymbol{\varrho}_k\}$	Codebuch mit Referenzvektoren
$\mathcal{K} = \{K_1, K_2, ..., K_k\}$	Menge der zu unterscheidenden Klassen
$\mathcal{T} = \{\boldsymbol{\xi}_1, \boldsymbol{\xi}_2, ..., \boldsymbol{\xi}_k\}$	Trainingsmenge
d_E	euklidscher Abstand zweier Vektoren
E_q	Quantisierungsfehler
f_S	Schwellwertfunktion
f_{sig}	Sigmoidfunktion
$h_{(i,b)}$	Nachbatschaftsfunktion
$I(k)$	Impulsantwort
K	Klasse
$p(\boldsymbol{\xi})$	Wahrscheinlichkeitsdichteverteilung der Daten
S	Speicherbedarf
w_{ij}	Gewicht zwischen den beiden Neuronen i und j
α	"Erinnerungsfaktor"
$\eta(t)$	Lernrate
$\sigma(t)$	Radius der Nachbarschaftsfunktion
$\boldsymbol{\varepsilon}(t)$	mehrdimensionales Störsignal (Rauschen)
$\boldsymbol{\varrho}$	Codebuchvektor
$\boldsymbol{\xi}$	Merkmalsvektor oder Trainingsvektor
$\boldsymbol{\zeta}$	Zielvektor
$\boldsymbol{\psi}$	Ausgabevektor

Anhang B

Abkürzungen

BP	Backpropagation
CESDIS	Center of Excellence in Space Data and Information Sciences
CPU	Central Processing Unit
IIR	Infinite Impulse Response
KNN	Künstliche Neuronale Netzwerke
LAM	Local Area Multicomputer
LVQ	Learning Vector Quantization
MLP	Multilayer-Perceptron
MPI	Message Passing Interface
MPICH	MPI Chameleon
MPMD	Multiple Program Multiple Data
PC	Personal Computer
PCA	Principal Component Analysis
PVM	Parallel Virtual Machine
RBF	Radial Basis Functions
RSOM	Recurrent Self-Organizing Map
SMP	Symmetric Multiprocessor
SOM	Self-Organizing Map
SPMD	Single Program Multiple Data
SSH	Secure Shell
TIKI	Theoretische Informatik und künstliche Intelligenz
TSP	Temporal Sequence Processing
VQ	Vektorquantisierung
WTA	Winner take all

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[Bai98] Volker Baier. *Klassifikation mit Support Vektor Maschinen*. Universität Ulm, 1998.

[BSMM99] I.N. Bronstein, K.A. Semendjajew, G. Musiol, and H. Mühlig. *Taschenbuch der Mathematik*. 4., überarbeitete und erweiterte Auflage. Verlag Harri Deutsch, 1999.

[dABA00] Guilherme de A. Barreto and Aluizio F. R. Araújo. Time in self-organizing maps: An overview of models, 2000.

[Fri96] Bernd Fritzke. Growing self-organizing networks - why?, 1996.

[Fri98] Bernd Fritzke. *Vektorbasierte Neuronale Netze*. Shaker Verlag, 1998.

[Hay94] Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall International, Inc., 1994.

[Kan94] Jari Kangas. *On the Analysis of Pattern Sequences by Self-Organizing Maps*. PhD thesis, Helsinki University of Technology, 1994.

[Koh90] Teuvo Kohonen. The self-organizing map. *Proceedings of the IEEE*, 78:1464–1480, 1990.

[Koh01] Teuvo Kohonen. *Self-Organizing Maps*. Springer Verlag, third edition, 2001.

[KVHK97] T. Koskela, M. Varsta, J. Heikkonen, and K. Kaski. Time series prediction using recurrent som with local linear models. Technical Report B15, 1997.

[PTVF92] William H. Press, Saul A. Teukolski, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, second edition, 1992.

[RMS91] Helge Ritter, Thomas Martinetz, and Klaus Schulten. *Neuronale Netze: Eine Einführung in die Neuroinformatik selbstorganisierender Netzwerke*. 2. erweiterte Ausgabe. Addison-Wesley Verlag, 1991.

[Str93] Bjarne Stroustrup. *Die C++ Programmiersprache*. 2. erweiterte Ausgabe. Addison-Wesley Verlag, 1993.

[VHdRM97] Markus Varsta, Jukka Heikkonen, and José del R. Millan. Context learning with the self-organizing map. In *Proceedings of WSOM'97, Workshop on Self-Organizing Maps, Espoo, Finland, June 4–6*, pages 197–202. Helsinki University of Technology, Neural Networks Research Centre, Espoo, Finland, 1997.

[WL] G. William and E. Lusk. User's guide for mpich, a portable implementation of mpi.

Anhang C

Index