Klassifizieren und Visualisieren von Daten mit Selbstorganisierenden Karten

Sven Schröder

Diplomarbeit

zur Erlangung des akademischen Grades
Diplominformatiker (FH)

eingereicht an der
Fachhochschule Brandenburg
Fachbereich Informatik und Medien

---

1. Prüfer: Dipl. Inform. I. Boersch
2. Prüfer: Prof. Dr. Ing. J. Heinsohn

Brandenburg, den 2. März 2006

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Aufgabenstellung

Ziel der Arbeit ist die Untersuchung von Visualisierungsmöglichkeiten von Daten mit Hilfe von Selbstorganisierenden Karten. Hierbei sind sowohl musterbezogene als auch musterunabhängige Varianten zu betrachten. Die Verfahren sollen prototypisch in ein Programmsystem umgesetzt werden, welches ohne Installation über das Netz nutzbar ist. Der Lernprozess ist geeignet darzustellen.
Hierzu sind Selbstorganisierende Karten im theoretischen Überblick zu beleuchten und für den Schwerpunkt Visualisierung trainierter Karten zu vertiefen.
Anhand der im Verlauf der Diplomarbeit erzielten praktischen und theoretischen Erkenntnisse soll eine Bewertung und Systematisierung der Visualisierungsvarianten erstellt werden. Dabei sollen Aspekte wie Lesbarkeit, Informationsgehalt, Umsetzungskomplexität und andere Kriterien beurteilt werden.

1.2 Motivation

Die heutige Gesellschaft wird auch als Informations- bzw. als Wissensgesellschaft bezeichnet. Es werden täglich mehr und mehr Datenbestände angehäuft. In [JH99] wird behauptet, dass Daten als Rohstoff der Zukunft gelten.

Gleichsam verstricken wir uns in einer Informationsflut. Täglich werden neue Daten erfasst, wobei der größte Teil nicht analysiert und verarbeitet wird. Der Grund liegt in der Datenmenge, fehlenden Zeit und Komplexität. Stetig wird es für den Menschen schwieriger aufkommende Daten zu analysieren und zu verarbeiten. Manche Daten sind so komplex, dass die Zusammenhänge vom Menschen allein nicht erkannt werden können. Somit steigt der Bedarf an rechnergestützten Verfahren um diesem Problem zu begegnen.
Oft werden Daten in Form von mehrdimensionalen Datensätzen abgelegt und sind somit schwer durch den Menschen erfassbar. Dies läßt sich am folgenden kleinen Beispiel leicht nachvollziehen. Es stammt aus [Gin05] und demonstriert die Relevanz des Themas.
Das Labor eines Krankenhauses ermittelt für 20 Patienten den „Gehalt an alkalischen Phosphaten“ und den „Eisengehalt“. Die Daten werden dabei tabellarisch erfaßt.

Patient-Nr.	AP(U/l)	FE(mg/l)	Patient-Nr.	AP(U/l)	FE(mg/l)
1	4.0	1.0	11	2.0	0.7
2	3.0	1.7	12	1.2	0.5
3	2.6	1.8	13	4.5	0.7
4	1.5	0.7	14	2.5	3.0
5	2.5	2.2	15	3.5	0.7
6	1.1	1.0	16	2.2	3.2
7	2.8	3.1	17	2.1	3.5
8	1.7	3.2	18	2.1	2.0
9	0.8	0.5	19	3.5	1.2
10	2.1	3.0	20	3.2	1.2

Schon bei dieser geringen Datendimensionalität und der Beschränkung auf nur 20 Patienten, fällt es einem Menschen schwer, den Zusammenhang in den Daten zu erkennen. Oder würden Sie die 4 Cluster, welche die Befunde für „Hepatitis“, „Leberzirrhose“, „Verschlußikterus“ und „Normal“ wiederspiegeln, erkennen und die Personen einordnen können? Was, wenn noch mehr Daten

pro Patient erfasst werden oder 10000 Patienten beurteilt werden sollen?
In der folgenden Arbeit soll ein Verfahren beschrieben werden, mit dem dieses Problem programmtechnisch gelöst werden kann. Das Besondere an diesem Verfahren ist, dass es lernfähig ist und neue Muster generalisiert werden. Die Analyseergebnisse können auf dem Bildschirm als Grafiken angezeigt werden und lassen eine sofortige visuelle Interpretation durch den Menschen zu. Man kann also die Zusammenhänge durch Betrachten von Bildern erkennen. Die Arbeit stellt einige Möglichkeiten der Visualisierung vor und vertieft sie durch die Entwicklung einer Beispielanwendung namens SOMARFF.

1.3 Aufbau der Arbeit

Der erste Teil der Arbeit befasst sich mit den theoretischen Grundlagen.
Zu Beginn wird knapp auf die „Künstliche Intelligenz“ im Allgemeinen und ihre Teilbereiche eingegangen.
Danach erfolgt eine Eingrenzung auf das Gebiet „Künstliche Neuronale Netze“. In diesem Teil geht der Autor auf die Vor- und Nachteile dieser Technologie ein. Es folgt eine Erläuterung des Grundmodells eines Neurons und eine kurze Vorstellung verschiedener Netzarchitekturen.
Anschließend setzt sich der Autor mit dem Thema „Lernen mit Künstlichen Neuronalen Netzen“ auseinander. Dazu erfolgt eine Klärung des Begriffs „Lernen“ und eine Ausarbeitung über die Möglichkeiten, um ein Lernen in „Künstlichen Neuronalen Netzen“ zu realisieren. Abschließend erfolgt eine kurze Vorstellung der verschiedenen Lernverfahren.
Das Kapitel „Kohonennetze“ bildet den theoretischen Kern der Arbeit. Es befaßt sich ausführlich mit dem biologischen Vorbild, dem prinzipiellen Aufbau und dem Lernalgorithmus dieser Netzarchitektur. Außerdem wird auf die Kodierung von Eingabemustern eingegangen und wichtige Anhaltspunkte, um gute Lernergebnisse zu erhalten, erläutert. Abschließend werden einige praktische Beispiele und Anwendungsmöglichkeiten aufgezeigt.
Der nächste Teil der Arbeit geht auf die Visualisierungsmöglichkeiten von „Kohonennetzen“ im Allgemeinen ein. Danach erfolgt eine Auseinanderset-

zung mit einigen Visualisierungsarten, ihrem Aufbau und Nutzen.
Nun erfolgt der Übergang in den praktischen Teil der Arbeit. Er ist in Konzeption, Implementierung und Test gegliedert.
Das Konzept beschäftigt sich mit der Ausarbeitung des Pflichtenheftes, dem Entwurf der Systemarchitektur und der Beschreibung der verwendeten Entwurfsmuster der zu erstellenden Software. Danach wird auf die Wahl der Programmiersprache, sowie die Datenein- und ausgabe eingegangen. Anschließend erfolgt eine Analyse der auftretenden Anwendungsfälle und Systemzustände.
Das Kapitel „Implementierung“ erläutert zuerst ausführlich den Programmaufbau mit Quellcode und UML-Klassendiagrammen der Softwarepakete. Darauf folgt die Bescheibung der Programmoberfläche und ein kleines Einführungsbeispiel.
Im Kapitel „Test“ erfolgt die Analyse von mehreren Datenfiles mit Behauptung, Durchführung und Ergebnisbeurteilung.
Den abschließenden Teil der Arbeit bildet das Kapitel „Bewertung und Ausblick“. In ihm wird kurz auf die Probleme der Arbeit eingegangen und die gewonnenen Erkenntnisse zusammengefasst sowie einige abschließende Worte des Autors gegeben.
Im Anhang findet der interessierte Leser die Historie der „Künstlichen Intelligenz“ allgemein und detailiert für den Bereich der „Künstlichen Neuronalen Netze“.

Kapitel 2

Theoretische Grundlagen

2.1 Künstliche Intelligenz

Künstliche Intelligenz (engl. „Artificial Intelligence“) ist ein Teilgebiet der Informatik und ist Gegenstand vieler Mißverständnisse und falscher Erwartungen. Künstlich bedeutet, dass eine Sache einem natürlichen Vorbild nachgebildet wird. Unter Intelligenz versteht man im Allgemeinen die Klugheit, Einsichtigkeit und geistige Begabung. Durch die wörtliche Übersetzung wird oft fälschlich angenommen, dass mit Hilfe von Künstlicher Intelligenz Maschinen denken können. Um diese Fähigkeit nachweisen zu können, wurde von Alan Turing Anfang der 50er Jahre ein Test entwickelt, der Turing-Test. Dabei kommuniziert ein menschlicher Fragesteller mit 2, von ihm abgeschirmten, Probanden. Proband 1 ist ein Computer, Proband 2 ein Mensch. Der Fragesteller weiß nicht, wer wer ist und muß nun herauszufinden, ob er mit dem menschlichen Probanden oder mit dem Computer redet. Dieser Test wurde bis heute nicht bestanden und seit Anfang der 90er Jahre ist ein Preis auf das beste Turing-Test-Programm im Rahmen eines Wettbewerbs ausgeschrieben (vgl. [CJ01],[JH01]).
Künstliche Intelligenz ist eine wissenschaftliche Disziplin mit dem Ziel, menschliche Wahrnehmungs- und Verstandesleistungen in Teilaufgaben zu zergliedern und zu präzesieren sowie durch technische informationsverarbeitende Systeme verfügbar zu machen (vgl. [RCR03]). Charakteristisch für das me-

thodische Vorgehen in der Künstlichen Intelligenz ist es, für den Beobachter scheinbar intelligentes Verhalten zu produzieren. Dabei wird der „umgekehrte Ansatz“ verfolgt. Es wird also zuerst ein System geschaffen, welches intelligentes Verhalten produziert. Anschließend wird versucht, daraus die menschliche Intelligenz verstehen zu lernen. Die Konzentration liegt auf der Lösung von Problemen, welche sich nicht, oder nicht effizient, durch einen Algorithmus lösen lassen. Die Künstliche Intelligenz beschäftigt sich mit Problemen, welche oft auf ungenauen, fehlenden oder schlecht definierten Informationen basieren.
Sie hat einen interdisziplinären Charakter und steht in engem Zusammenhang mit der Philosophie, Psychologie, Linguistik und den Neurowissenschaften. Diese Wissenschaften sind wichtig für die kognitionswissenschaftliche Komponente der Künstlichen Intelligenz. Im Unterschied zur klassischen Informatik legt die Künstliche Intelligenz besonderen Wert auf Wahrnehmung, Schlußfolgerung und Handeln. Im Gegensatz zur Psychologie liegt der Schwerpunkt auf der Berechnung.
In der Künstliche Intelligenz werden 2 Teilaspekte unterschieden.

1. kognitive Modellierung
   Dies ist die Simulation kognitiver Prozesse durch Informationsverarbeitung.

2. Konstruktion intelligenter Systeme
   Dies ist die maschinelle Verfügbarmachung von menschlichen Wahrnehmungs- und Verhaltensleistungen.

Für höhere Intelligenzfunktionen ist das Beurteilen von Situationen, und daraus Schlußfolgerungen ziehen, unabdingbar. Schlußfolgerndes Denken gibt einem Individuum die Fähigkeit, Handlungsweisen zu bewerten und auszuwählen, bevor eine Handlung vollzogen wird. Um dies zu erreichen, wird auf das Wissen der Welt sowie auf alternative Handlungsmöglichkeiten in dieser Welt zurückgegriffen.
Einige wichtige, bei einer Problemlösung bemerkbare, Merkmale von Intelligenz sind

- Lösungsart
- Effizienz
- Geschwindigkeit
- Adaption (Anpassung an Umwelt)
- Assimilation (Anpassung der Umwelt)

Unter Intelligenten Leistungen versteht man abstrakte Denkleistungen, wie logisches Denken, Rechnen, Gedächtnis und die Reflexion, wie auch den Umgang mit Sprache, das Erkennen von Gegenständen und Situationsabläufen, das Kombinieren von Lösungsmöglichkeiten und Kreativität. Demzufolge kann die Intelligenz in zwei Teile zergliedert werden.

- Leistungen durch personales Handeln,
  sie sind grundsätzlich mit formalen Systemen darstellbar, also auch durch einen Computer berechenbar.
- Leistungen durch nicht-personales Handeln,
  z.B. Kreativität und Bewußtsein.

Intelligente Leistungen, welche nicht-personales Handeln erfordern, sind für viele Forscher nicht berechenbar. Andere vertreten die „Harte KI-These".
Die „Harte KI-These" besagt, daß sämtliche Bewußtseinsprozesse nichts anderes als Berechnungsprozesse sind. Sie wurde noch nie bewiesen. Es gibt noch eine „Schwache KI-These". Sie besagt, dass Intelligenz auch Informationsverarbeitung ist.
Die Künstliche Intelligenz verfolgt die Grundidee, dass Intelligenz das Produkt der Interaktion vieler kleiner einfacher Prozesse ist und dadurch zur Entwicklung einer höheren Leistungsstufe aus einer niederen, mit einer höheren Qualität, erfolgt (Emergenz). Man kann also sagen, daß die Gesamtheit aller Teile mehr ist, als nur die Summe der Teilsysteme. Durch diese Modularisierung werden Prozessmodelle intelligenten Verhaltens durch den Computer im Detail untersuchbar. Die Systemarchitektur ist für solche „intelligenten" Systeme besonders wichtig, da sie für die Entstehung der Synergieeffekte

grundlegend ist.
Zusammenfassend kann man sagen, dass die Künstliche Intelligenz verschiedene Konzepte wie Logik, Wissensrepräsentation, Schlussfolgern, Heuristik und Suchalgorithmen, Lernen, Planen, Neuronale Netze, Genetische Algorithmen erforscht und in verschiedensten Bereichen einzusetzen versucht. Einige Bereiche davon sind das Lösen von Problemen, Mustererkennung, Verarbeitung natürlicher Sprache sowie Spracherkennung, maschinelles Übersetzen, Spiele (zB. Schach, Roboterfussball), Expertensysteme, kognitive Modellierung, KI-Programmiersprachen, Data-Mining, Agentensysteme, Robotik, Computervision und Künstliches Leben.(vgl. [HM00a])

2.1.1 KI-Teilgebiete

Die Künstliche Intelligenz besteht aus folgenden Teilgebieten.

Teilgebiet	Erläuterung
Automatisches Beweisen	Prüfen von mathematischen Beweisen, wobei Programme auf Lücken und Fehler in der Beweisführung hinweisen. Auch fällt das Finden von Widersprüchen und Zuständen in Datenbanken in diesen Bereich. Durch Einsatz bestimmter Programmiersprachen kann eine Problemlösung beschleunigt bzw. sogar erst ermöglicht werden, zB. Prolog.
Expertensysteme	Dieses Teilgebiet beschäftigt sich mit Systemen, welche Wissen über ein eng abgrenztes Gebiet speichern und aus diesem Wissen neue Schlussfolgerungen für die Problemlösung ziehen können.

Natürlich-sprachliche Kommunikation	Hier steht die Erkennung von Sprache, das Verarbeiten der darin enthaltenen Infomationen, sowie die Erzeugung von natürlicher Sprache zur Verständigung mit dem Menschen.
Bildverstehen und Animation	Das Teilgebiet befaßt sich mit dem Erkennen von Mustern in Pixelbildern sowie deren Interpretation. Außerdem versucht man hier Filme und Bilder zu synthetisieren, zB. um kybernetische Künstliche Welten zu erzeugen.
Robotik	Durch Künstliche Intelligenz lassen sich autonome Roboter konstruieren, wodurch sie in die Lage versetzt werden, ihre Umwelt zu bewerten und sich selbstständig in dieser Welt zu bewegen und zu planen. Ein bekanntes Beispiel dafür ist das Roboterfussball welches seit einigen Jahren als Meisterschaft ausgetragen wird.

(vgl.[ML01])

2.2 Künstliche Neuronale Netze

2.2.1 Über die Vor- und Nachteile Künstlicher Neuronaler Netze

Wenn mit einem Computer Probleme gelöst werden sollen, wird in der Regel ein Algorithmus entwickelt, dieser in einem Computerprogramm umgesetzt und Schritt für Schritt abgearbeitet. Oft geht dabei viel Zeit für die Erarbeitung, Implementierung, Test und Verbesserung verloren.
Weiterhin setzt diese Vorgehensweise die Existenz eines Algorithmus für das Lösen des zu bearbeitenden Problems voraus und stellt den Programmierer manchmal vor eine scheinbar unlösbare Aufgabe. Ändert sich die Problemstellung oder fallen Teile der Eingaben weg, arbeiten diese Programme nicht mehr korrekt und müssen vom Programmierer neu angepasst werden.

Die Natur liefert ein Vorbild, um diese Probleme meistern zu können. Es sind Neuronale Netze, wie z.B. das menschliche Gehirn. Das Gehirn des Menschen befähigt ihn zu Leistungen wie Gehen, Radfahren, Gesichtserkennung, Reden, Hören, Schreiben oder Lesen. Alle diese Fähigkeiten haben etwas gemeinsam, sie laufen im Unterbewußtsein und damit ohne Nachdenken ab. Nachdenken heißt, aus vorhandenem Wissen Schlussfolgerungen für neues Wissen ziehen. Ein Beispiel dafür ist das Formulieren eines Algorithmus in einer Programmiersprache. Tätigkeiten wie z.B. Radfahren können nicht mit Nachdenken funktionieren. Eine Verarbeitung der Sensordaten (Auge, Gleichgewicht etc.) würde zu lange dauern und es sind schnelle Reaktionszeiten gefordert. Neuronale Netze haben hier entscheidende Vorteile. Sie können ihr Verhalten antrainieren und sind nach der Trainingsphase ohne „Nachdenken“ einsetzbar.
Neuronale Netze können lernen und sich an neue Problemstellungen anpassen, ohne dass die Grundstruktur geändert werden muss. Fallen Neuronen aus, so kommt nicht das gesamte System zum Erliegen, sondern es stellt sich ein Leistungsabfall ein. Aufgrund seiner Struktur verarbeitet es Informationen hochgradig parallel, was trotz geringer Taktrate von ca. 1kHz, Leistungen ermöglicht, zu welchen viele Hochleistungsrechner bei herkömmlicher Programmierung nicht im Stande sind.(vgl. [Spi00],[JH01])
Künstliche Neuronale Netze, kurz KNN, sind der Versuch, dieses Vorbild im Computer nachzubilden. KNN benötigen keinen Algorithmus für die Problemlösung. Statt dessen lernt das Netz selber eine Lösung, in dem es trainiert wird. Sie erlernen ihre Fähigkeiten durch Nachahmung, also durch „Lernen aus Beispielen“. Hierbei kommen überwachte und auch unüberwachte Lernverfahren zum Einsatz (vgl. [JH01]).
Ein Nachteil ist, dass diese Trainingsphase sehr viel Zeit in Anspruch nehmen kann. Bei iterativem Lernen sind viele tausend Lernschritte nicht unüblich. An die Trainingsphase schließt sich die Arbeitsphase an, in welcher sehr wenig Rechenzeit aufgewendet werden muss.
Aufgrund der Parallelität steht das Netzergebnis kurz nach Eingabe zur Verfügung. Dies macht auch Anwendungen möglich, die auf extrem schnelle Reaktionen des Systems angewiesen sind, z.B. Steuerung von Flugkörpern. Ei-

ne Abarbeitung durch einen Algorithmus benötigt hier wesentlich mehr Zeit. Leider kann der Vorteil der Parallelität nur auf Parallelrechnern ausgenutzt werden (vgl. [Ten95],[Spi00]).
Ein Problem sollte dann mittels Neuronalen Netzen gelöst werden, wenn kein effizienter Algorithmus verfügbar ist und regelbasierte Ansätze gescheitert sind (vgl. [JH01]).

2.2.2 Begriffe, Definitionen, Funktionen

Grundgrößen für die Modellierung von Künstlichen Neuronalen sind

- die Aktivität (Erregung) und die Erregungsdynamik
- die Übertragungsgewichte und die Gewichtsdynamik
- die Netzwerktopologie und die Netzwerkdynamik

(vgl.[JH01])

Künstliche Neuronale Netze lassen sich nach ihrer Trainierbarkeit (trainierbar, nicht trainierbar), dem verwendeten Lernverfahren und ihrer Architektur unterscheiden. Bei trainierbaren Netzen ist die Anzahl der Lernzyklen, Menge an Neuronen und die Verfügbarkeit von einem Set an ausreichenden Trainingsdaten für eine erfolgreiche Entwicklung von entscheidener Bedeutung. Außerdem ist für einen Test auf Generalisierungsfähigkeit häufig ein 2 Satz an Testdaten notwendig. Modifikationsmöglichkeiten finden sich bei Künstlichen Neuronalen Netzen in den Lernparametern, dem Lernalgorithmus sowie bei der Netzarchitektur (vgl. [CJ01]).
In Bezug auf die Netzaufgabe werden 3 Grundtypen unterschieden.

1. Klassifikatoren,
   sie teilen den Raum der Inputvektoren in Klassen, z.B. gut, mittel und schlecht ein. Im einfachsten Fall findet eine Zuordnung durch ein einzelnes Outputneuron eines Beispiels zu einer Klasse statt (zugehörig/nicht zugehörig).
   Beispiel: Perzeptron

2. Assoziatoren,
   sie ordnen den Eingabevektoren die zugehörenden Outputneuronen zu. Dabei ist die Anzahl der assoziativen Paare relativ klein und auf jeden Fall endlich.
   Beispiel: Hopfield-Netz

3. Abbildende Netzwerke,
   sie dienen der Approximation von Funktionen und Operationen $\Re^I \rightarrow \Re^J$ durch eine Zuordnung zwischen beliebigen Inputvektoren und zugehörigen Outputvektoren.
   Beispiel: SOM

Neuronarten

Neuronale Netze bestehen aus 3 Typen von Neuronen.

- Eingabeneuronen,
  auch afferente Neuronen genannt, stellen die Schnittstelle des Neuronalen Netzes zur Außenwelt dar. Sie reagieren nicht auf synaptische Verbindungen, sondern auf äußere Einflüsse. Es gibt keine gerichtete Verbindung von einem Neuron j zu einem Eingangsneuron i. Die Menge aller Eingabeneuronen wird als Input-Layer bezeichnet.
  Sensor$\longrightarrow$Eingangsneuron

- Innere Neuronen,
  auch Hidden- oder innere Neuronen genannt, werden von den Eingangsneuronen beeinflußt und gehören weder zur Gruppe der Eingabe- noch zu den Ausgabeneuronen. Man nennt sie versteckt, weil sie keine direkte Verbindung zur Netzaußenwelt haben. Innere Neuronen können auf mehrere verdeckte Schichten, auch Hidden-Layer bezeichnet, verteilt werden.

- Ausgangsneuronen,
  auch efferente Neuronen genannt, stellen das Verarbeitungsergebnis der Netzverarbeitung dar. Sie verwirklichen die Einwirkung des Netzes auf

die Außenwelt. Bei vorwärstgerichteten Netzen erkennt man Ausgangsneuronen daran, dass sie keine Verbindungen zu anderen Neuronen besitzen. Ihre Synapsen enden an den Aktoren.
Ausgangsneuron⟶LED

(vgl. [CJ01],[ZS98])

2.2.3 Das Neuron als Modell

Im biologischen Vorbild gruppieren sich die Neuronen zu funktionalen Einheiten, Säulen genannt. Bei künstlichen neuronalen Netzen wird eine Säule vereinfacht durch ein einzelnes Neuron dargestellt. Neuronen sind Schaltelemente, welche viele Eingangssignale in ein Ausgangssignal umwandeln. Ein allgemeines Modell eines Neurons läßt sich dabei folgend verstehen. Ein

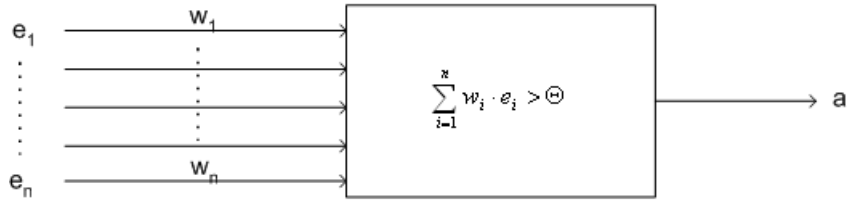

Abbildung 2.1: Modell eines Neurons

Neuron entspricht einem Prozessor, welcher nur 2 Zustände annehmen kann. „feuern“ und „nicht feuern“, bzw. 0 und 1. Die Eingangssignale werden an die Eingänge $e_1 ... e_n$ angelegt und bilden den Eingabevektor. Die erregende oder hemmende Wirkung der Synapsen wird in der Regel durch eine einfache Multiplikation der Eingangsgewichte mit positiven oder negativen reellen Zahlen $w_1 ... w_n$ realisiert. Das Neuron berechnet anschließend seine Aktivierung, indem es die Summe der gewichteten Eingangssignale, auch als Netzaktivität *net* bezeichnet, mit seinem Schwellwert Θ vergleicht. Wenn *net* größer ist als der Schwellwert Θ, dann feuert das Neuron. Das bedeutet, das Neuron legt auf seinem Ausgang a eine 1, ansonsten die 0. Da der Ausgang wieder den Eingabevektor nachfolgender Neuronen bilden kann, besteht der Eingabevektor aus den binären Werten 0 und 1. Der Schwellwert wird auch BIAS

genannt, hat den Standardwert 1 und wird für die Berechnung von net, mit zu den Gewichten gerechnet $(-\Theta)$. In der Regel wird als Propagierungsfunktion fast ausschließlich die Summe der eingehenden gewichteten Eingänge verwendet (vgl. [CJ01]). Statt der Summation kann jedoch auch die Multiplikation verwendet werden. Formal läßt sich ein Neuron wie folgt ausdrücken, wobei e den Eingangsvektor (Dendriten), w den Gewichtsvektor (Synapsen), net die Summe der gewichteten Eingangssignale, Θ den BIAS und a den über die Aktivierungsfunktion berechneten Ausgang (Axon) darstellt. Man nennt die Aktivierungsfunktion auch Transferfunktion. Als Aktivierungsfunktion $a = f(x)$ wird die binäre Funktion verwendet.
Eingangswerte e, für $e \in \{0, 1\}$
Synapsenwerte w, für $w \in \Re$

$$net = \sum_{i=0}^{n} e_i \cdot w_i \tag{2.1}$$

$$a = f(net - \Theta) = f(x) \begin{cases} 1 & \text{falls } x > 0 \\ 0 & \text{sonst } x \leq 0 \end{cases} \tag{2.2}$$

Dazu ein kleines Beipiel. Ein Neuron hat 2 Eingaben e_0 und e_1, welche mit 1 belegt sind. Das Gewicht w_0 ist 2 und w_1 ist 1. Der BIAS beträgt 2. Die

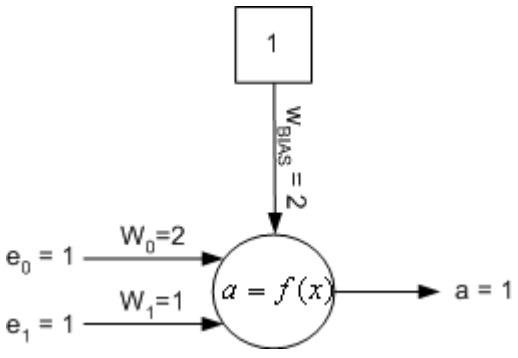

Abbildung 2.2: Beispielberechnung des Aktivierungszustandes eines Neurons mit 2 Eingabesignalen

Aktivierung errechnet sich nun folgendermaßen.

$net = e_0 \cdot w_0 + e_1 \cdot w_1 = 1 \cdot 2 + 1 \cdot 1 = 2 + 1 = 3$
$a = f(net - \Theta) = f(3 - 2) = f(1) = 1$

Die Beschreibung dieses Modellneurons kann für die Entwicklung von Künstlichen Neuronalen Netzwerken, weiterhin als KNN bezeichnet, in Teilen abgewandelt werden. So kann als Ein- und Ausgangssignal auch $e, a \in \Re$ gelten. Oft wird statt $e, a \in \{0, 1\}$ auch $e, a \in \{-1, 1\}$ verwendet (Signumfunktion). Wenn Differenzierbarkeit der Erregung gefordert ist, so kann die Sigmoidfunktion eingesetzt werden. Dies wird zum Beispiel beim Backpropagation-Algorithmus eingesetzt. Die Konstante k gibt dabei die Steilheit des Überganges von 0 nach 1 an. Bei $k \to \infty$ geht die sigmoide Funktion in die binäre Funktion über.

$$a = \frac{1}{1 + e^{-k\alpha}} \tag{2.3}$$

2.2.4 Netzarchitekturen

Bei der Entwicklung von Künstlichen Neuronalen Netzen unterscheidet man vorwärts verkettete und rückgekoppelte Architekturen, Architekturen aus vollvernetzten Schichten, autoassoziative und wachsende Architekturen.(vgl. [CJ01])

Vorwärts verkettete Architekturen

Diese Netzarchitektur wird auch „feed forward“-Netz genannt und besteht mindestens aus der Eingabe- und der Ausgabeschicht. Zusätzlich kann es notwendig werden, eine oder mehrere Schichten mit inneren Neuronen zu implementieren. Bei diesem Architekturtyp werden nur Neuronen aus verschiedenen Schichten miteinander verbunden. Die Verbindung der Neuronen erfolgt ausnahmslos in Richtung der Ausgabeneuronen. Beispiele für Netze, welche diesen Architekturtyp verwenden sind Multilayerperzeptron und Madaline.(vgl. [CJ01])

Rückgekoppelte Architekturen

Bei dieser Art von Netzen existieren Rückkopplungsverbindungen von den Ausgangsneuronen zurück zu den inneren Neuronen. Dadurch fließt die Netzausgabe $f(x)_t$ mit in die Netzausgabe $f(x)_{t+1}$ ein. t stellt dabei den Zeitpunkt der Netzausgabe dar. Durch die Realisierung dieses „Eingabefensters“, also z.B. aktueller Wert und vorheriger Wert, werden auch Prognose-Probleme lösbar. Jordan-Netze begründen sich auf diesem Architekturtyp. (vgl. [CJ01])

Architekturen aus vollvernetzte Schichten

Bei dieser Art von Netzarchitektur werden sämtliche Neuronen untereinander vollständig vernetzt. Eine Ausnahme davon bilden die Eingabeneuronen. Auch eine direkte Rückkopplung der Neuronen mit sich selber findet nicht statt. Diese Art von Netzarchitektur ist bei den Kohonen-Netzen verwirklicht und in einem späteren Kapitel genauer betrachtet.(vgl. [CJ01])

Autoassoziative Architekturen

Den Grundgedanken für die Architektur bei autoassoziativen Netzen liefert das Teilchenverhalten in Metallen. Wie bei den Netzen aus vollvernetzten Schichten, sind die Neuronen einer Schicht vollständig miteinander verbunden. Auch hier existieren keine Rückkopplungen der einzelnen Neuronen mit sich selber. Im Unterschied zu den vollvernetzten Schichten, bestehen autoassoziative Netze nur aus einer einzigen Neuronenschicht. Daraus schließt sich, dass auch keine spezielle Eingabeschicht, sowie eine spezielle Ausgabeschicht existiert.
Die Netzeingabe erfolgt bei diesem Netztyp durch eine initiale Aktivierungsvorgabe aller Neuronen. Aufgrund der Aktivierung der Neuronen zum Zeitpunkt t erfolgt durch gegenseitige Beeinflußung sämtlicher Neuronen untereinander die Aktivierung zum Zeitpunkt $t+1$. Die alte Aktivierung (Eingabe) hat also Einfluß auf die nächste Aktivierung, welche dann wieder als Eingabe fungiert. Das Netz kommt somit in einen schwingenden Zustand. Erreicht das Netz einen stabilen Zustand, wird die Netzausgabe durch die Aktivie-

rungen der Neuronen gebildet. Hopfield-Netze und die Bolzmann-Maschine begründen sich auf diesem Architekturtyp. ([CJ01])

Wachsende Architekturen

Die Frage nach der geeigneten Menge an Neuronen für ein Künstliches Neuronales Netz kann in vielen praktischen Anwendungen zu Problemen führen. Ein Versuch, dieses Problem zu vermeiden, stellen Netze aus wachsenden Architekturen dar. Dabei wird das Netztraining mit einer geringen Anzahl von Neuronen begonnen. Im Laufe des Trainings werden weitere Neuronen hinzugefügt, bis ein gewünschtes Verhalten erreicht ist. Dies kann dadurch erfolgen das z.B. zwischen den zwei schlechtesten Neuronen ein neues Neuron eingefügt wird. Kennzeichnent ist dabei der Netzfehler. Wachsende neuronale Gase sind ein Beispiel für diesen Architekturtyp. ([CJ01])

2.2.5 Lernen mit Künstlichen Neuronalen Netzen

Nach [unb06b] bedeutet Lernen,

> ... etwas in irgend welcher Weise, durch Anweisung, Beispiel oder Erfahrung gelehrtes sich aneignen.

bzw. versteht man unter Lernen nach [unb06d]

> Durch Erfahrung bedingte Veränderung einer Verhaltensweise.

Wie kann man ein Künstlichen Neuronales Netz dazu bringen zu Lernen? Theoretisch gibt es dafür 7 Arten, um ein Lernen zu ermöglichen.

1. Entwicklung neuer Verbindungen
2. Löschen existierender Verbindungen
3. Modifikation der Stärke w_{ij} von Verbindungen
4. Modifikation des Schwellwertes von Neuronen
5. Modifikation der Aktivierungs-, Propagierungs- oder Ausgabefunktion,

6. Entwicklung neuer Zellen

7. Löschen von Zellen

Diese Arten können einzeln oder gemischt eingesetzt werden, wobei die 3 die meist genutzte Art ist. Mit 3 können auch 1, 2 und 4 realisiert werden. 5 wird wenig genutzt. Die Arten 6 und 7 gewinnen zunehmend mehr an Bedeutung, da durch sie eine möglichst optimale Netztopologie für das gestellte Problem gefunden werden kann. (vgl. [Zel03])
Das Einstellen der Synapsengewichte auf möglichst optimale Gewichte kann auf 3 Arten geschehen.

1. direktes Interpretieren und manuelles Setzen der Gewichte
   Diese Aufgabe wird durch den Systementwickler durch eigene Berechnungen durchgeführt und scheidet oft wegen der Komplexität des Problems aus.

2. durch globale Analytische Verfahren
   Das analytische Verfahren wird von einem global unabhängigen Prozess durchgeführt. Das Maß der Abweichung zwischen aktuellen und gewünschtem Netzverhalten ergibt den zu minimierenden Funktionswert. Ein Beispiel dafür ist das Newtonsche Approximationsverfahren. Diese Art wird meist vernachlässigt, weil das Netzwerk stets ganzheitlich beachtet werden muss, was sehr unhandlich ist. Außerdem ist ein globaler Controller notwendig.

3. Lernen durch Selbstorganisation
   Bei diesem Verfahren ist die Aufgabe der Gewichtsbestimmung in das Netzwerk und seine Arbeitsweise integriert. Diese Art ist in der Praxis besonders wichtig und auch die meist verwendete. Die Selbstorganisation, also das Lernen, erfolgt dabei nach einfachen Regeln. Durch schrittweises Abändern jedes einzelnen Gewichtes in Richtung des gewünschten Outputvektors wird das Netzverhalten dem Ziel immer weiter angenähert. Dabei kann durch explizite Feedbacks ein überwachtes Lernen realisiert werden. Unüberwachtes Lernen erhält man allein durch Vorgabe von Restriktionen.

(vgl. [Dor91])
Es wird nun auf die dritte Art der Gewichtsbestimmung eingegangen. Das konnektionistische Lernen folgt dabei 2 Prinzipien. Erstens soll der Algorithmus zur Gewichtsanpassung lokal in den Neuronen oder an ihren Verbindungen erfolgen können und zweitens soll ein möglichst zerstörungsfreies Lernen in Bezug auf bereits gespeichertem Wissen realisiert werden (siehe Lernkonstante).
Man unterscheidet 2 Lernparadigmen, und zwar das iterative Lernen, sowie das nicht iterative Lernen.

- iterative Lernverfahren

  A. überwachtes Lernen

  B. rückgekoppeltes Lernen

  C. unüberwachtes Lernen

- nicht iterative Lernverfahren
  Bei diesen Verfahren werden die Gewichte direkt ausgerechnet. Dies kann nach dem Funktionsziel (z.B. Comparator-Netzwerk), nach der Codierung der Eingänge (z.B.. RAM) oder nach der Fehlerkompensation der Gewichte (Querpropergation-Netz mit Antineuronen) erfolgen.

(vgl. [Z98])

A. - Überwachtes Lernen

Beim überwachten Lernen ist die gewünschte Ausgabe zu bestimmten Eingaben bekannt. Es muß ein Lernalgorithmus implementiert werden, welcher die Gewichte so abändert, dass sich die Ausgabe für möglichst alle Eingaben verbessert. Dazu wird ein Fehlermaß (Netzfehler) eingeführt, welches im Laufe des Lernens minimiert wird. Als Fehlermaß besonders beliebt ist der durchschnittliche quadratische Fehler, auch MSE - für „mean square error“ genannt, über alle Trainingsdaten. Ein Lernschritt läuft beim überwachten Lernen in der Regel folgendermaßen ab:

1. Eingabevektor anlegen

2. Eingabevektor verarbeiten
3. Differenz zwischen Soll- und Ist-Ausgabe bestimmen (Fehlermaß)
4. Fehlerminimierung durch Modifikation der Gewichte (z.B. durch Backpropagation)

Ziel des Trainings ist es, zu unbekannten, aber den Trainingsdaten ähnlichen Eingaben, eine möglichst korrekte Ausgabe zu liefern. Dies kann nur an Daten nachgewiesen werden, die nicht im Training verwendet wurden. Aus diesem Grunde werden die vorhandenen Daten in 2 Sets zerlegt (Trainingsset, Testset). Das Hauptziel ist, den Fehler im Testset zu minimieren. Sind zu viele Verbindungsgewichte oder zu wenig Testdaten vorhanden, besteht das Phänomen des Overfittings. Dabei wird vom Neuralen Netz nicht die hinter den Daten stehende Regel gelernt, sondern das Netz lernt die Trainingsdaten auswendig. Erkennen kann man das Overfitting daran, dass die Fehlerrate im Trainingsset weiter fällt, im Testset jedoch steigt. In diesem Fall sollte das Training abgebrochen werden. In der Regel wird genau dieser Punkt angestrebt.(vgl. [unb06e])

B. - Rückgekoppeltes Lernen

Beim Rückgekoppelten Lernen, auch bestärkendes Lernen oder „reinforcement learning genannt“, sind die Neuronen so miteinander verkoppelt, dass sich ein kreisförmiger Signallauf bildet. Durch die Beeinflussung der Signalrückführung werden solange Änderungen an den Ausgängen Y^* durchgeführt, bis kein Unterschied mehr zwischen dem Eingangsvektor X und der Rückführgröße X_r besteht(vgl. [ZS98]).
Beim rückgekoppeltem Lernen wird dem Netz nur vermittelt, ob die Ausgabe richtig oder falsch war, evtl. auch der Grad der Richtigkeit. Explizite Zielwerte für die Ausgabeneuronen sind nicht vorhanden. Dies bedeutet, dass das Neuronale Netz die richtige Ausgabe für die Neuronen selber finden muss. Dieses Verfahren ist biologisch plausibler als das überwachte Lernen. Sowohl bei niederen als auch höheren Wesen kann man einfache Rückkopplungsmechanismen, z.B. durch Bestrafung und Belohnung, beobachten. Leider benö-

tigt man beim rückgekoppeltem Lernen wesentlich mehr Zeit für Aufgaben, bei denen die erwünschte Ausgabe genau bekannt ist, als beim überwachten Lernen. Der Grund dafür ist, dass bei diesem Lernen weniger Informationen zur korrekten Modifikation der Gewichte zur Verfügung stehen.(vgl. [Zel03])

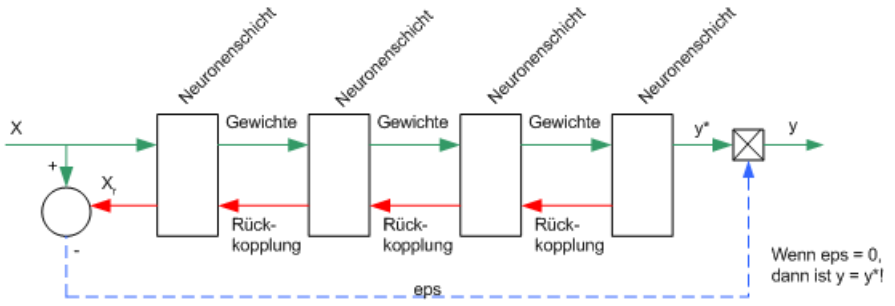

Abbildung 2.3: Lernprozess mittels Rückkopplung

C. - Unüberwachtes Lernen

Beim unüberwachten Lernen besteht die Trainingsmenge nur aus den, dem Netz angebotenen, Eingabedaten. Für das Lernen werden keine gewünschten Aus- oder Angaben genutzt. Das Lernen funktioniert ohne, dass das Netz eine Information darüber bekommt, ob die Daten richtig oder falsch klassifiziert wurden. Der Algorithmus versucht dazu selbstständig Cluster, also Gruppen, zu identifizieren und diese auf den Eingabevektoren ähnliche Neuronen, sowie deren Nachbarn, abzubilden. Mathematisch gesehen, extrahiert der Traingsprozess, die statistischen Eigenschaften der Trainingsdaten.
Das bekannteste Beispiel, welches das unüberwachte Lernen einsetzt sind die „Selbstorganisierenden Karten“, von Teuvo Kohonen. Auf diesen Netztyp wird im Kapitel „Kohonennetze“ besonders eingegangen.
Dieses Lernverfahren ist biologisch gesehen, am plausibelsten. Der Beweis, dass dieses Prinzip in der Natur angewendet wird, ist durch Untersuchungen an Säugetierhirnen nachgewiesen worden. (vgl. [Zel03])
Das Lernziel besteht darin, die Eingabemuster auf Neuronen, welche der

Wahrscheinlichkeitsdichtefunktion der Verteilung der Eingabemuster entsprechen, abzubilden. Ähnliche Eingabevektoren werden in ähnliche Klassen eingeteilt.
Durch unüberwachtes Lernen lassen sich auch Klassifikationsprobleme lösen. Sind Referenzvektoren vorhanden, dessen Klassenzugehörigkeit bekannt ist, so läßt sich die Klasse eines beliebigen Eingabevektors durch Berechnung der Nähe zu den Referenzvektoren bestimmen. Der Eingabevektor wird dann wie der nächstliegende Referenzvektor klassifiziert. (vgl. [Zel03])

Desweiteren werden mehrere Lernregeln unterschieden.

- Fehlerkorrektur
- Wahrscheinlichkeitslernen
- Konkurrenzlernen
- Hebbsche Regel

Im folgenden werden die Lernregeln kurz erläutert.

Lernen durch Fehlerkorrektur

Beim Lernen durch Fehlerkorrektur erfolgt das Einstellen der Gewichte durch die Optimierung des Netzfehlers E. Das Verfahren gehört zu den überwachten Lernverfahren, der Soll-Output d des Netzes ist bekannt und E wird mit der Formel

$$E = d - y(t) \tag{2.4}$$

berechnet. Der Netzfehler E ergibt sich dabei durch die Differenz zwischen dem gewünschten Soll-Output d und dem IST-Output y zum Zeitpunkt t. Bei der anschließenden Gewichtskorrektur wird das Gewicht W der Neuronenverbindung i zum Zeitpunkt t addiert mit dem Ergebnis der Fehlerfunktion $f(E)$.

$$W_{i(t+1)} = W_{i(t)} + f(E) \tag{2.5}$$

Der Netzfehler E kann positiv oder negativ sein und führt dazu, dass sich das Gewicht erhöht oder senkt. Wenn $y == d$ eintritt, ergibt sich $E = 0$, womit das Gewicht W nicht geändert wird. In $f(E)$ wird E mit einem Lernfaktor α multipliziert. Für ihn gilt $0 < \alpha < 1$. Die Fehlerkorrekturregel wird z.B. bei den Netztypen Perzeptron, Adaline und Madaline angewendet (vgl. [ZS98], S.26).

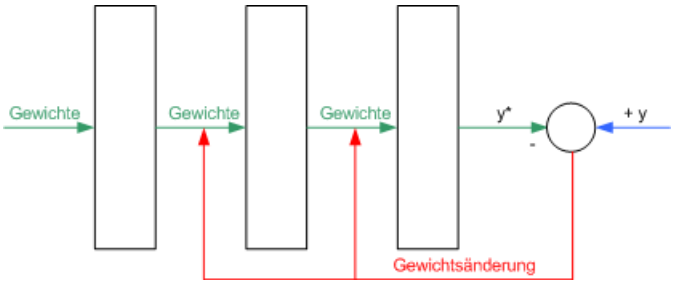

Abbildung 2.4: Lernen mittels Fehlerkorrektur (nach [ZS98])

rueckgekoppelteStruktur.png > Verweis Lernfaktor.

Wahrscheinlichkeitslernen

In autoassoziativen Netzen, wie dem Hopfield-Netz, erfolgt das Lernen nicht durch das Training der Verbindungsgewichte zwischen den Neuronen, sondern folgt aus den angelegten Mustern heraus. Die Netzeingabe ist somit die durch das angelegte Muster vorgegebene Aktivierung der Neuronen. Dabei verhalten sich Neuronen wie elektrisch geladene Atome. Nach Anlegen eines Musters arbeitet das Netz solange, bis das Netz einen stabilen Zustand erreicht hat. Dies geschieht durch gegenseitige Beeinflussung zwischen den Neuronen. Stabil bedeutet, dass keine Änderungen in der Aktivierung mehr erfolgt. Man sagt auch, dass Netz schwingt in einen stabilen Zustand. Ein wichtiges Merkmal ist die Netzernergie E welche durch das Netz im Laufe des Trainings minimiert wird. Jedes Muster entspricht dabei einem Energiewert, nach welchem sich die Muster klassifizieren und später erkennen lassen.[ZS98] Der Energiewert errechnet sich wie folgt.

$$E = -\frac{1}{2}\sum\sum(W_{ji} \cdot x_j \cdot x_i) \tag{2.6}$$

Wenn man autoassoziative Netze mit versteckten Neuronen erweitert, entsteht ein Lernen durch Wahrscheinlichkeit. Die Netzdynamik ist abhängig von der Netzenergie E sowie der Temperatur T des Systems. Neuronen kennen bei diesem Lernprozess nur 2 Zustände [0,1] oder [-1,1].

Bei Energieänderung um den Wert δE geht Neuron i_t in i_{t+1} über. Das neue Gewicht für Neuronenverbindung W_i ergibt sich durch

$$W_{i(t+1)}(+/-) = \frac{1}{1 + e^{-\frac{\triangle E_i}{T}}} \tag{2.7}$$

Um die Wahrscheinlichkeit eines solchen Zustandsübergangs zu berechnen, werden Formeln der statistischen Thermodynamik benutzt, wodurch das Netzverhalten nicht nur von der Netzenergie E, sondern auch von seiner „Temperatur“ T abhängig ist.
Es gibt beim Lernen nach Wahrscheinlichkeitsregeln 2 Betriebmodi.

Modus 1: Lernen Anpassung der Neuronen an die Eingangsvektoren

Modus 2: Erkennen Netz konvergiert zusammen mit verdeckten Neuronen in einen stabilen Zustand

In der Lerndatei werden die Neuronenzustände S für diese beiden Modi angegeben. Der “+“-Index steht für die Wahrscheinlichkeit, dass dieser Zustand beim Lernen erreicht wird. Der „-“-Index gibt an, mit welcher Wahrscheinlichkeit der Zustand beim Erkennen-Modus erreicht wird.

$$P^+(S_1), P^+(S_2), ...P^+(S_\alpha), ...P^+(S_r) \tag{2.8}$$

$$P^-(S_1), P^-(S_2), ...P^-(S_\alpha), ...P^-(S_r) \tag{2.9}$$

Aus diesen Angaben wird der Abstand G zwischen den beiden Mengen (P^+, P^-) berechnet.

$$G = \sum_\alpha P^+(S_\alpha) \cdot \ln \frac{P^+(S_\alpha)}{P^-(S_\alpha)} \tag{2.10}$$

Daraus ergibt sich die Gewichtsänderung $\triangle W_{ij}$ durch folgende Formel.

$$\triangle W_{ij} = \eta \cdot (p_{ij}^{+} - p_{ij}^{-}) \tag{2.11}$$

p_{ij}^{+} und p_{ij}^{-} sind die Wahrscheinlichkeitsmittelwerte dafür, dass sowohl beim Lernen, als auch beim Erkennen die Ausgänge mit 1 aktiviert werden.(vgl. [ZS98])

Konkurrenzlernen

Konkurrenzlernen gehört zur Gruppe der unüberwachten Lernverfahren, so dass keine Informationen über gewünschte Zustände (z.B. Soll-Output) für das Lernen nötig sind. Grundsätzliches Prinzip beim Konkurrenzlernen ist das Aussieben von Neuronen, welche dem Eingabevektor am besten entsprechen. Diese Neuronen heißen Gewinner-Neuronen und werden beim Lernschritt weiter in Richtung des Eingabevektors angenähert. Die Gewichte der Verbindungen der Gewinner-Neuronen werden verstärkt, die Verbindungsgewichte der restlichen Neuronen werden verkleinert. Es gibt 4 Verfahren des Konkurrenzlernens:

1. Competitive Lernen (CMPL)
   Um CMPL zu realisieren, werden die Neuronen der verdeckten Schicht oder der Ausgangsschicht eines Multilayerperzeptrons vollständig miteinander verbunden. Diese Verbindung ist stets negativ, die Neuronenverbindungen zwischen den Schichten haben ausschließlich positiven Charakter (vgl. [ZS98]).

2. Interactive Activision und Competition (IAC)
   Im Vergleich zum CMPL dürfen die Neuronenverbindungen beim IAC zwischen den Schichten auch negative Gewichte besitzen. Beim IAC bestehen die Neuronenschichten aus Clustern. Innerhalb dieser Cluster sind die Neuronenverbindungen negativ gewichtet. Die Verbindungen zwischen Clustern innerhalb einer Schicht sind positiv. Beim Lernen erhält das Gewinner-Neuron den maximalen Gewichtswert, die restlichen Neuronen sinken auf minimale Werte ab (vgl. [ZS98], S.30f).

3. Kohonen-Regel
   Bei diesem Lernverfahren wird ein Neuronennetz aus 2 Schichten gebildet. Die erste Schicht besteht aus Eingangsneuronen. An sie wird der Eingabevektor angelegt. Die zweite Schicht wird Kartenschicht oder auch Kohonenschicht genannt. Sie besteht aus vollständig miteinander verbundenen Neuronen (Gitterstruktur). Beide Schichten sind vollständig miteinander, durch gewichtete Verbindungen, vernetzt. Bei Eingabe eines Eingabevektors wird dann ein Neuron ermittelt, welches einen Gewichtsvektor besitzt, welcher dem Eingangsvektor am ähnlichsten ist. Dieses Neuron wird Gewinnerneuron genannt. Im Laufe des Lernens werden die Gewichtsvektoren an die Eingabemuster angepasst, was zu einer topologischen Abbildung des n-dimensionalen Eingaberaumes in einen m-dimensionalen Ausgaberaum führt. Im Kapitel zu den Kohonen-Netzen wird auf dieses Verfahren tiefer eingegangen.

4. Grossberg-Regel (Adaptive Resonance Theorie, ART)
   Bei diesem Lernverfahren werden neue Ausgangsneuronen für neue Klassen angelegt. Begonnen wird mit einer minimalen Ausgangsneuronenzahl, welche einer vorgegebenen Klasseneinteilung entspricht. Bei Eingabe eines Inputvektors entscheidet das Netz, ob der Eingabevektor zu einer bestimmten, durch ein Ausgangsneuron repräsentierten Klasse gehört. Ausschlaggebend dafür ist der Abstand, in welchem sich der Eingangsvektor zum Ausgangsneuron befindet. Wird kein passendes Ausgangsneuron gefunden, so wird ein neues Ausgangsneuron mit entspechenden Gewichten angelegt.

Hebb'sche Lernregel

These von Donald Hebb:
„Sofern ein Axon der Zelle A einer Zelle B nahe genug ist, um sie immer wieder zu erregen bzw. dafür zu sorgen, dass sie feuert, findet ein Wachstumsprozess oder eine metabolische Veränderung in einer der beiden Zellen oder in beiden statt, so dass die Effektivität der Zelle A, die Zelle B zu erregen gesteigert wird .(Zitat aus [Spi00])“

Anders ausgedrückt, wenn 2 Zellen A nach B über das Axon nah genug miteinander verbunden sind um sich wieder zu erregen, findet ein Wachstumsprozeß statt, der dafür sorgt, dass sich die Erregungseffektivität weiter verbessert. Ihre Verbindung verstärkt sich. Zur Nachbildung im Computer läßt sich dieser Effekt durch folgende Formel ausdrücken.

$$\triangle W_{ij}(t) = \eta \cdot y_j(t) \cdot x_i(t) \tag{2.12}$$

Der Wert, um welchen das Gewicht W_{ij} zum Zeitpunkt t vergrößert wird, ergibt sich durch Multiplikation von Ausgang y_j mit Eingang x_i sowie einem Lernfaktor η. Das Gewicht steigt beim mehrmaligen Anwenden dieser Regel exponentiell an. Dieses ist vermeidbar durch die Einbringung eines Begrenzungsfaktors α und führt zu folgender Formel.

$$\triangle W_{ij}(t) = \eta \cdot y_j(t) \cdot x_i(t) - \alpha \cdot y_j(t) \cdot W_{ij}(t) \tag{2.13}$$

Man kann den Lernfaktor η und den Begrenzungsfaktor α auch zu einer Konstante $C = \frac{\eta}{\alpha}$ zusammenfassen.

$$\triangle W_{ij}(t) = \alpha \cdot y_j(t) \cdot [C \cdot x_i(t) - W_{ij}(t)] \tag{2.14}$$

(vgl. [Z98])

2.3 Kohonennetze

2.3.1 Biologisches Vorbild

Alle biologischen Zellen reagieren auf Umweltreize, beispielsweise durch Berührung, Temperatur oder Chemikalien. Diese Reize beinflussen die Durchlässigkeit der Zellmembranen für Ionen. Zellen, die sich auf diesen Effekt spezialisiert haben, nennt man Neuronen. Sie sind besonders leitfähig und erregbar. Ein Neuron besteht aus dem Zellkörper, den Dendriten, den Synapsen und einem Axon. Die Dendriten sind baumartige weite Verzweigungen, welche für die Reizzuführung benutzt werden. Sie sind über eine chemische Verbindung zur Zellmembran, der Synapse, gebunden. Da Synapsen nicht

immer gleich gut leitfähig sind, erfolgt die Erregung von Neuron zu Neuron stets mehr oder weniger stark. Dies ist für die Funktion des Gesamtsystems sehr wichtig. Für die menschliche Großhirnrinde, auch Kortex genannt, sind Synapsen zwischen 1000 und 10000 je Neuron üblich. Die Verarbeitung erfolgt im Zellkörper. Das Axon dient dem Aussenden der Aktionspotenziale und geht in die Dendriten der nachgeschalteten Neuronen über. Somit erfolgt die Zuleitung der Eingangssignale zu einem Neuron über die Zuleitung der Ausgänge der anderen Neuronen.

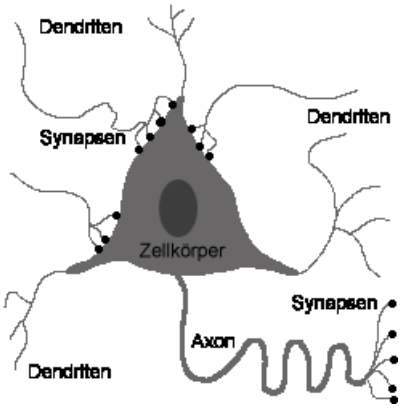

Abbildung 2.5: biologischer Aufbau eines Neurons

Bei Reizung kommt es zu einem Potentialunterschied an der Zellmembran. Das Ruhepotential liegt beim Menschen bei ca. 70 mV. Bei Erregung des Neurons wird das Ruhepotential mehr und mehr reduziert. Wird das Ruhepotential um mehr als 20 mV unterschritten, kommt es zu einer raschen Abfolge von Aktionspotentialen. Man sagt, das Neuron „feuert“. Diesen zu unterschreitenden Spannungswert nennt man Schwelle. Zuerst treten die Aktionspotentiale nur an bestimmten Orten der Zellmembran auf, breiten sich jedoch schnell entlang des Axons aus. Da Neuronen untereinander rückgekoppelt sind entsteht ein instabiler Zustand, welcher zur Informationsübertragung und Informationsverarbeitung ausgenutzt wird (vgl. [Spi00]). Nach der Entladung folgt eine Phase totaler Erregungsunempfindlichkeit.

Neuronen sind im Kortex nicht wahllos miteinander vernetzt, sondern bilden funktionelle Einheiten, auch Säulen genannt. Eine Säule besteht aus

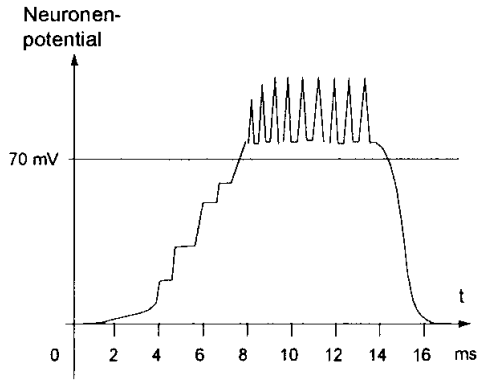

Abbildung 2.6: Potentialoutput am Axon bei Reizung

mehreren eng beieinander liegenden Neuronen, welche sich gegenseitig erregen und auch für die Weiterleitung der Erregung an benachbarte Säulen verantwortlich sind. Aufgrund ihrer Form nennt man sie auch Pyramidenzellen. Außerdem enthält eine Säule Interneuronen, welche für die Hemmung weiter entfernter Säulen zuständig sind. Eine durch einen bestimmten Reiz erregte Säule erregt somit ihre unmittelbaren Nachbarsäulen mit und hemmt weiter entfernte Säulen. Man nennt dieses Prinzip laterale Inhibition (Hemmung) (vgl. [Spi00]). Mehrere Säulen ergeben eine Neuronenkarte.

Es wurde nachgewiesen, dass die Verarbeitung im Gehirn nicht durch ein einzelnes großes Neuronennetz erfolgt, sondern in einzelne Module zerlegt wird. So existieren im Kortex topografische Karten z.B. für Hören, Tasten und Sehen. Diese Karten stellen die Eingangssignale, welche von den Eingangssensoren kommen, bezüglich ihrer Aufgabe geordnet nach Ähnlichkeit, Häufigkeit und Relevanz dar (vgl. S[00] S.116, 121). Häufigere Eingabesignale bilden dabei mehr größere Bereiche in der Karte als geringer auftretende Eingabesignale. Ein Beispiel dafür findet man beim Tastsinn. Sämtliche Tastempfindungen werden im Kortex durch einen bestimmten Bereich repräsentiert. Innerhalb dieses Bereiches findet man eine landkartenähnliche Repräsentation der Körperoberfläche, wobei die zugeordneten Neuronenbereiche nicht proportional zur Fläche, sondern zu deren Bedeutung sind. So bekommen Lippen und Hände wesentlich mehr Neuronen zugewiesen als die

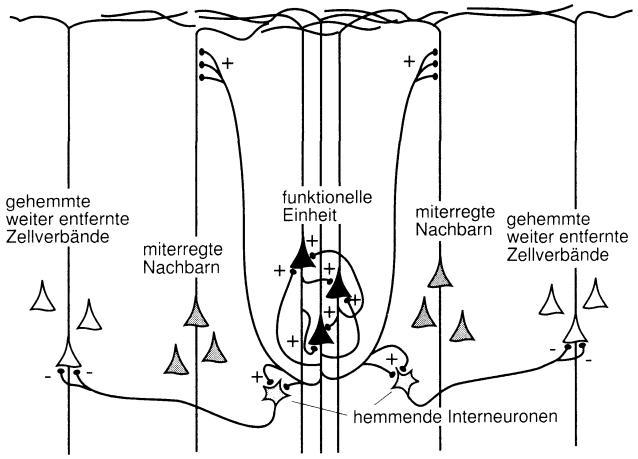

Abbildung 2.7: Verschaltung der Neuronen im Kortex (aus [Spi00])

Fläche des Rückens (vgl S[00] S.116). Ein anderes Beispiel dafür ist der Hörsinn der Fledermaus. Fledermäuse orientieren sich im Dunkeln mittels Ultraschall. Dazu stößt die Fledermaus einen 30 ms dauernden Ton von genau 61 kHz aus. Aus Untersuchungen weiß man, dass für die Verarbeitung von Tönen speziell um 61 kHz wesentlich mehr Neuronen zuständig sind als für die restlichen Töne. Sie kann also in einem bestimmten Frequenzbereich besonders gut hören. Analog dazu findet sich beim Menschen eine Zone schärfsten Sehens (Foeva), welche im Hirn durch einen größeren Neuronenbereich verarbeitet wird als die Randzonen der Netzhaut (vgl. S[00]).
Neuronale Karten dienen oft als Eingabesignal für nachfolgende Karten. Durch das Prinzip der Modularisierung können Aufgaben besser bewältigt werden, da jeweils nur Teilaufgaben gelöst werden müssen. Beispielsweise wird zur visuellen Wahrnehmung beim Menschen nach der Vorverarbeitung der Eingangssignale in der Netzhaut eine erste Verarbeitung in der primären Sehrinde (V1) vorgenommen. Der Output der V1-Karte liefert dann die Inputsignale für die Karten V2 bis V5. Diese Karten sind für die Verarbeitung bestimmter visueller Reize zuständig, z.B. V4 für das Farbsehen o. V5 für Bewegungssehen. (vgl. [Spi00]) Ein weiteres wichtiges Merkmal ist die To-

pologieerhaltung. Sensorenreize, welche nah beieinander liegen, also ähnlich sind, werden auch in der Karte durch beieinander liegende Neuronenbereiche bearbeitet. Die Repräsentation der Hautoberfläche der Hand liegt z.B. neben der des Armes. Evolutionsmäßig betrachtet, ist die kortikale Struktur als sehr erfolgreich zu betrachten und wurde stetig vergrößert.

2.3.2 Prinzipieller Aufbau

Im Kapitel zum biologischen Vorbild von Kohonennetzen wurde der Aufbau der Großhirnrinde (Kortex) sowie ihrer charakteristischen Merkmale in Bezug zur Verarbeitung von Eingabesignalen betrachtet. Wie kann man nun diese Struktur mit einem Computer nachbilden? Neuronale Karten werden im Prinzip genauso wie im biologischen Vorbild zusammengesetzt.

SOM-Architektur

Der finnische Ingenieur Teuvo Kohonen entwickelte mit Hilfe des Neuronenmodells einen Netzwerktyp der dem biologischen Vorbild sehr ähnlich aufgebaut ist und ganz ähnliche Eigenschaften besitzt. Dieser Netzwerktyp wurde nach ihm benannt und heißt Kohonennetz, auch als Selbstorganisierende Merkmalskarten bekannt. Kohonennetze bestehen aus 2 Schichten. Der Eingabeschicht und der Kartenschicht. Für die Kartenschicht wird meistens eine 2-dimensionale quadratische Neuronenanordnung verwendet. Es sind aber auch 1-dimensionale und 3-dimensionale Anordnungen in unterschiedlichsten Formen möglich (vgl. [CJ01]). Es ist wichtig, dass die Kartenschicht aus besonders vielen Neuronen besteht, damit Strukturen durch Emergenz sichtbar werden. Teuvo Kohonen hat nachgewiesen, dass ein 2-dimensionales Netzwerk genügt, um komplexe, auch hochdimensionale, Eigenschaften abzubilden. Die Neuronen der Eingabeschicht sind mit den Neuronen der Kartenschicht voll vernetzt. Das heißt, jedes Neuron der Eingabeschicht besitzt genau eine gewichtete Verbindung zu jedem Neuron der Kartenschicht. Die Anzahl der Attribute der Eingabemuster, also die Dimensionen der Muster, bestimmt die Menge der Eingabeneuronen. Zum Anpassen der Eingabe kann die Anzahl der Eingabeneuronen auch erhöht werden. Obwohl im späteren

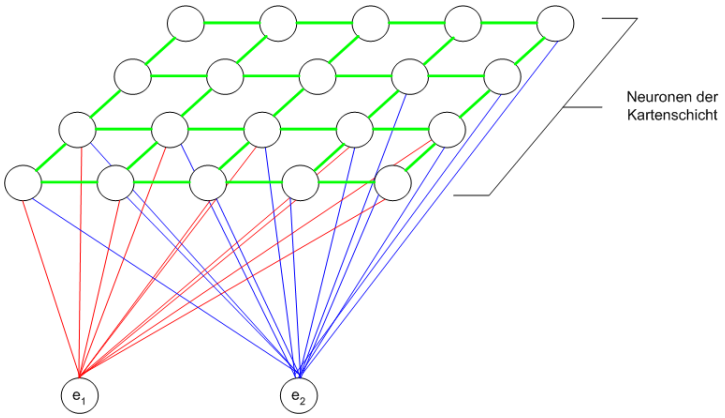

Abbildung 2.8: Aufbau einer SOM1

Lernalgorithmus keine Rückkopplungen zwischen den Neuronen der Kartenschicht berücksichtigt werden, sind sämtliche Neuronen der Kartenschicht untereinander vollständig verbunden.

Die Anordnung der Neuronen in der Kartenschicht kann sowohl in einem Quad-Gitter, als auch im Hex-Gitter erfolgen. Die hexagonale Implementierung soll dabei bessere Ergebnisse hervorbringen, ist jedoch schwerer zu implementieren. Anstatt einer planaren Topologie, bei welcher ein Kartenrand

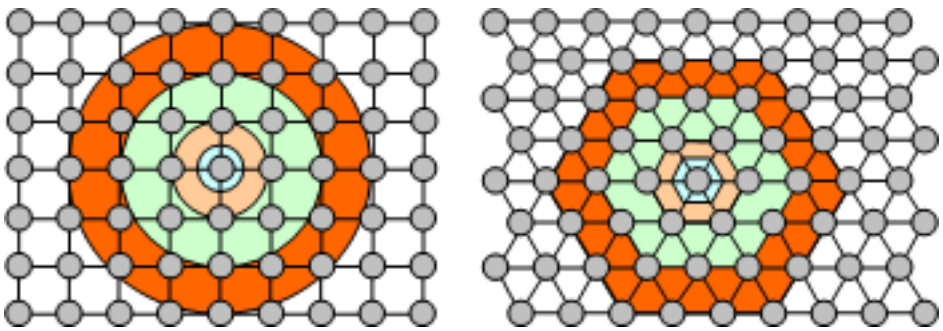

Abbildung 2.9: Neuronengitternetze der Kartenschicht (nach [Ult06])

vorhanden ist, kann die Kartenschicht auch als randloser Körper gestaltet werden, z.B. als Ring. Das bedeutet, dass der obere Rand mit dem unteren Rand verbunden ist und der linke Rand mit dem rechten Rand. Dadurch wer-

den beim Lernvorgang Effekte vermieden, bei denen die Cluster nur an den Kartenrändern gebildet werden und die Kartenmitte größtenteils frei bleibt. (vgl. [Ult06])

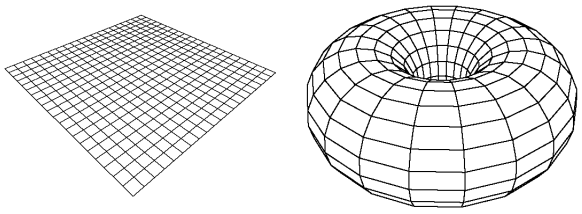

Abbildung 2.10: planare und toroide Topologie (aus [Ult06])

2.3.3 Lernalgorithmus

Kohonennetze werden nach dem Prinzip des Wettbewerbslernen trainiert. Das bedeutet, für die Aktivierung eines Neurons ist nicht nur die eigene Netzeingabe ausschlaggebend. Für die endgültige Aktivierung findet ein Vergleich der Netzeingaben aller Neuronen statt.
Bei diesem Vergleich wird ein „Siegerneuron“ ermittelt, was einer Klassifizierung der Netzeingabe gleichkommt. Bei der anschließenden Gewichtskorrektur unterscheidet man zwischen „hartem Wettbewerbslernen“, wobei nur die Gewichte des Gewinnerneurons angepasst werden, und dem „weichen Wettbewerbslernen“, bei dem auch eine Gewichtsänderung der Nachbarneuronen stattfindet.
Der Lernvorgang bei einem Kohonen-Netz erfolgt im Prinzip immer in 5 Schritten.

1. Initialisierung der Gewichte bei Lernstart
   In der Regel werden zu Beginn des Lernens sämtliche Synapsengewichte zufällig und mit kleinen Gewichten initialisiert, z.B. durch Anwendung der Normalverteilung oder innerhalb eines festen Intervalls. Je nach Problem kann aber auch eine spezielle Methode notwendig sein. So z.B. bei der Lösung des „Problem des Handlungsreisenden“, wobei die

Gewichte der Startroute angelehnt sind, z.B. Kreis, Quadrat.
Nach [unb06c] ist es jedoch auch möglich, alle Verbindungsgewichte auf einen Wert zu setzen. Dieser Wert kann entweder durch den Anwender vorgegeben werden, oder auch abhängig von der Anzahl der Eingabeneuronen sein. Ein globaler fester Initialisierungswert ist unproblematisch, da bei mehreren Siegerneuronen eines nichtdeterminsitisch ausgewählt und der symetrische Anfangszustand aufgebrochen wird. Oftmals kann mit einem festen Initialisierungswert die Netzkonvergenz schneller erreicht werden. Ein fester Wert kann durch folgende Formel berechnet werden.

$$W_{init} = \frac{1}{\sqrt{X_N}} \tag{2.15}$$

X_N steht dabei für die Anzahl der Eingabeneuronen. (vgl. [unb06c])

2. Eingabevektor anlegen
   Als Nächstes wird der Eingabevektor an die Eingabeneuronen angelegt. Der Eingabevektor bildet dabei ein Beispiel aus der zu lösenden Problemumgebung ab. Prinzipiell kann man dies auch wie einen Datensatz aus einer Datenbanktabelle verstehen. Wichtig ist, das die Beispiele insgesamt Regelmäßigkeiten aufweisen. Werden nur neue Muster angeboten, wird nichts gelernt. Die Datenmenge kann bei der Auswahl von Beispielen iterativ oder auch zufällig durchlaufen werden.

3. Gewinner-Neuron ermitteln
   Bei diesem Schritt wird ermittelt, welches Neuron dem Eingabevektor am ähnlichsten ist, also das Neuron, welches auf Grund der augenblicklichen Gewichtsverteilung am stärksten erregt wurde. Dieses Neuron wird Gewinnerneuron genannt. Um dieses Neuron zu ermitteln, können 2 mathematische Methoden genutzt werden, die „Euklidische Distanz“ und das „Skalarprodukt“. Dieser Schritt kann, entgegen dem biologischen Vorbild, in der künstlichen Umsetzung idealisiert werden. Dies geschieht dadurch, dass während des Lernens Rückkopplungen zwischen den Neuronen nicht berücksichtigt werden und dadurch kom-

pliziertere Berechnungen der Wechselwirkung zwischen einem potenziellen Gewinnerneuron und seiner Umgebung vermieden werden. Die Auswirkung auf das Ergebnis ist nicht gravierend, macht das Verfahren jedoch einfacher.(vgl. [Kin92], [CJ01])

4. Gewichte anpassen
   Steht das Gewinner-Neuron fest, so werden seine Synapsenstärken um den Lernfaktor ϵ weiter in Richtung des Eingabevektors verschoben. Daraus folgt, dass das Gewinnerneuron bei nochmaligem Anlegen des Eingabevektors noch eindeutiger erregt wird. Auch die umliegenden Neuronen des Gewinner-Neurons werden in die Gewichtsanpassung mit einbezogen, in der Regel aber nicht so stark wie das Gewinner-Neuron selbst. Dadurch wird die Wahrscheinlichkeit erhöht, dass bei erneutem Angebot dieses Eingabevektors dieselbe Neuronengruppe reagiert. (vgl. [Kin92])
   Diese Erregungsausbreitung kann durch unterschiedlichste Funktionen realisiert werden.

   - Mexikaner-Hut-Funktion
   - Gaussche Glockenkurve
   - Pyramide, Kegel, Zylinder, Quader

   In der Praxis werden die 2 erstgenannten Funktionen am meisten genutzt. Auch hat sich herausgestellt, dass die im biologischen Vorbild auftretende Hemmung am Rande des Erregungsgebietes nicht realisiert werden muss. Nach [CJ01] hat sich gezeigt, dass dadurch der Berechnungsaufwand steigt und das Neuronale Netz schlechter konvergieren kann. Neuronen, welche nicht im Erregungsgebiet liegen, sind von der Gewichtsänderung nicht betroffen. Wichtige Variablen für diesen Schritt sind der Lernfaktor ϵ, die Erregungsausbreitung sowie der Faktor, um den sich die Erregungsausbreitung während des Lernens verringert.

5. neuen Eingabevektor anlegen ($\uparrow$2)

Bei diesem iterative Verfahren erfolgt musterweises Lernen und es wird standardmäßig benutzt. Ein Lernschritt wird nach Anlegen und Auswerten eines einzigen Musters vollzogen. Man nennt das Verfahren auch „online-learning“ oder „pattern-mode-learning“. (vgl. [unb06c])
Es gibt jedoch auch ein anderes Verfahren, welches in der Praxis ein schnelleres Lernen ermöglicht. Bei diesem Verfahren wird in Epochen gelernt. Die Anpassung der Gewichte erfolgt erst nach Beendigung einer Epoche. Unter einer Epoche versteht man die einmalige Abarbeitung aller Muster. Es werden also nacheinander alle Muster einmal an das Netz angelegt. Bei jedem Muster erfolgt die Berechnung der Gewichte wie beim iterativen Verfahren, jedoch werden die Gewichte nur im „stillen“ verändert, also nicht tatsächlich. Stattdessen merkt sich das Netz die berechneten Gewichtsänderungen und akkumuliert sie. Sind alle Muster einmal durchlaufen, also eine Epoche beendet, so werden die akkumulierten Gewichtsänderungen auf einmal tatsächlich durchgeführt. Dieses Verfahren wird auch als „offline-learning“ oder „batch-mode-learning“ bezeichnet. (vgl. [unb06c])

Ermittlung des Gewinnerneurons über „Euklidischen Abstand“

Der „Euklidische Abstand“ definiert den normalen Abstand zwischen 2 Punkten oder Vektoren im Raum. Je kleiner dabei der Abstand zwischen den Punkten oder Vektoren ist, also je kleiner der Betrag, desto „ähnlicher“ sind die Punkte oder Vektoren. Die Formel dafür lautet:
$d(x, y) = ||x - y||$
$d(x, y) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + \ldots + (x_n - y_n)^2}$
$d(x, y) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$
Um dieses Verfahren beim Lernalgorithmus einzusetzen, wird als Vektor X der Eingangsvektor der Eingangsneuronen angenommen. Den Y-Vektor erhält man durch Verwendung der Gewichte W_{ij} des zu testenden Neurons i. j bezeichnet den Index des jeweiligen Gewichtes.
Berechnungsbeispiel:

Anhand der Abbildung ist ersichtlich:

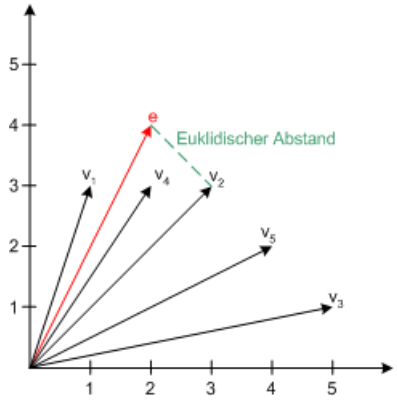

Abbildung 2.11: Euklidischer Abstand

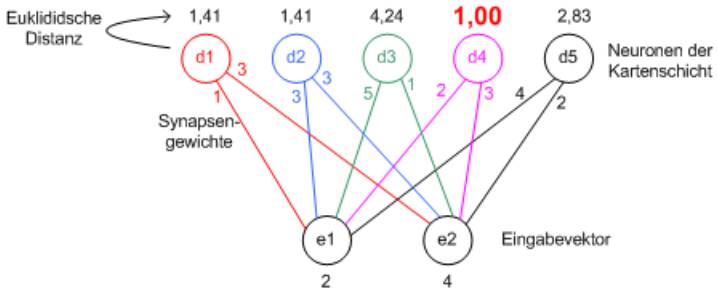

Abbildung 2.12: Beispiel-KNN zur Berechnung des Euklidischen Abstandes

Eingabevektor $\vec{e} = \begin{pmatrix} e_1 \\ e_2 \end{pmatrix} = \begin{pmatrix} 2 \\ 4 \end{pmatrix}$
und die Gewichtsvektoren $\vec{v_1}, \vec{v_2}, \vec{v_3}, \vec{v_4}, \vec{v_5}$:

$\vec{v_1} = \begin{pmatrix} v_{11} \\ v_{12} \end{pmatrix} = \begin{pmatrix} 1 \\ 3 \end{pmatrix}$

$\vec{v_2} = \begin{pmatrix} x_{21} \\ y_{22} \end{pmatrix} = \begin{pmatrix} 3 \\ 3 \end{pmatrix}$

$\vec{v_3} = \begin{pmatrix} x_{31} \\ y_{32} \end{pmatrix} = \begin{pmatrix} 5 \\ 1 \end{pmatrix}$

$\vec{v_4} = \begin{pmatrix} x_{41} \\ y_{42} \end{pmatrix} = \begin{pmatrix} 2 \\ 3 \end{pmatrix}$

$\vec{v_5} = \begin{pmatrix} x_{51} \\ y_{52} \end{pmatrix} = \begin{pmatrix} 4 \\ 2 \end{pmatrix}$

Daraus lassen sich nach der Formel folgende Distanzen zwischen den Vektoren $\vec{v_i}$ und dem Eingangsvektor $\vec{e}$ berechnen.

$d_{\vec{v_i}} = \sqrt{(x_{\vec{v_e}} - x_{\vec{v_i}})^2 + (y_{\vec{v_e}} - y_{\vec{v_i}})^2}$

$d_{\vec{v_1}} = \sqrt{(2-1)^2 + (4-3)^2} = \sqrt{1^2 + 1^2} = \sqrt{2} = 1,41$

$d_{\vec{v_2}} = \sqrt{(2-3)^2 + (4-3)^2} = \sqrt{(-1)^2 + 1^2} = \sqrt{2} = 1,41$

$d_{\vec{v_3}} = \sqrt{(2-5)^2 + (4-1)^2} = \sqrt{(-3)^2 + 3^2} = \sqrt{18} = 4,24$

$d_{\vec{v_4}} = \sqrt{(2-2)^2 + (4-3)^2} = \sqrt{0^2 + 1^2} = \sqrt{1} = 1,00$

$d_{\vec{v_5}} = \sqrt{(2-4)^2 + (4-2)^2} = \sqrt{(-2)^2 + 2^2} = \sqrt{8} = 2,83$

Wie man erkennen kann, gewinnt in diesem Beispiel das Neuron 4, da sein Gewichtsvektor den kleinsten euklidischen Abstand zum Eingabevektor aufweist.
Für den Fall, dass mehrere Neuronen den kleinsten Abstand aufweisen, so muss man sich entscheiden, welches man zum Sieger erklärt, z.B. das erste gefundene Neuron mit dem kleinsten Abstand.

Ermittlung des Gewinnerneurons über „Skalarprodukt“

Dies ist die zweite Möglichkeit, das Gewinnerneuron zu finden. Nach [Kin92] wird behauptet, dass das Skalarprodukt λ von $\vec{x} \cdot \vec{y}$ genau dann maximal ist, wenn sein euklidischer Abstand minimal ist. Voraussetzung dafür ist, dass $x^2 = z$ und $y^2 = z$ gilt und $\vec{x} \cdot \vec{y} \geq 0$ sind. Das Skalarprodukt λ für

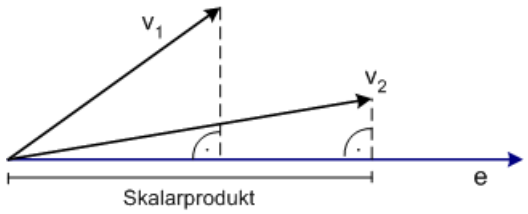

Abbildung 2.13: Skalarprodukt

die Vektoren $\vec{e}$, $\vec{v_i}$ läßt sich über folgende Formel ermitteln. $\cos\varphi$ bezeichnet

dabei den Winkel, welchen $\vec{e}$ und $\vec{v_i}$ einschließen.

$$\lambda = \vec{e} \cdot \vec{v_i} = |\vec{e}| \cdot |\vec{v_i}| \cdot \cos\varphi = x_e \cdot x_{v_i} + y_e \cdot y_{v_i} \tag{2.16}$$

Berechnungsbeispiel:

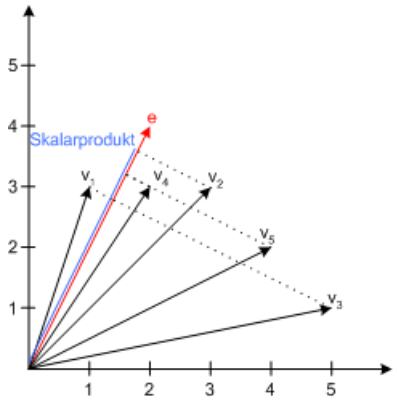

Abbildung 2.14: Beispielskizze Skalarprodukt

$\lambda_{v_i} = x_e \cdot x_{v_i} + y_e \cdot y_{v_i}$
$\lambda_{v_1} = 2 \cdot 1 + 4 \cdot 3 = 14$
$\lambda_{v_2} = 2 \cdot 3 + 4 \cdot 3 = 18$
$\lambda_{v_3} = 2 \cdot 5 + 4 \cdot 1 = 14$
$\lambda_{v_4} = 2 \cdot 2 + 4 \cdot 3 = 16$
$\lambda_{v_5} = 2 \cdot 4 + 4 \cdot 2 = 16$

In Fällen, in denen das Skalarprodukt der Vektoren $\vec{v_i}$ und $\vec{e}$ nicht auf $\vec{e}$ abgebildet werden kann, weil das Skalar außerhalb des Vektors $\vec{e}$ liegt, kann man durch tauschen von $\vec{e}$ und $\vec{v_i}$ in der Formel dennoch auf das Skalarprodukt schliessen.

$$\lambda_{(\vec{e},\vec{v_i})} = \vec{e} \cdot \vec{v_i} = \vec{v_i} \cdot \vec{e} \tag{2.17}$$

Wie man in der Skizze erkennen kann, ergibt sich bei dieser Lösung das Problem, dass Vektoren mit hohen Dimensionswerten bevorzugt werden. So würde im Beispiel ein Vektor $\vec{v_u}$ mit $x = 5, y = 3$ als Gewinner gefunden werden, obwohl er sich vom $\vec{v_e}$ deutlicher unterscheidet als die anderen Vektoren.

Aus diesem Grunde kann es sinnvoll sein, die Vektoren vor der Berechnung des Skalarproduktes zu normieren. Sie unterscheiden sich dann nicht mehr in ihrem Betrag $|\vec{v_i}|$, sondern lediglich im eingeschlossenen Winkel und damit in ihrer Richtung untereinander, welche wiederum Aufschluß über ihre Ähnlichkeit gibt (vgl. [Zel03]).

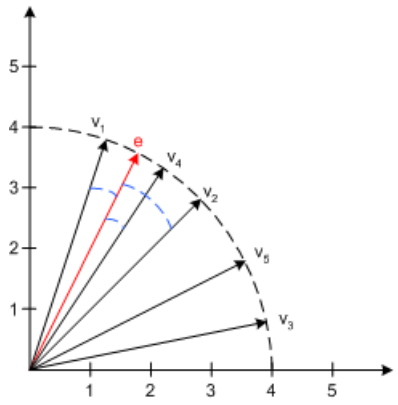

Abbildung 2.15: Beispielskizze normierte Vektoren zur Berechnung des Skalarproduktes

Anpassung der Synapsengewichte

Nachdem das Gewinnerneuron gefunden wurde, wird sein Gewichtsvektor dem Eingangsvektor so geändert, dass im Falle eines neuen Auftretens dieses Eingabevektors, das Neuron noch „besser“ erregt wird. Dies trifft auch auf seine Nachbarneuronen zu, wenn auch in abgeschwächter Form. Die Form und auch die Größe dieser Nachbarschaft hängt von der verwendeten Funktion und der Lernrate ab und kann sich im Lernverlauf ändern. Für die Eigenschaft der Topologieerhaltung ist es von größter Wichtigkeit, dass die Neuronen in der Nachbarschaft des Siegerneurons auch lernen! Nachbarneuronen müssen für das Lernen nicht „zweitähnlich“ o.ä. zum Eingabevektor sein, sondern es zählt für Nachbarneuronen nur die Position im Neuronennetz. (vgl. [unb06c]) Die Lernrate ist ein Parameter, der für die Größe der Gewichtsanpassung verantwortlich ist. Durch sie wird die Lerngeschwindigkeit gesteuert.

Beispiel:

Lernrate $\eta = 1$ Die Änderungen werden so groß, dass sich die Gewichte vollständig an den letzten Eingabevektor anpassen. Es wird also vorher Gelerntes vergessen und ist somit ungünstig.

Lernrate $\eta = 0.1 < x < 0.2$ Die Synapsengewichte werden um einen Betrag x geändert, welcher so klein ist, dass vorher Gelerntes behalten wird und neue Anpassungen nur zu einem Bruchteil mit in die Gewichte einfließen. Das Netzwerk erhält dadurch die Fähigkeit zum Generalisieren. Zu kleine Werte können sich jedoch ungünstig auf den Lernfortschritt auswirken, da sich die Lerndauer erhöht.

Wie man sieht, entsteht ein Konflikt zwischen notwendigem langsamen Lernen und der zur Verfügung stehenden Zeit als Kostenfaktor.
Interessanterweise lassen sich diese Probleme auch beim biologischen Vorbild beobachten. Lebewesen müssen einerseits möglichst gut lernen, haben dafür aber nicht unendlich viel Zeit. Außerdem steigt die Genauigkeit der Lernleistung nur um $\sqrt{n}$. n gibt dabei die Anzahl der Lernmuster an. Dies bedeutet, dass bei 100 gelernten Mustern, die ersten 25 so wichtig für die Anpassung der Neuronengewichte sind, wie die restlichen 75. Es ist also notwendig, für ein Verdoppeln der Lernleistung, 4x soviel Lernmuster anzubieten. (vgl. [Spi00])
Die Biologie löst dieses Problem, indem in den ersten Lebensabschnitten eine besonders hohe Lernrate angesetzt ist, diese aber im Laufe des Lebens abnimmt. Dies hat zur Folge, dass anfangs rasche Änderungen der Synapsengewichte für schnelle aber grobe Erfassung entscheidener Parameter stattfinden, man also schneller lernt. Im Alter dagegen nur noch Feinabstimmungen, in bereits erfasster stabiler Umgebung, durch eine geringe Lernrate erfolgen. (vgl. [Spi00])
Dieser Effekt läßt sich auch bei Kohonen-Netzen nutzen, in dem die Lernrate η im Laufe des Lernens reduziert wird. Dabei sind lineare, nichtlineare Funktionen oder auch Intervalle möglich. In der Regel wird eine monoton fallende Funktion mit $0 < \eta < 1$ angenommen. (vgl. [Spi00])
Die, neben dem Gewinnerneuron c, mit erregten, also benachbarten, Neuronen lassen sich durch Distanzfunktionen berechnen. Die Distanzfunktionen

werden in der Fachliteratur auch Nachbarschaftsfunktionen („neighborhood kernel“) genannt. Sie geben den Grad der Nachbarschaft h des Neurons j in Bezug zum Gewinnerneuron c an und fließen, wie auch die Lernrate η, mit in die Berechnung der Gewichtanpassung ein. (vgl. [Zel03])
Durch folgende Formel läßt sich der Nachbarschaftsgrad h_{cj} zwischen dem Erregungszentrum und dem Neuron j beschreiben. r_c und r_j stellen dabei die Positionen der Neuronen im Neuronengitter dar. z ist der Abstand zwischen dem Erregungszentrum (Gewinnerneuron c) und dem Neuron j. Mit t wird der Zeitpunkt dargestellt. Es gilt dabei $t \to 0$. Statt t wird auch häufig ein Distanzparameter d, welcher die zu betrachtende Umgebungsgröße symbolisiert, angegeben. Dabei gilt $d \to 0$ für $t \to \infty$. Den Distanzparameter d nennt man auch Steifheitsparameter. Falls d anfangs zu klein ist oder sinkt d im Laufe des Lernens zu schnell, so kann dies dazu führen, dass sich entfernte Neuronen nicht mehr gegenseitig beeinflussen können. Dies führt dazu, dass nur noch Teilbereiche des Netzes geordnet werden, die globale Ordnung findet nicht statt. Erkennbar sind solche Karten an ihren deutlichen topologischen Defekten, was bedeutet, dass die Karte nicht korrekt entfaltet wird und im Laufe des Trainings auch nicht berichtigt werden kann. Eine trainierte Karte ohne topologischen Defekt erkennt man daran, dass es keine Überschneidungen der Neuronenverbindungen gibt und sich die Karte im Eingaberaum korrekt entfaltet hat. (vgl. [Zel03],[CJ01])

$$h_{cj}(t) = h(\|r_c - r_j\|, t) = h(z, t) = h'(z, d) \tag{2.18}$$

Einige Möglichkeiten für Dinstanzfunktion sind:

Name	Formel
normalisierte Gaußfunktion	$$h_{cj}(t) = \frac{1}{\sigma(t)\sqrt{2\pi}} \cdot \exp\left(-\frac{\|r_c - r_j\|^2}{2\sigma^2(t)}\right) \quad (2.19)$$
vereinfachte Gaussfunktion	$$h_{cj}(z,d) = e^{-(\frac{z}{d})^2} \quad (2.20)$$
Zylinder (Blase)	$$h_{cj}(z,d) = \begin{cases} 1 & \text{falls } z < 0 \\ 0 & \text{sonst} \end{cases} \quad (2.21)$$
Kegel	$$h_{cj}(z,d) = \begin{cases} 1 - \frac{z}{d} & \text{falls } z < 0 \\ 0 & \text{sonst} \end{cases} \quad (2.22)$$
Cos	$$h_{cj}(z,d) = \begin{cases} \cos(\frac{z}{d}\frac{\pi}{2}) & \text{falls } z < 0 \\ 0 & \text{sonst} \end{cases} \quad (2.23)$$
Nachbar4	$$h_{cj}((r_c,r_j),d) = \begin{cases} 1 & \text{falls } c = j \\ \frac{1}{10} & \text{falls } r_j = N_4(r_c) \\ 0 & \text{sonst} \end{cases} \quad (2.24)$$
Nachbar8	$$h_{cj}((r_c,r_j),d) = \begin{cases} 1 & \text{falls } c = j \\ \frac{1}{10} & \text{falls } r_j = N_8(r_c) \\ 0 & \text{sonst} \end{cases} \quad (2.25)$$

Die Funktionen „Zylinder“ und „normalisierte Gauss‘s“ werden von Teuvo Kohonen empfohlen. Bei der „normalisierten Gauss‘s Funktion“ wird durch die Varianz σ die Ausdehnung des Nachbarschaftskerns definiert.
Wird statt dem Zeitparameter t der Distanzparameter d verwendet und gilt

$z = ||r_c - r_j||$, eignen sich auch die „vereinfachte Gauss‘s Funktion“, Zylinder, Kegel und Cos. Der Wertebereich dieser Funktionen liegt zwischen $[0, 1]$.
Funktionen wie die „mexikan hat“, „Gauss‘s2 Funktion“ oder ähnliche, bei denen negative Funktionwerte auftauchen, sind für nicht normierte Karten ungünstig. Entgegen der ursprünglichen Annahme, dass die Neuronen in den negativen Bereichen abgestoßen werden und sich die Karte dadurch schneller entfaltet, hat sich nach [Zel03] in praktischen Versuchen gezeigt, dass statt dessen eine Explosion der Karte eintreten kann, sämtliche Neuronen nur noch abgestoßen werden und die Karte nicht divergiert.
Die letzten beiden Funktionen, Nachbar4 und Nachbar8, sind Beispiele für Funktionen, bei denen die Nachbarschaft über die Menge der direkten Nachbarn des Neurons c im Neuronengitternetz definiert wird. Nachbar4 beschreibt eine Funktion, bei welcher das Neuron c zu 100% erregt wird. Seine direkten Nachbarn (oben, unten, links, rechts) werden zu 10% erregt und die restlichen Neuronen überhaupt nicht. Die Funktion Nachbar8 macht das Gleiche, jedoch werden auch die diagonalen Nachbarn mit erregt.

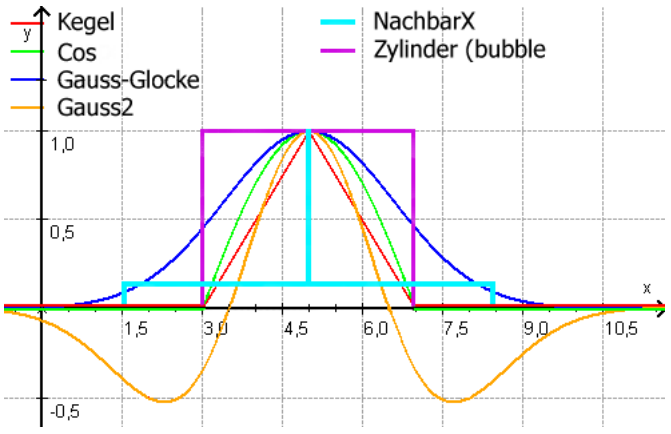

Abbildung 2.16: Grafische Darstellung einiger Distanzfunktionen (erstellt in FunkyPlot)

Die Distanz h_{cj} und die Lernrate η werden in der Formel für die Gewichtsanpassung eingesetzt.

$$W_j(t+1) = W_j(t) + \eta \cdot h_{cj}[X(t) - W_j(t)] \tag{2.26}$$

Für normierte Vektoren findet dagegen folgende Formel Anwendung.

$$W_j(t+1) = \begin{cases} \frac{W_j(t+1)=W_j(t)+\eta \cdot h_{cj}[X(t)-W_j(t)]}{\|W_j(t+1)=W_j(t)+\eta \cdot h_{cj}[X(t)-W_j(t)]\|} & \text{falls } j \in N_c(t) \\ W_j(t) & \text{sonst} \end{cases} \tag{2.27}$$

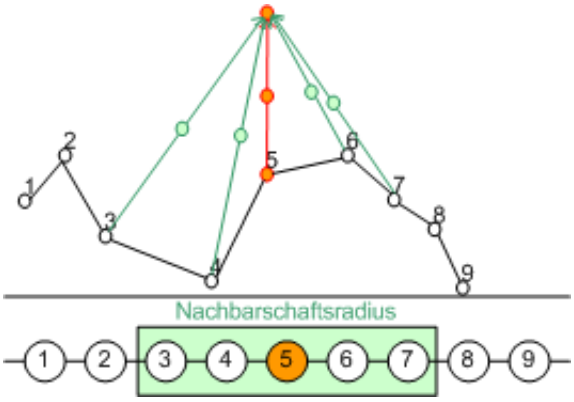

Abbildung 2.17: Bewegungsskizze von Gewichtsvektoren während des Lernens(nach [Zel03] erstellt)

Wie man in der Bewegungsskizze der Gewichtsvektoren sehen kann, werden die nicht erregten Neuronen (1, 2, 8, 9) nicht beeinflußt und damit auch nicht geändert. Die Gewichtsvektoren der Neuronen, welche sich im Nachbarschaftsradius befinden (3, 4, 6, 7), werden mit angepasst, jedoch nicht so stark wie der Gewichtsvektor des Gewinnerneurons (5). Dieses erfährt die stärkste Gewichtsänderung.

Bei „gut“ trainierten SOM kann man davon ausgehen, dass nach Trainingsende die Neuronen mit einem Gewichtsvektor, welcher dem des Eingabevektors ähnelt, auch in der Karte nebeneinander liegen. (vgl. [unb06c])

Ein Sonderfall tritt beim „winner-takes-all“-Prinzip ein. Dabei erfährt nur dass Siegerneuron eine Gewichtsanpassung, da sich die Nachbarschaft nur auf das Siegerneuron selbst beschränkt. In diesem Fall ist es natürlich nicht möglich, die Topologie in den Daten zu erhalten!

Trainingsabbruch/Qualitätsmessung

Der Trainingsabbruch einer SOM kann nach 3 Kriterien erfolgen.

1. Anzahl der Lernschritte oder Epochen
   Wird das Trainingsende auf diese Art eingeleitet, wurde eine bestimmte Anzahl an Lernschritten, bzw. Epochen, erreicht oder die Lernrate $\eta = 0$.

2. Quantisierungsfehler
   Der Quantisierungsfehler wird auch häufig als ein Maß für die Qualität einer SOM verwendet. Der Quantisierungsfehler gibt die durchschnittliche Distanz zwischen einem Satz Eingabemuster und ihren Gewinnerneuronen an. Mit ihm lässt sich jedoch keine Aussage darüber treffen, wie gut Ähnlichkeitsbeziehungen des Eingaberaums durch die Karte dargestellt werden! Den Quantisierungsfehler berechnet man wie folgt:

   $$\epsilon_{quant} = \frac{1}{n} \sum_{i=1}^{n} \|X_i - W_c\| \tag{2.28}$$

   X_i stellt dabei ein Muster mit dem Index i dar. X_n bezeichnet die Anzahl der dargebotenen Muster und W_c das Gewinnerneuron für das jeweilige Eingabemuster dar. (vgl. [Koh97])

3. Siegerabstand
   Bei dieser Abbruchart ermittelt man die Distanz zwischen Siegerneuron und dem Neuron, welches dem Eingabevektor am „zweitähnlichsten“ ist, also das „Zweit-Sieger-Neuron“. Man erhält den Siegerabstand durch folgende Formel:

   $$\epsilon_{winnerdist} = \frac{1}{X_n} \sum_{i=1}^{X_n} \|h_{\text{gold}}(i) - h_{\text{silber}}(i)\| \tag{2.29}$$

   h_{gold} bezeichnet dabei den Koordinatenvektor des ersten Siegerneurons und h_{silber} den Koordinatenvektor des „zweiten“ Siegers für das Muster i. Diese Art der Abbruchbestimmung stammt aus [unb06c]. Leider sagt der Autor der Quelle nichts über den zu erreichenden Wert aus. Nach Meinung des Autors, ist der Wert des „Siegerabstandes“ abhängig von den zugrunde liegenden Daten, sowie von der Netzdimension, so dass hier kein allgemeingültiger Wert erwartbar sein sollte.

2.3.4 Aktivierung

Um die Netzeingabe net_j für ein Kartenneuron j zu berechnen, wird die Grundformel der Netzeingabe für ein Neuron um die Rückkopplung der anderen Neuronen der Kartenschicht erweitert. Somit berechnet sich die Netzeingabe net_j durch folgende Formel.

$$net_j = net_e + net_o - \Theta = \sum_{i=0}^{n} e_i \cdot w_{ij} + \sum_{k=0}^{n} o_k \cdot w_{kj} - \Theta \tag{2.30}$$

Der erste Teil der Formel ist schon vom Neuron-Modell aus dem KNN-Kapitel bekannt. Θ stellt wieder den BIAS dar. net wird um den BIAS Θ verringert, um gegen 0 testen zu können.
Die Netzeingabe innerhalb der Kartenschicht berechnet sich genauso wie die Netzeingabe für die Eingangsneuronen. Nach Summierung der beide Netzeingaben net_e und net_o, wird der BIAS abgezogen.

Die Aktivierung act_j des Neurons j erfolgt über folgende logistische Funktion.

$$act_j = \frac{1}{1 + e^{-net_j}} \tag{2.31}$$

Da die Ausgabe out_j des Neurons j mittels der Identität gebildet wird, gilt $out_j = act_j$. (vgl. [CJ01]) Die Ausgabe zeigt die Erregung eines Neurons in der Kartenschicht für ein angelegtes Muster. Die dabei auftretenden Wechselwirkungen zwischen den Kartenneuronen finden somit nur bei der Interpretation von angelegten Mustern der trainierten Karte Beachtung, jedoch nicht beim Training der SOM. Der Lernprozess kann dadurch erheblich vereinfacht werden und ist ein Unterschied zwischen biologischem Vorbild und künstlicher Nachbildung.

2.3.5 Kodierung der Eingabedaten

In diesem Abschnitt werden Möglichkeiten beschrieben, um aus der Datenbeschreibung des ARFF-Files eine geeignete Codierung der Daten als Eingabe für ein Künstliches Neuronales Netz zu erhalten.

Die Attribute in ARFF-Files unterscheiden 2 Typen, numerische und nominale Daten. Numerische Daten lassen sich dazu einfach als Zahlwert übernehmen. Eine Normalisierung der Attribute kann von Vorteil sein.
Leider können nicht alle Daten stets wertgemäß wiedergegeben. Dies trifft zB. bei nominalen Werten wie auch Wertebereichen (z.B. Temperatur: $x < 10$, $10 < x < 40$, $40 < x$) zu.
Um ordinale Attribute abzubilden gibt es mehrere Möglichkeiten. Eine Möglichkeit ist, die Attribute auf positive ganze Zahlen abzubilden.
Beispiel:

$$\{x < 10, 10 < x < 40, 40 < x\} = \{kalt, normal, warm\} = \{0, 1, 2\} \quad (2.32)$$

Diese einfache Art der Codierung kann sich jedoch bei entsprechender Netztopologie als ungünstig erweisen. Speziell dann, wenn durch die Aktivierungsfunktion der Neuronen reelle Werte auftauchen können. Eine Möglichkeit dieses Problem zu umgehen ist, statt ganzzahliger Werte, die Attribute zwischen 0 und 1 differenziert abzubilden.
Beispiel:

$$\{x < 10, 10 < x < 40, 40 < x\} = \{kalt, normal, warm\} = \{0.0, 0.5, 1.0\} \quad (2.33)$$

Bei dieser Art von Codierung kann es vorkommen, dass die Daten nicht klar genug voneinander getrennt abgebildet werden und das Netz keine optimalen Ergebnisse liefert. Höhere Klarheit erhält man, indem die Daten binär abgebildet werden. Dazu wird nicht, wie bisher jedem Attribut, sondern jedem ordinalen Attribut ein binäres Eingabeneuron zugordnet. Beispiel:

$$\{kalt, normal, warm\} = \{(1, 0, 0), (0, 1, 0), (0, 0, 1)\} \quad (2.34)$$

Diese Art der Eingabecodierung hat den Vorteil, dass jedes Merkmal gleich deutlich abgebildet wird. Diesem Vorteil steht jedoch ein großer Nachteil entgegen. Nicht nur bei Auftreten eines neuen Attributes, sondern schon bei Auftreten eines neuen Attributwertes ist das Überarbeiten der Netztopologie notwendig. Beispiel:

$$\{kalt, normal, warm, heiss\} = \{(1, 0, 0, 0), (0, 1, 0, 0), (0, 0, 1, 0), (0, 0, 0, 1)\} \quad (2.35)$$

(vgl. [CJ01]). Nach Meinung des Autors macht für Attribute, wie beispielsweise eine Spalte mit Namen, nur die letzte Möglichkeit wirklich Sinn, da mit dieser Methode Abhängigkeiten zwischen den Distanzen der vergebenen Zahlenwerte vermieden werden und die Distanz zwischen den Werten mit 1 stets gleich bleibt.

2.3.6 Tips für „gute“ Maps

Topologische Defekte

Eine SOM enthält einen topologischen Defekt, wenn sich Teilbereiche der Kartenschicht gut ordnen, die globale Ordnung jedoch gestört ist. Dies lässt sich in der Gitternetzdarstellung sehr gut erkennen. Ein topologischer Defekt

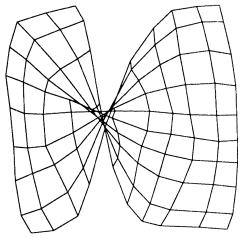

Abbildung 2.18: Gitternetzdarstellung einer SOM mit topologischem Defekt (aus [Zel03])

tritt bei ungünstiger Wahl von Distanzfunktion oder Lernrate auf. Nimmt die Lernrate zu früh ab, fixieren sich die Verbindungsgewichte zu früh. Das Gleiche tritt auf, wenn das Training mit zu kleinen Werten für Nachbarschaftsausdehnung (Varianz) oder der Lernrate erfolgt.

Randeffekte

Unter Randeffekten versteht man Bereiche am Rand der trainierten Kartenschicht, an denen Neuronen dichter zusammen liegen. Die Neuronen am Rand der Karte besitzen weniger Nachbarn. Dadurch wirkt sich die Nachbarschaftsfunktion für diese Neuronen geringer aus und sie werden nicht mehr

soweit nach außen gezogen. (vgl. [unb06c]) Eine mögliche Lösung besteht darin, eine toroide Netzstruktur zu wählen.

Kartenexplosion

Eine Kartenexplosion entsteht, wenn sich sämtliche Neuronen gegenseitig abstoßen und keine zusammenhängenden Neuronengebiete gebildet werden. Das Training konvergiert nicht, es divergiert. Dieser Zustand kann eintreten, wenn Distanzfunktionen mit negativem Wertebereich benutzt werden, wie bespielsweise der „mexikan hat“- oder „Gauss's2“-Funktion. (vgl. [Zel03])

Netzform

Um eine Rotation der Kartenneuronen zu verhindern, sollten im allgemeinen runde Kartenformen wie Kreise nicht benutzt, werden. Es hat sich auch gezeigt, dass eine SOM rechteckigen Kartenform gegenüber einer SOM mit quadratischer Kartenform, besser konvergiert. (vgl. [Lip06])

Netzstruktur

Nach [unb06c] wird empfohlen, hexagonale Gitterstrukturen zu bevorzugen, da sie sich besser visualisieren lassen. Quadratische Strukturen lassen sich jedoch besser implementieren.

seltene Fälle

Im praktischen Einsatz von SOM kommt es oft vor, dass wichtige Muster keine große Beachtung im Training finden und damit im Ergebnis untergehen. Aus diesem Grunde sollten solche Muster im Training künstlich verstärkt, also öfter dargeboten werden, als sie im praktischen Einsatz vorkommen.

2.4 Anwendungen von SOM und verwandte Arbeiten

Neuronale Netze sollen mit ihrer Umwelt interagieren. Dies geschieht auf 2 Weisen.

1. Beziehen von Informationen (Eingabe) z.B. von Sensoren, dem verteilten Speichern von Wissen sowie der Ausfilterung von verrauschten oder überflüssigen Sensorsignalen.

2. Dem Steuern von Effektoren, wobei häufig gegenläufige voneinander abhängige und überflüssige Signale auftreten. (vgl. [Koh97])

Selbstorganisierende Karten dienen dabei in erster Linie der Visualisierung von nichtliniearen Verbindungen mehrdimensionalen Daten. Es haben sich 3 große Anwendungsbereiche herausgebildet.

1. Steuerung und Automatisierung in der Industrie (z.B. zur Überwachung und Kontrolle)

2. Medizinische Anwendungen (Diagnose, Protesen, Erstellen von Patientenprofilen)

3. Anwendungen für die Telekommunikation (z.B. Zuweisung von Netzressourcen, Anpassungsfähige Demodulation, Übertragungsentzerrung)

(vgl. [Koh97])
Künstliche Neuronale Netze allein sind für die Lösung von realen Problemen in der Industrie oft ungeeignet, da die Sensordaten erst durch Signalvorverarbeitung zu geeigneten Eingabevektoren umgewandelt werden müssen. Der Anteil dieser Vorverarbeitung kann dabei viel größer ausfallen, als die eigentliche Problemlösung durch das Neuronale Netz. (vgl. [Koh97])

Ähnlichkeitsanalyse biologisch aktiver Moleküle

Ziel des Projektes ist die Entwicklung von Verfahren, um Abbildungen für eine große Zahl an Molekülen einer Ähnlichkeitsanalyse zu unterwerfen, wobei

zielgerichtet nach ähnlichen, aber noch nicht synthetisierten Verbindungen gesucht werden.
Eine Forschergruppe, welche sich in Kooperation mit Partnern des Computer-Chemie-Centrums der Universität Erlangen/Nürnberg und der pharmazeutischen Industrie befindet, beschäftigt sich mit Methoden, mit denen Ähnlichkeiten von biologisch aktiven Molekülen gefunden werden sollen.
Dies betrifft neben Neurotransmittern auch viele andere Stoffe, welche für die Entwicklung neuer Medikamente in Frage kommen. Biologisch aktive Moleküle binden sich an spezielle Rezeptoren auf der Zelloberfläche. Durch sie werden Reaktionen wie Ioneneinstrom, Proteinaktivierung, Energiefreisetzung sowie Abbau oder Synthese von Stoffen an der Zellmembran oder im Zellinneren in Gang gesetzt. Für diesen Mechanismus ist die räumliche Struktur und die Ladungsverteilung auf der Molekülоberfläche von entscheidender Bedeutung.
Für die Pharmazeutische Industrie ist es von großem Interesse, für bekannte Moleküle, Moleküle mit ähnlichen oder möglichst besseren Eigenschaften zu finden. Bisher existierten dafür große Datenbanken, welche aber nur die chemische Formel o. die 2-dimensionale Struktur der Moleküle enthalten. Aus diesen Informationen ist es möglich, ein statisches 3-dimensionales Bild der Molekülstruktur zu erzeugen. Zusätzlich zur Struktur sind auch Eigenschaften wie elektrisches Potential, Volumen, Fettlöslichkeit (Lipophilie).
Durch X. Li und J.Gassteiger von der TU München wurde ein Kohonen-Netz entwickelt, um eine Abbildung der Molekülоberfläche auf ein rechteckiges Neuronengitter zu erstellen. Diese Abbildungen wiesen unerwünschte topologische Defekte auf, weshalb die obige Forschergruppe das Kohonen-Netz zu einer „Self Organizing Surfaces“ (SOS„selbstorganisierende Oberfläche“) weiterentwickelt hat. Durch die SOS wurde es möglich, nichtzylindrische Moleküle auf eine Kugel besser abbilden zu können.
Das Projekt wird durch das Bundesministerium für Forschung und Technologie gefördert.
(vgl. [Zel03])

WEBSOM

Ziel dieses Projektes ist das Vergleichen und Sortieren von Textdokumenten bezüglich ihres Inhaltes, also der semantischen Bedeutung. Dies ist keine leichte Aufgabe, da es bei Sprachen oft Mehrdeutigkeiten gibt. Ein Beispiel dafür ist das deutsche Wort „Leiter“:

1. DIE *Leiter* IST LANG.
2. DER *Leiter* WAR UNZUFRIEDEN.

Wie man sieht, kommt es darauf an, den Textkontext bei der Analyse des Wortsinnes mit einzubeziehen. Bisher wurden Algorithmen verwendet, welche die Textdokumente auf bestimmte semantische Schlüsselwörter (Wortklassen) scannten. Es stellte sich heraus, dass es zu ungenau ist, nur auf Finden oder Fehlen von diesen Wörtern zu achten und auf den Inhalt zu schließen.
Um dieses Problem wurde WEBSOM entwickelt. Es funktioniert, in dem 2 Abstraktionen vorgenommen werden.

1. Die Wörter des Textdokumentes werden abhängig von ihrer Umgebung, also andere Wörter in ihrem Umfeld, analysiert. Daraus wird eine 2-dimensionale Abbildung, eine so genannte „word category map“, erstellt.
2. Die Positionen in dieser semantischen Karte ergeben dann für die Wörter eine Folge von Positionssequenzen (Histogramme) und damit eine semantische Abstraktion des Dokumenttextes. Ähnliche Dokumente ergeben damit ähnliche Sequenzen und können in der Dokumentenmap geordnet nach ihrem Inhalt topologisch sortiert visualisiert werden.

(vgl. [unb06g])

Music Miner

An der Philipps-Universität Marburg wurde ein, in Java geschriebenes und der GPL unterstehendes, Programm entwickelt, welches Sammlungen von Musikstücken analysiert und selbstständig, nach ihren Klangeigenschaften

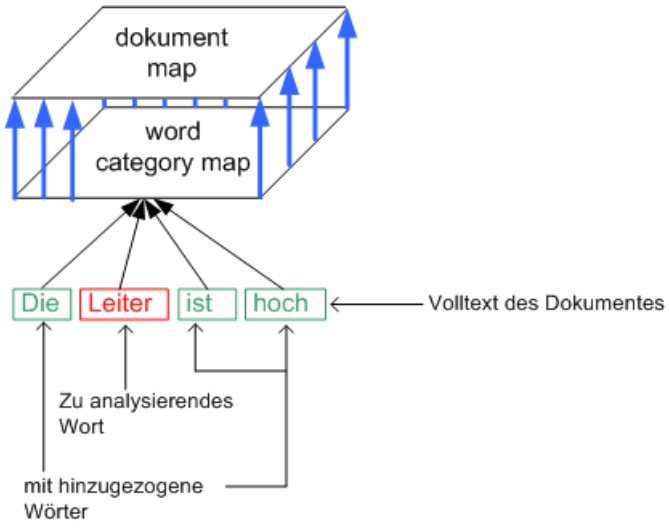

Abbildung 2.19: Verarbeitungsstruktur mit WEBSOM

geordnet, in einer „SoundMap“ darstellt.
Für die Analyse der Musikstücke werden viele selbständige Agenten eingesetzt. Sie untersuchen die Musikstücke auf Klangähnlichkeiten. Dazu werden sämtliche, für das menschliche Gehör optimierte, Merkmale aus den Musikstücken extrahiert. Anschließend werden die Klangmuster auf mit einer SOM organisiert.
Die von der SOM ausgegebene Karte ähnelt einer Landkarte mit Bergen und Tälern und lädt zum interaktiven Navigieren in den Musikstücken ein. Die Region eines Liedes entspricht dabei seinem klanglichen Kontext. Das bedeutet, Täler stellen ähnlich klingende Lieder dar, Berge entsprechen Trennungen zwischen sehr verschiedenen Liedern.
Mit Hilfe dieses Programmes ist es möglich, schnell einen Überblick über große Musiksammlungen zu erhalten. Außerdem können Abspiellisten erzeugt werden, welche Wegen durch die „SoundMap“ entsprechen. (vgl. [AU06])
Der „MusicMiner“ ist unter „https://sourceforge.net/projects/musicminer/“ frei downloadbar.

Abbildung 2.20: Kartenansicht beim MusicMiner (aus [AU06])

Kapitel 3

Visualisierung von SOM

Ziel der Visualisierung ist die Komplexitätsreduktion. Bei einer SOM wird dieses Ziel durch Abbildung von Daten aus einem n-dimensionalen Datenraum, in einen 2-dimensionalen Raum erreicht. Dadurch wird es möglich, komplexe Zusammenhänge in Wissensstrukturen schnell zu erkennen und sich damit den Informationsgehalt unmittelbar verständlich zu machen. Die Visualisierung macht ein interaktives Suchen in den Daten möglich. Sie wird durch Methoden moderner Computergrafik, wie z.B. colorierte Bilder, Animationen oder 3-dimensionale Darstellung verwirklicht. Computerbasierte Visualisierung wird in der Regel so verstanden, dass durch sie eine interaktive grafische Umsetzung von Daten erfolgt. Visualisierung kann entscheidend zur wissenschaftlichen Entdeckung beitragen.
Die Darstellung der reinen Daten, z.B. in Zahlen, führt oft zu Platzproblemen auf dem Ausgabemedium. Je mehr Platz die Daten einnehmen, desto schneller tritt das Platzproblem auf. Durch die Umformung der Daten in eine grafische Darstellung wird es möglich, die Daten und ihre Eigenschaften auf minimalstem Raum zu präsentieren. Die Kodierung kann dabei durch Farbe, Schattierung, Position, Größe erfolgen. Das Platzproblem ist damit nicht aufgehoben, wird aber wesentlich reduziert.
Ein anderer Aspekt ist, dass das menschliche Gehirn nur bei sehr kleinen Datenausschnitten in der Lage ist, eine numerische Ausgabe zu interpretieren. Werden die Datenmengen größer oder sind die Daten hochdimensional,

ist der Mensch nicht mehr in der Lage, sie zu interpretieren. Für den Menschen sind nur bis zu 3 räumliche Dimensionen darstellbar. Nicht räumliche Dimensionen, wie Farbe, Schattierung etc., sind durch die Leistungsfähigkeit des menschlichen Gehirns beschränkt und nur theoretisch unendlich.(vgl. [ME01])
Um Daten mittels Computer zu visualisieren, orientiert man sich meistens an der „Visualisierungspipeline“. Dieser Begriff fasst die Arbeitsabläufe zusammen, welche bei der Visualisierung in zeitlicher Abfolge auftreten. Dabei werden die Mess- oder Simulationsdaten, welche vom Rechner, Maschinen, Datenbanken oder anderen Datenquellen kommen, aufbereitet und gefiltert. Dies kann durch starkes ausdünnen, ergänzen oder projezieren der Daten erfolgen. Danach werden die Daten gemappt. Das bedeutet, dass eine Abbildung der Visualisierungsdaten auf darstellbare Repräsentationen, wie Linien, Flächen, Farben, Texturen o.ä., erfolgt. Im anschließenden Rendering werden aus diesen Repräsentationen die sichtbaren Objekte generiert. (vgl. [ML01])
In der Visualisierung haben sich zwei große Bereiche herausgebildet.

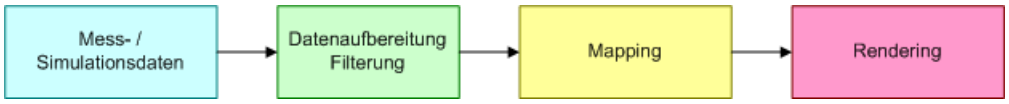

Abbildung 3.1: Visualisierungs-Pipeline

1. „scientific visualisation“,
   bei ihr geht es um die Visualisation physischer Daten.

2. „information visualisation“,
   bei ihr handelt es sich um abstrakte Daten.

Oft kommt es vor, dass beide Arten vermischt sind und ineinander übergehen. (vgl. [ME01])
Hochdimensionale Daten grafisch so umzusetzen, dass keine Verluste auftreten ist äußerst schwierig und oft sogar unmöglich.
Die Visualisierung mittels SOM hat den Nachteil, dass nicht rekonstruiert

werden kann, wie das Ergebnis entstanden ist. Nach [ME01] kann bei SOM keine Abschätzung der Komprimierungsqualität der Daten aus einem n-dimensionalen Raum in einen m-dimensionalen Raum vorgenommen werden. Das Erstellen der topologischen Eigenschaftsabbildungen erfolgt bei SOM nach 2 Prinzipien.

1. Ähnlichkeit
   Eingabemuster, welche sich ähnlich sind, werden auf der Karte auch näher beieinander repräsentiert. Unähnlichere Muster werden weiter voneinander entfernt abgebildet. Dies entspricht auch dem biologischen Vorbild.

2. Häufigkeit
   Eingabemuster, welche dem Netz öfter „gezeigt“ werden, werden durch eine größere Fläche symbolisiert als seltenere Muster. Dieser Aspekt ist auch wichtig bei der Auswahl der Eingabemuster beim Lernen. Auch dieses Prinzip lässt sich im biologischen Vorbild wiederfinden.

In Anlehnung an an die Prinzipien werden Kohonen-Netze auch als „Selbstorganisierende Eigenschafts- oder Merkmalskarten“ bezeichnet. Die treibende Kraft für die Selbstorganisation ist die bereits in den Eingabemustern enthaltene Ordnung. Kohonen-Netze extrahieren sämtliche Regelmäßigkeiten in den Eingabemustern. Sie decken sie auf und präsentieren sie geordnet. Dabei wird sich stets auf die räumlich-zeitliche Struktur der Eingabedaten bezogen.(vgl. [Spi00])
Da bei SOM unüberwachtes Lernen stattfindet, ergeben sich die gefundenen Klassen nur aus dem Lernalgorithmus. Eine Vorgabe von „Außen“ findet nicht statt!

Im folgenden werden gängige Visualisierungsmöglichkeiten von Kohonen-Netzen erläutert.
Dabei wird auf musterabhängige und musterunabhängige Varianten eingegangen. Musterabhängig bedeutet, dass sich die Darstellung auf ein spezielles angelegtes Muster (Datensatz) bezieht. Musterunabhängige Varianten

stellen den allgemeinen Netzzustand dar. Zum Schluss des Kapitels erfolgt eine Beschreibung von Voronoi-Diagrammen.

- Musterabhängige Visualisierungen
  - Distanzmatrix
  - Gewinnermatrix
- Musterunabhängige Visualisierungen
  - Gewichtsmatrix
  - Komponentenmatrix
  - U-Matrix
  - Intervertierte U-Matrix
  - P-Matrix
  - U*-Matrix
  - Histogramm über die relative Gewinnerverteilung
  - Verbindungs-Distanz-Matrix

Im speziellen Fall eines eindimensionalen Eingabevektors, lässt sich die Netzstruktur durch die Ausgabe des Verbindungsgewichtes zwischen dem Neuron der Kartenschicht und dem einzigen Eingabeneuron darstellen. Der Gewichtswert wird dabei in der Regel gerundet ausgegeben. Der Zahlenwert symbolisiert anschließend die Klasse, welcher das Kartenneuron zugeordnet wird. (vgl. [Kin92]) Grundsätzlich lassen sich die Visualisierungen als eine Matrix aus Zahlen darstellen, was jedoch schnell zur Unübersichtlichkeit und Uneindeutigkeit führen kann.
Um zu erkennen, wie gut der Eingaberaum durch ein 1D oder 2D Neuronennetz abgebildet wird, ist eine Darstellung der Neuronenlage als Gitter- oder Neuronenkette nützlich. Bei der Darstellung von Kohonennetzen durch ein Gitter werden Neuronen, welche im Netz nebeneinander liegen, durch eine Linie verbunden. Die aktuelle Lage der Neuronen wird dann über ein 2- oder 3-dimensionales Koordinatensystem angezeigt. Liegt dem Netz ein

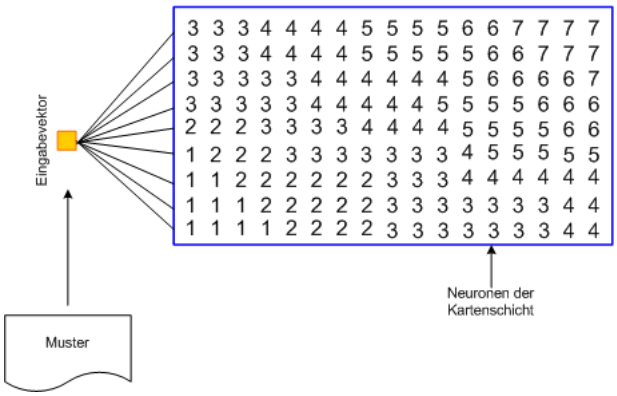

Abbildung 3.2: Darstellung des Gewichts zum Eingabeneuron durch ganze Zahlen bei einem eindimensionalen Eingabevektor

1-dimensionales Neuronennetz zugrunde, so entsteht dabei eine Kurve (Linie), bei 2-dimensionalem Netz ein Gitter. In der 3-dimensionalen Darstellung bilden sich skelettartige Abbildungen des Eingaberaums heraus. Die Kurvendarstellung eignet sich auch zu Lösung des Problems des Handlungsreisenden mittels SOM. Bei dieser Darstellungsart liegen die Neuronen noch wirr im Raum, jedoch lässt sich im Laufe des Lernens die Entfaltung der Karte recht deutlich erkennen.(vgl. [Kin92])

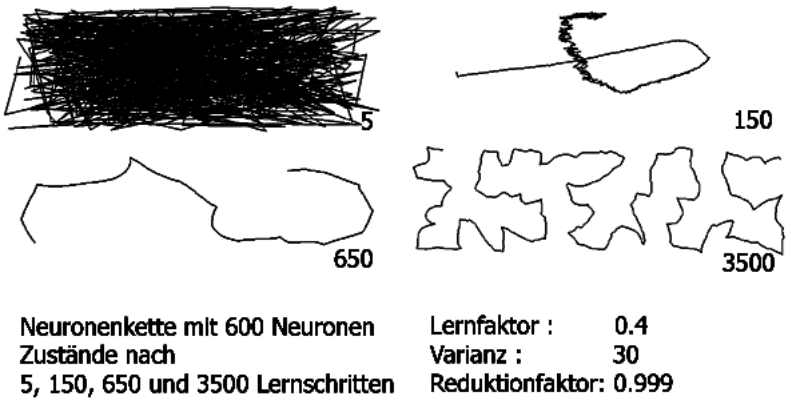

Abbildung 3.3: Beispiel für Kurvenausgabe durch eine 1-dimensionale SOM (Neuronenkette) aus einem Applet des Autors

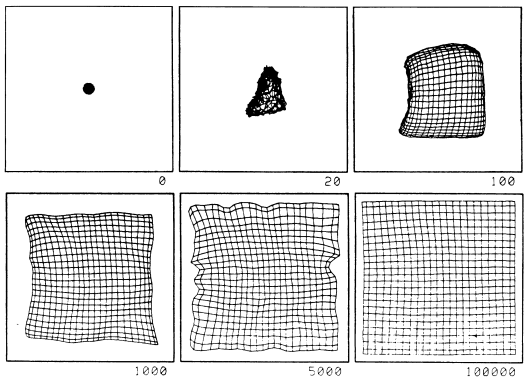

Abbildung 3.4: Beispielausgabe für eine 2-dimensionale SOM aus [Kin92]

Außerdem haben sich verschiedene 2D- und 3D-Darstellungen durchgesetzt und werden im folgenden näher beleuchtet.

Musterabhängige Visualisierungen

Distanzmatrix Die Distanz-Matrix zeigt die Aktivierung sämtlicher Neuronen für ein Muster, also für einen speziellen Eingabevektor, an. Es werden sämtliche Eingabeneuronen in die Darstellung mit einbezogen. Dabei wird mit unterschiedlichen Helligkeits- oder Farbwerten gearbeitet.

Mit dieser Dartstellungsart erfährt man, wie ähnlich der Gewichtsvektor jedes Neurons gegenüber dem Eingabevektor ist. Diese Art der Darstellung ist patternabhängig.

Gewinnermatrix In dieser Ansicht wird das Neuron hervorgehoben, welches durch seinen Gewichtsvektor dem Eingabevektor am ähnlichsten und somit das Gewinnerneuron ist. Außerdem wird der umliegende Erregungsbereich, dessen Gewichte neu angepasst werden, hervorgehoben. Dies wird durch unterschiedliche Farb- und/oder Helligkeitswerte verwirklicht. Das Aussehen der Darstellung ist stark vom Wert der verwendeten Lernparameter im Augenblick der Darstellung und von der benutzten Nachbarschaftsfunktion abhängig.

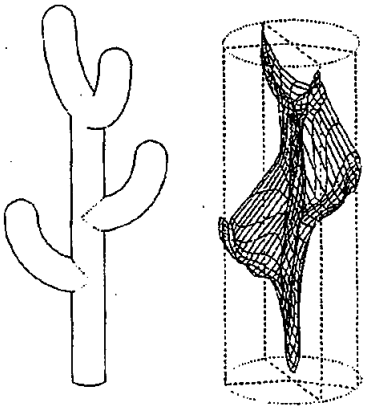

Abbildung 3.5: Beispielausgabe für eine 3-dimensionale SOM aus [Met96]

Musterunabhängige Visualisierungen

Gewichtsmatrix Die Gewichtsmatrix stellt vollständig die Gewichte der Verbindungen zwischen allen Kartenneuronen und den Eingabeneuronen dar. Dazu wird in einem zweidimensionalen Raster eine Achse durch die Eingabeneuronen und die andere Achse durch die Kartenneuronen dargestellt. Auf diese Weise ist es möglich, jede Gewichtsänderung im Laufe des gesamten Lernvorgangs für sämtliche Neuronenverbindungen zu visualisieren. Dieses Werkzeug ist für die Lernphase nützlich, da man so erkennen kann, ob sich nur noch bestimmte Verbindungen anpassen und wie weit das Lernen fortgeschritten ist.

Komponentenmatrix Die Komponenten-Matrix ähnelt der Gewichts-Matrix. Wie sie stellt die Komponentenmatrix die Verbindungsgewichte, jedoch in Bezug auf eine spezielle Dimension des Eingabevektors, dar. Das bedeutet, dass für jedes Eingabeneuron (Dimensionen des Eingabevektors), eine Komponentenkarte existiert.
Mit dieser Darstellung ist es also möglich, die Auswirkung von einzelnen Eingabeneuronen zu visualisieren. Dadurch lassen sich

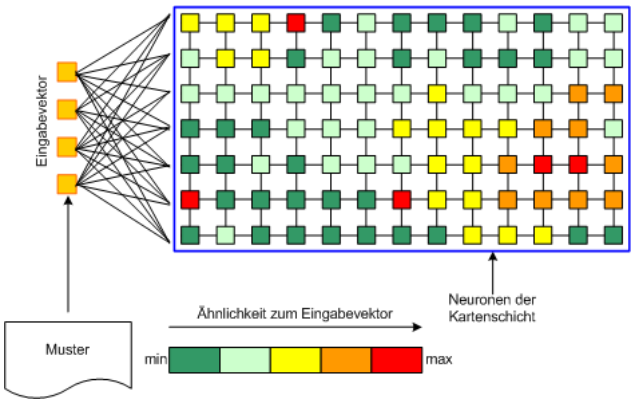

Abbildung 3.6: Distanzmatrix

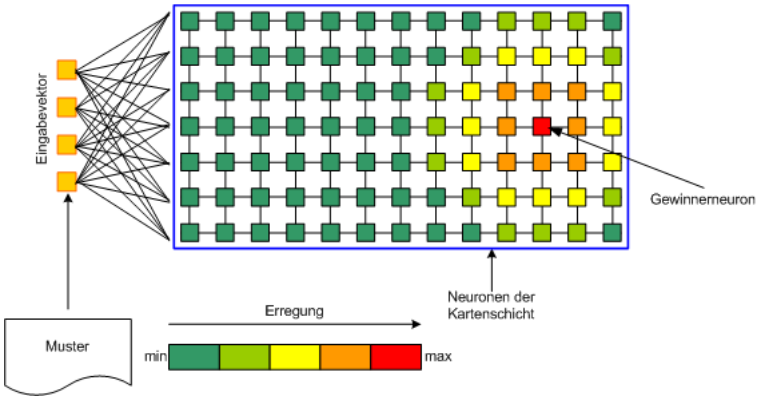

Abbildung 3.7: Gewinnervisualisierung

Eingabeneuronen finden, welche wenig oder keinen Einfluß auf das Ergebnis haben, also überflüssig sind für die Bildung der topologischen Karte und somit Ressourcen kosten. Man erkennt dies daran, dass 2 (oder noch mehr) Neuronen den selben Bereich der Kartenneuronen erregen. Zur Stimulation dieses Neuronenbereiches ist dann 1 Neuron ausreichend. Ein anderer Aspekt ist, dass durch solche Gebietsüberschneidungen positive und negative Korrelationen sichtbar werden. Das bedeutet, man kann damit Abhängigkeiten in den Attributen untereinander finden. Ein gutes Beispiel dafür findet sich in [AU06], wo u.a. Abhängigkeiten zwischen

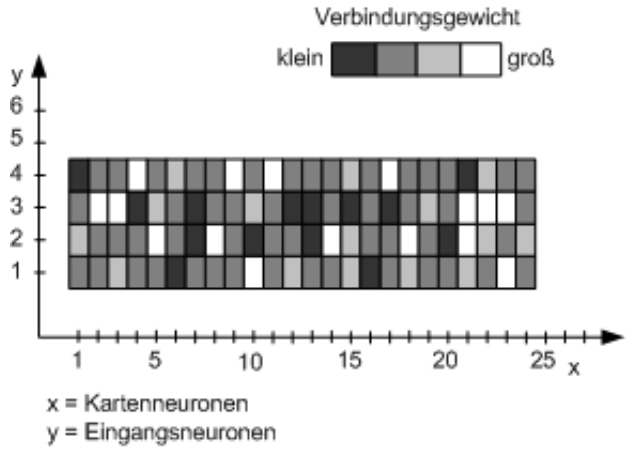

Abbildung 3.8: Gewichtsmatrix für 4 Eingangs- und 24 Kartenneuronen

Löhnen und Preisen (positive Korrelation), sowie Arbeitslosigkeit und offenen Stellen (negative korrelation) gefunden wurden.
Wie die Distanz-Matrix werden für die Visualisierung unterschiedliche Farb- und/oder Helligkeitswerte genutzt.

U-Matrix Wenn Daten nur durch ihre Position im Kartenraum dargestellt werden, kann es schwer sein Strukturen zu erkennen. Dies liegt daran, dass hochdimensionale Distanzen verzerrt dargestellt werden. Neuronen, welche auf dem Neuronengitter benachbart liegen, können im Datenraum weiter voneinander entfernt sein. Ein Lösungsansatz für dieses Problem findet sich in der U-Matrix. Sie stellt die lokalen Distanzbeziehungen im hochdimensionalen Datenraum in einer 2- oder 3-dimensionalen Karte dar. Die U-Matrix ähnelt dabei einer Landkarte mit Bergen und Tälern. Große Distanzen zwischen den Mustern werden in einer 3-dimensionalen Darstellung als Berge, kurze Distanzen als Täler dargestellt. Im der 2-dimensionalen Raum erfolgt die Darstellung mit Hilfe von Helligkeitswerten und Farben. Daten, welche in einem gemeinsamen Tal liegen, sind einander ähnlich und gehören zusammen. Diese Art der Visualisierung wird in dem SOM-Anwendungsbeispiel „Music Miner“ (siehe [AU06]) verwendet.

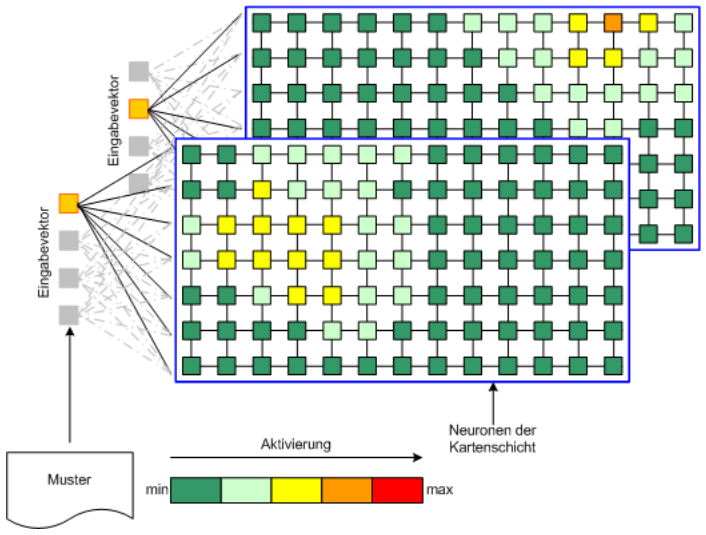

Abbildung 3.9: Komponentenmatrix für das erste und zweite Eingabeneuron

Intervertierte U-Matrix Die intervertierte U-Matrixdarstellung basiert auf der oben beschriebenen U-Matrix, jedoch wird bei dieser Darstellungsart nicht die komplette Neuronenzelle eingefärbt, sondern es werden Farb- oder Helligkeitsverläufe von den Zellmittelpunkten zu den Zellmittelpunkten ihrer jeweiligen Nachbarn dargestellt. Dadurch lassen sich harte Kanten an den Grenzen zwischen den Neuronen vermeiden und der Betrachter erhält ein weichere Abbildung.

P-Matrix Eine andere Art der Visualiserung findet sich durch die P-Matrix. Sie stellt die Dichte der Daten über den Neuronen dar. Zur Errechnung der Datendichte wird die Pareto-Dichteschätzung verwendet.

U*-Matrix Durch Kombination der U- mit der P-Matrix erhält man die U*-Matrix. Bei der U*-Matrix fällt die Distanz in Kartenbereichen mit hoher Dichte weniger ins Gewicht. Dünn besiedelte Kartenbereiche werden jedoch extra betont.

Histogramm über die relative Gewinnerverteilung Das Histogramm stellt für eine feste Menge an Mustern dar, wie oft welches Neuron

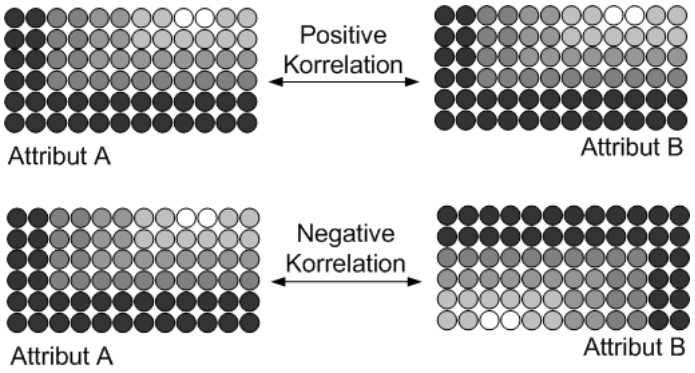

Abbildung 3.10: positive/negative Korrelation

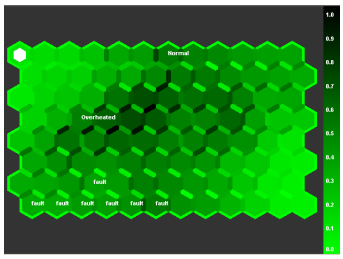

Abbildung 3.11: 2D U-Matrix aus dem Programm Nenet 1.1.

„gewonnen“ hat. Nach Meinung des Autors kann hierbei eine Unterteilung in 2 Arten erfolgen.

1. Es wird nur gezählt, wenn das Neuron dass Gewinnerneuron war.
2. Es wird auch gezählt, wenn das Neuron nur zu einem bestimmten Teil gewonnen hat, also in der Nachbarschaft des Gewinnerneurons lag. Ausschlaggebend für den Grad der Zählung ist der Grad der Nachbarschaft.

Verbindungs-Distanz-Matrix Die Verbindungs-Distanz-Matrix stellt die Neuronen als Punkte in ihrer Position im Gitter dar. Zwischen den Neuronen wird die Distanz zu ihren Nachbarneuronen, ähnlich der U-Matrix, als gefärbte und/oder unterschiedlich dicke Balken dargestellt. Diese Ausgabe eignet sich auch auf Systemen, die nur über begrenz-

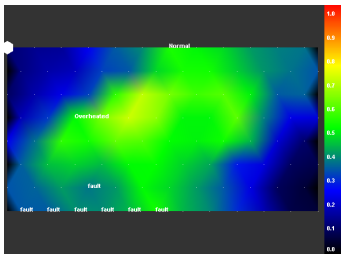

Abbildung 3.12: intervertierte 2D U-Matrix aus dem Programm Nenet 1.1.

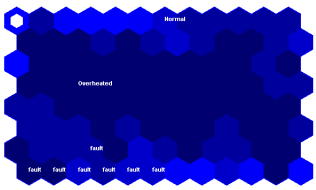

Abbildung 3.13: 2D Histogramm über relative Gewinnerverteilung aus Programm Nenet 1.1.

te Ausgabemöglichkeiten verfügen, z.B. bei monochromen Bildschirmen. Durch Nutzung von Schwellwerten und deren Verschiebung können Neuronen-Cluster sowie Verbindungen zwischen solchen gefunden werden.

Voronoi-Diagramme

Gelegentlich werden zur Darstellung auch Voronoi-Diagramme verwendet. Mit ihnen lassen sich Abstandsbeziehungen zwischen endlich vielen Punkten in der Ebene oder höherdimensionalen Punkten ausdrücken. Durch Voronoidiagramme lassen sich Bildpunkte ihrem nächstenliegenden Neuron zuordnen. Ergebnis ist dann eine lückenlose zellartige Struktur. Das Voronoi-Diagramm in der Abbildung wurde aus 14 Punkten erstellt. Die Punkte stellen dabei die Lage der Neuronen in der Ebene dar. Um das Voronoi-Diagramm zu erhalten, verbindet man benachbarte Punkte und fällt durch die Verbindungsmitte eine Senkrechte. Die Senkrechten werden dann am

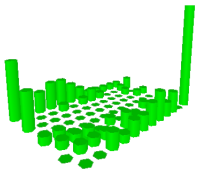

Abbildung 3.14: 3D Histogramm über relative Gewinnerverteilung aus Programm Nenet 1.1.

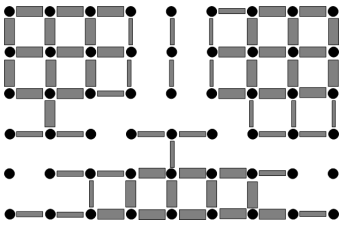

Abbildung 3.15: Verbindungs-Distanz-Matrix

Treffpunkt verbunden. In der Abbildung werden die Punkte durch schwarze Kreise, die Verbindungslinien blau und die miteinander verbundenen Senkrechten rot gestrichelt dargestellt.

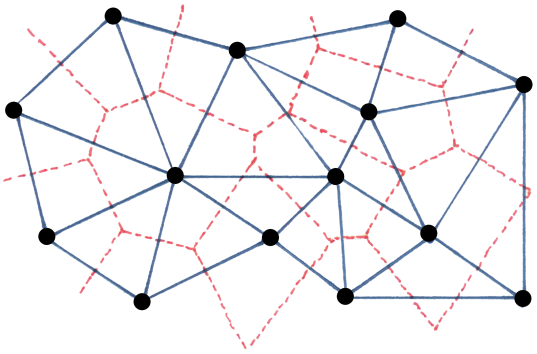

Abbildung 3.16: Voronoi-Diagramm

Kapitel 4

Konzeption

4.1 Pflichtenheft

Sinn und Zweck Das zu erstellende Programm soll in der Lehre helfen, sich in das Thema „Selbstorganisiation/Selbstorganisierende Karten“ einzuarbeiten. Es soll leicht verständlich und intuitiv bedienbar sein. Mit ihm soll es möglich sein, Daten aus *.arff-Dateien unter Verwendung eines Kohohnen-Netzes zu analysieren und klassifiziert auf dem Bildschirm auszugeben. Ein wichtiger Aspekt ist die Möglichkeit mit den verschiedenen Einstellparametern für das Kohonen-Netz und dem Trainingsvorgang experimentieren zu können. Auch liegt der Augenmerk auf der Darstellung verschiedenster Perspektiven auf die Kohonen-Karte um die Ergebnisse aus mehreren Blickwinkeln betrachten zu können. Da aufgrund Zeitmangels nicht alle möglichen Parameter und Perspektiven implementierbar sind, fällt ein dritter wichtiger Aspekt auf die Erweiterbarkeit und Wartbarkeit des Softwaresystems (Softwarearchitektur).

Anforderungen Es ist eine moderne mehrschichtige skalierbare Software zu erstellen, welche ohne Installation auf möglichst vielen Plattformen nutzbar ist. Als Dateneingabeformat muss das *.arff-Format unterstützt werden. Aus der Datenquelle hat das Programm Eingabevektoren zu erstellen und diese für das Training einer Neuronalen Karte (Kohonen-Netz) zu verwenden.

Der Lernvorgang sowie das Lernergebnis muss über geeignete Perspektiven zyklisch darstellbar sein. Musterabhänge und musterunabhängige Perspektiven sind dafür zu realisieren. Das Trainingsergebnis muss über eine gelabelte Karte in einer Bilddatei im *.jpg-Format speicherbar sein. Die Schriftart und die Farben müssen vom Benutzer änderbar sein. Ein jederzeitiges Stoppen des Lernvorgangs, Parameteränderung und wieder Fortfahren muss realisiert werden.

Kann-Kriterien Wünschenswert ist die Möglichkeit des Laden/Speichern von Kohonen-Netzen im *.xml-Format.

Einsatzbereich

Anwendungsbereich Das Programm soll als eigenständige Anwendung auf dem Zielsystem arbeiten. Zielsysteme sind Systeme, welche über eine Java Virtual Maschine (JVM) verfügen. Die Auslieferung hat als Download in Form einer *.jar-Datei über einer Server zu erfolgen.

Benutzergruppen Es werden keine expliziten Benutzergruppen unterschieden. Es gibt nur gleichrangige Anwender.

Technische Umgebung

Software Das Zielsystem muß über eine JVM in der Version 1.4 verfügen. Weiterhin wird eine Ausgabemöglichkeit auf einem Farbbildschirm mit min 256 Farben vorrausgesetzt.

Hardware Ein JAVA-fähiger Clientrechner ist ausreichend. Ausgeschlossen davon sind JAVA-fähige Handys, Handhelds und vergleichbare Geräte.

Funktionalität Es ist ein Dateizugriff für die Auswahl des Daten-Files zu realisieren. Das Kohonen-Netz muss zur Initialisierung über die Oberfläche parametrisierbar sein.

Folgende Parameter sind für die Initialisierung des Neuronalen Netzes zur Verfügung zu stellen:

- Anzahl der Netzneuronen in Länge und Breite
- Initialisierungsart

Folgende Parameter sind für die Anpassung des Netztrainings zu implementieren:

- Anzahl der Lernschritte
- Lernrate, Lernfaktor und Reduktionsart
- Nachbarfunktion, Varianz
- Abruchbedingung

Die Kartenperspektiven haben den Lernfortschritt zyklisch und farbig darzustellen. Der Lernzyklus muß angehalten und wieder gestartet werden können. Beim Pausieren des Lernzyklus müssen die Lernparameter veränderbar sein. Die Datensatzpositionen (Neuronen, welche den Datensatz am besten repräsentieren) haben in der Perspektive markiert und gelabelt dargestellt zu werden. Das Kartenbild soll jederzeit als Bilddatei im Dateisystem des Clientrechners gespeichert werden können. Farben der Karte und der Schriftarten der ausgewiesenen Datensätze müssen über die Benutzeroberfläche verändert werden können.

Daten

Eingabe Die Datenquelle der Trainingsmuster besteht aus Textdateien im *.arff-Format.

Ausgabe Die Ergebnisausgabe erfolgt auf dem Bildschirm und kann in Form einer Bilddatei im JPG-Format gespeichert werden.

Leistungen Es wurden keine Vorgaben bezüglich der Leistungsfähigkeit des Systems in zeitlicher Hinsicht getroffen.

Benutzeroberfläche Die Anwendung soll über eine fensterbasierte Benutzeroberfläche realisiert werden. Für das Festlegen von Parametern für den Parser, die Neuronale Karte und das Netztraining soll die Eingabe über spezielle Terminalfenster realisiert werden. Von den SOM-Perspektiven sollen meherere, auch mehrere Instanzen der gleichen Perspektive, gleichzeitig auf dem Bildschirm anzeigbar sein. Die Umsetzung erfolgt in JAVA mit Swing. Folgende Skizzen geben einen Überblick über die geplante Oberfläche.

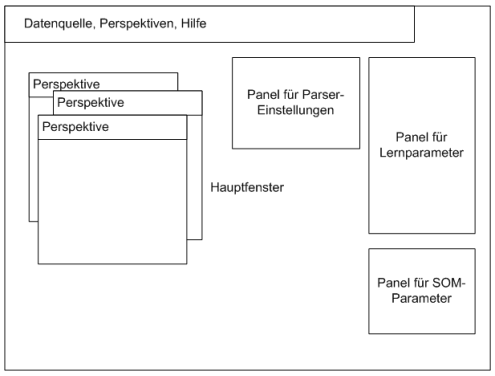

Abbildung 4.1: Skizze Anwendungshauptfenster

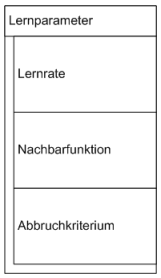

Abbildung 4.2: Skizze Panel Lernparameter

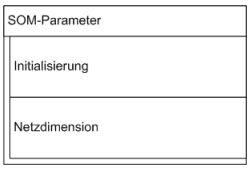

Abbildung 4.3: Skizze Panel SOM-Parameter

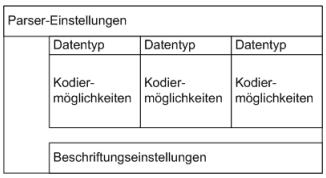

Abbildung 4.4: Skizze Panel Parser-Einstellungen

4.2 Systemarchitektur

Das System wurde in einer 3-Schichten-Architektur angelegt.

Präsentationsschicht In dieser Schicht befinden sich sämtliche, der Benutzeroberfläche zugehörige, Programmteile.

Hauptfenster Es stellt den äußeren Rahmen der Benutzeroberfläche dar. Ihm sind sämtliche Unterfenster und Terminals untergeordnet. Außerdem befindet sich in ihm ein Menü, über das auf die Programmfunktionen zugegriffen werden kann.

Perspektiven Sie stellen unterschiedlichste Betrachtungsarten des Kohonennetzes dar. Im Programm sind die Perspektiven

- Zahlen-Matrix
- Gitter-Matrix
- Distanz-Matrix
- Komponenten-Matrix
- Gewichts-Matrix
- Gewinner-Matrix
- U-Matrix

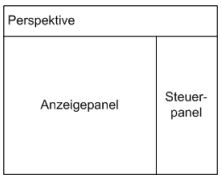

Abbildung 4.5: Skizze Perspektiven-Panels

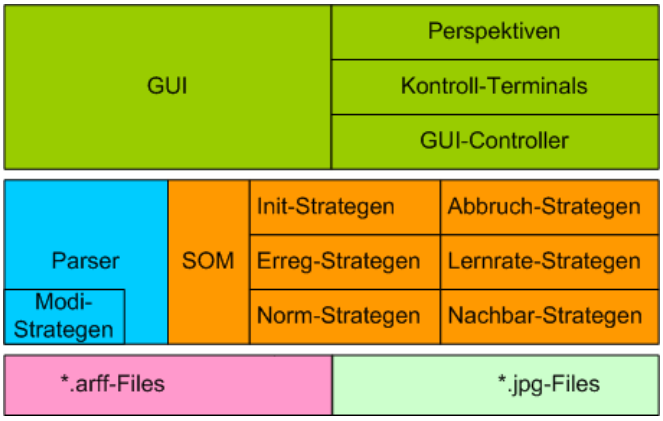

Abbildung 4.6: SOMARFF-Architektur

realisiert.

Kontroll-Terminals Dies sind die Eingabemasken und Panele für die Programmsteuerung, zum Beispiel um den Lernzyklus zu starten/-pausieren/stoppen zu lassen oder die Parameter für das Kohonennetz und das Lernen festzulegen. Auch die Oberflächen für den Parser (Datenblatt, Einstellungen) zählen hierzu.

Controller Er ist für die Verbindung zwischen Präsentationsschicht und Anwendungslogik verantwortlich.

Anwendungslogik Diese Schicht besteht aus 2 Komponenten.

Parser Die Komponente Parser ist für das Erstellen der Eingabemuster aus einer Datenquelle verantwortlich. Er ist als Schnittstelle angelegt. In SOMARFF wurde ein Parser implementiert, der arff-Dateien auswertet. Da die Komponente Parser jedoch als Schnitt-

stelle angelegt ist, ist eine Erweiterung um weitere Parser für andere Datenquellen möglich (z.B. für xml-Dateien, relationale Datenbanken)

SOM Die Komponente SOM kapselt das Kohonennetz und ist für den Lernvorgang verantwortlich. Um dies zu bewerkstelligen, nutzt sie einen Satz von Service-Klassen(Strategen), welche über ihre Schnittstellen erreichbar sind. Durch Implementieren der Schnittstelle in eine Klasse kann das Programm um neue Strategen erweitert werden. Folgende Strategen finden in SOMARFF Verwendung:

Init-Strategen Diese Schnittstelle stellt Klassen zur Verfügung, über die das Kohonen-Netz initialisiert werden kann. In SOMARFF wurden 2 Strategen implementiert. Der erste initialisiert das Netz mit Zufallszahlen im Bereich $[0, wert]$. Wobei *wert* ein vom Benuzer vorgebbarer Parameter ist. Der zweite Stratege initialisiert das Netz einheitlich mit dem vom Benutzer angegebenen Wert.

Erreg-Strategen Diese Komponente ist für das Finden des Siegerneurons zuständig. SOMARFF implementiert einen Erst-Sieger-Ermittler. Das bedeutet, dass jenes Neuron als Sieger gefunden wird, dessen Gewichtsvektor den kleinsten Euklidischen Abstand zum Eingabevektor als erstes Neuron aufweist. Nachfolgend betrachtete Neuronen mit gleichem Abstand wie das Erste, finden keine Betrachtung. Eine Erweiterung um andere Strategen ist über die Schnittstelle jedoch leicht möglich.

Norm-Strategen Die Schnittstelle stellt Klassen mit Algorithmen zur Normierung von numerischen Werten bereit. In SOMARFF wurde ein Standard-Normierer implementiert, welcher numerische Werte im Bereich zwischen $[-1, 1]$ abbildet, wobei negative Werte im Bereich $[-1, 0]$ und positive im Bereich $[0, 1]$ liegen.

Abbruch-Strategen klassen, welche diese Schnittstelle implementie-

ren, stellen Algorithmen zur Überprüfung der Abbruchbedingung zur Verfügung. In SOMARFF wurden 2 konkrete Abbruch-Strategen implementiert. Der Erste überprüft, ob die erforderliche Anzahl an Lernschritten absolviert wurde. Der Zweite berechnet den Quantisierungsfehler und prüft, ob er den vom Benutzer gewünschten Fehler erreicht oder unterschreitet.

Lernrate-Strategen Mit den Klassen dieser Schnittstelle kann die SOM ihre Lernrate anhand der gewünschten mathematischen Funktion reduzieren. In SOMARFF stehen eine lineare und eine exponentielle Lernratenreduktion zur Verfügung. Eine Implementierung weiterer Funktionen ist über die Schnittstelle leicht möglich.

Nachbar-Strategen Die SOM benötigt diese Schnittstelle, um die Nachbarschaft des Siegerneurons und deren Erregungskurve zu berechnen. In SOMARFF sind Zylinder, Gauss, Mexican Hat, Kegel, Cos implementiert. Wie bei den anderen Strategen ist auch hier eine Erweiterung über die Schnittstelle problemlos möglich.

Datenhaltungsschicht In der Datenhaltungsschicht befinden sich die Datenfiles (*.arff), welche bei Parsererweiterung auch auf andere Quellen ausgedehnt werden kann.
Das Speichern der Kartenabbildungen erfolgt in *.jpg-Dateien.

4.3 verwendete Entwurfsmuster

Unter Entwurfsmuster, oder auch „Design Pattern“ genannt, versteht man praktisch erfolgreich erprobte generische Lösungstrategien für häufig auftretende Entwurfsprobleme in Form von Vorlagen zur Problemlösung. Besonders bekannt wurden Entwurfsmuster durch die „Gang of Four“ (Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides). Folgende Entwurfsmuster finden bei SOMARFF Verwendung:

4.3.1 Observer

„Das Observer-Muster definiert eine Eins-zu-viele-Abhängigkeit zwischen Objekten in der Art, dass alle abhängigen Objekte benachrichtigt werden, wenn sich der Zustand des einen Objekts ändert.“[EF06]

Das ObserverMuster zählt zu den Verhaltensmustern und ermöglicht die Weitergabe von Änderungen eines Objektes an von ihm abhängige Objekte. Das Muster besteht aus einem zu beobachtenden Objekt (erweitert Observable) und einer variablen Anzahl von beobachtenden Klassen (implementieren Observer). Die beobachtete Klasse stellt Methoden für das An- und Abmelden der Beobachter zur Verfügung. Sie benötigt zudem keine Kenntnis über den Aufbau der Beobachter-Klassen. Mit der Implementation der Schnittstelle Observer „versprechen“ die Beobachter-Klassen, eine Update-Methode zu beinhalten, welche durch die beobachtete Klasse aufgerufen wird. Bei deren Aufruf behandeln die Beobachter die Änderung für sich.
Aufgrund dieser Eigenschaften eignete sich das Muster zur Aktualisierung der Perspektiven der SOM. Nach Beendigung eines Lernschrittes benachrichtigt die SOM-Klasse ihre Beobachter. Diese werden im Hauptfenster mit Hilfe des Factory-Musters instanziiert. Für die Anmeldung wurde das Muster leicht abgewandelt, indem die Observer im Hauptfenster nach der Erzeugung angemeldet werden.

4.3.2 Abstract Factory

„Das Abstract Factory-Muster bietet eine Schnittstelle zum Erstellen von Familien verwandter oder zusammenhängender Objekte an, ohne konkrete Klassen anzugeben.“[EF06]

Das FactoryMuster ist für Erzeugung von Objekten zur Programmlaufzeit verantwortlich und gehört somit zur Gruppe der Erzeugungsmuster. Bei diesem Muster übernimmt eine Klasse die Rolle einer Fabrik, welche auf Anfrage Objekte einer Klasse erstellt und an den Aufrufer zurückgibt.
Das Muster wurde vom Autor bei der Erstellung der Benutzeroberfläche ein-

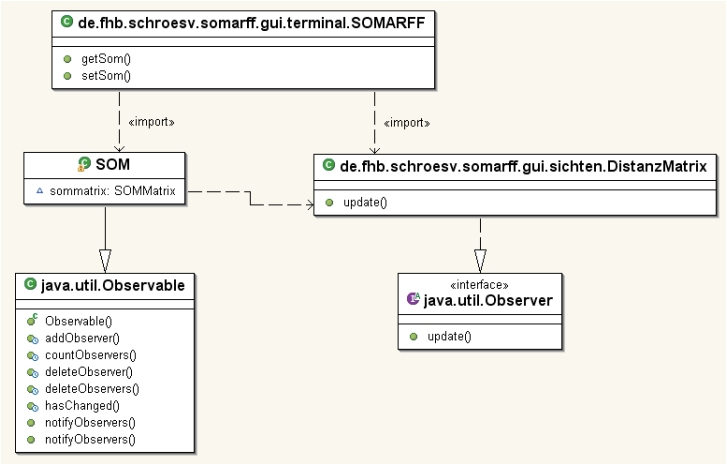

Abbildung 4.7: Entwurfsmuster: Observer

gesetzt. Die Fabrikklasse (GUI_Fabrik) erstellt auf Anfrage eine Instanz der geforderten Klasse für die Perspektive (zB. Gewinner-Matrix) und gibt sie an das Hauptfenster zurück.

4.3.3 Strategy

„Das Strategy-Muster definiert eine Familie von Algorithmen, kapselt sie einzeln und macht sie austauschbar. Das Strategy-Muster ermöglicht es, den Algorithmus unabhängig von den Clients die ihn einsetzen, variieren zu lassen.“ [EF06]

Das StrategyMuster gehört zur Gruppe der Verhaltensmuster. Es kapselt eine Familie von Algorithmen, so dass sie zur Programmlaufzeit austauschbar sind. Dadurch läßt sich das Problem der Unterklassenbildung und Mehrfachverzweigungen vermeiden. Außerdem erleichtert es die Wiederverwendung der Algorithmen.
Das Muster besteht aus einer Menge verwandter Klassen, welche sich nur in ihrem Verhalten unterscheiden. Das Muster definiert dazu eine Schnittstelle für die benötigten Algorithmen (Beispiel: „NachbarStratege“) und Klassen in denen die Algorithmen gekapselt sind (Beispiel: „Zylinder“, „Gauss“ etc.).

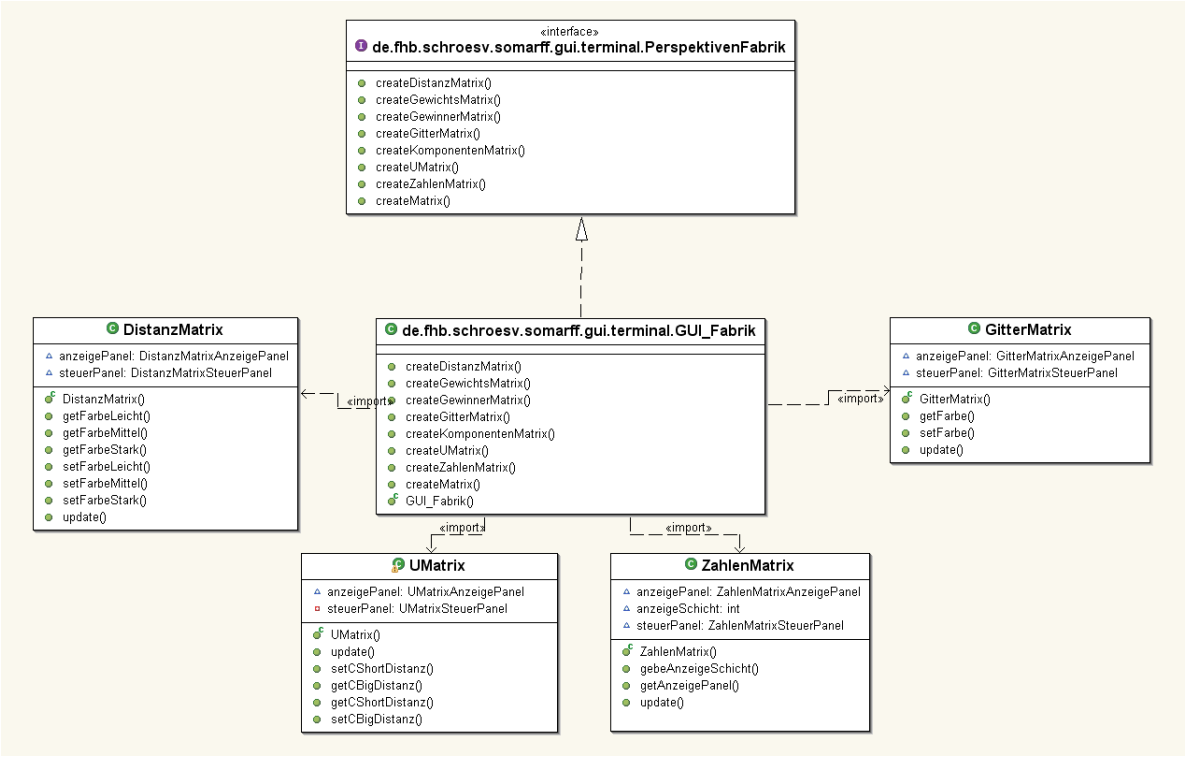

Abbildung 4.8: Entwurfsmuster: Factory

Verwendung fand dieses Entwurfsmuster bei sämtlichen ServiceStrategen, wodurch die Erweiterbarkeit des Systems sehr vereinfacht wurde.

4.4 Dateneingabe

4.4.1 ARFF-Files

ARFF-FilesARFF-Files sind einfache Textdateien zum Darstellen von einander unabhängigen und unsortierten Instanzen. Die Gesamtheit der Instanzen bilden das zu lernende Konzept in Bezug auf die Problemlösung ab. ARFF-Files bestehen aus

- Kommentaren
- dem Namen der abgebildeten Relation

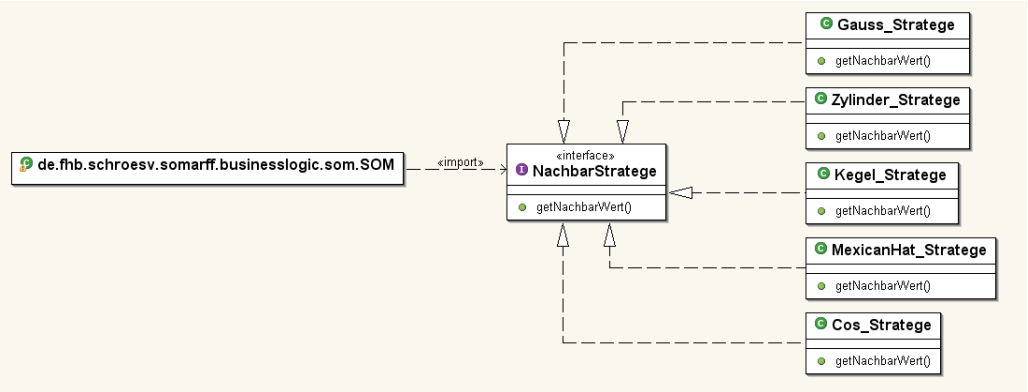

Abbildung 4.9: Entwurfsmuster: Strategy

- einem Block für die Beschreibung des Aufbaus einer Instanz
- dem Daten-Block

Kommentarzeilen beginnen mit einem %-Zeichen am Zeilenanfang und können an beliebiger Stelle in der Datei plaziert sein.
Beispiel:

```
%Dies ist ein Kommentar!
```

Der Realtionsname wird mit der Zeichenkette „@relation <name>“ beschrieben.
Beispiel:

```
@relation mietwagen
```

Nach dem Relationsnamen werden die Attributenamen der Instanzen mit dem zugehörigen Typ angegeben. Dies ähnelt den Spaltenköpfen einer Tabelle. Die Angabe erfolgt mit der Zeichenkette „@attribute <name> <typ>. Prinzipiell werden 2 Typen unterschieden, numerische und nominale. Bei numerischen Attributen folgt nach dem Attributnamen das Schlüsselwort numeric. Bei nominalen Attributen folgt dem Attributnamen eine Werteliste in geschweiften Klammmern.
Beispiel:

```
@attribute temperatur numeric
@attribute farbe {rot,schwarz,blau,gelb,grau,weiss}
```

Nach der Instanzbeschreibung folgt der Daten-Block, welcher mit der Zeichenkette „@data“ eingeleitet wird. Danach folgen die einzelnen Instanzen. Je Zeile eine Instanz. Die Attributwerte sind durch Komma getrennt. Fehlende Werte werden durch ein „?“ angegeben
Beispiel:

```
@data
90,rot
?,blau
30,?
95,schwarz
...
```

Programme, welche ARFF-Files verwenden, müssen prüfen, ob für jedes Attribut gültige Parameter vorhanden sind. (vgl. [WH01])

4.5 Datenausgabe

4.5.1 JPG-Dateien

Die Ablage der Analyseergebnisse im Dateisystem erfolgt über *.jpg-Dateien. JPG ist ein Format für Grafikdateien. Das Format verwendet Kompressionsverfahren um die Dateigröße zu verringern. Dies geschieht jedoch nicht verlustfrei. Die Daten werden als TrueColor-Bilder mit 24Bit je Pixel abgelegt.

4.6 Programmiersprache

Bei der Auswahl der Programmiersprache fiel die Wahl auf JAVA in der Version 1.4.
JAVA ist eine moderne Sprache und verwendet das objektorientierte Programmierparadigma. Mit ihr lassen sich leicht wiederverwendbare Softwarekomponenten entwickeln.

Ein weiterer Vorteil besteht in der Plattformunabhängigkeit der mit ihr erzeugten Programme. Sprachen wie „C“ oder „Delphi“ compilieren Quellcode für eine spezielle Plattform, zum Beispiel Windows oder Linux. Ein JAVA-Programm erzeugt beim Compilieren einen Bytecode, welcher zur Laufzeit auf dem ausführende Gerät durch eine Virtuelle-Maschine (JVM) in Maschinencode für die jeweilige Plattform interpretiert wird. Darin verbirgt sich auch ein Nachteil. JAVA-Programme sind durch den Interpretationsaufwand 3-10mal langsamer als direkt in Maschinencode vorliegende Programme. Dafür sind JAVA-Programme auf allen Geräten lauffähig, auf denen eine entsprechende JVM installiert wurde. Sie sind also plattformuniversal.
Weitere Vorteile von JAVA sind die Objektverwaltung durch den spracheigenen Garbage Collector, sowie die ausgereifte Ausnahmebehandlung.
SOMARFF benötigt keinen Zugriff auf plattformabhängige Ressourcen (zum Beispiel betriebssystemspezifische Aufrufe oder Ansteuerung eines Gerätes), außerdem verfügt der Autor über Erfahrung im Umgang mit der Programmiersprache, wodurch der Verwendung von JAVA nichts im Wege stand.

4.7 Anwendungsfallanalyse

4.7.1 Usecase-Diagramm

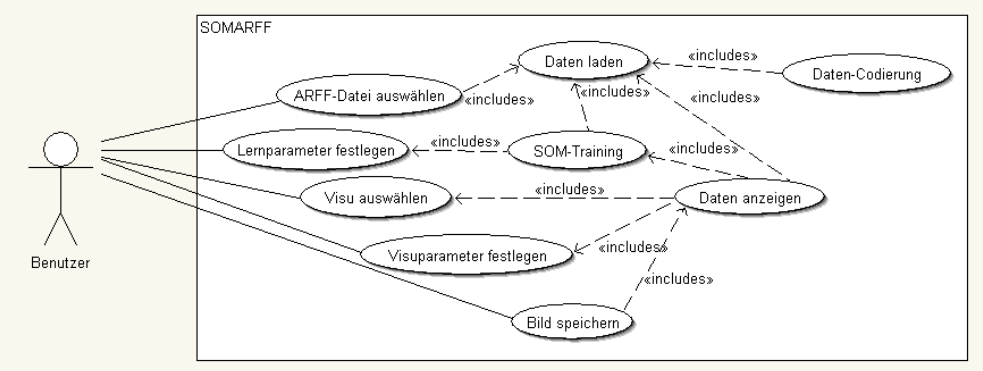

Abbildung 4.10: Anwendungsfalldiagramm

Die folgende Tabelle zeigt die identifizerten Anwendungsfälle sowie eine kurze Beschreibung.

Anwendungsfall	Beschreibung
Daten laden	Es werden Daten aus einer externen Datenquelle („*.arff“-Datei) gelesen und in einen Satz von Eingabemustern umcodiert. Der Anwendungsfall schließt die Anwendungsfälle „ARFF-Datei auswählen“ und „Daten-Codierung“ mit ein.
ARFF-Datei auswählen	Der Benutzer wählt als Datenquelle eine Datei vom Typ „*.arff“.
Codierregeln festlegen	Der Benutzer legt die Regeln für die Codierung der Attribute (Normierung, Zeichenkettenverarbeitung) fest. Außerdem legt der Benutzer fest, welche Spalte für das Beschriften (Labeln) in den Visualisierungen benutzt wird.
Daten-Codierung	Die durch den Benutzer gewählte Datei wird eingelesen und, gemäß den in Anwendungsfall „Codierregeln festlegen“ getroffenen Einstellungen, ein Satz von Eingabemustern aus dem Dateiinhalt erstellt.
SOM-Training	Dieser Anwendungsfall steht für den Lernvorgang der SOM. Er erhält aus Anwendungsfall „Lernparameter festlegen“ die für das Lernen notwendigen Parameter.
Lernparameter festlegen	Über diesen Anwendungsfall legt der Benutzer die Lernparameter (Abbruchkriterium, Lernrate, Nachbarfunktion) fest.
SOM laden	Der Benutzer kann eine SOM aus einer Datei laden.
SOM speichern	Der Benutzer kann eine SOM in einer Datei speichern.
Daten anzeigen	Dieser Anwendungsfall steht für die Darstellung der SOM in unterschiedlichsten Darstellungen. Dafür implementiert er die Anwendungsfälle „Visuparameter festlegen“ und „Visu auswählen“.
Visu auswählen	Der Benutzer wählt seine gewünscht(e)n Darstellungsart(en).
Visuparameter festlegen	Der Benutzer legt Darstellungsparameter wie Skalierung, Färbung usw. fest.
Bild speichern	Der Benutzer kann eine SOM-Darstellung als Bild speichern.

Bei der Erstellung eines UseCase-Diagramms konnte der Benutzer als einziger Akteur ermittelt werden.
Externe Systeme bzw. Sub-Systeme finden keine Verwendung.

4.7.2 Zustandsdiagramm und Szenarien

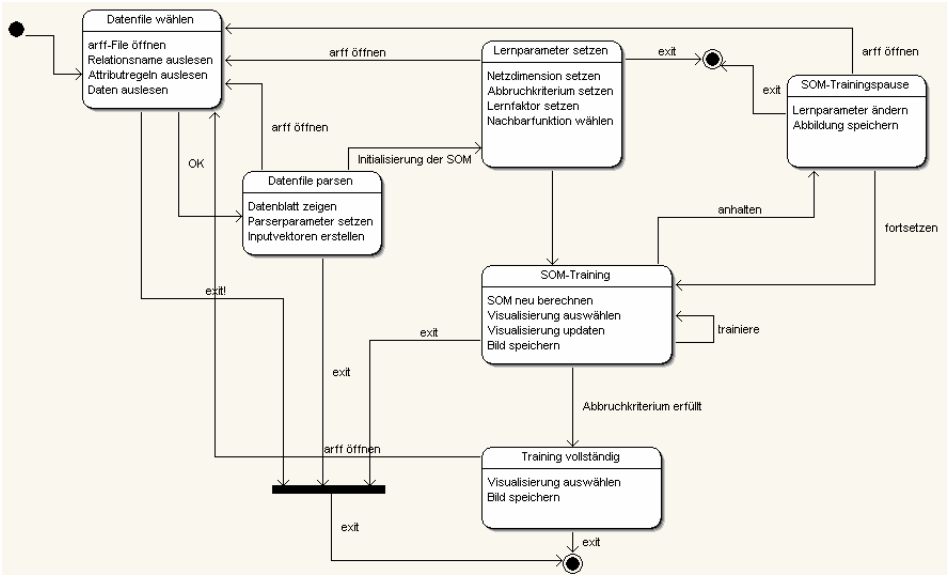

Abbildung 4.11: Statusdiagramm

Im Statusdiagramm werden sämtliche Zustände, in welche das System während des Betriebes übergehen kann, dargestellt. Das System kann aus jedem Zustand mittels „exit“ beendet werden.
Nach dem Start befindet sich das System im Zustand „Datenfile wählen“. In diesem Zustand wird die Datenquelle gewählt und ausgelesen.
Der nächste Zustand ist „Datenfile parsen“. In ihm erfolgt, nach eventueller Eingabe von Regeln, die Umwandlung der Daten in einen Satz von Eingabemustern sowie das Füllen der Datenblattansicht. Mit „arff öffnen“ erfolgt ein Übergang in Zustand „Datenfile wählen“.
Mit „Initialisierung der SOM“ erfolgt der Übergang in den Zustand „Parameter setzen“. In ihm können die SOM-Parameter für Struktur, Initialisierungsart und Netzdimension sowie die Lernparameter für Abbruchkriterium,

Lernrate und Nachbarfunktion angepasst werden. Mit „arff öffnen“ erfolgt ein Übergang in Zustand „Datenfile wählen“.
Der nachfolgende Zustand ist „SOM-Training“. In diesem Zustand wird die SOM unter Berücksichtigung der gesetzten Lernparameter iterativ neu berechnet und die SOM-Ansichten aktualisiert. Der Benutzer kann dabei frei aus den möglichen Visualisierungsarten wählen. Bei „anhalten“ geht das System in den Zustand „SOM-Trainingspause“ über. Wird das Abbruchkriterium erfüllt, ändert sich der Zustand in „Training vollständig.“
Im Zustand „SOM-Trainingspause“ können Visualisierungen gespeichert und die Lernparameter modifiziert werden. Wird das Netztraining gestoppt, geht das System in den Zustand „Datenfile wählen“ durch „arff öffnen“ über. Bei Fortsetzen fällt das System wieder in den iterativen Zustand SOM-Training. Nach Trainingsende befindet sich das Programm im Zustand „Training vollständig“. In diesem Zustand kann der Benutzer Visualisierungen auswählen und speichern sowie, durch „arff öffnen“ ein neues Datenfile wählen.

Das Sequenzdiagramm zeigt ein typisches Anwendungsszenario, an welchem die Objekte Benutzer, GUI, ARFF-Parser und SOM beteiligt sind. Benutzer steht für den menschlichen Bediener (Akteur). Das GUI stellt die Benutzeroberfläche und eine Schnittstelle für die Systembenutzung dar (Controller). Der ARFF-Parser steht für eine Komponente, welche Daten aus einer *.arff-Datei in eine Menge von Eingabemustern, die Inputvektoren), umwandelt. SOM stellt die Selbstorganisierende Karte dar.
Bei Szenariobeginn sendet der Benutzer dem GUI, welche Datei zum Training verwendet werden soll. Der ARFF-Parser liest anschließend den Dateiinhalt und parst aus ihm einen Pool von Eingabemustern. Der Benutzer stellt danach die gewünschten Parameter für die SOM (Struktur, Initialisierung, Dimension) und für den Lernvorgang (Nachbarfunktion, Abbruchkriterium, Lernrate) ein. Sendet der Benutzer die Nachricht für den Trainingsstart, wird ein SOM-Objekt erstellt und die Menge der Eingabemuster (MusterPool) an die SOM übergeben. Danach führt SOM eine Initialisierung der Synapsengewichte unter Beachtung der Parameter durch und startet den iterativen Lernvorgang. Während dieses Vorgangs hat der Benutzer jederzeit die Möglichkeit, gewünschte Visualisierungen zu öffnen oder zu schließen. Auch

mehrere Visualisierungen der gleichen Art sind möglich. Während des Trainigs werden die Visualiserungen über Änderungen der SOM informiert und aktualisiert. Wird vom Benutzer eine Nachricht für eine Trainingspause gesendet, so benachrichtigt das GUI die SOM, welche darauf im Lernschritt pausiert. Während die SOM pausiert, hat der Benutzer die Möglichkeit, Änderungen an den Lernparametern vorzunehmen, sie werden vom GUI an die SOM weitergeleitet. Nach Nachricht für Trainingfortsetzung, benachrichtigt das GUI die SOM und die SOM führt den Traingsprozess solange fort, bis das Abbruchkriterium erfüllt ist. Der Benutzer informiert die GUI, dass eine Visualisierung als Bilddatei gespeichert werden soll und schließt die Visualisierung anschließend. Zum Szenarioende empfängt das GUI vom Benutzer die Nachricht, dass die SOM gespeichert werden soll. Danach beendet der Benutzer die Anwendung.

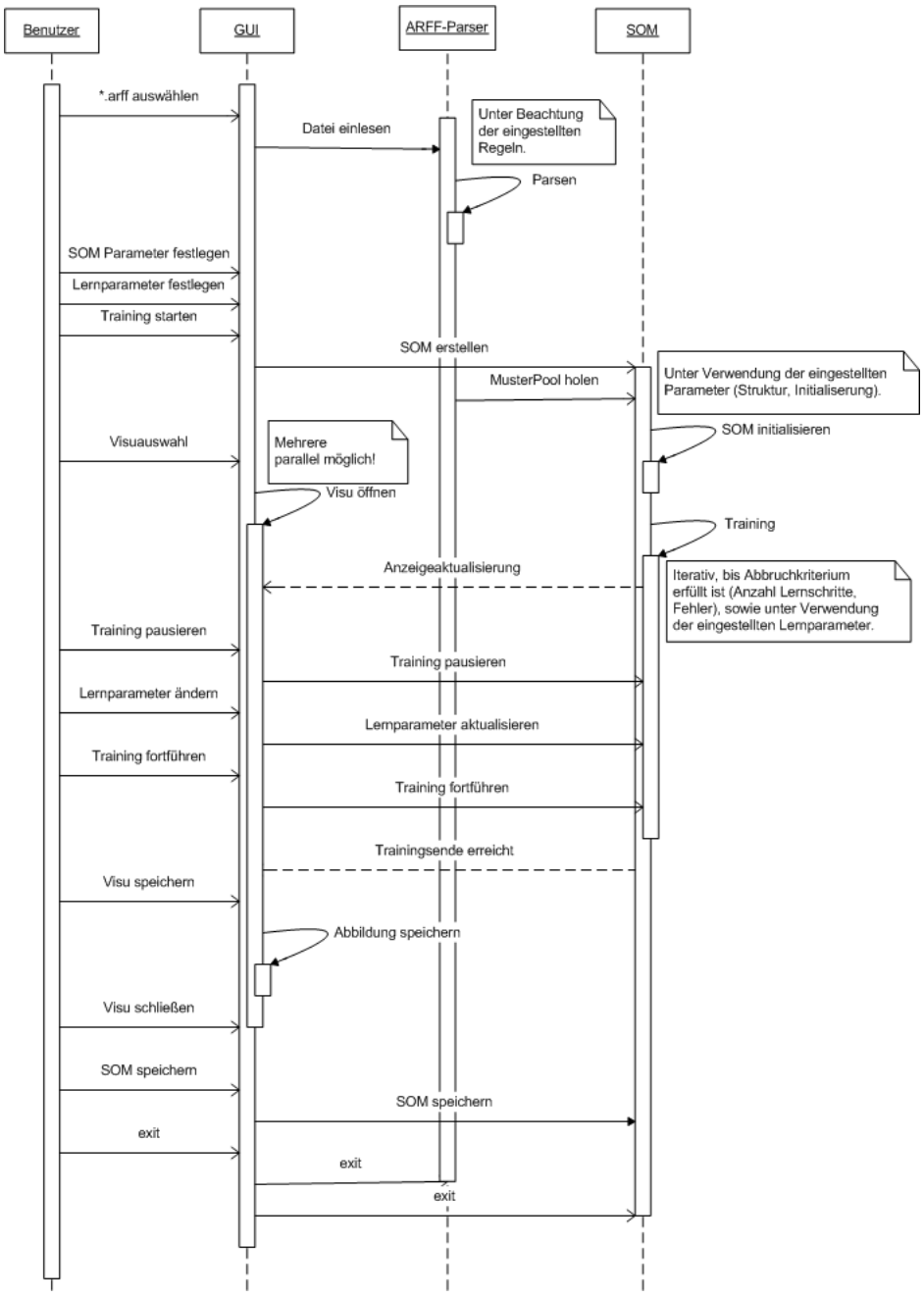

Abbildung 4.12: Sequenzdiagramm

Kapitel 5

Implementierung

5.1 Klassenbeschreibungen

5.1.1 Komponente Parser

Die Komponente Parser gehört zur Schicht der Anwendungslogik. Sie ist für das Einlesen der Datenquelle und das Umformen der Daten in eine Präsentationsart, mit welcher eine Kohohnenkarte trainiert werden kann, zuständig.

Die Komponente ist als Schnittstelle angelegt, über deren Methoden ein Zugriff von aussen auf die Komponente erfolgen kann. Die Komponente besitzt eine Klasse „ArffParser“, welche das Verarbeiten von Dateien im *.arff-Format unterstützt, kann jedoch durch Implementierung der Schnittstelle für andere Datenquellen erweitert werden, z.B. für MS Excel oder relationale Datenbanken.

Der Komponente wurde ein Satz an Service-Klassen, „Modifikatoren“ genannt, beigefügt. Diese Klassen beinhalten konkrete Algorithmen zur Verarbeitung der unterschiedlichen Attribute. Folgend werden alle zur Komponente gehörenden Pakete mit ihren Klassen beschrieben. Außerdem wird auf die Dokumentation, welche im jar.Archiv enthalten ist (javadoc), verwiesen.

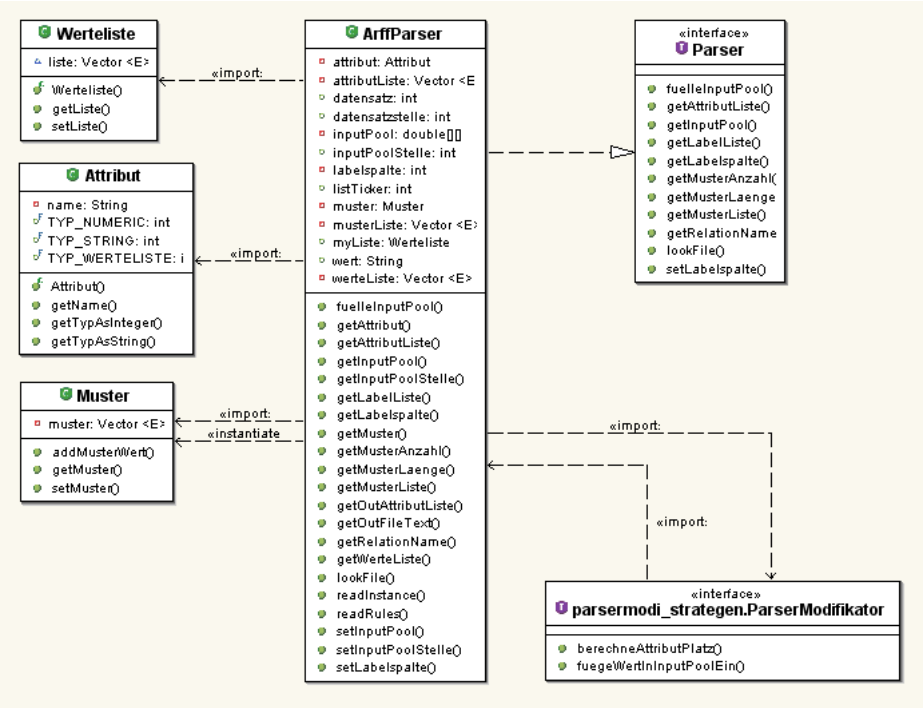

Abbildung 5.1: Klassendiagramm Parser-Komponente

Klasse Parser

Paket: de.fhb.schroesv.somarff.businesslogic.parser

Dies ist die Schnittstelle der Komponente für den Zugriff von aussen. Sie muss von der Basisklasse der Komponente, einem Parser für eine konkrete Datenquelle, implementiert werden.

Klasse ArrfParser

Paket: de.fhb.schroesv.somarff.businesslogic.parser

Diese Klasse ist die Basisklasse der Komponente. Sie implementiert die Schnittstelle Parser

Erfolgt ein Aufruf der Methode „lookFile“ von aussen, wird die Analyse ei-

ner *.arff-Datei gestartet. Um die Daten korrekt interpretieren zu können, ist eine Analyse des Attributblocks und des Datenblocks notwendig. Die Ermittlung des Realtionsname hat zwar nur informativen Charakter, wird jedoch dabei gleich mit erledigt.

```
1 public void lookFile(String pfad, String filename){
2   this.readRules(pfad,filename);
3   this.readInstance(pfad, filename);
4 }
```

Dazu wird die Methode „readRules“ aufgerufen. Sie öffnet einen Datenstrom in die Datei, lässt ihn über die Methode „dateiPufferAuslesen“ verarbeiten und schließt sie den Datenstrom danach wieder. Die Methode „dateiPufferAuslesen“ verarbeitet den Datenstrom zeilenweise. Die Methode prüft der Reihe nach durch, ob die Zeile den Namen der Relation, eine Attributdeklaration oder der Beginn des Datenblocks ist. Dafür benutzt sie die Methode „findOrder“. Diese Methode prüft lediglich, ob die ersten Zeichen der Zeile einen bestimmten String beinhalten (@relation, @attribut, @data) und gibt das Prüfergebnis zurück. Trifft ein Fall zu, so wird die eine entsprechende Handlungsmethode aufgerufen. Wird die Zeile für den Relationsnamen gefunden, so wird „extractRelationName“ und für eine Attributsdeklaration „extractAttribut“ aufgerufen. Wird der Beginn des Datenblocks gefunden, so wird lediglich die Zeilennummer gespeichert.

```
1  while(br.ready()){
2    str = br.readLine();
3    str.toLowerCase();
4
5    relation = findOrder("@relation", str);
6    if(relation) relationname = extractRelationName(str);
7    relation = false;
8
9    attribut = findOrder("@attribute", str);
10   if(attribut)extractAttribut(str);
11   attribut = false;
```

```
12
13    data = findOrder("@data", str);
14    if(data){
15     this.beginDataBlock = nr-1;
16    }
17    data = false;
18    nr++;
19  }
```

Die Methode „extractRelationName“ schneidet lediglich das Schlüsselwort von der Zeile ab, entfernt die führenden und schließenden Whitespacezeichen und gibt als Reststring den Ralationsnamen zurück. Die Methode „extractAttribut“ ist wesentlich komplizierter. Auch sie entfernt zunächst das Schlüsselwort und entfernt die Whitespacezeichen, jedoch muss hier zwischen dem Attributnamen und der folgenden Typinformation unterschieden werden. Dies wurde durch Suchen nach dem nächsten Leerzeichen realisiert, da dieses das Ende des Attributnamens markiert. Dieser wurde dann wie das Schlüsselwort entfernt und in einer Variable gespeichert. Um an den Attributtyp zu kommen, wurde wieder die Methode „findOrder“ auf den Reststring angewandt. Diesmal wurde nach Hinweisen auf die Attributtapen getestet. Die ist für den numerischen Typ „numeric“ und für den nominalen Typ das Zeichen „{“. Da in einigen, aus dem Internet geladenen, Dateien auch der Datentyp „String“ vorkam, wurde auch dieser eingeführt. Danach ist ein Attributobjekt erzeugt worden, dem der Name und der Typ (0=numeric, 1=String, Werteliste=3) übergeben wurde.

```
1  private void extractAttribut(String zeile){
2    String name = "";
3    boolean endOfName = false;
4    int i = 0;
5    zeile = zeile.substring("@attribute".length());
6    zeile = zeile.trim();
7    while(!endOfName){
8      String test = "" + zeile.charAt(i);
9      if(test.equals("␣")){
10       endOfName = true;
11     }
```

```
12      i++;
13    }
14    name = zeile.substring(0,i);
15    name = name.trim();
16    zeile = zeile.substring(name.length());
17    zeile = zeile.trim();
18    if(findOrder("numeric",zeile)){
19      Attribut a = new Attribut(name,0);
20      attributListe.add(a);
21    }else{
22      if(findOrder("String",zeile)){
23        Attribut a = new Attribut(name,1);
24        attributListe.add(a);
25      }else{
26        if(findOrder("{",zeile)){
27          Vector liste = new Vector();
28          liste = this.extractWerte(zeile);
29          Attribut a = new Attribut(name,2);
30          attributListe.add(a);
31          Werteliste wl = new Werteliste(liste);
32          werteListe.add(wl);
33        }else{
34          JOptionPane.showMessageDialog( null,
35            "Unbekannter Datentyp bei einem Attribut!"
36          );
37          throw new IllegalArgumentException(
38            "Unbekannter Datentyp!"
39          );
40        }
41      }
42    }
43  }
```

Wurde ein Attribut mit einer Liste von Werten, auch Werteliste genannt, gefunden, so wurde über die Methode „extractWerte“ ein Vector-Objekt mit den Listen Einträgen erzeugt und in einem weiteren Vector, welcher alle gefundenen Listen speichert, eingefügt. Um die gültigen nominalen Werte zu extrahieren, entfernt die Methode „extractWerte“ zunächst alle Zeichen, wel-

che vor und hinter den geschweiften Klammern stehen, inklusive der Klammer selbst. Danach wird stets bis zum nächsten Komma oder dem Stringende gescannt und in Blöcke geteilt. Die Blöcke ergeben dann die gültigen Attributwerte, welche in der zurückzugebenen Liste (Vector) eingefügt werden.

Nach Abarbeitung von „readRules“ startet die Methode“lookFile“ die Methode „readInstance“. Diese Methode springt an den Beginn des Datenblockes, initialisiert ein Muster-Objekt, und füllt dessen Muster-Vektor mit den einzelnen, in den Datenblock gefundenen, Mustern in Form von Strings. Dazu zerlegt es die Datenzeilen in einzelne Fragmente (durch „,“ getrennt). Nach Abarbeitung einer Datenzeile wird das Muster-Objekt in einen Vector, „musterListe“ genannt, eingefügt.

Ein Aufruf der Methode „fuelleInputPool“ erstellt aus der analysierten Datei einen Pool aus Eingabemustern, welche so umgeformt wurden, dass sie an das neuronale Netz angelegt werden und verarbeitet werden können. Die Methode erhält bei Aufruf 3 Integer-Zahlen, welche Aufschluss darüber geben, welcher Modifikator für die Attributverarbeitung benutzt werden soll (siehe Modifikatoren). Die Informationen werden durch die Steuerung des Programms durch den Benutzer über das GUI eingestellt. Bei Methodenstart wird die Methode „erstelleModifikator“ für die Attribuittypen aufgerufen. Die Methode „erstelleModifikator“ instanziiert eine Modifikatorklasse, welche über den, als Parameterwert angegebenen, Integerwert spezifiziert wurde und liefert ihn an die aufrufende Klasse zurück. Folgende Auflistung zeigt die verfügbaren Modifikatoren:

0 Für numerische Attribute, die einfach übernommen werden sollen.

1 Für numerische Attribute, welche normiert zwischen $[-1, 1]$ werden sollen.

10 Für nominale Attribute, welche durch ganze Zahlen die aufsteigend im Bereich von $[1, n]$ codiert werden sollen.

20 Für nominale Attribute, welche durch eine Liste von Werten spezifiziert werden und durch ganze Zahlen die aufsteigend im Bereich von $[1, n]$

codiert werden sollen.

21 Für nominale Attribute, welche durch eine Liste von Werten spezifiziert werden und durch normierte Zahlen zwischen $[0, 1]$ codiert werden sollen.

22 Für nominale Attribute, welche durch eine Liste von Werten spezifiziert werden und nach ihrem Vorkommen in Bitstellen codiert werden sollen.

Nach der Auswahl der Modifikatoren ermittelt die Methode „fuelleInputPool“ den nötigen Platzbedarf der Attribute in einem Muster. Dazu wird die Methode „berechneAttributPlatz“ in den Modifikatoren verwendet, welche in jedem Modifikator implementiert ist (siehe Modifikatoren). Mit den berechneten Werten wird anschliessend ein 2-dimensionales Double-Array initialisiert und mit Aufruf der Methode „initInputPool“ auf allen Feldern mit 0 initialisiert.

```
1  public double[][] fuelleInputPool(
2          int modusNumeric,
3          int modusString,
4          int modusListe){
5    ParserModifikator numericModifikator
6          = this.erstelleModifikator(modusNumeric);
7    ParserModifikator stringModifikator
8          = this.erstelleModifikator(modusString);
9    ParserModifikator listModifikator
10         = this.erstelleModifikator(modusListe);
11   int platz = 0;
12   int ticker = 0;
13   for(int i=0;i<this.getAttributListe().size();i++){
14     Attribut a = (Attribut)this.getAttributListe().get(i);
15     int typ = a.getTypAsInteger();
16     switch(typ){
17       case 0:
18         platz =
19           platz+numericModifikator.berechneAttributPlatz(this);
20         break;
21       case 1:
22         platz =
```

```
23            platz+stringModifikator.berechneAttributPlatz(this);
24          break;
25        case 2:
26          myListe = (Werteliste)this.werteListe.get(ticker);
27          platz =
28            platz+listModifikator.berechneAttributPlatz(this);
29          ticker++;
30          break;
31        default:
32      }
33    }
34    inputPool = new double[this.musterListe.size()][platz];
35    this.initInputPool(this.musterListe.size(), platz);
36    this.getOutInputPool();
37    this.umcodieren(
38          numericModifikator,
39          stringModifikator,
40          listModifikator);
41    this.getOutInputPool();
42          return this.inputPool;
43  }
44
45  private void umcodieren(
46          ParserModifikator numericModifikator,
47          ParserModifikator stringModifikator,
48          ParserModifikator listModifikator) {
49    for(int c=0;c<this.musterListe.size();c++){
50      Muster muster = (Muster)this.musterListe.get(c);
51      listTicker=0;
52      datensatzstelle=0;
53      inputPoolStelle=0;
54      for(int i=0;i<muster.getMuster().size();i++){
55        datensatz = c;
56        datensatzstelle=i;
57        wert = (String)muster.getMuster().get(datensatzstelle);
58        attribut =
59          (Attribut)this.attributListe.get(datensatzstelle);
60        wert=wert.trim();
61        switch(attribut.getTypAsInteger()){
```

```
62            case 0:
63              numericModifikator.fuegeWertInInputPoolEin(this);
64              break;
65            case 1:
66              stringModifikator.fuegeWertInInputPoolEin(this);
67              break;
68            case 2:
69              listModifikator.fuegeWertInInputPoolEin(this);
70              listTicker++;
71              break;
72            default:
73          }
74          this.inputPoolStelle++;
75        }
76      }
77  }
```

Danach erfolgt ein Aufruf der Methode „umcodieren“, welche dafür sorgt, dass der InputPool mit den richtigen Werten der Muster gefüllt wird. Sie durchläuft alle Muster der „musterListe“, holt die für die Codierung wichtigen Daten aus dem Muster und ruft den verantwortlichen Modifikator für den Attributtyp mit „fuegeWertInInputPoolEin“ auf. Diese Methode implementiert jeder Modifikator. Wenn alle Muster durchlaufen wurden, befinden sich die aus der Datei gelesenen Daten in gewünscht codierter Form im MusterPool, welcher über die „getInputPool“-Methode von Aussen geholt werden kann.

Klasse ParserModifikator

Paket: de.fhb.schroesv.somarff.businesslogic.parser.parsermodi_strategen

Dies ist die Schnittstelle der Parser-Modifikatoren. Sie wird von allen Modifikatoren implementiert, um einheitlich gleiche Zugriffsmethoden für den Parser sicherzustellen und die Modifikatorklassen austauschbar zu halten.

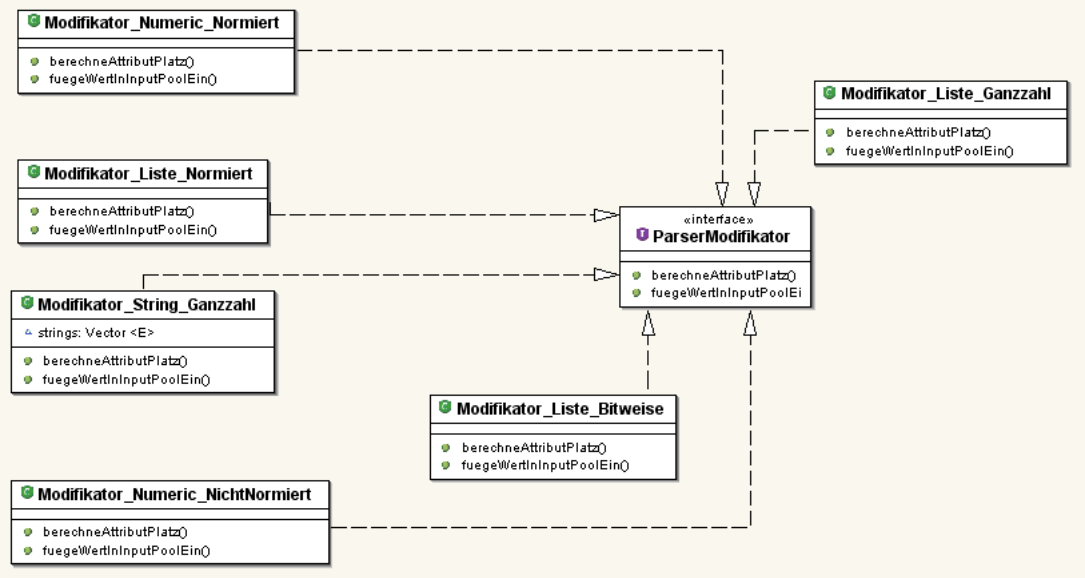

Abbildung 5.2: Klassendiagramm Parser-Modifikatoren

Klasse Modifikator_Numeric_NichtNormiert

Paket: de.fhb.schroesv.somarff.businesslogic.parser.parsermodi_strategen

Da die Klasse die Schnittstelle „ParserModifikator“ implementiert, verfügt sie über die öffentllichen Zugriffsmethoden „berechneAttributPlatz“ und „fuegeWertInInputPoolEin“.
Die Methode „berechneAttributPlatz“ gibt lediglich die Konstante 1 zurück, da nur ein Double-Feld im InputPool für den Wert dieses Typs benötigt wird. Die Methode „fuegeWertInInputPoolEin“ kopiert den String-Wert aus dem Muster, parst ihn in einen Double-Wert und fügt ihn in die übergebene Stelle im InputPool ein. Tritt dabei ein Fehler auf, z.B. wenn das Zeichen „?“ für unbekannte Werte enthalten ist, so wird es mit 0.0 codiert.

```
1 public void fuegeWertInInputPoolEin(ArffParser parser) {
2   double[][][] p = parser.getInputPool();
3   int dsNr = parser.datensatz;
4   String wert = parser.wert;
5   int ipstelle = parser.getInputPoolStelle();
```

```
6     try{
7       p[dsNr][ipstelle]=Double.parseDouble(wert);
8     }catch(NumberFormatException nfe){
9       p[dsNr][ipstelle]=0.0;
10    }
11    parser.setInputPool(p);
12  }
```

Klasse Modifikator_Numeric_Normiert

Paket: de.fhb.schroesv.somarff.businesslogic.parser.parsermodi_strategen

Da die Klasse die Schnittstelle „ParserModifikator“ implementiert, verfügt sie über die öffentllichen Zugriffsmethoden „berechneAttributPlatz“ und „fuegeWertInInputPoolEin“.
Die Methode „berechneAttributPlatz“ gibt lediglich die Konstante 1 zurück, da nur ein Double-Feld im InputPool für den Wert dieses Typs benötigt wird. Die Methode „fuegeWertInInputPoolEin“ funktioniert ähnlich wie die nicht normierte Version. Jedoch wird in diesem Modifikator zuerst der maximalste Wert der Attributspalte ermittelt und anschließend der geparste Wert mit der Klasse „Standard_Normalisierer“ in einen Wert im Bereich $[-1, 1]$ normiert. Anschließend wird der Wert an der übergebenen Methode im InputPool eingefügt. Tritt dabei ein Fehler auf, z.B. wenn das Zeichen „?“ für unbekannte Werte enthalten ist, so wird es mit 0.0 codiert.

```
1   public void fuegeWertInInputPoolEin(ArffParser parser) {
2     int dsNr = parser.datensatz;
3     int dsstelle = parser.datensatzstelle;
4     double[][] p = parser.getInputPool();
5     int ipstelle = parser.getInputPoolStelle();
6     String wert = parser.wert;
7     Normalisierer n = new Standard_Normalisierer();
8     double max = ermittleMaximum(parser, dsstelle);
9     try{
10      if(max!=0){
11        double outline =
12          n.normalisiereWert(Double.parseDouble(wert),max);
```

```
13          p[dsNr][ipstelle]=outline;
14        }else{
15          p[dsNr][ipstelle]=0.0;
16        }
17      }catch(NumberFormatException nfe){
18        p[dsNr][ipstelle]=0.0;
19      }
20      parser.setInputPool(p);
21    }
```

Klasse Modifikator_String_Ganzzahl

Paket: de.fhb.schroesv.somarff.businesslogic.parser.parsermodi_strategen

Da die Klasse die Schnittstelle „ParserModifikator“ implementiert, verfügt sie über die öffentllichen Zugriffsmethoden „berechneAttributPlatz“ und „fuegeWertInInputPoolEin“.
Die Methode „berechneAttributPlatz“ gibt lediglich die Konstante 1 zurück, da nur ein Double-Feld im InputPool für den Wert dieses Typs benötigt wird. Die Methode „fuegeWertInInputPoolEin“ erstellt eine Indexliste für sämtliche in der Attributspalte vorkommenden Stringwerte über alle Muster. Dazu benutzt sie die private Methode „ermittleVorhandeneWerte“ welche einen Vector mit den gefundenen Strings zurückgibt. Jeder Wert ist im Vector nur einmal vorhanden und über seine Position eindeutig. Danach wird die private Methode „ermittleGanzzahlWert“ aufgerufen. Sie vergleicht Attribut mit jedem Listeneintrag auf Gleichheit . Wird sie fündig, wird die Position im Vector zurückgegeben. Falls nicht, bleibt der Rückgabewert 0.0.

```
1  public void fuegeWertInInputPoolEin(ArffParser parser){
2    double[][] p = parser.getInputPool();
3    int dsNr = parser.datensatz;
4    int dsstelle = parser.datensatzstelle;
5    String wert = parser.wert;
6    int ipstelle = parser.getInputPoolStelle();
7    this.ermittleVorhandeneWerte(parser, dsstelle);
8    p[dsNr][ipstelle]=this.ermittleGanzzahlWert(wert);
```

```
9     parser.setInputPool(p);
10  }
11
12  private Vector ermittleVorhandeneWerte(
13          ArffParser parser,
14          int dsstelle)
15          throws NumberFormatException {
16    boolean gefunden = false;
17    for(int c=0; c<parser.getMusterListe().size();c++){
18      Muster muster = (Muster)parser.getMusterListe().get(c);
19      String testwert = (String)muster.getMuster().get(dsstelle);
20      for (Iterator i=strings.iterator();i.hasNext(); ){
21        String element = (String)i.next();
22        if(testwert.equals(element)){
23          gefunden = true;
24        }
25      }
26      if(gefunden==false){
27        strings.addElement(testwert);
28      }
29    }
30    return strings;
31  }
32
33  private int ermittleGanzzahlWert(String wert){
34    int ausgabewert = 0;
35    for(int i=0;i<strings.size();i++){
36      if(wert.equals(strings.get(i))){
37        ausgabewert = i;
38      }
39    }
40    return ausgabewert;
41  }
```

Klasse Modifikator_Liste_Bitweise

Paket: de.fhb.schroesv.somarff.businesslogic.parser.parsermodi_strategen

Da die Klasse die Schnittstelle „ParserModifikator“ implementiert, verfügt sie über die öffentllichen Zugriffsmethoden „berechneAttributPlatz“ und „fuegeWertInInputPoolEin“.
Die Methode „berechneAttributPlatz“ gibt die Anzahl der Einträge der übergebenen Werteliste zurück, da jeder Attributwert bei diesem Typ soviele Bitstellen, wie Elemente in der Liste vorhanden sind, benötigt. Zum Beispiel würde ein Wert für die Liste $\{schwarz, grau, weiss\}$ in 3 Bitstellen codiert werden, also für $schwarz = 001$, für $grau = 010$ und für $weiss = 100$.
Die Methode „fuegeWertInInputPoolEin“ holt sich das aktuelle Werteliste-Objekt aus dem aufrufenden Parser und prüft die Liste durch, ob der Attributwert einem Listenwert entspricht. Bei Gleichheit wird die entsprechende Bitstelle auf 1 gesetzt. Befindet sich der Attributwert nicht in Liste verbleiben alle Bits beim Wert 0. Anschließend wird der InputPool mit den Bitstellen überschrieben.

```
1  public void fuegeWertInInputPoolEin(ArffParser parser) {
2    double[][] p = parser.getInputPool();
3    int dsNr = parser.datensatz;
4    int ipstelle = parser.getInputPoolStelle();
5    String wert = parser.wert;
6    int ticker = parser.listTicker;
7    Werteliste myListe = null;
8    myListe = (Werteliste)parser.getWerteListe().get(ticker);
9    for(int c=0;c<myListe.getListe().size();c++){
10     if(wert.equals(myListe.getListe().get(c))){
11       p[dsNr][ipstelle +c] = 1;
12     }
13   }
14   parser.setInputPoolStelle(
15     ipstelle + myListe.getListe().size()-1
16     );
17   parser.setInputPool(p);
18 }
```

Klasse Modifikator_Liste_Ganzzahl

Paket: de.fhb.schroesv.somarff.businesslogic.parser.parsermodi_strategen

Da die Klasse die Schnittstelle „ParserModifikator“ implementiert, verfügt sie über die öffentllichen Zugriffsmethoden „berechneAttributPlatz“ und „fuegeWertInInputPoolEin“.
Die Methode „berechneAttributPlatz“ gibt lediglich die Konstante 1 zurück, da nur ein Double-Feld im InputPool für den Wert dieses Typs benötigt wird.
Die Methode „fuegeWertInInputPoolEin“ holt sich das aktuelle Werteliste-Objekt aus dem aufrufenden Parser und prüft die Liste durch, ob der Attributwert einem Listenwert entspricht. Bei Gleichheit wird die Position in Werteliste um 1 erhöht und an der entsprechenden Stelle im InputPool abgelegt. Befindet sich der Attributwert nicht in der Liste, so bleibt der Wert im InputPool bei 0.0.

```
1  public void fuegeWertInInputPoolEin(ArffParser parser) {
2    double[][] p = parser.getInputPool();
3    int dsNr = parser.datensatz;
4    String wert = parser.wert;
5    wert.trim();
6    int ticker = parser.listTicker;
7    int ipstelle = parser.getInputPoolStelle();
8    Werteliste myListe = null;
9    myListe = (Werteliste)parser.getWerteListe().get(ticker);
10   for(int c=0; c<myListe.getListe().size(); c++){
11     if(wert.equals(myListe.getListe().get(c))){
12       p[dsNr][ipstelle] = c+1;
13     }
14     if(wert.equals("?")){
15       p[dsNr][ipstelle] = 0;
16     }
17   }
18   parser.setInputPool(p);
19 }
```

Klasse Modifikator_Liste_Normiert

Paket: de.fhb.schroesv.somarff.businesslogic.parser.parsermodi_strategen

Da die Klasse die Schnittstelle „ParserModifikator“ implementiert, verfügt sie über die öffentllichen Zugriffsmethoden „berechneAttributPlatz“ und „fuegeWertInInputPoolEin“.
Die Methode „berechneAttributPlatz“ gibt lediglich die Konstante 1 zurück, da nur ein Double-Feld im InputPool für den Wert dieses Typs benötigt wird. Die Methode „fuegeWertInInputPoolEin“ funktioniert ähnlich der im Modifikator „Modifikator_Liste_Ganzzahl“, nur erfolgt hier eine Normierung des Wertes zwischen $[0, 1]$. Bei Gleichheit wird die Position in Werteliste um 1 erhöht und mit einem Multiplikator, welcher sich aus $\frac{1}{AnzahlListenelemente}$ ergibt, multipliziert. Der resultierende Wert wird anschließend an der entsprechenden Stelle im InputPool abgelegt. Befindet sich der Attributwert nicht in der Liste, so bleibt der Wert im InputPool bei 0.0.

```
1  public void fuegeWertInInputPoolEin(ArffParser parser) {
2    double[][] p = parser.getInputPool();
3    int dsNr = parser.datensatz;
4    String wert = parser.wert;
5    int ticker = parser.listTicker;
6    int ipstelle = parser.getInputPoolStelle();
7    Werteliste myListe = null;
8    int multiplikator = 0;
9    myListe = (Werteliste)parser.getWerteListe().get(ticker);
10   double wertigkeit = 1.0/myListe.getListe().size();
11   for(int c=0; c < myListe.getListe().size(); c++){
12     if(wert.equals(myListe.getListe().get(c)))  {
13       multiplikator = c+1;
14     }
15   }
16   p[dsNr][ipstelle] = multiplikator * wertigkeit;
17   if(wert.equals("?")){
18     p[dsNr][ipstelle]    =        0;
19   }
20   parser.setInputPool(p);
```

```
21 }
```

Klasse Attribut

Paket: de.fhb.schroesv.somarff.businesslogic.som

Die Klasse dient als Datencontainer zur Speicherung von Name und Attributtyp.

Klasse Muster

Paket: de.fhb.schroesv.somarff.businesslogic.som

Die Klasse speichert eingehende Musterfragmente und dient als Datencontainer eines einzelnen Musters.

Klasse WerteListe

Paket: de.fhb.schroesv.somarff.businesslogic.som

Die Klasse dient als Datencontainer zur Speicherung von Wertemengen des Typs „WerteListe“.

5.1.2 Komponente SOM

Die Komponente SOM gehört zur Schicht der Anwendungslogik. Sie ist für die Initialisierung des Kohonen-Netzes sowie des Trainingsvorgangs zuständig.

Der Komponente wurde ein Satz an Service-Klassen (Strategen) beigefügt. Diese Klassen beinhalten konkrete Algorithmen für die anfallenden Aufgaben. In der folgenden Tabelle werden die Strategen und ihre Aufgaben dargestellt.

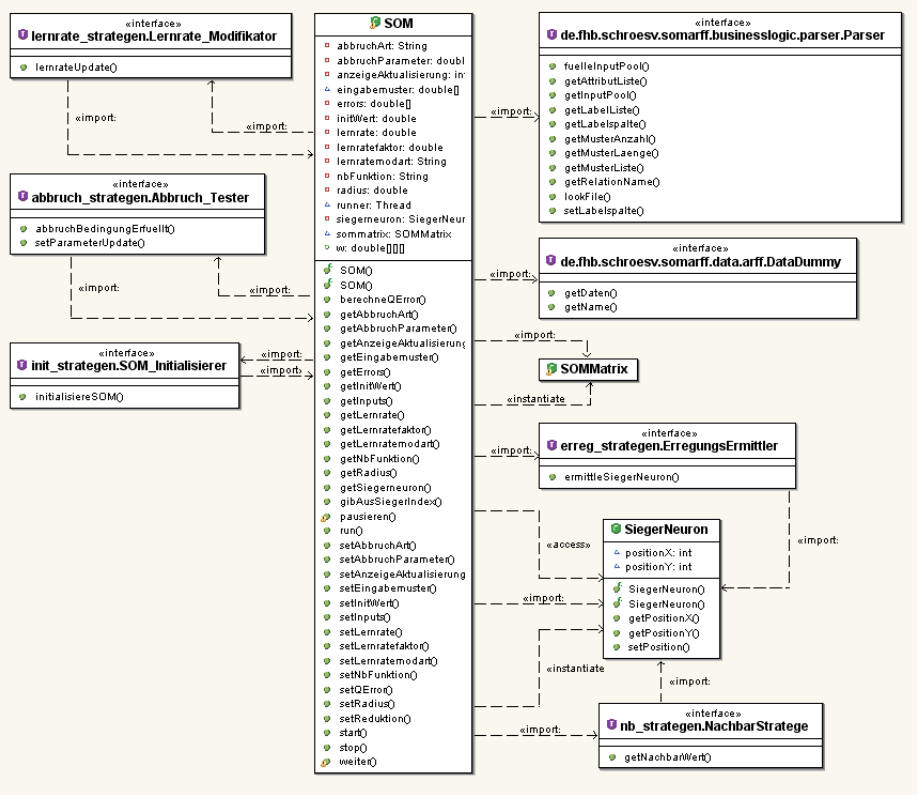

Abbildung 5.3: Klassendiagramm SOM-Komponente

Name der Schnittstelle	Aufgabe
SOM_Initialisierer	Legt die Werte der Verbindungsgewichte der SOM vor dem Netztraining fest.
ErregungsErnittler	Ermittelt das Siegerneuron für ein Muster.
LernrateModifikator	Reduktion der Lernrate nach mathematischer Funktion.
NachbarStratege	Ermittlung des Nachbarschaftswertes nach mathematischer Funktion.
Normalisierer	Normalisierung von numerischen Werten in einen bestimmten Wertebereich.
Abbruch_Tester	Prüfung der Abbruchsbedingung für das Netztraining.

Desweiteren verwendet die Komponente einige Klassen, welche zur Datenkapselung und Kommunikation genutzt werden.

Folgend werden alle zur Komponente gehörenden Pakete mit ihren Klassen beschrieben. Außerdem wird auf die Dokumentation, welche im jar.Archiv enthalten ist (javadoc), verwiesen.

Klasse SOM

Paket: de.fhb.schroesv.somarff.businesslogic.som

Die Klasse SOM ist die Basisklasse der Komponente. In ihr finden sich sämtliche Algorithmen und Aufrufe der Strategen für Netzinstanziierung, Initialisierung, Trainingsfortschritt und Abbruchsprüfung statt.
Die Klasse verfügt über 2 Konstruktoren. Ein Konstruktor ist für die Anfangsinitialisierung der SOM zuständig, ihm wird ein DataDummy-Objekt übergeben. Er kann auch für Testdatenobjekte verwendet werden. Der andere Konstruktor verlangt die Übergabe eines Parser-Datenobjektes, über dessen Methoden die Daten aus einer Datenquelle geholt werden. Weitere Parameter der Konstruktoren sind die Trainingsparameter, aufgrund deren die Feldgrößen und Variablen der Klasse initialisiert werden. Folgender Quellcodeausschnitt zeigt den Konstruktor mit dem Parser-Objekt.

```
1  public SOM(
2          Parser parser,
3          int neuronenAnzahlY,
4          int neuronenAnzahlX,
5          String somStruktur,
6          String somInitialisierungsart,
7          String initwert
8          ){
9    this.inputSet = parser.getInputPool();
10   this.labelListe = parser.getLabelListe();
11   this.anzahlEingabeNeuronen = inputSet[0].length;
12   this.eingabemuster = new double[inputSet[0].length];
13   this.initArt = somInitialisierungsart;
14   this.initWert = Double.parseDouble(initwert);
15   this.strukturArt = somStruktur;
```

```
16    this.nachbarFunktion = "Gauss";
17    this.nbwerte = new double[neuronenAnzahlX][neuronenAnzahlY];
18    this.runner = new Thread(this);
19    this.w = new double[neuronenAnzahlX]
20                       [neuronenAnzahlY]
21                       [inputSet[0].length];
22    this.siegerInputIndex = new int[inputSet.length][2];
23    this.errors = new double[inputSet.length];
24    for(int i=0;i<errors.length;i++){
25      errors[i]=999999.999999;
26    }
27  }
```

Nach der Erzeugung eines SOM-Objektes mit dem Konstruktor kann durch Aufruf der Methode „start“ der Thread der Klasse gestartet werden, da SOM die Schnittstelle „Runnable“ implementiert. Nach „start“ wird automatisch die Methode „run“ aufgerufen, welche mit einem Aufruf der Methode „hauptroutine“ überschrieben wurde.
In der Methode „hauptroutine“ erfolgt der Aufruf der Methode „initSOM“. Sie ermittelt und initialisiert den passenden InitStrategen. Anschließend ruft sie dessen Servicemethode „initialisiereSOM“ auf.

```
1   private void hauptroutine(){
2     runner = Thread.currentThread();
3     ran = new Random();
4     initSOM();
5     ...
6   }
7
8   private void initSOM() {
9     SOM_Initialisierer initialisierer;
10    if(this.initArt.equals("äzufllig")){
11      initialisierer = new Random_Initialisierer();
12    }else{
13      initialisierer = new Unifom_Initialisierer();
14    }
15    initialisierer.initialisiereSOM(this);
```

```
16 }
```

Nach der Initialisierung erfolgt eine erste Benachrichtigung der Observer der Klasse. Dazu erfolgt ein Update der Daten im Objekt „sommatrix“. Dieses Objekt fungiert als Transportklasse für die Daten. Sie wird an die Observer übermittelt, z.B. an die Perspektivfenster, welche die Daten wieder aus dem Objekt holen.

```
1 private void hauptroutine(){
2   ...
3   sommatrix.setAktLernrate(this.getLernrate());
4   sommatrix.setLabelListe(labelListe);
5   sommatrix.setW(w);
6   this.setChanged();
7   this.notifyObservers(this.sommatrix);
8   ...
9 }
```

Als nächstes erfolgt die Auswahl des gewünschten AbbruchStrategen. Dies erfolgt, indem der Inhalt der Variable „abruchArt“ getestet wird und ein, auf den Wert passender AbbruchStrategen instantziiert wird.
Danach folgt eine Programmschleife, die solange durchlaufen wird, bis die Abbruchbedingung durch den AbruchStrategen bestätigt wird. In ihr wird ein Inputmuster aus dem InputPool zufällig ausgewählt und sein Inhalt in ein 1-dimensionales double-Array „eingabemuster“ kopiert. Anschließend wird die Methode „Lernschritt“, welche für alle Tätigkeiten zur Ausführung eines Lernschritts verantwortlich ist, aufgerufen und der Lernschrittzähler um 1 erhöht.

```
1 private void hauptroutine(){
2   ...
3   do{
4     try{
5       this.aktMusterNr = ran.nextInt(inputSet.length);
6       for( int c=0 ; c<eingabemuster.length ; c++){
7         eingabemuster[c]=inputSet[this.aktMusterNr][c];
8       }
9       lernSchritt(counter);
10      counter++;
```

```
11      }catch(Exception e){
12      }
13      this.abruchtester.setParameterUpdate(this);
14    }while(!this.abruchtester.abbruchBedingungErfuellt(this));
15    ...
16  }
```

Zum Abschluss der Methode „hauptroutine“ erfolgt nochmals eine Benachrichtigung der Observer.

Wie oben bereits erwähnt wurde, ist die Methode „lernschritt“ für die Abarbeitung eines einzigen Lernschrittes inklusive der Aufrufe der Strategen für spezielle Aufgaben zuständig und soll nun beschrieben werden. Am Anfang holt sich die Methode die Nummer des aktuellen Lernschritts aus dem Parameter Counter. Danach erzeugt sie eine Instanz der Klasse„SiegerNeuron“ und ruft die Methoden „erstelleNachbarStratege“ und „erstelleLernrateModifikator“, welche für die Erzeugung passender Nachbar- und LernrateStrategen zuständig sind. Auf eine spezielle Methode für die Erzeugung eines ErregungsStrategen wurde verzichtet, da bisher nur die konkrete Klasse „Erst-Sieger_ErregungsErmittler“ existiert. Um der Oberfläche genügend Zeit für eine Aktualisierung der Oberfläche zu geben, wird der Thread für einige Zeit pausiert und danach selbstständig fortgesetzt. Über die Servicemethode „ermittleSiegerNeuron“ des ErregungsStrategen wird danach das Siegerneuron ermittelt, seine Position in der SOM in der Instanz des SiegerNeurons abgelegt und der SiegerIndex aktualisiert.

```
1  private void lernSchritt(int counter)
2          throws InterruptedException {
3    int lernSchrittNr = counter;
4    siegerneuron = new SiegerNeuron();
5    NachbarStratege nbs = this.erstelleNachbarStratege();
6    Lernrate_Modifikator lrm = this.erstelleLernrateModifikator();
7    eErmittler = new ErstSieger_ErregungsErmittler();
8    runner.sleep(100);
9    SiegerNeuron sn
10     = eErmittler.ermittleSiegerNeuron(w,eingabemuster);
11   siegerneuron.setPosition(sn.positionX, sn.positionY);
12   this.siegerInputIndex[this.aktMusterNr][0]
```

```
13      = this.siegerneuron.getPositionX();
14    this.siegerInputIndex[this.aktMusterNr][1]
15      = this.siegerneuron.getPositionY();
16    ...
```

Als nächstes folgt die Ermittlung der Nachbarschaftswerte durch den NachbarStrategen für alle Kartenneuronen. Die Werte werden im Array „nbwerte“ gespeichert. Anschliessend erfolgt die Speicherung der Nachbarwerte und des neuen SiegerIndex in dem Objekt sommatrix (siehe oben). Dies erfolgt über die Servicemethode „getNachbarWert“ des NachbarStrategen. Nun folgt die Berechnung des Fehlers zwischen dem aktuellen Muster und seinem Gewinnerneuron durch die Methode „berechneQError“. Durch Aufruf der Methode „berechneGewichtsanpassung“ werden die Verbindungsgewichte des Neuronennetzes aktualisiert. In ihr wird durch alle Eingangsverbindungen (i) aller Kartenneuronen an den Positionen (zeile = z, spalte = s) gegangen und das Gewicht aktualisiert.

```
1  private void lernSchritt(int counter)
2    ...
3    for(int zeile=0; zeile < w.length;zeile++){
4      for(int spalte=0; spalte < w[0].length; spalte++){
5        nbwerte[zeile][spalte]
6          =nbs.getNachbarWert(siegerneuron, zeile, spalte, radius);
7      }
8    }
9    sommatrix.setNbwerte(nbwerte);
10   sommatrix.setSiegerIndex(this.siegerInputIndex);
11   this.errors[this.aktMusterNr]=this.berechneQError();
12   berechneGewichtsanpassung();
13   ...
14 }
15
16 public double berechneQError(){
17   double error = 0.0;
18   double[] inputmuster = this.getEingabemuster();
19   double[] siegervector = new double[inputmuster.length];
20   for(int i=0; i<inputmuster.length;i++){
21     siegervector[i]      =
```

```
22        this.w[this.getSiegerneuron().getPositionX()]
23              [this.getSiegerneuron().getPositionY()]
24              [i];
25    }
26    for(int i=0; i < inputmuster.length; i++){
27      error = error +
28               (inputmuster[i]-siegervector[i])*
29               (inputmuster[i]-siegervector[i]);
30    }
31    error = Math.sqrt(error);
32    return error;
33  }
34
35  private void berechneGewichtsanpassung() {
36    for(int z=0; z < w.length;z++){
37      for(int s=0; s < w[0].length; s++){
38        for(int i=0;i<w[0][0].length;i++){
39          w[z][s][i] = w[z][s][i] + (lernrate * nbwerte[z][s]
40            * (eingabemuster[i] - w[z][s][i]));
41        }
42      }
43    }
44  }
```

Nun erfolgt die neue Berechnung der Lernrate über die Servicemethode „lernrateUpdate“ des Lernrate_Modifikators und anschließend die Reduktion des Erregungsradius. Dies findet im Programm linear statt und wurde so umgesetzt, dass der Reduktionsfaktor vom aktuellen Radius abgezogen wird, solange der Radius größer als 1 ist.
Die Werte für das aktuelle Muster, neue Lernrate, neuen Radius, Abbruchparameter, Labelliste und die Verbindungsgewichte werden nun im Objekt „sommatrix“ abgelegt. Ist die aktuelle Lernschrittzahl durch die gewünschte Anzahl der Lernschritte zwischen den Oberflächenupdates ohne Rest teilbar, erfolgt eine Benachrichtigung der Observer.

Klasse SiegerNeuron

Paket: de.fhb.schroesv.somarff.businesslogic.som

Die Klasse dient als Datencontainer zur Speicherung der Position eines Kartenneurons.

Klasse SOMMatrix

Paket: de.fhb.schroesv.somarff.businesslogic.som

Die Klasse wird als Datentransportobjekt benutzt und dient als eine Art „Briefumschlag“ um Objektdaten von der SOM an die Observer weiterzuleiten.

5.1.3 InitStrategen

Paket: de.fhb.schroesv.somarff.businesslogic.som.init_strategen

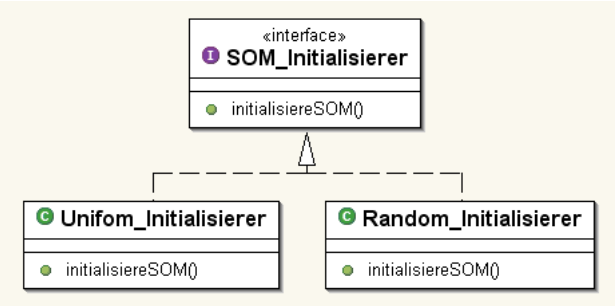

Abbildung 5.4: Klassendiagramm InitStrategen

Klasse SOM_Initialisierer

Die Klassen der InitialisierungsStrategen implementieren die Schnittstelle „SOM_Initialisierer“, wodurch sie sich dazu verpflichten, die Methode „initialisiereSOM“ zu implementieren. Die Methode dient der Belegung der Ver-

bindungsgewichte der Kartenneuronen zu den Eingangsneuronen mit Anfangswerten.

Klasse Random_Initialisierer

Paket: de.fhb.schroesv.somarff.businesslogic.som.init_strategen

Die Klasse initialisiert die Verbindungen der Kartenneuronen zu den Eingangsverbindungen mit einem Startwert. Dieser Wert ist ein Zufallswert und liegt im Bereich $[0, maxwert]$, wobei $maxwert$ ein vom Benutzer vorgegebener Wert ist.

Klasse Uniform_Initialisierer

Paket: de.fhb.schroesv.somarff.businesslogic.som.init_strategen

Die Klasse initialisiert die Verbindungen der Kartenneuronen zu den Eingangsverbindungen mit einem Startwert. Dieser Wert ist einheitlich für alle Neuronen gleich und wird vom Benutzer vorgegeben.

5.1.4 NormalisierungsStrategen

Paket: de.fhb.schroesv.somarff.businesslogic.som.norm_strategen

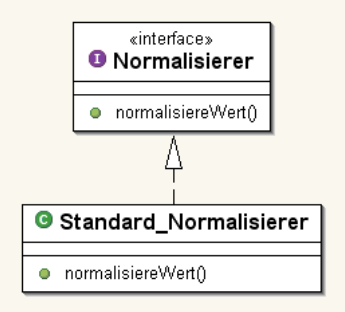

Abbildung 5.5: Klassendiagramm NormalisierungsStrategen

Klasse Normalisierer

Die Klassen der NormalisierungsStrategen implementieren die Schnittstelle „Normalisierer“, wodurch sie sich dazu verpflichten, die Methode „normalisiereWert“ zu implementieren. Die Methode dient der Bildung der Repräsentation eines Wertes aus einem Datenbereich $[-max, max]$ in einen gewünschten Wertebereich.

Klasse Standard_Normalisierer

Paket: de.fhb.schroesv.somarff.businesslogic.som.norm_strategen

Die Klasse bietet den Service der Repräsentation eines Wertes aus einem Datenbereich $[-max, max]$ in einen gewünschten Wertebereich von $[-1, 1]$ über die Methode „normalisiereWert“.

5.1.5 AbbruchStrategen

Paket: de.fhb.schroesv.somarff.businesslogic.som.abbruch_strategen

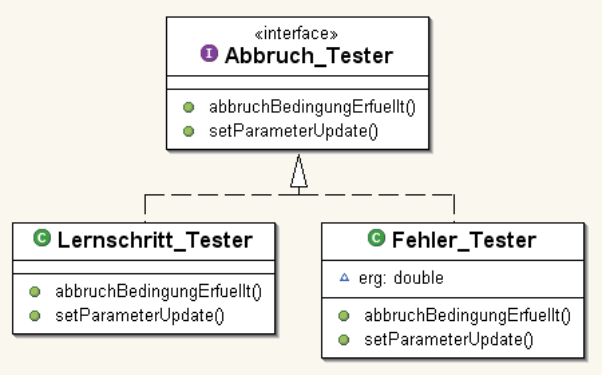

Abbildung 5.6: Klassendiagramm AbbruchStrategen

Klasse Abbruch_Tester

Die Klassen der AbbruchStrategen implementieren die Schnittstelle „Abbruch_Tester“, wodurch sie sich dazu verpflichten, die Methoden „abbruchBedingungErfuellt“ und „setParameterUpdate“ zu implementieren. Die Methdoden dienen der eigentlichen Prüfungen der Abbruchbedingung und der Aktualisierung der Testparameter in der aufrufenden SOM.

Klasse Lernschritt_Tester

Paket: de.fhb.schroesv.somarff.businesslogic.som.abbruch_strategen

Die Klasse prüft die Abbruchbedingung, indem sie die Werte für Lernrate und für noch zu vollziehende Lernschritte auf positive Werte über 0 testet. Ist ein Wert 0 oder kleiner, wird die Abbruchbedingung bestätigt. Dies geschieht in der Methode „abbruchBedingungErfuellt“.

```
1  public boolean abbruchBedingungErfuellt(SOM som) {
2    boolean ok = false;
3    if(som.getLernrate()<=0){
4      ok = true;
5    }
6    if(som.getAbbruchParameter()<=0){
7      ok=true;
8    }
9    return ok;
10 }
```

Die Methode „setParameterUpdate“ überschreibt den Wert in der SOM für noch zu vollziehende Lernschritte mit einem, um 1 reduzierten, Wert.

Klasse Fehler_Tester

Paket: de.fhb.schroesv.somarff.businesslogic.som.abbruch_strategen

Die Klasse prüft die Abbruchbedingung, indem sie den Wert für Lernrate

auf einen positiven Wert über 0 testet. Desweiteren ermittelt sie den Durchschnitt über alle aktuellen Quantisierungsfehler der Muster. Ist *Lernrate* <= 0 oder ist der $MSE <= x$, wobei x ein gewünschter Wert ist, wird die Abbruchbedingung bestätigt.

```
1  public boolean abbruchBedingungErfuellt(SOM som) {
2    double testwert = som.getAbbruchParameter();
3    boolean ok = false;
4    double summe = 0.0;
5    for(int i=0;i<som.getErrors().length;i++){
6      summe =summe+som.getErrors()[i];
7    }
8    erg = summe/som.getErrors().length;
9    if((erg<testwert)||(som.getLernrate()<=0.0)){
10     ok = true;
11     som.setQError(erg);
12   }
13   return ok;
14 }
```

Die Methode „setParameterUpdate“ überschreibt den Wert für den durchschnittlichen Quantisierungsfehler mit dem berechneten Wert.

5.1.6 LernrateStrategen

Paket: de.fhb.schroesv.somarff.businesslogic.som.lernrate_strategen

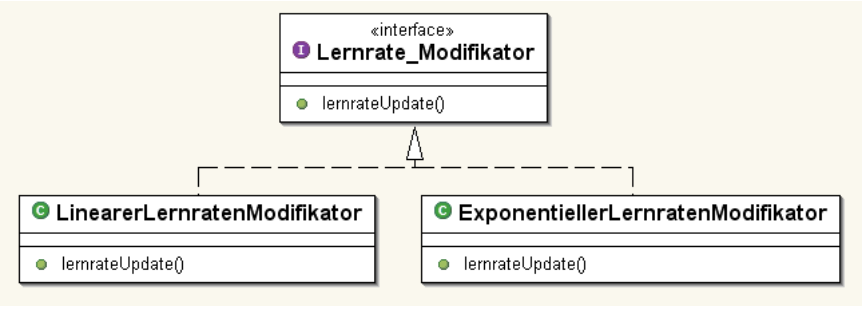

Abbildung 5.7: Klassendiagramm LernrateStrategen

Klasse Lernrate_Modifikator

Die Klassen der LernrateStrategen implementieren die Schnittstelle „Lernrate_Modifikator“, wodurch sie sich dazu verpflichten, die Methode „lernrateUpdate“ zu implementieren. Die Methdode berechnet die Anpassung der Lernrate aufgrund einer ihnen eigenen mathematischen Funktion.

Klasse LinearerLernratenModifikator

Paket: de.fhb.schroesv.somarff.businesslogic.som.lernrate_strategen

Die Klasse ermittelt die an den Trainingfortschritt angepasste Lernrate. Sie verwendet dafür folgende lineare Funktion.

$$L_n = L_0 - n * LF \tag{5.1}$$

Wie man sieht, errechnet sich die Lernrate L_n des Lernschrittes n, in dem ein bestimmter Lernfaktor LF mit der Anzahl der Lernschritte n multipliziert, und das Produkt von der Start-Lernrate L_0 substrahiert wird. Der LF wird vom Benutzer über das GUI vorgegeben.

Klasse ExponentiellerLernratenModifikator

Paket: de.fhb.schroesv.somarff.businesslogic.som.lernrate_strategen

Die Klasse ermittelt die an den Trainingfortschritt angepasste Lernrate. Sie verwendet dafür folgende exponentielle Funktion.

$$L_n = L_0 * LF^n \tag{5.2}$$

Wie man sieht, errechnet sich die Lernrate L_n für den Lernschritt n, indem ein bestimmter Lernfaktor LF mit der Anzahl der Lernschritte n potenziert und das Ergebnis mit der Start-Lernrate L_0 multipliziert wird. Der Lernfaktor wird vom Benutzer über das GUI vorgegeben.

5.1.7 NachbarStrategen

Paket: de.fhb.schroesv.somarff.businesslogic.som.nb_strategen

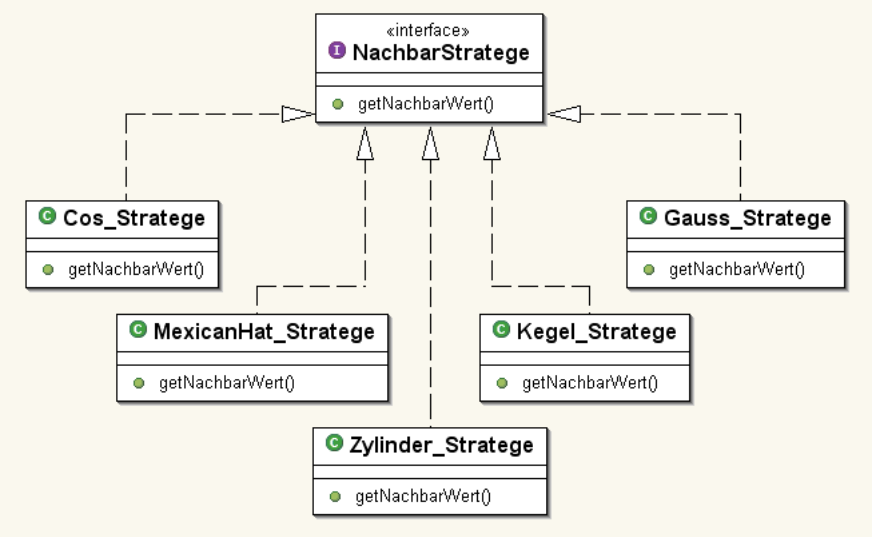

Abbildung 5.8: Klassendiagramm NachbarStrategen

Klasse NachbarStratege

Paket: de.fhb.schroesv.somarff.businesslogic.som.nb_strategen

Die Klassen der NachbarStrategen implementieren die Schnittstelle „NachbarStratege“, wodurch sie sich dazu verpflichten, die Methode „getNachbarWert“ zu implementieren. Die Methode berechnet den Nachbarwert zwischen dem Siegerneuron und einem Kartenneuron auf Grundlage einer mathematischen Funktion.

Klasse Cos_Strategie

Paket: de.fhb.schroesv.somarff.businesslogic.som.nb_strategen

Die Klasse ermittelt den Nachbarwert zwischen dem Siegerneuron und einem Kartenneuron auf Grundlage der Cosinus-Funktion.

```
1  public double getNachbarWert(
2          SiegerNeuron siegerneuron,
3          int positionX,
4          int positionY,
5          double radius){
6    double abstandX = siegerneuron.getPositionX()-positionX;
7    double abstandY = siegerneuron.getPositionY()-positionY;
8    double abstand;
9
10   abstandX = abstandX*abstandX;
11   abstandY = abstandY*abstandY;
12   abstand = Math.sqrt(abstandX+abstandY);
13
14   double erg;
15   if(abstand<=radius){
16     erg = Math.cos((abstand/radius)*(Math.PI/2));
17   }else{
18     erg = 0.0;
19   }
20   return erg;
21 }
```

Klasse Gauss_Strategе

Paket: de.fhb.schroesv.somarff.businesslogic.som.nb_strategen

Die Klasse ermittelt den Nachbarwert zwischen dem Siegerneuron und einem Kartenneuron auf Grundlage der Gauss-Funktion.

```
1  public double getNachbarWert(
2          SiegerNeuron siegerneuron,
3          int positionX,
4          int positionY,
5          double radius){
6    double abstandX=siegerneuron.getPositionX()-positionX;
7    double abstandY=siegerneuron.getPositionY()-positionY;
```

```
8    double abstand;
9
10   abstandX = abstandX * abstandX;
11   abstandY = abstandY * abstandY;
12   abstand = Math.sqrt(abstandX + abstandY);
13
14   return Math.exp(-1*(abstand/radius)*(abstand/radius));
15 }
```

Klasse Kegel_Stratege

Paket: de.fhb.schroesv.somarff.businesslogic.som.nb_strategen

Die Klasse ermittelt den Nachbarwert zwischen dem Siegerneuron und einem Kartenneuron auf Grundlage der Kegel-Funktion.

```
1  public double getNachbarWert(
2          SiegerNeuron siegerneuron,
3          int positionX,
4          int positionY,
5          double radius){
6    double abstandX = siegerneuron.getPositionX()-positionX;
7    double abstandY = siegerneuron.getPositionY()-positionY;
8    double abstand;
9
10   abstandX = abstandX*abstandX;
11   abstandY = abstandY*abstandY;
12   abstand = Math.sqrt(abstandX+abstandY);
13
14   if(abstand<radius){
15     return (1-(abstand/radius));
16   }else{
17     return 0;
18   }
19 }
```

Klasse MexicanHat_Stratege

Paket: de.fhb.schroesv.somarff.businesslogic.som.nb_strategen

Die Klasse ermittelt den Nachbarwert zwischen dem Siegerneuron und einem Kartenneuron auf Grundlage der MexicanHat-Funktion.

```
1  public double getNachbarWert(
2          SiegerNeuron siegerneuron,
3          int positionX,
4          int positionY,
5          double radius){
6    double abstandX = siegerneuron.getPositionX()-positionX;
7    double abstandY = siegerneuron.getPositionY()-positionY;
8    double abstand;
9    double d,z,element1,element2;
10
11   abstandX = abstandX*abstandX;
12   abstandY = abstandY*abstandY;
13   abstand = Math.sqrt(abstandX+abstandY);
14
15   double erg;
16   z = abstand * abstand;
17   d = radius * radius;
18   element1 = 1-(z/d);
19   element2 = (z/d)*(-2);
20   erg = element1 * (Math.exp(element2));
21   return erg;
22 }
```

Klasse Zylinder_Stratege

Paket: de.fhb.schroesv.somarff.businesslogic.som.nb_strategen

Die Klasse ermittelt den Nachbarwert zwischen dem Siegerneuron und einem Kartenneuron auf Grundlage der Zylinder-Funktion.

```
1  public double getNachbarWert(
2          SiegerNeuron siegerneuron,
```

```
3           int positionX,
4           int positionY,
5           double radius){
6     double abstandX = siegerneuron.getPositionX()-positionX;
7     double abstandY = siegerneuron.getPositionY()-positionY;
8     double abstand;
9     double d,z,element1,element2;
10
11    abstandX = abstandX*abstandX;
12    abstandY = abstandY*abstandY;
13    abstand = Math.sqrt(abstandX+abstandY);
14
15    double erg;
16    if(abstand<=radius){
17      erg = 1.0;
18    }else{
19      erg = 0.0;
20    }
21    return erg;
22  }
```

5.1.8 ErregungsStrategen

Paket: de.fhb.schroesv.somarff.businesslogic.som.erreg_strategen

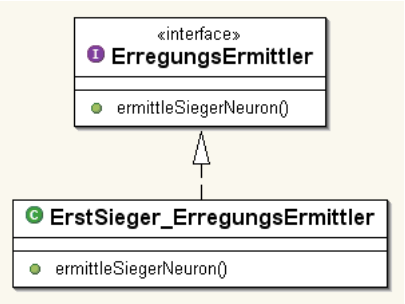

Abbildung 5.9: Klassendiagramm ErregungsStrategen

Klasse ErregungsErmittler

Paket: de.fhb.schroesv.somarff.businesslogic.som.erreg_strategen

Die Klassen der ErregungsStrategen implementieren die Schnittstelle „ErregungsErmittler“, wodurch sie sich dazu verpflichten, die Methode „ermittleSiegerNeuron“ zu implementieren. Die Methdode berechnet die Euklidische Distanz für jedes KartenNeuron und gibt das Neuron, welches dem Eingabevektor am nahesten, also am ähnlichsten ist, zurück.

Klasse ErstSieger_ErregungsErmittler

Paket: de.fhb.schroesv.somarff.businesslogic.som.erreg_strategen

Die Klasse ermittelt das SiegerNeuron in der Methode „ermittleSiegerNeuron“. Dabei geht sie durch alle Neuronen des Neuronalen Netzes und ruft die private Methode „berechneEuklidischenAbstand“ auf. Das Ergebnis wird in einem Feld zwischengespeichert und an die private Methode „ermittleSieger“ übergeben. Diese Methode ermittelt den minimalsten Wert des Feldes. Besitzen den minimalsten Abstand mehrere Neuronen, so wird das Neuron als SiegerNeuron bezeichnet, bei dem der Wert als erstes auftrat. Darauf weist auch der Name der Klasse hin.

```
1  public SiegerNeuron ermittleSiegerNeuron(
2          double[][][][] somMatrix,
3          double [] inputMuster) {
4    double abstaende[][]
5      = new double[somMatrix.length][somMatrix[0].length];
6                  this.getOutInputMuster(inputMuster);
7    for(int zeilenCounter=0;
8        zeilenCounter<somMatrix.length;
9        zeilenCounter++){
10     for(int spaltenCounter=0;
11         spaltenCounter<somMatrix[0].length;
12         spaltenCounter++){
13       abstaende[zeilenCounter][spaltenCounter]
```

```
14          = berechneEuklidischenAbstand(somMatrix,
15                                        inputMuster,
16                                        zeilenCounter,
17                                        spaltenCounter);
18      }
19    }
20    return ermittleSieger(abstaende);
21  }
22
23  private double berechneEuklidischenAbstand(
24          double[][][][] somMatrix,
25          double[] inputMuster,
26          int zeilenCounter,
27          int spaltenCounter){
28
29    double schicht[] = new double[inputMuster.length];
30    double difsum = 0;
31
32    for(int inputNeuron=0;
33            inputNeuron<somMatrix[0][0].length;
34            inputNeuron++){
35      schicht[inputNeuron]
36        = inputMuster[inputNeuron] -
37          somMatrix[zeilenCounter][spaltenCounter][inputNeuron];
38      schicht[inputNeuron]
39        = schicht[inputNeuron]*schicht[inputNeuron];
40    }
41    for(int s=0;s<schicht.length;s++){
42      difsum = difsum + schicht[s];
43    }
44    return Math.sqrt(difsum);
45  }
46
47  private SiegerNeuron ermittleSieger(double[][][] dif) {
48    SiegerNeuron siegerneuron = new SiegerNeuron();
49          double min = 999999999;
50
51    for(int zeilenCounter=0;
52        zeilenCounter<dif.length;
```

```
53          zeilenCounter++){
54        for(int spaltenCounter=0;
55            spaltenCounter<dif[0].length;
56            spaltenCounter++){
57          if(dif[zeilenCounter][spaltenCounter]<min){
58            min = dif[zeilenCounter][spaltenCounter];
59            siegerneuron.setPosition(zeilenCounter,spaltenCounter);
60          }
61        }
62      }
63      return siegerneuron;
64  }
```

Die Methode „setParameterUpdate“ überschreibt den Wert in der SOM für noch zu vollziehende Lernschritte mit einem, um 1 reduzierten, Wert.

5.1.9 GUI

Die Präsentationsschicht (GUI) wurde in 3 Pakete aufgeteilt.

controller In diessem Paket findet sich nur die Controller-Klasse. Sie ist für die Kommunikation der Oberflächenelemente mit der Anwendungsschicht verantwortlich.

sichten In diesem Paket befinden sich alle Perspektiv-Fenster mit ihren Steuer- und Anzeigepanelen.

terminal In diesem Paket finden sich die Panele und Konsolen, die für die Steuerung und Parametrisierung der SOM zuständig sind sowie das Hauptfenster der Anwendung mit der „main“-Methode.

Im folgenden werden die in den einzelnen Paketen enthaltenen Klassen beschrieben. Der Autor verweist zusätzlich auf die javadoc-Dokumentation, welche der Software beiliegt.

Klasse GUI_Controller

Paket: de.fhb.schroesv.somarff.gui.controller

Die Klasse GUI_Controller nimmt die, von den Oberflächenelementen erzeugten, Ereignisse und verarbeitet sie entsprechend. Bei Aufruf der „init“-Methode holt sich der Controller das aufrufende Hauptfenster und mit diesem auch das Parser-Objekt.
Da die Klasse die Schnittstelle „ActionListener“ implementiert, verfügt sie über die Methode „actionPerformed(ActionEvent event)“. In dieser Methode werden die empfangenden Ereignisse verarbeitet. Dazu wird „getActionCommand“ des Ereignisses aufgerufen und das Ergebnis auf eine bestimmte Zeichenkette überprüft. Folgender Code zeigt ein Beispiel.

```
1  public void actionPerformed(ActionEvent event){
2    ...
3    if(event.getActionCommand().equals("Einstellungen")){
4      zeigeChild(hauptfenster.parserTerminal);
5    }
6    if(event.getActionCommand().equals("Datenblatt")){
7      zeigeChild(hauptfenster.parserGUI);
8    }
9    if(event.getActionCommand().equals("Lernparameter")){
10     zeigeChild(hauptfenster.lernparameter);
11   }
12   ...
13   if(event.getActionCommand().equals("Stop")){
14     hauptfenster.theSOM.stop();
15     hauptfenster.getPControl().setStatus(3);
16     hauptfenster.setGUISperre(false);
17   }
18   ...
19 }
```

Wird erfolgreich auf „Datei auswählen“ geprüft, so wird ein File-Auswahl-Dialog erzeugt und angezeigt. Der Name und der Pfad der vom Benutzer gewählten Datei werden dem Parser durch Aufruf der Methode „lookFile“ übergeben. Anschließend werden die Daten des ParserGUIs aktualisiert. Danach wird die Nummer der Labelspalte aus dem Einstellungen-Panel des Parsers ausgelesen und an den Parser geleitet. Anschließend wird die Codierung des InputPools mit „fuelleInputPool“-Methode des Parser-Objektes

aufgerufen. Die Parameter sind die Einstellungen für die einzelnen Datentypen, welche vom Benutzer im Parser-Einstellungen-Panel getätigt wurden. Zum Schluss wird der InputPool vom Parser geholt.
Bei Zeichenkette „Einstellungen“ wird das Einstellungen-Panel für den Parser durch Aufruf von „zeigeChild“ ein-/ausgeblendet.
Ist die Zeichenkette „Datenblatt“ wird die Datenblatt-Ansicht durch Aufruf von „zeigeChild“ ein-/ausgeblendet.
Für das Panel der Lernparameter wird dies durch die Zeichenkette „Lernparameter“ durchgeführt.
Zeichenkette „Netzparameter“ sorgt für die Ein-/Ausblendung des Panels der Netzparameter.
Bei den Zeichenketten „Zahlen-Matrix“, „Gitter-Matrix“, „Komponenten-Matrix“, „Distanz-Matrix“, „Gewichts-Matrix“, „Gewinner-Matrix“ und „U-Matrix“ wird in der Hauptfensterklasse die Methode zur Erzeugung einer entsprechenden Perspektive aufgerufen.
Die Zeichenkette„Start“ wird aus den Einstellungen im SOM-Panel eine neue SOM instantiiert und an das Hauptfenster übergeben. Danach werden die Lernparameter an die SOM übergeben und das Lernen mit start für das SOM-Objekt gestartet. Abschließend werden die Oberflächenelemente, welche während des Trainings nicht benutzt werden sollen, gesperrt. Dies erledigt die Methode „setGUISperre“ des Hauptfensters. „Pause“, „Weiter“ und „Stop“ verfahren ähnlich. Bei ihnen wird der SOM-Thread pausiert, fortgeführt oder vollständig gestoppt. Außerdem erfolgt bei „Weiter“ eine Aktualisierung der Lernparameter.

Klasse PerspektivenFabrik

Paket: de.fhb.schroesv.somarff.gui.terminal

Die Klasse repräsentiert die Schnittstelle um Objekte von Perspektiv-Fenstern auf die SOM zu erzeugen. Ihr liegt das Entwurfsmuster „Abstrakte Fabrik“ zugrunde.

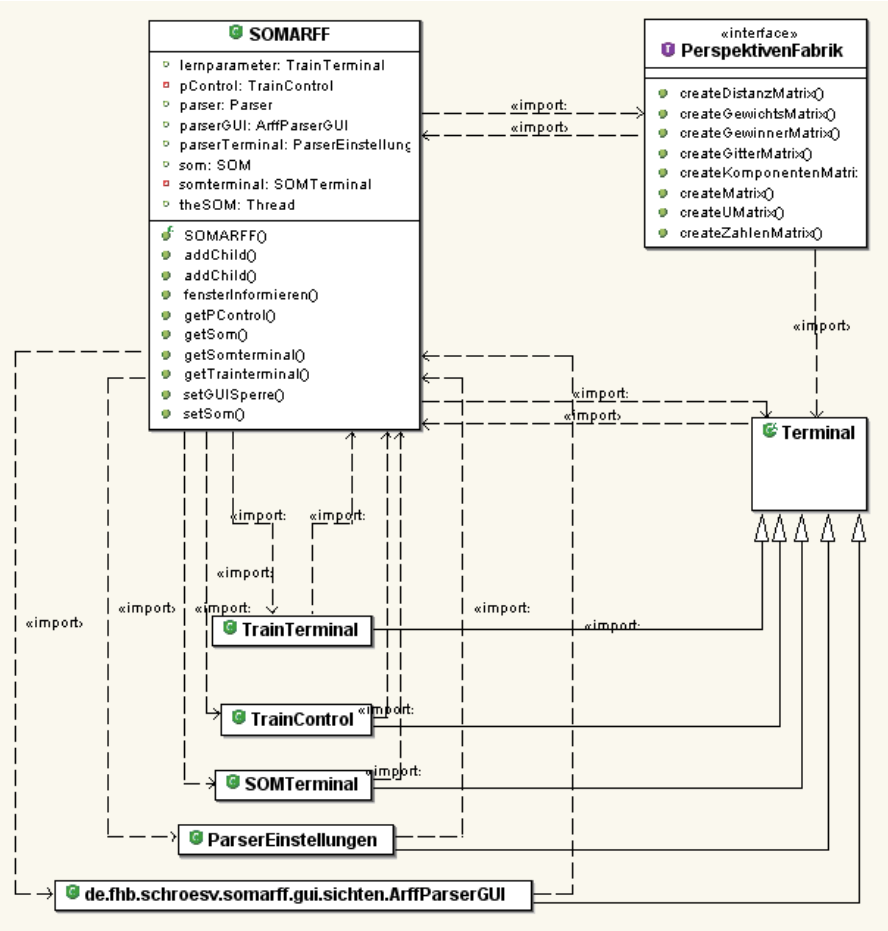

Abbildung 5.10: Klassendiagramm „de.fhb.schroesv.somarff.gui.terminal“ (vereinfacht)

Klasse GUI_Fabrik

Paket: de.fhb.schroesv.somarff.gui.terminal

Die Klasse implementiert die Schnittstelle „PerspektivenFabrik“ und ist als Dienstleister für Auswahl und Erzeugung der Perspektiv-Objekte zuständig. Sie wird in der Klasse „SOMARFF“ instanziiert und gibt durch Aufruf ihrer „createMatrix“-Methode die gewünschten Objekte zurück. Durch sie wurde der Entwurf in Bezug auf das Hinzufügen neuer Perspektiv-Klassen

optimiert. Zu Grunde liegt das Entwurfsmuster „Abstrakte Fabrik“.

Klasse ParserEinstellungen

Paket: de.fhb.schroesv.somarff.gui.terminal

Diese Klasse stellt ein Eingabeterminal für die Einstellungen des Parser auf dem Bildschirm dar. Es erweitert die Klasse Terminal.

Klasse SOMARFF

Paket: de.fhb.schroesv.somarff.gui.terminal

Die Klasse baut das Hauptfenster der Anwendung auf dem Bildschirm auf. Sie erbt von der Klasse „JFrame“. In ihr wird das Menü erstellt sowie die Perspektiven auf die SOM, Unterterminals für die Einstellungen des Parser, des Lernvorgangs und der Kohonen-Karte und die Konsole für die Lernzustände verwaltet. Die Ereignisse, welche aus dem Menü und den Steuerterminals kommen, werden an die Klasse „GUI-Controller“ weitergeleitet.

Klasse SOMTerminal

Paket: de.fhb.schroesv.somarff.gui.terminal

Diese Klasse erstellt ein Eingabeformular für die Eigenschaften der SOM. Sie erweitert die Klasse „Terminal“.

Klasse Terminal

Paket: de.fhb.schroesv.somarff.gui.terminal

Dies ist die Oberklasse aller Terminals und Perspektivfenster. Sie erweitert die Klasse „JInternalFrame“. Dadurch wird sie zum echten Unterfenster von „SOMARFF“, der Klasse, welche das Hauptfenster aufbaut. Desweiteren

wird die Schnittstelle „Observer“ implementiert, da für die Aktualisierung der Perspektiven das Entwurfsmuster „Observer“ verwendet wird.

Klasse TrainControl

Paket: de.fhb.schroesv.somarff.gui.terminal

Die Klasse ist für die Darstellung der Konsole für die Zustandsänderungen der SOM verantwortlich. Sie erbt von der Klasse Terminal.

Klasse TrainTerminal

Paket: de.fhb.schroesv.somarff.gui.terminal

Diese Klasse erstellt ein Eingabeformular für die Lernparameter der SOM. Sie erweitert die Klasse „Terminal“.

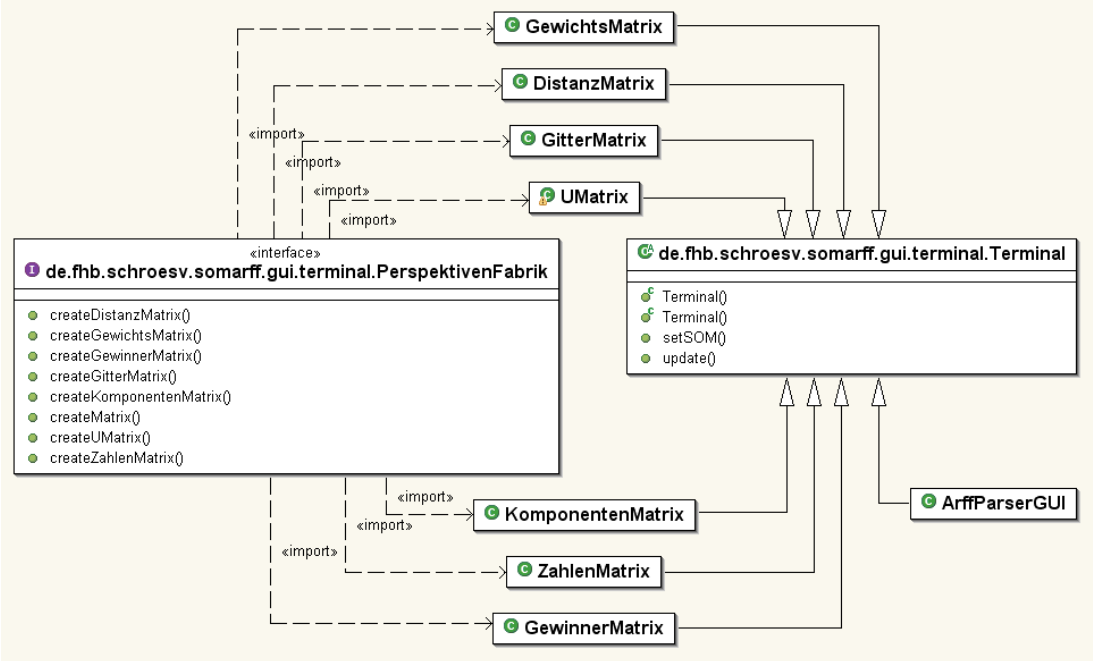

Abbildung 5.11: Klassendiagramm „de.fhb.schroesv.somarff.gui.sichten“ (vereinfacht)

Klasse ArffParserGUI

Paket: de.fhb.schroesv.somarff.gui.sichten

Die Klasse realisiert ein Unterfenster, in welchem die, vom Parser eingelesenen, Daten übersichtlich in einer Tabelle dargestellt sind. Sie erbt von der Klasse „Terminal“.

Klassen ZahlenMatrix, ZahlenMatrixAnzeigePanel, ZahlenMatrixSteuerPanel

Paket: de.fhb.schroesv.somarff.gui.sichten

Die Klasse „ZahlenMatrix“ realisiert ein Unterfenster zur Visualisierung der SOM in Form einer Matrix aus Zahlenwerten. Dazu wurde das Fenster in 2 Panele für Visualisierung (Klasse „ZahlenMatrixAnzeigePanel“) und Steuerung(Klasse „ZahlenMatrixSteuerPanel“) aufgeteilt.
Das Unterfenster wird durch das Observer-Entwurfmuster ständig mit aktualisierten Werten der Neuronengewichte versorgt und leitet Änderungen an das Anzeigepanel weiter.
Das Steuerpanel bietet Auswahlmöglichkeiten um die Aktivierung der Neuronen oder eine bestimmte Schicht der Neuronen mit der Wertangabe darzustellen. Je nach gewünschter Steuerungsoption durchläuft das Anzeigepanel die Neuronengewichte und stellt sie als Zahlenwerte im Panel dar. Dafür existieren 2 Methoden.
Methode „zeigeSchicht“ durchläuft die Verbindungsgewichte und stellt sie gerundet auf 3 Nachkommastellen an der entsprechenden Stelle im Neuronengitter dar. Je nach Auswahl im Steuerpanel kann dies auch normalisiert geschehen. Methode „zeigeAktivierungen“ läßt mit Aufruf der Methode „berechneAktivierung“ die Aktivierungen berechnen und in einem Array zurückgeben.

```
1 public double[][] berechneAktivierung(
2         double[][][][] somMatrix,
3         double[] inputMuster)
```

```
4           {
5     double schicht[]= new double[inputMuster.length];
6     double abstaende[][][]
7       = new double[somMatrix.length][somMatrix[0].length];
8     for(int zeilenCounter=0;
9         zeilenCounter<somMatrix.length;
10        zeilenCounter++){
11      for(int spaltenCounter=0;
12          spaltenCounter<somMatrix[0].length;
13          spaltenCounter++){
14        abstaende[zeilenCounter][spaltenCounter]
15          =berechneNetzeingabeFuerKartenNeuron(
16            somMatrix, inputMuster, schicht,
17            zeilenCounter, spaltenCounter);
18      }
19    }
20    return abstaende;
21  }
22
23  private double berechneNetzeingabeFuerKartenNeuron(
24          double[][][][] somMatrix,
25          double[] inputMuster,
26          double[] schicht,
27          int zeilenCounter,
28          int spaltenCounter)
29          {
30    double difsum=0;
31    for(int inputNeuron=0;
32        inputNeuron<somMatrix[0][0].length;
33        inputNeuron++){
34      schicht[inputNeuron]
35        =inputMuster[inputNeuron] -
36         somMatrix[zeilenCounter][spaltenCounter][inputNeuron];
37      schicht[inputNeuron]
38        =schicht[inputNeuron]*schicht[inputNeuron];
39    }
40    for(int s=0 ; s < schicht.length ; s++){
41      difsum = difsum + schicht[s];
42    }
```

```
43    return difsum;
44  }
```

Das Ergebnis wird dann an der entsprechenden Position im Gitter angezeigt. Auch diese Anzeige ist auf 3 Nachkommastellen gerundet und kann normalisiert erfolgen.

Klassen GitterMatrix, GitterMatrixAnzeigePanel, GitterMatrixSteuerPanel

Paket: de.fhb.schroesv.somarff.gui.sichten

Die Klasse „GitterMatrix“ realisiert ein Unterfenster zur Visualisierung der Entfernungsverhältnisse der Kartenneuronen und ihren direkten Nachbarneuronen im Neuronengitter in Form eines Netzes. Dazu wurde das Fenster in 2 Panele für Visualisierung (Klasse „GitterMatrixAnzeigePanel“) und Steuerung (Klasse „Gitter MatrixSteuerPanel“) aufgeteilt.
Das Unterfenster wird durch das Observer-Entwurfmuster ständig mit aktualisierten Nachbarwerten versorgt und leitet Änderungen an das Anzeigepanel weiter.
Das Anzeigepanel prüft die Dimension des Eingabevektors und ruft für einen 2-dimensionalen Vektor die Methode „draw2DMatrix“ auf. Diese Methode zeichnet in Abhängigkeit von der Netzdimension Verbindingslinien von Neuron zu Neuron. Die Lage eines Kartenneuron wird durch seinen Gewichtsvektor bestimmt. Zum Beispiel würde für ein Neuron mit einem Gewichtsvektor von $(1, 7)$ auf der x-Achse eines Koordinatensystems beim Wert 1 und auf der y-Achse beim Wert 7 liegen. Das Zeichnen auf einem Java-Graphics-Objekt unterscheidet sich von einem Zeichnen in einem Koordinatensystem nur darin, dass der Punkt $(0, 0)$ links oben liegt und die y-Achse nach unten ansteigt. Da dies für das Visualisierungsergebnis nicht weiter ausschlaggebend ist, wurden die Gewichtvektoren auf 1 normalisiert und dann als Zeichenvektoren auf dem Panel übernommern. Zusätzlich wurde ein offset-Wert addiert, um genügend Abstand zum Panelrand zu erhalten. Die Verbindungslinien zwischen

den Kartenneuronen folgen spaltenweise vom linken Neuron zum benachbarten rechten Neuron. Für die Zeilen fand die Zeichnungsrichtung vom oberen Neuron zum benachbarten unteren Neuron statt. Links, rechts, oben und unten beziehen sich dabei auf die Neuronenpositionen im Neuronengitter und nicht auf ihre Lage im Raum!

Klassen DistanzMatrix, DistanzMatrixAnzeigePanel, DistanzMatrixSteuerPanel

Paket: de.fhb.schroesv.somarff.gui.sichten

Die Klasse „DistanzMatrix“ realisiert ein Unterfenster zur Visualisierung der Neuronenaktivierung in Form einer Matrix aus Farbquadraten. Dazu wurde das Fenster in 2 Panele für Visualisierung (Klasse „DistanzMatrixAnzeigePanel“) und Steuerung(Klasse „DistanzMatrixSteuerPanel“) aufgeteilt.
Das Unterfenster wird durch das Observer-Entwurfmuster ständig mit aktualisierten Werten der Neuronengewichte versorgt und leitet Änderungen an das Anzeigepanel weiter.
Das Steuerpanel bietet Auswahlmöglichkeiten um die Farben der Neuronenerregung (schwach, mittel, stark) darzustellen. Dunkle Werte bedeuten dabei eine geringe oder keine Aktivierung, je mehr der Farbwert ansteigt, um so größer wird die Aktivierung. Zur Realisierung wurde eine Methode „berechneAktivierung“ implementiert. Sie ist identisch mit dem Algorithmus aus der Klasse „ZahlenMatrixAnzeigePanel“. Anschließend wurden die Werte normalisiert und mit dem Farbwert multipliziert.
Das Anzeigepanel durchläuft alle Neuronenaktivierungen und zeichnet für jedes Neuron ein Farbquadrat in der Malfarbe, welche dem Aktivierungswert entspricht. Die Größe der Kästchen kann über den Schalter im Steuerpanel reguliert werden.

Klassen KomponentenMatrix, KomponentenMatrixAnzeigePanel, KomponentenMatrixSteuerPanel

Paket: de.fhb.schroesv.somarff.gui.sichten

Die Klasse „KomponentenMatrix“ realisiert ein Unterfenster zur Visualisierung der Gewichtsverteilung nach Eingabeneuronen (Schichten) in Form einer Matrix aus Farbquadraten. Dazu wurde das Fenster in 2 Panele für Visualisierung (Klasse „KomponentenMatrixAnzeigePanel“) und Steuerung(Klasse „KomponentenMatrixSteuerPanel“) aufgeteilt.
Das Unterfenster wird durch das Observer-Entwurfmuster ständig mit aktualisierten Werten der Neuronengewichte versorgt und leitet Änderungen an das Anzeigepanel weiter.
Das Steuerpanel bietet Auswahlmöglichkeiten um die Farbe der Neuronenerregung darzustellen. Dunkle Werte bedeuten dabei ein geringes Gewicht, je mehr der Farbwert ansteigt, um so höher ist das Gewicht. Zur Realisierung wurden die Werte normalisiert und mit dem Farbwert multipliziert, stets für das Eingabeneuron, welches im Steuerpanel angegeben wurde.
Das Anzeigepanel durchläuft alle Verbindungsgewichte und zeichnet für jedes Neuron ein Farbquadrat in der Malfarbe, welche dem Gewichtswert entspricht. Die Größe der Kästchen kann über den Schalter im Steuerpanel reguliert werden.

Klassen GewichtsMatrix, GewichtsMatrixAnzeigePanel, GewichtsMatrixSteuerPanel

Paket: de.fhb.schroesv.somarff.gui.sichten

Die Klasse „GewichtsMatrix“ realisiert ein Unterfenster zur Visualisierung aller Synapsenverbindungen zwischen Kartenneuronen und Eingabeneuronen in Form einer Matrix aus Farbquadraten. Dazu wurde das Fenster in 2 Panele für Visualisierung (Klasse „GewichtsMatrixAnzeigePanel“) und Steuerung(Klasse „GewichtsMatrixSteuerPanel“) aufgeteilt.
Das Unterfenster wird durch das Observer-Entwurfmuster ständig mit ak-

tualisierten Werten der Neuronengewichte versorgt und leitet Änderungen an das Anzeigepanel weiter.
Das Steuerpanel bietet Auswahlmöglichkeiten um die Farbe der Neuronenerregung darzustellen. Dunkle Werte bedeuten dabei ein geringes Gewicht, je mehr der Farbwert ansteigt, um so höher ist das Gewicht. Zur Realisierung wurden die Werte normalisiert und mit dem Farbwert multipliziert.
Das Anzeigepanel durchläuft alle Verbindungsgewichte und zeichnet für jedes Neuron ein Farbquadrat in der Malfarbe, welche dem Gewichtswert entspricht. Die Matrix wurde so angelegt, dass die Verbindungen je Eingabeneuron von oben nach unten dargestellt werden. Also die Verbindungsgewichte des Eingabeneurons 0 stehen in der ersten Reihe, die für Eingabeneuron 1 in der zweiten Reihe und so weiter. Die Größe der Kästchen kann über Schalter im Steuerpanel reguliert werden.

Klassen GewinnerMatrix, GewinnerMatrixAnzeigePanel, GewinnerMatrixSteuerPanel

Paket: de.fhb.schroesv.somarff.gui.sichten

Die Klasse „GewinnerMatrix“ realisiert ein Unterfenster zur Visualisierung der Nachbarschaftswerte zwischen dem Siegerneuron und den umgebenen Kartenneuronen in Form einer Matrix aus Farbquadraten. Dazu wurde das Fenster in 2 Panele für Visualisierung (Klasse „GewinnerMatrixAnzeigePanel“) und Steuerung (Klasse „GewinnerMatrixSteuerPanel“) aufgeteilt.
Das Unterfenster wird durch das Observer-Entwurfmuster ständig mit aktualisierten Nachbarwerten versorgt und leitet Änderungen an das Anzeigepanel weiter.
Das Steuerpanel bietet die Auswahlmöglichkeit, die Nachbarschaftswerte in einer bestimmten Farbe darzustellen. Dunkle Werte bedeuten dabei einen geringen Nachbarschaftswert, je mehr der Farbwert ansteigt, um so höher ist der Nachbarschaftswert. Das Siegerneuron hat einen Nachbarschaftswert von 1, womit seine Malfarbe dem ausgewählten Wert entspricht. Die Farbwerte berechnen sich nach dem selben Algorithmus, wie aus DistanzMatrix,

KomponentenMatrix und GewichtsMatrix. Das Anzeigepanel durchläuft alle durch die SOM berechneten Nachbarschaftswerte und zeichnet für jedes Neuron ein Farbquadrat in der Malfarbe, welche dem Nachbarschaftswert entspricht. Die Farbquadrate für die Neuronen werden an ihre Positionen im Neuronengitter gesetzt. ´Die Größe der Kästchen kann über über Schalter im Steuerpanel reguliert werden.

Klassen UMatrix, UMatrixAnzeigePanel, UMatrixSteuerPanel

Paket: de.fhb.schroesv.somarff.gui.sichten

Die Klasse „UMatrix“ realisiert ein Unterfenster zur Visualisierung der Entfernungsverhältnisse der Neuronen und ihren direkten Nachbarneuronen im Neuronengitter in Form einer Matrix aus Farbquadraten. Dazu wurde das Fenster in 2 Panele für Visualisierung (Klasse „UMatrixAnzeigePanel“) und Steuerung (Klasse „UMatrixSteuerPanel“) aufgeteilt.
Das Unterfenster wird durch das Observer-Entwurfmuster ständig mit aktualisierten Nachbarwerten versorgt und leitet Änderungen an das Anzeigepanel weiter.
Diese Perspektive war in der Implementierung mit Abstand am aufwendigsten. Der Grund ist der, dass in der U-Matrix nicht die Darstellung als Neuron erfolgen kann, sondern lediglich die Verbindungsgewichte zwischen den Kartenneuronen farbig dargestellt werden. Der Autor hat hier einige Möglichkeiten entworfen und ausprobiert. Die letzlich implementierte Lösung wird folgend erklärt.
Um eine möglichst klare Abbildung zu bekommen, wurden die Bereiche um die Neuronen in 4 Quadranten unterteilt. Um dies zu erreichen, unterteilt man die Verbindungsstrecke zwischen den Neuronen in 2 Teile. Wenn also die Entfernung zwischen einem Neuron und einem anderen Neuron 6 beträgt, kann man sich die unterteilte Verbindung in 2 Teilstrecken, einer Verbindung von 6 vorstellen. Wenn dies für alle Neuronen zwischen den Spalten und Zeilen erfolgt ist, so bilden sich um die Neuronen 4 Quadrate, welche die Entfernungen zum jeweils nächsten Neuron angeben. Der Entfernungs-

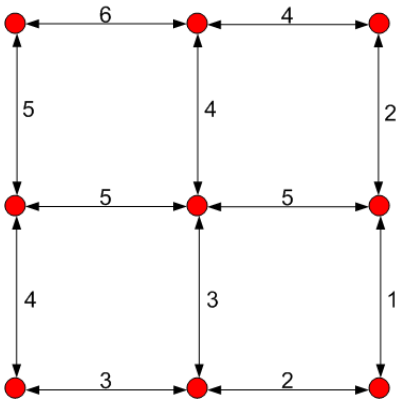

Abbildung 5.12: U-Matrix Beispiel: Neuronennetz mit Entfernungsangaben zwischen den Neuronen (Ausgangslage)

wert für ein Entfernungsquadrat ergibt sich aus dem Durchschnitt seiner Kantenentfernungen. Zum Beispiel hat ein Quadrat, welches die Kantenentfernungen 6 und 4 hat, einen Entfernungswert von 5, da $(6+4)/2 = 5$. Wird diese Methode auf alle Neuronen angewandt, so bilden sich gute Mischwerte zwischen den Neuronen, welche anschließend normalisiert werden. Auch die Berechnung des Farbwertes wurde für diese Matrix verbessert. Im Steuerpanel kann man 2 Farben für kurze und weite Entfernungen zwischen den Neuronen auswählen. Um die Malfarbe für ein Entfernungsquadrat zu ermitteln, wird die Methode „ermittleMalFarbe“ aufgerufen. Sie gibt ein Color-Objekt zurück, welches dem Entfernungswert entspricht. Die Methode normalisiert zuerst den Entfernungswert mit einem „Standard_Normalisierer“ aus dem Paket „de.fhb.schroesv.somarff.businesslogic.som.norm_strategen“. Anschließend ermittelt es die Distanzen für die einzelnen Farbanteile. Als Berechnungsgrundlage wurde das RGB-Farbmodell benutzt. Danach wird zu den Farbwerten der Farbe für kurze Distanz das Produkt aus dem Distanzwert des Farbanteils und dem normalisierten Entfernungswert hinzu addiert. Die berechneten neuen Farbanteile werden anschließend in eine Integerzahl gecastet und mit ihnen ein Color-Objekt erzeugt. Dieses wird dann als Rückgabewert ausgegeben.

```
1 private Color ermittleMalFarbe(double max, double entfernung){
```

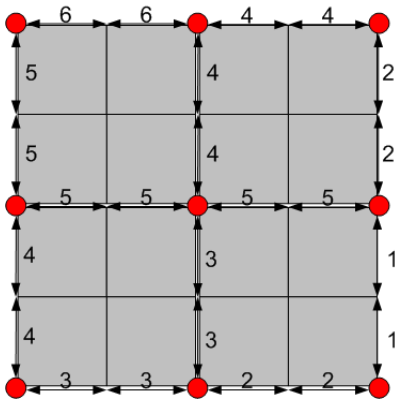

Abbildung 5.13: U-Matrix Beispiel: in Quandranten geteiltes Neuronennetz

```
2    entfernung = normalisierer.normalisiereWert(entfernung,max);
3
4    int distRed =
5      this.cBigDistanz.getRed() - this.cShortDistanz.getRed();
6    int distGreen =
7      this.cBigDistanz.getGreen() - this.cShortDistanz.getGreen();
8    int distBlue =
9      this.cBigDistanz.getBlue() - this.cShortDistanz.getBlue();
10
11   int red =
12     (int)(this.cShortDistanz.getRed() + distRed * entfernung);
13   int green =
14     (int)(this.cShortDistanz.getGreen() + distGreen * entfernung);
15   int blue =
16     (int)(this.cShortDistanz.getBlue() + distBlue * entfernung);
17
18   return (new Color(red, green, blue));
19 }
```

Die Matrix unterstützt das Labeln der Muster über ihre Gewinnerneuronen. Dazu wird die Labelliste und der Siegerindex aus dem Datenobjekt, welches die SOM während des Lernfortschritts sendet, gezogen und der in der Labelliste enthalte String an die entsprechender Position im Neuronennetz eingezeichnet. Dazu kann die zum labeln zu verwendende Spalte in den

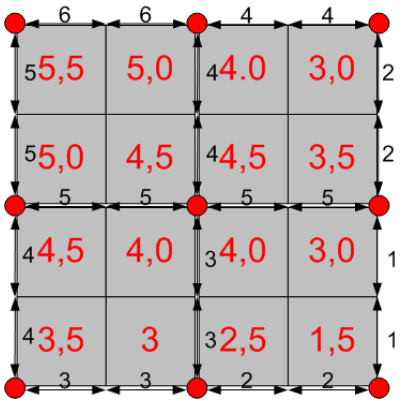

Abbildung 5.14: U-Matrix Beispiel: in Quandranten geteiltes Neuronennetz mit Entfernungsangabe für die Färbung

Parsereinstellungen angegeben werden. Sind ungültige Werte oder nichts eingetragen, so erfolgt die Beschriftung über die Musternummer.

```
1  private void matrixLabeln(Graphics2D screen){
2    screen.setColor(this.fontColor);
3    for(int i=0;i<this.siegerIndex.length;i++){
4      int y =
5        this.siegerIndex[i][0]*this.kantenlaenge+this.startpointY;
6      int x =
7        this.siegerIndex[i][1]*this.kantenlaenge+this.startpointX;
8      screen.drawLine(x,y,x,y);
9      if(!(labelListe[i].equals("DatensatzNr.")){
10       screen.drawString(
11         Integer.toString(i) + ":_" + labelListe[i], x+5, y-5);
12     }else{
13       screen.drawString(Integer.toString(i), x+5, y-5);
14     }
15   }
16 }
```

5.2 Benutzeroberfläche

5.2.1 Hauptfenster

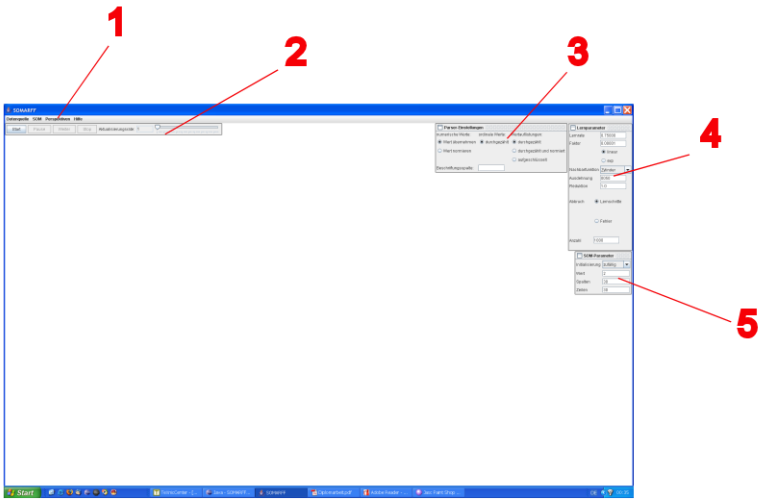

Abbildung 5.15: Hauptfenster von SOMARFF

Im oberen Teil des Hauptfensters findet der Benutzer das Menü (1). Darunter befindet sich die Konsole für die Steuerung der SOM-Zustände (2). Die Nummer (3) zeigt das Panel für die Einstellungen, nach denen der Parser die *.arff-Datei in einen Musterpool umwandelt. Das Panel (4) ist ein Formular zum Einstellen der Trainigsvorgaben. Unter der Nummer (5) findet sich das Panel für die SOM-Eigenschaften.

Das Menü

Es besteht aus 4 Menüreitern: „Datenquelle“, „SOM“, „Perspektiven“ und „Hilfe“.

Unter „Datenquelle“ läßt sich mit „Einstellungen“ das Panel für die Parsereinstellungen öffnen und schließen. Der Menüpunkt „Datei auswählen“ bringt einen Dateidialog zum Vorschein. In diesem Dialog kann die einzulesende *.arff-Datei ausgewählt werden. Wurde eine Datei gewählt und der Dialog mit „OK“ geschlossen, wird die Datei mit den Vorgaben aus dem Panel „Einstellungen“ in einen Satz von Mustern geparst. Der Menüpunkt „Datenblatt“ bietet eine Tabelle mit den eingegebenen Mustern.

Unter dem Menüreiter „SOM“ lassen sich die Panels für Trainingsparameter

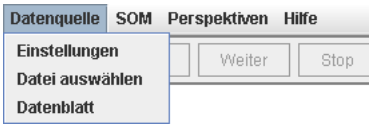

Abbildung 5.16: Menüpunkt Datenquelle

und für SOM-Eigenschaften ein und ausblenden.

Die verschiedenen Sichten auf die SOM wählt man unter „Perspektiven“.

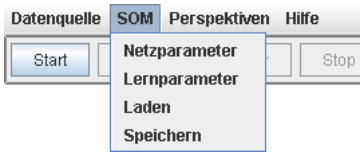

Abbildung 5.17: Menüpunkt SOM

Wird eine Perspektive angeklickt, so öffnet sich ein Fenster mit der gewünschten Sichtweise. Die verschiedenen Perspektiven können auch mehrmals gleichzeitig aktiviert werden, z.B. in verschiedenen Farben oder Zoom-Stufen.

Der Menüreiter „Hilfe“ bietet einen Info-Dialog zum Autor und eine kurze Anleitung.

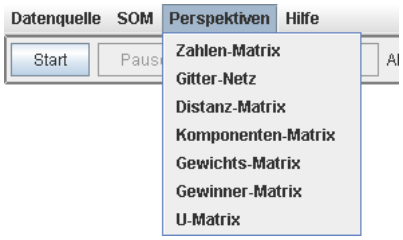

Abbildung 5.18: Menüpunkt Perspektiven

Abbildung 5.19: Menüpunkt Hilfe

5.2.2 Terminals

Parser-Einstellungen

Mit dem Formular kann festgelegt werden, wie die Datentypen aus der Quelldatei während des Parsens behandelt werden sollen. Dies geschieht durch selektieren des gewünschten Auswahlfeldes. Weiterhin kann im Textfeld darunter die Attributspalte angegeben werden, welche für die Beschriftung der Siegerneuronen genutzt wird. Die Bezeichnung der Spalte erfolgt über ihre Indexnummer, wobei von links nach rechts, von 0 beginnend, aufsteigend gezählt wird. (Bespiel: $0, 1, 2, ..., n$)Werden ungültige Werte angegeben, z.B. Werte, die außerhalb der Datendimension liegen, erfolgt die Beschriftung mit der Nummer des Musters.

Panel Lernparameter

Im Formular Lernparameter werden sämtliche Einstellungen für den Lernprozess vorgenommen.
Das Textfeld Lernrate erwartet einen Dezimalwert mit 5 Nachkommastellen. Nachkommastellen mit dem Wert 0 sind unbedingt mitanzugeben! (Beispiel: 0.30000). Es wird während des Lernvorgangs ständig aktualisiert. Erreicht

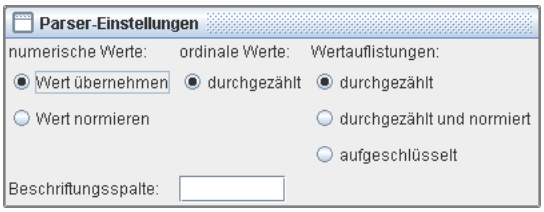

Abbildung 5.20: Panel Parser-Einstellungen

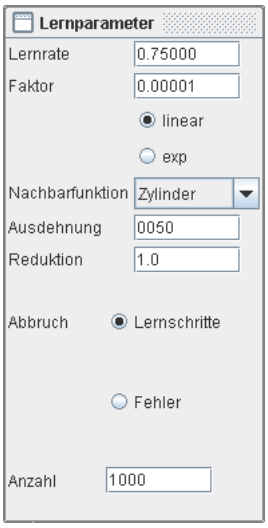

Abbildung 5.21: Panel Lernparameter

der Wert in diesem Textfeld einen Wert l mit $l \leq 0$, so wird das Training abgebrochen. Deshalb sollte das Feld stets vor Trainingsbeginn geprüft werden. Auch im Textfeld Lernfaktor muss eine Dezimalzahl mit 5 Nachkommastellen angegeben werden. Der Wert in diesem Feld wird zur Berechnung der nächsten Lernrate nach erfolgten Lernschritt benötigt und ist abhängig von der mathematischen Funktion zur Reduktion der Lernrate, welche über die Auswahlkästchen gewählt wurde. Wenn die lineare Reduktion ausgewählt ist, sollte der Wert sehr niedrig sein, z.B. 0.00100. Der Faktor wird dann bei jedem Lernschritt von der Lernrate abgezogen. Wird jedoch die exponentielle Funktion gewählt, sollte der Wert knapp unter 1 liegen, da sich die neue Lernrate durch die Multiplikation der alten Lernrate mit dem Faktor ergibt.

Unter den Auswahlkästchen für die Lernratenreduktion befindet sich ein Auswahlfeld zur Auswahl der Nachbarschaftsfunktion. Hier stehen mehrere zur Verfügung. Auch eine Erweiterung des Programms auf andere Funktionen sollte aufgrund der Softwarearchitektur kein Problem darstellen. Unter dem Auswahlfeld befindet sich ein Textfeld für die Ausdehnung der Erregung. Hier wird ein positiver ganzzahliger Wert erwartet, welcher während des Trainings ständig aktualisiert wird. Die Reduktion der Erregungsausdehung erfolgt linear in dem der Wert aus dem Textfeld Reduktion von der momentanen Ausdehnung abgezogen wird. Er ist bis auf 5 Nachkommastellen anzugeben. Dies geschieht solange, wie die Ausdehnung größer als 1 ist. Um die Erregungsausdehnung zur verfolgen, bietet sich besonders die „Gewinner-Matrix“ im Menü „Perspektiven“ an.
Unter dem Textfeld für die Erregungsreduktion befinden sich 2 Auswahlkästchen für die Art der Abbruchbedingung. Es kann ein Abbruch nach festgelegter Anzahl an Lernschritten, oder bei Erreichen eines bestimmten mittleren Quantisierungsfehlers, eingestellt werden. Nach Auswahl der Abbruchart muss im darunter liegenden Textfeld der Abbruchparameter, also die Anzahl der Lernschritte oder der Fehlerwert angegeben werden.

Panel SOM-Parameter

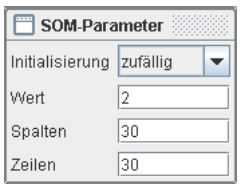

Abbildung 5.22: Panel SOM-Parameter

Das Panel SOM-Parameter ist für die Netzeigenschaften verantwortlich. In ihm wird die Art der Netzinitialisierung und die Netzdimension festgelegt.
Zur Initialisierung stehen 2 Initialisierungstypen über ein Auswahlfeld zur Verfügung. Wird die einheitliche Initialisierung ausgewählt, so werden die

Verbindungsgewichte zwischen den Kartenneuronen und den Eingangsneuronen mit einem einheitlichen Wert bei Trainingsstart initialisiert. Bei zufälliger Initialisierung erfolgt die Belegung mit Werten, welche zwischen 0 und dem angegebenen Wert liegen. Zur Angabe des Wertes für die Initialisierung steht unter dem Auswahlfeld ein Textfeld zur Verfügung, welches ganzzahlige Werte erwartet.
Die unteren beiden Textfelder erwarten als Eingabe die Netzdimension in Neuronen für Zeilen- und Spaltenrichtung.

Panel Steuerkonsole

Abbildung 5.23: Panel Steuerkonsole

Die Steuerkonsole dient dem Wechseln der SOM-Zustände. Über sie kann das Training begonnen (Schalter „Start“), pausiert (Schalter „Pause“), weiter fortgeführt (Schalter “Weiter“) und abgebrochen (Schalter „Stop“) werden.
Außerdem läßt sich hier einstellen, nach wievielen Lernschritten die Oberfläche aktualisiert werden soll. Dies geschieht über den Schieberegler. Die Anzahl an Lernschritten, während denen die Oberfläche „schläft“, wird im Textfeld angezeigt.

5.2.3 Perspektivfenster

Zahlenmatrix

Das Fenster ist aus dem Steuerpanel und der Anzeigefläche (5) zusammengesetzt. Im Steuerpanel kann zwischen der Darstellung der Neuronenaktivierung und der Darstellung der Komponente gewählt werden (1).
Bei der Berechnung der Aktivierung wurde die gegenseitige Beeinflussung zwischen den Kartenneuronen zur Vervahrensvereinfachung vernachlässigt. Die Aktivierung wird in grüner Schriftfarbe angezeigt. Bei der Darstellung

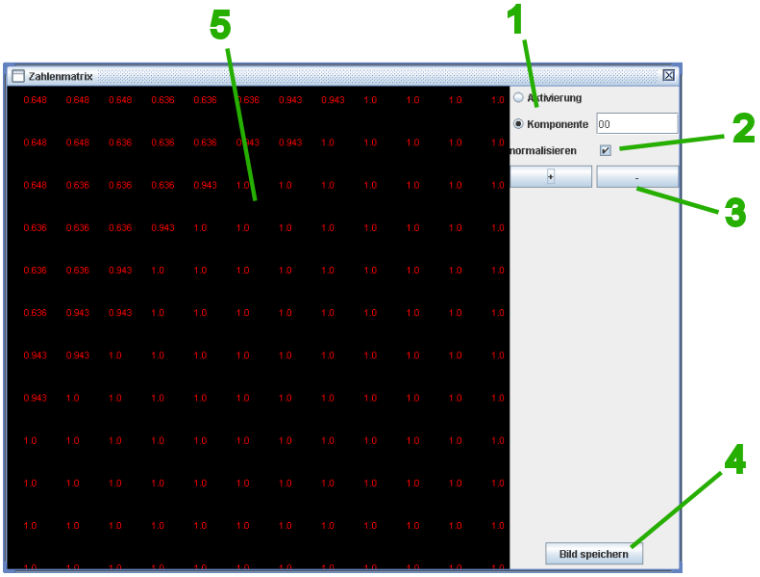

Abbildung 5.24: Perspektive Zahlenmatrix

der Komponente werden die Gewichte zwischen den Kartenneuronen und einem Eingabeneuron ausgegeben. Die Anzeige erfolgt in rot. Im Textfeld neben dem Auswahlfeld „Komponente“ wird das gewünschte Eingabeneuron angegeben.
Wird das Kontrollkästchen „normalisieren“ aktiviert, werden die Werte auf einen Bereich zwischen $[-1, 1]$ normalisiert ausgegeben (2). Über die Schlatflächen „+“ und „-“ kann die Schriftgröße verändert werden (3). Rechts unten befindet sich eine Schaltfläche, über die ein Bild der Anzeigefläche im JPG-Format gespeichert werden kann (4).
In der Anzeigefläche erscheinen die Daten in Form von gebrochenen Zahlen mit 3 Nachkommastellen (5).

Gitter-Matrix

Das Steuerpanel der Gitter-Matrix besteht nur aus 2 Schaltflächen für die Wahl der Malfarbe (1) und zum Abspeichern der Anzeigefläche als JPG-Datei (2). In der Anzeigefläche (3) wird das Neuronennetz wie ein Gumminetz angezeigt. In der aktuellen Softwareversion wurde die Perspektive nur für 2-

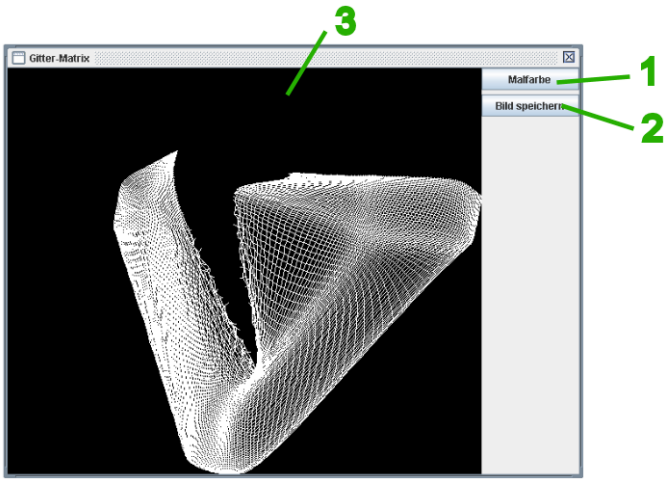

Abbildung 5.25: Perspektive Gitter-Matrix

dimensionale Eingabemuster realisiert.

Distanz-Matrix

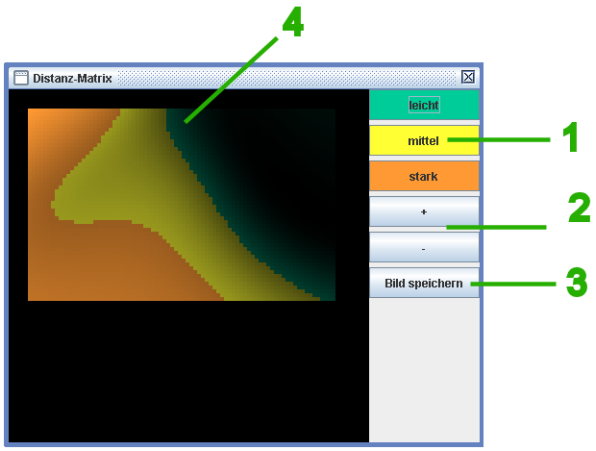

Abbildung 5.26: Perspektive Distanz-Matrix

Die Distanz-Matrix besitzt im Steuerpanel 3 Schaltflächen zur Auswahl der Malfarbe für geringe, mittelmäßige und starke Erregung (1).Darunter befinden sich 2 Schaltflächen zur Größenänderung der Kantenlänge für ein

Neuron (2). Neuronen werden im Anzeigepanel (4) als Quadrat dargestellt. Im unteren Bereich der Steuerkonsole befindet sich eine Schaltfläche, über die die Anzeigefläche (4) als JPG-Bild gespeichert werden kann (3).

Komponenten-Matrix

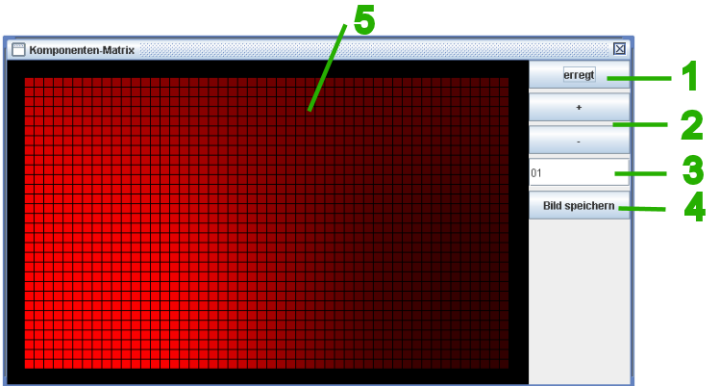

Abbildung 5.27: Perspektive Komponenten-Matrix

Mit diesem Panel kann die Gewichtsverteilung der Kartenneuronen zu einem Eingabeneuron als Matrix dargestellt werden. Die Gewichtswerte wurden auf 1 normalisiert. Für hohe Gewichte kann die Farbe über die Schaltfläche (1) festgelegt werden. Für Gewichte mit Werten nahe 0 geht die Farbe gegen schwarz. Die Angabe des Eingabeneurons erfolgt im Textfeld (3) über seine Indexnummer in den Mustern. Das erste Neuron hat die Nummer 0, das zweite die Nummer 1 und so weiter. Wird die Länge des Musters überschritten, so erfolgt die Anzeige für das letzte Eingabeneuron.
Die Neuronen werden in der Anzeigefläche (5) durch farbige Quadrate dargestellt, deren Kantenlänge über (2) variiert werden kann.
Über die Schaltfläche (4) kann eine Abbildung der Anzeigefläche(5) im JPG-Format gespeichert werden.

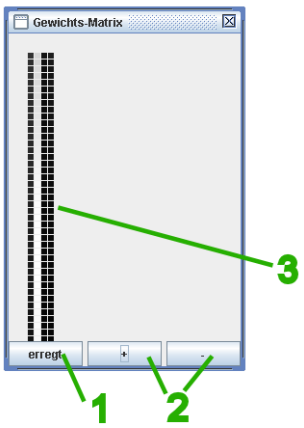

Abbildung 5.28: Perspektive Gewichts-Matrix

Gewichts-Matrix

Die Gewichts-Matrix bildet in der Anzeigefläche (3) sämtliche Verbindungsgewichte zwischen Kartenneuronen und Eingangsneuronen als farbige Quadrate ab. Dabei erfolgt die Anzeige in Form von Ketten senkrecht für jedes Eingangsneuron. Die Gewichtswerte wurden auf 1 normalisiert. Die Malfarbe ist über die Schaltfläche (1) änderbar. Die Kantenlänge kann über die danebenliegenden Schaltflächen (2) erhöht oder gesenkt werden.

Gewinner-Matrix

Die Gewinner-Matrix bildet in der Anzeigefläche (4) das Gewinnerneuron und seine Nachbarschaftswerte auf 1 normalisiert ab. Neuronen außerhalb des Nachbarschaftsradius werden mit schwarz gefärbten Quadraten dargestellt. Das Siegerneuronen erhält die Färbung, welche durch die Schaltfläche (1) bestimmt wird. Die Neuronen mit einem positiven Nachbarschaftswert erhalten Farbwerte im Bereich dazwischen. Neuronen mit negativem Nachbarschaftswert erhalten eine graue Färbung, z.B. bei der „Mexican Hat“-Funktion. Die Kantenlänge der Neuronen kann über die Schaltflächen (2) verändert werden. Die Schaltfläche (3) dient der Speicherung der Anzeigefläche (4) im JPG-Format.

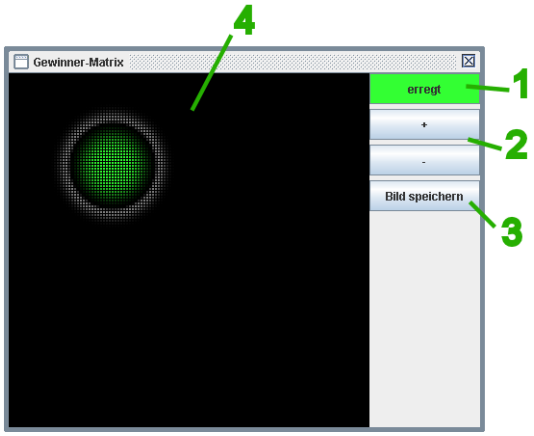

Abbildung 5.29: Perspektive Gewinner-Matrix

U-Matrix

Die Perspektive bildet in der Anzeigefläche (6) die berechnete U-Matrix, also die auf 1 normalisierten Entfernungen der Kartenneuronen zu ihren Nachbarneuronen farbig ab. Eine Abbildung der Anzeigefläche kann über die Schaltfläche (1) im JPG-Format gespeichert werden. Die Darstellung eines Neurons erfolgt in dieser Perspektive nicht durch einfache Quadrate, da sich herausstellte, dass die alleinige Darstellung des Verbindungswertes Darstellungsprobleme durch Lückenbildung brachte. Aus diesem Grunde wurden die Neuronen in Entfernungsquadrate aufgeteilt, 4 je Neuron. Die Entfernungsquadrate geben nun die gemittelte Entfernung zu ihren benachbarten Entfernungsquadraten an. Die Malfarbe für die Neuronen ist über die Schaltflächen (3) änderbar. Sie kann somit für nahe und für weite Entfernungen separat angegeben werden. Die Kantenlänge der Entfernungsquadrate ist über die Schaltflächen darunter änderbar (4). Desweiteren kann die Anzeigefläche (6) über die 4 Schaltflächen (4) in den Achsen verschoben werden. Die Schriftfarbe der Kartenbeschriftung kann über die Schaltfläche (2) eingestellt werden.

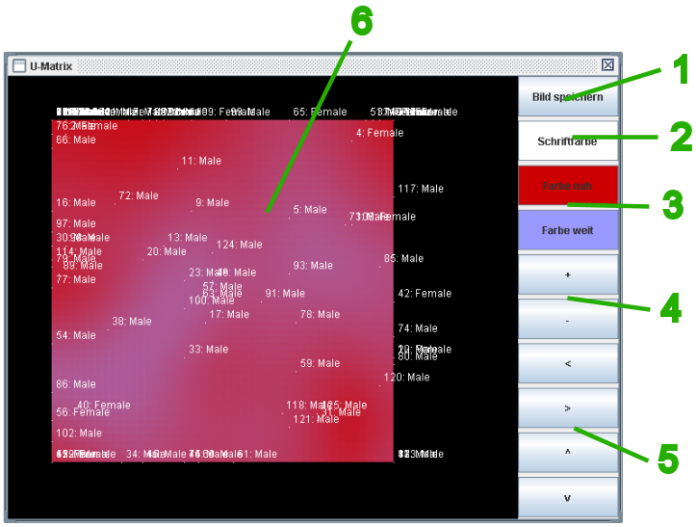

Abbildung 5.30: Perspektive U-Matrix

5.3 Anwendungsbeispiel

Im folgenden Teil soll die Bedienung von SOMARFF beispielhaft erläutert werden. Als arff-Datei wird die Datei „body.arff“ verwendet. Sie liegt auf dem Datenträger bei.
Nachdem SOMARFF gestartet wurde, klickt man im Menü auf den Reiter „Datenquelle“. Da die Einstellungen für den Parser geändert werden sollen, wird auf den Menüpunkt „Einstellungen“ geklickt. Dadurch wird das Panel für die Parser-Einstellungen sichtbar.
Hier stellt man für numerische Werte „normieren“ und für Wertelisten den Wert „aufgeschlüsselt“ ein. Als Labelspalte ist die Spalte 0 anzugeben, da das erste Attribut die Namen der Personen enthält. Das Panel kann nun wieder über den Menüpunkt „Datenquelle/Einstellungen“ geschlossen werden.
Nachdem nun die Parameter festgelegt wurden, laden wir die Datei „body.arff“ über den Menüpunkt „Datenquelle/Datei auswählen“. Die Daten können nun in der Datenblattansicht eingesehen werden. Sie kann über „Datenquelle/Datenblatt“ geöffnet und geschlossen werden.
Als nächstes soll die Netztopologie und die Initialisierung vorgegeben wer-

den. Im Panel „SOM-Parameter“ wird nun der Wert 50 für die Spalten und der Wert 40 für die Zeilen eingegeben. Die Initialisierung soll mit einem Zufallswert erfolgen, das Auswahlfeld kann deshalb auf „zufällig“ belassen werden. Die obere Grenze der Zufallszahl wird jedoch auf den Wert 5 korrigiert. Nun folgt die Angabe der Lernparameter. Das Training soll mit einer mittleren Lernrate beginnen und exponential fallen. Deshalb wird im Textfeld für die Lernrate der Wert 0.50000 und für den Lernfaktor 0.99850 eingestellt. Desweiteren wird das Auswahlkästchen „exp“ aktiviert. Als Nachbarfunktion stellt man die Funktion „Kegel“ ein. Der Radius soll anfänglich recht hoch sein so dass die gesamten Kartenneuronen stets betroffen sind. Als Startwert für den Radius wird deshalb grosszügig 80 gewählt. Dieser Wert steht immer in engem Zusammenhang mit der Netzdimension! Die Reduktion des Radius wird mit dem Wert 0.10000 parametrisiert. Das bedeutet, dass dem Radius in jedem Lernschritt 0.1 vom aktuellen Radius abgezogen wird. Das Training soll nach 2000 Lernschritten beendet werden. Darum wird die Auswahl für den Trainingsabbruch auf „Lernschritte“ belassen und der Wert 2000 im Textfeld Anzahl eingetragen.

Um die Reduktion des Erregungsbereiches gut mitzuverfolgen, wird die Perspektive „Gewinner-Matrix“ über das Menü „Perspektiven/Gewinner-Matrix“ aktiviert. Eine interessante Abbildung sollte auch die Komponenten-Matrix für die 2 Eingabeneuronen des Geschlechtes liefern. Da im Einstellungspanel des Parsers „aufgeschlüsselt“ für Wertelisten aktiviert wurde, berechnen sich diese Neuronen im Muster auf die Indexnummern 10 und 11. Über das Menü „Perspektiven/Komponenten-Matrix“ wird jeweils 1 Fenster für die Abbildungen geöffnet und die Neuronnummer im Textfeld angegeben. Um eine Abbildung der Daten in der U-Matrix zu bekommen, wird die Perspektive wie die anderen über das Menü geöffnet.

Der Trainingsstart erfolgt nun den Schalter „Start“ im Konsolen-Panel. Hier kann das Training auch pausiert, weitergeführt und beendet werden.

Nach 763 Lernschritten sind bereits eindeutige Ergebnisse zu beobachten. Die Gewinner-Matrix zeigt eine deutliche Verkleinerung des Erregungsbereiches. Die U-Matrix hat die Personen getrennt und topologisch sortiert abgebildet. Aufgrund der geringen Datenmenge des Beispiels lässt sich auch

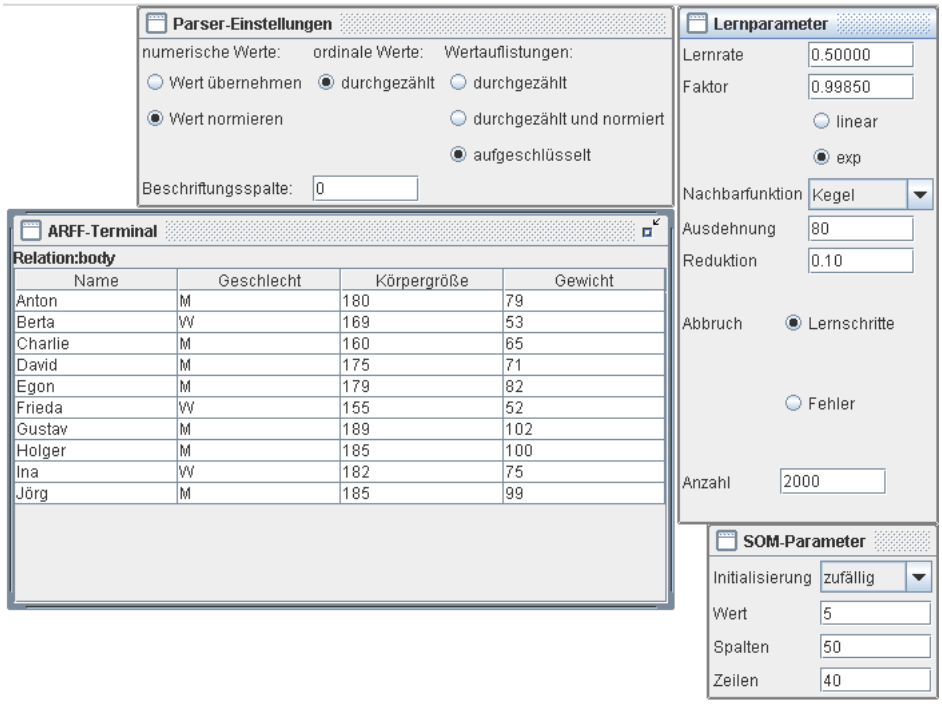

Abbildung 5.31: Startparameter zum Beispieltraining

die Sortierung der Daten relativ gut nachvollziehen. In den Komponenten-Matrizen lässt sich einerseits deutlich eine negative Korrelation der beiden Eingangsneuronen beobachten und außerdem die Trennung der Geschlechter in der U-Matrix nachvollziehen. Außerdem läßt sich auch erkennen, dass dem Netz „männliche Muster“ wesentlich häufiger angeboten wurden, als „weibliche Muster“, da der Bereich der erregten Neuronen für das Eingabeneuron 10 wesentlich mehr Fläche beansprucht als der Bereich für das Eingabeneuron 11.

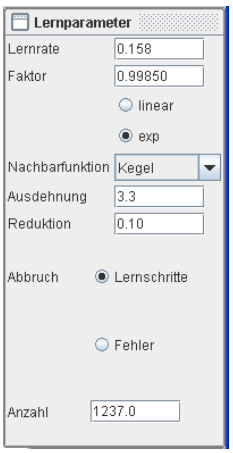

Abbildung 5.32: Lernparameter nach 763 Lernschritten

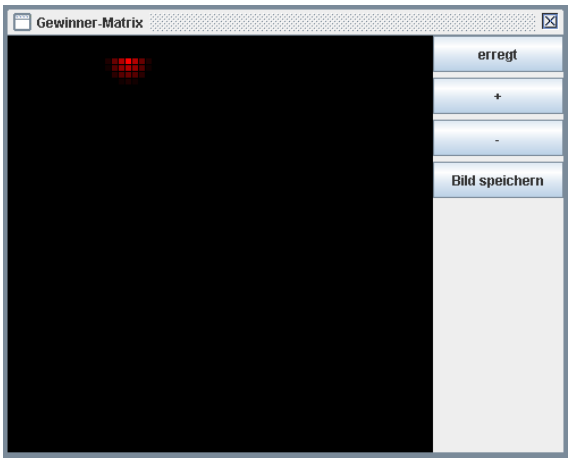

Abbildung 5.33: Gewinner-Matrix nach 763 Lernschritten

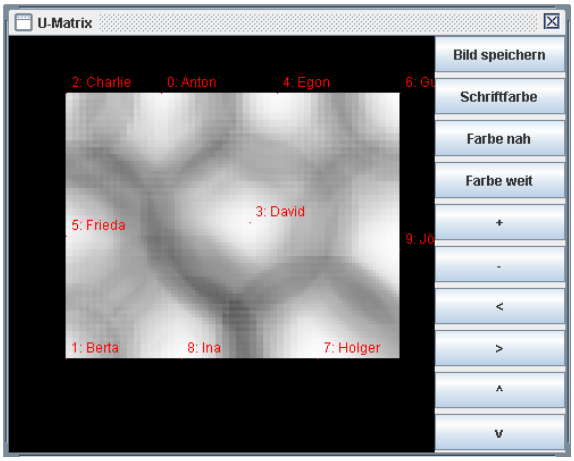

Abbildung 5.34: U-Matrix nach 763 Lernschritten

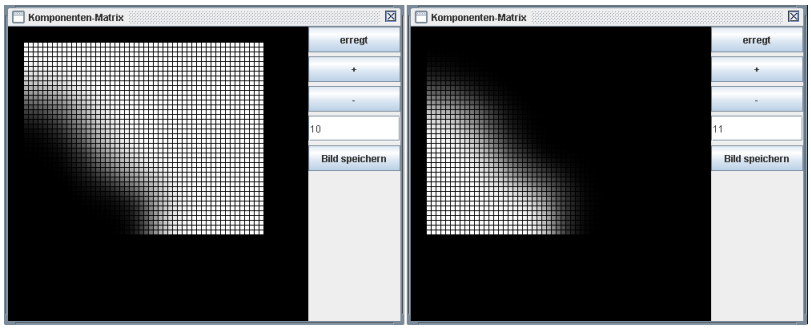

Abbildung 5.35: Komponenten-Matrizen für Eingabeneuron 10 und 11 nach 763 Lernschritten

Kapitel 6

Test

6.1 TestNr. 1

Beschreibung Der Test wurde im Zuge der Dokumentation des „Anwendungsbeispiel“ aus dem vorherigen Kapitel durchgeführt und wird dort ausführlich beschrieben. Die Daten wurden vom Autor willkürlich erfunden. Sie bilden eine kleine, übersichtliche Datenmenge ab, welche sich für das Nachvollziehen der Ergebnisse und Testen der grundlegenden Funktionalität gut eignet.
Analyseobjekte sind Personen, deren Name, Geschlecht, Größe und Gewicht als Attributtypen festgelegt wurden.
Als Testergebnis wird eine Trennung von männlichen und weiblichen Personen erwartet.

Datei:	body1.arff	Abbruchbedingung:	Lernschritte
numeric:	normiert	Abbruchparameter:	2000
String:	durchgezählt	SOM-Spalten:	50
Werteliste:	aufgeschlüsselt	SOM-Zeilen:	40
Labelspalte:	0	Initialisierungsart:	zufällig
Lernrate:	0.50000	Initialisierungswert:	5
Lernfaktor:	0.99850		
Reduktionsart:	exp		
Nachbarfunktion:	Kegel		
NB-Radius:	80		
NB-Reduktionswert:	0.10000		

Ergebnisauswertung Wie erwartet fand eine strikte Trennung der Personen nach Geschlecht statt. Die Neuronen 10 und 11 machen die Trennlinie zwischen den Geschlechtern durch ihre negative Korrelation besonders deutlich. Auch die Eigenschaft, dass häufige Muster mit größerer Fläche abgebildet werden, ließ sich sofort erkennen. Dieses wurde vom Autor erst bei der Betrachtung des Kartenergebnisses festgestellt und nicht bei der Erzeugung der Daten bedacht. Nach Prüfung der Daten wurde festgestellt, dass dieses Ergebnis schlüssig war, da nur 3 weibliche und 7 männliche Personen enthalten waren.

6.2 TestNr. 2

Beschreibung Bei diesem Test wurde die *.arff-Datei aus Test 1 um das Attribut „Bekleidung“ erweitert. Alle weiblichen Personen tragen Röcke und fast alle männlichen Personen tragen Hosen. Bis auf die Personen Gustav und Holger, da sie Schotten sind.
Da die Attribute der Wertelisten wieder aufgeschlüsselt werden, erfolgt die Abbildung des Attributes „Bekleidung“ durch die Neuronen 12 und 13. Die Neuronen 10 und 11 bilden wieder das Geschlecht ab.
Als Analyseergebnis wird erwartet, dass die Komponenten-Matrix den Erre-

gungsbereich des Eingabeneurons für „weiblich“ als Teilmenge des Erregungsbereichs für das Eingabeneuron des Attributes „Rock“ darstellt. Deckungsgleichheit wird nicht erwartet, da auch 2 männliche Personen Röcke tragen und somit der Erregungsbereich auch einen Teil der erregten Neuronen für „männlich“ mit abdeckt. Die U-Matrix sollte die Personen Gustav und Holger näher an den weiblichen Personen positionieren.

Datei:	body2.arff	Abbruchbedingung:	Lernschritte
numeric:	normiert	Abbruchparameter:	2000
String:	durchgezählt	SOM-Spalten:	50
Werteliste:	aufgeschlüsselt	SOM-Zeilen:	40
Labelspalte:	0	Initialisierungsart:	zufällig
Lernrate:	0.50000	Initialisierungswert:	5
Lernfaktor:	0.99850		
Reduktionsart:	exp		
Nachbarfunktion:	Kegel		
NB-Radius:	80		
NB-Reduktionswert:	0.10000		

Ergebnisauswertung Die Erwartungen an diesen Test wurden vollständig erfüllt. Das Trainingsergebnis stellte sich bereits nach 1000 Lernschritten ein.
In der U-Matrix läßt sich durch die dunkle Trennung deutlich zwischen den Geschlechtern unterscheiden. Weibliche Personen wurden unten links positioniert. Die beiden Mischformen „Gustav“ und „Holger“ liegen am linken Rand benachbart an Frauen und Männern, wobei die Trennung zwischen ihnen und den Frauen nicht so stark ausfällt, wie die Trennung der restlichen Männer und Frauen.

6.3 TestNr. 3

Beschreibung Für diesen Test wurde die Datei „clust3.arff“, welche zum dritten Übungsblatt der Vorlesung „Knowledge Discovery in Databases“ ge-

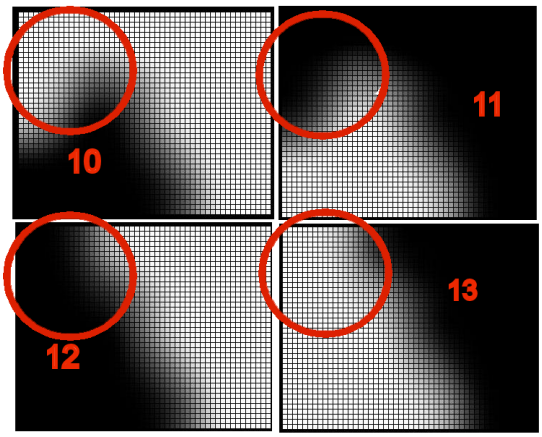

Abbildung 6.1: Test2: Komponenten-Matrizen der Neuronen 10, 11, 12, 13

hört, verwendet. Sie ist aus dem Internetauftritt der TU Ilmenau unter der Adresse [unb06a] und [unb06h] downloadbar. Das Übungsblatt mit den Kontaktinformationen und das Datenfile wurde außerdem auf dem Datenträger zur Diplomarbeit beigelegt.
Die Datei wurde für den Test aus folgenden Gründen gewählt:

1. Die Datei beinhaltet 2-dimensionale Muster aus numeric-Werten und eignet sich somit zum Test der Gitter-Matrix.

2. Es ist mit 600 Datensätzen eine große Anzahl von Mustern vorhanden, welche trotz der geringen Dimensionalität und Einfachheit der Daten im Selbstversuch nicht per Hand clusterbar war.

3. Das zu erwartende Ergebnis ist völlig unbekannt (Überaschungseffekt).

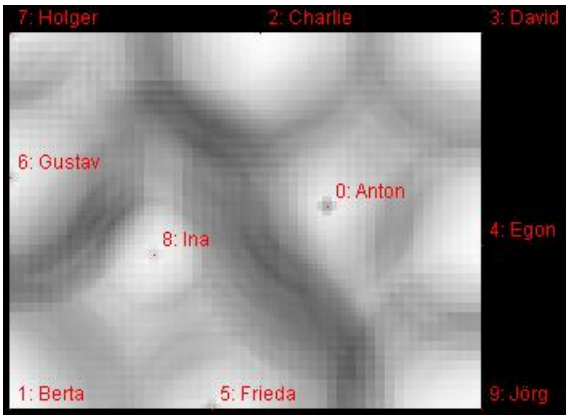

Abbildung 6.2: Test2: U-Matrix

Datei:	clust3.arff	Abbruchbedingung:	Fehler
numeric:	normiert	Abbruchparameter:	2.0
String:	durchgezählt	SOM-Spalten:	80
Werteliste:	aufgeschlüsselt	SOM-Zeilen:	50
Labelspalte:	ohne Angabe	Initialisierungsart:	zufällig
Lernrate:	0.50000	Initialisierungswert:	2
Lernfaktor:	0.99925		
Reduktionsart:	exp		
Nachbarfunktion:	Mexikaner Hut		
NB-Radius:	100		
NB-Reduktionswert:	0.02500		

Ergebnisauswertung Im Endergebnis wurden die Muster in 3 große Gruppen geclustert. Die Datenmenge war jedoch so groß, dass die Beschriftung die Identifizierung von einzelnen Mustern unmöglich machte. Dieses Problem kann durch den Zoom umgangen werden und ist durch Implementierung eines Infofensters in Kombination mit der Maus behebbar. Der Funktionalitätsnachweis wurde jedoch voll erbracht.
Die Abbildungen zeigen die einzelnen Phasen des Lernfortschritts. Die erste Reihe zeigt den Lernstand am Anfang, die darunter etwa zur Mitte des Trainings. Die beiden großen Bilder zeigen die U-Matrix und die Gitter-Matrix

am Trainingsende.
Der gewünschte Fehlergrad wurde nicht erreicht. Wahrscheinlich deshalb, weil nicht alle Muster zum Training angeboten wurden und der Quantisierungsfehler für die fehlenden Muster noch auf dem ursprünglichen Initialisierungswert stand.
In der U-Matrix lassen sich einwandfrei 3 große Cluster erkennen, welche durch dunkle Bereiche voneinander geteilt werden. Innerhalb der dunklen Bereiche haben sich die Kartenneuronen voneinander entfernt. Die helle Farbe innerhalb der Cluster zeigt die kurzen Entfernungen zwischen den Kartenneuronen.
Die Cluster sind auch in der Gitter-Matrix erkennbar, da die Maschen in den Ecken des Gitters dichter zusammenhängen als in der Mitte oder den Seiten. Außerdem erkennt man durch die Form den dreieckigen Eingaberaum.

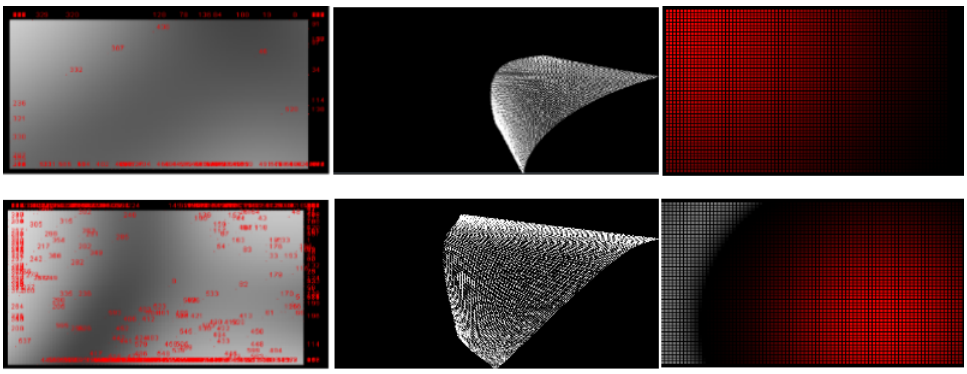

Abbildung 6.3: Test3: U-Matrix, Gitter-Matrix und Gewinner-Matrix bei Lernbeginn und in der Mitte des Trainings

Abbildung 6.4: Test2: U-Matrix am Trainingsende

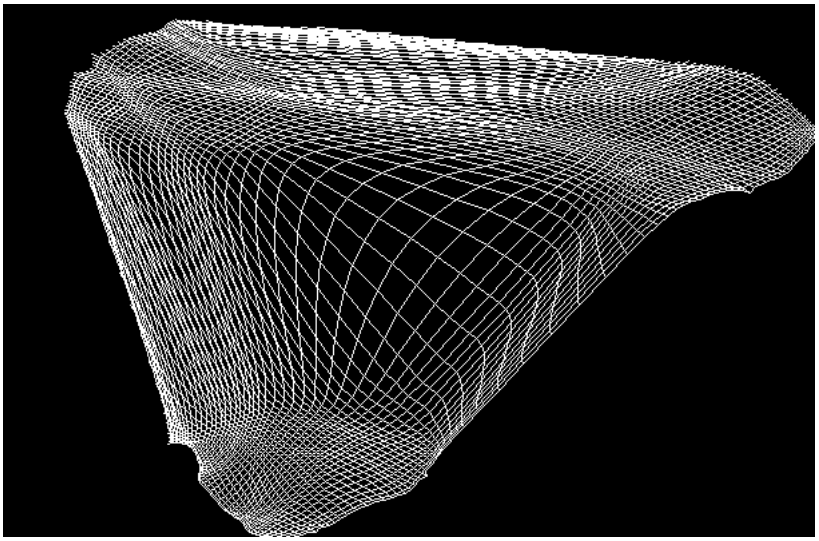

Abbildung 6.5: Test2: Gitter-Matrix am Trainingsende

Kapitel 7

Zusammenfassung und Ausblick

7.1 Zusammenfassung

Zentrale Aufgabe dieser Arbeit war das Klassifizieren und Visualisieren von Daten durch einen speziellen Typ neuronaler Netze, dem „Kohonen-Netz“. Ausgehend von den Grundlagen des Lernens in „Neuronalen Netzen“, geht die Arbeit im theoretischen Teil auf das biologische Vorbild und die algorithmische Umsetzung von „Kohonen-Netzen“ detailliert ein.
Desweiteren wurden gängige Visualisierungsmöglichkeiten vorgestellt.
Im praktischen Teil wurde ein Anwendungsprototyp erstellt. Mit ihm können Daten aus arff-Dateien gelesen, analysiert und klassifiziert werden.

7.2 Ergebnisse

Die in der Erarbeitung des theoretischen Teils gewonnenen Erkenntnisse konnten in der praktischen Umsetzung durch Tests bestätigt und vertieft werden. Der Algorithmus ist auch auf kleineren Systemen gut umsetzbar. Probleme gab es in der visuellen Ausgabe der Kartenergebnisse, da sie sehr rechenintensiv ist. Dieses Problem könnte evtl. durch den Wechsel zu einer SWT-Oberfläche oder das Auslagern in eigene Threads für die Berechnung gelöst werden. Als besonders kritisch erwies sich die Erstellung der U-Matrix. Hier wurden einige Möglichkeiten ausprobiert, beispielsweise durch Auftei-

lung der Neuronenzwischenräume in Zeilen und Spalten oder Bildung von Farbverläufen zwischen den Neuronen. Sie scheiterten an der Lesbarkeit oder der technischen Realisierung der Java-Komponenten, wodurch die Entwicklung eines eigenen Verfahrens notwendig wurde.
In den Tests konnte nachgewiesen werden, dass ein langsamer Lernfortschritt für die Funktionalität des Verfahrens von wesentlicher Bedeutung ist. Besonders die Verarbeitung von großen Datenmengen bereitete Probleme, da die Auswahl der Muster im Programm über Zufallsauswahl implementiert wurde. Wird der Lernprozess nicht genügend lange durchgeführt, werden nicht alle Muster angeboten, was wiederum zu Problemen mit der Ergebnisrepräsentation und der korrekten Berechnung des Quantisierungsfehlers führt. Eine Erweiterung des Programms wird deshalb als notwendig erachtet. Auch eine andere Strukturart, wie z.B. hexagonal konnte aufgrund der Bearbeitungsdauer nicht umgesetzt werden.
Wünschenswert, aber durch die begrenzte Zeit nicht umsetzbar, war eine 3-dimensionale Ausgabe über Java3D-API und eine Lade- und Speicherfunktion der SOM in XML-Dateien.

7.3 Ausblick

Trotz der langen Trainingsphasen sieht der Autor hohes Potential für diese Technologie. Dies gilt besonders für die Bereiche autonome Roboter und neuronale Implantate. Beispielsweise wäre es denkbar, eines Tages, defekte Hirnareale durch Module aus „Kohonen-Netzen“ ersetzen zu können.
Die Bearbeitung dieser Arbeit hat den Autor ambitioniert, sich weiterhin mit dieser Technologie zu beschäftigen. Speziell die Umsetzung des Verfahrens in einem autonomen System als sensorische und motorischsche Karten wird als höchst interessant erachtet.

Kapitel 8

Anhang

8.1 Geschichte der Künstlichen Intelligenz

Die Geschichte der Künstlichen Intelligenz gliedert sich in 4 Phasen, welche folgend beschrieben werden.

1. Ihre Gründungphase (Ende 50er Jahre), wird auch Power-Based-Approach genannt. In ihr wurden erste Ansätze nicht numerischer Informationsverarbeitung erforscht. Charakteristisch für diese Phase ist die Beschäftigung mit Problemen wie das Lösen von Puzzles, mathematisches Beweisen und symbolischen mathemtischen Operationen. Hauptsächlich wurde dabei Wert auf den Nachweis der Machbarkeit gelegt. Diese Phase führte zu einer zu großen und übertriebenen Erwartungshaltung. Produkte dieser Phase sind das erste neurale Netzwerk durch Minsky (1951), mit IPL-II die erste Programmiersprache für Künstliche Intelligenz oder MANIAC 1, das erste Schachprogramm, welches dafür entwickelt wurde, gegen einen Menschen zu spielen und ihn zu schlagen. McCarty gründete mit Minsky das KI-Labor am MIT und stellte die bis heute noch in der KI verwendete Programmiersprache LISP vor.

2. In dieser Entwicklungsphase (60er Jahre) entstanden erste Forschungsgruppen an führenden amerikanischen Universitäten mit dem Hauptaugenmerk auf der systematischen Bearbeitung von zentralen Fragestel-

lungen. Beispiele dafür sind die Sprachverarbeitung, das automatische Problemlösen sowie die visuelle Systemanalyse. In dieser Phase trat eine massive Förderung durch die Defense Advanced Research Projects Agency (DARPA), welche dem amerikanischen Verteidigungsministerium angehört, auf. 1969 veröffentlichen Minsky und Papert eine Studie, in welcher die Beschränktheit des einschichtigen Perzeptrons dargelegt wird. Durch diese Aussage kommt es fast zum Erliegen der Forschung im Bereich des konnektionistischen KI.

3. Diese Phase (70er Jahre) wird auch Knowledge-Based-Approach genannt. Mit ihr ging der Aufbau von Forschergruppen in Europa, besonders in Deutschland und Großbritannien, einher. Diese Phase wurde vom Entwurf integrierter Robotersysteme und ganz besonders von der Entwicklung von Expertensystemen geprägt. Die dabei erstellten Anwendungen machten regen Gebrauch von umfangreichen codierten Wissensbeständen. Im Gegensatz zur Power-Base-Approach stand dabei die Verwendung von formalisierten Problemlösungswissen und spezieller Verarbeitungstechniken im Vordergrund. In dieser Phase wurden große Fortschritte in den Bereichen der Wissenrepräsentation und der Systemarchitektur erreicht. Beispielanwendungen wie das Erkennen von kontinuierlich gesprochener Sprache, aus Analyse und Synthese in der Chemie, aus der medizinischen Diagnostik und Therapie oder Programme zur Fehleranalyse technischer Systeme fallen in diese Phase. Ein Beispiel dafür ist das 1971 entstandene Programm PARRY. Es simuliert durch sein Verhalten einen paranoiden Menschen und klinische Psychiater waren nicht imstande den Unterschied zwischen krankem Mensch und PARRY auszumachen. Ein anderes Beispiel dieser Epoche ist das 1974 durch Edward Shortliffe entwickelte Programm MYCIN, welches Ärzten bei der Diagnose von Blutinfektionen und der Behandlung durch Antibiotika unterstützte. 1972 entwickelt Roussel die heute noch populäre Logik-Programmiersprache PROLOG.

4. In der aktuellen Entwicklungsphase (seit 80er Jahre) fand eine umfassende Mathematisierung der Künstlichen Intelligenz statt. Außer-

dem wurde das Konzept der Wissenverarbeitung präzisiert. Desweiteren wurden neue Themen wie Situiertheit, verteilte Künstliche Intelligenz und Neuronale Netzwerke aufgegriffen. Seit 1990 ist ein eindeutiger Trend zu integrierten Lösungsansätzen zu beobachten. Desweiteren kam es in dieser Phase zu einer starken Einflußnahme durch das Internet und viele Forschungen wurden durch Anwendungen und Anwendungsperspektiven beeinflußt. Außerdem wurde der Bedarf deutlich, heterogene Wissenquellen in übergreifenden Anwendungen zusammenzuführen und vorhandene Wissenbestände anpassbar zu machen, damit sie für neue Anwendungen zugeschnitten und wiederverwendbar werden (Wissensbasen, Wissensmanagement).

(vgl. [HM00b],[RCR03])

8.2 Über die Entwicklung Künstlicher Neuronaler Netze

Anfangsphase (1942-1955)

1943 wurde der Begriff „Neuron“ erstmals vom Neurophysiologen W. S. McCulloch und dem Mathematiker W. Pitts eingeführt.
1949 wurde die „Hebbsche These“ vom Psychologen D. Hebb veröffentlicht. In ihrer allgemeinen Form ist sie bis heute Basis fast aller neuronaler Lernverfahren.
1950 vertrat der Neurophysiologe K. Lashley die These, dass die Information im Gehirn in verteilter Repräsentation gespeichert sein muss. Grundlegend dafür waren seine Experimente mit Ratten, bei denen die Gehirne teilweise zerstört wurden.
1951 entwickelte M. Minsky einen Neurocomputer „Snark“, er wurde aber nie praktisch eingesetzt.
(vgl. [unb06b], [HM00b], [ZS98])

Erste Blütezeit (1955-1969)

1957 entwickelte F. Rosenblatt das Perzeptron.
1957-58 wurde auch der erste erfolgreiche Neurocomputer, der „Mark-1“ von F. Rosenblatt, C. Wightman und Mitarbeitern des MIT gebaut. Er wurde für Mustererkennungsprobleme eingesetzt und ist mittels Motoren und variablen Potentiometern realisiert worden.
1958-1962 erfolgte die mathematische Beschreibung der Hebb'schen Lernregel durch B. Widrow. Widrow und Hoff entwickelten die Deltaregel, welche in einer ersten echten Anwendung eingesetzt wurde, mit deren Hilfe man den Nachhall in Telefonleitungen korrigierte.
1960 entwickelte B. Widrow das Adaline und dessen Erweiterung, das Madaline.
Mitte 60er Jahre kündigte F. Rosenblatt, mit Blick auf sein Perzeptron, eine neue Ära der Neurocomputer an.
K. Steinbuch stellte in seiner Arbeit „Die Lernmatrix“ Vorgänger der heutigen neuronalen Assoziativspeicher vor. Vorlage dafür waren die Pawlow'schen bedingten Reflexe.
1969 erschien das Buch „Perceptrons“, in dem M. Minsky und S. Papert behaupteten, dass bestimmte wichtige Problemstellungen mit einem Perzeptron niemals lösbar seien. Gemeint war das XOR-Problem sowie dessen Umkehrung, das „parity“-Problem und das „connectivity“-Problem (Figur einfach verbunden oder aus mehreren Figuren bestehend). Das Buch löste eine weltweite Verunsicherung gegenüber den Künstlichen Neuronalen Netzen aus und führte in den 70er Jahren zum Erliegen der Neuroforschung. Dies war auch Grund dafür, dass Forschungsgelder, welche von der DARPA zur Verfügung gestellt wurden, in andere Forschungszweige, z.B. der symbolverarbeitenden KI, flossen. Die Kritik an den KNN konnte später als unbegründet zurück gewiesen werden.
Durch den Einsatz versteckter Neuronen wurde es möglich, vorher als unlösbar angenommene Probleme zu lösen.
(vgl. [unb06b], [HM00b], [ZS98])

Die stillen Jahre (1969-1985)

1972 stellte Teuvo Kohonen ein Modell eines linearen Assoziators vor. Charakteristisch für dieses Modell waren lineare Aktivierungsfunktionen sowie kontinuierliche Werte für Gewichte, Aktivierungen und Ausgaben.
1973 entwickelte der Deutsche Christoph von der Malsburg ein komplexes Neuronenmodell, welches dem biologischen Vorbild stärker angelehnt war. Durch Computersimulationen konnten Ähnlichkeiten mit den neurophysiologischen Entdeckungen in den Arbeiten von Hubel und Wiesel nachgewiesen werden (Bildung rezeptiver Felder mit Orientierungsspezifität).
1974 wurde, neben Wettbewerbslernen und der Counterpropagation, der Backpropagation -Algorithmus von Paul Werbos gefunden. Er entstand durch Modifikation der Hebbschen Lernregel, der Deltaregel, sowie der Widrow-Hoff-Regel und gewann erst ca. 10 Jahre später durch Arbeiten von Rumelhart und McClelland an Bedeutung.
1975 entwickelte F. Fukushima aus dem Perzeptron das Cognitron, indem er Rückkopplungen innerhalb einer Neuronenschicht einfügte. Zu dieser Zeit wurden auch andere Alternativen zum Perzeptron erarbeitet, so z.B. das CMAC-Netz von J. Albus.
1980 wurde eine neue Klasse von selbstorganisierten Netzen entwickelt. Grundlegend dafür war die Entwicklung des unüberwachten Lernens, an der Teuvo Kohonen, S. Grossberg, G. Carpenter, D. Rumelhart und J. J. McClelland beteiligt waren. Hieraus enstanden die „Self-Organizing feature maps“ von Kohonen und die „Adaptive Resonance Theorie“ von Grossberg.
1981 schlugen McClelland und Rumelhart strukturierte parallelarbeitende Netze vor, welche in der Lage waren, aus der Erkennung von Teilmustern Buchstaben und aus Buchstaben Wörter zu bestimmen.
1982 entwickelte J.J.Hopfield aus magnetischen Anomalien eine neue Netztopologie. Hopfield veröffentlichte in diesem Jahr sein Konzept der globalen Energiefunktion, welches das neuronale Äquivalent des Ising-Modells der Physik darstellt. Das Modell hatte entscheidenen Einfluss auf die Entwicklung eines neuen breiten Interesses an den Künstlichen Neuronalen Netzen.
1983 stellte K. Fukushima, Miyake und Ito das Neocognitron (Erweiterung

des Congnitrons) vor. Dieses Modell diente der positions- und skalierungsinvarianten Erkennung handgeschriebener Zeichen und war dem visuellen System von Katzen nachempfunden. Im gleichen Jahr wiesen Kirkpatrick, Gelatt und Vecci nach, wie man mit der „Bolzmann-Maschine“ (statistisches Modell eines KNN) schwierige Optimierungsaufgaben wie Chip-Platzierung, Verdrahtungsprobleme und das Problem des Handlungsreisenden lösen kann. (vgl. [unb06b], [HM00b], [ZS98])

Renaissance (1985-heute)

1984 entwickelten T. Sejnowski und G. Hinton das stochastische Lernen (Bolzmann-Maschine).
1986 wurde durch C. Rosenberg und T. Sejnowski ein Programm namens „NetTalk“ entwickelt. Das Programm benötigte für das Lernen der englischen Sprache lediglich 16h. Bei konventioneller Programmierung hätten dafür nicht weniger als 20 Mann-Jahre ausgereicht (im Vergleich zum wissensbasierten System „DECtalk“)!
1985 wurde von B. Kosko eine neue Klasse von assoziativen Speichern vorgestellt (BAM).
1987 realisierten C. Mead und F. Faggin als hardwaretechnische Umsetzung von Künstlichen Neuronalen Netzen das „Silicon Eye“ und „Silicon Ear“.
Seit 1986 entwickelte sich das Forschungsgebiet der Künstlichen Neuronalen Netze explosionsartig. Derzeit sind einige Tausend Wissenschaftler in diesem Bereich tätig und es gibt unzählige Veröffentlichungen zu diesem Thema. Einige Fachgruppen, welche sich hauptsächlich mit Künstlichen Neuronalen Netzen beschäftigen, sind die „International Neural Network Society“ (INNS), die „European Neural Network Society“ (ENNS).

(vgl. [unb06b], [HM00b], [ZS98])

8.3 Inhalt des Datenträgers

Verzeichnis	Inhalt
\	Wurzelverzeichnis
\Datenfiles	Hier finden sich ARFF-Dateien zum Testen der Software.
\Diplomarbeit	Enthält die Diplomarbeit als PDF-File.
\Quellen	Enthält die, im Literaturverzeichnis erwähnten Websites und aus dem Internet bezogene PDF-Dateien. In der „Index.txt“ findet sich ein Verzeichnis, der Zugehörigkeit der Dateinamen zu ihren URL‘s.
\Sourcecode	Unter diesem Verzeichnis ist der Java-Code in einem Verzeichnisbaum abgelegt.
\Software\small	In diesem Verzeichnis befindet der erstellte Anwendungsprototyp in Form einer ausführbaren JAR-Datei ohne Quellcode und javadoc.
\Software\big	Der Anwendungsprototyp in Form einer ausführbaren JAR-Datei mit Quellcode und javadoc befindet sich in diesem Verzeichnis.

8.4 Erklärung

Hiermit erkläre ich, dass die vorliegende Diplomarbeit von mir selbst ohne fremde Hilfe erstellt wurde. Alle benutzten Quellen sind im Literaturverzeichnis aufgeführt. Diese Arbeit hat in dieser oder ähnlicher Form noch keinem anderen Prüfungsausschuss vorgelegen.

Brandenburg, den 2. März 2006, Sven Schröder

Literaturverzeichnis

[AU06] Alfred Ultsch, Fabian M.:
http://www.hmwk.hessen.de/md/content/forschung/cebit_uni_ma.pdf
Universität Marburg, 21.02.2006

[CJ01] Cleve J., Lämmel U.: *Künstliche Intelligenz.* Fachbuchverlag Leipzig, 2001

[Dor91] Dorffner: *Konnektionismus.* Teubner Verlag, 1991

[EF06] Eric Freeman, Elisiabeth F.: *Entwurfsmuster von Kopf bis Fuß.* O'REILLY, 1. Auflage 2006

[Gin05] Ginolas, Wolfgang:
http://www.fh-wedel.de/~iw/Lehrveranstaltungen/SS2005/SeminarKI
/Ausarbeitung2ClusteranalyseGinolas.pdf. In: *Seminar zum Thema Künstliche Intelligenz: Clusteranalyse* FH Wedel, 11.05.2005

[HM00a] Hannebauer M., Geske U.: *History of Artificial Intelligence, Exhibition ECAI 2000, GMD Report 111, Part I: Posters.* GMD-Forschungszentrum Informationstechnik GmbH, August 2000

[HM00b] Hannebauer M., Geske U.: *History of Artificial Intelligence, Exhibition ECAI 2000, GMD Report 112, Part II: Posters.* GMD-Forschungszentrum Informationstechnik GmbH, August 2000

[JH99] Jochen Heinsohn, Rolf Socher-Ambrosius: *Wissensverarbeitung.* Spektrum Akademischer Verlag GmbH, 1999

[JH01] *Kapitel* 16. In: J. HEINSOHN, R.Socher-Ambrosius: *Handbuch der Infomatik*. Fachbuchverlag Leipzig, 2001, S. 555ff

[Kin92] KINNEBROCK: *Neuronale Netze*. Oldenbourg Verlag, 1992

[Koh97] KOHONEN: *Self-Organizing Maps*. Second Edition. Springer Verlag, 1997

[Kre00] KREBS, Susanne: http://www.aifb.uni-karlsruhe.de/Lehre/Sommer2003/ISF /Seminararbeit_SOM.pdf. In: *Selbstorganisierende Karten*, 20.09.2000

[Lip06] LIPPE: http://cs.uni-muenster.de/Professoren/Lippe/lehre/skripte /wwwnnscript/strfx/koh_algorithmus.html. In: *Einführung in Neuronale Netze Der Algorithmus (SOM)*, 21.02.2006

[ME01] MAXIMILIAN EIBL, Thomas M.: http://www.uni-hildesheim.de/%7Emandl/Publikationen/ aiw_ab_01_2001_Eibl_Mandl_AB_2D.pdf. In: *Arbeitsbericht Nr.: 1/2001* Universität Hildesheim, 1/2001

[Met96] METZE, Florian: http://www.phonetik.uni-muenchen.de/Lehre/Skripten/Seminare/HS_WS1997/Kohonen.ps. In: *Kohonen-Netze*, 09.07.1996

[ML01] MEYERS LEXIKONREDAKTION, Prof.Dr.A.Schwill: *Duden Informatik*. Dudenverlag, 2001 (3. Auflage)

[RCR03] ROLLINGER C. R., Görz G.: *Handbuch der Künstlichen Intelligenz*. Oldenbourg Verlag, 2003

[Spi00] SPITZER: *Geist im Netz - Modelle für Denken und Handeln*. Spektrum Verlag, 2000

[Ten95] TENHAGEN, Andreas:
http://www.math.uni-muenster.de/SoftComputing/lehre/material/nn_einfuehrung.pdf, 1995

[Ult06] ULTSCH:
http://www.mathematik.uni-marburg.de/ databionics/-de//?q=esom Philipps-Universität Marburg, Arbeitsgruppe Datenbionik Fachbereich Informatik und Mathematik, 21.02.2006

[unb06a] UNBEKANNT:
http://www.tu-ilmenau.de/site/dbis/Data-Warehousing_und.3176.0.html. In: *Link zum Download der Datei clust3.arff* TU Ilmenau, 20.02.2006

[unb06b] UNBEKANNT:
http://germazope.uni-trier.de/Projects/WBB/woerterbuecher/dwb/wbgui?lemid=GL04849, 21.02.2006

[unb06c] UNBEKANNT:
http://lrs2.fmi.uni-passau.de/online/SOM/SOM.pdf. In: *Technische Anwendungen von Selbstorganisierenden Karten* Universität Passau, 21.02.2006

[unb06d] UNBEKANNT:
http://nwg.glia.mdc-berlin.de/teacher.php/de/default/glossar, 21.02.2006

[unb06e] UNBEKANNT:
http://www.computerbase.de/lexikon/ComputerBase Medien GbR, 21.02.2006

[unb06f] UNBEKANNT: http://www.ifs.tuwien.ac.at/ifs/research/pub_html/rau_oegai98/node4.html. In: *Distance Connections und ein 3d-User Interface* TU Wien, 21.02.2006

[unb06g] UNBEKANNT:
http://www.spinfo.uni-koeln.de/lehre/HS_Rolshoven/papers
/SOM/img31.html Universität Köln, 21.02.2006

[unb06h] UNBEKANNT:
http://www.tu-ilmenau.de/site/dbis/fileadmin/template/FakIA/Strukt-Fakultaet_IA/ipim/dbis/kdd/uebung3.pdf. In: *Übungsblatt 3 zur VL „Knowledge Discovery in Databases“* TU Ilmenau, 21.02.2006

[WH01] WITTEN H., Frank E.: *Data Mining.* Hanser Verlag, 2001

[Zel03] ZELL: *Simulation Neuronaler Netze.* Oldenbourg Verlag, 2003

[ZS98] ZAKHARIAN S., Thoer S.: *Neuronale Netze für Ingenieure.* VIEW-EG Verlag, 1998

Abbildungsverzeichnis