Technical Report

Pattern Recognition and Image Processing Group
Institute of Computer Aided Automation
Vienna University of Technology
Favoritenstr. 9/1832
A-1040 Vienna AUSTRIA
Phone: +43 (1) 58801-18351
Fax: +43 (1) 58801-18392
E-mail: selbalex@prip.tuwien.ac.at
URL: http://www.prip.tuwien.ac.at/

PRIP-TR-62 May 22, 2000

Modellselektion von Clusteringverfahren mittels minimaler Beschreibungslänge

Alexander Selb

Abstract

This diploma thesis presents the use of the *minimum description length* (*MDL*) principle for unsupervised clustering. Two frameworks are presented which determine the right size of neural networks used for *crisp/hard* and *fuzzy* clustering purposes. The optimization procedure starts with a network consisting of a large number of network nodes and iteratively reduces the net size until a complexity criterion is met. The *MDL* principle is used to measure the complexity of the network. Different instantiations of the *MDL* algorithm, depending on the coding procedure, are possible and derived in the thesis. In addition, we demonstrate how the *MDL* principle increases the robustness of the training algorithm by providing a criterion for *outlier* detection. Also a *growing* component based on *MDL* is presented to cope with the formation of new data clusters due to non-stationary data.

In order to overcome the problem of initialization the *growing neural gas* (GNG) algorithm is used which thereby also increases the computational efficiency of the *MDL* algorithm. Furthermore, we present additional methods to increase the computational efficiency. Typical applications, like 2 dimensional clustering, data compression, *RGB* color image segmentation and learning the *hand-eye* coordination for robot control, are used to demonstrate the behaviour of the *MDL* algorithm.

"... with a little help from my friends ..."
(Joe Cocker)

Danksagung

An dieser Stelle möchte ich all jenen danken, die mich während meines Studiums und meiner Diplomarbeit unterstützt haben.

Sehr großen Dank möchte ich Horst Bischof aussprechen, der meine Praktika und meine Diplomarbeit betreut und unterstützt hat. Weiters möchte ich ihm danken, daß er mir die Möglichkeit gegeben hat, in die *scientific community* hineinzuschnuppern. Thomas Melzer möchte ich danken, daß er mir durch Diskussionen über seine Diplomarbeit geholfen hat, mich schneller in das Thema *adaptive robot control* einzuarbeiten und Mißverständnisse auszuräumen.

Auch möchte ich Angela danken, daß sie auf manche gemeinsame Freizeit verzichtet hat, damit ich mich auf Prüfungen vorbereiten konnte. Den größten Dank gilt aber meinen Eltern, die meine Ausbildung überhaupt ermöglicht und alle meine Entscheidungen diesbezüglich akzeptiert und unterstützt haben.

Danke

Abstract

This diploma thesis presents the use of the *minimum description length* (*MDL*) principle for unsupervised clustering. Two frameworks are presented which determine the right size of neural networks used for *crisp/hard* and *fuzzy* clustering purposes. The optimization procedure starts with a network consisting of a large number of network nodes and iteratively reduces the net size until a complexity criterion is met. The *MDL* principle is used to measure the complexity of the network. Different instantiations of the *MDL* algorithm, depending on the coding procedure, are possible and derived in the thesis. In addition, we demonstrate how the *MDL* principle increases the robustness of the training algorithm by providing a criterion for *outlier* detection. Also a *growing* component based on *MDL* is presented to cope with the formation of new data clusters due to non-stationary data.

In order to overcome the problem of initialization the *growing neural gas* (GNG) algorithm is used which thereby also increases the computational efficiency of the *MDL* algorithm. Furthermore, we present additional methods to increase the computational efficiency. Typical applications, like 2 dimensional clustering, data compression, *RGB* color image segmentation and learning the *hand-eye* coordination for robot control, are used to demonstrate the behaviour of the *MDL* algorithm.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Algorithmenverzeichnis

Kapitel 1

Einleitung

Inhaltsangabe

1.1 Motivation und Zielsetzung

Bei der Anwendung von neuralen Netzen stellt sich oft die Frage, wie groß dieses Netzwerk sein soll, um die gestellte Aufgabe befriedigend zu lösen. Die Größe eines neuralen Netzwerkes bestimmt oft das Laufzeitverhalten der Methode, welche das Netzwerk verwendet. Im Gegensatz zum menschlichen Gehirn müssen bei der Softwareimplementierung die verschiedenen Abläufe im Netzwerk sequentiell abgearbeitet werden. Weiters leidet die Generalisierfähigkeit von neuralen Netzen, wenn zuviele Netzwerkeinheiten verwendet werden, da die zu lernende Information vom Netzwerk "auswendig" gelernt wird. Um die vielen Freiheitsgrade eines großen Netzwerkes trainieren zu können, benötigt man eine große Anzahl an Trainingsdaten.

Die ideale Netzwerkgröße ist jedoch von der Anwendung abhängig und kann sich im Laufe der Zeit ändern. Dadurch ist es sehr schwer, eine fixe, minimale Netzwerkgröße vorzugeben, da man dabei Gefahr läuft, im Verlauf der Anwendung ein zu großes oder ein zu kleines Netzwerk zu bekommen. In der Regel ist ein zu

großes Netzwerk weniger problematisch, jedoch kann ein zu kleines Netzwerk seine gestellte Aufgabe nicht mehr in der geforderten Genauigkeit lösen.

Bei der Modellselektion von neuralen Netzen wurden deshalb Kriterien entwickelt, welche die Modellkomplexität bestimmen und Anhaltspunkte für die Selektion der richtigen Netzwerkgröße liefern. Das *Akaike Information Criterion* (AIC), das *Network Information Criterion* (NIC), die *Minimum Description Length* (MDL) und die *Structural Risk Minimization* (SRM) werden oft in der Literatur für derartige Kriterien angeführt, wobei einige davon nicht für neurale Netze oder nur für *supervised* Netzwerke anwendbar sind. Das in der Diplomarbeit verwendete Kriterium ist das der *minimalen Beschreibungslänge* (MDL), welches ein Maß für die Netzwerkgröße in Abhängigkeit des resultierenden Fehlers darstellt. Es wurde ausgewählt, da die oben erwähnten Kriterien nur auf *supervised learning* Netzwerke angewandt werden können, das *MDL* Kriterium hingegen auch auf *unsupervised learning* Netzwerke.

Im Zuge dieser Diplomarbeit wird das Kriterium der *minimalen Beschreibungslänge* für die Modellselektion von *unsupervised* Netzwerken herangezogen und ein allgemeiner Algorithmus präsentiert, welcher die Netzwerkgröße und damit die Netzwerkkomplexität auf ein Minimum reduziert. Dabei geht der Algorithmus von einer großen Anzahl an Netzwerkeinheiten aus und reduziert in jedem seiner iterativen Schritte die Anzahl der Netzwerkeinheiten, bis eine minimale Beschreibung gefunden wird. Entscheidender Vorteil dieses Algorithmus ist seine Fähigkeit, während der Reduktion das Netzwerk zu trainieren. Man erhält dadurch ein trainiertes Netzwerk mit minimaler Größe.

Ein sehr häufiges Problem beim Training von neuralen Netzen ist die Existenz von gestörten Daten in der Trainingsmenge. In der Literatur nennt man diese Datenpunkte *outlier*, da sie nicht der wahren Verteilung der Daten entsprechen. Diese *outlier* können neben dem Lernen einer verfälschten Funktion auch zu einem zu großen Netzwerk führen, da zusätzliche Netzwerkeinheiten benötigt werden, um diese Ungenauigkeiten mitzulernen. Deshalb wurde der Algorithmus um die Fähigkeit erweitert, *outlier* zu detektieren, wodurch er bezüglich *outlier* robust wird.

Um die Verwendbarkeit dieses Ansatzes zu demonstrieren, werden in dieser Diplomarbeit beispielhafte Anwendungen präsentiert. Die Anwendungen sind:

Vektorquantisierung: Ziel der Vektorquantisierung ist es, die Zentren von *Datencluster* (Referenzvektoren) zu finden. Die meisten Algorithmen für Vek-

torquantisierung verlangen eine fixe vorgegebene Anzahl an zu positionierenden Referenzvektoren. Nachdem Vektorquantisierungsalgorithmen meist den quadratischen Fehler minimieren, liefern sie bei gestörten Daten (*outlier*) nicht erwünschte Ergebnisse (Frigui und Krishnapuram [10]). Mit Hilfe des *MDL* Algorithmus soll die richtige Anzahl an *Datencluster* und die Positionen der Zentren der *Datencluster* gefunden werden. Die Detektion von *outlier* soll den *MDL* Algorithmus unempfindlich gegenüber Störungen der Trainingsmenge machen. Weiters soll das *MDL* Kriterium sowohl auf *crisp/hard* als auch auf *fuzzy Clustering* Algorithmen angewendet werden.

Datenkompression: Die bei der Vektorquantisierung gefundenen *Datencluster* repräsentieren meist eine hohe Dichte an Information (Anhäufung von Datenpunkten). Die einzelnen Teilinformationen (Datenpunkte) repräsentieren dabei sehr ähnliche Informationen. Dieses Wissen kann helfen, Datenkompressoren zu entwickeln und die Information kompakt (speichersparend) zu beschreiben. Mit Hilfe der Vektorquantisierung soll gezeigt werden, wie man mit dem *MDL* Algorithmus Daten komprimieren kann. In unserem Fall beschränken wir uns auf die Bildkomprimierung.

RGB Farbsegmentierung: Ziel der Farbsegmentierung ist es, Pixel mit ähnlichen Farbnuancen zu Farbsegmenten zusammenzufassen. Dadurch könnte man bei einem Bild eines Menschen den Haarteil mit seinen unterschiedlichen Schwarznuancen zu einem Farbsegment der Haare zusammenfassen und so die Haare im Bild detektieren. Nun ist aber die Einteilung der für das Bild charakteristischen Farbwerte und dadurch die Anzahl an Farbsegmente in jedem Bild unterschiedlich. Betrachtet man die einzelnen Pixel (*RGB*-Tupel) eines *RGB* Farbbildes als Trainingsmenge, kann man mit Hilfe des *MDL* Algorithmus jene Farben suchen, welche am besten das *RGB* Farbbild charakterisieren.

Hand-Eye Koordination: In der Robotertechnik ist das Anfahren eines vorgegebenen Punktes eine typische Anwendung. Dabei wird dem Roboter durch manuelles Positionieren seines *Endeffektors* (z.B. Greifer) diese Position vorgegeben und die entsprechenden Roboterparameter abgespeichert. Diesen Vorgang nennt man auch *teach in*. Im Laufe der Zeit kann es, aufgrund von Last- und Verschleißerscheinungen, nicht mehr möglich sein, mit den abgespeicherten Daten die vorgegebenen Positionen zu erreichen. Martinetz [25] präsentierte eine adaptive Robotersteuerung, welche den *Endeffektor* über 2 *CCD* Kamera beobachtet und mit Hilfe der Information aus den beiden Kame-

ras die Ansteuerinformation des Roboterarmes mit Hilfe eines neuralen Netzes lernt. Dabei lernt das neurale Netz die auftretenden Positionierungenauigkeiten im Laufe des Betriebes mit, wodurch die anzufahrenden Positionen wieder erreicht werden. Mit Hilfe des *MDL* Algorithmus soll das von Martinetz [25] präsentierte Netzwerk in seiner Größe reduziert werden, ohne auf die gute Positioniergenauigkeit zu verzichten. Weiters soll das verwendete *Extended Neural Gas* Modell (*winner-takes-most* Philosophie) auch durch das Modell der *Local Linear Maps* (*winner-takes-all* Philosophie) mit Nachbarschaftskooperation ausgetauscht und deren Ergebnisse verglichen werden. Den Vorgang visuelle Repräsentationen eines Objektes (*Endeffektor*) in entsprechende Ansteuerinformationen (Gelenkwinkel) umzurechnen, nennt am auch *Hand-Eye Koordination*.

1.2 Überblick

Kapitel 2 beschäftigt sich mit den Grundlagen der verwendeten neuralen Netzwerke und Algorithmen. Dabei gibt Abschnitt 2.1 eine Einführung in die Vektorquantisierung, welche bei allen verwendete Netzwerken als Grundlage dient. Abschnitt 2.2 erklärt die Funktionsweise des *k-means* Algorithmus und Abschnitt 2.3 die des *fuzzy k-means* Algorithmus. Der *Learning Vector Quantization* Algorithmus wird als *online* Version des *k-means* Algorithmus in Abschnitt 2.4 präsentiert. Das dem *Extended Neural Gas* Modell (Abschnitt 5.4.1) als Grundlage dienende *Neural Gas* Modell wird in Abschnitt 2.5 diskutiert. Als Variante der zufälligen Initialisierung des *MDL* Algorithmus wird in Abschnitt 2.7 das *Growing Neural Gas* Modell beschrieben und Abschnitt 2.8 beschäftigt sich mit dem Modell der *Local Linear Maps*, welche, im Gegensatz zum *Extended Neural Gas* Modell, eine *winner-takes-all* Philosophie bei der Berechnung der Ausgabewerte für die Robotersteuerung verwendet.

Kapitel 3 zählt kurz die verschiedenen Kriterien der Modellkomplexität auf. Abschnitt 3.1 führt in die statistische Lerntheorie ein, welche den Lernvorgang eines neuralen Netzwerkes beschreibt. Darauf aufbauend behandelt Abschnitt 3.2 die Minimierung des *empirischen Risikos*, welches zur Beschreibung der Vorgangsweise der *Structural Risk Minimization* (Abschnitt 3.2.2) führt. Abschnitt 3.3 beschreibt die Idee die Netzwerkgröße mittels *Regularization/Penalization* während des Trainings des neuralen Netzwerkes zu beeinflussen. Die Komplexitätskriterien *Akaike Information Criterion*, *Network Information Criterion* und *Minimum Des-*

cription Length Kriterium werden in Abschnitt 3.4 angeführt. Ganz grob wird in Abschnitt 3.5 auf *network growing* und *network pruning* Algorithmen eingegangen, bevor Abschnitt 3.6 die verschiedenen Methoden der Modellselektion zusammenfaßt.

Kapitel 4 beschäftigt sich mit der Spezifikation des *Minimum Description Length* Kriteriums für das *Clustering* von Daten. Abschnitt 4.1 gibt eine Motivation für die Verwendung des *MDL* Algorithmus für Vektorquantisierungsaufgaben. Abschnitt 4.2 stellt den *MDL* Algorithmus für die *harte* (*crisp*) Zuordnung von Datenpunkten zu ihrem nächsten *Clusterzentrum* (Referenzvektor) dar und Abschnitt 4.3 beschreibt den *MDL* Algorithmus für die *fuzzy* Zuordnung der Datenpunkte. Weiters beschreibt dieses Kapitel einige Methoden zur Verbesserung des Laufzeitverhaltens des *MDL* Algorithmus (Abschnitt 4.4) und stellt auch eine Wachstumskomponente vor (Abschnitt 4.5).

In **Kapitel 5** wird dann auf die einzelnen Anwendungsbeispiele eingegangen und die verwendeten Instanzierungen des *MDL* Algorithmus beschrieben. Auch werden zu den jeweiligen Anwendungen die erhaltenen Resultate präsentiert.

Kapitel 6 faßt die erhaltenen Ergebnisse zusammen und gibt einen kurzen Ausblick über wünschenswerte Erweiterungen des beschriebenen *MDL* Kriteriums.

Kapitel 2

Neurale Netze Grundlagen

Inhaltsangabe

2.1 Vektorquantisierung

Die Aufgabe der Vektorquantisierung ist, eine Menge $S = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ von Datenpunkten $\mathbf{x}_i \in \mathbb{R}^d, 1 \leq i \leq n$, mit Hilfe einer Menge $A = \{\mathbf{w}_1, \ldots, \mathbf{w}_m\}$, von Referenzvektoren $\mathbf{w}_j \in \mathbb{R}^d, 1 \leq j \leq m$, zu kodieren. Dabei ist die Menge A der Referenzvektoren kleiner als die der Menge S der Datenpunkte ($m < n$). Die Funktion $\zeta : S \to A$ berechnet die Werte (Positionen) der Referenzvektoren $\mathbf{w}_j$, indem sie das Fehlermaß $E(S, A)$ bei Kodierung der Datenpunkte $\mathbf{x}_i$ mittels der Referenzvektoren $\mathbf{w}_j$ minimiert. Das Fehlermaß $E(S, A)$ wird definiert als

$$E(S, A) = \int_{\mathbf{x} \in S} \epsilon(\mathbf{x}, A) p(\mathbf{x}) d\mathbf{x}, \tag{2.1}$$

wobei $\epsilon(\mathbf{x}, A)$ der Fehler bei Kodierung des Datenpunktes $\mathbf{x}$ unter Verwendung der Menge A der Referenzvektoren ist. Die Datenpunkte $\mathbf{x}$ sind unabhängig und identisch verteilt und besitzen eine Verteilungdichte der Auftrittswahrscheinlichkeit $p(\mathbf{x})$. Nachdem in der Praxis das einzige Wissen über die Wahrscheinlichkeitsverteilung $p(\mathbf{x})$ die Menge S ist, reduziert sich Gleichung (2.1) zu*

$$E(S, A) = \sum_{\mathbf{x} \in S} \epsilon(\mathbf{x}, A). \tag{2.2}$$

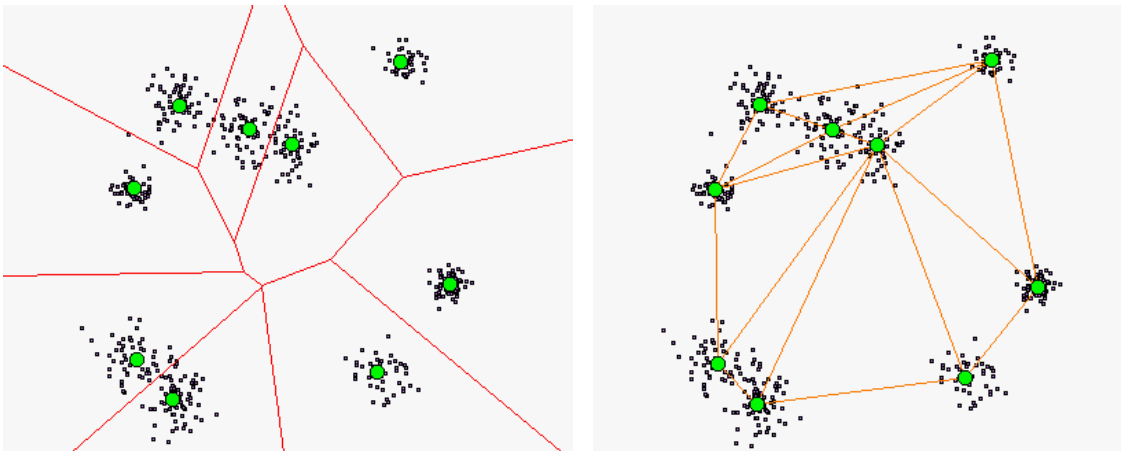
Für ein gegebenes A wird $E(S, A)$ minimal, wenn Datenpunkt $\mathbf{x}$ durch den ihm am nächsten gelegenen Referenzvektor $\mathbf{w}_i$, $i = \arg\min_{j \in \{1,\ldots,m\}} \epsilon(\mathbf{x}, A)$, kodiert wird. Typischerweise wird für $\epsilon(\mathbf{x}, A)$ die euklidische Metrik $||\mathbf{x} - \mathbf{w}_j||$ verwendet, wobei jede andere Metrik auch verwendet werden kann. Diese Metrik partitioniert die Menge S der Datenpunkte in disjunkte Teilmengen S_i, die man *Voronoi Regionen* nennt. Die Partitionierung der Menge S der Datenpunkte gilt nur, wenn man davon ausgeht, daß kein Datenpunkt zu zwei Referenzvektoren den gleichen Abstand besitzt und so auf der Grenze zweier benachbarter *Voronoi Regionen* liegt.

$$S_i = \{\mathbf{x} \in S | i = \arg \min_{j \in \{1,\ldots,m\}} \|\mathbf{x} - \mathbf{w}_j\|\}. \tag{2.3}$$

Abbildung 2.1(a) zeigt ein Beispiel einer Partitionierung einer Datenmenge, wobei auch die Grenzen der *Voronoi Regionen* eingezeichnet sind, welche das *Voronoi Diagramm* bilden. Der zum *Voronoi Diagramm* duale Graph nennt man *Delauny Triangulation*, die benachbarte Referenzvektoren verbindet (Abbildung 2.1(b)). Gleichung (2.2) führt zu folgender Definition des Fehlermaßes†, wobei $\mathbf{w}_i$ jener Referenzvektor ist, der die *Voronoi Region* S_i erzeugt.

*Tatsächlich müßte $E(S, A)$ in Gleichung (2.2) als $E(S, A) = \frac{1}{|S|} \sum_{\mathbf{x} \in S} \epsilon(\mathbf{x}, A)$ definiert sein. In der Praxis geht man von einer konstanten Größe der Trainingsmenge aus, wodurch der Term $\frac{1}{|S|}$ vernachlässigt werden kann, da $E(S, A)$ in beiden Fällen unter der selben Konfiguration minimal wird.

†Obwohl der Fehler der Summe der Fehlerquadrate (*sum-of-squares error*) als $E = \frac{1}{2} \sum_{i=1}^{m} \sum_{\mathbf{x} \in S_i} ||\mathbf{x} - \mathbf{w}_i||^2$ definiert ist, kann der Term $\frac{1}{2}$ vernachlässigt werden, da E in beiden Fällen unter der selben Konfiguration minimal wird.



(a) *Voronoi Regionen* **(b)** *Delauny Triangulation*

Abbildung 2.1: Vektorquantisierung einer aus 9 *Cluster* bestehenden Datenmenge. Die Anzahl Referenzvektoren wurde ebenfalls auf $m = 9$ gesetzt, wodurch die Referenzvektoren (Kreise) die Mittelpunkte der *Datencluster* markieren. Abbildung (a) zeigt die Grenzen der durch die Referenzvektoren aufgespannten *Voronoi Regionen* und Abbildung (b) zeigt die entsprechende *Delauny Triangulation*.

$$E(S, A) = \sum_{i=1}^{m} \sum_{\mathbf{x} \in S_i} \epsilon(\mathbf{x}, A) = \sum_{i=1}^{m} \sum_{\mathbf{x} \in S_i} \|\mathbf{x} - \mathbf{w}_i\|^2. \tag{2.4}$$

Das Fehlermaß $E(S, A)$ erreicht sein absolutes Minimum, wenn gleich viele Referenzvektoren wie zu kodierende Datenpunkte verwendet werden. In diesem Fall wird jeder Datenpunkt durch einen Referenzvektor beschrieben, wodurch eine Kodierung der Datenpunkte hinfällig würde. Die Anzahl m der Referenzvektoren bestimmt demnach die Güte der Kodierung. Für die Bestimmung der Positionen der Referenzvektoren werden oft *unsupervised learning* Algorithmen verwendet, welche versuchen aus einer Menge von Daten Strukturen zu erkennen.

2.2 k-means Algorithmus

MacQueen [24] entwickelte den *k-means* Algorithmus für das *Clustering* von Daten. Der Algorithmus versucht dabei das Fehlermaß $E(S, A)$ aus Gleichung (2.4) zu minimieren, wobei die Anzahl m der Referenzvektoren fix vorgegeben wird. Die fix vorgegebene Anzahl an Referenzvektoren symbolisiert der *k-means* Algorithmus mit

1: Initialisiere die Menge der Referenzvektoren A mit zufälligen Werten durch zufälliges Ziehen von Datenpunkten aus der Menge S. Keinen Datenpunkt doppelt ziehen.
2: $k := 0$.
3: **repeat**
4: $\quad k := k + 1$.
5: $\quad$ Berechne *Voronoi Regionen* $S_i, \forall i \in \{1, \ldots, m\}$:
$\quad$ siehe Gleichung (2.3).
6: $\quad$ Schätze die Positionen der Referenzvektoren $\bar{\mathbf{w}}_i, \forall i \in \{1, \ldots, m\}$:
$\quad$ siehe Gleichung (2.5).
7: $\quad$ Berechne Änderungen der Positionen der Referenzvektoren:
$\quad \Delta w_i := ||\mathbf{w}_i - \bar{\mathbf{w}}_i||, \forall i \in \{1, \ldots, m\}$.
8: $\quad$ Korrigiere Positionen der Referenzvektoren:
$\quad \mathbf{w}_i := \bar{\mathbf{w}}_i, \forall i \in \{1, \ldots, m\}$.
9: **until** $(\Delta w_1 \leq \rho \wedge \ldots \wedge \Delta w_m \leq \rho) \vee (k > k_{\max})$.

Algorithmus 2.1: k-means

dem k in seinem Namen. Zu Beginn werden die Positionen $\mathbf{w}_i$ der Referenzvektoren auf zufällige Werte aus der Menge S der Trainingsdaten gesetzt und deren *Voronoi Regionen* S_i mit Hilfe Gleichung (2.3) bestimmt. In der Regel stimmen die zufälligen Positionen der Referenzvektoren nicht mit jenen überein, die das Fehlermaß $E(S, A)$ minimieren. Deswegen werden die Referenzvektoren der aktuellen *Voronoi Regionen* S_i neu geschätzt.

$$\bar{\mathbf{w}}_i = \frac{1}{|S_i|} \sum_{\mathbf{x} \in S_i} \mathbf{x}, \qquad \forall i \in \{1, \ldots, m\}, \tag{2.5}$$

wobei $\bar{\mathbf{w}}_i$ die geschätzte Position des Referenzvektors $\mathbf{w}_i$ der *Voronoi Region* S_i ist und sich aus dem Mittel aller in der *Voronoi Region* S_i befindlichen Datenpunkte errechnet. Die Kardinalität der *Voronoi Region* S_i wird mit $|S_i|$ angegeben. Anschließend setzt man die Positionen der Referenzvektoren $\mathbf{w}_i$ auf die geschätzten Werte $\bar{\mathbf{w}}_i$. Durch die neue Position des Referenzvektors $\mathbf{w}_i$ können Datenpunkte aus benachbarten *Voronoi Regionen* in die *Voronoi Region* S_i wechseln und umgekehrt. Dies erfordert eine Neubestimmung der *Voronoi Regionen* entsprechend Gleichung (2.3).

Die Neubestimmung der *Voronoi Regionen* und die anschließende Schätzung

der Positionen der Referenzvektoren wird iterativ solange fortgesetzt, bis jeder der einzelnen Referenzvektoren eine maximale Änderung ρ erfährt oder eine maximale Anzahl an Iterationsschritten durchlaufen wurde. Algorithmus 2.1 stellt den *k-means* Algorithmus übersichtlich dar. Aufgrund seines iterativen Verhaltens und der zufälligen Initialisierung der Positionen der Referenzvektoren läuft der *k-means* Algorithmus Gefahr, in einem lokalen Minimum des Fehlermaßes $E(S, A)$ hängen zu bleiben. Weiters konvergiert der *k-means* Algorithmus nicht gegen die wahren Positionen der Referenzvektoren, sondern gegen das Mittel der geschätzten Positionen $\bar{\mathbf{w}}_i$.

Da für die Anwendung des *k-means* Algorithmus immer die gesamte Datenmenge S zur Verfügung stehen muß, wird der *k-means* Algorithmus in die Kategorie der *off-line* Lernalgorithmen eingeordnet. Kommt ein zusätzlicher Datenpunkt zur Datenmenge S hinzu, muß der gesamte Algorithmus neu durchlaufen werden, obwohl sich dadurch wahrscheinlich nur eine geringe Änderung der Positionen der Referenzvektoren ergibt.

2.3 Fuzzy k-means Algorithmus

Der *k-means* Algorithmus ordnet jedem Datenpunkt genau einen Referenzvektor zu, wodurch er auch als *crisp Clustering* oder *hard Clustering* Algorithmus bezeichnet wird. Das bedeutet, daß zwischen den Datenpunkten benachbarter *Voronoi Regionen* eine scharfe Grenze existiert. Bei der Verwendung von realen Daten gibt es aber diese scharfe Grenze nicht (Ripley [31]). Bezdek [2] hat deshalb den *k-means* Algorithmus modifiziert, um auf diesen Umstand Rücksicht nehmen zu können. Der *fuzzy k-means* Algorithmus, auch *fuzzy c-means* genannt, ordnet jeden Datenpunkt nicht mehr einem Referenzvektor zu, sondern führt für jeden Datenpunkt einen Zugehörigkeitsvektor $u_{\mathbf{x}} = \{u_{\mathbf{x}1}, \ldots, u_{\mathbf{x}m}\}$. Jedes Element $u_{\mathbf{x}i}$ des Zugehörigkeitsvektors gibt für jeden Datenpunkt $\mathbf{x}$ an, welche Zugehörigkeit er zum Referenzvektor $\mathbf{w}_i$ besitzt. Die Elemente $u_{\mathbf{x}i}$ werden *membership* Werte genannt. Für die einzelnen *membership* Werte müssen die folgenden Nebenbedingungen erfüllt sein

$$\sum_{i=1}^{m} u_{\mathbf{x}i} = 1, \qquad \forall \mathbf{x} \in S, \tag{2.6}$$

$$u_{\mathbf{x}i} \geq 0, \qquad \forall i \in \{1, \ldots, m\}, \forall \mathbf{x} \in S \tag{2.7}$$

1: Initialisiere die Menge der Referenzvektoren A mit zufälligen Werten durch zufälliges Ziehen von Datenpunkten aus der Menge S. Keinen Datenpunkt doppelt ziehen.
2: $k := 0$.
3: **repeat**
4: $k := k + 1$.
5: Berechne *membership* Werte $u_{\mathbf{x}i}, \forall i \in \{1, \ldots, m\}, \forall \mathbf{x} \in S$:
siehe Gleichung (2.8).
6: Schätze die Positionen der Referenzvektoren $\bar{\mathbf{w}}_i, \forall i \in \{1, \ldots, m\}$:
siehe Gleichung (2.9).
7: Berechne Änderungen der Positionen der Referenzvektoren:
$\Delta w_i := ||\mathbf{w}_i - \bar{\mathbf{w}}_i||, \forall i \in \{1, \ldots, m\}$.
8: Korrigiere Positionen der Referenzvektoren:
$\mathbf{w}_i := \bar{\mathbf{w}}_i, \forall i \in \{1, \ldots, m\}$.
9: **until** $(\Delta w_1 \leq \rho \wedge \ldots \wedge \Delta w_m \leq \rho) \vee (k > k_{\max})$.

Algorithmus 2.2: fuzzy k-means

Der Ablauf des *fuzzy k-means* Algorithmus ist dem des *k-means* Algorithmus ähnlich, jedoch hat beim *fuzzy k-means* Algorithmus die Bestimmung der *Voronoi Regionen* mit der Berechnung der *membership* Werte getauscht. Zunächst werden zufällige Werte für die Positionen $\mathbf{w}_i$ der Referenzvektoren generiert und die entsprechenden *membership* Werte berechnet.

$$u_{\mathbf{x}i} = \frac{1}{\sum_{j=1}^{m} \left(\frac{||\mathbf{x} - \mathbf{w}_i||}{||\mathbf{x} - \mathbf{w}_j||} \right)^{\frac{2}{p-1}}}. \tag{2.8}$$

Der Parameter p gibt den Grad der Unschärfe an. Für $p \to 1$ nähert sich das Ergebnis einem "scharfen" *Clusterergebnis* und für $p \to \infty$ streben die *membership* Werte $u_{\mathbf{x}i}$ gegen den Wert $\frac{1}{m}$. In der Praxis hat sich der Wert $p = 2$ bewährt (Ripley [31]). Bei der Abschätzung der Positionen der Referenzvektoren wird nun die gesamte Datenmenge S herangezogen.

$$\bar{\mathbf{w}}_i = \frac{\sum_{j=1}^{m} \sum_{\mathbf{x} \in S_j} (u_{\mathbf{x}i})^p \mathbf{x}}{\sum_{j=1}^{m} \sum_{\mathbf{x} \in S_j} (u_{\mathbf{x}i})^p} = \frac{\sum_{\mathbf{x} \in S} (u_{\mathbf{x}i})^p \mathbf{x}}{\sum_{\mathbf{x} \in S} (u_{\mathbf{x}i})^p}, \qquad \forall i \in \{1, \ldots, m\}. \tag{2.9}$$

Gleichung (2.9) zeigt, daß der *fuzzy k-means* Algorithmus die Positionen der

Referenzvektoren als gewichtetes Mittel aller Datenpunkte aus der Menge S bestimmt. Der *fuzzy k-means* Algorithmus minimiert demnach das gewichtete Fehlermaß $E(S, A)$

$$E(S, A) = \sum_{i=1}^{m} \sum_{\mathbf{x} \in S} (u_{\mathbf{x}i})^p \|\mathbf{x} - \bar{\mathbf{w}}_i\|^2. \tag{2.10}$$

Auch hier wird wieder ein iterativer Ablauf verwendet, um die Positionen der Referenzvektoren zu bestimmen. Das Abbruchkriterium ist wie beim *k-means* Algorithmus dann erreicht, wenn die Änderung der Positionen der Referenzvektoren unter einen Schwellwert ρ fallen oder die maximale Anzahl an Iterationsschritten überschritten wird. Der *fuzzy k-means* Algorithmus zählt ebenfalls zu den *off-line* Lernalgorithmen, da für die Berechnung der Werte $\mathbf{w}_i$ die gesamte Trainingsmenge S verwendet wird. Algorithmus 2.2 stellt den Ablauf des *fuzzy k-means* übersichtlich dar.

2.4 Learning Vector Quantization

Wie in Abschnitt 2.2 erwähnt, gehört der *k-means* Algorithmus zu der Klasse der *off-line* Lernalgorithmen, da immer die gesamte Trainingsmenge S zur Bestimmung der Referenzvektorpositionen hergezogen wird. Diese *off-line* Algorithmen haben den Nachteil, daß bei kontinuierlichen Datenströmen, welche häufig in der Signalverarbeitung auftreten, die Datenmenge S im Laufe der Zeit sehr groß werden kann. Kohonen [19] und [20] beschreibt mit dem *Learning Vector Quantization* (LVQ) Algorithmus eine *on-line* Variante des *k-means* Algorithmus, bei dem die Positionen der Referenzvektoren bei jedem Eintreffen eines neuen Datenpunktes (Signal) adaptiert werden. Für die Notation der Zeitabhängigkeit wird der Parameter t_i einführt, der den Zustand des Referenzvektors $\mathbf{w}_i$ zu den unterschiedlichen Zeitpunkten t_i angibt. Er beschreibt demnach die Anzahl Datenpunkte, für die der Referenzvektor $\mathbf{w}_i$ der nächste Referenzvektor war. Die gesamte Anzahl der Datenpunkte t ergibt sich demnach aus

$$t = \sum_{i=1}^{m} t_i. \tag{2.11}$$

Die Initialisierung (Zeitpunkt $t_i = 0, \forall i \in \{1, \ldots, m\}$) der Referenzvektoren

1: Initialisiere die Menge der Referenzvektoren A mit zufälligen Werten aufgrund der Datenverteilung $p(\mathbf{x})$.
2: Initialisiere $t_i := 0, \forall i \in \{1, \ldots, m\}$.
3: Initialisiere $t := 0$.
4: **loop**
5: $t := t + 1$.
6: Ziehe Datenpunkt $\mathbf{x}(t)$.
7: Bestimme Index s des nächsten Referenzvektors $\mathbf{w}_s$:
siehe Gleichung (2.12).
8: $t_s := t_s + 1$
9: Berechne Lernrate $\epsilon(t_s)$:
siehe Gleichung (2.14)
10: Berechne Änderung des nächsten Referenzvektors $\Delta\mathbf{w}_s(t_s)$:
siehe Gleichung (2.15)
11: Adaptiere nächsten Referenzvektor $\mathbf{w}_s(t_s)$
siehe Gleichung (2.16)
12: **end loop**

Algorithmus 2.3: Learning Vector Quantization

kann nun nicht mehr dadurch erfolgen, daß zufällige Werte aus der Datenmenge S gezogen werden, da S zu diesem Zeitpunkt leer ist. Man braucht in diesem Fall Wissen über die Verteilung $p(\mathbf{x})$ der Datenpunkte und wählt zufällige Werte für die Referenzvektorpositionen entsprechend dieser Verteilung $p(\mathbf{x})$ aus. Beim Eintreffen des Datenpunktes $\mathbf{x}(t)$ zum Zeitpunkt t ermittelt der *LVQ* Algorithmus den nächst gelegenen Referenzvektor $\mathbf{w}_s(t_s - 1)$ mit

$$s = \arg \min_{j \in \{1,\ldots,m\}} \|\mathbf{x}(t) - \mathbf{w}_j(t_j - 1)\|. \tag{2.12}$$

Nun adaptiert man den Referenzvektor $\mathbf{w}_s(t_s - 1)$ in Richtung des Datenpunktes $\mathbf{x}(t)$, wobei die Korrektur mit einer Lernrate $\epsilon(t_s)$ gedämpft wird. Alle anderen Referenzvektoren bleiben unverändert. Die Lernrate $\epsilon(t_s)$ beschreibt eine mit t_s abfallende, harmonische Folge.

$$t_s := t_s + 1. \tag{2.13}$$

$$\epsilon(t_s) = \frac{1}{t_s}. \tag{2.14}$$

$$\Delta \mathbf{w}_s(t_s) = \epsilon(t_s)(\mathbf{x}(t) - \mathbf{w}_s(t_s - 1)). \quad (2.15)$$

$$\mathbf{w}_s(t_s) = \mathbf{w}_s(t_s - 1) + \Delta \mathbf{w}_s(t_s). \quad (2.16)$$

Beim *LVQ* Algorithmus stehen demnach zu jedem Zeitpunkt t *quasi stationäre* Zustände der Positionen der Referenzvektoren fest. Ohne einer abfallende Lernrate $\epsilon(t_s)$ ist es für den *LVQ* Algorithmus nicht möglich, in einen stabilen Zustand zu kommen und die Positionen der Referenzvektoren würden nicht in einen Endzustand konvergieren. Ein *outlier* hätte eine große Änderung des nächsten Referenzvektors $\mathbf{w}_s(t_s - 1)$ zur Folge. Sei $\boldsymbol{\chi}_i(t_i)$ jener Datenpunkt $\mathbf{x}(t)$, der zum Zeitpunkt t den Referenzvektor $\mathbf{w}_i$ in den Zustand $\mathbf{w}_i(t_i)$ bringt. Der Referenzvektor $\mathbf{w}_i$ war demnach vor dem Zeitpunkt t für $t_i - 1$ Datenpunkte der nächste Referenzvektor. Der Referenzvektor $\mathbf{w}_i$ wurde bis zum Zeitpunkt t wie folgt adaptiert.

$$\mathbf{w}_i(0) = \text{zufällige Initialisierung entsprechend } p(\mathbf{x}). \quad (2.17)$$

$$\mathbf{w}_i(1) = \mathbf{w}_i(0) + \epsilon(1)(\boldsymbol{\chi}_i(1) - \mathbf{w}_i(0)) = \boldsymbol{\chi}_i(1). \quad (2.18)$$

$$\mathbf{w}_i(2) = \mathbf{w}_i(1) + \epsilon(2)(\boldsymbol{\chi}_i(2) - \mathbf{w}_i(1)) = \frac{\boldsymbol{\chi}_i(1) + \boldsymbol{\chi}_i(2)}{2}. \quad (2.19)$$

$$\vdots$$

$$\mathbf{w}_i(t_i) = \mathbf{w}_i(t_i - 1) + \epsilon(t_i)(\boldsymbol{\chi}_i(t_i) - \mathbf{w}_i(t_i - 1)) = \frac{\sum_{j=1}^{t_i} \boldsymbol{\chi}_i(j)}{t_i}. \quad (2.20)$$

Wie Gleichung (2.20) zeigt, berechnet sich die Position des Referenzvektors $\mathbf{w}_i$ aus dem arithmetischem Mittel jener Datenpunkte, die $\mathbf{w}_i$ als nächsten Referenzvektor zugeordnet bekommen haben. Einige der Datenpunkte $\boldsymbol{\chi}_i(t_i)$ können in einem der nachfolgenden Zeitpunkte nicht mehr in der *Voronoi Region* S_i des Referenzvektors $\mathbf{w}_i$ liegen, da sich die Grenzen der *Voronoi Regionen* im Laufe des Trainings verschieben. Gleichung (2.20) beweist auch, daß die Berechnung der Referenzvektorpositionen einer harmonischen Reihe entspricht, die divergent ist. Jedoch kann gezeigt werden, daß die Positionen der Referenzvektoren asymptotisch konvergieren (Fritzke [15]). Algorithmus 2.3 stellt den *LVQ* übersichtlich dar.

2.5 Neural Gas Modell

Der *k-means* und der *LVQ* Algorithmus werden auch als *hard competitive* oder *winner-takes-all* Lernalgorithmen bezeichnet, da jeder Referenzvektor um einen Da-

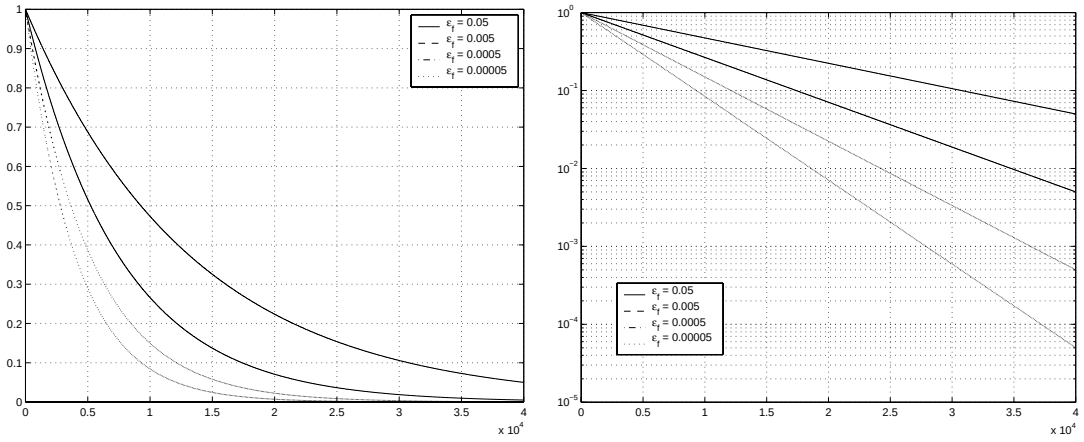


Abbildung 2.2: Darstellung der Funktion $\epsilon(t) = \epsilon_i \left(\frac{\epsilon_f}{\epsilon_i}\right)^{\frac{t}{t_{\max}}}$ mit verschiedenen Werten für $\epsilon_f \in \{0.05, 0.005, 0.0005, 0.00005\}$ ($\epsilon_i = 1.0, t_{\max} = 40000$). Abbildung (a) zeigt den Graphen $\epsilon(t)$ bei linearer y-Achse und Abbildung (b) zeigt den Graphen bei logarithmischer y-Achse. Das selbe Resultat gilt natürlich auch für die Funktion $\lambda(t)$.

tenpunkt "kämpft", jedoch der Datenpunkte seinem nächsten Referenzvektor zugeordnet wird. Diesen Referenzvektor nennt man *winner*. Bei sogenannten *soft competitive* oder *winner-takes-most* Lernalgorithmen lernt nicht nur der *winner*, sondern auch die restlichen Referenzvektoren. *Hard competitive* Lernalgorithmen haben den Nachteil, daß unterschiedliche Initialisierungen zu unterschiedlichen Ergebnissen führen. Weiters laufen sie aufgrund lokaler Adaptionen der Referenzvektoren Gefahr in lokalen Minima des Fehlermaßes $E(S, A)$ hängen zu bleiben. *Soft competitive* Lernalgorithmen versuchen diese Abhängigkeiten zu reduzieren, indem sie auch die restlichen Referenzvektoren mitlernen lassen.

Ein dem Kohonen Modell (Kohonen [19]) abgewandeltes Verfahren ist das *Neural Gas* Modell (Martinetz und Schulten [27]). Dieses *on-line* Verfahren berechnet zu einem Datenpunkt $\mathbf{x}(t)$ einen Ähnlichkeitsrang k_i für jeden Referenzvektor $\mathbf{w}_i$.

$$k_i = |W_{\mathbf{x}i}|, \qquad W_{\mathbf{x}i} = \left\{ \mathbf{w}_j \mid ||\mathbf{x} - \mathbf{w}_j|| < ||\mathbf{x} - \mathbf{w}_i||, \ \forall j \in \{1, \ldots, m\} \backslash \{i\} \right\} \quad (2.21)$$

Der Ähnlichkeitsrang gibt die Anzahl Referenzvektoren an, die zum Datenpunkt $\mathbf{x}$ näher sind, als der betrachtete Referenzvektor $\mathbf{w}_i$. Der nächste Referenz-

1: Initialisiere die Menge der Referenzvektoren A mit zufälligen Werten aufgrund der Datenverteilung $p(\mathbf{x})$.
2: Initialisiere $t := 0$.
3: **repeat**
4: Ziehe Datenpunkt $\mathbf{x}(t)$.
5: Bestimme Rangfolge k_i aller Referenzvektoren $\mathbf{w}_i$:
siehe Gleichung (2.21).
6: Bestimme Nachbarschaftswerte $h_\lambda(k_i)$ aller Referenzvektoren $\mathbf{w}_i$:
siehe Gleichung (2.22).
7: Berechne Lernrate $\epsilon(t)$:
siehe Gleichung (2.24).
8: Berechne Änderungen aller Referenzvektoren $\Delta\mathbf{w}_i$:
siehe Gleichung (2.25).
9: Adaptiere alle Referenzvektor $\mathbf{w}_i$:
$\mathbf{w}_i := \mathbf{w}_i + \Delta\mathbf{w}_i, \forall i \in \{1, \ldots, m\}$.
10: $t := t + 1$.
11: **until** $t = t_{\max}$.

Algorithmus 2.4: Neural Gas Modell

vektor $\mathbf{w}_s$ besitzt einen Ähnlichkeitsrang $k_s = 0$, der zweitnächste Referenzvektor $\mathbf{w}_r$ besitzt einen Ähnlichkeitsrang $k_r = 1$ und so weiter. Entsprechend des Ähnlichkeitsranges werden die Positionen der Referenzvektoren adaptiert. Je weiter ein Referenzvektor vom aktuellen Datenpunkt $\mathbf{x}(t)$ entfernt ist, je weniger wird der Referenzvektor in Richtung des Datenpunktes adaptiert. Wie stark die einzelnen Referenzvektoren durch den Ähnlichkeitsrang angesprochen werden, beschreibt die Nachbarschaftsfunktion $h_\lambda(k_i)$.

$$h_\lambda(k_i) = e^{-\frac{k_i}{\lambda(t)}}. \tag{2.22}$$

$$\lambda(t) = \lambda_i \left(\frac{\lambda_f}{\lambda_i}\right)^{\frac{t}{t_{\max}}}. \tag{2.23}$$

$$\epsilon(t) = \epsilon_i \left(\frac{\epsilon_f}{\epsilon_i}\right)^{\frac{t}{t_{\max}}}. \tag{2.24}$$

Die Nachbarschaftsfunktion $h_\lambda(k_i)$ ist eine mit der Zeit t exponentiell abfallende Funktion. Die Funktion $\lambda(t)$ gibt zu einem Zeitpunkt t die Nachbarschaftsreichweite des aktuellen Datenpunktes $\mathbf{x}$ an, die ebenfalls exponentiell abfällt. Mit

$\epsilon(t)$ wird wieder die Lernrate für den Adaptionsschritt notiert. Die beiden Funktionen für $\lambda(t)$ bzw. $\epsilon(t)$ liefern zum Zeitpunkt $t = 0$ die Werte $\lambda(0) = \lambda_i$ bzw. $\epsilon(0) = \epsilon_i$ und zum Zeitpunkt $t = t_{\max}$ die Werte $\lambda(t_{\max}) = \lambda_f$ bzw. $\epsilon(t_{\max}) = \epsilon_f$. Abbildung 2.2 zeigt den Kurvenverlauf der Funktion $\epsilon(t)$ für verschiedene Werte ϵ_f, wobei der selbe Kurvenverlauf für die Funktion $\lambda(t)$ gilt. Die Parameter ϵ_f und λ_f steuern demnach auch den exponentiellen Abfall der Funktionen $\epsilon(t)$ und $\lambda(t)$. Für die Adaption der einzelnen Referenzvektoren ergibt sich folgender Formalismus.

$$\Delta \mathbf{w}_i = \epsilon(t) h_\lambda(k_i)(\mathbf{x} - \mathbf{w}_i), \qquad \forall i \in \{1, \ldots, m\}. \tag{2.25}$$

Jedesmal, wenn ein neuer Datenpunkt behandelt wird, wird t um eins erhöht, wodurch t wieder die Anzahl der gesehenen Datenpunkte notiert. Algorithmus 2.4 zeigt, daß das Abbruchkriterium beim *Neural Gas* Modell die maximale Anzahl an Datenpunkte $t_{\max}$ ist.

2.6 Neural Gas Modell und Competitive Hebbian Learning

Die Aufgabe aller bisherigen Verfahren war es, die Positionen der Referenzvektoren zu bestimmen. Wie in Abschnitt 2.1 beschrieben, bestehen zwischen den einzelnen Referenzvektoren Nachbarschaftsbeziehungen. Die *Delauny Triangulation* geben durch Verbindungen zwischen Referenzvektoren an, ob 2 Referenzvektoren benachbart sind, d.h., ob die Grenzen ihrer *Voronoi Regionen* einander berühren. Mittels *Competitive Hebbian Learning* ist es nun möglich, die topologischen Zusammenhänge der Referenzvektoren zu lernen. Es handelt sich dabei nicht um einen Algorithmus zur Quantisierung der Trainingsmenge, sondern lediglich um einen Algorithmus, der die Verbindungen zwischen den Referenzvektoren generiert. Es werden beim *Competitive Hebbian Learning* keine Positionen der Referenzvektoren verändert (Martinetz und Schulten [27]; Martinetz [26]).

Der *Competitive Hebbian Learning* Algorithmus beruht auf der *Hebb'schen* Lernregel

$$\Delta \mathbf{w}_{ij} \propto z_i z_j, \tag{2.26}$$

welche besagt, daß die Änderung der Stärke einer Verbindung zwischen Referenzvektor $\mathbf{w}_i$ und Referenzvektor $\mathbf{w}_j$ proportional dem Produkt der Aktivierung der beiden Referenzvektoren ist. Beim *Neural Gas* Modell entspricht die Aktivierung z_i des Referenzvektors $\mathbf{w}_i$

$$z_i = h_\lambda(k_i) = e^{-\frac{k_i}{\lambda(t)}}. \tag{2.27}$$

Ein großer Wert für die Aktivierung z_i bedeutet, daß der Abstand $||\mathbf{x} - \mathbf{w}_i||$ klein ist. Das Produkt $z_i z_j$ wird dann maximal, wenn die Referenzvektoren $\mathbf{w}_i$ und $\mathbf{w}_j$ die beiden nächsten Referenzvektoren zum Datenpunkt $\mathbf{x}$ sind. Sei $\mathbf{w}_s$ der nächste Referenzvektor zum Datenpunkt $\mathbf{x}$ und $\mathbf{w}_r$ der zweitnächste Referenzvektor mit

$$s = \arg \min_{i \in \{1,\ldots,m\}} ||\mathbf{x} - \mathbf{w}_i||, \tag{2.28}$$

$$r = \arg \min_{i \in \{1,\ldots,m\} \setminus \{s\}} ||\mathbf{x} - \mathbf{w}_i||, \tag{2.29}$$

dann gilt demnach

$$z_s z_r \geq z_i z_j, \qquad \forall i, j \in \{1, \ldots, m\} \setminus \{s, r\}. \tag{2.30}$$

Die *Hebb'sche* Lernregel indiziert mit ihrem Maximum jene beiden Referenzvektoren, die durch den Datenpunkt $\mathbf{x}$ am stärksten aktiviert werden und demnach Nachbarn sein müssen. Um Verbindungen zwischen den einzelnen Referenzvektoren angeben zu können, verwendet man eine Verbindungsmatrix $\mathbf{B} = (b_{ij})$ mit Elementen $b_{ij} \in \{0, 1\}$. Da die Kanten ungerichtet sind, ist $\mathbf{B}$ symmetrisch. Der *Competitive Hebbian Learning* Algorithmus bestimmt in jedem Trainingsschritt die beiden nächsten Referenzvektoren $\mathbf{w}_s$ bzw. $\mathbf{w}_r$ und setzt eine Verbindung $b_{sr} = 1$ zwischen diese beiden. Martinetz und Schulten [28] zeigen, daß der *Competitive Hebbian Learning* Algorithmus bei $t \to \infty$ in der Verbindungsmatrix $\mathbf{B}$ die Repräsentation der *induzierten Delauny Triangulation* gelernt hat. Der Term *induziert* soll ausdrücken, daß die Ermittlung der *Delauny Triangulation* und der *Voronoi Regionen* nur aufgrund der bekannten Datenpunkte erfolgt und nicht über die Verteilung $p(\mathbf{x})$. Demnach sind die beiden *Voronoi Regionen* S_i und S_j nur dann benachbart, wenn es einen Datenpunkt $\mathbf{x}$ gibt, der $\mathbf{w}_i$ und $\mathbf{w}_j$ als nächsten und zweitnächsten Referenzvektor

1: Initialisiere die Menge der Referenzvektoren A mit zufälligen Werten aufgrund der Datenverteilung $p(\mathbf{x})$.
2: Initialisiere Verbindungsmatrix: $\mathbf{B} = \mathbf{0}$
3: Initialisiere $t := 0$.
4: **repeat**
5: Ziehe Datenpunkt $\mathbf{x}(t)$.
6: Bestimme nächsten Referenzvektoren $\mathbf{w}_s$:
siehe Gleichung (2.28).
7: Bestimme zweitnächsten Referenzvektoren $\mathbf{w}_r$:
siehe Gleichung (2.29).
8: Wenn noch keine Verbindung $\overline{\mathbf{w}_s\mathbf{w}_r}$ existiert, erzeuge sie:
$b_{sr} = b_{rs} = 1 \leftrightarrow \exists\, \overline{\mathbf{w}_s\mathbf{w}_r}$.
9: $t := t + 1$.
10: **until** $t = t_{\max}$.

Algorithmus 2.5: Competitive Hebbian Learning

besitzt oder umgekehrt. Deswegen nennt man diese *Voronoi Regionen* auch *maskierte Voronoi Regionen*. Algorithmus 2.5 stellt den *Competitive Hebbian Learning* Algorithmus kurz dar.

In der obigen Darstellung des *Competitive Hebbian Learning* Algorithmus, wurde angenommen, daß die Positionen der Referenzvektoren schon bestimmt wurden. Dabei wurde einfach noch einmal die Menge der Datenpunkte S durchgegangen und die entsprechende *induzierte Delauny Triangulation* ermittelt. Nun gibt es aber auch die Möglichkeit den *Competitive Hebbian Learning* Algorithmus parallel zu einem Quantisierungsalgorithmus ablaufen zu lassen. Martinetz und Schulten [27] bzw. [28] beschreiben, wie diese beiden Verfahren kombiniert werden können. Das Problem bei der kombinierten Variante ist, daß Verbindungen, welche am Beginn des Trainings entstehen, im Laufe des Trainings nicht mehr gültig sein können. Dies liegt daran, daß sich die Positionen der Referenzvektoren während des Trainings ändern und damit auch eine Änderung der Nachbarschaftsbeziehung auftreten kann.

Das Problem der ändernden Nachbarschaftsbeziehungen kann man dadurch in den Griff bekommen, daß man für jede Verbindung b_{ij} ihr Alter age_{ij} angibt. Überschreitet eine Verbindung das maximale Alter $\text{age}_{\max}(t)$, wird sie entfernt. Das Auftreten eines Datenpunktes $\mathbf{x}$ setzt das Alter der Verbindung zwischen dem nächsten Referenzvektor $\mathbf{w}_s$ und dem zweitnächsten Referenzvektor $\mathbf{w}_r$ wieder auf 0 (*refresh*), läßt aber alle anderen von Referenzvektor $\mathbf{w}_s$ ausgehenden Verbindungen

1: Initialisiere die Menge der Referenzvektoren A mit zufälligen Werten aufgrund der Datenverteilung $p(\mathbf{x})$.
2: Initialisiere Verbindungsmatrix: $\mathbf{B} = \mathbf{0}$
3: Initialisiere $t := 0$.
4: **repeat**
5: Ziehe Datenpunkt $\mathbf{x}(t)$.
6: Bestimme Rangfolge k_i aller Referenzvektoren $\mathbf{w}_i$:
siehe Gleichung (2.21).
7: Bestimme Nachbarschaftswerte $h_\lambda(k_i)$ aller Referenzvektoren $\mathbf{w}_i$:
siehe Gleichung (2.22).
8: Berechne Lernrate $\epsilon(t)$:
siehe Gleichung (2.24)
9: Berechne Änderungen aller Referenzvektoren $\Delta\mathbf{w}_i$:
siehe Gleichung (2.25)
10: Adaptiere alle Referenzvektor $\mathbf{w}_i$:
$\mathbf{w}_i := \mathbf{w}_i + \Delta\mathbf{w}_i, \forall i \in \{1, \ldots, m\}$.
11: Bestimme nächsten Referenzvektoren $\mathbf{w}_s$:
siehe Gleichung (2.28).
12: Bestimme zweitnächsten Referenzvektoren $\mathbf{w}_r$:
siehe Gleichung (2.29).
13: Wenn noch keine Verbindung $\overline{\mathbf{w}_s\mathbf{w}_r}$ existiert, erzeuge sie:
$b_{sr} = b_{rs} = 1 \;\leftrightarrow\; \exists\, \overline{\mathbf{w}_s\mathbf{w}_r}$.
14: Setze Alter age_{sr} der Verbindung $\overline{\mathbf{w}_s\mathbf{w}_r}$ auf 0 (*refresh*):
siehe Gleichung (2.32).
15: Laß alle von Referenzvektor $\mathbf{w}_s$ ausgehenden Verbindungen altern:
siehe Gleichung (2.33).
16: Berechne maximales Alter $\text{age}_{\max}$ der Verbindungen:
siehe Gleichung (2.31).
17: Wenn eine Verbindung $\overline{\mathbf{w}_i\mathbf{w}_j}$ das maximale Alter $\text{age}_{\max}$ überschreitet, lösche Verbindung:
$b_{ij} = b_{ji} = 0 \;\leftrightarrow\; \nexists\, \overline{\mathbf{w}_i\mathbf{w}_j}$.
18: $t := t + 1$.
19: **until** $t = t_{\max}$.

Algorithmus 2.6: Neural Gas Modell und Competitive Hebbian Learning

altern. Die Menge aller zu Referenzvektor $\mathbf{w}_i$ benachbarter Referenzvektoren wird mit D_i notiert.

$$\begin{aligned} \mathrm{age}_{\max}(t) &= \mathrm{age}_i \Big(\frac{\mathrm{age}_f}{\mathrm{age}_i}\Big)^{\frac{t}{t_{\max}}}. && (2.31)\\ \mathrm{age}_{sr} &= 0. && (2.32)\\ \mathrm{age}_{si} &:= \mathrm{age}_{si} + 1, \qquad \forall \mathbf{w}_i \in D_s. && (2.33)\\ D_s &= \{\mathbf{w}_i \in A \backslash \{\mathbf{w}_s\} \mid b_{si} = 1\}. && (2.34) \end{aligned}$$

Der Wert für das maximale Alter einer Verbindung ist zeitabhängig. Das Alter beginnt mit dem initialen Wert age_i und steigt exponentiell auf den Wert age_f. Es ist ausreichend, nur die vom nächsten Referenzvektor ausgehenden Verbindungen altern zu lassen, wenn man annimmt, daß alle Referenzvektoren im Schnitt gleich oft als *winner* hervorgehen. Zu Beginn des Trainings müssen relativ oft Verbindungen gelöscht werden, da hier eine große Änderung der Referenzvektorpositionen auftritt. Gegen Ende des Trainings ändern sich die Positionen der Referenzvektoren sehr wenig, wodurch die Verbindungen zwischen den Referenzvektoren sehr stabil bleiben. Um dem Rechnung zu tragen, wird das maximale Alter der Verbindungen im Laufe des Trainings erhöht. Kombiniert man nun das *Neural Gas* Modells mit dem *Competitive Hebbian Learning* Algorithmus und führt zusätzlich das *aging* der Kanten ein, entsteht ein *on-line* Algorithmus, der zur Erzeugung von *topologieerhaltenden Karten* verwendet werden kann (Algorithmus 2.6).

2.7 Growing Neural Gas Modell

Allen bisherigen Vektorquantisierungsalgorithmen mußte die Netzwerkgröße (Anzahl der Referenzvektoren) fix vorgegeben werden. Das *Growing Neural Gas* Modells (Fritzke [11]) ist eine Kombination des *Growing Cell Structures* Modell (Fritzke [12]) und des *Competitive Hebbian Learning* und initialisiert die Menge A der Referenzvektoren mit 2 Referenzvektoren. Während des Trainings werden mit Hilfe eines lokalen Fehlermaßes Referenzvektoren hinzugefügt. Jeder Datenpunkt $\mathbf{x}$ erhöht das Fehlermaß E_s des nächsten Referenzvektors $\mathbf{w}_s$ um

$$\Delta E_s = ||\mathbf{x} - \mathbf{w}_s||^2, \qquad (2.35)$$

wobei jedes Fehlermaß E_i, $i \in \{1, 2\}$, zu Beginn mit 0 initialisiert wurde. Im Intervall von η Trainingsschritten wird ein neuer Referenzvektor zur Menge A hinzugeführt. Sinnvoll ist es, diesen Referenzvektor an die Stelle zu positionieren, bei der der maximale Fehler auftritt. Zu diesem Zweck ermittelt man jenen Referenzvektor $\mathbf{w}_q$, der den maximalen, akkumulierten Fehler E_q und weiters jenen Referenzvektor $\mathbf{w}_f$, der den maximalen, akkumulierten Fehler E_f der direkten Nachbarn von $\mathbf{w}_q$ besitzt.

$$q = \arg \max_{i \in \{1,\ldots,m(t)\}} E_i. \tag{2.36}$$

$$f = \arg \max_{\mathbf{w}_i \in D_q} E_i. \tag{2.37}$$

Da die Netzwerkgröße zu verschiedenen Zeitpunkten t verschieden sein kann, notiert $m(t)$ die Netzwerkgröße zum Zeitpunkt t. Die Menge aller direkt benachbarter Referenzvektoren von $\mathbf{w}_i$ wird mit D_i angegeben (Gleichung (2.34)). Zwischen den beiden Referenzvektoren $\mathbf{w}_q$ und $\mathbf{w}_f$ wird nun der neue Referenzvektor $\mathbf{w}_n$ eingeführt. Dazu ist es auch notwendig, die Verbindung $\overline{\mathbf{w}_q\mathbf{w}_f}$ zu löschen und zwei neue Verbindungen $\overline{\mathbf{w}_n\mathbf{w}_q}$ bzw. $\overline{\mathbf{w}_n\mathbf{w}_f}$ zu erstellen.

$$\mathbf{w}_n = \frac{\mathbf{w}_q + \mathbf{w}_f}{2}. \tag{2.38}$$

$$b_{qf} = b_{fq} = 0. \tag{2.39}$$

$$b_{nq} = b_{qn} = 1. \tag{2.40}$$

$$b_{nf} = b_{fn} = 1. \tag{2.41}$$

Nachdem der neue Referenzvektor $\mathbf{w}_n$ eingefügt wurde, wird der lokale Fehler in diesem Bereich verringert, da $\mathbf{w}_n$ zusätzlich verwendet werden kann, die Datenpunkte in diesem Bereich zu beschreiben. Aus diesem Grund verringern sich die akkumulierten Fehler E_q und E_f um einen Faktor α. Der Fehler E_n des neuen Referenzvektors wird aus den beiden korrigierten Fehlermaßen E_q und E_f gemittelt.

$$\Delta E_q = -\alpha E_q. \tag{2.42}$$

$$\Delta E_f = -\alpha E_f. \tag{2.43}$$

$$E_n = \frac{E_q + E_f}{2}. \tag{2.44}$$

1: Initialisiere die Menge der Referenzvektoren A mit 2 Referenzvektoren und setze sie auf zufälligen Werte aufgrund der Datenverteilung $p(\mathbf{x})$.
2: Initialisiere Verbindungsmatrix $\mathbf{B} = \mathbf{0}$ und $t := 0$.
3: Initialisiere $E_i = 0$, $\forall i \in \{1, 2\}$.
4: **repeat**
5: Ziehe Datenpunkt $\mathbf{x}(t)$.
6: Bestimme nächsten Referenzvektoren $\mathbf{w}_s$: siehe Gleichung (2.28).
7: Bestimme zweitnächsten Referenzvektoren $\mathbf{w}_r$: siehe Gleichung (2.29).
8: Wenn noch keine Verbindung $\overline{\mathbf{w}_s\mathbf{w}_r}$ existiert, erzeuge sie:
$b_{sr} = b_{rs} = 1 \;\leftrightarrow\; \exists\, \overline{\mathbf{w}_s\mathbf{w}_r}$.
9: Setze Alter age_{sr} der Verbindung $\overline{\mathbf{w}_s\mathbf{w}_r}$ auf 0 (*refresh*): siehe Gleichung (2.32).
10: Akkumuliere Fehlermaß zu nächstem Referenzvektor: siehe Gleichung (2.35)
11: Adaptiere nächsten Referenzvektor $\mathbf{w}_s$ und alle seine topologischen Nachbarn D_s: siehe Gleichung (2.45) und (2.46).
12: Laß alle von Referenzvektor $\mathbf{w}_s$ ausgehenden Verbindungen altern: siehe Gleichung (2.33).
13: Wenn eine Verbindung $\overline{\mathbf{w}_i\mathbf{w}_j}$ das fix vorgegebene, maximale Alter $\text{age}_{\max}$ überschreitet, dann lösche diese Verbindung: $b_{ij} = b_{ji} = 0 \;\leftrightarrow\; \nexists\, \overline{\mathbf{w}_i\mathbf{w}_j}$.
14: Wenn Referenzvektoren keine Kanten besitzen, entferne diese Referenzvektoren: $\exists i$ mit $b_{ij} = b_{ji} = 0,\ \forall j \in \{1, \ldots, m(t)\} \rightarrow$ entferne $\mathbf{w}_i$.
15: **if** $\eta | t$ **then**
16: Bestimme Referenzvektor $\mathbf{w}_q$ mit maximal akkumulierten Fehler: siehe Gleichung (2.36).
17: Bestimme in der Nachbarschaft D_q Referenzvektor $\mathbf{w}_f$ mit maximal akkumulierten Fehler: siehe Gleichung (2.37).
18: Füge neuen Referenzvektor $\mathbf{w}_n$ in A ein: siehe Gleichung (2.38).
19: Lösche alte Verbindung und setze neue für $\mathbf{w}_n$: siehe Gleichung (2.39), (2.40) und (2.41).
20: Verringere Fehlermaß für $\mathbf{w}_q$ und $\mathbf{w}_f$: siehe Gleichung (2.42) und (2.43).
21: Interpoliere Fehlermaß für neuen Referenzvektor $\mathbf{w}_n$:
siehe Gleichung (2.44).
22: **end if**
23: Verringere Fehlermaß E_i für alle Referenzvektoren:
$\Delta E_i = -\beta E_i,\ \forall\ i \in \{1, \ldots, m(t)\}$.
24: $t := t + 1$.
25: **until** $t = t_{\max}$.

Algorithmus 2.7: Growing Neural Gas Modell

Das *Growing Neural Gas* Modell unterscheidet sich vom *Neural Gas* Modell mit *Competitive Hebbian Learning* Algorithmus zusätzlich dadurch, daß das maximale Alter der Verbindungen $\text{age}_{\max}$ als vorgegebener Parameter des Algorithmus definiert ist und weiters, daß die Korrektur der Referenzvektorpositionen in jedem Trainingsschritt nur am nächsten Referenzvektor $\mathbf{w}_s$ und allen seinen topologischen Nachbarn D_s vorgenommen wird. Die Parameter ϵ_b und ϵ_n geben die Lernraten für diese Korrekturen an und werden als konstante Werte zu Beginn des Algorithmus definiert. Das Abbruchkriterium des *Growing Neural Gas* Modells wird wieder bei Erreichen des maximalen Trainingsintervalls $t_{\max}$ erfüllt. Eine detailliertere Darstellung des *Growing Neural Gas* Modell erfolgt in Algorithmus 2.7.

$$\Delta\mathbf{w}_s = \epsilon_b(\mathbf{x} - \mathbf{w}_s). \tag{2.45}$$

$$\Delta\mathbf{w}_i = \epsilon_n(\mathbf{x} - \mathbf{w}_i), \qquad \forall\mathbf{w}_i \in D_s. \tag{2.46}$$

2.8 Local Linear Maps

Die Aufgabe von *unsupervised learning* Netzwerken ist es, Strukturen aus einer Datenmenge zu erlernen. Im Falle der *Vektorquantisierung* sind es die Positionen der Referenzvektoren. Eine typische Aufgabe von *supervised learning* Netzwerken ist das Lernen von Funktionen (Funktionsapproximation). Sei S wieder die Menge der Trainingsdaten mit $S = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ und $\mathbf{x}_i \in \mathbb{R}^d, 1 \leq i \leq n$, die Trainingsdaten. Weiters sei $Y = \{\mathbf{y}_1, \ldots, \mathbf{y}_n\}$ die Menge der entsprechenden Funktionswerte $\mathbf{y}_i \in \mathbb{R}^l$, $1 \leq i \leq n$, der Abbildung $f : \mathbb{R}^d \to \mathbb{R}^l$, $f(\mathbf{x}) = \mathbf{y}$.

Das Modell der *Local Linear Maps* (Ritter [33]; Ritter, Martinetz und Schulten [34]) wurde entwickelt, um die vektorielle Abbildung $f : \mathbb{R}^d \to \mathbb{R}^l$ zu lernen. *Local Linear Maps* bestehen aus m Netzwerkknoten, wobei jeder dieser Knoten die zu lernende Funktion lokal durch eine lineare Abbildung approximiert (stückweise Interpolation). Für einen gegebenen Datenpunkt $\mathbf{x}$ berechnet das *Local Linear Maps* Netzwerk folgenden Funktionswert $\mathbf{y}^{(\text{net})}$

$$\mathbf{y}^{(\text{net})} = \mathbf{y}_s(\mathbf{x}) = \mathbf{A}_s(\mathbf{x} - \mathbf{w}_s) + \boldsymbol{\alpha}_s, \tag{2.47}$$

wobei $\mathbf{w}_s$ der zum Datenpunkt $\mathbf{x}$ nächste Referenzvektor ist (siehe Gleichung (2.28)). Die Matrix $\mathbf{A}_s$ ($d \times l$ Matrix) bildet die Steigung der lokalen Abbildung $\mathbf{y}_s(\mathbf{x})$ und

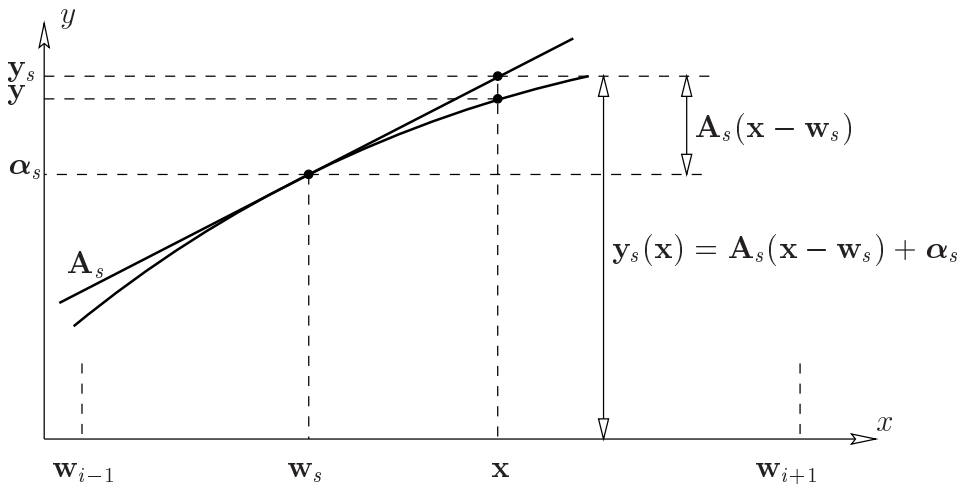


Abbildung 2.3: Funktionsapproximation mittels lokaler, linearer Abbildungen (*LLM*) für den eindimensionalen Fall. Das *LLM* Netzwerk berechnet aufgrund des Inputs $\mathbf{x}$ den Output $\mathbf{y}_s$. Der gewünschte Output wäre aber der Wert $\mathbf{y}$ gewesen. $\mathbf{w}_{i-1}$ und $\mathbf{w}_{i+1}$ sind die beiden zu $\mathbf{w}_s$ benachbarten Referenzvektoren.

$\boldsymbol{\alpha}_s$ den entsprechenden Funktionswert bei $\mathbf{x} = \mathbf{w}_s$ (siehe Abbildung 2.3). Das Modell der *Local Linear Maps* partitioniert mit seinen Referenzvektoren die Menge der Trainingsdaten und lernt für jede diese Partitionen die entsprechende, lineare Abbildung $f(\mathbf{x}) = \mathbf{y}$. Das heißt, daß in einem Trainingsschritt sowohl die Positionen der Referenzvektoren als auch die lokalen Abbildungen gelernt werden. Jedoch lernt nur der zum Datenpunkt $\mathbf{x}$ nächste Referenzvektor $\mathbf{w}_s$ (*winner-takes-all*).

$$\Delta\mathbf{w}_s = \epsilon_1(t)(\mathbf{x}-\mathbf{w}_s). \tag{2.48}$$

$$\Delta\boldsymbol{\alpha}_s = \epsilon_2(t)(\mathbf{y}-\mathbf{y}_s) + \mathbf{A}_s\Delta\mathbf{w}_s. \tag{2.49}$$

$$\Delta\mathbf{A}_s = \epsilon_3(t)\frac{1}{||\mathbf{x}-\mathbf{w}_s||^2}(\mathbf{y}-\mathbf{y}_s)(\mathbf{x}-\mathbf{w}_s)^T. \tag{2.50}$$

Die Lernraten $\epsilon_1(t)$, $\epsilon_2(t)$ und $\epsilon_3(t)$ werden wie in Gleichung (2.24) als exponentiell abfallende Funktionswerte definiert. Der Term $\mathbf{A}_s\Delta\mathbf{w}_s$ in Gleichung (2.49) berücksichtigt, daß bereits bei der Berechnung des Wertes $\mathbf{y}_s$ (Gleichung (2.47)) mit Hilfe des Terms $\mathbf{A}_s(\mathbf{x}-\mathbf{w}_s)$ der Wert $\mathbf{y}_s$ teilweise schon korrigiert wurde, nachdem der Referenzvektor $\mathbf{w}_s$ adaptiert wurde (Gleichung (2.48)). Fritzke [13] kombinierte das *Local Linear Maps* Modell mit dem Modell des *Growing Neural Gas*, um bei den *Local Linear Maps* nicht die Anzahl m der benötigten Netzwerkeinheiten definieren zu müssen. Algorithmus 2.8 stellt den Ablauf des *LLM* Modells übersichtlich dar.

1: Initialisiere die Menge der Referenzvektoren A mit zufälligen Werten durch zufälliges Ziehen von Datenpunkten aus der Menge S. Keinen Datenpunkt doppelt ziehen.
2: Initialisiere Vektoren $\boldsymbol{\alpha}_i$ und Matrizen $\mathbf{A}_i$, $\forall i \in \{1, \ldots, m\}$, auf zufällige Werte.
3: Initialisiere $t := 0$.
4: **repeat**
5: Ziehe Datenpunkt $\mathbf{x}(t)$ und entsprechenden Funktionswert $\mathbf{y}(t)$.
6: Bestimme nächsten Referenzvektor $\mathbf{w}_s$:
siehe Gleichung (2.28).
7: Bestimme Funktionswert des LLM Netzwerkes $\mathbf{y}_s$:
siehe Gleichung (2.47).
8: Berechne Lernraten $\epsilon_1(t)$, $\epsilon_2(t)$ und $\epsilon_3(t)$:
siehe Gleichung (2.24).
9: Adaptiere nächsten Referenzvektor $\mathbf{w}_s$:
siehe Gleichung (2.48)
10: Adaptiere Funktionswert $\boldsymbol{\alpha}_s$ bei $\mathbf{x} = \mathbf{w}_s$:
siehe Gleichung (2.49)
11: Adaptiere Steigung der linearen Abbildung $\mathbf{A}_s$:
siehe Gleichung (2.50)
12: $t := t + 1$.
13: **until** $t = t_{\max}$.

Algorithmus 2.8: Local Linear Maps

Kapitel 3

Modellselektion

Inhaltsangabe

Die Wahl eines guten Netzwerkmodells ist sehr entscheidend für die Qualität der erwarteten Ergebnisse. Meistens steht eine Menge an möglichen Modellen zur Auswahl, aus der man über Auswahlkriterien versucht, das ideale Modell zu bestimmen (*Modellselektion*). In unserem Fall beschränken wir uns bei der *Modellselektion*

auf die Netzwerkkomplexität (Größe) eines Modells, um die gestellte Aufgabe genügend gut lösen zu können. Dabei wurden unterschiedliche Kriterien entwickelt, welche meistens für *supervised learning* anwendbar sind. Das Prinzip der *Minimum Description Length* ist allerdings auch für *unsupervised learning* brauchbar und deshalb für *Clustering* Algorithmen sehr interessant.

3.1 Lerntheorie

Bei der Modellselektion von neuralen Netzwerken betrachtet man diese als *stochastische Input/Output-Systeme*. Dabei wurde zu den Werten $\mathbf{x}_i$ die Beobachtungen t_i getätigt $(1 \leq i \leq n)$, welche durch den Zusammenhang

$$t = g(\mathbf{x}) + \epsilon_g \tag{3.1}$$

erzeugt wurden, wobei $g(\mathbf{x})$ eine beliebige Funktion und ϵ_g der zufällige, erwartete Fehler ist (Haykin [18]). In diesem *regressiven Modell* wird die Funktion $g(\mathbf{x})$ als Erwartungswert der Beobachtung t für den Wert $\mathbf{x}$ definiert.

$$g(\mathbf{x}) = \mathbb{E}(t|\mathbf{x}). \tag{3.2}$$

Ein neurales Netzwerk, welches mit diesen Beobachtungen trainiert wurde, liefert für einen Wert $\mathbf{x}$ und dem trainierten Gewichtsvektor (Konfiguration) $\boldsymbol{\omega} \in \Omega$ folgenden Netzwerkoutput y.

$$y = f(\mathbf{x}, \boldsymbol{\omega}). \tag{3.3}$$

Die Adaptierung des Gewichtsvektor $\boldsymbol{\omega}$ erfolgt meistens in einem iterativen Prozeß, der durch den Trainingsfehler ϵ gesteuert wird.

$$\epsilon(\mathbf{x}, \boldsymbol{\omega}) = t - y. \tag{3.4}$$

Als Optimierungsaufgabe beim Training eines neuralen Netzes wird die Minimierung des mittleren, quadratischen Fehlers herangezogen, wodurch das Fehlermaß $E(\mathbf{x}, \boldsymbol{\omega})$ durch

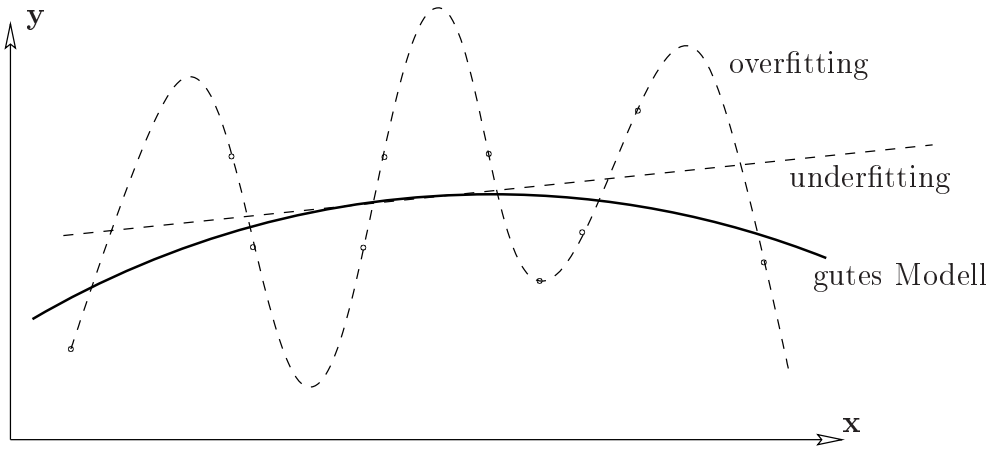


Abbildung 3.1: *Overfitting* und *Underfitting* von Datenpunkten.

$$
\begin{aligned}
E(\mathbf{x}, \boldsymbol{\omega}) &= \frac{1}{2}\mathbb{E}(\epsilon(\mathbf{x}, \boldsymbol{\omega})^2) = \frac{1}{2}\mathbb{E}((t-y)^2) && (3.5)\\
&= (\mathbb{E}(y) - t)^2 + \mathbb{E}((y - \mathbb{E}(y))^2) && (3.6)\\
&= \text{bias}^2 + \text{variance} && (3.7)
\end{aligned}
$$

beschrieben wird (Bishop [5]). Dabei wird der Fehler in zwei Teile zerlegt. Der *bias* gibt den Anteil des Fehlers an, der durch den Unterschied zwischen Netzwerkoutput y und Beobachtung t entsteht. Der *variance* Anteil gibt die Sensitivität des neuralen Netzes bezüglich Störungen der Daten an. Ideal wäre es, beide Teile minimieren zu können, das aber nicht möglich ist (*bias-variance Dilemma*; Geman, Bienenstock und Doursat [16]). Wählt man eine bestimmte Funktion $f(\mathbf{x}, \boldsymbol{\omega})$, welche ohne Rücksicht auf die Datenmenge ausgewählt wurde, so wird der *variance* Anteil verschwindend, der *bias* Anteil hingegen sehr groß. Paßt man auf der anderen Seite die Funktion $f(\mathbf{x}, \boldsymbol{\omega})$ so an, daß alle Datenpunkte exakt trainiert werden, wird der *bias* Anteil in der Umgebung der Datenpunkte verschwindend, der *variance* Anteil so groß wie die Störung der Daten.

Bei einem Netzwerkmodell mit zu hoher Netzwerkkomplexität spricht man von *overfitting* und bei einem Netzwerkmodell mit zu geringer Netzwerkkomplexität von *underfitting* (Abbildung 3.1). Weiters ist auch die Fehlercharakteristik bei der Bewertung des Netzwerkfehlers wichtig. Der *apparent error* ist jener Fehler, den man erhält, wenn man eine Validierung mit der Trainingsmenge vornimmt. Je größer die Anzahl Netzwerkparameter (Komplexität), desto geringer wird dieser Fehler, da das

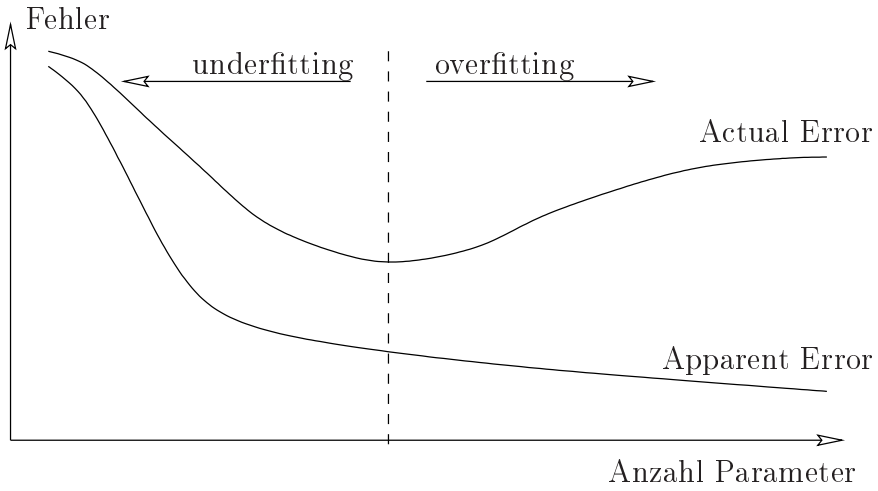


Abbildung 3.2: Abhängigkeit des *actual error* und *apparent error* von der Anzahl freier Modellparameter. Die optimale Modellkomplexität wird mit der strichlierten Linie angegeben.

Netzwerk beginnt die Trainingsdaten "auswendig" zu lernen. Der *actual error* ist der Fehler, den man erhält, wenn man mit einer eigenen Testmenge die Validierung des Netzwerkes vornimmt. Dabei wird die Generalisierfähigkeit des Netzwerkes ersichtlich. Abbildung 3.2 zeigt die Abhängigkeit dieses Fehlers von der Netzwerkkomplexität (Leisch [21]). Die strichlierte Linie zeigt die ideale Netzwerkkomplexität, bei der der *actual error* minimal ist. Steigert man die Netzwerkkomplexität, steigert sich auch der *actual error*, da die Generalisierfähigkeit des Netzwerkes reduziert wird.

Das Fehlermaß $E(\mathbf{x}, \boldsymbol{\omega})$ wird in der Literatur oft als *Verlustfunktion* (*loss*, *discrepancy*) bezeichnet. Im folgenden gehen wir nicht mehr von einem eindimensionalen Netzwerkoutput y aus, sondern von einem Outputvektor $\mathbf{y}$. Auch der *target* Wert von $\mathbf{x}$ wird dementsprechend als Vektor $\mathbf{t}$ notiert. Weiters definieren wir die *quadratische Verlustfunktion* wie folgt

$$E(\mathbf{x}, \boldsymbol{\omega}) = ||\mathbf{t} - \mathbf{y}||^2, \qquad \mathbf{t} = g(\mathbf{x}), \qquad \mathbf{y} = f(\mathbf{x}, \boldsymbol{\omega}). \tag{3.8}$$

Unter der Annahme, daß die Datenpunkte $(\mathbf{x}_i, \mathbf{t}_i), 1 \leq i \leq n$ (*supervised learning*), identisch und unabhängig verteilt sind, ergibt sich für die Wahrscheinlichkeitsdichte $p(\mathbf{x}, \mathbf{t})$

$$p(\mathbf{x}, \mathbf{t}) = p(\mathbf{t}|\mathbf{x})p(\mathbf{x}). \tag{3.9}$$

Die Aufgabe des Lernprozesses ist es, den Erwartungswert der *Verlustfunktion* $E(\mathbf{x}, \boldsymbol{\omega})$ über alle Konfigurationen $\boldsymbol{\omega} \in \Omega$ zu minimieren. Dieser Erwartungswert wird *Risikofunktion* (*risk functional*) genannt und ist definiert als

$$R(\boldsymbol{\omega}) = \int E(\mathbf{x}, \boldsymbol{\omega}) p(\mathbf{x}, \mathbf{t}) d\mathbf{x} d\mathbf{t}. \tag{3.10}$$

Die Berechnung der *Risikofunktion* $R(\boldsymbol{\omega})$ ist aufgrund der Tatsache, daß $p(\mathbf{x}, \mathbf{t})$ unbekannt ist, schwierig. Die einzige Information, die wir über die Verteilung der Datenpunkte haben, sind die Datenpunkte selber. Das Prinzip der *empirical risk minimization* umgeht dieses Problem dadurch, daß es keine Schätzungen der Wahrscheinlichkeitsverteilungen anstellt.

3.2 Empirical Risk Minimization

Zur Berechnung der *empirischen Risikofunktion* (*empirical risk functional*) wird lediglich die Trainingsmenge herangezogen.

$$R_{\text{emp}}(\boldsymbol{\omega}) = \frac{1}{n} \sum_{i=1}^{n} E(\mathbf{x}_i, \boldsymbol{\omega}). \tag{3.11}$$

Im Gegensatz zur *Risikofunktion* $R(\boldsymbol{\omega})$ ist es in der Praxis möglich, die *empirische Risikofunktion* $R_{\text{emp}}(\boldsymbol{\omega})$ über alle möglichen Konfigurationen $\boldsymbol{\omega} \in \Omega$ zu minimieren. Sei $\boldsymbol{\omega}_{\text{emp}}$ und $f(\mathbf{x}, \boldsymbol{\omega}_{\text{emp}})$ der Gewichtsvektor und die entsprechende Netzwerkfunktion, welche die *empirische Risikofunktion* minimal werden lassen und sei weiters $\boldsymbol{\omega}_{\text{o}}$ und $f(\mathbf{x}, \boldsymbol{\omega}_{\text{o}})$ der Gewichtsvektor und die entsprechende Netzwerkfunktion, welche die *Risikofunktion* $R(\boldsymbol{\omega})$ (*actual risk funktional*) aus Gleichung (3.10) minimieren. Es stellt sich nun die Frage, welche Bedingungen müssen gelten, daß $f(\mathbf{x}, \boldsymbol{\omega}_{\text{emp}})$ nahe genug an $f(\mathbf{x}, \boldsymbol{\omega}_{\text{o}})$ "herankommt".

Die *Risikofunktion* $R(\boldsymbol{\omega}_{\text{emp}})$ konvergiert in Wahrscheinlichkeit zum Minimum der *Risikofunktion* $R(\boldsymbol{\omega})$, wenn die Anzahl der Trainingsdaten n gegen unendlich geht und die *empirische Risikofunktion* $R_{\text{emp}}(\boldsymbol{\omega})$ gleichmäßig gegen die *Risikofunktion* $R(\boldsymbol{\omega})$ konvergiert. Gleichmäßige Konvergenz ist definiert als

$$\lim_{n \to \infty} \mathbb{P}[\sup_{\boldsymbol{\omega} \in \Omega} |R(\boldsymbol{\omega}) - R_{\text{emp}}(\boldsymbol{\omega})| > \epsilon] = 0, \tag{3.12}$$

wobei $\mathbb{P}$ die Wahrscheinlichkeit notiert. Diese Bedingung ist notwendig und hinreichend für das *principle of empirical risk minimization* (Cherkassky und Mulier [7]; Haykin [18]). Somit kann die *empirische Risikofunktion* R_{emp} anstatt der *Risikofunktion* $R(\boldsymbol{\omega})$ verwendet werden. Nachdem aber keine unendlich große Datenmenge zur Verfügung steht, wäre es wichtig, in Gleichung (3.12) eine obere Grenze α für die Wahrscheinlichkeit, daß $\sup_{\boldsymbol{\omega}\in\Omega} |R(\omega) - R_{\text{emp}}(\boldsymbol{\omega})| > \epsilon$ ist, angeben zu können.

$$\mathbb{P}[\sup_{\boldsymbol{\omega}\in\Omega} |R(\boldsymbol{\omega}) - R_{\text{emp}}(\boldsymbol{\omega})| > \epsilon] < \alpha, \tag{3.13}$$

3.2.1 VC Dimension

Die Grenze für die Gültigkeit der gleichmäßigen Konvergenz aus Gleichung (3.12) basiert auf der *Vapnik-Chervonenkis Dimension* (*VC-Dimension*; Vapnik und Chervonenkis [39]). Die *VC-Dimension* ist ein Maß für die Mächtigkeit einer Lernmaschine (neurales Netz), die Familie der Klassifikationsfunktionen darzustellen. Angenommen, ein neurales Netz soll die Familie der *dichotomy* Funktionen

$$\mathbb{D} = \{f(\mathbf{x}, \boldsymbol{\omega}), \boldsymbol{\omega} \in \Omega\}, \qquad f(\mathbf{x}, \boldsymbol{\omega}) \to \{0, 1\}, \tag{3.14}$$

darstellen. Dies sind jene Funktionen $f(\mathbf{x}, \boldsymbol{\omega})$, die jeden Datenpunkt $\mathbf{x}$ in eine von 2 Klassen $t \in \{0, 1\}$ einteilt. Somit gibt es $|\mathbb{D}| = 2^n$ Möglichkeiten die Datenpunkte $\mathbf{x}$ auf die beiden Klassen aufzuteilen. Die *VC-Dimension* h ist dann die maximale Anzahl an Datenpunkten über alle *dichotomy* Funktionen, die von der Lernmaschine (neurales Netz) ohne Fehler dargestellt werden können. Anders ausgedrückt, gibt die *VC-Dimension* die Anzahl Möglichkeiten an, wie eine Lernmaschine eine gegebene Datenmenge in 2 verschiedene Klassen einteilen kann. Mathematisch bedeutet dies

$$h = \max_{f(\mathbf{x},\boldsymbol{\omega})\in\mathbb{D}} |\mathbb{L}(f(\mathbf{x}, \boldsymbol{\omega}))|, \tag{3.15}$$

wobei $\mathbb{L}(f(\mathbf{x}, \boldsymbol{\omega})) \subseteq S$ die Menge der Datenpunkte ist, die von der Lernmaschine (neurales Netz) beim Lernen der Funktion $f(\mathbf{x}, \boldsymbol{\omega})$ richtig klassifiziert werden. Cherkassky und Mulier [7] geben Beispiele für die Berechnung der *VC-Dimension* für verschiedene Funktionen an. Für die Familie der *dichotomy* Funktionen gilt für eine Größe n der Trainingsmenge, der *VC-Dimension* h und einem $\epsilon > 0$

$$\mathbb{P}[\sup_{\boldsymbol{\omega}\in\Omega} |R(\boldsymbol{\omega}) - R_{\text{emp}}(\boldsymbol{\omega})| > \epsilon] < \left(\frac{2\ e\ n}{h}\right)^h e^{-\epsilon^2 n}, \tag{3.16}$$

wobei $R(\boldsymbol{\omega})$ der durchschnittliche Klassifikationsfehler (Fehlerrate), $R_{\text{emp}}(\boldsymbol{\omega})$ der Trainingsfehler (Anzahl Klassifikationsfehler während des Trainings) und e die Basis des natürlichen Logarithmus ist. Die *VC-Dimension* h kann deshalb als Maß für die Komplexität eines neuralen Netzwerkes angesehen werden, wobei die *VC-Dimension* h nicht in Verbindung mit der Anzahl freier Parameter stehen muß. Aus Gleichung (3.16) wird ersichtlich, daß die Anzahl n der Datenpunkte ein Vielfaches größer sein muß, als die *VC-Dimension* h ($n \gg h$). Ansonsten wird die Wahrscheinlichkeit α in Gleichung (3.13) für ein gegebenes ϵ sehr groß. Mit der Wahrscheinlichkeit von $(1-\alpha)$ gilt für alle Konfiguration $\boldsymbol{\omega} \in \Omega$ folgende Ungleichung.

$$R(\boldsymbol{\omega}) < R_{\text{emp}}(\boldsymbol{\omega}) + \epsilon(n, h, \alpha, R_{\text{emp}}), \qquad \forall \boldsymbol{\omega} \in \Omega, \tag{3.17}$$

wobei $\epsilon(n, h, \alpha, R_{\text{emp}}) = \epsilon$ als Konfidenzintervall für die Abweichung des *empirischen Risikos* $R_{\text{emp}}(\boldsymbol{\omega})$ und des *Risikos* $R(\boldsymbol{\omega})$ aufgefaßt werden kann. Eine genaue mathematische Definition von $\epsilon(n, h, \alpha, R_{\text{emp}})$ findet man in Haykin [18].

3.2.2 Structural Risk Minimization

Bezeichnen wir im folgenden mit $R_{\text{train}} = R_{\text{emp}}$ den Trainingsfehler und mit $R_{\text{gene}} = R$ den Generalisierungsfehler des trainierten Netzwerkes bei Anwendung von unbekannten Datenpunkten. Für eine bestimmte Größe $n > h$ der Trainingsmenge kann mit einer Wahrscheinlichkeit von $(1-\alpha)$ für alle Klassifikationsfunktionen $f(\mathbf{x}, \boldsymbol{\omega})$, analog zur Gleichung (3.17), folgender Zusammenhang angegeben werden (Haykin [18]).

$$R_{\text{gene}} < R_{\text{train}} + \epsilon(n, h, \alpha, R_{\text{train}}). \tag{3.18}$$

Abbildung 3.3 zeigt den garantierten Fehler ($R_{\text{train}} + \epsilon(n, h, \alpha, R_{\text{train}})$) in Abhängigkeit der *VC-Dimension* h bei einer bestimmten Anzahl n der Datenpunkte. Unterhalb des Minimums ist das Lernproblem *überbestimmt*, da die *VC-Dimension* für die Details der Datenpunkte zu gering ist. Oberhalb des Minimums ist das Lernproblem *unterbestimmt*, da die *VC-Dimension* für die Details der Datenpunkte

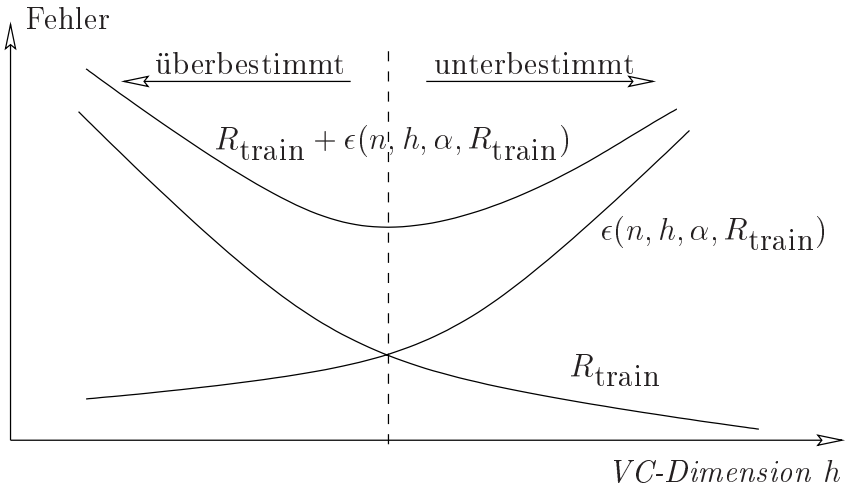


Abbildung 3.3: Abhängigkeit des garantierten Fehlers ($R_{\text{train}} + \epsilon(n, h, \alpha, R_{\text{train}})$) von der *VC-Dimension* h. Die optimale Modellkomplexität (Minimum des garantierten Fehlers) wird mit der strichlierten Linie angegeben.

zu hoch ist (siehe dazu auch *bias-variance Dilemma* in Abschnitt 3.1 und Abbildung 3.2). Die Methode der *structural risk minimization* (SRM) erlaubt es, die *VC-Dimension* einer Lernmaschine (neurales Netz) für die beste Generalisierfähigkeit zu bestimmen. Die Bedingung $n \gg h$ muß aber gelten.

Betrachten wir wieder die Familie der *dichotomy* Funktionen $\mathbb{D}$ und definieren hierarchisch angeordnete Untermengen dieser Familie

$$\mathbb{D}_k = \{f(\mathbf{x}, \boldsymbol{\omega}), \boldsymbol{\omega} \in \Omega_k\}, \qquad 1 \leq k \leq l, \tag{3.19}$$

sodaß für die Untermengen und deren *VC-Dimensionen* h_k gilt (Abbildung 3.4)

$$\mathbb{D}_1 \subset \mathbb{D}_2 \subset \ldots \subset \mathbb{D}_l, \tag{3.20}$$

$$h_1 < h_2 < \ldots < h_l. \tag{3.21}$$

SRM bestimmt für eine gegebene Datenmenge das optimale Modell in 2 Schritten.

1. Das *empirische Risiko* (Trainingsfehler) wird für jede Menge $\mathbb{D}_k$ bestimmt und jene Menge $\mathbb{D}_k^*$ ausgewählt, die den kleinsten Trainingsfehler aufweist.

2. Nun variiert man die *VC-Dimension* h, bis man den garantierten Fehler ($R_{\text{train}} + \epsilon(n, h, \alpha, R_{\text{train}})$) minimiert hat.

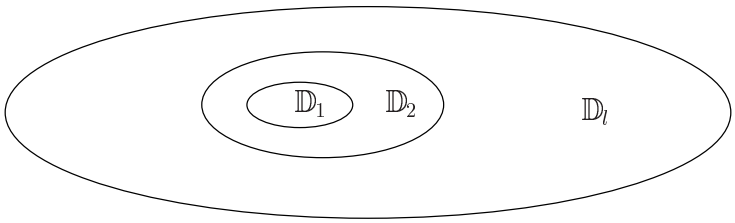


Abbildung 3.4: Verschachtelung der Mengen $\mathbb{D}_k$, $1 \leq k \leq l$.

Das bisherige *SRM* ist lediglich eine generelle Vorgangsweise und es existieren deshalb verschiedene Instanzierungen von *SRM*, die sich in der Wahl der Funktionsklassen $\mathbb{D}_k$ unterscheiden. Für die korrekte Anwendung von *SRM* muß gewährleistet sein, daß ein globales Optimum gefunden werden kann. Dies gilt zwar für die *Support Vector Machines*, jedoch nicht für die Veränderung der Anzahl *hidden neuron* bei *multilayer perceptrons* und *weight decay*. In der Praxis wird jedoch angenommen, daß das Finden des globalen Optimums für diese beiden letzten Instanzierungen gegeben ist.

Support Vector Machines: Angenommen es gibt 2 linear separable Mengen von Datenpunkten, d.h. es gibt eine Menge $\{\mathbf{x}_i, t_i\}$, $1 \leq i \leq n$, $t_i \in \{-1, 1\}$. Nun wäre es wünschenswert, eine Grenze (Hyperebene) zwischen diesen beiden Datenmengen zu legen, die zu beiden Datenmengen maximalen Abstand besitzt und zu dem noch so flach als möglich verläuft (siehe dazu Burges [6]). *Support Vector Machines* (SVM) liefern eine Hyperebene der Form $\boldsymbol{\omega}\mathbf{x}+b = 0$, wobei $\boldsymbol{\omega}$ normal zu dieser Ebene und $\frac{|b|}{||\boldsymbol{\omega}||}$ die Länge der Mittelsenkrechten zum Ursprung ist (Abbildung 3.5). Für den linear separablen Fall gelten für alle Trainingsdaten die Bedingungen

$$\mathbf{x}_i\boldsymbol{\omega} + b \geq \quad 1 \qquad \text{für } t_i = 1, \tag{3.22}$$

$$\mathbf{x}_i\boldsymbol{\omega} + b \leq \quad -1 \qquad \text{für } t_i = -1, \tag{3.23}$$

die zur Menge folgender Ungleichungen zusammengefaßt werden können.

$$t_i(\mathbf{x}_i\boldsymbol{\omega} + b) - 1 \geq 0, \qquad 1 \leq i \leq n. \tag{3.24}$$

Zwischen den beiden Hyperebenen $H_1 : \mathbf{x}_i\boldsymbol{\omega} + b = 1$ und $H_2 : \mathbf{x}_i\boldsymbol{\omega} + b = -1$, welche parallel sind und durch die äußersten Vertreter der entsprechenden Datenmenge verlaufen, befinden sich demnach keine Datenpunkte. Damit der

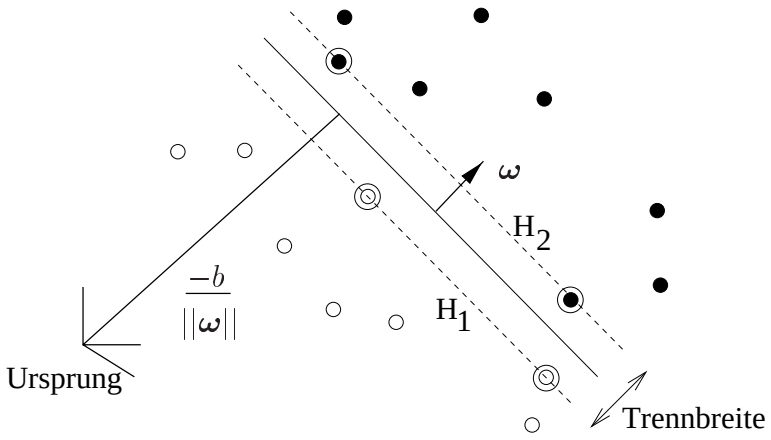


Abbildung 3.5: Lineare Hyperebene, welche zwei linear separable Datenmengen (gefüllte und nicht gefüllte Datenpunkte) trennt. Die *support vectors* sind zusätzlich eingekreist.

Abstand $\frac{2}{||\boldsymbol{\omega}||}$ zwischen diesen beiden Ebenen maximal ist, gilt es unter den Bedingungen aus Gleichung (3.24) den Term $||\boldsymbol{\omega}||^2$ zu minimieren. All jene Datenpunkte, die auf den Ebenen H_1 und H_2 liegen, werden *support vectors* genannt und sind die einzigen Datenpunkte, welche Einfluß auf das Ergebnis haben (umringte Datenpunkte in Abbildung 3.5). Führt man nun für die einzelnen Ungleichungen (3.24) positive Lagrangemultiplikatoren α_i, $1 \leq i \leq n$, ein, so ergibt sich die Optimierungsfunktion

$$L = \frac{1}{2}||\boldsymbol{\omega}||^2 - \sum_{i=1}^{n} \alpha_i t_i (\mathbf{x}_i \boldsymbol{\omega} + b) + \sum_{i=1}^{n} \alpha_i, \tag{3.25}$$

die es zu minimieren gilt. Dieses quadratische Optimierungsproblem kann mit herkömmlichen Methoden (z.B. *Karush-Kuhn-Tucker*) gelöst werden. Das Training von *SVM* findet immer ein globales Minimum der Optimierungsfunktion L. Für den Fall, daß die Datenpunkte nicht linear separabel sind, können *Kernelfunktionen* verwendet werden, um *Nonlinear Support Vector Machines* zu erzeugen.

Verändern der Anzahl *hidden neurons*: Die Untermengen werden entsprechend der Anzahl *hidden neurons* (z.B. bei *multilayer perceptrons* Netzwerken) erstellt. Die Aufgabe ist es, die optimale Anzahl *hidden neurons* für eine gegebene Datenmenge zu bestimmen.

Weight Decay: Die Untermengen werden aufgrund der euklidischen Norm

der Gewichtsvektoren $\boldsymbol{\omega}$ eingeteilt.

$$\mathbb{D}_k = \{f(\mathbf{x},\boldsymbol{\omega})|\ ||\boldsymbol{\omega}|| \leq c_k\},\ 1 \leq k \leq l, \qquad ||\boldsymbol{\omega}||^2 = \sum_j \omega_j^2, \quad (3.26)$$

$$c_1 < c_2 < \ldots < c_l. \quad (3.27)$$

Die Minimierung der *empirischen Risikofunktion* in jeder Untermenge $\mathbb{D}_k$ entspricht der Minimierung der erweiterten Risikofunktion

$$R(\boldsymbol{\omega},\lambda_k) = \frac{1}{n}\sum_{i=1}^{n} E(\mathbf{x},\boldsymbol{\omega}) + \lambda_k||\boldsymbol{\omega}||^2, \quad (3.28)$$

wobei λ_k der *regularization* Parameter ist (siehe Abschnitt 3.3), der zusammen mit der quadratischen euklidischen Norm eine Regulierung der Komplexität vornimmt.

3.3 Complexity Regularization und Penalization

Bisher wurde bei der Berechnung der *Risikofunktionen* immer nur die Abweichung der Netzwerkoutputs $\mathbf{y}$ von den zu lernenden Beobachtungen $\mathbf{t}$ verwendet. Bei der *complexity regularization* wird zusätzlich zum *empirischen Risikowert* eine von der Modellkomplexität abhängige Pönale (*penalty term*) hinzugefügt. Der unter der *complexity regularization* zu minimierende *Risikoterm* ergibt sich wie folgt (Cherkassky und Mulier [7]; Haykin [18]).

$$R_{\text{pen}}(\boldsymbol{\omega}) = R_{\text{emp}}(\boldsymbol{\omega}) + \lambda\Phi[f(\mathbf{x},\boldsymbol{\omega})], \quad (3.29)$$

wobei $\Phi[f(\mathbf{x},\boldsymbol{\omega})]$ eine nicht negative *penalty* Funktion ist und λ den *regularization* Parameter darstellt, der den *trade-off* zwischen $R_{\text{emp}}(\boldsymbol{\omega})$ und $\Phi[f(\mathbf{x},\boldsymbol{\omega})]$ steuert. Der Parameter λ und die Funktion $\Phi[f(\mathbf{x},\boldsymbol{\omega})]$ sind so zu wählen, daß der erwartete *Risikowert* $R_{\text{pen}}(\boldsymbol{\omega})$ minimiert werden kann. Die Funktion $\Phi[f(\mathbf{x},\boldsymbol{\omega})]$ wird einen geringen Wert returnieren, wenn die gelernte Funktion $f(\mathbf{x},\boldsymbol{\omega})$ eine geringe Komplexität besitzt. Ansonsten wird ein hoher Wert returniert, der zu zusätzlichen Kosten führt. Cherkassky und Mulier [7] bzw. Haykin [18] geben mögliche Definitionen der *regularization* Funktion an. Im folgenden werden kurz typische Implementierungen der *regularization* Theorie vorgestellt.

3.3.1 Weight Decay

Wie bereits in Gleichung (3.28) definiert, setzt man für die *regularization* Funktion die quadratische, euklidische Norm der Gewichtsvektoren* ein.

$$\Phi[f(\mathbf{x}, \boldsymbol{\omega})] = \frac{1}{2}||\boldsymbol{\omega}||^2. \tag{3.30}$$

Dabei wird davon ausgegangen, daß hohe Krümmungen der gelernten Funktion zu hohen Werten des Gewichtsvektors $\boldsymbol{\omega}$ führt. Gewichtsvektoren, welche geringen Einfluß auf den Netzwerkoutput haben, führen allerdings zu einem hohen Generalisierungsfehler, wenn willkürliche Inputwerte $\mathbf{x}$ verwendet werden. Die *regularization* mittels *weight decay* führt dazu, daß "unwichtige" Netzwerkelemente ω_i, Werte nahe bei 0 annehmen und so deren Einfluß minimiert wird.

3.3.2 Weight Elimination

Ein Verallgemeinerung der *weight decay* Methode ist die *weight elimination* Methode, welche folgende *regularization* Funktion besitzt.

$$\Phi[f(\mathbf{x}, \boldsymbol{\omega})] = \sum_j \frac{(\frac{\omega_j}{\omega_o})^2}{1 + (\frac{\omega_j}{\omega_o})^2}. \tag{3.31}$$

Der Parameter ω_o kann als freier Parameter der *weight decay* Methode angesehen werden. Der *penalty term* verändert sich asymptotisch mit $\frac{\omega_j}{\omega_o}$, wobei bei $|\omega_j| \ll \omega_o$ die Kosten gegen 0 gehen. Weiters soll das Netzwerkelement ω_j in diesem Fall entfernt werden, da es als "unzuverlässig" gilt. Im umgekehrten Fall $|\omega_j| \gg \omega_o$ geht der Kostenterm gegen 1, wodurch die Netzwerkeinheit ω_j als sehr wichtig gilt. Für den Fall, daß ω_o sehr groß gesetzt wird, ergibt sich als Spezialfall die *weight decay* Methode. Der *regularization* Parameter λ ist bei der *weight elimination* Methode sehr sensibel. Haykin [18] beschreibt heuristische Methoden zur Bestimmung von λ.

*Der Faktor $\frac{1}{2}$ kann auch in den *regularization* Parameter λ übernommen werden, wodurch sich auch wieder Gleichung (3.28) ergibt.

3.4 Complexity Criteria

Bisher wurde die Optimierungsaufgabe eines neuralen Netzwerkes über die Minimierung des erwarteten Verlustes (*Risiko*) definiert. Zur Berechnung des Verlustes wurde ein Fehlermaß zwischen aktuellem Netzwerkoutput $\mathbf{y}$ und erwartetem Netzwerkoutput $\mathbf{t}$ (Gleichung (3.8)) verwendet. In der Informationstheorie wird der Begriff der *Entropie* $H(p)$ eingeführt, der als Maß der Unsicherheit der Wahrscheinlichkeitsverteilung $p(\mathbf{x})$ definiert ist.

$$H(p) = -c \sum_{i=1}^{n} p(\mathbf{x}_i) \log p(\mathbf{x}_i) = -c\ \mathbb{E}(\log p(\mathbf{x})), \tag{3.32}$$

wobei c eine Konstante ist (Leisch [21]). Erweitert man dieses Modell, so kann man mit dem *Kullback-Leibler Informationsgehalt* (KLI) den Abstand zweier Verteilungsfunktionen G und F angeben. Wenn g und f die entsprechenden Dichtefunktionen* sind, dann ist der *KLI* definiert mit

$$I(g,f) = \begin{cases} \mathbb{E}_g[\log g(\mathbf{x}) - \log f(\mathbf{x})] & : \ G \ll F \\ \infty & : \ \text{sonst} \end{cases} \tag{3.33}$$

und gibt die Differenz zwischen der *Entropie* $H(g)$ und der *Cross-Entropie* $H(f|g)$ an.

$$I(g,f) = \mathbb{E}_g[\log g(\mathbf{x})] - \mathbb{E}_g[\log f(\mathbf{x})] = H(g) - H(f|g). \tag{3.34}$$

Der *KLI* bezieht sich immer auf die Verteilungsfunktion G und liefert ausschließlich den Wert 0, wenn die beiden Verteilungsfunktionen G und F identisch sind, ansonsten liefert der *KLI* einen positiven Wert. Der *KLI* kann als Maß für die Qualität der Approximation der Dichte g durch die Dichte f interpretiert werden. Die *Entropie* $H(g)$ in Gleichung (3.34) nimmt konstant einen Wert 1 an und deshalb kann entweder der *KLI* minimiert werden oder die *Cross-Entropie* maximiert werden. Die Minimierung der *Entropie* ist äquivalent der Maximierung des entsprechenden *log-likelihood* Schätzers von f. Der Lernprozeß eines neuralen Netzwerkes

*Im folgenden beschreiben die Funktionen g und f Wahrscheinlichkeitsdichten und nicht mehr die stochastischen Funktionen zur Berechnung der Werte $\mathbf{y}$ und $\mathbf{t}$, wie noch in den vorangegangenen Abschnitten.

kann auch als Schätzung der Dichte $p(\mathbf{x})$ der Datenpunkte durch die Dichte $f(\mathbf{x}, \boldsymbol{\omega})$ angesehen werden.

3.4.1 An Information Criterion – AIC

Angenommen die "wahre" Dichte $p(\mathbf{x})$ der Datenpunkte $\mathbf{x} \in S$ (Datenpunkte sind identisch und unabhängig verteilt) sei durch die Dichte $g(\mathbf{x}, \boldsymbol{\omega})$ beliebig gut approximierbar. Weiters sei $\boldsymbol{\omega}_o$ der "wahre" Gewichtsvektor, der die Dichtefunktion $g(\mathbf{x}, \boldsymbol{\omega}_o)$ als ideale Approximation der Dichte $p(\mathbf{x})$ definiert. Der vom neuralen Netzwerk ermittelte Gewichtsvektor $\widetilde{\boldsymbol{\omega}}$ gilt als Schätzer des Gewichtsvektor $\boldsymbol{\omega}_o$. Akaike [1] definiert die *Verlustfunktion* $E(\boldsymbol{\omega}_o, \widetilde{\boldsymbol{\omega}})$ und die *Risikofunktion* $R(\boldsymbol{\omega}_o, \widetilde{\boldsymbol{\omega}})$ wie folgt

$$E(\boldsymbol{\omega}_o, \widetilde{\boldsymbol{\omega}}) = 2I(g(\mathbf{x}, \boldsymbol{\omega}_o), g(\mathbf{x}, \widetilde{\boldsymbol{\omega}})), \tag{3.35}$$

$$R(\boldsymbol{\omega}_o, \widetilde{\boldsymbol{\omega}}) = \mathbb{E}_{\widetilde{\omega}} E(\boldsymbol{\omega}_o, \widetilde{\boldsymbol{\omega}}). \tag{3.36}$$

Die Bestimmung des *maximum likelihood* Schätzers $\hat{\boldsymbol{\omega}}$ ist äquivalent der Minimierung der *Verlustfunktion* $E(\boldsymbol{\omega}_o, \widetilde{\boldsymbol{\omega}})$, besitzt hingegen den Vorteil, daß er nicht vom Gewichtsvektor $\boldsymbol{\omega}_o$ abhängig ist.

$$\hat{\boldsymbol{\omega}} = \frac{1}{n} \arg\max_{\boldsymbol{\omega} \in \Omega} \lambda(\boldsymbol{\omega}), \qquad \lambda(\boldsymbol{\omega}) = \sum_{i=1}^{n} \log g(\mathbf{x}_i | \boldsymbol{\omega}). \tag{3.37}$$

Nachdem der *maximum likelihood* Schätzer $\hat{\boldsymbol{\omega}}$ nur bei $n \to \infty$ die Verlustfunktion $E(\boldsymbol{\omega}_o, \widetilde{\boldsymbol{\omega}})$ minimiert und bei endlicher Größe der Datenmenge einen hohen *bias* (hoher Trainingsfehler) zur Folge hat, gilt es, das *Risiko* $R(\boldsymbol{\omega}_o, \hat{\boldsymbol{\omega}})$ zu minimieren. Dieses *Risiko* ist jedoch wieder vom Gewichtsvektor $\boldsymbol{\omega}_o$, der unbekannt ist, abhängig. Das *AIC* (*an information criterion*, *Akaike information criterion*) liefert für hierarchische Modellklassen Ω_k, $1 \leq k \leq l$ ein Maß, welches minimal wird, wenn auch das *Risiko* $R(\boldsymbol{\omega}_o, \hat{\boldsymbol{\omega}}^k)$ minimal wird. Die hierarchische Struktur der Modellklassen definiert sich über

$$\Omega_1 \subset \Omega_2 \subset \ldots \subset \Omega_l, \tag{3.38}$$

$$m_1 < m_2 < \ldots < m_l, \tag{3.39}$$

wobei $\Omega_k \subset \mathbb{R}^{m_k}$ und m_k deswegen als Ordnung des $k-ten$ Modells angesehen werden kann (siehe Analogie zu *SRM* Abschnitt 3.2.2). Der Gewichtsvektor $\boldsymbol{\omega}^k \in \Omega_k$ kann deshalb wie folgt interpretiert werden.

$$\boldsymbol{\omega}^k = (\omega_1, \ldots, \omega_{m_k}, \omega_{m_{k+1}} = 0, \ldots, \omega_{m_l} = 0). \tag{3.40}$$

Das *AIC* ist definiert als

$$\mathrm{AIC}(k) = -2 \sum_{i=1}^{n} \log g(\mathbf{x}_i | \hat{\boldsymbol{\omega}}^k) + 2m_k, \tag{3.41}$$

wobei der Term $2m_k$ die Komplexität des Modells Ω_k berücksichtigt (siehe Analogie zu *regularization* Abbschnitt 3.3). Bei Polynome mit maximaler Ordnung l und Interpretation des Koeffizienten des Terms der $k-ten$ Ordnung durch ω_k, entspricht das Modell $\boldsymbol{\omega}^k$ einem Polynom $k-ter$ Ordnung und einem $m_k = k+1$. Das *AIC* liefert aber nur relative Werte und keinen absoluten Wert des erwarteten Verlustes (*Risiko*). Weiters funktioniert das *AIC* nur dann, wenn die Annahme der Dichte $g(\mathbf{x}, \boldsymbol{\omega})$ richtig war. Auch nimmt das *AIC* an, daß es einen idealen Gewichtsvektor $\boldsymbol{\omega}_o$ gibt. Netzwerke, welche Gefahr laufen, während des Lernens in einem lokalen Minimum der Verlustfunktion hängen zu bleiben, genügen dieser Annahme nicht, da nicht garantiert werden kann, daß $\boldsymbol{\omega}_o$ gefunden wird. Somit kann das *AIC* nicht für neurale Netze angewandt werden.

3.4.2 Network Information Criterion – NIC

Das *network information criterion* (NIC; Murata, Yoshizawa und Amari [30]) übernimmt die Idee des *AIC* und modifiziert es, daß es auch für neurale Netze angewandt werden kann. Dabei geht man nicht mehr von der Existenz des idealen Gewichtsvektor $\boldsymbol{\omega}_o$ aus, sondern versucht die gegebenen Daten so gut als möglich zu approximieren. Das heißt, man nimmt nicht mehr eine Dichtefunktion $g(\mathbf{x}, \boldsymbol{\omega})$ der Datenpunkte an. Weiters wurde im *AIC* die Minimierung der *Entropie* bzw. Maximierung des *log-likelihood* Schätzers als Optimierungsaufgabe der Verlustfunktion definiert, bei neuralen Netzen minimiert man meist den quadratischen Fehler.

Sei $p(\mathbf{x}, \mathbf{t})$ (Gleichung (3.9)) die wahre gemeinsame Dichte von Datenpunkt $\mathbf{x}$ und Beobachtung $\mathbf{t}$ und $f(\mathbf{t}|\mathbf{x}, \boldsymbol{\omega})$ die bedingte Dichte des Netzwerkoutputs. Den *Fehler* ϵ definiert man mit

$$\epsilon(\mathbf{x}, \mathbf{t}, p, \boldsymbol{\omega}) = d(\mathbf{x}, \mathbf{t}, p, \boldsymbol{\omega}) + s(\boldsymbol{\omega}), \tag{3.42}$$

wobei $d(\mathbf{x}, \mathbf{t}, p, \boldsymbol{\omega})$ ein Abstandmaß zwischen erwartetem Netzwerkoutput $\mathbf{t}$ und tatsächlichem Netzwerkoutput $\mathbf{y}$ ist. Der Term $s(\boldsymbol{\omega})$ ist ein optionaler *regularization* Term, der beispielsweise wie in Abschnitt 3.3.1 (*weight decay*) gesetzt werden kann.

$$d(\mathbf{x}, \mathbf{t}, p, \boldsymbol{\omega}) = \mathbb{E}_\epsilon ||\mathbf{t} - \widetilde{\mathbf{t}}||^2 = \int_{\mathbf{y}} ||\mathbf{t} - \mathbf{y}||^2 f(\mathbf{y}|\mathbf{x}, \boldsymbol{\omega}) d\mathbf{y}, \tag{3.43}$$

$$d(\mathbf{x}, \mathbf{t}, p, \boldsymbol{\omega}) = \log \frac{p(\mathbf{t}|\mathbf{x})}{f(\mathbf{t}|\mathbf{x}, \boldsymbol{\omega})}. \tag{3.44}$$

Gleichung (3.43) definiert den mittleren, quadratischen Fehler und Gleichung (3.44) die *Entropiedistanz*. Die *Verlustfunktion* und die *Risikofunktion* ergeben sich wie folgt, wenn die Punktepaare $(\mathbf{x}_i, \mathbf{t}_i)$ identisch und unabhängig verteilt sind.

$$L(p, \boldsymbol{\omega}) = \int \epsilon(\mathbf{x}, \mathbf{t}, p, \boldsymbol{\omega}) p(\mathbf{t}, \mathbf{x}) d\mathbf{x} d\mathbf{t}, \tag{3.45}$$

$$R(p, \widetilde{\boldsymbol{\omega}}) = \mathbb{E}_{\widetilde{\omega}} L(p, \widetilde{\boldsymbol{\omega}}), \tag{3.46}$$

wobei der Gewichtsvektor $\widetilde{\boldsymbol{\omega}}$ durch das Netzwerktraining bestimmt wurde. Das einzige Wissen über die Dichte $p(\mathbf{t}, \mathbf{x})$ ist jedoch die *empirische Dichte* $p_{\text{emp}}(\mathbf{t}, \mathbf{x})$.

$$p_{\text{emp}}(\mathbf{t}, \mathbf{x}) = \frac{1}{n} \sum_{i=1}^{n} \delta(\mathbf{x} - \mathbf{x}_i, \mathbf{t} - \mathbf{t}_i). \tag{3.47}$$

Beim *NIC* geht man, wie schon beim *AIC*, von einer hierarchischen Aufstellung der Modellklassen aus. Das *NIC* ist definiert als

$$\text{NIC}(k) = L(p_{\text{emp}}, \widetilde{\boldsymbol{\omega}}) + \frac{\text{tr}(P^k_{\text{emp}} (Q^k_{\text{emp}})^{-1})}{n}, \tag{3.48}$$

$$P^k_{\text{emp}} = \text{cov}[\nabla \epsilon(\mathbf{x}, \mathbf{t}, p_{\text{emp}}, \widetilde{\boldsymbol{\omega}}^k)], \tag{3.49}$$

$$Q^k_{\text{emp}} = \mathbb{E}_{p_{\text{emp}}}[\nabla \nabla \epsilon(\mathbf{x}, \mathbf{t}, p_{\text{emp}}, \widetilde{\boldsymbol{\omega}}^k)], \tag{3.50}$$

wobei die Gradienten in Abhängigkeit von $\widetilde{\boldsymbol{\omega}}^k$ berechnet werden. Die beiden Matrizen P^k_{emp} und Q^k_{emp} sind quadratisch mit Dimension m_k. Auch das *NIC* liefert

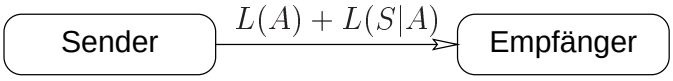


Abbildung 3.6: Übertragung der Datenmenge S. Zuerst wird das Modell A mit der Länge $L(A)$ übertragen und anschließend die Differenz zwischen Datenrepräsentation durch das Modell A und der Datenmenge S. Diese Länge beträgt $L(S|A)$.

relative Werte, wobei jenes Modell zu wählen ist, bei dem das *NIC* minimal ist. Im Fall der Definition der *Entropiedistanz* als Abstandsmaß zwischen erwartetem Netzwerkoutput $\mathbf{y}$ und tatsächlichem Netzwerkoutput $\mathbf{t}$ liefern das *NIC* und das *AIC* das selbe Ergebnis.

3.4.3 Minimum Description Length – MDL

Beim *AIC* und *NIC* wurde unter Minimierung einer *Risikofunktion* versucht ein Modell zu finden. Beim *minimum description length* (MDL) Prinzip (Rissanen [32]) geht man den umgekehrten Weg. Es existiert ein Modell zur Kodierung der Daten, wobei jenes Modell mit geringster Beschreibungslänge (*description length*) am meisten systematische Information aus den Daten herausgefunden hat und nur geringe zusätzliche Kodierungskosten für unsystematische Datenpunkte verwendet hat. Das *MDL* Prinzip ist konzeptionell identisch mit dem *BIC* (*bayes information criterion*, *Schwarz's criterion*).

Angenommen ein *Sender* möchte die Datenmenge S zu einem *Empfänger* übertragen und zwar mit *minimaler Beschreibungslänge*, um so die Ausnutzung der Bandbreite zu minimieren (Abbildung 3.6). In den meisten Fällen wird die Länge der Nachricht in bit gemessen. Die einfachste Kodierungsmöglichkeit ist die, der gleichen Kodierungslänge für jeden Datenpunkt. Wenn es hingegen in den Daten eine Systematik gibt, dann sollten die Daten entsprechend dieser Systematik kodiert werden, um so die Beschreibungslänge zu verringern. Sei A ein Modell, welches diese Systematik berücksichtigt und $L(A)$ die Kodierungslänge dieses Modells. Sei weiters $L(S|A)$ die Kodierungslänge der Fehlerinformation zwischen der Repräsentation der Daten durch das Modell A und der Daten S. Die gesamte Beschreibungslänge der Nachricht ist demnach

$$L(S(A)) = L(A) + L(S|A), \tag{3.51}$$

wobei $L(S|A)$ als Fehlerterm und $L(A)$ als Komplexitätsterm aufgefaßt werden kann. Ein einfaches Modell führt zu einer schlechten Approximation der Datenpunkte und deshalb zu einem hohen Fehlerterm. Auf der anderen Seite führt ein zu komplexes Modell ebenfalls zu einer hohen Beschreibungslänge, da der Komplexitätsterm groß wird. Intuitiv wird man annehmen, daß beim Minimum der gesamten Beschreibungslänge $L(S(A))$ die Approximation der Datenpunkte genügend gut und die Generalisierfähigkeit am besten ist (Bishop [5]).

Bei der "idealen" Kodierung der Datenpunkte wird für einen Datenpunkt $-\log \mathbb{P}(\mathbf{x})$ bits verwendet*. Wird ein parametrisches Modell verwendet, dann ergibt sich $\mathbb{P}(\mathbf{x}, \boldsymbol{\omega}) = f(\mathbf{x}, \boldsymbol{\omega})$. Es kann gezeigt werden, daß die minimale Beschreibungslänge der stochastischen Komplexität (SC; siehe dazu Leisch [21])

$$\mathrm{SC}(S) = \min_{m_k, \boldsymbol{\omega}^k} \mathrm{SC}(S, k) = \min_{m_k, \boldsymbol{\omega}^k} (-\sum_{i=1}^{n} \log f(\mathbf{x}_i, \boldsymbol{\omega}^k) + \frac{m_k}{2} \log(n)) \tag{3.52}$$

entspricht. m_k gibt wieder die Länge des Parametervektors $\boldsymbol{\omega}^k$ an. Der erste Term gibt die Kodierungslänge (in bits) der Datenpunkte bei Verwendung des Modells mit der Komplexität m_k und der zweite Term gibt die Kodierungslänge des Parametervektors $\boldsymbol{\omega}^k$ an. Der Vorteil des *MDL* Prinzips ist, daß die Beschreibungslänge ein absolutes Maß ist und deshalb keine hierarchische Struktur der Modelle vorliegen muß. Die unterschiedlichen Modelle können einfach miteinander verglichen werden. Weiters ist dieses Modell, wie wir später ausführlich diskutieren werden, auch für *unsupervised learning* Netzwerke geeignet.

3.5 Network Growing/Pruning

Um die Komplexität eines neurales Netzwerkes zu bestimmen, werden auch Algorithmen verwendet, die die Größe des Netzwerkes während des Trainings verändern können. Dabei unterscheidet man 2 Arten:

Network Growing: Bei *network growing* startet man mit einem Netzwerk minimaler Größe und fügt zusätzlich Netzwerkelemente hinzu, wenn die Strukturspezifikation des Netzwerkes nicht mehr erfüllt werden. Neben dem *GNG*

*Die Basis des Logarithmus hängt von der Basis des zugrundeliegenden Zahlensystems zur Kodierung ab. Bei Kodierung in bits ist der Logarithmus zur Basis 2 ($\log_2$) anzuwenden.

Modell (Abbschnitt 2.7) zählt auch die *cascade-correlation learning architecture* Methode (siehe dazu Bishop [5]) zu den Vertretern dieser Netzwerktypen.

Network Pruning: In diesem Fall startet man mit einem großen Netzwerk und entfernt Netzwerkelemente, die für die Erfüllung der gestellten Aufgabe nicht mehr notwendig sind. Die Methodik der *regularization* kann als *network pruning* Methodik angesehen werden.

Haykin [18] führt auch die Idee der *hessian matrix of error surface* an. Dabei wird die zweite Ableitung der Fehlerfunktion (*error surface*) als Information für den *trade-off* zwischen Netzwerkkomplexität und Trainingsfehler herangezogen. Die *optimal brain surgeon* Methode entfernt ein Gewicht ω_j, wenn eine Änderung des Wertes ω_j keinen Einfluß auf den Netzwerkoutput hat. Die Änderung ist von der Hessematrix abhängig. Die *optimal brain damage* Methode ist ein Spezialfall der *optimal brain surgeon* Methode und geht dabei von der Annahme aus, daß die Hessematrix eine Diagonalmatrix ist und deswegen nicht so rechenaufwendig wie die *optimal brain surgeon* Methode ist.

3.6 Zusammenfassung

Das Wesen der neuralen Netze als Lernmaschine ist, mittels Optimierungsmethoden (meistens Fehlerminimierung) die Werte der freien Parameter (Gewichte, Bias) zu bestimmen. Jedoch stellt sich die Frage nach der optimalen Größe (Anzahl freier Parameter) des neuralen Netzwerkes, die nicht von diesen Optimierungsalgorithmen beantwortet werden kann. Es hat sich gezeigt, daß in der Regel ein größeres Netzwerk einen kleineren Fehler zur Folge hat, wir aber mehr am Verhalten des Netzwerkes bei der Präsentation von neuen Daten interessiert sind (Generalisierfähigkeit). Wie man beim *bias-variance Dilemma* gesehen hat, lernen größere Netzwerke die Trainingsdaten genauer, jedoch zeigen sie große Unsicherheit bei der Präsentation von neuen Daten (hoher *bias* Anteil, *overfitting* des Modells). Im Fall kleinerer Netzwerke verhält es sich genau umgekehrt (hoher *variance* Anteil, *underfitting* des Modells).

Die interessante Eigenschaft eines neuralen Netzwerkes ist der Generalisierungsfehler, den man allerdings erst bestimmen muß. Eine Möglichkeit wäre, sowohl eine große Trainingsmenge für das Training als auch eine große Testmenge zur Bestimmung des Generalisierungsfehlers zu verwenden. Üblicherweise liegt nur eine

relativ kleine Datenmenge zur Verfügung, die man in Trainingsmenge und Testmenge zerteilen muß. Mit Hilfe dieser beiden kleinen Mengen ist es schwer möglich, einerseits ein Netzwerk gut zu trainieren und andererseits den Generalisierungsfehler zu bestimmen. In diesem Zusammenhang gilt es, die *cross validation* und die *boot strapping* Methode zu erwähnen (Haykin [18], Bishop [5]).

Eine andere Möglichkeit, den Generalisierungsfehler $\mathcal{E}_{\text{gen}}$ zu bestimmen, ist diesen aufgrund des Trainingsfehlers $\mathcal{E}_{\text{train}}$ und der Komplexitätspönale $\mathcal{P}$ des Modells zu schätzen.

$$\mathcal{E}_{\text{gen}} = \mathcal{E}_{\text{train}} + \mathcal{P}. \tag{3.53}$$

Die Komplexitätspönale $\mathcal{P}$ wird entsprechend dem Verfahren unter bestimmten Annahmen gewählt, wodurch sich auch die einzelnen Kriterien unterscheiden. Zusätzlich unterscheiden sich die *complexity criteria* von der *regularization* dadurch, daß sie den Trainingsfehler aus informationstheoretischer Sicht als Differenz der zu lernenden Dichtefunktion und der gelernten Dichte berechnen. Stimmt die Annahme der zu lernenden Dichtefunktion nicht mit der tatsächlichen Dichtefunktion überein, scheitern die *complexity criteria* bei der Bestimmung des optimalen Modells. Einzig das *MDL* Kriterium nimmt keine Schätzung des tatsächlichen Verteilungsmodells der Datenpunkte vor, sondern man definiert selbst ein (Kodierungs)Modell. Bei der *regularization* wird der *empirische* Trainingsfehler herangezogen, um den Term des Trainingsfehlers zu bestimmen. Auch das Prinzip der *SRM* funktioniert auf diese Art und Weise.

Nachdem die einzelnen Verfahren in der Lage sind, den Generalisierungsfehler $\mathcal{E}_{\text{gen}}$ zu schätzen, muß nur noch jenes Modell bestimmt werden, welches den besten *trade-off* zwischen Trainingsfehlers $\mathcal{E}_{\text{train}}$ und Komplexitätsterm $\mathcal{P}$ darstellt. Das *SRM* Modell und die *complexity criteria* generieren hierarchische Modellklassen und bestimmen jeweils die besten Vertreter. Anschließend wird jener Vertreter als bestes Modell ermittelt, welches den minimalen Generalisierungsfehler $\mathcal{E}_{\text{gen}}$ aufweist. *Pruning* und *growing* Methoden gehen von einem Netzwerk bestimmter Größe aus und entfernen bzw. fügen Netzwerkknoten hinzu. Dabei dient der Generalisierungsfehler als Entscheidungskriterium, die Größe eines neuralen Netzwerkes zu verändern. Als Vertreter der *pruning* Methoden kann die *regularization* angesehen werden.

Im folgenden Abschnitt wird ein iterativer *pruning* Algorithmus präsentiert, der mit Hilfe des *MDL* Prinzips ein Kriterium für die Entfernung eines Netzwerk-

knotens verwendet. Das *MDL* Kriterium erlaubt es, im Unterschied zu den anderen *complexity criteria*, verschiedene, nicht hierarchische Modelle zu vergleichen. Der Vorteil eines iterativen Algorithmus ist, daß nicht der gesamte Suchraum generiert werden muß, sondern man sich im Suchraum von Unterraum zu Unterraum bewegt. Nachteil ist jedoch, daß man Gefahr läuft, in einem lokalen Optimum hängen zu bleiben.

Kapitel 4

MDL und Clustering

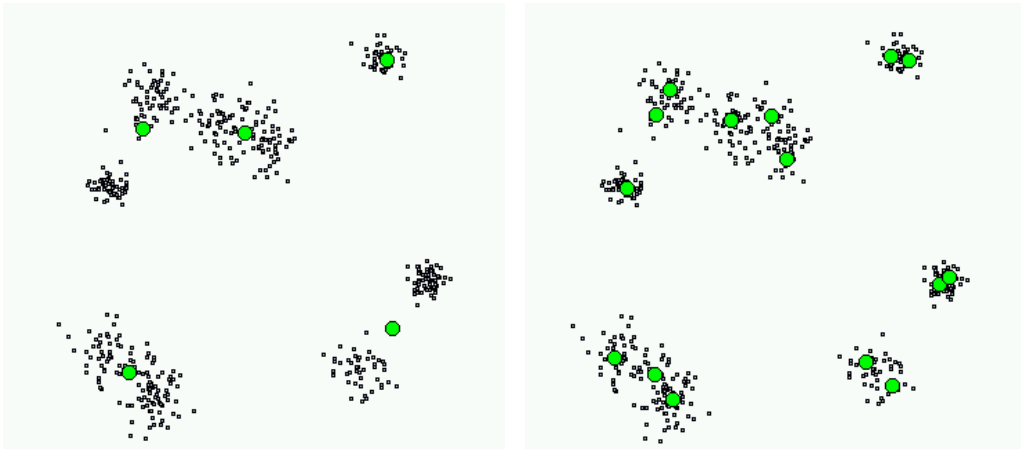
Inhaltsangabe

4.1 Die richtige Netzwerkgröße für Clustering

Sei im folgenden wieder die Menge $S = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ die Menge der Datenpunkte $\mathbf{x}_i \in \mathbb{R}^d$, $1 \leq i \leq n$, welche mit Hilfe einer Menge $A = \{\mathbf{w}_1, \ldots, \mathbf{w}_m\}$ von Referenzvektoren $\mathbf{w}_j \in \mathbb{R}^d, 1 \leq j \leq m$, zu kodieren sind. Kohonen [20] und [19] führt für die Menge A den Begriff des *code book* ein. Die Referenzvektoren beschreiben demnach die Kodierung und werden dadurch *code book vectors* genannt. Die Positionierung der Referenzvektoren aufgrund einer Datenmenge (Trainingsdaten) kann mittels Vektorquantisierung (Abschnitt 2.1) realisiert werden. Wie Kapitel 2 gezeigt hat, gibt es einige Algorithmen, welche zur Vektorquantisierung herangezogen werden können (*k-means*, *fuzzy k-means*, *Learning Vector Quantization*, udg.). Diese Algorithmen nehmen jedoch an, daß die Anzahl m zu findender *Clusterzentren a priori* bekannt ist. Diese Annahme trifft in der Praxis jedoch selten zu. Das Modell des *Growing Neural Gas* (Abschnitt 2.7) versucht zwar mit einem wachsenden *code book* die Anzahl *Clusterzentren* zu bestimmen, jedoch hängt das Ergebnis (Anzahl Referenzvektoren) stark von den gewählten Parametern ab, welche die wachsende Natur des *GNG* Modells bestimmen. Man wird in diesem Fall meistens mehr Referenzvektoren finden, als erforderlich sind. Auch ist das *GNG* Modell empfindlich gegenüber Ausreißern (*outlier*), welche Einfluß auf die Positionen der Referenzvektoren und/oder auf die Anzahl (zusätzlicher) Referenzvektoren haben.

Abbildung 4.1 zeigt Ergebnisse bei Quantisierung einer Datenmenge mit zu wenigen bzw. mit zu vielen Referenzvektoren. Werden zu wenige Referenzvektoren (Abbildung 4.1(a)) verwendet, tritt der Fall auf, daß ein Referenzvektor die Datenpunkte von mehreren Datencluster erklären muß. Abbildung 4.1(b) zeigt den Fall, bei dem zu viele Referenzvektoren eingesetzt wurden. Dabei werden die Datencluster von mehreren Referenzvektoren geteilt und beschrieben. Das gewünschte Ergebnis zeigt Abbildung 2.1(a), bei dem jeder Referenzvektor einen *Clustermittelpunkt* markiert. Weiters hängen die Ergebnisse in beiden Fällen von der Anfangsinitialisierung ab. Wird bei der Initialisierung ein Datencluster zufällig mit vielen Referenzvektoren beschrieben und ein weiterer Datencluster mit keinem, ist nicht gesichert, daß dieser Datencluster am Ende des *Clusterings* doch mit einem oder mehreren Referenzvektoren beschrieben wird. Der *Clustering* Algorithmus kann in einem lokalen Optimum hängen bleiben, wodurch dieser Fall auftreten kann.

Einige Methoden wurden entwickelt, um die richtige Anzahl an Referenzvektoren zu finden (z.B. *ISODATA*, der in einer Art *split and merge* die Anzahl der



(a) (b)

Abbildung 4.1: Vektorquantisierung einer aus 9 *Cluster* bestehenden Datenmenge (siehe auch Abbildung 2.1). Kleine Rechtecke repräsentieren Datenpunkte und Kreise stellen die Referenzvektoren dar. Abbildung (a) zeigt das Ergebnis bei Verwendung von $m = 5$ (zu wenigen) Referenzvektoren und Abbildung (b) zeigt das Ergebnis bei Verwendung von $m = 15$ (zu vielen) Referenzvektoren.

Voronoi Regionen verändert; siehe Duda [8]). Wie im vorhergehenden Abschnitt dargestellt, kann das *MDL* Prinzip dazu verwendet werden, die Generalisierfähigkeit verschiedener Modelle zu messen und zu vergleichen. Weiters kann man das *MDL* Prinzip mit einem *pruning* Algorithmus koppeln und so die Qualität des reduzierten Netzwerkes beurteilen. Das *MDL* Prinzip besitzt den Vorteil, daß es nicht versucht das den Datenpunkten zu Grunde liegende Verteilungsmodell zu schätzen, sondern mittels vorgegebenem (Kodierungs)Modell die Komplexität der Datenbeschreibung anzugeben. Nachdem man die Referenzvektoren von *Clustering* Algorithmen als grobe Beschreibungen der Datenpunkte ansehen kann, eignet sich das *MDL* Prinzip gut zur Beurteilung von *Clustering* Ergebnissen. Im folgenden wird die Anwendung des *MDL* Prinzips zur Beurteilung von *Clustering* Ergebnissen präsentiert. Weiters wird gezeigt, wie man dies mit einem iterativen *pruning* Algorithmus koppelt, welcher es ermöglicht, die richtige Netzwerkgröße für *Clustering* Netzwerke zu ermitteln.

4.2 MDL für Crisp Clustering

In diesem Abschnitt leiten wir ein *MDL* Kriterium für *crisp Clustering* her. *Crisp Clustering* bedeutet, daß ein bestimmter Datenpunkt genau einem Referenzvektor zugeordnet wird, der ihn beschreibt. Es gilt demnach ein (Kodierungs)Modell zu finden, das mit Hilfe der Referenzvektoren A die Menge S der Trainingsdaten beschreibt. Ein Datenpunkte $\mathbf{x} \in S$ kann durch folgenden Zusammenhang genau beschrieben werden.

$$\mathbf{x} = \mathbf{w}_s + \epsilon(\mathbf{x}, \mathbf{w}_s), \tag{4.1}$$

wobei $\mathbf{w}_s$ der zu $\mathbf{x}$ nächste Referenzvektor (Gleichung (2.28)) und $\epsilon(\mathbf{x}, \mathbf{w}_s)$ der Abstand (Fehler) zwischen $\mathbf{w}_s$ und $\mathbf{x}$ ist. Nachdem man aber die gesamte Menge S der Datenpunkte über die Referenzvektoren beschreiben möchte, bietet sich an, die Werte (Positionen) der Referenzvektoren in ein *code book* (ähnlich einer *look-up* Tabelle) zu speichern und für $\mathbf{w}_s$ in Gleichung (4.1) lediglich den Index des Referenzvektors im *code book* (Index($\mathbf{w}_s$)) zu verwenden.

$$\mathbf{x} = (\text{Index}(\mathbf{w}_s), \epsilon(\mathbf{x}, \mathbf{w}_s)). \tag{4.2}$$

4.2.1 Beschreibungslänge

Abbildung 4.2 stellt die Verwendung eines *code book* zur Kodierung der Datenpunkte übersichtlich dar. Seien $L(A)$ die Kosten, die für die Kodierung der Referenzvektoren (*code book*) benötigt werden. Weiters seien $L(\text{Index}(A))$ die Kosten der Indizierung der Referenzvektoren A für die Beschreibung der Datenpunkte und $L(\epsilon(S, A))$ die Kosten für die Kodierung aller Residuen zwischen den Datenpunkten und deren nächster Referenzvektoren. Dann ergeben sich für die gesamten Kodierungskosten (Beschreibungslänge) $L(S(A))$ der Menge S folgende Zusammenhänge.

$$L(S(A)) = L(A) + L(\text{Index}(A)) + L(\epsilon(S, A)). \tag{4.3}$$

Für die Bestimmung der richtigen Anzahl an Referenzvektoren könnte man alle möglichen Kombinationen an *code book* Größen erzeugen und für jede Klasse

Index	Position
1	$\mathbf{w}_1$
2	$\mathbf{w}_2$
$\vdots$	$\vdots$
m	$\mathbf{w}_m$

code book

Datenpunkt	Index $\mathbf{w}_s$	Residuum zu $\mathbf{w}_s$
$\mathbf{x}_1$	2	$\epsilon(\mathbf{x}_1, \mathbf{w}_2)$
$\mathbf{x}_2$	9	$\epsilon(\mathbf{x}_2, \mathbf{w}_9)$
$\vdots$	$\vdots$	$\vdots$
$\mathbf{x}_n$	5	$\epsilon(\mathbf{x}_n, \mathbf{w}_5)$

Index und Residuen der Datenpunkte

Abbildung 4.2: Kodierung der Datenpunkte mit Hilfe eines *code book* (links). Jeder Datenpunkt wird mit dem Index seines nächsten Referenzvektors indiziert und zusätzlich werden die Residuen zwischen Datenpunkte und nächste Referenzvektoren gespeichert (rechts).

von *code books* einer bestimmten Größe jene Positionen der Referenzvektoren bestimmen, die das Fehlermaß $E(S, A)$ (Gleichung (2.1)) minimieren. Anschließend bewertet man sie mittels der Beschreibungslänge aus Gleichung (4.3) und wählt jenes *code book*, welches $L(S(A))$ minimal werden läßt. Bei n Datenpunkten wären das n mögliche Klassen an *code books* und deshalb ist diese Vorgangsweise nicht durchführbar, da in der Praxis große Datenmengen auftreten und in jeder Klasse eine Positionierung der Referenzvektoren nötig ist. Der folgende *MDL* Algorithmus wurde von Bischof und Leonardis [3] beschrieben (siehe auch Bischof, Leonardis und Selb [4]; Selb, Bischof und Leonardis [37]).

Nehmen wir an, daß jeder Datenpunkt $\mathbf{x} \in \mathbb{R}^d$ mit K bits pro Dimension kodiert wird, wodurch auch die Referenzvektoren jeweils mit K bits pro Dimension kodiert werden. Für alle anderen zu kodierenden Einheiten soll eine Genauigkeit von η pro Dimension erreicht werden. Weiters seien die Datenpunkte identisch und unabhängig verteilt, wodurch gilt

$$p(\mathbf{x}_i, \mathbf{x}_j) = p(\mathbf{x}_i)p(\mathbf{x}_j), \qquad \forall i, j \in \{1, \ldots, n\}. \tag{4.4}$$

Unter den obigen Annahmen ergibt sich für die gesamte Beschreibungslänge

$$\begin{aligned} L(S(A)) &= L(A) + L(\text{Index}(A)) + L(\epsilon(S, A)) \\ &= m\ d\ K + L(\text{Index}(A)) + \sum_{i=1}^{m} \sum_{\mathbf{x} \in S_i} L(\epsilon(\mathbf{x}, \mathbf{w}_i)), \end{aligned} \tag{4.5}$$

wobei m die Anzahl Referenzvektoren und d deren Dimensionalität notiert. S_i ist wieder die *Voronoi Region* (Gleichung (2.3)) des Referenzvektors $\mathbf{w}_i$.

4.2.2 Outlier

Im vorhergehenden Abschnitt haben wir angenommen, daß die Datenmenge S gleich der Menge der *inlier* I ($S = I$) und die Menge der *outlier* O ($O = \emptyset$) leer ist. *Outlier** sind jene Datenpunkte, welche nicht der Datenverteilung $p(\mathbf{x})$ (siehe Abschnitt 2.1) genügen. Die beiden Mengen O und I sind natürlicherweise disjunkt.

$$I = S - O. \tag{4.6}$$

Wir wollen aber nur die Datenpunkte aus der Menge I der *inlier* durch das Modell kodieren, da die *outlier* zu einer Erhöhung der Modellkomplexität führen – da die Kodierung der *outlier* zusätzliche Referenzvektoren erfordert – und/oder die Kodierungsgenauigkeit der *inlier* darunter leidet. Deswegen ändert sich die Definition der gesamten Beschreibungslänge zu

$$\begin{aligned} L(S(A)) &= L(I) + L(O), & (4.7)\\ L(I) &= L(A) + L(\text{Index}(A)) + L(\epsilon(I, A)), & (4.8) \end{aligned}$$

wobei hier $L(I)$ die Beschreibungslänge der *inlier* und $L(O)$ die Beschreibungslänge der *outlier* notiert. Die *outlier* werden nun nicht mehr über das *code book* (Referenzvektoren) beschrieben, wodurch sich andere Werte für $L(A)$ und $L(\text{Index}(A))$ ergeben. Dies führt weiters auch zu einer Änderung des Fehlerterms, der nur noch von den *inlier* abhängt ($L(\epsilon(I, A))$). Da ein *outlier* nicht durch einen Referenzvektor kodiert wird, muß er mit $d\ K$ bits direkt kodiert werden.

$$\begin{aligned} L(S(A)) &= L(A) + L(\text{Index}(A)) + L(\epsilon(I, A)) + L(O)\\ &= m\ d\ K + L(\text{Index}(A)) + \sum_{i=1}^{m} \sum_{\mathbf{x} \in S_i} L(\epsilon(\mathbf{x}, \mathbf{w}_i)) + |O|\ d\ K, \quad (4.9) \end{aligned}$$

Bezüglich des *MDL* Modells sind jene Datenpunkte *outlier*, die durch direkte Kodierung ($d\ K$ bits) günstiger zu kodieren sind, als mit Hilfe der Referenzvektoren. Deswegen betrachtet man die Änderung der gesamten Beschreibungslänge

*Im *MDL* Modell werden die *outlier* entsprechend einem *MDL* Kriterium als solche klassifiziert. Die Menge O ist im *MDL* Modell nur eine Schätzung der *outlier*, welche nicht der Verteilungsdichte $p(\mathbf{x})$ genügen.

(Gleichung (4.9)), wenn man einen *inlier* $\mathbf{x}$ als *outlier* klassifiziert. Dabei sind 2 Fälle zu unterscheiden.

1. *Inlier* $\mathbf{x}$ ist nicht der einzige Datenpunkt in seiner *Voronoi Region* S_i:

   - Die Anzahl an Referenzvektoren (*Voronoi Regionen*) m bleibt gleich.
   - Der Index für den Referenzvektor von $\mathbf{x}$ wird eingespart.
   - Die Kodierung des Residuums $\epsilon(\mathbf{x}, \mathbf{w}_i)$ wird eingespart.
   - Der zusätzliche *outlier* $\mathbf{x}$ muß mit $d\ K$ bits kodiert werden.

   Ein *inlier* $\mathbf{x}$ wird in diesem Fall als *outlier* klassifiziert, wenn gilt

   $$d\ K < L(\text{Index}_{(-\mathbf{x})}(A)) + L(\epsilon(\mathbf{x}, \mathbf{w}_i)). \tag{4.10}$$

2. *Inlier* $\mathbf{x}$ ist der einzige Datenpunkt in seiner *Voronoi Region* S_i:

   - Der Referenzvektor $\mathbf{w}_i$ ist überflüssig und wird entfernt. Man erspart sich $d\ K$ bits Kodierungskosten im *code book*.
   - Der Index für den Referenzvektor von $\mathbf{x}$ wird eingespart.
   - Dadurch, daß der Index für $\mathbf{x}$ wegfällt, kann es, abhängig vom Kodierungsverfahren der Indizes, Einsparungen beim Kodierungsaufwand der Indizes im gesamten kommen.
   - Keine Einsparung im Fehlerterm, da der Referenzvektor den Datenpunkt mit einem Residuum von 0 beschrieben hat.
   - Der zusätzliche *outlier* $\mathbf{x}$ muß mit $d\ K$ bits kodiert werden.

   Ein *inlier* $\mathbf{x}$ wird in diesem Fall als *outlier* klassifiziert, wenn gilt

   $$d\ K < L(\text{Index}_{(-\mathbf{x})}(A)) + d\ K. \tag{4.11}$$

Faßt man nun die Einsparungen bzw. zusätzlichen Aufwände der beiden Möglichkeiten zusammen und stellt sie einander gegenüber, ergibt sich die Bedingung für einen Datenpunkt, um der Menge der *outlier* O zugeordnet zu werden.

$$d\ K < L(\text{Index}_{(-\mathbf{x})}(A)) + L(\epsilon(\mathbf{x}, \mathbf{w}_i)) + \mathcal{I}_{|S_i|=1} d\ K, \tag{4.12}$$

$$\mathcal{I}_{|S_i|=1} = \begin{cases} 1 : |S_i| = 1 \\ 0 : \text{sonst} \end{cases}, \tag{4.13}$$

wobei $\mathcal{I}_{|S_i|=1}$ die Indikatorfunktion ist, daß $\mathbf{x}$ der einzige Datenpunkt in *Voronoi Region* S_i ist und $L(\text{Index}_{(-\mathbf{x})}(A))$ die Kodierungskosten für die Indizes notiert, wenn der Datenpunkt $\mathbf{x}$ zum *outlier* wird.

4.2.3 MDL Kriterium

Bisher wurde beschrieben, wie Datenpunkte mit Hilfe eines *code book* kodiert/beschrieben werden können. Im folgenden soll ein Kriterium präsentiert werden, welches für die Kodierung überflüssige Referenzvektoren detektiert, die in weiterer Folge entfernt werden. Das *MDL* Kriterium ist demnach ein *pruning* Kriterium für ein vorliegendes Kodierungsmodell. Man geht dabei davon aus, daß bereits die Menge A der Referenzvektoren und eine entsprechende Beschreibungslänge $L(S(A))$ vorliegt. Auch die Menge O der *outlier* wurde bereits bestimmt. Man sucht nun jenen Referenzvektor $\mathbf{w}_j$, der eine Reduktion der Beschreibungslänge $L(S(A))$ zur Folge hat, wenn der Referenzvektor $\mathbf{w}_j$ aus der Menge A entfernt wird. Die Änderung der Beschreibungslänge bei Entfernung des Referenzvektors $\mathbf{w}_j$ ist demnach

$$\Delta L_{(-\mathbf{w}_j)} = L(S(A\backslash\{\mathbf{w}_j\})) - L(S(A)). \tag{4.14}$$

Kandidaten für eine Reduktion sind alle Referenzvektoren $\mathbf{w}_j$, die eine negative Änderung ($\Delta L_{(-\mathbf{w}_j)} < 0$) der Beschreibungslänge zur Folge haben. Die Änderung der Beschreibungslänge kann aber mit

$$\begin{aligned} \Delta\hat{L}_{(-\mathbf{w}_j)} &= -d\ K + L(\text{Index}(A\backslash\{\mathbf{w}_j\})) - L(\text{Index}(A)) \\ &+ \sum_{\mathbf{x}\in S_j} \Big(L(\epsilon(\mathbf{x}, A\backslash\{\mathbf{w}_j\})) - L(\epsilon(\mathbf{x}, \mathbf{w}_j)) \Big) \end{aligned} \tag{4.15}$$

abgeschätzt werden. $L(\text{Index}(A\backslash\{\mathbf{w}_j\}))$ gibt die Kodierungskosten der Indizes an, wenn der Referenzvektor $\mathbf{w}_j$ entfernt wird. In diesem Fall müssen alle Datenpunkte $\mathbf{x} \in S_j$ anderen Referenzvektoren zugeordnet werden. Das selbe gilt für die Kosten der Residuenkodierung $L(\epsilon(\mathbf{x}, A\backslash\{\mathbf{w}_j\}))$. Dabei werden alle Datenpunkte ihrem zweitnächsten Referenzvektor (Gleichung (2.29)) zugeordnet. Die Änderung der Beschreibungslänge in Gleichung (4.15) ist nur eine Schätzung von $\Delta L_{(-\mathbf{w}_j)}$, da sie den Fall nicht berücksichtigt, daß bei Entfernung des Referenzvektors $\mathbf{w}_j$ einige *inlier* zu *outlier* werden können. Der umgekehrte Fall ist nicht möglich, da die Positionen der Referenzvektoren dabei unverändert bleiben.

Ein *inlier* wird laut Definition der *outlier* (Gleichung (4.12)) nur dann zu einem *outlier*, wenn er selbst günstiger kodiert werden kann, als mit Hilfe von Referenzvektoren. Wird ein *inlier* zum *outlier* wird das die Beschreibungslänge $L(S(A))$ zusätzlich reduzieren. Somit ist gezeigt, daß gilt

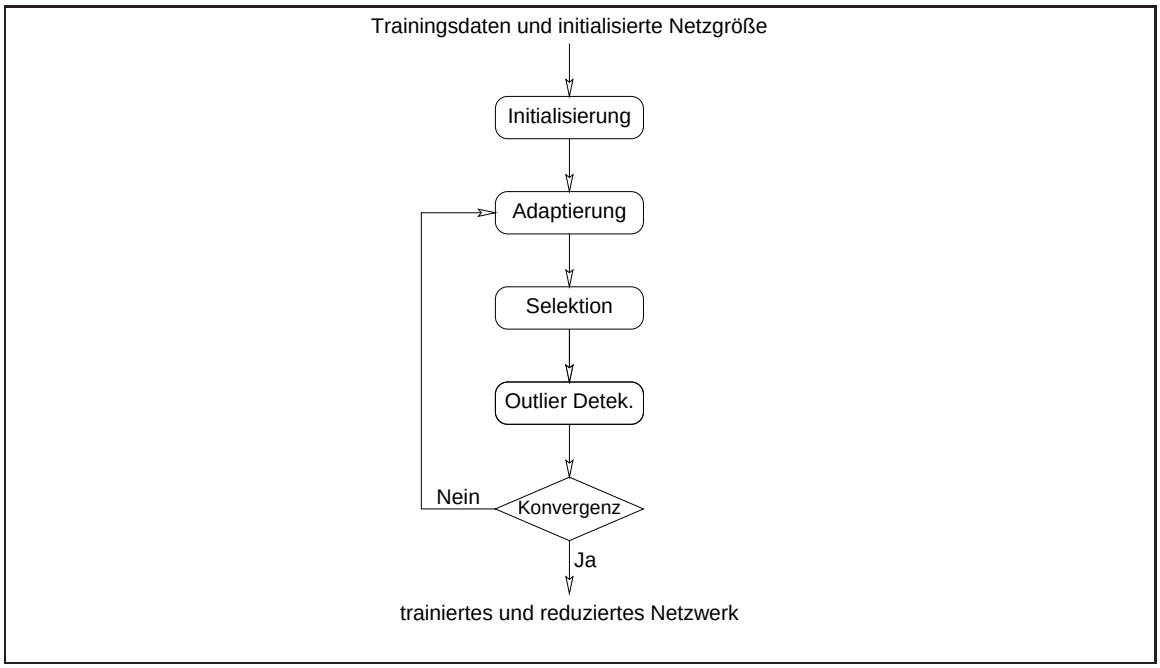
$$\Delta\hat{L}_{(-\mathbf{w}_j)} \geq \Delta L_{(-\mathbf{w}_j)}, \qquad \forall \mathbf{w}_j \in A. \tag{4.16}$$

4.2.4 MDL Algorithmus

Die vorhergehende Bedingung für die Entfernung eines Referenzvektors gilt jedoch nur für einen einzigen Referenzvektor. Bischof und Leonardis [3] haben gezeigt, daß es möglich ist, in einem Iterationsschritt alle nicht benachbarten Referenzvektoren, welche die Bedingung $\Delta\hat{L}_{(-\mathbf{w}_j)} < 0$ erfüllen, parallel zu entfernen. Es sind dann zwei Referenzvektoren benachbart, wenn zwischen ihnen eine Verbindung in der *induzierten Delauny Triangulation* existiert (Abschnitt 2.6).

Beweis für parallele Reduktion. Angenommen die beiden nicht benachbarten Referenzvektoren $\mathbf{w}_i$ und $\mathbf{w}_j$ haben jeweils eine Reduktion der gesamten Beschreibungslänge zur Folge, wenn sie entfernt würden, d.h. $\Delta\hat{L}_{(-\mathbf{w}_i)} < 0$ und $\Delta\hat{L}_{(-\mathbf{w}_j)} < 0$. Sei weiters $\Delta\hat{L}_{(-\mathbf{w}_i,\mathbf{w}_j)}$ jene Änderung der Beschreibungslänge, wenn der Referenzvektor $\mathbf{w}_i$ entfernt wird und anschließend $\mathbf{w}_j$. Für den Fall, daß beide Referenzvektoren $\mathbf{w}_i$ und $\mathbf{w}_j$ parallel entfernt werden dürfen, muß $\Delta\hat{L}_{(-\mathbf{w}_i,\mathbf{w}_j)} < 0$ gelten. Dabei ändert sich der Fehlerterm und der konstante Term $-d\,K$ in Gleichung (4.15) nicht. Deswegen reicht es für diesen Beweis zu zeigen, daß $L(\text{Index}(A\backslash\{\mathbf{w}_j\})) > L(\text{Index}(A\backslash\{\mathbf{w}_i,\mathbf{w}_j\}))$. Dies ist erfüllt, da sich kein vernünftiges Kodierungsmodell finden läßt, welches, mit einem Referenzvektor weniger, keine zusätzliche Reduktion der Kodierungskosten der Indizes zur Folge hat. □

Bei der parallelen Reduktion kann aber nicht garantiert werden, daß alle überflüssigen Referenzvektoren entfernt wurden. Deswegen wird ein iterativer Algorithmus benötigt, der fortlaufend überflüssige Referenzvektoren entfernt, bis kein Referenzvektor mehr der Bedingung $\Delta\hat{L}_{(-\mathbf{w}_j)} < 0$ genügt. Algorithmus 4.1 zeigt den schematischen Aufbau des *MDL* Algorithmus. Die Aufgaben der einzelnen Algorithmusschritte sind:



Algorithmus 4.1: Schematische Darstellung des iterativen Ablaufes des *MDL* Algorithmus für *crisp Clustering*.

Initialisierung: Die Menge der Referenzvektoren A wird mit der Menge der Trainingsdaten initialisiert ($I = S$). Das heißt, daß jeder Referenzvektor genau einen Datenpunkt erklärt und somit der Fehlerterm des *MDL* Algorithmus 0 ist. Jedoch sind die Kosten für das *code book* und der Indizierung maximal. Existieren weniger Datencluster als Datenpunkte, wird es im Selektionsschritt zu einer Reduktion der Anzahl an Referenzvektoren kommen. Abschnitt 4.4 wird zeigen, daß mit geschickter Initialisierung der Berechnungsaufwand reduziert werden kann.

Adaptierung: In diesem Schritt müssen die Positionen der Referenzvektoren adaptiert werden, wobei dafür nur die Menge I verwendet wird. Im Falle von $I = S$ bringt dieser Schritt keine Änderung der Referenzvektorpositionen. Dazu kann jeder *Clustering* Algorithmus verwendet werden. Beispiele dazu werden in Kapitel 2 gegeben. Abschnitt 4.4 wird zeigen, daß es nicht notwendig ist, die *Clustering* Algorithmen bis zur vollständigen Konvergenz zu betreiben, wodurch eine Verbesserung der Laufzeit des *MDL* Algorithmus erreicht werden kann.

Selektion: Die Selektion der überflüssigen Referenzvektoren (im Sinne der minimalen Beschreibungslänge) wird aufgrund dem oben beschriebenen *MDL* Kriterium getroffen. Wird in jedem Iterationsschritt nur ein Referenzvektor

entfernt, so ist jener zu wählen, der die maximale, negative Änderung der Beschreibungslänge zur Folge hat. Der effizientere Fall der parallelen Entfernung von Referenzvektoren wurde entsprechend einer Greedy Strategie vorgenommen. Dabei wird zuerst der Referenzvektor mit der maximalen, negativen Änderung der Beschreibungslänge entfernt, welches zur Folge hat, daß alle seine direkten Nachbarn nicht entfernt werden dürfen. Anschließend wird mit dem nächstbesten Referenzvektor fortgefahren und so weiter. Dadurch wird zwar nicht garantiert, daß in diesem Iterationsschritt das aktuelle Minimum der Beschreibungslänge gefunden wird, aber aufgrund des iterativen Ablaufes des *MDL* Algorithmus wird dies in einem der nächsten Iterationsschritt korrigiert.

Outlier Detektion: Entsprechend Gleichung (4.12) werden alle Datenpunkte der Menge S in die beiden Mengen der *inlier* I bzw. *outlier* O unterteilt. Dabei kann es vorkommen, daß Datenpunkte sowohl von der Menge I in die Menge O wechseln, als auch umgekehrt, wodurch die Bestimmung der *outlier* immer mit der gesamten Menge S erfolgen muß.

Konvergenz: Die Konvergenz ist erreicht, wenn im aktuellen Iterationsschritt keine zusätzlichen *outlier* detektiert wurden und keine Referenzvektoren entfernt wurden und die Änderung der Referenzvektorpositionen (*Adaptierungsschritt*) unter eine vorgegebene Schranke fällt.

In jedem der 3 Schritte *Adaptierung*, *Selektion* und *Outlier Detektion* erfolgt nie eine Verschlechterung der Beschreibungslänge und solange in jedem Iterationsschritt zumindest einer der 3 Schritte eine Reduktion der Beschreibungslänge ergibt, wird bei Erreichen der Konvergenz zumindest ein lokales Minimum der Beschreibungslänge $L(S(A))$ gefunden. Der Vorteil des *MDL* Algorithmus ist, daß er nicht nur die Netzwerkkomplexität reduziert, sondern während der Reduktion auch die Positionen der Referenzvektoren lernt. Abschnitt 4.4 wird sich noch mit einigen Verbesserungen des *MDL* Algorithmus beschäftigen. Bis jetzt wurde lediglich der Rahmen für den *MDL* Algorithmus präsentiert. Für eine Instanzierung des Algorithmus ist es hingegen notwendig, sich mit der Kodierung der Indizes bzw. Residuen zu beschäftigen. Die beiden nächsten Abschnitte werden jeweils 2 typische Möglichkeiten der Kodierung vorstellen.

4.2.5 Kodierung der Indizes

1. **Fixe Kodierungslänge:** Eine einfache Möglichkeit eine Kodierung der Indizes zu definieren, ist die Kodierung mit fixer Kodierungslänge (*fixed length code*). Dabei wird für jeden Referenzvektor ein Index der Länge $\log_2(m)$ (in bits) vorgesehen. Für die Kodierung der Indizes des gesamten *code books* werden

$$L(\text{Index}(A)) = |I| \log_2(m) \tag{4.17}$$

bits benötigt, wobei $|I|$ die Anzahl der *inlier* angibt. Wird ein Referenzvektor $\mathbf{w}_j$ entfernt, werden nur noch

$$L(\text{Index}(A \backslash \{\mathbf{w}_j\})) = |I| \log_2(m-1) \tag{4.18}$$

bits benötigt. Die Änderung der Kodierungskosten ergibt sich aus der Differenz der beiden oben definierten Indizierungskosten

$$\begin{aligned} \Delta L(\text{Index}(A \backslash \{\mathbf{w}_j\})) &= L(\text{Index}(A \backslash \{\mathbf{w}_j\})) - L(\text{Index}(A)) \\ &= |I|(\log_2(m-1) - \log_2(m)). \end{aligned} \tag{4.19}$$

2. **Variable Kodierungslänge:** Bei Kodierung der Indizes mit variabler Kodierungslänge (*variable length code*) verwendet man eine Kodierung, welche die Kodierungslänge entsprechend der Auftrittswahrscheinlichkeit der Referenzvektoren wählt. Sei $p_i = \frac{n_i}{|I|}$, $n_i = |S_i|$, die Auftrittwahrscheinlichkeit von Referenzvektor $\mathbf{w}_i$, dann ist die Kodierungslänge des Index eines Datenpunktes $\mathbf{x} \in S_i$ entsprechend $-\log_2(p_i)$. Die Kodierungskosten der Indizes im gesamten *code book* errechnen sich aus

$$L(\text{Index}(A)) = -\sum_{i=1}^{m} n_i \log_2(p_i). \tag{4.20}$$

Wird nun Referenzvektor $\mathbf{w}_j$ entfernt, müssen alle Datenpunkte $\mathbf{x} \in S_j$ durch ihre jeweiligen zweitnächsten Referenzvektoren $\mathbf{w}_k$ (Gleichung (2.29)) indiziert werden.

$$L(\text{Index}(A\backslash\{\mathbf{w}_j\})) = -\sum_{k=1,k\neq j}^{m}(n_k + n_{jk})\log_2(\frac{n_k + n_{jk}}{|I|}). \tag{4.21}$$

$$\Delta L(\text{Index}(A\backslash\{\mathbf{w}_j\})) = \sum_{k=1,k\neq j}^{m}(n_k + n_{jk})\log_2(\frac{n_k + n_{jk}}{|I|}) - \sum_{k=1}^{m} n_k \log_2(p_k) \tag{4.22}$$

$$= -n_j \log_2(p_j) + \sum_{k=1,k\neq j}^{m}\Big(n_{jk}\log_2(\frac{n_k + n_{jk}}{|I|}) + n_k \log_2(\frac{n_k + n_{jk}}{n_k})\Big). \tag{4.23}$$

Die Anzahl Datenpunkte, die bei Entfernung von $\mathbf{w}_j$ von der *Voronoi Region* S_j in die *Voronoi Region* S_k wechseln, wird mit n_{jk} angegeben. Wechseln sehr wenige Datenpunkte in eine benachbarte *Voronoi Region* ($n_k \gg n_{jk}$), kann Gleichung (4.23) durch folgenden Ausdruck approximiert* werden.

$$\Delta L(\text{Index}(A\backslash\{\mathbf{w}_j\})) = -n_j \log_2(p_j) + \sum_{k=1,k\neq j}^{m} n_{jk}\log_2(\frac{n_k + n_{jk}}{|I|}). \tag{4.24}$$

4.2.6 Kodierung der Residuen

1. **Unbekannte Fehlerverteilung:** Ist die Verteilung der Residuen $p(\epsilon(\mathbf{x}, \mathbf{w}_i))$ nicht bekannt, bietet sich eine Kodierung der Residuen proportional zu ihren Größe an, wobei eine Diskretisierungsgenauigkeit von η gefordert ist. Unter der Annahme, daß die Dimensionen des Residuums $(\mathbf{x} - \mathbf{w}_i)$ unabhängig voneinander sind, ergibt sich bei getrennter Kodierung jeder Dimension eine Kodierungslänge des Fehlers für Datenpunkt $\mathbf{x}$

$$L(\epsilon(\mathbf{x}, \mathbf{w}_i)) = \sum_{j=1}^{d}\max\Big(\log_2(\frac{|\mathbf{x} - \mathbf{w}_i|}{\eta}), 1\Big), \tag{4.25}$$

wobei $\mathbf{w}_i$ der zu $\mathbf{x}$ nächste Referenzvektor ist. Unter der Annahme, daß der Fehler pro Dimension im Mittel über alle Dimensionen gleich ist, kann für die Änderung der Kodierungskosten bei Entfernung von Referenzvektor $\mathbf{w}_j$ folgender Zusammenhang angegeben werden.

$$\Delta L(\epsilon(S, A\backslash\{\mathbf{w}_j\})) = d\sum_{\mathbf{x}\in S_j}\log_2\Big(\frac{||\epsilon(x, A\backslash\mathbf{w}_j)||}{||\epsilon(x, A)||}\Big), \tag{4.26}$$

*Für den Fall, daß $n_k \gg n_{jk}$ gilt, ergibt sich die Approximation von $\log_2(n_k + n_{jk}) \approx \log_2(n_k)$.

2. **Bekannte Fehlerverteilung (Beispiel: Gaußverteilung):** Kann man eine Annahme bezüglich der Fehlerverteilung $p(\epsilon(\mathbf{x}, \mathbf{w}_i))$ treffen, so läßt sich der Fehler von Datenpunkt $\mathbf{x}$ mit Hilfe dieser Verteilung mit

$$L(\epsilon(\mathbf{x}, \mathbf{w}_i)) = -\log_2(p(\epsilon(\mathbf{x}, \mathbf{w}_i))) \tag{4.27}$$

bits kodieren ($\mathbf{w}_i$ nächster Referenzvektor zu $\mathbf{x}$). Im folgenden wird eine konkrete Formulierung der Fehlerkosten für eine Gaußverteilung mit Mittelwert 0 und einer Varianz pro Dimension von σ^2 angegeben. Weiters sei die Fehlerverteilung unabhängig in jeder Dimension und $\mathbf{x}_i$ der Wert von $\mathbf{x}$ in Dimension i bzw. $\mathbf{w}_{ji}$ der Wert von Referenzvektor $\mathbf{w}_j$ in Dimension i. So ergibt sich für die Fehlerverteilung und die Kodierungskosten der Residuen* von Datenpunkt $\mathbf{x}$ ($\mathbf{w}_j$ nächster Referenzvektor zu $\mathbf{x}$).

$$\begin{aligned} p(\epsilon(\mathbf{x}, \mathbf{w}_j) &= \prod_{i=1}^{d} \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{\epsilon(\mathbf{x}_i, \mathbf{w}_{ji})^2}{2\sigma^2}}. && (4.28) \\ L(\epsilon(\mathbf{x}, \mathbf{w}_j) &= -\log_2(p(\epsilon(\mathbf{x}, \mathbf{w}_j))) \\ &= \sum_{i=1}^{d} \frac{\epsilon(\mathbf{x}_i, \mathbf{w}_{ji})^2}{2\ln(2)\sigma^2} + d\log_2(\sqrt{2\pi}\sigma) - d\log_2(\eta). && (4.29) \end{aligned}$$

Wird der Referenzvektor $\mathbf{w}_j$ aus der Menge A entfernt, und wird angenommen, daß die Datenpunkte in jedem Datencluster die selbe Varianz σ^2 besitzen, so gilt für die Änderung der Kodierungskosten der Residuen

$$\Delta L(\epsilon(\mathbf{x}, A \backslash \{\mathbf{w}_j\}) = \sum_{\mathbf{x} \in S_j} \sum_{i=1}^{d} \frac{(\mathbf{x}_i - \mathbf{w}_{ki})^2 - (\mathbf{x}_i - \mathbf{w}_{ji})^2}{2\ln(2)\sigma^2}, \tag{4.30}$$

wobei $\mathbf{w}_k$ der zum Datenpunkt $\mathbf{x}$ zweitnächste Referenzvektor ist. Der für den Fehlerterm wichtige Parameter der Varianz σ^2 müßte streng genommen ebenfalls durch das *MDL* Modell mitkodiert werden und hätte zusätzliche Kosten der einzelnen Referenzvektoren zur Folge. Wir nehmen aber an, daß die Varianz σ^2 sowohl dem Sender als auch dem Empfänger bekannt ist und deshalb seine Kodierung vernachlässigt wird.

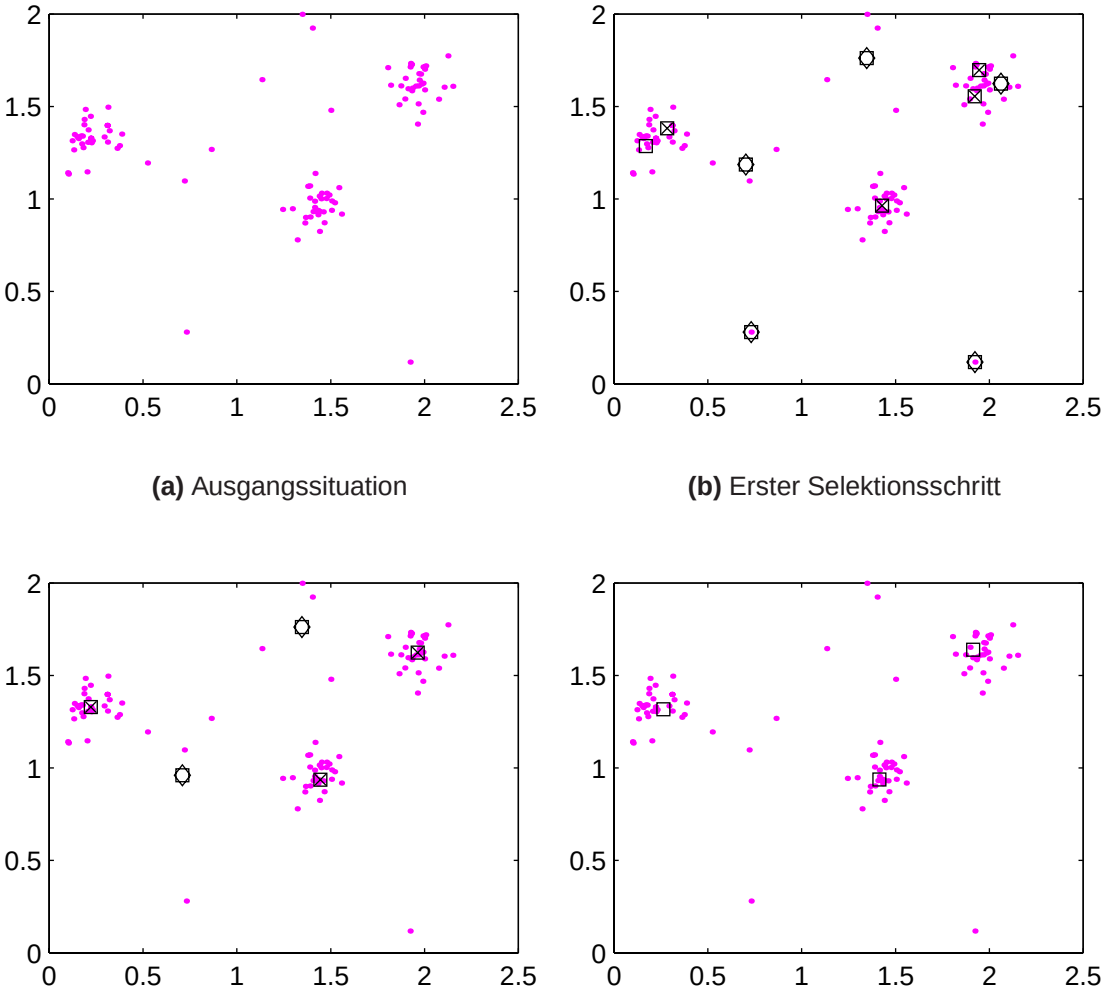
*Die Residuen in Gleichung (4.29) wurden wieder mit der Diskretisierungsgenauigkeit η in jeder Dimension getrennt bewertet.

4.2.7 Beispiel eines MDL Durchlaufes

Um einen beispielhaften Durchlauf des *MDL* Algorithmus zu präsentieren, wurden Testdaten erzeugt, die aus 3 gaußverteilten Datenclustern (je 30 Punkte, $\sigma = 0.07$) und 10 gleichverteilten Punkten bestehen (Abbildung 4.3(a)). Als *Clustering* Algorithmus wird der *k-means* Algorithmus verwendet, der mit 10 Referenzvektoren initialisiert wird (Kodierung der Residuen unter der Annahme der Gaußverteilung und optimale Kodierungslänge für die Kodierung der Indizes). In den Abbildungen 4.3 und 4.4 symbolisieren einfache Punkte ($\cdot$) die Datenvektoren, Achtecke ($\odot$) die zu entfernenden Referenzvektoren, angekreuzte Rechtecke ($\boxtimes$) die gesperrten Referenzvektoren, einfache Rechtecke ($\square$) die übriggebliebenen Referenzvektoren und angekreuzte Datenpunkte ($\times$) die *outlier*.

Wird der *MDL* Algorithmus ohne *outlier* Detektion gestartet (Abbildung 4.3), so bleiben nach dem ersten Selektionsschritt 5 Referenzvektoren übrig (Abbildung 4.3(b)). Die 4 gesperrten Referenzvektoren ergeben sich aufgrund ihrer Nachbarschaft zu den zu entfernenden Referenzvektoren. Auch ist ein Referenzvektor zu erkennen, welcher keine Reduktion der Beschreibungslänge zur Folge hat (symbolisiert mit $\square$). Abbildung 4.3(c) zeigt 2 zu entfernende und 3 gesperrte Referenzvektoren und Abbildung 4.3(d) zeigt das Endergebnis, bei dem alle 3 *Clusterzentren* gefunden wurden. In diesem Fall müssen auch alle *outlier* mittels den gefundenen Referenzvektoren kodiert werden.

Bei Verwendung der selben Datenpunkte wird in Abbildung 4.4 der *MDL* Algorithmus mit *outlier* Detektion angewandt. Ähnlich wie in Abbildung 4.3(b) werden in Abbildung 4.4(b) 5 Referenzvektoren eliminiert, jedoch 2 Datenpunkte als *outlier* detektiert. Diese beiden Datenpunkte werden im nächsten *Adaptierungsschritt* nicht berücksichtigt. Vergleicht man die Abbildungen 4.3(c) und 4.4(c), so erkennt man, daß im letzteren Fall die Referenzvektoren nicht mehr in Richtung der detektierten *outlier* hingezogen werden. Nach dem zweiten Selektionsschritt ist, aufgrund der Neupositionierung der Referenzvektoren, nur noch ein *outlier* detektiert worden. Das Endergebnis zeigt wieder die 3 gefundenen Referenzvektoren und die beiden detektierten *outlier*, welche keinen Einfluß auf die Positionen der Referenzvektoren haben (Abbildung 4.4(d)). Der Grund, weshalb die anderen *outlier* nicht detektiert wurden, ist, daß diese keine *outlier* im Sinne des *MDL* Kriteriums sind.



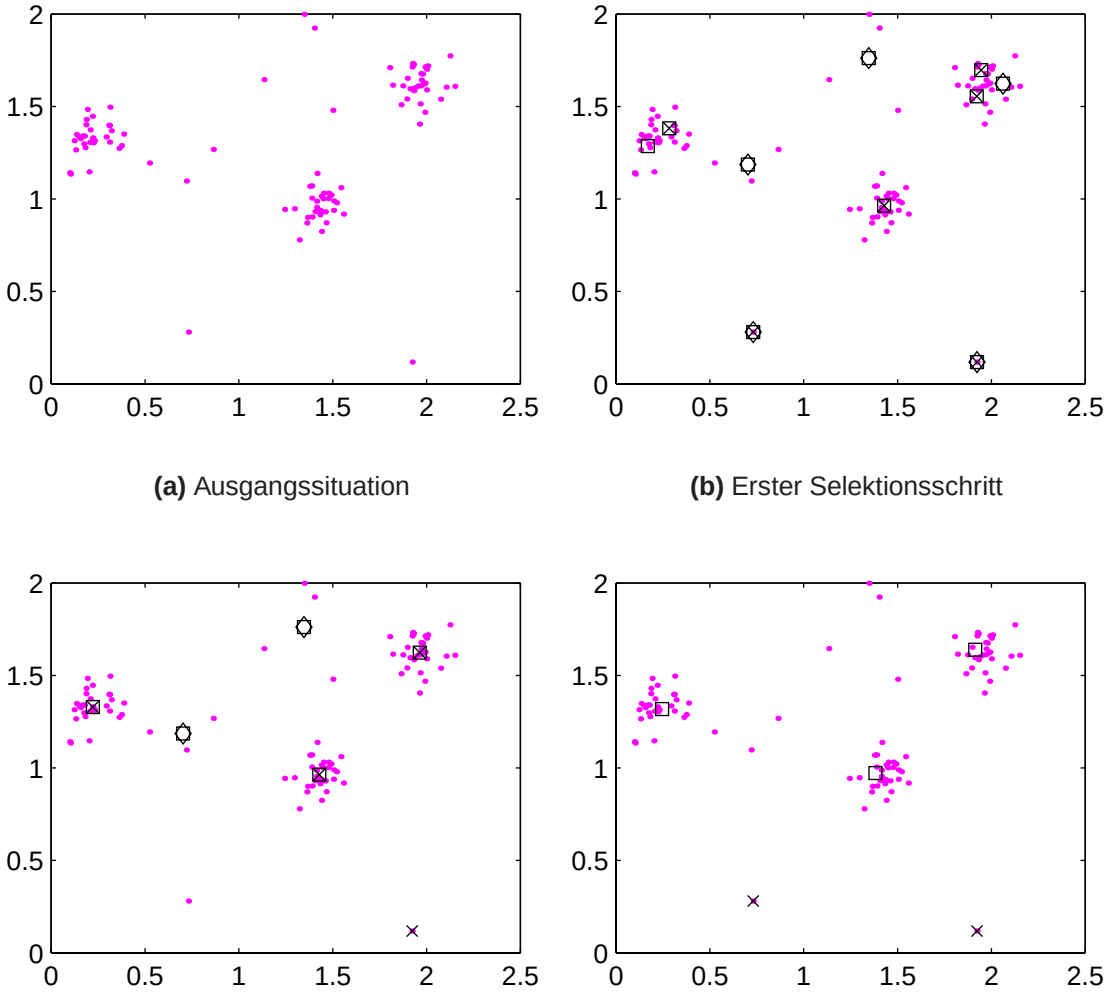
(a) Ausgangssituation

(b) Erster Selektionsschritt

(c) Zweiter Selektionsschritt

(d) Dritter und letzter Selektionsschritt

Abbildung 4.3: Quantisierungsergebnisse der einzelnen Iterationsschritte des *MDL* Algorithmus ohne Detektion von *outlier*. 3 Datencluster (je 30 Punkte, $\sigma = 0.07$) und 10 gleichverteilte Punkte (Symbole: Datenpunkte ($\cdot$), zu entfernende Referenzvektoren (Achteck), gesperrte Referenzvektoren ($\boxtimes$)und übriggebliebene Referenzvektoren ($\square$)).



(a) Ausgangssituation

(b) Erster Selektionsschritt

(c) Zweiter Selektionsschritt

(d) Dritter und letzter Selektionsschritt

Abbildung 4.4: Quantisierungsergebnisse der einzelnen Iterationsschritte des *MDL* Algorithmus mit Detektion von *outlier*. 3 Datencluster (je 30 Punkte, $\sigma = 0.07$) und 10 gleichverteilte Punkte (Symbole: Datenpunkte ($\cdot$), zu entfernende Referenzvektoren (Achteck), gesperrte Referenzvektoren ($\boxtimes$), übriggebliebene Referenzvektoren ($\square$) und *outlier* ($\times$)).

4.3 MDL für Fuzzy Clustering

Das bisherige *Minimum Description Length* Modell berücksichtigte nur *crisp Clustering* Algorithmen. Leonardis und Bischof [22] definierten ein *MDL* Modell, welches die Netzwerkkomplexität von *Radial Basis Function* (RBF) Netzwerken reduziert. Selb, Bischof und Leonardis [38] wendeten das *MDL* Modell für *RBF* Netzwerke auf den *fuzzy k-means* Algorithmus an und zeigten dadurch, daß dieses *MDL* Modell auch ein allgemeines Modell für *fuzzy Clustering* Algorithmen ist. Dabei wird wieder die Trainingsmenge S kodiert und die Reduktion der Netzwerkkomplexität (Anzahl Referenzvektoren) als Minimierung der Beschreibungslänge interpretiert.

4.3.1 Beschreibungslänge

Man geht von einer vorgegebenen Netzwerkgröße m aus und definiert einen Vektor $\mathbf{h}^T = \{h_1, \ldots, h_m\}$, mit $h_i \in \{0, 1\}$, der angibt, ob sich ein Referenzvektor $\mathbf{w}_i$ im aktuellen Modell befindet oder nicht. Nun gilt es jenen Vektor $\hat{\mathbf{h}}$ zu finden, der die Beschreibunglänge der Datenmenge S minimal werden läßt. Die Beschreibungslänge definiert man

$$L(S(\mathbf{h})) = L(S(\overline{A(\mathbf{h})})) + L(S(A(\mathbf{h}))). \tag{4.31}$$

Die gesamte Kodierungslänge $L(S(\mathbf{h}))$ setzt sich aus zwei Termen zusammen. Erstens aus der Kodierungslänge $L(S(A(\mathbf{h})))$ jener Datenpunkte aus S, die durch das von der Hypothese $\mathbf{h}$ repräsentierte Modell $A(\mathbf{h})$ erklärt werden und zweitens durch die Kodierungslänge $L(S(\overline{A(\mathbf{h})}))$ jener Datenpunkte aus S, welche nicht durch das Kodierungsmodell $A(\mathbf{h})$ beschrieben werden. Die Menge der Datenpunkte $S(\overline{A(\mathbf{h})})$ ist vergleichbar mit der Definition der *outlier* aus den vorhergehenden Abschnitten, jedoch werden die nicht durch das Modell $A(\mathbf{h})$ erklärten Datenpunkte, nun in das *MDL* Kriterium miteinbezogen. Für die Definition der gesamten Kodierungskosten kann man folgenden Zusammenhang angeben (Leonardis und Bischof [22]).

$$L(S(\mathbf{h})) = d\ K_1(n - n(\mathbf{h})) + K_2\xi(\mathbf{h}) + d\ K_3 N(\mathbf{h}), \tag{4.32}$$

wobei die gesamte Anzahl Datenpunkte mit $n = |S|$ und die Anzahl Datenpunkte, welche durch das Modell $A(\mathbf{h})$ erklärt werden, mit $n(\mathbf{h})$ angegeben werden. Der erste Term gibt deswegen die Kosten der Kodierung der Datenpunkte an, die von $A(\mathbf{h})$

nicht erklärt werden. Die Residuen, welche durch das Modell $A(\mathbf{h})$ entstehen, werden mit $\xi(\mathbf{h})$ notiert. Die Kosten für die Modellerklärung (Anzahl Netzwerkparameter) repräsentiert der Term $N(\mathbf{h})$. Die Parameter K_1, K_2 und K_3 dienen als Gewichte der einzelnen Kostenterme. Sie können entweder als informationstheoretisches Maß (in bit) oder als relative Gewichtungsterme der Kosten zueinander definiert werden. Abschnitt 4.3.5 wird sich mit diesen Parametern noch näher beschäftigen.

4.3.2 MDL Kriterium

Analog zum *MDL* Kriterium für *crisp Clustering* Algorithmen wird versucht, mittels *pruning* ein vorliegendes Netzwerk in seiner Komplexität zu reduzieren. Das bedeutet, daß eine Hypothese $\hat{\mathbf{h}}$ zu finden ist, welche die Beschreibungslänge der Datenmenge S (Gleichung (4.32)) minimal werden läßt.

$$L(S(\hat{\mathbf{h}})) = \min_{\mathbf{h}} L(S(\mathbf{h})). \tag{4.33}$$

Nachdem die Anzahl n der Datenpunkte S konstant ist, kann das Minimierungsproblem auf ein Maximierungsproblem umformuliert werden.

$$\begin{aligned} F(S(\hat{\mathbf{h}})) &= \max_{\mathbf{h}} F(S(\mathbf{h})) && (4.34)\\ &= \max_{\mathbf{h}} \Big(d\ K_1 n(\mathbf{h}) - K_2 \xi(\mathbf{h}) - d\ K_3 N(\mathbf{h}) \Big). && (4.35) \end{aligned}$$

Nun ist aber die Bestimmung des besten Modells $A(\mathbf{h})$ keine triviale Aufgabe, da es 2^m verschiedene Hypothesen zu berücksichtigen gilt. Deswegen versucht man einen Formalismus zu finden, der die Kodierungskosten auf die einzelnen Referenzvektoren verteilt und man mittels eines Greedy Algorithmus das Optimierungsproblem aus Gleichung (4.35) löst. Dabei werden die einzelnen Datenpunkte entsprechend ihrer *membership* Werte (siehe Abschnitt 2.3) in *Domains* R_i eingeteilt.

$$R_i = \{\mathbf{x} | u^p_{\mathbf{x}i} > \Theta\}. \tag{4.36}$$

Der *Domain* R_i des Referenzvektors $\mathbf{w}_i$ beinhaltet all jene Datenpunkte $\mathbf{x}$, welche eine größere Zugehörigkeit zu $\mathbf{w}_i$ haben, als der vorgegebene Schwellwert Θ. Die Bestimmung dieses Schwellwertes kann aufgrund des erwarteten Fehlers oder des

gewünschten Quantisierungsergebnisses erfolgen. Der Exponent notiert den Grad der Unschärfe (siehe Abschnitt 2.3). Für die Anzahl $n(\mathbf{h})$ (Gleichung (4.35)) der durch das Modell $A(\mathbf{h})$ erklärter Datenpunkte ergibt sich $n(\mathbf{h}) = \bigcup_{i=1}^{m} R_i$.

Es ist wichtig anzumerken, daß die verschiedenen *Domains* nicht paarweise disjunkt sein müssen. Die Anzahl Datenpunkte in der *Domain* R_i notieren wir mit n_i und die Anzahl Datenpunkte in den paarweise überlappenden *Domains* $R_i \cap R_j$ notieren wir mit n_{ij}.

$$n_i = |R_i|. \tag{4.37}$$

$$n_{ij} = |R_i \cap R_j|. \tag{4.38}$$

Sei weiters ξ_i das von *Domain* R_i erzeugte Residuum und ξ_{ij} jenes der Überlappdomain $R_i \cap R_j$, dann kann Gleichung (4.34) umformuliert werden in

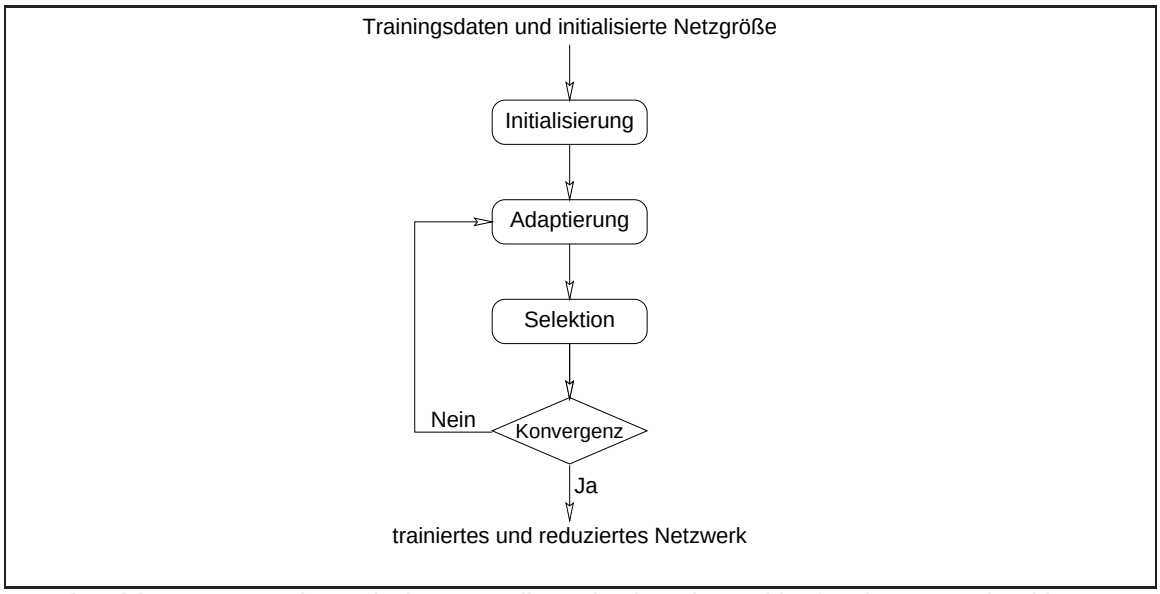
$$\begin{aligned} F(S(\mathbf{h})) &= [h_1, \ldots, h_m] \begin{bmatrix} q_{11} & \cdots & q_{1i} & \cdots & q_{1m} \\ \vdots & & \vdots & & \vdots \\ q_{i1} & \cdots & q_{ii} & \cdots & q_{im} \\ \vdots & & \vdots & & \vdots \\ q_{m1} & \cdots & q_{mi} & \cdots & q_{mm} \end{bmatrix} \begin{bmatrix} h_1 \\ \vdots \\ h_i \\ \vdots \\ h_m \end{bmatrix} \\ &= \mathbf{h}^T \mathbf{Q} \mathbf{h}, \end{aligned} \tag{4.39}$$

wobei nur die paarweisen Überlappbereiche berücksichtigt wurden (Bischof und Leonardis [22]). Diese Annahme ist solange gültig, solange die Überlappbereiche höherer Ordnung vernachlässigbaren Einfluß auf die Funktion $F(S(\mathbf{h}))$ haben. Die Matrix $\mathbf{Q}$ ist eine Kosten-Nutzen-Matrix, wobei die Diagonalterme q_{ii} die Kosten-Nutzen-Terme der Referenzvektoren $\mathbf{w}_i$ und die Terme $q_{ij}, i \neq j$, die Kosten-Nutzen-Terme der Überlappbereiche angeben. Die Matrix $\mathbf{Q}$ ist symmetrisch.

$$q_{ii} = d\ K_1 n_i - K_2 \xi_i - d\ K_3 N_i. \tag{4.40}$$

$$q_{ij} = \frac{-d\ K_1 n_{ij} + K_2 \xi_{ij}}{2}, \qquad i \neq j. \tag{4.41}$$

Der Term N_i gibt die Kosten des Referenzvektors $\mathbf{w}_i$ an und ist proportional der Dimensionalität der Datenpunkte. Die Lösung der Maximierung von Glei-



Algorithmus 4.2: Schematische Darstellung des iterativen Ablaufes des *MDL* Algorithmus für *fuzzy Clustering*.

chung (4.39) kann der Klasse der kombinatorischen Optimierungsproblemen (*quadratic boolean problem*) zugeordnet werden. Die Anzahl Lösungen steigt exponentiell mit der Anzahl der Problemparameter (Anzahl Referenzvektoren). Dadurch ist es sinnvoll, Methoden zu verwenden, die zwar nicht garantieren können, ein globales Maximum zu finden, aber mit vertretbarem Aufwand gute Lösungen finden. In den Experimenten wurde ein *Greedy* Algorithmus und der *Tabu Search* Algorithmus (Glover und Laguna [17]; siehe auch Lokketangen [23]) verwendet.

4.3.3 MDL Algorithmus

Im Gegensatz zum *MDL* Modell für *crisp Clustering* Algorithmen (Abschnitt 4.2.4) wird ein Referenzvektor nicht entfernt, wenn er die Beschreibungslänge in einem Iterationsprozeß verringert, sondern es wird in jedem Iterationsprozeß jene Hypothese $\hat{\mathbf{h}}$ gesucht, welche ein Minimum der Beschreibungslänge (Maximum von Gleichung (4.39)) in diesem Iterationsschritt zur Folge hat. Algorithmus 4.2 zeigt den schematischen Ablauf dieses iterativen *MDL* Algorithmus für *fuzzy Clustering* Algorithmen. Der Algorithmusablauf entspricht sehr jenem für *crisp Clustering* Algorithmen (Abschnitt 4.2.4). Eine Unterscheidung in *inlier* und *outlier* gibt es explizit nicht, sondern entsteht, wenn es einen Datenpunkt gibt, der zu keiner *Domain* R_i zugeordnet wird. Dabei hat der Schwellwert Θ der *membership* Werte entscheidenden Einfluß.

Initialisierung: Die Initialisierung der Menge der Referenzvektoren A wird mit der Menge der Datenpunkte vorgenommen, das heißt einen Referenzvektor pro Datenpunkt. Abschnitt 4.4 beschäftigt sich mit der Möglichkeit, mit einer geringeren, initialen Menge der Referenzvektoren zu beginnen.

Adaptierung: Für die Berechnung der Referenzvektorpositionen kann jeder *fuzzy Clustering* Algorithmus verwendet werden. Auch bei *fuzzy Clustering* Algorithmen ist es nicht notwendig, sie bis zur vollständigen Konvergenz zu trainieren (siehe Abbschnitt 4.4).

Selektion: Bevor die Hypothese $\hat{\mathbf{h}}$ gesucht werden kann, müssen zunächst die Elemente der Kosten-Nutzen-Matrix $\mathbf{Q}$ berechnet werden. Dabei kann die Symmetrieeigenschaft von $\mathbf{Q}$ genutzt werden, die Elemente rascher zu bestimmen. Mittels *Greedy* oder *Tabu Search* Algorithmus wird die Funktion $F(S(\mathbf{h}))$ (Gleichung (4.39)) maximiert und das entsprechende $\hat{\mathbf{h}}$ zur Bestimmung der überflüssigen Referenzvektoren verwendet. Es werden jene Referenzvektoren $\mathbf{w}_j$ aus der Menge A entfernt, welche ein entsprechendes Element $\hat{h}_i = 0$ besitzen.

Konvergenz: Die Konvergenz ist erreicht, wenn im aktuellen Iterationsschritt keine Referenzvektoren entfernt wurden und die Änderung der Referenzvektorpositionen (*Adaptierungsschritt*) unter eine vorgegebene Schranke fällt.

Das *MDL* Modell für *fuzzy Clustering* Algorithmen reduziert nicht nur die Netzwerkkomplexität, sondern liefert am Ende des iterativen Algorithmus auch die Positionen der Referenzvektoren. Das hier präsentierte Modell bildet nur einen Rahmen für die Anwendung auf *fuzzy Clustering* Algorithmen. Einen konkreten Programmablauf bietet Abschnitt 4.2.7, der exemplarisch – bis auf die Detektion der *outlier* – auch für *fuzzy Clustering* Algorithmen gültig ist. Für eine konkrete Instanzierung müssen die Parameter K_1, K_2 und K_3 und im weiteren ein Modell für die Kodierung der Residuen $\xi(\mathbf{h})$ definiert werden. Die beiden nächsten Abschnitte werden sich damit beschäftigen.

4.3.4 Kodierung der Residuen

Die Kodierung der Residuen richtet sich nach dem verwendeten *Clustering* Algorithmus und dem zu minimierenden Fehlermaß $E(S, A)$. Leonardis und Bischof [22] definieren die Residuen allgemein als

$$\xi_i = \sum_{\mathbf{x}\in R_i} [\mathbf{x}-\mathbf{w}_i]^2 = \sum_{\mathbf{x}\in R_i} \xi_i(\mathbf{x}). \tag{4.42}$$

$$\xi_{ij} = \sum_{\mathbf{x}\in R_i\cap R_j} \max\{[\mathbf{x}-\mathbf{w}_i]^2, [\mathbf{x}-\mathbf{w}_j]^2\}. \tag{4.43}$$

Wendet man das *MDL* Modell auf den *fuzzy k-means* Algorithmus (Abschnitt 2.3) an, so ergeben sich aufgrund des Fehlermaßes (Gleichung (2.10)) folgende Fehlerterme ξ_i und ξ_{ij} (Selb, Bischof und Leonardis [38]).

$$\xi_i = \sum_{\mathbf{x}\in R_i} ||\mathbf{x}-\mathbf{w}_i||^2 u_{\mathbf{x}i}^p. \tag{4.44}$$

$$\xi_{ij} = \max\{\sum_{\mathbf{x}\in R_i\cap R_j} ||\mathbf{x}-\mathbf{w}_i||^2 u_{\mathbf{x}i}^p, \sum_{\mathbf{x}\in R_i\cap R_j} ||\mathbf{x}-\mathbf{w}_j||^2 u_{\mathbf{x}j}^p\}. \tag{4.45}$$

4.3.5 Parameter K_1, K_2 und K_3

Die Parameter K_1, K_2 und K_3 können als informationstheoretische Kosten (in bit) oder als relative Gewichtung der Kostenterme zueinander definiert werden. K_1 gibt die durchschnittliche Anzahl bits an, die für jene Datenpunkte benötigt werden, die nicht durch das Modell $A(\mathbf{h})$ erklärt werden. Der Parameter K_2 gibt die durchschnittliche Anzahl an bits zur Kodierung der Residuen an und K_3 stellt die Kosten eines Netzwerkparameters (Parameter eines Referenzvektors) dar. Im relativen Fall wird $K_1 = 1$ gesetzt und die beiden anderen relativ dazu. K_3 wird dabei ein proportionales Maß der Dimensionalität der Datenpunkte sein. Für den *fuzzy k-means* Algorithmus könnte man $K_3 = 1.2$ setzen, da die Referenzvektoren die selbe Dimensionalität wie die Datenpunkte besitzen, jedoch noch die Varianz der Datencluster als zusätzlicher Kostenfaktor anfällt.

Unter der Annahme, daß die Datenpunkte unabhängig und identisch verteilt sind und die Fehlerverteilung einer Gaußverteilung mit Mittelwert 0 und Varianz σ^2 entspricht, entstehen Kodierungskosten für die Residuen (für den eindimensionalen Fall; d=1)

$$L(\epsilon(S,A)) = -\log_2\Big(\prod_{x\in S} p(x)t\Big) \tag{4.46}$$

$$= -\sum_{x\in S} \log_2 \left(\frac{t}{\sqrt{2\pi}\sigma} e^{-\frac{x^2}{2\sigma^2}} \right) \quad (4.47)$$

$$= -\log_2 e^{-\frac{1}{2}\sum_{x\in S}\frac{x^2}{\sigma^2}} + n(\log_2(\sigma) + \frac{1}{2}\log_2(2\pi) - \log_2 t), \quad (4.48)$$

wobei t das Diskretisierungsintervall der Wahrscheinlichkeitsverteilung $p(x)$ notiert. Schätzt man für die Varianz $\sigma^2 = \frac{1}{n}\sum_{x\in S} x^2$, dann ergibt sich für Gleichung (4.48)

$$L(\epsilon(S,A)) = \frac{n}{2}\log_2(e) + n(\log_2(\sigma) + \frac{1}{2}\log_2(2\pi) - \log_2 t) \quad (4.49)$$

$$= n\Big(\frac{\log_2(\frac{\sigma^2 2\pi e}{t^2})}{2}\Big). \quad (4.50)$$

Die Kosten für die Kodierung der Residuen wird in Gleichung (4.40) durch den Term $K_2\xi_i = K_2 n\sigma^2$ dargestellt, den man mit $L(\epsilon(S,A))$ (Gleichung (4.50)) gleichsetzt und man für K_2 erhält

$$K_2 = \frac{\log_2(\frac{\sigma^2 2\pi e}{t^2})}{2\sigma^2}. \quad (4.51)$$

Bei der Herleitung von K_2 wurde lediglich der eindimensionale Fall betrachtet. Geht man davon aus, daß die Verteilung der Residuen in jeder Dimension gleich und voneinander unabhängig ist, kann die Berechnung des Parameters K_2 aus Gleichung (4.51) auch für den mehrdimensionalen Fall verwendet werden. K_2 ist demnach abhängig von der Standardabweichung der Störung der Datenpunkte und verringert sich mit steigender Störung σ der Datenpunkte. Wenn sich σ in einem bestimmten Werteintervall bewegt, kann man K_2 in diesem Bereich abschätzen und konstant belassen. Die Berechnung von K_2 mittels Gleichung (4.51) kann als oberer Grenzwert für K_2 angesehen werden und der *MDL* Algorithmus sollte stabil bleiben, wenn niedrigere Werte für K_2 verwendet werden. Im relativen Fall der Parameterdefinition muß K_2 aus Gleichung (4.51) noch mit der mittleren Anzahl bits für die Kodierung der Trainingsdaten (K_1) dividiert werden.

Bei der Bestimmung der Gewichtungsterme ist es weniger wichtig, exakte Werte zu ermitteln, da sie einerseits nur Mittelwerte darstellen und andererseits die Robustheit des Algorithmus keine exakten Werte verlangt. Sind die Parameter jedoch sehr entscheidend für das *Clustering* Ergebnis, sollte man überprüfen, ob das Problem nicht selbst schlecht konditioniert ist. Dies tritt auf, wenn die Annahme, daß nur die paarweisen Überlappbereiche berücksichtigt werden müssen, nicht gültig

ist, da die Überlappbereiche höherer Ordnung doch großen Einfluß auf das Optimierungsproblem haben.

4.4 Verbesserung des Laufzeitverhaltens

In den beiden Abschnitten 4.2 und 4.3 wurden 2 *MDL* Modelle inklusive konkreter Instanzierungen für *crisp Clustering* und *fuzzy Clustering* präsentiert. Experimentelle Ergebnisse (Bischof, Leonardis und Selb [4]; Selb [35]) haben gezeigt, daß 10% bis 20% der Datenpunkte zur Initialisierung der Menge A der Referenzvektoren ausreichen, ohne jedoch dabei wichtige *Clusterzentren* zu "vergessen". Diese reduzierte Anzahl ist jedoch von der Verteilung der Datenpunkte abhängig, wodurch keine pauschale Definition für die Anzahl der initialen Referenzvektoren getroffen werden kann. Viel mehr wäre es interessant, einen Formalismus zu finden, der aufgrund der gegebenen Information der Verteilung der Datenpunkte, nämlich der Menge S, eine gute Schätzung der initialen Referenzvektoren und deren Positionen angibt. Anschließend kann der *MDL* Algorithmus die initiale Menge an Referenzvektoren reduzieren. Eine große Anzahl an initialen Referenzvektoren bedingt einen hohen Berechnungsaufwand des *MDL* Algorithmus, da sowohl der *Adaptierungsschritt* als auch der *Selektionsschritt* einen von der Anzahl der Referenzvektoren abhängigen Berechnungsaufwand besitzt.

Bei der Analyse des *MDL* Algorithmus zeigte sich auch, daß der *Adaptierungsschritt*, speziell in den beiden ersten Iterationen des *MDL* Algorithmus, einen hohen Berechnungsaufwand besitzt. Da zu Beginn die Positionen der Referenzvektoren durch den *Clustering* Algorithmus stark korrigiert werden müssen, werden viele Iterationen des *Clustering* Algorithmus benötigt. Leonardis und Bischof [22] und [3] zeigen, daß es nicht notwendig ist, den *Clustering* Algorithmus bis zur vollständigen Konvergenz zu trainieren, sondern nur einige wenige. Für den *MDL* Algorithmus sowohl für *crisp Clustering* Algorithmen als auch für *fuzzy Clustering* Algorithmen hat dies keine Auswirkung auf die erhaltene Anzahl und Position der Referenzvektoren.

In Abschnitt 2.7 wurde das *Growing Neural Gas* (GNG) Modell vorgestellt, welches mit seiner wachsenden Eigenschaft versucht, die Anzahl der Referenzvektoren an die Verteilung der Datenpunkte anzupassen. Es ist einfach Parameter des *GNG* Modells zu finden, welche in mehr Referenzvektoren resultieren, als die "richtige" Anzahl. Nun kann man die durch das *GNG* Modell gefundenen Referenzvektoren dazu verwenden, die initiale Menge A der Referenzvektoren zu setzen

und darauf den *MDL* Algorithmus anzuwenden. Dadurch ist gewährleistet, daß alle wichtigen Datencluster berücksichtigt und keiner "vergessen" wurde. Damit kann das *GNG* Modell dazu verwendet werden, eine verteilungsabhängige Initialisierung des *MDL* Algorithmus zu ermitteln.

Im folgenden werden kurz einige Untersuchungen der Laufzeit des *MDL* Algorithmus präsentiert, welche verdeutlichen sollen, wie mit den beiden oben dargestellten Möglichkeiten der Berechnungsaufwand des *MDL* Algorithmus reduziert werden kann. Dafür werden einerseits die Anzahl Iterationen des *Clustering* Algorithmus und andererseits die Anzahl benötigter Fließkommaoperationen (*flops*) als Geschwindigkeitsmaß verwendet. Wesentlich aussagekräftiger sind natürlich die benötigten Fließkommaoperationen.

4.4.1 Crisp Clustering

Im Falle eines *crisp Clustering* Algorithmus hätte eine reduzierte Anzahl an Iterationsschritten des *Clustering* Algorithmus eine schlechtere Positioniergenauigkeit der Referenzvektoren zur Folge. Dies wirkt sich auf den *MDL* Algorithmus in zusätzlichen Kosten der Residuen aus, wodurch in diesem Iterationsschritt des *MDL* Algorithmus weniger überflüssige Referenzvektoren anfallen und entfernt werden. Es kann also nie der Fall auftreten, daß zu viele Referenzvektoren entfernt werden. Durch die *Adaptierungsschritte* der folgendenen *MDL* Algorithmusiterationen werden diese schlechten Positioniergenauigkeiten der Referenzvektoren korrigiert, wodurch der Fehlerterm wieder geringer wird und somit überflüssige Referenzvektoren entdeckt und entfernt werden. Wird also der *Clustering* Algorithmus nicht bis zur vollständigen Konvergenz trainiert, entstehen zwar mehr Iterationsschritte des *MDL* Algorithmus, aber die Anzahl der Iterationsschritte des *Clustering* Algorithmus über den gesamten *MDL* Algorithmus reduziert sich.

Im folgenden wurde der *k-means* Algorithmus zur Adaptierung der Referenzvektoren verwendet, wobei er im *Adaptierungsschritt* mit nur einem Iterationsschritt ausgeführt wurde. Die Datenmenge bestand aus 5 Zentren, je 50 Datenpunkte und Standardabweichung $\sigma = 0.07$ um die *Clusterzentren*. Auch wurde keine Detektion der *outlier* angewandt. Es wurden 9 verschiedene Versionen der Datenmenge erzeugt und die Ergebnisse über die 9 Datenmengen gemittelt. Tabelle 4.1 zeigt die Aufwände des *MDL* Algorithmus einmal bei vollständiger Konvergenz und einmal bei nur einem Iterationsschritt des *Clustering* Algorithmus. Beide Varianten

	Minimum	Mittel	Maximum	Minimum	Mittel	Maximum
Konvergenz	9.0	14.29	22.0	0.90	1.16	1.74
eine Iteration	5.0	6.43	10.0	0.58	0.70	1.12

Tabelle 4.1: Rechenaufwand des *MDL* Algorithmus bei voller Konvergenz des *k-means* Algorithmus und bei lediglich einer Iteration pro *MDL* Iteration. Die ersten 3 Spalten geben die Anzahl Iterationen des *k-means* Algorithmus für den gesamten *MDL* Durchlauf und die letzten 3 Spalten geben die Anzahl Fließkommaoperationen (*flops*) des *MDL* Algorithmus an (Anzahl *flops* $\times 10^6$).

wurden aber mit der selben Menge A der Referenzvektoren initialisiert (30 initiale Referenzvektoren). Zwar ist die Anzahl der *MDL* Iterationen angestiegen, aber die Anzahl der rechenintensiven *k-means* Iterationen wurde um den Faktor 2 reduziert. Auch die durchschnittliche Anzahl *flops* aller *k-means* Iterationsschritte wurde um den Faktor 2 niedriger.

Die hohe Anzahl der Iterationsschritte des *Clustering* Algorithmus wird aber auch durch die Initialisierung mittels *GNG* Modell reduziert. Einerseits wird eine reduzierte, initiale Anzahl der Referenzvektoren entsprechend der Datenverteilung gesetzt und andererseits hat das *GNG* die Positionen der Referenzvektoren gut vorbestimmt. Beide Fälle wirken sich positiv auf die benötigte Anzahl an Iterationsschritte des *Clustering* Algorithmus aus. Tabelle 4.2 zeigt die Anzahl Iterationen bzw. *flops* des *k-means* Algorithmus bei Anwendung auf eine Datenmenge mit 8 gaußverteilten ($\sigma = 0.07$) Datencluster (je 80 Datenpunkte) mit 30 gleichverteilten Datenpunkten (*outlier*). Es wurden 5 Varianten erzeugt und die Ergebnisse über diese Varianten gemittelt. Bei der Initialisierung mittels *GNG* Modell wurden 15 Referenzvektoren gefunden und verwendet. Zum Vergleich wurde die Menge A der Referenzvektoren mit 10% bzw. 20% der Datenpunkte zufällig initialisiert.

Das *GNG* Modell initialisiert den *MDL* Algorithmus sehr gut, da die gesuchten 8 Datencluster nur leicht überschätzt werden. In den beiden anderen Fällen wird die Menge A 5 - 9 mal größer initialisiert als mittels *GNG* Modell. Dies ist auch der Grund, weshalb die ersten beiden Iterationen, im Verhältnis zur Gesamtanzahl, sehr viele *k-means* Iterationen benötigen. Bezüglich der gesamten Anzahl an *k-means* Iterationsschritten benötigt der *MDL* Algorithmus mit *GNG* Initialisierung um den Faktor 3 - 4 weniger *k-means* Iterationen. Noch deutlicher ist der Unterschied, wenn man die Anzahl der Fließkommaoperationen betrachtet. Hier wird um den Faktor 9 - 13 mehr Rechenleistung benötigt, als bei der Initialisierung mittels *GNG* Modell.

	Init.	1. Iter.	2. Iter.	Gesamt	1. Iter.	2. Iter.	Gesamt
GNG	15	9.4	6.6	13.0	1.14	0.38	1.80
10 %	67	17.3	14.3	53.0	7.02	2.93	17.12
20 %	134	11.0	14.0	56.6	10.07	6.63	23.56

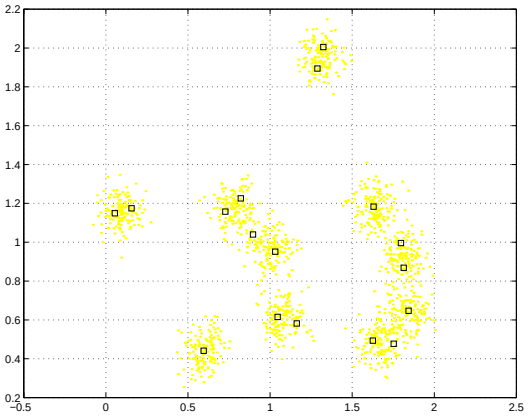
Tabelle 4.2: Rechenaufwand des *MDL* Algorithmus bei voller Konvergenz des k-*means* Algorithmus bei unterschiedlicher Initialisierung. Die erste Spalte gibt die initiale Anzahl Referenzvektoren an. Die nächsten 3 Spalten repräsentieren die Anzahl k-*means* Iterationen während der ersten beiden *MDL* Iterationen und des gesamten *MDL* Durchlaufes. Die letzten 3 Spalten geben die Anzahl Fließkommaoperationen (*flops*) des k-*means* Algorithmus an (Anzahl *flops* $\times 10^6$).

In Tabelle 4.2 wurden nur die Aufwände des k-*means* Algorithmus berücksichtigt. Das *GNG* Modell hatte aber bei der Initialisierung einen Berechnungsaufwand von $0.33(\pm 0.01) \times 10^6$ *flops*.

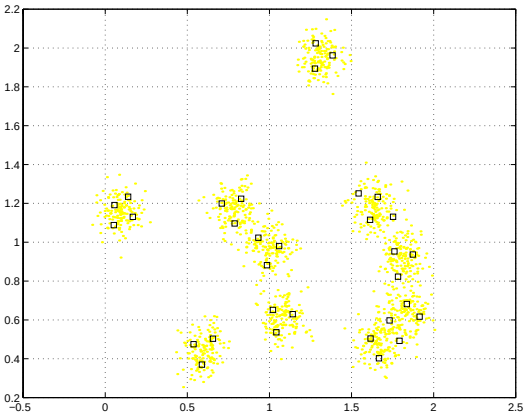
Abbildung 4.5 zeigt die gefundenen Referenzvektoren bei Anwendung des *MDL* Algorithmus mit k-*means*, des *GNG* Algorithmus und deren Kombination. Die Trainingsmenge bestand aus 10 *Cluster* mit je 80 Datenpunkte mit $\sigma = 0.07$ ohne *outlier*. Die Abbildungen 4.5(a) und 4.5(b) zeigen das Endergebnis nach Anwendung des *GNG* Algorithmus nach kurzem (17 Referenzvektoren) bzw. langem (32 Referenzvektoren) Training. Man erkennt, daß die Anzahl der resultierenden Referenzvektoren von der Anzahl der Iterationsschritte des *GNG* Modells abhängig ist. Die beiden nächsten Abbildungen zeigen das Endergebnis des *MDL* Algorithmus bei Initialisierung der Menge A mit 150 (Abbildung 4.5(c)) und 75 (Abbildung 4.5(d)) Referenzvektoren. Diese beiden Ergebnisse sind identisch. Die beiden unteren Abbildungen zeigen die Ergebnisse der Kombination von *MDL* und *GNG*. Abbildung 4.5(e) stellt das Resultat des *MDL* Algorithmus bei Initialisierung von A mit dem Ergebnis aus Abbildung 4.5(a) und Abbildung 4.5(f) jenes *MDL* Ergebnis bei Initialisierung mit den Referenzvektoren aus Abbildung 4.5(b). Auch sind die Resultate identisch und auch identisch zu den Abbildungen 4.5(c) und 4.5(d).

4.4.2 Fuzzy Clustering

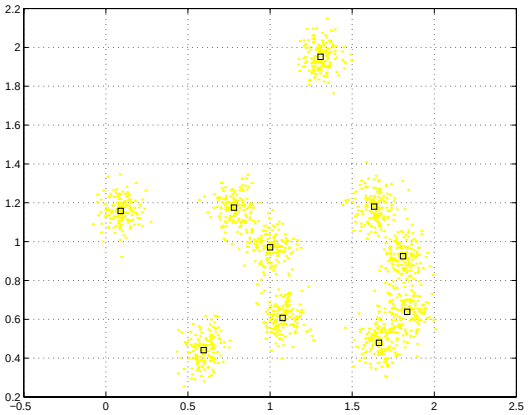
Bei *fuzzy Clustering* Algorithmen berechnet der *Adaptierungsschritt* nicht nur die Positionen der Referenzvektoren, sondern auch die *membership* Werte, welche sich aus diesen Positionen ergeben. Für den *fuzzy* k-*means* Algorithmus berechnen sich



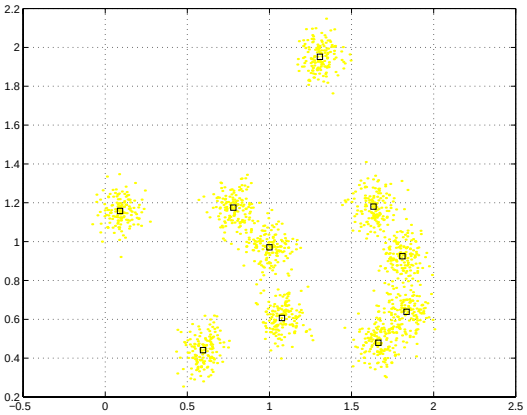
(a) kurzes Training mit *GNG*



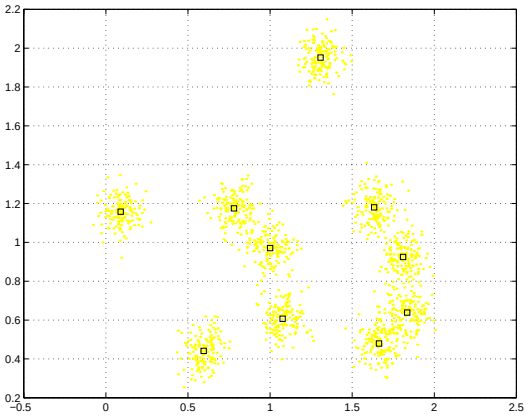
(b) langes Training mit *GNG*



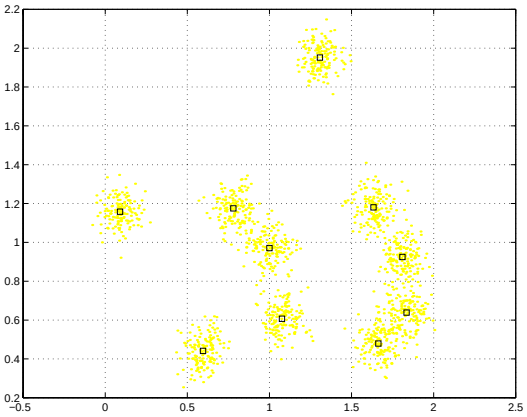
(c) *MDL* (150 Referenzvektoren)



(d) *MDL* (75 Referenzvektoren)



(e) *MDL* (Initialisierung mit (a))



(f) *MDL* (Initialisierung mit (b))

Abbildung 4.5: Vektorquantisierung unter verschiedenen Initialisierungen. Punkte symbolisieren Datenpunkte und Rechtecke stellen Referenzvektoren dar.

die *membership* Werte wie in Gleichung (2.8). Um den Berechnungsaufwand für den *MDL* Algorithmus für *fuzzy Clustering* Algorithmen zu reduzieren, können die selben Maßnahmen getroffen werden, wie für *crisp Clustering* Algorithmen.

Jedoch sollte bei der Reduktion der Iterationsschritte des *Clustering* Algorithmus bei der ersten Iteration des *MDL* Algorithmus ein voller Durchlauf des *Clustering* Algorithmus durchgeführt werden. Betrachtet man Gleichung (4.40) und (4.41), so erkennt man, daß bei nicht ausreichend genauer Positionierung der Referenzvektoren die Fehlerterme $K_2\xi_i$ und $K_2\xi_{ij}$ betragsmäßig größer als die Nutzenterme $K_1 n_i$ und $K_1 n_{ij}$ werden können. Speziell, wenn der ersten Iteration eine zufällige Initialisierung vorangegangen ist, tritt der Fall auf, daß alle Diagonalterme q_{ii} negativ und alle Überlappterme q_{ij} positiv werden. Jeder unser Optimierungsalgorithmen hätte die leere Menge als Lösungsmenge zurückgegeben und der *MDL* Algorithmus wäre bereits in seiner ersten Iteration mit einem falschen Ergebnis beendet worden.

Auch werden die Anzahl Iterationsschritte des *Clustering* Algorithmus nicht auf eine Iteration pro *MDL* Algorithmusiteration beschränkt, sondern auf 3. Prinzipiell ist es egal, wieviele Iterationen des *Clustering* Algorithmus verwendet werden, jedoch ist die Berechnung der Kosten-Nutzen-Matrix $\mathbf{Q}$ und die Lösung des Maximierungsproblems in jedem Iterationsschritt des *MDL* Algorithmus sehr aufwendig. Um die Anzahl der Iterationen des *MDL* Algorithmus deswegen gering zu halten, haben Experimente gezeigt, daß 3 Iterationen des *Clustering* Algorithmus einen guten Kompromiß darstellen.

Um die Einsparung des Berechnungsaufwandes zu demonstrieren, wurden 10 verschiedene Versionen einer Datenmenge mit 7 Datencluster mit je 50 Datenpunkten ($\sigma_1 = 0.07$) und 3 Datencluster mit je 80 Datenpunkten ($\sigma_2 = 0.15$), denen jeweils eine Gaußverteilung der Datenpunkte zugrunde liegt, erzeugt und verwendet (Abbildung 5.3). Die Tabelle 4.3 zeigt die Anzahl Iterationen des *fuzzy k-means* Algorithmus und die benötigten *flops*, wobei sowohl 10% und 20% der Trainingsdaten zur Initialisierung der Menge A der Referenzvektoren herangezogen wurden, als auch das *GNG* Modell. Das *GNG* Modell hatte einen durchschnittlichen Aufwand von $0.38(\pm 0.01) \times 10^6$ *flops*. Weiters wird zwischen 2 Optimierungsalgorithmen (*Greedy* und *Tabu Search*) unterschieden. Die ersten 6 Zeilen der Tabelle 4.3 zeigen den Berechnungsaufwand bei voller Konvergenz des *fuzzy k-means* Algorithmus und die letzten 6 Zeilen der Tabelle 4.3 bei Anwendung von lediglich 3 *fuzzy k-means* Iterationen.

Die Parameter des *GNG* Modells wurden so gewählt, daß 16 Referenzvektoren

		Iterationen fkm			flops fkm ($\times 10^6$)			Iterationen MDL			flops MDL ($\times 10^6$)		
		1. Iteration	2. Iteration	Gesamt	1. Iteration	2. Iteration	Gesamt	Minimum	Mittel	Maximum	Minimum	Mittel	Maximum
Tabu	10%	94.0	90.2	214.8	81.96	29.70	116.75	3.0	3.8	5.0	87.85	116.75	199.47
	20%	60.6	48.2	234.4	105.69	31.72	164.03	4.0	6.0	10.0	130.44	164.03	196.28
	GNG	45.6	20.8	75.2	10.94	3.33	15.41	2.0	2.6	3.0	11.16	15.41	19.39
Greedy	10%	94.0	97.4	225.4	81.96	31.56	119.11	3.0	4.0	5.0	84.11	119.11	199.47
	20%	60.6	47.6	250.2	105.69	31.46	167.29	4.0	6.4	11.0	146.93	167.29	192.18
	GNG	45.6	17.2	67.4	10.94	2.76	14.35	2.0	2.4	3.0	11.16	14.35	18.75
Tabu	10%	94.0	3.0	126.4	81.96	0.99	86.94	6.0	11.8	27.0	67.74	86.94	136.47
	20%	60.6	3.0	105.0	105.69	1.93	114.15	11.0	15.8	27.0	82.59	114.15	146.82
	GNG	45.6	3.0	63.0	10.94	0.47	13.35	5.0	6.8	8.0	10.41	13.35	15.66
Greedy	10%	94.0	3.0	120.4	81.96	0.97	86.30	6.0	9.8	14.0	63.60	86.30	136.47
	20%	60.6	3.0	103.2	105.69	1.93	113.70	9.0	15.2	27.0	82.50	113.70	146.86
	GNG	45.6	3.0	63.0	10.94	0.46	13.35	5.0	6.8	8.0	10.41	13.35	15.67

Tabelle 4.3: Rechenaufwand des *fuzzy k-means* und des *MDL* Algorithmus mit voller Konvergenz bzw. 3 Iterationen des *fuzzy k-means* Algorithmus. Die ersten 6 Spalten geben die Anzahl Iterationen und Fließkommaoperationen (*flops*) des *fuzzy k-means* Algorithmus während den ersten beiden *MDL* Iterationen und den Aufwand für den gesamten *MDL* Durchlauf an. Die letzten 6 Spalten geben die minimale, durchschnittliche und maximale Anzahl *MDL* Iterationen und *flops* des gesamten *MDL* Algorithmus an. Weiters wird sowohl der Optimierungsalgorithmus (*Greedy* und *Tabu Search*) als auch die initiale Anzahl Referenzvektoren unterschieden (10% und 20% der Trainingsmenge bzw. *GNG*). Die ersten 6 Zeilen gelten für den Fall, daß der *fuzzy k-means* Algorithmus mit voller Konvergenz und die letzten 6 Zeilen für den Fall, daß der *fuzzy k-means* Algorithmus mit 3 Iterationen ausgeführt wird.

zur Initialisierung verwendet wurden. Dadurch, daß die Positionen der Referenzvektoren durch das *GNG* Modell sehr gut vorgegeben werden, braucht der erste Selektionsschritt lediglich 50% bis 75% an *fuzzy k-means* Iterationen und gar nur ca. 10% an Fließkommaoperationen bei Initialisierung mit *GNG* (Tabelle 4.3; ersten 6 Zeilen). Der Unterschied zwischen den beiden Optimierungsalgorithmen ist minimal. Wird anstelle der vollen Konvergenz lediglich 3 Iterationsschritte des *fuzzy k-means* Algorithmus verwendet, erkennt man eine weitere Reduktion an Iterationsschritten gegenüber der *GNG* Initialisierung von 20% bis 60%, das eine Reduktion von 30% der Fließkommaoperationen zur Folge (Tabelle 4.3; letzten 6 Zeilen). Auch hier ist der Unterschied der Optimierungsalgorithmen vernachlässigbar gering.

4.5 Wachstumskomponente für den MDL Algorithmus

Der oben beschriebene *MDL* Algorithmus ist ein reiner *pruning* Algorithmus, der terminiert, wenn kein Referenzvektor mehr entfernt werden kann. Nun kann aber der Fall auftreten, daß man bereits ein trainiertes Netzwerk für eine *Clustering* Aufgabe besitzt und sich die Datenmenge (geringfügig) ändert. Besitzt die neue Datenmenge weniger *Clusterzentren* als die alte, würde der *MDL* Algorithmus die Referenzvektoren neu trainieren und die überflüssigen entfernen. Es ist aber wichtig anzumerken, daß dies nur funktioniert, wenn sich der aktuelle Zustand des neuralen Netzwerkes nicht in einem anderen lokalen Minimum befindet, als die neue Lösung erfordert. Fritzke [14] beschreibt dazu eine Modifikation des *k-means* Algorithmus, die es erlaubt, aus lokalen Minima auszubrechen, jedoch auch kein globales Minimum garantiert. Wendet man diese Modifikation auf den *MDL* Algorithmus für *crisp Clustering*, bei dem man den *k-means* Algorithmus verwendet, an, so könnte man das Problem, in einem lokalen Minima hängen zu bleiben, abschwächen und die veränderte Datenmenge richtig *clustern*.

Für den Fall, daß die veränderte Datenmenge aus mehr *Cluster* besteht als die alte, wird es dem *MDL* Algorithmus nicht möglich sein, das neurale Netzwerk richtig zu trainieren, da er keine Referenzvektoren hinzufügen kann. Dies ist jedoch recht einfach zu lösen, indem man einfach einen Referenzvektor hinzufügt und einen weiteren Reduktionsschritt startet. War der zusätzliche Referenzvektor nicht notwendig, wird er durch die Reduktion entfernt und sonst verbleibt er in der Menge

der Referenzvektoren. Das Hinzufügen von Referenzvektoren kann man nun so oft wiederholen, wie eine Reduktion der Menge der Referenzvektoren einsetzt und der *MDL* Algorithmus wieder in einen stabilen Zustand der Menge an Referenzvektoren konvergiert. Es drängt sich jedoch die Frage auf, an welche Position der neue Referenzvektor positioniert wird. Dabei übernehmen und modifizieren wir die Idee von Fritzke [14], der die Position des Referenzvektors in die Nähe jenes Referenzvektors setzt, der den maximalen Quantisierungsfehler aufweist.

Exemplarisch zeigen wir dies für den *MDL* Algorithmus für *crisp Clustering*, wobei als *Clustering* Algorithmus der *k-means* Algorithmus verwendet wird. Sei im folgenden $E(S, \mathbf{w}_i)$ jener Fehler, der von Referenzvektor $\mathbf{w}_i$ und seinen zugeteilten Datenpunkten erzeugt wird.

$$E(S, \mathbf{w}_i) = \sum_{\mathbf{x} \in S_i} ||\mathbf{x} - \mathbf{w}_i||^2. \tag{4.52}$$

Somit läßt sich der Referenzvektor $\mathbf{w}_q$ mit maximalem Fehler wie folgt bestimmen.

$$q = \arg \max_{i \in \{1,\ldots,m\}} E(S, \mathbf{w}_i). \tag{4.53}$$

Im ersten Moment würde man den neuen Referenzvektor $\mathbf{w}_n$ an die selbe Position wie $\mathbf{w}_q$ setzen. Somit wäre nicht eindeutig, welcher Datenpunkt der *Voronoi Region* S_q welchem der beiden Referenzvektoren $\mathbf{w}_q$ und $\mathbf{w}_n$ zugeordnet ist. Diesen Fall haben wir in Abschnitt 2.1 ausgeschlossen. Eine Lösung wäre, den neuen Referenzvektor $\mathbf{w}_n$ ein wenig versetzt zu $\mathbf{w}_q$, sagen wir um einen *offset* Vektor $\Delta\mathbf{w}_q$, anzuordnen. Die Standardabweichung σ eines *Clusters* kann man mit

$$\sigma = \sqrt{\frac{E(S, \mathbf{w}_i)}{|S_i|}} \tag{4.54}$$

abschätzen. Der *offset* Vektor $\Delta\mathbf{w}_q$ sollte auf jeden Fall kleiner sein, als die Standardabweichung σ. Die Position des neuen Referenzvektors $\mathbf{w}_n$ kann mit folgendem Zusammenhang berechnet werden.

$$\mathbf{w}_n = \mathbf{w}_q + \Delta\mathbf{w}_q, \tag{4.55}$$

$$\Delta\mathbf{w}_q = \epsilon \sqrt{\frac{E(S, \mathbf{w}_q)}{n}} \mathbf{u}, \tag{4.56}$$

wobei $|S_q|$ durch die gesamte Anzahl der Datenpunkte n ersetzt wurde, um die Standardabweichung σ sicher zu unterschätzen ($n \geq |S_q|$). Der Parameter ϵ gibt zusätzlich die Länge des *offset* Vektors an und sollte im Bereich $0 < \epsilon \ll 1$ liegen. Weiters stellt $\mathbf{u}$ einen zufälligen Vektor, der auf der d-dimensionalen Einheitskugel liegt, dar und die Richtung des *offset* Vektors vorgibt. Fritzke [14] schlägt auch vor, die Richtung des *offset* Vektors in Richtung des größten Eigenwertes der Kovarianzmatrix der Datenpunkte der *Voronoi Region* S_q zu legen.

Algorithmus 4.3 zeigt den um die Wachstumskomponente erweiterten *MDL* Algorithmus für *crisp Clustering*. Dabei ergeben sich 2 weitere Verarbeitungsschritte:

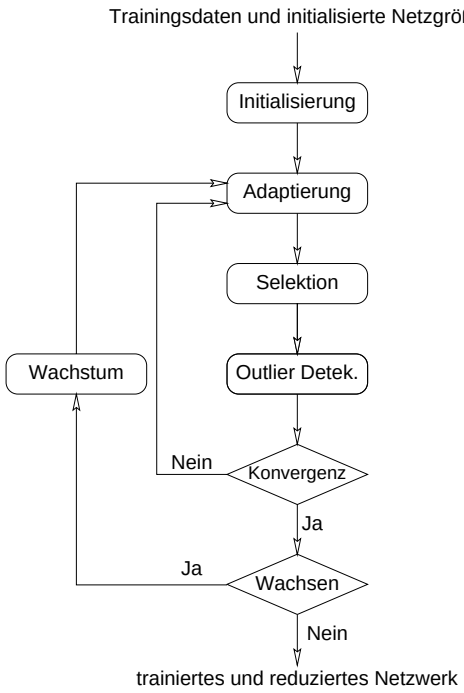
Wachsen: Es wird überprüft, ob ein weiterer Referenzvektor hinzugefügt werden muß. Sei m' jene Netzgröße, die das Netzwerk vor dem letzten Hinzufügen eines Referenzvektors hatte. Im Schritt **Initialisierung** wird $m' = 0$ gesetzt. Dann kann

$$m' < m \tag{4.57}$$

als Kriterium definiert werden, um einen weiteren Referenzvektor hinzuzufügen. Die aktuelle Netzwerkgröße wird mit m notiert. Dadurch, daß $m' = 0$ bis zum erstens Überprüfen des Wachstumskriterium gilt, wird beim ersten Erreichen der **Konvergenz** zum Schritt **Wachstum** weitergegangen.

Wachstum: Als erstes wird $m' = m$ gesetzt, um die Netzwerkgröße vor dem Hinzufügen des Referenzvektors zu speichern. Anschließend wird ein neuer Referenzvektor $\mathbf{w}_n$ (Gleichung (4.55)) hinzugefügt und die Menge der *inlier* $I = S$ und die Menge der *outlier* $O = \emptyset$ gesetzt, damit im folgenden Schritt (**Adaptierung**) die neue Menge der Referenzvektoren auf die gesamte Datenmenge S trainiert werden kann und die bereits detektierten *outlier* keine Verfälschung der Quantisierung zur Folge haben. Dies kommt einem Neustart des *MDL* Algorithmus mit guter Initialisierung der Referenzvektoren gleich.

Das Kriterium $m' < m$ um einen neuen Referenzvektor hinzuzufügen, ist nicht mehr gegeben, wenn zumindest einer der hinzugefügten Referenzvektoren wieder entfernt wurde. Exemplarisch wurde die Anwendung einer Wachstumskomponente für *crisp Clustering* demonstriert, wobei die Anwendung für *fuzzy Clustering* lediglich der Fehlerterm auf



1: Initialisiere $m' := 0$.
2: Initialisiere $Abbruch := 0$.
3: **Initialisierung**.
4: **repeat**
5: **repeat**
6: **Adaptierung**.
7: **Selektion**.
8: **Outlier Detektion**.
9: **until Konvergenz** des *MDL* Algorithmus.
10: Bestimme aktuelle Netzgröße m.
11: **if** $m' < m$ **then**
12: $m' := m$.
13: Hinzufügen des neuen Referenzvektors $\mathbf{w}_n$ (Gleichung (4.55)).
14: Setze $I = S$ und $O = \emptyset$.
15: **else**
16: $Abbruch := 1$.
17: **end if**
18: **until** $Abbruch = 1$.

Algorithmus 4.3: Schematische Darstellung des iterativen Ablaufes des *MDL* Algorithmus für *crisp Clustering* und der zusätzlichen Wachstumskomponente.

$$E(S, \mathbf{w}_i) = \sum_{\mathbf{x} \in S} (u_{\mathbf{x}i})^p ||\mathbf{x} - \mathbf{w}_i||^2. \tag{4.58}$$

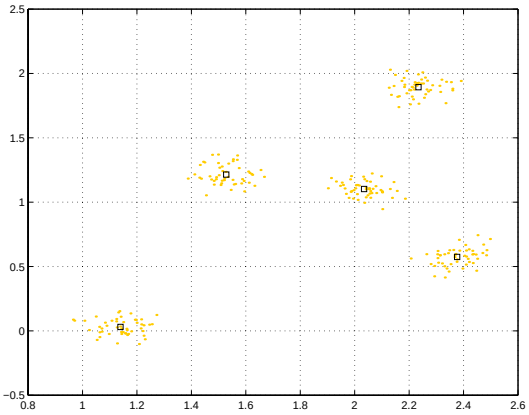
geändert werden muß. Zur Berechnung der Position des neuen Referenzvektors wird man zum Fall des *crisp Clustering* zurückkehren und die Datenpunkte in *Voronoi Regionen* einteilen und dann die Berechnung wie in Gleichung (4.55) vornehmen.

Dadurch, daß der *MDL* Algorithmus für *crisp Clustering* zusätzlich eine Wachstumskomponente besitzt, wird dieser zusätzlich robuster gegenüber der Initialisierung. Bisher war gefordert, daß mehr Referenzvektoren nach der Initialisierung benötigt werden, als man *Cluster* finden möchte. Dies ist nun nicht mehr gefordert, da der *MDL* Algorithmus selbst die Anzahl Referenzvektoren erhöhen kann und die Positionierung aufgrund des Quantisierungsfehlers vornimmt. Jedoch ist es erstrebenswert, mit einer größeren Menge an Referenzvektoren zu beginnen, da die Reduktion mehrerer Referenzvektoren parallel erfolgen kann und das Hinzufügen lediglich einzeln erfolgt. Somit dürfte die Reduktion nach guter Initialisierung (mittels *GNG*) in der Regel rascher erfolgen, als das Hinzufügen einzelner Referenzvektoren. Dies ist aber von der Verteilung der Datenpunkten und der Anzahl *Cluster* abhängig. Weiters erlaubt der Start mit einer genügend großen Menge an Referenzvektoren, daß man lokale Minima eher vermeiden kann oder in gute lokale Minima hineinkonvergiert. Dadurch, daß die Wachstumskomponente nicht eine so große Möglichkeit an Lösungen abdeckt, wird dies durch Hinzufügen von Referenzvektoren in der Regel eher nicht erreicht.

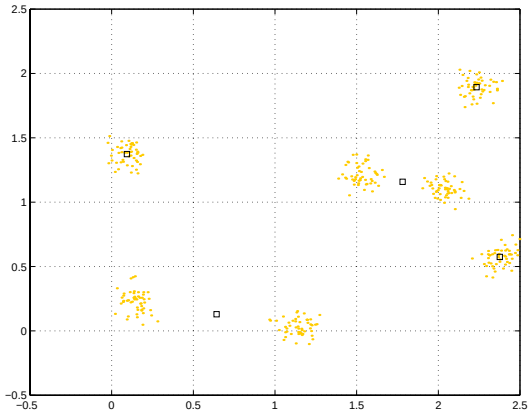
4.5.1 Beispiel eines Durchlaufes

Es wurden 5 *Cluster* mit jeweils 50 Datenpunkten und einer Gaußverteilung ($\sigma = 0.07$) erzeugt. Anschließend wurde der *MDL* Algorithmus für *crisp Clustering* mit dem *k-means Clustering* Algorithmus auf diese Daten angewandt, wodurch sich nach 5 Iterationen das Ergebnis aus Abbildung 4.6(a) einstellte (zufällige Initialisierug mit 30 Referenzvektoren). Alle *Clusterzentren* werden durch einen Referenzvektor markiert. Nun fügen wir 2 weitere Datencluster mit ebenfalls je 50 Datenpunkten hinzu (Gaußverteilung; $\sigma = 0.07$). Diese sind in Abbildung 4.6(b) im Bereich $x \in [0, \ldots, 0.5]$ und $y \in [0, \ldots, 1.5]$ zu finden. Weiters zeigt Abbildung 4.6(b) bereits das Ergebnis des *MDL* Algorithmus, wenn vom Ergebnis aus Abbildung 4.6(a) weitertrainiert wurde. An diesem Punkt ist die Konvergenz des *MDL* Algorithmus erreicht und ein neuer Referenzvektor wird hinzugefügt. Der Parameter ϵ wird

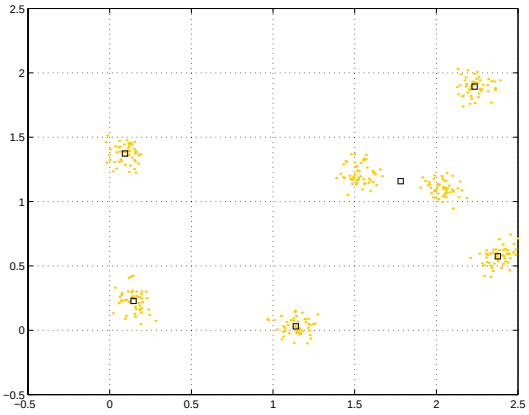
auf 0.0001 gesetzt. Das Ergebnis des *MDL* Algorithmus mit diesem zusätzlichen Referenzvektor ist in Abbildung 4.6(c) dargestellt. Weiters werden noch 2 weitere Referenzvektoren hinzugefügt (Abbildung 4.6(d) und 4.6(e)). In Abbildung 4.6(e) ist der dritte zusätzliche Referenzvektor vom *MDL* Algorithmus als überflüssig markiert worden und wird in weiterer Folge entfernt (Abbildung 4.6(f)). Dadurch, daß der dritte Referenzvektor entfernt wurde, wird das Kriterium für weiteres Wachstum und der *MDL* Algorithmus inklusive der Wachstumskomponente terminieren, wodurch das Endergebnis der 7 markierten *Datencluster* in Abbildung 4.6(f) zu sehen ist. Es wurden demnach 3 Referenzvektoren hinzugefügt, wobei der dritte durch den *MDL* Algorithmus entfernt wurde und so erkannt wird, daß kein zusätzlicher Referenzvektor notwendig ist.



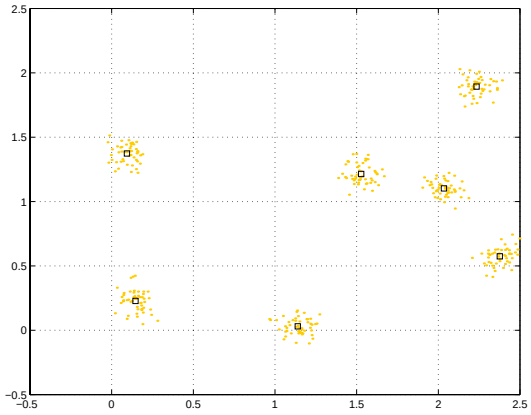
(a) Konvergenz des *MDL* Algorithmus



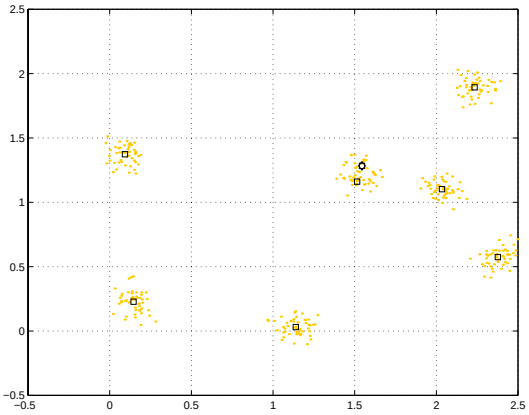
(b) neue Daten



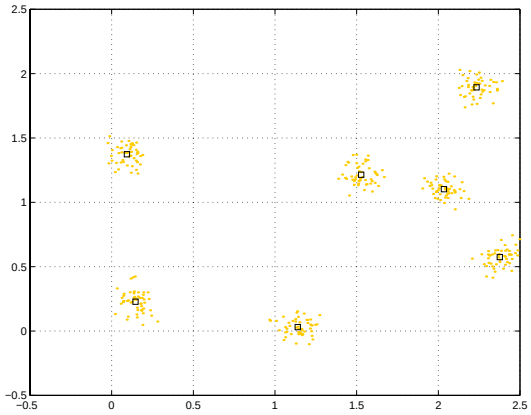
(c) ein neuer Referenzvektoren



(d) zwei neue Referenzvektoren



(e) drei neue Referenzvektoren



(f) Entfernung des dritten Referenzvektors

Abbildung 4.6: Hinzufügen neuer Referenzvektoren. (a) zeigt das Ergebnis des *MDL* Algorithmus. In (b) wurden 2 neue *Cluster* hinzugefügt ($x \in [0, \ldots, 0.5], y \in [0, \ldots, 1.5]$). (c) zeigt, wie ein Referenzvektor hinzugefügt wurde ((d) 2 neue Referenzvektoren, (e) 3 neue Referenzvektoren). In (e) symbolisiert der achteckige Referenzvektor, jenen überflüssigen Referenzvektor, der in (f) entfernt wurde.

Kapitel 5

Anwendungen

Inhaltsangabe

Im vorangegangenem Kapitel wurde das *Minimum Description Length* (MDL) Modell für *crisp Clustering* und *fuzzy Clustering* Algorithmen präsentiert. Viele

Anwendungen verwenden Vektorquantisierung, um ihre Trainingsdaten zu partitionieren und wichtige Eigenschaften (Verteilung der Trainingsdaten) dadurch zu erkennen. Im folgenden werden einige beispielhafte Anwendungen präsentiert, die vom Einsatz des *MDL* Modells profitieren.

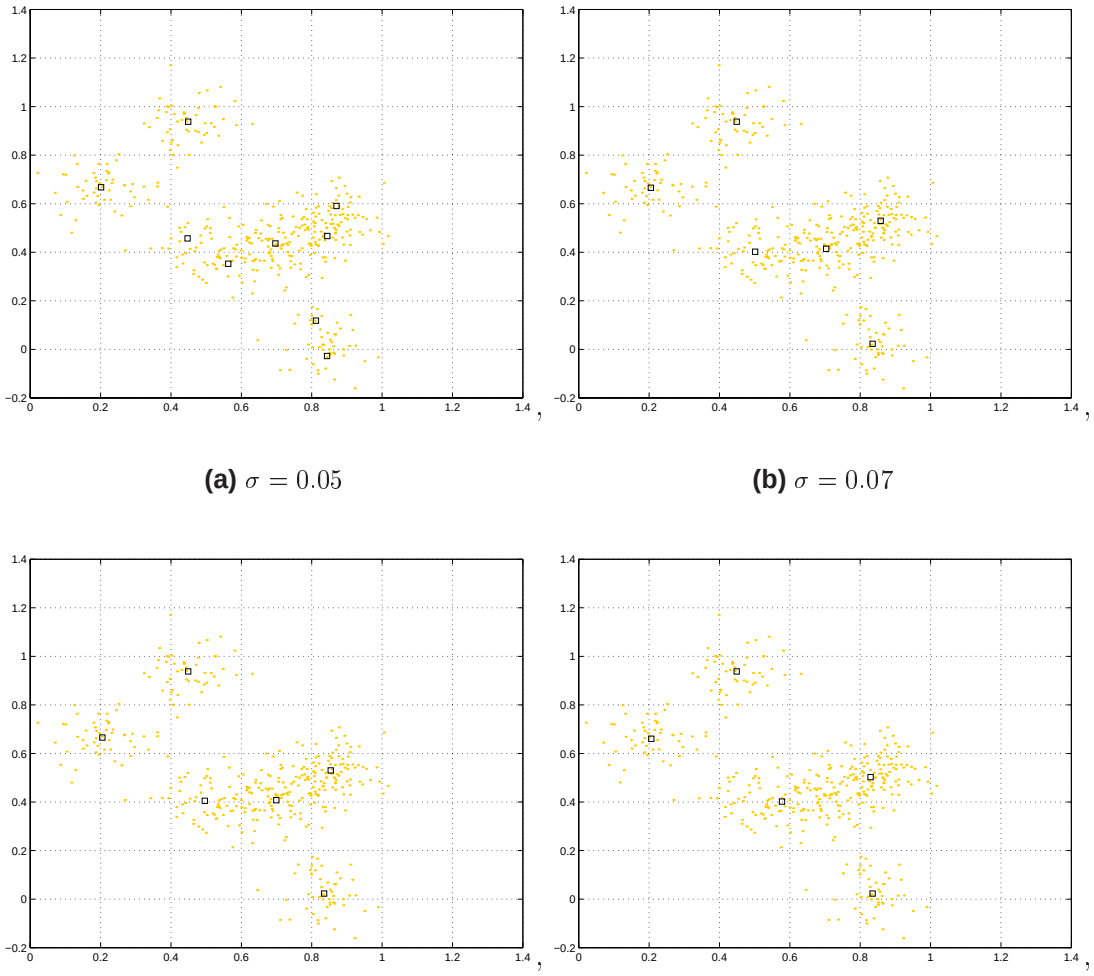
5.1 Vektorquantisierung

Dieser Abschnitt, soll zeigen, wie der *MDL* Algorithmus für *Clustering* Probleme anhand konkreter Datensätze angewandt wird. Dabei spielt die Bestimmung der Algorithmusparameter eine entscheidende Rolle.

5.1.1 Crisp Vektorquantisierung

Als Vertreter der *crisp Clustering* Algorithmen wählen wir den *k-means* Algorithmus und bei der Kodierung der Residuen nehmen wir eine Gaußverteilung an. Für die Kodierung der Indizes wählen wir die optimale Kodierungslänge. Um den *MDL* Algorithmus für *Clustering* anwenden zu können, muß zuerst eine Schätzung der Fehlerverteilung gemacht werden. Bei Annahme einer Gaußverteilung betrifft dies die Schätzung der Varianz σ^2.

Wir wählen eine Datenmenge mit 6 Datencluster (je 50 Datenpunkte, $\sigma = 0.07$) ohne *outlier* und initialisieren den *MDL* Algorithmus mit 50 Referenzvektoren. Abbildung 5.1 zeigt die Endergebnisse unter Verwendung verschiedener Standardabweichungen $\sigma \in \{0.05, 0.07, 0.09, 0.15\}$. Abbildung 5.1(b) zeigt das erwartete Endergebnis, wenn man den Parameter σ entsprechend der Standardabweichung der Residuen wählt (6 Cluster). Wählt man ein zu geringes σ, erhält man zu viele Referenzvektoren (9 Cluster bei $\sigma = 0.05$; Abbildung 5.1(a)). Bei einem zu hohem σ tritt der Fall ein, daß zu wenig Referenzvektoren gefunden werden (5 Cluster bei $\sigma = 0.15$; Abbildung 5.1(d)). Jedoch erhält man auch das erwartete Endergebnis, wenn der Parameter σ nicht exakt bestimmt wird (6 Cluster bei $\sigma = 0.09$; Abbildung 5.1(c)). Der *MDL* Algorithmus ist demnach robust gegenüber dem Parameter σ, wenn dieser nicht exakt bestimmt wurde. Abbildung 5.2 zeigt in Abhängigkeit des Parameters σ die erhaltene Anzahl Referenzvektoren, die in einem großen Bereich ($\sigma \in \{0.05, \ldots, 0.14\}$) konstant bleibt. Diese Eigenschaft kann man ausnutzen, um – speziell im hochdimensionalen Raum der Datenpunkte – festzustellen, ob die minimale Anzahl Referenzvektoren gefunden wurde. Weiters bietet sich die Möglichkeit



(a) $\sigma = 0.05$ **(b)** $\sigma = 0.07$

(c) $\sigma = 0.09$ **(d)** $\sigma = 0.15$

Abbildung 5.1: Endergebnisse bei Vektorquantisierung einer Datenmenge (6 Cluster, je 50 Datenpunkte, $\sigma = 0.07$) bei Anwendung verschiedener $\sigma \in \{0.05, 0.07, 0.09, 0.15\}$ ohne *outlier* Detektion.

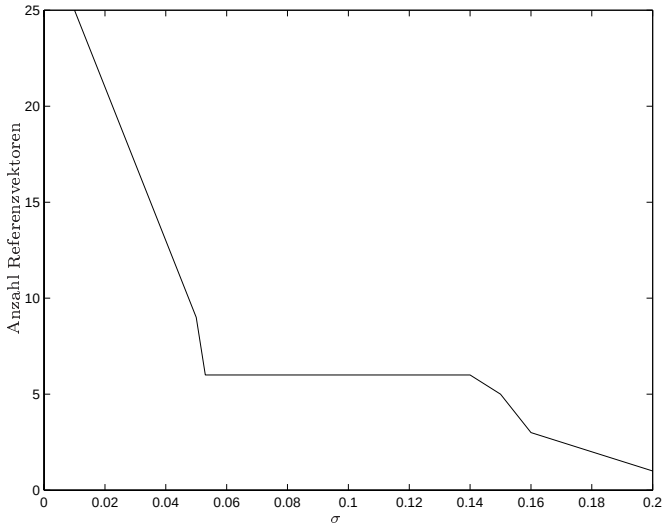


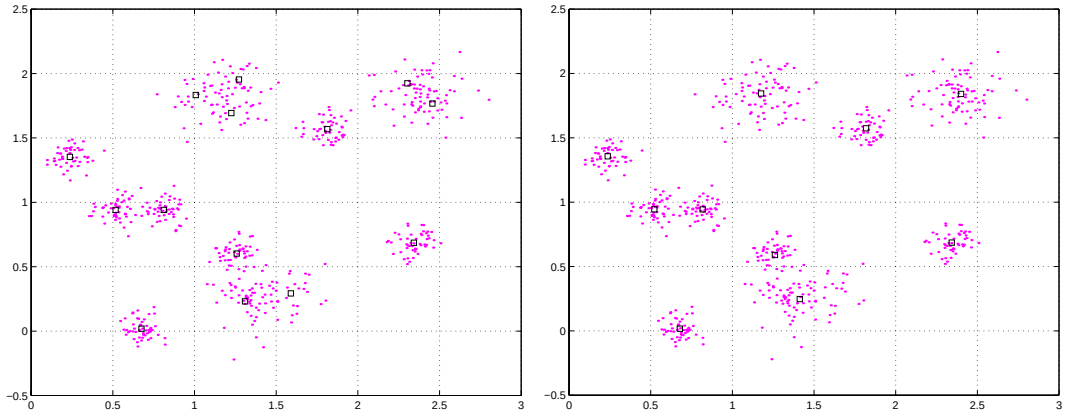
Abbildung 5.2: Anzahl Referenzvektoren in Abhängigkeit der beim *MDL* Algorithmus verwendeten Standardabweichung σ. Datenpunkte wie in Abbildung 5.1.

an, mittels Parameter σ das Quantisierungsergebnis (Anzahl Referenzvektoren) zu steuern und so einen hierarchischen *Clustering* Algorithmus aufzubauen.

Wie in Abschnitt 4.2.6 beschrieben, wurde bei der Kodierung der Residuen eine Diskretisierungsgenauigkeit η gefordert, welche Einfluß auf die Kodierungslänge der Residuen hat. Betrachtet man die Kostenfunktionen der Residuen (Gleichung (4.29) und (4.25)), so erkennt man, daß die Diskretisierungsgenauigkeit lediglich einen konstanten Term $-d\ \log_2(\eta)$, wobei d die Dimensionalität der Trainingsdaten angibt, der Kostenfunktion der Residuen darstellt. Somit ergibt sich, daß der Zuwachs der Kodierungskosten bei einer 10 mal höheren Genauigkeit lediglich um einen konstanten Term $d\ \log_2(10)$ (additiv!) zunimmt.

5.1.2 Fuzzy Vektorquantisierung

Um den Unterschied zwischen dem *MDL* Modell für *fuzzy Clustering* und dem *MDL* Modell für *crisp Clustering* zu demonstrieren, wurde eine Datenmenge erzeugt, welche aus unterschiedlich großen Datencluster besteht. Sie besteht aus 7 Datencluster mit je 50 Datenpunkten, welche je einer Gaußverteilung mit $\sigma_1 = 0.07$ entsprechen. Weiters besteht die Datenmenge aus 3 Datencluster mit je 80 Datenpunkten und einem $\sigma_2 = 0.15$ (Abbildung 5.3). Für beide Modelle initialisieren wir die Menge der Referenzvektoren mit 100 zufälligen Datenpunkten, wobei für beide die selbe



(a) *k-means* **(b)** *fuzzy k-means*

Abbildung 5.3: Ergebnis des *MDL* Algorithmus mit *k-means* und *fuzzy k-means* auf Datenmenge mit unterschiedlichen Clustergrößen (7 Datencluster mit je 50 Datenpunkten und $\sigma_1 = 0.07$ und 3 Datencluster mit je 80 Datenpunkten und $\sigma_2 = 0.15$).

Initialisierung verwendet wurde. Die Kodierung der Datenpunkte erfolgt mit 32 bit.

Zunächst wenden wir den *k-means* Algorithmus mit *MDL* Modell für *crisp Clustering* auf diese Datenmenge an. Der Parameter σ für den *MDL* Algorithmus bestimmen wir, indem man die Varianzen σ_1^2 und σ_2^2 mittelt und erhält $\sigma = 0.09$. Abbildung 5.3(a) zeigt das erhaltene Resultat (14 Referenzvektoren). Man erkennt deutlich, daß große Datencluster ($\sigma_2 = 0.15$) mit mehreren Referenzvektoren beschrieben werden, da der gewählte Parameter $\sigma = 0.09$ zu gering ist, um diese Datencluster mit nur einem Referenzvektor zu beschreiben. Die kleineren Datencluster ($\sigma_1 = 0.07$) werden idealerweise mit nur einem Referenzvektor bedacht, wodurch die Robustheit gegenüber dem Parameter σ gezeigt wird.

Bei der Bestimmung der Parameter des *MDL* Algorithmus für *fuzzy Clustering* verwenden wir ebenfalls ein $\sigma = 0.09$. Weiters kodieren wir die Residuen mit 4 bit, woraus sich ein Diskretisierungsfehler von $t = 1/2^4$ ergibt. Mittels Gleichung (4.51) ergibt sich ein $K_2 = 10$ (wurde noch durch 32 dividiert) und für K_3 nehmen wir einen Wert von $K_3 = 1.2$ an, welches relativ zu $K_1 = 1$ zusätzlich noch einen Kodierungsaufwand für den Parameter σ berücksichtigt. Bei einem Schwellwert $\Theta = 0.1$ findet der *MDL* Algorithmus die erwarteten 10 Datencluster (Abbildung 5.3(b)). Zwar wurde ebenfalls der Parameter $\sigma = 0.09$ verwendet, jedoch gewichten die *membership* Werte die Residuen (Gleichung (4.44) und (4.45)), wodurch weit entfernte Datenpunkte weniger Einfluß haben. Es ist auch wichtig darauf hinzuweisen,

daß der *fuzzy k-means* Algorithmus zur Berechnung der Referenzvektorpositionen jeweils die gesamte Datenmenge heranzieht und so jeder Datenpunkt in der *Domain* Auswirkung auf die Fehlerterme hat. Somit ist gezeigt, daß der *fuzzy k-means* Algorithmus Datenmengen mit unterschiedlichen Clustergrößen bei der Vektorquantisierung unterscheiden kann.

5.1.2.1 Beispiel: Iris-Datensatz

Um den *fuzzy k-means* Algorithmus im *MDL* Modell anhand einem konkreten Datensatz zu demonstrieren, wurde dafür der *Iris* (Orchideenart) Datensatz verwendet. Die *Iris* Daten beinhalten Informationen über 4 Merkmale von 3 *Iris* Spezien (Versicolor, Virginica und Setosa). Dabei wurden pro Spezies 50 Individuen als Meßobjekte herangezogen (Fisher [9]). Diese 4-dimensionalen Daten sind so beschaffen, daß ein Cluster (Setosa) sehr leicht von den beiden anderen separabel ist, hingegen die beiden anderen leicht ineinanderfließen.

Für die Bestimmung der Parameter K_1, K_2 und K_3 wurde angenommen, daß die Fehlerverteilung gaußverteilt ($\sigma = 15$ aufgrund visueller Abschätzung der Datenpunkte) ist. Die Datenpunkte werden mit 16 bit kodiert, wodurch sich der Diskretisierungsfehler von $t = 1/2^{16}$ (Residuen werden ebenfalls mit 16 bit kodiert) ergibt. Wir setzen $K_1 = 1$ und für K_2 ergibt sich aus Gleichung (4.51) ein Wert von $K_2 = 0.006$ (wurde noch durch 16 dividiert). Für K_3 wählen wir $K_3 = 1.1$, welches den erhöhten Speicheraufwand des Netzwerk berücksichtigt. Hier vernachlässigen wir die Abhängigkeit von K_2 vom Schwellwert Θ und variieren Θ für verschiedene *MDL* Durchläufe. Tabelle 5.1 zeigt die Anzahl resultierender Referenzvektoren in Abhängigkeit des Schwellwertes Θ. Mit steigendem Schwellwert Θ sinkt die Anzahl Referenzvektoren, da dieser den Einfluß weit entfernter Datenpunkte immer stärker einschränkt. Erst bei einem $\Theta > 0.6$ treten realistische (im Sinne der Klassifizierung) Werte der gefundenen Referenzvektoren auf, welche in diesem Bereich des Schwellwertes relativ konstant bleiben und so auf die "richtige" Anzahl Referenzvektoren hinweisen.

5.2 Datenkompression

Wie man leicht aus der Definition des *MDL* Prinzips (Abschnitt 3.4.3) erkennt, kann die Repräsentation der Datenpunkte durch das *code book* und die Kodierung

Schwellwert Θ	Anzahl Referenzvektoren
0.2	32
0.3	28
0.4	19
0.5	17
0.6	10
0.7	4
0.75	3

Tabelle 5.1: Anzahl Referenzvektoren bei verschiedenen Schwellwerten Θ anhand des *Iris* Datensatzes.

der Residuen als eine komprimierte Darstellung der Datenpunkte angesehen werden. Kodiert man die Datenpunkte nur über das *code book* (Referenzvektoren) und ignoriert den Fehlerterm, spricht man von *verlustbehafteter* Kompression. Wird der Fehlerterm in der Kodierung nicht ignoriert, sprich man von *verlustfreier* Kompression.

Die prinzipielle Vorgangsweise bei der Komprimierung mittels *MDL* Prinzip, soll anhand einer *Bildquantisierung* (*image quantization*) demonstriert werden. Ein Bild (Abbildung 5.4(a); 256 Graustufen) mit einer Größe von 768×512 Pixel wird in nichtüberlappende 8×8 Pixel große Blöcke zerlegt (Abbildung 5.5). Diese Blöcke werden dann als 64 dimensionale Datenvektoren interpretiert. Ersetzt man nun im Bild die Blöcke durch deren Index im *code book*, entsteht eine 96×64 große Indexmatrix. Dadurch würden alle 6144 Blöcke das Bild wieder vollständig rekonstruieren, jedoch tritt keine Kompression des Bildes auf. Im Gegenteil, es muß zusätzlich zum *code book*, die Indexmatrix gespeichert werden. Reduziert man jedoch die Größe des *code book* auf einen bestimmten Wert und läßt mittels *Clustering* Algorithmus die Positionen der Referenzvektoren bestimmen (Blöcke dienen dazu als Datenpunkte), reduziert sich der Kodierungsaufwand für das Bild.

Dieser Ablauf kann vom *MDL* Algorithmus für *crisp Clustering* durchgeführt werden, welcher sogar den Vorteil besitzt, daß die Größe des *code book* aufgrund der Bilddaten bestimmt wird (Bischof und Leonardis [3]; Bischof, Leonardis und Selb [4]). Dabei wurde das Bild *Baumstämme* aus Abbildung 5.4(a) verwendet und wie oben beschrieben in Datenvektoren zerlegt. Zum Training des *MDL* Algorithmus wurden, zugunsten der Laufzeit, durch zufälliges Ziehen die Hälfte der

(a) Originalbild

(b) reduziertes Bild

Abbildung 5.4: Bildquantisierung *Baumstämme*. Das 768×512 Bild (Abbildung (a)) wurde in 8×8 Blöcke zerteilt. Nach dem *MDL* Algorithmus bleiben 34 Blöcke (Referenzvektoren) übrig (Abbildung (b)). Die Differenz der beiden Bilder (Fehler durch Reduktion) zeigt Abbildung 5.6.

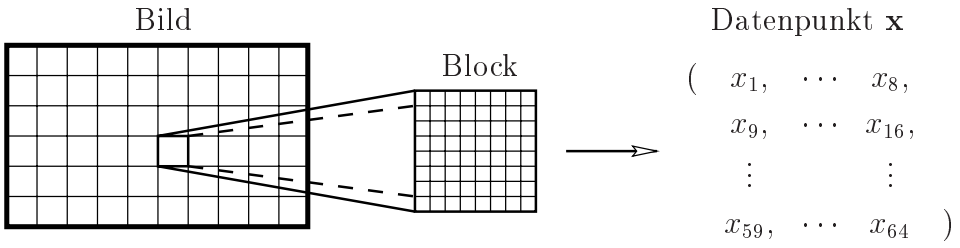


Abbildung 5.5: Unterteilung eines Bildes in 8×8 Blöcke und Notation eines Blockes als 64 dimensionaler Vektor. Diese Vektoren gelten als Datenpunkte für den *MDL* Algorithmus.

Datenmenge herangezogen. Weiters wurde zur Bestimmung der Kodierungskosten (Gleichung (4.7)) auf den Kodierungsterm der *outlier* $L(O)$ verzichtet. Bei der Kodierung der Indizes wurde die variable Kodierungslänge gewählt. Für die Verteilung der Residuen wurde Gaußverteilung angenommen ($\sigma = 20$, aufgrund visueller Abschätzung der Datenpunkte). Der *MDL* Algorithmus für *crisp Clustering* wurde mit 256, zufällig ausgewählten Datenpunkten* initialisiert und nach 7 Iterationsschritten blieben 34 Referenzvektoren übrig. Abbildung 5.4(b) zeigt das mit den 34 Referenzvektoren (ohne Fehlerterm; *verlustbehaftete* Kodierung) rekonstruierte Bild. Die Differenz der Grauwerte zwischen Originalbild und reduziertem Bild wird in Abbildung 5.6 gezeigt. Grau entspricht dem Fehler 0, helle Pixel stellen positive Differenzen und dunkle Pixel stellen negative Differenzen dar.

Wendet man sowohl auf das Originalbild, als auch auf das reduzierte Bild eine Lauflängenkodierung (RLE; *run-length encoding*) an, so reduziert sich die Dateigröße des Originals auf 6.35 bit/Pixel und des reduzierten Bildes auf 0.16 bit/Pixel. Unkomprimiert wären 8.0 bit/Pixel notwendig gewesen. Im Gegensatz zur *RLE* des Originalbildes (Komprimierungsfaktor 1.26) bringt die *RLE* des reduzierten Bildes einen Komprimierungsfaktor von 50. Der visuelle Fehler im reduzierten Bild kommt vom Blockeffekt der 8×8 Pixel großen Blöcke und kann mittels Glättungsalgorithmus reduziert werden. Durch die geringe Anzahl der Referenzvektoren ergeben sich gleiche Pixelsequenzen mit gleichen Grauwerten, die von der *RLE* erkannt und entsprechend kodiert werden.

*Dies ist möglich, wenn man annimmt, daß die Datenpunkte identisch und unabhängig verteilt sind.

Abbildung 5.6: Fehlerbild bei der Bildquantisierung *Baumstämme*.

5.3 RGB Farbsegmentierung

Man kann Vektorquantisierungsalgorithmen auch dazu verwenden, um *RGB* Farbbilder zu segmentieren. Dabei werden die einzelnen Pixel als dreidimensionale Datenpunkte aufgefaßt, wobei jede Dimension eine der Rot-, Grün- und Blauinformation darstellt. Bestimmt man eine bestimmte Anzahl Referenzvektoren, so wird man jene Farbstellvertreter finden, die den einzelnen Farben aus der Trainingsmenge am ähnlichsten sind. Man kann damit beispielsweise Bilder mit 32 bit Farbtiefe auf Bilder mit 16 bit Farbtiefe reduzieren und dabei einen minimalen Fehler der reduzierten Farbwerte erreichen. Die einzelnen Farbwerte werden dabei durch die Farbwerte ihrer nächsten Referenzvektoren ersetzt. Reduziert man die Anzahl Farben (Anzahl Referenzvektoren) im Bild immer weiter, tritt der Fall auf, daß lediglich Farbsegmente übrig bleiben.

Ein Farbsegment eines Bildes ist ein Indiz dafür, daß dieser Farbton im Bild sehr häufig auftritt. Bei einem Bild eines Menschen könnten die verschiedenen Braunwerte der Haare durch einen einzigen Braunton beschrieben werden und so einem Erkennungssystem dienen, die Haare vom restlichen Gesicht zu trennen. Bei der Bestimmung der Farbsegmente mit Hilfe der Vektorquantisierung der *RGB* Daten wird der Fehler zwischen einer Farbe und seinem nächsten Farbton minimiert. Hier stellt sich wieder die Frage, wie viele typische Farbtöne (Referenzvektoren) in einem gegebenen Bild auftreten.

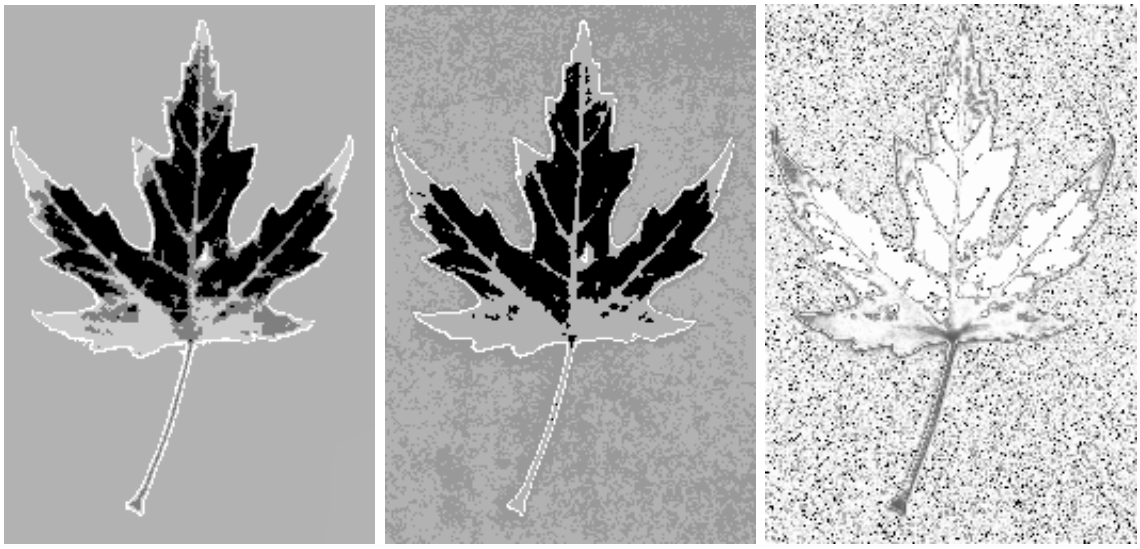
Im folgenden wird der *MDL* Algorithmus für *crisp Clustering* und für *fuzzy*

Abbildung 5.7: 24 bit *RGB* Bild eines Blattes.

Clustering am Beispiel der *RGB* Farbsegmentierung demonstriert. Dabei dient ein 24 bit *RGB* Farbbild eines Blattes als Referenz (Abbildung 5.7; 253 × 174 Pixel). Von den Datenpunkten wurden zufällig 10% (≈ 4400 Pixel) gezogen und als Trainingsmenge verwendet. Weiters wurden beide Versionen des *MDL* Algorithmus mit 50 Referenzvektoren initialisiert, wobei für beide die selbe initiale Menge der Referenzvektoren verwendet wurde. Die Kodierung der Datenpunkte erfolgte mit 32 bit pro Dimension. Nach erfolgreichem Durchlauf des *MDL* Algorithmus werden alle Farben durch die Farbe ihres nächsten Referenzvektors ersetzt.

Als *crisp Clustering* Algorithmus wurde der *k-means* Algorithmus verwendet (Bischof, Leonardis und Selb [4]) und zur Kodierung der Indizes die optimale Kodierungslänge gewählt. Als Ergebnis bleiben 5 Referenzvektoren (Farbtöne) übrig (Abbildung 5.8(a)). Dabei wird eine Farbe für den Hintergrund, 3 Farben für das Blatt und eine Farbe für den Rand des Blattes gefunden.

Bei der Verwendung des *fuzzy k-means* Algorithmus für das *MDL* Modell für *fuzzy Clustering* wurden folgende Parameter bestimmt (Selb, Bischof und Leonardis [38]). Der Parameter K_1 wird wieder auf 1 gesetzt und die beiden anderen relativ dazu ($K_2 = 0.1$, $K_3 = 1.2$). Der Parameter K_3 wurde auf einen höheren Wert als K_1 gesetzt, da er zusätzliche Kodierungskosten für den Parameter σ berücksichtigt. Der Schwellwert Θ wird auf 0.4 gesetzt. Dieser hohe Wert kommt deswegen zustande, da die *RGB* Datencluster nicht so kompakt sind, wie im Beispiel aus Abschnitt 5.1.2. Das Ergebnis wird in Abbildung 5.8(b) dargestellt. Dabei wurden ebenfalls 5 Referenzvektoren gefunden, nämlich 2 Farben für den Hintergrund, 2 Farben für das Blatt und eine für den Rand des Blattes.



(a) *k-means* **(b)** *fuzzy k-means* **(c)** *membership* Werte

Abbildung 5.8: Ergebnis der *RGB* Farbsegmentierung des Blattes mittels *MDL* Algorithmus mit *k-means* und *fuzzy k-means*. Abbildung (c) zeigt die maximalen *membership* Werte der einzelnen Punkte beim *fuzzy k-means* Algorithmus. Helle Punkte repräsentieren hohe *membership* Werte.

Die Segmentierung mittels *k-means* Algorithmus bevorzugt wegen der starren Definition der Clustergröße mittels Varianz σ^2 Datencluster, welche aus einer großen Anzahl Datenpunkte bestehen. Bei Verwendung des *fuzzy k-means* Algorithmus wird diese Schranke abgeschwächt und vermehrt auf die Farbwerte eingegangen. Das zusätzliche Farbsegment des Blattinneren von Abbildung 5.8(a) besitzt mehr Datenpunkte als das zusätzliche Farbsegment des Hintergrundes aus Abbildung 5.8(b), hingegen unterscheidet sich der Farbwert des zusätzlichen Farbsegmentes des Hintergrundes wesentlich stärker von den anderen. Dies wird untermauert, wenn man sich die *membership* Werte des *fuzzy k-means* Algorithmus ansieht. Abbildung 5.8(c) zeigt den maximalen *membership* Wert jedes Datenpunktes. Je heller ein Datenpunkt desto höher ist sein entsprechender maximaler *membership* Wert. Im Blattinneren treten Farben auf, die zu wesentlich kompakteren *Cluster* gehören. Dies zeigt sich, da die *membership* Werte sehr ähnliche Werte aufweisen. Im Hintergrund sind einige Datenpunkte, welche nicht so hohe *membership* Werte aufweisen, als andere. Da die *Cluster* der Hintergrundfarben nicht so kompakt sind, wird ein zusätzlicher Referenzvektor für den Hintergrund verwendet, verglichen mit dem Ergebnis des *k-means* Algorithmus.

Die Segmentierung mittels *fuzzy k-means* ist wesentlich homogener. Mittels Schwellwertparameter Θ kann das Segmentierungsergebnis (Anzahl Farbsegmente) gesteuert werden.

5.4 Hand-Eye Koordination

Roboter übernehmen heutzutage viele mechanische Aufgaben, die einst von Menschenhand bewältigt wurden, z.B. Schweißroboter in der Automobilindustrie. Die Roboterbewegungen werden der Robotersteuerung manuell gelernt (*teach in*), indem der *Roboterendeffektor* in die gewünschte Position und Orientierung gebracht wird und die entsprechenden Positionsparameter des Roboters gespeichert werden. Im Laufe des Betriebes können sich die *kinematischen Parameter* (Länge der Armsegmente, Orientierung der Gelenkachsen) verändern. Grund dafür sind beispielsweise Temperaturschwankungen, Abnutzung von Werkzeugen, zu transportierende Lasten udg. Diese Einflüsse führen zu Positionierungenauigkeiten, die eine Neujustierung (Rekalibrierung) des Robotersystems zur Folge haben.

Martinetz [25] beschreibt ein adaptives Steuerungssystem, welches in der Lage ist, auf die Positionierungenauigkeiten zu reagieren und diese auszugleichen. Dabei handelt es sich um ein neurales Netzwerk, welches die *hand-eye* Koordination eines Robotersystems lernt. Unter *hand-eye* Koordination versteht man den Berechnungvorgang der entsprechenden Gelenkwinkel eines Roboters aufgrund der visuellen Repräsentation eines Objektes (z.B. über zwei Kameras). Dazu sind 2 Schritte notwendig (Melzer [29]). Zunächst muß aufgrund der Bilddaten die entsprechende Weltkoordinate des Objektes berechnet werden und anschließend mittels *inverser Kinematik** die Weltkoordinate in entsprechende Gelenkswinkel transformiert werden. Der erste Schritt verlangt, daß bei Verwendung eines Stereobildverfahrens beide beobachtenden Kameras kalibriert sind, welches in der Praxis schwierig und aufwendig ist. Im zweiten Schritt ist gefordert, daß die *inverse Kinematik* des Roboters vorliegt, deren Bestimmung sehr komplex oder gar unmöglich ist.

Das von Martinetz [25] beschriebene Robotersteuerungssystem ist schematisch in Abbildung 5.9 und ein exemplarischer Versuchsaufbau in Abbildung 5.10 (Melzer [29]) dargestellt. Der anzufahrende Punkt in 3D Weltkoordinaten ist mit

*Die entsprechende Umkehrung der Berechnung, von Gelenkswinkel in entsprechende Weltkoordinaten, wird *Vorwärtskinematik* genannt und ist im Gegensatz zur *inversen Kinematik* eindeutig.

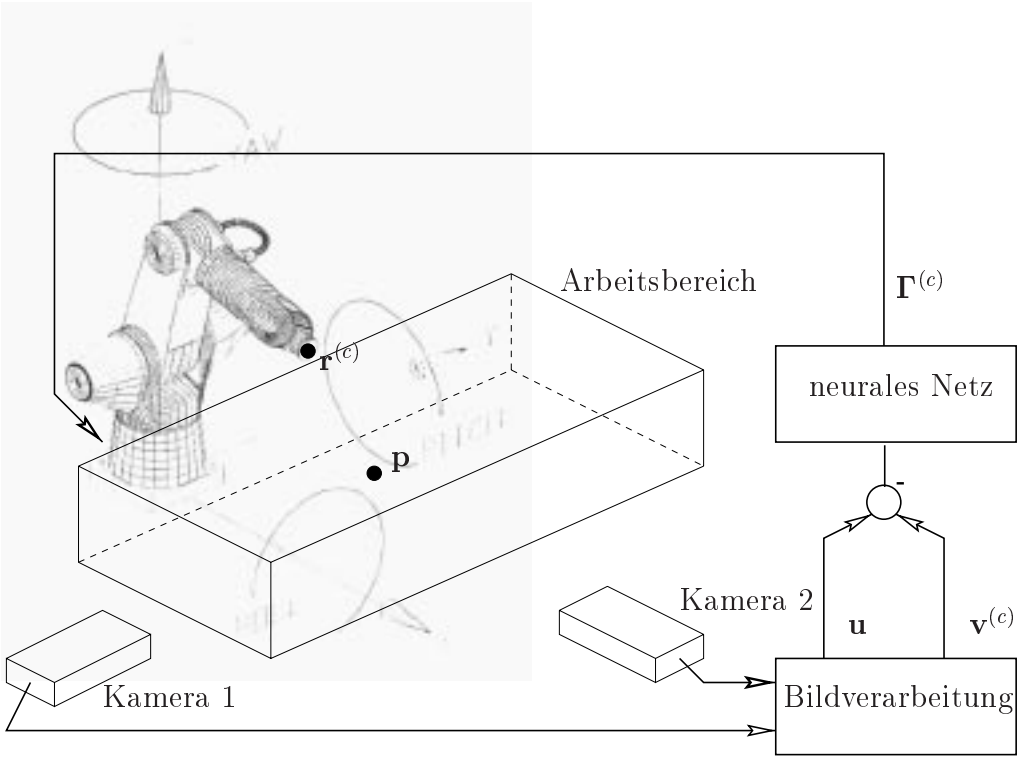


Abbildung 5.9: Schematische Darstellung des Positioniervorgangs. Der anzufahrende 3D Zielpunkt $\mathbf{p}$ wird mit Hilfe der Kameras über die Bildverarbeitungseinheit als 4D Vektor $\mathbf{u}$ interpretiert. Die aktuelle 3D Position des Roboters $\mathbf{r}^{(c)}$ besitzt den entsprechenden 4D Vektor $\mathbf{v}^{(c)}$. Die aktuellen Werte des Gelenkwinkels notiert $\mathbf{\Gamma}^{(c)}$, wobei mit c die aktuelle Korrekturbewegung angegeben wird.

$\mathbf{p} = (x, y, z)$ notiert. Die Aufgabe ist es nun, den *Roboterendeffektor*, dessen aktuelle Position in 3D Weltkoordinaten mit $\mathbf{r} \in \mathbb{R}^3$ symbolisiert ist, an die Zielposition $\mathbf{p} \in \mathbb{R}^3$ zu bewegen. Zwei festmontierte *CCD* Kameras beobachten den Arbeitsbereich des Roboters und liefern ihre Bilder an eine Bildverarbeitungseinheit. Die Bildverarbeitungseinheit ermittelt aus den beiden Kamerabildern die aktuelle Position des *Roboterendeffektor* $\mathbf{v} \in \mathbb{R}^4$ in 4D Kamerakoordinaten der entsprechenden Kamerabildebene.

$$\mathbf{v} = \Big((x_{Kamera1}, y_{Kamera1}), (x_{Kamera2}, y_{Kamera2}) \Big). \tag{5.1}$$

Melzer [29] brachte zur leichteren Identifikation des *Endeffektors* durch die Bildverarbeitungseinheit am *Endeffektor* eine Leuchtdiode an. Wie bei der Berechnung der Position $\mathbf{v}$ des *Roboterendeffektors* wird auch zum Zielpunkt $\mathbf{p}$ ein ent-

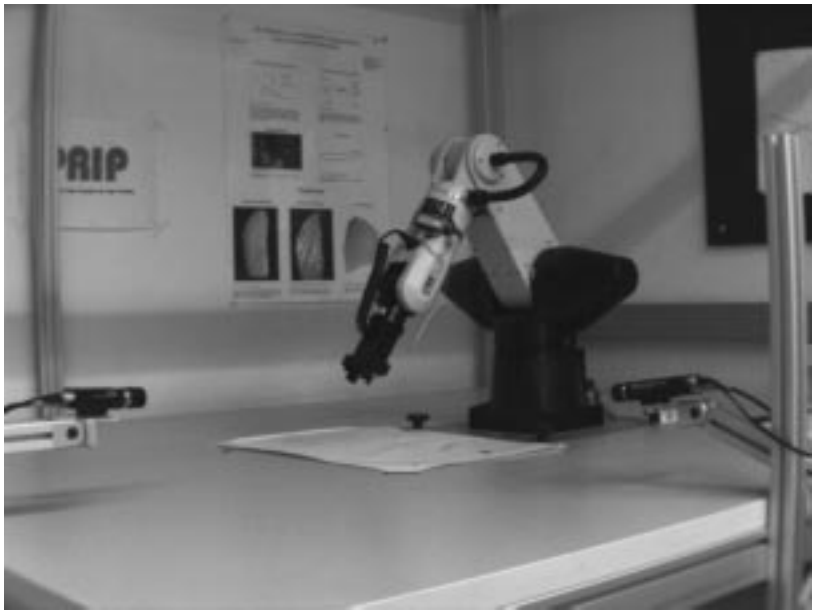


Abbildung 5.10: Der dem Simulator zugrundeliegende Industrieroboter *A465*. Zur Beobachtung der aktuellen Position des Roboterendeffektors werden 2 *CCD* Kameras angebracht.

sprechender 4-dimensionaler Vektor $\mathbf{u}$ in Kamerakoordinaten ermittelt. Aufgrund des Zielpunktes $\mathbf{u}$ berechnet das neurale Netz jene Werte der Gelenkwinkel $\mathbf{\Gamma}$, die entsprechend den *Roboterendeffektor* in den Zielpunkt $\mathbf{p}$ bewegen. Martinetz [25] verwendet eine Robotersteuerung mit 3 Freiheitsgraden (veränderbare Gelenkwinkel) $\mathbf{\Gamma} = (\Gamma_1, \Gamma_2, \Gamma_3)$, wodurch eine bijektive Abbildung zwischen Konfiguration der Gelenkwinkel und entsprechende Zielposition gegeben ist. Verwendet man mehr Freiheitsgrade, spricht man von *redundanten Freiheitsgraden*, wodurch ein Zielpunkt mit mehreren verschiedenen Winkelkonfigurationen erreicht werden kann. Um dieses *unterbestimmte* Problem in den Griff zu bekommen, werden in der Praxis Kostenfunktionen der verschiedenen Gelenkwinkelstellungen gewählt, deren Minimierung eine eindeutige Lösung bestimmen läßt.

Die erste Positionierung $\mathbf{r}^{(0)}$ (*open loop* Positionierung) wird in der Regel nicht mit dem Zielpunkt $\mathbf{p}$ übereinstimmen. Deswegen wird von der Bildverarbeitungseinheit die, der aktuellen *Roboterendeffektorposition* entsprechenden, 4D Kameraposition $\mathbf{v}^{(0)}$ bestimmt und der Positionierfehler $(\mathbf{u} - \mathbf{v}^{(0)})$ (in 4D Kamerakoordinaten) dem neuralen Netzwerk zugeführt. Dieses berechnet sich aus dieser Differenz eine korrigierte Gelenkwinkelkonstellation $\mathbf{\Gamma}^{(1)}$ und läßt den Roboter diese erste Korrekturbewegung ausführen. Dieser iterative Korrekturprozeß wird solange fortgeführt,

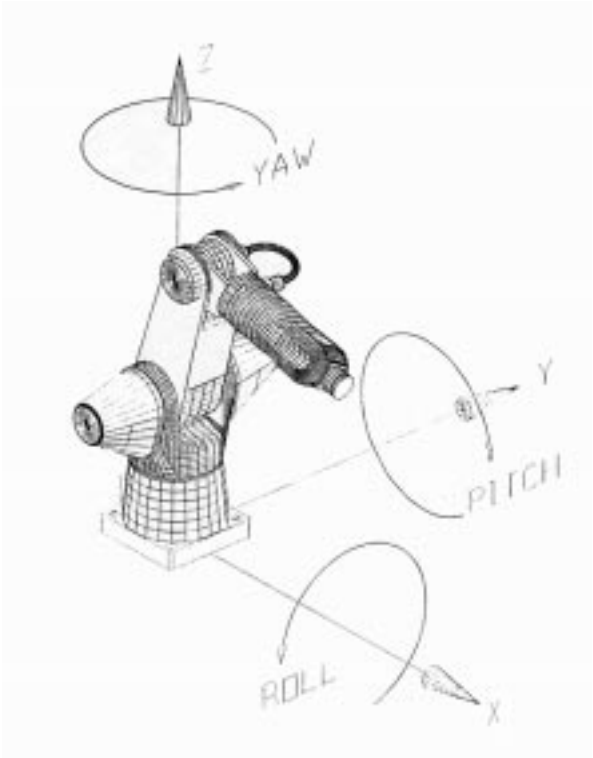


Abbildung 5.11: Das Weltkoordinatensystem des *A465* Roboters.

bis entweder eine Fehlerschranke unterschritten wird oder eine maximale Anzahl an Korrekturbewegungen durchgeführt wurde. Diese Korrekturschritte dienen nicht nur zur Verbesserung der Positioniergenauigkeit des *Roboterendeffektors*, sondern werden vom neuralen Netzwerk verwendet, um seine Abbildung $\mathbf{\Gamma} = f(\mathbf{u})$ zu trainieren. Während der einzelnen Trainingsschritte darf sich der Zielpunkt $\mathbf{p}$ oder dessen 4D Kameraäquivalent $\mathbf{u}$ nicht ändern, da der Lernprozeß des neuralen Netzes diese Information zu jeder Zeit benötigt (siehe später Abschnitt 5.4.1). Diese Art des Lernens wird auch *learning by doing* genannt.

Der Vorteil dieses Steuerungssystems ist, daß das neurale Netzwerk die gesuchte Gelenkwinkelkonstellation aufgrund der Positionsinformation, die von der Bildverarbeitungseinheit (in 4D Kamerakoordinaten) gewonnen wird, errechnet und so keine Kalibrierung der Kameras notwendig ist. Die Kalibrierung der Kameras und die des gesamten Robotersystems wird durch den Lernvorgang des neuralen Netzes implizit mitgelernt. Weiters haben die oben beschriebenen Positionierungenauigkeiten (aufgrund Last- und Verschleißerscheinungen) während des Betriebes keine Neujustierung des Robotersystems und der Kameras zur Folge, da durch den

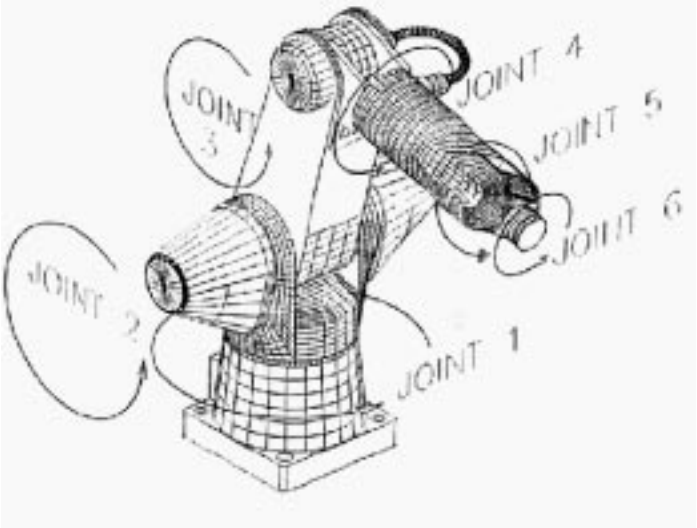


Abbildung 5.12: Die Achsen des *A465* Roboters. *Joint 1* Base Rotation, *Joint 2* Shoulder Rotation, *Joint 3* Elbow Rotation, *Joint 4* Euler Wrist (Wrist Rotate), *Joint 5* Wrist Bend, *Joint 6* Wrist Roll.

iterativen Korrekturprozeß dies einfach mitgelernt und korrigiert wird. Zu Beginn des Trainings des neuralen Netzes wurden die Netzparameter auf zufällige Werte gesetzt, wodurch eine unkontrollierte Positionierung des *Roboterendeffektors* durch das neurale Netz vorgenommen wird. Dies führt zu großen Positionierungenauigkeiten, die zwar im Laufe des Trainings korrigiert und gelernt werden würden, jedoch man in der Praxis mit folgenden Problemen konfrontiert ist (Melzer [29]):

- Die aktuelle Position des *Endeffektors* ist nicht mehr bestimmbar, da er sich außerhalb des Sichtbereiches der Kameras befindet.
- Der *Endeffektor* verläßt den vorgegebenen, sicheren Arbeitsbereich und kollidiert mit der Umgebung (umgebende Wände, Arbeitsplatte).
- Die aktuelle Position verlangt Gelenkwinkelstellungen, die Extrempositionen zur Folge haben und zu starker Belastung des Roboters führt (unnötige Abnutzung).

Deswegen sollte das zur Steuerung verwendete neurale Netzwerk vorher über einen Simulator vortrainiert und erst anschließend auf das reale System angewandt werden. Im folgenden wird nur noch von der Simulation des Robotersystems ausgegangen. Die Robotersimulation basiert auf dem *A465* Roboter (Abbildung 5.10),

dessen Weltkoordinatensystem in Abbildung 5.11 und Achsen in Abbildung 5.12 dargestellt sind. Dazu wird eine vorliegende *Vorwärtskinematik* des *A465* Roboters verwendet, um die Roboterbewegung zu simulieren. Weiters wird nur die Gelenkwinkelkonstellation $\mathbf{\Gamma} = (\Gamma_1 = \text{Joint1}, \Gamma_2 = \text{Joint2}, \Gamma_3 = \text{Joint3})$ verwendet und die restlichen Winkel konstant gehalten. Für eine genaue Beschreibung der Umsetzung der praktischen Robotersteuerung siehe Martinetz [25] und Melzer [29] und für die Beschreibung der Simulatorsoftware siehe Selb [36]. Die nächsten Kapitel beschreiben den Aufbau des neuralen Netzwerkes und die Verwendung des *MDL* Algorithmus zur Reduktion der Netzwerkkomplexität.

5.4.1 Extended Neural Gas Modell

Das von Martinetz [25] eingesetzte neurale Modell ist ein *Extended Neural Gas* (ENG) Modell, welches von Melzer [29] leicht modifiziert wurde. Das *ENG* Modell ist ein *Neural Gas* Modell (Abschnitt 2.5), welches zusätzlich in den Lage ist, eine Abbildung $f : \mathbb{R}^d \to \mathbb{R}^l$, $f(\mathbf{u}) = \Gamma$ zu lernen. Somit bekommt das *unsupervised learning* Vektorquantisierungsverfahren noch eine *supervised learning* Komponente hinzu. Das *ENG* Modell funktioniert auch für allgemeine vektorwertige Abbildungen, solange die Umkehrfunktion f^{-1} eindeutig* und bekannt ist.

Ähnlich dem Modell der *Local Linear Maps* (LLM; Abschnitt 2.8) wird jedem Netzwerkknoten $\mathbf{w}_i$, $1 \leq i \leq m$, eine Abbildung

$$\mathbf{y}_i = \mathbf{A}_i \mathbf{u} + \boldsymbol{\alpha}_i, \qquad \forall i \in \{1, \ldots, m\} \tag{5.2}$$

zugeordnet, wobei $\mathbf{A}_i$ eine $d \times l$ Matrix ($d = 4, l = 3$) ist und die Steigung der linearen Abbildung notiert. $\boldsymbol{\alpha}_i$ gibt den Anteil bei $\mathbf{u} = \mathbf{0}$ an. Im Unterschied zu den *LLM* besitzen alle Netzwerkknoten den selben Ursprung und alle Netzwerkknoten liefern einen Anteil zur gesamten Ausgabefunktion (*winner-takes-most*) entsprechend ihrer Nachbarschaftsreichweite $z_i = h_\lambda(k_i)$ (Gleichung 2.22).

$$z_i \quad = \quad h_\lambda(k_i) \tag{5.3}$$

*Dies ist in unserem Fall gegeben, da eindeutig einer Gelenkwinkelkonstellation $\mathbf{\Gamma}$ ein Punkt $\mathbf{r}$ in 3D Weltkoordinaten und daraus folglich ein eindeutiger Punkt $\mathbf{v}$ in 4D Kamerakoordinaten zugeordnet werden kann. Im Fall von *redundanten Freiheitsgraden* ist dies nicht gegeben.

1: Initialisiere die Menge der Referenzvektoren A mit zufälligen Werten.
2: Initialisiere $t := 0$.
3: **repeat**
4: Ziehe Datenpunkt $\mathbf{u} \in \mathbb{R}^4$. Berechnung des Datenpunktes $\mathbf{u}$ erfolgt gegebenfalls über die Bildverarbeitungseinheit.
5: Bestimme Rangfolge k_i aller Referenzvektoren $\mathbf{w}_i$: siehe Gleichung (2.21).
6: Bestimme Nachbarschaftswerte $z_i = h_\lambda(k_i)$ aller Referenzvektoren $\mathbf{w}_i$: siehe Gleichung (2.22).
7: Berechne Lernrate $\epsilon(t)$: siehe Gleichung (2.24).
8: Berechne Änderungen aller Referenzvektoren $\Delta\mathbf{w}_i$: siehe Gleichung (2.25).
9: Adaptiere alle Referenzvektor $\mathbf{w}_i$: $\mathbf{w}_i := \mathbf{w}_i + \Delta\mathbf{w}_i, \forall i \in \{1, \ldots, m\}$.
10: Normiere Gewichte z_i: siehe Gleichung (5.13).
11: Berechne Ausgabevektor für Gelenkwinkelansteuerung $\mathbf{\Gamma}^{(0)}$: siehe Gleichung (5.4).
12: Bewege *Roboterendeffektor* mit Gelenkwinkel $\mathbf{\Gamma}^{(0)}$ (*open loop* Bewegung) und ermittle entsprechende Kamerabildkoordinate $\mathbf{v}^{(0)}$.
13: Setze Zähler der Korrekturschritte $c = 1$.
14: **repeat**
15: Berechne Ausgabevektor für Gelenkwinkelkonstellation $\mathbf{\Gamma}^{(c)}$ für Korrekturbewegung: siehe Gleichung (5.6).
16: Bewege *Roboterendeffektor* mit Gelenkwinkel $\mathbf{\Gamma}^{(c)}$ (c-te Korrekturbewegung) und ermittle entsprechende Kamerabildkoordinate $\mathbf{v}^{(c)}$.
17: Ermittle Korrekturen der Netzwerkparameter $\mathbf{a}_{ij}$ und α_{ij}, $\forall i \in \{1, \ldots, m\}$, $\forall j \in \{1, \ldots, 3\}$: siehe Gleichungen (5.8) und (5.10).
18: Berechne Lernrate $\epsilon'(t)$: siehe Gleichung (2.24).
19: Adaptiere Netzwerkparameter $\mathbf{a}_{ij}$ und α_{ij}, $\forall i \in \{1, \ldots, m\}, \forall j \in \{1, \ldots, 3\}$: siehe Gleichungen (5.11) und (5.12).
20: $c := c + 1$
21: **until** $(c > c_{\max}) \vee (||\mathbf{u} - \mathbf{v}^{(c)}|| < \mathrm{err}_{\max})$
22: Reduktion des Netzwerkes: siehe Abschnitt 5.4.2
23: $t := t + 1$.
24: **until** $t = t_{\max}$.

Algorithmus 5.1: Extended Neural Gas Modell

$$\Gamma_j = \sum_{i=1}^{m} y_{ij} = \sum_{i=1}^{m} z_i(\mathbf{a}_{ij}\mathbf{u} + \alpha_{ij}), \qquad \forall j \in \{1, \ldots, l = 3\}. \tag{5.4}$$

Der Vektor $\mathbf{a}_{ij} \in \mathbb{R}^4$ ist der j-te Spaltenvektor der Matrix $\mathbf{A}_i$ und α_{ij} der j-te Wert des Vektors $\boldsymbol{\alpha}_i$ der Referenzvektors $\mathbf{w}_i$. Nach Berechnung der Gelenkwinkelkonstellation $\mathbf{\Gamma}^{(0)}$ in Gleichung (5.4) wird diese an den Roboter übergeben, welcher sich an die Position $\mathbf{r}^{(0)}$ (in 3D Weltkoordinaten) bewegt. Dies entspricht der *open loop* Bewegung. Über die Bildverarbeitungseinheit wird die aktuelle *Roboterendeffektorposition* $\mathbf{v}^{(0)}$ aus Sicht der Kameras ermittelt und der Positionierfehler $\mathbf{u} - \mathbf{v}^{(0)}$ (in 4D Kamerakoordinaten) an das Netzwerk übergeben. Abbildung 5.13 zeigt die Architektur des *ENG* Modells für eine Dimension der Gelenkwinkelkonstellation. Das Anlegen des Positionierfehlers, wie in Abbildung 5.13 gezeigt, entspricht der Präsentation des Eingabevektors $\mathbf{u} + (\mathbf{u} - \mathbf{v}^{(0)})$, der eine neue Gelenkwinkelkonstellation $\mathbf{\Gamma}^{(1)}$ zur Folge hat. Im c-ten Korrekturschritt wird dem neuralen Netzwerk der Vektor

$$\mathbf{u} + (\mathbf{u} - \mathbf{v}^{(0)}) + \ldots + (\mathbf{u} - \mathbf{v}^{(c-1)}) \tag{5.5}$$

präsentiert, das der Berechnung der Gelenkwinkelkonstellation von

$$\Gamma_j^{(c)} = \Gamma_j^{(c-1)} + \sum_{i=1}^{m} z_i(\mathbf{a}_{ij}(\mathbf{u} - \mathbf{v}^{(c-1)})), \qquad \forall j \in \{1, \ldots, 3\}. \tag{5.6}$$

gleichkommt. Bisher wurde lediglich beschrieben, wie die vom *ENG* zu lernende Funktion berechnet wird. Jedoch müssen die Parameter der zu lernenden Abbildung auch gelernt werden. Dies geschieht immer am Ende eines Korrekturschrittes ($c > 0$). Die Änderung des Vektors $\mathbf{a}_{ij}$ und des Skalars α_{ij} ergibt sich mittels Minimierung der Kostenfunktion $E_{\mathbf{a}}$ bzw. E_α.

$$E_{\mathbf{a}} = \frac{1}{2} \sum_{j=1}^{l=3} \Big[(\Gamma_j^{(c)} - \Gamma_j^{(c-1)}) - \sum_{i=1}^{m} z_i(\mathbf{a}_{ij}(\mathbf{v}^{(c)} - \mathbf{v}^{(c-1)})) \Big]^2, \tag{5.7}$$

$$\begin{aligned} \Delta\mathbf{a}_{ij} &= -\frac{\partial E_{\mathbf{a}}}{\partial \mathbf{a}_{ij}} \\ &= z_i \Big[(\Gamma_j^{(c)} - \Gamma_j^{(c-1)}) - \sum_{i=1}^{m} z_i(\mathbf{a}_{ij}(\mathbf{v}^{(c)} - \mathbf{v}^{(c-1)})) \Big] (\mathbf{v}^{(c)} - \mathbf{v}^{(c-1)}). \end{aligned} \tag{5.8}$$

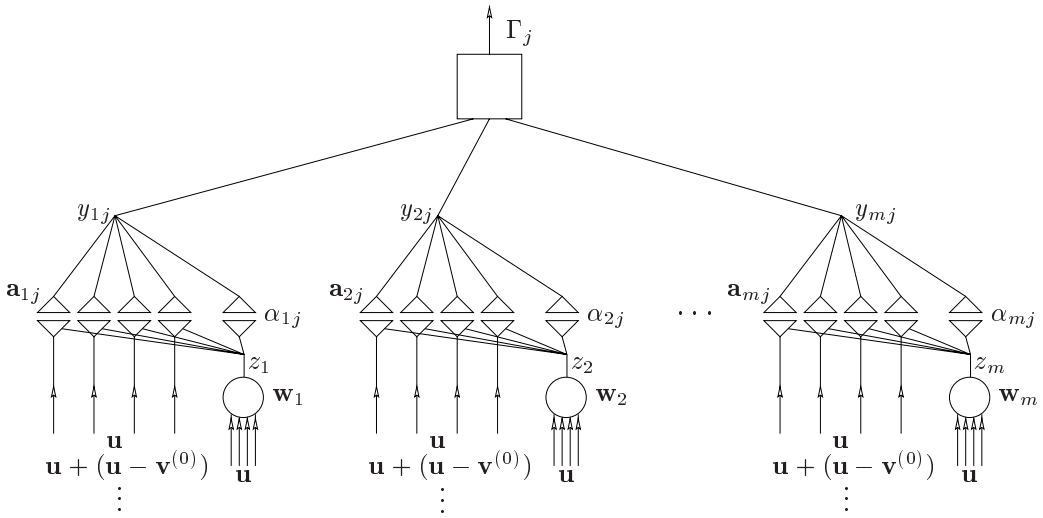


Abbildung 5.13: Architektur des *ENG* Modells für eine Dimension des Ausgabewertes (Gelenkwinkelkonstellation). Die *modulation units* (Dreiecke) berechnen aus dem Eingabesignal $\mathbf{u} + (\mathbf{u} - \mathbf{v}^{(0)}) + \ldots + (\mathbf{u} - \mathbf{v}^{(c-1)})$ die Aktivierung des jeweiligen Netzwerkknotens. Die *competing units* (Kreise) liefert die Gewichte z_i der Nachbarschaftsreichweite, mit denen die *modulation units* ihre Beiträge gewichten. Für den Ausgabewert Γ_j werden die Aktivierungen der einzelnen Netzwerkknoten aufsummiert (Quadrat).

$$
\begin{aligned}
E_\alpha &= \frac{1}{2}\sum_{j=1}^{l=3}[\Gamma_j^{(c)} - \Gamma_j^{(c-1)}]^2 \\
&= \frac{1}{2}\sum_{j=1}^{l=3}[\Gamma_j^{(c)} - \sum_{i=1}^{n}(z_i\alpha_{ij} + z_i\mathbf{a}_{ij}(\mathbf{u} + \mathbf{u} - \mathbf{v}^{(0)} + \ldots + \mathbf{u} - \mathbf{v}^{(c-2)}))]^2 \quad (5.9) \\
\Delta\alpha_{ij} &= -\frac{\partial E_\alpha}{\partial \alpha_{ij}} \\
&= z_i[\Gamma_j^{(c)} - \Gamma_j^{(c-1)}]. \quad (5.10)
\end{aligned}
$$

Nachdem das *ENG* einen *winner-takes-most* Lernmechanismus besitzt, lernen alle Netzwerkknoten entsprechend ihrer Nachbarschaftsrangfolge, wodurch die Adaptierung der Parameter $\mathbf{a}_{ij}$ und α_{ij} für alle Netzwerkknoten wie folgt vorgenommen wird.

$$
\begin{aligned}
\mathbf{a}_{ij} &:= \mathbf{a}_{ij} + \epsilon'(t)\left(\frac{1}{||\mathbf{v}^{(c)} - \mathbf{v}^{(c-1)}||^2}\right)\Delta\mathbf{a}_{ij}, \quad (5.11) \\
\alpha_{ij} &:= \alpha_{ij} + \epsilon'(t)\Delta\alpha_{ij}, \quad (5.12)
\end{aligned}
$$

wobei $\epsilon'(t)$ eine exponentiell abfallende Lernrate ist (siehe Gleichung 2.24) und t die

Anzahl Trainingsintervalle notiert. Der Gewichtungsfaktor ($\frac{1}{||\mathbf{v}^{(c)}-\mathbf{v}^{(c-1)}||^2}$) in Gleichung 5.11 wurde von Melzer [29] eingeführt und soll speziell in der Frühphase des Trainings ein unbeschränktes Anwachsen von $\mathbf{a}_{ij}$ verhindern. Weiters werden die Gewichte der Nachbarschaft z_i für die Berechnung der Ausgabewerte $\mathbf{\Gamma}$ und die Adaptierung der Parameter $\mathbf{a}_{ij}$ und α_{ij} normiert. Dadurch soll verhindert werden, daß diese Werte nicht nach oben skaliert werden, wodurch eine Verschlechtung der Positioniergenauigkeit nach anfänglicher Konvergenz auftreten kann (Melzer [29]).

$$z_i := \frac{z_i}{\sum_{j=1}^{m} z_j}, \qquad \forall i \in \{1, \ldots, m\}. \tag{5.13}$$

Algorithmus 5.1 stellt das *Extended Neural Gas* Modell übersichtlich dar, wobei das Originalmodell durch die oben erwähnten Änderungen modifiziert wurde. Zusätzlich wurde schon der Reduktionsschritt angeführt (grauhinterlegter Schriftzug), der im folgenden beschrieben wird.

5.4.2 MDL Algorithmus und ENG

Im Reduktionsschritt soll periodisch bei einer bestimmten Anzahl Lernschritte überprüft werden, ob und welche Netzwerkeinheiten entfernt werden können. Algorithmus 5.2 stellt den Ablauf der Reduktion dar. Wurde das *ENG* Modell reduziert, dauert es viele Lernschritte, bis sich wieder ein stabiler Netzwerkzustand einstellt, da die exponentiell abfallende Lernrate zum Zeitpunkt der Reduktion sehr niedrig ist. Jedoch stellen sich die Netzwerkparameter $\mathbf{a}_{ij}$ und α_{ij} sehr rasch auf stabile Werte ein, da die Nachbareinheiten ähnliche Werte dieser Parameter besitzen und deshalb lediglich geringfügige Änderungen erforderlich sind. Nach Entfernung von Netzwerkeinheiten wird es oft erforderlich sein, die Positionen der Referenzvektoren sehr stark zu adaptieren. Deswegen werden die Positionen der Referenzvektoren mit den $(t - t_{last}) \approx 500$ zuletzt gelernten Datenpunkten trainiert und dadurch eine Grobpositionierung nach Entfernung von Referenzvektoren vorgenommen. Dadurch soll gewährleistet werden, daß im nächsten Reduktionsschritt die Positionen der Referenzvektoren bereits einen stabilen Zustand erreicht haben. Bei der Bestimmung des Reduktionsintervalls η ist darauf zu achten, daß alle Netzwerkparameter ihren stabilen Zustand erreicht haben.

Aufgrund der *winner-takes-most* Philosophie des *ENG* Modells bei der Berechnung der Ausgabefunktion und der Adaptierung der Netzwerkparameter bietet

1: **if** $\eta|t$ **then**
2: Starte *MDL* Algorithmus.
3: **if** Wurde Netzwerk reduziert **then**
4: **for** $(t_{last} = t; t_{last} > T_{last}; t_{last}$- -) **do**
5: Ziehe Datenpunkt $\mathbf{u} = \mathbf{u}(t_{last}) \in \mathbb{R}^4$. Es wird der $(t - t_{last})$ zuletzt gelernte Datenpunkt gewählt.
6: Bestimme Rangfolge k_i aller Referenzvektoren $\mathbf{w}_i$: siehe Gleichung (2.21).
7: Bestimme Nachbarschaftswerte $z_i = h_\lambda(k_i)$ aller Referenzvektoren $\mathbf{w}_i$: siehe Gleichung (2.22).
8: Berechne Lernrate $\epsilon(t)$: siehe Gleichung (2.24).
9: Berechne Änderungen aller Referenzvektoren $\Delta\mathbf{w}_i$: siehe Gleichung (2.25).
10: Adaptiere alle Referenzvektor $\mathbf{w}_i$: $\mathbf{w}_i := \mathbf{w}_i + \Delta\mathbf{w}_i, \forall i \in \{1, \ldots, m\}$.
11: **end for**
12: **end if**
13: **end if**

Algorithmus 5.2: Reduktion des Extended Neural Gas Modell bzw. der Local Linear Maps mit Nachbarschaftskooperation

sich der *MDL* Algorithmus für *fuzzy Clustering* (Abschnitt 4.3) an. Anstelle der *membership* Werte in der Definition der *Domains* R_i (Gleichung (4.36)) treten nun die Gewichte der Nachbarschaftsreichweite z_i.

$$R_i = \{\mathbf{u}|z_i > \Theta\}, \qquad \forall i \in \{1, \ldots, m\}. \tag{5.14}$$

Aus der Definition des vom *ENG* Modell zu minimierendem Fehlermaß (Martinetz [25])

$$E = \frac{1}{2C(\lambda)} \sum_{i=1}^{m} \int_{\mathbf{v}} z_i (\mathbf{v} - \mathbf{w}_i)^2 p(\mathbf{v}) d\mathbf{v}, \tag{5.15}$$

beim dem $C(\lambda)^{-1} = \sum_{k=0}^{m-1} h_\lambda(k)$ nur ein Normalisierungsfaktor und $p(\mathbf{v})$ die Verteilung der Eingabesignale darstellt, ergibt sich für die Definition der Fehlerterme

$$\xi_i = \sum_{\mathbf{u} \in R_i} ||\mathbf{u} - \mathbf{v}^{(0)}||^2 z_i, \tag{5.16}$$

$$\xi_{ij} = \max\{\sum_{\mathbf{u}\in R_i\cap R_j} ||\mathbf{u}-\mathbf{v}^{(0)}||^2 z_i, \sum_{\mathbf{u}\in R_i\cap R_j} ||\mathbf{u}-\mathbf{v}^{(0)}||^2 z_j\}. \quad (5.17)$$

Der Term $\mathbf{v}^{(0)}$ soll andeuten, daß zur Berechnung der Fehlerterme lediglich der *open loop* Positionierfehler herangezogen wird. Weiters wird bei jenem Reduktionsschritt die gesamte Trainingsmenge herangezogen und mit dem aktuellen *ENG* Netzwerk der *open loop* Positionierfehler bestimmt. Dieser Positionierfehler wird dann entsprechend dem Gewicht z_i auf die einzelnen Referenzvektoren aufgeteilt. Würde man auch noch die Korrekturschritte berücksichtigen, dann müßte man noch jeweils das Netzwerk neu trainieren und dies wäre ein enormer Aufwand. Trotzdem bleibt der Aufwand enorm, wenn man bedenkt, daß man das zu jedem $\mathbf{r}$ entsprechende $\mathbf{v}$ über die Vorwärtskinematik, die Neuberechnung der z_i Werte und die Simulation der Kameras bestimmen muß (Selb [36]).

Aus Gleichung (5.14) sieht man, daß bei einem fixen Schwellwert Θ die *Domains* im Laufe des Trainings immer kleiner werden, da die z_i Werte im Laufe des Trainings abnehmen. Die Wahl eines guten Schwellwertes Θ ist demnach sehr schwierig. Auch haben experimentelle Versuche gezeigt, daß es bei der Definition der *Domains* aus Gleichung (5.14) nicht ausreichend ist, nur die paarweisen Überlappbereiche zu berücksichtigen, da sie im Verhältnis zu den *Domains* R_i ein sehr hohes Gewicht in der Optimierungsphase haben. Um aber dennoch die Gestalt der Optimierungsmethode aus Gleichung (4.39) beizubehalten und die Überlappbereiche höherer Ordnung vernachlässigen zu können, ändert man die Definition der *Domains* R_i in

$$R_i = \{\mathbf{u}|k_i > \Upsilon\}, \quad (5.18)$$

wobei k_i der Rang des Datenpunktes $\mathbf{u}$ zum Referenzvektor $\mathbf{w}_i$ ist. Um die Berücksichtigung der paarweisen Überlappbereiche weiterhin zu verwenden, setzt man $\Upsilon = 1$, da $k_i = 0$ beim nächsten und $k_i = 1$ beim zweitnächsten Referenzvektor gilt. Weitere experimentelle Versuche zeigen jedoch den Schwachpunkt des *MDL* Modells für *fuzzy Clustering* auf. Die Definition der Beschreibungslänge (Gleichung (4.32)) geht davon aus, daß es eine Hypothese $\hat{\mathbf{h}}$ gibt, die die Beschreibungslänge minimiert und nach der Entfernung der überflüssigen Referenzvektoren tatsächlich eine Verringerung der Beschreibungslänge aufgetreten ist. Wird ein vermeintlich überflüssiger Referenzvektor entfernt, tritt der Fall auf, daß sich die Werte der Fehlerterme ξ_i und ξ_{ij} erhöhen, da eine zusätzlich Positionierungenauigkeit auftritt. Die Definition der

Beschreibungslänge (Gleichung (4.32)) sieht jedoch keinen Mechanismus vor, der es erlaubt, den durch die Reduktion entstehenden, zusätzlichen Fehler abzuschätzen.

Im Beispiel der *fuzzy* Vektorquantisierung (Abschnitt 5.1.2) und der *RGB* Farbsegmentierung (Abschnitt 5.3) hatten von einem Referenzvektor weit entfernte Datenpunkte auch einen höheren Fehlerwert. Jedoch bei Verwendung des *ENG* Modells wird eine Abbildung gelernt, die diese Eigenschaft nicht besitzt, da weitentfernte Datenpunkte ähnliche Fehlerwerte zur Folge haben als nahe. Dadurch werden die Fehlerterme der weitentfernten Datenpunkte durch die Nachbarschaftsreichweite z_i abgeschwächt, wodurch sie geringeren Einfluß haben. Wird ein vermeintlich überflüssiger Referenzvektor entfernt, wird der Fall auftreten, daß weitentfernt Datenpunkte schlagartig nur mehr mit einem sehr hohem Positionierfehler angefahren werden können.

Deswegen kann die Berechnung des *MDL* Kriteriums nicht mehr für die einzelnen *Domains* getrennt vorgenommen werden. Man muß nun bei der Berechnung des *MDL* Kriteriums für einen Referenzvektor alle anderen Referenzvektoren mitberücksichtigen. Definiert man nun Gleichung (4.32) für den Fall, daß die Netzwerkeinheit $\mathbf{w}_k$ ($h_k = 0$) entfernt wird, so ergibt sich

$$L_{(-\mathbf{w}_k)}(S(\mathbf{h})) = d\ K_1(n - n(\mathbf{h})) + K_2\xi_{(-\mathbf{w}_k)}(\mathbf{h}) + d\ K_3 N_{(-\mathbf{w}_k)}(\mathbf{h}), \tag{5.19}$$

wobei der Index $(-\mathbf{w}_k)$ andeuten soll, daß der Netzwerkknoten von Referenzvektor $\mathbf{w}_k$ entfernt wurde. Die Änderung der Beschreibungslänge ergibt sich demnach aus

$$\begin{aligned} \Delta L_{(-\mathbf{w}_k)}(S(\mathbf{h})) &= L_{(-\mathbf{w}_k)}(S(\mathbf{h})) - L(S(\mathbf{h})) && (5.20)\\ &= K_2(\xi_{(-\mathbf{w}_k)}(\mathbf{h}) - \xi(\mathbf{h})) + d\ K_3(N_{(-\mathbf{w}_k)}(\mathbf{h}) - N(\mathbf{h})). && (5.21) \end{aligned}$$

Ist die Änderung der Beschreibungslänge negativ ($\Delta L_{(-\mathbf{w}_k)}(S(\mathbf{h})) < 0$), kann der Referenzvektor $\mathbf{w}_k$ entfernt werden, da die gesamte Beschreibungslänge verringert wird. Das *MDL* Kriterium für die Entfernung einer Referenzvektors lautet demnach

$$d\ K_3 N_{RV} > K_2(\xi_{(-\mathbf{w}_k)}(\mathbf{h}) - \xi(\mathbf{h})), \tag{5.22}$$

wobei N_{RV} die Kosten eines Referenzvektor angibt. Gleichung (5.22) besagt, daß dann ein Knoten überflüssig ist, wenn der erwartete Fehler weniger kostet, als die Kosten des entfernten Referenzvektors. Wie bereits in Abschnitt 4.2.4 erwähnt, können mehrere Knoten parallel entfernt werden, wobei alle Nachbarn eines entfernten Knotens nicht entfernt werden dürfen. Die Berechnung der Fehlerterme ist die Summe aller Positionierfehler über alle Trainingspunkte.

$$\xi(\mathbf{h}) = \sum_{\mathbf{u}} ||\mathbf{u} - \mathbf{v}^{(0)}||^2, \tag{5.23}$$

$$\xi_{(-\mathbf{w}_k)}(\mathbf{h}) = \sum_{\mathbf{u}} ||\mathbf{u} - \mathbf{v}^{(0)}_{(-\mathbf{w}_k)}||^2, \tag{5.24}$$

wobei bei der Berechnung von $\mathbf{v}^{(0)}_{(-\mathbf{w}_k)}$ der Referenzvektor $\mathbf{w}_k$ vorübergehend entfernt wird und eine Neuberechnung der Rangfolge und daraus resultierend der Nachbarschaftsreichweite durchgeführt wird. Nach der eigentlichen Entfernung eines Referenzvektors werden alle Positionen der Referenzvektoren neu trainiert (siehe Algorithmus 5.2). Dadurch kann sich auch die Rangfolge der Referenzvektoren einiger Datenpunkte ändern, die sich bei der Berechnung des *MDL* Kriteriums nicht ergeben hatte. Dies wirkt sich auch auf den jeweilige Aktivierungswert z_i aus, der dadurch den Netzoutputanteil eines anderen Referenzvektors stärker gewichtet als bei der Berechnung des *MDL* Kriteriums. Dies könnte den tatsächlichen Fehler $\xi_{(-\mathbf{w}_k)}$ größer werden lassen, als $\xi_{(-\mathbf{w}_k)}(\mathbf{h})$ aus Gleichung (5.24), wodurch das *MDL* Kriterium (Gleichung (5.22)) nicht mehr stimmen kann. Abhilfe dafür wäre, daß in jedem Reduktionsschritt nur ein Referenzvektor (nicht mehr parallel) entfernt wird und zusätzlich vor der Berechnung des *MDL* Kriteriums die Referenzvektoren (ohne $\mathbf{w}_k$) neu trainiert werden.

Dieser unterschätzte Fehler $\xi_{(-\mathbf{w}_k)}(\mathbf{h})$ wirkt sich jedoch nur gering aus, da einerseits die stärker gewichtete Netzwerkeinheit einen ähnlichen Netzwerkoutput liefert als die alte und andererseits dies nur bei Netzwerkeinheiten mit kleinen Werten der Nachbarschaftsfunktion z_i auftritt und so den Einfluß abschwächt. Weiters werden wir in Abschnitt 5.4.5.1 sehen, daß dieser *MDL* Algorithmus gegenüber dem Parameter K_2 sehr robust ist und dadurch eine kleine (positive) Änderung des Terms $(\xi_{(-\mathbf{w}_k)}(\mathbf{h}) - \xi(\mathbf{h}))$ in Gleichung (5.22) keinen Einfluß auf das *MDL* Kriterium hat.

Diese beiden Fehlerterme $\xi(\mathbf{h})$ und $\xi_{(-\mathbf{w}_k)}(\mathbf{h})$ (Gleichung (5.23) und (5.24)) gehen davon aus, daß die gesamte Trainingsmenge zur Berechnung herangezogen wird. Würde man die vollständige Datenmenge verwenden, wäre die Berechnung

der Reduktion mit Fortdauer des Trainings immer aufwendiger. Man kann jedoch auch nur eine Teilmenge zur Berechnung des *MDL* Kriteriums heranziehen, da man annimmt, daß die Datenpunkte der Trainingsmenge identisch und unabhängig verteilt sind. Auch wächst, bei reduzierter Datenmenge, der Wert für den Fehler nicht mehr mit der Größe der Trainingsmenge und das *MDL* Kriterium bleibt von der Größe der Trainingsmenge unabhängig. Dieses Kriterium ist aber auch nur für die Simulation der Robotersteuerung interessant, da man im realen System nie eine Trainingsmenge mit einer Größenordnung von 1000 Datenpunkten durchsimulieren kann (Selb [36]). Simulationsbeispiele und die Herleitung der Parameter des *MDL* Algorithmus wird in Abschnitt 5.4.5 vorgenommen.

5.4.3 Local Linear Maps mit Nachbarschaftskooperation

Das *ENG* Modell ist in der Lage eine vektorwertige Abbildung zu lernen, wobei jeder Netzwerkknoten zum Ergebnis beiträgt. Der Beitrag eines Netzwerkknotens ist von der Gewichtung der Nachbarschaftsreichweite z_i abhängig. Aufgrund der *winner-takes-most* Philosophie bei der Berechnung der Ausgabewerte eignet sich der *MDL* Algorithmus für *fuzzy Clustering* sehr gut. Das oben beschriebene *ENG* Modell wird so abgewandelt, daß es zur Berechnung der Ausgabewerte eine *winner-takes-all* Philosophie verwendet.

$$\Gamma_j = y_{sj} = (\mathbf{a}_{sj}\mathbf{u} + \alpha_{sj}), \qquad \forall j \in \{1, \ldots, l = 3\}, \tag{5.25}$$

wobei $\mathbf{w}_s$ der dem Datenpunkt $\mathbf{u}$ nächst gelegene Referenzvektor ist und $\mathbf{a}_{sj}$ bzw. α_{sj} seine Parameter, welche zur Berechnung der Gelenkwinkel $\mathbf{\Gamma}$ benötigt werden, sind. Entsprechendes ergibt sich auch bei der Berechnung der Korrekturbewegung

$$\Gamma_j^{(c)} = \Gamma_j^{(c-1)} + (\mathbf{a}_{sj}(\mathbf{u} - \mathbf{v}^{(c-1)})), \qquad \forall j \in \{1, \ldots, 3\}. \tag{5.26}$$

Zur Berechnung der Gelenkwinkelkonstellation $\mathbf{\Gamma}$ wird also immer nur der nächste Referenzvektor $\mathbf{w}_s$ herangezogen. Martinetz [25] zeigt, daß das Lernverhalten beim Lernen mit Nachbarschaftskooperation eine raschere Konvergenz zur Folge hat. Ohne Nachbarschaftskooperation ist es meistens nicht möglich, ein gutes Minimum des Positionierfehlers zu finden, da einige Netzwerkknoten in Nebenminima hängen bleiben. Deshalb lernen alle Netzwerkknoten mit, obwohl nur der Netzwerkknoten $\mathbf{w}_s$ zur Berechnung der Ausgabefunktion verwendet wurde.

1: Initialisiere die Menge der Referenzvektoren A mit zufälligen Werten.
2: Initialisiere $t := 0$.
3: **repeat**
4: Ziehe Datenpunkt $\mathbf{u} \in \mathbb{R}^4$. Berechnung des Datenpunktes $\mathbf{u}$ erfolgt gegebenfalls über die Bildverarbeitungseinheit.
5: Bestimme Rangfolge k_i aller Referenzvektoren $\mathbf{w}_i$: siehe Gleichung (2.21).
6: Bestimme Nachbarschaftswerte $z_i = h_\lambda(k_i)$ aller Referenzvektoren $\mathbf{w}_i$: siehe Gleichung (2.22).
7: Berechne Lernrate $\epsilon(t)$: siehe Gleichung (2.24).
8: Berechne Änderungen aller Referenzvektoren $\Delta\mathbf{w}_i$: siehe Gleichung (2.25).
9: Adaptiere alle Referenzvektor $\mathbf{w}_i$: $\mathbf{w}_i := \mathbf{w}_i + \Delta\mathbf{w}_i, \forall i \in \{1, \ldots, m\}$.
10: Normiere Gewichte z_i: siehe Gleichung (5.13).
11: Berechne Ausgabevektor für Gelenkwinkelansteuerung $\mathbf{\Gamma}^{(0)}$: siehe Gleichung (5.25).
12: Bewege *Roboterendeffektor* mit Gelenkwinkel $\mathbf{\Gamma}^{(0)}$ (*open loop* Bewegung) und ermittle entsprechende Kamerabildkoordinate $\mathbf{v}^{(0)}$.
13: Setze Zähler der Korrekturschritte $c = 1$.
14: **repeat**
15: Berechne Ausgabevektor für Gelenkwinkelkonstellation $\mathbf{\Gamma}^{(c)}$ für Korrekturbewegung: siehe Gleichung (5.26).
16: Bewege *Roboterendeffektor* mit Gelenkwinkel $\mathbf{\Gamma}^{(c)}$ (c-te Korrekturbewegung) und ermittle entsprechende Kamerabildkoordinate $\mathbf{v}^{(c)}$.
17: Ermittle Korrekturen der Netzwerkparameter $\mathbf{a}_{ij}$ und α_{ij}, $\forall i \in \{1, \ldots, m\}$, $\forall j \in \{1, \ldots, 3\}$: siehe Gleichungen (5.27) und (5.28).
18: Berechne Lernrate $\epsilon'(t)$: siehe Gleichung (2.24).
19: Adaptiere Netzwerkparameter $\mathbf{a}_{ij}$ und α_{ij}, $\forall i \in \{1, \ldots, m\}$, $\forall j \in \{1, \ldots, 3\}$: siehe Gleichungen (5.11) und (5.12).
20: $c := c + 1$
21: **until** $(c > c_{\max}) \vee (||\mathbf{u} - \mathbf{v}^{(c)}|| < \mathrm{err}_{\max})$
22: Reduktion des Netzwerkes: siehe Abschnitt 5.4.4
23: $t := t + 1$.
24: **until** $t = t_{\max}$.

Algorithmus 5.3: Local Linear Maps mit Nachbarschaftskooperation

$$\Delta \mathbf{a}_{ij} = z_i \Big[(\Gamma_j^{(c)} - \Gamma_j^{(c-1)}) - (\mathbf{a}_{sj}(\mathbf{v}^{(c)} - \mathbf{v}^{(c-1)})) \Big] (\mathbf{v}^{(c)} - \mathbf{v}^{(c-1)}). \quad (5.27)$$

$$\Delta \alpha_{ij} = z_i [\Gamma_j^{(c)} - \Gamma_j^{(c-1)}]. \quad (5.28)$$

Die beiden obigen Zusammenhänge zur Berechnung der Änderung der Netzwerkparameter unterscheiden sich von jenen des *ENG* Modells (Gleichung (5.8) und (5.10)) nur in der Definition des rückgekoppelten Änderungsterms. Beim *LLM* mit Nachbarschaftskooperation errechnet sich der Netzwerkoutput nur über den nächsten Referenzvektor $\mathbf{w}_s$. Das Gewicht der Nachbarschaftsreichweite z_i verteilt die Änderungen auf die einzelnen Referenzvektoren auf. Somit gibt der in Algorithmus 5.3 dargestellte Ablauf den des *LLM* mit Nachbarschaftskooperation an.

5.4.4 MDL Algorithmus und LLM mit Nachbarschaftkooperation

Die Reduktion der *LLM* mit Nachbarschaftskooperation läuft identisch mit der des *Extended Neural Gas* (ENG) Modells ab (Algorithmus 5.2). Jedoch wird bei den *LLM* mit Nachbarschaftskooperation der *MDL* Algorithmus für *crisp Clustering* (Abschnitt 4.2) verwendet, da die Berechnung der Ausgabefunktion $\mathbf{\Gamma}$ nur vom nächsten Referenzvektor $\mathbf{w}_s$ vorgenommen wird. Wird nun der Referenzvektor $\mathbf{w}_j$ entfernt, müssen alle Datenpunkte $\mathbf{u} \in S_j$ durch seine direkten Nachbarn übernommen werden. Somit ergibt sich die Änderung der Kodierungskosten der Indizes wie in Gleichung (4.23) (variable Kodierungslänge) dargestellt. Gehen wir von der Gaußverteilung als Verteilungsfunktion der Residuen aus, so ergibt sich äquivalent zu Gleichung (4.30)

$$\Delta L(\epsilon(\mathbf{u}, A \backslash \{\mathbf{w}_j\}) = \sum_{\mathbf{u} \in S_j} \sum_{i=1}^{d=4} \frac{(\mathbf{u}_i - \mathbf{v}_{ki}^{(0)})^2 - (\mathbf{u}_i - \mathbf{v}_{ji}^{(0)})^2}{2 \ln(2) \sigma^2}, \quad (5.29)$$

wobei $\mathbf{u}$ der aktuelle Zielpunkt ist. Die 4D Kameraposition $\mathbf{v}_j^{(0)}$ (*open loop* Bewegung) ist jene Position, welche von der Menge A der Referenzvektoren bestimmt wurde, bei der $\mathbf{w}_j$ der nächste Referenzvektor zu $\mathbf{u}$ ist. Wird der Referenzvektor $\mathbf{w}_j$ entfernt, muß die Berechnung des Netzwerkoutputs für $\mathbf{u}$ vom zweitnächsten Referenzvektor $\mathbf{w}_k$ übernommen werden, der die 4D Kameraposition $\mathbf{v}_k^{(0)}$ (*open loop*

Bewegung) zur Folge hat. Eine Berücksichtigung der *outlier* wird nicht vorgenommen. Dies macht in diesem Zusammenhang auch keinen Sinn, da die Datenpunkte **u** vom Roboter angesteuert werden sollen und vorgegeben werden.

Zwar wird in Gleichung (5.29) berücksichtigt, daß bei Entfernung eines Referenzvektors die Berechnung des Netzwerkoutputs durch seine direkten Nachbarn übernommen werden muß, jedoch nicht den Fall, daß alle Grenzen der *Voronoi Regionen* durch die Neupositionierung der Referenzvektoren verändert werden. Dadurch können Datenpunkte, die sich in der Nähe der Grenzen der *Voronoi Regionen* aufhalten, in die *Voronoi Region* der Nachbareinheit hinüberwandern. Viele Datenpunkte können meistens von ihrer Nachbareinheit nicht so gut beschrieben werden und führen deshalb zu einer Erhöhung des Fehlerterms und somit auch zu einer Erhöhung der Beschreibungslänge. Das *ENG* Modell hatte den Vorteil, daß dieser Effekt über die Nachbarschaftsfunktion abgeschwächt wird. Für das *LLM* Modell mit Nachbarschaftsfunktion gibt es nur die Möglichkeit, die Berechnung der Änderung der Beschreibungslänge wie aus Gleichung (4.15) vorzunehmen.

Für die Berechnung der Kodierungskosten für den Fall, daß ein Referenzvektor entfernt wird, muß beachtet werden,

- daß die Positionen der Referenzvektoren des reduzierten Netzwerkes neutrainiert werden müssen, um die veränderten *Voronoi Regionen* zu erhalten. Dies ist notwendig, um die Datenpunkte ihren neuen Referenzvektoren zu zuordnen. Aufgrund der *winner-takes-most* Lernstrategie und der *winner-takes-all* Strategie zur Berechnung des Netzwerkoutputs (Einteilung in *Voronoi Regionen*) kann der Fall eintreten, daß sich keine Datenpunkte in einer *Voronoi Region* befinden.

- daß für die Berechnung des Netzwerkoutputs die Positionen der Referenzvektoren des nicht reduzierten (ohne dem reduzierten Netzwerkknoten) Netzwerkes zu verwenden sind, damit sich die Berechnung des Netzwerkoutputs durch die in den originalen *Voronoi Regionen* verbleibenden Datenpunkte nicht verschlechtert. Dadurch tritt eine Erhöhung des Kostenterms ein, die verhindern würde, daß ein Referenzvektor entfernt wird.

- daß wegen der Neuordnung der *Voronoi Regionen* keine parallele Reduktion mehr möglich ist.

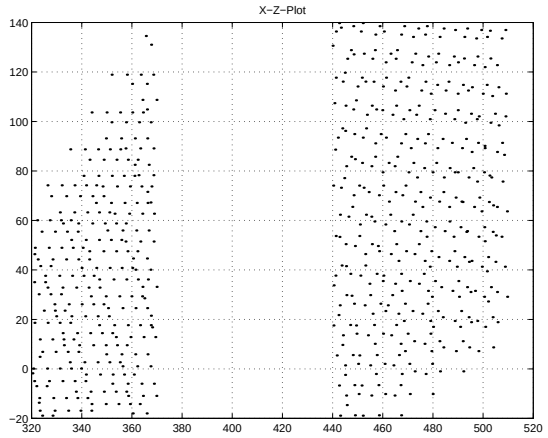
- daß die Berechnung des Fehlerterms sehr sensitiv gegenüber dem Diskretisierungsparameter η ist (Abschnitt 4.2.6).

Die Berechnung der Änderung der Kodierungskosten ist für das *LLM* Modell mit Nachbarschaftskooperation enorm rechenaufwendig, da neben der Neuberechnung der Positionen der Referenzvektoren auch eine Neuberechnung der Kodierungskosten der reduzierten Netze vorgenommen werden muß. Man kann aber den Abstand zwischen zwei Reduktionsschritten reduzieren, da einerseits die zu lernende Abbildung aufgrund der geringen Änderung in den Referenzvektoren relativ rasch adaptiert werden kann und andererseits die Positionen nach der Grobpositionierung nach der Reduktion auch schnell trainiert werden können. Jedoch kann die schöne Eigenschaft des *MDL* Algorithmus für *crisp Clustering*, bei der die Änderung der Beschreibungslänge lokal für einen Referenzvektor und seine direkten Nachbarn berechnet wird, nicht angewandt werden, wodurch es nicht sinnvoll ist, hier umfangreiche Experimente durchzuführen.

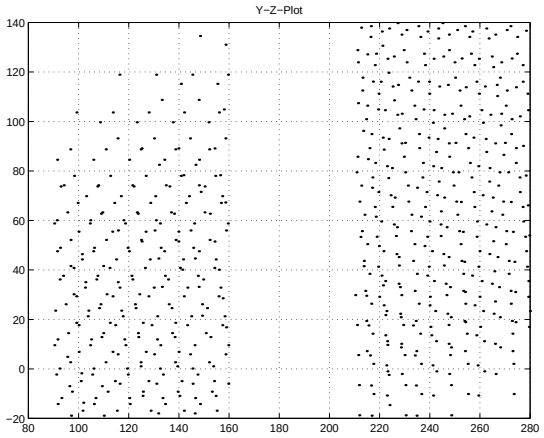
5.4.5 Experimente

Um die Funktionsweise des *ENG* und der *LLM* mit Nachbarschaftkooperation zum Erlernen der *hand-eye* Koordination zu demonstrieren, wurden 649 Datenpunkte in Form von 2 Quadern generiert ($x \in [320, 370]$, $y \in [90, 160]$ und $z \in [-20, 140]$ bzw. $x \in [440, 510]$, $y \in [210, 280]$ und $z \in [-20, 140]$ in mm und 3D Weltkoordinaten; Abbildung 5.14). Da sich die Datenpunkte am Rand des Roboterarbeitsbereiches befanden, bilden die Datenpunkte einmal keinen Quader. Die beiden Kameras befanden sich an den Punkten $p_{Kamera1} = (420, -320, 60)$ und $p_{Kamera2} = (902, 50, 60)$ mit Blickrichtung auf den Punkt $p_{BR} = (420, 180, 60)$. Das bedeutet, daß die Kamera 1 normal auf die xz-Ebene (von links) und die Kamera 2 um den Drehpunkt p_{BR} um 75° nach rechts (relativ zu Kamera 2) gedreht angeordnet ist. Abbildung 5.11 zeigt das zugrundeliegende Koordinatensystem. Die beiden Kameras hatten eine fokale Länge von 16 mm und die Matrix des Framebuffers eine Größe von 8.8×6.6 $mm \leftrightarrow 764 \times 576$ Pixel. Die aus der Bildverarbeitungseinheit gewonnenen Datenpunkte $\mathbf{u} = ((x_{Kamera1}, y_{Kamera1}), (x_{Kamera2}, y_{Kamera2}))$ wurden für jede Kamera getrennt normalisiert, daß sich die größte Seite im Intervall $[-0.5, \ldots, 0.5]$ befand.

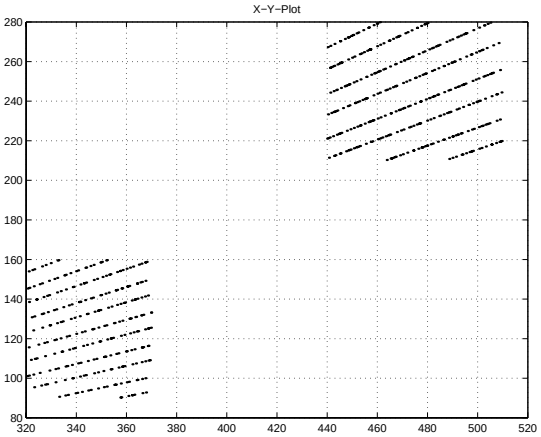
Die Anzahl Netzwerkknoten setzen wir beliebig auf 20 fest. Das Training für einen Datenpunkt wurde entweder nach 5 Korrekturschritten oder dem Unterschrei-



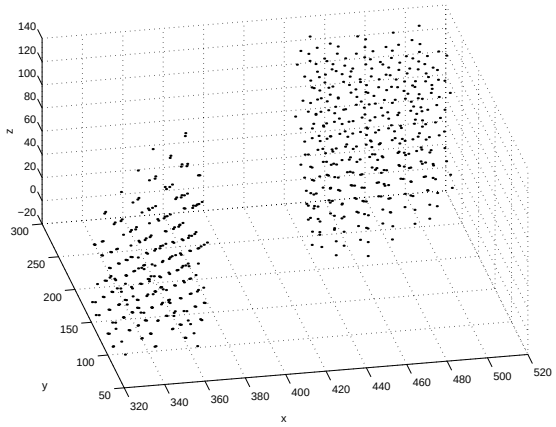
(a) von links



(b) von vorn



(c) von oben



(d) 3D Ansicht

Abbildung 5.14: Datenpunkte in 3D-Weltkoordinaten (in mm). Abbildungen (a), (b) und (c) zeigen den orthogonalen Blick auf die entsprechende Ebene (aus Sicht des Roboters; Abbildung 5.11). Abbildung (d) zeigt die dreidimensionale Ansicht der Datenpunkte.

bit	t
8	$3.9\ 10^{-3}$
16	$1.5\ 10^{-5}$
32	$2.3\ 10^{-10}$

Tabelle 5.2: Diskretisierungsfehler t für verschiedene Kodierungen.

ten des aktuellen Positionierfehlers von $\text{err}_{\max} = 0.1\ mm$ (in 3D Weltkoordinaten) abgebrochen und das Training mit dem nächsten Datenpunkt fortgefahren. Die Anzahl Trainingsschritte wird auf 15000 festgelegt, wobei die Datenpunkte zufällig aus der Trainingsmenge gezogen werden. Für die Lernraten $\epsilon(t)$, $\epsilon'(t)$ (Gleichung (2.24)) und $\lambda(t)$ (Gleichung (2.23)) geben wir folgende Parameter an, die während der Experimente gute Konvergenzeigenschaften der Positionierfehler zur Folge hatten.

$$\epsilon_i = 0.3, \qquad \epsilon_f = 0.02, \qquad t_{\max} = 5000 \tag{5.30}$$

$$\epsilon'_i = 0.9, \qquad \epsilon'_f = 0.1, \qquad t_{\max} = 5000 \tag{5.31}$$

$$\lambda_i = 7.0, \qquad \lambda_f = 1.0, \qquad t_{\max} = 1000 \tag{5.32}$$

5.4.5.1 Extended Neural Gas

Unter der Annahme, daß der Diskretisierungsfehler der Kameras 3 Pixel groß ist, ergibt sich unter Berücksichtigung der Normalisierung der Datenpunkte von $f_{norm} = \frac{1}{8.8}$ für σ:

$$\sigma = 3 \cdot \epsilon_{Pixel} \cdot f_{norm} = 3 \cdot \frac{8.8}{764} \cdot \frac{1}{8.8} \approx 0.0039 \tag{5.33}$$

Für den Diskretisierungsfehler t der Verteilungskurve verwenden wir 3 Werte, einmal für eine Kodierung mit 8 bit, einmal mit 16 bit und einmal mit 32 bit. Dies ergibt Werte für t wie in Tabelle 5.2 dargestellt. Für die Kodierung der Datenpunkte setzen wir eine Kodierungslänge von 32 bit pro Dimension. Weiters setzen wir $K_1 = 1.0$ und die beiden Parameter K_2 und K_3 relativ dazu. Aufgrund der Parameter t und σ ergeben sich Werte für den Parameter K_2 wie in Tabelle 5.3 angeführt (Gleichung (4.51)).

Sei im folgenden $\mathcal{D}(\mathbf{x})$ die Dimensionalität von $\mathbf{x}$, dann läßt sich der relative Parameter K_3 wie folgt bestimmen.

$$K_3 = \frac{\mathcal{D}(\boldsymbol{\alpha}) + \mathcal{D}(\mathbf{a}) + \mathcal{D}(\mathbf{w})}{\mathcal{D}(\mathbf{v})}. \tag{5.34}$$

Dies ergibt für $K_3 = 4.75$, da die einzelnen Terme folgende Werte ihrer Dimensionalität besitzen.

$$\begin{aligned} \boldsymbol{\alpha} \in \mathbb{R}^3 &\rightarrow \mathcal{D}(\boldsymbol{\alpha}) = 3 \\ \mathbf{a} \in \mathbb{R}^4 \times \mathbb{R}^3 &\rightarrow \mathcal{D}(\mathbf{a}) = 12 \\ \mathbf{w} \in \mathbb{R}^4 &\rightarrow \mathcal{D}(\mathbf{w}) = 4 \\ \mathbf{v} \in \mathbb{R}^4 &\rightarrow \mathcal{D}(\mathbf{v}) = 4 \end{aligned}$$

Zunächst wird das Training ohne Reduktionsschritt gestartet. Dabei stellt man den typisch stark abfallenden *open loop* bzw. *final** Fehler innerhalb der ersten 1000 Trainingseinheiten (Abbildung 5.15(a)) fest. Ab ca. 1000 Trainingseinheiten (Abbildung 5.15(b)) beginnt das Netzwerk den *Roboterendeffektor* mit einem nahezu konstanten *open loop* bzw. *final* Fehler anzusteuern. Der *open loop* Fehler pendelt sich auf ca. 2.2 *mm* und der *final* Fehler auf 0.1 *mm* ein (Werte in 3D Weltkoordinaten). Man erkennt, daß der *final* Fehler den vorgegebenen maximalen Fehler err_{max} erreicht.

Beim Training mit Reduktion wird festgelegt, daß der erste Reduktionsschritt nach 1500 Trainingsschritten gestartet wird. Alle weiteren Reduktionsschritte werden anschließend in einem Intervall von 1000 Trainingsschritten angestoßen. Abbildung 5.17 zeigt die letzten 14000 Trainingsschritte für verschiedene Kodierungen der Residuen, da der Verlauf der ersten 1000 Trainingsschritte, dem aus Abbildung 5.15(a) entspricht. Äquivalent dazu zeigt Abbildung 5.18 den Verlauf der Netzwerkgröße während des Trainings. Man erkennt, daß für den Fall eines niedrigeren Diskretisierungsfehlers t (16 und 32 bit) ein leicht besserer *open loop* Fehler erreicht wird (Abbildung 5.17(b) und 5.17(c)), als im Fall von 8 bit. Dies ist darauf zurückzuführen, daß in der 16 bzw. 32 bit Variante 7 Referenzvektoren übrigbleiben (Abbildung 5.18(b) und 5.18(c)) und in der 8 bit Variante nur 6. Je höher die Anzahl Netzwerkknoten, desto besser kann die zu lernende Funktion approximiert werden.

*Dies ist jener Fehler, der nach dem letzten Korrekturschritt vorliegt.

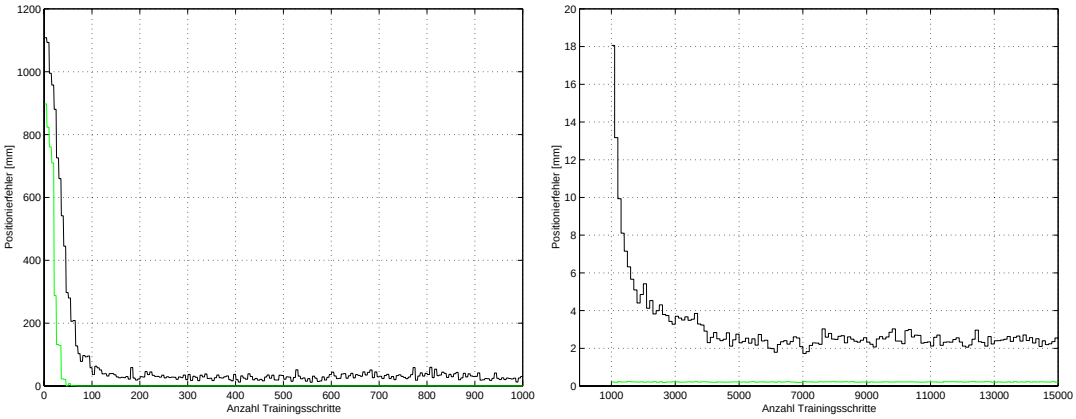
bit	K_2	$K_2/32$
8	$1.35\ 10^5$	$4.21\ 10^3$
16	$6.62\ 10^5$	$2.07\ 10^4$
32	$1.71\ 10^6$	$5.35\ 10^4$

Tabelle 5.3: Parameter K_2 für verschiedene Kodierungen.

Man erkennt auch, daß, trotz der stark unterschiedlichen Diskretisierungswerte t (Tabelle 5.2) und der daraus resultierenden Werte K_2 (Tabelle 5.3), die Anzahl resultierender Netzwerkknoten (Referenzvektoren) sehr ähnlich ist. Dies kann als Maß für die Robustheit des *MDL* Algorithmus gegenüber der Wahl der Parameterwerte gedeutet werden. In allen 3 Varianten ergab sich ein *open loop* Fehler von ca. 3.5 mm. Es ist demnach möglich, anstelle von 20 Netzwerkknoten die *hand-eye* Koordination mit nur 6 Netzwerkknoten zu erlernen und dabei nur einen geringfügig höheren *open loop* Fehler in Kauf zu nehmen. Allerdings erreicht der *final* Fehler die selbe Positioniergenauigkeit wie im Fall der höheren Netzwerkkomplexität. Dies liegt daran, daß die Korrekturschritte diesen *open loop* Fehler verbessern und den *Roboterendeffektor* mit einer maximalen Positionierungenauigkeit $\mathrm{err}_{\max}$ in Richtung Zielposition bewegen.

Abbildung 5.19 zeigt die Positionen der Referenzvektoren im letzten Trainingsschritt aus Sicht der beiden Kameras und in der 3D Darstellung. Da die Version mit 16 bit auch 7 resultierende Referenzvektoren zur Folge hat, wird nur die Version mit 32 bit Kodierungslänge angeführt.

Gehen wir von der Bestimmung des Gewichtungsparameters K_2 mit Hilfe der Diskretisierungsfehler σ und t ab und untersuchen das Verhalten des Trainings bei beliebigen Werten $K_2 \in \{10, 100\}$. Abbildung 5.20 zeigt den Verlauf des *open loop* bzw. des *final* Fehlers während der letzten 14000 Trainingsschritte. In Abbildung 5.20(a) sieht man, daß bis zum Trainingsschritt 4500 der Fehlerverlauf wie in den bisherigen Graphen verläuft. Bis zu diesem Zeitpunkt hat das Netzwerk eine Größe von 5 Knoten (Abbildung 5.21(a)). Im Trainingsschritt 4500 wird das Netzwerk auf 3 Knoten reduziert, wodurch ein sprunghaftes Ansteigen des *open loop* Fehlers auftritt, der sich im weiteren auf den Wert von ca. 5 mm einpendelt. Im anderen Fall ($K_2 = 100$; Abbildung 5.20(b)) pendelt sich der *open loop* Fehler auf einen Wert von ca. 3.7 mm ein und das *ENG* benötigt dafür lediglich 5 Netzwerkknoten (Abbildung 5.21(b)). Die Positionen der resultierenden Referenzvektoren zeigt



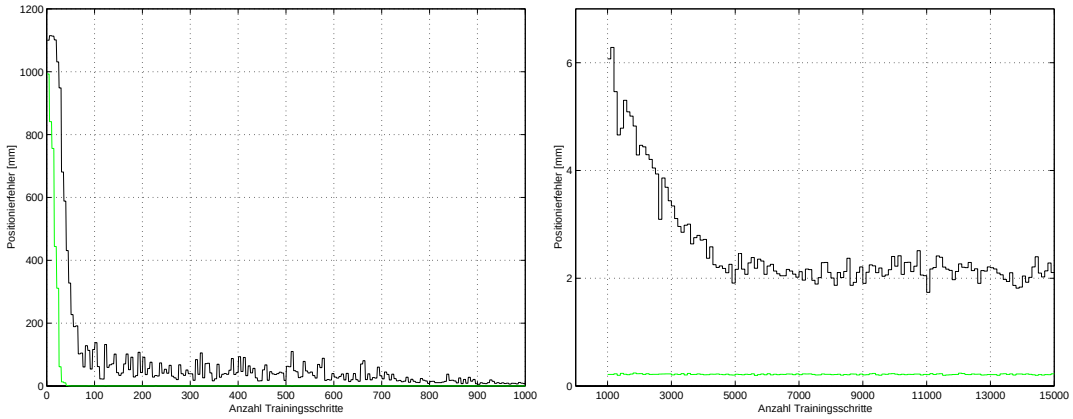
(a) **(b)**

Abbildung 5.15: *Extended Neural Gas:* (a) Typischer Verlauf des *open loop* bzw. des *final* Fehlers innerhalb der ersten 1000 Trainingsschritte (Werte entsprechen Mittelwerte über die letzten 20 Trainingsschritte). (b) Einpendeln des *open loop* Fehlers auf ca. $2.2\ mm$ und des *final* Fehlers auf $0.1\ mm$ (Werte entsprechen Mittelwerte über die letzten 100 Trainingsschritte). Dunkle Linie stellt *open loop* Fehler, helle Linie stellt *final* Fehler dar (in mm der 3D Weltkoordinaten).

Abbildung 5.22. Wie man sieht, wurden Werte für den Parameter K_2 aus einem großen Bereich ($K_2 \in [10, \ldots, 87700]$) ausgewählt und dabei festgestellt, daß sich die Reduktion mittels *MDL* Algorithmus für *fuzzy Clustering* sehr robust gegenüber dem Parameter K_2 verhält (*open loop* Positionierfehler $||\mathbf{p}-\mathbf{r}^{(0)}|| \in [3.6, \ldots, 5]\ mm$). Weiters ist darauf hinzuweisen, daß in allen Beispielen die erforderliche maximale Positionierungenauigkeit von $0.1\ mm$ erreicht wurde.

5.4.5.2 Local Linear Maps mit Nachbarschaftskooperation

Analog zum Beispiel des *ENG* Modells kodieren wir die Datenpunkte mit 32 bit und übernehmen auch die Berechnung der Standardabweichung $\sigma = 0.0039$ (Gleichung (5.33)). Startet man das Training der *LLM* mit Nachbarschaftskooperation ohne Reduktion, so ergibt sich der Verlauf des *open loop* bzw. *final* Fehlers wie in Abbildung 5.16 dargestellt. Die Netzwerkgröße betrug wieder 20 Netzwerkknoten. Vergleicht man den Verlauf der Positionierfehler mit jenen des *ENG* Modells (Abbildung 5.15), so erkennt man, daß beide Netzwerke vergleichbare Verläufe besitzen. Jedoch zeigt sich innerhalb der ersten 1000 Trainingsschritte, daß das *ENG* Mo-

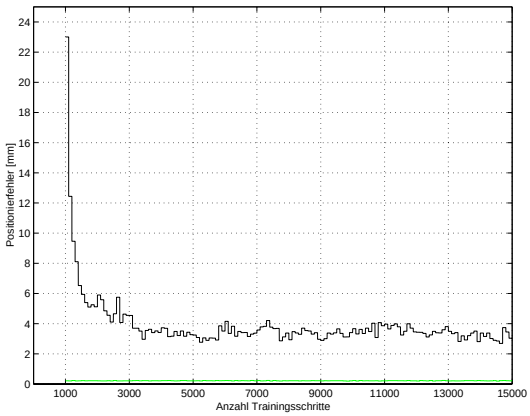


(a) **(b)**

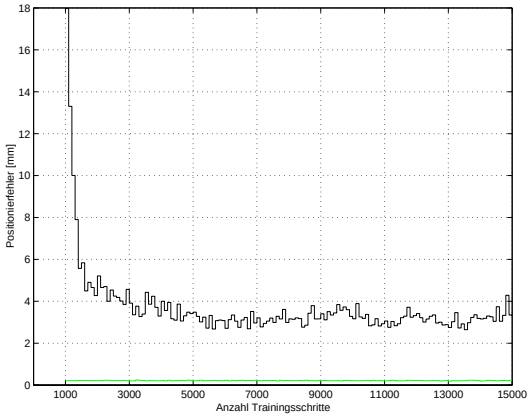
Abbildung 5.16: *Local Linear Maps:* (a) Typischer Verlauf des *open loop* bzw. des *final* Fehlers innerhalb der ersten 1000 Trainingsschritte (Werte entsprechen Mittelwerte über die letzten 20 Trainingsschritte). (b) Einpendeln des *open loop* Fehlers auf ca. $2.2\ mm$ und des *final* Fehlers auf $0.1\ mm$ (Werte entsprechen Mittelwerte über die letzten 100 Trainingsschritte). Dunkle Linie stellt *open loop* Fehler, helle Linie stellt *final* Fehler dar (in mm der 3D Weltkoordinaten).

dell eine wesentlich genauere *open-loop* Positionierung vornimmt. Dies liegt darin, daß der Netzwerkoutput nur durch eine einzige Netzwerkeinheit vorgenommen wird und diese zu Beginn eine schlechte lokale Approximation besitzt. Eine große Positionierungenauigkeit einer Netzwerkeinheit während der Anfangsphase wirkt sich dann beim Lernen auf alle anderen Netzwerkeinheit aus, da eine *winner-takes-most* Lernstrategie eingesetzt wird. Im Falle des *ENG* Modells wirkt sich eine ungenaue Netzwerkeinheit bei der Berechnung des Netzwerkoutputs nicht so stark aus, wie im Fall der *LLM* mit Nachbarschaftskooperation, da mittels *winner-takes-most* Strategie der Netzwerkoutput über alle Netzwerkeinheiten gewichtet wird. Nach etwa 1000 Trainingsschritten hat das *LLM* Modell jedoch alle Netzwerkeinheit so gut trainiert, daß sich das gleiche Verhalten einstellt, wie beim *ENG* Modell.

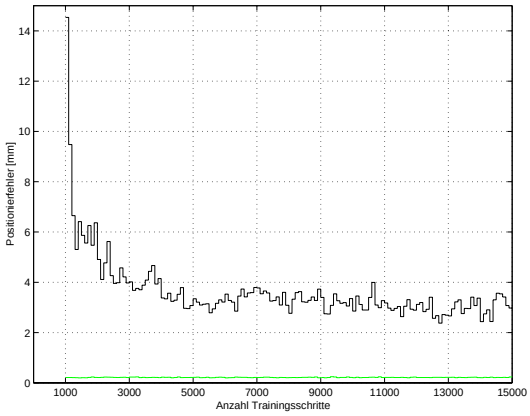
Wie Abbildung 5.16 zeigt, pendelt sich der *open-loop* Fehler auf ca. 2.2 mm und der *final* Fehler auf ca. 0.1 mm ein. Die Idee die Reduktion des *LLM* Modells mit Nachbarschaftskooperation mit Hilfe des *MDL* Algorithmus für *crisp Clustering* vorzunehmen hat gezeigt, daß sie sehr rechenintensiv ist und deshalb keine umfangreichen Experimente vorgenommen wurden. Die wenigen Versuche haben jedoch gezeigt, daß die Reduktion und deren Ergebnisse vergleichbar sind.



(a) 8 bit

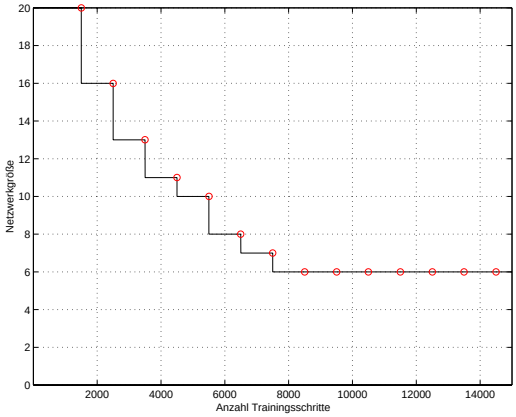


(b) 16 bit

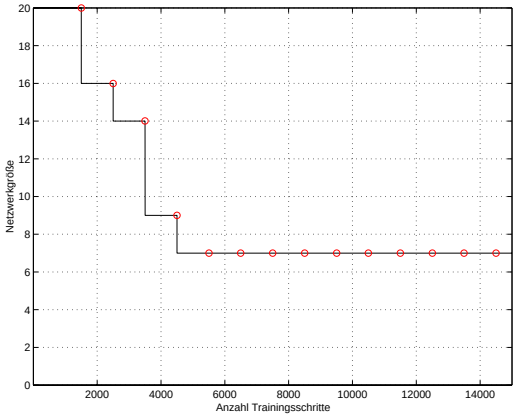


(c) 32 bit

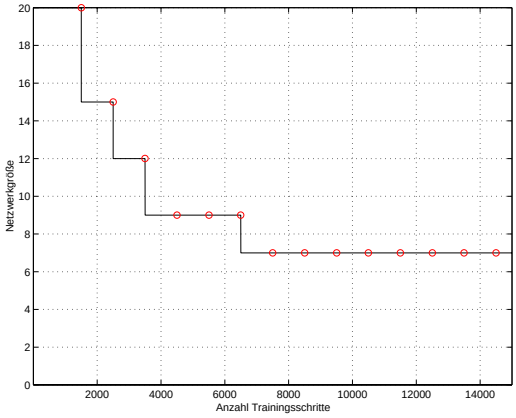
Abbildung 5.17: *open loop* bzw. *final* Fehler der letzten 14000 Trainingsschritte bei unterschiedlichen t Werte (Werte entsprechen Mittelwerte über die letzten 100 Trainingsschritte). Dunkle Linie stellt *open loop* Fehler, helle Linie stellt *final* Fehler dar (in mm der 3D Weltkoordinaten).



(a) 8 bit

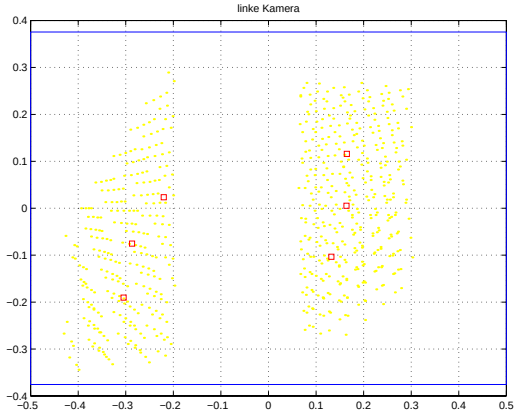


(b) 16 bit

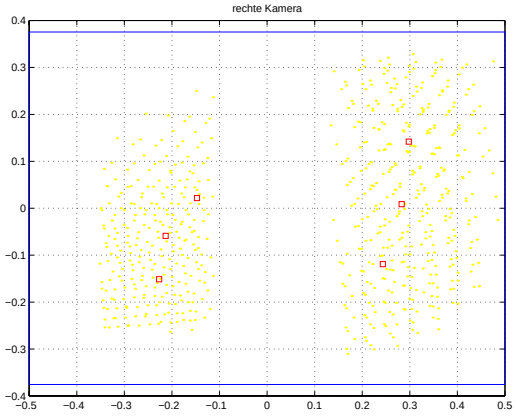


(c) 32 bit

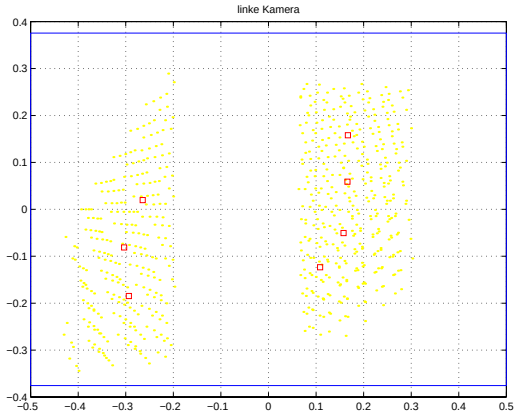
Abbildung 5.18: Netzgröße im Laufe des Trainings bei unterschiedlichen Werten t (Kreise geben Zeitpunkte des Starts der Reduktionmethode an).



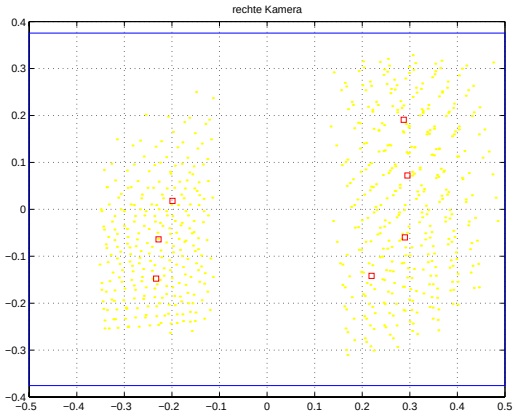
(a) 8 bit



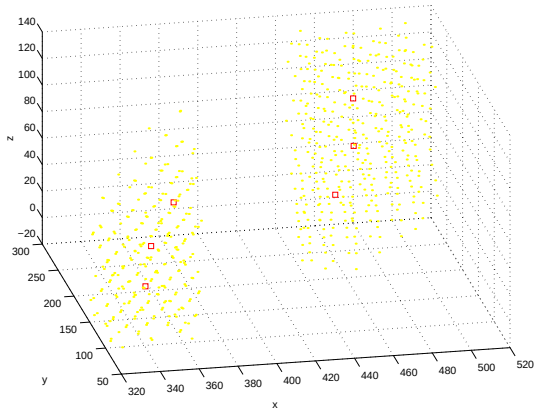
(b) 8 bit



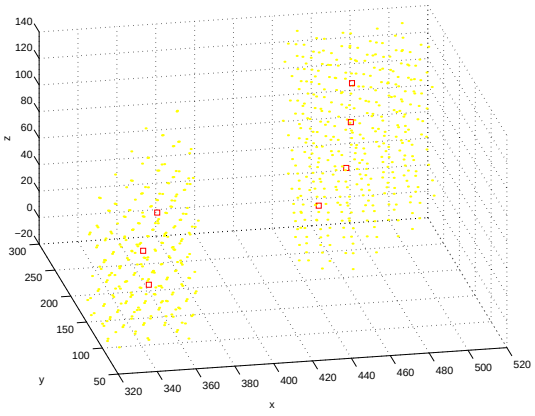
(c) 32 bit



(d) 32 bit

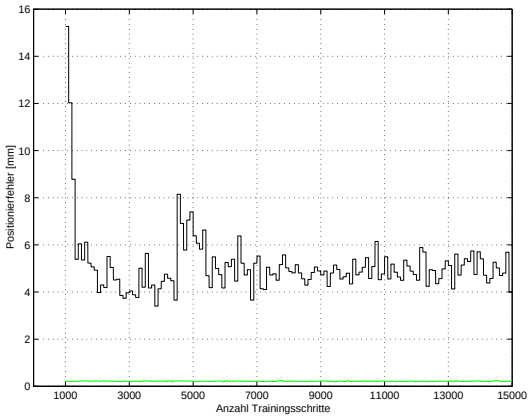


(e) 8 bit

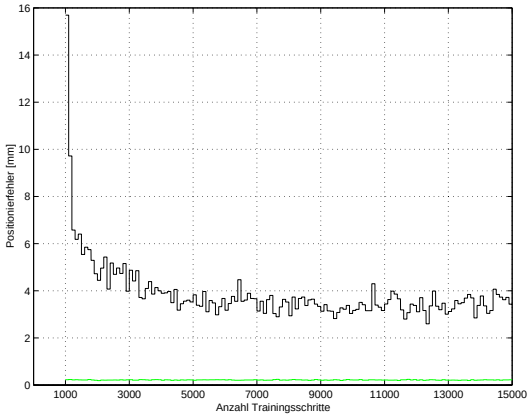


(f) 32 bit

Abbildung 5.19: Kameraansichten und 3D Ansichten der Datenpunkte (Punkte) und Referenzvektoren (Rechtecke) am Ende des Trainings.

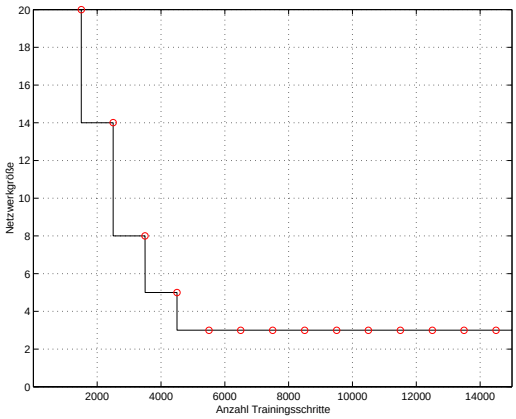


(a) $K_2 = 10$

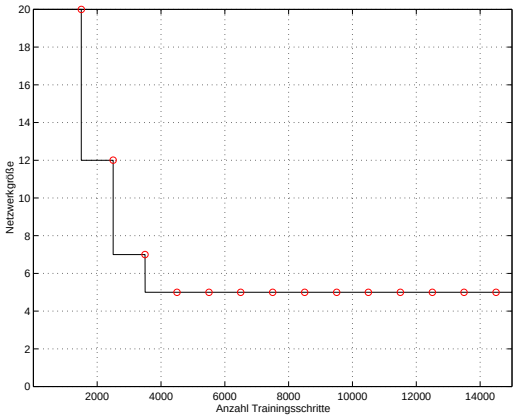


(b) $K_2 = 100$

Abbildung 5.20: Fehler der letzten 14000 Trainingsschritte. (Werte entsprechen Mittelwerte über die letzten 100 Trainingsschritte). Dunkle Linie stellt *open loop* Fehler, helle Linie stellt *final* Fehler dar (in mm der 3D Weltkoordinaten).

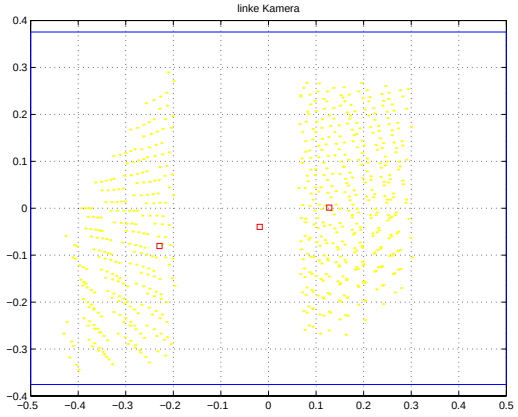


(a) $K_2 = 10$

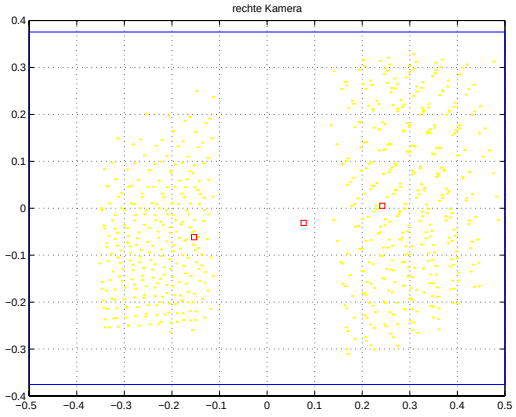


(b) $K_2 = 100$

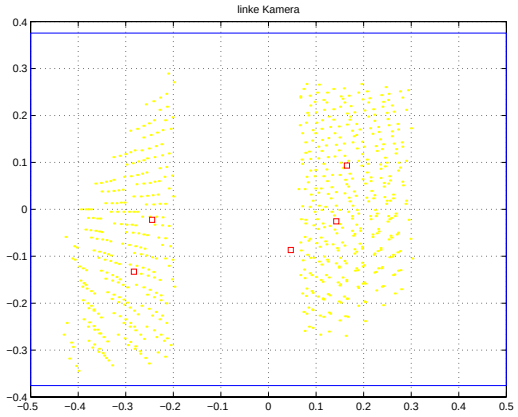
Abbildung 5.21: Netzgröße im Laufe des Trainings (Kreise geben Zeitpunkte des Starts der Reduktionmethode an).



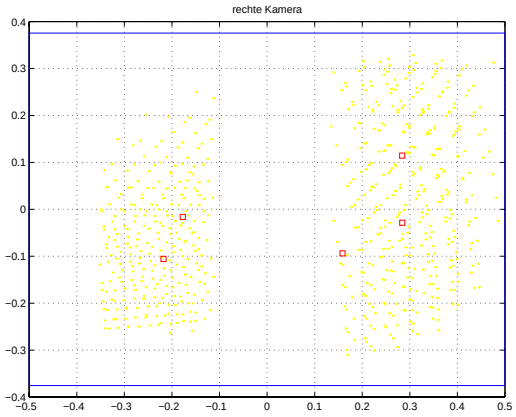
(a) $K_2 = 10$



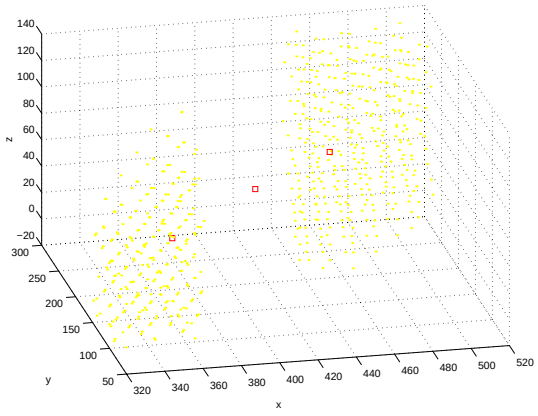
(b) $K_2 = 10$



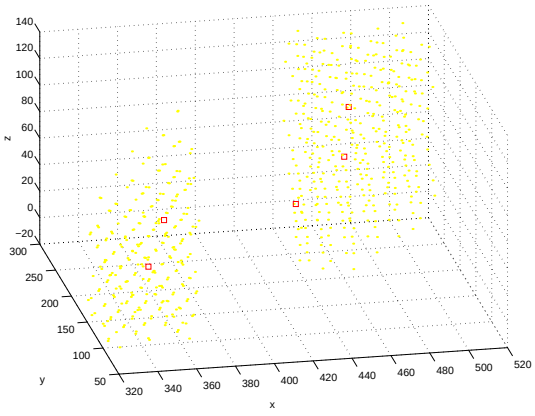
(c) $K_2 = 100$



(d) $K_2 = 100$



(e) $K_2 = 10$



(f) $K_2 = 100$

Abbildung 5.22: Kameraansichten und 3D Ansichten der Datenpunkte (Punkte) und Referenzvektoren (Rechtecke) am Ende des Trainings.

Kapitel 6

Diskussion

In dieser Diplomarbeit wurden zwei Modelle vorgestellt, welche in der Lage sind, die optimale Größe von neuralen Netzen für *Clustering* Aufgaben zu bestimmen, indem eine große initiale Menge an Referenzvektoren auf optimale Größe reduziert wird. Ein Modell kann speziell für *crisp Clustering* und eines für *fuzzy Clustering* Netzwerke verwendet werden. Dabei hat sich das Prinzip der *minimalen Beschreibungslänge* (*minimum description length*; MDL) als Komplexitätskriterium bewährt, welches neben seiner Anwendung für *unsupervised* Netzwerke auch die Möglichkeit bot, selbst in das Kodierungsmodell eingreifen zu können. Diese Tatsache erlaubt es, verschiedene Instanzierungen des *MDL* Algorithmus zu generieren, indem man die Kodierung der Fehlerterme entsprechend des verwendeten *Clustering* Algorithmus definiert. Somit ist es möglich verschiedene *Clustering* Algorithmen als Trainingsalgorithmen der Referenzvektorpositionen einzusetzen.

Weiters wurde ein *MDL* Kriterium präsentiert, welches erlaubt, *outlier* zu erkennen und so den *Clustering* Prozeß und die Bestimmung der Anzahl Datencluster robust gegenüber diesen *outlier* zu machen. Wir haben auch gezeigt, wie sich die einzelnen Parameter der *MDL* Algorithmen auf die *Clustering* Ergebnisse auswirken und man so die erwarteten Ergebnisse steuern kann. In diesem Zusammenhang bieten sich die *MDL* Algorithmen für die Anwendung von hierarchischem *Clustering* an. Um die Effizienz der *MDL* Algorithmen zu steigern, wurden das *Growing Neural Gas* (GNG) Modell verwendet, um die Initialisierung der Menge der Referenzvektoren vorzunehmen. Dies hatte den Vorteil, daß diese Initialisierung, gegenüber

der zufälligen Initialisierung, Rücksicht auf die Verteilung der Datenpunkte nimmt. Zusätzlich konnte dadurch die Anzahl benötigter Rechenschritte reduziert werden, da man eine relativ knappe initiale Menge an Referenzvektoren bekommt, die den Rechenaufwand der *Clustering* Algorithmen reduzierte. Eine weitere Reduktion des Rechenaufwandes konnte dadurch erreicht werden, indem man die *Clustering* Algorithmen nicht bis zur vollständigen Konvergenz ausführte. Dies macht die *MDL* Algorithmen zu sehr effizienten *pruning* Algorithmen. Um auf sich verändernde Datenmengen reagieren zu können, haben wir noch eine Wachstumskomponente präsentiert, die den *MDL* Algorithmus für *crisp Clustering* erlaubt, neue Referenzvektoren hinzuzufügen und so zusätzlich entstehende *Datencluster* zu berücksichtigen. Dies macht den *MDL* Algorithmus für *crisp Clustering* zusätzlich robust gegenüber der Anfangsinitialisierung, da nicht mehr zwangsweise mit einer großen initialen Menge an Referenzvektoren begonnen werden muß.

Im Zuge einiger Anwendungsbeispiele wurde demonstriert, wie sich die *MDL* Algorithmen verhalten. Neben den Beispielen der Vektorquantisierung, der Datenkompression (Bildquantisierung) und der *RGB* Farbbildsegmentierung wurde auch eine Anwendung zur Erlernung der *hand-eye* Koordination von Robotern präsentiert. In allen Anwendungen haben sich die *MDL* Algorithmen als robuste Algorithmen zur Reduktion der Netzwerkgrößen herausgestellt. Jedoch konnten die Eigenschaften der parallelen Reduktion und der lokalen Betrachtung der Änderung der Beschreibungslänge im Anwendungsbeispiel der *hand-eye* Koordination nicht immer angewandt werden. Dies rührt davon, daß die Fehlerterme vom Quantisierungsprozeß getrennt sind, da der Fehler aufgrund der zu lernenden Abbildung entsteht und nicht mehr durch die Quantisierung selbst. Trotzdem können die *MDL* Algorithmen angewandt werden, nur sind sie nicht mehr so effizient bezüglich der Rechenleistung wie in den anderen Anwendungen.

Die *MDL* Algorithmen waren jeweils als *off-line* Reduktionsalgorithmen eingesetzt. Einzig im Beispiel der *hand-eye* Koordination wurde ein *on-line* Lernverfahren eingesetzt, jedoch wurde beim Einsatz der *MDL* Algorithmen angenommen, daß sich bei jedem Start der Reduktion ein stabiler Netzwerkzustand eingestellt hat. Als Ausblick wäre es wünschenswert, eine Definition des *MDL* Kriteriums zur Reduktion/Erhöhung der Netzwerkgröße für *on-line* Lernverfahren zu finden und so die Netzwerkgröße bei jeder Präsentation eines neuen Datenpunktes effizient zu bestimmen.

Literaturverzeichnis

[1] H. Akaike. Information Theory and An Extension of the Maximum Likelihood Principle. In B. N. Petrov and F. Csaki, editors, *Second International Symposium on Information Theory*, Budapest, 1973. Akad. Kiado.

[2] J. C. Bezdek. *Pattern Recognition With Fuzzy Objective Function Algorithms.* Plenum Press, New York, 1981.

[3] H. Bischof and A. Leonardis. Vector Quantization and Minimum Description Length. *ICPR'98*, pages 355–364, 1998.

[4] H. Bischof, A. Leonardis, and A. Selb. MDL Principle for Robust Vector Quantization. *Pattern Analysis and Applications*, 2(1):59–72, 1999.

[5] C. M. Bishop. *Neural Networks for Pattern Recognition.* Oxford University Press, New York, 1995.

[6] C. J. C. Burges. A Tutorial on Support Vector Machines for Pattern Recognition. *Data Mining and Knowledge Discovery*, 2(2):121–167, 1998.

[7] V. S. Cherkassky and F. M. Mulier. *Learning from Data : Concepts, Theory, and Methods.* John Wiley and Sons, New York, 1998.

[8] R. O. Duda and P. E. Hart. *Pattern Classification and Scene Analysis.* Wiley, New York, 1973.

[9] R. A. Fisher. The Use of Multiple Measurements in Taxonomic Problems. *Annals of Eugenics 7*, 2:179–188, 1936.

[10] H. Frigui and R. Krishnapuram. A Robust Competitive Clustering Algorithm With Applications in Computer Vision. *PAMI*, 21(5):450–465, Mai 1999.

[11] B. Fritzke. Growing Cell Structures - A Self-organizing Network for Unsupervised and Supervised Learning. *Neural Networks*, 7(9):1441–1460, 1994.

[12] B. Fritzke. A Growing Neural Gas Network Learns Topologies. In G. Tesauro, D. Touretzky, and T. Leen, editors, *Advances in Neural Information Processing Systems 7*, pages 625–632. MIT-Press Cambridge MA, 1995.

[13] B. Fritzke. Incremental Learning of Local Linear Maps. In F. Fogleman and P. Gallinari, editors, *International Conference on Artificial Neural Networks*, pages 217–222, Paris, 1995.

[14] B. Fritzke. The LBG-U Method for Vector Quantization– an Improvement over LBG Inspired from Neural Networks. *Neural Processing Letters*, 5(1):35–45, 1997.

[15] B. Fritzke. Some Competitive Learning Methods. Technical report, Institute for Neural Computation, Ruhr-Universität Bochum, 1997.

[16] S. Geman, E. Bienenstock, and R. Doursat. Neural networks and the bias/variance dilemma. *Neural Computation*, 4:1–58, 1992.

[17] F. Glover and M. Laguna. Tabu search. *Modern Heuristic Techniques for Computational Problems*, pages 70–150, 1993.

[18] S. Haykin. *Neural Networks A Comprehensive Foundation*. McMillan College Publishing Company Inc., New York, 1994.

[19] T. Kohonen. *Self-Organization and Associative Memory*. Springer-Verlag, New York, 2nd edition, 1988.

[20] T. Kohonen. *Self-Organizing Maps*. Springer Verlag, Heidelberg, 1995.

[21] F. Leisch. Statistical Model Selection for Artificial Neural Networds: A Review. Technical report, Abteilung für Wahrscheinlichkeitstheorie und Stochastische Prozesse, Technische Universität Wien, 2000.

[22] A. Leonardis and H. Bischof. An Efficient MDL-based Construction of RBF Networks. *Neural Networks*, 11(5):963–973, 1998.

[23] A. Lokketangen. Tabu Search – Using the Search Experience to Guide the Search Process. An Introduction with Examples. *AI Communications*, 8(2):78–85, 1995.

[24] J. B. MacQueen. Some Methods for Classification and Analysis of Multivariate Observations. In L. M. LeCam and J. Neyman, editors, *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297, Berkeley, 1967. University of California Press.

[25] T. M. Martinetz. *Selbstorganisierende neuronale Netzwerkmodelle zur Bewegungssteuerung*. PhD thesis, TU München, St. Augustin, 1992.

[26] T. M. Martinetz. Competitive Hebbian Learning Rule Forms Perfectly Topology Preserving Maps. In *International Conference on Artificial Neural Networks*, pages 427–434, Amsterdam, 1993. Springer.

[27] T. M. Martinetz and K. J. Schulten. A Neural Gas Network Learns Topologies. In T. Kohonen, K. Mäkisara, O. Simula, and J. Kangas, editors, *Artificial Neural Networks*, pages 397–402, Amsterdam, 1991. Elsevier Science Publishers B.V.

[28] T. M. Martinetz and K. J. Schulten. Topology Representing Networks. *Neural Networks*, 7(3):507–522, 1994.

[29] T. Melzer. Adaptive Robotersteuerung mittels visueller Rückkopplung. Master's thesis, Technische Universität Wien, Wien, 1997.

[30] N. Murata, S. Yoshizawa, and S. Amari. Network Information Criterion - Determining the Number of Hidden Units for an Artificial Neural Network Modell. *IEEE Transactions on Neural Networks*, 5(6):865–872, 1994.

[31] B. D. Ripley. *Pattern Recognition and Neural Networks*. Cambridge University Press, New York, 1996.

[32] J. Rissanen. Modeling by Shortest Data Description. *Automatica*, 14:465–471, 1978.

[33] H. Ritter. Learning With The Self-Organizing Map. In T. Kohonen, K. Mäkisara, O. Simula, and J. Kangas, editors, *Artificial Neural Networks*, pages 357–364, Amsterdam, 1991. Elsevier Science Publishers B.V.

[34] H. Ritter, T. Martinetz, and K. Schulten. *Neural Computation and Self-Organizing Maps: An Introduction*. Addison-Wesley, New York, 1992.

[35] A. Selb. Praktikum: GNG und Fuzzy k-Means im MDL Modell. Technical report, Abteilung für Mustererkennung und Bildverarbeitung, Technische Universität Wien, 1999.

[36] A. Selb. Praktikum: Robotersimulator und Bestimmung der minmalen Netzwerkgröße. Technical report, Abteilung für Mustererkennung und Bildverarbeitung, Technische Universität Wien, 2000.

[37] A. Selb, H. Bischof, and A. Leonardis. Growing and Selection for Vector Quantization Design. In N. Brändle, editor, *Proceedings of the Computer Vision Winter Workshop'99*, pages 34–43, Wien, 1999. Pattern Recognition and Image Processing Group, Vienna University of Technology.

[38] A. Selb, H. Bischof, and A. Leonardis. Fuzzy C-means in an MDL-Framework. *ICPR 2000 (accepted)*, 2000.

[39] V. N. Vapnik and A. Y. Chervonenkis. On the Uniform Convergence of Relative Frequencies of Events to their Probabilities. *Theoretical Probability and Its Applications*, 17:264–280, 1971.