Diplomarbeit zum Thema

Entwurf und Implementierung einer Applikation zur Visualisierung von Lernvorgängen bei Selbstorganisierenden Karten

zur Erlangung des akademischen Grades
Diplom-Informatiker (FH)

vorgelegt dem
Fachbereich Informatik und Medien der
Fachhochschule Brandenburg

Benjamin Hoepner
25. Juli 2007

Erstprüfer: Dipl.-Inform. Ingo Boersch
Zweitprüfer: Prof. Dr.-Ing. Jochen Heinsohn

Inhaltsverzeichnis

1. Einführung

1.1. Aufgabenstellung

Selbstorganisierende Karten stellen eine besondere Form von künstlichen neuronalen Netzen dar, die sich unüberwacht trainieren lassen. Ziel der Arbeit ist die Konzeption und Implementierung einer Anwendung zum Training von selbstorganisierenden Karten. Schwerpunkt ist hierbei die Darstellung des Lernverlaufs und die Visualisierung der Karte.

Ausgangspunkt der Arbeit sei die vorhandene Applikation *SOMARFF* [Sch06], die in ihrem Funktionsumfang zu analysieren ist. Die neue Applikation soll den bestehenden Funktionsumfang in den Bereichen Datenvorverarbeitung, Training und Visualisierungen übernehmen und weitere Visualisierungen, wie „P-Matrix“ oder „Karte im Eingaberaum“ enthalten. Zusätzlich soll der Quantisierungsfehler geeignet dargestellt werden.

Wesentliche Eigenschaften selbstorganisierender Karten sollen abstrahiert und austauschbar gestaltet werden, um es zu ermöglichen, neue Topologien, Distanzmetriken, Datenquellen und Nachbarschaftsfunktionen zu integrieren. Besonderer Wert wird dabei auf die Wiederverwendbarkeit von Modulen und Erweiterbarkeit durch neue Module gelegt.

Bestandteil der Arbeit ist weiterhin eine aussagekräftige Dokumentation des Systems für Entwickler, eine Nutzeranleitung und der Nachweis der Funktionsfähigkeit des Programms durch geeignete Experimente.

1.2. Motivation

Seit dem Beginn der zentralisierten Datenhaltung sammelt sich eine unüberschaubare Flut an Informationen, die durch die Verbreitung des Internets neue Dimensionen erreicht haben. Diese Daten sind heute an vielen Orten auffindbar, jedoch ist ihr geheimes Potential weitestgehend ungenutzt. Denn es ist anzunehmen, dass Regelmäßigkeiten oder Muster darin bislang zwar unentdeckt aber höchstwahrscheinlich vorhanden sind.

Die Aufbereitung dieser Daten mit geeigneten Mitteln ist heute wichtiger denn je. So wird es möglich, die enthaltenen Informationen sichtbar zu machen, sie zu visualisieren, um unbekanntes Wissen aufzuspüren. Dadurch ist es sogar möglich, diese Daten so darzustellen, dass selbst ein Laie ohne fachspezifische Vorkenntnisse diese Zusammenhänge erkennen kann.

Ein Beispiel zur Verdeutlichung der Thematik wäre ein Krankenhaus, das seit langer Zeit Informationen über seine Patienten sammelt. Dazu gehört eine unglaublich lange Liste mit Krankheitssymptomen, von Fußpilz über Gelenkrheumatismus und Diabetes bis hin zu Knochenkrebs. Zu jedem Patienten kann mit dieser Liste eine Aussage getroffen werden, ob ein solches Symptom auf ihn zutrifft oder nicht. Man stelle sich vor, alle diese Daten auf einer einfachen Landkarte abzubilden, die z.B. Symptomhäufungen als Berge darstellt. Es wäre weiterhin denkbar, dass sich jeder einzelne Patient, bestimmt durch sein Krankheitsbild, auf einem Punkt dieser Landkarte aufhält. Ein Patient könnte sich auf dieser Karte in der Nähe eines Symptomberges, der z.B. Krebs repräsentiert, wiederfinden, obwohl eine solche Erkrankung bisher nicht diagnostiziert wurde. Ein solches Vorgehen würde eine große Hilfe für die Früherkennung von Krebsrisiken bedeuten.

Diese Arbeit befasst sich mit einem Verfahren, das eine Abbildung hochdimensionaler Daten auf eine zweidimensionale Karte erlaubt, und somit ungeahnte Möglichkeiten bei der Suche nach neuen Erkenntnissen bietet.

2. Selbstorganisierende Karten

Selbstorganisierende Karten sind eine besondere Form neuronaler Netze. Dieses Kapitel befasst sich mit deren Aufbau und Funktion, sowie mit ihrem Training. Ein wichtiger Aspekt dieser Arbeit sind die Darstellungsformen dieser Karten, sie werden ebenfalls einführend vorgestellt. Im Vorfeld werden noch die biologischen Urspünge und die daraus resultierenden künstlichen neuronalen Netze beleuchtet.

2.1. Künstliche neuronale Netze

Für das Verständnis der selbstorganisierenden Karten ist es notwendig, kurz auf die *künstlichen neuronalen Netze* (KNN) in ihrer Funktionalität und Struktur einzugehen, da sie die Grundlage für den Aufbau der SOM bilden.

Ein künstliches neuronales Netz ist ein technisches Konstrukt, das entworfen wurde, um komplexe Vorgänge zwischen Nervenzellen im Gehirn (auch *Neuronen* genannt) auf vereinfachte Art und Weise nachzuahmen. Spezielle Aufgaben der Nervenzellen werden in Betracht gezogen und abstrakt nachgestaltet, mit dem Ziel, eine lernfähige Struktur zu erzeugen.

2.1.1. Das biologische Vorbild

Auf den Aufbau und die Funktionalität von Nervenzellen wird in [Hof93] eingegangen. Die Abbildung 2.1 zeigt die wesentlichen Bestandteile eines Neurons im biologischen Sinne.

Der Zellkörper bildet den Hauptbestandteil jeder Zelle und ist durch die *Zellmembran* von der Außenwelt abgegrenzt.

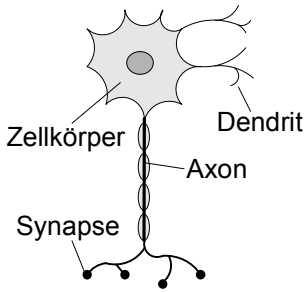

Abbildung 2.1.: Schematische Darstellung einer Nervenzelle

Dendriten sind kurze Fortsätze, die sich stark verzweigt und in großer Anzahl am Zellkörper befinden. Sie dienen der Verbindung mit anderen Nervenzellen.

Das Axon ist ein Auswuchs am Zellkörper, der bis zu einen Meter lang sein kann und sich an seinem Ende in viele Zweige aufteilt, die wiederum jeweils in einer *Synapse* enden.

Synapsen stellen mit anderen Nervenzellen eine Verbindung her, indem sie über einen *synaptischen Spalt* an deren Dendriten und ihre Zellkörper gekoppelt sind.

Zwischen den Nervenzellen besteht eine hochgradige Vernetzung, da viele mit tausenden Synapsen bedeckt sind. Die Verbindungen unterliegen temporalen Veränderungen, da die Synapsen wachsen, verkümmern und auch ganz verschwinden oder Axone neue Verzweigungen ausbilden und so mit anderen Nervenzellen Kontakt aufnehmen. So wird die Struktur des Neuronennetzes angepasst, sein Verhalten verändert sich, ein Lernprozess findet statt.

Die Funktionsweise einer einzelnen Nervenzelle kann wie folgt beschrieben werden: Ursprünglich befindet sie sich im Ruhezustand, sie hat dabei ein *Ruhepotential* von etwa -75 mV. Die Zelle kann über die Synapsen an ihrer Membran in einen Erregungszustand gebracht werden. Das geschieht, wenn die Summe aller anliegenden Synapsenpotentiale einen gewissen Schwellwert (etwa -50 mV) überschreitet. Das Neuron erhöht kurzzeitig durch eine so ausgelöste Kettenreaktion das eigene Potential, nun *Aktionspotential* genannt, auf 30 mV: Es „feuert“. Diese Reaktion wird über das Axon und seine Verästelungen an andere Nervenzellen weitergeleitet, die so über die Aktivität benachrichtigt werden und ebenfalls in den Erregungszustand übergehen können.

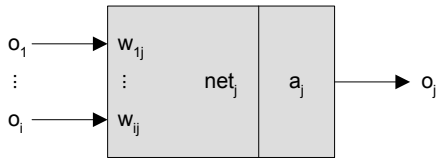

Abbildung 2.2.: Schematische Darstellung eines künstlichen Neurons

Nicht alle Synapsen, die bei einem solchen Energiefluss in Aktion treten, leiten das Potential einfach nur weiter. Sie können es auch hemmen, was dazu führen kann, dass eine feuernde Nervenzelle verhindert, dass ein anderes Neuron in der Kette erregt wird. Dieses Verhalten ist notwendig, um die Stabilität des Nervensystems zu gewährleisten.

Um mit der Umwelt zu interagieren, muss das Nervensystem mit Reizen versorgt werden und Reaktionen darauf auslösen können. Die Reize werden über Sinneszellen aufgenommen und über *afferente* Axone an das Nervensystem weitergeleitet. Die Reaktionen darauf werden mittels *efferenter* Axone an Muskelzellen gesendet, die daraufhin kontrahieren.

2.1.2. Aufbau eines künstlichen Neurons

Ein künstliches Neuron bildet eine Nervenzelle abstrakt in den für einen Lernvorgang wesentlichen Teilen nach, so dass sich der Zustand eines Neurons über eine berechenbare Funktion abbilden lässt.

Wie zuvor beschrieben, ist eine natürliche Nervenzelle mit anderen Nervenzellen verbunden und wird über die Synapsen durch sie beeinflusst. Analog dazu hat ein künstliches Neuron viele Eingänge und entsprechend genauso viele Gewichte, die die Synapsentätigkeit nachahmen, indem sie die Eingangssignale bewerten und somit ihren Einfluss auf die Aktivität des Neurons anpassen. Also muss auch ein künstliches Neuron über eine Aktivität (bzw. ein Potential) verfügen, aus der sich ein Ausgangswert, der der Reaktion über das Axon einer Nervenzelle entspricht, errechnen lässt.

In [Zel03] ist der Aufbau künstlicher neuronaler Netze beschrieben. Künstliche Neuronen sind die Grundelemente des Netzes. Die Abbildung 2.2 zeigt schematisch, wie eine solche Zelle aufgebaut ist.

Sie verfügt über eine *Netzeingabe* $net_j(t) \in \mathfrak{R}$, die als kumulierter Vertreter der Ein-

gangssignale zum Zeipunkt t steht. Sie wird durch die Ausgabe verbundener Neuronen bestimmt. In anderen Werken (wie [Hof93]) wird diese Größe auch *effektiver Eingangswert* ε genannt.

Der *Aktivierungszustand* $a_j(t) \in \Re$ beschreibt den Grad der Aktivität des Neurons j zum Zeitpunkt t, er repräsentiert das Membranpotential einer Nervenzelle. Eine *Aktivierungsfunktion* f_{act} berechnet den neuen Aktivierungsgrad $a_j(t+1)$ aus dem aktuellen Zustand $a_j(t)$ und der Netzeingabe $net_j(t)$. Häufig wird auch der Schwellenwert des Neurons $\theta_j \in \Re$ als Parameter mit angegeben. Formel 2.1 zeigt die allgemeine Gleichung einer solchen Funktion.

$$a_j(t+1) = f_{act}\left(a_j(t), net_j(t), \theta_j\right) \tag{2.1}$$

Durch die *Ausgabefunktion* $f_{out} : \Re \mapsto \Re$ wird die *Ausgabe* $o_j \in \Re$ des Neurons j über seine Aktivität a_j bestimmt. Die Formel 2.2 zeigt diese Funktion.

$$o_j(t) = f_{out}\left(a_j(t)\right) \tag{2.2}$$

Die Aktivierungs- und Ausgabefunktionen werden oft durch nichtlineare Funktionen umgesetzt. Eine sehr häufig eingesetzte Variante für die Aktivierung ist die binäre Schwellenwertfunktion (Gleichung 2.3). Für die Ausgabefunktion wird oft einfach die Äquivalenz genutzt.

$$a_j(t+1) = \begin{cases} 1 & (net_j \geq \theta_j) \\ 0 & (net_j < \theta_j) \end{cases} \tag{2.3}$$

2.1.3. Neuronenverbindungen

Ein weiterer wichtiger Bestandteil künstlicher neuronaler Netze sind die Verbindungen zwischen den Neuronen, das *Verbindungsnetzwerk*. Es handelt sich dabei um einen gerichteten Graphen, dessen Kanten die gewichteten Verbindungen zwischen diesen Neuronen repräsentieren. Ein solches Gewicht wird mit $w_{ij} \in \Re$ bezeichnet, die Indizes stehen für die Verbindung von Neuron i zum Neuron j. Die Verbindungsgewichte bestimmen den Einfluss der Ausgangswerte o_i der Neuronen i auf die Netzeingabe net_j des verbundenen Neurons j.

Jedes Neuron verfügt über einen *Gewichtsvektor* $\vec{w}_j \in \Re^n$, der die Gewichte aller

verbundenen Neuronen enthält. Es existiert auch eine andere Darstellung in Form einer quadratischen Matrix, die *Gewichtsmatrix* $W \in \Re^{n \times n}$ genannt wird und die Verbindungen aller Zellen repräsentiert.

Mit der *Propagierungsfunktion* (dargestellt in Formel 2.4) wird die Netzeingabe eines Neurons errechnet. Dabei werden die Ausgaben der verbundenen Neuronen und die entsprechenden Gewichte berücksichtigt, was die Synapsentätigkeit einer natürlichen Nervenzelle nachbildet.

$$net_j(t) = \sum_i w_{ij} \cdot o_i(t) \tag{2.4}$$

Besteht keine Verbindung zwischen zwei Neuronen i und j, so gilt für das entsprechende Gewicht $w_{ij} = 0$. Eine Hemmung von j durch i wird mit $w_{ij} < 0$ erreicht. Bei einem positiven Gewicht ($w_{ij} > 0$) regt das Neuron i seinen Nachfolger j an.

2.1.4. Netztopologien

Neuronen werden im Allgemeinen in drei Gruppen untergliedert. Es existieren Neuronen in der Eingabeschicht (*input units*), der Zwischenschicht (*hidden units*) und der Ausgabeschicht (*output units*).

Input units bilden die erste Schicht. Sie nehmen einen *Eingabevektor* auf und übertragen ihre Erregung an andere Neuronen innerhalb des Netzes.

Hidden units sind von außerhalb des Netzes nicht „sichtbar“, da ihre Ein- und Ausgaben nur mit Neuronen innerhalb des Netzes verbunden sind. Es können mehrere Zwischenschichten existieren, in denen die *hidden units* angeordnet sind, oder auch gar keine.

Output units bilden die letzte Schicht. Sie erhalten ihre Eingabe aus dem Inneren des Netzes und erzeugen einen *Ausgabevektor*, der die aus dem Eingabevektor resultierende Erregung des Netzes darstellt.

Abbildung 2.3 zeigt ein sogenanntes *Feedforward*-Netz. Hierbei handelt es sich um die gebräuchlichste Form künstlicher neuronaler Netze. Es wird durch einen azyklischen, gerichteten Graphen repräsentiert. Der Wert des Ausgabevektors ist ausschließ-

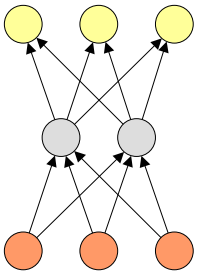

Abbildung 2.3.: Ein typisches Feedforward-Netz: Eingabeschicht unten, Zwischenschicht in der Mitte, Ausgabeschicht oben

lich von dem Eingabevektor abhängig, da nur vorwärtsgerichtete Verbindungen von der Eingabeschicht in Richtung der Ausgabeschicht existieren.

Andere Netze erlauben *Rückkopplungen*. Diese treten auf, wenn ein Neuron sich auf irgendeine Weise selbst beeinflusst. Rückkopplungen können zwischen den Schichten (entgegengesetzt der Ausgaberichtung) stattfinden, von einem Neuron zu sich selbst (*direkte Rückkopplung*) und innerhalb einer Schicht (*laterale Rückkopplung*).

Ein anderes Phänomen ist das Überspringen von Schichten. Es kann z.B. vorkommen, dass *input units* direkt mit *output units* verbunden sind, obwohl Zwischenschichten existieren. Dieser Vorgang wird *shortcut connection* genannt.

Ein Netz wird als *vollständig verbunden* bezeichnet, wenn jedes Neuron mit jedem anderen verbunden ist, nur nicht direkt mit sich selbst.

2.1.5. Lernparadigmen

Das Lernen bei neuronalen Netzen wird durch die Anpassung der Gewichte w_{ij} erreicht. Es können auf diese Weise auch neue Verbindungen aufgebaut und vorhandene gelöscht werden. Hier kommen Algorithmen zum Einsatz, die *Lernregeln* genannt werden.

Überwachtes Lernen

Das überwachte Lernen (*supervised learning*) ist dadurch gekennzeichnet, dass das gewünschte Ergebnis eines Netzes im Vorhinein bekannt ist. Es sind Trainingsdaten (als Eingabemuster) und die jeweilige richtige Ergebnismenge zu Beginn des Trainings vorhanden. Das Netz kann also nach der Präsentation der Muster mit Hilfe eines *Fehler-*

vektors aktualisiert werden. Dieses Paradigma ist sehr zielorientiert.

Ein großer Vorteil dieser Methode ist, dass das trainierte Netz auch mit unbekannten Eingabemustern umgehen kann. Die Trainingsdaten werden also nicht nur „auswendig gelernt“, es wird generalisiert.

Bestärkendes Lernen

Beim bestärkenden Lernen (*reinforcement learning*) werden Eingabemuster an das Netz angelegt. Anschließend wird dem Netz über einen reellen oder Wahrheitswert mitgeteilt, ob das erzeugte Resultat (in Form der Aktivierung der Ausgabeneuronen) richtig oder falsch ist. Dieser Wert kann auch den Grad der Richtigkeit oder Falschheit bestimmen. So wird es möglich, dem trainierten Netz eine Zielrichtung zu geben und es mehr auf die gestellte Aufgabe auszurichten.

Unüberwachtes Lernen

Das unüberwachte Lernen (auch *unsupervised learning* oder *self-organized learning* genannt) ist die im biologischen Sinne plausibelste Methode, allerdings ist sie nicht für alle Aufgabenbereiche einsetzbar. Es sind ausschließlich Eingabemuster gegeben. Das Netz vergleicht sie, um ähnliche Muster in ähnlichen Kategorien einzuordnen. Das beste Beispiel für KNN, die dieses Paradigma einsetzen, sind die *Kohonennetze*.

2.2. Kohonennetze

Nach [Sch03] existieren auf dem Kortex des Menschen sogenannte somatosensorische Areale, die von den sensitiven Regionen des Körpers *innerviert*, d.h. mit Nervenreizen versorgt werden. Dabei ist die Ausdehnung dieser Bereiche nicht proportional zu den flächenmäßigen Verhältnissen der sensitiven Körperregionen, sie wird allein durch die Dichte der Sensoren am entsprechenden Körperbereich bestimmt. So werden z.B. die Lippen und die Finger durch ein großes Areal auf dem Kortex repräsentiert, der Rücken innerviert nur einen kleinen Bereich (s. Abb. 2.4). Zudem zeigt sich eine weitgehend *somatotope Gliederung*, benachbarte Körperregionen sind auch auf dem Kortex benachbart angeordnet. Es existiert also eine Karte, auf der sich die sensitive Peripherie des Körpers abbildet.

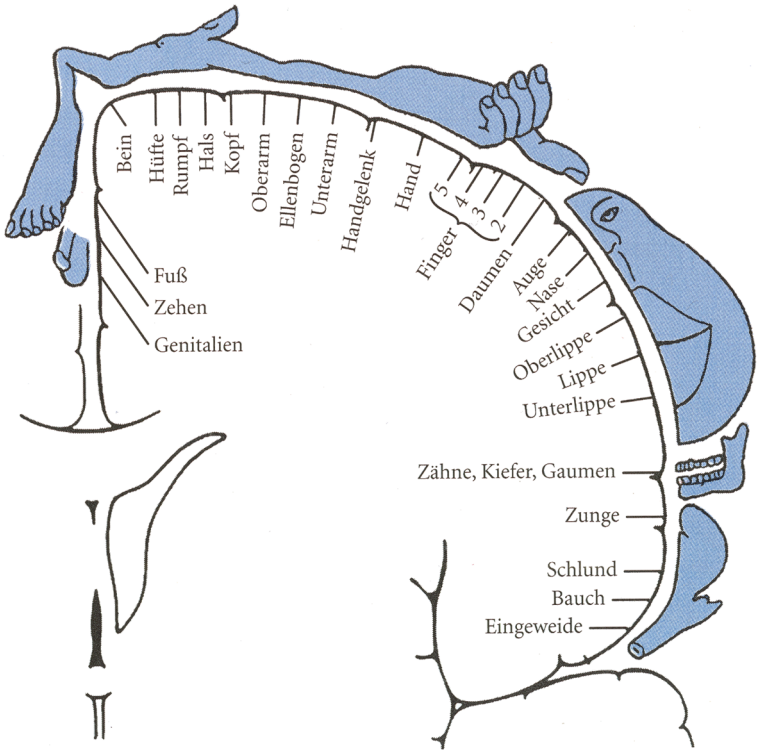

Abbildung 2.4.: Der somatosensorische Kortex des Menschen (aus [Sch03])

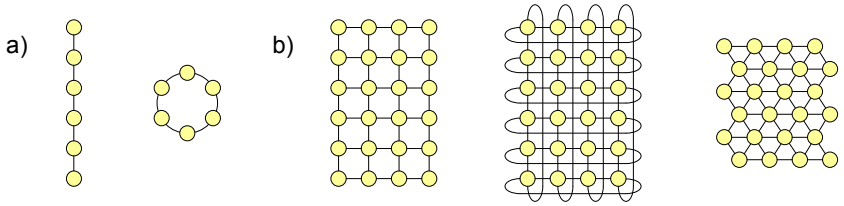

Abbildung 2.5.: Einige Beispiele für Topologien der SOM-Ausgabeschicht a) eindimensional: Neuronenkette, Ring b) zweidimensional: quadratisches Gitter, Torus, hexagonales Gitter (nach [BHS07])

Kommt es zu dem Verlust eines Körperteils, wird die entsprechende Region auf dem Kortex von anderen Bereichen übernommen. Die Areale von häufig stimulierter Peripherie wachsen an, vernachlässigte Bereiche verkleinern sich. Dieses Phänomen wird als *kortikale Plastizität* bezeichnet.

Kohonennetze (benannt nach ihrem Schöpfer Teuvo Kohonen) versuchen, diese Vorgänge nachzuahmen, indem hochdimensionale Daten topologieerhaltend auf wenige Dimensionen abgebildet werden. So sollen sich ähnliche Eingabedaten benachbart auf einer Karte wiederfinden. Oft angebotene Datenvektoren sollen einen größeren Bereich auf dieser Karte einnehmen. Das Training erfolgt unüberwacht, deshalb werden Kohonennetze auch *selbstorganisierende Karten* (*self-organizing maps, SOM*) genannt.

2.2.1. Aufbau von Kohonennetzen

Kohonenetze bestehen aus einer Eingabe- und einer Ausgabeschicht. Dabei sind die Neuronen der Eingabeschicht vollständig mit denen der Ausgabeschicht vernetzt, d.h. jedes Eingabeneuron i ist über ein Gewicht w_{ij} mit jedem Ausgabeneuron j verbunden. Resultierend wird der Gewichtsvektor (oder auch *Prototypvektor*) eines Ausgabeneurons j mit $\vec{w}_j$ bezeichnet. Die Anzahl der Eingabeneuronen richtet sich nach der Dimension der verwendeten Eingabedaten.

Es existieren auch laterale Verbindungen unter den Ausgabeneuronen, die lokale Nachbarschaftsverhältnisse beschreiben. Die Vernetzung innerhalb der Ausgabeschicht beschreibt die *Topologie* einer SOM. Innerhalb der Topologie werden den Neuronen Positionen zugeordnet und über die Nachbarschaft Distanzen zwischen ihnen definiert. Der Aufbau der Ausgabeschicht kann vielfältig gestaltet sein, einige Beispiele sind auf

Abbildung 2.5 dargestellt.

Es gibt eindimensionale Topologien (Abb. 2.5a), wie eine Kette oder ein Ring von Neuronen.

Zweidimensionale Topologien (Abb. 2.5b) kommen am häufigsten zum Einsatz, da diese sehr gut darstellbar sind und ein großes Aufgabenfeld bewältigen. Zu ihnen gehören unter anderem das quadratische Gitter, der Torus und das hexagonale Gitter. Das quadratische Gitter ist sehr einfach zu implementieren, die Neuronen sind in Zeilen und Spalten angeordnet, horizontal und vertikal nebeneinander liegende Neuronen sind verbunden. Der Torus basiert auf dieser Topologie, nur dass die Neuronen an den Rändern ebenfalls miteinander verbunden werden. Das hexagonale Gitter verbindet ein Neuron in der Regel mit sechs anderen, was die Ausbildung von runderen Strukturen ermöglichen soll.

Dreidimensionale Ausgabeschichten sind ebenfalls denkbar, z.B. können Neuronen in einem Quader angeordnet sein. Es sind fast beliebige Strukturen als Topologie denkbar, problematisch wäre da nur die Darstellung.

Kohonen empfiehlt in [Koh01] die Nutzung einer hexagonalen Topologie für die visuelle Inspektion, da horizontale und vertikale Richtungen nicht so sehr bevorzugt werden, wie bei rechteckigen Strukturen. Zell ([Zel03]) hingegen geht in seinen Erfahrungen von quadratischen Gittern aus, da diese einfacher zu implementieren sind und sich besser auf Parallelrechnern abbilden lassen.

2.2.2. Erregung einer SOM

Der Vorgang der Erregung eines Kohonennetzes wird in [Koh01] beschrieben. Es wird davon ausgegangen, dass eine eintreffende Nachricht in Form von parallelen Signalen mit allen Modellneuronen der Karte verglichen werden kann, was zu einer Stimulation dieser Neuronen, abhängig vom Grad der Äquivalenz, führt. Es tritt eine Art Wettbewerb in Kraft, in dem das Element mit den passendsten Parametern als *Gewinner* hervorgeht. In einem Gehirn wird beobachtet, dass stimulierte Zellen die Aktivität der anderen Zellen in ihrer Umgebung hemmen, was *laterale Inhibition* genannt wird. Das Gewinnerneuron tritt in Vorschein, wenn es trotz der Unterdrückung durch die Umgebung am meisten erregt ist.

2.2.3. SOM Training

Das Training von Kohonennetzen erfolgt unüberwacht. Im Anwendungsbereich dieser speziellen KNN ist es im Voraus weder möglich, einen gewünschten Zielzustand vorzugeben, noch die Ausgabe auf Korrektheit zu überprüfen. Wie bei den meisten neuronalen Netzen werden für das Lernen auch hier die Verbindungsgewichte angepasst. Die zu diesem Zweck eingesetzte Lernregel wird durch folgend beschriebenen Algorithmus definiert.

Die Trainingsdaten sind in sogenannten *Eingabevektoren* vorhanden, für die $\vec{x} \in \Re^n$ gilt. Ein solcher Eingabevektor wird parallel mit allen Gewichtsvektoren $\vec{w}_j \in \Re^n$ der Karte verglichen. Zu diesem Zweck wird eine Distanzmetrik (vgl. nächster Abschnitt) eingesetzt. Das Neuron c, dessen Gewichtsvektor $\vec{w}_c$ dem Eingabevektor $\vec{x}$ am ähnlichsten ist, wird *Gewinnerneuron* genannt. Es gilt die Formel 2.5. Dieses Verfahren ist eine vereinfachte Vorgehensweise, die laterale Inhibition wird dabei nicht berücksichtigt.

$$c = \arg\min_j \left(\left\| \vec{x} - \vec{w}_j \right\| \right) \tag{2.5}$$

Während des Lernvorgangs werden nicht nur die Gewichte des Gewinners angepasst. Die Gewichtsvektoren der Neuronen in seiner topologischen Nachbarschaft werden ebenfalls verändert, wenn auch in geringerem Umfang. Welche Nachbarn mit angepasst werden, bestimmt der *Lernradius* $r \in \Re$. Dieser kann sich über die Zeit verändern: In der Regel wird das Training mit einem großen Radius begonnen, der alle Neuronen der SOM umfasst. Anschließend wird r mit dem Fortschreiten des Lernens immer mehr reduziert. So wird erreicht, dass die Karte zum Ende des Trainings keinen großflächigen Veränderungen mehr unterliegt, sondern nur noch im Detail angepasst wird. Den Grad der Beeinflussung der Neuronengewichte innerhalb von r bestimmen *Nachbarschaftsfunktionen* f_{nb}, die folgend ab Seite 16 erläutert werden. Der Lernprozess wird formell durch Gleichung 2.6 beschrieben.

$$\vec{w}_j(t+1) = \vec{w}_j(t) + \eta(t) \cdot f_{nb}\left(d_{cj}, r(t)\right) \cdot \left(\vec{x} - \vec{w}_j(t)\right) \tag{2.6}$$

Der Wert $d_{cj} \in \Re$ repräsentiert die topologische Distanz des Neurons j zum Gewinnerneuron c, er wird für die Berechnung der Nachbarschaftsfunktion f_{nb} benötigt. Deren zweiter Parameter $r(t)$ steht für den Lernradius zum Zeitpunkt t.

Die Gewichte werden derart verändert, dass sie in Richtung des Eingabevektors verschoben werden. Dieser Vorgang wird mit der Abbildung 2.6 erläutert. Es ist eine eindimensionale Topologie (eine Neuronenkette) dargestellt, die Gewichtsvektoren $\vec{w}_j$ und der Eingabevektor $\vec{x}$ werden im zweidimensionalen Eingaberaum abgebildet. Das Gewinnerneuron ist 5, da es zum Zeitpunkt t dem Vektor $\vec{x}$ am nächsten ist. In seiner topologischen Nachbarschaft befinden sich die Neuronen 3, 4, 6 und 7 innerhalb des Lernradius. Die Gewichtsvektoren aller dieser Neuronen werden zum Zeitpunkt $t+1$ in Richtung von $\vec{x}$ verschoben (graue Pfeile). Man beachte, dass Neuron 8 nicht verschoben wird, obwohl es sich näher an dem Eingabevektor befindet, als beispielsweise Neuron 7. Es wird nur das Gewinnerneuron und seine Nachbarschaft angepasst. Die Funktionsweise einer Nachbarschaftsfunktion ist ebenfalls gut zu erkennen: Das Gewinnerneuron wird dem Eingabevektor verhältnismäßig am stärksten angenähert, die Anpassung der Nachbarn nimmt mit höherer topologischer Distanz zum Gewinner ab.

Über die *Lernrate* $\eta \in \Re$ wird bestimmt, wie stark diese Anpassung durchgeführt wird. Für sie gilt $0 \leq \eta \leq 1$, sie ist zeitlich variabel. Die Lernrate wird oft ebenfalls während des Trainingsverlaufs reduziert, um gegen Ende eine Stagnation der Karte zu erreichen. Hat sie den Wert 0, findet keine Veränderung statt, bei 1 nimmt das Gewinnerneuron die Werte des Eingabevektors vollständig in sich auf.

Der gesamte Trainingsablauf wird zyklisch wiederholt. Bei jedem Lernschritt wird ein neuer Eingabevektor $\vec{x}$ als Trainingsdatum eingesetzt.

Distanzmetriken

Zur Bestimmung des Abstands eines Datenvektors zu einem Gewichtsvektor sind beliebige Metriken einsetzbar, die der Gleichung 2.7 entsprechen. Denkbar für diese Aufgabe sind z.B. das Skalarprodukt, die Euklidische Distanz und die Manhattan-Distanz.

$$\|\vec{x}-\vec{y}\| \mapsto \Re;\ \vec{x},\vec{y} \in \Re^n \tag{2.7}$$

Das Skalarprodukt (Formel 2.8) ist nur auf normierte Vektoren anwendbar. Eine weitere Besonderheit dieser Metrik ist, dass der Gewichtsvektor mit maximalem Skalarprodukt das Gewinnerneuron ausmacht.

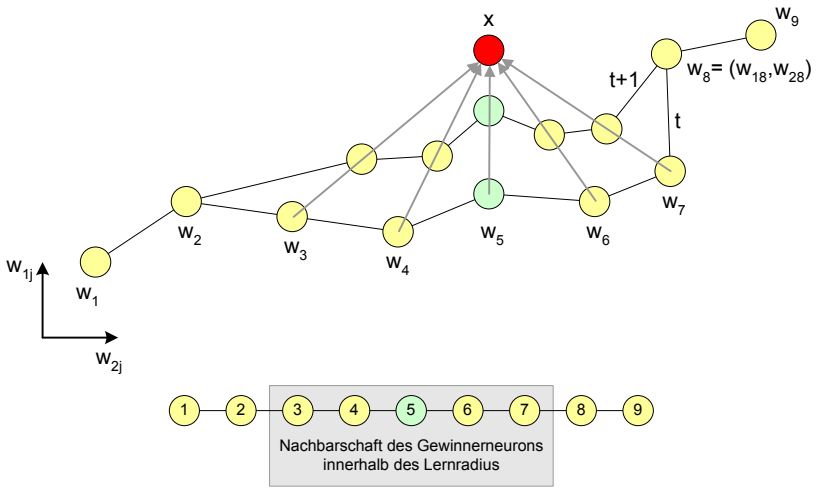

Abbildung 2.6.: Der Lernprozess von SOM (nach [Zel03]): Die Nachbarn des Gewinnerneurons 5 innerhalb des Lernradius werden ebenfalls an den Eingabevektor x angepasst

$$\|\vec{x}-\vec{y}\|_{skalar} = \sum_{i=1}^{n} x_i y_i \tag{2.8}$$

Die Euklidische Distanz ist durch die Gleichnung 2.9 definiert. Sie ist die bevorzugte Metrik zur Distanzberechnung.

$$\|\vec{x}-\vec{y}\|_{euklid} = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2} \tag{2.9}$$

Die Manhattan-Distanz ist definiert durch die Summe der absoluten Differenzen für jede Dimension der Vektoren. Gleichung 2.10 zeigt diese Metrik.

$$\|\vec{x}-\vec{y}\|_{manhattan} = \sum_{i=1}^{n} |x_i - y_i| \tag{2.10}$$

Distanzmetriken werden auch eingesetzt, um die Güte einer SOM zu bestimmen. Zu diesem Zweck wird der *durchschnittliche Quantisierungsfehler* eingesetzt. Der Quantisierungsfehler wird bestimmt durch die Distanz eines Eingabevektors $\vec{x}_i$ zum Gewichtsvektor des entsprechenden Gewinnerneurons $\vec{w}_{ci}$. Formel 2.11 verdeutlicht dies. Der durchschnittliche Quantisierungsfehler $\bar{e} \in \Re$ bildet den Durchschnitt der Fehler über

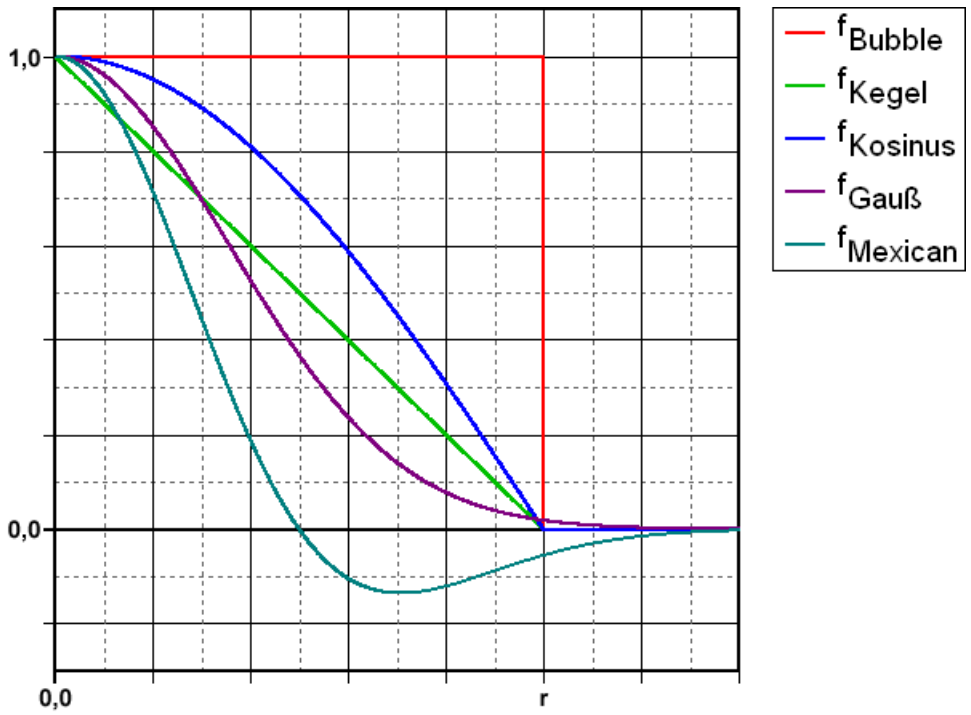

Abbildung 2.7.: Die Nachbarschaftsfunktionen (erstellt mit Easy Funktion)

alle Eingabevektoren, wie in Gleichung 2.12 dargestellt. Der Wert p steht dabei für die Anzahl aller Eingabevektoren.

$$e_i = \|\vec{x}_i - \vec{w}_{ci}\| \tag{2.11}$$

$$\overline{e} = \frac{1}{p}\sum_{i=1}^{p} \|\vec{x}_i - \vec{w}_{ci}\| \tag{2.12}$$

Nachbarschaftsfunktionen

Die *Nachbarschaftsfunktionen* bestimmen, inwiefern die Nachbarneuronen innerhalb des Lernradius an das trainierte Datum angeglichen werden, d.h. wie weit die Neuronengewichte an die an den Eingabeneuronen angelegten Werte angenähert werden. Zu diesem Zweck werden Funktionen eingesetzt, die von zwei Parametern abhängig sind: dem Lernradius r und dem topologischen Abstand des Gewinnerneurons zum Nachbarneuron d_{cj}. Formel 2.13 verdeutlicht dies.

$$f_{nb}(d_{cj}, r) \mapsto \Re;\ d_{cj}, r \in \Re \tag{2.13}$$

Die durch f_{nb} errechnete reelle Zahl bestimmt den Anpassungsgrad des jeweiligen Neurons. Ein Wert von 0 bedeutet keine Veränderung an den Neuronengewichten. Ein Wert von 1 legt fest, dass die Eingabewerte vollständig übernommen werden. Ein negativer Wert veranlasst das Neuron, sich weiter von den Eingabewerten zu distanzieren. Für gewöhnlich liegt der Wertebereich von f_{nb} in $[0,1]$ oder $[-1,1]$. Die Parameter r und d_{cj} sind von ihrer Definition her positiv.

Die hier verwendeten Nachbarschaftsfunktionen sind [Zel03] nachempfunden und in Abbildung 2.7 dargestellt. Sie werden folgend beschrieben.

Die Zylinderfunktion

Die Zylinderfunktion (zu sehen in Formel 2.14) wird auch Bubblefunktion genannt und ist die Einfachste der Nachbarschaftsfunktionen. Allen Neuronen innerhalb des Lernradius wird eine 1 zugewiesen, so wird erreicht, dass alle betroffenen Nachbarn ebenso wie das Gewinnerneuron angepasst werden. Alle anderen Neuronen bleiben unverändert. Dem Gewinnerneuron kommt also keine Sonderbehandlung zu.

$$f_{Bubble}(d_{cj}, r) = \begin{cases} 1 & (d_{cj} \leq r) \\ 0 & (d_{cj} > r) \end{cases} \tag{2.14}$$

Die Kegelfunktion

Die Kegelfunktion bietet eine einfache Möglichkeit, die Anpassung des Gewinnerneurons hervorzuheben und die der Nachbarn abzuschwächen. Dabei ist die Annäherung der Gewichte an die Eingabewerte entgegengesetzt proportional zum Abstand zum Gewinnerneuron. Die Formel 2.15 zeigt die Funktion.

$$f_{Kegel}(d_{cj}, r) = \begin{cases} 1 - \frac{d_{cj}}{r} & (d_{cj} \leq r) \\ 0 & (d_{cj} > r) \end{cases} \tag{2.15}$$

Die Kosinusfunktion

Die Kosinusfunktion wird so definiert, wie in der Formel 2.16 dargestellt.

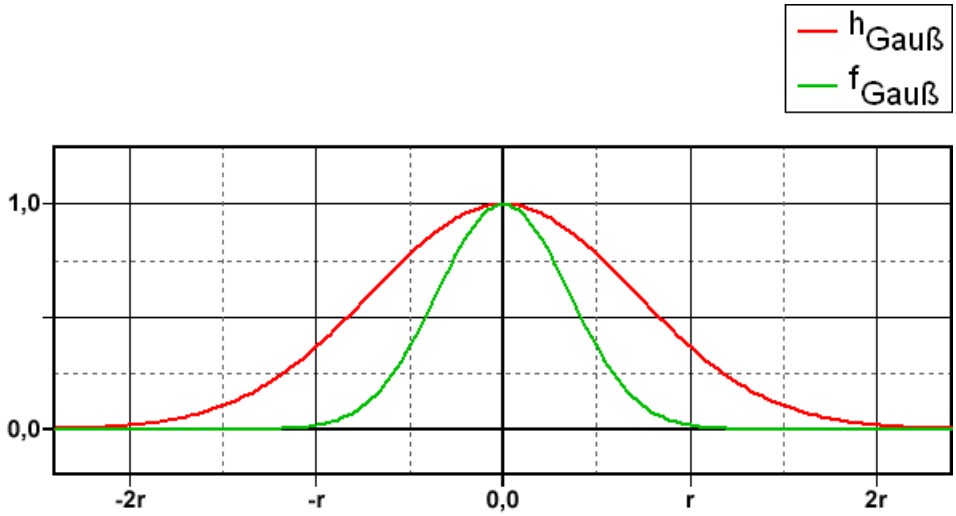

Abbildung 2.8.: Die in [Zel03] definierte Gaußfunktion im Vergleich mit der gestauchten Variante (erstellt mit Easy Funktion)

$$f_{Kosinus}(d_{cj},r) = \begin{cases} \cos\left(\frac{\pi \cdot d_{cj}}{2r}\right) & (d_{cj} \leq r) \\ 0 & (d_{cj} > r) \end{cases} \tag{2.16}$$

Die Gaußfunktion

Die Gaußfunktion (auch Gaußglocke genannt) ist nach [Zel03] definiert durch die Gleichung $h_{Gauss}(d_{cj},r) = e^{-(d_{cj}/r)^2}$. Bei dem Funktionsgraphen (s. Abb. 2.8) zeigt sich, dass auch die Neuronen, die sich deutlich außerhalb des Lernradius (im Bereich $r < d_{cj} \leq 2r$) befinden, noch maßgeblich beeinflusst werden. Der Meinung des Autors nach sollten nur die Neuronen innerhalb von r angepasst werden, da sich diese Nachbarschaftsfunktion dann besser mit den Anderen vergleichen lässt. Daher wird diese Gaußfunktion im Definitionsbereich gestaucht, indem d_{cj} mit 2 multipliziert wird. Die resultierende Funktion zeigt Gleichung 2.17, sie ist ebenfalls auf Abbildung 2.8 dargestellt.

$$f_{Gauss}(d_{cj},r) = e^{-\left(\frac{2d_{cj}}{r}\right)^2} \tag{2.17}$$

Die „mexican hat“-Funktion

Die sogenannte *mexican hat*-Funktion wird in Anlehnung an einen natürlichen Prozess eingeführt. Es geht darum, dass die Nachbarneuronen in unmittelbarer Nähe des Gewinners ebenfalls stark erregt, die Neuronen in weiterer Entfernung aber für den angebotenen Reiz unterdrückt werden.

Die Funktion wird in [Zel03] als ähnlich der Gleichung 2.18 beschrieben. Auch auf sie trifft es zu, dass Neuronen außerhalb des Lernradius stark beeinflusst werden, deshalb wird diese Funktion auf gleiche Weise gestaucht.

$$h_{Gauss2}(d_{cj}, r) = \left(1 - \left(\frac{d_{cj}}{r}\right)^2\right) \cdot e^{-\left(\frac{d_{cj}}{r}\right)^2} \tag{2.18}$$

Das Ergebnis zeigt Gleichung 2.19. Da aber auch diese Funktion der *mexican hat*-Funktion nur ähnelt, gilt ab sofort folgender Sachverhalt: Immer wenn von der *mexican hat*-Funktion die Rede ist, ist $f_{Mexican}$ gemeint.

$$f_{Mexican}(d_{cj}, r) = \left(1 - \left(\frac{2d_{cj}}{r}\right)^2\right) \cdot e^{-\left(\frac{2d_{cj}}{r}\right)^2} \tag{2.19}$$

Nach [Zel03, S. 183] eignen sich derartige Funktionen mit teilweise negativen Wertebereichen nicht für die Anwendung bei nicht normierten Kohonennetzen. Anstatt einer schnelleren Entfaltung der Karte kommt es zu einem Zustand, in dem sich die Neuronen nur noch abstoßen. Das Netz divergiert, man spricht dabei von einer Explosion der Karte.

2.3. Visualisierungsformen selbstorganisierender Karten

Es existiert eine Vielzahl an Darstellungsformen für Kohonennetze. Dabei wird auf unterschiedliche Aspekte der Karte eingegangen. Manche Visualisierungen sind nicht nur von der SOM selbst, sondern auch von den Trainingsmustern oder den Lernparametern abhängig. Nach diesen Ansichtspunkten lassen sich die Visualisierungsarten klassifizieren, was in Tabelle 2.1 geschieht.

Visualisierung	abh. von Mustern	abh. von Lernparametern
Distanzmatrix	1	-
Gewinnermatrix	1	r, f_{nb}
Gewinnhistogramm	alle	-
Kalibrierte Karte	alle	-
Karte im Eingaberaum	alle	-
Komponentenmatrix	-	-
L-Matrix	-	-
P-Matrix	alle	-
U-Matrix	-	-
U*-Matrix	alle	-

Tabelle 2.1.: Einige Visualisierungsarten von SOM und deren Abhängigkeit zu Eingabemustern und Lernparametern: -: unabhängig, 1: abhängig von einem Muster, alle: abhängig von allen Mustern

Einen Überblick über Visualisierungsmethoden ist in [Ves99] gegeben. Einige interessantere Exemplare werden im Anschluss kurz erläutert.

2.3.1. Karte im Eingaberaum

Bei dieser Visualisierung wird die SOM im meist zwei- oder dreidimensionalen Eingaberaum dargestellt. Die Gewichtsvektoren der Karte werden hierbei als Position im Eingaberaum interpretiert. Die durch die Topologie bestimmten Nachbarschaftsverhältnisse werden als Kanten zwischen den Positionen der Neuronen dargestellt.

Es besteht die Möglichkeit, auch die trainierten Eingabedaten in diesem Raum abzubilden (z.B. als Punktwolke). So lässt sich erkennen, wie sich die Karte im Verlauf des Lernprozesses an diese Punkte annähert.

2.3.2. U-Matrix

Die U-Matrix (von *unified distance matrix*, [VA00]) ist eine sehr häufig eingesetzte, distanzbasierte Methode, wenn *Cluster*, also Gruppen ähnlicher Elemente, in hochdi-

mensionalen Daten gefunden werden sollen. Zu diesem Zweck werden die lokalen Distanzen für jedes Neuron bestimmt und auf einer zweidimensionalen Karte farblich abgebildet. Auf dieser Karte entstehen, den Distanzen entsprechend, Berge und Täler. Die Täler markieren Cluster, Berge repräsentieren Clustergrenzen.

Dieses Verfahren setzt folgend beschriebenen Algorithmus ein: Für jedes Neuron i wird die Menge der direkten Nachbarn M in der Topologie bestimmt. Anschließend werden die Distanzen zwischen dem Gewichtsvektor von i und den Gewichtsvektoren der Neuronen m aus M berechnet und aufsummiert. Formell wird die Berechnung durch die Gleichung 2.20 dargestellt.

$$u_i = \sum_{m \in M} \|\vec{w}_i - \vec{w}_m\| \tag{2.20}$$

Das Ergebnis u_i wird anschließend nach dem höchsten aufgetretenen Wert normiert und üblicherweise in einen Grauwert überführt: Hohen Werten werden dunkle, niedrigen Werten helle Grautöne zugeordnet (vgl. [UM05]). Die Farbtöne werden auf einer Matrix entsprechend der SOM-Topologie den Neuronen zugeordnet. Es entsteht eine Karte, auf der Täler hell und Berge dunkel erscheinen. Die hellen Bereiche zeigen an, dass an dieser Stelle auf der Karte die Distanzen zwischen den benachbarten Neuronen sehr niedrig sind, es ist also sehr wahrscheinlich, dass sich dort ein Cluster befindet. Da sich die Neuronen in den dunklen Bereichen sehr unterscheiden, kann von einer Clustergrenze ausgegangen werden.

2.3.3. L-Matrix

Die L-Matrix (*length-of-vector matrix*) wurde vom Autor im Rahmen dieser Arbeit entwickelt. Es handelt sich hierbei um eine einfache Visualisierung, die die Länge der Gewichtsvektoren von Neuronen darstellt.

Gleichung 2.21 zeigt die Berechnung der L-Matrix: Für jedes Neuron i wird die Länge seines Gewichtsvektors $\vec{w}_i \in \Re^n$ bestimmt. Hierbei kommt eine Distanzmetrik zum Einsatz, indem der Abstand von $\vec{w}_i$ zum Nullvektor $\vec{o} \in \Re^n$ berechnet wird.

$$l_i = \|\vec{w}_i - \vec{o}\| \tag{2.21}$$

Anschließend wird auf den Bereich $[0,1]$ normiert. Die resultierenden Werte werden

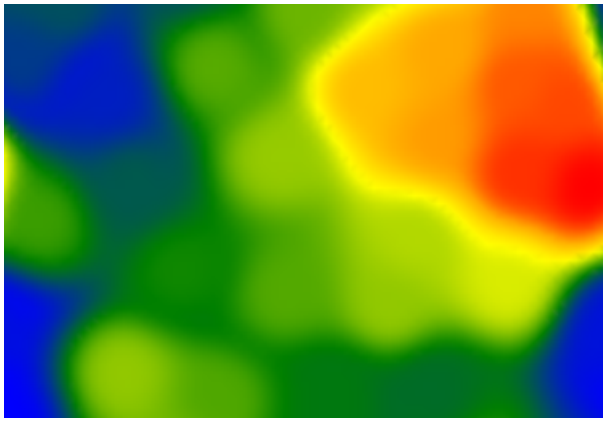

Abbildung 2.9.: Eine mit der L-Matrix visualisierte SOM

auf einem Farbverlauf abgebildet. Das Ergebnis ist in Abbildung 2.9 zu sehen. Homogene Bereiche auf der Karte weisen auf einen Cluster hin, Farbveränderungen sind an Clustergrenzen zu sehen. Diese Visualisierungsform kommt einer geographischen Karte sehr nahe. Sie liefert einen Überblick über die auf der SOM abgebildeten Strukturen.

2.3.4. P-Matrix

Die P-Matrix ist ein von Ultsch in [Ult03a, UM05] vorgeschlagenes Verfahren zur Visualisierung der Datendichte für die Clusteranalyse. Die Datendichte wird an der durch den Gewichtsvektor eines jeden Neurons bestimmten Position im Datenraum gemessen. Diese Visualisierung ist demnach abhängig von allen Mustern. Bereiche hoher Dichte sind hierbei Cluster, Bereiche niedriger Dichte zeigen Clustergrenzen.

Es wird eine Hyperkugel um die Gewichtsvektoren gebildet. Die Eingabemuster, die sich innerhalb dieser Kugel befinden, werden gezählt. Diese Anzahl wird als Höhe auf der visualisierten Karte dargestellt.

Die Schwierigkeit besteht darin, einen global optimalen Radius für diese Hyperkugel zu bestimmen, um ein ausgeglichenes Abbild zu erzeugen. Dieser Radius ist so zu wählen, dass möglichst alle eventuell in der Datenmenge vorhandenen Cluster in ihrer Mitte eine hohe und an ihren Grenzen eine niedrige Dichte aufweisen, um sich somit optimal voneinander zu unterscheiden. Dieses Ziel soll mit einem einzigen, für alle Elemente geltenden Radius erreicht werden.

Das hierzu eingeführte Verfahren verwendet einen im informationstheoretischen Sin-

ne optimalen Ansatz für die Bestimmung des Radius. Ultsch sieht diesen Ansatz als eine Bestätigung für die *Pareto-80/20 Regel* [Ult01] an und hat daher die Methode *Pareto Density Estimation* (*PDE*) genannt. Sie ist in [Ult03b, Ult04] beschrieben.

2.3.5. U*-Matrix

Die U*-Matrix [Ult03c] stellt die Kombination von U-Matrix und P-Matrix dar. Es werden sowohl die globale Datendichte als auch lokale Distanzen einbezogen. Dabei bekommen die Distanzen innerhalb eines Bereichs hoher Dichte (also innerhalb eines Clusters) weniger Gewicht, als die Distanzen an Clustergrenzen.

So werden die Vorteile beider Methoden zusammengeführt und aufeinander abgestimmt, was sehr gute Bedingungen für die Clusteranalyse liefert.

2.3.6. Kalibrierung der Karte

Die Kalibrierung (beschrieben in [Koh01, S. 114]) dient der Lokalisierung der Trainingsdaten auf einer Karte. Zu diesem Zweck wird für jeden Datenvektor das entsprechende Gewinnerneuron bestimmt. Da der Gewichtsvektor des Gewinnerneurons dem angebotenen Eingabevektor am ähnlichsten ist, wird davon ausgegangen, dass dieses Neuron den Datenvektor repräsentiert: Die Dateninstanz „befindet sich“ an dieser Position auf der Karte.

Viele Trainingsdatensätze verfügen über Attribute, die den Instanzen eine Bezeichnung zuordnen, wie z.B. einen Namen. Wenn ein solches Attribut vorhanden und bekannt ist, kann es dazu genutzt werden, die Karte an den bestimmten Positionen zu beschriften.

3. Konzeption

> „The existence of all these visualization methods does little good without a good implementation in a flexible and user-friendly data mining environment.“
>
> (Juha Vesanto, [Ves99])

In diesem Kapitel wird die Konzeptionsphase der Zielapplikation beschrieben. Besondere Eigenschaften, die das Programm erfüllen soll, werden beleuchtet. Es wird versucht, auftretende komplexe Situationen zu identifizieren und daraus resultierende Probleme im Vorfeld zu lösen. Es sollen möglichst viele Klassen unter Zuhilfenahme von UML entworfen werden, um den Implementierungsaufwand zu minimieren.

3.1. Anforderungen an die Applikation

Das Programm soll eine Reihe von Eigenschaften erfüllen, um die in dieser Arbeit beschriebenen Vorgänge durchzuführen und darzustellen (siehe Kapitel 1.1). Es werden noch weitere Anforderungen gestellt, um die Wiederverwendbarkeit und Erweiterungsfähigkeit der Applikation zu gewährleisten.

Die Anwendung hat zur Aufgabe, viele aufwändige Rechenprozesse zu verwalten. Das Training der Merkmalskarte ist beispielsweise ein solcher Prozess, der viel Rechenzeit in Anspruch nimmt. Bei serieller Abarbeitung dieser Aufgabe wird die gesamte verfügbare Leistung des Rechners beansprucht, was zur Folge hat, dass andere Aufgaben nicht mehr bearbeitet werden. So kann z.B. die grafische Benutzeroberfläche der Applikation nicht mehr auf Eingaben reagieren, bis der gesamte Trainingsvorgang beendet ist. Ein solcher Zustand ist unerwünscht, daher müssen Rechenprozesse und die Nutzeroberfläche nebenläufig organisiert werden. Hierzu werden sie in eigene *Threads* ausgelagert.

Bei einem Thread (zu Deutsch eigentlich „Faden“) handelt es sich nach [Ull06, S. 493ff] um einen Ausführungsstrang von Programmcode. Jede Anwendung verfügt über mindestens einen von ihnen, beim Einsatz von mehreren Threads werden die enthaltenen Aktivitäten quasi parallel ausgeführt. Hierbei ist ein besonderes Augenmerk auf die Verwaltung von Ressourcen zu legen, die von mehreren Threads gleichzeitig bearbeitet werden, da es sonst zu inkonsistenten Zuständen, Race-Bedingungen oder Deadlocks kommen kann.

Das `.NET` Framework von Microsoft bietet für diese Art von Aufgaben ein umfangreiches Threading-Konzept. Es ist beispielsweise möglich, gemeinsam genutzte Ressourcen für die exklusive Nutzung eines Threads temporär zu sperren. Für die Kommunikation zwischen verschiedenen Threads wird unter anderem ein ausgefeiltes Ereignissystem zur Verfügung gestellt.

Ein solches Ereignis (auch *Event* genannt) tritt auf, wenn ein Thread einen bestimmten Zustand annimmt. Das kann z.B. im Trainingsprozess geschehen, nachdem ein Lernschritt erfolgreich abgearbeitet wurde. Andere Threads, die an diesem Ereignis interessiert sind, können darauf mit einer bestimmten Aktion reagieren. Um über ein Ereignis informiert zu werden, muss ein Thread einen *Event Handler*, eine bestimmte Methode, die die durch den Event vorgeschriebene Form einhält, zur Verfügung stellen und anmelden. Dieses Vorgehen ähnelt in gewisser Weise dem Entwurfsmuster „Observer“.

3.2. Die Programmarchitektur

Die Architektur des Programms (zu sehen in Abbildung 3.1) baut auf der klassischen *3-Tier-Architecture* auf und erweitert diese um eine weitere Schicht, die die Erweiterungsfähigkeit gewährleisten soll.

Die Idee hinter dieser Architektur besteht darin, dass alle Komponenten des Programms ihrer Aufgabe entsprechend in drei Schichten eingeordnet werden: Dem Datenzugriff, der Geschäftslogik und der Präsentation. Jede Schicht kapselt ihren Aufgabenbereich derart, dass sie als Ganzes durch eine andere Implementierung ersetzbar ist, was eine gewisse Modularität voraussetzt. So kann zum Beispiel eine Präsentationsschicht, die eine Windows-Oberfläche implementiert, komplett durch ein Web-Frontend ersetzt werden, ohne dass Anpassungen in den anderen Schichten notwendig sind. Es bestehen Richtlinien, die den Zugriff von einer Schicht auf eine andere betreffen (dargestellt

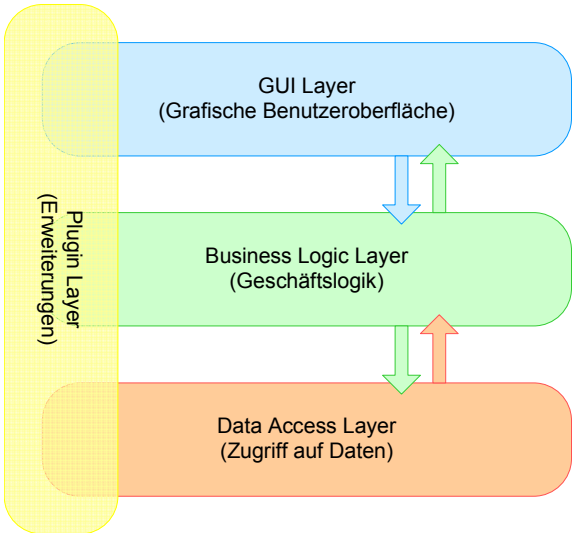

Abbildung 3.1.: Die vier Schichten der Programmarchitektur

durch die Pfeile in der Abbildung). Ein direkter Zugriff der Präsentationsschicht auf die Datenzugriffsschicht ist beispielsweise verboten. Die dort enthaltenen Daten sind nur indirekt über die Geschäftslogik verfügbar.

3.2.1. Datenzugriff - Die Data Access Layer

Die Data Access Layer erlaubt den Zugriff auf externe Ressourcen. Sie liefert die Daten, die das Programm verarbeitet oder stellt den Zugriff auf Peripherie, wie etwa Drucker, zur Verfügung.

Wie die gelieferten Daten gehalten werden, ist eine Frage der Implementierung. Es ist möglich, die Informationen zum Beispiel aus Dateien oder Datenbanken zu erhalten. Die Art der Datenhaltung ist den anderen Schichten unbekannt, deshalb werden die Informationen gewöhnlich über sogenannte *Value Objects*, also Objekte, die nur die Daten ohne jegliche Strukturinformationen oder Verarbeitungsmethoden enthalten, an andere Schichten übertragen. Die Daten werden also unverarbeitet an die Geschäftslogikschicht weitergegeben.

3.2.2. Geschäftslogik - Die Business Logic Layer

Die Business Logic Layer steuert die gesamte Datenverarbeitung. Sie empfängt die Daten aus der Data Access Layer, verarbeitet sie und gibt sie aufbereitet an die GUI Layer weiter.

Aus den Value Objects der Datenzugriffsschicht werden Strukturen erzeugt, die Geschäftsprozesse abbilden. Die Verarbeitung dieser Strukturen (Berechnungen etc.) findet ausschließlich hier statt. Für gewöhnlich werden die Value Objects der Data Access Layer nicht ohne voherige Bearbeitung oder Umgestaltung an die GUI Layer übertragen.

3.2.3. Präsentation und Nutzerinteraktion - Die GUI Layer

Die *GUI* Layer (Graphical User Interface - grafische Nutzerschnittstelle) erfüllt gleich zwei Aufgaben: Sie präsentiert die Daten und führt den Dialog mit dem Benutzer.

Die Datenstrukturen aus der Business Logic Layer werden interpretiert und dargestellt. Die Nutzereingaben werden zur Anpassung der Daten genutzt. Die Verarbeitung findet aber weiterhin in der Geschäftslogik statt.

3.2.4. Programmerweiterungen - Die Plugin Layer

Die Plugin-Layer kapselt Teile der anderen Schichten in sich. Dabei handelt es sich hauptsächlich um Schnittstellen, die alle austauschbaren Komponenten der Hauptapplikation beschreiben. Diese Schicht soll als eigenständiges Modul implementiert werden, das sowohl von der Applikation als auch von den Erweiterungsmodulen eingebunden wird. So stellt es die Kommunikationsbasis zu externen Programmerweiterungen dar.

Bei einem *Plugin* handelt es sich also um eine programmexterne Komponente, die bei der Hauptapplikation angemeldet und anschließend dynamisch nachgeladen wird. Sie erfüllt einen von der Applikation vorgeschriebenen Zweck. Nachfolgend werden alle vom Programm erlaubten Plugin-Typen aufgelistet und kurz beschrieben. Auf die genaue Funktionsweise und Struktur dieser Elemente wird in den folgenden Abschnitten eingegangen.

Die Datenzugriffsschicht erlaubt die Integration dieser Komponenten:

Datenquellen liefern die Daten, die für das Training der Karte genutzt werden. Die Struktur der Daten ist in der Schicht vorgegeben. Woher die Daten stammen wird in der Implementierung bestimmt.

Normierer wandeln die gelieferten Daten aus der Quelle in reelle Zahlenwerte um, damit das Netz diese überhaupt verarbeiten kann. Diese Vorverarbeitung ist auch notwendig zur Bestimmung der Anzahl der Eingabeneuronen.

Der Geschäftslogikschicht werden folgende Plugins zugeteilt:

Distanzmetriken bestimmen, wie die Distanz zwischen Realzahlvektoren errechnet wird. Das ist für die Berechnung des Quantisierungsfehlers oder des Abstands der Gewichte zweier Neuronen notwendig.

Netztopologien bestimmen den internen Aufbau der SOM. Die Nachbarschaftsverhältnisse der Kartenneuronen werden hier definiert.

Netzinitialisierer bestimmen, wie die Gewichte der selbstorganisierenden Karte vor dem Beginn des Trainings initialisiert werden.

Nachbarschaftsfunktionen sind ein Teil der beim Training eingesetzten Lernparameter. Sie bestimmen, wie stark die Nachbarneuronen des Gewinners beim Lernprozess beeinflusst werden.

In der Präsentationsschicht gibt es nur einen Plugin-Typ:

Visualisierungen stellen den aktuellen Zustand der SOM dar. Auf welche Eigenschaften dabei eingegangen wird, ist von der Art der Visualisierung abhängig.

3.3. Aufbau der SOM

Eine selbstorganisierende Karte hat viele Eigenschaften, die im Programm abgebildet werden sollen. Die Neuronen und ihre Gewichte bilden das Netz, eine Topologie bestimmt die Nachbarschaftsverhältnisse zwischen den Neuronen. Eine Distanzmetrik definiert die Berechnung der Abstände zwischen Neuronen (bzw. deren Gewichten).

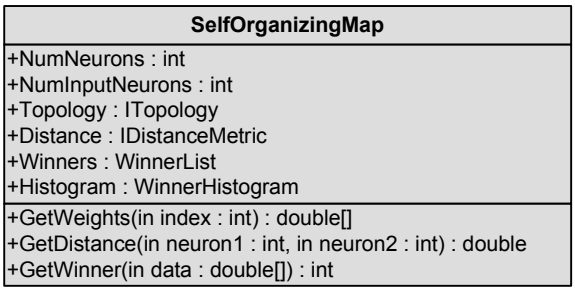

Abbildung 3.2.: Das UML-Diagramm einer selbstorganisierenden Karte

Es sollen noch andere Eigenschaften ergänzend mit in die Struktur einfließen: Eine Gewinnerliste enthält die Gewinnerneuronen für jeden Datensatz aus der Datenquelle und den entsprechenden Quantisierungsfehler. Ein kumuliertes Gewinnhistogramm soll darstellen, wie oft welches Neuron der Karte im Trainingsprozess ein Gewinnerneuron war.

Die Distanzmetrik und die Netztopologie sind austauschbar und werden über Plugins realisiert, die Gewinnerliste und das Gewinnhistogramm sind feste Bestandteile, die unabhängig von den austauschbaren Elementen funktionieren.

Die SOM selbst soll zusammenfassend folgende Informationen über sich preisgeben und Funktionalitäten zur Verfügung stellen:

- Die Anzahl enthaltener Neuronen
- Einen Vektor aus reellen Zahlen für jedes Neuron, der den Zustand der Gewichte repräsentiert
- Die Anzahl der Eingabeneuronen
- Die verwendete Netztopologie
- Die verwendete Distanzmetrik, sowie die Funktionalität zur Berechnung der Distanz der Gewichtsvektoren zweier Neuronen
- Die Gewinnerliste und das kumulierte Gewinnhistogramm
- Die Funktionalität, einem genormten Datensatz ein Gewinnerneuron zuzuordnen

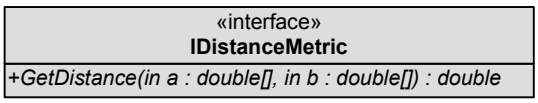

Abbildung 3.3.: Die Distanzmetrik braucht nur eine Methode zu implementieren: die Berechnung des Abstandes zweier Vektoren

Das UML-Diagramm in Abbildung 3.2 zeigt den Aufbau der Klasse, die eine selbstorganisierende Karte repräsentiert. Die benannten Eigenschaften und Methoden erfüllen die oben beschriebenen Anforderungen. Die Neuronen und deren Gewichte sollen über Indizes ansprechbar sein.

3.3.1. Distanzmetrik

Eine der Karte zugeordnete Distanzmetrik definiert, wie der Abstand zwischen den Gewichtsvektoren zweier Neuronen berechnet wird. Diese sollen austauschbar sein und werden deshalb als eigene Schnittstelle (s. Abbildung 3.3) vorgestellt.

Die einzige Methode `GetDistance` liefert der Formel 3.1 entsprechend zu zwei Realzahlvektoren eine Distanz. Diese soll auch zur Bestimmung des Quantisierungsfehlers eingesetzt werden.

$$dist : \Re^n \times \Re^n \mapsto \Re \tag{3.1}$$

3.3.2. Netztopologie

Die Netztopologie ist ein wesentlicher Bestandteil der Karte und soll als Plugin ausgelegt werden. Eine Topologie ist jeder SOM zugeordnet, da sie den internen Aufbau der Karte bestimmt. Sie soll über Nachbarschaftsstrukturen Auskunft geben und sofern möglich den Neuronen eine Position im Raum zuordnen. Die Ausdehnung des Netzes soll ebenfalls ein Bestandteil der Topologie sein.

Die Abbildung 3.4 zeigt die vorgesehene Schnittstelle für den Zweck. Die Methode `GetNeighbors` soll die Indizes der direkten Nachbarn eines Neurons liefern.

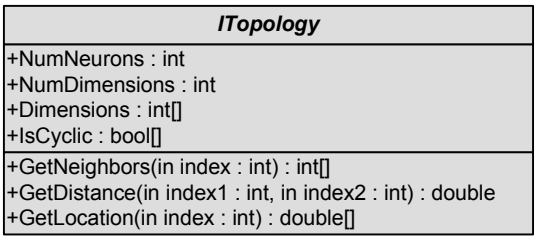

Abbildung 3.4.: Die Anforderungen an eine Netztopologie zusammengefasst in einer Schnittstelle

Mit `GetLocation` soll es möglich sein, einem Neuron eine Position zuzuordnen. Bei einer Matrixtopologie ist das die Position in der Matrix, z.B. `(0,0)` für das Neuron in der linken oberen Ecke. Wie viele Dimensionen diese Position hat, ist nicht vorgegeben, es können also auch ein- oder dreidimensionale Topologien existieren (die Anzahl der verwendeten Dimensionen soll sich über die Eigenschaft `NumDimensions` bestimmen lassen).

Die Ausdehnung der Karte soll in der Eigenschaft `Dimensions` stehen, bei der Matrixtopologie wären das die Breite und die Höhe der Matrix. Die Methode `GetDistance` soll den Abstand zweier Neuronen liefern. Dabei handelt es sich um den Abstand der Positionen der Neuronen in der Topologie, nicht zu verwechseln mit der Distanz zwischen den Gewichten, die in der SOM selbst definiert wird.

Mit der Eigenschaft `NumNeurons` wird die Anzahl der in der Topologie vorhandenen Ausgabeneuronen bekannt gegeben. In `IsCyclic` soll für jede Dimension ein Boolescher Wert stehen, der besagt, ob die Topologie in dieser Dimension zyklisch ist oder nicht. Die Matrixtopologie ist normalerweise nicht zyklisch, d.h. der Wert wäre für beide Dimensionen `false`. Ein Torus wäre zyklisch in beiden Dimensionen, ein Zylinder in nur einer der beiden. Diese Angabe ist beispielsweise auch für Visualisierungen interessant.

3.3.3. Gewinnerliste

Die Gewinnerliste stellt jedem Datensatz aus der Datenquelle ein Gewinnerneuron gegenüber, dessen Gewichte den normierten Daten aus dem Satz am ähnlichsten sind. Zusätzlich soll die jeweilige absolute Distanz zwischen den Gewichten und den Daten (bekannt als Quantisierungsfehler) zur Verfügung gestellt werden.

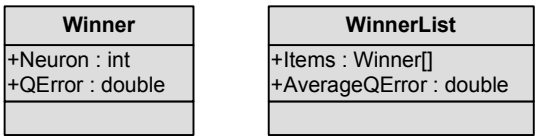

Abbildung 3.5.: Die Gewinnerliste enthält Gewinnerinstanzen und deren durchschnittlichen Quantisierungsfehler

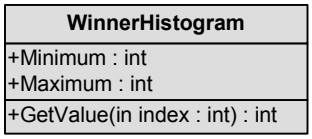

Abbildung 3.6.: Das kumulierte Gewinnhistogramm speichert die Gewinnanzahl aller Neuronen über den gesamten Trainingsverlauf

Die Datenstruktur, die einen solchen Gewinner darstellt, hat also nur zwei Eigenschaften: Den Index des Gewinnerneurons und den Quantisierungsfehler. Die Liste selbst soll die Menge aller Gewinner (z.B. in einem Array) und als zusätzliche Information den durchschnittlichen Quantisierungsfehler enthalten. In der Abbildung 3.5 sind die UML-Diagramme des Gewinners und der Liste zu sehen.

Die Gewinnerliste wird während des Trainings (siehe Kapitel 3.5) erzeugt und aktualisiert. Sie dient nicht nur statistischen Zwecken und der Berechnung des Quantisierungsfehlers, sondern ist auch für Visualisierungen nützlich. So soll über die Zuordnung eines Datensatzes zu einem Neuron bestimmt werden können, an welcher Position auf der Karte sich dieser Datensatz „befindet“ und seine Bezeichnung kann dort entsprechend angezeigt werden.

3.3.4. Kumuliertes Gewinnhistogramm

Das kumulierte Gewinnhistogramm dient hingegen nur der Statistik. Es werden die Gewinnerneuronen gezählt, die während des Trainingsprozesses ausgewählt wurden. Jedem Neuron der Karte wird die Gesamtanzahl seiner Gewinne zugeordnet.

Abbildung 3.6 zeigt die Gewinnhistogramm-Klasse. Jedem Neuron wird eine Zahl zugeordnet, diese kann über den Index des Neurons erreicht werden. Zusätzlich werden

noch das Minimum und das Maximum der Gewinnanzahlen angegeben.

3.4. Datenquellen und Normung

Datenquellen liefern die Informationen, mit denen das Netz trainiert werden soll. Sie stellen das Bindeglied zwischen den zu untersuchenden Daten und der SOM dar. Der Aufbau der Datenquelle bestimmt, über wie viele Eingabeneuronen die Karte verfügen muss. Die Daten selbst liegen meist in *Relationen* vor, die mit Datensammlungen oder durch Versuche aufgestellt werden.

Eine Relation enthält eine Liste von *Instanzen*, die Auskunft über einen gemeinsamen Satz untersuchter *Attribute* geben. Eine solche Relation lässt sich mit einer Tabelle darstellen, in der jede der Spalten ein Attribut repräsentiert, während jede der Zeilen eine Instanz beschreibt. Es kann auch vorkommen, dass Instanzen keine Information über einige Attribute enthalten: Das würde einer leeren Tabellenzelle an der betroffenen Stelle entsprechen.

Zur Beschreibung einer solchen Relation wurde an der Universität von Waikato das *ARFF*, das Attribute-Relation File Format, entwickelt, um es mit der Weka Software für maschinelles Lernen einzusetzen. Dieses Format ist in [WF01] dokumentiert und durch [Pay02] erweitert. Es stellt einen sehr nützlichen und überblickbaren Standard dar, der inzwischen bei vielen Instituten Anwendung findet. Die Datenquellen der Zielapplikation werden sich in den Grundzügen der Struktur an diesem Format orientieren, die ausschließliche Nutzung des ARFF ist aber nicht vorgeschrieben.

Da eine Relation durch eine Tabelle beschrieben werden kann, wird jede Datenquelle eine solche enthalten, die *Datenpool* genannt wird. Das .NET Framework bietet eine Klasse namens `DataTable`, die für diese Aufgabe hervorragend geeignet ist. In ihr kann für jede Spalte ein anderer Datentyp definiert werden und sie enthält eine Liste von `DataRow`-Objekten, die die Zeilen der Tabelle - und somit die Instanzen - repräsentieren.

3.4.1. Datentypen in Relationen

In Anlehnung an das ARFF wird die Verwendung von drei Hauptdatentypen vorgeschrieben, in denen die Daten aus der Quelle vorliegen sollen. Dazu gehören numerische, nominale und String-Attribute. ARFF enthält noch ein Viertes: das Datum. Dieser

Typ wird in dieser Arbeit aber nicht berücksichtigt, da es einfach möglich ist, ein Datum in einen numerischen Wert zu überführen.

Numerische Attribute stellen jegliche Art von Zahlen dar. Ganze und Fließkommazahlen sind gestattet.

Nominale Attribute werden einerseits genutzt, um einer Instanz der Relation einen Namen zu geben. Gewöhnlich werden zu diesem Zweck eindeutige Zeichenketten eingesetzt. Leere oder sich wiederholende Zeichenketten sind für diese Aufgabe daher wenig sinnvoll. Ein anderer Aspekt nominaler Attribute ist die Benennung von definierten Zuständen. So kann eine Liste von erlaubten Zuständen vorhanden sein, von denen einer bei der Instanz zutreffend ist.

String-Attribute sind Zeichenketten. Sie können mehrfach in einer Relation auftreten oder leer sein. Es gibt keine Beschränkungen für deren Inhalt.

Zur programminternen Repräsentation dieser Attributtypen wird eine Klasse namens `ColumnType` eingeführt. Sie soll abstrakt sein und durch die drei Klassen `NumericType`, `NominalType` und `StringType` implementiert werden, die jeweils eines der Attribute darstellen. Für jede der im Datenpool enthaltenen Spalten soll ein solcher Typ definiert sein.

3.4.2. Normung der Attribute

Jeweils eine Instanz aus der Relation wird dem Netz während des Trainings bzw. zur Bestimmung des Gewinnerneurons angeboten. Es werden die Attribute der Relation an die Eingabeneuronen der Karte angelegt. Damit das Netz mit den angebotenen Daten aber überhaupt umgehen kann, ist es notwendig, die Attribute der Instanz vorher in numerische Werte zu überführen, also zu *normieren*. Hierbei handelt es sich nicht um eine triviale Aufgabe, da beliebige Zeichenketten in den Attributen enthalten sein können.

$$norm : A^n \mapsto \Re^m;\ n,m \in \aleph \tag{3.2}$$

Die Formel 3.2 beschreibt diesen Vorgang. Zur Erfassung der möglichen Attributwerte wird das Symbol A eingeführt, es steht für die Menge aller numerischen, nominalen

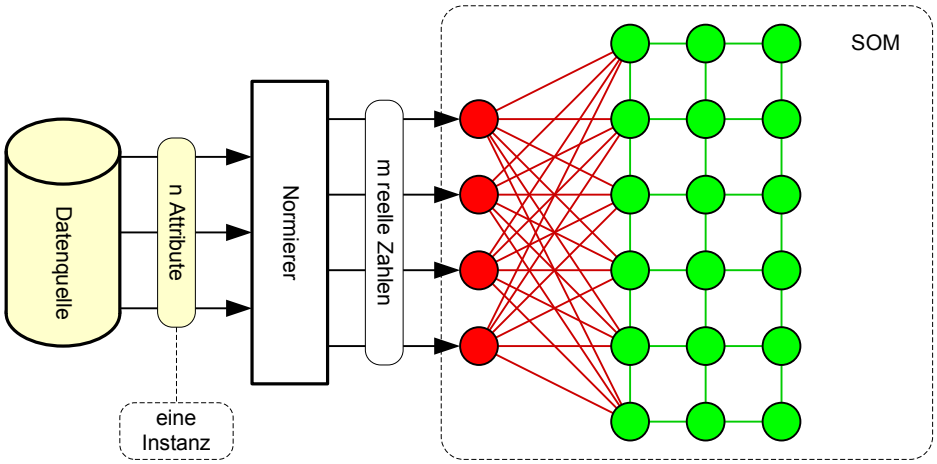

Abbildung 3.7.: Das Zusammenspiel von der Datenquelle und dem Normierer bestimmt die Anzahl der Eingabeneuronen (rot) der SOM

und String-Attribute. Die Funktion *norm* überführt eine Menge von n Attributen (also einer Instanz) in m reelle Zahlen. Man beachte die unterschiedliche Dimensionierung: Es kann vorkommen, dass einzelne Attribute während der Normung herausgefiltert oder in mehrere reelle Zahlen überführt werden.

Einige Attribute können für das Netz irrelevant sein, wie z.B. der Name einer Instanz, und deshalb bei der Normung ignoriert werden.

Nominale Attribute, die einen Zustand repräsentieren, können beispielsweise auch in mehrere reelle Werte - für jeden erlaubten Zustand einen - aufgeschlüsselt werden. Das würde bedeuten, dass dem zugrunde liegenden Zustand eine 1 und jedem anderen eine 0 zugeordnet wird. Das hätte den Vorteil, dass alle Zustände, die sich nicht in eine Reihenfolge bringen lassen, immer gleich weit voneinander entfernt sind. So wird vom Netz keine unnötige Verbindung zwischen zwei unterschiedlichen Zuständen interpretiert.

Aus diesen Gründen ist es zu verstehen, dass nicht die Datenquelle allein die Anzahl der Eingabeneuronen der Karte bestimmt, sondern dass diese nur im Zusammenspiel mit dem Normierer errechnet werden kann. Der Normierer selbst ist nämlich auch nur mit Hilfe der Eingabedaten aus der Quelle in der Lage, die Dimension seiner Ausgabe zu bestimmen. Abbildung 3.7 verdeutlicht diese Abhängigkeit.

3.4.3. Struktureller Aufbau

Sowohl die Datenquelle als auch der Normierer werden austauschbar gestaltet. So wird es ermöglicht, die Daten aus verschiedensten Quellen zu beschaffen, wie Dateien (z.B. ARFF oder XML), Datenbanken oder gar Tastatureingaben.

3.5. Training der Karte

Das Training einer selbstorganisierenden Karte ist eine der Hauptaufgaben der Applikation. Der Ablauf dieses Vorgangs soll über Lernparameter beeinflusst werden. Es soll möglich sein, den Verlauf des Trainings zu unterbrechen und die Lernparameter anzupassen.

Der Lernvorgang soll so gestaltet werden, dass er überwachbar ist. Es werden zu diesem Zweck Ereignisse definiert, die nach jedem Lernschritt und nach dem Beenden des Trainings ausgelöst werden. So wird es z.B. möglich, den aktuellen Zustand der Karte nach jedem Lernschritt an die Visualisierungen zu übermitteln.

3.5.1. Lernparameter

Die Lernparameter beeinflussen den Trainingsprozess maßgeblich. Zu ihnen gehören Die Lernrate, der Lernradius, die Nachbarschaftsfunktion und die Abbruchbedingung.

Die Lernrate bestimmt, wie stark sich das Netz während eines Lernschritts an die trainierte Dateninstanz annähert. Es gilt $0 \leq \eta \leq 1$; 0 bedeutet keinerlei Anpassung, 1 bedeutet, dass das Gewinnerneuron die Werte der Instanz vollständig für sich übernimmt.

Der Lernradius legt fest, welche Neuronen im Umkreis des Gewinnerneurons ebenfalls von der Anpassung der Karte betroffen sind.

Die Nachbarschaftsfunktion gibt vor, in wie weit sich die Nachbarneuronen innerhalb des Lernradius ebenfalls an die Dateninstanz annähern. Die Funktion wird mit der Formel $f_{Nb}(d,r) \mapsto \Re$ beschrieben. Über den Abstand des Nachbarn d und den Lernradius r wird eine reelle Zahl berechnet, die - multipliziert mit der Lernrate - die Anpassung des Nachbarn ergibt.

Die Abbruchbedingung legt fest, wann das Training für beendet erklärt wird. Das kann z.B. nach einer bestimmten Anzahl von Lernschritten oder nach Unterschreitung eines bestimmten Quantisierungsfehlers sein.

3.5.2. Schnappschüsse und Ereignisse

Der Trainingsprozess ist sehr rechen- und somit zeitaufwändig. Deshalb wird er in einem parallelen Thread ausgeführt, um den Ablauf des restlichen Programms nicht unnötig zu unterbrechen.

Um das Training trotzdem zu überwachen und auf Zustandsänderungen der selbstorganisierenden Karte reagieren zu können, werden zwei Ereignisse eingeführt, die über den Ablauf des Lernvorgangs informieren:

„Lernprozess beendet." Dieses Ereignis soll nur einmal während des Trainings auftreten, und zwar zu seinem Ende. Es soll beispielsweise dazu verwendet werden, grafische Steuerelemente der Benutzeroberfläche zu aktualisieren.

„Lernschritt beendet." Dieses Ereignis soll nach jedem Lernschritt ausgelöst werden. Es wird dafür benötigt, die Visualisierungen über den aktuellen Zustand der Karte zu informieren, damit sie sich neu berechnen und ihre Ansicht aktualisieren können.

Das „Lernschritt beendet"-Ereignis übergibt nicht nur den Zustand der SOM an alle Event Handler. Da einige Visualisierungen auch abhängig von der Datenquelle oder den Lernparametern sein können, müssen diese ebenfalls mit übergeben werden. Zu diesem Zweck soll eine Datenstruktur geschaffen werden, die *Schnappschuss* genannt wird. Es wird nämlich je ein Abbild der SOM, der Datenquelle und des Normierers, sowie der Lernparameter erzeugt, die in dieser Struktur gehalten werden. Die Erzeugung der Abbilder (also Klone) ist notwendig, da sich die Werte in den Urbildern während des Trainings ändern, und sich somit ein inkonsistenter Zustand ergeben kann. Schnappschüsse enthalten nur konstante und konsistente Zustände. Ein solcher Schnappschuss wird also nach einem Lernschritt erzeugt und an das entsprechende Event übergeben.

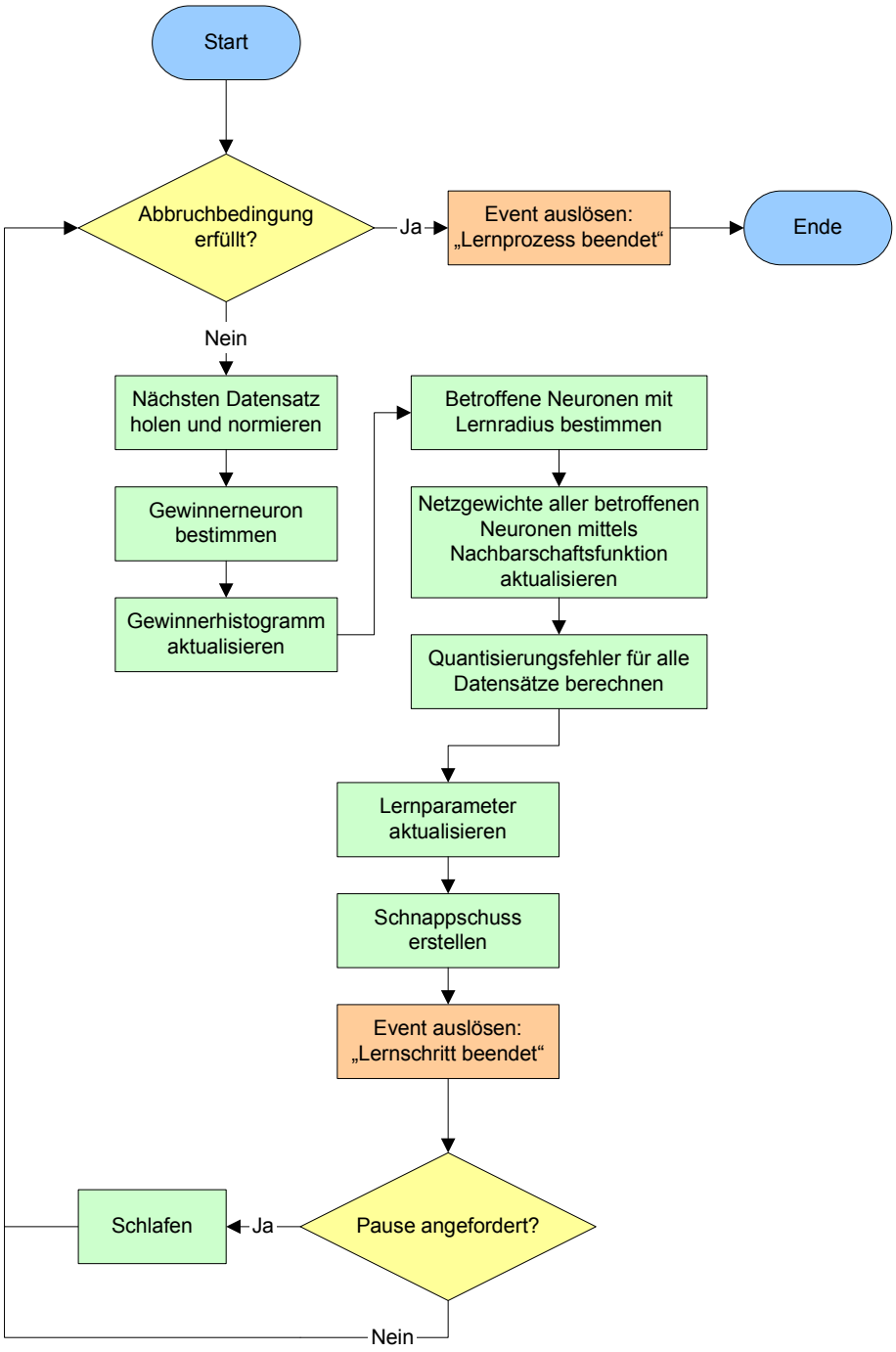

Abbildung 3.8.: Der nebenläufige Trainingsprozess lässt sich pausieren und mit Hilfe von definierten Ereignissen überwachen

3.5.3. Ablauf des Trainings

Der Trainingsprozess soll so gestaltet werden, dass er vom Nutzer unterbrochen und fortgesetzt werden kann.

Das Flussdiagramm in Abbildung 3.8 stellt des gesamten Ablauf dar. Dort ist zu erkennen, dass es sich um einen zyklischen Prozess handelt, bei dem zu jedem Durchlauf überprüft wird, ob die in den Lernparametern definierte Abbruchbedingung erfüllt ist. Sollte dem so sein, wird das Ereignis „Lernprozess beendet“ ausgelöst und das Training beendet.

Ansonsten wird der folgend beschriebene Ablauf wiederholt. Zuerst wird der nächste Datensatz aus der Datenquelle angefordert und anschließend mit Hilfe des Normierers in einen Realzahlvektor umgewandelt. Dieser wird wiederum an die SOM übergeben, um das entsprechende Gewinnerneuron zu bestimmen. Zu diesem Zweck soll die Distanzmetrik eingesetzt werden: Die Distanzen zwischen dem Datenvektor und den Wichtungsvektoren aller Neuronen werden berechnet, das Gewinnerneuron hat davon den minimalen Wert. Das kumulierte Gewinnhistogramm der Karte wird daraufhin an dieser Stelle um eins erhöht.

Die Netztopologie der Karte soll anschließend genutzt werden, um alle Nachbarneuronen, die sich innerhalb des Lernradius um das Gewinnerneuron herum befinden, zu bestimmen und deren Abstände zu berechnen. Die Abstände und der Lernradius werden an die Nachbarschaftsfunktion weitergeleitet, die daraufhin jedem betroffenen Neuron einen Lernfaktor zuweisen kann.

Die Aktualisierung der Kartengewichte soll nun wie folgt für jedes betroffene Neuron ablaufen: Der Gewichtsvektor des Neurons wird vom Datenvektor subtrahiert. Der Ergebnisvektor wird mit der Lernrate und dem entsprechenden Lernfaktor (von der Nachbarschaftsfunktion) multipliziert und zum Gewichtsvektor addiert. Formel 2.6 auf Seite 13 verdeutlicht dies noch einmal.

Nach der Aktualisierung der Gewichte soll die Gewinnerliste aktualisiert werden. Dafür wird für jede Instanz der Datenquelle das entsprechende Gewinnerneuron bestimmt und der Quantisierungsfehler (entspricht der Distanz von Gewichtsvektor und Datenvektor) berechnet.

Schlussendlich müssen nur noch die Lernparameter angepasst werden. Dem Nutzer soll es ermöglicht werden, die Lernrate und den Lernradius im Trainingsverlauf z.B.

linear oder exponentiell zu reduzieren, um eine Stagnation der Karte zu erreichen. Die Reduktion wird in diesem Teil vollzogen und der Lernschritt somit beendet.

Nun sollen die Interessenten mit Hilfe des „Lernschritt beendet“-Events über den Fortschritt informiert werden. Dazu muss ein Schnappschuss der Lernumgebung (bestehend aus Datenquelle, Normierer, SOM und Lernparameter) erzeugt und das Ereignis damit ausgelöst werden.

3.6. Visualisierungen

Visualisierungen sollen ebenfalls über Plugins an das Hauptprogramm gekoppelt werden. Sie sind die komplexesten aller Erweiterungen, an sie sind die höchsten Anforderungen gerichtet. Sie müssen sich an Strukturrichtlinien halten, um die vom Hauptprogramm geforderte Nebenläufigkeit zu unterstützen. Visualisierungen müssen selbst bestimmen können, ob sie im gegebenen Kontext anwendbar sind oder nicht.

3.6.1. Anwendbarkeit von Visualisierungen

Die Visualisierungen für die selbstorganisierenden Karten werden ausschließlich über Plugins zur Verfügung gestellt. Ihre Darstellung ist maßgeblich von der Topologie der verwendeten SOM abhängig. Da diese ebenfalls nur über Plugins realisiert werden, ist es notwendig, die Anwendbarkeit von Visualisierungen in Bezug auf die Topologie und anderen Eigenschaften der Karte zu untersuchen. Das Ziel dieses Vorgehens ist die Definition eines Katalogs, der Informationen über die (für Visualisierungen interessante) Eigenschaften des Netzes zusammenfasst. Diese Informationen sollen so weit gefächert sein, dass die meisten Topologien und die Anforderungen der meisten Visualisierungen abgedeckt sind. Die Visualisierungen sollen mit dem Katalog in der Lage sein, selbst zu erkennen, ob sie auf ein Netz anwendbar sind oder nicht.

Hierzu wird ein Blick zurück auf einige Visualisierungsformen geworfen und untersucht, welche Eigenschaften der SOM von Interesse für sie sind.

Karte im Eingaberaum. Die „Karte im Eingaberaum“ stellt Verbindungen benachbarter Neuronen und alle Eingabevektoren dar. Die Position eines jeden Neurons wird bestimmt, indem jedem Gewicht eine Achse des (meist zwei- oder dreidimensionalen) Eingaberaums zugeordnet wird. Die Netztopologie wird zumeist

als ein- bis dreidimensional vorausgesetzt. Die von dieser Visualisierungsart benötigten Informationen bestehen also aus den Dimensionierungen der Topologie und des Eingaberaums.

Gewinnermatrix. Da die Gewinnermatrix eine musterabhängige Visualisierungsform ist, braucht sie für ihre Darstellung Datensätze aus dem Datenpool. Außerdem werden Lernparameter benötigt, wie die Nachbarschaftsfunktion und der Lernradius, da nicht nur das Siegerneuron sondern auch seine beeinflusste Umgebung hervorgehoben wird. Diese Darstellungsart ist also ein gutes Beispiel dafür, dass Visualisierungen nicht nur von der Topologie selbst, sondern auch von der Datenquelle oder den Lernparametern abhängig sein können. Die Gewinnermatrix stellt aber keinerlei Einschränkungen bezüglich dieser Werte auf, daher ist sie auf jede Form von SOM anwendbar.

Gewichtsmatrix. Diese Darstellung stellt keine Anforderungen an das Netz. Da einfach nur die Gewichte aller Neuronen für die Ansicht verwendet werden, ist die Art und Dimensionierung der Topologie völlig unerheblich. Sie ist musterunabhängig. Diese Visualisierung ist also auch für jede Form von SOM verfügbar.

U-Matrix. Diese Visualisierungsform besteht auf eine zweidimensionale Netztopologie. Es werden die Distanzen der Neuronengewichte und die Positionszuordnungen der Neuronen als Darstellungsgrundlage genutzt. Zirkuläre Topologien, wie der Torus und der Zylinder, können hier speziell behandelt werden: Für den Torus kann die U-Matrix z.B. vierfach (auf einem 2×2 Feld) dargestellt werden, damit zusammenhängende Bereiche, die über die Matrixgrenzen hinausgehen, trotzdem zusammenhängend visualisiert werden können. Gleiches gilt für eine zylindrische Topologie, die zweifach dargestellt werden kann (2×1 oder 1×2, je nachdem, welche der Achsen zyklisch ist).

Zusätzlich wird die ID der Topologie mit in den Eigenschaftenkatalog aufgenommen. So kann man eine Visualisierung erzeugen, die ausschließlich für eine ganz bestimmte Topologie funktioniert. Das kann nützlich sein, wenn eine sehr komplexe Topologie erstellt wird (z.B. in Form eines unregelmäßigen Graphen), für die eine spezielle Visualisierung erforderlich ist. Die Tabelle 3.1 zeigt zusammenfassend die ausgewählten Eigenschaften und deren Informationsquellen.

Informationsquelle	Eigenschaften
Topologie	ID Anzahl Dimensionen Ausdehnung (für jede Dimension, z.B. Breite, Höhe) zyklisch: ja / nein (für jede Dimension) Anzahl Neuronen
Datenquelle	Dimension des Eingabedatenraums
Normierer	Dimension des normierten Eingabedatenraums (= Anzahl der Eingabeneuronen der SOM)

Tabelle 3.1.: Der Eigenschaftenkatalog zur Bestimmung der Anwendbarkeit von Visualisierungen mit den notwendigen Informationen über die SOM und deren Quellen

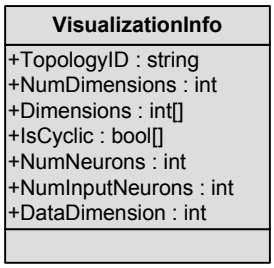

Abbildung 3.9.: Die Attribute aus dem Eigenschaftenkatalog überführt in eine Klasse

Andere Eigenschaften, wie etwa die Gewichte und Positionszuordnungen der Ausgabeneuronen der SOM, Nachbarschaftsverhältnisse, Distanzmetriken, Lernparameter oder Datensätze aus der Datenquelle, werden in den Katalog nicht mit aufgenommen. Sie dienen nur dem Datentransfer und sind daher von der Anwendbarkeit einer Visualisierung unabhängig. Diese Daten werden nach jeder Aktualisierung der SOM erneut zur Verfügung gestellt.

Der Eigenschaftenkatalog wird in die Klasse `VisualizationInfo` als Resultat dieser Betrachtung überführt. Die Abbildung 3.9 zeigt deren UML-Diagramm, das die ausgewählten Eigenschaften und ihre jeweiligen Datentypen benennt. Eine Instanz dieser Klasse wird im späteren Verlauf an alle Visualisierungen übergeben, die daraufhin ihre Einsatzmöglichkeit überprüfen.

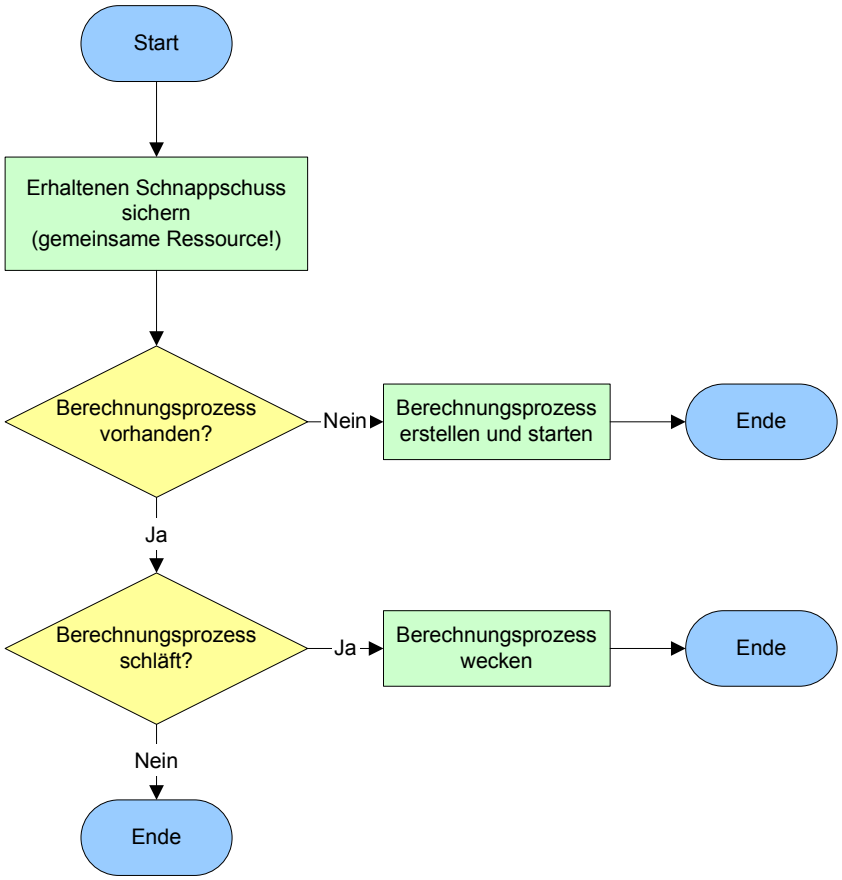

Abbildung 3.10.: Visualisierungen werden nach jedem Lernschritt mit einem Schnappschuss des aktuellen Kontextes aktualisiert, der Berechnungsprozess wird damit angestoßen

3.6.2. Visualisierungen und Nebenläufigkeit

Die Berechnung einer SOM-Visualisierung kann sich als sehr komplex herausstellen und gegebenenfalls sehr viel Rechenzeit in Anspruch nehmen. Die Darstellung soll aber stets responsiv bleiben und eine Navigation auf der Karte ohne merkliche Verzögerungen erlauben. Außerdem soll der laufende Trainingsprozess nicht unnötig unterbrochen werden. Aus diesen Gründen werden Strukturrichtlinien vorgegeben, die die Darstellung der Visualisierung und deren Berechnung parallelisieren.

Während des Trainings werden die Gewichte der Karte verändert. Nach jedem Lernschritt werden alle aktiven Visualisierungen aktualisiert, d.h. ein Schnappschuss des aktuellen Lernkontextes (die SOM, Lernparameter und Zustand der Datenquelle) wird

an sie übergeben. Sollte die Berechnung der Darstellung noch nicht beendet sein, wird der Schnappschuss gespeichert. Sollte bereits ein gespeicherter Schnappschuss vorhanden sein, so wird dieser überschrieben. Dieser Vorgang hat zur Folge, dass die Darstellung asynchron zum aktuellen Zustand der SOM stattfindet, was als Nachteil der Nebenläufigkeit angesehen werden kann. Dieser kann aber vernachlässigt werden, da die Darstellung nur wenige Lernschritte zurückliegt und durch das Überschreiben des Schnappschusses nicht jeder verpasste Zustand der Karte nachträglich visualisiert wird, was zur Folge hat, dass der Abstand zwischen der Darstellung und dem aktuellen Zustand nicht weiter wächst. Des Weiteren kann der Trainingsprozess pausiert werden, was automatisch zu einer Synchronisierung der Visualisierung führen würde.

Die Abbildung 3.10 stellt den Aktualisierungsprozess dar. Der übergebene Schnappschuss wird gespeichert, ein eventuell vorhandener wird überschrieben. Anschließend wird überprüft, ob bereits ein Berechnungsthread vorhanden ist. Dies ist nur bei der ersten Aktualisierung nicht der Fall, also wird unter Zuhilfenahme des Applikationskontextes ein Berechnungsthread erzeugt und gestartet. Sollte der Thread doch schon laufen wird untersucht, ob dieser „schläft“. Das kann vorkommen, wenn ein Berechnungszyklus beendet wurde und noch keine neuen Daten (also ein neuer Schnappschuss) vorhanden waren. Deshalb kann der Thread nun „geweckt“ werden, da wieder Arbeit für ihn ansteht.

Der zyklische Berechnungsprozess wird in Abbildung 3.11 gezeigt. Wie bereits erläutert legt er sich schlafen, wenn kein neuer Schnappschuss für die Berechnung vorhanden ist. Ansonsten wird der neue Schnappschuss geholt und an die Methode zur Berechnung der Visualisierung weitergegeben. Anschließend wird die Methode zur Darstellung aufgerufen. Es scheint, dass die Berechnung und die Darstellung seriell ablaufen, nicht parallel. Dem ist nicht so, da die Darstellungsmethode noch zu anderen Zeiten aufgerufen wird, z.B. bei Nutzerinteraktion mit der Visualisierung (Navigation auf der Karte) oder bei Größenänderung und Verschiebung des Fensters.

Der gesamte Ablauf der Aktualisierung und der Berechnungsprozess sollen bereits in der Basisklasse für Visualisierungsplugins implementiert werden. So braucht dieses komplexe Konstrukt nicht bei jeder neuen Implementierung erneut programmiert werden. Lediglich die Methoden zur Berechnung und zur Darstellung der Visualisierung werden virtuell gestaltet und müssen deshalb neu implementiert werden. Dabei ist darauf zu achten, dass auf die Ergebnisse der Berechnung, die in die Darstellung einfließen,

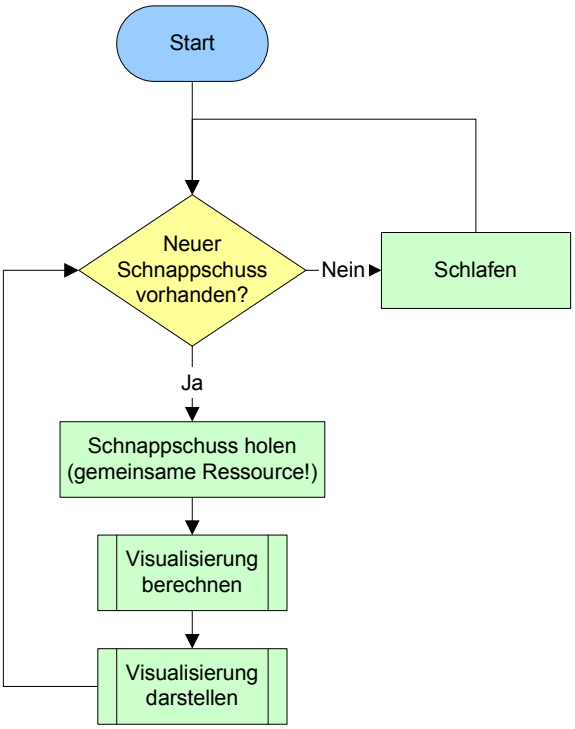

Abbildung 3.11.: Der Berechnungprozess der Visualisierung: Nach der Berechnung wird automatisch neu gezeichnet

threadsicher zugegriffen werden muss, da diese beiden Methoden parallel ablaufen können.

3.7. Plugins

Plugins beschreiben eine Reihe von Eigenschaften, die die Integration von externen Komponenten für festgelegte Aufgaben erlauben. Jedes Plugin erhält eine eigene ID und einen Namen, die in der Schnittstelle `IPlugIn` definiert sind.

Jedem Plugin-Typen wird eine abstrakte Basisklasse zur Verfügung gestellt, die die benötigten Schnittstellen (wie z.B. `IPlugIn`) implementiert. Es werden auch noch andere Eigenschaften oder Methoden in den Basisklassen eingeführt, die auf besondere Eigenheiten der Plugins eingehen. Es kann vorkommen, dass Teile einer Basisklasse bereits vorimplementiert sind, damit triviale oder identische Abläufe bei der späteren Plugin-Entwicklung nicht immer wieder neu programmiert werden müssen.

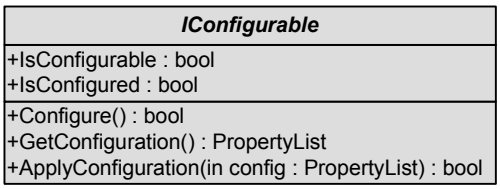

Abbildung 3.12.: Das UML-Diagramm der Schnittstelle, mit der die Konfigurierbarkeit eines Plugins definiert wird

Die Basisklassen entsprechen folgender Namenskonvention: Jede endet auf `*Base`, beispielsweise heißt die Basisklasse für Visualisierungen `VisualizationBase`. Der Konstruktor von ableitenden Klassen - also implementierten Plugins - muss die gleiche Signatur haben, wie der Konstruktor der Basisklasse.

3.7.1. Der Applikationskontext

Jedem Plugin soll es ermöglicht werden, Zugriff auf bestimmte Ressourcen der Hauptapplikation zu erhalten oder bestimmte Aufgaben an sie weiterzureichen. Außerdem soll der aktuelle Status der Anwendung bestimmt werden können. Zu diesem Zweck wird ein Konstrukt entworfen, das *Applikationskontext* genannt wird. So wird erreicht, dass das Plugin-Management-System nicht nur einen einseitigen Zugriff vom Hauptprogramm zu den Plugins ermöglicht, sondern dass eine beidseitige Interaktion stattfindet.

Allen Plugins wird damit ermöglicht, bestimmte Ressourcen zu erreichen, wie Standardlogger zur Fehlerausgabe oder zusätzliche Threads. Threads sollten generell nur über den Applikationskontext erzeugt werden, da so der Hauptapplikation ermöglicht wird, alle Threads in einem Pool zu halten und somit deren Ablauf zu überwachen. Nur so wird sichergestellt, dass beim Beenden des Programms auch alle laufenden Threads sicher beendet werden können.

Der Zugriff auf den Applikationskontext ist dadurch gewährleistet, dass eine Instanz davon bei der Erzeugung aller Plugins an den jeweiligen Konstruktor übergeben wird.

3.7.2. Konfigurierbarkeit von Plugins

Ein Plugin erhält die Möglichkeit, eigene Parameter, die der Hauptapplikation unbekannt sind, vom Nutzer konfigurieren zu lassen. Diese Parameter können zum Beispiel

die Datenbankverbindung eines Datenquellen-Plugins oder die Darstellungsfarben eines Visualisierungs-Plugins sein. Zu diesem Zweck werden Eigenschaften in einer weiteren Schnittstelle definiert, die den Zugang zu diesen Parametern ermöglichen. Der Name der Schnittstelle lautet `IConfigurable`. Wie in dem UML-Diagramm auf Abbildung 3.12 zu sehen, definiert sie die Eigenschaften `IsConfigurable` und `IsConfigured`, sowie die Methoden `Configure`, `GetConfiguration` und `ApplyConfiguration`.

Die Eigenschaft `IsConfigurable` liefert einen Booleschen Wert, der angibt, ob das betrachtete Plugin überhaupt konfiguriert werden kann. Es gibt Plugins, die immer gleich funktionieren und nicht von externen Parametern abhängig sind, für diese sollte der Wert `false` sein. Andererseits sollten Plugins, die über einstellbare Parameter verfügen, den Wert `true` zurückgeben.

Die Eigenschaft `IsConfigured` sagt aus, ob das Plugin in seinem momentanen Zustand über eine gültige Konfiguration verfügt. Wenn dieser Wert `false` enthält, müssen die Parameter des Plugins noch angepasst werden, es ist noch nicht einsetzbar. Ein Beispiel: Es gibt eine Datenquelle, die die Eingabewerte aus einer Textdatei bezieht. Es wurde dieser Datenquelle aber noch keine Datei zum Auslesen zugewiesen. Die Datenquelle kann also in ihrem momentanen Zustand nicht arbeiten, sie erwartet also die Festlegung der Quelldatei. Nicht konfigurierbare Plugins müssen bei dieser Eigenschaft immer `true` zurückgeben.

Die Methode `Configure` weist das Plugin an, dem Nutzer seine Parameter (z.B. über einen Dialog) offenzulegen und anpassen zu lassen. Anschließend muss der Wert der `IsConfigured`-Eigenschaft aktualisiert werden, da sich die Gültigkeit der Konfiguration geändert haben kann. Diese Methode braucht nicht implementiert zu werden, falls `IsConfigurable` den Wert `false` liefert. Der folgende Abschnitt befasst sich mit dem Vorgang der Selbstkonfiguration, der durch diese Methode angestoßen wird.

Die Methoden `GetConfiguration` und `ApplyConfiguration` sollen es ermöglichen, ein Plugin in seinem Zustand zu persistieren. Der Hintergrund dieser Aufgabe wird im folgenden Abschnitt „Persistenz von Plugins“ beleuchtet.

Selbstkonfiguration mit Hilfe dynamischer Konfigurationsdialoge

Die in der Schnittstelle definierte Methode `Configure` wird aufgerufen, um dem Plugin mitzuteilen, dass es nun mit der Selbstkonfiguration beginnen kann. So ist es beispiels-

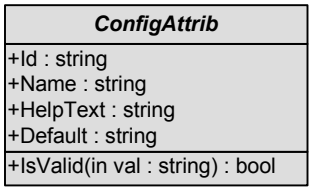

Abbildung 3.13.: Die abstrakte Basisklasse für alle Konfigurationsattribute

weise möglich, einen Konfigurationsdialog anzuzeigen, in dem die für das Plugin notwendigen Parameter vom Nutzer überprüft und bearbeitet werden können. Es ist jedoch oft sehr aufwendig, für jedes Plugin mit wenigen einfachen Parametern einen zusätzlichen Dialog zu erstellen. Zu diesem Zweck wird durch den Applikationskontext eine Möglichkeit geschaffen, solche Konfigurationsdialoge dynamisch von der Hauptanwendung erzeugen zu lassen.

Der Applikationskontext stellt für diese Aufgabe eine Methode zur Verfügung, die einen solchen Konfigurationsdialog erzeugt und zurückgibt. Seine Funktionalität wird über eine Schnittstelle namens `IConfigDialog` festgelegt. Durch sie wird dem Plugin der Zugriff auf den erzeugten Dialog ermöglicht. Der Methode muss über eine Liste von sogenannten *Konfigurationsattributen* mitgeteilt werden, wie er in Erscheinung treten soll.

Ein solches Konfigurationsattribut enthält Informationen über einen zu bearbeitenden Parameter des Plugins. Mit ihm wird der Name des Parameters festgelegt, der auf dem Dialog erscheint. Es muss eine Möglichkeit geschaffen werden, seinen durch den Nutzer bestimmten neuen Wert später auslesen zu können. Außerdem kann ein Hilfetext und eine Standardeinstellung für ihn angegeben werden. Jedes Konfigurationsattribut wird von einer abstrakten Basisklasse namens `ConfigAttrib` abgeleitet, die diese Grundfunktionalitäten definiert. Die Abbildung 3.13 zeigt das UML-Diagramm dieser Klasse. In ihr ist auch die abstrakte Methode `IsValid` definiert, die eine Zeichenkette als Parameter erwartet und die von den ableitenden Klassen implementiert werden muss. Der Konfigurationsdialog bekommt mit dieser Methode die Möglichkeit, Nutzereingaben zu überprüfen: Der Nutzer gibt in dem Dialog einen Wert für einen bestimmten Parameter ein, der vom Attribut als ungültig deklariert werden kann. Der Dialog ist anschließend in der Lage, den Nutzer über die Fehleingabe zu informieren und um Korrektur zu bitten. So ist garantiert, dass dem Plugin nur gültige Parameter übergeben werden können.

Es sind mehrere Klassen vorgesehen, die von `ConfigAttrib` erben. Sie geben die gebräuchlichsten Parametertypen für Plugins vor und definieren indirekt das mit dem Parameter verbundene Steuerelement (wie z.B. ein Texteingabefeld) auf dem Konfigurationsdialog. Diese Klassen werden nachfolgend genannt und beschrieben.

BoolAttrib stellt ein Konfigurationsattribut für einen booleschen Parameter dar, der nur die Werte `true` oder `false` annehmen kann. Eine Checkbox wird diesem Attribut auf dem Konfigurationsdialog zugeordnet.

DoubleAttrib ist für numerische, reelle Plugin-Parameter vorgesehen. Es ist möglich, den Wertebereich der Gleitkommazahl durch ein Minimum und ein Maximum zu begrenzen. Ein Texteingabefeld wird diesem Attribut als Steuerelement zugeordnet.

EnumAttrib definiert einen Parameter, der seinen Zustand aus einer endlichen Werteliste, einer *Enumeration* oder Aufzählung, erhält. Es wird eine 1-aus-n-Beziehung beschrieben. Die Menge der möglichen Zustände wird über eine Liste von Zeichenketten definiert. Der Konfigurationsdialog legt für dieses Attribut eine Auswahlbox fest.

FileAttrib ermöglicht einen Plugin-Parameter, der Zugriff auf eine Datei erlangt. Es wird festgelegt, ob die Zieldatei gelesen oder geschrieben werden soll, sowie welchen Typs die ausgewählte Datei sein darf. Für dieses Attribut legt der Dialog ein Texteingabefeld an. Ein zusätzlicher Knopf öffnet einen Dateiauswahldialog, mit dem die Zieldatei lokalisiert werden kann.

IntAttrib kommt zum Einsatz, wenn ein numerischer, ganzzahliger Parameter konfiguriert werden soll. Der Wertebereich der Ganzzahl kann, in Anlehnung an das `DoubleAttrib` Attribut, mit Minimum und Maximum beschränkt werden. Auch hier wird für den Parameter ein Texteingabefeld angelegt.

StringAttrib ist für Zeichenketten (wie Namen o.ä.) zuständig. Es gibt keinerlei Beschränkungen für die Eingabe. Ein Texteingabefeld ist das Steuerelement für diesen Parametertyp.

Wie diese Konfigurationsattribute angewendet werden, wird mit einem Codebeispiel im Anhang auf Seite 105 beschrieben.

Persistenz von Plugins

Damit der aktuelle Zustand (Trainingsfortschritte, aktuelle Eingabeparameter etc.) einer SOM gespeichert werden kann, ist es notwendig auch die verwendeten Plugins zu persistieren. So soll die Konfiguration eines Plugins (also z.B. der Zustand der Datenquelle und des Normierers oder die Eigenschaften der ausgewählten Topologie) ausgelesen werden können. Genauso soll die Konfiguration beim Ladeprozess wieder an das Plugin übergeben werden können.

Als Erscheinungsform einer solchen Konfiguration wird eine *Property List* gewählt. Dabei handelt es sich um eine Liste, die eine Menge von Schlüssel-Wert-Paaren enthält und somit einfach in eine speicherbare Struktur (wie etwa XML) integriert werden kann. Wie diese Paare aufgebaut sind und was sie enthalten, ist den Plugins freigestellt. Die einzige Anforderung besteht darin, dass der gesamte aktuelle Zustand des Plugins darin abgelegt und beim Laden auch wieder daraus hergestellt wird.

Die Methode `GetConfiguration` soll eine solche Liste erzeugen und zurückgeben, die den Gesamtstatus des Plugins enthält. Die Liste muss nicht nur aus den konfigurierbaren Parametern bestehen, es können auch noch andere interne Laufzeitparameter enthalten sein.

Mit `ApplyConfiguration` wird dem Plugin eine Property List übergeben, die einen früheren Zustand wiederherstellen soll. Diese Liste wurde dann höchstwahrscheinlich an einem früheren Zeitpunkt von dem Plugin selbst erstellt.

4. Implementierung

Dieses Kapitel behandelt die Implementationsphase des Programms, es wird also die Umsetzung des Konzepts beleuchtet. Hierbei kann aber nicht auf jede einzelne, während der Entwicklung entstandene Klasse eingegangen werden, da das den Umfang der Arbeit unnötig aufblähen würde. Daher wird nur auf die interessantesten Aspekte eingegangen.

Die Zielapplikation trägt den Namen *Sombrero*, deshalb kann dieser Begriff des Öfteren in der Arbeit auftreten. Wenn also darüber geschrieben wird, ist nicht von ausladenden mexikanischen Strohhüten die Rede.

Die Applikation selbst wird sehr schlank gehalten. Die meiste Funktionalität wird erst über Plugins zur Verfügung gestellt. Das Hauptprogramm ist ohne Plugins lauffähig, nur ist der mögliche Aufgabenbereich ohne vorhandene Netztopologien, Datenquellen u.s.w. doch sehr eingeschränkt.

Das Kapitel ist derart strukturiert, dass erst die Implementierung der vier Schichten der Applikation und anschließend die Entwicklung der externen Plugins beleuchtet wird. Zuvor werden die Anforderungen an das System genannt, auf dem *Sombrero* ausgeführt werden soll.

4.1. Entwicklungs- und Ausführungsumgebung

Das Programm wird in der Programmiersprache `C#` umgesetzt, einem modernen `C`-Dialekt, der 2002 mit der `.NET`-Laufzeitumgebung von Microsoft eingeführt wurde. Diese Umgebung wird bei der Ausführung benötigt und durch das `.NET Framework` zur Verfügung gestellt. Die Visualisierungen nutzen zur Darstellung der Ergebnisse die `Direct3D`-Hardwareschnittstelle, die durch eine Erweiterung des Frameworks zugänglich gemacht wird. `Visual C# 2005 Express Edition` wird zur Entwicklung der Applikation eingesetzt, eine kostenlose IDE (Integrated Development Environment)

von Microsoft. Zusammenfassend werden die Anforderungen an die Ausführungsumgebung aufgelistet:

- Microsoft `Windows XP SP2` oder `Windows 2000 SP3`
- `.NET Framework 2.0`
- `Managed DirectX 9.0c`

Obwohl `DirectX 9` für die Implementierung der Visualisierungen eingesetzt wird, ist keine Höchstleistungsgrafikkarte notwendig. Für die Ausführung von *Sombrero* sollte ein einfacher 3D-Beschleuniger mit 16 Mb Speicher vollkommen ausreichen.

4.2. Erweiterungsschicht

Die Erweiterungsschicht (Plugin Layer) der Anwendung wird in einer eigenen Laufzeitbibliothek, auch *Dynamic Link Library* oder *DLL* genannt, untergebracht. Diese wird im Hauptprogramm eingebunden.

Wenn Plugins für *Sombrero* entwickelt werden, geschieht das ebenfalls mit Hilfe von Bibliotheken. Diese müssen die Plugins-DLL der Hauptapplikation ebenfalls einbinden und die gewünschten Erweiterungen über die dort definierten Plugin-Basisklassen implementieren. Anschließend werden die neue DLL und die enthaltenen Erweiterungen bei der Applikation angemeldet. Die Verwaltung der Plugins ist eine Aufgabe der Datenzugriffsschicht, wie die Anmeldung und Einbindung funktioniert wird deshalb im entsprechenden Kapitel 4.3.2 beschrieben.

Die Erweiterungsschicht kapselt die Informationen, die die Plugins von der Hauptapplikation und umgekehrt benötigen. Dies geschieht schichtübergreifend für den Datenzugriff, die Geschäftslogik und die Präsentation. Aus diesem Grund existieren auch in dem Modul die Namensräume `DataAccess`, `BusinessLogic` und `Gui` unterhalb von `Sombrero.PlugIn`, dem Hauptnamensraum dieser Schicht.

Folgend werden die Basisklassen für die implementierbaren Erweiterungen und die im Zusammenhang stehenden Strukturen beleuchtet. Jede dieser Basisklassen definiert einen Konstruktor, der als Eingabeparameter den Applikationskontext (definiert im folgenden Abschnitt 4.2.1) erwartet. So wird der Kontext dem Plugin zugänglich gemacht.

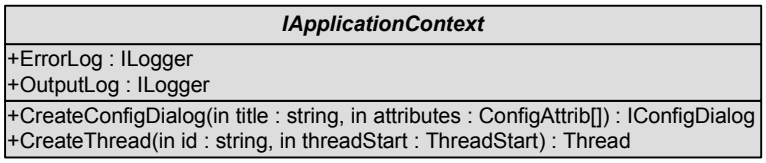

Abbildung 4.1.: Der Applikationskontext wird über die hier dargestellte Schnittstelle definiert

Die geschützte Eigenschaft `Context`, die in jeder der Basisklassen definert ist, ermöglicht den Zugriff darauf. Ableitende Plugins müssen einen öffentlichen Konstruktor mit gleicher Signatur definieren und den Kontext an den Konstruktor der Basisklasse übergeben. In `C#` wird diese Anforderung wie folgt umgesetzt:

```
// die Klasse PlugIn erbt von der imaginären Basisklasse PlugInBase
public class PlugIn : PlugInBase
{
  // über das Schlüsselwort "base" wird automatisch der Konstruktor
  // der Basisklasse aufgerufen, der Kontext wird direkt an ihn
  // übergeben
  public PlugIn(IApplicationContext context) : base(context)
  {
    // eigene Konstruktorlogik...
  }

  // ...
}
```

4.2.1. Die Applikationskontext-Schnittstelle

Der Applikationskontext (beschrieben im Kapitel 3.7.1) erlaubt es den Plugins, auf Ressourcen der Hauptanwendung zuzugreifen. Aus diesem Grund wird in dieser Schicht die Schnittstelle `IApplicationContext` definiert, deren UML-Diagramm auf Abbildung 4.1 zu sehen ist.

Die Eigenschaften `ErrorLog` und `OutputLog` ermöglichen den Zugriff auf die Standardlogger zur Ausgabe von Fehlermeldungen und Sonstigem auf jeder Ebene.

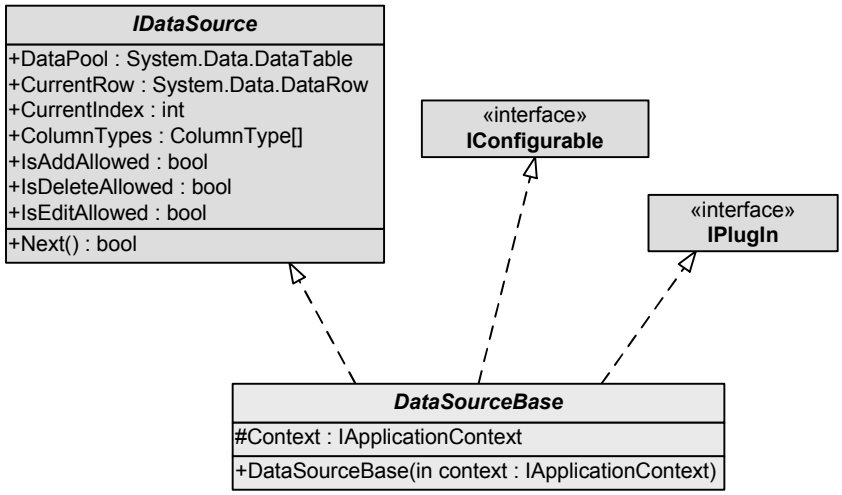

Abbildung 4.2.: Die Basisklasse für Datenquellen implementiert drei Schnittstellen

Mittels der Methode `CreateConfigDialog` wird die Applikation angewiesen, einen Konfigurationsdialog für die als Parameter übergebenen Attribute zu erzeugen. Die Hintergründe für diesen Vorgang sind in Kapitel 3.7.2 erläutert. Ein Beispiel für die Erzeugung eines solchen Dialogs befindet sich im Anhang, Kapitel A.1.

Die Methode `CreateThread` erzeugt einen von der Applikation verwalteten Thread, der in einem Pool gehalten wird. Plugins könnten ohne Weiteres auch selbst Threads erzeugen. Doch da das Hauptprogramm von diesen keine Kenntnis hätte, wäre es nicht imstande, die Threads beim Beenden anzuhalten.

4.2.2. Die Basisklasse für Datenquellen

Der Aufbau der Datenquellen (beschrieben in Kapitel 3.4) ist über die Schnittstelle `IDataSource` definiert. Auf den Datenpool kann mit der Eigenschaft `DataPool` zugegriffen werden. Es handelt sich hierbei um eine Instanz von `System.Data.DataTable`, einer Struktur des `.NET`-Frameworks, die zur Datenhaltung eingesetzt wird. Diese eignet sich hervorragend für die Ansprüche einer Datenquelle, da sie Objekte jeglichen Typs in einer Tabelle halten kann. Diese Tabelle entspricht also einer abgebildeten Relation und jede Zeile beschreibt eine Dateninstanz.

Mit der Methode `Next` wird die nächste Dateninstanz aus dem Pool angefordert. Diese kann anschließend über die Eigenschaft `CurrentRow` abgefragt werden, die eine In-

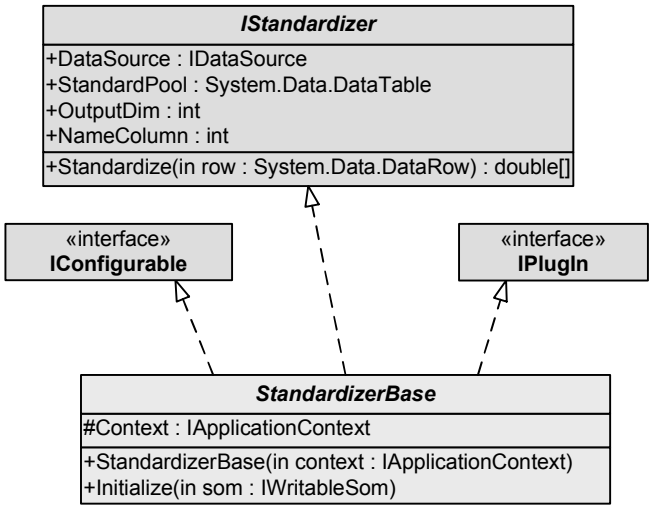

Abbildung 4.3.: Die Basisklasse für Normierer

stanz von `System.Data.DataRow` liefert, einer Zeile aus der Datenpool-Tabelle. Die Eigenschaft `CurrentIndex` gibt des Weiteren noch den Index der Dateninstanz aus dem Pool an.

Die Eigenschaft `ColumnTypes` enthält ein Array von `ColumnType`-Instanzen, die die verwendeten Attributtypen der Datenquelle beschreiben. Diese sind im Kapitel 3.4.1 erläutert.

Über die Booleschen Eigenschaften `IsAddAllowed` und `IsDeleteAllowed` wird bestimmt, ob dem Datenpool neue Daten hinzugefügt oder enthaltene Daten gelöscht werden dürfen. Mit `IsEditAllowed` wird die Veränderung der Daten gestattet. Die erlaubten Operationen können dann mit dem `Data Inspector` (beschrieben im Kapitel 4.5.3) durchgeführt werden.

Die Basisklasse für Datenquellen trägt den Namen `DataSourceBase`, sie implementiert die Schnittstellen `IDataSource`, `IPlugIn` und, da sie konfigurierbar sein soll, `IConfigurable`. Abbildung 4.2 zeigt das UML-Diagramm der Klasse und deren Abhängigkeiten.

4.2.3. Die Basisklasse für Normierer

Zur Beschreibung eines Normierers, dessen Aufgabe in Kapitel 3.4.2 beschrieben ist, wird die Schnittstelle `IStandardizer` eingeführt. Der Aufbau ähnelt einer Datenquel-

le. Es existiert eine Tabelle, die über die Eigenschaft `StandardPool` erreicht wird und die den Datenpool in genormter Form wiederspiegelt. Die mit dem Normierer verbundene Datenquelle ist über die Eigenschaft `DataSource` verfügbar. Die Eigenschaft `NameColumn` enthält den Index der Spalte im Datenpool, der den Namen einer Dateninstanz enthält. Das kann vor allem für Visualisierungen nützlich sein, um die dargestellte Karte zu beschriften. Mit der Eigenschaft `OutputDim` erhält man die Dimension der normierten Eingabedaten, sie bestimmt die Anzahl der Eingabeneuronen der Kohonenkarte.

Die Methode `Standardize` führt die Normung durch. Der Eingabeparameter `row` ist eine Dateninstanz, sie ist vom Typ `System.Data.DataRow`. Als Rückgabewert wird ein Realzahlvektor in Form eines `double`-Arrays geliefert, er enthält genau so viele Werte, wie `OutputDim` angibt.

In der Schnittstelle ist die Methode `Initialize` nicht aufgeführt, sie wird durch die Basisklasse `StandardizerBase` selbst definiert. Während der Kommunikation der Hauptanwendung mit den Plugins wird immer nur die Schnittstelle `IStandardizer` verwendet, diese Methode wird „versteckt“, damit sie nicht von einem der Plugins aufgerufen wird. Sie dient der Initialisierung des Normierers, indem die Datenquelle an ihn übergeben wird. So kann der Normierer feststellen, welche Dimension die normierten Eingabedaten haben (s. Kap. 3.4.2), damit das Netz darauf ausgerichtet wird.

Als Ergebnis dieser Betrachtungen entsteht die Basisklasse für Normierer, die die Schnittstellen `IPlugIn`, `IStandardizer`, und `IConfigurable` implementiert. Abbildung 4.3 zeigt das entsprechende UML-Diagramm.

4.2.4. Die Basisklasse für Distanzmetriken

Die Basisklasse für Distanzmetriken, die den Namen `DistanceBase` trägt, implementiert die Schnittstelle `IDistanceMetric`, die in Kapitel 3.3.1 definiert wurde, sowie `IPlugin`. Die Distanzmetrik ist nicht konfigurierbar, deshalb wird `IConfigurable` hier nicht verwendet.

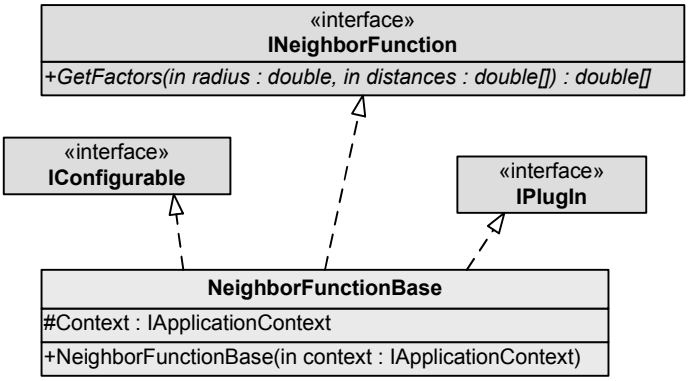

Abbildung 4.4.: Nachbarschaftsfunktionen werden über nur eine Methode definiert

4.2.5. Die Basisklasse für Nachbarschaftsfunktionen

Eine Nachbarschaftsfunktion wird für den Trainingsprozess benötigt und ist ein Teil der eingesetzten Lernparameter. Sie wird über die Schnittstelle `INeighborFunction` definiert. Sie beschreibt nur die eine Methode `GetFactors`. Mit ihr wird die Formel 2.13 von Seite 17 umgesetzt. Allerdings werden mit ihr eine ganze Reihe von Werten berechnet. Sie nimmt als Parameter den Lernradius und gleich mehrere Distanzen in einem `double`-Array entgegen. Der Grund für dieses Vorgehen ist die Tatsache, dass die Faktoren für alle sich im Lernradius befindlichen Neuronen auf einmal berechnet werden können. Als Ergebnis gibt die Methode ein `double`-Array zurück, das die Funktionswerte für jede erhaltene Distanz enthält.

Abbildung 4.4 zeigt die Basisklasse `NeighborFunctionBase` und die beschriebene Schnittstelle. Es werden auch `IPlugIn` und `IConfigurable` implementiert, es ist also erlaubt, die Nachbarschaftsfunktion konfigurierbar umzusetzen.

4.2.6. Die Basisklasse für SOM-Topologien

Eine Netztopologie definiert den Aufbau der Ausgabeschicht einer SOM, es werden die Nachbarschaftsverhältnisse der Ausgabeneuronen untereinander festgelegt. Hierzu wurde in Kapitel 3.3.2 eine Schnittstelle mit dem Namen `ITopology` entworfen. Sie wird zusammen mit `IPlugIn` und `IConfigurable` von der Basisklasse `TopologyBase` implementiert.

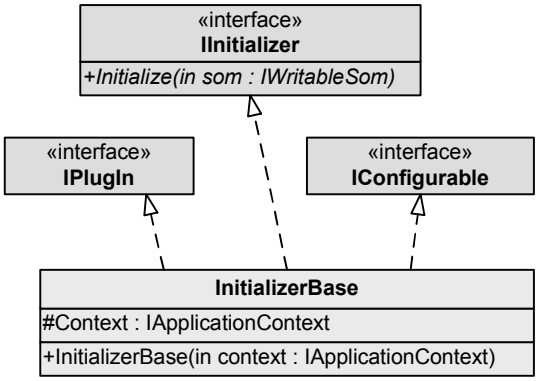

Abbildung 4.5.: Netzinitialisierer bestimmen den Startzustand eines Kohonennetzes

4.2.7. Die Basisklasse für Netzinitialisierer

Vor dem Beginn des Trainings kann eine SOM initialisiert werden, d.h. die Gewichtsvektoren der Ausgabeneuronen werden mit Startwerten belegt. Zu diesem Zweck wird die Schnittstelle `IInitializer` eingeführt, die die Funktionalität eines Netzinitialisierers definiert.

Es ist nur eine Methode beschrieben, die `Initialize` heißt. Sie erwartet als Parameter eine `IWritableSom`-Instanz, einer SOM, die das Ändern ihrer Gewichtsvektoren gestattet (vgl. Kap. 4.2.9). Die Gewichte können also nach Belieben in dieser Methode angepasst werden.

Die Basisklasse für Netzinitialisierer wird `InitializerBase` genannt. Sie implementiert die oben beschriebene Schnittstelle `IInitializer`, sowie `IPlugIn` und, um die Konfigurierbarkeit eines solchen Plugins zu gewährleisten, `IConfigurable`. Auf Abbildung 4.5 ist das entsprechende UML-Diagramm zu sehen.

4.2.8. Die Basisklasse für Visualisierungen

Als Ergebnis der im Kapitel 3.6 geführten Betrachtungen kann eine Schnittstelle definiert werden, die eine Visualisierung beschreibt und mit `IVisualization` bezeichnet wird. Ein UML-Diagramm zeigt das Resultat in Abbildung 4.6.

Die Methode `IsSupported` bestimmt, ob die Visualisierung im gegebenen Kontext anwendbar ist oder nicht. Sie erhält deshalb eine `VisualizationInfo`-Instanz als Parameter.

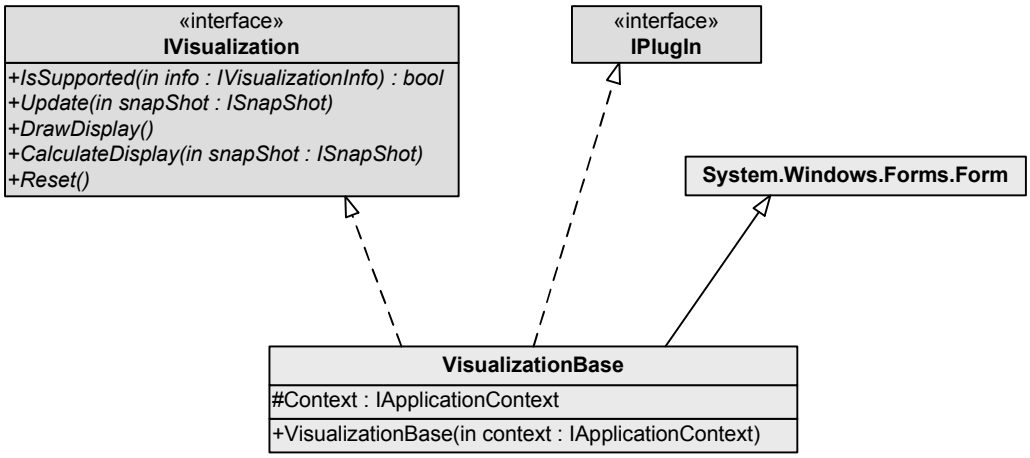

Abbildung 4.6.: Die Basisklasse für Visualisierungen und deren Abhängigkeiten

Die Methode `Update` startet den oben beschriebenen Aktualisierungsprozess mit einem Schnappschuss als Eingabeparameter. `CalculateDisplay` führt die Berechnung der Visualisierung durch und `DrawDisplay` stellt sie dar. Diese beiden Methoden werden in dem Berechnungsprozess (Abbildung 3.11) nacheinander aufgerufen, werden aber auch parallel ausgeführt.

Mit der Methode `Reset` wird der Visualiserung mitgeteilt, dass sie sich zurücksetzen soll. Das tritt beispielsweise auf, wenn eine neue SOM angelegt wurde und die momentan dargestellte Karte ungültig ist.

Die Klasse `VisualizationBase` implementiert als Basisklasse für Visualisierungsplugins die Schnittstellen `IPlugIn` und `IVisualization`. Die Methode `Update` wird bereits so implementiert, dass die gesamte parallele Verarbeitung von der Berechnung der Visualisierung und der anschließenden Darstellung umgesetzt ist. Die zu diesem Zweck eingesetzten Methoden `CalculateDisplay` und `DrawDisplay` werden deshalb virtuell gestaltet und bereits in den Prozess mit einbezogen. Bei der Implementierung einer Visualisierung müssen also nur noch diese beiden virtuellen Methoden überschrieben werden, sie werden dann automatisch bei der Aktualisierung aufgerufen.

Da jede Visualisierung in einem Dialog dargestellt wird, erbt diese Basisklasse noch von `System.Windows.Forms.Form`, der `.NET`-Klasse für Fensterobjekte. So wird gewährleistet, dass die Visualisierung dem Fenster der Hauptanwendung untergeordnet wird und dass die Applikation die vollständige Kontrolle über das Ein- und Ausblenden des Dialogs bekommt.

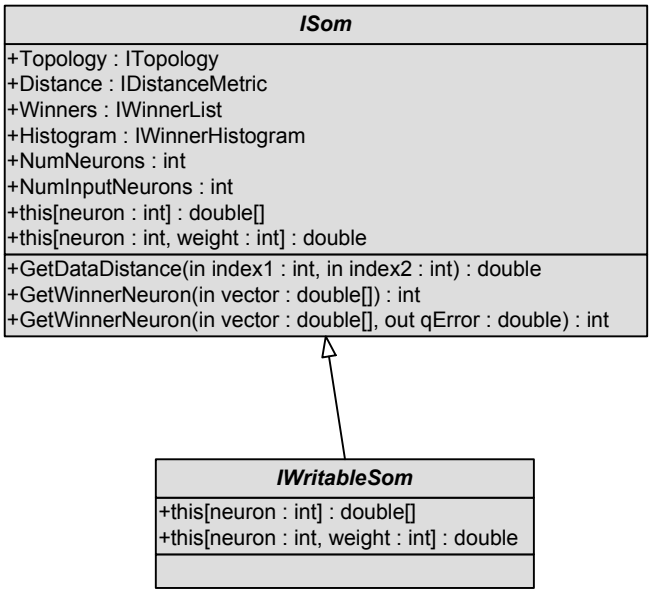

Abbildung 4.7.: Mit Hilfe dieser Schnittstellen wird die SOM in der Erweiterungsschicht bekannt gemacht

4.2.9. Bekanntmachung der SOM

Damit die Plugins Zugriff auf eine SOM bekommen, muss sie in dieser Schicht mittels einiger Schnittstellen bekannt gemacht werden. Welche Eigenschaften erfüllt sein müssen, wurde in Kapitel 3.3 erläutert.

Es werden zwei Schnittstellen definiert, die zwei Ansichten auf die gleiche Karte ermöglichen. Im Normalfall dürfen Plugins die Gewichte der SOM nicht verändern aber trotzdem auf ihre Eigenschaften und Gewichte lesend zugreifen. Zu diesem Zweck wird die Schnittstelle `ISom` eingeführt, die alle Eigenschaften der Karte, wie Topologie, Gewichtsvektoren, Distanzmetrik u.s.w. enthält und nur lesenden Zugriff auf sie erlaubt.

Allerdings exisitert ein Plugin, das die Gewichte der SOM verändern darf, sogar soll. Der Netzinitialisierer braucht schreibenden Zugriff auf die Gewichte der Karte. Deshalb wird eine weitere Schnittstelle vereinbart, die `IWritableSom` genannt wird. Sie erweitert `ISom` um beschreibbare Gewichtsvektoren. Abbildung 4.7 zeigt das UML-Diagramm der hier besprochenen Schnittstellen.

Für die Abfrage der Gewichte der SOM wird ein spezielles Konstrukt der Programmiersprache `C#` genutzt, das es ermöglicht, auf diese Daten wie auf ein Array zuzu-

greifen. Dazu werden in den Schnittstellen Indexer definiert. Beispielsweise sieht dieser Vorgang für die Schnittstelle `IWritableSom` wie folgt aus:

```
// Diese Schnittstelle erlaubt schreibenden Zugriff auf
// die Gewichte der SOM
public interface IWritableSom : ISom
{
  // Zugriff auf den Gewichtsvektor des Neurons mit dem
  // Index "neuron"
  double[] this[int neuron]
  {
    get;
    set;
  }

  // Zugriff auf das Gewicht "weight" des Neurons "neuron"
  double this[int neuron, int weight]
  {
    get;
    set;
  }
}
```

Es werden zwei Eigenschaften mit dem Schlüsselwort `this` definiert, auf die lesend und schreibend zugegriffen werden kann (zu erkennen an `get` und `set`). Diese Eigenschaften weisen unterschiedliche Indexer auf. Die erste erwartet einen Integer-Wert, der den Index des untersuchten Neurons darstellt. Als Rückgabewert wird ein `double`-Array festgelegt, das einen Gewichtsvektor repräsentiert. Die zweite Eigenschaft erwartet zwei Indizes, einen für das Neuron und einen für das Gewicht. Deshalb gibt sie auch nur einen `double`-Wert zurück, der das gewünschte Gewicht enthält.

Auf diese Eigenschaften kann wie folgt zugegriffen werden:

```
IWritableSom som = ...;
```

```
// Zugriff auf den Gewichtsvektor des Neurons 5:
double[] gewichte = som[5];

// Zugriff auf Gewicht 2 des Neurons 5:
double gewicht = som[5, 2];
```

4.3. Datenzugriffsschicht

Zur Implementierung der Datenzugriffsschicht, deren Hauptaufgabe die Verwaltung externer Ressourcen ist, wird der Namensraum `Sombrero.DataAccess` eingeführt, in den alle für diesen Zweck entworfenen Klassen eingebettet werden.

4.3.1. Logging

In der Erweiterungsschicht ist die Schnittstelle `ILogger` definert, die die Grundfunktionalität eines *Loggers* beschreibt. Ein solcher Logger wird genutzt, um bestimmte Abläufe oder aufgetretene Fehler zu dokumentieren, indem sie z.B. in eine Textdatei geschrieben werden. Eine Methode mit mehreren Überladungen namens `WriteLine` nimmt beispielsweise Zeichenketten und weitere Argumente als Parameter entgegen, die im „Logbuch“ gespeichert werden sollen.

Die Klasse `Logger` im Namensraum `Sombrero.DataAccess.Logging` implementiert diese Schnittstelle tatsächlich derart, dass übergebene Zeichenketten zusätzlich mit dem Zeitstempel des Auftretens in ein Textdokument geschrieben werden.

An den Konstruktor der Klasse wird der Pfad zur Zieldatei übergeben, in die der während der Laufzeit entstehende Text abgelegt wird. Es existieren zwei statische Eigenschaften namens `ErrorLog` und `OutputLog`, die die Standardlogger für die Fehlerdokumentation und sonstige Ausgaben darstellen. Sie speichern ihre Texte jeweils in die Dateien `error.log` und `output.log` im Wurzelverzeichnis der Applikation. Diese Logfiles sind für alle Interessenten auch über den Applikationskontext zugänglich, deshalb können auch alle Plugins Informationen in ihnen ablegen.

4.3.2. Verwaltung von Plugins

Die Verwaltung von Plugins ist ein Bestandteil der Datenzugriffsschicht, da externe Laufzeitbibliotheken dynamisch nachgeladen und enthaltene Klassen instanziert werden.

Die Bibliotheken und die darin befindlichen Plugins müssen bei der Applikation angemeldet werden. Für diese Aufgabe wird eine XML-Datei namens `plugins.xml` verwendet, die sich im Ordner `plugins` unter dem Hauptverzeichnis des Programms befindet. Die eingebundenen Plugin-DLL sollten sich ebenfalls in diesem Ordner befinden.

Die Struktur des XML-Dokuments ist vorgegeben, es wird wie folgt aufgebaut:

```
<?xml version="1.0" encoding="utf-8" ?>
<plugins>
  <assembly filename="plugins\MyPlugIn.dll">
    <!-- Data Access Layer -->
    <!-- Data Sources -->
    <plugin typename="MyPlugIn.DataAccess.ArffDataSource"/>
    <plugin typename="MyPlugIn.DataAccess.CustomDataSource"/>
    <!-- Standardizers -->
    <plugin typename="MyPlugIn.DataAccess.BasicStandardizer"/>

    <!-- Business Logic Layer -->
    <!-- Distance Metrics -->
    <plugin typename="MyPlugIn.BusinessLogic.EuclideanDistance"/>
    <plugin typename="MyPlugIn.BusinessLogic.ManhattanDistance"/>
    <!-- Topologies -->
    <plugin typename="MyPlugIn.BusinessLogic.Matrix2DTopology"/>
    <plugin typename="MyPlugIn.BusinessLogic.TorusTopology"/>
    <!-- Neighborhood Functions -->
    <plugin typename="MyPlugIn.BusinessLogic.GaussianFunction"/>
    <plugin typename="MyPlugIn.BusinessLogic.MexicanHatFunction"/>
    <plugin typename="MyPlugIn.BusinessLogic.BubbleFunction"/>
    <plugin typename="MyPlugIn.BusinessLogic.ConeFunction"/>
    <plugin typename="MyPlugIn.BusinessLogic.CosineFunction"/>
```

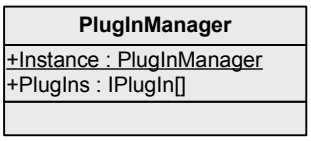

Abbildung 4.8.: Diese Klasse verwaltet die angemeldeten Plugins

```
    <!-- Initializers -->
    <plugin typename="MyPlugIn.BusinessLogic.RandomInitializer"/>

    <!-- GUI Layer -->
    <!-- Visualizations -->
    <plugin typename="MyPlugIn.Gui.WinnerHistogram"/>
    <plugin typename="MyPlugIn.Gui.MatrixInInputSpace"/>
    <plugin typename="MyPlugIn.Gui.WinnerMatrix"/>
    <plugin typename="MyPlugIn.Gui.UMatrix"/>
  </assembly>
</plugins>
```

Unter dem Wurzelknoten `<plugins>` existiert für jede eingebundene DLL ein Knoten namens `<assembly>`, dessen `filename`-Attribut auf die jeweilige Bibliothek verweist. Die implementierten Plugins werden in den darunter liegenden `<plugin>`-Knoten angemeldet. Das `typename`-Attribut enthält den Namen der Klasse, die von einem der Plugin-Basisklassen erbt, inklusive der umschließenden Namensräume.

Bei der Instanzierung der angegebenen Klassen wird automatisch erkannt, um welch einen Plugintypen es sich handelt. Die dargestellten Kommentare im XML-Dokument dienen nur der Übersicht.

Die Verarbeitung der XML-Datei und die Bereitstellung der Plugininstanzen übernimmt die Klasse `PlugInManager`. Sie erfüllt das *Singleton*-Entwurfsmuster, d.h. es existiert nur eine einzige Instanz dieser Klasse, die über eine statische Eigenschaft erreichbar ist. Der Konstruktor ist deshalb privat, also außerhalb der Klasse nicht sichtbar. Abbildung 4.8 zeigt das UML-Diagramm der `PlugInManager`-Klasse.

Beim erstmaligen Aufruf der statischen Eigenschaft `Instance` wird die eine Instanz der Klasse erzeugt. Dabei wird automatisch die Datei `plugins.xml` geöffnet und ausge-

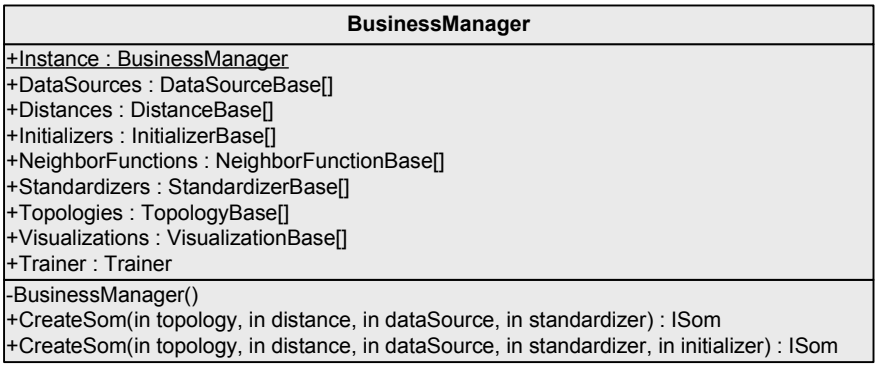

Abbildung 4.9.: Der Business-Manager verwaltet die Geschäftslogikschicht

wertet, die angemeldeten Plugins werden erstellt. Hier zeigt sich, warum die Konstruktoren der Plugins die gleiche Signatur haben müssen, wie die abgeleitete Basisklasse: Es wird immer ein Konstruktor erwartet, der den Applikationskontext als Parameter aufnimmt. Sollten daher Fehler in Form von ungültigen Konstruktoren oder Ähnlichem auftreten, die eine Instanzierung verhindern, wird automatisch eine entsprechende Information im Error-Logfile abgelegt.

Die Eigenschaft `PlugIns` enthält alle geladenen Plugins in Form eines Arrays aus `IPlugIn`-Instanzen. Eine Zuordung der unterschiedlichen Typen ist an dieser Stelle noch nicht erfolgt, diese Aufgabe wird der Geschäftslogikschicht zugeteilt.

4.4. Geschäftslogikschicht

Die Geschäftslogikschicht wird im Namensraum `Sombrero.BusinessLogic` untergebracht und wird hauptsächlich von der Klasse `BusinessManager` verwaltet. Durch sie haben die anderen Schichten Zugriff auf die Geschäftslogik.

4.4.1. Der Business-Manager

Die Klasse `BusinessManager` verwaltet die Geschäftslogikschicht. Sie ist der Hauptanlaufpunkt für die Klassen anderer Schichten, die Informationen benötigen oder Aufgaben für die Geschäftslogik haben. Abbildung 4.9 zeigt das UML-Diagramm dieser Klasse.

Auch der `BusinessManager` verwendet das *Singleton*-Entwurfsmuster. Die einzige Instanz der Klasse ist über die statische Eigenschaft `Instance` verfügbar, der Konstruktor ist nicht öffentlich.

Beim ersten Aufruf der `Instance`-Eigenschaft verbindet sich der `BusinessManager` mit dem `PlugInManager` aus der Datenzugriffsschicht. Er übernimmt die angemeldeten Plugins und bestimmt die jeweilig implementierten Typen. Diese Plugins sind im Anschluss über die Eigenschaften `DataSources`, `Standardizers`, `NeighborFunctions`, `Distances`, `Topologies` und `Visualizations` verfügbar, die jeweils alle angemeldeten Plugins des entsprechenden Typs in einem Array halten.

Mit der Eigenschaft `Trainer` lässt sich eine Instanz der für den Lernprozess verantwortlichen Klasse `Trainer` ansprechen, die in Kapitel 4.4.3 erläutert wird.

Über die Methode `CreateSom` wird eine neue Instanz einer selbstorganisierenden Karte erstellt. Die hierfür benötigten Informationen, wie verwendete Netztopologie, Distanzmetrik, Datenquelle und Normierer, werden als Parameter erwartet und außerdem automatisch im Applikationskontext abgelegt (mehr dazu im nächsten Abschnitt). Die Methode ist überladen, es existiert eine weitere Variante, die auch einen Netzinitialisierer als Parameter annimmt. Mit ihm werden die Gewichte der Karte nach der Instanzierung initialisiert.

4.4.2. Der Applikationskontext

Die Klasse `ApplicationContext` implementiert den Applikationskontext, der durch die Schnittstelle `IApplicationContext` aus der Plugin-Schicht definiert ist. Es werden noch weitere Funktionalitäten hinzugefügt, die nicht von der Schnittstelle abgedeckt sind und deshalb nur von der Hauptapplikation genutzt werden können.

Der gesamte Lernkontext wird hier verwaltet. Die aktuelle Instanz der SOM, die Lernparameter, die Datenquelle und der Normierer werden jeweils nach ihrer Instanzierung hier abgelegt, dazu werden die Eigenschaften `Som`, `LearnParams`, `DataSource` und `Standardizer` verwendet. Diese Informationen werden für einen Schnappschuss (siehe Kapitel 3.5.2) gebraucht. Dieser wird mit Hilfe der Methode `GetSnapShot` erzeugt.

Die Klasse enthält den Pool für alle von dem Programm verwalteten Threads. Über die Methoden `AbortThread` und `AbortAllThreads` werden entweder einzelne Threads

oder der gesamte Pool abgebrochen. Diese Aktion ist zum Beenden der Applikation notwendig, um zu vermeiden, dass einzelne Funktionsstränge auch danach noch weiterlaufen.

4.4.3. Der Trainingsprozess

Um den Trainingsablauf kümmern sich mehrere Klassen, die den untergeordneten Namensraum `Learning` für sich in Anspruch nehmen. Die Lernparameter werden in der Klasse `LearnParams` gekapselt. Der Trainingsprozess selbst wird durch die Klasse `Trainer` implementiert.

Die Lernparameter

Die Struktur der Lernparameter ist in der Erweiterungsschicht durch die Schnittstelle `ILearnParams` vorgegeben, die natürlich auch von der Klasse `LearnParams` implementiert wird. Einige Aspekte kommen in dieser Schnittstelle aber nicht zur Geltung, da sie im Kontext der Plugins nicht von Bedeutung sind. Dabei handelt es sich um die Abbruchbedingung des Trainings und um die Reduktion der Lernrate und des Lernradius, die nach jedem Lernschritt stattfinden.

Die Abbruchbedingung kann vielfältig gestaltet sein, deshalb wird zuerst eine neue Schnittstelle namens `ICompletion` eingeführt, die sie repräsentiert. Die definierte Methode `IsCompleted` liefert einen Booleschen Wert, der angibt, ob die enthaltene Bedingung erfüllt ist. Als Parameter erhält sie den bereits bekannten Schnappschuss des Lernkontexts, aus dem sich alle nötigen Informationen ziehen lassen. Es werden zwei mögliche Abbruchbedingungen implementiert, das Erreichen einer gewissen Lernschrittzahl und das Unterschreiten eines bestimmten Quantisierungsfehlers. Diese Bedingungen werden mit den Klassen `LearnStepCompletion` und `QErrorCompletion` umgesetzt.

Bei der Reduktion der Lernrate und des Lernradius handelt es sich mathematisch gesehen um die gleiche Aufgabe, daher wird nur eine Struktur erzeugt. Davon werden im Anschluss zwei Instanzen gebildet, die sich jeweils um einen der Parameter kümmern. Es wird wiederum eine Schnittstelle für diesen Zweck eingesetzt, die `IReduction` genannt wird. Darin wird eine `Minimum`-Eigenschaft definiert, die den minimalen erlaubten Wert des reduzierten Parameters enthalten soll. Die Methode `Reduce` dekrementiert den übergebenen Realzahl-Parameter und gibt ihn zurück. Es gilt die Formel 4.1.

$$red : \Re \times \Re \mapsto \Re;\ red(x, min) \geq min;\ x, min \in \Re \tag{4.1}$$

Die Schnittstelle wird durch zwei Klassen implementiert, die eine lineare und eine exponentielle Reduktion ermöglichen. Die lineare Reduktion wird durch die Subtraktion eines konstanten positiven Wertes beschrieben. Die exponentielle Reduktion wird durch die Multiplikation mit einem konstanten Faktor ρ, für den $0 \leq \rho \leq 1$ gilt, realisiert. Die entsprechenden Klassen tragen die Namen `LinReduction` und `ExpReduction`. Das eingesetzte Minimum für die Lernrate beträgt 0, das für den Lernradius ist bei 1 festgesetzt.

Aufgrund dieser Betrachtungen wird die Klasse `LearnParams` erweitert um die Eigenschaften `Completion`, `RateReduction` und `RadiusReduction`. Zudem werden die Methoden `ReduceRate` und `ReduceRadius` hinzugefügt, die die Berechnung der Reduktionen über die jeweilige Eigenschaft durchführen.

Der Trainer

Die Klasse `Trainer` implementiert den im Theorie- und Konzeptionsteil (Kapitel 2.2.3 und 3.5.3) beschriebenen Lernprozess. Zudem enthält sie Methoden, die den Ablauf des Trainings beeinflussen. Es werden Ereignisse definiert, die beim Erreichen bestimmter Zustände ausgelöst werden.

Der `Trainer` enthält vier Methoden, die Einfluss auf den Lernprozess nehmen. `Start` initiiert den Vorgang. Sie nimmt die Karte, Lernparameter, die verwendete Datenquelle und den Normierer als Parameter entgegen. Ein paralleler Arbeiter-Thread wird gestartet und das Training wird abgearbeitet. Die Methode `Pause` unterbricht den Lernvorgang. Der Ablauf wird aber nicht einfach angehalten, es wird vorher noch auf die Beendigung des aktuellen Lernschritts gewartet. Mit `Continue` kann das Training anschließend fortgesetzt werden. Die Methode akzeptiert neue Lernparameter, diese dürfen also während einer Unterbrechung noch angepasst werden. `Stop` beendet das Training, es kann anschließend nicht fortgesetzt sondern muss neu gestartet werden. Diesen Zustand nimmt der Trainer ebenfalls ein, wenn die Abbruchbedingung unter den Lernparametern das Ende des Trainings bekannt gibt.

Die verschiedenen Zustände des Trainers sind über die Eigenschaft `State` zu erfragen. Als Rückgabewert dieser Eigenschaft wurde `TrainerState` eingeführt, eine Enu-

meration, die die Werte `NotStarted`, `Running`, `Paused` und `Stopped` annehmen kann.

Die umgesetzten Events tragen die Namen `StepComplete` und `TrainingComplete`. Sie werden ausgelöst, sobald ein Lernschritt beendet bzw. wenn das Training abgeschlossen ist. Klassen, die auf diese Ereignisse reagieren möchten, müssen eine Methode implementieren, die konform zu dem im Umfeld des Trainers definierten Delegaten `TrainingEventHandler` ist und diese bei der eingesetzten `Trainer`-Instanz anmelden. Wird ein solches Ereignis ausgelöst, erhalten alle angemeldeten Eventhandler einen Schnappschuss des aktuellen Lernkontexts, der durch die Schnittstelle `ISnapShot` definiert ist.

4.5. Präsentationsschicht

Die Präsentationsschicht stellt das Hauptfenster der Applikation und einige weitere Dialoge zur Verfügung. Sie ist im Namensraum `Sombrero.Gui` untergebracht. Die Nebendialoge befinden sich im untergeordneten Namensraum `Dialogs`.

4.5.1. Das Hauptfenster

Sombrero wird als *MDI*-Applikation (*Multi Document Interface*) implementiert, d.h. es können mehrere Unterfenster gleichzeitig geöffnet und parallel bearbeitet werden. In diesem Fall wird es ermöglicht, mehrere Visualisierungen zu öffnen und gleichzeitig die Lernparameter zu bearbeiten und das Training zu überwachen.

Im Hauptfenster (implementiert durch die Klasse `MainForm`) ist eine Menüleiste integriert, die den Zugriff auf die Funktionalitäten und Dialoge des Programms ermöglicht. Der Menüpunkt „SOM“ enthält die folgenden Unterpunkte:

Neu legt eine neue Karte an. Zu diesem Zweck wird ein Dialog geöffnet, mit dem alle nötigen Einstellungen vorgenommen werden können. Dieser Vorgang wird in Kapitel 4.5.2 näher beschrieben.

Speichern und Öffnen sollen den aktuellen Zustand der SOM und aller eingesetzten Plugins auf einen Datenträger sichern oder einen abgespeicherten Zustand wiederherstellen.

Beenden schließt das Programm mit allen geöffneten Visualisierungen und beendet den Trainingsprozess.

Der Menüpunkt „Ansicht“ erlaubt es, die Standarddialoge und eingebundenen Visualisierungen ein- und auszublenden. Die Unterpunkte lauten wie folgt:

Data Inspector öffnet ein Fenster, das den aktuell ausgewählten Datenpool anzeigt. Kapitel 4.5.3 befasst sich mit diesem Dialog.

Trainer öffnet oder schließt den Dialog, mit dem das Training der Karte überwacht wird. Dies ist ein Standarddialog, der beim Programmstart automatisch eingeblendet ist. Eine nähere Beschreibung findet sich im Kapitel 4.5.5.

Lernparameter blendet den Lernparameterdialog (betrachtet in Kapitel 4.5.4) ein oder aus. Dieses Fenster ist ebenfalls standardmäßig beim Programmstart sichtbar.

Unter den oben genannten Punkten sind noch Einträge für alle angemeldeten Visualisierungen vorhanden, die deren Öffnen und Schließen ermöglichen. Unterstützt eine Visualisierung den aktuellen Kontext (bestehend aus Topologie, Datenquelle und Normierer) nicht, so wird der entsprechende Menüpunkt deaktiviert.

Da sie die eingebundenen Visualisierungen als hierarchisch untergeordnete Dialoge verwaltet, ist die Klasse für die Aktualisierung dieser Plugins verantwortlich. Deshalb reagiert das Hauptfenster auf Ereignisse, die vom `Trainer` ausgelöst werden. Wenn das Event `StepComplete` auftritt, wird automatisch die `Update`-Methode aller aktivierten Visualisierungsplugins aufgerufen.

4.5.2. Der „Neue SOM“-Dialog

Der Dialog „Neue SOM anlegen“ (zu sehen in Abbildung 4.10) dient der Erstellung einer neuen selbstorganisierenden Karte und wird mit der Klasse `InitSomDialog` implementiert. Es stehen zwei Reiter zur Verfügung, auf denen die Einstellungen bezüglich der Struktur der SOM sowie der gekoppelten Datenquelle vorgenommen werden.

Auf der „SOM“-Karteikarte werden die Netztopologie, die verwendete Distanzmetrik sowie ein Netzinitialisierer, der optional ist und mit einer Checkbox deaktiviert werden kann, gewählt. Alle diese Elemente werden durch Plugins zur Verfügung gestellt, daher

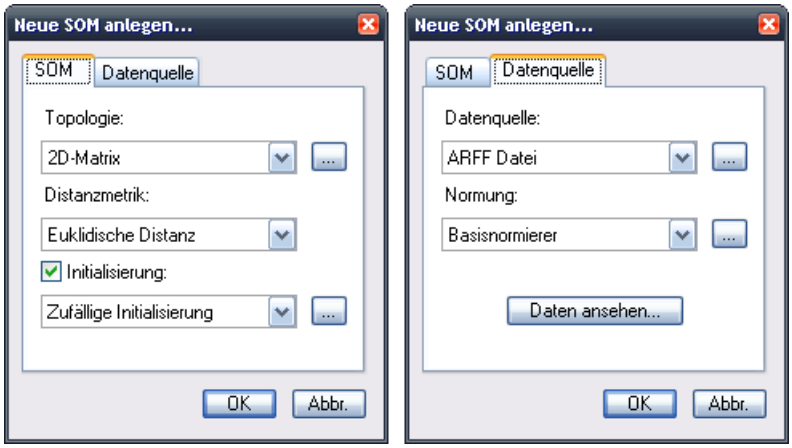

Abbildung 4.10.: Der Dialog „Neue SOM anlegen“ konfiguriert den Kontext zur Erstellung einer Karte: links der strukturelle Aufbau, rechts die Datenquelle und der Normierer

werden alle erfolgreich angemeldeten Plugins dieser Typen vom `BusinessManager` geholt und in den dafür vorgesehenen Klappboxen aufgelistet. Rechts neben den Klappboxen befinden sich je eine Schaltfläche mit der Aufschrift „ ... “. Mit ihr lässt sich der Konfigurationsdialog des in der Klappbox gewählten Plugins öffnen, um dessen Parameter anzupassen. Sollte ein Plugin nicht konfigurierbar sein (erkennbar durch die Eigenschaft `IsConfigurable`), so wird die Schaltfläche deaktiviert. Distanzmetriken-Plugins sind generell so aufgebaut, dass sie die Schnittstelle `IConfigurable` nicht implementieren, daher ist dort keine Schaltfläche zur Konfiguration vorgesehen.

Unter dem Reiter „Datenquelle“ werden die Plugins für die Datenquelle und den Normierer auf gleiche Weise ausgewählt. Zusätzlich existiert eine Schaltfläche „Daten ansehen“. Wird diese aktiviert, öffnet sich der *Data Inspector*, mit dem sich der Datenpool aus der aktuell gewählten Datenquelle betrachten lässt (hierzu mehr im nächsten Abschnitt).

Die Schaltfläche „Abbr.“ schließt den Dialog ohne Änderungen am Kontext des Programms vorzunehmen. Beim Klick auf die Schaltfläche „OK“ wird zuerst bei allen ausgewählten Plugins überprüft, ob sie korrekt konfiguriert sind (über die Eigenschaft `IsConfigured`). Sollte dies nicht der Fall sein, wird das entsprechende Plugin gelb hinterlegt und der Dialog bleibt geöffnet. Ansonsten wird der Dialog geschlossen. Die gewählten Plugins werden an den `BusinessManager` übergeben und mit Hilfe der Methode `CreateSom` wird eine neue Karte angelegt.

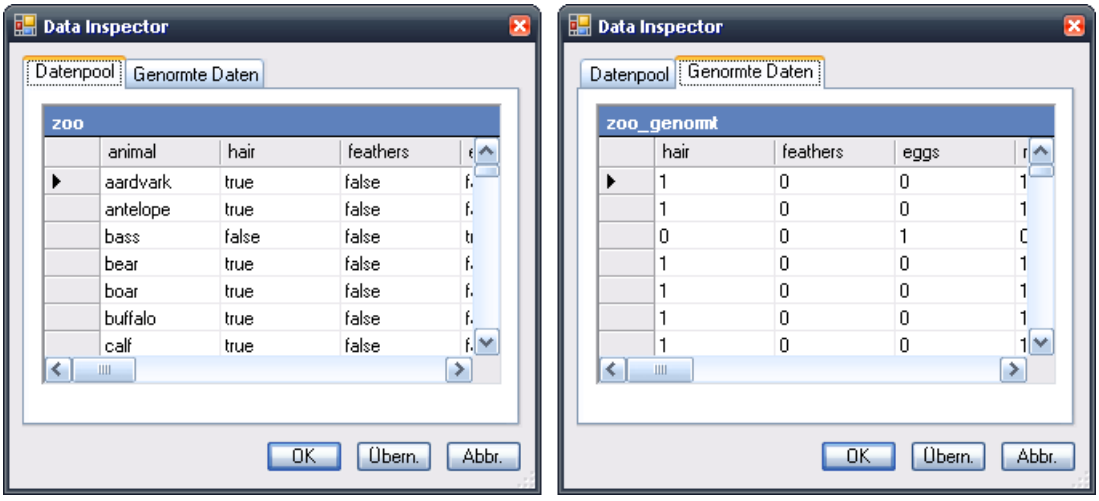

Abbildung 4.11.: Der Data Inspector zeigt den Datenpool der gewählten Quelle und die genormten Werte an

Die Plugins werden auch genutzt, um eine Instanz der Klasse `VisualizationInfo` zu erzeugen. Damit wird bei allen angemeldeten Visualisierungen überprüft, ob sie den aktuellen Kontext unterstützen. Dazu wird die Methode `IsSupported` aufgerufen und anhand des Rückgabewertes werden die entsprechenden Menüeinträge aktiviert oder deaktiviert. Nicht unterstützte Visualisierungen, die zu diesem Zeitpunkt geöffnet sind, werden automatisch geschlossen.

Die neue SOM wird zusammen mit der Datenquelle und dem Normierer in den Applikationskontext geschrieben, damit die Umgebung auf den aktuellen Stand gebracht wird.

4.5.3. Der Data Inspector

Der *Data Inspector*-Dialog (Abbildung 4.11) wird durch die Klasse `DataInspector` implementiert. Seine Aufgabe ist das Inspizieren des aktuellen Datenpools sowie der genormten Daten. Zu diesem Zweck verfügt er über zwei Reiter, treffend mit „Datenpool“ und „Genormte Daten“ bezeichnet.

Unter dem Reiter „Datenpool“ befindet sich eine Tabelle, die die durch die Datenquelle erzeugte Relation offenbart. Jede Spalte repräsentiert eines der betrachteten Attribute, jede Zeile enthält eine Dateninstanz. Der Reiter „Genormte Daten“ zeigt die Relation in genormter Form in einer weiteren Tabelle. Es besteht die Möglichkeit, dass sich - je

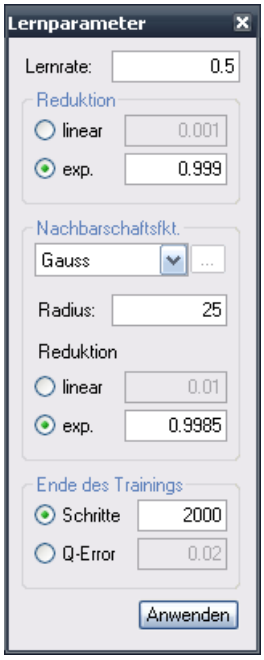

Abbildung 4.12.: Der Lernparameterdialog zur Konfiguration des Trainingsablaufs

nach Konfiguration des Normierers - die Anzahl der Spalten von denen des Datenpools unterscheidet. In der Abbildung ist z.B. zu erkennen, dass das Namensattribut „animal“ der geladenen Relation „zoo“ bei der Normung entfällt, es ist in der genormten Tabelle nicht mehr vorhanden.

Einige Datenquellen erlauben es, die Werte im Datenpool zu manipulieren. Zu diesem Zweck existieren die Booleschen Eigenschaften `IsAddAllowed`, `IsDeleteAllowed` und `IsEditAllowed`, die der Reihe nach bestimmen, ob das Hinzufügen einer neuen Instanz, das Löschen einer Instanz oder das Bearbeiten der Attribute einer Instanz gestattet ist. Sollte eine dieser Operationen erlaubt sein, kann sie auch mit dem *Data Inspector* durchgeführt werden - aber ausschließlich unter dem „Datenpool“-Reiter.

4.5.4. Der Lernparameterdialog

Die Klasse `LearnParamDialog` implementiert den in Abb. 4.12 dargestellten Lernparameterdialog. Dieser wird zur Festlegung der Lernparameter für den Trainingsprozess eingesetzt.

Das Eingabefeld am oberen Rand des Dialogs dient der Festlegung der Lernrate η,

für die $0 \leq \eta \leq 1$ gilt. Im Bereich darunter kann zwischen den Reduktionsarten für η gewählt werden. Mit den Radiobuttons wird also die für die Lernrate eingesetzte `IReduction`-Implementierung bestimmt: `LinReduction` oder `ExpReduction`. Das Eingabefeld hinter den Radiobuttons legt den für die Reduktionsberechnung eingesetzten Wert fest, also den Subtrahenden für die lineare, den Faktor für die exponentielle Reduktion.

Der darunter liegende Bereich befasst sich mit der Nachbarschaftsfunktion und dem zugehörigen Lernradius r. Die Klappbox enthält die Namen aller angemeldeten Plugins, die Nachbarschaftsfunktionen implementieren. Das Plugin mit dem gewählten Namen wird für die Berechnungen eingesetzt. Es besteht die Möglichkeit, eine konfigurierbare Nachbarschaftsfunktion zu integrieren, deshalb ist rechts von der Klappbox noch ein „...“-Button angebracht. Mit diesem wird der Konfigurationsdialog eines solchen Plugins geöffnet. Im Eingabefeld darunter wird der Lernradius festgelegt. Die Reduktionsarten für r werden genauso bestimmt wie bei der Lernrate.

Im Bereich „Ende des Trainings“ wird die Abbruchbedingung für den Lernvorgang bestimmt. Mit den Radiobuttons wird zwischen „Schritte“ und „Q-Error“ gewählt. Es wird also entschieden, ob `LearnStepCompletion` oder `QErrorCompletion` als Abbruchbedingung eingesetzt werden. Der Parameter auf der rechten Seite bestimmt die Anzahl der zu durchlaufenden Lernschritte oder den Quantisierungsfehler, den es zu unterschreiten gilt.

Die Schaltfläche „Anwenden“ übernimmt die getroffenen Einstellungen. Es wird eine `LearnParams`-Instanz erzeugt, an die die Werte und die gewählten Algorithmen aus dem Dialog übergeben werden. Das Objekt wird anschließend an den Applikationskontext übermittelt. Sollte ein Parameter nicht den Anforderungen entsprechen (z.B. eine Lernrate größer als 1), so wird das entsprechende Eingabefeld gelb hinterlegt und die Werte werden nicht übernommen.

4.5.5. Der Trainingsdialog

Der Trainingsdialog, der durch die Klasse `TrainingDialog` implementiert wird, ist in Abbildung 4.13 dargestellt. Mit ihm lässt sich der Ablauf des Lernprozesses beeinflussen und überwachen. Sein Aussehen ist an Software zur Medienwiedergabe angelehnt: Es existieren Schaltflächen zum Starten und zum Beenden des Trainingsvorgangs, die

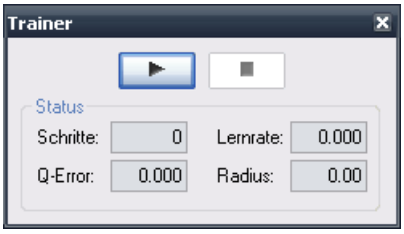

Abbildung 4.13.: Der Trainingsdialog startet, pausiert und beendet den Lernprozess

einem Wiedergabe- und einem Stopp-Knopf nachempfunden sind. Darunter befindet sich ein Statusbereich, der aktuelle Werte zur Überwachung des Trainings enthält. Dabei handelt es sich um die Lernrate, den Lernradius, die absolvierten Lernschritte und den Quantisierungsfehler.

Der Dialog ist nach dem Start der Applikation inaktiv, er ist erst nach dem Anlegen einer SOM bedienbar. Mit dem Startknopf wird der Lernprozess initiiert. Die Karte, die Datenquelle, der Normierer und die Lernparameter werden aus dem Applikationskontext geholt. Der `BusinessManager` hält eine `Trainer`-Instanz bereit, bei der anschließend die `Start`-Methode aufgerufen wird. Der Dialog reagiert auf die Events „Lernschritt beendet“ und „Lernprozess beendet“, die Statuswerte werden beim Auslösen dieser Ereignisse aktualisiert und bleiben so auf neuestem Stand.

Während des Lernprozesses wird der Startknopf zu einem Pauseknopf umfunktioniert. So ist es möglich, das Training zu unterbrechen indem `Trainer.Pause` aufgerufen wird. In diesem Zustand können die Lernparameter im entsprechenden Dialog angepasst werden. Wenn anschließend das Training fortgesetzt wird, gelten die neuen Werte. Der Pause-Modus eignet sich außerdem dazu, die Visualisierungen zu synchronisieren oder sich den Zwischenstand des Trainings anzusehen. Der Stoppknopf beendet den Lernvorgang über `Trainer.Stop`.

4.6. Entwicklung von Plugins

Das Programm wird noch mit Erweiterungen ausgestattet, damit sowohl die Grundfunktionalitäten mit Hilfe von Datenquellen und Topologien als auch einige Visualisierungen zur Verfügung gestellt werden. Es wird also eine Laufzeitbibliothek namens `DefaultPlugIn` erstellt, die alle hier implementierten Plugins enthält.

In der Tabelle 4.1 werden alle umgesetzten Erweiterungen deren Basisklassen zugeordnet. Die Interessantesten werden im Anschluss noch näher erläutert.

4.6.1. ARFF-Datei als Datenquelle

Eine entwickelte Datenquelle erlaubt den Zugriff auf ARFF-Dateien, um die darin beschriebene Relation zum Training der Karte einzusetzen. Dies ist wohl die gebräuchlichste Datenquelle, da Trainingsdaten für dieses Format sehr verbreitet sind. Eine solche Datei ist ein Textdokument, dass wie folgt aufgebaut ist:

```
@relation Städte

@attribute Name {Aachen, Berlin, Bremen, Chemnitz, ...}
@attribute X    NUMERIC
@attribute Y    NUMERIC

@data
Aachen, 1.5, 18.8
Berlin, 20.7, 11.5
Bremen, 9, 9.1
Chemnitz, 19.8, 18.8
...
```

Die abgebildete Relation trägt den Namen `Städte`, es werden drei Attribute `Name`, `X` und `Y` beschrieben. Das Attribut `Name` ist nominal. Es soll die Namen von deutschen Städten repräsentieren. Alle im abgebildeten Dokument genutzten Städte sind als Werteliste hinter dem Attribut aufgeführt. `X` und `Y` enthalten numerische Werte, in diesem Fall Koordinaten. Nach der Kennzeichnung `@data` beginnt der Datenblock, in dem jede Zeile eine Dateninstanz repräsentiert. Die Städte und zugehörigen Koordinaten werden durch Kommas getrennt aufgelistet.

Um eine ARFF-Relation im Programm abzubilden, werden einige Klassen angelegt, die zum Einen eine solche Relation auslesen und sie zum Anderen in Datenstrukturen nachahmen. Die zum Auslesen entworfenen Klassen befinden sich im Unternamensraum `Arff`, die Strukturnachbildungen in `Arff.Format`.

Basisklasse	Plugin	Plugin-Klasse
`DataSourceBase`	ARFF-Datenquelle	`ArffDataSource`
	ARFF-Datenstrom	`ArffStreamDataSource`
	benutzerdefinierte Datenquelle	`CustomDataSource`
`StandardizerBase`	Basisnormierer	`BasicStandardizer`
`DistanceBase`	Euklidische Distanz	`EuclideanDistance`
	Manhattan-Distanz	`ManhattanDistance`
`NeighborFuncBase`	Bubble-Funktion	`BubbleFunction`
	Kegelfunktion	`ConeFunction`
	Kosinusfunktion	`CosineFunction`
	Gauß-Funktion	`GaussianFunction`
	mexican hat-Funktion	`MexicanHatFunction`
`TopologyBase`	quadratisches Gitter	`Matrix2DTopology`
	Torus	`TorusTopology`
	hexagonales Gitter	`HexagonTopology`
	hexagonaler Torus	`HexTorusTopology`
`InitializerBase`	Zufallsinitialisierung	`RandomInitializer`
`VisualizationBase`	Komponentenmatrix	`ComponentMatrix`
	Gewinnhistogramm	`HitHistogram`
	L-Matrix	`LMatrix`
	Karte im Eingaberaum	`MapInInputSpace`
	U-Matrix	`UMatrix`
	kumul. Gewinnhistogramm	`WinnerHistogram`
	Gewinnermatrix	`WinnerMatrix`

Tabelle 4.1.: Die im Laufe der Implementierung umgesetzten Plugins, ihren Basisklassen zugeordnet

Die Klasse `ArffRelation` bildet eine solche Struktur in ihrer Gesamtheit nach. Sie enthält alle Informationen, die in einer ARFF-Relation vorhanden sind. Die Eigenschaft `Name` enthält den Namen einer Relation, `Attributes` spiegelt in einem Array die verwendeten Attributtypen wieder. Unter `Data` sind die Dateninstanzen der Relation abgelegt.

Drei Arten von ARFF-Attributen werden unterstützt, die numerischen, nominalen und String-Attribute. Jeder Typ wird durch eine der drei Klassen `NumericAttribute`, `NominalAttribute` und `StringAttribute` beschrieben. Sie erben von der abstrakten Klasse `ArffAttribute`. Die drei Attributstrukturen unterscheiden sich kaum voneinander, sie alle besitzen die Eigenschaft `Name`, den Namen des Attributs. Nur die Klasse `NominalAttribute` wird durch eine weitere Eigenschaft ergänzt: `Values` enthält eine Liste der gültigen Werte für den nominalen Typ.

Mit der Klasse `ArffParser` lässt sich eine ARFF-Datei öffnen, auslesen und in die `ArffRelation`-Programmstruktur überführen. Hierzu wird an den Konstruktor einfach der Dateipfad und die Kodierung der Datei übergeben. Anschließend wird durch Aufruf der Methode `Parse` eine Instanz der Klasse `ArffRelation` erzeugt, die alle ausgelesenen Informationen, wie Attributtypen und Daten, enthält.

Eine Datenquelle, die mit diesen Strukturen umgeht, wird implementiert, indem die Klasse `ArffDataSource` von `DataSourceBase` erbt. Das Plugin verfügt über einen konfigurierbaren Parameter, das ist die zu verwendende ARFF-Datei. Im Konfigurationsdialog lässt sich diese Datei nebst Textkodierung auswählen. Sie wird anschließend an den `ArffParser` weitergegeben, der die abgebildete ARFF-Relation erzeugt. Die enthaltenen Attribute werden in `ColumnTypes` überführt, die Dateninstanzen werden in den Datenpool geschrieben.

4.6.2. Der Basisnormierer

Der Basisnormierer stellt die Funktionalität zur Verfügung, die Attribute der Dateninstanzen aus der Datenquelle in reelle Zahlenvektoren zu überführen, die als Eingabevektoren der SOM fungieren. Die Klasse `BasicStandardizer` erbt hierfür von `StandardizerBase`, der Basisklasse für Normierer.

Die Hauptapplikation ruft zu gegebenem Anlass die in der Basisklasse definierte Methode `Initialize` auf und übergibt dabei eine Datenquelle an den Normierer. So kann

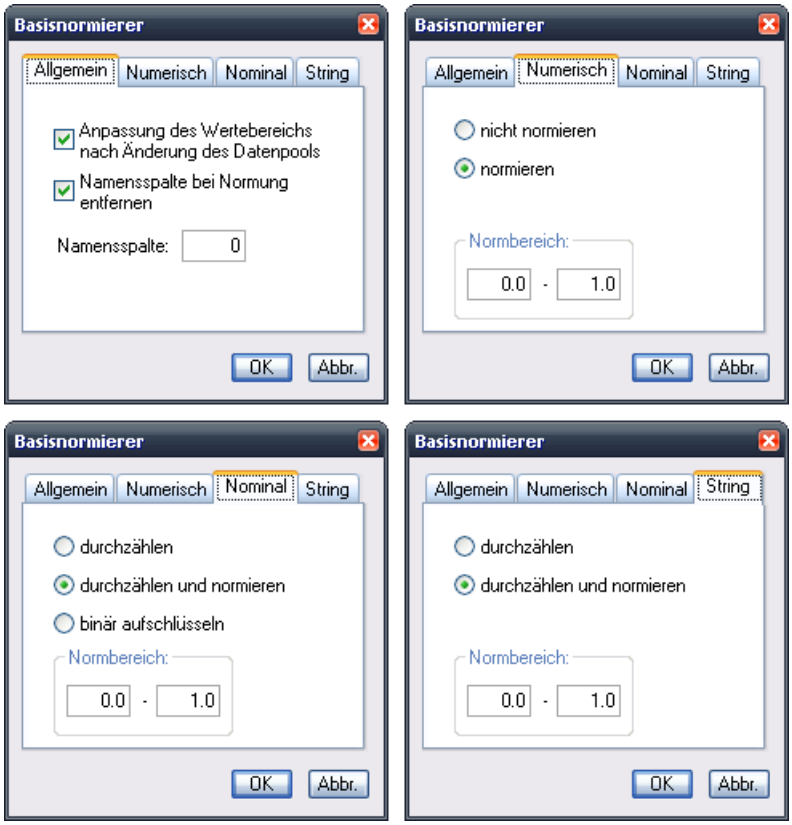

Abbildung 4.14.: Der Konfigurationsdialog des Basisnormierers erlaubt separate Einstellungen für jeden Attributtyp

er sich auf die Datenquelle einstellen und festlegen, wie die Dateninstanzen normiert werden. Die Dimension der Eingabevektoren wird dabei errechnet und in der Eigenschaft `OutputDim` abgelegt. Es kommt die Eigenschaft `ColumnTypes` der Datenquelle zum Einsatz, denn mit ihr wird der Typ (numerisch, nominal oder String) von jedem Attribut offenbart. In diesem Array werden Instanzen der Klassen `NumericType`, `NominalType` und `StringType` genutzt, um die Attribute entsprechend zu markieren.

Der Basisnormierer erlaubt dem Nutzer, über seinen Konfigurationsdialog (vgl. Abb. 4.14) einzustellen, wie die Normung durchgeführt werden soll.

Auf der Karteikarte „Allgemein“ wird über ein Eingabefeld die Namensspalte festgelegt. Der dort eingetragene nullbasierte Index bestimmt das Attribut, das den Namen einer Dateninstanz repräsentiert. Mit der Checkbox „Namensspalte bei Normung entfernen“ wird festgelegt, dass das Attribut mit dem angegebenen Index entfernt wird. Es wird der Karte beim Training nicht angeboten, da der Instanzname keine aussagekräftige Information ist. Stellt die Datenquelle kein Namensattribut zur Verfügung, muss diese Checkbox deaktiviert werden, da sonst wichtige Informationen verloren gehen. Die andere Checkbox ist mit „Anpassung des Wertebereichs nach Änderung des Datenpools“ beschriftet. Mit dieser Einstellung kann festgelegt werden, dass die für die Attribute festgelegten Wertebereiche neu berechnet werden, wenn es zu Änderungen im Datenpool kommt. Dieses Ereignis tritt ein, wenn Instanzen aus dem Pool gelöscht oder editiert werden.

Unter den Reitern „Numerisch“, „Nominal“ und „String“ wird bestimmt, wie mit den Attributen dieser Typen bei der Normung umgegangen werden soll. Die hier beschriebenen Verfahren wurden komplett von dem Programm *SOMARFF* übernommen.

Auf der „Numerisch“-Karteikarte wird bestimmt, wie alle in der Relation vohandenen numerischen Attribute (erkennbar an einer `NumericType`-Instanz) normiert werden. Über Radiobuttons stehen zwei Verfahren zur Auswahl:

nicht normieren Die numerischen Attribute werden für das Training genau so übernommen, wie sie von der Datenquelle zur Verfügung gestellt werden. Eine Normung findet nicht statt.

normieren Die Zahlenwerte werden auf einen vorgegebenen Wertebereich skaliert. Dieser wird in der Box „Normbereich“ festgelegt. Für die Berechnung werden für jedes entsprechende Attribut das Minimum und das Maximum aus dem Da-

tenpool bestimmt. Gleichung 4.2 beschreibt die durchgeführte Berechnung für einen Attributwert $a_i \in \Re$. Der Normbereich wird durch die Werte $norm_{min} \in \Re$ und $norm_{max} \in \Re$ begrenzt. $\max(a)$ steht für den maximalen Wert, den dieses Attribut annehmen kann, $\min(a)$ ist das entsprechende Minimum.

$$x_i = norm_{min} + \frac{(norm_{max} - norm_{min}) \cdot a_i}{\max(a) - \min(a)} \tag{4.2}$$

Unter dem Reiter „Nominal“ kann das Normungsverfahren für alle nominalen Attribute der Relation festgelegt werden. Diese Attribute werden durch ein `NominalType`-Objekt von der Datenquelle markiert. Alle erlaubten Zustände eines solchen Attributs sind in einem Array über die Eigenschaft `NominalType.Values` verfügbar. Auch hier bestimmen Radiobuttons die Methode der verwendeten Normung:

durchzählen Der Wert des Attributs a_i wird in dem `Values`-Array gesucht. Der entsprechende Index wird für den Eingabevektor genutzt. Formel 4.3 zeigt den Vorgang. Die Funktion $ind : A \mapsto \Re$ bestimmt den beschriebenen Index.

durchzählen und normieren Wie beim Durchzählen wird erst der Index des Attributwerts bestimmt. Anschließend wird, wie in Gleichung 4.4 dargestellt, auf den festgelegten Normbereich skaliert. Der Wert $ind_{max} \in \Re$ repräsentiert dabei den maximal möglichen Index für dieses Attribut.

binär aufschlüsseln Dieser Vorgang empfiehlt sich für ein nominales Attribut, dessen Werte nicht ordinal sind. Für jeden erlaubten Wert dieses Attributs wird eine neue Zeile im erzeugten Eingabevektor angelegt. Die Zeile des ausgewählten Werts wird mit 1, alle Anderen mit 0 belegt. Formel 4.5 stellt diese Normungsmethode dar. b wird hier als Laufparameter eingesetzt, der die Werte von 0 bis ind_{max} abarbeitet.

$$x_i = ind\,(a_i) \tag{4.3}$$

$$x_i = norm_{min} + \frac{(norm_{max} - norm_{min}) \cdot ind\,(a_i)}{ind_{max}} \tag{4.4}$$

$$x_{i+b} = \begin{cases} 1 & (ind\,(a_i) = b) \\ 0 & (ind\,(a_i) \neq b) \end{cases} \tag{4.5}$$

Auf der Karteikarte „String“ wird die Normung von `StringType`-Attributen festgelegt. Die Verarbeitung von nominalen und String-Attributen erfolgt fast identisch. Auch hier wird mit Radiobuttons das Verfahren ausgewählt:

durchzählen Die in dieser Spalte vorhandenen Strings aller Instanzen werden durchlaufen, ihnen wird ein Index zugewiesen. Der restliche Ablauf entspricht der Formel 4.3.

durchzählen und normieren Zuerst werden Indizes wie beim Durchzählen erzeugt. Im Anschluß wird auf den festgelegten Normbereich skaliert, wie in Gleichung 4.4 dargestellt.

4.6.3. Euklidische und Manhattan-Distanz

Die Distanzmetriken sind sehr einfache Plugins, deshalb werden die Implementierungen der Euklidischen und der Manhattan-Distanz in einem Abschnitt abgehandelt. Die Klassen `EuclideanDistance` und `ManhattanDistance` stellen die jeweilige Funktionalität bereit. Beide erben von `DistanceBase`, der Basisklasse für Distanzmetriken.

$$d_{Euklid}\,(a,b) = \sqrt{\sum_{i=1}^{n} (b_i - a_i)^2} \qquad a,b \in \Re^n \tag{4.6}$$

$$d_{Manhattan}\,(a,b) = \sum_{i=1}^{n} |b_i - a_i| \qquad a,b \in \Re^n \tag{4.7}$$

Die Methoden `GetDistance` dieser Plugins implementieren jeweils eine der Formeln 4.6 und 4.7.

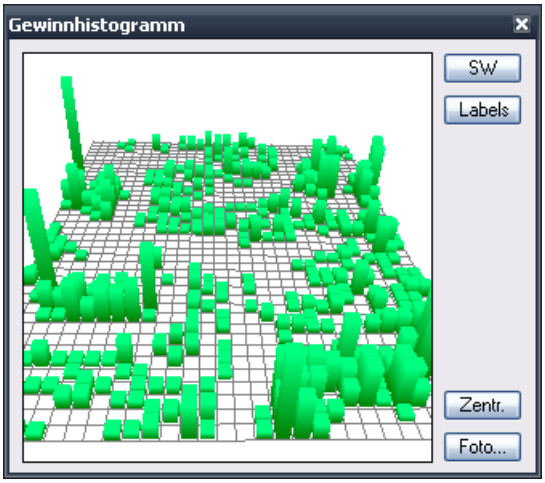

Abbildung 4.15.: Die Darstellung des Gewinnhistogramms erfolgt immer dreidimensional

4.6.4. Gewinnhistogramm

Abbildung 4.15 zeigt das Fenster des Gewinnhistogramms, das implementiert ist durch die Klasse `HitHistogram`. Sie leitet `VisualizationBase` ab, da es sich hierbei um eine Visualisierung handelt. Die Abbildung zeigt den Darstellungsbereich auf der linken und eine Schalterleiste auf der rechten Seite des Dialogs.

Im Darstellungsbereich wird ein dreidimensionales Balkendiagramm über einem quadratischen Gitter abgebildet. Die Ansicht kann vom Nutzer manipuliert werden. Das Diagramm lässt sich horizontal und vertikal verschieben, indem die linke Maustaste auf dem Darstellungsbereich festgehalten und der Zeiger anschließend bewegt wird. Mit der rechten Taste wird die Ansicht um das Diagramm rotiert. Gezoomt wird mit Hilfe des Mausrads, dazu muss aber der Darstellungsbereich vorher fokussiert worden sein.

Die Schalterleiste enthält vier Buttons, zwei am unteren Rand, die die Bezeichnungen „Zentr.“ und „Foto...“ tragen, und zwei am oberen Rand, die mit „SW“ und „Labels“ beschriftet sind. Bei den oberen Schaltflächen handelt es sich allerdings um *Umschalter* (auch *Push-Buttons* genannt), einer besonderen Darstellungsform der Checkbox. Sie können zwei verschiedene Zustände annehmen, aktiv und inaktiv, die mit jedem Klick alternieren. Die Darstellung des Umschalters lässt auf seinen Zustand schließen: Wenn er aktiv ist, sieht er herunter gedrückt aus, bei Inaktivität nicht. Die Schaltflächen des Dialogs erfüllen folgende Aufgaben:

SW schaltet zwischen Schwarzweiß- und Farbdarstellung um.

Labels aktiviert oder deaktiviert Beschriftungen am Diagramm. Es wird an der Oberseite jeden Balkens dessen Höhe numerisch dargestellt.

Zentr. zentriert die Kamera über dem Diagramm. Der Startzustand der Ansicht wird wieder hergestellt.

Foto... speichert eine Kopie des Darstellungsbereiches als Bilddatei ab. Es öffnet sich dafür ein Dialog, mit dem die Zieldatei festgelegt werden kann. Als unterstützte Formate sind *JPEG*, *PNG* und *BMP* vorgegeben.

Das hier dargestellte Diagramm berechnet sich wie folgt: Für jeden Eingabevektor $\vec{x}_i$ wird das Gewinnerneuron c_i bestimmt. Das Histogramm wird an der durch c_i bestimmten Stelle um eins erhöht. Wenn mehrere Eingabemuster das gleiche Gewinnerneuron haben, ist im Gewinnhistogramm ihre entsprechende Anzahl gespeichert. Hier zeigt sich, dass diese Visualisierung abhängig von allen Mustern ist, wie bereits in Tabelle 2.1 angedeutet wurde.

Das Diagramm wird mit `Direct3D` gerendert. Zur Erzeugung von 3D-Objekten (auch *Meshes* genannt) wird die für diesen Zweck entworfene Klasse `MeshCreator` eingesetzt. In diesem Fall wird über die `CreateBox`-Methode nur ein einziger Balken erzeugt, der mit einem grünen Farbverlauf versehen ist. Dieser wird im Renderingprozess für jedes Neuron (dem Histogrammwert entsprechend) skaliert und (der Neuronenposition entsprechend) verschoben.

4.6.5. Gewinnermatrix

Die Gewinnermatrix stellt das im Trainingsprozess ermittelte Gewinnerneuron und dessen beeinflusste Nachbarschaft dar. Sie wird durch die Klasse `WinnerMatrix` implementiert und erbt als Visualisierung von der Basisklasse `VisualizationBase`. Abbildung 4.16 zeigt das entsprechende Fenster.

Die Navigation auf der dargestellten Karte funktioniert auf genau die selbe Weise, wie beim Gewinnhistogramm (Kap. 4.6.4). Gleiches gilt für die Schaltflächen „SW“, „Zentr.“ und „Foto...“. Die Aufgabe der Umschalter „3D“ und „2×2“ wird folgend erläutert.

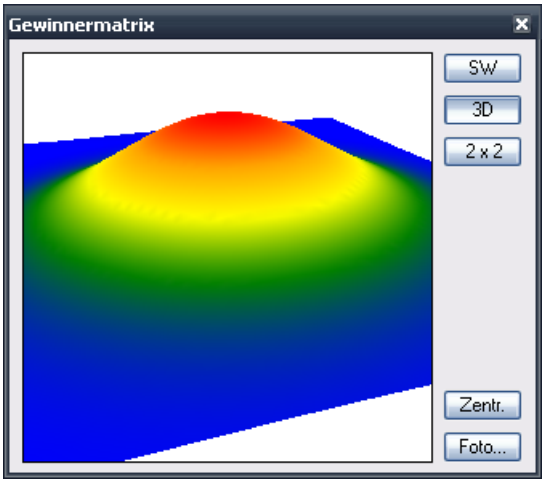

Abbildung 4.16.: Die Gewinnermatrix zeigt den Einflussbereich der Nachbarschaftsfunktion

3D aktiviert oder deaktiviert das Höhenprofil der Karte.

2×2 schaltet die Darstellung in den *Kachelmodus* um. Die Karte wird vierfach abgebildet, damit erregte Bereiche, die über den Rand der SOM hinausgehen, zusammenhängend betrachtet werden können. Diese Funktion ist nur auf zyklische Topologien anwendbar, daher ist der Umschalter auch nur bei Verwendung einer solchen verfügbar.

Die Visualisierung hebt das Gewinnerneuron c zum aktuellen Trainingsvektor $\vec{x}$ hervor, sowie die beeinflussten Nachbarneuronen in dem Bereich, der durch den Lernradius r gegeben ist. Der Grad der Beeinflussung ist durch die Nachbarschaftsfunktion f_{nb} definiert, er wird auf der Karte mit einem Farbverlauf gekennzeichnet. Bei aktiviertem „3D“-Umschalter wird der Farbverlauf zusätzlich mit Höhe versehen, wie auf der Abbildung dargestellt.

Auch diese Visualisierung verwendet `Direct3D` zur Darstellung der Karte. Über die Klasse `MeshCreator` wird mit der Methode `CreateMatrixMesh` ein planares 3D-Objekt erzeugt. Für jedes Neuron der SOM wird dazu ein *Vertex*, ein Punkt im dreidimensionalen Raum, angelegt. Der Netztopologie entsprechend werden Dreiecke zwischen den *Vertices* aufgespannt, die das flächige Objekt ausmachen. Die Farbdarstellung auf der *Mesh* wird dabei nicht durch eine Textur erzeugt. Stattdessen wird den *Vertices* jeweils eine Farbe aus einem vorher definierten Verlauf zugewiesen. Die zwischen

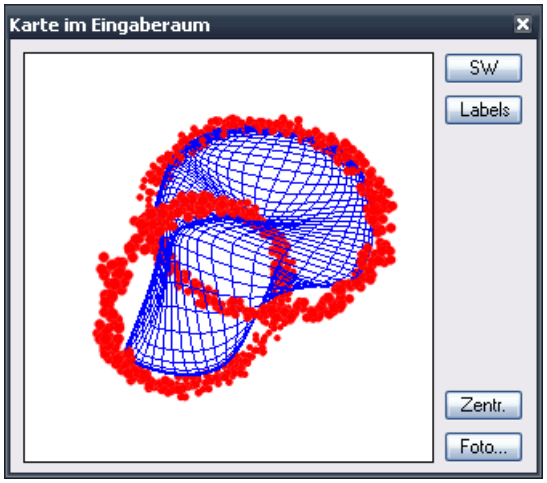

Abbildung 4.17.: Diese Visualisierung stellt die Nachbarschaftsverhältnisse der Neuronen (blau) und die Eingabevektoren (rot) im Eingaberaum dar

den Neuronen liegenden Farbwerte - die die Füllung der Dreiecke ausmachen - werden durch `Direct3D` daraufhin selbst berechnet. Wenn das Höhenprofil aktiviert ist, werden die entsprechenden *Vertices*, die sich innerhalb von r befinden, der Neuronenanpassung entsprechend nach oben verschoben.

4.6.6. Karte im Eingaberaum

Die „Karte im Eingaberaum“-Visualisierung stellt die SOM zusammen mit den Eingabevektoren im Eingaberaum dar, wie in Kapitel 2.3.1 beschrieben. Den entsprechenden Dialog zeigt Abbildung 4.17, der durch die Klasse `MapInInputSpace` implementiert ist.

Die Navigation erfolgt wie gewohnt. Der „Labels“-Umschalter verfügt in diesem Fall aber über eine etwas andere Funktion, als beispielsweise beim Gewinnhistogramm:

Labels schaltet die Kartenbeschriftungen ein oder aus. Es werden die Gewinnerneuronen für jeden Eingabevektor bestimmt. An der entsprechenden Position der SOM wird ein Schriftzug eingeblendet, der den Namen der Dateninstanz wiederspiegelt. Man spricht von der *Kalibrierung* der Karte.

Zur Erzeugung dieser Ansicht werden die Gewichtsvektoren eines jeden Neurons als Position im Eingaberaum interpretiert. Daher ist diese Visualisierung nur auf zwei- oder

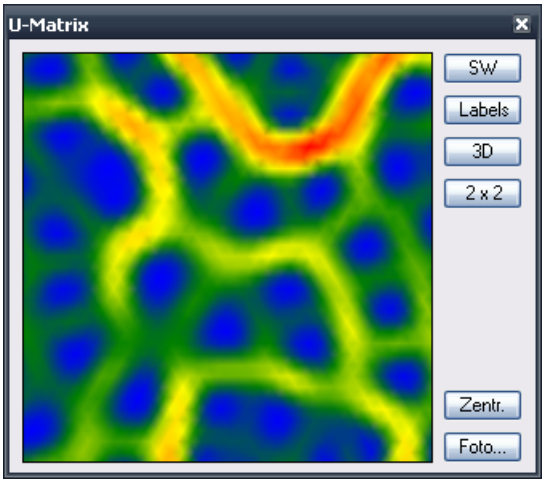

Abbildung 4.18.: Die U-Matrix zeigt die Distanzen der Neuronen zu ihren Nachbarn in einem Farbverlauf

dreidimensionale Eingabedaten anwendbar. Für jedes Neuron wird wieder ein *Vertex* erzeugt, dessen Position durch den Inhalt des entsprechenden Gewichtsvektors bestimmt ist. Durch die Netztopologie sind allen Neuronen direkte Nachbarn zugeordnet. Diese Information wird genutzt, indem die *Vertices* der Topologie entsprechend untereinander mit blauen Linien verbunden werden. So entsteht ein Gitter, das die SOM mit allen Nachbarschaftsverbindungen repräsentiert. Diese Aufgabe wird durch den Aufruf der Methode `MeshCreator.CreateWireMesh` durchgeführt. Die Eingabevektoren werden durch rote, kreisförmige *Sprites* (zweidimensionale Grafiken) im Eingaberaum dargestellt.

Im Verlauf des Trainings ist mit dieser Visualisierung zu beobachten, wie sich die Karte an die Eingabevektoren anschmiegt.

4.6.7. U-Matrix

Die U-Matrix implementiert die in Kapitel 2.3.2 beschriebene Visualisierung mit der Klasse `UMatrix`, die wiederum `VisualizationBase` ableitet. Das Visualisierungsfenster ist in Abbildung 4.18 dargestellt.

Die Navigation auf der Karte funktioniert, wie bei den anderen vorgestellten Visualisierungen. Zusammenfassend erfüllt die Schalterleiste auf der rechten Seite des Dialogs folgend beschriebene Funktionen.

SW schaltet zwischen Schwarzweiß und Farbe um.

Labels aktiviert oder deaktiviert die Beschriftung der Karte mit den Namen der Dateninstanzen.

3D verleiht der Karte ein Höhenprofil oder schaltet es ab.

2×2 aktiviert die gekachelte Darstellung für zyklische SOM.

Zentr. zentriert die Kamera über dem Mittelpunkt der Karte.

Foto... erstellt ein Bildschirmfoto des Darstellungsbereichs und erlaubt die Speicherung als Bilddatei.

Die Darstellung erfolgt wieder über `Direct3D`, es wird ein planares 3D-Objekt über `MeshCreator.CreateMatrixMesh` erzeugt. Der zur Einfärbung der Karte genutzte Algorithmus wurde in Kapitel 2.3.2 bereits ausführlich vorgestellt. Die Erstellung des Höhenprofils erfolgt, indem der für die U-Matrix berechnete Wert u_i für jedes Neuron nicht nur als Farbindikator, sondern auch als Höhe für den entsprechenden *Vertex* eingesetzt wird.

5. Experimente

In diesem Kapitel werden einige Experimente mit der Applikation *Sombrero* durchgeführt. Diese können durch den Leser nachvollzogen werden. Sie dienen als Einführung in die Funktionsweise und den Arbeitsablauf des Programms und können den allgemeinen Umgang mit ihm darstellen.

5.1. Training mit hochdimensionalen Daten

Hier wird ein sehr typischer Fall erläutert, für den Kohonennetze eingesetzt werden. Es existiert ein Datensatz mit vielen Attributen, der auf eine zweidimensionale Karte abgebildet werden soll. Diese kann im Anschluss betrachtet werden. Es ist möglich, eventuell vorhandene Cluster in den Eingabedaten zu entdecken.

Für dieses Experiment wird die Datei `zoo.arff` als Datenquelle eingesetzt. Sie enthält eine Reihe von Tieren, denen verschiedenste Eigenschaften zugeordnet werden, z.B. Zahl der Beine, Art (Säugetier, Fisch, ...), Federn, Zähne u.s.w. Es handelt sich hierbei um hochdimensionale Daten, jede Eigenschaft entspricht einer Dimension.

Nach dem Start des Programms *Sombrero* wird zuerst eine neue selbstorganisierende Karte angelegt. Hierzu wird unter dem Menüpunkt „SOM“ der Eintrag „Neu...“ gewählt. Im erscheinenden Dialog wird auf die Karteikarte „Datenquelle“ gewechselt. Standardmäßig ist „ARFF Datei“ als Datenquelle voreingestellt. Mit einem Klick auf die nebenliegende Schaltfläche „...“ wird sie konfiguriert. Es öffnet sich ein Dialog, der einen Dateinamen anfordert. Über den entsprechenden „...“-Button wird ein Dateidialog geöffnet, mit dem die Datei `zoo.arff` lokalisiert und ausgewählt wird. Mit „OK“ wird die Eingabe bestätigt und der Konfigurationsdialog der Datenquelle geschlossen. Anschließend werden die vorgenommenen Einstellungen im „Neue SOM“-Dialog mit „OK“ bestätigt, das Fenster schließt sich.

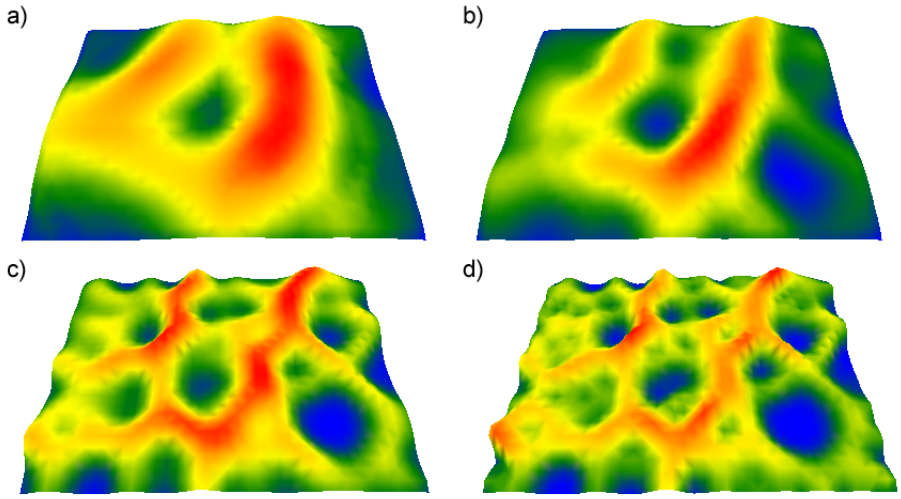

Abbildung 5.1.: Die U-Matrix im Verlauf des Trainings mit hochdimensionalen Daten a) nach 200 Lernschritten b) nach 500 Lernschritten c) nach 1000 Lernschritten d) nach 2000 Lernschritten

Vor dem Training wird die „U-Matrix“-Visualisierung geöffnet, indem im Menü „Ansicht“ der gleichnamige Eintrag aktiviert wird. Diese Darstellung wird zur Clusteranalyse bei hochdimensionalen Daten gern eingesetzt. Ähnliche Bereiche werden durch ein Tal, stark unterschiedliche Bereiche durch einen Berg repräsentiert.

Im Moment sind allerdings noch keine Cluster zu erkennen, stattdessen ist eine zufällig initialisierte Karte mit enormem „Rauschen“ sichtbar. Das Visualisierungsfenster lässt sich vergrößern, um eine detailliertere Ansicht zu bekommen. Die Karte kann mit gedrückter linker Maustaste im Darstellungsbereich verschoben werden, mit dem Mausrad wird auf Details gezoomt. Mit der rechten Maustaste kann die Ansicht auch rotiert werden. Über die Aktivierung des Umschalters „3D“ wird der Karte ein Höhenprofil verliehen.

Nun wird der Lernprozess gestartet, indem im „Trainer“-Dialog der Startknopf angeklickt wird. Man kann auf der U-Matrix beobachten, wie sich erst grobe Strukturen auf der Karte abbilden, die dann immer detaillierter ausgeprägt werden (s. Abb. 5.1). Dies ist auf die Lernparameter zurückzuführen. Die Standardeinstellungen beinhalten einen großen Lernradius und eine große Lernrate zu Beginn des Trainings, die dann beide im weiteren Verlauf immer mehr reduziert werden. Der aktuelle Zustand dieser Werte ist im Trainingsdialog abzulesen.

Nach 2000 Lernschritten ist der Trainingsprozess beendet. Auf der U-Matrix sind nun Cluster in Form von Tälern gut zu erkennen. Mit einem Klick auf den Umschalter „Labels“ werden die Beschriftungen aktiviert, die Karte wird *kalibriert*. Nun sind alle Tiere, mit denen die SOM trainiert wurde, in der Darstellung ihren Positionen entsprechend angeordnet.

Es gibt einige Fälle, wo sich mehrere Tiere eine Position teilen, sie haben das gleiche Gewinnerneuron. Aus diesem Grund wird unter „Ansicht“ die „Gewinnhistogramm“-Visualisierung geöffnet. Sie lässt sich fast genauso bedienen, wie die U-Matrix. Es ist ein dreidimensionales Balkendiagramm abgebildet, das die Häufung von Gewinnern unter allen Neuronen darstellt.

Ein weiterer interessanter Aspekt ist die Frage, wie sich die Tierarten auf der Karte verteilt haben. Es wird die „Komponentenmatrix“ (wieder im „Ansicht“-Menü) geöffnet. Anfangs ist bei dieser Darstellung noch kein Muster zu erkennen. Es muss zuerst ein Attribut der Trainingsdaten gewählt werden, damit seine Verteilung auf der Karte sichtbar wird. Also wird in der Klappbox auf dem Visualisierungsfenster das Attribut „type“ gewählt. Auch hier können die Tiernamen über „Labels“ eingeblendet werden. Es zeigt sich, dass z.B. alle Säugetiere in einem blauen Cluster angeordnet sind, die Fische teilen sich den gelben Bereich.

Auf allen Visualisierungen ist der sogenannte *Randeffekt* zu erkennen: Viele Tiere sammeln sich an den Rändern der Karte, das Zentrum ist im Verhältnis recht dünn besiedelt. Dieser Effekt tritt auf, weil die gewählte Standardtopologie (quadratisches Gitter) nicht kontinuierlich ist und deshalb die Neuronen am Rand der SOM weniger direkte Nachbarn haben, als zentral gelegene Neuronen. Das kann starke Auswirkungen auf das Training haben und zu dieser Randanordnung führen.

Um diesen Effekt zu meiden gibt es zyklische Netztopologien, deren Randneuronen mit den gegenüberliegenden verbunden sind. Das Experiment wird mit solch einer zyklischen Topologie wiederholt.

Über „SOM / Neu...“ wird eine neue selbstorganisierende Karte angelegt. Auf der „SOM“-Karteikarte im erscheinenden Dialog wird als Topologie „Torus“ mit der entsprechenden Klappbox gewählt. Die restlichen Einstellungen bleiben bestehen. Mit einem Klick auf „OK“ wird der Dialog wieder geschlossen, eine neue Karte mit Torus-Topologie wird erzeugt.

Die im Training eingesetzten Lernparameter müssen wieder auf ihren Startzustand

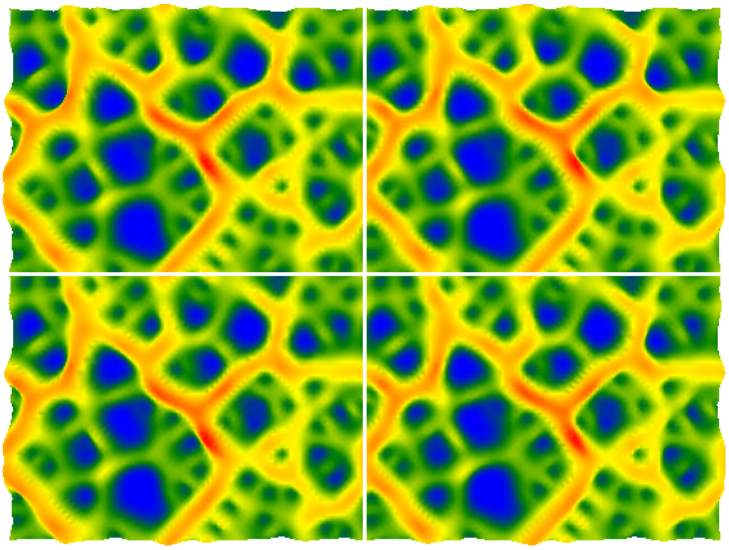

Abbildung 5.2.: Die U-Matrix einer zyklischen Karte im Kachelmodus

gebracht werden. Das geschieht mit einem Klick auf „Anwenden“ im Lernparameterdialog. Im Anschluss wird der Trainingsprozess erneut mit dem Startknopf angestoßen.

Nach 2000 Lernschritten lässt sich die Karte auf gleiche Weise untersuchen, wie schon beim ersten Durchlauf geschehen. Es ist festzustellen, dass die Tiere sich viel gleichmäßiger auf der SOM verteilt haben. Es kann aber vorkommen, das Cluster sich über die Grenzen der Karte hinaus erstrecken. Zu diesem Zweck existiert bei vielen Visualisierungen der „2×2“-Umschalter. Damit wird erreicht, dass die SOM gekachelt dargestellt wird. So wird es ermöglicht, alle Cluster auch über die Begrenzungen hinaus zu betrachten. Abbildung 5.2 zeigt die U-Matrix einer so trainierten Karte im Kachelmodus.

5.2. Das Travelling Salesman Problem

Das Travelling Salesman Problem (*TSP*) ist eine sehr beliebte Aufgabe für einige Methoden der künstlichen Intelligenz. Ein Kaufmann geht auf Reise und will eine Menge von Städten besuchen, um dort Handel zu treiben. Dabei will er jede Stadt nur einmal besuchen und trotzdem eine möglichst kurze Gesamtstrecke zurücklegen.

Bekannt sind die Städte und deren zweidimensionale Koordinaten auf der Landkarte. Bei einigen Varianten werden die erste und letzte zu besuchende Stadt vorgegeben,

oder mit einer Kostenfunktion wird die Beschwerlichkeit der Wege beschrieben. Diese Beispiele können auf herkömmliche Art durch SOM nicht gelöst werden.

Für die Durchführung dieses Experiments wurde vom Autor eine ARFF-Datei erzeugt, die eine Relation mit einigen zufällig gewählten deutschen Städten und deren Koordinaten beschreibt. Diese Datei wird hier als Datenquelle fungieren.

Nach dem Start von *Somrero* wird über „Neu...“ unter dem „SOM“-Menüpunkt eine neue Karte angelegt, indem sich der entsprechende Dialog öffnet. Die Handelsroute soll nachgebildet werden, daher kommen nur eindimensionale Topologien in Frage. Die Drop-Down-Liste für Topologien auf der „SOM“-Karteikarte bietet aber leider nur zweidimensionale Exemplare, daher wird ein wenig improvisiert. Die „Quadratisches Gitter“-Topologie wird gewählt, ihr Konfigurationsdialog wird über die „...“-Schaltfläche geöffnet. Eine eindimensionale Neuronenkette wird erzeugt, indem für „Breite“ der Wert 120, für „Höhe“ nur eine 1 gewählt wird. Mit „OK“ wird die Eingabe bestätigt, es kommt also eine eindimensionale Topologie mit einer Länge von 120 Neuronen zum Vorschein.

Unter dem Reiter „Datenquelle“ wird nun „ARFF Datei“ als eine solche gewählt. Die Datei wird im Konfigurationsdialog (wieder über „...“) festgelegt. Dort befindet sich ein Eingabefeld, das mit „Dateiname“ beschriftet ist. Auf der rechten Seite befindet sich eine weitere „...“-Schaltfläche, mit der lässt sich ein Dateidialog öffnen. Die Datei `cities.arff` wird ausgewählt, sie enthält die vorher beschriebene Relation mit deutschen Städten und ihren Koordinaten. Mit „OK“ werden die Einstellungen übernommen und der Konfigurationsdialog der Datenquelle geschlossen. Alle weiteren Optionen auf dem „Neue SOM anlegen“-Dialog behalten ihre Standardwerte, es wird mit „OK“ bestätigt.

Vor dem Training müssen noch die Lernparameter angepasst werden. Im entsprechenden Dialog werden alle Standardeinstellungen beibehalten, nur der Lernradius wird auf einen Startwert von 50 festgelegt und die Anzahl der Lernschritte auf 3000 erhöht. Der Radius wird anfangs so hoch gewählt, da es sich um eine recht lange Neuronenkette handelt und bei den ersten Lernschritten möglichst viele beeinflusst werden sollen. Mit „Anwenden“ werden die geänderten Parameter übernommen.

Die Anzahl der Städte aus der Datenquelle beträgt 41, die Zahl der verwendeten Neuronen sollte mindestens diese Höhe haben. Es wird dem Netz mit 120 Neuronen hier mehr als genug Spielraum gelassen.

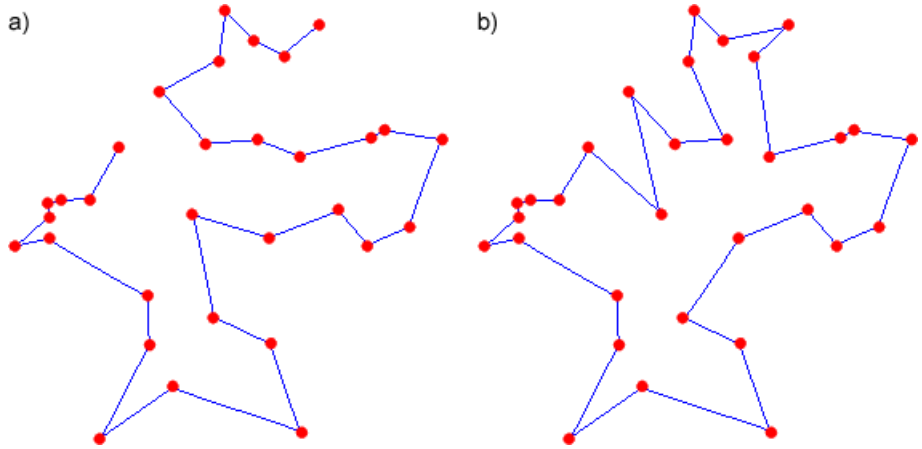

Abbildung 5.3.: Die eindimensionale Karte im Eingaberaum der Koordinaten deutscher Städte a) Neuronenkette als Topologie b) ringförmige Topologie

Unter dem Menüpunkt „Ansicht“ wird der Eintrag „Karte im Eingaberaum“ gewählt, die gleichnamige Visualisierung öffnet sich. Es zeigt sich ein Durcheinander blauer Linien und einige rote Punkte. Die Linien repräsentieren die nachbarschaftlichen Verbindungen der Neuronen, deren Gewichtsvektoren mit Zufallswerten initialisiert wurden. Die roten Punkte entsprechen den Eingabewerten, also den zweidimensionalen Koordinaten der deutschen Städte. Man kann das Fenster nach Belieben vergrößern. Mit gedrückter linker Maustaste im Darstellungsbereich lässt sich die Karte verschieben, mit der rechten Maustaste sogar drehen (was in diesem Fall aber aufgrund der Zweidimensionalität wenig Sinn macht). Mit dem Mausrad lässt sich auf interessante Bereiche zoomen. Die ursprüngliche Ansicht kann über „Zentr.“ wieder hergestellt werden.

Mit einem Klick auf den Startknopf im Trainingsdialog wird der Lernprozess angestoßen. Man erkennt in dem Visualisierungsfenster, wie die Neuronenkette anfängt, sich an die Eingabewerte anzupassen und sich ihnen immer mehr zu nähern. Nach wenigen Sekunden ist der Vorgang beendet und es präsentiert sich ein Resultat, das der Abbildung 5.3a ähnelt. Alle roten Punkte sind durch die Linien miteinander verbunden, eine mögliche Route des Handelsreisenden wird angezeigt. Die erste und letzte Stadt der Reise werden durch den Zufall bei der Initialisierung der Karte bestimmt.

Die Namen der Städte können mit dem Umschalter „Labels“ eingeblendet werden. Dabei werden aber nicht die Eingabedaten (rote Punkte) beschriftet, sondern die entsprechenden Gewinnerneuronen, also die Ecken der blauen Linie, die den Punkten am

nächsten sind. Beim näheren Betrachten der Karte fällt auf, dass sie vertikal ein wenig gestaucht erscheint. Das liegt an der Arbeit des Normierers, der die Eingabewerte anpasst. Die Koordinaten beider Achsen werden auf einen Bereich zwischen 0 und 1 skaliert, deshalb ergibt sich ein quadratischer Eingaberaum.

Das Experiment lässt sich einfach wiederholen, indem der „Neue SOM anlegen“-Dialog erneut geöffnet (Eintrag „Neu...“ unter dem „SOM“-Menüpunkt) und sofort mit „OK“ wieder geschlossen wird. Der Dialog hat die vorher getroffenen Einstellungen beibehalten. Das Netz wurde wieder mit Zufallswerten initialisiert, es ist also möglich, dass bei mehreren Durchläufen unterschiedliche Ergebnisse erzielt werden. Vor dem erneuten Start des Trainings müssen aber die Lernparameter wieder mit „Anwenden“ übernommen werden.

Nun wird das Experiment mit einer kleinen Änderung erneut durchgeführt. Es sollen wieder alle Städte durchlaufen werden, allerdings soll dies mit einer Rundreise geschehen, die erste und letzte besuchte Stadt sollen identisch sein. Das spricht für eine ringförmige Netztopologie.

Der „Neue SOM anlegen“-Dialog wird also geöffnet, auf der „SOM“-Karteikarte wird die „Torus“-Topologie ausgewählt, da diese zyklisch ist. Ihre Einstellungen werden über „...“ genau so angepasst, wie beim ersten Durchlauf: Die „Breite“ wird auf 120, die „Höhe“ auf 1 gesetzt, mit „OK“ wird bestätigt. Die restlichen Einstellungen bleiben bestehen, mit „OK“ wird eine neue Karte mit Ringtopologie erzeugt.

Die Lernparameter werden erneut mit „Anwenden“ übernommen, das Training mit dem Startknopf initiiert. Das Ergebnis ist in Abbildung 5.3b zu sehen. Es ist zu erkennen, dass der Ring alle Städte durchläuft. Die Grafik im Darstellungsbereich der Visualisierung lässt sich mit einem Klick auf die Schaltfläche „Foto...“ in verschiedenen Formaten abspeichern. Die Kartenabbildungen in diesem Kapitel wurden alle auf diese Art erstellt.

5.3. Das Amputationsexperiment

Bei einem Experiment, das unter anderem in [BHS07] erwähnt ist, werden die somatosensorischen Felder des Kortex eines ausgewachsenen Affen überwacht, während die Fingerspitzen einer seiner Hände einem Stimulus ausgesetzt sind. Die nebeneinander liegenden Bereiche werden ausgemessen, die bei der Reizung der einzelnen Finger an-

Abbildung 5.4.: Der Konfigurationsdialog der benutzerdefinierten Datenquelle mit der manuell erzeugten Relation „Hand“

gesprochen werden. Dem Affen wird der Mittelfinger amputiert. Nach zwei Monaten ist zu beobachten, dass sich der stimulierte Bereich auf dem Kortex den angebotenen Reizen der verbliebenen Finger angepasst hat. Der Bereich des Mittelfingers wurde von denen der Anderen vollständig assimiliert und somit steht jedem übrigen Finger eine größere Erregungsfläche zur Verfügung.

Dieses Experiment wird nun mit Hilfe der selbstorganisierenden Karten und *Sombrero* rekonstruiert. Nach dem Start des Programms wird über den Menüeintrag „Neu...“ eine neue Karte angelegt.

Im erscheinenden Dialog wird auf der Karteikarte „SOM“ erst die „Hexagon-Torus“-Topologie gewählt, dabei handelt es sich um ein toroides, hexagonales Neuronennetz. Die Standardeinstellungen der Topologie (40×30 Neuronen und euklidische Nachbarschaft), die Distanzmetrik und Initialisierung werden beibehalten.

Unter dem Reiter „Datenquelle“ wird als solche „Benutzerdefiniert“ ausgewählt. Dies ist die einzige Quelle, die die Veränderung des Datenbestandes während des Trainings gestattet. Das ist für die Durchführung dieses Experiments notwendig, da ein „Finger“ entfernt werden soll. Der entsprechende Konfigurationsdialog wird über „...“ geöffnet. Nun wird der verwendete Datenbestand manuell angelegt. Dazu wird im Menü „Tabelle“ auf „Neu...“ geklickt. Der Name der erzeugten Relation soll „Hand“ sein, über „OK“ schließt sich das Eingabefenster. Die neue Tabelle wird um ein Attribut über „Spalte hinzufügen...“ ergänzt. Es handelt sich um ein nominales Attribut, es soll den Namen eines Fingers repräsentieren. Daher wird die Spalte „Finger“ genannt und bei Typ „nominal“

gewählt, anschließend wird mit „OK“ bestätigt. Nun können der Relation die Werte „kleiner Finger“, „Ringfinger“, „Mittelfinger“, „Zeigefinger“ und „Daumen“ über das Eingabefeld in der dargestellten Tabellenansicht hinzugefügt und jeweils mit der Eingabetaste bestätigt werden. Abbildung 5.4 zeigt den Konfigurationsdialog mit der erzeugten Relation. Ein Klick auf „OK“ übernimmt die Einstellungen und schließt den Dialog.

Die Normung der Relation übernimmt der Basisnormierer. Allerdings ist seine aktuelle Konfiguration ungeeignet, da die Namensspalte, die in diesem Fall die Fingerbezeichnungen enthält, beim Normungsprozess entfernt wird. Deshalb wird mit „...“ der Konfigurationsdialog des Normierers geöffnet und unter dem Reiter „Allgemein“ die Checkbox mit der Bezeichnung „Namensspalte bei Normung entfernen“ deaktiviert. Auf der Karteikarte „Nominal“ wird als Normungsverfahren „binär aufschlüsseln“ gewählt, da für die Finger keine Ordnung existiert. Die restlichen Einstellungen bleiben wie sie sind, der Dialog wird mit „OK“ geschlossen.

Die getroffene Konfiguration der Datenquelle und des Normierers können durch die Schaltfläche „Daten ansehen“ noch einmal überprüft werden. Die Einstellungen für die neue SOM sind vollzogen, der entsprechende Dialog wird ebenfalls durch einen Klick auf die „OK“-Schaltfläche geschlossen.

Nun werden noch die Lernparameter angepasst, bevor endlich mit dem Training begonnen werden kann. Es wird ein natürlicher Prozess nachgebildet: die Anpassung der somatosensorischen Felder eines Kortex. Dieser Vorgang ist kontinuierlich und dynamisch, deshalb wird von einer konstanten Lernrate und einem ebenfalls konstanten Lernradius ausgegangen. Die Lernrate wird relativ klein gewählt, damit in diesem Ablauf die gelernten Daten nicht sofort wieder „vergessen“ werden. Dafür wird der Lernradius relativ groß gewählt, so wird gewährleistet, dass die trainierte Karte nicht stagniert. Im Lernparameterdialog werden also die Lernrate auf 0.1, der Radius auf 15 gesetzt. Um eine Konstanz bei beiden Werten zu erreichen, wird jeweils die exponentielle Reduktion und ein Faktor von 1 gewählt. Die Gaußglocke soll wieder als Nachbarschaftsfunktion fungieren. Der Lernvorgang soll nach 2000 Lernschritten vorerst beendet sein. Mit „Anwenden“ werden die Einstellungen übernommen. In der Abbildung 5.5 ist die Konfiguration der Lernparameter noch einmal dargestellt.

Unter dem Menüpunkt „Ansicht“ wird die „L-Matrix“-Visualisierung aktiviert. Diese kann nun nach Belieben ausgerichtet werden: Die Karte wird mit gedrückter linker

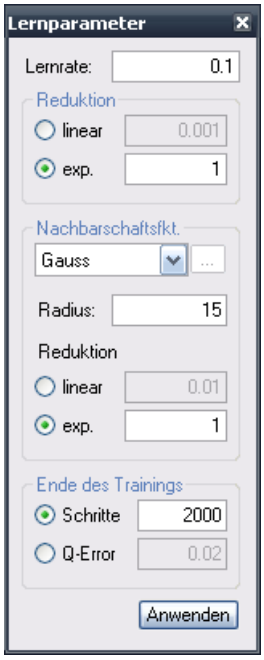

Abbildung 5.5.: Die Lernparameter-Einstellungen für das Amputationsexperiment

Maustaste horizontal und vertikal verschoben, mit rechts wird sie um ihre Achsen rotiert. Das Mausrad kann zum Zoomen eingesetzt werden, wenn die Karte vorher mit einem Klick fokussiert wurde. Mit dem Umschalter „3D“ wird der Karte ein Höhenprofil der Einfärbung entsprechend angelegt. Die Kartenbeschriftungen sollten mit dem Umschalter „Labels“ aktiviert werden, damit der Lernfortschritt und die Verschiebung der Kortexbereiche den einzelnen Fingern zugeordnet werden kann.

Nun wird der Trainingsprozess mit dem Startknopf in Gang gesetzt. Auf der L-Matrix ist der Vorgang gut zu beobachten. Es ist zu erkennen, wie die einzelnen Finger auf der Karte um die Aktivierungsbereiche konkurrieren. Nach einer Weile sollten die Beschriftungen aufhören, unkontrolliert umher zu springen, sondern sich nur noch sehr langsam bewegen. In den Grenzbereichen der sich anbahnenden Cluster kommt es noch zu leichten Verschiebungen.

Nach den 2000 Lernschritten müsste dann ein recht stabiler Zustand erreicht sein, der mit der Abbildung 5.6a vergleichbar ist. Hier sind eindeutig fünf Cluster zu erkennen, die den Einzugsbereichen der Finger entsprechen. Zur Beachtung: Da eine toroide Netztopologie vorliegt, gehen die Ränder einiger Cluster über die Kartengrenzen hinaus, um auf der entgegengesetzten Seite wieder einzutreten. Aus diesem Grund existiert

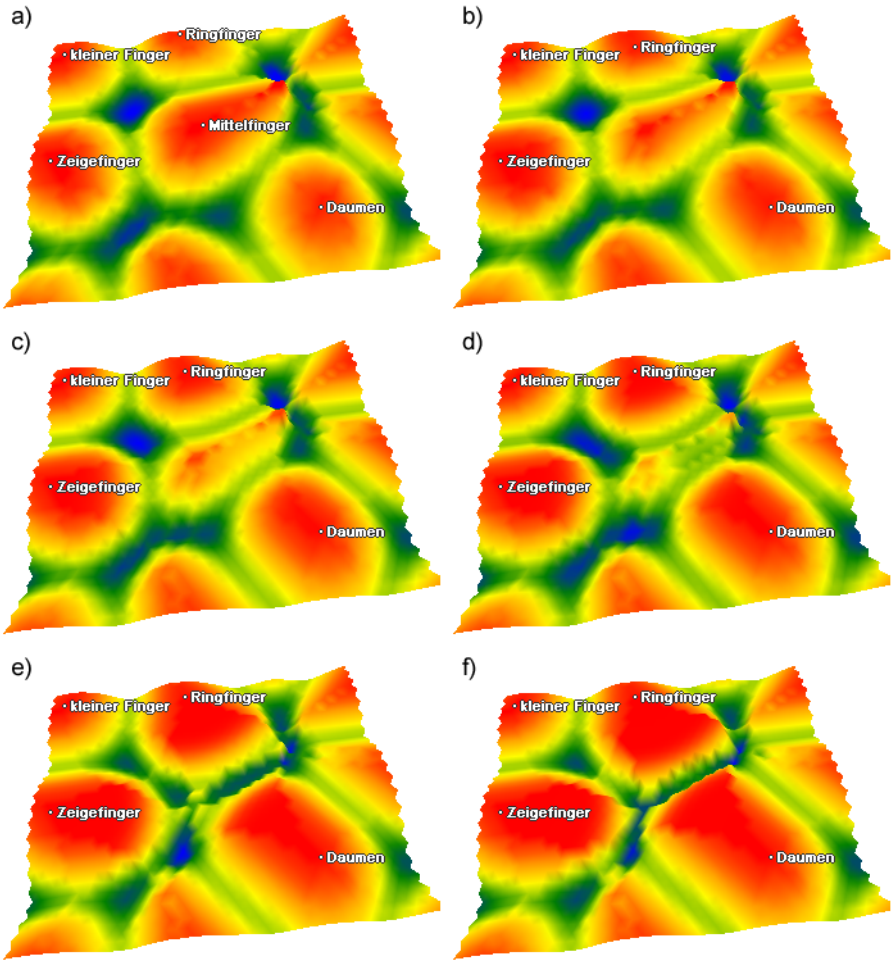

Abbildung 5.6.: L-Matrix im Verlauf des Amputationsexperiments a) 2000 Lernschritte mit allen Fingern b) 2050 Lernschritte, davon 50 ohne Mittelfinger c) 2100 Lernschritte d) 2200 Lernschritte e) 2500 Lernschritte f) 3000 Lernschritte

der Schalter „2×2“, mit dem die Karte vierfach gezeichnet wird. Damit können auch zerrissene Bereiche zusammenhängend betrachtet werden.

Jetzt wird die Amputation des Fingers durchgeführt. Hierzu wird unter „Ansicht“ der *Data Inspector* geöffnet. Auf der Karteikarte „Datenpool“ ist die Tabelle mit den Fingernamen zu erkennen. Es wird die Spalte mit dem Mittelfinger gewählt, indem auf das graue Feld links daneben geklickt wird. Mit einem Druck auf die „Entf“-Taste wird die Instanz gelöscht, der Finger wird amputiert. „OK“ übernimmt die Einstellungen und schließt den *Data Inspector*.

Anschließend wird der Karte erlaubt, sich auf die Änderungen einzustellen. Deshalb werden bei den Lernparametern 1000 Schritte eingegeben und mit „Anwenden“ erneut übernommen. Ein erneuter Klick auf den Startknopf setzt das Training fort. Es ist zu beobachten, wie mit der Zeit der Bereich der Karte, der durch den Mittelfinger belegt war, mehr und mehr schrumpft und durch die anderen stetig wachsenden Cluster vereinnahmt wird. Die Abbildungen 5.6b bis f zeigen einen solchen Verlauf nach weiteren 50, 100, 250, 500 und 1000 Lernschritten.

6. Zusammenfassung

Diese Arbeit gewährt einen Einblick in die Funktionsweise selbstorganisierender Karten. Dazu wird der allgemeine Aufbau von neuronalen Netzen beleuchtet, deren biologisches Vorbild und das Lernen. Die Kohonennetze werden mit Aufbau, Trainingsverfahren und Visualisierungsformen erläutert.

Eine komplexe Applikation wird entwickelt, die den Trainingsablauf von SOM und deren Visualisierung zur Aufgabe hat. Diese Arbeit beschreibt die dafür benötigten Vorgänge in den Bereichen Design und Implementierung. Hierbei wird auf spezielle Verfahren eingegangen, die z.B. Parallelität von Arbeitsprozessen und die Erweiterung des Programms in vielen Bereichen ermöglichen. Und schließlich führen einige beschriebene Experimente in den praktischen Umgang mit der Applikation ein.

6.1. Wertung der Ergebnisse

Die im Rahmen dieser Arbeit entstandene Applikation *Sombrero* macht einen ausgereiften Eindruck, sie ist zu einem nützlichen Werkzeug avanciert. Trotz seiner hohen Komplexität ist das Programm benutzerfreundlich umgesetzt. Der Trainingsablauf bei selbstorganisierenden Karten wurde mit hohem Parametrisierungsgrad verwirklicht, es stehen vier SOM-Topologien und fünf Nachbarschaftsfunktionen zur Wahl. Es kommen moderne, interaktive Darstellungsformen für die Visualisierungen zum Einsatz, von denen mindestens sieben (u.a. die U-Matrix und die Karte im Eingaberaum) implementiert sind. Das Programm ist durch sein Plugin-Management hochgradig erweiterbar.

Leider war es dem Autor aus Zeitmangel nicht mehr möglich, einige der algorithmisch komplexeren Visualisierungsmethoden, wie die von Ultsch ([Ult03a, Ult03c]) vorgeschlagene P-Matrix und U*-Matrix, zu implementieren. Da das Programm aber über entsprechende Plugin-Schnittstellen verfügt, besteht die Möglichkeit der Weiterentwicklung in diesem und anderen Bereichen.

Des Weiteren fehlt die Persistierbarkeit der trainierten Karte und der genutzten Plugins. Das Konzept für diese Anforderung besteht und die Strukturen für diesen Zweck sind vorhanden, die Hauptapplikation macht aber keinen Gebrauch von ihnen. Denkbar wäre eine Umsetzung mittels XML.

6.2. Ausblick

Die Idee der Selbstorganisation ist ein faszinierender Aspekt der neuronalen Modellierung. Die vielseitige Einsatzfähigkeit im Bereich des Data Mining und der Analyse hochdimensionaler Daten verspricht Zukunftssicherheit für diese Verfahren. Beispielsweise kommt der Einsatz in der Medizin bei der Untersuchung von DNS und zur Unterstützung von Diagnosen in Betracht. In der Wirtschaft können SOM z.B. zur Vorhersage von Aktienkursen und zur Analyse von Kundendaten eingesetzt werden.

Dennoch ist zu bedenken, dass diese Technologie auch missbraucht werden kann. Die Sammlung von Daten spielt schon heutzutage eine bedeutende Rolle. Ob beim Online-Einkauf oder im Supermarkt mit sogenannten „Payback“-Karten, es wird genau erfasst, welche Person wann welche Produkte gekauft hat (Stichwort: Gläserner Kunde). Diese Daten werden mit unterschiedlichsten Verfahren analysiert, selbstorganisierende Karten eignen sich sehr für diese Aufgabe. Das Resultat solcher Betrachtungen sind z.B. gezielte Werbemaßnahmen. Krankenkassen, Versicherungen und Kreditinstitute können sich ebenfalls dieser Mittel bedienen, um die Aufnahme neuer Kunden und die Bewilligung von Krediten in ihrem Sinne zu manipulieren. Sollte ein Austausch von Daten unter diesen Konzernen stattfinden, könnte sich ein branchenübergreifendes Informationskartell entwickeln. Der Gläserne Kunde würde ein Gläserner Mensch. Daher ist es dringend erforderlich, Richtlinien im Umgang mit persönlichen Daten gesetzlich zu festigen und so den Ruf dieser Technologie zu wahren.

A. Anhang

A.1. Beispielcode zur Verwendung des Konfigurationsdialogs

In diesem Abschnitt wird die Verwendung des automatisch erzeugten Konfigurationsdialogs mit einem Codebeispiel erläutert. Die unten erzeugte Klasse erbt von einer imaginären Basisklasse für Plugins. Diese dient nur als Beispiel, in Wirklichkeit werden die Klassen `VisualizationBase`, `DataSourceBase` u.s.w. abgeleitet. Es wird keine vollständige Klasse implementiert, sondern nur zwei Methoden, die die Verwendung des Dialogs veranschaulichen.

```
using System.Windows.Forms;

// benötigte Namensräume aus der Plugin-Layer
using Sombrero.PlugIn.BusinessLogic;
using Sombrero.PlugIn.BusinessLogic.Configuration;
using Sombrero.PlugIn.BusinessLogic.Configuration.Attributes;

// selbstdefinierter Namensraum
namespace MyNameSpace
{

// Das Plugin erbt von einer imaginären Basisklasse
public class PlugIn : PlugInBase
{
  // diese private Eigenschaft der Klasse enthält den Dialog
  private IConfigDialog dialog;
```

```
// diese Variablen sollen vom Nutzer konfigurierbar sein
private int zahl;
private string listenWert;
private string jpegDatei;


// der Konstruktor (nach Vorschrift)
public PlugIn(IApplicationContext context) : base(context)
{
  // Dialog erzeugen
  dialog = InitDialog();

  // ...
}



// ... sonstige Implementierung



// Erzeugt den Konfigurationsdialog für ein imaginäres Plugin.
private void InitDialog()
{
  // Erzeugung der Attribute für die konfigurierbaren Eigen-
  // schaften dieses Plugins

  // erste Eigenschaft: ein Zahlenwert, der im Bereich von 1 bis
  // 10 liegen soll
  IntAttrib zahlAttrib = new IntAttrib("Zahl:", "meineZahl");
  zahlAttrib.Minimum  = 1;
  zahlAttrib.Maximum  = 10;
```

```
  zahlAttrib.Default  = "5";
  zahlAttrib.HelpText = "Bitte eine Zahl von 1 bis 10 eingeben!";

  // zweite Eigenschaft: eine Wertliste bestehend aus ("Eins",
  // "Zwei", "Drei")
  EnumAttrib listenAttrib = new EnumAttrib(
    "Wert aus Liste:",
    "meineWertliste",
    new string[] {"Eins", "Zwei", "Drei"});
  listenAttrib.Default  = "Zwei";
  listenAttrib.HelpText = "Eins, Zwei oder Drei?";

  // dritte Eigenschaft: eine JPEG-Datei (zum Öffnen)
  FileAttrib jpegAttrib = new FileAttrib(
    "JPEG-Datei:", // Bezeichnung des Eingabefeldes
    "meinJPEG",    // ID
    "JPEG-Bilder|*.jpg;*.jpeg", // Filter für Dateidialog
    false          // ohne Speichern
  );
  jpegAttrib.HelpText = "Bitte ein Bild auswählen.";

  // Erzeugung des Konfigurationsdialogs über die
  // "Context"-Eigenschaft (der Applikationskontext)
  dialog = Context.CreateConfigDialog(
    "Konfiguration von meinem Plugin", // Dialogtitel
    new IConfigAttrib[] {
      zahlAttrib,
      listenAttrib,
      jpegAttrib
    });
}
```

```
  // Diese Methode ist die Aufforderung der Applikation an das
  // Plugin, sich selbst zu konfigurieren.
  public override bool Configure()
  {
    // den Konfigurationsdialog öffnen
    // -> wurde mit OK abgeschlossen?
    if (dialog.ShowDialog() == DialogResult.OK)
    {
      // Einstellungen vom Dialog holen
      PropertyList config = dialog.Configuration;

      // die Werte aus der Konfiguration an die Variablen übergeben
      // -> dazu werden die oben definierten IDs genutzt!
      zahl       = Int32.Parse(config["meineZahl"]);
      listenWert = config["meineWertliste"];
      jpegDatei  = config["meinJPEG"];
    }

    // ... überprüfen, ob die erhaltenen Werte gültig sind
    bool isValid = ...;

    return isValid;
  }
}

}
```

Der resultierende Dialog kann auf Abbildung A.1 betrachtet werden. Die bei den Konfigurationsattributen angegebenen Bezeichnungen für die Parameter werden links von den Eingabefeldern dargestellt. Die aufgeführten Standardwerte (festgelegt mit der `Default`-Eigenschaft) werden automatisch in die entsprechenden Engabefelder eingetragen. Die `HelpText`-Eigenschaft wird zur Darstellung eines Tooltips genutzt, wenn der Mauszeiger über eines der Eingabefelder geführt wird. Rechts neben dem Eingabe-

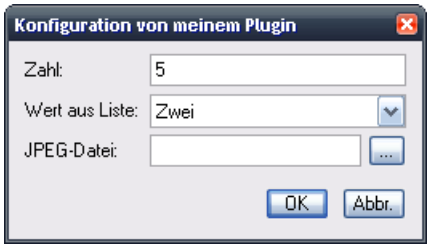

Abbildung A.1.: Dieser Dialog wird für die angegebenen Konfigurationsattribute vom Applikationskontext erzeugt

feld des `FileAttrib`-Konfigurationsattributs wird eine Schaltfläche mit der Beschriftung „...“ erzeugt. Wenn auf sie geklickt wird, öffnet sich ein Dateidialog, der die Auswahl eines JPEG-Bildes fordert.

A.2. Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Brandenburg, 25. Juli 2007 _____

Unterschrift

Literaturverzeichnis

[BHS07] BOERSCH, Ingo ; HEINSOHN, Jochen ; SOCHER, Rolf: *Wissensverarbeitung*. 2. Auflage. Elsevier, Februar 2007. – ISBN 978–3–8274–1844–9

[Hof93] HOFFMANN, Norbert: *Kleines Handbuch Neuronale Netze - Anwendungsorientiertes Wissen zum Lernen und Nachschlagen*. 1. Auflage. Vieweg, 1993. – ISBN 3–528–05239–2

[Koh01] KOHONEN, Teuvo: *Self-Organizing Maps*. 3rd Edition. Springer, 2001. – ISBN 3–540–67921–9

[Pay02] PAYNTER, Gordon. *Attribute-Relation File Format (ARFF)*. April 2002

[Sch03] SCHANDRY, Rainer: *Biologische Psychologie*. 1. Auflage. Beltz PVU, 2003. – ISBN 3–621–27485–5

[Sch06] SCHRÖDER, Sven: *Klassifizieren und Visualisieren von Daten mit Selbstorganisierenden Karten*, Fachhochschule Brandenburg, Diplomarbeit, März 2006

[Ull06] ULLENBOOM, Christian: *Java ist auch eine Insel*. 5. Auflage. Galileo Press, 2006. – ISBN 3–89842–747–1

[Ult01] ULTSCH, Alfred: Eine Begründung der Pareto-80/20 Regel und Grenzwerte für die ABC-Analyse / Universität Marburg. 2001. – Forschungsbericht

[Ult03a] ULTSCH, Alfred: Optimal Density Estimation in Data Containing Clusters of Unknown Structure / Universität Marburg. 2003. – Forschungsbericht

[Ult03b] ULTSCH, Alfred: Pareto Density Estimation: A Density Estimation for Knowledge Discovery / Universität Marburg. 2003. – Forschungsbericht

[Ult03c] ULTSCH, Alfred: U*-Matrix: a Tool to visualize Clusters in high dimensional Data / Universität Marburg. 2003. – Forschungsbericht

[Ult04] ULTSCH, Alfred: Density Estimation and Visualization for Data containing Clusters of unknown Structure / Universität Marburg. 2004. – Forschungsbericht

[UM05] ULTSCH, Alfred ; MÖRCHEN, Fabian: ESOM-Maps: tools for clustering, visualization, and classification with Emergent SOM / Universität Marburg. 2005. – Forschungsbericht

[VA00] VESANTO, Juha ; ALHONIEMI, Esa: Clustering of the Self-Organizing Map. In: *IEEE-NN* 11 (2000), Mai, Nr. 3, S. 586–600

[Ves99] VESANTO, Juha: SOM-based data visualization methods. In: *Intelligent-Data-Analysis* 3 (1999), S. 111–130

[WF01] WITTEN, Ian H. ; FRANK, Eibe: *Data Mining*. 1. Auflage. Hanser, 2001. – ISBN 3–446–21533–6

[Zel03] ZELL, Andreas: *Simulation neuronaler Netze*. 4. Auflage. Oldenbourg, 2003. – ISBN 3–486–24350–0