Fachbereich 4: Informatik

LexLearn – Emergenz eines gemeinsam genutzten Lexikons

Diplomarbeit

zur Erlangung des Grades eines Diploms
im Studiengang Informatik

vorgelegt von

Christian Klein
Dennis Fuchs

Betreuer: Prof. Dr. Klaus G. Troitzsch,
Institut für Wirtschafts- und Verwaltungsinformatik, Fachbereich 4
Zweitgutachter: Dr. Michael Möhring,
Institut für Wirtschafts- und Verwaltungsinformatik, Fachbereich 4

Koblenz, im September 2007

Erklärung

Wir versichern, dass wir die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt haben.

Mit der Einstellung dieser Arbeit in die Bibliothek sind wir einverstanden. Der Veröffentlichung dieser Arbeit im Internet stimmen wir zu.

Die Unterkapitel 2.1, 2.2, 4.1 und 6.2 wurden weitestgehend von Christian Klein erarbeitet.
Die Unterkapitel 2.3, 2.4, 4.2 und 6.3 wurden weitestgehend von Dennis Fuchs erarbeitet.
Alle weiteren Kapitel wurden in Teamarbeit verfasst.

Koblenz, den 28.09.2007 _____ _____

(Unterschrift) (Unterschrift)

Kurzfassung

Die vorliegende Diplomarbeit untersucht die Emergenz eines gemeinsam genutzten Lexikons anhand einer Implementierung des Modells von Edwin Hutchins und Brian Hazlehurst.

Zunächst erläutert diese Arbeit das oben genannte Modell und analysiert sowohl verschiedene Simulationstechniken, insbesondere die der Multi-Agenten-Simulation, als auch den Aufbau und die Funktionalität künstlicher neuronaler Netze, welche als Hauptbestandteile des Modells angesehen werden können.

Der Modellbeschreibung folgen Entwurf, Architektur und eine detaillierte Erläuterung der Implementierung des Werkzeugs LexLearn. LexLearn ist ein in Java komplett neu konzipiertes und implementiertes Programm und bietet dem Benutzer die Möglichkeit, die Emergenz einer gemeinsam genutzten Sprache innerhalb verschiedener Agentengemeinschaften zu simulieren und zu analysieren.

In mehreren Simulationsdurchläufen werden sowohl die Ergebnisse von Edwin Hutchins und Brian Hazlehurst reproduziert als auch weitere Erkenntnisse durch neu erstellte Simulationen gewonnen.

Die Arbeit entstand an der Universität Koblenz-Landau im Rahmen des EU Projekts „*Emergence in the Loop*“ (EMIL).

Abstract

This thesis investigates the emergence of a shared lexicon with a reimplementation of a model introduced by Edwin Hutchins and Brian Hazlehurst.

At first the thesis describes the abovementioned model and analyses both different simulation techniques, in particular multi agent simulation and design and functionality of artificial neural networks which can be seen as the essential parts of the model.

The model description is followed by draft, architecture and a detailed illustration of an implementation named LexLearn. LexLearn is an entirely newly-designed java based tool which provides users the opportunity to simulate and analyse the emergence of a commonly used language within different communities of agents.

The results of Edwin Hutchins and Brian Hazlehurst are replicated by several simulation runs. Furthermore new cognitions are obtained by newly created simulations.

This thesis was written at University of Koblenz-Landau within the EU project „*Emergence in the Loop*“ (EMIL).

Inhaltsverzeichnis

Abbildungsverzeichnis

1. Einführung

„Language is a purely human and noninstinctive method of communicating ideas, emotions, and desires by means of a system of voluntarily produced symbols. […] There is no discernible instinctive basis in human speech as such, however much instinctive expressions and the natural environment may serve as a stimulus for the development of certain elements of speech, however much instinctive tendencies, motor and other, may give a predetermined range or mold to linguistic expression." [Sap21, S. 5]

„Daß wir miteinander reden können, macht uns zu Menschen."[1]

Edward Sapir[2] sieht in dem oben aufgeführten Zitat Sprache als eine rein menschliche, nicht angeborene Methode an, über Ideen, Gefühle und Wünsche unter Berücksichtigung einer Symbolmenge zu kommunizieren. Die Einschränkung, dass eine Sprache nur zwischen Menschen existieren kann, ist sicherlich in der heutigen Zeit nicht mehr gültig, da sowohl weitere natürliche Lebensformen als auch künstliche Intelligenz Sprachen ausgebildet haben bzw. ausbilden können.
Eine gemeinsame Sprache und die damit verbundene Möglichkeit, Ideen, Gefühle und Wünsche zu äußern, ist der Grundpfeiler eines sozialen Zusammenlebens. Karl Theodor Jaspers sieht die soziale Interaktionsfähigkeit als Folge der Emergenz einer Sprache, da nach ihm die Fähigkeit, miteinander reden zu können, d. h. eine Sprache ausgebildet zu haben, einen Menschen, also das Lebewesen mit der höchsten Komplexität des sozialen Zusammenlebens ausmacht. Wie hängen soziale Verbindungen und Sprachentstehung zusammen?
Edward Sapir sieht neben der sozialen Komponente die natürliche Umgebung als einen „Stimulus" der Sprachausbildung.
Welche Interaktionsformen zwischen Individuen und welche Anreize der natürlichen Umgebung sind notwendig, um eine gemeinsam genutzte Sprache auszubilden? Wie entsteht eine Sprache? Wie ist ein solcher Prozess modellierbar?
Ein Modell des Prozesses der Sprachentwicklung der realen Welt kann nur sehr vereinfacht dargestellt werden. Einerseits existiert eine Vielzahl von Komponenten des sozialen Zusammenlebens, andererseits ist die Anzahl der Elemente der natürlichen Umgebung nahezu unendlich.
Hutchins und Hazlehurst erläutern in ihrem Modell [HH95] aus dem Jahre 1995 einen Ansatz, der diesen Prozess simulierbar macht. In einer Gemeinschaft von Agenten, deren Wortschatz dem eines Neugeborenen gleicht, kreiert jeder Agent separat, an Hand von visuellen Reizen, zufällig Worte für das, was er sieht. Durch

[1] Zitat von Karl Theodor Jaspers (1883 – 1969), deutscher Psychiater und Philosoph.
[2] Edward Sapir (1884 – 1939), amerikanischer Ethnologe und Linguist, Professor für Anthropologie und Linguistik an den Universitäten Chicago und Yale.

Kommunikation und gegenseitige Anpassung der Worte der Agenten entsteht in einem mehrere tausend Schritte dauernden Prozess letztendlich ein gemeinsam genutztes Vokabular, ein Lexikon.
Die visuellen Reize der Umgebung sind hier auf eine kleine Menge von Eingabemustern beschränkt, welche Elemente der realen Welt repräsentieren. Die einzige soziale Komponente, die zwischen Agenten ausgebildet ist, ist die Tatsache, dass sie miteinander sprechen können.
Das Ziel dieser Diplomarbeit ist die Implementierung dieses Modells. Mit einem vollkommen neu erstellten Programm soll es möglich sein, nahezu beliebig viele Agenten mit neuronalen Netzen variabler Größe und frei definierbaren visuellen Szenen den vorhin beschriebenen Prozess durchleben zu lassen.

Der chronologische Aufbau dieser Arbeit orientiert sich an den aus der Softwaretechnik bekannten Grundsätzen zur Softwareentwicklung. Einer ausführlichen Beschreibung des Modells von Hutchins und Hazlehurst und den damit verbundenen theoretischen Hintergründen in Kapitel 2 folgen Anforderungsdefinition, softwarearchitektonische Grundlagen sowie ein Entwurf einer möglichen Implementierung des oben genannten Modells in Kapitel 3. Die Struktur der Anwendung LexLearn, einer Realisierung des beschriebenen Entwurfs wird in Kapitel 4 beschrieben. Die von Hutchins und Hazlehurst erzielten Ergebnisse mehrerer Simulationsdurchläufe werden in Kapitel 5 den unter Verwendung von LexLearn erlangten Simulationsergebnissen gegenübergestellt. Das abschließende sechste Kapitel ordnet diese Arbeit in das EU-Projekt „EMIL – Emergence in the Loop“ ein und gibt einen kurzen Ausblick über mögliche Erweiterungen des von Hutchins und Hazlehurst beschriebenen Modells. Im Anhang der Arbeit befindet sich ein Benutzerhandbuch zu LexLearn in englischer Sprache.

2. Modell zur Entstehung eines gemeinsam genutzten Lexikons

Dieses Kapitel beinhaltet sowohl eine Erläuterung der Begriffe *Modell* und *Simulation* im Sinne der Informatik als auch eine Erklärung grundlegender Bausteine der Simulationstechniken, die im Rahmen dieser Diplomarbeit verwendet werden.
Es werden zunächst die Grundlagen der *Multi-Agenten-Simulation* erklärt, die dazu verwendet wird, das Äußere der Agenten, ihre Interaktion miteinander und ihrer Umwelt, zu simulieren. Das darauf folgende Unterkapitel behandelt den Aufbau und die Funktionsweise *neuronaler Netze*, die in LexLearn das Gehirn bzw. den Kern der Agenten darstellen. Im letzten Unterkapitel wird das Modell von Hutchins und Hazlehurst beschrieben.
Das Ziel dieses Kapitels ist es, eine Basis für einen *Entwurf*, eine *Architektur* und eine *Implementierung*, die in den folgenden Kapiteln beschrieben werden, zu schaffen.

2.1 Simulation und Modell

Dieses Unterkapitel orientiert sich an den Ausführungen von K. G. Troitzsch und N. Gilbert in [TG05, S. 15-27].
Simulation ist eine Vorgehensweise zur Analyse dynamischer Phänomene innerhalb real existierender Systeme, welche *Zielsysteme* genannt werden. Da ein Zielsystem in der Realität generell sehr komplex ist und dadurch eine computergestützte Simulation in gleicher Komplexität kaum in der Lage wäre, zukunftsbezogene Daten zu liefern, wird das Zielsystem vereinfacht, es wird ein Modell von ihm erstellt. Das Bestreben dieser Modellierung ist es, ein möglichst einfaches Modell zu kreieren, das sowohl komplex genug ist, alle in der Realität für einen Sachverhalt relevanten Faktoren abzubilden, als auch kompakt genug ist, um das Modell mit einem vertretbaren Zeitaufwand simulieren zu lassen. Simulation bedeutet in diesem Sinne ein temporäres Fortschreiten des Modells, in dem sich die Parameter der Bestandteile des Modells, beispielsweise Agenten und ihre Umgebung, anhand stochastischer Methoden verändern. Das Ziel der Simulationstechnik ist, dass die Phänomene, die in einer zukunftsgerichteten Simulation des Modells eintreten, ebenfalls in der Realität, also im Zielsystem, in der Zukunft zu beobachten sein werden.

Innerhalb der Sozialwissenschaften ist das Zielsystem stets ein dynamisches Gebilde, bestehend aus einer *Struktur* und einem *Verhalten*. Dadurch bedingt muss das Modell ebenfalls dynamisch sein, es muss sich mit der Zeit verändern können.

Wie aber kann abgeschätzt werden, welche Parameter des Zielsystems mit in ein Modell übernommen werden müssen und auf welche verzichtet werden kann? Wie aber kann ein Modell „validiert[3]“ werden?

Die wichtigste Voraussetzung für ein gelungenes Modell ist die Existenz zuverlässiger und präziser Daten des zu simulierenden Systems über eine möglichst lange Zeitspanne. Denn dann besteht die Möglichkeit, mit dem abstrahierten Modell genau die Periode ex post zu simulieren, von der bereits Daten des Endzustands vorhanden sind. Ähneln die durch die Simulation erhaltenen Daten den erhobenen Daten der realen Welt, hat das Modell seine Ähnlichkeit für den gegebenen Zeitraum belegt.

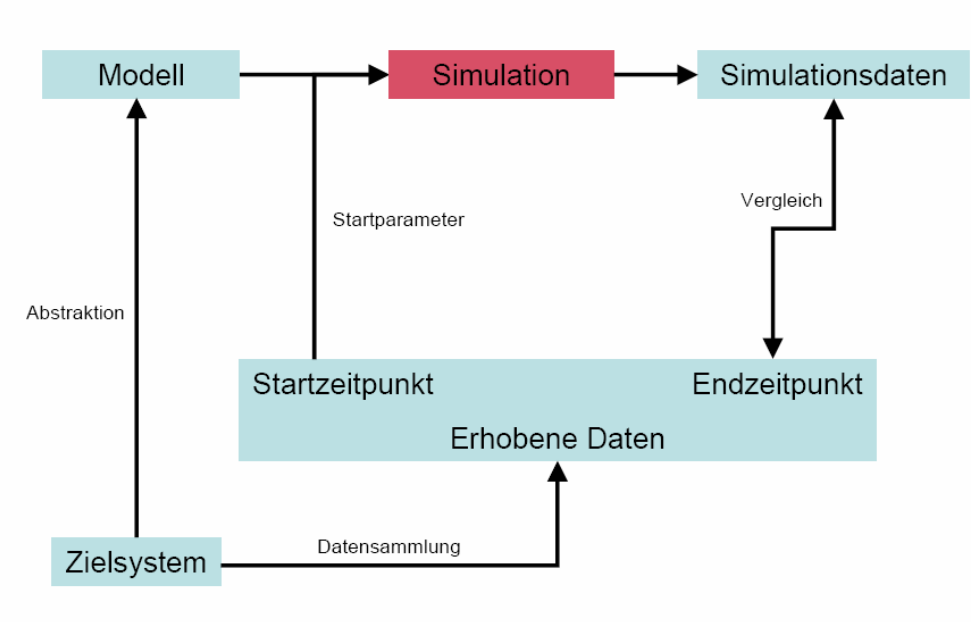


Abb. 1 Verifikation von Simulationsmodellen
Quelle: Neuzeichnung [TG05, S. 17, Abb. 2.2]

Der soeben beschriebene Effekt hat keine Aussagefähigkeit über eine zukunftsbezogene Ähnlichkeit zwischen Zielsystem und Modell, jedoch ist im Allgemeinen davon auszugehen, dass Modelle, die über einen längeren Simulationszeitraum ihre Zuverlässigkeit bewiesen haben, auch eine verlässliche Prognose des Verhaltens des realen Systems in zumindest nahe gelegener Zukunft erstellen können.
Die Simulationen dieser Modelle enden demnach nicht mit dem Endzeitpunkt der verfügbaren erhobenen Daten, sondern sie simulieren die Zukunft. Dies wird in

[3] Validierung ist hier im Sinne der externen Validierung, der analytischen Beweisführung, ob eine konsistente Beziehung zwischen den Ergebnissen des Software-Maßes und den empirisch verfügbaren, externen Attributen besteht (vgl. [EB96, S. 260]), zu sehen.

Abbildung 2 verdeutlicht. Die aktuellsten erhobenen Daten sind demnach der Startdatensatz der zukunftsgerichteten Simulation. Die in der Simulation auftretenden zukünftigen Phänomene geben eine Idee dafür, was im Zielsystem in Zukunft geschehen kann. Es können also Rückschlüsse auf zukünftiges Verhalten der modellierten Objekte gezogen werden.

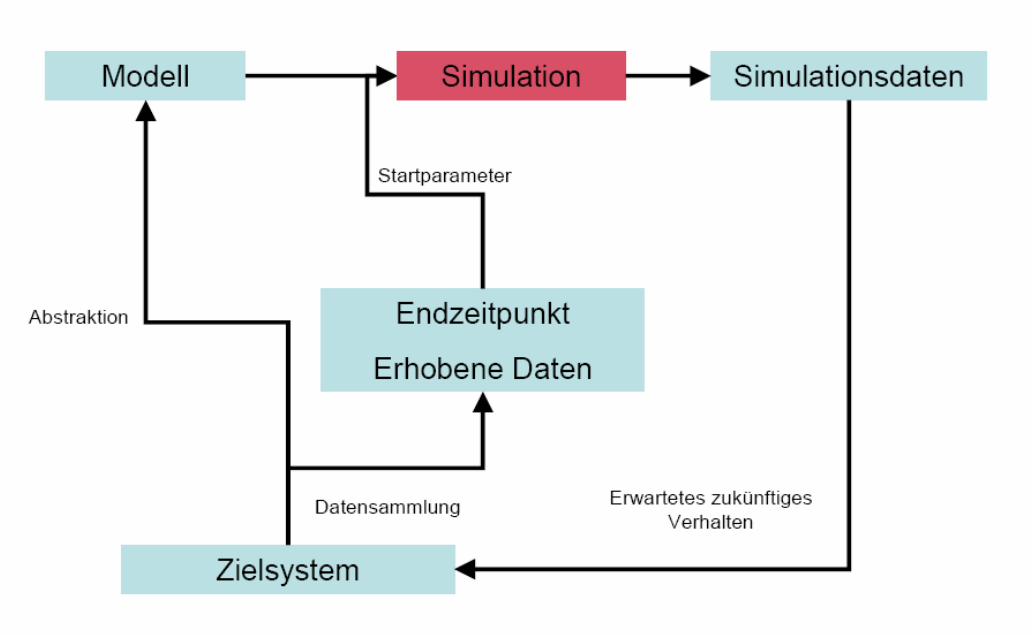


Abb. 2 Rückschlüsse durch eine Simulation

Eine Simulation der Entwicklung der Bevölkerungsstruktur ist ein triviales Beispiel, um den Nutzen einer solchen Simulation zu verdeutlichen. Sie erlaubt es, ex ante eine zukünftige Gefahr der Überalterung einer Gesellschaft zu zeigen, und anhand veränderter Parameter innerhalb der Simulation lassen sich ebenfalls mögliche Auswege finden.
Die hier beschriebenen Grundsätze sind auf das Modell von Hutchins und Hazlehurst nur bedingt übertragbar, da es keine zuverlässigen erhobenen Daten über den weit in die Vergangenheit zurückliegenden Prozess der Sprachentstehung gibt. Versuche, diesen Prozess an realen Menschen wieder zu erleben, sind in der heutigen Zeit ethisch nicht vertretbar und wurden bekanntlich von Friedrich II[4] durchgeführt, der eine Gruppe Säuglinge von Ammen säugen ließ, denen es jedoch verboten war mit den Kindern zu sprechen oder ihnen körperliche Zuwendung zukommen zu lassen. Das Ziel dieses Versuchs bestand darin, den Entstehungsprozess der „menschlichen Ursprache“ zu rekonstruieren, die die Säuglinge unter sich ausbilden sollten. Er scheiterte, da die Neugeborenen aufgrund mangelnder Zuwendung und menschlicher Nähe zu Grunde gingen. [Sal13]

[4] Friedrich II (1194 – 1250), römisch-deutscher Kaiser von 1220 bis 1250.

2.2 Multi-Agenten-Simulation

Dieses Kapitel widmet sich speziell der Technik der Multi-Agenten-Simulation, einem mächtigen Simulationskonzept, welches die Möglichkeit bietet, komplexe Verhaltensweisen von Individuen und ihre Interaktion sowohl miteinander als auch mit ihrer Umwelt zu simulieren. Im Folgenden werden diese beiden Bestandteile der Multi-Agenten-Modelle genauer erklärt. Der Unterpunkt Agent orientiert sich dabei an den Erläuterungen von K. G. Troitzsch und N. Gilbert in [TG05, S. 172-198].

2.2.1 Agent

Während es noch keine allgemein gültige Definition für einen Software-Agenten gibt, wird der Begriff gewöhnlich dazu benutzt, ein Programm zu beschreiben, das seine eigenen, auf seiner Sichtweise der Operationsumgebung basierenden Aktionen kontrollieren kann. (siehe [HS98])
Das Ziel ist es, einen Software-Agenten zu kreieren, der intelligent mit seiner Umwelt interagiert. Das Konzept eines Software-Agenten wird oft dazu benutzt, eine zielgerichtete Aufgabe eigenständig zu erfüllen. Das Vorbild für eine solche Struktur eines Programms ist der Mensch selbst. So sollen auch Agenten Konzepte wie einen freien Willen und eine Absicht, seine eigenen Ziele zu verwirklichen, besitzen.

Eigenschaften

Software-Agenten besitzen folgende Eigenschaften (Woolbridge und Jennings 1995, in Analogie zu [TG05, S. 173]):

- **Eigenständigkeit**
  Agenten agieren, ohne dass andere direkte Kontrolle über ihre Aktionen oder inneren Zustände haben.

- **Sozialvermögen**
  Agenten interagieren mit anderen Agenten durch eine formale Sprache.

- **Reaktionsfähigkeit**
  Agenten sind in der Lage ihre Umwelt zu erkennen und Schlüsse daraus zu ziehen.

- **Eigeninitiative**
  Agenten können eine Eigeninitiative ergreifen, indem sie ein zielgerichtetes Verhalten ausüben.

Agenten werden oft mit einem gewissen Grad an Absichtlichkeit ausgestattet, wodurch ihr Verhalten menschlichen Bedürfnissen wie Willen, Glauben oder gar Emotionen zu folgen scheint. Diese Eigenschaften sind für den Modellbildungsprozess von großer Bedeutung, es sollte jedoch klar sein, dass ein Software-Agent diese Eigenschaften nicht besitzt, sondern lediglich menschliches Verhalten simuliert.
Daraus folgend lassen sich einige Attribute darlegen, die mit Software-Agenten simuliert werden können:

Simulationsattribute (siehe [TG05, S. 174-177])

- **Wissen und Belief**
  Agenten treffen ihre Entscheidungen anhand von dem, was sie über ihre Umwelt und über andere Agenten wissen. Einige Informationen mögen nicht der Wahrheit entsprechen, möglicherweise verursacht durch falsche Wahrnehmung, falsche Folgerungen oder unvollständiges Wissen. Eine derart eventuell fehlerbehaftete Information wird Belief genannt, um sie von wahrem Wissen zu unterscheiden. Die Agenten im Modell von Hutchins und Hazlehurst erlernen die Zuordnung einer visuellen Szene zu einem Wort. Bedingt durch den Simulationsablauf ist es dabei nicht notwendig, zwischen Wissen und Belief zu unterscheiden, eine falsche Wahrnehmung ist im hier behandelten Basismodell nicht vorhergesehen, Folgerungen und unvollständiges Wissen ebenfalls nicht. Dies könnte jedoch eine sinnvolle Erweiterung sein und wäre damit ein mögliches Thema der weiteren Forschung.

- **Inferenz**
  Aus einer gegebenen Menge an Annahmen sollen Agenten in der Lage sein weitere Informationen zu schlussfolgern. Dieses Attribut wird in dem in [HH95] beschriebenen Grundmodell nicht angewendet, wohl aber im erweiterten Modell [HH91][5]. In einer Implementierung dieses Modells läge der Fokus aber ebenfalls auf diesem Aspekt.

- **Soziale Modelle**
  Einige Agententypen sollen in der Lage sein, Beziehungen zwischen anderen Agenten in ihrer Welt zu erkennen oder auch selbst auszubilden. Dieses weit verbreitete Simulationsattribut findet im hier betrachteten Modell keine Anwendung, es wäre jedoch eine zweckmäßige Erweiterung, besonders um eine mögliche „sozialbedingte Dialektbildung“ zu simulieren. Auch dies könnte ein sinnvolles Forschungsobjekt weiterer Arbeiten sein.

[5] In diesem erweiterten Modell erlernen die Agenten die Zuordnung der visuellen Szenen der Mondphasen zu den Gezeiten.

- **Wissensrepräsentation**
  Die Agenten benötigen eine Struktur, um das Wissen und die subjektiven Annahmen der Welt in einer geeigneten Weise zu speichern. Dazu gibt es drei vorherrschende Ansätze:

  - die Speicherung in prädikatenlogischen deklarativen Aussagen;
  - das Halten in semantischen baumartigen Strukturen, in der die allgemeinen Fakten an der Wurzel liegen und die Spezifizierung der Information mit der Baumtiefe zunimmt;
  - die Wissensrepräsentation durch neuronale Netze. Diese Variante ist im hier behandelten Modell vorgesehen und daher wird die Funktionsweise der neuronalen Netze in Unterkapitel 2.3 genauer erklärt.

- **Ziele**
  Bedingt durch den Wunsch, Agenten autonom und zweckbestimmt agieren zu lassen, ist es notwendig, sie mit einem internen Ziel auszustatten, dessen Verwirklichung die Maxime ihres Handelns ist. Agenten können ebenfalls mehrere, gegebenenfalls miteinander konkurrierende Ziele erhalten, auch eine Hierarchie in Ziele und untergeordnete Subziele ist gelegentlich zweckmäßig. Hutchins und Hazlehurst haben die Agenten ebenfalls mit der Erfüllung zweier Ziele ausgestattet, die sich durchaus in Teilen widersprechen können. Eine genauere Erläuterung dieser beiden Ziele befindet sich im Unterkapitel 2.4.

- **Planung**
  Ein Agent benötigt einen Algorithmus, um entscheiden zu können, welches zukünftige Verhalten seinerseits am wahrscheinlichsten zu einem Erreichen seiner Ziele führt. Diese Art der Planung impliziert ein Verständnis für die Existenz der Wege, die den Agenten zu seinem Ziel führen. Er muss ebenfalls erkennen können, welche Vorzustände und Aktionen auf dem Weg zum Ziel eingenommen und ausgeführt werden müssen. Dieser Algorithmus der Wegplanung wird so lange rekursiv durchlaufen, bis der Agent seinen jetzigen Zustand erreicht. Daraufhin versucht er einen der theoretisch möglichen, geplanten Wege zu beschreiten, um sein Ziel zu erreichen.
  Diese Eigenschaft wird von den Agenten in diesem Modell jedoch nicht gefordert, ihre Auflistung geschieht aus Gründen der Vollständigkeit.

- **Sprache**
  Alle Multi-Agenten-Modelle beinhalten eine Form von Interaktion zwischen Agenten oder mindestens eine indirekte Interaktion, indem Agenten nur mit der Umwelt interagieren und andere Agenten eine veränderte Umwelt antreffen. Es existieren vielfältige Interaktionsmöglichkeiten, von einem einfachen Austausch von Informationen bis hin zu einem komplexen Aushandeln von Verträgen. Solche Interaktionen können entweder über eine, eventuell fehleranfällige, formale Sprache bewerkstelligt werden oder aber durch den direkten Austausch von Informationen von „Gehirn zu Gehirn“. Die

Auswahl einer der beiden Kommunikationsarten ist abhängig vom Modell. Es kann durchaus gewünscht sein, eine fehleranfällige Kommunikationsform zu benutzen, beispielsweise um die Auswirkungen von Sprachveränderungen zu simulieren.
Im hier behandelten Modell ist eine direkte Interaktion zwischen zwei Agenten durch Sprache vorhergesehen. Da die Übertragung aber ohne Störsignale stattfindet, entspricht sie hier der direkten Übertragung von Informationen von „Gehirn zu Gehirn" bzw. von neuronalem Netz zu neuronalem Netz.
Die Implementierung einer fehleranfälligen Art der Interagentenkommunikation wäre ebenfalls eine mögliche Erweiterung dieses Modells.

- **Emotionen**
  Es ist bis heute eine ungeklärte Frage, ob emotionale Zustände Entitäten an sich sind oder aber Merkmale, die aus anderen bewussten und unbewussten Zuständen entstehen. Ebenfalls ist der Zusammenhang zwischen Zielen und Emotionen unklar. Daher ist auch die Implementierung von Emotionen stark abhängig vom entsprechenden Modell.
  Im Modell von Hutchins und Hazlehurst spielen Emotionen jedoch keine Rolle, daher wird auf eine genauere Erläuterung hier verzichtet.

Architektur eines Agenten (siehe [TG05, S. 178-180])

1. Produktionssystem

Agenten in Multi-Agenten-Simulationen nutzen meist eine Art Regelsystem, deren einfachste Ausprägung ein Produktionssystem ist. Ein Produktionssystem besteht aus drei Komponenten:

- *Regelmenge*
- *Arbeitsspeicher*
- *Regelinterpreter*

Regeln der Regelmenge bestehen jeweils aus zwei Teilen, einer *Bedingung*, die spezifiziert, wann genau eine Regel angewendet werden kann, und einer *Aktion*, die bestimmt, was genau geschehen soll, wenn die Bedingung eintritt. Die Überprüfung, ob eine Bedingung zu einem bestimmten Zeitpunkt erfüllt ist, geschieht, indem der Arbeitsspeicher des Agenten, der sämtliche dem Agenten bewussten Fakten wie seine inneren Zustände, seine Ziele und sein Wissen über die Umwelt beinhaltet, mit dem Bedingungsteil der Regel verglichen wird. Dies ist die Aufgabe des Regelinterpreters. In jedem Zug des Agenten überprüft er, ob die Bedingung einer Regel eingetreten ist, und falls ja, so erledigt er die auszuführende Aktion.
Der Hauptvorteil eines Produktionssystems besteht darin, dass der Modellierer nicht im Voraus entscheiden muss, in welcher Reihenfolge die zutreffenden

Aktionen ausgelöst werden sollen. Diese soll der Agent selbstständig, in einer, in seiner aktuellen Situation sinnvollen Reihenfolge, abhängig von seiner aktuellen Arbeitsspeicherbelegung und dementsprechend seiner vorhergegangenen Erfahrungen, selbstständig auswählen.
Was geschieht, wenn es mehrere Regeln gibt, deren Bedingungen zu einem Zeitpunkt *t* erfüllt sind? Dazu existieren mehrere Strategien. Entweder können alle Aktionen dieser Regeln ausgelöst werden oder aber auch nur die Aktion, deren zugehörige Regel zuerst überprüft wird. Eine etwas komplexere Lösungsform ist eine interne Konfliktbehandlung im Agenten, die anhand bestimmter Kriterien eine Teilmenge der zutreffenden Regeln auswählt, deren Aktionen ausgeführt werden. Dies ist jedoch dem Modellierer überlassen und soll daher hier nicht weiter erläutert werden.
Nachdem eine Aktion ausgeführt wurde, werden erneut alle Regeln nach zutreffenden Bedingungen überprüft. Dies ist notwendig, da die vorher ausgeführte Aktion die Inhalte des Arbeitsspeichers verändert haben kann und so der Bedingungsteil einer Regel eventuell nicht mehr erfüllt ist. Generell beeinflussen Aktionen entweder direkt den Arbeitsspeicher des Agenten oder aber sie verändern die Umwelt in einer für den Agenten begreifbaren Art und Weise, was ebenfalls zu einer Veränderung seines Arbeitsspeichers führt.

2. Puffer

In eine Implementierung eines Agenten werden dieser theoretischen Struktur noch Puffer hinzugefügt, um eine serielle Abarbeitung der Ein- und Ausgaben zu gewährleisten. Dies umfasst sowohl jeweils einen Puffer für ein- und ausgehende Nachrichten als auch einen Puffer für Aktionen, die vom Agenten ausgeführt werden.

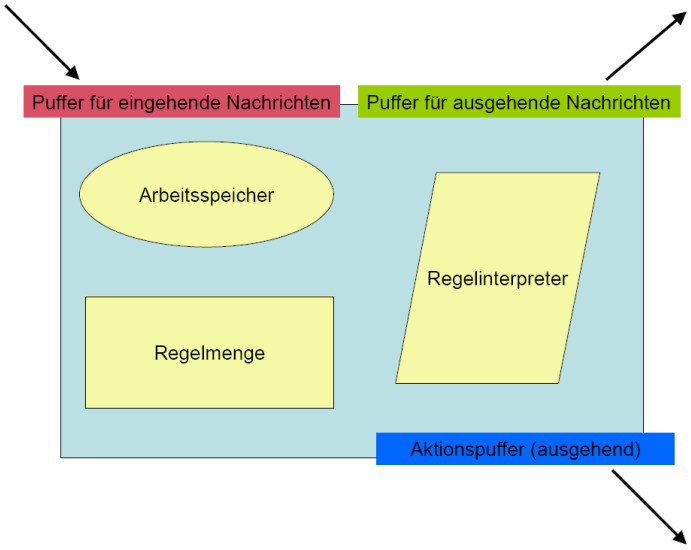


Abb. 3 Aufbau eines Agenten
Quelle: angelehnt an Foliensatz Simulation, K. G. Troitzsch, 15.12.2006

Im Modell von Hutchins und Hazlehurst ist es vorgesehen, die Agenten mit einem neuronalen Netz auszustatten. So werden die drei Elemente Arbeitsspeicher, Regelmenge und Regelinterpreter gegen ein neuronales Netz ausgetauscht, das die Aufgaben aller drei aufgezählten Elemente übernimmt. Das neuronale Netz ist der Arbeitsspeicher eines Agenten, die Regeln sind in der Gewichtung der Kanten kodiert und daher können sie nicht mehr getrennt betrachtet werden. Eine genauere Erläuterung der Funktionsweise neuronaler Netze befindet sich in Unterkapitel 2.3.

2.2.2 Umwelt

Die Simulationsumgebung ist das Universum, in dem die Agenten existieren. Es beinhaltet sowohl eine räumliche als auch eine zeitliche Komponente.

Raum

Räumlich gibt es die Möglichkeit, die Agenten sich in einer gitternetzartigen Simulationswelt bewegen zu lassen. Dieses Verfahren bildet eine geografische Umgebung ab und so ist es möglich, eine Umwelt zu kreieren, in der an bestimmten Koordinaten bestimmte Ereignisse eintreten, in gewissen Regionen andere Umstände herrschen als in anderen Regionen. Hierbei ist eine starke Ähnlichkeit zum zellulären Automaten erkennbar, jedoch ist der Agentenaufbau den weniger komplexen Zellen überlegen.
Ein einfaches Beispiel ist eine Landkarte, die aus Gitterzellen besteht, die entweder Erdboden oder aber Wasser repräsentieren. So ist es denkbar ein einfaches Modell eines Landesabschnitts mit Flüssen und Seen nachzubilden. Es ist leicht ersichtlich, dass, sobald ein Agent auf eine Wasserzelle stößt, andere Aktionen möglich sind als bei einer Zelle aus festem Untergrund. Ein triviales Beispiel ist die Entscheidung der Fortbewegungsart, ob er also in diesem Fall besser geht oder schwimmt.

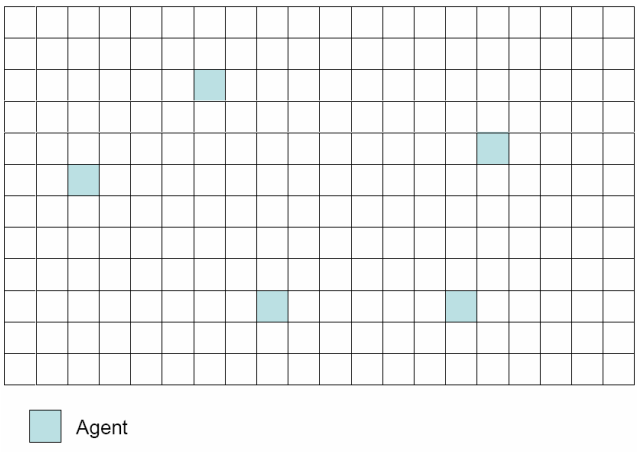


Abb. 4 Agenten in Gitternetzumgebung

Eine weitere Möglichkeit der Darstellung eines Agentennetzwerks ist die eines gewichteten Graphen. Die Agenten als Knoten des Graphen sind durch Kanten miteinander vernetzt. Die Entfernung zwischen den Agenten kann in diesem Falle durch ein Kantenattribut repräsentiert werden. Der Vorteil dieser Darstellungsmöglichkeit liegt darin, dass Graphen nicht nur zur räumlichen Entfernungsangabe benutzt werden können, sondern auch andere Attribute beherbergen können, beispielsweise gegenseitige Sympathie oder aber Wahrscheinlichkeiten der Kommunikation mit einem anderen Agenten.

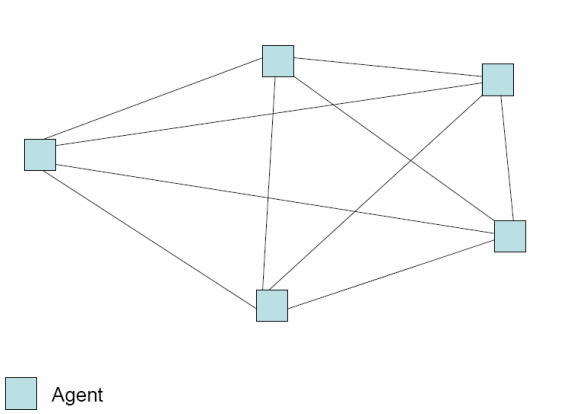


Abb. 5 Agenten verbunden durch Graphen

Die beiden hier genannten Möglichkeiten sind ebenfalls kombinierbar. Es ist oft von Vorteil beide Topologien in einem einzigen Modell zu implementieren, beispielsweise in einer Simulationswelt, in der sich Agenten in einem Raum fortbewegen können und in dem jeder Agent eine gewisse Sympathie den anderen Agenten entgegenbringt. Während der Raum durch ein Gitternetz repräsentiert wird, lässt sich die Sympathie durch einen gerichteten Graphen darstellen. In diesem Modell ist es fortan möglich, diese beiden Distanzen zu verknüpfen, indem die Wahrscheinlichkeit, dass eine Kommunikation zwischen zwei Agenten stattfindet, die sowohl von der räumlichen Entfernung im Gitternetzmodell als auch von der gegenseitigen Sympathie zweier Agenten abhängt.
Nachdem Agenten in einer Umgebung positioniert worden sind, benötigen sie die Fähigkeit ihre Umgebung wahrzunehmen und zu verändern. Diese „Sinneswahrnehmungen“ werden beispielsweise für eine Inter-Agenten-Kommunikation benötigt, sodass Agenten die Fähigkeit zu *sprechen* und zu *hören* besitzen müssen. Das Hören einer Äußerung eines anderen Agenten entspricht in diesem Falle der Wahrnehmung, das Sprechen dem Verändern der Umwelt.

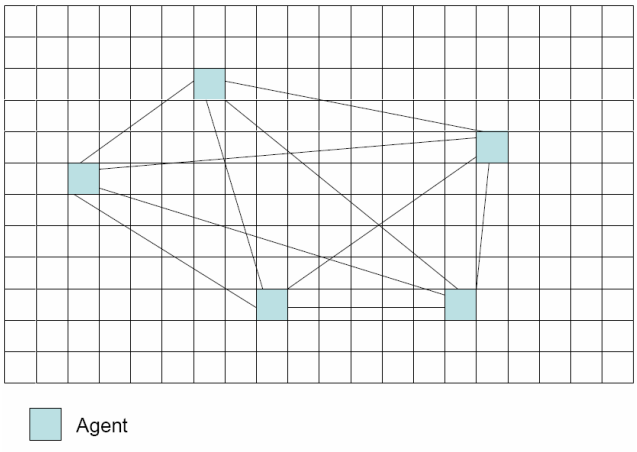


Abb. 6 Agenten in einer kombinierten Umgebung

Zeit

Es gibt zwei Möglichkeiten, den Ablauf der Zeit in einer Simulation nachzubilden. In einer *kontinuierlichen Simulation* schreitet die Zeit iterativ voran. Innerhalb eines Zeitschritts arbeiten alle Agenten idealerweise parallel. Da dies in der Praxis, aufgrund sequenziell arbeitender Computer, nicht ohne Weiteres möglich ist, muss die Parallelität simuliert werden, im einfachsten Falle durch ein Rundlauf-Verfahren, in dem der nächste Agent zufällig ausgesucht wird. Da diese Simulationstechnik keine echte Parallelität gewährleistet, sondern die zufällige Reihenfolge der Agentenwahl den Ablauf der Simulation erheblich beeinflusst, wird das Modell im Falle einer parallel ablaufenden Simulation um einen Nachrichten- und Aktionspuffer in der Umgebung erweitert. Dort werden alle in einer Runde ausgesendeten Nachrichten und durchgeführten Aktionen der Agenten zwischengespeichert und den Agenten gänzlich im nächsten Simulationsschritt geliefert.
Eine zweite Möglichkeit ist die *diskrete Simulation* von Zeit. Hier arbeiten die Agenten ähnlich, sie werden jedoch nicht in jedem Simulationsschritt angesprochen. Vielmehr senden sie, nachdem sie eine Aktion ausgeführt haben, eine Nachricht an den Simulationszeitgeber, die die Dauer der aktuell ausgeführten Aktion beinhaltet und aus der sich somit der Zeitpunkt des nächsten Aufrufs dieses Agenten errechnen lässt.

2.3 Künstliche neuronale Netze

Im folgenden Unterkapitel wird eine Einführung bis hin zur detaillierten Beschreibung von Lernalgorithmen künstlicher neuronaler Netze gegeben. Die Idee, vereinfachte mathematische Modelle, die den Nervenzellvernetzungen im menschlichen Gehirn ähneln, zu entwickeln (vgl. [Rum94, S. 87]), ist ein Zweig der künstlichen Intelligenz und Forschungsgegenstand der Neuroinformatik[6]. Neben der hohen Fehlertoleranz und Parallelität ist die Lernfähigkeit künstlicher neuronaler Netze ihr größter Vorteil, da sie unterschiedliche Aufgaben anhand von Trainingsbeispielen erlernen. Innerhalb des Hutchins/Hazlehurst-Modells werden Agenten jeweils mit einem künstlichen neuronalen Netz ausgestattet, sodass eine genauere Betrachtung unabdingbar ist.

2.3.1 Aufbau und Bestandteile

Künstliche neuronale Netze bestehen aus zwei verschiedenen Mengen: einer Menge *N* von stark idealisierten *Neuronen*, die als Basisprozesseinheit gesehen werden können, und einer Menge *V* von *Verbindungen*. Die Struktur der Netze ist die eines gerichteten Graphen, wobei die Neuronen als Knoten und die Verbindungen als Kanten dienen. Jeder Knoten (Neuron) besitzt eine beliebige Menge ein- und ausgehender Kanten (Verbindungen) und einen *Aktivierungswert*, welcher im Allgemeinen ein reeller Wert zwischen 0.0 und 1.0 ist. Alle Verbindungen sind mit einer *Gewichtung*, einem beliebig wählbaren reellen Zahlenwert versehen (vgl. [Lip05, S. 51]).
Jedes künstliche neuronale Netz soll in einer vorgegebenen Umwelt mit Hilfe von *Lernalgorithmen* und *Trainingsdurchläufen* durch Verändern der eigenen Verbindungsgewichtungen ein vordefiniertes Verhalten zeigen, welches als die *Funktion* des Netzes bezeichnet wird.
Bei den für eine Implementierung des Modells von Hutchins und Hazlehurst benötigten künstlichen neuronalen Netzen handelt es sich um *Autoassociator Netze*, welche aus drei verschiedenen Schichttypen bestehen: einer *Eingabeschicht*, beliebig vieler *verborgener Schichten* und einer *Ausgabeschicht*. Die Funktion dieses Netzes ist es, ein von der Umwelt vorgegebenes Muster (in Form eines Vektors reeller Zahlen) nach Einlesen in der Eingabeschicht in der Ausgabeschicht exakt wiederzugeben (vgl. [HH95, S. 162]). Abbildung 7 zeigt den Aufbau eines solchen Autoassociator Netzes, für das gelten muss:

$$x_1 = o_1, x_2 = o_2, \ldots, x_i = o_i .$$

[6] Die Neuroinformatik, als Teil der Informatik, nimmt es sich als Ziel, die führenden Prinzipien der Lösung von Problemen durch Gehirne auf Computersysteme zu übertragen (vgl. [Rus90, S. 3]).

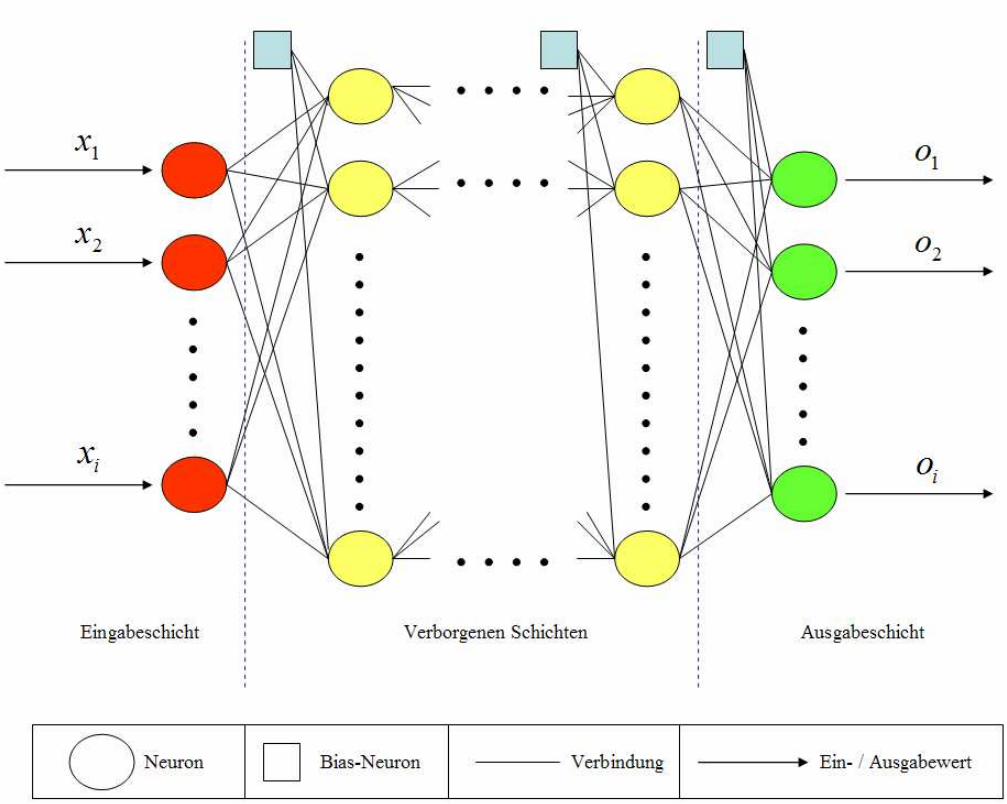


Abb. 7 Aufbau neuronales Netz
Quelle: Neuzeichnung in Anlehnung an [Lip05, S. 51, Abb. 2.8]

Hutchins und Hazlehurst nutzen die Eigenschaften von *Feedforward*[7] Netzen, um Aktivierungen innerhalb des neuronalen Netzes zu propagieren. Die Ausgangslage ist dabei ein untrainiertes Netz, dessen Verbindungsgewichtungen alle mit einem zufällig gewählten reellen Wert (idealerweise zwischen -0.5 und +0.5) belegt wurden.
Die verborgenen Schichten und die Ausgabeschicht beinhalten zusätzlich ein Bias-Neuron. Dieses besondere Neuron besitzt keine eingehenden Verbindungen und gibt als Aktivierungswert den Wert 1.0 zurück [siehe LIP05]. Ihre Existenz ist für den Lernprozess entscheidend, ohne sie kann ein neuronales Netz ein Eingabemuster, welches nur aus Nullwerten besteht, nicht sinnvoll verarbeiten.

[7] Feedforward-Nezte ist der Begriff für künstliche neuronale Netze, bei denen kein Pfad, der von einem gegebenen Neuron direkt oder über zwischengeschaltete Neuronen wieder zu diesem Neuron zurückführt, existiert. Die mathematische Topologie eines solchen Netzes ist die eines azyklischen Graphen (siehe [Lip05, S. 53]).

2.3.2 Aktivierungspropagierung durch Feedforward Netze

Feedforward *Netze* bieten eindeutige Funktionen, die Aktivierungswerte einzelner Neuronen von der Eingabeschicht zur Ausgabeschicht weiterzuleiten; hierbei spielen die Neuronen als Hauptprozesseinheit eine wesentliche Rolle.
Ein künstliches Neuron mit dem Index *j* wird durch 5 Eigenschaften charakterisiert (vgl. [Lip05, S. 47]:

- *Eingabevektor* $\vec{x}$
- *Gewichtungsvektor* $\vec{w}$
- *Netzeingabefunktion* net_j
- *Aktivierungsfunktion* φ_j
- *Ausgabefunktion* o_j

Jeder dieser 5 Eigenschaften ist abhängig von der Schicht, in der sich das entsprechende Neuron befindet. Abbildung 8 zeigt den Aufbau eines einzelnen Neurons.

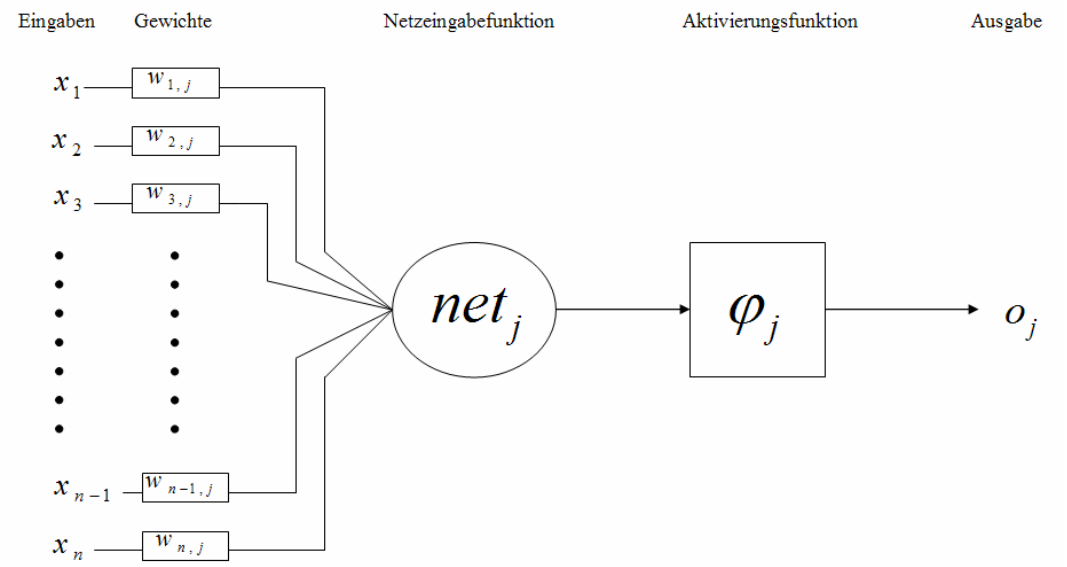


Abb. 8 Künstliches Neuron mit Index j
Quelle: Neuzeichnung in Anlehnung an [Lip05, S. 46, Abb. 2.2]

Der Kern eines Neurons besteht aus der *Netzeingabefunktion*, welche die aufsummierten Produkte der einzelnen Eingaben und den dazugehörigen Gewichten zurückgibt. Sie ist für alle Neuronen mit dem Index j, außer für die der Eingabeschicht definiert als:

$$net_j = \sum_{i=1}^{n} w_{ij} x_i + \beta_j$$

wobei w_{ij} die Gewichtung der Verbindung von Neuron *i* zu Neuron *j*, β_j der Aktivierungswert des Bias-Neurons und x_i die Ausgabe des Neurons *j* ist.
Die Netzeingabe verringert sich bei negativen Gewichtungen zu Vorgängerneuronen, bei positiven wird sie im Gegenzug erhöht.
Die Netzeingabe für Neuronen der Eingabeschicht beschränkt sich auf die Übernahme des Eingabevektors, der in diesem Fall nur aus einem von der Umwelt vorgegebenen reellen Wert besteht.
Die *Ausgabe* eines Neurons besteht in den meisten Fällen nicht aus einer einfachen linearen Weitergabe des Netzeingabewertes an eventuell folgende Neuronen. *Aktivierungsfunktionen* schränken die Ausgabe auf einen bestimmten Wertebereich ein. Alle Neuronen der verborgenen Schichten nutzen die sigmoide Funktion

$$\varphi_j = \frac{1}{1 + e^{\frac{-net_j}{T}}} .$$

Neuronen der Eingabe- bzw. Ausgabeschicht nutzen die Identitätsfunktion als Aktivierungsfunktion. Abbildung 9 zeigt den Verlauf der sigmoiden Funktion mit Streckfaktor $T = 1$.

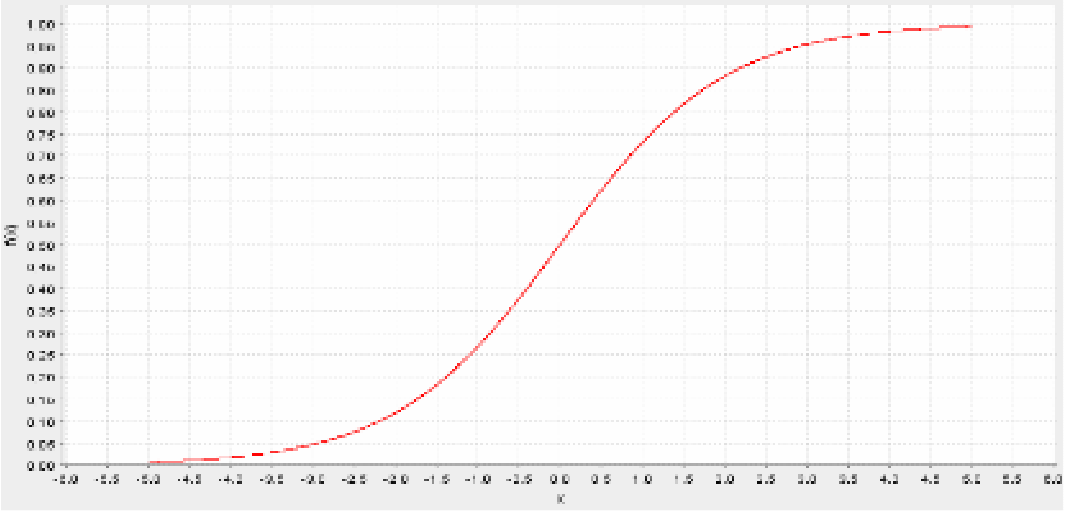


Abb. 9 Sigmoide Funktion mit T=1

Für die Ausgabefunktion aller Neuronen ergibt sich somit:

$$o_j = \varphi_j(net_j)$$

Der iterative Algorithmus zur Berechnung aller Aktivierungswerte und Ausgaben, häufig unter dem Namen „*Forward-Pass*“ zu finden (siehe [Lip05, S. 89]), innerhalb eines neuronalen Netzes, wie in Abbildung 2 aufgebaut, kann halbformal zusammengefasst werden (es sei S die Anzahl der Schichten und #S die Anzahl der Neuronen innerhalb einer Schicht, H ist die aktuelle Schicht):

1. H = 1: Für alle Neuronen j = 1 bis #H

$$net_j = x_j$$
$$\varphi_j = net_j$$
$$o_j = \varphi_j(net_j)$$

2. 1 < H < S: Für alle Neuronen j = 1 bis #H

$$net_j = \sum_{i=1}^{\#(H-1)} w_{ij} o_i + \beta_j$$
$$\varphi_j = \frac{1}{1 + e^{\frac{-net_j}{T}}}$$
$$o_j = \varphi_j(net_j)$$

3. H=S: Für alle Neuronen j = 1 bis #H

$$net_j = \sum_{i=1}^{\#(H-1)} w_{ij} o_i + \beta_j$$
$$\varphi_j = net_j$$
$$o_j = \varphi_j(net_j)$$

2.3.3 Gewichtungsmodifikation durch die generalisierte Delta-Regel

Im folgenden Unterkapitel wird die *Gewichtungsmodifikation* und die damit verbundene *generalisierte Delta-Regel*[8] in Anlehnung an Rumelhart, Williams und Hinton (vgl. [Rum86, S. 322-327]) erläutert.
Das Problem des *Lernens* innerhalb eines neuronalen Netzes besteht hauptsächlich aus dem Finden einer geeigneten Verteilung der Verbindungsgewichtungen, sodass die Funktion des Netzes nach Berechnung aller Aktivierungen der Ausgabeschicht erfüllt wird.
Einem untrainierten neuronalen Netz wird ein Trainingsmuster als Eingabe gegeben. Nach Berechnung aller Aktivierungen werden die Werte der Ausgabeschicht mit den gewünschten Werten des Zielmusters verglichen. Stimmen diese im Sinne der Funktion des neuronalen Netzes überein, werden keine Veränderungen vorgenommen. Falls sich die Ausgaben von dem Zielmuster unterscheiden, müssen Verbindungsgewichtungen innerhalb des Netzes geändert werden. Um zu ermitteln, welche Gewichtungen für fehlerhafte Aktivierungen der Ausgabeschicht verantwortlich sind, wird ein *Fehlermaß* E_p für die Leistungsfähigkeit eines neuronalen Netzes mit Eingabemuster *p* definiert:

[8] Im Gegensatz zur einfachen Delta-Regel, welche auf neuronale Netze ohne verborgene Schichten anzuwenden ist, erweitert die generalisierte Delta-Regel den Anwendungsbereich auf neuronale Netze mit beliebig vielen verborgenen Schichten (vgl. [Rum86, S. 324]).

$$E_p = \frac{1}{2}\sum_j (t_{pj} - o_{pj})^2 ,$$

wobei *j* die Anzahl der Ausgabeneuronen, $t_{j,p}$ der Wert des Zielmusters an der Stelle *j* und $o_{j,p}$ die Ausgabe des j-ten Neurons repräsentiert.
Für die Funktionsfähigkeit *E* des gesamten Netzes werden die Fehlermaße für jedes Trainingsmuster aufsummiert und es gilt:

$$E = \sum_p E_p .$$

Die Modifikation der Verbindungsgewichte innerhalb eines neuronalen Netzes mit beliebig vielen verborgenen Schichten erfolgt nach dem B*ackpropagation of Error*[9] Verfahren, welches auf der generalisierten Delta-Regel

$$\Delta_p w_{ji} = \eta \delta_{pj} o_{pi}$$

basiert, die rückwärtsgerichtet auf alle Verbindungsgewichtungen zwischen den einzelnen Schichten, beginnend bei der Ausgabeschicht, angewandt wird. Innerhalb der Regel gibt *p* an, welches Trainingsmuster gerade an die Eingabeschicht angelegt wird, w_{ji} ist die Verbindungsgewichtung zwischen dem *j*-ten Neuron der Schicht *H* und dem *i*-ten Neuron der Vorgängerschicht *H-1*. δ_{pj} ist das *Fehlersignal* des *j*-ten Neurons der Schicht *H* und muss zuvor berechnet sein, wobei sich die Berechnung der Fehlersignale der Ausgabeneuronen von denen der anderen unterscheidet. o_{pi} ist die Ausgabe des *i*-ten Neurons der Schicht *H-1*. η ist die *Lernrate* des neuronalen Netzes.
Eine Veränderung der Verbindungsgewichtungen nach der oben genannten generalisierten Delta-Regel minimiert das Fehlermaß E_p, da die Ableitung von E_p hinsichtlich der Gewichtungen zur vorgegeben Delta-Regel mit negativem Proportionalitätsfaktor proportional ist. Diese negative Proportionalität gleicht dem größtmöglichen Gradientenabstieg[10] auf einer Oberfläche, deren Höhe dem Fehlermaß E_p entspricht.
Es muss also gezeigt werden, dass

$$\Delta_p w_{ij} \propto -\frac{\delta E_p}{\delta w_{ji}}$$

Wie in Abschnitt 2.3.2 bereits erläutert, geben Neuronen der verborgenen Schicht nicht direkt ihre Netzeingabe

$$net_{pj} = \sum_i w_{ji} o_{pi}$$

[9] Dieses Verfahren wurde ab 1970 von mehreren Autoren unabhängig vorgeschlagen, u. a. in der Dissertation von Paul Werbos. Anfang der 80er Jahre rückte es durch Rumelhart, Hinton und Williams [Rum86] wieder in den Fokus (siehe [Lip05, S. 87]).
[10] Das Backpropagation of Error Verfahren beruht auf dem Gradientenabstiegsverfahren. In einem Punkt $\vec{w}$ wird die Tangente der Fehleroberfläche bestimmt und um eine gewisse Länge abgestiegen, wodurch man eine neue Gewichtungsverteilung $\vec{w}$ erhält. Dieses Verfahren wird so lange wiederholt, bis ein lokales Minimum der Fehleroberfläche erreicht ist (siehe [Lip05, S. 95]).

mit Hilfe der linearen Identitätsfunktion an nachfolgende Neuronen weiter, sondern wenden die semilineare, differenzierbare, steigende sigmoide Funktion φ_j als Aktivierungsfunktion auf die Netzeingabe an. Dieses hat für die Ausgabe dieser Neuronen zur Folge:

$$o_{pj} = \varphi_j(net_{pj}) .$$

Die Ableitung des Fehlermaßes E_p hinsichtlich der Verbindungsgewichtung w_{ji} wird mit Hilfe der Kettenregel in ein Produkt

$$\frac{\delta E_p}{\delta w_{ji}} = \frac{\delta E_p}{\delta net_{pj}} \frac{\delta net_{pj}}{\delta w_{ji}}$$

umgeformt. Der erste Faktor reflektiert die Wirkung der Netzeingabe net_{pj} auf das Fehlermaß E_p, der zweite Faktor den Einfluss einer geänderten Verbindungsgewichtung auf die Netzeingabe net_{pj}.
Aus dem zweiten Faktor wird die Ableitung

$$\frac{\delta net_{pj}}{\delta w_{ji}} = o_{pi}$$

gebildet.
Als Fehlersignal δ_{pj} des *j*-ten Neurons der Schicht *H* wird nun festgelegt:

$$\delta_{pj} = -\frac{\delta E_p}{\delta net_{pj}} .$$

Nun ergibt sich für die Ableitung des Fehlermaßes E_p durch Substitution der beiden Faktoren die äquivalente Form

$$-\frac{\delta E_p}{\delta w_{ji}} = \delta_{pj} o_{pi} ,$$

welche die Proportionalität zwischen der Ableitung und der generalisierten Delta-Regel zeigt.
Bevor die generalisierte Delta-Regel zur Gewichtungsmodifikation genutzt werden kann, müssen zunächst für alle Neuronen des neuronalen Netzes, bis auf die der Eingabeschicht, Fehlersignale rekursiv rückwärtsgerichtet berechnet werden. Es unterscheidet sich dabei die Berechnung der Fehlersignale für Neuronen der Ausgabeschicht von den Neuronen der verborgenen Schichten. Das Fehlersignal δ_{pj} wird durch Anwendung der Kettenregel wieder in ein Produkt zerlegt. Es gilt:

$$\delta_{pj} = -\frac{\delta E_p}{\delta net_{pj}} = -\frac{\delta E_p}{\delta o_{pj}} \frac{\delta o_{pj}}{\delta net_{pj}} .$$

Der erste Faktor gibt den Fehler als Funktion der Ausgabe eines Neurons wieder, wohingegen der zweite Faktor die Ausgabe eines Neurons als Funktion seiner Netzeingabe sieht.
Der zweite Faktor kann mit Hilfe der Ableitung der Aktivierungsfunktion φ notiert werden als:

$$\frac{\delta o_{pj}}{\delta net_{pj}} = \varphi'(net_{pj})$$

Für die Ableitung des ersten Faktors werden zwei Fälle unterschieden. Für alle Neuronen der Ausgabeschicht gilt:

$$\frac{\delta E_p}{\delta o_{pj}} = -(t_{pj} - o_{pj})$$

Somit ergibt sich für das Fehlersignal aller Neuronen der Ausgabeschicht:
$\delta_{pj} = (t_{pj} - o_{pj})\varphi'(net_{pj})$
Für die Neuronen aller anderen Schichten errechnet sich ihr Fehlersignal aus den Fehlersignalen der *k* nachfolgenden Neuronen, dies bedeutet für die Herleitung des ersten Faktors:

$$\frac{\delta E_p}{\delta o_{pj}} = \sum_k \frac{\delta E_p}{\delta net_{pk}} \frac{\delta net_{pk}}{\delta o_{pj}} = \sum_k \frac{\delta E_p}{\delta net_{pk}} w_{kj} = -\sum_k \delta_{pk} w_{kj}$$

In Folge dessen ergibt sich für diese Neuronen das Fehlersignal

$$\delta_{pj} = \varphi'(net_{pj}) \sum_k \delta_{pk} w_{kj}$$

2.3.4 Lernalgorithmus

Das folgende Unterkapitel orientiert sich weitestgehend an den Ausführungen von Rumelhart, Williams und Hinton (vgl. [Rum86, S. 327 – 329]).
Die Ausgangslage, um ein neuronales Netz im Sinne seiner Funktion zu trainieren, ist eine Menge *P* an Trainingsmustern, die dem Netz präsentiert werden.
Der Backpropagation of Error *Lernalgorithmus*, auch „Backward-Pass“ genannt (siehe [Lip05, S. 91]), und die damit verbundenen Gewichtungsmodifikationen, kann in *3 Phasen* unterteilt werden. In der ersten Phase wird ein zufällig ausgewähltes Trainingsmuster *p* aus der Menge *P* dem Netz präsentiert, d. h. die Neuronen der Eingabeschicht übernehmen die Werte des Trainingsmusters. Diese Werte werden dann im gesamten Netz, wie in Abschnitt 2.3.2 beschrieben, bis zur Ausgabeschicht propagiert. Stimmen die Aktivierungen der Ausgabeneuronen nicht mit denen des gewünschten Zielmusters überein, wird die 2. Phase begonnen, in der die Fehlersignale für jedes Neuron rekursiv und rückwärtsgerichtet berechnet werden. Im ersten Schritt dieser Phase werden zunächst die Fehlersignale δ_{pj} für jedes Neuron *j* der Ausgabeschicht nach folgender Regel berechnet:

$$\delta_{pj} = (t_{pj} - o_{pj})\varphi'(net_{pj})$$

Für jedes Ausgabeneuron wird das Produkt aus Differenz von gewünschter zu tatsächlicher Ausgabe und der Ableitung seiner Aktivierungsfunktion als Fehlersignal festgelegt. Die Aktivierungsfunktion eines Ausgabeneurons kann sowohl die Identitätsfunktion, deren Ableitung 1 ist, als auch die sigmoide Funktion, deren Ableitung $\varphi' = \varphi(1-\varphi)$ ist, sein. Sind alle Fehlersignale der Neuronen der Ausgabeschicht berechnet, werden die Fehlersignale der Neuronen der verborgenen Vorgängerschichten nach folgender Regel berechnet:

$$\delta_{pj} = \varphi'(net_{pj}) \sum_k \delta_{pk} w_{kj}$$

Um das Fehlersignal des *j*-ten Neurons einer verborgenen Schicht zu berechnen, müssen zunächst alle Fehlersignale δ_{pk} aller mit Neuron *j* verbundenen Neuronen k, multipliziert mit der entsprechenden Verbindungsgewichtung, aufsummiert werden. Neuron *j* erhält das Produkt aus dieser Summe und der Ableitung seiner Aktivierungsfunktion, die Ableitung der sigmoiden Funktion, als Fehlersignal δ_{pj}.
Für die Neuronen der Eingabeschicht wird keinerlei Fehlersignal berechnet.
Sind alle Fehlersignale berechnet, beginnt Phase 3, die Gewichtungsmodifikation nach der generalisierten Delta-Regel

$$\Delta_p w_{ji} = \eta \delta_{pj} o_{pi}$$

Ausgehend von den Verbindungen aller Neuronen *j* der Ausgabeschicht zu allen Neuronen *i* der letzten verborgenen Schicht werden alle Gewichtungen um $\Delta_p w_{ji}$ geändert. Die Lernrate η sollte dabei als eine möglichst große reelle Zahl zwischen 0.0 und 1.0 gewählt werden, um ein schnelles Lernen des Netzes zu gewährleisten. Es folgt somit für die neuen Verbindungsgewichtungen:

$$w_{ji}(t+1) = w_{ji}(t) + \Delta_p w_{ji}(t)$$

Wenn diese Gewichtungsmodifikation rückwärtsgerichtet auf alle Verbindungen einschließlich der Verbindungen zu den Neuronen der Eingabeschicht durchgeführt wurde, gilt die 3. Phase und ein Durchlauf des Algorithmus als beendet.
Dieser Lernalgorithmus wird nun für alle Trainingsmuster einmal durchgeführt.

2.3.5 Momentum-Version

Das Ziel der Momentum-Version[11], einer Modifikation des Backpropagation of Error Verfahrens, ist es, das Gradientenabstiegsverfahren so zu beeinflussen, dass sich auf flachen Niveaus der Fehleroberfläche die *Schrittweite* η des Abstiegs erhöht bzw. in den Tälern der Oberfläche reduziert. Dieses Verhalten wird dadurch erreicht, dass die in der Vergangenheit erfolgten Veränderungen der Verbindungsgewichtungen einen Einfluss auf die aktuelle Gewichtungsveränderung haben (vgl. [Lip05, S. 109f]). Demnach wird im Simulationsschritt *t* jede Gewichtung $w_{ji}(t)$ des Netzes folgendermaßen modifiziert:

$$w_{ji}(t+1) = w_{ji}(t) + \Delta_p w_{ji}(t)$$

$$\Delta_p w_{ji}(t) = (1-\alpha)\, \eta \delta_{pj} o_{pi} + \alpha\, \Delta_p w_{ji}(t-1)$$

Dabei ist η die Lernrate und $\alpha \in [0,1[$ das *Momentum*. Der Term $\Delta_p w_{ji}(t-1)$ beinhaltet die zuletzt durchgeführte Gewichtungsmodifikation und wird mit dem

[11] Diese Version des Backpropagation of Error Verfahrens geht auf Hinton und Williams zurück und wurde erstmals in [Rum86, S. 327] erwähnt. „Konjugierter Gradientenabstieg“ wird als synonyme Bezeichnung für dieses Verfahren verwendet (vgl. [Lip05, S. 109]).

Faktor α in die aktuell durchgeführte Modifikation eingerechnet. Setzt man den *Trägheitsmoment* $\alpha = 0$, so erfolgt die Gewichtungsmodifikation wieder gemäß der in Abschnitt 2.3.4 erläuterten generalisierten Delta-Regel. Idealerweise wird für das Momentum ein Wert von 0,9 gewählt, um den Gradientenabstieg auf den meist flachen Fehleroberflächen zu beschleunigen. Auf sehr stark gekrümmten Oberflächen ist ein zu groß gewähltes Momentum für das Backpropagation of Error Verfahren eher hinderlich.

2.4 Modell Hutchins/Hazlehurst

Ziel dieses Unterkapitels ist es, das von Edwin Hutchins und Brian Hazlehurst erstellte Modell zur Emergenz eines Lexikons innerhalb einer Gemeinschaft von Agenten vorzustellen. Es wird hierbei sowohl genauer auf die Struktur eines Lexikons als auch auf bestimmte Eigenschaften der mit neuronalen Netzen ausgestatteten Agenten und deren Gemeinschaft eingegangen.

2.4.1 Bedingungen an ein gemeinsam genutztes Lexikon

Die Erläuterungen im folgenden Unterkapitel orientieren sich an den Ausführungen von Hutchins und Hazlehurst zum in der Überschrift genannten Thema (vgl. [HH95, S. 161–165]).
Das Erfinden eines gemeinsam genutzten Lexikons beinhaltet das zentrale Problem einer exakten Darstellung der Zuordnung einer visuellen Szene der Umwelt zu einem Symbol, welches jeder Agent für sich erstellt.
Ausgehend von zwei Agenten einer Gemeinschaft, A und B, und einer Menge von visuellen Szenen einer Welt, nummeriert mit 1,2,3,4,….*m*, kann die Konkatenation des Buchstabens des Agenten mit der Ziffer einer Szene als mögliche Darstellung der oben genannten Zuordnung gesehen werden. So beinhaltet „A*3*“ zum Beispiel das Symbol, welches Agent A zur Darstellung der dritten Szene einer Umwelt nutzt. Soll ein neu entstandenes Lexikon von beiden Agenten A und B genutzt werden, muss es zwei grundlegende Bedingungen erfüllen:

1. Das Wort, welches A für eine spezielle Szene nutzt, muss mit dem von B benutzten Wort übereinstimmen. Allgemein gesehen bedeutet dies:
   A1 = B1, A2 = B2, A3 = B3, ….., A*m* = B*m* .
2. Die Wörter, die ein Agent A und B für eine bestimmte Menge Szenen 1,2,3….*m* erstellt, müssen sich voneinander unterscheiden. Es muss somit gelten: A1 ≠ A2 ≠ A3 ≠ … A*m* und B1 ≠ B2 ≠ B3 ≠ … B*m* .

Ein neu entwickeltes Lexikon kann dementsprechend als Konsens der Einhaltung dieser beiden Bedingungen gesehen werden.

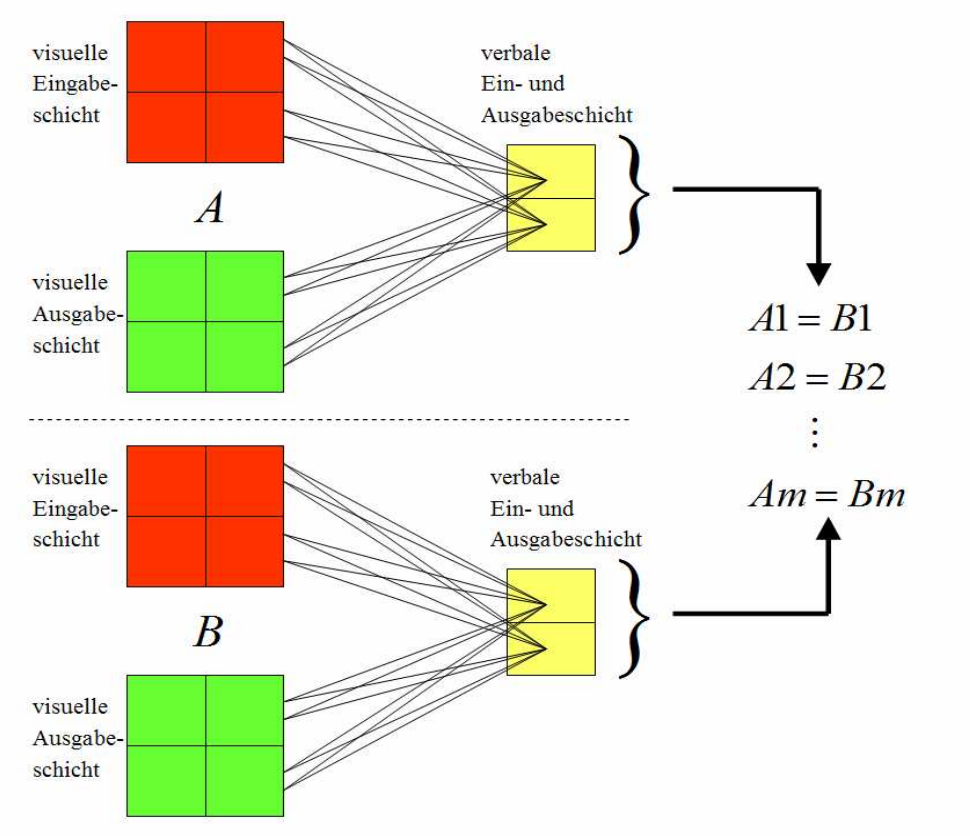


Abb. 10 Schema der Konsensbildung
Quelle: Neuzeichnung [HH95, S. 166, Figure 9.4]

Jedes neuronale Netz aller Agenten ist darauf ausgelegt, diese Bedingungen einzuhalten. Die Netze der Agenten des Modells von Hutchins und Hazlehurst besitzen eine *visuelle* Eingabe- und Ausgabeschicht, d. h. ihre Fähigkeiten beschränken sich auf das Sehen und Abzeichnen eines von der Umwelt vorgegebenen Musters. Die letzte verborgene Schicht repräsentiert das *Wort* oder *Symbol* eines Agenten für die aktuell angelegte visuelle Szene der Umwelt und wird als *verbale* Ein- und Ausgabeschicht definiert. Da der Inhalt dieser Schicht für andere Agenten sichtbar ist, handelt es sich nicht um eine verborgene Schicht.
Das Erfüllen der *ersten Bedingung* besteht darin, dass zwei oder mehrere Agenten das gleiche Wort für eine bestimmte Szene der Umwelt benutzen. Abbildung 10 skizziert dabei die Konsensbildung zweier Agenten *A* und *B*.
Neuronale Netze erfüllen diese Bedingung, indem sie als Gemeinschaft auftreten und nicht wie oftmals nur individuell betrachtet werden. Im Falle der oben genannten zwei Agenten A und B und deren neuronalen Netzen A und B sehen beide Netze das andere jeweils als Lehrer an und versuchen dessen verbale Ausgabe als verbales Zielmuster zu übernehmen. Für diesen Fall wird es nach mehrfachen Durchläufen des Lernalgorithmus zu einem Konsens auf der Wortebene zu einer bestimmten Szene der Umwelt kommen.

Das Erfüllen der *zweiten Bedingung*, kein gleiches Wort für zwei oder mehrere Szenen der Umwelt zu benutzen, wird durch die neuronalen Autoassociator Netze automatisch erfüllt. Rumelhart, Hinton and Williams (vgl.[HH95, S. 165]) haben gezeigt, dass unter bestimmten Umständen vollkommen trainierte Netze die Aktivierungen der Neuronen der verborgenen Schichten die Eingabemuster der Umwelt effizient und überschneidungsfrei verschlüsseln. Werden zum Beispiel an ein Autoassociator Netz mit 4 Neuronen in der Eingabeschicht, 2 Neuronen in der verborgenen Schicht und 4 Neuronen in der Ausgabeschicht 4 orthogonale Eingabemuster angelegt, so sollten die Aktivierungen der Neuronen der verborgenen Schicht zu {(0,0),(0,1),(1,0),(1,1)} konvergieren.

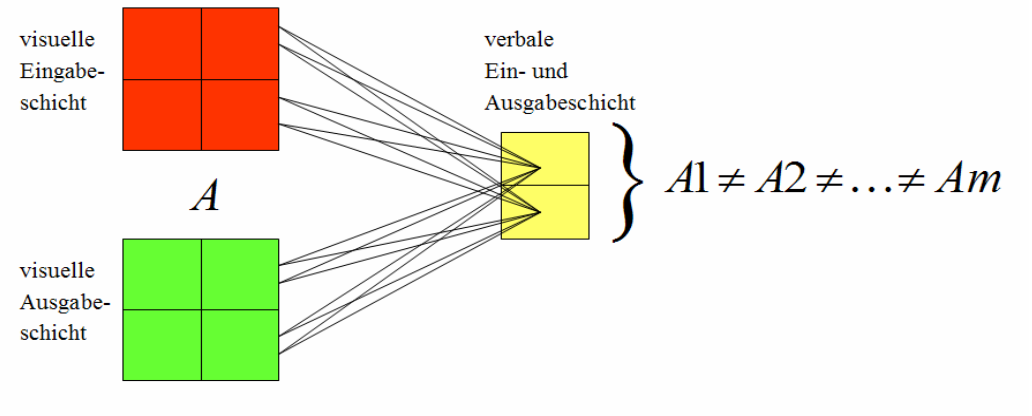


Abb. 11 Schema der Wortdifferenz
Quelle: Neuzeichnung [HH95, S. 165, Figure 9.3]

2.4.2 Die Eigenschaften Avg1 und Avg2

Die Ausführungen in diesem Abschnitt orientieren sich an den Erläuterungen von Hutchins und Hazlehurst [HH95, S. 185f].
Innerhalb einer Gemeinschaft von *n* Agenten, die in einer Welt mit *m* visuellen Szenen leben, kann die Gemeinschaftssprache L_t, welche die Wörter aller Agenten *n* für alle Szenen *m* beinhaltet, in Form der Matrix

$$L_t = \begin{matrix} s_{1,1}(t) & s_{1,2}(t) & \dots & \dots & s_{1,j}(t) \\ s_{2,1}(t) & \ddots & & & \vdots \\ \vdots & & \ddots & & \vdots \\ \vdots & & & \ddots & \vdots \\ s_{i,1}(t) & s_{i,2}(t) & \dots & \dots & s_{i,j}(t) \end{matrix}$$

notiert werden. Die Sequenz der Matrizen $\{L_0, L_1, ..., L_\varphi\}$ stellt die Evolution der Sprache über den Zeitraum t = 0 bis t = φ dar.

$s_{i,j}(t)$ steht dabei für das Wort des Agenten *j* für die visuelle Szene *i* zum Zeitpunkt *t*.

Zu jedem Zeitpunkt *t* kann der Grad der Entwicklung der Sprache L_t durch zwei Eigenschaften ausgedrückt werden:

- Avg1, die durchschnittliche Differenz zwischen allen Wortpaaren eines Agenten für alle Agenten und Szenen;
- Avg2, die durchschnittliche Differenz zwischen allen Wortpaaren der gleichen Szene für alle Agenten und Szenen.

Avg1 bietet demnach ein Maß der *Fähigkeit eines Agenten* Wörter zu kreieren, die zwischen den einzelnen Szenen unterscheiden, während Avg2 ein Maß der *Fähigkeit einer Gemeinschaft* ist, sich auf gleiche Wörter für eine bestimmte Szene zu einigen.
Formal gesehen sind $s_{i,j}(t)$ und $s_{q,p}(t)$ Vektoren mit reellen Werten der Länge γ, wobei *i* und *q* zwei einander verschiedene Elemente der Menge der visuellen Szenen bzw. *j* und *p* zwei unterschiedliche Agenten einer Gemeinschaft sind.

Als Distanzmetrik *d* wird definiert:

$$d(s_{i,j}(t), s_{q,p}(t)) = \sqrt{\frac{\sum_{k=1}^{\gamma} (r_{i,j}^k - r_{q,p}^k)^2}{\gamma}}$$

Daraus folgt für die Eigenschaften Avg1 und Avg2:

$$Avg\ 1(t) = \frac{\sum_{j=1}^{n} \left(\frac{\sum_{(i_1, i_2) \in P_2(m)}^{\frac{m^2 - m}{2}} d\left(s_{i_1, j}(t), s_{i_2 j}(t)\right)}{\frac{m^2 - m}{2}} \right)}{n}$$

$P_2(m)$ ist die Menge aller Paare ganzzahliger, positiver Zahlen von 1 bis *m*, $\frac{m^2 - m}{2}$ gibt die Länge dieser Menge $P_2(m)$ an.

Auf die gleiche Weise gilt:

$$Avg\ \ 2(t) = \frac{\sum_{i=1}^{m} \left(\frac{\sum_{(j_1, j_2) \in P_2(n)}^{\frac{n^2 - n}{2}} d\left(s_{i,j_1}(t), s_{i,j_2}(t)\right)}{\frac{n^2 - n}{2}} \right)}{m}$$

2.4.3 Implementierung des Modells

Die folgende Erläuterung einer Implementierung des in den vorherigen Abschnitten beschrieben Modells gibt die Vorstellungen von Hutchins und Hazlehurst wieder [HH95, S. 166f].
Die Simulation erfolgt über *Interaktionen* innerhalb einer *Simulationsumgebung,* welche aus einer *Umwelt* mit *m* visuellen Szenen und einer *Gemeinschaft* aus n Agenten besteht. Genau eine Interaktion ist mit einem Zeitschritt innerhalb der Simulation gleichzusetzen und beinhaltet, dass zwei ausgewählten Agenten der Gemeinschaft, einem *Sprecher A* und einem *Zuhörer B,* eine der *m* visuellen Szenen präsentiert wird. Die Auswahl der Szene und der beiden Individuen erfolgt über das *Interaktionsprotokoll* und wird typischerweise durch eine einfache, *zufällige* Auswahl implementiert. Sprecher *A* reagiert auf die Präsentation einer Szene *m*, indem er über sein Feedforward Netz die Aktivierungen innerhalb seiner verbalen Ausgabeschicht berechnet. Hörer *B* reagiert in gleicher Weise, nutzt aber das Wort von *A* als Ziel, um seine eigenen Aktivierungen innerhalb der verbalen Ausgabeschicht zu korrigieren. Des Weiteren möchte *B* die aktuelle Szene erlernen und nutzt sein erzeugtes Wort, neben dem Abgleich zum Wort von A, dazu, durch Propagierung der Aktivierungen auf der visuellen Schicht eine Ausgabe zu erzeugen. Dieses Verhalten von B hat zwei wesentliche Effekte zur Folge, zum einen gleicht er sein Wort für die aktuell präsentierte Szene an das Wort von A an, zum anderen versucht er mit seinem Wort die aktuelle Szene in seiner Ausgabeschicht korrekt wiederzugeben.
Über die Zeit wird durch die zufällige Auswahl der interagierenden Agenten und der präsentierten Szenen garantiert, dass jedes Individuum einer Gemeinschaft die Chance bekommt, sowohl in sprechender als auch hörender Rolle mit allen anderen Individuen der Gemeinschaft über alle visuellen Szenen der Umwelt zu kommunizieren.
Hutchins und Hazlehurst untersuchen den Ablauf der Simulation anhand zweier verschiedener Simulationen, einer ersten Simulation mit einer komplexen Netzstruktur innerhalb der Agenten und einer großen Menge visueller Szenen, die das Verhalten innerhalb der Simulation eher qualitativ veranschaulichen soll, und

einer zweiten Simulation mit einer einfachen Menge visueller Szenen (vier orthogonale Vektoren {(1,0,0,0), (0,1,0,0), (0,0,1,0), (0,0,0,1)}) und einer einfacheren Netzstruktur innerhalb der Agenten, welche eher einer analytischen Untersuchung der Simulation dient (siehe Kapitel 5).

3. Anforderungsdefinition, Architektur und Entwurf

Das dritte Kapitel beschreibt die Anforderungen, die benötigten Architekturen und eine Entwurfsvorstellung für die spätere Implementierung des Modells von Hutchins und Hazlehurst. Insbesondere in diesem Kapitel wird eine iterative Vorgehensweise deutlich: Ausgehend von der Modellbeschreibung aus Unterkapitel 2.4 leiten sich mehre Anforderungen ab, aus denen ein Entwurf erstellt wird, der die Einbindung verschiedener Architekturen fordert.

3.1 Anforderungsdefinition

Im folgenden Unterkapitel werden die Anforderungen für eine Implementierung des Modells von Hutchins und Hazlehurst hinsichtlich der Funktionalität aufgelistet. Der größte Teil dieser Anforderungen ergibt sich bereits aus der Beschreibung des Modells (vgl. Kapitel 2) und wird in diesem Unterkapitel im Stile einer formalen Anforderungsdefinition wiedergegeben.
Unter Berücksichtigung erster Entwurfsüberlegungen wird zwischen einer Dreiteilung in funktionale Anforderungen der Simulation, der Benutzerinteraktion und zwischen sonstigen Anforderungen unterschieden.

3.1.1 Simulation

Simulation

- Die Simulation muss agentenbasiert sein.
- Die Simulation muss eine Simulationsumgebung besitzen.
- Die Simulation muss deterministisch ablaufen. Eine festgelegte Aktion bei einem festgelegten Zustand muss immer zu einem identischen Folgezustand führen.
- Die Simulation muss den in Kapitel 2 beschriebenen Fehlerindikator Avg1 optimieren.
- Die Simulation muss den in Kapitel 2 beschriebenen Fehlerindikator Avg2 optimieren.
- Eine Simulation kann aus mehreren Simulationsumgebungen bestehen.

Agent

- Ein Agent muss aus einem neuronalen Netz bestehen.
- Ein Agent muss zwei Eingabeschnittstellen besitzen. Eine muss dabei der Sinneswahrnehmung des Hörens entsprechen, eine der Sinneswahrnehmung des Sehens.
- Ein Agent muss mit anderen Agenten interagieren können.

Simulationsumgebung

- Eine Simulationsumgebung muss eine beliebig große Menge visueller Szenen besitzen.
- Eine Simulationsumgebung muss aus einer beliebig großen Gemeinschaft von Agenten bestehen.

3.1.2 Benutzerinteraktion

Benutzergruppe

- Die Simulation muss von Mitgliedern der Projektgruppe EMIL verwendet werden können.
- Die Simulation soll von Benutzern ohne Programmierkenntnisse verwendet werden können.
- Die Simulation soll von Benutzern ohne Erfahrung im Umgang mit neuronalen Netzen verwendet werden können.

Benutzerschnittstelle

- Die Benutzerschnittstelle muss graphisch umgesetzt werden.
- Die Benutzerschnittstelle muss das Speichern von Simulationen ermöglichen.
- Die Benutzerschnittstelle muss das Laden von Simulationen ermöglichen.
- Die Benutzerschnittstelle muss das Erstellen von Simulationen ermöglichen.
- Die Benutzerschnittstelle muss das Editieren von Simulationen ermöglichen.

- Die Benutzerschnittstelle muss den Fehlerindikator Avg1 in Form eines Graphen ermöglichen.
- Die Benutzerschnittstelle muss den Fehlerindikator Avg2 in Form eines Graphen ermöglichen.
- Die Benutzerschnittstelle muss einfach strukturiert gestaltet werden

3.2 Architektur

Eine Softwarearchitektur beschreibt die grundlegenden Komponenten eines Softwaresystems und ihr Zusammenwirken miteinander. Helmut Balzert[12] nennt sie „eine strukturierte oder hierarchische Anordnung der Systemkomponenten sowie Beschreibung ihrer Beziehungen“ (siehe [Bal01], S. 715).
In diesem Unterkapitel werden die fertigen Softwarepakete beschrieben, auf denen die Implementierung des Tools basiert. Die selbst entworfenen Teile des Programms werden im Unterkapitel 3.3 genauer erläutert.
Abbildung 12 zeigt die für eine Implementierung intendierte Architektur im Schichtenstil. Die unterste Schicht stellt dabei die Applikationsschicht dar, während die oberste die Interaktionsschicht repräsentiert.

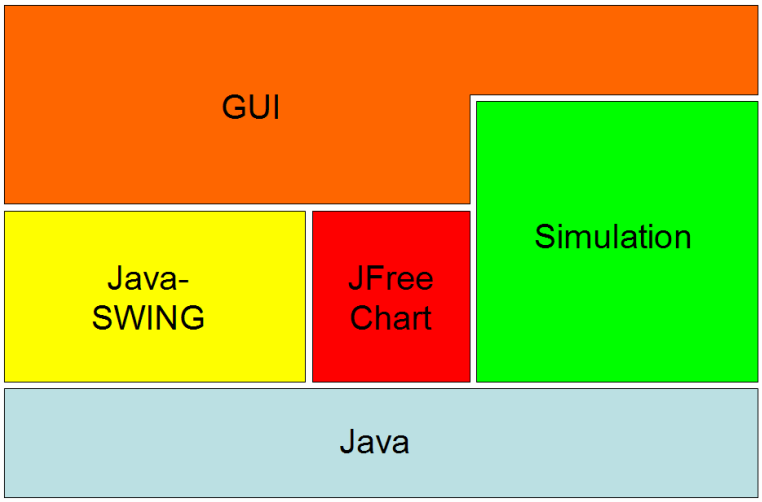


Abb. 12 Architektur

[12] Helmut Balzert ist Inhaber des Lehrstuhls für Software-Technik in der Fakultät für Elektrotechnik und Informationstechnik an der Ruhr-Universität Bochum.

3.2.1 Java

Java[13], eine objektorientierte Hochsprache, wurde als Programmiersprache des zu implementierenden Tools ausgewählt, da es sowohl eine *plattformübergreifende Nutzung* des Programms gewährleistet als sich auch durch eine hohe Verbreitung auszeichnet.

3.2.2 Java-SWING

Um eine plattformübergreifende Nutzung von LexLearn zu garantieren, wurden für die Erstellung einer grafischen Benutzeroberfläche die SWING Grafikbibliotheken der Java Foundation Classes (JFC) ausgewählt. In Anlehnung an Guido Krüger (siehe [Kru00, S. 740-743]) bietet SWING, welches auf dem Abstract Windowing Toolkit (AWT) aufbaut, drei entscheidende Eigenschaften:

Leichtgewichtige Komponenten

Swing-Komponenten nutzen plattformspezifische GUI-Ressourcen nur noch in sehr eingeschränkter Form, d. h. abgesehen von Top-Level-Fenstern, Dialogen und grafischen Primitivoperationen werden alle GUI-Elemente von SWING eigenständig erzeugt. Neben der erheblichen Codevereinfachung und der plattformübergreifenden Vereinheitlichung von Aussehen und Bedienbarkeit, ist das Erstellen von komplexeren Dialogelementen wie Bäumen, Tabellen und Tooltips einer der Hauptvorteile der SWING Bibliotheken. Abbildung 13 gibt eine Übersicht über die in SWING vorhandenen Komponenten.
Ein *Container* nimmt SWING-Komponenten auf und setzt sie mit Hilfe eines Layoutmanagers an die richtige Position.

Austauschbares „Look-and-Feel“

Eine der weiteren wichtigen Eigenschaften, welche in LexLearn jedoch nicht genutzt wird, von SWING ist die Möglichkeit das Aussehen und die Bedienung („Look-and-Feel“) einer Anwendung zur Laufzeit umzuschalten. Mögliche Varianten für ein „Look-and-Feel“ sind beispielsweise Swing, Motif oder Windows.

[13] http://java.sun.com

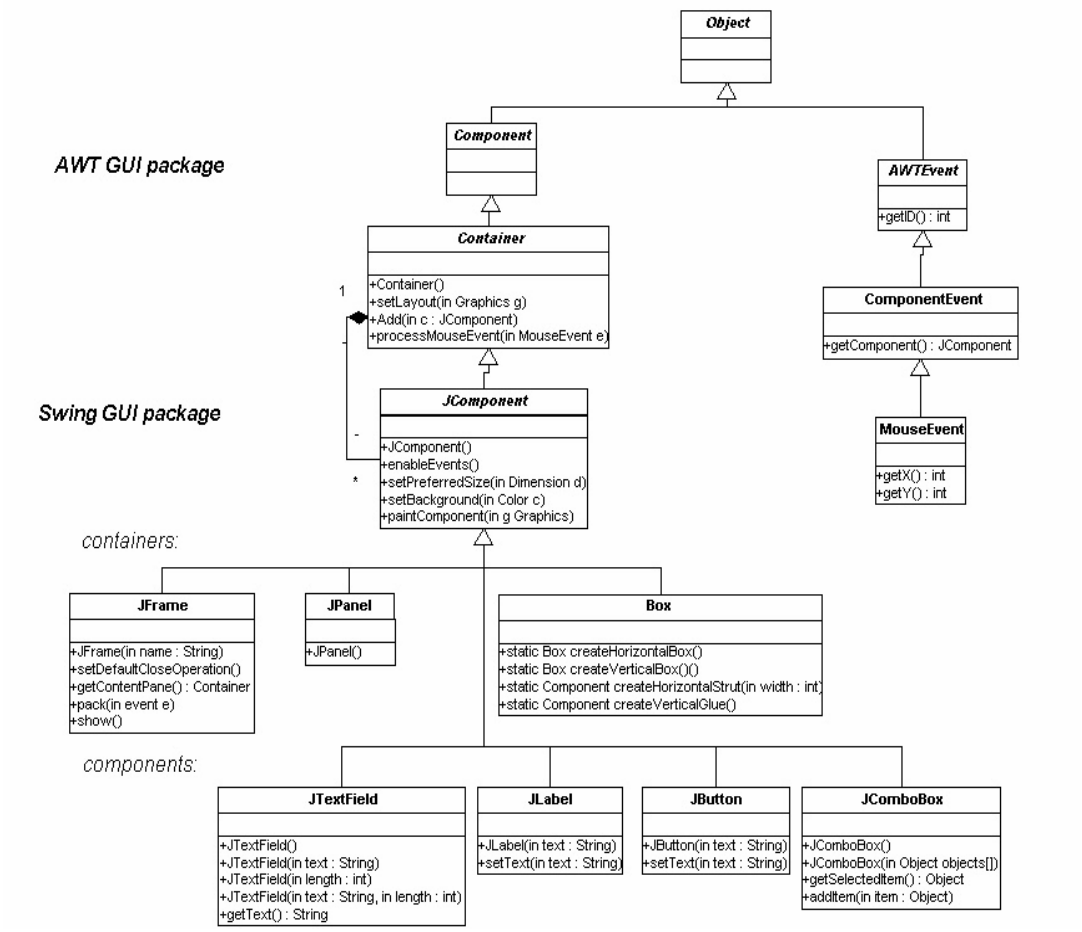


Abb. 13 Modell der Klasse JComponent
Quelle: Parallel Computing Laboratory, Vrije Universiteit Brussel[14]

Model-View-Controller-Prinzip

Unter dem Model-View-Controller-Prinzip versteht man das Konzept, drei verschiedene Bestandteile eines grafischen Elements zu unterscheiden:

- das *Modell*, welches den Zustand und die Daten eines Dialogelements abspeichert,
- der *View*, welcher für die grafische Darstellung der Komponente verantwortlich ist und
- der *Controller*, der Tastatur- und Mausereignisse empfängt und die erforderlichen Veränderungen von Model und View bewirkt.

[14] http://parallel.vub.ac.be/documentation/java/swing/Swing_GUI_Class_model.jpg

Die bei Swing-Dialogelementen genutzte Variante *Model-Delegate-Prinzip* fasst aus Komplexitätsgründen die Funktionalität von View und Controller in einer Benutzerschnittstelle *Delegate* zusammen.

Neben den oben beschriebenen positiven Eigenschaften von Swing besteht ein erheblicher Nachteil: Swing-Anwendungen sind wegen ihrer plattformunabhängigen, selbst erstellenden Komponenten ressourcenhungrig und verlangsamen bei nicht ausreichender Rechnerleistung eine Anwendung. Aufgrund der geringen strukturellen Komplexität der in LexLearn verwendeten grafischen Benutzerschnittstelle und den immer weiter steigenden Rechnerleistungen ist dieser Nachteil in LexLearn aber nicht spürbar.

3.2.3 JFreeChart

Bei JFreeChart[15] handelt es sich um eine frei verfügbare Java Diagramm Bibliothek, die, wie von den Entwicklern beschrieben, folgende besondere Merkmale besitzt:

- eine konsistente und gut dokumentierte Programmierschnittstelle (API), welche eine große Menge von Diagrammtypen unterstützt,
- ein einfaches, flexibles Design, welches einfach zu erweitern ist und auf sowohl server- als auch clientseitige Anwendungen zielt,
- eine große Menge von Ausgabeformaten für die erzeugten Diagramme: neben den gängigen Bilddateiformaten (JPG, PNG) und Vektorgrafikformaten (PDF, EPS, SVG) auch Java SWING Komponenten, welche für eine spätere Implementierung wichtig sind,
- es handelt sich um eine freie „open-source“ Software, welche nach der GNU Lesser General Public Licence (LGPL)[16] vertrieben werden kann.

Für die in Abschnitt 2.4.2 beschriebene Darstellung der Fehlerindikatoren Avg1 und Avg2 ist das Packet **`org.jfree.chart`** der JFreeChart-Bibliothek von besonderem Interesse. Es werden hier Objekte der Klasse **`JFreeChart`**, welche die eigentlichen Diagramme repräsentieren, und Instanzen der Klasse **`ChartPanel`**, Java SWING Komponenten, für die Anzeige eins JFreeChart-Objekts, benötigt[17].
JFreeChart benötigt das JDK 1.3 oder höher.

[15] http://www.jfree.org/jfreechart/
[16] http://www.gnu.org/licenses/lgpl.html
[17] Eine komplette Übersicht der JFreeChart API gibt es unter http://www.jfree.org/jfreechart/api/javadoc/index.html

3.3 Entwurf

Im folgenden Unterkapitel werden die im Rahmen dieser Arbeit selbst entworfenen Teile der Architektur genauer erläutert. Zuerst liegt das Augenmerk auf der Komponente Simulation, die direkt und nur auf Java aufsetzt und in der die eigentlichen Simulationen ablaufen. Der zweite Abschnitt beschreibt den Entwurf der graphischen Benutzeroberfläche, die die Schnittstelle zwischen den Simulationsklassen und dem Benutzer darstellt. Im letzten Abschnitt wird das Zusammenwirken aller Entwurfsteile, als Übergang zur im nächsten Kapitel beschriebenen Implementierung, kurz erläutert.

3.3.1 Simulation

Der Entwurf der Simulation leitet sich nahezu ausschließlich aus dem Modell von Hutchins und Hazlehurst und den Grunddatentypen von Java ab.
Es gibt eine Klasse Simulation, die sowohl als Schnittstelle zur graphischen Benutzeroberfläche als auch zum Halten und Steuern mehrerer Simulationsumgebungen bestimmt ist. Es soll gewährleistet sein, dass ein Speichern oder Laden dieser Klasse samt aller in ihr enthaltenen Attribute ausreicht, um eine Simulation zu sichern bzw. wiederherzustellen.
Innerhalb der Simulation befinden sich mehrere Simulationsumgebungen, wie sie im Modell beschrieben werden. Neben den Methoden der Berechnung der Fehlermaße Avg1 und Avg2 und der zufälligen Auswahl von Agenten und visuellen Szenen befinden sich in ihnen die Menge der Agenten und die Menge der visuellen Szenen.
Die Klasse Agent als Bestandteil einer Umgebung beinhaltet neben den benötigten und in Unterkapitel 2.3 beschriebenen Algorithmen die Bestandteile seines neuronalen Netzes, welches - aufgrund der Beschränkungen der Programmiersprache Java - in zwei Klassen aufgeteilt wird.
So besteht es aus einer Menge *n* von Klassen des Typs Schicht und einer Menge *n-1* von Klassen des Typs Verbindung, die zwei Schichten miteinander verknüpfen.
Eine Schicht besteht folglich aus einer Menge von Neuronen, einer Klasse, die nur aus primitiven Datentypen zum Halten reeller Werte besteht, während eine Verbindung aus einer Menge von Gewichtungen besteht. Die Klasse Verbindung besteht ebenfalls nur aus primitiven Datentypen, sodass ab diesem Punkt keine weitere Abstraktion mehr notwendig erscheint.

3.3.2 Grafische Benutzeroberfläche

Der Entwurf einer graphischen Benutzeroberfläche sieht vor, dass das zu erstellende Werkzeug die in der Anforderungsdefinition aufgelisteten Benutzerinteraktionen ermöglicht. Zudem soll durch eine klare strukturelle Aufteilung der Oberfläche in verschiedene Funktionalitätsbereiche eine

Erweiterungsmöglichkeit um zusätzliche Interaktionen in zukünftigen Versionen garantiert werden.
In erster Instanz teilt sich der Entwurf der Benutzerschnittstelle des Werkzeugs in ein *Hauptfenster* und ein *Analysefenster* auf.

Hauptfenster

Der Entwurf des Hauptfensters ist in Abbildung 14 dargestellt, jede einzelne Komponente bietet dem Benutzer eine Menge von zusammenhängenden Interaktionsmöglichkeiten.

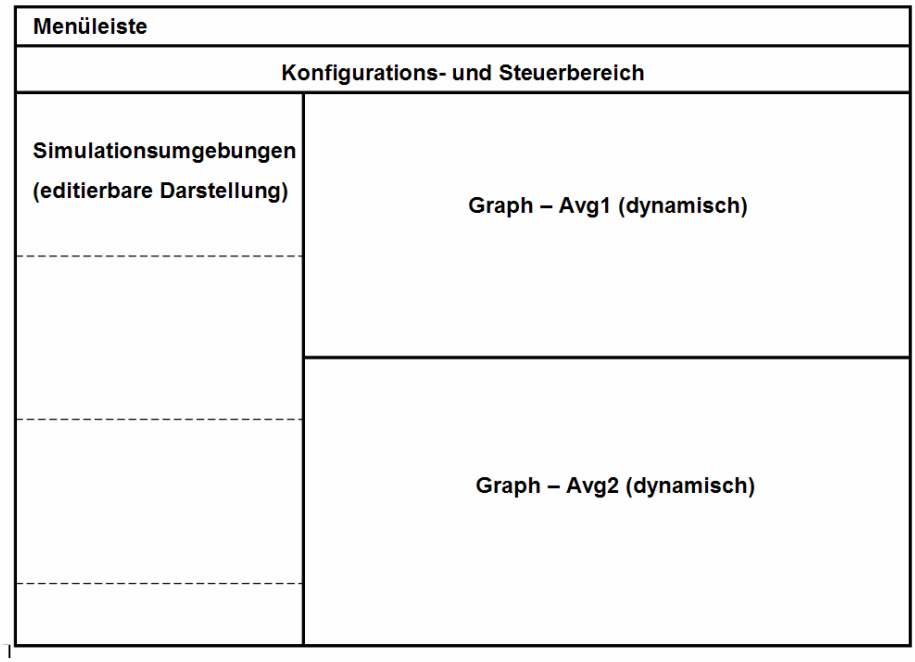


Abb. 14 Entwurf des Hauptfensters

Innerhalb der Menüleiste gibt es ein Dateimenü, welches dem Benutzer die Möglichkeit bietet, eine Simulation zu erstellen, zu laden oder zu speichern, und ein Informationsmenü, durch welches Informationen über die aktuelle Version des Werkzeugs abgerufen werden können. Unterhalb der Menüleiste befindet sich der Konfigurations- und Steuerbereich des Werkzeugs, welcher dem Anwender die Möglichkeit bietet, sowohl konfigurierende Parameter einzugeben als auch einen Simulationsdurchlauf zu steuern. Zu den konfigurierenden Parametern, welche für alle erstellten Simulationsumgebungen gelten, zählen die Eingabe der Anzahl von Simulationsschritten und das Einlesen einer Menge von Eingabemustern. Innerhalb dieses Bereiches wird angezeigt, ob ein Eingabemuster bereits eingelesen wurde und es wird zusätzlich die Möglichkeit geben, ein eingelesenes Muster graphisch zu betrachten. Die Steuerung eines Simulationsdurchlaufs, also

das Starten, Pausieren und Beenden einer Simulation, wird durch drei verschiedene Schaltflächen geregelt.
Des Weiteren sieht der Entwurf der graphischen Benutzeroberfläche vor, die nach dem Erstellen erzeugten Simulationsumgebungen innerhalb eines scrollbaren Rahmens einzeln darzustellen. Die Darstellung umfasst dabei das Auflisten aller Attribute einer Umgebung durch editierbare Textfelder, sodass Änderungen an den Attributen zur Laufzeit vorgenommen werden können. Ein Button am Ende des Rahmens einer jeweiligen Simulationsumgebung dient als Schnittstelle zu dem Analysefenster, welches zur Analyse der entsprechenden Umgebung und deren Agenten dient.
Den größten Anteil der Fläche des Hauptfensters nehmen die Rahmen zur graphischen Darstellung des Avg1 und Avg2 ein. Jede der Darstellungen erfolgt durch ein zweidimensionales Liniendiagramm, dessen Werte der X-Achse den Simulationsfortlauf in Simulationsschritten und Werte der Y-Achse den jeweiligen Fehlerindikator darstellen. Für jede Simulationsumgebung wird innerhalb eines Koordinatensystems ein eigener Graph gezeichnet, welcher sich je nach Simulationsfortschritt dynamisch ändert.

Analysefenster

Der Entwurf des Analysefensters, welches zur Analyse der Ergebnisse einer Simulationsumgebung dient, teilt das Fenster wie in Abbildung 15 in zwei Bereiche auf.

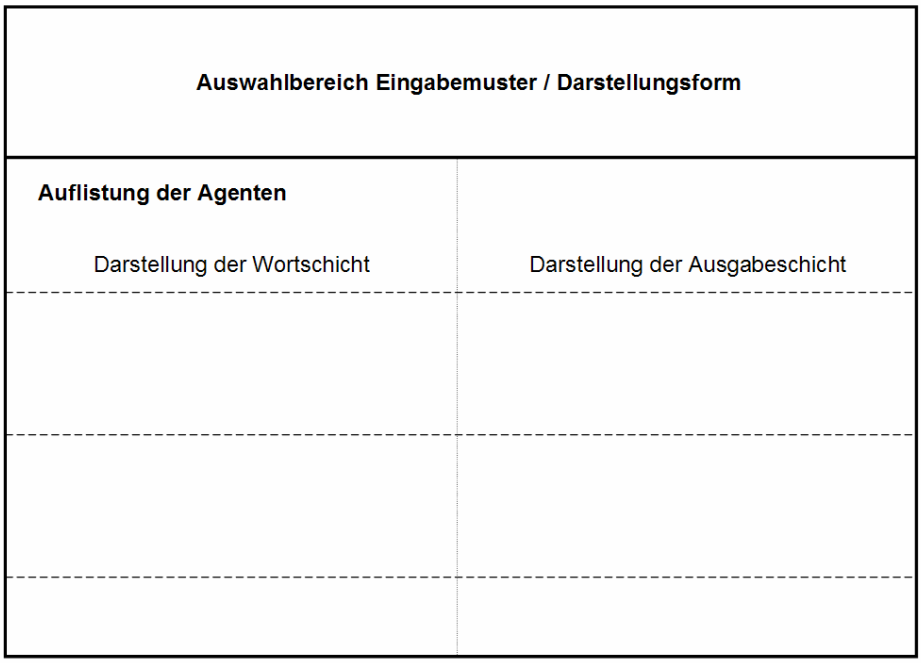


Abb. 15 Entwurf des Analysefensters

Innerhalb des oberen Auswahlbereichs kann der Benutzer zum einen das Eingabemuster, zu welchem er die Wörter und Abzeichnungen der Agenten betrachten möchte, auswählen. Neben dem Index wird eine graphische Darstellung des Eingabemusters als Information gegeben. Zum anderen kann der Benutzer in diesem Bereich zwischen verschiedenen Darstellungsformen der Wörter und Abzeichnungen der Agenten wählen. Zur Darstellung der Wörter kann der Benutzer zwischen

- den exakten, reellen Aktivierungswerten
- den auf zwei Nachkommastellen gerundeten Aktivierungswerten
- einer Abbildung: Aktivierungswert $\mapsto$ Buchstabenfolge

der einzelnen Neuronen der Wortschicht wählen. Zur Darstellung der Deduktion eines Musters kann sich der Nutzer zwischen

- den exakten, reellen Aktivierungswerten
- den auf zwei Nachkommastellen gerundeten Aktivierungswerten
- einer grafischen Darstellung der Aktivierungswerte

der Neuronen der Ausgabeschicht entscheiden.
Die Auflistung der Agenten erfolgt innerhalb eines scrollbaren Rahmens. Je nach ausgewählter Darstellungsform werden für jeden Agenten sein Wort und seine Abzeichnung des ausgewählten Eingabemusters ausgegeben.

3.3.3 Gesamtentwurf

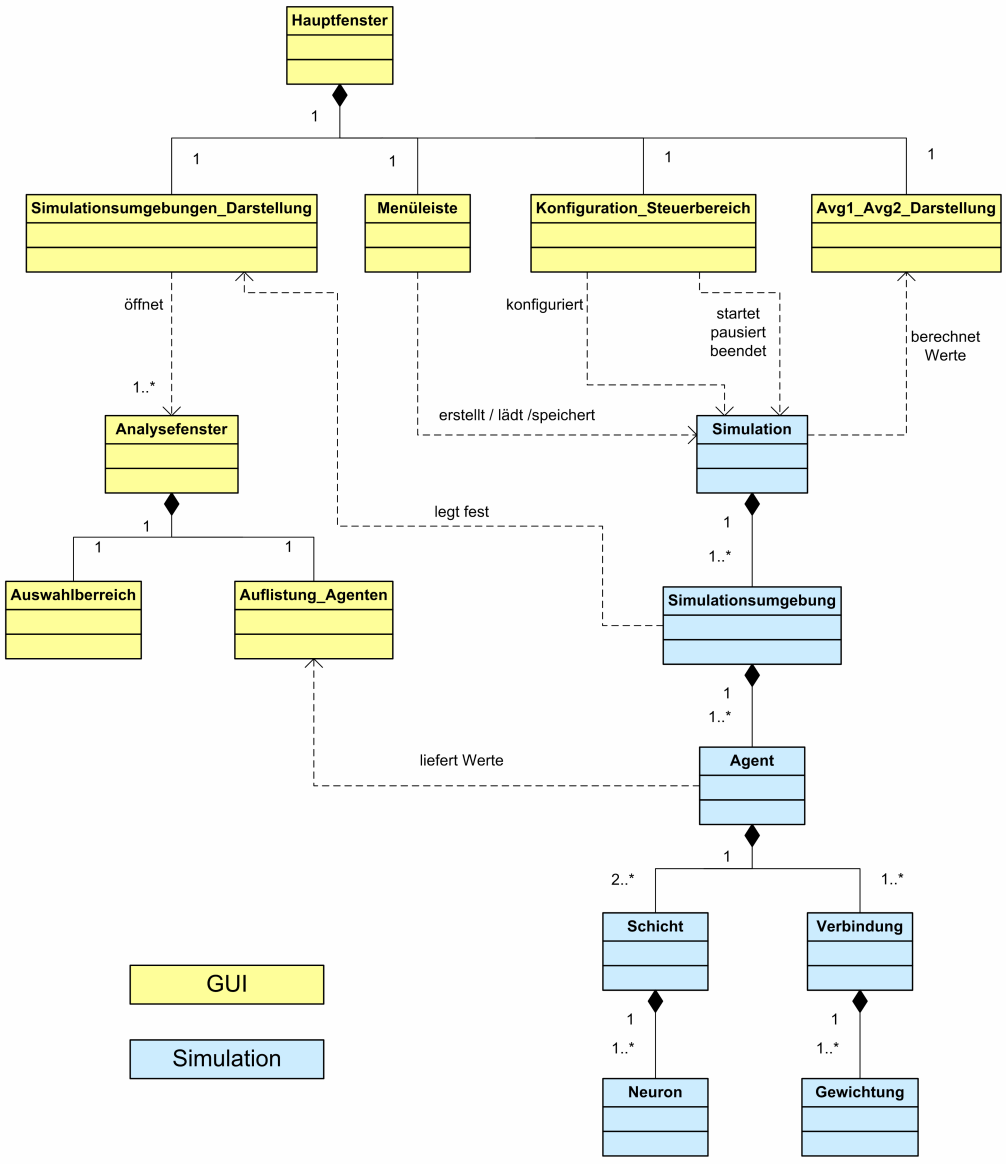


Abb. 16 Gesamtentwurf

Abbildung 16 zeigt das Zusammenwirken der in den Unterkapiteln 3.1 und 3.2 beschriebenen Komponenten des Gesamtentwurfs. Es wird deutlich, dass die Klasse Simulation die zentrale Schnittstelle zwischen der graphischen Benutzeroberfläche und dem Simulationsteil des Tools ist.

4. LexLearn – Die Implementierung

In diesem Kapitel wird die Implementierung des in Kapitel 3 erläuterten Entwurfs unter Berücksicht der angestrebten Architektur beschrieben. Die strikte Trennung zwischen Simulation und graphischer Benutzeroberfläche des Entwurfs teilt auch dieses Kapitel in zwei Unterkapitel. Abbildung 17 zeigt die Umsetzung des in Abbildung 16 dargestellten Gesamtentwurfs in der Anwendung LexLearn.

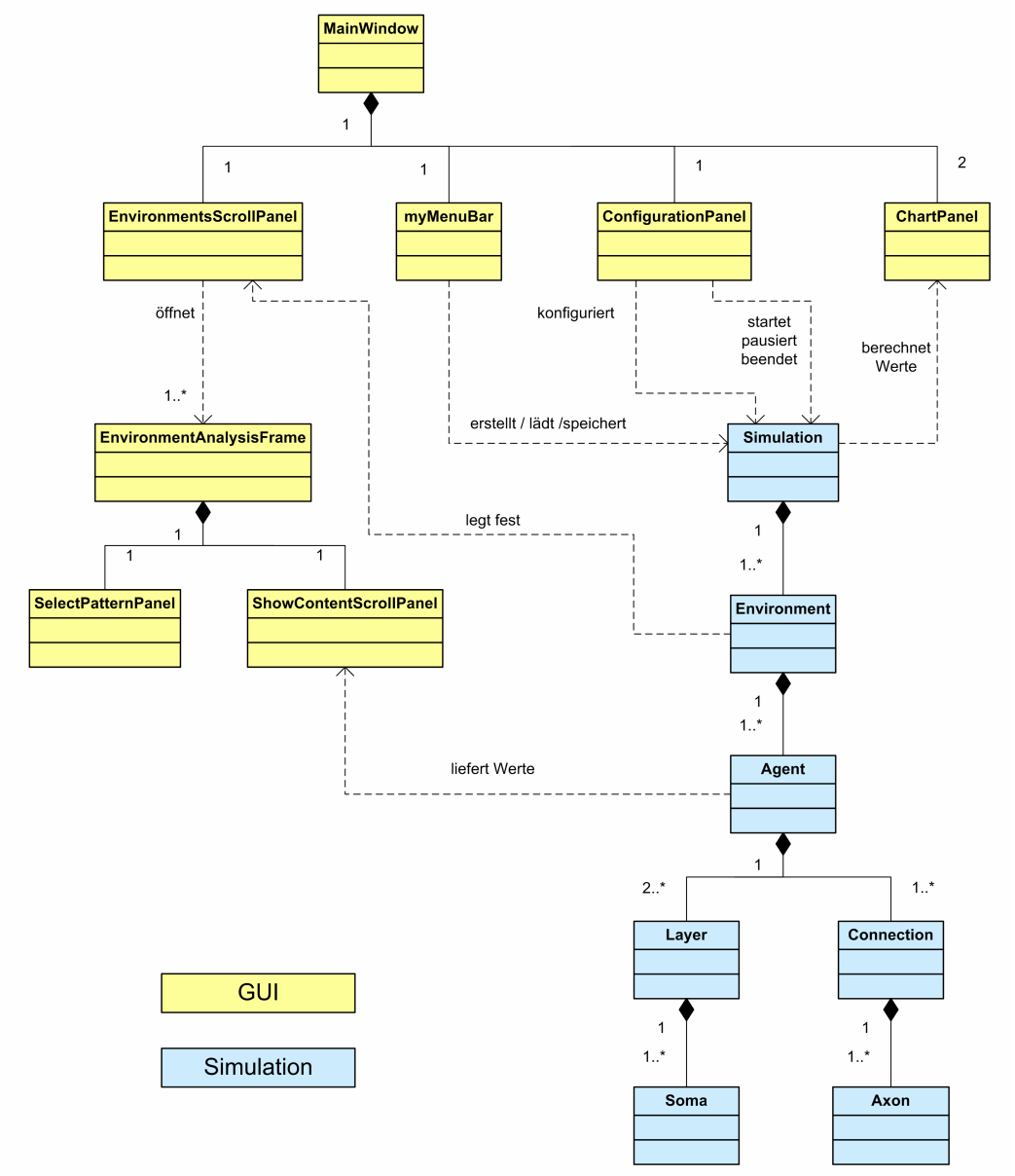


Abb. 17 Klassendiagramm der Implementierung

Abbildung 17 macht deutlich, dass die im Gesamtentwurf verwendeten Klassen durch die in LexLearn tatsächlich genutzten Klassen ersetzt wurden. Die Struktur des Entwurfs wurde in der Realisierung gänzlich übernommen. Auch die in Unterkapitel 3.2 beschriebene Architektur findet sich im Klassendiagramm der gesamten Implementierung wieder. Die Graphische Benutzeroberfläche dient als auslösende, steuernde und konfigurierende Schicht für die Simulation, deren zentrale Schnittstelle durch die Klasse **`Simulation`** repräsentiert wird.
Die Implementierung erfolgte in einer Eclipse Umgebung unter Nutzung eines CVS-Servers an der Universität Koblenz-Landau. Erste Versuche zur Realisierung von neuronalen Netzen bildeten die Basis für eine Implementierung des Modells von Hutchins und Hazlehurst.
Die detaillierte Beschreibung des Aufbaus der Anwendung LexLearn erfolgt sowohl textuell als auch mit Hilfe von UML-Diagrammen und elementaren Auszügen aus dem Quellcode.

4.1 Simulation

Die Implementierung der Simulation unterteilt sich in die drei Hauptbestandteile Agent, Environment und Simulation und stellt somit eine Umsetzung des in Abschnitt 3.3.1 geforderten Entwurfs dar.

4.1.1 Aufbau der Klasse Agent

Die Klasse **`Agent`** beinhaltet sämtliche dem Agenten zugerechneten Attribute und Methoden. Sie besteht nicht nur aus einem neuronalen Netz, dem Hirn der Agenten, sondern ebenfalls aus einer Menge von Methoden, die für die Evolution des Netzes während eines Simulationsdurchlaufs benötigt werden. So gibt es Methoden für eine zufällige initiale Belegung des neuronalen Netzes, den Forward-Pass Algorithmus, die Fehlerberechnung für das Backpropagation of Error Verfahren, eine Methode, die anhand dieser Fehlerberechnung die Kantengewichtungen verändert und eine Reihe kleinerer, meist für die Ein- und Ausgabe benötigter Methoden, die jedoch im Laufe dieses Unterkapitels noch genauer erläutert werden.

Der Aufbau des neuronalen Netzes

Das neuronale Netz wird in der Klasse **`Agent`** in zwei *Arrays* gehalten. Zum einen existiert ein Array aus Objekten des Typs **`Layer`** namens **`layers`** der variablen Länge n aus *Schichten*, welches, beginnend bei der Eingabeschicht an Position *0*, über die verborgenen Schichten von Position *1* bis *n-1*, bis hin zur Ausgabeschicht, die sich an Position *n* des Arrays befindet, beinhaltet.
Die Verbindungen zwischen den einzelnen Schichten werden in einem weiteren Array gehalten. Es besteht aus Instanzen der Klasse **`Connection`** und heißt **`con`**.

Des Weiteren existieren zwei Arrays des Typs **`int`**, in welchen die Längen und Breiten der einzelnen Schichten gelistet werden, folglich heißen sie **`layerlengths`** und **`layerwidths`**.
Eine **`String`** Variable namens **`name`** beinhaltet den Namen des Agenten, die Variablen **`learningRate`** und **`momentum`** des Typs **`double`** halten die Lernrate und das Momentum des Netzes und eine **`int`** Variable mit dem Namen **`wordLocation`** bestimmt die Position der Wortschicht im neuronalen Netz.

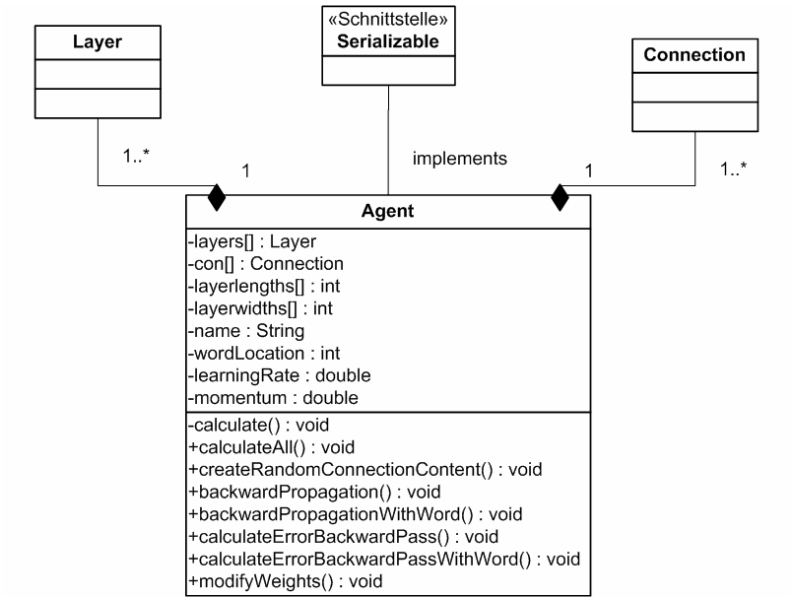


Abb. 18 Klassendiagramm der Klasse Agent

Die Klasse Layer

Eine Schicht des neuronalen Netzes wird durch die Klasse **`Layer`** repräsentiert. Sie besteht aus einem **`String name`** mit dem Namen der Schicht, einem zweidimensionalen Array **`field`** des Typs **`Soma`**, welches die Neuronenschicht darstellt, einem Biasneuron **`bias`** des Typs **`SomaBias`** und einem zweidimensionalen Array des Typs **`double`** der Verbindungen des Biasneurons zu den einzelnen Neuronen der Schicht, welches **`biasCon`** heißt.
Darüber hinaus enthält die Klasse **`Layer`** einige kleinere Methoden, von denen die wichtigsten im Folgenden aufgelistet und kurz beschrieben werden:

`Layer copy()`
Diese Methode fertigt eine exakte Kopie der aktuellen Schicht an und liefert sie als Objekt des Typs **`Layer`** zurück.

`void createRandomBiasConContent()`
In dieser Methode werden die Verbindungsgewichtungen vom Biasneuron zu den Neuronen der Schicht mit reellen Zufallswerten zwischen -0.5 und +0.5 belegt.

`double[][] getValues()`
`double[][] getActivations()`
`double[][] getErrors()`
Diese drei Methoden arbeiten ähnlich, **`getValues()`** liefert ein zweidimensionales Array des Typs double, welches die **`value`** Werte der Soma enthält, aus denen das **`field`** Array besteht. **`getActivations()`** liefert die Aktivierungen der **`Soma`** und **`getErrors()`** die Fehlerwerte.

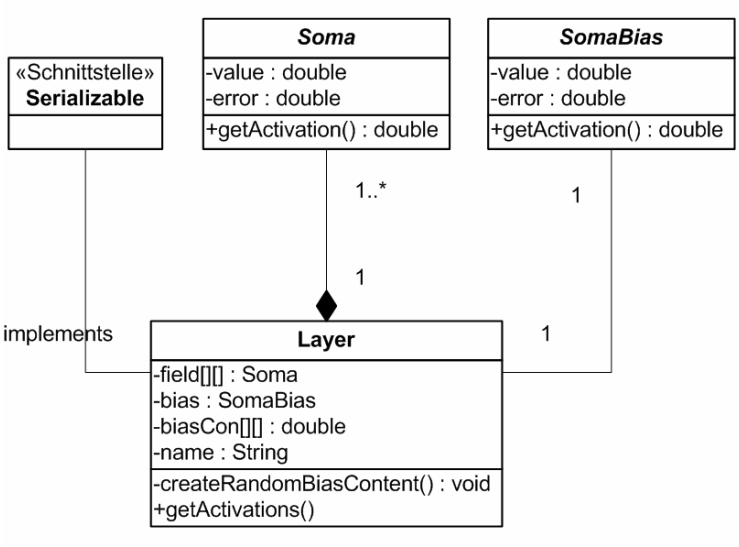


Abb. 19 Klassendiagramm der Klasse Layer

Die Soma-Klassen

Die Klasse **`Soma`** und ihre Derivate **`SomaBias`**, **`SomaInput`**, **`SomaHidden`** und **`SomaOutput`** repräsentieren die Neuronen eines neuronalen Netzes.
`Soma` steht dabei für ein Neuron und beinhaltet lediglich zwei Variablen des Typs **`double`**, **`value`**, den aktuellen Wert der aufsummierten Größen der eingehenden Verbindungen des Neurons und **`error`**, im dem der Fehlerwert des Neurons gespeichert wird (vgl. Abschnitt 2.3.3).
Neben den obligatorischen Methoden

- **`getValue()`**
- **`setValue(double value)`**
- **`getError()`**
- **`setError(double error)`**

enthält die Klasse nur eine weitere Methode namens

- **`getActivation(),`**

die jedoch als **`abstract`** deklariert ist und somit in den abgeleiteten Klassen definiert werden muss.

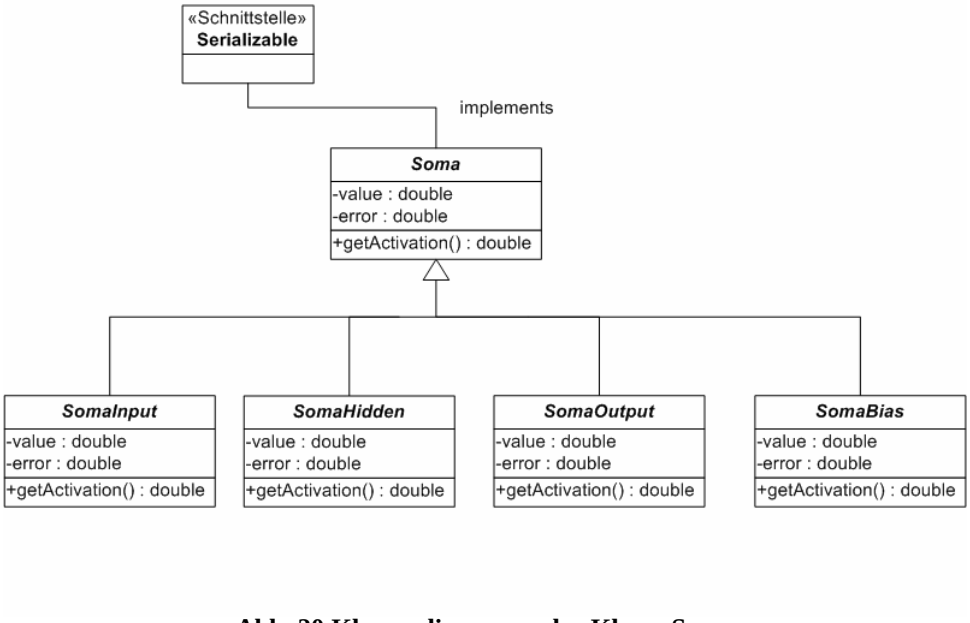


Abb. 20 Klassendiagramm der Klasse Soma

Die notwendige Unterscheidung der einzelnen **`Soma`**-Varianten liegt darin begründet, dass die Aktivierungsfunktion eines Soma von seiner Position im neuronalen Netz abhängig ist. So wird die abstrakte Methode **`getActivation()`** in jeder abgeleiteten Klasse verschieden implementiert:

- **`SomaInput`**
  Die Klasse **`SomaInput`** symbolisiert ein Neuron der Eingabeschicht. Laut Definition entspricht die Aktivierungsfunktion hier der Identitätsfunktion, d. h. die Eingabewerte werden weitergereicht. Folglich liefert **`getActivation()`** hier den Inhalt der Variable **`value`**.

- **`SomaHidden` und `SomaOutput`**
  Die Klassen **`SomaHidden`** und **`SomaOutput`** stehen für Neuronen der verborgenen und der Ausgabeschicht. Ihre Aktivierungen entsprechen der dem Rückgabewert der sigmoiden Funktion an der Stelle **`value`**. Inhaltlich sind diese beiden Klassen identisch, ihre Unterscheidung dient lediglich der Übersichtlichkeit.

- **`SomaBias`**
  Das Biasneuron gibt stets den Wert 1 zurück. Folglich ist 1 die Belegung des Werts **`value`** und auch der Rückgabewert der Methode **`getActivation()`**.

Die Klasse **`Connection`**

Die Verbindungen innerhalb der neuronalen Netze werden durch die Klasse **`Connection`** dargestellt. Hierbei steht *eine* Instanz dieser Klasse für alle Verbindungen zwischen *zwei* Schichten.
Neben dem String **`name`**, der den Namen der **`Connection`** enthält, besteht die Klasse **`Connection`** aus einem vierdimensionalen Array namens **`field`** aus Objekten des Typs **`Axon`**, in denen die Eigenschaften der Verbindung gehalten werden. Hierbei stehen die ersten beiden Dimensionen des Arrays für die *Startposition der Verbindung* in der vorderen zweidimensionalen Schicht, während die letzten beiden Dimensionen auf das Ziel-Soma in der hinteren Schicht deuten und entsprechend die *Koordinaten des Ziel-Somas* repräsentieren. Die Verbindung an Position *[a][b][c][d]* verbindet folglich das Soma an Position *[a][b]* der vorderen Schicht mit dem Soma an Position *[c][d]* der hinteren Schicht. Abbildung 22 zeigt dies an dem Beispiel eines zweischichtigen neuronalen Netzes. Die wichtigsten Methoden der Klasse **`Connection`** werden im Folgenden erläutert:

- **`void createRandomContent()`**
  Diese Methode belegt die Verbindungsgewichtungen der **`Connection`** mit Zufallswerten zwischen -0.5 und +0.5 und wird beim Start einer Simulation ausgeführt.

- **`void show()`**
  Diese Methode gibt die aktuelle Belegung der Verbindungsgewichtungen auf der Konsole aus. Auch wenn sie in LexLearn selbst nicht genutzt wird, ist sie für eine genauere Analyse der Verbindungsgewichtungen unerlässlich.

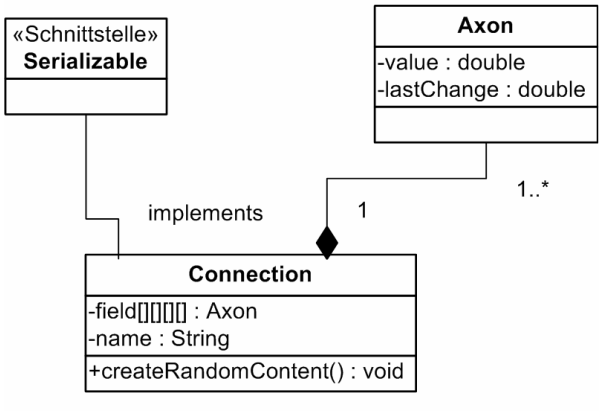


Abb. 21 Klassendiagramm der Klasse Connection

Die Klasse **`Axon`**

Das vierdimensionale Array **`field`** der Klasse **`Connection`** besteht aus Axonen. Eine Instanz der Klasse **`Axon`** symbolisiert dort genau eine Verbindung von einem **`Soma`** einer Schicht *n* zu einem **`Soma`** der Schicht *n+1*.
Es existieren zwei Klassenvariablen, **`value`** vom Typ **`double`**, die den Wert der Verbindungsgewichtung hält, und **`lastChange`**, ebenfalls vom Typ **`double`**, die die letzte Änderung der Verbindungsgewichtung speichert.

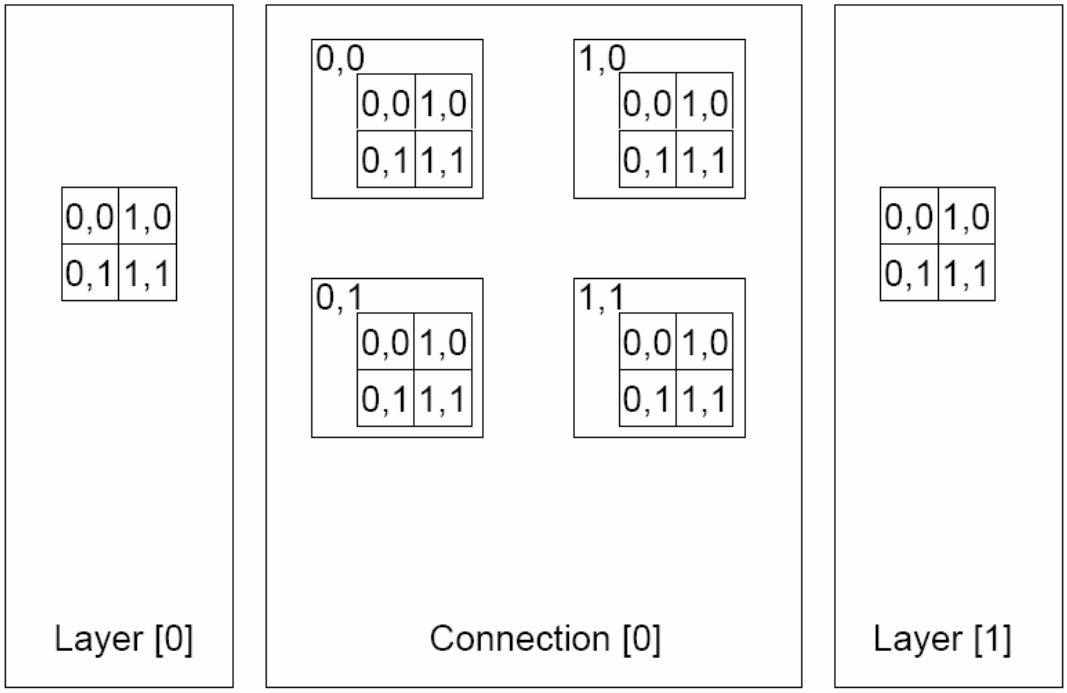


Abb. 22 Koordinatenansicht eines zweischichtigen implementierten neuralen Netzes

Die Instanzierung

Die Instanzierung der Klasse **`Agent`** erfolgt durch die Übergabe der Variablen

- **`double learningRate`**
- **`double momentum`**
- **`int[] layerlengths`**
- **`int[] layerwidths`**
- **`int wordLocation`**

innerhalb des Konstruktors. Während die Variablen **`learningRate`**, **`momentum`** und **`wordLocation`** lediglich lokal gespeichert werden, wird die komplette Netzstruktur aus **`layerlenghts`** und **`layerwidths`** abgeleitet.
Die Größen der einzelnen Schichten sind direkt aus den beiden Arrays ersichtlich. Daher wird das Array **`layers`** direkt mit in der Größe angepassten Instanzen der Klasse **`Layer`** gefüllt. Hierbei wird die Position der Schicht berücksichtigt, d. h. die Eingabeschicht besteht aus Objekten der Klasse **`SomaInput`**, die verborgenen Schichten aus Instanzen der Klasse **`SomaHidden`** und die Ausgabeschicht aus

Objekten der Klasse **`SomaOutput`**. Der folgende Quellcodeauszug führt die beschriebene Instanzierung aus.

```
…
for (int i = 0; i<laylengths.length; i++)
{
      if (i==0)
      {
            SomaInput[][] array = new SomaInput[laylengths[i]]
                                                [laywidths[i]];
            for (int a=0;a<laylengths[i];a++)
            {
                  for(int b=0; b<laywidths[i];b++)
                  {
                        array[a][b] = new SomaInput();
                  }
            }
            layers[i] = new Layer(array,"Layer "+i,
                              laylengths[i+1],laywidths[i+1]);
      }
      else{
            if (i== laylengths.length-1)
            {
                  SomaOutput[][] array = new SomaOutput
                                    [laylengths[i]][laywidths[i]];
                  for (int a=0;a<laylengths[i];a++)
                  {
                        for(int b=0; b<laywidths[i];b++)
                        {
                              array[a][b] = new SomaOutput();
                        }
                  }
                  layers[i] = new Layer(array,"Layer "+i);
            }
            else{
                  SomaHidden[][] array = new SomaHidden
                                    [laylengths[i]][laywidths[i]];
                  for (int a=0;a<laylengths[i];a++)
                  {
                        for(int b=0; b<laywidths[i];b++){
                              array[a][b] = new SomaHidden();
                        }
                  }
                  layers[i] = new Layer(array,"Layer "+i,
                                laylengths[i+1],laywidths[i+1]);
            }
      }
}
…
```

Die Verbindungen werden ebenfalls direkt bei der Instanzierung erzeugt. Die Struktur des vierdimensionalen Arrays richtet sich, wie bereits im vorherigen Abschnitt ausgeführt, ebenfalls nach **`layerlengths`** und **`layerwidths`**.

```
…
for (int j = 0; j<layers.length-1;j++)
{
      con[j] = new Connection(new Axon [laylengths[j]]
             [laywidths[j]][laylengths[j+1]][laywidths[j+1]]);
}
…
```

Um eine zufällige Belegung der Verbindungsgewichtungen zu gewährleisten, wird nach der Instanzierung die Methode

- **`void createRandomConnectionContent()`**

ausgeführt, die iterativ alle Instanzen der Klasse **`Connection`** im **`con`** Array durchläuft und dort lokal die Methode **`createRandomContent()`** aufruft.

Die Forward-Pass Berechnung

Die Forward-Pass Berechnung des neuronalen Netzes wird durch folgende fünf Methoden bewerkstelligt:

- **`void calculate(Layer in, Layer out, Connection con)`**
  Die Methode **`calculate()`** ist die grundlegende Methode der Propagierung von Werten innerhalb der Feedforward Netze. Sie bestimmt anhand der Aktivierungswerte der Schicht **`in`** und der Verbindung **`con`**, die **`in`** und **`out`** verbindet, die Netzeingaben der Ausgabeschicht **`out`**.
  Dabei wird das Produkt aller Gewichtungen der in ein Soma eingehenden Verbindungen mit den Aktivierungswerten der Soma, die die Startpunkte der jeweiligen Verbindungen darstellen, aufsummiert.

- **`void calculateAll()`**
  Diese Methode ruft iterativ, beginnend bei der Eingabeschicht bis hin zur Ausgabeschicht, die Methode **`calculate()`** auf und führt so eine Berechnung aller Werte innerhalb eines kompletten Feedforward Netzes durch.

- **`void calculateAll (double[][] in)`**
  Diese Methode setzt zuerst das in Array als Eingabewerte der Eingangsschicht und führt danach **`calculateAll()`** aus.

- **`void calculateFromUntil (int start, int end)`**
  Mit **`calculateFromUntil()`** ist eine partielle Berechnung des Forward-Pass Algorithmus möglich. Hierbei steht start für die Schicht, in der die Berechung startet, und end für die letzte zu berechnende Schicht.

- **`void calculateFromUntil (double[][] in, int start, int end)`**
  Analog zu **`calculateAll()`** existiert auch bei **`calculateFromUntil()`** eine Methode, die zuerst die Eingabeschicht neu belegt und danach die Berechnung startet.

Die Fehlerberechnung des Backward-Pass Algorithmus

Die Fehlerberechnung der aktuellen Belegung des neuronalen Netzes erfolgt in der Methode

- **`void calculateErrorBackwardPassWithWord (double[][] target,double[][] word,int location)`**

Die übergebenen Arrays stellen hier die Ziele der Verbesserung des neuronalen Netzes dar. Das Array **`target`** repräsentiert das Zielmuster, das in der letzten Schicht des neuronalen Netzes abgezeichnet werden soll, das Array **`word`** ist das Wort des Agenten, das der sein Netz optimierende Agent hört und das er als Ziel für seine Wortbildung nimmt (vgl. Abschnitt 2.3.3). Die Variable indiziert die Position der Wortschicht innerhalb des neuronalen Netzes, da es bei mehreren verborgenen Schichten theoretisch mehrere mögliche Positionen der Wortplatzierung gibt.

a) Output
Der Algorithmus durchläuft das neuronale Netz rückwärts, beginnend bei der Ausgabeschicht. Dort wird für jedes Neuron ein individueller Fehlerwert berechnet. Dieser ergibt sich aus der Differenz zwischen dem Wert der entsprechenden Zelle des Zielmusters und dem Produkt aus dem aktuellen Aktivierungswert mit dem Funktionswert der sigmoiden Ableitung der Netzeingabe des Neurons. Dieser Wert wird daraufhin in der Variable **`error`** der Klasse **`SomaOutput`** gespeichert.

b) Hidden
Von diesem Fehlerwert aus können auch die theoretischen Fehlerwerte der Neuronen der verborgenen Schichten berechnet werden. Beginnend bei der hintersten verborgenen Schicht, durchläuft eine Schleife alle verborgenen Schichten bis hin zur vordersten, die auf die Eingabeschicht folgt. Die Berechnung der Fehlerwerte ist in jeder Schicht identisch, eine notwendige Abweichung ergibt sich aber in der Schicht, die das Wort repräsentiert.

- kein Wort
  Ist die aktuell zu berechnende Schicht ungleich der Wortschicht, so wird in jedem Neuron, ebenfalls im Biasneuron, zunächst die Summe aller

Produkte der Stärken aller ausgehenden Verbindungen mit den Fehlerwerten der Neuronen, die das Ziel der Verbindungen darstellen, errechnet. Dieser Wert wird darauf mit der Ableitung der sigmoiden Funktion der Netzeingabe des Neurons multipliziert. Der sich daraus ergebende Wert wird in der Variable **`error`** der Klasse **`SomaHidden`** gespeichert.

- Wort
  In der Wortschicht funktioniert die Fehlerberechnung ähnlich. Die einzige Erweiterung des bereits beschriebenen Algorithmus ist, dass bei der Aufsummierung der Produkte von Verbindungsstärken und Fehlern der nachfolgenden Schicht auch für jedes Neuron die Differenz zwischen dem Zielwert, also dem Wert des Wortes des anderen Agenten, und dem aktuellen Wert des Neurons mit berücksichtigt wird. Diese Differenz bildet gewissermaßen einen „Extra-Summanden“.

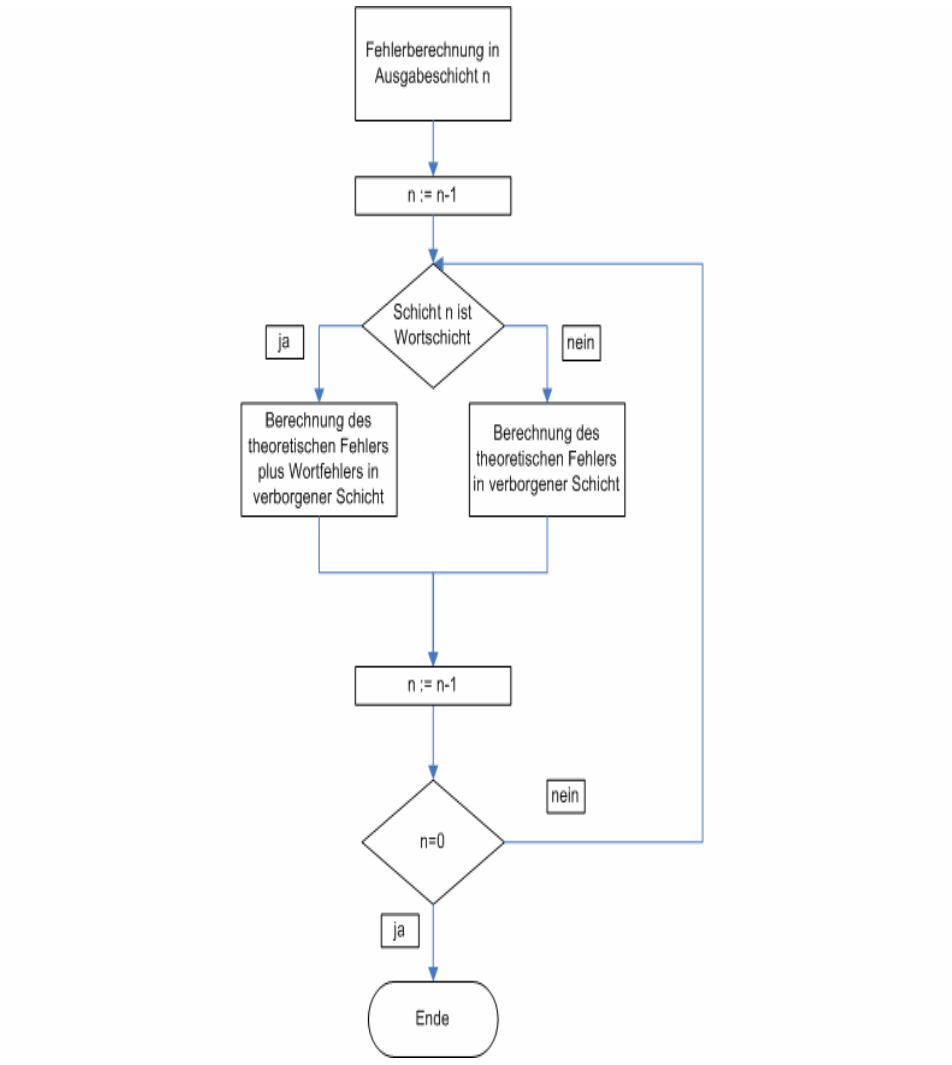


Abb. 23 Flussdiagramm der Fehlerberechnung

Der folgende Auszug dem Quellcode zeigt die in Abbildung 23 dargestellte Fehlerberechnung.

```
…
for (int n=getLayers().length-2;n>0;n--)
{
  for (int j=0; j<(getLayers()[n].getField().length);j++)
  {
   for (int k=0; k<getLayers()[n].getField()[j].length;k++)
   {
    for (int l=0;l<getLayers()[n+1].getField().length;l++)
    {
     for (int m=0; m<getLayers()[n+1].getField()[l].length;
                                                                  m++)
     {
      getLayers()[n].getField()[j][k].setError(getLayers()[n]
                .getField()[j][k].getError()+getLayers()[n+1]
                    .getField()[l][m].getError()* getCon()[n]
                         .getField()[j][k][l][m].getValue());
     }
    }
    double wordError = 0;
    if (n==location)
    {
      wordError = word[j][k]-getLayers()[n].getField()[j][k]
                                                  .getActivation();
    }
    getLayers()[n].getField()[j][k].setError
                   ((getLayers()[n].getField()[j][k].getError()
            +wordError)*Tools.sigmoidDerivation(getLayers()[n]
                                .getField()[j][k].getValue()));
   }
  }
  //bias Error
  for(int l=0;l<getLayers()[n+1].getField().length;l++)
  {
   for (int m=0; m<getLayers()[n+1].getField()[l].length;m++)
   {
      getLayers()[n].getBias().setError(getLayers()[n]
      .getBias().getError()+getLayers()[n+1].getField()[l][m]
             .getError()* getLayers()[n].getBiasCon()[l][m]);
   }
  }
  getLayers()[n].getBias().setError(getLayers()[n]
                .getBias().getError()*Tools.sigmoidDerivation
                      (getLayers()[n].getBias().getValue()));
}
…
```

Die Gewichtsmodifikation des Backward Pass Algorithmus

Die Gewichtsmodifikation des neuronalen Netzes anhand der vorher berechneten Fehlerwerte geschieht in der Methode

- **`void modifyWeights()`**

In einer Schleife werden alle Instanzen der Klasse **`Connection`** im Array **`con`** durchlaufen, beginnend bei der letzten Stelle des Arrays, welche die Verbindungen zwischen der hintersten verborgenen Schicht und der Ausgabeschicht darstellt, bis hin zur ersten Position, wo die Verbindungen zwischen der Eingabeschicht und der ersten verborgenen Schicht zu finden sind.
Dort wird für jede Verbindung aller Neuronen, ebenfalls dem Bias-Neuron, der vorderen zur hinteren Schicht zuerst die Variable **`deltawij`** des Typs **`double`**, die Differenz zwischen der aktuellen und der gewünschten Verbindungsstärke, bestimmt. Zur Berechnung von **`deltawij`** wird zuerst das Produkt aus Lernrate, dem Fehlerwert des Endneurons einer Verbindung und dem Aktivierungswert des Startneurons einer Verbindung gebildet. Daraufhin wird **`deltawij`** in der Art neu gewichtet, dass die aktuell berechnete Veränderung mit dem Faktor *(1-***`momentum`***)* multipliziert und die Veränderung des vorherigen Simulationsschritts gewichtet mit dem Faktor **`momentum`** hinzuaddiert wird. Zuletzt wird **`deltawij`** zur aktuellen Verbindungsgewichtung addiert. Damit ist die Gewichtsmodifikation abgeschlossen. Der folgende Ausschnitt aus dem Quellcode, eines der Herzstücke LexLearns, stellt die Gewichtsmodifikation explizit dar.

…

```
for (int h = getCon().length-1; h>=0;h--)
{
 for (int i = 0; i<getCon()[h].getField().length;i++)
 {
  for (int j = 0; j<getCon()[h].getField()[i].length;j++)
  {
   double deltawij = 0;
   for (int k = 0; k<getCon()[h].getField()[i][j].length;k++)
   {
    for(int l=0;l<getCon()[h].getField()[i][j][k].length;l++)
    {
     deltawij=learningRate*getLayers()[h+1].getField()[k][l]
                    .getError()*getLayers()[h].getField()[i][j]
                                                .getActivation();
     deltawij = (1-momentum)*deltawij+momentum*getCon()[h]
                          .getField()[i][j][k][l].getLastChange();
     getCon()[h].getField()[i][j][k][l].setLastChange(deltawij);
     getCon()[h].getField()[i][j][k][l].setValue(getCon()[h]
                    .getField()[i][j][k][l].getValue()+deltawij);
```

```
   }
  }
 }
 double deltawij = 0;
 for(int l=0;l<getLayers()[h].getBiasCon().length;l++) {
  for (int m=0; m<getLayers()[h].getBiasCon()[l].length;m++)  {
   deltawij =  learningRate*getLayers()[h+1].getField()[l][m]
                                                    .getError();
   getLayers()[h].getBiasCon()[l][m] += deltawij;
  }
 }
}
…
```

4.1.2 Aufbau der Klasse Environment

Die Klasse **`Environment`** repräsentiert eine Simulationsumgebung und besteht aus einer Reihe von Klassenvariablen, die die notwendigen Attribute einer Umgebung speichern.

- **`int numberOfAgents`**
  Diese Integervariable, die die Klasse **`Environment`** bei der Instanzierung erhält, speichert die Anzahl der Agenten, die sich in der Simulationsumgebung befinden.

- **`Agent[] agents`**
  Dieses Array aus Instanzen der Klasse Agent beinhaltet die Agenten der Umgebung.

- **`double[][][] inputPattern`**
  Dieses dreidimensionale Array des Typs **`double`** beinhaltet die Menge der visuellen Szenen. Hierbei indiziert die erste Dimension die Position der Szene im Array, die zweite und dritte Dimension stehen für die Höhe und Breite des Patterns.

- **`int simulationStep`**
  Die Variable **`simulationStep`** speichert den aktuellen Simulationsschritt der Simulationsumgebung.

Bei der Instanzierung der Klasse **`Environment`** werden dem Konstruktor folgende Attribute übergeben:

- **`int numberOfAgents`**
  Die Variable **`numberOfAgents`** bestimmt die Anzahl der Agenten, aus denen die Umgebung bestehen soll.

- **`double[][][] inputPattern`**
  Das Array mit dem Namen **`inputPattern`** beinhaltet die Menge der visuellen Szenen.
- **`double learningRate`**
- **`double momentum`**
- **`int[] laylengths`**
- **`int[] laywidths`**
- **`int wordLocation`**
  Diese Werte werden ausschließlich zur Instanzierung der Klasse Agent benötigt (vgl. Konstruktor der Klasse Agent).

Ebenso beinhaltet die Klasse **`Environment`** eine Menge von Methoden, die sowohl für die Instanzierung der Agenten als auch für den Ablauf der Simulation benötigt werden. Im Folgenden werden die wichtigsten kurz erläutert:

- **`void generateAgents (double learningRate, double momentum, int[] laylengths, int[] laywidths, int wordLocation)`**
  Diese Methode instanziert die Agenten der Umgebung und führt in jedem Agenten die Methode **`createRandomConnectionContent()`** aus, mit der die Verbindungsstärken zufällig belegt werden. Damit sind die Agenten vollständig auf einen Simulationslauf vorbereitet.

- **`Agent selectAgent()`**
  Mit der Methode **`selectAgent()`** wird zufällig ein Agent aus dem Array von Agenten ausgewählt. Sie wird bei der Auswahl zweier Agenten in einem Simulationsschritt benötigt.

- **`double[][] selectInputPattern()`**
  Analog zur vorher genannten Methode wird hier zufällig ein Pattern aus der Menge der visuellen Szenen ausgewählt.

- **`int[][] createSceneSet()`**
  Die Methode **`createSceneSet()`** erstellt die Menge aller Zweitupel von visuellen Szenen, welche bei der Berechnung des Avg1 benötigt wird.

- **`int[][] createAgentSet()`**
  Diese Methode kreiert die Menge aller Zweitupel von Agenten. Sie wird bei der Berechnung des Avg2 benötigt.

- **`double calculateDistance(double[][] in, double[][] out)`**
  Die hier genannte Methode berechnet die Differenz zwischen zwei übergebenen Mustern. Sie wird sowohl zur Berechnung des Avg1, in der sie die Differenz zweier Wortschichten von Agenten bestimmt, als auch bei der Bestimmung des Avg2, in der sie den Unterschied zwischen den Ausgabeschichten bestimmt, benötigt.

- **`double calculateAvg1()`**
  In der Methode **`calculateAvg1()`** wird der Fehlerquotient Avg1 berechnet. Dies geschieht durch Aufsummieren und anschließende Mittelwertsbildung der Differenz der Wortschichten eines Agenten für jeweils zwei visuelle Szenen für alle Tupel der Menge, die in **`createSeceneSet()`** gebildet wurde. Dies wird für jeden Agenten der Gemeinschaft berechnet und es wird der Mittelwert der individuellen Abweichungen der Agenten gebildet.

- **`double calculateAvg2()`**
  Die Methode **`calculateAvg2()`** errechnet den Fehlerquotient Avg2. Dies geschieht durch die Differenzbildung der Inhalte der errechneten Ausgabeschicht zweier Agenten. So wird für jede visuelle Szene und jeden möglichen Tupel zweier Agenten die Differenz aufsummiert und danach der Mittelwert berechnet.

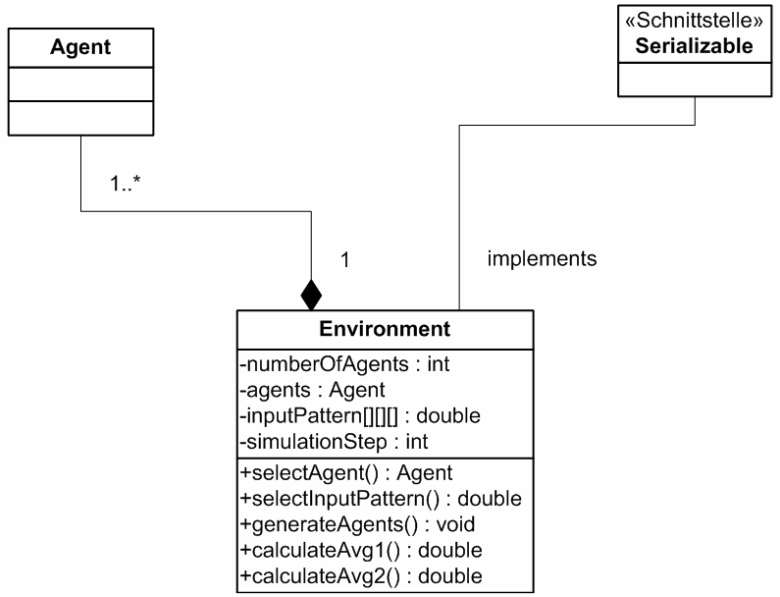


Abb. 24 Klassendiagramm der Klasse Environment

4.1.3 Aufbau der Simulation

Die Klasse **`Simulation`** ist die Hauptklasse des Simulationsbereichs von LexLearn. Durch die in diesem Unterkapitel beschriebenen Komponenten der Klasse **`Simulation`** ist es möglich, alle Arten von Simulationen zu deklarieren und durchzuführen. Die Klasse grenzt damit klar den Simulationsbereich von anderen Teilen des Programms ab und ist so die einzige Schnittstelle zur grafischen Benutzeroberfläche, die im nächsten Kapitel erläutert wird.
In der Klasse **`Simulation`** kommt das Singleton Pattern zur Anwendung, damit gewährleistet ist, dass keine weitere Instanz, außer der im Vorhinein statisch angelegten **`instance`**, erzeugt wird. Auf sie kann mit Hilfe der Methode **`getInstance()`** zugegriffen werden. Dadurch ergibt sich, dass es keinen

öffentlichen Konstruktor der Klasse **`Simulation`** geben kann. Alle Klassenvariablen müssen folglich explizit gesetzt werden.

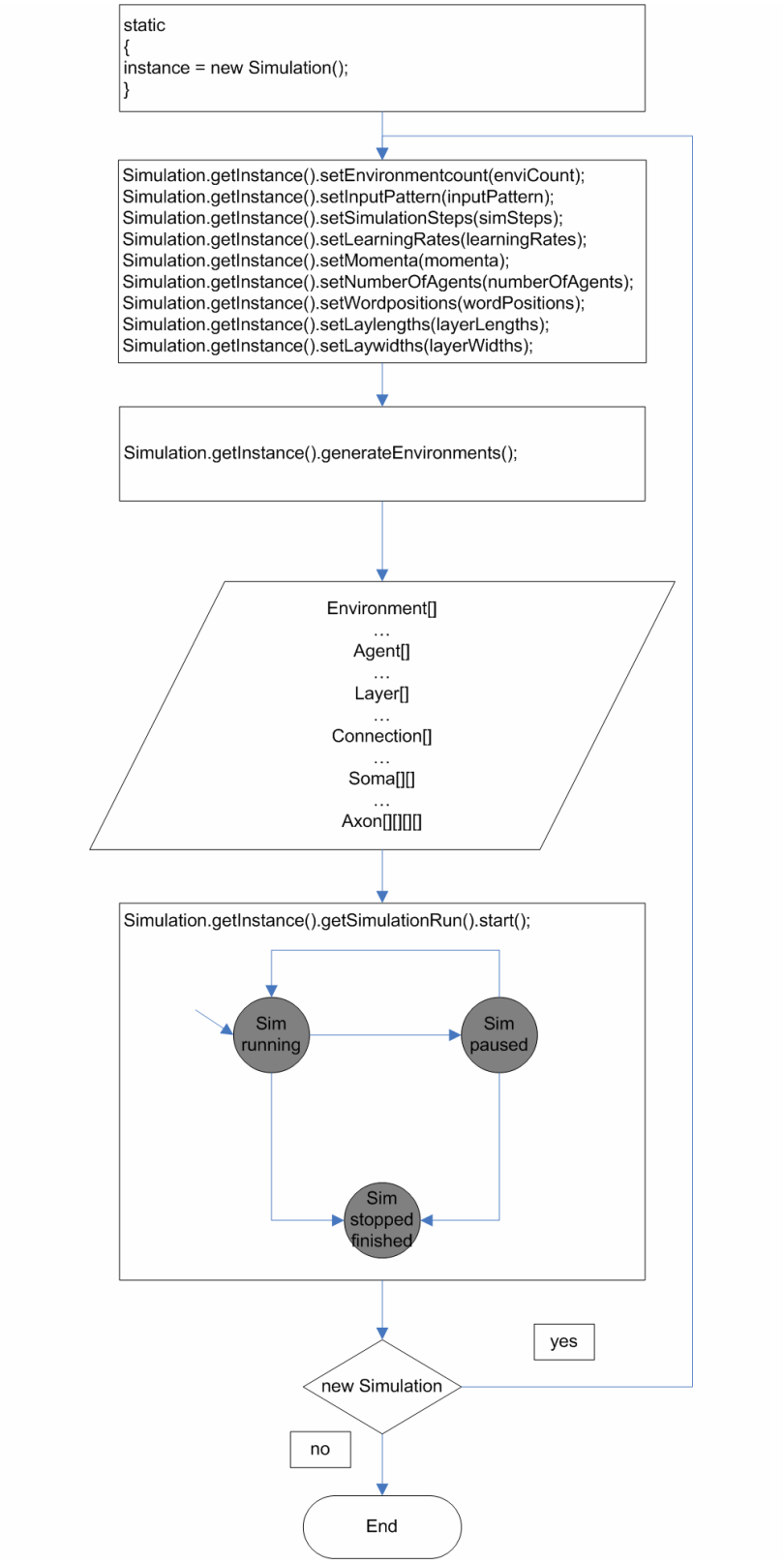


Abb. 25 Flussdiagramm zu Erstellung und Durchlauf einer Simulation

Durch die Notwendigkeit, die Daten mehrerer Simulationsumgebungen mit allen notwendigen Angaben in der Klasse **`Simulation`** zu speichern, ergibt sich eine Reihe von Klassenvariablen, die im Folgenden erläutert werden:

- **`int environmentCount`**
  Diese Variable speichert die Anzahl der Simulationsumgebungen.

- **`int simulationSteps`**
  Die Variable **`simulationSteps`** speichert die Anzahl der Simulationsschritte.

- **`double[][][] inputPattern`**
  Dieses dreidimensionale Array des Typs **`double`** speichert die Sequenz der visuellen Szenen, die den Agenten der Simulationsumgebungen präsentiert wird.

- **`int[] numberOfAgents`**
  Das Array **`numberOfAgents`** speichert die Agentenanzahl der verschieden Simulationsumgebungen lokal.

- **`double[] learningRates`**
- **`double[] momenta`**
- **`int [][] laylengths`**
- **`int [][] laywidths`**
- **`int[] wordpositions`**
  Diese Arrays speichern die Eigenschaften der Agenten aller Instanzen der Klasse **`Environment`**. Die erste Dimension der Arrays indiziert die Simulationsumgebung, die folgenden Bedeutungen werden im Abschnitt 4.1.2 erläutert.

- **`Environment[] environments`**
  In diesem Array aus Instanzen der Klasse **`Environment`** werden die Simulationsumgebungen des Programms gehalten. Sie werden durch die Methode **`generateEnvironments()`** erzeugt.

- **`int status`**
  Die Variable **`status`** speichert den aktuellen Status eines Simulationsdurchlaufs. Hierbei gibt es vier verschiedene Belegungen:
  0. Die Instanzen der Klasse **`Environment`** sind noch nicht erzeugt worden.
  1. Die Simulationsumgebungen wurden erzeugt und die Simulation läuft.
  2. Die Simulation wurde durch den Benutzer angehalten und befindet sich im Pause-Zustand.
  3. Die Simulation wurde beendet. Dieser Zustand kann sowohl durch Abarbeiten aller Simulationsschritte als auch durch Drücken des Stop-Buttons durch den Benutzer erreicht werden.

- **`DoSimulation simulationRun`**
  Die benutzeroberflächenspezifischen Teile der Berechnung eines Simulationsdurchlaufs sind in die Klasse **`DoSimulation`** ausgelagert. Der Vorteil dieser Implementierungsvariante ist, dass die Berechnung so in einem eigenen Thread geschieht. Dies hat für den Simulationsteil von LexLearn zwar keine weiteren Konsequenzen, jedoch ist es dank dieser Konstruktion möglich, stets den aktuellen Stand der Avg1 und Avg2 Berechnungen anzuzeigen. Der Nutzer wird also permanent mit den neuesten Daten der Simulationsentwicklung versorgt und muss nicht bis zum Ende der Simulationsdurchführung warten, bis der Simulationsverlauf sichtbar ist (vgl. Abschnitt 4.2.1 Unterabschnitt Chartpanel).

Die wichtigsten Methoden der Klasse Simulation werden im Folgenden beschrieben.

- **`static void save (String path)`**
  Die Methode **`save`** speichert den aktuellen Zustand der Klasse **`Simulation`** binär unter dem im Konstruktor übergebenen Dateipfad.

- **`static void load (String path)`**
  Analog zur Methode **`save`** lädt die Methode **`load`** die kompletten Daten einer vorher gespeicherten Simulationsklasse.

- **`void generateEnvironments()`**
  Diese Methode wird beim Starten eines Simulationsdurchlaufs ausgeführt. Erst zu diesem Zeitpunkt werden die Instanzen der Klasse **`Environment`** generiert, auf denen danach die Simulation ausgeführt wird. Der Vorteil dieser Variante ist, dass sich Änderungen in den Simulationsparametern so nur auf lokal gespeicherte Variablen auswirken und keine Daten in komplexeren, ineinander verschachtelten Klassen verändert werden müssen.

- **`void simulationStart()`**
  Die Methode **`simulationStart()`**, die beim Starten eines Simulationsdurchlaufs ausgeführt wird, instanziert die Klasse **`DoSimulation`** und führt dort die Methode **`start()`** aus.

- **`void simulationPause()`**
  Diese Methode hält den Thread, in dem die Simulation durchgeführt wird, bei laufender Berechnung an; sollte sie bereits im Pausezustand sein, so wird die Simulation fortgesetzt.

- **`void simulationStop()`**
  Mit der Methode **`simulationStop()`** wird der Simulationsdurchlauf und damit der zugehörige Thread beendet.

- **`void doSimulation (JFreeChart chart1,XYSeries[] avg1Series,JFreeChart chart2,XYSeries[] avg2Series)`**
  In dieser Methode wird die Simulation durchgeführt. Aus Gründen der Visualisierung wird sie aus der Klasse **`DoSimulation`** aufgerufen. Neben dem eigentlichen Simulationsablauf, also der Auswahl der Agenten und visuellen Szenen, und der Berechnung der Fehlerindizes Avg1 und Avg2 ist sie auch Schnittstelle zur Benutzeroberfläche, sie modifiziert die Daten der Chartfenster (vgl. Abschnitt 4.2.1 Unterabschnitt Chartpanel).

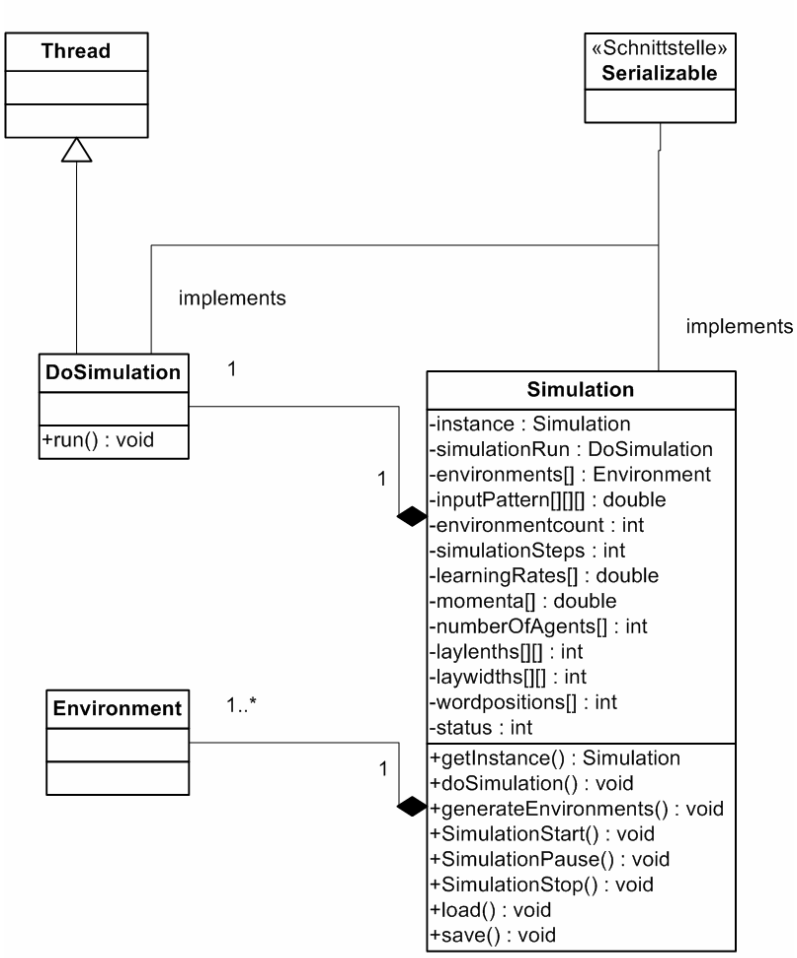


Abb. 26 Klassendiagramm der Klasse Simulation

4.2 Graphische Benutzeroberfläche

Das Design der graphischen Benutzeroberfläche in LexLearn ist auf die Erfüllung zweier Eigenschaften ausgelegt. Zum einen ist der strukturelle Aufbau sehr *einfach* gestaltet, um einen mit neuronalen Netzen nicht erfahrenen Benutzer einen barrierefreien Einstieg zu gewährleisten. Zum anderen garantiert die Oberfläche eine größtmögliche Ausschöpfung aller *Funktionalitäten* der darunter liegenden Simulationsklassen, um auch komplexere Simulationen erstellen zu können.

4.2.1 Aufbau Hauptfenster

Das Hauptfenster in LexLearn ist eine Instanz der Klasse **`MainWindow,`** welche aus der Klasse **`JFrame`** abgeleitet ist. Die Klasse **`JFrame`** bildet die Hauptfensterklasse innerhalb der Swing-Bibliotheken, deren Hauptfenster im Unterschied zum AWT nur eine einzige Hauptkomponente **`JRootPane`**, die alle anderen Komponenten aufnimmt, besitzt.

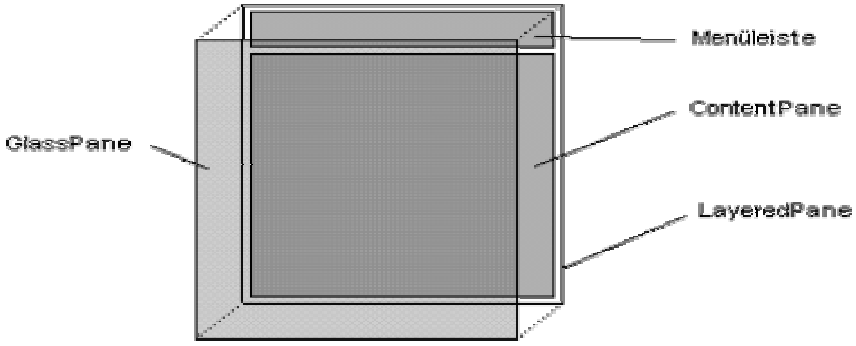


Abb. 27 Struktur einer RootPane
Quelle: [Kru00, S. 755, Abb. 36.2]

Während die über der LayeredPane liegende GlassPane in der Regel nicht zur Grafikausgabe genutzt wird, enthält eine aus **`JLayerdPane`** abgeleitete LayeredPane die Menuleiste (abgeleitet aus **`JMenuBar`**) und Dialogelemente innerhalb der ContentPane (abgeleitet aus **`Container`**). Abbildung 28 zeigt das zugehörige Klassendiagramm der Klasse **`MainWindow`**.

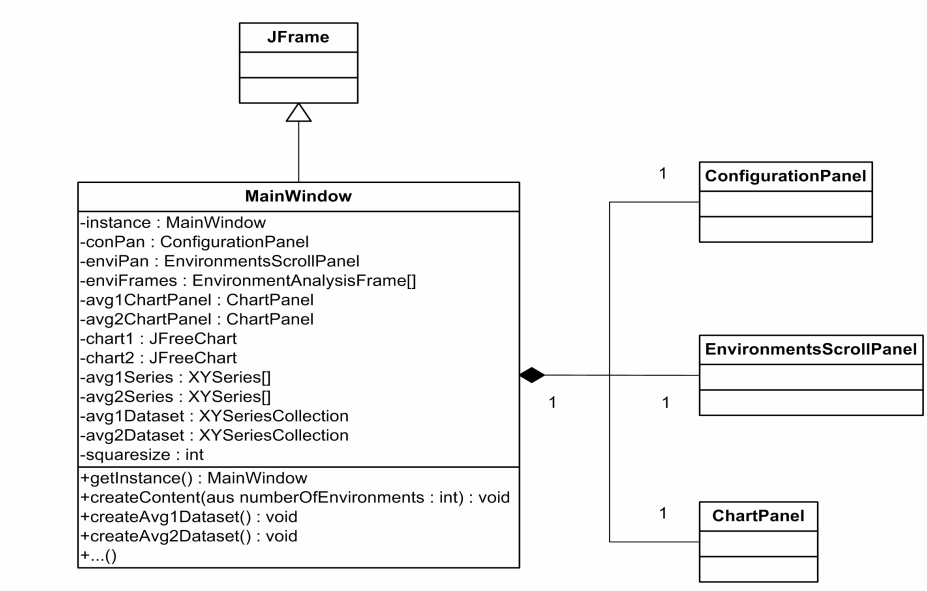


Abb. 28 Klassendiagramm der Klasse MainWindow

Mit Hilfe des Singelton-Patterns kann pro Programmstart immer nur eine Instanz der Klasse **`MainWindow`** erzeugt werden, die mit Hilfe der Methode **`getInstance()`** aufgerufen werden kann. Die ContentPane dieser Instanz besteht aus vier weiteren Komponenten, die jeweils aus **`JPanel`** abgeleitet sind:

- ein **`ConfigurationPanel`**, welches alle Dialogelemente zur Konfiguration einer Simulation beinhaltet;
- ein **`EnvironmentsScrollPanel`**, ein scrollbares JPanel, das alle erstellten Simulationsumgebungen auflistet;
- zwei **`ChartPanel`**, welche zur graphischen Darstellung der beiden Fehlerindikatoren Avg1 und Avg2 benötigt werden.

Alle vier Komponenten werden mit Hilfe des *Layoutmanagers* **`GridBagLayout`** innerhalb der ContentPane ausgerichtet.

```
…
Container cp = getContentPane();
cp.setLayout(new GridBagLayout());
GridBagConstraints c = new GridBagConstraints();
conPan = new ConfigurationPanel();
…
cp.add(conPan,c);
…
```

Durch das Belegen einzelner Attribute der **`GridBagConstraints c`**, wie z. B. x-/y- Koordinate, Höhe und Breite, wird die Lage und Ausrichtung einer Komponente innerhalb der ContentPane festgelegt, sodass der Aufbau des Hauptfensters dem in Abschnitt 3.3.2 beschriebenen Entwurfsschema entspricht.
Neben den einzelnen Instanzen der Komponenten des Hauptfensters, werden in der Klasse **`MainWindow`** weitere Attribute gehalten:

- **`EnvironmentAnalysisFrame[] enviFrames`** beinhaltet ein Array der zur Laufzeit geöffneten Analysefenster, welche in Abschnitt 4.2.2 genauer erläutert werden. Wird eine Simulation neu gestartet, so müssen alle aktuell geöffneten Analysefenster wieder geschlossen werden.
- **`XYSeriesCollection avg1Dataset`** und **`XYSeriesCollection avg2Dataset`** geben jeweils Informationen über die Anzahl und Bezeichnungen der in **`chart1`** und **`chart2`** zu erstellenden JFreeCharts.
- **`XYSeries[] avg1Series`** und **`XYSeries avg2Series`** enthalten jeweils die Punktemengen für jeden zu zeichnenden Graphen. Eine genauere Erläuterung folgt in dem Unterabschnitt ChartPanel.
- **`int squaresize`** beinhaltet die Kantenlänge der – zur Darstellung der verwendeten Eingangsmuster benutzten – Einzelquadrate. Sie ist vom Benutzer nicht veränderbar und mit dem Wert 25 (Pixel) belegt.

Die wichtigsten Methoden der Klasse `MainWindow` sind:

- **`MainWindow getInstance()`**, sie gibt immer die aktuelle Instanz der Klasse **`MainWindow`** zurück;

- **`void createContent(numberOfEnvironments:int)`**, welche den kompletten Inhalt der ContentPane des Hauptfensters in Abhängigkeit von der Anzahl der Simulationsinstanzen generiert;

- **`void createAvg1Dataset()`** und **`void createAvg2Dataset()`**, die in Abhängigkeit von der aktuellen Anzahl der Simulationsumgebungen die Inhalte des **`avg1Dataset`** und **`avg2Dataset`** berechnen.

Im Folgenden werden die einzelnen Komponenten des Hauptfensters genauer betrachtet.

Menuleiste

Die Menuleiste in LexLearn wird durch eine Instanz der Klasse **`myMenuBar`** repräsentiert und ist ebenfalls mit Hilfe des Singletonpatterns erstellt worden, sodass nur eine einmalige Instanzierung möglich ist.

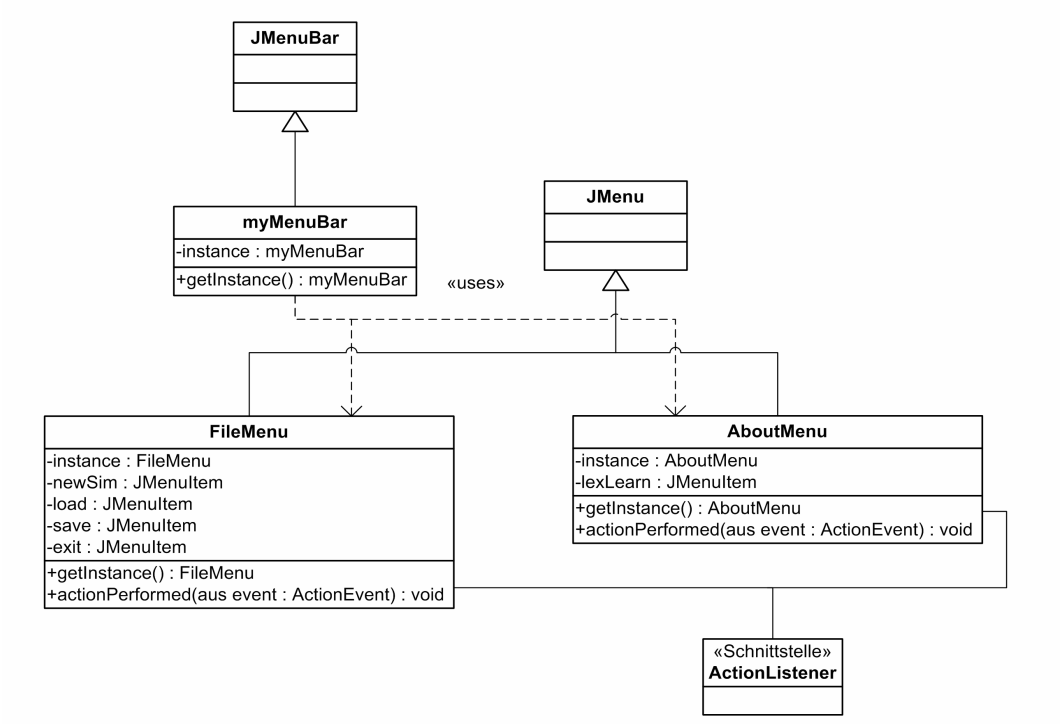


Abb. 29 Klassendiagramm der Klasse myMenuBar

Wie in Abbildung 29 ersichtlich, besteht die Menuleiste aus einem Datei- und Infomenu. Das Dateimenu besteht aus vier verschiedenen Elementen des Typs **`JMenuItem`**. Durch das Implementieren der **`ActionListener`** Schnittstelle und

das Überschreiben der Methode **`actionPerformed(ActionEvent event)`** kann jedes Element des Dateimenus seine folgende Aufgabe erfüllen:

- Das **`JMenuItem newSim`** ist für das Erstellen eines **`NewSimulationDialogs`** verantwortlich und ermöglicht dem Benutzer, eine neue Simulation zu konfigurieren (siehe Unterabschnitt NewSimulationDialog).

- Das Menuelement **`load`** ist für das Laden einer bereits gespeicherten Simulation zuständig. Eine Instanz der Klasse **`JFileChooser`** übernimmt hierbei die Auswahl eines Dateipfades **`path`** mit Hilfe eines Dialogs. Über **`Simulation.load(path)`**werden alle Attribute der ausgewählten Simulation geladen und das Hauptfenster entsprechend dieser neu gezeichnet, d. h. dass sich sowohl die EnvironmentScrollPanel anhand der in der Simulation abgespeicherten Simulationsumgebungen und deren einzelnen Attributen aufbaut als auch die für das Darstellen des Avg1 und Avg2 verantwortlichen ChartPanels gezeichnet und mit einer Legende versehen werden.

- Das **`JMenuItem save`** ermöglicht das Speichern einer Simulation. Wie schon beim Laden einer Simulation übernimmt ein Objekt der Klasse **`JFileChooser`** die Auswahl des Dateipfades, in dem eine Simulation abgespeichert werden soll. Als Dateinamen gibt dieser Dialog **`*.sim`** vor, dies kann jedoch vom Benutzer beliebig geändert werden. Nach sukzessivem Einlesen der einzelnen Textfelder innerhalb der EnvironmentsScrollPanel (siehe EnvironmentsScrollPanel) werden alle Attribute der einzelnen Simulationsumgebungen innerhalb der Instanz der Klasse **`Simulation`** gesetzt. Durch den Aufruf der Methode **`Simulation.save(path)`** wird die Simulation binär abgespeichert (siehe Abschnitt 4.1.3).

- Das Menüelement **`exit`** beendet durch den Aufruf von **`System.exit(0)`** das Programm.

Das Infomenu bietet lediglich eine Übersicht über die aktuelle Version von LexLearn.

ConfigurationPanel

Innerhalb der ConfigurationPanel befinden sich wesentliche Konfigurations- und Steuerungselemente für eine erstellte Simulation. Wie in Abbildung 30 ersichtlich, besteht dieser Teil des Hauptfensters aus zwei beschrifteten Textfeldern und fünf verschiedenen JButtons. Über das **`JTextField simStepsText`** kann die Anzahl der Simulationsschritte eingeben werden. Das **`JTextField inputPatternText`** zeigt durch eine Beschriftung und seine Hintergrundfarbe an, ob eine Menge von Eingangsmustern geladen wurde (grün) oder nicht (rot).

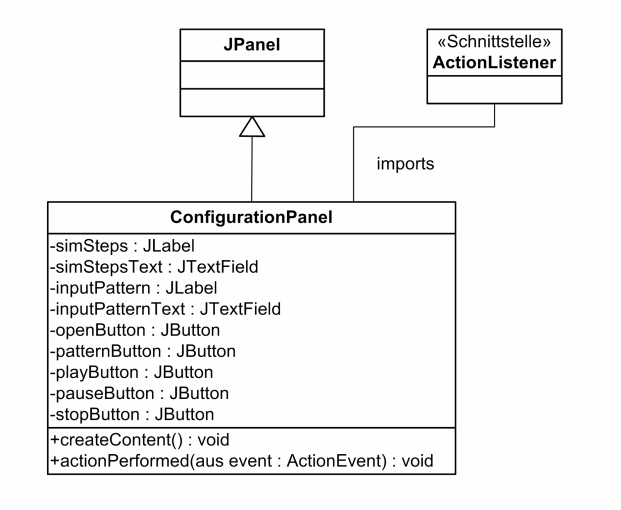


Abb. 30 Klassendiagramm der Klasse ConfiguarationPanel

Mit Hilfe des **`JButton openButton`** kann über ein Dateidialogfenster eine CSV-Datei als Eingabemuster eingelesen werden. Dabei wird unter Berücksichtigung des eingelesenen Dateipfades eine Instanz der Klasse **`CSVParser`** des Paketes **`tools`** erzeugt und mit Hilfe der Methode **`readFile()`** eine Menge von Eingabemustern in der aktuellen Simulation als dreidimensionales Array von Double-Werten gesetzt.

```
…
String path = choose.getSelectedFile().getPath();
…
CSVParser parse = new CSVParser(path);
try {
      Simulation.getInstance().setInputPattern(parse.readFile());
      …
      patternText.setBackground(Color.GREEN);
      patternText.setText(ConstantStrings.INPUPATTERNLOADED);
      …
      }
catch (Exception e) {
      System.out.println("Error! Pattern can't be read!!");
      e.printStackTrace();
}
…
```

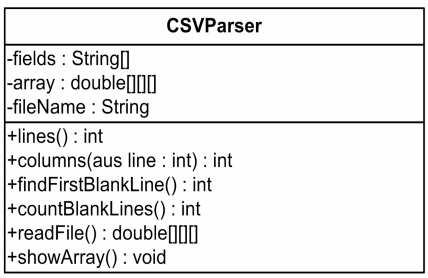


Abb. 31 Klassendiagramm der Klasse CSVParser

Damit eine CSV-Datei erfolgreich geparst werden kann, müssen die einzelnen Eingabemuster durch eine Leerzeile getrennt und von gleicher Länge bzw. Breite sein. Innerhalb der Muster werden die einzelnen Felder, welche aus positiven, reellen Zahlen bestehen, durch „ ; “ oder „ , “ getrennt.
Nach erfolgreichem Einlesen wird der **`JButton patternButton`** aktiviert und durch die Erzeugung einer Instanz der Klasse **`ShowPatternDialog`** wird eine visuelle Übersicht auf die geladene Menge von Eingabemustern erzeugt. Jeder **`ShowPatternDialog`** besitzt ein Objekt der Klasse **`PatternScrollPanel`**, um eine Analyse größerer Mengen von Eingabemustern zu ermöglichen. Dieses scollbare JPanel ist wiederum ein Container für eine Instanz der Klasse **`PatternPanel`**, welche alle Eingabemuster in Form von **`SinglePatternPanel`** Objekten beinhaltet.

```
…
for (int i=0;i<inputPattern.length;i++)
{
 singPat[i] = new SinglePatternPanel(inputPattern[i],squaresize,i);
 add(singPat[i]);
}
…
```

Ein **`SinglePatternPanel`** Objekt besitzt jeweils ein zweidimensionales Array mit den reellen Zahlenwerten des Eingabemusters, eine ganzzahlige Angabe **`squaresize`** über die Größe der zu zeichnenden Quadrate und den jeweiligen Index `i` eines Musters als Attribute, welche es zu einer zweidimensionalen Darstellung der Muster in Form von Quadraten nutzt. Das eigentliche Zeichnen dieser Werte erfolgt mit Hilfe der Klasse **`GraphicPattern`**, innerhalb derer eine Überschreibung der aus dem AWT-Toolkit bekannten Methode **`paint()`**erfolgt.

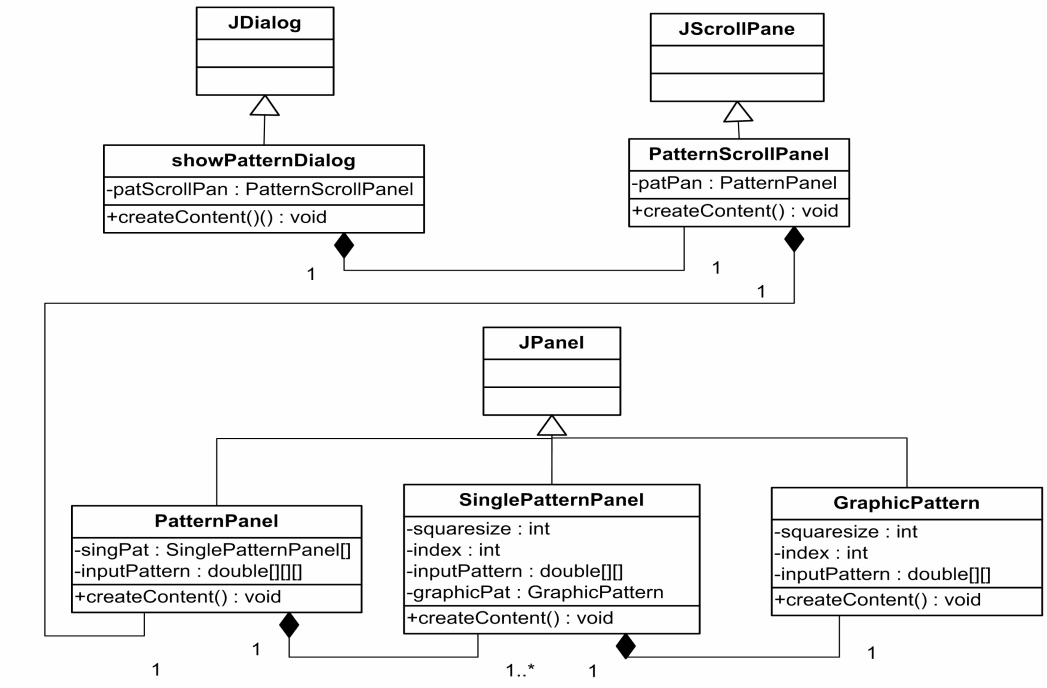


Abb. 32 Klassendiagramm der Klasse ShowPatternDialog

Um das Eingabemuster in seiner Länge und Breite korrekt darzustellen, werden die einzelnen reellen Werte in zwei Schleifen durchlaufen und die Anfangskoordinaten der einzelnen Quadrate, wie in Abbildung 32 dargestellt, festgelegt.

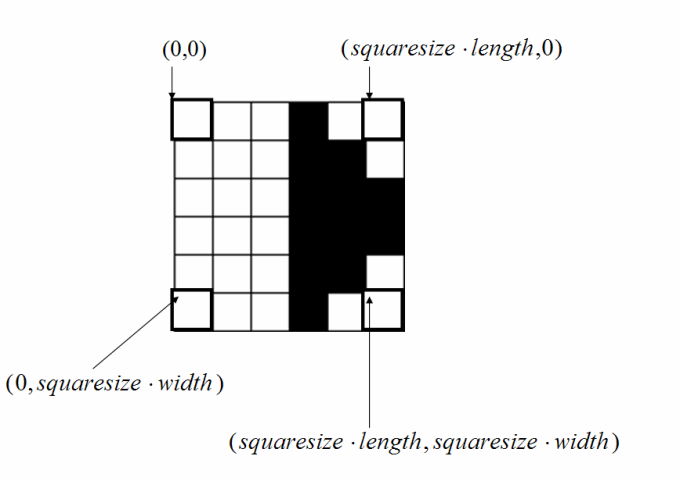


Abb. 32 Darstellung der Eingabemuster 1

Der Farbton, welches jedes einzelne Quadrat annimmt, hängt von dem reellen Zahlenwert des Eingabemusters an der darzustellenden Position ab und wird mit Hilfe der statischen Methode **`Color getColor(double inputvalue)`** der Klasse **`Tools`** ermittelt.

…
```
g.setColor(Tools.getColor(inputPattern[i][j]));
```
…

Diese statische Methode nutzt den Konstruktor der Klasse **`Color`**, indem sie bei Übergabe eines Eingabemusterwertes von 1.0 alle RGB-Werte auf 0.0 setzt, welches der Definition der Farbe Schwarz entspricht. Dementsprechend werden alle Grautöne bis hin zur Farbe Weiß, welche dem Eingabemusterwert von 0.0 bzw. den RGB-Werten von 1.0 entspricht, erzeugt. Mit Hilfe der Klasse **`SinglePatternPanel`** erhält jedes Eingabemuster einen Rahmen mit seinem jeweiligen Index und wird in einem Containerobjekt der Klasse **`PatternPanel`** aufgenommen.
Neben den beschriebenen Konfigurationselementen besitzt eine ConfigurationPanel Steuerungselemente in Form eines Start-, Pause- und Stoppbuttons, welche eine Simulation jeweils starten, anhalten und beenden. Nachdem eine Simulation vollständig konfiguriert wurde, ist nur der Startbutton auf aktiv gesetzt. Durch Anklicken dessen wird zunächst geprüft, ob alle Parameter der erstellten Simulation korrekt angegeben sind, d. h. es wird geprüft ob:

- alle Parameter syntaktisch und gemäß ihres Typs korrekt eingegeben sind,
- eine nicht leere Menge von Eingabemustern eingelesen wurde,
- die Struktur der Menge der eingelesenen Eingabemuster auf die angegebene Struktur der neuronalen Netze passt.

Für den Fall, dass eine dieser Prüfungen fehlschlägt, öffnet sich ein Fehlerdialog, der auf die Art des bestehenden Fehlers hinweist, und es wird keine Simulation gestartet. Falls alle konfigurierenden Daten korrekt eingegeben wurden, werden zunächst alle noch offenen Analysefenster geschlossen und die eigentliche Simulation erzeugt und gestartet, was zur Folge hat, dass der Startknopf inaktiv, Pause- und Stoppbutton aktiv werden. Um eine Simulation zu erzeugen, müssen sowohl das Simulationsschritttextfeld und die geparste Menge der Eingabemuster als auch die Textfelder mit den Konfigurationsdaten der einzelnen Simulationsumgebungen (siehe Unterpunkt EnvironmentScrollPanel) eingelesen werden, sodass die entsprechenden Werte an die Instanz der Klasse **`Simulation`** weitergegeben werden können.

```
…
Simulation.getInstance().getNumberOfAgents()[i] = agents;
Simulation.getInstance().getLearningRates()[i] = learningRate;
Simulation.getInstance().getMomenta()[i] = momentum;
Simulation.getInstance().getWordpositions()[i] = word;
Simulation.getInstance().getLaylengths()[i] = lengths;
Simulation.getInstance().getLaywidths()[i] = widths;
…
```

Bevor eine Simulation ausgeführt werden kann, müssen zuerst die einzelnen Simulationsumgebungen generiert und die für die grafische Darstellung des Avg1 und Avg2 erforderlichen Datensätze kreiert werden (siehe Unterpunkt ChartPanel):

```
…
Simulation.getInstance().generateEnvironments();
MainWindow.getInstance().createAvg1Dataset();
MainWindow.getInstance().createAvg2Dataset();
…
```

Das Starten einer Simulation erfolgt durch den Aufruf der Methode

```
…
Simulation.getInstance().simulationStart();
…
```

innerhalb eines eigenen Threads.
Ein laufender Thread kann durch Anklicken des Pausebuttons angehalten werden und es kann eine Analyse der einzelnen Simulationsumgebungen durchgeführt werden. Durch erneutes Anklicken schließen alle geöffneten Analysefenster und die Simulation wird fortgeführt.
Durch Anklicken des Stoppbuttons oder nach erfolgreichem Durchlauf durch alle Simulationsschritte wird der aktuelle Thread und die damit verbundene aktuell laufende Simulation beendet und sowohl der Pause- als auch der Stoppbutton wieder auf inaktiv gesetzt.

EnvironmentsScrollPanel

Die Aufgabe der EnvironmentScrollPanel ist es, sowohl eine Übersicht über die erstellten Simulationsumgebungen zu geben als auch eine Veränderbarkeit dieser im Einzelnen zu gewährleisten. Abbildung 33 zeigt den Aufbau dieses JScrollPanel in Form eines UML-Diagramms.

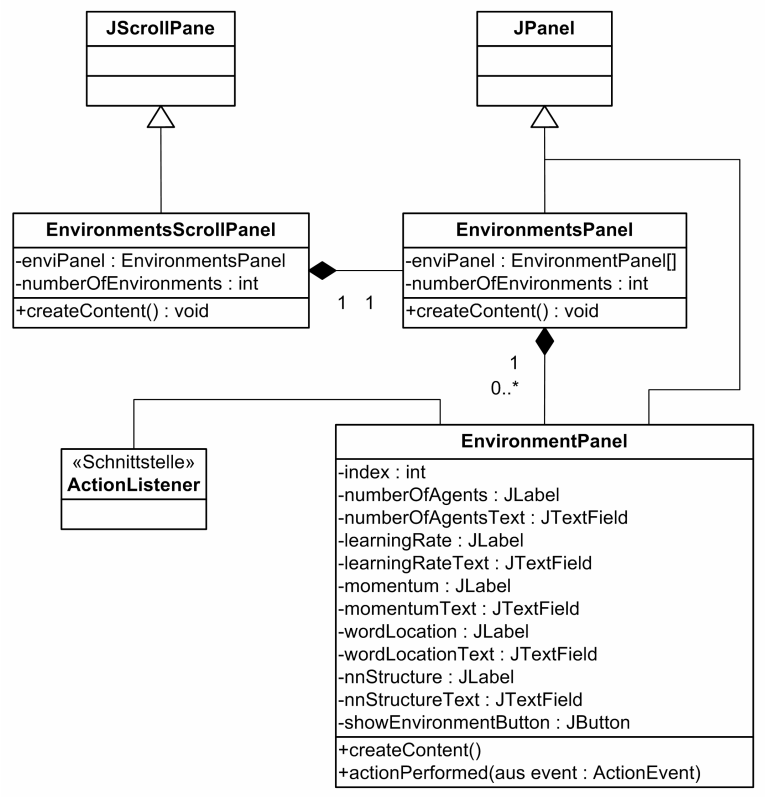


Abb. 33 Klassendiagramm der Klasse EnvironmentsScrollPanel

Jede einzelne Simulationsumgebung wird mit Hilfe eines Objekts der Klasse **`EnvironmentPanel`** dargestellt und als Array innerhalb einer Instanz der Container-Klasse **`EnvironmentsPanel`** abgelegt. Damit beliebig viele Simulationsumgebungen erstellt werden können, wird dieses EnvironmentsPanel einem scrollbaren Objekt der Klasse **`EnvironmentsScrollPanel`** zugeordnet. Jede Simulationsumgebung wird durch die Attribute Anzahl der Agenten, Lernrate, Momentum, Wortposition sowie Struktur des neuronalen Netzes der einzelnen Agenten bestimmt, wodurch jedes Objekt der Klasse für jedes genannte Attribut ein beschriftetes JTextField erhält. Der Benutzer kann somit jede Umgebung individuell konfigurieren. Die syntaktische und strukturelle Prüfung der einzelnen Eingaben erfolgt, wie bereits erwähnt, nach dem Drücken des Startbuttons.
Des Weiteren beinhaltet jedes EnvironmentPanel einen Analysebutton, welcher zu Beginn einer Simulation auf inaktiv gesetzt wird. Sobald ein Simulationsdurchlauf startet, wird dieser Button aktiv und es kann während einer laufenden Simulation eine Analyse einer bestimmten Umgebung durchgeführt werden. Nach Anklicken des Analysebuttons öffnet sich mit Hilfe des Aufrufs

```
…
MainWindow.getInstance().getEnviFrames()[Integer.parseInt(
                          e.getActionCommand())].createContent();
…
```

das Analysefenster der durch das **`ActionEvent e`** bestimmten Simulationsumgebung. Falls dieses Fenster bereits einmal erzeugt wurde, setzt der Aufruf

```
…
MainWindow.getInstance().getEnviFrames()[Integer.parseInt(
                              e.getActionCommand())].requestFocus();
…
```

das Analysefenster wieder in den Focus der grafischen Oberfläche. Hierbei ist zu beachten, dass die Klasse **`MainWindow`** ein Array von Instanzen der Klasse **`EnvironmentAnalysisFrame`**, welches nach dem Neuerstellen oder Laden einer Simulation erzeugt wurde, als Attribut enthält. Kapitel 4.2.2 zeigt, wie durch den Aufruf der Methode **`createContent()`** ein entsprechendes Analysefenster gezeichnet wird.

ChartPanel

Im Gegensatz zu den bis jetzt genannten Komponenten des Hauptfensters nutzt LexLearn zur grafischen Darstellung von Avg1 und Avg2 die vordefinierte Klasse **`ChartPanel`** des Frameworks JFreeChart. Ein Hauptfenster besitzt somit zwei Objekte dieser Klasse **`ChartPanel`**, welche wiederum für die Darstellung zweier Objekte der Klasse **`JFreeChart`**, einem **`chart1`** zur Darstellung des Avg1 und einem **`chart2`** zur Darstellung des Avg2 verantwortlich sind. Eine Instanz der Klasse **`JFreeChart`** wird durch den Aufruf von

```
…
chart1 = ChartFactory.createXYLineChart(
            ConstantStrings.AVG1, //Title
            ConstantStrings.SIMULATIONSTEPS, // x-axis Label
            ConstantStrings.AVG1, // y-axis Label
            avg1Dataset,// Dataset
            PlotOrientation.VERTICAL, // Plot Orientation
            true, // Show Legend
            true, // Use tooltips
            false // Configure chart to generate URLs
          );
…
```

erzeugt. Der Fehlerindikator Avg1 wird mit Hilfe eines Liniendiagramms mit dem Titel „Avg1“ dargestellt, wobei die X-Achse die Anzahl der Simulationsschritte und die Y-Achse den entsprechenden Avg1-Wert repräsentiert. Die zu zeichnenden Daten werden durch die **`XYSeriesCollection avg1Dataset`**, welche in der Klasse **`MainWindow`** als Attribut gehalten wird, hinzugefügt. Abbildung 34 zeigt die Abhängigkeiten der einzelnen Objekte.

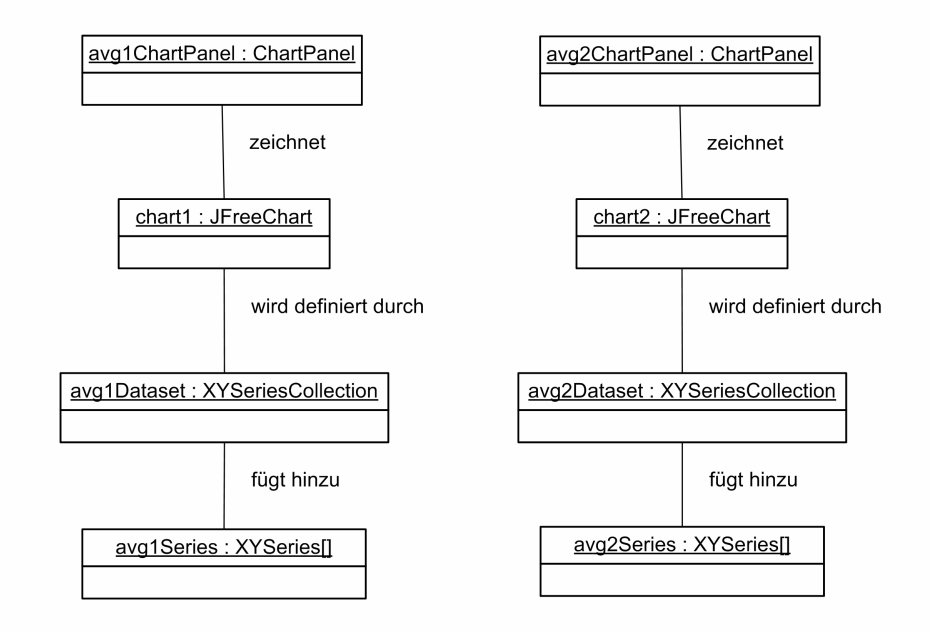


Abb. 34 Objekt-Beziehungsdiagramm ChartPanel

Ein **`avg1Dataset`** setzt sich dabei aus einem Array von **`XYSeries`** Objekten zusammen und wird durch den Aufruf der Methode **`createAvg1Dataset()`**der Klasse **`MainWindow`** gefüllt. Jedes einzelne **`XYSeries`** Objekt repräsentiert einen Liniengraphen für jeweils eine Simulationsumgebung. Der Name des jeweiligen Graphen, welcher sich aus „Environment“ und dem Index der Simulationsumgebung zusammensetzt, wird mit Hilfe des Konstruktoraufrufs

…
```
avg1Series[i] =
           new XYSeries(ConstantStrings.ENVIRONMENT+i+" ");
```
…

gesetzt und durch

…
```
avg1Dataset.addSeries(avg1Series[i]);
```
…

einem **`XYSeriesCollection`** Objekt hinzugefügt. Jede einzelne **`avg1Series`** enthält dabei die Punkte (x;y), deren Graph innerhalb eines Objektes der Klasse **`JFreeChart`** gezeichnet werden soll. Analog ist die Darstellung des Avg2 durch den **`avg2Dataset`** zu sehen.

Nach dem Start von LexLearn enthalten alle diese Objekte noch keine Daten und es wird kein Graph, jedoch das Koordinatensystem gezeichnet. Nach dem Erstellen oder Laden einer Simulation werden die Namen der einzelnen Graphen erzeugt und in den jeweiligen ChartPanels hinzugefügt.
Beginnt ein Simulationsdurchlauf durch Drücken des Startbuttons, so wird nach jedem tausendsten Simulationsschritt für das Zeichnen des Fehlerindikators Avg1 der `avg1Series` jeder Simulationsumgebung ein Punkt (aktueller Simulationsschritt; Wert des Avg1) und für das Zeichnen des Fehlerindikators Avg2 der `avg2Series` jeder Simulationsumgebung ein Punkt (aktueller

Simulationsschritt; Wert des Avg2) hinzugefügt. Durch Aufruf der Methode **`chart1.fireChartChanged()`** bzw. **`chart2.fireChartChanged()`** wird signalisiert, dass sich die Punktemenge eines Graphen geändert hat und er somit neu gezeichnet wird.

```
…
if(i % 1000 == 0)
{
      for(int j=0;j<environments.length;j++)
      {
            avg1Series[j].add(i,environments[j].calculateAvg1());
            avg2Series[j].add(i,environments[j].calculateAvg2());
      }
      chart1.fireChartChanged();
      cp1.setChart(chart1);
      chart2.fireChartChanged();
      cp2.setChart(chart2);
}
…
```

Mit Hilfe der Methode **`cp1.setChart(chart1)`** und **`cp2.setChart(chart1)`**werden die neu gezeichneten ChartPanels **`cp1`** und **`cp2`** in das Hauptfenster eingefügt.

NewSimulationDialog

Der NewSimulationDialog dient in LexLearn zur Erstellung einer neuen Simulation. Der Aufbau dieses Dialogfensters ähnelt dabei dem Aufbau einer Instanz der Klasse **`EnvironmentPanel`** und ist in Abbildung 35 als Klassendiagramm dargestellt.

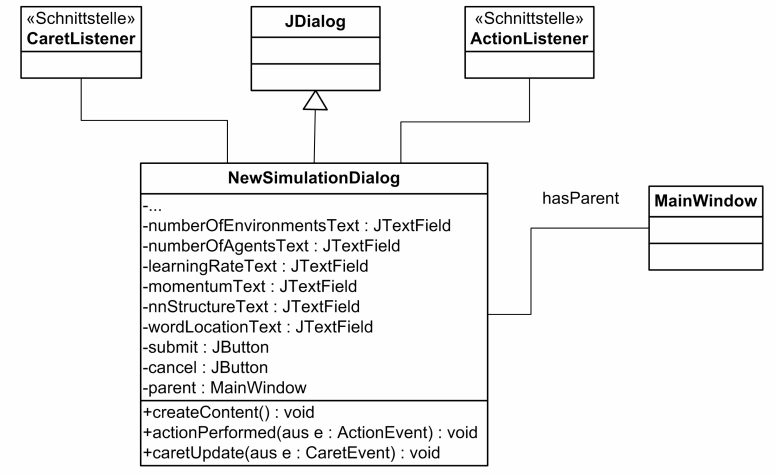


Abb. 35 Klassendiagramm der Klasse NewSimulationDialog

Ein Objekt der Klasse **`NewSimulationDialog`** wird nach Klicken des **`JMenuItem newSim`** innerhalb des Dateimenus (siehe Unterabschnitt Menuleiste) erzeugt und erhält über einen Konstruktoraufruf das aktuelle Hauptfenster als dialogauslösende Klasse. Die Angabe der Anzahl an Simulationsumgebungen über

das beschriftete **`JTextField numberOfEnvironmentsText`** legt fest, wie viele Simulationsumgebungen die neu erstellte Simulation besitzen soll. Des Weiteren müssen für die Anzahl der Agenten, die Lernrate, neuronale Netzstruktur und Platzierung des Wortes innerhalb der verborgenen Schichten Werte eingeben werden, welche als Attribute für jede Simulationsumgebung standardmäßig übernommen werden. Dieses Setzen der Attribute erfolgt nach Klicken des **`JButtons submit`** mit Hilfe der Schnittstelle **`ActionListener`** innerhalb der Methode **`actionPerformed(ActionEvent e)`**:

```
…
ArrayParser parse = new ArrayParser(nnStructureText.getText());

for (int i=0;i<numberOfEnvironments;i++)
{
learningRates[i] = Double.parseDouble(learningRateText.getText());
momenta[i] = Double.parseDouble(momentumText.getText());
numberOfAgents[i] = Integer.parseInt(numberOfAgentsText.getText());
wordLocations[i] = Integer.parseInt(wordLocationText.getText());
if (parse.successful())
      {
      layerLengths[i] = parse.getArrayLengths();
      layerWidths[i] = parse.getArrayWidths();
      }
}
…
…
Simulation.getInstance().setLearningRates(learningRates);
Simulation.getInstance().setMomenta(momenta);
Simulation.getInstance().setNumberOfAgents(numberOfAgents);
Simulation.getInstance().setWordpositions(wordLocations);
Simulation.getInstance().setLaylengths(layerLengths);
Simulation.getInstance().setLaywidths(layerWidths);
…
```

Das Einbinden der Schnittstelle **`CaretListener`** garantiert, dass der **`JButton submit`** erst aktiv wird, wenn alle Textfelder ausgefüllt wurden.
Nach dem Speichern der einzelnen Simulationsumgebungseigenschaften werden sowohl das EnvironmentsScrollPanel, in der die standardmäßigen Attribute einer Umgebung individuell geändert werden können, als auch die beiden ChartPanels des Hauptfensters erzeugt und gezeichnet. Das Zeichnen der ChartPanels erfolgt dabei gemäß dem in Unterabschnitt ChartPanel beschriebenen Prozess.

4.2.2 Aufbau Analysefenster

In LexLearn ist jedes Analysefenster eine Instanz der Klasse **`EnvironmentAnalysisFrame`**, welche in Abbildung 36 als Klassendiagramm dargestellt ist.

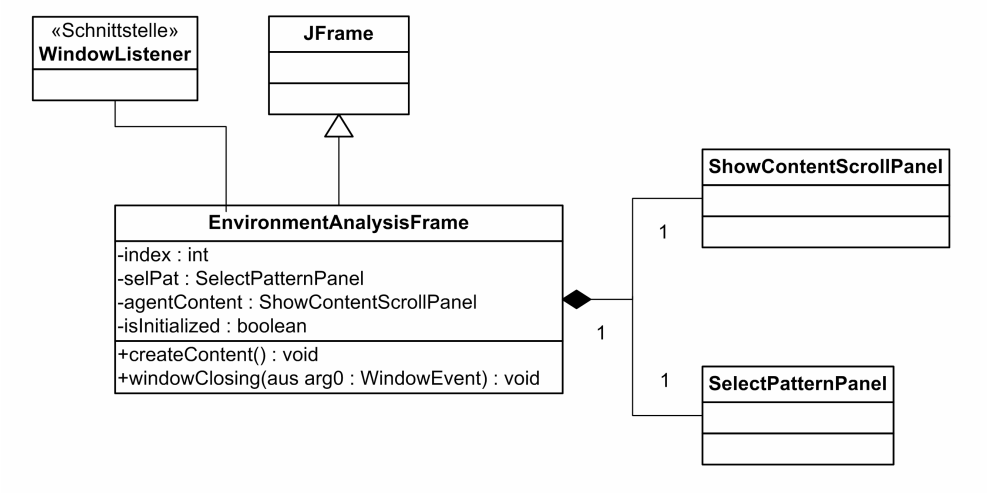


Abb. 36 Klassendiagramm der Klasse EnvironmentAnalysisFrame

Ein Objekt der Klasse **`EnvironmentAnalysisFrame`** enthält dabei als Attribute und Methoden:

- **`int index`**: gibt die Simulationsumgebung an, zu der ein Analysefenster gezeichnet werden soll;
- **`SelectPatternPanel selPat`**: enthält das für die Konfiguration der Analyse notwendige Objekt der Klasse **`SelectPatternPanel;`**
- **`ShowContentScrollPanel agentContent`**: enthält die visuelle Auflistung einer Simulationsumgebung in Form einer Instanz der Klasse **`ShowContentScrollPanel;`**
- **`boolean isInitialized`**: gibt an, ob ein Fenster aktuell schon gezeichnet ist und somit eventuell nur wieder in den Fokus des Displays gestellt werden muss;
- **`void createContent()`**: zeichnet das Analysefenster der im **`index`** angegeben Simulationsumgebung;
- **`void windowClosing(WindowEvent arg0)`**: setzt beim Schließen des Fensters das Attribut **`isInitialized`** auf **`false`** und signalisiert dadurch, dass dieses Fenster aktuell nicht gezeichnet ist.

Wie in Abschnitt 4.2.1 beschrieben, wird nach dem Laden oder Erstellen einer Simulation für jede Simulationsumgebung durch den Konstruktoraufruf von

```
…
EnvironmentAnalysisFrame[] temp = new EnvironmentAnalysisFrame
[Simulation.getInstance(). getEnvironmentcount()];

      for (int i=0;i<temp.length;i++)
      {
            temp[i] = new EnvironmentAnalysisFrame(i);
      }
MainWindow.getInstance().setEnviFrames(temp);

…
```

ein Analysefenster erstellt und mit Hilfe eines Arrays in der Klasse **`MainWindow`** abgelegt. Durch Anklicken des Analysebuttons innerhalb einer Simulationsumgebung *i* wird das entsprechende Analysefenster an der Stelle *i*

durch den Aufruf der Methode **`createContent()`** neu gezeichnet. Der Aufbau eines Analysefenster teilt sich gemäß dem in Abschnitt 3.3.2 beschriebenen Entwurf in zwei Komponenten auf: einem Objekt der Klasse **`SelectPatternPanel`** und der Klasse **`ShowContentScrollPanel`**.

SelectPatternPanel

Eine Instanz der Klasse **`SelectPatternPanel`** bietet Benutzer zum einen die Möglichkeit, ein spezielles Eingabemuster auszuwählen, um die Wörter bzw. Deduktionen der einzelnen Agenten zu diesem Muster zu untersuchen. Zum anderen kann der Benutzer zwischen verschiedenen Darstellungsformen der Wörter bzw. Abzeichnungen wählen. Abbildung 37 zeigt das zugehörige Klassendiagramm.

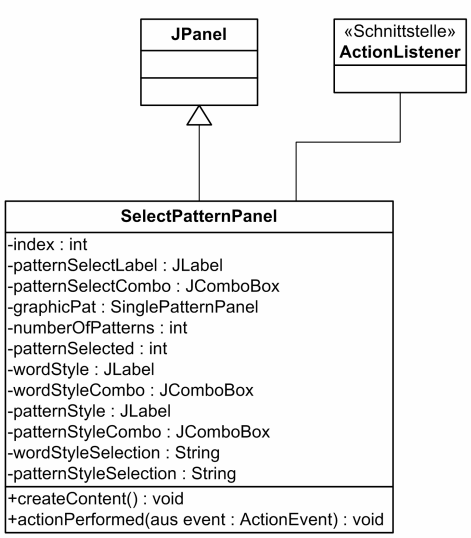


Abb. 37 Klassendiagramm der Klasse SelectPatternPanel

Die wichtigsten Attribute bzw. Methoden und deren Funktionalitäten im Überblick:

- **`int index`**: beinhaltet den Index der Simulationsumgebung
- **`JComboBox patternSelectCombo:`** eine Combobox zur Auswahl eines Eingabemusters (die Menge der Muster ist durch die Klasse **`Simulation`** eindeutig bestimmt)
- **`SinglePatternPanel graphPat`**: bietet eine grafische Darstellung des in der Combobox **`patternSelectCombo`** aktuell ausgewählten Eingabemusters (siehe Abschnitt 4.2.1). Über die **`ActionListener`** Schnittstelle wird ein neu ausgewähltes Eingabemuster direkt neu gezeichnet.
- **`JComboBox wordStyleCombo`**: eine Combobox zur Auswahl von verschiedenen Darstellungsmöglichkeiten der Wörter der Agenten. Sie umfasst die Punkte „standard“, „rounded“ und „literal“. Je nach Auswahl werden über die Schnittstelle **`ActionListener`** die Wörter der Agenten

innerhalb des ShowContentScrollPanel neu dargestellt (siehe Unterpunkt ShowContentScrollPanel).

- **`JComboBox patternStyleCombo`**: eine Combobox zur Auswahl von verschiedenen Darstellungsmöglichkeiten der Deduktionen oder Abzeichnungen der Agenten. Sie umfasst die Punkte „standard“, „rounded“ und „graphical“. Je nach Auswahl werden über die Schnittstelle **`ActionListener`** die Deduktionen der Agenten innerhalb des ShowContentScrollPanel neu dargestellt (siehe Unterpunkt ShowContentScrollPanel).
- **`void createContent()`**: diese Methode ordnet die oben beschriebenen JKomponenten mit Hilfe des LayoutManagers **`GridBagLayout`** an und zeichnet sie.
- **`void actionPerformed(ActionEvent event)`**: innerhalb dieser überschrieben Methode der Schnittstelle **`ActionListener`** unterscheidet das Fenster, welche Komponente des ShowContentScrollPanel **`scsp`** durch den Aufruf von

  …
  ```
  scsp.createContent(patternSelected,wordStyleSelection,
                     patternStyleSelection);
  ```
  …

  neu und in welcher Weise gezeichnet werden muss.

Abbildung 38 gibt eine Übersicht über die verschiedenen Darstellungsmöglichkeiten der Wörter bzw. Deduktionen einzelner Agenten.

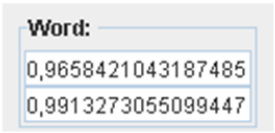
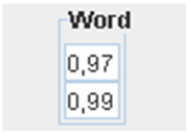
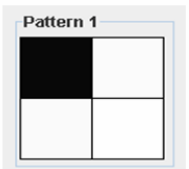
Auswahl Combobox	Beschreibung	Screenshot
standard	Aktivierungswerte der Neuronen der Wort- / Ausgabeschicht auf 16 Nachkommstellen gerundet	
rounded	Aktivierungswerte der Neuronen der Wort- / Ausgabeschicht auf zwei Nachkommstellen gerundet	
literal	Aktivierungswerte der Neuronen der Wortschicht auf Silben abgebildet	
graphical	Aktivierungswerte der Neuronen der Ausgabeschicht grafisch dargestellt	

Abb. 38 Darstellungsformen der Wörter und Deduktionen 1

ShowContentScrollPanel

Die Klasse **`ShowContentScrollPanel`** repräsentiert die zweite Komponente des Analysefensters und ist für die Auflistung der Wörter und Deduktionen für ein bestimmtes Eingabemuster aller Agenten innerhalb einer Simulationsumgebung verantwortlich. Ihr Aufbau, welcher dem Aufbau des EnvironmentScrollPanel ähnelt, ist in Abbildung 39 in Form eines UML-Diagramms dargestellt.

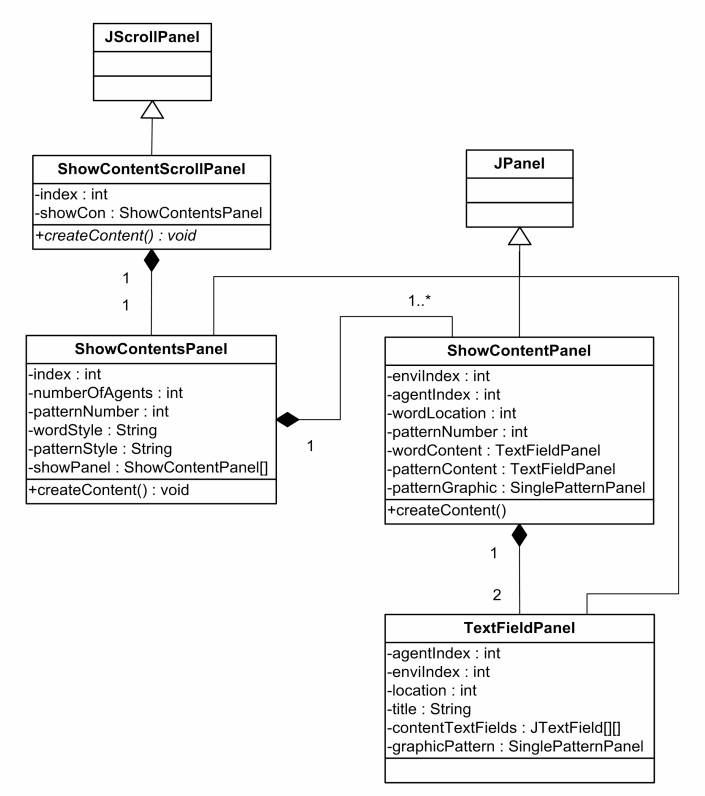


Abb. 39 Klassendiagramm der Klasse ShowContentScrollPanel

Ein Objekt der Klasse **`ShowContentScrollPanel`** gibt einen Rahmen für eine Instanz der Klasse **`ShowContentsPanel`** vor, welche die Wörter und Deduktionen eines jedes Agenten in Form eines Array von **`ShowContentPanel`** Objekten enthält. Neben diesem Array besitzt die Klasse **`ShowContentsPanel`** als weitere Attribute: den Index der zu analysierenden Simulationsumgebung, dessen Anzahl an Agenten, die Nummer des ausgewählten Eingabemusters **`patternNumber`** und Angaben über die Darstellungsform der Wörter bzw. Deduktionen in Form des Strings **`wordStyle`** bzw. **`patternStyle`**, welche durch den Konstruktor der Klasse

```
…
public ShowContentsPanel(int index, int patternNumber,String
wordStyle, String patternStyle)
{
  this.index = index;
  this.numberOfAgents =
         Simulation.getInstance().getNumberOfAgents()[index];
```

```
  this.patternNumber = patternNumber;
  this.wordStyle=wordStyle;
  this.patternStyle=patternStyle;
  createContent(patternNumber,wordStyle,patternStyle);
}
…
```

gesetzt werden. Durch den Aufruf der Methode **`createContent( patternNumber,wordStyle,patternStyle)`** innerhalb des Konstruktors wird dieses Panel gezeichnet. Jeder Agent einer Simulationsumgebung erhält dabei ein Objekt der Klasse **`ShowContentPanel`**, welche mit Hilfe des Layoutmanagers **`GridLayout`** innerhalb ShowContentsPanels angeordnet werden:

```
…
GridLayout myGridLayout = new GridLayout(0,1);
myGridLayout.setVgap(20);
setLayout(myGridLayout);

showPanel = new ShowContentPanel[numberOfAgents];
for (int i=0;i<numberOfAgents;i++)
{
  showPanel[i] = new ShowContentPanel(index,i);
  showPanel[i].createContent(patternNumber,wordStyle,patternStyle);
  add(showPanel[i]);
}
…
```

Der Inhalt des ShowContentPanel richtet sich dabei nach der aktuell ausgewählten visuellen Szene und der Darstellungsform des Wortes bzw. der Deduktion. Wurde „standard“, „rounded“ oder „literal“ als Darstellungsform für das Wort eines Agenten bzw. „standard“ oder „rounded“ als Darstellungsform für die Deduktion ausgewählt, erfolgt deren Darstellung in Form zweier Objekte der Klasse **`TextFieldPanel`**.

```
…
wordContent = new   TextFieldPanel(enviIndex,agentIndex,
      Simulation.getInstance().getWordpositions()[enviIndex],
      ConstantStrings.WORD);
wordContent.createContent(patternNumber,wordStyle);
…
patternContent = new TextFieldPanel(enviIndex,agentIndex,
      Simulation.getInstance().getEnvironments()[enviIndex]
      .getAgents()[agentIndex].getLayers().length-1,
      ConstantStrings.DRAWING);
patternContent.createContent(patternNumber,patternStyle);
…
```

Ein Objekt der Klasse **`TextFieldPanel`** besteht aus einem zweidimensionalen Array von JTextfeldern, in denen die Aktivierungen der Neuronen der Wortschicht bzw. der Ausgabeschicht innerhalb dieser Textfelder ausgegeben werden (siehe Abbildung 38).

Falls für die Darstellungsform des Wortes „literal“ ausgewählt wurde, erfolgt die Darstellung innerhalb eines Strings, indem die Aktivierungen der Neuronen der Wortschicht innerhalb einer Schleife, mit Hilfe der statischen Methode **`getSyllable(double in)`** der Klasse **`Tools`**, auf konkrete Silben abgebildet werden. Eine Veränderung dieser Abbildung auf Silben kann der Benutzer in der aktuellen Version nur im Quellcode durchführen.

```
…
public static String getSyllable (double in)
{
      String out = new String("");

      if (in < 0.1)  out += "al"; else
      if (in < 0.2)  out += "as"; else
      if (in < 0.3)  out += "es"; else
      if (in < 0.4)  out += "el"; else
      if (in < 0.5)  out += "il"; else
      if (in < 0.6)  out += "is"; else
      if (in < 0.7)  out += "os"; else
      if (in < 0.8)  out += "ol"; else
      if (in < 0.9)  out += "ul"; else
      if (in < 1.0)  out += "us";

      return out;
}
…
```

Wählt ein Benutzer als Darstellungsform der Deduktion „graphical“, so erfolgt die Darstellung in Form der in Abschnitt 4.2.1 beschriebenen graphischen Zeichnung der Aktivierungen der Ausgabeschicht innerhalb der Klasse **`ShowContentPanel`**.

```
…
patternGraphic = newSinglePatternPanel(pattern,squaresize,
                                       patternNumber);
…
```

5. Simulationsdurchführung

In diesem Kapitel werden mit Hilfe des geschaffenen Werkzeugs LexLearn sowohl die von Hutchins und Hazlehurst beschrieben Simulationen [HH95, S.167-177] als auch eigenständig erstellte Simulationen durchgeführt. Die daraus resultierenden Ergebnisse, insbesondere die Entwicklung der Fehlerindikatoren Avg1 und Avg2, werden mit den von Hutchins und Hazlehurst eruierten Ergebnissen verglichen und im Falle der neu erstellten Simulation interpretiert. In jedem der Unterkapitel werden die Attribute einer Simulation durch eine Instanzierung der folgenden Variablen und Funktionen festgelegt:

Visuelle Szenen:

- m = Anzahl der visuellen Szenen
- S = Menge der visuellen Szenen $\{ s_1, s_2, \ldots, s_m \}$.

Agenten:

- n = Anzahl der Agenten innerhalb einer Gemeinschaft
- f_{arch} = Funktion, welche die Netzstruktur jedes Agenten bestimmt
- W = Menge der initialen Verbindungsgewichtungen des neuronalen Netzes eines jeden einzelnen Agenten
- μ = Lernrate
- ψ = Momentum

Interaktionsprotokoll:

- f_{pop} = Kontrollfunktion der Agentenpopulation
- f_{scene} = Funktion zur Auswahl einer visuellen Szene
- f_{ind} = Funktion zur Auswahl eines Agenten

Die in Abschnitt 5.1 und 5.2 beschriebenen Simulationen wurden unter Verwendung von LexLearn mit den von Hutchins und Hazlehurst festgelegten Simulationsparametern durchgeführt. Erste Simulationsdurchläufe brachten die Erkenntnis, dass LexLearn die von Hutchins und Hazlehurst propagierten Ergebnisse nur mit größerer Anzahl an Simulationsschritten bestätigen konnte. Nach einer Änderung der Lernrate von $\mu = 0.075$ auf $\mu = 0.75$ trat dieser Effekt nicht weiter auf. Aus diesem Grund sind die durch LexLearn erstellten Ergebnisse alle mit einer Lernrate von $\mu = 0.75$ erzielt worden.

5.1 Simulation 1

Hutchins und Hazlehurst versuchen durch die in diesem Unterkapitel beschriebene Menge von Simulationsdurchläufen die Eigenschaften *einfacherer* Systeme, die sich insbesondere durch eine kleine, einfach strukturierte Menge von Eingabemustern und die damit verbundene Simplizität der neuronalen Netze aller Agenten auszeichnen, zu untersuchen.

5.1.1 Simulationsparameter

Visuelle Szenen:

- $m = 4$
- S = { ((1,0),(0,0)),
  ((0,1),(0,0)),
  ((0,0),(1,0)),
  ((0,0),(0,1)) }

Agenten:

- $n \in \{5, 10, 15\}$
- f_{arch} = Ein- und Ausgabeschicht mit 4 Neuronen, eine verborgene Schicht mit 2 Neuronen (siehe Abbildung 40)
- W = Menge von zufälligen reellen Zahlen zwischen -0.5 und +0.5
- μ = 0.075 (Hutchins und Hazlehurst), 0.75 (LexLearn)
- ψ = 0.9

Interaktionsprotokoll:

- f_{pop} = die Agenten existieren vom Start bis zum Ende eines Simulationsdurchlaufs; es werden keine neuen Agenten erzeugt
- f_{scene} = zufällige Auswahl einer visuellen Szene aus der Menge S
- f_{ind} = zufällige Auswahl zweier Individuen aus der Gemeinschaft der Agenten

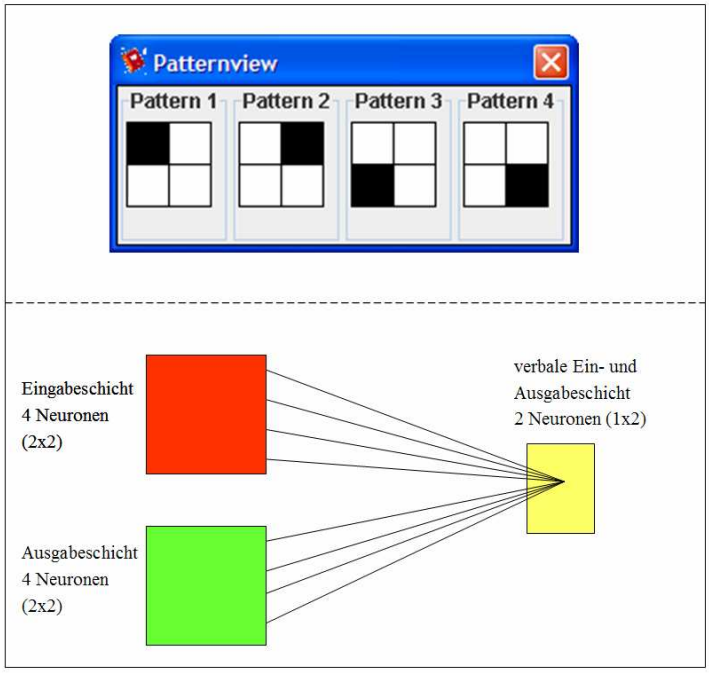


Abb. 40 Eingabemuster (oben) und neuronale Netzstruktur (unten) in Simulation 1

5.1.2 Durchführung und Analyse der Simulation

Simulationsdurchlauf mit 5 Agenten

Eine erste Simulationsdurchführung untersucht die Emergenz eines gemeinsam genutzten Lexikons innerhalb einer Gemeinschaft von 5 Agenten. Hutchins und Hazlehurst verwenden zwei Varianten, um zu verdeutlichen, dass die entstandenen Wörter der einzelnen Agenten die in Abschnitt 2.4.1 erläuterten Bedingungen an ein Lexikon erfüllen.
Die erste Variante stellt die Entwicklung der Aktivierungen der Wortschicht-Neuronen innerhalb eines zwei-dimensionalen Diagramms, dessen X-Achse dem Aktivierungswert des ersten Neurons und dessen Y-Achse dem Aktivierungswert des zweiten Neurons entspricht, gegenüber. Der Verlauf jedes einzelnen Graphen repräsentiert dabei die Evolution des Wortes eines Agenten für eine bestimmte visuelle Szene. Alle Graphen beginnen dabei fast in der Mitte des Koordinatenabschnitts in der Nähe des Punktes (0,5; 0,5), da die neuronalen Netze der einzelnen Individuen noch mit zufällig ausgewählten, unveränderten Verbindungsgewichtungen ausgestattet sind. Abbildung 42 zeigt die Ergebnisse dieser von Hutchins und Hazlehurst durchgeführten Simulation, während Abbildung 43 die Ergebnisse des Simulationsdurchlaufs mit den gleichen Parametern unter Verwendung des Werkzeugs LexLearn zeigt. Jedes Diagramm stellt dabei die Evolution eines Wortes für eine der vier verschiedenen visuellen Szenen dar. In beiden Fällen ist deutlich zu erkennen, dass die Graphen der einzelnen Agenten in allen vier Diagrammen jeweils in einer unterschiedlichen Ecke enden, was der größtmöglichen Differenz zwischen den Wörtern für die vier

visuellen Szenen und somit dem Erfüllen der ersten Bedingung an ein Lexikon entspricht. Die Tatsache, dass die Graphen innerhalb eines Diagramms alle in die gleiche Ecke laufen und somit alle Agenten das gleiche Wort für eine Szene gebildet haben, kommt der Erfüllung der zweiten Bedingung an ein gemeinsam genutztes Lexikon gleich.
Die Ergebnisse des Simulationsdurchlaufs der verschiedenen Implementierungen werden in Abbildung 41 dargestellt.

Index	Visuelle Szene	Wort Hutchins & Hazlehurst	Wort LexLearn	Wort LexLearn (literal)
a / 1	(1, 0, 0, 0)	(1, 1)	(1, 1)	usus
b / 2	(0, 1, 0, 0)	(1, 0)	(1, 0)	usal
c / 3	(0, 0, 1, 0)	(0, 1)	(0, 0)	alal
d / 4	(0, 0, 0, 1)	(0, 0)	(0, 1)	alus

Abb. 41 Entstandene Wörter nach Simulationsdurchlauf mit 5 Agenten

Die Zuordnung einer visuellen Szene zu einem Wort erfolgt aufgrund der zufällig gewählten Initialbelegung der neuronalen Netze nach keinem festen Schema. So entstehen, wie in Abbildung 41 ersichtlich, für die dritte und vierte visuelle Szene vertauschte Aktivierungen innerhalb der Neuronen der Wortschicht.

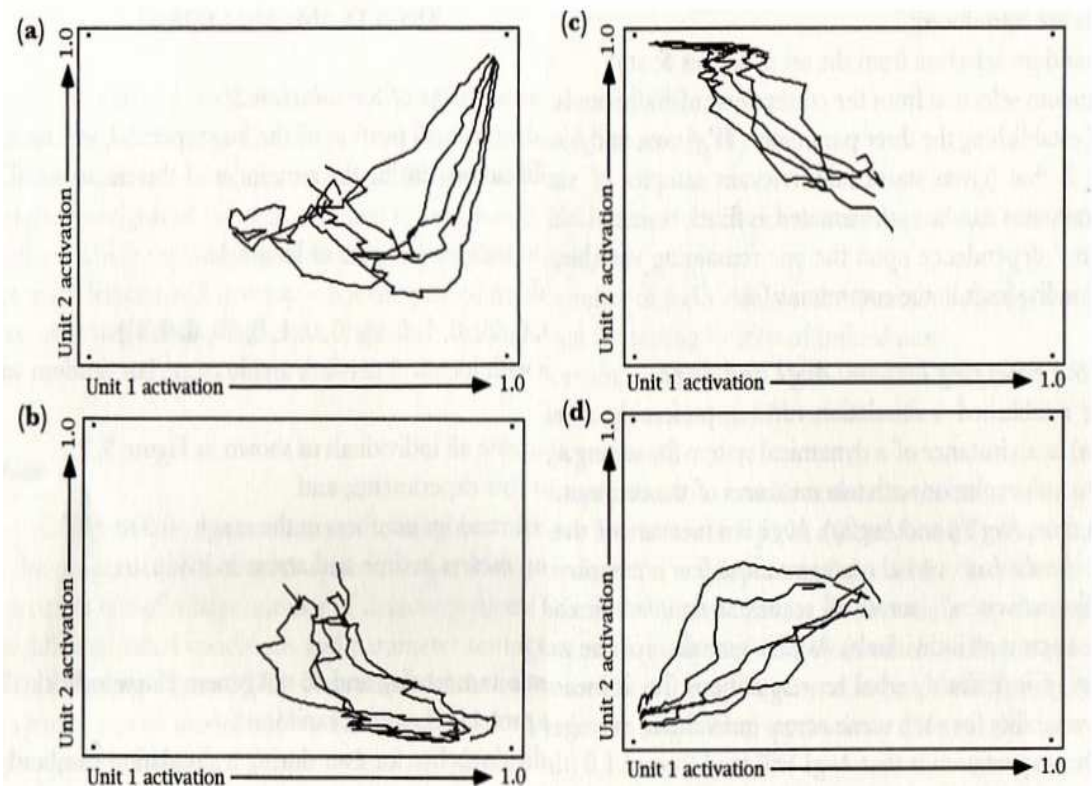


Abb. 42 Ergebnisse Simulation 1
Quelle: [HH95, S. 172, Figure 9.9]

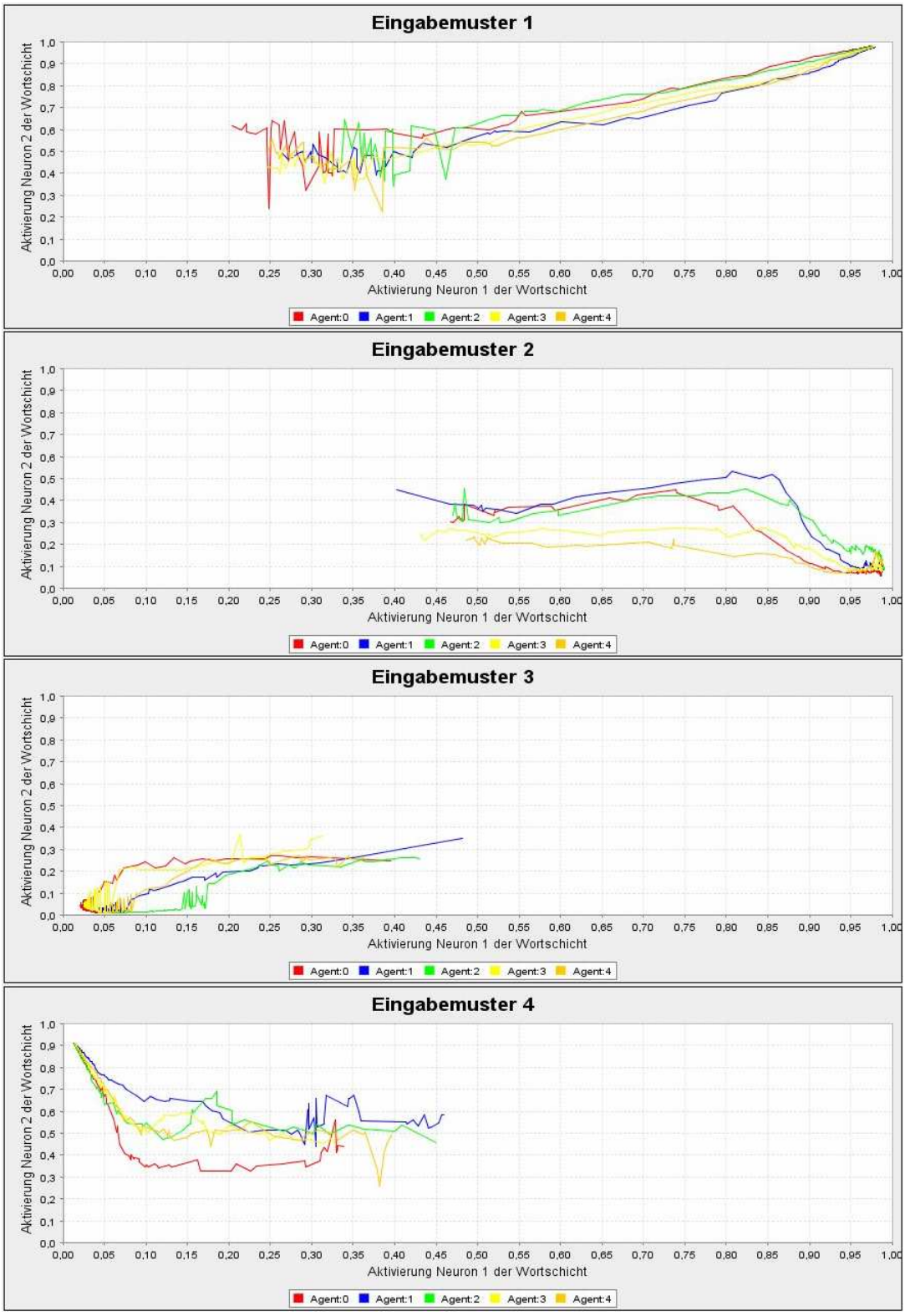


Abb. 43 LexLearn Ergebnisse Simulation 1

Die zweite Variante stellt den Verlauf von Avg1 und Avg2 während eines Simulationsdurchlaufs in Form zweier Liniendiagramme dar. LexLearn nutzt diese Variante zur Darstellung und Analyse der Emergenz eines Lexikons. Der Fehlerindikator Avg1 stellt, wie in Abschnitt 2.4.2 beschrieben, die durchschnittliche Differenz der Wörter für alle visuellen Szenen eines Agenten dar und sollte zu dem Wert 1.0, der größtmöglichen Differenz zwischen den Wörtern, konvergieren. Die Unterschiedlichkeit der Wörter aller Agenten für eine bestimmte visuelle Szene, welche durch den ebenfalls in Abschnitt 2.4.2 beschriebenen Fehlerindikator Avg2 ausgedrückt werden, sollte gegen den Wert 0.0, der minimalsten Differenz, tendieren. Abbildung 44 und 45 stellen den Verlauf von Avg1 und Avg2 innerhalb eines Simulationsdurchlaufs mit 5 Agenten in der Implementierung von Hutchins und Hazlehurst und unter Verwendung von LexLearn gegenüber.

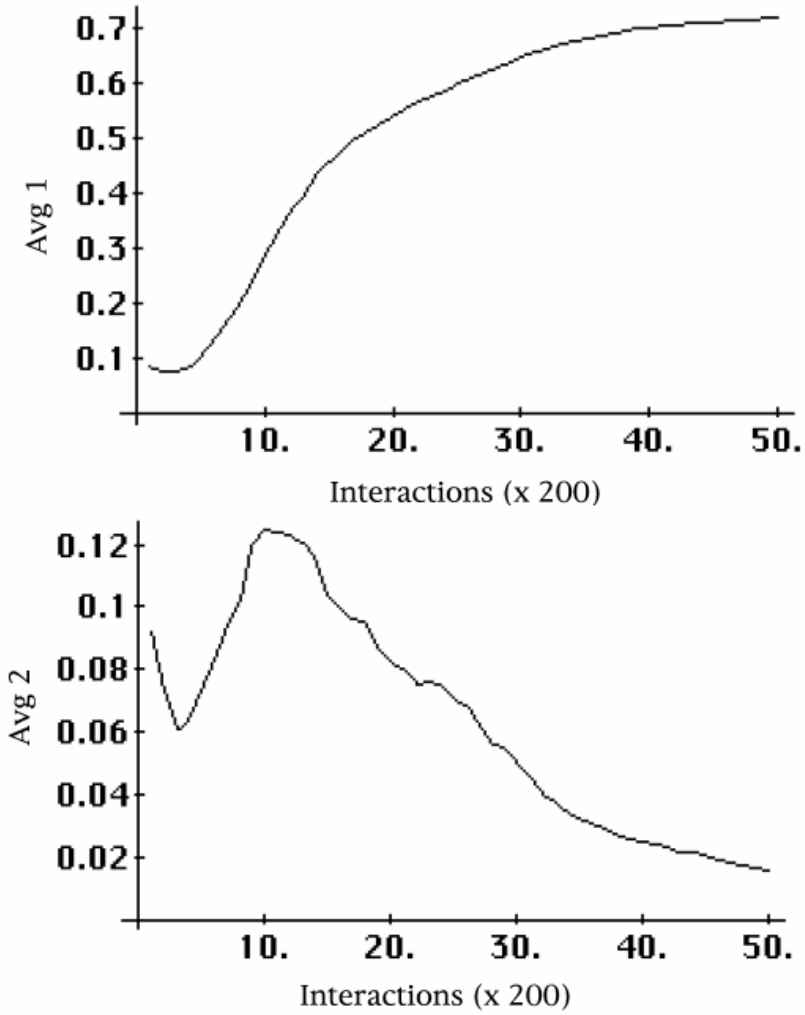


Abb. 44 Verlauf von Avg1 und Avg2 in der Implementierung von Hutchins und Hazlehurst Quelle: [HH95, S. 173, Figure 9.10]

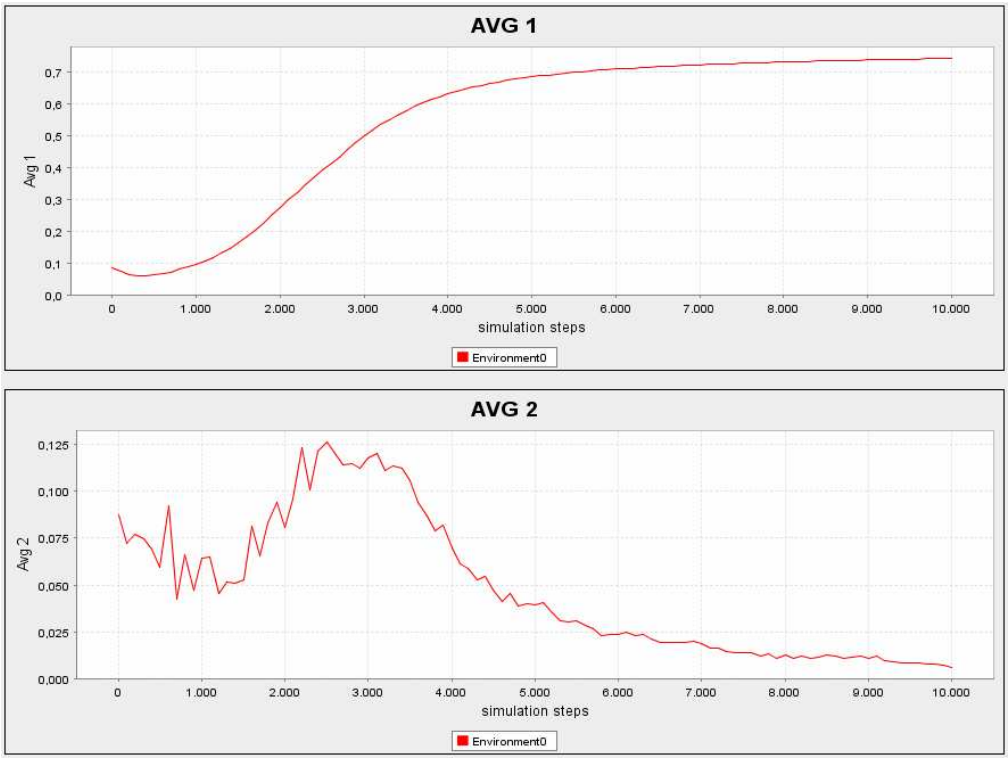


Abb. 45 Verlauf von Avg1 und Avg2 unter der Verwendung von LexLearn

Aus der Abbildung 44 und 45 geht hervor, dass sich der Verlauf der beiden Fehlerindikatoren Avg1 und Avg2 in beiden Implementierungen des Modells von Hutchins und Hazlehurst ähnelt. Aufgrund der zufällig gewählten Verbindungsgewichtungen besteht zu Beginn der Simulation keine große Differenz zwischen den einzelnen Wörtern der Agenten, was einem niedrigen Avg1-Wert entspricht. Im Gegensatz dazu stimmen die Wörter der Agenten für eine einzelne visuelle Szene nahezu überein, welches den geringen Avg2-Wert zur Folge hat. Beginnen die Agenten ihre Wörter zu differenzieren, so steigen Avg1 und Avg2 an. Die Wörter der Agenten für eine visuelle Szene differieren nun stärker. Ist die größtmögliche Differenz zwischen den Wörtern eines Agenten erreicht, beginnen die Agenten sich auf ein gemeinsames Lexikon zu einigen, welches ein Absinken des Avg2-Wertes zur Folge hat.

Simulationsdurchlauf mit unterschiedlicher Agentenanzahl

In den folgenden Simulationsdurchläufen wird untersucht, welche Auswirkungen eine veränderte Anzahl von Agenten innerhalb einer Gemeinschaft auf den Verlauf der Fehlerindikatoren Avg1 und Avg2 hat. Wie in Abbildung 46 bis 51 ersichtlich, werden drei verschiedene Durchläufe mit jeweils $n = 5$, $n = 10$ und $n = 15$ Agenten simuliert und die Ergebnisse von Hutchins und Hazlehurst mit den unter Verwendung von LexLearn erstellten Ergebnissen verglichen. Um allgemeine Aussagen über die Emergenz eines Lexikons treffen zu können, besteht jeder Durchlauf aus 15 Simulationsinstanzen.

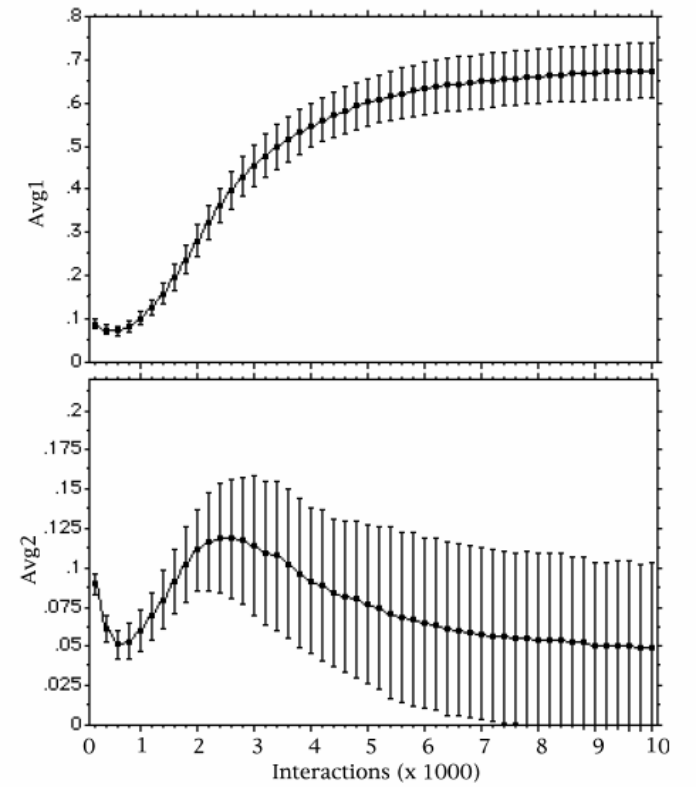


Abb. 46 Hutchins & Hazlehurst: Simulationsdurchlauf mit 15 Simulationsinstanzen und einer Gemeinschaft von 5 Agenten
Quelle: [HH95, S. 174, Figure 9.11.a]

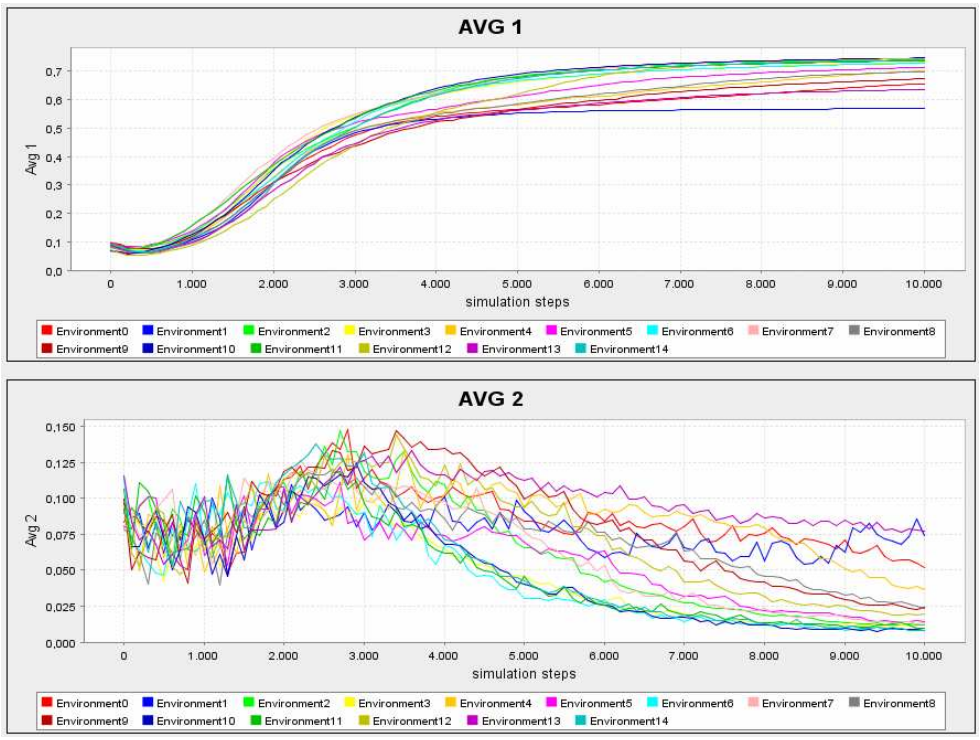


Abb. 47 LexLearn: Simulationsdurchlauf mit 15 Simulationsinstanzen und einer Gemeinschaft von 5 Agenten

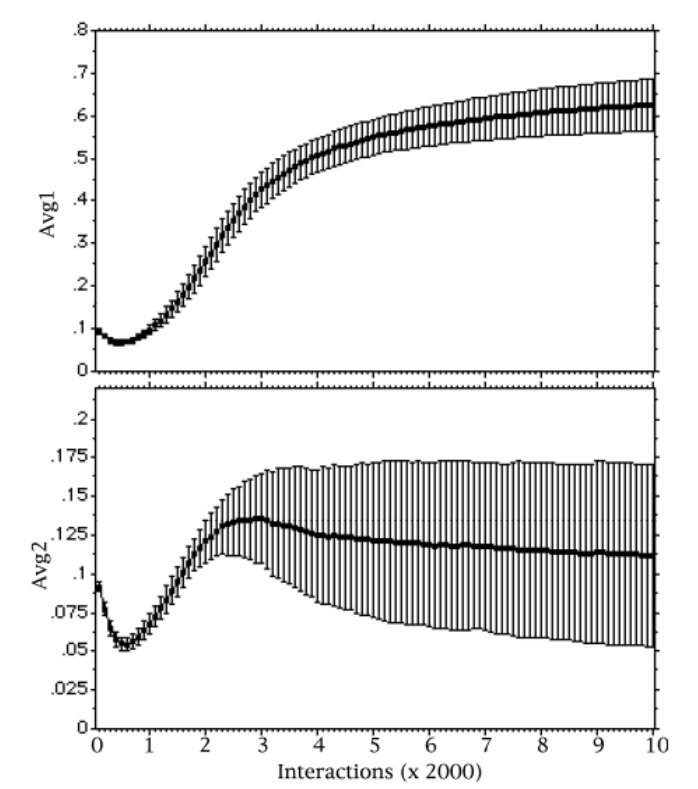


Abb. 48 Hutchins & Hazlehurst: Simulationsdurchlauf mit 15 Simulationsinstanzen und einer Gemeinschaft von 10 Agenten
Quelle: [HH95, S. 175, Figure 9.11.b]

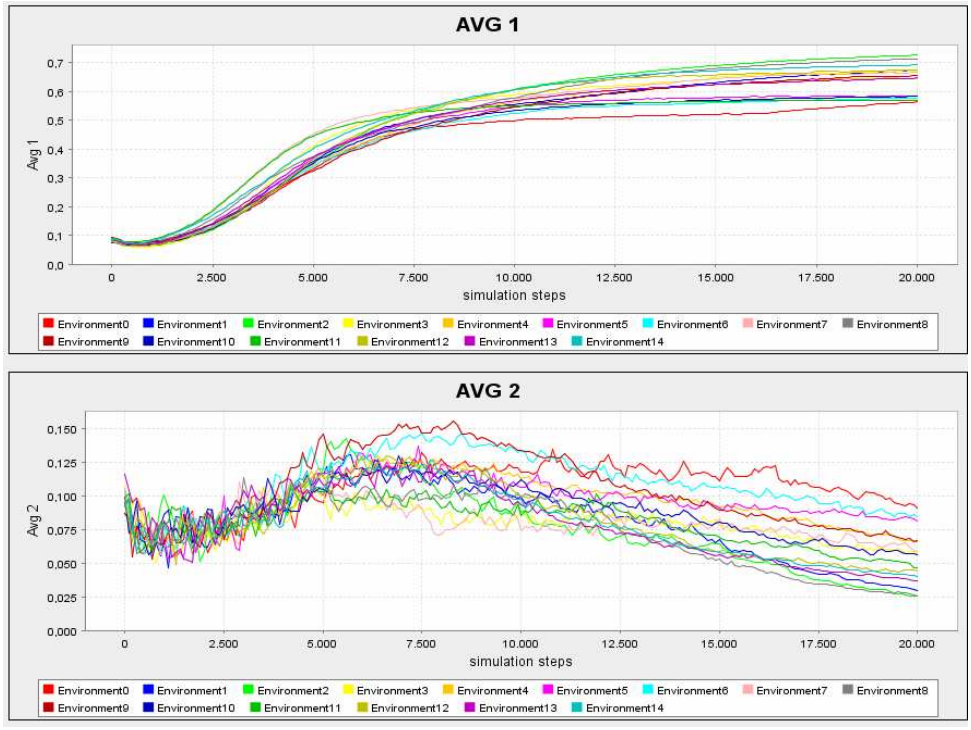


Abb. 49 LexLearn: Simulationsdurchlauf mit 15 Simulationsinstanzen und einer Gemeinschaft von 10 Agenten

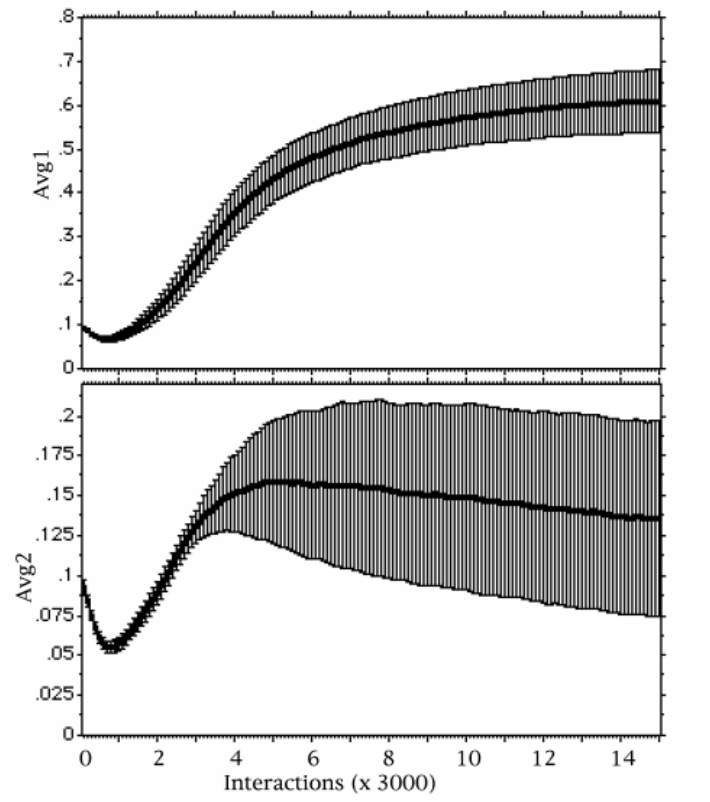


Abb. 50 Hutchins & Hazlehurst: Simulationsdurchlauf mit 15 Simulationsinstanzen und einer Gemeinschaft von 15 Agenten
Quelle: [HH95, S. 175, Figure 9.11.c]

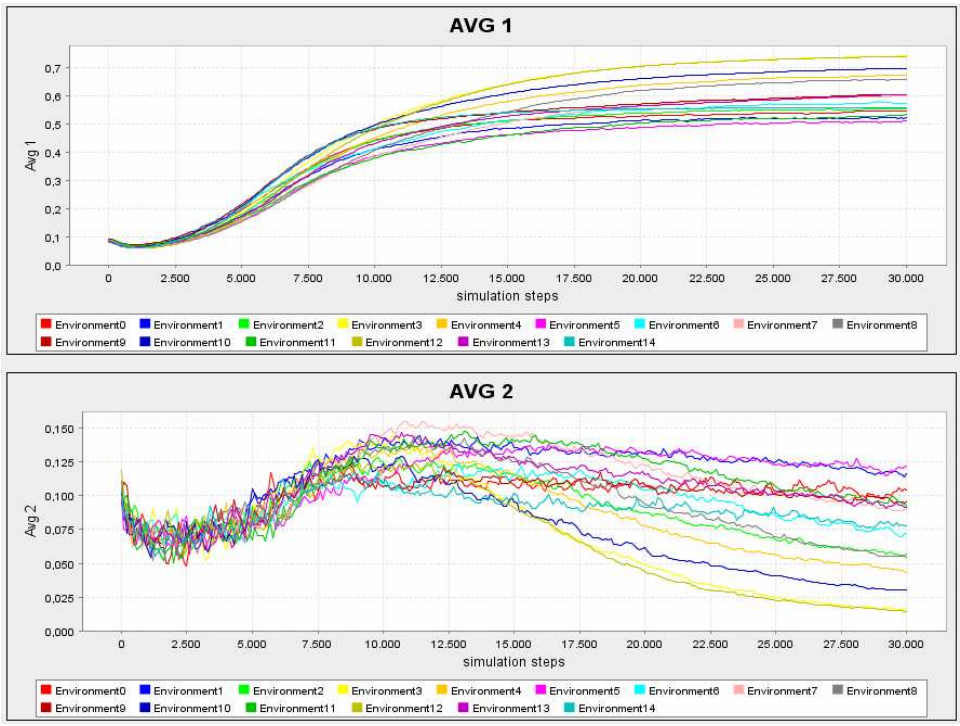


Abb. 51 LexLearn: Simulationsdurchlauf mit 15 Simulationsinstanzen und einer Gemeinschaft von 10 Agenten

In jeder der in Abbildung 46 bis 51 dargestellten Simulationsdurchläufe interagiert jeder Agent durchschnittlich 2000 Mal sowohl in der Rolle des Sprechers als auch in der Rolle des Zuhörers (vgl. Abschnitt 2.4.3). Dadurch erhöht sich die Anzahl der Simulationsschritte mit zunehmender Gemeinschaftsgröße von 5 Agenten um 10000 Schritte. Es wird deutlich, dass sich mit zunehmender Anzahl von Agenten innerhalb einer Gemeinschaft die Fehlerindikatoren Avg1 und Avg2 verschlechtern, was mit einer ansteigenden Schwierigkeit größere Gemeinschaften zu organisieren begründet ist. Die Abbildungen 52 und 53 fassen die Ergebnisse der insgesamt 90 Simulationsdurchläufe zusammen, indem sie den Verlauf der durchschnittlichen Werte des Avg1 und Avg2 in Bezug auf die Gemeinschaftsgröße gegenüberstellen.

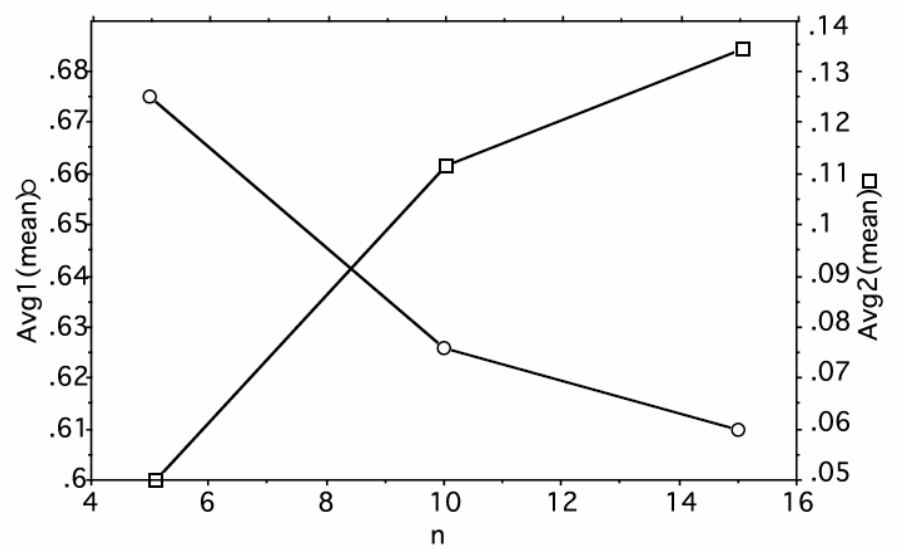


Abb. 52 Hutchins & Hazlehurst: Durchschnittswerte des Avg1 und Avg2 bei unterschiedlicher Gemeinschaftsgröße
Quelle: [HH95, S. 176, Figure 9.12]

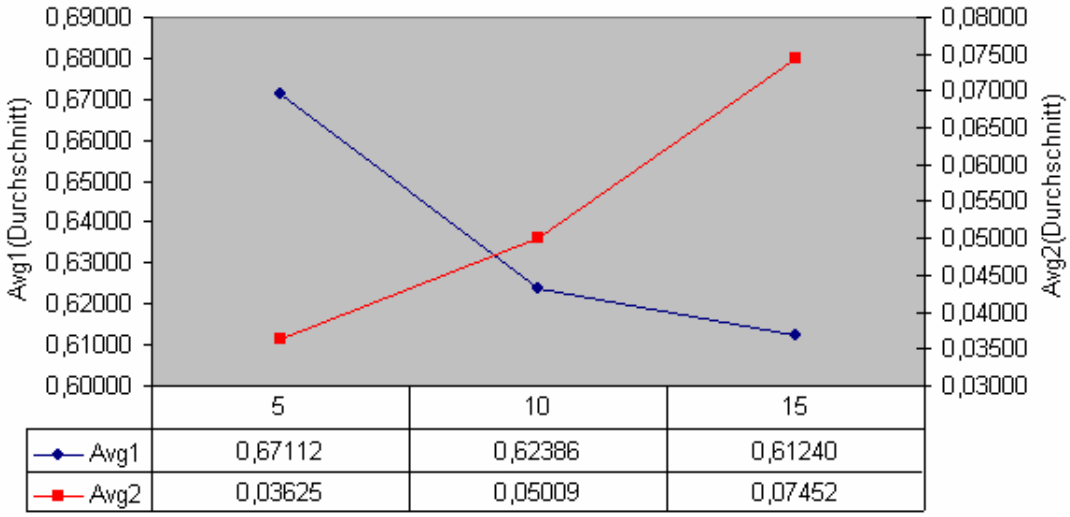


Abb. 53 LexLearn: Durchschnittswerte des Avg1 und Avg2 bei unterschiedlicher Gemeinschaftsgröße

Hierbei unterscheiden sich die durchschnittlichen, durch LexLearn berechneten Werte des Avg1 nur marginal von den durch Hutchins und Hazlehurst erlangten Ergebnissen. Die Werte des Fehlerindikators Avg2 differieren in den beiden Implementierungsvarianten zu Gunsten von LexLearn um den Faktor 2.
Hutchins und Hazlehurst stellen fest, dass sich die Fähigkeit einer Gemeinschaft ein „gutes“ Lexikon zu formen, mit ansteigender Größe exponentiell mit dem Faktor 1/*n* verschlechtert (siehe [HH95, S. 177]).

5.2 Simulation 2

Hutchins und Hazlehurst untersuchen in einer zweiten Simulation das Verhalten ihres Modells anhand eines Durchlaufs mit komplexeren Simulationsparametern. Im Vergleich zur ersten Simulation wird die Anzahl und Größe der visuellen Szenen deutlich erhöht, was automatisch zu einer komplexeren Struktur der neuronalen Netze führt. Die Menge der visuellen Eingabemuster entspricht hierbei den Mondphasen in der realen Welt. Die Ergebnisse werden in diesem Unterkapitel, weit weniger analytisch als im vorangegangenen Unterkapitel, mit den Ergebnissen eines Durchlaufs mit gleichen Parametern unter der Verwendung von LexLearn verglichen.

5.2.1 Simulationsparameter

Visuelle Szenen:

- $m = 5$
- S = {((0,0,0,0,0,0),(0,0,0,0,0,0),(0,0,0,0,0,1),(0,0,0,0,0,1),(0,0,0,0,0,0),(0,0,0,0,0,0)),
  ((0,0,0,0,0,0),(0,0,0,0,1,0),(0,0,0,0,1,1),(0,0,0,0,1,1),(0,0,0,0,1,0),(0,0,0,0,0,0)),
  ((0,0,0,1,0,0),(0,0,0,1,1,0),(0,0,0,1,1,1),(0,0,0,1,1,1),(0,0,0,1,1,0),(0,0,0,1,0,0)),
  ((0,0,1,1,0,0),(0,0,1,1,1,0),(0,0,1,1,1,1),(0,0,1,1,1,1),(0,0,1,1,1,0),(0,0,1,1,0,0)),
  ((0,0,1,1,0,0),(0,1,1,1,1,0),(0,1,1,1,1,1),(0,1,1,1,1,1),(0,1,1,1,1,0),(0,0,1,1,0,0)),
  ((0,0,1,1,0,0),(0,1,1,1,1,0),(1,1,1,1,1,1),(1,1,1,1,1,1),(0,1,1,1,1,0),(0,0,1,1,0,0)),
  ((0,0,1,1,0,0),(0,1,1,1,1,0),(1,1,1,1,1,0),(1,1,1,1,1,0),(0,1,1,1,1,0),(0,0,1,1,0,0)),
  ((0,0,1,1,0,0),(0,1,1,1,0,0),(1,1,1,1,0,0),(1,1,1,1,0,0),(0,1,1,1,0,0),(0,0,1,1,0,0)),
  ((0,0,1,0,0,0),(0,1,1,0,0,0),(1,1,1,0,0,0),(1,1,1,0,0,0),(0,1,1,0,0,0),(0,0,1,0,0,0)),
  ((0,0,0,0,0,0),(0,1,0,0,0,0),(1,1,0,0,0,0),(1,1,0,0,0,0),(0,1,0,0,0,0),(0,0,0,0,0,0)),
  ((0,0,0,0,0,0),(0,0,0,0,0,0),(1,0,0,0,0,0),(1,0,0,0,0,0),(0,0,0,0,0,0),(0,0,0,0,0,0)),
  ((0,0,0,0,0,0),(0,0,0,0,0,0),(0,0,0,0,0,0),(0,0,0,0,0,0),(0,0,0,0,0,0),(0,0,0,0,0,0))}.

Agenten:

- $n = 5$
- f_{arch} = Ein- und Ausgabeschicht mit 36 Neuronen, zwei verborgene Schichten mit jeweils 4 Neuronen (siehe Abbildung 54)
- W = Menge von zufälligen reellen Zahlen zwischen - 0.5 und + 0.5

- μ = 0.075 (Hutchins und Hazlehurst), 0.75 (LexLearn)
- ψ = 0.9

Interaktionsprotokoll:

- f_{pop} = die Agenten existieren vom Start bis zum Ende eines Simulationsdurchlaufs; es werden keine neuen Agenten erzeugt
- f_{scene} = zufällige Auswahl einer visuellen Szene aus der Menge *S*
- f_{ind} = zufällige Auswahl zweier Individuen aus der Gemeinschaft der Agenten

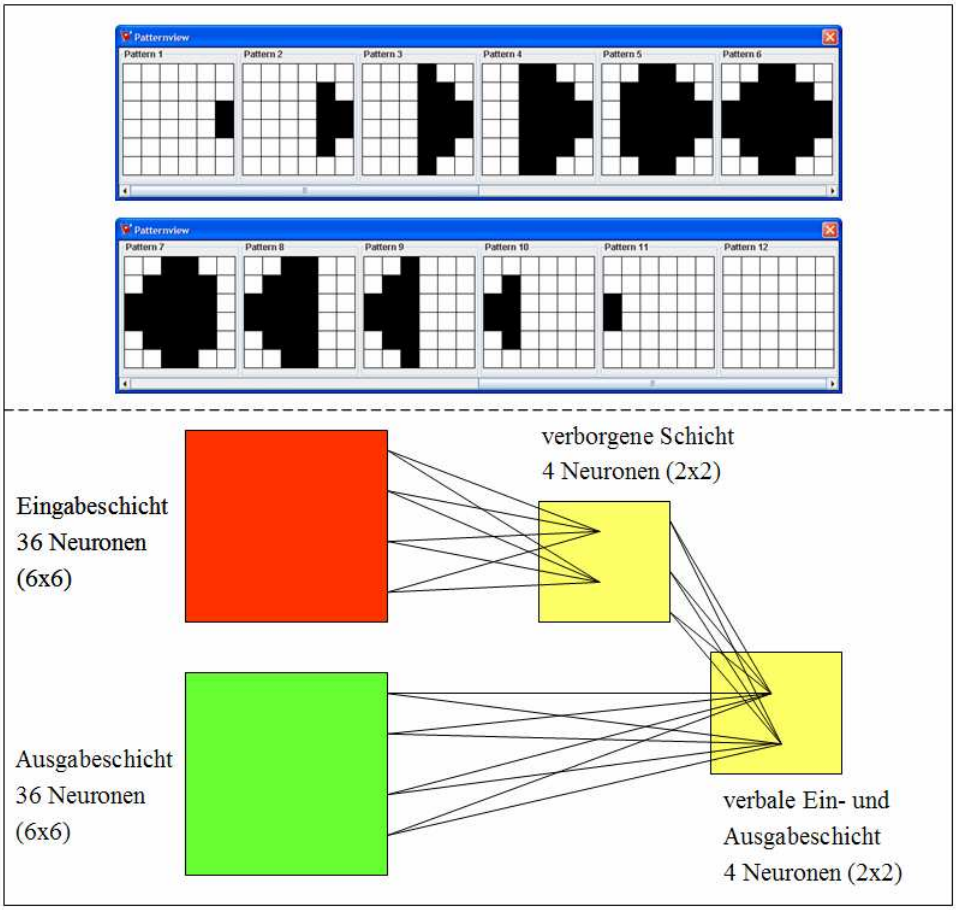


Abb. 54 Eingabemuster (oben) und neuronale Netzstruktur (unten) in Simulation 2
Quelle: Neuzeichnung von [HH95, S. 167, Figure 9.5]

5.2.2 Durchführung und Analyse der Simulation

Die Analyse der Emergenz eines gemeinsam genutzten Lexikons beschränkt sich in diesem Abschnitt auf die drei-dimensionale Darstellung der Aktivierungswerte der Wortschicht-Neuronen von vier Agenten zu Beginn eines Simulationsdurchlaufs und nach 2000 Interaktionen. Die folgenden Abbildungen stellen die Ergebnisse von Hutchins und Hazlehurst den durch LexLearn erzielten Resultaten gegenüber.

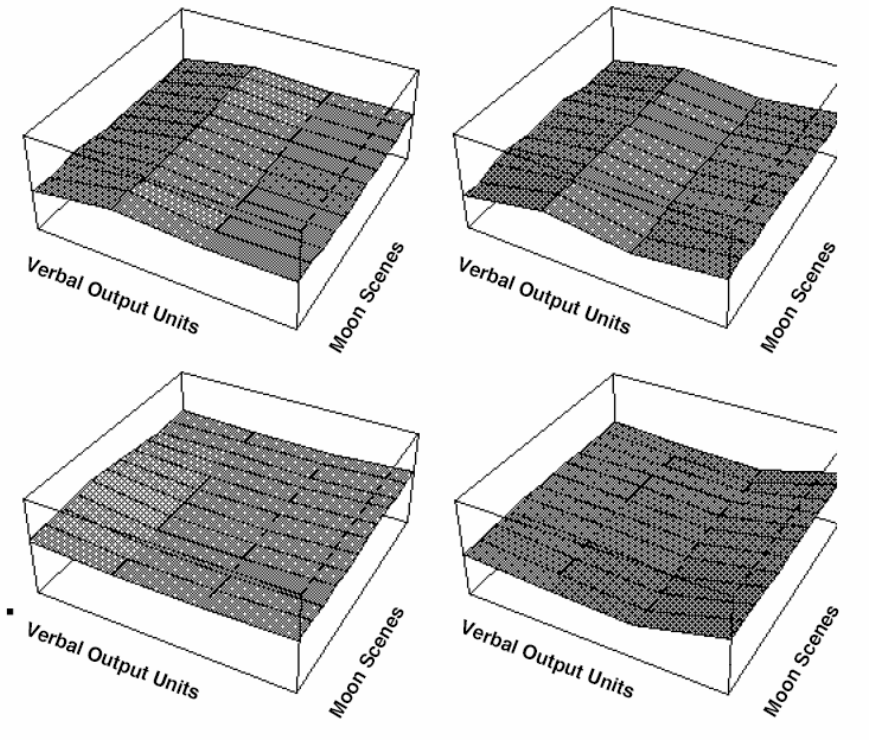


Abb. 55 Hutchins & Hazlehurst: Aktivierungsebenen der Wortschicht von 4 Agenten zu Beginn des Simulationsdurchlaufs
Quelle: [HH95, S. 168, Figure 9.7]

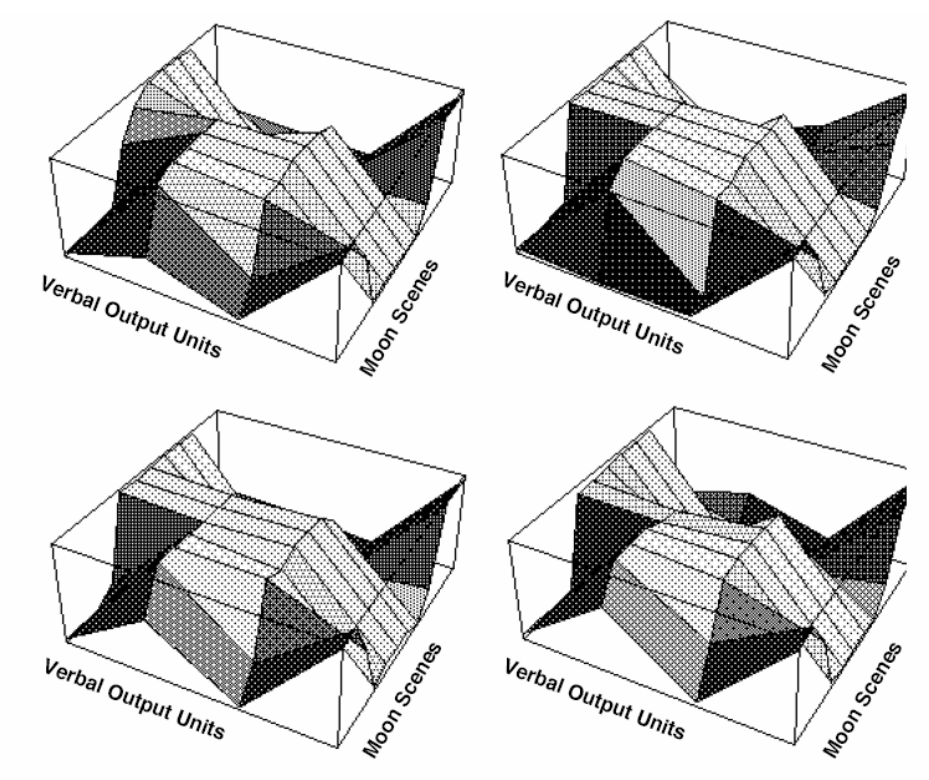


Abb. 56 Hutchins & Hazlehurst: Aktivierungsebenen der Wortschicht von 4 Agenten nach 2000 Interaktionen
Quelle: [HH95, S. 168, Figure 9.8]

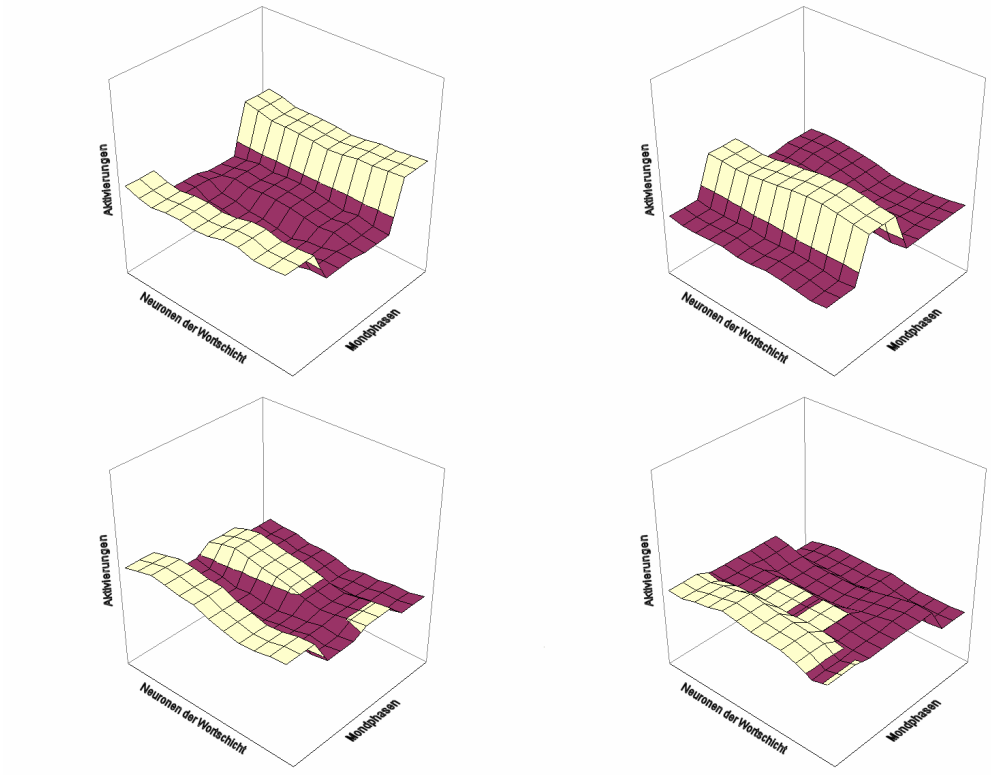


Abb. 57 LexLearn: Aktivierungsebenen der Wortschicht von 4 Agenten zu Beginn des Simulationsdurchlaufs

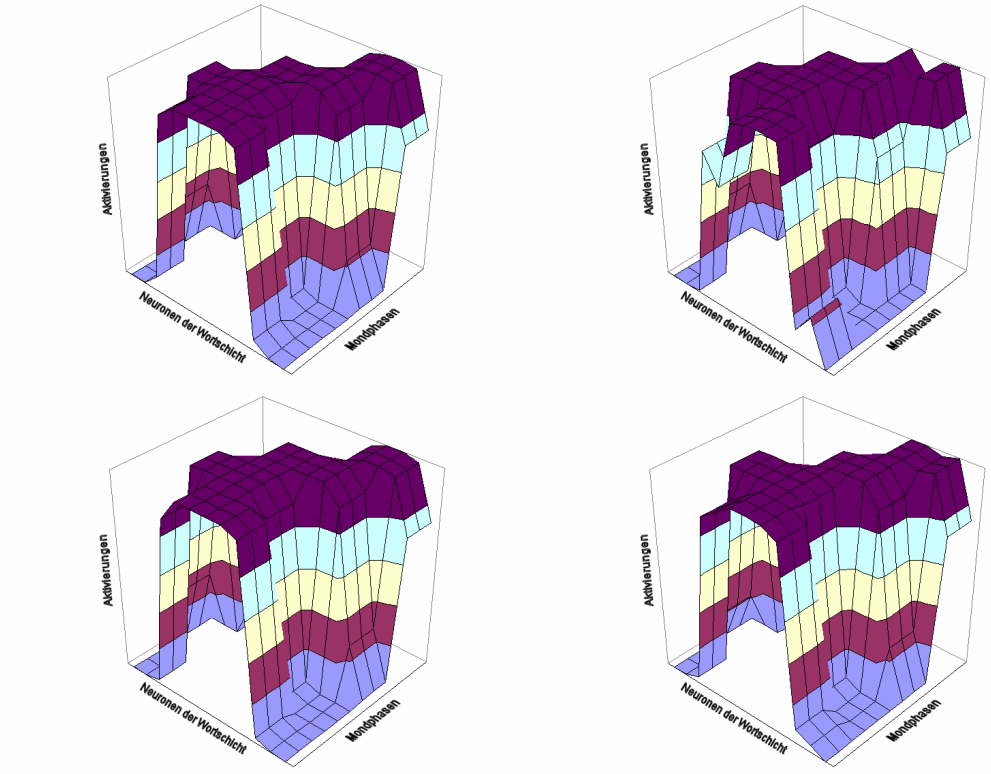


Abb. 58 LexLearn: Aktivierungsebenen der Wortschicht von 4 Agenten nach 2000 Interaktionen

Die Abbildungen 55 und 57 zeigen den Aufbau der Wörter für alle Mondphasen zu Beginn des Simulationsdurchlaufs. Die Tatsache, dass die Aktivierungswerte der Wortschichtneuronen eines Agenten für alle visuellen Eingabemuster kaum differieren und somit keine Unterscheidung zwischen den visuellen Szenen stattfindet, zeigt die Nichtexistenz eines Lexikons. Aufgrund der zufällig gewählten Verbindungsgewichtungen innerhalb der neuronalen Netze liegen die Aktivierungswerte der Wortschicht-Neuronen *aller* Agenten um den Wert 0.5.
Die Struktur der Wörter aller Agenten nach 2000 Interaktionen, was bei Anzahl von 5 Agenten innerhalb einer Gemeinschaft 10000 Simulationsschritten entspricht, ist in den Abbildungen 56 und 58 ersichtlich. Die Emergenz eines Lexikons ist in beiden Implementierungsvarianten an zwei Merkmalen erkennbar.
Jeder Agent bildet für jede visuelle Szene ein unterschiedliches Wort, was an den verschiedenen Aktivierungswerten der Neuronen der Wortschicht erkennbar ist. Dies entspricht dem Erfüllen der ersten Bedingung an ein gemeinsam genutztes Lexikon (vgl. Abschnitt 2.4.1). In den Grafiken äußert sich dies durch die Bildung von Höhen und Tiefen innerhalb der Ebene der Aktivierungswerte.
Das Erfüllen der zweiten Bedingung eines Lexikons, die Tatsache, dass alle Agenten das gleiche Wort für eine visuelle Szene nutzen, zeigt sich grafisch durch die Ähnlichkeit der Aktivierungsebenen aller Agenten.
Beide Merkmale finden sich sowohl in den Ergebnissen von Hutchins und Hazlehurst als in den durch LexLearn erlangten Resultaten wieder, Unterschiede zwischen beiden Varianten lassen sich in dieser Simulation nicht feststellen.

5.3 Simulation 3

In den ersten beiden Simulationen wurde gezeigt, welche Auswirkungen eine Veränderung der Gemeinschaftsgröße bzw. der Menge und Größe der Eingabemuster auf die Ergebnisse eines Simulationsdurchlaufs hat. Eine dritte Simulation, welche von Hutchins und Hazlehurst nicht durchgeführt wurde, zeigt die Auswirkungen einer in der Größe veränderten Wortschicht auf die Resultate eines Durchlaufs mit ansonsten gleichen Parametern. In der dritten Simulation entspricht die Menge der Eingabemuster den Buchstaben des deutschsprachigen Alphabets ohne die Umlaute.

5.3.1 Simulationsparameter

Interaktionsprotokoll:

- f_{pop} = die Agenten existieren vom Start bis zum Ende eines Simulationsdurchlaufs; es werden keine neuen Agenten erzeugt
- f_{scene} = zufällige Auswahl einer visuellen Szene aus der Menge *S*
- f_{ind} = zufällige Auswahl zweier Individuen aus der Gemeinschaft der Agenten

Visuelle Szenen:

- $m = 5$
- S = { ((1,1,1,1,1),(1,0,0,0,1),(1,0,0,0,1),(1,1,1,1,1),(1,0,0,0,1),(1,0,0,0,1),(1,0,0,0,1)),
  ((1,1,1,1,0),(1,0,0,1,0),(1,0,0,1,0),(1,1,1,1,1),(1,0,0,0,1),(1,0,0,0,1),(1,1,1,1,1)),
  ((1,1,1,1,1),(1,0,0,0,0),(1,0,0,0,0),(1,0,0,0,0),(1,0,0,0,0),(1,0,0,0,0),(1,1,1,1,1)),
  ((1,1,1,1,0),(1,0,0,1,1),(1,0,0,0,1),(1,0,0,0,1),(1,0,0,0,1),(1,0,0,1,1),(1,1,1,1,0)),
  ((1,1,1,1,1),(1,0,0,0,0),(1,0,0,0,0),(1,1,1,1,0),(1,0,0,0,0),(1,0,0,0,0),(1,1,1,1,1)),
  ((1,1,1,1,1),(0,0,1,0,0),(0,0,1,0,0),(0,1,1,1,0),(0,0,1,0,0),(0,0,1,0,0),(0,0,1,0,0)),
  ((1,1,1,1,1),(1,0,0,0,0),(1,0,0,0,0),(1,0,1,1,1),(1,0,1,0,1),(1,0,0,0,1),(1,1,1,1,1)),
  ((1,0,0,0,1),(1,0,0,0,1),(1,0,0,0,1),(1,1,1,1,1),(1,0,0,0,1),(1,0,0,0,1),(1,0,0,0,1)),
  ((0,1,1,1,0),(0,0,1,0,0),(0,0,1,0,0),(0,0,1,0,0),(0,0,1,0,0),(0,0,1,0,0),(0,1,1,1,0)),
  ((0,0,0,0,1),(0,0,0,0,1),(0,0,0,0,1),(0,0,0,0,1),(0,0,0,0,1),(0,1,0,0,1),(0,1,1,1,1)),
  ((1,0,0,0,1),(1,0,0,1,0),(1,0,1,0,0),(1,1,0,0,0),(1,0,1,0,0),(1,0,0,1,0),(1,0,0,0,1)),
  ((1,0,0,0,0),(1,0,0,0,0),(1,0,0,0,0),(1,0,0,0,0),(1,0,0,0,0),(1,0,0,0,0),(1,1,1,1,1)),
  ((1,0,0,0,1),(1,1,0,1,1),(1,0,1,0,1),(1,0,1,0,1),(1,0,0,0,1),(1,0,0,0,1),(1,0,0,0,1)),
  ((1,0,0,0,1),(1,1,0,0,1),(1,1,0,0,1),(1,0,1,0,1),(1,0,1,0,1),(1,0,0,1,1),(1,0,0,1,1)),
  ((0,1,1,1,0),(1,0,0,0,1),(1,0,0,0,1),(1,0,0,0,1),(1,0,0,0,1),(1,0,0,0,1),(0,1,1,1,0)),
  ((1,1,1,1,1),(1,0,0,0,1),(1,0,0,0,1),(1,1,1,1,1),(1,0,0,0,0),(1,0,0,0,0),(1,0,0,0,0)),
  ((0,1,1,1,0),(1,0,0,0,1),(1,0,0,0,1),(1,0,0,0,1),(1,0,1,0,1),(1,0,0,1,0),(0,1,1,0,1)),
  ((1,1,1,1,1),(1,0,0,0,1),(1,0,0,0,1),(1,1,1,1,1),(1,1,0,0,0),(1,0,1,1,0),(1,0,0,0,1)),
  ((0,1,1,1,1),(1,0,0,0,0),(1,0,0,0,0),(1,1,1,1,1),(0,0,0,0,1),(0,0,0,0,1),(1,1,1,1,0)),
  ((1,1,1,1,1),(0,0,1,0,0),(0,0,1,0,0),(0,0,1,0,0),(0,0,1,0,0),(0,0,1,0,0),(0,0,1,0,0)),
  ((1,0,0,0,1),(1,0,0,0,1),(1,0,0,0,1),(1,0,0,0,1),(1,0,0,0,1),(1,0,0,0,1),(1,1,1,1,1)),
  ((1,0,0,0,1),(1,0,0,0,1),(1,0,0,0,1),(1,0,0,0,1),(1,0,0,0,1),(1,1,0,1,1),(0,1,1,1,0)),
  ((1,0,0,0,1),(1,0,0,0,1),(1,0,0,0,1),(1,0,1,0,1),(1,0,1,0,1),(1,1,0,1,1),(1,0,0,0,1)),
  ((1,0,0,0,1),(0,1,0,1,0),(0,0,1,0,0),(0,0,1,0,0),(0,0,1,0,0),(0,1,0,1,0),(1,0,0,0,1)),
  ((1,0,0,0,1),(1,0,0,0,1),(1,0,0,0,1),(0,1,1,1,0),(0,0,1,0,0),(0,0,1,0,0),(0,0,1,0,0)),
  ((1,1,1,1,1),(0,0,0,0,1),(0,0,0,1,1),(0,0,1,1,0),(0,1,1,0,0),(1,1,0,0,0),(1,1,1,1,1)) }

Agenten:

- $n = 5$
- f_{arch} = Ein- und Ausgabeschicht mit 35 Neuronen, zwei verborgene Schichten (vgl. Abbildung 59) mit jeweils: - 5 Neuronen
  - 10 Neuronen
  - 15 Neuronen
- W = Menge von zufälligen reellen Zahlen zwischen - 0.5 und + 0.5
- $\mu = 0.75$
- $\psi = 0.9$

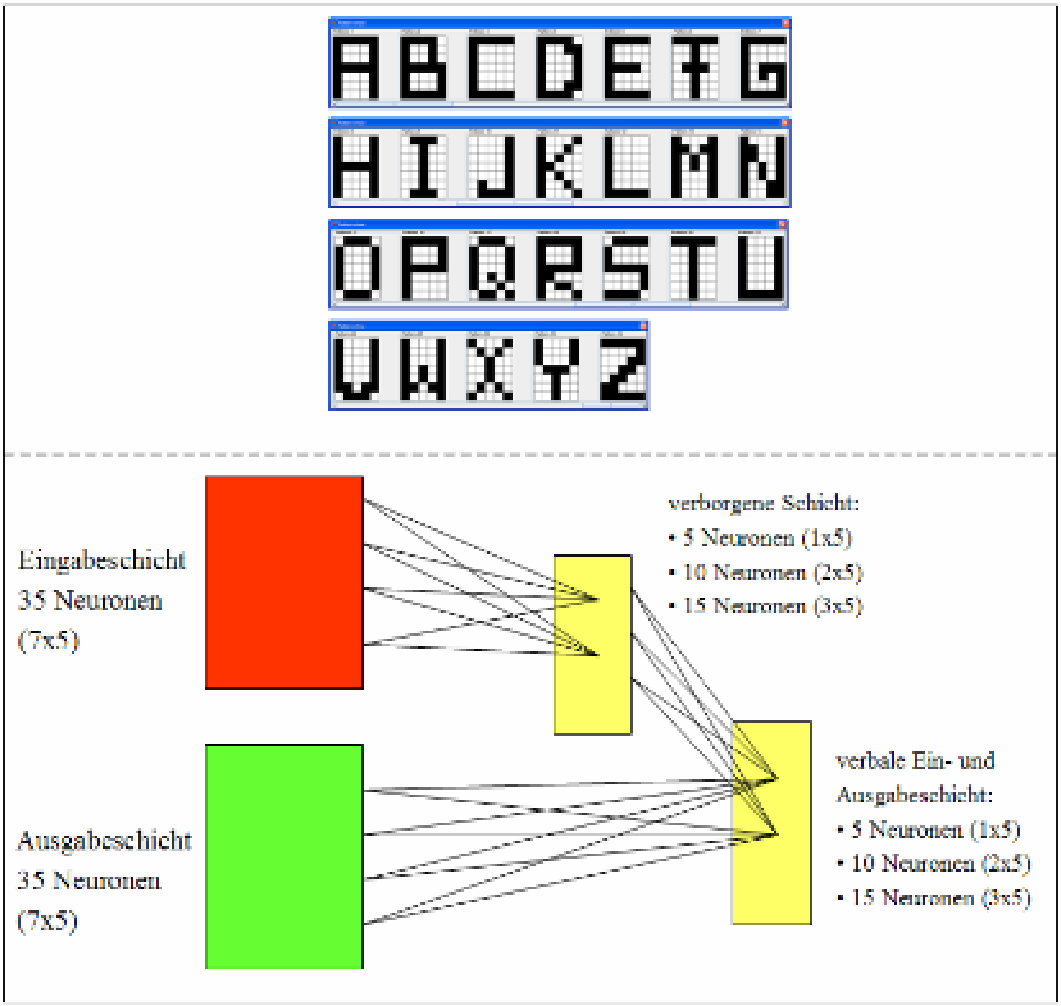


Abb. 59 Eingabemuster (oben) und neuronale Netzstruktur (unten) in Simulation 3

5.3.2 Durchführung und Analyse der Simulation

Wie aus den Simulationsparametern ersichtlich, werden in diesem Abschnitt die Ergebnisse aus drei verschiedenen Simulationsdurchläufen miteinander verglichen. In einem ersten Durchlauf besitzt jeder Agent fünf Neuronen in der verborgenen und in der Wortschicht, welche für die Bildung eines Lexikons mit 26 Wörtern ausreichen. In den weiteren Simulationsdurchläufen wird die Größe der verborgenen bzw. der Wortschicht eines Agenten um jeweils fünf Neuronen erhöht.
Abbildung 60 stellt die Auswirkungen einer veränderten Struktur der neuronalen Netze innerhalb der unterschiedlichen Simulationsdurchläufe auf den Verlauf der Fehlerindikatoren Avg1 und Avg2 gegenüber. Die Agenten innerhalb der ersten Simulationsumgebung (roter Graph) besitzen 5 Neuronen in den verborgenen Schichten, die der zweiten Umgebung (blauer Graph) 10 Neuronen und die der dritten Umgebung (grüner Graph) 15 Neuronen.
Wie in Abbildung 60 ersichtlich, zeigen sich zwei unterschiedliche Eigenschaften. Die Fähigkeit einer Agentengemeinschaft, differierende Wörter für alle visuellen Szenen zu bilden, nimmt mit zunehmender Größe der verborgenen Schichten ab, was sich aus dem Verlauf der Graphen des Avg1 ableiten lässt.

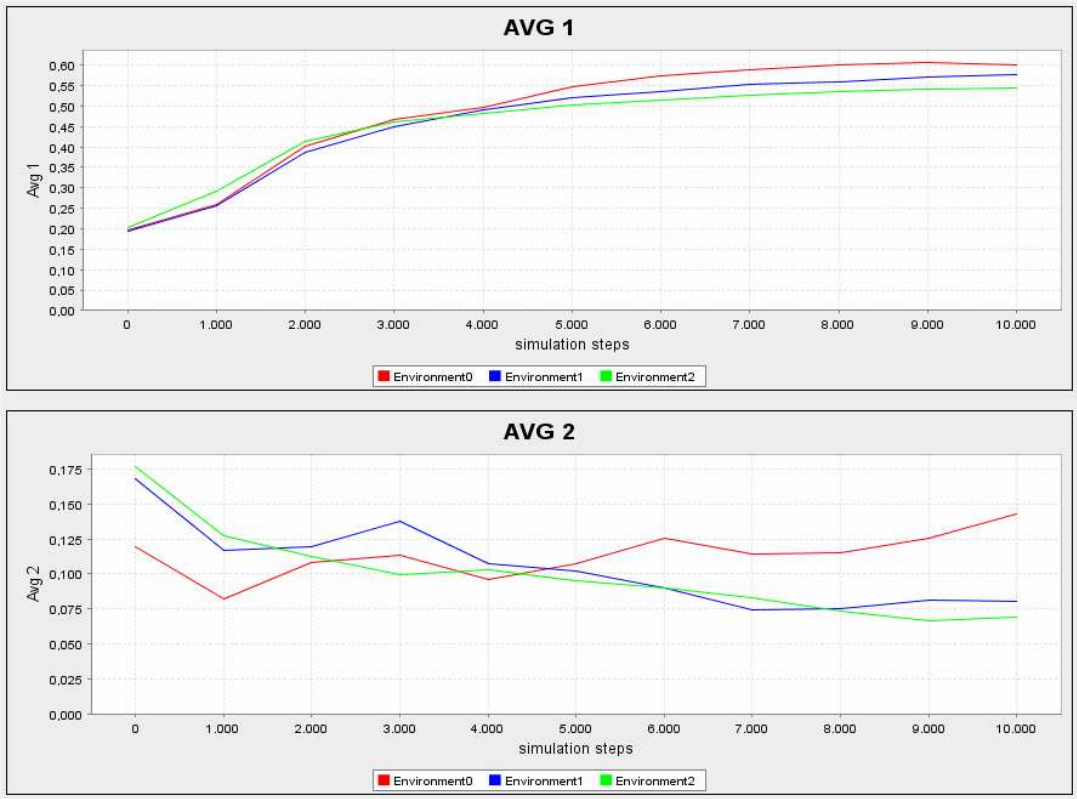


Abb. 60 Simulationsdurchlauf mit unterschiedlichen Strukturen der neuronalen Netze

Im Gegensatz dazu steigt die Fähigkeit einer Gemeinschaft, sich auf gemeinsame Wörter für alle visuellen Szenen zu einigen, welches durch den Verlauf der Graphen des Avg2 verdeutlicht wird.
In Abbildung 61 wird eine weitere Eigenschaft, die durch die Erhöhung der Anzahl von Neuronen innerhalb der verborgenen Schichten entsteht, dargestellt.

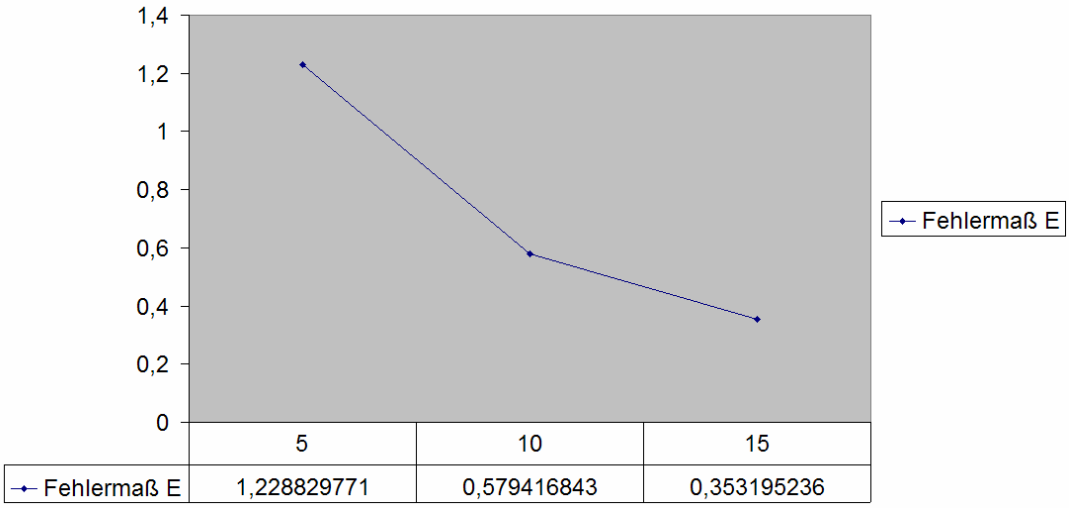


Abb. 61 Entwicklung des Fehlermaßes E bei 5, 10 und 15 Neuronen innerhalb der verborgenen Schichten

Das Fehlermaß *E* beschreibt, wie bereits in Abschnitt 2.3.3 erläutert, dabei die durchschnittliche Qualität der Deduktion einer visuellen Szene aller Agenten, also die Fähigkeit jedes Agenten, ein visuelles Eingabemuster korrekt abzuzeichnen.
Die Tabelle in Abbildung 61 stellt mehrere Varianten von Deduktion innerhalb der drei unterschiedlichen Simulationsdurchläufe dar. Die erste Zeile verdeutlicht, dass die Wahrscheinlichkeit, ähnlich wirkende visuelle Eingabemuster, in diesem Fall die Buchstaben „B“ und „G“, zu verwechseln, mit zunehmender Anzahl von Neuronen innerhalb der verborgenen Schichten abnimmt. Visuelle Eingabeszenen, die sich von anderen Eingabemustern abgrenzen, werden, wie in der zweiten Zeile ersichtlich, trotz geringer Neuronenanzahl gut abgezeichnet. Nimmt die Komplexität des Eingabemusters, wie bei dem Buchstaben „Z“ in Zeile 3 der Tabelle, zu, so fällt es den Agenten mit niedriger Neuronenanzahl in der verborgenen Schicht wesentlich schwerer, die visuelle Szene korrekt abzuzeichnen.

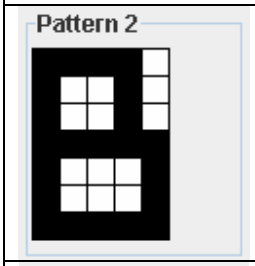
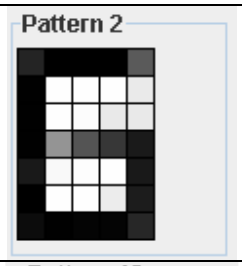
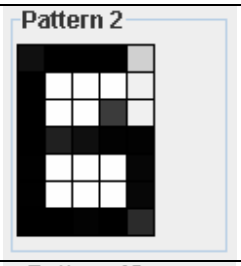
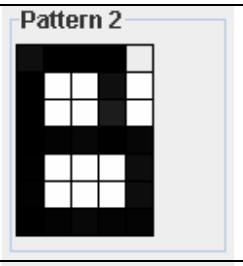
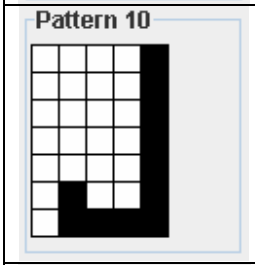
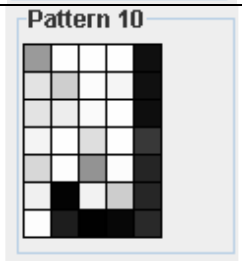
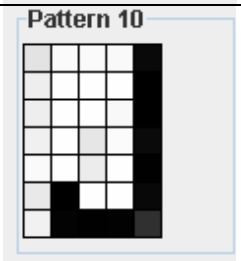
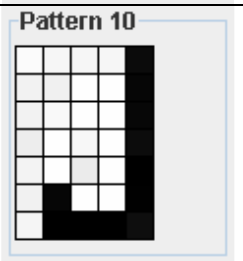
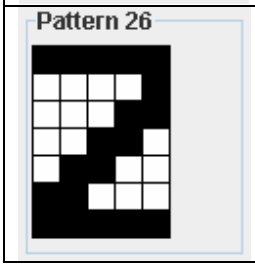
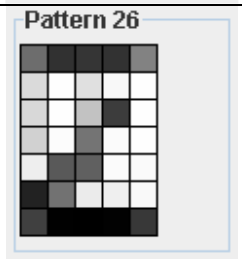
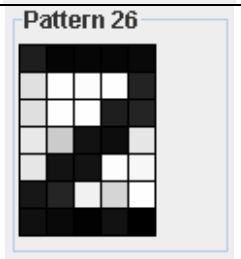
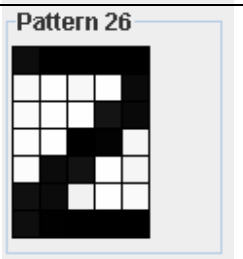
Buchstabe	Deduktion mit 5 Neuronen in verborgenen Schichten	Deduktion mit 10 Neuronen in verborgenen Schichten	Deduktion mit 15 Neuronen in verborgenen Schichten
			
			
			

Abb. 62 Deduktionen von Agenten mit unterschiedlicher Anzahl an Wortschicht-Neuronen

6. Einordnung in EMIL und Ausblick

Dieses Kapitel ordnet die vorliegende Arbeit in das EU-Projekt EMIL ein und bietet eine Übersicht über die Erweiterungsmöglichkeiten der Anwendung LexLearn.

6.1 Einordnung in EMIL

Das Projekt „Emergence in the Loop: Simulating the two way dynamics of norm innovation“(EMIL[18]) legt den Fokus auf die Innovation von Normen und besitzt die folgenden drei theoretischen Hauptziele (siehe [EMILA06], S. 4):

- die Komplexität in sozialen Systemen mit autonomen Agenten zu verstehen,
- das Entstehen und Ausbreiten von neuen Konventionen und Normen in diesen Systemen zu verstehen,
- die Innovation von Normen mittels agentenbasierter Simulation zu analysieren.

Das technologische Hauptziel des Projekts ist die Erstellung eines Simulators, um die Innovation von Normen zu erforschen.
Das von der Europäischen Union geförderte Projekt unterteilt sich in die folgenden drei Projektabschnitte (siehe [EMILA06], S. 21ff.):

- EMIL-M: Das Ziel dieses Aufgabenpakets ist die Erstellung eines Modells zur Innovation von Normen in komplexen sozialen Prozessen zwischen intelligenten sozialen Agenten. Empirische Vorbilder der Emergenz von Normen tragen zur Entwicklung des Modells bei.
- EMIL-S: Der Zweck dieses Projektabschnitts ist die Implementierung eines softwarebasierten Simulators, welcher Simulationen durchführt um deren Ergebnisse mit den aus EMIL-M verfügbaren empirischen Daten zu vergleichen. Der Simulator soll sowohl mehrere Vorlagen für Agenten als auch für Simulationsumgebungen bereitstellen.
- EMIL-T: Die Zielvorgabe dieses Aufgabenpakets ist die Evaluation der mittels des Simulators erhaltenen Ergebnisse. Hierbei werden sowohl bereits formulierte als auch durch EMIL-S neu gewonnene Erkenntnisse untersucht.

Diese Diplomarbeit ordnet sich dabei in den Projektabschnitt EMIL-S ein. Anhand dieser Implementierung eines bereits beschriebenen Modells sollen weitere Erkenntnisse über die Anforderungen an die in EMIL-S beabsichtigte Architektur und das Design, insbesondere der Agenten und der Simulationsumgebungen, gewonnen werden. Die Normentstehung könnte dabei ähnlich der Entstehung eines

[18] siehe http://emil.istc.cnr.it/

Lexikons simuliert werden. Die Verhältnisse von Formen zu Namen und von Verhalten zu einer Regel sind dabei ähnlich. Hierbei muss jedoch berücksichtigt

werden, dass sowohl Verhalten viel komplexer als eine Form als auch eine Regel bei weitem komplexer als ein Wort ist. (siehe [EMILD07], Folie 16)
An dem Projekt, welches für den Zeitraum September 2006 bis August 2009 angelegt ist, sind folgende Einrichtungen beteiligt:

- Institute for Cognitive Science and Technology, CNR
- University of Bayreuth, Department of Philosophy
- University of Surrey, Centre for Research on Social Simulation
- University of Koblenz-Landau, Department of Computer Science[19]
- Manchester Metropolitan University, Centre for Policy Modelling
- AITIA International Inc.

6.2 Erweiterungen des Modells

In diesem Unterkapitel werden abschließend zum bereits beschriebenen Modell mögliche Erweiterungsmöglichkeiten vorgestellt. Basierend auf dem Werkzeug LexLearn sind die hier erläuterten drei Veränderungen relativ leicht realisierbar, was jedoch zukünftigen Arbeiten überlassen wird. Dieses Unterkapitel orientiert sich dabei an den Ausführungen von Hutchins und Hazlehurst in [HH95] auf den Seiten 177ff.

6.2.1 Die Notwendigkeit einer kritischen Phase des Spracherlernens

In der vorliegenden Arbeit wird stets die Entstehung eines Lexikons innerhalb einer Gemeinschaft von Agenten mit untrainierten neuronalen Netzen untersucht. Was jedoch geschieht, wenn neue unerfahrene Agenten neu in eine Gruppe mit einem bereits bestehenden Lexikon aufgenommen werden?
Hutchins und Hazlehurst führten diese Experimente bereits durch und beobachteten unterschiedliche Effekte: Während untrainierte Agenten in größeren Gemeinschaften relativ schnell „integriert“ werden, d. h. sie das Lexikon der Gruppe erlernen, kann die Aufnahme eines untrainierten Agenten in eine kleinere Gemeinschaft weit drastischere Folgen haben. So können neue Agenten das bereits existierende Lexikon innerhalb der Gruppe derart stören, dass es der Gruppe auch nach mehreren tausend weiteren Simulationsschritten nicht mehr gelingt, ein allgemein gültiges Lexikon zu bilden.
Hutchins und Hazlehurst schlagen daher vor, entsprechend dem menschlichen Leben, die Agenten mit einer *kritischen Phase des Spracherlernens* zu Beginn ihres „Lebens“ auszustatten und sie mit steigendem „Alter“ weniger sensibel für Veränderungen des Wortschatzes zu machen. So würde eine Gemeinschaft gerade

[19] siehe http://www.uni-koblenz.de/FB4/Institutes/IWVI/AGTroitzsch/Projects/ReGhost

entstandener Agenten, im menschlichen Fall Neugeborene, sich einerseits schnell auf ein gemeinsames Lexikon einigen können. Mit steigendem Alter jedoch würden sie unsensibler für die „Störungen“ der neu zur Gemeinschaft hinzu stoßenden Agenten, im menschlichen Falle Nachkommen der bereits existierenden Population. Hutchins und Hazlehurst implementierten diesen Ansatz jedoch nicht, daher kann von keinen Beobachtungen zu dieser Variante berichtet werden.
Zur Realisierung dieses Modells müssen in LexLearn mehrere kleine Veränderungen aufgenommen werden. Zum einen existiert die Notwendigkeit, die Agenten mit einem Attribut auszugestatten, welches ihr Alter kodiert und somit innerhalb eines jeden Simulationsschritts inkrementiert wird. Des Weiteren muss die Möglichkeit geschaffen werden, Agenten aus einer Simulationsumgebung zu entfernen bzw. neue, untrainierte Agenten einer Simulationsumgebung hinzuzufügen. Dies entspräche, auf den Menschen bezogen, dem Tod und der Geburt eines Individuums. Eine letzte Änderung ist die Schaffung einer Abhängigkeit zwischen dem Alter eines Agenten und seiner Lernrate, wobei die Art der Beziehung beider Variablen noch geklärt werden muss.

6.2.2 Das Erlernen eines kohärenten Lexikons

Eine weitere Frage, die Hutchins und Hazlehurst ansprechen und die mit dem bereits in Abschnitt 6.2.1 erläuterten Problem zusammenhängt, ist, wie leicht es einem Agenten fällt, das bereits bestehende Lexikon einer Gemeinschaft zu erlernen. Anders ausgedrückt, wie unpräzise darf ein Lexikon sein, damit es dennoch gelernt werden kann?
Hutchins und Hazlehurst fanden dabei heraus, dass es einer Agentengemeinschaft leichter fällt, ein Lexikon aus einem Zustand zu bilden, in dem alle Agenten untrainierte neuronale Netze besitzen als aus einem Zustand, in dem die Netze der Agenten „falsch“ organisiert sind. Eine Falschorganisation darf in diesem Falle als eine große Varianz innerhalb der Zuordnung von visueller Szene zu einem Wort verstanden werden. Anders ausgedrückt kann man von einem Zerfall der Agentengemeinschaft in mehrere Untergemeinschaften mit verschiedenen teilweise ähnlichen Sprachen, auch Dialekte genannt, sprechen. Diese Thematik wird jedoch ausführlicher in Abschnitt 6.2.3 erläutert.
Innerhalb mehrerer Simulationsdurchläufe, die jedoch nicht genauer beschrieben werden, beobachteten Hutchins und Hazlehurst, dass die Fähigkeit der Agenten, eine visuelle Szene einem Wort zuzuordnen, mit der Existenz eines kohärenten Lexikons steigt. Genauer gesagt wirkt sich eine kleinere Varianz innerhalb der Zuordnung von visueller Szene zu Wort positiv auf die Geschwindigkeit des Prozesses aus, der innerhalb der verborgenen Schicht die Zuordnung von Szene zu Wort kodiert.
Die Realisierung dieses erweiterten Modells in LexLearn scheint auf den ersten Blick nicht sonderlich aufwändig. So könnten neue Agenten in einer veränderten Implementierung nicht nur mit Individuen einer Simulationsumgebung kommunizieren, sondern Interaktionen zwischen Mitgliedern verschiedener Agentengemeinschaften gestattet werden. Auf diese Art könnte eine Varianz innerhalb der Zuordnung von visueller Szene zu einem Wort relativ leicht simuliert

und Beobachtungen zu Agenten, die mit beiden Gemeinschaften kommunizieren, getroffen werden.

6.2.3 Die Entstehung von Dialekten

Gelegentlich gelingt es einer Gemeinschaft von Agenten nicht, ein gemeinsam genutztes Lexikon auszubilden. Dieser Effekt, der sowohl in der Implementierung von Hutchins und Hazlehurst als auch in LexLearn zu beobachten ist, tritt auf, weil es von Zeit zu Zeit vorkommt, dass zufällige Initialgewichtungen mehrerer Agenten, gepaart mit einer „unglücklichen“ Auswahl derer innerhalb des Interaktionsprotokolls zu einer Divergenz der verbalen Repräsentationen der Agenten einer Gemeinschaft führt, die nicht wieder überwunden werden kann. In diesem Fall ist die Sprache, die eine Teilmenge der Agenten erlernen wird, inkompatibel zu der Sprache anderer Agenten. Beide Untergruppen werden sich folglich niemals auf ein gemeinsames Lexikon einigen können.
Der Tatsache entsprechend, dass einige Initialgewichtungen zu einigen kompatibler sind als zu anderen, ermöglichten es Hutchins und Hazlehurst in einer Erweiterung ihres Modells den Agenten, sich ihren Konversationspartner selbst auszuwählen. Die Agenten wurden derart erweitert, dass sie sich die vergangenen Äußerungen anderer Agenten merken konnten und bei der Auswahl ihrer Gesprächspartner Agenten bevorzugten, deren vergangene Äußerungen denen des Agenten ähnlich waren.
Das Ergebnis dieser Veränderung des Interaktionsprotokolls war die Entstehung von Dialekten bzw. eine Partitionierung der Agentengemeinschaft in Gruppen von Agenten, die jeweils für sich ein kohärentes Lexikon ausbildeten.
Um eine Implementierung dieser Erweiterung in LexLearn zu ermöglichen, müssen einige Veränderungen vorgenommen werden. Einerseits muss den Agenten eine Art Gedächtnis implementiert werden, damit sie die vergangenen Äußerungen anderer Individuen speichern können, andererseits muss das Interaktionsprotokoll in der Art abgeändert werden, dass ein Sprecher seinen Zuhörer nicht mehr zufällig auswählt, sondern anhand seiner Präferenzen.

6.3 Evolution von sprachlichen Strukturen

Die in dieser Arbeit vorgestellte Implementierung des Modells von Hutchins und Hazlehurst zur Emergenz eines gemeinsam genutzten Lexikons zeigt, dass sich eine Gemeinschaft von Software-Agenten auf eine primitive, lexikalische Sprache über eine visuelle Umwelt einigen kann. Doch unter welchen Voraussetzungen und in welchen Schritten entwickelt sich eine primitive zu einer fast menschlichen Sprache?
Zum Abschluss untersucht dieses Kapitel die Evolution einer sprachlichen Struktur, ausgehend von lexikalischen bis hin zu grammatikalischen Kommunikations-Systemen. Die im Folgenden dargestellten Theorien orientieren sich an den Ausführungen von Luc Steels in [STE05]. Er unterscheidet zunächst drei

verschiedene Aspekte einer Sprache, die im Laufe einer Evolution ständig Änderungen ausgesetzt sind (vgl. [STE05, S. 214f.]):

- *die biologische Kompetenz* für eine Sprache, d. h. die physiologischen Komponenten, die einen Agenten befähigen an der Kommunikation innerhalb einer Gemeinschaft teilzunehmen;
- das *Sprachinventar* eines *individuellen Agenten*, welches er zur Abbildung eines Wortes auf dessen Bedeutung und umgekehrt nutzt;
- die *kommunale Sprache*, der Konsens einer Gemeinschaft Bedeutungen in gleicher Weise auszudrücken, ohne dass dieser Konsens explizit vorgegeben wurde.

Der Fortschritt des Evolutionsprozesses einer Sprache kann laut Steels anhand dreier verschiedener Eigenschaften bestimmt werden: der *Ursprung* einer Evolution, die *Art* der *Informationsweitergabe* innerhalb der eigentlichen Kommunikation und die *Stufe* eines sich entwickelnden Kommunikations-Systems.

Soziobiologischer versus soziokultureller Ursprung einer Evolution

Der *soziobiologische* Ansatz zur Herkunft der Entwicklung einer Sprache basiert sowohl auf der genetischen Programmierung als auch der natürlichen Auslese von Strategien der Kommunikation und kann in drei Punkten zusammengefasst werden:

1. Die Wahrscheinlichkeit der Individuen, die erfolgreicher an einer Kommunikation teilnehmen, sich fortzupflanzen ist höher als die der Individuen, deren Kommunikationserfolg geringer ausfällt.
2. Wenn sich eine Strategie *X* als überlegen zeigt, werden sich alle Individuen mit dieser angeborenen Strategie *X* mit größerer Wahrscheinlichkeit fortpflanzen.
3. Aus diesem Grund setzt sich Strategie X unter allen Individuen einer Gemeinschaft durch und dient allein zur Aufrechterhaltung des Kommunikations-Systems innerhalb dieser Gemeinschaft.

Die möglichen Kommunikationsstrategien erstrecken sich dabei von einer bidirektionalen Erstellung eines Lexikons bis hin zur Nutzung von grammatikalischen Konstruktionen (siehe Unterpunkt Stufen eines sich entwickelnden Kommunikations-Systems).
Der *soziokulturelle* Ansatz zum Ursprung der Entwicklung einer Sprache erfolgt sowohl durch kulturelle Auswahl als auch durch direkte Rückmeldung von Erfolg oder Misserfolg der Kommunikation und kann wiederum in drei Punkten zusammengefasst werden:

1. Jedes Individuum besitzt verschiedene Strategien ein Kommunikations-System zu nutzen.
2. Es existieren effektivere Strategien, die zu einer höheren Ausdrucksweise führen und somit zu größerem Kommunikationserfolg mit geringerem Aufwand führen.

3. Individuen versuchen den Kommunikationserfolg zu maximieren und den dazu benötigten Aufwand zu minimieren. Die Strategie, die dies am stärksten unterstützt, wird sich innerhalb der Gemeinschaft durchsetzen.

Die Kommunikationsstrategien dieses Ansatzes gleichen den Strategien des soziobiologischen Ansatzes.
Anhand der Ausführungen von Steel (siehe [STE05, S.7f.]) wird deutlich, dass kaum ein Forschungsansatz existiert, der die Ursprünge der Evolution einer Sprache nur in einem der beiden Ansätze sieht.

Signalisierende Systeme versus Sprachspiele

Die Evolution eines Lexikons hängt auch von der Art und Weise der eigentlichen Kommunikation ab, d. h. in welcher Form eine Informationsübertragung zwischen zwei Individuen stattfindet. Steel unterscheidet dabei zwei verschiedene Varianten.

In der ersten Variante steht die effiziente und zuverlässige *Kodierung* bzw. *Dekodierung* einer Information, der *Nachricht*, in Form eines Signals im Mittelpunkt. Der Erfolg einer Kommunikation wird in dieser Variante durch den Vergleich der Information eines Senders mit der dekodierten Information des Empfängers gemessen. Eine agentenbasierte Simulation, in der die Agenten, aus einer Menge von Informationen, Nachricht-Signal-Matrizen entwickeln, zeigt die Evolution einer Sprache innerhalb dieser Gemeinschaft. Die Leistungsfähigkeit dieser Sprache kann anhand der Unterschiedlichkeit der Matrizen einzelner Agenten gemessen werden.
Die zweite Variante betont die „schlussfolgernden" Eigenschaften einer Sprache, welche einen Hörer zu einer, von einem Sprecher intendierten, Handlung innerhalb einer Interaktion bewegt. Eine Interaktion zwischen Sprecher und Hörer kommt hierbei einem *Sprachspiel* gleich und besteht aus drei Aufgaben:

- *Konzeptualisierungsaufgabe:* sowohl Sprecher als auch Hörer bilden für jedes Objekt der Umwelt eine unverwechselbare Beschreibung;
- *Produktion:* der Sprecher ruft eine Konzeptualisierung des Hörers durch Verwendung einer gemeinsamen Menge an Signalen auf;
- *Parsing:* der Hörer nutzt diese Signale, um die gewünschte Konzeptualisierung zu rekonstruieren.

Der Sprecher erkennt eine erfolgreiche Kommunikation anhand der vom Hörer ausgeführten Handlung. Im Gegensatz dazu wird der Hörer auf eine fehlerhafte Kommunikation durch Korrekturen des Sprechers hingewiesen.
Die beiden hier aufgeführten Varianten unterscheiden sich in zwei Aspekten:

1. Die interagierenden Individuen einer schlussfolgernden Kommunikation können aus einem Signal unterschiedliche Informationen ziehen, die trotzdem zu einem gewünschten Kommunikationserfolg führen. So kann in einer Umwelt mit einem roten Ball und einer blauen Schachtel der Sprecher die Information „rot" in das Signal „alalus" umwandeln, der

Hörer rekonstruiert aus diesem Signal die Information „Ball“, was zu einer erfolgreichen Kommunikation führt.

2. Innerhalb einer schlussfolgernden Kommunikation werden die Konzeptualisierungen des Sprechers lediglich als Hinweise gesehen. Der Hörer besitzt somit einen gewissen Interpretationsspielraum, der durchaus zu Mehrdeutigkeiten innerhalb einer Kommunikation führen kann.

Ein Hauptvorteil der schlussfolgernden Kommunikation ist es also, dass die sprachlichen Bestände der an der Kommunikation beteiligten Individuen nicht identisch sein müssen.

Stufen der Evolution eines Kommunikations-Systems

Die in diesem Unterpunkt beschriebenen Stufen und Schritte spiegeln den Entwicklungsprozess einer Sprache am deutlichsten wider. In lexikalischen, also syntaxfreien Kommunikations-Systemen unterscheidet Luc Steel drei aufeinander aufbauende Stufen:

Stufe I: Das „Namengebungs-Spiel“

Die erste Stufe repräsentiert die Evolution einer Sprache auf der niedrigsten Ebene. Das in dieser Arbeit geschaffene Werkzeug LexLearn simuliert ein Kommunikations-System, welches eine auf dieser Stufe angesiedelte Sprache nutzt. Auf dieser Stufe weist jedes Individuum einer Gemeinschaft jedem Objekt der Umwelt einen *Namen* in Form eines eindeutigen Signals oder Wortes zu. Um ein gemeinsam-genutztes Lexikon aufzubauen und aufrechtzuerhalten, halten die Individuen einer Gemeinschaft laut Luc Steel folgende Vereinbarungen innerhalb der Sprachspiele ein:

- *Erfindung*: wird ein neuer Name benötigt, generiert der Sprecher ein neues Wort für ein Objekt;
- *Aneignung*: trifft ein Hörer auf ein neues Wort und ist er in der Lage das entsprechende Objekt durch ein Sprachspiel zu referenzieren, so assoziiert ab diesem Zeitpunkt immer dieses neue Wort mit diesem Objekt;
- *Anpassung*: alle Individuen sollten die Stärke ihrer Assoziationen auf den Kommunikationserfolg innerhalb einer Gemeinschaft anpassen.

Simulationen, welche Sprachen dieser Stufe erzeugen, sind wie LexLearn oftmals mit neuronalen Netzen ausgestattet.

Stufe II: Kategorisierung

In dieser Stufe der Evolution eines lexikalischen Kommunikations-Systems werden die in Stufe I erzeugten Namen durch *Kategorien* ersetzt. Demnach müssen Individuen in der Lage sein, einzelnen Objekten Kategorien wie z. B. „rot“ oder „Auto“ zuzuordnen. Um ein Repertoire an solchen Kategorien aufzubauen und

aufrechtzuerhalten, müssen die Individuen einer Gemeinschaft ähnliche Vereinbarungen wie in Stufe I einhalten:

- *Erfindung*: wird eine neue Kategorie benötigt, generiert der Sprecher eine neue Kategorie für ein Objekt;
- *Aneignung*: trifft ein Hörer auf eine neue Kategorie und ist er in der Lage das entsprechende Objekt durch ein Sprachspiel zu referenzieren, so assoziiert ab diesem Zeitpunkt immer diese neue Kategorie mit diesem Objekt;
- *Anpassung*: alle Individuen sollten die Stärke ihrer Assoziationen auf den Kommunikationserfolg innerhalb einer Gemeinschaft anpassen.

Stufe III: Komposition

Individuen eines Kommunikations-Systems, dessen Sprache sich auf dieser Stufe befindet, nutzen Kompositionen von Kategorien um ein Objekt der Umwelt zu beschreiben. Ein Signal besteht somit aus einem Satz, dessen einzelne Wörter für Kategorien stehen, was eine gewisse strukturelle Vielseitigkeit der Objekte der Umwelt voraussetzt. Um diese Stufe zu erreichen, benötigen die Agenten einer Simulation komplexere Berechnungsmechanismen. Nicolas Neubauer hat gezeigt, dass das Nutzen von Kompositionen zur Objektbeschreibung bei ausreichender Struktur der Objekte der Umwelt vorteilhaft ist (siehe [Neu03]).

Die Evolution einer Sprache über die dritte Stufe hinaus erfordert neben den lexikalischen Strukturen das Hinzufügen von *grammatikalischen* Elementen. Luc Steel sieht eine vordefinierte Syntax für eine Komposition von lexikalischen Wörtern als die einfachste aller grammatikalischen Strukturen. Die Komposition „rot“, „Ball“, „unter“ „blau“, „Tisch“ kann in einem rein lexikalischen Kommunikations-System zu mehreren, doppeldeutigen Informationen auf der Hörerseite führen. In einem grammatikalischen Kommunikations-System wird diese Komposition mit einer ersten Syntax versehen, welche festlegt, dass die Wörter „rot“ und „Ball“ bzw. „blau“ und „Tisch“ jeweils das gleiche Objekt referenzieren. Eine solche syntaktische Bindung kann beispielsweise durch das Anhängen eines gleichen Affixes erreicht werden, welches zu der Komposition „rot-ba“, „ball-ba“, „unter-un“, „blau-ti“, „Tisch-ti“ führt. Das Sprachspiel zwischen einem Sprecher und Hörer wird durch das Einführen einer Syntax wesentlich vereinfacht, da der Interpretationsprozess des Hörers von eventuellen Mehrdeutigkeiten innerhalb einer Nachricht befreit ist.
Die Entwicklung einer Syntax innerhalb einer Gemeinschaft von Individuen gleicht der Entwicklung eines Lexikons. Individuen formen somit neue syntaktische Strukturen und übernehmen die für die Kommunikation erfolgreichste Struktur, sodass sich die Gemeinschaft auf eine Syntax einigt.
Die in diesem Unterkapitel beschriebenen Evolutionsstufen der Emergenz einer Sprache geben den Anreiz das Modell von Hutchins und Hazlehurst, welches als Basis dieser Evolution gesehen werden kann, und die damit verbundene Anwendung LexLearn zu erweitern.

LexLearn - User Guide

Starting LexLearn you will find the *main window* as it is shown in figure 1.

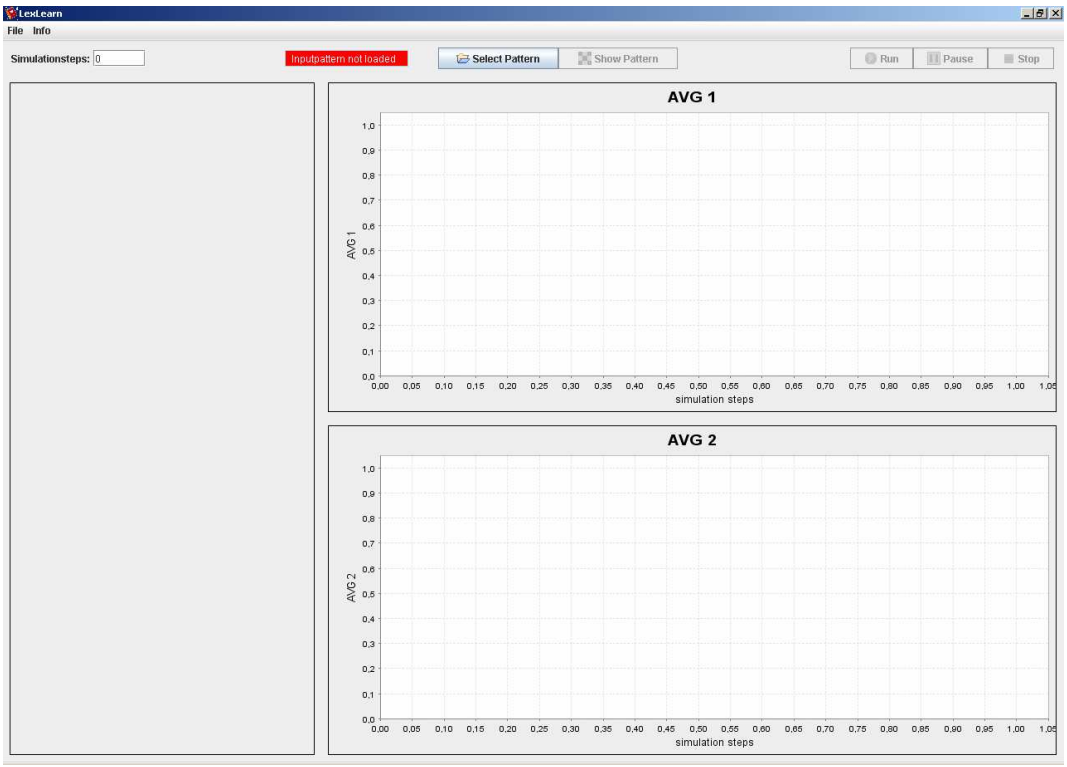


Figure 1: Main Window

LexLearn's main window is divided into six mayor components which are enumerated in figure 2 and explained in the following.

1. The first component which is placed in the upper left corner is the menu bar, consisting of a *file menu* and an *info menu*. Inside the file menu, you will find the items *New*, *Load*, *Save as* and *Exit*, allowing you to create new, load already existing or save your actual simulations and to exit the program. The info menu only contains the item *About LexLearn*, showing you again the splash screen that appears when starting the program.
2. The second component is the *simulation steps* text field. Here the number of simulation steps has to be entered by the user.
3. The third component concerns the simulation's *input pattern*. It can be selected after pushing the *Select Pattern* button and looked at after pressing the *Show Pattern* button. The coloured field left of both buttons signalizes whether an input pattern is successfully loaded (green colour) or not (red colour).
4. The fourth component controls the simulation runs. Here you can find the buttons *Start*, *Pause* and *Stop*, starting, pausing or continuing and stopping your simulation runs.

5. The fifth component which will be empty when starting LexLearn is the environments panel, listing all the simulated environments including their attributes.
6. The sixth component consists of the charts for the error measures Avg1 and Avg2. They will be dynamically updated during simulation runs.

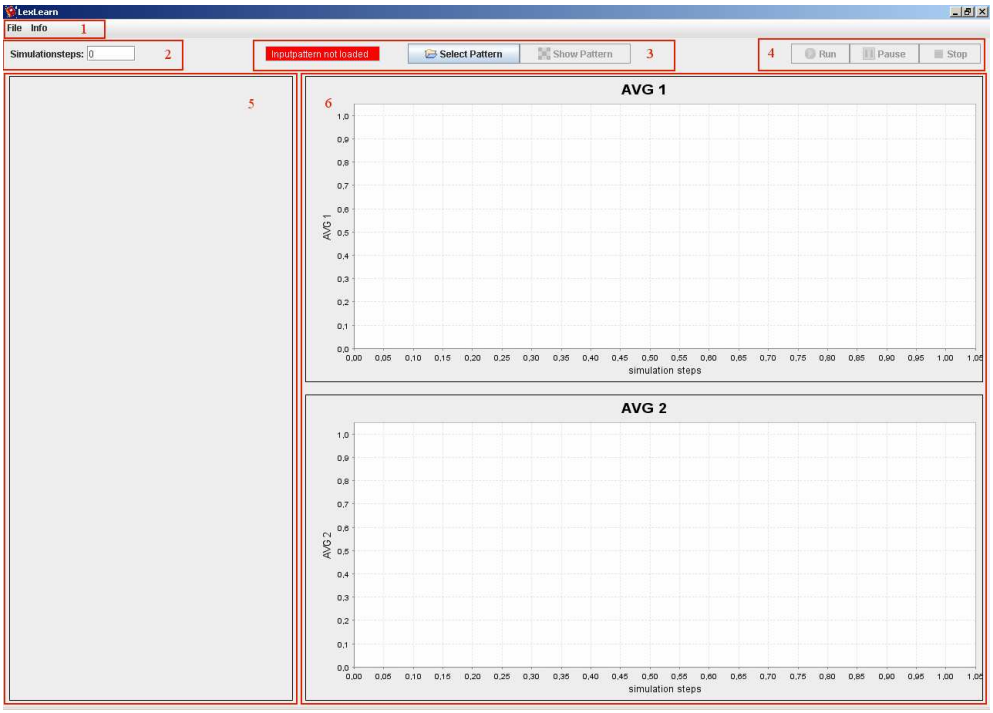


Figure 2: Main Window (Components' View)

How to create a new Simulation

New Simulations can be built using the dialog to create new simulations which can be found in the File Menu. (cf. figure 3)

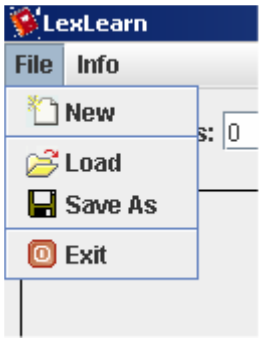


Figure 3: File Menu (Detail)

Within the appearing dialog screen which is shown in figure 4 the number of simulation environments and standard values for the number of agents, the learning

rate, the momentum and the neural network's structure, consisting of the word location and the dimensions of the network itself, have to be entered.

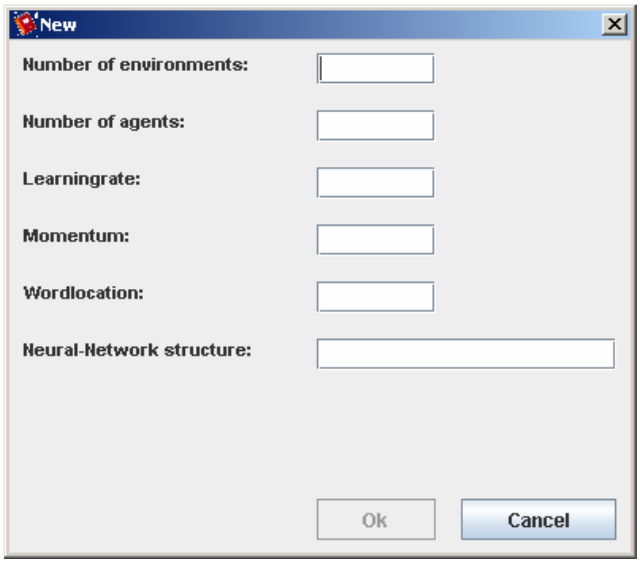


Figure 4: New Simulation Dialog at Beginning

The network structure is declared in the following way: Layers are always two-dimensional and are separated by ";". The two dimensions of each layer are separated by the character "x". No separator is necessary behind the last layer, as it is shown for one example in figure 5. The word location indicates the word representation layer. Layer numbers start from 0 which is the input layer.

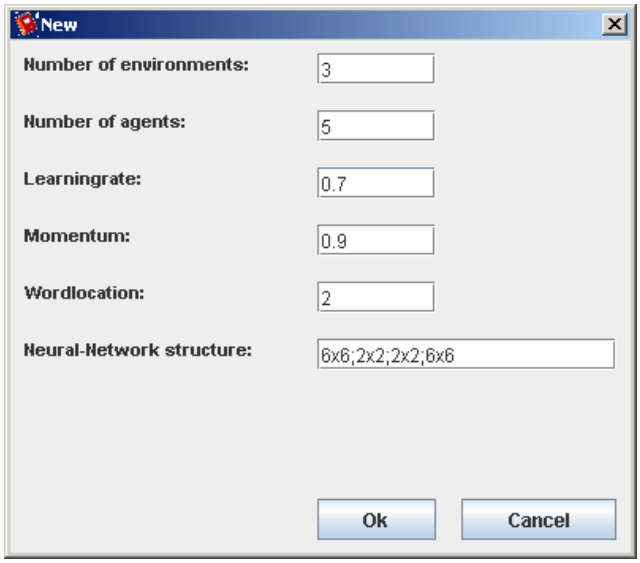


Figure 5: New Simulation Dialog with Values

The values declared here will be the standard values in all environments, but can be changed afterwards in the Environment Panel. Confirming the dialog the main window looks similar to the example in figure 6. Now, the attributes in each environment can be changed independently from all the other environments.

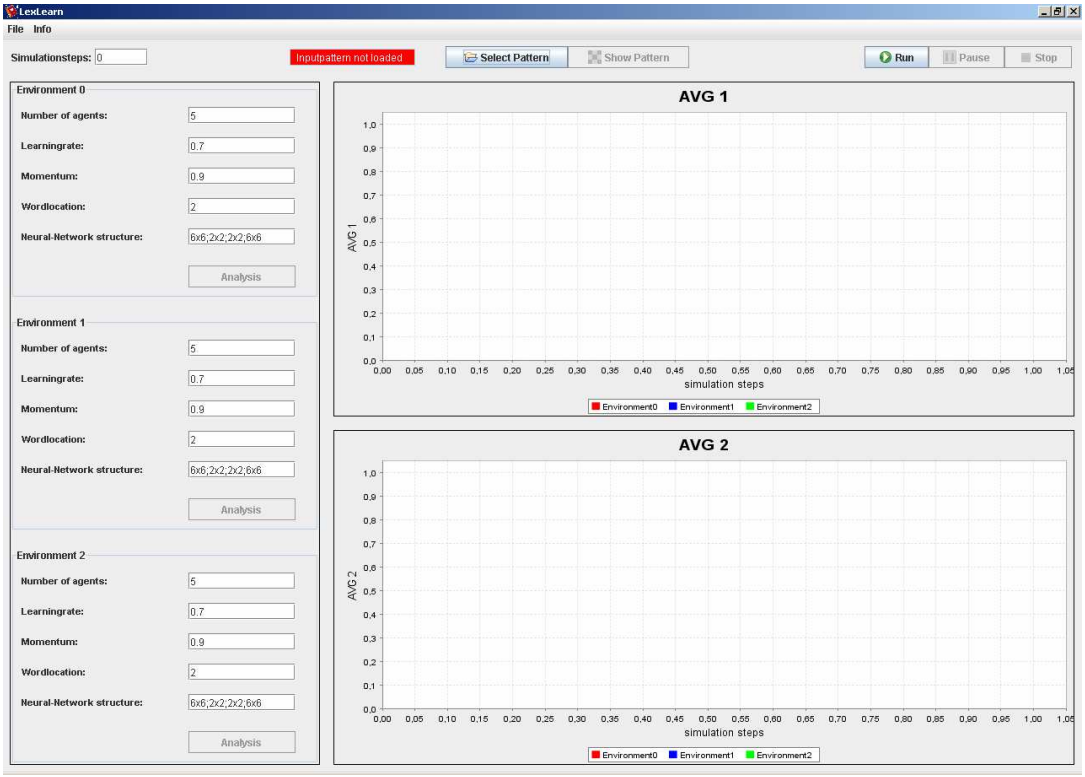


Figure 6: Main Window with Environments

The second necessary step to make a simulation run is the definition of a sequence of input patterns. Therefore a CSV-file can be selected after pressing the Select Pattern button. The structure of a CSV-file must be as follows: (cf. figure 7)

- Patterns are separated by a blank line, before the first and after the last pattern there must not be a blank line.
- One pattern is a two dimensional array of double values, separated by ";" in a line, representing the horizontal dimension (no ";" at the end of a line) and by a new line in the vertical dimension.

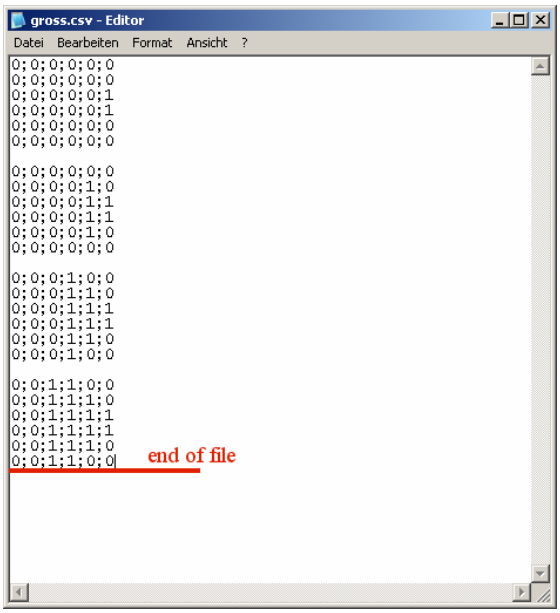


Figure 7: A Valid CSV File

Having selected a CSV File, the red field changes its colour to green (cf. figure 8) and the pattern sequence can be looked at after pressing the Show Pattern button. Then a new window pops up showing the input pattern. (cf. figure 9)

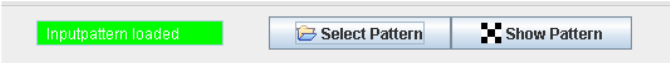


Figure 8: Pattern Selected (Detail)

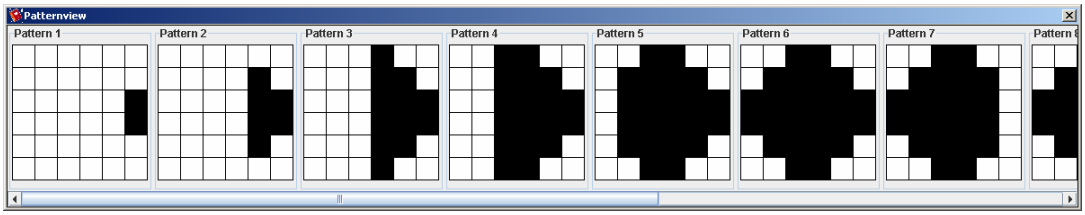


Figure 9: Pattern View

Finally only the number of simulation steps has to be entered in the simulation steps field.
Now the simulation is completely created and a simulation run can be started.

Running a Simulation

Simulations can be started by pressing the Start button. Then the agents' networks will be randomly initiated and all the simulation environments will be simulated simultaneously. While running a simulation the Avg1 and Avg2 chart panels will be updated every 1000 simulation steps. Simulations can be paused pushing the Pause

button once, and continued pressing it a second time. To stop a simulation, push the Stop button.
Being in the pause state or after a simulation run has finished it's possible to open an environments analysis window by clicking the *Analysis* button which can be found in an environment's attributes list within the environments panel. (cf. figure 10)

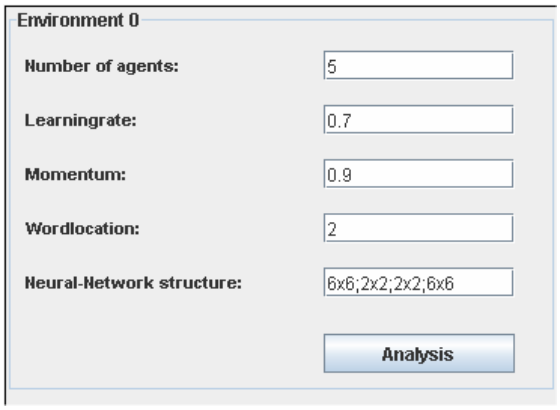


Figure 10: Environment Attribute List (Detail)

Analysis Window

The analysis window gives you detailed information about the *activation values* of the word layer and the output layer. (cf. figure 11)
Three combo boxes are placed in the upper part of the window. The left one lets you choose the input pattern which will be shown in the top middle of the window. The second combo box allows you to define whether to see the word layer's activation values, related to the chosen input pattern, in a *standard* view, showing the real values that are calculated inside the word layer, a *rounded* view, rounding those values to two decimal places or a *literal* view, mapping each activation value to a syllable in the way it is shown in figure 15. The third combo defines the way the output layer will be represented. Besides *standard* and *rounded* view a *graphical* view can be chosen here to let you see how each agent redraws the visual scene it was shown.
In the lower part you will find an agents list showing both layers' activation values, represented in the chosen way (standard, rounded, literal/graphical). Figure 11 gives you a full-screen example of the standard representation while figure 12 and figure 13 show you both, a word layer and an output layer in the rounded mode. Figure 14 shows the same word in the literal view, using the syllable matching mechanism. In figure 16 you can see an agent's output layer's graphical representation.

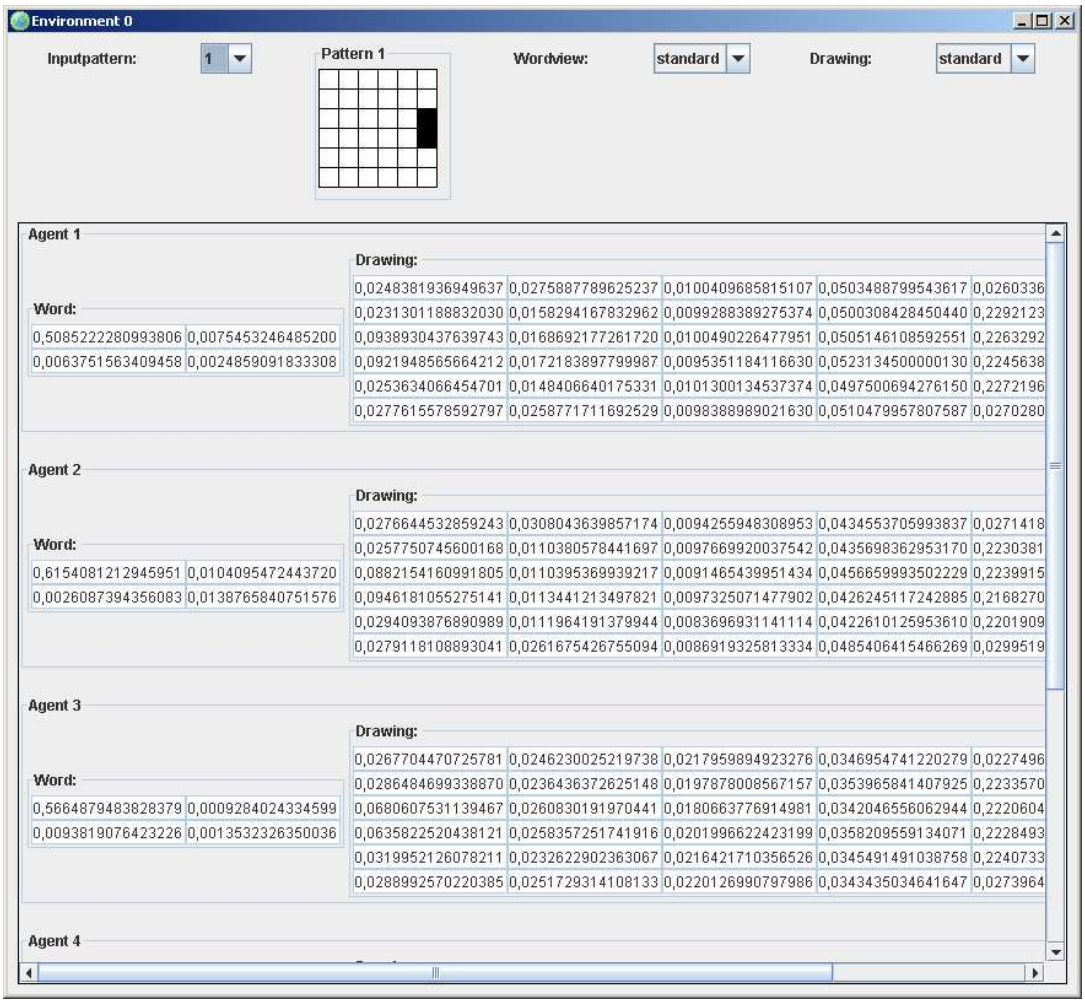


Figure 11: Analysis Window

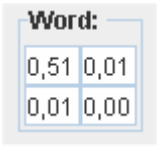


Figure 12: Word Rounded View (Detail)

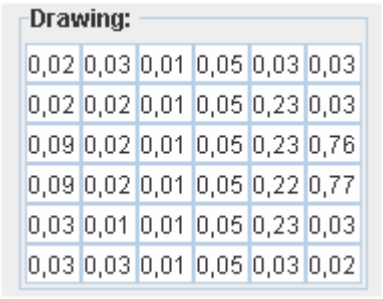


Figure 13: Drawing Rounded View (Detail)

Figure 14: Word Literal View (Detail)

```
public static String getSyllable (double in)
{
String out = new String("");

        if (in < 0.1)  out += "al"; else
        if (in < 0.2)  out += "as"; else
        if (in < 0.3)  out += "es"; else
        if (in < 0.4)  out += "el"; else
        if (in < 0.5)  out += "il"; else
        if (in < 0.6)  out += "is"; else
        if (in < 0.7)  out += "os"; else
        if (in < 0.8)  out += "ol"; else
        if (in < 0.9)  out += "ul"; else
        if (in < 1.0)  out += "us";

        return out;
}
```

Figure 15: Syllable Matching Mechanism

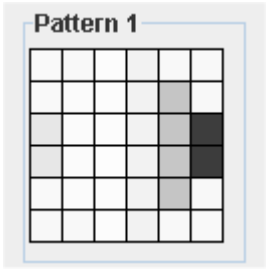


Figure 16: Drawing Graphical View (Detail)

Saving and Loading a Simulation

A simulation can be saved after clicking the save item in the File menu. In that case all the created environments with all contained attributes will be stored. The results of simulation runs themselves though, will not be held. (Note: As neural networks are randomly initiated after pressing the Start button even agents that already optimized their networks will be starting the next simulation run in the same way "untrained" agents would.)
Two steps are necessary, choosing the directory where to save the file and entering a valid name for the *.sim* file. Simulations are saved as binary files, so editing saved simulations is nearly impossible without the use of LexLearn.

For loading a *.sim* file just click the load item in the File menu. Choose a directory and a file and confirm your choice.

Further Information

For further information on LexLearn feel free to contact us:

Dennis Fuchs mopus@uni-koblenz.de
Christian Klein anaconda@uni-koblenz.de

Literaturverzeichnis

[Bal01] Balzert, Helmut: *Lehrbuch der Software-Technik*. 2. Auflage, Spektrum Akademischer Verlag, 2001.

[EB96] Ebert, C., Dumke, R.: *Software- Metriken in der Praxis: Einführung und Anwendung von Software- Metriken in der industriellen Praxis*, Springer, 1996.

[EMILA06] *EMIL – Emergence in the Loop: simulating the two way dynamics of norm innovation*, Contract: Annex I – "Description of Work", Brüssel, 2006.
URL: http://emil.istc.cnr.it/files/EMIL-AnnexI%20-%2031%20March%20rev%20-%20final_0.pdf, Stand: 09.09.2007.

[EMILD07] Troitzsch, Klaus G.: *EMIL D 3.1, Requirements that EMIL-S must meet*, Version 1.0, Koblenz, 2007.
URL: http://emil.istc.cnr.it/files/Requirements.pdf, Stand 09.09.2007.

[HH91] Hutchins, Edwin; Hazlehurst, Brian: *Learning in the Cultural Process.* In: Farmer, D.; Langton, C; Rasmussen, S.; Taylor, C.: *Artificial Life II.* Addison-Wesley, 1991.

[HH95] Hutchins, Edwin; Hazlehurst, Brian: *How to Invent a Lexicon: The Development of Shared Symbols in Interaction.* In: Gilbert, Nigel; Conte, Rosaria: *Artificial Societies: The Computer simulation of social life*. London, UCL Press, 1995.

[HS98] Huhns, M., Singh, M. P.: *Readings in Agents*, Morgan Kaufmann, 1998.

[Kru00] Krüger, Guido: *GoTo Java 2, Handbuch der Java-Programmierung*. 2. Auflage, München, Addison-Wesley, 2000.

[Lip05][20] Lippe, Wolfram-Manfred: *Soft-Computing: Mit neuronalen Netzen, mit Fuzzy-logic und evolutionären Algorithmen*. Springer, 2005.

[Neu03] Neubauer, Nicolas: *Emergence in a multiagent simulation of communicative behaviour*. In *PICS, Publication Series of the Institute of Cognitive Science*, Volume 11, University of Osnabrück, 2004.

[Rum86] Rumelhart, David E., Hinton Geoffrey G., Williams R. J.: *Learning Internal Representation by Error Propagation*. In: Rumelhart, David E., McClelland, James L.: *Parallel Distributed Processing*, Volume 1: *Foundations*, Seiten 318-362. London, MIT Press, 1986.

[20]Teile daraus sind unter http://cs.uni-muenster.de/Professoren/Lippe/lehre/skripte/wwwnnscript als interaktives Skript erhältlich.

[Rum94] Rumelhart, David E.; Widrow, Bernhard; Lehr Michael A.: *The Basis Ideas in Neural Networks*. In *Communication of the ACM*, Volume 37, Issue 3, Seiten 87-92. New York, ACM Press, 1994.

[Rus90] Russel, Beale; Jackson, Tom: *Neural Computing: An Introduction*. Bristol, CRC Press, 1990.

[Sal13] von Parma, Salimbene: *Chroniken.* In: G. G. Coulton: *St. Francis to Dante*. London: David Nutt, 1906, S. 242f.
URL: http://www.fordham.edu/halsall/source/salimbene1.html, Stand 09.08.2007

[Sap21] Sapir, Edward: *Language: An Introduction to the Study of Speech*. New York: Harcourt, Brace & World, 1949.

[STE05] Steels, Luc: *The Emergence and Evolution of Linguistic Structure: From Lexical to Grammatical Communication Systems*. In *Connection Science*, Volume 17, Issue 3 & 4, Seiten 213 – 230, Taylor & Francis, London, 2005.

[TG05] Gilbert, Nigel; Troitzsch, Klaus G.: *Simulation for the Social Scientist.* Buckingham, Philadelphia, Open University Press, 2005.