Lernen mit einer Hybride aus Lernendem Klassifizierendem System und Selbstorganisierender Karte

Dissertation
zur Erlangung des Grades
Doktor der Naturwissenschaften

am Fachbereich Physik, Mathematik und Informatik
der Johannes Gutenberg-Universität in Mainz

vorgelegt von

Thomas Hillebrand

geboren in Koblenz

Mainz, im Januar 2010

Datum der mündlichen Prüfung: 30.08.2010

D77 – Mainzer Dissertation

Dank

Eine Dissertation ist ein umfangreiches Projekt, das ohne die Unterstützung der Menschen im persönlichen und beruflichen Umfeld kaum zu bewältigen ist. Daher möchte ich an dieser Stelle allen danken, die mich dabei unterstützt haben.

Zusammenfassung

Im Forschungsgebiet der Künstlichen Intelligenz, insbesondere im Bereich des maschinellen Lernens, hat sich eine ganze Reihe von Verfahren etabliert, die von biologischen Vorbildern inspiriert sind. Die prominentesten Vertreter derartiger Verfahren sind zum einen Evolutionäre Algorithmen, zum anderen Künstliche Neuronale Netze.
Die vorliegende Arbeit befasst sich mit der Entwicklung eines Systems zum maschinellen Lernen, das Charakteristika beider Paradigmen in sich vereint: Das **Hybride Lernende Klassifizierende System** (HCS) wird basierend auf dem reellwertig kodierten eXtended Learning Classifier System (XCS), das als Lernmechanismus einen Genetischen Algorithmus enthält, und dem Wachsenden Neuralen Gas (GNG) entwickelt.
Wie das XCS evolviert auch das HCS mit Hilfe eines Genetischen Algorithmus eine Population von Klassifizierern – das sind Regeln der Form ‚`WENN` *Bedingung* `DANN` *Aktion*', wobei die Bedingung angibt, in welchem Bereich des Zustandsraumes eines Lernproblems ein Klassifizierer anwendbar ist. Beim XCS spezifiziert die Bedingung in der Regel einen achsenparallelen Hyperquader, was oftmals keine angemessene Unterteilung des Zustandsraumes erlaubt. Beim HCS hingegen werden die Bedingungen der Klassifizierer durch Gewichtsvektoren beschrieben, wie die Neuronen des GNG sie besitzen. Jeder Klassifizierer ist anwendbar in seiner Zelle der durch die Population des HCS induzierten Voronoizerlegung des Zustandsraumes, dieser kann also flexibler unterteilt werden als beim XCS. Die Verwendung von Gewichtsvektoren ermöglicht ferner, einen vom Neuronenadaptationsverfahren des GNG abgeleiteten Mechanismus als zweites Lernverfahren neben dem Genetischen Algorithmus einzusetzen. Während das Lernen beim XCS rein evolutionär erfolgt, also nur durch Erzeugen neuer Klassifizierer, ermöglicht dies dem HCS, bereits vorhandene Klassifizierer anzupassen und zu verbessern.
Zur Evaluation des HCS werden mit diesem verschiedene Lern-Experimente durchgeführt. Die Leistungsfähigkeit des Ansatzes wird in einer Reihe von Lernproblemen aus den Bereichen der Klassifikation, der Funktionsapproximation und des Lernens von Aktionen in einer interaktiven Lernumgebung unter Beweis gestellt.

Abstract

In the research area of artificial intelligence and particularly for purposes of machine learning a wide range of methods inspired by biological models has been established. Most prominent among these are Evolutionary Algorithms on the one hand and Artificial Neural Networks on the other.
The present thesis deals with the development of a machine learning system that combines characteristics of both paradigms: The **Hybrid Learning Classifier System** (HCS) is derived from the real-valued eXtended Learning Classifier System (XCS), which uses a genetic algorithm as a learning mechanism, and the Growing Neural Gas (GNG).
As well as XCS, HCS makes use of a genetic algorithm to evolve a population of classifiers – these are rules of the form ‚`IF` *condition* `THEN` *action*‘, the condition indicating in what part of a learning problem's state space a classifier is applicable. In case of the XCS a condition usually specifies an axis-parallel hyperrectangular subspace of a state space. In many learning problems this does not yield an appropriate partitioning of the state space. HCS, however, uses weightvectors – just like the ones of GNG's neurons – as classifier conditions. Thus HCS's population induces a Voronoi tessellation of the state space. Each classifier being applicable in its Voronoi cell, this allows for a much more flexible partitioning. Further, using weightvectors as classifier conditions allows for using a second learning method derived from GNG's mechanism of classifier adaptation besides the genetic algorithm. Thus HCS may adapt and improve existing classifiers while learning in XCS is purely evolutionary, relying on the creation of new and better classifiers to improve performance.
To evaluate HCS, it ist tested on various learning problems. The method's capabilities are demonstrated in several learning experiments comprising problems from the domains of classification and function approximation as well as the learning of actions in an intercative learning environment.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

Bereits in der Zeit der ersten elektronischen Rechenanlagen in den 1940er Jahren kam die Idee auf, diese nicht nur für umfangreiche Berechnungen – wie etwa bei der Erstellung ballistischer Tabellen – einzusetzen, sondern auch für Aktivitäten, die mehr „Intelligenz“ erfordern. So diskutierte etwa Alan Turing in seinem 1948 verfassten – jedoch erst 1969 veröffentlichten – Aufsatz *Intelligent Machinery*, wie Computer dazu gebracht werden könnten, intelligentes Verhalten zu zeigen [Turing 1969].
Aus dieser Idee entwickelte sich seit den späten 1950er Jahren die Wissenschaftsdisziplin *Künstliche Intelligenz* und etablierte sich als Teilgebiet der Informatik. Dieser in [Kurzweil 1990] als *„. . . the art of creating machines that perform functions that require intelligence when performed by people“* charakterisierte Forschungsbereich kann mittlerweile eine Reihe beeindruckender Erfolge vorweisen, so etwa den Sieg des Schachcomputers *Deep Blue* über den damaligen amtierenden Schachweltmeister Garri Kasparow im Jahr 1997 [IBM]. Bei genauer Betrachtung werden diese Erfolge jedoch durch die Erkenntnis relativiert, dass es sich bei ihnen im Wesentlichen um Lösungen eng umrissener Problemstellungen handelt und die zugehörigen Lösungswege zwar möglicherweise komplex, im Prinzip jedoch klar ersichtlich sind. Die Erfolge der klassischen Künstlichen Intelligenz beruhen somit hauptsächlich auf der – im Vergleich zum Menschen – höheren Rechenleistung der Computer und deren zuverlässigerem elektronischen Gedächtnis.
Eine Vielzahl anderer Probleme hingegen, bei denen die Aufgabenstellung und vor allem der Lösungsweg weniger genau spezifiziert sind, verschließen sich einer Lösung durch die klassischen Methoden der Informatik und Künstlichen Intelligenz. Interessanterweise sind dies oft gerade Aufgaben, die von Menschen und sogar Tieren relativ problemlos bewältigt werden. So finden beispielsweise schon kleine Kinder – oder auch Hunde – ohne Probleme einen Ball auf einem Rasen. Computern beziehungsweise Robotern gelingt dies bestenfalls unter sehr genau festgelegten Rahmenbedingungen; bereits leichte Abweichungen von diesen – etwa bezüglich der Lichtverhältnisse – lassen sie ziellos umher irren. Es liegt somit nahe, in der Natur nach Lern-Mechanismen zu suchen, die als Vorbild für neuartige technische Ansätze zur Lösung derartiger Probleme dienen können. Die wohl bekanntesten Vertreter solcher biologisch inspirierten Verfahren im Bereich der Künstlichen Intelligenz sind zum einen *Künstliche Neuronale Netze* und zum anderen *Evolutionäre Algorithmen*.

Künstliche Neuronale Netze sind Systeme, die Informationsverarbeitungsmechanismen tierischer und menschlicher Nervensysteme in technischer Form nachahmen. Auf diese Weise wird versucht, Computern zu ermöglichen, in ähnlicher Weise wie ein biologisches Gehirn zu lernen. Um ein Künstliches Neuronales Netz zum Lösen eines Problems zu befähigen, wird es zunächst einem Lern- oder Trainingsprozess unterzogen, in dessen Verlauf die Neuronen, das sind die „Knoten“ des Netzes, und deren Verbindungen auf Grundlage von Beispieleingaben angepasst werden, welche repräsentativ für die Menge aller Eingabedaten sind. Nach Abschluss dieses Trainings verhält sich das Neuronale Netz bei Konfrontation mit einer unbekannten Eingabe so, wie es dies bei einer ähnlichen bekannten Eingabe tun würde, und ist damit in der Lage, auch auf nicht explizit trainierte Eingaben angemessen zu reagieren.
Gehirne und Nervensysteme – die Vorbilder Künstlicher Neuronaler Netze – gehören zu den komplexesten der Menschheit bekannten Strukturen. Entstanden sind sie, als Bestandteile heute existierender Lebensformen, durch natürliche Evolution. Diese wird – darüber herrscht in der Biologie heute weitgehende Einigkeit – durch die Darwin'sche Evolutionstheorie zutreffend als Wechselspiel von Selektion und Variation beschrieben: Besser an ihre Umwelt angepasste Individuen haben größere Chancen, sich fortzupflanzen (Selektion) und ihr Erbgut an die nächste Generation weiterzugeben. Im Zuge dieser Weitergabe erfolgt eine Variation, zum einen, indem das Erbgut der Eltern bei der Weitergabe an die gemeinsamen Nachkommen „vermischt“ wird, zum anderen durch zufällige Mutationen. Die Nachkommen stellen somit keine exakten Kopien, sondern Varianten ihrer Eltern dar. Da unter den Nachkommen wiederum die am besten angepassten einen Selektionsvorteil haben, setzen sich gute Varianten im Laufe vieler Generationen durch. Evolutionäre Algorithmen ahmen diese Mechanismen in stark vereinfachter und abstrahierter Form nach und nutzen sie, um komplexe (Optimierungs-) Probleme zu lösen: Ausgehend von zufällig generierten Lösungskandidaten werden in einem iterativen Prozess die jeweils besten Lösungen durch bevorzugte Vermehrung sowie zufällige Variation und Kombination weiterentwickelt; schlechtere Lösungen „sterben aus“.

Sowohl das Gebiet der Evolutionären Algorithmen als auch das der Neuronalen Netze haben Ansätze zur Klassifikation von Daten hervorgebracht. Zwei dieser Ansätze – je einer aus jedem der beiden Bereiche – bilden den Hintergrund der vorliegenden Arbeit und werden in den folgenden beiden Kapiteln vorgestellt.
Das Thema von Kapitel 2 sind *Lernende Klassifizierende Systeme*, eine spezielle Form Evolutionärer Algorithmen, wobei insbesondere auf das derzeitige Standardsystem – das *eXtended Learning Classifier System (XCS)* – eingegangen wird. Bei diesen Systemen erfolgt die Klassifikation eines Eingabedatums explizit: es wird eine von mehreren möglichen Aktionen ausgegeben. Die Auswahl der jeweils angemessenen Aktion stützt sich auf eine aus sogenannten Klassifizierern bestehende Wissensbasis. Ein Klassifizierer enthält eine Bedingung und schlägt als Reaktion auf Eingaben, die diese erfüllen, eine bestimmte Aktion vor; zudem macht er eine Vorhersage über die Belohnung, die das System bei deren Ausführung von der Lernumgebung zu erwarten hat. Erfüllt eine Eingabe die Bedingungen mehrerer Klassifizierer, werden deren Vorhersagen bei der Auswahl der auszuführenden Aktion berücksichtigt. Die Auswahl der jeweils optimalen Aktion ist somit nur möglich, wenn die Vorhersagen der Klassifizierer zumindest annähernd zutreffend sind. Um dies zu erreichen, werden einerseits die Vorhersagen ausgehend von den tatsächlich beobachteten Belohnungen adaptiert, andererseits wird aber auch die Wissensbasis des Systems evolviert, um die Bedingungen der Klassifizierer an die Struktur der Lernumgebung anzupassen – die Bedingung eines Klassifizierers sollte idealerweise nur von Eingaben erfüllt werden, für die dessen Aktion zu (ungefähr) glei-

chen Belohnungen führt. Die künstliche Evolution wirkt somit auf eine hinsichtlich der Aktionswahl sinnvolle Klassifikation hin.
Kapitel 3 befasst sich mit sogenannten *Selbstorganisierenden Karten*, einer Klasse Künstlicher Neuronaler Netze. Diese stellen ein abstraktes Modell somatotopischer Karten dar, das sind Nervenstrukturen des Gehirns, in denen ähnliche äußere Reize zu einer Erregung benachbarter Nervenzellen führen. In Analogie hierzu lernen Selbstorganisierende Karten – sofern gewisse Voraussetzungen erfüllt sind – eine nachbarschaftserhaltende Abbildung von Eingabedaten auf Neuronen. Dazu werden – basierend auf den auftretenden Eingaben – die den Neuronen einer Selbstorganisierenden Karte zugeordneten Positionen im Eingaberaum während eines Trainingsprozesses angepasst. Derart bildet die Karte eine Struktur aus, durch die eine implizite Klassifikation der Eingabedaten gegeben ist: Jedes Neuron wird als Repräsentant der Klasse aller auf es abgebildeten Eingaben aufgefasst.
Die von einer Selbstorganisierenden Karte entwickelte Struktur wird im Wesentlichen durch die Häufigkeitsverteilung der Eingabedaten bestimmt. Dies bedeutet einerseits, dass der Lernvorgang ohne Belohnungen und Beurteilungen auskommt und unüberwacht erfolgen kann, andererseits heißt es aber auch, dass kaum Einfluss auf die Kriterien der Klassifikation genommen werden kann. Soll, wie es in vielen Lernumgebungen der Fall ist, basierend auf der vorgenommenen Klassifikation der Eingabedaten eine Aktionswahl erfolgen – dies erfordert die Nachschaltung eines geeigneten Mechanismus, etwa eines Reinforcement-Learning-Verfahrens – ist somit nicht sicher gestellt, dass die durch die Struktur der Selbstorganisierenden Karte implizierte Klassifikation eine sinnvolle Grundlage dieser Aktionswahl bildet. Dies stellt einen klaren Nachteil gegenüber der expliziten Klassifikation der Lernenden Klassifizierenden Systeme dar. Ein großer Vorteil Selbstorganisierender Karten besteht darin, dass sie sehr viel flexiblere Generalisierungen ermöglichen als Lernende Klassifizierende Systeme: Sowohl Neuronen als auch Klassifizierer sprechen im Allgemeinen auf mehrere Eingaben an. Dabei hat jedoch die Menge von Eingaben, die die Bedingung eines Klassifizierers erfüllen, üblicherweise eine sehr einfache Gestalt – bei reellwertigen Eingaben handelt es sich meist um einen Hyperquader im Eingaberaum. In vielen Fällen können mit solch einfachen Bedingungen die durch die jeweilige Lernumgebung gegebenen Möglichkeiten zur Generalisierung nicht voll ausgeschöpft werden. Demgegenüber zeichnen sich die Neuronen Selbstorganisierender Karten durch sehr flexible „Einzugsbereiche“ aus – sie können nahezu beliebige konvexe Teilmengen des Eingaberaums abdecken.
Motiviert durch die genannten Vor- und Nachteile der beiden Ansätze wird in Kapitel 4 ein *Hybrides Lernendes Klassifizierendes System (HCS)* entworfen, das die flexiblen Einzugsbereiche der Neuronen mit der expliziten Klassifikation der Klassifizierer verbindet. Ferner integriert dieses System sowohl das evolutionäre Lernen der Lernenden Klassifizierenden Systeme wie auch den Netz-Lernmechanismus Selbstorganisierender Karten. Neben dem evolutionären Lernen, bei dem die Klassifizierer unveränderlich sind und Verbesserungen nur im Rahmen der Erzeugung neuer Klassifizierer erfolgen können, wird somit ein zweites Lernverfahren eingebunden, das es ermöglicht, bereits existierende Klassifizierer zu modifizieren.
Für die Untersuchung des HCS wurde im Rahmen der vorliegenden Arbeit eine Simulations- und Visualisierungssoftware entwickelt, die interaktive Experimente ebenso ermöglicht wie die automatisierte Durchführung von Versuchsreihen. Dieser Simulationsumgebung widmet sich Kapitel 5, es spannt den Bogen von einem Überblick über die an den Simulator gestellten Anforderungen über das dem Entwurf der Software zugrunde liegende Konzept bis hin zu einer kurzen Einführung in das Nutzungskonzept und die Verwendung des Programmes.

Unterstützt von dieser Simulationsumgebung wurden zur Evaluation des HCS Experimente in verschiedenen Lernumgebungen durchgeführt, wobei Vertreter jedes der drei klassischen Einsatzgebiete Lernender Klassifizierender Systeme berücksichtigt wurden. Die Ergebnisse dieser Experimente werden in den Kapiteln 6 bis 8 präsentiert:
In Kapitel 6 werden zunächst Klassifikations-Probleme betrachtet, das sind Lernumgebungen, bei denen die präsentierten Lernumgebungszustände einer von mehreren Klassen zuzuordnen sind.
Im Anschluss werden in Kapitel 7 Funktionsapproximations-Probleme betrachtet, bei denen die Aufgabe eines Lernenden Klassifizierenden Systems darin besteht, ausgehend von den präsentierten Eingaben und den diesen zugehörigen – als Belohnungen übermittelten – Funktionswerten zu lernen, eine vorgegebene, auf dem Eingaberaum definierte Funktion zu approximieren.
Schließlich wird in Kapitel 8 das HCS zum Lernen von Aktionen eingesetzt. Beim Aktionenlernen agiert das Lernende Klassifizierende System in sogenannten Mehrschritt-Lernumgebungen, bei denen – anders als bei den beiden zuvor betrachteten Problemklassen – die präsentierten Eingaben sowohl voneinander als auch von den vom System gewählten Aktionen abhängig sind. Das Lernen in derartigen Lernumgebungen ist insofern schwieriger, als Belohnungen meist erst zeitlich verzögert als Resultat einer ganzen Reihe von Aktionen auftreten. Sie sind somit nur sehr schwer einzelnen Klassifizierern zuzuordnen, wodurch deren Beurteilung erschwert wird.
Aus jeder dieser drei Problemklassen wurden mehrere Vertreter ausgewählt, in denen sowohl mit dem HCS als auch mit einer Neuimplementation des XCS Lernexperimente durchgeführt wurden. Die Ergebnisse dieser Experimente werden verglichen und diskutiert und ferner zur Beurteilung verschiedener Aspekte des Lernverhaltens des HCS herangezogen.
Mit einer Zusammenfassung der erzielten Ergebnisse schließt Kapitel 9 die Arbeit ab. Ferner werden Ansatzpunkte für weiter gehende Forschungen benannt, die einen Ausblick auf mögliche Weiterentwicklungen des HCS gewähren.

Kapitel 2

Lernende Klassifizierende Systeme

Lernende Klassifizierende Systeme versuchen, einen Satz von Regeln zur Bewältigung einer spezifischen Aufgabe zu finden, beispielsweise für die Klassifikation von Eingabedaten oder für optimales Verhalten in einer vorgegebenen Umwelt. Sie verwenden dazu Genetische Algorithmen, ein von der biologischen Evolution inspiriertes Optimierungsverfahren, das in Abschnitt 2.1 näher erläutert wird. Diese *evolutionäre* Suche nach optimalen Regeln wird von Belohnungen respektive Bestrafungen, die das System von der Lernumgebung erhält, geleitet. Daher werden Lernende Klassifizierende Systeme den Reinforcement-Learning-Verfahren zugerechnet. Abschnitt 2.2 charakterisiert das Reinforcement-Learning-Szenario und stellt einige verbreitete Reinforcement-Learning-Verfahren vor. Damit sind dann alle Voraussetzungen geschaffen, um in Abschnitt 2.3 Lernende Klassifizierende Systeme einzuführen. In Abschnitt 2.4 wird das eXtended Classifier System (XCS) vorgestellt, das als Basis des in dieser Arbeit zu entwickelnden Hybridsystems dient. Abschnitt 2.5 fasst die wesentlichen Inhalte des Kapitels kurz zusammen.

2.1 Genetische Algorithmen

Optimierungsprobleme im weitesten Sinne treten in allen Bereichen des menschlichen Lebens auf und entsprechend vielfältig sind die zu ihrer Lösung entwickelten Methoden. Traditionell wird zwischen analytischen, enumerativen und zufallsbasierten Verfahren unterschieden. Jeder dieser Ansätze weist Nachteile auf, die seine Anwendbarkeit einschränken: Analytische Verfahren stellen hohe Anforderungen an die Struktur des zu lösenden Problems; enumerative und zufallsbasierte Verfahren sind nur dann sinnvoll einsetzbar, wenn die Menge der potentiellen Lösungen klein ist [Goldberg 1989].
Demgegenüber offenbart die biologische Evolution in der ungeheuren Vielfalt der Eigenschaften und Fähigkeiten, die Lebewesen entwickelt haben, um in einer komplexen Umwelt selbst unter den ungünstigsten Bedingungen überleben zu können, ein gewaltiges, umfassendes Problemlösungspotential.

Es ist daher nicht verwunderlich, dass schon bald nach Entwicklung der ersten elektronischen Rechner die Idee aufkam, deren Rechenleistung mit den Prinzipien der biologischen Evolution zu kombinieren, um (nahezu) beliebige Optimierungsprobleme durch einen simulierten Evolutionsprozess zu lösen[1].
Aus dieser Idee entwickelte sich seit den 1960er Jahren das Forschungsgebiet *Evolutionärer Algorithmen*, dem zum Beispiel die *Evolutionsstrategien* [Rechenberg 1973], das *Evolutionäre Programmieren* [Fogel u. a. 1965] und das *Genetische Programmieren* [Koza 1992] zuzurechnen sind. Der wohl bekannteste Ansatz dieses Forschungsbereiches sind jedoch die von John H. Holland in den 1970er Jahren entwickelten *Genetischen Algorithmen* [Holland 1992], die in diesem Abschnitt eingeführt werden.

2.1.1 Biologische Motivation

Im Folgenden werden jene Aspekte der biologischen Evolutionstheorie kurz umrissen, denen die zentralen Konzepte Evolutionärer Algorithmen entlehnt sind. Für eine detailliertere Darstellung sei auf die biologische Fachliteratur – etwa [Campbell 1997] – verwiesen.

Die Darwin'sche Theorie der Evolution

Die Theorie der Evolution, die der britische Naturforscher Charles Darwin (1809-1892) in seinem Werk *The Origin of species* [Darwin 1859] darlegte, hat sich – gestützt durch biologische Beobachtungen und paläontologische Funde – zur heutigen Standardtheorie der Evolution weiterentwickelt. Den Kern der Darwin'schen Theorie bilden die folgenden Beobachtungen und Folgerungen [hier zitiert nach Campbell 1997]:

Beobachtung 1 Das Fortpflanzungspotential jeder Art ist so hoch, dass ihre Populationsgröße exponentiell anwachsen würde, wenn alle Individuen, die geboren werden, sich erfolgreich fortpflanzen würden.

Beobachtung 2 Die Populationsgrößen sind normalerweise stabil – mit Ausnahme saisonaler Schwankungen.

Beobachtung 3 Die natürlichen Ressourcen sind beschränkt, die Umwelt kann also nur eine beschränkte Anzahl von Individuen unterhalten.

Beobachtung 4 Die Individuen einer Population variieren in ihren Merkmalen; keine zwei Individuen sind exakt gleich.

Beobachtung 5 Ein großer Teil dieser Variabilität ist erblich.

Folgerung 1 Zwischen den Individuen einer Population findet ein Kampf ums Überleben statt; in jeder Generation überlebt nur ein Teil des Nachwuchses.

Folgerung 2 Das Überleben in diesem Kampf beruht nicht auf Zufall, sondern hängt unter anderem von der erblichen Konstitution der überlebenden Individuen ab. Je besser ein Individuum durch seine ererbten Merkmale an die Umwelt angepasst ist, desto mehr Nachkommen wird es wahrscheinlich haben (*survival of the fittest*).

Folgerung 3 Die ungleichen Überlebens- und Fortpflanzungsfähigkeiten der Individuen führen zu einem allmählichen Wandel in einer Population, wobei sich vorteilhafte Merkmale im Laufe der Generationen anhäufen und durchsetzen.

[1] Alan M. Turing schlug bereits in einem 1948 verfassten – jedoch erst 1969 veröffentlichten – Artikel ein derartiges Suchverfahren vor: *„There is the genetical or evolutionary search by which a combination of genes is looked for, the criterion being survival value."* [Turing 1969]

Darwins Theorie beschreibt Evolution somit als Ergebnis eines ständigen Wechselspiels zwischen Selektion und Variation: Je besser ein Individuum an seine Umwelt angepasst ist, desto öfter wird es Nachkommen erzeugen und somit seine Eigenschaften weitergeben (Selektion). Bei dieser Weitergabe erfolgt eine Variation dieser Eigenschaften, die die Anpassung der Nachkommen an die Umwelt (positiv oder negativ) beeinflusst. Positive Varianten erhalten einen Selektionsvorteil, sodass sie sich im Laufe der Generationen durchsetzen.
Dem Wissensstand seiner Zeit entsprechend[2], betrafen Darwins Aussagen nur das sichtbare Erscheinungsbild von Individuen; über die der Vererbung zugrunde liegenden Mechanismen macht die Darwin'sche Theorie keine Aussage.

Molekulargenetische Grundlagen der Evolution

Die Grundlage der Molekulargenetik bildet die Erkenntnis, dass die äußere Erscheinung eines Lebewesens, der Phänotyp, aus seiner genetischen Ausstattung, dem Genotyp, resultiert. Als Träger der genetischen Information wurde in den 1940er Jahren die DNA (Desoxyribonukleinsäure – engl: desoxyribonucleic acid) identifiziert. Beginnend mit der Analyse der Struktur der DNA durch James Watson und Francis Crick im Jahr 1953 konnten nach und nach jene Mechanismen identifiziert werden, durch die Erbinformation von einem Lebewesen an seine Nachkommen weitergegeben wird. Diese Mechanismen liefern die molekulargenetische Erklärung für die von Darwin postulierten und phänomenologisch beobachteten Vorgänge von Vererbung und dabei auftretender Variation.
DNA-Moleküle haben die Form einer aus zwei antiparallelen Molekülsträngen zusammengesetzten Doppelhelix; die beiden Teilstränge sind durch „Brücken" (Basenpaare) aus je zwei Kernbasen (Nukleotide) miteinander verbunden. Die vier Kernbasen Adenin, Guanin, Cytosin und Thymin treten stets nur in den Paarungen Adenin-Thymin und Cytosin-Guanin auf; diese Komplementarität ermöglicht es, aus einem Teilstrang den kompletten DNA-Doppelstrang zu rekonstruieren und gewährleistet somit eine gute Replizierbarkeit der DNA.
Einzelne – meist einige hundert Basenpaare umfassende – Abschnitte der DNA, die etwa die Struktur eines Proteins codieren, werden als Gene bezeichnet. Verschiedene Varianten eines Gens werden Allele genannt.
Als Genom wird die Gesamtheit der Erbinformation eines Lebewesens bezeichnet, die bei den meisten höheren Lebewesen nicht aus einem einzelnen DNA-Doppelstrang besteht, sondern von mehreren Chromosomen[3] gebildet wird. Bei vielen Lebewesen liegen diese in jeweils mehrfacher Ausfertigung vor, so sind zum Beispiel alle Wirbeltiere diploid – jedes Chromosom liegt paarweise vor[4].
Die von Darwin beobachtete Weitergabe von phänotypischen Merkmalen eines Individuums an dessen Nachkommen ergibt sich aus der vollständigen – etwa bei der Zellteilung – oder teilweisen – bei der sexuelle Fortpflanzung – Weitergabe der Erbinformation; die bei dieser Weitergabe auftretenden Variationen lassen sich durch auf molekularer Ebene angesiedelte Mechanismen erklären:

[2] Die von Johann Gregor Mendel (1822-1884) entdeckten und nach ihm benannten Gesetze erlaubten zwar Rückschlüsse auf die der Vererbung zugrunde liegenden Mechanismen, blieben jedoch weitgehend unbekannt und wurden erst zu Beginn des 20. Jahrhunderts wiederentdeckt.

[3] Ein Chromosom besteht aus einem DNA-Molekül und einer „Verpackung" aus Proteinen.

[4] Die beiden geschlechtsbestimmenden *Gonosomen* weisen zwar – bei jeweils einem Geschlecht – strukturelle Unterschiede auf, werden üblicherweise aber dennoch als Paar betrachtet.

Mutationen sind zufällige Veränderungen an der DNA, die unter anderem durch chemische Einflüsse, Radioaktivität oder UV-Strahlung ausgelöst werden können. Mögliche Formen sind beispielsweise Austausch (Punktmutation), Einfügen (Insertion) oder Auslassen (Deletion) eines oder mehrerer Nukleotide während einer Replikation der DNA.

Rekombination bezeichnet die Vermischung von Erbinformation auf der Ebene der Chromosomensätze: Sich sexuell fortpflanzende diploide Lebewesen bilden haploide, nur einen einfachen Chromosomensatz enthaltende, Keimzellen (Samen- respektive Eizellen). Durch Verschmelzung einer Samen- mit einer Eizelle entsteht ein Kindindividuum mit wieder diploidem Chromosomensatz, der je zur Hälfte von einem Elter stammt.

Crossing-Over stellt ebenfalls eine Vermischung von Erbinformation dar: Bei einem diploiden Individuum kommt es zum Austausch von Teilen zwischen dessen beiden Chromosomensätzen, bevor diese getrennt werden, um haploide Keimzellen zu bilden.

Die biochemischen Details dieser Prozesse sind höchst kompliziert und zum Teil noch nicht vollständig verstanden; ihre prinzipielle Funktionsweise jedoch ist so eingängig, dass diese Mechanismen – in stark abstrahierter Form – die Entwicklung der Evolutionären Algorithmen, insbesondere der Genetischen Algorithmen, geprägt haben.

2.1.2 Optimierungsprobleme

Die intuitive Vorstellung von Optimierungsproblemen subsumiert eine große Bandbreite verschiedenster Aufgabenstellungen; Michalewicz [Michalewicz 1996] geht sogar soweit, jedes denkbare Problem als Optimierungsproblem aufzufassen:

> *In general, any abstract task to be accomplished can be thought of as solving a problem, which, in turn, can be perceived as a search through a space of potential solutions. Since we are after „the best" solution, we can view this task as an optimization process.*

Hier jedoch wird der Begriff des Optimierungsproblems enger gefasst, indem von einem solchen die folgenden Eigenschaften gefordert werden:

(i) Der *Suchraum* Ω, die Menge aller möglichen Lösungen, ist in einer geeigneten Weise (Enumeration, Angabe definierender Eigenschaften, ...) spezifiziert, es ist also bekannt, wie die möglichen Lösungen beschaffen sind.

(ii) Die Qualität eines Lösungskandidaten kann durch einen einzelnen skalaren Wert beschrieben werden und es ist eine *Bewertungsfunktion* $f : \Omega \to \mathbb{R}$ bekannt, die jedem Lösungskandidaten diesen *Gütewert* zuweist.

Um ein derartiges Optimierungsproblem zu lösen, muss somit die Menge

$$O(f) = \left\{\omega \in \Omega |\ \forall \omega' \in \Omega : f(\omega) \succeq f(\omega')\right\}$$

der globalen Optima von f auf Ω gefunden werden; die Vergleichsrelation $\succ \in \{<, >\}$ gibt an, ob f auf Ω minimiert oder maximiert werden soll.
Für zwei Lösungskandidaten $\omega, \omega' \in \Omega$ wird definiert:

$$\omega \succ \omega' \ :\Longleftrightarrow\ f(\omega) \succ f(\omega')$$

Ist $\omega \succ \omega'$, so heißt ω *besser* als ω'.

Die folgenden Abschnitte befassen sich mit Genetischen Algorithmen zur Lösung derartiger Optimierungsprobleme. Darüber hinaus wurden auch für viele Optimierungsprobleme, die die obigen Eigenschaften nicht aufweisen, spezielle Evolutionäre Algorithmen entwickelt; etwa die *Multikriteriellen Genetischen Algorithmen* (siehe z.B. [Deb 2001]), die auf Probleme spezialisiert sind, bei denen die Qualität einer Lösung von mehreren Kriterien abhängt und nicht durch nur *eine* Bewertungsfunktion erfasst werden kann. Auf diese Spezialformen kann im Rahmen dieser Arbeit jedoch nicht eingegangen werden.

2.1.3 Struktur Evolutionärer Algorithmen

Wie bereits erwähnt, beschreibt Darwin Evolution als ein ständiges Wechselspiel von Selektion und Variation, das zur Entwicklung immer besser angepasster Individuen führt. Ein derartiger Zyklus – ergänzt um Initialisierung und ein Termininierungskriterium – bildet auch die in Abbildung 2.1 gezeigte Grundstruktur Evolutionärer Algorithmen.

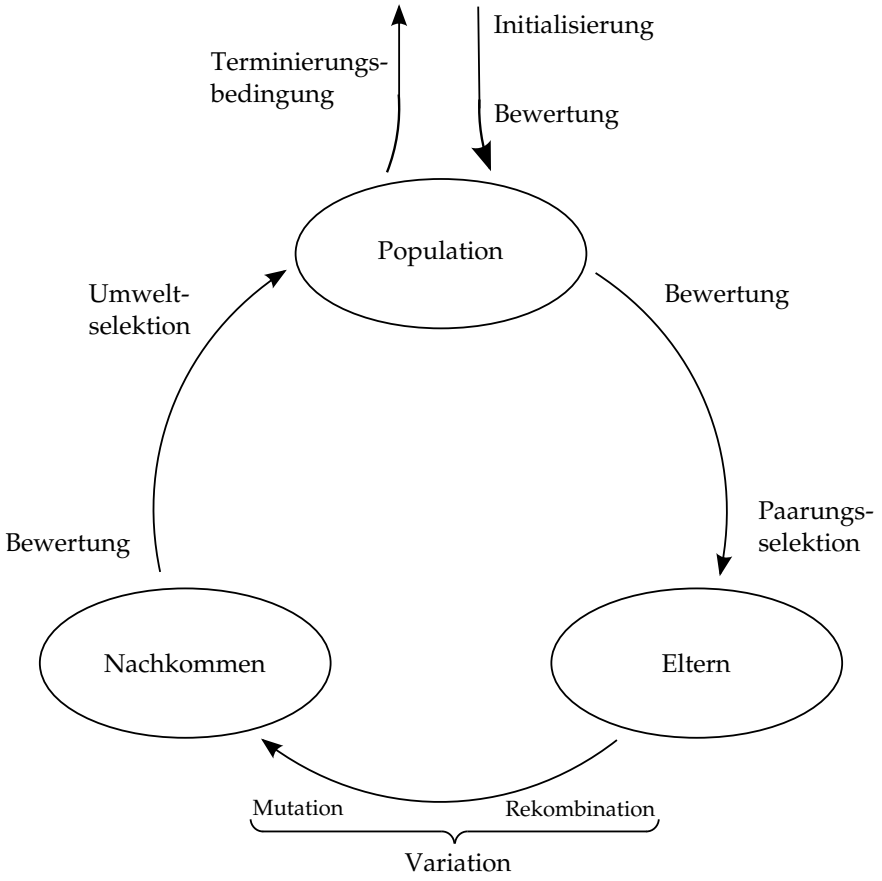


Abbildung 2.1: Schematische Darstellung des evolutionären Zyklus Evolutionärer Algorithmen. (Nach [Weicker 2002], verändert.)

Die Grundidee, eine Menge von möglichen Lösungen eines Optimierungsproblems, die *Population*, einer simulierten Evolution zu unterziehen, um so immer bessere Lösungen zu finden, ist allen Typen Evolutionärer Algorithmen gemein. Die Unterschiede der verschiedenen Ansätze zeigen sich in den Details der Modellierung der einzelnen Komponenten und Operatoren. Von zentraler Bedeutung ist ferner die verwendete Darstellung (Repräsentation) potentieller Lösungen.

2.1.4 Komponenten Genetischer Algorithmen

Im Folgenden wird näher auf die Umsetzung der Grundstruktur Evolutionärer Algorithmen im Falle der Genetischen Algorithmen eingegangen, wobei an dieser Stelle natürlich keine umfassende Darstellung Genetischer Algorithmen erfolgen kann; insbesondere theoretische Aspekte werden hier bewusst ausgeklammert. Weitere Ausführungen finden sich zum Beispiel in [Goldberg 1989], [Eiben u. Smith 2003], [Weicker 2002], [Gerdes u. a. 2004], [Michalewicz 1996] oder [Mitchell 1996].

Repräsentation

Die evolutionäre Suche Genetischer Algorithmen erfolgt im Allgemeinen nicht im Suchraum Ω, sondern in einer Menge $\mathcal{G}$, deren Elemente die in Ω enthaltenen Lösungskandidaten repräsentieren. Um das biologische Vorbild zu betonen, wird $\mathcal{G}$ als *Genraum* und Ω als *Phänotyp-Raum* bezeichnet. Die Elemente von $\mathcal{G}$ werden als *Genotypen* oder *Chromosome* bezeichnet, jene von Ω als *Phänotypen*.
Traditionell verwenden Genetische Algorithmen eine binäre Repräsentation: Lösungen werden als binäre Zeichenketten einer geeignet gewählten Länge n dargestellt:

$$g_1 g_2 \dots g_n \in \{0,1\}^n =: \mathcal{G}$$

Die binäre Repräsentation hat eine Reihe von Vorteilen: Sie ist relativ universell einsetzbar, erlaubt eine einfache Struktur der genetischen Operatoren und ihre Eigenschaften sind theoretisch gut untersucht. Andererseits führt die bei vielen kontinuierlichen Problemen geforderte Genauigkeit schnell zu langen Genotypen und somit zu einem sehr großen Genraum ($|\mathcal{G}| = 2^n$), wodurch die Leistung des Genetischen Algorithmus negativ beeinflusst werden kann.
Daher werden – gerade im Anwendungsbereich – oft auch reellwertige Repräsentationen gewählt, Genotypen also als Tupel reeller Zahlen aus $\mathcal{G} \subset \mathbb{R}^n$ dargestellt.

$$(g_1, g_2, \dots, g_n) \in \mathcal{G} \subset \mathbb{R}^n$$

Die einzelnen Stellen der Chromosome (Binärbits respektive Tupeleinträge) werden als *Gene* bezeichnet, ihre verschiedenen Ausprägungen als *Allele*.
Nach der Wahl einer Repräsentation muss eine Kodierung angegeben werden, die die Elemente von Ω auf $\mathcal{G}$ abbildet und den folgenden Anforderungen genügt:

- Um zu garantieren, dass das gesuchte Optimum gefunden werden kann[5], muss jeder Lösungskandidat $\omega \in \Omega$ durch (mindestens) einen Genotypen $g \in \mathcal{G}$ repräsentiert werden.
- Jeder Genotyp $g \in \mathcal{G}$ muss einem Lösungskandidaten $\omega \in \Omega$ entsprechen, um sicherzustellen, dass die Variationsoperatoren keine Genotypen erzeugen, die keine Entsprechung in Ω haben.

Potentielle Lösungen eines Optimierungsproblems werden im Kontext Genetischer Algorithmen oft als *Individuen* bezeichnet. Da ein Phänotyp durchaus durch mehrere Genotypen repräsentiert werden kann, wird ein Individuum in erster Linie durch seinen Genotyp gekennzeichnet, weswegen der Begriff des Individuums nicht gleichbedeutend mit dem im Problemkontext verwendeten Begriff des Lösungskandidaten ist. Letzterer ist vielmehr ein Synonym für den Phänotypen.

[5] Dies ist natürlich keine Garantie, dass es tatsächlich gefunden wird.

Bewertungs- oder Fitnessfunktion

Während in der natürlichen Evolution die „Güte“ eines Lebewesens implizit durch die Zahl seiner Nachkommen gegeben ist, müssen Genetische Algorithmen die Qualität ihrer Individuen explizit bestimmen. Dies ist die Aufgabe der – in Anlehnung an das von Darwin postulierte *survival of the fittest* – sogenannten *Fitnessfunktion*, die jedem Phänotyp einen *Fitnesswert* zuweist. Die Fitnessfunktion entspricht weitestgehend der Bewertungsfunktion bei Optimierungsproblemen, allerdings legt die Funktionsweise vieler gebräuchlicher Selektionsmethoden es nahe, sie so zu wählen, dass nur nicht-negative Werte angenommen werden und die Qualität eines Phänotyps um so höher ist, je größer sein Fitnesswert ist[6]. Um die Qualität eines Individuums zu bestimmen, das im Genotyp vorliegt, muss dieser zunächst dekodiert, also der resultierende Phänotyp bestimmt werden, der dann durch die Fitnessfunktion bewertet werden kann. Somit induziert die Dekodierung, wie in Abbildung 2.2 dargestellt, eine Fitnessfunktion auf $\mathcal{G}$, die es erlaubt, sinnvoll von der Fitness eines Genotyps zu sprechen.

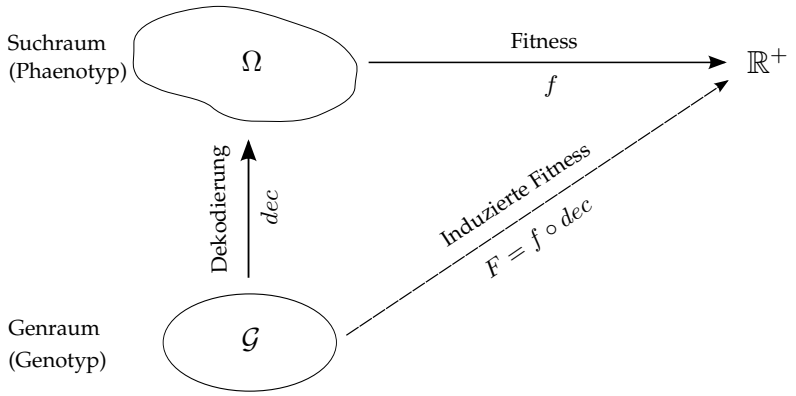


Abbildung 2.2: Schematische Darstellung des Zusammenhangs zwischen Genraum und Phänotyp-Raum. (Nach [Weicker 2002], verändert.)

Im Folgenden wird nicht streng zwischen der auf Ω definierten und der auf $\mathcal{G}$ induzierten Fitnessfunktion unterschieden werden.

Population

In Anlehnung an das biologische Vorbild werden Ansammlungen von Individuen als *Populationen* bezeichnet. Diese werden üblicherweise als Multimengen modelliert, sind also – wie gewöhnliche Mengen – unsortiert, können Individuen aber mehrfach enthalten. Ferner wird die Populationsgröße N im Allgemeinen fest gewählt.
Die Population P oder genauer gesagt die Folge $P_t, t \in \mathbb{N}_0$ von Populationen, die das wiederholte Durchlaufen des evolutionären Zyklus hervorbringt, stellt in gewissem Sinne die zentrale Komponente eines Genetischen Algorithmus dar: Veränderung durch Evolution findet auf der Ebene der Population statt, die einzelnen Individuen sind statisch.
Die Populationen der Folge $P_t, t \in \mathbb{N}_0$ werden auch als *Generationen* bezeichnet. Diese Bezeichnung ist dadurch motiviert, dass viele einfache Genetische Algorithmen in jedem Evolutionsschritt, also während jedes Durchlaufens des evolutionären Zyklus, die komplette Population austauschen. Dies birgt natürlich die Gefahr, dass bereits gefundene

[6]Dies stellt keine wirkliche Einschränkung dar: Da $\mathbb{R}$ bijektiv auf das Intervall $(0, 1)$ abgebildet werden kann, und ferner die Minimierung einer Funktion h mit Werten in diesem Intervall äquivalent zur Maximierung der Funktion $1 - h$ ist, kann jede Bewertungsfunktion der in 2.1.2 genannten Form in eine Fitnessfunktion mit den geforderten Eigenschaften überführt werden.

gute Lösungen wieder verloren gehen – was aber leicht vermieden werden kann, indem stets die besten Individuen einer Generation unverändert in die nächste übernommen wird; dies wird als *Elitismus* bezeichnet. Im Gegensatz zu diesen *generationsbasierten* Verfahren ersetzen die als *Steady-State*-Algorithmen bekannten Genetischen Algorithmen stets nur einen Teil der aktuellen Population durch neu erzeugte Individuen.

Initialisierung und initiale Bewertung

Die Initialisierung eines Evolutionären Algorithmus besteht darin, eine Startpopulation als Ausgangspunkt für die evolutionäre Suche bereitzustellen. Meist werden die Individuen dieser initialen Population rein zufällig erzeugt, es ist aber auch möglich, vorhandenes Wissen über das Problem oder Ergebnisse anderer Optimierungsverfahren einzubringen.

Selektion

Die Aufgabe des *Selektion* besteht – grob gesagt – darin, die evolutionäre Suche in die richtige Richtung zu führen. Sie konzentriert die Suche auf die Bereiche des Genraums, aus denen Individuen mit (relativ zu den anderen Individuen der jeweils aktuellen Population) hoher Fitness stammen. Abhängig davon, ob Zufallseinflüsse bei der Selektion eine Rolle spielen, wird zwischen *probabilistischen* und *deterministischen* Selektionsmethoden unterschieden. Selektion kann im evolutionären Zyklus an zwei Stellen auftreten:

Die *Eltern-* oder *Paarungsselektion* wählt Individuen aus, die zur Erzeugung neuer Individuen herangezogen werden; dabei kommen meist probabilistische Methoden zum Einsatz, etwa eine der folgenden:

Fitnessproportionale oder Rouletterad-Selektion ist das wohl am häufigsten eingesetzte Verfahren. Die Auswahl der Eltern erfolgt so, dass der Anteil der Nachkommen eines Individuums an der nächsten Generation im Mittel seiner relativen Fitness – dem Quotienten aus seiner Fitness und der Summe der Fitnesswerte der gesamten Population – entspricht. Diese Selektionsmethode entspricht dem Drehen eines Glücksrades, auf dem jedem Individuum der Population ein Abschnitt mit einer seiner relativen Fitness entsprechender Größe zugeordnet ist[7] (siehe Abbildung 2.3(a)). Die in der Literatur übliche Bezeichnung dieser Selektionsmethode als Rouletterad-Selektion ist etwas irreführend, da beim Rouletterad alle Felder gleich groß sind.

Turnier- oder Wettkampfselektion wählt zunächst zwei (oder mehr) Individuen aus der Population aus und selektiert dann das beste als Elter; die Auswahl der „Turniergegner“ kann zufällig gleichverteilt oder fitnessproportional erfolgen (siehe Abbildung 2.3(b)).

Die meist deterministische *Umweltselektion* entscheidet, welche der durch Variation neu erzeugten Individuen in die nächste Generation übernommen werden; gebräuchlich sind zum Beispiel die folgenden Verfahren[8]:

[7] Die noch einzuführenden Lernenden Klassifizierenden Systeme verwenden zu verschiedenen Zwecken Auswahlverfahren, die nach dem Prinzip der Rouletterad-Selektion funktionieren, den Auswahlwahrscheinlichkeiten jedoch andere Attribute als die Fitness zugrunde legen. Ein solches Verfahren soll als *Rouletterad-Selektion bezüglich des jeweiligen Attributs* bezeichnet werden.

[8] Im Sinne einer einheitlichen Notation werden hier die in der Literatur üblichen Bezeichnungen ‚(μ, λ)-‘ beziehungsweise ‚$(\mu + \lambda)$-Selektion‘ abgeändert.

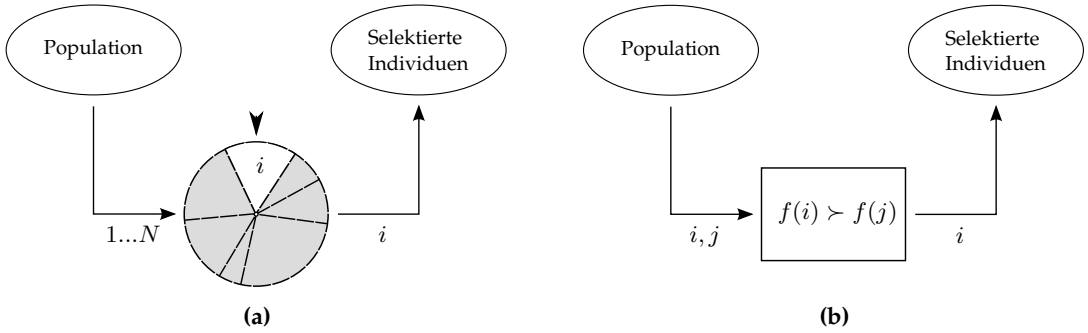


Abbildung 2.3: Veranschaulichung von **(a)** fitnessproportionaler Selektion (Rouletterad-Selektion) und **(b)** Turnierselektion.

(μ, N)-Selektion wählt unter den $\mu \geq N$ erzeugten Nachkommen die N besten aus. Weit verbreitet ist insbesondere der Sonderfall $\mu = N$, also die komplette Ersetzung der Population.

($\mu + N$)-Selektion wählt aus der Vereinigung von aktueller Population und $\mu \in \mathbb{N}$ neu erzeugten Nachkommen die N besten Individuen aus. Dieses Verfahren ist typisch für Steady-State-Algorithmen.

Variation

Die Variationsoperatoren haben die Aufgabe, neue Individuen aus den bestehenden zu erzeugen. Obwohl ihre genaue Form und Wirkungsweise stark von der verwendeten Repräsentation abhängig ist, können auf einer abstrakten Ebene zwei Typen unterschieden werden:

Rekombinations- oder Crossing-Over-Operatoren sind binäre Operatoren, die die Genotypen von (meist) zwei Elternindividuen kombinieren[9] und so einen, zwei oder mehr Nachkommen erzeugen – in der Hoffnung, dass die Nachkommen die guten Eigenschaften ihrer Eltern in sich vereinen und somit eine höhere Fitness aufweisen. Zum Einsatz kommen beispielsweise die folgenden Verfahren:

m-Punkt-Crossing-Over ist bei binärer wie auch bei reellwertiger Repräsentation anwendbar. Zufällig werden m Kreuzungspunkte $0 \leq x_1 \leq x_2 \leq \ldots \leq x_m \leq n$ gewählt, die Chromosomen der beiden Elternindividuen an diesen Stellen „zerschnitten“ und – wie in Abbildung 2.4(a) für den Fall $m = 2$ gezeigt – zu den Genotypen der Nachkommen neu zusammengesetzt.

Uniformes Crossing-Over ist ebenfalls sowohl bei binärer wie auch bei reellwertiger Repräsentation anwendbar. Für jedes Gen eines Nachkommens wird zufällig entschieden, das Allel welches Elternindividuums übernommen wird (Abbildung 2.4(b)).

Intermediäres Crossing-Over ist nur im Falle reellwertiger Repräsentation anwendbar. Zunächst wird zufällig ein Gewicht $w \in [0, 1]$ gewählt. Der Genotyp eines Nachkommen ergibt sich komponentenweise als gewichtete Summe der Genotypen der beiden Elternindividuen, wobei ein Elter mit w, der andere mit $(1 - w)$ gewichtet wird (Abbildung 2.4(c)).

[9] Die Bezeichnung ‚Crossing-Over‘ ist nicht konsistent mit der biologischen Terminologie. In der Biologie bezeichnet Crossing-Over einen intraindividuellen Genaustausch, bei den Genetischen Algorithmen einen interindividuellen.

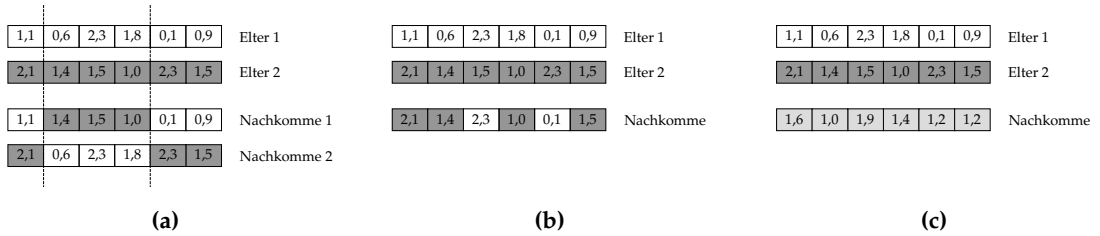


Abbildung 2.4: Veranschaulichung von **(a)** 2-Punkt-Crossing-Over, **(b)** Uniformem Crossing-Over und **(c)** Intermediärem Crossing-Over.

Mutationsoperatoren sind unäre Variationsoperatoren, die aus einem Individuum einen leicht veränderten Nachkommen erzeugen; die Veränderung erfolgt zufällig und ungezielt. Wohl am verbreitetsten ist die *Punktmutation*: Jedes Gen eines zu mutierenden Genotyps wird mit einer (sehr kleinen) Wahrscheinlichkeit p_{mut} mutiert. Im binären Fall wird das Allel des betroffenen Gens ausgetauscht ($0 \leftrightarrow 1$), im reellwertigen Fall wird dessen Wert um einen kleinen Betrag verringert oder erhöht[10].
Genetische Algorithmen verwenden üblicherweise sowohl Rekombination als auch Mutation: Die als Eltern selektierten Individuen werden zunächst mit hoher Wahrscheinlichkeit p_{cross} rekombiniert. Anschließend werden die entstandenen Individuen oder, falls keine Rekombination erfolgte, Kopien der Elternindividuen mutiert.
Die Mutationswahrscheinlichkeit wird stets sehr klein gewählt, da Mutationen leicht auch zu einer Verschlechterung bereits gefundener Lösungen führen können, was umso wahrscheinlicher ist, je besser diese sind. Dennoch sollte nicht komplett auf Mutation verzichtet werden, denn im Gegensatz zur Rekombination, die nur die in der Population bereits vorhandenen Allele neu kombinieren kann, ist die Mutation auch in der Lage, neue Allele in die Population einzuführen.

Terminierungsbedingung

Im Gegensatz zur natürlichen Evolution sollen Genetische Algorithmen terminieren, sie werden daher mit einer Terminierungsbedingung ausgestattet, die am Ende jedes Evolutionsschrittes überprüft wird. Üblich ist es, den Genetischen Algorithmus zu beenden, wenn die beste Lösung einen vorgegebenen Fitnesswert erreicht, eine bestimmte Generationenzahl durchlaufen oder über eine bestimmte Generationenzahl hinweg keine Verbesserung der gefundenen Lösungen mehr festgestellt wurde. Oft werden auch Kombinationen dieser Kriterien verwendet.

2.2 Reinforcement Learning

Der Begriff ‚Reinforcement-Learning' wird in zwei unterschiedlichen Bedeutungen verwendet: Er bezeichnet einerseits ein bestimmtes Lernszenario und andererseits „*the learning techniques typically studied by the RL community*" [Holland u. a. 2000]. Um Zweideutigkeiten zu vermeiden, wird daher im Folgenden zwischen dem Reinforcement-Learning-*Szenario* und den Reinforcement-Learning-*Verfahren* unterschieden. Die Darstellung in diesem Abschnitt orientiert sich vornehmlich an [Sutton u. Barto 1998].

[10] Die Größe dieser Inkremente respektive Dekremente ist meist nicht fest, sondern folgt einer um 0 zentrierten Wahrscheinlichkeitsverteilung, etwa einer Normalverteilung.

2.2.1 Das Reinforcement-Learning-Szenario

Das *Reinforcement-Learning-Szenario* ist dadurch charakterisiert, dass ein Lerner sein Verhalten in einer dynamischen Umwelt oder Lernumgebung aufgrund Versuchs und Irrtums anpassen muss. Die Umwelt ist durch die Menge $\mathcal{S}$ ihrer möglichen Zustände und die Menge $\mathcal{A}$ der in ihr möglichen Aktionen beschrieben und reagiert auf die vom Lerner ausgeführten Aktionen mit Belohnungen und Bestrafungen[11]:
Der auch *Agent* genannte Lerner interagiert zu Zeitpunkten $t = 0, 1, 2, \ldots$ mit der Umwelt; in jedem Zeitschritt t nimmt er den Zustand $s_t \in \mathcal{S}$ der Umwelt wahr und reagiert darauf mit einer Aktion $a_t \in \mathcal{A}$. Im nächsten Zeitschritt $t + 1$ erhält der Agent dann eine *Belohnung* $r_{t+1} \in \mathbb{R}$ und wird mit einer neuen Situation s_{t+1} konfrontiert. Abbildung 2.5 stellt die Agent-Umwelt-Interaktion schematisch dar.

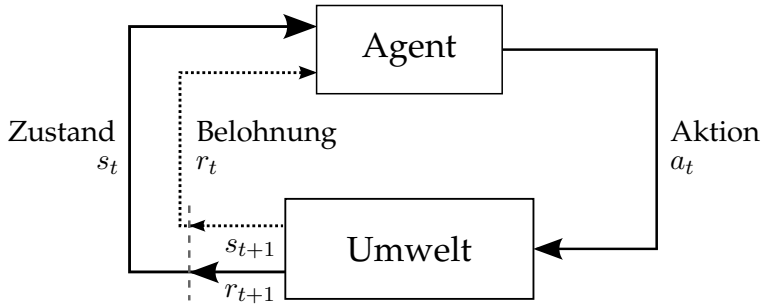


Abbildung 2.5: Schematische Darstellung der Interaktion von Agent und Umwelt. (Nach [Sutton u. Barto 1998].)

Welche Aktion der Agent jeweils ausführt, entscheidet er aufgrund seiner aktuellen *Politik* π_t. Diese gibt an, mit welcher Wahrscheinlichkeit $\mathrm{P}\left(a_t = a | s_t = s\right)$ eine Aktion a in einem Zustand s ausgeführt wird:

$$\pi_t : \mathcal{S} \times \mathcal{A} \to [0, 1]\,; \;\; \pi_t\left(s, a\right) \mapsto \mathrm{P}\left(a_t = a | s_t = s\right) \tag{2.1}$$

Ein Spezialfall sind die *deterministischen Politiken*, bei denen in jedem Zustand je eine Aktion Wahrscheinlichkeit 1 hat, bei Erreichen dieses Zustands also stets ausgeführt wird.
Die Aufgabe eines Agenten besteht darin, eine Politik zu wählen, die seinen *Return*, das ist die Gesamtheit der Belohnungen, die er im Laufe der Zeit erhält, maximiert. Dazu ist es nötig, abzuschätzen, welche zukünftigen Zustände und Belohnungen sich als Konsequenz einer ausgeführten Aktion ergeben werden. Dies ist am einfachsten, wenn der aktuelle Zustand s_t und die gewählte Aktion a_t bereits alle Informationen enthalten, die benötigt werden, um die Wahrscheinlichkeiten für bestimmte Nachfolgezustände s_{t+1} und Belohnungen r_{t+1} zu bestimmen und diese Wahrscheinlichkeiten nicht von weiter zurückliegenden Zuständen und Aktionen abhängen:

$$\mathrm{P}\left(s_{t+1} = s', r_{t+1} = r' | s_t, a_t\right) = \mathrm{P}\left(s_{t+1} = s', r_{t+1} = r' | s_t, a_t, s_{t-1}, a_{t-1}, ..., s_0, a_0\right) \tag{2.2}$$

Ein Reinforcement-Learning-Problem, das diese *Markov-Eigenschaft* besitzt und dessen *Zustandsmenge* $\mathcal{S}$ und *Aktionsmenge* $\mathcal{A}$ darüber hinaus endlich sind, wird als *endlicher Markov-Entscheidungsprozess* bezeichnet. Im Folgenden werden Reinforcement-Learning-Probleme stets als endliche Markov-Entscheidungsprozesse vorausgesetzt, da die meisten Reinforcement-Learning-Verfahren – in der Theorie – nur auf derartige Probleme anwendbar sind. In der Praxis genügt es meist schon, dass die Markov-Eigenschaft näherungsweise erfüllt ist.

[11] Bestrafungen können als negative Belohnungen angesehen werden, weswegen im Folgenden nur noch von Belohnungen gesprochen wird.

2.2.2 Wertefunktionen

Um den Return zu maximieren, genügt es im Allgemeinen nicht, die Aktionen zu wählen, die kurzfristig zu den höchsten Belohnungen führen; ein Agent muss auch darauf achten, nicht in Zustände zu gelangen, von denen aus nur noch niedrige Belohnungen erzielbar sind. Die Wahl der auszuführenden Aktionen wird zusätzlich dadurch erschwert, dass Belohnungen meist nicht direkt auf eine Aktion folgen, sondern erst verzögert als Ergebnis einer ganzen Reihe von Aktionen gewährt werden. Somit ist oft nicht klar, welchen Anteil eine bestimmte Aktion an der erhaltenen Belohnung hat (*temporal-credit-assignment-Problem*).
Bei der Definition des Returns muss zwischen zwei Typen von Reinforcement-Learning-Problemen unterschieden werden: *Kontinuierliche* und *episodische*. Ein Problem heißt episodisch, wenn es über *Terminalzustände* verfügt, die es erlauben, die Folge der Agent-Umwelt-Interaktionen in endliche Sequenzen zu unterteilen. Ein Beispiel für eine episodische Aufgabe ist das Erlernen eines Spieles: der Agent spielt wiederholt „gegen" die Umwelt, jedes einzelne Spiel bildet eine abgeschlossene *(Lern-)Episode*, die in einem der Terminalzustände ‚Spiel gewonnen' oder ‚Spiel verloren' endet. Nicht-episodische Probleme heißen kontinuierlich. Von der Lernepisode zu unterscheiden ist der *Lernschritt*, der einem einzelnen Durchlauf des in Abbildung 2.5 dargestellten Zyklus entspricht.
Bei kontinuierlichen Problemen wird der Return als gewichtete Summe aller zukünftigen Belohnungen definiert, wobei Belohnungen mit umso geringerem Gewicht eingehen, je weiter sie in der Zukunft liegen. Diese *Diskontierung* gewährleistet, dass der Return endlich bleibt[12]. Am verbreitetsten ist die *geometrische Diskontierung*:

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}, \quad 0 \leq \gamma < 1 \tag{2.3}$$

Bei episodischen Problemen wird der Return analog definiert, es werden jedoch nur die bis zum Ende der aktuellen Episode (Zeitpunkt T) auftretenden Belohnungen berücksichtigt, die Summation läuft also nur bis $T - t - 1$. Zudem kann auf die Diskontierung verzichtet werden ($0 \leq \gamma \leq 1$).
Der erzielte Return ist offensichtlich abhängig von der verfolgten Politik. Die *Zustands-Wertefunktion* V^π gibt für jeden Zustand $s \in \mathcal{S}$ den Return an, den ein Agent erwarten kann, wenn er, im Zustand s beginnend, der Politik π folgt (E_π bezeichnet den von der Politik π abhängigen Erwartungswert):

$$V^\pi(s) = \mathrm{E}_\pi\left(R_t | s_t = s\right) = \mathrm{E}_\pi\left(\sum_{k=1}^{K} \gamma^k r_{t+k+1} | s_t = s\right) \tag{2.4}$$

Analog liefert die *Aktions-Wertefunktion* Q^π den zu erwartenden Return, wenn im Zustand s die Aktion a ausgeführt und danach der Politik π gefolgt wird:

$$\begin{aligned} Q^\pi(s,a) &= \mathrm{E}_\pi\left(R_t | s_t = s, a_t = a\right) = \mathrm{E}_\pi\left(\sum_{k=1}^{K} \gamma^k r_{t+k+1} | s_t = s, a_t = a\right) \\ &= \mathrm{E}_\pi\left(r_{t+1} + \gamma V^\pi(s_{t+1}) | s_t = s, a_t = a\right) \end{aligned} \tag{2.5}$$

Die Werte aufeinanderfolgender Zustände respektive Zustands-Aktions-Paare sind klarerweise voneinander abhängig. Der Zusammenhang zwischen dem Wert eines Zustandes s und den Werten möglicher Nachfolgezustände s' wird mit den von der gewählten Aktion a abhängigen Übergangswahrscheinlichkeiten

$$P^a_{ss'} := \mathrm{P}\left(s_{t+1} = s' | s_t = s, a_t = a\right) \tag{2.6}$$

[12] Zusätzlich muss die Folge $\{r_k\}$ der Belohnungen beschränkt sein.

und den Erwartungswerten der Belohnungen bei diesen Übergängen

$$R^a_{ss'} := \mathrm{E}\left(r_{t+1}|s_t = s, a_t = a, s_{t+1} = s'\right) \tag{2.7}$$

durch die *Bellman-Gleichung* ausgedrückt[13]:

$$V^\pi(s) = \sum_a \pi(s,a) \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V^\pi(s')\right] \tag{2.8}$$

Für Aktions-Wertefunktionen nimmt die Bellman-Gleichung die folgende Form an:

$$\begin{aligned} Q^\pi(s,a) &= \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma \sum_{a'} \pi(s',a') Q^\pi(s',a')\right] \\ &= \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V^\pi(s')\right] \end{aligned} \tag{2.9}$$

Anhand ihrer Wertefunktionen können Politiken bewertet und verglichen werden. Eine Politik π' ist in einem Zustand s besser als eine Politik π, wenn $V^{\pi'}(s) > V^\pi(s)$, und π' heißt *besser* als π, wenn π' in allen Zuständen $s \in \mathcal{S}$ besser als π ist:

$$\pi' > \pi : \Longleftrightarrow \; V^{\pi'}(s) > V^\pi(s) \;\; \forall s \in \mathcal{S} \tag{2.10}$$

Die Wertefunktionen liefern auch einen Anhaltspunkt dafür, wie eine gegebene Politik π verbessert werden kann: Ist eine Politik π' in einem Zustand s besser als π, so kann π verbessert werden, indem für den Zustand s die Aktionswahrscheinlichkeiten von π' übernommen werden. Somit gibt es stets mindestens eine *optimale Politik* π^*, die besser oder ebenso gut wie alle anderen ist. Alle optimalen Politiken haben dieselbe optimale Zustands-Wertefunktion

$$V^*(s) = \max_\pi V^\pi(s) \;\; \forall s \in \mathcal{S} \tag{2.11}$$

und dieselbe eindeutige optimale Aktions-Wertefunktion :

$$Q^*(s,a) = \max_\pi Q^\pi(s,a) \;\; \forall s \in \mathcal{S} \wedge \; \forall a \in \mathcal{A} \tag{2.12}$$

Viele Reinforcement-Learning-Verfahren versuchen eine optimale Politik zu finden, indem sie die Technik der *Politik-Iteration* anwenden: Abwechselnd wird die Wertefunktion der aktuellen Politik geschätzt (*Politik-Evaluation*) und basierend auf dieser Schätzung eine Verbesserung der Politik vorgenommen. Dabei wird unterschieden zwischen *on-policy*-Verfahren, die die Politik bewerten, die sie aktuell verfolgen und *off-policy*-Verfahren, die eine (meist geringfügig) andere Politik verfolgen als die, die sie bewerten und verbessern.
Die Politik-Verbesserung geschieht im Allgemeinen dadurch, dass – ausgehend von der im vorangegangenen Evaluations-Schritt erhaltenen Schätzung V^π der Wertefunktion der aktuellen Politik π – eine neue *gierige (greedy)* deterministische Politik π' generiert wird. Diese führt in jedem Zustand s die Aktion a aus, die das beste $Q^\pi(s,a)$ liefert:

$$\begin{aligned} \pi' &= \arg\max_a Q^\pi(s,a) \\ &= \arg\max_a \mathrm{E}\left(r_{t+1} + \gamma V^\pi(s_{t+1})|s_t = s, a_t = a\right) \\ &= \arg\max_a \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V^\pi(s')\right] \end{aligned} \tag{2.13}$$

[13] Eine Herleitung findet sich z.B in [Sutton u. Barto 1998].

Der Politik-Verbesserungs-Satz garantiert, dass π' mindestens ebenso gut ist wie π:

Satz (Politik-Verbesserung)
Seien π und π' zwei Politiken, sodass

$$\sum_a \pi'(s,a)Q^\pi(s,a) \geq \sum_a \pi(s,a)Q^\pi(s,a)$$

Dann ist π' besser oder genau so gut wie π:

$$V^{\pi'}(s) \geq V^\pi(s) \;\; \forall s \in \mathcal{S}.$$

Zum Beweis siehe [Sutton u. Barto 1998] □

Somit führt die ständige Wiederholung von Evaluation ($\xrightarrow{E}$)und Verbesserung ($\xrightarrow{I}$) zu einer Reihe sich stetig verbessernder Politiken und Wertefunktionen:

$$\pi_0 \xrightarrow{E} V^{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} V^{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \cdots \xrightarrow{I} \pi_* \xrightarrow{E} V^* \tag{2.14}$$

Für endliche Markov-Entscheidungsprozesse kann gezeigt werden, dass eine Politik optimal ist, wenn sich durch eine Politik-Verbesserung die Werte der einzelnen Zustände nicht ändern [Sutton u. Barto 1998]. Somit führt jeder Politik-Verbesserungs-Schritt in der Sequenz (2.14) zu einer echten Verbesserung der Politik, sofern diese nicht bereits optimal ist. Da es für einen endlichen Markov-Entscheidungsprozess offensichtlich nur endlich viele deterministische Politiken gibt, muss die Politik-Iteration in endlich vielen Schritten eine optimale Politik finden.

2.2.3 Klassische Reinforcement-Learning-Verfahren

Im Folgenden werden drei verbreitete Klassen von Reinforcement-Learning-Verfahren beschrieben, die nach dem Prinzip der Politik-Iteration funktionieren: *Dynamisches Programmieren*, *Monte-Carlo-Methoden* und *Temporal-Difference-Methoden*. Da der Politik-Verbesserungs-Schritt bei allen drei Verfahren in der oben beschriebenen Weise erfolgt, wird im Folgenden nur noch auf den Politik-Evaluations-Schritt eingegangen.

Dynamisches Programmieren

Beim Dynamischen Programmieren werden die Übergangswahrscheinlichkeiten $P^a_{ss'}$ und die Erwartungswerte $R^a_{ss'}$ als bekannt vorausgesetzt und benutzt, um die Wertefunktion V^π der aktuellen Politik π iterativ zu approximieren, indem die Bellman-Gleichung (2.8) als Iterationsvorschrift aufgefasst wird:

$$V^\pi_{k+1}(s) = \sum_a \pi(s,a) \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V^\pi_k(s')\right] \tag{2.15}$$

Ein solches Verfahren, das neue Schätzungen $V^\pi_{k+1}(s)$für einen Zustand s basierend auf aktuellen Schätzungen $V^\pi_k(s')$ für die möglichen Nachfolgezustände s' berechnet, wird als *Bootstrapping* bezeichnet. Es kann gezeigt werden, dass für beliebige Anfangsnäherungen V^π_0 die Folge V^π_k für $k \to \infty$ gegen V^π konvergiert [Sutton u. Barto 1998].

Monte-Carlo-Methoden

Ein großer Nachteil des Dynamischen Programmierens besteht darin, dass $P^a_{ss'}$ und $R^a_{ss'}$ für die Berechnung der Wertefunktion als bekannt vorausgesetzt werden müssen. Monte-Carlo-Verfahren versuchen – einem intuitiven Begriff von „Lernen" entsprechend –, die Wertefunktion einer Politik π aufgrund von Erfahrungen zu lernen, die bei der Interaktion mit der Umwelt gewonnen werden. Sie schätzen den Wert $V^\pi(s)$ beziehungsweise $Q^\pi(s,a)$ bestimmter Zustände s respektive Zustands-Aktions-Paare (s,a) durch den Durchschnitt aller für diese beobachteten Returns ab. Die Aktualisierung dieser Abschätzung erfolgt jeweils nach Beendigung einer Lernepisode. Monte-Carlo-Methoden sind jedoch nicht für kontinuierliche Lernaufgaben definiert, da sie aus vollständigen Returns lernen, die gemäß (2.3) nur in episodischen Lernproblemen sicher bestimmt werden können.

Temporal-Difference-Methoden

Temporal-Difference-Methoden [Sutton 1988] kombinieren das „Lernen aus Erfahrung" der Monte-Carlo-Methoden mit dem Bootstrapping-Ansatz des Dynamischen Programmierens: Die Schätzung des Wertes eines Zustandes s_t wird auf Grundlage nicht des Returns, sondern der unmittelbaren Belohnung r_{t+1} und der aktuellen Schätzung des Wertes des Nachfolgezustands s_{t+1} aktualisiert:

$$V(s_t) \leftarrow V(s_t) + \beta\left[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)\right] \tag{2.16}$$

Die Anpassung erfolgt zum Zeitpunkt $t+1$, also sobald s_{t+1} und r_{t+1} bekannt sind. Die Lernrate $0 \leq \beta \leq 1$ bestimmt, wie stark die Schätzung von $V(s_t)$ an die Summe aus direkter Belohnung r_{t+1} und diskontiertem Wert $V(s_{t+1})$ des Nachfolgezustands angepasst wird. Wird β während des Lernprozesses in geeigneter Weise verringert, so konvergiert (2.16) mit Wahrscheinlichkeit 1 gegen die optimale Wertefunktion V^* [Dayan 1992].
Temporal-Difference-Methoden benötigen somit weder Kenntnis von $P^a_{ss'}$ und $R^a_{ss'}$ noch benötigen sie vollständige Returns, um die Schätzung der Wertefunktion anzupassen; sie sind somit auch für kontinuierliche Reinforcement-Learning-Probleme geeignet. Zwei Beispiele für Temporal-Difference-Methoden sind die folgenden Verfahren:

SARSA Der SARSA-Algorithmus [Rummery u. Niranjan 1994] verwendet eine zu (2.16) analoge Anpassungsregel für Aktions-Wertefunktionen[14]:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \beta\left[r_{t+1} + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)\right] \tag{2.17}$$

Wie bei allen on-policy-Verfahren stellt sich auch beim SARSA-Algorithmus das *exploration-exploitation-Problem*: Wird stets ausschließlich die vorhandene Erfahrung benutzt (*exploitation*), also die aktuell als am besten angesehene Aktion ausgeführt, so können andere, möglicherweise bessere, Aktionen nie getestet (*exploration*) werden. Bei der Politik-Verbesserung muss daher statt der gierigen Politik (2.13) eine Politik gewählt werden, die das „Ausprobieren" anderer Lösungen ermöglicht und somit nicht optimal sein kann. Häufig benutzt werden *ε-gierige* Politiken, die mit Wahrscheinlichkeit ε eine zufällige Aktion wählen und sich ansonsten gierig verhalten.

[14] Falls s_{t+1} ein Endzustand ist, also keine Aktion a_{t+1} mehr durchgeführt wird, so wird in (2.17) $Q(s_{t+1},a_{t+1}) = 0$ gesetzt.

Q-Learning Im Gegensatz zum SARSA-Algorithmus handelt es sich beim Q-Learning [Watkins 1989] um eine off-policy-Methode. Bei der Anpassung der Aktions-Wertefunktion wird statt des Q-Wertes der tatsächlich durchgeführten Aktion das Maximum der Q-Werte des Nachfolgezustands benutzt[15]:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \beta \left[r_{t+1} + \gamma max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right] \tag{2.18}$$

Temporal-Difference-Methoden können auch dahingehend erweitert werden, dass in die Schätzung der Wertefunktion nicht nur die unmittelbar folgende Belohnung, sondern auch weiter zurückliegende eingehen. Um dies zu erreichen, werden sogenannte Aktivierungsspuren (*eligibility traces*) benutzt: In jedem Zeitschritt t wird die *Aktivierung* $e_t(s), s \in \mathcal{S}$ aller Zustände um einen Faktor $\gamma\lambda$ verringert, die Aktivierung des besuchten Zustandes s_t wird um 1 erhöht:

$$e_t(s) = \begin{cases} \gamma\lambda e_{t-1}(s) & \text{für } s \neq s_t \\ \gamma\lambda e_{t-1}(s) + 1 & \text{für } s = s_t \end{cases} \tag{2.19}$$

Anschließend werden dann alle Zustände aktualisiert:

$$V(s) \leftarrow V(s) + \beta e_t(s) \left[r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \right] \tag{2.20}$$

Der Parameter $0 \leq \lambda \leq 1$ regelt dabei, wie stark weiter in der Zukunft liegende Belohnungen sich auf den Wert eines Zustandes auswirken und motiviert die Bezeichnung TD(λ) für diese Methoden. Die durch (2.16) gegebene Methode entspricht TD(0). TD(1) berücksichtigt alle Belohnungen bis zum Ende einer Episode und entspricht somit den Monte-Carlo-Methoden - mit dem Unterschied, dass TD(1) auch für kontinuierliche Aufgaben geeignet ist, da die Anpassung der Wertefunktion inkrementell erfolgt.

Ein wesentlicher Nachteil der bisher betrachteten Reinforcement-Learning-Verfahren besteht darin, dass sie für jeden Zustand respektive jedes Zustands-Aktions-Paar eine Schätzung seines Wertes – etwa in Form einer Tabelle - speichern. Je größer der Zustandsraum $\mathcal{S}$ und die Aktionsmenge $\mathcal{A}$ sind, desto mehr Zustände / Zustands-Aktions-Paare müssen bewertet werden und umso länger dauert, es gute Lernergebnisse zu erzielen. Daher sind derartige *tabellarische Reinforcement-Learning-Verfahren* im Falle sehr großer Zustandsräume nicht praktikabel, im Falle kontinuierlicher Zustandsräume ist eine Tabellierung der Wertefunktion überhaupt nicht möglich. Eine Möglichkeit dennoch Wertefunktionen zu lernen, besteht darin, den Zustandsraum zunächst geeignet zu verkleinern respektive zu diskretisieren und dann ein tabellarisches Verfahren auf den so erhaltenen Zuständen anzuwenden. Eine weitere Möglichkeit ist die Approximation der Wertefunktion durch auf Gradientenabstiegsverfahren basierenden Funktionsapproximatoren, vergleiche hierzu zum Beispiel [Sutton u. Barto 1998; Kaelbling u. a. 1996]. Einen anderen nicht auf der Tabellierung von Wertefunktionen basierenden Ansatz zum Reinforcement-Learning stellen Lernende Klassifizierende Systeme dar.

2.3 Grundlagen Lernender Klassifizierender Systeme

Lernende Klassifizierende Systeme stellen ein im Kontext Genetischer Algorithmen entstandenes Paradigma maschinellen Lernens dar, das Charakteristika sowohl des evolutionären Lernens als auch des Reinforcement-Learnings vereinigt.

[15] Ist s_{t+1} ein Endzustand, wird $\max_a Q(s_{t+1})$ als Null angenommen.

2.3.1 Evolutionäres Reinforcement-Learning

Die Idee, Genetische Algorithmen nicht nur zur Lösung klassischer Optimierungsprobleme zu benutzen, sondern auch in Reinforcement-Learning-Szenarien einzusetzen, geht auf John H. Holland zurück. In seinem Werk *Adaptation in Natural and Artificial Systems* [Holland 1992] führte er 1975 nicht nur Genetische Algorithmen ein[16], sondern entwickelte auch eine formale Sprache über einem zehn-elementigen Alphabet, die *Broadcast-Language*, um Verhalten in Form von Zeichenketten darzustellen und somit der Optimierung durch Genetische Algorithmen zugänglich zu machen; die Bewertung des Verhaltens, also die Fitnessberechnung, sollte basierend auf den von der Umwelt erhaltenen Belohnungen erfolgen.
Holland ging von der Überlegung aus, dass die Aktionen eines beliebigen Systems durch den Zustand seiner Umwelt (und ggf. seiner selbst) bedingt sind und aus diesen durch eine Reihe von „Verarbeitungsschritten" hervorgehen. Dementsprechend repräsentieren die Symbole der Broadcast-Language teils Daten, teils Operatoren zu deren Manipulation; die Worte dieser Sprache beschreiben in Form von Regeln, wie Eingaben (Zustände der Umwelt) zu Ausgaben verarbeitet werden. Die Ausgabe einer solchen Regel ist wiederum ein Wort der Broadcast-Language, kann also ebenso eine Aktion in der Umwelt wie auch einen internen Zustand des Systems oder sogar eine neue Regel repräsentieren. Erst 2007 wurde ein auf der vollständigen Broadcast-Language basierendes Lernverfahren implementiert [Decraene u. a. 2007], was wohl ihrer extremen Komplexität zuzuschreiben ist. Die der Broadcast-Language zugrunde liegende Idee, das Verhalten eines Reinforcement-Learning-Agenten in Form von Regeln darzustellen und zu evolvieren, war jedoch der Ausgangspunkt der Entwicklung Lernender Klassifizierender Systeme, deren Funktionsweise – sehr grob – wie folgt beschrieben werden kann:
Lernende Klassifizierende Systeme nehmen den Zustand ihrer Umwelt wahr, bestimmen mit Hilfe von Regeln, den *Klassifizierern*, eine angemessene Aktion und führen diese in der Umwelt aus. „Gutes" Verhalten wird von der Umwelt belohnt; die Belohnungen bestimmen – direkt oder indirekt – die Fitness der Klassifizierer, die ein Genetischer Algorithmus benutzt, um bestehende Regeln zu verbessern oder neue, bessere Klassifizierer einzuführen – mit dem Ziel einen Satz von Klassifizierern zu finden, der alle Zustände der Umwelt abdeckt und in jedem Zustand eine den Return maximierende Aktion wählt.
Es liegt somit nahe, Lernende Klassifizierende Systeme als eine spezielle Form Genetischer Algorithmen zu betrachten und ihre Reinforcement-Learning-Komponente als komplizierte Form der Fitnessberechnung. Dieser Standpunkt kommt in der von Goldberg verwendeten Bezeichnung *Genetics-based machine learning* [Goldberg 1989] zum Ausdruck. Andererseits können Lernende Klassifizierende Systeme aber auch als Reinforcement-Learning-Verfahren angesehen werden, die einen Genetischen Algorithmus nur benutzen (z.B. [Holland u. a. 2000, Stellungnahme von Colombetti und Dorigo]). Diese Sichtweise ist insbesondere bei neueren Lernenden Klassifizierenden Systemen, wie dem in Abschnitt 2.4 vorgestellten XCS, angemessen [Kovacs 2002].
Zwischen diesen beiden Extremen ist natürlich Platz für eine ganze Reihe weiterer Sichtweisen, einen Überblick liefert eine Reihe von elf Essays, in denen auf dem Gebiet führende Wissenschaftler die Frage „*What is a Learning Classifier System?*" aus ihrem jeweiligen Blickwinkel beantworten [Holland u. a. 2000].

Generell können anhand der Gestalt der vom Genetischen Algorithmus verwendeten Individuen zwei Typen Lernender Klassifizierender Systeme unterschieden werden:

[16] Holland veröffentlichte *Adaptation in Natural and Artificial Systems* erstmals 1975, verfügbar sind heute aber praktisch nur Exemplare der 1992 erschienenen erweiterten und veränderten Ausgabe.

Beim weniger verbreiteten *Pittsburgh-Ansatz* stellt jedes Individuum eine Gesamtlösung des betrachteten Problems, also eine Gruppe von Klassifizierern, dar (Abbildung 2.6(a)). Beispiele für Systeme dieses Typs sind LS-1 [Smith 1980] und GABIL [De Jong u. a. 1993]. Beim verbreiteteren *Michigan-Ansatz*[17] hingegen kodiert jedes Individuum einen einzelnen Klassifizierer (Abbildung 2.6(b)). Diesem Typ gehören insbesondere das System von Holland (Abschnitt 2.3.3) und das derzeitige Standardsystem XCS (Abschnitt 2.4) an.

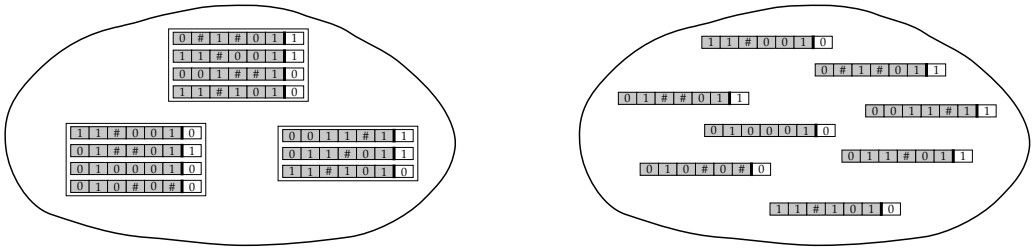


Abbildung 2.6: Binär kodierte Lernende Klassifizierende Systeme vom **(a)** Pittsburgh-Typ und **(b)** Michigan-Typ.

Lernende Klassifizierende Systeme bestimmen die Fitness der Individuen auf Grundlage der von der Umwelt erhaltenen Belohnungen. Beim Michigan-Ansatz stellt jeder Klassifizierer ein Individuum des Genetischen Algorithmus dar und muss ausgehend von den erhaltenen Belohnungen bewertet werden. Problematisch dabei ist, dass eine Belohnung oft nicht das unmittelbare Ergebnis der Anwendung eines einzelnen Klassifizierers ist, sondern sich zeitlich verzögert als Resultat einer ganzen Reihe von Aktionen – mithin aus dem Zusammenwirken mehrerer nacheinander aktiver Klassifizierer – ergibt. Der Verlust nur eines von diesen kann zum Zusammenbruch der Aktivierungskette und damit zu einer Schwächung des gesamten Systems führen. Eine geschickte Verteilung auftretender Belohnungen (*credit-assignment*) auf alle beteiligten Klassifizierer ist daher Voraussetzung für eine gute Gesamtlösung.
Beim Pittsburgh-Ansatz stellt sich dieses Problem nicht, da jedes Individuum einen vollständigen Regelsatz darstellt, der als Ganzes bewertet wird. Dies bedingt jedoch, da zur angemessenen Bewertung eines Individuums, das einen ganzen Regelsatz darstellt, dessen Anwendung auf eine größere Zahl von Zuständen beobachtet werden muss, dass längere Evaluationsperioden zwischen den Evolutionsschritten liegen. Dementsprechend werden Pittsburgh-Systeme üblicherweise nur zum Offline-Lernen eingesetzt [Butz 2006], während Systeme vom Michigan-Typ, die ihre Klassifizierer im schnellen Wechsel evaluieren und evolvieren [J.Bacardit u. a. 2008], typische Online-Lerner sind.

Auch wenn dem Pittsburgh-Ansatz in jüngerer Zeit wieder mehr Aufmerksamkeit [z.B. J.Bacardit u. a. 2004] zuteil wird, verwenden die meisten Arbeiten im Bereich der Lernenden Klassifizierenden Systeme nach wie vor den Michigan-Ansatz, der sich bereits in den 1980er Jahren zum Standard-Ansatz entwickelte. Insbesondere das XCS als derzeit wohl am weitesten verbreitetes Lernendes Klassifizierendes System und das ausgehend von diesem in der vorliegenden Arbeit entwickelte HCS sind Vertreter dieses Ansatzes, weswegen im Folgenden auch nicht näher auf den Pittsburgh-Ansatz eingegangen wird; eine Einführung in auf dem Pittsburgh-Ansatz basierende evolutionäre Reinforcement-Learning-Verfahren findet sich zum Beispiel in [Moriarty u. a. 1999].

[17] Die Namensgebung der beiden Ansätze bezieht sich auf die Standorte der Universitäten, an denen der erste Vertreter des jeweiligen Typs entwickelt wurde.

2.3.2 Grundstruktur Lernender Klassifizierender Systeme vom Michigan-Typ

Die in dieser Arbeit betrachteten Lernenden Klassifizierenden Systeme erscheinen auf den ersten Blick recht unterschiedlich, auf einer abstrakten Ebene weisen sie jedoch die gleiche Grundstruktur auf.

Zentrale Komponenten

Die Grundstruktur Lernender Klassifizierender Systeme geht zurück auf das erste Lernende Klassifizierende System überhaupt, das Cognitive-System-1 (CS-1) von Holland und Reitman [Holland u. Reitman 1978]:

> *The type of cognitive system (CS) studied here has four basic parts: (1) a set of interacting elementary productions, called lassifiers [sic!], (2) a performance algorithm that directs the action of the system in the environment, (3) a simple learning algorithm that keeps a record of each classifier's success in bringing about rewards, and (4) a more complex learning algorithm, called the genetic algorithm, that modifies the set of classifiers so that variants of good classifiers persist and new, potentially better ones are created . . .*

Diese vier grundlegenden Bestandteile finden sich auch bei den aktuellen Lernenden Klassifizierenden Systemen, zumeist unter den folgenden Bezeichnungen [Holmes u. a. 2002]:

Wissensbasis/Population: Die Wissensbasis enthält das „Wissen" des Systems in Form einer Population von Klassifizierern, die generell in der Form ‚*Bedingung : Aktion*' vorliegen.

Performance-Komponente: Die Performance-Komponente empfängt die Signale der Umwelt, wählt eine Aktion und führt diese aus.

Reinforcement-Komponente: Die Reinforcement-Komponente (das „einfache" Lernverfahren des obigen Zitats) verteilt die von der Umwelt erhaltenen Belohnungen an die Klassifizierer, die zu deren Erhalt beigetragen haben. Daher wird sie auch als *credit-assignment*-Komponente bezeichnet.

Discovery-Komponente: Die Discovery-Komponente übernimmt die Aufgabe, mit Hilfe eines Genetischen Algorithmus und eventuell weiterer Mechanismen bessere Klassifizierer zu finden respektive bestehende Klassifizierer zu verbessern.

Interaktion mit der Umwelt

Neben den vier zentralen Komponenten verfügt das CS-1 über *Detektoren* und *Aktuatoren* (auch *Effektoren* genannt) als Schnittstellen zur Umwelt. Die Detektoren nehmen den Zustand der Umwelt wahr und übersetzen ihn in Nachrichten „in der Sprache des Systems"; die Aktuatoren interpretieren die Aktionssignale des Systems und führen die entsprechenden Aktionen in der Umwelt aus – etwa im Falle eines von einem Lernenden Klassifizierenden System gesteuerten Roboters: Infrarotsensoren als Detektoren messen Abstände zu umgebenden Gegenständen und „übersetzen" diese in ein Zustandssignal, mit dem das Klassifizierende System arbeiten kann. Dessen Aktionssignale werden von Motoren (Aktuatoren) in Bewegungen umgesetzt.
Diese technische Sichtweise legt es nahe, Detektoren und Aktuatoren als Teile des Lernenden Systems oder als eigenständige – weder dem System noch der Umwelt zugehörige – Komponenten zu betrachten. Andererseits ist für das Lernende Klassifizierende Sy-

stem selbst nur interessant, was „auf seiner Seite“ der Aktuatoren und Detektoren passiert; seine abstrakten Zustände und Aktionen – nicht deren Entsprechungen in der Umwelt – sind für es die „wahren“ Zustände und Aktionen. Somit ist es auch legitim, Aktuatoren und Detektoren im Folgenden als bereits der Umwelt zugehörige Komponenten aufzufassen[18] und nicht näher zu betrachten.

2.3.3 Hollands Lernendes Klassifizierendes System

Das im Folgenden beschriebene Lernende Klassifizierende System entspricht im Wesentlichen der von Holland überarbeiteten Form ([Holland 1986]) des Cognitive-System-1, die – in einer Reihe von Variationen – lange Zeit das Standardmodell eines Lernenden Klassifizierenden Systems darstellte. Abbildung 2.7 zeigt den Aufbau dieses Standardmodells.

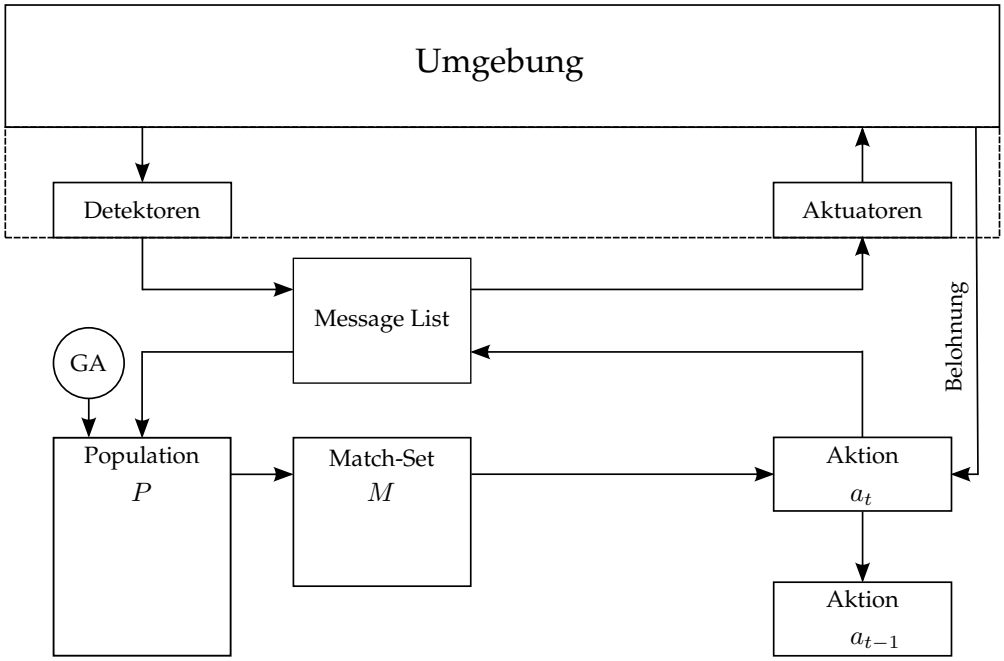


Abbildung 2.7: Schematische Darstellung des LCS in der von Holland vorgeschlagenen Form. (Nach [Bull u. Kovacs 2005], verändert.)

Wissensbasis/Population

Hollands System setzt voraus, dass die Zustände der Umwelt durch (externe) Nachrichten in Form binärer Zeichenketten einer beliebigen, aber festen Länge n übermittelt werden und auch die Aktionen, die in der Umwelt ausgeführt werden können, in dieser Weise kodiert sind, also $\mathcal{S} = \{0, 1\}^n$ und $\mathcal{A} = \{0, 1\}^n$.
Die Wissensbasis des Systems bildet eine Population P fester Größe N von Klassifizierern der Form

Bedingung : Aktion,

[18] Diese Sichtweise entspricht auch der pragmatischen Vorgehensweise bei der Simulation einer Umwelt im Rechner; üblicherweise wird die Umwelt so gestaltet, dass sie „die Sprache des Lernenden Klassifizierenden Systems spricht“.

wobei *Bedingung* und *Aktion* jeweils Zeichenketten der Länge n über dem ternären Alphabet $\{0, 1, \#\}$ sind.
Ein Klassifizierer ist nach Erhalt einer Nachricht *anwendbar*, er *passt* zu der Nachricht (respektive zu dem entsprechenden Zustand der Umwelt), wenn seine Bedingung an allen Stellen mit der Nachricht übereinstimmt. Das *Wildcard-Symbol* # dient dabei in der Bedingung als ‚egal'-Zeichen oder Platzhalter, stimmt also sowohl mit einer `0` als auch einer `1` in der Nachricht überein. Im Aktionsteil eines Klassifizierers wird das Wildcard-Symbol benutzt, um das Symbol an der entsprechenden Stelle der Nachricht in die Aktion „durchzureichen".

Beispiel: Der Klassifizierer `1#0:0#0` passt sowohl zu der Nachricht `100` als auch zu der Nachricht `110`. Im Falle der Nachricht `100` wird seine Aktion zu `000`, bei der Nachricht `110` zu `010`.

Diese Form der Repräsentation von Bedingungen und Aktionen stellt eine radikal vereinfachte Variante der Broadcast-Language dar. Die Verwendung von Wildcard-Symbolen erlaubt eine Generalisierung: Klassifizierer können zu mehreren Nachrichten passen und eine von der jeweiligen Nachricht abhängige Aktion haben. Je weniger Wildcard-Symbole ein Klassifizierer enthält, um so spezifischer ist er; als Maß für die *Spezifizität* eines Klassifizierers dient der Anteil von Nicht-Wildcard-Zeichen, die er enthält.
Neben Bedingung und Aktion verfügt jeder Klassifizierer über ein weiteres Attribut, die *Stärke* (strength) $p \in \mathbb{R}$. Diese schätzt den Return, den der Klassifizierer zu erwarten hat, ab und dient dem Genetischen Algorithmus als Fitnesswert des Klassifizierers.

Performance-Komponente

In jedem Lernschritt erhält das System eine externe Nachricht und schreibt diese auf eine *Nachrichtenliste* (message-list) fester Länge k. Die Population wird dann nach Klassifizierern durchsucht, die zu einer der Nachrichten in dieser Liste passen; diese Klassifizierer bilden das sogenannte *Match-Set* M und konkurrieren darum, aktiviert zu werden. Dazu gibt jeder Klassifizierer cl im Match-Set ein *Gebot (Bid)* ab, das seiner Stärke p und seiner Spezifizität proportional ist ($\rho << 1$):

$$\text{Bid} = \rho \cdot p \cdot \text{Spezifizität} \tag{2.21}$$

Basierend auf diesen Geboten werden dann $k - 1$ Klassifizierer aus dem Match-Set *aktiviert*. Die Auswahl erfolgt entweder deterministisch – die Klassifizierer mit den höchsten Geboten werden ausgewählt – oder probabilistisch in Form einer auf die Gebote bezogenen Rouletterad-Selektion. Die aktivierten Klassifizierer schreiben ihre Aktionen als Nachrichten in die zuvor geleerte Nachrichtenliste, die Aktion des aktivierten Klassifizierers mit dem höchsten Gebot wird als externe Aktion in der Umwelt ausgeführt.

Reinforcement-Komponente

Die Aktualisierung der Stärkewerte der Klassifizierer erfolgt in drei Schritten. Zuerst wird die Stärke aller aktivierten Klassifizierer um ihr jeweiliges Gebot verringert. Die Gebote werden im sogenannten *Bucket* gesammelt. Im zweiten Schritt wird der Inhalt des Buckets auf alle im vorangegangenen Lernschritt aktivierten Klassifizierer zu gleichen Teilen verteilt, deren Stärke also entsprechend erhöht. Zuletzt wird, falls das System eine Belohnung von der Umwelt erhält, diese an die aktivierten Klassifizierer verteilt – entweder zu gleichen Teilen oder unter Bevorzugung des aktivierten Klassifizierer mit dem höchsten Gebot.

Von Belohnungen profitieren somit unmittelbar nur die aktivierten Klassifizierer. Wenn diese das nächste mal um eine Aktivierung konkurrieren geben sie jedoch gemäß (2.21) aufgrund ihrer nun größeren Stärke ein höheres Gebot ab, das im Falle der Aktivierung an die direkt vor ihnen aktivierten Klassifizierer abgegeben wird. Diese können somit in Zukunft ebenfalls höhere Gebote machen. Auf diese Weise werden im Zuge mehrfacher Wiederholungen ähnlicher Zustände die Belohnungen immer weiter zurück gereicht und auf eine ganze Kette hintereinander aktivierter Klassifizierer verteilt. Holland bezeichnet diesen Algorithmus als *Bucket-Brigade*.

Discovery-Komponente

Für die Erzeugung neuer Regeln benutzt das System einen einfachen Genetischen Algorithmus vom Steady-State-Typ, der in jedem Lernschritt mit einer Wahrscheinlichkeit p_{GA} ausgeführt wird. Die Stärke der Klassifizierer wird als Fitness benutzt, um durch Rouletterad-Selektion zwei Klassifizierer als Eltern auszuwählen. Diese werden rekombiniert und mutiert, dabei kommen 1-Punkt-Crossing-Over und ein ternärer Punktmutationsoperator[19] zum Einsatz. Die so erzeugten neuen Klassifizierer ersetzen zwei Individuen der Population, die durch eine Rouletterad-Selektion bezüglich dem Reziproken der Stärke ausgewählt werden.

Die Aufgabe des Genetischen Algorithmus besteht hier – anders als bei „normalen" Optimierungsproblemen – nicht darin, *eine* beste Lösung zu finden, sondern darin, einen Satz kooperierender Regeln zu evolvieren, die das betrachtete Reinforcement-Learning-Problem gemeinsam lösen. Die Bucket-Brigade bewirkt, dass sich Subpopulationen von Klassifizierern entwickeln können, die jeweils für einen Teil der möglichen Zustände der Umwelt und des entsprechenden Verhaltens „zuständig" sind [Booker 1982].

2.3.4 Wilsons ZCS

In der Praxis zeigte sich, dass das gewünschte Lernverhalten und die erhoffte Leistung mit Hollands System nur schwer zu realisieren sind [Bull u. Kovacs 2005]. Dass auch diverse an dem System vorgenommene Modifikationen des Systems – für einen Überblick siehe [Wilson u. Goldberg 1989], [Lanzi u. Riolo 2000] – daran nichts änderten, führte Stewart W. Wilson darauf zurück, dass die komplexe Struktur von Hollands System eine tief gehende Analyse und ein genaues Verständnis des Lernverhaltens des Systems unmöglich machten [Wilson 1994]. Folgerichtig entwickelte er ein radikal vereinfachtes Lernendes Klassifizierendes System, das *Zeroth Level Classifier System* (ZCS). Dieses unterscheidet sich von Hollands System hauptsächlich durch die Reinforcement-Komponente und das Fehlen einer Nachrichtenliste, also dem Verzicht auf interne Nachrichten und ein „Gedächtnis". Beim Problemlösen können somit keine Informationen aus früheren Lernschritten verwendet werden, sondern nur die im aktuellen Umgebungszustand enthaltenen. Während Hollands System prinzipiell auch in Nicht-Markov-Umgebungen eingesetzt werden kann, ist das ZCS daher auf Markov-Entscheidungsprozesse beschränkt[20]. In diesem Rahmen aber ist – wie bei traditionellen

[19] Der ternäre Punktmutationsoperator arbeitet analog dem in 2.1.4 vorgestellten binären – bei Mutation eines Gens wird das vorhandene Allel (`0`,`1` oder #) zufällig gegen eines der beiden anderen ausgetauscht.

[20] Allerdings existiert auch eine Variante des ZCS mit einem (stark vereinfachten) internen Speicher, die auch für „leichte" Nicht-Markov-Umgebungen geeignet ist [Cliff u. Ross 1994].

Reinforcement-Learning-Verfahren auch – das Lösen eines Problems in mehreren Schritten möglich. Die Struktur des ZCS ist in Abbildung 2.8 dargestellt.

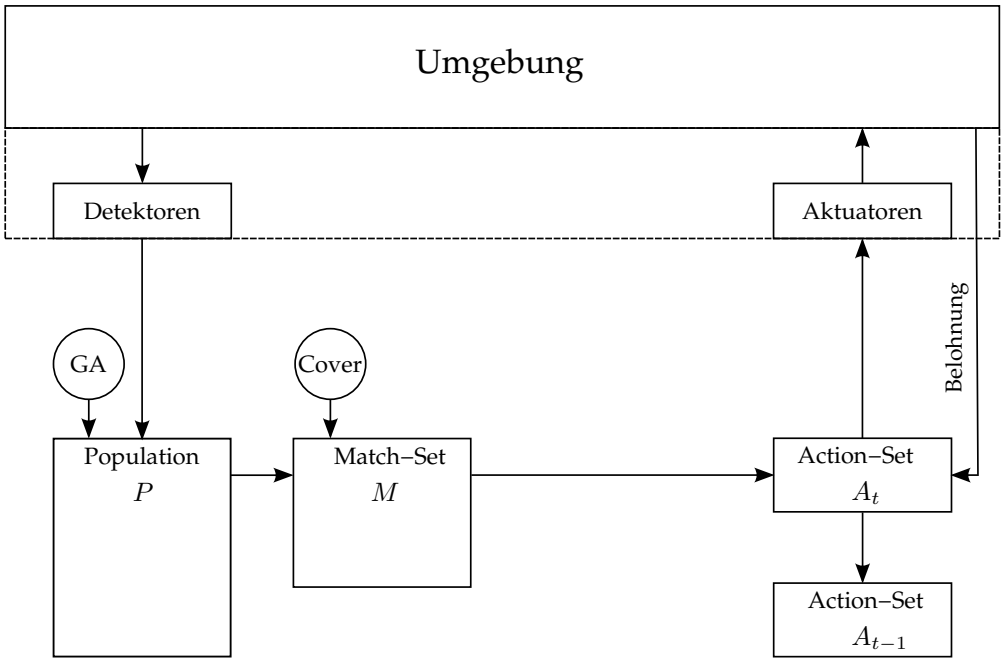


Abbildung 2.8: Schematische Darstellung von Wilsons ZCS. (Nach [Bull u. Kovacs 2005], verändert.)

Die mit dem ZCS erzielten Ergebnisse zeigten, dass Hollands Ideen auch in einem stark vereinfachten Rahmen funktionieren können; in einigen Testproblemen erzielte das ZCS sogar optimale Ergebnisse [Bull u. Hurst 2002].

Wissensbasis/Population

Wie das System von Holland benutzt das ZCS eine Klassifizierer-Population P fester Größe. Die Klassifizierer selbst unterscheiden sich von denen Hollands nur im Aktionsteil. In diesem sind keine Wildcard-Symbole mehr erlaubt und er kann, aufgrund des Verzichts des ZCS auf die Verwendung interner Nachrichten, eine andere Länge als Nachrichten und Bedingungen haben.

Performance-Komponente

In jedem Lernschritt wird aus den Klassifizierern der Population, die zum Zustand der Umwelt passen, das Match-Set M gebildet. Zur Bestimmung der externen Aktion des Systems wird durch Rouletterad-Selektion bezüglich der Stärke ein Klassifizierer ausgewählt[21]; alle in M enthaltenen Klassifizierer mit dem gleichen Aktionsteil wie dieser bilden das sogenannte *Action-Set* A.

Reinforcement-Komponente

Die Stärkewerte der Klassifizierer des aktuellen Action-Sets und des Action-Sets des vorangegangenen Lernschrittes werden wie folgt aktualisiert: Die Stärkewerte p der im

[21] Die Wahrscheinlichkeit für eine bestimmte Aktion ist somit proportional der Summe der Stärken aller Klassifizierer im Match-Set, die diese Aktion vorschlagen.

Action-Set enthaltenen Klassifizierer werden um einen Anteil βp, $0 < \beta \leq 1$, verringert; die Anteile werden im Bucket B gesammelt. Ein Anteil γB, $0 < \gamma \leq 1$, des Buckets wird anschließend gleichmäßig auf die Klassifizierer des Action-Sets des vorherigen Lernschrittes verteilt. Erhält das System eine direkte Belohnung r, so wird ein Anteil βr dieser Belohnung gleichmäßig auf die Klassifizierer des aktuellen Action-Sets verteilt.
Den Zweck dieser, als *implizite Bucket-Brigade* bezeichneten, Prozedur verdeutlicht die folgende Überlegung: Die Klassifizierer des Action-Sets A_t im Lernschritt t sind gerade die Elemente der Population, die auf den Zustand s_t der Umwelt anwendbar sind und die ausgeführte Aktion a_t vorschlagen. Die Summe p_{A_t} ihrer Stärkewerte kann somit als Maß für die Nützlichkeit der Aktion a_t in der Situation s_t angesehen werden. Die Aktualisierungen in den Lernschritten t und $t+1$ verbessern die Abschätzung von p_{A_t}:

$$p_{A_t} \leftarrow p_{A_t} + \beta \left\{ r + \gamma p_{A_{t+1}} - p_{A_t} \right\} \tag{2.22}$$

Diese Formel ist strukturell identisch mit der der Aktualisierungsregel (2.17) des SARSA-Algorithmus. Der Unterschied zu SARSA liegt im Wesentlichen darin, dass beim ZCS die Bewertung von Zustands-Aktions-Paaren implizit durch die Bewertung von Action-Sets beziehungsweise der diesen angehörenden Regeln erreicht wird, während SARSA Zustands-Aktions-Paare durch die Aktions-Wertefunktion Q explizit bewertet [Wilson 1994].
Die Reinforcement-Komponente des ZCS umfasst einen weiteren Aktualisierungsschritt: Die Stärke aller Klassifizierer, die im Match-Set, aber nicht im Action-Set enthalten sind, wird um einen Bruchteil $0 < \tau \leq 1$ reduziert; zum Zustand der Umwelt passende Klassifizierer, die eine andere als die ausgeführte Aktion haben, werden somit bestraft. In Verbindung mit der Rouletterad-Selektion zur Auswahl der auszuführenden Aktion ist dies der Ansatz, mit dem das ZCS dem exploration-exploitation-Problem begegnet: Die Exploration von Alternativen wird zugunsten der Ausnutzung vorhandenen Wissens, also der Ausführung der als am besten angesehenen Aktion, immer mehr eingeschränkt.

Discovery-Komponente

Das ZCS verwendet zur Einführung neuer respektive zur Verbesserung bestehender Regeln zwei Mechanismen. Dies ist zum einen ein Genetischer Algorithmus, der wie bei dem System von Holland arbeitet, zum anderen ein *Covering-Operator*. Dieser wird eingesetzt, wenn entweder das Match-Set leer ist – die Population also keine zu einer externen Nachricht passenden Klassifizierer enthält – oder die Summe der Stärkewerte der Klassifizierer im Match-Set kleiner ist als ein festgelegter Bruchteil des Durchschnitts der Stärkewerte der gesamten Population. In diesen Fällen erzeugt der Covering-Operator einen zu der externen Nachricht passenden Klassifizierer mit einer zufälligen Anzahl von Wildcard-Symbolen in der Bedingung, einer zufällig gewählten Aktion und einem Stärkewert, der dem Durchschnitt der Stärkewerte der Population entspricht. Dieser ersetzt einen Klassifizierer der Population, der durch eine auf die reziproken Stärkewerte bezogene Rouletterad-Selektion bestimmt wird. Nach Einsatz des Covering-Operators wird das Match-Set neu gebildet und wie üblich fortgefahren.

2.4 Das eXtended Classifier System

Wiederum Stewart W. Wilson war es, der 1995 das eXtended Classifier System (XCS) einführte ([Wilson 1995]), das sich von früheren Lernenden Klassifizierenden Systemen

in der Art der Fitnessberechnung für den Genetischen Algorithmus unterscheidet. Das XCS entwickelte sich in der Folgezeit zum De-facto-Standard und wurde bereits in einer Reihe von Realwelt-Problemen erfolgreich eingesetzt, beispielsweise in den Bereichen Datamining und Robotik [z.B. Bull 2004].

2.4.1 Stärke vs. Genauigkeit

Die meisten früheren Lernenden Klassifizierenden Systeme, auch das System von Holland und das ZCS, benutzen die Stärke eines Klassifizierers – die Höhe seines vorhergesagten Returns – als Maß für dessen Fitness. Beim XCS hingegen wird die Fitness von der Genauigkeit dieser Vorhersage bestimmt [Wilson 1995]. Diese neue Form der Fitnessberechnung ist durch verschiedene Probleme der stärkebasierten Methode motiviert:

- Im Allgemeinen führen selbst optimale Aktionen eines Lernenden Klassifizierenden Systems in verschiedenen Zuständen der Umwelt zu unterschiedlich hohen Belohnungen. Der Return, den ein Klassifizierer zu erwarten hat und damit seine Stärke/Fitness ist daher nicht nur von seiner Aktion abhängig, sondern auch von den Zuständen, in denen er anwendbar ist. Klassifizierer, die in „gewinnträchtigen" Bereichen der Umwelt anwendbar sind, werden sich dementsprechend häufiger vermehren als in Zuständen mit niedrigerem Return-Niveau passende Klassifizierer. Diese *gierige Erzeugung von Klassifizierern* kann dazu führen, dass die in den letztgenannten Zuständen anwendbaren Klassifizierer mit der Zeit verloren gehen, was zur Schwächung des Gesamtsystems führt[Cliff u. Ross 1994].

- Bei Lernproblemen mit verzögerten Belohnungen ist eine ganze Kette nacheinander aktiver Klassifizierer nötig, um letztlich eine Belohnung zu erhalten. Die bei der Weitergabe von Belohnungen an früher aktive Klassifizierer übliche Diskontierung führt dazu, dass Klassifizierer am Ende der Kette höhere Stärkewerte haben als solche an deren Anfang. Letztere werden daher vom Genetischen Algorithmus „ausgesondert" und längere Ketten können nicht unterhalten werden.

- Durch die Verwendung von Wildcard-Symbolen können Klassifizierer in mehreren Situationen anwendbar sein. Eine solche Generalisierung ist prinzipiell wünschenswert, wird aber problematisch, wenn ein Klassifizierer *übergenerell* wird – wenn seine Aktion nicht in allen Zuständen, auf die er passt, optimal ist. Der Genetische Algorithmus kann nicht zwischen einem solchen übergenerellen Klassifizierer und einem Klassifizierer gleicher Stärke, dessen Aktion stets optimal ist, unterscheiden. Die stärkebasierte Fitnessberechnung liefert also keinen Anreiz für die Entwicklung von Klassifizierern, die generell sind, aber nicht übergenerell.

- Eine Verschärfung des Problems übergenereller Klassifizierer stellen die *starken übergenerellen Klassifizierer* dar – Klassifizierer, die in Bereichen mit hohem Return-Niveau optimal sind und suboptimal in Zuständen mit niedrigem Return-Niveau. Ein solcher Klassifizierer wird aufgrund seiner größeren Stärke einen nur im Bereich niedriger Belohnungen anwendbaren, dort aber optimalen Klassifizierer sowohl bei der Aktionswahl als auch bei der Reproduktion dominieren und kann so zu dessen „Aussterben" führen [Kovacs 2001].

Insbesondere die beiden letzten Punkte geben Anlass dazu, bei der Fitnessberechnung die Genauigkeit der Return-Voraussagen der Klassifizierer zu berücksichtigen. In

[Kovacs 1997] wurde gezeigt, dass diese neue Art der Fitnessberechnung es dem XCS unter bestimmten Bedingungen ermöglicht, Regelsätze zur Lösung Boolescher Funktionen zu finden, die exakt, vollständig und minimal sind; insbesondere sind die Klassifizierer, aus denen diese Regelsätze bestehen, so generell wie möglich, ohne übergenerell zu sein. Ferner wurde in [Kovacs 2000] ein Vergleich zwischen traditioneller stärkebasierter Fitnessberechnung und der auf Genauigkeit basierenden Fitnessberechnung des XCS vorgenommen, der der letztgenannten Methode signifikante Vorteile bescheinigt – insbesondere eine geringere Anfälligkeit für die Bildung starker übergenereller Klassifizierer.
Die Einführung der genauigkeitsbasierten Fitnessberechnung führt zu einer Annäherung Lernender Klassifizierender Systeme an traditionelle Reinforcement-Learning-Verfahren: Lernende Klassifizierende Systeme mit stärkebasierter Fitnessberechnung konzentrieren sich auf „gewinnträchtige" Zustands-Aktions-Paare: Zum einen entwickeln sie oft keine Klassifizierer für Zustände, von denen aus nur niedrige Belohnungen erreicht werden können. Zum anderen suchen sie stets nur die beste Aktion für einen Zustand und kümmern sich kaum um die Konsequenzen als suboptimal eingeschätzter Aktionen. Fehleinschätzungen sind daher nur sehr schwer zu korrigieren.
Das XCS hingegen lernt eine vollständige Zuordnung von Returns zu Zustands-Aktions-Paaren – in der Terminologie des Reinforcement-Learning: eine Aktions-Wertefunktion. Dabei werden die einzelnen Zustands-Aktions-Paare jedoch nicht vollständig aufgelöst; die Generalisierungsmöglichkeiten des XCS gestatten es, dass ein Klassifizierer mehrere Zustands-Aktions-Paare zusammenfasst. Anders als die im Abschnitt 2.2.3 vorgestellten traditionellen, auf der Tabellierung von Wertefunktionen beruhenden, Reinforcement-Learning-Verfahren ist das XCS daher auch in Lernumgebungen mit sehr vielen Zuständen und Aktionen einsetzbar [Bull u. Kovacs 2005].

2.4.2 Welches XCS?

Der Einführung des XCS im Jahre 1995 folgend, wurde eine Reihe von Modifikationen und Erweiterungen vorgeschlagen, von denen einige zu festen Bestandteilen des Systems wurden. Als De-Facto-Standard kann die in [Butz u. Wilson 2000] vorgeschlagene Variante angesehen werden, die insbesondere die in [Wilson 1998] vorgenommenen Änderungen am Genetischen Algorithmus, die Subsumtions-Operatoren [Wilson 1998], die in [Kovacs 1999] eingeführte Löschmethode und die in [Wilson 2000a] vorgeschlagene Form der Genauigkeits-Berechnung beinhaltet. Die im Folgenden beschriebene Variante des XCS, deren Struktur in 2.9 dargestellt ist, weicht von dieser Standard-Form nur durch die Verwendung des Spezifizierungs-Operator [Lanzi 1997, 1999] ab.

2.4.3 Wissensbasis/Population

Das XCS benutzt keine Population fester Größe, sondern gibt nur eine Maximalgröße $N_{\max} \in \mathbb{N}$ der Population P vor. Diese wird zumeist leer initialisiert und erst während des Lernens gefüllt.

Die Klassifizierer des XCS sind etwas komplizierter gebaut als die der früheren Systeme. Als Bedingungen werden auch beim XCS Zeichenketten der Länge n über dem ternären Alphabets $\{0, 1, \#\}$ verwendet; der Aktionsteil ($action$) eines Klassifizierers schlägt eine der in der Lernumgebung bekannten Aktionen vor: $action \in \mathcal{A} = \{a_1, a_2, \cdots, a_m\}$; die Aktionen müssen jedoch nicht binär kodiert sein. Neben Bedingung und Aktion verfügen

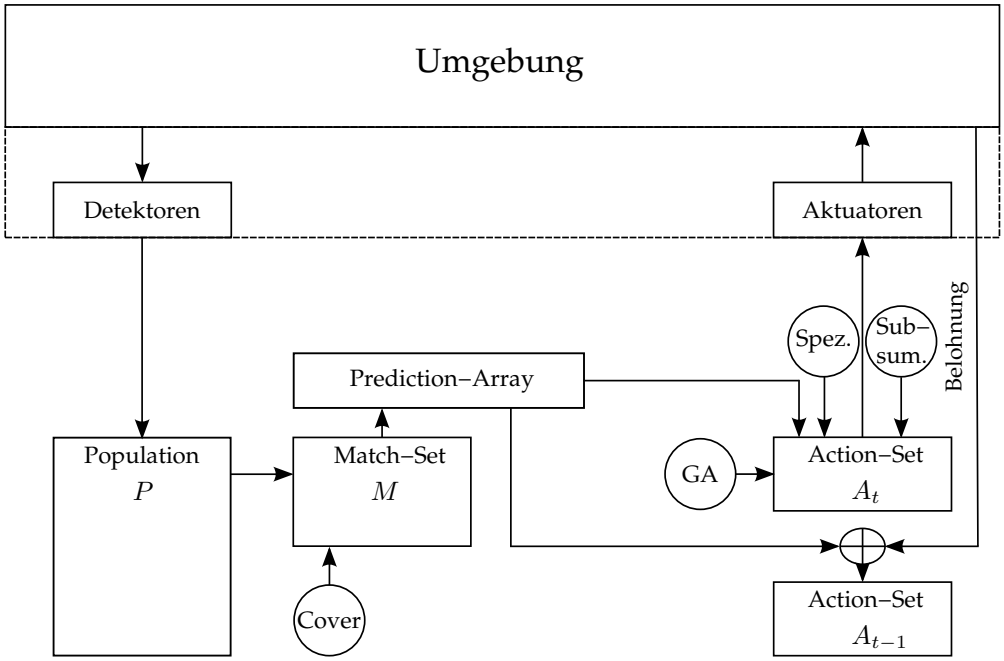


Abbildung 2.9: Schematische Darstellung von Wilsons XCS. (Nach [Bull u. Kovacs 2005], verändert.)

die Klassifizierer des XCS über eine Reihe von Attributen: Die *Erfahrung (experience)* $exp \in \mathbb{N}$ eines Klassifizierers gibt an, wie oft dieser in einem Action-Set enthalten war und damit auch, wie oft er bewertet wurde. Der Stärke der früheren Systeme entspricht beim XCS die *Vorhersage (prediction)* $p \in \mathbb{R}$, sie schätzt den zu erwartenden Return des Klassifizierers ab. Der Fehler dieser Abschätzung wird durch den *Vorhersagefehler (prediction error)* $\varepsilon \in \mathbb{R}$ angegeben. Basierend auf dem Vorhersagefehler wird die *Genauigkeit (accuracy)* κ des Klassifizierers berechnet, die ihrerseits die Grundlage für die Berechnung seiner Fitness $f \in \mathbb{R}$ bildet. Weiterhin speichert jeder Klassifizierer die durchschnittliche *Action-Set-Größe (action set size)* $as \in \mathbb{R}$ der Action-Sets, denen er angehört hat und ist mit einem *Zeitstempel (time stamp)* $ts \in \mathbb{N}$ versehen, der angibt, wann zuletzt der Genetische Algorithmus auf einem dieser Action-Sets ausgeführt wurde.
Die Population eines Lernenden Klassifizierenden Systems kann natürlich mehrere Kopien eines Klassifizierers enthalten, in diesem Fall müssen normalerweise viele Berechnungen mehrfach – für jede Kopie einzeln – durchgeführt werden. Um dies zu umgehen, sind die Klassifizierer des XCS mit einer *Vielfachheit (numerosity)* num versehen, die angibt, in wie vielen Kopien der Klassifizierer vorliegt; sie werden daher auch als *Makroklassifizierer* bezeichnet. Ein neuer Klassifizierer wird nur dann (mit Vielfachheit 1) tatsächlich in die Population eingefügt, wenn nicht bereits ein Klassifizierer mit gleicher Bedingung und Aktion existiert; andernfalls wird die Vielfachheit des existierenden Klassifizierers erhöht. Entsprechend wird auch beim Löschen nur die Vielfachheit eines Klassifizierers verringert, es sei denn er hat bereits nur noch die Vielfachheit 1. Auch bei allen anderen Operationen des XCS wird die Vielfachheit berücksichtigt, sodass das System sich verhält, als arbeite es mit „normalen“ Klassifizierern.

Im Folgenden wird die aus der objektorientierten Programmierung bekannte Punktnotation verwendet, um ein Attribut eines Klassifizierers zu referenzieren, beispielsweise wird mit $cl.exp$ die Erfahrung eines Klassifizierers cl bezeichnet.

2.4.4 Performance-Komponente

Auch das XCS bildet in jedem Lernschritt zunächst das Match-Set M aus allen Klassifizierern der Population, die zum Zustand s der Umwelt passen. Dann wird für jede Aktion $a \in \mathcal{A}$ der Durchschnitt der Vorhersagen aller im Match-Set enthaltenen Klassifizierer mit dieser Aktion berechnet; durch eine Gewichtung mit der Fitness werden dabei Klassifizierer mit genauerer Vorhersage stärker berücksichtigt:

$$PA_a = \sum_{\substack{cl \in M, \\ cl.action = a}} cl.p \cdot cl.f \cdot cl.num \; / \sum_{\substack{cl \in M, \\ cl.action = a}} cl.f \cdot cl.num \tag{2.23}$$

Falls es zu einer Aktion a keinen Klassifizierer im Match-Set gibt, ist PA_a undefiniert. Die *Systemvorhersagen* PA_a bilden den *Prediction-Array* PA, der die Grundlage für die Auswahl der auszuführenden Aktion bildet: Mit der *Explorationswahrscheinlichkeit* p_{explr} erfolgt die Auswahl einer der im Match-Set vertretenen Aktionen zufällig, mit Wahrscheinlichkeit $1 - p_{\text{explr}}$ wird die Aktion mit der höchsten System-Vorhersage gewählt. Je nachdem, ob die Aktionswahl rein zufällig ($p_{\text{explr}} = 1$) oder unter Bevorzugung der als am besten eingeschätzten Aktion erfolgt ($0 < p_{\text{explr}} < 1$), wird zwischen *reiner* und *bevorzugender Exploration* unterschieden. Alle im Match-Set enthaltenen Klassifizierer mit der ausgewählten Aktion bilden das Action-Set.

2.4.5 Reinforcement-Komponente

Die Reinforcement-Komponente des XCS aktualisiert in jedem Lernschritt t zunächst die im Action-Set A_{t-1} des vorangegangenen Zeitschrittes enthaltenen Klassifizierer – sofern nicht mit dem Lernschritt t eine neue Lernepisode beginnt und daher A_{t-1} leer ist.
Für jeden dieser Klassifizierer wird zunächst der Erfahrungswert erhöht:

$$exp \leftarrow exp + 1 \tag{2.24}$$

Dann wird die Vorhersage p an die aktuelle Returnabschätzung

$$\mathcal{P} = \mathcal{P}_t = r_t + \gamma \max_i PA_{a_i}, \tag{2.25}$$

die sich aus der Belohnung r_t des Lernschrittes $t-1$ und dem diskontierten Maximum der aktuellen Systemvorhersagen ergibt, angepasst. Weiterhin wird der Vorhersagefehler ε an die Differenz $|\mathcal{P} - p|$ von Vorhersage und aktuellem Return und die Abschätzung as der Action-Set-Größe an die Mächtigkeit $|A_{t-1}| = \sum_{cl \in A_{t-1}} cl.num$ des Action-Sets A_{t-1} angepasst. Die Stärke der Anpassung wird von der Lernrate β bestimmt, die auch festlegt, in welcher Form die Aktualisierung erfolgt.
Bei unerfahrenen ($exp < 1/\beta$) Klassifizierern kommt die *Moyenne-Adaptive-Modifiée-Technik* [Venturini 1994] zum Einsatz, die die Attributwerte als Durchschnitt der früheren und der aktuellen Werte von Return, Fehler beziehungsweise Action-Set-Größe bestimmt, sodass sich die Attribute schnell den „wahren" Werten nähern und der Einfluss der Initialwerte reduziert wird:

$$p \leftarrow p + (\mathcal{P} - p)/exp \tag{2.26}$$

$$\varepsilon \leftarrow \varepsilon + (|\mathcal{P} - p| - \varepsilon)/exp \tag{2.27}$$

$$as \leftarrow as + (|A_{t-1}| - as)/exp \tag{2.28}$$

Bei erfahrenen ($exp \geq 1/\beta$) Klassifizierern erfolgt die Aktualisierung gemäß der *Widrow-Hoff-Delta-Regel* [Widrow u. Hoff 1960]:

$$p \leftarrow p + \beta \cdot (\mathcal{P} - p) \tag{2.29}$$

$$\varepsilon \leftarrow \varepsilon + \beta \cdot (|\mathcal{P} - p| - \varepsilon) \tag{2.30}$$

$$as \leftarrow as + \beta \cdot (|A_{t-1}| - as) \tag{2.31}$$

Die Aktualisierung der Vorhersage p entspricht bei erfahrenen Klassifizierern somit strukturell der Aktualisierung der Aktions-Wertefunktion (2.18) beim Q-Learning; der Unterschied zum Q-Learning liegt im Wesentlichen darin, dass die Vorhersage p eines Klassifizierer nicht die Bewertung eines Zustands-Aktions-Paares darstellt, sondern, aufgrund der Generalisierungs-Möglichkeit in der Klassifizierer-Bedingung, eine Bewertung mehrerer Zustands-Aktions-Paare.
Im Anschluss an diese Aktualisierungen wird die Genauigkeit κ der Klassifizierer berechnet. Ein Klassifizierer gilt als absolut genau, wenn sein Fehler ε einen Schwellenwert ε_0 unterschreitet, der üblicherweise im Bereich von etwa einem Prozent der maximal möglichen Belohnung liegt. Andernfalls berechnet sich die Genauigkeit eines Klassifizierers gemäß einer Potenzfunktion:

$$\kappa = \begin{cases} 1 & \text{für } \varepsilon < \varepsilon_0 \\ \alpha \cdot \left(\frac{\varepsilon}{\varepsilon_0}\right)^{-\nu} & \text{für } \varepsilon \geq \varepsilon_0 \end{cases} \tag{2.32}$$

Die Werte der Parameter α und ν sind problemabhängig experimentell zu bestimmen.
Nachdem alle anderen Attribute aktualisiert sind, wird die Fitness der Klassifizierer des Action-Sets A_{t-1} an ihre relative Genauigkeit angepasst, wiederum in Form einer Widrow-Hoff-Delta-Regel:

$$f \leftarrow f + \beta \left(\frac{\kappa}{\sum_{cl \in A_{t-1}} cl.\kappa \cdot cl.num} - f \right) \tag{2.33}$$

Sofern im Lernschritt t das Ende einer Episode des Reinforcement-Learning-Problems – also ein Terminalzustand – erreicht wird, folgt der Aktualisierung des Action-Sets A_{t-1} eine analoge Aktualisierung des aktuellen Action-Sets A_t, die als Returnabschätzung jedoch die aktuelle Belohnung verwendet:

$$\mathcal{P} = \mathcal{P}_t = r_{t+1} \tag{2.34}$$

2.4.6 Discovery-Komponente

Die Discovery-Komponente des XCS beinhaltet neben Covering-Operator und Genetischem Algorithmus zwei weitere Mechanismen, Subsumtion und Spezifizierung, die auf die Entwicklung genereller und genauer, also nicht übergenereller, Klassifizierer hinwirken.

Covering-Operator

Das XCS sieht vor, dass im Match-Set immer eine Mindestzahl von Θ_{MNA} verschiedenen Aktionen vertreten ist[22] . Ist dies nicht der Fall, werden durch den Covering-Operator zur

[22] MNA steht für *minimal number of actions*. Meist entspricht Θ_{MNA} der Anzahl der in der Umwelt möglichen Aktionen.

aktuellen Eingabe passende Klassifizierer mit zufälligen, nicht im Match-Set vertretenen Aktionen erzeugt und in die Population eingefügt. An jeder Stelle der Bedingung dieser Klassifizierer wird mit Wahrscheinlichkeit $p_\#$ ein Wildcard-Symbol gesetzt.

Genetischer Algorithmus

Der Genetische Algorithmus des XCS operiert nicht auf der Population, sondern auf den Action-Sets. Er wird jeweils nach der Aktualisierung eines Action-Sets A ausgeführt, sofern die durchschnittlich vergangene Zeit, in der die Klassifizierer dieses Action-Sets keinem Genetischen Algorithmus unterworfen waren, größer ist als ein Schwellenwert Θ_{GA}:

$$\frac{\sum_{cl\in A}(t - cl.ts)\cdot cl.num}{\sum_{cl\in A} cl.num} \stackrel{!}{>} \Theta_{\mathrm{GA}} \tag{2.35}$$

Diese Bedingung ist wie folgt motiviert: Würde der Genetische Algorithmus einfach (mit einer gewissen Wahrscheinlichkeit) in jedem Action-Set ausgeführt, würde ein Klassifizierer, der zu häufig auftretenden Zuständen passt – also häufig in einem Action-Set enthalten ist – sich öfter reproduzieren als ein Klassifizierer mit gleicher Fitness, dessen Bedingung von eher selten beobachteten Zuständen erfüllt wird. Da die vom Genetischen Algorithmus erzeugten Klassifizierer meist wieder zu dem Zustand passen, der vor der Bildung des Action-Sets, dem ihre Eltern entnommen wurden, beobachtet wurde, ergäbe sich somit eine Konzentration der Population auf Bereiche häufig auftretender Zustände. Die Bedingung (2.35) bewirkt, dass ein Klassifizierer innerhalb von Θ_{GA} Lernschritten im Mittel nur einmal Gelegenheit zur Reproduktion erhält, egal wie oft er währenddessen einem Action-Set angehört. Damit erhalten alle Klassifizierer – abgesehen von nur extrem selten einem Action-Set angehörenden – etwa gleich oft die Gelegenheit, als Eltern selektiert zu werden und die Population verteilt sich gleichmäßig über den gesamten Zustandsraum.
Der Genetische Algorithmus wählt durch Rouletterad-Selektion zwei Klassifizierer des Action-Sets als Eltern aus und erzeugt zwei neue Klassifizierer, indem die Eltern mit Wahrscheinlichkeit p_{cross} einem 2-Punkt-Crossing-Over unterzogen und dann mutiert werden. Die Bedingung eines Klassifizierers wird mutiert, indem jedes ihrer Gene mit Wahrscheinlichkeit p_{mut} derart verändert wird, dass der Klassifizierer noch zu dem Zustand s passt, der zur Bildung des Action-Sets führte: Ein spezifisches Allel der Bedingung wird durch ein Wildcard-Symbol ersetzt, ein Wildcard-Symbol gegen das durch den Zustand s bestimmte Allel. Der Aktionsteil eines Klassifizierers wird mit Wahrscheinlichkeit p_{mut} mutiert, indem seine Aktion gegen eine zufällig gewählte andere Aktion ausgetauscht wird. Ob die neuen Klassifizierer in die Population eingefügt werden, entscheidet ein *Subsumtions-Operator*.

Subsumtions-Operatoren

Die beiden Subsumtions-Operatoren [Wilson 1998] des XCS fördern die Entwicklung genereller und genauer Klassifizierer, indem sie „Spezialfälle“ solcher Klassifizierer aus der Population entfernen. Zur näheren Erläuterung zunächst zwei Definitionen:

(i) Ein Klassifizierer cl_1 heißt *genereller als* ein Klassifizierer cl_2, wenn die Menge der Zustände, in denen cl_1 anwendbar ist, die Menge der Zustände, in denen cl_2 anwendbar ist, echt enthält[23].

[23] Ein binärer Klassifizierer cl_1 ist also genereller als cl_2, wenn seine Bedingung an jeder Stelle entweder mit der von cl_2 übereinstimmt oder ein Wildcard-Symbol enthält und mindestens ein Wildcard-Symbol mehr enthält als die Bedingung von cl_2.

(ii) Ein Klassifizierer cl_1 *subsumiert* einen Klassifizierer cl_2, wenn er dieselbe Aktion wie cl_2 hat, hinreichend genau ($cl_1.\varepsilon < \varepsilon_0$), hinreichend erfahren ($cl_1.exp > \Theta_{\text{SUB}}$)[24] und genereller als cl_2 ist.

Der *GA-Subsumtions-Operator* überprüft, ob die vom Genetischen Algorithmus erzeugten Klassifizierer von ihren Eltern subsumiert werden. Ist das der Fall, wird der betroffene neue Klassifizierer nicht in die Population eingefügt; stattdessen wird die Vielfachheit des subsumierenden Elters erhöht.
Der *Action-Set-Subsumtions-Operator* wird nach jeder Aktualisierung eines Action-Sets aufgerufen. In diesem sucht er den Klassifizierer, der in den meisten Zuständen anwendbar ist (bei binären Bedingungen also den mit den meisten Wildcard-Symbolen), löscht alle im Action-Set enthaltenen Klassifizierer, die von diesem subsumiert werden und erhöht die Vielfachheit des subsumierenden Klassifizierers entsprechend der Anzahl und Vielfachheit der gelöschte Klassifizierer.

Spezifizierungs-Operator

Die Subsumtions-Operatoren können in Lernproblemen, deren Umwelt nur geringfügige Möglichkeiten der Generalisierung bietet, eine Tendenz zur Übergeneralisierung bewirken. Der *Spezifizierungs-Operator* [Lanzi 1997, 1999] wirkt dem entgegen, indem er übergenerelle Klassifizierer „korrigiert“:
Wenn in einem Lernschritt der durchschnittliche Vorhersagefehler der im Action-Set enthaltenen Klassifizierer über dem Zweifachen des durchschnittlichen Fehlers in der Population liegt, so wird durch eine auf den Vorhersagefehler bezogene Rouletterad-Selektion ein Klassifizierer aus dem Action-Set ausgewählt. Anschließend wird dann eine spezifischere Version dieses Klassifizierers erzeugt, indem jedes Wildcard-Symbol seiner Bedingung mit Wahrscheinlichkeit p_{spec} durch das entsprechende Symbol des korrespondierenden Umgebungszustandes ersetzt wird. Dieser neue Klassifizierer wird in die Population eingefügt.

Löschen von Klassifizierern

Das Einfügen neuer Klassifizierer durch den Covering-Operator, den Genetischen Algorithmus und den Spezifizierungs-Operator kann dazu führen, dass die Maximalgröße $N_{\max}$ der Population überschritten wird. In diesem Fall müssen Klassifizierer aus der Population gelöscht werden; die Auswahl der zu löschenden Klassifizierer erfolgt durch eine Rouletterad-Selektion bezüglich der *Löschvoten*. Mit der durchschnittlichen Fitness

$$\bar{f} = \frac{\sum_{cl \in P} cl.f \cdot cl.num}{\sum_{cl \in P} cl.num}$$

der Population P berechnet sich das Löschvotum $\text{vote}(cl)$ eines Klassifizierers cl als:

$$\text{vote}(cl) = \begin{cases} cl.as \cdot cl.num \cdot \frac{\bar{f}}{cl.f} & \text{für } cl.exp > \Theta_{\text{DEL}} \wedge cl.f < \delta \cdot \bar{f} \\ cl.as \cdot cl.num & \text{sonst} \end{cases} \tag{2.36}$$

Die Löschvoten basieren auf der Action-Set-Größe: Indem bevorzugt Klassifizierer gelöscht werden, die oft in großen Action-Sets enthalten sind, wird versucht, eine gleichmäßige Abdeckung des Zustandsraums durch Klassifizierer zu erreichen. Ferner steigt die Selektionswahrscheinlichkeit erfahrener Klassifizierer ($exp > \Theta_{\text{DEL}}$) proportional ihrer reziproken Fitness, sofern diese kleiner als ein Bruchteil δ der durchschnittlichen Fitness der Population ist. Diese Löschmethode wurde in [Kovacs 1999] eingeführt.

[24] Θ_{SUB} ist ein weiterer Systemparameter des XCS, der sogenannte Subsumtionsschwellenwert.

2.4.7 Klassifizierer für reellwertige Zustände

In vielen Reinforcement-Learning-Problemen, insbesondere in Realwelt-Problemen, sind die Zustände der Umwelt durch kontinuierliche Größen, beispielsweise Positionsangaben, bestimmt. Die traditionelle binäre Repräsentation ist daher oftmals unangemessen und Zustände sind viel natürlicher durch Tupel reeller Zahlen beschreibbar.
Es liegt daher nahe, das XCS so zu modifizieren, dass es mit reellwertigen Zuständen arbeitet. Die einzige grundlegende Änderung, die dazu nötig ist, betrifft die Klassifizierer-Bedingungen; diese müssen reellwertig repräsentiert werden, was noch einige kleinere Änderungen an den Operatoren der Discovery-Komponente nach sich zieht; an der Performance- und der Reinforcement-Komponente muss hingegen nichts verändert werden, da diese von der Gestalt der Bedingungen unabhängig sind.

Die verbreitetste Form von Bedingungen zur Verwendung mit reellwertigen Zuständen sind die *intervallbasierten* Bedingungen, diese haben die Gestalt $2n$-dimensionaler Tupel reeller Zahlen, von denen je zwei ein Intervall in $\mathbb{R}$ repräsentieren:

$$c = (\underbrace{c_1 \ , \ c_2}_{\downarrow \ [l_1, u_1)} \ , \ \underbrace{c_3 \ , \ c_4}_{\downarrow \ [l_2, u_2)} \ , \ \cdots \ , \ \underbrace{c_{2n-1} \ , \ c_{2n}}_{\downarrow \ [l_n, u_n)}) \tag{2.37}$$

Ein Zustand $s = (\xi_1, \xi_2, \ldots, \xi_n)^T \in \mathcal{S}$ erfüllt eine solche Bedingung, wenn:

$$\xi_i \in [l_i, u_i) \quad \text{für} \quad i = 1, 2, \ldots, n$$

Somit passt eine Bedingung zu allen Zuständen, die in dem Kreuzprodukt der Intervalle $[l_i, u_i), i = 1, 2, \ldots, n$ enthalten sind; sie deckt einen im Zustandsraum enthaltenen Hyperquader ab. Generalisierung erfolgt bei intervallbasierten Bedingungen somit quasi automatisch, Wildcard-Symbole sind nicht mehr nötig.
Die Berechnung der Intervallgrenzen l_i und u_i aus der Bedingung c kann auf verschiedene Weisen erfolgen, gebräuchlich sind die drei folgenden:

Centre-Spread-Repräsentation [Wilson 2000a]	$l_i = c_{2i-1} - c_{2i}$ $u_i = c_{2i-1} + c_{2i}$
Ordered-Bound-Repräsentation [Wilson 2000b]	$l_i = c_{2i-1}$ $u_i = c_{2i}$
Unordered-Bound-Repräsentation [Stone u. Bull 2003]	$l_i = \min(c_{2i-1}, c_{2i})$ $u_i = \max(c_{2i-1}, c_{2i})$

Tabelle 2.1: Repräsentation von Intervallgrenzen bei verschiedenen Typen intervallbasierter Bedingungen.

Jede dieser Repräsentationen hat spezifische Vor- und Nachteile [Stone u. L.Bull 2003; Stone u. Bull 2003], sodass die jeweils geeignete problemabhängig zu wählen ist.

Der Genetische Algorithmus benutzt bei Verwendung intervallbasierter Bedingungen einen 2-Punkt-Crossing-Over-Operator und reellwertige Punktmutation (siehe 2.1.4)[25].
Die Covering-Operator wird in naheliegender Weise angepasst: Er erzeugt Bedingungen mit zufälligen Intervallgrenzen, sodass der aktuelle Umgebungszustand in dem sich ergebenden Hyperquader liegt. Die zulässige Größe der einzelnen Intervalle wird dabei

[25] Bei der Ordered-Bound-Repräsentation muss gegebenenfalls durch Vertauschen zweier Einträge der Bedingung sichergestellt werden, dass stets $c_{2i-1} < c_{2i}, i = 1, 2, \ldots, n$ gilt.

meist durch Vorgabe einer maximalen Intervallbreite s_0 eingeschränkt. Auch die Subsumtionsmechanismen können leicht angepasst werden: Bei der Action-Set-Subsumtion wird der Klassifizierer gesucht, der den – bezogen auf das Volumen – größten Hyperquader des Zustandsraumes $\mathcal{S}$ abdeckt; ein intervallbasierter Klassifizierer cl_1 ist genereller als ein Klassifizierer cl_2, wenn der von cl_1 abgedeckte Teil des Zustandsraumes den von cl_2 abgedeckten Hyperquader echt enthält.

2.4.8 Funktionsapproximation mit dem XCS

Bisher wurden Lernende Klassifizierende Systeme hier stets aus einer klassischen Reinforcement-Learning-Perspektive betrachtet – als Systeme, deren Lernziel darin besteht, jedem Umgebungszustand die beste Aktion zuzuordnen. Dies ist per definitionem gerade die Aktion, deren Ausführung im jeweils betrachteten Umgebungszustand den höchsten Return nach sich zieht. Dementsprechend lernen Lernende Klassifizierende Systeme die richtige Zuordnung von Aktionen zu Zuständen, indem sie – ausgehend von den beobachteten Belohnungen respektive Returns – den Wert der verschiedenen Aktionen in den verschiedenen Lernumgebungszuständen abschätzen. Speziell im Falle des XCS stellen die als Grundlage der Aktionswahl in jedem Lernschritt berechneten Systemvorhersagen nichts anderes dar als eine Abschätzung der bei Ausführung der verschiedenen Aktionen im jeweils aktuellen Lernumgebungszustand zu erwartenden Returns.
Folglich kann das reellwertig kodierte XCS problemlos und unmodifiziert[26] auch zur Approximation reellwertiger, auf $\mathcal{S} \subseteq \mathbb{R}^n$ definierter Funktionen $f : \mathcal{S} \to \mathbb{R}$ eingesetzt werden: Die zu approximierende Funktion f wird zu diesem Zweck durch eine Lernumgebung repräsentiert, deren Zustandsraum $\mathcal{S} \subseteq \mathbb{R}^n$ dem Definitionsbereich von f entspricht und deren Aktionsmenge $\mathcal{A}$ lediglich eine einzige (Platzhalter-)Aktion enthält, durch deren Ausführung das XCS der Lernumgebung signalisieren kann, dass es für den nächsten Lernschritt bereit ist. Der Ausführung dieser Aktion folgt in jedem Fall die Auszahlung einer Belohnung, die dem Wert der zu approximierenden Funktion an der zuvor präsentierten Stelle des Zustandsraums entspricht.
Bei $\mathcal{S}$ handelt es sich im Allgemeinen um eine kontinuierliche Teilmenge des $\mathbb{R}^n$. Dann gibt es, da die maximale Populationsgröße $N_{\max}$, also die maximal zulässige Anzahl von Klassifizierern, klein ist im Vergleich zur Anzahl potentiell möglicher Zustände, stets zusammenhängende Bereiche des Zustandsraums, für die sich das gleiche Match-Set und damit auch die gleiche Systemvorhersage ergibt. Die vom XCS gelernte Approximation der Zielfunktion f ist somit stückweise konstant.
Um die Güte der erzielten Approximation zu verbessern, wurde in [Wilson 2002] vorgeschlagen, die durch einen skalaren Wert gegebene Klassifizierervorhersage p durch eine in jeder Komponente des Umgebungszustandes $s = (\xi_1, \xi_2, \ldots, \xi_n)^T \in \mathcal{S}$ lineare Funktion[27] zu ersetzen:

$$p(s) = v_0 \cdot \xi_0 + \sum_{i=1}^{n} v_i \cdot \xi_i \tag{2.38}$$

Neben dem vom Umgebungszustand s abhängigen zweiten Term enthält (2.38) den Offset-Term $v_0 \cdot \xi_0$, der durch den neu eingeführten Systemparameter ξ_0 bestimmt wird. Anstelle der Anpassung (2.26) respektive (2.29) der skalaren Klassifizierervorhersage p

[26] Der Einsatz des XCS zur Funktionsapproximation erfordert tatsächlich keinerlei Modifikation. Es ist aber natürlich sinnvoll, die Ausgabe des XCS, die sich im Normalfall auf die gewählte Aktion beschränkt, um die Systemvorhersage, also den geschätzten Funktionswert, zu erweitern.

[27] Prinzipiell sind auch kompliziertere Formen der Abhängigkeit denkbar, in [Lanzi u. a. 2005a] etwa sind die Klassifizierervorhersagen durch Polynome vorgegebenen Grades gegeben.

erfolgt bei dem so modifizierten XCS eine Anpassung des (Klassifizierer-spezifischen) Gewichtsvektors $\boldsymbol{v} = (v_0, v_1, \ldots, v_n)^T$. In [Wilson 2002] erfolgte diese Anpassung mittels einer modifizierten Widrow-Hoff-Regel:

$$v_i \leftarrow v_i + \frac{\beta_o}{\|s'\|^2} \cdot (f(s) - p(s)) \cdot \xi_i,\ i = 0, \ldots, n \tag{2.39}$$

Dabei bezeichnet $s' = (\xi_0, \xi_1, \xi_2, \ldots, \xi_n)^T$ den um ξ_0 „erweiterten" Zustand s, β_o die Lernrate für die Anpassung der Gewichte und $f(s)$ den Wert der zu approximierenden Funktion an der Stelle s, der dem System in Form des der Präsentation von s folgenden Returns mitgeteilt wird. Nach [Haykin 1999] kann diese Form der Anpassung angesehen werden als ein Gradientenabstiegsverfahren, das zum Ziel hat, die *Fehlerfunktion*

$$\zeta_{WH}(\boldsymbol{v}) = \frac{1}{2} (f(s) - p(s))^2 \tag{2.40}$$

zu minimieren. In [Lanzi u. a. 2005b] wurde experimentell gezeigt, dass die Approximation der Zielfunktion weiter verbessert werden kann, indem nicht nur der Approximationsfehler $f(s) - p(s)$ des aktuellen Lernschrittes, sondern auch der weiter zurückliegender Lernschritte berücksichtigt wird: Die *Methode der kleinsten Quadrate* gestattet es, die Gewichte $\boldsymbol{v}$ so zu bestimmen, dass die Fehlerfunktion

$$\zeta_{LSQ}(\boldsymbol{v}) = \frac{1}{2} \sum_{i=1}^{m} (f(s_{t_i}) - p(s_{t_i}))^2 \tag{2.41}$$

minimiert wird. Mit der aus den m letzten um ξ_0 erweiterten Zuständen $s'_{t_1}, s'_{t_2}, \ldots, s'_{t_m} = s'$, zu denen der betrachtete Klassifizierer passte, gebildeten Matrix

$$X = \left[s'_{t_1}, s'_{t_2}, \ldots, s'_{t_m}\right]^T$$

und dem Vektor

$$\boldsymbol{f} = [f_{t_1}, f_{t_2}, \ldots, f_{t_m}]^T$$

der zugehörigen Werte $f_{t_1}, f_{t_2}, \ldots, f_{t_m} = f(s)$ der zu approximierenden Funktion , kann der die Fehlerfunktion (2.41) minimierende Vektor $\hat{\boldsymbol{v}}$ unter Zuhilfenahme der Pseudoinversen X^+ von X oder der Singulärwertzerlegung von $X^T X$ als Lösung der Gleichung

$$X^T X \boldsymbol{v} = X^T \boldsymbol{f} \tag{2.42}$$

berechnet werden[28]. Im Prinzip können bei der Minimierung von (2.41) alle Zustände berücksichtigt werden, zu denen der Klassifizierer jemals passte. Angesichts der langen Trainingszeiten des XCS können dies jedoch sehr viele Zustände sein, sodass sowohl der Speicherbedarf – jeder Klassifizierer muss die Zustände zu denen er passte sowie die zugehörigen Funktionswerte speichern – als auch der Rechenaufwand immens würden. Eine Berücksichtigung nur weniger Zustände birgt andererseits das Risiko, dass diese nicht alle Zustände, zu denen der Klassifizierer passt, angemessen repräsentieren und eine direkte Übernahme der Lösung von (2.42) dessen Gewichte unangemessen stark ändert. Indem vermöge

$$\boldsymbol{v} \leftarrow (1 - \beta_o)\boldsymbol{v} + \beta_o \hat{\boldsymbol{v}} \tag{2.43}$$

[28] Auf den mathematischen Hintergrund wird in Anhang A kurz eingegangen, er findet sich unter dem Stichwort ‚Lineares Ausgleichsproblem' aber auch in den meisten Lehrbüchern der numerischen Mathematik, etwa in [Stoer 1994; Hanke-Bourgeois 2002].

die neuen Gewichte des Klassifizierers als gewichtetes Mittel der bisherigen und der als Lösung $\hat{\boldsymbol{v}}$ von (2.42) berechneten ermittelt werden, wird dies verhindert.

Obwohl primär zum Zwecke der Funktionsapproximation eingeführt, kann die Verwendung berechneter Klassifizierervorhersagen durchaus auch sinnvoll sein, wenn das XCS zum Lernen von Aktionen eingesetzt wird – immer dann, wenn die Werte der zur Verfügung stehenden Aktionen durch stückweise konstante Funktionen auf dem Zustandsraum nicht genau genug abgeschätzt werden können, um eine gute Aktionswahl zu ermöglichen [Lanzi u. a. 2005c, d]. In diesem Fall merkt sich ein Klassifizierer einen Zustand, zu dem er passte, nur dann, wenn er in dem nach dessen Präsentation gebildeten Action-Set enthalten ist; an die Stelle der Werte $f_{t_1}, f_{t_2}, \ldots, f_{t_m}$ der zu approximierenden Funktion treten die gemäß (2.25) respektive (2.34) ermittelten zugehörigen Returnabschätzungen $\mathcal{P}_{t_1}, \mathcal{P}_{t_2}, \ldots, \mathcal{P}_{t_m}$.

2.5 Zusammenfassung

Die biologische Forschung hat seit dem 19. Jahrhundert mehr und mehr Hinweise darauf geliefert, dass die heute bekannten Lebensformen in erdgeschichtlichen Zeiträumen durch einen Evolutionsprozess hervorgebracht wurden, dessen grundlegender Mechanismus ein Wechselspiel von Selektion und Variation ist. Aus einer Übertragung dieses Mechanismus in einen technisch-informatischen Kontext entwickelte sich das Forschungsgebiet evolutionärer Algorithmen, dessen bekanntesten Ansatz die zum Lösen von Optimierungsproblemen verwendeten Genetischen Algorithmen darstellen.
Beim Reinforcement-Learning lernt ein Agent, sich in einer interaktiven Lernumgebung angemessen zu verhalten. Das Lernen erfolgt dabei nach dem Prinzip von Versuch und Irrtum: der Agent probiert verschiedene Aktionen aus, beobachtet die Reaktionen der Lernumgebung und passt sein Verhalten basierend auf diesen an.
Lernende Klassifizierende Systeme verbinden Genetische Algorithmen mit einem Reinforcement-Learning-Ansatz. Sie evolvieren Regeln der Form *Bedingung : Aktion*, die als Verhaltensanweisungen für einen Agenten in einem Reinforcement-Learning-Szenario interpretiert werden können: Wenn der Zustand der Lernumgebung die *Bedingung* erfüllt, führe die *Aktion* aus. Die Fitness dieser Regeln, die darüber entscheidet, ob eine Regel eine „evolutionäre Sackgasse“ darstellt oder die Chance erhält, sich unter dem Genetischen Algorithmus weiter zu entwickeln, wird ausgehend von den Reaktionen der Lernumgebung – Belohnungen oder Bestrafungen – auf ihre Anwendung bestimmt.
Eines der ersten Lernenden Klassifizierenden Systeme war Hollands CS-1, das jedoch nicht die gewünschte Leistung erbrachte. Zudem war es so komplex, das praktisch nicht zu analysieren war, wodurch genau dies verhindert wurde. Mit dem ZCS führte Wilson eine radikal vereinfachte Variante des CS-1 ein, die das Verständnis Lernender Klassifizierender Systeme verbessern sollte, die aber darüber hinaus auch gute, in einigen Testproblemen sogar optimale, Lernergebnisse erzielte.
Wie die meisten frühen Lernenden Klassifizierenden Systeme basiert die Fitness der Klassifizierer sowohl beim CS-1 als auch beim ZCS allein auf der Höhe ihrer Belohnungsvorhersagen. Das wiederum von Wilson entwickelte XCS hingegen berechnet die Fitness ausgehend von der Genauigkeit dieser Vorhersagen. Mit diesem Ansatz wurde eine deutliche Leistungssteigerung erreicht, sodass das XCS mit seinen diversen Varianten heute das wohl am weitesten verbreitete Lernende Klassifizierende System ist.

Kapitel 3

Selbstorganisierende Karten

Die Frage nach der Funktionsweise des Gehirns beschäftigt Menschen seit Hunderten von Jahren und so ist es nur natürlich, dass auch Computer eingesetzt werden, um den Antworten näher zu kommen. Bereits seit den 1940er Jahren wurde eine Vielzahl *Künstlicher Neuronaler Netze* modelliert, die Vorgänge in Netzwerken biologischer Neuronen imitieren. Die Hoffnung, durch deren Untersuchung und computergestützte Simulation Erkenntnisse über die Prinzipien der Informationsverarbeitung im menschlichen Gehirn zu erhalten, hat sich (bisher) jedoch nur ansatzweise erfüllt. Mit zunehmender Leistungsfähigkeit der Computerhardware gewann jedoch auch der Aspekt der technischen Anwendbarkeit derartiger Modelle an Bedeutung; Künstliche Neuronale Netze wurden zur Lösung einer Vielzahl von Problemen erfolgreich eingesetzt und stellen heute etablierte Verfahren der Informatik dar. Einen Überblick über verschiedene Typen Künstlicher Neuronaler Netze geben etwa [Brause 1995] und [Patterson 1996].
In diesem Kapitel wird eine Klasse solcher Künstlichen Neuronalen Netze beschrieben, die *Selbstorganisierenden Karten*, die selbstständig eine Abbildung von Eingabesignalen auf Neuronen finden und insbesondere bei der Analyse und Visualisierung hochdimensionaler Daten zum Einsatz kommen.
Der Abschnitt 3.1 widmet sich dem biologischen Hintergrund Selbstorganisierender Karten und deren allgemeinem Aufbau und schafft damit die Grundlage für den Abschnitt 3.2, in dem drei Varianten Selbstorganisierender Karten detaillierter betrachtet werden. Anschließend werden in Abschnitt 3.3 zwei Möglichkeiten angesprochen, Selbstorganisierende Karten zum Lernen von Aktionen zu befähigen: Motorische Karten und die Kombination mit einem Reinforcement-Learning-Verfahren. In Abschnitt 3.4 wird das Kapitel zusammengefasst.

3.1 Grundlagen

Selbstorganisierende Karten finden selbstständig (unüberwacht) eine Abbildung mehrdimensionaler Eingabedaten auf eine Struktur künstlicher Neuronen. Die dabei verwendeten Lernmechanismen sind von in der Großhirnrinde ablaufenden Prozessen inspiriert,

die im Folgenden kurz skizziert werden[1]. Ausgehend von dieser Basis, wird als „Vorstufe“ Selbstorganisierender Karten ein Neuronen-Konkurrenzmodell motiviert, dessen Vereinfachung und Abstraktion auf den grundlegenden Aufbau Selbstorganisierender Karten führt.

3.1.1 Neurophysiologischer Hintergrund

Die höheren Gehirnfunktionen, etwa die Verarbeitung sensorischer Reize, das assoziative Denken, die Erkennung und Erzeugung von Sprache und die Bewegungskontrolle, sind im *Neokortex* lokalisiert, der etwa 2-3 Millimeter dicken und beim Menschen ungefähr 0.2 Quadratmeter großen Großhirnrinde.
Die Grundeinheiten der Informationsverarbeitung des Gehirns sind die Nervenzellen (*Neuronen*); der im Neokortex häufigste Neuronentyp ist die Pyramidenzelle, deren Aufbau in Abbildung 3.1 schematisch dargestellt ist: Das Neuron kann grob in den *Dendritenbaum*, den *Zellkörper* (*Soma*) und das *Axon* untergliedert werden. Letzteres ist vielfach verzweigt und liegt mit den Enden seiner Ausläufer dicht an Dendriten oder Soma von bis zu 1000 anderen Neuronen an. Diese Kontaktstellen werden als *Synapsen* bezeichnet und dienen der Signalübertragung zwischen den Neuronen.

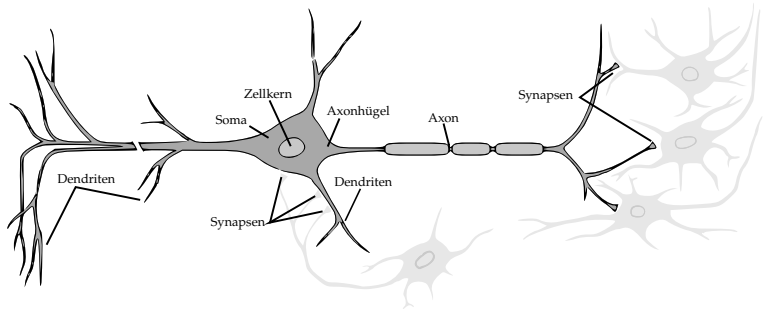


Abbildung 3.1: Schematische Darstellung eines Neurons (Pyramidenzelle).
(Nach [Kandel u. a. 1996], verändert.)

Die Erregungsleitung erfolgt innerhalb der Neuronen durch elektrische Impulse; erreicht ein solcher eine Synapse, löst er dort die Ausschüttung eines Überträgerstoffes (*Neurotransmitters*) aus, der von Rezeptoren des „Empfängerneurons“ aufgenommen wird und dessen elektrisches Potential erhöht (*exzitatorische* Synapse) oder verringert (*inhibitorische* Synapse). Die an den Synapsen entstehenden Potentiale werden im Soma gewissermaßen aufsummiert, besondere Bedeutung kommt dabei dem *Axonhügel* zu: Sobald das aufsummierte Potential einen Schwellenwert überschreitet, wird dort ein elektrischer Impuls (*Aktionspotential*) erzeugt und über das Axon an alle über Synapsen nachgeschalteten Neuronen weitergeleitet.
Aus dieser einfachen[2] Reizverarbeitung einzelner Nervenzellen ergibt sich durch die große Zahl von Neuronen (etwa 100.000 pro Quadratmillimeter des Neokortex, etwa 100 Milliarden im menschlichen Gehirn insgesamt) und deren hohen Vernetzungsgrad die Fähigkeit des Gehirns zur komplexen Informationsverarbeitung.

[1] Für eine genauere Darstellung muss an dieser Stelle auf die biologische Fachliteratur, etwa [Campbell 1997; Kandel u. a. 1996], verwiesen werden.

[2] Einfach sind nur die Prinzipien neuronaler Reizverarbeitung; die beteiligten und in der obigen Darstellung übergangenen biochemischen und biophysikalischen Prozesse sind weitaus komplizierter.

In vielen Bereichen des Neokortex lässt sich erkennen, dass benachbarte Neuronen zu funktionalen Einheiten verschaltet sind. Dieses Prinzip setzt sich auf höheren Organisationsebenen fort, sodass verschiedene Funktionen des Neokortex in räumlich getrennten *Rindenfeldern* zusammengefasst sind (Abbildung 3.2(a)). Diese Felder sind ihrerseits untereinander sowie mit Gehirn- und Nervenstrukturen außerhalb des Neokortex verbunden. Viele dieser Verbindungen zeichnen sich dadurch aus, dass benachbarte Neuronen auf der „Ausgangsseite" mit benachbarten Neuronen auf der „Zielseite" verbunden sind. So werden etwa Reize an benachbarten Tastrezeptoren in der Haut benachbarten Neuronen im *somatosensorischen Kortex* zugeführt, der somit eine „Karte" der Körperoberfläche enthält [Penfield u. Boldrey 1937; Kaas u. a. 1979] (vgl. Abbildung 3.2(b)).

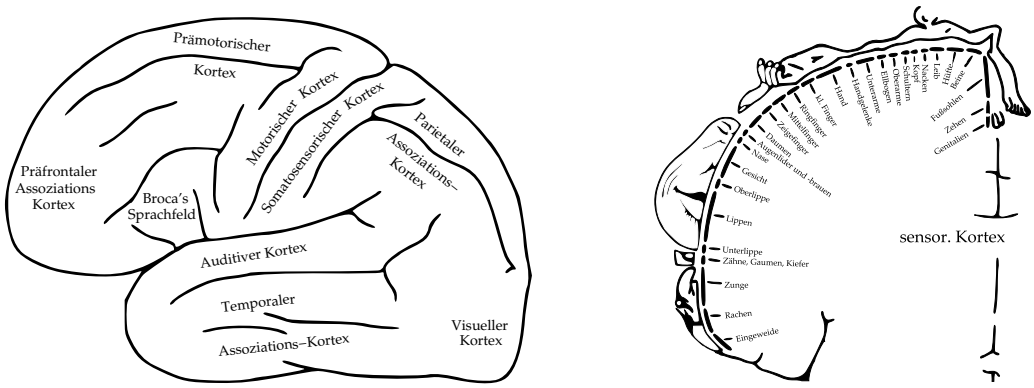


Abbildung 3.2: Rindenfelder des menschlichen Neokortex: **(a)** Rindenfelder in der linken Hemisphäre des Großhirns (nach [Ritter u. a. 1990]) **(b)** Somatosensorischer Kortex mit Zuordnung der assoziierten Körperteile (nach [Penfield u. Rasmussen 1950] (Bildquelle: [Homunculus])), verändert.).

Die Rindenfelder des Neokortex stellen somit „*systeminterne Repräsentationen äußerer Gegebenheiten, die die Nachbarschaftsstruktur dieser externen Gegebenheiten respektieren*" [Polani 1996] dar. Diese spezielle Anordnung der Neuronen spiegelt sich in deren „Verschaltung" wider, die dem Schema der *lateralen Inhibition* oder *Umfeldhemmung* folgt: Räumlich benachbarte Neuronen sind durch exzitatorische, weiter entfernt liegende Neuronen durch inhibitorische Synapsen verbunden. Diese Verschaltungen sind keineswegs vollständig genetisch festgelegt, sondern werden zumindest teilweise erst unter Mitwirkung sensorischer Reize gelernt und können sich darüber hinaus in Abhängigkeit von der Stimulation zeitlebens verändern.
Derartige Lernvorgänge führt die Neurophysiologie auf Veränderungen in der Vernetzung der Neuronen zurück; Donald Hebb postulierte 1949 einen Mechanismus, der die Stärke der Reizweiterleitung einer Synapse – die *Synapsenstärke* – abhängig von der korrelierten Aktivität der Neuronen „vor" und „hinter" dieser Synapse anpasst [Hebb 1949]:

> *When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B is increased.*

Dass korrelierte post- und präsynaptische Aktivität eine Rolle bei der Anpassung von Synapsenstärken spielt, also an Lernvorgängen des Gehirns beteiligt ist, konnte experimentell nachgewiesen werden [Kelso u. a. 1986], jedoch ist umstritten, ob diese Anpassungen in der von Hebb angenommenen Art erfolgen [Seung 2000].

3.1.2 Ein Neuronen-Konkurrenzmodell

Selbstorganisierende Karten gehen auf die von Teuvo Kohonen 1982 vorgestellte Kohonenkarte (Abschnitt 3.2) zurück. Diese entstand durch Vereinfachung des Reizverarbeitungsprozesses des *Neuronen-Konkurrenzmodells* [Kohonen 1982][3]– eines Künstlichen Neuronalen Netzes, das unüberwacht eine Abbildung mehrdimensionaler Eingabedaten auf eine Struktur künstlicher Neuronen findet.
Dabei stellt ein *(künstliches) Neuron* eine Recheneinheit dar, die über einen Eingang und einen Ausgang verfügt[4]. Dem Eingang werden über mit Gewichten (Synapsenstärken) versehene Verbindungen Signale zugeleitet. In vorgegebenen Zeitschritten wird die *Aktivierung* des Neurons als mit den Synapsenstärken gewichtete Summe der Eingabesignale berechnet und daraus über eine *Ausgabefunktion* der am Ausgang des Neurons ausgegebene Wert. Die Ausgabefunktion wird meist sigmoid gewählt: Die Ausgabe wächst mit steigender Aktivierung zunächst schwach, in der Umgebung eines Schwellenwertes dann stark und nähert sich schließlich asymptotisch einem Maximalwert.
Indem in geschickter Weise Aus- und Eingänge mehrerer Neuronen verbunden werden, entstehen Künstliche Neuronale Netze[5]. Exzitatorische und inhibitorische Synapsen werden dabei durch positive beziehungsweise negative Synapsenstärken modelliert. Einige Neuronen eines solchen Netzes dienen als *Eingabeneuronen*, über ihre nicht mit anderen Neuronen verbundenen Eingänge werden dem Netz Daten (*Eingaben, Reize*) zur Verarbeitung zugeführt. Analog werden die Resultate dieser Verarbeitung über die Ausgänge der *Ausgabeneuronen* abgeführt.

Kohonens Konkurrenzmodell, dessen Struktur in Abbildung 3.3 dargestellt ist, besteht aus zwei Schichten von Neuronen: Einer nach dem Prinzip der lateralen Inhibition verschalteten zweidimensionalen Konkurrenzschicht $\mathcal{K}$ ist eine Schicht $\mathcal{I}$ von Eingabeneuronen vorgeschaltet.

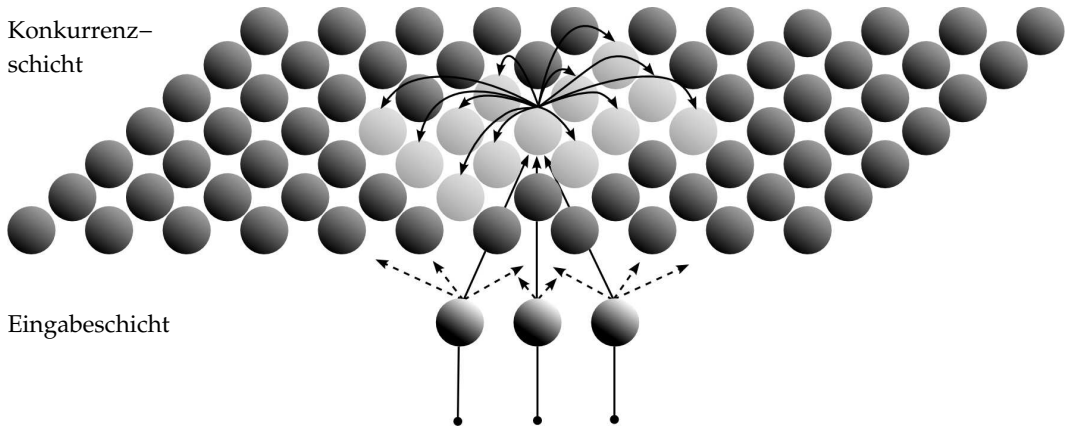


Abbildung 3.3: Zweischicht-Neuronen-Konkurrenzmodell. Es sind nur die exzitatorischen Synapsen eines einzelnen Neurons der Konkurrenzschicht dargestellt. Ferner sind die Verbindungen zwischen der Eingabe- und der Konkurrenzschicht nur angedeutet. (Nach [Polani 1996], verändert.)

[3] Kohonens Beschreibung des Konkurrenzmodells ist recht knapp, eine genauere Darstellung und Untersuchung einer Neuimplementierung findet sich in [Miikkulainen 1991].

[4] Dies ist genau genommen nur ein Spezialfall. Eine allgemeine, formale Definition wird zum Beispiel in [Polani u. Uthmann 1992] gegeben

[5] Eine kurze Übersicht über verschiedene Typen Künstlicher Neuronaler Netze findet sich zum Beispiel in [Polani u. Uthmann 1992; Ritter u. a. 1990]

Die an die Eingabeneuronen angelegten äußeren Eingaben $\xi_k \in \mathbb{R}$ bilden den *Eingabe-* oder *Reizvektor* $s = (\xi_1, \ldots, \xi_n)^T$, wobei n die Zahl der Neuronen der Eingabeschicht ist. Jedes Neuron e_k der Eingabeschicht ist durch eine Synapse der Stärke $w_{i,k} \in \mathbb{R}$ mit jedem Neuron c_i der Konkurrenzschicht verbunden. Der Vektor $\boldsymbol{w}_i = (w_{i,1}, \ldots, w_{i,n})^T$ wird als *Gewichtsvektor* des Konkurrenzneurons c_i bezeichnet. Gewichts- und Reizvektoren werden als auf die Länge 1 normiert vorausgesetzt. Innerhalb der Konkurrenzschicht sind alle Neuronen c_i,c_j miteinander durch Synapsen der Stärken $\gamma_{i,j} \in \mathbb{R}$ verbunden; bezüglich einer geeigneten, auf der Neuronenschicht definierten Metrik benachbarte Neuronen exzitatorisch, alle anderen inhibitorisch. Die Eingabeneuronen verwenden als Ausgabefunktion die Identität, geben die angelegten Eingaben also unverändert an die Neuronen der Konkurrenzschicht weiter. Die Ausgabefunktion σ der Konkurrenzneuronen ist sigmoid.
Nach Anlegen eines Reizvektors s werden die Ausgaben der Konkurrenzneuronen iterativ berechnet, bis das Netz einen stabilen Zustand erreicht; die Ausgabe $o_i(t)$ des Konkurrenzneurons c_i im t-ten Berechnungsschritt ergibt sich als[6]:

$$o_i(t) = \sigma\left(\sum_{k=1}^{n} w_{i,k} \cdot \xi_k + \sum_{c_j \in \mathcal{K}} \gamma_{i,j} \cdot o_j\,(t-1)\right) \tag{3.1}$$

Die zweite Summe modelliert den Effekt der lateralen Inhibition, die erste Summe die Reaktion auf die externen Eingaben. Diese entspricht gerade dem Skalarpodukt von $\boldsymbol{w}_i$ und s und ist daher aufgrund der Normierung dieser Vektoren um so größer, je kleiner der von ihnen gebildete Winkel ist.
Nach Anlegen eines Reizvektors s und Erreichen eines stabilen Ausgabemusters werden die Synapsenstärken $\boldsymbol{w}_i$ proportional den Ausgaben o_i der Konkurrenzneuronen und den Ausgaben s der Eingabeneuronen gemäß einer durch das Hebb'sche Postulat motivierten Lernregel angepasst:

$$\boldsymbol{w}_i \leftarrow \frac{\boldsymbol{w}_i + \eta \cdot o_i \cdot s}{\|\boldsymbol{w}_i + \eta \cdot o_i \cdot s\|} \tag{3.2}$$

Aufgrund der Normierung bewirkt dies eine Drehung der Gewichtsvektoren in Richtung des Reizvektors, die umso stärker ausfällt, je größer die Ausgabe o_i ist. Sind die Lernrate $0 \leq \eta \leq 1$ und die Synapsenstärken $\gamma_{i,j}$ geeignet gewählt, führt diese Lernregel zu einer Selbstorganisation der Konkurrenzneuronen in dem Sinne, dass, ausgehend von zufällig initialisierten Gewichtsvektoren, nach Präsentation von ausreichend vielen Reizvektoren (etwa 5000 - 10000) in der Konkurrenzschicht benachbarte Neuronen auch benachbarte Gewichtsvektoren[7] haben.
Das Konkurrenzmodell weist dann die folgende Ausgabecharakteristik auf: Wird ein Reizvektor s angelegt, so führt die laterale Inhibition gemäß (3.1) nach einer kurzen Einschwingphase zur Ausbildung eines Clusters von Neuronen mit hoher Ausgabe um dasjenige Neuron $c_{w(s)}$, dessen Gewichtsvektor $\boldsymbol{w}_{w(s)}$ den kleinsten Winkel mit dem Reizvektor s bildet. Aufgrund der Normierung minimiert dieses Neuron auch den euklidischen Abstand zum Reizvektor. Die Ausgabe der Neuronen nimmt mit zunehmender Entfernung vom Reiz ab (und damit nach (3.2) auch die Stärke ihrer Anpassung). Die durch dieses Ausgabeverhalten implizierte Zuordnung $s \mapsto c_{w(s)}$ stellt somit eine unüberwacht gelernte Abbildung der Reizvektoren auf die Neuronen der Konkurrenzschicht dar.

[6] Im ersten Berechnungsschritt wird die zweite Summe gleich 0 gesetzt [Miikkulainen 1991].

[7] Die Nachbarschaft von Gewichtsvektoren kann durch die von diesen induzierte Voronoizerlegung (siehe 3.1.4) des $\mathbb{R}^n$ definiert werden: Zwei Gewichtsvektoren sind benachbart, wenn der Schnitt ihrer Voronoizellen nicht leer ist.

3.1.3 Übergang zu Selbstorganisierenden Karten

Um zu den noch stärker vom biologischen Vorbild abstrahierenden Selbstorganisierenden Karten zu kommen, werden einige Vereinfachungen vorgenommen:

- Da die Neuronen der Eingabeschicht nichts weiter tun, als die angelegten Eingaben an die Konkurrenzschicht „durchzureichen", wird die Eingabeschicht weggelassen. Übrig bleibt eine der Konkurrenzschicht entsprechende Menge von Neuronen.
- An die Stelle der Aktivierung eines Neuronenclusters durch iterative Berechnung von (3.1) tritt eine Abbildung, die jedem Reizvektor $s \in \mathbb{R}^n$ in einem Schritt dasjenige Neuron zuordnet, dessen Gewichtsvektor den geringsten euklidischen Abstand zum Reizvektor hat.
- Die durch Lernregel (3.2) gegebene Drehung wird ersetzt durch eine Verschiebung der Gewichtsvektoren in Richtung auf den Reizvektor. Dies macht – in Verbindung mit dem vorgenannten Punkt – die Normierung der Eingabe- und Gewichtsvektoren überflüssig.
- Es erfolgt keine explizite Modellierung lateraler Inhibition; lediglich der Effekt der aus der lateralen Inhibition resultierenden Ausgabecharakteristik wird imitiert: Ein Neuron wird um so weniger angepasst, je weiter entfernt es vom Reizvektor respektive dem Gewinnerneuron ist.
- Viele Typen Selbstorganisierender Karten verbinden Neuronen durch ungerichtete Kanten und versehen somit die Menge der Neuronen mit einer Nachbarschaftsstruktur; diese gestattet es, auf der Menge der Neuronen eine Metrik zu definieren, die zur Bestimmung der Stärke der Anpassung benutzt werden kann. Alternativ kann auch eine Metrik des Eingaberaums zugrunde gelegt werden.

3.1.4 Grundlegender Aufbau Selbstorganisierender Karten

Selbstorganisierende Karten bilden eine Klasse Künstlicher Neuronaler Netze, die in der Lage sind, unüberwacht, das heißt ohne Vorgabe von Sollausgaben, eine Abbildung von Eingabesignalen (Elementen eines *Eingaberaums*) auf eine Menge von Neuronen zu finden. In dieser Arbeit werden die Eingabesignale stets reellwertige Eingabevektoren $s \in \mathbb{R}^n$ sein. Daher wird in den folgenden Ausführungen nur der Fall betrachtet, dass es sich beim Eingaberaum um den $\mathbb{R}^n$ handelt; eine Darstellung unter Berücksichtigung der Möglichkeit allgemeinerer Eingaberäume findet sich zum Beispiel in [Polani 1996].
Selbstorganisierende Karten (kurz: *Karten, Netze*) sind Strukturen, die aus einer Menge

$$\mathcal{N} = \{c_1, c_2, \ldots, c_N\}$$

von Neuronen und einer (möglicherweise leeren) Menge $\mathcal{E}$ ungerichteter *Kanten* zwischen diesen Neuronen bestehen. Jedem Neuron c_i ist eine als *Gewichtsvektor* bezeichnete Position $\boldsymbol{w}_i \in \mathbb{R}^n$ im Eingaberaum zugeordnet. Da ein Neuron im Wesentlichen durch seinen Gewichtsvektor beschrieben ist, wird im Folgenden nicht streng zwischen einem Neuron und seiner Position unterschieden und oft auch dann von Neuronen gesprochen, wenn eigentlich Positionen gemeint sind. Sind zwei Neuronen c_i und c_j durch eine Kante verbunden, so heißen sie *benachbart*; die sie verbindende Kante wird mit $e_{i,j} = e_{j,i}$ bezeichnet. Ist $N_i \in \mathbb{N}$ die Anzahl der Nachbarn eines Neurons c_i, so werden diese Nachbarn folgendermaßen indiziert: $c_{i,j}, j = 1, 2, \ldots, N_i$. Analog wird der Gewichtsvektor des j-ten Nachbarn von c_i mit $\boldsymbol{w}_{i,j}$ bezeichnet.

Selbstorganisierende Karten realisieren eine Abbildung vom Eingaberaum auf die Menge der Neuronen, ordnen also jedem Eingabevektor s ein Neuron $c_{w(s)}$ zu:

$$\Phi : \mathbb{R}^n \to \mathcal{N},\ s \mapsto c_{w(s)} \tag{3.3}$$

Das *Gewinnerneuron* $c_{w(s)}$ ist dabei dasjenige Neuron, dessen Gewichtsvektor dem Eingabereiz bezüglich einer Metrik des Eingaberaums am nächsten ist. Hier wird diese Metrik stets der euklidische Abstand $\|\cdot,\cdot\|$ sein:

$$c_{w(s)} := \arg\min_{c_i \in \mathcal{N}} \|\boldsymbol{w}_i - s\| \tag{3.4}$$

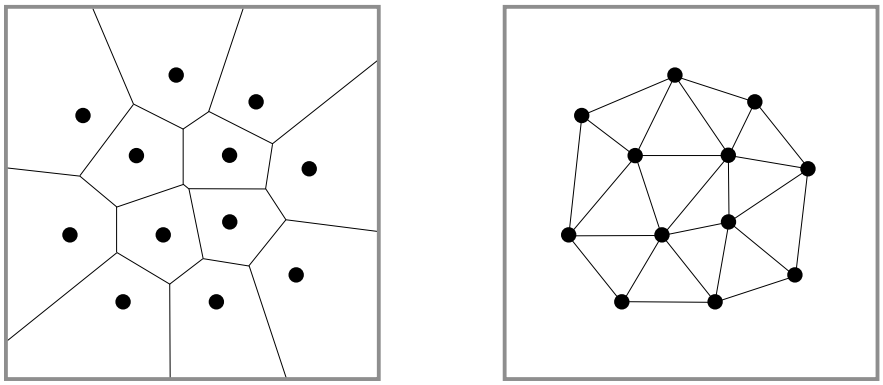
Diese Definition ist streng genommen nicht eindeutig, da der Abstand zum Eingabevektor von mehreren Neuronen gleichzeitig minimiert werden kann. Für die Theorie Selbstorganisierender Karten ist dies unwesentlich, da die Menge der Eingabevektoren für die (3.4) mehrdeutig ist, eine lesbesguesche Nullmenge ist[8] und demzufolge die Auftrittswahrscheinlichkeit Null hat. Bei der Simulation im Computer gilt dies aufgrund der zwangsläufig erfolgenden Diskretisierung nicht mehr; das Problem ist aber leicht zu umgehen – beispielsweise indem stets das zuerst gefundene der den Abstand zum Eingabevektor minimierenden Neuronen zum Gewinnerneuron bestimmt wird. Im Folgenden wird daher stillschweigend die Eindeutigkeit der Zuordnung (3.4) vorausgesetzt.
Statt $c_{w(s)}$ wird im Folgenden nur c_w geschrieben, wenn klar ist, dass das zum jeweils aktuellen Eingabevektor gehörende Gewinnerneuron gemeint ist.
Als Hilfsmittel zur Analyse und insbesondere zur Darstellung Selbstorganisierender Karten wird oft die durch (3.3) induzierte *Voronoizerlegung* des Eingaberaumes in *Voronoizellen* herangezogen. Die Voronoizelle eines Neurons c_i ist die Menge aller Punkte $\boldsymbol{x} \in \mathbb{R}^n$ für die $\boldsymbol{w}_i$ ein Gewichtsvektor mit dem geringsten Abstand zu $\boldsymbol{x}$ ist:

$$V_i := \{\boldsymbol{x} \in \mathbb{R}^n \mid \|\boldsymbol{x} - \boldsymbol{w}_i\| \leq \|\boldsymbol{x} - \boldsymbol{w}_k\|, k = 1, \ldots, N\} \tag{3.5}$$

Der Gewichtsvektor $\boldsymbol{w}_i$ wird auch als *Zentrum* von V_i bezeichnet. Ausgehend von der Voronoizerlegung wird die Nachbarschaft von Gewichtsvektoren im Eingaberaum definiert: Zwei Gewichtsvektoren $\boldsymbol{w}_i, \boldsymbol{w}_j \in \mathbb{R}^n$ heißen *benachbart*, wenn der Schnitt ihrer Voronoizellen nicht leer ist: $V_i \cap V_j \neq \emptyset$. Abbildung 3.4(a) illustriert die Voronoizerlegung eines zweidimensionalen Eingaberaumes. Die Zentren der Voronoizellen werden durch Punkte markiert, ihre Grenzen durch Linien. Der Voronoizerlegung dual ist die *Delaunay-Triangulierung* der Gewichtsvektoren. Diese entsteht, indem benachbarte Gewichtsvektoren miteinander verbunden werden, wie in Abbildung 3.4(b) gezeigt. Werden die Kanten aus $\mathcal{E}$ als Verbindungen zwischen Gewichtsvektoren aufgefasst, so bilden sie bei einigen Typen Selbstorganisierender Karten (näherungsweise) einen Subgraphen der Delaunay-Triangulierung.

A priori sind den Neuronen einer Selbstorganisierenden Karte zufällig Gewichtsvektoren zugeordnet, die von der Karte repräsentierte Abbildung des Eingaberaum auf die Neuronenmenge ist also gewissermaßen beliebig. Dies ändert sich im Verlauf eines Trainingsprozesses, der die Gewichtsvektoren in Abhängigkeit von den angelegten Eingabevektoren adaptiert. Letztere stammen entweder aus einer vorgegebenen Menge $\{s_1, s_2, \ldots, s_T\}$ von Eingabevektoren oder werden entsprechend einer auf dem Eingaberaum gegebenen

[8] Die bezüglich der euklidischen Metrik gleich weit von zwei Gewichtsvektoren entfernten Punkte des $\mathbb{R}^n$ liegen in einer Hyperebene, also einer Nullmenge. Die Menge der Eingabevektoren, für die das Gewinnerneuron durch (3.4) nicht eindeutig bestimmt ist, ist – aufgrund der endlichen Anzahl von Neuronen – die Vereinigung endlich vieler solcher Hyperebenen, also wiederum eine Nullmenge.



(a) **(b)**

Abbildung 3.4: (a) Voronoizerlegung eines zweidimensionalen Eingaberaums. **(b)** Die zu (a) duale Delaunay-Triangulierung.

Häufigkeitsverteilung erzeugt, bilden also eine Folge unabhängig identisch verteilter Zufallsvariablen s_t.

Die Adaption der Gewichtsvektoren geschieht in Form einer Verschiebung in Richtung auf den Eingabevektor und wird durch eine Lernregel der folgenden Form beschrieben:

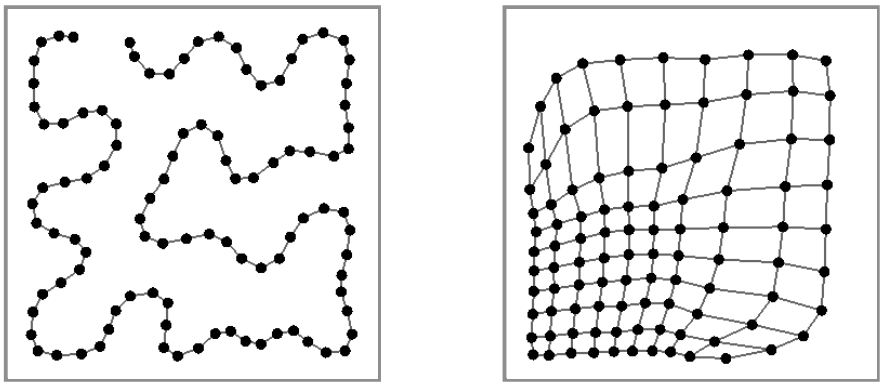
$$\boldsymbol{w}_i \leftarrow \boldsymbol{w}_i + \eta_i(t) \cdot h_i(d,t) \cdot (s - \boldsymbol{w}_i) \tag{3.6}$$

Die *Lernrate* $0 \leq \eta_i(t) \leq 1$ legt in Abhängigkeit von der Anzahl t der bereits vergangenen Trainingsschritte fest, wie stark der Gewichtsvektor des Neurons c_i angepasst wird, falls c_i das Gewinnerneuron ist.
Die *Nachbarschaftsfunktion* $h_i(d,t)$ bestimmt, wie stark diese Anpassung relativ zu $\eta_i(t)$ ausfällt, wenn c_i nicht das Gewinnerneuron ist. Sie ist im Allgemeinen abhängig von der Zahl t der vergangenen Trainingsschritte und einem Wert $d \in \mathbb{R}^+$, der ein Maß für die Entfernung des Neurons c_i zum Eingabevektor darstellt. Dies kann beispielsweise der euklidische Abstand $\|\boldsymbol{w}_i - s\|$ des Gewichtsvektors $\boldsymbol{w}_i$ zum Eingabevektor s sein. Oft wird d aber auch ausgehend von einer auf der Neuronenstruktur definierten Metrik $d^{\mathcal{N}}(\cdot,\cdot)$ bestimmt. Die Nachbarschaftsfunktion erreicht ihr Maximum für $d = 0$ und nimmt mit wachsendem d monoton ab.
Durch die Indizierung $\cdot_i$ der Lernraten und Nachbarschaftsfunktionen wird die Möglichkeit angedeutet, dass jedes Neuron eine eigene Lernrate und eine eigene Nachbarschaftsfunktion haben kann, oft wird aber auch eine einheitliche Lernrate und/oder eine einheitliche Nachbarschaftsfunktion benutzt. Ferner weisen Lernrate und Nachbarschaftsfunktion auch nicht in allen Fällen eine Zeitabhängigkeit auf.

Der beschriebene Traingsprozess hat eine Selbstorganisation zur Folge: Die a-priori ungeordnete Karte wird zunehmend geordneter in dem Sinne, dass die Karte sich so verändert,

(i) dass die Gewichtsvektoren von in der Neuronenstruktur benachbarten Neuronen im Eingaberaum benachbart sind

(ii) und dass die Verteilung der Neuronen die Häufigkeitsverteilung der Eingabevektoren widerspiegelt: In Bereichen des Eingaberaumes, aus denen viele Eingabevektoren stammen, liegen auch viele Neuronen (Abbildung 3.5(b)).



(a) **(b)**

Abbildung 3.5: Ausbreitung zweier Kohonenkarten (siehe 3.2.1) **(a)** Mit auf dem Einheitsquadrat $[0,1]^2$ gleichverteilten Eingabevektoren trainierte Neuronenkette. **(b)** Zweidimensionales Neuronengitter, trainiert mit Eingabevektoren s aus dem Einheitsquadrat ($\mathrm{P}(s \in [0,0.5]^2) = 2 \cdot \mathrm{P}(s \in [0,1]^2 \setminus [0,0.5]^2)$).

Neben der durch (3.6) beschriebenen Adaption der Gewichtsvektoren führen einige Typen Selbstorganisierender Karten während des Trainings eine Anpassung der Struktur der Karte durch, indem Neuronen und Kanten hinzugefügt oder gelöscht werden. Die dabei zur Anwendung kommenden Verfahren sind jedoch zu unterschiedlich, um hier in allgemeiner Form darstellbar zu sein; in Abschnitt 3.2 werden jedoch zwei derartige Kartentypen beschrieben: Das Neurale Gas und das Wachsende Neurale Gas.
Mit diesen und ähnlichen Varianten Selbstorganisierender Karten kann das sogenannte *Topologie-Lernen* realisiert werden. Darunter wird die Ausbildung einer Nachbarschaftsstruktur auf der Neuronenmenge verstanden, die die Topologie der Verteilung der Eingabevektoren widerspiegelt, bei der also

(iii) ähnliche Eingabevektoren auf benachbarte Neuronen abgebildet werden.

Diese Eigenschaft kann von einer fest vorgegebenen Neuronenstruktur im Allgemeinen nicht erfüllt werden. Dies ist deutlich zu erkennen, wenn eine zweidimensionale Eingabeverteilung auf eine Kette von Neuronen trainiert wird, wie in Abbildung 3.5(a) dargestellt. Die resultierende Karte hat die Eigenschaft (i), nicht jedoch die Eigenschaft (iii).
Eine Selbstorganisierende Karte, die beide Eigenschaften (i) und (iii) aufweist, wird als *topologieerhaltend*[9] bezeichnet.

3.2 Typen Selbstorganisierender Karten

Die in 3.1.4 beschriebene grundlegende Funktionsweise lässt Spielraum für eine große Bandbreite von Variationen. Dementsprechend existiert eine große Vielfalt von Algorithmen, die anhand der folgenden Kriterien klassifiziert werden können:

Art des kompetitiven Lernen

Einige Algorithmen, beispielsweise der k-means-Algorithmus [MacQueen 1967], passen nur das Gewinnerneuron an den Eingabevektor an. Dies wird als *hartes kompetitives*

[9] Eine formale Definition wird in [Martinetz u. Schulten 1994] gegeben.

Lernen bezeichnet, im Gegensatz zum *weichen kompetitiven Lernen*, bei dem noch weitere Neuronen angepasst werden; ein Beispiel dafür ist die Kohonenkarte (siehe 3.2.1).

Neuronenzahl und -anordnung

Die Anzahl der Neuronen kann fest vorgegeben sein oder sich während des Trainingsprozesses ändern. In beiden Fällen kann ferner die „Art", in der die Neuronen verbunden sind, einem festen Schema folgen oder variabel sein: Die Neuronen sowohl der Kohonenkarte als auch des Growing Grid [Fritzke 1995b] bilden eine (meist zwei- oder dreidimensionale) Gitterstruktur. Während jedoch die Kohonenkarte eine feste Anzahl von Neuronen hat, kann diese beim Growing Grid durch das Einfügen von „Neuronenreihen" verändert werden. Das mit Competitive Hebbian Learning kombinierte Neurale Gas (siehe 3.2.2) hat eine feste Zahl von Neuronen, Verbindungen zwischen diesen werden während des Trainings entsprechend der Häufigkeitsverteilung der Eingabevektoren gebildet; beim Wachsenden Neuralen Gas (siehe 3.2.3) ist darüber hinaus die Neuronenzahl variabel.

Parameter-Variation

Die Parameter einer Selbstorganisierenden Karte, etwa die Lernrate, können entweder globale Gültigkeit für alle Neuronen besitzen oder lokal für jedes Neuron einzeln spezifiziert sein. Ferner kann zwischen Varianten mit konstanten und solchen mit zeitlich veränderlichen Parametern unterschieden werden. Die bereits erwähnte Kohonenkarte verwendet globale, zeitlich veränderliche Parameter, das ebenfalls schon erwähnte Wachsende Neurale Gas globale, aber zeitlich konstante Parameter. Ein Beispiel für die Verwendung lokaler, zeitlich veränderlicher Parameter ist das DyCoN-Netz [Perl 2001].

Online- und Offline-Algorithmen

In dieser Arbeit wird nur der Fall betrachtet, dass das Auftreten der Eingabevektoren sich nach einer Häufigkeitsverteilung auf einem kontinuierlichen Eingaberaum richtet, die Menge der Eingabevektoren also beliebig groß ist. In diesem Fall kommen *Online-Algorithmen* zum Einsatz, die nach jeder Präsentation eines Eingabevektors eine Anpassung der Gewichtsvektoren durchführen. Alle in dieser Arbeit betrachteten Algorithmen fallen in diese Kategorie. Dem gegenüber stehen die *Offline-Algorithmen*, die die Anpassung der Gewichtsvektoren nach Präsentation aller Eingabevektoren durchführen, also eine endliche Menge von Eingabevektoren voraussetzen. Ein Beispiel hierfür ist der LBG-Algorithmus [Linde u. a. 1980], der zunächst alle Eingabevektoren präsentiert und dann den neuen Gewichtsvektor eines Neurons als Schwerpunkt der Eingabevektoren, für die dieses Neuron das Gewinnerneuron war, festlegt.

3.2.1 Die Kohonenkarte

Bei der von Teuvo Kohonen 1982 eingeführten *Kohonenkarte* (*Kohonen Feature Map*) [Kohonen 1982] sind die Neuronen so durch Kanten verbunden, dass sie eine periodische, meist zwei- oder dreidimensionale, Gitterstruktur bilden. Diese wird, wie auch die Anzahl N der Neuronen, während des Trainings nicht verändert. Die initialen Gewichtsvektoren der Neuronen werden entweder zufällig gewählt oder gleichmäßig im Eingaberaum verteilt.

Die abstrakte Adaptationsregel (3.6) nimmt im Falle der Kohonenkarte die folgende konkrete Form an:

$$\boldsymbol{w}_i \leftarrow \boldsymbol{w}_i + \eta(t) \cdot h(d^{\mathcal{N}}(c_w, c_i), r_a(t)) \, (s - \boldsymbol{w}_i) \, , \, i = 1, 2, \ldots, N \tag{3.7}$$

Dabei ist die für alle Neuronen identische Lernrate $\eta(t)$ üblicherweise monoton fallend mit $\lim_{t\to\infty} \eta(t) = 0$. Häufig verwendet wird etwa die folgende, durch Wahl der Konstanten $a,\ b \in \mathbb{R}^+$ problemspezifisch anpassbare Funktion:

$$\eta(t) = \frac{a}{1 + b \cdot t} \tag{3.8}$$

Die ebenfalls für alle Neuronen identische Nachbarschaftsfunktion $h(d^{\mathcal{N}}(c_w, c_i), r_a(t))$ ist abhängig vom Abstand $d^{\mathcal{N}}(\cdot, \cdot)$ des zu adaptierenden Neurons c_i vom Gewinnerneuron c_w und dem Aktivierungsradius $r_a(t)$. Der Abstand zweier Neuronen wird dabei über die auf der Neuronenmenge definierte Manhattan-Metrik bestimmt – als die minimale Anzahl von Kanten, die auf dem Weg von dem einen zum anderen Neuron passiert werden muss.
Der Aktivierungsradius $r_a(t)$ legt fest, in welchem Bereich um das Gewinnerneuron eine Anpassung der Neuronen stattfindet. Dies kann in Form einer scharfen Abgrenzung erfolgen, so passt etwa das *0-1-Aktivierungsprofil*

$$h(d^{\mathcal{N}}(c_w, c_i), r_a(t)) = \begin{cases} 1 & \text{für } d^{\mathcal{N}}(c_w, c_i) \leq r_a(t) \\ 0 & \text{sonst} \end{cases} \tag{3.9}$$

alle Neuronen innerhalb einer durch $r_a(t)$ bestimmten Umgebung in gleichem Maße an, alle anderen Neuronen nicht. Ebenso ist es aber auch möglich, eine mit der Entfernung zum Gewinnerneuron stetig abnehmende Adaptation umzusetzen, etwa durch Verwendung des *Gauß-Aktivierungsprofils*

$$h(d^{\mathcal{N}}(c_w, c_i), r_a(t)) = \exp\left(-\frac{d^{\mathcal{N}}(c_w, c_i)^2}{r_a(t)^2}\right) \tag{3.10}$$

Durch den Aktivierungsradius $r_a(t)$ wird in diesem Fall bestimmt, innerhalb welcher Entfernung die Stärke der Anpassung auf $1/e$ der Stärke der Anpassung des Gewinnerneurons absinkt.
Der Aktivierungsradius $r_a(t)$ selbst ist eine zeitabhängige Funktion mit oft stufenförmigem Verlauf: Zu Beginn des Trainings entspricht sein Wert in etwa dem maximal möglichen Manhattan-Abstand zweier Neuronen der Kohonenkarte und wird im Laufe des Trainings – jeweils nach einer bestimmten Zahl von Trainingsschritten – um einen kleinen Betrag herabgesetzt, bis ein vorgegebener Endwert erreicht ist. Die Schrittweiten und Stufenhöhen müssen jeweils problemspezifisch gewählt werden. Der Endwert des Aktivierungsradius wird meist größer als Null gewählt, sodass auch in späten Trainingsphasen nicht nur das Gewinnerneuron adaptiert wird.
Eine derartige Wahl von Aktivierungsradius respektive Nachbarschaftsfunktion bewirkt in Verbindung mit einer Lernrate der Form (3.8), dass zu Beginn des Trainings viele Neuronen in großen Schritten angepasst werden, während später nur noch lokale, kleine Adaptionen vorgenommen werden. Eine zufällig initialisierte Kohonenkarte (Abb. 3.6(a)) breitet sich daher im Training zunächst grob im Eingaberaum aus (Abb. 3.6 (b), (c)) und führt dann durch lokale Änderungen eine Feinanpassung durch (Abb. 3.6 (d)) .

3.2.2 Das Neurale Gas

Das *Neurale Gas* [Martinetz u. Schulten 1991] ist eine Selbstorganisierende Karte mit fester Neuronenzahl, die gänzlich ohne Kanten auskommt. Die Anpassung der – mit zufälligen Positionen initialisierten – Neuronen geschieht abhängig vom euklidischen Abstand ihrer Gewichtsvektoren zum Eingabevektor: Nach Präsentation eines Eingabevektors wird für jedes Neuron c_i dessen *Rang* k_i bestimmt, das ist die Anzahl von Neuronen, deren Gewichtsvektoren einen geringeren euklidischen Abstand zum Eingabevektor haben als der Gewichtsvektor von c_i. Mit diesem Rang und einem zeitabhängigen Parameter $\lambda(t)$, der dem Aktivierungsradius $r_a(t)$ der Kohonenkarte entspricht, regelt die Nachbarschaftsfunktion

$$h(k_i, \lambda(t)) = \exp\left(-\frac{k_i}{\lambda(t)}\right) \tag{3.11}$$

die Stärke der Anpassung der Neuronen:

$$\boldsymbol{w}_i \leftarrow \boldsymbol{w}_i + \eta \cdot e^{-k_i/\lambda}\,(s - \boldsymbol{w}_i)\,, i = 1, 2, \ldots, N \tag{3.12}$$

Der Parameter $\lambda(t)$ sinkt dabei ausgehend von einem Startwert λ_{initial} innerhalb von $t_{\max}$ Trainingsschritten auf seinen Endwert λ_{final} ab und bleibt dann konstant:

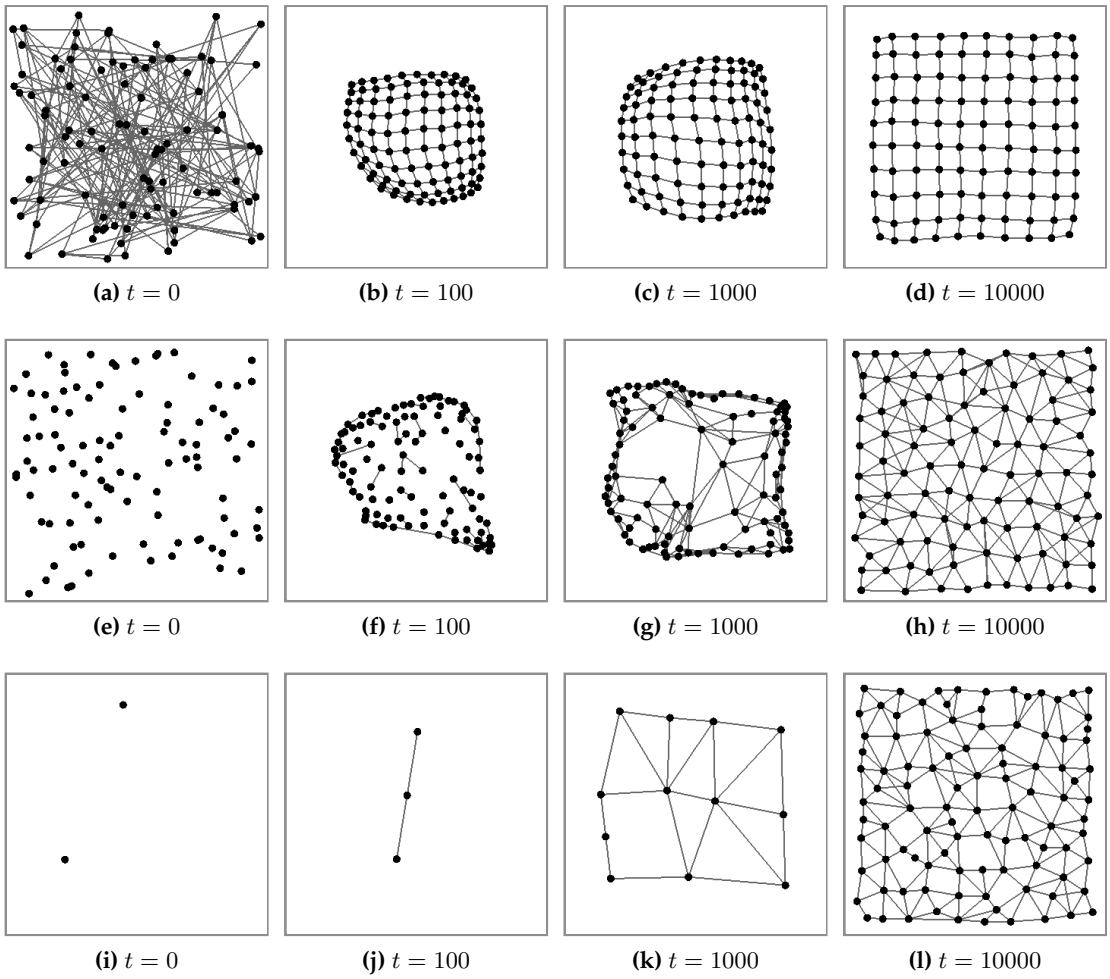
$$\lambda(t) = \begin{cases} \lambda_{\text{initial}} \cdot \left(\frac{\lambda_{\text{final}}}{\lambda_{\text{initial}}}\right)^{\frac{t}{t_{\max}}} & \text{für } t \leq t_{\max} \\ \lambda_{\text{final}} & \text{für } t > t_{\max} \end{cases} \tag{3.13}$$

Von der gleichen Form ist auch die Zeitabhängigkeit der Lernrate η:

$$\eta(t) = \begin{cases} \eta_{\text{initial}} \cdot \left(\frac{\eta_{\text{final}}}{\eta_{\text{initial}}}\right)^{\frac{t}{t_{\max}}} & \text{für } t \leq t_{\max} \\ \eta_{\text{final}} & \text{für } t > t_{\max} \end{cases} \tag{3.14}$$

Competitive Hebbian Learning

Das Neurale Gas wird oftmals kombiniert mit einem als *Competitive Hebbian Learning* [Martinetz 1993] bezeichneten Verfahren, das es ermöglicht, eine topologieerhaltende Abbildung einer Eingabeverteilung mit komplexer oder a-priori unbekannter Topologie zu lernen. Dabei werden Kanten zwischen Neuronen, deren Gewichtsvektoren im Eingaberaum benachbart sind, eingefügt. Auf die Anpassung der Gewichtsvektoren hat dies keinen Einfluss. Da sich die Nachbarschaftsbeziehungen der Neuronen während des Trainings noch ändern, müssen gegebenenfalls Kanten auch wieder entfernt werden; dies wird über das *Alter* der Kanten gesteuert: In jedem Trainingsschritt wird zwischen dem Gewinnerneuron c_w und dem dem Eingabevektor zweitnächsten Neuron c_s eine Kante $e_{w,s}$ mit dem Alter $age_{w,s} = 0$ eingefügt oder, sofern diese beiden Neuronen bereits durch eine Kante verbunden sind, deren Alter auf 0 zurückgesetzt. Anschließend wird das Alter aller zwischen c_w und anderen Neuronen bestehenden Kanten um 1 erhöht; Kanten, deren Alter dann einen Maximalwert age_{max} überschreitet, werden gelöscht.
Bei der so erzeugten Verbindungsstruktur handelt es sich (näherungsweise) um einen Subgraphen der Delaunay-Triangulierung der Neuronen. Abbildung 3.6 (e) - (h) zeigt das Training eines Neuralen Gases und die Entwicklung von Verbindungen zwischen dessen Neuronen.



(a) $t = 0$ **(b)** $t = 100$ **(c)** $t = 1000$ **(d)** $t = 10000$

(e) $t = 0$ **(f)** $t = 100$ **(g)** $t = 1000$ **(h)** $t = 10000$

(i) $t = 0$ **(j)** $t = 100$ **(k)** $t = 1000$ **(l)** $t = 10000$

Abbildung 3.6: Training verschiedener Netztypen mit auf dem Einheitsquadrat $[0, 1]^2$ gleichverteilten Eingaben; jeweils nach – von links nach rechts – 0, 100, 1000 und 10000 Trainingsschritten: **(a) - (d)** Kohonenkarte, **(e) - (h)** Neurales Gas, **(i) - (l)** Wachsendes Neurales Gas

3.2.3 Das Wachsende Neurale Gas

Sowohl bei der Kohonenkarte wie auch beim Neuralen Gas müssen die (meist zufällig initialisierten) Neuronen zunächst durch große Anpassungen grob an die Eingabeverteilung adaptiert werden, bevor die Stärke der Anpassungen reduziert wird, um eine Stabilisierung der Karte zu erreichen. Hieraus ergibt sich die Notwendigkeit, die für das Erreichen einer guten Abbildung der Eingabeverteilung nötige Trainingsdauer im Voraus abzuschätzen und den zeitlichen Verlauf der Parameter entsprechend festzulegen. Ferner muss auch eine Abschätzung der nötigen Zahl von Neuronen vorgenommen werden, da diese während des Trainings nicht mehr verändert werden kann.

Das *Wachsende Neurale Gas* (*Growing Neural Gas*) [Fritzke 1995a], das als inkrementelle Variante des Neuralen Gases betrachtet werden kann, umgeht diese Probleme: Es startet mit nur zwei Neuronen, denen während des Trainings sukzessive weitere hinzugefügt werden, um eine gute Anpassung an die Eingabeverteilung zu erreichen. Somit sind große Anpassungsschritte zu Beginn des Trainings nicht mehr nötig; von Anfang an genügen in jedem Trainingsschritt kleine Anpassungen weniger Neuronen. Daher können alle Parameter des Wachsenden Neuralen Gases konstant gewählt werden.

Der größte Unterschied zum Neuralen Gas besteht in der Bestimmung der anzupassenden Neuronen. Während die Nachbarschaftsfunktion des Neuralen Gases auf Entfernungen im Eingaberaum beruht, basiert die des Wachsenden Neuralen Gases auf Nachbarschaften bezüglich einer auf der Neuronenmenge definierten Topologie. Offensichtlich muss diese, während das Wachsende Neurale Gas wächst, nach und nach erzeugt und angepasst werden. Der dazu verwendete Mechanismus entspricht im Wesentlichen dem im vorangegangenen Abschnitt vorgestellten Competitive Hebbian Learning.

In jedem Lernschritt werden zunächst das dem Reiz nächste (c_w) und zweitnächste (c_s) Neuron ermittelt. Zwischen diesen wird eine Kante $e_{w,s}$ mit dem Alter $age_{w,s} = 0$ eingefügt oder, falls eine solche Kante bereits existiert, deren Alter auf 0 zurückgesetzt. Anschließend werden die Positionen des Gewinnerneurons und aller seiner Nachbarn angepasst, letztere mit einer Lernrate η_n, die niedriger als die Lernrate η_w für die Adaptation des Gewinnerneurons ist:

$$\boldsymbol{w}_w \leftarrow \boldsymbol{w}_w + \eta_w \left(s - \boldsymbol{w}_w\right) \tag{3.15}$$

$$\boldsymbol{w}_{w,j} \leftarrow \boldsymbol{w}_{w,j} + \eta_n \left(s - \boldsymbol{w}_{w,j}\right),\ j = 1, 2, \ldots, N_i \tag{3.16}$$

Dann wird das Alter aller Kanten, die c_w als Endpunkt besitzen, um 1 erhöht und es werden alle Kanten $e_{i,j}$, deren Alter $age_{i,j}$ den zulässigen Maximalwert $age_{\max}$ überschreitet, gelöscht. Ebenfalls gelöscht werden danach Neuronen, die keine Nachbarn mehr besitzen, also nicht mehr über Kanten mit anderen Neuronen verbunden sind.

Das Einfügen eines neuen Neurons erfolgt in vorgegebenen Abständen von jeweils n_{steps} Lernschritten. Die Positionen, an denen neue Neuronen eingefügt werden, werden auf Grundlage von den Neuronen zugeordneten Fehlervariablen err_i bestimmt. Diese schätzen den lokalen Quantisierungsfehler, das ist die mittlere Entfernung eines Neurons von den Eingabevektoren, für die es das Gewinnerneuron ist, ab[10]. In jedem Trainingsschritt wird der Fehler err_w des Gewinnerneurons c_w um dessen quadrierten Abstand zum Eingabevektor s erhöht:

$$err_w \leftarrow err_w + \|\boldsymbol{w}_w - s\|^2 \tag{3.17}$$

Ferner erfolgt, um den Fehler weiter zurückliegender Schritte weniger stark zu berücksichtigen, in jedem Trainingsschritt eine Diskontierung des Quantisierungsfehlers aller Neuronen:

$$err_c \leftarrow err_c - \delta \cdot err_c, c \in \mathcal{N} \tag{3.18}$$

Alle n_{steps} Trainingsschritte[11] wird das Neuron, das den höchsten Quantisierungsfehler aufweist, sowie dessen Nachbar mit dem größten Fehler bestimmt:

$$c_q = \arg\max_{c_i \in \mathcal{N}} err_i \tag{3.19}$$

$$c_f = \arg\max_{j=1,\ldots,N_q} err_{q,j}, \tag{3.20}$$

Zwischen diesen wird ein neues Neuron c_n mit $\boldsymbol{w}_n = 0.5 \cdot (\boldsymbol{w}_q + \boldsymbol{w}_f)$ eingefügt und die Topologie der Neuronenmenge dementsprechend angepasst, indem die Kante $e_{q,f}$

[10] Der Quantisierungsfehler eines Neurons c_i ist mit der Verteilung $P(x)$ der Eingabevektoren gegeben durch das Integral $\int_{V_i} \|x - w_i\|^2 \cdot P(x)dx$.

[11] Das Einfügen erfolgt meist bis zum Erreichen einer Höchstzahl $N_{\max}$ von Neuronen. Alternativ könnte es auch beendet werden, wenn ein vorgegebener Quantisierungsfehler nicht mehr überschritten wird.

gelöscht wird und stattdessen die Kanten $e_{q,n}$ und $e_{f,n}$ in die Kantenmenge $\mathcal{E}$ eingefügt werden. Abschließend werden die Fehlervariablen von c_q und c_f diskontiert und die von c_n wird initialisiert:

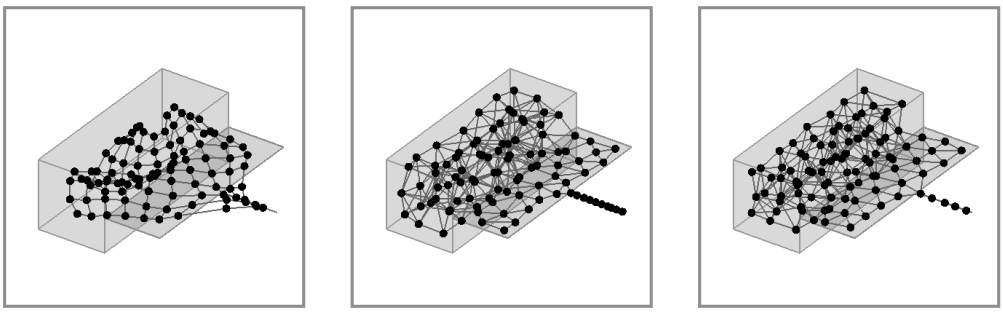
$$err_q \leftarrow err_q - \eta \cdot err_q \tag{3.21}$$

$$err_f \leftarrow err_f - \eta \cdot err_f \tag{3.22}$$

$$err_n \leftarrow 0.5 \cdot (err_q + err_f) \tag{3.23}$$

In Abbildung 3.6 (i) - (l) ist die Entwicklung eines Wachsenden Neuralen Gases während des Trainings mit im Einheitsquadrat $[0, 1]^2$ gleichverteilten Eingabevektoren dargestellt. Es ist deutlich zu erkennen, wie die Karte während des Trainings anwächst.

Wie bereits erwähnt, sind das Neurale Gas und das Wachsende Neurale Gas in der Lage, sich an die Topologie der Eingabeverteilung anzupassen. Dies ist insbesondere dann ein Vorteil gegenüber der Kohonenkarte, wenn die Eingabeverteilung lokal unterschiedliche Dimensionalität aufweist. Einen Vergleich der drei Netztypen in einem solchen Fall gestattet die Abbildung 3.7[12].



(a) **(b)** **(c)**

Abbildung 3.7: Abbildung einer Eingabeverteilung mit lokal unterschiedlicher Dimensionalität durch **(a)** eine Kohonenkarte, **(b)** ein Neurales Gas und **(c)** ein Wachsendes Neurales Gas.

Die Eingabevektoren sind hier Elemente eines Quaders, einer Ebene und einer Linie. Die Kohonenkarte bildet nur den zweidimensionalen Bereich topologieerhaltend ab. Der dreidimensionale Bereich ist, verglichen mit den anderen Kartentypen, nur dünn besetzt und eine ungefähre Abdeckung wird nur durch eine Faltung der zweidimensionalen Kohonenkarte erreicht. Im Bereich der Linie schließlich wird die zweidimensionale Karte „zusammengepresst“ und es bilden sich „Knoten“ von Neuronen, das Ende der Linie wird gar nicht abgedeckt. Wenngleich die Struktur von Neuralem Gas und Wachsendem Neuralen Gas im dreidimensionalen Bereich der Eingabeverteilung etwas unübersichtlich wirkt, was an der dort für die Topologieerhaltung nötigen höheren Anzahl von Kanten liegt, ist doch zu erkennen, dass diese beiden Kartentypen in allen drei Bereichen eine der lokalen Dimension angepasste Struktur entwickeln. Ein Vorteil des Wachsenden gegenüber dem einfachen Neuralen Gas zeigt sich im eindimensionalen Bereich der Verteilung: Beim Neuralen Gas werden alle in der Nähe der Linie initialisierten Neuronen auf diese gezogen; das Resultat ist eine unverhältnismäßig große Zahl von Neuronen in diesem Bereich. Beim Wachsenden Neuralen Gas hingegen werden dort nur so viele Neuronen eingefügt wie nötig, dies führt zu einer ausgewogeneren Verteilung der Neuronen.

[12] Die Neuronen der Kohonenkarte bilden in diesem Beispiel eine zweidimensionale Gitterstruktur.

3.3 Aktionenlernen mit Selbstorganisierenden Karten

Selbstorganisierende Karten lernen eine Abbildung von Eingabevektoren auf Neuronen, also eine Klassifizierung der Eingabedaten, sind jedoch zunächst nicht in der Lage, zu den Eingabevektoren passende Aktionen[13] zu lernen. Mit Hilfe geeigneter Erweiterungen ist jedoch auch dies möglich.

3.3.1 Motorische Karten

Viele im Gehirn für die Muskelansteuerung zuständige Bereiche sind, ähnlich wie die sensorischen Areale, in Kartenform organisiert. Eine lokalisierte Erregung eines solchen Bereichs führt zur Auslösung einer Bewegung, die in gesetzmäßiger Weise mit dem Ort der Erregung variiert. Dies motivierte die Erweiterung der Kohonenkarte zur motorischen Karte [Ritter u. a. 1990], bei der jedem Neuron c_i ein Ausgabevektor $\boldsymbol{u}_i \in \mathbb{R}^m$ zugeordnet ist.
Beim Training wird der Karte neben dem jeweiligen Eingabevektor s eine zu diesem passende Soll-Ausgabe $\boldsymbol{\chi}$ präsentiert, an die die Ausgaben des Gewinnerneurons und seiner Nachbarschaft analog zu (3.6) angepasst werden:

$$\boldsymbol{u}_i \leftarrow \boldsymbol{u}_i + \tilde{\eta}_i(t) \cdot \tilde{h}_i(d,t) \cdot (\boldsymbol{\chi} - \boldsymbol{u}_i) \tag{3.24}$$

Somit wird die Abbildung (3.3) des Eingaberaumes $\mathbb{R}^n$ auf die Menge $\mathcal{N}$ der Neuronen zu einer Abbildung des Eingaberaumes $\mathbb{R}^n$ in den Ausgaberaum $\mathbb{R}^m$ erweitert:

$$\tilde{\Phi} : \mathbb{R}^n \to \mathbb{R}^m,\ s \mapsto \boldsymbol{u}_{w(s)} \tag{3.25}$$

In der beschriebenen Form handelt es sich bei der Motorischen Karte um ein überwachtes Lernverfahren[14], da zu jedem Eingabevektor s die korrekte Ausgabe vorgegeben wird. In [Ritter u. a. 1990] wird zusätzlich eine Version der Motorischen Karte vorgeschlagen, die anstelle vorgegebener korrekter Ausgaben eine Bewertungsfunktion verwendet: In jedem Trainingsschritt wird die Ausgabe des Gewinnerneurons zufällig leicht abgewandelt. Erhält die abgewandelte Version eine Bewertung, die über dem bisherigen Durchschnitt der Bewertungen des Neurons liegt, wird die Änderung übernommen, andernfalls verworfen.
Analog können auch andere Typen Selbstorganisierender Karten erweitert werden. Ein generelles Problem besteht dabei darin, dass sich die Verteilung der Neuronen im Eingaberaum im Wesentlichen nach der Auftrittshäufigkeit der Eingabevektoren richtet und das eigentliche Ziel, das Lernen angemessener Aktionen, nicht berücksichtigt. Wünschenswert wäre hingegen eine Verteilung, bei der in Bereichen des Eingaberaumes, in denen die korrekte Ausgabe stark variiert, viele Neuronen liegen und die Bereiche mit gleicher korrekter Ausgabe mit nur wenigen Neuronen abdeckt.

[13] Unter einer Aktion wird hier ganz allgemein eine Ausgabe verstanden, die als Reaktion auf eine Eingabe erfolgt. Dies kann – muss aber nicht – eine Aktion im eigentlichen Sinne sein, beispielsweise könnte es sich auch um eine Klassifikation der vorangegangenen Eingabe handeln.

[14] Im Bereich der Künstlichen Intelligenz wird unterschieden zwischen überwachtem Lernen, Reinforcement-Learning und unüberwachtem Lernen. Beim überwachten Lernen lernt der Lerner aus Paaren von Eingaben und zugehörigen korrekten Ausgaben, beim Reinforcement-Learning erhält er in Form einer Belohnung (oder Bestrafung) lediglich eine Bewertung seiner Ausgabe, beim unüberwachtem Lernen ist für ihn überhaupt keine Information über die korrekten Ausgaben verfügbar [siehe z.B. Russell u. P.Norvig 2003].

Eine mögliche Lösung dieses Problems stellt die Verwendung eines angepassten Wachsenden Neuralen Gases dar, das Neuronen nicht auf Grundlage des Quantisierungsfehlers, sondern aufgrund der akkumulierten Abweichungen der tatsächlichen von den korrekten Ausgaben einfügt [Fritzke 1995c]:

$$err_w = err_w + \|\boldsymbol{u}_w - \chi\|^2 \tag{3.26}$$

Neue Neuronen werden also in Gebieten eingefügt, in denen die Ausgaben der Neuronen stark von den optimalen Ausgaben abweichen.
Alle derartigen Verfahren setzen jedoch voraus, dass korrekte Ausgaben vorgegeben werden oder zumindest eine direkte Bewertung der erfolgten Ausgaben vorgenommen wird; für das Lernen von Aktionen im für viele Reinforcement-Learning-Probleme typischen Fall verzögerter Belohnungen ist keines dieser Verfahren geeignet.

3.3.2 Kombination mit Reinforcement-Learning-Verfahren

In der Literatur sind eine ganze Reihe von zum Teil sehr unterschiedlichen Ansätzen zum Lernen von Aktionen zu finden, die auf einer Kombination einer Selbstorganisierenden Karte mit einem Reinforcement-Learning-Verfahren basieren. Dieses Feld ist zu weit, um an dieser Stelle einen auch nur ansatzweise vollständigen Überblick geben zu können. Im Folgenden wird daher nur auf die wohl am häufigsten zu findende Form einer derartigen Kombination eingegangen.

Wie am Ende von Abschnitt 2.2 ausgeführt, besteht ein wesentlicher Nachteil traditioneller tabellarischer Reinforcement-Learning-Verfahren darin, dass sie für jeden Zustand respektive jedes Zustands-Aktions-Paar eine Schätzung seines Wertes speichern müssen, wodurch ihre praktische Anwendbarkeit auf Probleme mit kleinen Zustandsräumen und Aktionsmengen beschränkt ist. Oft gestattet jedoch eine geeignete Verkleinerung des Zustandsraums auch bei Problemen mit eigentlich großem Zustandsraum die Anwendung eines tabellarischen Verfahrens.
Vielfach wurden in diesem Sinne Selbstorganisierende Karten dazu verwendet, den kontinuierlichen Zustandsraum eines Reinforcement-Learning-Problems vermöge der Abbildung (3.3) zu diskretisieren, um dann auf den so erhaltenen – von den Neuronen der Selbstorganisierenden Karte dargestellten – Zuständen ein klassisches Reinforcement-Learning-Verfahren, beispielsweise das in 2.2.3 vorgestellte Q-Learning anzuwenden. In dieser Weise wurden Kohonenkarten (z.B.[Barreca u. Buttazzo 1993]) ebenso eingesetzt wie Neurale Gase (z.B. [Herrmann u. Der 1995]) und Wachsende Neurale Gase (z.B. [Stephan u. a. 2003]), wobei in einigen Fällen, etwa in [Herrmann u. Der 1995], auch die Kriterien, nach denen die Positionierung der Neuronen erfolgt, angepasst wurden.
Ein Problem dieser Ansätze ist darin zu sehen, dass das Training der Selbstorganisierenden Karte parallel zum Lernen der Aktionen durch den mit dieser kombinierten Reinforcement-Learning-Algorithmus erfolgt. Da die Zustände respektive Zustands-Aktions-Paare für die Werte gelernt werden, die durch die Position der Neuronen bestimmt werden, birgt dies die Gefahr, dass bereits gelernte Werte durch die Bewegung der Neuronen wieder vergessen werden.
Ein weiteres Problem besteht darin, dass die auftretenden Zustände in Reinforcement-Learning-Szenarien nur in bestimmten Reihenfolgen auftreten: Abgesehen von den Startzuständen der Lernepisoden wird jeder Zustand – also jede Eingabe – durch den vorangegangenen Zustand und die ausgeführte Aktion bestimmt. Dies kann dazu führen, dass

später in einer Lernepisode auftretende Anpassungen der Neuronen frühere Änderungen überlagern und sich die Neuronen in der Nähe der Episoden-Endzustände konzentrieren, sodass nur noch deren Werte gelernt werden.

3.4 Zusammenfassung

Künstliche Neuronale Netze stellen – neben den Evolutionären Algorithmen – die zweite große Gruppe biologisch inspirierter Lernverfahren im Bereich der Künstlichen Intelligenz dar. Sie imitieren – in stark abstrahierter Form – die Informationsverarbeitungsmechanismen tierischer respektive menschlicher Nervensysteme und Gehirne.
Eine Klasse Künstlicher Neuronaler Netze bilden die von den somatotopischen Karten des Gehirns inspirierten Selbstorganisierenden Karten. Diese lernen eine Abbildung eingegebener Reize auf (künstliche) Neuronen, mithin kann ein solches Neuron als Repräsentant aller auf es abgebildeten Reize angesehen werden. Eine wesentliche Eigenschaft besagter Abbildung besteht darin, dass ähnliche Neuronen stets ähnliche Reize repräsentieren. Unter gewissen Umständen gilt auch umgekehrt, dass ähnliche Reize durch ähnliche Neuronen repräsentiert werden.
Selbstorganisierende Karten nehmen somit eine Klassifikation der ihnen präsentierten Eingaben vor, die jedoch nur implizit durch die während eines Trainingsprozesses gebildete Struktur der Karte gegeben ist. Soll die vorgenommene Klassifikation als Grundlage für die Auswahl von zu den jeweiligen Eingaben passenden Ausgaben – etwa im Falle des Lernens von Aktionen – dienen, muss die Selbstorganisierende Karte erweitert oder mit einem anderem Lernverfahren, zum Beispiel einem Reinforcement-Learning-Verfahren, gekoppelt werden.

Kapitel 4

Ein Hybrides Lernendes Klassifizierendes System

In diesem Kapitel wird ein neu entwickeltes Lernendes Klassifizierendes System für den Einsatz in reellwertig kodierten Lernumgebungen eingeführt, das Charakteristika sowohl des XCS wie auch eines Wachsenden Neuralen Gases aufweist und daher als *Hybrides Lernendes Klassifizierendes System* (HCS) bezeichnet werden soll. In seiner Struktur und prinzipiellen Funktionsweise gleicht dieses System dem XCS, jedoch sind die Bedingungen seiner Klassifizierer durch Gewichtsvektoren gegeben, wie sie die Neuronen Selbstorganisierender Karten aufweisen. Ferner nutzt das HCS zusätzlich zum Genetischen Algorithmus einen weiteren Lernmechanismus, der an das Neuronenadaptationsverfahren des (Wachsenden) Neuralen Gases angelehnt ist.
In Abschnitt 4.1 wird zunächst die Entwicklung eines derartigen Hybridsystems motiviert: Ausgehend von der Betrachtung zweier auf den ersten Blick sehr ähnlicher Lernumgebungen wird ein grundlegendes Problem intervallbasierter Klassifiziererbedingungen identifiziert und es wird dargelegt, dass die Verwendung von Gewichtsvektoren als Klassifiziererbedingungen einen Ansatzpunkt zu dessen Lösung bietet.
An diese Motivation schließt eine detaillierte Beschreibung des HCS an, die sich entsprechend der vier – auf Seite 23 genannten – zentralen Komponenten Lernender Klassifizierender Systeme gliedert: Der Abschnitt 4.2 widmet sich der Wissensbasis des HCS und geht insbesondere auf dessen Klassifizierer und deren Unterschiede zu den intervallbasierten Klassifizierern des reellwertig kodierten XCS ein. In Abschnitt 4.3 wird die Performance-Komponente des HCS beschrieben, die gegenüber der des XCS eine ganze Reihe von Modifikationen aufweist. Auch die Reinforcement-Komponente kann nicht unverändert vom XCS übernommen werden, wie in Abschnitt 4.4 dargelegt wird. Mit der Discovery-Komponente des HCS und insbesondere mit dem neu eingeführten Verfahren zur Adaptation der Klassifizierer befasst sich der Abschnitt 4.5. Abschnitt 4.6 stellt die zuvor einzeln beschriebenen Komponenten in den Kontext des Gesamtsystems und gewährt so einen Überblick über das HCS, bevor dieses in Abschnitt 4.7 mit anderen in der Literatur beschriebenen Kombinationen von Lernenden Klassifizierenden Systemen und Neuronalen Netzen verglichen wird. Abschnitt 4.8 schließlich fasst das Kapitel zusammen.

4.1 Motivation

Lernende Klassifizierende Systeme – insbesondere das XCS – wurden erfolgreich zum Lösen von Klassifikationsproblemen, zur Funktionsapproximation und zum Lernen von Aktionen eingesetzt. Ihre Fähigkeit zur Generalisierung über Lernumgebungszustände ermöglicht ihre Verwendung auch in Lernumgebungen mit großem diskretem oder kontinuierlichem Zustandsraum, in denen klassische (tabellarische) Reinforcement-Learning-Verfahren, die jeden Zustand respektive jedes Zustands-Aktions-Paar einzeln bewerten, nicht ohne weiteres eingesetzt werden können.
Um beim Lernen von dieser Generalisierungsfähigkeit profitieren zu können, müssen jedoch zwei Bedingungen erfüllt sein: Erstens muss die jeweils betrachtete Lernumgebung Generalisierungen überhaupt zulassen – es muss Umgebungszustände geben, in denen gleiche Aktionen zu gleichen Belohnungen und Returns führen. Zweitens müssen die in der Lernumgebung möglichen Generalisierungen auch durch Klassifiziererbedingungen ausdrückbar sein. Wie diese zweite Bedingung zu verstehen ist, kann anhand der in Abbildung 4.1 skizzierten, einfach strukturierten Lernumgebungen verdeutlicht werden:

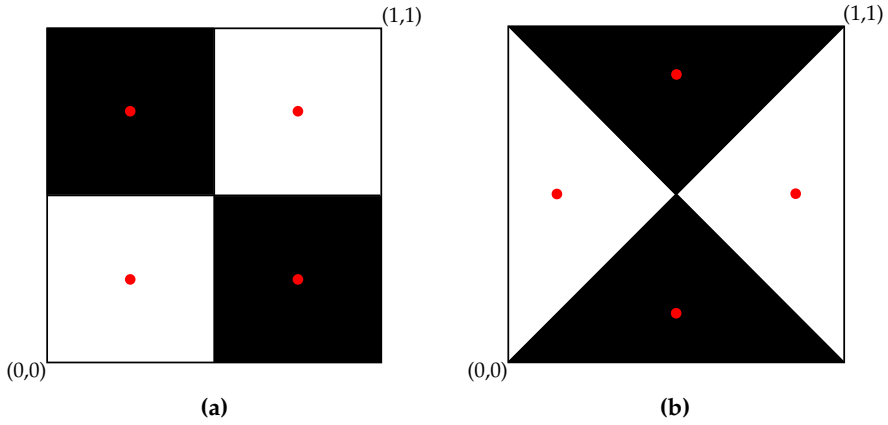


Abbildung 4.1: Zwei Lernumgebungen mit Zustandsraum $\mathcal{S} = [0,1)^2$ und Aktionsmenge $\mathcal{A} = \{a_w, a_s\}$. In den weißen (schwarzen) Teilen des Zustandsraumes ist $a_w(a_s)$ die beste Aktion: **(a)** Achsenparallele und **(b)** Diagonale Unterteilung von $\mathcal{S}$.

In beiden Lernumgebungen ist der zweidimensionale Zustandsraum $\mathcal{S} = [0,1)^2$ in vier Gebiete gleicher Größe unterteilt. Von den zwei möglichen Aktionen a_w und a_s ist in den in der Abbildung weiß dargestellten Gebieten die erste optimal, in den schwarzen Bereichen hingegen die zweite. Die Ausführung einer richtigen Aktion wird jeweils mit einer Belohnung $r = 1000$ honoriert, bei Ausführung einer falschen Aktion wird keine Belohnung gewährt ($r = 0$). Die beiden Lernumgebungen unterscheiden sich allein durch die Art der Unterteilung des Zustandsraums: In der ersten sind die „Grenzen“ zwischen den vier Gebieten achsenparallel ausgerichtet, in der zweiten verlaufen sie diagonal.
Unter Verwendung intervallbasierter Bedingungen kann im Fall der ersten Lernumgebung über jeden der vier Bereiche des Zustandsraums als Ganzes generalisiert werden, sodass acht Klassifizierer[1] ausreichen, um jeden Zustand der Lernumgebung richtig zu klassifizieren. Im Fall der zweiten Lernumgebung muss jeder Bereich des Zustandsraums unter Verwendung vieler Klassifizierer approximiert werden; es werden also deutlich mehr Klassifizierer benötigt, um das Problem (näherungsweise) zu lösen.
Dieses Problem der schlechten Abbildbarkeit der in einer Lernumgebung möglichen Generalisierungen durch intervallbasierte Bedingungen betrifft natürlich nicht allein das

[1] Die Anzahl 8 ergibt sich als Produkt der Zahl der Gebiete (4) und der Zahl möglicher Aktionen (2).

obige einfache Beispiel – es ist vielmehr zu vermuten, dass dieser Typ von Bedingungen vielen Lernumgebungen – insbesondere solchen, die Realwelt-Probleme modellieren – nicht gerecht wird. Es liegt daher nahe, nach alternativen Bedingungstypen zu suchen, die möglichst vielfältige Generalisierungen erlauben und damit kompaktere – weniger Klassifizierer benötigende – Lösungen ermöglichen.
In [Butz 2005; Butz u. a. 2006] wurden *kernelbasierte ellipsoidale Bedingungen* eingeführt, die von Zuständen aus einem hyperellipsoid-förmigen Bereich des Zustandsraumes erfüllt werden. Da neben dem Mittelpunkt dieser Hyperellipsoide auch die Länge und Orientierung ihrer Hauptachsen evolviert wird, gibt es im Hinblick auf die Bedingungsstruktur keine ausgezeichneten Achsen im Zustandsraum. Viele Unterteilungen eines Zustandsraumes sind daher besser abbildbar als bei Verwendung intervallbasierter Bedingungen. Noch flexibler sind *auf konvexen Hüllen basierende Bedingungen* [Lanzi u. Wilson 2006]. Eine solche wird von allen Zuständen in der konvexen Hülle der sie definierenden $k \geq 3$ Punkte des Zustandsraumes erfüllt. Beide Bedingungstypen führen zu guten Lernergebnissen, jedoch sind die mit ihnen evolvierten Populationen in vielen Fällen ebenso groß oder größer als die unter Verwendung intervallbasierter Bedingungen gefundenen – das Ziel einer „effizienteren" Generalisierung erreichen sie also nicht. Zudem sind beide Typen deutlich komplizierter aufgebaut als intervallbasierte Bedingungen, sodass sich die Frage stellt, ob es nicht eine „einfachere" Alternative gibt.
Einen Ansatzpunkt liefern die Beispiele aus Abbildung 4.1: In beiden Lernumgebungen kann die Unterteilung des Zustandsraums in Bereiche mit gleicher optimaler Aktion durch eine Voronoizerlegung mit jeweils vier Zentren – in der Abbildung durch rote Punkte markiert – beschrieben werden. Dies legt es nahe, Bedingungen zu benutzen, die ebenfalls eine Voronoizerlegung des Zustandsraumes induzieren – Bedingungen also, die durch Gewichtsvektoren beschrieben werden und von allen in der zugehörigen Voronoizelle liegenden Zuständen erfüllt werden.
Derartige Bedingungen ermöglichen es, sehr viele Unterteilungen eines Zustandsraumes – alle als Voronoizerlegungen beschreibbaren – abzubilden. Sie bieten zudem den Vorteil eines kleineren Suchraums des Genetischen Algorithmus: Es werden n-dimensionale reellwertige Vektoren evolviert. Bei auf konvexen Hüllen basierenden Klassifiziererbedingungen hingegen treten Chromosomen der Länge $k \cdot n$ auf, bei hyperellipsoidalen Bedingungen ist der Genraum sogar $(n + n^2)$-dimensional.
Diesen Vorteilen steht ein wesentlicher Nachteil gegenüber: Klassifizierer, die Gewichtsvektoren als Bedingungen verwenden, sind – anders als „klassische" Klassifizierer – stark voneinander abhängig: Die Voronoizelle eines derartigen Klassifizierers – also die Menge der zu ihm passenden Lernumgebungszustände – ist durch seine Gewichtsvektor-Bedingung allein nicht festgelegt, sie wird vielmehr wesentlich von den Gewichtsvektoren umliegender Klassifizierer mitbestimmt. Um ein gutes Lernergebnis zu erzielen, müssen daher die Klassifizierer nicht nur absolut, sondern auch relativ zueinander genau positioniert werden. Dies lässt es sinnvoll erscheinen, neben dem Genetischen Algorithmus ein weiteres Lernverfahren einzusetzen, das die „Feinpositionierung" der Klassifizierer übernimmt. Es ist naheliegend, dieses ähnlich den Neuronenadaptationsmechanismen Selbstorganisierender Karten zu gestalten und dabei die Gewichtsvektoren der Klassifizierer in Abhängigkeit von der Genauigkeit ihrer Vorhersagen anzupassen.
Eine Population derartiger, mit Bedingungen in Form von Gewichtsvektoren ausgestatteter Klassifizierer kann als Klassifizierer-Netz angesehen werden, bei dem – analog einer Selbstorganisierenden Karte – ähnliche Zustände die Bedingungen benachbarter Klassifizierer erfüllen. Anders als bei einer Selbstorganisierenden Karte ist die Klassifikation der Eingaben jedoch nicht implizit durch die Struktur der Karte respektive der Population gegeben, sondern erfolgt explizit durch die Aktionen der Klassifizierer.

4.2 Wissensbasis/Population

Bei der Population des HCS handelt es sich ebenso wie bei der des XCS um eine Menge

$$P = \{cl_1, cl_2, \ldots, cl_N\} \tag{4.1}$$

von (Makro-)klassifizierern. Die Größe der Population ist variabel, unterliegt jedoch der Einschränkung, dass die Summe der Vielfachheiten der in der Population P enthaltenen Klassifizierer einen Maximalwert $N_{\max}$ nicht überschreiten darf:

$$\sum_{cl \in P} cl.num \overset{!}{\leq} N_{\max} \tag{4.2}$$

Während aber die Population des XCS eine reine Ansammlung von Klassifizierern darstellt, wird die Population des HCS im Laufe des Lernvorganges – ähnlich wie beim Topologielernen des (Wachsenden) Neuralen Gases – durch Einfügen von Kanten zwischen Klassifizierern mit einer topologischen Struktur versehen. Genauer gesagt wird, da das HCS nur Kanten zwischen Klassifizierern mit gleicher Aktion gestattet, für jede Aktion $a \in \mathcal{A}$ die durch

$$P_a := \{cl \in P \,|\, cl.action = a\} \tag{4.3}$$

definierte *Teilpopulation mit Aktion* a mit einer eigenen Topologie ausgestattet, die durch die Menge $\mathcal{E}_a$ aller Kanten, die zwei in P_a enthaltene Klassifizierer verbinden, beschrieben wird. Die Menge aller Kanten zwischen Klassifizierern der Population P wird mit $\mathcal{E}$ bezeichnet und es gilt:

$$\mathcal{E} = \bigcup_{a \in \mathcal{A}} \mathcal{E}_a \tag{4.4}$$

Die Erzeugung dieser Topologie(n) erfolgt integriert in die Bildung des Match-Sets und wird daher erst in Abschnitt 4.3.1 thematisiert. Hier wird zunächst näher auf die Klassifizierer des HCS – insbesondere auf die Syntax ihrer Bedingungen und ihre Attribute – eingegangen. Anschließend werden dann die Verfahren besprochen, die verwendet werden, um Klassifizierer in die Population einzufügen und aus ihr zu löschen.

4.2.1 Die Klassifizierer des HCS

Die Klassifizierer des HCS unterscheiden sich hinsichtlich ihrer Grundstruktur nicht von denen anderer Lernender Klassifizierender Systeme; als zentrale Bestandteile besitzen sie eine Bedingung und eine Aktion. Da ihre Bedingungen durch Gewichtsvektoren $\boldsymbol{w} \in \mathbb{R}^n$ repräsentiert werden, gleichen sie aber auch Neuronen Selbstorganisierender Karten, die um eine Ausgabe – in Gestalt der Klassifiziereraktion – erweitert wurden.
Die Dimension n der Gewichtsvektoren entspricht der Anzahl der Zustandsvariablen, also der Dimension des Zustandsraumes $\mathcal{S} \in \mathbb{R}^n$, der jeweils verwendeten Lernumgebung. Bei den Einträgen der Gewichtsvektoren handelt es sich um – im Prinzip beliebige – reelle Zahlen; welche Werte tatsächlich auftreten, hängt in erster Linie von der jeweils verwendeten Lernumgebung ab. Als Klassifiziereraktion kann jedes Element der zur Lernumgebung gehörenden Aktionsmenge $\mathcal{A} = \{a_1, a_2, \ldots, a_m\}$ in Erscheinung treten.

Bedingung der HCS-Klassifizierer

Die Bestimmung der zu einem Umgebungszustand passenden Klassifizierer erfolgt analog der Ermittlung des Gewinnerneurons bei einer Selbstorganisierenden Karte: Ein

HCS-Klassifizierer passt genau dann zu einem Zustand $s \in \mathcal{S}$, wenn der euklidische Abstand seines Gewichtsvektors $\boldsymbol{w}$ von s kleiner oder gleich dem euklidischen Abstand zwischen s und dem Gewichtsvektors jedes anderen in der Population enthaltenen Klassifizierers mit der gleichen Aktion ist. Offensichtlich können somit die auf Seite 46 zur Eindeutigkeit der Gewinnerneuronen bei Selbstorganisierenden Karten gemachten Anmerkungen unmittelbar auf das HCS übertragen werden. Dementsprechend wird im Folgenden davon ausgegangen, dass es in einer HCS-Population zu jedem Umgebungszustand $s \in \mathcal{S}$ und jeder Aktion $a \in \mathcal{A}$ stets höchstens einen passenden Klassifizierer $cl^a_{w(s)}$ gibt:

$$cl^a_{w(s)} := \arg\min_{cl \in P_a} \|cl.\boldsymbol{w} - s\| \tag{4.5}$$

In Anlehnung an die bei Selbstorganisierenden Karten gebräuchliche Bezeichnung *Gewinnerneuron* sollen passende HCS-Klassifizierer als *Gewinnerklassifizierer* oder kurz als *Gewinner* bezeichnet werden. Wenn im Folgenden von einem solchen die Rede ist, so geht im Allgemeinen aus dem Kontext eindeutig hervor, auf welchen Umgebungszustand sich dessen Gewinnerstatus bezieht. Ist das der Fall, wird ab jetzt statt $cl^a_{w(s)}$ nur noch cl^a_w geschrieben.
Analog zum Gewinner cl^a_w wird der *Zweite* cl^a_s definiert als der Klassifizierer mit Aktion a, dessen Gewichtsvektor den zweitniedrigsten euklidischen Abstand zum aktuellen Umgebungszustand aufweist[2]:

$$cl^a_s := \arg\min_{cl \in P_a \setminus \{cl^a_w\}} \|cl.\boldsymbol{w} - s\| \tag{4.6}$$

Bezüglich dieser und ähnlicher Abstandsbetrachtungen wird ab jetzt nicht mehr streng zwischen einem HCS-Klassifizierer und seinem Gewichtsvektor unterschieden: Wenn im Folgenden vom Abstand eines Klassifizierers von einem Umgebungszustand die Rede ist, so ist damit der euklidische Abstand des Gewichtsvektors des Klassifizierers von dem Umgebungszustand gemeint. Entsprechend ist unter dem Abstand zweier HCS-Klassifizierer die euklidische Distanz ihrer Gewichtsvektoren zu verstehen.

Attribute der HCS-Klassifizierer

Die Klassifizierer des HCS besitzen eine Reihe von Attributen, die – wie der Tabelle 4.1 zu entnehmen ist – weitgehend den auf Seite 31 genannten Attributen der Klassifizierer des XCS entsprechen. In drei Punkten jedoch unterscheiden sich die Klassifizierer des HCS hinsichtlich ihrer Attribute von denen des XCS:
Erstens ist die Erfahrung exp anders definiert als das Attribut gleichen Namens beim XCS: Die Erfahrung eines Klassifizierers soll bei beiden Systemen als Kriterium für die „Verlässlichkeit“ seiner übrigen Attribute dienen, indem sie Aufschluss darüber gibt, wie oft er angewendet und bewertet wurde. Während aber das XCS stets die Attribute aller in einem Action-Set enthaltenen Klassifizierer aktualisiert, passt das HCS nur die der enthaltenen Gewinnerklassifizierer an. Dementsprechend gibt die Erfahrung eines HCS-Klassifizierers nicht an, wie oft er Teil eines Action-Sets war, sondern wie oft er ein Gewinner war, dessen Aktion zur Ausführung kam.

Zweitens verfügen die Klassifizierer des HCS über kein Attribut, das die Größe der Action-Sets, denen sie angehören, abschätzt: Beim XCS passen die in einem Action-Set enthaltenen Klassifizierer stets zumindest teilweise zu den gleichen Umgebungs-

[2] Das s in der Bezeichnung cl^a_s steht nicht für den Umgebungszustand, sondern ist als Abkürzung für 'Second' zu verstehen; das w in cl^a_w steht entsprechend für 'Winner'.

p	Die Vorhersage $p \in \mathbb{R}$ schätzt den bei Anwendung des Klassifizierers zu erwartenden Return ab.
ε	Der Vorhersagefehler $\varepsilon \in \mathbb{R}$ schätzt die Abweichung der beobachteten Returns von der Return-Vorhersage p ab.
f	Die Fitness $f \in \mathbb{R}$ des Klassifizierers bestimmt dessen Chancen, vom Genetischen Algorithmus als Elternindividuum selektiert zu werden.
κ	Die Genauigkeit $\kappa \in \mathbb{R}$ des Klassifizierers bildet die Grundlage für die Berechnung seiner Fitness.
exp	Die Erfahrung $exp \in \mathbb{N}$ des Klassifizierers gibt an, wie oft er ein Gewinner war und bewertet wurde.
ts	Der Zeitstempel $ts \in \mathbb{N}$ gibt an, wann zuletzt ein Genetischer Algorithmus auf einem Action-Set ausgeführt wurde, dem der Klassifizierer angehörte.
num	Die Vielfachheit $num \in \mathbb{N}$ gibt an, in wie vielen Kopien der Klassifizierer in der Population vorliegt.
$avgd$	Die Distanzabschätzung $avgd \in \mathbb{R}$ schätzt den mittleren Abstand von den Umgebungszuständen, für die der Klassifizierer ein Gewinner ist, ab.
r	Der Subsumtions-Radius $r \in \mathbb{R}$ definiert eine Sphäre um den Gewichtsvektor in der die Abweichung der beobachteten Returns von p kleiner ist als ε_0.

Tabelle 4.1: Die Attribute der Klassifizierer des HCS

zuständen. Derart stellt jedes Action-Set beim XCS eine Menge von in einer bestimmten „Nische“ der Lernumgebung anwendbaren Klassifizierern dar. Mit dem Ziel, die beschränkte Anzahl zur Verfügung stehender Klassifizierer gleichmäßig auf diese Nischen zu verteilen, werden Klassifizierer, die regelmäßig großen Action-Sets angehören, mit höherer Wahrscheinlichkeit gelöscht. Beim HCS hingegen definiert sich jeder einzelne Klassifizierer durch seine Voronoizelle seine eigene Nische, sodass es nicht sinnvoll ist, eine Action-Set-Größen-Abschätzung bei der Bestimmung zu löschender Klassifizierer zu berücksichtigen.

Drittens sind mit der *Distanzabschätzung* $avgd$ und dem *Subsumtionsradius* r zwei, im Vergleich zum XCS neue, Attribute hinzugekommen:

Distanzabschätzung Während an binär kodierten und intervallbasierten Klassifiziererbedingungen direkt ablesbar ist, wie generell ein Klassifizierer ist, ist dies aus einem Gewichtsvektor nicht ersichtlich. Die Distanzabschätzung eines HCS-Klassifizierers liefert ein – wenn auch grobes – Maß für die Größe seiner Voronoizelle und damit für seine Allgemeinheit, indem sie den mittleren euklidischen Abstand des Klassifizierers von den Zuständen, für die er ein Gewinner ist, abschätzt.

Subsumtionsradius Der Subsumtionsradius eines HCS-Klassifizierers schätzt ab, innerhalb welches Radius um seinen Gewichtsvektor Zustände liegen, für die die bei Anwendung des Klassifizierers beobachteten Returns um einen Betrag kleiner der Fehlerschranke ε_0 von seiner Vorhersage abweichen.

Der Quotient aus dem Subsumtionsradius r und der Distanzabschätzung $avgd$ eines HCS-Klassifizierers schätzt somit den Anteil seiner Voronoizelle ab, innerhalb dessen er die zu erwartenden Returns mit einer Abweichung kleiner als ε_0 vorhersagt. Dieser

Quotient gestattet es, die Genauigkeit eines Klassifizierers differenzierter zu beurteilen, als es allein aufgrund seines Vorhersagefehlers möglich wäre. Dieser nämlich ist unter Umständen relativ ungenau: Umfasst etwa nach einer kleinen Verschiebung eines Klassifizierers dessen Voronoizelle an ihrem Rand einen – wenn auch nur kleinen – Bereich von Zuständen, deren Returnniveau sich stark vom dem im Rest der Voronoizelle unterscheidet[3], so führt eine Anwendung des Klassifizierers auf einen solchen Zustand zu einer deutlichen Erhöhung seines Vorhersagefehlers, wobei oft die Fehlerschranke ε_0 überschritten wird. Somit kann der Vorhersagefehler eines HCS-Klassifizierers auch dann einen großen Wert annehmen, wenn seine Vorhersage für einen großen Bereich von Zuständen (im Inneren seiner Voronoizelle) nur wenig von den tatsächlich zu erwartenden Returns abweicht.

Berechnete Klassifizierervorhersagen

Die Vorhersage eines Klassifizierers kann – wie schon beim XCS – ein skalarer Wert oder aber eine in jeder Komponente des Lernumgebungszustandes $s = (\xi_1, \xi_2, \ldots, \xi_n)^T \in \mathcal{S}$ lineare Funktion sein:

$$p(s) = v_0 \cdot \xi_0 + \sum_{i=1}^{n} v_i \cdot (\xi_i - \boldsymbol{w}_i) \tag{4.7}$$

Dabei ist wiederum ξ_0 ein Parameter des Lernenden Klassifizierenden Systems und $\boldsymbol{v} = (v_0, v_1, \ldots, v_n)^T$ ein Klassifizierer-spezifischer Vektor von Gewichten für die Vorhersagenberechnung.
Denkbar sind im Prinzip auch andere Formen der Abhängigkeit der Klassifizierervorhersage vom Lernumgebungszustand, die durch eine parametrisierte Funktion vorgegeben werden können. Eine solche Funktion wird jedoch im Allgemeinen nicht linear in den Parametern sein, sodass die Bestimmung respektive Anpassung der Parameter auf das Lösen eines nicht-linearen Ausgleichsproblems hinausläuft, was in der Regel nur durch ein iteratives Verfahren möglich ist [Stoer 1994]. Der Einsatz eines iterativen und damit zeitaufwändigen Verfahrens aber ist in einem eher auf das Online-Lernen ausgerichtetem Lernenden Klassifizierenden System wie dem HCS (oder dem XCS) wenig wünschenswert[4]. In der vorliegenden Arbeit werden berechnete Klassifizierervorhersagen daher nur in der durch (4.7) gegebenen Form betrachtet.

Kontextsensitivität der HCS-Klassifizierer

Aufgrund der oben beschriebenen Syntax der Gewichtsvektor-Bedingungen der HCS-Klassifizierer besteht trotz gleicher Grundstruktur ein wesentlicher konzeptioneller Unterschied zwischen diesen und den Klassifizierern des XCS. Letztere sind – wie auch die Klassifizierer anderer herkömmlicher Lernender Klassifizierender Systeme – *kontextfrei* in dem Sinne, dass der *Matchbereich* eines Klassifizierers cl – das ist die Menge

$$\mathcal{M}_{cl} := \{ s \in \mathcal{S} \,|\, s \text{ erfüllt die Bedingung von } cl \} \tag{4.8}$$

[3] Dieses Szenario betrifft, was das Problem verschärft, in besonderem Maße (annähernd) maximal generelle Klassifizierer.

[4] Bei der in Abschnitt 2.4.8 erwähnten Erweiterung des XCS um polynomielle Klassifizierervorhersagen in [Lanzi u. a. 2005c] wurde auf einen „Trick“ zurückgegriffen, der es erlaubt, polynomielle Klassifizierervorhersagen durch eine in den Komponenten des Umgebungszustandes lineare Funktion zu berechnen: Statt den eigentlichen Zuständen $s = (\xi_1, \xi_2, \ldots, \xi_n)^T$ wurden dem XCS um Potenzterme erweiterte Zustände $\hat{s} = (\xi_1, \xi_1^2, \ldots, \xi_1^k, \ldots, \xi_n, \xi_n^2, \ldots, \xi_n^k)^T$ präsentiert. Eine in jeder Komponente von $\hat{s}$ lineare Klassifizierervorhersage ist offenbar ein Polynom vom Grad k in jeder Komponente von s.

der Zustände, zu denen cl passt – allein durch dessen Bedingung festgelegt und vollständig unabhängig ist von den anderen in der Population eines solchen Systems enthaltenen Klassifizierern. Dies gilt für ternär kodierte wie auch für intervallbasierte Bedingungen[5]. Der Matchbereich eines HCS-Klassifizierers hingegen ist die Voronoizelle

$$\mathcal{M}_{cl} = V_{cl.\boldsymbol{w}} := \left\{ s \in \mathcal{S} \,\middle|\, \left\| s - cl.\boldsymbol{w} \right\| \leq \left\| s - \bar{cl}.\boldsymbol{w} \right\|, \bar{cl} \in P_{cl.action} \right\} \tag{4.9}$$

seines Gewichtsvektors $\boldsymbol{w}$ und wird somit wesentlich durch die anderen, in der Teilpopulation $P_{cl.action}$ enthaltenen Klassifizierer mitbestimmt. Die Klassifizierer des HCS sind somit *kontextsensitiv*: Ob ein Umgebungszustand die Bedingung eines Klassifizierers erfüllt, hängt davon ab, welche anderen Klassifizierer in der Population enthalten sind.
Infolge dieser Kontextsensitivität sind die Matchbereiche der Klassifizierer beim HCS, anders als bei anderen Lernenden Klassifizierenden Systemen, nicht unveränderlich, ihre Voronoizellen passen sich vielmehr bei jeder Modifikation der Population an: Wird etwa ein neuer Klassifizierer in die Population eingefügt, so „übernimmt" er Teile der Matchbereiche anderer Klassifizierer. Wird hingegen ein Klassifizierer aus der Population gelöscht, vergrößern sich die Voronoizellen der ihn umgebenden Klassifizierer. Beim Verschieben, also der Veränderung des Gewichtsvektors eines Klassifizierers, schließlich treten beide Effekte auf – bei dem verschobenen Klassifizierer ebenso wie bei anderen.

Die Kontextsensitivität der HCS-Klassifizierer und die Veränderlichkeit ihrer Matchbereiche wirkt sich während des Lernvorganges des HCS auch auf die Attribute der Klassifizierer aus, was am Beispiel der Klassifizierervorhersagen leicht zu sehen ist: Die Vorhersage eines Klassifizierers, die den bei dessen Anwendung zu erwartenden Return abschätzt, basiert auf den nach früheren Anwendungen beobachteten Returns und bezieht sich somit auf die Umgebungszustände, in denen diese Anwendungen erfolgten. Der zu erwartende Return hingegen hängt davon ab, zu welchen Umgebungszuständen der Klassifizierer aktuell passt. Nach einer Veränderung des Matchbereichs eines Klassifizierers ist daher dessen Vorhersage oft nicht mehr zutreffend. In gleicher Weise sind auch alle anderen Attribute betroffen, die direkt oder indirekt davon abhängen, zu welchen Umgebungszuständen ein Klassifizierer passt, das sind der Vorhersagefehler, die Fitness, die Genauigkeit, der Subsumtionsradius und die Distanzabschätzung.
Natürlich werden durch die von der Reinforcement-Komponente vorgenommenen Aktualisierungen auch die nicht mehr zutreffenden Attributwerte der von einer Modifikation der Population betroffenen Klassifizierer korrigiert. Da dies jedoch nicht instantan, sondern erst im Rahmen nachfolgender Anwendungen dieser Klassifizierer geschieht, hinken die Anpassungen der Attribute den Veränderungen der Matchbereiche hinterher. Die Attribute der HCS-Klassifizierer sind somit tendenziell umso weniger präzise und aussagekräftig, je öfter Veränderungen an der Population und damit an den Matchbereichen der Klassifizierer erfolgen. Nun ist es naheliegend, anzunehmen, dass nach einer kleinen Veränderung des Matchbereichs eines Klassifizierers dessen Attribute weniger stark angepasst werden müssen als nach einer großen. Modifikationen der Population sollten daher stets nur in möglichst geringem Umfang erfolgen: Erstens sollte der Netzlernmechanismus die Gewichtsvektoren der HCS-Klassifizierer entweder nur selten oder nur in kleinen Schritten, das heißt mit einer niedrigen Lernrate, anpassen. Zweitens sollte auch das Einfügen neuer Klassifizierer in die Population auf das nötigste beschränkt werden, dieser Forderung trägt das im folgenden Abschnitt beschriebene Verfahren zum

[5] Ferner gilt es auch für die hyperellipsoidalen und die durch konvexe Hüllen repräsentierten Bedingungen, auf die in Abschnitt 4.1 verwiesen wurde.

Einfügen von Klassifizierern Rechnung. Die dritte die Population verändernde Operation ist das Löschen von Klassifizierern. Da dieses automatisch erfolgt, wenn die zulässige Maximalgröße $N_{\max}$ der Population überschritten wird, kann seine Häufigkeit nicht direkt beeinflusst werden – sie sinkt jedoch automatisch, wenn seltener neue Klassifizierer in die Population eingefügt werden.
Zusätzlich zur Beschränkung von Häufigkeit und Stärke von Modifikationen der Population kann – im Sinne einer symptomatischen Behandlung – versucht werden, die Attribute der Klassifizierer unter Verwendung einer höheren Lernrate stärker anzupassen.

Sowohl das Beschränken von Veränderungen der Population als auch das stärkere Anpassen an aktuelle Beobachtungen haben sich in den mit dem HCS durchgeführten Experimenten als hilfreich erwiesen. Allerdings besitzen beide Ansätze auch das Potential, das Lernverhalten ihrerseits negativ zu beeinflussen, sodass sie nur mit entsprechender Vorsicht einzusetzen sind: Durch eine zu starke Einschränkung der Veränderungen an der Population, insbesondere durch eine zu seltene Ausführung des Genetischen Algorithmus[6], kann der Lernprozess unnötig in die Länge gezogen werden. Die Verwendung einer zu großen Lernrate kann zu sprunghaften Änderungen der Attributwerte führen und so die Entwicklung einer stabilen Population behindern.

4.2.2 Einfügen von Klassifizierern

Beim Einfügen eines Klassifizierers in die Population wird gemäß Algorithmus 4.1 vorgegangen: Zunächst wird geprüft, ob es sich bei dem einzufügenden Klassifizierer um eine Kopie eines bereits in der Population vorhandenen handelt. In diesem Fall wird lediglich dessen Vielfachheit erhöht. Andernfalls ist die Vorgehensweise komplizierter, da es nicht immer sinnvoll ist, einen „unbekannten" Klassifizierer in die Population aufzunehmen: Durch Aufnahme eines neuen Klassifizierers verlieren andere, die bereits in der Population enthalten sind und die gleiche Aktion vertreten, an Allgemeinheit. Das Einfügen von Klassifizierern in Bereichen des Zustandraumes, für die andere Klassifizierer schon genaue Vorhersagen machen, kann somit die Entwicklung genauer und zugleich genereller Klassifizierer behindern und sollte daher unterbleiben.
Um zu entscheiden, ob ein unbekannter Klassifizierer $\tilde{cl}$ der Population hinzugefügt werden soll, wird geprüft, ob es in der Population einen Klassifizierer cl mit der gleichen Aktion gibt, innerhalb dessen Subsumtionsradius der Gewichtsvektor von $\tilde{cl}$ liegt:

$$\left\| cl.\boldsymbol{w} - \tilde{cl}.\boldsymbol{w} \right\| \overset{!}{\leq} cl.r \tag{4.10}$$

Gibt es einen solchen, so läge nach dem Einfügen von $\tilde{cl}$ zumindest ein bedeutender Teil der Voronoizelle von $\tilde{cl}$ innerhalb des Subsumtionsradius von cl, also in einem Bereich des Zustandsraumes, für den cl eine genaue Returnvorhersage macht. Anstatt $\tilde{cl}$ in die Population einzufügen, wird in diesem Fall – was die Bezeichnung *Subsumtionsradius* motiviert – die Vielfachheit von cl erhöht. Liegt $\tilde{cl}$ innerhalb des Subsumtionsradius mehrerer Klassifizierer, so wird die Vielfachheit desjenigen erhöht, zu dem $\tilde{cl}$ den geringsten Abstand hat. Gibt es keinen Klassifizierer cl, für den die Bedingung (4.10) erfüllt ist, wird $\tilde{cl}$ in die Population aufgenommen.

[6] Neue Klassifizierer werden primär durch den Genetischen Algorithmus in die Population eingefügt.

1. Falls $cl.\boldsymbol{w} = \tilde{cl}.\boldsymbol{w}$ für einen Klassifizierer $cl \in P_{\tilde{cl}.action}$:

1.1. Setze: $cl.num \leftarrow cl.num + 1$

2. Andernfalls:

2.1. Initialisiere: $subsumer \leftarrow \perp$.

2.2. Überprüfe, ob es einen $\tilde{cl}$ subsumierenden Klassifizierer gibt, setze ggf.:

$$subsumer \leftarrow \quad \underset{\substack{cl \in P_a, \\ \left\| cl.\boldsymbol{w} - \tilde{cl}.\boldsymbol{w} \right\| \leq cl.r}}{\arg\min} \quad \left\| cl.\boldsymbol{w} - \tilde{cl}.\boldsymbol{w} \right\|$$

2.3. Falls $subsumer \neq \perp$:

2.3.1. Setze: $subsumer.num \leftarrow subsumer.num + 1$

2.4. Andernfalls:

2.4.1. Setze: $P \leftarrow P \cup \left\{ \tilde{cl} \right\}$

Algorithmus 4.1: Das Einfügen eines Klassifizierers $\tilde{cl}$ in die Population P.

4.2.3 Löschen von Klassifizierern

Ein Löschen von Klassifizierern erfolgt beim HCS – mit einer Ausnahme, auf die weiter unten eingegangen wird – nur dann, wenn die Anzahl der in der Population enthaltenen Klassifizierer die maximal erlaubte Populationsgröße $N_{\max}$ überschreitet. Ob das der Fall ist, wird nach jedem Einsatz des Covering-Operators und des Genetischen Algorithmus geprüft: Da nur durch diese beiden Discovery-Mechanismen Klassifizierer in die Population eingefügt werden, können auch nur sie eine Überschreitung der maximalen Populationsgröße $N_{\max}$ verursachen. Wird eine solche festgestellt, kommt Algorithmus 4.2 zum Einsatz, um die Populationsgröße wieder unter den Maximalwert zu senken:

1. Berechne die durchschnittliche Fitness der Population P:

$$\bar{f} = \frac{\sum_{cl \in P} cl.f \cdot cl.num}{\sum_{cl \in P} cl.num} \tag{4.11}$$

2. Berechne die Löschvoten für alle Klassifizierer der Population P:

$$vote(cl) = \begin{cases} \frac{\bar{f}}{cl.f} & \text{für } cl.exp > \Theta_{\text{DEL}} \wedge cl.f < \delta \cdot \bar{f} \\ 1 & \text{sonst} \end{cases} \tag{4.12}$$

3. Wähle durch Rouletterad-Selektion bezüglich der Löschvoten einen Klassifizierer $\tilde{cl}$ aus.

4. Falls $\tilde{cl}.num > 1$:

4.1. Verringere die Vielfachheit von $\tilde{cl}$ um 1:

$$\tilde{cl}.num \leftarrow \tilde{cl}.num - 1$$

5. Andernfalls:

5.1. Lösche alle von $\tilde{cl}$ ausgehenden Kanten.

5.2. Setze: $P \leftarrow P \setminus \left\{ \tilde{cl} \right\}$

Algorithmus 4.2: Das Löschen eines Klassifizierers bei Überschreitung der Maximalgröße der Population.

Zunächst werden die durchschnittliche Fitness (4.11) und die Löschvoten (4.12) aller in der Population P enthaltenen Klassifizierer berechnet, wobei anstelle der gewöhnlichen Fitness die geteilte Fitness (siehe Seite 89) verwendet werden kann. Durch eine auf die Löschvoten bezogene Rouletterad-Selektion wird dann ein Klassifizierer ausgewählt. Ist dessen Vielfachheit größer als 1, so wird diese um 1 verringert. Andernfalls werden zunächst alle Kanten, die den ausgewählten Klassifizierer als Endpunkt besitzen und anschließend dieser selbst gelöscht.
Das Löschvotum (4.12) eines Klassifizierers ist proportional seiner Vielfachheit, ferner ist die Wahrscheinlichkeit eines Klassifizierers, gelöscht zu werden, proportional dem Reziproken seines Fitnesswerts – dies aber nur, wenn dieser kleiner als ein Bruchteil δ der durchschnittlichen Fitness $\bar{f}$ der Population ist und sich zudem auf eine genügend große Erfahrung ($exp > \Theta_{\mathrm{DEL}}$) gründet. Mit dieser Form der Löschvotenberechnung werden – wie beim XCS – zwei Ziele verfolgt: Erstens soll die zur Verfügung stehende Zahl von Klassifizierern annähernd gleichmäßig auf die Makroklassifizierer verteilt werden, zweitens sollen Klassifizierer mit sehr niedriger Fitness aus der Population entfernt werden.

Wie bereits angekündigt, gibt es noch einen Sonderfall, in dem Klassifizierer unabhängig davon, ob die maximale Populationsgröße überschritten wurde, gelöscht werden: Klassifizierer, die nicht mehr durch Kanten mit anderen Klassifizierern verbunden sind, werden – ungeachtet ihrer Vielfachheit – vollständig aus der Population entfernt. Ausgenommen davon sind Klassifizierer, die nicht durch Kanten mit anderen verbunden sind, weil sie seit ihrer Aufnahme in die Population noch nicht als Gewinner in Erscheinung getreten sind. Wie beim Wachsenden Neuralen Gas altern die Kanten auch beim HCS, in jedem Lernschritt wird jedoch die Kante zwischen Gewinner und Zweitem erneuert. Ein Klassifizierer, der regelmäßig Gewinner oder Zweiter ist, wird daher nur selten alle seine Kanten verlieren und gelöscht werden. Betroffen sind von diesem Sonderfall also praktisch nur Klassifizierer, die nie oder nur extrem selten Gewinner oder Zweiter sind. Da die Erfahrung solcher eigentlich überflüssigen Klassifizierer nur langsam wächst, bleiben ihre Löschvoten lange klein, sodass das reguläre Löschverfahren sie nur mit geringer Wahrscheinlichkeit aus der Population entfernt. Ohne das Löschen kantenloser Klassifizierer würde die Beschränkung der Populationsgröße häufiger überschritten, sodass das reguläre Löschverfahren häufiger eingesetzt werden müsste und dementsprechend die Gefahr einer Löschung „guter“ Klassifizierer steigen würde.

4.3 Performance-Komponente

Die Aufgabe der Performance-Komponente besteht auch beim HCS in der Auswahl der in der Lernumgebung auszuführenden Aktionen. Hierzu wird in jedem Lernschritt zunächst ein Match-Set gebildet, das alle Klassifizierer enthält, deren Vorhersagen bei der anschließenden Berechnung der Systemvorhersagen zu berücksichtigen sind. Die Systemvorhersagen schätzen die bei Anwendung der verschiedenen Aktionen zu erwartenden Returns ab und bilden somit die Grundlage für die Wahl der auszuführenden Aktion. Die diese vertretenden, im Match-Set enthaltenen Klassifizierer bilden das Action-Set.
Die prinzipiellen Abläufe entsprechen somit denen beim XCS, auf einer weniger abstrakten Ebene hingegen zeigen sich deutliche Unterschiede: Dass es im Falle des HCS zu jedem Umgebungszustand pro Aktion stets höchstens einen passenden Klassifizierer gibt, bedingt eine gänzlich andere Form der Berechnung der Systemvorhersagen. Infolge dessen muss auch das Match-Set in anderer Weise als beim XCS gebildet werden.

4.3.1 Match-Set-Bildung

Die Performance-Komponenten sowohl des XCS wie auch des HCS bestimmen in jedem Lernschritt zunächst die Teilmenge der Population, die die für die Berechnung der Systemvorhersagen benötigten Klassifizierer enthält. Beim XCS sind dies gerade die zum jeweils aktuellen Umgebungszustand $s \in \mathcal{S}$ passenden Klassifizierer, was die Bezeichnung dieser Menge als ‚Match-Set' motiviert. Beim HCS hingegen sind, wie im Folgenden ausgeführt wird, in dieser Menge auch Klassifizierer enthalten, deren Bedingung von s nicht erfüllt wird. Dennoch soll diese Menge auch beim HCS als Match-Set bezeichnet werden, da ihre Funktion der des Match-Sets des XCS entspricht.

Vorüberlegungen zur Match-Set-Bildung

Würde sich das HCS bei der Berechnung der Systemvorhersagen nur auf die zum jeweils aktuellen Umgebungszustand passenden Klassifizierer stützen, wären die Systemvorhersage für die verschiedenen Aktionen offensichtlich durch die Vorhersagen der entsprechenden Gewinnerklassifizierer festlegt. Aufgrund der in Abschnitt 4.2 diskutierten Auswirkungen der Kontextsensitivität der HCS-Klassifizierer auf deren Attribute erscheint es jedoch wenig sinnvoll, die Systemvorhersagen – und damit letztlich die Aktionswahl – auf die Vorhersagen einzelner Klassifizierer zu stützen[7]. Neben den Gewinnern sollten daher noch weitere – dann zwangsläufig nicht zum aktuellen Umgebungszustand passende – Klassifizierer, respektive deren Vorhersagen, berücksichtigt und zu diesem Zweck ins Match-Set aufgenommen werden.

Dementsprechend ist zunächst zu überlegen, welche Klassifizierer zur Abschätzung eines Returns beitragen können, der bei Ausführung ihrer Aktion in einem Zustand, zu dem sie nicht passen, zu erwarten ist.
In Abschnitt 4.1 wurde dargelegt, dass ein Lernendes Klassifizierendes System umso besser für eine Lernumgebung geeignet ist, je genauer sich die in dieser Umgebung bestehenden Möglichkeiten zur Generalisierung durch dessen Klassifiziererbedingungen ausdrücken lassen. Die Entwicklung des HCS wurde durch den Wunsch nach einem System motiviert, das diesbezüglich flexibler ist als das XCS mit intervallbasierten Bedingungen. Beide Systemen machen jedoch stillschweigend eine Voraussetzung über die betrachteten Lernumgebungen und die in ihnen möglichen Verallgemeinerungen: Dass bei beiden Systemen eine Klassifiziererbedingung stets von allen Zuständen aus einer kontinuierlichen Teilmenge des Zustandsraums (Hyperquader beziehungsweise Voronoizelle) erfüllt wird, impliziert die Annahme, dass es zusammenhängende Bereiche des Zustandsraums gibt, in denen die Ausführung einer bestimmten Aktion zu (zumindest annähernd) gleichen Returns führt[8].
Im Hinblick auf die obige Fragestellung legt diese Annahme es nahe, bei der Berechnung der Systemvorhersagen für einen Zustand neben den Gewinnern für diesen Zustand auch Klassifizierer heranzuziehen, die zu ähnlichen Zuständen passen – Klassifizierer also, die „in der Nähe" des betrachteten Zustandes respektive der zugehörigen Gewinner liegen. Gestützt wird ein derartiges Vorgehen auch durch das folgende Argu-

[7] Dies gilt zumindest während des größten Teils des Lernprozesses des HCS. Nach ausreichend langem Lernen und Herausbildung einer im Wesentlichen stabilen Klassifiziererpopulation mag es sich ändern.

[8] Diese Annahme ist keineswegs zwingend, es ist leicht, eine Lernumgebung zu konstruieren, in der sie nicht erfüllt ist – etwa die folgende: Der Zustandsraum sei das Intervall $\mathcal{S} = [0, 1)$, es gebe zwei Aktionen a_1 und a_2, von denen in jedem Zustand nur eine richtig ist. Für deren Ausführung erhalte das Lernende Klassifizierende System die Belohnung $r = 1000$, für die falsche Aktion erhalte es $r = 0$. Für Zustände $s \in \mathcal{S} \cap \mathbb{Q}$ sei Aktion a_1 richtig, für alle anderen a_2.

ment: Die Notwendigkeit, bei der Berechnung der Systemvorhersagen nicht passende Klassifizierer einzubeziehen, wurde damit begründet, dass die Vorhersage eines Gewinnerklassifizierers infolge seiner Kontextsensitivität möglicherweise nicht zutrifft, wenn der betrachtete Lernumgebungszustand erst seit kurzem in seiner Voronoizelle liegt. In diesem Fall kann die Vorhersage des Klassifizierers, dessen Matchbereich der Zustand zuvor angehörte, aussagekräftiger sein. Dies wird in der Regel ein nahe dem Gewinner positionierter respektive diesem benachbarter Klassifizierer sein, da ja Veränderungen an der Population klein gehalten werden.

Die Bestimmung von Klassifizierern, die im obigen Sinne „in der Nähe" des betrachteten Zustandes respektive der zugehörigen Gewinner liegen, kann ausgehend von geometrischen Informationen, das heißt basierend auf Distanzen im Zustandsraum, vorgenommen werden. Alternativ ist es aber auch möglich, die Klassifiziererpopulation – wie bereits in Abschnitt 4.2.1 angekündigt – mit einer Topologie zu versehen und bei der Systemvorhersagenberechnung die Gewinnerklassifizierer sowie die ihnen bezüglich dieser Topologie benachbarten Klassifizierern zu berücksichtigen. Im Rahmen dieser Arbeit wurden beide Ansätze verfolgt, die entsprechenden Match-Set-Bildungsalgorithmen werden im Folgenden vorgestellt[9].

Topologie-basierte Match-Set-Bildung

Die Bildung eines Match-Sets auf Grundlage topologischer Information setzt natürlich voraus, dass auf der Klassifiziererpopulation respektive auf jeder der Teilpopulationen $P_a, a \in \mathcal{A}$, eine Topologie definiert ist. Diese kann, da sich die Population unter dem Einfluss der Discovery-Komponente erst nach und nach entwickelt, nicht a-priori festgelegt werden, sondern muss im Laufe des Lernprozesses erzeugt werden.
Dazu müssen zum einen Kanten zwischen als benachbart erkannten Klassifizierern eingefügt werden, zum anderen müssen nicht länger benötigte Kanten aber auch wieder gelöscht werden. Diese beiden Aufgaben übernimmt ein Topologielernverfahren, dass weitgehend dem Competitive Hebbian Learning des (Wachsenden) Neuralen Gases entspricht. Bevor näher auf dieses eingegangen wird, müssen zunächst noch einige mit den Kanten des HCS zusammenhängende Bezeichnungen eingeführt werden. Diese entsprechen den von den Selbstorganisierenden Karten her bekannten.

Zwei Klassifizierer cl_i und cl_k heißen *benachbart*, wenn sie durch eine Kante verbunden sind, diese wird mit $e_{i,k} = e_{k,i}$ bezeichnet. Für die Anzahl der Nachbarn eines Klassifizierers cl_i wird das Symbol N_i verwendet, die Nachbarn selbst werden mit $cl_{i,j},\ j = 1, 2, \ldots, N_i$ bezeichnet, sodass durch

$$\begin{aligned}\mathcal{N}_i &:= \{cl_{i,j} \,|j = 1, 2, \ldots, N_i\} \\ &= \{cl_k \in P_{cl_i.action} \,|e_{i,k} \in \mathcal{E}_{cl_i.action}\} \\ &= \{cl_k \in P \,|e_{i,k} \in \mathcal{E}\}\end{aligned}$$

die Menge der Nachbarn von cl_i gegeben ist.
Da das Topologielernen des HCS – wie das Competitive Hebbian Learning – nicht mehr benötigte Kanten mit Hilfe eines Alterungsmechanismus identifiziert, verfügen auch die

[9] Die distanzbasierte Variante der Match-Set-Bildung erfordert im Gegensatz zur Topologie-basierten eine explizite Angabe der Zahl der ins Match-Set aufzunehmenden Klassifizierer. Wird diese geeignet gewählt, unterscheidet sich die Leistung beider Varianten des HCS nur wenig, eine schlechte Wahl kann sich jedoch stark auswirken. Insofern ist die Topologie-basierte Variante leichter zu handhaben und wurde daher für alle finalen, in den Kapiteln 6 bis 8 noch zu diskutierenden Experimente verwendet.

Kanten des HCS über ein ihr Alter angebendes Attribut $age \in \mathbb{N}_0$. Um das Alter einer bestimmten Kante zu benennen, kommt die gleiche Indizierung zum Einsatz wie bei den Kanten selbst. Das Alter der Kante $e_{i,k}$ wird also mit $age_{i,k}$ bezeichnet. Ferner wird für die Kante zwischen einem Gewinner cl_w^a und dem zugehörigen Zweiten cl_s^a die Bezeichnung $e_{w,s}^a$ verwendet, deren Alter ist durch $age_{w,s}^a$ gegeben.

1. Initialisiere ein leeres Match-Set: $M = \emptyset$.

2. Für jede Aktion $a \in \mathcal{A}$:

2.1. Bestimme den dem aktuellen Umgebungszustand $s \in \mathcal{S}$ nächsten (cl_w^a) und zweitnächsten (cl_s^a) Klassifizierer mit Aktion a:

$$\|cl_w^a.\boldsymbol{w} - s\| \leq \|cl.\boldsymbol{w} - s\|\,,\ cl \in P_a$$

$$\|cl_s^a.\boldsymbol{w} - s\| \leq \|cl.\boldsymbol{w} - s\|\,,\ cl \in P_a \setminus \{cl_w^a\}$$

2.2. Erzeuge die Kante $e_{w,s}$ zwischen cl_w^a und cl_s^a, sofern sie nicht existiert:

$$\mathcal{E}_a \leftarrow \mathcal{E}_a \cup \left\{e_{w,s}^a\right\}$$

2.3. Setze: $age_{w,s}^a \leftarrow 0$.

2.4. Falls berechnete Klassifizierervorhersagen verwendet werden:

2.4.1. Erhöhe das Alter aller von cl_w^a ausgehenden Kanten um 1.

2.5. Andernfalls:

2.5.1. Erhöhe das Alter aller Kanten in $\mathcal{E}_a$ um 1.

3. Kopiere die Gewinner und alle ihre Nachbarn in das Match-Set M.

4. Solange nicht alle möglichen Aktionen zweimal in M vertreten sind, wiederhole:

4.1. **Ergänze das Match-Set durch Covering.**
(siehe Algorithmus 4.8)

4.2. **Lösche Klassifizierer aus P, falls durch das Covering die maximale Populationsgröße $N_{\max}$ überschritten wurde.**
(siehe Algorithmus 4.2)

Algorithmus 4.3: Die Bildung des Match-Sets unter Berücksichtigung von Nachbarschaften der Klassifizierer.

Wie bereits erwähnt, erfolgt das Topologielernen beim HCS als Bestandteil der topologiebasierten Variante der Match-Set-Bildung. Wie dem Algorithmus 4.3 zu entnehmen ist, werden dabei im Anschluss an die Initialisierung eines leeren Match-Sets nacheinander die Topologien aller Teilpopulationen $P_a, a \in \mathcal{A}$ angepasst (Schritt 2). Dazu werden für jede Aktion $a \in \mathcal{A}$ der Gewinner cl_w^a und der Zweite cl_s^a für den aktuellen Lernumgebungszustand $s \in \mathcal{S}$ bestimmt. Sofern die Kante $e_{w,s}^a$ bereits existiert, wird deren Alter auf 0 zurückgesetzt, andernfalls wird $e_{w,s}^a$ mit Alter $age_{w,s}^a = 0$ erzeugt und in die Kantenmenge $\mathcal{E}_a$ eingefügt. Sodann erfolgt die Alterung der Kanten. Werden berechnete Klassifizierervorhersagen verwendet, wird wie beim Competitive Hebbian Learning des (Wachsenden) Neuralen Gases das Alter aller Kanten, die den Gewinner cl_w^a als Endpunkt haben, um 1 erhöht; werden skalare Klassifizierervorhersagen benutzt, wird das Alter aller in $\mathcal{E}_a$ enthaltenen Kanten um 1 erhöht. Anschließend werden alle Kanten, die das zulässige Maximalalter $age_{\max} \in \mathbb{N}$ überschritten haben, gelöscht[10], womit die Anpassung der Topologie von P_a für den aktuellen Lernschritt beendet ist. Die Fallunterscheidung bei

[10] Ebenfalls gelöscht werden Klassifizierer aus P_a, die alle ihre Kanten „verloren“ haben.

der Alterung der Kanten ist dadurch begründet, dass bei Verwendung skalarer Vorhersagen der Positionsadaptationsmechanismus des HCS die anzupassenden Klassifizierer respektive deren Gewichtsvektoren unter bestimmten Bedingungen von den beobachteten Zuständen wegbewegt (siehe Abschnitt 4.5.3) und nicht – wie der des (Wachsenden) Neuralen Gases – nur zu den Eingaben hin. Somit können Klassifizierer aus dem Zustandsraum „hinausgeschoben" werden und treten dann (außer in Ausnahmefällen) nicht mehr als Gewinner oder Zweite auf. Würden nur die von Gewinnern ausgehenden Kanten gealtert, so blieben Kanten zwischen derartig positionierten Klassifizierern – und damit auch diese Klassifizierer selbst – sehr lange erhalten. Dies widerspricht jedoch der Grundidee des Topologielernens, die darin besteht, nur regelmäßig erneuerte Kanten zu behalten und so einen Subgraphen der Delaunay-Triangulierung der Klassifizierer zu erzeugen. Durch die beschriebene Modifikation des Alterungsmechanismus wird dies wieder ermöglicht.

Der Anpassung der Topologien der Teilpopulationen folgt im dritten Schritt des Algorithmus 4.3 die eigentliche Bildung des Match-Sets, das sich aus allen Gewinnern sowie deren Nachbarn zusammensetzt:

$$M = \bigcup_{a \in \mathcal{A}} (\{cl_w^a\} \cup \mathcal{N}_w^a) \tag{4.13}$$

Anschließend wird das Match-Set gegebenenfalls durch den Covering-Operator ergänzt. Anders als beim XCS, bei dem nur eine frei wählbare minimale Anzahl Θ_{MNA} von Aktionen im Match-Set vertreten sein muss, sollen beim HCS stets alle in der verwendeten Lernumgebung zulässigen Aktionen vertreten sein. Die Gründe hierfür werden bei der Diskussion des Covering-Operators in Abschnitt 4.5.1 dargelegt. Wird durch das Covering die maximal erlaubte Größe der Population überschritten, wird durch Löschen von Klassifizierern entsprechend dem Algorithmus 4.2 die Populationsgröße wieder unter den Maximalwert gesenkt.

Distanz-basierte Match-Set-Bildung

Algorithmus 4.4 verdeutlicht den Ablauf der distanzbasierten Variante der Match-Set-Bildung. Vorbereitend wird zunächst ein leeres Match-Set initialisiert und für jeden in der Population enthaltenen Klassifizierer der euklidische Abstand zum aktuellen Zustand $s \in \mathcal{S}$ der Lernumgebung berechnet. Im dritten Schritt des Algorithmus werden dann aus jeder Teilpopulation $P_a, a \in \mathcal{A}$ die $2 \leq k$ Klassifizierer mit den niedrigsten euklidischen Abständen zu s in das Match-Set übernommen. In zwei Sonderfällen weicht dabei die Anzahl übernommener Klassifizierer von k ab:

- Enthält eine Teilpopulation P_a weniger als k Klassifizierer, so werden nur diese übernommen.
- Enthält P_a nach Übernahme von k Klassifizierern weitere Klassifizierer, deren euklidischer Abstand vom Zustand ebenso groß ist, wie der des zuletzt übernommenen Klassifizierers, werden diese zusätzlich ins Match-Set eingefügt, da von mehreren Klassifizierern mit gleichem Abstand zum Zustand keiner bei der Match-Set-Bildung bevorzugt werden sollte.

Schließlich wird das Match-Set auch hier durch Covering ergänzt. Dabei wird wiederum sichergestellt, dass jede Aktion durch mindestens zwei Klassifizierer in der Population vertreten ist.

1. Initialisiere ein leeres Match-Set: $M = \emptyset$.

2. Für alle Klassifizierer $cl \in P$:

  2.1. Berechne den euklidischen Abstand $\|cl.\boldsymbol{w} - s\|$ zum aktuellen Zustand $s \in \mathcal{S}$.

3. Für jede Aktion $a \in \mathcal{A}$:

  3.1. Kopiere die $k \in \mathbb{N}$ Klassifizierer mit den niedrigsten Abständen aus P_a nach M.

4. Solange nicht alle möglichen Aktionen zweimal in M vertreten sind, wiederhole:

  4.1. Ergänze das Match-Set durch Covering.
  (siehe Algorithmus 4.8)

  4.2. Lösche Klassifizierer aus P, falls durch das Covering die maximale Populationsgröße $N_{\max}$ überschritten wurde.
  (siehe Algorithmus 4.2)

Algorithmus 4.4: Die Bildung des Match-Sets unter Berücksichtigung der Ähnlichkeit im Zustandsraum.

Im Anschluss an das Covering wird überprüft, ob die maximal erlaubte Größe der Population überschritten wurde. Ist das der Fall, wird die Populationsgröße unter Verwendung des Löschalgorithmus 4.2 wieder unter den Maximalwert gebracht. Dabei unterbleibt, da die mit der distanzbasierten Match-Set-Bildung arbeitende Variante des HCS keine Kanten verwendet, selbstverständlich das Entfernen von Klassifizierern, die nicht durch Kanten mit anderen verbunden sind.

Nachdem auf eine der beiden beschriebenen Weisen ein Match-Set gebildet wurde, müssen ausgehend von den Vorhersagen der in diesem enthaltenen Klassifizierer die Systemvorhersagen für den jeweils aktuellen Zustand berechnet werden. Da in jede Systemvorhersage nur die Vorhersagen von Klassifizierern mit der entsprechenden Aktion eingehen, ist es sinnvoll – analog zur Teilpopulation P_a – das *Teil-Match-Set*

$$M_a := \{\, cl \in M \,|\, cl.action = a \,\} \tag{4.14}$$

zu definieren und – da dem Gewinner eine besondere Rolle zukommt – ferner die Bezeichnung

$$M_a{}^* := M_a \setminus \{\, cl_w^a \,\} \tag{4.15}$$

für die Menge der außer dem Gewinner cl_w^a in M_a enthaltenen Klassifizierer einzuführen[11].

4.3.2 Berechnung der Systemvorhersagen

Hier ist zuerst die Frage zu beantworten, wie genau aus den Vorhersagen der im Match-Set enthaltenen Klassifizierer die Systemvorhersagen zu berechnen sind.

[11] Im Falle der topologischen Match-Set-Bildung entspricht $M_a{}^*$ der Menge $\mathcal{N}_w^a$ der Nachbarn des Gewinners cl_w^a.

Vorüberlegungen zur Systemvorhersagenberechnung

Zunächst ist festzustellen, dass es nicht sinnvoll ist, die Berechnungsmethode des XCS auf das HCS zu übertragen, also die Systemvorhersage für eine Aktion a als Fitness-gewichtetes Mittel der Vorhersagen aller im Teil-Match-Set M_a enthaltenen Klassifizierer zu berechnen: Beim XCS ist dieses Verfahren angebracht, es berücksichtigt zum einen, dass alle in M_a enthaltenen Klassifizierer[12] zum betrachteten Umgebungszustand s passen und in diesem Sinne gleichberechtigt sind (Mittelung) und zum anderen, dass sie in direkter Konkurrenz zueinander stehen und sich hinsichtlich der Systemvorhersagenberechnung nur durch die – sich in ihrer Fitness widerspiegelnde – Genauigkeit ihrer Vorhersagen voneinander abheben (Gewichtung mit der Fitness).
Demgegenüber enthält das Teil-Match-Set M_a beim HCS ein ausgezeichnetes Element – den Gewinner cl_w^a, dem als dem einzigen zu s passendem Klassifizierer mit Aktion a besonderes Gewicht zukommen sollte; seine Vorhersage sollte den Ausgangspunkt für die Berechnung der zugehörigen Systemvorhersage bilden. Ferner haben die in M_a enthaltenen Klassifizierer beim HCS völlig unterschiedliche Matchbereiche; somit stehen sie erstens nicht in direkter Konkurrenz zueinander und zweitens beziehen sich ihre Genauigkeits- respektive Fitnesswerte auf unterschiedliche Bereiche des Zustandsraums. Beides spricht dagegen, eine Gewichtung mit der Fitness vorzunehmen[13].

Die Einbeziehung von den Gewinnern ähnlichen Klassifizierern bei der Berechnung der Systemvorhersagen des HCS wurde oben unter anderem damit begründet, dass ein erst seit kurzem im Matchbereich eines Gewinners enthaltener Zustand s zuvor wahrscheinlich im Matchbereich eines ähnlichen Klassifizierers gelegen hat, sodass dessen Vorhersage möglicherweise eine bessere Abschätzung des zu erwartenden Returns liefert als die des Gewinners. Bei dieser Argumentation wurde vorausgesetzt, dass sich die Matchbereiche durch Veränderungen an der Population stets nur leicht verändern. Unter dieser Voraussetzung ist es auch gerechtfertigt, anzunehmen, dass Zustände umso seltener „den Klassifizierer wechseln“, je näher sie an dessen Gewichtsvektor liegen. Ferner ist es plausibel, davon auszugehen, dass ein vom Gewinner aus gesehen in Richtung des Zustandes liegender Klassifizierer eher schon zu diesem gepasst hat als ein in einer anderen Richtung gelegener Klassifizierer. Somit können die beiden folgenden Anforderungen an die Berechnung der Systemvorhersagen formuliert werden:

(i) Es sollten vor allem jene ähnlichen Klassifizierer betrachtet werden, die vom Gewinner aus gesehen in Richtung des Zustandes liegen.

(ii) Die Vorhersagen der nicht-passenden Klassifizierer sollten umso stärker einfließen, je weiter der Zustand vom Gewinner entfernt ist.

Hinzu kommt der bereits oben erwähnte Punkt:

(iii) Den Gewinnern sollte besonderes Gewicht beigemessen werden, indem ihre Vorhersagen als Ausgangspunkt der Systemvorhersagen-Berechnung verwendet werden.

Neben der Sonderstellung der Gewinner spricht hierfür noch ein weiteres Argument: Die Einbeziehung der Vorhersagen nicht-passender Klassifizierer in die Berechnung der Systemvorhersagen steht in einem gewissen Widerspruch zum Wunsch nach der Evolution

[12] Die Definition (4.14) von M_a bezieht sich nur auf das HCS, kann aber offenbar ganz analog auf das XCS übertragen werden.

[13] Eine Berücksichtigung der Fitness wäre allenfalls in der Form denkbar, dass die Vorhersagen von Klassifizierern mit zu niedriger Fitness gar nicht in die Systemvorhersagen einfließen. Diese Idee wird hier jedoch nicht weiter verfolgt.

maximal genereller Klassifizierer: Ein solcher würde im Idealfall einen der angenommenen Bereiche, in denen eine Aktion zu gleichen Returns führt, komplett abdecken. Dann würde die Berücksichtigung ähnlicher Klassifizierer die Genauigkeit der Systemvorhersagen für Zustände aus diesem Bereich offenbar eher negativ beeinflussen und es wäre angebracht, in diesem Spezialfall die Vorhersage des maximal generellen Klassifizierers doch direkt als Systemvorhersage zu verwenden. Dies ist jedoch nicht umsetzbar, da es im Allgemeinen keine Möglichkeit gibt, festzustellen, ob ein Klassifizierer (nahezu) maximal generell ist oder nicht. Durch eine besondere Gewichtung der Vorhersagen der Gewinner wird dieses Problem zwar nicht gelöst, aber doch entschärft. Das im Folgenden beschriebene Berechnungsverfahren setzt die drei genannten Anforderungen um.

Algorithmus zur Systemvorhersagenberechnung

Die Grundidee der Systemvorhersagenberechnung des HCS besteht darin, für jede Aktion $a \in \mathcal{A}$ zunächst den aktuellen Umgebungszustand $s \in \mathcal{S}$ als Linearkombination

$$s = cl_w^a.\boldsymbol{w} + \sum_{cl \in M_a{}^*} c_{cl} \cdot l_{cl} \tag{4.16}$$

aus dem Gewichtsvektor des Gewinners cl_w^a und einigen, von diesem zu den anderen im Teil-Match-Set M_a enthaltenen Klassifizierern zeigenden, normierten Differenzvektoren

$$l_{cl} := \frac{(cl.\boldsymbol{w} - cl_w^a.\boldsymbol{w})}{\|cl.\boldsymbol{w} - cl_w^a.\boldsymbol{w}\|} \tag{4.17}$$

auszudrücken respektive zu approximieren und anschließend mit den dabei ermittelten Koeffizienten c_{cl} analog die Systemvorhersage PA_a zu interpolieren:

$$PA_a = cl_w^a.p + \sum_{cl \in M_a{}^*} c_{cl} \cdot \frac{(cl.p - cl_w^a.p)}{\|cl.\boldsymbol{w} - cl_w^a.\boldsymbol{w}\|} \tag{4.18}$$

Die zentrale Aufgabe ist somit die geeignete Bestimmung der Koeffizienten c_{cl} in (4.16). Entsprechend der oben formulierten Anforderung (i) sollen in die Systemvorhersage PA_a nur Vorhersagen von Klassifizierern eingehen, die vom Gewinner cl_w^a aus gesehen „in Richtung“ des Zustandes s liegen. Darunter sind Klassifizierer zu verstehen, für die die Differenzvektoren (4.17) mit dem vom Gewinner zum Zustand s weisenden Vektor

$$s_{\text{rel}} := s - cl_w^a.\boldsymbol{w} \tag{4.19}$$

einen Winkel von weniger als 90° einschließen. Daher werden für alle in M_a enthaltenen Klassifizierer cl, für die das Skalarprodukt $l_{cl} \cdot s_{\text{rel}}$ keinen positiven Wert annimmt, die zugehörigen Koeffizienten c_{cl} von vorne herein gleich Null gesetzt.

Da der Zustand s möglichst genau approximiert werden soll, kann das Problem der Bestimmung der verbleibenden Koeffizienten als Ausgleichsproblem [z.B.: Stoer 1994]

$$\min_{c \in \mathbb{R}^k} \|L \cdot c - s_{\text{rel}}\|^2 \tag{4.20}$$

formuliert werden, wobei es sich bei den Einträgen des (Koeffizienten-)Vektors $c \in \mathbb{R}^k$ um die noch zu bestimmenden Koeffizienten und bei den Spalten der Matrix $L \in \mathbb{R}^{n \times k}$ um die zugehörigen Differenzvektoren (4.17) handelt[14].

[14] Zur Erinnerung: n ist die Dimension des Zustandsraumes $\mathcal{S}$.

Dieses Minimierungsproblem hat eine eindeutige Lösung genau dann, wenn die Matrix L vollen Spaltenrang hat ($Rang(L) = k$) – wenn also die in L berücksichtigten Differenzvektoren linear unabhängig sind. Andernfalls hat (4.20) unendlich viele Lösungen. Durch die zusätzliche Forderung, dass die gesuchte Lösung unter diesen minimale euklidische Norm aufweist, kann aber auch in diesem Fall Eindeutigkeit erzwungen werden. Diese Forderung ist sinnvoll: Je (betragsmäßig) größer die Koeffizienten, desto stärker wird bei der Berechnung (4.18) der Systemvorhersagen extrapoliert; kleine Koeffizienten sind somit zu bevorzugen[15].
Die im Rahmen dieser Arbeit erstellte Implementation des HCS bestimmt aus Gründen der numerischen Stabilität die Koeffizienten, indem an Stelle von (4.20) das – stets eindeutig lösbare – Tychonoff-regularisierte Ausgleichsproblems

$$\min_{c \in \mathbb{R}^k} \left[\|L \cdot c - s_{\text{rel}}\|^2 + \mu \|c\|^2 \right], \; 0 < \mu < 1 \tag{4.21}$$

betrachtet wird. Für kleine μ ergibt sich dabei näherungsweise die Minimum-Norm-Lösung von (4.16)[16]. Der zweite Term in (4.21) kann als Strafterm für betragsmäßig große Koeffizienten interpretiert werden.
Obwohl bei dieser Art der Berechnung der Koeffizienten die Norm des Koeffizientenvektors c automatisch minimiert wird, kann es dennoch vorkommen, dass einzelne Koeffizienten betragsmäßig sehr groß werden – insbesondere dann, wenn zwei Differenzvektoren (nahezu) kollinear sind[17]. Große Koeffizienten bewirken jedoch eine starke Extrapolation bei der Berechnung (4.18) der Systemvorhersagen. Um dies zu verhindern, wird nach der Koeffizientenberechnung stets überprüft, ob die folgende Bedingung erfüllt ist:

$$|c_{cl}| \leq \|cl.\boldsymbol{w} - cl^a_w.\boldsymbol{w}\|, cl \in M_a{}^* \tag{4.22}$$

Ist dies nicht der Fall, wird ein Differenzvektor – eine Spalte von L – entfernt und der zugehörige Koeffizient gleich 0 gesetzt. Die Koeffizienten der verbleibenden Differenzvektoren werden als Lösung des derart reduzierten Ausgleichsproblems neu bestimmt. Dies wird solange wiederholt, bis die Bedingung (4.22) erfüllt ist. Anschaulich bedeutet dies, dass bei der Approximation (4.16) des Zustandes s keine Vektoren benutzt werden, die länger sind als die entsprechenden (unnormierten) Differenzvektoren.
Bei der Bestimmung der zu entfernenden Differenzvektoren kommt die folgende einfache Heuristik zur Anwendung:

- Zunächst werden die beiden „linear abhängigsten" Spalten l_i und l_j von L bestimmt, das heißt die beiden Spalten mit maximalem Skalarprodukt $l_i \cdot l_j$.
- Von diesen beiden wird die mit dem kleineren Skalarprodukt $l_k \cdot s_{\text{rel}}$ entfernt, also der Differenzvektor, der den größeren Winkel mit s_{rel} bildet und somit „weniger" in Richtung von s zeigt.

Nach der endgültigen Bestimmung der Koeffizienten c_{cl} wird die Systemvorhersage PA_a gemäß (4.18) berechnet. Das beschriebene Vorgehen wird für alle Aktionen $a \in \mathcal{A}$ durchgeführt, eine Ausnahme hiervon wird nur dann gemacht, wenn der Vorhersagefehler

[15] Die Zusatzforderung nach Minimalität der Norm des Koeffizientenvektors motiviert auch die Verwendung normierter Differenzvektoren in (4.16), da unter dieser Forderung die gefundene Lösung offensichtlich abhängig von der Skalierung der – durch die Differenzvektoren bestimmten – Spalten von L. ist

[16] Die Verwendung dieser auch als *Methode der regularisierten kleinsten Quadrate* oder *Ridge-Regression* bekannten Vorgehensweise wird in Anhang A motiviert.

[17] Dies kann beispielsweise dann eintreten, wenn ein Match-Set einen durch intermediäres Crossing-Over erzeugten Klassifizierer und dessen Eltern enthält.

eines Gewinners cl_w^a kleiner ist als eine als Parameter des HCS einzustellende Fehlerschranke ε_p. In diesem Fall wird die Vorhersage dieses Gewinners als zuverlässig genug angesehen, um direkt als Systemvorhersage für die Aktion a zu dienen. Algorithmus 4.5 fasst das Vorgehen bei der Berechnung der Systemvorhersage für eine Aktion $a \in \mathcal{A}$ zusammen.

1. Initialisiere die Systemvorhersage für a als undefiniert: $PA_a = \perp$.

2. Falls der Vorhersagefehler des Gewinners cl_w^a unterhalb der Fehlerschranke ε_p liegt ($cl_w^a.\varepsilon \leq \varepsilon_p$):

2.1. Setze:

$$PA_a \leftarrow cl_w^a.p$$

3. Andernfalls:

3.1. Ermittle die Menge

$$M_a{}^+ = \{cl \in M_a{}^* \mid (cl.\boldsymbol{w} - cl_w^a.\boldsymbol{w}) \cdot s_{\text{rel}} > 0\}\,.$$

und setze

$$c_{cl} \leftarrow 0, cl \in M_a{}^* \setminus M_a{}^+.$$

3.2. Solange nicht $|c_{cl}| \leq \|cl.\boldsymbol{w} - cl_w^a.\boldsymbol{w}\|$ für alle $cl \in M_a{}^+$ gilt, wiederhole:

3.2.1. Stelle den Zustand s als Linearkombination des Gewichtsvektors $cl_w^a.\boldsymbol{w}$ des Gewinners und der Differenzvektoren $l_{cl} := \frac{(cl.\boldsymbol{w} - clwa.\boldsymbol{w})}{\|cl.\boldsymbol{w} - cl_w^a.\boldsymbol{w}\|}$, $cl \in M_a{}^+$ dar:

$$s = cl_w^a.\boldsymbol{w} + \sum_{cl \in M_a{}^+} c_{cl} \cdot l_{cl}$$

3.2.2. Falls $|c_{cl}| > \|cl.\boldsymbol{w} - cl_w^a.\boldsymbol{w}\|$ für (mindestens) ein $cl \in M_a{}^+$:

3.2.2.1. Bestimme $cl_i, cl_j \in M_a{}^+, i \neq j$ so, dass:

$$l_{cl_i} \cdot l_{cl_j} \geq l_{cl_1} \cdot l_{cl_2} \forall cl_1, cl_2 \in M_a{}^+.$$

3.2.2.2. Für $k := \arg\max_{k \in \{i,j\}} \{l_{cl_k} \cdot s_{\text{rel}}\}$ setze:

$$M_a{}^+ \leftarrow M_a{}^+ \setminus \{cl_k\}$$

und

$$c_{cl_k} \leftarrow 0$$

3.3. Setze:

$$PA_a \leftarrow cl_w^a.p + \sum_{cl \in M_a{}^*} c_{cl} \cdot \frac{(cl.p - cl_w^a.p)}{\|cl.\boldsymbol{w} - cl_w^a.\boldsymbol{w}\|}$$

Algorithmus 4.5: Die Berechnung der Systemvorhersage für eine Aktion $a \in \mathcal{A}$.

4.3.3 Aktionswahl und Bildung des Action-Sets

Ihrer beiden verbleibenden Aufgaben, der Auswahl der auszuführenden Aktion und der Bildung des Action-Sets, entledigt sich die Performance-Komponente des HCS – wie ihr Gegenstück beim XCS – auf die durch Algorithmus 4.6 beschriebene Weise:

1. Wähle die auszuführende Aktion $\tilde{a} \in \mathcal{A}$:
Die Wahrscheinlichkeit für die Wahl einer Aktion $a \in \mathcal{A}$ ist gegeben durch:

$$\mathrm{P}\left(\tilde{a} = a\right) = \begin{cases} 1 - \frac{m-1}{m} \cdot p_{\text{explr}} & \text{für } a = \arg\max_{a' \in \mathcal{A}} PA_{a'} \\ \frac{p_{\text{explr}}}{m} & \text{sonst} \end{cases} \tag{4.23}$$

2. Initialisiere: $A = \emptyset$.

3. Kopiere alle in M enthaltenen Klassifizierer mit der Aktion $\tilde{a}$ ins Action-Set A:

$$A := M_{\tilde{a}} \tag{4.24}$$

Algorithmus 4.6: Die Wahl einer Aktion und die Bildung des Action-Sets.

Die Aktion, die das HCS in der Lernumgebung ausführen wird, wird ε-greedy gewählt:

- Mit einer Wahrscheinlichkeit $0 \leq p_{\text{explr}} \leq 1$ wird rein zufällig eine der im Match-Set vertretenen Aktionen ausgewählt.
- Mit Wahrscheinlichkeit $1 - p_{\text{explr}}$ wird die beste Aktion, also die mit der höchsten Systemvorhersage, selektiert.

Insgesamt ist somit die Wahrscheinlichkeit für die Auswahl einer bestimmten Aktion $a \in \mathcal{A}$ durch 4.23 gegeben. Das gerade beschriebene Vorgehen umfasst als Spezialfälle die rein explorative Aktionswahl ($p_{\text{explr}} = 1$) sowie eine durch die Systemvorhersagen determinierte Selektion der auszuführenden Aktion ($p_{\text{explr}} = 0$). Andere als das geschilderte Auswahlverfahren werden in dieser Arbeit nicht betrachtet, könnten aber prinzipiell in das HCS integriert werden. Denkbar wäre beispielsweise die Auswahl einer Aktion durch eine auf die Systemvorhersagen bezogene Rouletterad-Selektion.
Nach der Aktionswahl formt das HCS aus den die ausgewählte Aktion $\tilde{a}$ vertretenden Klassifizierer des Match-Sets das Action-Set, das also dem Teil-Match-Set $M_{\tilde{a}}$ entspricht. Das Action-Set des HCS ist von deutlich anderer Art als das des XCS: Letzteres kann als Menge gleichberechtigter, simultan aktivierter Klassifizierer angesehen werden, da alle seine Elemente zu dem zuvor wahrgenommenen Umgebungszustand $s \in \mathcal{S}$ passen und die gewählte Aktion vertreten. Beim HCS hingegen enthält das Action-Set genau einen zu s passenden Klassifizierer – den Gewinner $cl_w^{\tilde{a}}$. Dieser ist somit auch der einzige im betreffenden Lernschritt aktive Klassifizierer. Aus diesem Grund werden auch nicht alle in den Action-Sets des HCS enthaltenen Klassifizierer von der Reinforcement-Komponente aktualisiert, sondern nur die in ihnen enthaltenen Gewinner. Somit dienen die Action-Sets des HCS primär als Menge potentieller Elternindividuen für den Genetischen Algorithmus.

4.4 Reinforcement-Komponente

Die Reinforcement-Komponente des HCS übernimmt – wie die des XCS – die Aufgabe, die Attribute der aktivierten Klassifizierer aufgrund der Erfahrungen, die das System sammelt, anzupassen; den Ausgangspunkt hierfür bilden die Returns, die das System beobachtet. Der wesentliche Unterschied zum XCS ist darin zu sehen, dass diese Anpassungen nicht bei allen Klassifizierern vorgenommen werden, die einem Action-Set angehören, sondern nur bei den in einem solchen enthaltenen Gewinnern:

Zwar kann, wie in 4.3 erläutert, die Vorhersage eines HCS-Klassifizierers unter gewissen Umständen sinnvolle Informationen über die zu erwartenden Returns für Zustände, zu denen der Klassifizierer nicht passt, liefern, sodass es sinnvoll ist, nicht nur passende Klassifizierer ins Match-Set aufzunehmen und deren Vorhersagen bei der Berechnung der Systemvorhersagen zu berücksichtigen. Umgekehrt beziehen sich die tatsächlich beobachteten Returns jedoch stets auf die Ausführung einer bestimmten Aktion in einem bestimmten Zustand der Lernumgebung und machen im Allgemeinen keinerlei Aussage über die in anderen Zuständen zu erwartenden Returns. Die Aktualisierung der Attribute eines Klassifizierers auf Grundlage eines beobachteten Returns $\mathcal{P}$ ist daher nur dann sinnvoll, wenn dieser zu dem entsprechenden Zustand passt (und die Aktion vertritt, die vor Beobachtung des Returns ausgeführt wurde). Diese Bedingung wird beim HCS nur von den in den Action-Sets enthaltenen Gewinnern erfüllt.

Das Vorgehen bei der Aktualisierung der Klassifiziererattribute ist dem beim XCS sehr ähnlich: Außer zu Beginn einer neuen Lernepisode werden in jedem Lernschritt t zunächst die Attribute des im Action-Set A_{t-1} des vorangegangenen Lernschrittes enthaltenen Gewinners $cl^a_{w(s_{t-1})}$ aktualisiert. Die Returnabschätzung $\mathcal{P}$ wird dabei wie beim XCS gemäß (2.25) als Summe aus der Belohnung r_t des Zeitschrittes $t-1$ und dem diskontierten Maximum der aktuellen Systemvorhersagen gebildet. Falls das Ende einer Lernepisode erreicht ist, werden anschließend auch die Attribute des im aktuellen Action-Set A_t enthaltenen Gewinners angepasst, wobei der zu erwartende Return durch die aktuelle Belohnung geschätzt, also $\mathcal{P} = r_{t+1}$ gesetzt wird. Die Aktualisierung der Attribute des Gewinners erfolgt jeweils in der durch den Algorithmus 4.7 beschriebenen Weise.

Bei der Anpassung von Erfahrung, Vorhersage und Vorhersagefehler in den ersten drei Schritten dieses Algorithmus wird genau wie beim XCS vorgegangen: Zunächst wird des Gewinners Erfahrungswert exp um 1 erhöht. Ist der inkrementierte Erfahrungswert dann kleiner als der Kehrwert der Lernrate β, gilt der Klassifizierer als unerfahren und seine Vorhersage p sowie sein Vorhersagefehler ε werden mit der Moyenne-Adaptive-Modifiée-Technik angepasst. Andernfalls folgt die Anpassung einer Widrow-Hoff-Delta-Regel. Dies bedeutet insbesondere, dass der eigentliche Reinforcement-Schritt – die Anpassung der Vorhersage p – im Falle erfahrener Klassifizierer auch beim HCS dem Muster des Q-Learning entspricht.
Die Genauigkeit κ eines HCS-Klassifizierers wird ebenfalls in der gleichen Weise wie beim XCS berechnet (Schritt 4 des Algorithmus 4.7): Sie wird gleich 1 gesetzt, wenn der Vorhersagefehler des Klassifizierers unterhalb der Fehlerschranke ε_0 liegt, andernfalls ist sie durch eine negative Potenz des Quotienten aus dem Vorhersagefehler ε des Klassifizierers und der Fehlerschranke ε_0 gegeben.
Die Berechnung des neuen Fitnesswertes des zu aktualisierenden Klassifizierers (Schritt 5 des Algorithmus 4.7) unterscheidet sich in mehreren Punkten von der beim XCS: Erstens: Die Fitnessberechnung des XCS bezieht sich auf die relative Genauigkeit der zu aktualisierenden Klassifizierer, das ist deren Genauigkeit κ dividiert durch die Summe der Genauigkeiten aller – mit ihrer Vielfachheit gezählten – Angehörigen des Action-Sets, als dessen Teile die betrachteten Klassifizierer aktualisiert werden. Beim XCS ist dies sinnvoll, da dort alle Klassifizierer eines Action-Sets zumindest teilweise zu den gleichen Zuständen passen und – natürlich – die gleiche Aktion vertreten, somit also direkte Konkurrenten darstellen. Beim HCS hingegen gibt es keine derartige direkte Konkurrenz, da sich die Matchbereiche von HCS-Klassifizierern mit gleicher Aktion nie überschneiden. Die Betrachtung einer relativen Genauigkeit erübrigt sich somit, in die Fitnessberechnung geht die absolute Genauigkeit κ direkt ein.

1. Inkrementiere den Erfahrungswert von cl_w^a:

$$exp \leftarrow exp + 1 \tag{4.25}$$

2. Aktualisiere die Vorhersage p von cl_w^a:

$$p \leftarrow \begin{cases} p + (\mathcal{P} - p)/exp & \text{, falls } exp < 1/\beta \\ p + \beta \cdot (\mathcal{P} - p) & \text{, falls } exp \geq 1/\beta \end{cases} \tag{4.26}$$

3. Aktualisiere den Vorhersagefehler ε von cl_w^a:

$$\varepsilon \leftarrow \begin{cases} \varepsilon + (|\mathcal{P} - p| - \varepsilon)/exp & \text{, falls } exp < 1/\beta \\ \varepsilon + \beta \cdot (|\mathcal{P} - p| - \varepsilon) & \text{, falls } exp \geq 1/\beta \end{cases} \tag{4.27}$$

4. Aktualisiere die Genauigkeit κ von cl_w^a:

$$\kappa = \begin{cases} 1 & \text{, falls } \varepsilon < \varepsilon_0 \\ \alpha \cdot \left(\frac{\varepsilon}{\varepsilon_0}\right)^{-\nu} & \text{, falls } \varepsilon \geq \varepsilon_0 \end{cases} \tag{4.28}$$

5. Aktualisiere die Fitness f von cl_w^a:

$$f \leftarrow \begin{cases} f + (\kappa \cdot \frac{r}{avgd} - f)/exp & \text{, falls } exp < 1/\beta \\ f + \beta \cdot (\kappa \cdot \frac{r}{avgd} - f) & \text{, falls } exp \geq 1/\beta \end{cases} \tag{4.29}$$

6. Aktualisiere die Distanzabschätzung $avgd$ von cl_w^a:

$$avgd \leftarrow avgd + \beta_r \cdot (\|cl_w^a - s\| - avgd) \tag{4.30}$$

7. Wenn $cl_w^a.exp > 1$, aktualisiere den Radius r von cl_w^a:

$$r \leftarrow \begin{cases} r + \beta_r \cdot (\|cl_w^a - s\| - r) & \text{, falls } |p - \mathcal{P}| < \varepsilon \wedge \|cl_w^a - s_t\| > r \\ r + \beta_r \cdot (\|cl_w^a - s\| - r) & \text{, falls } |p - \mathcal{P}| \geq \varepsilon \wedge \|cl_w^a - s_t\| < r \end{cases} \tag{4.31}$$

Algorithmus 4.7: Die Aktualisierung des in einem Action-Set A enthaltenen Gewinners cl_w^a für den Zustand s unter Berücksichtigung der Returnabschätzung $\mathcal{P}$.

Zweitens geht neben κ auch der in Abschnitt 4.2 als weiteres Kriterium für die Genauigkeit eines Klassifizierers eingeführte Quotient aus dem Subsumtionsradius und der Distanzabschätzung in die Fitnessberechnung ein – die Fitness von HCS-Klassifizierern wird an das Produkt aus diesem Quotienten und der Genauigkeit κ angepasst.
Drittens wird – anders als beim XCS, das die Fitness stets entsprechend einer Widrow-Hoff-Delta-Regel anpasst – bei der Fitnessaktualisierung des HCS zwischen erfahrenen und unerfahrenen Klassifizierern unterschieden. Die Aktualisierungsvorschrift für die erstgenannten hat auch beim HCS die Form einer Widrow-Hoff-Delta-Regel, die letztgenannten werden unter Verwendung der Moyenne-Adaptive-Modifiée-Technik angepasst, um den Einfluss initialer Fitness-Werte zu verringern.

Zuletzt werden in den Schritten 6 und 7 des Algorithmus die Distanzabschätzung $avgd$ und der Subsumtionsradius r aktualisiert. Die Anpassung beider Attribute folgt einer Widrow-Hoff-Delta-Regel. Während jedoch die Distanzabschätzung immer aktualisiert wird, geschieht dies beim Subsumtionsradius nur, wenn der aktuell beobachtete Return darauf hinweist, dass der aktuelle Wert des Subsumtionsradius zu groß oder zu klein ist: Da der Subsumtionsradius eines Klassifizierers der Abschätzung der Größe eines um dessen Gewichtsvektor zentrierten Bereiches dient, in dem die Klassifizierervorhersage um weniger als die Fehlerschranke ε_0 von den beobachteten Returns abweicht, ist dies

nur in zwei Situationen der Fall: Zum einen, wenn der beobachtete Return $\mathcal{P}$ um weniger als ε_0 von der Vorhersage des Klassifizierers abweicht und der Zustand, für den er beobachtet wurde, weiter als r von dessen Gewichtsvektor entfernt ist – in diesem Fall ist der aktuelle Wert des Subsumtionsradius zu klein und muss vergrößert werden. Zum anderen ist eine Anpassung auch dann nötig, wenn der aktuelle Subsumtionsradius r zu groß ist, das heißt, wenn für einen Zustand, dessen Abstand zum Gewichtsvektor kleiner als r ist, ein Return beobachtet wird, der um mehr als ε_0 von der Vorhersage abweicht. Im diesem Fall wird der Subsumtionsradius verkleinert. Außer in diesen beiden Fällen ist keine Aussage darüber möglich, ob der aktuelle Wert des Subsumtionsradius zu klein oder zu groß ist und es wird keine Aktualisierung vorgenommen. Die Aktualisierung des Subsumtionsradius ist zudem an die Bedingung geknüpft, dass die Erfahrung des betroffenen Klassifizierers echt größer als 1 ist. Der Subsumtionsradius wird also frühestens dann aktualisiert, wenn dieser zum zweiten Mal als Gewinner einem Action-Set angehört. Der Grund hierfür ist, dass bei der ersten Aktualisierung eines Klassifizierers zum Zeitpunkt der Anpassung des Subsumtionsradius die Abweichung $|\mathcal{P} - p|$ verschwindet, da die Vorhersage des Klassifizierers zuerst aktualisiert wird. Somit hat diese Abweichung erst ab der zweiten Aktualisierung eines Klassifizierers einen Wert, der als Basis für die Entscheidung dienen kann, ob der Subsumtionsradius angepasst werden muss.

Bisher wurde hier nur der Fall der Verwendung skalarer Klassifizierervorhersagen betrachtet. Werden die Klassifizierervorhersagen stattdessen entsprechend (4.7) als Funktion des Umgebungszustandes berechnet, so tritt an die Stelle der Anpassung (4.26) der skalaren Klassifizierervorhersage eine Anpassung der Gewichte $\boldsymbol{v}$ für die Vorhersagenberechnung. Diese Anpassung erfolgt unter Verwendung eines auf der Methode der kleinsten Quadrate basierenden Verfahrens[18]. Um dieses einsetzen zu können speichert – wie bereits beim XCS – jeder Klassifizierer die letzten m Lernumgebungszustände $s_{t_1}, s_{t_2}, \ldots, s_{t_m}$, zu denen er passte und nach deren Präsentation er im Action-Set enthalten war, sowie die zugehörigen Returnabschätzungen $\mathcal{P}_{t_1}, \mathcal{P}_{t_2}, \ldots, \mathcal{P}_{t_m}$. Wird das HCS zur Funktionsapproximation eingesetzt, entsprechen die Returnabschätzungen $\mathcal{P}_{t_i}$ den Funktionswerten f_{t_i} an den Stellen s_{t_i}.
Bei der Anpassung der Gewichte $\boldsymbol{v}$ wird zunächst unter Verwendung der wie in Abschnitt 2.4.8 definierten erweiterten Zustände $s'_{t_1}, s'_{t_2}, \ldots, s'_{t_m}$ sowie des *erweiterten Gewichtsvektors* $\boldsymbol{w}' = (0, w_1, w_2, \ldots w_n)$ des betrachteten Klassifizierers die Matrix

$$X = \left[s'_{t_1} - \boldsymbol{w}', s'_{t_2} - \boldsymbol{w}', \ldots, s'_{t_m} - \boldsymbol{w}'\right]^T$$

berechnet. Der die Fehlerfunktion (2.41) minimierende Vektor $\hat{\boldsymbol{v}}$ wird dann unter Zuhilfenahme der Pseudoinversen X^+ von X oder der Singulärwertzerlegung von $X^T X$ als Lösung der Gleichung

$$X^T X \boldsymbol{v} = X^T \boldsymbol{f} \tag{4.32}$$

berechnet, wobei $\boldsymbol{f}$ der Vektor

$$\boldsymbol{f} = [\mathcal{P}_{t_1}, \mathcal{P}_{t_2}, \ldots, \mathcal{P}_{t_m}]^T$$

der zu den Lernumgebungszuständen $s_{t_1}, s_{t_2}, \ldots, s_{t_m}$ gespeicherten Returnabschätzungen ist. Die neuen Gewichte für die Vorhersagenberechnung werden dann wie schon beim XCS vermöge

$$\boldsymbol{v} \leftarrow (1 - \beta_o)\boldsymbol{v} + \beta_o \hat{\boldsymbol{v}} \tag{4.33}$$

als gewichtetes Mittel der bisherigen Gewichte und der als Lösung $\hat{\boldsymbol{v}}$ von (4.32) berechneten ermittelt.

[18] Eine zu (2.39) analoge Anpassung der Gewichte $\boldsymbol{v}$ unter Verwendung einer Widrow-Hoff-Delta-Regel ist ebenfalls denkbar, wird in dieser Arbeit aber nicht eingesetzt.

4.5 Discovery-Komponente

Beim HCS tritt neben die althergebrachten Discovery-Mechanismen Lernender Klassifizierender Systeme – den Genetischen Algorithmus und den Covering-Operator – ein dritter, der sich von den bei Selbstorganisierenden Karten eingesetzten Verfahren zur Anpassung der Neuronenpositionen ableitet. Dieser dritte Mechanismus ist insofern von anderer Art als die beiden zuerst genannten, als er im Gegensatz zu diesen keine neuen Klassifizierer erzeugt und in die Population einfügt, sondern bereits in der Population vorhandene Klassifizierer durch eine Adaptation ihrer Gewichtsvektoren modifiziert.
Während der Covering-Operator – wie bereits in 4.3.1 gesehen – nur bei Bedarf im Verlauf der Match-Set-Bildung aufgerufen wird, kommen der Genetische Algorithmus und der Positionsadaptationsmechanismus (in dieser Reihenfolge) nach jeder Aktualisierung eines Action-Sets zum Einsatz.

4.5.1 Der Covering-Operator

Der Covering-Operator des HCS stellt sicher, dass jede in der verwendeten Lernumgebung mögliche Aktion in allen gebildeten Match-Sets durch mindestens zwei Klassifizierer vertreten wird. Das HCS stellt somit etwas höhere Anforderungen an die „Vielfältigkeit“ der Match-Sets als das XCS, dem es genügt, wenn eine Mindestanzahl Θ_{MNA} verschiedener Aktionen einmal vertreten ist[19].

Motivation

Um ein optimales Verhalten lernen zu können, muss ein Lernendes Klassifizierendes System im Allgemeinen alle in seiner Lernumgebung ausführbaren Aktionen evaluieren können – diese müssen also sämtlich in seiner Population vertreten sein. Anders als beim XCS kann, sofern die Population des HCS überhaupt einen Klassifizierer mit einer bestimmten Aktion enthält, zu jedem beliebigen Umgebungszustand auch ein Gewinnerklassifizierer mit dieser Aktion bestimmt werden. Somit ist im Falle des HCS eine Aktion genau dann nicht im Match-Set vertreten, wenn sie in der Population nicht vertreten ist. Durch die Forderung, dass alle möglichen Aktionen im Match-Set vertreten sein müssen, wird also sichergestellt, dass alle Aktionen in der Population auftreten und getestet und bewertet werden können. Eine andere Möglichkeit, fehlende Aktionen in die Population einzuführen besitzt das HCS nicht: Sein Genetischer Algorithmus mutiert – anders als der des XCS – nur Bedingungen, keine Aktionen, sodass durch den Genetischen Algorithmus erzeugte Klassifizierer stets dieselbe Aktion wie ihre (aus der Population selektierten) Eltern besitzen.
Zu erläutern bleibt noch, warum jede Aktion durch mindestens zwei Makroklassifizierern im Match-Set – und damit auch in der Population – vertreten sein soll:
Um eine gute Aktionswahl treffen zu können, muss ein Lernendes Klassifizierendes System die Returns, zu denen die zur Verfügung stehenden Aktionen in verschiedenen Teilen des Zustandsraums führen, möglichst genau abschätzen. Dazu werden – außer in pathologischen Fällen – mehrere Klassifizierer pro Aktion benötigt. Diese müssen beim HCS – ausgehend von den initial durch den Covering-Operator erzeugten Klassifizierern – nach und nach durch den Genetischen Algorithmus erzeugt werden. Steht diesem nur ein Elternindividuum zur Verfügung, läuft der Crossing-Over-Operator ins Leere und die erzeugten Kinder stellen durch Mutation entstandene Varianten des Elters dar.

[19] Allerdings wird auch beim XCS meist Θ_{MNA} gleich der Anzahl möglicher Aktionen gesetzt.

Da der Mutationsoperator den Gewichtsvektor nur leicht verändert, unterscheiden sich die so erzeugten Kinder nur wenig von ihrem Elter, oftmals sogar nur so geringfügig, dass sie innerhalb dessen Subsumtionsradius liegen und dementsprechend nicht als „neue“ Klassifizierer, sondern als Kopien des Elters in die Population eingefügt werden. Hierdurch kann die Evolution der benötigten Anzahl von Klassifizierern verzögert werden. Indem der Covering-Operator sicherstellt, dass jede Aktion durch mindestens zwei Klassifizierer in Population und Match-Set vertreten ist, ermöglicht er einen sinnvollen Einsatz des Crossing-Over-Operators und beschleunigt so die Entwicklung einer Population, die eine der Lernumgebung angemessene Zahl von Makroklassifizierern enthält.

Ablauf

Wird der Covering-Operator aufgerufen, so durchläuft er wiederholt den Algorithmus 4.8, bis alle zur Verfügung stehenden Aktionen zweimal im Match-Set vertreten sind.

1. Wähle eine beliebige, nicht zweimal im Match-Set M vertretene Aktion a.

2. Falls a überhaupt nicht in M vertreten ist:

  2.1. Erzeuge einen Klassifizierer cl_1 mit
  - dem aktuellen Zustand s entsprechendem Gewichtsvektor $cl_1.\boldsymbol{w}$,
  - der Aktion a vertritt und
  - dessen Attribute wie folgt initialisiert werden:
    $cl_1.p \leftarrow p_{\text{initial}}$, $cl_1.\varepsilon \leftarrow \varepsilon_{\text{initial}}$, $cl_1.f \leftarrow f_{\text{initial}}$,
    $cl_1.avgd \leftarrow r_{\text{initial}}$, $cl_1.r \leftarrow r_{\text{initial}}$,
    $cl_1.exp \leftarrow 0$, $cl_1.num \leftarrow 1$, $cl_1.ts \leftarrow t$

  2.2. Setze: $r := r_{\text{initial}}$

3. Andernfalls:

  3.1. Setze: $r := cl^a_{w(s)}.r$.

4. Erzeuge einen Klassifizierer cl_2 mit
- zufälligem Gewichtsvektor $cl_2.\boldsymbol{w}$, wobei $\|cl_2.\boldsymbol{w} - cl_2.\boldsymbol{w}\| > r$,
- der Aktion a vertritt und
- dessen Attribute wie oben initialisiert werden.

5. Füge cl_2 und gegebenenfalls cl_1 in M und die Population P ein.

Algorithmus 4.8: Ergänzung des Match-Sets durch Covering.

Nach jedem Durchlauf wird überprüft, ob die maximale Größe $N_{\max}$ der Population überschritten wurde. Ist das der Fall, werden unter Verwendung von Algorithmus 4.2 Klassifizierer aus der Population gelöscht. Müssen zwei neue Klassifizierer erzeugt werden, wird der Gewichtsvektor des ersten dem aktuellen Umgebungszustand entsprechend gewählt, für den Gewichtsvektor des zweiten wird eine zufällige Position im Reizraum – jedoch außerhalb des Subsumtionsradius des ersten erzeugten Klassifizierers – gewählt. Ist nur ein Klassifizierer zu erzeugen, wird analog verfahren, sein zufällig bestimmter Gewichtsvektor muss dann außerhalb des Subsumtionsradius des bereits vorhandenen Klassifizierers mit der betrachteten Aktion liegen.
Neben Bedingung und Aktion müssen auch die Attribute der vom Covering-Operator erzeugten Klassifizierer festgelegt werden: Die neu erzeugten Klassifizierer haben Erfahrung 0 und Vielfachheit 1, sie erhalten einen dem aktuellen Lernschritt t entsprechenden Zeitstempel ts. Vorhersage, Vorhersagefehler und Fitness werden mit als Systemparametern des HCS gegebenen Initialwerten p_{initial}, $\varepsilon_{\text{initial}}$, f_{initial} belegt. Ein weiterer Systempa-

rameter, r_{initial}, dient der Initialisierung von Subsumtionsradius und Distanzabschätzung der neuen Klassifizierer.

4.5.2 Der Genetische Algorithmus

Beim Genetischen Algorithmus des HCS handelt es sich – wie bei dem des XCS – um einen auf den Action-Sets operierenden Steady-State-Algorithmus: Bei jeder Ausführung werden zwei Elternklassifizierer aus einem Action-Set selektiert und ausgehend von diesen durch Rekombination und Mutation zwei neue Klassifizierer erzeugt, die – sofern gewisse Bedingungen erfüllt sind – in die Population eingefügt werden.

Bedingungen für die Ausführung des Genetischen Algorithmus

Ob der Genetische Algorithmus in einem Lernschritt ausgeführt werden soll, wird jeweils nach der Aktualisierung eines Action-Sets – respektive des darin enthaltenen Gewinners – durch die Reinforcement-Komponente entschieden. Im Falle der Ausführung des Genetischen Algorithmus dient diesem das zuvor aktualisierte Action-Set als Elternpopulation – außer am Ende einer Lernepisode also immer das Action-Set des vorangegangenen Lernschritts.
Die Grundbedingung für die Ausführung des Genetischen Algorithmus ist die gleiche wie beim XCS: Die in der potentiellen Elternpopulation, dem betrachteten Action-Set A, enthaltenen Klassifizierer dürfen im Mittel seit mindestens Θ_{GA} Lernschritten keinem Action-Set angehört haben, aus dem der Genetische Algorithmus Elternklassifizierer selektiert hat:

$$\frac{\sum_{cl \in A}(t - cl.ts) \cdot cl.num}{\sum_{cl \in A} cl.num} \stackrel{!}{>} \Theta_{\text{GA}} \tag{4.34}$$

Um prüfen zu können, ob diese Bedingung erfüllt ist, wird nach jeder Ausführung des Genetischen Algorithmus der Zeitstempel ts aller in dem als Elternpopulation verwendeten Action-Set enthaltenen Klassifizierer gleich der Anzahl t der seit Beginn des Lernvorganges vergangenen Lernschritte gesetzt. Indem die Ausführung des Genetischen Algorithmus an die Bedingung (4.34) geknüpft wird, soll – wie schon in Abschnitt 2.4.6 ausgeführt – verhindert werden, dass Klassifizierer, die sehr oft einem Action-Set angehören, sich übermäßig stark vermehren.
Anders als beim XCS, das den Genetischen Algorithmus stets ausführt, wenn die zu (4.34) analoge Bedingung (2.35) erfüllt ist, geschieht dies beim HCS nur mit einer gewissen Wahrscheinlichkeit p_{GA}: In Abschnitt 4.2.1 wurde ausgeführt, dass ein zu häufiges Einfügen neuer Makroklassifizierer sich negativ auf Lernverlauf und Lernerfolg des HCS auswirken kann. Die Häufigkeit derartiger Einfügevorgänge hängt aber direkt ab von der Häufigkeit der Ausführung des Genetischen Algorithmus und kann somit über diese reguliert werden[20]. Prinzipiell wäre es möglich, die Häufigkeit der Ausführung des Genetischen Algorithmus direkt über den Schwellenwert Θ_{GA} zu beeinflussen. Da jedoch dessen eigentliche Aufgabe darin besteht, die Reproduktionsraten der Klassifizierer zu nivellieren, ist es klarer, dafür die Ausführungswahrscheinlichkeit p_{GA} zu nutzen. Dies hat zudem den Vorteil, dass eine Variation der Ausführungshäufigkeit nur innerhalb des durch (4.34) gegebenen Rahmens möglich ist.

[20] Das Einfügen von Klassifizierern durch den Covering-Operator kann in diesem Kontext unberücksichtigt bleiben. Es erfolgt nur, wenn eine Aktion überhaupt nicht oder nur einmal durch Klassifizierer in der Population vertreten ist und hat daher sicher keine negativen Auswirkungen auf den Lernverlauf.

Zusätzliche Einschränkungen bei Verwendung berechneter Klassifizierervorhersagen

Während der Entwicklung des HCS wurden verschiedene Versuche unternommen, die Häufigkeit potentiell störender Einfügevorgänge insbesondere in Bereichen des Zustandsraumes, in denen genaue Klassifizierer vorhanden sind, zu reduzieren. Zwei zusätzliche Maßnahmen zur Beschränkung der Aktivität des Genetischen Algorithmus haben sich dabei als erfolgreich erwiesen und zu einer Verbesserung von Lernergebnissen geführt – jedoch nur im Falle der Verwendung von Klassifizierern, deren Vorhersagen gemäß (4.7) berechnet werden. Bei Verwendung skalarer Klassifizierervorhersagen hingegen wurde das Lernergebnis bei den durchgeführten Tests sogar leicht negativ beeinflusst. Da der Grund hierfür bisher nicht ermittelt werden konnte, werden besagte Maßnahmen nur bei Verwendung berechneter Klassifizierervorhersagen getroffen:
In diesem Fall wird zum einen die Ausführung der Genetischen Algorithmus auf Fälle beschränkt, in denen der durchschnittliche Vorhersagefehler der im Action-Set enthaltenen Klassifizierer den in 4.3.2 eingeführten Schwellenwert ε_p überschreitet. Zum anderen wird der Genetische Algorithmus mit einer Wahrscheinlichkeit $p = \min(p_{\mathrm{GA}}, 1 - \kappa)$ ausgeführt, die die Genauigkeit κ des im als Elternpopulation in Betracht kommenden Action-Set enthaltenen Gewinners berücksichtigt. Im Falle einer hohen Genauigkeit $\kappa > 1-p_{\mathrm{GA}}$, wird die Wahrscheinlichkeit einer Ausführung des Genetischen Algorithmus also geringer. Ist der Gewinner vollständig genau ($\kappa = 1$), wird der Genetische Algorithmus nicht ausgeführt.

Ablauf des Genetischen Algorithmus

Wird der Genetische Algorithmus schließlich aufgerufen, so entspricht sein Ablauf im Wesentlichen einem Durchgang durch den in Abbildung 2.1 dargestellten Zyklus – also einer Generation eines gewöhnlichen Genetischen Algorithmus. Die dabei abzuarbeitenden Schritte sind in Algorithmus 4.9 zusammengefasst und werden im Folgenden besprochen.

Selektion

Hinsichtlich der Auswahl von Elternklassifizierern kommt theoretisch nahezu jedes aus der Literatur bekannte Selektionsverfahren Genetischer Algorithmen für einen Einsatz im HCS in Frage. Die im Rahmen dieser Arbeit erstellte Implementation stellt zwei alternativ einzusetzende Methoden zur Verfügung, die sich bereits beim XCS bewährt haben [Butz u. a. 2002; Kharbat u. a. 2005].
Dabei handelt es sich zum einen um die Rouletterad-Selektion, die vom XCS standardmäßig eingesetzt wird, zum anderen um eine Variante der Turnier-Selektion, bei der die Turniergröße, das ist die Anzahl miteinander verglichener Klassifizierer, nicht fest, sondern der Größe der Elternpopulation – also des betrachteten Action-Sets A – proportional ist. Es müssen jedoch stets mindestens zwei Klassifizierer an einem Turnier teilnehmen. Mit einem Proportionalitätsfaktor τ ist die Turniergröße demnach durch

$$\max(2, \tau \cdot |A|)$$

gegeben. Die Funktionsweise beider Selektionsverfahren wurde bereits in Kapitel 2 erläutert, weswegen hier nicht näher auf sie eingegangen wird.

1. *Selektion*: Selektiere zwei Elternklassifizierer $parent_1$ und $parent_2$ aus A.

2. *Variation*: Erzeuge zwei neue Klassifizierer $child_1$ und $child_2$:

2.1. Erzeuge die Gewichtsvektoren $child_1.\boldsymbol{w}$, $child_2.\boldsymbol{w}$ durch Variation der Gewichtsvektoren $parent_1.\boldsymbol{w}$, $parent_2.\boldsymbol{w}$ der ausgewählten Eltern:

2.1.1. Wende mit Wahrscheinlichkeit p_{cross} den Crossing-Over-Operator auf die Gewichtsvektoren $parent_1.\boldsymbol{w}$, $parent_2.\boldsymbol{w}$ an.

2.1.2. Mutiere die durch Crossing-Over entstandenen Gewichtsvektoren – oder $parent_1.\boldsymbol{w}$, $parent_2.\boldsymbol{w}$, falls kein Crossing-Over erfolgte.

2.2. Übernehme die Aktion $parent_1.a$ der Eltern für die Kinder. Für $i = 1, 2$:

2.2.1. Setze: $child_i.a \leftarrow parent_1.a$

2.3. Initialisiere die Attribute der Kinder. Für $i = 1, 2$:

2.3.1. Setze: $child_i.p \leftarrow p_{\text{initial}}$, $child_i.\varepsilon \leftarrow \varepsilon_{\text{initial}}$, $child_i.f \leftarrow f_{\text{initial}}$
$child_i.avgd \leftarrow r_{\text{initial}}$, $child_i.r \leftarrow r_{\text{initial}}$
$child_i.exp \leftarrow 0$, $child_i.num \leftarrow 1$, $child_i.ts \leftarrow t$

3. *Umweltselektion*: **Füge die Klassifizierer $child_1$ und $child_2$ in die Population P ein** *(siehe Algorithmus 4.1)*

4. Aktualisiere die Zeitstempel aller in A enthaltenen Klassifizierer. Für alle $cl \in A$:

4.1. Setze: $cl.ts \leftarrow t$

5. **Lösche Klassifizierer aus P, falls durch das Einfügen der neu erzeugten Klassifizierer die maximale Populationsgröße $N_{\max}$ überschritten wurde.** *(siehe Algorithmus 4.2)*

Algorithmus 4.9: Durchführung des Genetischen Algorithmus zum Zeitpunkt t unter Verwendung eines Action-Sets A als Elternpopulation.

Erzeugung zweier neuer Klassifizierer (Variation)

Im Anschluss an die Selektion werden zwei neue Klassifizierer erzeugt. Deren Klassifiziereraktion wird unverändert von den ausgewählten Eltern übernommen[21], aus deren Bedingungen werden durch Variation die Bedingungen der neuen Klassifizierer erzeugt. Dies geschieht, wie von Genetischen Algorithmen gewohnt, in zwei Schritten: Zunächst werden – mit einer gewissen Wahrscheinlichkeit p_{cross} – die Bedingungen der Elternklassifizierer unter Einsatz eines Crossing-Over-Operators rekombiniert. Auf die sich ergebenden Bedingungen wird dann der Mutationsoperator angewendet. Falls kein Crossing-Over durchgeführt wurde, ergeben sich die Bedingungen der neuen Klassifizierer allein durch Mutation der Bedingungen der Eltern.

Crossing Over kann das HCS in der für die vorliegende Arbeit erstellten Implementation in Form der drei in in Kapitel 2 eingeführten Varianten durchführen, wobei jede Variante Vor- und Nachteile aufweist:
2-Punkt-Crossing-Over und Uniformes Crossing-Over führen bei Anwendung auf Gewichtsvektoren zu einer besonderen Konstellation von Eltern und Kindern: Werden die Gewichtsvektoren der Elternklassifizierer als Koordinaten zweier einander „diagonal" gegenüberliegender Eckpunkte eines (achsenparallelen) Hyperquaders im Zustandsraum $\mathcal{S} \in \mathbb{R}^n$ aufgefasst, so liegen auch alle durch die beiden genannten Crossing-Over-

[21] Da die Elternklassifizierer aus dem gleichen Action-Set stammen, vertreten sie auch die gleiche Aktion.

Varianten erzeugbaren Gewichtsvektoren auf den Ecken dieses Hyperquaders. Dies stellt – wenn auch nicht bei allen Lernproblemen – einen Nachteil dar, da so die Koordinatenachsen wieder zu ausgezeichneten Achsen des Zustandsraumes werden: Bei Verwendung einer dieser Crossing-Over-Varianten zeigen die Klassifizierer des HCS die Tendenz, sich in einem achsenparallelen Gitter anzuordnen, was durch den Positionsadaptationsmechanismus gegebenenfalls wieder kompensiert werden muss. Ein Vorteil dieser Formen der Rekombination ist darin zu sehen, dass die erzeugten Kindklassifizierer sich relativ stark von ihren Eltern unterscheiden, sodass die Ausbreitung der Population im Zustandsraum gefördert wird. Andererseits ist es, wenn die erzeugten Kinder ihren Eltern relativ unähnlich sind, natürlich fraglich, ob durch Rekombination zweier Klassifizierer mit hoher Fitness wieder „gute" Klassifizierer entstehen.
Intermediäres Crossing-Over erzeugt – geometrisch betrachtet – Gewichtsvektoren, die auf der Verbindungsstrecke der elterlichen Gewichtsvektoren liegen. Bei Verwendung dieser Form der Rekombination liegen daher die erzeugten Kindklassifizierer mit hoher Wahrscheinlichkeit[22] in der konvexen Hülle der Gewichtsvektoren der Klassifizierer des Action-Sets, dem ihre Eltern entnommen wurden. Tritt der vor Bildung dieses Action-Sets beobachtete Umgebungszustand erneut auf, sind die auf diese Weise erzeugten Klassifizierer somit in vielen Fällen wieder Teil des entsprechenden Match-Sets. In diesem Sinne stellt das Intermediäre Crossing-Over einen „Nischenoperator" dar. Somit ist eher als beim Uniformen oder 2-Punkt-Crossing-Over anzunehmen, dass durch Rekombination zweier guter Klassifizierer wieder gute neue Klassifizierer entstehen. Dafür erfolgt aber die Exploration des Zustandsraums, das heißt die Erzeugung von Klassifizierern in „unbesiedelten" Bereichen desselben, weniger zügig.

Mutation erfolgt beim Genetischen Algorithmus des HCS in Form der in Kapitel 2 vorgestellten reellwertigen Punktmutation: Jeder Eintrag eines zu variierenden Gewichtsvektors wird mit einer geringen Wahrscheinlichkeit p_{mut} verändert, indem sein aktueller Wert um einen kleinen Betrag erhöht oder vermindert wird.
Die hier verwendete Implementation des HCS stellt zwei Varianten der Punktmutation zur Verfügung, die alternativ zu verwenden sind. Der einzige Unterschied zwischen diesen besteht darin, wie die bei der Mutation einzelner Gene verwendeten Inkremente[23] verteilt sind:

- Bei der ersten – der „klassischen" – Variante folgen diese einer stetigen Gleichverteilung $\mathcal{U}\left[-inc_{\mathrm{mut}},\ inc_{\mathrm{mut}}\right]$, der Parameter inc_{mut} des HCS gibt in diesem Fall den maximal zulässigen Betrag der Inkremente an.
- Die zweite Variante verwendet normalverteilte Inkremente, wobei der Systemparameter inc_{mut} die Varianz der verwendeten Normalverteilung $\mathcal{N}\left(0, inc_{\mathrm{mut}}\right)$ festlegt.

Während die erste Variante – bei entsprechend gewähltem inc_{mut} – ausschließlich kleine Inkremente erlaubt, treten bei der zweiten auch größere Inkremente auf; das Hauptgewicht liegt jedoch weiterhin auf kleinen Inkrementen.

Initialwerte der Klassifizierer-Attribute werden, nachdem Bedingung und Aktion der neu erzeugten Klassifizierer feststehen, in exakt der gleichen Weise festgelegt, wie beim

[22] Da im Anschluss an das Crossing-Over noch eine Mutation erfolgt, ist dies nicht immer der Fall.
[23] Der Begriff Inkrement wird im Folgenden als sowohl positive wie auch negative Inkremente (Dekremente) umfassend verstanden.

Covering: Die initialen Werte der Vorhersage p, des Vorhersagefehlers ε, der Fitness f, der Distanzabschätzung $avgd$ und des Subsumtionsradius r sind durch die Systemparameter p_{initial}, $\varepsilon_{\text{initial}}$, f_{initial} und r_{initial} vorgegeben, wobei der Wert von r_{initial} zur Initialisierung sowohl des Subsumtionsradius als auch der Distanzabschätzung verwendet wird. Die neuen Klassifizierer haben keine Erfahrung ($exp = 0$) und Vielfachheit 1. Ihr Zeitstempel ts wird, ebenso wie die Zeitstempel aller dem als Elternpopulation verwendeten Action-Set angehörenden Klassifizierer, gleich der Anzahl t bereits vergangener Lernschritte gesetzt.

Umweltselektion

Anders als bei einfachen Genetischen Algorithmen ist es im Fall eines Lernenden Klassifizierenden Systems nicht möglich, die Fitness eines Klassifizierers unmittelbar nach dessen Erzeugung zu ermitteln – dies ist erst möglich, wenn der Klassifizierer (mehrfach) aktiviert und bewertet wurde. Dementsprechend kann es eine Umweltselektion im herkömmlichen Sinne beim HCS nicht geben.
In gewisser Weise übernimmt jedoch der in 4.2.2 beschriebene Einfügemechanismus des HCS die Rolle einer Umweltselektion: Er entscheidet, ob ein Klassifizierer in die Population übernommen wird, oder ob stattdessen ein bereits in dieser enthaltener allgemeiner(er) und genauer Klassifizierer durch Erhöhung seiner Vielfachheit gestärkt wird.

Fitness-Sharing

Bei der Selektion von Elternindividuen und bei der Berechnung der Löschvoten (4.12) verwendet das HCS (optional) eine *geteilte Fitness* anstelle der gewöhnlichen Fitness der Klassifizierer.
Werden Genetische Algorithmen auf Probleme angewendet, die mehrere optimale Lösungen besitzen, so führen stochastische Effekte bei der Selektion schnell zu einer Konvergenz der Population auf eine dieser Lösungen; dieser Effekt wird als *genetische Drift* bezeichnet. Ein Mechanismus[24], der dem entgegenwirkt, ist das sogenannte *Fitness-Sharing* [siehe z.B.: Gerdes u. a. 2004], dessen Vorbild die Natur liefert: Die Fortpflanzungschancen eines Individuums – etwa eines Tieres – hängen nicht nur von seiner genetischen Ausstattung ab, sondern zum Beispiel auch davon, ob es genug Nahrung zur Verfügung hat. Je mehr Individuen sich einen (begrenzten) Lebensraum teilen müssen, desto weniger Ressourcen bleiben für jedes einzelne und desto geringer ist die Fortpflanzungswahrscheinlichkeit, also die Fitness, jedes einzelnen Individuums. Übertragen auf Genetische Algorithmen hat dies zur Folge, dass, wenn eine von mehreren optimalen Lösungen von vielen, eine andere nur von wenigen Individuen repräsentiert wird, die Fitness der die erste Lösung vertretenden Individuen relativ zur Fitness der die zweite vertretenden sinkt, wodurch die Konvergenz der Population auf nur eine optimale Lösung verhindert wird. Es wird die Bildung von Subpopulationen für durch die optimalen Lösungen definierte Nischen des Suchraums gefördert.
Im Falle des HCS erfolgt eine Nischenbildung bereits implizit dadurch, dass der Genetische Algorithmus Elternindividuen aus den Action-Sets selektiert. Diese definieren, da sie jeweils einen zu einem zuvor präsentierten Zustand passenden sowie einige diesem ähnliche Klassifizierer enthalten, Nischen im Zustandsraum. Die Verwendung von Fitness-Sharing kann jedoch helfen, die zur Verfügung stehende Zahl von Klassifizierern gleichmäßiger auf diese Nischen zu verteilen. Dazu trägt zum einen bei, dass

[24] Es gibt mehrere derartige Verfahren, die unter der Bezeichnung *Niching-Verfahren* oder *nischenbildende Techniken* firmieren.

durch Fitness-Sharing die Wahrscheinlichkeit der Selektion eines Klassifizierers verringert wird, wenn es viele ihm ähnliche Klassifizierer gibt, zum anderen aber auch, dass infolge der Verwendung der geteilten Fitness bei der Berechnung der Löschvoten die Löschwahrscheinlichkeit eines solchen Klassifizierers steigt.
Die geteilte Fitness f_s eines Klassifizierers wird berechnet, indem dessen Fitness durch einen *Nischen-Zähler* geteilt wird:

$$cl.f_s = \frac{cl.f}{\sum_{\tilde{cl} \in P} Sh(\tilde{cl}, cl)} \tag{4.35}$$

Die Sharingfunktion

$$Sh(\tilde{cl}, cl) = \begin{cases} \left(1 - \left(\frac{\|cl.\boldsymbol{w} - \tilde{cl}.\boldsymbol{w}\|}{\sigma_{Sh}}\right)^{\nu_{Sh}}\right) \cdot \tilde{cl}.num & \text{, falls } \left\|cl.\boldsymbol{w} - \tilde{cl}.\boldsymbol{w}\right\| < \sigma_{Sh} \\ 0 & \text{, sonst} \end{cases} \tag{4.36}$$

bewirkt, dass Klassifizierer sich umso weniger beeinflussen, je größer der euklidische Abstand ihrer Gewichtsvektoren ist, überschreitet deren Entfernung einen vorgegebenen *Nischenradius* σ_{Sh}, verschwindet der gegenseitige Einfluss auf die geteilte Fitness.

4.5.3 Der Netzlernmechanismus

Der zur Anpassung der Positionen der Klassifizierer eingesetzte Mechanismus ist – neben der Verwendung von Gewichtsvektoren und dem Topologielernen – das dritte von Selbstorganisierenden Karten inspirierte Element des HCS. Wie in Abschnitt 4.1 angemerkt wurde, müssen die Klassifizierer des HCS aufgrund ihrer Kontextsensitivität sowohl absolut wie auch relativ zueinander sehr genau im Zustandsraum positioniert werden. Dies kann der Genetische Algorithmus alleine nicht leisten: Dieser ist gut geeignet, den Zustandsraum zu explorieren und die Population in diesem auszubreiten. Sobald jedoch eine der jeweiligen Lernumgebung angemessene Zahl schon einigermaßen genauer Klassifizierer vorhanden ist und „nur" noch eine Feinanpassung der Klassifizierer nötig ist, kann durch den Genetischen Algorithmus kaum noch eine Verbesserung erzielt werden. Selbst wenn durch Genetische Variation ein Klassifizierer mit optimalem Gewichtsvektor erzeugt würde, läge dieser doch mit hoher Wahrscheinlichkeit innerhalb des Subsumtionsradius eines bereits vorhandenen Klassifizierers und würde beim Einfügen lediglich dessen Vielfachheit erhöhen. Während also der Genetische Algorithmus die Exploration des Zustandsraums übernimmt, obliegt die „Feinanpassung" der Klassifiziererpositionen dem Netzlernmechanismus des HCS.

Positionsadaptation

Die Lernregeln Selbstorganisierender Karten zielen in der Regel darauf ab, die Neuronen so zu positionieren, dass die Verteilung der Eingabedaten möglichst gut wiedergegeben wird. Die Kohonenkarte, aber auch das Wachsende Neurale Gas, etwa verteilen die Neuronen – wie das Beispiel in Abbildung 3.5(b) zeigt – so, dass deren Verteilung die Häufigkeitsverteilung der Eingaben widerspiegelt: Treten aus einem Bereich häufiger als aus anderen Eingaben auf, so liegen die Neuronen dort dichter.
Im Falle des HCS ist eine derartige Verteilung von Klassifizierern im Allgemeinen nicht vereinbar mit dem Wunsch, möglichst generelle Klassifizierer zu entwickeln. Die Adaptation der Klassifizierer muss vielmehr darauf ausgerichtet sein, diese so zu positionie-

ren, dass sie genaue Vorhersagen machen und dadurch die Wahl der jeweils richtigen Aktion unterstützen.

Im Allgemeinen wird der Matchbereich eines Klassifizierers sowohl Zustände enthalten, für die seine Vorhersage zutreffend ist, als auch solche, für die dies nicht der Fall ist. Die Grundidee des Positionsadaptationsverfahrens des HCS besteht darin, Klassifizierer auf die erstgenannten Zustände hin und von den letztgenannten weg zu bewegen. Ob die Vorhersage eines Klassifizierers für einen bestimmten Zustand zutreffend ist oder nicht, wird ermittelt, indem des Klassifizierers Vorhersage für diesen Zustand mit dem Return verglichen wird, der nach Ausführung der von dem Klassifizierer vertretenen Aktion in dem betrachteten Zustand zu beobachten ist. Dies impliziert, dass stets nur die Position von in Action-Sets enthaltenen Gewinnern adaptiert wird. Eine Vorhersage gilt im Kontext des angesprochenen Vergleichs dann als zutreffend, wenn ihre Abweichung vom zugehörigen Return kleiner ist als der Vorhersagefehler des betrachteten Klassifizierers, andernfalls wird sie als unzutreffend angesehen. Die „Vorzeichen“ ι der Bewegung eines zu adaptierenden Klassifizierers cl ergibt sich demnach wie folgt:

$$\iota = \begin{cases} 1 & \text{, falls } |cl.p - \mathcal{P}| < cl.\varepsilon \\ -1 & \text{, sonst} \end{cases} \tag{4.37}$$

Die Returnabschätzung $\mathcal{P}$ ist dabei in der gleichen Weise wie bei der in Abschnitt 4.4 erläuterten Aktualisierung der Attribute eines Gewinnerklassifizierers definiert.
Auf den ersten Blick liegt es vielleicht näher, anstelle des Vorhersagefehlers des jeweils betrachteten Klassifizierers eine feste Fehlerschranke zu verwenden. Dies hat sich in während der Entwicklung des HCS durchgeführten Versuchen jedoch als nicht zielführend erwiesen: Ein angemessen kleiner Wert einer solchen Fehlerschranke wird zu Beginn des Lernvorganges nur in den seltensten Fällen unterschritten, sodass die Klassifizierer an die Grenzen des Zustandsraums und über diese hinaus geschoben werden, wodurch ein Lernerfolg verzögert oder sogar verhindert wird. Ein großer Wert hingegen entspricht offenbar nicht dem Ziel, Klassifizierer zu entwickeln, die die zu erwartenden Returns möglichst exakt vorhersagen. Die Richtungsbestimmung in der durch (4.37) beschriebenen Form hingegen wirkt stets auf eine Verbesserung relativ zur jeweils aktuellen Güte der Vorhersage eines Klassifizierers hin.
Während die Richtung der Anpassung eines Klassifizierer-Gewichtsvektors nur von der Exaktheit der Vorhersage für den aktuell betrachteten Zustand abhängt, wird die Stärke der Anpassung von der generellen Genauigkeit des betrachteten Klassifizierers bestimmt. Es liegt nahe, die Positionen von Klassifizierern, die bereits eine hohe Genauigkeit aufweisen, weniger stark anzupassen als die von Klassifizierern, die ungenau sind. Um dies zu erreichen, verwendet das HCS bei der Adaptation der Klassifizierer-Gewichtsvektoren eine Lernrate der folgenden Form:

$$\eta(cl) = \eta_0 + \eta \cdot (1 - cl.\kappa) \tag{4.38}$$

Diese setzt sich aus einem variablen, von der Genauigkeit κ des anzupassenden Klassifizierers cl abhängigen Teil und einem konstanten Beitrag η_0 zusammen. Letzterer stellt die Lernrate für die Anpassung vollständig genauer ($\kappa = 1$) Klassifizierer dar.
Insgesamt ergibt sich für die Positionsveränderung eines Gewinnerklassifizierers cl_w^a bei Adaptation an einen Zustand s also:

$$\Delta \boldsymbol{w} = \iota \cdot (\eta_0 + \eta \cdot (1 - cl_w^a.\kappa)) \cdot (s - cl_w^a.\boldsymbol{w}) \tag{4.39}$$

Diese Positionsveränderung wird in jedem Lernschritt für den im Action-Set des vorangegangenen Lernschrittes enthaltenen Gewinner sowie am Ende einer Lernepisode für

den im aktuellen Action-Set enthaltenen berechnet. Eine Veränderung der Position von Klassifizierern erfolgt jedoch immer erst am Ende einer Lernepisode:

Lernen aus Trajektorien

Beim Einsatz eines Lernenden Klassifizierenden Systems zur Klassifikation oder zur Funktionsapproximation haben alle Lernepisoden die Länge 1, sodass die auftretenden Zustände als zufällig verteilt vorausgesetzt werden können. Sollen hingegen in Interaktion mit einer Lernumgebung Aktionen gelernt werden, so treten auch Lernepisoden mit einer Länge größer als 1 auf. In diesem Fall sind die dem System präsentierten Zustände nicht mehr unabhängig voneinander; vielmehr bilden in jeder Lernepisode die aufeinander folgenden Zustände eine Trajektorie durch den Zustandsraum. Jede solche Trajektorie endet in einem von in der Regel wenigen möglichen Terminalzuständen. Oftmals sind zumindest einige in einer Episode aufeinander folgende Zustände einander ähnlich und haben auch ähnliche Zustands-Aktions-Werte. Dies kann dazu führen, dass ein Klassifizierer in mehreren aufeinander folgenden Lernschritten Gewinner ist.
Beim HCS bestünde daher, würde die Positionsadaptation in jedem Lernschritt direkt erfolgen, die Gefahr, dass Klassifizierer „mitgezogen" werden und sich in der Nähe von Terminalzuständen konzentrieren. Um dies zu vermeiden, werden die Gewichtsvektoren der Klassifizierer während einer Episode nicht verändert und erst am Ende einer Episode adaptiert. Zu diesem Zweck wird jeder HCS-Klassifizierer mit zwei zusätzlichen (Hilfs-) Attributen ausgestattet:

- In $\tilde{\boldsymbol{w}}$ werden während einer Episode die entsprechend (4.39) berechneten Veränderungen der Position des jeweiligen Klassifizierers aufaddiert.
- In einem Zähler z wird vermerkt, wie oft $\tilde{\boldsymbol{w}}$ in der jeweils aktuellen Lernepisode geändert wurde.

Algorithmus 4.10 fasst die in einem Lernschritt erfolgende Berechnung der Positionsveränderung eines Gewinner-Klassifizierers und die anschließend erfolgende Aktualisierung seiner oben genannten Hilfsattribute zusammen:

1. Bestimme die „Richtung" der Verschiebung des Gewichtsvektors des in A enthaltenen Gewinners cl_w^a:

$$\iota := \begin{cases} 1 & \text{, falls } |cl_w^a.p - \mathcal{P}| < cl_w^a.\varepsilon \\ -1 & \text{, sonst} \end{cases}$$

2. Berechne die Stärke der Adaption:

$$\eta(cl_w^a) := \eta_0 + \eta \cdot (1 - cl_w^a.\kappa)$$

3. Akkumuliere die Adaptationen von cl_w^a:

$$cl_w^a.\tilde{\boldsymbol{w}} \leftarrow cl_w^a.\tilde{\boldsymbol{w}} + \iota \cdot \eta(cl_w^a) \cdot (s - cl_w^a.\boldsymbol{w}) \tag{4.40}$$

$$cl_w^a.z \leftarrow cl_w^a.z + 1 \tag{4.41}$$

Algorithmus 4.10: Die Berechnung der Positionsanpassung des in einem Action-Set A enthaltenen Gewinners cl_w^a für den Zustand s unter Berücksichtigung der Returnabschätzung $\mathcal{P}$.

Am Ende einer Lernepisode werden dann alle Klassifizierer, die in dieser Episode als Gewinner in einem Action-Set enthalten waren, adaptiert, indem zu ihrem Gewichtsvektor

der Durchschnitt der für sie akkumulierten Positionsveränderungen addiert wird, wie in Schritt 2.1.1 des Algorithmus 4.11 angegeben.

1. Initialisiere: $P_{adapted} = \emptyset$

2. Für alle Klassifizierer $cl \in P$:

2.1. Falls $cl.z > 0$:

2.1.1. Übernimm die erfolgten Positionanpassungen:

$$cl.\boldsymbol{w} \leftarrow cl.\boldsymbol{w} + \frac{cl.\tilde{\boldsymbol{w}}}{cl.z} \tag{4.42}$$

2.1.2. Setze: $cl.\tilde{\boldsymbol{w}} \leftarrow \perp$
$cl.z \leftarrow 0$
$P_{adapted} \leftarrow P_{adapted} \cup \{cl\}$

3. Für alle Klassifizierer $cl \in P_{adapted}$:

3.1. Initialisiere: $subsumer \leftarrow \perp$.

3.2. Überprüfe, ob es einen $\tilde{cl}$ subsumierenden Klassifizierer gibt, setze ggf.:

$$subsumer \leftarrow \underset{\substack{cl \in P_a, \\ \left\|cl.\boldsymbol{w} - \tilde{cl}.\boldsymbol{w}\right\| \leq cl.r, \\ cl.exp > \tilde{cl}.exp}}{\arg\min} \left\|cl.\boldsymbol{w} - \tilde{cl}.\boldsymbol{w}\right\|$$

3.3. Falls $subsumer \neq \perp$:

3.3.1. Setze: $subsumer.num \leftarrow subsumer.num + \tilde{cl}.num$
$subsumer.exp \leftarrow subsumer.exp + \tilde{cl}.exp$

3.3.2. Für alle von $\tilde{cl}$ ausgehenden Kanten:

3.3.2.1. Ersetze den Endpunkt $\tilde{cl}$ durch $subsumer$

3.3.3. Setze: $P \leftarrow P \cup \left\{\tilde{cl}\right\}$

Algorithmus 4.11: Das Übernehmen der Positionsanpassungen am Ende einer Lernepisode.

Subsumtion bei der Positionsadaptation

Ebenso wie das Einfügen neuer Klassifizierer in Bereiche des Zustandraumes, für die andere Klassifizierer schon genaue Vorhersagen machen, kann auch das Verschieben existierender Klassifizierer in solche Bereiche die Entwicklung genauer und zugleich genereller Klassifizierer behindern. Aus diesem Grund kommt auch im Kontext der Positionsadaptation ein Subsumtionsmechanismus zum Einsatz (Schritt 3 des Algorithmus 4.11). Auf die Funktionsweise dieses Subsumtionsmechanismus muss hier nicht mehr eingegangen werden, da er sich nur in drei Punkten von dem beim Einfügen neuer Klassifizierer verwendeten unterscheidet:

- Als zusätzliche Anforderung an einen subsumierenden Klassifizierer wird gestellt, dass seine Erfahrung größer sein muss als die des subsumierten.
- Die Vielfachheit des subsumierenden Klassifizierers wird um die Vielfachheit des subsumierten erhöht, nicht nur um 1. Ferner wir dem subsumierenden Klassifizierer die Erfahrung des subsumierten übertragen.

- Wird die topologiebasierte Variante der Match-Set-Bildung eingesetzt, so werden alle Kanten, über die der subsumierte Klassifizierer mit anderen verbunden ist, zu dem subsumierenden Klassifizierer „umgebogen", das Kantenalter bleibt erhalten.

Der letzte Punkt ist wesentlich: Würden mit dem subsumierten Klassifizierer auch die von ihm ausgehenden Kanten gelöscht, entstünde ein „Loch" in der Topologie der jeweiligen Teilpopulation. Dadurch könnten, da die Bestimmung der bei der Systemvorhersagenberechnung berücksichtigten Klassifizierer auf der Topologie basiert, die Genauigkeit der Systemvorhersagen und damit die Leistung des Systems beeinträchtigt werden.

Positionsadaptation bei Verwendung berechneter Klassifizierervorhersagen

Werden HCS-Klassifizierer verwendet, deren Vorhersagen gemäß (4.7) berechnet werden, so erfolgt die Positionsadaptation in leicht modifizierter Form. Dies ist einem Programmierfehler bei der Implementation des HCS zu schulden, der dazu führte, dass im Falle einer unzutreffenden Vorhersage die Position des betreffenden Klassifizierers nicht verändert wurde. Nach Korrektur dieses Bugs wurden zuvor durchgeführte Testläufe wiederholt. Dabei verbesserten sich die Lernergebnisse bei Verwendung skalarer Vorhersagen, verschlechterten sich jedoch im Allgemeinen, wenn die Vorhersagen berechnet wurden. Dementsprechend wurde das HCS dergestalt verändert, dass in diesem Fall

$$\iota' = \begin{cases} 1 & \text{, falls } |cl.p - \mathcal{P}| < cl.\varepsilon \\ 0 & \text{, sonst} \end{cases} \tag{4.43}$$

anstelle von (Gleichung (4.37)) in die Berechnung der Positionsveränderung (4.39) eingeht. Der Gewichtsvektor eines Klassifizierers wird also nur verändert, wenn seine Vorhersage für einen Zustand s um einen Betrag kleiner seines Vorhersagefehlers vom beobachteten zugehörigen Return abweicht. Im Falle, dass der Klassifizierer eine unzutreffende Vorhersage macht, wird seine Position nicht verändert.
Ein Grund dafür, dass dieses Verfahren im Falle der Verwendung berechneter Klassifizierervorhersagen zu besseren Ergebnissen führt, könnte der folgende sein: Die Aktions-Wertefunktion einer Lernumgebung wird im Allgemeinen nur lokal durch eine lineare Vorhersage der Form (4.7) gut approximierbar sein. Dementsprechend sollten die Zustände, basierend auf denen die Gewichte $\boldsymbol{v}$ für die Vorhersagenberechnung ermittelt werden, zumindest nicht allzu verschieden sein. Dies sind, wie in 4.4 erläutert wurde, Zustände für die der betrachtete Klassifizierer Gewinner und im Action-Set enthalten war. Wird ein (neuer) Klassifizierer, der berechnete Vorhersagen verwendet, bewegt, solange seine Vorhersage noch ungenau ist, besteht die Gefahr, dass die Zustände, zu denen er im Laufe der Zeit passt, zu verschieden sind, um als Grundlage für die Bestimmung sinnvoller Gewichte $\boldsymbol{v}$ zu dienen.
Wenn diese Vermutung zutrifft, sollte eine Berechnung von Positionsanpassungen unter Verwendung von (4.37) im Falle berechneter Klassifizierervorhersagen weniger problematisch sein, sofern die Aktions-Wertefunktion einer Lernumgebung in großen Bereichen des Zustandsraumes linear ist. Um dies zu überprüfen, wurde die Funktion

$$f_{\text{roof}}(x, y) = \begin{cases} x + y & \text{, falls } x + y < 1 \\ 2 - (x + y) & \text{, sonst} \end{cases}$$

auf dem Einheitsquadrat approximiert[25] – einmal unter Benutzung von (4.43), ein weiteres Mal unter Verwendung von (4.37). In der Tat hat sich bei dieser, aus zwei linearen

[25] Die Vorgehensweise entsprach dabei der bei den in Kapitel 7 noch zu besprechenden Experimenten zur Funktionsapproximation.

Teilen zusammengesetzten Funktion, die Anwendung von (4.37) bei der Berechnung der Positionsanpassungen nicht nur nicht negativ ausgewirkt, sondern im Gegenteil sogar eine etwas bessere Approximation der Zielfunktion zur Folge gehabt.

4.6 Übersicht über das HCS

Nachdem in den vorangegangenen Abschnitten die Komponenten des HCS im Detail besprochen wurden, bleibt noch, diese in den Zusammenhang des Gesamtsystems einzuordnen. Algorithmus 4.12 beschreibt einen Lauf des HCS: Die die Schritte 2 bis 10 dieses Algorithmus umfassende *Hauptschleife* des HCS, die in jedem Lernschritt durchlaufen wird und in der auch die Interaktion mit der Lernumgebung erfolgt, wird eingerahmt von einem Initialisierungsschritt und der Überprüfung eines Terminierungskriteriums.

4.6.1 Initialisierung

Wird das HCS gestartet, müssen zunächst der Lernschritt-Zähler t, dem der Wert 0 zugewiesen wird, sowie die Population P und die Kantenmenge $\mathcal{E}$ initialisiert werden. Die einfachste Möglichkeit letzteres zu tun, besteht darin, beide Mengen leer zu belassen und ihre „Befüllung“ ganz dem HCS selbst zu überlassen . Ebenso denkbar ist es aber auch, eine nicht-leere Startpopulation zu verwenden. Eine solche könnte zum Beispiel einige gänzlich zufällig erzeugte Klassifizierer oder auch eine Reihe gleichmäßig im Zustandsraum positionierter Klassifizierer enthalten.
Für die Verwendung einer nicht-leeren Startpopulation spricht vor allem, dass beim HCS wesentlich weniger Klassifizierer durch Covering zu Beginn eines Laufes erzeugt werden als etwa beim XCS. Die evolutionäre Suche nach guten Klassifizierern startet also von einer vergleichsweise schmalen Basis. Eine Verbreiterung dieser Basis durch Bereitstellung einer Startpopulation kann den Lernvorgang unter Umständen beschleunigen.
Wird eine Startpopulation verwendet, muss deren Größe mit Bedacht gewählt werden. Einerseits wird eine nur wenige Klassifizierer enthaltende Startpopulation wenig Nutzen bringen, andererseits kann eine zu große im weiteren Lernverlauf die Entwicklung genereller Klassifizierer erschweren. Im Falle der Verwendung der topologiebasierten Variante der Match-Set-Bildung ist ferner zu überlegen, ob und in welcher Weise eine Topologie auf der Startpopulation vorgegeben werden soll.

Bei den im Rahmen dieser Arbeit durchgeführten Experimenten wurde, nachdem sich die Lernergebnisse mit und ohne Startpopulation in einigen vorbereitenden Tests nur geringfügig unterschieden, stets die einfachere Variante gewählt und die Population leer initialisiert. Der erste Schritt des Algorithmus 4.12 spiegelt dies wider.

4.6.2 Die Hauptschleife des HCS

Performance-, Reinforcement- sowie Discovery-Komponente des HCS sind in die Hauptschleife des HCS eingebettet, die die Schritte 2 bis 10 des Algorithmus 4.12 umfasst und die das System in jedem Lernschritt durchläuft.
Die Performance-Komponente – in den Schritten 3 bis 5 des Algorithmus – wird eingefasst von der Interaktion des HCS mit seiner Lernumgebung in den Schritten 2 und 6.

1. Initialisiere die Parameter des HCS.
Initialisiere eine leere Population: $P \leftarrow \emptyset$, $\mathcal{E} \leftarrow \emptyset$
Setze $t := 0$.

2. Nehme Zustand $s_t \in \mathcal{S}$ der Lernumgebung war.

3. **Generiere das Match-Set M.**
(siehe Algorithmus 4.3 oder 4.4)

4. Für jede Aktion $a \in \mathcal{A}$:

4.1. **Berechne die Systemvorhersage PA_a**
(siehe Algorithmus 4.5)

5. **Wähle die auszuführende Aktion $\tilde{a} \in \mathcal{A}$ und generiere das Action-Set A_t.**
(siehe Algorithmus 4.6)

6. Führe $\tilde{a}$ in der Lernumgebung aus und empfange die Belohnung $r_{t+1} \in \mathbb{R}$.

7. Falls $A_{t-1} \neq \emptyset$:

7.1. **Aktualisiere die Attribute des in A_{t-1} enthaltenen Gewinners bezüglich des Returns $\mathcal{P} = r_t + \gamma \cdot \max_{a \in \mathcal{A}}(PA)$.**
(siehe Algorithmus 4.7)

7.2. Falls Bedingung (4.34) erfüllt ist:

7.2.1. **Führe den Genetischen Algorithmus auf A_{t-1} aus.**
(siehe Algorithmus 4.9)

7.3. **Akkumuliere die Positionsveränderung für den in A_{t-1} enthaltenen Gewinnerklassifizierer.**
(siehe Algorithmus 4.10)

8. Falls das Ende eines Lernepisode erreicht ist:

8.1. **Aktualisiere die Attribute des in A_t enthaltenen Gewinners bezüglich des Returns $\mathcal{P} = r_{t+1}$.**
(siehe Algorithmus 4.7)

8.2. Falls Bedingung (4.34) erfüllt ist:

8.2.1. **Führe den Genetischen Algorithmus auf A_t aus.**
(siehe Algorithmus 4.9)

8.3. **Akkumuliere die Positionsveränderung für den in A_t enthaltenen Gewinnerklassifizierer.**
(siehe Algorithmus 4.10)

8.4. **Übernimm Änderungen der Gewichtsvektoren aller in der Population enthaltenen Klassifizierer.**
(siehe Algorithmus 4.11)

8.5. Setze: $A_t \leftarrow \emptyset$

9. Setze: $t \leftarrow t + 1$

10. Falls $A_{t-1} \neq \emptyset$:

10.1. Gehe zu Schritt 2.

11. Falls ein vorgegebenes Stopp-Kriterium noch nicht erfüllt ist, gehe zu Schritt 2.

Algorithmus 4.12: Ein Lauf des HCS.

Jeder Lernschritt beginnt damit, dass das System den aktuellen Zustand der Lernumgebung wahrnimmt und auf eine der in Abschnitt 4.3.1 beschriebenen Weisen ein Match-Set bildet. Es folgen die Berechnung der Systemvorhersagen, die Auswahl der auszuführenden Aktion $\tilde{a}$ und die Bildung des Action-Sets A_t des aktuellen Lernschritts. Die ausgewählte Aktion wird sodann in der Lernumgebung ausgeführt, woraufhin das HCS eine Belohnung von dieser entgegennehmen kann.
Sofern mit dem aktuellen Lernschritt keine neue Lernepisode begonnen hat, sodass das Action-Set A_{t-1} nicht leer ist, werden die Attribute des in diesem enthaltenen Gewinners auf die in Abschnitt 4.4 beschriebene Weise durch die Reinforcement-Komponente aktualisiert (Schritt 7.1). Anschließend erfolgt der Einsatz des Discovery-Komponente (Schritte 7.2 und 7.3): Unter den in Abschnitt 4.5.2 genannten Bedingungen wird zunächst der Genetische Algorithmus ausgeführt. Danach wird die Positionsveränderung für den in A_{t-1} enthaltenen Gewinner berechnet und akkumuliert.
Wurde mit dem aktuellen Lernschritt das Ende einer Lernepisode erreicht, kommen Reinforcement- und Discovery-Komponente nochmals zum Einsatz und verfahren in gleicher Weise mit dem im Action-Set A_t enthaltenen Gewinner (Schritte 8.1 bis 8.3).
Am Ende jeder Episode werden die akkumulierten Änderungen der Klassifiziererpositionen übernommen (Schritt 8.4) und das Action-Set A_t wird geleert (Schritt 8.5).
Jeder Lernschritt endet mit der Inkrementierung (Schritt 9) des Lernschritt-Zähler t. Sofern nicht das Ende einer Lernepisode erreicht wurde, folgt unmittelbar der nächste Lernschritt, in dem die Hauptschleife erneut durchlaufen wird.

4.6.3 Terminierung

Nach jeder Lernepisode überprüft das HCS, ob ein Terminierungskriterium erfüllt ist (Schritt 11 des Algorithmus 4.12). Ist das der Fall, endet der Lauf des HCS, andernfalls beginnt die nächste Lernepisode.
Die Zahl möglicher Terminierungskriterien ist groß: Ein Lauf könnte zum Beispiel beendet werden, wenn eine vorgegebene Zahl aufeinander folgender Zustände richtig klassifiziert wurde. Im Falle des Einsatzes zur Funktionsapproximation könnte auch das Unterschreiten eines vorgegebenen Approximationsfehlers den Lauf beenden. Diese und ähnliche Kriterien haben jedoch den Nachteil, dass nicht sichergestellt werden kann, dass sie jemals erfüllt werden. Aus diesem Grund wird in dieser Arbeit stets das denkbar einfachste Terminierungskriterium angewandt: Ein Lauf endet nach einer zuvor festgelegten Anzahl von Lernepisoden.

4.6.4 Parameter des HCS

Das HCS verfügt über eine ganze Reihe von Parametern, die sein Verhalten und den Lernverlauf beeinflussen. Diese wurden im Laufe des Kapitels bereits an geeigneten Stellen eingeführt, wobei auch ihre Bedeutung erläutert wurde. Hier werden sie noch einmal in einer tabellarischen Übersicht zusammengefasst.
Neben den in Tabelle 4.2 aufgeführten skalaren Parametern können auch die Möglichkeiten zur Wahl zwischen alternativen Vorgehensweisen und Operatoren, die das HCS an einigen Stellen bietet, als Parametrisierung des Systems aufgefasst werden:
Die auszuführende Aktion kann rein zufällig oder unter Bevorzugung der besten Aktion bestimmt werden. Ferner besteht die Wahl zwischen topologie- und distanzbasierter Match-Set-Bildung. Es können skalare oder als Funktion des Lernumgebungszustandes

$N_{\max}$	maximale Größe der Population
p_{explr}	Explorationswahrscheinlichkeit (bevorzugende Exploration)
ξ_0	Offset-Parameter für die Vorhersagenberechnung
β_0	Lernrate für die Gewichte der Vorhersagenberechnung
β	Lernrate für die Klassifizierer-Attribute, außer Subsumtionsradius
β_r	Lernrate für den Subsumtionsradius
ε_0	Fehlerschranke unterhalb der ein Klassifizierer als vollkommen genau gilt
α	Abfallrate für die Genauigkeitsberechnung
ν	Exponent für die Genauigkeitsberechnung
γ	Diskontierungsfaktor für die Returnabschätzung
p_{initial}	initiale Vorhersage neu erzeugter Klassifizierer
$\varepsilon_{\text{initial}}$	initialer Vorhersagefehler neu erzeugter Klassifizierer
f_{initial}	initiale Fitness neu erzeugter Klassifizierer
r_{initial}	initialer Subsumtionsradius neu erzeugter Klassifizierer
Θ_{DEL}	Erfahrungsschwellenwert für das Löschen von Klassifizierern
δ	Bruchteil der durchschnittlichen Fitness der Population, den die Fitness eines Klassifizierers unterschreiten muss, um bei der Berechnung seines Löschvotums berücksichtigt zu werden
Θ_{GA}	Schwellenwert für die Ausführung des Genetischen Algorithmus
p_{GA}	Wahrscheinlichkeit für die Ausführung des Genetischen Algorithmus
τ	Anteil des Action-Sets, der an einem Selektionsturnier teilnimmt
p_{cross}	Rekombinationswahrscheinlichkeit
p_{mut}	Mutationswahrscheinlichkeit für die Mutation eines Allels
inc_{mut}	Parameter für die Verteilung der Mutationsinkremente
ν_σ	Exponent der Sharing-Funktion
σ	Sharing-Radius
η	Lernrate für die Anpassung der Gewichtsvektor-Bedingungen
η_0	minimale Lernrate für die Anpassung der Gewichtsvektor-Bedingungen
$age_{\max}$	maximales Kantenalter
k	Anzahl ins Match-Set zu übernehmender Klassifizierer (distanzbasierte Match-Set-Bildung)
μ	Regularisierungsparameter für die Berechnung der Systemvorhersagen
ε_p	Fehlerschranke für die Berechnung der Systemvorhersagen

Tabelle 4.2: Die Parameter des HCS

berechnete Klassifizierervorhersagen verwendet werden. Am variabelsten jedoch ist der Genetische Algorithmus. In der im Rahmen dieser Arbeit erstellten Implementation des HCS stehen drei Crossing-Over-Operatoren sowie je zwei Selektions-und Mutationsmethoden zur Auswahl, aber auch der Einsatz weiterer Varianten ist denkbar. Schließlich kann optional Fitness-Sharing verwendet werden.

4.6.5 Lernumgebung

Die Verwendung von Gewichtsvektoren zur Repräsentation von Klassifiziererbedingungen impliziert, dass auch die Zustände der Lernumgebungen, in denen das HCS eingesetzt werden soll, durch reellwertige (Zustands-) Vektoren repräsentiert sein müssen. Für den Einsatz in binär kodierten Lernumgebungen, wie klassische Lernende Klassifizierende Systeme sie benutzen, ist das HCS daher nicht geeignet[26]. Bei den einzelnen Zustandsvariablen – den Einträgen der die Lernumgebungszustände darstellenden Vektoren – kann es sich im Prinzip um beliebige reelle Zahlen handeln. Da jedoch bei der Bestimmung der auf einen Zustand anwendbaren Klassifizierer die Bestimmung von euklidischen Distanzen zwischen Zustands- und Gewichtsvektoren eine zentrale Rolle spielt, sollten sich alle Zustandsvariablen in der gleichen Größenordnung bewegen, damit nicht eine Variable mit „großen“ Werten die Abstandsbestimmung dominiert. Im Folgenden wird daher, sofern nicht explizit anders angegeben, davon ausgegangen, dass alle Zustandsvariablen nur Werte im halboffenen Intervall $[0, 1)$ annehmen. Der Zustandsraum ist somit $\mathcal{S} = [0, 1)^n$ mit einer durch die jeweilige Lernumgebung festgelegten Zustandslänge n. Ebenfalls durch die Lernumgebung vorgegeben wird die Aktionsmenge $\mathcal{A} = \{a_1, a_2, \ldots, a_m\}$ und damit die Anzahl m zur Verfügung stehender Aktionen.

4.7 Vergleich mit anderen neuroevolutionären Ansätzen

Die Idee, neuronale und evolutionäre Ansätze miteinander zu kombinieren und so dem Vorbild der Natur zu folgen, die dies mit ausgesprochen überwältigenden Ergebnissen getan hat, ist keinesfalls neu. In der Literatur finden sich zahlreiche Beispiele für den Einsatz Genetischer Algorithmen zur Optimierung der Verbindungsgewichte von Neuronen in einer fest vorgegebenen (Feed-Forward[27]-) Netzstruktur sowie zur Optimierung der Verbindungsstruktur Neuronaler Netze dieses Typs. Ein Überblick über solche Ansätze wird zum Beispiel in [Schaffer u. a. 1992; Nolfi u. Floreano 1999] gegeben. Auch zur Optimierung der Topologie Selbstorganisierender Karten wurden Genetische Algorithmen bereits eingesetzt [Polani 1996]. Verfahren allerdings, die – wie das in dieser Arbeit entwickelte HCS – einen neuronalen Ansatz in ein Lernendes Klassifizierendes System integrieren, sind selten, dem Autor dieser Arbeit sind lediglich zwei derartige Ansätze bekannt. Auf diese sowie einen weiteren neuroevolutionären Ansatz, bei dem es sich jedoch nicht um ein Lernendes Klassifizierendes System handelt, wird im Folgenden kurz eingegangen, um aufzuzeigen, inwiefern sich das HCS von ihnen unterscheidet.

[26] Es ist jedoch immer möglich, zu einer binär kodierten Lernumgebung eine reellwertige „Übersetzung“ anzugeben, in der das HCS eingesetzt werden könnte. Ein Beispiel hierfür ist die in Kapitel 6 betrachtete Multiplexer-Lernumgebung. Ob ein solches Vorgehen im Allgemeinen sinnvoll ist, ist jedoch fraglich.

[27] Nähere Erläuterungen zu diesem Netztypus finden sich zum Beispiel in [Haykin 1999].

X-NCS

X-NCS [O'Hara 2006] ersetzt die Bedingungs-Aktions-Regel herkömmlicher Klassifizierer durch eine neuronale Regel in Form eines Multi-Layer-Perceptrons [siehe z.B. Haykin 1999]. Dieses besteht aus einer Eingabeschicht mit einer der Dimension des Zustandsraumes entsprechenden Zahl von Neuronen, einer versteckten Schicht und einer Ausgabeschicht, die ein Neuron mehr enthält, als es Aktionen in der Lernumgebung gibt. Eines der Ausgabeneuronen dient dazu, die Match-Set-Zugehörigkeit des Klassifizierers zu bestimmen, den übrigen ist jeweils eine der möglichen Aktionen zugeordnet. In jedem Lernschritt berechnen die neuronalen Regeln aller Klassifizierer – mit dem aktuellen Lernumgebungszustand als Eingabe – die Ausgabewerte ihrer Ausgabeneuronen. Hat das Match-Set-Zugehörigkeits-Neuron den höchsten Ausgabewert, so gehört der Klassifizierer nicht dem Match-Set an, andernfalls ist er in diesem enthalten und vertritt die dem Ausgabeneuron mit dem höchsten Ausgabewert zugeordnete Aktion. Ein Klassifizierer kann also in verschiedenen Zuständen verschiedene Aktionen vertreten.
In [O'Hara 2006] wird X-NCS sowohl in einer Variante verwendet, die keine Adaptation der neuronalen Regeln vorsieht, als auch in solchen, die einen Backpropagation-Algorithmus nutzen, um die Verbindungsgewichte der neuronalen Regeln der in einem Action-Set enthaltenen Klassifizierer anzupassen. Dabei werden als Soll-Ausgaben unter anderem die Ausgabewerte der neuronalen Regel desjenigen im jeweiligen Action-Set enthaltenen Klassifizierers verwendet, der die höchste Fitness aufweist.
Abgesehen von der Match-Set-Bildung und der Adaptation der neuronalen Regeln, arbeitet X-NCS wie das XCS, wobei natürlich die Discovery-Mechanismen (Covering-Operator und Genetischer Algorithmus) an die Gestalt der Klassifizierer angepasst werden müssen.
Der X-NCS-Ansatz ist dem HCS insofern ähnlich, als auch er auf dem XCS aufbaut und dessen grundlegende Struktur übernimmt. Funktionsweise und interne Abläufe des Systems werden beim HCS allerdings deutlich stärker modifiziert. Als Hauptunterschied ist – abgesehen davon, dass X-NCS und HCS ganz unterschiedliche Typen Neuronaler Netze verwenden – anzuführen, dass jedes Individuum in der Population des X-NCS ein vollständiges Neuronales Netz darstellt, während das HCS einzelne Neuronen evolviert.

HLS

Das *Hybrid Learning System* [Ball 1994; Warwick u. Ball 1996] weist nur wenig Ähnlichkeit mit einem herkömmlichen Lernenden Klassifizierenden System auf. So ist schon die Interaktion mit der Lernumgebung auf die Wahrnehmung von Zuständen und die Ausführung von Aktionen beschränkt. Ein Feedback in Form von Belohnungen, die die Lernumgebung gewährt, ist nicht vorgesehen. Die Qualität eines Klassifizierers wird dementsprechend auch nicht daran gemessen, welche Returns seiner Aktivierung folgen, oder daran, wie genau er diese vorhersagt. Vielmehr bemisst sich die Fitness eines Klassifizierers daran, wie effektiv er Unterschiede zwischen beobachteten Lernumgebungszuständen und Zielzuständen, die das System ausgehend von einem internen Modell der Lernumgebung bestimmt, verringert.
Eine (modifizierte) Kohonenkarte dient in Gestalt des sogenannten *Feature Correlation Network* als Gedächtnis des HLS, das basierend auf den während des Lernvorganges gemachten Beobachtungen angepasst wird. Das Feature Correlation Network korreliert Zustände der Lernumgebung mit der Erfüllung interner Zielvorgaben des Systems. Diese Karte wird – vereinfacht ausgedrückt – in jedem Lernschritt verwendet, um zum aktuellen Zustand der Lernumgebung einen (möglichst ähnlichen) Zielzustand zu bestimmen,

durch dessen Erreichen die internen Zielvorgaben erfüllt würden[28].
Auch um zu bestimmen, welcher Klassifizierer in einem Lernschritt aktiviert wird, werden Kohonenkarten verwendet: Jeder Klassifizierer enthält pro Zustandsvariable eine Kohonenkarte mit zweidimensionalen Gewichtsvektoren. Diese werden verwendet, um den Zustand der Lernumgebung nach Aktivierung des Klassifizierers im aktuellen Zustand vorherzusagen: In jeder der Karten wird das Neuron bestimmt, dessen Gewichtsvektor den geringsten Abstand zu dem Paar aus dem aktuellen Wert der jeweiligen Zustandsvariable und deren durch den Zielzustand gegebenen Wert aufweist. Der jeweils zweite Eintrag des Gewichtsvektors eines Gewinners wird interpretiert als vorhergesagter Wert der Zustandsvariable nach Aktivierung des Klassifizierers. Aktiviert wird dann der Klassifizierer, dessen vorhergesagter Zustand am wenigsten vom gewünschten Zielzustand abweicht. Die Kohonenkarten aktivierter Klassifizierer werden unter Verwendung von Paaren aus Werten von Zustandsvariablen vor und nach ihrer Aktivierung trainiert. In den angegebenen Veröffentlichungen zum HLS wird leider nichts Näheres zur Funktionsweise des Genetischen Algorithmus des Systems gesagt.
Das HLS evolviert komplette Netze – genauer gesagt sogar Mengen von Netzen – nicht einzelne Neuronen wie das HCS. Ein weiterer wesentlicher Unterschied ist der Verzicht auf ein Feedback der Lernumgebung zugunsten der Entwicklung eines internen Modells der Umwelt. Dies ist nur möglich, indem das System Zustände der Lernumgebung anhand interner, für die jeweilige Lernumgebung spezifischer Zielvorgaben bewertet, die dem System vorgegeben werden müssen. Dass die in einem Klassifizierer enthaltenen Kohonenkarten basierend auf Paaren von Zuständen vor und nach dessen Aktivierung trainiert werden, setzt offenbar voraus, dass diese Zustände voneinander abhängen. Somit scheint die Anwendbarkeit des Ansatzes auf Probleme beschränkt, deren Lernepisoden – zumindest in der Regel – länger als ein Zeitschritt sind. Während ein Einsatz zur Klassifikation wohl noch möglich ist – etwa indem die Umgebung nach einer richtigen Klassifikation in einen ‚Richtig'-Zustand übergeht, dessen Erreichen als Ziel vorgegeben wird –, scheint ein Einsatz zur Funktionsapproximation ausgeschlossen, da das HLS nur Zustände und Aktionen kennt, diese aber nicht mit einem (Funktions-) Wert verbindet.

SANE

Sowohl X-NCS als auch HLS evolvieren – wie nahezu alle neuroevolutionären Ansätze – komplette Neuronale Netze. Eines der wenigen Verfahren, die wie das HCS einzelne Neuronen evolvieren ist SANE [Moriarty u. Miikkulainen 1998], bei dem es sich aber nicht um ein Lernendes Klassifizierendes System handelt. SANE evolviert Neuronen eines Feed-Forward-Netzes. Um die Fitness der in der Population enthaltenen Neuronen zu bestimmen, werden einige von ihnen in durch eine sogenannte Blaupause spezifizierter Weise zu einem Netz zusammengeschaltet, dessen Leistung beim Einsatz in der betrachteten Lernumgebung dann bewertet wird. Dies wird für alle Elemente einer – ebenfalls evolvierten – Blaupausen-Population wiederholt. Als Fitness eines Neurons dient die durchschnittliche Bewertung der Netze, in denen es eingebaut war. Außer durch den Typ der evolvierten Neuronen unterscheidet SANE sich vor allem dadurch vom HCS, dass zum einen die evolvierten Neuronen kein feststehendes Netz bilden, sondern nur zur Evaluation zu Netzen zusammengebaut werden, und zum anderen das Lernen rein evolutionär erfolgt, also kein Netzlernverfahren verwendet wird.

[28] Dies lässt sich anhand eines Beispiels verdeutlichen: Beschreibt etwa die Lernumgebung einen Wald, in dem ein durch das HLS gesteuerter Agent nach Futter sucht, so könnte ein Zustand durch eine Positionsangabe (und eine Information, ob an dieser Stelle Futter zu finden ist) beschrieben sein. Eine naheliegende interne Zielvorgabe wäre es, möglichst schnell Futter zu finden. Als Zielzustand würde dann ein dem aktuellen Zustand möglichst naher Zustand der Lernumgebung bestimmt, in dem schon Futter gefunden wurde.

4.8 Zusammenfassung

In diesem Kapitel wurde das Hybride Lernende Klassifizierende System (HCS) eingeführt. Da dieses in Anlehnung an das XCS entwickelt wurde, gleicht seine – in Abbildung 4.2 dargestellte – Struktur der des XCS.

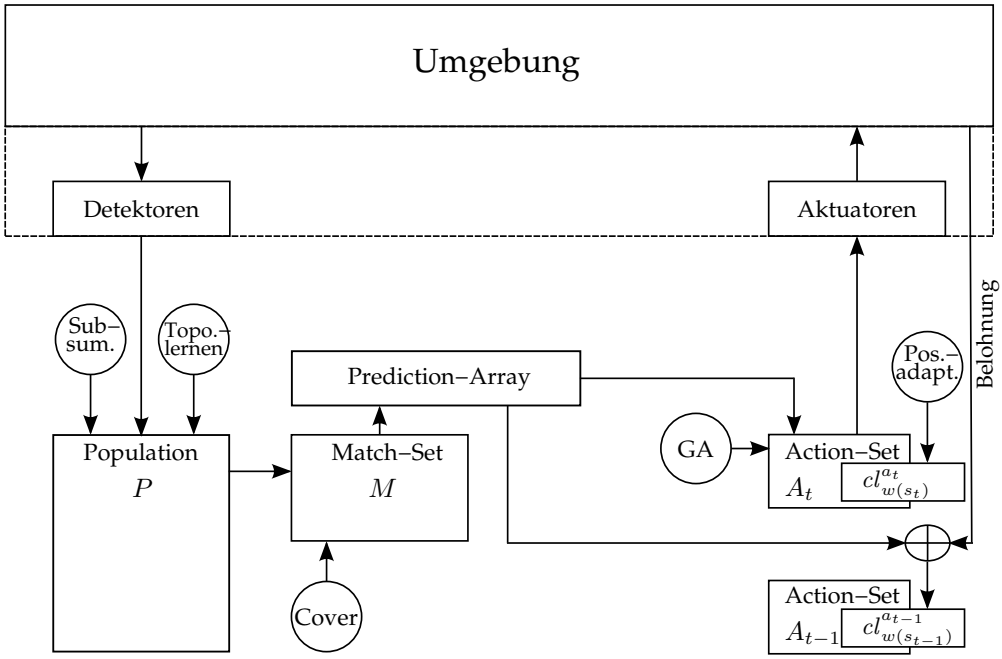


Abbildung 4.2: Schematische Darstellung des HCS.

Die wesentlichen Unterschiede zwischen XCS und HCS ergeben sich aus der Integration von drei Elementen Selbstorganisierender Karten in das HCS:
Erstens wurden die Klassifiziererbedingungen klassischer Lernender Klassifizierender Systeme durch Gewichtsvektoren ersetzt. Dies hat zur Folge, dass der Matchbereich eines Klassifizierers – die Menge von Zuständen, in denen er anwendbar ist – nicht unveränderlich ist, sondern sich während des Lernvorganges verändert, etwa durch das Einfügen oder Löschen von Klassifizierern in die respektive aus der Population. Die Verwendung von Gewichtsvektor-Bedingungen impliziert ferner, dass es zu jedem Lernumgebungszustand stets nur einen passenden Klassifizierer gibt, sodass die Match-Set-Bildung – wobei die Bezeichnung ‚Match-Set' nicht wirklich zutreffend ist – und die Berechnung der Systemvorhersagen in gänzlich anderer Weise erfolgen müssen als etwa beim XCS. Aus dem gleichen Grund werden auch nicht die Attribute aller in einem Action-Set enthaltenen Klassifizierer angepasst, sondern nur die des in diesem enthaltenen, zum zuvor betrachteten Lernumgebungszustand passenden Gewinners.
Als zweites von Selbstorganisierenden Karten inspiriertes Element wurde das Topologielernen, wie es etwa das (Wachsende) Neurale Gas verwendet, übernommen – mit dem Ziel, die bei der Berechnung der Systemvorhersagen zu berücksichtigenden Klassifizierer basierend auf topologischen Strukturen zu bestimmen.
Schließlich wurde noch ein an die Lernregeln Selbstorganisierender Karten angelehnter Lernmechanismus eingeführt, der die Feinanpassung der evolvierten Klassifizierer übernehmen soll, während der Genetische Algorithmus hauptsächlich für die globale Ausbreitung der Population – also für die Exploration des Zustandsraums – zuständig ist.

Kapitel 5

Implementation eines Simulators zur Untersuchung des HCS

Lernende Klassifizierende Systeme und Selbstorganisierende Karten weisen, jeweils für sich genommen, bereits eine erhebliche Komplexität auf. Diese macht eine umfassende theoretische Analyse des Verhaltens dieser Systeme, beispielsweise mit mathematischen Mitteln, praktisch unmöglich[1]. Dies gilt umso mehr für ein System wie das im vorangegangenen Kapitel eingeführte Lernverfahren, das Charakteristika beider Ansätze koppelt. Zentral für die Untersuchung derartiger komplexer Systeme sind Simulationen, die unter Zuhilfenahme geeigneter Software durchgeführt werden.
Dieses Kapitel befasst sich mit der im Rahmen der vorliegenden Arbeit entwickelten Simulationssoftware für das in Kapitel 4 beschriebene System, mit der die in den Kapiteln 6 bis 8 noch zu beschreibenden Experimente durchgeführt wurden.
Abschnitt 5.1 listet zunächst die zentralen Anforderungen an die Simulationsumgebung auf, mit deren Umsetzung im generellen Entwurf der Software sich der Abschnitt 5.2 befasst. In Abschnitt 5.3 wird die Auswahl der Programmiersprache C# für die Entwicklung des Simulationssystems motiviert. Abschnitt 5.4 geht auf das Nutzungskonzept und die Benutzeroberfläche der erstellten Software ein, bevor das Kapitel in Abschnitt 5.5 mit einer kurzen Zusammenfassung schließt.

5.1 Anforderungen

Der erste Schritt auf dem Weg zur Erstellung jeder Software sollte die genaue Spezifizierung der an diese gestellten Anforderungen sein. Da jedoch eine vollständige Spezifikation des hier benötigten Simulators umfangreicher als die vorliegende Arbeit wäre,

[1] Auf diesen Umstand ist – zumindest in Teilen – die auf die Entwicklung des Holland'schen Systems folgende, fast vollständige Stagnation in der der Entwicklung Lernender Klassifizierender Systeme zurückzuführen, die erst durch Einführung des stark vereinfachten ZCS überwunden wurde.

werden hier, im Sinne einer Produktvision, nur die wichtigsten Anforderungen dokumentiert.

5.1.1 Durchführbare Experimente

Das Hauptziel bei der Erstellung des Simulators für das HCS war es, ein Werkzeug zur Verfügung zu stellen, mit dem alle im Rahmen dieser Arbeit nötigen Experimente mit diesem in komfortabler Weise durchgeführt werden können. Dabei ist zu unterscheiden zwischen Einzelläufen einerseits und Serienläufen andererseits; diese beiden Kategorien stellen gänzlich unterschiedliche Anforderungen an die Simulationssoftware:

Einzelläufe

Einzelläufe dienen in erster Linie dazu, dem Benutzer einen *unmittelbaren* Eindruck vom Verhalten des simulierten Systems zu geben. Hieraus ergeben sich die folgenden Forderungen:

E.1 Die Simulationssoftware muss es ermöglichen, einen Simulationslauf zu konfigurieren, zu starten und zu beenden.

E.2 Um den Einfluss verschiedener Parameter auf das Verhalten des Lernenden Klassifizierenden Systems zu untersuchen, soll es möglich sein, alle Parameter des Lernenden Klassifizierenden Systems interaktiv – auch während eines Simulationslaufs – zu verändern

E.3 Die Simulationssoftware muss dem Benutzer einen möglichst guten Einblick in die ablaufenden (Lern-)Prozesse gewähren. Ein zentraler Bestandteil soll daher die graphische Darstellung der zeitlichen Entwicklung bestimmter Kenngrößen sein[2].

E.4 In vielen Fällen ist es auch aufschlussreich, die Verteilung der Klassifizierer im Reizraum und ihre Nachbarschaftsbeziehungen zu untersuchen. Daher soll deren visuelle Inspektion in der Benutzeroberfläche des Simulators ermöglicht werden – in Form einer zwei- respektive dreidimensionalen Darstellung ähnlich den Abbildungen 3.6 und 3.7.

E.5 Um Klassifizierer und deren Attribute genauer in Augenschein nehmen zu können, soll eine tabellarische Darstellung der Klassifiziererpopulationen des Lernenden Klassifizierenden Systems integriert werden.

E.6 In vielen Reinforcement-Learning-Problemen ist nicht allein von Interesse, *ob* ein Agent ein Problem löst, sondern auch *wie* er dies tut (Läuft ein Agent auf direktem Weg zum Ziel oder macht er Umwege?). Daher sollen auch Lernumgebungen und die in diesen ausgeführten Aktionen visualisiert werden.

Die oben genannten Anforderungen implizieren die Verwendung einer graphischen Benutzeroberfläche für den Simulator.

Serienläufe

In Serienläufen werden *unbeaufsichtigt* mehrere Simulationsläufe durchgeführt, meist mit einer der beiden folgenden Zielsetzungen:

[2] Welche dies sind ist unter anderem vom Problemtypus abhängig, vergleiche hierzu die Kapitel 6 und 8.

- Indem mehrere Testläufe mit verschiedenen Werten eines Systemparameters unter ansonsten unveränderten Bedingungen durchgeführt werden, wird der Einfluss dieses Parameters auf das Lernverhalten des Systems untersucht.
- Die in dieser Arbeit betrachteten Systeme unterliegen, wie in den vorangegangenen Kapiteln gesehen, Zufallseinflüssen: Genetischer Algorithmus, zufällige Wahl der Probleminstanzen und zufällige Aktionswahl in der Lernphase. Um aussagekräftige Ergebnisse zu erhalten, ist es daher unumgänglich, stets mehrere Läufe mit gleichen Parametern aber anderer Initialisierung des Pseudozufallszahlengenerators durchzuführen, um durch Mittelung der interessierenden Kenngrößen zufallsbedingte Schwankungen zu reduzieren.

In beiden genannten Einsatzszenarien sind die Ergebnisse der einzelnen Simulationsläufe von nur untergeordnetem Interesse; wichtig ist vor allem das sich aus dem Vergleich beziehungsweise der Mittelung über die Einzelläufe ergebende Endresultat. Somit werden die bei Einzelläufen zentralen Möglichkeiten zur Interaktion und visuellen Inspektion bei Serienläufen nicht benötigt, sondern sind im Gegenteil sogar eher hinderlich, da sie Rechner-Ressourcen blockieren. Hieraus ergeben sich die folgenden Anforderungen:

S.1 Die Simulationssoftware muss es ermöglichen, skriptgesteuerte Serienläufe ohne Verwendung einer graphischen Benutzeroberfläche durchzuführen.

S.2 Die Ergebnisse der einzelnen Läufe eines Serienlaufs müssen für die spätere Verwendung in Dateien abgelegt werden. Dies sollte sinnvollerweise in einem sowohl menschenlesbaren wie auch zu gängigen Spreadsheet-Programmen (etwa Microsoft-Excel) kompatiblem Format geschehen, zum Beispiel im CSV-Format.

S.3 Im zweiten der oben genannten Einsatzszenarien ist eine zumindest teilweise automatisierte Auswertung von Serienläufen möglich und sollte in die Simulationssoftware integriert werden.

Die Erstellung einer Konfigurationsdatei für einen Serienlauf ist prinzipiell natürlich „von Hand“ möglich. Sinnvoller, da weniger fehleranfällig, ist jedoch eine Erstellung über ein spezielles Konfigurationsprogramm:

S.4 Die Konfiguration von Serienläufen soll in komfortabler Weise über eine graphische Benutzeroberfläche möglich sein.

5.1.2 Vergleich mit dem XCS

Da bei der Entwicklung eines neuen Lernenden Klassifizierenden Systems natürlich insbesondere auch der Vergleich mit bereits bestehenden Systemen von Interesse ist, ergeben sich die folgenden beiden Anforderungen an die Simulationssoftware:

V.1 Neben dem HCS soll auch eine Implementierung des derzeitigen Standardsystems, also des XCS, in den Simulator integriert werden.

V.2 Die Lernproblem-Instanzen, die einem Lernenden Klassifizierenden System präsentiert werden, werden zufällig erzeugt. Um einen fairen Vergleich der Lernerfolge verschiedener Systeme (oder des gleichen Systems mit verschiedenen Parametersätzen) zu gewährleisten, müssen diese jedoch mit jeweils den gleichen Probleminstanzen konfrontiert werden. Dies muss durch geeignete technische Maßnahmen ermöglicht werden.

5.1.3 Flexibilität

Obwohl das Simulationssystem in erster Linie entwickelt wird, um im Rahmen dieser Arbeit mit dem HCS zu experimentieren und seine Lernerfolge mit denen des XCS zu vergleichen, soll die Möglichkeit offengehalten werden, auch zukünftige Varianten dieses Systems oder andere auf dem XCS aufbauende Lernende Klassifizierende Systeme zu integrieren. Ebenso soll auch das Einbinden zusätzlicher Lernumgebungen mit geringem Aufwand möglich sein:

I.1 Für die Simulationssoftware ist eine Struktur zu wählen, die das Einbinden weiterer Lernender Klassifizierender Systeme und Lernumgebungen ermöglicht, ohne größere Änderungen am bestehenden Quellcode zu erfordern[3].

Zuletzt ist noch ein Punkt zu nennen, der dezidiert *keine* Anforderung an die zu erstellende Software darstellt: Im Hinblick auf die geforderte Flexibilität des Simulators soll hier kein Wert auf eine Laufzeitoptimierung gelegt werden, da eine solche sich bei vielen vorliegenden Implementierungen von Verfahren aus dem Bereich des Maschinellen Lernens und der Künstlichen Intelligenz negativ auf deren Wartbarkeit und Erweiterbarkeit auswirkt. Dies ist vertretbar, da sich die Qualität eines Lernverfahrens eher in der Anzahl benötigter Lernschritte bis zum Erreichen eines Lernerfolges zeigt, als in der dafür benötigten Rechenzeit.

5.2 Entwurf der Simulationsumgebung

Ausgehend von den im vorangegangenen Abschnitt formulierten Anforderungen musste zunächst ein Konzept für den generellen Aufbau der Simulationsumgebung entwickelt werden, das insbesondere der Anforderung I.1 gerecht wird. Das Ergebnis der hierzu angestellten Überlegungen zeigt die Abbildung 5.1. Im Folgenden werden die einzelnen Bestandteile der dargestellten Struktur kurz erläutert. Auf die interne Organisation der einzelnen Komponenten wird dabei nicht näher eingegangen, da dies nur im Rahmen einer langwierigen und recht technischen Diskussion möglich wäre, die das Verständnis der Gesamtstruktur jedoch nur unwesentlich verbessern würde.

Der Simulatorkern

Die in Abbildung 5.1 als Szenario, LCS, Environment und Basis bezeichneten Komponenten bilden den eigentlichen Kern der Simulationsumgebung. Die Basis kapselt dabei die von den Lernenden Klassifizierenden Systemen und den Lernumgebungen gemeinsam genutzten Datenstrukturen, das sind im Wesentlichen die Repräsentationen von Zuständen (der Lernumgebung) und Aktionen.
Alle Lernumgebungen werden als Spezialisierungen einer abstrakten Basisklasse umgesetzt und in der Environment-Komponente zusammengefasst. Die Schnittstelle, über die ein Lernendes Klassifizierendes System mit seiner Lernumgebung interagiert, wird vollständig von dieser Basisklasse spezifiziert – somit ist das Einpassen weiterer Lernumgebungen einfach durch Ableiten einer neuen spezialisierten Klasse möglich.
Die LCS-Komponente umfasst die implementierten Lernenden Klassifizierenden Systeme (XCS und HCS) und die von diesen benötigten Datenstrukturen: Bedingungen, Klas-

[3] Dies bezieht sich in erster Linie auf den eigentlichen Simulator; Änderungen an der Oberfläche, beispielsweise in Auswahldialogen, sind bei Erweiterungen kaum zu vermeiden.

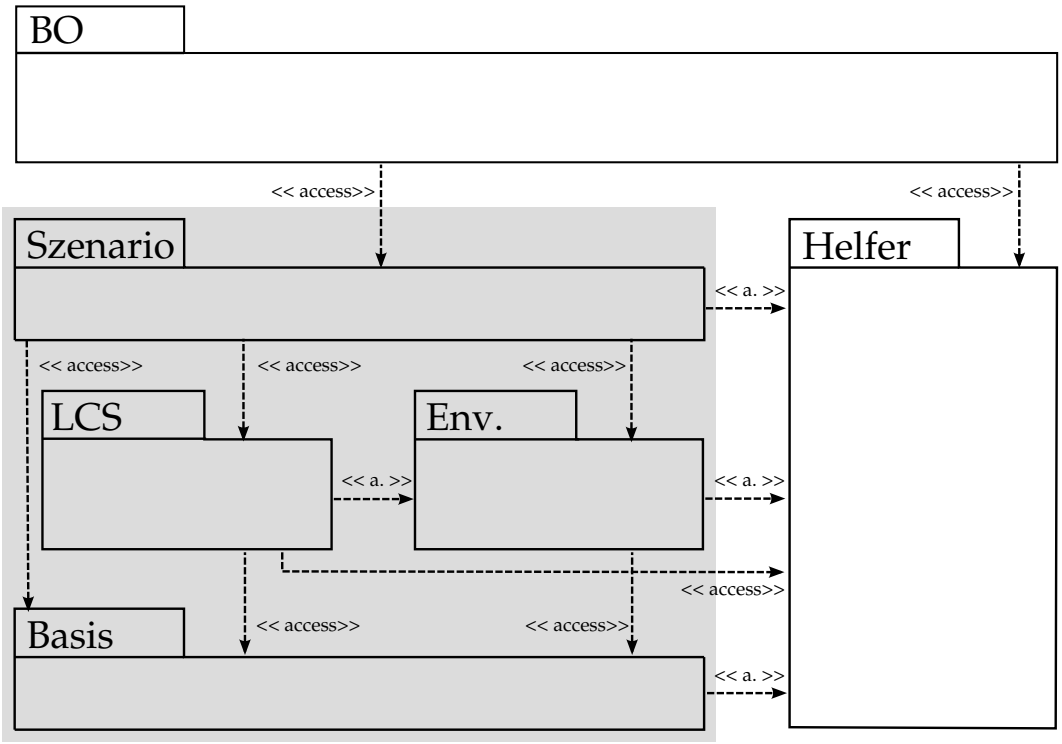


Abbildung 5.1: Die Struktur der Simulationsumgebung. Die grau hinterlegten Komponenten bilden den eigentlichen Simulatorkern.

sifizierer, Populationen und Parametersätze. Auch hier wird die Erweiterbarkeit mittels objektorientierter Mechanismen wie Vererbung und Polymorphie sowie durch die Verwendung geeigneter Entwurfsmuster sichergestellt.
Die Szenario-Komponente enthält eine Klasse zur Definition von Lernszenarien. Ein Lernszenario fasst dabei eine Lernumgebung und ein Lernendes Klassifizierendes System zusammen. Die Szenario-Klasse dient ferner als Schnittstelle zwischen dem Simulatorkern und der Benutzeroberfläche.

Die Benutzerschnittstelle

Die als Benutzerschnittstelle (BO) bezeichnete Komponente enthält zum einen die komplette graphische Benutzeroberfläche, auf die in Abschnitt 5.4 noch näher eingegangen wird und zum anderen ein Konsolenprogramm für die Steuerung des Simulators bei der Durchführung von Serienexperimenten.

Die „Helfer"-Komponente

Die in Abbildung 5.1 so bezeichnete Helfer-Komponente unterstützt die anderen Komponenten; sie bildet ein „Sammelbecken" für all jene benötigten Funktionalitäten, die keiner der anderen Komponenten in naheliegender und eindeutiger Weise zuzuordnen sind.

5.3 Auswahl einer Entwicklungssprache und -umgebung

Die oben beschriebene Grundstruktur des zu entwickelnden Simulationssystems legte die Verwendung einer objektorientierten Programmiersprache nahe, schränkte die Wahl dieser Sprache jedoch nicht weiter ein. Daher wurde zunächst überprüft, ob die Möglichkeit besteht, bereits existierende Implementierungen – beispielsweise des XCS oder Selbstorganisierender Karten – wiederzuverwenden und in den Simulator zu integrieren. Bei dieser Recherche zeigte es sich, dass zwar einige solcher Implementationen existieren und auch im Quellcode verfügbar sind[4], für eine Wiederverwendung jedoch nur sehr eingeschränkt brauchbar sind. Dies ist begründet in der Vielzahl der verwendeten Programmiersprachen und dem naturgemäß nicht aufeinander abgestimmten Entwurf der Programme verschiedener Autoren. Somit fiel die Entscheidung zugunsten einer vollständigen Neuimplementierung aller benötigten Algorithmen.

In der Folge wurde – hauptsächlich aufgrund persönlicher Präferenz des Autors – die noch recht junge Programmiersprache C# für die Umsetzung der Simulationsumgebung gewählt. C# wurde als grundständige objektorientierte Sprache entwickelt und setzt daher viele Konzepte der objektorientierten Programmierung sauberer um, als die im Bereich des Maschinellen Lernens und der Künstlichen Intelligenz weit verbreitete Programmiersprache C++, die als Erweiterung der imperativen Sprache C aufgefasst werden kann. Zudem entbinden C# beziehungsweise die zugehörige Laufzeitumgebung, das Microsoft .NET-Framework, den Benutzer von einer Reihe technischer Aufgaben, etwa durch die automatische Garbage Collection sowie die automatische Serialisierung (Speicherung) auch komplexer Datenstrukturen, was das Arbeiten mit dieser Sprache sehr angenehm macht.
Entscheidend für die Wahl von C# war auch die Verfügbarkeit einer komfortablen Entwicklungsumgebung, in der insbesondere die geforderte graphische Benutzeroberfläche möglichst leicht realisiert werden konnte: Die Entwicklung der Simulationsumgebung wurde mit dem Borland C#-Builder 1.0 begonnen; mit Veröffentlichung der Sprachversion 2.0 von C# wurde jedoch auf die Entwicklungsumgebung Microsoft Visual Studio C# Express 2005/2008 gewechselt, um neu hinzu gekommene Sprachfeatures, etwa generische Typen, verwenden zu können.
Bei diesem Wechsel zeigte sich ein weiterer Vorteil von C#: Die bei der Entwicklung der Oberfläche zum Einsatz kommenden Bedienelemente, wie Knöpfe, Eingabefehler und Dialoge zum Öffnen und Speichern von Dateien werden vom Microsoft .NET-Framework zur Verfügung gestellt; die einmal erstellte Oberfläche „übersteht“ somit den Wechsel der Entwicklungsumgebung problemlos. Im Gegensatz dazu ist beispielsweise die vom Borland C++-Builder für die Erstellung von Oberflächen benutzte Visual Component Library spezifisch für diese Entwicklungsumgebung.
Ein klarer Nachteil von C# ist seine Plattformabhängigkeit, eine vollständige Umsetzung des Sprachstandards existiert bisher nur auf neueren Microsoft-Windows-Plattformen. Allerdings sind Projekte zur Portierung auf andere Plattformen, etwa Linux, bereits angelaufen [Mono; DotGNU].

[4] Beispielsweise liegen eine C-Implementation des XCS [Butz 1999] und eine JAVA-Implementation des Wachsenden Neuralen Gases [Fritzke u. Loos 1998]vor.

5.4 Überblick über die Simulationssoftware

Dieser Abschnitt beschreibt in groben Zügen das Nutzungskonzept der erstellten Simulationssoftware. Der Schwerpunkt liegt dabei auf der für die Durchführung von Einzelläufen verwendeten Variante. Auf die für die Durchführung von Serienläufen erstellte Programmversion ohne graphische Oberfläche wird nur kurz eingegangen, da der Benutzer nicht direkt mit ihr arbeitet. Ferner wird auch die als eigenständiges Programm umgesetzte Konfiguration von Serienläufen vorgestellt.

5.4.1 Durchführung von Einzelläufen

Zunächst soll nun die die Durchführung von Einzelläufen unterstützende Variante der Simulationssoftware respektive deren Benutzeroberfläche betrachtet werden. Einer kurzen Diskussion des der Benutzeroberfläche zugrunde liegenden Konzepts folgend werden die verschiedenen Teile der Oberfläche und deren Aufgaben beschrieben.

Konzeption der Benutzeroberfläche

Wie bereits in Abschnitt 5.1 erwähnt, soll die Oberfläche dem Benutzer einen möglichst vielfältigen Einblick in die ablaufenden Prozesse gewähren und ihm ferner auch eine direkte Interaktion – in Form der Variation von Parametern des Lernenden Klassifizierenden Systems – ermöglichen. Diese Anforderungen stehen aufgrund der Vielfalt der zu integrierenden Darstellungsformen und der großen Zahl beeinflussbarer Parameter im Widerspruch zu einer ganz allgemeingültigen Erwartung an die Gestaltung einer Benutzeroberfläche: dem Wunsch nach größtmöglicher Übersichtlichkeit.
Daher wurden die diversen Elemente der Oberfläche auf mehrere Fenster verteilt, die mit den in 5.1.1 genannten Anforderungen E.1 bis E.6 korrespondieren. Das Hauptfenster, die zentrale Komponente der Benutzeroberfläche, setzt zwei dieser Anforderungen um – neben den Elementen zur grundlegenden Steuerung (Starten, Pausieren, Stoppen, ...) von Einzelläufen (E.1) enthält es die Darstellung der zeitlichen Entwicklung von Kenngrößen des Lernverlaufs (E.3). Ferner gestattet es auch die Auswahl der anzuzeigenden weiteren Fenster: Alle zur Konfiguration der Parameter des Lernenden Systems (E.2) nötigen Steuerelemente wurden in einem eigenen Fenster konzentriert; auch für die visuelle (E.4) und tabellarische (E.5) Darstellung der Klassifizierer sowie die Lernumgebungsvisualisierung (E.6) wurden jeweils eigene Fenster entworfen. Daneben existiert noch eine Reihe kleinerer Dialoge, auf die hier jedoch nicht weiter eingegangen werden soll, da sie selbsterklärend sind.

Das Hauptfenster

Das Hauptfenster (siehe Abbildung 5.2) ist die zentrale Komponente der Benutzeroberfläche. Die Steuerung der Simulationsumgebung erfolgt zum größten Teil über die Menüleiste dieses Fensters: Über das Datei-Menü erfolgt die Konfiguration von Einzelläufen, das Lauf-Menü gestattet deren Steuerung. Im Ansicht-Menü können die weiteren Fenster der Simulationsumgebung je nach Bedarf ein- und ausgeblendet werden. Im Options-Menü schließlich finden sich Einstellmöglichkeiten für Parameter der Simulationsumgebung, etwa für die Konfiguration der Pseudozufallszahlengeneratoren.

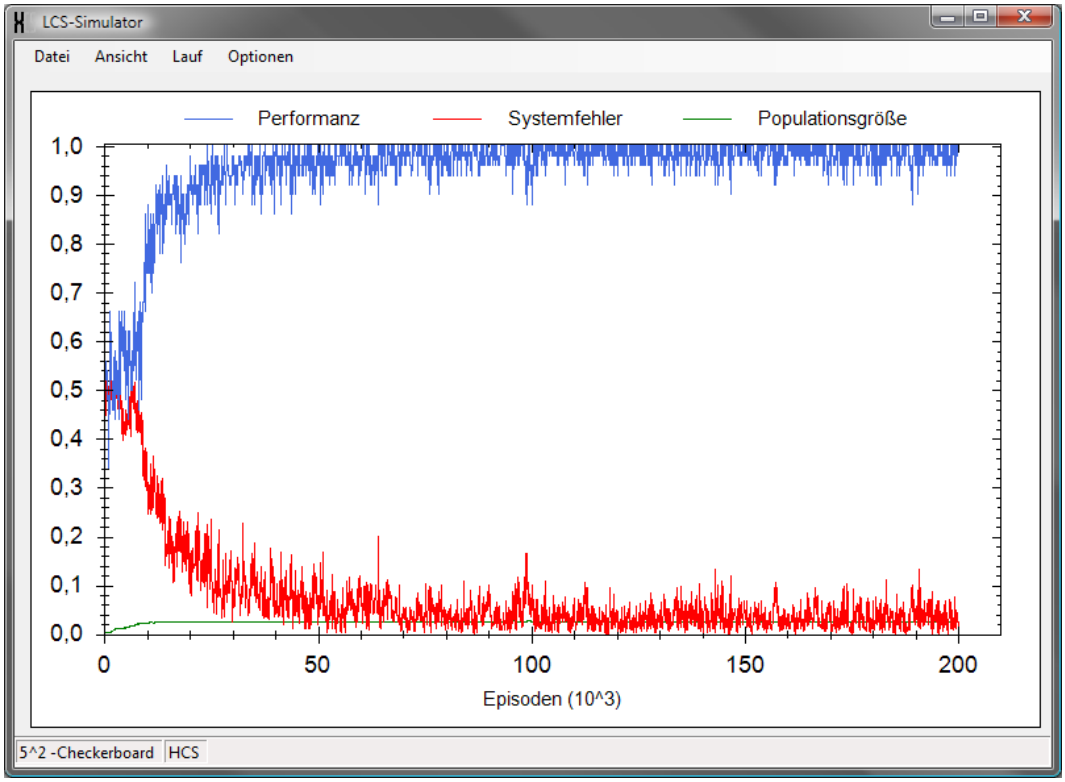


Abbildung 5.2: Das Hauptfenster der graphischen Benutzeroberfläche.

Ferner enthält das Hauptfenster eine Graphikkomponente[5], in der wesentliche Kenngrößen des Lernverlaufs in ihrer zeitlichen Entwicklung dargestellt werden. In der Statusleiste am unteren Rand des Hauptfensters werden Informationen zur aktuell gewählten Lernumgebung sowie dem aktiven Lernenden Klassifizierenden System angezeigt.

Variation von Parametern

Das Parameterfenster (siehe Abbildung 5.3) stellt alle Parameter des aktiven Lernenden Klassifizierenden Systems dar und erlaubt deren Bearbeitung. Sowohl das Ändern einzelner Parameter wie auch das Laden eines kompletten Parametersatzes aus einer Datei ist möglich. Um das Parameterfenster möglichst übersichtlich zu gestalten, sind die einzelnen Parameter gemäß ihrer Bedeutung gruppiert, etwa die Initialwerte für die Attribute neu erzeugter Klassifizierer oder die für die Genauigkeitsberechnung maßgeblichen Parameter.

Visuelle Darstellung der Lernumgebung

Insbesondere im Falle von Reinforcement-Learning-Problemen mit verzögerten Belohnungen kann es sinnvoll sein, die Auswirkungen der Aktionen des Lerners in der Lernumgebung zu betrachten. Für Lernumgebungen, bei denen dies sinnvoll war, wurde daher eine entsprechende spezifische Visualisierung implementiert. Wird eine dieser Lernumgebungen verwendet, stellt die Benutzeroberfläche ein Fenster zu deren Darstellung bereit.

[5] Hierfür wurde die freie Klassenbibliothek ZedGraph [ZedGraph] verwendet .

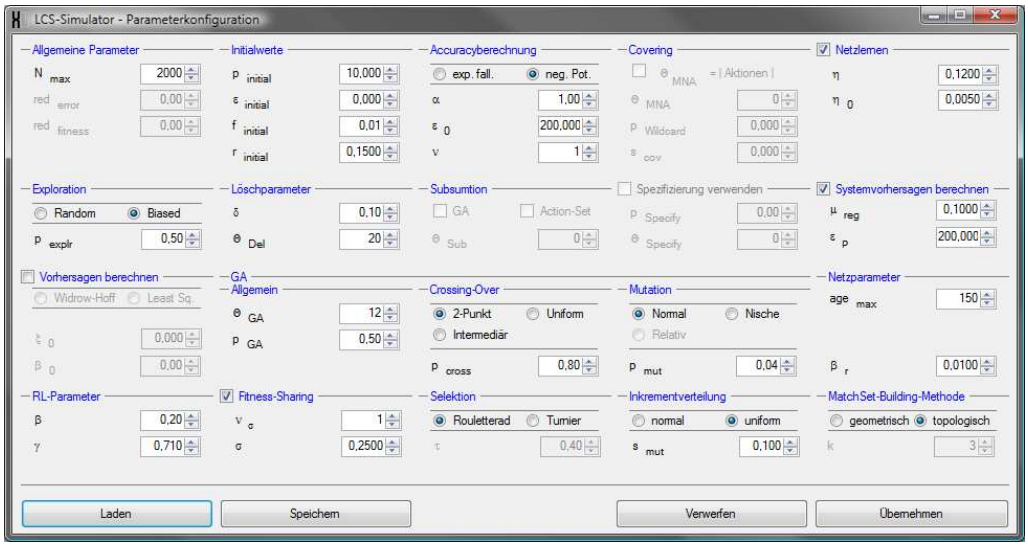


Abbildung 5.3: Ansicht des Fensters für die Darstellung und Bearbeitung der System-Parameter.

Visuelle Darstellung der Klassifizierer im Zustandsraum

Um einen Überblick über die Verteilung der Klassifizierer im Eingaberaum zu ermöglichen, wurde eine (je nach verwendeter Lernumgebung) zwei- oder dreidimensionale Darstellung mittels OpenGL umgesetzt (siehe Abbildung 5.4). Im Falle einer zwei- oder

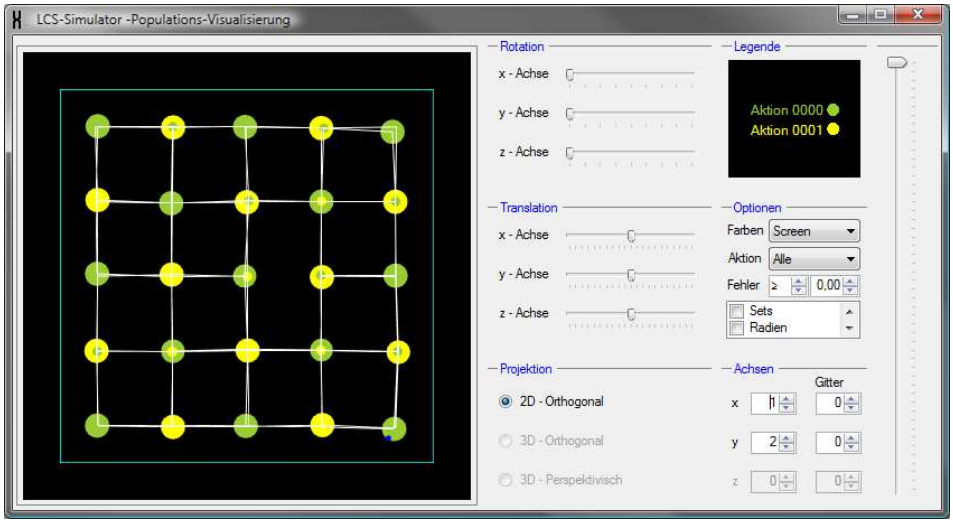


Abbildung 5.4: Fenster für die OpenGL-Visualisierung der Klassifizierer im Eingaberaum.

dreidimensionalen Repräsentation der Zustände der Lernumgebung können Klassifizierer somit direkt dargestellt werden; im Falle einer höheren Dimensionalität der Lernumgebungszustände handelt es sich bei der Darstellung um eine Projektion der Klassifizierer. Der Benutzer kann die zu projizierenden Dimensionen interaktiv auswählen. Im Normalfall zeigt die Darstellung den gesamten Eingaberaum; es ist aber auch möglich, einzelne Bereiche in der Visualisierung hervorzuheben und „aus der Nähe“ zu betrachten.

Tabellarische Darstellung der aktuellen Klassifiziererpopulationen

Die in der Population, dem Match-Set und dem aktuellen Action-Set des untersuchten Lernenden Klassifizierenden Systems enthaltenen Klassifizierer können mit ihrer Bedingung und Aktion sowie ihren Attributen in tabellarischer Form dargestellt werden:

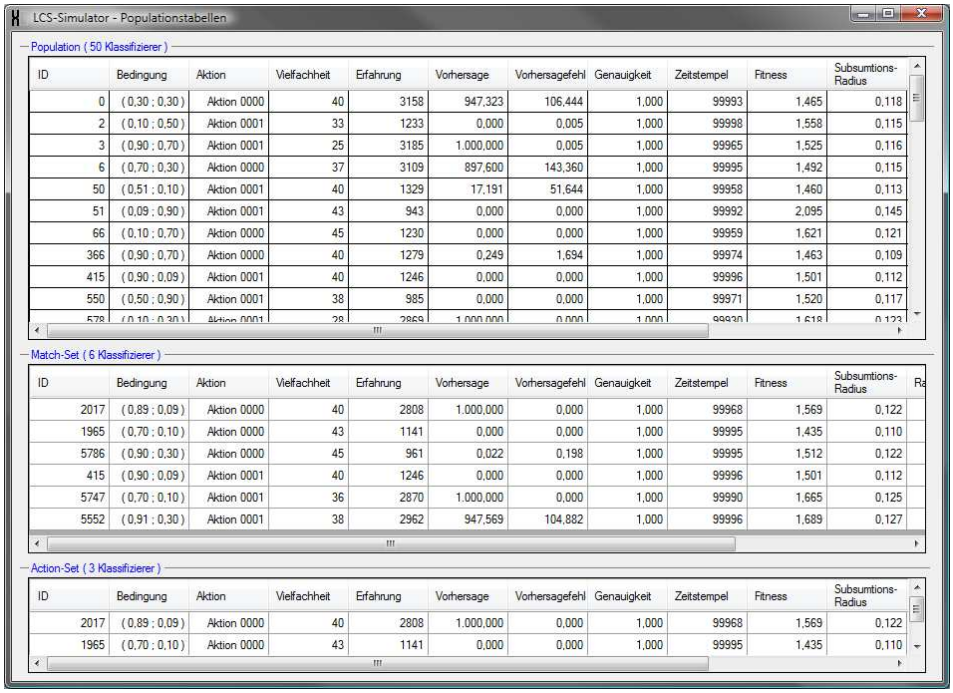


Abbildung 5.5: Fenster für die tabellarische Darstellung der aktuellen Klassifiziererpopulationen.

Während die visuellen Darstellungen der Klassifizierer und der Lernumgebung sich während eines Laufs des Lernenden Klassifizierenden Systems selbstständig aktualisieren, muss eine Aktualisierung der tabellarischen Darstellung vom Benutzer angefordert werden. Die tabellarische Darstellung der Klassifiziererpopulation kann als CSV-Datei exportiert werden[6].

5.4.2 Konfiguration von Serienläufen

Die Anforderung S.4 aus 5.1 wurde durch ein eigenständigen Programm zur Konfiguration von Serienläufen umgesetzt, dessen Benutzeroberfläche die Abbildung 5.6 zeigt. Serienexperimente werden konfiguriert, indem die gewünschte Anzahl enthaltener Einzelläufe festgelegt wird. Für diese müssen dann die zu verwendende Lernumgebung, das zu untersuchende Lernverfahren und gegebenenfalls der Typ der zu benutzenden Klassifizierer spezifiziert werden. Optional können für jeden Lauf auch Parametersätze und Startpopulationen angegeben werden. Geschieht dies nicht, werden Standardparameter und eine leere Startpopulation verwendet. Um die Reproduzierbarkeit jedes Laufes zu garantieren, werden auch Startwerte für den Pseudozufallszahlengenerator des Simulators festgelegt, diese werden vom Programm vorgeschlagen, können aber auch editiert werden. Die Konfiguration der Einzelläufe erfolgt in einer Reihe von Eingabedialogen.

[6] Die Benutzeraktion erfolgt in diesem Fenster über ein Kontextmenü.

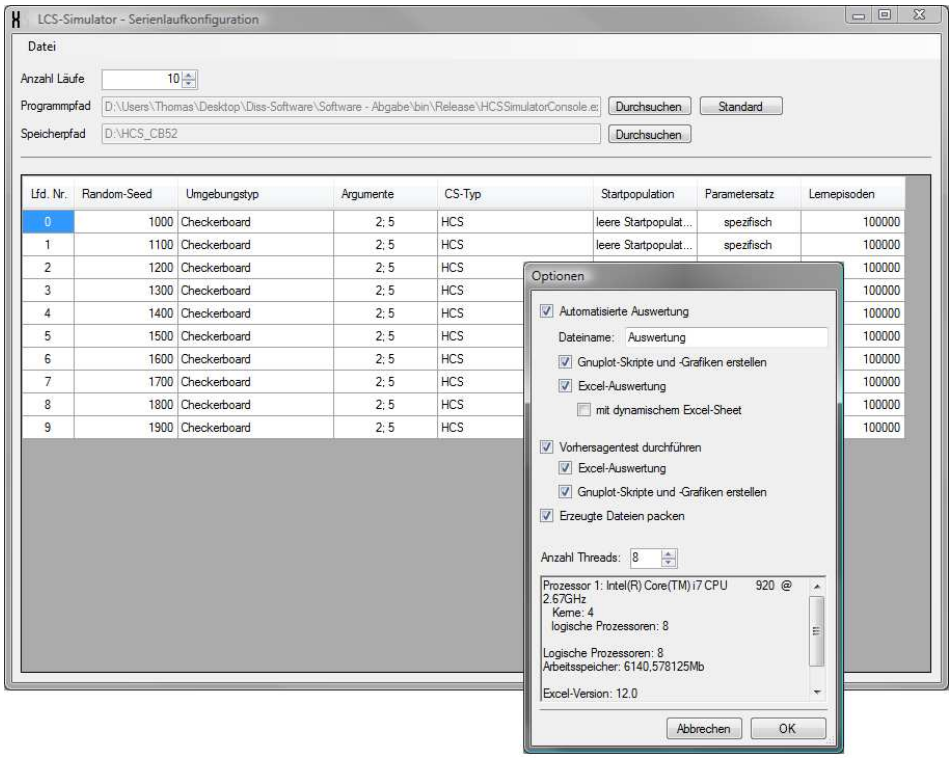


Abbildung 5.6: Ansicht des Fensters für die Konfiguration von Serienexperimenten und des Dialogs für die Auswahl der automatisiert durchzuführenden Auswertungen.

Die Benutzereingaben werden dabei überprüft, um fehlerhafte Konfigurationen weitgehend zu vermeiden. Die vorgenommenen Einstellungen werden in der in 5.6 zu erkennende Tabelle angezeigt. Zusätzlich zu den Konfigurationsdaten der Einzelläufe müssen ein Verzeichnis zum Speichern der Ergebnisse des Serienlaufs sowie der Pfad zum Konsolenprogramm, das diese durchführt, angegeben werden.
Das Programm erstellt schließlich am angegebenen Speicherort eine Konfigurationsdatei für das Serienexperiment, die die nötigen Angaben für jeden zugehörigen Einzellauf enthält. Zuvor können in einem Dialog zusätzliche Optionen gewählt werden, möglich ist etwa die automatisierte Erzeugung von den Lernverlauf darstellenden Diagrammen mit Excel oder GnuPlot. Ebenfalls am angegebenen Speicherort wird eine Batch-Datei angelegt, die die nicht mit einer graphischen Oberfläche ausgestattete Variante der Simulationssoftware mit den den ausgewählten Optionen entsprechenden Kommandozeilen-Parametern startet um den Serienlauf durchzuführen

5.4.3 Durchführung von Serienläufen

Für die Durchführung der Serienläufe wurde, wie bereits erwähnt, eine Variante der Simulationssoftware ohne graphische Benutzeroberfläche erstellt, die die Anforderungen S.1 bis S.3 erfüllt und deren Verwendung denkbar einfach ist: Um den Serienlauf durchzuführen, muss der Benutzer nur die im vorigen Abschnitt erwähnte Batch-Datei starten; die Durchführung des Serienlaufs erfolgt dann vollständig automatisch. Für jeden Einzellauf werden am angegebenen Speicherort der Lernverlauf, die Endpopulation und

das komplette Lernszenario gespeichert[7], ebenso gegebenenfalls Ergebnisse weiterer optionaler Auswertungen. Der Ablauf des Serienexperiments wird in einer Log-Datei festgehalten; in dieser wird jeder erfolgreich durchgeführte Einzellauf mit der benötigten Zeit vermerkt, gegebenenfalls wird dort auch das Auftreten eines Fehlers während eines Testlaufs protokolliert.

5.5 Zusammenfassung

Das Kapitel gibt einen Überblick über die im Rahmen der vorliegenden Arbeit entwickelte Simulationsumgebung für das HCS und das zu Vergleichszwecken ebenfalls implementierte XCS, mit der die in den folgenden Kapiteln zu diskutierenden Experimente durchgeführt wurden.
Bei der Entwicklung dieser Software wurde weitgehend nach den Prinzipien des Software-Engineering verfahren. Das Resultat ist ein im Wesentlichen aus drei Paketen bestehendes Softwaresystem:
Das erste Paket enthält Hilfsfunktionalitäten (Exportroutinen, Datenstrukturen, mathematische Funktionen, ...), die von den beiden anderen Paketen verwendet werden. Das zweite Paket bildet der Simulatorkern, der die Kernelemente der Simulationssoftware, das heißt die implementierten Lernenden Klassifizierenden Systeme und Lernumgebungen, enthält. Innerhalb des Simulatorkernes wurde darauf geachtet, eine mögliche Erweiterung der Software um weitere Lernumgebungen und Varianten der enthaltenen Lernenden Klassifizierenden Systeme durch Anwendung geeigneter Entwurfsmuster und konsequente Verwendung objektorientierter Techniken zu erleichtern. Der Simulatorkern ist vollständig unabhängig vom dritten Paket, das grob gesagt die komplette Benutzeroberfläche umfasst und nur einseitig über eine kleine Schnittstelle mit dem Simulatorkern kommuniziert. Dies ermöglicht es, die Benutzeroberfläche auszutauschen, ohne den Simulatorkern zu verändern.
Von dieser Möglichkeit wurde insofern schon Gebrauch gemacht, als das Oberflächenpaket der Software zwei voneinander unabhängige Benutzeroberflächen enthält, von denen jedoch nur die erste diese Bezeichnung verdient: Für die Durchführung von Einzelläufen war die Entwicklung einer übersichtlichen graphischen Benutzeroberfläche von zentraler Bedeutung, die die visuelle Darstellung des Lernverlaufs und der sich entwickelnden Klassifiziererpopulation während des Laufes ermöglicht und zudem die interaktive Veränderung von Systemparametern ermöglicht. Da die Simulationssoftware aber auch zur Durchführung von Serienexperimenten verwendet werden sollte, bei denen eine – Rechnerressourcen blockierende – graphische Oberfläche nicht benötigt wird, wurde als zweite „Benutzeroberfläche“ ein Konsolenprogramm implementiert, das den Simulatorkern während der Durchführung von Serienexperimenten kontrolliert. Die Interaktion des Benutzers mit diesem beschränkt sich jedoch, da alle Serienexperimente skriptgesteuert ablaufen, darauf, es unter Angabe des zu verwendenden Konfigurationsskriptes zu starten.
Für die Erstellung dieser Skripte wurde neben der eigentlichen Simulationssoftware ein komfortables Konfigurationsprogramm erstellt.

[7] Das Speichern des kompletten Szenarios gestattet es, einen Lauf fortzusetzen.

Kapitel 6

Klassifikation mit dem HCS

Zur Evaluation des HCS wurde dieses in verschiedenen Lernumgebungen aus den drei typischen Einsatzgebieten Lernender Klassifizierender Systeme – Klassifikation, Funktionsapproximation und Aktionenlernen – getestet. Die Ergebnisse der durchgeführten Experimente werden in diesem und den beiden nächsten Kapiteln präsentiert. Hier geht es zunächst um den Einsatz des HCS zur Klassifikation; Kapitel 7 ist der Funktionsapproximation gewidmet, Kapitel 8 schließlich betrachtet das Lernen von Aktionen.
Bei Klassifikationsproblemen besteht die Aufgabe eines Lernenden Klassifizierenden Systems darin, die ihm von der Lernumgebung präsentierten Eingaben – Elemente des Zustandsraums $\mathcal{S}$ – in Klassen einzuteilen, die durch die Elemente der Aktionsmenge $\mathcal{A}$ beschrieben werden – deren Anzahl also durch die Mächtigkeit von $\mathcal{A}$ fest vorgegeben ist. Dementsprechend erfolgt die Klassifikation einer Eingabe in Form der Ausführung der vom Lernenden Klassifizierenden System ausgewählten Aktion, weswegen in diesem Kapitel die Begriffe *Aktion* und *Klassifikation* synonym verwendet werden. Wie zutreffend die vorgenommene Klassifikation ist, wird dem Lernenden Klassifizierenden System durch die unmittelbar folgende Belohnung mitgeteilt.
In Abschnitt 6.1 wird zunächst kurz das Vorgehen bei den im Rahmen dieser Arbeit durchgeführten Experimenten erläutert. Dieses ist unabhängig davon, welchem der drei oben genannten Einsatzgebiete Lernender Klassifizierendes Systeme die verwendete Lernumgebung entstammt. Demgegenüber hängen die Kriterien, anhand derer Lernerfolg und Leistung eines Lernenden Klassifizierenden Systems beurteilt werden, stark vom jeweiligen Einsatzszenario ab. Die in diesem Kapitel, also im Falle der Klassifikation, verwendeten Kriterien werden im Abschnitt 6.2 erläutert. Im Anschluss daran werden in den Abschnittten 6.3 und 6.4 die zur Klassifikation mit dem HCS durchgeführten Experimente besprochen. Als Lernumgebungen wurden je zwei Vertreter der Multiplexer- und der Checkerboard-Problemfamilie verwendet, die häufig als Testprobleme für (reellwertig kodierte) Lernende Klassifizierende Systeme eingesetzt werden[1]. Eine kurze Zusammenfassung der gewonnenen Erkenntnisse in Abschnitt 6.5 schließt das Kapitel.

[1]Multiplexer-Probleme wurden in einer ganzen Reihe von Publikationen verwendet, die (später eingeführten) Checkerboard-Probleme (bisher) weniger oft. Dies liegt aber auch daran, dass viele neuere Veröffentlichungen zum XCS dessen Anwendung auf und Anpassung an speziellere Probleme zum Thema haben, also kein Standard-Testproblem benötigen.

6.1 Experimentelles Vorgehen und Parameterwahl

Das Vorgehen bei allen im Rahmen dieser Arbeit durchgeführten Experimente orientierte sich an dem in [Wilson 1994] und [Wilson 1995] vorgeschlagenen, welches in vielen weiteren Veröffentlichungen übernommen wurde. Es berücksichtigt, dass Lernende Klassifizierende Systeme einer ganzen Reihe von Zufallseinflüssen unterliegen:

- Die Auswahl der Probleminstanzen, mit denen das System konfrontiert wird, erfolgt zufällig.
- Die ausgeführten Aktionen werden teilweise zufällig gewählt (Exploration).
- Die Bedingungen der vom Covering-Operator erzeugten Klassifizierer werden zum Teil (Wildcards, Intervallgrenzen) vom Zufall bestimmt.
- Die Discovery-Komponente verwendet in Form des Genetischen Algorithmus ein randomisiertes Suchverfahren.

Als Konsequenz dieser Zufallseinflüsse unterliegt auch die zeitliche Entwicklung von zur Beurteilung der Leistung eines Lernenden Klassifizierenden System herangezogenen Größen Schwankungen und erfolgt auch bei gleichen Startbedingungen nicht stets in exakt gleicher Weise. Um diese Schwankungen auszugleichen, wurden in allen Experimenten in diesem und den beiden folgenden Kapiteln jeweils 10 Läufe durchgeführt, über die die jeweils zur Beurteilung der Leistung des Lernenden Klassifizierenden System herangezogenen Größen gemittelt wurden. Die Läufe eines Experimentes unterschieden sich dabei allein in den verwendeten Startwerten der Pseudozufallszahlgeneratoren.
Der erste der oben genannten Punkte birgt hinsichtlich des Vergleichs Lernender Klassifizierender Systeme ein weiteres Problem: Der Lernerfolg eines Lernenden Klassifizierenden Systems kann auch davon abhängen, welche Lernumgebungszustände ihm in welcher Reihenfolge präsentiert werden. Ein fairer Vergleich zweier Systeme setzt also voraus, dass diese mit den gleichen Zuständen in der gleichen Reihenfolge konfrontiert werden[2]. Dies wurde sichergestellt, indem der für die Zustands-Generierung verwendete Pseudozufallszahlgenerator in allen Experimenten mit den gleichen vorgegebenen Startwerten – für jeden der 10 Einzelläufe ein anderer – initialisiert wurde.
Aus der Notwendigkeit der Exploration, also daraus, dass sowohl XCS wie auch HCS, um Fehleinschätzungen hinsichtlich der Qualität von Aktionen korrigieren zu können, mit einer gewissen Explorationswahrscheinlichkeit p_{explr} auch andere als die als optimal angesehenen Aktionen ausführen müssen, ergibt sich eine weitere Schwierigkeit bei der Beurteilung ihrer Leistung: sie zeigen sie nicht immer.
Diesem Problem wird begegnet, indem die Lernenden Klassifizierenden Systeme während der Experimente in zwei Modi – dem *Explorations-* und dem *Exploitationsmodus* – betrieben werden, wobei nach jeder Lernepisode der Modus gewechselt wird:

- Im Explorationsmodus verhalten sich die Systeme wie in Abschnitt 2.4 respektive Kapitel 4 beschrieben. Insbesondere führen sie also nicht immer die als am besten angesehene Aktion aus. Je nachdem, ob die Aktionswahl rein zufällig ($p_{\text{explr}} = 1$) oder unter Bevorzugung der als am besten eingeschätzten Aktion ($0 < p_{\text{explr}} < 1$) erfolgt, wird zwischen *reiner* und *bevorzugender Exploration* unterschieden. Bei allen in der vorliegenden Arbeit beschriebenen Experimenten wurde bevorzugende Exploration mit $p_{\text{explr}} = 0.5$ eingesetzt.

[2] Bei kontinuierlichen Lernproblemen und episodischen Lernproblemen mit einer Episodenlänge größer als 1 betrifft dies nur die Episodenstartzustände, da die innerhalb einer Episode auftretenden Zustände durch die Aktionen des Lernenden Klassifizierenden Systems mitbestimmt werden.

- Im Exploitationsmodus hingegen wird stets die als am besten angesehene Aktion ausgeführt. Die Reinforcement-Komponente sowie die Discovery-Komponente des jeweiligen Systems werden in diesem Modus nicht verwendet[3].

Bei der Beurteilung der Leistung der Lernenden Klassifizierenden Systeme werden nur die im Exploitationsmodus durchgeführten Lernepisoden berücksichtigt.

Sowohl das XCS wie auch das HCS verfügen über eine ganze Reihe von Parametern, deren Setzung erheblichen Einfluss auf die Leistung des jeweiligen Systems hat. Eine umfassende systematische Untersuchung der Auswirkungen verschiedener Parameterwerte und derer Kombinationen ist aufgrund der hohen Anzahl von Parametern jedoch praktisch nicht möglich. Ferner ist aufgrund der hohen Komplexität Lernender Klassifizierender Systeme auch eine mathematische Analyse des Einflusses der verschiedenen Parameter und damit deren Optimierung auf analytischem Weg nicht durchführbar. Geeignete Parametersätze für die durchgeführten Experimente wurden daher wie folgt bestimmt:

- Zur Bestimmung der für das HCS zu verwendenden Parameter wurden für jedes Experiment Tests mit unterschiedlichen Parametern durchgeführt, bei der abschließenden Durchführung des Experiments wurde dann der Parametersatz verwendet, mit dem in den vorangegangenen Tests das beste Ergebnis erzielt wurde.
- Für die mit dem XCS durchgeführten Vergleichsläufe wurden – mit wenigen Ausnahmen, auf die an entsprechender Stelle hingewiesen wird – die in der Literatur genannten Parameter verwendet[4].

Da hier keine systematischen Parameteruntersuchungen vorgenommen wurden und die verwendeten Parametersätze zwar die Leistung der Lernenden Klassifizierenden Systeme beeinflussen, für deren Beurteilung jedoch keine Rolle spielen, wird auf eine Angabe der vollständigen Parametersätze im Folgenden verzichtet, um die Lesbarkeit nicht zu beeinträchtigen. Die verwendeten Parametersätze sind jedoch in Anhang C sowie auf der zu dieser Arbeit gehörenden DVD-ROM (siehe Anhang D) zu finden.

6.2 Beurteilung der Klassifikationsleistung

Die Leistung und der Lernerfolg eines Lernenden Klassifizierenden Systems werden je nach betrachtetem Einsatzgebiet respektive Problemtyp an zumindest teilweise unterschiedlichen Kriterien gemessen. Beim Einsatz eines Lernenden Klassifizierenden Systems zur Klassifikation ist die zentrale Frage offenbar, ob es in der Lage ist, die ihm präsentierten Zustände richtig zu klassifizieren. Da die Aktionswahl – also die Klassifikation – basierend auf der Abschätzung der zu erwartenden Belohnungen erfolgt, ist die zweite wichtige Frage die nach der Fähigkeit, diese exakt vorherzusagen. Schließlich stellt sich noch die Frage nach der Fähigkeit zur korrekten Generalisierung, also zur Bildung maximal genereller Klassifizierer. Der Beurteilung dieser drei Fähigkeiten dienen die im Folgenden erläuterten Kriterien:

[3] Mit Ausnahme des Covering-Operators, dieser spielt in der Regel aber nur zu Beginn eines Experimentes eine Rolle.

[4] Genaue Quellenangaben folgen in den jeweiligen Abschnitten dieses und der nächsten Kapitel.

Performanz

Ein naheliegendes Maß für die Fähigkeit eines Lernenden Klassifizierenden Systems, die ihm präsentierten Lernumgebungszustände richtig zu klassifizieren, ist der als *Performanz* bezeichnete Quotient aus der Anzahl optimaler und der Zahl insgesamt vorgenommener Klassifikationen[5]:

$$\mathsf{p} = \frac{\#optimale\ Klassifikationen}{\#vorgenommene\ Klassifikationen} \tag{6.1}$$

Wie bereits in Abschnitt 6.1 angekündigt, werden bei der Berechnung der Performanz nur die im Exploitationsmodus vorgenommenen Klassifikationen berücksichtigt. Ferner werden von diesen auch nur die letzten n einbezogen[6]. Derart wird die Performanz im Wesentlichen durch Klassifikationen bestimmt, die basierend auf der aktuellen Klassifiziererpopulation vorgenommen wurden, beschreibt also die aktuelle Klassifikationsfähigkeit des Lernenden Klassifizierenden Systems.

Systemfehler

Im Hinblick auf die zweite der obigen Fragen ist der *Systemfehler* von Interesse. Darunter wird die über die jeweils letzten n im Exploitationsmodus erfolgten Klassifikationen gemittelte absolute Differenz zwischen der Systemvorhersage PA_k für die vorgenommene Klassifikation k und der tatsächlich erhaltenen Belohnung r verstanden. Meist wird diese auf die Differenz zwischen in der Lernumgebung minimal und maximal möglicher Belohnung normiert, sodass der Systemfehler e_T im Lernschritt T gegeben ist durch:

$$\mathsf{e}_T = \frac{1}{n} \sum_{t=T-n+1}^{T} \frac{|PA_{i,t} - r_t|}{r_{\max} - r_{\min}} \tag{6.2}$$

Der Systemfehler gibt also an, wie stark die von einem Lernenden Klassifizierenden System erwarteten Belohnungen im Mittel von den tatsächlich erhaltenen abweichen.

Populationsgröße

Ein weiteres Kriterium zur Beurteilung eines Lernenden Klassifizierenden Systems ist die Größe der zur Lösung eines Problems evolvierten Population, also die Anzahl der die Population bildenden Makroklassifizierer. Um deren zeitlichen Verlauf leichter in einem Diagramm mit Performanz und Systemfehler auftragen zu können, ist es sinnvoll, eine Normierung auf die maximal zulässige Größe $N_{\max}$ der Population vorzunehmen:

$$\mathsf{s} = \frac{\#Makroklassifizierer}{N_{\max}} \tag{6.3}$$

Ist im Folgenden von der *Populationsgröße* die Rede, so wird in der Regel dieser Quotient gemeint sein. Sollte die tatsächliche Größe der Population gemeint sein, wird sich dies aus dem Kontext ergeben.
Im Idealfall entspricht die Größe der Population der Anzahl maximal genereller Klassifizierer, die eine korrekte Klassifizierung aller Lernumgebungszustände gestattet. Diese Anzahl hängt nicht nur von der verwendeten Lernumgebung ab, sondern auch von dem

[5] Um die Performanz berechnen zu können, muss natürlich bekannt sein, welcher Klasse jeder Zustand zuzuordnen ist. Bei den hier betrachteten Testproblemen ist diese Voraussetzung natürlich gegeben, im Allgemeinen ist sie jedoch nicht erfüllt.

[6] In den im Rahmen dieser Arbeit durchgeführten Experimenten wurde stets $n = 50$ gewählt.

eingesetzten Lernenden Klassifizierenden System: Einerseits ist sie davon abhängig, in welchem Maße in der jeweiligen Lernumgebung prinzipiell generalisiert werden kann, andererseits aber auch davon, wie gut die möglichen Generalisierungen durch die Syntax der Klassifiziererbedingungen abgebildet werden können. Ist diese Anzahl bekannt – was bei den meisten „echten" Lernproblemen natürlich nicht der Fall ist –, gestattet die Größe der evolvierten Population eine grobe Beurteilung der Fähigkeit eines Lernenden Klassifizierenden Systems zur korrekten Generalisierung[7]:
Liegt die Größe der Population unter dem Idealwert, so weist dies auf eine Tendenz zur Übergeneralisierung hin. Diese geht oft mit einem hohen Systemfehler einher, jedoch keineswegs immer. Betrifft die Übergeneralisierung zum Beispiel nur selten auftretende Zustände oder nur Zustände mit ähnlichen Belohnungsniveaus, bleibt die Auswirkung auf den Systemfehler gering, sodass ein niedriger Systemfehler auch bei Übergeneralisierung möglich ist. Liegt die Größe der Population über dem Idealwert, deutet dies darauf hin, dass das System weniger Generalisierungen als möglich vornimmt und viele „Spezialisten" hervorbringt, die nur in wenigen Lernumgebungszuständen anwendbar sind. Der Systemfehler liefert in diesen Fällen einen Hinweis darauf, ob diese Spezialisten zumindest genau sind oder nicht. Eine Größe der Population, die ungefähr dem Idealwert entspricht, weist in Kombination mit einem niedrigen Systemfehler auf eine gut entwickelte Fähigkeit zur korrekten Generalisierung hin. In Kombination mit einem hohen Systemfehler hingegen ist sie ein Anzeichen dafür, dass entweder nicht richtig generalisiert wird oder die Klassifizierer des Systems ungenau sind (oder beides).

		Größe der Population		
		$<$ Idealwert	$\approx$ Idealwert	$>$ Idealwert
Systemfehler	niedrig	Übergeneralisierung	korrekte Generalisierung	Überspezialisierung
	hoch	Übergeneralisierung	falsche Generalisierung, ungenaue Klassifizierer	Überspezialisierung, ungenaue Klassifizierer

Tabelle 6.1: Abschätzung der Fähigkeit eines Lernenden Klassifizierenden Systems zur korrekten Generalisierung.

Die gerade geschilderten und in Tabelle 6.1 zusammengefassten Zusammenhänge gestatten eine grobe Einschätzung der Generalisierungsfähigkeit eines Lernenden Klassifizierenden Systems. Dabei ist jedoch zu berücksichtigen, dass die genannten Kombinationen von Populationsgröße und Systemfehler auch auf andere Art zustande kommen können. So ist es beispielsweise denkbar, dass eine ideale Größe der Population daraus resultiert, dass ein Lernendes Klassifizierendes System teilweise übergenerelle Klassifizierer und teilweise zu spezielle Klassifizierer bildet.
Für eine genaue Beurteilung der Fähigkeit eines Lernenden Klassifizierenden Systems zur korrekten Generalisierung ist daher eine direkte Inspektion der evolvierten Klassifizierer unumgänglich.

[7] Hier wird angenommen, dass $N_{\max}$ größer als die benötige Anzahl maximal genereller Klassifizierer ist.

6.3 Der reellwertige 6-Multiplexer

Bei der Untersuchung binär kodierter Lernender Klassifizierender Systeme wurden oft Probleme aus der Familie der *Booleschen Multiplexer* verwendet, so auch im Falle des ZCS [Wilson 1994] und beim XCS [Wilson 1995]. Um diese bewährten Testprobleme auch beim reellwertig kodierten XCS einsetzen zu können, führte Wilson gleichzeitig mit diesem eine reellwertige Kodierung für Multiplexer-Probleme ein [Wilson 2000a].

6.3.1 Problembeschreibung

Unter einem Multiplexer wird in der Digitaltechnik und technischen Informatik ein Schaltnetz verstanden, das es ermöglicht, eines aus einer Reihe von Eingabesignalen auszuwählen. Zu diesem Zweck verfügt ein Multiplexer über 2^k Dateneingänge, einen Datenausgang[8] und k Steuereingänge. Die an den letztgenannten anliegenden Signale indizieren, als Binärzahl interpretiert, einen der Dateneingänge; das an diesem anliegende Signal wird auf den Datenausgang durchgeschaltet. In der technischen Informatik ist es üblich, einen Multiplexer durch Angabe der Anzahl der Datenein- und -ausgänge näher zu bestimmen; beispielsweise wird der Multiplexer mit $k = 2$ als 4:1-Multiplexer bezeichnet. In der Literatur zu Lernenden Klassifizierenden Systemen hat sich jedoch eine andere Nomenklatur durchgesetzt, die auch hier verwendet werden soll. Ein Multiplexer wird dabei durch die Summe $l = k+2^k$ aus der Anzahl der Steuer- und der Dateneingänge spezifiziert, dem 4:1-Multiplexer entspricht in dieser Notation also der 6-Multiplexer. Dessen Aufbau aus UND- und ODER-Gattern ist in Abbildung 6.1(a) dargestellt.

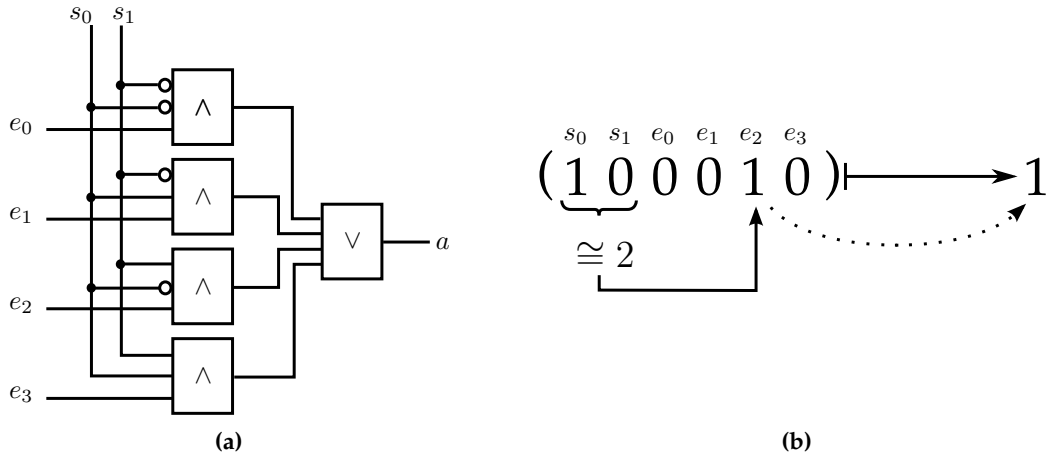


Abbildung 6.1: Der 6-Multiplexer: **(a)** Aufbau aus UND- und ODER-Gattern. **(b)** Ermittlung der korrekten Ausgabe. Die Addressbits `10` kodieren die Zahl 2, der Funktionswert ist somit der Wert des Datenbits e_2, also 1.

Das Verhalten des l-Multiplexers kann durch eine l-stellige Boolesche Funktion

$$F_l : \{0,1\}^l \rightarrow \{0,1\}$$

beschrieben werden, die im Folgenden ebenfalls als Multiplexer bezeichnet werden soll.

[8] Eine Verallgemeinerung stellen Multiplexer mit mehreren Datenausgängen dar. Diese sind hier jedoch nicht von Interesse, da die Klassifizierer Lernender Klassifizierender Systeme nur über eine Aktion - entsprechend einem Ausgang - verfügen.

Beispiel:

Der Boolesche 6-Multiplexer wird beschrieben durch die Funktion

$$F_6(s_0 s_1 e_0 e_1 e_2 e_3) = (e_0 \wedge \neg s_0 \wedge \neg s_1) \vee (e_1 \wedge s_0 \wedge \neg s_1) \vee (e_2 \wedge \neg s_0 \wedge s_1) \vee (e_3 \wedge s_0 \wedge s_1)$$

Da in jedem der geklammerten Ausdrücke die Steuerbits s_0 und s_1 respektive deren Negation in anderer Kombination auftreten, kann für jede Belegung der Funktion höchstens einer der geklammerten Ausdrücke wahr werden, der Wert des in diesem auftretenden Datenbits e_i wird zum Wert der Funktion. Die Funktion ist so gestaltet, dass der Index i dieses Datenbits gerade der von s_0 uns s_1 binär kodierten Zahl entspricht. So ist etwa $F_6(\texttt{100010}) = \texttt{1}$ (Abbildung 6.1(b)).

Als Testprobleme für Lernende Klassifizierende Systeme sind Multiplexer-Funktionen unter anderem interessant, da sie gut geeignet sind, deren Fähigkeit zur korrekten Generalisierung zu testen - die Fähigkeit, einen Satz von Klassifizierern zu evolvieren, der nicht nur eine korrekte Lösung eines Problems darstellt, sondern auch maximal generell, insbesondere also auch möglichst kompakt ist. Diese Eignung ergibt sich aus den weitreichenden Möglichkeiten zur Generalisierung, die Multiplexer bieten: Die korrekte Ausgabe ist für jede Eingabe nur abhängig von den Werten der Steuerbits und dem Wert des von diesen indizierten Datenbits; über alle übrigen Datenbits kann generalisiert werden. Von Vorteil ist dabei ferner die einfache Struktur der Multiplexer, durch die die Überprüfung der Korrektheit einer vorgenommenen Generalisierung sehr leicht wird.

Beispiel:

Im Falle des 6-Multiplexers liefert der Klassifizierer `10##1#:1` nicht nur für die Eingabe `100010` aus dem obigen Beispiel die richtige Ausgabe, sondern auch für die übrigen sieben Eingaben mit $s_0 = \texttt{1}$, $s_1 = \texttt{0}$, $e_2 = \texttt{1}$. Insgesamt sind beim 6-Multiplexer für eine vollständig korrekte Klassifikation aller 64 möglichen Eingaben nur 8 solcher maximal generellen Klassifizierer pro Aktion nötig; beim nächstgrößeren Multiplexer, dem 11-Multiplexer, genügen 16 Klassifizierer pro Aktion für die korrekte Klassifikation aller 2048 Eingaben.

Bei der Einführung des reellwertigen XCS in [Wilson 2000a] wurden auch Multiplexer mit reellwertigen Eingaben definiert. Der reelle Multiplexer der Länge l wird beschrieben durch die Funktion

$$RF_l = F_l \circ b : [0,1)^l \to \{0,1\}$$

in der die Umkodierungsfunktion b jedem Eingabewert x_i einen Binärwert b_i zuordnet:

$$b : [0,1)^l \to \{0,1\}^l$$
$$(x_0, x_1, \ldots, x_{l-1}) \mapsto b_0 b_1 \ldots b_{l-1}$$

$$b_i = \begin{cases} 1 & \text{für } x_i < \Theta_i \\ 0 & \text{sonst} \end{cases}, i = 0, 1, \ldots, l-1$$

Die Schwellenwerte $\Theta_i, i = 0, 1, \ldots, l-1$ werden üblicherweise auf den Wert 0.5 gesetzt, da so der binär kodierte Multiplexer am genauesten imitiert wird. Diese Setzung wird im Folgenden auch hier stets verwendet.
Trotz der sehr direkten „Übersetzung“ stellt ein reellwertig kodierter Multiplexer ein weit schwierigeres Lernproblem dar als sein binäres Gegenstück. Zur Validierung der Neuimplementierung des XCS für die in Kapitel 5 beschriebene Software wurden die in [Wilson 2000a] für den 6-Multiplexer dargestellten Ergebnisse reproduziert. In Abbildung 6.2 ist die Performanz des XCS für die binäre sowie die reellwertige Kodierung gegen die Anzahl präsentierter Probleminstanzen aufgetragen. Zu erkennen ist, dass im reellen Fall ein Maximum von etwa 0.98 erst nach etwa 15000 Lernschritten erreicht wird, während

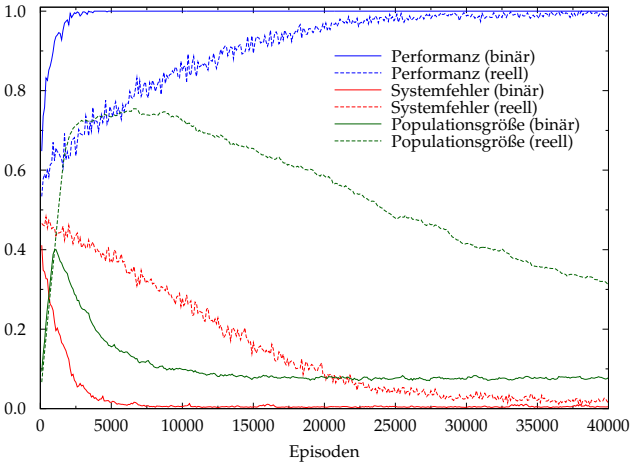


Abbildung 6.2: Zeitliche Entwicklung von Performanz, Systemfehler und Populationsgröße bei Anwendung des binär kodierten XCS und des reellwertig kodierten XCS mit Centre-Spread-Bedingungen auf den (entsprechend kodierten) 6-Multiplexer.

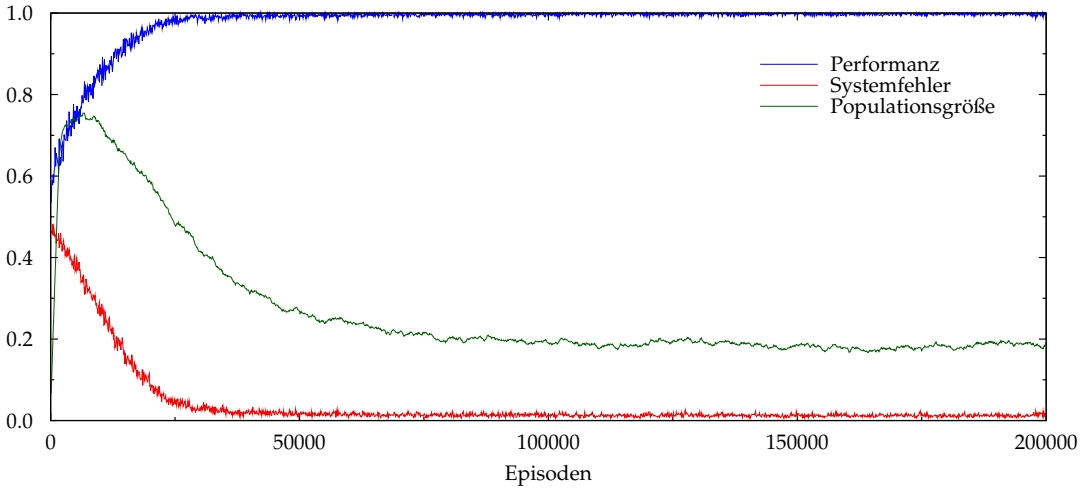
im binären Fall alle Eingaben bereits nach nur 1500 Lernschritten korrekt klassifiziert werden. Wilson nennt hierfür zwei mögliche Gründe: Zum einen haben die intervallbasierten Klassifiziererbedingungen zwölf Allele, die binären hingegen nur sechs Allele[9]; zum anderen ist der Zustandsraum und folglich auch der Suchraum des Genetischen Algorithmus im Falle der reellwertigen Kodierung deutlich größer – gibt es im binären Fall nur 64 verschiedene Eingaben, sind es im reellen Fall im Prinzip unendlich viele.

6.3.2 Der Standard-6-Multiplexer

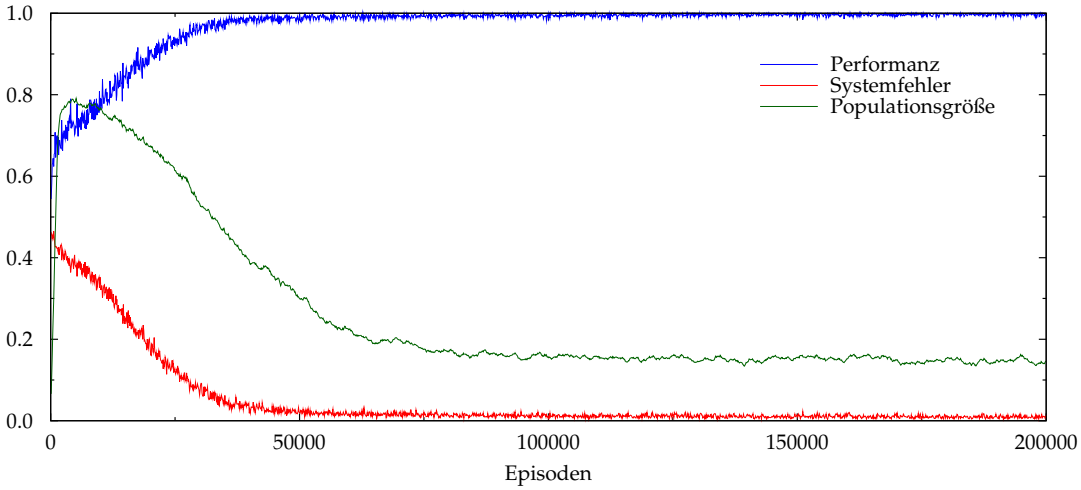
Im Rahmen dieser Arbeit wurde das HCS mit zwei Varianten des Multiplexer-Problems getestet. Die erste ist der Standard-6-Multiplexer, bei dem die Belohnung, die das Lernende Klassifizierende System für jede korrekte Klassifikation erhält, den Wert 1000 hat, während jede falsche Klassifikation mit einer Belohnung der Höhe 0 „honoriert“ wird. Die Abbildung 6.3 zeigt die Ergebnisse der Anwendung zweier XCS-Varianten[10] und des HCS auf das 6-Multiplexer-Lernproblem. Während beide Varianten des XCS eine Performanz von nahezu 1 erreichen, klassifiziert das HCS nur etwa 96% der Eingaben korrekt. Auch der Systemfehler des HCS liegt deutlich über dem der beiden XCS-Varianten. Positiv zu vermerken ist hingegen, dass die Populationsgröße beim HCS nur geringfügig über dem Optimalwert liegt, während sie diesen bei beiden XCS-Varianten deutlich übersteigt: Die optimale Lösung besteht beim XCS aus 8 Klassifizierern pro Aktion (vergleiche das zweite Beispiel auf Seite 120). Bei maximal $N_{\max} = 800$ erlaubten Klassifizierern entsprechen die erreichten Endwerte der Populationsgröße von etwa 0.19 beim XCS mit Centre-Spread-Bedingungen und etwa 0.15 beim XCS mit Unordered-Bound-Bedingungen etwa 150 respektive 120 Makroklassifizierern. Demgegenüber entspricht die vom HCS erreichte Populationsgröße von ungefähr 0.063 bei einer maximalen Populationsgröße von $N_{\max} = 2000$ etwa 130 Klassifizierern, was nur geringfügig über dem für das HCS gültigen Minimum von 128 Klassifizierern (siehe unten) liegt.

[9] Der binäre 11-Multiplexer (11 Allele) benötigt etwa 10000 Lernschritte um eine Performanz von 1.0 zu erreichen [Wilson 1998].

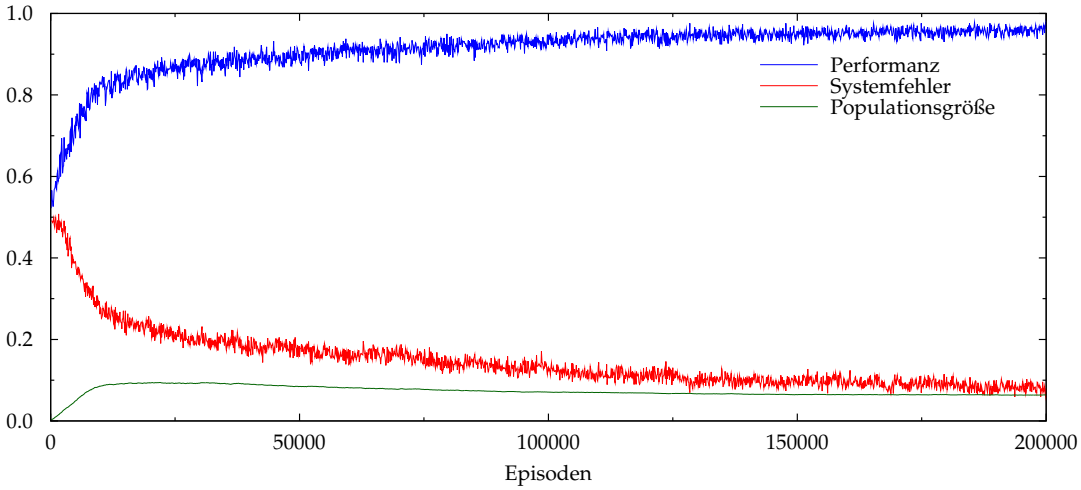
[10] Für die Testläufe des XCS wurden die in [Stone u. Bull 2003] angegebenen Parameter verwendet.



(a)



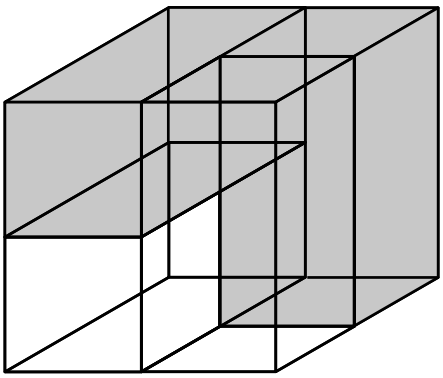
(b)



(c)

Abbildung 6.3: Zeitliche Entwicklung von Performanz, Systemfehler und Populationsgröße bei Anwendung des **(a)** XCS mit Centre-Spread-Bedingungen, **(b)** XCS mit Unordered-Bound-Bedingungen und **(c)** HCS auf den 6-Multiplexer.

Die schlechtere Performanz des HCS und dessen höherer Systemfehler lassen sich dadurch erklären, dass der Zustandsraum des 6-Multiplexers eine für das HCS sehr herausfordernde Struktur besitzt, an die andererseits die intervallbasierten Bedingungen des reellwertig kodierten XCS perfekt angepasst sind. Dies soll hier – der besseren Darstellbarkeit wegen – anhand des reellwertig kodierten 3-Multiplexers erläutert werden, dessen Zustandsraum in Abbildung 6.4(a) dargestellt ist, in den schattierten Bereichen ist Aktion 1 richtig, in den unschattierten Bereichen die Aktion 0:



(a)

Klassifizierer mit richtiger Aktion
$[0.0\|0.5) \times [0.0\|0.5) \times [0.0\|1.0) \Rightarrow 0$
$[0.0\|0.5) \times [0.5\|1.0) \times [0.0\|1.0) \Rightarrow 1$
$[0.5\|1.0) \times [0.0\|1.0) \times [0.0\|0.5) \Rightarrow 0$
$[0.5\|1.0) \times [0.0\|1.0) \times [0.5\|1.0) \Rightarrow 1$

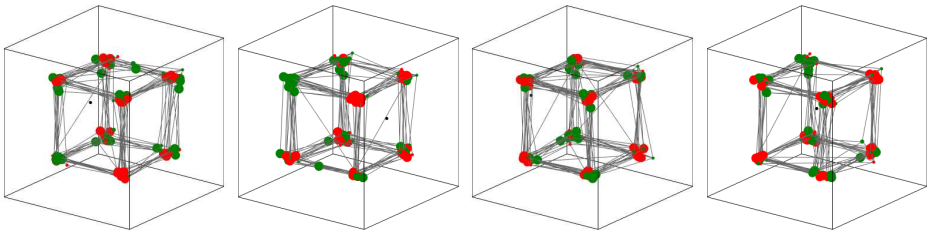
Klassifizierer mit falscher Aktion
$[0.0\|0.5) \times [0.0\|0.5) \times [0.0\|1.0) \Rightarrow 1$
$[0.0\|0.5) \times [0.5\|1.0) \times [0.0\|1.0) \Rightarrow 0$
$[0.5\|1.0) \times [0.0\|1.0) \times [0.0\|0.5) \Rightarrow 1$
$[0.5\|1.0) \times [0.0\|1.0) \times [0.5\|1.0) \Rightarrow 0$

(b)

Abbildung 6.4: Der reellwertig kodierte 3-Multiplexer: **(a)** Zustandsraum und **(b)** Perfekte XCSR-Lösung.

Um den 3-Multiplexer korrekt zu lösen, benötigt das reellwertig kodierte XCS nur die acht in Tabelle 6.4(b) angegebenen Klassifizierer.
Hingegen ist die in Abbildung 6.4(a) gezeigte Unterteilung des Zustandsraums offenbar nicht als Voronoizerlegung mit 4 Zentren darstellbar, es sind vielmehr mindestens acht Zentren nötig[11]. Dementsprechend benötigt das HCS pro Aktion 8 Klassifizierer um den 3-Multiplexer vollständig korrekt zu lösen. Die optimale Klassifiziererpopulation induziert dabei eine Voronoizerlegung des Zustandsraums, in der jeder Voronoizelle mindestens eine weitere Voronoizelle mit der gleichen optimalen Aktion benachbart ist. Dies erschwert dem Netzlernmechanismus des HCS die richtige Positionierung der Klassifizierer gegenüber dem Fall, in dem in benachbarten Voronoizellen stets unterschiedliche Aktionen optimal sind oder zumindest die gleiche optimale Aktion zu verschieden hohen Belohnungen führt: Da der Netzlernmechanismus die Klassifizierer von Zuständen wegschiebt, für die ihre Vorhersagen schlecht waren, können die Klassifizierer in diesem Fall ihre Position nur wenig verändern, sobald die von der Population induzierte Voronoizerlegung die Struktur des Zustandsraums einigermaßen gut abbildet. Im Falle des Multiplexer-Problems ist hingegen immer ein „Ausbrechen“ eines Klassifizierers in (mindestens) einer Richtung möglich. Im Falle des 6-Multiplexers verschärft sich dieses Problem noch: Die optimale Klassifiziererpopulation besteht aus $2^6 = 64$ Klassifizierern pro Aktion, die eine Voronoizerlegung des Zustandsraums induzieren, in der jeder Voronoizelle zwischen drei und fünf Voronoizellen mit gleicher optimaler Aktion benachbart sind. Dass es dem HCS dennoch gelingt, einen Performanzwert von 0.96 zu erreichen, zeigt jedoch, dass es trotz der geschilderten Schwierigkeiten in der Lage ist, seine Klassifizierer nahezu korrekt zu positionieren. Dies bestätigt auch die Abbildung 6.5, die Projektionen der Endpopulation eines Laufes des HCS auf je drei Dimensionen des Zustandsraumes – die Steuerdimensionen und je eine der Datendimensionen – zeigt:

[11] Durch Halbierung jeder Dimension kann der Zustandsraum in 8 Würfel zerlegt werden, deren Zentren eine geeignete Voronoizerlegung induzieren.

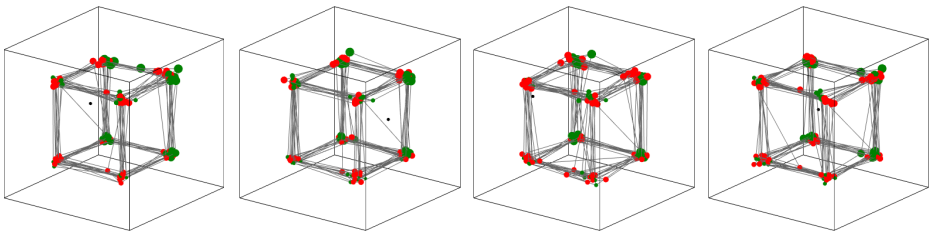


(a) **(b)** **(c)** **(d)**

Abbildung 6.5: Projektion einer vom HCS zur Lösung des reellwertig kodierten 6-Multiplexer evolvierten Population in die Dimensionen **(a)** s_0,s_1,e_0, **(b)** s_0,s_1,e_1, **(c)** s_0,s_1,e_2, **(d)** s_0,s_1,e_3.

6.3.3 Der gestaffelte Multiplexer

Die im Vergleich zum XCS schlechtere Performanz des HCS beim Lernen des 6-Multiplexers wurde im vorangegangenen Abschnitt darauf zurückgeführt, dass die von der optimalen Population zu induzierende Voronoizerlegung des Zustandsraums benachbarte Voronoizellen mit gleicher optimaler Aktion und gleichem Belohnungsniveau enthält, wodurch es dem Netzlernmechanismus des HCS erschwert wird, die Klassifizierer korrekt zu positionieren. Um diese Erklärung auch experimentell zu stützen, wurde eine reellwertig kodierte Variante des in [Wilson 1995] verwendeten gestaffelten 6-Multiplexers verwendet. Der gestaffelte Multiplexer erleichtert dem HCS die Positionierung, da in jedem Teil seines Zustandsraum, der durch einen maximal generellen Klassifizierer des XCS abgedeckt werden könnte[12], die Belohnungen eine andere Höhe haben. Damit ist jede Voronoizelle der durch die optimale HCS-Population induzierten Voronoizerlegung des Zustandsraums nur noch genau drei Voronoizellen benachbart, in denen die gleiche optimale Aktion zur gleich hohen Belohnungen führt[13]. Treffen die oben angestellten Überlegungen zu, sollte der gestaffelte Multiplexer für das HCS leichter zu lernen sein, als der Standard-Multiplexer. Dies ist tatsächlich der Fall, wie in Abbildung 6.7(c) zu erkennen ist: Das HCS erreicht sowohl eine höhere Performanz als auch einen niedrigeren Systemfehler als beim Standard-Multiplexer. Auch die Positionierung der Klassifizierer gelingt noch etwas besser, vergleiche Abbildung 6.6. Die Leistung des XCS erreicht das HCS aber auch hier nicht, vergleiche die Abbildungen 6.7(a) und 6.7(b).

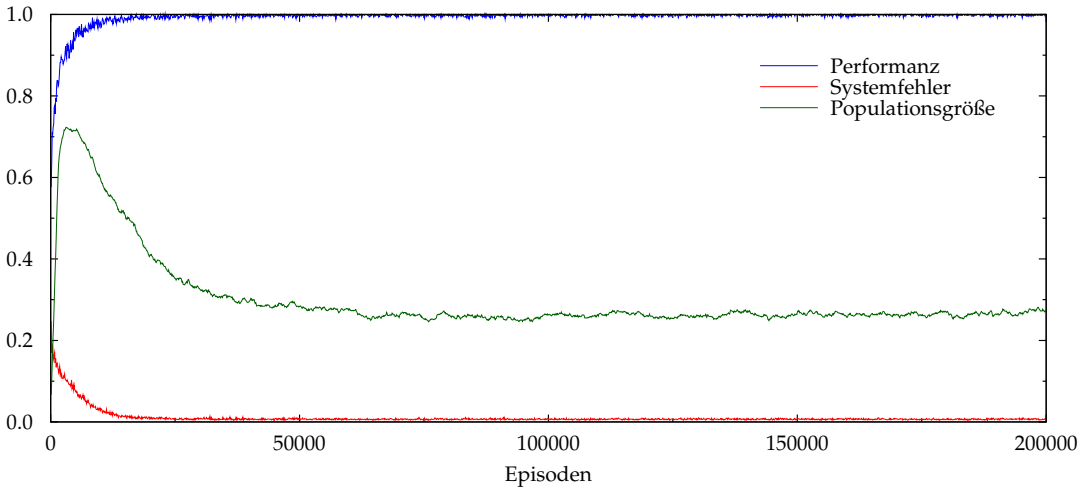


(a) **(b)** **(c)** **(d)**

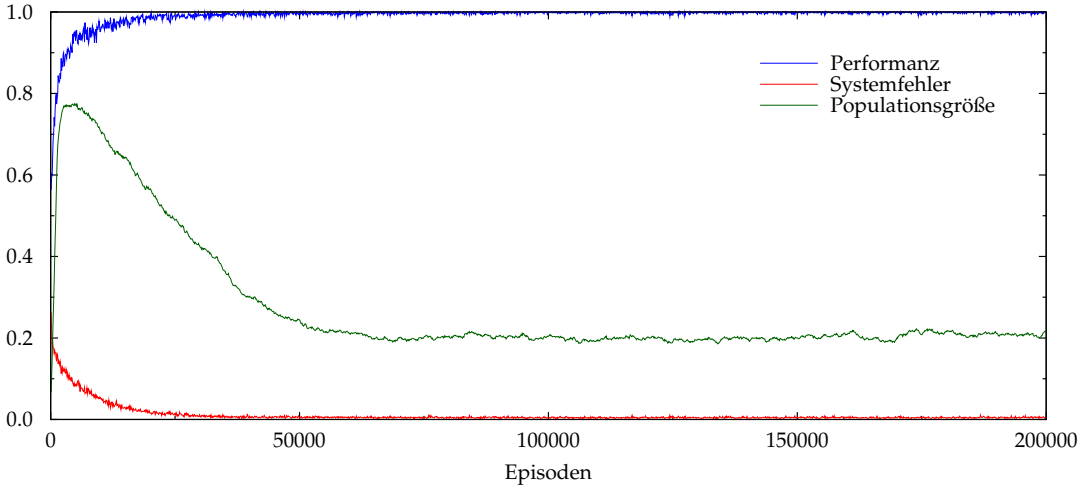
Abbildung 6.6: Projektion einer vom HCS zur Lösung des gestaffelten reellwertig kodierten 6-Multiplexer evolvierten Population in die Dimensionen**(a)** s_0,s_1,e_0, **(b)** s_0,s_1,e_1, **(c)** s_0,s_1,e_2, **(d)** s_0,s_1,e_3.

[12] Das sind die 6-dimensionalen Entsprechungen der Quader in Abbildung6.4(a).

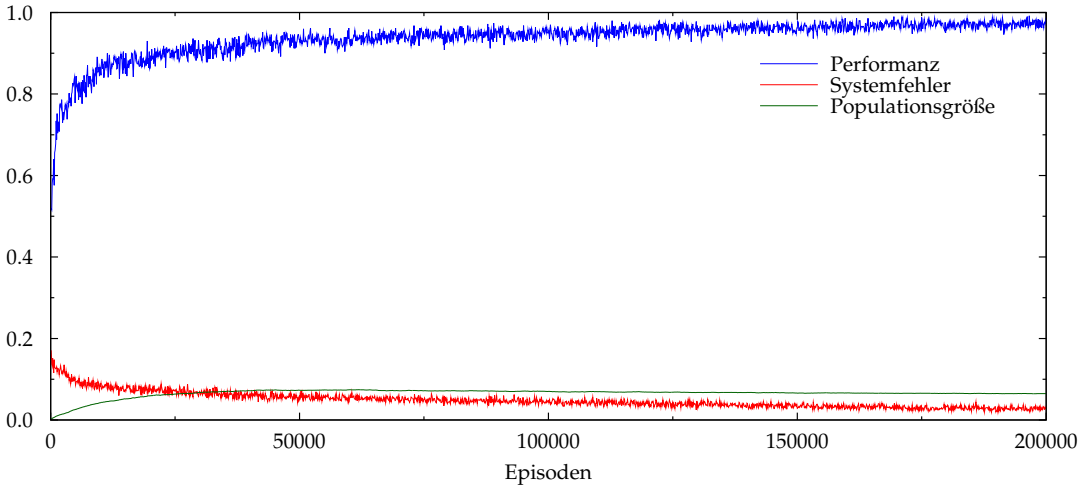
[13] Die Leistung des HCS könnte weiter verbessert werden, wenn diese Belohnungsstaffelung so weitergeführt würde, dass in benachbarten Voronoizellen mit gleicher optimaler Aktion stets unterschiedliche Belohnungen gewährt würden. Dies wäre der Problemstruktur des Multiplexers jedoch nicht angemessen.



(a)



(b)



(c)

Abbildung 6.7: Zeitliche Entwicklung von Performanz, Systemfehler und Populationsgröße bei Anwendung **(a)** des XCS mit Centre-Spread-Bedingungen, **(b)** des XCS mit Unordered-Bound-Bedingungen und **(c)** des HCS auf den gestaffelten 6-Multiplexer.

6.4 Checkerboard-Probleme

Bei den in [Stone u. Bull 2003] eingeführten Checkerboard-Problemen handelt es sich um eine Familie abstrakter Lernprobleme, die den n-dimensionalen Zustandsraum entlang jeder Dimension n_d-fach unterteilen. Die Checkerboard-Probleme kennen zwei Aktionen, von denen in jedem der so entstehenden Hyperkuben eine richtig und die andere falsch ist, wobei in benachbarten Hyperkuben stets unterschiedliche Aktionen richtig sind. (Werden diese Hyperkuben entsprechend der in ihnen richtigen Aktion schwarz oder weiß gefärbt, ergibt sich ein Schachbrettmuster, wodurch sich die Benennung der Problemfamilie erklärt.) Abbildung 6.8 zeigt die beiden im Rahmen dieser Arbeit untersuchten Vertreter der Checkerboard-Familie.

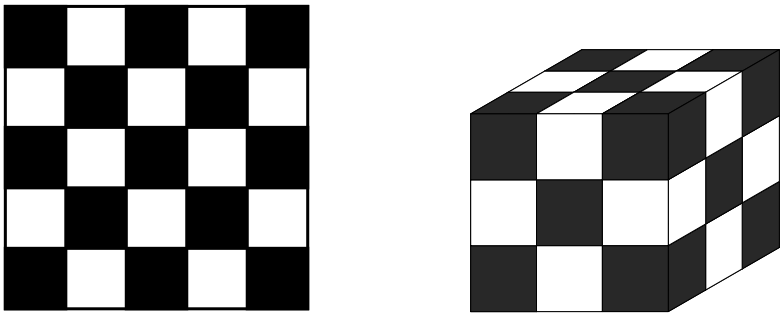
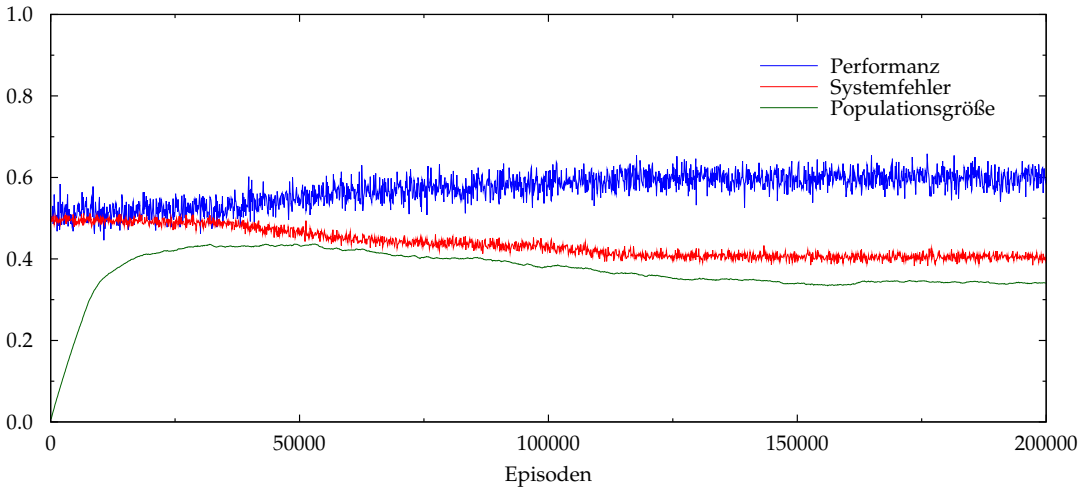


Abbildung 6.8: Zwei Checkerboard-Varianten: **(a)** Zweidimensionales Checkerboard-Problem mit $n_d = 5$. **(b)** Dreidimensionales Checkerboard-Problem mit $n_d = 3$. Die Größe der Klassifizierer spiegelt hier die Höhe ihrer Belohnungsvorhersagen wider.

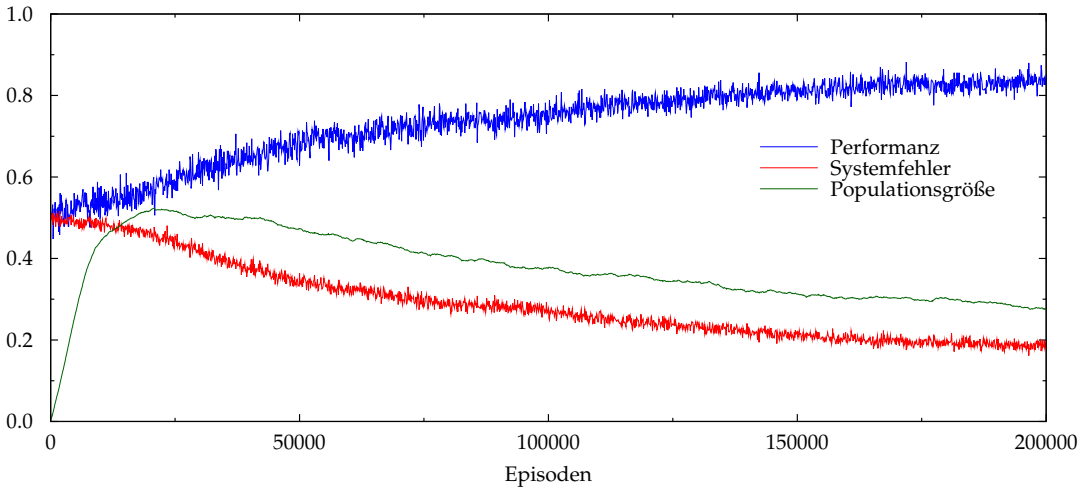
Die Schwierigkeit eines Checkerboard-Problems hängt zum einen von seiner Dimension n ab, zum anderen von der Anzahl n_d der Unterteilungen pro Dimension. Die Struktur der Checkerboard-Probleme impliziert, das diese theoretisch sowohl vom XCS mit intervallbasierten Bedingungen als auch vom HCS exakt gelöst werden können. Eine optimale Lösung besteht bei beiden Systemen aus jeweils $2 \cdot (n_d)^n$ maximal generellen Klassifizierern; pro Aktion und „Feld“ des Checkerboards wird ein Klassifizierer benötigt: Beim XCS korrespondieren deren Bedingungen direkt mit den jeweiligen Hyperkuben, beim HCS besteht die optimale Lösung aus Klassifizierern, deren Gewichtsvektoren den Schwerpunkten der Hyperkuben entsprechen.
Bei den im folgenden diskutierten Experimenten mit den Checkerboard-Varianten wurden für die Testläufe der eingesetzten XCS-Varianten die in [Stone u. Bull 2003] angegebenen Parameter verwendet.

6.4.1 Zweidimensionales Checkerboard mit fünf Unterteilungen

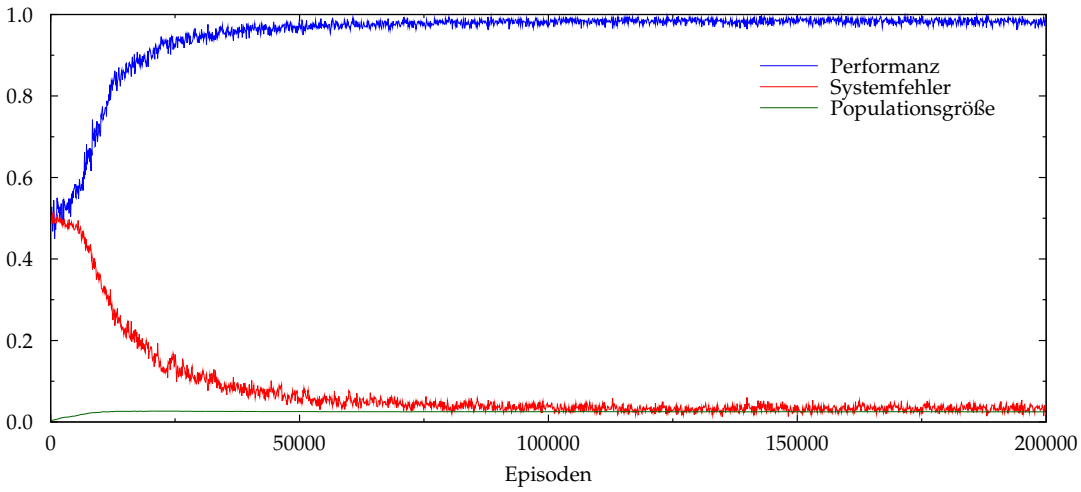
Die Abbildung 6.9 zeigt den Lernverlauf zweier XCS-Varianten und des HCS bei Anwendung auf das zweidimensionale Checkerboard mit fünf Unterteilungen. Dabei fällt als erstes die schlechte Leistung des XCS mit Centre-Spread-Intervallrepräsentation auf, das beim 6-Multiplexer noch ein sehr gutes Ergebnis erzielt hatte.
Zurückzuführen ist dies darauf, dass die Centre-Spread-Repräsentation bevorzugt Intervalle kodiert, die an mindestens einer Seite an die Grenzen des Zustandsraums stoßen [Stone u. Bull 2003]: Ist zum Beispiel der Zustandsraum durch $\mathcal{S} = [0, 1)^n$ gegeben, so kodiert die Centre-Spread-Repräsentation jedes Intervall als ein Tupel (c, s) mit $c, s \in [0, 1)$.



(a)



(b)



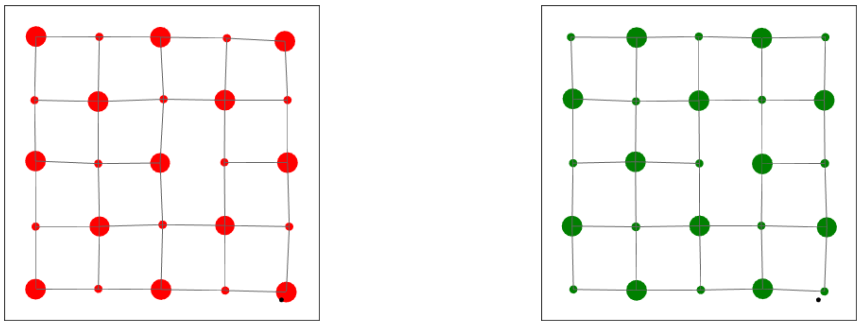
(c)

Abbildung 6.9: Zeitliche Entwicklung von Performanz, Systemfehler und Populationsgröße bei Anwendung des **(a)** XCS mit Centre-Spread-Bedingungen, **(b)** XCS mit Unordered-Bound-Bedingungen und **(c)** HCS auf das zweidimensionale Checkerboard mit fünf Unterteilungen.

Das von einem solchen Tupel kodierte Intervall ergibt sich vermöge der Abbildung

$$(c, s) \longmapsto [c - s, c + s) \cap [0, 1) \,.$$

Im Falle $s \geq c$ hat das kodierte Intervall also die untere Grenze 0, im Falle $s \geq 1 - c$ die obere Grenze 1. Es ist leicht zu sehen, dass nur ein Viertel aller Tupel (c, s) mit $c, s \in [0, 1)$ keine dieser Bedingungen erfüllt, während ein Viertel sogar beide Bedingungen erfüllt, mithin das Intervall $[0, 1)$ kodiert. Bei Anwendung auf den reellwertig kodierten Multiplexer ist dies ein Vorteil: Die Art und Weise, in der dieser konstruiert wurde, impliziert, dass für eine perfekte intervallbasierte Lösung nur die Intervalle $[0, 0.5)$, $[0.5, 1)$ und $[0, 1)$ benötigt[14] werden – vergleiche auch Abbildung 6.4(b) –, also gerade Intervalle der von der Centre-Spread-Repräsentation bevorzugten Form. Im Falle des Checkerboards hingegen sind die meisten in den Bedingungen der perfekten Klassifiziererpopulation auftretenden Intervalle nicht von dieser speziellen Form, sodass es dem XCS mit Centre-Spread-Repräsentation schwerer fällt, diese Klassifizierer zu evolvieren[15].
Die Unordered-Bound-Repräsentation hingegen weist keine Bevorzugung bestimmter Intervalle auf; jedes Intervall $[a, b)$, $a, b \in [0, 1)$ wird durch genau zwei Tupel kodiert: (a, b) und (b, a). So lernt denn auch das XCS mit dieser Intervallrepräsentation das Checkerboard im Mittel deutlich besser als das XCS mit Centre-Spread-Repräsentation. Dazu ist aber anzumerken, dass sich die einzelnen Testläufe, deren gemitteltes Resultat die Abbildung 6.9(b) darstellt, stark unterscheiden – während in einigen Läufen das Problem erfolgreich gelernt wird und nahezu alle Eingaben richtig klassifiziert werden (Performanz > 0.95), wird in anderen Läufen praktisch nichts gelernt (Performanz ≈ 0.5). Ferner liegt die Größe der Endpopulation selbst bei den besten Testläufen deutlich über der optimalen Größe, der niedrigste erreichte Wert liegt bei 0.13. Dies entspricht etwa 260 Makroklassifizierern, also mehr als dem fünffachen der optimalen Anzahl.



(a) **(b)**

Abbildung 6.10: Endopulation des HCS beim zweidimensionalen Checkerboard mit fünf Unterteilungen. **(a)** Klassifizierer mit Aktion 0. **(b)** Klassifizierer mit Aktion 1.

Eindeutig das beste Ergebnis beim zweidimensionalen Checkerboard mit fünf Unterteilungen erzielt das HCS - im Hinblick sowohl auf die Performanz wie auch den Systemfehler und die Populationsgröße. Mit Ausnahme eines Laufes, dessen Endpopulation aus

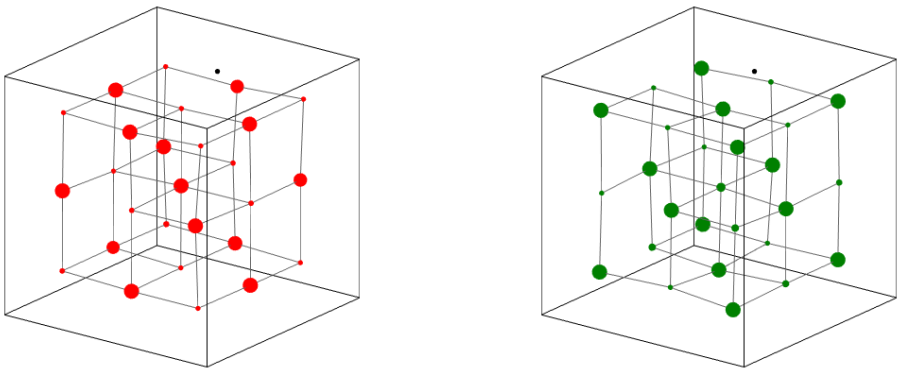
[14] Dies gilt im in Abschnitt 6.3.2 betrachteten Standardfall, im allgemeinen Fall (siehe Seite 120) sind es das Intervall $[0, 1)$ sowie Intervalle der Form $[0, \theta)$ respektive $[\theta, 1)$.

[15] In modifizierter Form ist aber auch das XCS mit Centre-Spread-Repräsentation zum Lernen des Checkerboards fähig, wie in [Stone u. Bull 2003] am Beispiel des dreidimensionalen Checkerboard mit drei Unterteilungen gezeigt wurde. Dazu wurde der Covering-Operator derart modifiziert, dass er nur Intervalle erzeugt, deren Spread s kleiner oder gleich 0.5 ist. Dies erschwert allerdings das Lernen des Multiplexers.

51 Makroklassifizierer besteht, enthalten die Endpopulationen aller Läufe des Experiments genau 50 Makroklassifizierer, was exakt der Anzahl der Makroklassifizierer in der optimalen Population entspricht. Auch gelingt es dem HCS, diese nahezu optimal anzuordnen: Abbildung 6.10 zeigt die Endpopulation eines der Testläufe, alle Klassifizierer sind nahe den Zentren der Schachbrettfelder positioniert.

6.4.2 Dreidimensionales Checkerboard mit drei Unterteilungen

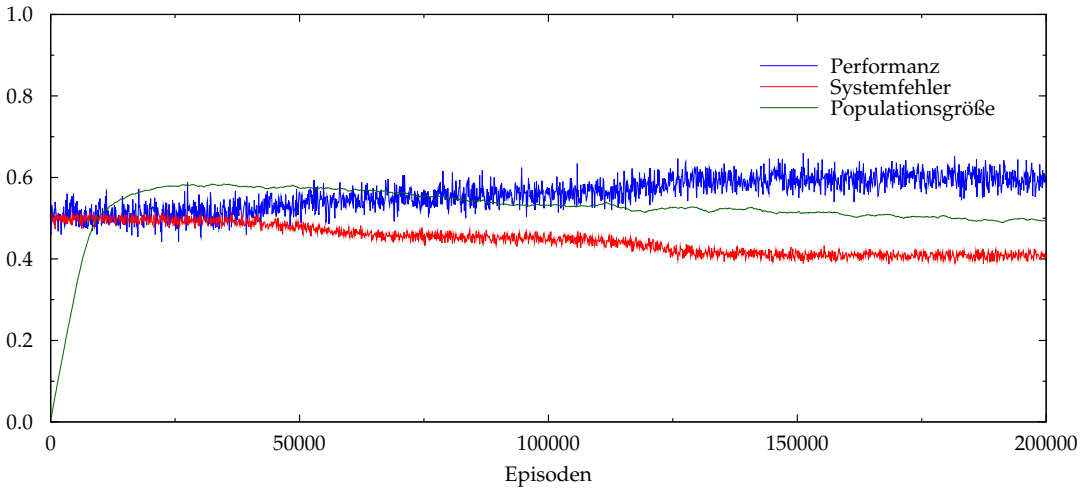
Im Falle des dreidimensionalen Checkerboard-Problems mit drei Unterteilungen entsprechen die Ergebnisse der durchgeführten Experimente im Wesentlichen den im letzten Abschnitt diskutierten Ergebnissen beim zweidimensionalen Checkerboard.
Das XCS mit Centre-Spread-Intervallrepräsentation erzielt das schlechteste Ergebnis, wobei das Resultat des hier durchgeführten Experimentes sogar besser ist als das in [Stone u. Bull 2003] publizierte[16]. Dort bleibt das XCS mit Centre-Spread-Repräsentation auf dem Performanz-Niveau einer zufälligen Aktionswahl, also bei Werten um 0.5.
Das XCS mit Unordered-Bound-Repräsentation erzielt eine deutliche bessere Performanz, die die HCS am Ende sogar leicht übersteigt: Über die letzten 10000 Episoden gemittelt erzielt das XCS mit Unordered-Bound-Repräsentation eine Performanz von 0.983, das HCS „nur“ eine Performanz von 0.973. Entsprechend sinkt auch der Systemfehler des XCS gegen Ende des Experiments unter den des HCS. Hierfür benötigt das XCS allerdings auch eine deutlich größere Population, deren Größe den optimalen Wert deutlich überschreitet: Die optimale Population besteht sowohl beim XCS wie auch beim HCS aus $2 \cdot 3^3 = 54$ Klassifizierern, das entspricht bei einer maximal zulässigen Anzahl von$N_{\max} = 2000$ Klassifizierern einer Populationsgröße von 0.027. Während das XCS diesen Wert – in allen 10 Läufen des Experiments – genau erreicht, liegt der über die Läufe gemittelte Endwert der Populationsgröße beim XCS mit Unordered-Bound-Repräsentation bei 0.24, also bei etwa dem neunfachen des optimalen Wertes. Wie schon im zweidimensionalen Fall gelingt es dem HCS auch hier bei allen Läufen, die Klassifizierer annähernd optimal, also in den Zentren der das dreidimensionale Checkerboard bildenden Würfel, zu positionieren. Eine dieser Populationen zeigt die Abbildung 6.11.



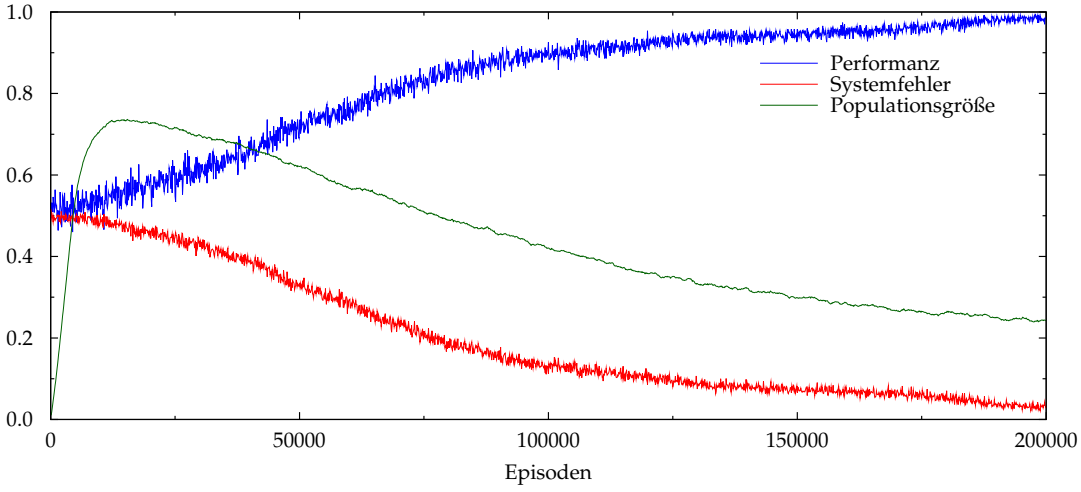
(a) **(b)**

Abbildung 6.11: Endpopulation des HCS beim dreidimensionalen Checkerboard mit drei Unterteilungen. **(a)** Klassifizierer mit Aktion 0. **(b)** Klassifizierer mit Aktion 1.

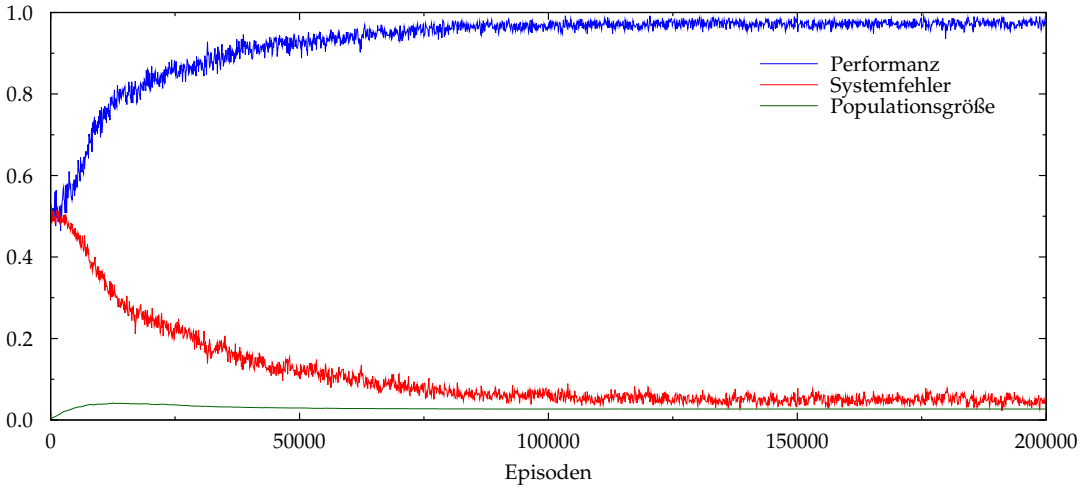
[16] Eine Ursache für diese Diskrepanz könnte die Verwendung einer anderen Darstellung reeller Zahlen sein. In [Stone u. Bull 2003] gibt es Hinweise, jedoch keine klare Aussage, dass – anders als in der hier verwendeten Implementation des XCS – keine Gleitkommadarstellung reeller Zahlen verwendet wurde.



(a)



(b)



(c)

Abbildung 6.12: Zeitliche Entwicklung von Performanz, Systemfehler und Populationsgröße bei Anwendung des **(a)** XCS mit Centre-Spread-Bedingungen, **(b)** XCS mit Unordered-Bound-Bedingungen und **(c)** HCS auf das dreidimensionale Checkerboard mit drei Unterteilungen.

6.5 Zusammenfassung

In diesem Kapitel wurden die bei Anwendung des HCS auf je zwei Vertreter der Multiplexer- und der Checkerboard-Familie von Klassifikationsproblemen erzielten Ergebnisse vorgestellt und analysiert. Zum Vergleich herangezogen wurden mit zwei Varianten des XCS durchgeführte Experimente.
Im Falle der Multiplexer-Probleme blieb die Leistung des HCS hinter der des XCS zurück. Als Ursache hierfür konnte die besondere Struktur des Zustandsraums dieser Probleme identifiziert werden, an die das XCS mit intervallbasierten Bedingungen perfekt angepasst ist. Mit einer Performanz größer als 0.95, also mehr als 95% richtig klassifizierten Eingaben, kann die Leistung des HCS aber auch nicht als schlecht angesehen werden.
Im Falle der Checkerboard-Probleme sind die erzielten Ergebnisse weniger eindeutig: Während das HCS beim zweidimensionalen Checkerboard mit fünf Unterteilungen eine bedeutend höhere Performanz erzielt als beide XCS-Varianten, wird es beim dreidimensionalen Checkerboard mit drei Unterteilungen vom XCS mit Unordered-Bound-Bedingungen geringfügig übertroffen. Dabei lernt das XCS jedoch deutlich langsamer als das HCS. Während letzteres im Mittel bereits nach 50000 Lernepisoden eine Performanz von 0.90 erreicht, benötigt das XCS hierfür etwa doppelt so lange. Anders als das XCS mit Unordered-Bound-Bedingungen weist das XCS mit Centre-Spread-Bedingungen auch bei der dreidimensionalen Checkerboard-Variante eine sehr niedrige Performanz auf.
Eine falsche Klassifikation ist stets darauf zurückzuführen, dass das Lernende Klassifizierende System die bei Ausführung der verschiedenen Aktionen respektive Klassifikationen zu erwartenden Returns falsch einschätzt. Dementsprechend weist das HCS bei allen Problemen, bei denen es eine niedrigere Performanz als das XCS erzielt, auch einen höheren Systemfehler als dieses auf.
Auf die Frage, warum das HCS die zu erwartenden Returns schlechter abschätzen kann als das XCS, wurde hier nicht eingegangen. Diese Fragestellung lässt sich, da die Abschätzung zu erwartender Returns nichts anderes ist als eine Approximation der Aktions-Wertefunktion der jeweils betrachteten Lernumgebung, besser in Kapitel 7 behandeln, das sich mit dem Einsatz des HCS zur Funktionsapproximation befasst. Dort wird dargelegt werden, dass stückweise konstante Funktionen – wie die Aktions-Wertefunktion der in diesem Kapitel betrachteten Lernprobleme – für das HCS schwieriger zu approximieren sind als für das XCS mit intervallbasierten Bedingungen.
Positiv zu vermerken ist noch, dass das HCS bei allen in diesem Kapitel betrachteten Problemen deutlich kleinere – also aus weniger Makroklassifizierern bestehende – Populationen evolviert als beide XCS-Varianten. Eine genauere Betrachtung zeigt sogar, dass diese Populationen im Wesentlichen den für die Lösung der betrachteten Klassifikationsprobleme optimalen HCS-Populationen entsprechen.

Kapitel 7

Funktionsapproximation mit dem HCS

Bei der Funktionsapproximation besteht die Aufgabe eines Lernenden Klassifizierenden Systems darin, eine durch die Lernumgebung vorgegebene reellwertige Funktion $f : \mathcal{S} \rightarrow \mathbb{R}$ möglichst gut zu approximieren. Bei XCS und HCS ist dabei der vom System geschätzte Funktionswert an einer Stelle $s \in \mathcal{S}$ durch die Systemvorhersage für die Ausführung der einzigen[1] in einer Funktionsapproximations-Lernumgebung bekannten (Platzhalter-)Aktion a im Zustand s gegeben.
Bei Funktionsapproximations-Problemen stellt – wie bereits bei den im letzten Kapitel betrachteten Klassifikations-Problemen – jeder Lernschritt eine eigene Lernepisode dar, die in der folgenden Weise abläuft: Nachdem das Lernende Klassifizierende System den Zustand s der Lernumgebung wahrgenommen hat, berechnet es die Systemvorhersage für diesen und führt dann die Platzhalter-Aktion a aus. In Gestalt der anschließend gewährten Belohnung übermittelt daraufhin die Lernumgebung dem Lernenden Klassifizierenden System den korrekten Funktionswert $f(s)$ der zu approximierenden Funktion an der Stelle s. Dann kommen in der üblichen Weise die Reinforcement- und gegebenenfalls die Discovery-Komponente zum Einsatz, bevor mit der Präsentation eines neuen Zustandes die nächste Lernepisode beginnt. Im Falle der Funktionsapproximation lernt ein Lernendes Klassifizierendes System folglich überwacht, das heißt aus Eingaben und den zugehörigen korrekten Ausgaben.
Die Durchführung der Experimente zur Funktionsapproximation mit dem HCS erfolgt in der bereits im vorangegangenen Kapitel geschilderten Weise, lediglich die Beurteilung der Ergebnisse geschieht – der veränderten Zielsetzung im Falle der Funktionsapproximation Rechnung tragend – nach teilweise anderen Kriterien, worauf in Abschnitt 7.1 eingegangen wird. Anschließend werden in den Abschnitten 7.2 und 7.3 die Ergebnisse der mit dem HCS zur Approximation von – größtenteils der Literatur entnommenen – Funktionen durchgeführten Experimente präsentiert und mit unter Einsatz des XCS erzielten verglichen. Abschnitt 7.4 fasst die wesentlichen Ergebnisse des Kapitels zusammen.

[1] Vergleiche hierzu Abschnitt 2.4.8.

7.1 Beurteilung der Approximationsgüte

Im Falle des Einsatzes zur Funktionsapproximation ist das primäre Kriterium zur Beurteilung der Leistung eines Lernenden Klassifizierenden Systems offensichtlich die Qualität der gelernten Approximation. Um diese zu quantifizieren, wird hier der *mittlere absolute Fehler* (*MAE, mean absolute error*) verwendet, also die über eine Stichprobenmenge $S \subset \mathcal{S}$ gemittelte absolute Abweichung des vom System vorhergesagten Funktionswertes $\hat{f}(x)$ vom tatsächlichen Wert $f(x)$ der zu approximierenden Funktion:

$$\mathrm{MAE} = \frac{1}{|S|} \sum_{x \in S} \left| f(x) - \hat{f}(x) \right| \tag{7.1}$$

Während eines Laufes eines Lernenden Klassifizierenden Systems dienten bei allen hier durchgeführten Experimenten – analog dem Vorgehen bei der Berechnung des Systemfehlers[2] in Kapitel 6 – jeweils die letzten $n = 50$ im Exploitationsmodus präsentierten Zustände als Basis für die Berechnung des MAE, was eine gute Beurteilung seiner zeitlichen Entwicklung während des Laufes gestattet.
Zusätzlich wurde in den Experimenten am Ende jedes Einzellaufes der MAE auf einer größeren Stichprobe berechnet, um eine genauere Beurteilung der Qualität der letztendlich erzielten Approximation zu erhalten. Ferner gestattet dies, da stets die gleiche Stichprobenmenge S zugrunde gelegt wird, eine Einschätzung, wie stark die Qualität der erzielten Approximation – über die Läufe eines Experimentes betrachtet – variiert.

Als zweites Beurteilungskriterium wird wieder die Größe der evolvierten Population betrachtet. Diese dient hier dazu, die relative Generalisierungsfähigkeit von HCS und XCS einzuschätzen: Approximieren zwei Lernende Klassifizierende Systeme eine Funktion gleich gut, generalisiert offenbar das System besser, das weniger Klassifizierer benötigt. Da der Fall einer wirklich gleich guten Approximation natürlich nie eintritt, ist dies allerdings ein sehr vages Kriterium, das wirklich nur eine grobe Einschätzung ermöglicht. Eine Beurteilung der absoluten Generalisierungsfähigkeit anhand eines Vergleiches der Größe der evolvierten Population mit der Größe einer idealen Population, wie sie in Kapitel 6 angesprochen wurde, ist im Falle der Funktionsapproximation nicht möglich: Da außer in Ausnahmefällen – etwa der weiter unten betrachteten Funktion f_1 – eine exakte Approximation (also $\hat{f}(x) = f(x)$ für alle $s \in \mathcal{S}$) nicht möglich ist, ist es nicht einmal sinnvoll, von einer idealen Populationsgröße zu sprechen, da die Approximationsgüte – zumindest theoretisch – durch Hinzunahme weiterer Klassifizierer zu einer Population stets verbessert werden kann[3].
Die Betrachtung der Populationsgröße ist ferner auch unter einem eher praktischen Gesichtspunkt von Interesse: Da die von einem Lernenden Klassifizierenden System benötigte Rechenzeit wesentlich durch die Anzahl der in seiner Population enthaltenen

[2] Wie ein Vergleich mit 6.2 zeigt, entspricht der Systemfehler, wenn von der Normierung abgesehen wird, dem MAE.

[3] Wird etwa eine auf einem Intervall $[a, b)$ definierte Funktion f unter Verwendung eines HCS approximiert, so ist – unter der vereinfachenden Annahme, dass der Vorhersagefehler aller Klassifizierer unter dem Schwellenwert ε_p liegt – der approximierte Funktionswert $\hat{f}(x)$ gerade durch die Vorhersage des Gewinners für den Zustand x gegeben. Die Funktion f wird dann durch eine stückweise konstante (konstante Klassifizierer-Vorhersagen) beziehungsweise stückweise lineare Funktion (berechnete Klassifizierer-Vorhersagen) approximiert. Durch Hinzunahme weiterer Klassifizierer (mit passenden Vorhersagen) können die Intervalle, auf denen die Approximation konstant respektive linear ist, stets verkleinert, die Approximation also verbessert werden.

(Makro-)Klassifizierer beeinflusst wird, stellt es einen Vorteil dar, wenn ein System kleinere Populationen evolviert[4].

Als weiteres Beurteilungskriterium kann die in Lernepisoden gemessene Zeit, die zum Lernen einer guten Approximation benötigt wird, dienen. Eine Angabe der tatsächlichen Rechenzeit ist weniger sinnvoll, da diese stets implementationsabhängig ist und ferner wesentlich durch die verwendete Hardware bestimmt wird.

Zum Vergleich wird in diesem Kapitel stets das XCS in der – bei Funktionsapproximationsaufgaben üblicherweise verwendeten – Variante mit Ordered-Bound-Intervallrepräsentation herangezogen. Die Parameter für das XCS wurden in Abschnitt 7.2 entsprechend den in [Lanzi u. a. 2005a] angegebenen gewählt, die in Abschnitt 7.3 benutzten basieren auf aus [Butz 2005] entnommenen.
Die Parameter für das HCS wurden nicht für jede der hier betrachteten Funktionen einzeln optimiert, vielmehr wurden nur drei verschiedene Parametersätze verwendet, einer für die eindimensionalen – in Abschnitt 7.2 betrachteten – Funktionen, die beiden anderen für je drei der zweidimensionalen Funktionen in Abschnitt 7.3.
Sowohl das XCS als auch das HCS wurden in den in diesem Kapitel diskutierten Experimenten stets mit gemäß (2.38) respektive (4.7) berechneten Klassifizierervorhersagen verwendet. Die Anpassung der Gewichte für diese Berechnung erfolgte jeweils basierend auf der Methode der kleinsten Quadrate, wie für das XCS in Abschnitt 2.4.8 und für das HCS in Abschnitt 4.4 beschrieben[5].

7.2 Approximation eindimensionaler Funktionen

In einer ersten Reihe von Experimenten wurde das HCS eingesetzt, um vier aus [Lanzi u. a. 2005a] entnommene, eindimensionale Funktionen auf dem Intervall $[0, 1)$ zu approximieren. Dabei handelt es sich um das Polynom

$$f_{\mathrm{P}}(x) = 1 + (2x - 1) + (2x - 1)^2 + (2x - 1)^3, \tag{7.2}$$

die Überlagerungen von Sinus-Funktionen

$$f_{\mathrm{s3}}(x) = \sin(2\pi x) + \sin(4\pi x) + \sin(6\pi x) \tag{7.3}$$

und

$$f_{\mathrm{s4}}(x) = \sin(2\pi x) + \sin(4\pi x) + \sin(6\pi x) + \sin(8\pi x) \tag{7.4}$$

sowie die Funktion

$$f_{\mathrm{abs}}(x) = |\sin(2\pi x) + |\cos(2\pi x)|| \,. \tag{7.5}$$

Die Graphen dieser Funktionen zeigt die Abbildung 7.1, die Ergebnisse der durchgeführten Experimente, also die über die 10 Einzelläufe jedes Experimentes gemittelte zeitliche Entwicklung von MAE und Populationsgröße bei HCS und XCS gibt die Abbildung 7.2 wieder.

[4] Bei der praktischen Anwendung von Verfahren aus dem Bereich der Künstlichen Intelligenz respektive des maschinellen Lernens werden oft sogar Abstriche bei der Qualität einer Lösung in Kauf genommen, wenn diese dafür schnell verfügbar ist.

[5] Für die Berechnung wurden jeweils die letzten 50 Zustände, zu denen ein Klassifizierer passte – respektive ein Gewinner war –, verwendet.

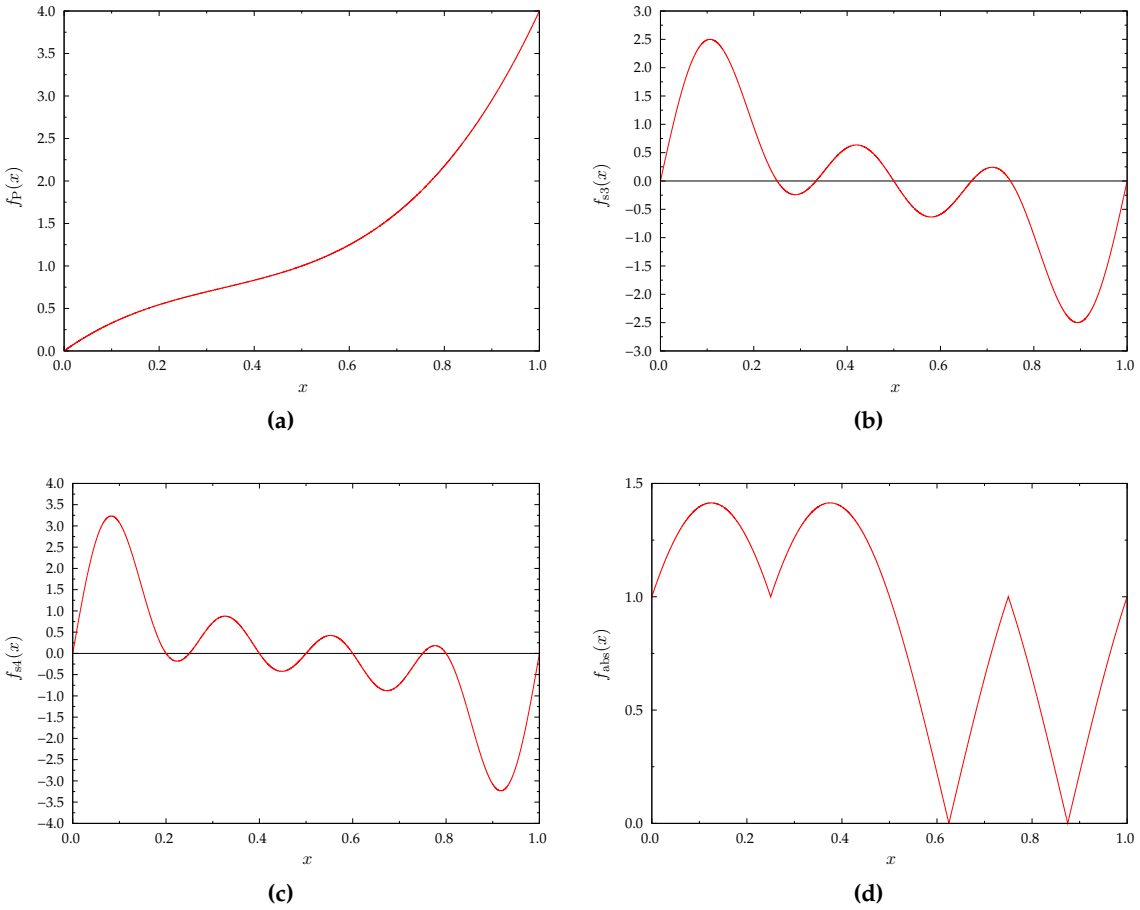
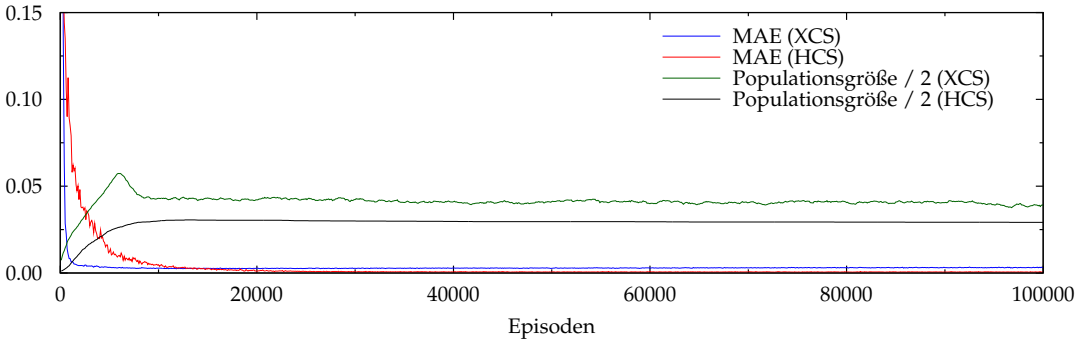


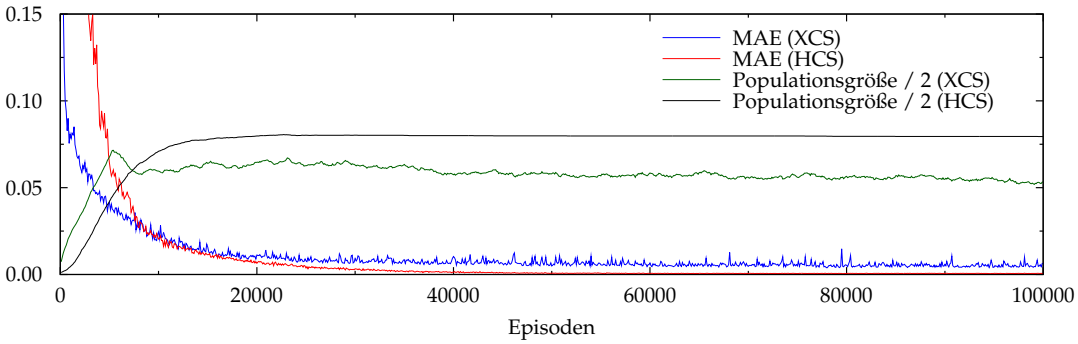
Abbildung 7.1: Die verwendeten Testfunktionen: **(a)** dasPolynom f_{P}, **(b)** die Summe f_{s3} von drei Sinusfunktionen, **(c)** die Summe f_{s4} von vier Sinusfunktionen sowie **(d)** die Funktion f_{abs}.

Die Entwicklung der Approximationsleistung erfolgt bei allen vier Testfunktionen in ähnlicher Weise: Zunächst sinkt der MAE beim XCS schneller als beim HCS, nach etwa 10000 bis 15000 Lernepisoden unterschreitet jedoch der MAE des HCS den des XCS. Bei allen vier Funktionen gelingt dem HCS eine bessere Approximation der Zielfunktion als dem XCS.
Deutlichere Unterschiede zwischen den vier Experimenten zeigen sich hinsichtlich der Populationsgröße: Im Falle des Poynoms f_{P} liegt die Populationsgröße des HCS unter der des XCS, während bei den Funktionen f_{s3} und f_{s4} das HCS die größeren Populationen evolviert. Bei der Testfunktion f_{abs} schließlich evolvieren beide Systeme Populationen von etwa gleicher Größe.

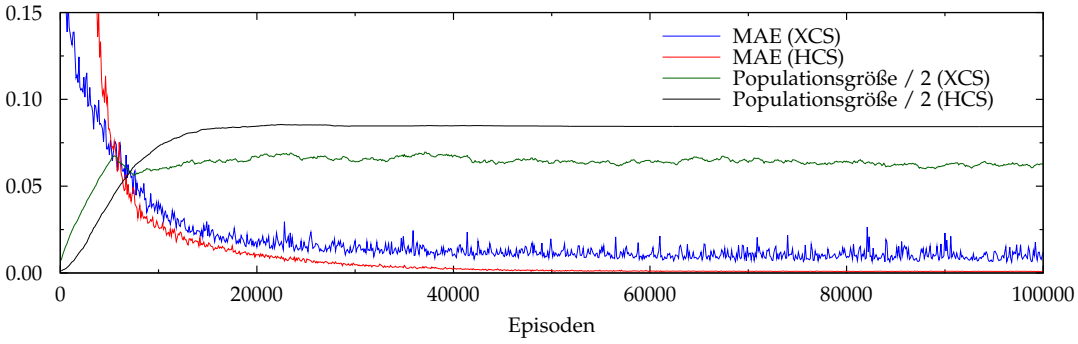
Eine Voraussetzung für das Erlernen einer – auch nur einigermaßen brauchbaren – Approximation einer Zielfunktion besteht offensichtlich darin, dass genügend Klassifizierer in der Population enthalten sind, um den Zustandsraum in einer dem Problem angemessenen Weise abzudecken. Diese Voraussetzung müssen die hier betrachteten Lernenden Klassifizierenden Systeme erst schaffen, indem sie ihre – zunächst leere – Population „füllen“. Das zunächst schnellere Absinken des MAE beim XCS lässt sich daher durch die Unterschiede in der Art und Weise, wie das HCS und das XCS Klassifizierer erzeugen, erklären: Beim XCS wird die Population zu Beginn eines Laufes relativ schnell durch den Covering-Operator gefüllt: Sobald ein Zustand auftritt, zu dem es noch keinen passenden



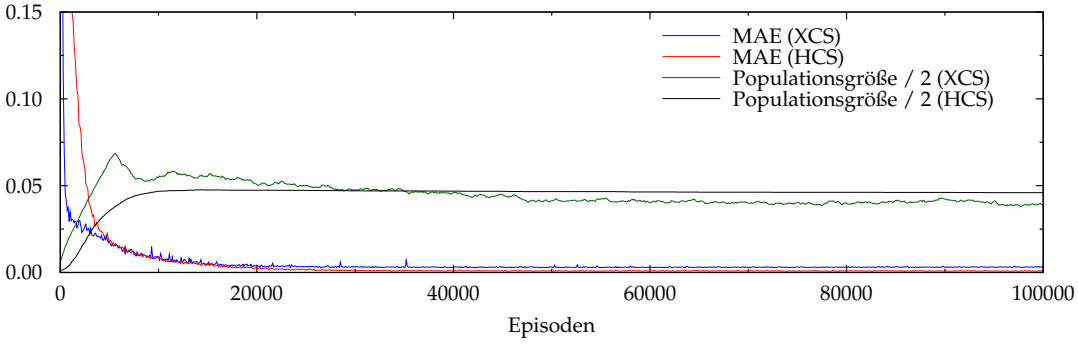
(a)



(b)



(c)



(d)

Abbildung 7.2: Zeitliche Entwicklung von MAE und Populationsgröße bei Einsatz des XCS und des HCS zur Approximation **(a)** des Polynoms f_{P}, **(b)** der Überlagerung dreier Sinusfunktionen f_{s3} **(c)** der Überlagerung vierer Sinusfunktionen f_{s4} und **(d)** der Funktion f_{abs}.

Klassifizierer gibt, wird ein solcher erzeugt. Ferner fügt auch der Genetische Algorithmus neue Klassifizierer in die Population ein. Das HCS hingegen verwendet den Covering-Operator (im Regelfall) nur zur Erzeugung der ersten Klassifizierer der Population, alle weiteren Klassifizierer müssen durch den Genetischen Algorithmus erzeugt werden, der zudem auch seltener ausgeführt wird als beim XCS. Folglich wächst, wie in der Abbildung 7.2 auch klar zu sehen ist, die Population beim HCS langsamer an als beim XCS – es dauert beim HCS also länger, bis die genannte Voraussetzung erfüllt ist. Hinzu kommt, dass in dieser Anfangsphase des Lernprozesses die Klassifizierer noch relativ stark durch den Netzlernmechanismus des HCS bewegt werden. Die damit einhergehenden Veränderungen der Matchbereiche/Voronoizellen der Klassifizierer erschweren natürlich die Anpassung der Gewichte für die Berechnung von in den jeweiligen Matchbereichen zutreffenden Klassifizierervorhersagen.
Dass diese Überlegungen zutreffen, wird anhand der Abbildung 7.3 ersichtlich. Diese stellt den bereits in der Abbildung 7.2(b) gezeigten Lernverläufen von XCS und HCS bei der Approximation der Funktion f_{s3} den Lernverlauf eines HCS gegenüber, das mit einer aus 25 Klassifizierern bestehenden Startpopulation initialisiert wurde. Bei diesem sinkt der MAE schneller als sowohl beim HCS ohne Startpopulation als auch beim XCS.

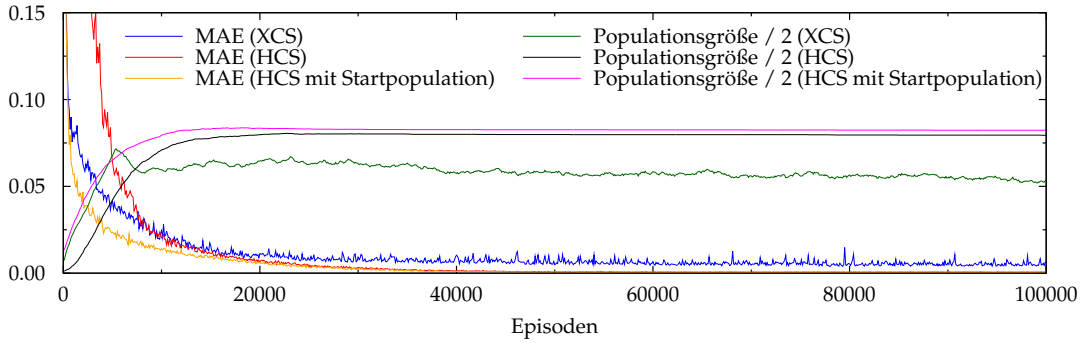


Abbildung 7.3: Zeitliche Entwicklung von MAE und Populationsgröße bei Einsatz des XCS, des HCS ohneStartpopulation zur Approximation von f_{s3}.

Um die Qualität der gelernten Approximation im Detail zu betrachten, wurden in den hier durchgeführten Experimenten am Ende jedes Laufes die approximierten Funktionswerte $\hat{f}(x)$ für 100 äquidistant im Intervall $[0, 1)$ verteilte Punkte bestimmt.
Die Mittelwerte der für jeden dieser 100 Punkte in den 10 Läufen eines Experimentes geschätzten Funktionswerte sind in der Abbildung 7.4 eingetragen. Die Fehlerbalken geben die zugehörige Standardabweichung an. Diese spiegelt in erster Linie wider, wie stark die Schätzung des Funktionswertes an der jeweiligen Stelle – über die Läufe eines Experimentes betrachtet – variiert. Da aber, wie der Abbildung zu entnehmen ist, die Mittelwerte der geschätzten Funktionswerte den wahren Funktionswert an fast allen Stellen sehr gut wiedergeben, stellt die Standardabweichung hier auch einen Indikator dafür da, wie stark die vorhergesagten Funktionswerte – über die Läufe betrachtet – vom wahren Wert der zu approximierenden Funktion abweichen.
Es zeigt sich, dass die Abweichungen beim HCS nahezu durchgängig kleiner sind als beim XCS. Insbesondere die Approximation der Maxima und Minima der Funktionen f_{s3}, f_{s4} und f_{abs} fällt dem XCS offenbar deutlich schwerer als dem HCS. Dies schlägt sich auch in der sehr unruhigen Entwicklung des MAE des XCS bei der Approximation der Funktionen f_{s3} und f_{s4} nieder. Zwar variieren die vorgenommenen Schätzungen auch beim HCS im Bereich der Optima etwas stärker als an den übrigen Stellen, jedoch keineswegs in dem Maße wie beim XCS.

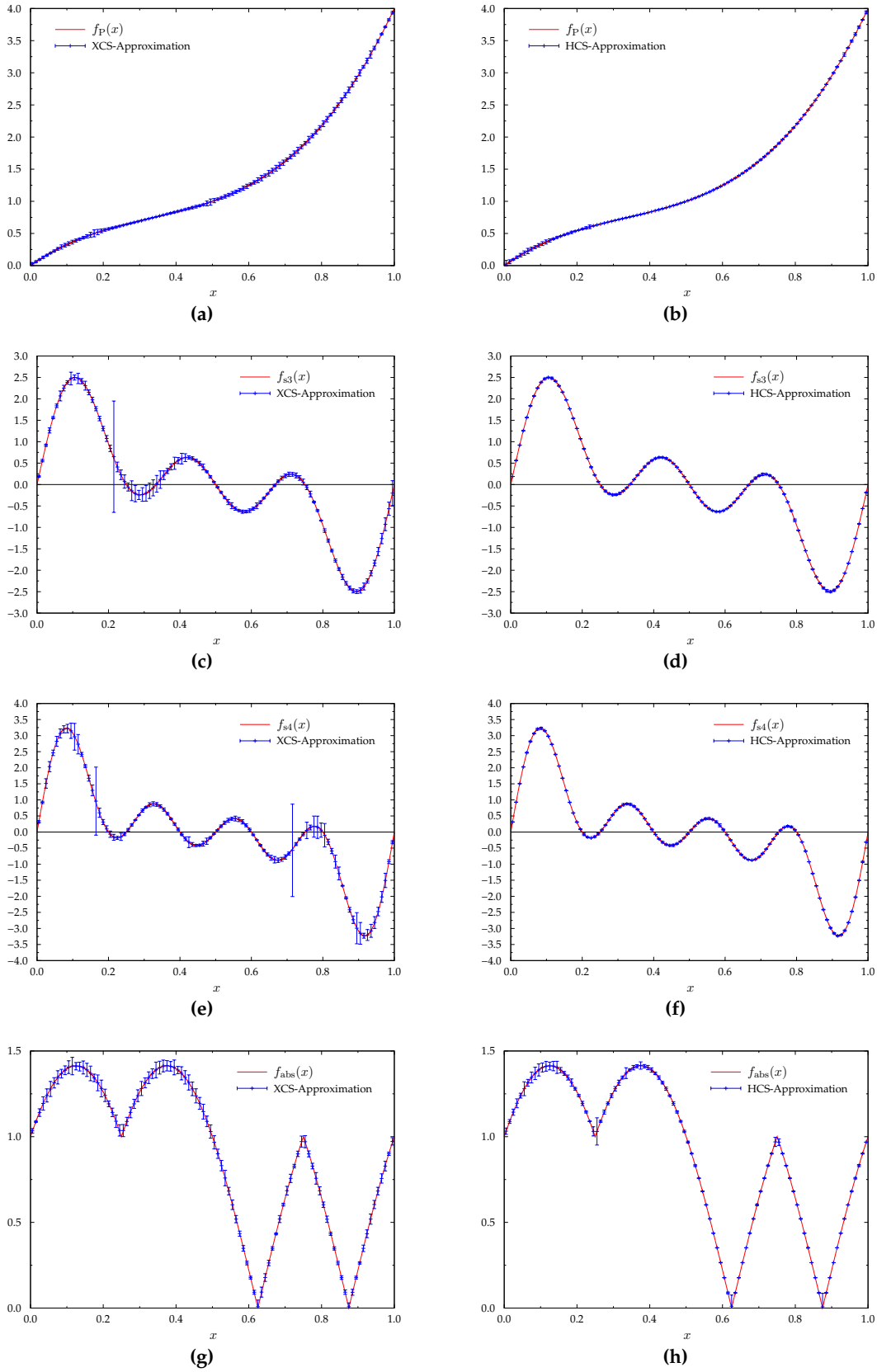


Abbildung 7.4: Durchschnittlicher approximierter Funktionswert beim XCS (jeweils links) und HCS (jeweils rechts) im Falle des Polynoms f_{P} (**(a)** und **(b)**), der Summe f_{s3} von drei Sinusfunktionen (**(c)** und **(d)**), der Summe f_{s4} von vier Sinusfunktionen (**(e)** und **(f)**) sowie der Funktion f_{abs} (**(g)** und **(h)**). Die Größe der Fehlerbalken entspricht dem zehnfachen der Standardabweichung der approximierten Funktionswerte.

In einem speziellen Fall jedoch fällt die korrekte Approximation der Zielfunktion dem HCS schwerer: An den Stellen, an denen die Funktion f_{abs} nicht differenzierbar ist, variieren die Vorhersagen des HCS stärker als die des XCS. Dies kann wie folgt erklärt werden: Liegt eine Undifferenzierbarkeitsstelle „mitten“ im Matchbereich eines Klassifizierers, passt dieser also sowohl zu kleineren als auch zu größeren Zuständen, wird seine als lineare Funktion des Zustandes berechnete Vorhersage eine Gerade beschreiben, die die Spitze der zu approximierenden Funktion „abschneidet“. Für eine gute Approximation einer Funktion an einer solchen Stelle werden also Klassifizierer benötigt, die diese nur am Rand ihres Matchbereiches beinhalten.
In den Abbildungen 7.5(a) und 7.5(b) stellt jede der kurzen Linien die als lineare Funktion des Zustandes berechnete Vorhersage eines Klassifizierers in dessen Matchbereich dar. Die Grenzen der Matchbereiche der Klassifizierer sind durch Kreuze gekennzeichnet. (Um die Darstellung der Klassifizierervorhersagen besser erkennbar zu machen, wurde die Zielfunktion f_{abs} um 0.01 nach unten verschoben aufgetragen.) Es ist zu erkennen, dass sowohl das XCS als auch das HCS in der Regel Klassifizierer der benötigten Form evolviert haben. Beide Systeme haben aber auch Klassifizierer hervorgebracht, deren Matchbereich eine Undifferenzierbarkeitsstelle mittig enthält; etwa bei $x = 0.25$.

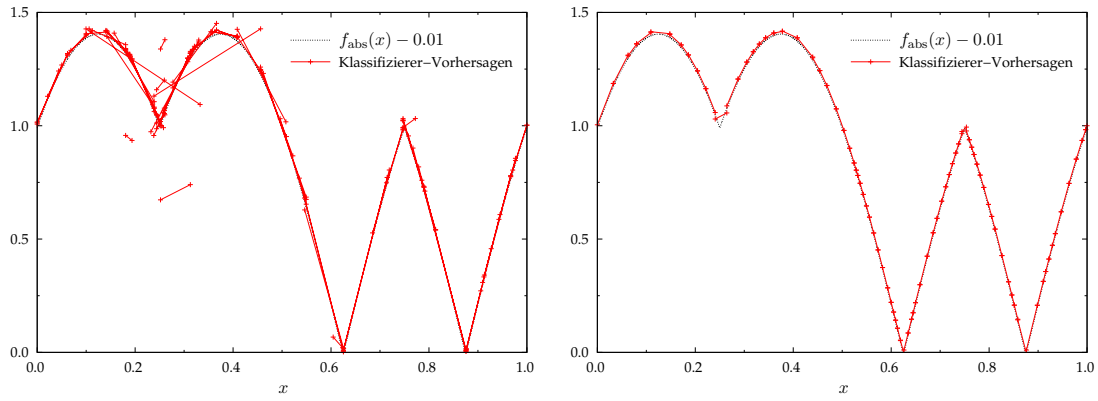


Abbildung 7.5: Vorhersagen der Klassifizierer einer **(a)** vom XCS und **(b)** vom HCS zur Approximation der Funktion f_{abs} evolvierten Population.

Beim XCS ist dies jedoch weniger problematisch: Die Systemvorhersage – also der approximierte Funktionswert – wird ja als gewichtetes Mittel der Vorhersagen aller an der jeweiligen Stelle passenden Klassifizierer berechnet. Beim HCS hingegen bildet die Vorhersage des einen passenden Klassifizierers die Basis für die Berechnung der Systemvorhersage, im Falle eines eindimensionalen Zustandsraums geht in der Regel nur ein weiterer benachbarter Klassifizierer mit ein. Sobald also beim HCS ein Klassifizierer auftritt, der eine Undifferenzierbarkeitsstelle mittig enthält, wird die Funktion im Bereich dieser Stelle schlecht approximiert werden. Dass derartige Klassifizierer erzeugt werden, ist jedoch nicht zu vermeiden: Wird dem HCS ein Zustand in direkter Nähe einer Undifferenzierbarkeitsstelle präsentiert, so wird das gebildete Match-Set nahezu immer sowohl links als auch rechts der Undifferenzierbarkeitsstelle positionierte Klassifizierer enthalten. Werden diese als Elternklassifizierer für den Genetischen Algorithmus selektiert, werden die durch Intermediäres Crossing-Over[6] gebildeten Kind-Klassifizierer im Allgemeinen die angegebene ungünstige Gestalt haben.

[6]Der Einsatz von Uniformem Crossing-Over oder Zwei-Punkt-Crossing-Over ist zum Zwecke der genetischen Variation eindimensionaler Gewichtsvektoren offenbar nicht sinnvoll.

7.3 Approximation zweidimensionaler Funktionen

In einer zweiten Reihe von Experimenten wurden HCS und XCS zur Approximation von auf dem Einheitsquadrat $[0,1) \times [0,1)$ definierten Funktionen eingesetzt, die zum Teil aus [Lanzi u. Wilson 2006] entnommen wurden.

Bei den beiden ersten der betrachteten Funktionen handelt es sich um Stufenfunktionen:

$$f_1(x,y) = \frac{1}{3}\left(\lfloor 3x \rfloor \mod 3\right) + \frac{1}{3}\left(\lfloor 3y \rfloor \mod 3\right) \tag{7.6}$$

$$f_2(x,y) = \frac{1}{6}\left(\lfloor 2\,(x+y) \rfloor \mod 4\right) \tag{7.7}$$

Die Funktionen f_1 und f_2 unterscheiden sich in erster Linie durch ihre Orientierung im Zustandsraum; während die Stufen der Funktion f_1 achsenparallel ausgerichtet sind, liegen die der Funktion f_2 diagonal im Zustandsraum. Ein weiterer Unterschied besteht darin, dass bei der Funktion f_1 an vier Stellen Stufen dreier verschiedener Höhen direkt aufeinander treffen, während bei der Funktion f_2 stets nur zwei Stufen aneinander angrenzen.
Als dritte Funktion wurde die stückweise lineare stetige Funktion

$$f_{\text{roof}}(x,y) = \min(x+y, 2-(x+y)) \tag{7.8}$$

betrachtet, deren Graph an ein Dach erinnert. Ferner wurden die Sinusfunktion

$$f_3(x,y) = \sin\left(2\pi(x+y)\right) \tag{7.9}$$

sowie die als Maximum dreier Exponentialfunktionen definierte Funktion

$$f_{\text{e3}}(x,y) = \max\left(e^{-10(2x-1)^2}, e^{-50(2y-1)^2}, \frac{5}{4}e^{(-5(2x-1)^2+(2y-1)^2)}\right) \tag{7.10}$$

untersucht. Schließlich wurde noch eine Variante der Funktion f_{e3} betrachtet, die entsteht, wenn f_{e3} um 45° um die durch den Punkt (0.5; 0.5; 0) gehende Parallele zur z-Achse gedreht wird. Mit der Transformation

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos(\pi/4) & -\sin(\pi/4) \\ \sin(\pi/4) & \cos(\pi/4) \end{pmatrix} \begin{pmatrix} x-0.5 \\ y-0.5 \end{pmatrix} + \begin{pmatrix} 0.5 \\ 0.5 \end{pmatrix} \tag{7.11}$$

kann diese Funktion in der Form

$$f_{\text{e3r}}(x,y) = f_{\text{e3}}(x',y') \tag{7.12}$$

geschrieben werden. Die Graphen aller sechs Funktionen sind in Abbildung 7.6 dargestellt.

Die Lernverläufe von XCS und HCS bei der Approximation dieser Funktionen – also die über die Läufe des jeweiligen Experimentes gemittelte Entwicklung von MAE und Populationsgröße – werden in den Abbildungen 7.7 und 7.8 einander gegenübergestellt. Außer im Falle der Funktion f_{roof}, bei der sich der MAE bei beiden Systemen nahezu gleich entwickelt, sinkt der MAE des HCS bei allen Zielfunktionen zunächst deutlich schneller ab als der des XCS. Diesen „Vorsprung“ kann das HCS jedoch nur bei den Funktionen f_3 und f_{e3r} dauerhaft halten. Die Funktionen f_2 und f_{roof} werden am Ende der Experimente von HCS und XCS etwa gleich gut approximiert, bei der Funktion f_{e3} erreicht das XCS einen etwas niedrigeren, bei f_1 einen deutlich niedrigeren MAE als das HCS. Einheitlicher als die des MAE zeigt sich die Entwicklung der Populationsgröße: In allen Experimenten entwickelt das XCS deutlich größere Populationen als das HCS.

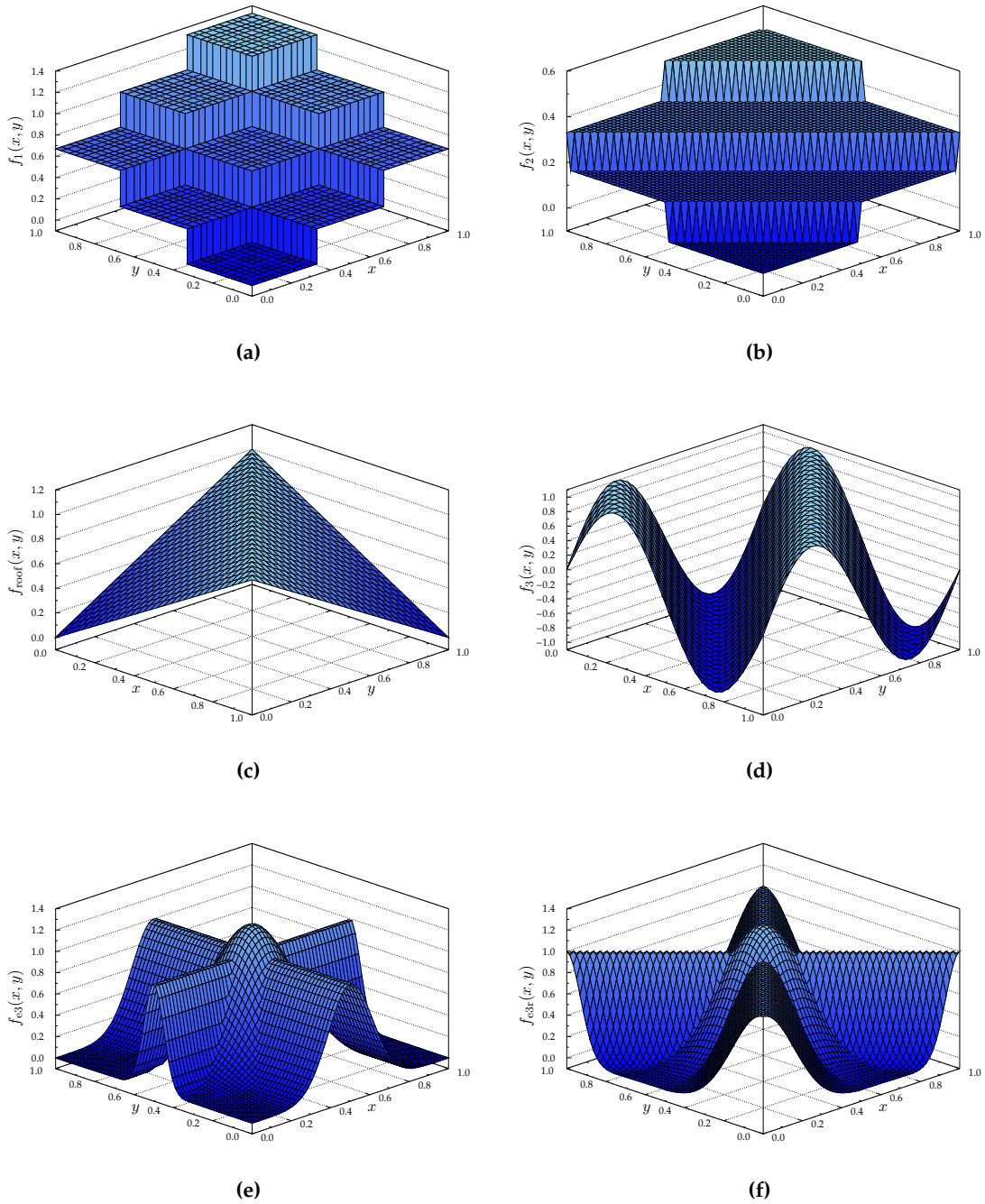
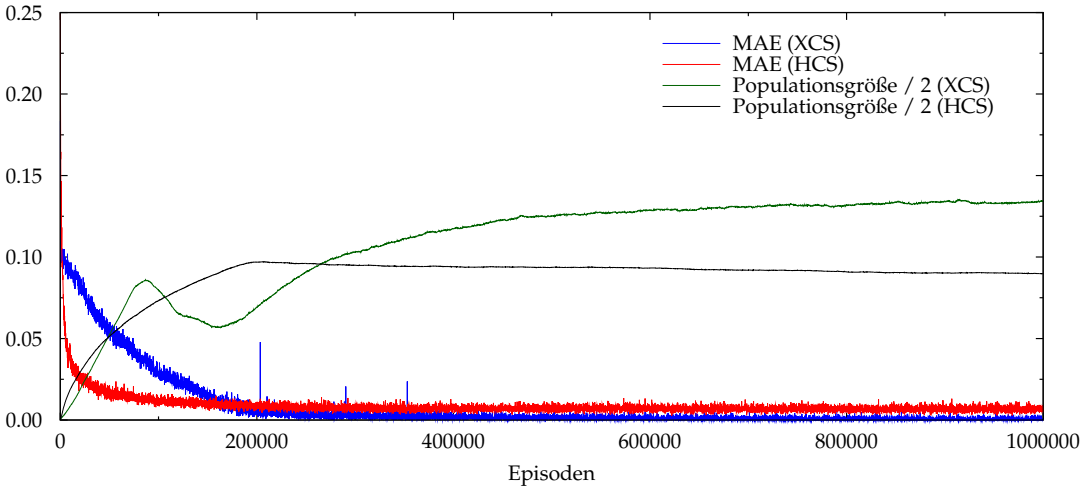
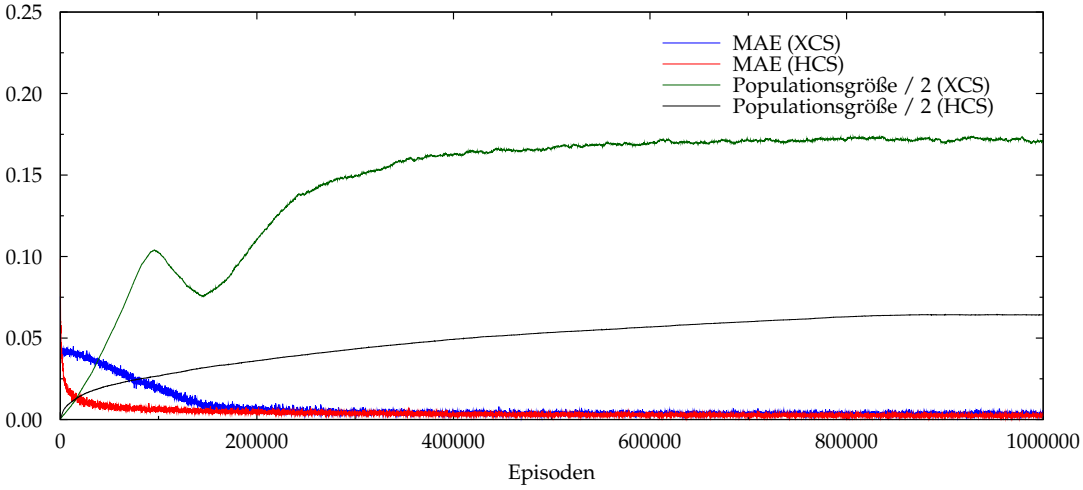


Abbildung 7.6: Die verwendeten zweidimensionalen Testfunktionen: **(a)** die Stufenfunktion f_1, **(b)** die Stufenfunktion f_2, **(c)** die Dachfunktion f_{roof}, **(d)** die Sinusfunktion f_3, **(e)** die Überlagerung von Exponentialfunktionen f_{e3} und **(f)** f_{e3r}, die rotierte Variante von f_{e3}.

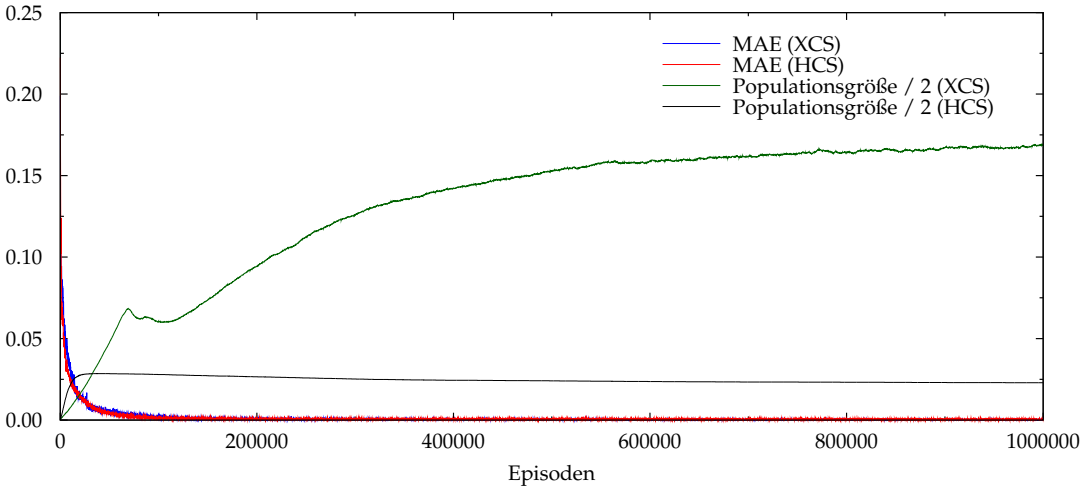
Analog zum Vorgehen bei den Experimenten zur Approximation eindimensionaler Funktionen wurde am Ende jedes Einzellaufes eines Experimentes der approximierte Funktionswert an 100×100 gitterartig im Einheitsquadrat $[0,1)^2$ gelegenen Stellen ermittelt. Aus den derart für jede dieser Stellen im Verlauf eines Experimentes erhaltenen 10 Werten wurden Mittelwert und Standardabweichung des approximierten Funktionswertes an der jeweiligen Stelle berechnet. Die so erhaltenen durchschnittlichen Approximationen der Zielfunktionen durch XCS und HCS sind in den Abbildungen 7.10 und



(a)

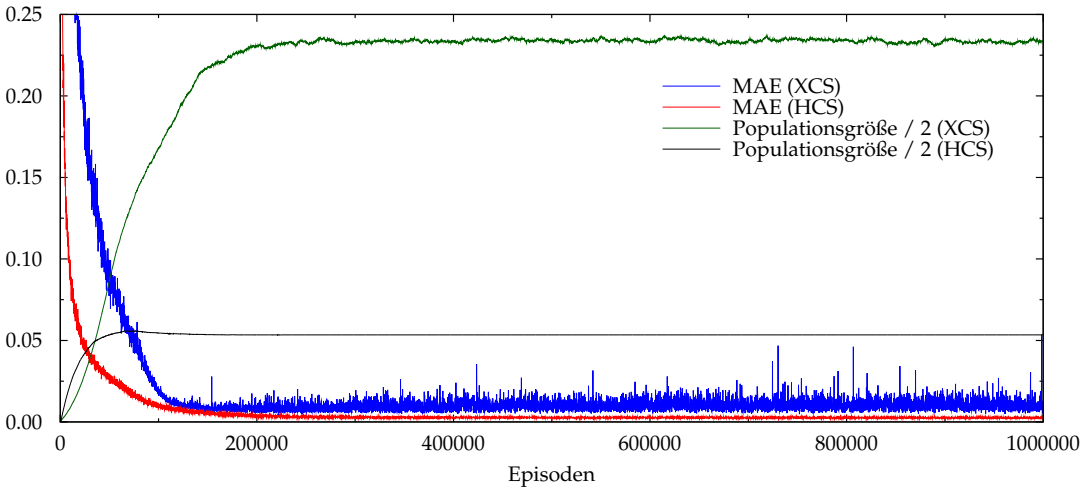


(b)

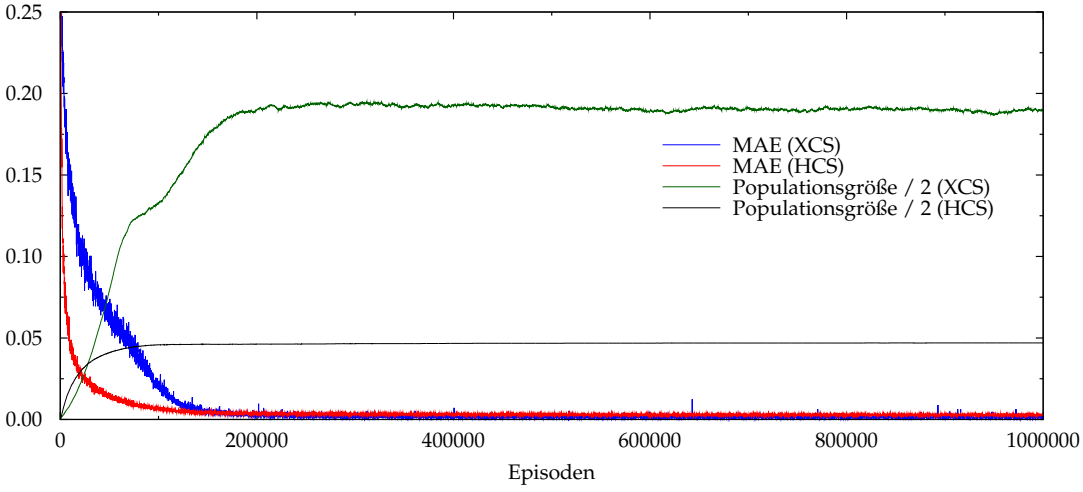


(c)

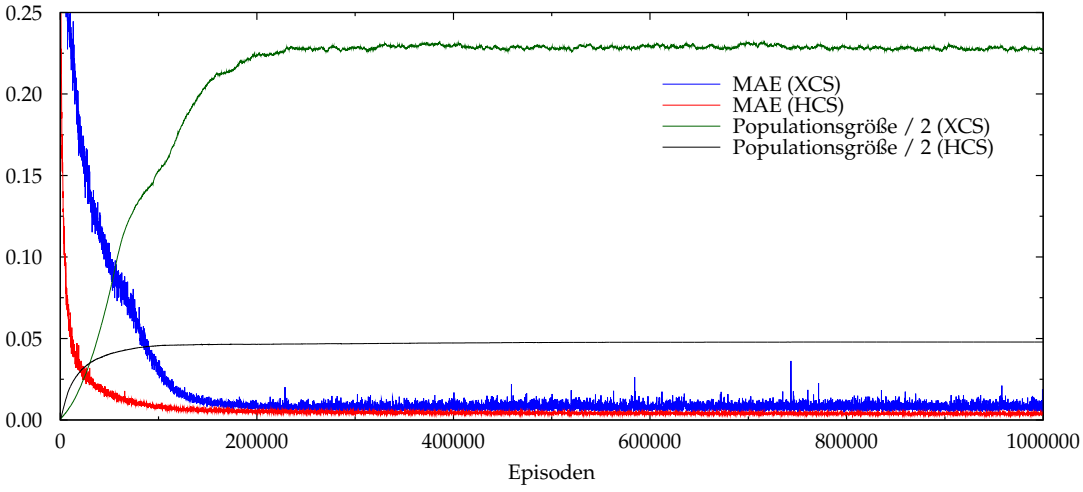
Abbildung 7.7: Zeitliche Entwicklung von MAE und Populationsgröße bei Einsatz des XCS und des HCS zur Approximation der Stufenfunktionen **(a)** f_1 und **(b)** f_2 sowie **(c)** der Dachfunktion f_{roof} .



(a)



(b)



(c)

Abbildung 7.8: Zeitliche Entwicklung von MAE und Populationsgröße bei Einsatz des XCS und des HCS zur Approximation **(a)** der Sinusfunktion f_3, **(b)** der Überlagerung f_{e3} dreier Exponentialfunktionen und **(c)** der rotierten Variante f_{e3r} von f_{e3} .

7.11 aufgetragen. Da die Standardabweichungen nicht wie in Abbildung 7.4 in Form von Fehlerbalken eingetragen werden konnten, wurden stattdessen die Graphen der approximierten Funktionen basierend auf den Standardabweichungen eingefärbt.

Die Approximation der Funktion f_1 stellt für das HCS offenbar ein deutlich schwierigeres Problem dar als für das XCS. Die Ursache hierfür ist die gleiche, die dem HCS die Approximation der Funktion f_{abs} an ihren Undifferenzierbarkeitsstellen erschwert. Eine exakte Vorhersage des Funktionswertes ist einem Klassifizierer – gleich welchen Systems – im Falle einer stückweise konstanten Funktion offenbar nur dann möglich, wenn diese Funktion eingeschränkt auf den durch den Matchbereich des Klassifizierers beschriebenen Teil des Zustandsraums konstant ist. Für eine gute Approximation einer Stufenfunktion müssen demnach Klassifizierer evolviert werden, deren Matchbereiche nicht über die „Stufenkanten" dieser Funktion reichen. Abbildung 7.9 zeigt entlang eines „Schnittes" durch den Zustandsraum bei $y = 0.5$ die Vorhersagen der jeweils passenden Klassifizierer von HCS und XCS bei Approximation der Funktion f_1. Gut zu erkennen ist, dass die Matchbereiche der weitaus meisten Klassifizierer die genannte Bedingung erfüllen. Aus den bereits bei der Diskussion der Approximation der Funktion f_{abs} genannten Gründen bringen XCS und HCS dennoch stets auch Klassifizierer hervor, die „auf einer Kante liegen" und daher nur ungenaue Vorhersagen machen können. Das XCS, bei dem stets mehrere Klassifizierer zu einem Zustand passen, kann dies kompensieren, das HCS hingegen nicht[7]. Dass sich dieses Problem bei der Approximation der Funktion f_1 stärker auswirkt als bei der Funktion f_{abs}, liegt wohl darin begründet, dass zum einen f_1 „mehr" problematische Stellen besitzt als f_{abs}, zum anderen aber natürlich auch darin, dass sich der Wert der Funktion f_1 an ihren Unstetigkeitsstellen stärker und abrupter verändert, als es an den Undifferenzierbarkeitsstellen der Funktion f_{abs} der Fall ist.

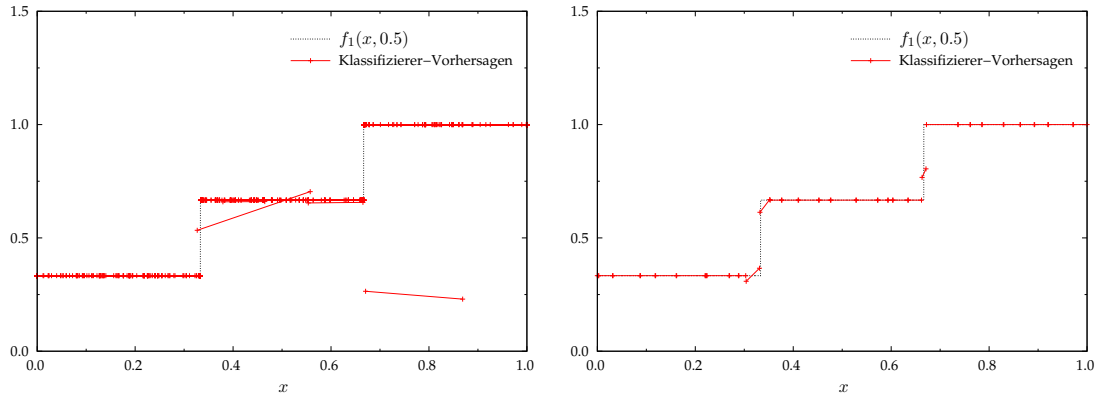


Abbildung 7.9: Vorhersagen der Klassifizierer einer **(a)** vom XCS und **(b)** vom HCS zur Approximation der Funktion f_1 evolvierten Population bei $y = 0.5$.

Eingedenk der geschilderten Probleme bei der Approximation stückweise konstanter Funktionen ist es zunächst überraschend, dass die Funktion f_2 vom HCS sogar etwas genauer approximiert wird als vom XCS. Ursächlich hierfür ist die zu Beginn des Kapitels 4 geschilderte Problematik intervallbasierter Klassifiziererbedingungen. Während das XCS die achsenparallel ausgerichteten Stufen der Funktion f_1 sehr gut abbilden kann,

[7] Dies erklärt, da analog auch im Falle der Verwendung konstanter Klassifizierervorhersagen argumentiert werden kann, auch den vergleichsweise hohen Systemfehler des HCS bei den in Kapitel 6 betrachteten Klassifikationsproblemen, deren Aktions-Wertefunktionen ja auch stückweise konstant waren.

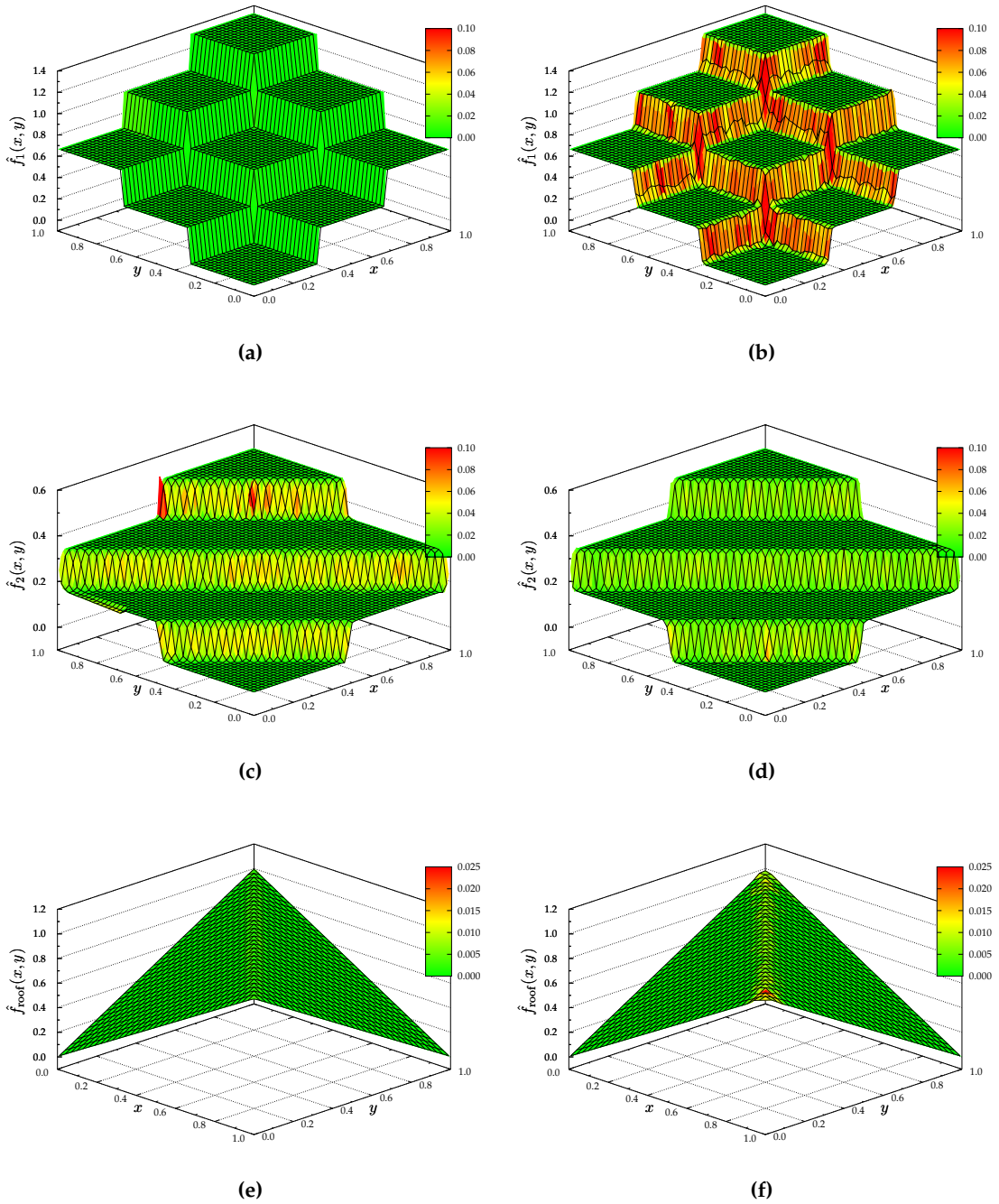


Abbildung 7.10: Durchschnittlicher approximierter Funktionswert bei XCS (jeweils links) und HCS (jeweils rechts) im Falle der Stufenfunktionen f_1 (**(a)** und **(b)**) und f_2 (**(c)** und **(d)**) sowie der Dachfunktion f_{roof} (**(e)** und **(f)**). Die Färbung gibt die Standardabweichung des approximierten Funktionswertes an.

gelingt ihm dies bei den nicht-achsenparallelen Stufen der Funktion f_2 nicht. Für das HCS hingegen stellen diese keine zusätzliche Schwierigkeit dar, die Funktion f_2 ist für das HCS nicht schwerer zu approximieren, als die Funktion f_1. Allerdings auch nicht leichter, wie der im Vergleich niedrigere MAE vermuten lassen könnte: Dieser ist nur darauf zurückzuführen, dass der Wertebereich der Funktion f_2 kleiner ist als der der Funktion f_1. Dementsprechend fallen auch die Abweichungen zwischen approximiertem

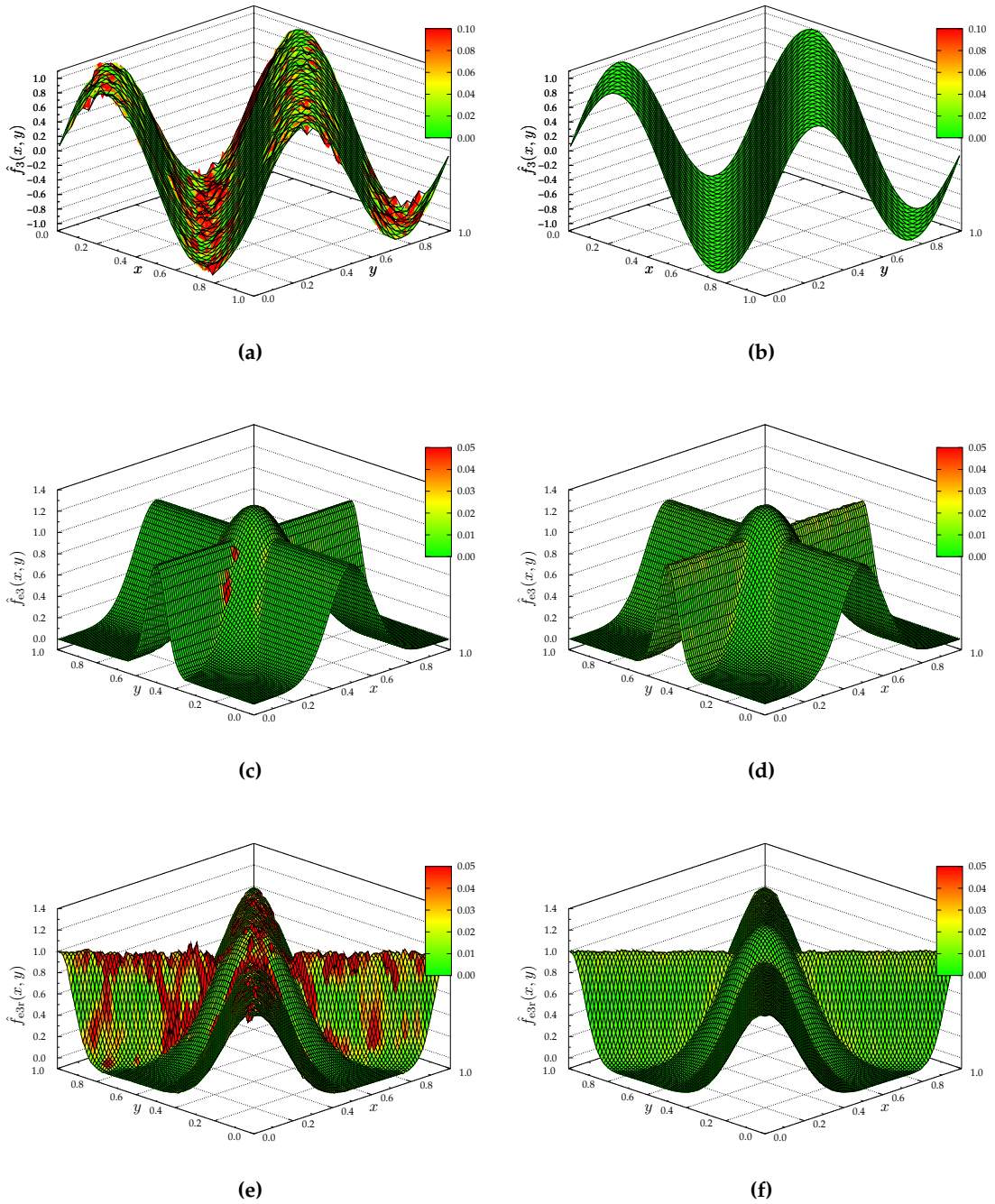


Abbildung 7.11: Durchschnittlicher approximierter Funktionswert bei XCS (jeweils links) und HCS (jeweils rechts) im Falle der Sinusfunktion f_3 (**(a)** und **(b)**), der Überlagerung f_{e3} von Exponentialfunktionen (**(c)** und **(d)**) sowie deren rotierter Variante f_{e3r} (**(e)** und **(f)**). Die Färbung gibt die Standardabweichung des approximierten Funktionswertes an.

und tatsächlichem Funktionswert und damit letztlich der MAE im Falle der Funktion f_2 kleiner aus als bei Approximation der Funktion f_1.
Die Approximation der Funktion f_{roof} stellt offenbar weder das XCS noch das HCS vor größere Probleme. Die dabei von beiden Systemen erreichten MAE-Werte unterschreitet in den hier durchgeführten Experimenten allein das XCS bei Approximation der Funktion f_1. Beide Systeme approximieren die Funktion f_{roof} annähernd gleich gut; lediglich

entlang des „Firsts“ der Dachfunktion ist die Approximation des XCS etwas[8] exakter als die des HCS. Für diese minimal bessere Approximation evolviert das XCS allerdings Populationen, die nahezu siebenmal so groß sind wie die des HCS.
Dies hat natürlich Auswirkungen auf die Dauer der Experimente. Obwohl in dieser Arbeit generell auf die Angabe von Laufzeiten verzichtet wird, da diese zum einen hardwareabhängig sind und zum anderen bei der Implementation der für die Experimente verwendeten Software kein Wert auf eine Laufzeit-Optimierung gelegt wurde, sollen an dieser Stelle einmalig – nur um eine Vorstellung zu vermitteln – konkrete Laufzeiten genannt werden: Das Experiment zur Approximation der Funktion f_{roof} mit dem XCS benötigte auf einem Computer mit Intel Core-i7-920-Prozessor und 6 GByte RAM etwa 17 Stunden und 20 Minuten, wobei bis zu 8 Einzelläufe parallel ausgeführt wurden, um alle Prozessorkerne des Rechners auszulasten. Das unter den gleichen Bedingungen durchgeführte Experiment mit dem HCS war bereits nach 45 Minuten beendet.
Bei der Approximation der Funktion f_3 zeigt sich ein ähnlich großer Unterschied in der Leistung der beiden Systeme wie bei der Funktion f_1 – allerdings mit umgekehrten Vorzeichen: Während das HCS die Sinusfunktion durchgängig sehr genau approximiert, gelingt dem XCS eine gute Approximation lediglich an den „Flanken“ der Funktion, im Bereich ihrer Optima weicht die Approximation des XCS stark von den wahren Werten der Funktion ab. Die schlechte Leistung des XCS erklärt sich wohl dadurch, dass bei der Funktion f_3 zwei Faktoren zusammen kommen, die dem XCS die Approximation einer Funktion erschweren: Zum einen weist die Funktion f_3 – ähnlich wie die Funktionen $f_{\mathrm{s}3}$ und $f_{\mathrm{s}4}$ aus Abschnitt 7.2 an vielen Stellen eine starke Krümmung auf, zum anderen ist sie achsendiagonal orientiert, sodass die Nachteile der Verwendung intervallbasierter Bedingungen zum Tragen kommen. Dem HCS bereiten beide Faktoren, wie schon bei den Funktionen $f_{\mathrm{s}3}$ und $f_{\mathrm{s}4}$ sowie f_2 gesehen, keine besonderen Schwierigkeiten.
Interessant ist in diesem Kontext ein Vergleich der Lernverläufe beider Systeme bei der Approximation der Funktion $f_{\mathrm{e}3}$ und ihrer gedrehten Variante $f_{\mathrm{e}3\mathrm{r}}$, da diese sich nur durch ihre Orientierung im Zustandsraum unterscheiden – $f_{\mathrm{e}3}$ ist achsenparallel, $f_{\mathrm{e}3\mathrm{r}}$ achsendiagonal ausgerichtet. MAE und Populationsgröße des HCS entwickeln sich bei beiden Funktionen nahezu gleich. Dass der MAE des HCS bei der Funktion $f_{\mathrm{e}3\mathrm{r}}$ etwas größer bleibt, ist darauf zurückzuführen, dass bei $f_{\mathrm{e}3\mathrm{r}}$ durch die Drehung die – leicht zu approximierenden – flachen Ausläufer der Funktion $f_{\mathrm{e}3}$ „wegfallen“ und dafür die – schwerer zu approximierenden – Grate länger werden. Wird dies berücksichtigt, indem bei der Berechnung des MAE nur die approximierten Funktionswerte an Stellen $s = (x, y)^T$ des Zustandsraums berücksichtigt werden, für die sowohl $(x, y)^T$ als auch der gemäß (7.11) berechnete Punkt $(x', y')^T$ im Einheitsquadrat liegen, ist der MAE beim HCS für beide Varianten der Funktion annähernd gleich groß. Das XCS hingegen approximiert die Funktion $f_{\mathrm{e}3\mathrm{r}}$ auch unter Berücksichtigung dieses Effektes deutlich schlechter als die Funktion $f_{\mathrm{e}3}$. Wie schon bei der Funktion f_3 sind hauptsächlich die Grate der Funktion $f_{\mathrm{e}3\mathrm{r}}$ für das XCS schwer zu approximieren, da sie mit intervallbasierten Bedingungen nur schlecht abgebildet werden können. Bei der Variante $f_{\mathrm{e}3}$, deren Grate achsenparallel verlaufen, tritt dieses Problem nicht auf und das XCS weist bei dieser Variante sogar einen etwas niedrigeren MAE auf als das HCS. Letzterem gelingt die Approximation des schmaleren Grats der Funktion bei $y = 0.5$ nicht so gut wie dem XCS. Dies ist, da sich an diesem schmalen Grat die Funktion auf engem Raum stark verändert, wohl auf die gleichen Effekte zurückzuführen, wie die Schwierigkeiten bei der Approximation der Funktion f_{abs} an ihren Undifferenzierbarkeitsstellen und der Stufenfunktionen f_1 und f_2 an ihren Unstetigkeitsstellen.

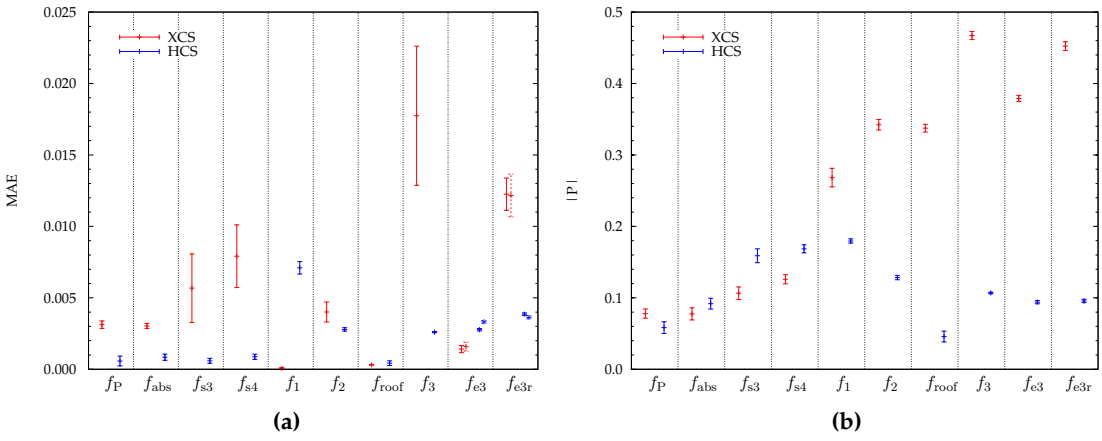
[8] Zu beachten ist die veränderte Farbskala bei den Abbildungen 7.10(e)und 7.10(f).

Wie bereits in Abschnitt 7.1 angekündigt, wurde am Ende jedes Einzellaufs der MAE auf einer größeren Stichprobe – nämlich den 100 respektive 100×100 Punkten, für die der Approximationsfehler ermittelt wurde – berechnet. Der über die Einzelläufe gebildete Durchschnitt $\overline{\text{MAE}}$ des mittleren absoluten Fehlers für die einzelnen Experimente ist mit Standardabweichung in Tabelle 7.1 aufgeführt, die ferner auch die mittleren Populationsgrößen am Ende der durchgeführten Experimente enthält. Für die Funktionen f_{e3} und f_{e3r} sind jeweils zwei Werte des $\overline{\text{MAE}}$ angegeben. Der erste ist der sich bei Berücksichtigung aller 100×100 Punkten ergebende Wert, der zweite ergibt sich, wenn nur Punkte innerhalb eines dem Einheitsquadrat eingeschriebenen Kreises berücksichtigt werden, also nur die einander entsprechenden Teile der beiden Funktionen.

	XCS		HCS	
f	$\overline{\text{MAE}} \pm \sigma$	$\lvert P\rvert \pm \sigma$	$\overline{\text{MAE}} \pm \sigma$	$\lvert P\rvert \pm \sigma$
f_{P}	0.003124 ± 0.000264	78.00 ± 6.45	0.000577 ± 0.000347	58.40 ± 8.17
f_{s3}	0.005677 ± 0.002398	106.50 ± 8.80	0.000594 ± 0.000172	158.90 ± 9.71
f_{s4}	0.007913 ± 0.002197	126.00 ± 6.41	0.000878 ± 0.000182	168.60 ± 5.67
f_{abs}	0.003030 ± 0.000182	77.50 ± 8.53	0.000846 ± 0.000217	91.90 ± 7.52
f_1	0.000062 ± 0.000061	1716.0 ± 83.78	0.007101 ± 0.000435	1149.20 ± 190.14
f_2	0.004010 ± 0.000702	2190.70 ± 47.03	0.002791 ± 0.000121	821.70 ± 19.01
f_{roof}	0.000315 ± 0.000019	2159.50 ± 34.50	0.000429 ± 0.000163	292.90 ± 48.73
f_3	0.017746 ± 0.004870	2989.40 ± 36.19	0.002598 ± 0.000046	683.70 ± 7.12
f_{e3}	0.001416 ± 0.000249 0.001584 ± 0.000312	2425.90 ± 28.15	0.002771 ± 0.000085 0.003320 ± 0.000097	601.60 ± 14.16
f_{e3r}	0.012251 ± 0.001128 0.012162 ± 0.001498	2895.40 ± 39.21	0.003860 ± 0.000087 0.003640 ± 0.000109	612.10 ± 14.96

Tabelle 7.1: Durchschnittlicher mittlerer absoluter Fehler und durchschnittliche Populationsgröße am Ende der mit XCS und HCS durchgeführten Experimente zur Funktionsapproximation.

Die in dieser Tabelle enthaltenen Werte sind zusätzlich in Abbildung 7.12 aufgetragen:



(a) **(b)**

Abbildung 7.12: (a) Durchschnittlicher mittlerer absoluter Fehler und **(b)** durchschnittliche Populationsgröße am Ende der mit XCS und HCS durchgeführten Experimente zur Funktionsapproximation.

Hier wird noch einmal auf einen Blick deutlich, dass das XCS nur bei zwei der zehn Testfunktionen die Approximationsleistung des HCS merklich übertreffen kann, nämlich bei den Funktionen f_1 und $f_{\mathrm{e}3}$. In Abbildung 7.12(b) ist ferner zu erkennen, dass das XCS insbesondere bei den zweidimensionalen Funktionen deutlich größere Populationen evolviert als das HCS[9].

Wie schon einmal erwähnt, wurde das XCS in der Vergangenheit in verschiedener Weise modifiziert, um seine Fähigkeiten bei der Funktionsapproximation zu verbessern, wobei auch einige der in diesem Kapitel verwendeten Funktionen als Testfunktionen verwendet wurden:

- In [Lanzi u. a. 2005a] wurden die Funktionen f_{P}, $f_{\mathrm{s}3}$, $f_{\mathrm{s}4}$ und f_{abs} benutzt, um eine XCS-Variante zu testen, deren Klassifizierervorhersagen Polynome in den Komponenten des Umgebungszustandes sind. Allerdings wurde dabei, wie es scheint, ein ganzzahlig kodiertes XCS verwendet, sodass etwa statt der Funktion $f_{\mathrm{s}3}$ die Funktion $F_{s3}(x) = 1000 \cdot f_{\mathrm{s}3}\left(\frac{x}{1000}\right)$ mit $x \in \{0, 1, 2, \ldots, 1000\}$ betrachtet wurde. Die dort angegebenen Ergebnisse sind daher, insbesondere aufgrund der Verwendung eines diskreten anstelle eines kontinuierlichen Zustandsraums, nicht mit den hier unter Einsatz des HCS erhaltenen vergleichbar. Ein naiver Vergleich ist dennoch insofern möglich, als die in dem Artikel angegebenen $\overline{\mathrm{MAE}}$-Werte unter Verwendung des in die Transformation der Funktionsvorschriften eingehenden Faktors 1000 umgerechnet werden können, wobei sich durchwegs größere Werte des $\overline{\mathrm{MAE}}$ ergeben, als hier mit dem HCS erzielt wurden.

- In [Butz 2005] und [Lanzi u. Wilson 2006] wurde das XCS mit kernelbasierten ellipsoidalen Bedingungen respektive mit auf konvexen Hüllen basierenden Bedingungen eingesetzt, um die Funktionen f_1, f_2 und f_3 zu approximieren. Die Ergebnisse dieser Experimente werden jedoch nur in Form von Darstellungen des Lernverlaufs – analog etwa den in Abbildung 7.7 gezeigten – dargestellt, aus denen die erreichten Werte des Approximationsfehlers und der Populationsgröße nicht genau abgelesen werden können. In beiden Fällen scheint aber die Funktion f_1 besser als durch das HCS approximiert zu werden, während sich der Fehler bei den Funktionen f_2 und f_3 in der Größenordnung des vom HCS erreichten bewegt, vermutlich aber etwas größer ist. Zweifelsfrei abgelesen werden kann allerdings, dass das XCS mit auf konvexen Hüllen basierenden Bedingungen deutlich größere Populationen evolviert als das HCS, auch das XCS mit ellipsoidalen Bedingungen bringt, außer im Fall der Funktion f_2, größere Populationen hervor als das HCS.

Wünschenswert wäre es natürlich gewesen, einen genaueren Vergleich der von den genannten XCS-Varianten erzielten Ergebnisse mit denen des HCS durchzuführen. Aufgrund der genannten Probleme wäre es dazu jedoch nötig gewesen, diese XCS-Varianten nachzuimplementieren und die entsprechenden Experimente mit diesen selbst durchzuführen. Hierauf wurde verzichtet, da es einen unverhältnismäßig hohen Aufwand bedeutet hätte.

[9] Zu diesem Punkt ist anzumerken, dass bei in [Butz 2005] durchgeführten Experimenten zur Approximation der Funktionen f_1 und f_2 deutlich kleinere Populationen evolviert wurden, wobei der erreichte Approximationsfehler allerdings auch höher war. Ursächlich hierfür dürfte zum einen sein, dass der hier verwendete XCS-Parametersatz gegenüber dem in [Butz 2005] benutzten leicht verändert wurde, zum anderen, dass die Anpassung der Gewichte für die Vorhersagenberechnung dort unter Verwendung einer Widrow-Hoff-Delta-Regel erfolgte, während hier das auf der Methode der kleinsten Quadrate basierende Verfahren zum Einsatz kam.

7.4 Zusammenfassung

In diesem Kapitel wurde das HCS zur Approximation einer ganzen Reihe von ein- und zweidimensionalen Funktionen eingesetzt. Dabei konnte es die Approximationsleistung des zum Vergleich herangezogenen XCS in fast allen Fällen übertreffen oder zumindest erreichen.
Lediglich bei einer der betrachteten Funktionen – einer achsenparallelen Stufenfunktion – blieb die Qualität der vom HCS entwickelten Approximation deutlich hinter der der XCS-Approximation zurück. Dies konnte darauf zurückgeführt werden, dass ein Klassifizierer mit einer in den Komponenten des Lernumgebungszustandes linearen Vorhersage, dessen Matchbereich eine Unstetigkeitsstelle der zu approximierenden Funktion enthält, deren Wert in seinem Matchbereich nur unzutreffend vorhersagen kann. Zwar gilt dies ebenso für Klassifizierer des XCS wie für solche des HCS, doch kann das XCS, da seine Population in der Regel mehrere zu einem Zustand passende Klassifizierer enthält, dies kompensieren, wozu das HCS, dessen Systemvorhersage für einen Zustand stets basierend auf der Vorhersage des einen zu diesem Zustand passenden Klassifizierers berechnet wird, nicht in der Lage ist. Ferner konnte festgestellt werden, dass aus dem selben Grund – also des Vorhandenseins stets nur eines passenden Klassifizierers – neben Unstetigkeitsstellen auch andere Stellen, an denen die Zielfunktion sich auf engem Raum stark verändert, für das HCS tendenziell schwerer zu approximieren sind als für das XCS. Beispiele hierfür sind Undifferenzierbarkeitsstellen, aber auch schmale (lokale) Optima, selbst wenn die Funktion in ihnen differenzierbar ist.
Die Entwicklung des HCS wurde dadurch motiviert, dass das XCS mit intervallbasierten Bedingungen nur über hyperquaderartige Bereiche des Zustandsraums generalisieren kann und daher seine Leistung stark von der Struktur der jeweils betrachteten Lernumgebung abhängt. Dieses Problem wurde auch in diesem Kapitel offenbar, insbesondere bei der Betrachtung zweier Varianten einer Funktion, von denen die eine achsenparallel, die andere achsendiagonal orientiert war. Während das XCS die zweite Variante bedeutend schlechter als die erste approximierte, gelang dem HCS in beiden Fällen eine nahezu gleich gute Approximation. Auch bei der Betrachtung je einer achsenparallelen und einer achsendiagonalen Stufenfunktion war zu beobachten, dass das HCS durch die Ausrichtung der Funktion wesentlich weniger beeinflusst wurde als das XCS. Als weiteres Ergebnis dieses Kapitels kann also festgehalten werden, dass es gelungen ist, mit dem HCS ein Lernendes Klassifizierendes System zu entwickeln, dessen Leistung weniger von der Struktur der verwendeten Lernumgebung abhängt, als es beim XCS mit intervallbasierten Bedingungen der Fall ist.
Zuletzt ist noch zu erwähnen, dass, wie schon bei den in Kapitel 6 betrachteten Klassifikationsproblemen, die Größe der vom HCS evolvierten Populationen auch in diesem Kapitel fast immer deutlich unter der der vom XCS hervorgebrachten lag.

Kapitel 8

Aktionenlernen mit dem HCS

Die Reihe von Experimenten mit dem HCS abschließend, wurde dieses auch zum Lernen von Aktionen eingesetzt. Die dabei erzielten Ergebnisse werden in diesem Kapitel vorgestellt und mit Resultaten aus mit dem XCS durchgeführten Experimenten verglichen.
Beim Aktionenlernen soll der Lernende Agent – hier also das HCS – in Interaktion mit einer Lernumgebung ein in dieser optimales Verhalten entwickeln. Dass ‚in Interaktion mit der Lernumgebung' gelernt werden soll, impliziert, dass – anders als bei Klassifikation und Funktionsapproximation – die Zustände mit denen das Lernende System konfrontiert wird, sowohl voneinander als auch von den vom System ausgeführten Aktionen abhängen. Lernumgebungen, in denen dies der Fall ist, werden, da in ihnen Lernepisoden in der Regel mehr als einen Lernschritt umfassen, auch als *Mehrschritt-Lernumgebungen* bezeichnet. Als optimal wird in solchen Lernumgebungen ein Verhalten angesehen, dass – auf lange Sicht betrachtet – den Return des Agenten maximiert. Um ein optimales Verhalten zu lernen, muss der Agent demzufolge die Werte der verschiedenen Zustände beziehungsweise Zustands-Aktions-Paare lernen; Aktionenlernen kann also auch als Approximation der Aktions-Wertefunktion der betrachteten Lernumgebung aufgefasst werden. Anders als bei den in Kapitel 7 betrachteten reinen Funktionsapproximationsproblemen werden dem System jedoch die korrekten Werte der zu approximierenden Funktion nicht mitgeteilt. Es muss sie vielmehr basierend auf den nach Ausführung von Aktionen folgenden Belohnungen sowie den geschätzten Werten der Zustände, in die diese Aktionen es führen, abschätzen.
Die drei in diesem Kapitel betrachteten Lernumgebungen wurden bereits in [Lanzi u. a. 2005c] verwendet, um die Eignung des reellwertig kodierten XCS zum Lernen in Mehrschritt-Lernumgebungen zu untersuchen. Zwei von ihnen entstammen ursprünglich der Reinforcement-Learning-Literatur ([Boyan u. Moore 1995]).
In Abschnitt 8.1 werden zunächst einige Anmerkungen zu Durchführung und Bewertung von Experimenten in Mehrschritt-Lernumgebungen gemacht. Die drei darauf folgenden Abschnitte widmen sich jeweils einer der hier betrachteten Lernumgebungen: Abschnitt 8.2 bespricht die in der Korridor-Lernumgebung durchgeführten Experimente, Abschnitt 8.3 geht auf die Empty-Room-Lernumgebung ein und Abschnitt 8.4 befasst sich mit der Puddles-Lernumgebung. Das Kapitel abschließend, werden die in diesem gewonnenen Erkenntnisse in Abschnitt 8.5 zusammengefasst.

8.1 Experimentelles Vorgehen und Beurteilungskriterien

Die Durchführung der Experimente in den hier betrachteten Mehrschritt-Lernumgebungen erfolgt in der bereits bekannten Weise: In jedem Experiment werden 10 Einzelläufe durchgeführt, über die dann die zur Beurteilung der Leistung des untersuchten Systems herangezogenen Größen gemittelt werden.
In Abschnitt 6.1 wurde dargelegt, dass beim Vergleich zweier Systeme diesen in den durchgeführten Läufen die gleichen Zustände präsentiert werden sollten. Im Falle von Mehrschritt-Lernumgebungen betrifft dies natürlich nur die Startzustände von Lernepisoden, da die in einer Episode auftretenden Folgezustände von den Aktionen des Lernenden Systems bestimmt werden.

Bei der Durchführung von Experimenten in Mehrschritt-Lernumgebungen stellt sich ein Problem, dass in Lernumgebungen, deren Episoden stets nur einen Lernschritt beinhalten, nicht auftreten kann: Da eine Lernepisode erst dann endet, wenn der Lernende Agent einen Terminalzustand erreicht, kann eine Episode sehr lange dauern oder überhaupt nicht von selbst enden – beispielsweise, wenn das gelernte Verhalten den Agenten zwischen zwei Zuständen hin und her wechseln lässt. Damit durch derartige Effekte der Lernvorgang nicht übermäßig in die Länge gezogen wird, ist es üblich, Lernepisoden nach einer zuvor festgelegten Zahl von Lernschritten abzubrechen. Bei den hier beschriebenen Experimenten geschieht dies, wenn nach 100 Lernschritten kein Terminalzustand erreicht ist.

Für die mit dem XCS durchgeführten Experimente zum Aktionenlernen werden die in [Lanzi u. a. 2005c] angegebenen Parameter übernommen. Wie dort wird das XCS mit Ordered-Bound-Intervallrepräsentation und berechneten Klassifizierervorhersagen verwendet. Abweichend vom dortigen Vorgehen werden jedoch die Gewichte für die Berechnung der Klassifizierervorhersagen hier stets unter Verwendung der Methode der kleinsten Quadrate angepasst. Die Ergebnisse der hier mit dem XCS durchgeführten Experimente weichen in zwei Punkten von den in [Lanzi u. a. 2005c] veröffentlichten ab: Zum einen evolviert das XCS in den hier durchgeführten Experimenten deutlich größere Populationen, als dort angegeben ist[1], zum anderen gelingt es ihm in der Empty-Room-Lernumgebung nicht in allen Läufen, ein gutes Verhalten zu lernen, wozu es in [Lanzi u. a. 2005c] offenbar in der Lage war.
Für das HCS wurden in vorbereitenden Tests zwei Parametersätze für die Verwendung in den Experimenten zum Aktionenlernen bestimmt, einer für die eindimensionale Korridor-Lernumgebung, der andere für die beiden Lernumgebungen mit zweidimensionalem Zustandsraum.

In den in diesem Kapitel betrachteten Lernumgebungen werden Belohnungen derart vergeben, dass der Agent nur bei Erreichen eines Zielzustandes eine Belohnung der Höhe 0 erhält, in allen anderen Fällen hat die Belohnung einen negativen Wert. Werden diese negativen Belohnungen als Kosten der Ausführung einer Aktion durch den Agenten interpretiert, so entspricht das Ziel der Maximierung des Returns einer Minimierung der Kosten zum Erreichen des Ziels. Für den Fall, dass die Kosten jedes Schrittes gleich hoch

[1] Ein Ergebnis dieses Kapitels wird sein, dass das HCS in den betrachteten Lernumgebungen kleinere Populationen evolviert als das XCS. Dieses Ergebnis hat Bestand, auch wenn die in [Lanzi u. a. 2005c] angegebenen Populationsgrößen des XCS zugrunde gelegt werden, wenngleich dann die Unterschiede weniger groß sind.

sind, die negativen Belohnungen also stets den gleichen Wert haben, folgt, dass das Verhalten des Agenten optimal ist, wenn er den Zielzustand stets möglichst schnell erreicht. Die Leistung eines Lernenden Systems beim Aktionenlernen wird hier, wie in [Lanzi u. a. 2005c], an der durchschnittlichen Zahl von Lernschritten, die es zum Erreichen des Zielzustandes benötigt, gemessen. Der Berechnung dieses Durchschnittes werden die jeweils letzten 50 im Exploitationsmodus durchlaufenen Lernepisoden zugrunde gelegt.
Als zweites Beurteilungskriterium dient wie in den vorangegangenen Kapiteln die durchschnittliche Größe der evolvierten Populationen.

8.2 Die Korridor-Lernumgebung

Bei der ersten in diesem Kapitel betrachteten Lernumgebung handelt es sich um eine recht einfach strukturierte eindimensionale Lernumgebung, die als Korridor interpretiert werden kann, in dem ein Reinforcement-Learning-Agent lernen soll, sich möglichst schnell zum Ausgang am rechten Ende zu begeben.

8.2.1 Problembeschreibung

Der Zustand der Korridor-Lernumgebung [Lanzi u. a. 2005c] wird beschrieben durch eine reelle Zahl $s \in [0, 1) = \mathcal{S}$, die die Position eines Reinforcement-Learning-Agenten in einem Korridor beschreibt. Das linke Ende des Korridors entspricht dem Zustand $s = 0$, das rechte Ende, das zugleich das Ziel darstellt, zu dem sich der Agent bewegen soll, liegt dementsprechend bei $s = 1$. Dem Agenten stehen zwei Aktionen zur Verfügung. Die Aktion *‚rechts‘* führt zu einer Veränderung der Position des Agenten um $+\Delta s$, durch die Aktion *‚links‘* wird die Position dementsprechend um $-\Delta s$ verändert. An den Enden des Korridors läuft der Agent dabei sozusagen „gegen die Wand“, die kleinste mögliche Position ist $s = 0$, die größte $s = 1$. Die Position s_{t+1} des Agenten zum Zeitpunkt $t + 1$ ergibt sich also wie folgt aus seiner Position s_t zum Zeitpunkt t und der ausgeführten Aktion a_t:

$$s_{t+1} = \begin{cases} \min(s_t + \Delta s, 1) & \text{, falls } a_t = \textit{‚rechts‘} \\ \max(s_t - \Delta s, 0) & \text{, falls } a_t = \textit{‚links‘} \end{cases} \tag{8.1}$$

Am Anfang einer Lernepisode wird der Agent zufällig irgendwo im Korridor „abgesetzt“, lediglich die Zielposition $s = 1$ ist ausgeschlossen. Die Lernepisode endet, wenn der Agent das rechte Ende des Korridors bei $s = 1$ erreicht. In diesem Fall erhält er eine Belohnung der Höhe 0, in allen anderen Fällen hat die Belohnung – unabhängig von der ausgeführten Aktion – den Wert -0.5.
Die ideale Politik ist in dieser Lernumgebung offensichtlich; sie besteht einfach darin, sich immer nur nach rechts zu bewegen, also stets die Aktion *‚rechts‘* zu wählen. Somit ergibt sich die Anzahl von Schritten, die ein sich optimal verhaltender Agent, der in einem Zustand $s \in [0, 1)$ startet, zum Erreichen der Zielposition benötigt, als:

$$steps_{min}(s) = \left\lceil \frac{1 - s}{\Delta s} \right\rceil \tag{8.2}$$

Da dies bekannt ist, kann auch die optimale Aktions-Wertefunktion Q^* für die Korridor-Lernumgebung explizit angegeben werden. Der optimale Aktionswert für ein Zustands-Aktions-Paar (s, a) ergibt sich als Summe der diskontierten Belohnungen, die das System

erhält, wenn es im Zustand s die Aktion a ausführt und danach der optimalen Politik folgt, also nur noch die Aktion *‚rechts'* ausführt. Wird bereits im Startzustand s die Aktion *‚rechts'* ausgeführt, so ist die Anzahl von Schritten bis zum Erreichen des Endzustandes gerade durch (8.2) gegeben, sodass gilt:

$$Q^*(s, \textit{‚rechts'}) = -0.5 \cdot \sum_{k=0}^{steps_{min}(s)-2} \gamma^k \tag{8.3}$$

Die Summe läuft hier nur bis $steps_{min}(s) - 2$, da nach dem letzten Schritt – also beim Erreichen des Ziels – eine Belohnung der Höhe 0 gewährt wird.
Wird zunächst einmalig die Aktion *‚links'* ausgeführt und erst dann der optimalen Politik gefolgt, so werden zum Erreichen des Ziels in der Regel zwei Schritte mehr benötigt: der Schritt nach links und ein weiterer Schritt, um auf die Startposition zurückzukehren. Ist allerdings die Entfernung des Agenten von der linken Seite des Korridors kleiner als die Schrittweite Δs, so läuft der Agent beim Schritt nach links „gegen die Wand" und der zusätzliche Schritt, der ihn auf seine Startposition zurückführt, entfällt:

$$Q^*(s, \textit{‚links'}) = \begin{cases} -0.5 \cdot \sum_{k=0}^{steps_{min}(s)-1} \gamma^k & \text{, falls } s < \Delta s \\ -0.5 \cdot \sum_{k=0}^{steps_{min}(s)} \gamma^k & \text{, sonst} \end{cases} \tag{8.4}$$

In Abbildung 8.1 ist die durch (8.3) und (8.4) gegebene Aktions-Wertefunktion der Korridor-Lernumgebung für verschiedene Werte der Schrittweite Δs und des Diskontierungsfaktors γ aufgetragen.

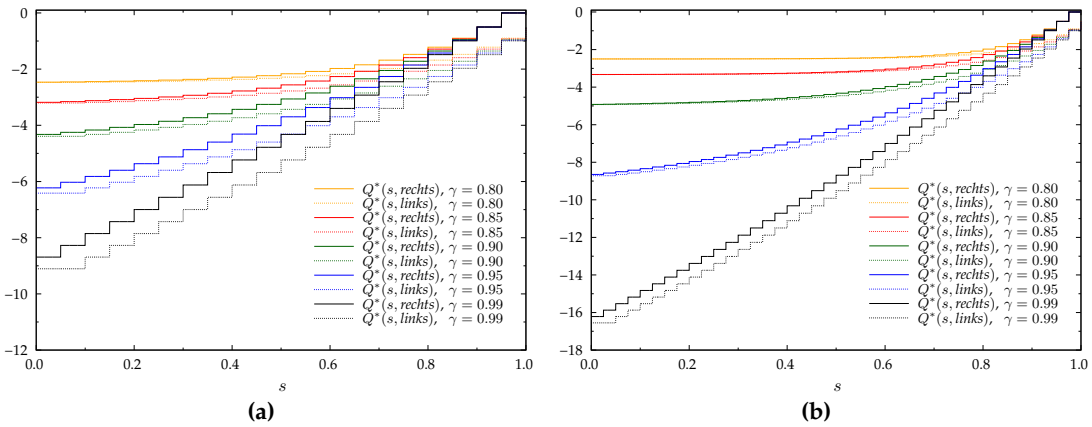


Abbildung 8.1: Aktions-Wertefunktion der Korridor-Lernumgebung für verschiedene Werte des Diskontierungsfaktors γ und die Schrittweiten **(a)** $\Delta s = 0.05$ sowie **(b)** $\Delta s = 0.025$

Deutlich zu sehen ist hier, dass der Unterschied zwischen den Werten der beiden möglichen Aktionen für eine Zustand umso größer ist, je kleiner zum einen die (in Schritten gemessene) Entfernung des Zustandes von der Zielposition bei $s = 1$ ist und je größer zum anderen der Diskontierungsfaktor γ gewählt wird. Je kleiner also die Schrittweite und der Diskontierungsfaktor, umso genauer muss ein Lernender Agent die Aktionswerte abschätzen, um stets die richtige Aktion wählen zu können. In diesem Sinne beeinflussen beide Parameter die Schwierigkeit der Korridor-Lernumgebung.

8.2.2 Experimente

Mit dem HCS ebenso wie mit dem XCS wurden jeweils mehrere Experimente in der Korridor-Lernumgebung durchgeführt, wobei sowohl die Schrittweite Δs als auch der Diskontierungsfaktor γ variiert wurden. Die Lernverläufe der beiden Systeme in diesen Experimenten sind in der Abbildung 8.2 dargestellt: In 8.2(a) und 8.2(c) ist die durchschnittliche Anzahl von Schritten, die das XCS respektive das HCS zum Erreichen der Zielposition benötigen, über der Anzahl absolvierter Episoden aufgetragen. Zudem ist auch die durchschnittliche Anzahl von Schritten eingetragen, die ein sich optimal verhaltender Agent benötigt, wenn ihm die gleichen Episodenstartzustände wie den Lernenden Klassifizierenden Systemen präsentiert werden. Die Abbildungen 8.2(b) und 8.2(d) zeigen die Entwicklung der durchschnittlichen Populationsgröße beider Systeme in den durchgeführten Experimenten.
Die Ergebnisse der in der Korridor-Lernumgebung durchgeführten Experimente können wie folgt zusammengefasst werden:

- Im Fall der größten Schrittweite ($\Delta s = 0.05$) entspricht das Verhalten des HCS bei jedem der drei betrachteten Werte des Diskontierungsfaktors bereits nach etwa 1000 Episoden nahezu durchgängig bis zum Ende des Experimentes dem Verhalten des optimalen Agenten[2]. Auch das XCS lernt bei dieser Schrittweite und bei allen Werten des Diskontierungsfaktors schnell ein Verhalten, das dem optimalen nahe kommt, Abweichungen treten jedoch weitaus häufiger auf und sind größer, als es beim HCS der Fall ist.

- Bei der Schrittweite $\Delta s = 0.025$ ist das Verhalten des HCS für $\gamma = 0.95$ und $\gamma = 0.99$ nach etwa 2000 Episoden nahezu optimal. Im Falle $\gamma = 0.90$ gelingt es dem HCS nicht, ein optimales Verhalten zu lernen. Die Ergebnisse des XCS entsprechen denen im Fall der größeren Schrittweite: Für alle Werte des Diskontierungsfaktors kommt das gelernte Verhalten dem optimalen nahe, ohne es jedoch zu erreichen.

- Bei der kleinsten betrachteten Schrittweite ($\Delta s = 0.01$) lernt das HCS immerhin noch für $\gamma = 0.99$, sich optimal zu verhalten, wenn auch erst nach etwa 12000 Episoden. Im Fall $\gamma = 0.95$ benötigt es deutlich mehr Schritte als der optimale Agent. Der dem Optimum annähernd parallele Verlauf der Kurve deutet allerdings darauf hin, dass zumindest in einigen der Einzelläufe des Experimentes ein optimales Verhalten gelernt wurde. Dies bestätigt sich bei Untersuchung der (hier nicht abgebildeten) Lernverläufe der Einzelläufe: in acht von ihnen entspricht das Verhalten des HCS nach etwa 10000-12000 Episoden dem Optimum. Dem XCS gelingt es bei dieser kleinen Schrittweite bei keinem Wert des Diskontierungsfaktors ein auch nur annähernd optimales Verhalten zu lernen.

- Wie schon bei der Klassifikation und der Funktionsapproximation evolviert das HCS auch hier deutlich kleinere Populationen als das XCS. Eine Ausnahme bilden die vom XCS mit den Diskontierungsfaktoren $\gamma = 0.95$ und $\gamma = 0.99$ für die Korridor-Lernumgebung mit Schrittweite $\Delta s = 0.01$ hervorgebrachten Populationen. In den entsprechenden Experimenten, in denen es dem XCS nicht gelingt, ein angemessenes Verhalten zu lernen, enthalten die Populationen am Ende nur noch einige übergenerelle Klassifizierer, deren Matchbereich nahezu der gesamte Zustandsraum ist. Wieso sich diese Populationen in den genannten Fällen so bilden,

[2] Aus der durchschnittlichen Anzahl benötigter Schritte zum Ziel kann in dieser einfachen Lernumgebung unmittelbar auf das zugrunde liegende Verhalten geschlossen werden: Jedes von der oben angegebenen optimalen Politik abweichende Verhalten hat eine höhere Anzahl benötigter Schritte zur Folge.

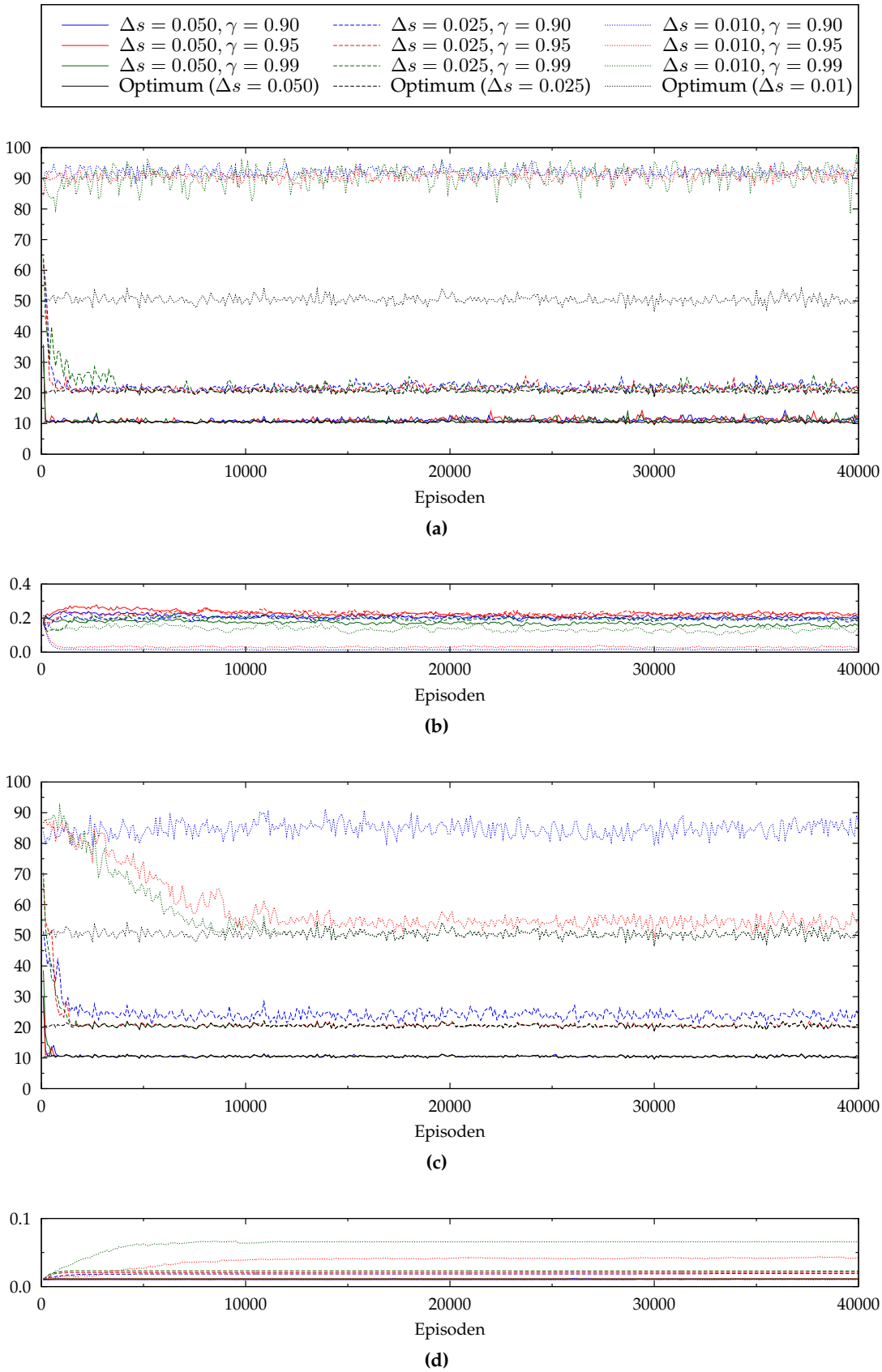


Abbildung 8.2: Lernverlauf von XCS und HCS in der Korridor-Lernumgebung für verschiedene Werte der Schrittweite Δs und des Diskontierungsfaktors γ: Durchschnittliche Anzahl von Schritten bis zum Ziel bei **(a)** XCS und **(c)** HCS sowie Populationsgröße von **(b)** XCS und **(d)** HCS.

wurde hier nicht untersucht, da das XCS nicht primärer Gegenstand der vorliegenden Arbeit ist. In [Lanzi u. a. 2005c] ist dieses Verhalten des XCS nicht beschrieben, dort wurde aber auch keine kleinere Schrittweite als $\Delta s = 0.025$ betrachtet.

Insgesamt ist festzustellen, dass das HCS in der Korridor-Lernumgebung erfolgreicher lernt als das XCS, insbesondere im Fall der kleinsten hier betrachteten Schrittweite bleibt die Leistung des XCS deutlich hinter der des HCS zurück, dem es zumindest für einen Wert des Diskontierungsfaktors gelingt, ein optimales Verhalten zu lernen. Zwar bringt auch das HCS nicht für alle Werte der Schrittweite und des Diskontierungsfaktors ein optimales Verhalten hervor , wenn es ihm gelingt, zeigt seine Leistung jedoch deutlich weniger Schwankungen und Abweichungen vom Optimum als es beim XCS der Fall ist.

8.3 Die Empty-Room-Lernumgebung

Die in [Boyan u. Moore 1995] eingeführte Empty-Room-Lernumgebung kann als eine zweidimensionale Variante der Korridor-Lernumgebung aufgefasst werden: Ein Reinforcement-Learning-Agent soll lernen, möglichst schnell einen leeren Raum zu verlassen, dessen Ausgang sich in der nordöstlichen Ecke befindet.

8.3.1 Problembeschreibung

Die Empty-Room-Lernumgebung ist analog der Korridor-Lernumgebung aufgebaut: Ein Zustand $s = (x, y)^T \in [0, 1)^2$ beschreibt die Position eines Reinforcement-Learning-Agenten in einem leeren Raum. Dem Agenten stehen vier Aktionen zur Auswahl, die Bewegungen in die vier Himmelsrichtungen entsprechen: $\mathcal{A} = \{$‚*N*‘, ‚*O*‘, ‚*S*‘, ‚*W*‘$\}$. Der Zustand, in den der Agent bei Ausführung der verschiedenen Aktionen in einem Zustand s_t gelangt, kann wie folgt berechnet werden:

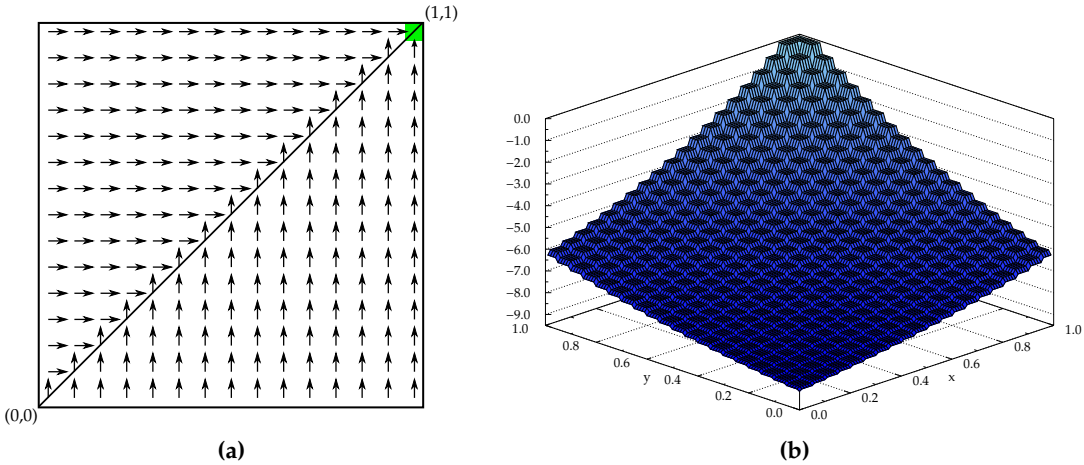
$$s_{t+1} = (x_{t+1}, y_{t+1})^T = \begin{cases} (x_t, \min(y_t + \Delta s, 1))^T & \text{, falls } a_t = \text{‚}N\text{‘} \\ (\max(x_t + \Delta s, 1), y_t)^T & \text{, falls } a_t = \text{‚}O\text{‘} \\ (x_t, \max(y_t - \Delta s, 1))^T & \text{, falls } a_t = \text{‚}S\text{‘} \\ (\min(x_t - \Delta s, 1), y_t)^T & \text{, falls } a_t = \text{‚}W\text{‘} \end{cases} \tag{8.5}$$

Am Anfang jeder Lernepisode wird der Agent wiederum irgendwo „abgesetzt“ und muss sich dann zum Ausgang bewegen; dieser ist erreicht, wenn beide Komponenten des Zustandes größer oder gleich 0.95 sind. Das Belohnungsschema der Empty-Room-Lernumgebung entspricht der der Korridor-Lernumgebung: Bei Erreichen des Zieles hat die Belohnung, die der Agent erhält, den Wert 0, andernfalls stets den Wert -0.5.
Es ist leicht zu sehen, dass, von einem Zustand $s = (x, y)^T$ aus startend, die minimal benötigte Zahl von Schritten bis zum Ziel durch

$$steps_{min}(s) = \max\left(\left\lceil \frac{\max(0, 0.95 - x)}{\Delta s} \right\rceil + \left\lceil \frac{\max(0, 0.95 - y)}{\Delta s} \right\rceil, 1\right) \tag{8.6}$$

gegeben ist[3]. Analog zu (8.4) und (8.3) kann damit auch die Aktions-Wertefunktion der Empty-Room-Lernumgebung angegeben werden, worauf hier jedoch verzichtet wird.

[3] Die Bildung des äußeren Maximums in (8.6) ist darauf zurückzuführen, dass eine Episode im Zielbereich ($x \geq 0.95, y \geq 0.95$) beginnen kann, jedoch erst überprüft wird, ob der Agent sich dort befindet, nachdem er eine Aktion – also einen Schritt – ausgeführt hat.



(a) **(b)**

Abbildung 8.3: Empty-Room-Lernumgebung: **(a)** Optimale Politik und **(b)** Wertefunktion V^* der optimalen Politik für Schrittweite $\Delta s = 0.05$ und Diskontierungsfaktor $\gamma = 0.95$.

Anschaulicher als die Formeldarstellung ist in jedem Fall die in Abbildung 8.3(b) aufgetragene Zustands-Wertefunktion V^*, die jedem Zustand das Maximum der zugehörigen Aktionswerte zuweist. Daneben visualisiert die Abbildung 8.3(a) eine[4] optimale Politik für die Empty-Room-Lernumgebung. Diese besteht darin, im südöstlichen Teil des Raumes stets nach Norden, im nordwestlichen Teil immer nach Osten zu gehen.

8.3.2 Experimente

Analog dem Vorgehen im Fall der Korridor-Lernumgebung wurden auch in der Empty-Room-Lernumgebung Experimente mit verschiedenen Setzungen der Schrittweite Δs und des Diskontierungsfaktors γ durchgeführt, wobei jedoch nur zwei Werte der Schrittweite Δs betrachtet wurden, da die Experimente mit dem XCS Laufzeiten von bis zu 3 Tagen hatten, also ausgesprochen langwierig waren. Aus dem gleichem Grund und weil das XCS für $\Delta s = 0.05$ bereits bei Verwendung des Diskontierungsfaktors $\gamma = 0.95$ ein schlechtes Ergebnis erzielte, wurde im Fall des XCS auch auf die Durchführung eines Experiments mit $\Delta s = 0.05$ und $\gamma = 0.90$ verzichtet. Abbildung 8.4 zeigt die Lernverläufe in den durchgeführten Experimenten.
Deren Ergebnisse entsprechen im wesentlichen den bei der Korridor-Lernumgebung erhaltenen:

- Im Fall der größten betrachteten Schrittweite ($\Delta s = 0.1$) gelingt es unabhängig vom verwendeten Diskontierungsfaktor sowohl dem HCS als auch dem XCS, ein (nahezu) optimales Verhalten zu entwickeln. Dies erreicht das XCS teilweise sogar schneller als das HCS, allerdings zeigen sich beim XCS über die gesamte Dauer des Experimentes immer wieder auch „Zacken“ in der Kurve des Lernverlaufs, die Abweichungen vom optimalen Verhalten anzeigen, diese treten bei allen betrachteten Diskontierungsfaktoren auf. Beim HCS hingegen zeigen sich nur bei Verwendung des Diskontierungsfaktors $\gamma = 0.90$ kleinere Abweichungen vom optimalen Kur-

[4] Die Empty-Room-Lernumgebung besitzt offensichtlich keine eindeutige optimale Politik: Außer entlang der nördlichen und östlichen Wand des Raumes sind die Aktionen ‚*O*‘ und ‚*N*‘ stets gleichermaßen optimal.

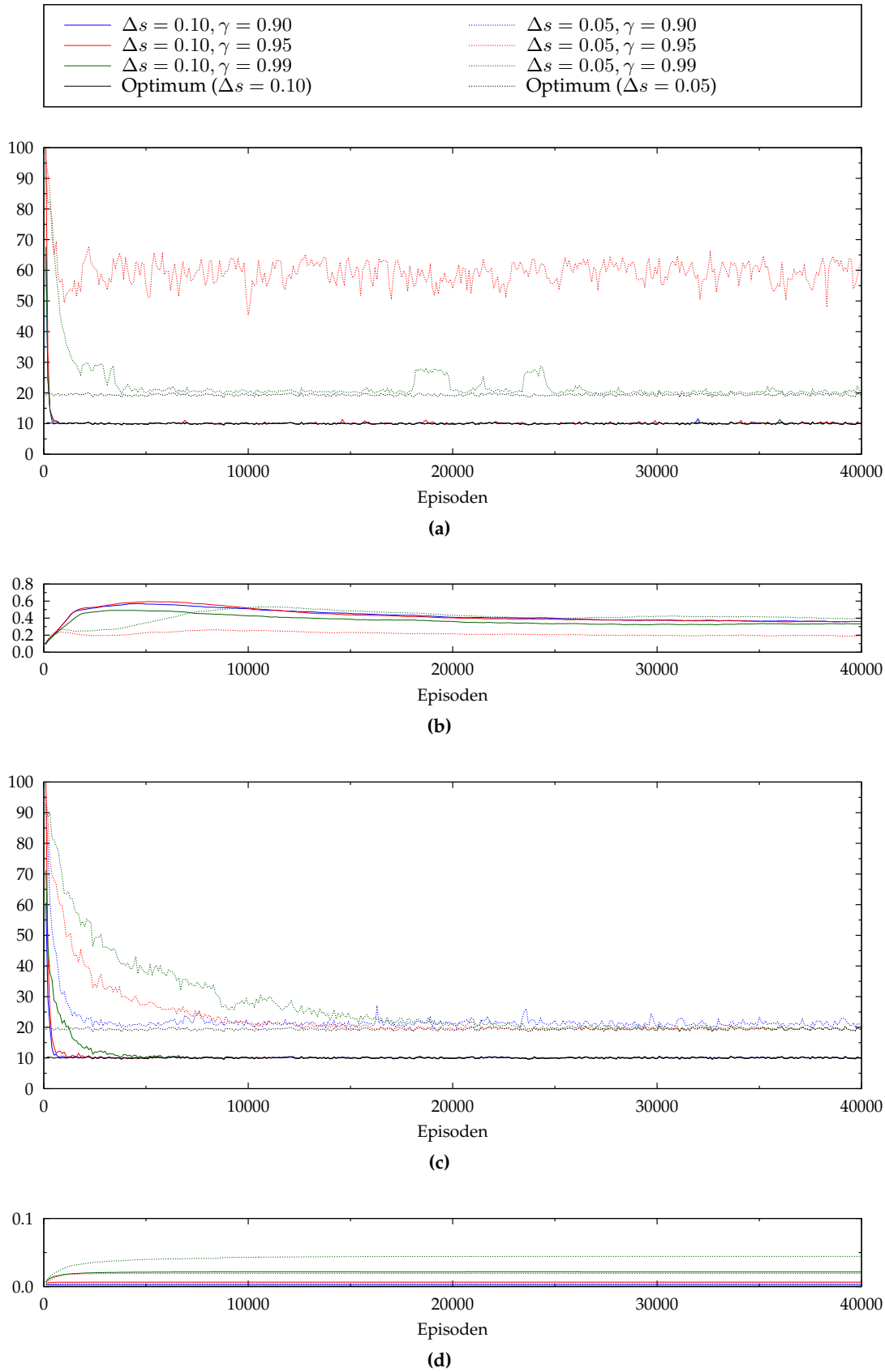


Abbildung 8.4: Lernverlauf von XCS und HCS in der Empty-Room-Lernumgebung für verschiedene Werte der Schrittweite Δs und des Diskontierungsfaktors γ: Durchschnittliche Anzahl von Schritten bis zum Ziel bei **(a)** XCS und **(c)** HCS sowie Populationsgröße von **(b)** XCS und **(d)** HCS.

venverlauf. Bei den Experimenten mit den beiden anderen Werten des Diskontierungsfaktors sind derartige Abweichungen nicht zu beobachten.

- Bei den Experimenten in der Empty-Room-Lernumgebung mit der kleinsten hier untersuchten Schrittweite ($\Delta s = 0.05$) entwickelt das HCS außer bei Verwendung des Diskontierungsfaktors $\gamma = 0.90$ wiederum ein nahezu optimales Verhalten, braucht dafür aber deutlich länger als im Fall der Schrittweite $\Delta s = 0.1$. Auch für $\gamma = 0.90$ kommt die vom HCS benötigte Anzahl von Schritten dem Optimum noch recht nahe. Dem XCS hingegen gelingt es – im Schnitt über die Läufe betrachtet – weder für $\gamma = 0.95$ noch für $\gamma = 0.99$ ein optimales Verhalten zu lernen, insbesondere im Fall $\gamma = 0.95$ benötigt es – im Durchschnitt über die Läufe des Experimentes betrachtet – zu jedem Zeitpunkt deutlich mehr Schritte als bei optimalem Verhalten. Eine Betrachtung der einzelnen Läufe zeigt jedoch, dass das XCS für $\gamma = 0.99$ in neun der zehn Einzelläufe ein fast optimales Verhalten lernt und nur im zehnten Lauf deutlich mehr Schritte benötigt als bei optimalem Verhalten. Im Fall $\gamma = 0.95$ gelingt es dem XCS immerhin noch in der Hälfte der Läufe, ein gutes oder sogar annähernd optimales Verhalten zu lernen, während es in der anderen Hälfte der Läufe nur in wenigen Fällen die Zielposition erreicht, bevor die Lernepisode abgebrochen wird.

- Wie schon bei fast allen bisher besprochenen Experimenten sind auch die vom HCS in der Empty-Room-Lernumgebung evolvierten Populationen deutlich kleiner als die vom XCS entwickelten. Der bereits in der Korridor-Lernumgebung aufgetretene Effekt, dass das XCS in einigen Fällen Populationen evolviert, die nur noch übergenerelle Klassifizierer enthalten, ist auch hier zu beobachten – bei den Läufen nämlich, in denen das XCS für $\Delta s = 0.05$ und $\gamma = 0.95$ kein sinnvolles Verhalten lernt. Die in diesen Läufen deutlich kleineren Populationen des XCS sind dafür verantwortlich, dass die mittlere Populationsgröße des XCS in dem mit diesen Parametern durchgeführten Experiment verhältnismäßig klein bleibt.

Auch wenn es beim HCS in der Empty-Room-Lernumgebungen bei einigen Setzungen der Schrittweite und des Diskontierungsfaktors länger als beim XCS dauert, ein optimales Verhalten zu lernen, so kann dem HCS insgesamt doch die bessere Leistung in dieser Lernumgebung bescheinigt werden: Für fast alle Kombinationen von Schrittweite und Diskontierungsfaktor lernt das HCS bis zum Ende des jeweiligen Experimentes ein nahezu optimales Verhalten. Aber auch in den Fällen, in denen ihm dies nicht gelingt, lernt es ein besseres Verhalten als das XCS. In den Experimenten, in denen das XCS letztlich ein optimales Verhalten lernt, zeigen sich doch immer wieder auch Abweichungen von diesem, was beim HCS nicht zu beobachten ist. In diesem Sinne entwickelt das HCS also ein stabileres Verhalten als das XCS. Schließlich sind auch in dieser Lernumgebungen die vom HCS entwickelten Populationen deutlich kleiner als die vom XCS evolvierten.

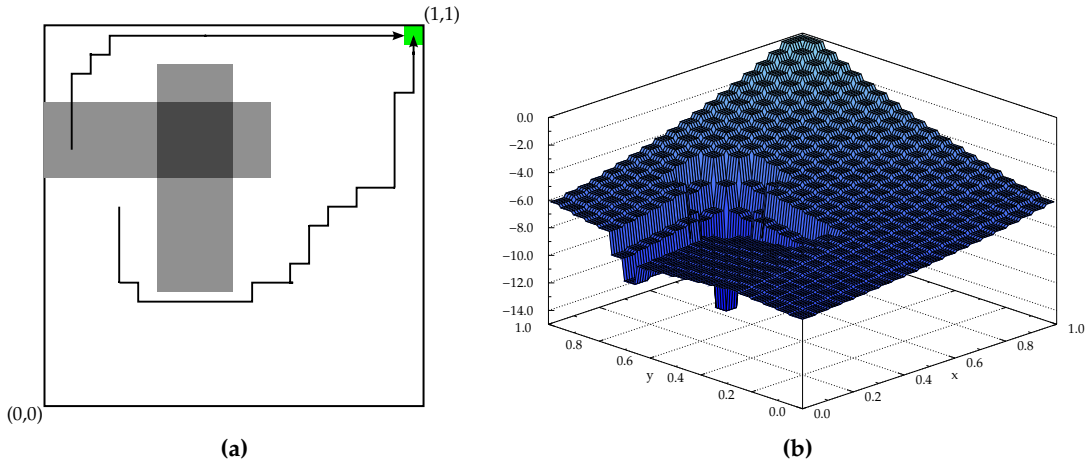
8.4 Die Puddles-Lernumgebung

Bei der Puddles-Lernumgebung handelt es sich um eine Erweiterung der Empty-Room-Lernumgebung. Dieser fügt sie Hindernisse hinzu, die ein in ihr lernender Reinforcement-Learning-Agent beachten muss.

8.4.1 Problembeschreibung

Die Puddles-Lernumgebung unterscheidet sich von der Empty-Room-Lernumgebung durch Hindernisse – „Pfützen“ – in Gestalt von Bereichen des Zustandsraumes, in denen die Aktionen des Agenten mit höheren Kosten verbunden sind – in denen also die von der Lernumgebung gewährten Belohnungen einen betragsmäßig größeren negativen Wert haben, als es sonst der Fall ist. Diese Bereiche sind in Abbildung 8.5(a), die den Zustandsraum der Puddles-Lernumgebung zeigt[5], grau dargestellt. Erreicht der Agent eine Position innerhalb der hellgrauen Bereiche, erhält er eine Belohnung der Höhe -2, gelangt er in den dunkelgrauen Bereich, hat die gewährte Belohnung den Wert -4. In allen anderen Fällen entsprechen die Belohnungen denen in der Empty-Room-Lernumgebung.

Anders als bei den bisher betrachteten Mehrschritt-Lernumgebungen ist es im Fall der Puddles-Lernumgebung nicht trivial, eine optimale Politik explizit anzugeben. Als Bezugspunkt für die Beurteilung und den Vergleich der Leistungen von XCS und HCS wird daher ein durch tabellarisches Reinforcement-Learning (Q-Learning) trainierter Agent herangezogen. Abbildung 8.5(b) zeigt die von diesem gelernte Wertefunktion der Puddles-Lernumgebung mit Schrittweite $\Delta s = 0.05$ bei Verwendung des Diskontierungsfaktors $\gamma = 0.95$.



(a) **(b)**

Abbildung 8.5: Die Puddles-Lernumgebung: **(a)** Skizze der Lernumgebung und **(b)** mittels tabellarischen Reinforcement-Learnings bestimmte Wertefunktion V für Schrittweite $\Delta s = 0.05$ und Diskontierungsfaktor $\gamma = 0.95$.

Die Politik, die sich ergibt, wenn der Agent sich stets in Richtung des steilsten Anstiegs dieser Wertefunktion bewegt, erscheint durchaus plausibel, auch wenn ihre Optimalität hier nicht nachgewiesen werden kann. In Abbildung 8.5(a) sind zwei Pfade eines dieser Politik folgenden Agenten eingezeichnet: In einem Fall umgeht der Agent die Hindernisse auf dem Weg zum Ziel, folgt also nicht dem kürzesten Weg zu diesem. Im anderen Fall, in dem er in einem Hindernis startet, verlässt er dieses nicht auf dem schnellsten Weg, sondern nimmt einen direkten Weg zum Ziel. Würde er den Bereich des Hindernisses schnellstmöglich (nach Süden) verlassen, wären die Kosten des resultierenden Umwegs größer als die durch die Bewegung durch das Hindernis verursachten.

[5] Die Anordnung der Pfützen wurde aus [Lanzi u. a. 2005c] übernommen.

8.4.2 Experimente

In der Puddles-Lernumgebung wurden Experimente mit einem Wert des Diskontierungsfaktors ($\gamma = 0.95$) und zwei Werten der Schrittweite ($\Delta s = 0.05, 0.1$) durchgeführt. Die Lernverläufe in diesen Experimenten sind in Abbildung 8.6 dargestellt.

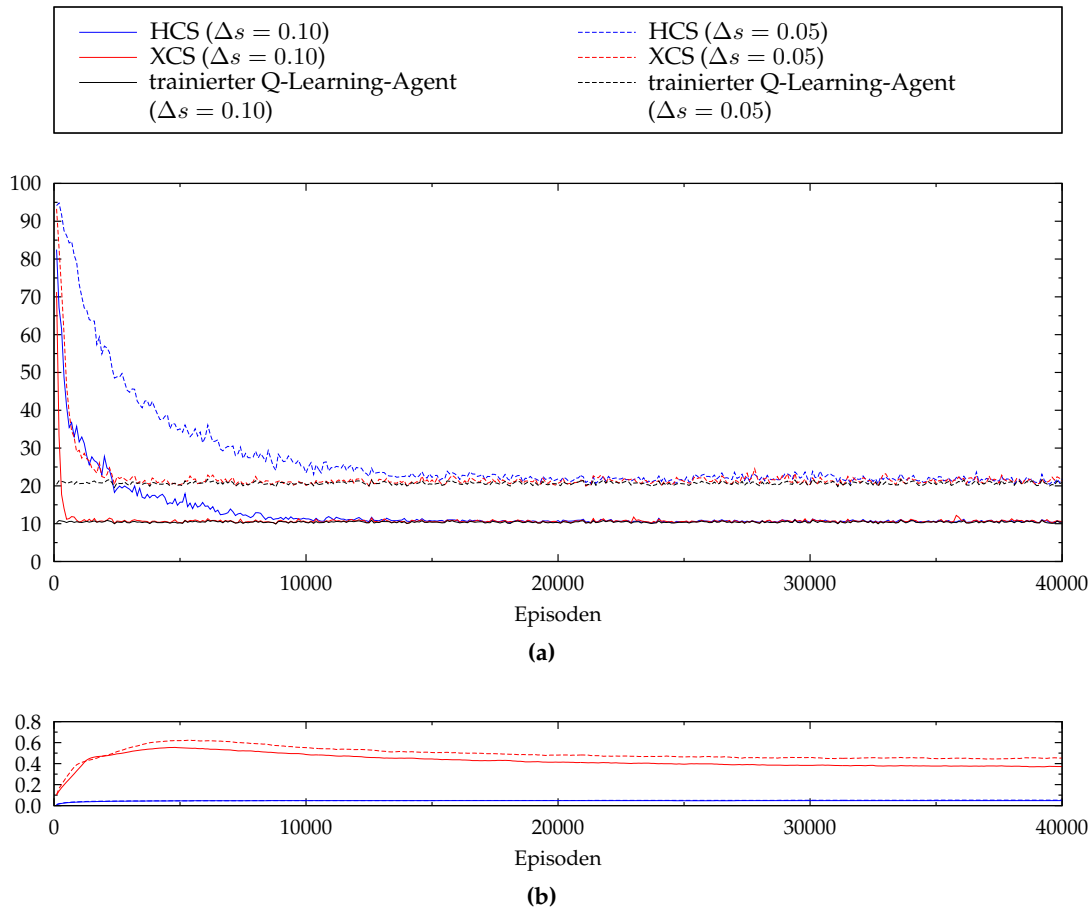


Abbildung 8.6: Lernverlauf von XCS und HCS in der Puddles-Lernumgebung für verschiedene Werte der Schrittweite Δs: **(a)** Durchschnittliche Anzahl von Schritten bis zum Erreichen des Ziels und **(b)** Entwicklung der Populationsgröße

Es lassen sich die folgenden Ergebnisse der in der Puddles-Lernumgebung durchgeführten Experimente festhalten:

- In der Puddles-Lernumgebung mit Schrittweite $\Delta s = 0.1$ benötigen sowohl das XCS als auch das HCS am Ende des jeweiligen Experimentes durchschnittlich etwa ebensoviele Schritte bis zum Ziel, wie der mit tabellarischem Reinforcement-Learning trainierte Agent. Das HCS benötigt allerdings deutlich mehr Lernepisoden als das XCS, um dies zu erreichen. Andererseits zeigen sich beim XCS wie schon bei der Empty-Room-Lernumgebung gelegentlich Spitzen in der Lernkurve, die auf eine zumindest temporär schlechtere Leistung hinweisen. Dies ist beim HCS nicht mehr der Fall, nachdem es einmal ein gutes Verhalten gelernt hat.

- Auch im Fall der kleineren Schrittweite $\Delta s = 0.05$ erlernen beide Systeme bis zum Ende der Experimente ein – gemessen an der Zahl durchschnittlich benötig-

ter Schritte – etwa gleich gutes Verhalten. Das HCS benötigt allerdings länger um dies zu erreichen. Beide Systeme benötigen im Schnitt mehr Schritte als der Agent, dessen Verhalten durch tabellarisches Reinforcement-Learning gelernt wurde.

- Die vom HCS evolvierten Populationen sind wiederum deutlich kleiner als die des XCS. Zudem liegt die Zahl der Klassifizierer des HCS auch deutlich unter der der Zustands-Aktions-Paare, deren Wert beim tabellarischen Reinforcement-Learning gelernt werden muss: Dabei wurde hier eine Diskretisierung des Zustandsraum in Gestalt eines Gitters von 20×20 Punkten verwendet, was in Verbindung mit den vier möglichen Aktionen zu 1600 Zustands-Aktions-Paare führt, deren Werte zu lernen waren. Demgegenüber enthalten die Endpopulationen des HCS in den durchgeführten Läufen durchschnittlich nur etwa 520 Klassifizierer.

Die Anzahl von Schritten, die sowohl das XCS als auch das HCS im Fall der Schrittweite $\Delta s = 0.05$ benötigen, liegt im Schnitt über der Zahl, die bei Beachtung der mit tabellarischem Reinforcement-Learning gelernten Politik gebraucht wird. Um diesen Punkt näher zu untersuchen, wurde das jeweilige System im Exploitationsmodus betrieben, um unter Verwendung der in den Einzelläufen der Experimente evolvierten Populationen zu bestimmen, wie viele Schritte es von 100×100 gitterartig im Zustandsraum verteilten Punkten bis zum Ziel benötigt. Dabei wurde festgestellt, dass die höhere Zahl benötigter Schritte in erster Linie dadurch zustande kommt, dass es bei nahezu allen der in den Einzelläufen evolvierten Populationen einige Zustände gibt, von denen aus das Ziel nicht erreicht wird, bevor die jeweilige Episode abgebrochen wird. Die im Schnitt höhere Zahl abgebrochener Episoden weist dabei das XCS auf. Im Schnitt erreicht dieses von 2.5% der Startzustände aus nicht das Ziel, im schlechtesten Fall gelang dies sogar von 14% der Startzustände aus nicht. Die diesbezügliche Bilanz des HCS fällt positiver aus: Es erreicht im Schnitt nur von 0.9% der Startzustände aus nicht das Ziel, der maximale Prozentsatz abgebrochener Episoden lag bei knapp 3%.

8.5 Zusammenfassung

In diesem Kapitel wurde das HCS zum Aktionenlernen in interaktiven Lernumgebungen eingesetzt. Die dabei erzielten Ergebnisse wurden verglichen mit den Resultaten analoger, unter Verwendung des XCS durchgeführter Experimente.
Betrachtet wurden zunächst zwei Lernumgebungen, in denen für jeden Zustand direkt angegeben werden kann, welche Aktionen in ihm optimal sind. Dies ermöglichte es, das von XCS respektive HCS gelernte Verhalten nicht nur miteinander, sondern auch mit dem Verhalten eines optimal agierenden Agenten zu vergleichen.
In beiden Lernumgebungen wurden jeweils mehrere Experimente durchgeführt, wobei deren Schwierigkeit durch Variation zweier Parameter – eines Parameters der Lernumgebung und eines Parameters des Lernenden Klassifizierenden Systems – beeinflusst wurde. Es zeigte sich, dass das HCS für jeden Wert des Parameters der Lernumgebung zumindest mit einem Wert des Systemparameters ein nahezu optimales Verhalten hervorbringen konnte. Dem XCS war dies nicht immer möglich. Zudem wies auch in den Fällen, in denen das XCS ein gutes Verhalten lernen konnte, die Entwicklung der die Leistung des Systems beschreibenden Größe während des Lernvorganges beim XCS deutlich stärkere Schwankungen auf als beim HCS.
Auch in der dritten hier verwendeten Lernumgebung, in der das zu erlernende Verhalten komplexer als in den beiden zuvor betrachteten war, wurden Experimente mit un-

terschiedlichem Schwierigkeitsgrad durchgeführt, wozu ein Parameter der Lernumgebung variiert wurde. Nur im leichteren Fall konnten XCS und HCS die Leistung eines zum Vergleich herangezogenen, mittels tabellarischen Reinforcement-Learnings trainierten Agenten erreichen. Im schwierigeren Fall gelang dies nicht, vielmehr gab es bei beiden Systeme auch am Ende nahezu aller Einzelläufe des jeweiligen Experiments noch Zustände, von denen aus das Ziel nicht vor Abbruch der Lernepisode erreicht wurde. Die Zahl dieser Zustände war beim XCS jedoch höher als beim HCS.
Wie es schon bei den zur Klassifikation und den meisten der zur Funktionsapproximation durchgeführten Experimenten der Fall war, evolvierte das HCS auch beim Aktionenlernen deutlich kleinere Populationen als das XCS.

Kapitel 9

Zusammenfassung und Ausblick

Zum Abschluss der Arbeit werden in diesem Kapitel deren Ergebnisse zusammengefasst und es werden Ansatzpunkte für weitergehende Forschungen aufgezeigt.

9.1 Zusammenfassung

In Gestalt des Hybriden Lernenden Klassifizierenden Systems (HCS) wurde in der vorliegenden Arbeit ein System zum maschinellen Lernen entwickelt, das in sich Charakteristika eines Lernenden Klassifizierenden Systems – also eines evolutionären Ansatzes – einerseits und eines Künstlichen Neuronalen Netzes andererseits vereint.
Motivation für diese Entwicklung war die Feststellung, dass die intervallbasierten Klassifiziererbedingungen, die das als derzeitiger Standard anzusehende eXtended Learning Classifier System (XCS) üblicherweise verwendet, nur Generalisierungen über Hyperquader in einem Zustandsraum erlauben, was der Struktur vieler Lernumgebungen nicht gerecht wird. Alternative Bedingungstypen, die für das XCS entwickelt wurden, überwinden zwar diese Einschränkung, sind jedoch recht kompliziert aufgebaut. Zudem evolviert das XCS auch unter Einsatz dieser Alternativen meist Populationen, die mindestens ebenso groß sind wie bei Verwendung intervallbasierter Bedingungen – das Ziel einer effizienteren Generalisierung wird also nicht erreicht.
Die Grundstruktur des HCS entspricht der gewöhnlicher Lernender Klassifizierender Systeme. Es verfügt über eine Wissensbasis in Form einer Population von Klassifizierern, eine für die Auswahl und Ausführung von Aktionen verantwortliche Performance-Komponente, eine Reinforcement-Komponente, die erhaltene Belohnungen an die Klassifizierer verteilt, und eine Discovery-Komponente, die neue Klassifizierer erzeugt.
Um mit Bedingungen möglichst einfacher Struktur möglichst vielfältige Generalisierungen zu ermöglichen, greift das HCS auf Gewichtsvektoren zurück, wie sie die Neuronen Selbstorganisierender Karten aufweisen. Für jede der in seiner Lernumgebung möglichen Aktionen verfügt das HCS über eine (Teil-)Population von Klassifizierern, die diese vertreten. Diese Populationen induzieren jeweils eine Voronoizerlegung des Zustandsraums der Lernumgebung, sodass es zu jedem Zustand genau einen passenden Klassifizierer in

jeder dieser Populationen gibt – den, in dessen Voronoizelle der Zustand liegt. Darin unterscheidet sich das HCS stark von herkömmlichen Lernenden Klassifizierenden Systemen, die in der Regel über mehrere passende Klassifizierer für jede Kombination von Zustand und Aktion verfügen.
Um die in einem präsentierten Zustand auszuführende Aktion zu bestimmen, betrachten die Performance-Komponenten Lernender Klassifizierender Systeme die Return-Vorhersagen der zu diesem passenden Klassifizierer. Meist werden dabei Systemvorhersagen für die einzelnen Aktionen berechnet, indem über die Vorhersagen der die gleiche Aktion vertretenden passenden Klassifizierer gemittelt wird, sodass ungenaue Vorhersagen einzelner Klassifizierer keinen großen Einfluss haben.
Eine Systemvorhersagenberechnung dieser Art ist beim HCS natürlich nicht möglich, da es stets nur einen passenden Klassifizierer pro Aktion gibt. Da anzunehmen ist, dass – zumindest in vielen Lernumgebungen – die Ausführung der gleichen Aktion in ähnlichen Zuständen auch zu ähnlichen Returns führt, berechnet das HCS die Systemvorhersage für eine Aktion stattdessen, indem es zwischen der Vorhersage des zum jeweils aktuellen Zustand passenden Klassifizierers der entsprechenden Teilpopulation und den Vorhersagen von zu ähnlichen Zuständen passenden Klassifizierern interpoliert.
Um die zuletzt genannten Klassifizierer zu bestimmen, wird jede Teilpopulation des HCS in einer dem Competitive Hebbian Learning des (Wachsenden) Neuralen Gases entsprechenden Weise mit einer Topologie versehen. Dazu werden in jedem Lernschritt die beiden dem präsentierten Zustand nächsten Klassifizierer durch eine Kante verbunden. Kanten werden wieder entfernt, wenn die durch sie verbundenen Klassifizierer über einen gewissen Zeitraum nicht als Paar aus Gewinner und Zweitem in Erscheinung treten. In die Berechnung der Systemvorhersage für eine Aktion gehen die Vorhersage des zum betrachteten Zustand passenden Klassifizierers der entsprechenden Teilpopulation sowie die Vorhersagen der mit diesem über Kanten verbundenen Klassifizierer ein.
Alternativ kann auf das Topologielernen verzichtet werden. In diesem Fall werden aus jeder Teilpopulation neben dem passenden Klassifizierer – also dem Klassifizierer mit dem geringsten euklidischen Abstand zum betrachteten Zustand in der Teilpopulation – auch die Klassifizierer mit den nächstkleineren Abständen bei der Systemvorhersagenberechnung berücksichtigt.
Die Reinforcement-Komponente des HCS arbeitet analog der des XCS. Basierend auf den beobachteten Belohnungen, die das System von der Lernumgebung erhält, passt sie die Return-Vorhersagen und weitere Attribute der aktivierten passenden Klassifizierer an.
Herkömmliche Lernende Klassifizierende Systeme lernen rein evolutionär; unter Verwendung eines Genetischen Algorithmus erzeugen sie durch Variation als gut erkannter Klassifizierer neue, potentiell bessere. Das HCS hingegen verwendet neben dem Genetischen Algorithmus ein weiteres, von den Neuronenadaptationsmechanismen Selbstorganisierender Karten inspiriertes Lernverfahren, das die Gewichtsvektor-Bedingungen der Klassifizierer an die beobachteten Zustände anpasst. Dies erfolgt unter Berücksichtigung der Genauigkeit und des Vorhersagefehlers: Macht ein Klassifizierer eine zutreffende Returnvorhersage für einen Zustand, so wird sein Gewichtsvektor in Richtung dieses Zustands verschoben, ist die Vorhersage hingegen unzutreffend, erfolgt eine Verschiebung von dem entsprechenden Zustand weg. Die Stärke der Anpassung ist von der Genauigkeit des Klassifizierers abhängig – je genauer ein Klassifizierer insgesamt ist, um so weniger stark wird sein Gewichtsvektor verändert. Dieser zusätzliche Lernmechanismus versetzt das HCS in die Lage, existierende Klassifizierer anpassen und verbessern

zu können, was bei Verwendung eines Genetischen Algorithmus als einzigem Lernverfahren nicht möglich ist.

Zur Evaluation des HCS wurde eine Simulations-Software entwickelt, die es in komfortabler Weise ermöglicht, sowohl mit dem HCS als auch mit dem zum Vergleich herangezogenen XCS Experimente durchzuführen. Mit Hilfe dieser Software wurde das HCS in verschiedenen Experimenten zum Lernen in einer ganzen Reihe von Lernumgebungen eingesetzt, die die drei klassischen Einsatzgebiete Lernender Klassifizierender Systeme abdecken: Klassifikation, Funktionsapproximation und Aktionenlernen.

Zunächst wurde das HCS an einigen Klassifikationsproblemen getestet. Dabei erzielte es nur teilweise bessere Ergebnisse – einen höheren Anteil richtiger Klassifikationen – als das XCS. Hierfür konnten zwei Ursachen ausgemacht werden: Erstens weisen die betrachteten Klassifikationsprobleme Zustandsräume auf, in denen nur Generalisierungen über achsenparallele Hyperquader möglich sind und an die daher das XCS mit intervallbasierten Klassifiziererbedingungen perfekt angepasst ist. Zweitens musste das HCS in der Lernumgebung, bei der seine Leistung am deutlichsten unter der des XCS lag, eine Voronoi-Zerlegung des Zustandsraums abbilden, in der stets in einigen benachbarten Voronoizellen den gleichen Aktionen die gleichen Belohnungen folgten. Dies erschwerte dem HCS die korrekte Positionierung seiner Klassifizierer. Dennoch gelang es dem HCS bei allen betrachteten Klassifikationsproblemen, Populationen zu evolvieren, die – bis auf kleine Ungenauigkeiten der Klassifiziererpositionen – den jeweils optimalen HCS-Populationen entsprachen. Auch waren die vom HCS evolvierten Populationen stets deutlich kleiner als die vom XCS hervorgebrachten.

In der zweiten Reihe durchgeführter Experimente wurde das HCS verwendet, um Funktionen zu approximieren. Lediglich im Fall einer Zielfunktion – einer achsenparallelen Stufenfunktion, für deren Approximation intervallbasierte Bedingungen ideal geeignet sind – blieb die Qualität der vom HCS entwickelten Approximation deutlich hinter der vom XCS erreichten zurück. Drei weitere Zielfunktionen wurden von beiden Systemen etwa gleich gut approximiert, bei den übrigen sechs betrachteten Funktionen übertraf die Approximationsgüte des HCS die des XCS.
Festzustellen war allerdings, dass dem HCS die Approximation von Funktionen an Stellen, an denen der Funktionswert sich schnell und stark verändert – zum Beispiel an Unstetigkeits- und Undifferenzierbarkeitsstellen –, schwerer fällt als dem XCS. Ursächlich hierfür ist, dass beim HCS stets nur ein Klassifizierer zu einem Zustand passt. Sagt dieser – was an Stellen der genannten Art kaum zu vermeiden ist – den Wert der Zielfunktion ungenau voraus, so kann das HCS dies nicht kompensieren. Dem XCS hingegen, dessen Population in der Regel zu jedem Zustand mehr als nur einen passenden Klassifizierer enthält, ist das eher möglich.
Wie schon im Fall der Klassifikation waren die vom HCS hervorgebrachten Populationen auch beim Einsatz zur Funktionsapproximation meist deutlich kleiner als die vom XCS evolvierten.

Schließlich wurde das HCS noch zum Aktionenlernen eingesetzt. In zwei Lernumgebungen, in denen die optimalen Aktionen für jeden Zustand bekannt sind, wurden jeweils mehrere Experimente mit durch verschiedene Setzungen von Parametern der Lernumgebung und des Lernenden Klassifizierenden Systems variiertem Schwierigkeitsgrad durchgeführt. Während das HCS für jeden Wert des variierten Parameters der Lernumgebung mit zumindest einem Wert des veränderten Systemparameters eine optimale Ak-

tionswahl lernen konnte, gelang dies dem HCS nicht in allen Fällen. Zudem zeigte sich, das die Leistung des HCS während eines Experimentes weit weniger starke Schwankungen aufwies als die des XCS.
In einer weiteren Lernumgebung war ein komplexeres Verhalten zu lernen als in den beiden zuerst betrachteten. In dieser Lernumgebung wurden zwei Experimente mit durch einen Parameter der Lernumgebung variierter Schwierigkeit durchgeführt. Als Bezugspunkt für die Leistung des HCS und des XCS wurde ein mittels tabellarischen Reinforcement-Learnings trainierter Agent herangezogen. Die Qualität dessen Verhaltens konnten sowohl das HCS wie auch das XCS nur im leichteren der durchgeführten Experimente erreichen. Im schwierigeren gab es bei beiden Systemen am Ende noch Zustände, von denen aus der zu erreichende Terminalzustand nicht vor Abbruch der jeweiligen Lernepisode erreicht wurde. Beim HCS gab es jedoch deutlich weniger derartige Zustände als beim XCS.
Auch bei den zum Aktionenlernen durchgeführten Experimenten enthielten die vom HCS evolvierten Populationen deutlich weniger Klassifizierer als die des XCS.

9.2 Ausblick

Obwohl das HCS in den im Rahmen dieser Arbeit durchgeführten Experimenten seine Leistungsfähigkeit unter Beweis stellen konnte und ferner aus diesen Experimenten einige seine Stärken und Schwächen betreffenden Erkenntnisse abgeleitet werden konnten, kann die Untersuchung und Entwicklung des HCS sicher nicht als mit dieser Arbeit abgeschlossen betrachtet werden. Daher sollen nun noch einige Richtungen skizziert werden, in die weiter gearbeitet werden könnte:

Das HCS verfügt über eine recht große Zahl einstellbarer Parameter, die sein Verhalten beeinflussen. Eine genauere Untersuchung der Bedeutung einzelner Parameter wäre wünschenswert. Zum einen, um das Verständnis des Systems und seiner Funktionsweise zu verbessern, zum anderen aber auch mit dem Ziel, für zumindest einen Teil der Parameter „gute“ Standardwerte zu ermitteln. Reizvoll wäre auch, zu untersuchen, ob eventuell einzelne Parameter automatisch in Abhängigkeit von anderen festgelegt werden können. Durch eine derartige Verringerung der Zahl vom Benutzer des Systems einzustellender Parameter könnte die praktische Anwendbarkeit des HCS sicher verbessert werden.

Auch eine zweite interessante Fragestellung bezieht sich auf die Parameter des Systems: Bei Selbstorganisierenden Karten hat sich die Verwendung lokaler, also neuronenspezifischer, Parameter oft als vorteilhaft erwiesen. Beim HCS haben bisher nahezu alle Parameter globale Gültigkeit; lediglich die Lernrate für die Adaptation der Gewichtsvektor-Bedingungen wird in Abhängigkeit von der Klassifizierergenauigkeit lokal angepasst. Eventuell wäre es sinnvoll, auch andere auf die Klassifizierer bezogene Parameter – etwa die Lernrate für die Anpassung der Klassifiziererattribute oder den Sharing-Radius – lokal zu steuern.

Der Genetische Algorithmus des HCS bietet einen weiteren Ansatzpunkt für zukünftige Untersuchungen: Das HCS wurde mit dem Ziel entwickelt, die aus der Verwendung intervallbasierter Bedingungen resultierende Beschränkung auf achsenparallele und vielen

Lernumgebungen nicht angemessene Unterteilungen des Zustandsraums zu überwinden. In dieser Hinsicht könnte sich die Verwendung von Uniformem oder Zwei-Punkt-Crossing-Over als kontraproduktiv erweisen, da durch diese Operatoren Gewichtsvektoren erzeugt werden, die auf den Knoten eines achsenparallelen Gitters liegen. Eine Evaluation anderer Crossing-Over-Operatoren für den Einsatz im HCS könnte dementsprechend sinnvoll sein. Auch die Effekte der Verwendung anderer als der bisher eingesetzten Selektions- und Mutationsmethoden zu untersuchen, könnte interessant sein.

Interessante Ansätze für mögliche Erweiterungen des HCS liefert auch die Literatur zum beständig weiterentwickelten XCS. Einige der dort verfolgten Ideen könnten auch auf das HCS übertragen werden. In [Lanzi u. D.Loiacono 2007] etwa wurde eine Variante des XCS beschrieben, die kontinuierliche Aktionen lernt. Dies ist sicher auch für das HCS eine sinnvolle Erweiterung. Ferner gibt es Ansätze, XCS-Klassifizierer mit verschiedenen Typen von Vorhersagen (skalar, linear, polynomiell,...) in einer Population zu evolvieren, um stets die lokal geeignetste Form für die Approximation einer Zielfunktion zu nutzen [Lanzi u. a. 2008]. Es bietet sich an, auch diese Erweiterung auf das HCS zu übertragen.

Neben den bisher genannten Forschungsansätzen stellt natürlich auch die weitere Evaluation des HCS, bei der auch die Komplexität der betrachteten Lernumgebungen erhöht werden kann, eine Richtung dar, in die weiter gearbeitet werden sollte.

Anhang A

Lineare Ausgleichsprobleme

Die Systemvorhersagenberechnung des HCS beinhaltet, wie in Kapitel 4 ausgeführt, an zentraler Stelle die Approximation eines Umgebungszustandes durch den Gewichtsvektor eines Gewinners und Differenzvektoren zu anderen im Match-Set enthaltenen Klassifizierern, mithin die Lösung eines linearen Ausgleichsproblems. Auf diesen Punkt wurde in Kapitel 4 jedoch nicht näher eingegangen, da es sich bei der Lösung derartiger Probleme um eine Standardaufgabe der numerischen Mathematik handelt, die dementsprechend auch in den meisten Lehrbüchern dieser Disziplin behandelt wird, etwa in [Stoer 1994; Hanke-Bourgeois 2002]. Der Vollständigkeit der Darstellung halber wird in diesem Anhang dennoch ein kurzer Überblick über lineare Ausgleichsprobleme und deren Lösung gegeben, insbesondere wird das in Kapitel 4 nur kurz erwähnte „Ausweichen" auf eine mittels Tychonoff-Regularisierung berechnete Näherungslösung begründet.

Im Folgenden seien $A \in \mathbb{R}^{m \times n}$ und $b \in \mathbb{R}^m$ gegeben. Gesucht wird eine Lösung $\bar{x} \in \mathbb{R}^n$ des linearen Gleichungssystems $Ax = b$ oder, falls dieses keine echte Lösung besitzt, ein $\bar{x} \in \mathbb{R}^n$, das die euklidische Norm des Residuums $Ax - b$ minimiert. Dies leistet auch eine echte Lösung, sofern eine solche existiert, es ist also das *lineare Ausgleichsproblem*

$$\min_{x \in \mathbb{R}^n} \|Ax - b\| \tag{A.1}$$

zu lösen.

Satz

(i) Das Ausgleichsproblem (A.1) besitzt mindestens eine Lösung.

(ii) $\hat{x} \in \mathbb{R}^n$ löst (A.1) genau dann, wenn $\hat{x}$ die *Normalengleichung* $A^T Ax = A^T b$ erfüllt.

(iii) Das Ausgleichsproblem (A.1) ist eindeutig lösbar genau dann, wenn $Rang(A) = n$.

(iv) Ist $Rang(A) < n$, so existiert genau eine Lösung $\bar{x}$ von (A.1) mit minimaler Norm. Für diese gilt $\bar{x} \in \mathrm{Kern}\,(\mathrm{A})^{\perp}$ und die Menge aller Lösungen ist $X = \{\bar{x} + v \mid v \in \mathrm{Kern}\,(\mathrm{A})\}$.

Beweis *Setze* $L := \mathrm{Bild}\,(\mathrm{A}) = \{Ax \mid x \in \mathbb{R}^n\}$ *und* $L^{\perp} := \mathrm{Bild}\,(\mathrm{A})^{\perp} = \left\{r \,\middle|\, A^T r = 0\right\}$. *Dann ist* $\mathbb{R}^m = L \oplus L^{\perp}$ *und somit*

$$b = s + r \tag{$*$}$$

mit eindeutig bestimmten $s \in L$, $r \in L^{\perp}$. *Ferner gilt* $s = Ax_0$ *für mindestens ein* $x_0 \in \mathbb{R}^n$. *Sei nun* $x \in \mathbb{R}^n$ *beliebig. Aus* $z := Ax - Ax_0 \in L$ *folgt* $r^T z = 0$ *und daher gilt:*

$$\|Ax - b\|^2 = \|z + Ax_0 - s - r\|^2 = \|z - r\|^2 = \|r\|^2 + \|z\|^2 \geq \|r\|^2 = \|Ax_0 - b\|^2$$

Somit ist x_0 *eine Lösung von* (A.1) *und es folgt (i).*

Ist $\hat{x} \in \mathbb{R}^n$ eine beliebige Lösung von (A.1)*, so gilt:*

$$\|r\|^2 = \|Ax_0 - b\|^2 = \|A\hat{x} - b\|^2 = \|A\hat{x} - Ax_0 - r\|^2 = \|A\hat{x} - Ax_0\|^2 + \|r\|^2$$

und daher $A\hat{x} = Ax_0$. Wegen $A^T r = 0$ folgt:

$$A^T b = A^T s + A^T r = A^T s = A^T A x_0 = A^T A\hat{x}$$

$\hat{x}$ löst also die Normalengleichungen.
Erfülle umgekehrt $\hat{x} \in \mathbb{R}^n$ die Normalengleichungen. Setze $s' := A\hat{x} \in L$, $r' := b - A\hat{x}$. Wegen

$$A^T A\hat{x} = A^T b = A^T s' + A^T r' = A^T A\hat{x} + A^T r'$$

folgt $A^T r' = 0$, also $r' \in L^\perp$. Die Eindeutigkeit der Zerlegung $(*)$ liefert: $A\hat{x} = s' = s = Ax_0$. Somit gilt $\|A\hat{x} - b\|^2 = \|Ax_0 - b\|^2$ und $\hat{x}$ ist eine Lösung von (A.1)*. Damit ist (ii) gezeigt.*

Wegen $x^T A^T Ax = \|Ax\|^2 \geq 0 \forall x \in \mathbb{R}^n$ ist die Matrix $A^T A$ positiv semidefinit. Für jedes $x \in \mathrm{Kern}\left(\mathrm{A^T A}\right)$ gilt wegen $x^T A^T Ax = \|Ax\|^2$ offenbar $Ax = 0$, also $x \in \mathrm{Kern}\,(\mathrm{A})$. Ferner ist offensichtlich $\mathrm{Kern}\,(\mathrm{A}) \subseteq \mathrm{Kern}\left(\mathrm{A^T A}\right)$, insgesamt folgt also: $\mathrm{Kern}\,(\mathrm{A}) = \mathrm{Kern}\left(\mathrm{A^T A}\right)$. Analog ergibt sich: AA^T ist positiv semidefinit und $\mathrm{Kern}\left(\mathrm{A^T}\right) = \mathrm{Kern}\left(\mathrm{AA^T}\right)$. Somit:

$$\begin{aligned}\mathrm{Rang}\left(\mathrm{A^T A}\right) &= n - dim(\mathrm{Kern}\left(\mathrm{A^T A}\right) = n - dim(\mathrm{Kern}\,(\mathrm{A})) \\ &= \mathrm{Rang}\,(\mathrm{A}) = \mathrm{Rang}\left(\mathrm{A^T}\right) \\ &= m - dim(\mathrm{Kern}\left(\mathrm{A^T}\right)) = m - dim(\mathrm{Kern}\left(\mathrm{AA^T}\right) = \mathrm{Rang}\left(\mathrm{AA^T}\right)\end{aligned}$$

Nach (ii) hat (A.1) *eine eindeutige Lösung genau dann, wenn das lineare Gleichungssystem $A^T Ax = A^T b$ eindeutig lösbar ist, wenn also die $n \times n$-Matrix $A^T A$ regulär ist. Da dies, wie gerade gesehen, genau dann der Fall ist, wenn $\mathrm{Rang}\,(\mathrm{A}) = n$ gilt, folgt (iii).*

Seien x_1, x_2 beliebige Lösungen von (A.1)*. Da $\mathbb{R}^n = \mathrm{Kern}\,(\mathrm{A}) \oplus \mathrm{Kern}\,(\mathrm{A})^\perp$, existieren eindeutige Darstellungen $x_i = u_i + v_i$, $u_i \in \mathrm{Kern}\,(\mathrm{A})^\perp, v_i \in \mathrm{Kern}\,(\mathrm{A}), i = 1, 2$. Aus dem Beweis von (i) ergibt sich: $0 = Ax_1 - Ax_2 = A(x_1 - x_2)$, also $x_1 - x_2 = (u_1 - u_2) + (v_1 - v_2) \in \mathrm{Kern}\,(\mathrm{A})$, mithin $u_1 - u_2 \in \mathrm{Kern}\,(\mathrm{A}) \cap \mathrm{Kern}\,(\mathrm{A})^\perp$, das heißt $u_1 = u_2 =: \bar{x}$.*
Jede Lösung $\hat{x}$ von (A.1) *kann also in der Form $\hat{x} = \bar{x} + \hat{v},\ \hat{v} \in \mathrm{Kern}\,(\mathrm{A})$ dargestellt werden. Wegen*

$$A^T b = A^T Ax_1 = A^T A\bar{x} A^T Av_1 = A^T A\bar{x}$$

ist $\bar{x}$ selbst eine Lösung von (A.1)*. Da ferner $\bar{x}^T v = 0$ für alle $v \in \mathrm{Kern}\,(\mathrm{A})$ gilt, folgt:*

$$\|\hat{x}\|^2 = \|\bar{x}\|^2 + \|\hat{v}\|^2 \geq \|\bar{x}\|^2$$

Gleichheit ist offenbar nur für $\hat{v} = 0$ gegeben, somit folgt die Behauptung (iv).

Satz und Definition
Es existiert genau eine Matrix $A^\dagger \in \mathbb{R}^{n \times m}$, die die folgenden vier Bedingungen erfüllt:

(i) $\left(AA^\dagger\right)^T = AA^\dagger$ (iii) $AA^\dagger A = A$

(ii) $A^\dagger A = \left(A^\dagger A\right)^T$ (iv) $A^\dagger AA^\dagger = A^\dagger$

Die durch (i) – (iv) eindeutig bestimmte Matrix $A^\dagger$ heißt die *Pseudoinverse* von A. Im Falle, dass A regulär ist, erfüllt offenbar die Inverse A^{-1} die Bedingungen (i) – (iv). Die Pseudoinverse einer regulären Matrix stimmt also mit ihrer Inversen überein.

Beweis *Sei P die Orthogonalprojektion von $\mathbb{R}^n$ auf $\mathrm{Kern}\,(\mathrm{A})^\perp$ und $\bar{P}$ die Orthogonalprojektion von $\mathbb{R}^m$ auf $\mathrm{Bild}\,(\mathrm{A})$. Ist $y \in \mathrm{Bild}\,(\mathrm{A})$ so gilt*

$$y = Ax = A(Px + (I-P)x) = APx \text{ für ein geeignetes } x \in \mathbb{R}^n$$

es existiert also ein $x_1 \in \mathrm{Kern}\,(\mathrm{A})^\perp$ mit $y = Ax_1$. Dieses ist eindeutig bestimmt: Für $x_1, x_2 \in \mathrm{Kern}\,(\mathrm{A})^\perp$ mit $Ax_1 = Ax_2$ gilt $0 = Ax_1 - Ax_2 = A(x_1 - x_2)$, also $x_1 - x_2 \in \mathrm{Kern}\,(\mathrm{A}) \cap \mathrm{Kern}\,(\mathrm{A})^\perp$, mithin $x_1 = x_2$.
Somit ist die Abbildung $f : \mathrm{Bild}\,(\mathrm{A}) \to \mathrm{Kern}\,(\mathrm{A})^\perp$, die jedem Element des Bildes von A dessen in $\mathrm{Kern}\,(\mathrm{A})^\perp$ eindeutiges Urbild zuordnet, wohldefiniert. Da sie offenbar auch linear ist, ist auch die Abbildung

$$A^\dagger := f \circ \bar{P} : \mathbb{R}^m \to \mathbb{R}^n$$

wohldefiniert und linear. Nach Konstruktion von $A^\dagger$ und f gilt:

- $AA^\dagger y = A(f(\bar{P}(y))) = \bar{P}y \; \forall y \in \mathbb{R}^m$, *d.h.* $AA^\dagger = \bar{P}$, *wegen* $\bar{P} = \bar{P}^T$ *folgt (i).*
- $A^\dagger Ax = f(\bar{P}(Ax)) = f(Ax) = Px \; \forall x \in \mathbb{R}^n$, *d.h.* $A^\dagger A = P$, *wegen* $P = P^T$ *folgt (ii).*
- $AA^\dagger Ax = \bar{P}Ax = Ax \; \forall x \in \mathbb{R}^n$, *also (iii).*
- $A^\dagger AA^\dagger y = A^\dagger \bar{P}y = f(\bar{P}^2 y) = f(\bar{P}y) = A^\dagger y \; \forall y \in \mathbb{R}^m$, *also (iv).*

Die oben konstruierte Abbildung $A^\dagger$ erfüllt also die Bedingungen (i) - (iv). Ist andererseits Z eine beliebige Abbildung, die (i) - (iv) erfüllt, so gilt:

$$\begin{aligned} Z &= ZAZ = ZAA^\dagger AZ = ZAA^\dagger AA^\dagger AA^\dagger AZ = A^T Z^T A^T A^{\dagger T} A^\dagger A^{\dagger T} A^T Z^T A^T \\ &= (AZA)^T A^{\dagger T} A^\dagger A^{\dagger T} (AZA)^T = A^T A^{\dagger T} A^\dagger A^{\dagger T} A^T = A^\dagger AA^\dagger AA^\dagger = A^\dagger \end{aligned}$$

Somit ist $A^\dagger$ die einzige Abbildung, die (i) - (iv) erfüllt.

Lemma
Die eindeutig bestimmte Minimum-Norm-Lösung des linearen Ausgleichsproblems (A.1) kann unter Benutzung der Pseudoinversen wie folgt dargestellt werden:

$$\bar{x} = A^\dagger b \tag{A.2}$$

Beweis *Für beliebiges $x \in \mathbb{R}^n$ gilt: $Ax - b = A(x - A^\dagger b) - (b - AA^\dagger b)$.*
Da $u := A(x - A^\dagger b) \in \mathrm{Bild}\,(\mathrm{A})$ und $v := b - AA^\dagger b = b - \bar{P}b \in \mathrm{Bild}\,(\mathrm{A})^\perp$ folgt:

$$\|Ax - b\|^2 = \|u\|^2 + \|v\|^2 \geq \|v\|^2 = \left\| AA^\dagger b - b \right\|^2, \; x \in \mathbb{R}^n$$

Also ist $A^\dagger b$ eine Lösung von (A.1). *Da ferner $A^\dagger b \in \mathrm{Kern}\,(\mathrm{A})^\perp$, folgt $A^\dagger b = \bar{x}$.*

Satz und Definition
Es existieren orthogonale Matrizen $U \in \mathbb{R}^{m \times m}$ und $V \in \mathbb{R}^{n \times n}$ sowie eine Matrix

$$\Sigma := (\sigma_i \delta_{ij}) \in \mathbb{R}^{m \times n}, \sigma_1 \geq \ldots \geq \sigma_r > \sigma_{r+1} = \ldots = \sigma_{\min(m,n)} = 0,$$

sodass

$$A = U \Sigma V^T \tag{A.3}$$

Dabei ist r der Rang der Matrix A. Die Werte $\sigma_1, \ldots, \sigma_r$ heißen Singulärwerte von A und die Darstellung (A.3) eine Singulärwertzerlegung von A.

Beweis *Da die Matrix $A^TA \in \mathbb{R}^{n\times n}$ symmetrisch und positiv semidefinit ist, sind alle ihre Eigenwerte größer oder gleich 0 und A^TA besitzt n linear unabhängige (normierte) Eigenvektoren $v_1, \ldots, v_n$[1]. Die Nummerierung dieser Vektoren kann o.B.d.A. so gewählt werden, dass die zugehörigen Eigenwerte $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_r > \lambda_{r+1} = \ldots = \lambda_n = 0$ absteigend sortiert sind. Die mit diesen Eigenvektoren als Spalten gebildete Matrix $V := (v_1 \cdots v_n)$ ist offenbar orthogonal. Mit den Setzungen $V_1 := (v_1 \cdots v_r)$, $V_2 := (v_r + 1 \cdots v_n)$, $D = \mathrm{diag}(\lambda_1, \ldots, \lambda_r)$ folgt:*

$$V^TA^TAV = \begin{bmatrix} V_1^T \\ V_2^T \end{bmatrix} A^TA \begin{bmatrix} V_1 & V_2 \end{bmatrix} = \begin{bmatrix} V_1^TA^TAV_1 & V_1^TA^TAV_2 \\ V_2^TA^TAV_1 & V_2^TA^TAV_2 \end{bmatrix} = \begin{bmatrix} D & 0 \\ 0 & 0 \end{bmatrix}$$

Da sich der Rang einer Matrix durch Multiplikation mit einer orthogonalen Matrix nicht ändert, ist offenbar $r = \mathrm{Rang\,(A)}$ und es gilt $AV_2 = 0$.

Setze nun $\sigma_i := \sqrt{\lambda_i}, i = 1, \ldots r$ und $\Sigma = \begin{bmatrix} D^{1/2} & 0 \\ 0 & 0 \end{bmatrix}$, $D^{1/2} = \mathrm{diag}(\sigma_1, \ldots, \sigma_r)$.

Setze ferner $u_i := \frac{Av_i}{\sigma_i}, i = 1, \ldots, r$. Wegen

$$u_i^Tu_j = \frac{v_i^TA^TAv_j}{\sigma_i\sigma_j} = \frac{v_i^T\lambda_jv_j}{\sigma_i\sigma_j} = \frac{\sigma_j}{\sigma_i}v_i^Tv_j = 0\ ,\ u_i^Tu_i = \frac{\sigma_i}{\sigma_i}v_i^Tv_i = 1$$

sind $u_1, \ldots, u_r$ orthonormal und können durch geeignet gewählte Vektoren $u_{r+1}, \ldots, u_m$ zu einer Orthonormalbasis des $\mathbb{R}^m$ ergänzt werden, sodass die Matrix $U := (u_1 \cdots u_m)$ orthogonal ist. Mit $U_1 := AV_1D^{-1/2} = (u_1 \cdots u_r)$, $U_2 := (u_{r+1} \cdots u_m)$ folgt wegen $U_1D^{1/2} = AV_1$:

$$U\Sigma V^T = \begin{bmatrix} U_1 & U_2 \end{bmatrix} \begin{bmatrix} D^{1/2} & 0 \\ 0 & 0 \end{bmatrix} V^T = \begin{bmatrix} U_1D^{1/2}0 \end{bmatrix} V^T = \begin{bmatrix} AV_1 & AV_2 \end{bmatrix} V^T = AVV^T$$

Aufgrund der Orthogonalität von V gilt somit $A = U\Sigma V^T$, q.e.d.

Lemma
Sei eine Singulärwertzerlegung $U\Sigma V^T$ von A gegeben.

(i) Durch $V\Sigma^\dagger U^T$ ist eine Singulärwertzerlegung der Pseudoinversen $A^\dagger$ von A gegeben, wobei

$$\Sigma^\dagger := \left(\sigma_i^\dagger\delta_{ij}\right) \in \mathbb{R}^{n\times m}, \sigma_i^\dagger = \begin{cases} \frac{1}{\sigma_i} & , i = 1, \ldots, r \\ 0 & \imath = r+1, \ldots, \min(m,n) \end{cases}$$

(ii) Die Lösung (A.2) des Ausgleichsproblems (A.1) kann wie folgt geschrieben werden:

$$\bar{x} = \sum_{i=1}^{r} \frac{u_i^Tb}{\sigma_i}v_i \tag{A.4}$$

Dabei bezeichnen u_i und v_i die i-ten Spalten von U respektive V.

Beweis *Bezeichnet I_r die r-dimensionale Einheitsmatrix, so gilt offensichtlich*

$$\Sigma\Sigma^\dagger = \begin{bmatrix} I_r & 0 \\ 0 & 0 \end{bmatrix} = (\Sigma\Sigma^\dagger)^T \text{ und analog } \Sigma^\dagger\Sigma = (\Sigma^\dagger\Sigma)^T.$$

Somit folgt $\Sigma\Sigma^\dagger\Sigma = \Sigma$, $\Sigma^\dagger\Sigma\Sigma^\dagger = \Sigma^\dagger$ und weiter:

$$\begin{aligned} (A(V\Sigma^\dagger U^T))^T &= (U\Sigma V^TV\Sigma^\dagger U^T)^T = (U\Sigma\Sigma^\dagger U^T)^T \\ &= U\Sigma\Sigma^\dagger U^T = U\Sigma V^TV\Sigma^\dagger U^T \qquad = A(V\Sigma^\dagger U^T) \\ ((V\Sigma^\dagger U^T)A)^T &= (V\Sigma^\dagger U^TU\Sigma V^T)^T = (V\Sigma^\dagger\Sigma V^T)^T \\ &= V\Sigma^\dagger\Sigma V^T = V\Sigma^\dagger U^TU\Sigma V^T \qquad = (V\Sigma^\dagger U^T)A \end{aligned}$$

[1] Ein Beweis hierfür findet sich zum Beispiel in [Stoer u. Bulirsch 1978], Kapitel 6.4.

$$A(V\Sigma^\dagger U^T)A = U\Sigma V^T V\Sigma^\dagger U^T U\Sigma V^T = U\Sigma\Sigma^\dagger\Sigma V^T = U\Sigma V^T = A$$
$$(V\Sigma^\dagger U^T)A(V\Sigma^\dagger U^T) = V\Sigma^\dagger U^T U\Sigma V^T V\Sigma^\dagger U^T = V\Sigma^\dagger\Sigma\Sigma^\dagger U^T = V\Sigma^\dagger U^T$$

Da somit $V\Sigma^\dagger U^T$ die die Pseudoinverse von A bestimmenden Bedingungen erfüllt, ist (i) bewiesen. Es folgt unmittelbar, dass $\bar{x} = A^\dagger b$ die in (ii) angegebene Darstellung besitzt.

Die Berechnung der Singulärwertzerlegung einer Matrix ist numerisch stabil und effizient möglich, ein Algorithmus hierfür findet sich etwa in [W.H.Press u. a. 1992]. An gleicher Stelle wird auch empfohlen, diese zur Ermittlung der Minimum-Norm-Lösung $\bar{x}$ eines linearen Ausgleichsproblems (A.1) zu verwenden, was jedoch nur dann problemlos möglich ist, wenn keine sehr kleinen Singulärwerte auftreten. Solche können sich zum Beispiel in dem in 4.3.2 angesprochenen Fall ergeben, dass die Matrix A zwei oder mehr nahezu kollineare Spalten enthält. Aufgrund der Kehrwertbildung in (A.4) können sehr kleine Singulärwerte dazu führen, dass sich für zwei sehr ähnliche Ausgleichsprobleme stark voneinander abweichende Lösungen ergeben, wie etwa in den folgenden Beispielen[2]:

Für

$$A := \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}, b := \begin{pmatrix} 3 \\ 2 \\ 4 \\ 3 \\ 2 \end{pmatrix}$$

ergibt sich

$$\Sigma = \begin{pmatrix} \approx\sqrt{3} & 0 & 0 & 0 \\ 0 & \approx\sqrt{2} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

und damit

$$\bar{x} = \begin{pmatrix} 2.66666666666667 \\ 1 \\ 4 \\ 1 \end{pmatrix}$$

Für

$$A := \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1e^{-7} \end{pmatrix}, b := \begin{pmatrix} 3 \\ 2 \\ 4 \\ 3 \\ 2 \end{pmatrix}$$

ergibt sich

$$\Sigma = \begin{pmatrix} \approx\sqrt{3} & 0 & 0 & 0 \\ 0 & \approx\sqrt{2} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & \approx 6e^{-8} \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

und damit

$$\bar{x} = \begin{pmatrix} 3 \\ 10000002 \\ 4 \\ -10000000 \end{pmatrix}$$

Im rechten der obigen Beispiele ist die Kollinearität der zweiten und der vierten Spalte von A leicht gestört. Der Effekt des sich daraus ergebenden sehr kleinen Singulärwertes besteht darin, dass die entsprechenden Koeffizienten der berechneten Lösung $\bar{x}$ sehr große, betragsmäßig fast gleiche, jedoch mit verschiedenen Vorzeichen behaftete Werte haben, sodass sich die Beiträge dieser Spalten bei der Berechnung von $A\bar{x}$ nahezu aufheben, die kleine Differenz jedoch extrem verstärkt wird.
Da sich sehr kleine Singulärwerte auch durch Rundungsfehler bei numerischen Berechnungen, ergeben können, wird anstelle von $\bar{x}$ wird oft die folgende Näherungslösung benutzt, die sich für $\mu \to 0$ offenbar $\bar{x}$ annähert und die den Einfluss sehr kleiner Singulärwerte gegen Null senkt:

$$\tilde{x} = \sum_{i=1}^{r} \frac{\sigma_i}{\sigma_i^2 + \mu} u_i^T v_i,\ 0 < \mu < 1 \tag{A.5}$$

[2]Die Berechnung der hier aufgeführten Lösungen erfolgte mit dem auch von der HCS-Implementation dieser Arbeit bei der Berechnung von Singulärwertzerlegungen verwendeten Programmcode.

Hierdurch wird der oben beschriebene Effekt unterbunden, in den obigen Beispielen ergibt sich mit $\mu = 0.01$:

$$\tilde{x} = \begin{pmatrix} 2.66657778074064 \\ 0.999950002499875 \\ 3.999600039996 \\ 0.999950002499875 \end{pmatrix} \qquad \tilde{x} = \begin{pmatrix} 2.6665777474212 \\ 1.00028327475995 \\ 3.999600039996 \\ 0.99961669691257 \end{pmatrix}$$

Die Verwendung von $\tilde{x}$ entspricht der Einbeziehung eines Strafterms für betragsmäßig große Koeffizienten in die durchzuführende Minimierung:

Satz

(i) Bezeichnet I die n-dimensionale Einheitsmatrix, so ist die Matrix $A^TA + \mu I$ stets regulär. Das Gleichungssystem $\left(A^TA + \mu I\right) x = A^Tb$ besitzt somit immer eine eindeutige Lösung. Diese ist gerade durch $\tilde{x}$ aus (A.5) gegeben.

(ii) $\hat{x} \in \mathbb{R}^n$ löst das regularisierte Ausgleichsproblem

$$\min_{x \in \mathbb{R}^n} \left[\|A \cdot x - b\|^2 + \mu \|x\|^2 \right], \ 0 < \mu < 1 \tag{A.6}$$

genau dann, wenn $\hat{x}$ die *Normalengleichungen* $\left(A^TA + \mu I\right) x = A^Tb$ erfüllt. Nach (i) besitzt (A.6) also die eindeutige Lösung $\tilde{x}$.

Beweis *Ist durch $A = U\Sigma V^T$ eine Singulärwertzerlegung von A gegeben, so folgt*

$$A^TA + \mu I = V\Sigma^T U^T U \Sigma V^T + \mu V V^T = V(\Sigma^T\Sigma + \mu I)V^T,$$
$$\Sigma^T\Sigma + \mu I = diag(\sigma_1^2 + \mu, \ldots \sigma_r^2 + \mu, \mu, \ldots, \mu)$$

Da diese Diagonalmatrix offensichtlich regulär ist und V orthogonal, ist auch $A^TA + \mu I$ regulär. Das Gleichungssystem $(A^TA + \mu I)x = A^Tb$ besitzt somit die eindeutige Lösung $(A^TA + \mu I)^{-1}A^Tb$. Mit der Singulärwertzerlegung $A = U\Sigma V^T$ von A folgt, dass diese in der in (A.5) *angegebenen Form dargestellt werden kann:*

$$\begin{aligned}(A^TA + \mu I)^{-1}A^Tb &= V(\Sigma^T\Sigma + \mu I)^{-1}V^T = V(\Sigma^T\Sigma + \mu I)^{-1}\Sigma^T U^T b \\ &= V \cdot \operatorname{diag}\left(\frac{\sigma_1}{\sigma_1^2 + \mu}, \ldots, \frac{\sigma_r}{\sigma_r^2 + \mu}, 0, \ldots, 0\right) \cdot U^T b = \tilde{x}\end{aligned}$$

Somit ist (i) bewiesen. Betrachte nun die Funktion $h : \mathbb{R}^n \to \mathbb{R},\ x \longmapsto \|Ax - b\|^2 + \mu \|x\|^2$. Wegen (i) ist $\tilde{x}$ der einzige kritische Punkt von h:

$$\operatorname{grad}(h(x)) = 2\left[\left(A^TA + \mu I\right)x - A^Tb\right] = 0 \Longleftrightarrow x = (A^TA + \mu I)^{-1}A^Tb = \tilde{x}$$

Da A^TA positiv semidefinit und das Skalarprodukt auf $\mathbb{R}^n$ positiv definit ist, ist ferner die Hesse-Matrix $H(h) = 2\left[A^TA + \mu I\right]$ von h positiv definit:

$$x^TH(h)x = 2\left(\underbrace{x^TA^TAx}_{\geq 0} + \mu \underbrace{x^Tx}_{>0}\right) > 0,\ x \in \mathbb{R}^n \setminus \{0\}$$

Insgesamt hat also die Funktion h in $\tilde{x}$ ihr einziges Minimum, damit ist auch (ii) bewiesen.

Die beschriebene Vorgehensweise – die Betrachtung des regularisierten Ausgleichsproblems (A.6) zur Bestimmung einer Näherungslösung des eigentlich zu lösenden linearen Ausgleichsproblems (A.1) – ist als *Tychonoff-Regularisierung*, *Methode der regularisierten kleinsten Quadrate* oder *Ridge-Regression* bekannt.

Anhang B

Notationsübersicht

Im Folgenden wird die in der vorliegenden Arbeit verwendete Notation zusammengefasst. Die Auflistung wurde nach den Abschnitten, in denen die Bezeichner das erste Mal auftreten, gegliedert. Da stets versucht wurde, die in der Literatur üblichen Bezeichnungen zu verwenden, war leider nicht zu vermeiden, dass manche Bezeichner mehrfach in unterschiedlichen Bedeutungen auftreten. In der Arbeit geht jedoch stets aus dem Kontext hervor, in welcher Bedeutung eine Bezeichnung verwendet wird. Bezeichnungen, die sich auf einzelne der bei der Untersuchung des HCS verwendeten Lernumgebungen beziehen, wurden hier nicht aufgenommen.

Optimierungsprobleme (Abschnitt 2.1.2)

Ω	Suchraum, die Menge der möglichen Lösungen eines Optimierungsproblems
ω	Lösungskandidat, ein Element von Ω
f	Bewertungsfunktion $f : \Omega \to \mathbb{R}$, ordnet jedem Lösungskandidaten ω einen Gütewert zu
$O(f)$	Menge der globalen Optima von f auf Ω
$\succ$	Vergleichsoperation ‚besser als' zum Vergleich von Lösungskandidaten

Genetische Algorithmen (Abschnitt 2.1.4)

Ω	Phänotypraum, entspricht dem Suchraum bei Optimierungsproblemen
ω	Phänotyp, ein Element von Ω
$\mathcal{G}$	Genraum
g	Genotyp, ein Element von $\mathcal{G}$
f	Fitnessfunktion $f : \Omega \to \mathbb{R}$, ordnet jedem Phänotypen ω einen Fitnesswert zu
dec	Dekodierungsfunktion $dec : \mathcal{G} \to \Omega$
F	Induzierte Fitnessfunktion $F = f \circ dec$, ordnet jedem Genotypen g einen Fitnesswert zu
P, P_t	Population (nach t-maligem Durchlaufen des evolutionären Zyklus)
N	(feste) Größe einer Population
μ	Anzahl der in einer Generation erzeugten Nachkommen
p_{mut}	Wahrscheinlichkeit für die Mutation eines Allels
p_{cross}	Rekombinationswahrscheinlichkeit

Reinforcement-Learning-Szenario (Abschnitt 2.2.1)

t	(diskreter) Zeitschritt, Lernschritt
$\mathcal{S}$	(diskrete) Zustandsmenge einer Lernumgebung
n	Dimension des Zustandsraums
s	Lernumgebungszustand, ein Element von $\mathcal{S}$
s_t	Zustand der Lernumgebung im Zeitschritt t
$\mathcal{A}$	Aktionsmenge $\widehat{=}$ Menge der in einer Lernumgebung möglichen Aktionen
m	Mächtigkeit der Aktionsmenge $\mathcal{A}$
a	Aktion, ein Element von $\mathcal{A}$
a_t	im Zeitschritt t ausgeführte Aktion
r	Belohnung
r_t	Belohnung, die die Lernumgebung zu Beginn des Zeitschritts t gewährt
π	Politik $\pi : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$; gibt an, mit welcher Wahrscheinlichkeit eine Aktion a in einem Zustand s ausgeführt wird.
π_t	die von einem Agenten im Zeitschritt t verfolgte Politik

Wertefunktionen (Abschnitt 2.2.2)

R_t	Return, der auf den Zeitschritt t folgt
T	Zeitpunkt des Erreichens eines Terminalzustandes
γ	Diskontierungsfaktor für die Returnabschätzung
V^π	Zustands-Wertefunktion $V^\pi : \mathcal{S} \rightarrow \mathbb{R}$ zur Politik π
Q^π	Aktions-Wertefunktion $Q^\pi : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ zur Politik π
$P^a_{ss'}$	Wahrscheinlichkeit für den Übergang von einem Zustand s in einen Zustand s' bei Ausführung der Aktion a
$R^a_{ss'}$	Erwartungswert der Belohnung beim Übergang von einem Zustand s in einen Zustand s' unter Ausführung der Aktion a
π^*	optimale Politik
V^*	Zustands-Wertefunktion der optimalen Politik π^*
Q^*	Aktions-Wertefunktion der optimalen Politik π^*

Reinforcement-Learning-Verfahren (Abschnitt 2.2.3)

V^π_k	k-te Iteration bei Schätzung der Zustands-Wertefunktion V^π zur Politik π
$V(s)$	geschätzter Wert eines Zustandes s
$Q(s, a)$	geschätzter Wert einer Aktion a im Zustand s
β	Lernrate
ε	Explorationsparameter (ε-gierige Politik)
$e_t(s)$	Aktivierung eines Zustandes s im Zeitschritt t
λ	Diskontierungsfaktor der Aktivierung

Holland'sches LCS (Abschnitt 2.3.3)

#	Wildcard-Symbol
p	Stärke (strength) eines Klassifizierers
k	Länge der Nachrichtenliste
Bid	Gebot eines Klassifizierers
ρ	Proportionalitätsfaktor der Gebots-Berechnung

ZCS (Abschnitt 2.3.4)

M, M_t	Match-Set eines Lernenden Klassifizierenden Systems (im Lernschritt t)
A, A_t	Action-Set eines Lernenden Klassifizierenden Systems (im Lernschritt t)
B	Bucket
β	Bruchteil der Stärke, den im Action-Set enthaltene Klassifizierer abgeben, Bruchteil erhaltener Belohnungen, der auf sie verteilt wird
γ	Anteil des Buckets, der auf Klassifizierer des Action-Sets des vorherigen Lernschritts verteilt wird
p_{A_t}	Summe der Stärkewerte der im Action-Set enthaltenen Klassifizierer
τ	Reduktionsfaktor für die Stärkewerte der im Match-Set, aber nicht im Action-Set, enthaltenen Klassifizierer

XCS (Abschnitt 2.4)

Wissensbasis (Abschnitt 2.4.3)

$N_{\max}$	maximal zulässige Größe einer Population
cl	Klassifizierer
$action$	Aktionsteil eines Klassifizierers
exp	Erfahrung eines Klassifizierers
p	Vorhersage eines Klassifizierers
ε	Vorhersagefehler eines Klassifizierers
κ	Genauigkeit eines Klassifizierers
f	Fitness eines Klassifizierers
ts	Zeitstempel eines Klassifizierers
as	Action-Set-Größen-Abschätzung eines Klassifizierers
num	Vielfachheit eines Klassifizierers

Performance-Komponente (Abschnitt 2.4.4)

PA_a	Systemvorhersage für eine Aktion a
PA	Prediction-Array aller Systemvorhersagen
p_{explr}	Explorationswahrscheinlichkeit für die Ausführung einer anderen als der am besten eingeschätzten Aktion

Reinforcement-Komponente (Abschnitt 2.4.5)

$\mathcal{P}$	Returnabschätzung
γ	Diskontierungsfaktor für die Returnabschätzung
β	Lernrate für die Klassifizierer-Attribute
ε_0	Fehlerschwellenwert
α	Abfallrate für die Genauigkeitsberechnung
ν	Exponent für die Genauigkeitsberechnung

Discovery-Komponente (Abschnitt 2.4.6)

Θ_{MNA}	Anzahl von Aktionen, die mindestens im Match-Set vertreten sein muss
$p_\#$	Wahrscheinlichkeit für das Auftreten eines Wildcards an einem Allel einer Klassifiziererbedingung
Θ_{GA}	Schwellenwert für die Ausführung des Genetischen Algorithmus
Θ_{SUB}	Erfahrung, die ein Klassifizierer haben muss, um einen anderen subsumieren zu können
p_{spec}	Ersetzungswahrscheinlichkeit des Spezifizierungs-Operators
$\bar{f}$	durchschnittliche Fitness der in der Population P enthaltenen Klassifizierer
$\mathrm{vote}(\cdot)$	Löschvotum eines Klassifizierers
Θ_{DEL}	Erfahrungsschwellenwert für das Löschen von Klassifizierern
δ	Anteil von $\bar{f}$, den die Fitness eines Klassifizierers unterschreiten muss, um bei der Berechnung seines Löschvotums berücksichtigt zu werden

Klassifizierer für reellwertige Zustände (Abschnitt 2.4.7)

ξ_k	k-tes Element eines Zustandes $s = (\xi_1, \xi_2, \ldots, \xi_n)^T \in \mathcal{S} \subset \mathbb{R}^n$
l_i, u_i	untere bzw. obere Grenze des i-ten Intervalls einer intervallbasierten Bedingung

Funktionsapproximation mit dem XCS (Abschnitt 2.4.8)

$p(s)$	berechnete Vorhersage $p : \mathcal{S} \to \mathbb{R}$ eines Klassifizierers
$\boldsymbol{v}$	Vektor der für die Vorhersagenberechnung verwendeten Gewichte $\boldsymbol{v} = (v_0, v_1, \ldots, v_n)^T$
ξ_0	Offset-Parameter für die Vorhersagenberechnung
s'	um ξ_0 erweiterter Zustand s: $s'(\xi_0, \xi_1, \xi_2, \ldots, \xi_n)^T$
β_0	Lernrate für die Gewichte $\boldsymbol{v}$ der Vorhersagenberechnung

Neuronenkonkurrenzmodell (Abschnitt 3.1.2)

s	Reizvektor $s = (\xi_1, \xi_2, \ldots, \xi_n)^T \subseteq \mathbb{R}^n$
$\mathcal{I}$	Eingabeschicht
n	Anzahl der Neuronen der Eingabeschicht
e_k	Neuron der Eingabeschicht
$\mathcal{K}$	Konkurrenzschicht
c_i	Neuron der Konkurrenzschicht
$w_{i,k}$	Stärke der Synapse zwischen e_k und c_i
$\boldsymbol{w}_i$	Gewichtsvektor $\boldsymbol{w}_i = (w_{i,1}, \ldots, w_{i,n})^T$ des Neurons c_i
$\gamma_{i,j}$	Stärke der Synapse zwischen Neuronen c_i und c_j der Konkurrenzschicht
σ	Ausgabefunktion der Konkurrenzneuronen
$o_i(t)$	Ausgabe des Neurons c_i zur Zeit t
η	Lernrate für die Anpassung des Gewichtsvektoren der Neuronen der Konkurrenzschicht
$\dot{}_{w(s)}$	das Gewinnerneuron zur Eingabe s bezeichnende Indizierung

Selbstorganisierende Karten (Abschnitt 3.2)

$\mathcal{N}$	Knotenmenge (Menge der Neuronen einer Selbstorganisierenden Karte)
N	Anzahl von Neuronen einer Selbstorganisierenden Karte
c_i	Neuron, Element von $\mathcal{N}$
$\boldsymbol{w}_i$	Gewichtsvektor des Neurons c_i
$\mathcal{E}$	Kantenmenge einer Selbstorganisierenden Karte
$e_{i,j}$	Kante zwischen den Neuronen c_i und c_j
N_i	Anzahl der Nachbarn des Neurons c_i
$c_{i,j}$	j-ter Nachbar des Neurons c_i
$\boldsymbol{w}_{i,j}$	Gewichtsvektor von $c_{i,j}$
Φ	die durch eine Selbstorganisierende Karte realisierte Abbildung $\Phi : \mathbb{R}^n \to \mathcal{N}$
V_i	Voronoizelle des Neurons c_i
$\eta_i(t)$	zeitabhängige Lernrate für die Anpassung des Gewichtsvektors des Neurons c_i
$d^{\mathcal{N}}(\cdot, \cdot)$	Metrik auf $\mathcal{N}$
$h_i(d, t)$	Nachbarschaftsfunktion des Neurons c_i

Kohonenkarte (Abschnitt 3.2.1)

$\eta(t)$	zeitabhängige globale Lernrate für die Anpassung des Gewichtsvektors
$r_a(t)$	Aktivierungsradius

Neurales Gas (Abschnitt 3.2.2)

k_i	Rang des Neurons c_i
$\lambda(t)$	zeitabhängiger Parameter der Nachbarschaftsfunktion
λ_{initial}	Startwert von $\lambda(t)$
λ_{final}	Endwert von $\lambda(t)$
η_{initial}	Startwert der globalen Lernrate
η_{final}	Endwert der globalen Lernrate
$t_{\max}$	Zeitpunkt bis zu dem $\lambda(t)$ ($\eta(t)$) auf λ_{final} (η_{final}) abgesunken ist
$\cdot_w$	ein Gewinnerneuron kennzeichnende Indizierung
$\cdot_s$	einen Zweiten kennzeichnende Indizierung
$age_{i,j}$	Alter der Kante $e_{i,j}$
$age_{\max}$	maximales Kantenalter

Wachsendes Neurales Gas (Abschnitt 3.2.3)

η_w	Lernrate des Gewinnerneurons c_w
η_n	Lernrate der Nachbarn des Gewinnerneurons
err_i	Quantisierungsfehler des Neurons c_i
δ	Reduktionsrate des Quantisierungsfehler
n_{steps}	Anzahl von Zeitschritten zwischen dem Einfügen neuer Neuronen

Motorische Karten (Abschnitt 3.3.1)

$\boldsymbol{\chi}$	Soll-Ausgabe
$\boldsymbol{u}_i$	Ausgabe-Gewichtsvektor des Neurons c_i
m	Dimension des Ausgaberaums
$\tilde{\eta}_i(t)$	zeitabhängige Lernrate für die Anpassung der Ausgabe
$\tilde{h}_i(d,t)$	zeitabhängige Nachbarschaftsfunktion für die Anpassung der Ausgabe
$\tilde{\Phi}$	die durch eine Motorische Karte realisierte Abbildung $\Phi : \mathbb{R}^n \to \mathbb{R}^m$

HCS (Kapitel 4)
Bezeichnungen die vom XCS übernommen wurden, zum Beispiel Bezeichner von Klassifizierer-Attributen oder Parametern, werden im Folgenden nicht erneut aufgeführt.

Wissensbasis (Abschnitt 4.2)

P_a	Teilpopulation der die Aktion a vertretenden Klassifizierer
$\mathcal{E}$	Kantenmenge der Population P
$\mathcal{E}_a$	Kantenmenge der Teilpopulation P_a
$\boldsymbol{w}$	Gewichtsvektor eines HCS-Klassifizierers
cl_w^a	Gewinner-Klassifizierer mit Aktion a
cl_s^a	Zweiter

$cl^a_{w(s)}$	Gewinner-Klassifizierer für den Zustand s mit Aktion a
$avgd$	Distanzabschätzung eines HCS-Klassifizierers
r	Subsumtionsradius eines HCS-Klassifizierers
$\mathcal{M}_{cl}$	Matchbereich des Klassifizierers cl

Performance-Komponente (Abschnitt 4.3)

$e_{i,j}$	Kante zwischen den Klassifizierern cl_i und cl_j
$e^a_{w,s}$	Kante zwischen cl^a_w und cl^a_s
$age_{i,j}$	Alter der Kante $e_{i,j}$
$age^a_{w,s}$	Alter der Kante $e^a_{w,s}$
$\mathcal{N}_i$	Menge der Nachbarn des Klassifizierers cl_i
N_i	Anzahl der Nachbarn des Klassifizierers cl_i
$\mathcal{N}^a_w$	Menge der Nachbarn des Gewinners cl^a_w
N^a_w	Anzahl der Nachbarn des Gewinners cl^a_w
$age_{\max}$	maximales Kantenalter
k	Anzahl ins Match-Set zu übernehmender Klassifizierer im Fall der distanzbasierten Match-Set-Bildung
M_a	Teil-Match-Set mit Aktion a
$M_a{}^*$	Teil-Match-Set mit Aktion a ohne den Gewinner cl^a_w
l_{cl}	normierter Differenzvektor zwischen einem Klassifizierer cl und dem die gleiche Aktion vertretenden Gewinner
c_{cl}	Koeffizient eines Klassifizierers cl bei der Systemvorhersagenberechnung
s_{rel}	von einem Gewinner zum aktuellen Zustand weisender Vektor
L	aus den Vektoren l_{cl} als Spalten gebildete Matrix
μ	Regularisierungsparameter

Reinforcement-Komponente (Abschnitt 4.4)

β_r	Lernrate für den Subsumtionsradius eines HCS-Klassifizierers

Discovery-Komponente (Abschnitt 4.5)

p_{initial}	initiale Vorhersage neu erzeugter Klassifizierer
$\varepsilon_{\text{initial}}$	initialer Vorhersagefehler neu erzeugter Klassifizierer
f_{initial}	initiale Fitness neu erzeugter Klassifizierer
r_{initial}	initialer Subsumtionsradius neu erzeugter Klassifizierer
p_{GA}	Wahrscheinlichkeit für die Ausführung des Genetischen Algorithmus in einem Lernschritt
τ	Anteil des Action-Sets, der im Fall der Verwendung von Turnierselektion an einem Selektionsturnier teilnimmt

inc_{mut}	Parameter für die Verteilung der Mutationsinkremente
f_s	geteilte Fitness
$Sh(\cdot,\cdot)$	Sharing-Funktion
ν_σ	Exponent der Sharing-Funktion
σ	Sharing-Radius
ι	Bewegungsrichtung für die Positionsanpassung eines Klassifizierer-Gewichtsvektors
ι'	Bewegungsrichtung für die Positionsanpassung eines Klassifizierer-Gewichtsvektors bei Verwendung berechneter Klassifizierer-Vorhersagen
$\eta(cl)$	Lernrate für die Anpassung der Gewichtsvektor-Bedingung eines Klassifizierers cl
η	Maximalwert des klassifiziererabhängigen Teils der Lernrate $\eta(cl)$
η_0	klassifiziererunabhängiger Teil der Lernrate $\eta(cl)$
$\tilde{\boldsymbol{w}}$	Hilfsattribut eines HCS-Klassifizierers zur Akkumulierung von Positionsänderungen
z	Positionsanpassungs-Zähler eines HCS-Klassifizierers

Kriterien zur Beurteilung der Leistung eines LCS (Abschnitt 6.2, Abschnitt 7.1)

p	Performanz eines Lernenden Klassifizierenden Systems; entspricht dem Anteil optimaler Aktionen an des ausgeführten
e_T	Systemfehler im Lernschritt T
s	‚Populationsgröße'; entspricht dem Quotienten aus der Anzahl von Makroklassifizierern in einer Population und der maximalen Größe $N_{\max}$ der Population
MAE	mittlerer absoluter Approximationsfehler
$\overline{\mathrm{MAE}}$	der über die Einzelläufe eines Experimentes gemittelte MAE

Anhang C

Verwendete Parametersätze

In den Kapiteln 6 bis 8, in denen die im Rahmen dieser Arbeit mit dem HCS durchgeführten Experimente besprochen wurden, wurde zugunsten eines besseren Leseflusses auf eine Angabe der jeweils verwendeten Parametersätze verzichtet. Dies wird im Folgenden nachgeholt. Neben den für das HCS verwendeten Parametersätze werden auch die in den Vergleichstests mit dem XCS benutzten angegeben.

C.1 Klassifikation

In den Experimenten zur Klassifikation kamen neben dem HCS zwei Varianten des XCS zum Einsatz, die jeweils gleiche Parametersätze verwendeten. Bei beiden war nur der GA-Subsumtions-Operator aktiv, Action-Set-Subsumtion wurde nicht eingesetzt.
Die Genetischen Algorithmen aller drei Systeme verwendeten Rouletterad-Selektion, einen Zwei-Punkt-Crossing-Over-Operator sowie Standard-Punktmutation mit gleichverteilten Inkrementen. Ferner wurden von allen Systemen skalare Klassifizierervorhersagen benutzt, sodass in diesem Abschnitt keine für die Anpassung der Gewichte der Vorhersagenberechnung benötigten Parameter anzugeben sind.
Im Fall der betrachteten Multiplexer-Probleme erfolgte die explorative Aktionswahl beim XCS rein zufällig, in allen anderen Experimenten aber unter Bevorzugung der jeweils als am besten angesehenen Aktion.
Die übrigen verwendeten Parameter können den nun folgenden Tabellen entnommen werden. Die Tabellen C.1 und C.2 enthalten die von den XCS-Varianten respektive dem HCS im Fall der Multiplexer-Lernumgebungen verwendeten Parameter. Die Parametersätze, mit denen die Systeme in den Checkerboard-Lernumgebungen eingesetzt wurden, sind in den Tabellen C.3 bis C.4 angegeben

Allg. Parameter		Initialwerte		Genauigkeit		GA-Parameter	
$N_{\max}$	800	p_{initial}	10.00	α	0.1	Θ_{GA}	12
p_{explr}	-/-	$\varepsilon_{\text{initial}}$	0.00	ε_0	10.0	p_{GA}	1.0
γ	-/-	f_{initial}	0.01	ν	5	τ	-/-
Vorhersagen		Covering		Lösch-Parameter		p_{cross}	0.8
ξ_0	-/-	Θ_{MNA}	2	δ	0.1	p_{mut}	0.04
β_0	-/-	$p_{\#}$	-/-	Θ_{DEL}	20	inc_{mut}	0.1
Lernrate		s_{cov}	1	Subsumtion			
β	0.20			Θ_{SUB}	20		

Tabelle C.1: Parameter des XCS für die Multiplexer-Lernumgebungen

Allg. Parameter		Initialwerte		Genauigkeit		GA-Parameter	
$N_{\max}$	2000	p_{initial}	10.00	α	1.0	Θ_{GA}	25
p_{explr}	0.5	$\varepsilon_{\text{initial}}$	0.00	ε_0	200	p_{GA}	0.5
γ	-/-	f_{initial}	0.01	ν	1	τ	-/-
Vorhersagen		r_{initial}	0.30	ε_p	200	p_{cross}	0.8
ξ_0	-/-	Netzlernen		Lösch-Parameter		p_{mut}	0.04
β_0	-/-	η	0.20	δ	0.1	inc_{mut}	0.1
Lernraten		η_0	0.01	Θ_{DEL}	20	Fitness-Sharing	
β	0.20	$age_{\max}$	300	Regularisierung		ν_σ	-/-
β_r	0.0001			μ	0.1	σ	-/-

Tabelle C.2: Parameter des HCS für die Multiplexer-Lernumgebungen

Allg. Parameter		Initialwerte		Genauigkeit		GA-Parameter	
$N_{\max}$	2000	p_{initial}	10.00	α	0.1	Θ_{GA}	12
p_{explr}	0.5	$\varepsilon_{\text{initial}}$	0.00	ε_0	10.0	p_{GA}	1.0
γ	-/-	f_{initial}	0.01	ν	5	τ	-/-
Vorhersagen		Covering		Lösch-Parameter		p_{cross}	0.8
ξ_0	-/-	Θ_{MNA}	2	δ	0.1	p_{mut}	0.04
β_0	-/-	$p_\#$	-/-	Θ_{DEL}	20	inc_{mut}	0.1
Lernrate		s_{cov}	1	Subsumtion			
β	0.20			Θ_{SUB}	20		

Tabelle C.3: Parameter des XCS für die Checkerboard-Lernumgebungen

Allg. Parameter		Initialwerte		Genauigkeit		GA-Parameter	
$N_{\max}$	2000	p_{initial}	10.00	α	1.0	Θ_{GA}	12
p_{explr}	0.5	$\varepsilon_{\text{initial}}$	0.00	ε_0	200	p_{GA}	0.5
γ	-/-	f_{initial}	0.01	ν	1	τ	-/-
Vorhersagen		r_{initial}	0.15	ε_p	200	p_{cross}	0.8
ξ_0	-/-	Netzlernen		Lösch-Parameter		p_{mut}	0.04
β_0	-/-	η	0.120	δ	0.1	inc_{mut}	0.1
Lernraten		η_0	0.005	Θ_{DEL}	20	Fitness-Sharing	
β	0.20	$age_{\max}$	150	Regularisierung		ν_σ	1
β_r	0.01			μ	0.1	σ	0.25/0.40

Tabelle C.4: Parameter des HCS für die Checkerboard-Lernumgebungen

In Tabelle C.4 sind zwei Werte für den Sharing-Radius σ angegeben. Der erste wurde im Fall des zweidimensionalen Checkerboards mit fünf Unterteilungen, der zweite beim dreidimensionalen Checkerboard mit drei Unterteilungen verwendet.

C.2 Funktionsapproximation

Bei der Funktionsapproximation wurde die Leistung des HCS mit der des XCS mit Ordered-Bound-Bedingungen verglichen. Beide Systeme verwendeten Klassifizierer, deren Vorhersagen als lineare Funktion des Lernumgebungszustandes berechnet wurden.

Die Anpassung der dabei verwendeten Gewichte erfolgte mittels der in Abschnitt 2.4.8 beziehungsweise Abschnitt 4.2.1 erläuterten, auf der Methode der kleinsten Quadrate basierenden Methode. Bei beiden Systemen wurde die auszuführende Aktion im Explorationsmodus unter Bevorzugung der besten Aktion gewählt. Beim XCS kam, wie schon bei der Klassifikation der GA-Subsumtions-Operator, nicht aber der Action-Set-Subsumtions-Operator zum Einsatz.
Der Genetische Algorithmus des XCS setzte bei der Approximation zweidimensionaler Funktionen Turnier-Selektion ein, in allen anderen Fällen wurde Rouletterad-Selektion verwendet. Als Crossing-Over-Operator setzte das XCS im Falle eindimensionaler Funktionen Zwei-Punkt-Crossing-Over, im Falle zweidimensionaler Funktionen Uniformes Crossing-Over ein. Das HCS benutzte stets Intermediäres Crossing-Over. Außer bei der Approximation zweidimensionaler Funktionen, bei der das XCS eine aus [Butz 2005] übernommene Form der Mutation, die sogenannten Relative Real Valued Mutation benutzte, fand immer die Standard-Punktmutation mit gleichverteilten Inkrementen Anwendung.
Die im Fall der Approximation eindimensionaler Funktionen von XCS und HCS benutzten Parameter können den Tabellen C.5 respektive C.6 entnommen werden. Bei der Approximation zweidimensionaler Funktionen wurde vom XCS der in Tabelle C.7 angegebene Parametersatz verwendet. Das HCS benutzte bei der Approximation der Funktionen f_1, f_2 und f_{roof} die in Tabelle C.8, bei der Approximation der übrigen zweidimensionalen Funktionen die in Tabelle C.9 angegebenen Parameter

Allg. Parameter		Initialwerte		Genauigkeit		GA-Parameter	
$N_{\max}$	1000	p_{initial}	-/-	α	0.1	Θ_{GA}	50
p_{explr}	0.5	$\varepsilon_{\text{initial}}$	0.00	ε_0	0.005	p_{GA}	1.0
γ	-/-	f_{initial}	0.01	ν	5	τ	-/-
Vorhersagen		Covering		Lösch-Parameter		p_{cross}	0.8
ξ_0	1.0	Θ_{MNA}	1	δ	0.1	p_{mut}	0.04
β_0	0.2	$p_{\#}$	-/-	Θ_{DEL}	50	inc_{mut}	0.2
Lernrate		s_{cov}	0.1	Subsumtion			
β	0.2			Θ_{SUB}	50		

Tabelle C.5: Parameter des XCS für die Approximation eindimensionaler Funktionen

Allg. Parameter		Initialwerte		Genauigkeit		GA-Parameter	
$N_{\max}$	1000	p_{initial}	-/-	α	1.0	Θ_{GA}	25
p_{explr}	0.5	$\varepsilon_{\text{initial}}$	0.00	ε_0	0.005	p_{GA}	0.15
γ	-/-	f_{initial}	0.01	ν	1	τ	-/-
Vorhersagen		r_{initial}	0.0025	ε_p	0.02	p_{cross}	1.0
ξ_0	1.0	Netzlernen		Lösch-Parameter		p_{mut}	0.05
β_0	0.5	η	0.02	δ	0.1	inc_{mut}	0.5
Lernraten		η_0	0.0	Θ_{DEL}	50	Fitness-Sharing	
β	0.20	$age_{\max}$	50	Regularisierung		ν_σ	1
β_r	0.01			μ	0.1	σ	0.005

Tabelle C.6: Parameter des HCS für die Approximation eindimensionaler Funktionen

Allg. Parameter		Initialwerte		Genauigkeit		GA-Parameter	
$N_{\max}$	6400	p_{initial}	-/-	α	1	Θ_{GA}	50
p_{explr}	0.5	$\varepsilon_{\text{initial}}$	0.00	ε_0	0.002	p_{GA}	1.0
γ	-/-	f_{initial}	0.01	ν	5	τ	0.4
Vorhersagen		Covering		Lösch-Parameter		p_{cross}	1.0
ξ_0	1.0	Θ_{MNA}	1	δ	0.1	p_{mut}	0.05
β_0	0.5	$p_{\#}$	-/-	Θ_{DEL}	20	inc_{mut}	0.1
Lernrate		s_{cov}	1	Subsumtion			
β	0.5			Θ_{SUB}	20		

Tabelle C.7: Parameter des XCS für die Approximation zweidimensionaler Funktionen

Allg. Parameter		Initialwerte		Genauigkeit		GA-Parameter	
$N_{\max}$	6400	p_{initial}	-/-	α	1.0	Θ_{GA}	12
p_{explr}	0.5	$\varepsilon_{\text{initial}}$	0.00	ε_0	0.002	p_{GA}	0.20
γ	-/-	f_{initial}	0.01	ν	1	τ	-/-
Vorhersagen		r_{initial}	0.005	ε_p	0.02	p_{cross}	1.0
ξ_0	1.0	Netzlernen		Lösch-Parameter		p_{mut}	0.04
β_0	0.8	η	0.02	δ	0.1	inc_{mut}	0.5
Lernraten		η_0	0.0	Θ_{DEL}	20	Fitness-Sharing	
β	0.50	$age_{\max}$	50	Regularisierung		ν_σ	1
β_r	0.01			μ	0.01	σ	0.025

Tabelle C.8: Parameter des HCS für die Approximation der zweidimensionalen Funktionen f_1, f_2 und f_{roof}

Allg. Parameter		Initialwerte		Genauigkeit		GA-Parameter	
$N_{\max}$	6400	p_{initial}	-/-	α	1.0	Θ_{GA}	12
p_{explr}	0.5	$\varepsilon_{\text{initial}}$	0.00	ε_0	0.002	p_{GA}	0.20
γ	-/-	f_{initial}	0.01	ν	1	τ	-/-
Vorhersagen		r_{initial}	0.025	ε_p	0.02	p_{cross}	1.0
ξ_0	1.0	Netzlernen		Lösch-Parameter		p_{mut}	0.04
β_0	0.8	η	0.02	δ	0.1	inc_{mut}	0.5
Lernraten		η_0	0.0	Θ_{DEL}	20	Fitness-Sharing	
β	0.50	$age_{\max}$	50	Regularisierung		ν_σ	1
β_r	0.01			μ	0.01	σ	0.05

Tabelle C.9: Parameter des HCS für die Approximation der zweidimensionalen Funktionen f_3, f_{e3} und f_{e3r}

C.3 Aktionenlernen

Beim Aktionenlernen wurde das HCS wiederum mit dem XCS mit Ordered-Bound-Bedingungen verglichen. Beide Systeme wählten auch hier die im Explorationsmodus auszuführenden Aktionen unter Bevorzugung der als am besten angesehenen Aktion.

Ebenso wurden wieder Klassifizierer mit berechneten Vorhersagen verwendet, die Anpassung der Gewichte für die Vorhersagenberechnung erfolgte basierend auf der Methode der kleinsten Quadrate. Auch hier nutzte das XCS nur den GA-Subsumtions-Operator. Der Genetische Algorithmus verwendete bei allen Experimenten die Roulette-Rad-Selektionsmethode sowie Punktmutation mit gleichverteilten Inkrementen. Beim XCS kam Zwei-Punkt-Crossing-Over, beim HCS Uniformes Crossing-Over zum Einsatz.
In den Tabellen C.10 und C.11 sind die beim Lernen in der Korridor-Lernumgebung verwendeten Parameter angegeben, die für die Experimente mit der Empty-Room- sowie der Puddles-Lernumgebung benutzten Parameter sind den Tabellen C.12 und C.13 zu entnehmen.

Allg. Parameter		Initialwerte		Genauigkeit		GA-Parameter	
$N_{\max}$	400	p_{initial}	-/-	α	0.1	Θ_{GA}	50
p_{explr}	0.5	$\varepsilon_{\text{initial}}$	0.00	ε_0	0.01	p_{GA}	1.0
γ	0.90/0.95/0.99	f_{initial}	0.01	ν	5	τ	-/-
Vorhersagen		Covering		Lösch-Parameter		p_{cross}	0.8
ξ_0	1.0	Θ_{MNA}	2	δ	0.1	p_{mut}	0.04
β_0	0.3	$p_{\#}$	-/-	Θ_{DEL}	50	inc_{mut}	0.5
Lernrate		s_{cov}	0.25	Subsumtion			
β	0.2			Θ_{SUB}	50		

Tabelle C.10: Parameter des XCS für die Korridor-Lernumgebung

Allg. Parameter		Initialwerte		Genauigkeit		GA-Parameter	
$N_{\max}$	400	p_{initial}	-/-	α	1.0	Θ_{GA}	50
p_{explr}	0.5	$\varepsilon_{\text{initial}}$	0.00	ε_0	0.01	p_{GA}	0.5
γ	0.90/0.95/0.99	f_{initial}	0.01	ν	1	τ	-/-
Vorhersagen		r_{initial}	0.1	ε_p	0.25	p_{cross}	0.8
ξ_0	1.0	Netzlernen		Lösch-Parameter		p_{mut}	0.04
β_0	0.5	η	0.05	δ	0.1	inc_{mut}	0.1
Lernraten		η_0	0.0	Θ_{DEL}	50	Fitness-Sharing	
β	0.20	$age_{\max}$	100	Regularisierung		ν_σ	1
β_r	0.001			μ	0.1	σ	0.1

Tabelle C.11: Parameter des HCS für die Korridor-Lernumgebung

Allg. Parameter		Initialwerte		Genauigkeit		GA-Parameter	
$N_{\max}$	10000	p_{initial}	-/-	α	0.1	Θ_{GA}	50
p_{explr}	0.5	$\varepsilon_{\text{initial}}$	0.00	ε_0	0.005	p_{GA}	1.0
γ	0.90/0.95/0.99	f_{initial}	0.01	ν	5	τ	-/-
Vorhersagen		Covering		Lösch-Parameter		p_{cross}	0.8
ξ_0	1.0	Θ_{MNA}	4	δ	0.1	p_{mut}	0.04
β_0	0.5	$p_{\#}$	-/-	Θ_{DEL}	50	inc_{mut}	0.5
Lernrate		s_{cov}	0.25	Subsumtion			
β	0.2			Θ_{SUB}	50		

Tabelle C.12: Parameter des XCS für die Empty-Room- und die Puddles-Lernumgebung

Allg. Parameter		Initialwerte		Genauigkeit		GA-Parameter	
$N_{\max}$	10000	p_{initial}	-/-	α	1.0	Θ_{GA}	25
p_{explr}	0.5	$\varepsilon_{\text{initial}}$	0.00	ε_0	0.01	p_{GA}	0.5
γ	0.90/0.95/0.99	f_{initial}	0.01	ν	1	τ	-/-
Vorhersagen		r_{initial}	0.05	ε_p	0.3	p_{cross}	0.8
ξ_0	1.0	Netzlernen		Lösch-Parameter		p_{mut}	0.04
β_0	0.5	η	0.075	δ	0.1	inc_{mut}	0.1
Lernraten		η_0	0.002	Θ_{DEL}	50	Fitness-Sharing	
β	0.20	$age_{\max}$	100	Regularisierung		ν_σ	1
β_r	0.001			μ	0.1	σ	0.2

Tabelle C.13: Parameter des HCS für die Empty-Room- und die Puddles-Lernumgebung

In der Korridor- und der Empty-Room-Lernumgebung wurden mit dem XCS und dem HCS jeweils mehrere Experimente durchgeführt, bei denen der Diskontierungsfaktor γ variiert wurde. Dementsprechend sind in den Tabellen C.10 bis C.13 mehrere Werte dieses Parameters angegeben.

Anhang D

Inhalt der zur Arbeit gehörenden DVD-ROM

Sämtliche in der vorliegenden Arbeit beschriebenen Experimente mit dem HCS wurden mit der in Kapitel 5 beschriebenen Software durchgeführt, die im Rahmen dieser Arbeit implementiert wurde. Um interessierten Lesern eine Reproduktion dieser Experimente sowie die Durchführung eigener Versuche zu ermöglichen, ist es sinnvoll, diese Software allgemein verfügbar zu machen.

D.1 Bezugsmöglichkeit

Den zur Begutachtung eingereichten Exemplaren dieser Arbeit liegt jeweils eine DVD-ROM bei, auf der die Software inklusive Quellcode sowie einige zusätzliche Informationen zu finden sind.
Um Lesern anderer Exemplare ebenfalls die Möglichkeit zu geben, auf diese Ressourcen zuzugreifen, ist der Inhalt dieser DVD-ROM als ISO-Image unter der URL

```
http://www.informatik.uni-mainz.de/~hillet/dissertation/diss_dvd.iso
```

abgelegt. Das Brennen[1] dieser Image-Datei auf eine DVD-ROM sollte einen Datenträger ergeben, der inhaltlich identisch mit den den Abgabeexemplaren beiliegenden ist.

D.2 Inhalt

Genauere Angaben zum Inhalt der DVD-ROM finden sich umseitig. In der dort abgedruckten Tabelle sind die wichtigsten (Unter-) Verzeichnisse der DVD-ROM mit einer Beschreibung ihres Inhaltes aufgelistet. Bei Verwendung der DVD-ROM unter einem Windows-Betriebssystem steht aber auch ein interaktives Menü zur Verfügung, das den Zugriff auf alle wesentlichen Inhalte der DVD-ROM ermöglicht. Dieses Menü sollte beim Einlegen der DVD-ROM automatisch gestartet werden. Wurde die Autorun-Funktionalität der verwendeten Windows-Installation deaktiviert, kann das Menü über die im Stammverzeichnis der DVD-ROM zu findende Datei ‚AutoRun.exe' gestartet werden.

[1] Hierzu kann zum Beispiel das Freeware-Programm *CDBurnerXP* verwendet werden, das unter der URL `http://www.cdburnerxp.se/` zu finden ist.

Verzeichnis	**Inhalt**
\ [Wurzel]	Dateien, die unter Microsoft Windows ein automatisches Anlaufen des interaktiven Inhaltsverzeichnisses der DVD realisieren
`\Ausarbeitung`	Volltext dieser Arbeit im PDF-Format
`\Quellcode`	Microsoft Visual Studio 2008 Projektmappe und Sandcastle Help File Builder Projektdatei
`\Quellcode\src`	Quellcode der HCS-Simulations-Software
`\Quellcode\doc`	Dokumentation des Quellcodes im CHM-Format (kompilierte HTML-Hilfe)
`\Quellcode\ThirdParty`	Von der HCS-Simulations-Software verwendete Drittanbieter-Komponenten (sämtlich Open Source)
`\Setup`	Setup-Programm für die HCS-Simulations-Software. Installiert die Simulationssoftware (Windows XP, Vista, 7)
`\Setup\DotNetFX35SP1`	Installationspaket für .NET-Framework 3.5 mit Servicepack 1 (wird – sofern nicht vorhanden – vom Setup der HCS-Simulations-Software installiert)
`\Setup\WindowsInstaller3_1`	Installationspaket für Windows-Installer 3.1 (wird – sofern nicht vorhanden – vom Setup der HCS-Simulations-Software installiert)
`\Parametersätze`	Parametersätze aller in dieser Arbeit angesprochenen Experimente im PDF- und XML-Format

Tabelle D.1: Inhalt der zur vorliegenden Arbeit gehörenden DVD-ROM

Literaturverzeichnis

[Ball 1994] BALL, N.: Organizing an Animat's Behavioural Repertoires Using Kohonen Feature Maps. In: CLIFF, D. (Hrsg.) ; HUSBANDS, P. (Hrsg.) ; MEYER, J. (Hrsg.) ; WILSON, S. (Hrsg.): *From Animals to Animats 3*. Cambridge, MA : MIT Press, 1994.

[Barreca u. Buttazzo 1993] BARRECA, D.M. ; BUTTAZZO, G.C.: Learning Control Tasks by Failure Signals. In: *Proceedings of the 1993 World Congress on Neural Networks* Bd. 4, 1993, S. 518–522.

[Booker 1982] BOOKER, L.B.: *Intelligent Behaviour as an Adaptation to the Task Environment*, University of Michigan, Diss., 1982.

[Boyan u. Moore 1995] In: BOYAN, J.A. ; MOORE, A.W.: *Generalization in reinforcement learning: Safely approximating the value function*. Cambridge, MA : MIT Press, 1995, S. 369 – 376.

[Brause 1995] BRAUSE, R.: *Neuronale Netze: Eine Einführung in die Neuroinformatik*. 2., überarbeitete und erweiterte Auflage. Stuttgart : B.G. Teubner, 1995 (Leitfäden der Informatik).

[Bull 2004] BULL, L. (Hrsg.): *Applications of Learning Classifier Systems*. Berlin : Springer, 2004 (Studies in Fuzziness and Soft Computing).

[Bull u. Hurst 2002] BULL, L. ; HURST, J.: ZCS Redux. In: *Evolutionary Computation* 10 (2002), Nr. 2, S. 185–205.

[Bull u. Kovacs 2005] BULL, L. ; KOVACS, T.: Foundations of Learning Classifier Systems: An Introduction. In: BULL, L. (Hrsg.) ; KOVACS, T. (Hrsg.): *Foundations of Learning Classifier Systems*. Berlin : Springer, 2005 (Studies in Fuzziness and Soft Computing), S. 1–17.

[Butz 1999] BUTZ, M.V.: An Implementation of the XCS Classifier System in C / Illinois Genetic Algorithm Laboratory, University of Illinois at Urbana-Champaign. 1999 (IlliGAL99021). – Forschungsbericht. –
Der Quellcode ist verfügbar unter: `http://www.illigal.uiuc.edu/pub/src/XCS-C1.1.tar.Z`, Abruf: 10.10.2009.

[Butz 2005] BUTZ, M.V.: Kernel-Based, Ellipsoidal Conditions in the Real-valued XCS Classifier System. In: *GECCO '05: Proceedings of the 2005 Conference on Genetic and Evolutionary Computation*. New York : ACM Press, 2005, S. 1835–1842.

[Butz 2006] BUTZ, M.V.: *Rule-Based Evolutionary Online Learning Systems: A Principled Approach to LCS Analysis and Design*. Berlin : Springer, 2006 (Studies in Fuzziness and Soft Computing 191).

[Butz u. a. 2006] BUTZ, M.V. ; LANZI, P.L. ; WILSON, S.W.: Hyper-Ellipsoidal Conditions in XCS: Rotation, Linear Approximation, and Solution Structure. In: *GECCO '06: Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*. New York : ACM Press, 2006, S. 1457–1464.

[Butz u. a. 2002] BUTZ, M.V. ; SASTRY, K. ; GOLDBERG, D.E.: Tournamentselction in XCS / Illinois Genetic Algorithm Laboratory, University of Illinois at Urbana-Champaign. 2002 (IlliGAL2002020). – Forschungsbericht.

[Butz u. Wilson 2000] BUTZ, M.V. ; WILSON, S.W.: An Algorithmic Description of XCS. In: **[Lanzi u. a. 2000a]**, S. 253–272.

[Campbell 1997] CAMPBELL, N.A. ; MARKL, J. (Hrsg.): *Biologie*. Heidelberg : Spektrum Verlag, 1997.

[Cliff u. Ross 1994] CLIFF, D. ; ROSS, S.: Adding Temporary Memory to ZCS. In: *Adaptive Behaviour* 3 (1994), Nr. 2, S. 101–150.

[Darwin 1859] DARWIN, Ch.: *On the Origin of Species by Means of Natural Selection or the Preservation of Favored Races in the Struggle for Life*. London : John Murray, 1859.

[Dayan 1992] DAYAN, P.: The Convergence of TD(λ) for General λ. In: *Machine Learning* 8 (1992), S. 341–362.

[De Jong u. a. 1993] DE JONG, K.A. ; SPEARS, W.M. ; GORDON, D.F.: Using Genetic Algorithms for Concept Learning. In: *Machine Learning* 13 (1993), S. 161–188.

[Deb 2001] DEB, K.: *Multi-Objective Optimization using Evolutionary Algorithms*. New York : Wiley & Sons, 2001 (Wiley Interscience Series in Systems and Optimization).

[Decraene u. a. 2007] In: DECRAENE, J. ; MITCHELL, G.G. ; MCMULLIN, B. ; KELLY, C.: *The Holland Broadcast Language and the Modeling of Biochemical Networks*. Berlin : Springer, 2007 (LNCS 4445), S. 361–370.

[DotGNU] *DotGNU Project - GNU Freedom for the Net*. `http://www.gnu.org/software/dotgnu/`, Abruf: 10.10.2009. – Projekt zur Portierung von .NET.

[Eiben u. Smith 2003] EIBEN, A.E. ; SMITH, J.E.: *Introduction to Evolutionary Computation*. New York : Springer, 2003 (Natural Computing Series).

[Fogel u. a. 1965] FOGEL, L.J. ; OWENS, A.J. ; WALSH, M.J.: Artificial Intelligence through a Simulation of Evolution. In: MAXFIELD, M. (Hrsg.) ; CALLAHAN, A. (Hrsg.) ; FOGEL, L.J. (Hrsg.): *Biophysics and Cybernetics Systems: Proc. of the 2nd Cybernetic Sciences Symposium*. Washington DC : Spartan Books, 1965, S. 131–155.

[Fritzke 1995a] FRITZKE, B.: A Growing Neural Gas Network Learns Topologies. In: G.TESAURO (Hrsg.) ; TOURETZKY, D.S. (Hrsg.) ; LEEN, T.K. (Hrsg.): *Advances in Neural Information Processing Systems 7*. Cambridge, MA : MIT Press, 1995.

[Fritzke 1995b] FRITZKE, B.: Growing Grid - a Self-Organizing Network With Constant Neighborhood Range and Adaptation Strength. In: *Neural Processing Letters* 2 (1995), Nr. 5, S. 9–13.

[Fritzke 1995c] FRITZKE, B.: Incremental Learning of Local Linear Mappings. In: F.FOGELMAN (Hrsg.) ; GALLINARI, P. (Hrsg.): *Proceedings of ICANN'95, International Conference on Artificial Neural Networks*. Paris : EC2 & Cie, 1995, S. 217–222.

[Fritzke u. Loos 1998] FRITZKE, B. ; LOOS, H.S.: *DemoGNG (Version 1.5)*. 1998. – JAVA-Applet unter: `http://www.neuroinformatik.ruhr-uni-bochum.de/ini/VDM/research/gsn/DemoGNG/GNG.html`, Abruf: 10.10.2009.

[Gerdes u. a. 2004] GERDES, I. ; KLAWONN, F. ; R.KRUSE: *Evolutionäre Algorithmen*. Wiesbaden : Vieweg, 2004 (Computational Intelligence).

[Goldberg 1989] GOLDBERG, D.E.: *Genetic Algorithms in Search, Optimization and Machine Learning*. Reading, MA : Addison Wesley, 1989.

[Hanke-Bourgeois 2002] HANKE-BOURGEOIS, M.: *Grundlagen der Numerischen Mathematik und des Wissenschaftlichen Rechnens* . Stuttgart : B.G. Teubner, 2002.

[Haykin 1999] HAYKIN, S.: *Neural Networks: A Comprehensive Approach*. 2nd Edition. Prentice-Hall, 1999.

[Hebb 1949] HEBB, D.O.: *The Organization of Behaviour*. New York : John Wiley, 1949.

[Herrmann u. Der 1995] HERRMANN, M. ; DER, R.: Efficient Q-Learning by Division of Labour. In: *Proceedings of ICANN'95, International Conference on Artificial Neural Networks* Bd. 2. Paris : EC2 & Cie, 1995, S. 129–134.

[Holland 1986] HOLLAND, J.H.: Escaping brittleness: The Possibilities of General-Purpose Learning Algorithms Applied to Parallel Rule-based Systems. In: MICHALSKI, R.S (Hrsg.) ; CARBONELL, J.G. (Hrsg.) ; MITCHELL, T.M. (Hrsg.): *Machine Learning, an Artificial Intelligence Approach, Volume II*. Los Altos, CA : Morgan Kaufmann, 1986, Kapitel 20.

[Holland 1992] HOLLAND, J.H.: *Adaptation in Natural and Artificial Systems*. First MIT Press edition. Cambridge, MA : MIT Press, 1992. – First edition 1975 (The University of Michigan).

[Holland u. a. 2000] HOLLAND, J.H. ; BOOKER, L.B. ; COLOMBETTI, M. ; DORIGO, M. ; GOLDBERG, D.E. ; R.L. RIOLO, S. F. ; SMITH, R.E. ; P.L.LANZI ; STOLZMANN, W. ; WILSON, S.W.: What Is A Learning Classifier System? In: **[Lanzi u. a. 2000b]**, S. 3–32.

[Holland u. Reitman 1978] HOLLAND, J.H. ; REITMAN, J.S.: Cognitive Systems Based on Adaptive Algorithms. In: WATERMAN, D.A. (Hrsg.) ; HAYES-ROTH, F. (Hrsg.): *Pattern-directed Inference Systems*. Orlando, FL : Academic Press, 1978. – Nachdruck in: FOGEL, D.B.: *Evolutionary Computation. The Fossil Record*. Piscataway, NJ: IEEE Press, 1998.

[Holmes u. a. 2002] HOLMES, J.H. ; LANZI, P.L. ; STOLZMANN, W. ; WILSON, S.W.: Learning Classifier Systems: New Models, Successful Applications. In: *Information Processsing Letters* 82 (2002), Nr. 1, S. 23–30.

[Homunculus] *Homunculus-de.svg*. `http://upload.wikimedia.org/wikipedia/commons/5/51/Homunculus-de.svg`, Abruf: 12.04.2007. – Nach: LOVE, R.J. ; WEBB, W.G.: *Neurology for the Speech-Language Pathologist*. Oxford: Butterworth-Heinemann, 1992. Veränderte elektronische Fassung.

[IBM] *IBM Research — Deep Blue — Overview*. `http://www.research.ibm.com/deepblue`, Abruf: 9.10.2009.

[J.Bacardit u. a. 2008] J.BACARDIT ; BERNADÓ-MANSILLA, E. ; BUTZ, M. V.: Learning Classifier Systems: Looking Back and Glimpsing Ahead. In: GOEBEL, R. (Hrsg.) ; J.SIEKMANN (Hrsg.) ; W.WAHLSTER (Hrsg.): *Learning Classifier Systems*. Berlin : Springer, 2008 (LNAI 4998), S. 1–21.

[J.Bacardit u. a. 2004] J.BACARDIT ; GOLDBERG, D.E. ; BUTZ, M.V. ; LLORÀ, X. ; GARRELL, J.M.: Speeding-up Pittsburgh learning classifier systems: Modelling time and accuracy. In: YAO, X. (Hrsg.) ; BURKE, E. (Hrsg.) ; LOZANO, J.A. (Hrsg.) ; SMITH, J. (Hrsg.) ; MERELO-GUERVÓS, J.J. (Hrsg.) ; BULLINARIA, J.A. (Hrsg.) ; ROWE, J. (Hrsg.) ; P.TINO (Hrsg.) ; KABAN, A. (Hrsg.) ; SCHWEFEL, H.P. (Hrsg.): *Parallel Problem Solving from Nature - PPSN VIII : 8th International Conference, Birmingham, UK, September 18-22, 2004. Proceedings* . Berlin : Springer, 2004, S. 1021–1031.

[Kaas u. a. 1979] KAAS, J.H. ; NELSON, R.J. ; SUR, M. ; C.S.LIN ; MERZNICH, M.M.: Multiple Representations of the Body within the Primary Somatosensory Cortex of Primates. In: *Science* 204 (1979), S. 521–523.

[Kaelbling u. a. 1996] KAELBLING, L.P. ; LITTMAN, M.L. ; MOORE, A.W.: Reinforcement Learning: A Survey. In: *Journal of Artificial Intelligence Research* 4 (1996), S. 237–285.

[Kandel u. a. 1996] KANDEL, W.R. (Hrsg.) ; SCHWARTZ, J.H. (Hrsg.) ; JESSEL, Th.M. (Hrsg.): *Neurowissenschaften: Eine Einführung*. Heidelberg : Spektrum akademischer Verlag, 1996.

[Kelso u. a. 1986] KELSO, S.R. ; GANONG, A.H. ; BROWN, Th.H.: Hebbian Synapses in Hippocampus. In: *Proceedings of the National Academy of Sciences of the United States of America* 83 (1986), S. 5326–5330.

[Kharbat u. a. 2005] KHARBAT, F. ; BULL, L. ; ODEH, M.: Revisiting Genetic Selection in the XCS Learning Classifier System. In: *Proceedings of the IEEE Congress on Evolutionary Computation – CEC-2005* Bd. 3. Piscataway, NJ : IEEE, 2005, S. 2061.

[Kohonen 1982] KOHONEN, T.: Self-Organized Formation of Topologically Correct Feature Maps. In: *Biological Cybernetics* 43 (1982), S. 59–69. – Nachdruck in: ANDERSON, J.A. (Hrsg.) ; ROSENFELD, E. (Hrsg.): *Neuro Computing: Foundations of Research*. Cambridge, MA: MIT Press, 1988.

[Kohonen 2001] KOHONEN, T.: *Self Organizing Maps*. Third Edition. Berlin : Springer, 2001 (Springer Series in Information Sciences).

[Kovacs 1997] KOVACS, T.: XCS Classifier System Reliably Evolves Accurate, Complete and Minimal Representations for Boolean Functions. In: CHAWDHRY, P.K. (Hrsg.) ; ROY, R. (Hrsg.) ; PANT, R.K. (Hrsg.): *Soft Computing in Engineering Design and Manufactoring*. Springer, 1997, S. 59–68.

[Kovacs 1999] KOVACS, T.: Deletion Schemes For Classifier Systems. In: BANZHAF, W. (Hrsg.) ; J, Daida (Hrsg.) ; EIBEN, A.E. (Hrsg.) ; GARZON, M.H. (Hrsg.) ; HONAVAR, V. (Hrsg.) ; JAKIELA, M. (Hrsg.) ; SMITH, R.E. (Hrsg.): *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-99)*. San Francisco, CA : Morgan Kaufmann, 1999, S. 329–336.

[Kovacs 2000] KOVACS, T.: Strength or Accuracy? Fitness Calculation in Learning Classifier Systems. In: **[Lanzi u. a. 2000b]**, S. 143–160.

[Kovacs 2001] KOVACS, T.: Towards a Theory of Strong Overgeneral Classifiers. In: MARTIN, W. (Hrsg.) ; SPEARS, W. (Hrsg.): *Foundations of Genetic Algorithms 6*. San Francisco, CA : Morgan Kaufmann, 2001, S. 165–184.

[Kovacs 2002] KOVACS, T.: Two Views of Classifier Systems. In: LANZI, P.L. (Hrsg.) ; STOLZMANN, W. (Hrsg.) ; WILSON, S.W. (Hrsg.): *Advances in Learning Classifier Systems: Fourth International Workshop (IWLCS 2001)*. Berlin : Springer, 2002 (LNAI 2321), S. 74–87.

[Koza 1992] KOZA, J.R.: *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge, MA : MIT Press, 1992.

[Kurzweil 1990] KURZWEIL, R.: *The Age of Intelligent Machines*. Cambridge, MA : MIT Press, 1990.

[Lanzi 1997] LANZI, P.L.: A Study of the Generalization Capabilities of XCS. In: BAECK, T. (Hrsg.): *Proceedings of the Seventh International Conference on Genetic Algorithms*. San Francisco, CA : Morgan Kaufmann, 1997, S. 418–425.

[Lanzi 1999] LANZI, P.L.: An Analysis of Generalization in the XCS Classifier System. In: *Evolutionary Computation* 7 (1999), Nr. 2, S. 125–149.

[Lanzi u. D.Loiacono 2007] LANZI, P.L. ; D.LOIACONO: Classifier systems that compute action mappings / Illinois Genetic Algorithm Laboratory, University of Illinois at Urbana-Champaign. 2007 (IlliGAL2007002). – Forschungsbericht.

[Lanzi u. a. 2008] In: LANZI, P.L. ; D.LOIACONO ; ZANINI, M.: *Evolving Classifiers Ensembles with Heterogeneous Predictors*. Berlin : Springer, 2008 (LNCS 4998), S. 218–234.

[Lanzi u. a. 2005a] LANZI, P.L. ; LOIACONO, D. ; WILSON, S.W. ; GOLDBERG, D.E.: Extending XCSF beyond linear approximation. In: *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*. New York : ACM, 2005, S. 1827–1834.

[Lanzi u. a. 2005b] LANZI, P.L. ; LOIACONO, D. ; WILSON, S.W. ; GOLDBERG, D.E.: Generalization in the XCSF Classifier System: Analysis, Improvement and Extension / Illinois Genetic Algorithm Laboratory, University of Illinois at Urbana-Champaign. 2005 (IlliGAL2005012). – Forschungsbericht.

[Lanzi u. a. 2005c] LANZI, P.L. ; LOIACONO, D. ; WILSON, S.W. ; GOLDBERG, D.E.: XCS with Computed Prediction in Continuous Multistep Environments. In: *Proceedings of the IEEE Congress on Evolutionary Computation – CEC-2005* Bd. 3. Piscataway, NJ, 2005, S. 2032–2039.

[Lanzi u. a. 2005d] LANZI, P.L. ; LOIACONO, D. ; WILSON, S.W. ; GOLDBERG, D.E.: XCS with Computed Prediction in Multistep Environments. In: *Genetic and Evolutionary Computation – GECCO-2005*. New York : ACM Press, 2005, S. 1827–1834.

[Lanzi u. Riolo 2000] LANZI, P.L. ; RIOLO, R.L.: A Roadmap to the Last Decade of Learning Classifier System Research. In: **[Lanzi u. a. 2000b]**, S. 33–61.

[Lanzi u. a. 2000a] LANZI, P.L. (Hrsg.) ; STOLZMANN, W. (Hrsg.) ; WILSON, S.W. (Hrsg.): *Advances in Learning Classifier Systems: Third International Workshop (IWLCS 2001)*. Berlin : Springer, 2000 (LNAI 1813).

[Lanzi u. a. 2000b] LANZI, P.L. (Hrsg.) ; STOLZMANN, W. (Hrsg.) ; WILSON, S.W. (Hrsg.): *Learning Classifier Systems: From Foundations to Applications*. Berlin : Springer, 2000 (LNAI 1813).

[Lanzi u. Wilson 2006] LANZI, P.L. ; WILSON, S.W.: Using Convex Hulls to Represent Classifier Conditions. In: *GECCO '06: Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*. New York : ACM Press, 2006, S. 1481–1488.

[Linde u. a. 1980] LINDE, Y. ; BUZO, A. ; GRAY, R.M.: An Algorithm For Vector Quantizer Design. In: *IEEE Transactions on Communications* COM-28 (1980), S. 84–95.

[MacQueen 1967] MACQUEEN, J.B.: Some Methods For Classification and Analysis of Multivariate Observations. In: LE CAM, L. M. (Hrsg.) ; NEYMAN, J. (Hrsg.): *Proceedings of the Fifth Berkeley Smposium on Mathematical Statistics and Probability*, University of California Press, 1967, S. 281–297.

[Martinetz 1993] MARTINETZ, Th.: Competitive Hebbian Learning Rule Forms Perfectly Topology Preserving Maps. In: *Proceedings of ICANN'93, International Conference on Artificial Neural Networks*. Berlin : Springer, 1993, S. 427 – 434.

[Martinetz u. Schulten 1991] MARTINETZ, Th. ; SCHULTEN, K.: A Neural-Gas Network Learns Topologies. In: KOHONEN, T. (Hrsg.) u. a.: *Artificial Neural Networks* Bd. 1. Amsterdam : North Holland, 1991, S. 397–402.

[Martinetz u. Schulten 1994] MARTINETZ, Th. ; SCHULTEN, K.: Topology Representing Networks. In: *Neural Networks* 7 (1994), Nr. 3, S. 507–522.

[Michalewicz 1996] MICHALEWICZ, Z.: *Genetic Algorithms + Data Structures = Evolution Programs*. Third, Revised and Extended Edition. Berlin : Springer, 1996.

[Miikkulainen 1991] MIIKKULAINEN, R.: Self-Organizing Process Based on Lateral Inhibition and Synaptic Resource Redistribution. In: KOHONEN, T. (Hrsg.) ; MÄKISARA, K. (Hrsg.) ; SIMULA, O. (Hrsg.) ; KANGAS, J. (Hrsg.): *Proceedings of the International Conference on Artificial Neural Networks*. Amsterdam : North-Holland, 1991, S. 415–420.

[Mitchell 1996] MITCHELL, M.: *An Introduction to Genetic Algorithms*. Cambridge, MA : MIT Press, 1996 (Complex Adaptive Systems).

[Mono] *Main Page - Mono*. `http://www.mono-project.com`, Abruf: 10.10.2009. – Das Mono-Projekt - ein Projekt zur Portierung von .NET auf Plattformen wie Linux, Solaris, Unix und MacOS X.

[Moriarty u. Miikkulainen 1998] MORIARTY, D.E. ; MIIKKULAINEN, R.: Forming Neural Networks Through Efficient and Adaptive Coevolution. In: *Evolutionary Computation* 5 (1998), Nr. 4, S. 373–399.

[Moriarty u. a. 1999] MORIARTY, D.E. ; SCHULTZ, A.C. ; GREFENSTETTE, J.J.: Evolutionary Algorithms for Reinforcement Learning. In: *Journal of Artificial Intelligence Research* 11 (1999), S. 241–276.

[Nolfi u. Floreano 1999] NOLFI, St. ; FLOREANO, D.: Learning and Evolution. In: *Autonomous Robots* 7 (1999), Nr. 1, S. 89–113.

[O'Hara 2006] O'HARA, T.: Learning Classifier System with Neural Network Representation / University of the West of England. 2006 (UWELCSG06-008). – Forschungsbericht.

[Patterson 1996] PATTERSON, D.: *Künstliche neuronale Netze: das Lehrbuch*. München : Prentice Hall, 1996.

[Penfield u. Boldrey 1937] PENFIELD, W. ; BOLDREY, E.: Somatic Motor and Sensory Representation in the Cerebral Cortex of Man as Studied by Electrical Stimulation. In: *Brain* 60 (1937), S. 389–443.

[Penfield u. Rasmussen 1950] PENFIELD, W. ; RASMUSSEN, Th.: *The Cerebral Cortex of Man. A Clinical Study of Localization of Function*. New York : Macmillan, 1950.

[Perl 2001] PERL, J.: Ein dynamisch gesteuertes Netz zur Modellierung und Analyse von Prozessen im Sport. In: PERL, J. (Hrsg.): *Sport & Informatik VIII*. Köln : Strauß, 2001.

[Polani 1996] POLANI, D.: *Adaption der Topologie von Kohonen-Karten durch Genetische Algorithmen*, Johannes Gutenberg-Universität Mainz, Diss., 1996.

[Polani u. Uthmann 1992] POLANI, D. ; UTHMANN, Th.: Neuronale Netze - Grundlagen und ausgewählte Aspekte der Theorie / Institut für Informatik, Johannes Gutenberg-Universität Mainz. 1992 (2/92). – Forschungsbericht.

[Rechenberg 1973] RECHENBERG, I.: *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Stuttgart : Fromman-Holzbog, 1973.

[Ritter u. a. 1990] RITTER, H. ; SCHULTEN, K. ; MARTINETZ, Th.: *Neuronale Netze: eine Einführung in die Neuroinformatik selbstorganisierender Netzwerke*. Bonn : Addison-Wesley, 1990.

[Rummery u. Niranjan 1994] RUMMERY, G.A. ; NIRANJAN, M.: On-Line Q-Learning Using Connectionist Systems / Engineering Department, Cambridge University. 1994 (CUED/F-INFENG/TR 166). – Forschungsbericht.

[Russell u. P.Norvig 2003] RUSSELL, St. ; P.NORVIG: *Artificial Intelligence: A Modern Approach*. 2nd edition. Upper Saddle River, NJ : Prentice Hall, 2003.

[Schaffer u. a. 1992] SCHAFFER, J. ; WHITLEY, D. ; ESHELMAN, L.J.: Combinations of Genetic Algorithms and Neural Networks: a Survey of the State of the Art. In: *Proceedings of the Conference on Combinations of Genetic Algorithms and Neural Networks*. Piscataway, NJ : IEEE, 1992, S. 1–37.

[Seung 2000] SEUNG, H.S.: Half a Century of Hebb. In: *Nature Neuroscience* 3 (2000), S. 1166.

[Smith 1980] SMITH, S.: *A Learning System Based on Genetic Algorithms*, University of Pittsburgh, Diss., 1980.

[Stephan u. a. 2003] STEPHAN, V. ; SAUPE, M. ; BISCHOFF, M. ; REINDANZ, H. ; GROSS, H.-M.: Is Reinforcement-Learning Able to Solve Real-World Challenges. In: *Proceedings of ICANN'03, International Conference on Artificial Neural Networks*, Bogaziai University Press, 2003, S. 346–349.

[Stoer 1994] STOER, J.: *Numerische Mathematik 1*. Siebente, neubearbeitete und erweiterte Auflage. Berlin : Springer, 1994.

[Stoer u. Bulirsch 1978] STOER, J. ; BULIRSCH, R.: *Einführung in die Numerische Mathematik II*. Zweite, neubearbeitete und erweiterte Auflage. Berlin : Springer, 1978.

[Stone u. Bull 2003] STONE, C. ; BULL, L.: For Real! XCS with Continuous-Valued Input. In: *Evolutionary Computation* 11 (2003), Nr. 3, S. 299–336.

[Stone u. L.Bull 2003] STONE, C. ; L.BULL: A Note on Crossover with Interval Representation / University of the West of England, Learning Classifier Systems Group. 2003 (UWELCSG03-002). – Forschungsbericht.

[Sutton 1988] SUTTON, R.S.: Learning to Predict by the Methods of Temporal Differences. In: *Machine Learning* 3 (1988), S. 9–44.

[Sutton u. Barto 1998] SUTTON, R.S ; BARTO, A.G.: *Reinforcement Learning*. Cambridge, MA : MIT Press, 1998.

[Turing 1969] TURING, A.M.: Intelligent machinery. In: *Machine Intelligence* 5 (1969), S. 3–23. – Nachdruck in: INCE, D. C. (Hrsg.): *Collected works of A.M. Turing: Mechanical Intelligence*. Amsterdam: North-Holland, 1992.

[Venturini 1994] VENTURINI, G.: *Apprentissage adaptif et Apprentissage Supervisé par Algorithme Génétique*, Université de Paris-Sud, Diss., 1994.

[Warwick u. Ball 1996] WARWICK, K. ; BALL, N.: Self-Organising Neural Networks for Adaptive Control. In: *Journal of Intelligent and Robotic Systems* 15 (1996), S. 153 – 163.

[Watkins 1989] WATKINS, C.J.: *Learning from Delayed Rewards*, Cambridge University, Diss., 1989.

[Weicker 2002] WEICKER, K.: *Evolutionäre Agorithmen*. Stuttgart : B.G. Teubner, 2002 (Leitfäden der Informatik).

[W.H.Press u. a. 1992] W.H.PRESS ; TEUKOLSKY, S.A. ; VETTERLING, W.T. ; FLANNERY, B.P.: *Numerical Recipes in C:The Art of Scientific Programming*. Second Edition. Cambridge University Press, 1992.

[Widrow u. Hoff 1960] In: WIDROW, B. ; HOFF, M.: *Adaptive Switching Circuits*. IRE, 1960, S. 96–104. – Nachdruck in: ANDERSON, J.A. (Hrsg.) ; ROSENFELD, E. (Hrsg.): *Neuro Computing: Foundations of Research*. Cambridge, MA: MIT Press, 1988.

[Wilson 1994] WILSON, S.W.: ZCS: A Zeroth-level Classifier System. In: *Evolutionary Computation* 2 (1994), Nr. 1, S. 1–18.

[Wilson 1995] WILSON, S.W.: Classifier Fitness Based on Accuracy. In: *Evolutionary Computation* 3 (1995), Nr. 2, S. 149–176.

[Wilson 1998] WILSON, S.W.: Generalization in the XCS Classifier System. In: KOZA, J.R. (Hrsg.) ; BANZHAF, W. (Hrsg.) ; CHELLAPILLA, K. (Hrsg.) ; DEB, K. (Hrsg.) ; DORIGO, M. (Hrsg.) ; FOGEL, D.B. (Hrsg.) ; GARZON, M.H. (Hrsg.) ; GOLDBERG, D.E. (Hrsg.) ; IBA, H. (Hrsg.) ; R.RIOLO (Hrsg.): *Genetic Programming 1998: Proceedings of the Third Annual Conference*. San Francisco, CA : Morgan Kaufmann, 1998, S. 665–674.

[Wilson 2000a] WILSON, S.W.: Get Real! XCS with Continuous-Valued Inputs. In: **[Lanzi u. a. 2000b]**, S. 209–219.

[Wilson 2000b] WILSON, S.W.: Mining Oblique Data with XCS. In: **[Lanzi u. a. 2000a]**, S. 158–174.

[Wilson 2002] WILSON, S.W.: Classifiers that approximate functions. In: *Natural Computing* 1 (2002), Nr. 2 - 3, S. 211 – 233.

[Wilson u. Goldberg 1989] WILSON, S.W. ; GOLDBERG, D.E.: A Critical Review of Classifier Systems. In: *Proceedings of the 3rd International Conference on Genetic Algorithms*. San Francisco, CA : Morgan Kaufmann, 1989, S. 244–255.

[ZedGraph] *Main Page - ZedGraph Wiki*. `http://zedgraph.org`, Abruf: 10.10.2009. – Wiki zur ZedGraph Charting Class Library.