Bachelorarbeit

KOOPERATIVES REINFORCEMENT LERNEN IN MULITAGENTENSYSTEMEN

Johannes Knabe
Cognitive Science, Universität Osnabrück
jknabe@uni-osnabrueck.de

Betreuer
Prof. Dr. Martin Riedmiller
Prof. Dr. Volker Sperschneider

April 2005

Abstract

Reinforcement Learning methods have been used successfully to let agents learn behaviour adapted to their environment. The RoboCup federation is a challenge for those methods as it provides a highly complex environment where among other difficulties several agents need to cooperate and coordinate. This thesis at first traces the extension of the theory of Reinforcement Learning methods to the multiagent case and then shows ways of how to adopt them to such complex environments as the RoboCup simulator is. Based on these grounds a multiagent Reinforcement Learning algorithm is introduced and applied to learn a team strategy for overcoming the opponent's defense. The outcomes with different learning settings are empirically evaluated. Finally a team of agents with a learnt strategy is shown to be superior to the former one without this strategy under tournament conditions.

Addendum; April, 10th: Using the described hybrid strategy the Brainstormers team placed first at the RoboCup German Open 2005. Admittedly other parts of the behaviour have been improved as well, so the strategies exact portion of the outcome can not be numeralised.

Zusammenfassung

Reinforcement Lernverfahren werden immer wieder erfolgreich eingesetzt, um Agenten ihrer Umwelt angepasste Verhalten lernen zu lassen. Die RoboCup Fussballliga stellt eine Herausforderung für diese Verfahren dar, da in der hoch komplexen Umgebung neben anderen Schwierigkeiten mehrere Agenten kooperieren und sich koordinieren müssen. Diese Arbeit zeichnet zunächst die Erweiterung der Theorie der Reinforcement Lernverfahren auf den Multiagentenfall nach und zeigt dann Wege auf, diese an so komplexe Umgebungen wie den RoboCup Simulator anzupassen. Auf dieser Vorarbeit aufbauend wird dann ein Mulitagenten Reinforcement Lernverfahren vorgestellt und angewendet um eine Teamstrategie zum Überwinden der gegnerischen Abwehr zu lernen. Die Ergebnisse bei verschiedenen Lerneinstellungen werden evaluiert. Abschliessend wird gezeigt, daß eine Mannschaft von Agenten mit einer gelernten Strategie der ursprünglichen ohne diese Strategie unter Turnierbedingungen überlegen ist.

Addendum; 10. April: Unter Benutzung der beschriebenen hybriden Strategie konnte das Team der Brainstormers bei den RoboCup German Open 2005 den ersten Platz belegen. Allerdings wurden auch andere Teile des Verhaltens verbessert, genau kann der Anteil dieser Strategie am Endergebnis also nicht beziffert werden.

Inhaltsverzeichnis

1 Einführung und Überblick

Der Begriff des Multiagentensystems (MAS) hat sich in den letzten Jahren ausgeprägt um den Gegensatz zu alleine operierenden Agenten herauszustellen. Anders als letztere können solche Agenten koordiniert interagieren, was in vielen Anwendungen nicht nur von Vorteil sondern gar inherente Vorraussetzung ist (man denke an offene, verteilte, heterogene Systeme, e.g. "intelligente" Haushaltsgeräte, s.a. [Wei99]). Klassische Argumente für automatische Lernverfahren wie schnelle Adaptivität und langfristige Arbeitsersparnis treffen hier verstärkt zu, denn bei zunehmender Komplexität steigt der Arbeitsaufwand einer handkodierten Programmierung enorm an, weshalb man verstärkt spezielle Lernverfahren zu entwickeln sucht. Als realitätsnahe Test- und Benchmarkumgebung der Lernfähigkeiten sowohl von Einzelagentenfähigkeiten als auch der Kooperation im Team hat sich der RoboCup etabliert. In verschiedenen Ligen treten hier Roboter in einer dem Fussball ähnlichen Disziplin gegeneinander an, wobei momentan für MAS-Lernverfahren hauptsächlich noch die Simulationsliga von Interesse ist, da Probleme mit teurer und fehleranfälliger Hardware wegfallen.
Das Rückverstärkungs- oder Reinforcement Lernen (RL) geht ultimativ auf die erstmalig von Thorndike [Tho11] formulierte Beobachtung zurück, dass adaptionsfähige Lebewesen aus Handlungsalternativen auf Grund der zu erwartenden Konsequenzen auswählen und dass solche Alternativen mit Situationen assoziiert werden (Law of Effect). RL Verfahren haben sich auf Grund ihrer allgemeinen Formulierung bei vielen Lernproblemen bewährt (zur Geschichte von RL siehe auch [SB98]), weshalb eine Erweiterung auf den Multiagenten Fall nahe lag und liegt. Um diese Entwicklung nachzuvollziehen werden neben der Modellierung der Umwelt als Lernproblem zunächst grundlegende Ein-Agenten RL Verfahren vorgestellt und diese dann auf den

allgemeineren Fall ausgedehnt.
Das Ziel für die Brainstormers[1] Roboterfussball-Mannschaft ist, komplett mit RL Methoden zu funktionieren. Hierbei erfolgen Rückmeldungen der "Aussenwelt" als Indikatoren für die Güte des gezeigten Verhaltens nur sporadisch - wenn nämlich ein Tor gefallen ist. Auf Grund der hohen Komplexität einer solchen Mannschaftsstrategie wurde eine hierarchische Modularisierung vorgenommen, so dass handkodierte Teile nach und nach durch gelernte Verhalten ersetzt werden können. Grundsätzlich wurde automatisches Lernen für einfache Ein-Agenten Verhalten erreicht, nun kann man sich also, darauf aufbauend, Multiagenten Strategien widmen. Konkret wird in dieser Arbeit ein Angriffsverhalten gelernt, welches zum Ziel hat, das Verhalten der attackierenden Agenten kurz vor dem gegnerischen Tor dahingehend zu optimieren, dass die Erfolgschance maximal wird.
Trotz der wenigen bislang vorhandenen theoretischen Aussagen zum Ablauf und Ergebnis von Multiagenten RL (insb. garantierte Konvergenz zu einer optimalen Strategie) wurde eine ganze Reihe von RL Verfahren für den Multiagentenfall vorgeschlagen; praktisch funktionieren die Varianten auch meist sehr gut bei den zum Demonstrationszwecke konstruierten simplen Problemen. Solche Ergebnisse sind jedoch nicht immer direkt auf komplexere Probleme übertragbar, da bspw. vereinfachende Annahmen wie ein perfektes System-Modell den Lernverlauf stark beeinflussen können. Weiterhin sind die in verschiedenen Umgebungen erzielten Resultate untereinander kaum zu vergleichen. Daher wird im empirischen Teil in der echten Umgebung untersucht, welche Methode und welche Parameter Vorteile mit sich bringen.

[1] Entwickelt von einem Team um M. Riedmiller in Karlsruhe, Dortmund und Osnabrück. Vgl. http://amy.informatik.uos.de/asg/projects/brainstormers/

2 Theoretische Grundlagen

Reinforcement Lernmethoden werden auf viele Lernprobleme angewandt, bei denen aus fortgesetzten Interaktionen mit der Umwelt ein zielgerichtetes Verhalten gelernt werden soll. Genauer ist RL dann anwendbar, wenn ein Problem auf drei Signale reduziert werden kann, die in diskreten Zeitschritten zwischen Agent (bzw. Lerner/Controller) und Umwelt ausgetauscht werden: ein Signal für den Systemzustand, eines für die gewählte Aktion sowie ein Reinforcementsignal (als Indikator für die Güte der letzten Aktion). Da der RoboCup Simulator später zum Zwecke des RL als solch ein sequentielles Auswahlproblem für Agenten modelliert werden soll, beschäftigen wir uns mit den sog. Markovschen[1] Entscheidungsprozessen (MDP, Markov decision process) etwas ausführlicher.

2.1 Markovsche Entscheidungsprozesse

Wir beginnen zunächst mit klassischen Ein-Agenten MDP, da diese als Grundlage für Multiagentensysteme dienen werden. Wie wir weiter unten sehen werden, sind die Ergebnisse mit gewissen Einschränkungen auf letzteren Bereich übertragbar.

Ein MDP M ist gegeben durch ein Tupel (S, A, R_i, p_i) bestehend aus:

- Menge der Zustände S,
- Menge der Aktionen A,

[1] Andere Schreibweisen: Markow, Markoff

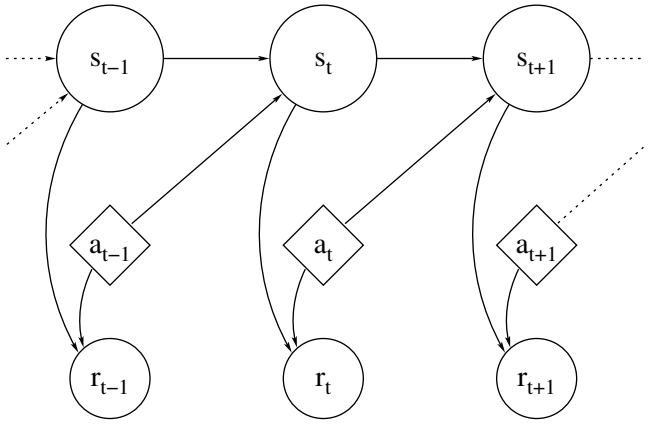

Abbildung 2.1: *MDP aus Sicht des Agenten.* Zu sehen sind beobachtete Zustände (Kreise s_k) sowie die gewählte Aktion (Rauten a_k). Diese zwei beeinflussen die Kosten (kleine Kreise r_k) und den nächsten Systemzustand.

- Reinforcementverteilung $R_t : \mathbb{R} \times S \times A \to [0, 1]$ sowie der
- Übergangswahrscheinlichkeit $p_t : S \times S \times A \to [0, 1]$.

Wendet der Agent zum Zeitpunkt $t \in \mathbb{N}$ eine Aktion $a \in A$ auf den MDP M im Zustand s an, erhält er mit Wahrscheinlichkeit $R_t(r|s, a)$ das Reinforcement r und M geht mit Wahrscheinlichkeit $p_t(s'|s, a)$ in Zustand s' über (vgl. Abb. 2.1).

Sowohl die Reinforcementverteilung R_t als auch die Übergangswahrscheinlichkeit p_t könnten also mit der Zeit t variieren. Wir beschränken uns aber auf sogenannte stationäre MDPs mit

$$\begin{aligned} R_t(r|s, a) &= R_{t'}(r|s, a) \forall t, t' \in \mathbb{N}, \\ p_t(s'|s, a) &= p_{t'}(s'|s, a) \forall t, t' \in \mathbb{N} \end{aligned} \tag{2.1}$$

Daher lassen wir die Indizes t meist weg.
Ferner werden im Folgenden nur endliche Mengen S und A betrachtet, was uns zu sogenannten finiten MDP führt und die Behandlung vereinfacht, da bspw. statt Integralen Summen anfallen.
Bei $p(s'|s, a)$ handelt es sich um eine Wahrscheinlichkeitsverteilung auf S; es muss

also gelten $\sum_{s' \in S}(s'|s,a) = 1, \forall s \in S, a \in A$.

Uns interessiert vor allem das zu erwartende unmittelbare Reinforcement, welches wir definieren als $r : S \times A \to \mathbb{R}$ mit $r(s,a) = E\{r|s,a\} = \int_{-\infty}^{\infty} xR(x|s,a)dx$.

Für einen MDP wird weiterhin die Markoveigenschaft angenommen (auch Markov-Kette erster Ordnung), die definiert ist als:

$$\begin{aligned} p(s_{t+1}|s_t,a_t) &= p(s_{t+1}|s_t,a_t,s_{t-1},a_{t-1},...,s_0,a_0), \forall s_i \in S, a_i \in A, \\ r(s_t,a_t) &= r(s_t,a_t,s_{t-1},a_{t-1},...,s_0,a_0), \forall s_i \in S, a_i \in A \end{aligned} \tag{2.2}$$

Die Umweltdynamik soll also ausschliesslich vom jüngsten Zustand/Aktions-Paar abhängen und nicht von der Historie beeinflusst sein[2].

2.2 Strategie

Nun stellt sich die Frage, welche Aktionen der Agent wählen sollte.
Hierfür definieren wir zunächst ein Markovsches Verhalten als Funktion

$$v : A \times S \to [0,1] \text{ mit } \sum_{a \in A} v(a|s) = 1, \forall t \in \mathbb{N}, s \in S \text{ [3]}$$

Es bezeichnet $v(a|s)$ die Wahrscheinlichkeit, mit der der Agent in einem Zustand s die Aktion a wählt[4].
Deterministische Verhalten, d.h. pro Zustand genau eine Aktion, ergeben sich als

[2]Eine Bedingung, die oftmals aufgelockert wird, was bei kleinen Einflüssen auch nicht stark ins Gewicht fällt.

[3]Natürlich sollte wegen der Markoveigenschaft, Def. 2.2, die beste Strategie ohne Betrachtung des Geschichtshorizonts in einem MDP genau so gut sein wie die beste Strategie mit Geschichtshorizont.

[4]Im Allgemeinen hängen die wählbaren Aktionen vom aktuellen Zustand ab, $A(s) \subset A$. Um die Notation zu vereinfachen setzen wir $v(a|s) = 0$ für $a \notin A(s), \forall s \in S$

Sonderfall mit der weiteren Forderung:

$$v(a|s) \in \{0,1\}, \forall a \in A, s \in S \tag{2.3}$$

Es kann auch sinnvoll sein, das Verhalten mit der Zeit zu ändern, was uns zum allgemeineren Begriff der Strategie führt. Eine Strategie π ist eine Folge von Verhalten $\pi = (v_1, v_2, ...)$ mit dem Spezialfall der stationären Strategie $\pi = (v, v, ...) = v^\infty$, bei der Verhalten und Strategie zusammenfallen. Im Folgenden ist, wenn nicht anders angegeben, mit $\pi(a|s)$ immer der stationäre Fall gemeint.

2.3 Bewertungsfunktionen

Fast alle RL Algorithmen basieren auf der Schätzung von Bewertungsfunktionen, die angeben, wie gut es für den Agenten ist, in einem bestimmten Zustand zu sein, bzw. in einem Zustand eine bestimmte Aktion auszuführen. Die Güte ist bestimmt durch die künftig zu erwartenden Kosten[5] welche natürlich von der Strategie abhängen.
Um dies formal auszudrücken, definieren wir als Maß für die bei Befolgung von Strategie π von Zustand s aus zu erwartenden Kosten als:

$$V^\pi(s) = E\{\sum_{k=0}^{N} \gamma^k R^\pi_{s,k+1}\}, \tag{2.4}$$

wobei N der zufällige aber finite Endzeitpunkt des Lernproblems ist und die Zufallsvariable $R^\pi_{s,i}$ den Kosten des Agenten nach i Zeitschritten korrespondiert wenn ab Zustand s Strategie π verwendet wird.

Die Diskontierung $\gamma \in [0,1]$ kann als Weitsichtigkeit des Agenten betrachtet werden, mit kleiner werdendem γ fallen künftige Kosten weniger ins Gewicht. Im Extremfall $\gamma = 0$ würden nur die unmittelbar nächsten Kosten berücksichtigt[6].

[5] In dieser Arbeit werden die Reinforcements als Kosten interpretiert, bei der Interpretation als Belohnungen ändern sich aber nur die Vorzeichen und es wird maximiert statt minimiert.

[6] Bei Problemen ohne festlegbaren Endzeitpunkt dient $\gamma < 1$ ausserdem dazu, die Endlichkeit der Summe sicherzustellen.

Die rekursiven Eigenschaften solcher Bewertungsfunktionen zeigen sich in der sog. Bellman-Gleichung für V^π:

$$\begin{aligned} V^\pi(s) &= E\{\sum_{k=0}^{\infty} \gamma^k R^\pi_{s,k+1}\} = E\{R^\pi_{s,1} + \sum_{k=1}^{\infty} \gamma^k R^\pi_{s,k+1}\} \\ &= E\{R^\pi_{s,1} + \sum_{k=0}^{\infty} \gamma^{k+1} R^\pi_{s,k+2}\} = E\{R^\pi_{s,1} + \gamma \sum_{k=0}^{\infty} \gamma^k R^\pi_{s,k+2}\} \\ &= \sum_{a\in A} \pi(a|s) \sum_{s'\in S} p(s'|s,a)[r(s,a) + \gamma E\{\sum_{k=0}^{\infty} \gamma^k R^\pi_{s',k+1}\}] \\ &= \sum_{a\in A} \pi(a|s) \sum_{s'\in S} p(s'|s,a)[r(s,a) + \gamma V^\pi(s')], \forall s \in S \ ^7 \end{aligned} \tag{2.5}$$

Wobei wir anstatt eines Rekursionsabbruches von einem MDP ausgehen, der keinen Terminalzustand erreicht. Durch Einführung eines absorbierenden Zustandes, der nicht mehr verlassen werden kann und für keine Aktion Kosten verursacht, können wir aber endliche Probleme in unendliche überführen (vgl. [Put94]).
Die Bellman Gleichung drückt also die konsistente Beziehung zwischen dem Wert eines Zustandes und derer der Nachfolgezustände aus; der Wert eines Zustandes ist gleich der gemittelten Werte aller (γ-diskontierten) Folgemöglichkeiten plus der jeweils unmittelbar entstehenden Kosten, wobei jede Alternative mit ihrer Auftretenswahrscheinlichkeit gewichtet wird.

Der Agent soll sich natürlich möglichst erfolgreich in seiner Umwelt verhalten, was in unserem Sinne heisst, eine Strategie zu wählen, die die Kosten so gering wie möglich hält.
Das ist aber genau dann der Fall, wenn die zu erwartenden Kosten bei dieser Strategie für jeden Zustand kleiner oder gleich aller alternativen Strategien sind. Optimal ist

[7] Weiterhin gilt:

$$\sum_{a\in A} \pi(a|s) \sum_{s'\in S} p(s'|s,a)[r(s,a) + \gamma V^\pi(s')] = \sum_{a\in A} \pi(a|s)[r(s,a) + \gamma \sum_{s'\in S} p(s'|s,a) V^\pi(s')]$$

also

$$V^*(s) = \min_{\pi} V^{\pi}(s), \forall s \in S. \tag{2.6}$$

Auch für V^* gelten die Konsistenz-Eigenschaften der Bellmann-Gleichung, da es jedoch mehrere optimale Strategien geben kann[8], schreibt man sie ohne Bezug auf eine spezielle Strategie:

$$V^*(s) = \min_{a \in A}\{\sum_{s' \in S} p(s'|s,a)[r(s,a) + \gamma V^*(s')]\}, \forall s \in S \tag{2.7}$$

Hier spiegelt sich die Intuition wieder, dass die beste Bewertung eines Zustands von der Auswahl der Aktion mit den geringsten zu erwatenden Folgekosten ausgehen sollte[9].

Haben wir V^* bestimmt, ergeben sich die zugehörigen optimalen Strategien π^* mittels:

$$\pi^*(a|s) > 0 \Leftrightarrow a \in \arg\min_{a \in A} \sum_{s' \in S} p(s'|s,a)[r(s,a) + \gamma V^*(s')], \forall s \in S \tag{2.8}$$

Es kann also in einem Zustand mehrere optimale Aktionen geben, deren Ausführung aber jeweils dieselben minimalen zukünftigen Kosten erwarten lässt. Nur solchen Aktionen werden optimale Strategien eine Auswahlwahrscheinlichkeit grösser Null zuordnen, mit dem Sonderfall der deterministischen Auswahl genau einer Aktion pro Zustand (vgl. 2.3).

In einfachsten Fällen kann man die Bellman-Gleichung für V^* für jeden Zustand s aufstellen und das entstehende Gleichungssystem lösen. Bei vielen Zuständen wird dies aber schnell zu rechenaufwändig, weshalb man iterative Verfahren verwendet, die unter den Oberbegriff dynamisches Programmieren fallen.
Beim sogenannten Value Iteration Algorithmus wird die Bellman-Gleichung in eine

[8] Die jedoch die Bewertungsfunktion V^* teilen; für finite MDPs ist V^* die eindeutige Lösung ihrer Bellman-Gleichung (Beweis in [Put94]).

[9] Eleganterweise reicht es, nur einen Schritt voraus zu schauen, um Langzeit-Optimalität zu erreichen, da die weitere Entwicklung bereits in V^* enthalten ist

Aktualisierungsregel umgewandelt:

$$V_{t+1}(s) = \min_{a \in A} \sum_{s' \in S} p(s'|s,a)[r(s,a) + \gamma V_t(s')], \forall s \in S \tag{2.9}$$

Bei beliebigem V_0 konvergiert die Folge $\{V_t\}$ gegen V^* für $t \to \infty$. (Beweis in [Ber95])

Die Konvergenz zur optimalen Lösung im Grenzfall $t \to \infty$ ist garantiert, während man praktisch abbricht, wenn die Änderungen nur noch sehr gering ausfallen. Da die vom Agenten zum Lernen verwendete Strategie keine Rolle spielt, solange sie nicht einige Aktionen gar nicht verwendet, spricht man von einem off-policy Verfahren.
Man beachte jedoch, dass wir hier immer vorrausgesetzt haben, dass die Systemdynamik, also $p()$ und $r()$, bekannt ist.

In unbekannten Umgebungen bieten sich Methoden an, bei denen nur aus Erfahrungen gelernt wird; d.h. zufälligen Folgen von Interaktionen mit der Umwelt.
Beobachten wir bei Ausführung von Aktion a in Zustand s ein Reinforcement r und ein Übergang nach Zustand s', ergibt sich formal ein Tupel der Form (s, a, r, s').
Von den diversen existierenden Algorithmen, die auf Basis solcher Tupel lernen, werden wir hier nur Q-Lernen [WD92] betrachten, vor allem wegen seiner Beziehung zur Bewertungsfunktion V^*.

Bezeichnen wir nämlich mit $Q^\pi(s,a)$ die zu erwartenden Kosten bei Auswahl von Aktion a in Zustand s und anschliessender Befolgung von Strategie π, so ergibt sich in Analogie zur Bellman-Gleichung (Gl. 2.7) Q^* als:

$$Q^*(s,a) = \sum_{s' \in S} p(s'|s,a)[r(s,a) + \gamma \min_{a' \in A}(Q^*(s',a'))], \forall s \in S, a \in A \tag{2.10}$$

Hierbei haben wir für den rekursiven Teil der Bellman-Gleichung ausgenutzt, dass $V^*(s') = \min_{a' \in A}(Q^*(s',a'))$ gilt.

Da $\sum_{s' \in S} p(s'|s,a) \min_{a' \in A}(Q^*(s',a'))$ durch $\min_{a' \in A}(Q^*(s',a'))$ geschätzt werden kann, wenn der Folgezustand s' stets entsprechend $p()$ ausgewählt wird, können wir auch

den letzten Rest Vorwissen über die Systemdynamik verschwinden lassen.

Dies (siehe auch Fussnote 7) führt zur iterativen Aktualisierungsregel des Q-Lernens bei Beobachtung von (s, a, r, s'):

$$Q_{t+1}(s,a) = (1-\alpha_t)Q_t(s,a) + \alpha_t(r + \gamma \min_{a' \in A} Q_t(s', a')) \tag{2.11}$$

Mit einer kleinen Lernrate α_t zur Gewichtung zwischen bisheriger Approximation und neuer Beobachtung.[10] Wiederum ist die Konvergenz der Folge $\{Q_t\}$ gegen Q^* für $t \to \infty$ gesichert; es müssen nur unendlich viele Tupel (s, a, r, s') beobachtet werden (vgl. [JJS94]). Wiederum handelt es sich um ein off-policy Verfahren.

Mit Q^* ergeben sich optimale Strategien noch leichter:

$$\pi^*(a|s) > 0 \Leftrightarrow a \in \arg\min_{a \in A} Q^*(s,a) \tag{2.12}$$

2.4 Allgemeine Multiagenten MDPs

Im folgenden modellieren wir Multiagentensysteme, in denen mehrere Agenten parallel in sequentiellen Entscheidungsprozessen agieren, als sogenannte Multiagenten MDPs (MMDP) - in der Literatur auch stochastic games genannt.
Dabei ist ein (stationärer) n Agenten MDP M_n gegeben durch ein Tupel (S, A, R, p) bestehend aus:

- Menge der Zustände S,
- Menge der Aktionen $A = A_1 \times \cdots \times A_n$,
- Reinforcementverteilung $R : \mathbb{R}^n \times S \times A \to [0, 1]$ sowie der
- Übergangswahrscheinlichkeit $p : S \times S \times A \to [0, 1]$

[10] Im Allgemeinen kann α_t nicht nur Zeit- sondern auch Zustands- und Aktionsabhängig sein, $\alpha_t = \alpha_t(s, a)$.

Der i. Agent kann zum Zeitpunkt $t \in \mathbb{N}$ eine Aktion $a_i \in A_i$ auf den MDP M_n im Zustand s anwenden und somit die Gesamtaktion $a \in A$ beeinflussen. Er erhält ein Reinforcement r_i, der i. Skalar des Vektors r und M_n geht mit Wahrscheinlichkeit $p(s'|s,a)$ in Zustand s' über.
Wir gehen wieder vom finiten Fall aus, A und S seien also endlich, und definieren das zu erwartende unmittelbare Reinforcement als $r : S \times A \to \mathbb{R}^n = E\{r|s,a\}$.

Es soll eine sinnvolle Strategie gefunden werden, mit der der i. Agent seine individuellen Kosten minimal hält. Wie verhalten sich aber die anderen Agenten? Eine Grundannahme der meisten RL Algorithmen ist, dass die Systemdynamik stationär ist, also insbesondere nicht von anderen lernenden Agenten abhängt[11].
Deren Strategien beeinflussen den Folgezustand und damit mittelbar die Langzeitkosten. Hängen die Kosten des i. Agenten, $r_i(s,a)$, auch von den Aktionen der übrigen Agenten ab, ändern sich sogar die unmittelbaren Kosten.
Im Allgemeinen ist es dann nicht mehr möglich, ein eindeutiges V^* zu definieren, vgl. [Mer99].

2.5 Kooperative MMDPs

Anders sieht das bei kooperativen MMDPs aus, auf die wir uns im Weiteren konzentrieren werden. In diesem Fall arbeiten alle (lernenden) Agenten auf ein gemeinsames Ziel hin, was sich darin ausdrückt, dass sie dieselben Kosten erhalten.
Wir fordern also für einen kooperativen MMDP $M_n = (S, A, r, p)$ zusätzlich:

$$r_i(s,a) = r_j(s,a), \forall i,j \in \{1,...,n\}, \forall s \in S, a \in A \tag{2.13}$$

Da jeder Agent gleiche Kosten erhält, haben weiterhin alle gemeinsame optimale Bewertungsfunktionen V^* bzw. Q^*.

[11] Nicht lernende Agenten können als Teil der Systemdynamik betrachtet werden, in diesem Fall reduziert sich das Problem auf einen normalen MDP.

Nehmen wir nun an, wir hätten einen einzelnen Agenten, der komplexe Aktionen $a \in A = A_1 \times \cdots \times A_n$ ausführen kann. Für diesen ergibt sich der klassische MDP $M = (S, A, r_1, p)$ mit komplexen Strategien $\pi = (\pi_1, ..., \pi_n)$, wobei π_i jeweils Aktionen aus A_i wählt.
Wegen dieser Parallelen zum MDP Fall existiert für den kooperativen Fall folgendes Ergebnis:

In einem kooperativen MMDP M_n existieren stationäre deterministische Strategien $(\pi_1^*, ..., \pi_n^*)$, die optimal sind. Beweis in [Mer99]. (2.14)

Bei verteilt lernenden Agenten ist aber i. A. anzunehmen, dass ein Agent, der selber nur Teilaktion a_i auswählt, nicht die anderen Aktionen beobachten kann (nur deren Auswirkungen auf den Zustand) und damit nicht weiss, wie die Gesamtaktion $a \in A = A_1 \times \cdots \times A_n$ ausgesehen hat.

2.6 Verteiltes Q-Lernen gemeinsamer Aktionen

Dies ist aber auch gar nicht nötig, da mit etwas initialer Koordination die Gesamtaktion trotzdem von jedem Agenten individuell rekonstruiert werden kann. Man denke an einen Trainer beim Fussball, der vor dem Spiel seiner Mannschaft eine Taktik mitgibt.
Glücklicherweise handelt es sich sowohl bei Value Iteration als auch beim Q-Lernen um sogenannte off-policy Verfahren. D.h. die Konvergenz zur optimalen Lösung ist gesichert, unabhängig von der während des Lernens benutzten Strategie – solange prinzipiell jedes Zustands/Aktions-Paar unendlich oft besucht wird.
Um dies zu gewährleisten, sollten die Lern-Strategien dann nicht-stationär sein $\pi_i^t =$

$(v_i^1, v_i^2, ...)$ und bspw. von einem Pseudo-Zufallszahlengenerator[12] abhängen:

$$v_i^t(s, a_j) = \begin{cases} 1 & j = rand_i^t \bmod |A_i| \\ 0 & \text{sonst} \end{cases} \tag{2.15}$$

Mit der Pseudo-Zufallszahl $rand_i^t \in \mathbb{N}$[13] und a_j der j. Aktion der auf konsistente Weise in eine Folge umgewandelten Menge A_i.
Starten nun alle n Agenten mit derselben Bewertungsfunktion $Q_0^i = Q_0$ wird diese Eigenschaft auch im Lernverlauf beibehalten, wie folgender Beweis zeigt[14]:

Es gelte $Q_t^i(s, a) = Q_t^j(s, a) \forall i, j \in \{1, ..., n\}, \forall s \in S, a = (a_1, ..., a_n) \in A$.
Das System sei zu einem Zeitpunkt t in einem Zustand s. Jeder Agent i wählt die Aktion a_i mit $v_i^t(s, a_i) = 1$ zur Ausführung, woraufhin das System in einen Folgezustand s' wechselt und er Kosten von r_i erhält. Die Gesamtaktion $a = (a_1, ..., a_n)$ mit $a_i \in A_i, v_i^t(s, a_i) = 1$ wurde ausgeführt[15]. Die Kosten r ergeben sich als $(r_i, ..., r_i) \in \mathbb{R}^n$. Mit dem Tupel (s, a, r, s') wendet nun jeder Agent die Aktualisierungsregel (vgl. 2.11) an:

$$Q_{t+1}^i(s, a) = (1 - \alpha_t) Q_t^i(s, a) + \alpha_t (r + \gamma \min_{a' \in A} Q_t^i(s', a')) \tag{2.16}$$

Da alle dasselbe Tupel benutzen sind auch nach der Aktualisierung alle Bewertungsfunktionen gleich.

Nach dem Lernvorgang konstruiert jeder Agent die optimale deterministische Strategie – die wegen Satz 2.14 existieren muss – in Analogie zum Ein-Agenten-Fall (vgl. 2.12) wie folgt:

$$\pi^*(a|s) = \begin{cases} 1 & a = sel(\arg\min_{a \in A} Q^*(s, a)) \\ 0 & \text{sonst} \end{cases} \tag{2.17}$$

[12] Diese generieren bei Initialisierung mit bspw. der Nummer i des Agenten immer dieselbe Sequenz von gleichverteilten Zahlen.
[13] Genauer: Dem t. Element der Sequenz von Pseudo-Zufallszahlen.
[14] Wir beschränken uns hier auf Q-Lernen, Value Iteration funktioniert aber analog.
[15] Die Gesamtaktion lässt sich rekonstruieren, da jeder die Strategien und somit die ausgeführten Teilaktionen der anderen Agenten kennt.

Hierbei könnte sich ein Koordinationsproblem ergeben, und zwar wenn mehrere Gesamtaktionen in einem Zustand gleiche minimale Kosten erwarten lassen. Hätten wir für zwei Agenten bspw. die Gesamtaktionen (a, b) und (b, a) als optimal gegeben, wollen wir keinesfalls, dass beide dieselbe Teilaktion wählen. (a, a) bzw. (b, b) könnten sehr viel schlechtere Kosten erwarten lassen. Dieses Problem lässt sich durch Definition einer Ordnungsrelation auf A lösen, so dass die Menge wohlgeordnet wird, d.h. sich in jeder Teilmenge von A ein minimales Element bestimmen lässt, welches jeder Agent dann wählt[16].
Ein ähnlicher Ansatz wird in [LR04] verfolgt; ebenfalls mit umfassender Koordination vorab aber ohne explizite Repräsentation der Gesamtaktion.

Man kann die Durchsuchung des Zustandsraumes sogar effektiver gestalten (bspw. bei sehr großen Zustandsmengen), indem man bevorzugt solche Aktionen auswählt, die zu Zuständen führen, die auf Grund der bisherigen Erfahrung wahrscheinlich geringe Kosten erzeugen[17].

Dafür berechnet jeder Agent auf Grundlage seiner aktuellen Approximation der Bewertungsfunktion die beste stationäre deterministische Strategie $\pi_i^{\sim}$ und befolgt dann die Strategie $\pi_i^{\epsilon,t}$:

$$v_i^{\epsilon,t}(s, a_j) = \begin{cases} v_i^{t+1}(s, a_j) & rand_i^t \bmod 4 = 0 \\ \pi_i^{\sim}(s, a_j) & \text{sonst} \end{cases} \tag{2.18}$$

Der Agent wird also in den meisten Fällen[18] die bislang optimal erscheinende Aktion auswählen, aber manchmal immer noch explorieren ($v_i^t(s, a_j)$ sei wie in Gl. 2.15 definiert).

Leider steigt bei diesem Verfahren, da die Gesamtaktion explizit repräsentiert wird,

[16] Wobei die Funktion *sel* für diesen Fall in konsistenter Weise die Auswahl treffen soll.

[17] Eine solche Strategie nennt man ϵ-greedy; Die Verstärkte Suche um erfolgversprechende Zustände bringt erfahrungsgemäß in der Praxis Vorteile - für die theoretische Konvergenz spielt dies keine Rolle.

[18] Wobei anstatt des Wertes 4 natürlich auch ein anderes $n \in \mathbb{N}$ gewählt werden kann, i. A. sollte dieser Wert von ϵ abhängen!

die Grösse der Tabelle zum Speichern der $Q(s, a = (a_1, ..., a_n))$ exponentiell mit der Zahl der Agenten an. Solch eine durch extensive Absprache hergestellte zentralistische Koordination ist aber nicht immer gewollt bzw. möglich.

2.7 Verteiltes unabhängiges Q-Lernen

Aus den o.g. Gründen wäre es wünschenswert, dass die Agenten unabhänging jeweils eine Bewertungsfunktion $Q_i(s, a_i)$ lernen, also ohne die Strategien der übrigen Agenten und damit deren Teilaktionen kennen zu müssen, und dennoch die Konvergenz zu optimalen Teilstrategien gesichert ist. Die Vorteile des verteilten, unabhängigen Lernens von Bewertungen für Paare Zustand/Teilaktion werden aber erkauft durch den Übergang in eine höhere Problemklasse, der wir uns nun zuwenden.

Den bislang vorgestellten Lernverfahren gemeinsam ist das Problem, dass sie von einer festen Systemdynamik ausgehen. Anderen Agenten in einem MMDP beeinflussen jedoch mit ihren Aktionen sowohl Übergangswahrscheinlichkeiten als auch entstehende Kosten. Wenn sie ebenfalls lernen, erfolgt diese Beeinflussung nicht in konsistenter Weise, da sie ihre Strategien fortlaufend anpassen. Bei mehreren unabhängig lernenden Agenten erscheint die Systemdynamik dem einzelnen Agenten daher nicht mehr stationär (vgl. Def. 2.1).
Unter diesen Vorraussetzungen so zu lernen, dass zuletzt in jedem Zustand die optimale Gesamtaktion ausgeführt wird, ist nicht einfach, wie folgendes Beispiel illustriert:
Nehmen wir einen indeterministischen MDP für einen Agenten mit zwei Aktionen $\{a, b\}$ an (vgl. Abb. 2.2), bei dem wir bei Ausführung von a mit Wahrscheinlichkeit 1 in einen Zustand kommen, der Kosten von 1 verursacht. Wenn bei Ausführung von b in der Hälfte der Fälle Kosten von 0 und sonst Kosten in Höhe von 4 entstehen, ergibt sich a als optimale Aktion, da dies im Schnitt billiger ist. So weit, so gut.

Sei nun aber ein Zwei-Agenten-MDP mit je zwei Aktionen $\{a, b\}$ gegeben (vgl. Abb. 2.3), bei dem deterministisch diese Kosten entstehen:

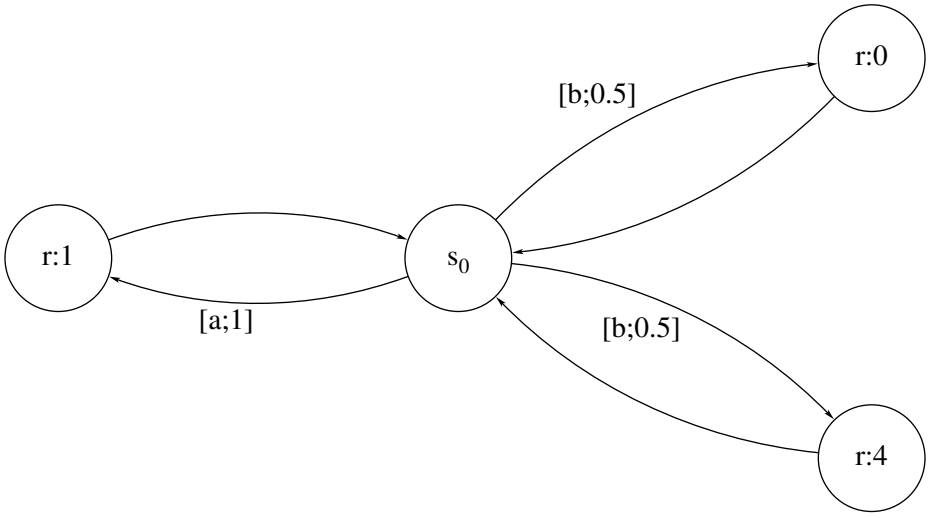

Abbildung 2.2: *Ein-Agenten Beispiel-MDP (schematische Darstellung).* An den Pfeilen zwischen den Zuständen in eckigen Klammern die zugehörige Aktion mit Übergangswahrscheinlichkeit. Da uns nur s_0 interessiert finden sich an den Rückpfeilen keine Werte, was andeuten soll, dass jede beliebige Aktion die im Kreis angegeben Kosten verursacht und wieder zu s_0 führt.

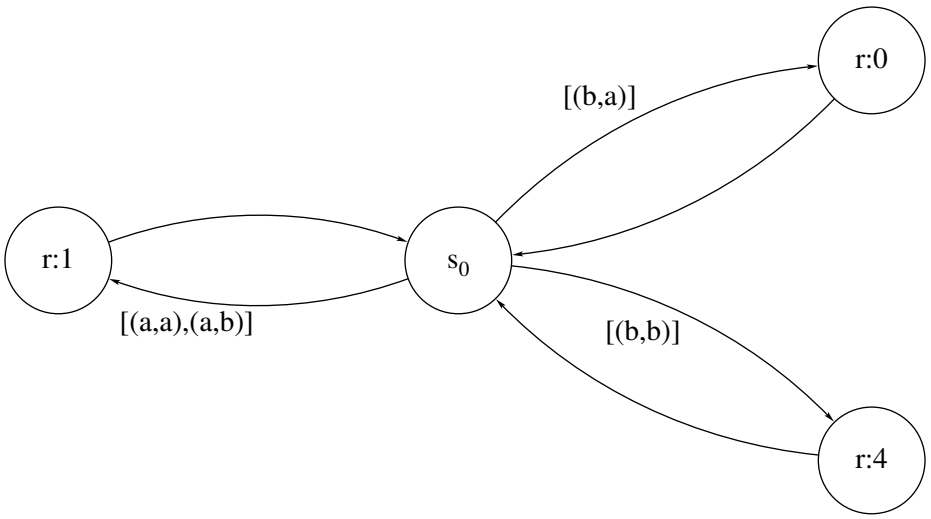

Abbildung 2.3: *Zwei-Agenten Beispiel-MDP (schematische Darstellung).* Hier an den Pfeilen nur Aktionspaare – da es sich um einen deterministischen MMDP handelt sind keine Übergangswahrscheinlichkeiten angegeben, sonst wie in Abb.2.2.

Gesamtaktion	Kosten
(a, a)	1
(a, b)	1
(b, a)	0
(b, b)	4

Die Gesamtaktion (b, a) wäre also die optimale Wahl. Wenn aber der zweite Agent bspw. seine Aktionen gleichverteilt zufällig auswählte, also die Strategie $\pi(x|s) = |A_i|^{-1} = 0.5$ verwendet, würde unser erster Agent beim Lernen den Ein-Agenten-MDP von eben (Abb. 2.2) wahrnehmen - der andere Agent "ersetzt" die Stochastizität des Systems. Folglich würde Agent 1 wie oben am Ende des Lernvorgangs Aktion a bevorzugen, was aber nicht seine optimale Teilaktion ist[19].
Die Erwartungswerte der $Q(s, a)$ alleine enthalten also zu wenig Information, um immer die optimale Teilaktion zu lernen, solange auch nur ein anderer Agent nicht seine optimale Teilaktion spielt: Wir bleiben in lokalen Optima stecken.

Man könnte auf die Idee kommen, die Agenten anstatt der Erwartungswerte für die Folgekosten die minimal entstehenden Folgekosten lernen zu lassen. Das hilft jedoch nur bei deterministischen Zustandsübergängen, denn dann wird dieses Minimum auch tatsächlich immer erreicht (vgl. [LR00]). Bei indeterministischen Systemen könnte aber eine Gesamtaktion durch einen sehr unwahrscheinlichen Zustandsübergang mit kleinsten Kosten uns einen Strich durch die Rechnung machen. Deren Teilaktionen wären dann die neuen Minima obschon im Schnitt höhere Kosten erzeugt würden.
Auch die Approximation der individuellen Übergangswahrscheinlichkeiten $p_i(s'|a_i, s)$ hilft i. A. nicht, da sich hier die Übergangswahrscheinlichkeiten $p(s'|a, s)$ des Systems mit den Aktionen der anderen Agenten mischen. Das Erreichen des besten Nachfolgezustandes kann unwahrscheinlich sein, weil nur verhältnismässig wenige Gesamtaktionen zu ihm hinführen oder weil dieser Übergang im System an sich sehr unwahrscheinlich ist. Ersteres können die Agenten beeinflussen, letzteres nicht. Ein Unterscheiden

[19] Solche Beispiele lassen sich natürlich auch für andere Werte finden, aber für 0.5 ist es besonders anschaulich.

dieser Fälle benötigt wahrscheinlich aber eine gewisse Koordination, wie Kommunikation oder vorige "Absprache".

Spannend ist noch die Frage, ob nicht mit möglichst wenig Kommunikation[20] und ohne erschöpfende Vorabkoordination zwischen den Agenten doch noch die Konvergenz zu optimalen Strategien bewiesen werden kann.
Bei obigem Beispiel würde es immer noch möglich sein, durch abwechselndes Lernen die optimale Strategie zu finden. Würde nämlich zunächst der erste Agent lernen, während der zweite gleichverteilt zufällig spielt, ist das Gelernte nicht optimal, wie oben gesehen. Kommunizieren sie dann aber einen Rollentausch, dass also der zweite lernt während der Rest (hier nur Agent 2) gleichverteilt spielt, wird der zweite Agent die optimale Teilstrategie lernen. Lernt in der nächsten Iteration Agent 1 während Agent 2 seine erlernte Strategie testet, wird auch vom ersten Agenten die optimale Teilstrategie erlernt.

Im folgenden Beispiel (Abb. 2.4) zeigt sich aber, dass auch wechselnde Rollen uns nicht immer aus den lokalen Optima heraushelfen, da unter Umständen jeder Agent die falsche Strategie lernt:
Es gibt drei Agenten mit je drei Aktionen, und für jeden von ihnen führen viele Gesamtaktionen, bei denen sie immer die gleiche (in der Abb. durch Pfeil angedeutete) Teilaktion spielen müssen, zu relativ niedrigen Folgekosten[21]. Nur eine einzige Gesamtaktion erzeugt aber minimale Kosten, (c, a, b). Jeder einzelne Agent würde also, bei gleichverteilt zufälligem Spiel der anderen, beobachten, dass die Aktion mit den vielen Übergängen häufig auftritt, im Schnitt also am wenigsten Kosten verursacht und daher zu wählen ist.

Am praktikabelsten scheint daher fast, die minimalen Folgekosten wie oben beschrieben zu lernen und jeweils die scheinbar beste Teilstrategie zu wählen. Werden dann

[20] Zumindest aber nur endlich viel, denn sonst könnten wir die jeweiligen Aktionen genau koordinieren und hätten den Fall von oben (vgl. 2.6).

[21] In Abb. 2.4 ist der Übersicht halber für jeden Agent ein "eigener" relativ guter Folgezustand eingezeichnet, man kann diese aber auch zusammenlegen.

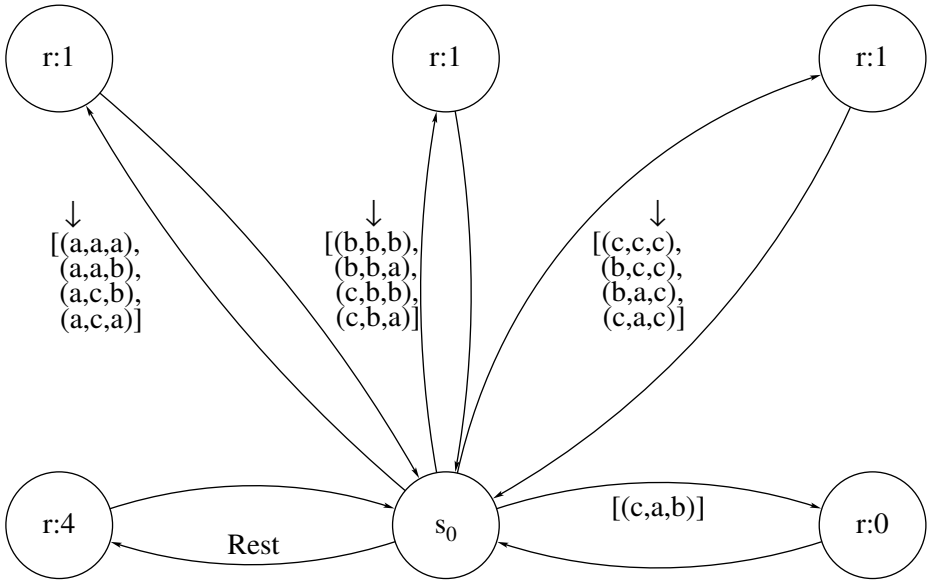

Abbildung 2.4: *Drei-Agenten MDP (schematische Darstellung).* Wir gehen wieder von deterministischen Zustandsübergängen aus; "Rest" steht für die 14 übrigbleibenden Tripel, bspw. (a, b, c).

auf Grund der Systemdynamik von einem Zustand aus im Schnitt höhere Kosten erzeugt, als die zweitbeste Aktion bestenfalls[22] erwarten lässt, wird der Agent i diese Teilstrategie ausprobieren und dies den anderen mitteilen[23]und so fort. Für dieses Verfahren lassen sich leicht Beispiele konstruieren, bei denen alle Permutationen der Teilstrategien[24] getestet werden müssen. Es bliebe nur zu hoffen, dass in tatsächlichen Anwendungen die Anzahl der Zustände, bei denen viele Teilaktionen probiert werden müssen bis das Optimum gefunden ist, gering bliebe. Solch ein Vorgehen kann also recht aufwendig sein; insbesondere in komplexen Anwendungen werden Algorithmen bevorzugt, die nicht diskret sondern graduell stochastisch zur besten Lösung kommen und auch bei frühem Abbrechen mit hoher Wahrscheinlichkeit ein sehr gutes Ergebnis liefern. Inwiefern es möglich ist, durch Nutzung von Kommunikation und geeigneter stochastischer Strategien den lokalen Optima zu entgehen und im Grenzfall

[22]Nämlich wenn deren $Q_i(s, a_i)$ Wert tatsächlich immer erreicht würde.

[23]Ein Strategiewechsel muss bekannt gemacht werden, da sonst die Erwartungswerte der anderen mit gleichbleibender Strategie verwischt würden.

[24]Was endlich viele sind, da es optimale stationäre deterministische Strategien geben muss und wir von einem finiten MMDP ausgehen (vgl. Satz 2.14).

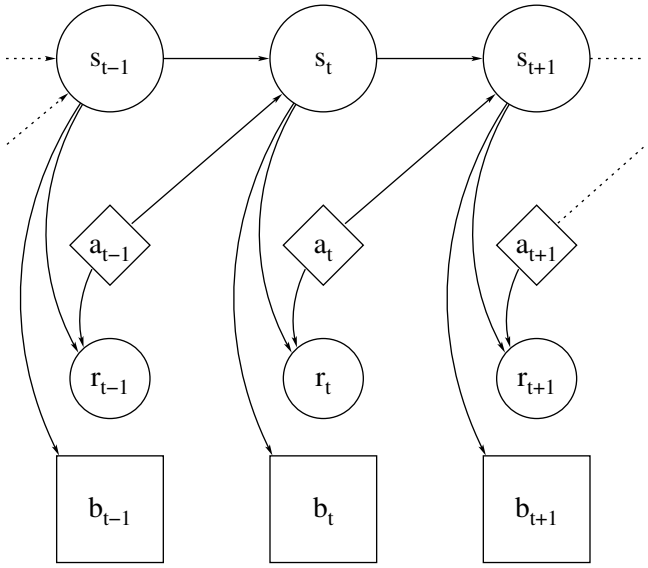

Abbildung 2.5: *POMDP aus Sicht des Agenten.* Zu sehen sind beobachtete Zustände (Quadrate b_k) sowie die gewählte Aktion (Rauten a_i) und der vor dem Agenten versteckte Systemzustand (große Kreise s_k). Die zwei letzteren beeinflussen die Kosten (kleine Kreise r_k) und den nächsten Systemzustand.

mit Wahrscheinlichkeit 1 beste Teilstrategien zu finden, bedarf noch der Klärung.

2.8 Teilweise beobachtbare MDPs

Die meisten realistischen Entscheidungsprobleme, ob für mehere Agenten oder nicht, gehen zusätzlich mit eingeschränkter Weltinformation einher und sind dann nicht unbedingt markovsch in den Beobachtungen. Ein Lösungsansatz sind sogenannte teilweise beobachtbare MDPs (partially observable MDP, POMDP). Hierbei wird davon ausgegangen, daß der modellierte Prozess eigentlich Markovscher Natur ist, dessen Zustandsvariablen jedoch nicht vollständig beobachtet werden können. Die tatsächlich beobachteten Zustandsvariablen und Reinforcements zum Zeitpunkt t sollen jedoch ausschließlich vom unsichtbaren Systemzustand zur Zeit t abhängen (vgl. Abb. 2.5). Formal fordern wir, daß die Beobachtungen b bei Systemzustand s durch eine Wahrscheinlichkeitsverteilung $q(b|s)$ ausgewählt werden.

Auch für finite POMDPs gibt es exakte Lösungsalgorithmen[25] (e.g. [CLZ97], [Son76]) die aber nur für sehr kleine Probleme nützlich sind und einige approximative Verfahren (e.g. [PR95]), einen Überblick geben [Sal02] sowie [KLC98]. Bei mehr als einem lernenden Agenten übertragen sich natürlich alle vorerwähnten Probleme.

2.9 Zusammenfassung

In diesem Kapitel wurden Grundlagen der Modellierung sequentieller Entscheidungsprozesse vorgestellt und beschrieben, wie optimale Strategien für Agenten aussehen bzw. unter bestimmten Vorraussetzungen auch garantiert gefunden werden können. Im Weiteren werden wir immer wieder auf diese theoretischen Erkenntnisse zurückgegreifen. Da in realistisch großen Anwendungsfällen wie der in dieser Arbeit betrachteten RoboCup-Umgebung nicht alle gemachten Annahmen erfüllt werden, wird manchmal von den beschriebenen Verfahren abgewichen werden, um in der Praxis gute Ergebnisse zu erzielen.
Denn trotz der teilweise negativen theoretischen Ergebnisse gibt es eine Reihe erfolgreicher Anwendungen von verteiltem Reinforcement Lernen. Dies liegt daran, dass viele Umgebungen nicht ganz so harte Anforderungen stellen wie obige Beispiele suggerieren mögen, also bspw. nur ein relativ kleiner Teil des Zustandsraumes überhaupt als Nachfolger des aktuellen Zustands in Frage kommt und Aktionen der anderen Agenten zwar nicht direkt bekannt werden aber doch implizite Spuren im Zustand hinterlassen.
Wie so eine Umwelt konkret aussehen kann erfahren wir im nächsten Kapitel.

[25] Der interessierte Leser sei hier auf die Verwandtschaft zu den sogenannten versteckten Markov Modellen (Hidden Markov Models, HMMs) hingewiesen. HMMs fehlt nur die Entscheidungskomponente - der Beobachter kann keine Aktionen ausführen um das System und damit die Zustandsübergänge zu beeinflussen. Auf Grund ihrer sehr allgemeinen Formulierung wurden sie verstärkt untersucht und diverse Algorithmen gefunden, Anwendungsbereiche liegen vor allem in der Bioinformatik und Sprachverarbeitung (bspw. [Rab89]). Ein finiter POMDP entspricht einem HMM mit einer eigenen Zustandsübergangsmatrix für jede Aktion; diese Ähnlichkeit ist Grundlage mancher POMDP Lösungsverfahren.

3 Roboterfussball als MMDP

3.1 RoboCup

Fernziel des RoboCup ist es, dass im Jahre 2050 die amtierenden menschlichen Fussballweltmeister von einer Robotermannschaft unter Beachtung der offiziellen Spielregeln besiegt werden. Da die Roboter noch große Defizite sowohl gegenüber der Hard- als auch Software der menschlichen Vorbilder haben, werden in mehreren Ligen Wettkämpfe der konkurrierenden Teams ausgetragen. In Zukunft sollen diese Ligen dann zusammenwachsen um das ambitionierte Ziel zu erreichen.
Vorerst ist der RoboCup hauptsächlich Testfeld und ein international anerkannter Benchmark für diverse Techniken, die in mit der Robotik zusammenhängenden Forschungsgebieten entwickelt werden. In dieser Arbeit interessieren wir uns nur für die sogenannte 2D Simulationsliga, denn hier wird der am weitesten gehende Gebrauch von Verfahren der Künstlichen Intelligenz gemacht, da durch Abstraktionen einige Probleme vorwiegend technischer Natur wegfallen.

3.2 Soccer Server

Der Soccer Server ist die Simulationsumgebung der 2D Simulationsliga, er stellt das Spielfeld und einen automatischen Schiedsrichter zur Verfügung. Über ein Netzwerkprotokoll[1] können sich dort die Agenten (Spieler) zweier Mannschaften anmelden

[1] Den Details des benutzten Netzwerkprotokolls wollen wir uns nicht näher widmen, ausführlich erläutert werden diese in [CFH+02].

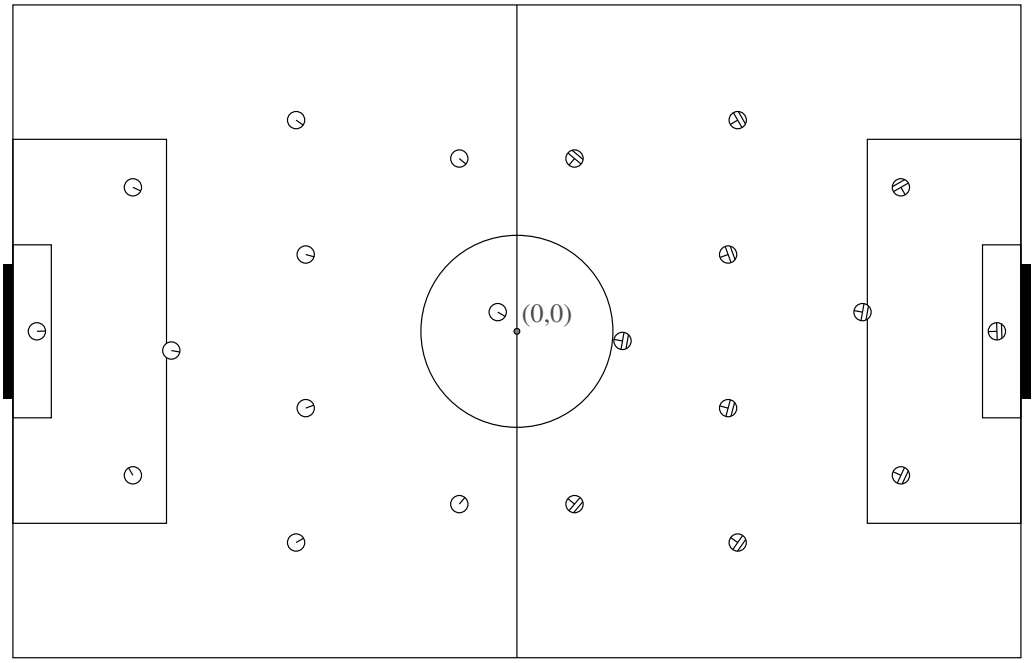

Abbildung 3.1: *Soccer Server Umgebung* Dargestellt ist das $[-52.5, 52.5] \times [34.0, 34.0]$ Feld mit einer möglichen Anfangsaufstellung.

und gegeneinander antreten (Abb. 3.1). In diskreten Zeitzyklen werden den Agenten die Positionen der wichtigen Objekte übermittelt, woraufhin diese mit Aktionen den nächsten Systemzustand beeinflussen können. Das System läuft in Echtzeit, d.h. wenn ein Agent zwischen zwei Wahrnehmungszyklen kein Signal sendet, wird er übergangen. Wichtig ist auch, dass zwar Hardwareprobleme und Objekterkennung wegfallen, aber sonst größtmögliche Realitätsnähe angestrebt wird: Insbesondere ist jeder Agent ein unabhängiger Prozess, der seine Entscheidungen auf Grund der ihm vom Server gelieferten Daten trifft. Kommunikation mit anderen Agenten ist zwar möglich, aber nur über den Server und in sehr begrenztem Rahmen. Weiterhin sind die Informationen, die der Agent über seine unmittelbare Umgebung erhält beschränkt und nicht notwendigerweise exakt[2] (vgl. POMDP, Abschnitt 2.8).

[2]Auf diese Weise wird graduelles "aus den Augen verlieren" simuliert; je weiter weg ein Objekt desto ungenauer die Daten bis es schliesslich ganz verschwindet.

Ein Zustand des Soccer Servers besteht aus dem Spielmodus[3] m und Informationen über die bis zu 22 Spieler: Die Positionen $p_i = (x, y)$ in der Spielfeld-Ebene, Körperorientierungen o_i, Geschwindigkeiten $v_i = (v_x, v_y)$ und Energielevel e_i, sowie Position p_b und Geschwindigkeit v_b des Balls. Die Physik ist markovsch[4], d.h. der nächste Zustand hängt nur von den Aktionen der Agenten und dem vorigen Zustand ab (vgl. Def. 2.2). Die wichtigsten Aktionen sind[5]:

Aktion	Wirkung
`kick`(k, w)	Der Ball wird mit Kraft k in Richtung w beschleunigt, falls nahe genug.
`turn`(w)	Drehe um Winkel w.
`dash`(k)	Beschleunige in aktuelle Richtung mit Kraft k.

Die Auswirkungen dieser Aktionen hängen allerdings auch von der Energie ("Kondition") des Agenten ab und können noch auf verschiedene Weisen verrauscht werden, bspw. mit "Wind". Zusätzlich gibt es die Möglichkeit, mit heterogenen Spielern zu arbeiten. Dann weist der Soccer Server zu Beginn jedem Spieler einen zufälligen Typ zu, der u.a. seine Energiewerte festlegt. "Fitte" Spieler brauchen dann für dieselbe Strecke weniger Zeit, was wir in dieser Arbeit allerdings nicht berücksichtigen. Eine detaillierte Beschreibung sowie die exakten Formeln für die Systemdynamik können in [CFH+02] nachgeschlagen werden. Soweit nicht anders angegeben wurden auch die dort genannten Standardwerte für die Parameter benutzt.
Für uns reicht es zu wissen, dass dieser Prozess angenähert als (teilweise beobachtbarer) MMDP beschrieben werden kann. Wichtig ist auch, dass in den meisten Fällen die Aktion eines Agenten i nur kleine Änderungen am Zustand bewirkt - nämlich an seiner eigenen Repräsentation (p_i, o_i, v_i, e_i). Der Agent im Ballbesitz ändert natürlich auch die Werte (p_b, v_b). Es kann auch vorkommen, dass zwei Agenten den Ball mit einem `kick`-Kommando beeinflussen oder zwei Agenten zusammenstossen, und insbesondere ein Tor verändert den gesamten Systemzustand gewaltig - aber diese Ereignisse sind

[3] Es gibt tatsächlich mit Freistößen und ähnlichem 18 Modi. Uns interessieren aber nur `play_on` und `goal_`*`Seite`*`.`

[4] Mit wenigen Ausnahmen wie der Abseitsregel.

[5] Es gibt noch weitere, bspw. `catch`() für den Torwart.

relativ selten. Deshalb und da die stochastischen Einflüsse auf den Systemübergang nicht so groß sind, wird die Menge der möglichen nächsten Zustände am stärksten durch den aktuellen Zustand beschränkt.
Was für Informationen erhält der Agent vom Server? Es gibt eine weitere Aktion, die zwar nicht den Folgezustand beeinflusst, dafür aber wichtig für die Wahrnehmung des Agenten ist: `turnneck`(w) ändert die Blickrichtung relativ zur Körperorientierung und damit das Sichtfeld des betreffenden Agenten. In der Standardkonfiguration beträgt das Sichtfeld des Agenten 90 Grad und die Distanz für exakte Wahrnehmung 20 Meter. Weiter entfernte Agenten können nur noch mit einer gewissen Wahrscheinlichkeit wahrgenommen werden.
Dem einzelnen Agenten wird hier also - ganz abgesehen von den Aktionen der anderen Agenten - nicht der volle Zustand des Systems bekannt (vgl. POMDP, Abschnitt 2.8) und folglich nehmen verschiedene Agenten meist auch den Systemzustand verschieden war, was die Koordination zusätzlich erschwert.
Alle im Zusammenhang mit dem Soccer Server besprochenen Variablen wie Position in der Ebene, Kickkraft, usf. sind weiterhin kontinuierlich[6]. Da Aktionsmenge und Zustandsmenge nicht mehr endlich sind müssen wir uns von der Vorstellung eines finiten MDP verabschieden.
Die nächsten Abschnitte befassen sich daher mit Techniken, die hohe Komplexität der Soccer Server Umgebung so in den Griff zu bekommen, dass die eingangs besprochenen Modellierungen und zugehörigen RL-Verfahren angewendet werden können.

3.3 Modularisierung der Strategien

Naheliegend ist eine hierarchische Gliederung des Problems, um so die Entscheidungskomplexität zu verringern[7]. Bei den Brainstormer Agenten wird auf der obersten Entscheidungsebene nach einer groben Kategorisierung der Spiellage verzweigt[8]; bspw.

[6] Soweit man das von Fließkommazahlen auf Digitalrechnern behaupten kann.

[7] Tatsächlich benutzen die meisten Teams irgendeine Form der Modularisierung, vgl. etwa [WKMS03],[SM01].

[8] Momentan sind diese fest kodiert, da vorhandene Verfahren zur automatischen Modularisierung (etwa [MB01]) noch nicht mithalten können, wenn die Kriterien wie hier auf der Hand liegen.

welche Mannschaft im Ballbesitz ist und in welchem Abschnitt sich der Ball befindet - was die verbleibenden zu analysierenden Möglichkeiten schon stark einengt. Detaillierte Beschreibungen zur Architektur der Brainstormers finden sich in [RMN+03] sowie [RGKS05]. Die Teilprobleme der mittleren Ebene entscheiden über die Ausführung von Makroaktionen[9], wobei eine Makroaktion einfache Ein-Agenten-Züge wie

- `geheNach`(x, y)
- `passeZu`(n)
- `fangeBallAb`$()$

kapselt. Bei Auswahl auf der Makroebene verwandeln diese jeweils unter Benutzung der o.g. primitiven Aktionen die gegebene Situation in eine Folgesituation. Im Allgemeinen wird solch eine Makroaktion aus mehreren Aktionen über ebensoviele Zeitschritte N bestehen und, da Systemstochastik und evtl. Aktionen anderer Agenten Einfluss nehmen, in einem Zustand s' aus einer Menge von Möglichkeiten S^{term} terminieren.
Um die optimale Sequenz von Aktionen für eine Makroaktion zu bestimmen, wurden bereits erfolgreich Reinforcement Lernmethoden eingesetzt (vgl. etwa [Mei00]), wobei die Reinforcementfunktion $r : S \times A \to \mathbb{R}$ definiert wurde als:

$$r(s,a) = \begin{cases} 0 & s \in S^+ \\ 1 & s \in S^- \\ \mathsf{c} & \text{sonst} \end{cases} \tag{3.1}$$

Mit S^+, den erwünschten, und S^- den unerwünschten Ausgängen; $S^+ \cup S^- = S^{\text{term}}$, $S^+ \cap S^- = \emptyset$, $\mathsf{c} \in]0,1[$.
Es entstehen also keine Kosten für den Fall, dass der Agent einen erfolgreichen Terminalzustand erreicht, maximale Kosten bei einem unerwünschten Endzustand und in allen übrigen Fällen kleine konstante Kosten. Durch letztere Bedingung wird ein

[9] In anderen Arbeiten auch Optionen oder Moves genannt.

zeitoptimales Verhalten gefördert; die akkumulierten diskontierten Kosten fallen geringer aus, je schneller die Makroaktion beendet ist.
Die Anzahl der Entscheidungen N wird also möglichst klein gehalten und die Aktionen der Mitspieler sind größtenteils vernachlässigbar (sie werden im Zustand des Lernproblems also auch nicht repräsentiert), was sicher beides zu den guten Ergebnissen beiträgt.

Untersuchtes Teilproblem

In dieser Arbeit werden wir ein Teilproblem betrachten, dass eine Ebene höher als die o.g. Ein-Agenten-Züge angesiedelt ist. Die zugehörige Makrostrategie[10] wird von der obersten Ebene aktiviert, wenn die eigene Mannschaft im Angriff und der Ball nur noch 20 Meter vom gegnerischen Tor entfernt ist. Es ist unschwer zu erraten, dass eine Strategie gelernt werden soll, die eine solche Situation recht häufig in einen Torerfolg verwandelt. Die Entfernung ist so gewählt, dass die Anzahl der Entscheidungen, um einen Terminalzustand zu erreichen nicht besonders groß sein muss. Ein Umkreis von 20 Metern um das gegnerische Tor mag sich nach einer sehr starken Einschränkung anhören; praktisch ist es jedoch so, dass ein Angriff – so er denn das Mittelfeld passiert – fast immer auch in diese Region vorstossen kann. Die beste Verteidigungstaktik besteht nämlich oftmals darin, sich in die Nähe des eigenen Tores zurückzuziehen und dort zu versuchen, den Ball abzufangen um nicht hinterlaufen zu werden. Aus Entfernungen grösser 20 Meter ist es nicht möglich, Tore zu schiessen; es ist also mindestens eine Aktionsentscheidung dieser Strategie nötig, um erfolgreich zu sein.

[10] Eine Makrostrategie ist eine Strategie, die die Wahrscheinlichkeit der Auswahl einer Makroaktion in einem Zustand angibt. Im Weiteren bezeichnen wir sie mit Strategie, da die gemeinte Ebene aus dem Kontext hervorgeht.

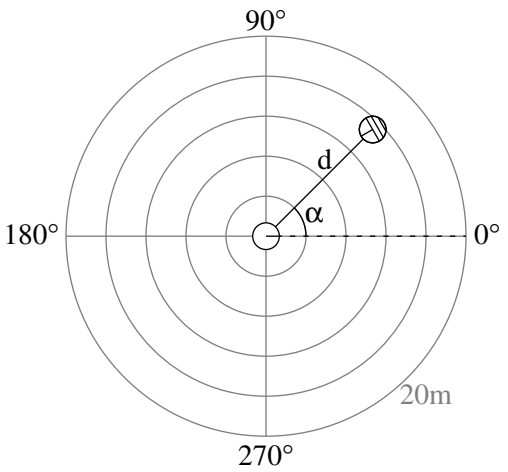

Abbildung 3.2: *Polarkoordinaten* Relativ zu Position und Blickrichtung eines Spielers (nicht massstabsgetreu).

3.4 Reduktion des Zustandsraums

Oftmals ist eine gute Methode, die Komplexität eines Problems zu reduzieren, nicht direkt auf den Zuständen zu lernen, sondern auf Eigenschaften (Abstraktionen, Features) von diesen. Damit wird zwar selbst aus einem reinen MDP ein POMDP, aber meist ist die Zustandsinformation bis zu einem gewissen Grade redundant oder irrelevant, der Informationsverlust daher nicht unbedingt wesentlich. Man kann Lernalgorithmen anwenden, die unter gewissen Vorraussetzungen ein optimales Verhältnis zwischen relevanter Information und sparsamer Kodierung lernen; wir werden uns hier aber darauf beschränken, sehr allgemeines Vorwissen einzubringen. Dies äussert sich wie folgt:

Alle absoluten Positionsdaten werden in relative umgewandelt, genauer gesagt in Polarkoordinaten (vgl. Abb. 3.2). Dabei wird aus $p_{\text{abs}} = (x, y)$ $p_{\text{rel}} = (d, \alpha)$ mit dem Winkel α zur Polarrichtung und der Entfernung d zum Koordinatenursprung[11], wobei in dieser Arbeit $d \in [0, 20]$. Entfernungen grösser 20 Meter werden "abgeschnitten" und somit nicht exakt berücksichtigt, was aber nicht weiter schlimm ist, da erfahrungsgemäß so weit entfernte Objekte erstmal keine Rolle spielen. Probleme können

[11] Der Koordinatenursprung wird auf die Position des Spielers im Ballbesitz gelegt, hierauf werden wir im nächsten Kapitel noch genauer eingehen.

sich hieraus ergeben, wenn der Terminalzustand nicht in greifbarer Nähe liegt. Dann kann entscheidend sein, ob eine Entfernung 20 oder 30 Meter beträgt, dennoch stellen sich beide Zustände für uns gleich dar.
Die beschriebene Koordinatenumwandlung wird durchgeführt für die repräsentierten[12] Gegenspieler, Mitspieler, den Ball sowie den Tormittelpunkt. Weiterhin wird die Norm der Ballgeschwindigkeit sowie wiederum deren Winkel relativ zur Blickrichtung in die Zustandsrepräsentation einbezogen. Weitere Zustandsdaten werden nicht berücksichtigt um die Komplexität des Problems zu verringen. Wir werden diese Abstraktion vom Zustand s des Soccer Server als $a(s)$ notieren.

3.5 Diskretisierung der Aktionsmenge

Auch mit der oben erläuterten Modularisierung und dem Lernen von Makrostrategien gibt es immer noch eine kontinuierliche Menge von möglichen Aktionen[13], man denke an `geheNach`(x, y) mit $x, y \in \mathbb{R}$. Da wir eine Zustandsbewertungsfunktion lernen wollen, müssen die Ergebnisse der Aktionen simuliert werden um dann den Wert des resultierenden Zustands zu ermitteln[14]. Mitunter ist das sehr zeitaufwändig, weshalb nur sehr wenige Aktionen untersucht werden, womit wir natürlich auch riskieren, dass die optimale Aktion eventuell gar nicht berücksichtigt wird.
In dieser Arbeit hängt die Menge der zur Verfügung stehenden Aktionen A vom Zustand des jeweiligen Agenten ab, und zwar wird unterschieden nach: Ball befindet sich im Schussbereich / nicht im Schussbereich.
Die Spieler ohne Ball untersuchen für 36 Positionen ihrer Umgebung, in Schritten von zwanzig Grad in einem und dann in zwei Meter Entfernung um ihre aktuelle Position, ob es sich lohnt, diese anzusteuern (Abb. 3.3).
Der im Ballbesitz befindliche Spieler vergleicht die Auswirkungen von

[12] Wir werden nicht alle Spieler repräsentieren; so dürfte bspw. die Position unseres Torwarts für ein Angriffsverhalten unbedeutend sein.

[13] Ab hier bezeichnen wir mit Aktionen immer Makroaktionen, wenn nicht explizit anders angegeben.

[14] Bei einer Q-Funktion müsste für jedes Zustand/Aktions Paar eine Bewertung gespeichert werden. Dafür entfällt die Simulation mittels Modell; hierauf wird später noch eingegangen.

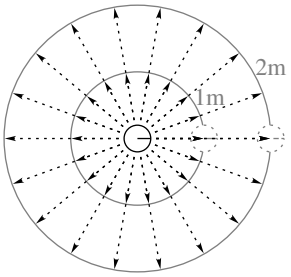

Abbildung 3.3: *Ohne Ball*

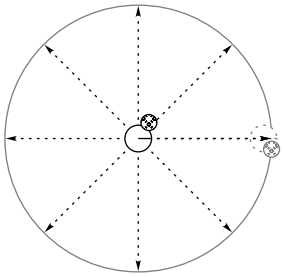

Abbildung 3.4: *Selbstpässe*

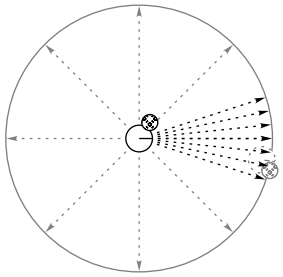

Abbildung 3.5: *Mehrstufig*

- Pass zum Mitspieler mit kleinster Entfernung,
- Pass zum zweitnächsten Mitspieler[15],
- Vorwärts dribbeln (Ball im Schussbereich halten),
- und Selbstpässe in acht Richtungen (Abb. 3.4).

Insbesondere bei letzterem würde sich eine mehrstufige Diskretisierung im Gegensatz zur gewählten linearen Diskretisierung anbieten (vgl. [Mei00]), also unter der Annahme einer stetigen Bewertungsfunktion nach ersten Stichproben erfolgversprechende Zielzustandsregionen genauer zu untersuchen (Abb. 3.5). Auf Grund des zusätzlichen Rechenaufwandes wurde auf diese lokale Suche aber verzichtet.
Bei der Implementation wurden noch einige Sonderregeln realisiert, so daß die Menge der tatsächlich zu prüfenden Aktionen meist noch etwas weiter reduziert werden konnte (siehe Abschnitt 4.5).

3.6 Approximation der Bewertungsfunktion

Die umgewandelte Zustandsrepräsentation hat zwar die Komplexität schon verringert, aber auch Polarkoordinaten sind noch kontinuierlich. Die eingangs vorgestellten Lernverfahren gingen jedoch vom Vorliegen einer finiten Zustandsmenge aus, um dann tabellenartig Bewertungen speichern zu können.

[15]Für die beiden Passoptionen gilt: Zum nächsten bzw. zweitnächsten *repräsentierten* Mitspieler.

Eine Diskretisierung wie bei der Aktionenmenge kommt hier nicht in Frage, denn selbst bei einer Einteilung in Ein-Meter- und Zehn-Grad-Schritte bleiben $(20*36)^{n+m+2}$ Möglichkeiten alleine für die Koordinaten der repräsentierten n Mitspieler, m Gegenspieler sowie Ball und Tor. Noch grobere lineare Auflösungen sind nicht sinnvoll, da dann zuviel wichtige Information verloren ginge (ein Meter kann durchaus schon über Tor oder nicht entscheiden). Abgesehen von der enormen Zahl der benötigten Speicherplätze spielt auch die Zeit und Datenmenge eine nicht zu vernachlässigende Rolle; man erinnere sich, dass wir theoretisch in jedem Zustand jede Aktion unendlich oft ausführen müssten. In der Praxis ist man natürlich auf eine endliche, größere Anzahl von Wiederholungen beschränkt. Aber auch das ist nicht immer in ausreichendem Maße möglich.

Die Lösung heisst Generalisierung von Erfahrung, hier erreicht durch Funktions-Approximation. Das heisst, die Bewertungsfunktion zur Zeit t, $V_t : S \rightarrow \mathbb{R}$ wird nicht mehr für jeden einzelnen Zustand explizit repräsentiert[16], sondern als ganzes durch eine parametrisierte Funktion $V_{\vec{\Phi}_t} : S \rightarrow \mathbb{R}$ angenähert. Da die Zahl der Parameter in dem die Funktion bestimmenden Vektor $\vec{\Phi}_t$ üblicherweise sehr viel niedriger ist als die Zahl der Zustände, wird bei Update eines Zustandes $s \in S$ die Bewertung vieler Zustände verändert (Generalisierung). Dies kann natürlich von Vorteil sein, wenn die Zustände, deren Werte mit verändert wurden, ähnliche Eigenschaften aufweisen wie s, unter Umständen aber zu "unzulässigen Verallgemeinerungen" und gar Divergenz des Lernvorgangs (vgl. [Bai95]) führen[17].

In dieser Arbeit werden zur Funktions-Approximation künstliche Neuronale Netze verwendet. Es lassen sich aber auch andere Verfahren anwenden; [SB98], Kap.8, gibt hierzu einen Überblick.

Formal handelt es sich bei den hier benutzten vorwärtsgerichteten, geschichteten neuronalen Netzen K um 5-Tupel $K = (M, C, F, w, \Theta)$, mit

- einer Menge $M = \{1, ..., n\}$ nummerierter Neuronen,

[16] Wie oben besprochen (Abschnitt 3.4) lernen wir nicht direkt auf den Zuständen $s \in S$, es sollte korrekterweise $V_t : A \rightarrow \mathbb{R}$ mit $A = \{a(s) | s \in S\}$ heissen.

[17] Die theoretischen Konvergenzeigenschaften von RL-Methoden in Kombination mit Funktionsapproximation sind ein aktives Forschungsgebiet, siehe z.B. [MS04].

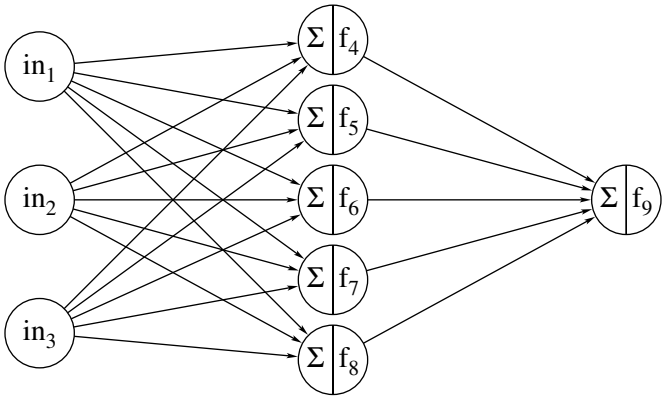

Abbildung 3.6: *Neuronales Netz* Dargestellt ist die Architektur eines vorwärtsgerichteten neuronalen Netzes mit drei voll verbundenen Schichten.

- der Verbindungsrelation C, wobei $i, j \in M$ verbunden sind, wenn $(i, j) \in C$,
- der Menge $F = (f_1, ..., f_n)$ der Aktivierungsfunktionen $f_i : \mathbb{R} \to \mathbb{R}$,
- den Gewichten[18] $w_{ji} \in \mathbb{R}$ für jede Verbindung $(i, j) \in C$,
- den Schwellwerten $\Theta_i \in \mathbb{R}, i \in [1, ..., n]$

Weiterhin sei (M, C) ein azyklischer Graph. Dann gibt es mindestens ein Neuron ohne Vorgänger, solche Neuronen i heißen Eingabeneuron und haben die Aktivierungsfunktion $f_i(v) = v$, sind also nur "Puffer" für die Eingangswerte. Die übrigen Neuronen bekommen als Eingangswert die gewichtete Summe der Ausgaben ihrer Vorgänger, substrahieren ihren Schwellwert[19], füttern damit ihre Aktivierungsfunktion und geben das Resultat aus.
Die Schichtung des Netzes stellen wir sicher, indem sich M in p disjunkte Teilmengen $M = M_0 \cup ... \cup M_p,\ M_i \cap M_j = \emptyset \leftrightarrow i \neq j,\ (x, y) \in C \leftrightarrow x \in M_i \wedge y \in M_{i+1}$ zerlegen lassen muss. M_0 soll alle Eingabeneuronen enthalten und nur M_p Neuronen ohne Nachfolger, die Ausgabeneuronen; wir können das Netz dann als Funktion $K : \mathbb{R}^{|M_0|} \to \mathbb{R}^{|M_p|}$ betrachten. Man differenziert zwischen der Architektur (M, C, F)

[18] Bzw. der $n \times n$ Gewichtsmatrix mit $(i, j) \notin C \to w_{ji} = 0$.

[19] Diese Notation erleichtert zwar die Vorstellung, in den meisten Implementationen werden die Schwellwerte Θ_i aber ersetzt durch ein zusätzliches Neuron, das konstant 1 ausgibt und mit dem Gewicht $-\Theta_i$ an Neuron i angeschlossen ist.

eines neuronalen Netzes (Abb. 3.6) und den zugehörigen Parametern (w, Θ).
Detailierte Informationen zu Neuronalen Netzen sind in diversen Einführungen nachzulesen, vgl. bspw. [Spe03]. Wichtig ist für uns, dass unter leichten Bedingungen jede stetige mehrdimensionale Funktion von einem neuronalen Netz mit $p \geq 2$ Schichten und einer bestimmten Zahl Neuronen durch Veränderung der Parameter beliebig genau angenähert werden kann (vgl. [Roj93]). Abgesehen davon, dass die Neuronenzahl schnell extrem groß werden kann, ist das Verhältnis zwischen Wiedergabegenauigkeit und Verallgemeinerung abzuwiegen. Große Netze sind besonders für das sogenannte Overfitting anfällig: Es werden zwar die Trainingsdaten sehr genau wiedergegeben, durch die entstehende hohe Varianz, aber der Fehler auf neuen Daten erhöht. Andererseits kann ein zu kleines Netz auch nur simple Funktionen darstellen.
Die in dieser Arbeit verwendeten Netzarchitekturen wurden experimentell gefunden; leider gibt es kein Patentrezept und auch die automatische Bestimmung bspw. mit lokaler Suche ist bei solch komplexen Problemen nach Ermessen des Autors noch nicht praktikabel[20]. Als Aktivierungsfunktion wurde für alle Neuronen, die nicht in der Eingabeschicht M_0 liegen, die logistische Standard-Funktion gewählt:

$$f_i(v) = \begin{cases} v & i \in M_0 \\ \frac{1}{1+e^{-v}} & \text{sonst} \end{cases} \tag{3.2}$$

Die Parameter des Netzes, $(w, \Theta)_t = \vec{\Phi}_t$ zur Zeit t, werden im Lernverlauf mit Hilfe des Backpropagation-Algorithmus angepasst. Dessen Grundidee ist, für eine zu approximierende Funktion $f : \mathbb{R}^I \to \mathbb{R}^O$ die nur durch Beispiele $(i, o), i \in \mathbb{R}^I, o \in \mathbb{R}^O$ bekannt ist, die komponentenweise Abweichung der Netzausgabe $K(i)$ von o zu verkleinern, und zwar graduell mit einer Lernrate $\alpha > 0$.
Für ein Netz K mit Eingabeneuronen M_0, $|M_0| = I$, und Ausgabeneuronen M_p, $|M_p| = O$, ist dies der quadratische Fehlerterm

$$E^{(i,o)}(w) = \frac{1}{2} \sum_{j \in M_p} (K_j(i) - o_j)^2.$$

[20] Siehe aber [SM02] für einen Versuch mit genetischen Algorithmen.

Man beachte, dass wir hier nicht direkt den Gesamtfehlerterm über alle möglichen Datenpaare verringern, sondern dies nur für ein (i, o) tun. Dies hat den Effekt, dass der summarische Fehlerterm nicht zwangsläufig in jedem Schritt kleiner wird; dafür kann man aber eventuell lokale Minima wieder verlassen. Auch hier möchte ich wieder auf die Literatur verweisen, wo sich ausführliche Darstellungen und spezielle Verfahren finden lassen, bspw. [RB94].

Im nächsten Kapitel werden nun die vorgestellten Methoden kombiniert, um konkret in der komplexen Soccer Server Umgebung ein Spielverhalten lernen zu können.

4 Angewendetes Lernverfahren

Für die Durchführung eines Lernvorgangs wird zunächst das Verhalten der Agenten zufällig initialisiert, woraufhin solange Lernepisoden iteriert werden bis ein zufriedenstellendes Ergebnis erreicht ist oder uns die Rechenzeit ausgeht. Eine Episode besteht aus einem Erfahrungsteil, in dem Erfahrungen bei Interaktionen mit der Umwelt gesammelt werden und einem Lernteil, bei dem das Verhalten der Agenten den Erfahrungen entsprechend angepasst wird. Der Erfahrungsteil der Episoden gliedert sich wiederum in Trajektorien. Eine Trajektorie beginnt mit der Initialisierung des Systems in einem Startzustand und endet, wenn ein Terminalzustand erreicht wurde. Zu Beginn einer Episode laden alle eigenen Spieler die Parameter und insbesondere die bisherige Approximation der gemeinsamen Bewertungsfunktion. Dann werden Trajektorien erzeugt bis genügend Daten vorhanden sind, woraufhin mit dem Anpassen der Bewertungsfunktion die Episode beendet wird.
Diese Steuerung des Lernvorgangs übernimmt ein sogenannter "offline coach" (Trainer)[1]. Der Trainer setzt also je Trajektorie den Soccer Server in den gewünschten Startzustand und führt am Ende der Episode die Aktualisierung der Bewertungsfunktion, das Lernen an sich, durch.
Die Lerndaten werden allerdings von den Spielern gespeichert. Da einerseits allen Agenten dieselben Kosten entstehen und wir daher nur eine Bewertungsfunktion brauchen, andererseits aber die Zustandsabstraktion mit Polarkoordinaten benutzen wollen, müssen alle Spieler die externen Informationen konsistent umwandeln. Um dies

[1] "Offline" da dieser aktiv ins Spiel eingreifen kann um bspw. alle Objekt-Positionen zu verändern, was natürlich nur zu Test- und Trainingszwecken erwünscht ist, keinesfalls aber bei offiziellen Turnieren.

zu erreichen, erfolgt die Umwandlung immer relativ zum Agenten im Ballbesitz[2].
In die Zustandsabstraktion (vgl. Abschnitt 3.4) gehen neben Ball und Tor nur die Daten von n Mit- und m Gegenspielern ein, um die Lernkomplexität zu verringern. Dies geschieht unter der Annahme, dass die übrigen Spieler keinen starken Einfluss auf den Ablauf einer Trajektorie haben, bspw. wegen zu großer Entfernung zum Ort des Geschehens. Von hier aus ist es auch kein großer Schritt mehr, unter derselben Annahme überhaupt nur die Agenten zu starten, die auch repräsentiert werden - für kleine n und insbesondere m kann die Annahme der Nichtbeeinflussung natürlich nicht aufrecht erhalten werden. Da aber jeder gestartete Agent Entscheidungen trifft und damit Rechenzeit verbraucht nehmen wir dieses Risiko in Kauf.
Normalerweise läuft der Soccer Server wie oben beschrieben mit stochastischen Zustandsübergängen und die Spieler nehmen den Systemzustand nur partiell und verrauscht wahr. Auf Grund begrenzter Rechenresourcen können wir aber bei der Kostenvorhersage trotz bekanntem Systemmodell und bekannter Verteilung der Stochastik, nicht alle möglichen Folgezustände berücksichtigen. Ohne Berücksichtigung zumindest vieler der möglichen Folgezustände machten uns beim Lernen aber evtl. stochastische Ausreisser das Leben schwer. Für die Dauer des Lernvorgangs wird daher das Verrauschen der Zustandsübergänge abgestellt und beiden Mannschaften die volle Zustandsinformation zur Verfügung gestellt[3]. Da den Agenten die von den Mit- und Gegenspielern gewählten Aktionen nicht bekannt sind werden bisweilen dennoch Modellfehler zugelassen[4]. Zu diesem Thema wird unten noch mehr zu sagen sein. Zunächst kommen wir jedoch zu einer konkreteren Beschreibung des Verfahrens.

[2]Was für korrektes Verhalten vorraussetzt, dass immer nur ein Agent im Ballbesitz sein kann. Aufgrund der weiter unten beschriebenen Vermeidungstaktik trifft dies in den allermeisten Fällen zu. Wenn der Ball während eines Passes unterwegs ist, wird der künftige Empfänger als Ballbesitzer simuliert, s.u. für Details.

[3]Wir lernen also im MMDP, nicht in einem POMDP (vgl. Abschnitt 2.8).

[4]Womit wir in die Fußstapfen von [Mei00] treten, der mit diesem Vorgehen gute Ergebnisse erzielt hat.

4.1 Pseudocode

Algorithmus Agenten:

```
i=0;
n=0;
Φ⃗_0=load();
forever do
      if(loadAdvise())then
      begin //new episode
           i=i+1;
           n=0;
           Φ⃗_i=load();
      end
      s_n=percieve();
      a_n = selectAction(s_n, Φ⃗_i);
      memorize(s_n, a_n);
      n=n+1;
done
```

Algorithmus Trainer:

```
Φ⃗_0=random();
i=0; //start players
while not(abort()) do
      s=initSystem(S^0);
      while not (trajEnde(s)) do
           s=percieve();
      done
      if(episodeFull())then
      begin
           collectData();
           Φ⃗_{i+1}=update(Φ⃗_i);
           adviseLoad(Φ⃗_{i+1});
           i=i+1;
      end
done //shutdown players
```

4.2 Die Methode initSystem

Wie erwähnt kann der Trainer die Eigenschaften aller Objekte im Soccer Server manipulieren. Bei Aufruf dieser Methode werden alle beteiligten Spieler und der Ball in einen Zustand aus der Menge der Startzustände S^0 gebracht. Unten werden wir das Lernverhalten bei Präsentation immer desselben Startzustandes ($|S^0| = 1$) vergleichen mit vielen verschiedenen Startzuständen ($|S^0| = \infty$). Natürlich wird auch in ersterem Fall meist nur der Startzustand derselbe sein. Die Chancen, im weiteren Verlauf noch einmal exakt denselben Zustand zu erleben, sind verschwindend gering.

4.3 Die Methode `trajEnde`

Diese Methode prüft, ob der aktuelle Zustand einen Endzustand darstellt. Ist dies der Fall, also $s_n \in S^+ \cup S^-$, wird die Trajektorie beendet. S^+ besteht aus allen Zuständen mit dem Ball im Tor. S^- ist die Menge der Misserfolge, wenn also der Ball an den Gegner verloren oder eine festgelegte maximale Entfernung zum gegnerischen Tor überschritten hat. Der Ball zählt als verloren, wenn kein Mitspieler, dafür aber ein Gegenspieler den Ball in seinem Aktionsradius hat. Weiterhin wurde eine Trajektorie nach 100 Zeitzyklen beendet, um Zeit zu sparen, denn diese Zeitspanne ist mehr als ausreichend um das Tor zu erreichen.

4.4 Die Methode `update`

Der Trainer sammelt nach einer Episode die Zustands/Aktionspaare, die von dem jeweils im Ballbesitz befindlichen Spieler gespeichert wurden[5]. Jede Trajektorie beginnt mit einem Startzustand und setzt sich fort bis ein Endzustand erreicht wurde. Welcher Spieler im Ballbesitz war, ist nicht relevant. Die Kosten für einen jeden beobachteten Endzustand definieren wir als 1 für nicht erfolgreiche Terminalzustände bzw. 0 im Erfolgsfall. Die vorangehenden Zustände werden an dem Endergebnis beteiligt, und zwar nimmt mit zunehmendem Abstand die Verantwortung ab. Formal ergibt sich für eine Trajektorie mit n Makroaktionen $(s_1, ..., s_n)$

$$v(s_i) = \begin{cases} 0 & s_i \in S^+ \\ 1 & s_i \in S^- \\ |v(s_n) - \rho * d(i,n)| & \text{sonst} \end{cases} \tag{4.1}$$

mit dem Diskontierungsfaktor ρ und $d(i,n)$ dem Abstand zwischen i und n. Naheliegend ist hier natürlich $d(i,n) = n - i$ zu setzen, was aber nicht selbstverständlich ist.

[5]Da die Spieler die vollständige Weltinformation haben, reicht es, die Daten des Ballbesitzers zu betrachten.

In diesem Fall erzeugt jede Makroaktion gleich hohe "Kosten"[6]. Es könnte aber auch von Vorteil sein, die tatsächlich abgelaufenen Systemzeitschritte zu verwenden. Ersteres begünstigt die Verwendung langwieriger Aktionen wie Pässe, letzteres schnelle Erfolge; weiter unten werden die Auswirkungen auf den Lernvorgang untersucht.
Der Diskontierungsfaktor ρ muss in Abhängigkeit von der zu erwartenden Trajektorienlänge $E(n)$ gewählt werden um auszuschliessen, dass frühe Zustände einer negativen Trajektorie kleinere und bessere Werte zugewiesen bekommen als solche einer positiven Trajtektorie.
Für das Training des Neuronalen Netzes K werden dann als Lerndaten die Tupel $(s_i, v(s_i))$ benutzt, wobei nach Präsentation der Beispiele wie oben beschrieben (Abschnitt 3.6) die bisherige Bewertung $K(s_i)$ graduell zu $v(s_i)$ verschoben wird[7]. Die Lernrate α, welche die Schrittweite der Anpassung angibt, ist ein wichtiger Parameter des Lernalgorithmus, auf den wir im nächsten Kapitel noch zurückkommen werden.

4.5 Die Methode `selectAction`

Die Methode `selectAction` wird zwar jeden Zeitzyklus erneut aufgerufen. Aber natürlich ist von dieser Makrostrategie nicht jedesmal eine Entscheidung zu treffen, da sich Makroaktionen i. A. über längere Zeiträume erstrecken. Eine gewählte Makroaktion wird einfach jeden Zyklus ausgerufen bis sie Vollzug zurückmeldet.
Wie aber erfolgt die Auswahl unter den Makroaktionen? Da wir bei weitem nicht jedes Zustand/Aktions Paar vielfach besuchen können, bemühen wir uns, bereits gemachte Erfahrungen auszunutzen und dadurch den Zustandsraum effizienter zu erforschen: Indem die gewählte Strategie von der bislang gelernten Bewertungsfunktion abhängt

[6] Dies ist auch der Grund warum der Ballbesitzer und nicht der Trainer die Zustandsinformationen zum Lernen speichert. Der Trainer hat zwar die volle Zustandsinformation, weiss aber nicht, wann ein Spieler eine Markoaktion beginnt. Der Ballbesitzer weiss dies zwar auch nicht von seinen Mitspielern, aber deren Aktionen lassen wir außen vor.

[7] Allerdings wählen wir hier statt einem Einzel- oder summarischem Fehlerterm einen Mittelweg. Die Abweichung einer grösseren Menge von Zustandswerten wird bei jedem Lerndurchgang verkleinert, was einerseits stochastische Ausreisser ermöglicht und andererseits können die Daten in endlicher Rechenzeit erzeugt werden.

und bevorzugt solche Aktionen wählt, die geringe Folgekosten versprechen[8], verbessern wir hauptsächlich die Vorhersagegenauigkeit für erfolgversprechende Regionen des Zustandsraumes. Dies geht unter Umständen zu Lasten anderer Teile des Zustandsraumes. Wenn wir solche Konstellationen aber auch im Spiel selten bis nie erreichen, ist das nicht weiter problematisch für die Gesamtperformanz.
Man erinnere sich an das Verfahren für die optimale Strategie (Gl. 2.8), wo nur Aktionen mit zu erwartenden minimalen Folgekosten eine Auswahlwahrscheinlichkeit grösser Null erhielten:

$$\pi^*(a|s) > 0 \Leftrightarrow a \in \arg\min_{a \in A} \sum_{s' \in S} p(s'|s,a)(r(s,a) + \gamma V^*(s')), \forall s \in S$$

Wegen des kontinuierlichen Zustandsraumes wäre hier sogar das Integral zu bilden:

$$\arg\min_{a \in A} \int_{s' \in S} p(s'|s,a)(r(s,a) + \gamma V^*(a(s')))$$

Glücklicherweise haben in der Soccer Server Umgebung aber nur wenige Folgezustände nennenswerte Auftretenswahrscheinlichkeiten, weshalb die Annäherung durch eine Summe gute Ergebnisse liefern kann.
Je mehr Folgezustände wir berücksichtigen, desto genauer ist natürlich das Resultat aber auch desto aufwändiger die Berechnung. Weiterhin sind uns die Übergangswahrscheinlichkeiten $p(s'|s,a)$ nicht genau bekannt; wir kennen zwar das physikalische Modell der Umwelt, Unsicherheit bringen aber Gegnermodelle und die Ausführung der Makroaktionen.
Idealerweise sollte das Verhalten der Gegenspieler mit einem parametrisierten Modell vorhergesagt werden, wobei die Parameter auf früheren Erfahrungen basieren könnten oder während des Spiels angepasst würden. Zunächst kämen dann in jedem Zustand s für einen Spieler alle Aktionen gleichberechtigt in Frage, nach einigen Beobachtungen ist aber die Wahrscheinlichkeit zunehmend höher, dass dieser Spieler bei

[8]Nicht aber ausschliesslich, während des Lernvorgangs soll jede Aktion in jedem Zustand weiter eine Chance haben, ausgewählt zu werden!

Wiederholung von s dieselbe Aktion erneut wählen würde[9]. Die Komplexität eines solchen Ansatzes ist nicht zu unterschätzen (pro Zustand eine Aktion merken oder gar eine Wahrscheinlichkeitsverteilung über A). Deshalb gehen wir im hier verwendeten Ansatz stark vereinfachend davon aus, dass ein gutes deterministisches Modell ausreichend genau das Verhalten wiederspiegelt; in Ermangelung besserer Modelle wird unser eigenes Brainstormer Torwart- und Verteidiger-Verhalten zur Vorhersage verwendet[10].

Das zweite Problem ist die Ausführung der Makroaktionen. Trotz Prüfung enden diese nicht immer wie gewünscht. Leider haben wir aber keine Wahrscheinlichkeitsverteilung über die möglichen Endzustände s' bei Ausführung von Aktion a, $p_a(s'), s' \in S^+ \cup S^-$ zur Verfügung, weshalb nur ein Zustand nach erfolgreicher Ausführung berücksichtigt wird.

Wie [Wit04] werden auch wir uns darauf alleine nicht verlassen und zum Vergleich eine weitere Version probieren. Auch hier wird ein Modell verwendet, allerdings nur um den Zustand einen Zeitzyklus nach Beginn der Aktion vorherzusagen. Auf so kurze Zeit wirkt sich hauptsächlich die Physik des Soccer Server aus, die uns bekannt ist. Diese Vorhersage werden wir im Folgenden als "modellfrei" bezeichnen.

Die hier verwendete Lösung berücksichtigt also in jedem Falle genau einen Folgezustand bei Verwendung von Aktion a; wir bezeichnen die Vorhersagefunktion als $m_a : S \to S$, die Übergangswahrscheinlichkeit erübrigt sich (wir nehmen an, dass $p(m_a(s)|s, a) = 1$). Aktionen a, für die das Modell aus dem aktuellen Zustand s einen Ballverlust vorhersagt, werden keiner weiteren Bewertung unterworfen sondern direkt aussortiert, $m_a(s) \in S^- \to \pi(a|s) = 0$.

Die Menge der im Zustand s minimale Folgekosten bezüglich der Bewertungsfunktion $V_{\vec{\Phi}}$ verheissenden Aktionen $A^{\vec{\Phi}}_{\min}(s)$ definieren wir als

$$A^{\vec{\Phi}}_{\min}(s) = \arg\min_{a \in A(s)} (r(s, a) + \gamma V(a(m_a(s)))), \tag{4.2}$$

[9] Die Sicherheit wird aber auch bei deterministischen Strategien nie 1, da wir bei eingeschränkter Weltsicht nicht sicher sein können, dass dieser Agent denselben Zustand wahrnimmt wie wir.

[10] Leider musste die exakte Modellierung auf fünf Zeitzyklen beschränkt werden, um die Echtzeitbedingungen des Soccer Server zu erfüllen. Pässe dauern oftmals länger, in solchen Fällen wurde in der Zustandsrepräsentation nur noch der Ball zu dem Zielspieler verschoben und der Torwart auf eine Position, die einen direkten Torschuss unmöglich macht.

wobei für die Kosten $r(s,a)$ hier das gleiche wie in Gl. 4.1 gilt. Entweder wir berechnen für jede Makroaktion gleiche direkte Kosten ρ oder multiplizieren ρ mit der (geschätzten) Aktionsdauer in Systemzeit.
Damit ergibt sich die fixe, nach dem Lernvorgang zu verwendende, Strategie als[11]:

$$\pi(a|s) = \begin{cases} \frac{1}{|A^{\vec{\Phi}}_{\min}(s)|} & a \in A^{\vec{\Phi}}_{\min}(s) \\ 0 & \text{sonst} \end{cases} \tag{4.3}$$

Wie bereits erwähnt soll aber während des Lernens exploriert werden. Also muss jede Aktion immer zumindest eine kleine Chance haben, ausgewählt zu werden. Wir werden zwei Explorationsstrategien unterscheiden und später vergleichen. Die Explorationsrate, also die Wahrscheinlichkeit, dass nicht eine der bestbewerteten Aktionen gewählt wird, bezeichnet man mit ϵ. Diesem Umstand verdankt die erste Strategie ihren Namen, ϵ-greedy Auswahl:

$$\pi(a|s) = \begin{cases} \frac{1-\epsilon}{|A^{\vec{\Phi}}_{\min}(s)|} & a \in A^{\vec{\Phi}}_{\min}(s) \\ \frac{\epsilon}{|A(s) \setminus A^{\vec{\Phi}}_{\min}(s)|} & \text{sonst} \end{cases} \tag{4.4}$$

Die vermeintlich nicht-optimalen Aktionen haben alle dieselbe Wahrscheinlichkeit, ausgewählt zu werden – unabhängig davon, ob sie die zweitbeste oder schlechteste Bewertung haben. Bei der softmax Auswahl werden die Auswahlwahrscheinlichkeiten hingegen in Abhängigkeit von ihren geschätzten Werten vergeben[12]:

$$\pi(a|s) = \begin{cases} \frac{1-\epsilon}{|A^{\vec{\Phi}}_{\min}(s)|} & a \in A^{\vec{\Phi}}_{\min}(s) \\ \epsilon * \frac{1-(r(s,a)+\gamma V(a(m_a(s))))}{\sum_{a' \in A(s) \setminus A^{\vec{\Phi}}_{\min}(s)} (1-(r(s,a')+\gamma V(a(m_{a'}(s)))))} & \text{sonst} \end{cases} \tag{4.5}$$

Die Wahrscheinlichkeit hängt hier umgekehrt proportional von den geschätzten Folgekosten ab.

[11] Der Einfachheit halber werden alle besten Aktionen mit gleicher Wahrscheinlichkeit gewählt, natürlich sind Strategien mit $\pi(a|s) > 0 \leftrightarrow a \in A^{\vec{\Phi}}_{\min}(s)$ genauso gut. Man könnte dann noch Vorwissen einbringen, wie: Im Zweifelsfall lieber den Ball nach vorne bringen!

[12] Wobei die hier angegebene nur eine von vielen möglichen Varianten ist.

Bei den empirischen Untersuchungen werden wir noch darauf zu sprechen kommen, welche ϵ-Werte Sinn machen.
Anzumerken sind noch einige Sonderfälle: So wird, wenn der Ballbesitzer sich für einen Pass entscheidet, dies über den Soccer Server kommuniziert. Dann wählt der Empfänger nicht mehr frei Aktionen aus, sondern bleibt stehen und nimmt den Ball an, während die übrigen Spieler für ihre Positionswahl nicht den aktuellen Zustand zugrunde legen, sondern bewerten, welche Positionen erfolgversprechend sind, wenn der Ball angekommen sein wird. Dabei handelt es sich strenggenommen also nicht um unabhängiges Aktionslernen, da den Spielern bis zu einem gewissen Grade bekannt ist, was die anderen machen werden.
Spieler ohne Ball ignorieren solche Aktionen ohne weitere Prüfung, die sie so nah an die Position eines anderen Spielers bringen würden, dass sich die Bereiche, in denen sie den Ball kontrollieren könnten, überlappen. Dies trägt zur Vermeidung von Situationen bei, in denen mehr als ein eigener Spieler im "Ballbesitz" ist.
Weiterhin geht, wenn der Ball nicht unter der Kontrolle eines Spielers ist, er also bspw. in Folge eines Fehlpasses frei herum rollt, automatisch der am nächsten befindliche Spieler zum Ball.
Last but not least wird vom Ballbesitzer in jedem Zyklus die Prüfungsroutine der bereits vorhandenen Makroaktion `schiesseTor` ausgerufen und, im Falle eines positiven Ergebnisses, der Torschuss ausgeführt. Da sich bei Tests ergab, dass eine positive Prüfung in über 90 Prozent der Fälle einen Torerfolg nach sich zieht, ersparen wir uns, hier etwas neues lernen zu wollen.

4.6 Die übrigen Methoden

Da es eine gemeinsame Bewertungsfunktion für alle Agenten gibt und der Trainer das Lernen durchführt, müssen die aktuellen Parameter des Funktionsapproximators irgendwie an die Spieler verteilt werden. Daher ist `adviseLoad` als Nachricht des Trainers zu verstehen, dass neue Parameter zu laden sind, was von den Spielern mit

`loadAdvise` wahrgenommen und mit `load` ausgeführt wird.
Auch die verbleibenden Methoden sind fast selbsterklärend:

- `random` initialisiert Gewichte und Schwellwerte des neuronalen Funktionsapproximators (im Bereich [-0.7,0.7]),
- `percieve` liefert den aktuellen Systemzustand zurück,
- `memorize` speichert Aktion, Zustand und Zeitpunkt,
- `collectData` sammelt die via `memorize` gespeicherten Daten ein,
- `episodeFull` prüft, ob wir die gewünschte Anzahl an positiven wie negativen Trajektorien erzeugt haben um zu lernen,
- und `abort` beendet den Lernablauf, wenn wir zufrieden sind oder keine Rechenzeit mehr haben.

5 Empirische Untersuchungen

Wie eingangs angekündigt, wollen wir verschiedene Möglichkeiten des Lernens in der Soccer Server Umgebung erproben.
Jeder Lernverlauf wurde über 100 Episoden durchgeführt. Bei allen Versuchen wurden drei Angreifer sowie drei Verteidiger plus Torwart repräsentiert, also 14 Eingabewerte plus je zwei für Polarkoordinaten zum Tor und zum Ball. Die Bewertungsfunktion $V : \mathbb{R}^{18} \rightarrow \mathbb{R}$ wurde durch eine 18-36-18-9-1 Netzarchitektur[1] approximiert, da sich in einigen Experimenten trotz der Komplexität von drei versteckten Schichten leichte Vorteile zeigten. Weitere Tests wären nötig, um jeden Zufall auszuschliessen.

5.1 Statistik

Um den Lernfortschritt verfolgen zu können, wurde für die jeweils ersten zehn Episoden und danach noch für jede zehnte Episode die Erfolgsrate der bis dahin gelernten Strategie festgestellt. Hierfür wurden genau 100 Trajektorien gespielt (allerdings ohne Exploration).
Als Indikator, ob und wie gut gelernt wurde, dienen die Erfolgsraten von Referenzstrategien. Referenzstrategie 1 ist die gleichverteilt zufällige Auswahl von Aktionen aller lernenden Spieler, also für Agent i die Strategie $\pi_i(a|s) = |A_i(s)|^{-1}$. Zweite Vergleichsstrategie ist das Verhalten der Brainstormer Agenten von der RoboCup German Open 04 Meisterschaft (GO04).

[1] Wie oben beschrieben (vgl. Abschnitt 3.6) handelt es sich um ein vorwärstgerichtetes, geschichtetes Neuronales Netz. Die fünf Zahlen geben die Anzahl der Neuronen in der jeweiligen Schicht an.

Als Verteidiger wurden die vorerwähnten Brainstormer GO 04 Agenten und teilweise die Trilearn Agenten[2], ebenfalls in der GO04[3] Version, verwendet.

5.2 Parameter

Pro Episode wurden zehn positive sowie zehn negative Trajektorien zum Lernen verwendet. Das heisst, es wurden solange Trajektorien gespielt, bis genügend Beispiele zum Lernen gesammelt waren. Aus Zeitgründen wurde aber in jedem Fall nach 200 Trajektorien abgebrochen, auch wenn noch nicht genügend Erfolge erzielt worden waren.
Natürlich wollen wir möglichst viele Lerndaten; da sich aber bei Versuchsreihen auch ergab, dass eine etwa ausgeglichene Anzahl von erfolgreichen und nicht erfolgreichen Trajektorien die Stabilität des Lernverlaufs unterstützen, wurden überzählige negative Trajektorien ignoriert. Die Zahlen sind so gewählt, dass mindestens fünf Prozent der Trajektorien zu einem Tor führen mussten um soviele positive Daten wie gewünscht zu haben, was meist gelang.

Klare Vorteile zeigten sich für die Abstandsfunktion $d(i, n) = n - i$ (vgl. Gl. 4.1), also die Kosten für jede Makroaktion gleich zu wählen.Wurden die Kosten hingegen anhand der tatsächlichen Aktionsdauer in Systemzeit berechnet, hatten Pässe kaum eine Chance und es wurde eher ein “Einzelkämpfer“-Verhalten gefördert.

Lernrate

Zunächst wurden Experimente mit einer konstanten Lernrate α durchgeführt. Für kleine α (etwa 0.05) ging der Lernvorgang recht schleppend vonstatten, für große α (etwa 0.2) war das Netz aber recht instabil. Diese Instabilitäten zeigten sich durch stark variierende Erfolgsraten und wurden von einem Umlernen des Netzes begleitet

[2] An der Universität Amsterdam entwickelt, vgl. http://staff.science.uva.nl/ jellekok/robocup/.
[3] Bei dieser Meisterschaft konnte Trilearn den ersten und die Brainstormers den dritten Platz belegen.

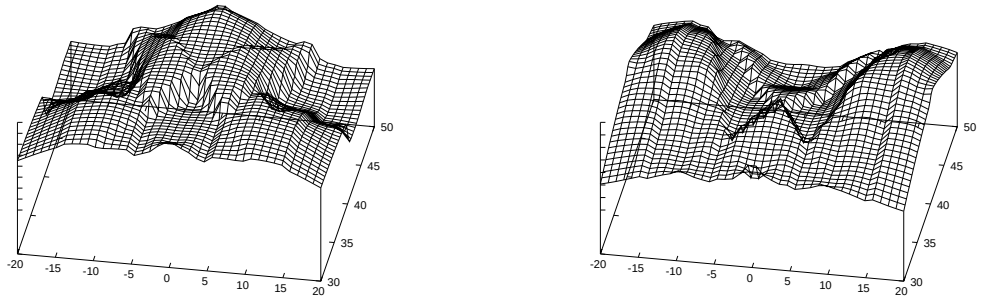

Abbildung 5.1: *Eine Bewertungsfunktion; links vor, rechts nach dem Umlernen (geglättet).* Zur Veranschaulichung der im Text beschriebenen Lerninstabilitäten. Natürlich können wir nur zwei Dimensionen visualisieren; gezeigt ist jeweils der Funktionswert bei Verschiebung des Ballbesitzers während die anderen Spieler wie in der Standard-Situation (Abb. 5.2) festgehalten werden.

(ein Beispiel zeigt Abb. 5.1). Erfolgreicher war der Mittelweg einer zeitabhängigen Lernrate α_t, die immer kleiner wird und somit den einmal eingeschlagenen Weg nicht wieder verlässt, sondern nur noch verfeinert. Dieses Vorgehen erhöht natürlich die Abhängigkeit des Lernverlaufs von der anfänglichen Initialisierung der Netzparameter und den ersten Lerndaten.
Der Wert von α wurde nach jeder Epoche um 2% verringert, also:

$$\alpha_t = \begin{cases} \mathsf{c} & t = 0 \\ \alpha_{t-1} * 0.98 & \text{sonst} \end{cases} \tag{5.1}$$

Verwendet wurde $\mathsf{c} = 0.2$ als initialer Wert.

Exploration

Von den beiden oben beschriebenen Auswahlstrategien, die während des Lernvorgangs jeder Aktion in jedem Zustand eine gewisse Auswahlwahrscheinlichkeit zuordnen, stellte sich die softmax Auswahl (vgl. Def. 4.5) als etwas erfolgreicher heraus.

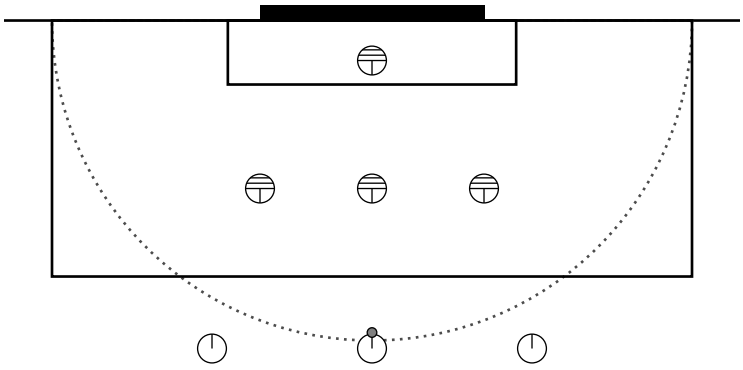

Abbildung 5.2: *Standard-Startsituation 3 gegen 3 plus Torwart.* Die Stürmer sind 20.5 Meter von der Torlinie entfernt, der Ball 19.5 Meter. Die Verteidigungslinie steht bei 42 Metern, in 10.5 Meter Entfernung vom Tor.

Allerdings war der Vorteil gegenüber der ϵ-greedy Strategie (Def. 4.4) nicht signifikant. Dennoch wurde im weiteren Verlauf die den geschätzten Kosten umgekehrt proportionale softmax Aktionsauswahl benutzt.
Bei Explorationsraten ϵ um 0.1 liessen sich teilweise gute aber teilweise auch schlechte Ergebnisse erzielen was auf eine erhöhte Abhängigkeit von den initialen Bedingungen schliessen lässt. Daher wurde die Explorationsrate auf 0.2 festgelegt. Es könnte sinnvoll sein, diese ähnlich der Lernrate mit der Zeit zu verkleinern, um die Stabilität weiter zu erhöhen.

5.3 Eine Startsituation

Bei dieser Versuchsreihe traten drei Angreifer gegen drei Verteidiger plus Torwart an. Gestartet wurde jedesmal wieder aus derselben Startsituation s_0 (vgl. Abb. 5.2), also $S^0 = \{s_0\}$. Es handelt sich um eine schwierige Ausgangslage, denn die Verteidigung steht schon gut organisiert und auf jeden Stürmer kommt ein Gegenspieler. Die Erfolgsraten der Refernzstrategien sind in Tabelle 5.1 aufgelistet. Da sich die Lernstrategie zu Beginn kaum anders verhält als die "zufällige Aktionswahl"[4], war nicht

[4]Man beachte, daß nur jene Aktionen zur Auswahl stehen, die nicht von vornherein aussortiert wurden (vgl. Abschnitt 4.5). Auch die bei `selectAction` beschrieben Sonderregeln finden Anwendung, so wird bspw. in jedem Zeitzyklus die Möglichkeit eines Torschusses geprüft. Das erklärt auch,

Paarung	Erfolgsquote
Zufall vs. Brainstormers GO04	2.3%
Zufall vs. TriLearn GO04	2.7%
Brainstormers GO04 vs Brainstormers GO04	31.3%
Brainstormers GO04 vs Trilearn GO04	35.4%

Tabelle 5.1: *Erfolgsquoten mit Referenzstrategien.* 3 gegen 3 plus Torwart aus der Standard-Startsituation (Abb. 5.2), je 1000 Trajektorien

damit zu rechnen, dass hier in den ersten Episoden nach 200 Trajektorien schon 10 Erfolge auftreten würden. Tatsächlich war dies nicht der Fall und die unausgewogenen Lerndaten führten teilweise zu sehr unvorhersehbaren Lernverläufen. Zur Abhilfe wurde eine Bewertungsfunktion mit einer ähnlichen Startsituation trainiert, wobei aber ein Verteidiger weniger gestartet wurde[5]. Diese Ausgangslage ist einfacher zu lösen, das Brainstormers GO04 Verhalten erzielt hier Erfolgsraten um 50%. Eine solcherart vortrainierte Bewertungsfunktion wurde dann als Startpunkt für die ursprüngliche "3 gegen 3" Situation verwendet. Natürlich wird hier zunächst ein Gegenspieler komplett ignoriert, da er für den vorigen Lernvorgang unerheblich war, aber zumindest können damit sofort in über fünf Prozent der Fälle Erfolge erzielt werden.
Um das Lernverfahren zu testen, lernt (und exploriert) zunächst nur der Ballbesitzer, während seine Mitspieler das Brainstormers GO04 Verhalten verwendeten. Erwartungsgemäß war hier die Erfolgsrate etwas höher als in dem Szenario wo alle lernen, da unter anderem die Freilauf-Strategie der Mitspieler ohne Ball sehr ausgefeilt ist. Auffällig ist noch, dass der Lernerfolg in beiden Fällen für die Minimal-Modell Version besser ist, besonders deutlich jedoch hervortritt, wenn alle Angreifer lernen. Die Ergebnisse fasst Abbildung 5.3 zusammen.

warum von dieser Strategie überhaupt Tore geschossen wurden.

[5]Dessen Repräsentation in der Zustandsbewertung wurde auf die maximale Entfernung, 20 Meter, gesetzt.

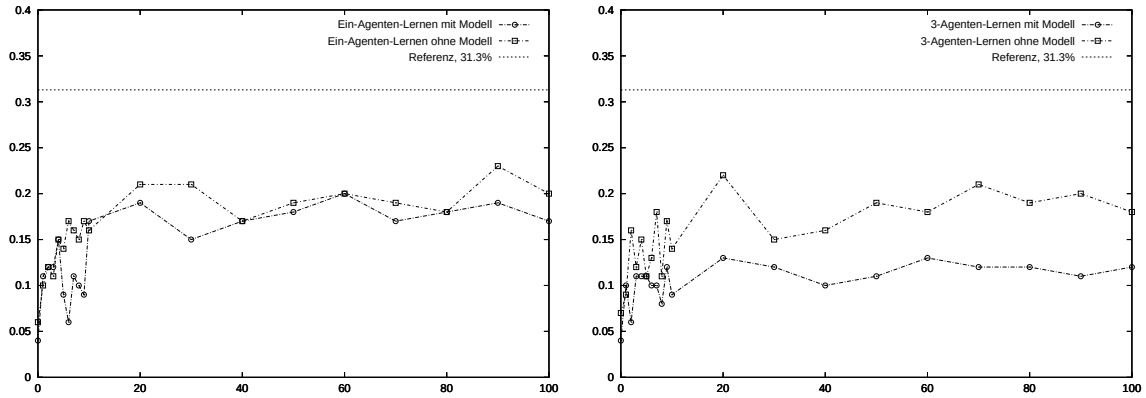

Abbildung 5.3: *Erfolgsrate während exemplarischer Lernverläufe, eine Startsituation.* Links für einen lernenden Ballbesitzer, rechts bei lernen aller Angreifer

5.4 Viele Startsituationen

In einem normalen Spielverlauf finden sich die Spieler natürlich in vielen verschiedenen Ausgangslagen wieder und müssen aus diesen dann das Beste machen. Ausserdem sind die Verhältnisse selten so geordnet wie in der eben beschriebenen Startsituation. Daher wurde in einer weiteren Versuchsreihe mit einer Heuristik je Trajektorie ein neuer Startzustand generiert. Dabei setzte das erzeugende Verfahren die Angreifer auf eine Position, die zwischen 15 und 20 Meter vom Tor entfernt ist, jedoch nicht zu nah beieinander. Der Torwart wurde so gesetzt, dass er einen direkten Torschuss blockiert. Die Verteidiger wurden in etwa zehn Metern Entfernung zum Tor aufgestellt, mit genügend Abstand zueinander und einer kleinen zufälligen Verschiebung (vgl. Abb. 5.4).

Aus diesen Ausgangslagen kann auch ein nicht vortrainiertes Netz von Anfang an genügend positive Trajektorien erzeugen, wie die Erfolgsraten der Zufallsstrategie von ca. 15 Prozent zeigen (Tabelle 5.2). Aus der Übersichtstabelle geht auch hervor, dass die Brainstormers GO04 Verteidigung offensichtlich bei solchen Situationen eine Schwäche hat. Daher wurden die Lernverläufe mit der Trilearn GO04 Verteidigung durchgeführt.

Bei den durchgeführten Versuchen blieben die gelernten Strategien nur knapp unter

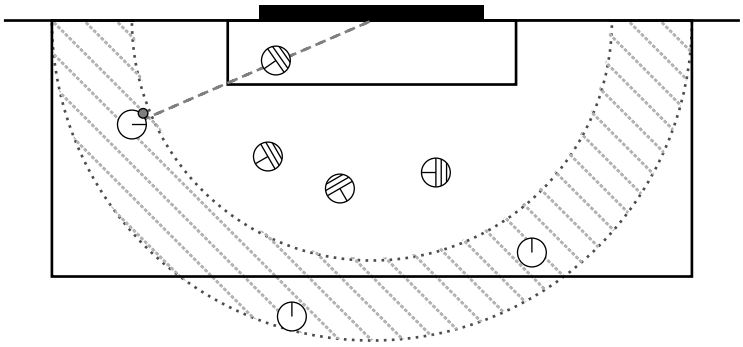

Abbildung 5.4: *Situation 3 gegen 3 plus Torwart.* Das erzeugende Verfahren setzt die Angreifer auf eine Position, die zwischen 15 und 20 Meter vom Tor entfernt ist (schraffierter Bereich), jedoch nicht zu nah beieinander. Der Torwart wird so gesetzt, dass er einen direkten Torschuss blockiert, die Verteidiger kommen auf Positionen aus $[-7, 7] \times [40, 44]$, wiederum nicht zu dicht aufeinander.

Paarung	Erfolgsquote
Zufall vs. Brainstormers GO04	16.7%
Zufall vs. TriLearn GO04	14.4%
Brainstormers GO04 vs Brainstormers GO04	43.4%
Brainstormers GO04 vs Trilearn GO04	34.0%

Tabelle 5.2: *Erfolgsquoten mit Referenzstrategien.* 3 gegen 3 plus Torwart bei generierten Startsituationen (Abb. 5.2), je 1000 Trajektorien

der Erfolgsrate der Referenzstrategie, wiederum zeigten sich Vorteile für die modellfreie Vorhersage (vgl. Abb. 5.5). Die Länge der Trajektorien und damit die Anzahl der zu treffenden Entscheidungen war hier im Durchschnitt nur etwa halb so lang wie bei der Versuchsreihe mit einer einzigen, schwierigen Startsituation.

5.5 Spiel

Natürlich war das Interesse des Autors auch, ein Verhalten zu schaffen, das praktische Vorteile unter Turnierbedinungen mit sich bringt, d.h. bei Spielen mit vollen 22 Spielern und eingeschränkter Weltinformation. Wie aus obigen Ergebnissen ersichtlich können die gelernten Verhalten (noch) nicht das analytische übertrumpfen. Mit

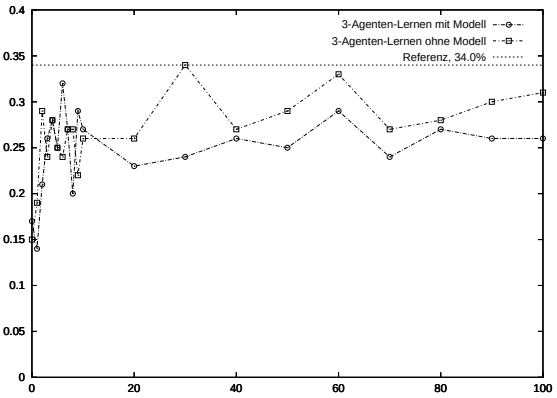

Abbildung 5.5: *Erfolgsrate während eines exemplarischen Lernverlaufs, generierte Startsituationen.*

einem Trick lässt sich das Gelernte aber doch noch erfolgreich integrieren. Die Idee ist, die erlernte Strategie zu verwenden, wenn diese zuversichtlich ist und andernfalls auf die analytisch einprogrammierte Strategie zurückzugreifen. Der Wert für die Zuversicht wurde relativ zu dem üblichen Ausgabelevel berechnet, nämlich indem die Ausgaben der Bewertungsfunktion für 100.000 zufällig erzeugte Situationen gemittelt wurden. Kommt der Soccer Server nun in einen Zustand, bei dem die Vorraussetzungen für unsere Strategie erfüllt sind (also insbesondere die Entfernung des Balls vom Tor kleiner 20 Meter ist) wird stets das gelernte Verhalten und damit die Bewertungsfunktion aufgerufen. Liefert sie für die beste Aktion erwartete Kosten größer dem ermittelten Durchschnittswert, trifft die bisherige, handprogrammierte Strategie eine Entscheidung, sonst führt unsere Strategie die zugehörige Makroaktion aus.
Hiermit liess sich ein leichter Vorteil gegenüber dem alten Verhalten zeigen; bei 400 Spielen mit ansonsten gleichem Verhalten und gleichen Vorraussetzungen für beide Mannschaften ergaben sich die in Tabelle 5.3 gezeigten Resultate. Verwendet wurde

Ausgang	Absolut	Anteil
Siege für hybrides Verhalten	80	20%
Niederlagen für hybrides Verhalten	58	14.5%
Unentschieden	262	65.5%

Tabelle 5.3: *Resultate Hybrides Verhalten vs. Brainstormers GO04.* 400 Spiele, unter Turnierbedingungen.

eine Bewertungsfunktion aus der Versuchsreihe mit jeweils zufällig generierten Startsituationen und modellfreier Vorhersage.
Auch zu diesen Ergebnissen beigetragen hat wohl die benutzte Repräsentationsstrategie für den Ballbesitzer. Da die Bewertungsfunktion nur für drei Angreifer trainiert wurde, stellte sich die Frage, was mit weiteren Mitspielern passieren sollte. Anstatt einfach die zwei nächsten zu wählen, wurden zufällig zwei aus der Menge aller Mitspieler die nicht weiter als 20 Meter vom Ballbesitzer entfernt waren, ausgewählt. Das ist zwar eine recht grobe Heuristik, die sich aber bewährt hat. Auf diese Weise ist es möglich, daß der Ballbesitzer einen Pass zu einem weiter entfernten Spieler bspw. auf der anderen Seite des Tores spielt – auch wenn um ihn herum ein "Knubbel" von Mitspielern ist.

6 Diskussion

Wie die empirischen Ergebnisse zeigen, ist Reinforcement Lernen durchaus auch in komplexen, kontinuierlichen Multiagenten-Umgebungen möglich. Allerdings ist das Ziel noch nicht erreicht worden, mit einer komplett eingelernten Makrostrategie die handkodierte Angriffsstrategie zu überbieten und zu ersetzen. Einige Gründe und eventuelle zukünftige Abhilfen werden zum Abschluss diskutiert.

Bei Betrachtung der Lernversuche fällt sofort die im Gegensatz zu anderen RL Versuchen geringe Anzahl an Lernepisoden auf. Wegen der mit der Zeit sinkenden Lernrate verändert sich zwar nach 100 Episoden nicht mehr viel; wie weit die Strategie aber noch von einem Optimum weg ist lässt sich daraus aber nicht schliessen. Will man nicht wochenlang auf ein Ergebnis warten, hilft momentan leider nur eine Verkürzung der Lernverläufe. Im Soccer Server existiert aber ein Modus, in dem das System in einen neuen Zustand übergeht, sobald alle Agenten ihre Aktionen an den Server gesendet haben. Verteilt man die Agenten dann auf viele Rechner ist eine starke Beschleunigung des Ablaufs zu erwarten.

Die Versuchsergebnisse suggerieren zwar, dass die modellfreie Vorhersage generell zu besseren Erfolgen führt. Erinnert man sich aber an die Erläuterung des Modells in Abschnitt 4.5 wird offenbar, dass wohl eher das verwendete Modell nicht adäquat ist. Da für jede Makroaktion immer nur genau ein Folgezustand berücksichtigt wird, kann es große Diskrepanzen zwischen vorhergesagten und tatsächlichen Folgekosten geben. In Zukunft könnte die Vorhersagegenauigkeit erhöht werden, indem man zumindest für die binäre Kategorisierung in Erfolg (S^+) / Misserfolg (S^-) die Auftretenswahrscheinlichkeiten $\sum_{s' \in S^+} p(s'|s,a)$ bzw. $\sum_{s' \in S^-} p(s'|s,a)$ abschätzt, bspw. durch neuronale Lernverfahren. Nennen wir diese Schätzungen $e_a^{S^+}$ und $e_a^{S^-}$, mit $e_a^{S^+}+e_a^{S^-}=1$. Von

jeder Kategorie nimmt man dann einen "typischen" Folgezustands-Vertreter s'_+, s'_- und schätzt die Zustandsbewertung $V^*(s)$ (Gl. 2.7) mit

$$min_{a \in A}\{e_a^{S^+}[r(s,a) + \gamma V^*(s'_+)] + e_a^{S^-}[r(s,a) + \gamma V^*(s'_-)]\}.$$

Natürlich gilt: Je mehr mögliche Folgezustände berücksichtigt werden und je genauer ihre Auftretenswahrscheinlichkeit bekannt ist (adaptives Gegenspielermodell), desto besser.

Das Ignorieren der Geschwindigkeiten und Blickrichtungen der Spieler bei der Abstraktion vom tatsächlichen Systemzustand (Abschnitt 3.4) ist einer Verringerung der Komplexität geschuldet. Diese Informationen sind aber unerlässlich, will man die Erfolgswerte der handprogrammierten Strategie für jede Ausgangslage überbieten. Ohne diese Daten stellen sich für die Bewertungsfunktion Zustände gleich dar, die qualitativ sehr unterschiedlich sind; so kann es bspw. für ein erfolgreiches Umlaufen eines Gegenspielers sehr wichtig sein, ob dieser gerade in meine Richtung beschleunigt oder in die Gegenrichtung[1].

Interessant wäre auch, ähnlich dem in Abschnitt 5.5 beschriebenen hybriden Verfahren, mehrere Bewertungsfunktionen zu koppeln. Dann könnte man die Bewertungsfunktion zur Aktionsentscheidung heranziehen, die für einen Folgezustand die am höchsten unter ihrem jeweiligen durchschnittlichen Ausgabewert liegende Kostenschätzung abgibt. Evtl. könnte man mit so einem Vorgehen der Beobachtung Rechnung tragen, dass sich einzelne Netze spezialisieren. Auf Grund der zufälligen anfänglichen Parameter des Bewertungsnetzes kommen sie während des Lernvorgangs immer wieder in bestimmte Regionen des Zustandsraumes und lernen dementsprechend dort besseres Verhalten.

Weiterhin ist auch immer auf die aktuelle Forschung auf theoretischem Gebiet zu hoffen: Neue Ergebnisse könnten hinreichende Kriterien für die Konvergenz des Multiagentenlernens liefern und somit manuelles ausprobieren der Parameter und die Abhängigkeit von den Startbedingungen minimieren.

[1] Um diesen Unterschied lernen zu können müsste man natürlich auch mit reduzierter Weltinformation lernen. Bei voller Weltinformation wüsste der Gegenspieler auch bei entgegengesetzter Blickrichtung, was der Angreifer macht.

Literaturverzeichnis

[Bai95] Leemon C. Baird. Residual Algorithms: Reinforcement Learning with Function Approximation. In *ICML*, pages 30–37, 1995.

[Ber95] D.P. Bertsekas. *Dynamic Programming and Optimal Control.* Athena Scientific, 1995.

[CFH+02] Mao Chen, Ehsan Foroughi, Fredrik Heintz, ZhanXiang Huang, Spiros Kapetanakis, Kostas Kostiadis, Johan Kummeneje, Itsuki Noda, Oliver Obst, Patrick Riley, Timo Steffens, Yi Wang, , and Xiang Yin. *Users Manual - RoboCup Soccer Server - for Soccer Server version 7.07 and later.* The RoboCup Federation, 2002.

[CLZ97] Anthony R. Cassandra, Michael L. Littman, and Nevin Lianwen Zhang. Incremental Pruning: A Simple, Fast, Exact Method for Partially Observable Markov Decision Processes. In *UAI*, pages 54–61, 1997.

[JJS94] T. Jaakkola, M.L. Jordan, and S.P. Singh. On the Convergence of Stochastic Iterative Dynamic Programming Algorithms. *Neural Computation*, (6):1185–1201, 1994.

[KLC98] Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and Acting in Partially Observable Stochastic Domains. *Artif. Intell.*, 101(1-2):99–134, 1998.

[LR00] Martin Lauer and Martin Riedmiller. An algorithm for distributed reinforcement learning in cooperative multi-agent systems. In *ICML*, pages 535–542, 2000.

[LR04] Martin Lauer and Martin Riedmiller. Reinforcement Learning for Stochastic Cooperative Multi-Agent Systems. In *AAMAS*, pages 1516–1517, 2004.

[MB01] Amy McGovern and Andrew G. Barto. Automatic Discovery of Subgoals in Reinforcement Learning using Diverse Density. In *ICML*, pages 361–368, 2001.

[Mei00] David Meier. *Weiterentwicklung neuronaler Reinforcement Lernverfahren am Beispiel RoboCup*. Diplomarbeit, Universität Karlsruhe (TH), 2000.

[Mer99] Artur Merke. *Reinforcement Lernen in Multiagentensystemen*. Diplomarbeit, Universität Karlsruhe (TH), 1999.

[MS04] Artur Merke and Ralf Schoknecht. Convergence of synchronous reinforcement learning with linear function approximation. In *ICML*, 2004.

[PR95] Ronald Parr and Stuart J. Russell. Approximating Optimal Policies for Partially Observable Stochastic Domains. In *IJCAI*, pages 1088–1095, 1995.

[Put94] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., 1994.

[Rab89] Lawrence R. Rabiner. A tutorial on hidden Markov models and selected application in speech recognition. In *Proc. IEEE 77*, pages 257–286, 1989.

[RB94] M. Riedmiller and H. Braun. *RPROP – Description and Implementation Details*, 1994.

[RGKS05] M. Riedmiller, T. Gabel, J. Knabe, and H. Strasdat. *Brainstormers 2D - Team Description 2005*. AG Neuroinformatik, Universität Osnabrück, 2005.

[RMN+03] M. Riedmiller, A. Merke, M. Nickschas, W. Nowak, and D. Wothof. *Brainstormers 2003: Team Description*. Lehrstuhl Informatik I, Universität Dortmund, 2003.

[Roj93] Raúl Rojas. *Theorie der neuronalen Netze: Eine systematische Einführung*. Springer-Verlag, 1993.

[Sal02] Brian Sallans. *Reinforcement Learning for factored Markov Decision Processes*. PhD thesis, Graduate Department of Computer Science, University of Toronto, 2002.

[SB98] Richard S. Sutton and A. G. Barto. *Reinforcement learning: An Introduction*. MIT Press, 1998.

[SM01] Peter Stone and David A. McAllester. An architecture for action selection in robotic soccer. In *Agents*, pages 316–323, 2001.

[SM02] Kenneth O. Stanley and Risto Miikkulainen. Efficient Reinforcement Learning Through Evolving Neural Network Topologies. In *GECCO*, pages 569–577, 2002.

[Son76] E. J. Sondik. *The Optimal Control of Partially Observable Markov Processes.* PhD thesis, Department of Engineering-Economic Systems, Stanford University, 1976.

[Spe03] Volker Sperschneider. *Skript zur Vorlesung Neuronale Netze*. Universität Osnabrück, 2003.

[Tho11] Edward L. Thorndike. *Animal Intelligence*. Macmillan, 1911.

[WD92] Christopher J. C. H. Watkins and Peter Dayan. Technical Note Q-Learning. *Machine Learning*, 8:279–292, 1992.

[Wei99] Gerhard Weiss, editor. *Multiagent systems : a modern approach to distributed artificial intelligence*. MIT Press, Cambridge, Mass., 1999.

[Wit04] Daniel Withopf. *Untersuchungen zum Reinforcement Lernen in Multi-Agenten-Systemen*. Diplomarbeit, Universität Karlsruhe (TH), 2004.

[WKMS03] Shimon Whiteson, Nate Kohl, Risto Miikkulainen, and Peter Stone. Evolving Keepaway Soccer Players through Task Decomposition. In *GECCO*, pages 356–368, 2003.

Die meisten referenzierten Artikel wurden auch im Internet veröffentlicht und sind dem Autor von dort bekannt. Obige Angaben zum Ort der Erstveröffentlichung stammen zu einem großen Teil von http://dblp.uni-trier.de

Hereby I confirm that I wrote this thesis independently and that I have not made use of any other resources or means than those indicated.

Hiermit bestätige ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Osnabrück, April 2005