Kapitel 3

Lernen in der Robotik

In diesem Kapitel gebe ich einen Überblick häufig verwendeter Lernverfahren, sowie die Anwendung von Lernmethoden im RoboCup und der Robotik im Allgemeinen. Es werden einige Arbeiten vorgestellt, die beispielhaft unterschiedliche Ansätze zeigen. Zusätzlich wird in den Kapiteln 5 bis 9 auf unterschiedliche Arbeiten eingegangen, die mit den dort vorgestellten Ansätzen in engem Zusammenhang stehen.

In einem einflussreichen Aufsatz haben Brooks und Mataric vier Domänen ausfindig gemacht, in denen Lernen für Roboter eine Bedeutung hat [Brooks, 1993]:

- Lernen von Parametern,
- Lernen von Weltwissen,
- Lernen von Verhalten und
- Lernen von Kooperation.

RoboCup bietet sich für alle diese Bereiche als Testplattform an. Das Lernen bestimmter *Parameter* ist in allen RoboCup-Ligen mit realen Robotern sinnvoll und wird dort auch eingesetzt. *Weltwissen* zu akquirieren ist besonders in der Rescue-Liga notwendig, um dem Operator, zum Beispiel mit automatisch erstellten Karten, hilfreiche Informationen zu geben. Lernen von *Verhalten* und *Kooperation* ist eines der ältesten Forschungsthemen im RoboCup, vor allem in der Simulationsliga.

3.1 Lernalgorithmen

Lernverfahren beziehen sich meistens auf ganz bestimmte Anwendungs- und Problem-Gebiete. In diesem Abschnitt werden weit verbreitete Methoden vor-

gestellt, die in vielen Bereichen eingesetzt werden können und auf die in späteren Kapiteln bezug genommen wird.

3.1.1 Vorwärtsgerichtete Neuronale Netze

Neuronale Netze werden gerne verwendet, wenn es darum geht, etwas zu lernen oder Lernprozesse von biologischen Systemen zu verstehen. Vorwärtsgerichtete, mehrschichtige neuronale Netze sind im Prinzip Funktionskompositionen, die zu einem Eingabevektor eine bestimmten Ausgabevektor erzeugen. Dabei sind die Grundfunktionen, aus denen sich das Netz zusammenbaut möglichst einfach gehalten.

Ein neuronales Netz kann so entworfen werden, dass es eine bestimmte Abbildungsfunktion erfüllt. Populär sind die Netze aber dadurch, dass sie Funktionen an Hand von Beispielen lernen können. Zum Training eines neuronalen Netzes wird oft der Backpropagation-Algorithmus verwendet, ein Verfahren, welches durch Gradientenabstieg und Zurück-Propagierung in die vorderen Schichten den Ausgabefehler den Netzes minimiert.

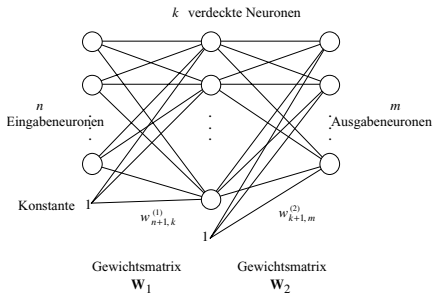

Abbildung 3.1: Ein dreischichtiges neuronales Netz. Die Neuronenschichten werden mit einer Konstanten erweitert, damit auch bei der Eingabe eines Nullvektors eine Ausgabe ungleich des Nullvektor erzeugt werden kann. Bearbeitetes Bild aus [Rojas, 1996]

In Abbildung 3.1 ist ein dreischichtiges neuronales Netz zu sehen. Die Eingabeneuronen geben nur die Eingabe weiter, sie berechnen nichts. Die Neuronen der verdeckten Schicht und der Ausgabeschicht bestehen aus einer gewichteten Summe der Ausgaben der vorhergehenden Schicht und einer Transferfunktion. Damit beim Backpropagation-Schritt der Gradient der Transferfunktion berechnet werden kann, muss diese differenzierbar sein. Gerne wird eine Sigmoid-Funktion der Form $s_c(x) = \frac{1}{1+e^{-cx}}$ eingesetzt, wobei c eine Konstante ist, die die Steigung am Übergang von negative auf positive Argumente bestimmt.

Vorwärtsberechnung

Zu einer Eingabe $\hat{v} = (v_1, v_2, \ldots, v_n, 1)$ wird die Gewichtematrix W_1 multipliziert, und auf Sigmoidfunktion komponentenweise angewendet. Wir erhalten die Aktivierung der verdeckten Schicht $h = s(\hat{v}W_1)$. Dieser Vektor wird wieder um die Konstante 1 erweitert und wir erhalten die Ausgabe o des Netzes durch die Multiplikation mit der Gewichtematrix W_2 und der komponentenweisen Anwendung der Transferfunktion, also $o = s_c(\hat{h}W_2)$ mit $\hat{h} = (h_1, h_2, \ldots, h_k, 1)$.

Rückpropagierung und Korrektur

Beim Training des Netzes wird der quadratische Fehler $E = e^T e$ mit $e = o - t$ der Ausgabe o und der gewünschten Ausgabe t berechnet. Anschließend werden die Korrekturen für die Gewichte so berechnet, dass die Verbindungen zu den Neuronen der vorhergehenden Schicht, die den Fehler verkleinern würden, verstärkt und die Verbindungen, deren Neuronen den Fehler maßgeblich vergrößern würden, geschwächt. Wenn $D_2 = \text{diag}(s_c'(\hat{h}W_2))$ die Diagonalmatrix ist, die in der Diagonalen die Ableitungen der Ausgabeneuronen enthält, dann ergibt sich der rückgerechnete Fehler zur Ausgabeschicht durch $\delta_2 = D_2 e$. Für den rückgerechneten Fehler zur verdeckten Schicht ergibt sich äquivalent $\delta_1 = D_1 W_2 \delta_2$ mit $D_1 = \text{diag}(s_c'(\hat{v}W_1))$. Die Gewichtematrizen ändern sich proportional zu den berechneten Fehlern, der Aktivität der (vorderen) Neuronen und einer Lernkonstanten γ durch $\Delta W_2^T = -\gamma \delta_2 \hat{h}$ und $\Delta W_1^T = -\gamma \delta_1 \hat{v}$.

Eine ausführliche Erläuterung des Backpropagation-Algorithmus ist in [Rojas, 1996] zu finden.

3.1.2 Verstärkungslernen

Verstärkungslernen (RL, Reinforcement Learning) ist eine sehr allgemeine Lernmethode. Die Umwelt und der lernende Agent besteht aus einer Menge von Zuständen, einer Menge von Aktionen, einer Bewertungsfunktion, einer Strategiefunktion und einer Belohnungsfunktion. Auf die einzelnen Funktionen wird im folgenden genauer eingegangen.

Ein Agent bekommt als Eingabe den aktuellen *Zustand der Welt* (Environment State) und wählt aus einer Menge von Aktionen eine Aktion (Action) aus, die meistens den Zustand der Welt für den nächsten — diskreten — Zeitschritt ändert. Es gibt nur endlich viele Zustände der Welt. Daraus folgt, dass bei vielen Anwendungen die reale Welt diskretisiert werden muss. Man bezeichnet diese Auswahl der Aktion als *Strategiefunktion* (Policy-Function). Diese Funktion ist nur vom aktuellen Zustand der Welt abhängig. Innere Zustände des Agenten, zum Beispiel bedingt durch die Speicherung alter Zustände, dürfen keine Rolle spielen. Dies ist ein entscheidendes Merkmal der Definition von RL. Dem Agenten wird außerdem mitgeteilt, ob die Welt jetzt besser oder schlechter geworden

ist (siehe Abbildung 3.2). Dies geschieht über eine *Belohnungsfunktion* (Reward-Function). Der Agent „merkt“ sich die Reaktion und kann auf diese Weise eine zeitlich längerfristige *Bewertung* (Value-Function, Action-Value-Function) der Aktion zum gegebenen Zustand vornehmen. Diese Bewertung wirkt sich jedoch nicht immer unmittelbar auf die Strategie aus, sondern wird meistens erst später in einem Aktualisierungsschritt berücksichtigt. Das Resultat einer Aktion zu einem gegebenen Zustand — der neue Zustand oder die Belohnung — muss nicht deterministisch sein. Fehlt ein explizites Weltmodell im Agenten, so spricht man von „model free reinforcement learning“ und der Agent muss erst mühsam ein implizites Modell lernen, um planen zu können. Die Art, wie der Agent lernt, ob er sich vergangene Bewertungen merkt, wie er die Aktionen berechnet usw. ist nicht festgelegt. Das wichtigste Kriterium zum Unterschied von RL zu anderen Lernmethoden ist dabei, dass dem Agenten das Ziel (goal) nicht direkt mitgeteilt wird, sondern nur indirekt über die Belohnung.

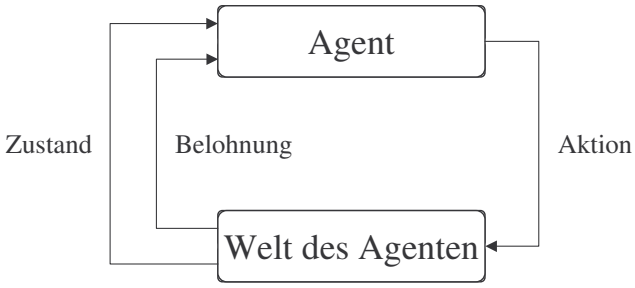

Abbildung 3.2: Kommunikation eines Agenten mit seiner Umgebung. Die Bewertungs- und Strategiefunktion ist im Agenten versteckt. Der Agent wählt eine Aktion, welche den Zusand der Umwelt ändert. Darauf hin wird der neuen Zustand bewertet.

Die bekannten Verfahren kann man grob in die Gruppen Dynamische Programmierung, Monte Carlo Methoden, Zeitreihenlernen[1] (TD-Learning, Temporal Difference Learning) sowie Generalisierung und Funktionsapproximation[2] (Generalization and Function Approximation) einteilen.

Das sehr ausführliche Standardwerk zu RL ist [Sutton, 1998]. Eine kompakte und verständliche Einführung Liefern [Kaelbling, 1996] und [Harmon, 1996].

Beispiel Backgammon

Ein praxisbezogenes Beispiel, in dem RL eingesetzt wurde, ist TD-Gammon [Tesauro, 1995]. Backgammon zu spielen, ist eine sehr komplexe Auf-

[1] Hierzu gehören Q-Learning und R-Learning.
[2] Hierzu gehören klassische neuronale Netze.

gabe. Das Spielfeld (siehe Abbildung 3.3) besteht aus 24 Feldern, 30 Steinen sowie zwei Auszonen. Um das Spiel spannender zu gestalten gibt es auch zwei Würfeln. Es ist also unmöglich alle Spielsituationen zu speichern, zu bewerten und vor allem eine Bewertung zu lernen. In jeder Spielsituation gibt es für einen Wurf etwa 20 Möglichkeiten seine Steine zu setzen. Eine heuristische Tiefensuche ist bei zusätzlich 21 möglichen Augenpaaren, bedingt durch die hohe Verzweigungszahl, schnell uneffektiv. Klassische Backgammonprogramme wurden wie Schach programmiert. Es wurden von professionellen Backgammonspielern Stellungen bewertet und Heuristiken gebildet. Im Gegensatz dazu hat TD-Gammon gegen sich selbst gespielt und dabei gelernt, gegen alle vorherigen Programme zu gewinnen und auf dem Niveau eines Weltklassespielers zu spielen. In der Version 3 ist das Programm nahezu unschlagbar und wird sogar für Spielanalysen benutzt, ist also Lehrer für Menschen.

Ein bis dahin sehr starkes Programm war Neurogammon. Es wurde auch von Tesauro entwickelt und bestand, wie der Name schon andeutet, auch aus einem neuronalen Netz, das ähnlich aufgebaut war, wie das NN aus TD-Gammon. Das Programm enthielt jedoch programmiertes Vorwissen und erhielt als Eingabe Expertenspiele, lernte also nicht selbst. Es wurde von TD-Gammon deklassiert.

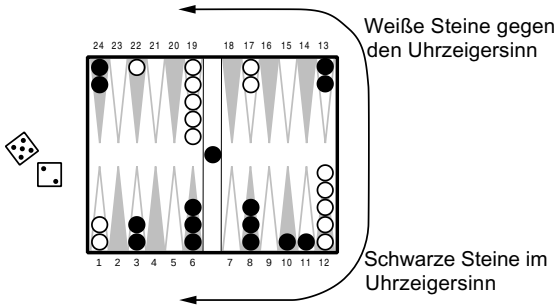

Abbildung 3.3: Eine Typische Spielsituation im Backgammon. Weiß spielt von unten nach oben, Schwarz umgekehrt. Der hier auch nicht gezeigte Verdoppelungswürfel wurde bei TD-Gammon nicht berücksichtigt. Abbildung aus [Sutton, 1998].

Der Kern von TD-Gammon besteht aus einem dreischichtigen neuronalen Netz als Bewertungsfunktion. Das Netz besteht aus 198 Eingabeneuronen für die kodierte Spielsituation, 40 – 80 verdeckten Neuronen und vier Ausgabeneuronen für die vier möglichen Endsituationen Gewinn oder Gammon für Weiß oder Schwarz. Ein Backgammon wurde nicht berücksichtigt. Die Eingabe setzt sich folgendermaßen zusammen: Für jedes Feld gibt es für jede Farbe vier Neuronen, die anzeigen, ob 1 bis 4 Steine auf der Position von der entsprechenden Farbe liegen. Mehr als vier Steine werden durch unterschiedliche Aktivierung des Neurons für vier Steine kodiert. Die restlichen Neuronen kodieren die Anzahl der Steine auf der Bar und die Farbe, die gerade am Zug ist. Es gibt unterschied-

liche Versionen von TD-Gammon, daher variiert die Anzahl der Neuronen. In der Version 3 gibt es nur noch 160 Eingabeneuronen, wobei auch Vorwissen verwendet wird.

Das Netz wurde zufällig initialisiert und mit einem modifizierten Backpropagation-Algorithmus — TD(λ) genannt — trainiert. Die Variable λ gibt dabei an, wie groß der Einfluss zukünftiger Stellungen einer Episode auf die aktuelle Stellung ist. Zum Training wurden 1,5 Mio. Spiele durchgeführt.[3] Jeder Zug in diesem Spiel ist eine Eingabe für das Netz. Die gewünschte Ausgabe ist das Ergebnis des ganzen Spiels (der Episode).

3.1.3 Genetische Algorithmen

Der Begriff genetischer Algorithmus ist der Evolutions-Biologie entlehnt [Rechenberg, 1994, Goldberg, 1989]. Die Gene sind dabei Parameter eines Systems, das optimiert werden soll. Für die Verbesserung des Systems werden die evolutionäre Prozesse *Vererbung*, *Rekombination*, *Mutation* und *Selektion* simuliert.

Bei einem genetischen Algorithmus wird mit einer Population von Parametersätzen gestartet. Diese Sätze werden durch eine Fitnessfunktion evaluiert. Für eine neue Population werden einige Datensätze zufällig, aber abhängig von der Bewertung, ausgewählt. Die Parameter dieser Sätze werden zufällig variiert, kopiert und mit anderen Sätzen der Population gemischt. Der Prozess beginnt von vorne.

Beispiel acht Damen

Ein Beispiel soll den genetischen Algorithmus veranschaulichen (entnommen aus [Russell, 2004]). Es sollen acht Damen so auf einem 8×8-Spielfeld verteilt werden, ohne sich gegenseitig zu bedrohen. Damen bedrohen sich gegenseitig, wenn sie in der gleichen Zeile, Spalte oder Diagonalen stehen. In jeder Spalte steht eine Dame. Die Zeile darf frei gewählt werden. Eine Brettstellung wird als eine Zeichenkette von acht Zahlen kodiert, welche die Zeile angibt, in der die Dame der zugehörigen Spalte steht. Die Fitnessfunktion gibt die Anzahl sich nicht angreifender Damenpaare an (siehe Abbildung 3.4). Nach der Bewertung der Stellungen werden jeweils zwei weiterverwendete Zeichenketten an einer Stelle aufgetrennt und die Enden vertauscht. Aus den Zeichenketten „67151582“ und „83472635“ wird dann zum Beispiel „671 72635“ und „834 51582“. Anschließend werden einige Ziffern mit einer gewissen Wahrscheinlichkeit geändert.

[3] Vernünftige Spiele bestehen aus 50 bis 60 Zügen, bei zufälligen Zügen dauert ein Spiel wesentlich länger.

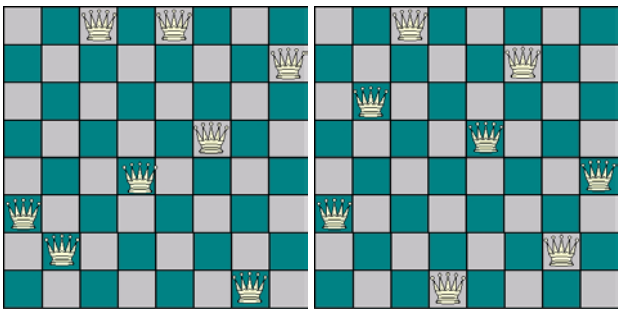

Abbildung 3.4: Die Stellung links ist als „67151582“ kodiert und hat eine Fitness von 23. In der Stellung mit der Kodierung „63184275“ bedrohen sich keine Damen gegenseitig (Fitnesswert 28). Die Lösung wurde mit Hilfe eines genetischen Algorithmus gefunden.

3.2 Lernmethoden und Anwendungen

Die im vorherigen Abschnitt vorgestellten einfachen Beispiele der Algorithmen bezogen sich auf eine Welt im Computer. Erst die Anwendung auf ein physikalisches Problem zeigt die Praktikabilität des Verfahrens, speziell den Lernerfolg für die Robotik. Es werden nun einige Modelle und Anwendungsbeispiele der Robotik vorgestellt, in denen erfolgreich Lernalgorithmen eingesetzt wurden.

3.2.1 Simulation

Das offensichtlichste und auch klassische Verfahren um die Lehrzeit realer Systeme zu minimieren ist die Simulation. Je genauer die Simulation ist, desto kürzer ist auch die anschließende Feinabstimmung des physikalischen Agenten. So wurde zum Beispiel ein Khepera Roboter der Firma K-Team[4] (siehe Abbildung 3.5) trainiert, in einem Labyrinth umherzufahren, ohne an Wänden anzustoßen [Janusz, 1995]. Die Welt des Agenten bestand aus den Daten von acht Abstandssensoren. Der Agent konnte aus 5 Aktionen (langsam links, schnell links, vorwärts, langsam rechts und schnell rechts) wählen. Eine Episode war immer 48 Zeitschritte lang. Wenn der Roboter in dieser Zeit kein Hindernis berührte, gab es eine Belohnung. Nach 50000 Trainingsdurchläufen wurde das Gelernte auf den realen Roboter übertragen. Die Erfolgsquote sank von 95% bei der Simulation auf 90% bei dem realen Roboter.

[4] www.k-team.com

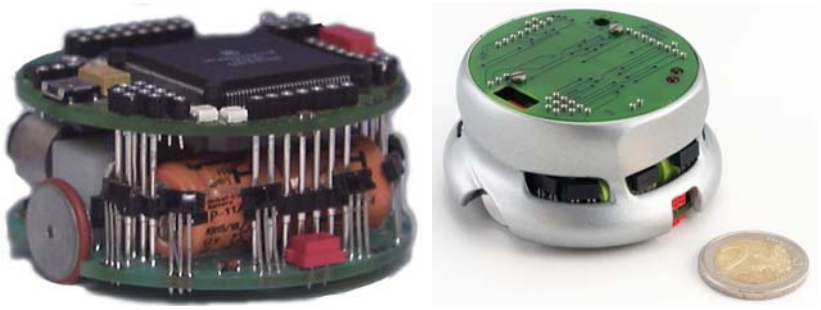

Abbildung 3.5: Zwei Generationen des Khepera Roboters. Der Khepera Roboter ist einer der beliebtesten Roboter für Lernaufgaben. Links die erste und rechts die zweite Version. Die Roboter können durch verschiedene Module erweitert werden, die auf die Grundplattform aufgesteckt werden. So gibt es Kameras, Infrarotsensoren, Greifarme usw. Die Roboter haben einen Durchmesser von 7cm, wiegen 80g und können eine maximale Geschwindigkeit von 1m/s erreichen. Die zweite Version hat einen Motorola 68331 Prozessor mit 25MHz, 512kByte RAM und 512 kByte Flash-Speicher. Programmiert wird der Roboter über eine serielle Schnittstelle. Abbildungen von der K-Team Webseite.

3.2.2 Modularisierung

Ein Beispiel wurde von Kalmár u. a. vorgestellt [Kalmár, 1998]. Die Aufgabe für einen Khepera Roboter bestand darin, einen Ball zu suchen, diesen mit einem Greifarm aufzunehmen, zu einem Pfahl zu fahren und den Pfahl mit dem Ball zu berühren. Dem Roboter wurde die Aufgabe jedoch so gestellt, dass er aus einer Menge von Verhalten (die der oberen Vorgehensweise entsprechen) ein Verhalten auswählen kann und nur keine negative Belohnung bekam wenn er mit dem Ball den Pfahl berührte.

Für die Aufgabenstellung war die Aufteilung in die Module offensichtlich. Außerdem sind die Teilaufgaben nicht abhängig voneinander.

Uchibe u. a. haben ein anderes Problem vorgestellt, in dem die Teilaufgaben stark interferieren [Uchibe, 1996]: Die Aufgabe bestand aus den zwei Teilaufgaben „schieße ein Tor“ und „weiche dem Torwart aus“. Die Teilaufgaben wurden getrennt voneinander trainiert. Das Problem besteht jedoch darin, dass beim Zusammenfügen die beiden Verhalten teilweise gegensätzliche Ziele haben, zum Beispiel, wenn der Ball nahe am Torwart liegt. Es gibt unterschiedliche Ansätze, um dieses Dilemma zu lösen, auf die hier nicht genauer eingegangen wird [Asada, 1994, Arseneau, 2000].

Die Modularisierung hat aus der Sicht der Biologie den Nachteil, dass die Verhalten, die ausgewählt werden können, vom Programmierer vorgegeben werden. Deshalb wird immer wieder der Versuch unternommen, mit einfachen Modellen

— neuronale Netze, evolutionäre Algorithmen usw. — Roboter einfache Aufgaben lernen zu lassen [Nolfi, 1994, Walker, 2003]. Wirklich komplexe Verhaltensweisen konnten aber bisher mit diesen Ansätzen noch nicht gelernt werden. Meistens handelt es sich um motorische Aufgaben wie Laufen oder einfache Labyrinthaufgaben.

3.2.3 Lernen aus Übungsaufgaben

Die Formulierung „learning from easy missions“ wurde von [Asada, 1996] eingeführt. Die Idee ist es, den Agenten zu Beginn des Lernverfahrens nur mit einfachen Situationen nahe am Ziel zu konfrontieren und dem Agenten schrittweise schwierigere Aufgaben zuzuweisen. Man nutzt das Vorwissen also nicht, um das Problem direkt zu vereinfachen, sondern indirekt über eine Variation der Aufgabenstellung. Das Verfahren ist vergleichbar mit dem Schritt von Q-Learning — bei dem nur die Bewertung des nächsten Schritts berücksichtigt wird — zu TD(λ)-Learning, bei dem viele Schritte in die Zukunft geschaut wird.

3.2.4 Lernen durch Emotionen

Ein aus der Kognitionswissenschaft stammender Ansatz lernender Roboter ist die Motivation durch emotionale Zustände. Die Verbindung von Verstärkungslernmethoden zur Biologie ist die Belohnungsfunktion, die immer für eine spezielle Aufgabe definiert wird. Biologisch passender ist es, die Belohnung über emotionale Zustände, wie Hunger, Schmerz, Unruhe, usw. zu definieren [Gadanho, 2002]. Der Lernerfolg gegenüber direkteren Belohnungsfunktionen ist dabei genauso gut. Der Vorteil liegt in der natürlichen Definition der Belohnung.

3.3 Lernen beim RoboCup

Erste Ansätze zum Einsatz von Lernmethoden kamen innerhalb RoboCups aus der Simulationsliga. Peter Stone hatte als Mitglied des CMUnites Teams erstmals Arbeiten über Lernmethoden in der Simulationsliga verfasst [Stone, 1998]. Für reale Roboter wurden innerhalb RoboCups bisher wenig Experimente durchgeführt und bei Wettbewerben eingesetzt. Dies war auch einer der Gründe, sich innerhalb dieser Promotion mit diesem Thema zu beschäftigen.

Für reale Roboter sind Optimierungen der Lernmethoden, bzw. der Definition der Problemstellung zwingend notwendig, um die Grenzen bisheriger Verfahren zu überwinden. Es werden im Folgenden drei Ansätze vorgestellt, die benutzt wurden, um reale Roboter lernen zu lassen. Bei der Simulation geht es in erster Linie um die Beschleunigung der simulierten Zeit, bei den anderen Verfahren um Problemvereinfachung durch Vorwissen.

3.3.1 Simulationsliga

Das bekannteste Beispiel aus der Simulationsliga ist das 3 vs. 2 Keepaway [Stone, 2001], für das mittlerweile ein Framework existiert, damit unterschiedliche Lernverfahren mit dem gleichen Problem verglichen werden können.[5] Dabei spielen drei Angreifer gegen zwei Verteidiger auf einem quadratischen Spielfeld (siehe Abbildung 3.6). Die Angreifer sind im Ballbesitz und deren Aufgabe ist es, den Ball möglichst lange zu behalten. Wenn sie den Ball an die Verteidiger verlieren oder der Ball außerhalb des Spielfelds ist, endet das Spiel. Die einzige Strategie der Verteidiger besteht darin, zum Ball zu laufen. Sie werden nicht trainiert. Die Angreifer können aus den Aktionen „Ball halten“, „Ball passen“, „gehe zum Ball“ und „freistellen“ auswählen. Es ist also eine Modularisierung vorhanden. Die Agenten lernten mit Q-Learning nach 20000 Episoden, den Ball durchschnittlich 14 Sekunden zu behalten.

Abbildung 3.6: Abbildung aus [Stone, 2001].

In der Simulationsliga wird auch versucht, das Verhalten des Gegners zu analysieren und dementsprechend das eigene Verhalten anzupassen. So wird von Visser und Weiland ein Verfahren beschrieben, welches das Verhalten des gegnerischen Torwarts und das Passverhalten des gegnerischen Teams online zu lernen [Visser, 2003]. Dabei werden bestimmte Attribute zur Analyse ausgewählt. Mit diesen Datensätzen werden dann Entscheidungsbäume nach dem ID3- oder C4.5-Verfahren[6] aufgebaut. Zum Beispiel soll gelernt werden, wann der Torwart das Tor verlässt. Durch das Wissen über den gegnerischen Torwart kann das eigene Verhalten dann parametrisiert werden. Einen ähnlichen Ansatz hat ein spanisches Team benutzt [Ledezma, 2004]. Sie haben jedoch zusätzlich zu dem mit C4.5 aufgebauten Entscheidungsbaum auch noch eine Vorhersage mit dem selbstgewählten Namen M5 für kontinuierliche Ausgaben benutzt. So kann beispielsweise nicht nur vorhergesagt werden ob der Gegner schießen wird, sondern auch wie stark dieser Schuss voraussichtlich sein wird.

Einen impliziten Ansatz zur Berücksichtigung des Gegners benutzten Buck und

[5] www.cs.utexas.edu/~AustinVilla/sim/keepaway
[6] Ein Überblick über Entscheidungsbäumen gibt [Duda, 2001].

Riedmiller durch neuronale Netze, welche die Erfolgsaussichten von Aktionen bewerten [Buck, 2000]. Dabei haben die Roboter eine beschränkte Auswahl an Aktionen zur Verfügung. Zu jeder Aktion gehört ein neuronales Netz, welches den Erfolg der Aktion bewertet (folgt ein Tor oder nicht). Die Eingaben des neuronalen Netzes sind Beobachtungen der Spielfeldsituation: Abstand zum Gegner, Winkel zum Tor, usw. Die Aktion mit der höchsten Erfolgsaussicht wird gewählt.

3.3.2 Middle-Size-Liga

An der Universität Freiburg wurde ein Experiment in die Mid-Size-Liga umgesetzt [Kleiner, 2003]. Die ersten Versuche wurden auch hier mit einem Simulator durchgeführt und später mit einem realen Roboter validiert. Das Ziel war es, unterschiedliche Eigenschaften der Roboter zu kompensieren. So waren die Roboter zwar alle gleich aufgebaut, aber die Schussstärke war etwas unterschiedlich. Es sollten aber nicht die Verhalten der Roboter von Hand optimiert werden, sondern die Anpassung der Verhalten sollte gelernt werden. Die Grundverhalten waren bei allen Robotern gleich, aber die Aktivierung der Verhalten wurde mit Hilfe des Q-Learning Algorithmus gelernt. Dadurch haben Roboter mit einem starken Schuss früher geschossen und Roboter mit einem schwachen Schuss etwas später. Die Anzahl der erfolgreichen Torschüsse stieg stärker an, als durch handoptimierte Parameter.

3.3.3 4-Legged-Liga

In der AIBO-Liga liegt der Fokus von Lernmethoden bei der Optimierung der Laufgeschwindigkeit. Es gibt Untersuchungen zur Parameteranpassung von Gelenk-Trajektorien mit einer Verstärkungslernmethode [Kohl, 2003] oder mit einem genetischen Algorithmus [Röfer, 2005]. Die Optimierungen erhöhen die ursprüngliche Geschwindigkeit des AIBOs von 100mm/s auf rund 300mm/s. Eine Methode, die das Laufen des Roboters direkt lernt, wurde im RoboCup bisher nicht eingesetzt. Auf die Verstärkungslernmethode von Kohl und Stone kommen wir in Kapitel 9 zurück.

Zusammenfassend kann gesagt werden, dass Lernen schon in vielen Bereichen der Robotik und insbesondere innerhalb von RoboCup erfolgreich angewendet wird. Nur in der Small-Size-Liga, in der sich die Roboter und der Ball relativ zur Spielfeldgröße am Schnellsten bewegen, gibt es nicht viele Vorarbeiten. Dies kann daran liegen, dass in dieser Liga mehr Wert auf Schnelligkeit und ein extrem reaktives Verhalten gelegt wird. Die vorliegende Arbeit zeigt jedoch, dass gerade diese Optimierung sehr gut mit Lernverfahren durchgeführt werden kann. Im nächsten Kapitel wird erst das Small-Size-System der FU-Fighters vorgestellt, das als Grundlage für die ab Kapitel 5 folgenden Lernmethoden dient.