Universität Koblenz-Landau
Fachbereich Informatik
Arbeitsgruppe Künstliche Intelligenz

STUDIENARBEIT

Reinforcement Learning als Methode zur Entscheidungsfindung beim simulierten Roboterfußball

Irene Markelić
<markelic@uni-koblenz.de>

Betreuer:
Dipl. Inform. Oliver Obst

Koblenz, 01.01.04

Inhaltsverzeichnis

1 Vorwort

Anfang des Sommersemesters 2003 begann ich, bei der Arbeitsgruppe Künstliche Intelligenz als Wissenschaftliche Hilfsmitarbeiterin zu arbeiten. Genauer gesagt bei der Gruppe „Robolog“. Dies ist der Name der Fussballmannschaft der Simulationsliga, welche an der Universität Koblenz entwickelt wurde. In diesem Bereich wollte ich auch meine Studienarbeit anfertigen. Zu Beginn hatte ich allerdings noch keine Idee, worin meine Aufgabe bestehen könnte. Klar war, dass sie der Verbesserung des Spielverhaltens der Mannschaft dienen sollte. Also begann ich damit, mir Spiele unseres Teams während der kurz zuvor stattgefundenen Weltmeisterschaft in Padua anzuschauen, bei der wir schon in der Vorrunde ausgeschieden waren. Woran lag das? Schnell bemerkte ich, dass wir bei grundlegenden Techniken nicht wesentlich schlechter waren als unsere Gegner. Wir dribbelten und passten ganz gut. Allerdings verlor unser Team häufig den Ball. Es erfolgten Pässe, die mit der Übernahme des Balls durch den Gegner endeten, und mindestens genauso oft Dribbelaktionen, die ebenfalls den Ballverlust zur Folge hatten. Wahrscheinlich hätte man durch geschicktere Wahl der Aktionen Dribbeln und Passen den Ball innerhalb der Mannschaft halten können. Meine Schlussfolgerung war folgende: Wenn der Spieler bessere Entscheidungen träfe, bezüglich seiner Spielaktionen, würde das Spielverhalten insgesamt besser. Damit war die Aufgabenstellung geboren. Ergänze einen Spieler um die Fähigkeit, selbstständig zwischen mehreren möglichen Aktionen die beste auszuwählen. Das erklärt den ersten Teil des Titels dieser Arbeit, „Entscheidungsfindung bei der Auswahl von Spielaktionen“. Doch wie könnte ich das umsetzen? Ich könnte unmöglich jede Situation bedenken und dem Agenten sagen, was er wann zu tun hat, dafür gab es einfach zu viele Situationen. Eine andere Möglichkeit bestand darin, dass der Agent sich selbst dieses Wissen aneignete. Er musste irgendwie erlernen, wie er handeln sollte. Bei meiner Literaturrecherche stiess ich auf einen interessanten Lösungsansatz. Nämlich das sogenannte Reinforcement Learning. Was das ist, wie es funktioniert, wie ich es für mein Problem nutze, und welche Resultate es letztendlich liefert, ist Thema dieser Arbeit.

Kapitel 1 Gibt einen Überblick über Roboterfußball allgemein und führt den Leser in die dabei auftretenden Problemstellungen ein.

Kapitel 2 Beschreibt alle theoretischen Grundlagen für die praktische Lösung des Problems. Erklärt was Reinforcement Learning ist, indem zum einen die zugrunde liegende Idee dieses Lernverfahrens erklärt wird, und zum anderen die dazu entwickelten theoretischen Konzepte vorstellt. Darauf aufbauend wird schließlich

der später von mir benutzte Algorithmus (SARSA(λ)) evaluiert.

Kapitel 3 Zu Beginn des Kapitels erkläre ich meinen Entwurf für die theoretische Lösung des Problems. Ziel ist es, einen Spieler zu entwickeln, dessen Entscheidungsfindung auf Reinforcement-Learning beruht. Im weiteren Teil setze ich diesen dann in die Tat um und stelle die praktische Durchführung vor.

Kapitel 4 Um festzustellen von welcher Qualität das Spielverhalten des Reinforcement-Learning-Spielers im Vergleich zu einem herkömmlichen Robolog-Spieler ist, werden jeweils Spielsequenzen generiert. Ihre Ausgänge werden in diesem Kapitel aufgelistet und ausgewertet.

Kapitel 5
Enthält ein Resumee über Erfolg oder Mißerfolg des Reinforcement-Spielers. Desweiteren stelle ich Ideen vor, die den Spieler meiner Meinung nach noch verbessern könnten, und konstatiere meine Meinung zur Eignung des RL für das Roboterfußballproblem.

2 Roboterfussball

Wie der Name erahnen lässt, kämpfen beim Roboterfußball Maschinen statt Menschen auf einem Spielfeld darum, den Ball möglichst oft ins gegnerische Tor zu befördern und so zu gewinnen. Allerdings ist dies nicht der einzige Unterschied zum echten Fußball. Je nach Disziplin ist die Anzahl der Spieler und die Größe des Feldes sehr unterschiedlich, genauso wie die an die Roboter gestellten Anforderungen. Momentan gibt es ca. sechs verschiedene Arten von Roboterfußball, u.a. die Simulation League, für die es seit diesem Jahr Wettkämpfe in 2-D und in 3-D gibt, die Small-Sized League (s. Bild 1), Middle-Sized League (s. Bild 2), und die 4-legged League (s. Bild 3). Bei

Abbildung 1: Small-Size league, RoboCup in Fukuoka 2002

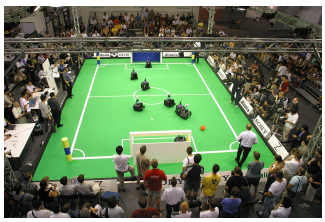

Abbildung 2: Middle-Size league, RoboCup in Padua 2003

Abbildung 3: 4-legged league, RoboCup in Fukuoka 2002

den drei letzten handelt es sich bei den Spielern tatsächlich um reale Roboter, während die der Simulation-League Computerprogramme sind, welche ihre Derbys auf einem ebenfalls am Rechner simulierten Spielfeld, dem Soccerserver, austragen. Dabei spielen elf individuelle Programme pro Team gegeneinander. Die Visualisierung erfolgt über den Soccer-Monitor und ist in Bild 4 zu sehen. Jedes Jahr gibt es Wett-

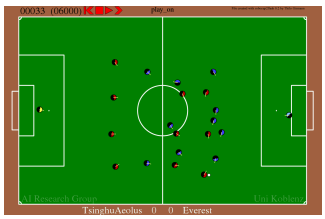

Abbildung 4: Simulation league, RoboCup in Fukuoka 2002

kämpfe und sogar eine Weltmeisterschaft, den sogenannte RoboCup (ursprünglich: Robot World Cup Initiative), um die beste Mannschaft der jeweiligen Liga ausfindig zu machen. In 2002 in Fukuoka-Busan in Japan traten einhundertachtundachtzig Mannschaften aus neunundzwanzig verschiedenen Ländern gegeneinander an, was zeigt das international reges Interesse am Roboterfußball vorhanden ist. Das langfristige Ziel ist es bis, zum Jahr 2050 eine Mannschaft entwickelt zu haben, die in der Lage ist, den dann echten amtierenden Fußballweltmeister zu schlagen. Natürlich ist Roboterfußball sehr medienwirksam, aber es handelt sich dabei keineswegs nur

um Spielerei, sondern um eine internationale Forschungs- und Lehrinitiative, mit der Absicht den Fortschritt im Maschinen Lernen voranzutreiben. Echte Wissenschaft also, was die zu bewältigenden Aufgaben widerspiegeln. Allerdings sind diese - wie zuvor erwähnt - je nach Liga unterschiedlich. So unterscheiden sich die echten Fußballroboter von den simulierten nicht nur durch die aufwendige Hardwarebetreuung, die sie benötigen. Sie nehmen ihre Umwelt über bestimmte Sensoren, meist Kameras war. Um ihre eigene Position, die der Gegner und des Balles, anhand des Bildmaterials herauszufinden, muß viel Bildverarbeitung betrieben werden. Probleme von denen die Programmierer der Simulations-Spieler verschont bleiben. Diese können sich ganz auf die Interaktionsmöglichkeiten der Spieler untereinander konzentrieren. Trotzdem haben sie alle eine Gemeinsamkeit. Jeder einzelne Spieler, egal welcher Liga, ist autonom. Jedes Team stellt somit ein Multiagenten-System dar, in dem versucht wird gemeinsam eine Aufgabe in einem dynamischen Umfeld zu erledigen, noch dazu in Konkurrenz mit einem Gegner. Damit sich die Agenten unter den eben angesprochenen Gesichtspunkten möglichst gut verhalten, werden viele Methoden der Künstlichen Intelligenz , wie z.B. Planen, Lernverfahren (überwachte und unüberwachte) u.v.m. eingesetzt.

3 Reinforcement Learning

Es ist eine bekannte Tatsache, dass Lebewesen durch Interaktion mit ihrer Umgebung lernen. Ein hungriges Baby z.B. schreit, woraufhin es von der Mutter gefüttert wird. Ist es das nächste Mal hungrig, wird es wahrscheinlich wieder schreien, da es gelernt hat, so sein Ziel - gesättigt sein - erreichen zu können. Das Ganze passiert ohne eine Form von Lehrer, nur aus der Motivation heraus, etwas erreichen zu wollen und der Beobachtung der Reaktion der Umwelt auf eine Aktion. Wir Menschen tun dies ständig, bewusst wie unbewusst, immer mit dem Ziel so zu handeln, dass das Resultat möglichst vorteilhaft für uns ist. Frei nach dem Motto Versuch macht klug, probieren wir einige Handlungen aus und beobachten wie unsere Umwelt darauf reagiert. Später können wir dieses Wissen nutzen, um bestimmte Ziele zu erreichen. Beim Autofahrenlernen etwa, gibt es diesen brenzligen Augenblick beim Anfahren, bei dem man von der Kupplung gehen und gleichzeitig Gas geben muß[1]. Wie lernen wir, diese zwei Aktionen so zu kombinieren, damit die Anfahrt reibungslos von statten geht? Ein Fahrlehrer kann uns zwar unterstützen, indem er uns Anweisungen gibt, aber ausreichend ist das nicht, sonst könnte sich jeder Novize in ein Auto setzen und losfahren. In der Regel versuchen wir es ein paar Mal und bekommen so ein „Gefühl“ für das „richtige“ Verhalten. Untersuchen wir, wie es dazu kommt: Angenommen wir befinden uns in der eben geschilderten Anfahrtssituation: der Motor ist an, wir gehen von der Kupplung, geben etwas

[1] Vorausgesetzt es handelt sich um ein Auto mit herkömmlicher Schaltung ohne Automatik.

Gas und plötzlich geht der Motor aus. Was haben wir falsch gemacht? Diese Frage kann man auch anders stellen: Welche Aktion haben wir ausgeführt, die zu dieser unerwünschten Situation führte? Angenommen unsere Handlung war „schnell von Kupplung gehen“. Wir knüpfen einen Zusammanhang mit dieser Handlung und der damit erreichten Situation. Also wenn wir beim Anfahren in der Situation „Motor ist an“ sind und „schnell von Kupplung gehen“ ausführen, führt dies zu „Motor geht aus“. Da das nicht unser gewünschtes Ziel war, war dies wahrscheinlich die falsche Handlung. Beim nächsten Mal werden wir eine andere Aktion ausführen, in der Hoffnung, mit dieser erfolgreicher zu sein. Angenommen die neue Aktion ist „langsam von Kupplung gehen“ und die Folgesituation ist „Motor geht nicht aus“, was natürlich gewünscht war, so haben wir gelernt, dass es in der Anfahrtssituation „Motor ist an“ besser ist, die Aktion „langsam von Kupplung gehen“ auszuführen, als „schnell von Kupplung gehen“. Um das zu lernen, haben wir uns schon zwei Mal in der Anfahrtssituation „Motor ist an“ befunden. Je öfter wir in dieser Situation waren und je mehr verschiedene Aktionen wir ausgeführt haben, desto mehr wissen wir bezüglich der Güte der möglichen Handlungen, die uns zur Verfügung stehen. In anderen Worten: Desto mehr Erfahrung haben wir gesammelt. Z.B. Wir haben folgendes Wissen gesammelt: In Anfahrtssituation „Motor ist an“ , führt
Handlung 1(= gehe schnell von Kupplung) zu einem schlechtem Ergebnis,
Handlung 2(= gehe langsam von Kupplung) zu gutem Ergebnis, und
Handlung 3(= gehe langsam von Kupplung und gib gleichzeitig Gas)
zum besten Ergebnis. Werden wir wieder mit der Situation konfrontiert, können wir anhand dieser Erfahrung die beste Aktion auswählen. Hier wäre das die Handlung 3. Anhand dieses Beispiels wird auch deutlich, dass wir natürlich nicht tatsächlich die beste Handlung ausführen, sondern lediglich die beste, die wir *kennen*. Womit wir zum ersten einer Reihe wichtiger Probleme kommen:

Problem Nummer 1: Wolllten wir sicher gehen, wirklich die beste Handlung zu kennen, so müßten wir alle möglichen Handlungen ausführen und ihren Wert für uns prüfen. - Ist das überhaupt möglich? Man muss irgendwie sicherstellen, dass alle Aktionen eine Chance haben, gewählt zu werden

Problem Nummer 2: Außerdem kann es vorkommen, dass wir unser Ziel erst am Ende einer sehr langen Handlungskette erreichen. Es ist dann nicht mehr so leicht zu sagen, welche Aktionen tatsächlich hilfreich waren, und welche eher kontraproduktiv. - Wie bewerten wir sie also?

Problem Nummer 3: Bei der Handhabung sehr langer Handlungsketten, ergibt sich eine weitere Schwierigkeit. Wie man aus den obigen Beispielen leicht erkennen kann, gewinnt der Lernende erst dann Information über die Güte seiner Handlungen, wenn er in die gewünschte Situation kommt, also der Grund für seine

Motivation befriedigt wird. Je länger die Handlungskette, desto länger dauert es natürlich bis er diese Informationen erhält, sprich, desto länger dauert es bis er etwas gelernt hat. Auch beim RL verlangsamt dies die Lernzeit.

Problem Nummer 4: Wenn wir schon ein paar Erfahrungen gesammelt haben und in einem bestimmten Bereich wissen, wie wir an unser Ziel gelangen können, so könnten wir zukünftig immer diesen Weg gehen, also von der Erfahrung profitieren. Allerdings lernen wir in diesem Fall nichts mehr hinzu. Es könnte immerhin sein, dass wir zwar einen Weg gefunden haben, dieser aber noch lange nicht der beste ist (vgl. Problem Nummer 1). Würden wir allerdings immer nur wahllos Dinge ausprobieren, so würden wir zwar viel lernen, allerdings wäre unser Verhalten, bezogen auf die Motivation unser Ziel zu erreichen, höchst ineffektiv. Dieser Sachverhalt ist auch bekannt als das „problem of exploration and exploitation“[2].

Diesen vier Schwierigkeiten muß stets besondere Beachtung geschenkt werden, da sie sich durch die gesamte RL-Thematik ziehen. Später werden wir sehen, welche Lösungsansätze es dafür gibt.
Im Grunde sammeln wir Informationen über die Güte von Aktionen, indem wir sie in bestimmten Situationen einfach ausprobieren. Im nachhinen können wir dann jeder Situation eine Aktion zuordnen, die sich als besonders vorteilhaft für den weiteren Verlauf der Problemlösung erwiesen hat. Das Reinforcement Learning, kurz RL, versucht, genau dieses Prinzip für Maschinen nutzbar zu machen.
Dazu wird ein Agent gebraucht, der in der Lage ist, seine Umwelt wahrzunehmen, und Aktionen ausführen kann, die diese verändern (s. Bild 5). Auch beim RL

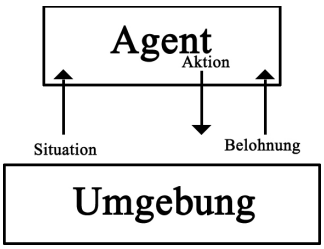

Abbildung 5: Agent, der mit seiner Umwelt interagieren kann.

spricht man von **Situationen** und **Aktionen**. Wie beim Autofahrbeispiel ist mit dem Wort Situation die Lage gemeint, die der Agent wahrnehmen, und in der er sich befinden kann. Aktion ist intuitiverweise eine der Handlungen, die er ausführen kann. Natürlich muss es ein **Ziel** geben, welches der Agent erreichen soll. Für

[2] exploration engl. = Erforschung , exploitation engl. = Ausbeutung/Ausnutzung

das Ziel bekommt er eine **Belohnung**, die als Motivation für alle Handlungen dient. Normalerweise ist die Belohnung ein numerischer Wert, den der Agent für das Erreichen einer bestimmten Situation erhält, z.B. für ein geschossenes Tor beim Roboterfussball könnte er +100 erhalten. Oder für jedes gegnerischen Tor -100. Die Belohnung ist das einzige Mittel für den Programmierer dem Agenten das Ziel zu kommunizieren. Deswegen muss man Ziel und Art der Belohnung sehr vorsichtig wählen, damit der Agent nicht plötzlich etwas unerwartetes lernt. Angenommen man möchte einem Agenten das Dame-Spielen beibringen, so sollte man eine Belohnung für das Gewinnen eines Spiel geben, und *nicht* für jede weitere Dame, die er erlangt. Natürlich sind viele Damen hilfreich, aber sie sind nur ein untergeordnetes Ziel auf dem Weg zum Erfolg. Das ist eine Eigenschaft von RL, die man immer bedenken sollte, und welche meiner Meinung nach auch eine große Stärke dieses Verfahrens ist. Es kümmert sich um das Lösen eines Problems im Ganzen, um die dafür notwendigen Teilziele braucht man sich nicht kümmern, sie werden automatisch mitgelöst.
Wie sieht es aber nun in der Praxis aus? Bisher haben wir nicht mehr als eine Menge von Situationen $S = \{s_0, s_1, s_2...\}$, eine Menge von Aktionen $A = \{a_0, a_1, a_2...\}$ sowie einen Zielzustand Z, für den es eine Belohnung (engl. reward) $r(s, a)$ gibt. Das Problem besteht darin, eine Handlungssequenz zu erlernen, um ein vorgegebenes Ziel möglichst gut zu erreichen. Dafür müssen wir den jeweiligen Situationen die Aktionen zuordnen, die am ehesten zum Ziel führen. Vgl. Autofahrbeispiel. Eine Zuordnung dieser Art, also eine Anleitung, die unserem Agenten in jeder Situation sagt, welche Aktion er ausführen muss, wird **Policy** genannt und mit π abgekürzt. Wenn wir eine Policy haben, die jeder Situation eine Aktion derartig zuordnet, dass der Agent damit sein Ziel nicht nur erreicht, sondern sogar bestmöglich erreicht, spricht man von einer **Optimalen Policy**, π^*. „Bestmöglich“ bedeutet in unserem Fall, so dass die Belohnung über einen Zeitraum maximal wird. Genau das ist die Lösung, die wir wollen: die optimale Policy[3]. Das Problem, sie zu erlernen wird auch als „Control-Problem“ bezeichnet (vgl. [5]). Die folgenden Abschnitte zeigen anhand eines Beispiels, wie man vorgehen kann, um π^* zu finden.

3.1 Value-Function

Die optimale Policy sagt dem Agenten in jeder Situation, welche Aktion besonders gut ist, sich besonders lohnt auszuführen. Eine gute Aktion ist natürlich eine, die dem Erreichen des Ziels dienlich ist, und eine schlechte, eine die davon wegführt. Dementsprechend gibt es auch gute und schlechte Zustände. Ein guter Zustand ist einer, von dem wir schnell in den Zielzustand kommen. Je schneller das geht, desto besser ist der Zustand. Wir brauchen etwas, um die Qualität eines Zustands ma-

[3]Genauer gesagt, eine davon, denn oftmals gibt es mehrere.

thematisch beschreiben zu können. Genau das macht die **Value-Function V**. Sie ordnet jeder Situation gemäß einer gegebenen Policy einen Wert zu, mit Hilfe dessen man die Güte der einzelnen Situationen vergleichen kann. $V^{\pi}(s_t)$ gibt die kumulative Belohnung an, die, von Zustand s zum Zeitpunkt t ausgehend, mit der angegebenen Policy π erreicht werden kann. Sie ist folgendermaßen definiert:

Definition 3.1 (V-Funktion) $V^{\pi}(s_t) \doteq r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ...$

Wobei γ ein sogenannter Discount-Factor ist, mit dem Zweck, Belohnungen, die weiter in der Zukunft erhalten werden, weniger Gewichtung beizumessen[4]. Die Zeit wird durch t repräsentiert, und r steht für die reward (engl. Belohnung), die der Agent beim Ausführen der jeweiligen Aktion erhält. Betrachten wir ein Beispiel. Wir befinden uns in der sogenannten Gridworld (engl. Gitterwelt), welche in Abbildung 6 zu sehen ist. Wie wir sehen, gibt es sechs verschiedene Zustände,

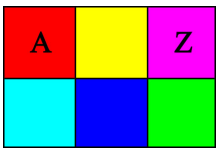

Abbildung 6: Gridworld mit Agent und Zielzustand.

symbolisiert durch sechs verschiedenfarbige Kästchen. Unser Agent kann sich der vier Aktionen „gehe nach rechts“, „gehe nach links“, „gehe nach oben“ und „gehe nach unten“ bedienen, um seinen Aufenthaltsort zu verändern. Sie werden durch Pfeile in die jeweilige Richtung symbolisiert. In jeder Situation steht ihm ein Teil dieser Aktionen zur Verfügung. Außerdem gibt es einen Zielzustand Z, für dessen Erreichen es eine Belohnung von +100 gibt. Es handelt sich um einen sogenannten absorbierenden Zustand oder Endzustand. Gelangt der Agent einmal in ihn hinein, kommt er nie wieder heraus, und der Durchgang ist beendet. Man spricht in diesem Fall auch von einer **Episode**. In vielen Fällen von RL wird über Episoden gelernt, was sich meist durch die Art des Problems ergibt. Will man z.B. ein Spiel erlernen, z.B. Dame oder Schach, so hat man einen genau definierten Endzustand. Entweder man gewinnt selbst, der Gegner, oder es gibt ein Unentschieden. Dann fängt man wieder von vorne an, startet ein neues Spiel - eine neue Episode. Im Gegensatz dazu gibt es **kontinuierliches Lernen**. Dies ist der Fall, wenn es keinen Endzustand gibt. Episodisches Lernen ist mathematisch einfacher. Da ich in dieser Arbeit auch ein episodisches Problem behandeln werde, gehe ich nicht näher auf kontinuierliche Probleme ein[5]. Wie in Abbildung 6 zu sehen,

[4] Man beachte, daß dieses eine Lösung für Problem Nummer 2 darstellt.
[5] Bei weiterem Interesse an kontinuierlichen Problemen s. [5] Kapitel 3.3.

befindet sich unser Agent A zu Beginn der Episode in Situation . Formal haben wir:

S = { , , , , , },

A = {→ , ↓ , ↑ , ← },

Sei π = { → , ↓ , → , ↑ , ↑ }

Sei außerdem r(↑) = 100 und r(→) = 100 .

Die Belohnung für alle anderen Situations-Aktions-Paare sei null.

Ausgehend vom Startzustand des Agenten ergibt sich gemäß der vorgegebenen Policy der in Bild 7 dargestellte Weg. Die Zahlen an den Pfeilen symbolisieren die Belohnungen, die der Agent beim jeweiligen Übergang erhält.

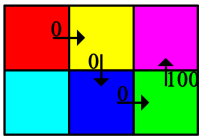

Abbildung 7: Weg des Agenten vom Start- zum Zielzustand, gemäß vorgegebener Policy.

Natürlich ist es besser für den Agenten, mit seiner Policy in zu sein, als in , da er dann viel näher an seinem Ziel ist. Angenommen γ sei 0.9, so kann man kann den V-Wert der besuchten Situationen leicht mit oben angegebener Formel 3.1berechnen:

V^{π}( ) = $0 + 0.9 * 0 + (0.9)^2 * 0 + (0.9)^3 * 100 = 73$

V^{π}( ) = $0 + (0.9) * 0 + (0.9)^2 * 100 = 81$

V^{π}( ) = $0 + (0.9) * 100 = 90$

V^{π}( ) = 100

vgl. Abbildung 8

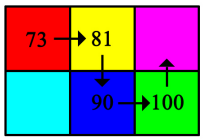

Abbildung 8: V^{π}

Eine Value-Function, die jeder Situation den Wert gemäß der Belohnung zuordnet, den man maximal ausgehend von dieser Situation erreichen kann, heißt **Optimale Value Function,** $V^*(s)$. Wenn man die optimale Value Function hat, ist es einfach, die optimale Policy zu finden. Denn man muss in jeder Situation nur die Aktion wählen, die zur Folgesituation mit dem größtem V-Wert führt.

Definition 3.2 (Optimale V-Funktion) $V^* = V^{\pi^*} \doteq argmax(\pi)V^{\pi}(s), (\forall s)$

Die optimale Value Function für Gridworld, und die davon abgeleitete optimale Policy ist in Bild 9 und 10 zu sehen: Die optimale Value Function zu finden ist nicht

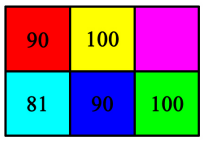

Abbildung 9: V^* für Gridworld

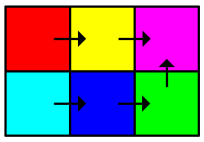

Abbildung 10: π^* für Gridworld

so leicht. Dafür müßte der Agent jeden Zustand als Startzustand gehabt, und in jedem Zustand jede Aktion ausgeführt haben. Denn nur dann hätte er das Wissen über die optimale Handlung für jeden Zustand (siehe Problem Nr. 1). Das ist für Probleme mit größerem Zustandsraum kaum praktikabel. Hinzu kommt außerdem, daß der Agent sich nicht immer sicher sein kann, durch die Ausführung einer bestimmten Aktion auch tatsächlich die gewünschte Situation erreichen zu können. In Gridworld ist dies zwar der Fall, so kommt man durch Ausführung der Handlung „gehe nach rechts“ in Situation ■ immer in Situation ■, -man spricht von einem deterministischen Problem - aber wie man sich leicht vorstellen kann, gibt es auch andere Fälle. So zum Beispiel, wenn die Aktion darin besteht, einen Würfel zu werfen. Die Zuordnung einer Folgesituation zu einem Situations-Aktions-Paar wird oft als **Delta-Funktion** $\delta(s,a)$ bezeichnet.

3.2 Q-Function

Anstatt die Situation zu bewerten, können wir natürlich auch den Aktionen in einer Situation Werte geben. Dies entspräche dem Autofahr-Beispiel zu Beginn des Kapitels. In der Situation „Motor ist an“ wurde die Aktion „schnell von Kupplung gehen“ als schlecht bewertet, „langsam von Kupplung gehen“ als gut, und die beste war „langsam von Kupplung gehen und Gas geben“. Eine mathematische Funktion, die uns hilft, so etwas formal auszudrücken, ist die **Q-Function**

Definition 3.3 (Q-Funktion) $Q(s,a) \doteq r(s,a) + \gamma V(\delta(s,a))$

Wobei $\delta(s,a)$ die Situation angibt, die eintritt, wenn in Situation s Aktion a ausgeführt wurde, und γ ist wieder der Discount-Factor. Der Q-Wert eines Situations-Aktions-Paares ergibt sich also aus der tatsächlichen Belohnung, die für das Ausführen der Aktion erhalten wird und dem V-Wert der darauf folgenden Situation. Angenommen wir möchten den Q-Wert des Paares (■, →) mit $\gamma = 0.9$ und
δ(■, →)=■ gemäß der Optimalen Policy berechnen, so erhalten wir:
Q(■ →) $= 0 + 0.9 * 100 = 90$
Je höher der Q-Wert eines Paares (s,a), desto besser ist es, in Situation s Aktion a auszuführen. In Bild 11 sehen wir alle Q-Werte für die Optimale Value-Funktion V^*.

3.3 GPI

Wenn man für jede Situation weiß, wie gut die zur Verfügung stehenden Aktionen sind, dann hat man die optimale Policy gefunden. Um Werte für *alle* Aktionen bestimmen zu können, brauchen wir eine Policy, die jeder Aktion eine Chance gibt, gewählt zu werden - eine stochastische. Bisher haben wir nur den Fall der deterministischen Policy betrachtet. Letztere ordnet jeder Situation fest eine Aktion zu.

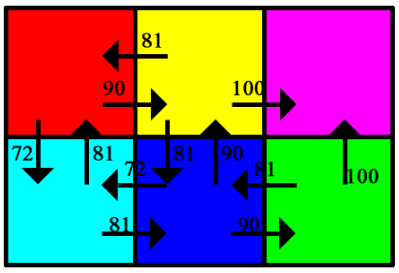

Abbildung 11: Q* für Gridworld

Episoden, die denselben Startzustand haben werden immer gleich aussehen. Deswegen kann man leicht mit der angegebenen Formel für die V-Funktion den Wert dieses Startzustandes (oder eines beliebig anderen) berechnen. Bei einer stochastischen Policy aber, werden zwei Episoden selten genau gleich ablaufen. Angenommen beide Episoden starten in Situation S_0, Episode eins könnte folgendes Resultat ergeben: S_0A_1, S_3A_3, Finaler Zustand. Episode zwei hingegen sähe vielleicht so aus: S_0A_1, S_3A_2, S_4A_1, Finaler Zustand.
Der V-Wert einer Situation soll darüber Aufschluß geben, wieviel Belohnung über einen Zeitraum ausgehend von diesem Zustand erreicht werden kann. Um diesen Wert für eine Situation gemäß einer stochastischen Policy ermitteln zu können, berechnet man den Durchschnitt der V-Werte aller Episoden. Für den Zustand S_0 würde man in Episode eins vielleicht den V-Wert 60 erhalten und für Episode zwei den V-Wert 40. Also bekäme S_0 den V-Wert 50. Selbstverständlich handelt es sich dabei nicht um den tatsächlichen V-Wert, sondern nur um eine Schätzung, dargestellt durch V'. Sei k die Anzahl der zur Verfügung stehenden Episoden, so gilt gemäß dem Gesetz der großen Zahlen[6]:
$k \to \infty \Rightarrow V' \to V$.
Wie schon erwähnt ist das Ziel beim RL, die optimale Policy zu finden. Eine Möglichkeit dieses zu tun, bietet folgender Ansatz: Man beginnt mit irgendeiner beliebigen Policy und berechnet Q^π, wie im Beispiel oben. Anhand dieser Werte kann man eine

[6] „Das Gesetz der großen Zahlen besagt, dass sich die relative Häufigkeit eines Zufallsergebnisses immer weiter an die theoretische Wahrscheinlichkeit für dieses Ergebnis annähert, je häufiger das Zufallsexperiment durchgeführt wird."[1]

neue Policy bestimmen, die in jeder Situation jeweils die besten Aktionen wählt. Für diese neue sogenannte Greedy-Policy[7] π' gilt, sie ist besser oder mindestens genausogut wie die alte Policy π[8]. Um π' bestimmen zu können, reicht oft eine recht grobe Schätzung von Q' aus, wie man in Bild 12 erkennen kann[9]. In der lin-

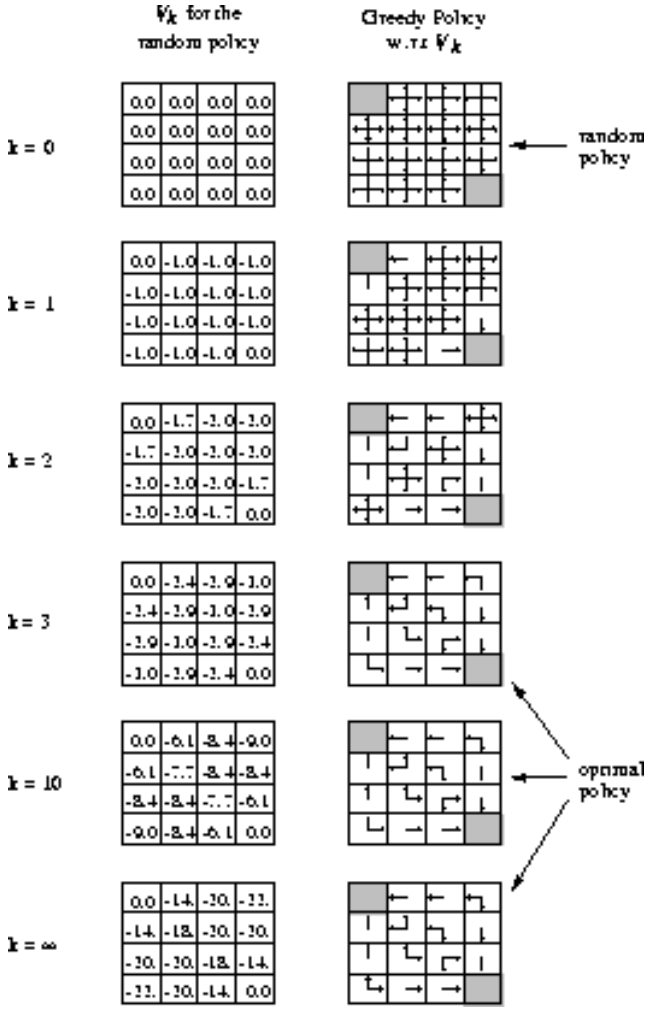

Abbildung 12: Links ist die V-Funktion für eine etwas größere Gridworld als bisher zu sehen. Je höher k ist, desto genauer ist V'. Rechts ist jeweils die dazugehörige Greedy-Policy dargestellt.

ken Bildfolge ist zu sehen, wie Q für die gleichwahrscheinliche (equiprobable) Policy

[7]Sie heisst „greedy“ (engl.: gierig), weil sie immer die vielversprechendste Aktion wählt.
[8]Beweis s. Policy Improvement Theorem in [5] Kapitel 4.7
[9]Dieses Bild ist aus [5] Kapitel 4.1 entnommen.

bestimmt wird. Mit jedem Durchgang k werden die Werte genauer. Rechts davon ist jeweils die dazugehörige Greedy-Policy dargestellt. Nach dem dritten Durchgang wird zwar Q' weiterhin verfeinert, aber die Policy ändert sich nicht mehr. Es reicht offensichtlich Q' etwas in Richtung von Q zu bewegen um π' bestimmen zu können. Anhand dieser neuen, hoffentlich besseren Policy, kann man ein neues Q" ableiten. Zum Schluss erhält man die optimale Policy. Dieses Prinzip wird als **Generalized Policy Iteration**, kurz **GPI** bezeichnet und ist Grundlage vieler Lernalgorithmen für RL.

3.3.1 ϵ-Greedy

Bei der Ableitung der neuen Policy π' von Q' ist noch eine Feinheit zu beachten. Zuvor sagte ich zwar, daß man immer die die optimale Aktion wählen sollte, aber dies führt zu einem Problem. Ginge man so vor, so würde man sich jeder Chance berauben, weitere Informationen über die bisher noch nicht benutzen Aktionen zu erhalten (s. Problem Nummer 1 und 2 zu Beginn des Kapitels). Deswegen bedient man sich des folgenden Tricks. Man wählt in jeder Situation meistens die optimale Aktion, außer mit der Wahrscheinlichkeit ϵ willkürlich eine andere. So ist sichergestellt, dass der Agent kontinuierlich lernt. Die so vorgehende Policy wird als **ϵ-Greedy-Policy bezeichnet**.

3.4 SARSA

Der SARSA-Algorithmus ist eine Möglichkeit, Q* zu bestimmen. Er baut auf dem eben vorgestellten GPI-Prinzip auf. Zu Beginn gibt man eine willkürliche Policy an, die immer weiter verfeinert wird, bis daraus die optimale Policy geworden ist. Bei SARSA dient dieselbe Policy auch gleichzeitig zum Entscheidungsfinden. Ist das der Fall, so spricht man von einem **On-Policy**-Verfahren[10]. SARSA funktioniert, indem es den Fehler zwischen der alten Schätzung von Q(s,a) und der des Nachfolgepaars Q(s',a') bestimmt, also (Q(s',a') - Q(s,a)). Der so berechnete Fehler beruht auf den Werten zweier Schätzungen für zwei verschiedene Zeitpunkte: Q(s,a) bezieht sich auf den Zeitpunkt t und Q(s',a') auf den Zeitpunkt t+1. Wenn Lernen auf dieser Art der Fehlerberechnung basiert, so spricht man von **Temporal-Difference-Learning**, kurz **TD-Learning** (s. Buch [5] Kapitel 6). Damit gehört SARSA zur Gruppe der TD-Algorithmen.
Damit dem weiter in der Zukunft liegenden Wert nicht zu viel Bedeutung beigemessen wird, taucht auch hier wieder der schon angesprochene Discount-Faktor γ auf, ein konstanter Wert zwischen null und eins. Die neue Schätzung von Q(s,a) ergibt

[10] Im Gegensatz dazu, gibt es auch ein Off-Policy-Verfahren, bei dem eine Policy zum Entscheidungsfinden benutzt wird und eine andere evaluiert. Dieses Prinzip wird z.B. beim bekannten Q-Lernen genutzt.

sich aus der Belohnung r, die durch Ausführen von Aktion a in Situation s erlangt wird, plus dem Fehler $\gamma * Q(s', a') - Q(s, a)$. Dieser Wert wird zusätzlich mit einer konstanten α (auch ein Wert zwischen null und eins) gewichtet. Insgesamt ergibt das:
$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)]$
Der Wert eines Situations-Aktions-Paares wird also durch die Berechnung rechts neben dem Pfeil verfeinert. Dieser Teil wird als **Backup** bezeichnet. Man beachte, dass es für die Bestimmung des Backups eines beliebigen Situations-Aktions-Paares nicht nötig ist, bis zum Ende der Episode abzuwarten. Der gesamte Backup beruht nur auf Schätzungen. Damit ist Problem Nummer 3 gelöst (siehe Liste der Probleme zu Beginn des Kapitels). Außerdem ist auch die Kenntnis von δ nicht erforderlich. Anstatt den Folgezustand mühsam vorherzusagen zu versuchen, führt SARSA einfach die Aktion aus und beobachtet Belohnung und Folgezustand. Der Q-Wert jedes Situations-Aktions-Paares des finalen Zustandes ist übrigens null.

Der komplette Algorithmus für SARSA[11][12]:

Initialisiere Q(s,a) willkürlich
Wiederhole (für jede Episode):
    initialisiere s
    Wähle a aus s gemäß der benutzen Policy
    Wiederhole (für jeden Schritt der Episode):
        führe Handlung a aus,
        stelle Belohnung r und Nachfolgesituation s' fest
        Wähle a' aus s' gemäß der benutzten Policy
        $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)]$
        $s \leftarrow s'; a \leftarrow a';$
    bis s Zielzustand ist

Dass SARSA tatsächlich mit der Wahrscheinlichkeit eins gegen Q* konvergiert, wurde von Satinder Singh (s. [5] Kapitel 6.4) behauptet. Der Beweis allerdings ist noch nicht veröffentlicht. Trotzdem entschied ich mich für diesen Algorithmus, da er ein On-Policy-Verfahren ist, was für meine Ansprüche sehr passend war und ich ihn für recht leicht verständlich und einfach implementierbar hielt. Außerdem läßt er sich gut mit den sogenannten „Eligibilty Traces“ kombinieren, die Thema des nächsten Absatzes sein werden.

[11] s. Kapitel 6.4 in [5]
[12] Für die neue Schätzung eines Q-Wertes wird das Tupel (s,a,r,s',a') gebraucht, daraus ergibt sich der Name SARSA.

3.4.1 Eligibilty Traces

Eligibiltiy Traces[13] helfen einige TD-Algorithmen allgemeiner und effizienter zu gestalten. Bei SARSA wurde z.B. in jedem Lernschritt nur das zum Zeitpunkt t aktuelle $Q(s,a)_t$ aufgrund des TD-Fehlers neu berechnet. Besser wäre es, das neue Wissen (also den berechneten Fehler) auch an die Aktions-Situations-Paare davor weiterzugeben, also an $Q(s,a)_{t-1}$, $Q_(s,a)_{t-2}$ etc.. Natürlich sollen sie, je weiter sie zeitlich entfernt liegen, umso weniger verändert werden. Genau diese Idee wird mit Eligibiltiy-Traces, kurz E-Traces, verwirklicht. Ein solcher E-Trace protokolliert das Eintreten bestimmter Ereignisse, z.B. das Befinden in einer bestimmten Situation, oder das Ausführen einer bestimmten Handlung in einer Situation. Soll Lernen stattfinden, so hilft diese Aufzeichnung, die in Frage kommenden Zustände, deren Werte besonders verändert werden sollen, zu bestimmen. Wie wir bisher gesehen haben, funktioniert Lernen bei RL meist so, dass versucht wird, V oder Q zu bestimmen, um die optimale Policy zu finden. Beim Lernen mit E-Traces ordnet man jeder Situation (wenn man V berechnen will), oder jedem Aktions-Situations-Paar (sofern es sich um die Berechnung von Q handelt), einen E-Trace zu.

Definition 3.4 (Eligibility Trace) $e_t(s) \in \mathbb{R}^+$

Wobei t für den Zeitpunkt steht, und s die Situation bezeichnet. Jedesmal, wenn ein Zustand (oder ein Aktions-Situations-Paar) besucht wird, erhöht man seinen E-Trace um eins. Der E-Trace aller zerfällt dabei um den Faktor $\gamma\lambda$, vgl. Abbildung 13[14] Dabei ist γ wie gehabt der Discount-Faktor, und λ soll Trace-Decay-Parameter[15] heißen[16] , nach [5], Kapitel 7.3. Wie zu erkennen ist, haben kürzlich besuchte Situationen einen höheren E-Trace-Wert als solche, deren Vorkommen schon länger her ist. Dieser Wert kann nun in einen TD-Algorithmus eingearbeitet werden, um oben angesprochene Idee zu verwirklichen.

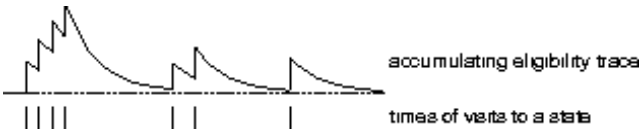

Abbildung 13: Verhalten von E-Traces über Zeit.

[13] eligibility engl. = Eignung, trace engl. = Spur
[14] Dieses Bild ist [5], Kapitel 7.3 entnommen.
[15] decay engl. = Zerfall, Verfall
[16] Die Bedeutung von λ ist viel weitreichender, für weitere Informationen s. [5] Kapitel 7.1 und 7.2.

3.4.2 *SARSA*(λ)

SARSA(λ) ist die Kombination des Algorithmus SARSA mit oben beschriebenen Eligibilty-Traces. Der komplette Algorithmus sieht folgendermassen aus:[17]

Initialisiere Q(s,a) willkürlich und e(s,a)=0, für alle s,a
Wiederhole für jede Episode:
    Initialisiere s, a
    Wiederhole für jeden Schritt der Episode:
        Führe Aktion a aus und beobachte Belohnung r
        und Folgezustand s'
        Wähle a' von s' gemäß der von Q veränderten Policy
        (z.B. ϵ-Greedy)
        $\delta \leftarrow r + \gamma Q(s', a') - Q(s, a)$
        $e(s, a) \leftarrow e(s, a) + 1$
        Für alle s,a:
            $Q(s, a) \leftarrow Q(s, a) + \alpha \delta e(s, a)$
            $e(s, a) \leftarrow \gamma \lambda e(s, a)$
        $s \leftarrow s'; a \leftarrow a'$
bis s terminal ist.

4 Durchführung

In diesem Kapitel zeige ich, wie ich die Ideen und Techniken des RL's auf mein Problem des Roboterfußballs anwende. Dazu müssen Lernziel und Belohunung, Aktionen und Episode definiert werden.
Meine Idee war, das Spielverhalten eines Agenten zu verbessern, indem er gute Entscheidungen bezogen auf die, für ihn zur Verfügung stehenden Aktionen zu treffen lernt. Wenn er seine Sache gut macht, sollte er den Ball nicht verlieren. Tut er es aber doch, so ist eine unerwünschte Situation eingetreten, die es zu vermeiden gilt. Das erreiche ich, indem ich dem Agenten für diese Situation eine Bestrafung von -100 erteile. Daraus ergibt sich eine logische Einheit, nämlich die Zeit, von Annahme des Balles durch den Spieler bis zum Verlust des Balles. Der Agent wird - motiviert durch die Bestrafung - lernen, die unerwünschte Situation nicht eintreten zu lassen. Die einzige Art und Weise, wie er das bewerkstelligen kann, ist, den Ball so geschickt zu dirigieren, daß der Gegner ihn nicht bekommt. Also muß er genau das Lernen, was ich mir vorgestellt habe, nämlich eine kluge Wahl seiner Handlungen.[18]

[17] entnommen aus [5] Kapitel 7.5
[18] Falls sich später herausfstellen sollte, daß dies nicht der Fall ist, sollte dieser Punkt nochmal überdacht werden.

4.1 Aktionen

Die Menge, der dem Agenten zugänglichen Aktionen zu bestimmen ist einfach, sie besteht nur aus „dribbeln“ und „passen“, also:
$A = \{dribbeln, passen\}$

4.2 Episode

Es bietet sich an, die angesprochene Zeiteinheit als Episode zu betrachten. Startsituation ist erstmalige Annahme des Balles durch den RL-Spieler und finaler Zustand Verlust des Balles. Verlust des Balles bedeutet, Übernahme durch den Gegner, oder aber Fehlspiel, wie Abseits, Aus, Foul etc. welches ebenfalls mit Abgabe des Balls an den Gegner endet. Ein Tor hingegen beendet eine Episode nicht. Startsituation oder erstmalige Annahme bedeutet, daß eine Episode erst dann beginnt, wenn der RL-Spieler den Ball erhält, wenn er:
a) den Ball vorher noch nie gehabt hat, oder
b) davor Gegner in Ballbesitz war.
Insbesondere bedeutet dies, dass eine Episode nicht beginnt, wenn der RL-Spieler den Ball gepasst hat, und er ihm von einem Mitspieler wieder zugespielt wird. Ich erwarte von meinem „ausgelernten“ RL-Spieler, dass die Dauer einer Episode (gemessen in Zyklen) länger ist, als die eines ungelernten. Das ist zugleich das Ziel, nämlich die durchschnittliche Episodendauer zu steigern.

4.3 Reduzierte Strafe

Da als gültige Spielaktion „passen“ zugelassen ist, ergeben sich einige Probleme bezogen auf die Episode. So kann es vorkommen, dass der RL-Agent den Ball abspielt, er noch eine Weile in Teambesitz ist, aber dann einer seiner Mitspieler den Ballverlust verschuldet. Inwiefern soll der RL-Agent dafür verantwortlich gemacht werden? Meine Idee ist folgende: Hat er den Ball abgespielt, so wird gezählt wie viele Zyklen lang der Ball noch in Teambesitz ist, folgt der Ballverlust, so erhält der RL-Agent eine gemilderte negative Belohnung, abhängig von der Zeit, die nach Abspiel des Balles bis hin zu dessen Verlust vergangen ist. Je länger diese Zeit war, desto geringer fällt die Bestrafung aus. Das erreiche ich einfach, indem ich die Belohnung durch die verstrichene Zeit teile.

4.4 Situation

Nachdem ich die Idee verworfen hatte, Situationen mittels Neuronaler Netze zu bestimmen, überlegte ich folgendes: Das Fußballfeld kann als zwei-dimensionales Feld angesehen werden, auf dem Objekte (Spieler, Ball, Tor etc.) platziert sind. Die Situationen würden einfach über die Positionen der Objekte definiert. Das komplette

Feld abzudecken, mit allen Spielern aus zwei Mannschaften, ließe den Zustandsraum explodieren. Also mußte ich reduzieren. Ich beschloß, mich auf den RL-Spieler zu konzentrieren und nur die Koordinaten des nächsten Mitspielers und des nächsten Gegenspielers als relevant zu erachten. Anstatt das ganze Fußballfeld zu betrachten, verkleinerte ich den Bereich auf ein selbstdefinierbares Gitter, in dessen Mittelpunkt der RL-Agent steht, s. Bild 14. Selbst wählbar sind die Größe des Gitters und die

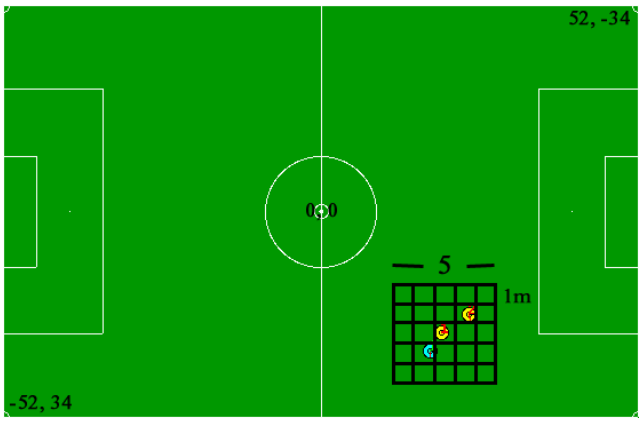

Abbildung 14: Eine „Situation“: Der RL-Spieler mit der Nummer 1 befindet sich an x=20 y=20, der nächste Mitspieler an Stelle x=24 y=17 und der nächste Gegner an x=18 y=23.

Granularität, im Bild z.B ist die Größe des Gitters fünf und die Granularität beträgt einen Meter. Da davon ausgegangen wird, dass der RL-Spieler gleich weit in alle Richtungen sehen können sollte, muß die Gittergröße immer ein ungerader Wert sein. In meinem Programm gehe ich so vor, daß ich die Koordinaten vom nächsten Mitspieler und nächstem Gegner in Abhängkeit vom RL-Spieler zunächst in Gitterkoordinaten umrechne und diese dann für ein eindimensionales Array bestimme.
Bsp.: Angenommen der
RL-Spieler hat die Koordinaten 20, 20,
der nächste Mitspieler, kurz NP, befinde sich an Position 24, 17
und der nächste Gegner, O, bei 18, 23.
Zusätzlich sei das Gitter der Größe 5 mit Granularität 2. Die Fußballfeld-Koordinate 20,20 entspricht der Gitter-Koordinate 2,2. Demnach ist der Mitspieler an Stelle 4,0 im Gitter zu platzieren und der Gegner an Position 1,3 (vgl. Abbildung 15). Die Umrechnung von Gitter- in eindimensionale Koordinaten ergibt den Wert 4 für den RL-Agenten, 12 für den nächsten Mitspieler und 16 für den nächsten Gegner vgl. 16). Die verschiedenen Situationen werden über die eindimensionalen Positionen des nächsten Mitspielers und des nächsten Gegenspielers definiert. Im obigen Beispiel wäre das Tupel, welches die Situation bestimmt (12,16). Die Anzahl der verschiedenen Situationen richtet sich nach der gewählten Gittergröße. Bei einer Gittergröße

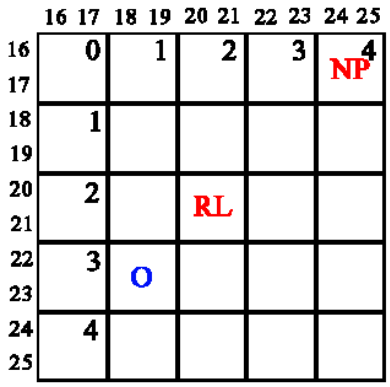

Abbildung 15: Umrechnung von Fußballfeld-Koordinaten (die Zahlen von 16 bis 25) in Gitter-Koordinaten.

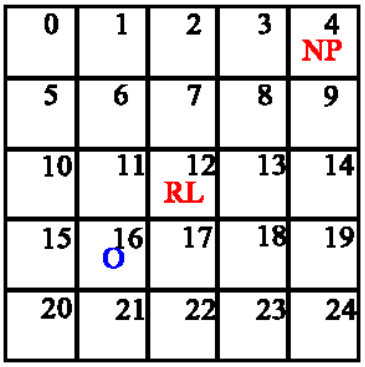

Abbildung 16: Umrechnung von 2-D in 1-D Koordinaten.

von x gibt es x*x verschiedene Positionen, die ein Agent einnehmen kann. In Bild 16 z.B. ist die Gittergröße 5 und somit kann jeder Agent 25 verschiedene Positionen, 0-24, einnehmen. Dazu kommt die spezielle Position „undefiniert“ für den Fall, dass der Agent sich gar nicht innerhalb des Gitters befindet. Die Variable Agent kann also (x*x)+1 verschiedene Werte annehmen. Da wir zwei Agenten haben, ergeben sich k^n verschiedene Kombinationsmöglichkeiten beider auf dem Gitter, wobei k = (x*x)+1 ist und n = zwei = Anzahl der Agenten. Bei einem Gitter der Größe fünf hat man es mit 676 möglichen Kombinationen zu tun. Jede dieser Kombinationen stellt eine gültige Situation dar. Natürlich muß pro Situations-Aktions-Paar ein Q-Wert und ein E-Wert abgespeichert werden. Bei zwei Aktionen macht dies eine Größe von insgesamt 2705 Einträgen. Beim Testen meiner Implementation arbeite ich mit einem Gitter der Größe 15 á 1m, also 51 076 Situationen. Der Agent kann dabei eine Fläche von $225m^2$ „überblicken“.

4.5 Aufruf SARSA

Jetzt ist alles vorhanden, um Sarsa(λ) benutzen zu können. Zunächst werden die Q- und E-Werte für alle Situationen mit 0 initialisiert[19]. Der RL-Agent funktioniert folgendermaßen:

[19] Eigentlich sollen diese Werte willkürlich initialisiert werden. Da ich aber nicht weiß, welcher Zahlenbereich dafür geeignet wäre, setze ich sicherheitshalber alles auf null.

```
In jedem Zyklus tue:
    \* Variable, die die Zeit zählt, um gegebenenfalls die
     Strafe zu vermindern (vgl. ReduzierteStrafe)*\
     static int discounter = 1
     Wenn Episode endet:
          s' = -2
          a' = -2
          reward = -100 / discounter
          ruf SARSA auf mit Parametern (s,a,reward,s',a')
          verlasse Zyklus
     Wenn Episode beginnt:
          discounter = 1
          s = bestimme Situation
          a = bestimme zu s gehörige Aktion epsilon-greedy, man
          kann der Methode übergeben, wie hoch epsilon sein
          soll.
          Wenn der RL-Spieler keinen Passpartner finden kann,
          oder er sich selbst als Passpartner erkennt:
               s = dribbeln
          Wenn Möglichkeit zum Torschuss besteht:
               schiesse auf Tor
          Führe Aktion aus und verlasse Zyklus
     Wenn mitten in Episode:
          Wenn RL-Agent aktiv:
               s = Situation
               a = Aktion
               s' = bestimme Situation
               a' = bestimme zu s' gehörende Aktion,
               epsilon-greedy
               reward = 0
               ruf SARSA auf mit Parametern (s,a,reward,s',a')
               discounter = 1
          Wenn nicht aktiv (z.B. wenn er den Ball an seinen
          Mitspieler gepasst hat.)
               discounter++;
```

Die gelernten Daten werden in eine Datei geschrieben („values.txt“). Wenn man den RL-Agenten neu startet wird zunächst im aktuellen Ordner nach dieser Datei gesucht. Ist sie vorhanden wird sie benutzt, sprich die darin enthaltenen Daten werden eingelesen, wenn nicht, wird eine neue angelegt.

5 Testergebnisse

Um die Performanz des RL-Agenten testen zu können, habe ich folgenden Aufbau gewählt: Der Agent, ein Mitspieler und ein Gegner werden auf jeweils fixe Positionen auf das Fußballfeld gesetzt (s. Abbildung 17) und eine Episode gestartet. Nach

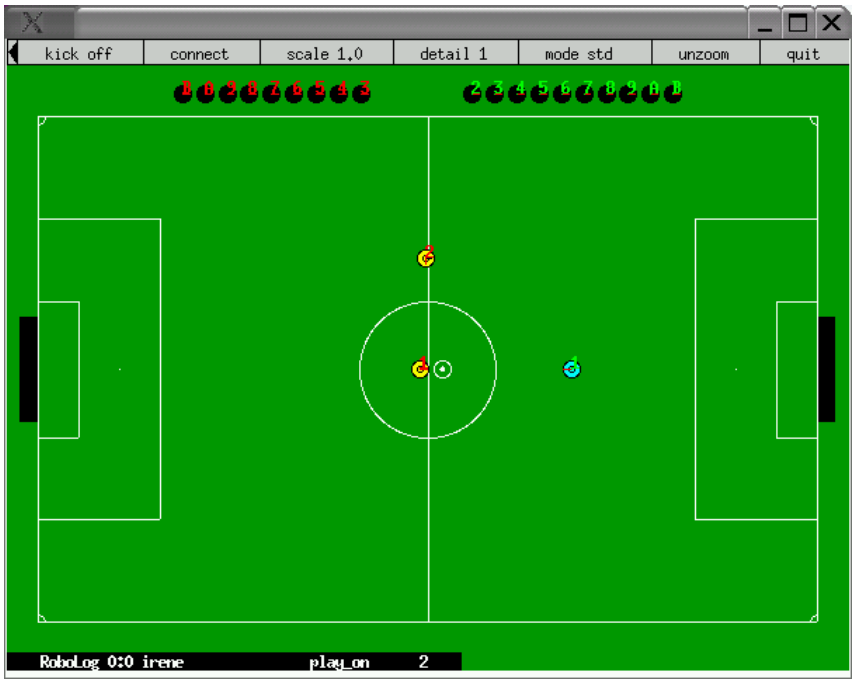

Abbildung 17: Testumgebung.

jedem Episodenende und jedem Tor werden die Spieler wieder auf diese Initialpositionen zurückgesetzt. Zusätzlich wird die Episodendauer - sofern es sich zuvor um ein Episodenende handelte - in eine Datei geschrieben. Beim ersten Test setzte ich meinen RL-Agenten an die Position eins, d.h. der gelbe Spieler mit der Nummer eins in Bild 17, und lasse ihn 20 Episoden lang lernen. Die Ergebnisse sind in Grafik 18 dargestellt. Auf der Y-Achse ist die Dauer der Episoden, gemessen in Zyklen, abgetragen und auf der X-Achse die Anzahl der Episoden. Um einen Vergleichswert zu haben, habe ich denselben Aufbau mit einem Spieler, der Entscheidungen nicht lernt, sondern auf Grund des Zufalls trifft an gleicher Position wiederholt. Diese Werte sind in der Grafik in blau dargestellt, die des RL-Agenten in schwarz. Dasselbe mit dem Lerner an Position zwei ergibt die in 19 dargestellten Resultate. Beim nächsten Testlauf habe ich mich nicht auf 20 Episoden beschränkt, sondern habe 600 Episoden lang lernen lassen. Bei den Y-Werten handelt es sich um die durchschnittliche Zykellänge aus jeweils 10 Episoden. In Abbildung 20 ist die Performanz des Lernes

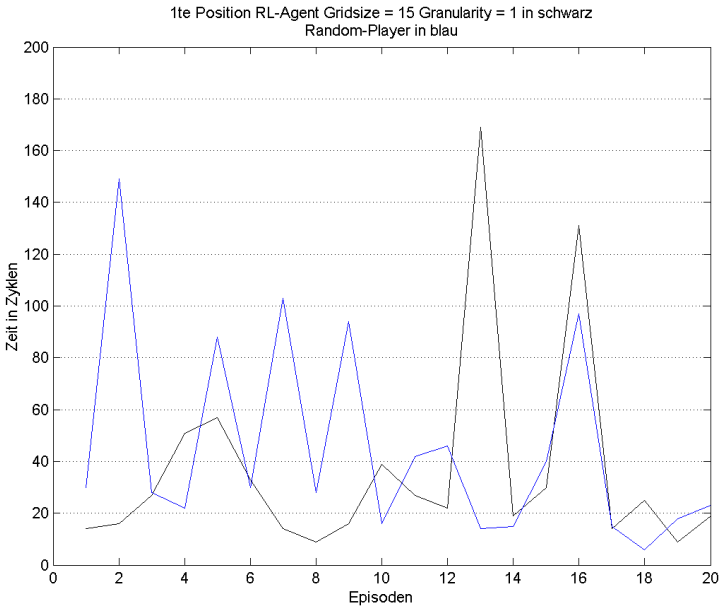

Abbildung 18:

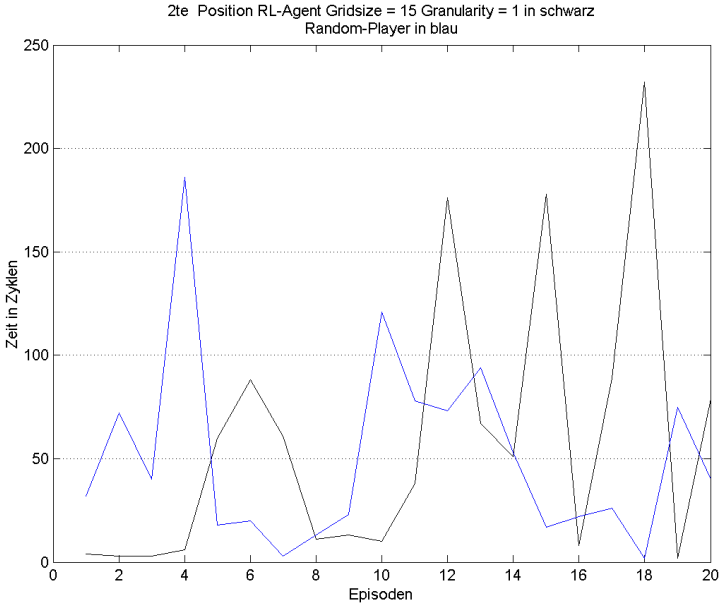

Abbildung 19:

an Position eins dargestellt und im Vergleich dazu die des Zufallsspielers in Bild 21. Analog dazu dasselbe Setting mit dem Lerner an 2ter Position, Bild 22, sowie der Vergleichsgrafik 23 mit dem Zufallsspieler.

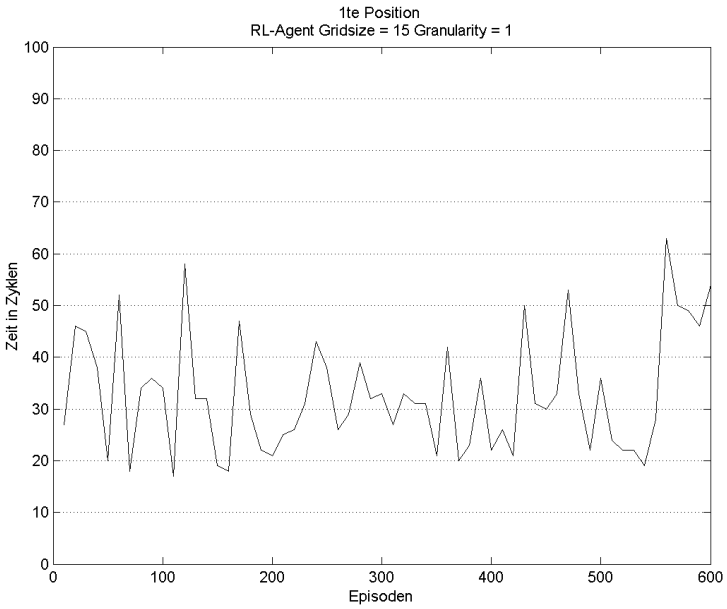

Abbildung 20:

6 Fazit

Wie man den Grafiken entnehmen kann, hat sich meine Hoffnung nicht erfüllt, und die Episodendauer eines gelernten Spielers unterscheidet sich kaum von der des Zufallsspielers. Interessant ist das starke Oszillieren der Graphen beider Spieler, also des Lerners sowie des Zufallsspielers. Die Erklärung dafür ergibt sich aus der Testumgebung. Wenn der Agent als Nummer eins beginnt, muß er den Gegner umspielen. Dazu kann er den Ball frühzeitig an seinen Partner passen, oder erst etwas später, wenn er sich näher am Gegenspieler befindet. Passt er früh, endet die Episode fast immer mit einer durchschnittlichen Dauer. Passt er spät, so endet die Episode entweder sofort, oder sie wird sehr lang.
Um das Verhalten des Spielers zu verbessern, kann man natürlich versuchen die benutzten Parametern λ, α und γ zu justieren. Allerdings denke ich, daß die Performanz des Lerners vor allem auf zwei Dinge zurückzuführen ist. Nämlich zum einen eine unglückliche Formulierung des Lernziels und zum anderen eine zu reduzierte

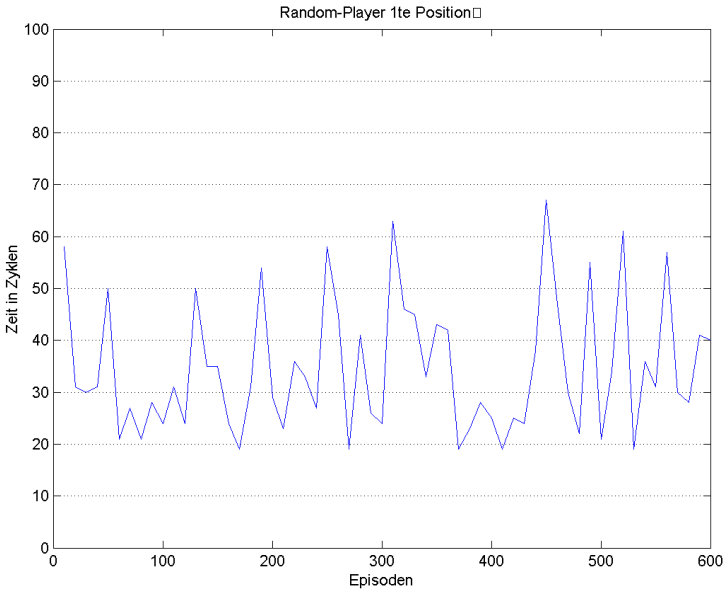

Abbildung 21:

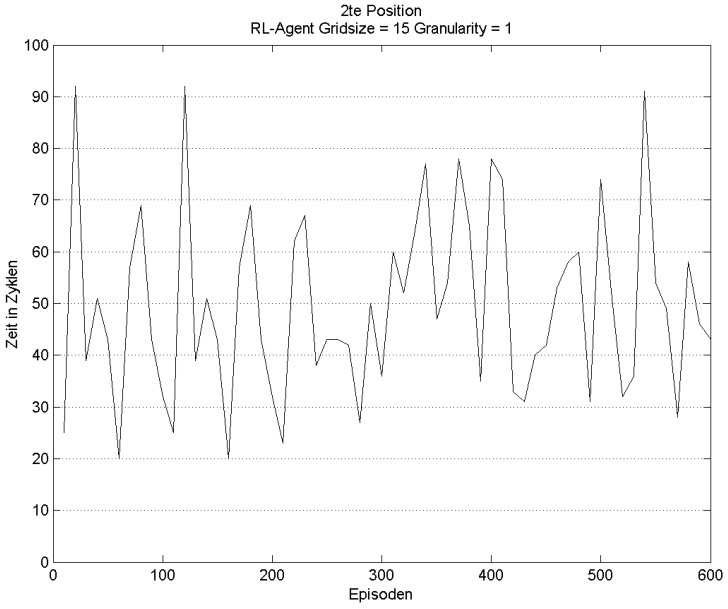

Abbildung 22:

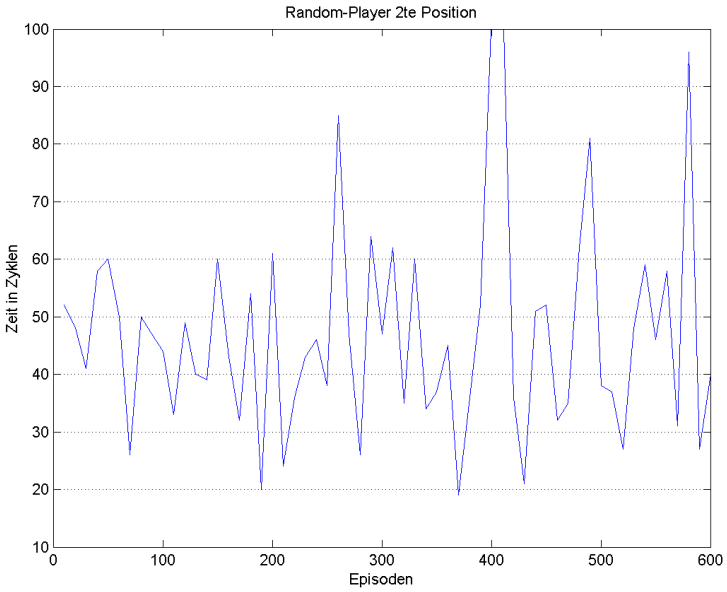

Abbildung 23:

Form der Situationsrepresentation.

6.1 Lernziel

Zu Beginn nahm ich an, mein Spieler würde lernen, wie er am besten den Gegner ausspielen müßte. Dass er das nicht tat, liegt daran, dass Dribbeln und Passen schon fix implementiert sind. Er kann diese zwei Aktionen ausführen, aber sie nicht ändern, also z.B. nicht in eine etwas andere Richtung dribbeln. Könnte er das, so würde man sicherlich bessere Ergebnisse erzielen, allerdings würde mit mehr möglichen Aktionen pro Situation auch das Problem komplexer. Dann dachte ich, dass der Agent lernen würde, immer sehr spät zu passen, so dass er wenigstens die Chance auf eine lange Episode hätte. Stattdessen lernte er, so zu reagieren, dass eine durchschnittliche Zykeldauer erreicht wurde. Nach und nach wurde der Wert für die dafür verantwortlichen Situations-Aktionspaare immer niedriger, bis es ihm besser erschien, anders zu reagieren. Also passt er ab und zu spät, erreicht damit auch einige lange Episoden, aber dann endet er mit einer sehr kurzen Dauer und dieses Situations-Aktions-Paar (und auch die davor liegenden) bekommen ein sehr hohes negatives Reinforcement, so dass er wieder das Verhalten wählt, welches zu einer mittelmäßigen Episode führt. So erklärt sich das Auf und Ab des Graphen, zumindest für den Lerner an Position eins. Für Position zwei verhält es sich ähnlich.

Im Grunde genommen hat der RL-Agent damit auch das Lernziel erreicht, denn er hat gelernt meistens so zu handeln, dass die Epsiodendauer optimal ist, in diesem Fall scheint das wohl eine durchschnittliche Länge zu sein. Interessant wäre es, zu testen, was passiert, wenn der Agent jedes Aktions-Situationspaar nur nach dem bestmöglichen (also der damit längsten erreichten Episode) Ergebnis beurteilte. Sicherlich würde er dann das späte Passen vorziehen.
Schlussfolgerung: Er hat das Lernziel erreicht, aber es drückt nicht aus, was wir eigentlich wollen.

6.2 Situationsbeschreibung

Ein weiterer Grund für die schlechte Performanz könnte die Situationsbeschreibung sein. Sie war stark limitiert, dadurch, dass sie sich nur aus der Position des nächsten Mitspielers und Gegners ergab. Das resultiert darin, dass man für ein Situations-Aktions-Paar mal eine hohe Belohnung erhält, und mal eine sehr niedrige. Auch als Mensch wüßte man in einem solchen Fall nicht wie man sich verhalten sollte. Um die Situationen besser voneinander differenzieren zu können, müßte man mehr Faktoren berücksichtigen, z.B. die Position, Geschwindigkeit und Richtung des Balles. Man muß sich allerdings immer bewußt sein, dass mehr Parameter den Zustandsraum immens vergrößern und sich die Frage stellen, wie man diesen dann handhaben kann. Eine Tabelle wäre sicherlich nicht mehr die geeignete Speicherform. Eine gute Situationsbeschreibung zu finden und zu realisieren, erscheint mir alles andere als trivial.
Naheliegende Ideen für die Verbesserung der aktuellen Architektur wären z.B.:
- Das Grid innen feiner zu granulieren, als aussen.
- Um die Anzahl der Situationen zu reduzieren, könnte man Situationen, die sich durch Rotation um den Mittelpunkt ineinander überführen lassen, zu einer einzigen zusammenfassen.
Oder - meiner Meinung nach die bessere Lösung - man überlegt sich eine ganz andere Art, den Zustandsraum darzustellen. Z.B. durch Kohonen-Netze.
Schlussfolgerung: Die Situationsbeschreibung enthält zu wenig Informationen.

6.3 Eignung des RL für Roboterfußball

Die Frage, ob sich Reinforcement Learning zum Entscheidungsfinden beim Roboterfußball eignet kann trotz der hier erzielten Ergebnisse nicht mit einem Nein beantwortet werden. Ich bin mir sicher, dass, wenn man es schafft die beiden oben beschriebenen Probleme zu lösen, man prinzipiell gut mit diesem Verfahren arbeiten

kann[20]. Meiner Meinung nach liegt die Hauptschwierigkeit, oder auch Kunst, beim RL darin, eine geeignete Situationsbeschreibung zu finden. Schön wäre es, wenn es dafür eine gute, allgemeingültige Anleitung gäbe. Da diese nicht existiert, muß jeder, der Probleme mit RL lösen will, viel Zeit für eine Evalution der Situationsbeschreibung einplanen. Allerdings gibt es noch eine offene Frage, die bleibt. Eine besondere Eigenschaft von Reinforcement Learning, ist, dass Handlungssequenzen gelernt werden. Ausgehend von einer bestimmten Situation wird in die Zukunkt geschaut, was wir aus ihr heraus erreicht werden könnte. Dementsprechend wird gehandelt. Die Frage ist, ob das bei einem Spiel, in dem sich Situationen so schnell ändern, wie beim Fußball überhaupt nötig ist. Angenommen wir würden zwei Neuronale Netze programmieren, je eins für jede Aktion (Passen, Dribbeln). Wir würden jedes Netz so trainieren, dass es in der Lage wäre, für eine beliebige Situation vorherzusagen wie wahrscheinlich der Ballverlust für die jeweilige Aktion wäre. Also wir würden z.B. als Netzinput die aktuelle Situation eingeben und das Netz für die Aktion „Passen“, gäbe den Output, dass mit 0.1die Zahl 0.8 ausg. Also wäre es besser zu Dribbeln. Wir bestimmen hier die Entscheidung nur auf Grund der aktuellen Situation. Nicht eingebettet in einen Handlungskontext. Angenommen wir hätten einen Agenten, der auf diese Weise seine Entscheidungen träfe und einen trainierten RL-Agenten (natürlich einen, bei dem die oben beschriebenen Probleme zuvor gelöst worden sind) und liessen beide gegeneinander antreten. Es wäre interessant zu sehen, welcher von beiden die besseren Resultate erzielte. Ist es nicht der RL-Agent, so könnte man sich - zumindest bei der von mir gestellten Aufgabe - viel Mühe sparen, da das Problem dann einfacher und auch noch besser zu lösen ginge.

[20] Ein berühmtes Beispiel für das Potenzial von Reinforcement Learning, ist der von Backgammon-Lerner von Gerry Tesauro (vgl. [5] Kapitel 11.1).

Literatur

[1] http://de.wikipedia.org/wiki/Gesetz_der_gro%DFen_Zahl.

[2] www.heise.de/tp/deutsch/special/game/9212/1.html.

[3] www.tecchannel.de/software/1162/1.html.

[4] www.robocup.org.

[5] Richard S. Sutton Andres G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998. http://www.anw.cs.umass.edu/~rich/book/the-book.html.

[6] Tom M. Mitchell. *Machine Learning*. Prentice Hall, 1997.