

**Seminar im Rahmen der
Projektgruppe
Paderkicker III
Reinforcement Learning**

Andreas Koop

Universität Paderborn

Wintersemester 2003/04

Betreuer: Dr. Bernd Kleinjohann, Dipl-Inf. Dirk Stichling

18. Dezember 2003

Inhaltsverzeichnis

1	Reinforcement Learning	3
1.1	Einführung und Motivation	3
1.2	Grundlagen	4
1.2.1	Agent-Umgebung Interaktion	4
1.2.2	Episodische/ kontinuierliche Tasks	5
1.2.3	Exploration vs. Exploitation	7
1.2.4	Die Markov Eigenschaft	7
1.2.5	Markov Entscheidungsprozesse (MDP)	8
1.3	Hauptbestandteile	9
1.3.1	Strategie (Policy π)	9
1.3.2	Vergütungsfunktion (Reward function)	9
1.3.3	Wertefunktion (Value function)	10
1.3.4	Umgebungsmodell (optional)	12
1.4	Lernverfahren	12
1.4.1	Dynamisches Programmieren	12
1.4.2	Monte Carlo Methoden	13
1.4.3	Temporal Difference Methoden	13
1.5	Erfolgsgeschichten	16
1.5.1	Backgammon	16
1.5.2	Karlsruher Brainstormer	17
1.5.3	Fahrstuhlsteuerung	17
1.6	Fazit & Ausblick	17

1 Reinforcement Learning

1.1 Einführung und Motivation

Noch immer gibt es zahlreiche Probleme, die nur mit einer speziell dafür angefertigten Software oder noch gar nicht gelöst werden können. Insbesondere im Bereich autonomer Agentensysteme (Flugzeugsteuerung, Robotik, etc.) sind diese Probleme nicht einfach zu lösen, da nur sehr schwierig festzustellen ist, welche Aktion ein Programm in einer bestimmten Situation ausführen soll. Dabei liegt es nicht an mangelnder Rechen- oder Speicherleistung, sondern in der Tat daran, dass ein Programm nicht weiß, wie es sich verhalten soll. Es wäre von großer Hilfe, wenn es durch eine geschickte Versuch-und-Irrtum-Methodik selbst eine Lösung erlernen könnte.

Das Reinforcement Learning (RL) ist ein Bereich des Maschinellen Lernens und basiert im Wesentlichen auf der Idee, dass ein Agent durch Belohnung bzw. Bestrafung selbständig – ohne einen Trainer – seine Arbeitsweise zu einer besseren hin verändert. Im Gegensatz zum *Beaufsichtigten Lernen*, z.B. Künstlich Neuronale Netzen (KNN), lernt der Agent hier wirklich autonom im Hinblick auf ein vorher definiertes Ziel. Reinforcement Learning steht daher für echtes “Erlernen” und nicht einfach nur für “Anlernen” von Sachverhalten. Es ist 1989 von Watkins [1] entdeckt worden und entwickelte seit den 90er Jahren ein stark wachsendes Interesse, das bis heute nicht abgenommen hat.

Die Motivation für diesen Ansatz stammt wie bei so vielen technischen Innovationen von der Natur: Durch die Interaktion mit ihrer Umwelt passen Lebewesen ihr Verhalten entsprechend eines Feedbacks an. Dabei gibt es besonders wünschenswerte oder besonders schlechte Folgezustände der Umwelt. Auf der einen Seite “Belohnung” und auf der anderen Seite “Bestrafung”. Desweiteren benötigen Lebewesen keinen expliziten Lehrer oder jemanden, der die richtige Handlung voführt. Vielmehr wird die durchgeführte Handlung mit “gut” oder “schlecht” bewertet. Das Ziel dabei ist, die Anzahl der positiven Bewertungen zu maximieren.

Ein wichtiges Merkmal von Reinforcement Learning ist, dass es sich um eine Klasse des Maschinellen Lernens handelt und nicht um einen speziellen Lernalgorithmus.

In dieser Seminararbeit werden zunächst die Grundlagen beschrieben, Beispiele gegeben, einige effiziente Algorithmen vorgestellt und schließlich Erfolgsgeschichten genannt, in denen Reinforcement Learning bislang erfolgreich eingesetzt werden konnte. Zum Schluß wird ein Fazit gezogen und ein kurzer Ausblick gegeben.

1.2 Grundlagen

Dieser Abschnitt behandelt grundsätzliche Fragen und soll als Einstieg in die prinzipielle Funktionsweise des Reinforcement Learnings dienen. Zunächst wird ein Einblick in die Interaktion eines Agenten mit der Umwelt gegeben, anschließend einige mathematische Formalismen und Modelle zur Formalisierung des Reinforcement Learning Problems eingeführt.

1.2.1 Agent-Umgebung Interaktion

Nach [2] ist das Reinforcement Learning Problem eine einfache Umrahmung des Problems durch Interaktion zu lernen, um ein bestimmtes Ziel zu erreichen. Der Lerner und Entscheidungsträger ist in diesem Fall der sogenannte *Agent*. Dabei interagiert er mit allen Dingen, die ihn umgeben: seiner Umwelt.

Der Agent in Abb. 1.1 befindet sich in einer unbekanntenen Umgebung und agiert zu diskreten Zeitpunkten $t = 0, 1, 2, \dots$ zunächst nur auf Grund der erfassten Reize. Die über Sensoren wahrgenommenen Reize werden ausgewertet und lösen eine entsprechende Reaktion mit Hilfe der Aktorik aus. Auf der Modellebene bedeutet dies, dass

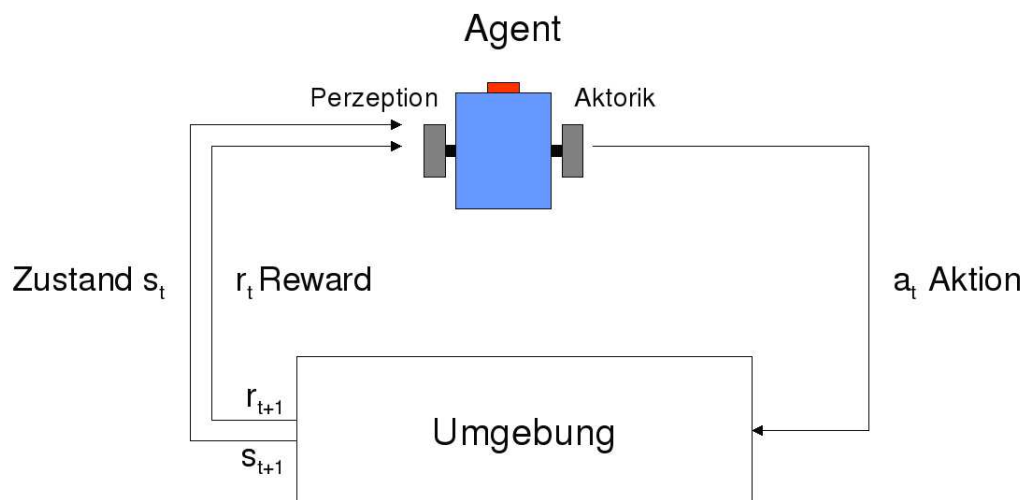


Abbildung 1.1: Agent-/Umgebung-Interaktion

der Agent als Folge einer Aktion a_t aus einer Menge möglicher Aktionen im aktuellen Zustand $A(s_t)$ in einen neuen Zustand s_{t+1} wechselt.

Um das Verhalten des Agenten zu optimieren, bewertet man die Aktion des Agenten

nun in jedem Zeitschritt, was einen Reward $r_{t+1} \in \mathfrak{R}$, sprich eine Vergütung, zur Folge hat. Infolgedessen weiß der Agent, ob seine letzte Entscheidung gut oder schlecht war. Das bedeutet: Falls er in absehbarer Zeit wieder in die gleiche Situation gerät, kann er sein Verhalten auf Grund der letzten Bewertung beibehalten oder gegebenenfalls revidieren und somit eine eventuell bessere Aktion $a_t \in A(t)$ ausprobieren. Dadurch wird es dem Agenten möglich, sich vollkommen eigenständig in einer dynamischen Umgebung zurechtzufinden.

1.2.2 Episodische/ kontinuierliche Tasks

Wie bereits im vorhergehenden Abschnitt angeklungen, ist die primäre Aufgabe oder das *Ziel* des Agenten die Anzahl der Belohnungen zu maximieren, was einer optimalen Aktionsfolge gleich kommt. Wenn man die Reihenfolge der erhaltenen Belohnungen in jedem Zeitschritt als $r_{t+1}, r_{t+2}, r_{t+3}, \dots$ bezeichnet, so lässt sich der *Erwartete Return* R_t im episodischen Fall als

$$R_t = r_{t+1} + r_{t+2} + r_{t+3} + \dots + r_T \quad (1.1)$$

darstellen, wobei T für den finalen Zeitpunkt einer Episode steht. Überall dort also, wo eine Agent-Umgebung Interaktion in mehrere Sequenzen unterteilt ist, spricht man von *Episodischen Tasks*. Dieses können Durchgänge in einem Spiel, Durchläufe in einem Labyrinth oder jegliche Art von Wiederholungen einer Interaktion sein. Im Robocup-Umfeld können Episoden je nach Definition oder Interpretation z. B. einzelne Spielabschnitte bis zur Erzielung eines Tores oder ganze Spiele beschreiben. Dabei endet jede Episode in einem ausgezeichneten Zustand, dem *Endzustand*. Nach dem Neustart wird anschließend von einem wohldefinierten Anfangszustand ein neuer Durchlauf gestartet.

Demgegenüber gibt es in vielen Fällen der Agent-Umgebung Interaktion keine natürliche Aufteilung in Episoden. In diesem Fall spricht man von *Kontinuierlichen Tasks*. Das heißt, dass die Interaktion ohne ein festgesetztes Limit ins Unendliche läuft. Als Beispiel seien hier die kontinuierliche Prozesskontrolle oder eine Anwendung für einen Roboter mit langer Lebensdauer genannt. Um auch im kontinuierlichen Fall den Return R_t zu formalisieren, bedarf es eines neuen Konzeptes, da man mit der Gleichung 1.1 für $T = \infty$ kein aussagekräftiges Ergebnis bekommt. Sutton und Barto verwenden dazu in [2] das Konzept der *Diskontierung*: Mit zunehmender Zeit fließt der erhaltene Reward zum Zeitpunkt t nur noch zu einem Bruchteil in die Berechnung des Returns ein. Mit diesem Ansatz lässt sich der *Diskontierte Return* wie folgt berechnen:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \quad (1.2)$$

Dabei fungiert γ als sogenannter *Diskontfaktor* und nimmt Werte im Bereich $0 \leq \gamma \leq 1$ an. Er bestimmt damit den “gegenwärtigen Wert” zukünftiger Belohnungen. Möchte man weitsichtig agieren, so wählt man einen möglichst hohen Wert für γ , wohingegen man einen kleinen Wert nimmt, falls nur die Belohnungen in unmittelbarer Zukunft von Belang sind.

Zur Verdeutlichung der in diesem Abschnitt vorgestellten Betrachtungsweisen einer Agent-Umgebung Interaktion soll das in [3] vorgestellte Stab-Balancierungs Problem dienen. Abbildung 1.2 zeigt den prinzipiellen Aufbau: Die Aufgabe besteht darin, die

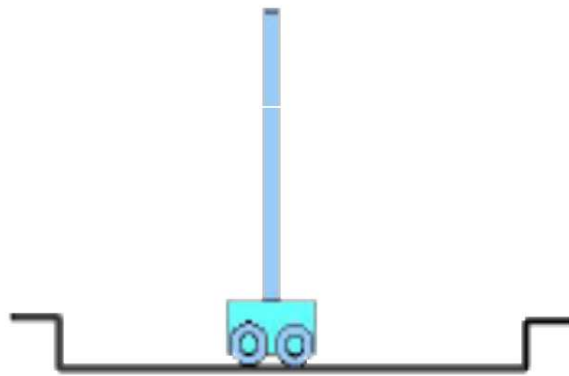


Abbildung 1.2: Balancierung eines langen Stabes

Position x des Wagens innerhalb der Begrenzung so zu steuern, dass der Stab aufrecht, also möglichst in einem Winkel von 90° zur Gravitationsäquipotentialfläche, stehen bleibt. Die Aufgabe gilt als gescheitert, sobald der Stab über einen kritischen Winkel fällt oder der Wagen eine der Begrenzungsmarken tangiert. Um das Problem mit Hilfe von Reinforcement Learning zu lösen, kann es auf 2 unterschiedliche Arten modelliert werden:

Episodisch: (Ende nach Scheitern)

Belohnung = 0 für jeden Schritt vor dem Scheitern, sonst -1

Return = $r_{t+1} + r_{t+2} + r_{t+3} + \dots + r_T$ = Anzahl der Schritte bis zum Scheitern

Kontinuierlich: (Diskontierter Return)

Belohnung = -1 für jeden Schritt vor dem Scheitern, sonst 0

Return = $\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} = -\gamma^k$ für k Schritte vor dem Scheitern

Wie man leicht nachvollziehen kann, leisten beide Ansätze das gleiche: Sie maximieren

den Return-Wert in Abhängigkeit von der Zeit und tragen somit zur Lösung des Problems bei.

1.2.3 Exploration vs. Exploitation

Das eigentliche Lernen findet erst dann statt, wenn der Agent in einem Zustand nicht die Aktion mit der bislang besten Belohnung auswählt, sondern willkürlich irgendeine Aktion “ausprobiert”. Dadurch ist gewährleistet, dass er in diesem Zustand eine eventuell günstigere Aktion findet und somit tatsächlich etwas lernt. Auf diese Weise kann er vollkommen autonom Aufgaben in einer unbekanntem Umgebung optimal lösen. Anwenden des Gelernten und Ausprobieren, oder *Exploitation* und *Exploration*, wechseln dabei ständig ab und führen in der Regel zu einer Konfliktsituation.

Dieser Konflikt zwischen Lernen und Anwenden wird mittlerweile seit mehreren Jahrzehnten erforscht und hat 3 wesentlichen Vorgehensweisen hervorgebracht:

“greedy”: Wie der Name bereits verrät, handelt es sich dabei um eine Strategie, die zu jedem Zeitpunkt die Aktion mit der größten erwarteten Belohnung wählt. Der Nachteil dabei ist jedoch, dass der Agent eine eventuell besser vergütete Aktion nicht entdeckt.

“wacky”: Bei diesem Ansatz wird konträr zur “greedy”-Strategie stets eine neue Aktion ausprobiert, so dass dem Agenten keine optimalere Aktion entgeht. Der Nachteil hier ist, dass das Gelernte nicht konsequent in die Tat umgesetzt wird.

ϵ -“greedy”: Diese Methode stellt einen Kompromiss der beiden vorhergehenden Methoden dar: Der Agent agiert überwiegend “greedy”, wobei ϵ für den Anteil steht, zu dem sich der Agent “wacky” verhält ([4]).

In der Praxis hat es sich bewahrheitet, sich zunächst “wacky” zu verhalten, um die Umgebung zu erforschen, mit zunehmender Erfahrung “greedy”, um vom Gelernten zu profitieren, und nur noch gelegentlich Explorieren, um auf Umweltänderungen rechtzeitig angemessen reagieren zu können.

1.2.4 Die Markov Eigenschaft

Beim Reinforcement Learning trifft der Agent seine Entscheidungen entsprechend dem Zustand, in dem er sich aktuell befindet. Darüberhinaus ist das auch alles, was er über seine Umgebung weiß. Eine Zustandsmenge hat nun die *Markov Eigenschaft*, wenn die Belohnung und der Folgezustand einer Aktion nur vom aktuellen Zustand und nicht von vorherigen Zuständen und Aktionen abhängt.

Mathematisch lässt sich die Markov Eigenschaft im diskreten Fall über die folgenden Wahrscheinlichkeitsverteilungen beschreiben:

$$Pr\{s_{t+1} = s', r_{t+1} = r \mid s_t, a_t, r_t, s_{t-1}, a_{t-1}, \dots, r_1, s_0, a_0\} \quad (1.3)$$

$$Pr\{s_{t+1} = s', r_{t+1} = r \mid s_t, a_t\} \quad (1.4)$$

Der erste Fall gibt die Wahrscheinlichkeit an, zum Zeitpunkt $t + 1$ in den Zustand s' bei einer Belohnung von r zu wechseln in Abhängigkeit von allen vorher geschehenen Ereignissen, also $s_t, a_t, r_t, \dots, r_1, s_0, a_0$. Hat auf der anderen Seite der Zustand jedoch die Markov Eigenschaft, so ist die Wahrscheinlichkeit beim nächsten Übergang in s' zu landen nur von dem unmittelbar vorhergehenden Zustand und der zugehörigen Aktion abhängig, d. h. Gleichung 1.3 und 1.4 stimmen überein.

Die Markov Eigenschaft ist für ein Reinforcement Learning System von Bedeutung, weil angenommen wird, dass die Entscheidung, welche Aktion gewählt werden soll, nur auf den Information im aktuellen Zustand und nicht auf denen der Vergangenheit basiert.

1.2.5 Markov Entscheidungsprozesse (MDP)

Als *Markov Entscheidungsprozess* bezeichnet man eine Reinforcement Learning Aufgabe, welche die Markov Eigenschaft erfüllt. Es ist ein grundlegendes mathematisches Modell und dient dem Verständnis von nahezu 90% der modernen Reinforcement Learning Ansätze. Definiert wird ein Markov Entscheidungsprozess – auch **Markov Decision Process** genannt – als ein Quintupel:

$$MDP := \{S, A, T, \gamma, R\} \quad (1.5)$$

bestehend aus den Komponenten

S : Menge von möglichen Zuständen $s \in S$, in der sich der Agent befinden kann.

A : Menge von Aktionen $a \in A(s)$, die ein Agent in einem Zustand ausführen darf.

T : Zustandsübergangswahrscheinlichkeiten $T = \{P_{sa}(s' \mid s \in S, a \in A)\}$, wobei $P_{sa}(s' \mid s, a)$ die bedingte Wahrscheinlichkeit für die Transition von Zustand s nach s' darstellt, wenn Aktion a ausgeführt wird.

γ : Der in Abschnitt 1.2.2 eingeführte Diskontfaktor

R : Bezeichnet die sogenannte Reward- oder Belohnungsfunktion. Sie hat die Aufgabe, einer ausgeführten Aktion $a \in A$ in einem Zustand $s \in S$ einen skalaren Wert $r \in \mathfrak{R}$ zu zuordnen. Eine exakte Definition erfolgt noch in Abschnitt 1.3.2.

Anhand dieser Definition lassen sich nun Reinforcement Learning Systeme beschreiben. Ein deterministisches System kann beispielsweise beschrieben werden, indem man $P_{sa}(s' \mid sa) = 1$ für alle vorhandenen Zustände $s \in S$ setzt.

1.3 Hauptbestandteile

Dieser Abschnitt behandelt die Hauptbestandteile eines Reinforcement Learning Systems und dient in erster Linie einer übersichtlichen Darstellung der benötigten Komponenten: *Strategie*, *Vergütungsfunktion*, *Wertefunktion* und ein optionales *Modell* der Umgebung.

1.3.1 Strategie (Policy π)

Die eigentliche Herausforderung beim Reinforcement Learning ist die Erforschung einer optimalen Strategie in einer unbekanntenen Umgebung in Bezug auf ein vorher festgelegtes Ziel. Im konkreten Fall bedeutet dies, eine geeignete Aktionsfolge, z. B. $(a_1, a_2, a_3, \dots, a_k)$, auf einer Menge von Zuständen S zu finden. Die Strategie oder auch *Policy* π definiert somit eine Abbildung von Zuständen auf Aktionen, die in diesen Zuständen die größte Belohnung erzielen.

1.3.2 Vergütungsfunktion (Reward function)

Mit Hilfe der Vergütungsfunktion – auch *reward function* genannt – wird das übergeordnete Ziel eines Reinforcement Learning Problems festgelegt. Dabei ist die Aufgabe des Agenten, die gesamte Vergütung bzw. Belohnung im Verlauf seiner Interaktion mit der Umwelt zu maximieren. Mathematisch lässt sich die Vergütungsfunktion wie folgt beschreiben:

$$R : S \times A \rightarrow \mathfrak{R} \quad (1.6)$$

Damit wird jedem Zustand-Aktions Paar eine Vergütung in Form eines skalaren Wertes zugeordnet.

Es ist die Aufgabe des Designers eines Reinforcement Learning Systems, eine geschickte Vergütungsfunktion zu wählen, welche schließlich das angestrebte Ziel des Agenten bestimmt. Obwohl viele komplexe Vergütungsfunktionen definiert werden können, lassen sich die meisten nach [5] in drei Klassen unterteilen.

Delayed Reward : Die Vergütungsfunktionen dieser Klasse ordnen jedem Zustand-Aktions Paar 0 zu, außer denen, die zum Endzustand führen. Das Vorzeichen der Vergütung im letzteren Fall legt dabei fest, ob der Endzustand erstrebenswert ist oder vermieden werden sollte. Die Vergütungsfunktion des Stab Balancierungsproblems aus Abschnitt 1.2.2 ist nach genau diesen Kriterien gewählt.

Minimum Time to Goal : Diese Kategorie beinhaltet Vergütungsfunktionen, die den Agenten dazu “zwingen”, Aktionen auszuführen, welche den kürzesten Pfad zum Endzustand darstellen. Erreicht wird dies, indem alle Zustand-Aktions Paare mit

−1 und der Endzustand mit 0 vergütet wird. Da der Agent die Vergütungen zu maximieren versucht, lernt er einen kürzesten Pfad zum Endzustand.

Games : Bislang wurde angenommen, dass der Agent die Vergütungen zu maximieren versucht. Jedoch muss das nicht immer so sein: Der Agent könnte genauso versuchen die Vergütungsfunktion zu minimieren. Insbesondere dann, wenn es darum geht, ein Ziel zu erreichen und dabei so wenig wie möglich Ressourcen zu verschwenden. Im Roboterfussball könnte mit dieser Technik das Ziel festgelegt werden, ein Tor zu schießen, bei möglichst wenig verbrauchter Energie.

Die Idee hier ist nur der Vollständigkeit halber angeführt. Eine genauere Ausführung zu diesem Ansatz findet sich in [6].

1.3.3 Wertefunktion (Value function)

Die Wertefunktion – engl. *value function* – ist eine der entscheidenden Komponenten eines Reinforcement Learning Systems. Sie gibt an, welche Aktionsfolgen zur Problemlösung mehr oder weniger beitragen und beschreibt dadurch die langfristige Attraktivität eines Zustandes. Denn eine Aktion mit der besten Vergütung zu wählen, muss nicht zwangsläufig zur optimalen Strategie führen. Es kann durchaus vorkommen, dass zwischenzeitige Aktionen mit geringerer Vergütung eine bessere Gesamtstrategie liefern. Die Wertefunktion hilft somit, lokale Minima in Kauf zu nehmen, um eine bessere Strategie zu erlernen.

In der Literatur unterscheidet man aus gutem Grund zwischen *Zustands-Wertefunktion*

$$V^\pi(s) = E_\pi\{R_t \mid s_t = s\} = E_\pi\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s\right\} \quad (1.7)$$

und *Aktions-Wertefunktion*

$$Q^\pi(s, a) = E_\pi\{R_t \mid s_t = s, a_t = a\} = E_\pi\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a\right\} \quad (1.8)$$

Der marginale, aber so entscheidende Unterschied ist: $V^\pi(s)$ gibt den erwarteten Rückgabewert (Return) für den Fall zurück, dass man im Zustand s startet und die Strategie π verfolgt. Die Aktions-Wertefunktion $Q^\pi(s, a)$ hingegen liefert den erwarteten Rückgabewert für den Fall, dass man im Zustand s eine Aktion a bei der verfolgten Strategie π wählt. Zusammenfassend bedeutet es, dass die Zustand-Wertefunktion *Zustände* bewertet, die Aktions-Wertefunktion *Zustand-Aktions Paare*.

Um sich die Arbeitsweise der Wertefunktion besser vorstellen zu können, betrachten wir ein Beispiel aus der Blockwelt. Die Felder einer 4×4 Matrix stellen dabei

16 Zustände dar, ein Übergang aus einem Zustand zu einem der 4 Nachbarfelder, eine Aktion, also: links, rechts, oben, unten. Die Vergütungsfunktion ordnet jedem Übergang, der nicht zum linken oberen oder rechten unteren Feld führt, einen Reward von -1 zu, sonst 0 . Damit ist das Ziel eines in dieser Blockwelt befindlichen Agenten festgelegt – den kürzesten Pfad zum linken oberen oder rechten unteren Feld zu finden, was sogleich den beiden Endzuständen entspricht. Bei Anwendung der *Zustand*-Wertefunktion und einer zufälligen Strategie, erhält man nach einiger Zeit die Zustandswerte, die Abbildung 1.3 wiedergibt.

0	-14	-20	-22
-14	-18	-22	-20
-20	-22	-18	-14
-22	-20	-14	0

Abbildung 1.3: Erwartete Zustandswerte bei einer zufälligen Strategie

Mit anderen Worten: Der Agent benötigt z. B. im Durchschnitt 22 Schritte von der linken unteren Ecke bis zu einem Endzustand, falls er jedesmal zufällig eine Richtung wählt. Abbildung 1.4 zeigt hingegen die Zustandswerte einer optimalen Wertefunktion.

0	-1	-2	-3
-1	-2	-3	-2
-2	-3	-2	-1
-3	-2	-1	0

Abbildung 1.4: Zustandswerte einer optimalen Zustands-Wertefunktion

Startend in der linken unteren Ecke, summieren sich die Rewards diesmal nur bis -3 .

Hat man einmal die optimale Wertefunktion gefunden, so ist die optimale Strategie trivial abzuleiten. Angefangen in einem beliebigen Zustand, wählt man einfach immer

die Richtung, welche den unmittelbaren Reward maximiert. Abbildung 1.5 zeigt die resultierende Strategie.

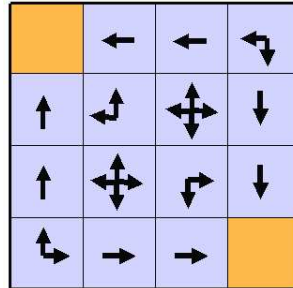


Abbildung 1.5: Optimale Strategie im Blockwelt-Beispiel

1.3.4 Umgebungsmodell (optional)

Wie bereits erwähnt lernt ein Agent eine Abbildung von Zuständen auf Aktionen ausschließlich durch die Interaktion mit seiner Umgebung. Sie wird vom Agenten über Sensoren mit Hilfe symbolischer Informationen (z. B. Markern) wahrgenommen und ermöglicht ein vorausschauendes Handeln, schließt also ein Planen mit ein. Es ist für ein Reinforcement Learning System jedoch nicht zwingend notwendig und kann bei mangelnden Informationen der Umgebung folglich weggelassen werden.

1.4 Lernverfahren

Wie in Abschnitt 1.3.3 erläutert, ist die Berechnung einer guten Strategie von der Wertefunktion abhängig. In diesem Teil der Ausarbeitung sollen einige der wichtigsten Methoden und Algorithmen vorgestellt werden, um eine optimale Wertefunktion zu finden.

1.4.1 Dynamisches Programmieren

Das Konzept der *Dynamischen Programmierung (DP)* ist ein allgemein bekanntes Paradigma in der Mathematik und bezieht sich hier auf eine Sammlung mehrerer Algorithmen zur Berechnung einer optimalen Wertefunktion. Ein großer Nachteil dieser Methode ist, dass sie nur auf Reinforcement Learning Probleme anwendbar ist, bei denen ein perfektes Modell der Umgebung – als Markov Entscheidungsprozess – gegeben ist. Die Aktualisierung der Zustandswerte geschieht mit Hilfe einer *Zustandswertefunktion* und beruht auf dem sogenannten *Bootstrapping Verfahren*: Der Wert

im aktuellen Zustand wird auf Grund der Werte der Folgezustände bis hin zu einem Endzustand berechnet. Man spricht in diesem Zusammenhang auch von *rückwärts propagierten* Rewards.

Wie das Reinforcement Learning, ist DP selbst auch nur eine *Klasse von Algorithmen* zur Berechnung einer optimalen Wertefunktion. Um nur einige von ihnen zu nennen, seien *Policy Evaluation*, *Policy Iteration*, *Value Iteration* erwähnt. Da diese Algorithmen generell jedoch im Roboter Umfeld auf Grund der hohen Speicher- und Berechnungskosten bislang keinen Einsatz finden, sei an dieser Stelle zur weiteren Vertiefung auf [2] verwiesen.

1.4.2 Monte Carlo Methoden

Ebenso wie das Dynamische Programmieren, sind *Monte Carlo Methoden (MC)* universelle Verfahren, die basierend auf stochastischen Modellen die Lösung eines Problems approximieren, da eine Lösung mit exakter Mathematik nicht in angemessener Zeit berechnet werden kann oder erst gar nicht vorhanden ist.

Bei der Berechnung einer Wertefunktion, liefern bisherige Ergebnisse eine gute Approximation der optimalen Strategie. Die wichtigsten Merkmale von MC sind: Keine Verwendung von Bootstrapping (s. 1.4.1), Aktualisierung der Werte erst nach Erreichen des Endzustandes, Notwendigkeit von Erfahrungswerten aus Beispielerisoden. Damit liegen die Vorteile zu DP auf der Hand. MC benötigt kein vollständiges Umgebungsmodell und lernt dadurch *direkt* aus der Interaktion mit der Umgebung. Nur die tatsächlich besuchten Zustände fließen in die Berechnung ein. Nachteilig wirkt sich aus, dass es keine Garantie für ein globales Optimum der evaluierten Strategie gibt, da bei MC nicht notwendigerweise alle Zustände besucht werden müssen. Eine weitere Einschränkung ist, dass MC nur für episodische Tasks geeignet ist.

Im nächsten Abschnitt wird eine Methode vorgestellt, die den Nachteil von MC und DP nicht hat.

1.4.3 Temporal Difference Methoden

Eine sehr gute Beschreibung der *Temporal Difference Methoden (TD)* findet sich in [7] wieder:

“In einer komplizierten Welt werden die Konsequenzen einer Handlung meistens nicht unmittelbar sichtbar. Der Agent nimmt sie oft erst viel später wahr. Das Problem besteht nun darin, festzustellen, welche Aktion für den positiven oder negativen Reward verantwortlich gewesen ist. TD-Methoden bezeichnen Verfahren, mit denen dieser zeitlich verzögerte

Reward so verarbeitet wird, dass der Agent das nächste Mal schon etwas früher weiß, welche Konsequenzen die betreffende Handlung haben wird.”

TD-Methoden haben sich in der Praxis bewährt und einen festen Platz im Reinforcement Learning Umfeld eingenommen. Sie kombinieren die Verfahren von DP und MC und nutzen somit die Vorteile beider Verfahren aus. Im Folgenden werden drei der wichtigsten Algorithmen, die auf dem Prinzip der zeitlichen Verzögerung basieren, näher vorgestellt.

TD-Prediction

Dieser Ansatz verwendet die Zustands-Wertefunktion und beschreibt einen Algorithmus zur Aktualisierung der Zustandswerte. Der entscheidende Aktualisierungsschritt wird in Zeile 6 vorgenommen.

- 0: Initialisiere $V(s)$ beliebig
- 1: **Repeat** (für jede Episode):
- 2: Initialisiere Zustand s
- 3: **Repeat** (für jeden Schritt der Episode):
- 4: Wähle Aktion a aus s anhand der zu evaluierenden Strategie
- 5: Führe Aktion a aus, beobachte Reward r und Folgezustand s'
- 6: $V(s) \leftarrow V(s) + \alpha[r + \gamma V(s') - V(s)]$
- 7: $s \leftarrow s'$
- 8: **until** $s = \text{Endzustand}$

γ bezeichnet den in Abschnitt 1.2.2 vorgestellten Diskontfaktor und α die hier neu eingeführte Lernrate. Genau wie γ , nimmt α Werte im Bereich von 0 bis 1 an. Die Lernrate sollte im Laufe der Zeit runtergesetzt werden, damit kurzzeitige Schwankungen nicht negativ auf die Strategie einwirken. Sie sollte aber immer einen Wert größer 0 annehmen, da der Agent sonst auf zukünftige Änderungen in der Umwelt nicht mehr reagieren kann. Für $\alpha = 0$ ergibt Zeile 6 $V(s) \leftarrow V(s)$. Das bedeutet, dass der Agent die Zustandswerte nicht mehr ändert und somit nichts mehr lernt. Im Kontrast dazu ergibt sich für $\alpha = 1$ $V(s) \leftarrow r + \gamma V(s')$, was den erhaltenen Reward und diskontierten Wert des Folgezustands zu 100% in den aktuellen Zustandswert einfließen lässt.

Sarsa

Im Gegensatz zu TD-Prediction arbeitet *Sarsa* mit Hilfe der Aktion-Zustands-Wertefunktion. Zur Aktualisierung wird daher folgende Gleichung verwendet:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - A(s_t, a_t)] \quad (1.9)$$

Die Formel beschreibt, wie zum Zeitpunkt t der Wert des *Zustand-Aktionspaares* – auch Q-Wert genannt – aktualisiert wird, wenn der Agent die Aktion a_t im Zustand s_t gewählt hat.

Der Algorithmus hierzu sieht wie folgt aus:

- 0: Initialisiere $Q(s, a)$ beliebig
- 1: **Repeat** (für jede Episode):
- 2: Initialisiere Zustand s
- 3: Wähle Aktion a aus s unter Benutzung der Strategie aus Q
- 4: **Repeat** (für jeden Schritt der Episode):
- 5: Führe Aktion a aus, beobachte Reward r und Folgezustand s'
- 6: Wähle Aktion a' aus s' unter Benutzung der Strategie aus Q
- 7: $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)]$
- 8: $s \leftarrow s'$
- 9: $a \leftarrow a'$
- 10: **until** $s =$ Endzustand

Q-Learning

Einer der mächtigsten Algorithmen im Bereich des Reinforcement Learnings ist der *Off-Policy TD Control* Algorithmus *Q-Learning*, der bereits 1989 von Watkins [1] entwickelt wurde und die Aktualisierung der Q-Werte anhand folgender Gleichung durchführt:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_i Q(s_{t+1}, i) - A(s_t, a_t)] \quad (1.10)$$

Der markante Unterschied zur Sarsa-Methode liegt darin, dass hier der Q-Wert des *optimalen* Zustand-Aktionspaares verwendet wird und nicht der, welcher aus der Strategie heraus gewählt worden wäre. Die Q-Learning Methode evaluiert somit eine andere Strategie als sie in Wirklichkeit benutzt – arbeitet also *off-policy*. Sarsa hingegen arbeitet *on-policy*.

Der konkrete Algorithmus nach [2]:

- 0: Initialisiere $Q(s, a)$ beliebig
- 1: **Repeat** (für jede Episode):
- 2: Initialisiere Zustand s
- 3: **Repeat** (für jeden Schritt der Episode):
- 4: Wähle Aktion a aus s unter Benutzung der Strategie aus Q
- 5: Führe Aktion a aus, beobachte Reward r und Folgezustand s'

6: $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_i Q(s', i) - Q(s, a)]$
 7: $s \leftarrow s'$
 8: **until** $s = \text{Endzustand}$

1.5 Erfolgsgeschichten

Um die Leistungsstärke von Reinforcement Learning unter Beweis zu stellen, werden in diesem Abschnitt einige Erfolgsgeschichten aufgelistet und kurz vorgestellt.

1.5.1 Backgammon

Einer der ersten Durchbrüche auf dem Gebiet des Reinforcement Learning gelang 1992 am Backgammon Spiel. Abbildung 1.6 zeigt eine Spielsituation aus Sicht des weißen Spielers im Anfangsstadium einer Partie. Gerry Tesauro's *TD-Gammon* Programm hat Backgammon mit nur einwenig Hintergrundwissen so gut lernen können, dass es gegen die weltbesten Spieler erfolgreich angetreten ist.

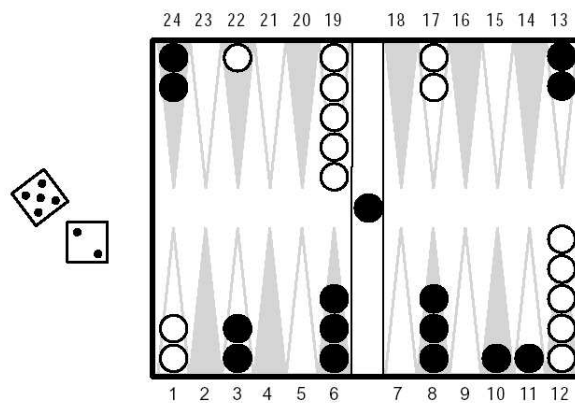


Abbildung 1.6: Eine Spielsituation beim Backgammon (aus [2])

Nach nur 300.000 Testspielen ist TD-Gammon bereits besser als jedes andere Computerprogramm. Die Implementierung basiert dabei auf einem TD-Algorithmus, kombiniert mit einem Künstlichen Neuronales Netz. Der Zustandsraum umfasst 10^{20} Zustände und nimmt daher recht viel Speicherplatz in Anspruch. Weitere Einzelheiten zum Algorithmus finden sich in [2].

1.5.2 Karlsruher Brainstormer

Die *Karlsruher Brainstormer* sind eine Mannschaft in der Robocup *Simulation-League* und wenden Reinforcement Learning auf einige Basisverhalten ihrer Agenten an. Sie formulieren das Lernen eines Verhaltens als dynamisches Optimierungsproblem und interpretieren die Wertefunktion als Kostenfunktion $J^\pi(s)$. Jeder Übergang von einem Zustand zum anderen wird mit bestimmten Kosten k belegt. Das Ziel des Agenten ist es nun, die Kostenfunktion zu minimieren – also einen kürzesten Weg zu einem Endzustand zu finden. Da der Zustandsraum prinzipiell stetig ist, benutzen sie desweiteren Künstlich Neuronale Netze (KNN), um die Kostenfunktion zu approximieren. Da die Einführung von KNN den Rahmen dieser Arbeit sprengen würde, sei an dieser Stelle für eine ausführliche Darstellung auf [8] verwiesen.

Laut [9] stieg die Erfolgsquote eines Reinforcement Learning basierten Angriff von 35% auf 85% gegen einen Verteidiger, bzw. von 10% auf 55% bei zwei Verteidigern, gegenüber dem komplizierten statisch programmierten Angriff.

1.5.3 Fahrstuhlsteuerung

Ein allgemein bekanntes Problem ist die Steuerung von Fahrstühlen. Wenn gleichzeitig mehrere Anfragen zu einigen Fahrstühlen in einem Gebäudekomplex kommen, so gibt es zur Zeit noch keine optimale Strategie, um die Wartezeiten so minimal wie möglich zu halten.

1995 gelang es Crites und Barto ([2]) jedoch mit Hilfe von Reinforcement Learning Verfahren die bis heute beste Strategie für die Situation von 4 Fahrstühlen in einem 10 stockigem Hochhaus zu finden. Zum Einsatz kommt dabei ein modifizierter Q-Learning Algorithmus, der bei grober Diskretisierung auf einer Zustandsmenge von ca. 10^{22} arbeitet. Das Ergebnis ist beeindruckend, obwohl die optimale Strategie noch immer nicht bekannt ist.

1.6 Fazit & Ausblick

In dieser Seminararbeit wurde eine leistungsstarke Klasse im Bereich des Maschinellen Lernens vorgestellt. Trotz der bisherigen Erfolge ist man jedoch noch lange nicht am Ende der Forschung. Die wirklichen Probleme in Real-World Anwendungen sind viel kompliziert und lassen sich oft nicht mit einem einfachen MDP modellieren. Nicht selten hat man es mit Nichtdeterministischen Übergängen und kontinuierlichen und/oder unendlichen Zustandsräumen zu tun. Hier ist noch großer Bedarf an Beschleunigung, Robustheit und Effizienz. Zwar gibt es schon Ansätze wie *Multi-Agent Markov Decision Processes (MMDP)* ([10]), welche kooperatives Reinforcement Learning in Multiagenten-Systemen ermöglichen, sowie Beschleunigungsmethoden mit Hil-

fe von *Lyapunov*-Funktionen ([11]), doch sind sie noch nicht genau erprobt. Zudem ist die Zustandsmenge eines Agenten in der *Midsize-League* enorm groß, was einem jetzigen Einsatz von Reinforcement Learning Methoden noch im Wege steht.

Abschließend bleibt zu sagen, dass das Reinforcement Learning die KI sehr bereichert hat und noch viel Potential für die Zukunft in sich birgt.

Literaturverzeichnis

- [1] C. J. C. H. Watkins. *Learning from Delayed Rewards*. PhD thesis, Kings's College, Cambridge, UK, 1989.
- [2] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, 1998.
- [3] Stuart Russel and Peter Norvig. *Artificial Intelligence, A Modern Approach*, chapter 20, pages 598–624. Prentice-Hall, Englewood Cliffs, New Jersey, 1995.
- [4] C. J. C. H. Watkins and P. Dayan. *Q-Learning*, chapter 8(3), pages 279–292. Kluwer Academic Publishers, 1992.
- [5] M. Harmon. Reinforcement learning: a tutorial. view or download on <http://www.nada.kth.se/kurser/kth/2d1432/2003/rltutorial.pdf>, december 1996., 1996.
- [6] Leslie Pack Kaelbling and Michael L. Littman. Reinforcement learning: A survey. view or download on <http://www-2.cs.cmu.edu/afs/cs/project/jair/pub/volume4/kaelbling96a-html/rl-survey.html>, 1996.
- [7] Willi Richert. *Reinforcement Learning*. PhD thesis, Universität Paderborn, 2002.
- [8] Arthur Merke. *Reinforcement Lernen in Multiagentensystemen*. PhD thesis, Universität Karlsruhe (TH), 1999.
- [9] Dieter Kerkfeld. Agenten und robotfussball, http://wwwmath.uni-muenster.de/u/lammers/edu/ss03/robotfussball/ausarbeitungen/reinforcement-learning/ausarbeitung_dieter_kerkfeld.pdf, 2003.
- [10] C. Boutilier. Sequential optimality and coordination in multiagent systems. In *Proceedings of the Sixteenth International Joint Conferences on Artificial Intelligence*, 1999.
- [11] Maxim Neumann. Beschleunigtes reinforcement learning unter verwendung von potentialfunktionen, http://www.cs.tu-berlin.de/mki/lehre/muster/rl_seminar.pdf, 2003.