Technische Universität München
Institut für Informatik

Informatik I — Angewandte Softwaretechnik

Reinforcment Learning

Hauptseminar Machine Learning

TEX

Bearbeiter: Alexander, Camek

Themensteller: Prof. G. Klinker, Prof. Dr. S. Kramer

Betreuer: Prof. Dr. S. Kramer

Wintersemester 2003

Inhaltsverzeichnis

1 Einführung in Reinforcment Learning 2

2 passives Lernen in bekannter Umgebung 4

3 passives Lernen in unbekannter Umgebung 9

4 aktives Lernen in unbekannter Umgebung 10

5 Exploration 12

6 Q–Values und Q–Learning 14

7 Generalisierung im Reinforcment Learning 16

8 Beispiel: Backgammon 18

9 Literatur 20

1 Einführung in Reinforcment Learning

Der Bereich Machine Learning konzentriert sich darauf Informationen zu sammeln, zu strukturieren und mit Hilfe dieser sich Wissen anzueignen.

Um dieses Ziel zu erreichen, gibt es ein Vielzahl von algorithmischen Methoden. Eine davon ist das Reinforcment Learning — auch reward learning genannt.

Reinforcment Learning möchte aus Erfahrungen grundlegendes Wissen abstrahieren und sich mit Hilfe dieses Wissens besser in seinem Anwendungsgebiet verhalten. Der Unterschied zu anderen Lernmethoden ist das Fehlen eines Lehreres, der dem Lerner Wissen vermittlet. So zu sagen muß der Lerner sein Wissen sich selbst erarbeiten und seine eigenen Schlüsse aus den Fehlern, bzw. aus den Belohnungen ziehen. Das Hauptproblem einer solchen Methode befindet sich im Bereich Erfahrungen sammeln, denn der Lerner weiß in Situationen, in den er etwas falsch gemacht hat, nicht was er falsch gemacht hat. Gleichzeitig muß der Lerner seine Umgebung entweder kennen oder in Erfahrung bringen, was zu weiteren falschen Schlußfolgerungen führen könnte. Denn eine Umgebung kann für ihn unzugänglich, also er muß die Wahrnehmung auf innere Zustände abbilden, oder zugänglich sein. Was natürlich zu Problemen im Design des Lerner führen kann! Im ersteren Fall muß der Agent sein Wissen über die Umgebung selber erarbeiten, also ein Modell lernen, und dadurch zusätzliches Wissen sich aneignen. Auf der anderen Seite kann ihm natürlich das Wissen über seine Umgbung und Ergebnisse seiner Aktionen vorgegeben sein.

Belohnungen, der Grund warum der Lerner Erfahrungen sammelt, können natürlich in jedem Schritt, den der Lerner ausführt, vergeben werden oder nur am Ende, wenn das Ziel erreicht ist! Geleichzeitig ist zu überlegen, wie Belohnugen in die Umgebung zu integrieren sind. Sind also die Belohnungen Teil der Aufgabenstellung (z.B. Punkte bei Tischtennis) oder dienen sie als Hilfestellungen (z.B. im Schach:"guter Zug").

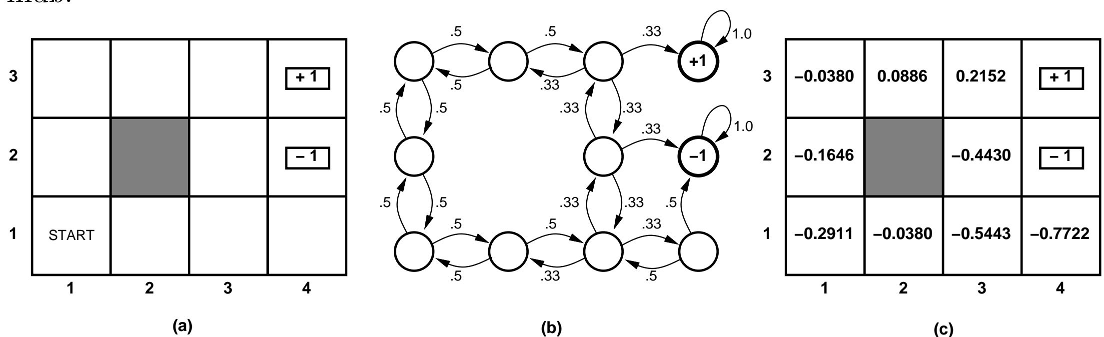
Sind diese Designprobleme gelöst, so ist zu entscheiden, ob der Lerner aktiv oder passiv sein soll. Denn ein passiver Lerner wird nur seine Welt beobachten und dabei seine Aufgabe lernen. Auf der anderen Seite soll der Lerner aktiv sein, so muß natürlich sicher gestellt sein, daß dieser seien Welt erkundet und dabei auch auf die ausgeführten Lernschritte Rücksicht genommen werden.

Dies läßt zwei Grunddesigns zu.

Beim ersten lernt der Agent sein Wissen statisch und benutzt dieses Wissen, um den bestmöglichsten Profit zu erreichen.

Beim anderen lernt der Agent Aktionsmuster aufgrund seiner Reaktionen innerhalb seiner Aufgabe, so daß diese Variante kein Modell der Umgebung zur Verfügung gestellt werden muß.

Als Beispiel diene eine 4*3 Gridumgebung, in der der Agent sich zurecht finden muß.



Alle folgenden Diagramme haben diese Umgebung zur Auswertung als Grundlage. Der linke Teil stellt die verwendete 4*3 Umgebung mit den beiden Endzuständen "+1" und "−1" dar. In der Mitte erkennt man ohne weiteres das Markovmodell der Umgebung mit den übergangswahrscheinlichkeiten von einem Zustand i in den nächsten Zustand j. Rechts hingegen sind die wirklichen Nutzenwerte der Umgebung für spätere vergleiche bei Diagrammen gegeben.

Gleichzeitig steigert sich der Lerner bei uns von einem passiven Lerner in bekannter Umgebung bis zu einem aktiven Lerner in unbekannter Umgebung. Auerdem zeige ich wie die zwei Grunddesigns sich auf den Agenten auswirkt und wie man diese Varianten algorithmisch darstellt.

2 passives Lernen in bekannter Umgebung

Der Lerner in seinem passiv Zustand beobachtet also die Welt und lernt somit neues Wissen. Gleichzeitig ist ihm in dieser Variante auch noch ein Weltenmodell gegeben, so daß er nur noch die Nutzen einzelner Zustände mittels Testdaten ermittlen muß. Der Agent besitzt ein Modell M_{ij}, das repräsentiert wird durch Wahrscheinlichkeiten für den Zustandsübergang von i nach j. Der Nutzen einer Trainingssequenz ist die Summe der zu jedem Zustand angesammelten Belohungen. Stellen wir uns vor, der Agent erhält von uns zum Beginn ein paar Testreihen. Diese werden vom Agenten durchlaufen und dabei wird er seine erhaltenen Belohnungen und die Nutzenwerte einzelner Zustände anpassen. So ist ein erwarteter Nutzen eines Zustandes gleich der erwarteten Summe der Belohnungen von diesem Zustand bis zum Endzustand. Diese Summe von Belohnungen wird auch als der so genannte **reward–to–go** eines Zustandes bezeichnet.

Folgender Algorithmus stellt einen passive Reniforcment Learning Agent dar.

```
function PASSIVE-RL-AGENT(e) returns an action
  static: U, a table of utility estimates
          N, a table of frequencies for states
          M, a table of transition probabilities from state to state
          percepts, a percept sequence (initially empty)

  add e to percepts
  increment N[STATE[e]]
  U ← UPDATE(U, e, percepts, M, N)
  if TERMINAL?[e] then percepts ← the empty sequence
  return the action Observe
```

In diesem Algorithmus werden Tabellen für die verschiedenen Werte zur Speicherung genutzt. In der Realität können auch andere Datentypen verwendet werden. In der Tabelle U steht also der Nutzenwert für den jeweiligen Zustand. N stellt die Anzahl dar, wie oft ein Zustand besucht wurde. Und M ist die Tabelle, die die übergangswahrscheinlichkeiten enthält. Der Algorithmus bekommt eine Wahrnehmung e übergeben und mu eine Aktion zurückliefern (hier: Beobachtung).

Das wichtigste bei Reinforcment Learning ist der Update–Algorithmus, der die Nutzenwerte der Trainingsreihe aktualisiert.
Wie sollte am besten ein solcher Update–Algorithmus aussehen?

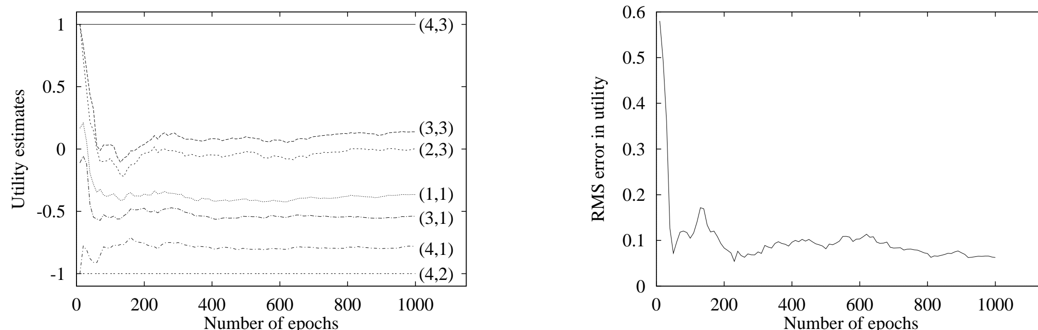
Ich möchte hier drei verschieden Ansätze darstellen. Der erste Ansatz ist ziem-

lich naiv. Er basiert auf der Idee, daß es möglich wäre aus dem beobachteten reward–to–go direkt auf den im Moment erwarteten reward–to–go zu schließen. Dies würde am Ende eine Ermittelung des beobachteten reward–to–go der ganzen Trainingsreihe und des geschätzten Nutzens ermöglichen. Der Ansatz trägt den Namen **LMS** (least mean square), dieser erzeugt einen Nutzenschätzer mit minimalen mittleren quadratischen Fehler im Bezug auf die beobachteten Daten. Somit reduziert dieser Ansatz das ganze Lernen auf ein inuktives Lernverfahren.

Der Algorithmus für dieses Verfahren ist wie folgt:

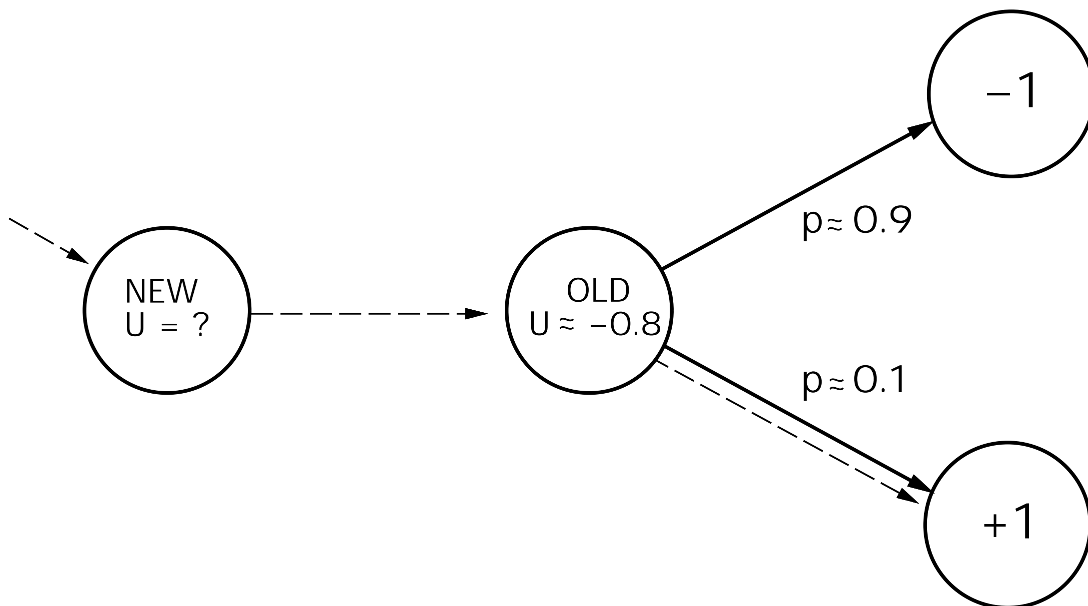
```
function LMS-UPDATE(U, e, percepts, M, N) returns an updated U

  if TERMINAL?[e] then reward-to-go ← 0
  for each e_i in percepts (starting at end) do
      reward-to-go ← reward-to-go + REWARD[e_i]
      U[STATE[e_i]] ← RUNNING-AVERAGE(U[STATE[e_i]], reward-to-go, N[STATE[e_i]])
  end
```

Die Nutzenberechnung erfolgt mittels gleitendem Durchschnitt über die Werte der bisherigen Nutzen, der Anzahl wie oft der Zustand besucht wurde und dem ermittelten reward–to–go.



Das linke Diagramm stellt einen Testlauf in der Gridworld dar. Man sieht, da nach tausend Durchläufen der Algorithmus auf die wahren Nutzenwerte kommt. Hingegen auf der rechten Seite wird der mittlere Fehler dargestellt.

Ein weiterer Ansatz wäre das **ADP**–Verfahren (adaptive dynamic programming). Dieses Verfahren nutzt das Wissen über die Umgebung in der sich der Agent aufhält. Wird in der Gridworldumgebung ein neuer Zustand (hier: "NEW") hinzugefügt. Der Agnet gehe nun von "NEW" über "OLD" zu dem Endzustand mit dem Wert "+1", dann hätte der "NEW"–Zustand beim LMS–Verfahren einen Wert von +1. Dies ist aber zu optimistisch, da er eigentlich einen Wert in der Nähe von "OLD" besitzen müsste!



Deshalb benutzt das ADP–Verfahren sein Wissen über die Umgebung. Sei dieses nun in der Tabelle M_{ij} gespeichert. So reduziert sich das ganze Problem zu einem wohldefinierten sequentiellen Entscheidungsproblem, so bald alle Belohnungen jedes Zustandes und alle Zustände bekannt sind. Die Nutzenberechnung eines Zustandes i geschieht beim ADP mittels folgender Gleichung, die auch als Update–Funktion verwendet werden kann.

$$U(i) = R(i) + \sum_j M_{ij} U(j)$$

wobei R(i) für die Belohnung im Zustand i steht und j ein Nachfolgezustand von i ist.

Die letzte Variante ist die **TD**–Variante (temporal difference learning). Die Idee dieses Ansatzes ist im Grunde die gleiche wie beim ADP–Verfahren, somit wird die gleiche Update–Funktion verwendet. Der einzige Unterschied zwischen diesen beiden Ansätzen besteht darin, daß der TD–Ansatz nicht auf dem ganzen Modellraum arbeitet. Das Verfahren verwendet die beobachteten Übergänge um

die Werte der beobachteten Zustände anzupassen. Bei einer großen Differenz von Nachfolger– und Vorgängernutzen verändert der Ansatz den Vorgänger so, daß dieser zu seinem Nachfolger passt.

Ein kleines Beispiel zur Veranschaulichung:
Der Agent wandert von einem Zustand i zu einem Zustand j. Dabei wurde der Nutzen für die jeweiligen Zustände ermittelt: U(i) = -0.5 und U(j) = +0.5. Jetzt wird mittels TD–Ansatz der Wert von U(i) so angepasst, daß dieser besser zu dem Wert von U(j), seinem Nachfolger, paßt.

Deshalb wird für den TD–Ansatz folgende Lernregel verwedet:

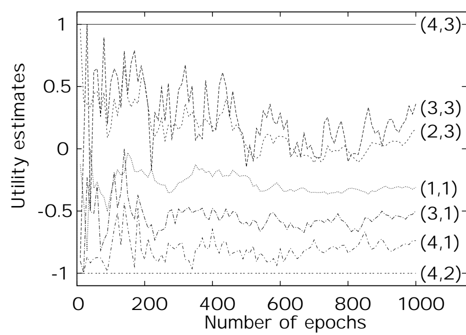
$$U(i) \leftarrow U(i) + \alpha\Big(\mathrm{R(i)} + \mathrm{U(j)} - \mathrm{U(i)}\Big)$$

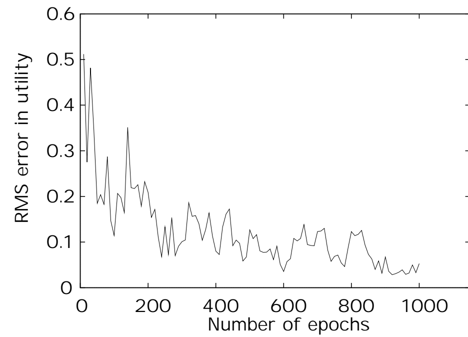
mit α als Lernrate und U(j) – U(i) als Fehlerminimierung.
Am Ende des Trainingssatzes stimmt das Ergebniss mit der Formel des ADP–Ansatzes überein.

Der Algorithmus für die Update–Funktion des TD–Ansatzes sieht folgendermassen aus:

```
function TD-UPDATE(U, e, percepts, M, N) returns the utility table U

  if TERMINAL?[e] then
      U[STATE[e]] ← RUNNING-AVERAGE(U[STATE[e]], REWARD[e], N[STATE[e]])
  else if percepts contains more than one element then
      e' ← the penultimate element of percepts
      i, j ← STATE[e'], STATE[e]
      U[i] ← U[i] + α(N[i])(REWARD[e'] + U[j] - U[i])
```

Das Spezielle in diesem Fall ist der else–Fall. Denn dort wird die neue Update–Funktion für den TD–Ansatz verwendet.





Das linke Diagramm stellt wieder den Verlauf dar, den der Agent braucht um auf die wahren Nutzenwerte zu gelangen. Dieses Mal sind die Schwankungen während der Iteration ziemlich stark. Auch der Fehler, im rechten Diagramm dargestellt, wird nicht so schnell wie beim LMS–Verfahren minimiert und unterliegt auch starken Schwankungen.

3 passives Lernen in unbekannter Umgebung

In dem vorherigen Kapitel besaß der Agent Wissen über seine Umgebung. Dies soll nun angepaßt werden, damit der Lerner sich auch in unbekannten Umgebungen zurecht findet. Dabei wird ihm ein geschätztes Modell am Anfang übergeben, das während seiner Bewegung angepaßt wird. In diesem Fall ist der ADP–Ansatz besser als der TD, da bei der ADP–Variante auf dem ganzen Modell gearbeitet wird. Mittels geringfügiger Änderungen kann der TD–Ansatz wieder verwendet werden. Da der TD nur auf lokalen Änderungen arbeitet, müsste man den Algorithmus so anpassen, daß dieser Änderungen über eine große Menge von Zustanden änderungen durchführt. Dies kann vernachlässigt werden, da die Häufigkeit jeden Nachfolger zu besuchen näherungsweise proportianal zu dessen Übergans-wahrscheinlichkeit ist. Außerdem muß ein Umgebungsmodell zum Generieren von Pseudo–Experimenten benutzt werden, da der Lerner im TD–Ansatz nur lokal arbeite. So wird für jeden beobachteten Übergang eine große Anzahl von imaginären Übergängen kreiert. Dadurch wird dem TD die Möglichket geschaffen sein Modell anzupassen und sich dem ADP–Ansatz anzunähren. Dies wird mit einem höheren Berechnungsaufwand erkauft.

Ähnlich ließe sich der ADP–Ansatz ändern, damit dieser eine höhere Effizienz beim Beginn der Lernphase hat. Eine Möglichkeit wäre den ADP zu approximieren durch Begrenzung der Anpassungsschritte. Eine andere Möglichkeit wäre die Verwendung von Heuristicen wie den **prioritized–sweeping**. Dieses Verfahren macht Änderungen in den Zuständen nötig, so daß diese nun ihren Vorgänger kennen. Dann werden die Zustände mit einer Priorität versehen, da der wahrscheinlichste Nachfolger die größte Anpassung über sich ergehen lassen muß. Nun werden die Zustände mit der höchsten Priorität aktualisiert. Zuerst aber merket sich der Lerner den aktuellen Nutzenwert ($U_{old} = U(i)$). Danach wird der Statuswert berechnet mittels der Formel für den ADP. Als nächstes wird die Zustandspriorität auf 0 gesetzt und ein Deltawert aus dem neuen Wert und dem gemerkten berechnet. Dieser Deltawert wird dann benutzt, um die Priorität des Vorgängerzustands zu ändern.

Diese Änderungen bringen den Vorteil, daß der Agent am Anfang keine exacte Berechnung der tatsächlichen Nutzenfunktion und somit keine aufwändigen Iterationen durchführen muß. Gleichzeitig wird am Anfang die Anzahl der Iterationen verringert und somit die Lerngeschwindigkeit erhöht.

4 aktives Lernen in unbekannter Umgebung

Der Lerner hat bis jetzt keine eigenständigen Aktionen ausgeführt, um seine Umgebung zu erkunden. Denn bisher hat der Agent nur mittels Testreihen gelernt. Die Interaktion mit der Umgebung soll ab sofort dem Lerner ermöglicht werden. Der Aktivelerner muß nun darüber nachdenken, welche Aktionen er ausführt, was die Ergebnisse seiner Aktionen sind und wie diese sich auf seine Belohnung auswirken.

Deshalb müssen einige Änderungen zur Wandelung von passiv in aktiv am Agenten durchgeführt werden. Insbesondere ist die Repräsentation des Modells M_{ij} in die neue Version M_{ij}^a überzuführen, da der Agent vom Zustand i in den Zustand j nur mittels Aktion a kommt. Gleichzeitig muß auch berücksichtigt werden, daß der Agent nun eine Wahlmöglichkeit für die durchgeführte Aktion hat. Zusätzlich möchte auch ein rationaler Agent immer seinen Nutzen maximieren.

Deshalb muß die Update–Funktion wie folgt geändert werden:

$$U(i) = R(i) + \max_a \sum_j M_{ij}^a U(j)$$

Neben der Änderung der Funktion muß nun auch sichergestellt werden, daß der Agent in jedem Schritt eine Aktion auswählen muß.

Nachdem die Änderungen vollzogen wurden, sieht der Agent so aus:

```
function ACTIVE-ADP-AGENT(e) returns an action
  static: U, a table of utility estimates
          M, a table of transition probabilities from state to state for each action
          R, a table of rewards for states
          percepts, a percept sequence (initially empty)
          last-action, the action just executed

  add e to percepts
  R[STATE[e]] ← REWARD[e]
  M ← UPDATE-ACTIVE-MODEL(M, percepts, last-action)
  U ← VALUE-ITERATION(U, M, R)
  if TERMINAL?[e] then
     percepts ← the empty sequence
  last-action ← PERFORMANCE-ELEMENT(e)
  return last-action
```

Es fällt auf, da nun die Tabelle mit den Anzahlen der besuchten Zustände herausgelassen wurde, da man diese nicht mehr braucht. Gleichzeitig muß der Agent

seine letzte ausgeführte Aktion merken! Dies tut er nun mittels der Funktion PERFORMANCE–ELEMENT(e). Auerdem wird neben den Nutzenwerten auch das Modell M beim Durchlauf aktualisiert.

Aber die Änderungen schlagen sich nicht nur im Agenten nieder, sondern auch in unseren Algorithmen! Deshalb müssen wir auch hier einige Änderungen durchführen.

Bei ADP wird die Änderung des M_{ij}–Modells zum M_{ij}^{a}–Modell mittels Hinzufügung einer dritten Dimension erreicht. Natürlich beschränkt sich dies nicht auf den Agent selbst, sondern auch die Update–Funktion des ADPs muß angepasst werden. Das Modell beim TD ist dasgleiche wie beim ADP, aber die Update–Funktion muß nicht geändert werden. Da die Update–Funktion der TD–Variante keine Berechnung auf dem Modell ausführt.

5 Exploration

Was bis jetzt unbeantwortet geblieben ist, wie der Agent herausfindet welche Aktionen er ausführen soll. Es ist klar, daß der Lerner die Aktion ausführen soll, die für ihn in der aktuellen Umgebung den höchsten Nutzen hat. Doch dabei wird das Lernen vergessen. Natürlich will der Lerner für den aktuellen Trainingssatz die maximale Belohnung erhalten, aber man muß auch dem Agent ermöglichen zu lernen. Denn damit wird sicher gestellelt für zukünftige Trainingssätze die maximale Belohnung zu erhalten. Somit steht der Agent in einem Zwiespalt zwischen sofortigem Gewinn und der Aussicht auf zukünftigen maximalen Gewinn.
Am besten wäre es ein Mittelweg zwischen dem Erhalt für den sofortigen maximalen Gewinn und dem Lernen zur Erhaltung zukünftiger maximaler Gewinne zu finden. Da die reine Gewinnmaximierung irgendwann nicht mehr funktionieren könnte und reines Lernen kein reales Ergebniss erzielt.

Bei der Benutzung einer Greedy–Strategie gibt es verschieden Varianten. Eine davon ist die Wahl von Aktionen mit nicht starkem negativen Einfluß. Dem gegenüber gestellt gibt es auch bei den so gennanten Wacky–Strategien mehrere Varianten. Ein Beispiel dieser Variante wäre die Boltzmann–Exploration bei der mittels folgender Wahrscheinlichkeitsberechnung die Auswahl der Aktionen durchgeführt wird:

$$P(a) = \frac{e^{ER(a)/T}}{\sum_{\acute{a} \in A} e^{ER(\acute{a})/T}}$$

wobei ER(a) die erwartete Belohnung bei Durchführung ist.

Der Agent hingegen soll beides machen. Deshalb wird die Nutzenfunktion geändert, so daß unerkundete Zustände einen hohen Nutzen bekommen. Dies geschieht mit Hilfe eines optimistischen Nutzenschätzer ($U^+(i)$), der in die Update–Funktion einfließt. Die neue Update–Funktion für den ADP sieht dann so aus:

$$U^+(i) \leftarrow R(i) + \max_a f\Big(\sum_j M_{ij}^a U^+(j), N(a,i)\Big)$$

wobei N(a,i) die Anzahl wie oft die Aktion a im Zustand i ausgeführt wurde und f(u,n) die Explorationsfunktion ist. Für die Explorationsfunktion gibt es sehr viele Möglichkeiten, eine davon ist

$$f(u,n) = \begin{cases} R^+ & , n < N_e \\ U & , \text{sonst} \end{cases}$$

hier entspricht R^+ einem optimistischen Schätzer für den bestmöglichen erreichbaren Gewinn und N_e für einen festen Wert wie oft jedes Aktions–Zustandspaar

ausprobiert wird.

U^+ steht in der Update–Funktion auch auf der rechten Seite, da am Anfang sonst der Lerner abgeneigt wäre weiter draußen in seiner Umgebung zu erkunden. So nimmt der Erkundungswert vom Rand der unerforschten Gebiete ab und dadurch werden Aktionen in unerforschten Gebieten stärker honoriert.

6 Q–Values und Q–Learning

Ein anderer Ansatz ist das Verfahren des Q–Learning und das Lernen von Q–Values. Diese Varianten verzichten auf ein Modell. Das Hauptaugenmerk dieser Methoden ist die Zuornung eines erwarteten Nutzen einer Durchführung zu einer Aktion in einem bestimmten Zustand. Was also uns nun interessiert ist die Bewertung der einzelnen Aktionen für die Zustandsübergänge. Dies wird ausgedrückt durch sogennante Q–Values Q(a,i). Q–Values sind direkt mit dem Nutzenwert verbunden mittels

$$U(i) = \max_a Q(a, i)$$

Neben dem Verzicht auf ein Modell profitiert diese Variante von der Möglichkeit durch Belohnungsfeedback zulernen. Wenn also Q–Values korrekt sind, dann muß folgendes gelten:

$$Q(a, i) = R(i) + \sum_j M_{ij}^a \max_{\acute{a}} Q(\acute{a}, i)$$

Dies kann aber auch als Update–Funktion bei der ADP–Variante verwendet werden.

Bei der TD–Variante gibt es zwei Möglichkeiten die Update–Funktion zu gestallten. Die eine nennt sich SARSA und versucht die besten Q–Values zu finden.

Die Update–Funktion ist

$$Q(a, i) \leftarrow Q(a, i) + \alpha(\mathrm{R(i) + Q(\acute{a}, j) - Q(a, i)})$$

Wie man leicht erkennt ist das die gleiche Funktion wie beim Ansatz des Lernens mittels Modell. Nur daß jetzt nicht Zustandsnutzen angepaßt werden, sondern Q–Values verwendet werden.

Die andere Variante wird als Q–Learning bezeichnet und versucht einen deterministischen optimalen Q–Value zu finden.

Die Update–Funktion für diese Variante lautet

$$Q(a, i) \leftarrow Q(a, i) + \alpha(\mathrm{R(i)} + \max_{\mathrm{\acute{a}}} \mathrm{Q(\acute{a}, j) - Q(a, i)})$$

Hier wird die maximal nützlichste Aktion vom Agenten ausgewählt.

Wie schon erwähnt und bestimmt durch die Update–Funktionen bemerkt bedeutet Q–Values zu lernen, lernen ohne ein Modell der Umgebung.

Deshalb schaut ein Q–Learning Agent wie folgt aus:

```
function Q-LEARNING-AGENT(e) returns an action
   static: Q, a table of action values
           N, a table of state-action frequencies
           a, the last action taken
           i, the previous state visited
           r, the reward received in state i

   j ← STATE[e]
   if i is non-null then
      N[a, i] ← N[a, i] + 1
      Q[a, i] ← Q[a, i] + α(r + max_a' Q[a', j] − Q[a, i])
   if TERMINAL?[e] then
      i ← null
   else
      i ← j
      r ← REWARD[e]
   a ← arg max_a' f(Q[a', j], N[a', j])
   return a
```

So wurde die Nutzentabell U durch die Tabell für die Q–Values ersetzt. Die Tabelle für das Modell ist im Q–learning–Verfahren hinfällig und wurde deshalb aus dem Algorithmus entfernt. Der Rest ist das so fast identisch mit dem Aktiven–Agenten, nur die Teile wurden für das Q–learning angepaßt.

7 Generalisierung im Reinforcment Learning

In den bisher vorgestellten Varianten wurde alles Wissen im Agenten mittels Tabellen oder anderen Datenstrukturen dargestellt. Dies ist eine explizite Darstellung einer Ausgabe für jedes Eingabetupel. Bei dieser Darstellung steigt die Zeit bei großen Zustandsmengen für die Konvergenz und der Iteration rapide, was ein großer Nachteil ist. So bestizt zum Beispiel Schach oder Backgammon einen Zustandsraum von 10^{50} bis 10^{120} Zuständen. Eine Lösung dieses Problems ist eine Änderung zur impliziten Darstellung. Diese erlaubt die Berechnung der Ausgabe für jede Eingabe, so reduzieren sich die 10^{120} Zustände auf n Gewichte im Schach. Dies geschieht mittels Nutzung von linear gewichtete Funktion

$$U(i) = w_1 f_1(i) + w_2 f_2(i) + ... + w_n f_n(i)$$

mit $f_1(i), ..., f_n(i)$ als Merkmale der Umgebung.

Das eigentlich Intressante neben der Kompression ist die Möglichkeit des Lerners von besuchten Zuständen auf nicht besuchte Zustände zu schließen. Dies wird auch als **input generalization** bezeichnet. Das Problem bei dieser Methode ist die Findung einer Funktion, die der wirklichen Nutzenfunktion ziemlich ähnlich ist. Gleichzeitig erhöht sich die Zeitdauer, die gebraucht wird, um die Funktion zu lernen, mit der Größe des Suchraums.

Wie kann dies nun bei den bisherigen Algorithmen verwendet werden?

Beim LMS wird am Ende jeder Trainingssequenz ein reward–to–go mit jedem besuchten Zustand verknüpft und danach als ein benanntes Beispiel für einen induktiven Lernalgorithmus verwendet. Das Ergbeniss dieses Lernalgorithmus ist eine Nutzenfunktion U(i). Mit der dann in Verlaufe des Lernprozesses weiter gearbeitet wird bis sich alle Werte und die Ergebnissfunktion des Lernalgorithmusses übereinstimmen.

Hingegen beim TD gibt es zwei Varianten.

Die erste ersetzt die U– bzw. Q–Tabelle durch eine implizite Darstellung. Dann werden die Ergebnisse der Update–Funktion statt in die Tabelle zu schreiben als Beispiel für den Lernalgorithmus verwendet. Die gelernte Funktion muß dann beim nächsten Updatedurchlauf wieder verwendet werden. Somit ist der Lernalgorithmus dieser Variante inkrementell.

Die andere Möglichkeit verwendet eine lernende Funktion mit einem Gewichtsvektor **w**, ähnlich wie bei einem Neuronalennetz oder einer linear gewichteter Funktion. Statt aber jetzt die Nutzenfunktion anzupassen, werden jetzt die Gewichte angepaßt, so daß der zeitliche Unterschied in U zwischen dem jetzigen Status und seinem Nachfolger verringert wird.
Deshalb benötigt der Agent eine neue Update–Regel:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha[\mathrm{r} + \mathrm{U_w(i)}]\nabla\mathbf{w}\mathrm{U_w(i)}$$

wobei $\nabla\mathbf{w}U_w(i)$ zur Minimierung des Fehlers dient.

Eine ähnliche Update–Funktion kann beim Q–Learning verwendet werden.

Da die Nutzen– und Aktions–Werte–Funktion reelle Ausgaben haben, können dies für die Durchführungselemente Neuronaler–Netze und anderer kontinuierlicher Funktionen verwendet werden.

8 Beispiel: Backgammon

Ein berühmtes Beispiel im Reinforcment Learning ist das TD–Gammon Programm von Tesauro aus dem Jahre 1992. Der dort verwendete Algorithmus war eine Kombination aus dem TD(α) und einer nichtlinearen Funktionsapproximation.

Beim Backgammon gibt es einen sehr großen Zustandsraum, da das Spiel von 2 Spielern mit 2 Würfeln und 30 Spielsteinen gespielt wird. Somit besitzt Backgammon 24 Positionen (26 mit Spielfeldrahmen und Aussenbereich des Spielfeldes) an denen ein Spielstein sein kann. Es gibt 20 verschiedene Wege bei einem typischen Wurf zu spielen, gleichzeitig muß auch der Gegner und seien Züge berücksichtigt werden. Deshalb besitzt dieses Spiel einen Spielbaum von über 400 Verzweigungen und dies ist definitiv zu Groß für die Verwendung von Standardheuristicen.

Tesauro nutzte also ein Neuronalesnetz mit 198 Eingabeknoten und 40–80 verdeckten Knoten für die Einschätzung der Position. Dieses Neuronalenetz bekamm als Eingabe eine Backgammonposition und lieferte als Ausgabe die Einschätzung dieser Position. Warum aber 198 Eingabeknoten? Nun jeder Punkt auf dem Spielfeld wurde durch vier Einheiten repräsentiert. Diese hatten als Wert entweder 0 oder 1. Also wenn sich kein Stein auf einem Backgammonzacken befindet, dann hatten alle 4 Knoten den Wert 0. Bei einem Stein bekamm der 1.Knoten eine 1, bei zwei Steinen der 1. und 2. Knoten, usw. bis die Zahl größer als 3 war, denn dann bekamm der vierte Knoten einen Wert von $\frac{(n-3)}{2}$.

Somit gibt es also 4 Knoten für die Farbe weiß und 4 Knoten für die Farbe schwarz für 24 Positionen, was das ganze auf einen Wert von 192 Knoten bringt. Dann gab es noch 2 Knoten für die Anzahl der Spielsteine auf dem Rand mit dem Wert von $\frac{n}{2}$, 2 Knoten für die Anzahl der entfernten Spielsteine mit einem Wert von $\frac{n}{15}$ und 2 Knoten für die Anzeige ob weiß oder schwarz am Zug ist.

Alles in allem also 198 Eingabeknoten.

Diese Eingaben wurden dann mit ihren Gewichten multipliziert und dann addiert. Diese Werte bekammen dann die verdeckten Knoten. Diese berechneten dann mit Hilfe einer Sigmoid–Funktion wiederum ihre Ausgabe.

So berechnete der verdeckte Knoten j mittels

$$h(j) = \sigma\Big(\sum_{\mathrm{i}} \mathrm{w_{ij}}\phi(\mathrm{i})\Big) = \frac{1}{1 + \mathrm{e}^{-\sum_{\mathrm{i}} \mathrm{w_{ij}}\phi(\mathrm{i})}}$$

seine Ausgabe.

Das Lernen geschah im Bereich der Gewichtveränderungen im Neuronalennetz mittels der Update–Funktion:

$$\vec{\theta} = \vec{\theta} + \alpha\Big[\mathrm{r} + \mathrm{U(j)} - \mathrm{U(i)}\Big]\vec{\mathrm{e}},$$

wobei $\vec{\theta}$ den Vektor der Gewichte (zu Beginn randomisiert) darstellt und $\vec{e}$ den Vector der Auswahlspuren für jede Komponente von $\vec{\theta}$. Der letzte Vektor wird mittels einer eigenen Update–Funktion aktualiesiert:

$$\vec{e} = \lambda\vec{\mathrm{e}} + \nabla_{\vec{\theta}}\mathrm{U(i)}$$

Das Programm lernte dann durch Spiel gegen sich selbst. Um einen Zug zu wählen, wurde jeder der 20 Wege, die je Wurf gespielt werden konnte, und ihre Ergebnissposition berücksichtigt. Danach wurde der Zug mit dem höchsten Nutzen ausgewählt.

Am Anfang dauerten die Spiele einige hundert oder tausende Züge bis einer zufällig gewann. Aber nach einigen dutzend Spielen stieg die Performance rapide an. Nun war TD-Gammon in der Lage Neurogammon (ein früherer Bruder von TD-Gammon und gleichzeitig Olympiagewinner in Backgammon von 1989) zu besiegen. Spätere Versionen von TD–Gammon benutzten zwei/drei–schichtige Suchprozeduren. Die erste Suche berechnete eine Menge von guten Positionen für den Spielzug, danach suchte TD–Gammon aus diesem Raum den besten mittels der zweiten Suchprozedur heraus. Somit konnte jetzt TD–Gammon die besten Spieler der Welt schlagen.

9 Literatur

"Artificial Intelligence – A Modern Approach",
Stuart Russel und Peter Norvig;

"Reinforcment Learning",
Sutton und Baro;

"Reinforcment Learning – A Survey",
Leslie Pack Kaelbling, Michael Littman und Andrew W. Moore;

"online Fitted Reinforcment Learning",
Geoffrey Gordon;

"Reinforcment Learning with Perceptual Aliasing",
Loonie Chrisman;

"Packet Routing in Dynamically Changing Networks",
Boyan und Littman;

"Reinforcment Learning Methods for Military Applications",
Malcom Strens;

"a distributed dynamic world model for Robot soccer",
Frans Groen, Jeroen Roodhart und Joost Vunderink;