Reinforcment learning

Author: Alexander Camek

Betreuer: Prof. Kramer

03.02.2004

Inhalt

Einführung in den Bereich Learning

Was heißt eigentlich ”Learning” ? Was versteht man darunter ?

Definition

Lernen bedeutet sich besser zu verhalten auf Grund von Erfahrungen

Unterteilung:

- supervised learning
- reinforcment learning

Inhalt

Reinforcment Learning

Grundidee ist die Belohnung oder Bestrafung für das erreichte Ziel.
Egal, ob dies vom Lehrer oder vom Programm selbst kommt.

Unterschied:

- Es gibt nur Belohnung oder Bestrafung
- Es wird **NICHT** gesagt,
  was schlecht war oder
  wie es besser gemacht werden könnte.

Nachteil: Der Agent weiß nicht was er falsch gemacht hat!

Reinforcment Learning

Folgende Aufgabenstellungen sind dabei zu beachten:

- Die Umgebung kann zugänglich oder unzugänglich sein:
  - zugänglich: Zustände und Wahrnehmungen sind verknüpft
  - unzugänglich: innere Zustände bilden Umgebung ab
- vorgegebenes Wissen über Umgebung und Ergebnisse oder Lernen eines Modells und zusätzlichem Wissen
- Belohnung für erreichte Ziele oder für irgendeinen Zustand
- Belohnungen:
  - Teil der zu erfüllenden Aufgabe (z.B. Punkte im Tischtennis)
  - Hilfestellungen für die Aufgabe (z.B. ”guter Zug”)
- Art des Lernens:
  - passiv
  - aktiv

Reinforcment Learning

Der Agent lernt eine Nutzenfunktion von Zuständen.
Benutzt diese, um mittels Aktionen seinen erwarteten Nutzen zu maximieren.

Der Agent lernt eine action-value Funktion.
Diese berechnet den erwarteten Nutzen einer Aktion in einem Zustand.
(auch ”Q–learning” genannt)
Vorteil:

- Kenntnisse nur über die erlaubten Bewegungen
- einfacheres Design

Nachteil:

- nicht vorausschauend, Zielrichtung nicht bekannt
- Lernfähigkeit eingeschränkt

passives Lernen in bekannter Umgebung

Geschieht durch das Generieren von Zustandsübergängen ähnlich wie beim Automaten.

Modell: M_{ij}, Wahrscheinlichkeit eines Zustandsübergangs von i nach j

Ziel:

- Erhalt des erwarteten Nutzens in einem bestimmten Zustand
- Erhalt des Nutzen einer Trainingssequenz

Definition: reward–to–go

Die Summe der Belohnung von diesem Zustand bis zum Endzustand

$\Rightarrow$ erwarteter Nutzen eines Zustandes = erwarteter reward–to–go dieses Zustandes

passives Lernen in bekannter Umgebung

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]]
  $U \leftarrow$ UPDATE($U, e,$ *percepts*$, M, N$)
  if TERMINAL?[e] **then** *percepts* $\leftarrow$ the empty sequence
  return the action *Observe*

passives Lernen in bekannter Umgebung

$U \leftarrow UPDATE(U, e, percepts, M, N)$

Das Wichtigste bei Reinforcment Learning ist der UPDATE–Algorithmus

3 Mögliche Ansätze:

- Naiveransatz mittels LMS (least mean square)
- adaptive dynamic programming (ADP)
- temporal difference (TD)

passives Lernen in bekannter Umgebung

Naiver Ansatz:

Idee:

beobachteter reward–to–go jedes Zustandes
läßt sich auf den im Moment
erwarteten reward–to–go direkt zurückschließen.

Ziel:

- Ermittlung des beobachteten reward–to–go
- Ermittlung des geschätzten Nutzens

LMS erzeugt einen Nutzenschätzer mit minimalen durchschnittlichen quadratischen Fehler in Bezug auf die beobachteten Daten.

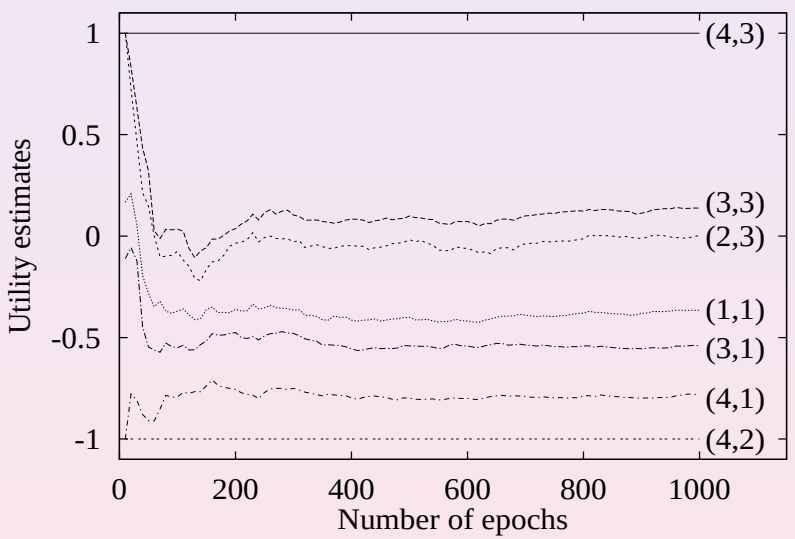
$\Rightarrow$ Reduzierung auf induktives Lernen

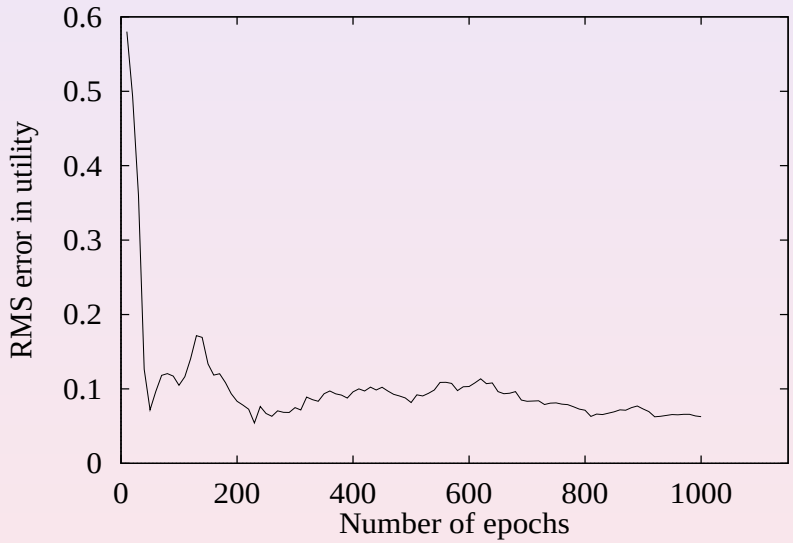
passives Lernen in bekannter Umgebung

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

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

passives Lernen in bekannter Umgebung



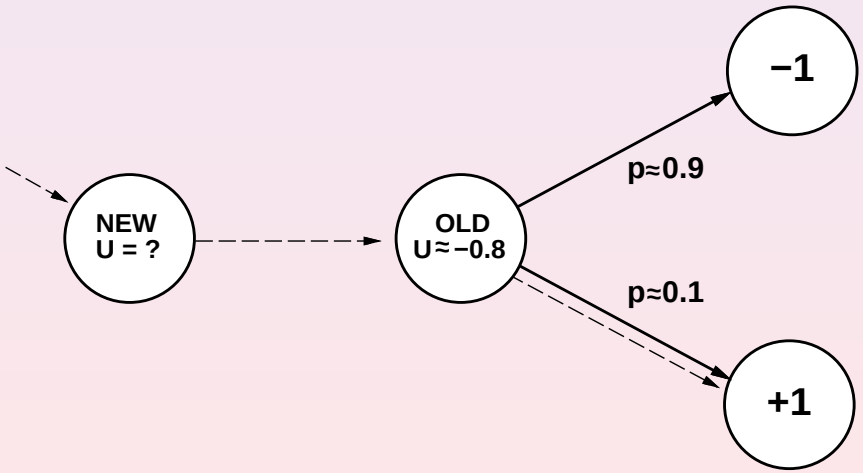


passives Lernen in bekannter Umgebung

Adaptive dynamic programming (ADP)
Idee:

- Nutzen des Wissens über die Umgebung
- Nutzen eines Modells

Unterschied:



bei LMS hätte NEW einen Nutzen von 1

passives Lernen in bekannter Umgebung

Lösung: Speicherung der Übergangswahrscheinlichkeiten in Tabelle M_{ij}

$\Rightarrow$ Das Problem reduziert sich zu einem wohldefinierten sequentiellen Entscheidungsproblem, sobald alle Zustände und deren Belohnungen bekannt sind

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

passives Lernen in bekannter Umgebung

Temporal difference learning (TD)

Idee:

- $U(i) = R(i) + \sum_j M_{ij} U(j)$
- ohne dies für alle möglichen Zustände zu tun

Lösung:

Nutzen der beobachteten Übergänge zur Wertanpassung der beobachteten Zustände

Beipiel:

- Übergang i $\rightarrow$ j $\Rightarrow$ U(i) = -0.5 und U(j) = +0.5
- Verbesserung von U(i), so daß der Wert besser zum Nachfolger paßt

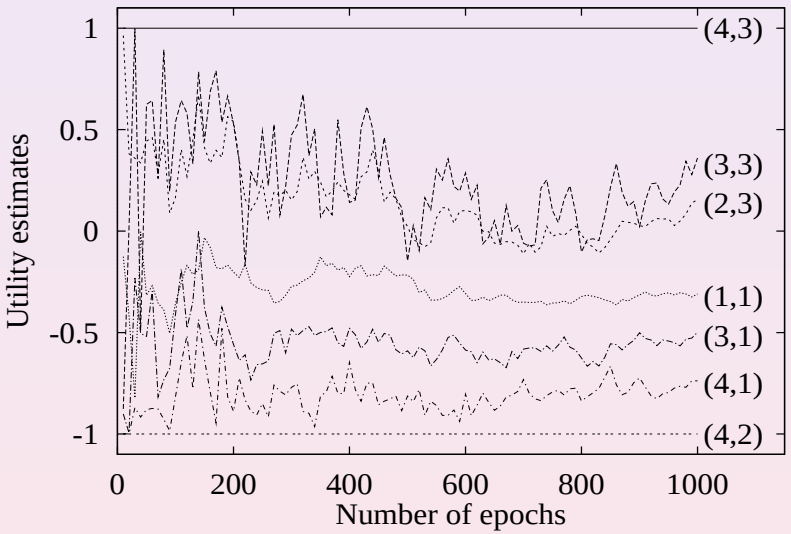
$$U(i) \leftarrow U(i) + \alpha(R(i) + U(j) - U(i))$$

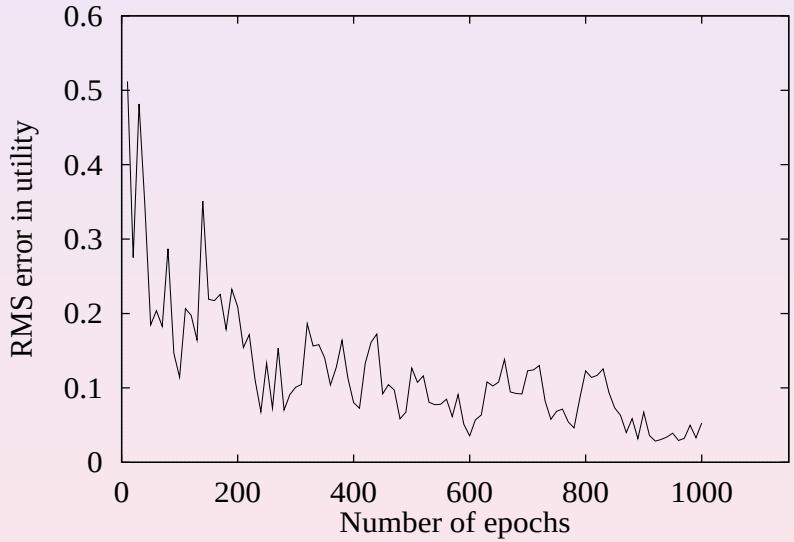
passives Lernen in bekannter Umgebung

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

if TERMINAL?[e] **then**
U[STATE[e]] $\leftarrow$ RUNNING-AVERAGE(U[STATE[e]], REWARD[e], N[STATE[e]])
else if *percepts* contains more than one element **then**
e' $\leftarrow$ the penultimate element of *percepts*
i, j $\leftarrow$ STATE[e'], STATE[e]
U[i] $\leftarrow$ U[i] + $\alpha(N[i])$(REWARD[e'] + $U[j]$ - $U[i]$)

passives Lernen in bekannter Umgebung





passives Lernen in unbekannter Umgebung

Unterschied: Modell wird angepaßt

$\Rightarrow$ ADP-Ansatz funktioniert besser, da dieser auf dem gesamten Modellraum arbeitet

Lösung für TD–Nutzung:

- durchschnittliche Anpassung über eine große Menge von Zustandsübergängen, da Häufigkeiten jedes Nachfolgers nährungsweise proportional zu seiner Wahrscheinlichkeit ist
- Benutzung eines Umgebungsmodells zur Generierung von pseudo–Experimenten
  $\Rightarrow$ TD $\rightarrow$ ADP

Nachteil: höherer Berechnungsaufwand

passives Lernen in unbekannter Umgebung

Anpassung des ADP–Ansatzes:
Approximation: Begrenzung der Anpassungsschritte
Heuristic: prioritized–sweeping
prioritizied–sweeping:

- Hinzufügen von Wissen über Vorgänger
- Priorisierung der wahrscheinlichsten Nachfolger
- Aktualisierung des Zustands mit höchster Priorität
- Ablauf:

Step1 $U_{old} = U(i)$
Step2 $U(i) = R(i) + \sum_j M_{ij} U(j)$
Step3 Zustandspriorität O
Step4 $\Delta = |U_{old} - U(i)|$
Step5 Δ nutzen zur Prioritätsänderung bei Vorgänger

$\Rightarrow$ approximierter ADP $\approx$ voller ADP
Vorteil: keine aufwendigen Iterationen am Anfang

aktives Lernen in unbekannter Umgebung

Unterschied:

- Welche Aktionen werden ausgeführt
- Was sind die Ergebnisse
- Wie wirkt das ganze auf die Belohnung

Lösung zur Wandlung passiv in aktiv:

- Modell der Übergänge wird geändert.
  Zustand j wird aus i erreicht mittels Aktion a.
  kurz: M_{ij}^{a}
- Berücksichtigung der Wahlmöglichkeiten
  Ein rationaler Agent will grundsätzlich Nutzenmaximierung
  Deshalb: $U(i) = R(i) + \sum_j M_{ij} U(j) \rightarrow U(i) = R(i) + \max_a \sum_j M_{ij}^{a} U(j)$
- Der Agent MUSZ in jedem Schritt eine Aktion ausführen

aktives Lernen in unbekannter Umgebung

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]] $\leftarrow$ REWARD[e]
  $M \leftarrow$ UPDATE-ACTIVE-MODEL(M, *percepts*, *last-action*)
  $U \leftarrow$ VALUE-ITERATION(U, M, R)
  if TERMINAL?[e] **then**
    percepts $\leftarrow$ the empty sequence
  last-action $\leftarrow$ PERFORMANCE-ELEMENT(e)
  return *last-action*

aktives Lernen in unbekannter Umgebung

Änderungen in den bisherigen Algorithmen:

ADP:

- Änderung des M_{ij}-Modells zum M_{ij}^{a}-Modell
- Hinzufügen einer dritten Dimension
- Änderung auch bei der UPDATE-Funktion

TD:

- Das Modell ist dasselbe wie beim ADP
- UPDATE-Funktion muß **NICHT** geändert werden
- TD $\equiv$ ADP, wenn die Anzahl der Trainingssätze gegen ∞ gehen

Exploration

last action ←PRFORMANCE–ELEMENT(e)

Frage: Was soll als Ergebnis zurückgeliefert werden?

Klar: Der Agent sollte die höchst nützlichste Aktion auswählen

Aber: Was ist mit dem Lernen?

Gedanken:

- Erhalt der Belohnung für den aktuellen Trainingssatz
- Ermöglichen des Lernens für den Erhalt zukünftiger Belohnungen

Problem: Zwiespalt zwischen sofortigem Gewinn und dem Lernen

Deshalb: Finden einse Mittelweges

Exploration

greedy–Ansatz:

Optimierung durch Auswahl von Aktionen
mit nicht starkem negativen Einfluß

wacky–Ansatz:

z.B. Boltzmann exploration
Treffen der Auswahl mittels Zufall
ER(a) = erwartete Belohnung bei Ausführung von Aktion a

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

Idee:

- Kombination aus beidem
- Änderung der Nutzenfunktion
- unerkundeter Zustand = hoher Nutzen

Exploration

$\Rightarrow U^+(i)$ = optimisticher Schätzer

$N(a,i)$ = Anzahl, wie oft a in i ausgeführt wurde

$\Rightarrow$ ADP mit Wertiteration: $U^+(i) \leftarrow R(i) + \max_a f\left(\sum_j M_{ij}^a U^+(j), N(a,i)\right)$

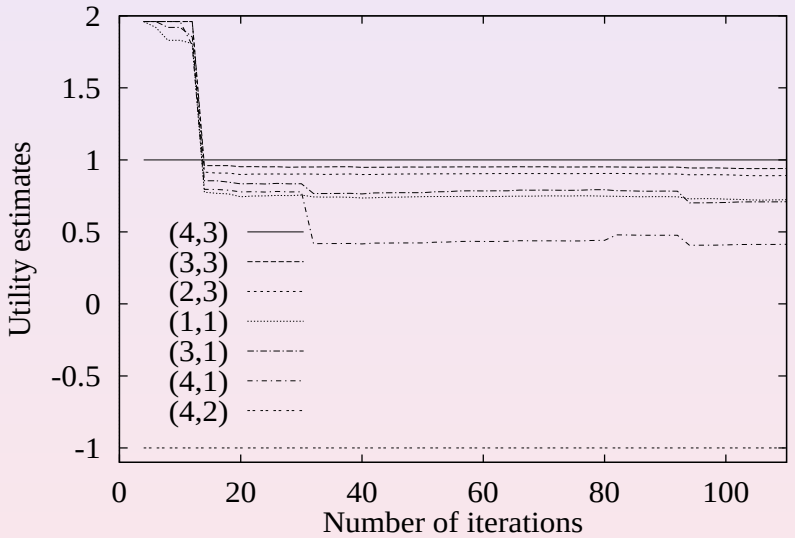
$f(u,n)$ = Explorationsfunktion,

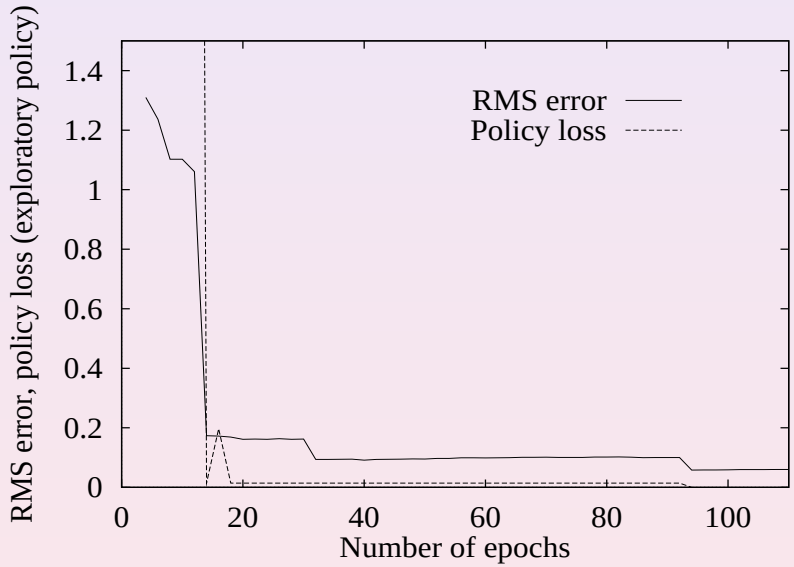
z.B. $f(u,n) = \begin{cases} R^+, n < N_e \\ U, sonst \end{cases}$

R^+ = optimistischer Schätzer für bestmöglichen Gewinn

N_e = fest, Grenze wie oft Aktions–Zustandspaar ausprobiert wird.

Exploration





Q-Values und Aktion-Wertepaare

Definition: ”action–value function”

Ordnet einem erwarteten Nutzen
eine Durchführung von einer Aktion in einem Zustand zu.
$\Rightarrow$ Q–values, kurz: $Q(a, i)$
Q–values sind direkt mit Nutzenwerte verbunden: $U(i) = \max_a Q(a, i)$

Wichtig:

- Genügen für Entscheidung ohne Modell
- Lernbar durch Belohnungsfeedback

Bei Korrektheit **muß** folgendes gelten: $Q(a, i) = r(i) + \sum_j M_{ij}^a \max_{á} Q(á, i)$

Dies kann als Updatefunktion beim ADP benutzt werden.

Q-Values und Aktion-Wertepaare

Beim TD Q-learning gilt:

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

SARSA Methode:

$$Q(a,i) \leftarrow Q(a,i) + \alpha(R(i) + Q(á,j) - Q(a,i))$$

Unterschied:

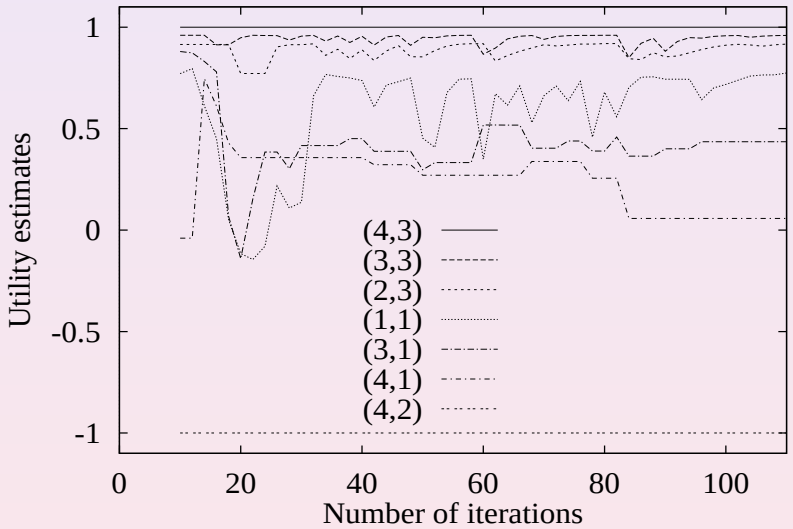
- SARSA versucht den besten Q–Value zu finden
- Q-learning lernt einen deterministischen optimalen Q–Value

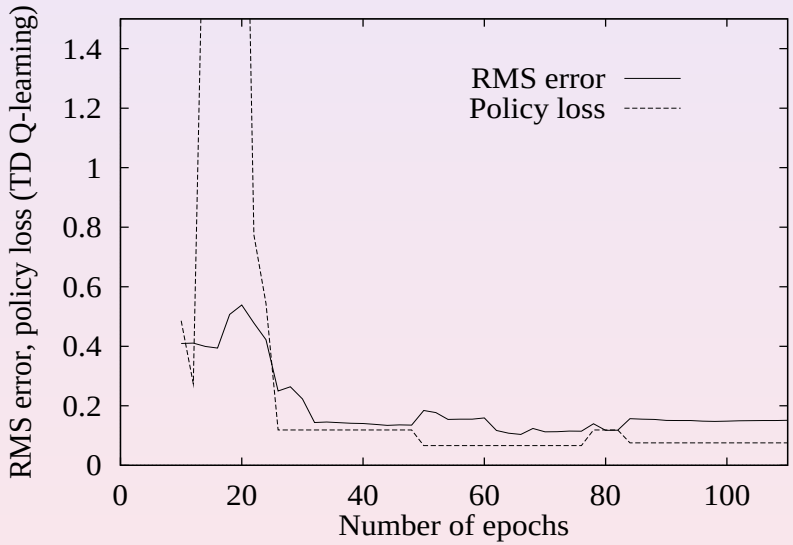
Q-Values und Aktion-Wertepaare

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 $\leftarrow$ STATE[e]
  if i is non-null **then**
    $N[a,i] \leftarrow N[a,i] + 1$
    $Q[a,i] \leftarrow Q[a,i] + \alpha(r + \max_{a'} Q[a',j] - Q[a,i])$
  if TERMINAL?[e] **then**
    $i \leftarrow$ null
  else
    $i \leftarrow j$
    $r \leftarrow$ REWARD[e]
  $a \leftarrow \arg\max_{a'} f(Q[a',j], N[a',j])$
  return a

Q-Values und Aktion-Wertepaare





Generalisierung in Reinforcment Learning

Stand:

- Agent(U, M, R, Q)
- interne Darstellung mittels Tabellen
- explizite Darstellung einer Ausgabe für jedes Eingabetupel

Nachteil:

bei großen Zustandsmengen steigt die Zeit für die Konvergenz und der Iteration rapide

Beispiel: Schach oder Backgammon besitzt 10^{50} bis 10^{120} Zustände

Generalisierung in Reinforcment Learning

implizite Darstellung

- Erlaubt die Berechnung der Ausgabe pro Eingabe
- z.B. Brettspiel: $U(i) = w_1 f_1(i) + w_2 f_2(i) + ... + w_n f_n(i)$

  $\Rightarrow$ Reduzierung von 10^{120} Werte auf n Gewichte
  (bei Schach 10 Gewichte)

Vorteil:

- Kompression
- Rückschluß von besuchten Zuständen auf nicht besuchte Zustände
  $\Rightarrow$ input generalization

Nachteil:

- Keine Funktion findbar, die der Nutzenfunktion ähnlich ist
- Größe des Suchraums = Lernzeit der Funktion

Generalisierung in Reinforcment Learning

Auswirkung auf die bisherigen Funktionen

LMS:

- Ende jeder Trainingssquenz wird ein reward–to–go mit jedem besuchten Zustand verknüpft

  $\Rightarrow < Zustand, Belohnung >$

- Verwendung des Paares als benanntes Beispiel für einen induktiven Lernalgorithmus

  $\Rightarrow$ Ergebniss ist eine Nutzenfunktion: $U(i)$

Generalisierung in Reinforcment Learning

TD:

- U- und/oder Q-Tabelle ersetzen durch implizite Darstellung
- Rückgabewert der Updatefunktion ist Beipiel für Lernalgorithmus
- gelernte Funktion wird beim nächsten Updatedurchlauf benutzt
  $\Rightarrow$ Lernalgorithmus ist inkrementell

- Lernfunktion nutzt einen Gewichtsvektor $\mathbf{w}$
- Gewichte werden angepaßt, so daß zeitlicher Abstand zwischen Zustand und Nachfolger verringert wird
- neue Update–Regel: $\mathbf{w} \leftarrow \mathbf{w} + \alpha[r + U_w(i)]\nabla \mathbf{w} U_w(i)$
  $U_w(i)$ = Nutzenfunktion
  $\Rightarrow$ ähnliche Funktion kann beim Q–learning angewandt werden

Inhalt

Militärbereich

- Simulation (z.B. Flugsimulationen)
- logistischen Planung
  - Resourcenplanung
  - Zeitplanung
- autonoumen Entscheidungen
  - Zielsuche gegen reagierende Ziele
  - Trajectionsoptimierung in wechselnden Umgebungen
  - Sensorkontrolle und dynamische Resourcenbelegung
- automatische Entscheidungen
  - schnelle Reaktion (z.B. elektronische Kriegsführung)
  - low–level Kontrolle (z.B. Kontrolle für Roboter mit Füssen)

Roboterbereich

- Lernen des Balanzierens und
  der Koordination beim Laufen oder Jonglieren
- Roboterfußball (z.B. Clockwork–Orange Team)
- Kartierung von Innenräumen und der Lokalisation

Schach

Die erste Anwendung von Reinforcment Learning war das Schachprogramm von Arthur Samuel (1959;1967)

Bestandteile des Programms:

- weighted linear function zur Positionsevaluierung
- Gewichte wurden angepaßt
- bis zu 16 Terme genutzt

Unterschied:

- Anpassung der Gewichte mittels Unterschied zwischen aktuellem und gemerkten Wert
- Vorausschau im Suchbaum
- keine Benutzung von ermittelten Belohnungen
  (Aber: Voraussetzung, daß Gewichte immer positiv)

Backgammon

Neurogammon (Tesauro und Sejnowski, 1989)

- Neuronalesnetzwerk mit Q(a,i)
- Vorkenntnisse im Backgammonspiel
- Beispiele mit bewerteten Zügen
- Gewann die Backgammon–Olympiade 1989
- Nachteil: sehr gute Spieler konnten es besiegen

Backgammon

TD–Gammon (Tesauro, 1992)

Algorithmus:
Kombination aus TD(λ) und nichtlinearer Funktionsapproximation

Problem:

- 30 Spielsteine und 24 Positionen
  (26 mit Rahmen und dem Außenbereich)
- 20 verschiedene Wege je nach Wurf
- Reaktion auf Gegner

$\Rightarrow$ Spielbaum hat über 400 Verzweigungen und zu viele Zustände

Backgammon

Design:

- Belohnung nur am Ende des Spiels
- Neuronalesnetzwerk mit 198 Eingabeknoten und 40–80 versteckten Knoten
- Eingabe: Backgammonposition
- Ausgabe: Einschätzung der Position

Jeder Punkt im Spielfeld wird dargestellt mittels 4 Eingabeknoten pro Farbe.

	kein Stein:	alle 4 Knoten sind 0
	ein Stein:	1.Knoten ist 1
$\Rightarrow$	zwei Steine:	1. und 2.Knoten ist 1
	drei Steine:	1., 2. und 3.Knoten ist 1
	vier Steine:	4.Knoten ist $\frac{(n-3)}{2}$ (if n $>$ 3)

mit n = Anzahl der Steine auf dem Punkt.

Backgammon

4 weiße Knoten + 4 schwarze Knoten für 24 Positionen $\Rightarrow$ 192 Knoten

+ 2 Knoten für die Anzahl der Spielsteine auf dem Spielfeldrand (Knotenwert = $\frac{n}{2}$)

+ 2 Knoten für die Anzahl der entfernten Spielsteine (Knotenwert = $\frac{n}{15}$)

+ 2 Knoten für die Anzeige des aktuellen Zuges

$\Rightarrow$ 198 Knoten

Backgammon

Berechnung im Neuronalen–Netz:

Zuerst werden die Eingaben mit ihren Gewichten multipliziert und dann bei den verdeckten Knoten aufsummiert.

Ausgabe des verdeckten Knoten j — h(j) — ist eine nichtlinear Sigmoid–Funktion:

$$h(j) = \sigma\left(\sum_i w_{ij}\phi(i)\right) = \frac{1}{1+e^{-\sum_i w_{ij}\phi(i)}}$$

mit $\phi(i)$ = Wert des i.Eingabeknotens
und w_{ij} = Gewicht von Eingabe- zum verdeckten Knoten.

Der Ausgabeknoten generiert die gewichtete Summe und schickt diese durch die selbe sigmoid Funktion.

$\Rightarrow$ Einschätzung der Position

Backgammon

Lernen:

Update–Regel: $\vec{\theta} = \vec{\theta} + \alpha \left[r + U(j) - U(i) \right] \vec{e}$

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

$\vec{\theta}$ = Vektor der Gewichte (zu Beginn randomisiert)
$\vec{e}$ = Vector der Auswahlspuren für jede Komponente von $\vec{\theta}$

TD–Gammon lernte durch spielen mit sich selbst.

Dabei wählte es Züge indem jeder der 20 Wege pro Wurf und die Ergebnisspositionen brücksichtigt wurden.
Danach wurde der Zug mit dem höchsten Nutzen ausgewählt.

Backgammon

Netzgewichte wurden am Anfang zufällig gewählt.
$\Rightarrow$ erste Spiele dauerten hunderte bis tausende Züge bis eine Seite durch Zufall gewann.

Nach einem dutzend Spiele stieg die Performance rapide.

TD–Gammon besiegt somit Neurogammon.

Spätere Versionen hatten eine zwei/drei–schichtige Suchprozedur.
Auswahl eines Zuges erfolgt durch Überlegung, welchen Zug Gegner spielen könnte.

Zusammenfassung

Wegen seinem Potential zur Eleminierung von menschlichem Code bei Kontrollstrategien ist Reinforcment Learning eine der wichtigsten Gebiete der machine learning Forschung.

Starke Einsetzung im Bereich der autonomen Roboter.

Problem:

- Erweiterung dieser Methoden
- unzugänglichen Umgebungen

Zusammenfassung

Vielen Dank

für

Ihre Aufmerksamkeit!

Literaturhinweise

- Artificial Intelligence – A Modern Approach
  Stuart Russel und Peter Norvig
- Reinforcment Learning
  Sutton und Baro
- Reinforcment Learning – A Survey
  Kaelbling, Littman und Moore
- Online Fitted Reinforcment Learning
  Geoffrey Gordon

Literaturhinweise

- 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