

Grundlagen der Künstlichen Intelligenz

17 Verstärkendes Lernen

Passives und aktives verstärkendes Lernen

Volker Steinhage

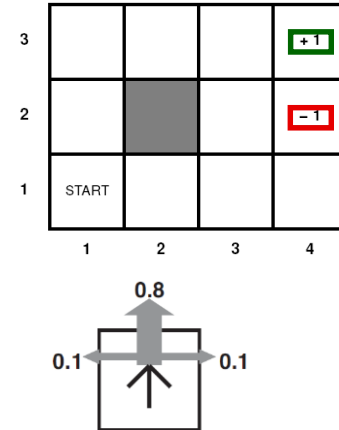
Inhalt

- Motivation
- Rückblick: MDP-basierter Agent
- Passives verstärkendes Lernen
 - Naives Aktualisieren
 - Adaptive Dynamische Programmierung
 - Zeitliche Differenz
- Aktives verstärkendes Lernen
 - Exploration
 - Q-Lernen
- Eingabe-Generalisierung
- Einige Anwendungen

Grundidee des verstärkenden Lernens

Geg.: MDP-basierte Agent

```
function SIMPLE-POLICY-AGENT(percept) returns an action  
static:  $M$ , a transition model  
          $U$ , a utility function on environment histories  
          $P$ , a policy, initially unknown  
  
if  $P$  is unknown then  $P \leftarrow$  the optimal policy given  $U, M$   
return  $P[\textit{percept}]$ 
```



Der Agent hat

- *keinen Lehrer*, der Ein-/Ausgabe-Beispiele bzw. richtige Aktionen vorgibt
- *kein Modell* $M_{ss'}^a$ der Umgebung
- *keine Nutzenfunktion* $U(s)$ für Zustände
- *keine Strategie* P

Der Agent erfährt nur, ob eine Aktion/Aktionsfolge belohnt wurde oder nicht

~ Belohnung/Bestärkung bzw. Reward/Reinforcement

Beispiele für verstärkendes Lernen

Beispiel 1:

- zufällige Züge in einem Brettspiel
- mögliche **Rewards**:
 - Gewinn bzw. Verlust einzelner Spielfiguren während des Spiels
 - Erreichen von bestimmten Spielstellungen (z.B. Schach)
 - Gewinn, Remis bzw. Verlust am Spielende

Beispiel 2:

- zufällige Schläge im Tischtennis
- mögliche **Rewards**:
 - Punktevergabe nach Schlagwechseln
 - Spielausgang

Passives vs. aktives verstärkendes Lernen

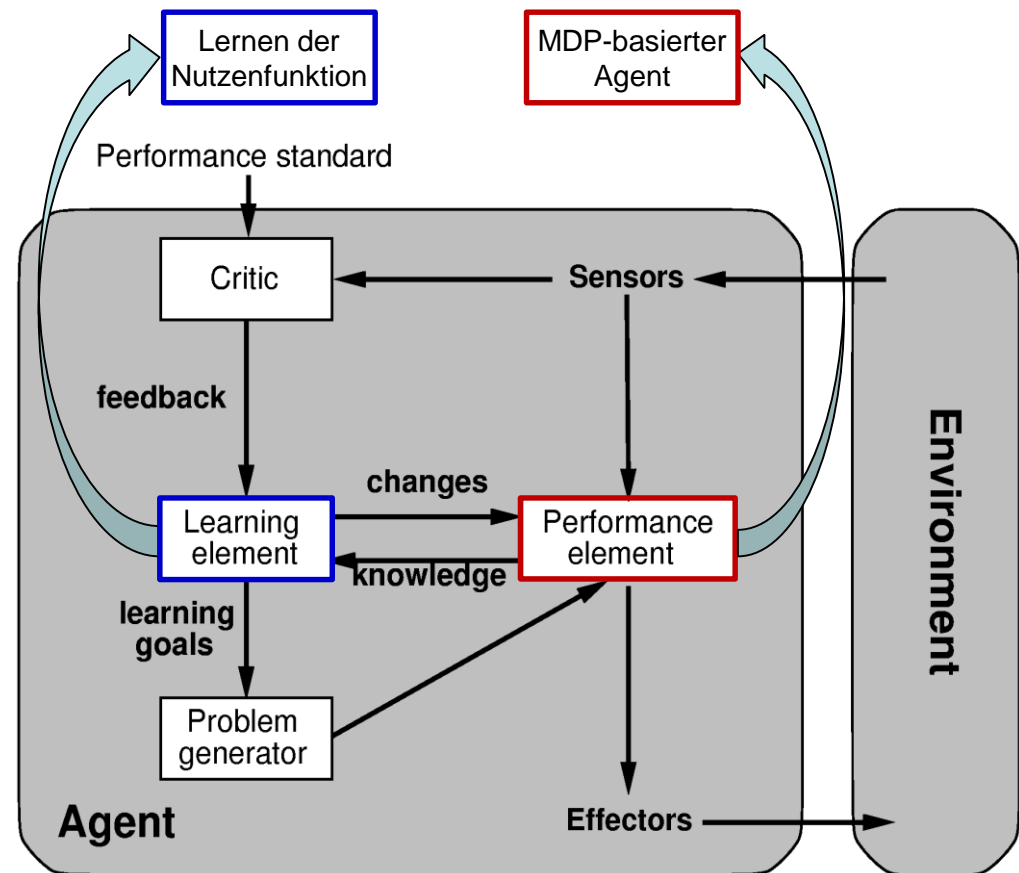
Passives verstärkendes Lernen:

der *Agent beobachtet lediglich* die Zustände und Änderungen in seiner Umwelt und versucht, den Nutzen verschiedener Zustände einzuschätzen

Aktives verstärkendes Lernen:

der *Agent selbst führt explorative Aktionen* aus, um unbekannte Zustände seiner Umgebung zu erkunden

→ Problem Generator



Rückblick: MDP-basierter Agent (1)

Der Begriff *Belohnung* wurde als „*Reward*“ im MDP-basierten Agentenkonzept eingeführt:

```
function SIMPLE-POLICY-AGENT(percept) returns an action  
  static:  $M$ , a transition model  
            $U$ , a utility function on environment histories  
            $P$ , a policy, initially unknown  
  
  if  $P$  is unknown then  $P \leftarrow$  the optimal policy given  $U, M$   
  return  $P[\textit{percept}]$ 
```

Gegeben:

- Menge von Zuständen in einer zugänglichen, stochastischen Umgebung
- Menge von Zielzuständen
- Menge von Aktionen
- Transitionsmodell $M_{ss'}^a$
- Nutzenfunktion

Transitionsmodell $M_{ss'}^a$: Wahrscheinlichkeiten, mit denen Zustände s' nach Ausführung von Aktionen a in Zustände s erreicht werden

Strategie (Policy): vollständige Abbildung von der Menge der Zustände auf die Menge der Aktionen

Gesucht: optimale Strategie, die den erwarteten Nutzen maximiert

Rückblick: MDP-basierter Agent (2)

Eine Nutzenfunktion U_h auf Zustandsfolgen heißt *separierbar* gdw. es eine Funktion f derart gibt, dass

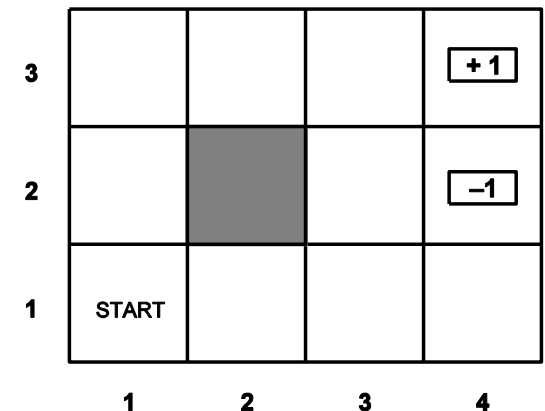
$$U_h([s_0, s_1, \dots, s_n]) = f(s_0, U_h([s_1, \dots, s_n]))$$

Die einfachste Form ist eine *additive* Gewinnfunktion R (*Reward*):

$$U_h([s_0, s_1, \dots, s_n]) = R(s_0) + U_h([s_1, \dots, s_n]).$$

Beispiel:

- M : $p(\text{Transition in gewählte Richtung}) = 0,8$
 $p(\text{Transition in gewählte Richtung } +90^\circ) = 0,1$
 $p(\text{Transition in gewählte Richtung } -90^\circ) = 0,1$
- $U(\text{Aktionsfolge}) = U(\text{Endzustand}) - \#\text{Aktionen}/25$



Rewards im Beispiel: $R((4, 3)) = +1$, $R((4, 2)) = -1$, $R(\text{sonstige}) = -1/25 = -0.04$.

Rückblick: MDP-basierter Agent (3)

Der Nutzen eines Zustandes s ist durch den erwarteten Nutzen der optimalen Strategie $policy^*$ unter Transitionsmodell M in s bestimmt:

Wegen der Additivität der Nutzenfunktion kann man die Utility $U(s)$ eines Zustands s auf den maximalen erwarteten Nutzen seiner Nachfolger in Form der Bellman-Gleichung reduzieren:

$$U(s) = R(s) + \max_a \sum_{s'} M_{ss'}^a U(s')$$

Die optimale Strategie ist dann:

$$\pi^*(s) = \arg \max_a \sum_{s'} M_{ss'}^a U(s')$$

Rückblick: MDP-basierter Agent (4)

Umsetzung: Approximative Berechnung durch iterative Anwendung der sogenannten **Bellmann-Aktualisierung**:

$$U_{t+1}(s) \leftarrow R(s) + \max_a \sum_{s'} M_{ss'}^a U_t(s'),$$



wobei $U_t(s)$ die Utility des Zustands s nach t Iterationen ist

Beobachtung: Mit $t \rightarrow \infty$ konvergieren die Utilities aller Zustände

Rückblick: MDP-basierter Agent (5)

function VALUE-ITERATION(M, R) **returns** a utility function

inputs: M , a transition model

R , a reward function on states

local variables: U , utility function, initially identical to R

U' , utility function, initially identical to R

repeat

$U \leftarrow U'$

for each state i **do**

$U'[i] \leftarrow R[i] + \max_a \sum_j M_{ij}^a U[j]$

end

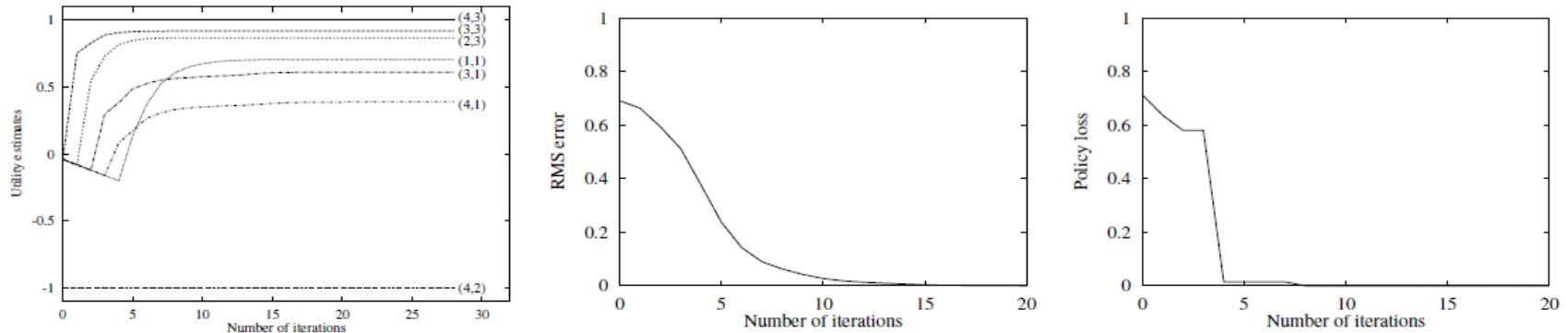
until CLOSE-ENOUGH(U, U')

return U



Rückblick: MDP-basierter Agent (6)

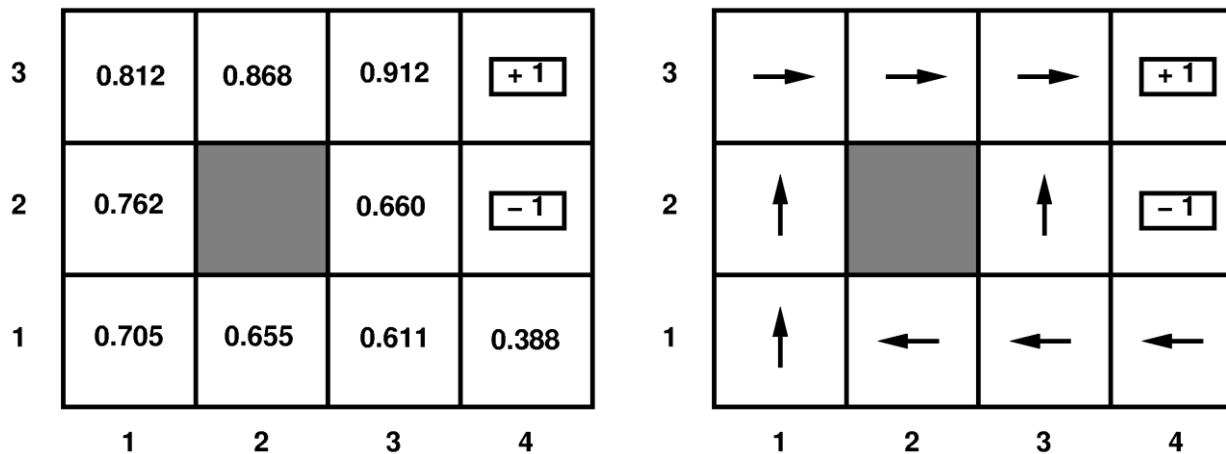
Konvergenz von Nutzen, Nutzenfehlern und Policy-Fehler:



Nutzenfehler: mittlerer quadrat. Fehler zw. aktueller und optimaler Nutzenfunktion.


Policy-Fehler: Differenz zw. erwartetem Nutzen bei aktueller und optimaler Policy.

Nutzen der einzelnen Zustände nach Konvergenz und resultierende Policy:



Passives verstärkendes Lernen

```
function PASSIVE-RL-AGENT(e) returns an action
  static: U, a table of utility estimates (initially empty)
           N, a table of frequencies for states (initially zero)
           M, a table of transition frequencies from state to state (initially zero)
           percepts, a percept sequence (initially empty)
            $\pi$ , a fixed policy
  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  $\pi$ (STATE[e])
```



Zunächst drei verschiedene Aktualisierungsfunktionen für den passiven Agenten:

1. Naives Aktualisieren (ohne explizites Transitionsmodell)
2. Adaptives Dynamisches Programmieren (mit explizitem Transitionsmodell)
3. Zeitliche Differenz (ohne explizites Transitionsmodell)

Passives Lernen einer Nutzenfunktion

Gegeben:

- vollständig beobachtbare Weltzustände
- Rewards der Zustände
- Passives Lernen: Umwelt erzeugt Zustandsübergänge, die der Agent wahrnimmt
 - damit legt die Umwelt die Policy π fest
 - der Agent hat keine Aktionsauswahl,
in jedem Zustand s wird die Aktion $\pi(s)$ ausgeführt

Unbekannt:

- Transitionsmodell $M_{ss'}^a$
- Nutzen der Zustände $U(s)$

Ziel:

- Lernen der Nutzen $U^\pi(s)$

Beispiel der bekannten 4x3-Welt

Annahme (oBdA): **die Umwelt führe** die uns bekannte optimale Policy aus

3	0.812	0.868	0.912	+ 1	3	→	→	→	+ 1
2	0.762		0.660	- 1	2	↑		↑	- 1
1	0.705	0.655	0.611	0.388	1	↑	←	←	←
	1	2	3	4		1	2	3	4

Der Agent beobachtet Trainingssequenzen vom Startzustand (1,1) zu den Endzuständen (4,2) bzw. (4,3) entsprechend der Policy π . Der Agent kann sowohl den Zustand als auch den damit verbundenen Reward erkennen.

Typische Trainingssequenzen sind z.B.:

$(1, 1) \xrightarrow{\frac{-1}{25}} (1, 2) \xrightarrow{\frac{-1}{25}} (1, 3) \xrightarrow{\frac{-1}{25}} (1, 2) \xrightarrow{\frac{-1}{25}} (1, 3) \xrightarrow{\frac{-1}{25}} (2, 3) \xrightarrow{\frac{-1}{25}} (3, 3) \xrightarrow{\frac{-1}{25}} (4, 3)_{+1}$

$(1, 1) \xrightarrow{\frac{-1}{25}} (1, 2) \xrightarrow{\frac{-1}{25}} (1, 3) \xrightarrow{\frac{-1}{25}} (2, 3) \xrightarrow{\frac{-1}{25}} (3, 3) \xrightarrow{\frac{-1}{25}} (3, 2) \xrightarrow{\frac{-1}{25}} (3, 3) \xrightarrow{\frac{-1}{25}} (4, 3)_{+1}$

$(1, 1) \xrightarrow{\frac{-1}{25}} (1, 2) \xrightarrow{\frac{-1}{25}} (1, 3) \xrightarrow{\frac{-1}{25}} (2, 3) \xrightarrow{\frac{-1}{25}} (3, 3) \xrightarrow{\frac{-1}{25}} (3, 2) \xrightarrow{\frac{-1}{25}} (4, 2)_{-1}$

Naives Aktualisieren (1)

Annahme: Nutzen $U(s)$ eines Zustandes s sei die *erwartete Summe der Rewards von s bis zum Endzustand*: $U^\pi(s) = E\left(\sum_{t=0}^{\infty} R(s_t) \mid \pi, s_0 = s\right)$

Lernen durch Trainingssequenzen: jede Sequenz (*Trial*) liefert Lernbeispiele für die besuchten Zustände

Beispiel für 1. Sequenz:

$(1, 1) \xrightarrow{\frac{-1}{25}} (1, 2) \xrightarrow{\frac{-1}{25}} (1, 3) \xrightarrow{\frac{-1}{25}} (1, 2) \xrightarrow{\frac{-1}{25}} (1, 3) \xrightarrow{\frac{-1}{25}} (2, 3) \xrightarrow{\frac{-1}{25}} (3, 3) \xrightarrow{\frac{-1}{25}} (4, 3)_{+1}$

~ Rewardsummen:

- für (1,1): $7 \cdot (-0.04) + 1 = 1 - 0.28 = 0.72$
- für (1,2): $6 \cdot (-0.04) + 1 = 1 - 0.24 = 0.76$ und $4 \cdot (-0.04) + 1 = 1 - 0.16 = 0.84$
- für (1,3): $5 \cdot (-0.04) + 1 = 1 - 0.20 = 0.80$ und $3 \cdot (-0.04) + 1 = 1 - 0.12 = 0.88$

usw.

Naives Aktualisieren (2)

Beispiel für 1. Sequenz:

$$(1, 1) \xrightarrow{\frac{-1}{25}} (1, 2) \xrightarrow{\frac{-1}{25}} (1, 3) \xrightarrow{\frac{-1}{25}} (1, 2) \xrightarrow{\frac{-1}{25}} (1, 3) \xrightarrow{\frac{-1}{25}} (2, 3) \xrightarrow{\frac{-1}{25}} (3, 3) \xrightarrow{\frac{-1}{25}} (4, 3) +_1$$

↷ Rewardsummen $RS(s)$:

- für $s = (1,1)$: $7 \cdot (-0.04) + 1 = 1 - 0.28 = 0.72$
 - für $s = (1,2)$: $6 \cdot (-0.04) + 1 = 1 - 0.24 = 0.76$ und $4 \cdot (-0.04) + 1 = 1 - 0.16 = 0.84$
 - für $s = (1,3)$: $5 \cdot (-0.04) + 1 = 1 - 0.20 = 0.80$ und $3 \cdot (-0.04) + 1 = 1 - 0.12 = 0.88$
- usw.

Mit jedem Besuch eines Zustandes s wird der „Besuchszähler $N[s]$ (vergl. **function PASSIVE-RL-AGENT**, Folie 12) inkrementiert und der Nutzen $U^\pi(s)$ durch **kumulative Mittelung** mit der neu beobachteten **Rewardsumme $RS(s)$** aktualisiert:

$$U^\pi(s) \leftarrow \frac{U^\pi(s) \cdot (N[s] - 1) + RS}{N[s]} \quad \text{für } N[s] \geq 1 \text{ und Start mit } U^\pi(s) = 0 \text{ für alle } s$$

Im Grenzfall von unendlich vielen Sequenzbeobachtungen konvergiert $U^\pi(s)$ durch die kumulative Mittelung gegen

$$U^\pi(s) = E\left(\sum_{t=0}^{\infty} R(s_t) \mid \pi, s_0 = s\right).$$

Naives Aktualisieren (3)

Für den *Passive-RL-Agent* (Folie 12) realisiert die folgende Funktion *NAIVE UPDATE* den dortigen Aufruf $UPDATE(U, e, PERCEPTS, M, N)$ für das naive Aktualisieren.

NAIVEUPDATE braucht die Rewardsummen, die aber erst am Ende einer Trainingssequenz feststehen. Dann wurden in *function Passive-RL-Agent* die Zähler $N[STATE[e_i]]$ aber bereits auf die gesamte Zahl der Besuche von $STATE[e_i]$ hochgezählt. Daher wird für die Berechnung der laufenden Mittel in NAIVEUPDATE erneut gezählt über $N'[STATE[e_i]]$.

```
function NAIVEUPDATE ( $U, e, percepts, M, N$ ) return an updated  $U$ 
  if TERMINAL? $[e]$  then  $reward-to-go \leftarrow 0$ 
    for each  $e_i$  in  $percepts$  do  $N'[STATE[e_i]] \leftarrow 0$ 
    for each  $e_i$  in  $percepts$  (starting at end) do
       $N'[STATE[e_i]] \leftarrow N'[STATE[e_i]] + 1$ 
       $reward-to-go \leftarrow reward-to-go + REWARD[e_i]$ 
       $U[STATE[e_i]] = \text{RUNNING-AVERAGE} (U[STATE[e_i]], reward-to-go, N'[STATE[e_i]])$ 
```

```
function RUNNING-AVERAGE ( $U[s], reward-to-go, N[s]$  )
   $U[s] = [ U[s] * (N[s] - 1) + reward-to-go ] / N[s]$ 
```

Die *Rewardsummen* werden hier als *reward-to-go* bezeichnet und das naive Aktualisieren startet mit $U^{\pi}(s) = 0$ für alle Zustände s .

Naives Aktualisieren (4)

Hinweis zur Anwendung der Funktion *Naive Update*:

```
function NAIVEUPDATE ( $U, e, \text{percepts}, M, N$ ) return an updated  $U$ 
  if TERMINAL?[ $e$ ] then  $\text{reward-to-go} \leftarrow 0$ 
    for each  $e_i$  in  $\text{percepts}$  do  $N'[\text{STATE}[e_i]] \leftarrow 0$ 
    for each  $e_i$  in  $\text{percepts}$  (starting at end) do
       $N'[\text{STATE}[e_i]] \leftarrow N'[\text{STATE}[e_i]] + 1$ 
       $\text{reward-to-go} \leftarrow \text{reward-to-go} + \text{REWARD}[e_i]$ 
       $U[\text{STATE}[e_i]] = \text{RUNNING-AVERAGE} (U[\text{STATE}[e_i]], \text{reward-to-go}, N'[\text{STATE}[e_i]])$ 
```

```
function RUNNING-AVERAGE ( $U[s], \text{reward-to-go}, N[s]$  )
   $U[s] = [ U[s] * (N[s] - 1) + \text{reward-to-go} ] / N[s]$ 
```

Funktion *Naive Update* berechnet die Nutzen also „von hinten“ startend und bei mehrfach besuchten Zuständen überschreibend durch das laufende Mittel. Für Sequenz 1:

$$(1, 1) \frac{-1}{25} \mapsto (1, 2) \frac{-1}{25} \mapsto (1, 3) \frac{-1}{25} \mapsto (1, 2) \frac{-1}{25} \mapsto (1, 3) \frac{-1}{25} \mapsto (2, 3) \frac{-1}{25} \mapsto (3, 3) \frac{-1}{25} \mapsto (4, 3)_{+1}$$

→ $U(4,3) = +1$, $U(3,3) = 0.96$, $U(2,3) = 0.92$, $U(1,3) = 0.88$, $U(1,2) = 0.84$, $U(1,3) = 0.84$, $U(1,2) = 0.80$, $U(1,1) = 0.72$

Naives Aktualisieren (4)

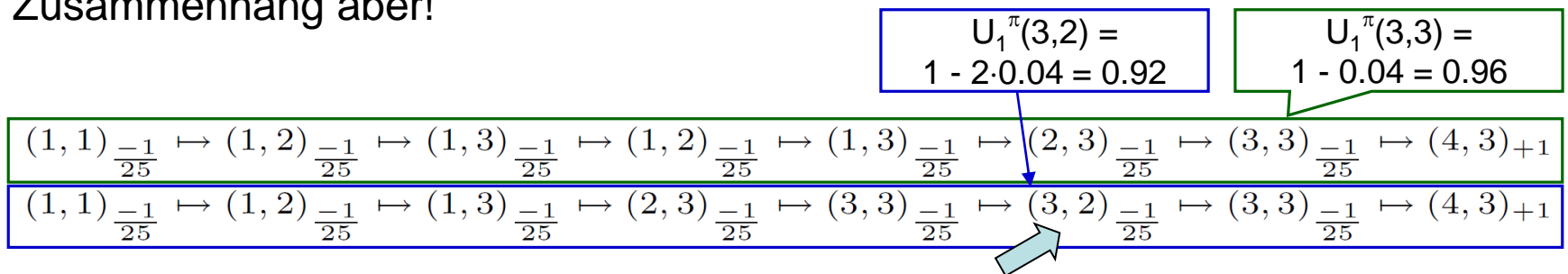
Beobachtung:

Das naive Aktualisierungsverfahren (NA) reduziert verstärkendes Lernen auf *überwachtes Lernen*, bei dem die Ein-/Ausgabepaare aus den beobachteten Zuständen und den ermittelten Rewardsummen $RS_t(s)$ bestehen.

Problem:

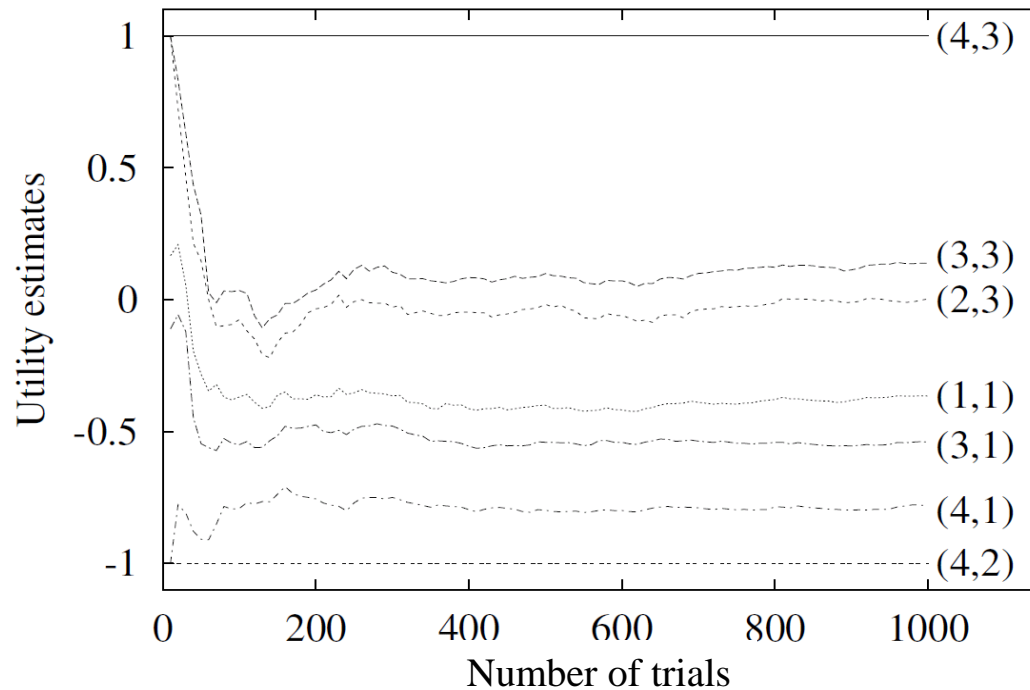
NA ignoriert den Einfluss der W'keit für bestimmte Zustandsübergänge!

Beispiel: In der 2. Sequenz wird Zustand (3,2) zum ersten Mal beobachtet und danach (3,3). Zustand (3,3) wurde bereits in der ersten Sequenz hoch bewertet, also könnte (3,2) auch eine hohe Bewertung erhalten. NA vernachlässigt diesen Zusammenhang aber!



Naives Aktualisieren (5)

~ NA sucht in einem unnötig großen Hypothesenraum für $U^\pi(s)$ und die Lernkurve konvergiert sehr langsam (mehr als 1000 Sequenzen sind nötig).




Der tatsächliche Nutzen eines Zustandes ist eben nach der Bellmann-Gleichung der **eigene Reward + gewichteter Nutzen seiner Nachfolgezustände**:

$$U^\pi(s) = R(s) + \sum_{s'} M_{ss'}^{\pi(s)} U^\pi(s').$$

Diesen Zusammenhang berücksichtigt NA nicht!

Adaptives Dynamisches Programmieren (1)

Um die Zusammenhänge zwischen den Zuständen nutzen zu können, kann der Agent das **Transitionsmodell** $M_{ss'}^{\pi(s)}$ lernen! 

Schrittweise ist das aktuell gelernte Transitionsmodell $M_{ss'}^{\pi(s)}$ dann mit den beobachteten Rewards zur Berechnung der Nutzenfunktion einzusetzen:

$$U^{\pi}(s) = R(s) + \sum_{s'} M_{ss'}^{\pi(s)} U^{\pi}(s').$$

Das Lernen des Transitionsmodells $M_{ss'}^{\pi(s)}$ ist möglich aufgrund der vollständig beobachtbaren Umwelt!

Adaptives Dynamisches Programmieren (2)

Das Lernen des Transitionsmodells $M_{ss'}^{\pi(s)}$ ist realisierbar durch eine Tabelle, die für alle s , $\pi(s)$ und s' Buch führt, wie oft der Zustand s' durch eine Ausführung der Aktion $\pi(s)$ im Zustand s in den Trainingssequenzen erreicht wird.

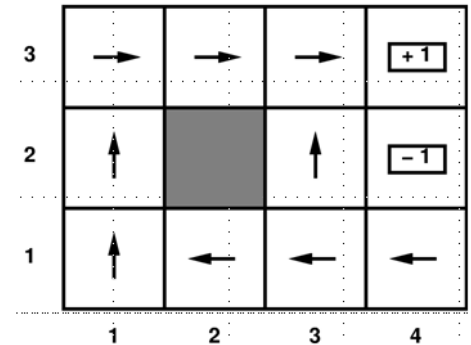
Beispiel:

$$\begin{array}{l}
 (1, 1) \xrightarrow{\frac{-1}{25}} (1, 2) \xrightarrow{\frac{-1}{25}} (1, 3) \xrightarrow{\frac{-1}{25}} (1, 2) \xrightarrow{\frac{-1}{25}} (1, 3) \xrightarrow{\frac{-1}{25}} (2, 3) \xrightarrow{\frac{-1}{25}} (3, 3) \xrightarrow{\frac{-1}{25}} (4, 3)_{+1} \\
 (1, 1) \xrightarrow{\frac{-1}{25}} (1, 2) \xrightarrow{\frac{-1}{25}} (1, 3) \xrightarrow{\frac{-1}{25}} (2, 3) \xrightarrow{\frac{-1}{25}} (3, 3) \xrightarrow{\frac{-1}{25}} (3, 2) \xrightarrow{\frac{-1}{25}} (3, 3) \xrightarrow{\frac{-1}{25}} (4, 3)_{+1} \\
 (1, 1) \xrightarrow{\frac{-1}{25}} (1, 2) \xrightarrow{\frac{-1}{25}} (1, 3) \xrightarrow{\frac{-1}{25}} (2, 3) \xrightarrow{\frac{-1}{25}} (3, 3) \xrightarrow{\frac{-1}{25}} (3, 2) \xrightarrow{\frac{-1}{25}} (4, 2)_{-1}
 \end{array}$$

In den drei Sequenzen wird Aktion *Right* vier Mal im Zustand (1,3) nach Strategie π ausgewählt und führt dabei drei Mal bzw. ein Mal zu den Folgezuständen (2,3) bzw. (1,2).

≈ Schätzung von $M_{(1,3),(2,3)}^{Right}$ zu 3/4

≈ Schätzung von $M_{(1,3),(1,2)}^{Right}$ zu 1/4



Adaptives Dynamisches Programmieren (3)

function PASSIVE-ADP-AGENT (*percept*) returns an action

inputs: *percept*, a **percept** (s', r') indicating the **current state** s' and reward r'

static: π , a fixed policy

mdp , an MDP with rewards R , model M , initially unknown

U , a table of utilities, initially empty

N_{sa} , a table of frequencies for state-action pairs, initially zero

$N_{sas'}$, a table of frequencies for state-action-state triples, initially zero

s, a , the **previous state and action**, initially null

if s' is new **then do** $U(s') \leftarrow r'; R(s') \leftarrow r'$

if s is not null **then do**

increment $N_{sa}[s, a]$ and $N_{sas'}[s, a, s']$

for each s'' such that $N_{sas'}[s, a, s'']$ is nonzero **do**

$M[s, a, s''] \leftarrow N_{sas'}[s, a, s''] / N_{sa}[s, a]$

$U \leftarrow \text{VALUE-DETERMINATION}(\pi, U, mdp)$

// plug observed R and current M into Bellmann equations

if **TERMINAL?** $[s']$ **then** $s, a \leftarrow \text{null}$ **else** $s, a \leftarrow s', \pi[s']$

return a

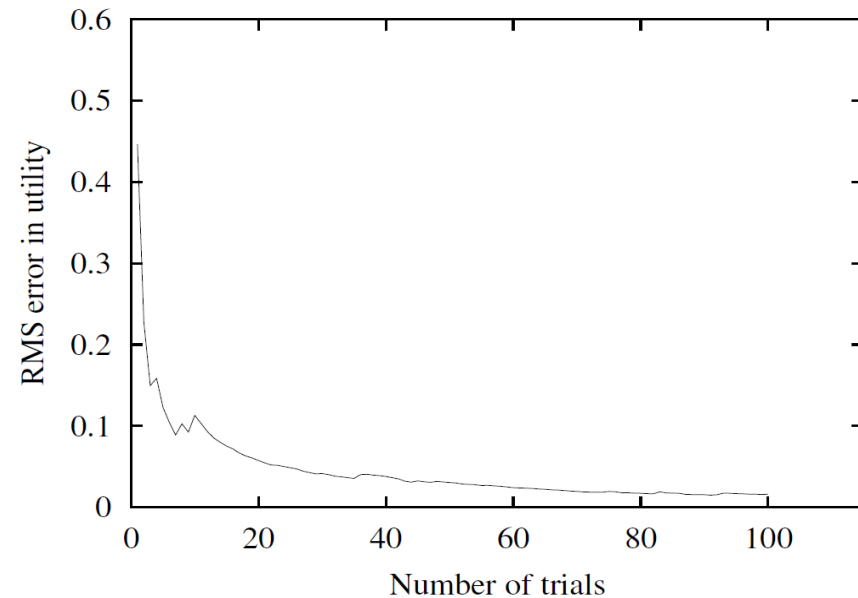
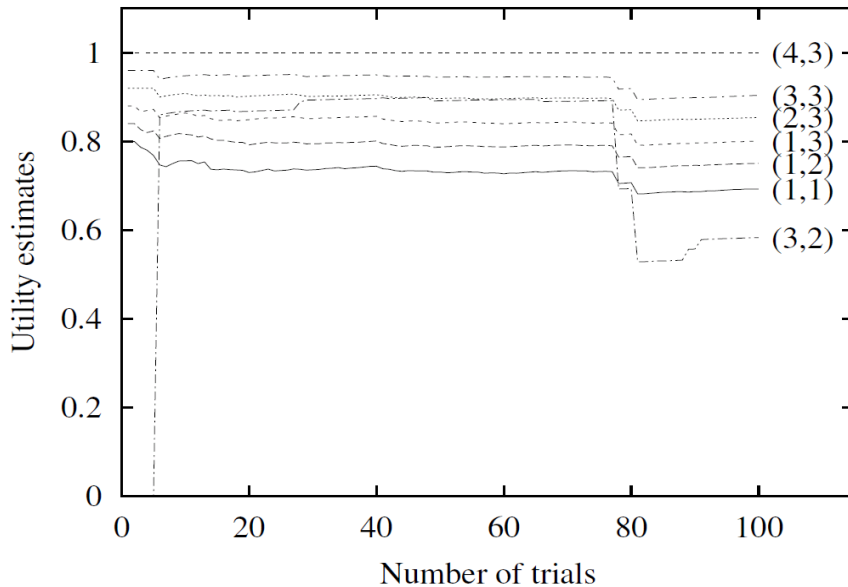
Ähnlich der Value-Iteration, aber mit
festgelegter Strategie π :
 $U(s) \leftarrow R(s) + \sum_{s'} M(s, \pi(s), s') U(s')$



Adaptives Dynamisches Programmieren (4)

Lernkurve für ADP beim 4x3-Beispiel:

- deutlich schnellere Konvergenz als naive Aktualisierung (vgl. Folie 20)



Einbruch bei Sequenz 78 wegen ersten Erreichens des -1-Zustands (4,2)



Adaptives Dynamisches Programmieren (5)

ADP reduziert verstärkendes Lernen ebenfalls auf ein überwachtes Lernen, bei dem die Ein-/Ausgabepaare gegeben sind durch Zustands-Aktions-Paare (s,a) als Eingaben und Folgezustände s' als Ausgaben.

ADP ist aufgrund seiner schnellen Konvergenz ein *Standard*, mit dem andere Reinforcement-Lernverfahren bewertet werden.

ADP ist aber *nicht handhabbar für große Zustandsräume*: Für Backgammon müssten 10^{50} Gleichungen in 10^{50} Unbekannten gelöst werden.

~ Ansatz des *Temporal Difference Learning*
ohne explizite Herleitung des Transitionsmodells.

Temporal Difference Lernen (1)

Frage: Wie können die *Zusammenhänge* zwischen den Zuständen *ohne* Lernen des expliziten Transitionsmodells berücksichtigt werden?

Lösung: Die Bewertung des Nutzens eines beobachteten Zustands s muss beim Lernen (1) die **beobachteten Transitionen** *und* (2) die **Nutzen der Folgezustände** berücksichtigen!

Beispiel (1. Sequenz):

$$(1, 1) \frac{-1}{25} \mapsto (1, 2) \frac{-1}{25} \mapsto \boxed{(1, 3)} \frac{-1}{25} \mapsto (1, 2) \frac{-1}{25} \mapsto \boxed{(1, 3)} \frac{-1}{25} \mapsto \boxed{(2, 3)} \frac{-1}{25} \mapsto (3, 3) \frac{-1}{25} \mapsto (4, 3)_{+1}$$

Als Ergebnis der ersten Sequenz werde **naiv aktualisiert**:

$$U^\pi(1,3) = \underline{0.84} \text{ (durch Mittelung aus 0.88 und 0.80) und } U^\pi(2,3) = 0.92.$$

Wegen der Möglichkeit, direkt von (1,3) zu (2,3) zu kommen, scheint eine höhere Bewertung – insbes. bei wiederholtem Auftreten dieser Transition – angemessen, nämlich:

$$U^\pi(1,3) = -0.04 + U^\pi(2, 3) = \underline{0.88}.$$

Temporal Difference Lernen (2)

Allgemein: Bei einem beobachteten Zustandsübergang von s nach s' wird der Wert von $U^\pi(s)$ an den Wert von $U^\pi(s')$ mit Lernrate α angepasst:

$$U_{t+1}^\pi(s) \leftarrow U_t^\pi(s) + \alpha \left(R(s) + U_t^\pi(s') - U_t^\pi(s) \right)$$



Reward $R(s)$

Nutzendifferenz zw. Nachfolgezustand s' und aktuellem Zustand s

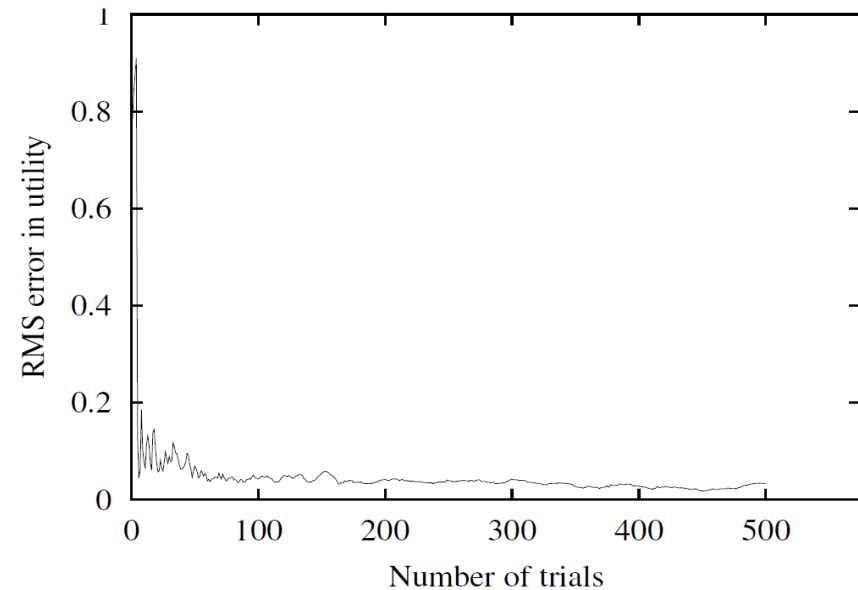
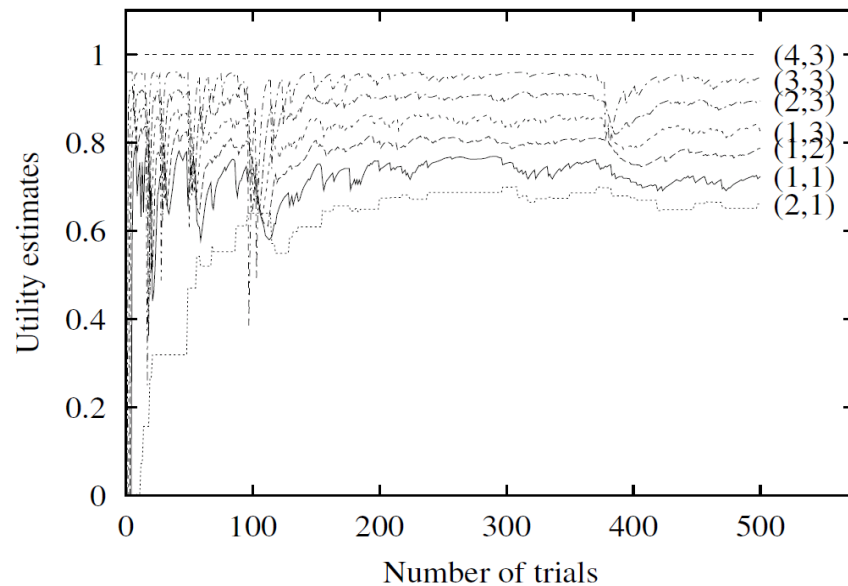
Da die Aktualisierung auf den Nutzendifferenzen *aufeinander folgender Zustände* basiert \leadsto *Temporal Difference Learning*.

Für die schnelle Konvergenz kann statt konstanter Lernrate α ein Funktional $\alpha(n)$ verwendet werden, dessen Wert mit zunehmender Anzahl n der Beobachtungen eines Zustands s abnimmt.

Temporal Difference Lernen (3)

Lernkurve für TDL beim 4x3-Beispiel:

- Langsamere Konvergenz mit stärkeren Ausreißern als ADP
- Schnellere Konvergenz als naives Aktualisieren
- Verwendete Lernrate $\alpha(n) = 60/(59 + n)^*$



* Allg. Forderung: $\sum_{n=1}^{\infty} \alpha(n) = \infty, \sum_{n=1}^{\infty} \alpha^2(n) < \infty$. Beispiel : $\alpha(n) = 1/n$

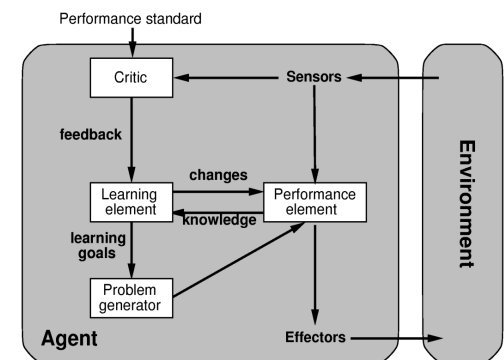
Aktives Lernen in einer unbekannten Umgebung

- Beim **passiven Lernen** hatten wir angenommen, dass der Agent keine aktive Rolle in der Wahl der Aktionen hat \leadsto **durch Umwelt vorgegebene Policy π** .
- Beim **aktiven Lernen** muss jetzt ein **Modell $M_{ss'}^a$** für alle möglichen Aktionen a des Agenten unabhängig von einer festen Policy berücksichtigt werden. Dazu bietet sich **ADP** als Verfahren an.

Die zu lernenden Nutzen der Zustände sind durch die *optimale Strategie* entsprechend den Bellmann-Gleichungen bestimmt :

$$U(s) = R(s) + \max_a \sum_{s'} M_{ss'}^a U(s'), \quad \pi^*(s) = \operatorname{argmax}_a \sum_{s'} M_{ss'}^a U(s').$$

Der resultierende *Active-ADP-Agent* verwendet sein **Performance Element**, um ein Aktionen auszuwählen, wobei die Exploration die Funktion des **Problemgenerators** umsetzt.



Exploration (1)

Gibt es eine optimale Erkundungsstrategie?

Ein **naiver Agent** wählt zufällig Aktionen aus, in der Hoffnung, so die Umgebung vollständig zu erkunden.

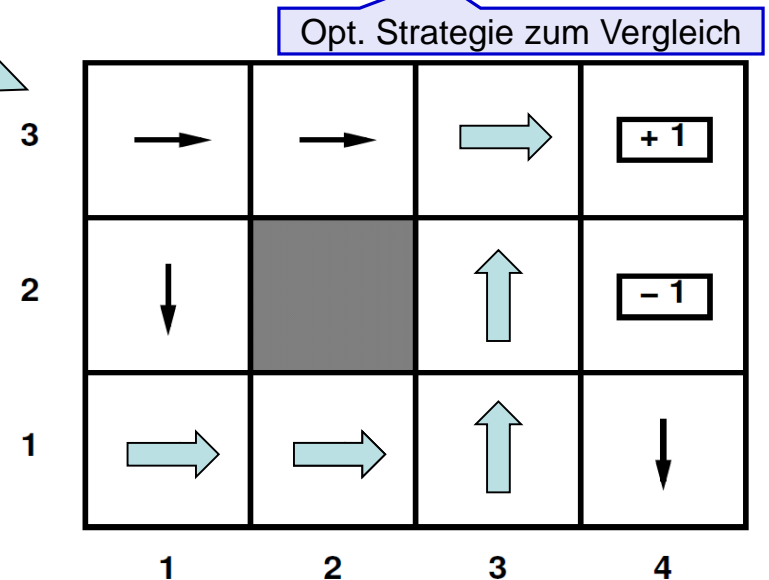
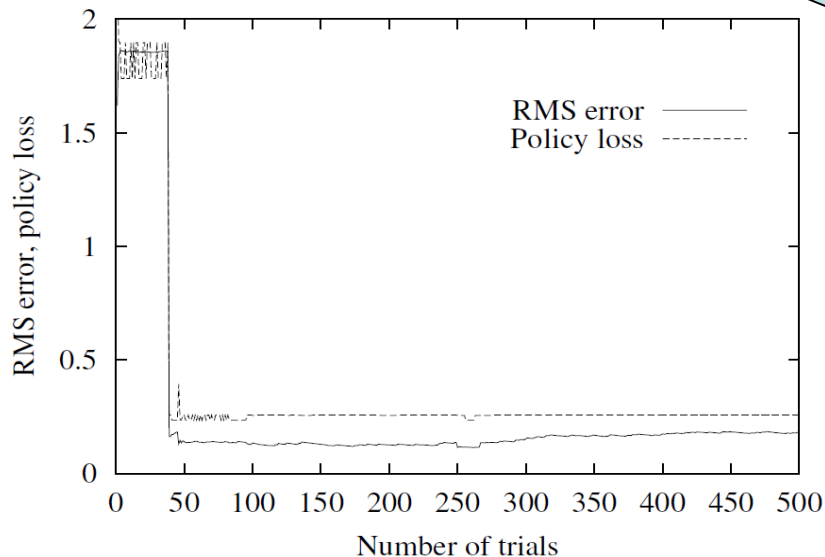
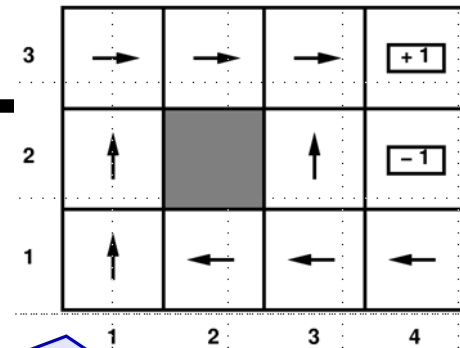
~ ein naiver Agent findet eher zufällig eine Lösung

~ ein naiver Agent optimiert seinen Nutzen nicht zielgerichtet

Alternative: der **gierige Agent** maximiert seinen gegenwärtigen Nutzen auf Basis der aktuellen Schätzungen!

Exploration (2)

Bspl.: Der „gierige“ Agent finde in der 39. Sequenz eine Strategie, die über (1,1), (2,1), (3,1), (3, 2) und (3, 3) zum Feld (4, 3) mit +1-Reward gelangt.



Nach einigen kleineren Variationen bleibt der gierige Agent ab Sequenz 276 bei dieser Strategie und findet nicht die optimale Strategie (s.o.)

~ hat der gierige Agent einmal eine Lösung gefunden, wird er nicht besser und finden daher nicht immer die optimale Lösung!

Lösung: *Unbekannten* Zustand-Aktions-Paaren höhere Nutzen einräumen!

Exploration (3)

Exploration, die **unbekannten Zustands-Aktions-Paaren** höhere Nutzen einräumt:

- $N(a,s)$: Wie oft wurde Aktion a in Zustand s ausgeführt.
- $U^+(s)$: Erwartete Utility von s ... unter Berücksichtigung von $N(a,s)$.

Die Aktualisierungsgleichung für die Value-Iteration des aktiven ADP-Agenten:

$$U^+(s) \leftarrow R(s) + \max_a \underbrace{f\left(\sum_{s'} M_{ss'}^a U^+(s'), \underbrace{N(a,s)}\right)}_{\text{Exploration}}$$

$f(u,n)$ heißt **Explorationfunktion** und bestimmt, in welchem Verhältnis Neugier und Nutzenmaximierung stehen. $f(u,n)$ sollte mit zunehmendem u und abnehmendem n wachsen. Beispiel:

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

R^+ ist eine fixe, optimistische Reward-Schätzung und führt mit der Konstanten N_e dazu, dass jedes Zustands-Aktions-Paar mindestens N_e Mal gewählt wird.

Aktiver ADP-Agent

function ACTIVE-ADP-AGENT (*percept*) returns an action

inputs: *percept* , a percept (s', r') indicating the **current state s' and reward r'**

static: *mdp*, an MDP with rewards R , model M , initially unknown

U , a table of utilities, initially empty

N_{sa} , a table of frequencies for state-action pairs, initially zero

$N_{sas'}$, a table of frequencies for state-action-state triples, initially zero

s, a , the previous state and action, initially null

if s' is new **then do** $U(s') \leftarrow r'$; $R(s') \leftarrow r'$

if s is not null **then do**

 increment $N_{sa}[s, a]$ and $N_{sas'}[s, a, s']$

for each t such that $N_{sas'}[s, a, t]$ is nonzero **do**

$M[s, a, t] \leftarrow N_{sas'}[s, a, t] / N_{sa}[s, a]$

$U \leftarrow \text{VALUE-ITERATION}_{\text{Explore}}(U, mdp)$

// plug observed R and current M into Bellmann equations for exploration

if TERMINAL? $[s']$ **then** $s, a \leftarrow \text{null}$

else $s, a \leftarrow s', \text{argmax}_a f(\sum_s M[s, a', s'] U(s'), N_{sa}[s', a'])$

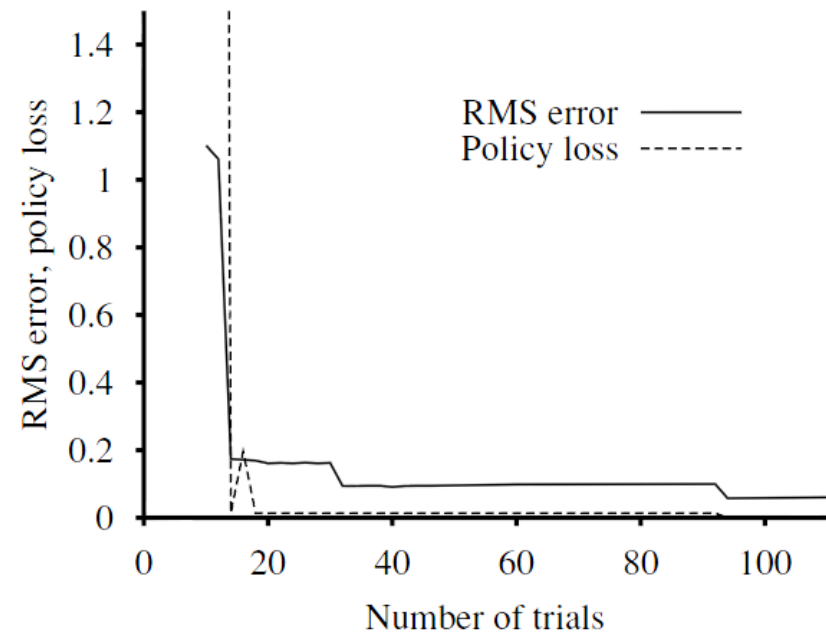
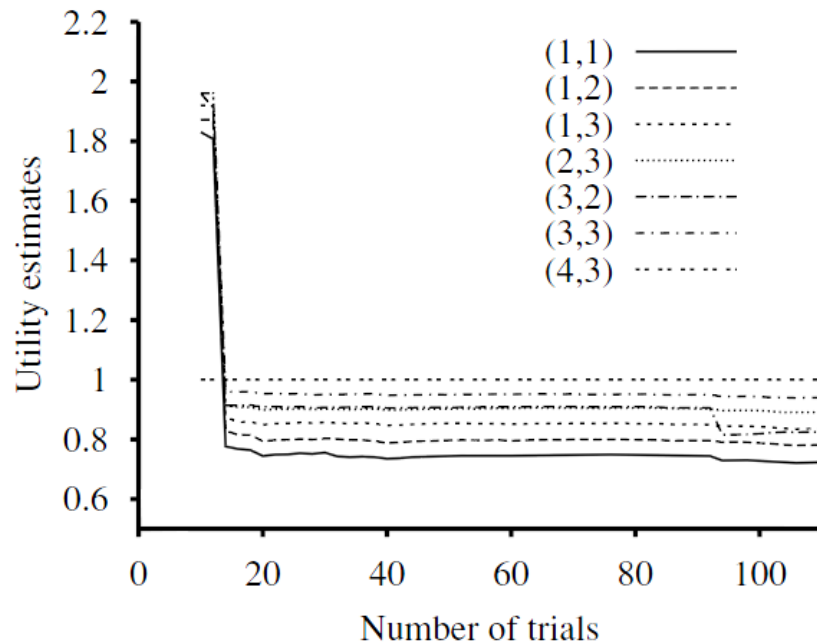
return a



Exploration (4)

Lernkurve für explorierenden ADP-Agenten beim 4x3-Beispiel:

- Fast optimale Strategie wird bereits nach 18 Sequenzen gefunden.
- $R^+ = 2$, $N_e = 5$



Q-Lernen (1)

Analog zum explorierenden ADP-Agenten soll ein **explorierender TDL-Agent** gestaltet werden, der ohne Lernen des Transitionsmodells arbeiten kann.

Der **passive TDL-Agent** berücksichtigt die beobachteten Transitionen **und** die Nutzen der Folgezustände beim Lernen der **Nutzenfunktion** $U^\pi(s)$.

$$U_{t+1}^\pi(s) \leftarrow U_t^\pi(s) + \alpha(R(s) + U_t^\pi(s') - U_t^\pi(s))$$

Reward $R(s)$

Nutzendifferenz zw. Nachfolgezustand s'
und aktuellem Zustand s

Der **aktive TDL-Agent** muss die zu wählenden Transitionen und die Nutzen der Folgezustände berücksichtigen. Daher ist ein Lernen der Bewertung von **Zustands-Aktions-Paaren** $Q(a,s)$ mit einer **Q-Funktion** umsetzen.

$Q(a,s)$ bewertet
die Ausführung
von Aktion a in
Zustand s .

Q-Lernen (2)

Zusammenhang von Nutzenfunktion und Zustands-Aktions-Paaren:

$$U(s) = \max_a Q(a,s).$$

Vorteil: Ein TDL-Agent, der eine Q-Funktion lernt,
benötigt keine Modelle für die Aktionsauswahl:

- Q-Werte dienen direkt der Entscheidungsfindung
- Q-Werte werden direkt aus dem Reward-Feedback gelernt

Q-Lernen (3)

Q-Lernen mit ADP-Ansatz würde bedeuten:

$$U(s') = \max_{a'} Q(a', s')$$

$$Q(a, s) = R(s) + \sum_{s'} M_{ss'}^a \max_{a'} Q(a', s').$$

Q-Lernen mit TDL-Ansatz benötigt kein explizites Modell $M_{ss'}^a$:

$$Q_{t+1}(a, s) \leftarrow Q_t(a, s) + \alpha \left(R(s) + \max_{a'} Q_t(a', s') - Q_t(a, s) \right)$$

Zum Vergleich die TDL-Aktualisierung:
 $U_{t+1}(s) \leftarrow U_t(s) + \alpha (R(s) + U_t(s') - U_t(s))$

Reward $R(s)$

Bewertungsdifferenz zw. Nachfolgezustand s' und aktuellem s

Der **explorierende Q-lernende TDL-Agent** benutzt wieder eine **Explorationsfunktion** $f(u, n)$ wie der explorierende ADP-Agent.

Q-Lernen (4)

```
function Q-LEARNING-AGENT (percept) returns an action
inputs: percept , a percept ( $s', r'$ ) indicating the current state  $s'$  and reward  $r'$ 
static:  $Q$ , a table of action values indexed by state and action, initially zero
           $N_{sa}$ , a table of frequencies for state-action pairs, initially zero
           $s, a, r$  the previous state, action, and reward, initially null

if  $s$  is not null then do
    increment  $N_{sa}[s, a]$ 
     $Q[a, s] \leftarrow Q[a, s] + \alpha (r + \max_a Q[a', s'] - Q[a, s])$ 
if  $\text{TERMINAL?}[s']$  then  $s, a, r \leftarrow \text{null}$ 
    else  $s, a, r \leftarrow s', \text{argmax}_a f(Q[a', s'], N_{sa}[s', a']), r'$ 

return  $a$ 
```

Der Q-lernende TDL-Agent lernt ohne Modell ... aber auch deutlich langsamer als der explorierende ADP-Agent.

Eingabe-Generalisierung

Die Zustandsräume von Schach und Backgammon haben ca. 10^{120} bzw. 10^{50} Zustände. Es ist absurd zu verlangen, dass man all diese Zustände besuchen muss, um spielen zu lernen.

Wir benötigen eine *implizite Repräsentation* der Nutzenfunktion. Zum Beispiel lässt sich jede Position im Schach anhand weniger Merkmale bewerten.

↪ ca. 10 Merkmale anstelle von 10^{120} Zuständen.

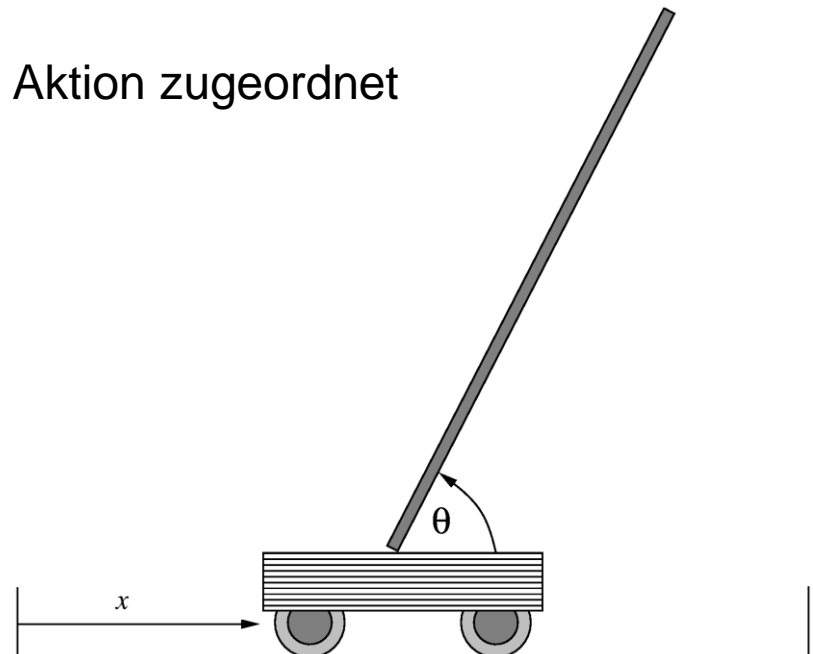
↪ z.B. gewichtete lineare Nutzenfunktion $U(s) = w_1 f_1(s) + \dots + w_{10} f_{10}(s)$

Eingabe-Generalisierung impliziert auch für die Exploration eine Generalisierung bzgl. besuchter Zustände und nicht besuchter Zustände.

Klassische Anwendungen

- Im Spielbereich: Checkers, TD-Gammon
 - ~ Eingabe-Generalisierung
- Im Robotikbereich: Invertiertes Pendel
 - mit Zustandsvariablen x , θ , x' , θ'
 - Versuchssequenzen, die mit Umfallen der Stange oder Spurende terminieren
 - Negative Verstärkung wurde der letzten Aktion zugeordnet und durch die Folge zurückgereicht
 - BOXES-Algm.* konnte Stange nach 30 Versuchen 1h im Gleichgewicht halten

* D. Michie, R. A. Chambers: BOXES: *An experiment in adaptive control*. In Dale, E., & Michie, D. (Eds.), *Machine Intelligence* (1968)



Erfolgreiche Anwendungen

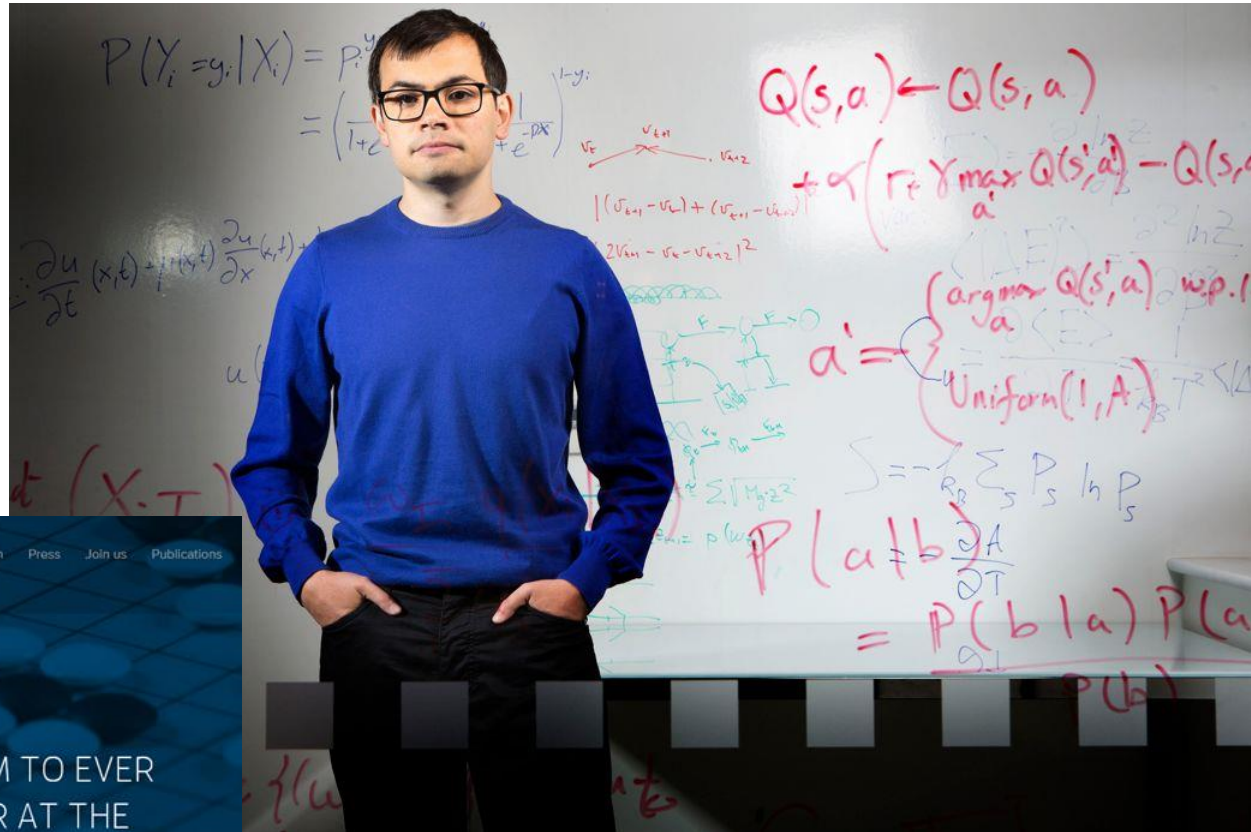
Demis Hassabis: Gründer von *DeepMind*, das er 2014 an Google verkaufte.

→ *Deep Reinforcement Learning*

schlägt weltbesten

Go-Spieler Lee Sedol

im März 2016



. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis. *Nature* (October 2017)

Zusammenfassung

Je nach der Rolle des Agenten unterscheiden wir **aktives** (exploratives Handeln) und **passives Lernen** (Beobachten der Umwelt).

Der **Nutzen** eines Zustands ist die Summe der **Belohnungen**, die ein Agent zwischen seinem aktuellen Zustand und dem Endzustand erwartet.

Drei Ansätze, um die Nutzenfunktion zu lernen:

1. Naives Aktualisieren.
2. Adaptive Dynamische Programmierung.
3. Zeitliche Differenz.

Explorative Agenten müssen alle möglichen Aktionen berücksichtigen und verwenden eine Explorationsfunktion.