Reinforcement Learning

VL Algorithmisches Lernen, Teil 3e

Jianwei Zhang

University of Hamburg
MIN Faculty, Dept. of Informatics
Vogt-Kölln-Str. 30, D-22527 Hamburg
zhang@informatik.uni-hamburg.de

15/07/2008

Reinforcement Learning

VL Algorithmisches Lernen, Teil 3e

Jianwei Zhang

University of Hamburg
MIN Faculty, Dept. of Informatics
Vogt-Kölln-Str. 30, D-22527 Hamburg
zhang@informatik.uni-hamburg.de

15/07/2008

Terminübersicht: Part 3

- 17/06/2009 Dimensionsproblem, PCA
- 18/06/2009 Support-Vektor Maschinen
- 24/06/2009 Support-Vektor Maschinen (4st.)
- 25/06/2009 Übung
- 01/07/2009 Funktionsapproximation, Fuzzy-Logik (4st.)
- 02/07/2009 Übung (Prof. Dr. Menzel)
- 08/07/2009 Verstärkungslernen (1)
- 09/07/2009 Verstärkungslernen (2)
- 15/07/2009 **Verstärkungslernen (3)**
- 16/07/2009 **Anwendungen in der Robotik**

Gliederung

Introduction
Dynamic Programming
Example: Gridworld
Asynchrones DP
Q-Lernen
Beschleunigung des Lernens
Bahnplanung mit DP
Grasping mit RL: 2-Backen-Greifer
Grasping mit RL: Barret-Hand
- Automatischer Value Cutoff
- Automatischer Value Cutoff
- Evaluation
- Experimente mit TASER

Dynamic Programming

Ziele dieses Abschnitts:

- Überblick über eine Sammlung klassischer Lösungsmethoden für MDPs, die *dynamic programming* genannt werden
- Demonstration des Einsatzes von DP beim Errechnen von Wertefunktionen und somit von optimalen *Policies*
- Diskussion der Effektivität und des Nutzens von DP

Dieser Teil ist aus "Reinforcement Learning: An Introduction", Richard S. Sutton and Andrew G. Barto

http://www.cs.ualberta.ca/~sutton/book/ebook/index.html

Dynamische Programmierung für Modellbasiertes Lernen

Wenn die Dynamik der Umwelt bekannt ist, kann die Klasse der Dynamischen Programmierung verwendet werden.
Bei der Dynamischen Programmierung handelt es sich um eine Sammlung von Ansätzen, bei der ein perfektes Modell des MDP's vorhanden ist. Um die optimale Policy zu berechen, werden die Bellmann Gleichungen in Update-Regeln eingebettet, die dann die gewünschte Werte-Funktion (V) approximieren.
Drei Schritte:

1. Policy Evaluation
2. Policy Improvement
3. Policy Iteration

Policy Evaluierung - I

Policy Evaluierung: Errechne die Zustands-Wertefunktion V^π für eine gegebene *Policy* π.
Erinnern wir uns:
Zustands-Wertefunktion für Policy π:

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

Bellman-Gleichung für V^π:

$$V^\pi(s) = \sum_a \pi(s,a)\sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V^\pi(s')]$$

– ein System von $|S|$ gleichzeitigen linearen Gleichungen

Policy Evaluation - II

Unter *Policy Evaluation* versteht man das Problem, aus einer willkürlichen festen Policy π die Werte-Funktion V^π zu berechnen. Ausgehend von der Bellmann-Gleichung kann man eine Update-Regel entwerfen, in der approximierte Werte-Funktionen $V_0, V_1, V_2, V_3, \ldots$ berechnet werden:

$$V_{k+1}(s) = \sum_a \pi(s,a) \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V_k(s')] \qquad \forall s \in S$$

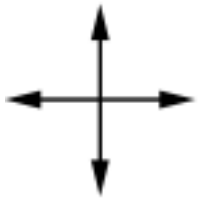
Es wird ausgehend von der k-ten Näherung V_k für jeden Zustand s sukzessiv die $k+1$-te Näherung V_{k+1} berechnet, mit anderen Worten wird der alte Funktionswert von s durch den neuen Wert ersetzt, der mit den alten Werten gemäss der Iterationsregel berechnet wird.

Iterative Policy Evaluierung

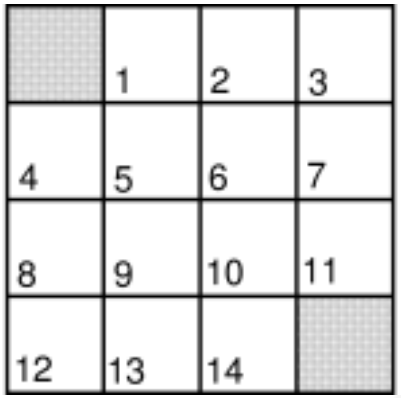
`Input` π, die zu evaluierende *Policy*
`Initialisiere` $V(s) = 0$, für alle $s \in S^+$
`Wiederhole`
  $\Delta \leftarrow 0$
  Für jedes $s \in S$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \sum_a \pi(s,a) \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V_k(s')]$
    $\Delta <\leftarrow \max(\Delta, |v - V(s)|)$

bis $\Delta < \theta$ (eine kleine positive Zahl)
Output $V \approx V^\pi$

Eine kleine *Gridworld*



actions



$r = -1$
on all transitions

- Eine episodische Aufgabe (*undiscounted*)
- Nicht-terminale Zustände: $1, 2, \ldots, 14$;
- Ein terminaler Zustand (zweimal als schattiertes Quadrat dargestellt)
- Aktionen, die den Agenten aus dem *Grid* nehmen würden, lasssen den Zustand unverändert
- Der *Reward* ist - 1 bis der terminale Zustand erreicht ist

Iterative Policy-Evaluierung für die kleine Gridworld - I

	V_k for the Random Policy	Greedy Policy w.r.t. V_k
$k = 0$	0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0	↔↕ ↔↕ ↔↕ ↔↕ ↔↕ ↔↕ ↔↕ ↔↕ ↔↕ ↔↕ ↔↕ ↔↕ ↔↕ ↔↕ ← random policy
$k = 1$	0.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 0.0	← ↔↕ ↔↕ ↑ ↔↕ ↔↕ ↔↕ ↔↕ ↔↕ ↔↕ ↓ ↔↕ ↔↕ →
$k = 2$	0.0 -1.7 -2.0 -2.0 -1.7 -2.0 -2.0 -2.0 -2.0 -2.0 -2.0 -1.7 -2.0 -2.0 -1.7 0.0	← ← ↔↕ ↑ ↔ ↔↕ ↓ ↑ ↔↕ ↱ ↓ ↔↕ → →

links: $V_k(s)$ für die *random* Policy π (zufällige Züge)

rechts: Züge der greedy Policy auf $V_k(s)$

Iterative Policy-Evaluierung für die kleine Gridworld - II

$k = 3$

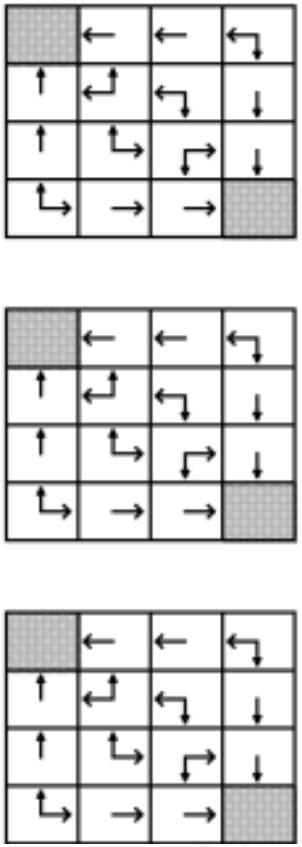
0.0	-2.4	-2.9	-3.0
-2.4	-2.9	-3.0	-2.9
-2.9	-3.0	-2.9	-2.4
-3.0	-2.9	-2.4	0.0

$k = 10$

0.0	-6.1	-8.4	-9.0
-6.1	-7.7	-8.4	-8.4
-8.4	-8.4	-7.7	-6.1
-9.0	-8.4	-6.1	0.0

$k = \infty$

0.0	-14.	-20.	-22.
-14.	-18.	-20.	-20.
-20.	-20.	-18.	-14.
-22.	-20.	-14.	0.0



optimal policy

In diesem Beispiel: die greedy Policy für $V_k(s)$ ist für $k \geq 3$ optimal

Policy Verbesserung

Man betrachtet jetzt die Aktions-Werte-Funktion $Q^\pi(s,a)$, bei der in Zustand s Aktion a ausgewählt wird, und dann die Policy π verfolgt wird:

$$Q^\pi(s,a) = \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V^\pi(s')]$$

Man sucht in jedem Zustand die Aktion, die die Aktions-Werte-Funktion maximiert. Somit wird eine greedy Policy π' für eine gegebene Werte-Funktion V^π generiert:

$$\begin{aligned}\pi'(s) &= \arg\max_a Q^\pi(s,a) \\ &= \arg\max_a \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V^\pi(s')]\end{aligned}$$

Policy Verbesserung

Angenommen, wir haben V^π für eine deterministische *Policy* π berechnet.
Wäre es besser für einen gegebenen Zustand s eine Aktion $a \neq \pi(s)$ durchzuführen?
Der Wert, wenn a im Zustand s durchgeführt wird, ist:

$$\begin{aligned} Q^\pi(s,a) &= E_\pi\{r_{r+1} + \gamma V^\pi(s_{t+1}) | s_t = s, a_t = a\} \\ &= \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V^\pi(s')] \end{aligned}$$

Es ist genau dann besser, zur Aktion a für den Zustand s zu wechseln, wenn

$$Q^\pi(s,a) > V^\pi(s)$$

Policy Verbesserung Forts.

Führe dies für alle Zustände durch, um eine neue *Policy*π zu bekommen, die in Bezug auf V^π *greedy* ist:

$$\begin{aligned} \pi^{'}(s) &= \operatorname*{argmax}_a Q^\pi(s,a) \\ &= \operatorname*{argmax}_a \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V^\pi(s')] \end{aligned}$$

Dann $V^{\pi'} \geq V^\pi$

Policy Verbesserung Forts.

Was wenn $V^{\pi'} = V^{\pi}$?
Z.B. für alle $s \in S$, $\quad V^{\pi'}(s) = \max_a \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V^{\pi}(s')]$?

Merke: dies ist die optimale Bellman-Gleichung.

Also sind $V^{\pi'} = V^*$ und sowohl π als auch π' optimale Policies.

Iterative Methoden

$$V_0 \rightarrow V_1 \rightarrow \ldots \rightarrow V_k \rightarrow V_{k+1} \rightarrow \ldots \rightarrow V^\pi$$

$\Uparrow$ ein "Durchlauf"

Ein Durchlauf besteht darin, auf jeden Zustand eine *Backup*-Operation anzuwenden.

Ein *full-policy* Evaluierungs-Backup:

$$V_{k+1}(s) \leftarrow \sum_a \pi(s,a) \sum_{s'} P^a_{ss'} [R^a_{ss'} + \gamma V_k(s')]$$

Policy Iteration - I

Verwendet man abwechselnd Policy Improvement und Policy Evaluation, so heisst das, dass mit einer festen Werte-Funktion V^{π} die Policy π verbessert wird und dann für die bessere Policy π' die dazugehörige Werte-Funktion $V^{\pi'}$ berechnet wird.
Man wendet wieder Policy Improvement an, um eine noch bessere Policy π'' zu bekommen, und so weiter ...:

$$\pi_0 \xrightarrow{\text{PE}} V^{\pi_0} \xrightarrow{\text{PI}} \pi_1 \xrightarrow{\text{PE}} V^{\pi_1} \xrightarrow{\text{PI}} \pi_2 \xrightarrow{\text{PE}} V^{\pi_2} \cdots \xrightarrow{\text{PI}} \pi^* \xrightarrow{\text{PE}} V^*$$

Hierbei steht $\xrightarrow{\text{PI}}$ für das Anwenden des Policy Improvement und analog $\xrightarrow{\text{PE}}$ für Policy Evaluation.

Policy Iteration

$$\pi_0 \longrightarrow V^{\pi_0} \longrightarrow \pi_1 \longrightarrow V^{\pi_1} \longrightarrow \ldots \pi^* \longrightarrow V^* \longrightarrow \pi^*$$

Policy-Evaluierung $\Uparrow$ $\Uparrow$ Policy-Verbesserung "Greedification"

Policy Iteration

1. Initialisierung
   $V(s) \in \Re$ und $\pi(s) \in A(s)$ beliebig für alle $s \in S$

2. Policy-Evaluierung
   Wiederhole
   $\Delta \leftarrow 0$
   `Für jedes` $s \in S$:
   $v \leftarrow V(s)$
   $V(s) \leftarrow \sum_{s'} P_{ss'}^{\pi(s)} [R_{ss'}^{\pi(s)} + \gamma V(s')]$
   $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   bis $\Delta < \theta$ (eine kleine positive Zahl)

Policy Iteration Forts.

3. Policy-Verbesserung

Policy-stable $\leftarrow$ wahr
Für jedes $s \in S$:
$b \leftarrow \pi(s)$
$\pi(s) \leftarrow \arg\max_a \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V(s')]$
Wenn $b \neq \pi(s)$, dann *Policy-stable* $\leftarrow$ falsch
Wenn *Policy-stable*, dann aufhören; ansonsten gehe zu 2

Wert Iteration

Erinnern wir uns an das **full policy Evaluierungs-Backup**

$$V_{k+1}(s) \leftarrow \sum_a \pi(s,a) \sum_{s'} P^a_{ss'} [R^a_{ss'} + \gamma V_k(s')]$$

Hier ist das **full value Iterations-Backup**:

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} P^a_{ss'} [R^a_{ss'} + \gamma V_k(s')]$$

Wert Iteration Forts.

Initialisiere V beliebig, z.B. $V(s) = 0$, für alle $s \in S^+$
Wiederhole
$\Delta \leftarrow 0$
`Für jedes` $s \in S$:
$v \leftarrow V(s)$
$V(s) \leftarrow \max_a \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V(s')]$
$\Delta \leftarrow \max(\Delta, |v - V(s)|)$
bis $\Delta < \theta$ (eine kleine positive Zahl)

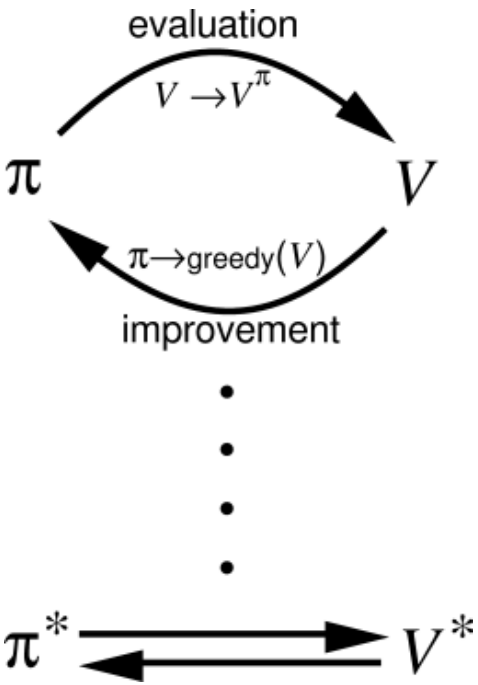
Output ist eine deterministische Policy π, so dass
$\pi(s) = \arg\max_a \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V(s')]$

Asynchrones DP

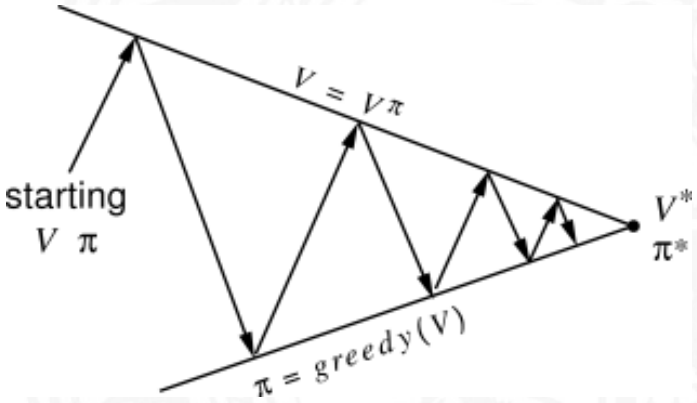
- Alle DP Methoden, die bisher beschrieben wurden, erfordern vollständige Durchläufe des gesamten Zustands-Sets
- Das asynchrone DP benutzt keine Durchläufe. Stattdessen funktioniert es so:
  - Wiederhole bis das Konvergenzkriterium eingehalten wird:
    - Suche einen Zustand zufällig heraus und wende das passende *Backup* an
- Benötigt immer noch viel Berechnung, aber hängt nicht in hoffnungslos langen Durchläufen fest
- Kann man Zustände für die Anwendung des *Backup* intelligent auswählen? JA: die Erfahrung eines Agenten kann als Führer dienen

Generalized Policy Iteration (GPI)

Generalized Policy Iteration (GPI):
jede Interaktion zwischen Policy-Evaluierung und Policy-Verbesserung, unabhängig von ihrer Granularität.



Eine geometrische Metapher für die Konvergenz von GPI:



Effizienz des DP

- Eine optimale *Policy* zu finden ist polynomisch mit der Zahl der Zustände...
- ABER, die Zahl der Zustände ist oft astronomisch, z.B. wächt sie oft exponentiell mit der Zahl der Zustands-Variablen (was Bellman "den Fluch der Dimensionalität" nannte)
- In der Praxis kann das klassische DP auf Probleme mit ein paar Millionen Zuständen angewandt werden
- Das asynchrone DP kann auf größere Probleme angewandt werden und eignet sich für parallele Berechnung
- Es ist überraschend leicht, MDPs zu finden, für die DP Methoden unpraktisch sind

Zusammenfassung: Dynamisches Programming

- Policy Evaluierung: *Backups* ohne Maximum
- Policy Verbesserung: bilde eine *greedy Policy*, wenn auch nur lokal
- Policy Iteration: wechsle die beiden obigen Prozesse ab
- Wert Iteration: *Backups* mit Maximum
- Vollständige *Backups* (die später Beispiel-Backups gegenübergestellt werden)
- Generalized Iteration (GPI)
- Asynchrones DP: eine Methode, um vollständige Durchläufe zu vermeiden
- *Bootstrapping*: Schätzungen durch andere Schätzungen auf den neuesten Stand bringen

Q-Lernen

- Q-Funktion
- Q-Lernalgorithmus
- Konvergenz
- Beispiel: GridWorld
- Experience-Replay

Die Q-Funktion

Man definiert die Q-Funktion wie folgt:

$$Q(s,a) \equiv r(s,a) + \gamma V^*(\delta(s,a))$$

Damit lässt sich π^* schreiben als

$$\pi^*(s) = arg \max_a Q(s,a)$$

D.h.: Die optimale Policy kann erlernt werden, indem Q gelernt wird, auch wenn r und δ nicht bekannt sind.

Q-Lernalgorithmus - I

Q und V^* stehen in enger Beziehung zueinander:

$$V^*(s) = \max_{a'} Q(s, a')$$

Damit lässt sich die Definition von $Q(s, a)$ umschreiben:

$$Q(s, a) \equiv r(s, a) + \gamma \max_{a'} Q(\delta(s, a), a')$$

Diese rekursive Definition von Q bildet die Basis für einen Algorithmus, der Q iterativ approximiert.

Q-Lernalgorithmus - II

Im folgenden sei $\hat{Q}$ der aktuelle Schätzwert für Q. s' sei der neue Zustand nach Ausführung der gewählten Handlung und r sei der dabei erzielte Reward.
Dann ergibt sich aus der rekursiven Definition von Q die Iterationsvorschrift wie folgt:

$$Q(s,a) \equiv r(s,a) + \gamma \max_{a'} Q(\delta(s,a), a')$$

$\Rightarrow$:

$$\hat{Q}(s,a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s', a')$$

Q-Lernalgorithmus - III

Damit lautet der Algorithmus:

1. Initialisierte alle Tabelleneinträge von $\hat{Q}$ zu 0.
2. Ermittle aktuellen Zustand s.
3. Loop
   - Wähle Handlung a und führe sie aus.
   - Erhalte Reward r.
   - Ermittle neuen Zustand s'.
   - $\hat{Q}(s, a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s', a')$
   - $s \leftarrow s'$.

   Endloop

Konvergenz des Q-Lernens

Satz: Es seien folgende Bedingungen erfüllt:

- $|r(s,a)| < \infty, \forall s, a$
- $0 \leq \gamma < 1$
- jedes (s,a)-Paar wird unendlich oft besucht

Dann konvergiert $\hat{Q}$ gegen die richtige Q-Funktion.

Kontinuierliche Systeme

Die Q-Funktion sehr großer oder kontinuierlicher Zustandsräume lässt sich nicht durch eine explizite Tabelle darstellen.
Man verwendet stattdessen einen Funktionsapproximations-Algorithmus, z.B. ein Neuronales Netz oder B-Splines.
Die Ausgaben des Q-Learning-Algorithmus dienen dem Neuronalen Netz als Trainingsbeispiele.
Konvergenz ist dann aber nicht mehr garantiert!

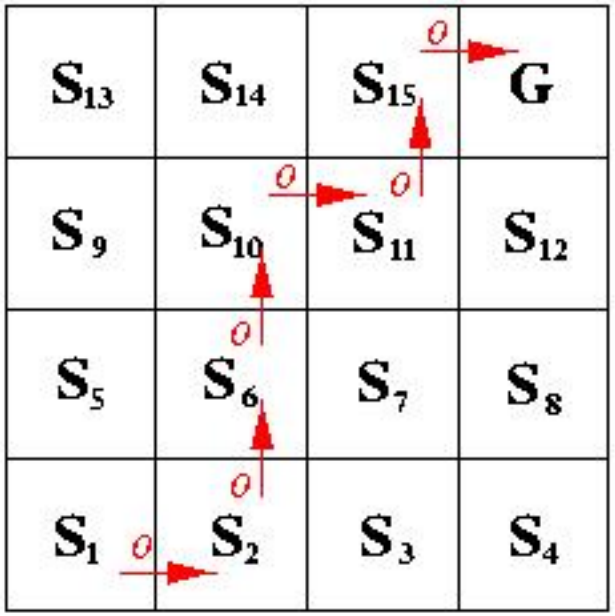
Beispiel: GridWorld - I

gegeben: $m \times n$-Grid

- $S = \{(x, y) | x \in \{1, \cdots, m\}, y \in \{1, \cdots, n\}\}$
- $A = \{up, down, left, right\}$
- $r(s, a) = \begin{cases} 100, \text{falls } \delta(s, a) = \text{Goalstate} \\ 0, sonst. \end{cases}$
- $\delta(s, a)$ gibt den Folgezustand entsprechend der durch a gegebenen Bewegungsrichtung an.

Beispiel: GridWorld - II

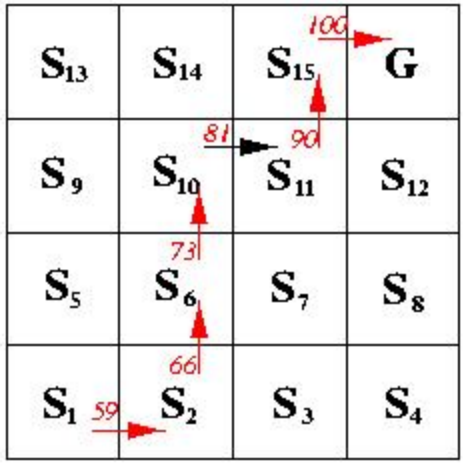
Beispiel für einen Pfad durch einen Zustandsraum:



Die Zahlen an den Pfeilen geben die momentanen Werte von $\hat{Q}$ an.

Beispiel: GridWorld - III

Entwicklung der $\hat{Q}$-Werte:



$$\hat{Q}(S_{11}, up) = r + \gamma \max_{a'} \hat{Q}(s', a') = 0 + 0.9 * 100 = 91$$

$$\hat{Q}(S_{10}, right) = 0 + 0.9 * 91 = 81$$

...

Q-Lernen - offene Fragen

- oft nicht vorhanden: Markov-Annahme, Beobachbarkeit
- kontinuierliche Zustand-Aktion-Räume
- Generalisierung über Zustand und Aktion
- Kompromiss zwischen "Exploration" und "Exploitation"
- Generalisierung der automatischen Bewertung (Kredit-Vergabe)

Ad-hoc Explorationsstrategie

- Zu ausgiebiges Erforschen bedeutet, dass der Agent auch nach langem Lernen noch ziellos im meist sehr großen Zustandsraum umherwandert. Dadurch werden auch Bereiche intensiv untersucht, die für die Lösung der Aufgabe gar nicht relevant sind.
- Zu frühes Ausbeuten der gelernten Approximation der Q-Funktion bewirkt möglicherweise, dass sich ein suboptimaler, d.h. längerer Pfad durch den Zustandsraum, der zufällig zuerst gefunden wurde, etabliert und die optimale Lösung nicht mehr gefunden wird.

Es existieren:

- "Greedy strategies"
- "randomized strategies"
- "interval-based techniques"

Beschleunigung des Lernens

Einige Ad-hoc Techniken:

- Experience Replay
- Backstep Punishment
- Reward Distance Reduction
- Lerner-Kaskade

Experience Replay - I

Ein Pfad durch den Zustandsraum gilt als beendet, sobald G erreicht wurde.
Nun sei angenommen, im Zuge des Q-Learning werde dieser Pfad wiederholt durchlaufen.
Oftmals sind echt neue Lernschritte sehr viel kostenintensiver und/oder zeitaufwendiger als interne Wiederholungen bereits gespeicherter Lernschritte. Aus den genannten Gründen bietet es sich an, die Lernschritte abzuspeichern und intern zu wiederholen.
Das Verfahren wird **Experience Replay** genannt.

Experience Replay - II

Eine *Erfahrung* e (engl.: *experience*) ist ein Tupel

$$e = (s, s', a, r)$$

mit $s, s' \in S$, $a \in A$, $r \in \mathbb{R}$. e repräsentiert einen Lernschritt, d.h. einen Zustandsübergang, wobei s der Ausgangszustand, s' der Zielzustand, a die Aktion, die zu dem Zustandsübergang führte, und r das dabei erhaltene Reinforcement-Signal ist.

Ein *Lernpfad* ist dann eine Serie $e_1 \ldots e_{L_k}$ von Erfahrungen (L_k ist die Länge des k-ten Lernpfades).

Experience Replay - III

Der ER-Algorithmus:

for $k = 1$ to N
    for $i = L_k$ down to 1
        update(e_i aus Serie k)
    end for
end for

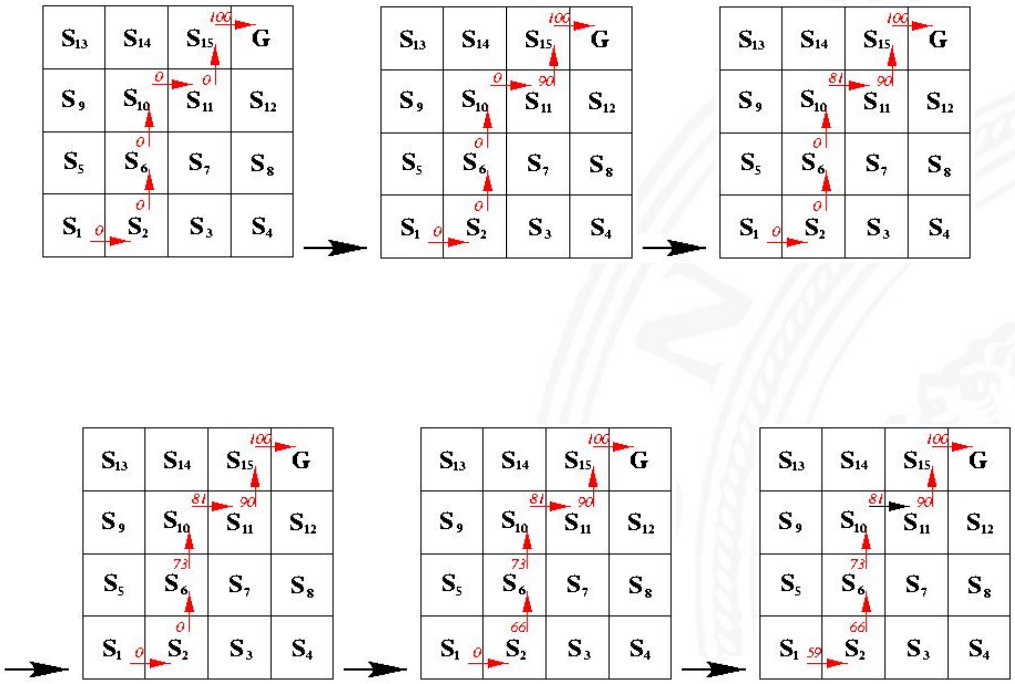
Experience Replay - IV

Vorteile:

- Interne Wiederholungen gespeicherter Lernschritte verursachen meist weit weniger Kosten als echt neue Lernschritte.
- Intern läßt sich ein Lernpfad in umgekehrter Reihenfolge durchlaufen, was die Informationsausbreitung beschleunigt.
- Wenn sich Lernpfade kreuzen, können sie sozusagen "voneinander lernen", d.h. Information austauschen. Experience Replay macht diesen Austausch unabhängig von der Reihenfolge, in der die Lernpfade erstmals ausgeführt wurden.

Experience Replay - Beispiel

Entwicklung der $\hat{Q}$-Werte bei wiederholtem Durchlaufen eines Lernpfades:



Backstep Punishment

Eine Explorationsstrategie wird benötigt, die für eine etwas "geradlinigere" Bewegung des Agenten durch den Zustandsraum sorgt. Dazu müssen vor allem Rückschritte vermieden werden.
Eine sinnvollere Methode scheint es zu sein, für den Fall, dass der Agent einen Rückschritt auswählt, ihn diesen ausführen zu lassen, aber ein *künstliches, negatives Reinforcement-Signal* zu generieren.
Kompromiss zwischen "Sackgasse-Vermeidung" und "schnellem Lernen".
Beim zielorientierten Lernen könnte eine entsprechend erweiterte Reward-Funktion wie folgt aussehen:

$$r_{\mathrm{BP}} = \begin{cases} 100 & \text{falls Übergang in einen Zielzustand erfolgte} \\ -1 & \text{falls ein Rückschritt gemacht wurde} \\ 0 & \text{sonst} \end{cases}$$

Reward Distance Reduction

Die Reward-Funktion kann eine intelligentere Bewertung der Aktionen des Agenten vornehmen. Dies setzt aber Kenntnisse über die Struktur des Zustandsraumes voraus.
Wenn man zusätzlich die Kodierung des Zielzustandes kennt, dann kann es eine gute Strategie sein, die euklidische Distanz zwischen dem aktuellen Zustand und dem Zielzustand zu verringern.
Man kann die Rewardfunktion so erweitern, dass Aktionen, die die euklidische Distanz zum Zielzustand verringern, belohnt werden (*reward distance reduction*, *RDR*):

$$r_{\text{RDR}} = \begin{cases} 100 & \text{falls } \vec{s}' = \vec{s}_g \\ 50 & \text{falls } |\vec{s}' - \vec{s}_g| < |\vec{s} - \vec{s}_g| \\ 0 & \text{sonst} \end{cases}$$

wobei $\vec{s}$, $\vec{s}'$ und $\vec{s}_g$ die den Ausgangszustand, den Endzustand und den Zielzustand kodierenden Vektoren sind.

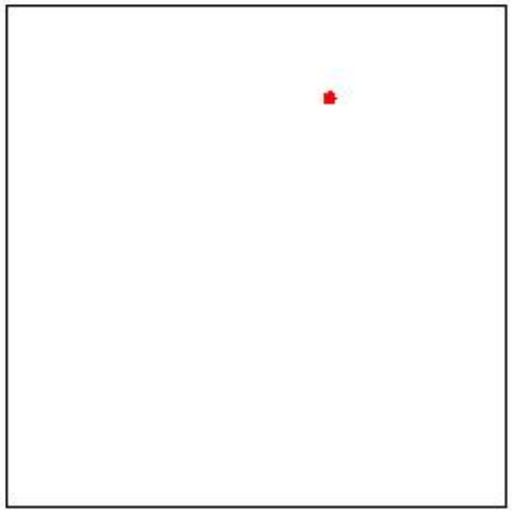
Lerner-Kaskade - I

Es ist für die Genauigkeit der Positionierung entscheidend, wie fein der Zustandsraum eingeteilt ist.
Andererseits steigt mit wachsender Feinheit der Diskretisierung auch die Anzahl der Zustände und damit der Aufwand für das Lernen.
Man muss sich vor dem Lernen für eine Diskretisierung entscheiden und dabei abwägen zwischen Lernaufwand und Genauigkeit der Positionierung.

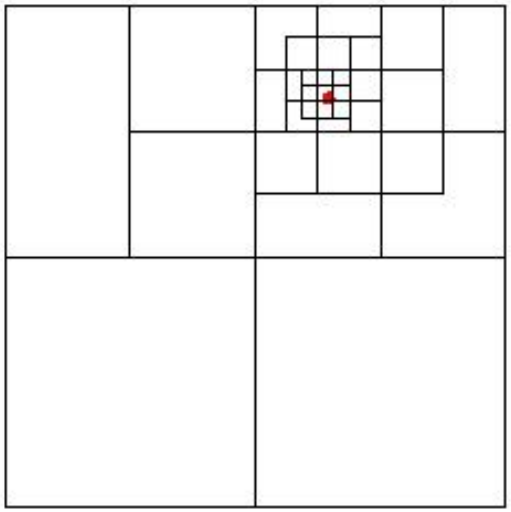
Lerner-Kaskade - Variable Diskretisierung

Ein Beispiel-Zustandsraum a) ohne Diskretisierung b) mit variabler Diskretisierung

a)



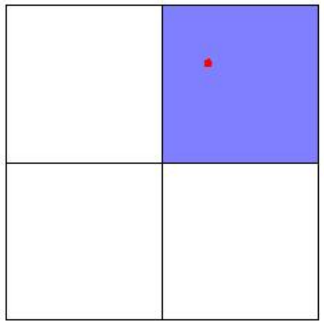
b)

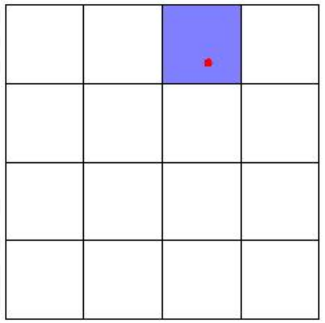


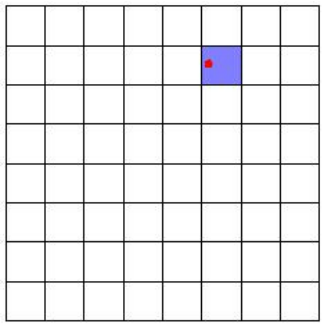
Dies setzt aber Kenntnisse über die Struktur des Zustandsraumes voraus.

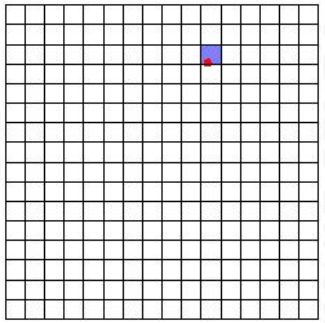
Lerner-Kaskade - *n*-stufige Lerner-Kaskade

Einteilungen des Zustandsraumes einer vierstufigen Lerner-Kaskade:

1. Lerner: 

2. Lerner: 

3. Lerner: 

4. Lerner: 

Bahnplanung mit der Policy Iteration - II

- Für die Rewardfunktion $R^a_{ss'}$ gilt:
  $R^a_{ss'} =$ **Reward_nicht_im_Ziel** $\quad \forall s \in S, a \in A(s), s' \in S$ und
  $R^a_{ss'} =$ **Reward_im_Ziel** $\quad \forall s \in S, a \in A(s), s' = s_t$.
  Also nur für das Erreichen des Zielzustandes wird ein anderer Rewardwert geliefert. Für alle anderen Zustände gibt es einen einheitlichen Betrag
- Die Policy $\pi(s, a)$ sei ebenfalls deterministisch, dh. es gibt genau ein a mit $\pi(s, a) = 1$
- Eine Schranke Θ, bei der die Policy Evaluation beendet wird, muss gewählt werden
- Für das Problem ist das Infinite-Horizon Discounted Model am sinnvollsten, deshalb muss γ entsprechend gesetzt werden

Q-Lernen: Zusammenfassung

- Q-Funktion
- Q-Lernalgorithmus
- Konvergenz
- Beschleunigung des Lernens
- Beispiele

Bahnplanung mit DP - I

Zum finden des Pfades zwischen Start- und Zielkonfiguration wird die Policy Iteration benutzt.
Folgende Größen müßen dementsprechend beim vorliegenden Planungsproblem definiert werden:

- Als Zustandsmenge S dient der diskrete Konfigurationsraum, dh. jedes mögliche Gelenkwinkeltupel $(\theta_1, \theta_2, \ldots, \theta_{Dim})$ ist genau ein Zustand der Menge S mit Ausnahme der Zielkonfiguration
- Die Menge $A(s)$ aller möglichen Aktionen für einen Zustand s sind die Übergänge zu Nachbarkonfigurationen im K-Raum, also für einen oder mehrere Gelenkwinkel θ_i die Änderung um $\pm Dis$. Dabei sind nur Aktionen a in $A(s)$ enthalten, die nicht zu K-Hindernissen führen, dh. die nicht mit Hindernissen kollidieren, und nicht die Grenzen des K-Raumes verletzen

Bahnplanung mit DP - II

- Für die Rewardfunktion $R^a_{ss'}$ gilt:

  $R^a_{ss'} =$ **Reward_nicht_im_Ziel** $\quad \forall s \in S, a \in A(s), s' \in S$ und
  $R^a_{ss'} =$ **Reward_im_Ziel** $\quad \forall s \in S, a \in A(s), s' = s_t$.

  Also nur für das Erreichen des Zielzustandes wird ein anderer Rewardwert geliefert. Für alle anderen Zustände gibt es einen einheitlichen Betrag

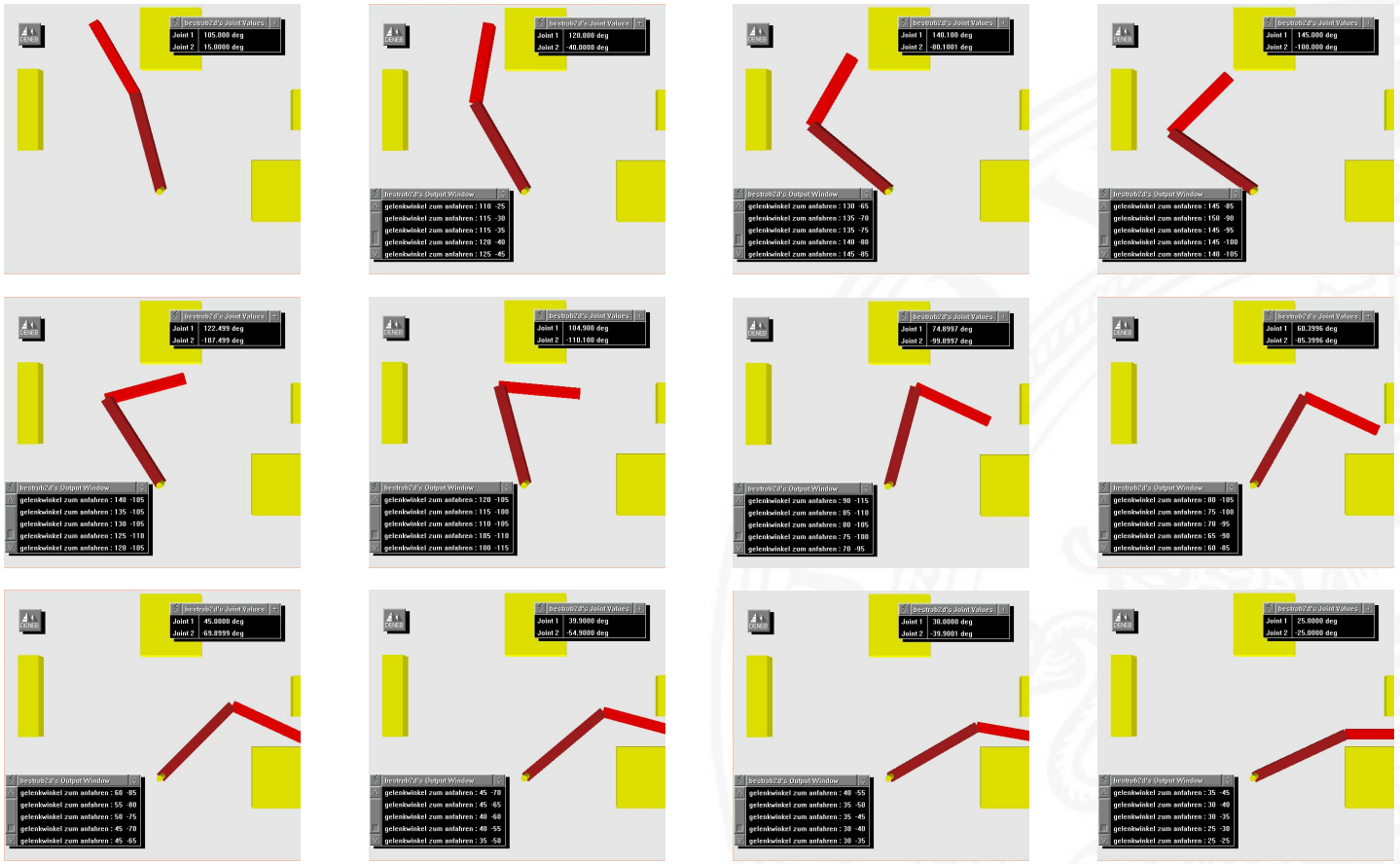
- Die Policy $\pi(s, a)$ sei ebenfalls deterministisch, dh. es gibt genau ein a mit $\pi(s, a) = 1$

- Eine Schranke Θ, bei der die Policy Evaluation beendet wird, muss gewählt werden

- Für das Problem ist das Infinite-Horizon Discounted Model am sinnvollsten, deshalb muss γ entsprechend gesetzt werden

2-gelenkiger Roboter

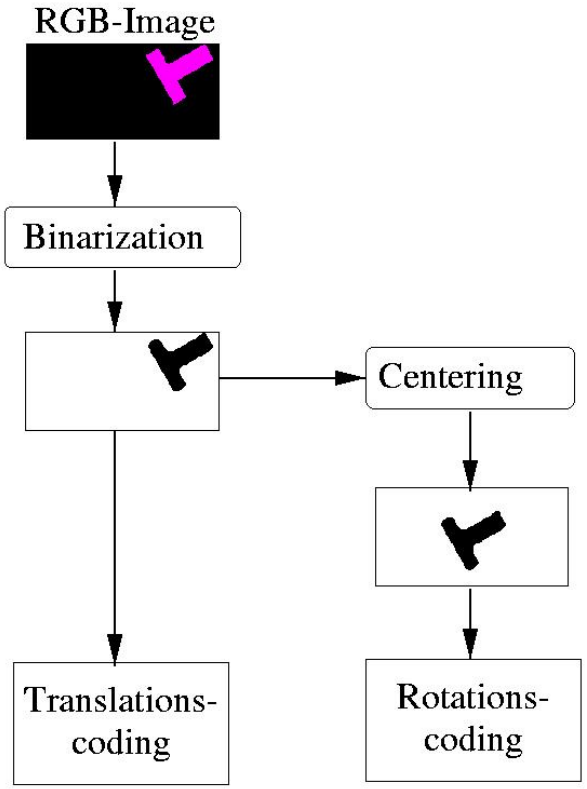
Die gefundene Bewegungssequenz des 2-gelenkigen Roboters (von links nach rechts, von oben nach unten):



Grasping mit RL: Aufteilung der DOFs

Üblicherweise braucht ein Roboterarm 6 DOFs um ein Objekt aus irgendeiner Position und Orientierung zu greifen.
Um quasi-planare Objekte zu greifen nehmen wir an, dass der Greifer senkrecht zum Tisch steht und dass sein Gewicht bekannt ist.
Es bleiben immer noch drei DOFs zur Kontrolle des Roboterarms: Bewegungen parallel zur Tischebene (x, y) und die Rotation um die senkrecht zum Tisch stehende Achse (θ).

Grasping mit RL: Aufteilung der DOFs(2)



Kontrolle von x, y, θ. Die vor-prozessierten Bilder werden für die Rotations-Kodierung zusätzlich zentriert.

Grasping mit RL: Aufteilung der DOFs(3)

Um einen kleinen *Zustandsraum* zu erzeugen, wird das Lernen zwischen zwei Lernern aufgeteilt:
einer für die *translationale* Bewegung auf der Ebene,
der andere für die Rotationsbewegung.
Der *Translations-Lerner* kann dann vier Aktionen auswählen (zwei in x- und zwei in y-Richtung).
Der *Rotations-Lerner* kann zwei Aktionen auswählen (Rotation im und gegen den Uhrzeigersinn).
Diese Aufteilung hat folgende Vorteile gegenüber einem monolithischem Lerner:

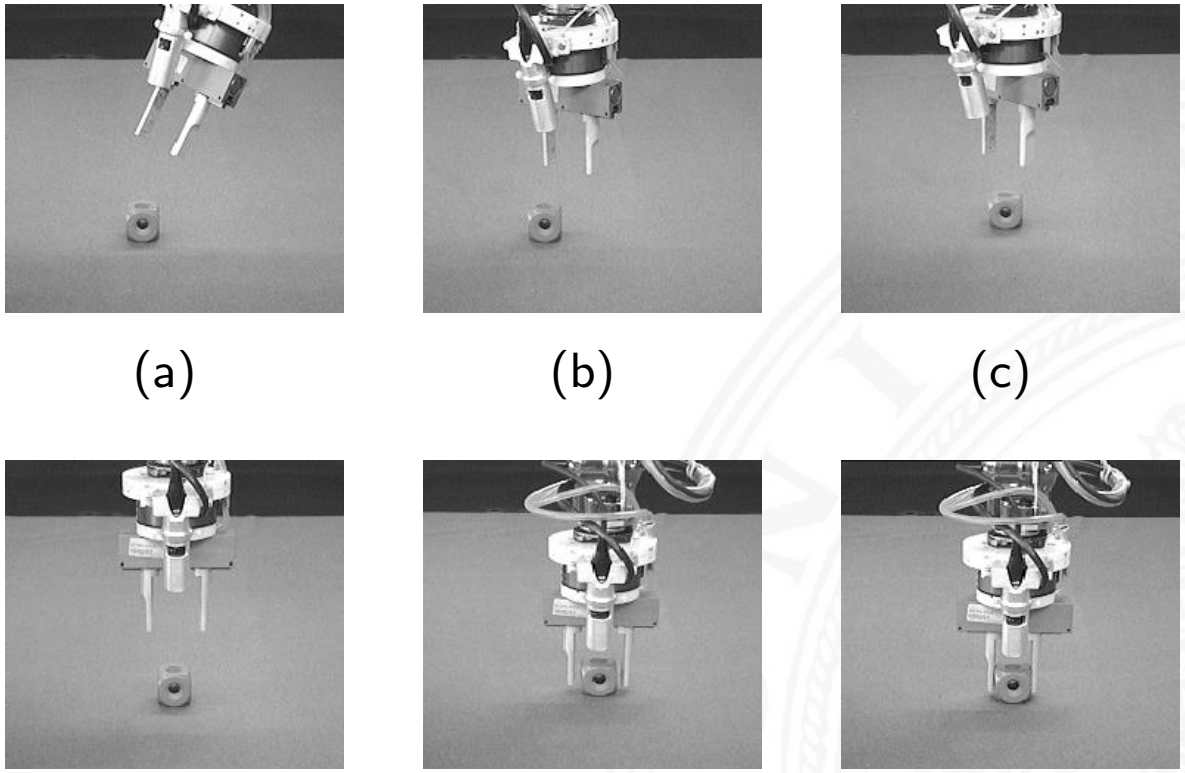
- Der Zustandsraum ist wesentlich kleiner.
- Die Zustandskodierung ist so entworfen, dass die Zustandsvektoren nur die relevanten Informationen für den entsprechenden Lerner enthalten.

Grasping mit RL: Aufteilung der DOFs(4)

In der Praxis werden die beiden Lerner abwechselnd angewandt.
Zuerst wird der Translations-Lerner in langen Lern-Schritten angewandt, bis er das in seiner Zustands-Kodierung definierte Ziel erreicht hat.
Dann wird der Translations-Lerner durch den Rotations-Lerner ersetzt, der ebenfalls in langen Lern-Schritten angewandt wird bis sein Ziel erreicht ist.

Zu diesem Zeitpunkt kann es passieren, dass der Translations-Zielzustand vom Rotations-Lerner gestört wird. Daher wird der Translations-Lerner noch einmal aktiviert. Diese Prozeduren werden wiederholt, bis beide Lerner berichten, dass sie alle den Zielzustand erreichen. Dieser Zustand ist daher der gemeinsame Zielzustand.

Grasping mit RL: Aufteilung der DOFs(5)



Positionierung und Orientierungs-Kontrolle mit 6 DOFs in vier Schritten.

Grasping mit RL: Aufteilung der DOFs(6)

Um alle sechs DOFs zu benutzen sollten zusätzliche Lerner eingeführt werden.

1. Der erste Lerner besitzt zwei DOFs und seine Aufgabe ist dass das Objekt von einer bestimmten Perspektive aus betrachtet wird. Für einen ebenen Tisch bedeutet dies typischerweise, dass der Greifer senkrecht zur Tischoberfläche positioniert werden muss (a $\rightarrow$ b).
2. Wende den x/y Lerner an (b $\rightarrow$ c).
3. Wende den θ Rotations-Lerner an (c $\rightarrow$ d).
4. Der letzte Höhen-Lerner wird die z-Koordinate korrigieren, die Höhe über dem Tisch (d $\rightarrow$ e).

Visuell geführtes Greifen mittels selbstbewertendem Lernen

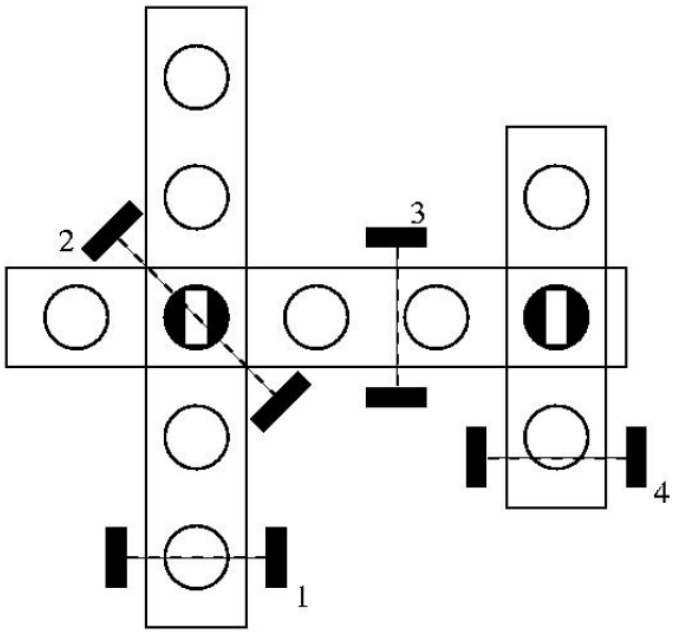
Griff ist optimal bzgl. lokaler Kriterien:

- Die Finger des Greifers können das Objekt am Greifpunkt umschließen
- Keine Reibung tritt zwischen Fingern und Objekt auf

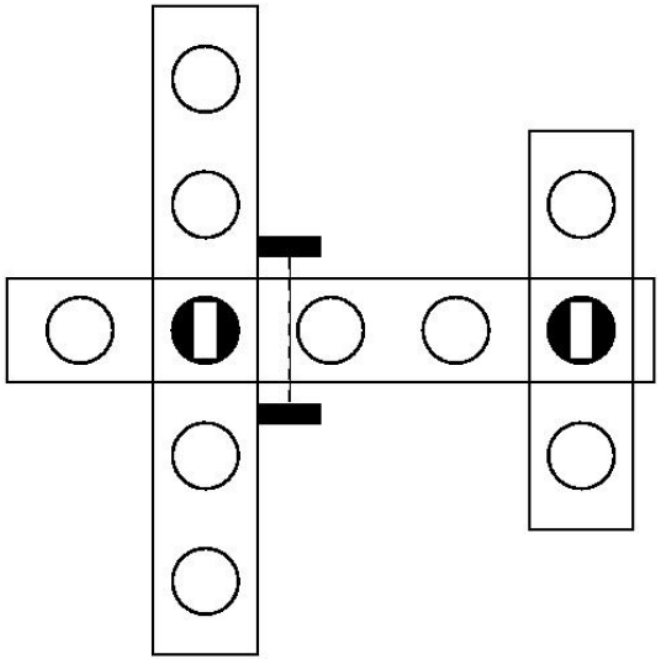
Griff ist optimal bzgl. globaler Kriterien:

- Kein bzw. minimales Drehmoment an den Fingern
- Objekt rutscht nicht aus dem Greifer
- Der Griff ist stabil, d.h. Objekt sitzt fest zwischen den Fingern

Bsp. Lokale und globale Kriterien



(a)



(b)

Zwei Ansätze

Ein Lerner:

- Die Zustände bestehen aus einem Satz $m + n$ lokaler und globaler Eigenschaften: $s = (f_{l_1}, \ldots, f_{l_m}, f_{g_1}, \ldots, f_{g_n})$.
- Der Lerner versucht sie auf die Aktionen $a = (x, y, \phi)$ abzubilden, wo x und y in Richtung x und y translationale Komponenten sind und ϕ die Rotation um den Annäherungsvektor des Greifers ist.

Zwei Lerner:

- Die Zustände für den ersten Lerner liefern nur die lokalen Eigenschaften $s = (f_{l_1}, \ldots, f_{l_m})$.
- Der Lerner versucht sie auf Aktionen abzubilden, die nur aus der Rotationskomponente $a = (\phi)$ bestehen.
- Der zweite Lerner versucht, Zustände globaler Eigenschaften $s = (f_{g_1}, \ldots, f_{g_n})$ auf Aktionen translationaler Komponenten abzubilden: $a = (x, y)$.

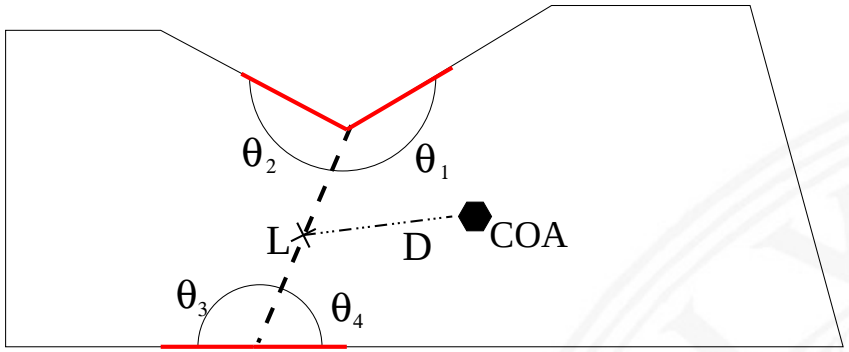
Aufbau

- Zweikomponentiges Lernsystem:

1. Orientierungs-Lerner	**2. Orts-Lerner**
Operiert auf lokalen Kriterien	Operiert auf globalen Kriterien
Gleich für jedes Objekt	Unterschiedlich für jedes neue Objekt

- Einsatz von Multisensorik:
  - Kamera
  - Kraft-Moment-Sensor
- Beide Lerner arbeiten aufeinanderfolgend im Perzeptions-Aktions-Zyklus

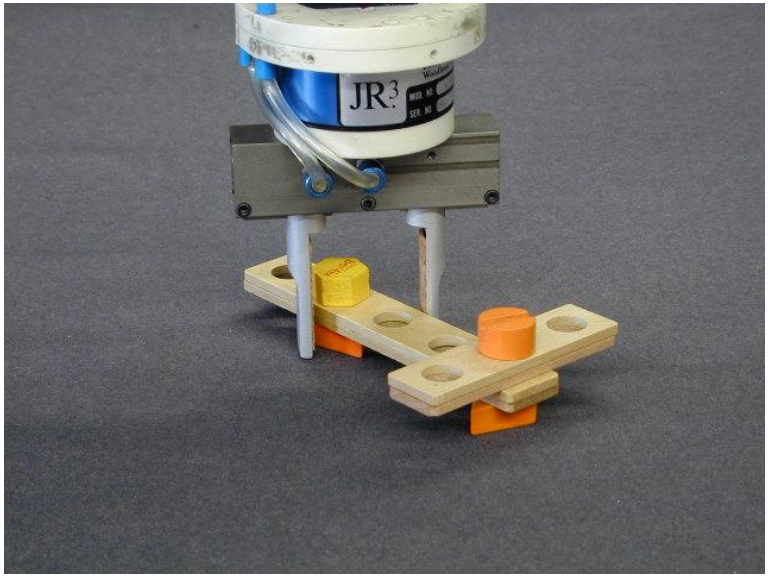
Zustandskodierung

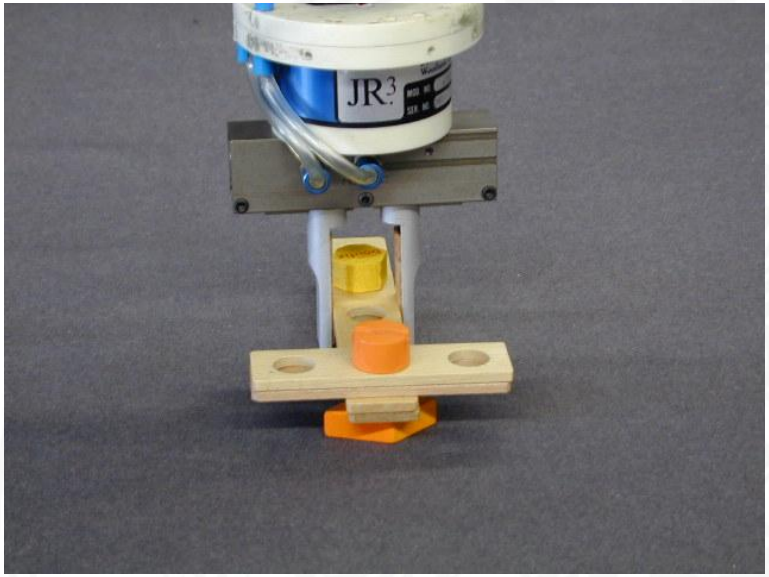


Der Orientierungs-Lerner benutzt Länge L sowie Winkel $\Theta_1, \ldots, \Theta_4$ während der Orts-Lerner die Distanz D zwischen Mittelpunkt der Greiflinie und Bildschwerpunkt des Objektes nutzt.

Selbstbewertungsmaßnahmen im Orientierungs-Lerner

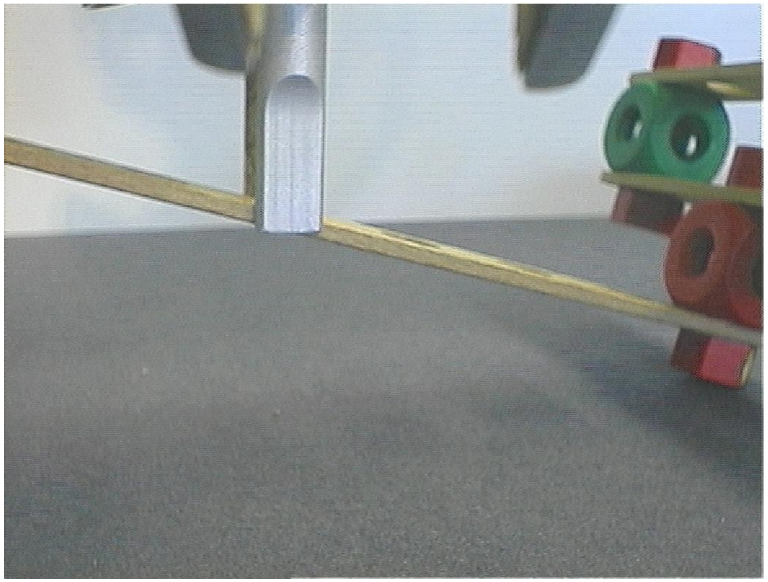
Visuelle Bewertung des Grifferfolges:
Reibung resultiert in Rotation oder Versatz des Objektes

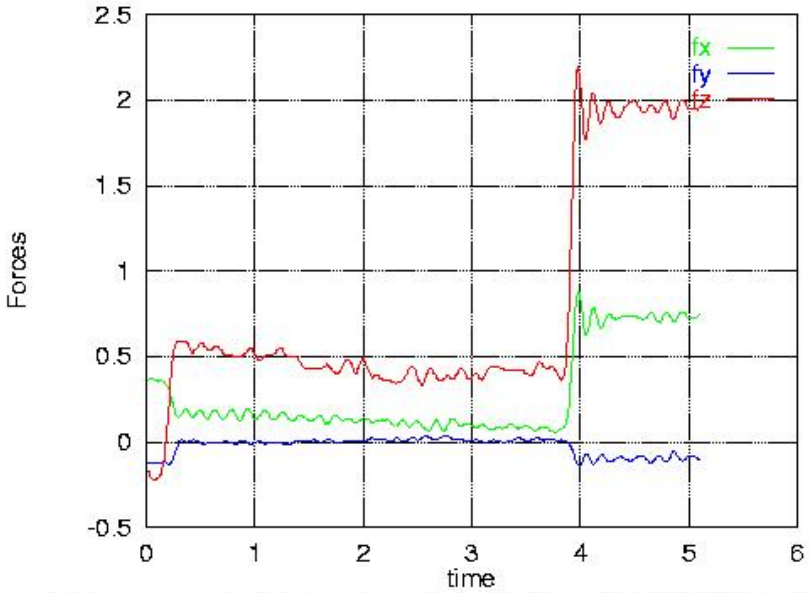




Selbstbewertungsmaßnahmen im Orts-Lerner I

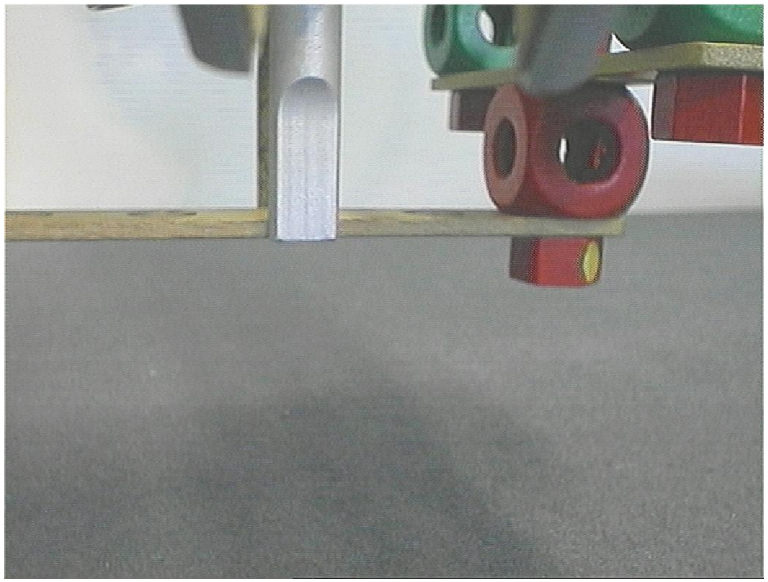
Bewertung mittels Kraftmomenten-Sensors:
Instabiler Griff – analysiert durch Kraftmessung

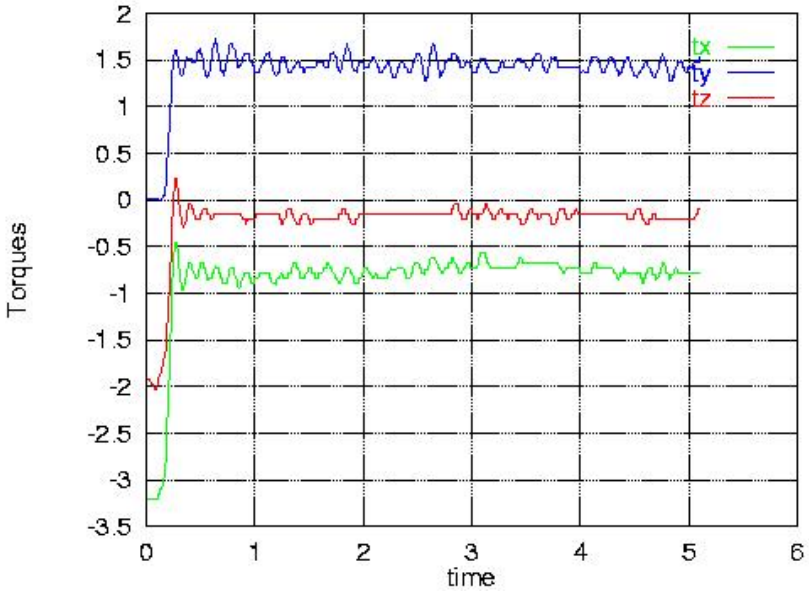




Selbstbewertungsmaßnahmen im Orts-Lerner II

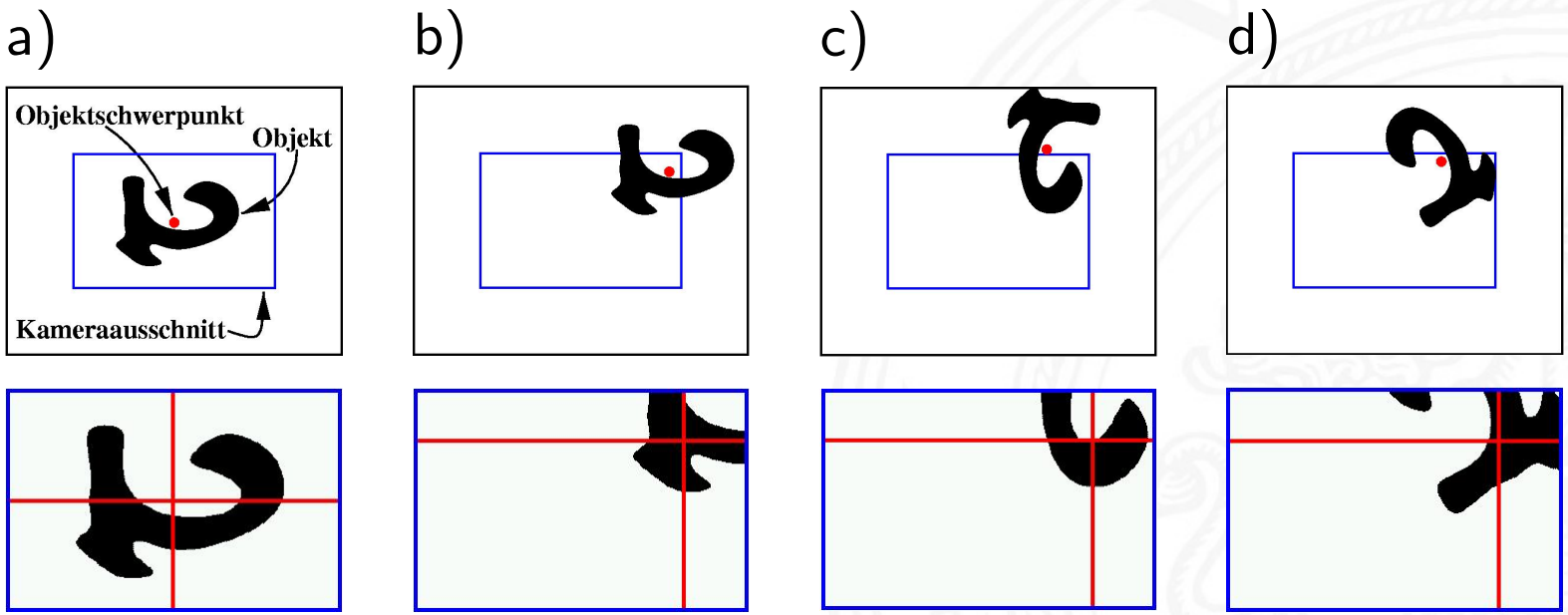
Suboptimaler Griff – analysiert über Drehmomente





Das Problem der unvollständigen Zustandsinformation

Man spricht auch von **verborgenen Zuständen** (engl.: *hidden states*).
Beispiel für unvollständige Zustandsinformation:

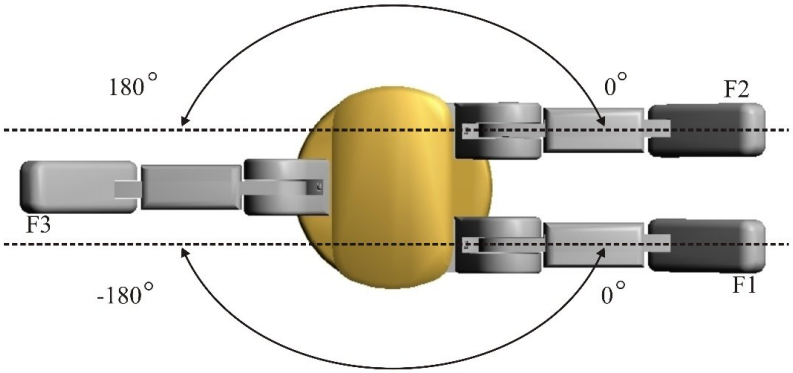


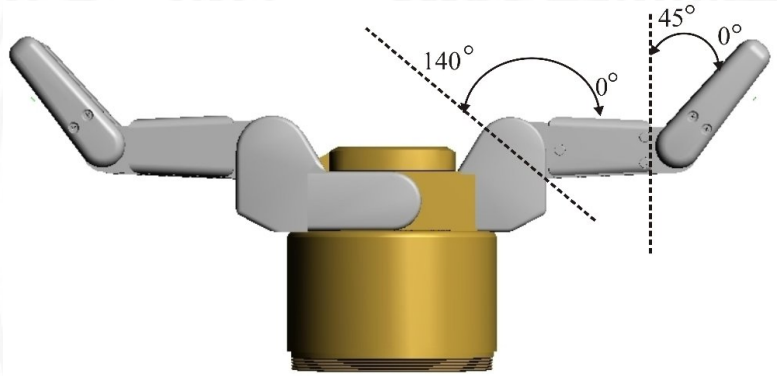
Handhabung von Alltagsgegenständen

- Lernen des Greifens von Alltagsgegenständen
- Reinforcement-Learning-Prozess basiert auf Simulation
- Ziel: möglichst viele geeignete Griffe finden
- Berücksichtigung möglicher Folgeoperationen
- anwendbar auf beliebige Objekttypen
- Effizienz bezüglich:
  - Speicherbedarf
  - gefundene Griffe/Lernepisode

BarretHand BH-262

- 3-Finger Roboter-Hand
- Finger können einzeln öffnen/schließen
- variabler Spreizwinkel
- Positions- und Kraftmessung
- TorqueSwitch-Mechanismus
- Spannweite 32 cm





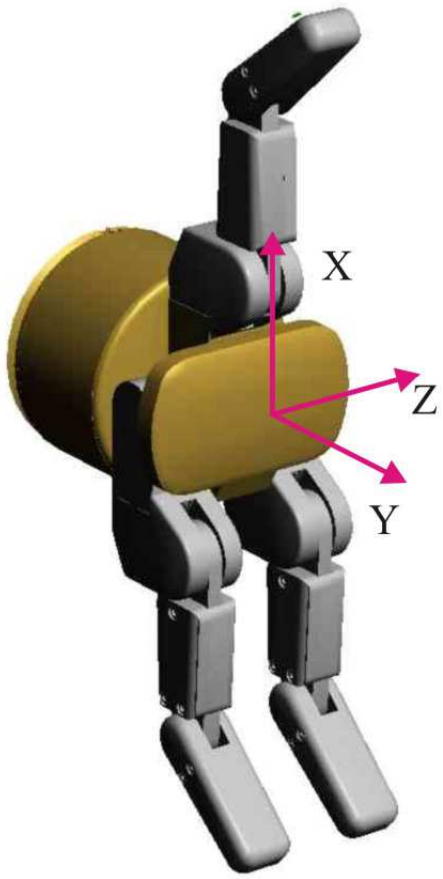
Angewandtes Modell

Zustände S:

- Position des Greifers zum Objekt
- Spreizwinkel der Hand
- Griff bereits ausgeführt?

Aktionen a:

- Translation (x-Achse, y-Achse, z-Achse)
- Rotation(roll-, yaw-, pitch-Achse)
- Veränderung des Spreizwinkels
- Griff ausführen



Angewandtes Modell (cont.)

Action-Selection:

- ϵ-greedy (mit Wahrscheinlichkeit ϵ Zufallsaktion)

Reward-Funktion:

- Reward abhängig von Stabilität des Griffes
- Stabilität wird über "wrench-space" (GWS) berechnet (Ferrari und Canny, 1992)
- kleiner negativer Reward bei instabilen Griffen
- großer negativer Reward falls Objekt verfehlt

$$r(s,a) = \begin{cases} -100 & \text{if number of contact points} < 2 \\ -1 & \text{if } GWS(g) = 0 \\ GWS(g) & \text{otherwise} \end{cases}$$

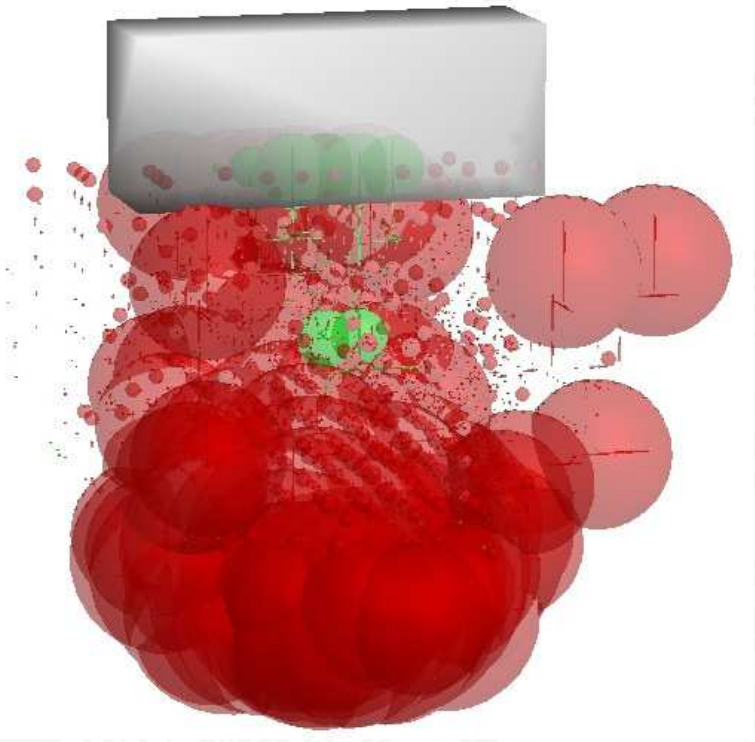
Lern- Strategie

Problem: Der Zustandsraum ist extrem groß:

- TD-(λ)-Algorithmus
- Lernen in Episoden
- eine Episode endet:
  - nach einer festen Anzahl Schritten
  - nach einen Greif-Versuch
- Q-table wird dynamisch erstellt
- Zustände werden erst beim ersten Auftreten hinzugefügt

Problem der Terminalzustände

Problem: Mit der Aktion „Griff“ kann jeder Zustand in einen Terminalzustand überführt werden. Möglicherweise findet man nur ein lokales Maximum und somit nicht den optimalen Griff.



Problem der Terminalzustände

Auswirkungen auf das Lernverhalten:

- instabile Griffe werden mehrmals getestet
- Agent verbleibt in lokalen Minima
- nicht alle Griffe werden gefunden

$\Longrightarrow$ Kein Lernen am Ende einer Episode - Verschwendung von Rechenzeit

Dies kann nicht durch Anpassen der RL-Lernparameter gelöst werden.

Automatischer Value Cutoff (cont.)

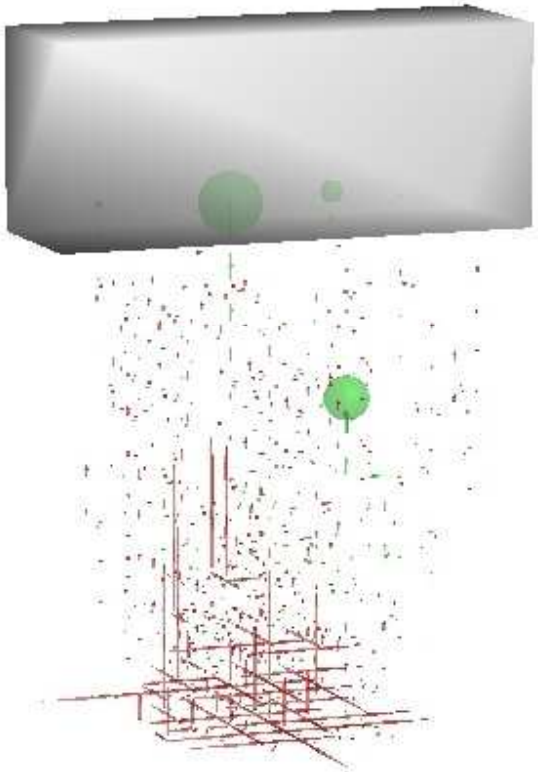
Automatischer Value Cutoff: Löschen von Aktionen

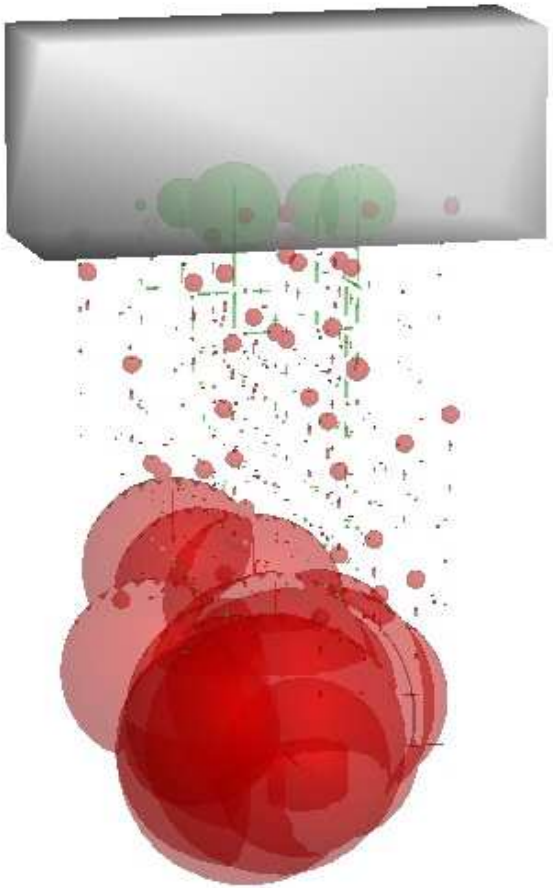
- die zu instabilen Griffen führen (Reward < 0)
- die bereits mehrfach evaluiert sind

$$Q(s,a) \leftarrow \begin{cases} Q(s,a) + \alpha \left[r + \gamma Q(s',a') - Q(s,a) \right] \\ \text{falls } 0 \leq Q(s,a) < r * \beta \\ \\ \text{anderenfalls entferne } Q(s,a) \text{ aus } Q \end{cases}$$

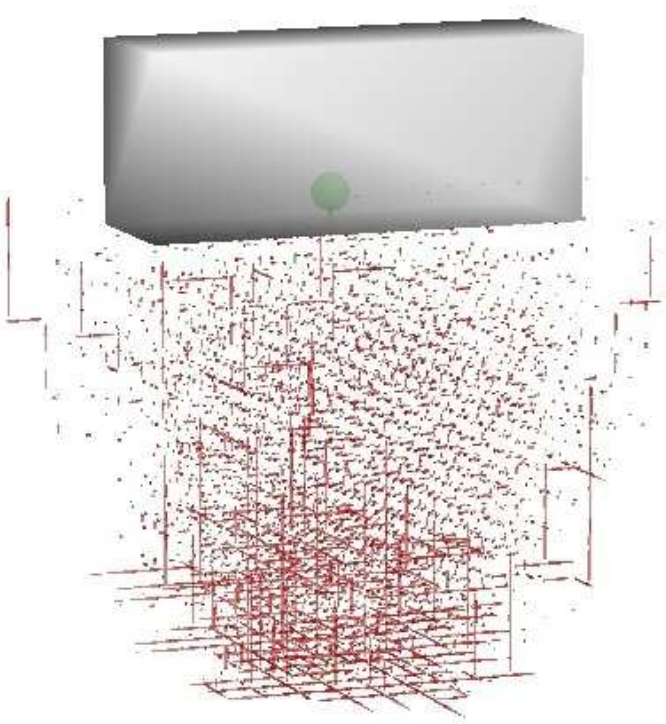
mit $0 \leq \beta \leq 1$. (gute Ergebnisse lieferte $\beta = 0.95$)

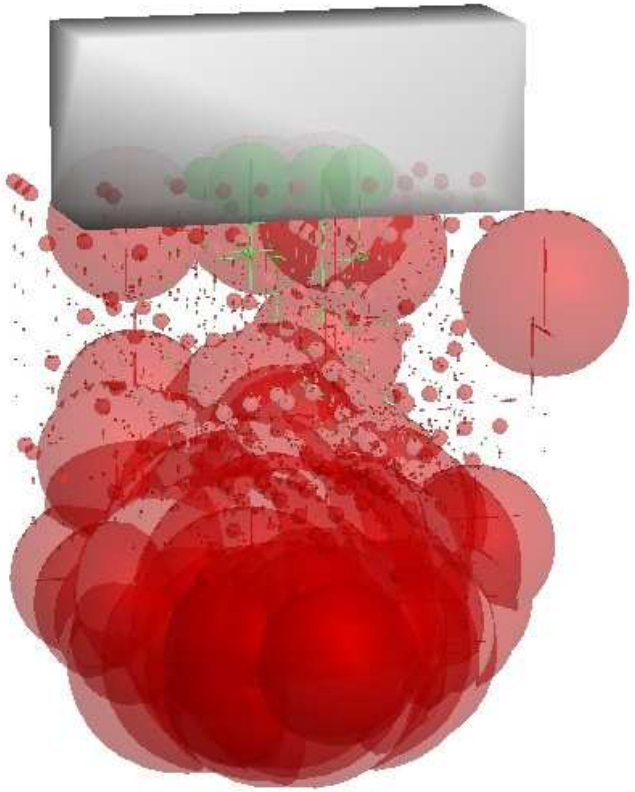
Automatischer Value Cut-Off: 100 Episoden



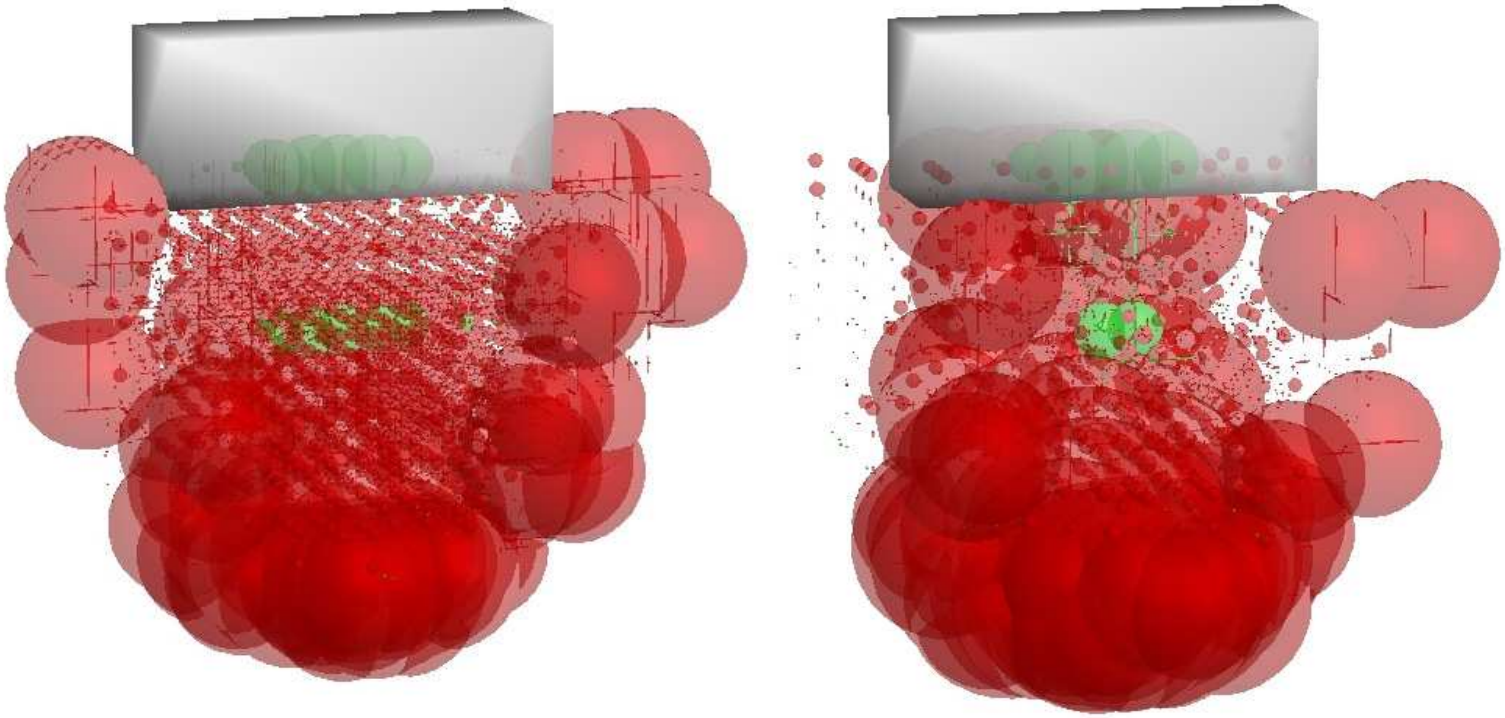


Automatischer Value Cut-Off: 750 Episoden

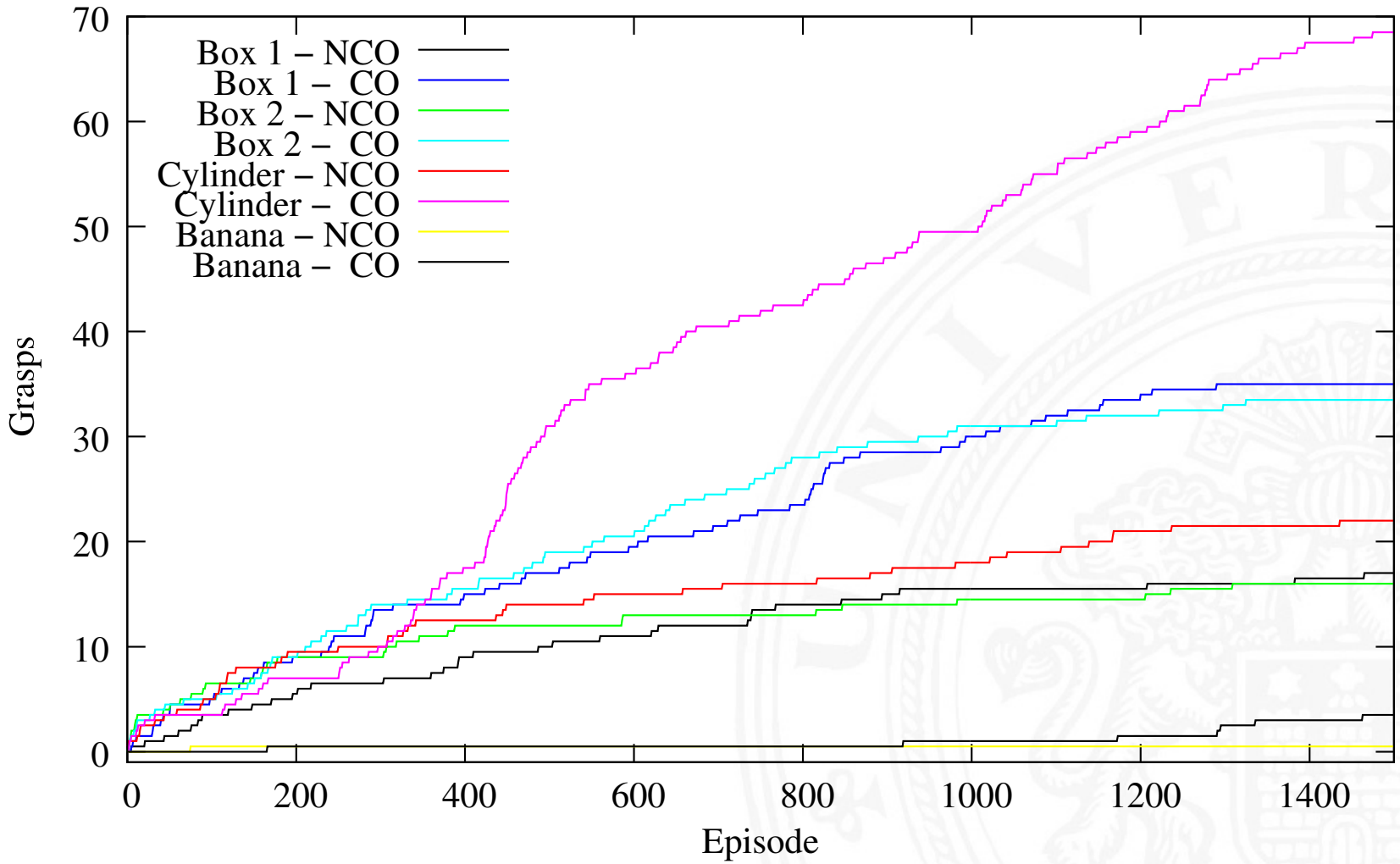




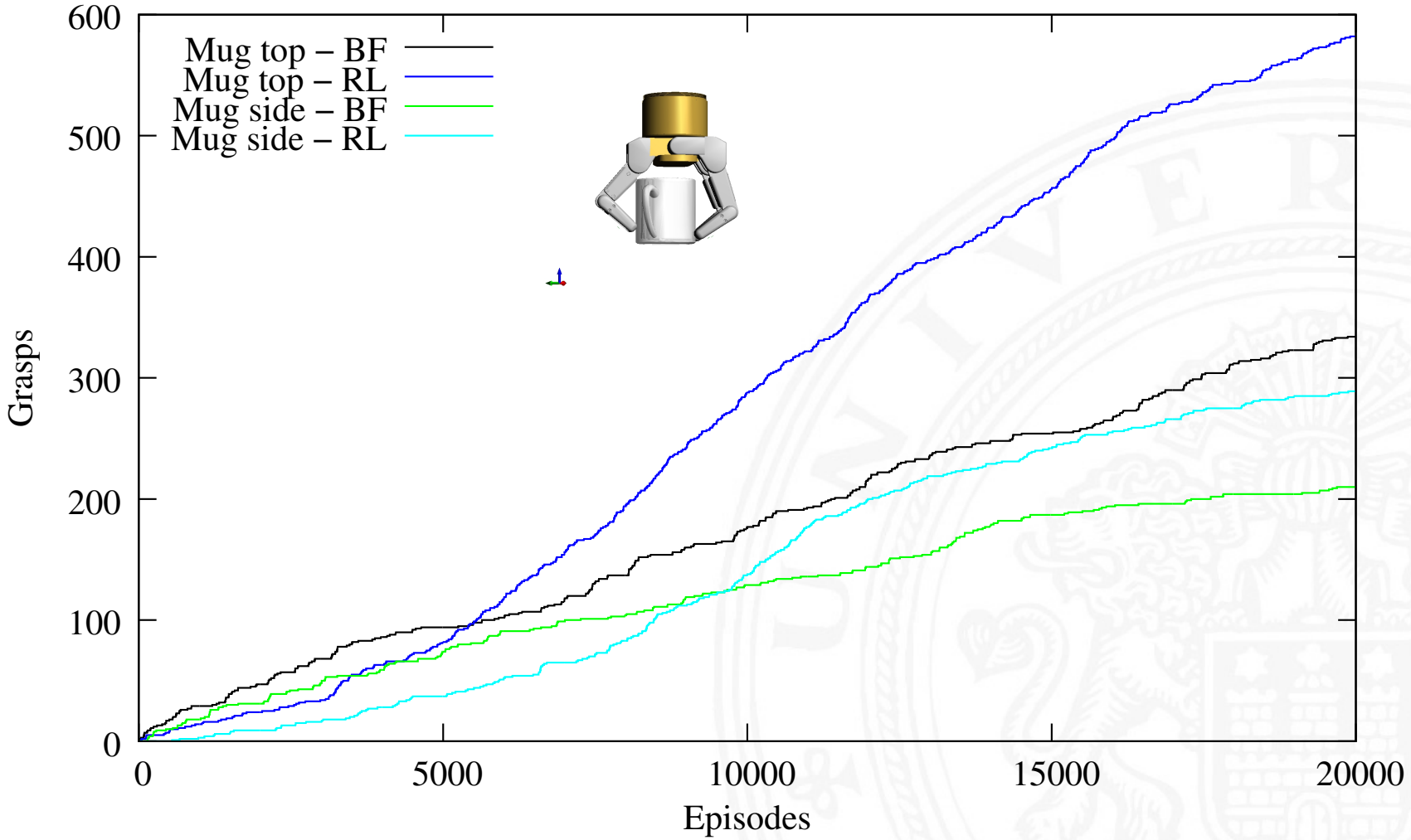
Automatischer Value Cut-Off: 1500 Episoden



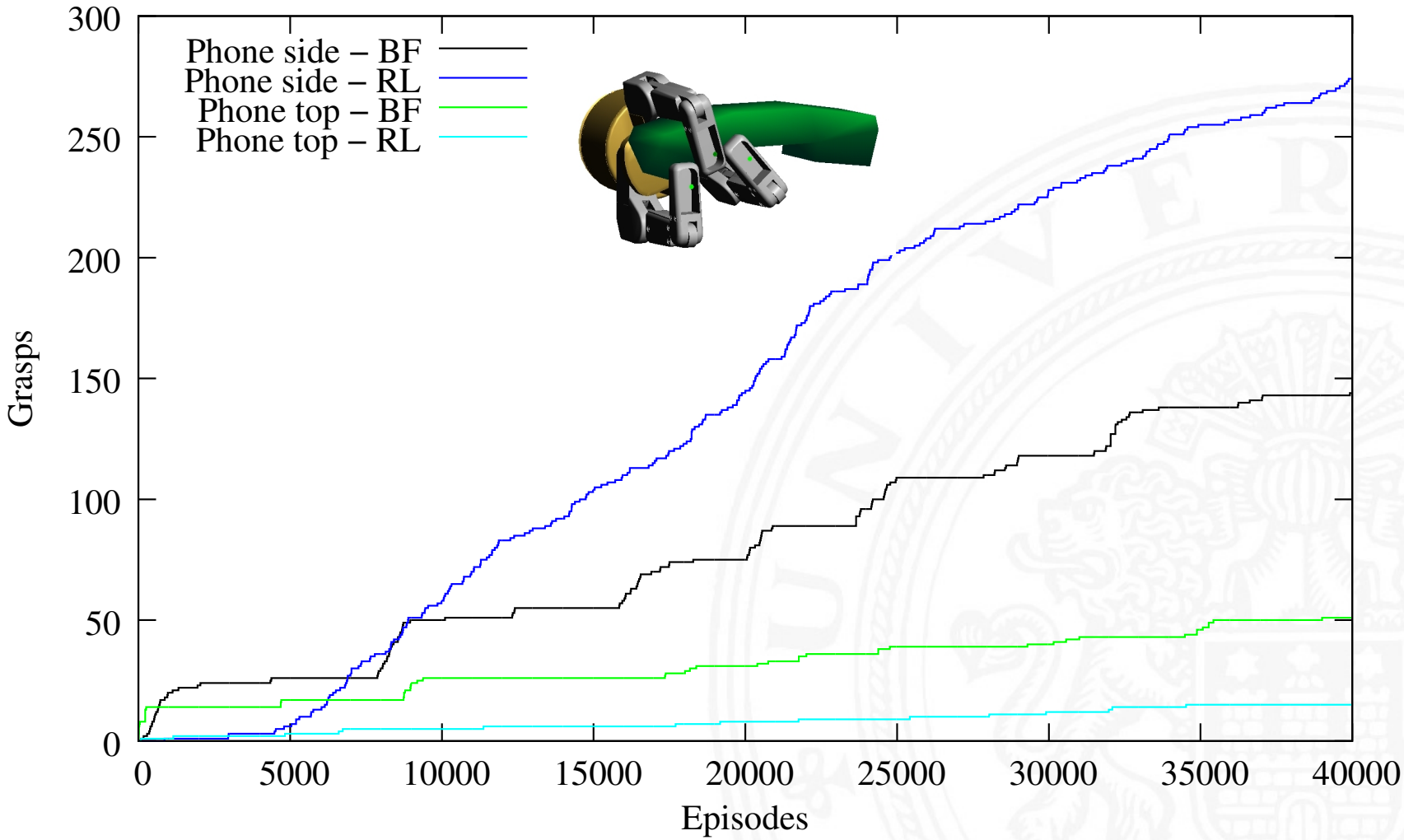
Automatischer Value Cutoff vs. kein Cutoff



Reinforcement Learning vs. Brute Force: Tasse

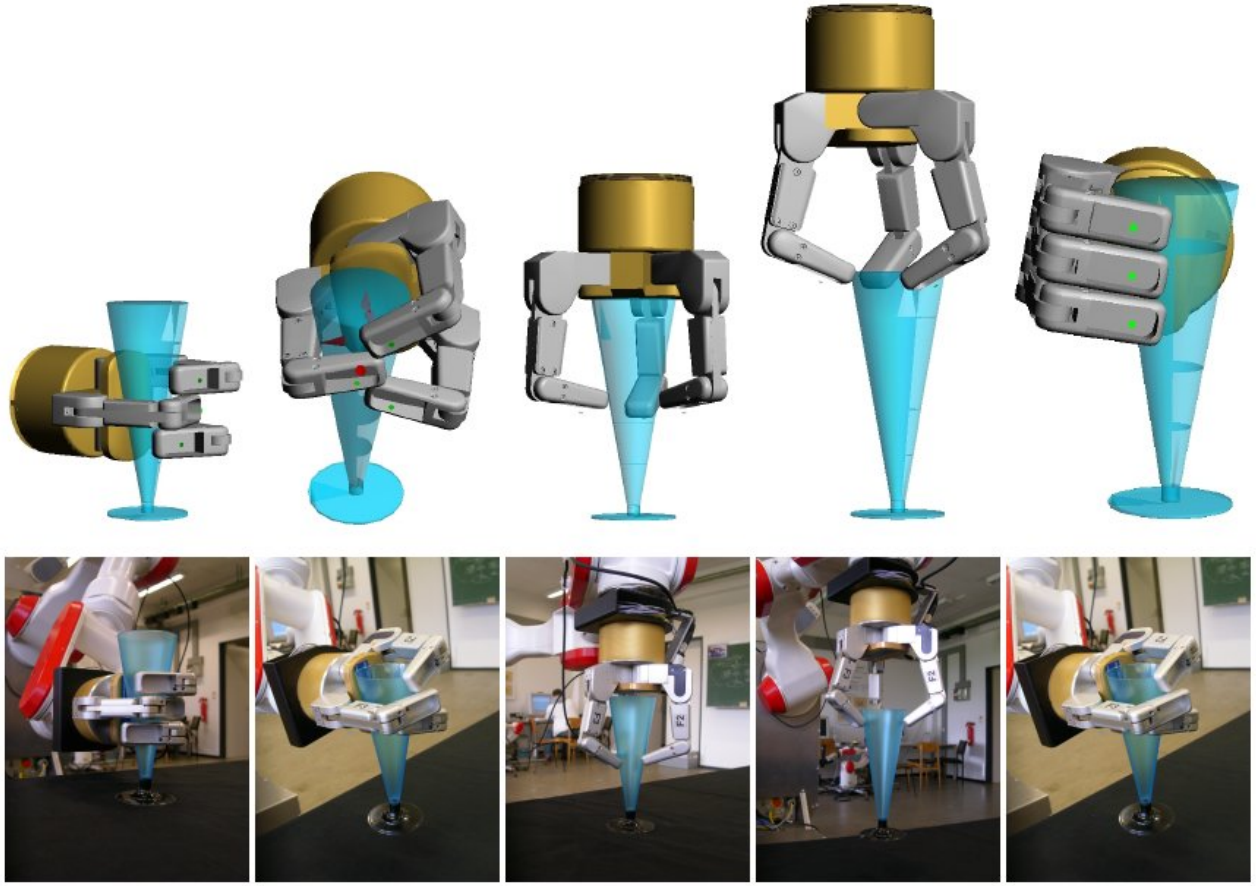


Reinforcement Learning vs. Brute Force: Telefon



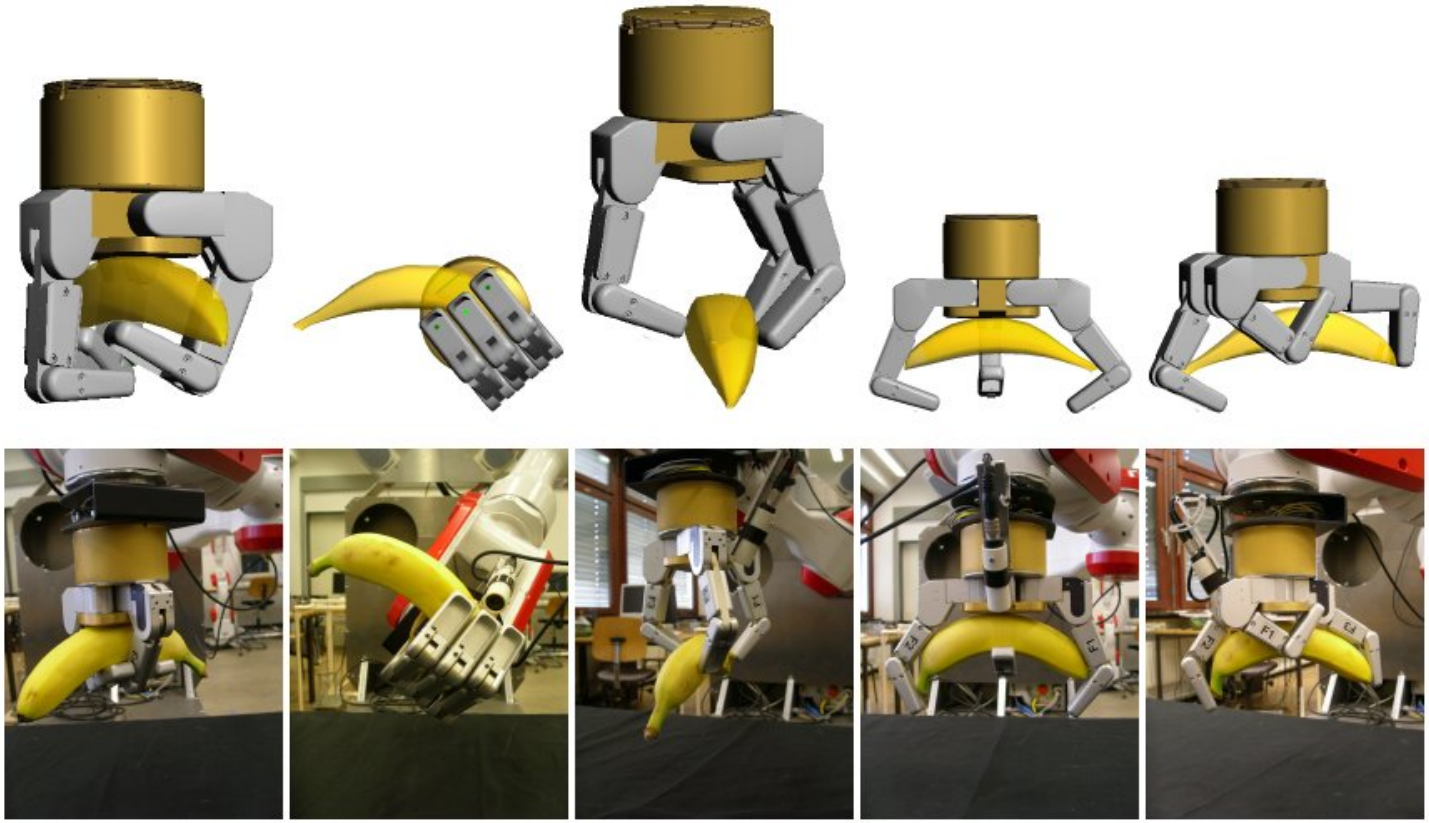
Experimente mit TASER

Test einiger Griffe auf der Roboterplattform:



Experimente mit TASER (cont.)

Test einiger Griffe auf der Roboterplattform:



Erweiterungsmöglichkeiten

Automatische Ausführung der Griffe auf TASER:

- Kamera-basierte Objekterkennung und Positionsbestimmung
- Trajektorie-Generierung
- Griff-Ausführung

Automatische Generierung von Objektmodellen:

- 3D Modell basierend auf Bilddaten
- Erkennung von Ähnlichkeiten/Wiederverwendung von Griffen
- bei unbekannten Objekten: Neuberechnung