# Reinforcement Learning

## Algorithmic Learning 64-360, Part 14

Jianwei Zhang

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

13/07/2011

# Contents

# Dynamic Programming

Content:

- Overview of a collection of classical solution methods for MDPs, called *dynamic programming*
- Demonstration of the application of DP — calculating value functions and therefore optimal *policies*
- Discussion about the effectiveness and usefulness of DP

This part is based on "Reinforcement Learning: An Introduction", Richard S. Sutton and Andrew G. Barto

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

# Dynamic Programming for Model-based Learning

If the dynamics of the environment is known, the class of Dynamic Programming methods can be used.
Dynamic Programming is a collection of approaches that can be used if a perfect model of the MDP's is available. In order to calculate the optimal policy, the Bellman Equations are embedded into an update function that approximates the desired value function $V$.
Three steps:

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

# Policy Evaluation (1)

*Policy Evaluation*: Calculate the state-value function $V^\pi$ for a given *policy* $\pi$.
Remember:
*state-value function for policy* $\pi$:

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

*Bellman-Equation for* $V^\pi$:

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

– a system of $|S|$ linear equations

# Policy Evaluation (2)

*Policy Evaluation* is a process of calculating the value-function $V^\pi$ for an arbitrary *policy* $\pi$ .
Based on the Bellman equation, an update rule can be created that calculates the approximated value-function $V_0, V_1, V_2, V_3, \ldots$

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

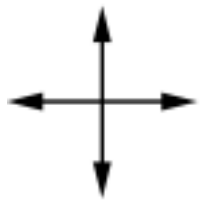
Based on the $k$-th approximation $V_k$ for each state $s$ successively the $k+1$-th approximation $V_{k+1}$ is calculated, thus the old value of $s$ is replaced by the new one, that has been calculated with the iteration rule based on the old values.
It can be shown that the sequence of the iterated value-functions $\{V_k\}$ converges to $V^\pi$, if $k \to \infty$.

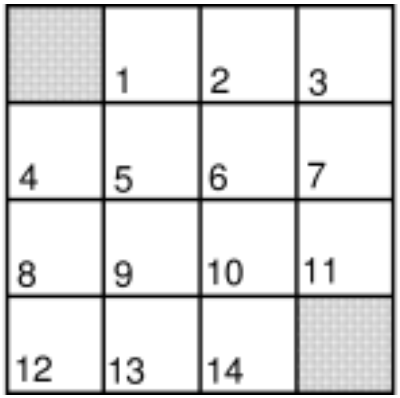
# Iterative Policy Evaluation

Input $\pi$, the *policy* to evaluate
Initialize $V(s) = 0$, for all $s \in S^+$
Repeat
$\Delta \leftarrow 0$
For every $s \in S$:
$v \leftarrow V(s)$
$V(s) \leftarrow \sum_a \pi(s,a) \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V_k(s')]$
$\Delta \leftarrow \max(\Delta, |v - V(s)|)$

until $\Delta < \theta$ (a small positive real number)
Output $V \approx V^\pi$

# Example: A small Gridworld

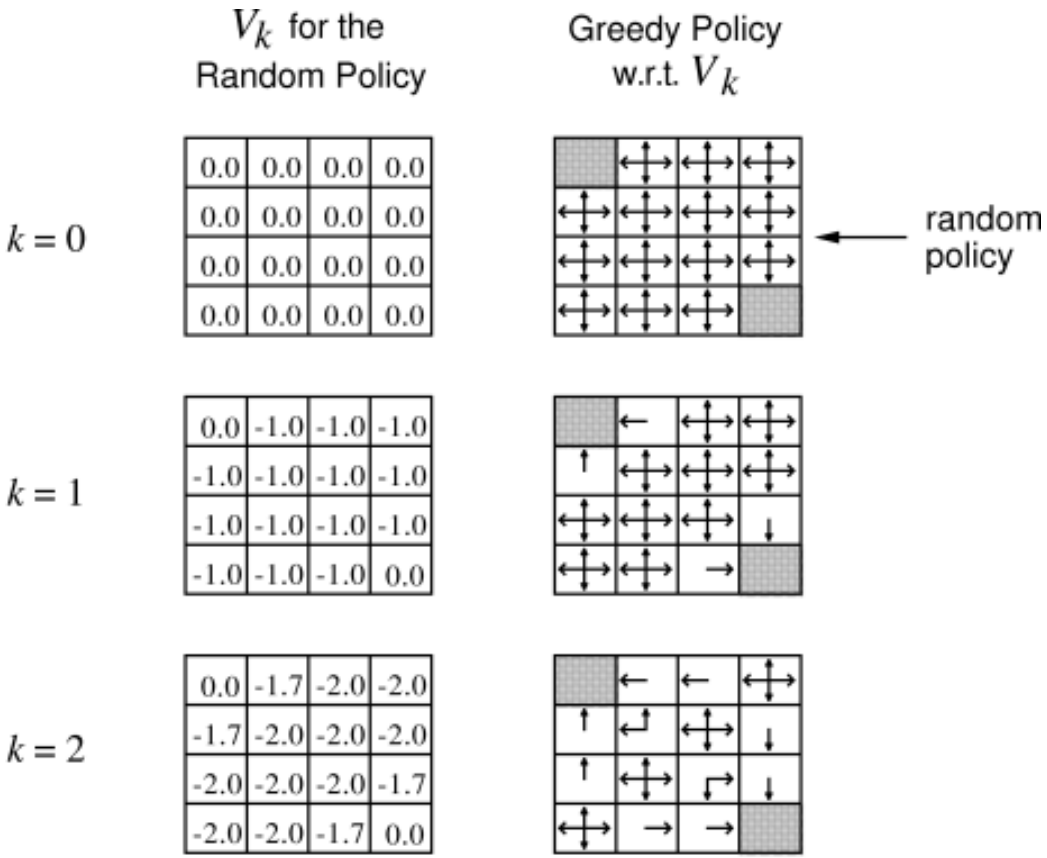



$r = -1$
on all transitions

- An episodic task (*undiscounted*)
- Non-terminal states: $1, 2, \ldots, 14$;
- One terminal state (twice, represented as a shaded square)
- Actions that would take the agent from the *Grid*, leave the state unchanged
- The *reward* is - 1 until the terminal state is reached

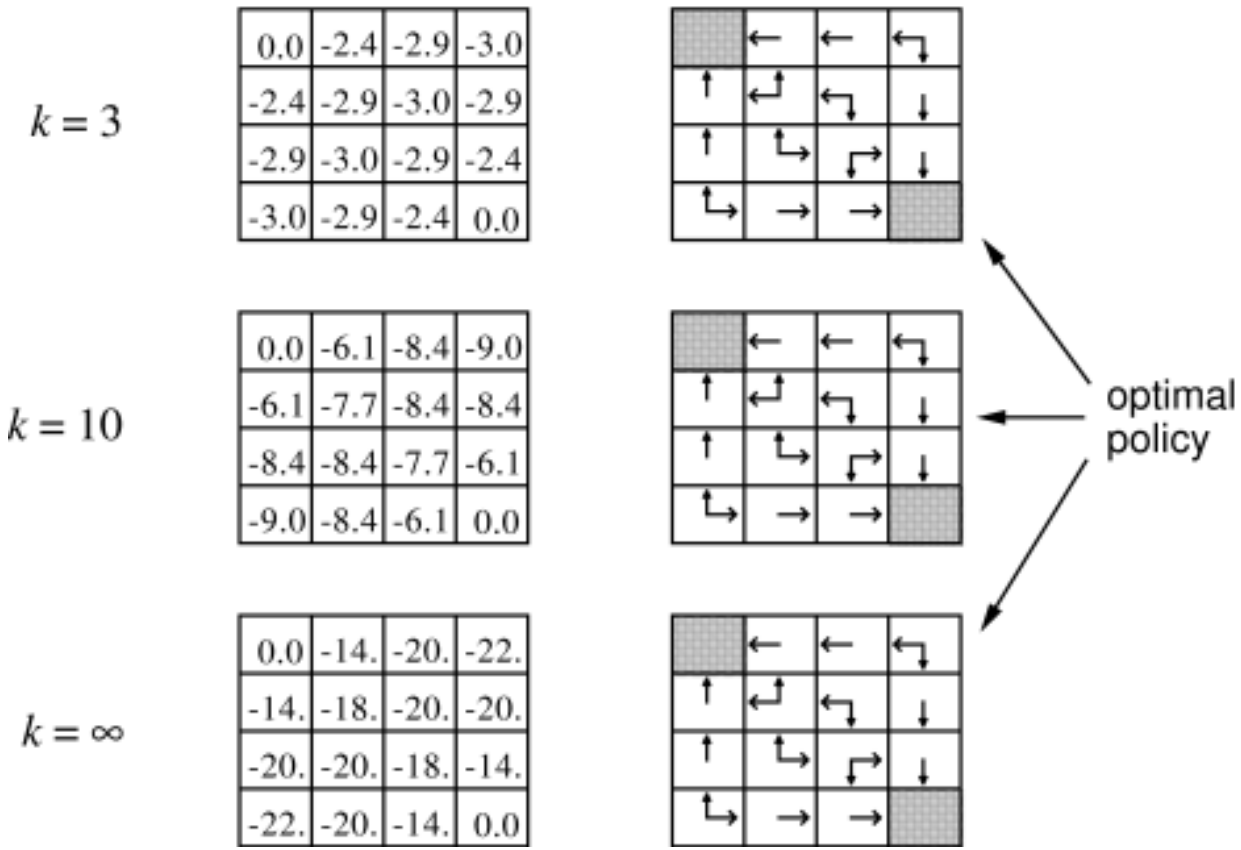
# Iterative Policy-Evaluation for the Gridworld (1)




left: $V_k(s)$ for the *random* policy $\pi$ (random moves)

right: moves according to the greedy policy $V_k(s)$

# Iterative Policy-Evaluation for the Gridworld (2)




In this example: The greedy Policy for $V_k(s)$ is optimal for $k \geq 3$.

# Policy Improvement (1)

We now consider the action value function $Q^\pi(s,a)$, when action $a$ is chosen in state $s$, and afterwards Policy $\pi$ is pursued:

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

In each state we look for the action that maximizes the action value function.
Hence a greedy policy $\pi'$ for a given value function $V^\pi$ is generated:

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

# Policy Improvement (2)

Suppose we have calculated $V^\pi$ for a deterministic *policy* $\pi$.
Would it be better to choose an action $a \neq \pi(s)$ for a given state?
If $a$ is chosen in state $s$, the value is:

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

It is better to switch to action $a$ in state $s$, if and only if

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

# Policy Improvement (3)

Perform this for all states, to get a new *policy* $\pi$, that is *greedy* in terms of $V^{\pi}$:

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

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

# Policy Improvement (4)

What if $V^{\pi'} = V^{\pi}$?
e.g. for all $s \in S$, $\quad V^{\pi'}(s) = \max_a \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V^{\pi}(s')]$?

Notice: this is the optimal Bellman-Equation.

Therefore $V^{\pi'} = V^*$ and both $\pi$ and $\pi'$ are optimal policies.

# Iterative Methods

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

$$\Uparrow \text{ an “iteration”}$$

An iteration comprises one *backup*-operation for each state.

A *full-policy* evaluation-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 (1)

If Policy Improvement and Policy Evaluation are performed alternating, this means that the policy $\pi$ is improved with a fixed value function $V^{\pi}$, and then the corresponding value function $V^{\pi'}$ is calculated based on the improved policy $\pi'$.
Afterwards again policy improvement is used, to get an even better policy $\pi''$, and so forth ...:

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

Here $\xrightarrow{\text{PI}}$ stands for performing policy improvement and $\xrightarrow{\text{PE}}$ for policy evaluation.

# Policy Iteration (2)

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

policy-evaluation $\Uparrow$ $\Uparrow$ policy-improvement "greedification"

# Policy Iterationi (3)

1. initialization
   $V(s) \in \Re$ and $\pi(s) \in A(s)$ arbitrarily for all $s \in S$

2. policy-evaluation
   Repeat
   $\Delta \leftarrow 0$
   for every $s \in S$:
   $v \leftarrow V(s)$
   $V(s) \leftarrow \sum_{s'} P_{ss'}^{\pi(s)}[R_{ss'}^{\pi(s)} + \gamma V(s')]$
   $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small real positive number)

# Policy Iteration (4)

3. policy-improvement

*policy-stable* $\leftarrow$ true
for every $s \in S$:
  $b \leftarrow \pi(s)$
  $\pi(s) \leftarrow \arg\max_a \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V(s')]$
  if $b \neq \pi(s)$, then *policy-stable* $\leftarrow$ false
If *policy-stable*, then stop; else goto 2

# Value Iteration

Remember the **full policy evaluation-backup**

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

Here the **full value iterations-backup** is:

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

# Value Iteration

Initialize $V$ arbitrarily, e.g. $V(s) = 0$, for all $s \in S^+$
repeat
  $\Delta \leftarrow 0$
  For every $s \in S$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_a \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V(s')]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive real number)

Output is a deterministic policy $\pi$ with
  $\pi(s) = \arg\max_a \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V(s')]$

# Asynchronous Dynamic Programming

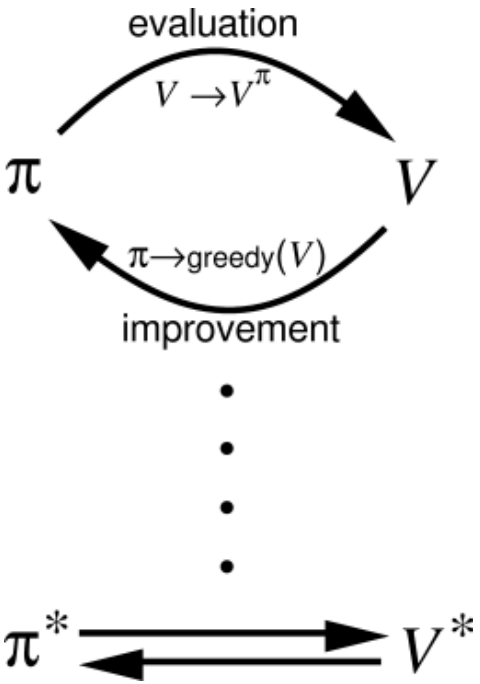
- All DP-methods described so far require complete iterations over the entire set of states.
- Asynchronous DP does not perform complete iterations, instead, it works like this:

Repeat until the convergence criterion is met:

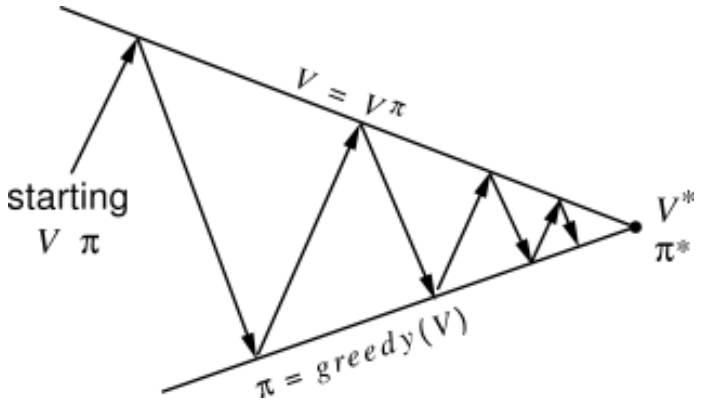
- Pick a random state and apply the appropriate *backup*.
- Still requires much computation, but does not use hopeless long loops
- Can states for the application of the backup been selected intelligently? YES: the experience of an agent can serve as a guide.

# Generalized Policy Iteration (GPI)

*Generalized Policy Iteration* (GPI):
Every interaction between policy evaluation and policy improvement, independent from their granularity.




A geometric metaphor
for the convergence of GPI:



# Efficiency of DP

- Finding an optimal *policy* is polynomial with the number of states...
- BUT, the number of states is often extremely high, e.g. grows often exponentially with the number of state-variables (what Bellman called "the curse of dimensionality").
- In practice, the classical DP can be applied to problems with a few million states.
- The asynchronous DP can be applied to larger problems and is suitable for parallel computation.
- It is surprisingly easy to find MDPs, where DP methods can not be applied.

# Summary: Dynamic Programming

- Policy Evaluation: *Backups* without maximum
- Policy Improvement: form a *greedy policy*, even if only locally
- Policy Iteration: alternate the above two processes
- Value Iteration: *backups* with maximum
- Complete *Backups* (that are later compared to example-Backups)
- Generalized Iteration (GPI)
- Asynchronous DP: a method to avoid complete iterations
- *Bootstrapping*: Update approximations with different approximations.

# Q-Learning

- Q-Function
- Q-Learning algorithm
- Convergence
- Example: GridWorld
- Experience-Replay

# The Q-Function

We define the Q-function as follows:

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

So $\pi^*$ can be written as

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

I.e.: The optimal policy can be learned, as $Q$ is learned, even if $r$ and $\delta$ are unknown.

# Q-Learning Algorithm (1)

$Q$ and $V^*$ are closely related:

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

This allows the re-definition of $Q(s, a)$:

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

This recursive definition of $Q$ is the basis for an algorithm that approximates $Q$ iteratively.

# Q-Learning Algorithm (2)

Let $\hat{Q}$ the current approximation for $Q$. Let $s'$ be the new state after execution of the chosen action and let $r$ be the obtained reward.
Based on the recursive definition of $Q$ the iteration-rule can be written as:

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

$\Rightarrow$:

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

# Q-Learning Algorithm (3)

The algorithm:

1. Initialize all table entries of $\hat{Q}$ to 0.
2. Determine the current state $s$.
3. Loop
   - Choose action $a$ and execute it.
   - Obtain reward $r$.
   - Determine new state $s'$.
   - $\hat{Q}(s,a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s',a')$
   - $s \leftarrow s'$.

   Endloop

# Convergence of Q-Learning

Theorem: If the following conditions are met:

- $|r(s,a)| < \infty, \forall s, a$
- $0 \leq \gamma < 1$
- Every $(s,a)$-pair is visited infinitely often

Then $\hat{Q}$ converges to the correct Q-function.

# Continuous Systems

The Q function of very large or continuous state spaces cannot be represented by an explicit table.
Instead function-approximation-algorithms are used, e.g a neural network or B-splines.
The neural network uses the output of the Q-learning algorithm as training examples.
Convergence is then no longer guaranteed!

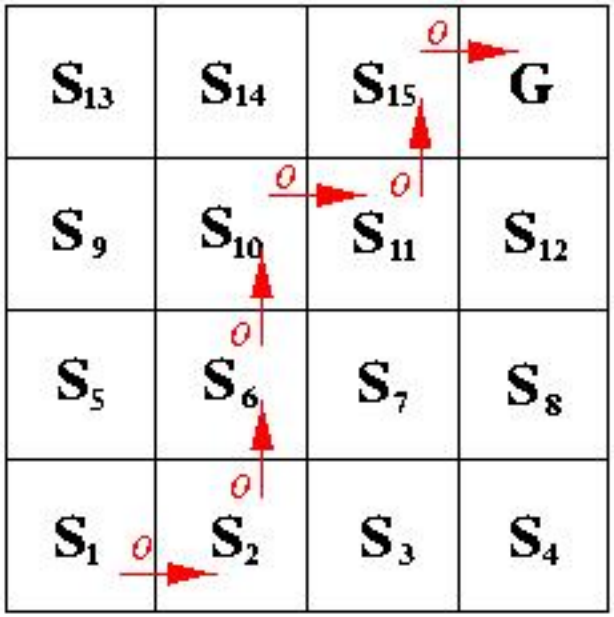
# Example: GridWorld (1)

given: $m \times n$-Grid

- $S = \{(x, y) | x \in \{1, \cdots, m\}, y \in \{1, \cdots, n\}\}$
- $A = \{up, down, left, right\}$
- $r(s, a) = \begin{cases} 100, \text{if } \delta(s, a) = \text{Goalstate} \\ 0, else. \end{cases}$
- $\delta(s, a)$ determines the following state based on the direction given with $a$.

# Example: GridWorld (2)

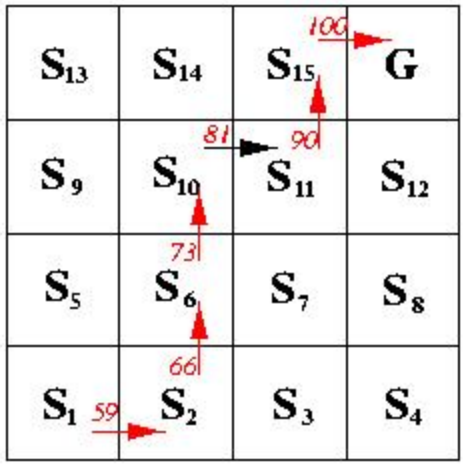
Example of a path through a state space:




The numbers on the arrows show the current values of $\hat{Q}$.

# Example: GridWorld (3)

Progression of the $\hat{Q}$-values:




$$\hat{Q}(S_{11}, up) = r + \gamma \max_{a'} \hat{Q}(s', a') = 0 + 0.9 * 100 = 90$$

$$\hat{Q}(S_{10}, right) = 0 + 0.9 * 90 = 81$$

...

# Q-Learning - open Questions

- often not met: Markov assumption, visibility of states
- continuous state-action spaces
- generalization concerning the state and action
- compromise between “exploration” and “exploitation”
- generalization of automatic evaluation

# Ad-hoc Exploration-Strategies

- Too extensive "exploration" means, that the agent is acting aimlessly in the usually very large state space even after a long learning period. Also areas are investigated, that are not relevant for the solution of the task.
- Too early "exploitation" of the learned approximation of the $Q$-function probably causes that a sub-optimal, i.e. longer path through the state space, that has been found by occasion establishes and the optimal solution will not be found.

There are:

- "greedy strategies"
- "randomized strategies"
- "interval-based techniques"

# Acceleration of Learning

Some ad-hoc methods:

- Experience Replay
- Backstep Punishment
- Reward Distance Reduction
- Learner-Cascade

# Experience Replay (1)

A path through the state space is considered as finished as soon as $G$ is reached.
Now assume that during the $Q$-learning the path is repeatedly chosen.

Often new learning steps are much more cost- and time-consuming than internal repetitions of previously stored Learning steps. For these reasons it makes sense to store the learning steps and repeat them internally. This method is called **Experience Replay**.

# Experience Replay (2)

An *experience* $e$ is a tuple

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

with $s, s' \in S$, $a \in A$, $r \in \mathbb{R}$. $e$ represents a learning step, i.e. a state of transition, where $s$ the initial state, $s'$ the goal state, $a$ the action, which led to the state transition, and $r$ the reinforcement signal that is obtained.

A *learning path* is a series $e_1 \dots e_{L_k}$ of experiences ($L_k$ is the length of the $k$-th learning path).

# Experience Replay (3)

The ER-Algorithm:

for $k = 1$ to $N$
    for $i = L_k$ down to 1
        update( $e_i$ from series $k$)
    end for
end for

# Experience Replay (4)

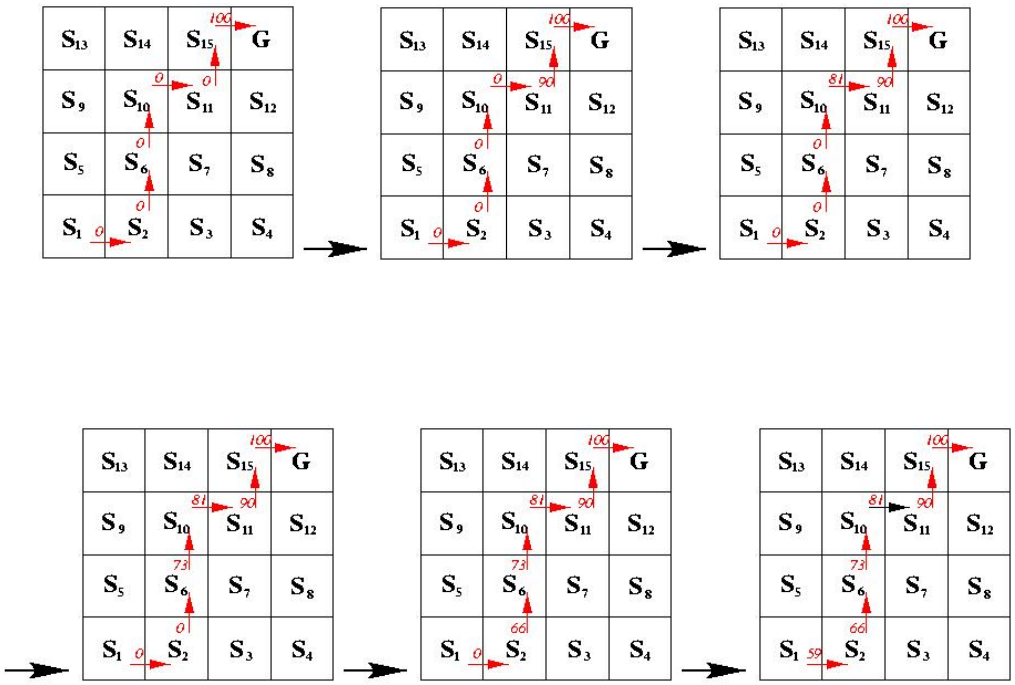
Advantages:

- Internal repetitions of stored learning steps usually cause far less cost than new learning steps.
- Internally a learning path can be used in the reverse direction, thus the information is spread faster.

- If learning paths cross, they can "learn form each other", i.e. exchange information. Experience Replay makes this exchange regardless of the order in which the learning path was firstly executed.

# Experience Replay - Example

Progression of the $\hat{Q}$-values when the learning path is repeatedly run through:



# Backstep Punishment

Usually an exploration strategy is needed, that ensures a straightforward movement of agents through the state space. Therefore backsteps should be avoided.
An useful method seems to be, that in case the agent chooses a backstep, the agent may carry out this step but an *artificial, negative reinforcement-signal* is generated.
Compromise between the "dead-end avoidance" and "fast learning".

In context of a goal-oriented learning an extended reward function could look as follows:

$$r_{BP} = \begin{cases} 100 & \text{if transition to goal state} \\ -1 & \text{if backstep} \\ 0 & \text{else} \end{cases}$$

# Reward Distance Reduction

The reward function could perform a more intelligent assessment of the actions. This presupposes knowledge of the structure of the state space. If the encoding of the target state is known, then it can be a be good strategy to reduce the Euclidean distance between the current state and the target state.
The reward functions can be extended the way that actions that reduce the Euclidean distance to the target state, get a higher reward. (*reward distance reduction*, *RDR*):

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

where $\vec{s}$, $\vec{s}'$ and $\vec{s}_g$ are the vectors that encode the current state, the following state, and the goal state.

# Learner-Cascade

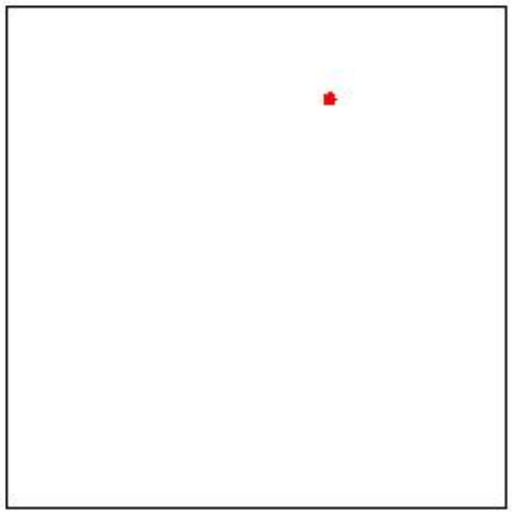
The accuracy of positioning depends on how fine the state space is divided.
On the other hand the number of states increases with increasing fineness of the discretization and therefore also the effort of learning increases.

Before the learning procedure a trade-off between effort and accuracy of positioning has to be chosen.

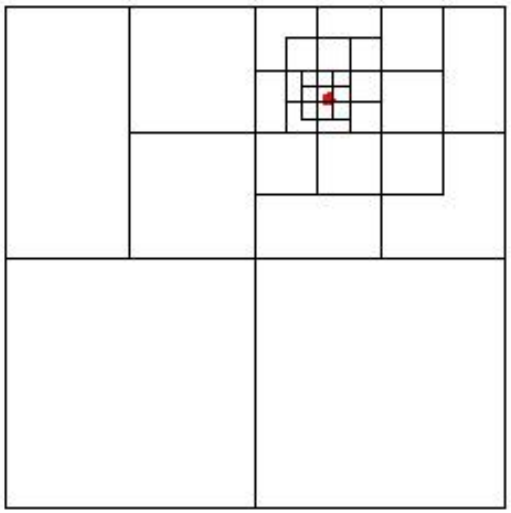
# Learner-Cascade - Variable Discretization

An example state space without discretization (a) with variable discretization (b)

a)



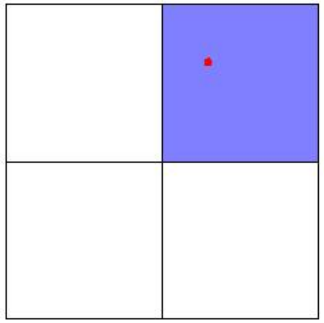
b)

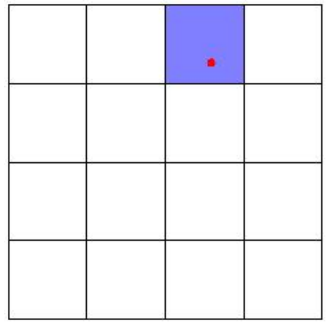


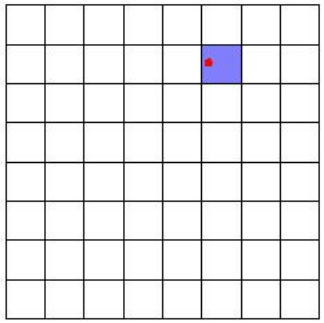
This requires knowledge of the structure of the state space.

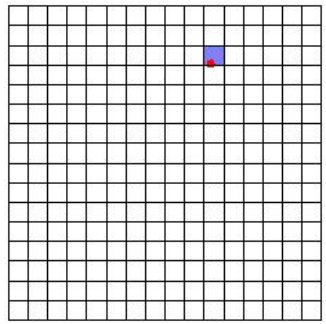
# Learner-Cascade - *n*-Stage Learner-Cascade

Divisions of the state space of a four-stage learner cascade:

1. learner: 

2. learner: 

3. learner: 

4. learner: 

# Q-Learning: Summary

- Q-Function
- Q-Learning algorithm
- convergence
- acceleration of learning
- examples

# Path Planning with Policy Iteration (1)

Policy iteration is used to find a path between start- and goal-configuration.
The following sets need to be defined to solve the problem.

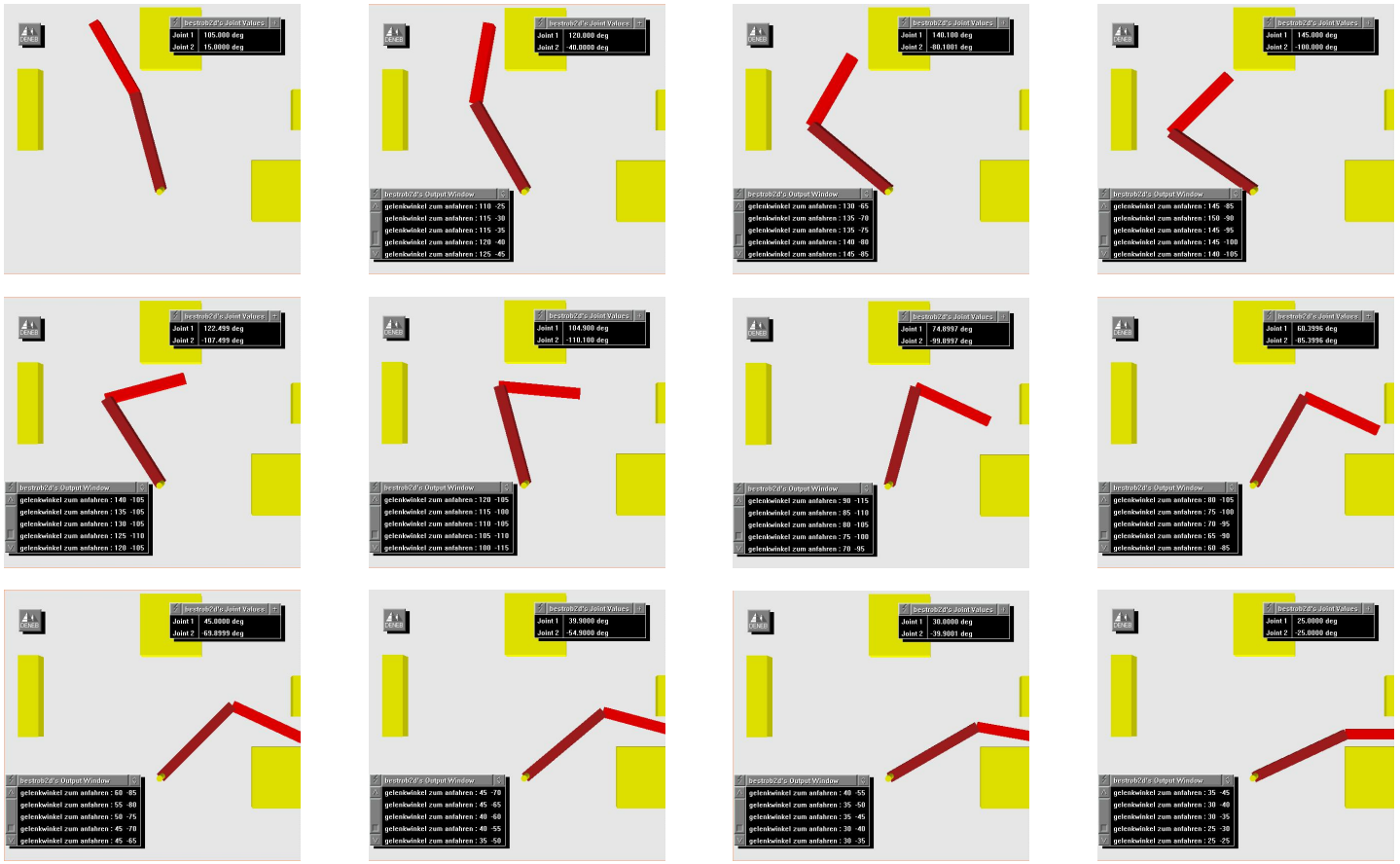
- The state space $S$ is the discrete configuration space, i.e. every combination of joint angles $(\theta_1, \theta_2, \ldots, \theta_{Dim})$ is exactly one state of the set $S$ except the target configuration
- The set $A(s)$ of all possible actions for a state $s$ includes the transitions to neighbor-configurations in the K-space, so one or more joint angles differ by $\pm Dis$. Only actions $a$ in $A(s)$ are included, that do not lead to K-obstacles and do not exceed the limits of the K-space

# Path Planning with Policy Iteration (2)

- Let $R^a_{ss'}$ be the reward function:
  $R^a_{ss'} =$ **Reward_not_in_goal** $\forall s \in S, a \in A(s), s' \in S$ and
  $R^a_{ss'} =$ **Reward_in_goal** $\forall s \in S, a \in A(s), s' = s_t$.
  Only if the target state is reached a different reward value is generated. For all other states there is the same reward.
- Let the policy $\pi(s, a)$ be deterministic, i.e. there is exactly one $a$ with $\pi(s, a) = 1$
- A threshold $\Theta$ needs to be chosen, where the policy evaluation terminates
- For the problem the Infinite-Horizon Discounted Model is the best choice, therefore $\gamma$ needs to be set accordingly.

# 2-Joint Robot

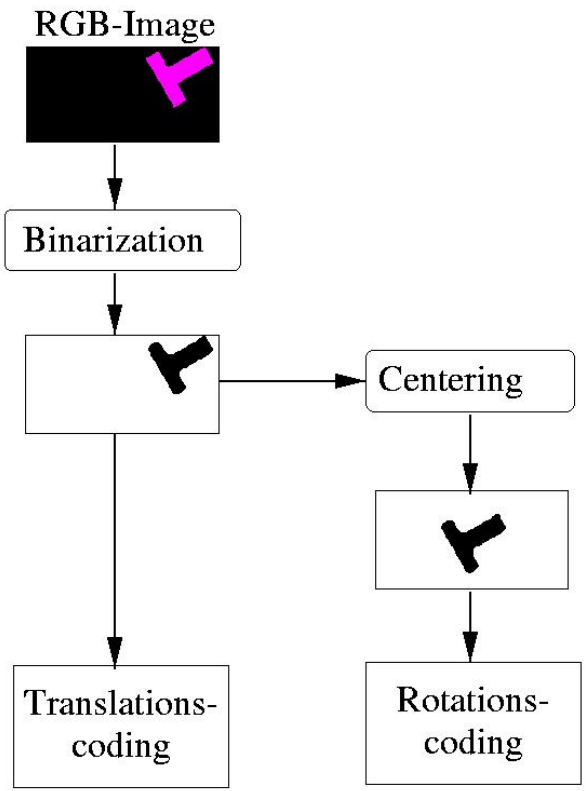
The found sequence of motion for the 2-joint robot (left to right, top to bottom):



# Grasping with RL: Partition of DOFs

Normally a robot arm needs six DOFs to grasp an object from any position and orientation.
To grasp virtually planar objects, we suppose that the gripper is perpendicular to the table and its weight is known.
There still remain three DOFs to control the robot arm: parallel to the table-plane $(x, y)$ and the rotation around the axis perpendicular to the table($\theta$).

# Grasping with RL: Partition of DOFs (2)




Control of $x, y, \theta$. The pre-processed images will be centered for the detection of the rotation.

# Grasping with RL: Partition of DOFs (3)

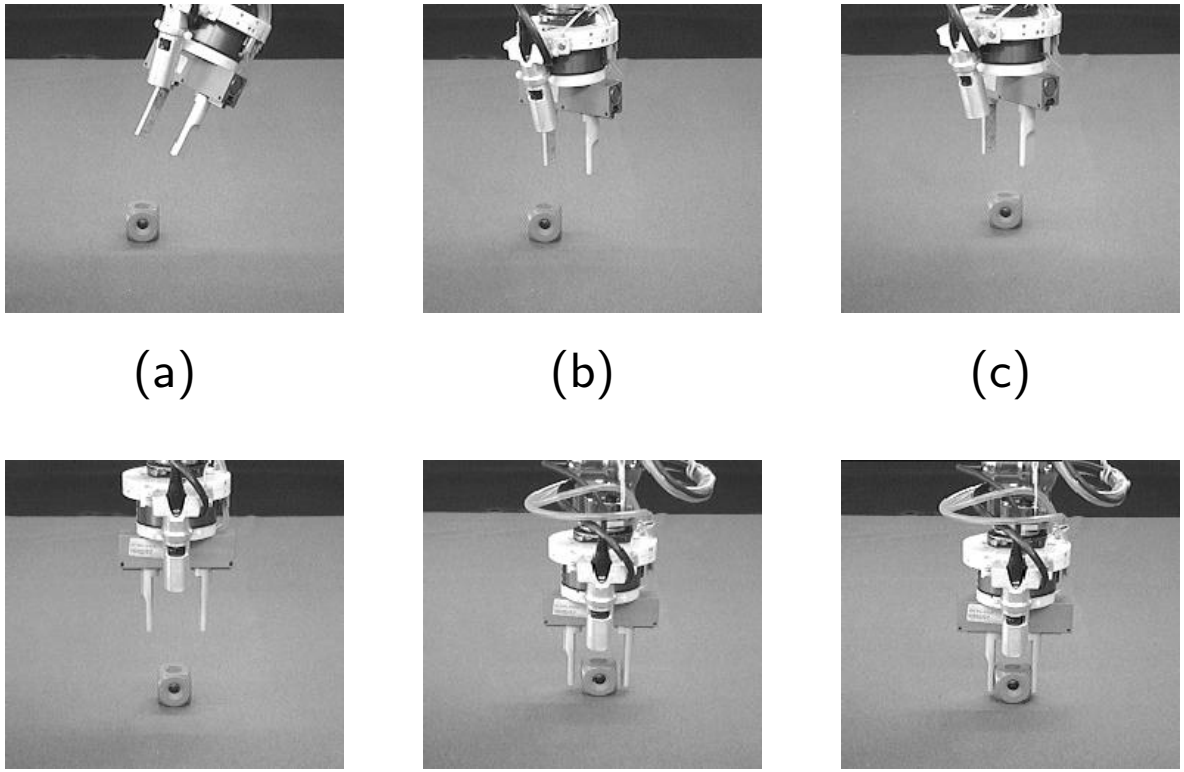
To achieve a small *state space*, learning will be distributed to two learners:
one for the *translational* movement on the plane,
the other one for the *rotational* movement.
The *translation-learner* can choose four actions (two in $x$- and two in $y$-direction).
The *rotation-learner* can choose two actions (rotation clockwise/counterclockwise).
The partition has advantages compared to a monolithic learner:

- The state space is much smaller.
- The state-encoding is designed the way that the state-vectors contain only the relevant information for the corresponding learner.

# Grasping with RL: Partition of DOFs (4)

In practice, the two learners will be used alternatingly. Firstly the *translation-learner* will be run in long learning-steps, until it has reached the goal defined in its state-encoding.
Then the *translation-learner* is replaced by the *rotation-learner*, which is also used in long learning-steps until it reaches its goal.
At this time it can happen, that the *translation-learner* is disturbed by the *rotation-learner*. Therefore the *translation-learner* is activated once again. The procedure is repeated, until both learners reach their goal state. This state is the common goal state.

# Grasping with RL: Partition of DOFs (5)



(a) (b) (c)

(d) (e) (f)

Position and orientation-control with 6 DOFs in four steps.

# Grasping with RL: Partition of DOFs (6)

To use all six DOFs, additional learners need to be introduced.

1. The first learner has two DOFs and its task is that the object can be looked upon from a defined perspective. For a plane table this typically means, that the gripper is positioned perpendicularly to the surface of the table (a $\rightarrow$ b).
2. Apply the $x/y$ learner (b $\rightarrow$ c).
3. Apply the $\theta$ rotation learner(c $\rightarrow$ d).
4. The last learner controls the height and corrects the $z$-coordinate (d $\rightarrow$ e).

# Visually Guided Grasping using Self-Evaluative Learning

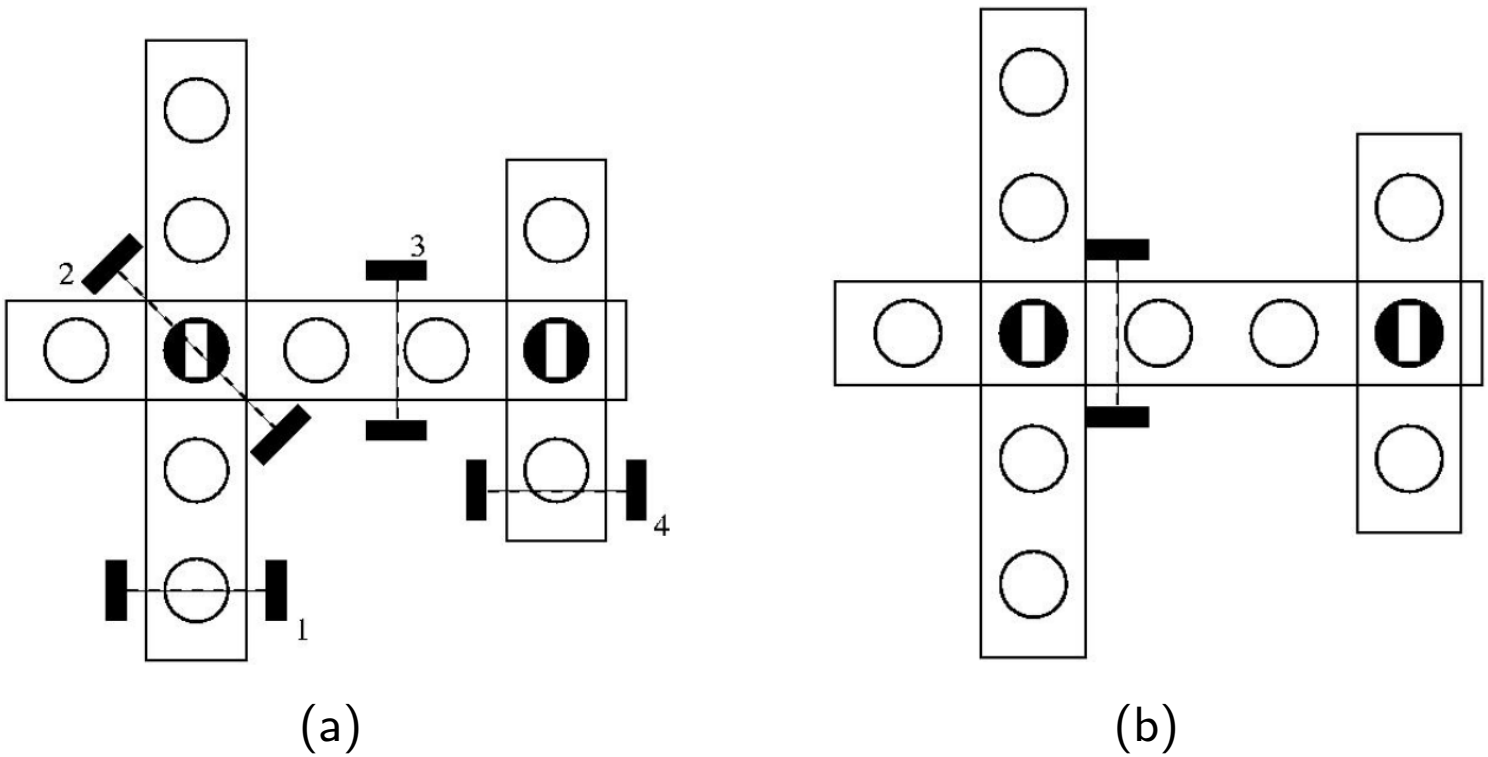
**Grip is optimal with respect to local criteria:**

- The fingers of the gripper can enclose the object at the gripping point
- No slip occurs between the fingers and object

**Grip is optimal with respect to global criteria:**

- No or minimal torque on fingers
- Object does not slip out of the gripper
- The grasp is stable, i.e. the object is held rigidly between the fingers

# Local and Global Criteria



# Two approaches

**One learner:**

- The states consist of a set of $m + n$ local and global properties: $s = (f_{l_1}, \ldots, f_{l_m}, f_{g_1}, \ldots, f_{g_n})$.
- The learner tries to map them to actions $a = (x, y, \phi)$, where $x$ and $y$ are translational components in $x$- and $y$-direction and $\phi$ is the rotational component around the approach vector of the gripper.

**Two learners:**

- The states for the first learner only supply the local properties $s = (f_{l_1}, \ldots, f_{l_m})$.
- The learner tries to map them to actions, that only consist of a rotational component $a = (\phi)$.
- The second learner tries to map states of global properties $s = (f_{g_1}, \ldots, f_{g_n})$ to actions concerning the translational component $a = (x, y)$.

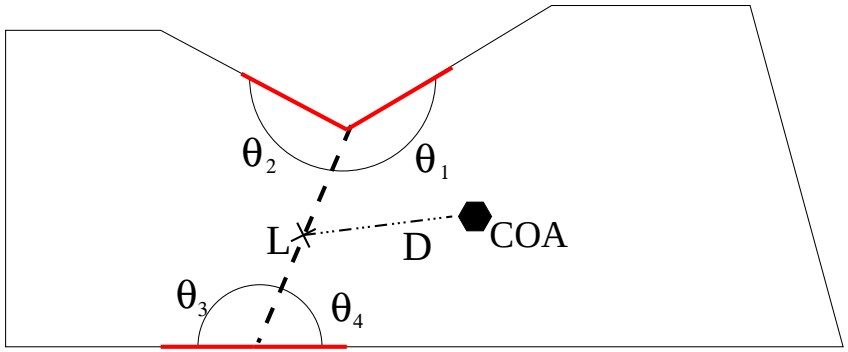
# Setup

- Two-component learning system:

| **1. orientation learner** | **2. position learner** |
|---|---|
| operates on local criteria | operates on global criteria |
| equal for every object | different for each new object |

- Use of Multimodal sensors:
  - Camera
  - Force / torque sensor
- Both learners work together in the perception-action cycle.

# State Encoding

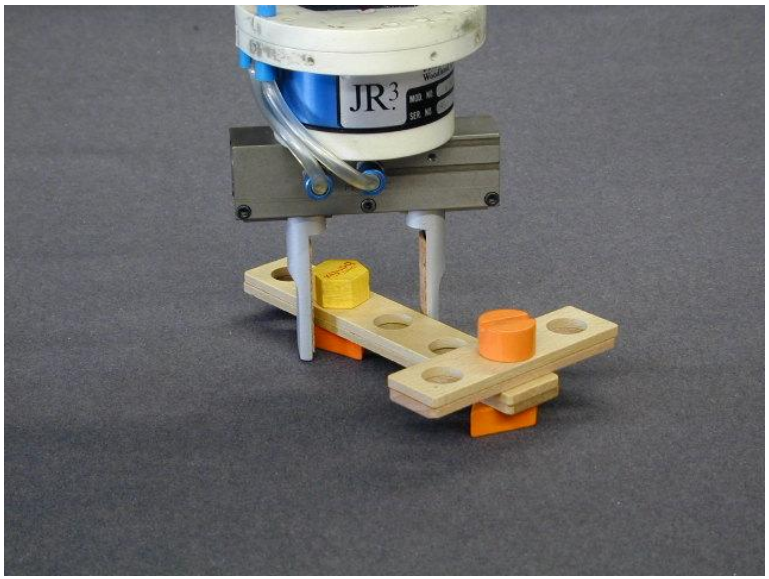


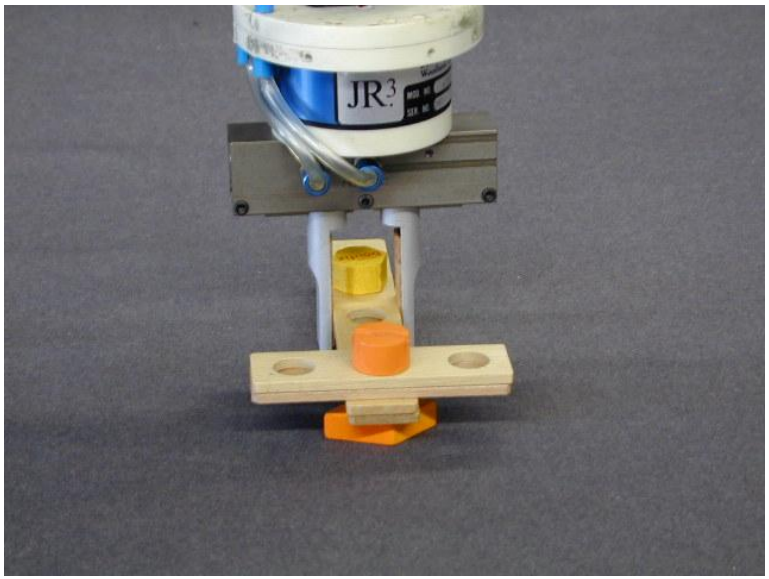

The orientation Lerner uses length $L$ as well as the angles $\Theta_1, \ldots, \Theta_4$, while the position learner uses the distance $D$ between the center of the gripper-line and the optical center of gravity of the object.

# Measures for Self-Evaluation in the Orientation Learner

**Visual feedback of the grasp success:**

Friction results in rotation or misalignment of the object.

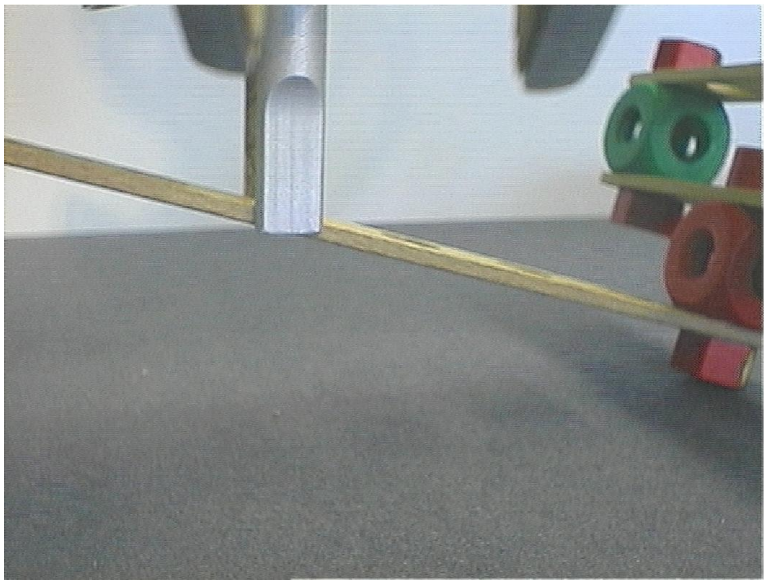


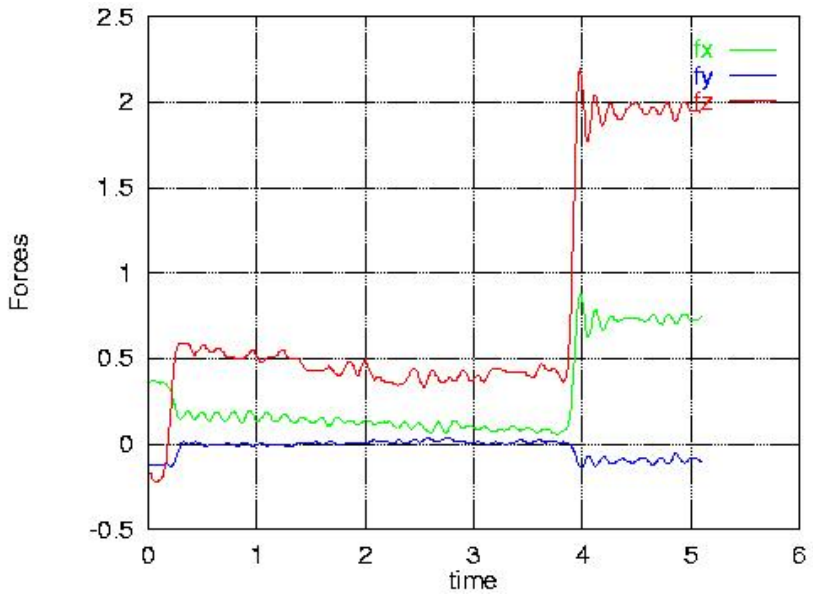



# Measures for Self-Evaluation in the Position Learner (1)

**Feedback using force torque sensor:**

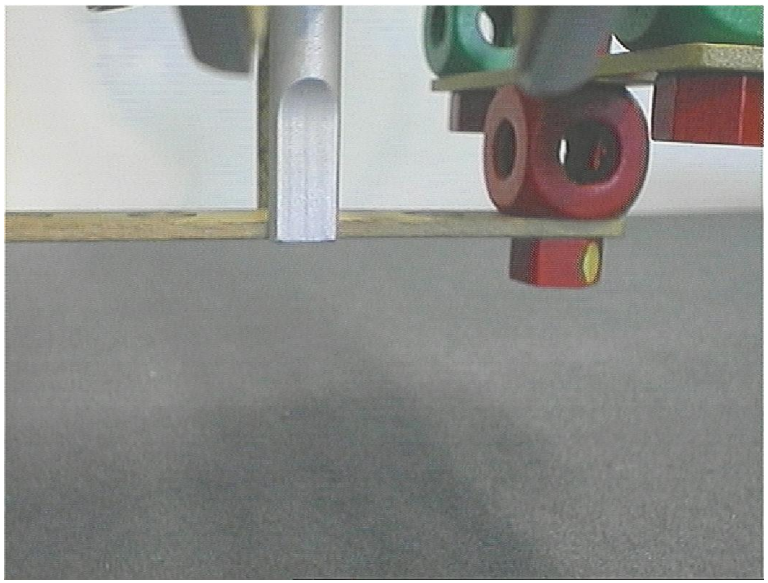
Unstable grasp – analyzed by force measurement

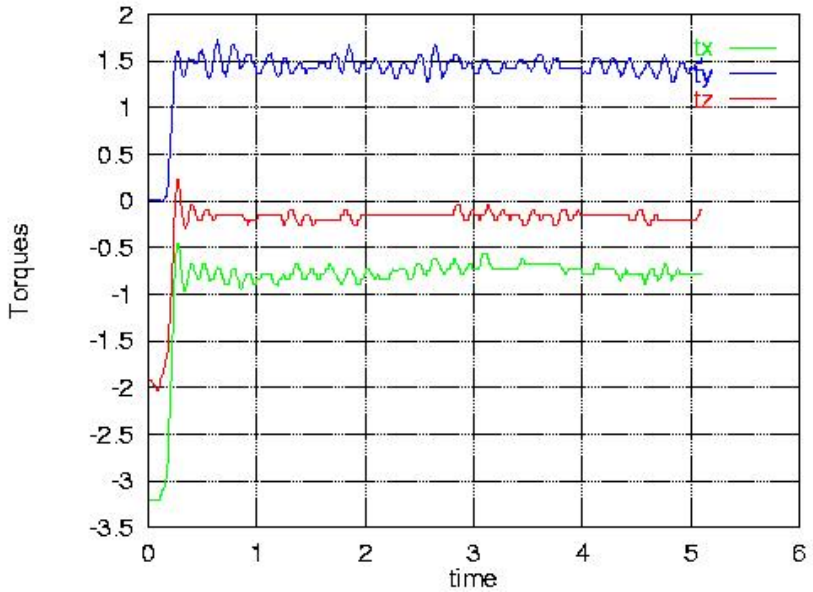




# Measures for Self-Evaluation in the Position Learner (2)

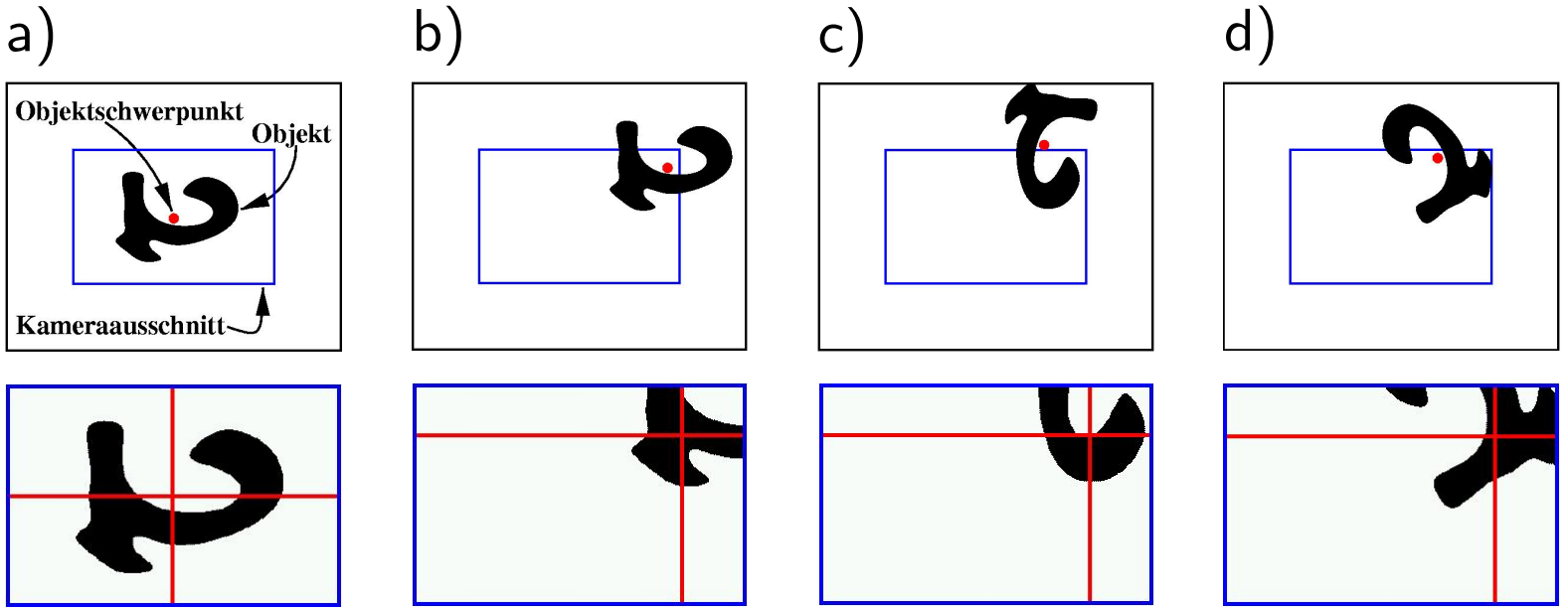
## Suboptimal grasp – analyzed by torques





# The Problem of Hidden States

Examples for incomplete state information:

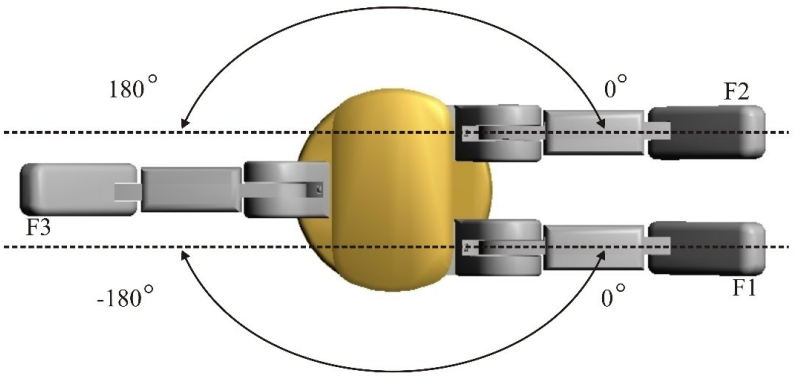


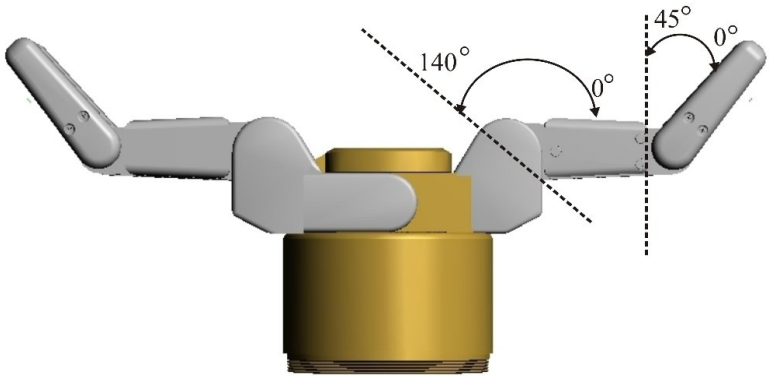
# Grasping mit RL: Barrett-Hand

- learn to grasp everyday objects with artificial robot hand
- reinforcement-learning process based on simulation
- find as many suitable grasps as possible
- support arbitrary type of objects
- efficiency
  - memory usage
  - found grasps/episodes

# BarretHand BH-262

- 3-finger robotic hand
- open/close each finger independently
- variable spread angle
- optical encoder
- force measurement

# Applied model

States:

- pose of gripper to object
- spread angle of hand
- grasp tried yet

Actions:

- translation (x-axis, negative y-axis, z-axis)
- rotation(roll-axis, yaw-axis, pitch-axis)
- alteration of spread-angle
- grasp-execution

# Applied model (cont.)

Action-Selection:

- $\epsilon$-greedy (highest rated, with probability $\epsilon$ random)

Reward-Function:

- reward for grasps depend on stability
- stability is evaluated by wrench-space-calculation (GWS) (introduced 1992 by Ferrari and Canny)
- small negative reward if grasp unstable
- big negative reward if object is missed

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

# Learning Strategy

Problem: The state-space is extremely large

- TD-($\lambda$)-algorithm
- learning in episodes
- episode ends
  - after fixed number of steps
  - after grasp trial
- Q-table built dynamically
- states are only included if they occur

# Automatic Value Cutoff

Problem: There exist many terminal states (grasp can be executed every time)

- instable grasps are tried several times
- agent gets stuck in local minimum
- not all grasps are found

$\Longrightarrow$ No learning at the end of an episode - waste of computing time

This can not simply be solved by adapting RL-parameters.

# Automatic Value Cutoff (cont.)

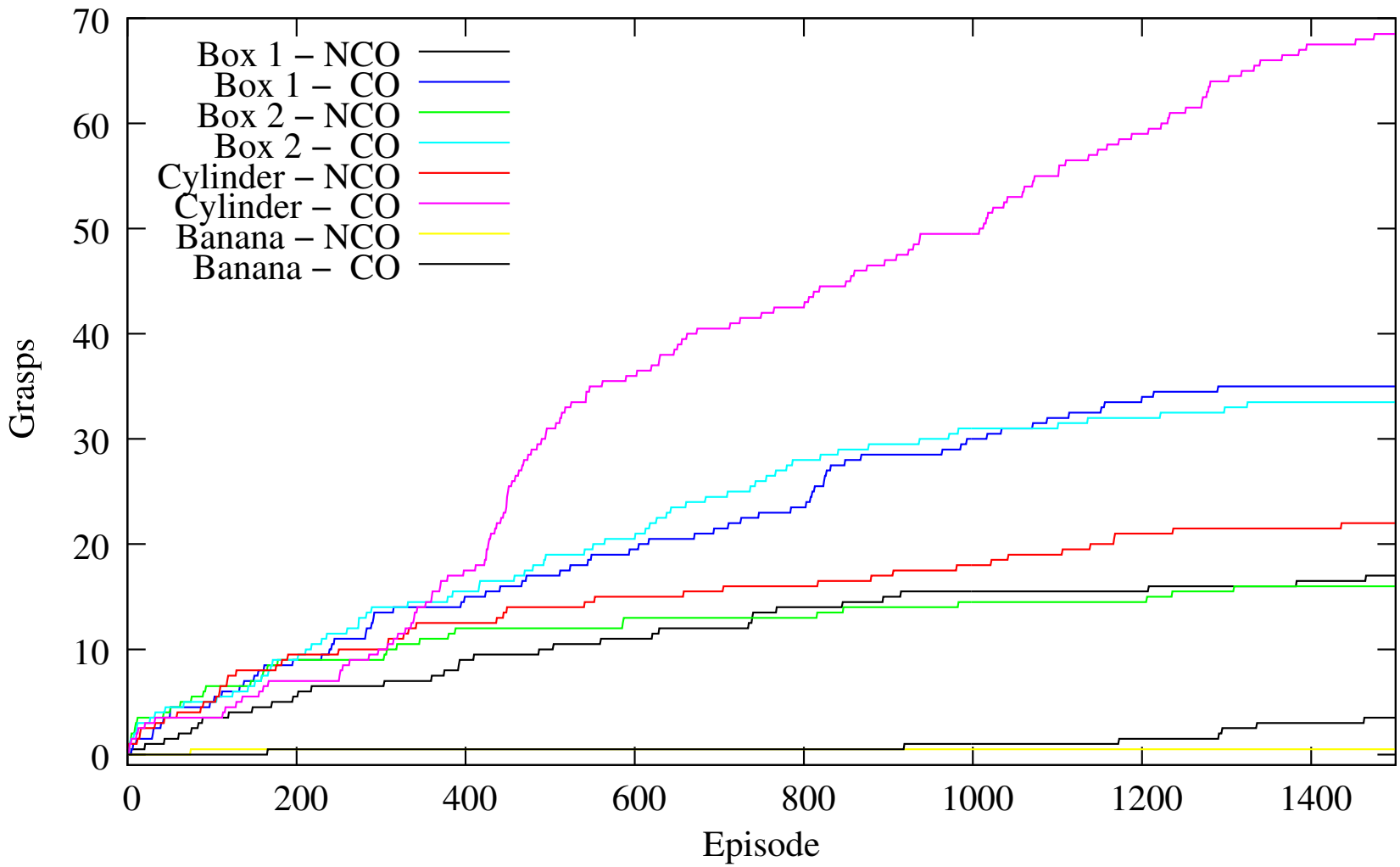
Automatic Value Cutoff: remove actions

- leading to instable grasps (if reward negative)
- that have been evaluated sufficiently

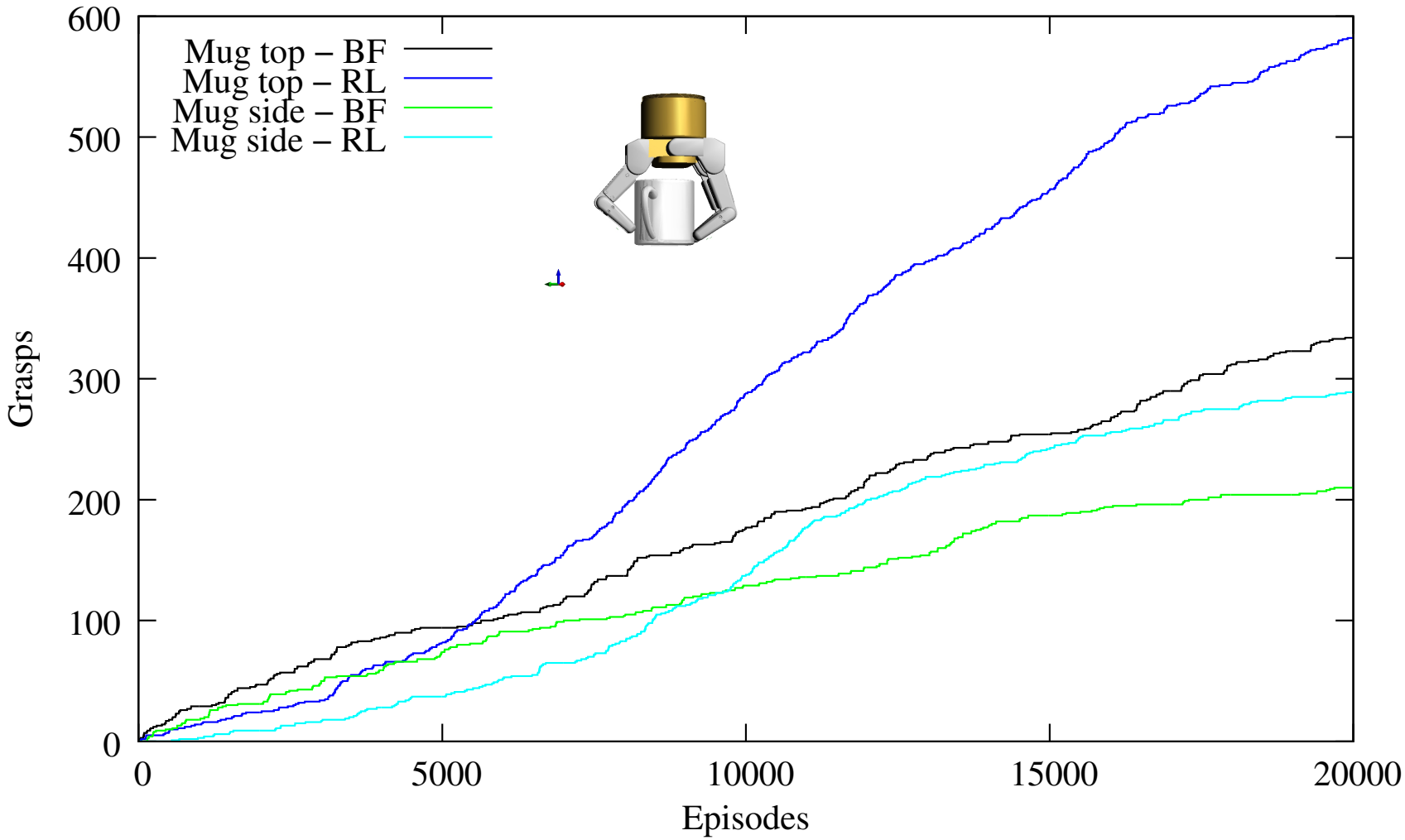
$$Q(s,a) \leftarrow \begin{cases} Q(s,a) + \alpha \left[ r + \gamma Q(s',a') - Q(s,a) \right] \\ \quad \text{if } 0 \leq Q(s,a) < r * \beta \\ \\ \text{remove } Q(s,a) \text{ from } Q \text{ otherwise} \end{cases}$$

with $0 \leq \beta \leq 1$. (we had good results with $\beta = 0.95$)

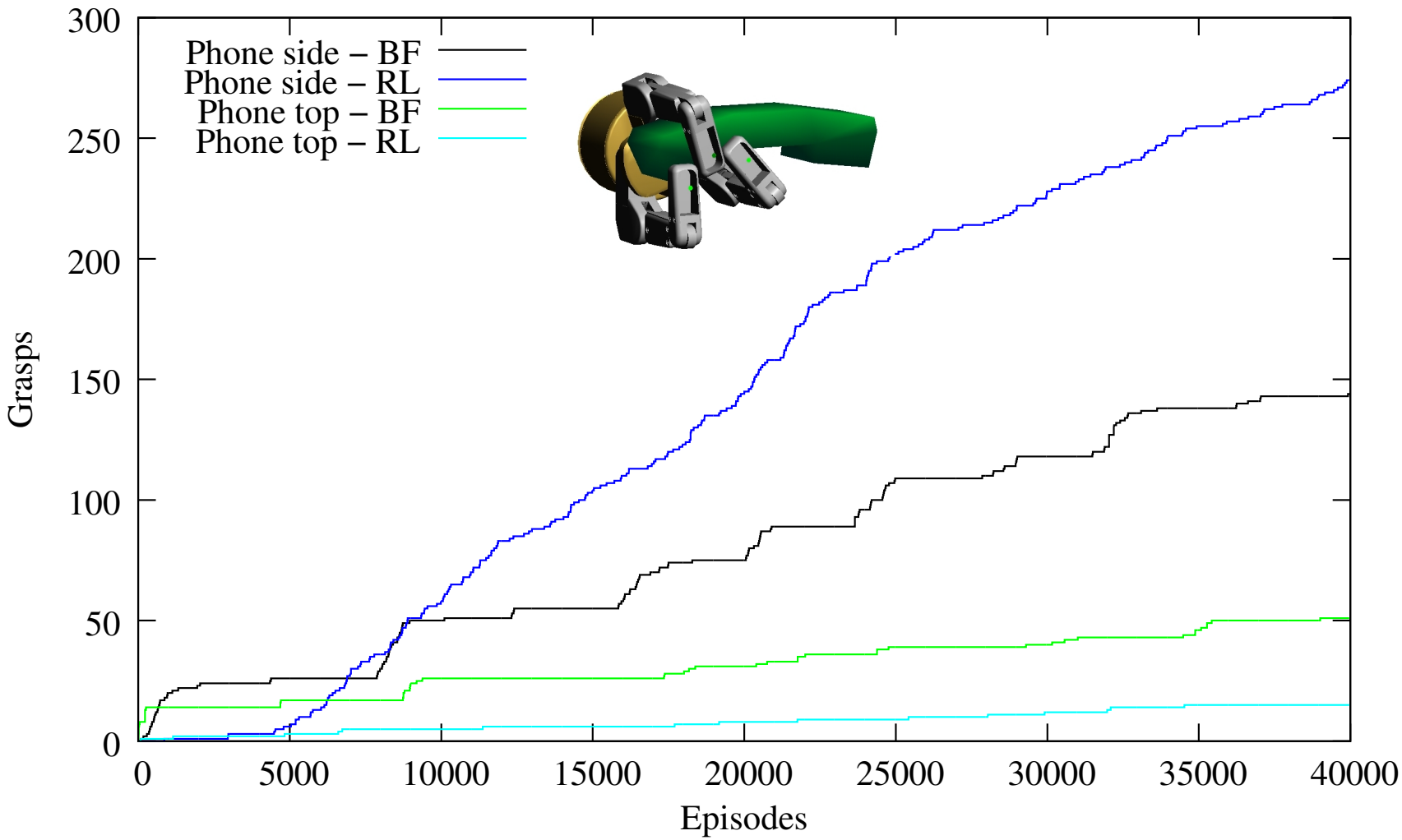
# Automatic Value Cutoff vs. no Cutoff



# Reinforcement Learning vs. Brute Force: Mug

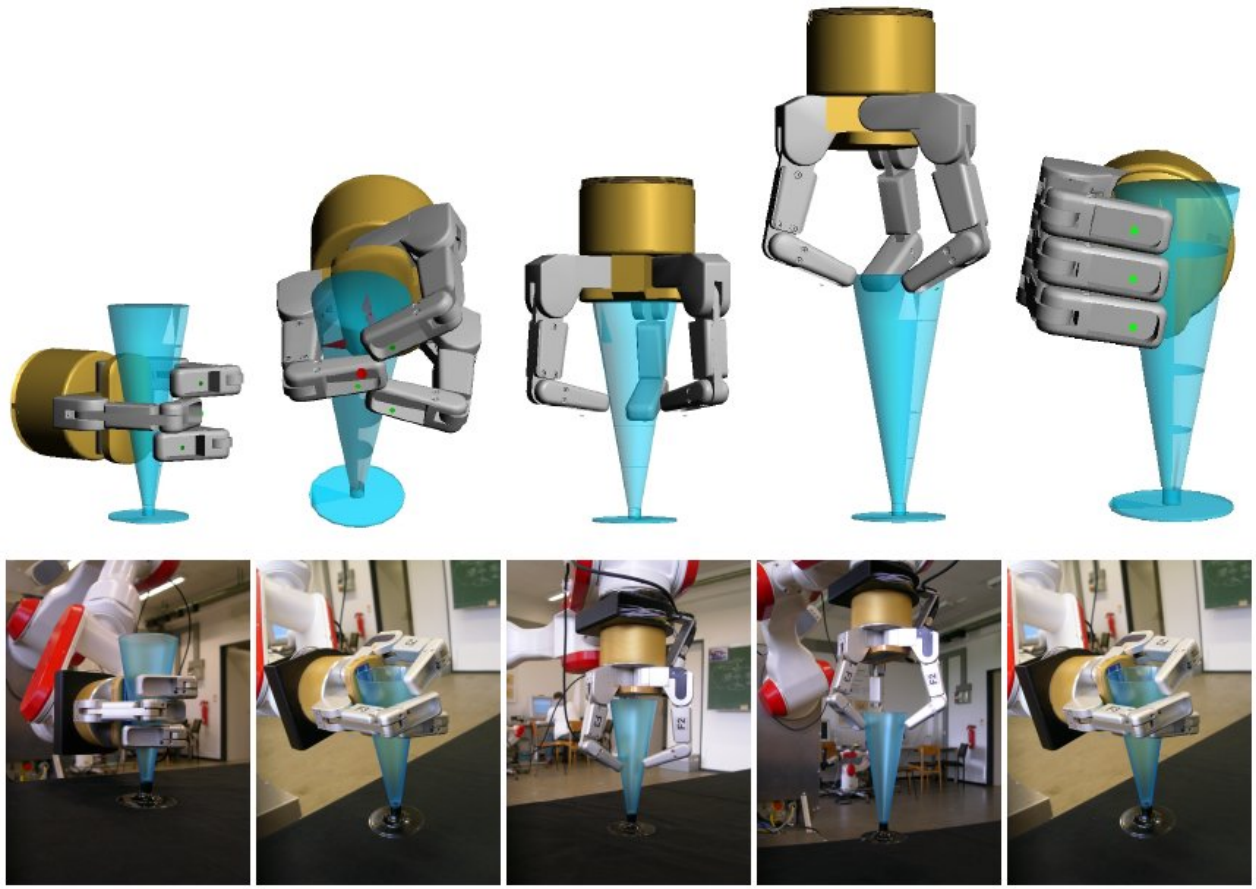


# Reinforcement Learning vs. Brute Force: Telephone



# Experimental Results

Testing some grasps with the service robot TASER:



# Experimental Results (cont.)

Testing some grasps with the service robot TASER: