# Reinforcement Learning (2)

## Algorithmic Learning 64-360, Part II

Norman Hendrich

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

26/06/2013

# Contents

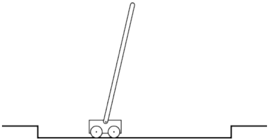
# Three classical RL examples: Matlab demos

- pole-balancing cart
- underpowered mountain-car
- robot inverse-kinematics

- those are all toy problems
  - small state-spaces
  - simplified environment models (e.g., no realistic physics)
  - but intuitive interpretation and visualization
- the learning-rules (Q, SARSA) will be explained later

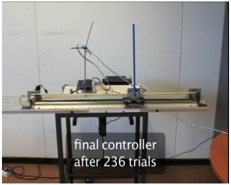
Demos from Jose Antonio Martin, Madrid: http://www.dacya.ucm.es/jam/download.htm

# Pole-balancing cart



- balance a pole against gravity
- simplest version is 1D: pole on cart on tracks
- 1D-balancing already requires 4 DOF:
  - $x$: position of the cart (should be near origin)
  - $\dot{x}$: velocity of the cart
  - $\theta$: angle of the pole (zero assumed to be upright)
  - $\dot{\theta}$: angular velocity of the pole

- simplified physics, discretization of state-space
- model as an episodic task
  - cart position $|x| > x_{max}$ terminates the task
  - pole falling down $|\theta| > \theta_{max}$ terminates the task



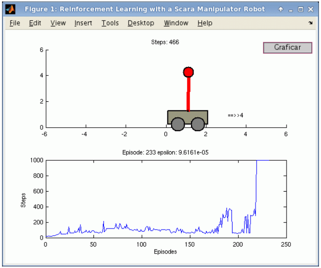
# Pole-balancing cart: reward function

if $|x| > 4$ or $|\theta| > 25^o$: (failure)

$$r = -10000 - 50 * |x| - 100 * |\theta|$$

else: $r = 10 - 10 * |10 * \theta|^2 - 5 * |x| - 10 * |\dot{\theta}|$

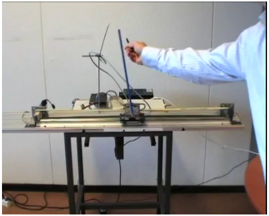
- punishing deviations from $x = 0$ and $\theta = 0$
- in other words, use prior knowledge to guide the learner



# Pole-balancing cart: videos

Many related problems:

- pole-balancing in 2D
  - decouple into two 4-DOF problems
  - don't try 8-DOF
- inverse pendulum
- multi-joint inverse-pendulum
- acrobot
- ...



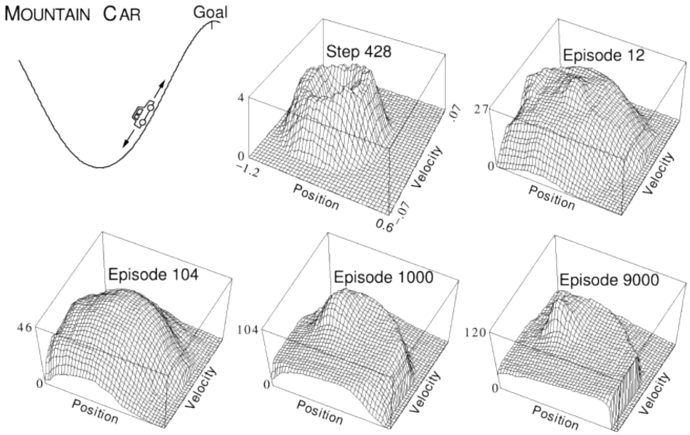
In theory, all those problems could be modelled analytically, using the corresponding differential-equations for the system. But in practice, this is impossible due to modelling errors, e.g. unknown mass and inertia of the moving parts.

# Mountain-car

- underpowered car should climb a mountain-slope
- simplified physics model
- actions are full-throttle $a \in \{-1, 0, +1\}$
- but constant $a = +1$ is not sufficient to reach the summit
- car must go backwards first a bit or even oscillate to build sufficient momentum to climb the mountain

- simple example of problems where the agent cannot reach the goal directly, but must explore intermediate solutions that seem counterintuitive
- typical example of *delayed reward*

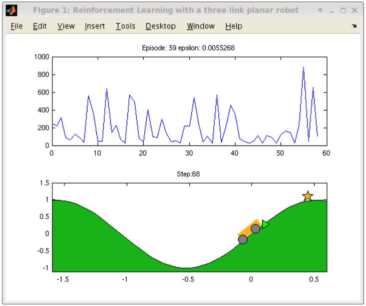
# Mountain-car



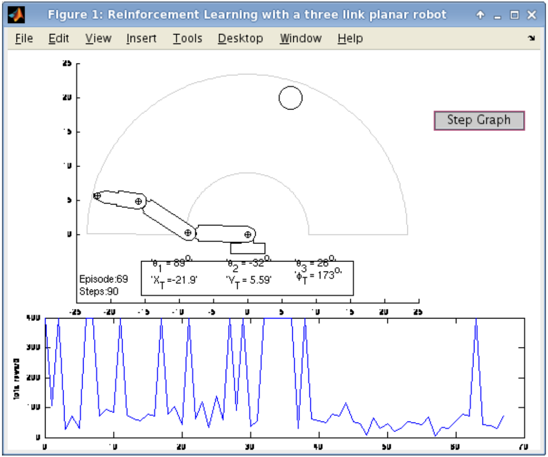

Details: Sutton and Barto, chapter 8.10

# Mountain-car: Matlab demo reward function

- $+100$ reward for reaching the mountain-summit
- $-1$ reward for every timestep without reaching the summit
- every episode is terminated after 1000 timesteps



# Inverse-Kinematics: planar robot



# Inverse-Kinematics: planar robot

- planar three-link robot (2D)
- try to reach given position $(x, y)$ position
- need to calculate joint-angles $\{\theta_1, \theta_2, \theta_3\}$
- actions are joint-movements $\Delta\theta_i \in \{-1, 0, +1\}$

Note:

- calculating $(x, y)$ for given $(\theta_1, \theta_2, \theta_3)$ is straightforward,
- but inverse-kinematics is much harder
- in general, no analytical solutions possible
- especially with high-DOF systems:
  humans, animals, humanoid robots: 70-DOF+

# Dynamic Programming

- overview of classical solution methods for MDPs, called *dynamic programming*
- based on the Bellman-equations
- also, the main theory behind RL

- demonstration of the application of DP — examples for calculating value functions and optimal *policies*
- discussion about the effectiveness and usefulness of DP

Details: Sutton and Barto, chapter 4

# Dynamic Programming for model-based learning

Dynamic Programming is a collection of approaches that can be used if a perfect model of the MDP's is available: We assume the Markov property, and $P^a_{ss'}$ and $R^a_{ss'}$ are known.
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* $\pi$.

*State-value function for policy* $\pi$:

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

*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 $V^\pi(s)$ 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 $s$, the $(k+1)$-th approximation $V_{k+1}$ is calculated iteratively. The old value of $s$ is replaced by an updated 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 $V \approx V^\pi$

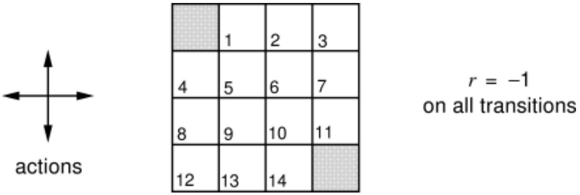
# Iterative Policy Evaluation

Note: this algorithm is conceptually very simple, but computationally very expensive. It requires a full sweep across all actions $a$ for all states $s$ in the state-space $S$:

- typically, exponential in the number of dimensions of $(s, a)$; all states and actions are tested many times
- requires full knowledge of the *dynamics* of the environment, $P^a_{ss'}$ and $R^a_{ss'}$ are required
- the algorithm only terminates if the change in $V(s)$ is smaller than $\Delta$, even if the policy $\pi$ is still suboptimal

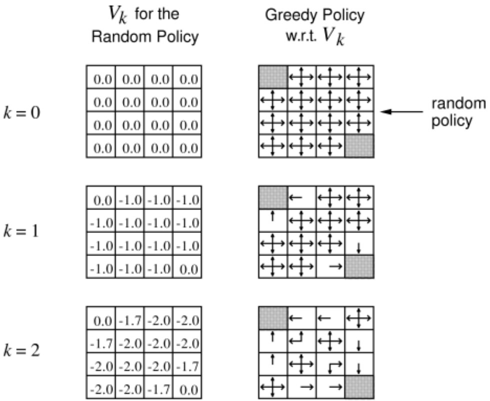
- better algorithms try to reduce the number of policy-evaluation steps, or try to bypass this step entirely

# Example: another gridworld




- agent moves across the grid: $a \in \{\text{up, down, left, right}\}$
- non-terminal states: $1, 2, \ldots, 14$
- one terminal state: $\{0, 15\}$ (shaded: one state, but two squares)
- therefore: an episodic task, undiscounted ($\gamma = 1$)
- 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)

$k = 3$

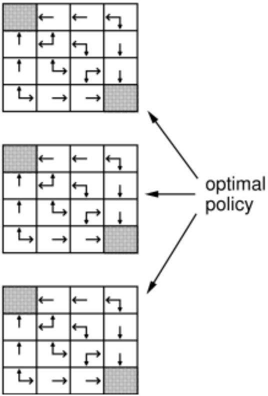
| | | | |
|---|---|---|---|
| 0.0 | -2.4 | -2.9 | -3.0 |
| -2.4 | -2.9 | -3.0 | -2.9 |
| -2.9 | -3.0 | -2.9 | -2.4 |
| -3.0 | -2.9 | -2.4 | 0.0 |

$k = 10$

| | | | |
|---|---|---|---|
| 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 |




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

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

$\Uparrow$ 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 in turn, 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 Iteration (3)

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

2. policy-evaluation
   Repeat
   $\quad \Delta \leftarrow 0$
   $\quad$ `for every` $s \in S$:
   $\quad\quad v \leftarrow V(s)$
   $\quad\quad V(s) \leftarrow \sum_a \pi(s,a) \sum_{s'} P_{ss'}^{\pi(s)} [R_{ss'}^{\pi(s)} + \gamma V(s')]$
   $\quad\quad \Delta \leftarrow \max(\Delta, |v - V(s)|)$
   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

Note: this algorithm converges to the optimal value-function $V^*(s)$ and the optimal policy $\pi^*$ (but may take a long time).

# Example: Jack's car rental

Jack manages two locations for a car rental company. Every day some number of customers arrive at each location to rent cars.

If Jack has a car available, he rents it out and is credited by $10 by the company. If he is out of cars at that location, then the business is lost. Cars become available for renting the day after they are returned. To help ensure that cars are available where they are needed, Jack can move them between the two locations overnight, at a cost of $2 per car moved.

# Jack's car rental (2)

We assume that the number of cars requested and returned at each location are Poisson random variables, meaning that the probability that the number is $n$ is $\frac{\lambda^n}{n!}e^{-\lambda}$, where $\lambda$ is the expected number.

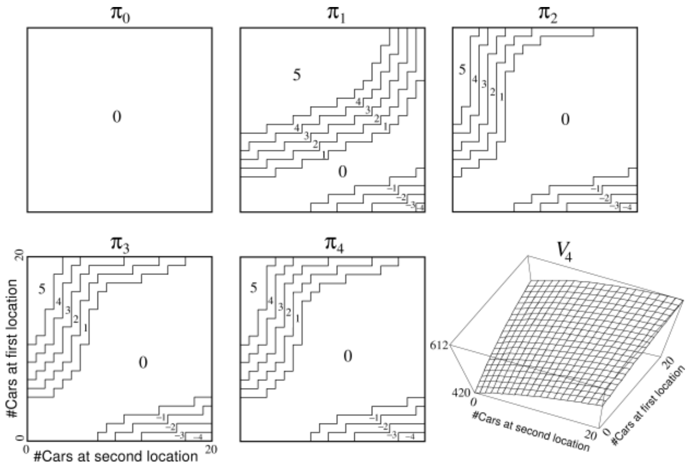
- note: Poisson-probabilities are hard to handle analytically
- assume $\lambda$ is 3 and 4 for car rental requests,

  and $\lambda$ is 3 and 2 for car returns

  at Jack's first and second locations
- also assume there are $n < 20$ cars
- a maximum of $m < 5$ cars can be moved between the two locations overnight
- discount rate s $\gamma = 0.9$, time-steps are days.

# Jack's car rental (3)

## cars moved from 1 to 2: Policies $\pi_0$ up to $\pi_4$ and value function $V_4(s)$



# Value Iteration

A method to improve convergence speed. In policy-evaluation, we use the *full policy-evaluation backup*:

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

Instead, the *full value-iteration 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

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

# Gambler's Problem: Example

A gambler has the opportunity to make bets on the outcomes of a sequence of coin flips. If the coin comes up heads, he wins as many dollars as he has staked on that flip; if it is tails, he loses his stake.

The game ends when the gambler wins by reaching his goal of $100, or loses by running out of money. On each flip, the gambler must decide what portion of his capital to stake, in integer number of dollars.

The problem can be formulated as an undiscounted episodic finite MDP. The state is the gambler's capital, $s \in \{1, 2, \dots, 99\}$ and the actions are stakes, $a \in \{1, 2, \dots, \min(s, 100 - s)\}$.

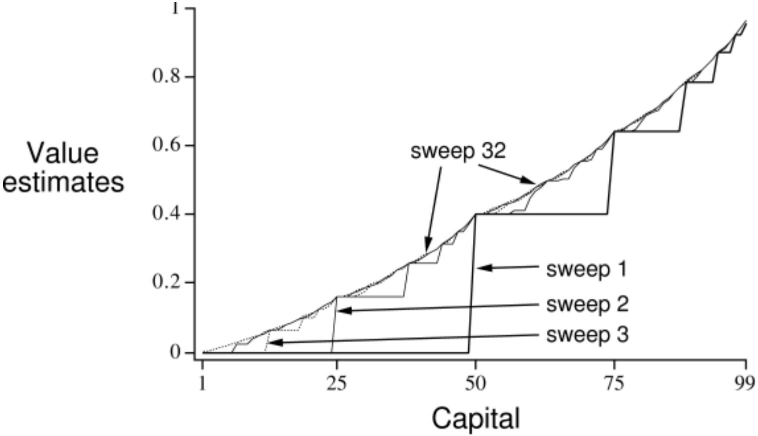
# Gambler's Problem (2)

The reward is zero on all transitions except those on which the gambler reaches his goal, when it is $+1$. The state-value function then gives the probability of winning from each state. A policy is a mapping from levels of capital to stakes. The optimal policy maximizes the probability of reaching the goal (winning $100).

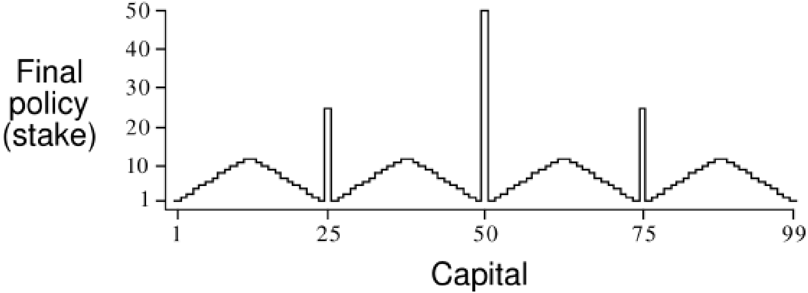
Let $p$ denote the probability of the coin coming up heads. If $p$ is known, then the entire problem is known and it can be solved, for instance by value-iteration. The next figure shows the change in the value function $V_{p=0.4}(s)$ over successive sweeps of value iteration, and the final policy found, for the case of $p = 0.4$.

# Gambler's Problem (3)

## Estimates of the value function $V(s)$ for $p = 0.4$



# Optimal policy for $p = 0.4$




- Why would this be a good policy?
- e.g, for $s = 50$ betting all on one flip, but nor for $s = 51$

# 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:

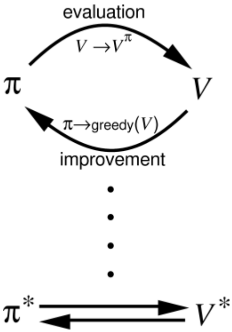
Repeat until the convergence criterion is met:

- pick a random state and apply the appropriate *backup*.

- this still requires a lot of computation, but allowing a tradeoff between PE and PI
- the states for the application of the backup can be selected, e.g. the experience of an agent can serve as a guide.

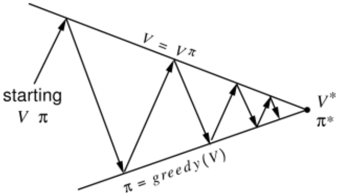
# Generalized Policy Iteration (GPI)

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




geometric metaphor

for the convergence of GPI:



# Efficiency of DP

- finding an optimal *policy* is polynomial in the number of states $s \in S$
- unfortunately, the number of states is often extremely high; typically grows exponentially with the number of state-variables: Bellman's *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 also suitable for parallel computation
- it is surprisingly easy to find MDPs, where DP methods can not be applied

# Summary: Dynamic Programming

- Policy Evaluation: complete *backups* without maximum
- Policy Improvement: form a *greedy policy*, even if only locally
- Policy Iteration: alternate the above two processes
- Value Iteration: use backups with maximum

- Complete *Backups* (over the full state-space $S$)
- Generalized Policy Iteration (GPI)
- Asynchronous DP: a method to avoid complete backups

# Monte Carlo methods

- catchy name for many algorithms based on random numbers
- in the context of reinforcement learning: refers to algorithms that learn directly from the agents' experience
  - use the collected return to estimate $V(s)$
  - then derive good policy $\pi$
  - only defined for episodic tasks
- *online* learning: no model of the environment required, but still converges to the optimal policy $\pi^*$
- *simulation*: learn a policy for the real environment on a simulated (partial) model of the environment

# Monte Carlo methods: basic idea

- given: MDP, initial policy $\pi$
- goal: learn $V^{\pi}(s)$

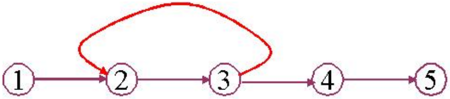
- run a (large) number of episodes,
- record all states $s_i$ visited by the agent,
- record the final return $R_t$ collected at the end of each episode

- for all states $s_i$ visited during one episode, update $V(s_i)$ based on the return collected in that episode
- this converges for all states that are visited "often"

# First-visit and Every-visit Monte Carlo

Two ways to update $V^\pi(s)$ from an episode trace:

- *first-visit MC*: update $V^\pi(s_i)$ only for the first time that state $s_i$ is visited by the learner
- *every-visit MC*: update $V^\pi(s_i)$ every time that state $s_i$ is visited.




- first-visit MC convergence: each return is an independent, identically distributed estimate of $V^\pi(s)$. By the law of large numbers the sequence of averages converges to the expected value. The standard deviation of the error falls as $1/\sqrt{n}$, where $n$ is the number of returns averaged.
- both algorithms converge asymptotically

# Monte Carlo first-visit policy evaluation

Initialize:

$\pi$: the policy to evaluate
$V$: initial estimate of the value function
$Returns(s)$: an empty list of returns, for all $s \in S$

Repeat:

(1) generate an episode using policy $\pi$
(2) for every state $s$ visited in the episode:
$R \leftarrow Return$ for the first visit of $s$
append $R$ to $Returns(s)$
$V(s) \leftarrow \text{average}(Returns(s))$

# Example: playing Blackjack

Goal: obtain cards whose sum is as large as possible, but not exceeding 21. Face cards count as 10, an ace can count either as 1 or 11. Here, considering a single player vs. the dealer.

Game begins with two cards dealt to both dealer and player. One of the dealer's cards is faceup, the other is facedown. Player can ask for additional cards (*hit*) or stop (*stick*), and loses (*goes bust*) when the sum is over 21. Then it is the dealer's turn, who has a fixed strategy: dealer sticks on any sum of 17 or greater, and hits otherwise. If the dealer goes bust, then the player wins; otherwise the outcome is determined by whose final sum is closer to 21.

# Modeling Blackjack

Model as an episodic finite MDP; each game is one episode. All rewards within a game are zero, no discount ($\gamma = 1$). Assume that cards are dealt from an infinite deck, so no need to keep track of cards already dealt.

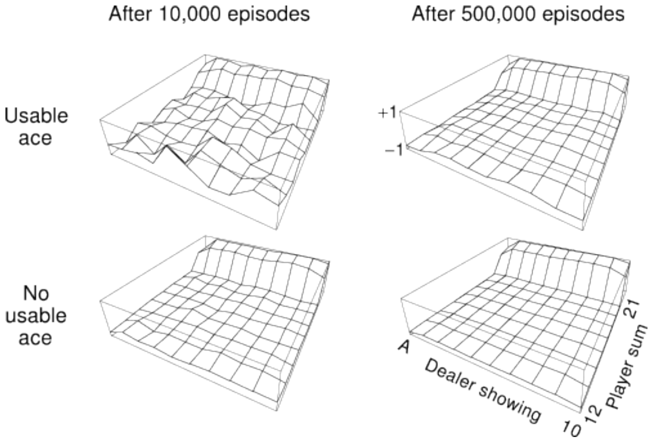
- *states*:
  - sum of current cards (12 .. 21)
  - visible cards of the dealer (ace .. 10)
  - player has useable ace (1 or 11)
  - total number of states is 200.
- *reward*: $+1$ for winning, 0 for draw, -1 for losing
- *return*: same as reward
- *actions*: hit or stick

# Solving Blackjack

- example policy $\pi$ is: stick if sum is 20 or 21, otherwise hit (i.e., ask for another card)
- simulate many blackjack games using the policy, and average the returns following each state
- this uses random numbers for the card generation
- states never recur in the task, so no difference between first-visit and every-visit MC

- next slide shows the estimated value-function $V(s)$ after 10000 and 500000 episodes. Separate functions are shown for whether the player holds an useable ace or not.

# Solving Blackjack: $V^{\pi}(s)$




Approximated state-value functions $V(s)$ for the policy that sticks only on 20 or 21. Not the best policy...

# Solving Blackjack: $V^{\pi}(s)$

Think about the estimated function:

- why is the "useable-ace" function more noisy?
- why are the values higher for the useable-ace case?
- how to explain the basic shape of the value function?
- why does the function drop-off on the left?
- ...

# Solving Blackjack: MC vs. DP approaches

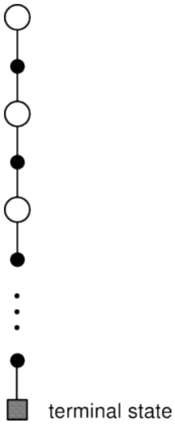
Note: we have complete knowledge of the environment here, but it would still be very difficult to apply Dynamic Programming for solving the game:

- DP requires the distribution of next events and rewards, namely $P^a_{ss'}$ and $R^a_{ss'}$
- those are not easy to compute

- on the other hand, generating the sample games required for MC is straightforward
- convergence may require many episodes

# Monte Carlo backup diagram

- consists of the whole episode from the start-state to the goal-state
- exactly one action considered in each state, namely, the action actually taken
- no bootstrapping: estimates for one state do not depend on other estimates
- time required to estimate the value of $V(s)$ for a given state $s$ does not depend on the number of states
- option to learn $V(s)$ only for those states that seem interesting



# Monte Carlo estimation of action values

When a model of the MDP is available:

- estimate $V^{\pi}(s)$ for initial policy $\pi$
- use greedification to find better policy
- but: this requires a look-ahead one step to find the action that leads to the best combination of reward and next state

Without a model:

- state values $V(s)$ are not enough: we don't know $P^a_{ss'}$ or $R^a_{ss'}$
- instead, consider $Q(s,a)$ for all actions to find a better policy
- need MC method to estimate $Q^{\pi}(s,a)$

# Monte Carlo estimation of $Q^\pi(s, a)$

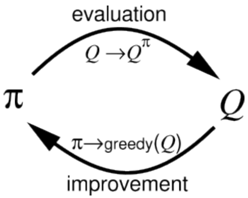
- $Q^\pi(s, a)$: the expected return when starting in state $s$, taking action $a$, and thereafter following policy $\pi$
- the Monte Carlo method is essentially unchanged:
  - create and update data-structure for $Q(s, a)$
  - first-visit: average the returns following the first time in each episode that $s$ was visited and $a$ was selected
  - every-fisit: average the returns following every visit to a state $s$ and taking action $a$
  - again, convergence can be shown if every state-action pair is sampled "often"
- but one major complication: for a deterministic policy $\pi$, many actions $a$ may not be taken at all, and the MC estimates for those $(s, a)$ pairs will not improve
- need to *maintain exploration*

# Monte Carlo exploring starts

- start each episode using a given $(s, a)$ pair
  — instead of starting in state $s$ and taking $a$ according to the current policy
- ensure that all possible $(s, a)$ pairs are used as the starting state with a probability $p(s, a) > 0$
- the $Q^\pi(s, a)$ estimate will then converge

- using non-deterministic policies may be easier and quicker
- e.g., using a $\epsilon$-greedy policy will also guarantee that all $(s, a)$ pairs are visited often

# Monte Carlo control: improving the policy




- estimate $Q^{\pi}(s, a)$ using (one or many) MC episodes
- from time to time, update policy $\pi$ using greedification

$$\pi_0 \xrightarrow{E} Q_0^{\pi} \xrightarrow{I} \pi_1 \xrightarrow{E} Q_1^{\pi} \xrightarrow{I} \pi_2 \ldots \xrightarrow{I} \pi^* \xrightarrow{E} Q^*$$

# Monte Carlo first-visit with exploring starts

Initialize for all $s \in S, a \in A(s)$:

$\pi$: the policy to evaluate

$Q(s, a)$: initial estimate of the value function

$Returns(s, a)$: an empty list of returns, for all pairs $(s, a)$

Repeat:

(1) generate an episode using exploring starts and policy $\pi$

(2) for every pair $(s, a)$ visited in the episode:

$R \leftarrow Return$ for the first visit of $(s, a)$

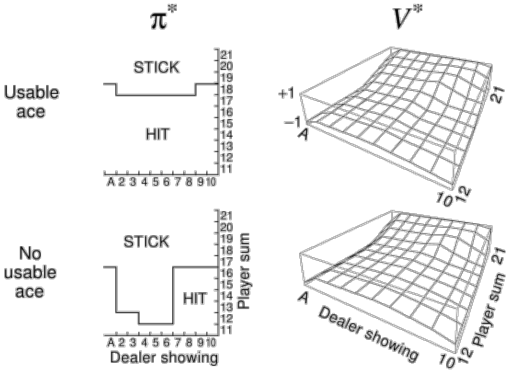
append $R$ to $Returns(s, a)$

$Q(s, a) \leftarrow \text{average}(Returns(s, a))$

(3) for every $s$ visited in the episode:

$\pi(s) \leftarrow \arg\max_a Q(s, a)$

# Solving Blackjack: Optimal policy



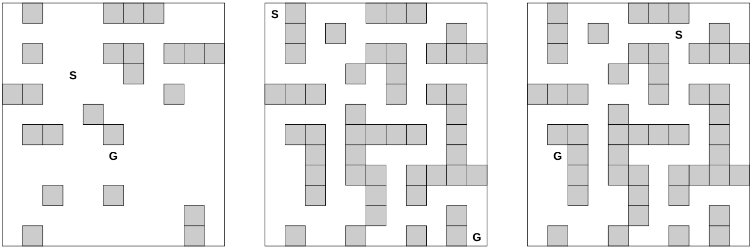

- remember: this policy is for fresh-cards in every game
- state-space is much more complex without card replacement

# Monte Carlo: summary

- our third approach to estimate $V(s)$ or $Q(s, a)$
- several advantages over DP methods:
  - no model of the environment required
  - learning directly from interaction with the environment
  - option to learn only parts of the state-space
  - less sensitive should the Markov property be violated

- using a deterministic policy may fail to learn
- need to maintain exploration
  - exploring starts
  - non-deterministic policies, e.g. $\epsilon$-greedy
- no bootstrapping

# Intermission: labyrinth task




- learner should find way from start-state $S$ to goal-state $G$
- actions are $\{up, down, left, right\}$
- agent cannot move into walls (or outside world)
- first challenge is to design an appropriate reward-function

# Intermission: labyrinth task

- another gridworld-style example
- represent $V(s)$ as 1D- or 2D-array
- actions $\{up, down, left, right\}$ implemented as
  $\Delta x = \pm 1, \Delta y = \pm 1$ (2D)
  $\Delta x = \pm 1, \Delta y = \pm n$ (1D)
  optionally, add extra grid-cells for the outer walls
  explicit representation of actions using `switch` statement

- problem can be either easy or very (exponentially) hard depending on the number of obstacles
- compare RL with planning algorithms (e.g. $A^*$)
- can random action-selection work on difficult labyrinths?

# Intermission: gridworld example code

Example C-code for estimation of $V(s)$ for a gridworld:

- $V(s)$ implemented as 2D-array `W_matrix`
- code keeps separate array $V'(s)$ for updated values
- $V(s) \leftarrow V'(s)$ after each sweep through all states
- action-selection and reward calculation coded explicitly using a `switch`-statement

- similar when using $Q(s, a)$ representation

tams.informatik.uni-hamburg.de/lectures/ss2013/AL/uebungen/gridworld2.c

# Temporal-Difference Learning

- TD-Learning
- Q-Function
- Q-Learning algorithm
- Convergence
- Example: GridWorld
- SARSA

# Temporal-Difference Learning

One of the *key ideas* for solving MDPs:

- learn from raw experience without a model of the environment
- update estimates for $V(s)$ or $Q(s, a)$ based on other learned estimates
- i.e., *bootstrapping* estimates

- combination of Monte Carlo and DP algorithms
- $TD(\lambda)$ *interpolates* between MC and DP

- algorithms can be extended to non-discrete state-spaces

# Temporal-Difference Learning: TD Prediction

- MC methods wait until episode ends before updating $V$
- the MC update uses the actual return $R_t$ received by the agent:

$$V(s_t) \leftarrow V(s_t) + \alpha \Big[ R_t - V(s_t) \Big]$$

- TD methods only wait one time-step, updating using the observed immediate reward $r_{t+1}$ and the estimate $V(s_{t+1})$.
- the simplest method, $TD(0)$ uses:

$$V(s_t) \leftarrow V(s_t) + \alpha \Big[ r_{t+1} + \gamma V(s_{t+1}) - V(s_t)) \Big]$$

- basically, MC udpate is $R_t$ while TD update is $r_{t+1} - \gamma V_t(s_{t+1})$

# TD Learning: TD Prediction

Reminder (definition of Return, and Bellman-Equation):

$$
\begin{aligned}
V^{\pi}(s) &= E_{\pi}\{R_t | s_t = s\} && (1)\\
&= E_{\pi}\left\{\sum_{k=0}^{\infty} \gamma r_{t+k+1} \;\middle|\; s_t = s\right\} && (2)\\
&= E_{\pi}\left\{r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma r_{t+k+2} \;\middle|\; s_t = s\right\} && (3)\\
&= E_{\pi}\left\{r_{t+1} + \gamma V^{\pi}(s_{t+1}) \;\middle|\; s_t = s\right\} && (4)
\end{aligned}
$$

- Monte Carlo algorithms use (1)
- Dynamic Programming uses (4)

# The Q-Function

We define the Q-function as follows:

$$Q^{\pi}(s,a) \equiv r(s,a) + \gamma V^{\pi}(\delta(s,a))$$

$\pi^*$ 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 reward distribution $r$ and dynamics $\delta$ 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 optimal $Q^*$-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!

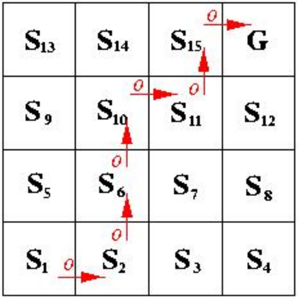
# Example: GridWorld (1)

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)

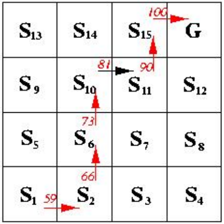
Example of a path through a state space:




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

# Example: GridWorld (3)

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




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

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

...

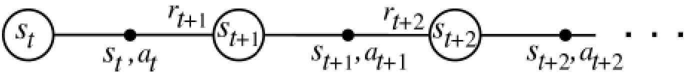
# SARSA: On-policy learning

iterative estimation of the $Q(s, a)$ function:

- state $s$, chose action $a$ according to current policy
- check reward $r$, enter state $s'$
- select next action $a'$ also according to current policy
- update

$$Q_{t+1}(s, a) \leftarrow Q_t(s, a) + \alpha \left[ r_{t+1} + \gamma Q(s', a') - Q(s, a) \right]$$

- quintuple $(s, a, r, s', a')$: SARSA



# SARSA: the algorithm

Initialize $Q(s, a)$ arbitrarily (e.g. zeros)
Repeat (for each episode):
  Initialize $s$
  Choose $a$ from $s$ using policy derived from $Q$ (e.g. $\epsilon$-greedy)
  Repeat (for each step of episode):
    Take action $a$, observe $r$, $s'$
    Choose $a'$ from $s'$ using policy derived from $Q$ (e.g. $\epsilon$-greedy)
    $Q_{t+1}(s, a) \leftarrow Q_t(s, a) + \alpha\left[r_{t+1} + \gamma Q(s', a') - Q(s, a)\right]$
    $s \leftarrow s'; a \leftarrow a';$
  until $s$ is terminal

# SARSA vs. Q-learning

- SARSA: select $a'$ according to current policy

$$Q_{t+1}(s,a) = Q_t(s,a) + \alpha \left[ r_{t+1} + \gamma Q_t(s',a') - Q_t(s,a) \right]$$

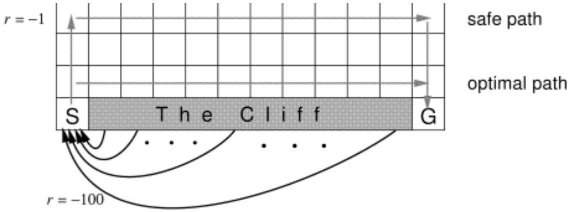
- Q-learning:

$$Q_{t+1}(s,a) = Q_t(s,a) + \alpha \left[ r_{t+1} + \gamma \max_{a'} Q_t(s',a') - Q_t(s,a) \right]$$

learning rate $\alpha$, $0 < \alpha < 1$

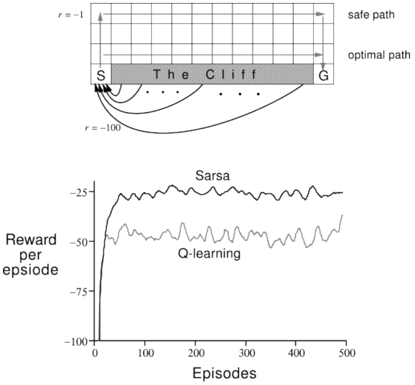
discount factor $\gamma$, $0 \leq \gamma < 1$

# The cliff: SARSA vs. Q-learning




- gridworld, start state S, goal state G
- reward -1 for every step, falling from the cliff -100
- $\epsilon$-greedy action selection, $\epsilon = 0.1$
- Q-learning learns the optimal policy: shortest path
- SARSA learns a longer but safer path, because it takes the current policy into account („on policy learning“)

# The cliff: SARSA vs. Q-learning

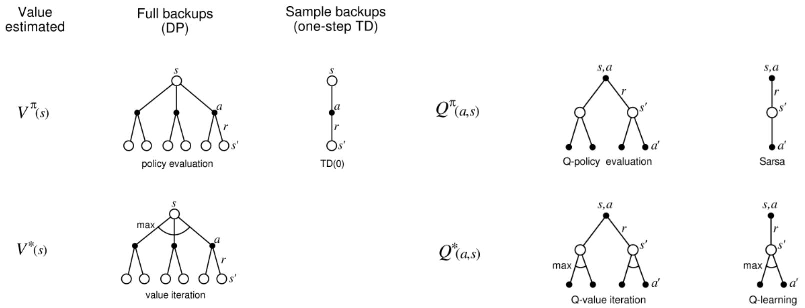



- $\epsilon$-greedy action selection, $\epsilon = 0.1$
- therefore, risk of taking an explorative step off the cliff

# 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

# Backup diagrams: summary




- policy-evaluation, $TD(0)$, SARSA
- value-iteration, Q-learning

# 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)

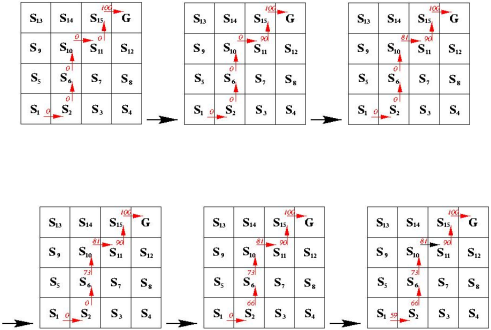
Advantages:

- 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:

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

# 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.

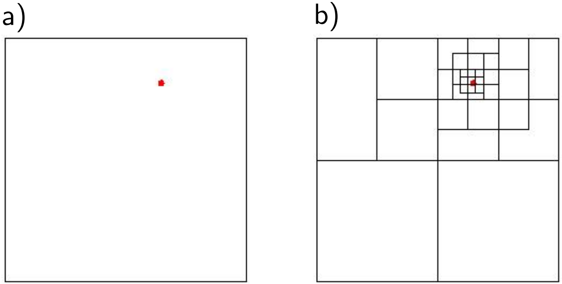
# Learner-Cascade

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

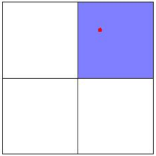
An example state space without discretization (a) with variable discretization (b)

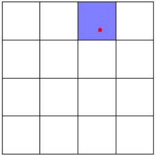


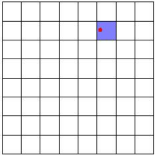

This requires knowledge of the structure of the state space.

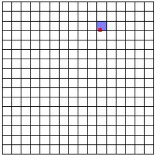
# Learner-Cascade - *n*-Stage Learner-Cascade

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.

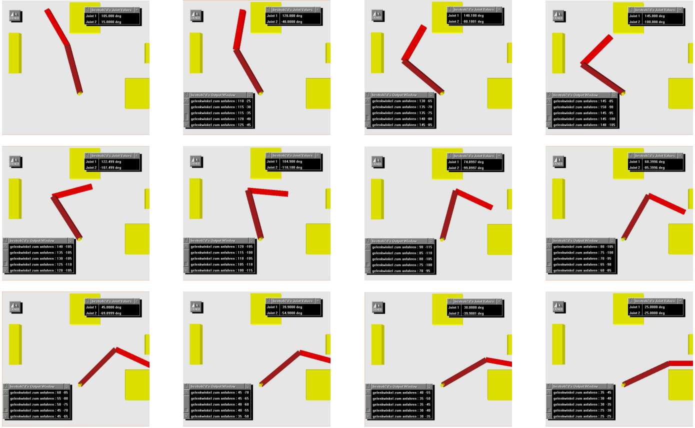
- The state space $S$ is the discrete configuration space, i.e. every combination of joint angles $(\theta_1, \theta_2, \ldots, \theta_{Dim})$ is exactly one state of the set $S$ except the target configuration
- The set $A(s)$ of all possible actions for a state $s$ includes the transitions to neighbor-configurations 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:
  $R^a_{ss'} = \textbf{Reward\_not\_in\_goal} \quad \forall s \in S, a \in A(s), s' \in S$ and
  $R^a_{ss'} = \textbf{Reward\_in\_goal} \quad \forall s \in S, a \in A(s), s' = s_t$.
  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):



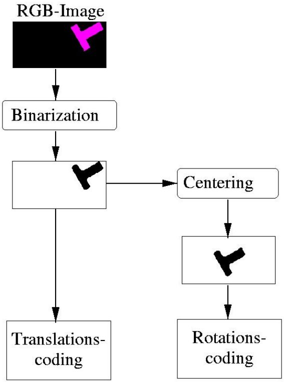
# Grasping with RL: Partition of DOFs

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:

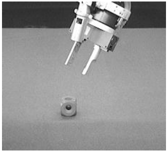
- The state space is much smaller.
- The state-encoding is designed the way that the state-vectors contain only the relevant information for the corresponding learner.

# Grasping with RL: Partition of DOFs (4)

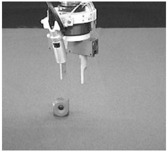
In practice, the two learners will be used alternatingly. Firstly the *translation-learner* will be run in long learning-steps, until it has reached the goal defined in its state-encoding.
Then the *translation-learner* is replaced by the *rotation-learner*, which is also used in long learning-steps 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.

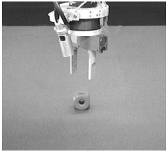
# Grasping with RL: Partition of DOFs (5)



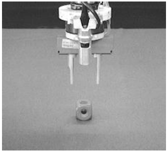
(a)



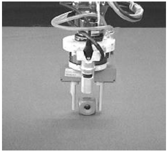
(b)



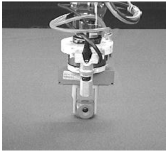
(c)



(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

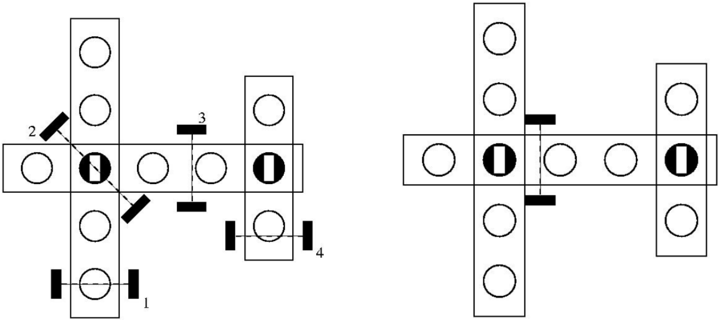
**Grip is optimal with respect to local criteria:**

- 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




(a) (b)

# Two approaches

**One learner:**

- The states consist of a set of $m+n$ local and global properties: $s = (f_{l_1}, \dots, f_{l_m}, f_{g_1}, \dots, f_{g_n})$.
- 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 $s = (f_{l_1}, \dots, f_{l_m})$.
- 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 $s = (f_{g_1}, \dots, f_{g_n})$ to actions concerning the translational component $a = (x, y)$.

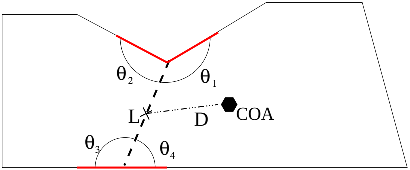
# Setup

- 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.

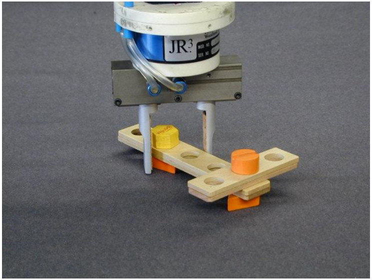
# State Encoding

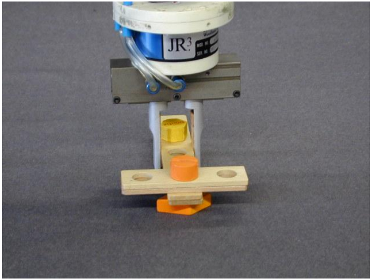



The orientation Lerner uses length $L$ as well as the angles $\Theta_1, \ldots, \Theta_4$, while the position learner uses the distance $D$ between the center of the gripper-line and the optical center of gravity of the object.

# Measures for Self-Evaluation in the Orientation Learner

**Visual feedback of the grasp success:**
Friction results in rotation or misalignment of the object.

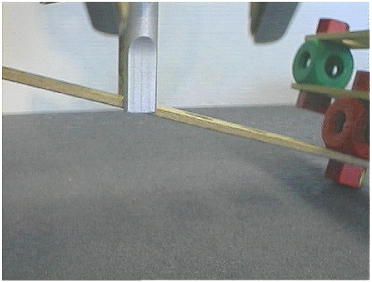


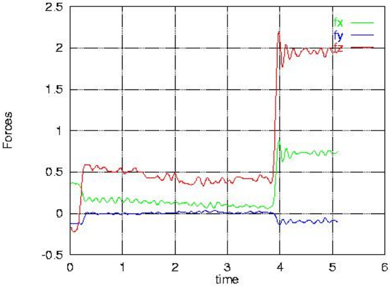



# Measures for Self-Evaluation in the Position Learner (1)

**Feedback using force torque sensor:**

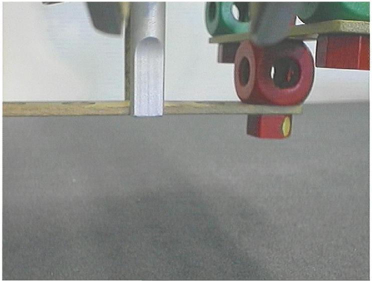
Unstable grasp – analyzed by force measurement

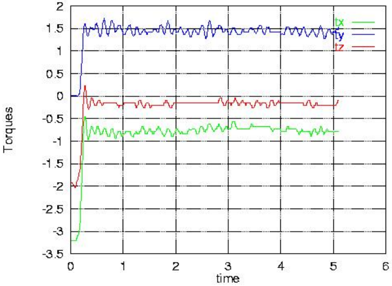




# Measures for Self-Evaluation in the Position Learner (2)

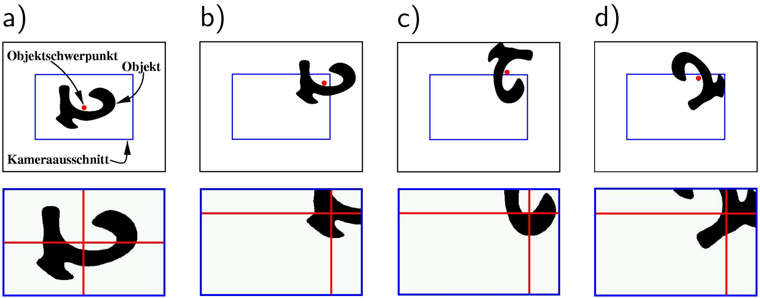
Suboptimal grasp – analyzed by torques





# The Problem of Hidden States

Examples for incomplete state information:

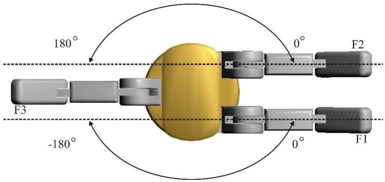


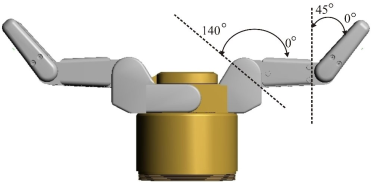
# Grasping mit RL: Barrett-Hand

- 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

# BarrettHand 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.)

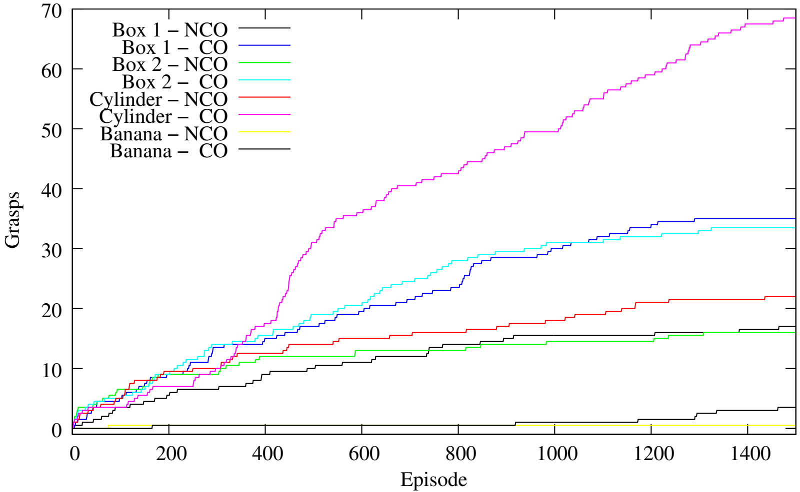
Automatic Value Cutoff: remove actions

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

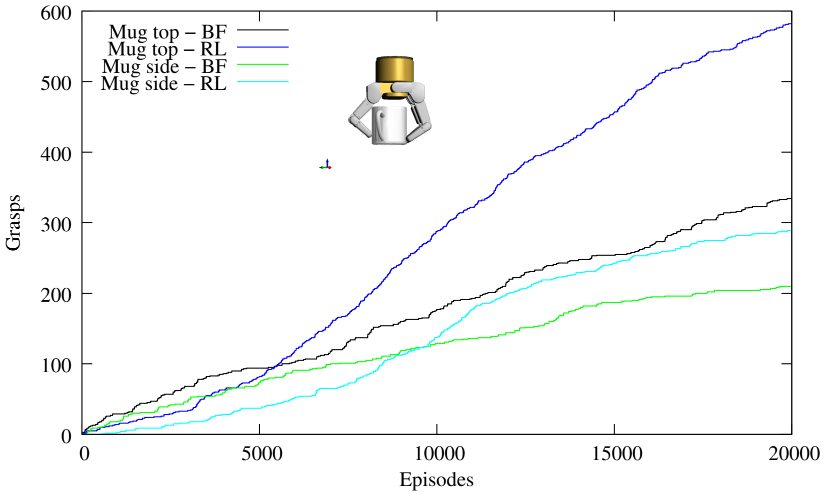
$$Q(s,a) \leftarrow \begin{cases} Q(s,a) + \alpha \left[ r + \gamma Q(s',a') - Q(s,a) \right] \\ \qquad \text{if } 0 \leq Q(s,a) < r * \beta \\ \\ \text{remove } Q(s,a) \text{ from } Q \text{ otherwise} \end{cases}$$

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

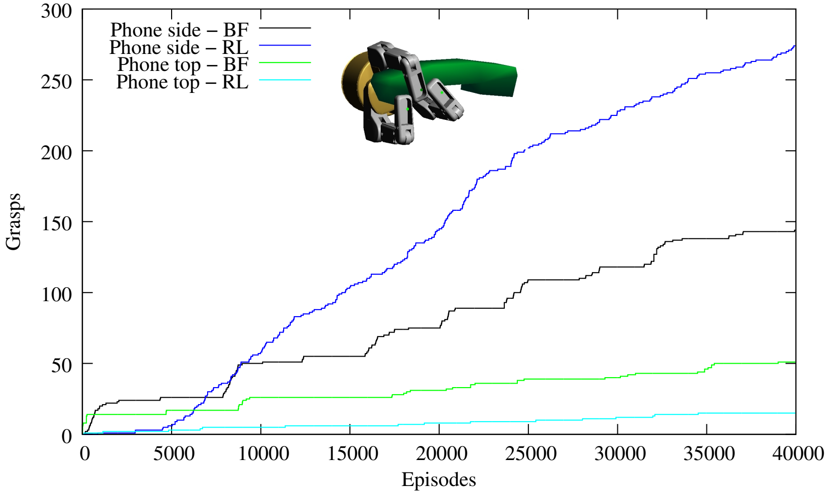
# Automatic Value Cutoff vs. no Cutoff



# Reinforcement Learning vs. Brute Force: Mug

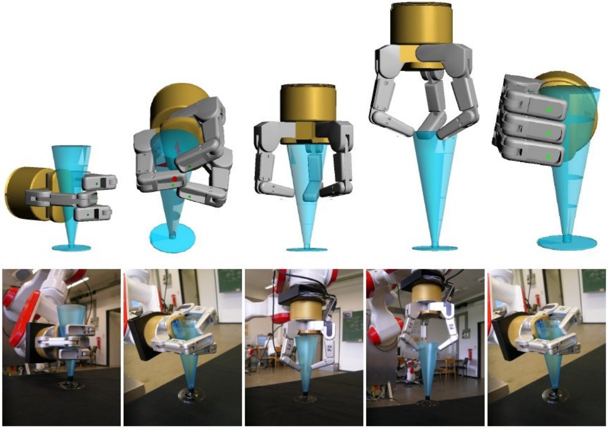


# Reinforcement Learning vs. Brute Force: Telephone



# Experimental Results

Testing some grasps with the service robot TASER:



# Experimental Results (cont.)

Testing some grasps with the service robot TASER: