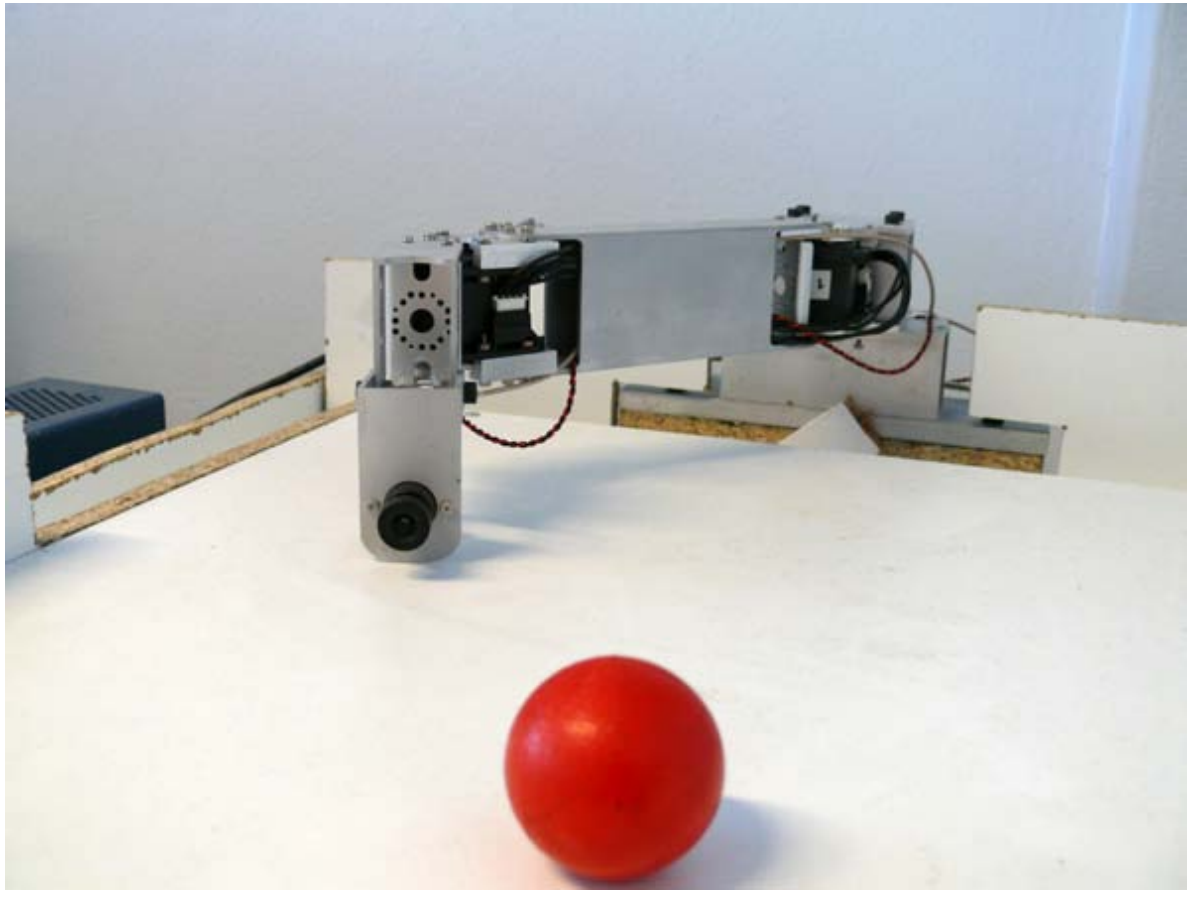
Reinforcement Learning for Robot Control

Lucian Buşoniu
DCSC, TUDelft

SC4240TU Lecture 7
1 March 2011

RL for a robot goalkeeper

Learn how to catch ball, using video camera image



RL for helicopter control (Stanford)

Learn to perform aerobatics from expert demonstrations

Outline

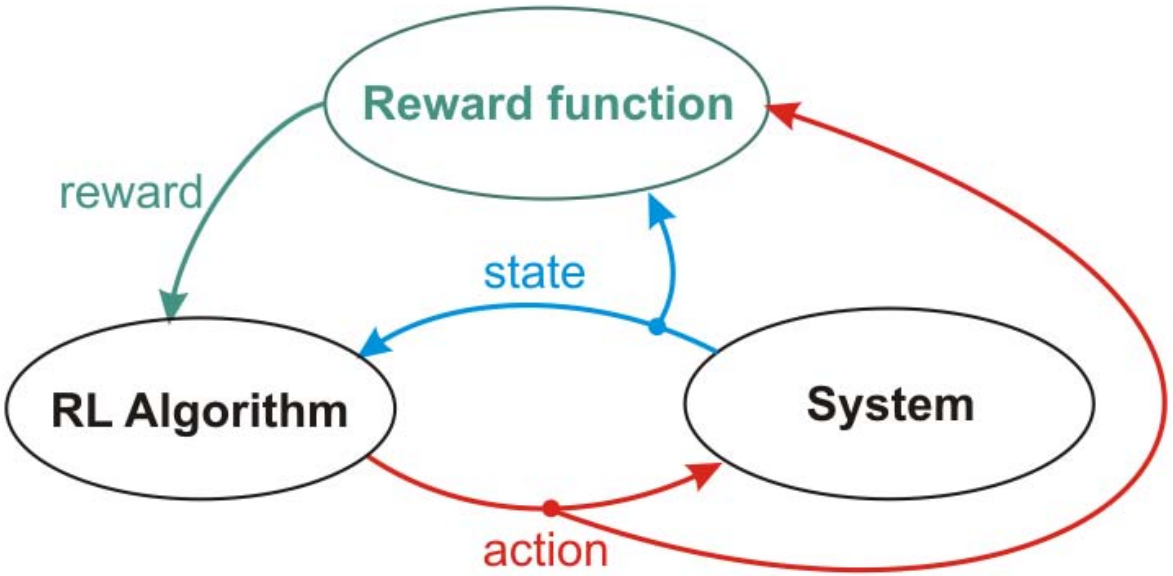
Why learning?

Learning can find solutions that:

1. cannot be found in advance
   - environment or robot too complex
   - problem not fully known beforehand
2. steadily improve
3. adapt to time-varying environments

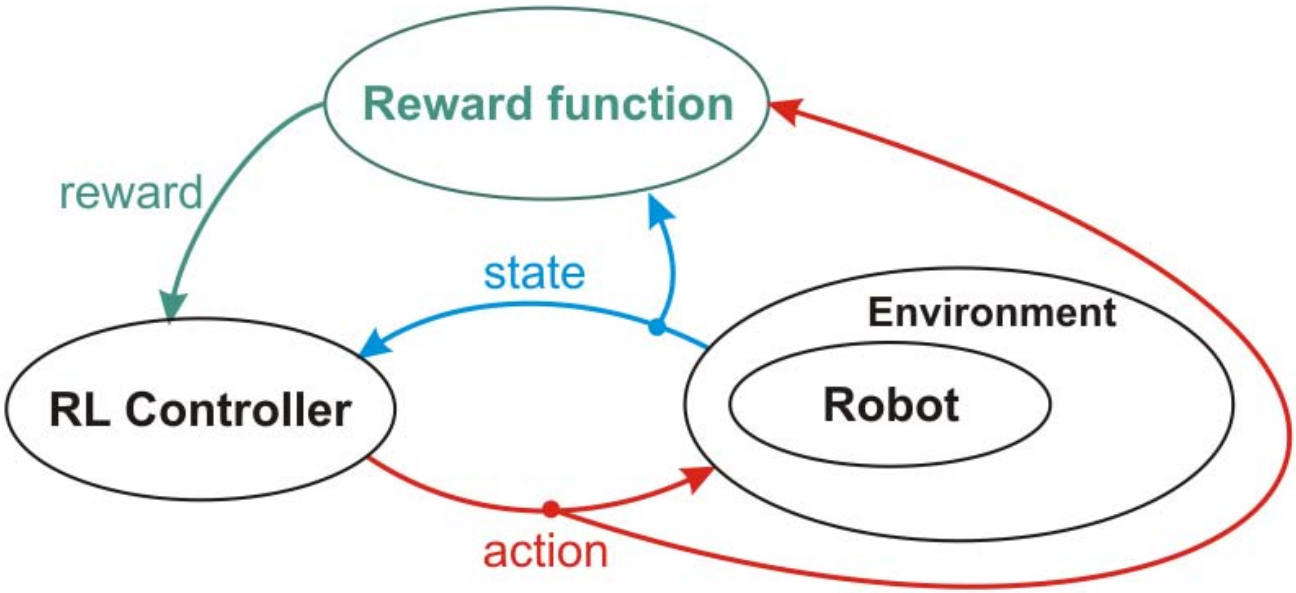
Essential for any **intelligent** robot

Principle of RL



- Interact with a system through states and actions
- Receive rewards as performance feedback
- Inspired by human and animal learning

RL for robot control



- Algorithm = controller
- System = robot + environment
- Adaptive, model-free, optimal control

Part I: Classical RL

Discrete states and actions

A simple cleaning robot example

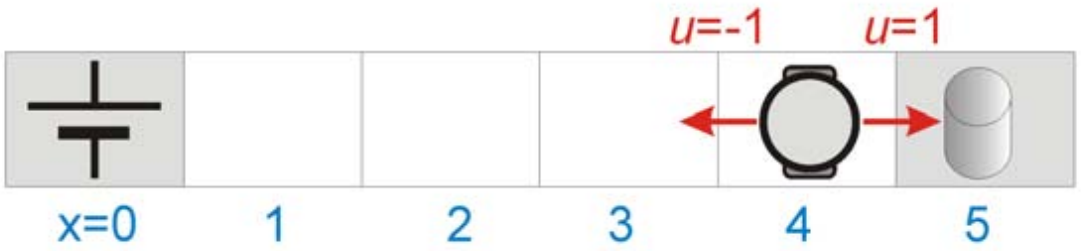
- Cleaning robot in a 1-D world
- Either pick up trash (reward $+5$) or power pack (reward $+1$)
- After picking up item, **trial** terminates

Cleaning robot: State, action, transition, & reward

- Robot in given state x (cell)
- and takes action u (e.g., move right)

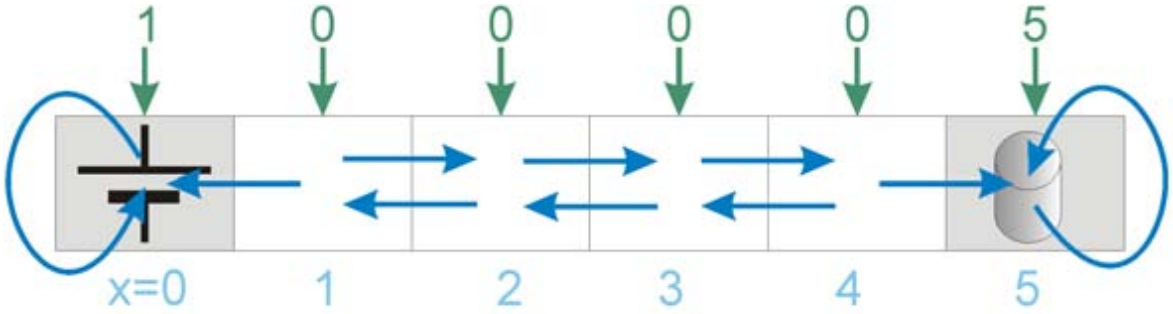
- Robot reaches next state x'
- and receives reward r = quality of transition (here, $+5$ for collecting trash)

Cleaning robot: State & action space



- State space $X = \{0, 1, 2, 3, 4, 5\}$
- Action space $U = \{-1, 1\} = \{\text{left, right}\}$

Cleaning robot: Transition & reward functions



- Transition function (process behavior):

$$x' = f(x,u) = \begin{cases} x & \text{if } x \text{ is terminal (0 or 5)} \\ x+u & \text{otherwise} \end{cases}$$

- Reward function (immediate performance):

$$r = \rho(x,u) = \begin{cases} 1 & \text{if } x = 1 \text{ and } u = -1 \text{ (powerpack)} \\ 5 & \text{if } x = 4 \text{ and } u = 1 \text{ (trash)} \\ 0 & \text{otherwise} \end{cases}$$

- Note: **terminal** states cannot be left
  & do not accumulate rewards!

Markov decision process

1. State space X
2. Action space U
3. Transition function $x' = f(x, u)$
4. Reward function $r = \rho(x, u)$

... form a **Markov decision process**

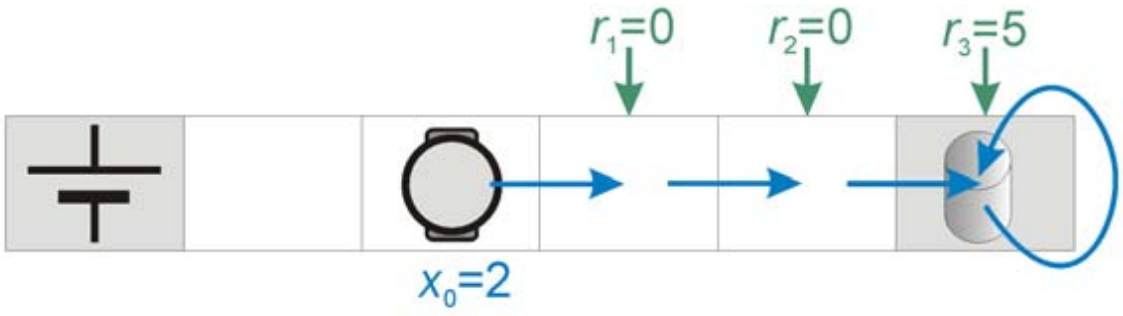
Note: stochastic formulation possible

Policy

- Policy h: mapping from x to u (state feedback)
- Determines controller behavior

Example: $h(0) = *$ (terminal state, action is irrelevant), $h(1) = -1$, $h(2) = 1$, $h(3) = 1$, $h(4) = 1$, $h(5) = *$

Cleaning robot: Return



Assume h always goes right

$$R^h(2) = \gamma^0 r_1 + \gamma^1 r_2 + \gamma^2 r_3 + \gamma^3 0 + \gamma^4 0 + \dots$$
$$= \gamma^2 \cdot 5$$

Because x_3 is terminal, all remaining rewards are 0

Learning goal

Find h that maximizes discounted return:

$$R^h(x_0) = \sum_{k=0}^{\infty} \gamma^k r_{k+1} = \sum_{k=0}^{\infty} \gamma^k \rho(x_k, h(x_k))$$

from any x_0

Discount factor $\gamma \in [0, 1)$:

- induces a "pseudo-horizon" for optimization
- bounds infinite sum
- encodes increasing uncertainty about the future

Q-function

- **Q-function** of policy h:

$$Q^h(x_0, u_0) = \rho(x_0, u_0) + \gamma R^h(x_1)$$

(return after taking u_0 in x_0 and then following h)

- Why Q-function? Useful to choose actions

Optimal solution

- **Optimal Q-function**:

$$Q^* = \max_h Q^h$$

$\Rightarrow$ Greedy policy in Q^*:

$$h^*(x) = \arg\max_u Q^*(x, u)$$

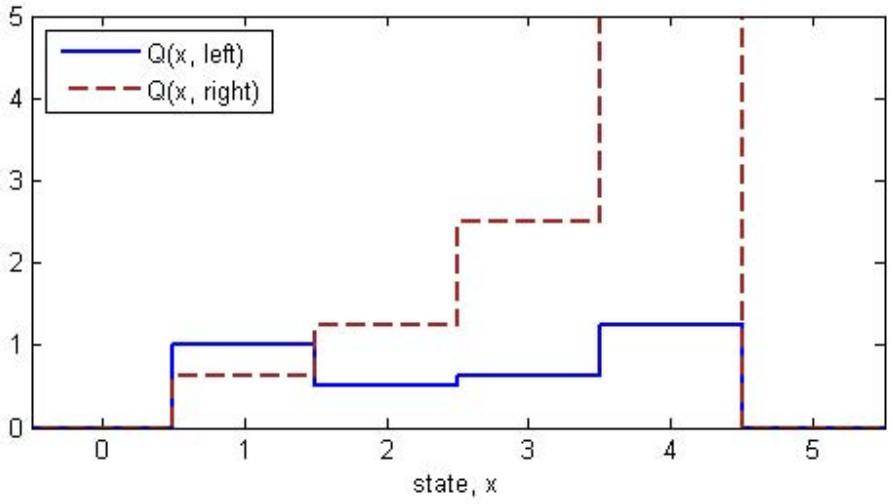
is **optimal** (achieves maximal returns)

Bellman optimality equation

$$Q^*(x, u) = \rho(x, u) + \gamma \max_{u'} Q^*(f(x, u), u')$$

Cleaning robot: Optimal solution

Discount factor $\gamma = 0.5$



Classical algorithms presented

1. Q-iteration
   Model-based: f, ρ known
2. Q-learning
   Model-free & online: f, ρ unknown,
   learn by interacting online with the system

Q-iteration

- Turn Bellman optimality equation:

$$Q^*(x,u) = \rho(x,u) + \gamma \max_{u'} Q^*(f(x,u),u')$$

into an **iterative update**:

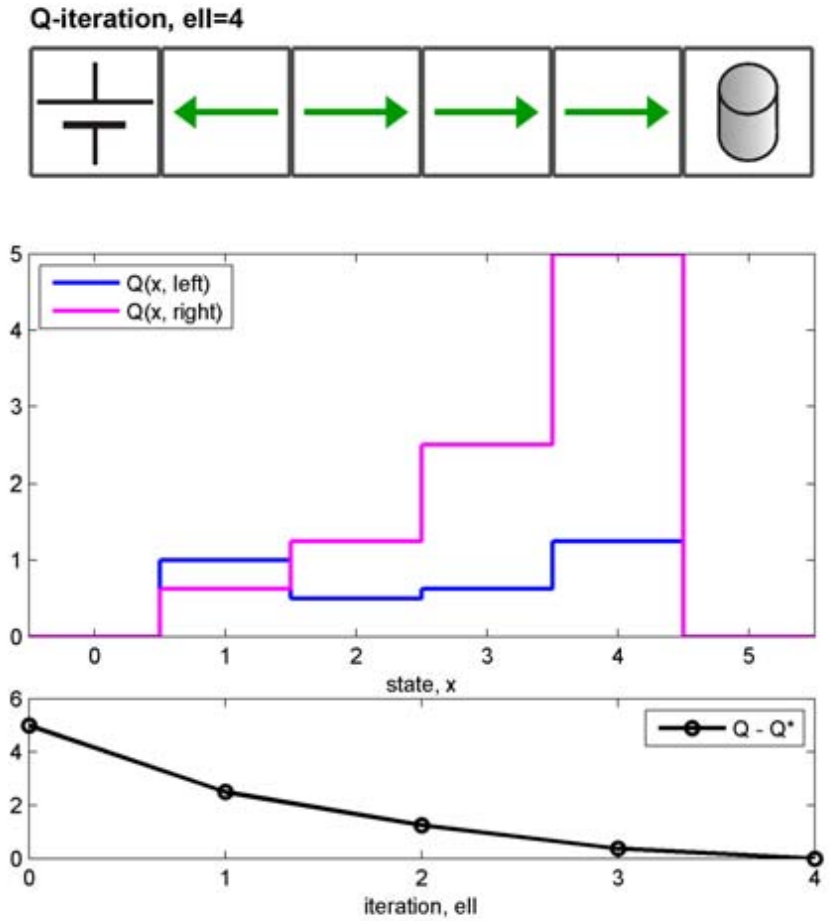
Q-iteration

repeat at each iteration ℓ
  for all x, u **do**
    $Q_{\ell+1}(x,u) \leftarrow \rho(x,u) + \gamma \max_{u'} Q_\ell(f(x,u),u')$
  end for
until convergence to Q^*

- $Q_{\ell+1}$ closer to Q^* than Q_ℓ; convergence to Q^* guaranteed
- Once Q^* available: $h^*(x) = \arg\max_u Q^*(x,u)$

Cleaning robot: Q-iteration demo

Discount factor: $\gamma = 0.5$



Q-learning

1. Take Q-iteration update:
$$Q_{\ell+1}(x,u) \leftarrow \rho(x,u) + \gamma \max_{u'} Q_\ell(f(x,u),u')$$

2. Instead of model, use **transition sample** $(x_k, u_k, x_{k+1}, r_{k+1})$ at each step k:
$$Q(x_k,u_k) \leftarrow r_{k+1} + \gamma \max_{u'} Q(x_{k+1},u')$$
Note: $x_{k+1} = f(x_k,u_k), r_{k+1} = \rho(x_k,u_k)$

3. Make update **incremental**:
$$Q(x_k,u_k) \leftarrow Q(x_k,u_k) + \alpha_k \cdot \underbrace{[r_{k+1} + \gamma \max_{u'} Q(x_{k+1},u') - Q(x_k,u_k)]}_{\text{temporal difference}}$$

$\alpha_k \in (0,1]$ learning rate

Q-learning algorithm

Q-learning

initialize x_0

for each step k **do**

take action u_k

measure x_{k+1}, receive r_{k+1}

$$Q(x_k, u_k) \leftarrow Q(x_k, u_k) + \alpha_k \cdot [r_{k+1} + \gamma \max_{u'} Q(x_{k+1}, u') - Q(x_k, u_k)]$$

end for

- Learns by online interaction

Exploration-exploitation tradeoff

- Essential condition for convergence to Q^*:
  all (x, u) pairs must be visited infinitely often

$\Rightarrow$ **Exploration** necessary:
sometimes, choose actions randomly

- **Exploitation** of current knowledge is also necessary:
  sometimes, choose actions greedily:

$$u_k = \arg\max_u Q(x_k, u)$$

Exploration-exploitation tradeoff crucial
for performance of online RL

Exploration-exploitation: ε-greedy strategy

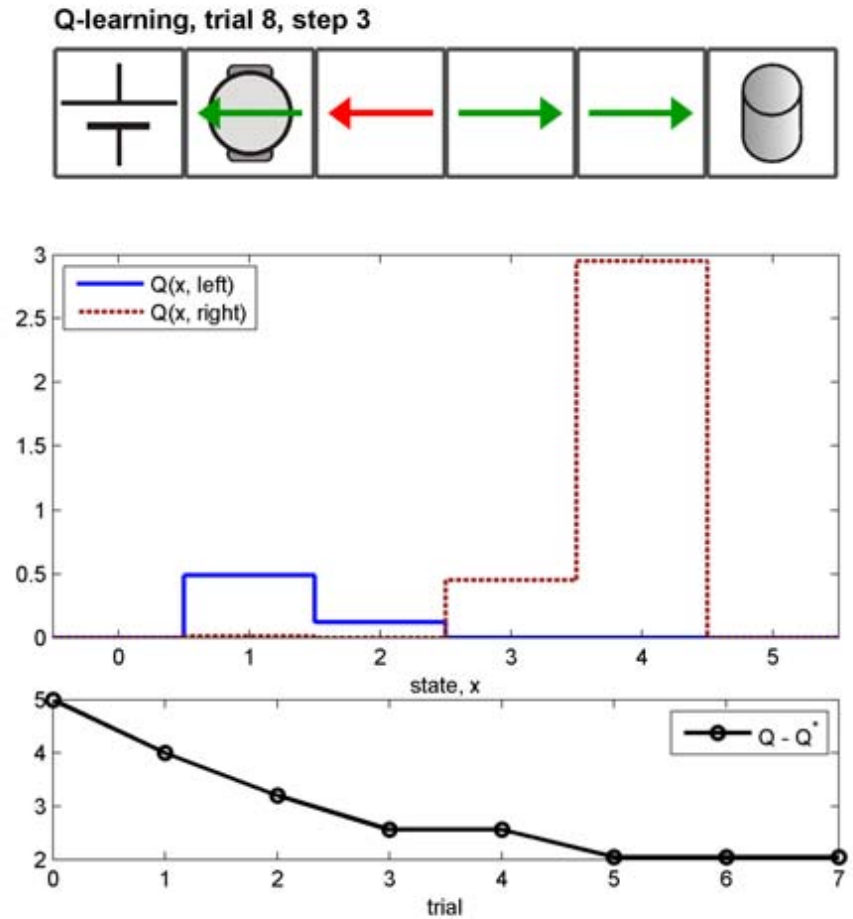
- Simple solution: **ε-greedy**

$$u_k = \begin{cases} \arg\max_u Q(x_k, u) & \text{with probability } (1 - \varepsilon_k) \\ \text{a random action} & \text{with probability } \varepsilon_k \end{cases}$$

- Exploration probability $\varepsilon_k \in (0, 1)$ is usually decreased over time

Cleaning robot: Q-learning demo

Parameters: $\alpha = 0.2$, $\varepsilon = 0.3$ (constant)
$x_0 = 2$ or 3 (randomly)



Summary

Summary

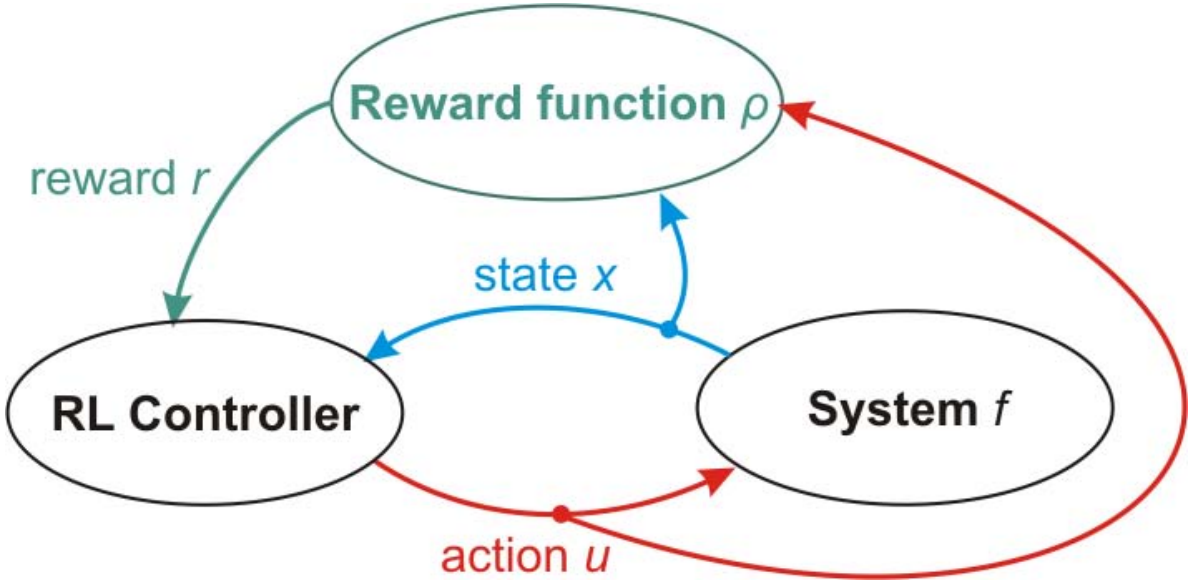
- Reinforcement learning =
  adaptive, model-free, optimal control
- Part I: small, discrete X and U – tabular representation:
  separate Q-value for each x and u

- But in real-life robot control, X, U continuous
  $\Rightarrow$ **cannot store Q-function!**
- How to solve? **Part II**

Part II: Approximate RL

Continuous states and actions

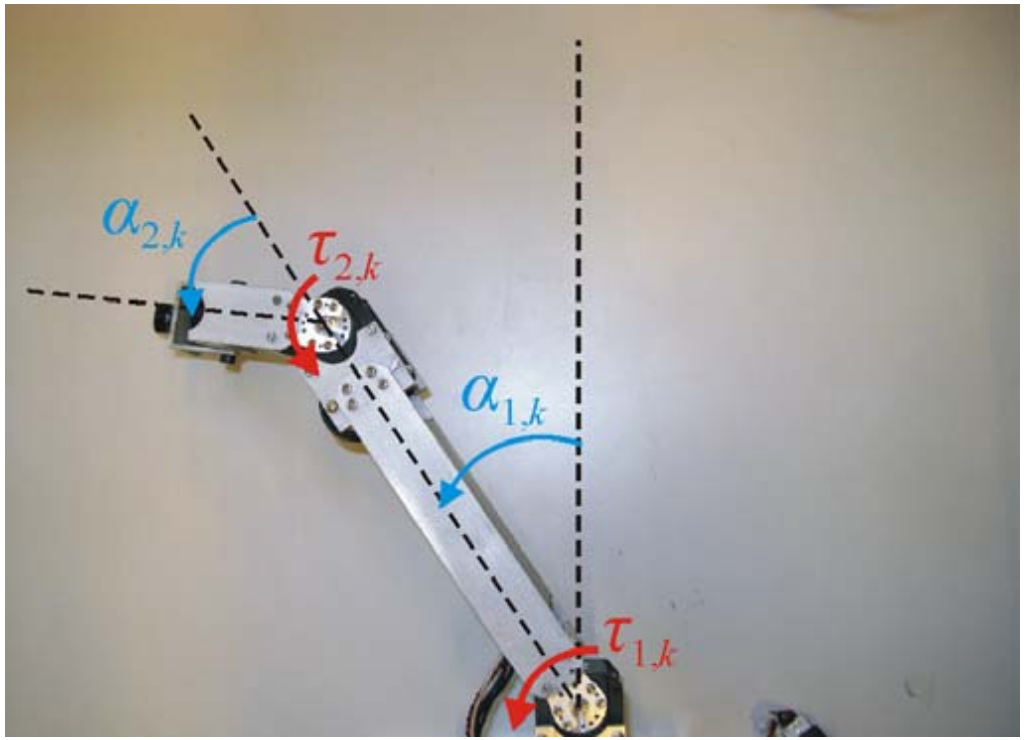
Recall: Reinforcement learning



- Interact with a system through states and actions
- Receive rewards as performance feedback

Need for approximation

- Classical RL – tabular representation
- But in real-life control, x, u **continuous**!



- **Tabular representation impossible**

Need for approximation (cont'd)

In real-life (robot) control,
must **use approximation**

Note: Approximation required
even if not using Q-functions

Approximate algorithms presented

1. Approximate Q-learning
   Representative for algorithms that use greedy policies
2. Actor-critic
   Representative for policy-gradient algorithms

Both algorithms work online

Greedy-policy algorithms

- Policy not explicitly represented
- Instead, greedy actions computed on demand from $\widehat{Q}$:

$$h(x) = \arg\max_u \widehat{Q}(x, u)$$

- Approximator must ensure **efficient** arg max **solution**

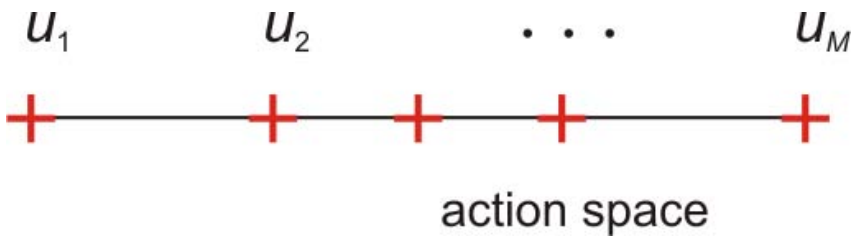
Action discretization

- Approximator must ensure efficient "arg max" solution

$\Rightarrow$ Typically: **action discretization**

- Choose M discrete actions $u_1, \ldots, u_M \in U$
  Solve "arg max" by explicit enumeration

- Example: grid discretization

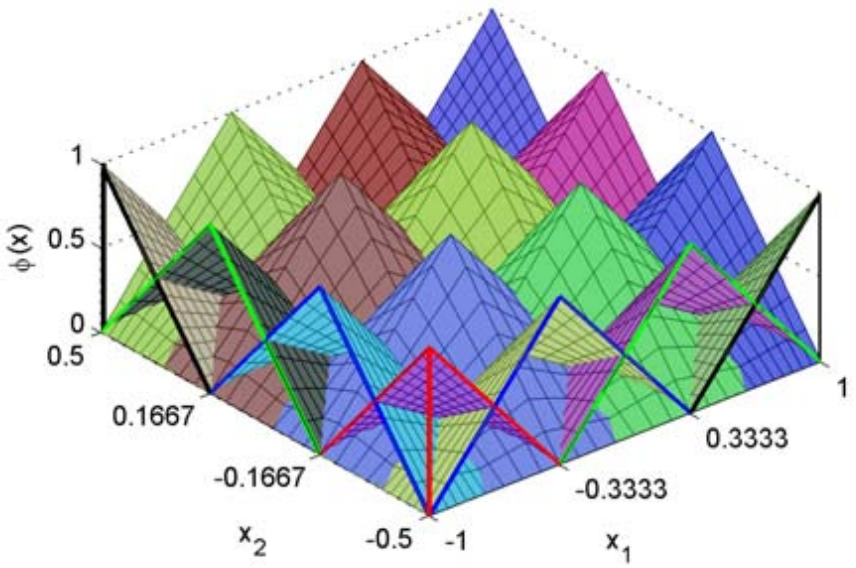


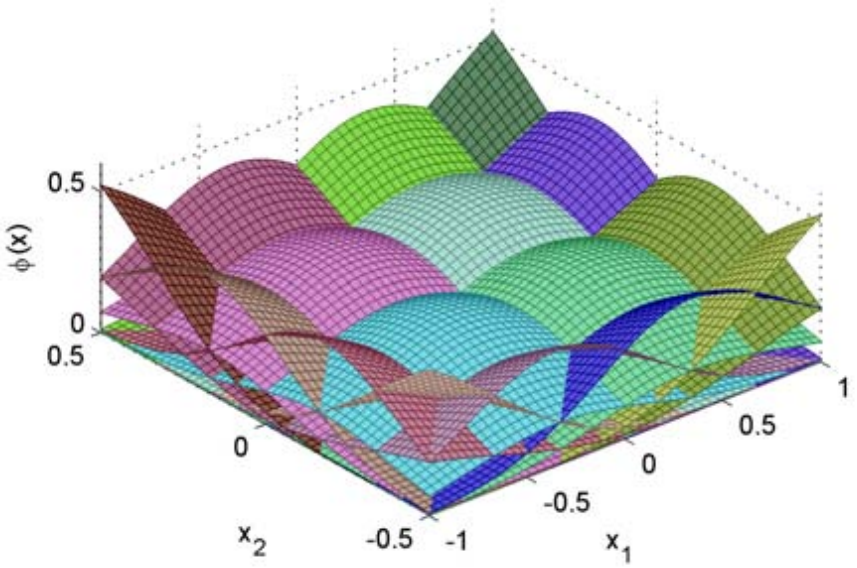
State space

- Typically: **basis functions**

$$\phi_1, \ldots, \phi_N : X \to [0, \infty)$$

- Examples: fuzzy approximation, RBF approximation





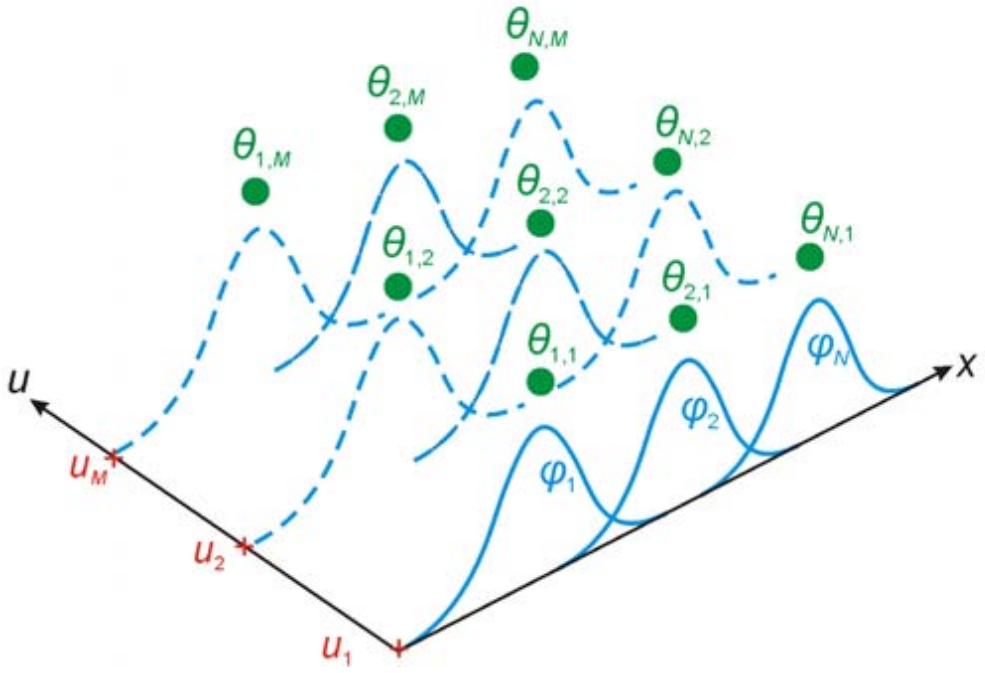
Linear Q-function parametrization

Given:

1. N basis functions $\phi_1, \ldots, \phi_N$
2. M discrete actions $u_1, \ldots, u_M$

Store:

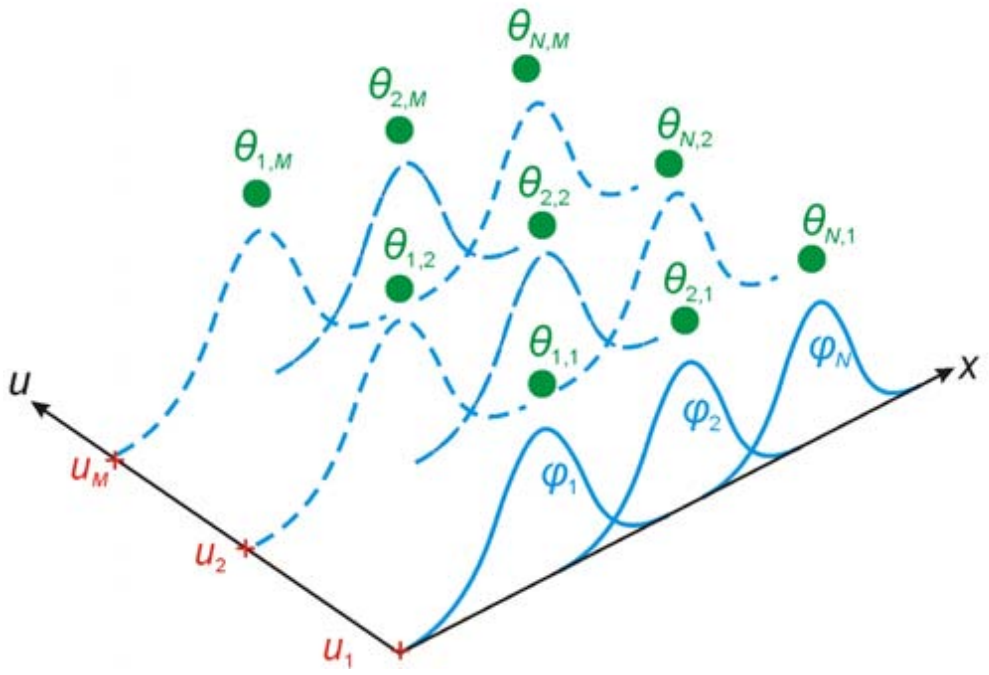
3. $N \cdot M$ **parameters** θ
   (one for each pair basis function–discrete action)



Linear Q-function parametrization (cont'd)

Approximate Q-function:

$$\widehat{Q}(x, u_j; \theta) = \sum_{i=1}^{N} \phi_i(x)\theta_{i,j} = [\phi_1(x) \dots \phi_N(x)] \begin{bmatrix} \theta_{1,j} \\ \vdots \\ \theta_{N,j} \end{bmatrix}$$



Approximate Q-learning

Recall classical Q-learning:

$$Q(x_k, u_k) \leftarrow Q(x_k, u_k) + \alpha_k [r_{k+1} + \gamma \max_{u'} Q(x_{k+1}, u') - Q(x_k, u_k)]$$

In approximate Q-learning:

- update parameters
- use approximate Q-values
- update along gradient of Q

$$\theta \leftarrow \theta + \alpha_k \left[r_{k+1} + \max_{u'} \widehat{Q}(x_{k+1}, u'; \theta) - \widehat{Q}(x_k, u_k; \theta) \right] \frac{\partial \widehat{Q}(x_k, u_k; \theta)}{\partial \theta}$$

Approximate Q-learning algorithm

Approximate Q-learning

initialize x_0, θ

for each step k **do**

take action u_k

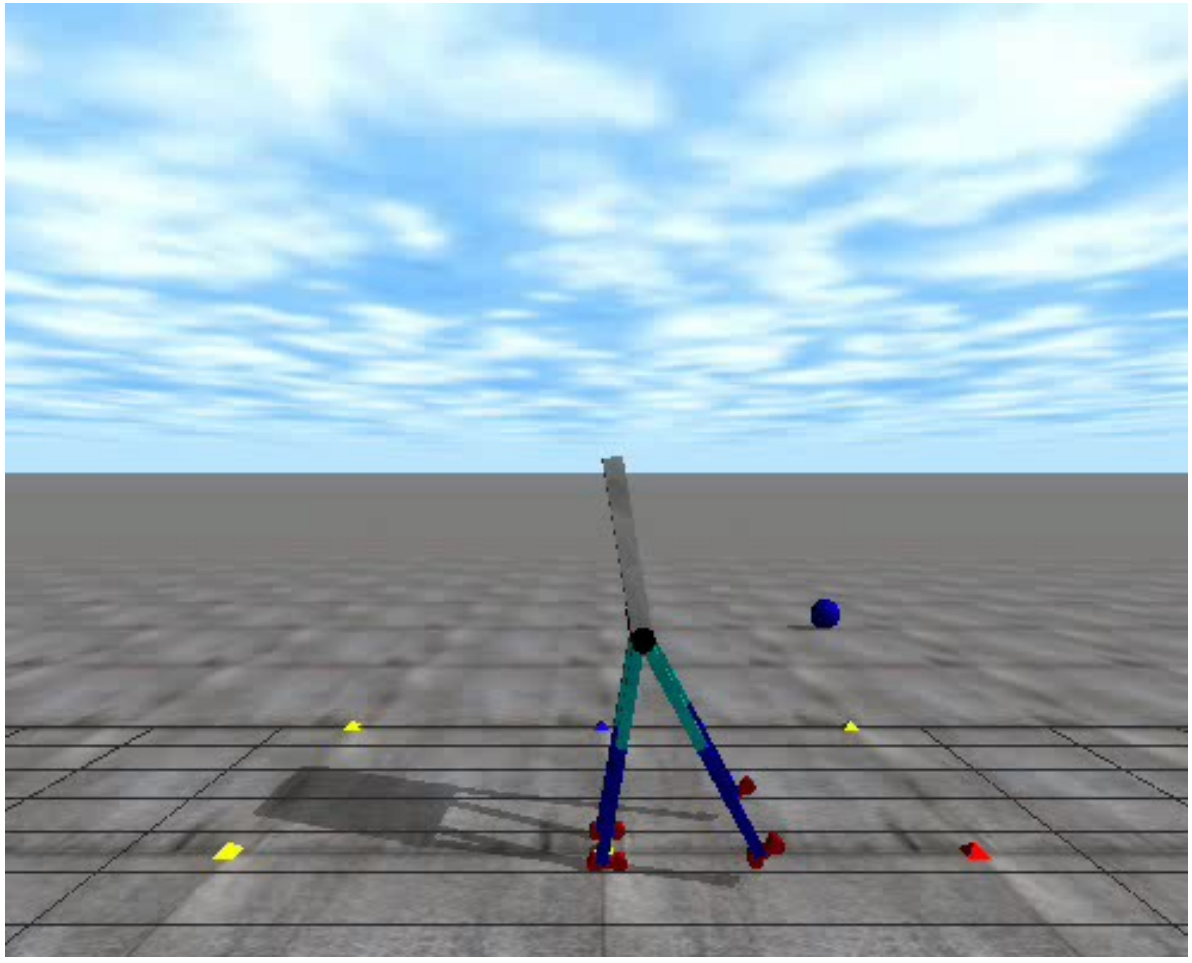
measure x_{k+1}, receive r_{k+1}

$\theta \leftarrow \theta + \alpha_k \cdot$

$$\left[r_{k+1} + \max_{u'} \widehat{Q}(x_{k+1}, u'; \theta) - \widehat{Q}(x_k, u_k; \theta) \right] \frac{\partial \widehat{Q}(x_k, u_k; \theta)}{\partial \theta}$$

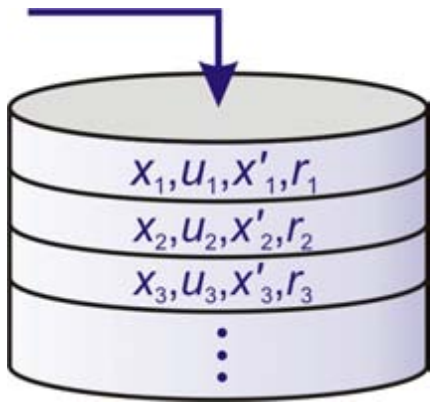
end for

Demo: Q-learning for walking robot (Erik Schuitema)



Experience replay

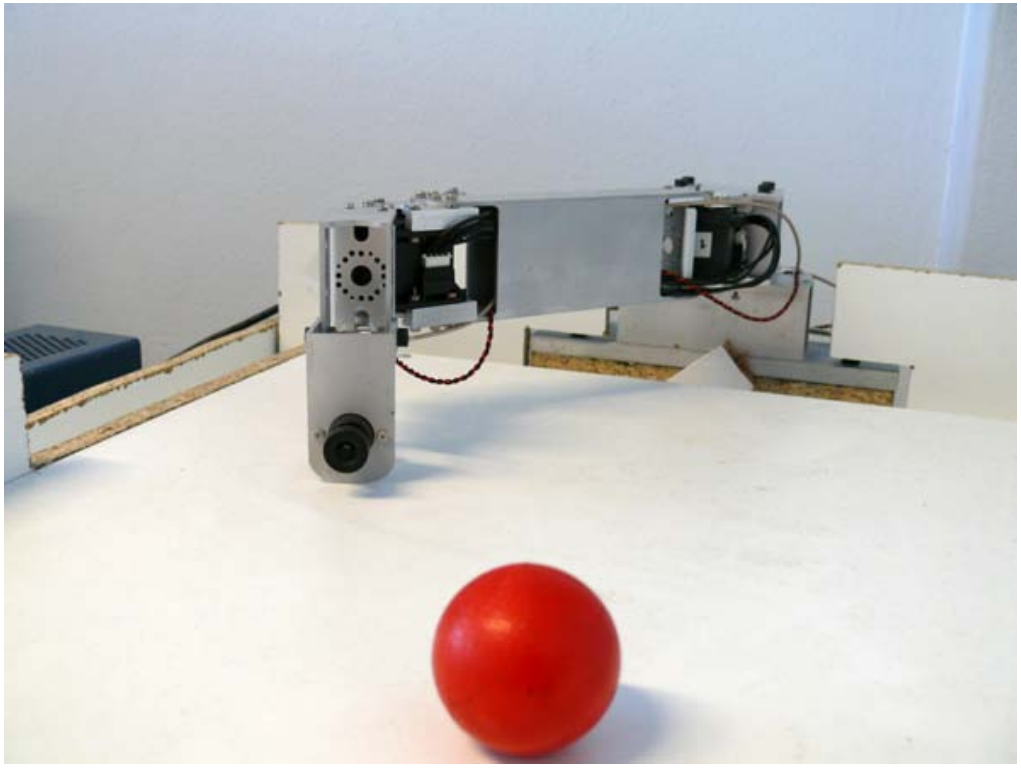
- Reuse data (experience) to accelerate learning
- Store each transition sample $(x_k, u_k, x_{k+1}, r_{k+1})$ into a database



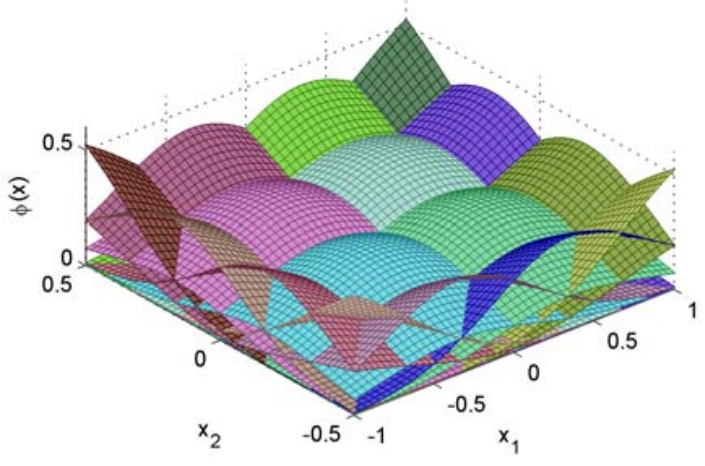
- At every step, **replay** n transitions from the database (in addition to regular updates)

Demo: ER RL for goalkeeper robot (Sander Adam)

Real-life RL control with experience replay



RBF approximation:



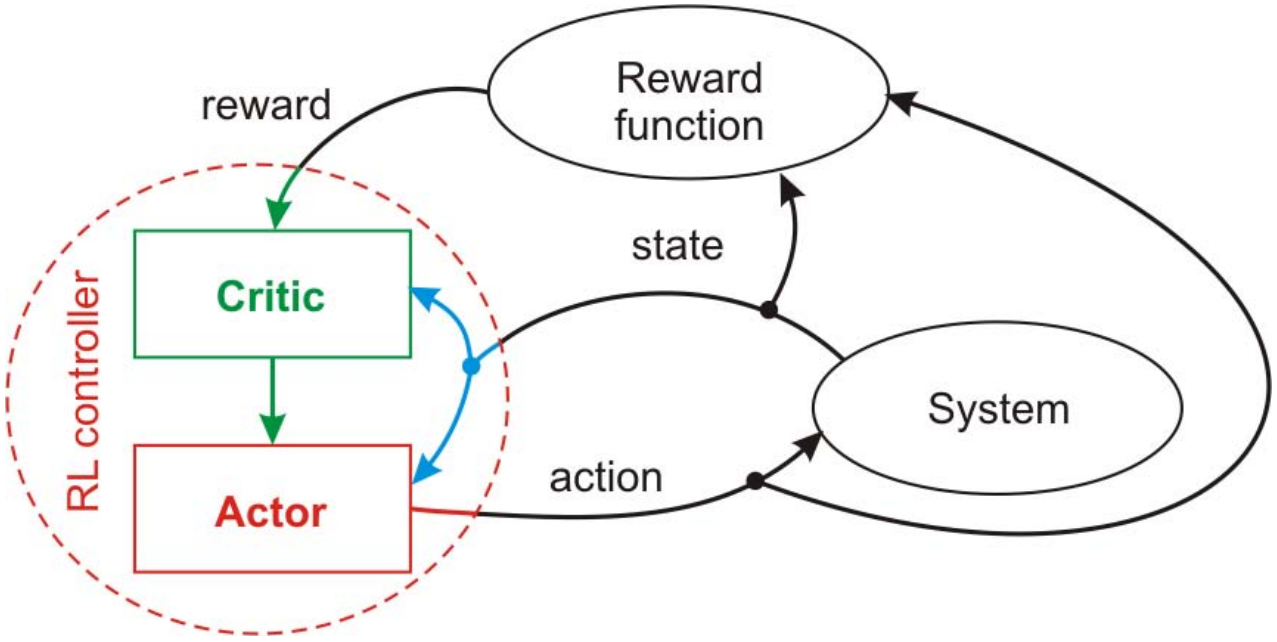
Policy gradient algorithms

- Policy explicitly represented: $\widehat{h}(x;\vartheta)$
- Parameters ϑ updated using gradient methods

Advantages in robotics:

- **Continuous actions** easy to use
- Representation can incorporate **prior knowledge**

Actor-critic scheme



- Actor: policy $\widehat{h}(x;\vartheta)$
- Critic: value function $\widehat{V}(x;\theta)$

  Greedy actions not needed, so action factored out:

$$V^h(x) = Q^h(x, h(x))$$

Critic update

Gradient on the temporal difference:

$$\theta \leftarrow \theta + \alpha_k^{\text{critic}} [r_{k+1} + \widehat{V}(x_{k+1};\theta) - \widehat{V}(x_k;\theta)] \frac{\partial \widehat{V}(x_k;\theta)}{\partial \theta}$$

$$= \theta + \alpha_k^{\text{critic}} \Delta_k \frac{\partial \widehat{V}(x_k;\theta)}{\partial \theta}$$

Recall approximate Q-learning:

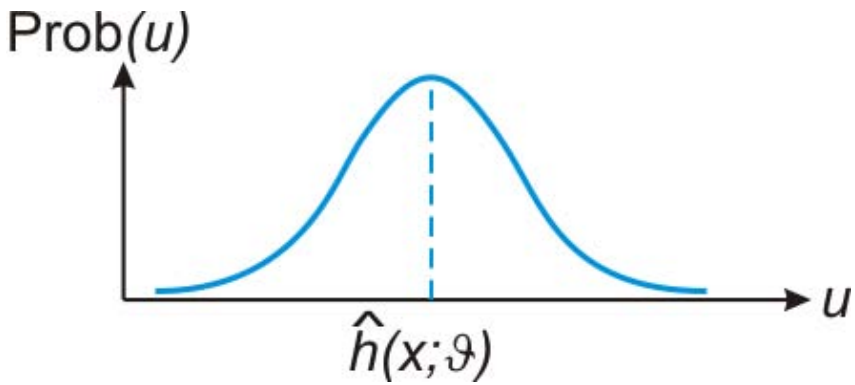
$$\theta \leftarrow \theta + \alpha_k \cdot \left[r_{k+1} + \max_{u'} \widehat{Q}(x_{k+1}, u';\theta) - \widehat{Q}(x_k, u_k;\theta) \right] \frac{\partial \widehat{Q}(x_k, u_k;\theta)}{\partial \theta}$$

Exploration

- Being online RL, actor-critic must explore
- Example: Gaussian exploration

$$u_k = \widehat{h}(x_k; \vartheta) + u'$$

where exploration u' zero-mean Gaussian



Actor update

Actor update:

$$\vartheta \leftarrow \vartheta + \alpha_k^{\text{actor}} [u_k - \widehat{h}(x_k; \vartheta)] \Delta_k \frac{\partial \widehat{h}(x_k; \vartheta)}{\partial \vartheta}$$

- If $\Delta_k > 0$, that is $r_{k+1} + \widehat{V}(x_{k+1}; \theta) > \widehat{V}(x_k; \theta)$, performance better than predicted
  $\Rightarrow$ adjust **toward** exploratory u_k
- If $\Delta_k < 0$, performance worse than predicted
  $\Rightarrow$ adjust **away** from u_k

Actor-critic algorithm

Actor-critic

initialize x_0, θ, ϑ

for each step k **do**

$u_k \leftarrow \widehat{h}(x_k;\vartheta)+$ exploration

measure x_{k+1}, receive r_{k+1}

$\Delta_k \leftarrow r_{k+1} + \widehat{V}(x_{k+1};\theta) - \widehat{V}(x_k;\theta)$

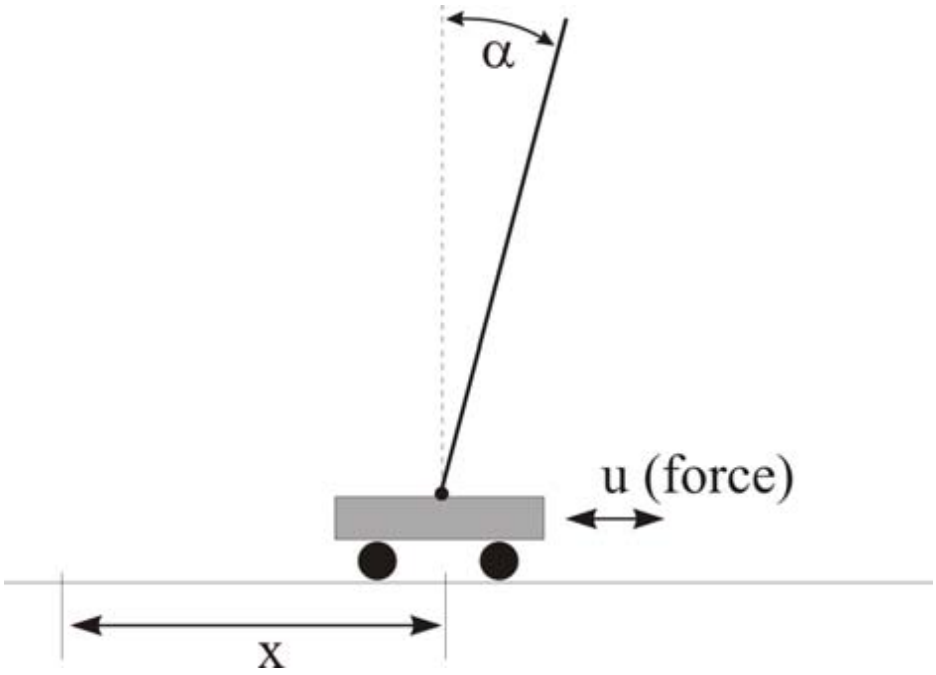
$\theta \leftarrow \theta + \alpha_k^{\text{critic}} \Delta_k \frac{\partial \widehat{V}(x_k;\theta)}{\partial \theta}$

$\vartheta \leftarrow \vartheta + \alpha_k^{\text{actor}} [u_k - \widehat{h}(x_k;\vartheta)] \Delta_k \frac{\partial \widehat{h}(x_k;\vartheta)}{\partial \vartheta}$

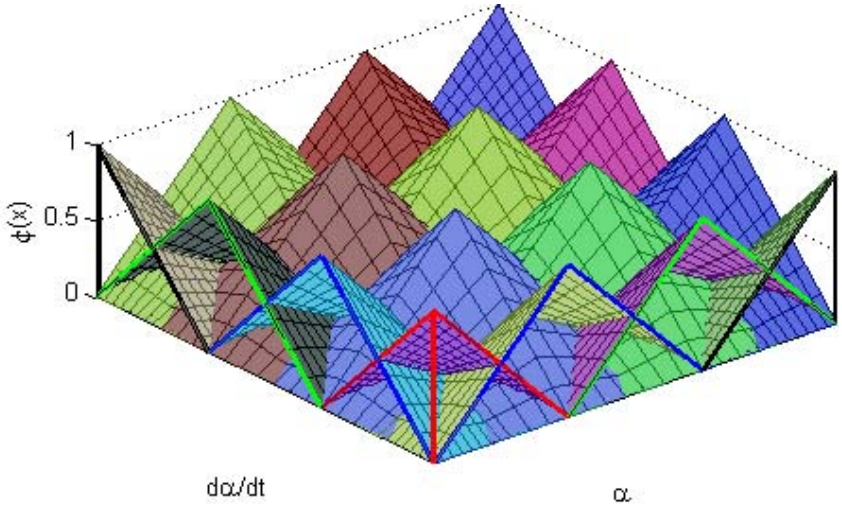
end for

- Note different learning rates for actor & critic

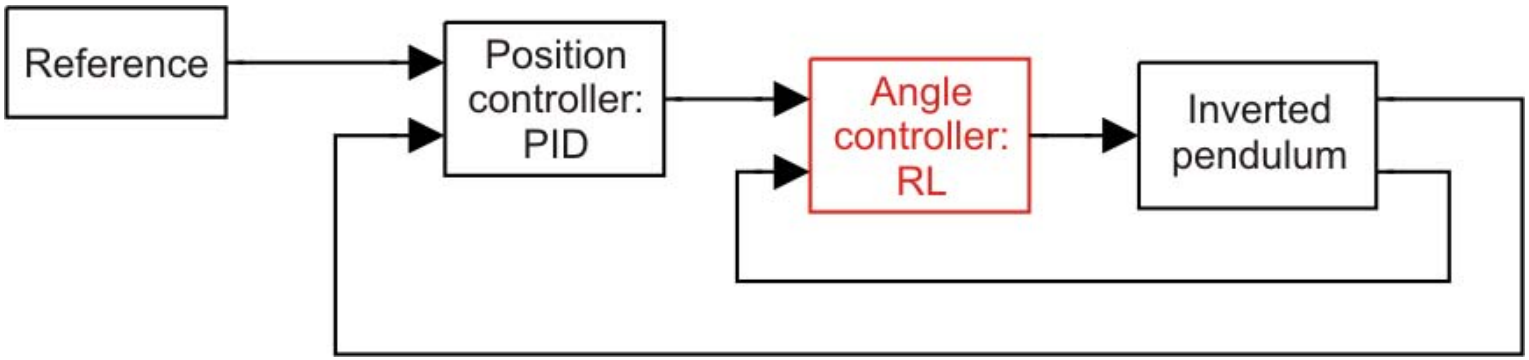
Demo: Actor-critic for the inverted pendulum



Both actor and critic:
fuzzy approximation



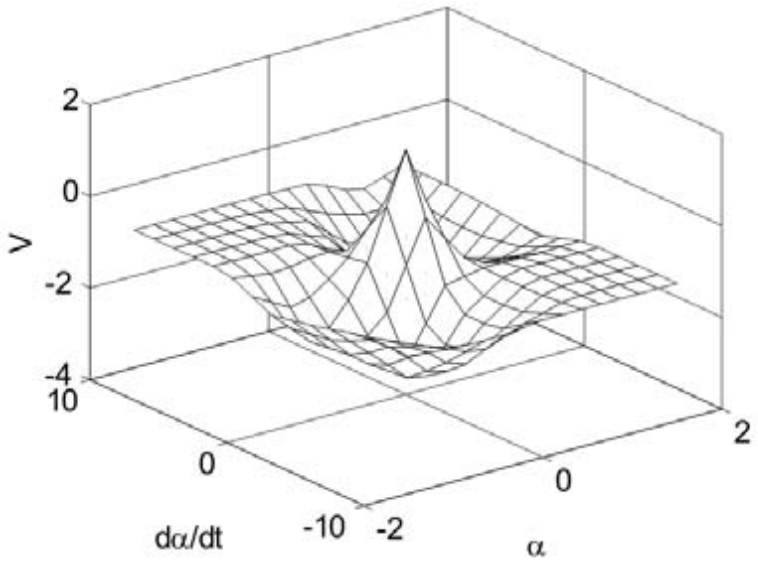
Control scheme



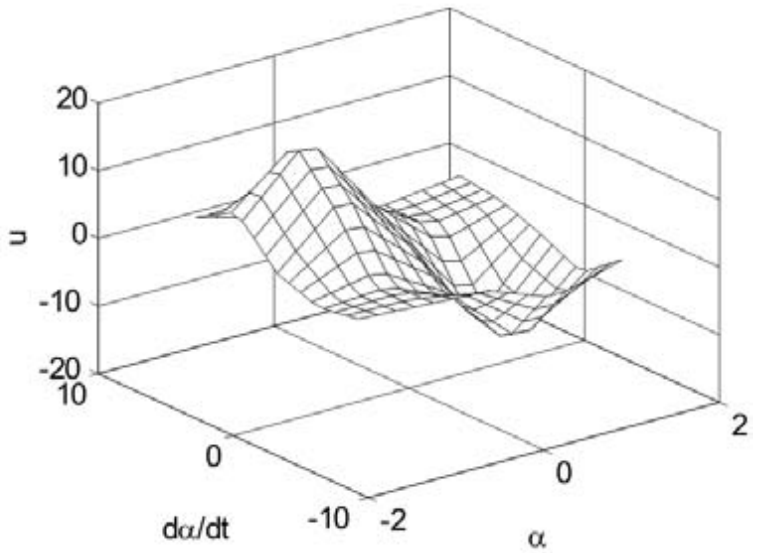
- Outer, position loop: classical PID
- Inner, angle loop: **actor-critic**

Results

Critic surface

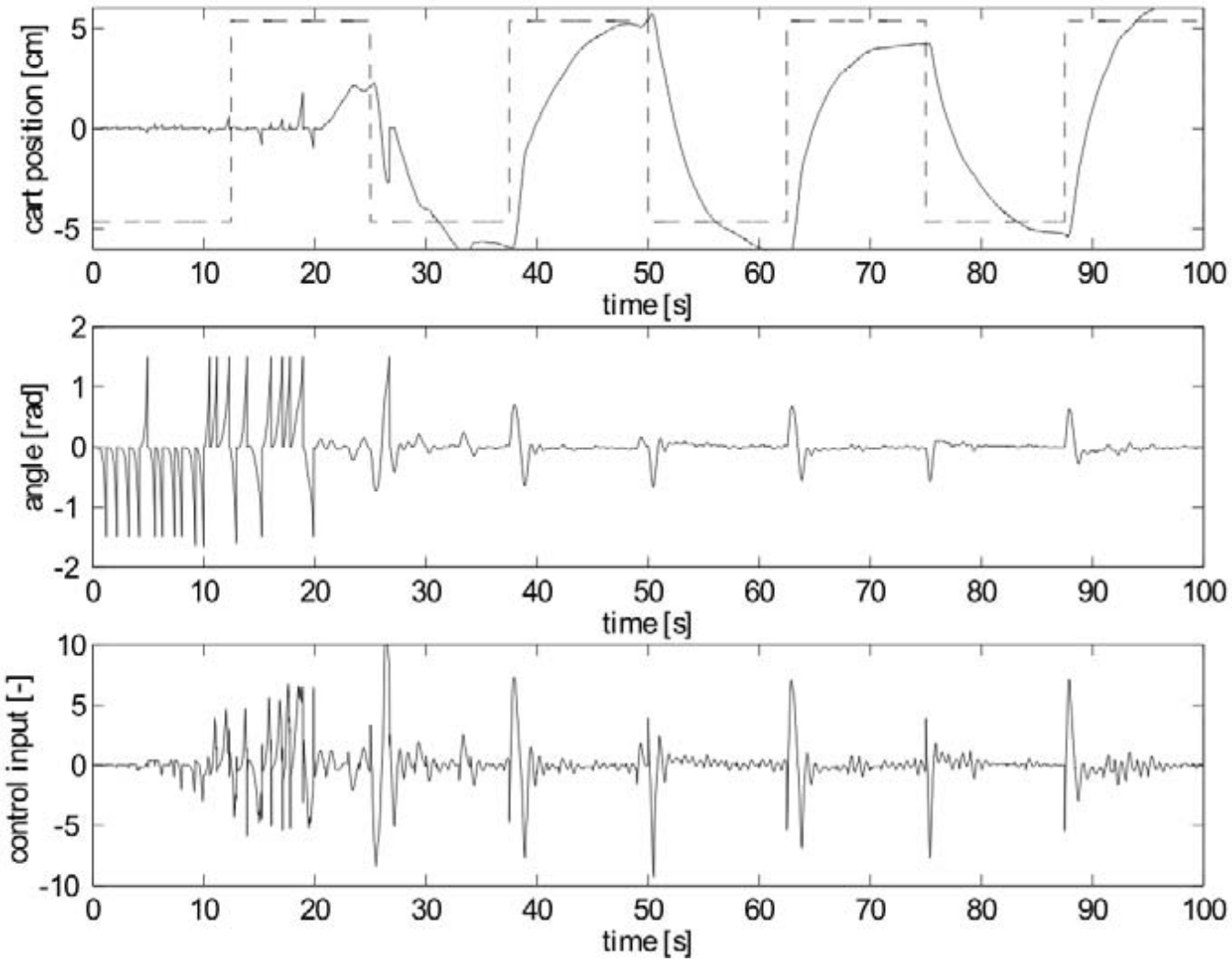


Actor surface



Results (cont'd)

Trajectory while learning



Actor-critic enhancements

- Natural policy gradient to speed up learning
- Compatible BFs for the critic to guarantee convergence
- Learn model of system to:
  - generate new data
  - improve gradient updates

Model learning for helicopter control (Stanford)

Learn system model & trajectory from expert demonstrations
Specialized RL algorithm to track trajectory

Conclusion

Reinforcement learning =
Enabling **intelligent robots** to
learn from experience