



# Reinforcement Learning

## Part II: RL using function approximation

Ivo Grondman   Robert Babuška

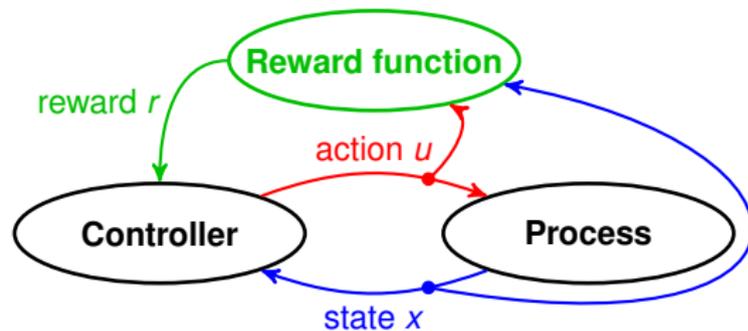
Knowledge-Based Control Systems  
2012-03-07

# Announcement

## MATLAB/SIMULINK assignment

- Hard-copy of assignment available during the break
- Work in groups of two
- Due date: Tuesday 10 April, 2012 @ 12:00 (noon!)

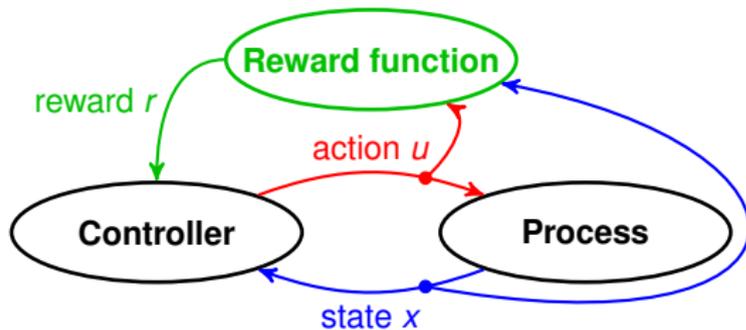
# Principle of RL



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

This lecture: **approximate RL** – continuous states & actions

# Principle of RL



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

This lecture: **approximate RL** – continuous states & actions

# Outline

- 1 Introduction
- 2 Dealing with continuous spaces
  - Approximating the Q-function
    - Fuzzy Q-iteration
  - Actor-critic methods
    - Model Learning Actor-Critic
- 3 Demo of walking robot

## Recall: Solution of the RL problem

- Q-function  $Q^\pi$  of policy  $\pi$
- Optimal Q-function  $Q^* = \max_{\pi} Q^\pi$   
Satisfies Bellman optimality equation:

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

- Optimal policy  $\pi^*$  – greedy in  $Q^*$ :

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

## Recall: Solution of the RL problem

- Q-function  $Q^\pi$  of policy  $\pi$
- Optimal Q-function  $Q^* = \max_\pi Q^\pi$   
Satisfies Bellman optimality equation:

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

- Optimal policy  $\pi^*$  – greedy in  $Q^*$ :

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

## Recall: Solution of the RL problem

- Q-function  $Q^\pi$  of policy  $\pi$
- Optimal Q-function  $Q^* = \max_\pi Q^\pi$   
Satisfies Bellman optimality equation:

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

- Optimal policy  $\pi^*$  – greedy in  $Q^*$ :

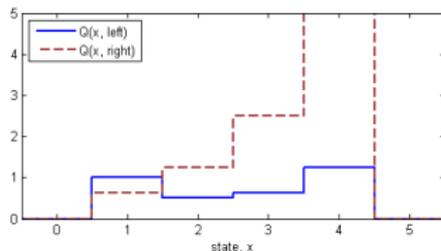
$$\pi^*(x) = \arg \max_u Q^*(x, u)$$



## Why approximation?

- Classical RL – tabular representation of Q-functions: separate Q-value for each  $x$  and  $u$
- Example: cleaning robot

0	1	.5	0.625	1.25	0
0	0.625	1.25	2.5	5	0





## Why approximation? (cont'd)

- Tabular representation contains  $|X| \cdot |U|$  elements.
  - In real-life control,  $X$ ,  $U$  **continuous!**  
Tabular representation impossible
- ⇒ need to **approximate the Q-function**



- 1 Introduction
- 2 Dealing with continuous spaces
  - Approximating the Q-function
    - Fuzzy Q-iteration
  - Actor-critic methods
    - Model Learning Actor-Critic
- 3 Demo of walking robot

# Q-function approximation

- In real-life control,  $X$ ,  $U$  continuous  
⇒ **approximate Q-function**  $\hat{Q}$  must be used
- Policy is greedy in  $\hat{Q}$ , computed on demand for given  $x$ :

$$\pi(x) = \arg \max_u \hat{Q}(x, u)$$

# Q-function approximation (cont'd)

- One option: use linearly parameterized approximation

$$\hat{Q} = \sum_{i=1}^N \theta_i \phi_i(x, u)$$

with  $\phi_i(x, u) : X \times U \mapsto \mathbb{R}$ .

- $\pi(x) = \arg \max_u \hat{Q}(x, u)$  is now a continuous optimization procedure!
- Approximator must ensure **efficient arg max solution**

# Q-function approximation (cont'd)

- One option: use linearly parameterized approximation

$$\hat{Q} = \sum_{i=1}^N \theta_i \phi_i(x, u)$$

with  $\phi_i(x, u) : X \times U \mapsto \mathbb{R}$ .

- $\pi(x) = \arg \max_u \hat{Q}(x, u)$  is now a continuous optimization procedure!
- Approximator must ensure **efficient arg max solution**

## Q-function approximation (cont'd)

- One option: use linearly parameterized approximation

$$\hat{Q} = \sum_{i=1}^N \theta_i \phi_i(x, u)$$

with  $\phi_i(x, u) : X \times U \mapsto \mathbb{R}$ .

- $\pi(x) = \arg \max_u \hat{Q}(x, u)$  is now a continuous optimization procedure!
- Approximator must ensure **efficient arg max solution**

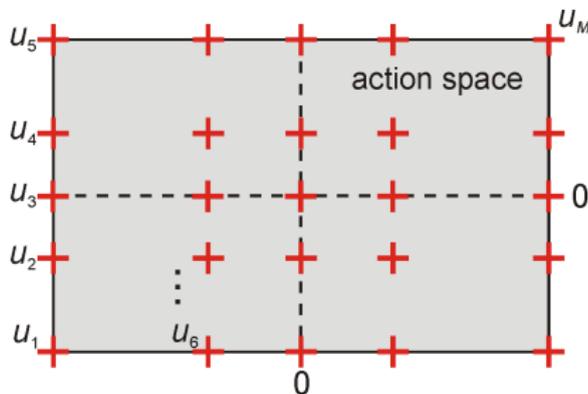


## Approximating over the action space

- Approximator must ensure efficient “arg max” solution
- ⇒ Typically: **action discretization**
- Choose  $M$  discrete actions  $u_1, \dots, u_M \in U$   
Solve “arg max” by explicit enumeration
- Example: **grid discretization**

# Approximating over the action space

- Approximator must ensure efficient “arg max” solution
- ⇒ Typically: **action discretization**
- Choose  $M$  discrete actions  $u_1, \dots, u_M \in U$   
Solve “arg max” by explicit enumeration
- Example: **grid discretization**



# Approximating over the state space

- Typically: **basis functions**

$$\phi_1, \dots, \phi_N : X \rightarrow [0, \infty)$$

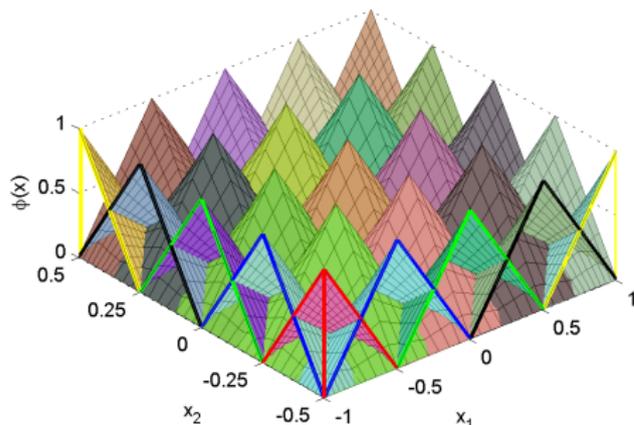
- Usually normalized:  $\sum_i \phi_i(x) = 1$
- E.g., fuzzy approximation, RBF network approximation

# Approximating over the state space

- Typically: **basis functions**

$$\phi_1, \dots, \phi_N : X \rightarrow [0, \infty)$$

- Usually normalized:  $\sum_i \phi_i(x) = 1$
- E.g., **fuzzy approximation**, **RBF network approximation**



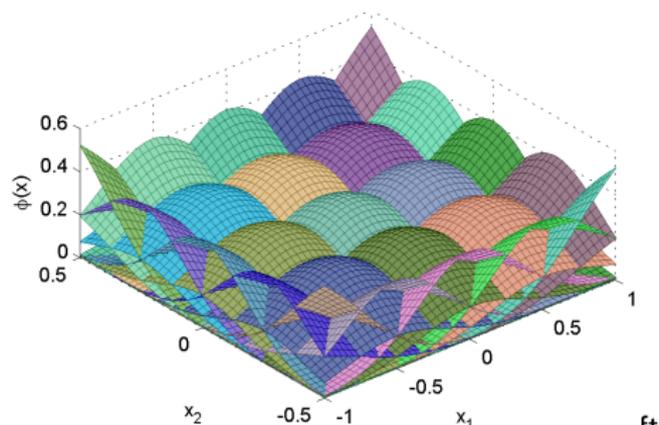
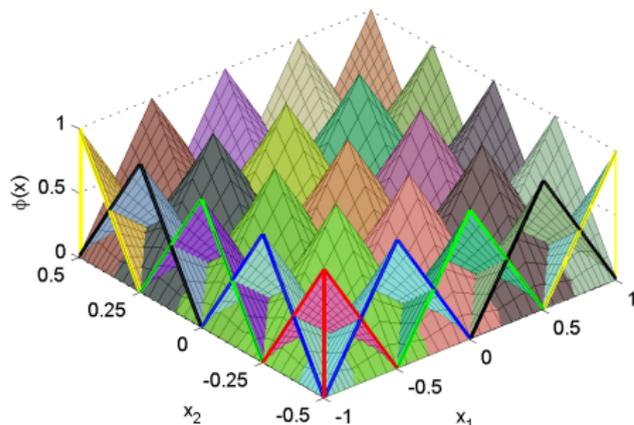


# Approximating over the state space

- Typically: **basis functions**

$$\phi_1, \dots, \phi_N : X \rightarrow [0, \infty)$$

- Usually normalized:  $\sum_i \phi_i(x) = 1$
- E.g., fuzzy approximation, RBF network approximation





# Linear Q-function approximation

Given:

- 1  $N$  basis functions  $\phi_1, \dots, \phi_N$
- 2  $M$  discrete actions  $u_1, \dots, u_M$

Store:

- 3  $N \times M$  matrix of **parameters**  $\theta$   
(one for each pair basis function–discrete action)

Approximate Q-function

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

# Linear Q-function approximation

Given:

- 1  $N$  basis functions  $\phi_1, \dots, \phi_N$
- 2  $M$  discrete actions  $u_1, \dots, u_M$

Store:

- 3  $N \times M$  matrix of **parameters**  $\theta$   
(one for each pair basis function–discrete action)

## Approximate Q-function

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

# Linear Q-function approximation

Given:

- 1  $N$  basis functions  $\phi_1, \dots, \phi_N$
- 2  $M$  discrete actions  $u_1, \dots, u_M$

Store:

- 3  $N \times M$  matrix of **parameters**  $\theta$   
(one for each pair basis function–discrete action)

## Approximate Q-function

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

# Policy from approximate Q-function

- Recall optimal policy:

$$\pi^*(x) = \underset{u}{\operatorname{arg\,max}} Q^*(x, u)$$

- Policy with discretized actions:

$$\hat{\pi}^*(x) = \underset{u_j, j=1, \dots, M}{\operatorname{arg\,max}} \hat{Q}^{\theta^*}(x, u_j)$$

( $\theta^*$  = converged parameter matrix)

# Offline, off-policy: Q-iteration, **discrete case**

- 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  $\ell$

**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  $Q^*$

- Once  $Q^*$  available:  $\pi^*(x) = \arg \max_u Q^*(x, u)$

# Offline, off-policy: Q-iteration, discrete case

- 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  $\ell$

**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**

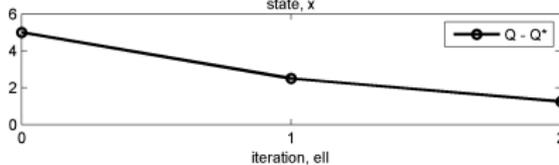
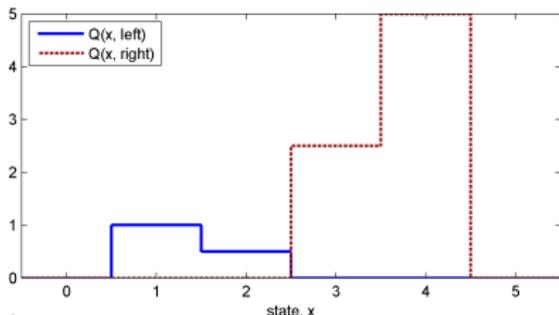
**until** convergence to  $Q^*$

- Once  $Q^*$  available:  $\pi^*(x) = \arg \max_u Q^*(x, u)$

# Cleaning robot: Q-iteration demo

Discount factor:  $\gamma = 0.5$

Q-iteration,  $\text{ell}=2$





# Cleaning robot: Q-iteration (cont'd)

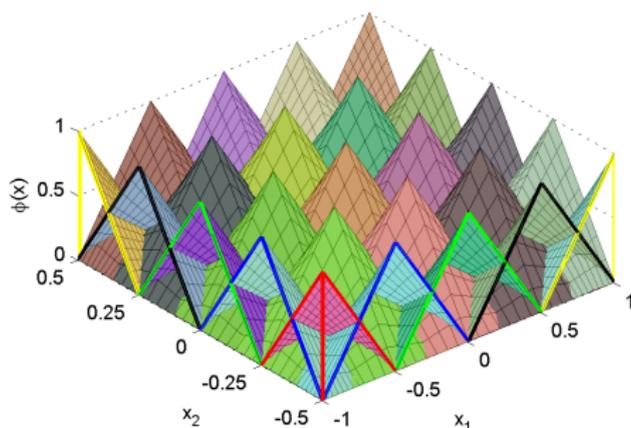
$$Q_{\ell+1}(x, u) \leftarrow \rho(x, u) + \gamma \max_{u'} Q_{\ell}(f(x, u), u')$$

	$x = 0$	$x = 1$	$x = 2$	$x = 3$	$x = 4$	$x = 5$
$Q_0$	0; 0	0; 0	0; 0	0; 0	0; 0	0; 0
$Q_1$	0; 0	1; 0	0; 0	0; 0	0; 5	0; 0
$Q_2$	0; 0	1; 0	0.5; 0	0; 2.5	0; 5	0; 0
$Q_3$	0; 0	1; 0.25	0.5; 1.25	0.25; 2.5	1.25; 5	0; 0
$Q_4$	0; 0	1; 0.625	0.5; 1.25	0.625; 2.5	1.25; 5	0; 0
$Q_5$	0; 0	1; 0.625	0.5; 1.25	0.625; 2.5	1.25; 5	0; 0
$\pi^*$	*	-1	1	1	1	*
$V^*$	0	1	1.25	2.5	5	0

Note:  $Q_{\ell} = Q(x, \text{left}) ; Q(x, \text{right})$

# Fuzzy approximator

- Basis functions: **pyramidal membership functions** (MFs)  
= cross-product of triangular MFs



- Each MF  $i$  has core (center)  $x_i$
- $\theta_{i,j}$  can be seen as  $\widehat{Q}(x_i, u_j)$

# Fuzzy Q-iteration

Recall classical Q-iteration:

**repeat** at each iteration  $\ell$

**for all**  $x, u$  **do**

$$Q_{\ell+1}(x, u) = \rho(x, u) + \gamma \max_{u'} Q_{\ell}(f(x, u), u')$$

**end for**

**until** convergence

## Fuzzy Q-iteration

**repeat** at each iteration  $\ell$

**for all** cores  $x_i$ , discrete actions  $u_j$  **do**

$$\theta_{\ell+1,i,j} = \rho(x_i, u_j) + \gamma \max_{j'} \hat{Q}^{\theta_{\ell}}(f(x_i, u_j), u_{j'})$$

**end for**

**until** convergence

# Another example: Inverted pendulum swing-up

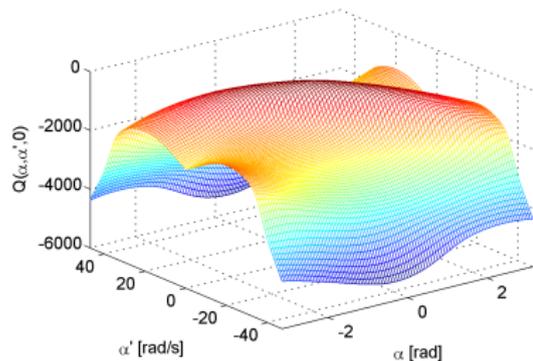


- $x = [\text{angle } \alpha, \text{ velocity } \dot{\alpha}]^T$
  - $u = \text{voltage}$
  - $\rho(x, u) = -x^T \begin{bmatrix} 5 & 0 \\ 0 & 0.1 \end{bmatrix} x - u^T 1 u$
  - Discount factor  $\gamma = 0.98$
- 
- **Goal:** stabilize pointing up
  - Insufficient actuation  $\Rightarrow$  need to swing back & forth

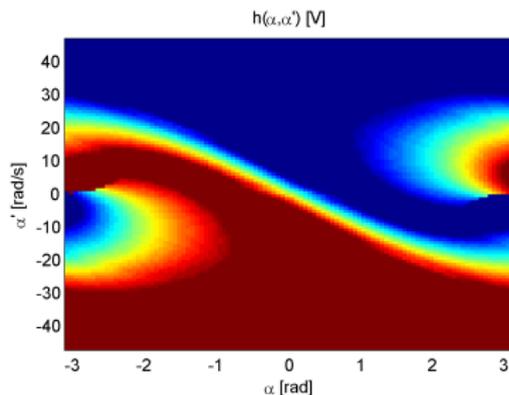


# Inverted pendulum: Near-optimal solution

Left: Q-function for  $u = 0$



Right: policy

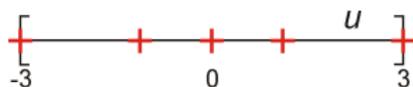
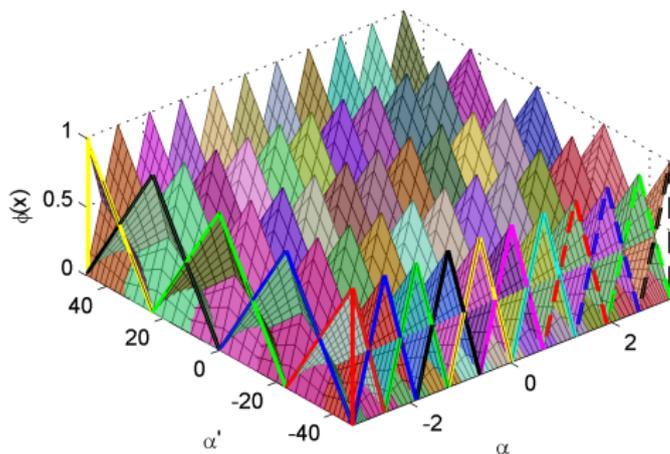




# Inverted pendulum: Fuzzy Q-iteration demo

MFs:  $41 \times 21$  equidistant grid

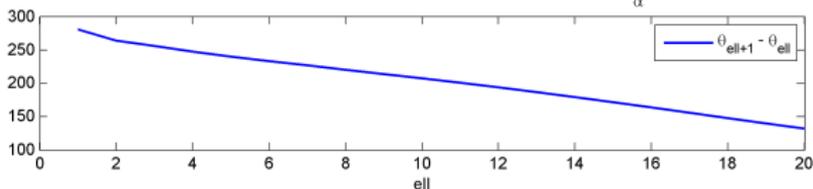
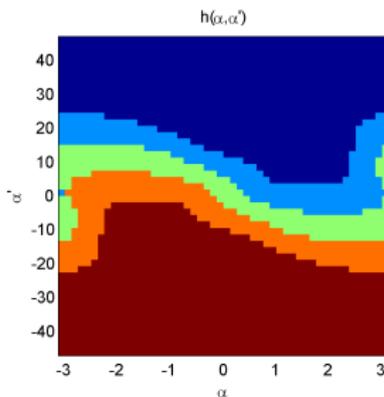
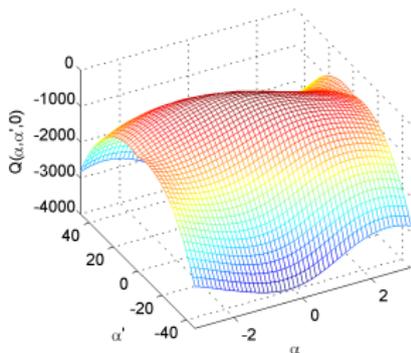
Discretization: 5 actions, logarithmically spaced around 0



# Inverted pendulum: Fuzzy Q-iteration demo

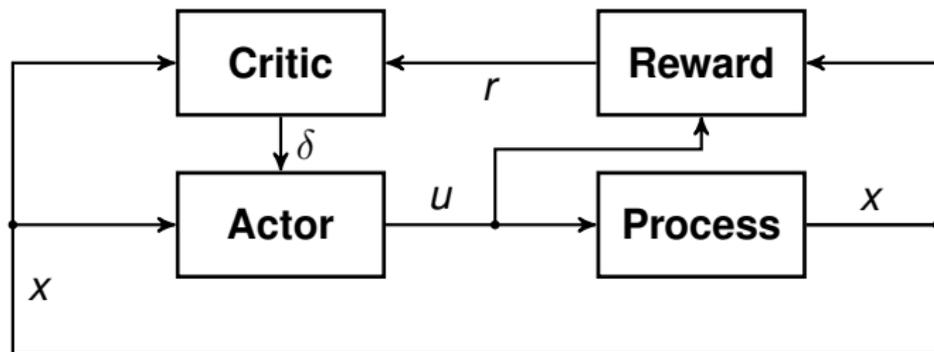
## Demo

Fuzzy Q-iteration, ell=20



- 1 Introduction
- 2 Dealing with continuous spaces
  - Approximating the Q-function
    - Fuzzy Q-iteration
  - Actor-critic methods
    - Model Learning Actor-Critic
- 3 Demo of walking robot

# Ingredients



- Explicitly separated value function and policy
- **Actor** = control policy  $\pi(x)$
- **Critic** = state value function  $V(x)$

# Continuous action/state space

To deal with continuity:

- Actor parameterized in  $\varphi$ :  $\hat{\pi}(x, \varphi)$
- Critic parameterized in  $\theta$ :  $\hat{V}(x, \theta)$

Parameters  $\varphi$  and  $\theta$  have finite size, but approximate functions on continuous (infinitely large) spaces!

## Remarks on lecture notes

Paragraph 9.4.6 has some peculiarities:

- Equation (9.49) should not contain  $u_k$
- Time indices  $k$  and  $i$  are used in a confusing way
- Reward not usually  $r_k \in \{0, -1\}$ , but any value

Terminology:

- “Performance Evaluation Unit” = reward function
- “Control Unit” = actor (i.e. the policy)
- “Stochastic Action Modifier” = exploration

## Remarks on lecture notes

Paragraph 9.4.6 has some peculiarities:

- Equation (9.49) should not contain  $u_k$
- Time indices  $k$  and  $i$  are used in a confusing way
- Reward not usually  $r_k \in \{0, -1\}$ , but any value

Terminology:

- “Performance Evaluation Unit” = reward function
- “Control Unit” = actor (i.e. the policy)
- “Stochastic Action Modifier” = exploration



# Algorithm

On-policy: **find**  $Q^\pi$ , improve  $\pi$ , repeat

- 1 Take Bellman equation for  $V^\pi$ , at some  $x_k$ :

$$V^\pi(x) = \rho(x, \pi(x)) + \gamma V^\pi(f(x, \pi(x)))$$

- 2 Take temporal difference  $\Delta$ :

$$\Delta = \rho(x, \pi(x)) + \gamma V^\pi(f(x, \pi(x))) - V^\pi(x)$$

- 3 Use sample  $(x_k, u_k, x_{k+1}, r_{k+1})$  at each step  $k$  and parameterized  $V$ :

$$\Delta_k = r_{k+1} + \gamma \hat{V}^\pi(x_{k+1}, \theta_k) - \hat{V}^\pi(x_k, \theta_k)$$

Note:  $u_k = \hat{\pi}(x_k, \varphi_k) + \tilde{u}_k$ ,  $\hat{\pi}$  = actor,  $\tilde{u}_k$  = **exploration**



# Algorithm

On-policy: **find**  $Q^\pi$ , improve  $\pi$ , repeat

- 1 Take Bellman equation for  $V^\pi$ , at some  $x_k$ :

$$V^\pi(x) = \rho(x, \pi(x)) + \gamma V^\pi(f(x, \pi(x)))$$

- 2 Take temporal difference  $\Delta$ :

$$\Delta = \rho(x, \pi(x)) + \gamma V^\pi(f(x, \pi(x))) - V^\pi(x)$$

- 3 Use sample  $(x_k, u_k, x_{k+1}, r_{k+1})$  at each step  $k$  and parameterized  $V$ :

$$\Delta_k = r_{k+1} + \gamma \hat{V}^\pi(x_{k+1}, \theta_k) - \hat{V}^\pi(x_k, \theta_k)$$

Note:  $u_k = \hat{\pi}(x_k, \varphi_k) + \tilde{u}_k$ ,  $\hat{\pi}$  = actor,  $\tilde{u}_k$  = **exploration**

# Algorithm

On-policy: **find**  $Q^\pi$ , improve  $\pi$ , repeat

- 1 Take Bellman equation for  $V^\pi$ , at some  $x_k$ :

$$V^\pi(x) = \rho(x, \pi(x)) + \gamma V^\pi(f(x, \pi(x)))$$

- 2 Take temporal difference  $\Delta$ :

$$\Delta = \rho(x, \pi(x)) + \gamma V^\pi(f(x, \pi(x))) - V^\pi(x)$$

- 3 Use sample  $(x_k, u_k, x_{k+1}, r_{k+1})$  at each step  $k$  and parameterized  $V$ :

$$\Delta_k = r_{k+1} + \gamma \hat{V}^\pi(x_{k+1}, \theta_k) - \hat{V}^\pi(x_k, \theta_k)$$

Note:  $u_k = \hat{\pi}(x_k, \varphi_k) + \tilde{u}_k$ ,  $\hat{\pi}$  = actor,  $\tilde{u}_k$  = **exploration**

# Algorithm (cont'd)

- 4 Use  $\Delta_k$  for critic update:

$$\theta_{k+1} = \theta_k + \alpha_c \Delta_k \left. \frac{\partial \hat{V}(x, \theta)}{\partial \theta} \right|_{\substack{x=x_k \\ \theta=\theta_k}}$$

$\alpha_c > 0$ : learning rate of critic

- $\Delta_k > 0 \Rightarrow r_{k+1} + \gamma \hat{V}^\pi(x_{k+1}, \theta_k) > \hat{V}^\pi(x_k, \theta_k)$ : old estimate too low, increase  $\hat{V}$ .
- $\Delta_k < 0 \Rightarrow r_{k+1} + \gamma \hat{V}^\pi(x_{k+1}, \theta_k) < \hat{V}^\pi(x_k, \theta_k)$ : old estimate too high, decrease  $\hat{V}$ .

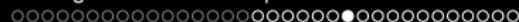
# Algorithm (cont'd)

- 4 Use  $\Delta_k$  for critic update:

$$\theta_{k+1} = \theta_k + \alpha_c \Delta_k \left. \frac{\partial \hat{V}(x, \theta)}{\partial \theta} \right|_{\substack{x=x_k \\ \theta=\theta_k}}$$

$\alpha_c > 0$ : learning rate of critic

- $\Delta_k > 0 \Rightarrow r_{k+1} + \gamma \hat{V}^\pi(x_{k+1}, \theta_k) > \hat{V}^\pi(x_k, \theta_k)$ : old estimate too low, increase  $\hat{V}$ .
- $\Delta_k < 0 \Rightarrow r_{k+1} + \gamma \hat{V}^\pi(x_{k+1}, \theta_k) < \hat{V}^\pi(x_k, \theta_k)$ : old estimate too high, decrease  $\hat{V}$ .



## Algorithm (cont'd)

Recall:  $u_k = \hat{\pi}(x_k, \varphi_k) + \tilde{u}_k$ ,  $\hat{\pi}$  = actor,  $\tilde{u}_k$  = **exploration**

- 5 Use  $\Delta_k$  and exploration term  $\tilde{u}_k$  for actor update:

$$\varphi_{k+1} = \varphi_k + \alpha_a \Delta_k \tilde{u}_k \left. \frac{\partial \hat{\pi}(x, \varphi)}{\partial \varphi} \right|_{\substack{x=x_k \\ \varphi=\varphi_k}}$$

$\alpha_a \in (0, 1]$ : learning rate of actor

- Product  $\Delta_k \tilde{u}_k$  determines sign in update
- $\Delta_k > 0 \Rightarrow r_{k+1} + \gamma \hat{V}^\pi(x_{k+1}, \theta_k) > \hat{V}^\pi(x_k, \theta_k)$ :  $\tilde{u}_k$  had positive effect on performance. Move in direction of  $u_k$ .
- $\Delta_k < 0 \Rightarrow r_{k+1} + \gamma \hat{V}^\pi(x_{k+1}, \theta_k) < \hat{V}^\pi(x_k, \theta_k)$ :  $\tilde{u}_k$  had negative effect on performance. Move away from  $u_k$ .



## Algorithm (cont'd)

Recall:  $u_k = \hat{\pi}(x_k, \varphi_k) + \tilde{u}_k$ ,  $\hat{\pi} = \text{actor}$ ,  $\tilde{u}_k = \text{exploration}$

- 5 Use  $\Delta_k$  and exploration term  $\tilde{u}_k$  for actor update:

$$\varphi_{k+1} = \varphi_k + \alpha_a \Delta_k \tilde{u}_k \left. \frac{\partial \hat{\pi}(x, \varphi)}{\partial \varphi} \right|_{\substack{x=x_k \\ \varphi=\varphi_k}}$$

$\alpha_a \in (0, 1]$ : learning rate of actor

- Product  $\Delta_k \tilde{u}_k$  determines sign in update
- $\Delta_k > 0 \Rightarrow r_{k+1} + \gamma \hat{V}^\pi(x_{k+1}, \theta_k) > \hat{V}^\pi(x_k, \theta_k)$ :  $\tilde{u}_k$  had positive effect on performance. Move in direction of  $u_k$ .
- $\Delta_k < 0 \Rightarrow r_{k+1} + \gamma \hat{V}^\pi(x_{k+1}, \theta_k) < \hat{V}^\pi(x_k, \theta_k)$ :  $\tilde{u}_k$  had negative effect on performance. Move away from  $u_k$ .

## Algorithm (cont'd)

Recall:  $u_k = \hat{\pi}(x_k, \varphi_k) + \tilde{u}_k$ ,  $\hat{\pi}$  = actor,  $\tilde{u}_k$  = **exploration**

- 5 Use  $\Delta_k$  and exploration term  $\tilde{u}_k$  for actor update:

$$\varphi_{k+1} = \varphi_k + \alpha_a \Delta_k \tilde{u}_k \left. \frac{\partial \hat{\pi}(x, \varphi)}{\partial \varphi} \right|_{\substack{x=x_k \\ \varphi=\varphi_k}}$$

$\alpha_a \in (0, 1]$ : learning rate of actor

- Product  $\Delta_k \tilde{u}_k$  determines sign in update
- $\Delta_k > 0 \Rightarrow r_{k+1} + \gamma \hat{V}^\pi(x_{k+1}, \theta_k) > \hat{V}^\pi(x_k, \theta_k)$ :  $\tilde{u}_k$  had positive effect on performance. Move in direction of  $u_k$ .
- $\Delta_k < 0 \Rightarrow r_{k+1} + \gamma \hat{V}^\pi(x_{k+1}, \theta_k) < \hat{V}^\pi(x_k, \theta_k)$ :  $\tilde{u}_k$  had negative effect on performance. Move away from  $u_k$ .

# Complete actor-critic algorithm

## Actor-critic

**for** every trial **do**

initialize  $x_0$ , choose initial action  $u_0 = \tilde{u}_0$

**repeat** for each step  $k$

apply  $u_k$ , measure  $x_{k+1}$ , receive  $r_{k+1}$

choose **next** action  $u_{k+1} = \hat{\pi}(x_{k+1}, \varphi_k) + \tilde{u}_{k+1}$

$\Delta_k = r_{k+1} + \hat{V}(x_{k+1}, \theta_k) - \hat{V}(x_k, \theta_k)$

$\theta_{k+1} = \theta_k + \alpha_c \Delta_k \left. \frac{\partial \hat{V}(x, \theta)}{\partial \theta} \right|_{\substack{x=x_k \\ \theta=\theta_k}}$

$\varphi_{k+1} = \varphi_k + \alpha_a \Delta_k \tilde{u}_k \left. \frac{\partial \hat{\pi}(x, \varphi)}{\partial \varphi} \right|_{\substack{x=x_k \\ \varphi=\varphi_k}}$

**until** terminal state

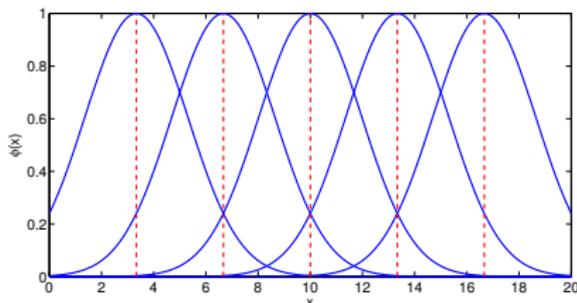
**end for**

# Radial Basis Functions

$$\hat{f}(x) = \theta^T \tilde{\phi}(x)$$

where  $\tilde{\phi}(x)$  is a column vector with the value of normalized RBFs:

$$\tilde{\phi}_i(x) = \frac{\phi_i(x)}{\sum_j \phi_j(x)} \quad \text{with} \quad \phi_i(x) = e^{-\frac{1}{2}(x-c_i)^T B^{-1}(x-c_i)}$$



# Evolution of a policy

Figure: Value function and policy in learning phase.

# Policy after saturation

Figure: Trajectory of pendulum.

# Final policy

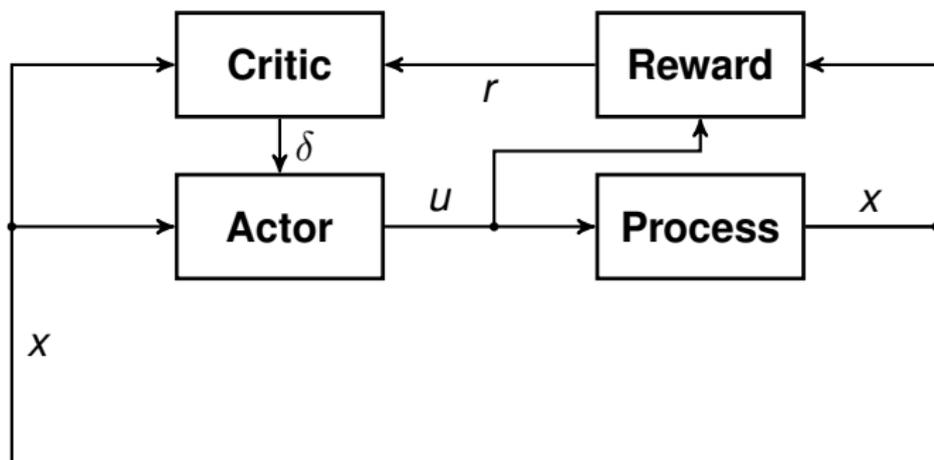


Figure: Solution to pendulum swing-up problem.

# Model Learning Actor-Critic

## Main idea

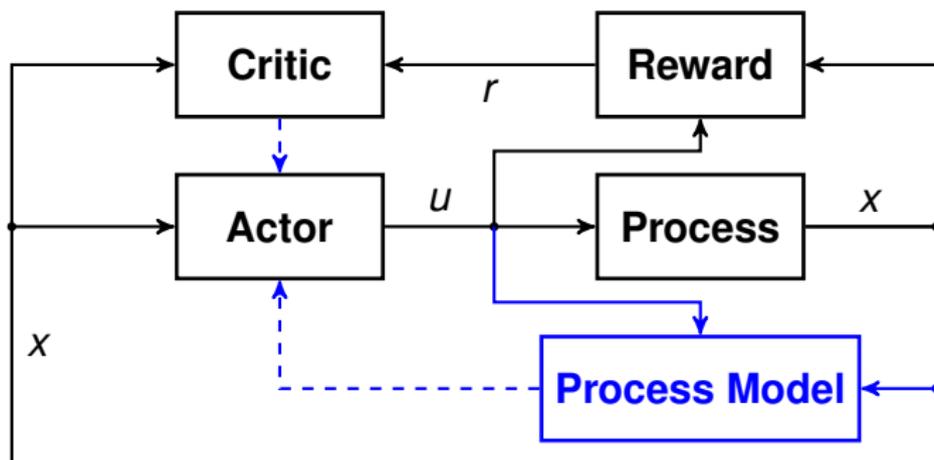
Use a learned process model to get a more efficient policy update.



# Model Learning Actor-Critic

## Main idea

Use a learned process model to get a more efficient policy update.



## Model Learning Actor-Critic (2)

- Process model  $x' = \hat{f}(x, u)$  provides  $\frac{\partial \hat{f}}{\partial u}$ . This allows the actor update

$$\vartheta_{k+1} = \vartheta_k + \alpha_a \frac{\partial V}{\partial x} \frac{\partial \hat{f}}{\partial u} \frac{\partial \pi}{\partial \vartheta}$$

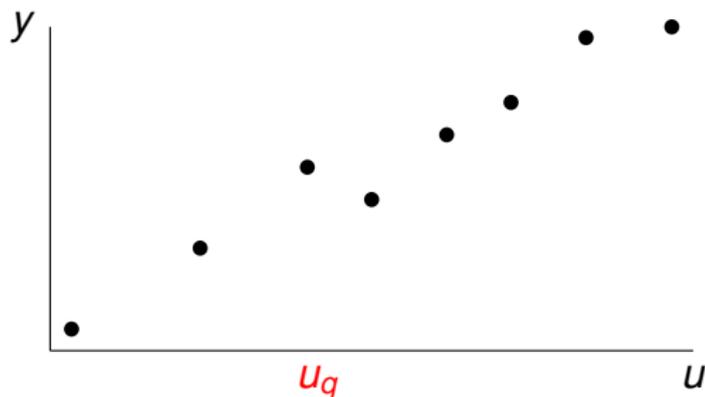
- Critic update is the original TD update

$$\theta_{k+1} = \theta_k + \alpha_c \delta_k \frac{\partial V}{\partial \theta}$$

- Process model update e.g. by fitting (supervised learning!).

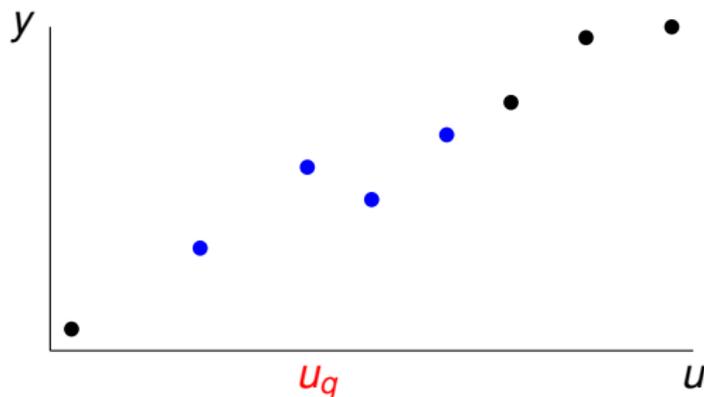
# Local Linear Regression

- LLR is function approximator for actor, critic and process model
- Memory-based: memory samples are input/output pairs.
- Approximation of output by regression on  **$k$  nearest neighbours**.
- Example ( $k = 4$ ):



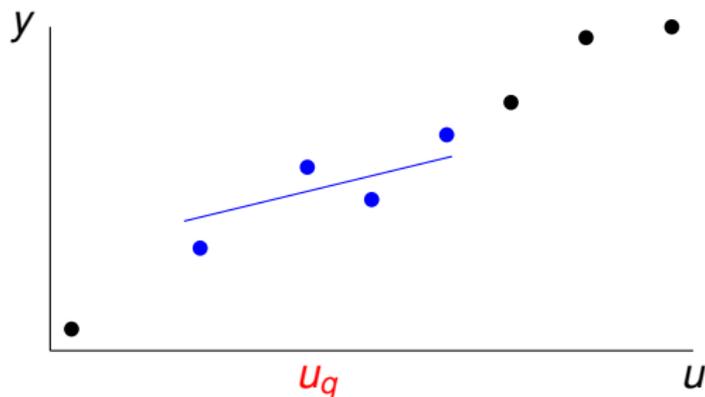
# Local Linear Regression

- LLR is function approximator for actor, critic and process model
- Memory-based: memory samples are input/output pairs.
- Approximation of output by regression on  **$k$  nearest neighbours**.
- Example ( $k = 4$ ):



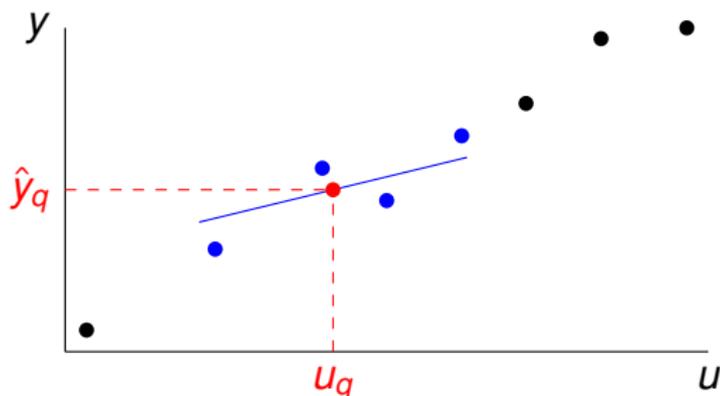
# Local Linear Regression

- LLR is function approximator for actor, critic and process model
- Memory-based: memory samples are input/output pairs.
- Approximation of output by regression on  **$k$  nearest neighbours**.
- Example ( $k = 4$ ):



# Local Linear Regression

- LLR is function approximator for actor, critic and process model
- Memory-based: memory samples are input/output pairs.
- Approximation of output by regression on  **$k$  nearest neighbours**.
- Example ( $k = 4$ ):



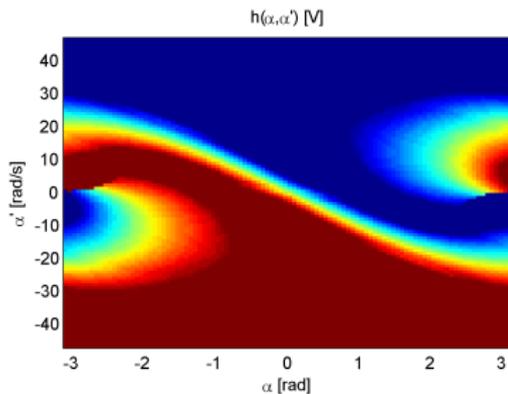
# Memories involved

Three LLR memories used:

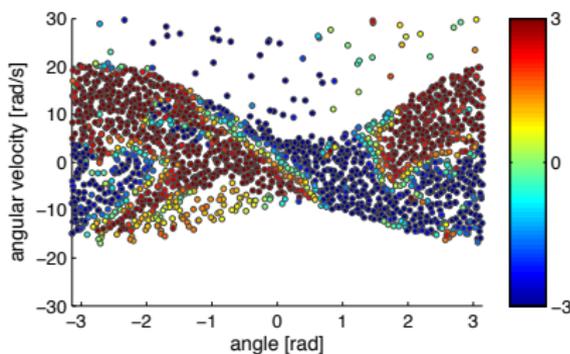
- Actor  $u = \pi(x)$ , with samples  $[ x \mid u ]$ .
- Critic  $\hat{V}(x)$ , with samples  $[ x \mid \hat{V} ]$ .
- Process model  $x' = \hat{f}(x, u)$ , with samples  $[ x \ u \mid x' ]$ , where  $x'$  is the next state given by the dynamics of the system.

# Inverted pendulum: Model Learning Actor-Critic

Left: Optimal policy

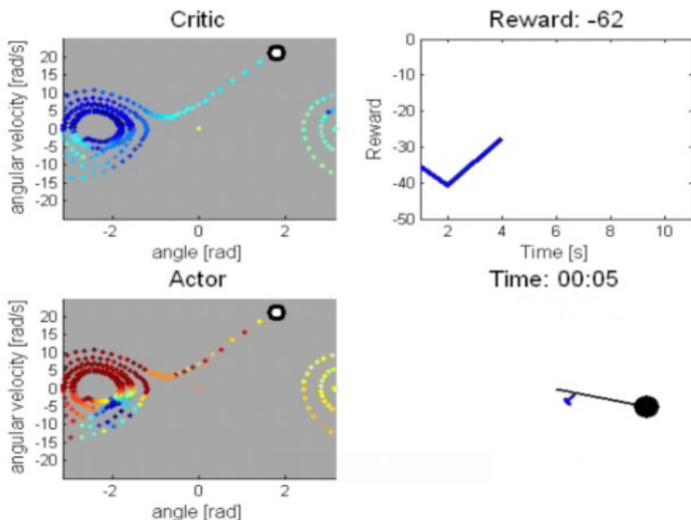


Right: Policy obtained with MLAC



# Model Learning Actor-Critic in action

One trial of the Model Learning Actor-Critic algorithm

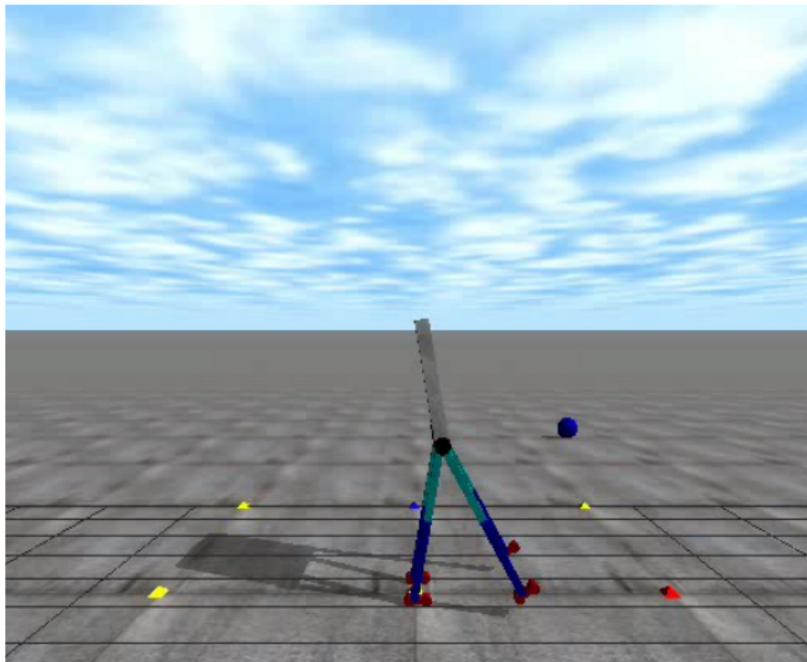




- 1 Introduction
- 2 Dealing with continuous spaces
- 3 Demo of walking robot**

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

The Q-learning algorithm from the previous lecture can be adapted for continuous states/actions too



## Take-home message

(Approximate) Reinforcement Learning =  
**Learn** how to **optimally** control  
complex systems, possibly from scratch