How to decide – Machine Learning with Python

Michael Röttger, Andreas W. Liehr

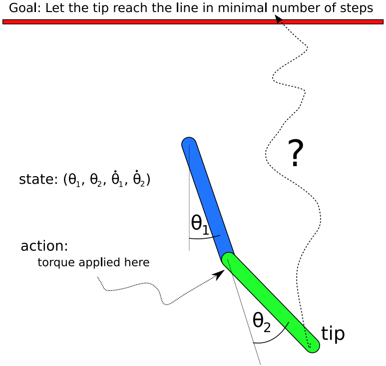
Materials Research Center, Freiburg, Germany
Service-group “Scientific Information processing”

2008-07-27

1. Reinforcement Learning (RL) by example
2. Results for examples
3. Debugging the learning process
4. Summary

Outline

Example control problem: acrobot



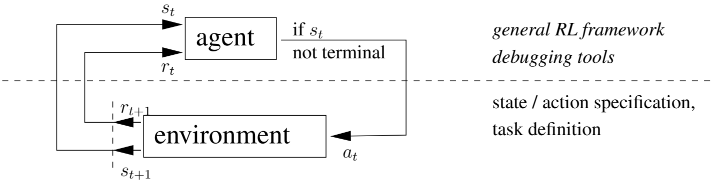
every time step: new torque, constant reward $r \equiv -1$

Maximize total reward $\longleftrightarrow$ minimize numer of time steps

model taken from [2]

RL's view of control problems

s_t	*state* of the environment at time t
a_t	agent's *action* at time t
r_{t+1}	*reward* for doing a_t



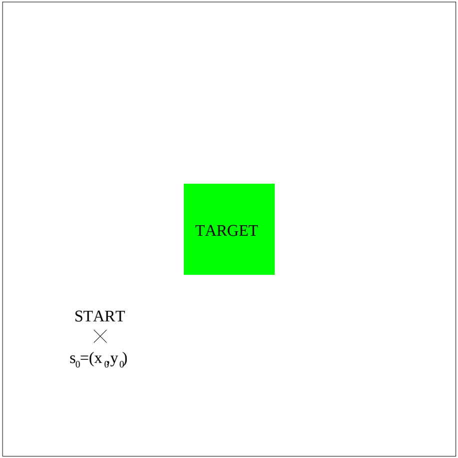
general RL framework
debugging tools

state / action specification,
task definition

Agent's challenge

Assumption: Modell for transition $s \xrightarrow{a} r', s'$ is *unknown*!
How to to **maximize the estimation of the total reward**?

Simple example: path finder



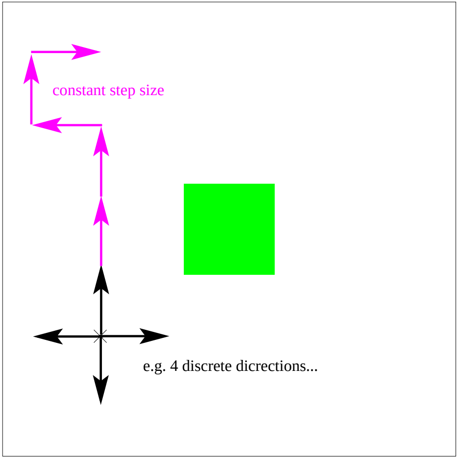
state s

position (x, y)
continuous
bounded

reward r

always -1 on every step

Simple example: path finder



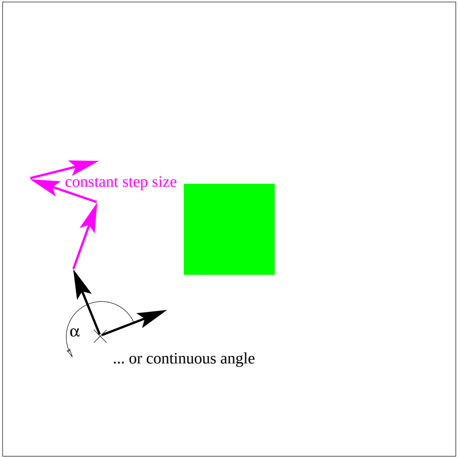
action *a* (*discrete*)

direction as angle α
e.g. 4 discrete values

$$0^\circ, 90^\circ, 180^\circ, 270^\circ$$

(or 8, 16,. . . values)

Simple example: path finder

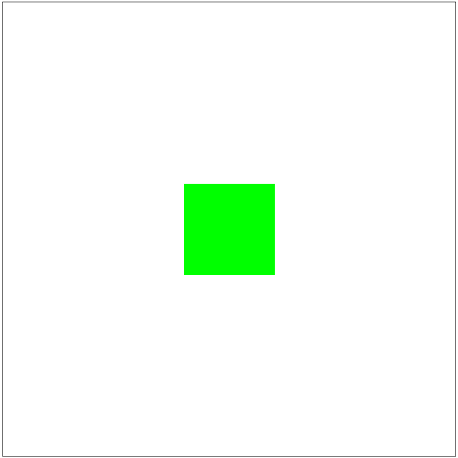


action *a* (*continuous*)

direction as angle α
with continuous values

$$\alpha \in [0^\circ, 360^\circ[$$

Example policy for path finder, continuous state



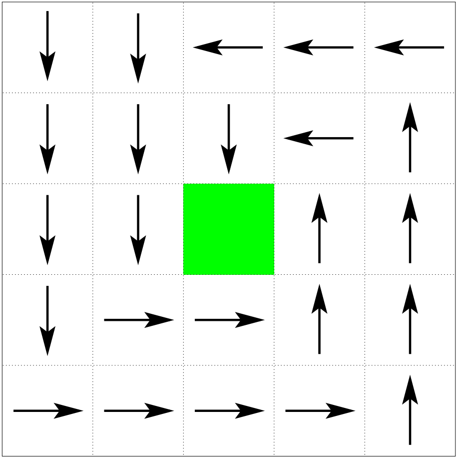
Mission

Maximize total reward!
→ Enter the target with minimal number of steps!
From anywhere!

Maybe not the most direct way .. what would be better?

Example policy for path finder, continuous state

"Spiral policy" $\pi^{\text{spiral}}(x, y)$



policy / strategy π

Mapping for all states:

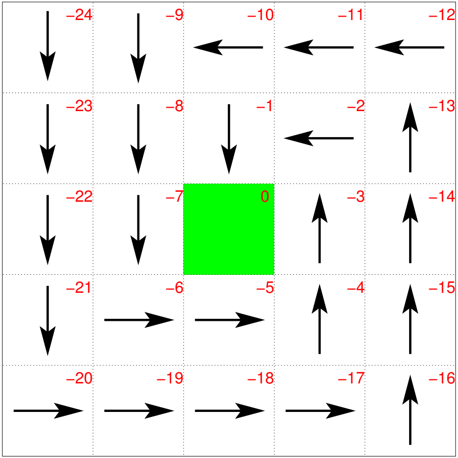
$$\text{state } s \quad \rightarrow \quad \text{action } a$$

Here: $s = (x, y)$ (position)

Maybe not the most direct way .. what would be better?

Example policy for path finder, continuous state

"Spiral policy" $\pi^{\text{spiral}}(x, y)$, $V^{\text{spiral}}(x, y)$,



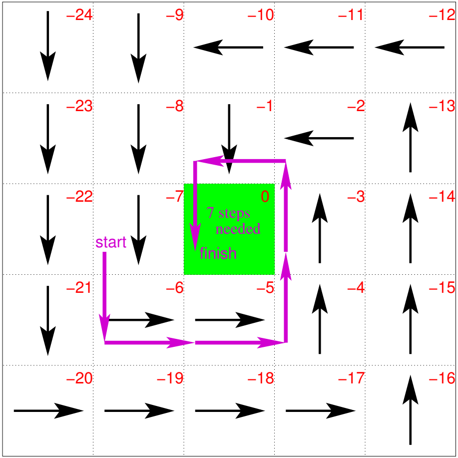
Maybe not the most direct way .. what would be better?

state value $V^\pi(s)$

What *total* reward one can expect during one episode when starting in state s and following policy π?

Example policy for path finder, continuous state

“Spiral policy” $\pi^{\text{spiral}}(x, y)$, $V^{\text{spiral}}(x, y)$,



Maybe not the most direct way .. what would be better?

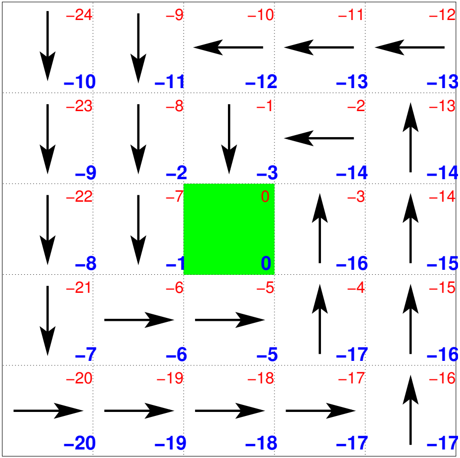
Example episode

$r \equiv -1$ and 7 steps

$$\Rightarrow \quad V(s_0) = -7$$

Example policy for path finder, continuous state

"Spiral policy" $\pi^{\text{spiral}}(x, y)$, $V^{\text{spiral}}(x, y)$, $Q^{\text{spiral}}(x, y, \rightarrow)$



action value $Q^{\pi}(s, a)$

What total reward one can expect during one episode when starting in state s, **doing** a and **then** following policy π?

Maybe not the most direct way .. what would be better?

Optimal policy and value functions

ordering relation

$$\pi \geq \pi' \;:\Leftrightarrow\; V^{\pi}(s) \geq V^{\pi'}(s)\ \forall s.$$

optimal policy π^* (*not* unique)

$$\pi^* \geq \pi \quad \forall\, \pi$$

optimal value functions (unique for all π^*)

$$Q^*(s,a) := Q^{\pi^*}(s,a)$$

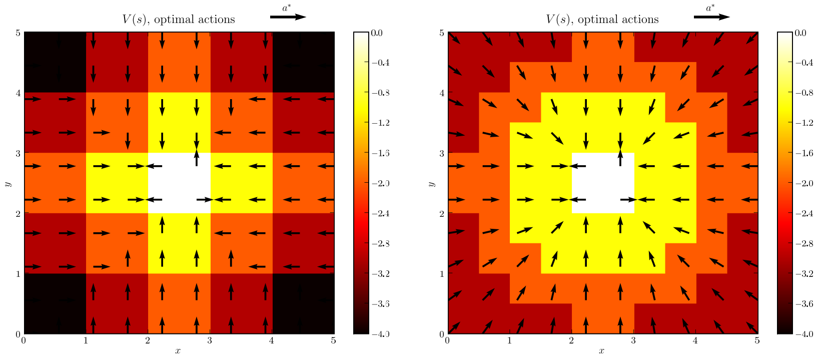
$$V^*(s) := V^{\pi^*}(s) = \max_a Q^*(s,a)$$

Q^* would be very useful, because

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

Optimal solution for pathfinder

4 discrete angles | continuous angles



optimal solution is known exactly
→ pathfinder useful for comparing different algorithms!

Task definition

Inherit from `rl.taskskel.Environment`

define system's dynamics and reward using property sets

```
.__init__(state_ps, action_ps, gamma, *args, **kwds)
.terminal(state=None)
.state_valid(state)          # optionally
.action_valid(state, action) # optionally
.next_state(state, action)
.reward(state, action, next_state)
```

optionally, if known, define optimal solution

```
.state_value(state)
.action_value(state, action)
.optimal_action(state)
```

Excerpt of task definition for path finder

```
class PathFinderEnvironment(rl.taskskel.Environment):
    ...
    def next_state(self, state, action):
        x,y = state
        angle = action[0]

        x += STEP_WIDTH*math.cos(angle)
        y += STEP_WIDTH*math.sin(angle)

        # crop state to make it valid
        if x < self._min_x:
           x = self._min_x
        elif x > self._max_x:
           x = self._max_x
        ...
        return (x,y)

    def reward(state, action, next_state):
        return -1
    ...
```

Preparing property sets

pathfinder's state

```
state_ps = rl.properties.PropertySet('state')
state_ps.add_continuous('x', minx, maxx)
state_ps.add_continuous('y', miny, maxy)
```

pathfinder's action

```
action_ps = rl.properties.PropertySet('action')
# discrete actions ..
angles = N.arange(0, twopi,
                  twopi / num_discrete_angles )
action_ps.add_discrete('angle', angles)

# .. or continuous actions
action_ps.add_continuous('angle', 0, twopi)
```

Purpose of property sets

Property sets are helpers for

- validation:
  `state_ps.valid((3,-5))` →`False`
- generation of valid random values:
  `state_ps.random()`
- generation of sample points:
  `state_ps.samples(nx,ny)`
- generalisation

Objectives

We want to have a simple RL framework in Python

- which can be easily applied to new control tasks
- to evaluate algorithms by comparision with known optimal solutions
- to find and test methods for searching for optimal policies π^* for continuous actions
- to compare solutions with discrete/continuous states/actions
- which allows separation of computation from analysis

Outline

How to find Q^*?

Challenge: Learn from rewards and by *trial and error*!

Bellman optimality equation for action values

$$Q^*(s,a) = E\left\{ r(s_t, a_t, s_{t+1}) + \gamma \max_{a'} Q^*(s_{t+1}, a') \,\middle|\, s_t = s, a_t = a \right\}$$

$\rightarrow$ foundation for many iterative methods with scheme

$$Q_0 \overset{\text{argmax}}{\rightarrow} \pi_0 \overset{\text{learning}}{\rightarrow} Q_1 \overset{\text{argmax}}{\rightarrow} \pi_1 \overset{\text{learning}}{\rightarrow} \dots \overset{\text{learning}}{\rightarrow} Q^* \leftrightarrow \pi^*$$

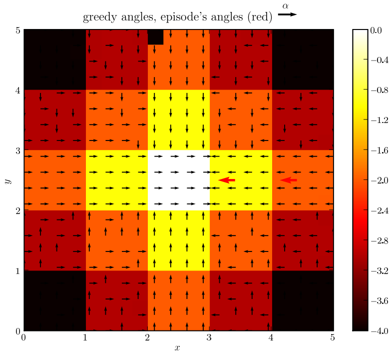
like

TD methods: Sarsa(λ), Q(λ) [3], ...

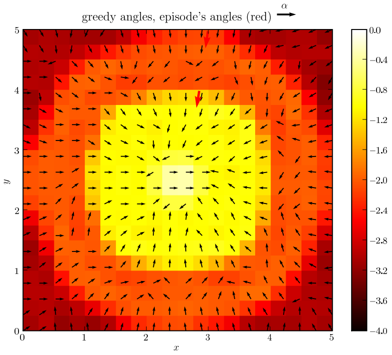
PI methods: LSPI [1], KLSPI [4],...

Policies found for pathfinder

4 discrete angles



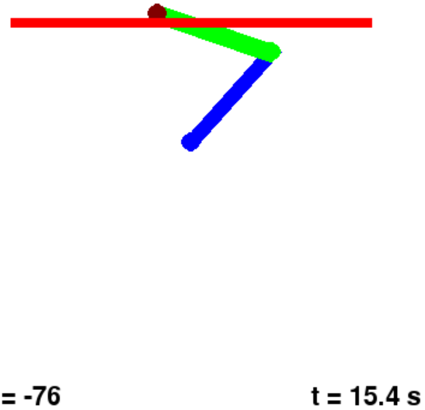
continuous angles



(used for continuous a: Sarsa(λ), linear approximation, RBF features for states/actions, ...)

Solution for acrobot

Discrete actions: $\tau \in [-1, 0, 1]$



(Sarsa(λ), linear approximation, tilings as state features, eligibility traces. . .)

Outline

Saving results in HDF5 files

Separation of calculation and analysis

Data → HDF 5 file using *PyTables*

- parameters, command line switches
- task description
- start/end time of execution
- *sampling states/actions:* $\tilde{s}_i,\ \tilde{a}_j \quad \forall i, j$
- *optimal solution, if available:*

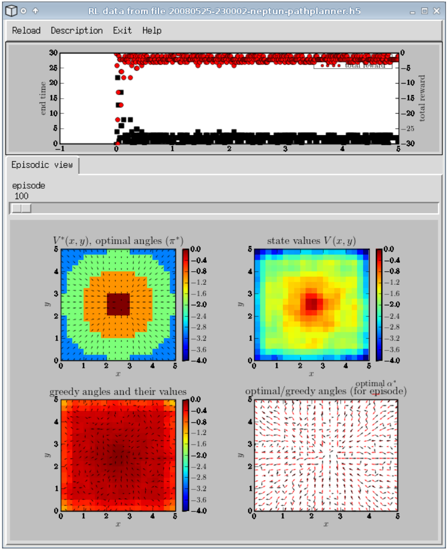
$$Q^*(\tilde{s}_i,\ \tilde{a}_j), V^*(\tilde{s}_i), \pi^*(\tilde{s}_i), Q^*(\tilde{s}_i,\ \pi^*(\tilde{s}_i))$$

Two tables for each saved episode:

for last time T: $Q(\tilde{s}_i,\ \tilde{a}_j), V(\tilde{s}_i), \pi(\tilde{s}_i), Q(\tilde{s}_i,\ \pi(\tilde{s}_i)), \ldots$

for each time step: state, action, reward, ...

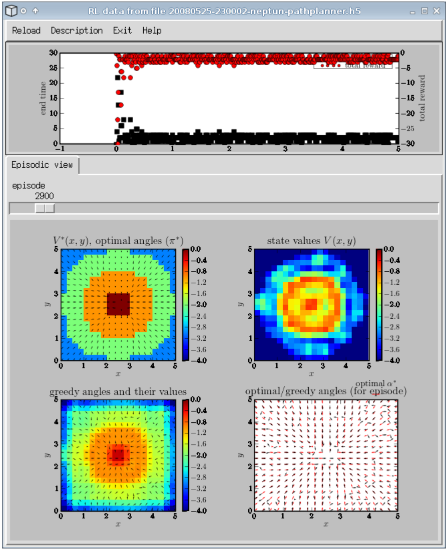
Review of resulting episodes



`view.py` provides a GUI which

- maps HDF tables/arrays to plots
- plot descriptions read from *YAML* file
- gives an overview about all episodes (total reward, end time)
- allows browsing through episodes by selecting rows in HDF tables
- uses *Tkinter/Tix* with an embedded *Matplotlib* figures

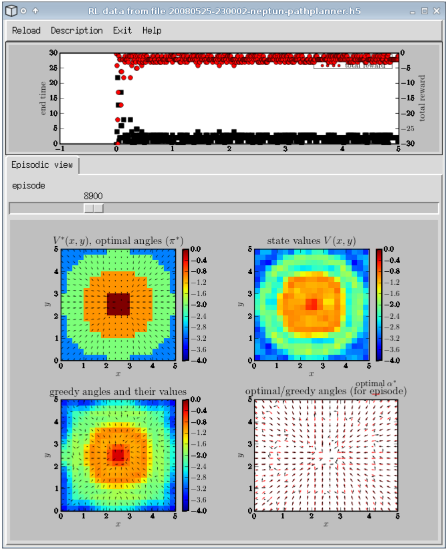
Review of resulting episodes



`view.py` provides a GUI which

- maps HDF tables/arrays to plots
- plot descriptions read from *YAML* file
- gives an overview about all episodes (total reward, end time)
- allows browsing through episodes by selecting rows in HDF tables
- uses *Tkinter/Tix* with an embedded *Matplotlib* figures

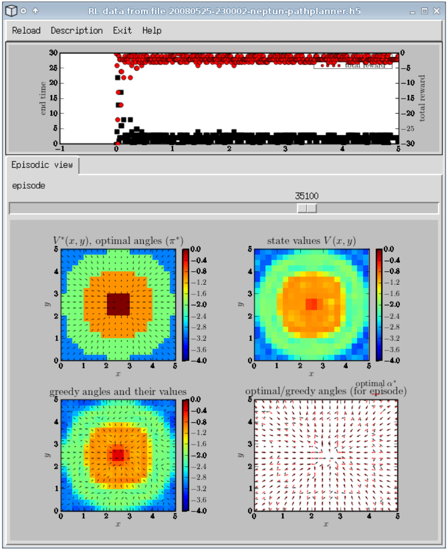
Review of resulting episodes



`view.py` provides a GUI which

- maps HDF tables/arrays to plots
- plot descriptions read from *YAML* file
- gives an overview about all episodes (total reward, end time)
- allows browsing through episodes by selecting rows in HDF tables
- uses *Tkinter/Tix* with an embedded *Matplotlib* figures

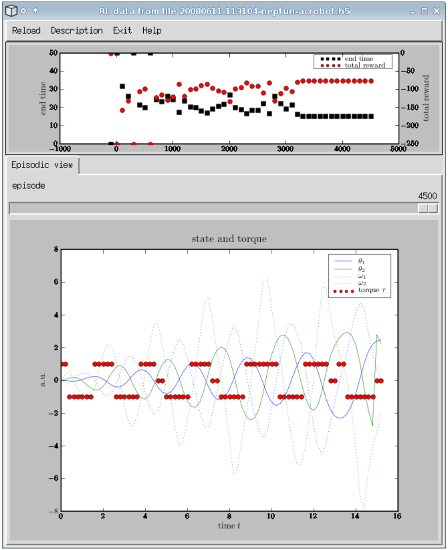
Review of resulting episodes



`view.py` provides a GUI which

- maps HDF tables/arrays to plots
- plot descriptions read from *YAML* file
- gives an overview about all episodes (total reward, end time)
- allows browsing through episodes by selecting rows in HDF tables
- uses *Tkinter/Tix* with an embedded *Matplotlib* figures

Mapping HDF data to plots with YAML



Example for plot description

```
-
type:  line plot
title: state and torque
xdata:  interactions.time{episode}
xlabel: time $t$
ydata:
  -
    locator: interactions.state{episode}[:,0]
    label: $\theta_1$
    style: 'b-'
  -
    locator: interactions.state{episode}[:,1]
    label: $\theta_2$
    style: 'g-'
  -
    locator: interactions.state{episode}[:,2]
    label: $\omega_1$
    style: 'b:'
  -
    locator: interactions.state{episode}[:,3]
    label: $\omega_2$
    style: 'g:'
  -
    locator: interactions.action{episode}[:,0]
    label: torque $\tau$
    style: 'ro'

ylabel: a.u.
```

Comparing solution methods

There are many combinations of different settings:

algorithms	Sarsa(λ), Q(λ), KLSPI, ...
approximations	linear, ... (?)
feature models	binary, radial based, tile coding, ALD, ...
action models	discrete, continuous (,mixed?)
control tasks	pathfinder, mountain car, acrobot, dispatcher, ...

Some combinations

- may be better/faster/simpler than others
- don't even work

How to compare them without losing overview?

Preconditions for each combination of settings

For continuous states / actions, an approximation ist needed

- which should be able to represent the optimal solution “well enough”, e.g. for given tolerance δ

$$Q \approx Q^* \;:\Leftrightarrow\; |Q(\tilde{s}_i,\, \tilde{a}_j) - Q^*(\tilde{s}_i,\, \tilde{a}_j)| < \delta \;\forall i, j$$

- this should be stable when continuing to learn, e.g.

$$Q_0 := Q^*;\; Q_0 \to Q_1 \to \ldots \to Q_{10} \overset{!}{\approx} Q^*$$

- the solution should be found without prior knowledge of Q^*

$$Q_0 := \text{arbitrary};\; Q_0 \to Q_1 \to \ldots \to Q_n \overset{!}{\approx} Q^*$$

- ...

Objective: Check these for many combinations, get a report

Test generation

Test conditions are coded in a YAML file using the CLI, e.g.

```
label: check-value-functions-convergence
systems:
        - " -T 20 -N 20001 -L sarsa-lambda --save-optimal"
variants:
      -
        - '-F binary'
        - '-F radial_based'
        - '-F tilings'
      -
        - '-G binary'
        - '-G radial_based'
tests:
        - "Q->Qopt,tolerance:0.1"
        - "V->Vopt,tolerance:0.1"
```

$\rightarrow (3 * 2) * 2 = 12$ tests are built from this

Performing tests, generate report

Same interface for different control tasks, here e.g. for task defined in module *pathplanner*:

perform tests: Test descriptions → doctests → test results

```
python systest.py -o results.yaml pathplanner tests.yaml
```

format results: results → LaTeX

```
python format.py results.yaml > results.tex
```

Performing tests, generate report

Same interface for different control tasks, here e.g. for task defined in module *pathplanner*:

perform tests: Test descriptions → doctests → test results

```
python systest.py -o results.yaml pathplanner tests.yaml
```

format results: results → LaTeX

```
python format.py results.yaml > results.tex
```

Performing tests, generate report

Report of test results

date: `2008-07-11`, time: `13:37:07` host: `saturn`

1 Tests labeled `test-Qopt-remains-Qopt`

Summary of tests

systems	test condition	result	remarks
S `:-N 11 -e 1 --set-optimal-Q --save-optimal -L sarsa-lambda -l 0.0 --set-state-resolutions=10,10 --nums-intervals-state-feature=10,10 -F binary -G binary --Q-argmax-flavour=discrete`	$Q \overset{0.01}{\approx} Q^*$	Passed	• generated output file '20080711-133636-saturn-pathplanner.h5'
S `:-N 11 -e 1 --set-optimal-Q --save-optimal -L sarsa-lambda -l 0.0 --set-state-resolutions=10,10 --nums-intervals-state-feature=10,10 -F radial_based -G binary --Q-argmax-flavour=discrete`	$Q \overset{0.01}{\approx} Q^*$	Passed	• generated output file '20080711-133644-saturn-pathplanner.h5'

2 Tests labeled `Q-can-be-set-to-Qopt`

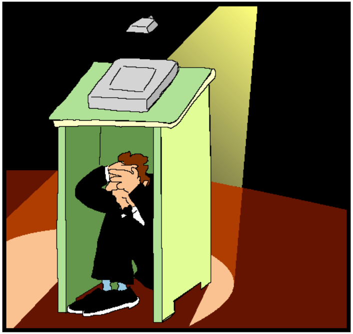
Outline

Summary

- overview of Reinforcement Learning (RL)
- we're developing a lightweight framework
  - allowing an easy definition of RL tasks
  - in order to incorperate algorithms for discrete and continuous actions
  - to test different combinations of algorithms, methods for generalisation and global optimization, discrete/continuous states/actions
- so far we're using *Matplotlib, Tkinter/Tix, Pyrex, Numpy, scipy.optimize, scipy.integrate, scipy.linalg, syck (YAML), PyTables (HDF), Pygame*

Thank you for working on these packages!

Thank you for your attention!



Questions?

bibliography I

[1] Michail G. Lagoudakis, Ronald Parr, and Michael L. Littman. Least-squares methods in reinforcement learning for control. In *Methods and Applications of Artificial Intelligence : Second Hellenic Conference on AI, SETN 2002. Thessaloniki, Greece, April 11-12, 2002. Proceedings*, pages 752–752, 2002.

[2] Richard S. Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. In David S. Touretzky, Michael C. Mozer, and Michael E. Hasselmo, editors, *Advances in Neural Information Processing Systems*, volume 8, pages 1038–1044. The MIT Press, 1996.

[3] C. J. C. H. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8(3-4):279–292, May 1992.

[4] Xin Xu, Dewen Hu, and Xicheng Lu. Kernel-based least squares policy iteration for reinforcement learning. *Neural Networks, IEEE Transactions on*, 18(4):973–992, July 2007.