MACMACHINE LEARNING

section 8

Reinforcement Learning tutorial

Contact Information

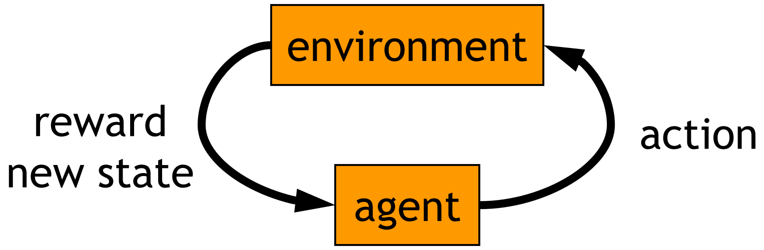
Instructor:	Qi Hao
E-mail:	hao.q@sustc.edu.cn
Office:	Nanshan iPark A7 Room 906
Office Hours:	M 2:00-4:00pm

Available other times by appointment or the open door policy

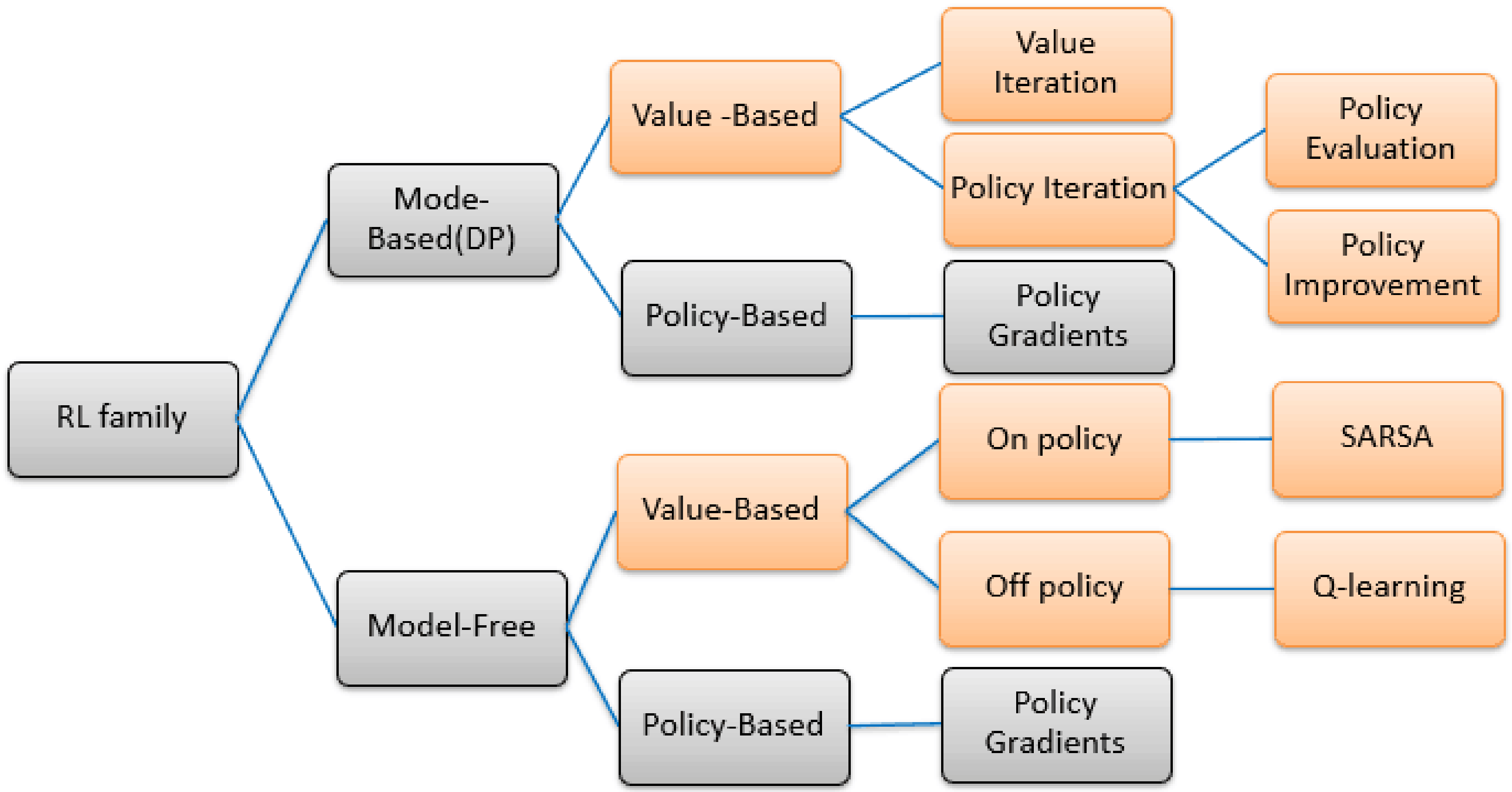
Office Phone:	(0755) 8801-8537
QQ:	463715202 机器学习2018
Web:	*http://hqlab.sustc.science/teaching/*

Previous Lectures

- Supervised learning
  - classification, regression
- Unsupervised learning
  - clustering
- Reinforcement learning
  - more general than supervised/unsupervised learning
  - learn from interaction w/ environment to achieve a goal



Structure

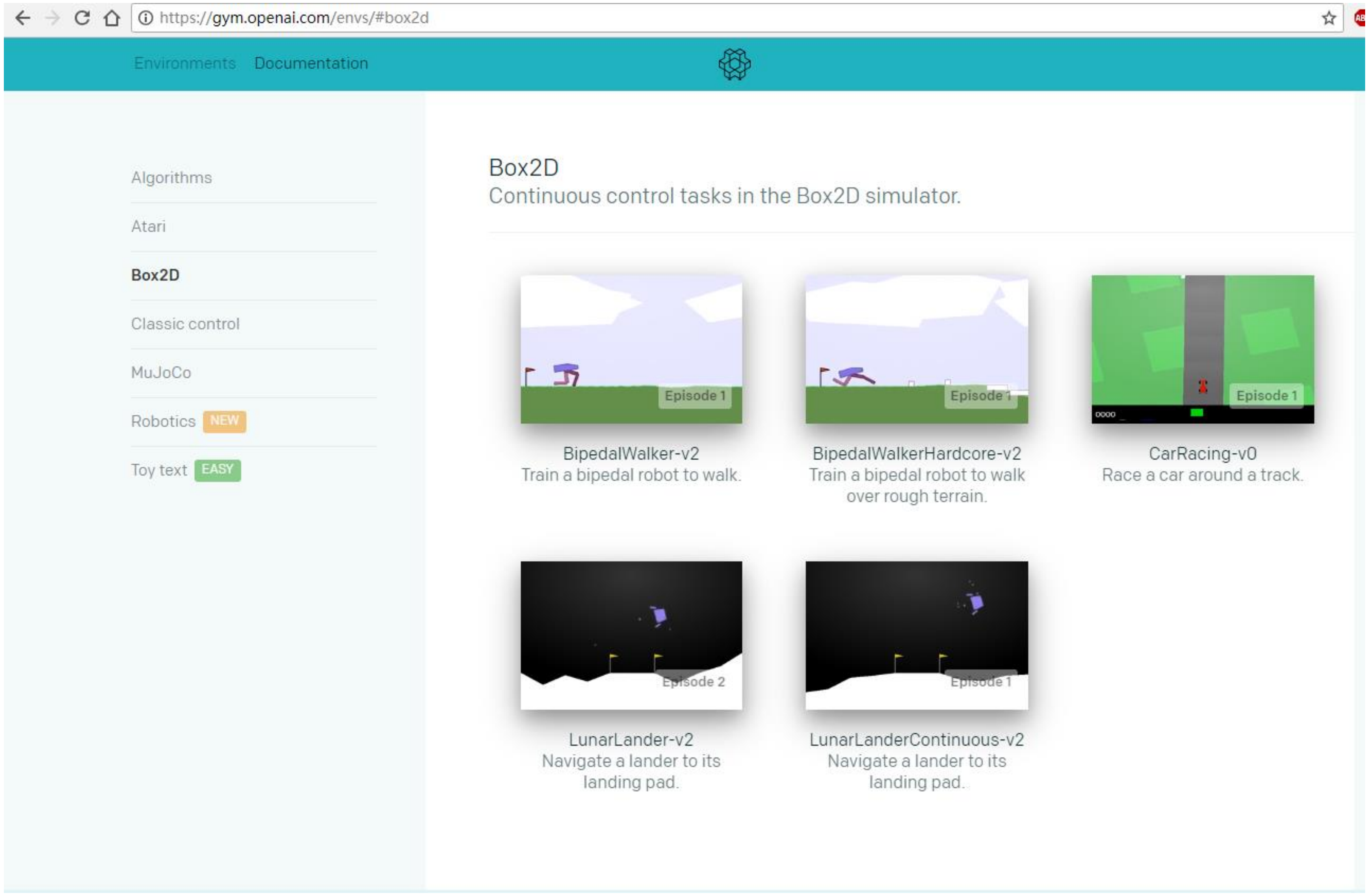

RL family
Mode-Based(DP)
Model-Free
Value -Based
Policy-Based
Value Iteration
Policy Iteration
Policy Evaluation
Policy Improvement
Policy Gradients
Value-Based
Policy-Based
On policy
Off policy
SARSA
Q-learning
Policy Gradients

Today

- examples
- defining an RL problem
  - Markov Decision Processes
- solving an RL problem
  - Dynamic Programming
  - Temporal-Difference learning

Lab Evironment: OpenAI Gym

https://gym.openai.com/



Box2D
Continuous control tasks in the Box2D simulator.

BipedalWalker-v2
Train a bipedal robot to walk.

BipedalWalkerHardcore-v2
Train a bipedal robot to walk over rough terrain.

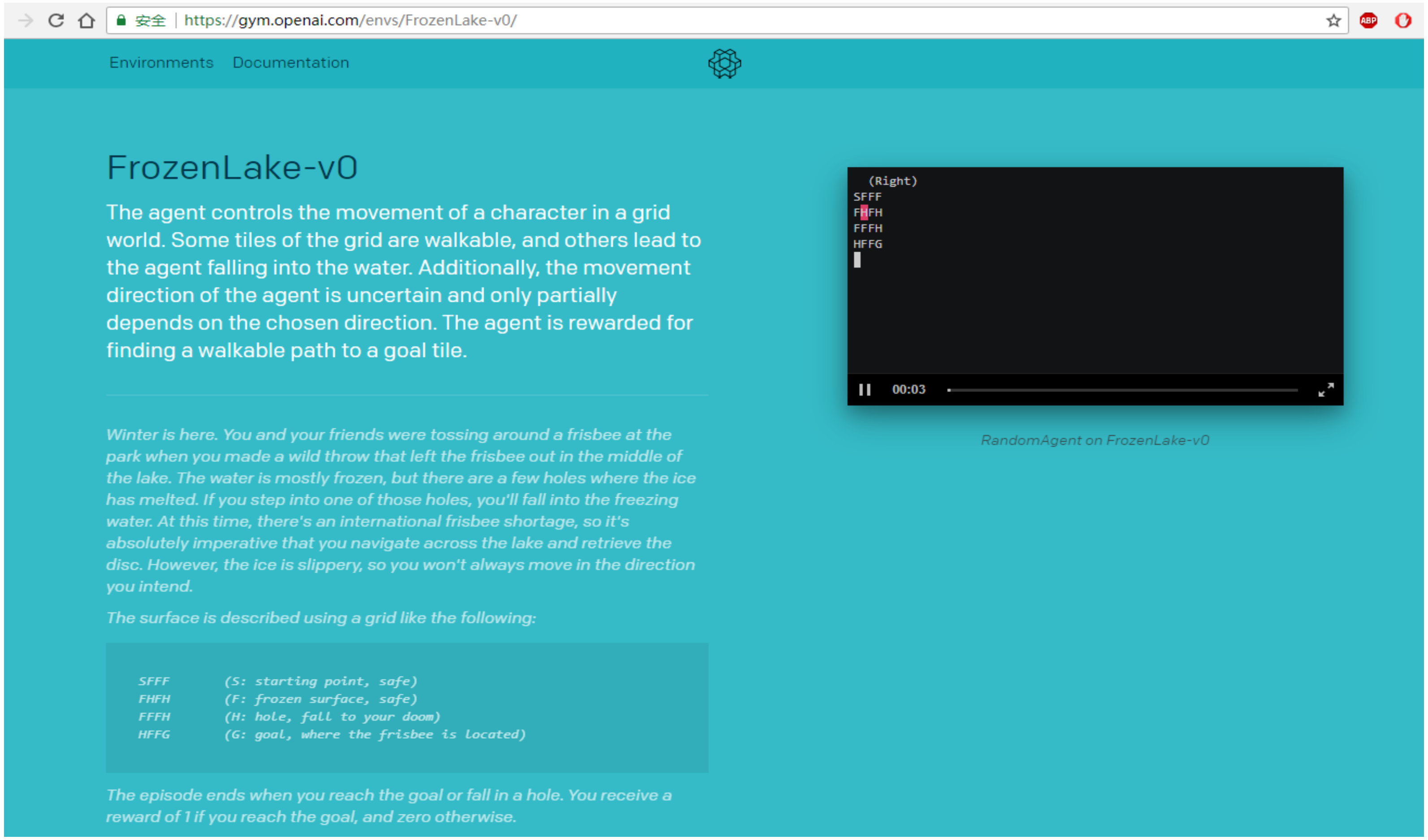
CarRacing-v0
Race a car around a track.

LunarLander-v2
Navigate a lander to its landing pad.

LunarLanderContinuous-v2
Navigate a lander to its landing pad.

OpenAI Gym

FrozenLake-v0 is a simple toy-text environment for you to get start.



FrozenLake-v0

The agent controls the movement of a character in a grid world. Some tiles of the grid are walkable, and others lead to the agent falling into the water. Additionally, the movement direction of the agent is uncertain and only partially depends on the chosen direction. The agent is rewarded for finding a walkable path to a goal tile.

Winter is here. You and your friends were tossing around a frisbee at the park when you made a wild throw that left the frisbee out in the middle of the lake. The water is mostly frozen, but there are a few holes where the ice has melted. If you step into one of those holes, you'll fall into the freezing water. At this time, there's an international frisbee shortage, so it's absolutely imperative that you navigate across the lake and retrieve the disc. However, the ice is slippery, so you won't always move in the direction you intend.

The surface is described using a grid like the following:

```
SFFF       (S: starting point, safe)
FHFH       (F: frozen surface, safe)
FFFH       (H: hole, fall to your doom)
HFFG       (G: goal, where the frisbee is located)
```

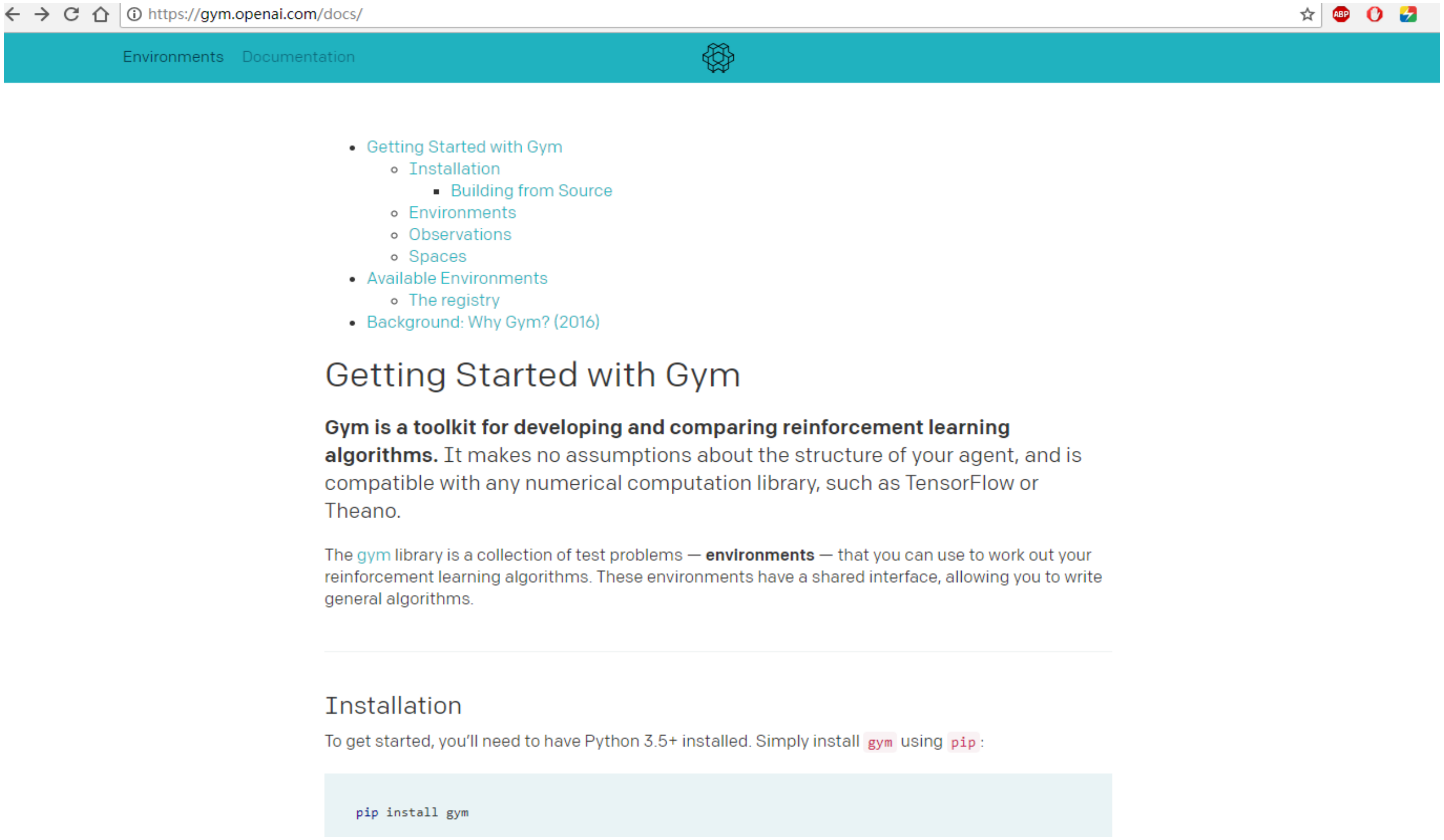
The episode ends when you reach the goal or fall in a hole. You receive a reward of 1 if you reach the goal, and zero otherwise.

```
  (Right)
SFFF
FHFH
FFFH
HFFG
```

RandomAgent on FrozenLake-v0

OpenAI Gym

It is easy for you to install OpenAI Gym toolkit. Just Follow the document.
https://gym.openai.com/docs/



- Getting Started with Gym
  - Installation
    - Building from Source
  - Environments
  - Observations
  - Spaces
- Available Environments
  - The registry
- Background: Why Gym? (2016)

Getting Started with Gym

Gym is a toolkit for developing and comparing reinforcement learning algorithms. It makes no assumptions about the structure of your agent, and is compatible with any numerical computation library, such as TensorFlow or Theano.

The gym library is a collection of test problems — **environments** — that you can use to work out your reinforcement learning algorithms. These environments have a shared interface, allowing you to write general algorithms.

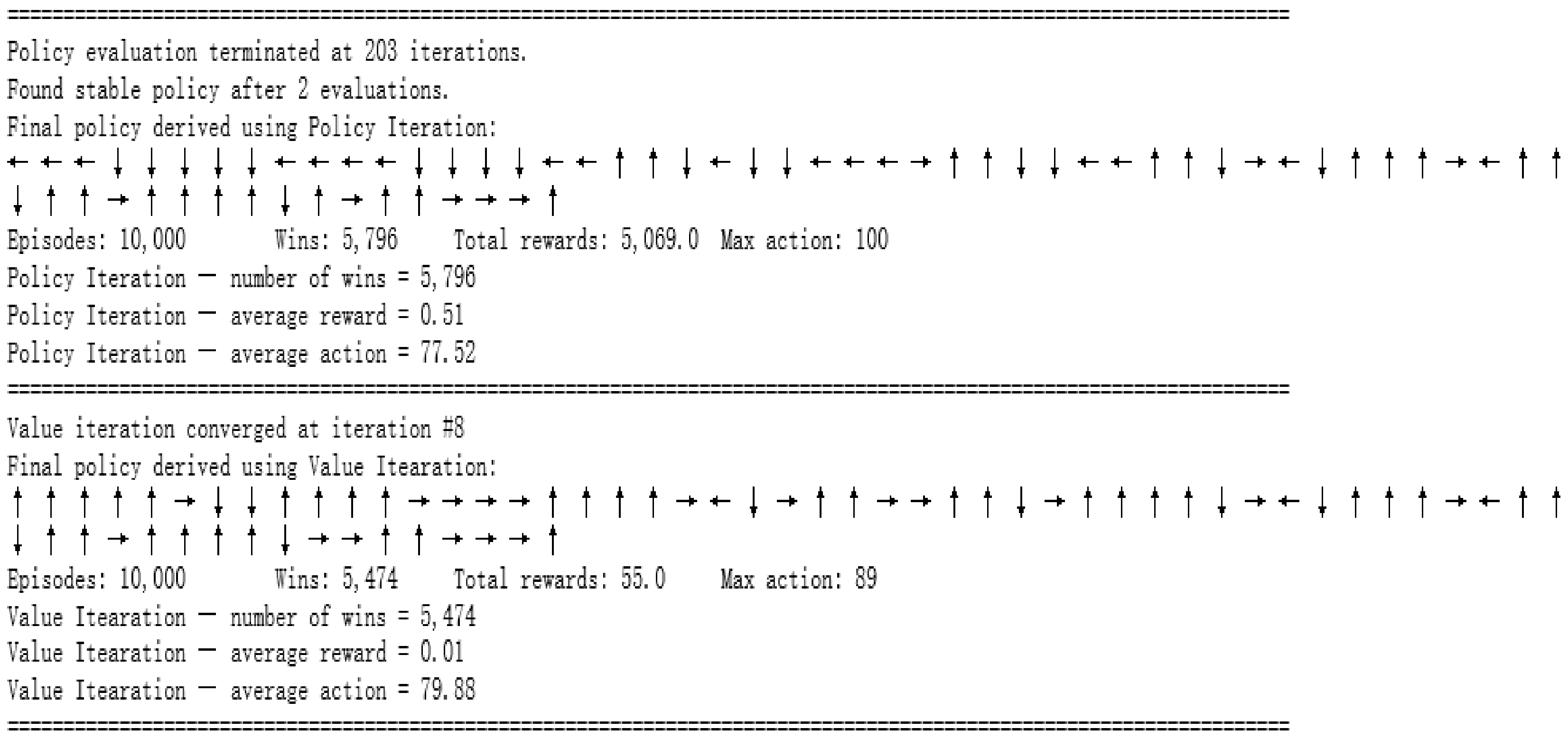
Installation

To get started, you'll need to have Python 3.5+ installed. Simply install `gym` using `pip`:

```
pip install gym
```

Lab Output sample

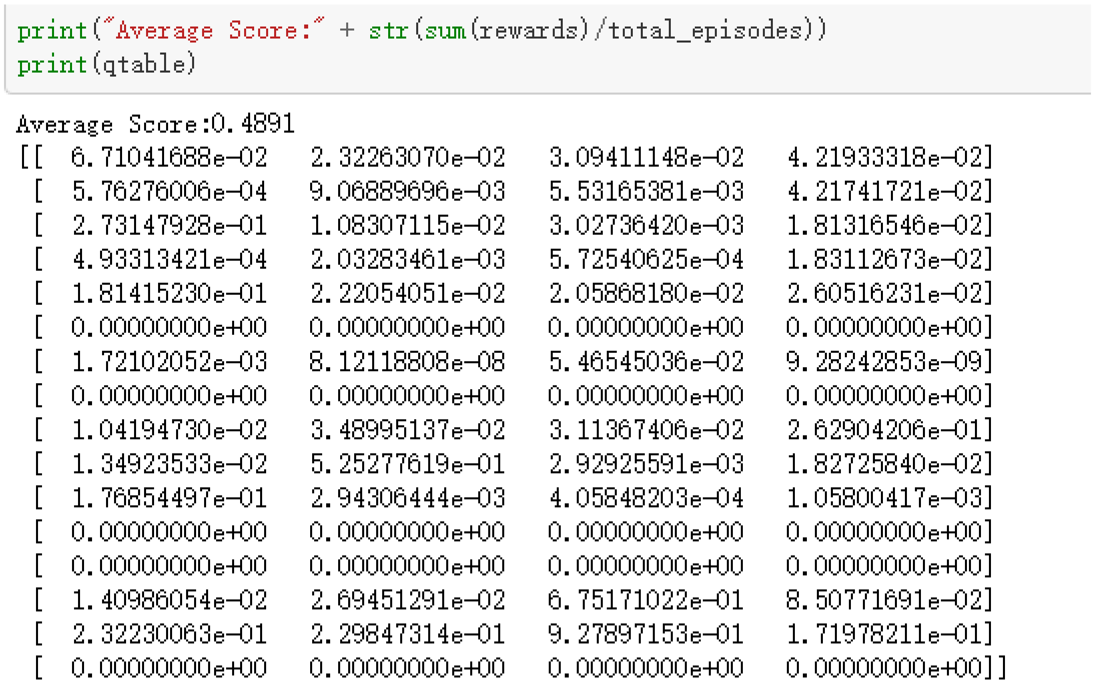
Policy Iteration and Value Iteration



```
==========================================================================================================
Policy evaluation terminated at 203 iterations.
Found stable policy after 2 evaluations.
Final policy derived using Policy Iteration:
← ← ← ↓ ↓ ↓ ↓ ↓ ← ← ← ← ↓ ↓ ↓ ↓ ← ← ↑ ↑ ↓ ← ↓ ↓ ← ← ← → ↑ ↑ ↓ ↓ ← ← ↑ ↑ ↓ → ← ↓ ↑ ↑ ↑ → ← ↑ ↑
↓ ↑ ↑ → ↑ ↑ ↑ ↑ ↓ ↑ → ↑ ↑ → → → ↑
Episodes: 10,000       Wins: 5,796     Total rewards: 5,069.0  Max action: 100
Policy Iteration — number of wins = 5,796
Policy Iteration — average reward = 0.51
Policy Iteration — average action = 77.52
==========================================================================================================
Value iteration converged at iteration #8
Final policy derived using Value Itearation:
↑ ↑ ↑ ↑ ↑ → ↓ ↓ ↑ ↑ ↑ ↑ → → → → ↑ ↑ ↑ ↑ → ← ↓ → ↑ ↑ → → ↑ ↑ ↓ → ↑ ↑ ↑ ↑ ↓ → ← ↓ ↑ ↑ ↑ → ← ↑ ↑
↓ ↑ ↑ → ↑ ↑ ↑ ↑ ↓ → → ↑ ↑ → → → ↑
Episodes: 10,000       Wins: 5,474     Total rewards: 55.0     Max action: 89
Value Itearation — number of wins = 5,474
Value Itearation — average reward = 0.01
Value Itearation — average action = 79.88
==========================================================================================================
```

Lab Output sample

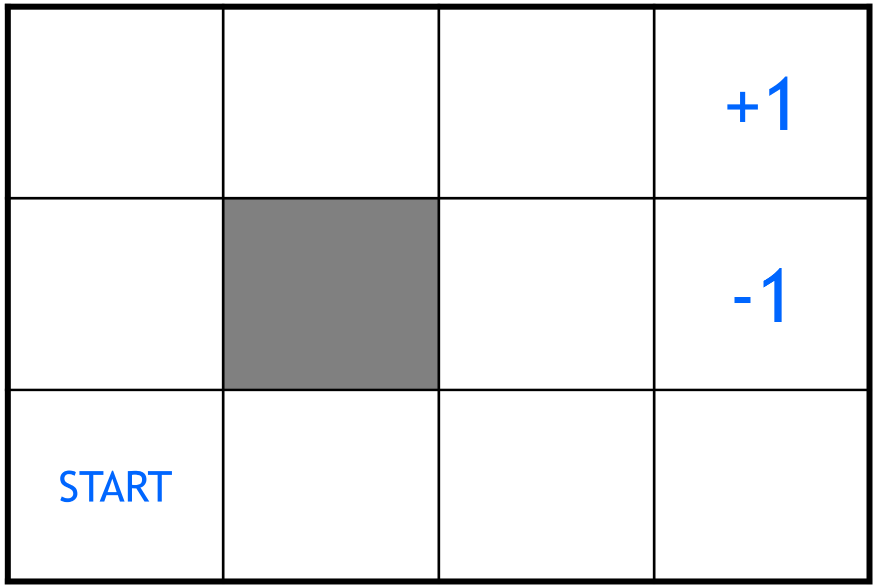
Q-Learning



```python
print("Average Score:" + str(sum(rewards)/total_episodes))
print(qtable)
```

```
Average Score:0.4891
[[ 6.71041688e-02  2.32263070e-02  3.09411148e-02  4.21933318e-02]
 [ 5.76276006e-04  9.06889696e-03  5.53165381e-03  4.21741721e-02]
 [ 2.73147928e-01  1.08307115e-02  3.02736420e-03  1.81316546e-02]
 [ 4.93313421e-04  2.03283461e-03  5.72540625e-04  1.83112673e-02]
 [ 1.81415230e-01  2.22054051e-02  2.05868180e-02  2.60516231e-02]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 1.72102052e-03  8.12118808e-08  5.46545036e-02  9.28242853e-09]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 1.04194730e-02  3.48995137e-02  3.11367406e-02  2.62904206e-01]
 [ 1.34923533e-02  5.25277619e-01  2.92925591e-03  1.82725840e-02]
 [ 1.76854497e-01  2.94306444e-03  4.05848203e-04  1.05800417e-03]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 1.40986054e-02  2.69451291e-02  6.75171022e-01  8.50771691e-02]
 [ 2.32230063e-01  2.29847314e-01  9.27897153e-01  1.71978211e-01]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  0.00000000e+00]]
```

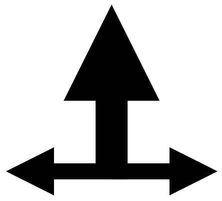
Robot in a room



actions: UP, DOWN, LEFT, RIGHT

UP

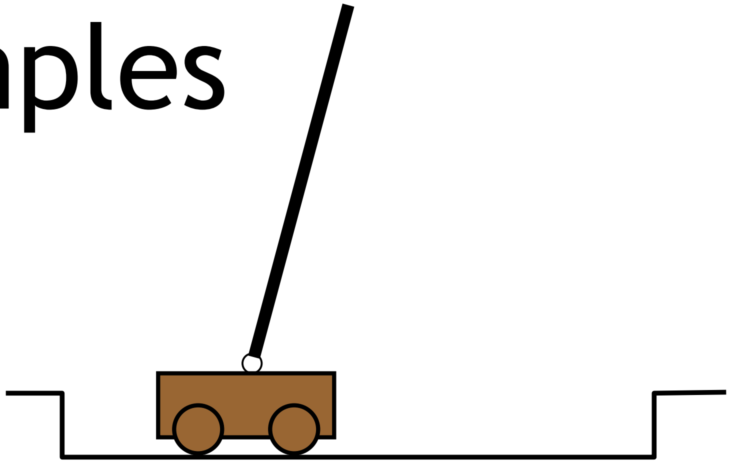
80%	move UP
10%	move LEFT
10%	move RIGHT



- reward +1 at [4,3], -1 at [4,2]
- reward -0.04 for each other state

- what's the strategy to achieve max reward?
- what if the actions were deterministic?

Other examples



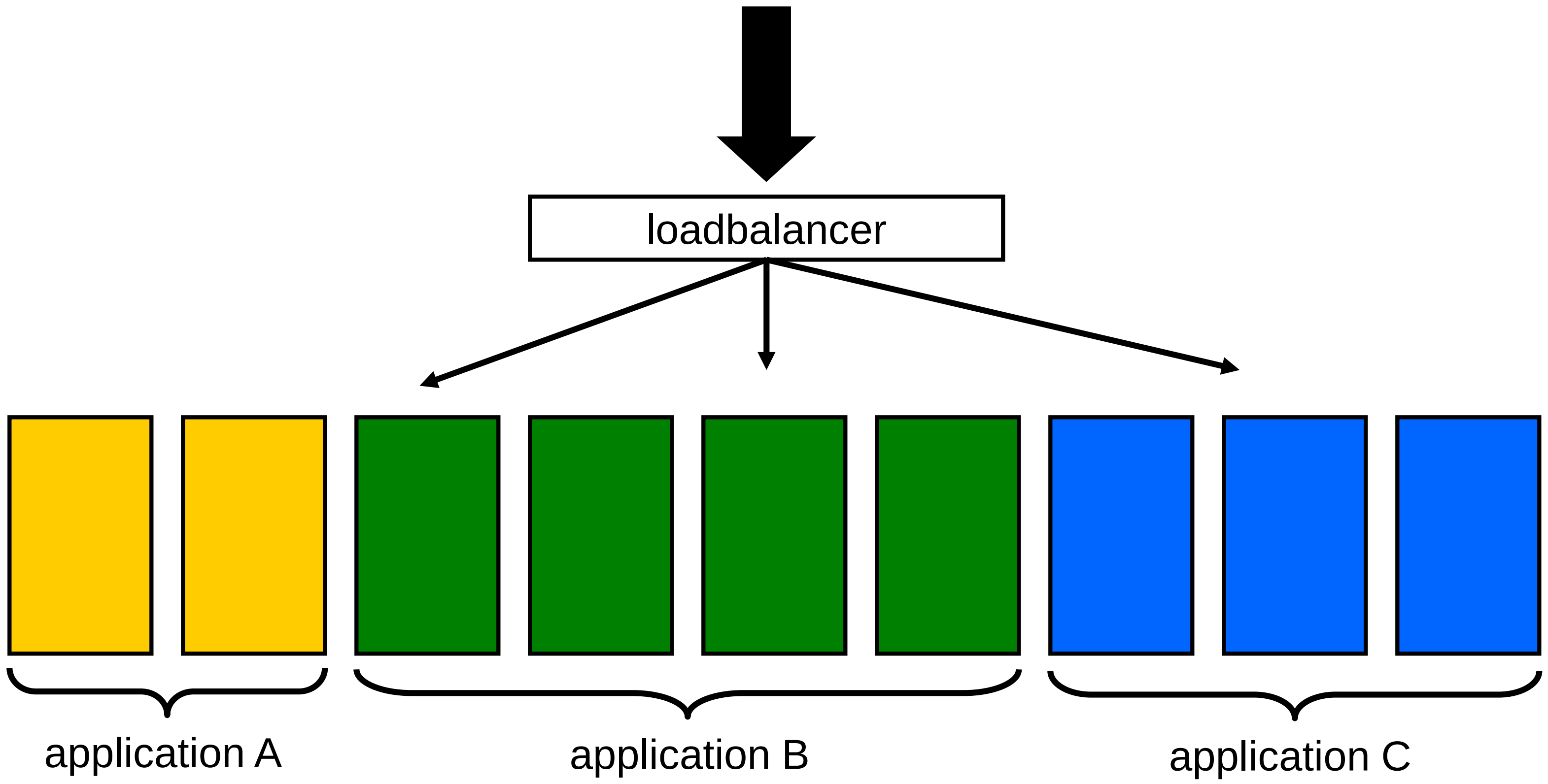
- pole-balancing
- TD-Gammon [Gerry Tesauro]
- helicopter [Andrew Ng]

- no teacher who would say "good" or "bad"
  - is reward "10" good or bad?
  - rewards could be delayed

- similar to control theory
  - more general, fewer constraints

- explore the environment and learn from experience
  - not just blind search, try to be smart about it

Resource allocation in datacenters



- A Hybrid Reinforcement Learning Approach to Autonomic Resource Allocation
  - Tesauro, Jong, Das, Bennani (IBM)
  - ICAC 2006

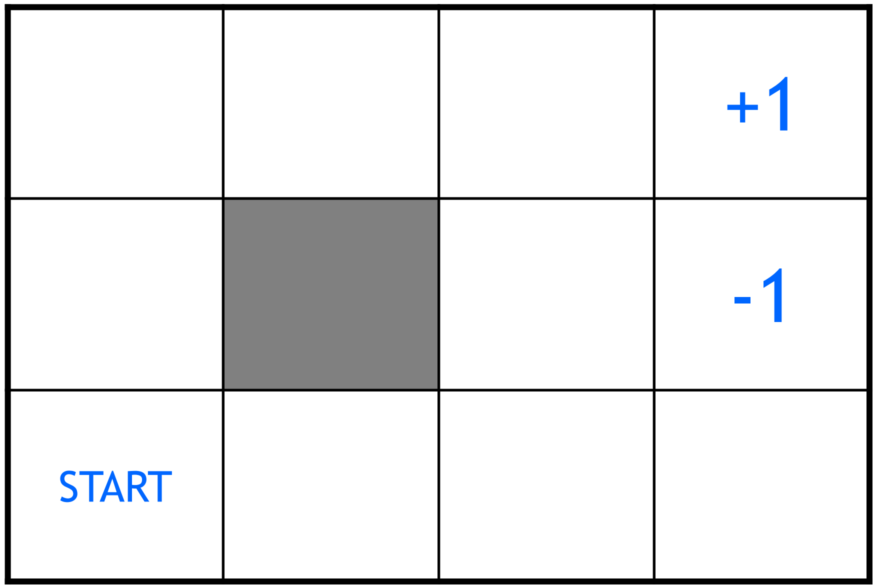
Outline

- examples

- defining an RL problem
  - Markov Decision Processes

- solving an RL problem
  - Dynamic Programming
  - Temporal-Difference learning

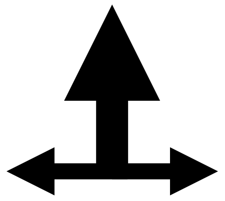
Robot in a room



actions: UP, DOWN, LEFT, RIGHT

UP

80%	move UP
10%	move LEFT
10%	move RIGHT

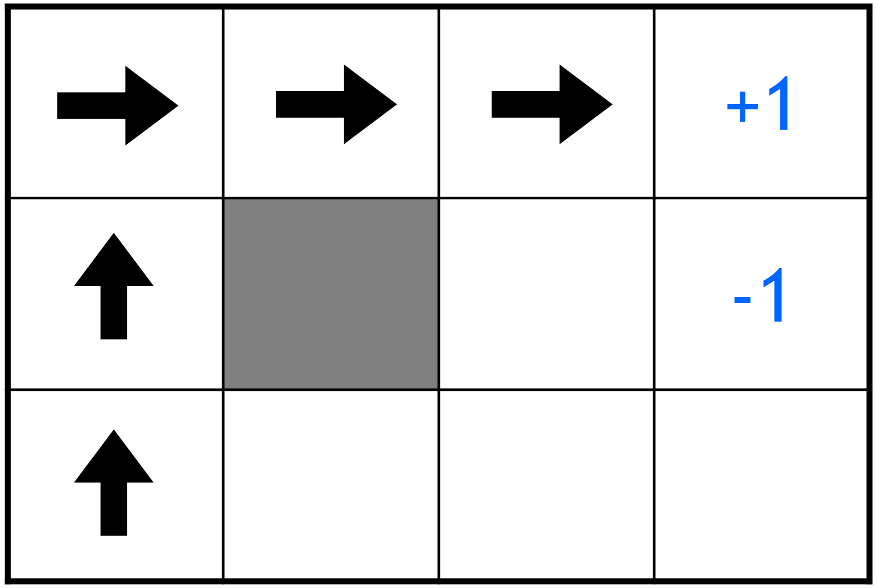


reward +1 at [4,3], -1 at [4,2]
reward -0.04 for each other state

- states
- actions
- rewards

- what is the solution?

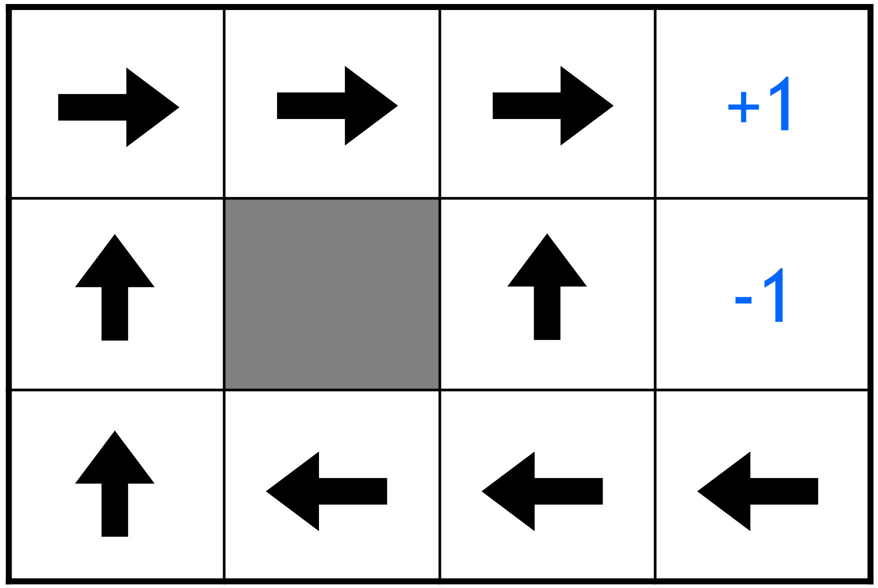
Is this a solution?



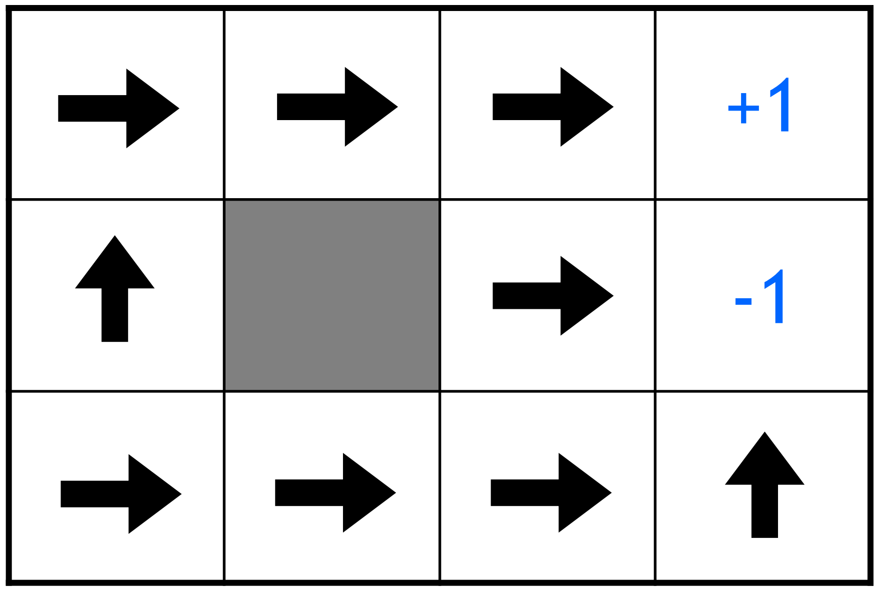
- only if actions deterministic
  - not in this case (actions are stochastic)

- solution/policy
  - mapping from each state to an action

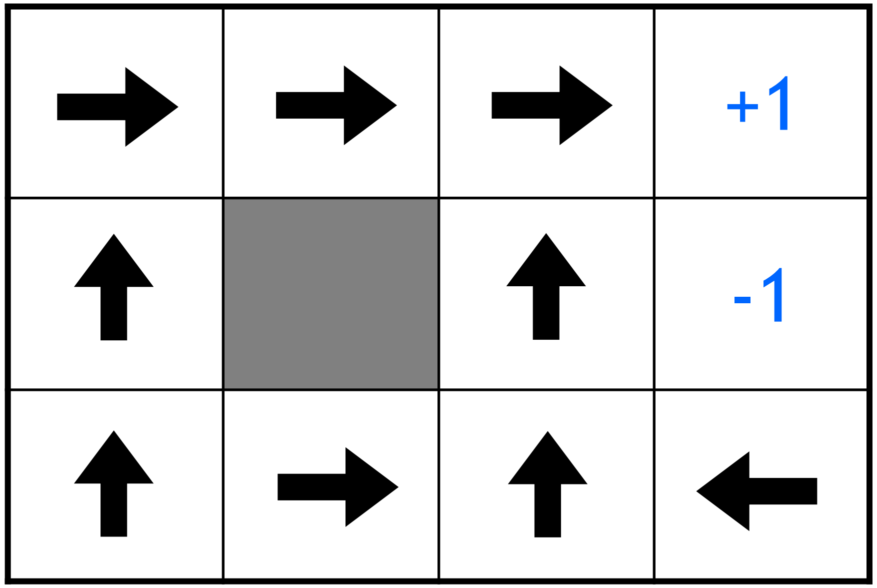
Optimal policy



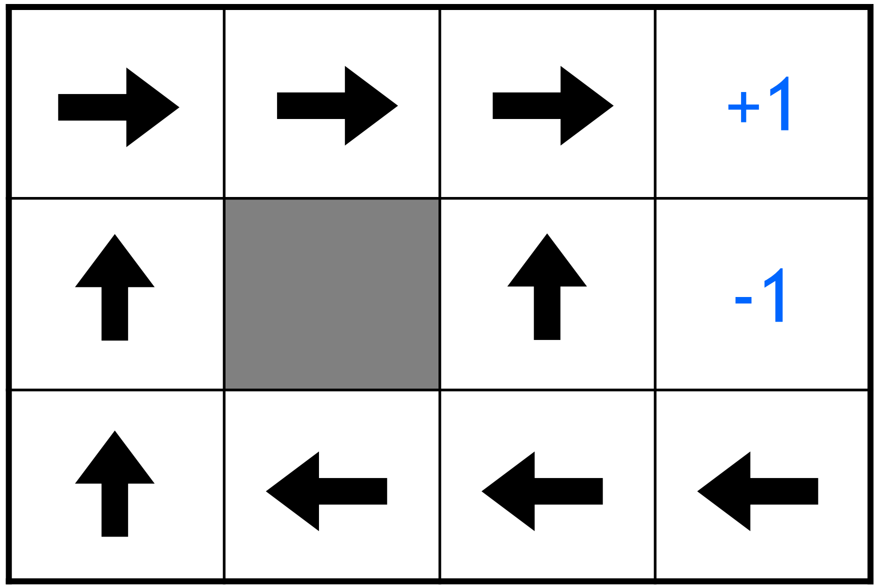
Reward for each step: -2



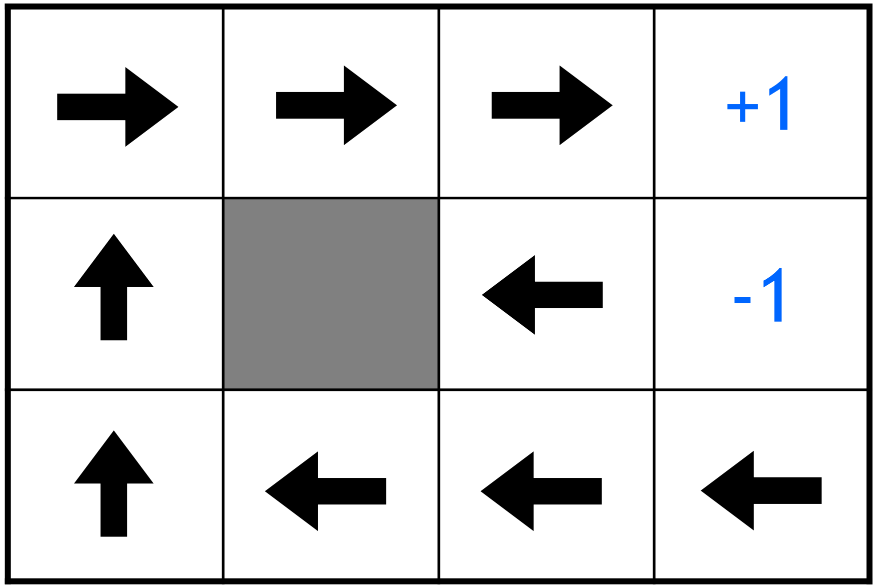
Reward for each step: -0.1



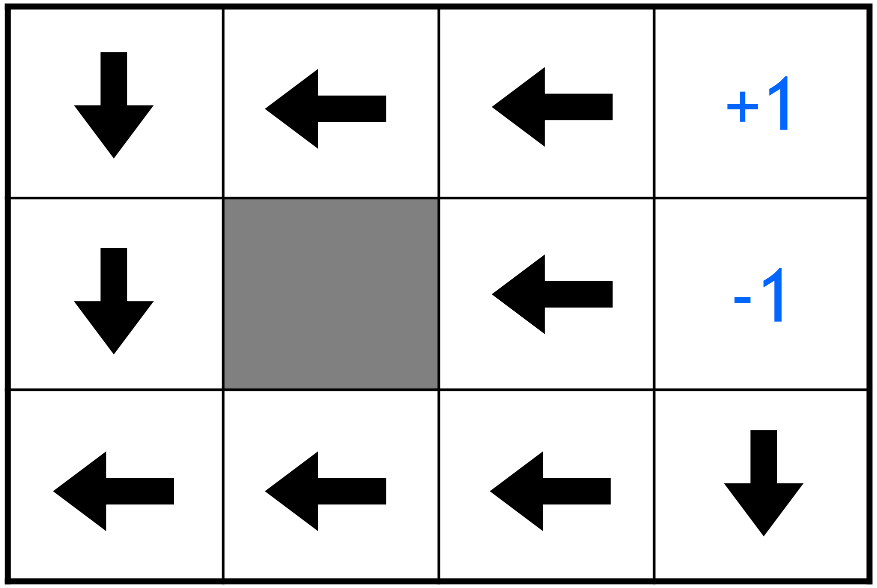
Reward for each step: -0.04



Reward for each step: -0.01

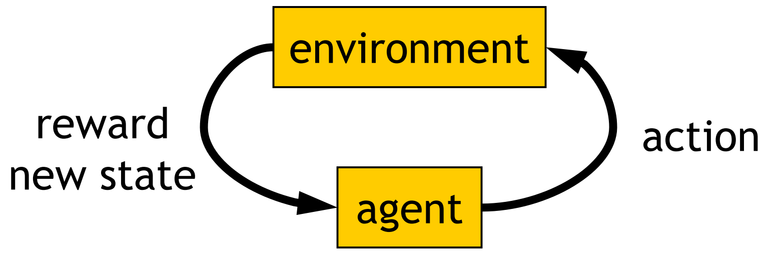


Reward for each step: +0.01



Markov Decision Process (MDP)

- set of states S, set of actions A, initial state S_0
- transition model P(s,a,s')
  - P([1,1], up, [1,2]) = 0.8
- reward function r(s)
  - r([4,3]) = +1
- goal: maximize cumulative reward in the long run



- policy: mapping from S to A
  - π(s) or π(s,a) (deterministic vs. stochastic)

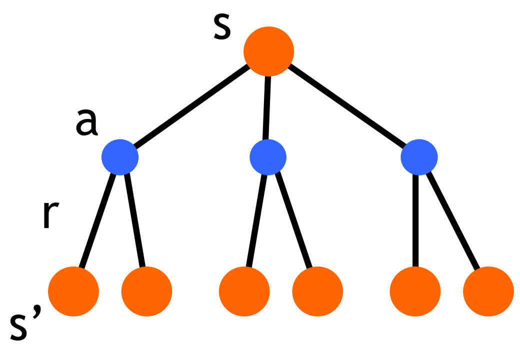
- reinforcement learning
  - transitions and rewards usually not available
  - how to change the policy based on experience
  - how to explore the environment

Computing return from rewards

- episodic (vs. continuing) tasks
  - “game over” after N steps
  - optimal policy depends on N; harder to analyze
- additive rewards
  - $V(s_0, s_1, \ldots) = r(s_0) + r(s_1) + r(s_2) + \ldots$
  - infinite value for continuing tasks
- discounted rewards
  - $V(s_0, s_1, \ldots) = r(s_0) + \gamma * r(s_1) + \gamma^2 * r(s_2) + \ldots$
  - value bounded if rewards bounded

Value functions

- state value function: $V^{\pi}(s)$
  - expected return when starting in *s* and following π
- state-action value function: $Q^{\pi}(s,a)$
  - expected return when starting in *s*, performing *a*, and following π
- useful for finding the optimal policy
  - can estimate from experience
  - pick the best action using $Q^{\pi}(s,a)$



- Bellman equation

$$V^{\pi}(s) = \sum_{a} \pi(s,a) \sum_{s'} P^{a}_{ss'} \left[r^{a}_{ss'} + \gamma V^{\pi}(s') \right] = \sum_{a} \pi(s,a) Q^{\pi}(s,a)$$

Optimal value functions

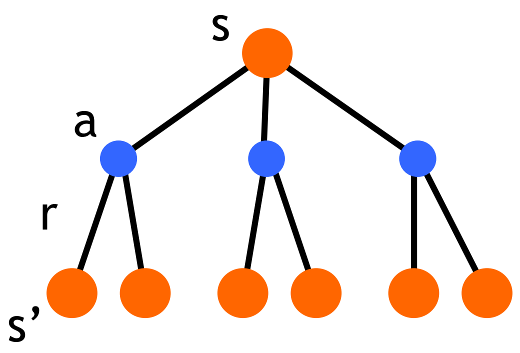
- there's a set of *optimal* policies
  - V^π defines partial ordering on policies
  - they share the same optimal value function

$$V^*(s) = \max_\pi V^\pi(s)$$

- Bellman optimality equation

$$V^*(s) = \max_a \sum_{s'} P^a_{ss'} \left[r^a_{ss'} + \gamma V^*(s') \right]$$

  - system of n non-linear equations
  - solve for V*(s)
  - easy to extract the optimal policy



- having Q*(s,a) makes it even simpler

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

Outline

- examples
- defining an RL problem
  - Markov Decision Processes
- solving an RL problem
  - Dynamic Programming
  - Monte Carlo methods
  - Temporal-Difference learning

Dynamic programming

- main idea
  - use value functions to structure the search for good policies
  - need a perfect model of the environment

- two main components
  - policy evaluation: compute V^π from π
  - policy improvement: improve π based on V^π

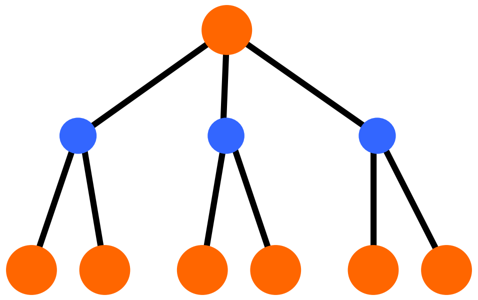
  - start with an arbitrary policy
  - repeat evaluation/improvement until convergence

Policy evaluation/improvement

- policy evaluation: π -> V^π
  - Bellman eqn’s define a system of n eqn’s
  - could solve, but will use iterative version

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

  - start with an arbitrary value function V_0, iterate until V_k converges



- policy improvement: V^π -> π’

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

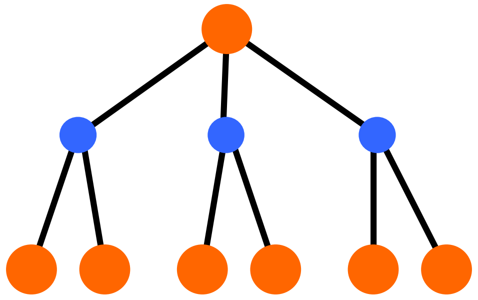
  - π’ either strictly better than π, or π’ is optimal (if π = π’)

Policy/Value iteration

- Policy iteration

$$\pi_0 \xrightarrow{E} V^{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} V^{\pi_1} \xrightarrow{I} \dots \xrightarrow{I} \pi^* \xrightarrow{E} V^*$$

  - two nested iterations; too slow
  - don't need to converge to V^{π_k}
    - just move towards it



- Value iteration

$$V_{k+1}(s) = \max_a \sum_{s'} P^a_{ss'} \left[r^a_{ss'} + \gamma V_k(s') \right]$$

  - use Bellman optimality equation as an update
  - converges to V*

Using DP

- need complete model of the environment and rewards
  - robot in a room
    - state space, action space, transition model

- can we use DP to solve
  - robot in a room?
  - back gammon?
  - helicopter?

Outline

- examples
- defining an RL problem
  - Markov Decision Processes
- solving an RL problem
  - Dynamic Programming
  - Monte Carlo methods
  - Temporal-Difference learning
- miscellaneous
  - state representation
  - function approximation
  - rewards

Monte Carlo methods

- don’t need full knowledge of environment
  - just experience, or
  - simulated experience
- but similar to DP
  - policy evaluation, policy improvement
- averaging sample returns
  - defined only for episodic tasks

Monte Carlo policy evaluation

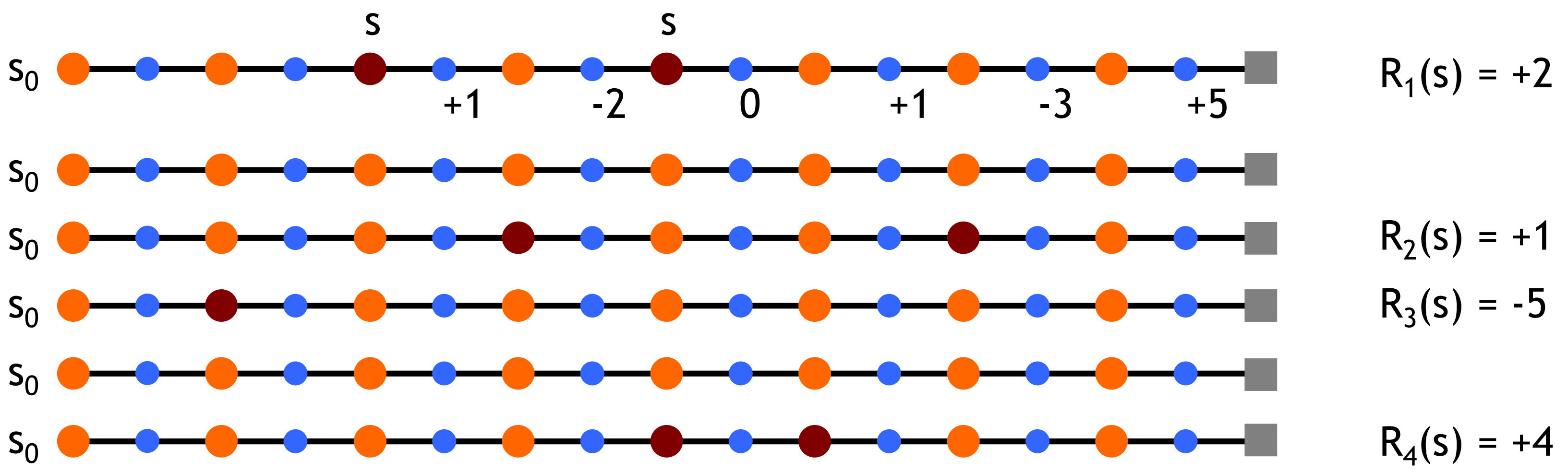
- want to estimate $V^{\pi}(s)$

  = expected return starting from s and following π

  – estimate as average of observed returns in state s

- first-visit MC

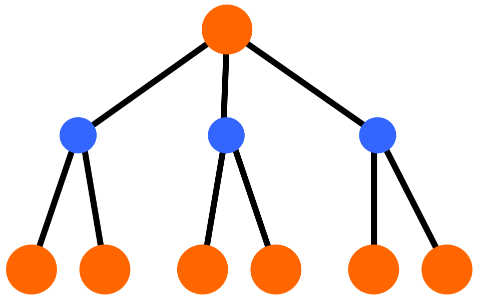
  – average returns following the first visit to state s



$V^{\pi}(s) \approx (2 + 1 - 5 + 4)/4 = 0.5$

Monte Carlo control

- V^π not enough for policy improvement
  - need exact model of environment



- estimate $Q^\pi(s,a)$

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

- MC control

  $$\pi_0 \rightarrow^E Q^{\pi_0} \rightarrow^I \pi_1 \rightarrow^E Q^{\pi_1} \rightarrow^I \ldots \rightarrow^I \pi^* \rightarrow^E Q^*$$

  - update after each episode

- non-stationary environment

  $$V(s) \leftarrow V(s) + \alpha\,[R - V(s)]$$

- a problem
  - greedy policy won't explore all actions

Maintaining exploration

- deterministic/greedy policy won't explore all actions
  - don't know anything about the environment at the beginning
  - need to try all actions to find the optimal one

- maintain exploration
  - use *soft* policies instead: $\pi(s,a)>0$ (for all s,a)

- ε-greedy policy
  - with probability 1-ε perform the optimal/greedy action
  - with probability ε perform a random action

  - will keep exploring the environment
  - slowly move it towards greedy policy: ε -> 0

Summary of Monte Carlo

- don’t need model of environment
  - averaging of sample returns
  - only for episodic tasks
- learn from sample episodes or simulated experience
- can concentrate on “important” states
  - don’t need a full sweep
- need to maintain exploration
  - use soft policies

Outline

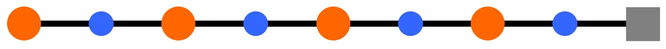
- examples
- defining an RL problem
  - Markov Decision Processes
- solving an RL problem
  - Dynamic Programming
  - Monte Carlo methods
  - Temporal-Difference learning
- miscellaneous
  - state representation
  - function approximation
  - rewards

Temporal Difference Learning

- combines ideas from MC and DP
  - like MC: learn directly from experience (don't need a model)
  - like DP: learn from values of successors
  - works for continuous tasks, usually faster than MC

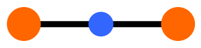
- constant-alpha MC:
  - have to wait until the end of episode to update

$$V(s_t) \leftarrow V(s_t) + \alpha\,[\underbrace{R_t}_{\text{target}} - V(s_t)]$$



- simplest TD
  - update after every step, based on the successor

$$V(s_t) \leftarrow V(s_t) + \alpha\left[\underbrace{r_{t+1} + \gamma V(s_{t+1})}_{\text{target}} - V(s_t)\right]$$



MC vs. TD

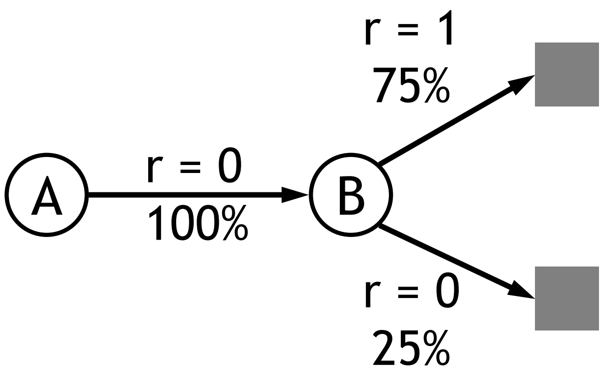
- observed the following 8 episodes:

A – 0, B – 0	B – 1	B – 1	B - 1
B – 1	B – 1	B – 1	B – 0

- MC and TD agree on V(B) = 3/4

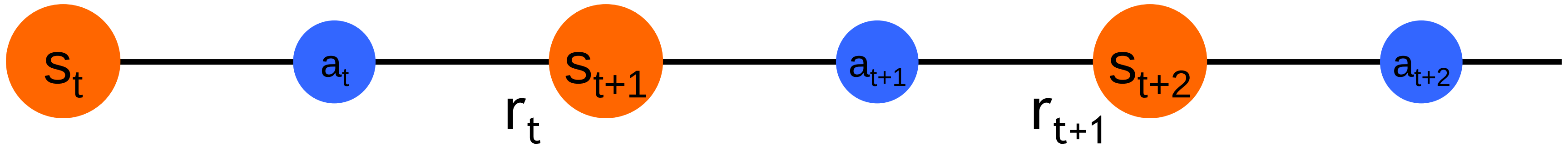
- MC: V(A) = 0
  - converges to values that minimize the error on training data

- TD: V(A) = 3/4
  - converges to ML estimate of the Markov process



Sarsa

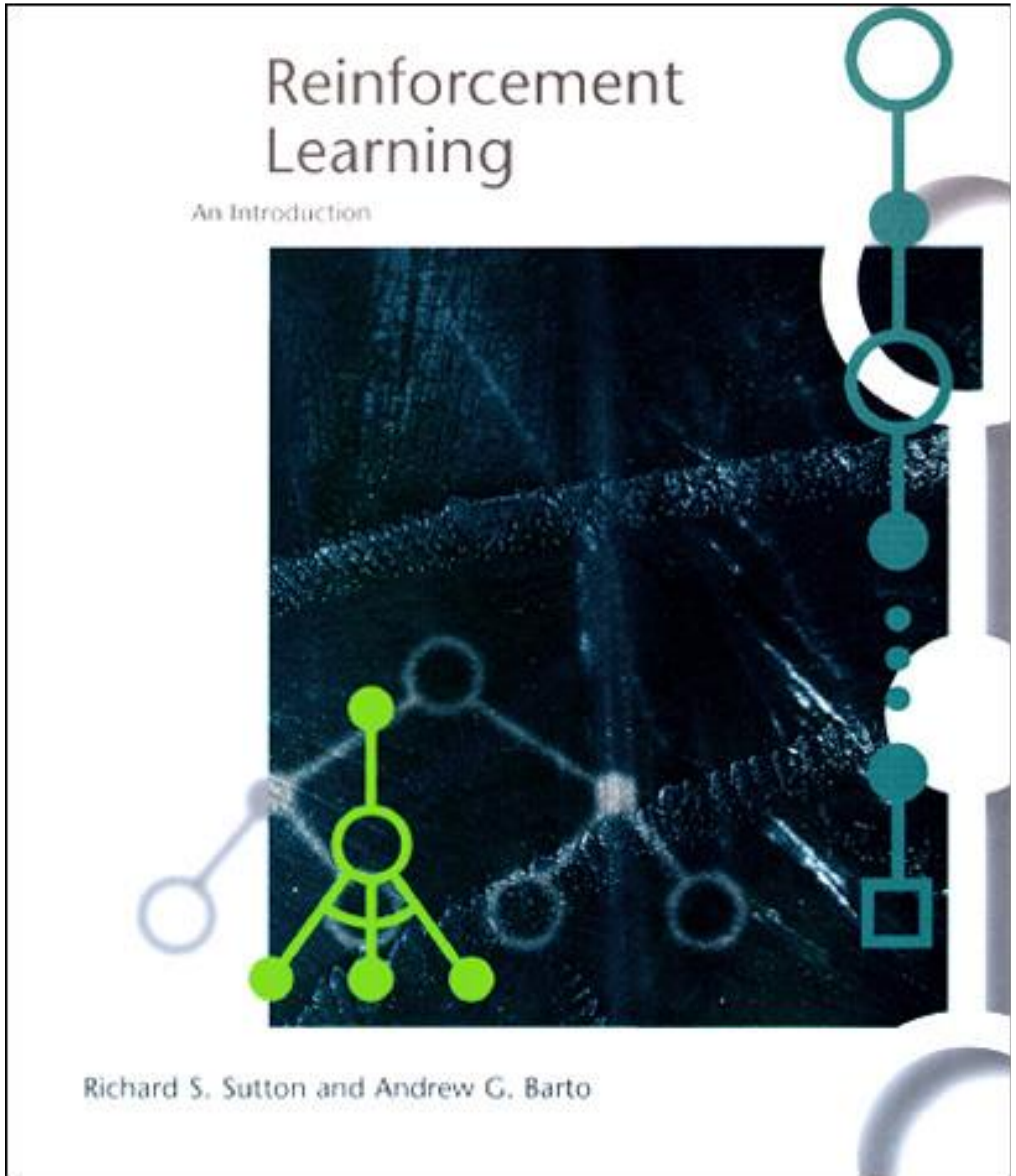
- again, need Q(s,a), not just V(s)



$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right]$$

- control
  - start with a random policy
  - update Q and π after each step
  - again, need ε-soft policies

The RL Intro book



Richard Sutton, Andrew Barto
Reinforcement Learning,
An Introduction

http://www.cs.ualberta.ca/
~sutton/book/the-book.html

Q-learning

- before: on-policy algorithms
  - start with a random policy, iteratively improve
  - converge to optimal

- Q-learning: off-policy
  - use any policy to estimate Q

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right]$$

  - Q directly approximates Q* (Bellman optimality eqn)
  - independent of the policy being followed
  - only requirement: keep updating each (s,a) pair

- Sarsa

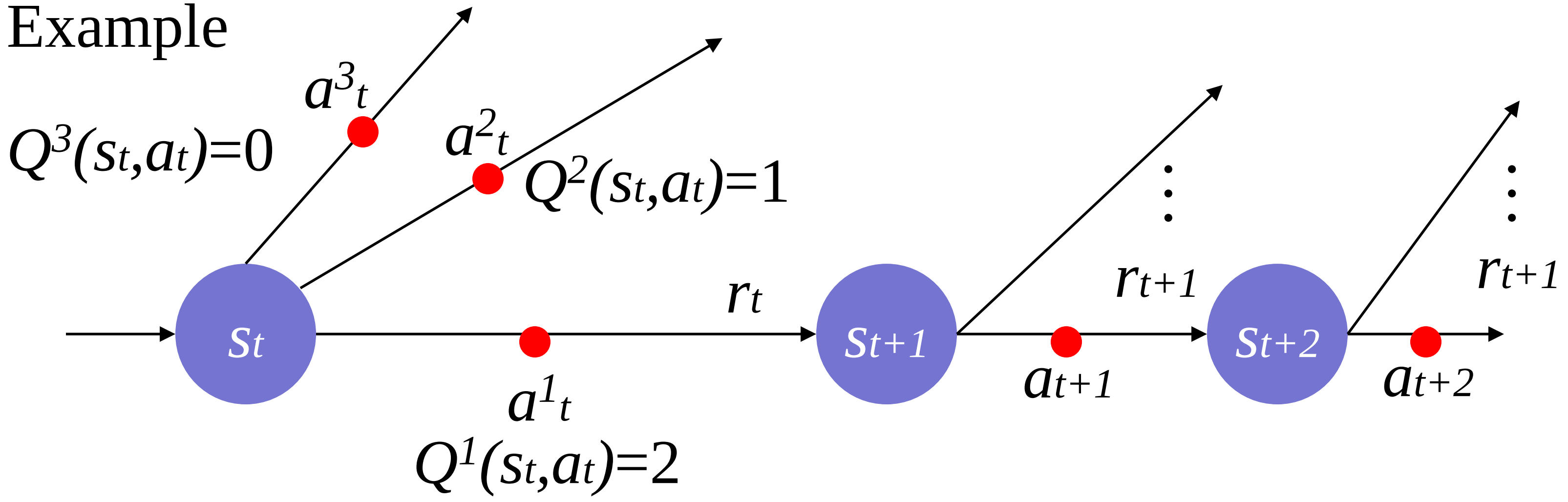
$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right]$$

Q value

When an agent take action a_t in state s_t at time t,
the predicted future rewards is defined as $Q(s_t,a_t)$.

$$Q(s_t,a_t) = E\{r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} + \cdots\}$$

Example



Generally speaking, an agent should take action a^1_t because the corresponding Q value $Q^1(s_t,a_t)$ *is max*.

Q learning

First, Q value can be transformed as follows.

$$
\begin{aligned}
Q(s_t, a_t) &= E\left\{r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} + \cdots\right\} \\
&= E\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}\right\} \qquad \text{———————— ①} \\
&= E\left\{r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2}\right\} \\
&= E\left\{r_{t+1} + \gamma Q(s_{t+1}, a_{t+1})\right\}
\end{aligned}
$$

(①
)

As a result, the Q value at time *t* is easily calculated by r_{t+1} and Q value of the next step.

Q learning

Q values is updated every step.

When an agent take action a_t in state s_t,
and gets reward r, the Q value is updated as follows.

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha\underbrace{\left[\underbrace{r + \gamma \max_a Q(s_{t+1},a)}_{\text{target value}} - \underset{\text{current value}}{Q(s_t,a_t)}\right]}_{\text{TD error}}$$

α: step size parameter (learning rate)

Q learning algorithm

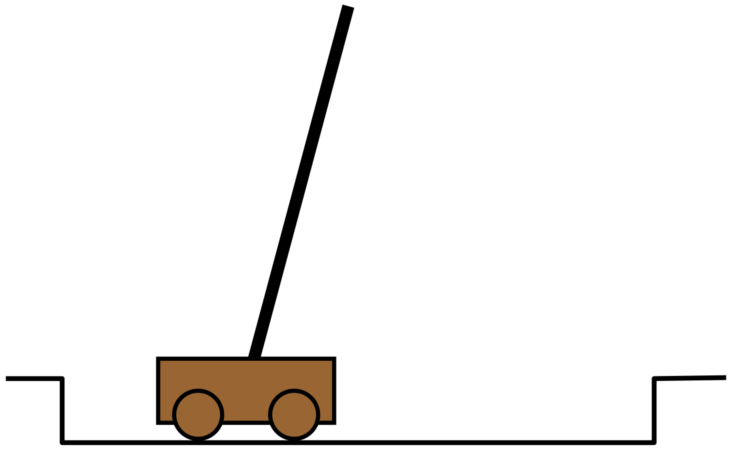
```
Initialize $Q(s,a)$ arbitrarily
Repeat (for each episode):
    initialize $s$
    Repeat (for each step of episode):
        Choose $a$ from $s$ using policy derived from $Q$
                                    (e.g., greedy, ε-greedy)
        take action $a$, observe $r$, $s'$
        $Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\right]$
        $s \leftarrow s'$;
    until $s$ is terminal
```

Outline

- examples
- defining an RL problem
  - Markov Decision Processes
- solving an RL problem
  - Dynamic Programming
  - Monte Carlo methods
  - Temporal-Difference learning
- miscellaneous
  - state representation
  - function approximation
  - rewards

State representation

- pole-balancing
  - move car left/right to keep the pole balanced
- state representation
  - position and velocity of car
  - angle and angular velocity of pole
- what about *Markov property*?
  - would need more info
  - noise in sensors, temperature, bending of pole
- solution
  - coarse discretization of 4 state variables
    - left, center, right
  - totally non-Markov, but still works



Function approximation

- represent V_t as a parameterized function
  - linear regression, decision tree, neural net, …
  - linear regression: $V_t(s) = \vec{\theta}_t^T \vec{\phi}_s = \sum_{i=1}^{n} \theta_t(i)\phi_s(i)$
- update parameters instead of entries in a table
  - better generalization
    - fewer parameters and updates affect "similar" states as well
- TD update

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

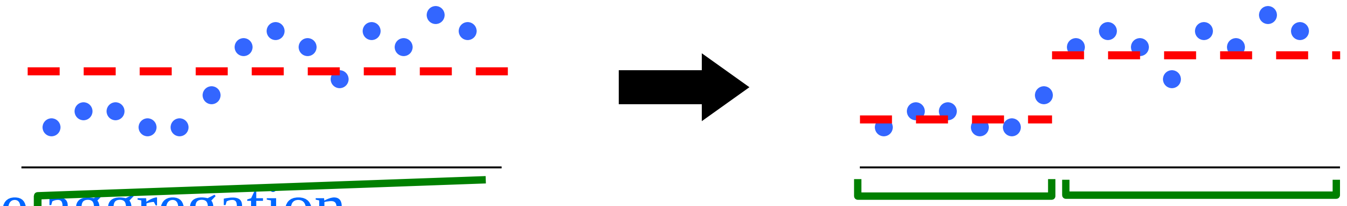
$$V\underbrace{(s_t)}_{x} \mapsto \underbrace{r_{t+1} + \gamma V(s_{t+1})}_{y}$$

  - treat as one data point for regression
  - want method that can learn on-line (update after each step)

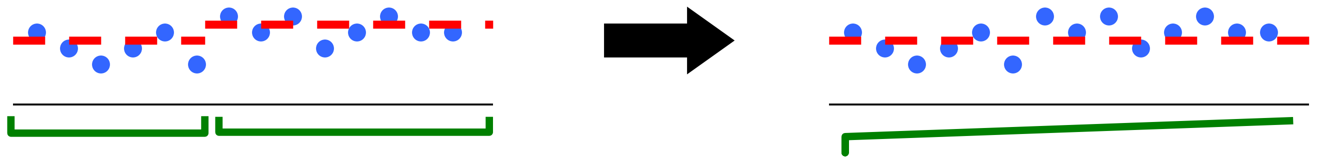
Splitting and aggregation

- want to discretize the state space
  - learn the best discretization during training

- splitting of state space
  - start with a single state
  - split a state when different *parts of that state* have different values



- state aggregation
  - start with many states
  - merge states with similar values



Designing rewards

- robot in a maze
  - episodic task, not discounted, +1 when out, 0 for each step
- chess
  - GOOD: +1 for winning, -1 losing
  - BAD: +0.25 for taking opponent's pieces
    - high reward even when lose
- rewards
  - rewards indicate what we want to accomplish
  - NOT how we want to accomplish it
- shaping

  - positive reward often very "far away"
  - rewards for achieving subgoals (domain knowledge)
  - also: adjust initial policy or initial value function

Summary

- Reinforcement learning
  - use when need to make decisions in uncertain environment

- solution methods
  - dynamic programming
    - need complete model

  - Monte Carlo
  - time-difference learning (Sarsa, Q-learning)

- most work
  - algorithms simple
  - need to design features, state representation, rewards