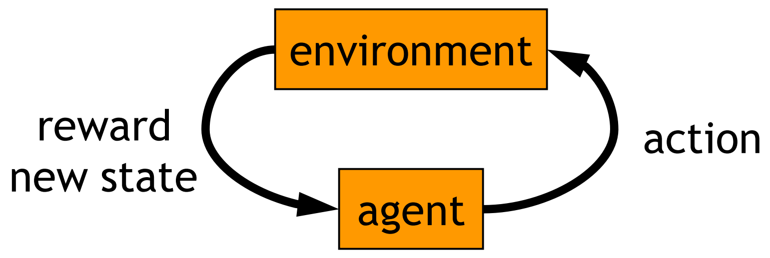
Reinforcement Learning

Faculty
R. Rajkumar
School of Computing | SRMIST

Machine Learning

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



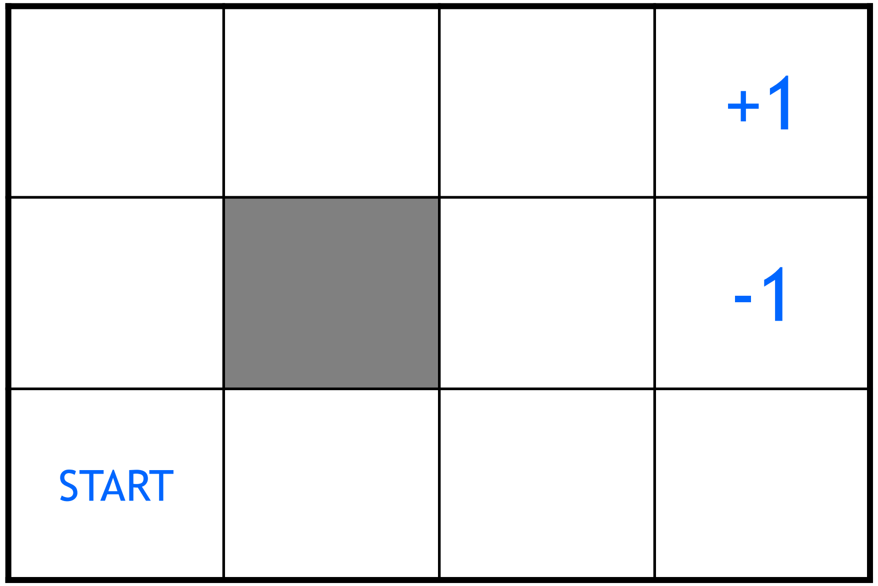
Reinforcement Learning

- examples

- defining an RL problem
  - Markov Decision Processes

- solving an RL problem
  - Dynamic Programming
  - Monte Carlo methods
  - Temporal-Difference learning

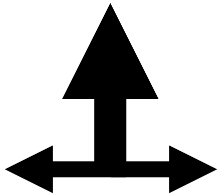
Robot in a room



actions: UP, DOWN, LEFT, RIGHT

UP

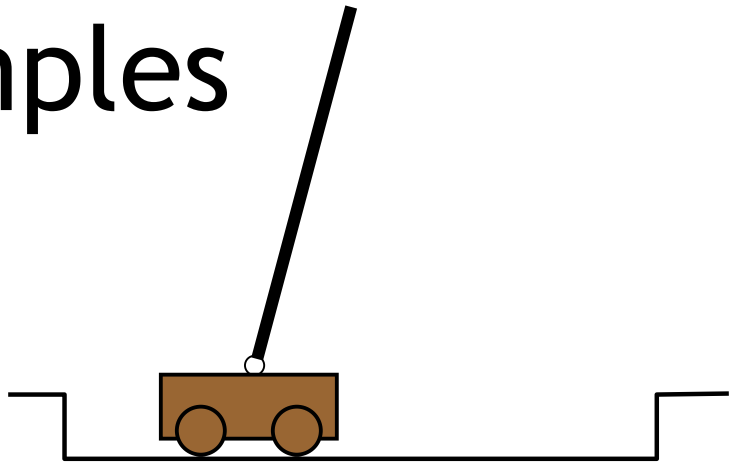
80% move UP
10% move LEFT
10% move RIGHT



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

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

Other examples



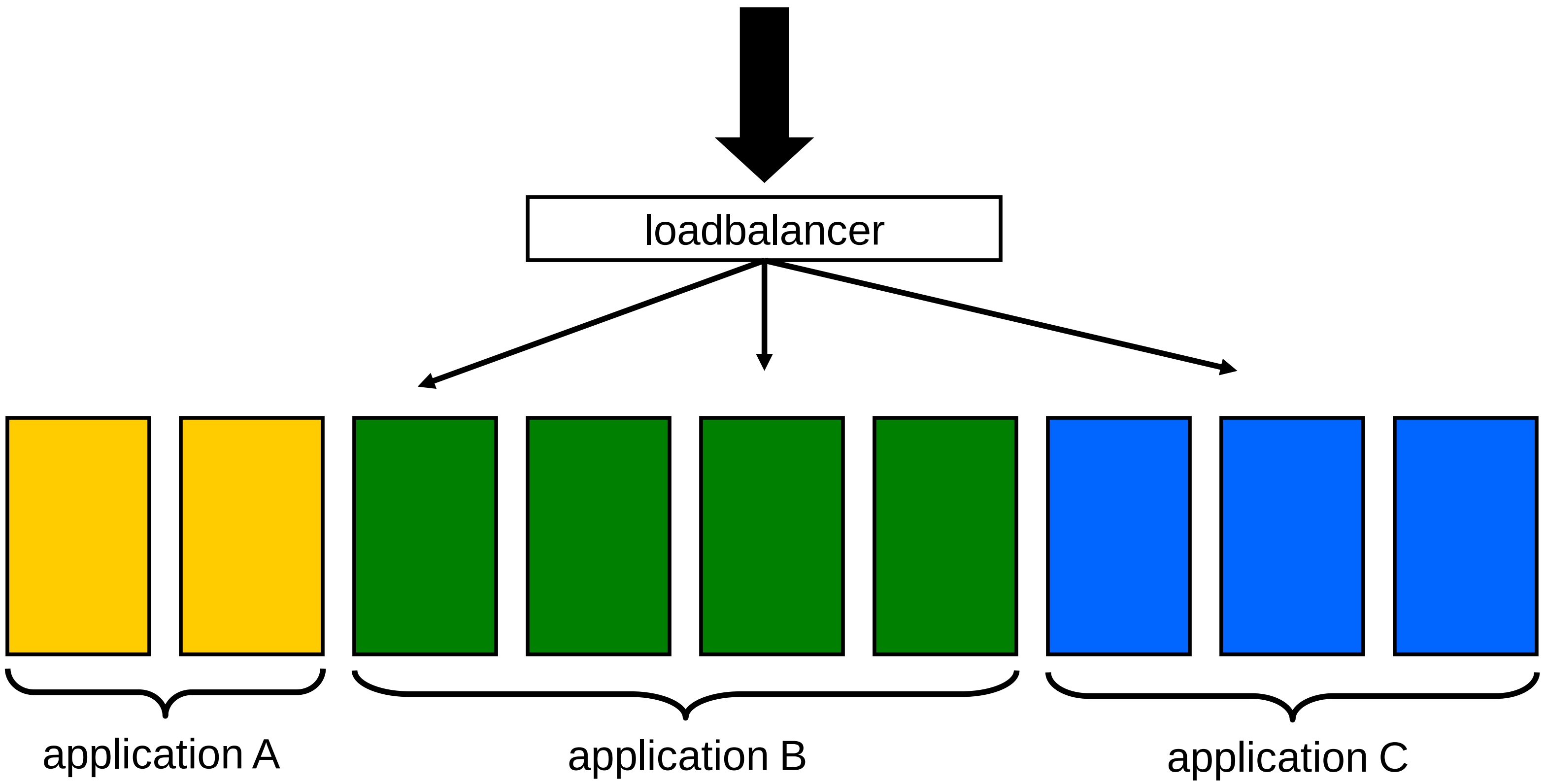
- pole-balancing
- TD-Gammon [Gerry Tesauro]
- helicopter [Andrew Ng]

- 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

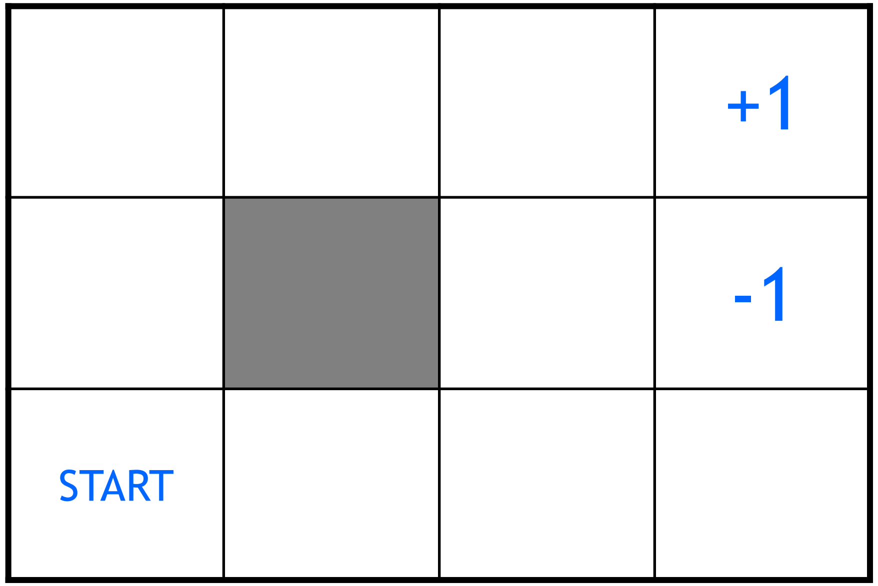


- A Hybrid Reinforcement Learning Approach to Autonomic Resource Allocation
  - Tesauro, Jong, Das, Bennani (IBM)
  - ICAC 2006

Outline

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

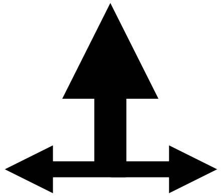
Robot in a room



actions: UP, DOWN, LEFT, RIGHT

UP

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

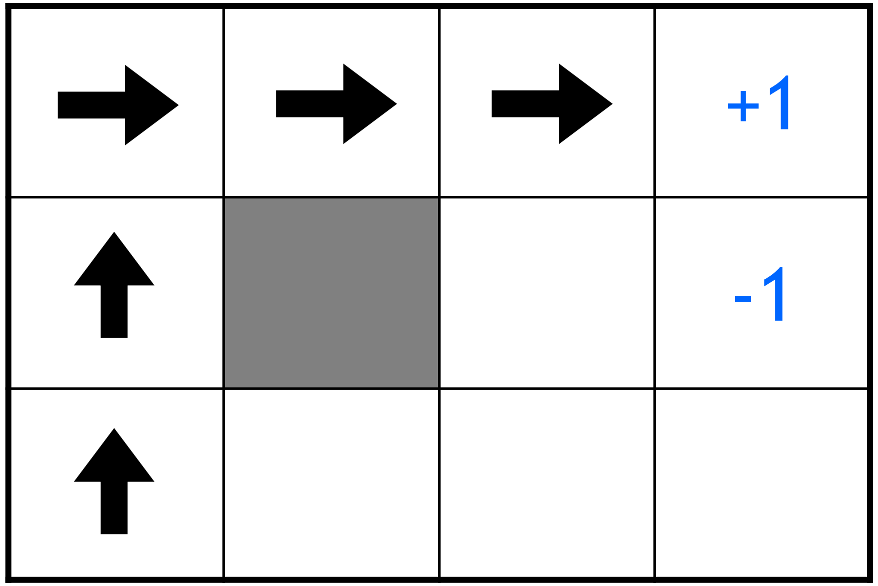


reward +1 at [4,3], -1 at [4,2]
reward -0.04 for each step

- states
- actions
- rewards

- what is the solution?

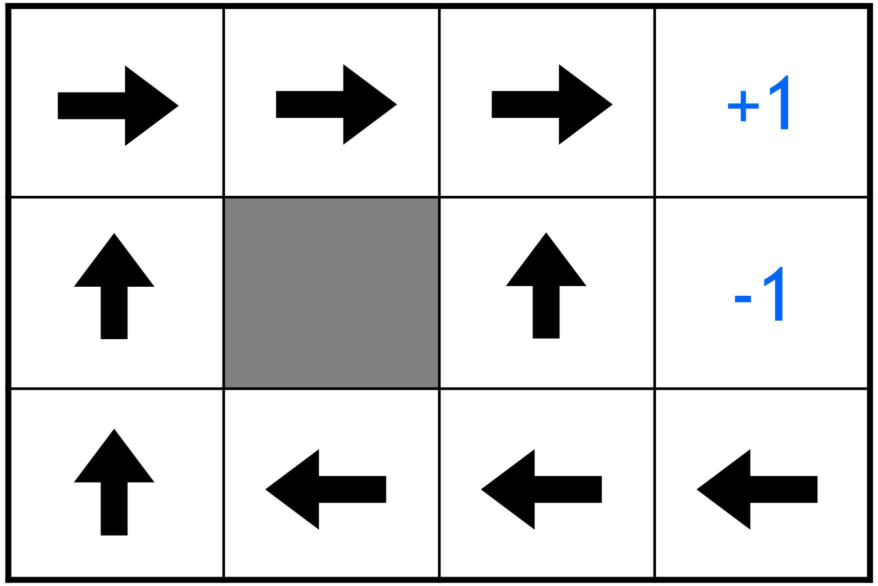
Is this a solution?



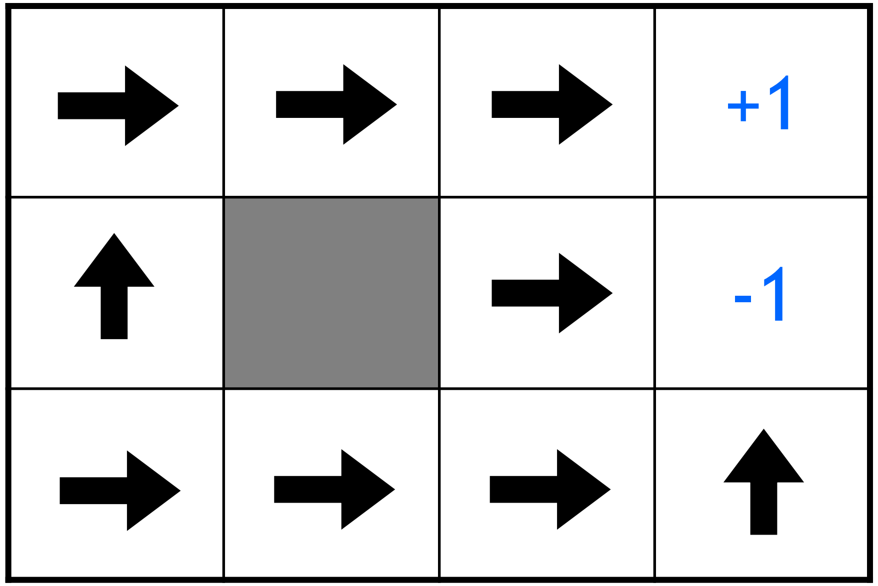
- only if actions deterministic
  - not in this case (actions are stochastic)

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

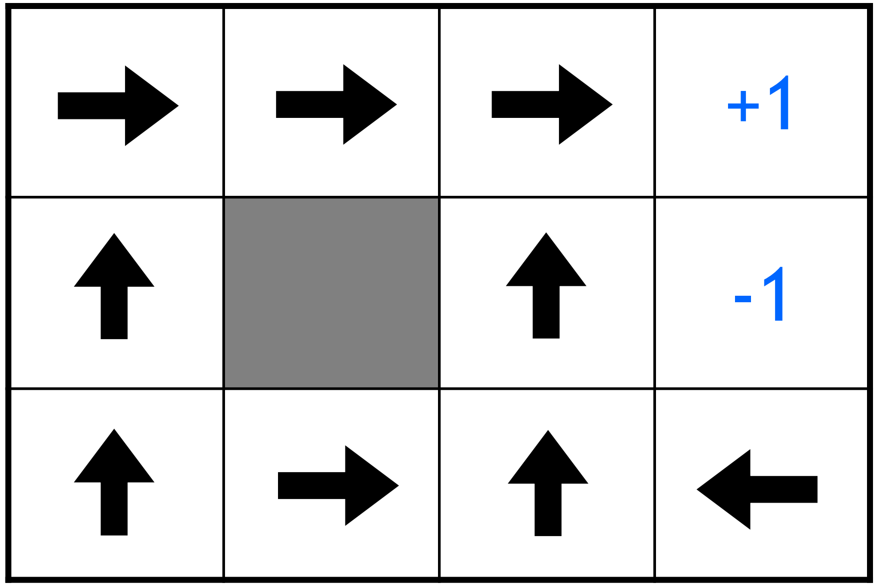
Optimal policy



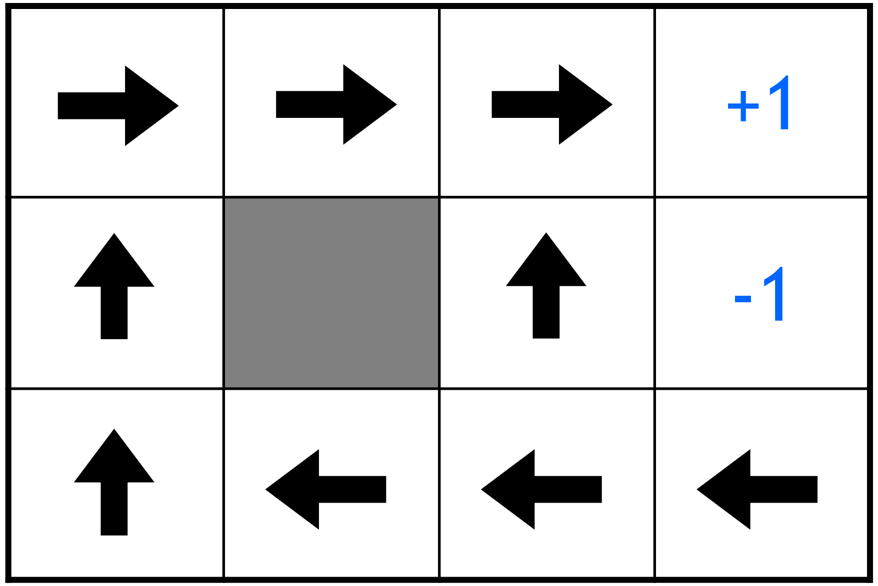
Reward for each step: -2



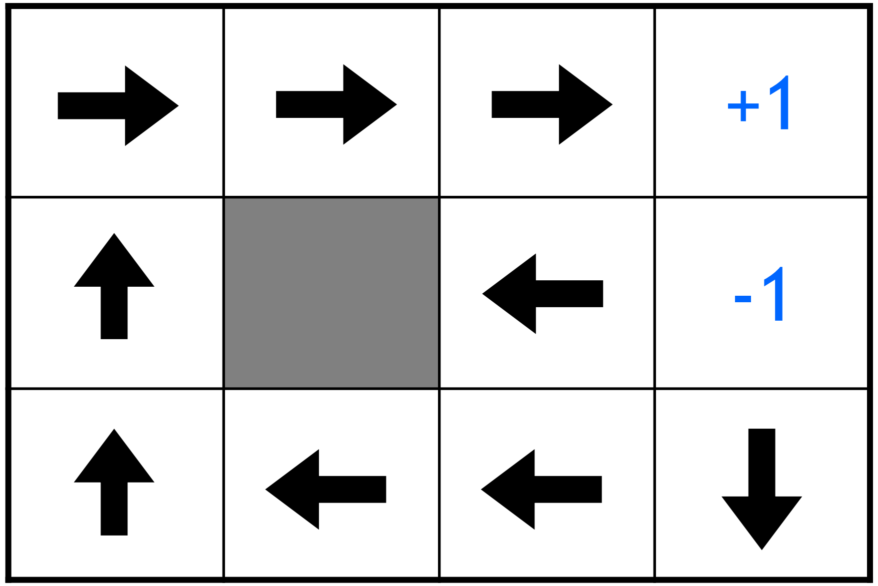
Reward for each step: -0.1



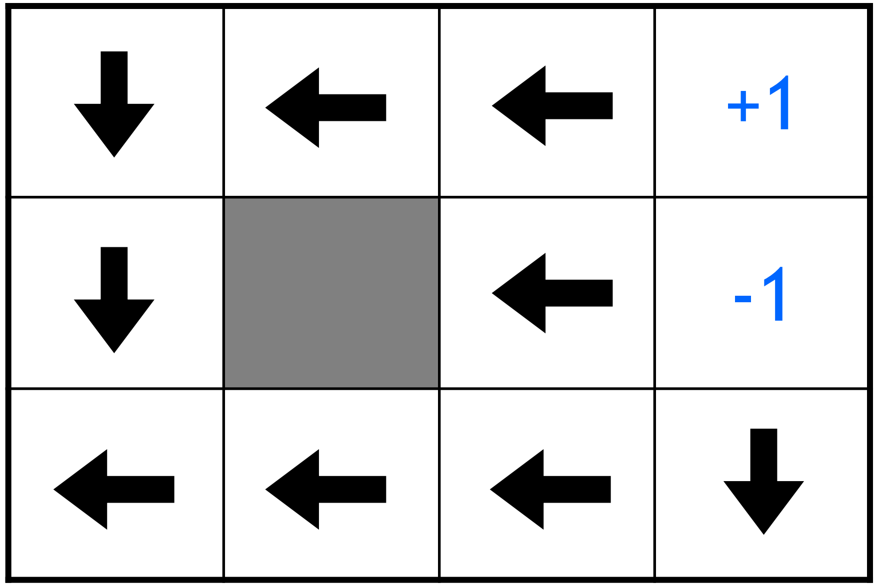
Reward for each step: -0.04



Reward for each step: -0.01

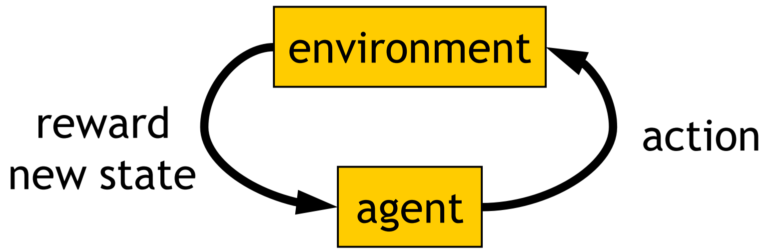


Reward for each step: +0.01



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
  - $\pi(s)$ or $\pi(s,a)$ (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

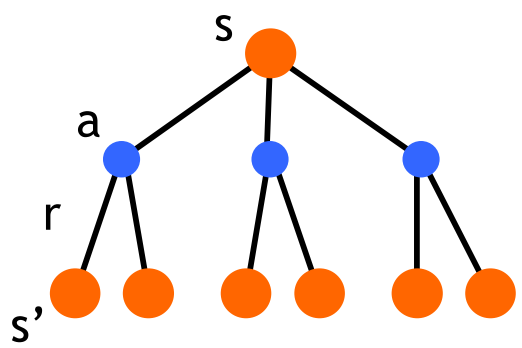
- episodic (vs. continuing) tasks
  - "game over" after N steps
  - optimal policy depends on N; harder to analyze

- 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

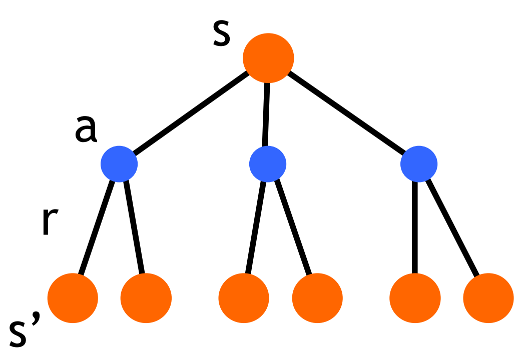
- there's a set of *optimal* policies
  - V^π defines partial ordering on policies
  - they share the same optimal value function

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

- 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

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

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

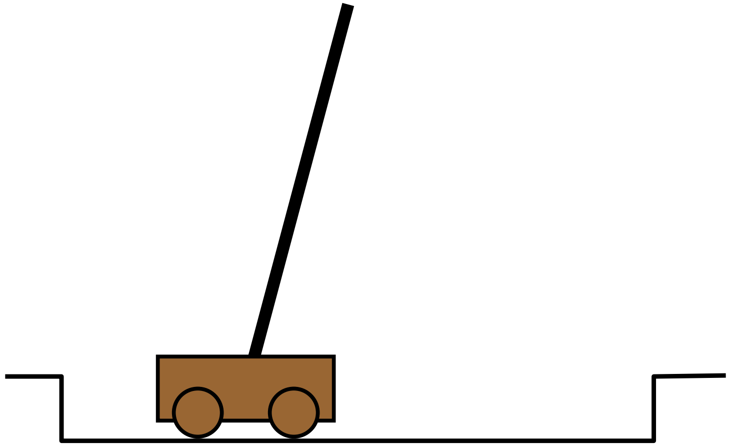
  - Q directly approximates Q* (Bellman optimality eqn)
  - independent of the policy being followed
  - only requirement: keep updating each (s,a) pair

- Sarsa

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

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

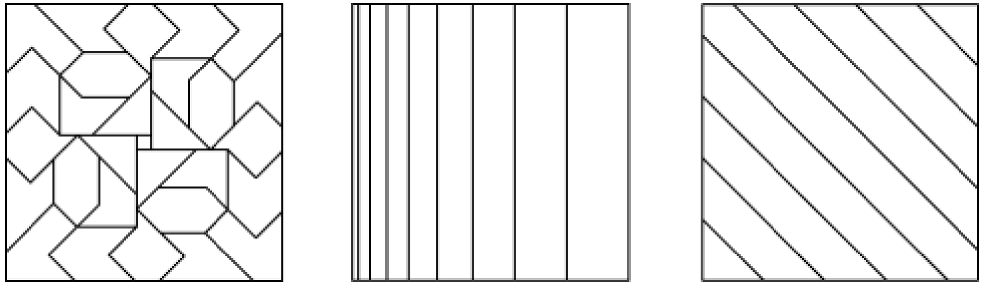
$$V(s_t) \leftarrow V(s_t) + \alpha \left[r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \right]$$

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

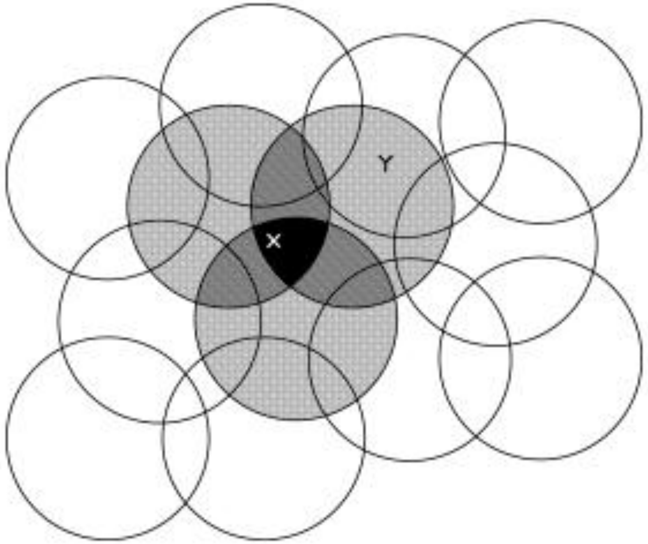
  - treat as one data point for regression
  - want method that can learn on-line (update after each step)

Features

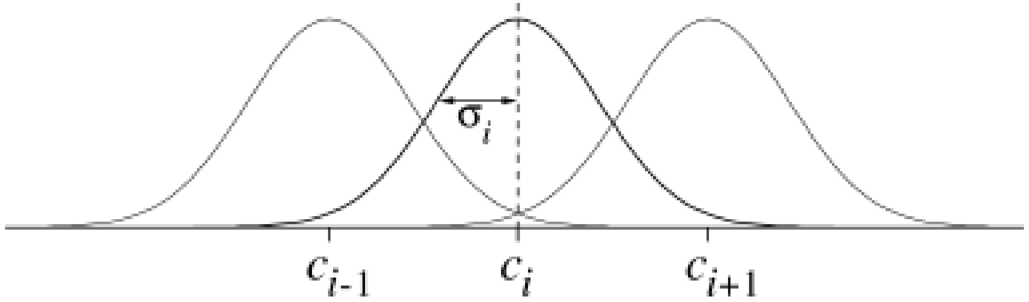
- tile coding, coarse coding
  - binary features



a) Irregular　　b) Log stripes　　c) Diagonal stripes



- radial basis functions
  - typically a Gaussian
  - between 0 and 1

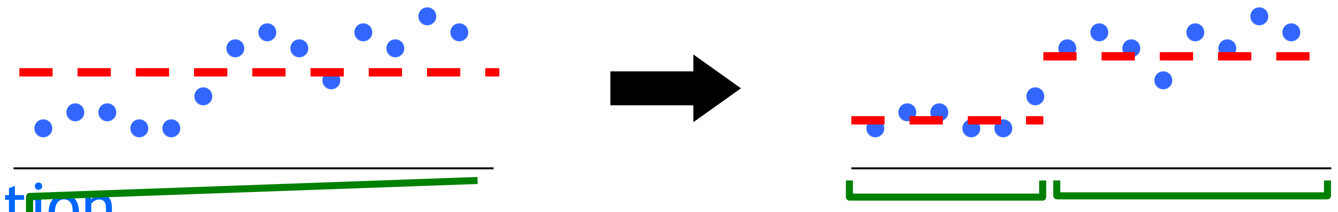


[Sutton & Barto, Reinforcement Learning]

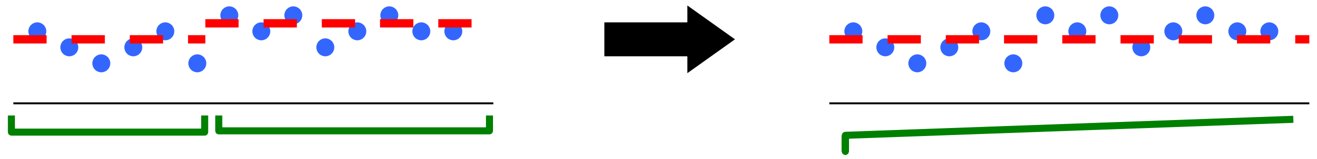
Splitting and aggregation

- 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



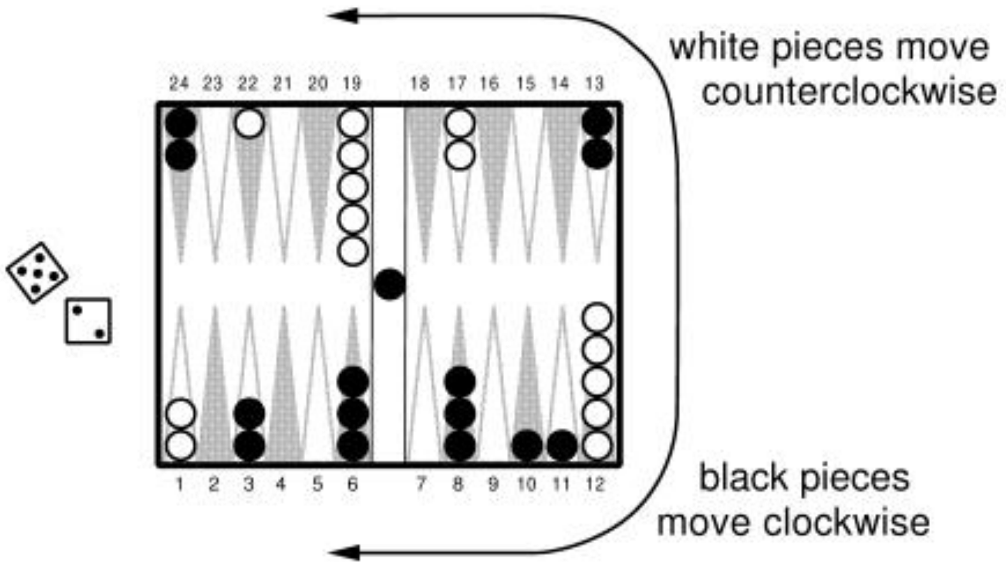
Designing rewards

- 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

Case study: Back gammon

- rules
  - 30 pieces, 24 locations
  - roll 2, 5: move 2, 5
  - hitting, blocking
  - branching factor: 400
- implementation
  - use TD(λ) and neural nets
  - 4 binary features for each position on b
  - no BG expert knowledge
- results
  - TD-Gammon 0.0: trained against itself (300,000 games)
    - as good as best previous BG computer program (also by Tesauro)
    - lot of expert input, hand-crafted features
  - TD-Gammon 1.0: add special features
  - TD-Gammon 2 and 3 (2-ply and 3-ply search)
    - 1.5M games, beat human champion



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

Demo

https://www.youtube.com/watch?v=aTpJJR1WBuc