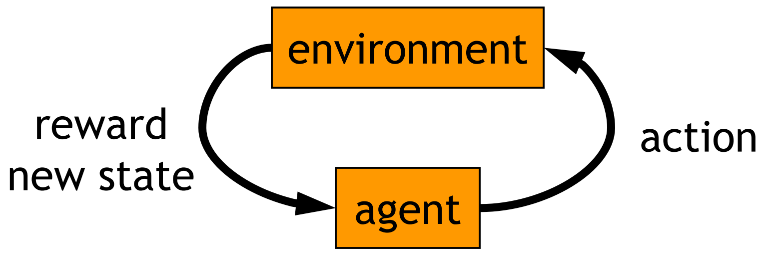
# Reinforcement Learning Tutorial

Peter Bodík
RAD Lab, UC Berkeley

# Previous Lectures

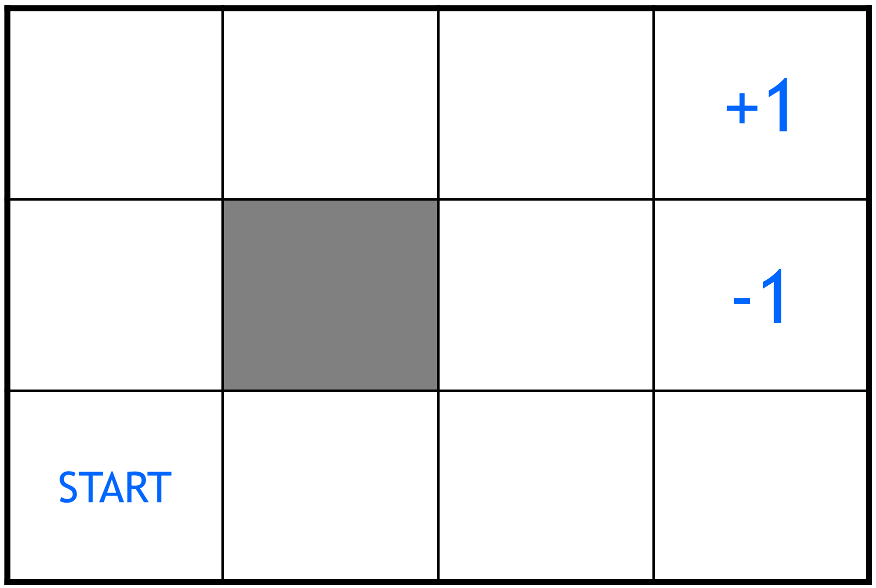
- Supervised learning
  - classification, regression
- Unsupervised learning
  - clustering
- Reinforcement learning
  - more general than supervised/unsupervised learning
  - learn from interaction w/ environment to achieve a goal



# Today

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

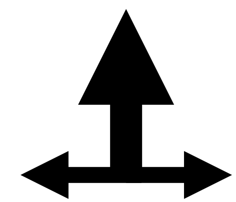
# Robot in a room




actions: UP, DOWN, LEFT, RIGHT

**UP**

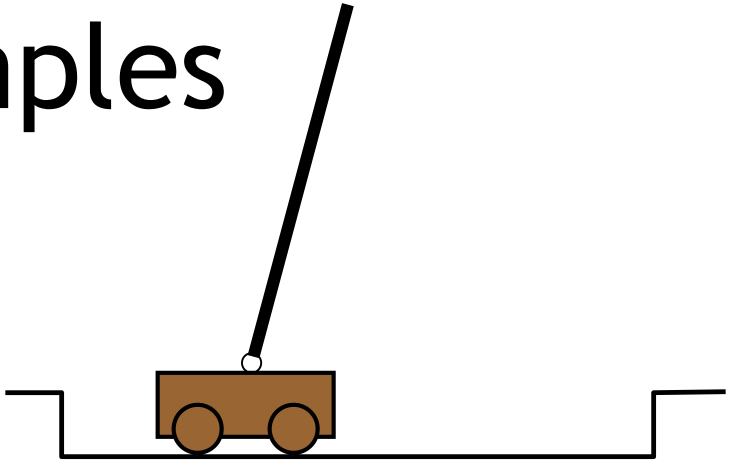
80%    move UP
10%    move LEFT
10%    move RIGHT



- 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



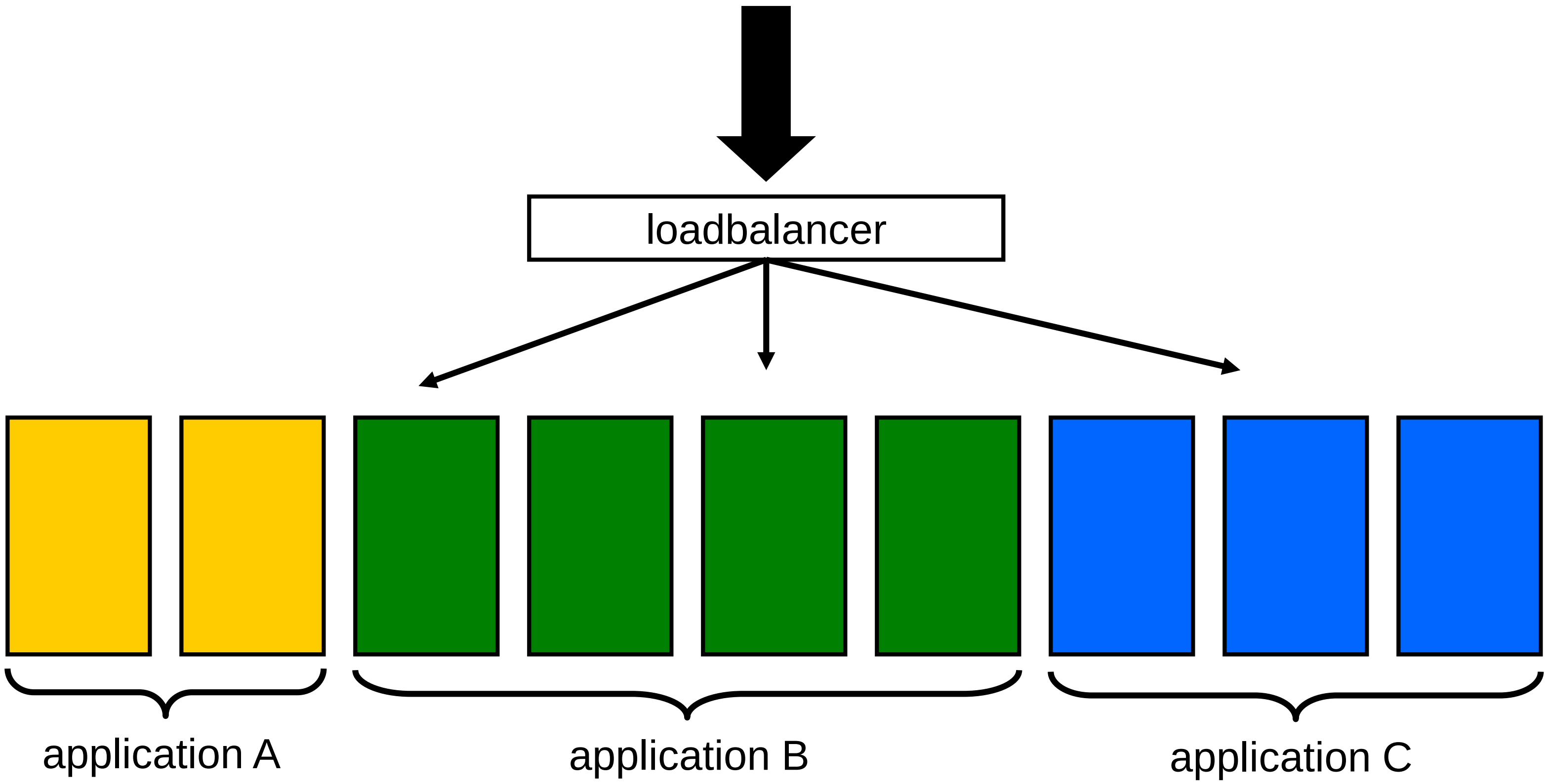
- pole-balancing
- TD-Gammon [Gerry Tesauro]
- helicopter [Andrew Ng]

- 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

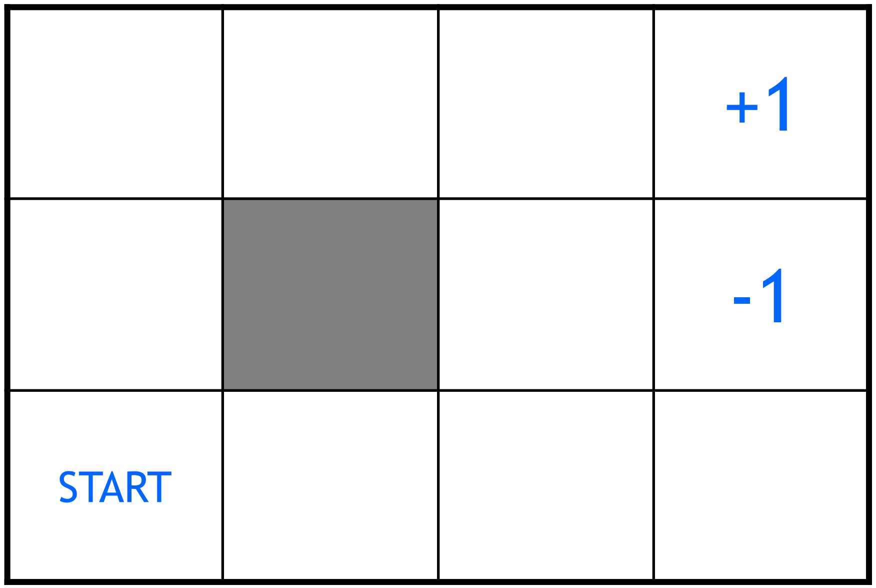



- A Hybrid Reinforcement Learning Approach to Autonomic Resource Allocation
  - Tesauro, Jong, Das, Bennani (IBM)
  - ICAC 2006

# Outline

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

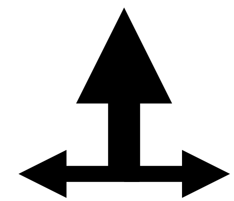
# Robot in a room




actions: UP, DOWN, LEFT, RIGHT

**UP**

| | |
|---|---|
| 80% | move UP |
| 10% | move LEFT |
| 10% | move RIGHT |

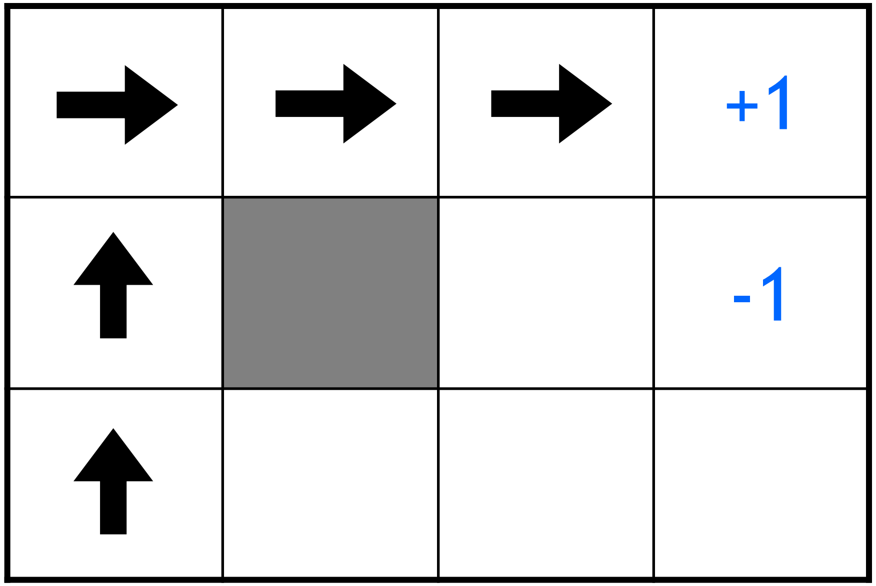


reward +1 at [4,3], -1 at [4,2]
reward -0.04 for each step

- states
- actions
- rewards

- what is the solution?

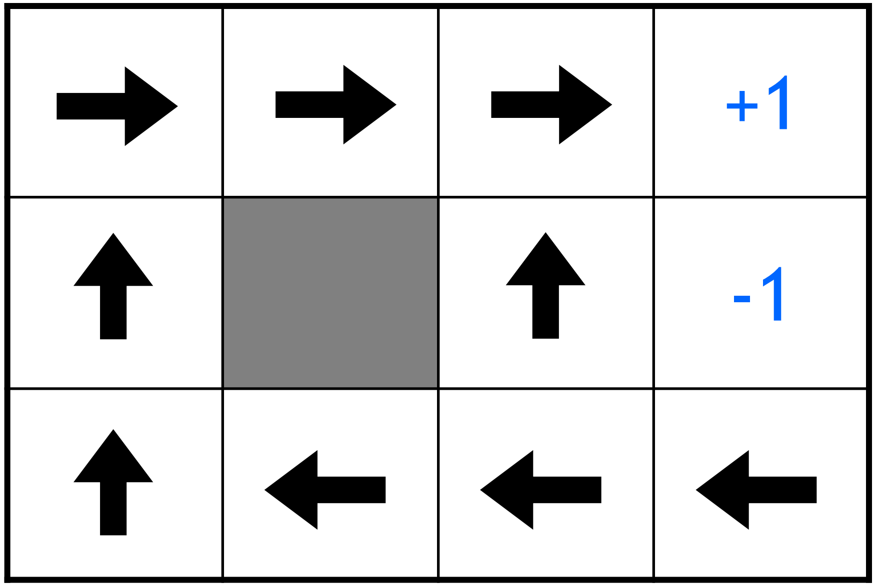
# Is this a solution?



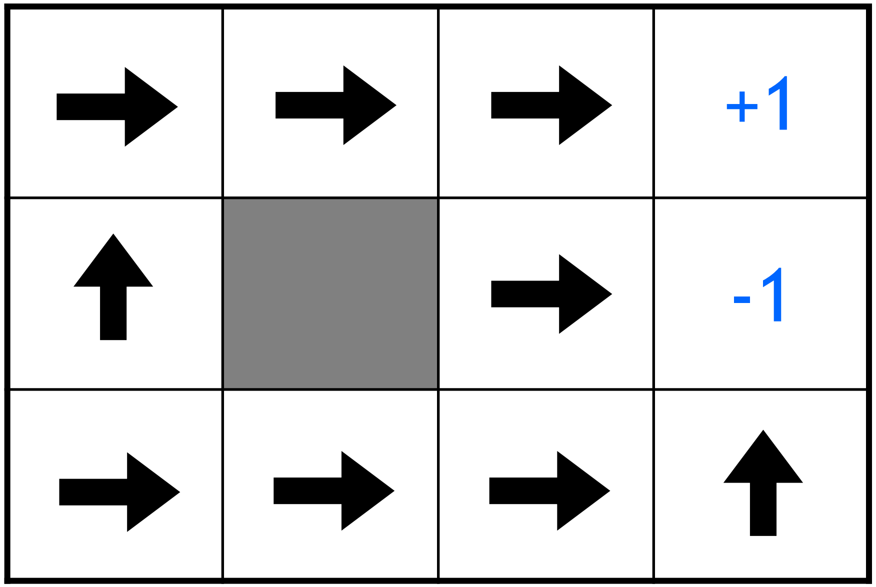

- only if actions deterministic
  - not in this case (actions are stochastic)

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

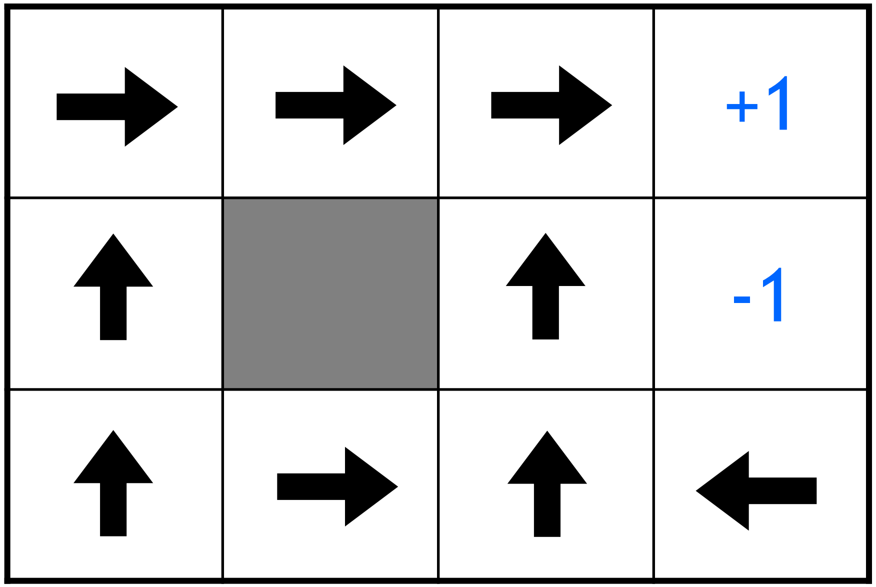
# Optimal policy



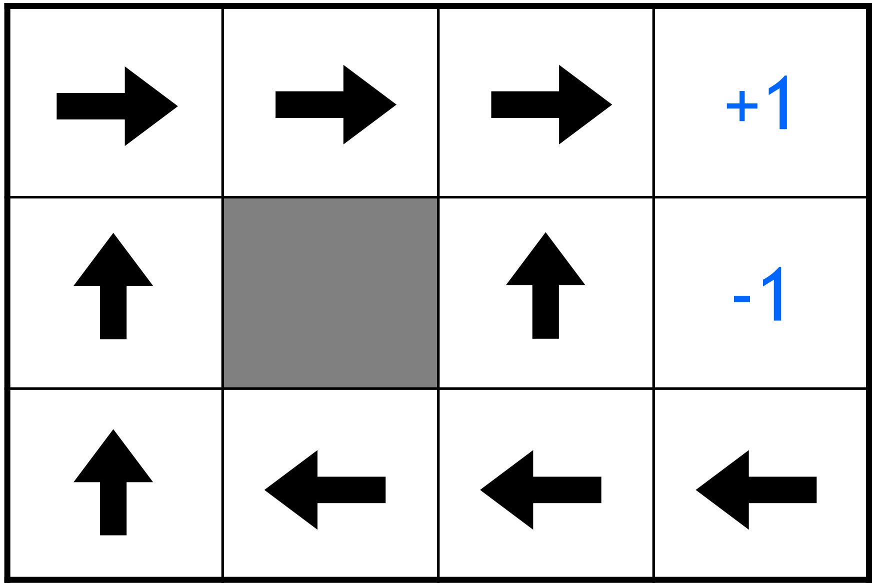
# Reward for each step: -2



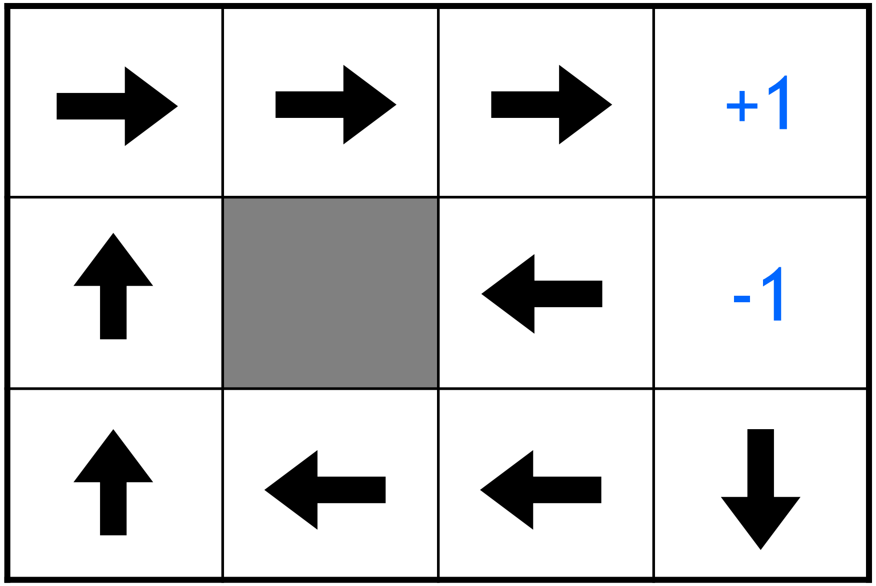
# Reward for each step: -0.1



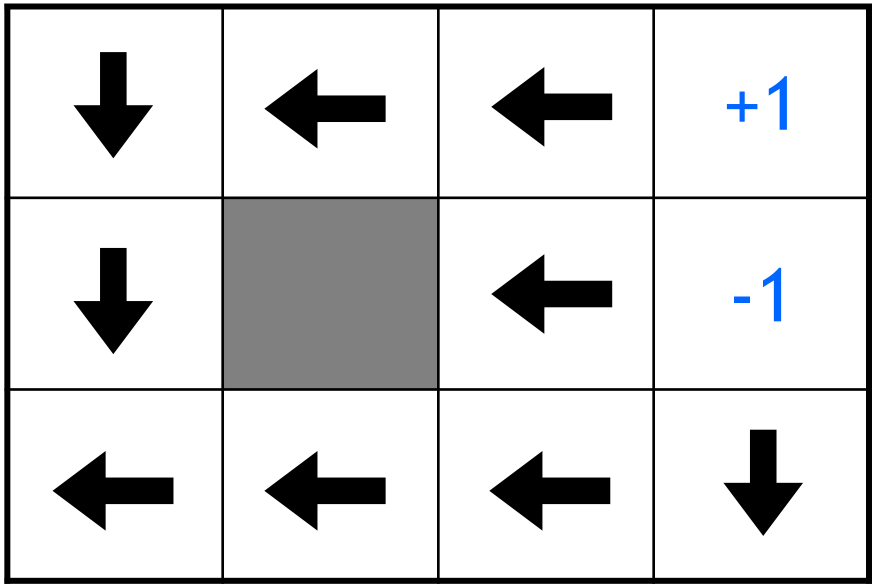
# Reward for each step: -0.04



# Reward for each step: -0.01

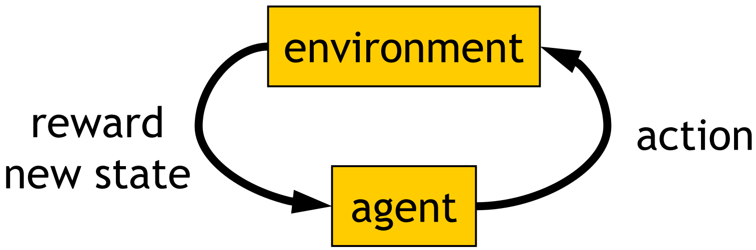


# Reward for each step: +0.01



# 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)

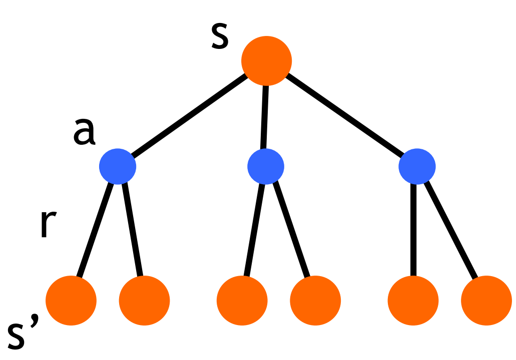
- reinforcement learning
  - transitions and rewards usually not available
  - how to change the policy based on experience
  - how to explore the environment

# 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 $\pi$
- state-action value function: $Q^\pi(s,a)$
  - expected return when starting in *s*, performing *a*, and following $\pi$
- 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

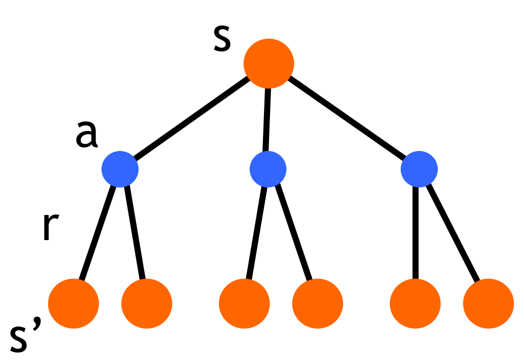
- there's a set of *optimal* policies
  - $V^\pi$ defines partial ordering on policies
  - they share the same optimal value function

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

- 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 $V^{\pi}$ from $\pi$
  - policy improvement: improve $\pi$ based on $V^{\pi}$

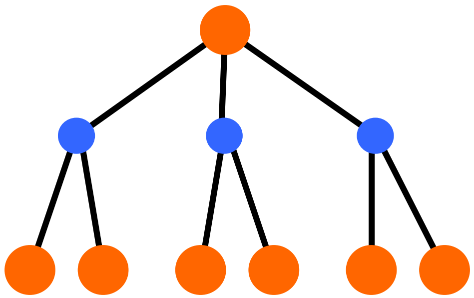
  - start with an arbitrary policy
  - repeat evaluation/improvement until convergence

# Policy evaluation/improvement

- policy evaluation: $\pi$ -> $V^\pi$
  - 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: $V^\pi$ -> $\pi$'

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

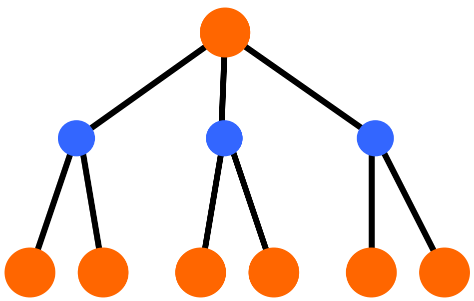
  - $\pi$' either strictly better than $\pi$, or $\pi$' is optimal (if $\pi$ = $\pi$')

# Policy/Value iteration

- Policy iteration

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

  - two nested iterations; too slow
  - don't need to converge to $V^{\pi_k}$
    - 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

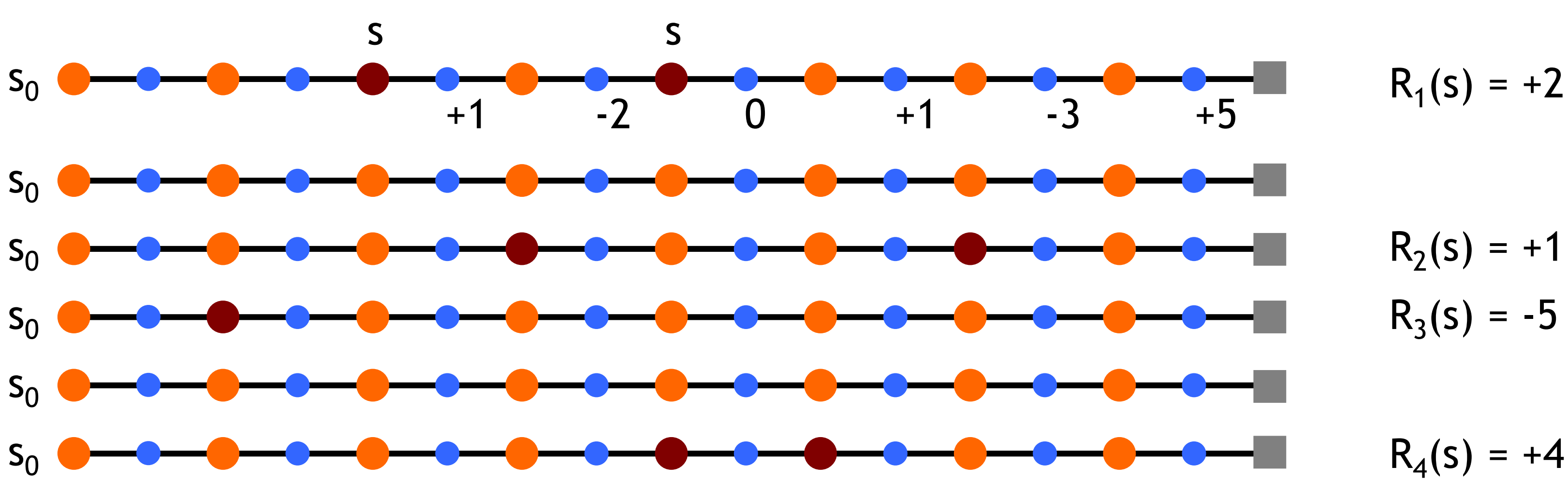
# Monte Carlo methods

- 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 $\pi$
  - estimate as average of observed returns in state s

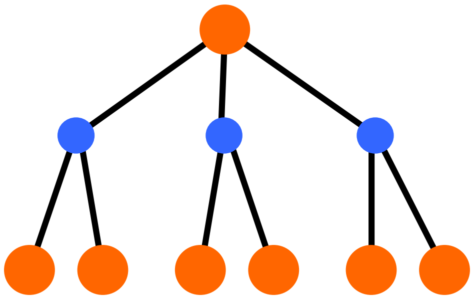
- first-visit MC
  - average returns following the first visit to state s




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

# Monte Carlo control

- $V^\pi$ 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

$$\pi_0 \longrightarrow^E Q^{\pi_0} \longrightarrow^I \pi_1 \longrightarrow^E Q^{\pi_1} \longrightarrow^I \dots \longrightarrow^I \pi^* \longrightarrow^E Q^*$$

  - 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

# Simulated experience

- 5-card draw poker
  - $s_0$: A♣, A♦, 6♠, A♥, 2♠
  - $a_0$: discard 6♠, 2♠
  - $s_1$: A♣, A♦, A♥, A♠, 9♠ + dealer takes 4 cards
  - return: +1 (probably)
- DP
  - list all states, actions, compute P(s,a,s')
    - P( [A♣,A♦,6♠,A♥,2♠], [6♠,2♠], [A♠,9♠,4] ) = 0.00192
- MC
  - all you need are sample episodes
  - let MC play against a random policy, or itself, or another algorithm

# 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

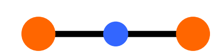
- combines ideas from MC and DP
  - like MC: learn directly from experience (don't need a model)
  - like DP: learn from values of successors
  - works for continuous tasks, usually faster than MC

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

$$V(s_t) \leftarrow V(s_t) + \alpha \left[R_t - V(s_t)\right]$$

target

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



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

# MC vs. TD

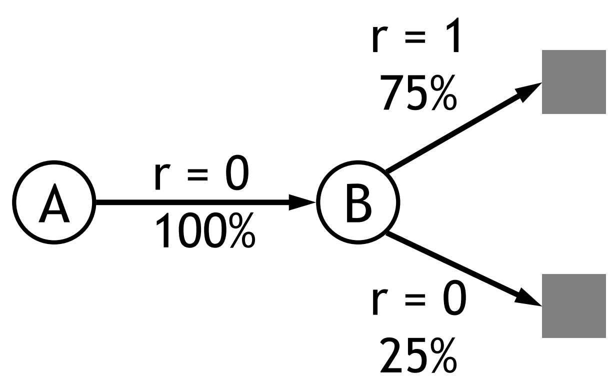
- observed the following 8 episodes:

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

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

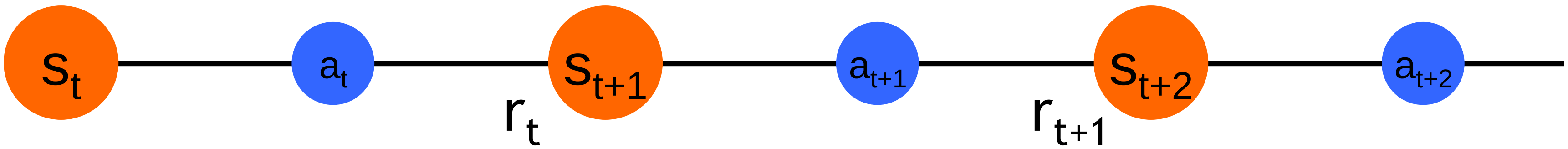
- MC: $V(A) = 0$
  – converges to values that minimize the error on training data

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



# Sarsa

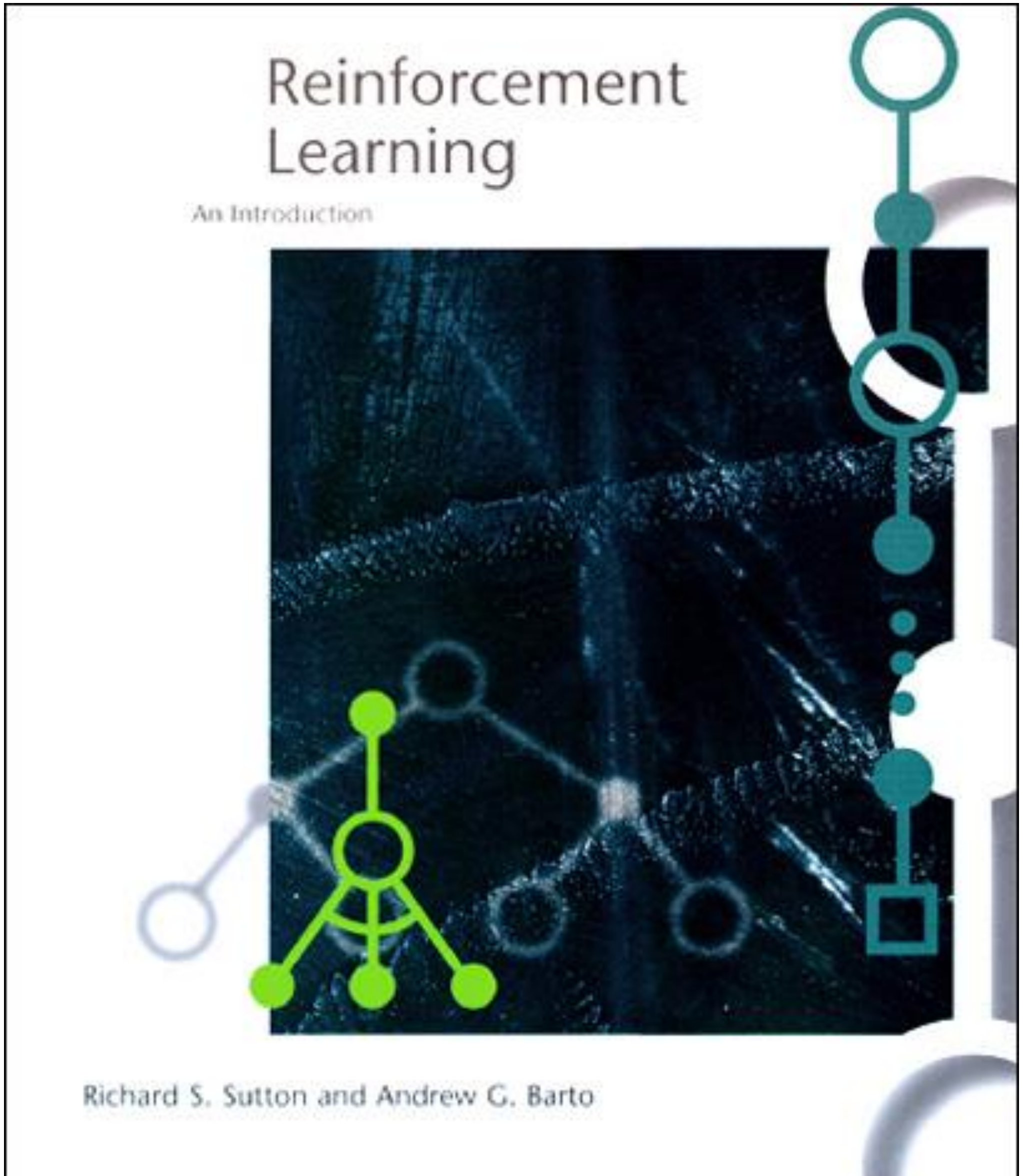
- again, need Q(s,a), not just V(s)




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

- control
  - start with a random policy
  - update Q and $\pi$ after each step
  - again, need $\varepsilon$-soft policies

# The RL Intro book




Richard Sutton, Andrew Barto
Reinforcement Learning,
An Introduction

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

# Backup slides

# 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

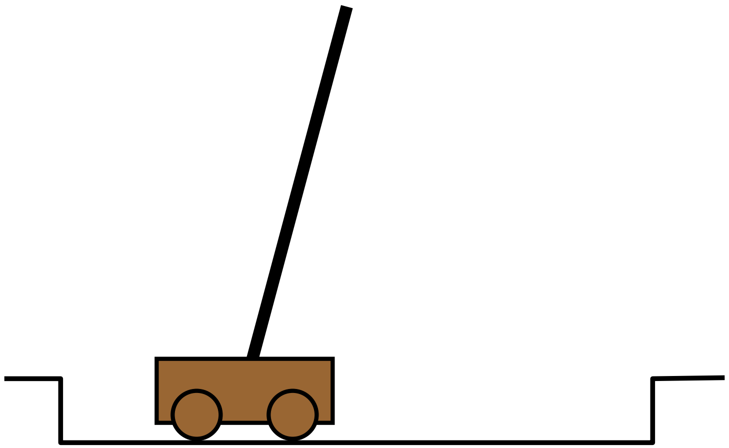
$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[ r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right]$$

# 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

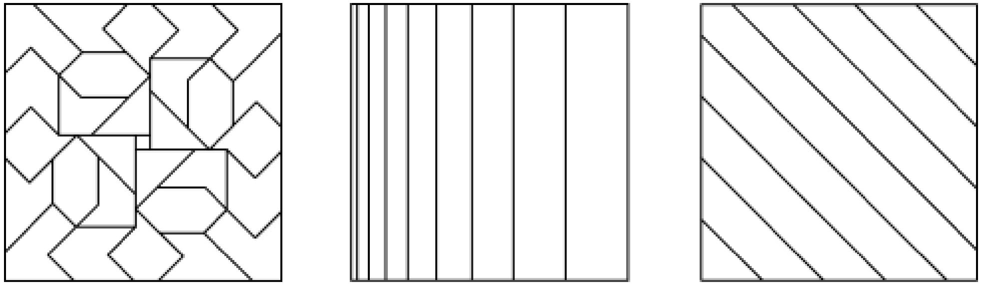
$$V(s_t) \leftarrow V(s_t) + \alpha \left[ r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \right]$$

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

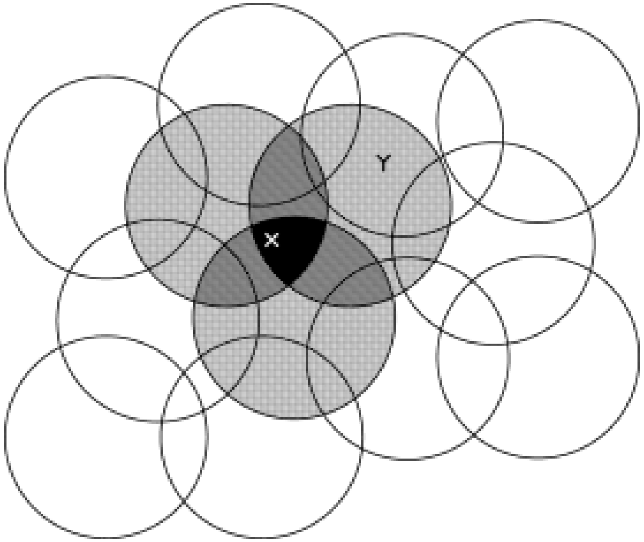
  - treat as one data point for regression
  - want method that can learn on-line (update after each step)

# Features

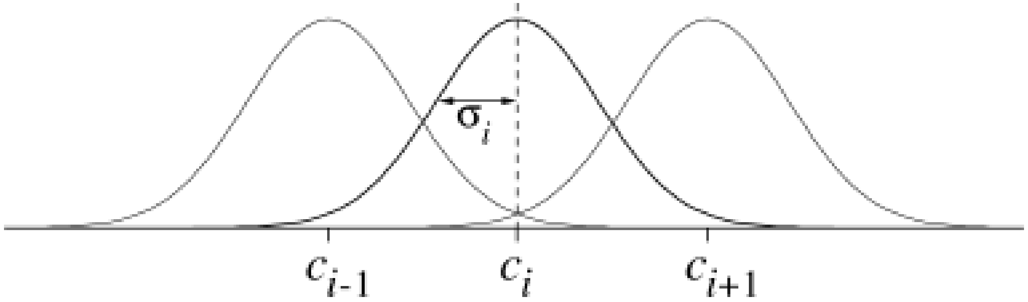
- tile coding, coarse coding
  - binary features



a) Irregular b) Log stripes c) Diagonal stripes




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

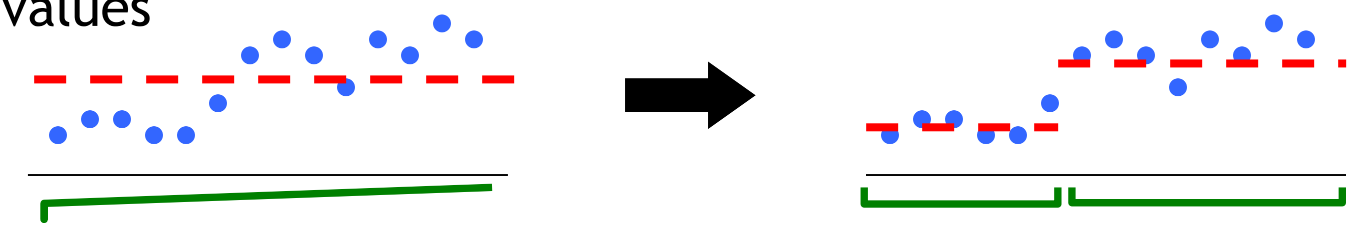



[ Sutton & Barto, Reinforcement Learning ]

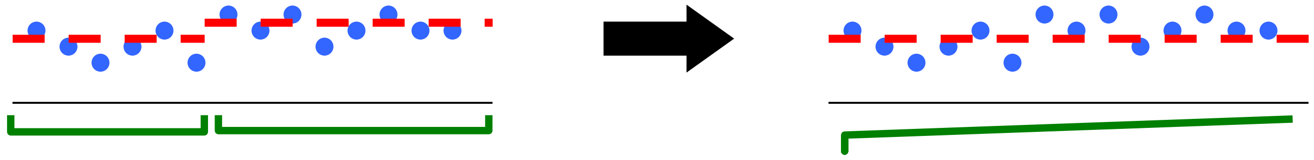
# Splitting and aggregation

- 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



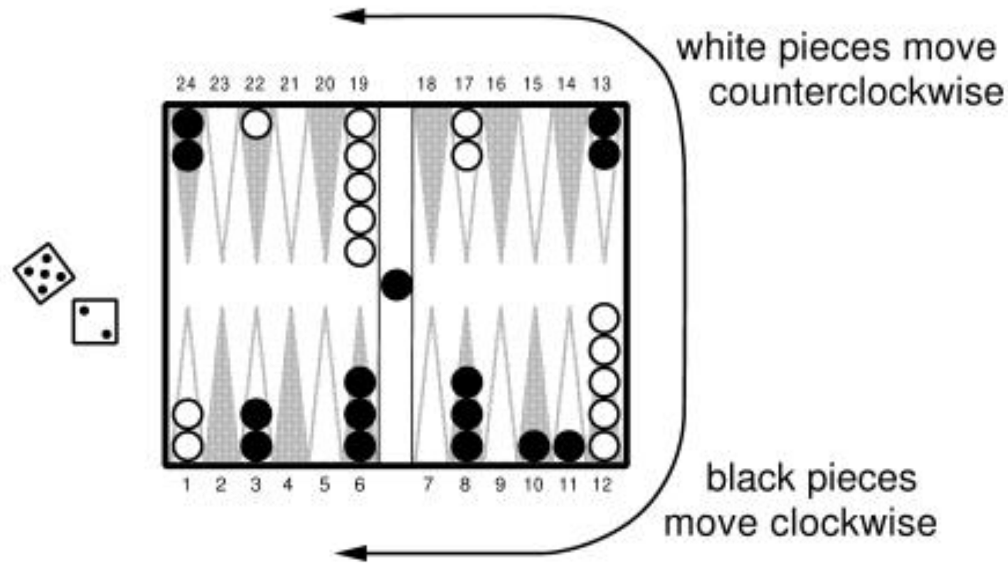
# Designing rewards

- 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($\lambda$) and neural nets
  - 4 binary features for each
  - 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