Machine Learning

10-701/15-781, Spring 2008

Reinforcement learning 2

Eric Xing

Lecture 28, April 30, 2008

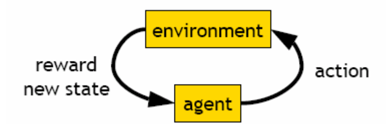
Reading: Chap. 13, T.M. book

Outline

- 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

Markov Decision Process (MDP)

- set of states S, set of actions A, initial state S0
- 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 $\pi(s,a)$
- reinforcement learning
  - transitions and rewards usually not available
  - how to change the policy based on experience
  - how to explore the environment

Dynamic programming

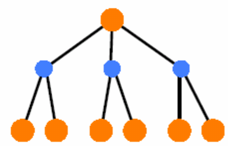
- Main idea
  - use value functions to structure the search for good policies
  - need a perfect model of the environment
- Two main components
  - policy evaluation: compute V^π from π
  - policy improvement: improve π based on V^π
  - start with an arbitrary policy
  - repeat evaluation/improvement until convergence

Policy/Value iration

- 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?
- DP bootstraps
  - updates estimates on the basis of other estimates

Passive learning

- The agent see the sequences of state transitions and associate rewards

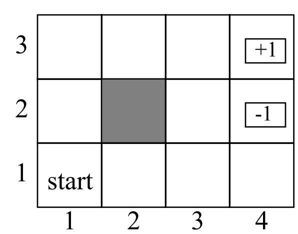
Epochs = training sequences:

(1,1)→(1,2)→(1,3)→(1,2)→(1,3)→(1,2)→(1,1)→(1,2)→(2,2)→(3,2) –1
(1,1)→(1,2)→(1,3)→(2,3)→(2,2)→(2,3)→(3,3) +1
(1,1)→(1,2)→(1,1)→(1,2)→(1,1)→(2,1)→(2,2)→(2,3)→(3,3) +1
(1,1)→(1,2)→(2,2)→(1,2)→(1,3)→(2,3)→(1,3)→(2,3)→(3,3) +1
(1,1)→(2,1)→(2,2)→(2,1)→(1,1)→(1,2)→(1,3)→(2,3)→(2,2)→(3,2) -1
(1,1)→(2,1)→(1,1)→(1,2)→(2,2)→(3,2) -1

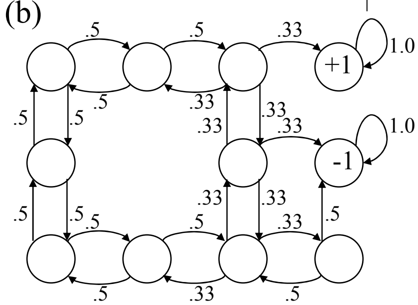
- Key idea: updating the utility value using the given training sequences.

Passive learning …

(a)



(b)



(c)

	1	2	3	4
3	-0.0380	0.0886	0.2152	+1
2	-0.1646		-0.4430	-1
1	-0.2911	-0.0380	-0.5443	-0.7722

(a) A simple stochastic environment.

(b) Each state transitions to a neighboring state with equal probability among all neighboring states. State (4,2) is terminal with reward –1, and state (4,3) is terminal with reward +1.

(c) The exact utility values.

LMS – updating
[Widrow & Hoff 1960]

function LMS-UPDATE(*V*,*e*,*percepts*,*M*,*N*) **returns** an update *V*

if TERMINAL?[*e*] then *reward-to-go* ← 0
for each e_i in *percepts* (starting at end) **do**
reward-to-go ← *reward-to-go* + REWARD[e_i]
V[STATE[e_i]] ← RUNNING-AVERAGE (*V*[STATE[e_i]], *reward-to-go*, *N*[STATE[e_i]])
end

simple average
batch mode

Average reward-to-go that state has gotten

- ***Reward to go*** of a state the sum of the rewards from that state until a terminal state is reached
- Key: use observed ***reward to go*** of the state as the direct evidence of the actual expected utility of that state
- Learning utility function directly from sequence example

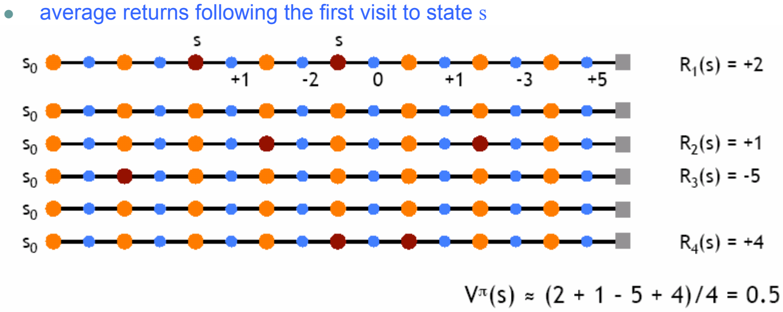
Monte Carlo methods

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

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



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

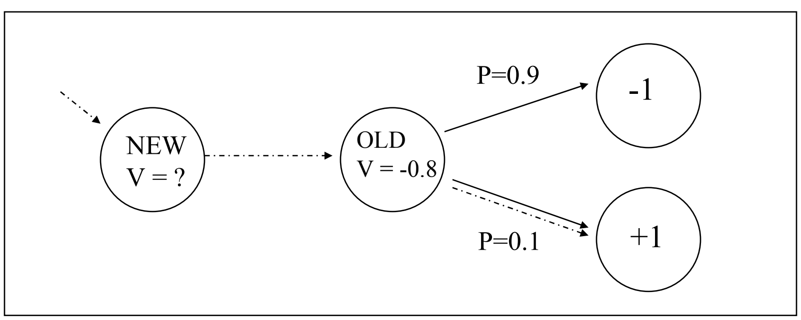
Simulated experience

- 5-card draw poker
  - s0: A♣, A♦, 6♠, A♥, 2♠
  - a0: discard 6♠, 2♠
  - s1: 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
- Learn from simulated experience
- Can concentrate on "important" states
  - don't need a full sweep
- No bootstrapping
  - less harmed by violation of Markov property
- Need to maintain exploration
  - use soft policies

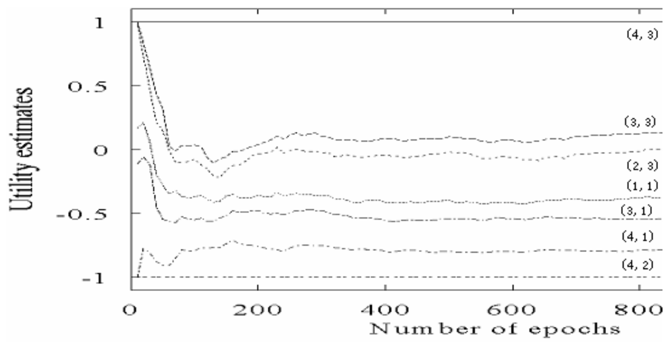
Utilities of states are not independent!



An example where MC and LMS does poorly. A new state is reached for the first time, and then follows the path marked by the dashed lines, reaching a terminal state with reward +1.

LMS updating algorithm in passive learning

- Drawback:
  - The actual utility of a state is constrained to be probability- weighted average of its successor's utilities.
  - Converge very slowly to correct utilities values (requires a lot of sequences)
    - *for our example, >1000!*



Temporal Difference Learning

- Combines ideas from MC and DP
  - like MC: learn directly from experience (don't need a model)
  - like DP: bootstrap
  - works for continuous tasks, usually faster then MC
- Constant-alpha MC:
  - have to wait until the end of episode to update

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

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

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

TD in passive learning

- TD(0) key idea:
  - adjust the estimated utility value of the current state based on its immediately reward and the estimated value of the next state.
- The updating rule

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

  - α is the learning rate parameter
  - Only when α is a function that decreases as the number of times a state has been visited increased, then can *V(s)* converge to the correct value.

Algorithm TD(λ)
(not in Russell & Norvig book)

- Idea: update from the whole epoch, not just on state transition.

$$V(s_t) \leftarrow V(s_t) + \alpha \sum_{t=k}^{\infty} \lambda^{t-k}[r_{t+1} + V(s_{t+1}) - V(s_t)]$$

  - Special cases:

    λ=1: LMS

    λ=0: TD
- Intermediate choice of λ (between 0 and 1) is best.
- Interplay with α …

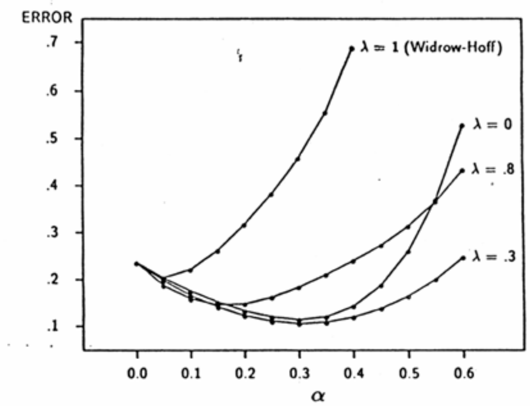
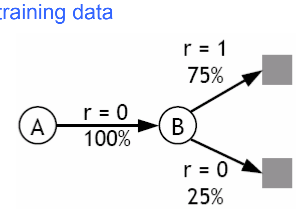


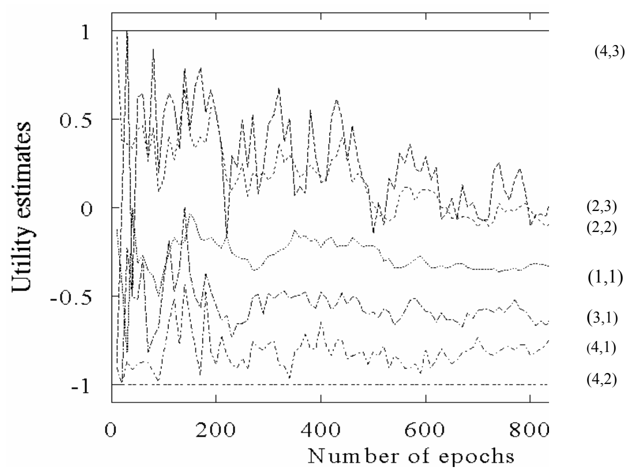
Figure 4. Average error on random walk problem after experiencing 10 sequences. All data are from TD(λ) with different values of α and λ. The dependent measure is the RMS error between the ideal predictions and those found by the learning procedure after a single presentation of a training set. This measure was averaged over 100 training sets. The $\lambda = 1$ data points represent performances of the Widrow-Hoff supervised-learning procedure.

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



The TD learning curve



Adaptive dynamic programming(ADP) in passive learning

- Different with LMS and TD method (model free approaches)
- ADP is a model based approach!
- The updating rule for passive learning

$$V^*(s) = \max_{a \in A} \left(R(s,a) + \gamma \sum_{s' \in S} P_{sa}(s') V^*(s') \right)$$

- However, in an unknown environment, P is not given, the agent must learn P itself by experiences with the environment.
- How to learn P?

Active learning

- An active agent must consider
  - what actions to take?
  - what their outcomes maybe (both on learning and receiving the rewards in the long run)?

- Update utility equation

$$V^*(s) = \max_{a \in A} \left(R(s,a) + \gamma \sum_{s' \in S} P_{sa}(s') V^*(s') \right)$$

- Rule to chose action

$$a^*(s) = \arg\max_{a \in A} \left(R(s,a) + \gamma \sum_{s' \in S} P_{sa}(s') V^*(s') \right)$$

Active ADP algorithm

Initialize *s* to current state that is perceived

Loop forever

{

Select an action ***a*** and execute it (using current model ***R*** and ***P***) using

$$a^*(s) = \arg\max_{a \in A} \left(R(s,a) + \gamma \sum_{s' \in S} P_{sa}(s') V^*(s') \right)$$

Receive immediate reward ***r*** and observe the new state ***s'***

Using the transition tuple <*s,a,s',r*> to update model ***R*** and ***P*** (see further)

For all the sate ***s***, update ***V(s)*** using the updating rule

$$V^*(s) = \max_{a \in A} \left(R(s,a) + \gamma \sum_{s' \in S} P_{sa}(s') V^*(s') \right)$$

s = s'

}

How to learn model?

- Use the transition tuple <s, a, s', r> to learn T(s,a,s') and R(s,a). That's supervised learning!
  - Since the agent can get every transition (s, a, s',r) directly, so take (s,a)/s' as an input/output example of the transition probability function *T*.
  - Different techniques in the supervised learning (see further reading for detail)
  - Use *r* and *P(s,a,s')* to learn *R(s,a)*

$$R(s,a) = \sum_{s' \in S} P_{s,a}(s') r(s',a)$$

ADP approach pros and cons

- Pros:
  - ADP algorithm converges far faster than LMS and Temporal learning. That is because it use the information from the the model of the environment.
- Cons:
  - Intractable for large state space
  - In each step, update *U* for all states
  - Improve this by *prioritized-sweeping* (see further reading for detail)

Another model free method– TD-Q learning

- Define Q-value function

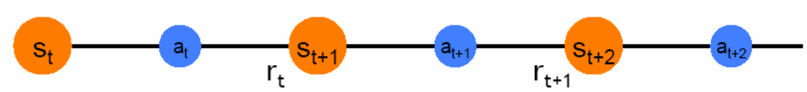
$$V(s) = \max_a Q(s,a)$$

- Q-value function updating rule
  - See subsequent slides
- Key idea of TD-Q learning
  - Combined with temporal difference approach
- Rule to chose the action to take

$$a = \arg\max_a Q(s,a)$$

Sarsa

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



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

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

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

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

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

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

TD-Q learning agent algorithm

For each pair (*s, a*), initialize *Q(s,a)*

Observe the current state s

Loop forever

{

Select an action ***a*** (optionally with ε-exploration) and execute it

$$a = \arg\max_a Q(s, a)$$

Receive immediate reward ***r*** and observe the new state ***s'***

Update *Q(s,a)*

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r_{t+1} + \gamma \max_{a'} Q(s', a') - Q(s, a)]$$

s=s'

}

Exploration

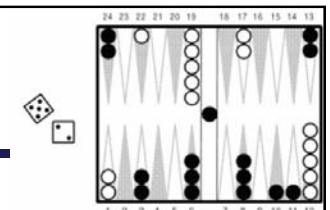
- Tradeoff between exploitation (control) and exploration (identification)
- Extremes: greedy vs. random acting

  (n-armed bandit models)

Q-learning converges to optimal Q-values if

- Every state is visited infinitely often (due to exploration),
- The action selection becomes greedy as time approaches infinity, and
- The learning rate a is decreased fast enough but not too fast (as we discussed in TD learning)

A Success Story



- TD Gammon (Tesauro, G., 1992)

  - A Backgammon playing program.

  - Application of temporal difference learning.

  - The basic learner is a neural network.

  - It trained itself to the world class level by playing against itself and learning from the outcome. So smart!!

  - More information: http://www.research.ibm.com/massive/tdl.html

Pole-balancing

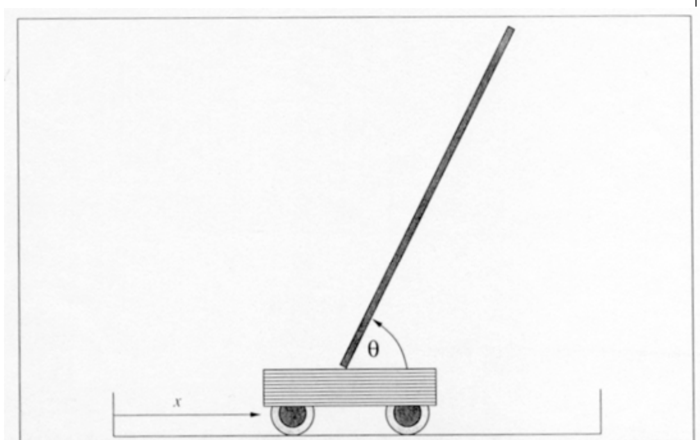
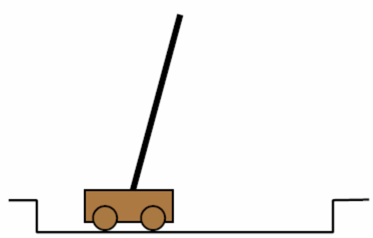


Figure 20.14 Setup for the problem of balancing a long pole on top of a moving cart. The cart can be jerked left or right by a controller that observes x, θ, $\dot{x}$, and $\dot{\theta}$.

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

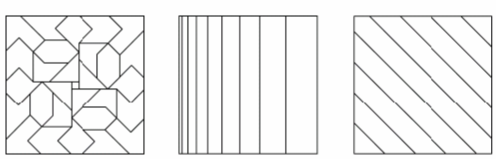
$$V(s_t) \leftarrow V(s_t) + \alpha \left[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)\right]$$

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

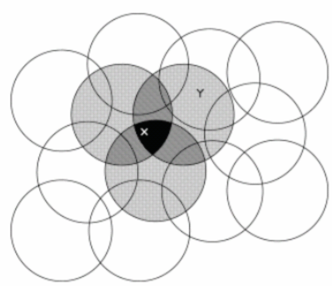
  - treat as one data point for regression
  - want method that can learn on-line (update after each step)

Features

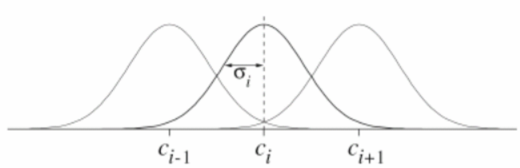
- tile coding, coarse coding
  - binary features



a) Irregular b) Log stripes c) Diagonal stripes



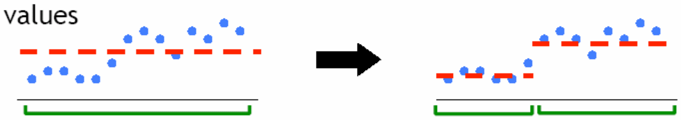
- radial basis functions
  - typically a Gaussian
  - between 0 and 1



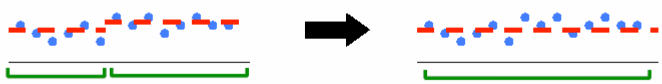
[Sutton & Barto, Reinforcement Learning]

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



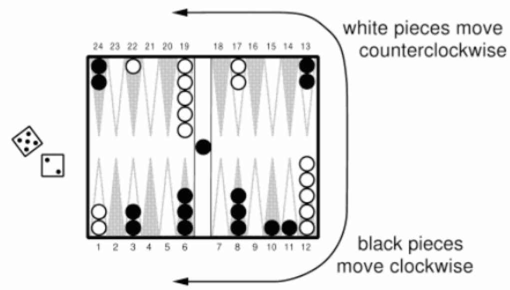
Designing rewards

- 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
  - 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

Future research in RL

- Function approximation (& convergence results)
- On-line experience vs. simulated experience
- Amount of search in action selection
- Exploration method (safe?)
- Kind of backups
  - Full (DP) vs. sample backups (TD)
  - Shallow (Monte Carlo) vs. deep (exhaustive)
    - λ controls this in TD(λ)
- Macros
  - Advantages
    - Reduce complexity of learning by learning subgoals (macros) first
    - Can be learned by TD(λ)
  - Problems
    - Selection of macro action
    - Learn models of macro actions (predict their outcome)
    - How do you come up with subgoals