

# 12: Monte Carlo Methods (Ch. 5)

---

- Last handout was about Dynamic Programming
  - An efficient way to compute value functions, and hence optimal policies for MDPs
  - Update value estimates using backups:

$$V(s) \leftarrow \sum P[R + \gamma V(s')]$$

- Problem: to do backup we need full access to
  - $P$ : the probability distribution over successor states
  - $R$ : the probability distribution over rewards
  - we often don't have them

# MC Methods

---

- This handout: MC methods
  - A new way to compute value functions
  - Usually less efficient than DP
  - But we only need *samples* from  $P$  and  $R$ , not distributions
  - MC is heavily used in some areas
  - Not used much in RL
    - We look at them for comparison with DP (ch. 4) and TD methods (ch. 6)

# Full Models

---

- We defined an RL problem as an MDP
  - Set of states  $S$ , set of actions  $A$ , transition probabilities  $P$ , reward probabilities  $R$
- DP uses  $P$  and  $R$  to do backups
  - If the learner knows  $P$  and  $R$  we say it has a *full model*
  - It knows everything there is to know about  $P$  and  $R$
  - Mathematically:  $P(s,a,s') \rightarrow$  probability

# Sample Models

---

- Sometimes we have only a *sample model*
  - A function which samples *one* successor state
    - Mathematically:  $F(s,a) \rightarrow s'$
    - Not a distribution over successors (which  $P$  has)
  - A function which samples one reward
    - Not a distribution over rewards (which  $R$  has)
- E.g. a chess playing program
  - It returns one successor state each time we move
- Sometimes we can reset the sample model to any state
  - But sometimes we can't
  - Most chess programs probably only play the current state or restart at the start state (although we could write a chess program that accepts arbitrary states)

# The Real World

---

- ❑ Sometimes we run RL in the real world
  - E.g. on a real robot
- ❑ This is a kind of sample model
  - We get one successor state each time we move
  - May or may not be able to reset
- ❑ Often RL is trained on a simulator (fast/easy to reset)
- ❑ Learned policy is then transferred to real robot
- ❑ Learning continues on real robot (slow)
- ❑ Robot has to adapt to difference between simulator and real world

# Models

---

- ❑ Surprisingly often we have a sample model but not a full model. E.g. playing chess:
  - It's easy to generate successor states using a chess simulator
  - It's harder to specify the probability of each successor state
- ❑ We can always use a full model as a sample model
- ❑ We can learn a full model from a sample model
  - Revisit a state/action many times and keep statistics
  - But this takes time
  - Especially if the simulation cannot be reset to arbitrary state/actions

# Monte Carlo Methods

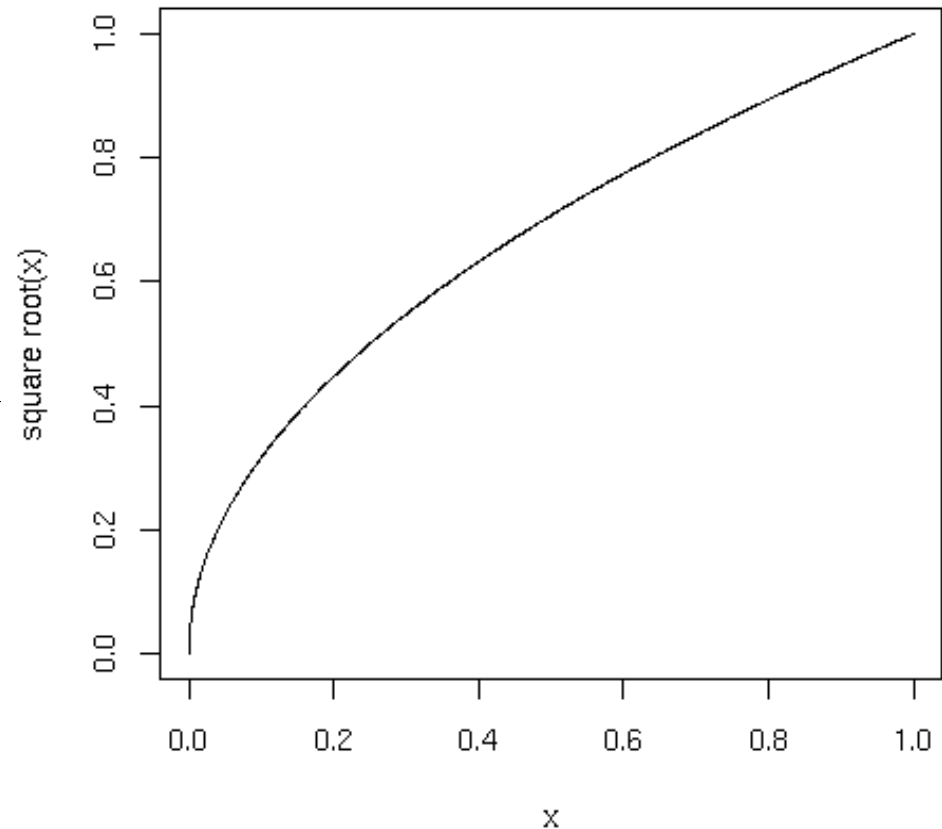
---

- Monte Carlo methods learn directly from experience
  - *On-line*: Learn from sample experience and still attain optimality
  - *Simulated*: No need for a *full* model
- Called *model-free RL* because they don't need a model
  - Can learn from real-world experience
- MC: a big subject
  - We look at only a small part
  - Basically, MC means “averaging output of a non-deterministic process”
  - Called 'Monte Carlo Methods' due to games of chance (card games, roulette etc.) in famous casinos in the city of Monte Carlo

# Monte Carlo

---

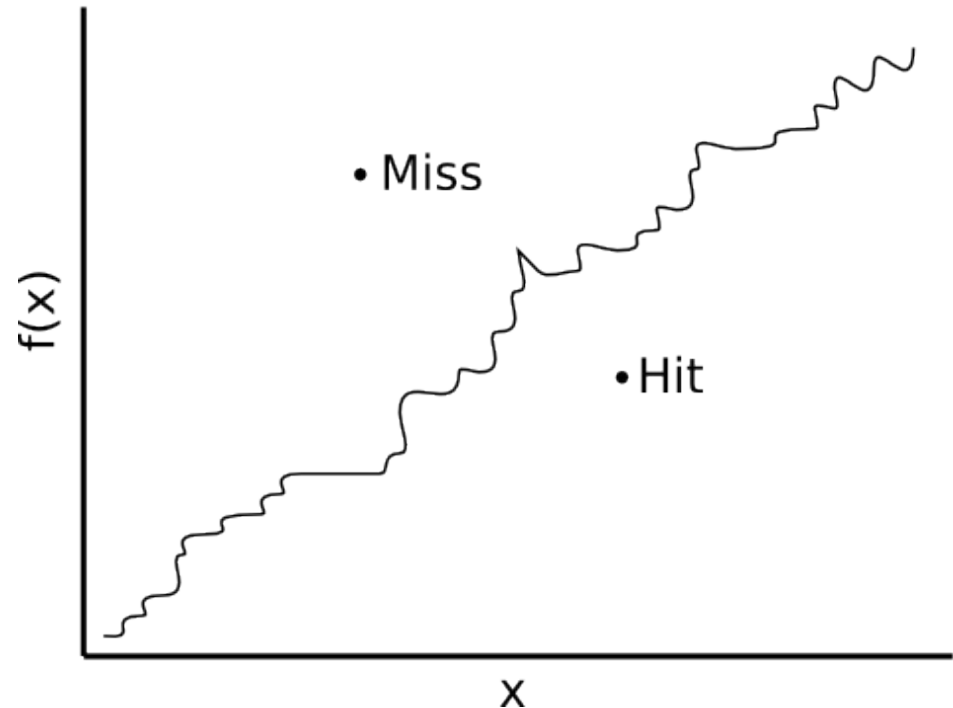
- Suppose we want to integrate a function  $f(x)$ 
  - I.e. find area under curve
- If the function is simple we can use calculus to get an exact answer
- The more complex the function the more difficult this is



# Integration by Monte Carlo

---

- ❑ MC can approximate the integral
- ❑ Imagine throwing darts at function while blindfolded
  - hits = misses = 0
  - Repeat N times
    - Generate a random  $x, y$  point
    - If  $f(x) > y$  hits ++
    - Else misses ++
  - Estimated integral = hits / (hits + misses)



❑ Bigger N = better estimate

# Monte Carlo for RL

---

- For RL, instead of a single point we generate an episode
  - A sequence of states, actions, rewards, successor states...
  - To estimate a state's value, we average (over many episodes) the return which follows the state
  - Return: some function of the sequence of rewards (ch. 3)
    - Usually the sum of discounted rewards
- Monte Carlo methods learn from *complete* sample returns
  - We have to wait till end of episode to find total return
    - So MC only works for episodic tasks
    - In continuing tasks, the return sequence never ends!
  - No backup from successor state; just the return
    - MC does not bootstrap one estimate from another

# Evaluation then Improvement

---

## □ First we cover *Policy Evaluation*

- Given a policy  $\pi$ , what is  $\pi$ 's value function  $V^\pi$ ?
- Also called the *prediction problem* since we're predicting the value of each state

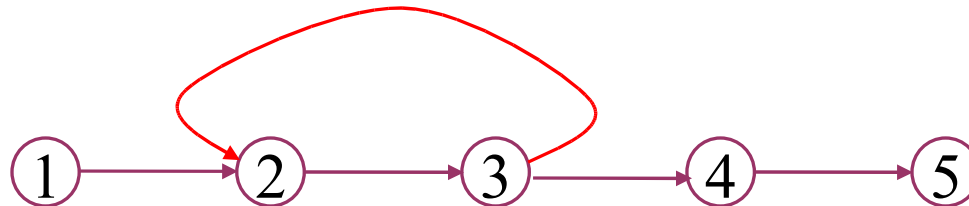
## □ Then we cover *Policy Improvement*

- Given  $V^\pi$ , can we get an improved policy  $\pi'$ ?
- Also called the *control problem* since we learn a new policy to control an agent
- Need to handle exploration

# Monte Carlo Policy Evaluation

---

- ❑ *Goal:* learn  $V^\pi(s)$
- ❑ *Given:* some number of episodes under  $\pi$  which contain  $s$
- ❑ *Idea:* Average returns observed after visits to  $s$



- ❑ *Every-Visit MC:* average returns for *every* time  $s$  is visited in an episode
- ❑ *First-visit MC:* average returns only for *first* time  $s$  is visited in an episode
- ❑ Both converge asymptotically to  $V^\pi(s)$

# First-visit Monte Carlo policy evaluation

---

Initialize:

$\pi$  = policy to be evaluated

$V$  = an arbitrary state-value function

$Returns(s)$  = a vector of empty lists, one for each state

Repeat Forever:

(a) Generate an episode using  $\pi$

(b) For each state  $s$  appearing in the episode:

$R$  = return following the first occurrence of  $s$

Append  $R$  to  $Returns(s)$

$V(s) = \text{average}(Returns(s))$

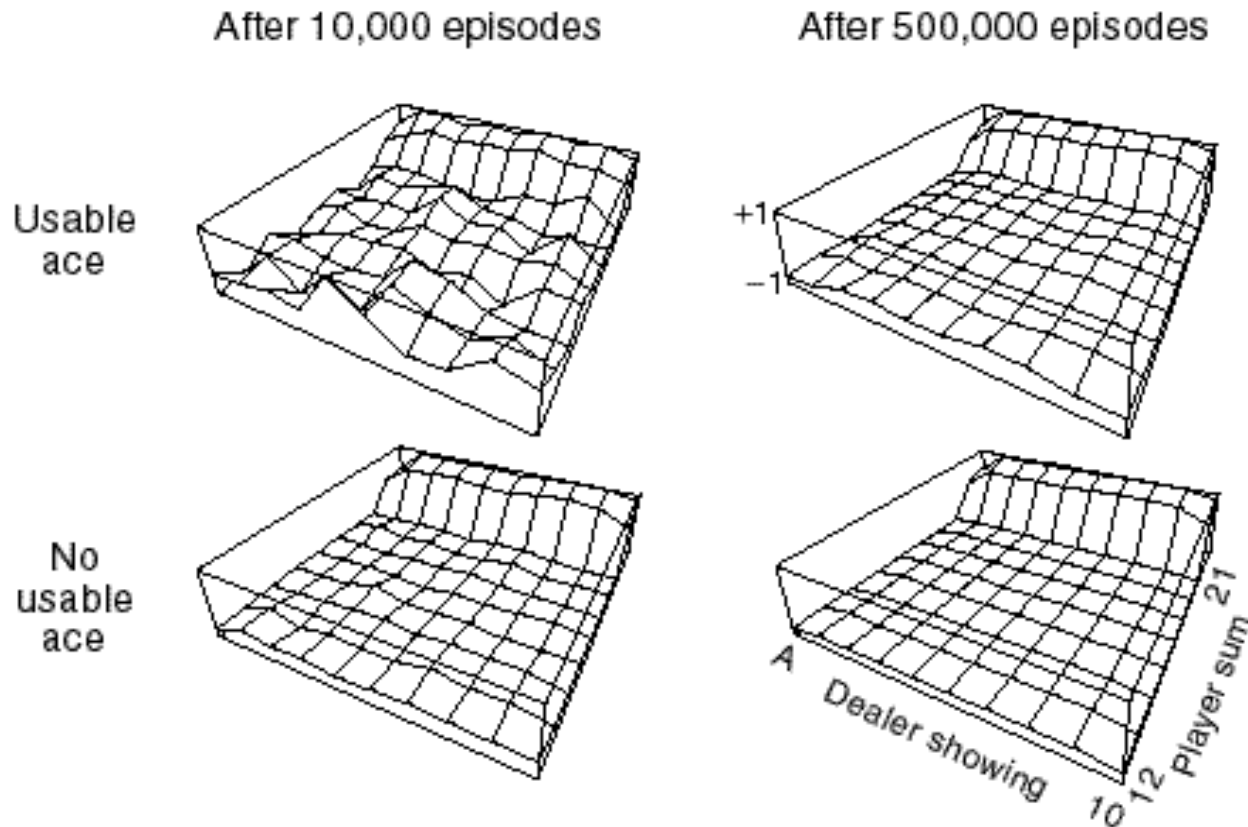
# Blackjack example

---

- ❑ *A card game*: many variations. Here: 1 player, 1 dealer
  - Dealer gets 2 cards. One is visible the other not
- ❑ *Object*: Have your card sum be greater than the dealer's without exceeding 21.
- ❑ *States* (200 of them) and 3 dimensions
  - current sum (12-21)
  - dealer's showing card (ace-10)
  - do I have a useable ace? (Ace counts as either 1 or 11)
- ❑ *Reward*: +1 for winning, 0 for a draw, -1 for losing
- ❑ *Actions*: stick (stop receiving cards), hit (receive another card)
- ❑ *Policy*: Stick if my sum is 20 or 21, else hit



# Blackjack value functions

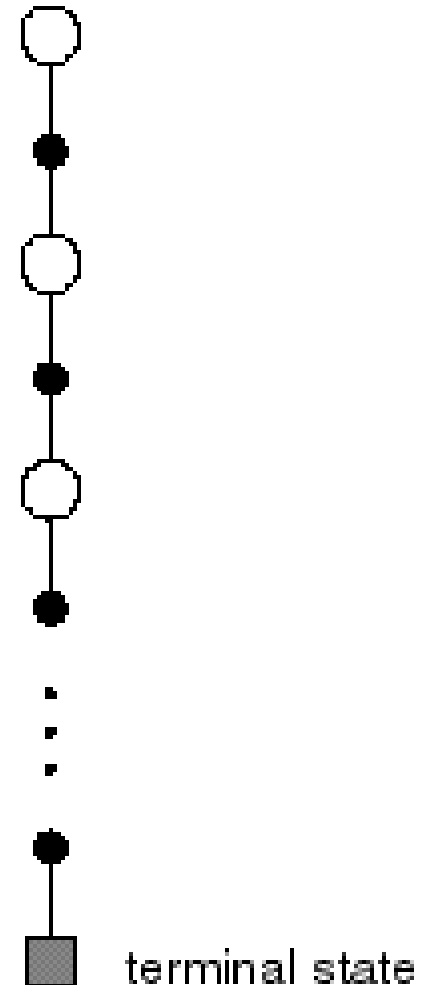


- 3-dimensional state plus 4<sup>th</sup> dimension for value needs 2 plots
  - left pair and right pair after different number of episodes
  - many episodes needed for good approximation
  - but this can be done quickly (episodes are very short)

# Backup diagram for Monte Carlo

---

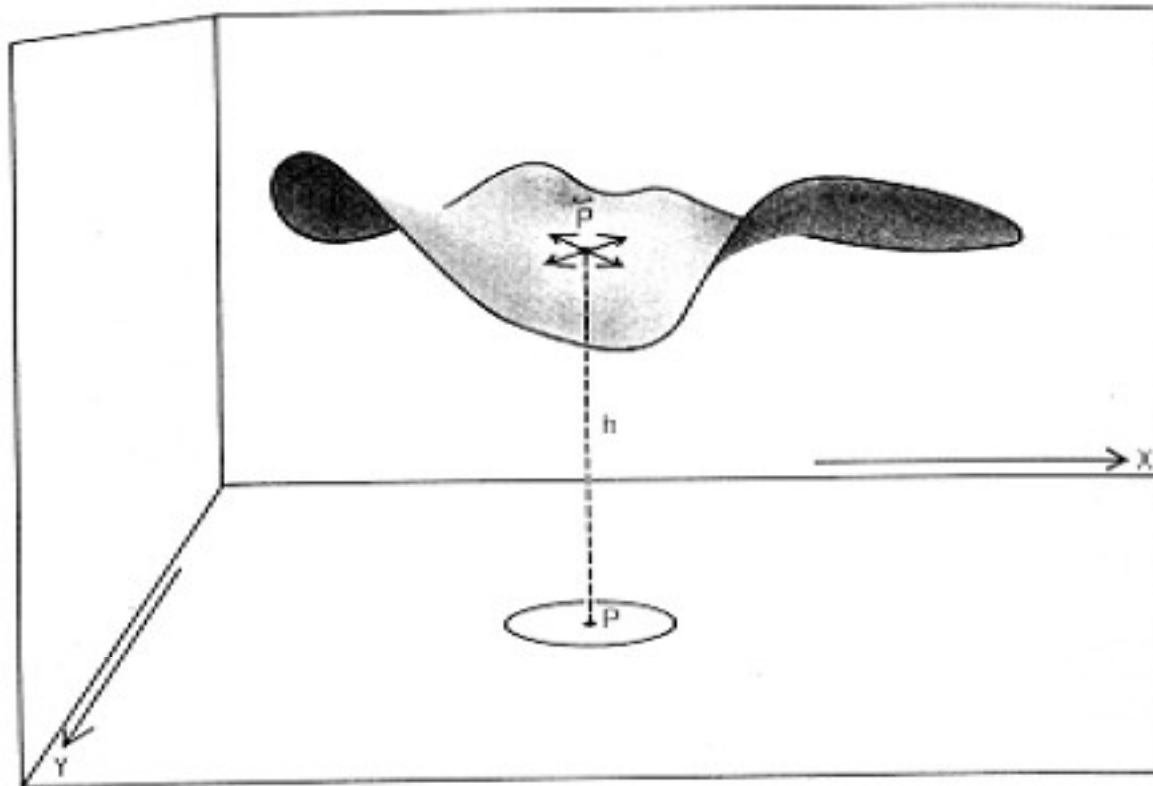
- ❑ Entire episode included
- ❑ Only one action considered at each state (unlike DP)
- ❑ MC does not bootstrap
  - So we can evaluate only the subset of states we're interested in (saves time).



# The Power of Monte Carlo

e.g., Elastic Membrane (Dirichlet Problem)

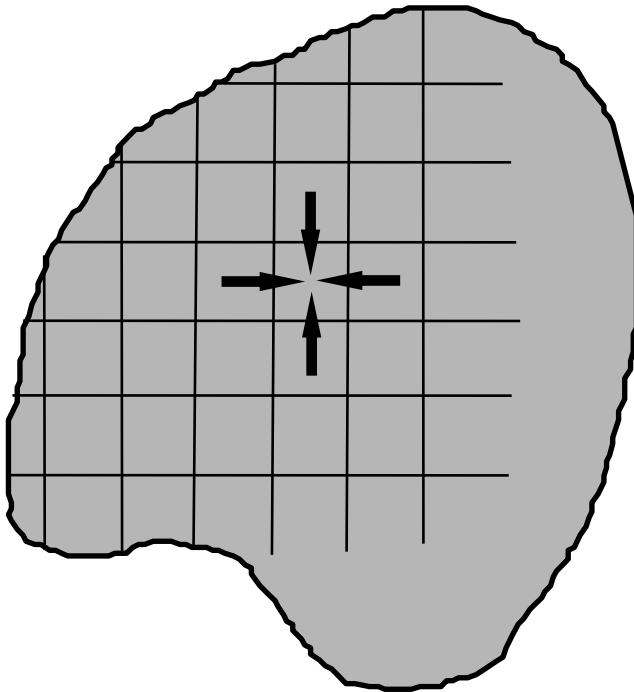
- How do we compute the height of the membrane/bubble?
- Very important in mathematical physics



# Relaxation Approach

---

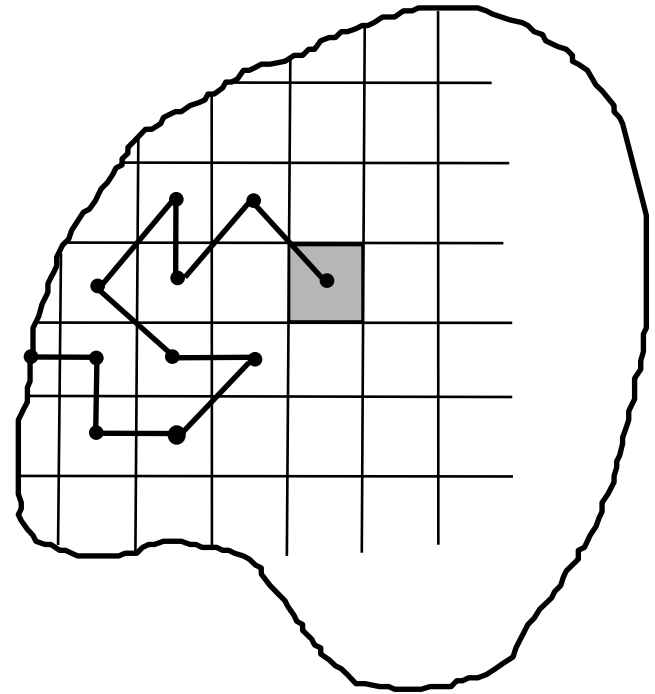
- Very much like DP
  - Height of wire frame is fixed.
    - Frame = terminal states
  - Put a grid over surface
  - Height of any cell is the average of its neighbours
  - Can be computed with iterative backups



# Kakutani's Algorithm (1945)

---

- A MC approach
  - Put the grid on the surface
  - Choose a start state
  - Do a random walk until you reach the frame
  - Average the heights of each point on the frame you reached
  - Converges to same value as the DP approach
- More efficient than DP if you only want to know the value of a few states



---

# Policy Improvement

# Monte Carlo Estimation of Action Values (Q)

---

- Monte Carlo is most useful when a full model is not available
  - If we had a full model we'd use DP
- With TTT (ch. 1) and DP (ch. 4) we learned state values  $V^\pi(s)$ 
  - We had a model to tell us what the successor states were
    - We looked up the successors and their values  $V^\pi(s')$
    - This told us the best action in  $s$
  - If we have no model, we can't do that...
  - Instead we learn action-values  $Q^\pi(s, a)$ 
    - now we don't need to know the successor states
    - because we already know the action values  $Q^\pi(s, a)$

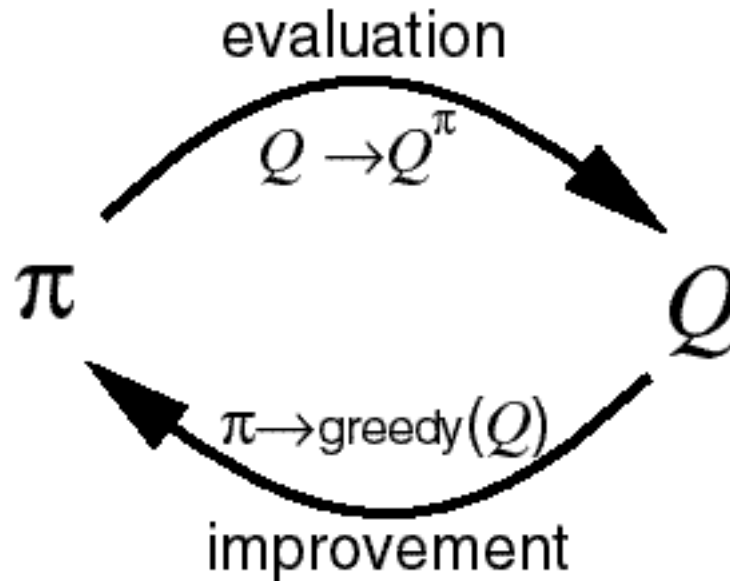
# Exploration

---

- ❑ To take the greedy policy we need the Q-value of each possible action  $Q^\pi(s, a)$
- ❑ Problem: some policies (e.g. deterministic ones) do not visit all state-action pairs
- ❑ One solution: *Exploring starts*.
  - Every state-action pair has a non-zero probability of being the starting pair in an episode
  - A simple solution
  - Not always possible (can we reset the problem to any state-action pair?)
  - Will try other solutions later: soft policies, off-policy learning

# Monte Carlo Control

---



- ❑ **MC policy iteration:** policy evaluation using MC methods followed by policy improvement
- ❑ **Policy improvement step:** greedify with respect to action-value function
- ❑ A form of GPI, just like DP

# Monte Carlo GPI

---

- ❑ MC evaluation converges to the true value function
  - ... given infinite visits to every state/action pair
- ❑ Of course then we never get around to improving!
- ❑ In practice, we do some evaluation then an improvement
  - Hard to say how much evaluation is optimal
    - Just like with DP evaluation & improvement
  - We can do 1 evaluation before each improvement
  - Or many
  - We can try to estimate error in policy evaluation and continue until we think it's low enough

# Monte Carlo Exploring Starts

---

Initialize, for all states and actions

$Q(s,a)$  = arbitrary

$\pi(s)$  = arbitrary

$Returns(s,a)$  = empty list

No known proof of convergence to  $\pi^*$

Repeat forever:

New parts underlined

(a) Generate an episode using exploring starts and  $\pi$

(b) For each pair  $s, \underline{a}$  appearing in the episode:

$R$  = return following the first occurrence of  $s, a$

Append  $R$  to  $Returns(s,a)$

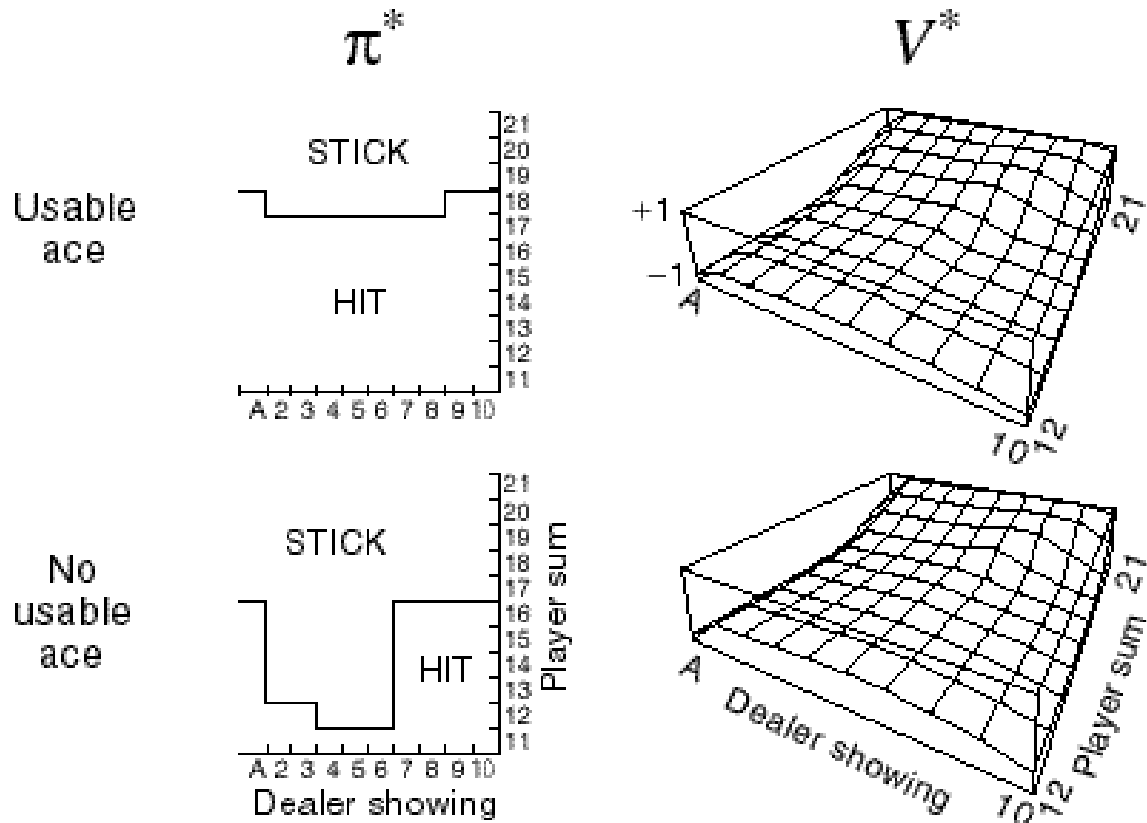
$Q(s,a)$  = average( $Returns(s,a)$ )

(c) For each  $s$  in the episode:

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

# Blackjack example continued

- Exploring starts
- Initial policy as described before
- Learned policy almost same as in book on Blackjack



# Soft Policy Monte Carlo Control

---

- How do we get rid of exploring starts?
  - One way: use *soft* policies:  $\pi(s,a) > 0$  for all  $s$  and  $a$
  - Soft policies take exploratory (non-greedy) actions
  - e.g.  $\epsilon$ -soft policy probabilities (see ch.2):

$$\frac{\epsilon}{|A(s)|} \qquad 1 - \epsilon + \frac{\epsilon}{|A(s)|}$$

non-greedy      greedy action

- Policy improvement:
  - Deterministic policies: we use the greedy action
  - For  $\epsilon$ -soft policies: move policy *towards* greedy policy (i.e. make it  $\epsilon$ -greedy)

# Soft Policy MC Control

---

Initialize, for all states and actions

$Q(s,a)$  = arbitrary

$Returns(s,a)$  = empty list

$\pi$  = an arbitrary soft policy

Repeat forever:

(a) Generate an episode using  $\pi$

(b) For each pair  $s,a$  appearing in the episode:

$R$  = return following the first occurrence of  $s,a$

Append  $R$  to  $Returns(s,a)$

$Q(s,a)$  = average( $Returns(s,a)$ )

(c) For each  $s$  in the episode:

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

For all actions in  $s$ :

$$\pi(s,a) = \begin{cases} 1 - \epsilon + \epsilon / |A(s)| & \text{if } a = a^* \\ \epsilon / |A(s)| & \text{if } a \neq a^* \end{cases}$$

New part

# Problem with Soft Policy Control

---

- ❑ Converges to best  $\epsilon$ -soft policy
  - Will not generate as much reward as deterministic greedy policy, since it takes exploratory actions
  - What if we want to stop exploring and maximise reward?
    - E.g. training phase over
  - We can reduce  $\epsilon$ , but that changes policy and hence value function
    - We have to learn a new value function for new policy
    - Takes time
  - Alternative: off-policy control

# Off-policy Monte Carlo control

---

- So far all the methods we've seen are *on-policy*
  - They learn the value function for the policy which selects actions
- *Off-policy*: Follow one policy and learn about another
  - *Behavior policy* generates behavior in environment
  - *Estimation policy* is policy being learned about
- Gives us another way to explore
  - Behaviour policy can be soft (so exploring starts not needed)
  - Estimation policy can be deterministic
    - so we can switch to it any time and maximise rewards

# Off-Policy Learning

---

How do we evaluate one policy following another?

- ❑ Weight returns from behavior policy by probability the same experience would occur using the estimation policy
- ❑ The higher this probability, the more we can learn about one policy while following the other
  - Details are in Sutton & Barto
- ❑ The prob. of generating the same experience is low if
  - the transition function is deterministic
  - the policies are very different
    - E.g. if they take different actions with prob. 1 they are guaranteed not to generate the same experience
  - TD methods (ch. 6) are more suitable for off-policy learning

# Summary

---

- ❑ MC has several advantages over DP:
  - No need for full models:
    - Can learn directly from interaction with real world
    - Can learn from sample models / simulations
  - No need to learn about ALL states
  - Less harmed by non-Markov states (later in book)
- ❑ MC methods provide an alternate policy evaluation process for GPI
- ❑ One issue to watch for: maintaining sufficient exploration
  - exploring starts, soft policies, off-policy learning
- ❑ No bootstrapping (unlike DP)