Reinforcement learning

Stuart Russell, UC Berkeley

Outline

◇ Sequential decision making

◇ Dynamic programming algorithms

◇ Reinforcement learning algorithms
– temporal difference learning for a fixed policy
– Q-learning and SARSA

◇ Function approximation

◇ Exploration

◇ Decomposing agents into modules

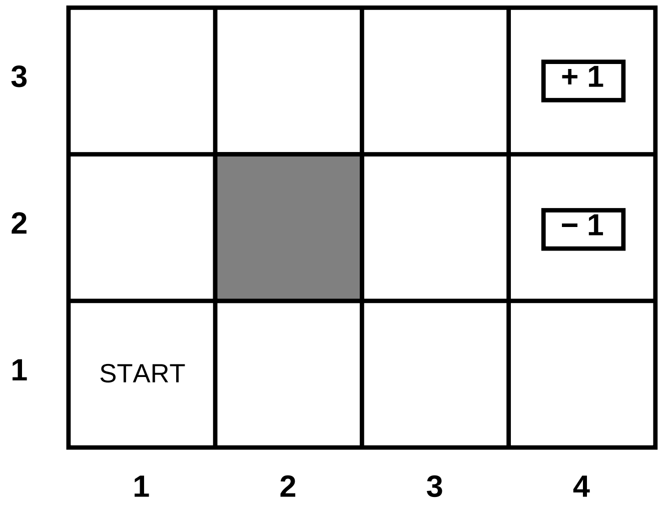
Sequential decision problems

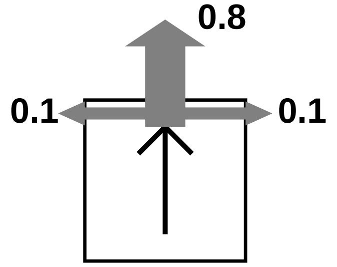
Add uncertainty to state-space search $\longrightarrow$ MDP

Add sequentiality to Bayesian decision making $\longrightarrow$ MDP
I.e., any environment in which rewards are not immediate

Examples:
- Tetris, spider solitaire
- Inventory and purchase decisions, call routing, logistics, etc. (OR)
- Elevator control
- Choosing insertion paths for flexible needles
- Motor control (stochastic optimal control)
- Robot navigation, foraging

Example MDP





States $s \in S$, actions $a \in A$

Model $T(s, a, s') \equiv P(s'|s, a)$ = probability that a in s leads to s'

Reward function $R(s)$ (or $R(s, a)$, $R(s, a, s')$)

$$= \begin{cases} -0.04 & \text{(small penalty) for nonterminal states} \\ \pm 1 & \text{for terminal states} \end{cases}$$

Solving MDPs

In search problems, aim is to find an optimal *sequence*

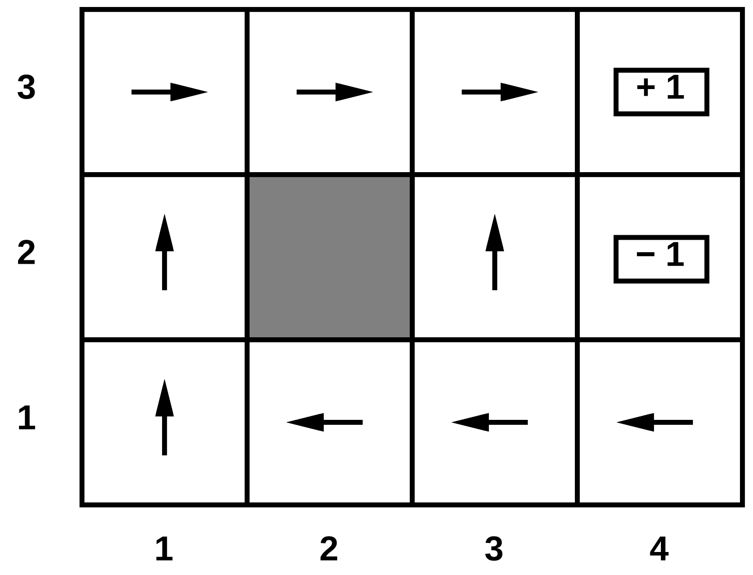
In MDPs, aim is to find an optimal *policy* $\pi^*(s)$

i.e., best action for every possible state s

(because can't predict where one will end up)

The optimal policy maximizes (say) the *expected sum of rewards*

Optimal policy when state penalty $R(s)$ is –0.04:



Utility of state sequences

Need to understand preferences between *sequences* of states

Typically consider stationary preferences on reward sequences:

$$[r, r_0, r_1, r_2, \ldots] \succ [r, r'_0, r'_1, r'_2, \ldots] \Leftrightarrow [r_0, r_1, r_2, \ldots] \succ [r'_0, r'_1, r'_2, \ldots]$$

Theorem (Koopmans, 1972): there is only one way to combine rewards over time:

– *Additive discounted* utility function:
  $U([s_0, s_1, s_2, \ldots]) = R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \cdots$
  where $\gamma \leq 1$ is the discount factor

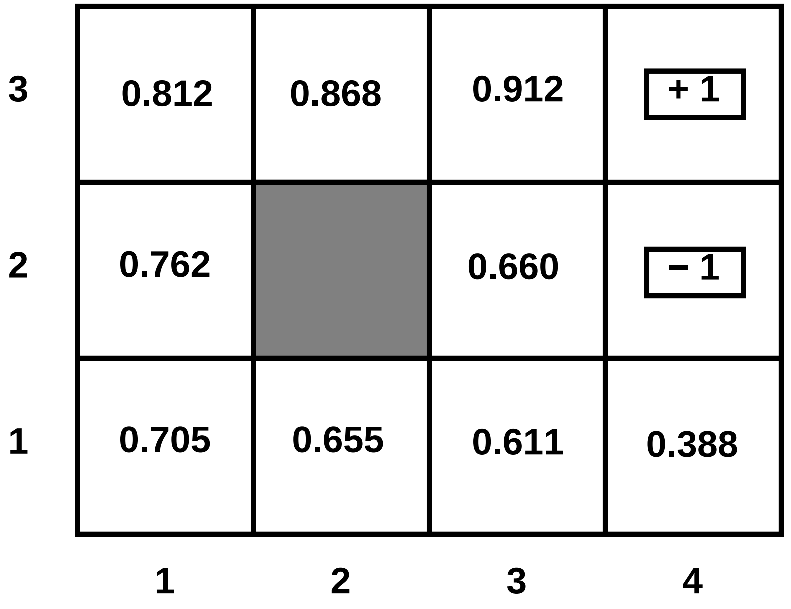
[Humans may beg to differ]

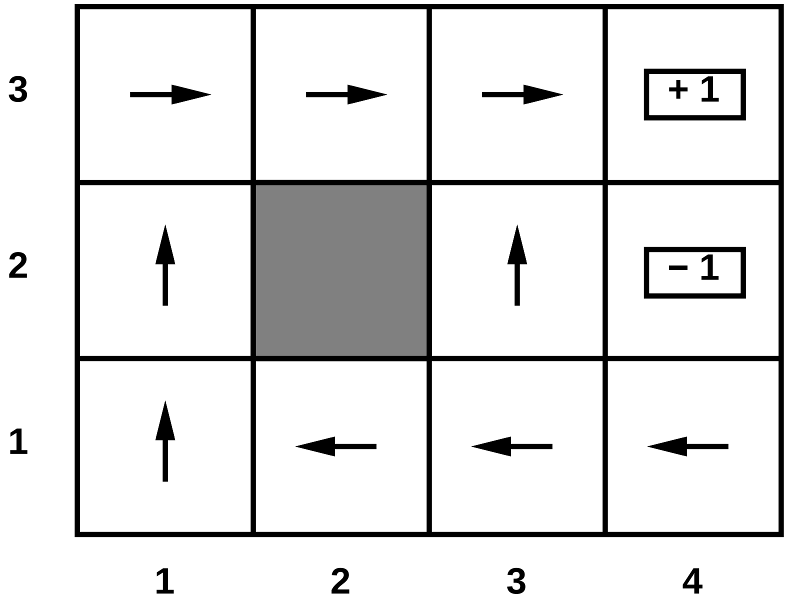
Utility of states

Utility of a *state* (a.k.a. its *value*) is defined to be

$U(s) =$ expected (discounted) sum of rewards (until termination)
assuming optimal actions

Given the utilities of the states, choosing the best action is just MEU:
maximize the expected utility of the immediate successors





Utilities contd.

Problem: infinite lifetimes $\Rightarrow$ undiscounted ($\gamma = 1$) utilities are infinite

1) Finite horizon: termination at a *fixed time* T

$\Rightarrow$ nonstationary policy: $\pi(s)$ depends on time left

2) Absorbing state(s): w/ prob. 1, agent eventually "dies" for any π

$\Rightarrow$ expected utility of every state is finite

3) Discounting: assuming $\gamma < 1$, $R(s) \leq R_{\max}$,

$$U([s_0, \ldots s_\infty]) = \Sigma_{t=0}^{\infty} \gamma^t R(s_t) \leq R_{\max}/(1-\gamma)$$

Smaller $\gamma \Rightarrow$ shorter horizon

4) Maximize average reward per time step

– sometimes more appropriate than discounting

Dynamic programming: the Bellman equation

Definition of utility of states leads to a simple relationship among utilities of neighboring states:

expected sum of rewards
$\quad =$ current reward
$\qquad + \gamma \times$ expected sum of rewards after taking best action

Bellman equation (1957) (also Shapley, 1953):

$$U(s) = R(s) + \gamma \max_a \Sigma_{s'} U(s')T(s,a,s')$$

$$\begin{array}{ll}
U(1,1) = -0.04 & \\
\quad + \gamma \max\{0.8U(1,2) + 0.1U(2,1) + 0.1U(1,1), & \textit{up} \\
\qquad\qquad 0.9U(1,1) + 0.1U(1,2) & \textit{left} \\
\qquad\qquad 0.9U(1,1) + 0.1U(2,1) & \textit{down} \\
\qquad\qquad 0.8U(2,1) + 0.1U(1,2) + 0.1U(1,1)\} & \textit{right}
\end{array}$$

One equation per state $= n$ **nonlinear** equations in n unknowns

Value iteration algorithm

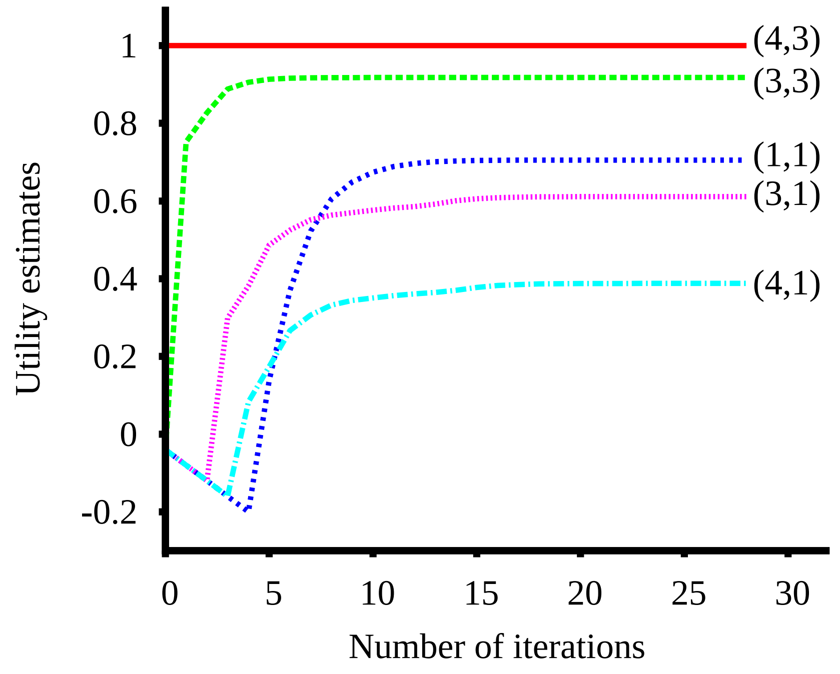
Idea: Start with arbitrary utility values

Update to make them **locally consistent** with Bellman eqn.

Everywhere locally consistent $\Rightarrow$ global optimality

Repeat for every s simultaneously until "no change"

$$U(s) \leftarrow R(s) + \gamma \max_a \Sigma_{s'} U(s')T(s,a,s') \qquad \text{for all } s$$



Convergence

Define the max-norm $||U|| = \max_s |U(s)|$,
so $||U - V||$ = maximum difference between U and V

Let U^t and U^{t+1} be successive approximations to the true utility U

Theorem: For any two approximations U^t and V^t

$$||U^{t+1} - V^{t+1}|| \leq \gamma \, ||U^t - V^t||$$

I.e., Bellman update is a **contraction**: any distinct approximations must get closer to each other
so, in particular, any approximation must get closer to the true U
and value iteration converges to a unique, stable, optimal solution

But MEU policy using U^t may be optimal long before convergence of values
. . .

Policy iteration

Howard, 1960: search for optimal policy and utility values simultaneously

Algorithm:

```
$\pi \leftarrow$ an arbitrary initial policy
repeat until no change in $\pi$
    compute utilities given $\pi$
    update $\pi$ as if utilities were correct (i.e., local depth-1 MEU)
```

To compute utilities given a fixed π (value determination):

$$U(s) = R(s) + \gamma \textstyle\sum_{s'} U(s') T(s, \pi(s), s') \qquad \text{for all } s$$

i.e., n simultaneous **linear** equations in n unknowns, solve in $O(n^3)$

Q-iteration

Define $Q(s,a)$ = expected value of doing action a in state s and then acting optimally thereafter

$= R(s) + \gamma \sum_{s'} U(s') T(s,a,s')$

i.e., $U(s) = \max_a Q(s,a)$

Q-iteration algorithm: like value iteration, but do

$$Q(s,a) \leftarrow R(s) + \gamma \sum_{s'} T(s,a,s') \max_{a'} Q(s',a') \qquad \text{for all } s, a$$

Q-values represent a policy with no need for transition model (unlike U)

General asynchronous dynamic programming

Local value and policy updates steps can be mixed in any order on any states, with convergence guaranteed as long as every state gets updated infinitely often

Reinforcement learning algorithms operate by performing such updates based on the observed transitions made in an initially unknown environment

Invariance transformations

Nonsequential behaviour invariant under positive affine transform:

$$U'(s) = k_1 U(s) + k_2 \quad \text{where } k_1 > 0$$

Sequential behaviour with additive utility is invariant
wrt addition of any potential-based reward:

$$R'(s,a,s') = R(s,a,s') + F(s,a,s')$$

where the added reward must satisfy

$$F(s,a,s') = \gamma\Phi(s') - \Phi(s)$$

for some potential function Φ on states [Ng, Harada, Russell, ICML 99]

Often useful to add "shaping rewards" to guide behavior

Partial observability

POMDP has an observation model $O(s, e)$ defining the probability that the agent obtains evidence e when in state s

Agent does not know which state it is in
$\Rightarrow$ makes no sense to talk about policy $\pi(s)$!!

Theorem (Astrom, 1965): the optimal policy in a POMDP is a function $\pi(b)$ where b is the belief state ($P(S|e_1, \ldots, e_t)$)

Can convert a POMDP into an MDP in (continuous, high-dimensional) belief-state space,
where $T(b, a, b')$ is essentially a filtering update step

Solutions automatically include information-gathering behavior

The real world is an unknown POMDP

Other Issues

Complexity: polytime in number of states (by linear programming)
  but number of states is exponential in number of state variables
  $\longrightarrow$ Boutilier *et al*, Parr & Koller: use structure of states
    (but U, Q summarize infinite sequences, depend on everything)
  $\longrightarrow$ reinforcement learning: sample S, approximate $U/Q/\pi$
  $\longrightarrow$ hierarchical methods for policy construction (next lecture)

Unknown transition model: agent cannot solve MDP w/o $T(s, a, s')$
  $\longrightarrow$ reinforcement learning

Missing state: there are state variables the agent doesn't know about
  $\longrightarrow$ [your ideas here]

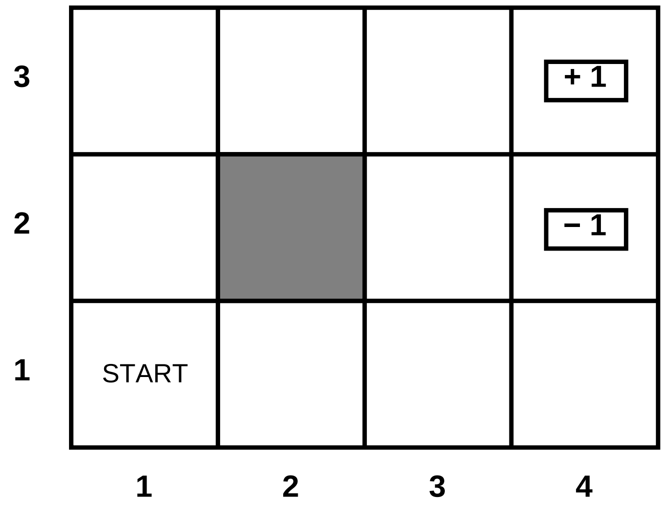
Reinforcement learning

Agent is in an unknown MDP or POMDP environment

Only feedback for learning is percept + reward

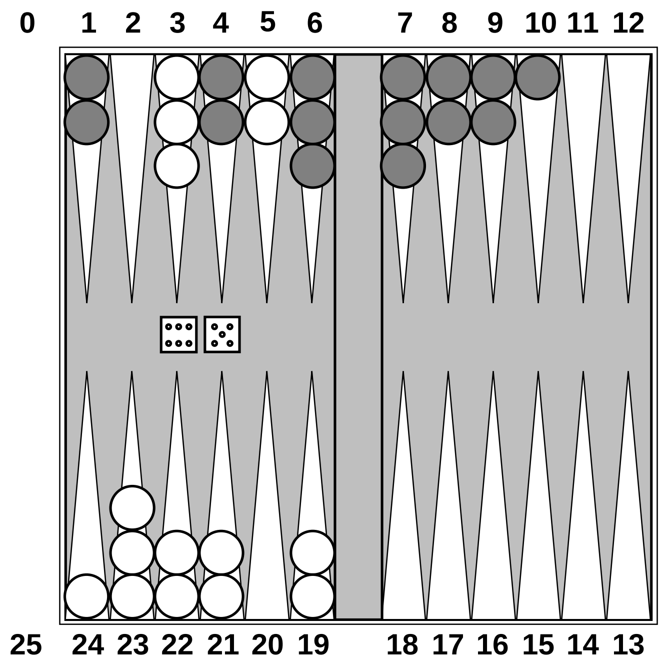
Agent must learn a policy in some form:
- transition model $T(s, a, s')$ plus value function $U(s)$
- action-value function $Q(a, s)$
- policy $\pi(s)$

Example: 4×3 world



$$(1,1)_{\mathbf{-.04}}\!\rightarrow(1,2)_{\mathbf{-.04}}\!\rightarrow(1,3)_{\mathbf{-.04}}\!\rightarrow(1,2)_{\mathbf{-.04}}\!\rightarrow(1,3)_{\mathbf{-.04}}\!\rightarrow\ \cdots\ (4,3)_{\mathbf{+1}}$$
$$(1,1)_{\mathbf{-.04}}\!\rightarrow(1,2)_{\mathbf{-.04}}\!\rightarrow(1,3)_{\mathbf{-.04}}\!\rightarrow(2,3)_{\mathbf{-.04}}\!\rightarrow(3,3)_{\mathbf{-.04}}\!\rightarrow\ \cdots\ (4,3)_{\mathbf{+1}}$$
$$(1,1)_{\mathbf{-.04}}\!\rightarrow(2,1)_{\mathbf{-.04}}\!\rightarrow(3,1)_{\mathbf{-.04}}\!\rightarrow(3,2)_{\mathbf{-.04}}\!\rightarrow(4,2)_{\mathbf{-1}}\ .$$

Example: Backgammon



Reward for win/loss only in terminal states, otherwise zero

TDGammon learns $\hat{U}(s)$, represented as 3-layer neural network

Combined with depth 2 or 3 search, one of top three players in world (after 2 million games)

Example: Animal learning

RL studied experimentally for more than 60 years in psychology

Rewards: food, pain, hunger, recreational pharmaceuticals, etc.

[Details in later lectures]

Digression: what is the animal's reward function?

Inverse reinforcement learning [Ng & Russell, 2000; Sargent, 1978]:
estimate R given samples of (presumably) optimal behavior

Issue: degenerate solutions (e.g., $R=0$)

Choose R to make observed π "very optimal" or "likely assuming noisy selection"

Example: Autonomous helicopter

Reward = – squared deviation from desired state

Temporal difference learning

Fix a policy π, execute it, learn $U^{\pi}(s)$

Bellman equation:

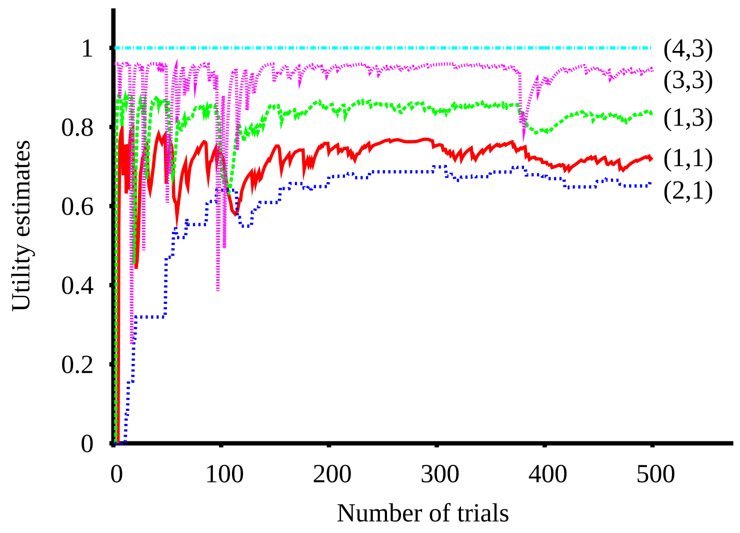
$$U^{\pi}(s) = R(s) + \gamma \sum_{s'} T(s, \pi(s), s')U^{\pi}(s')$$

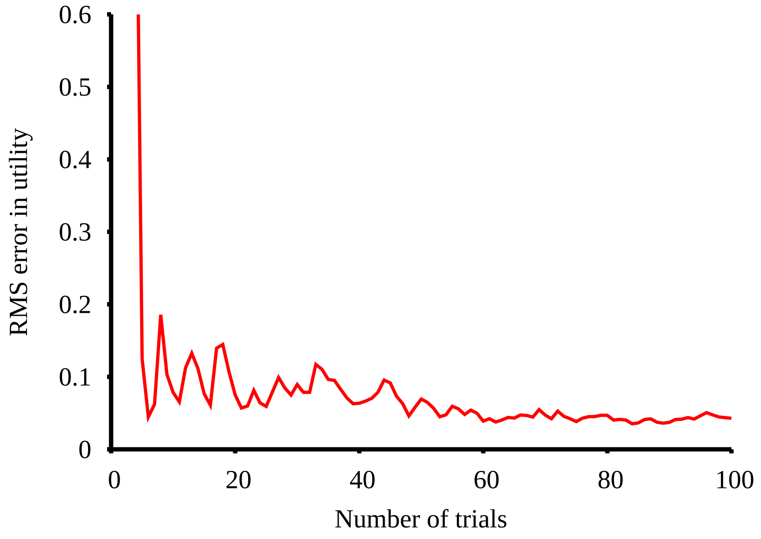
TD update adjusts utility estimate to agree with Bellman equation:

$$U^{\pi}(s) \leftarrow U^{\pi}(s) + \alpha(R(s) + \gamma\, U^{\pi}(s') - U^{\pi}(s))$$

Essentially using sampling from the environment instead of exact summation

TD performance





Q-learning [Watkins, 1989]

One drawback of learning $U(s)$: still need $T(s,a,s')$ to make decisions

Learning $Q(a,s)$ directly avoids this problem

Bellman equation:

$$Q(a,s) = R(s) + \gamma \sum_{s'} T(s,\pi(s),s') \max_{a'} Q(a',s')$$

Q-learning update:

$$Q(a,s) \leftarrow Q(a,s) + \alpha(R(s) + \gamma \max_{a'} Q(a',s') - Q(a,s))$$

😈 Q-learning is a **model-free** method for learning and decision making

☹ Q-learning is a **model-free** method for learning and decision making (so cannot use model to constrain Q-values, do mental simulation, etc.)

SARSA [Rummery & Niranjan, 1994]

Instead of using $\max_{a'} Q(a', s')$, use **actual action** a' taken in s'

SARSA update:

$$Q(a,s) \leftarrow Q(a,s) + \alpha(R(s) + \gamma\, Q(a',s') - Q(a,s))$$

Q-learning can execute any policy it wants while still learning the optimal Q

SARSA learns the Q-value for the policy the agent actually follows

Function approximation

For real problems, cannot represent U or Q as a table!!

Typically use linear function approximation (but could be anything):

$$\hat{U}_\theta(s) = \theta_1 \, f_1(s) + \theta_2 \, f_2(s) + \cdots + \theta_n \, f_n(s) \ .$$

Use a gradient step to modify θ parameters:

$$\theta_i \ \leftarrow \ \theta_i + \alpha \, [R(s) + \gamma \, \hat{U}_\theta(s') - \hat{U}_\theta(s)] \frac{\partial \hat{U}_\theta(s)}{\partial \theta_i}$$

$$\theta_i \ \leftarrow \ \theta_i + \alpha \, [R(s) + \gamma \, \max_{a'} \hat{Q}_\theta(a', s') - \hat{Q}_\theta(a, s)] \frac{\partial \hat{Q}_\theta(a, s)}{\partial \theta_i}$$

Often very effective in practice, but convergence not guaranteed
(Some narrow results for linear and instance-based approximators)

Policy search

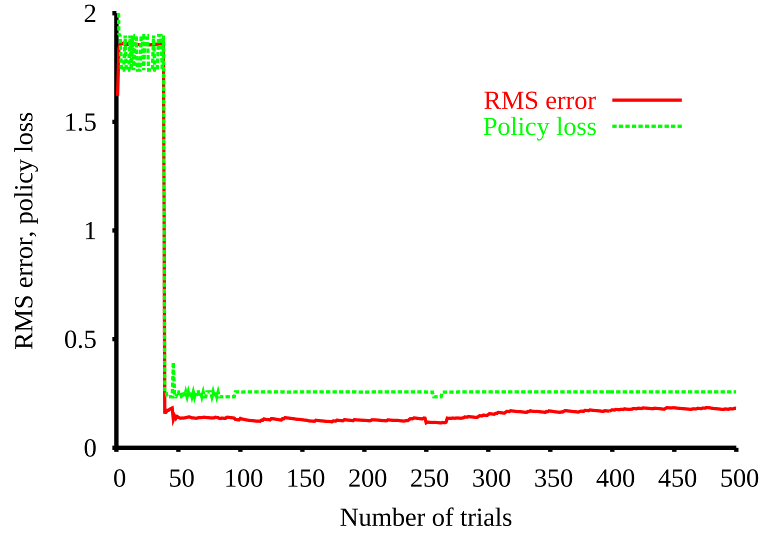
Simplest possible method:
- parameterized policy $\pi_{theta}(s)$
- try it out, see how well it does
- try out nearby values of θ, see how well they do
- follow the empirical gradient

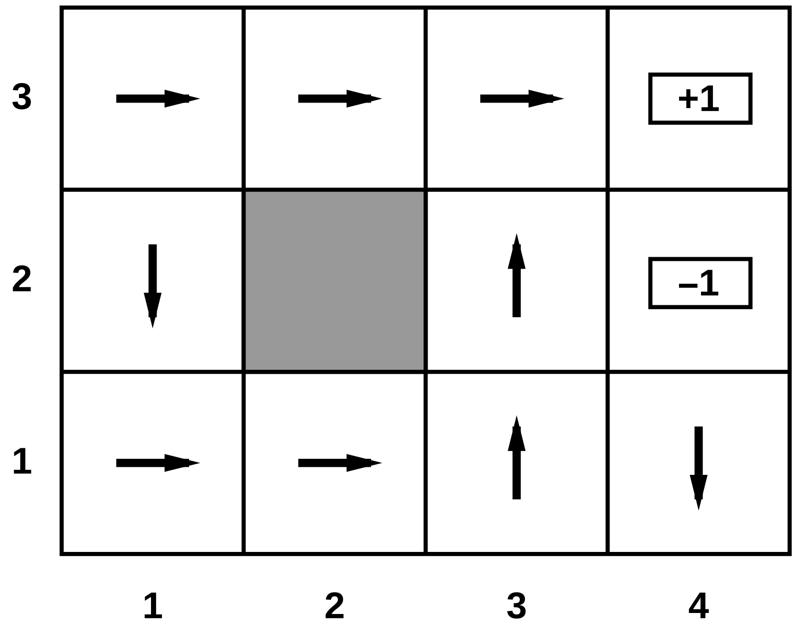
Problems: local maxima, uncertainty in policy value estimates

Useful idea [Ng & Jordan, 2000; Hammersley, 1960s]: (in simulated domains) reuse random seed across trial sets for different values of θ, thereby reducing variance in estimate of value differences

Exploration

How should the agent behave? Choose action with highest expected utility?





Exploration vs. exploitation: occasionally try "suboptimal" actions!!

Really an (intractable) "exploration POMDP" where observations give the agent information about which MDP it's in

Functional decomposition

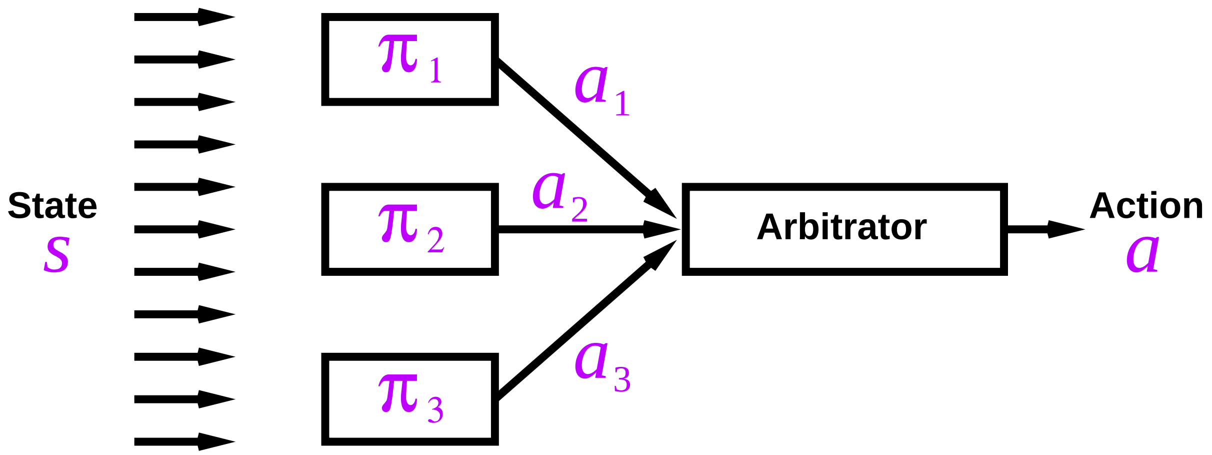
Do RL agents have to be monolithic?

Is it possible to have modules for different functions,
e.g., navigation, eating, obstacle avoidance, etc.?

. . . whilst retaining global optimality?

Command arbitration

Each sub-agent recommends an action,
arbitration logic selects among recommended actions
(e.g., subsumption architecture (Brooks, 1986))



Command arbitration contd.

Problem: Each recommendation ignores other sub-agents
 $\Rightarrow$ arbitrarily bad outcomes

Problem: Arbitration is difficult, domain-specific

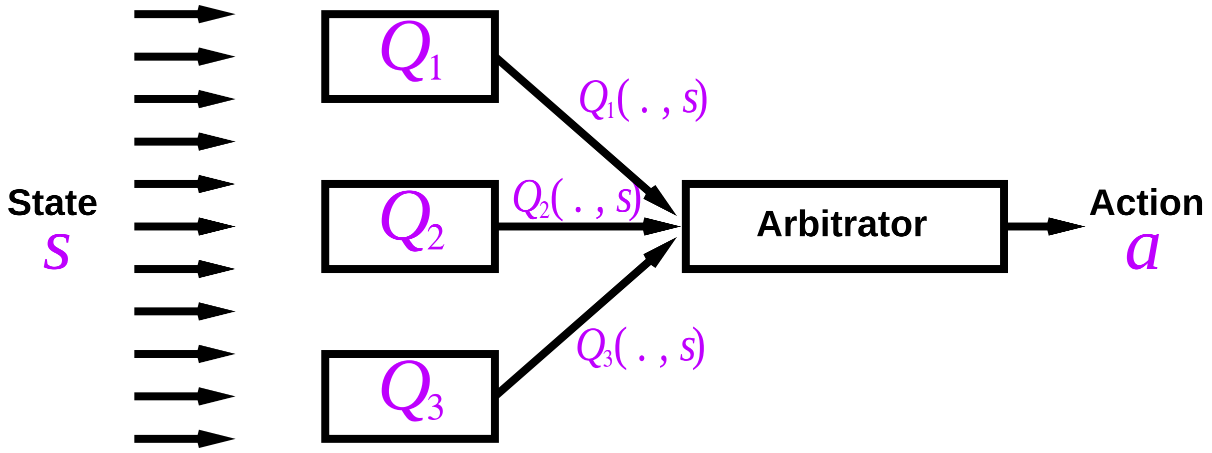
Why not "blend" the recommendations?

Problem: Blending "swerve left" and "swerve right" is a bad idea and blending "Ra6" and "Qg7" is meaningless

Q-decomposition [Russell & Zimdars, 2003]

A very obvious idea:

- ◇ **Each sub-agent embodies a local Q_j function**
- ◇ **Given current state s, sends $Q_j(s,a)$ for each a**
- ◇ **Arbitrator chooses $\arg\max_a \Sigma_j Q_j(s,a)$**



Additive rewards

Each sub-agent aims to maximize own $R_j(s, a, s')$

Additive decomposition: $R = \Sigma_j R_j$

Trivially achievable (not *relying* on any state decomposition)
but often R_j may depend on subset of state variables
while Q_j depends on all state variables

What are the Q_js?

Q_j = expected sum of R_j rewards under **globally optimal policy**

Define $Q_j^\pi(s,a) = E_{s'}\left[R_j(s,a,s') + \gamma Q_j^\pi(s',\pi(s'))\right]$

$$\begin{aligned}\Rightarrow \quad & Q^\pi(s,a) = E_{s'}[R(s,a,s') + \gamma Q^\pi(s',\pi(s'))] \\ & \quad = \Sigma_j\, Q_j^\pi(s,a)\end{aligned}$$

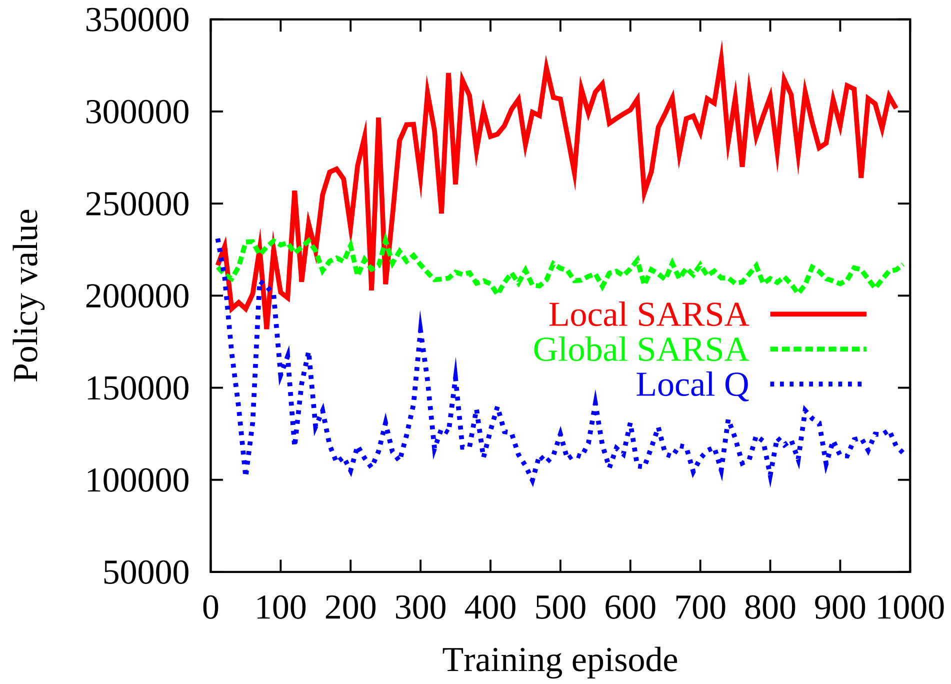
$$\Rightarrow \quad \arg\max_a \Sigma_j\, Q_j^{\pi^*}(s,a) = \arg\max_a Q^{\pi^*}(s,a)$$

I.e., arbitrator decision is globally optimal when
local Q_j functions anticipate globally optimal behavior

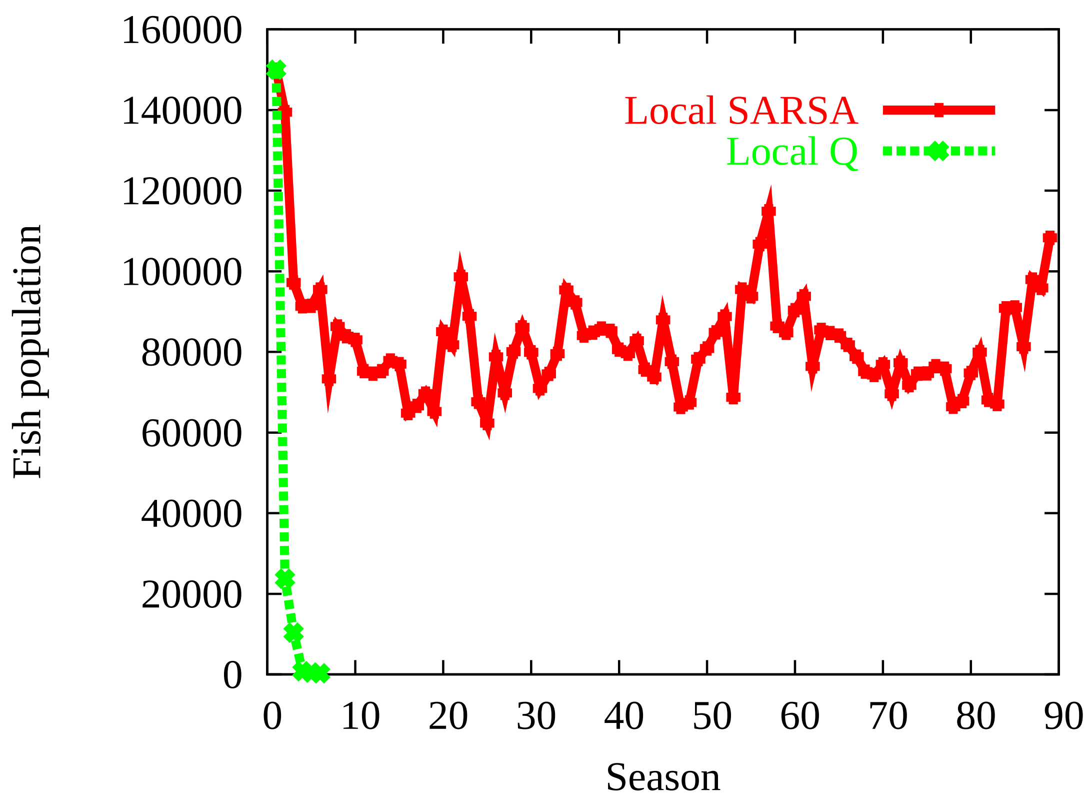
Theorem: Local SARSA converges to globally optimal Q_j
(whereas local Q-learning yields greedy sub-agents)

Example: Fisheries

Fleet of boats, each has to decide how much to catch



Fisheries results contd.



Summary

MDPs are models of sequential decision making situations

Dynamic programming (VI, PI, QI) finds exact solutions for small MDPs

Reinforcement learning finds approximate solutions for large MDPs

Work directly from experience in the environment, no initial model

Q-learning, SARSA, policy search are completely model-free

Function approximation (e.g., linear combination of features) helps RL scale up to very large MDPs

Exploration is required for convergence to optimal solutions

Agents can be decomposed into modules and still be globally optimal