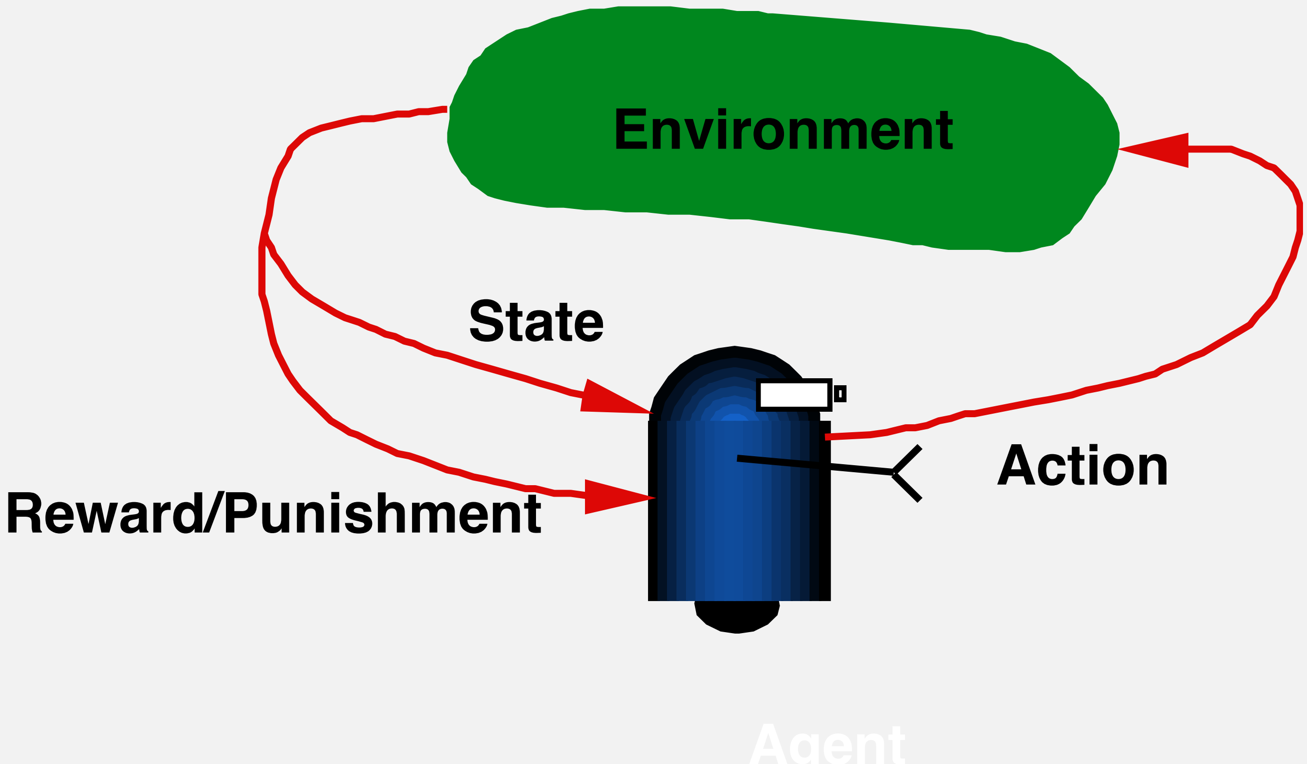
Acting under uncertainty

Vasant Honavar
Artificial Intelligence Research Laboratory
Informatics Graduate Program
Computer Science and Engineering Graduate Program
Bioinformatics and Genomics Graduate Program
Neuroscience Graduate Program

Center for Big Data Analytics and Discovery Informatics
Huck Institutes of the Life Sciences
Institute for Cyberscience
Clinical and Translational Sciences Institute
Northeast Big Data Hub
Pennsylvania State University

vhonavar@ist.psu.edu
http://faculty.ist.psu.edu/vhonavar
http://ailab.ist.psu.edu

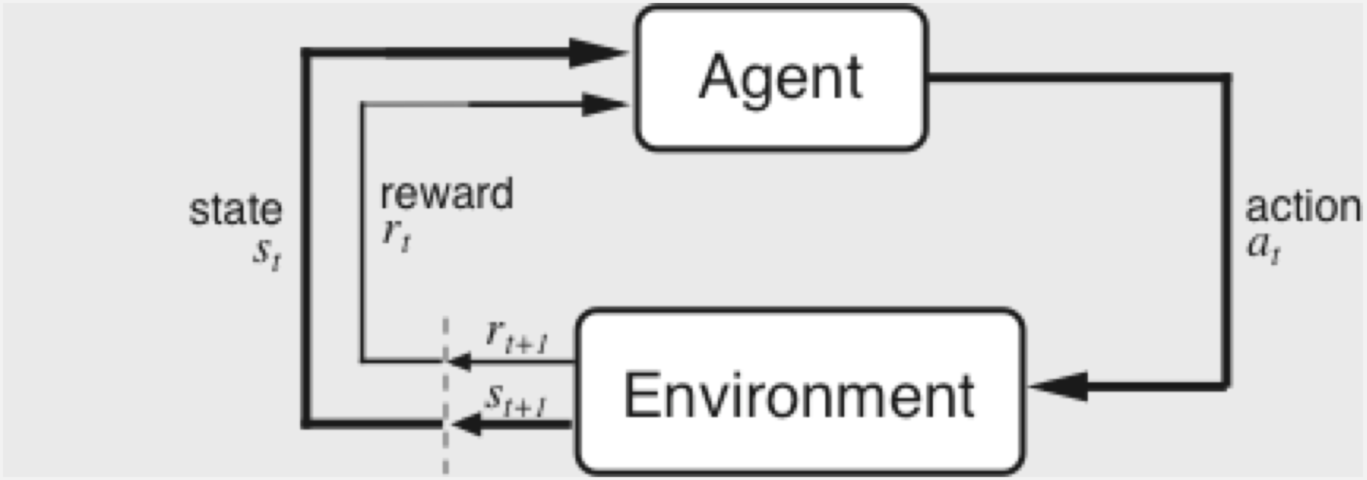
Agent in an environment



Markov Decision Processes

- Assume
  - Finite set of states S
  - Finite set of actions A
- At each discrete time
  - The agent observes state $s_t \in$ S and chooses action $a_t \in A$, receives immediate reward r_t
  - Environment state changes to s_{t+1}
- Markov assumption: $s_{t+1} = \delta(s_t, a_t)$ and $r_t = r(s_t, a_t)$
  - i.e., r_t and s_{t+1} depend only on current state and action
  - functions δ and r may be nondeterministic
  - functions δ and r may not necessarily be known to the agent – reinforcement learning

Acting rationally in the presence of delayed rewards



Agent and environment interact at discrete time steps: $t = 0, 1, 2, \ldots$

Agent observes state at step t: $s_t \in S$

produces action at step t: $a_t \in A(s_t)$

gets resulting reward: $r_{t+1} \in \Re$

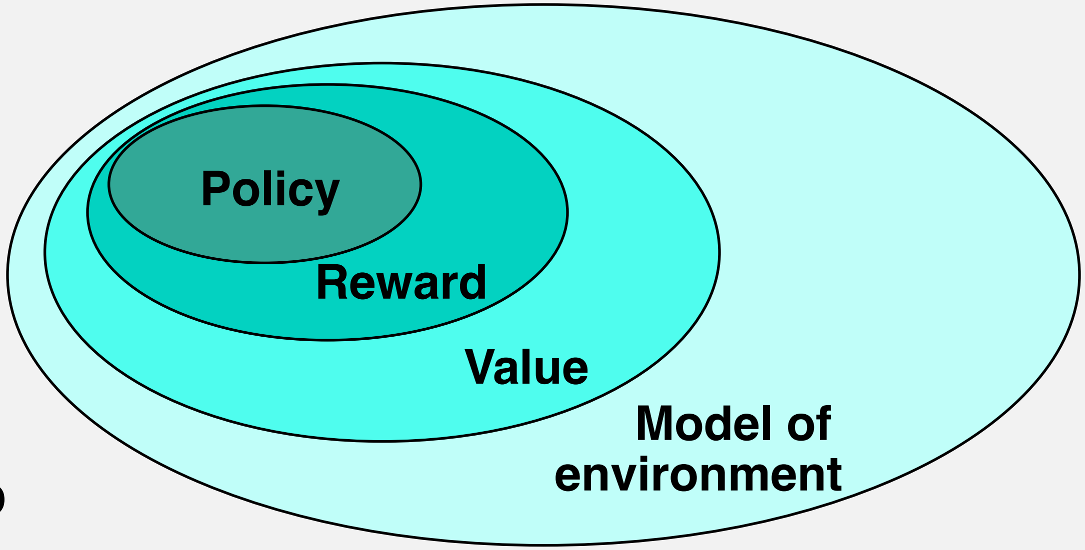
and resulting next state: s_{t+1}

$$\cdots \; \underline{\quad} \; s_t \; \xrightarrow[a_t]{r_{t+1}} \; s_{t+1} \; \xrightarrow[a_{t+1}]{r_{t+2}} \; s_{t+2} \; \xrightarrow[a_{t+2}]{r_{t+3}} \; s_{t+3} \; \xrightarrow[a_{t+3}]{} \cdots$$

Learning from Interaction with the world

- An agent receives sensations or percepts from the environment through its sensors and acts on the environment through its effectors and occasionally receives rewards or punishments from the environment
- The goal of the agent is to maximize its reward (pleasure) or minimize its punishment (or pain) as it stumbles along in an a-priori unknown, uncertain, environment

Key elements of an RL System



- Policy – what to do
- Reward – what is good
- Value – what is good because it predicts reward
- Model – what follows what

The Agent Uses a Policy to select actions

Policy at step t, π_t:

a mapping from states to action probabilities

$$\pi_t(s,a) = \text{ probability that } a_t = a \text{ when } s_t = s$$

- A rational agent's goal is to get as much reward as it can over the long run

Goals and Rewards

- Is a scalar reward signal an adequate notion of a goal? – maybe not, but it is surprisingly flexible.
- A goal should specify what we want to achieve, not how we want to achieve it.
- A goal is typically outside the agent's direct control
- The agent must be able to measure success:
  - explicitly
  - frequently during its lifespan

Rewards for Continuing Tasks

Continuing tasks: interaction does not have natural episodes

Cumulative discounted reward

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1},$$

where $\gamma, 0 \leq \gamma < 1$, is the discount rate.

shortsighted $0 \leftarrow \gamma \rightarrow 1$ farsighted

Rewards

Suppose the sequence of rewards after step t is :

$$r_{t+1}, r_{t+2}, r_{t+3}, \ldots$$

What do we want to maximize?

In general, we want to maximize the expected return, $E\{R_t\}$, for each step t.

Episodic tasks – interaction breaks naturally into episodes, e.g., plays of a game, trips through a maze.

$$R_t = r_{t+1} + r_{t+2} + \cdots + r_T,$$

where T is a final time step at which a terminal state is reached, ending an episode.

Markov Decision Processes

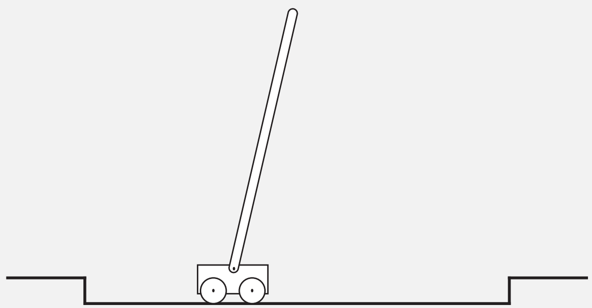
- If a reinforcement learning task has the Markov Property, it is called a Markov Decision Process (MDP).
- If state and action sets are finite, the MDP is a finite MDP.
- To define a finite MDP, you need to specify:
  - state and action sets;
  - one-step dynamics defined by transition probabilities:

$$P_{ss'}^{a} = \Pr\{s_{t+1} = s' \mid s_t = s, a_t = a\} \quad \forall\, s, s' \in S, a \in A(s).$$

  - reward probabilities:

$$R_{ss'}^{a} = E\{r_{t+1} \mid s_t = s, a_t = a, s_{t+1} = s'\} \quad \forall\, s, s' \in S, a \in A(s).$$

Example – Pole Balancing Task



Avoid failure: the pole falling beyond a critical angle or the cart hitting end of track.

As an episodic task where episode ends upon failure:

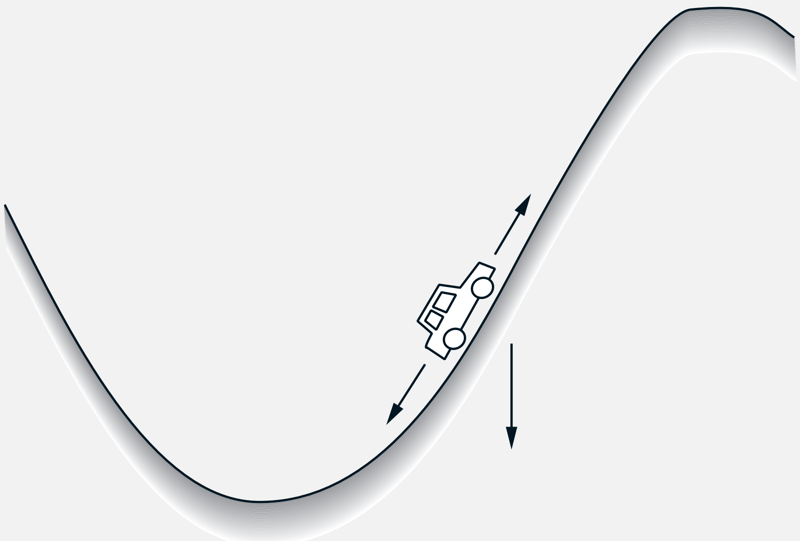
$$\text{reward} = +1 \text{ for each step before failure}$$
$$\Rightarrow \quad \text{return} = \text{number of steps before failure}$$

As a continuing task with discounted return:

$$\text{reward} = -1 \text{ upon failure; } 0 \text{ otherwise}$$
$$\Rightarrow \quad \text{return} = -\gamma^k, \text{ for } k \text{ steps before failure}$$

In either case, return is maximized by avoiding failure for as long as possible.

Example -- Driving task



Get to the top of the hill as quickly as possible.

$$\text{reward } = -1 \text{ for each step when } \textbf{not} \text{ at top of hill}$$
$$\Rightarrow \text{ return} = -\text{number of steps before reaching top of hill}$$

Return is maximized by minimizing the number of steps taken to reach the top of the hill

Finite MDP Example

Recycling Robot

- At each step, robot has to decide whether it should
  a) actively search for a can,
  b) wait for someone to bring it a can, or
  c) go to home base and recharge.
- Searching is better but runs down the battery; if runs out of power while searching, has to be rescued (which is bad).
- Decisions made on basis of current energy level: **`high`**, **`low`**.
- Reward = number of cans collected

Some Notable RL Applications

- TD-Gammon
  - Worlds best backgammon program
- Elevator scheduling
- Inventory Management
  - 10% – 15% improvement over the state-of-the art methods
- Dynamic Channel Assignment –
  - high performance assignment of radio channels to mobile telephone calls

The Markov Property

- By the state at step *t*, we mean whatever information is available to the agent at step *t* about its environment.
- The state can include immediate sensations, highly processed sensations, and structures built up over time from sequences of sensations.
- Ideally, a state should summarize past sensations so as to retain all essential information – it should have the Markov Property:

$$\Pr\left\{s_{t+1} = s', r_{t+1} = r \mid s_t, a_t, r_t, s_{t-1}, a_{t-1}, \ldots, r_1, s_0, a_0\right\} = \Pr\left\{s_{t+1} = s', r_{t+1} = r \mid s_t, a_t\right\}$$

$$\forall s', r, \text{ and histories } s_t, a_t, r_t, s_{t-1}, a_{t-1}, \ldots, r_1, s_0, a_0.$$

Reinforcement learning

- Learner is not told which actions to take
- Rewards and punishments may be delayed
  - Sacrifice short-term gains for greater long-term gains
- The need to tradeoff between exploration and exploitation
- Environment may not be observable or only partially observable
- Environment may be deterministic or stochastic

Value Functions

- The value of a state is the expected return starting from that state; depends on the agent's policy:

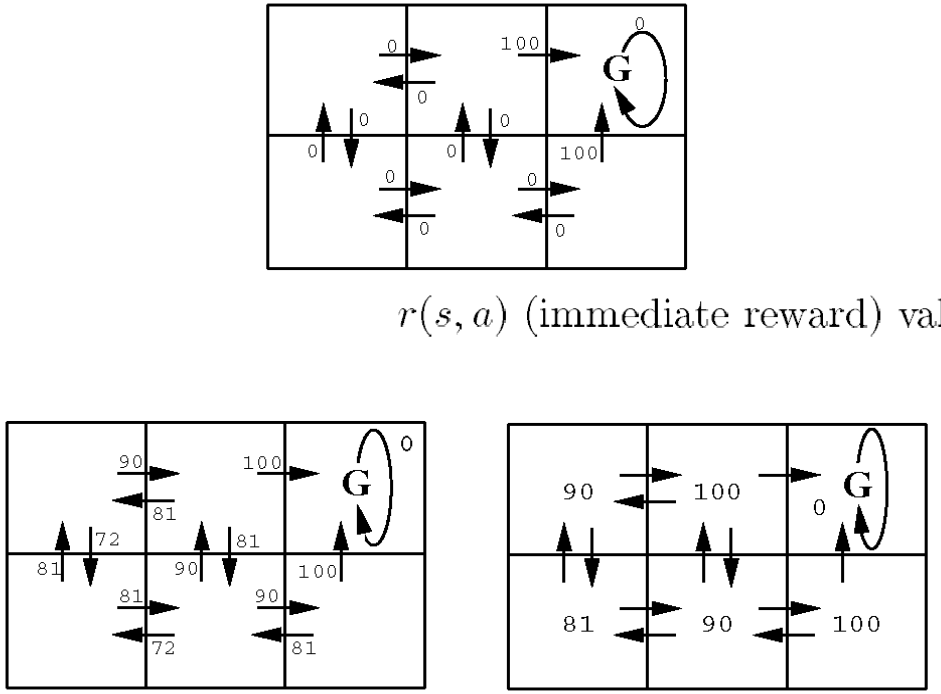
State - value function for policy π :

$$V^{\pi}(s) = E_{\pi}\{R_t \mid s_t = s\} = E_{\pi}\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \,\middle|\, s_t = s\right\}$$

The value of taking an action in a state under policy π is the expected cumulative reward starting from that state, taking that action, and thereafter following π :

Action - value function for policy π :

$$Q^{\pi}(s,a) = E_{\pi}\{R_t \mid s_t = s, a_t = a\} = E_{\pi}\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \,\middle|\, s_t = s, a_t = a\right\}$$



$r(s, a)$ (immediate reward) values

$Q(s, a)$ values

$V^*(s)$ values

Bellman Equation for a Policy π

The basic idea:

$$\begin{aligned} R_t &= r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} \cdots \\ &= r_{t+1} + \gamma\left(r_{t+2} + \gamma r_{t+3} + \gamma^2 r_{t+4} \cdots\right) \\ &= r_{t+1} + \gamma R_{t+1} \end{aligned}$$

So:

$$\boxed{\begin{aligned} V^\pi(s) &= E_\pi\left\{R_t \mid s_t = s\right\} \\ &= E_\pi\left\{r_{t+1} + \gamma V\left(s_{t+1}\right) \mid s_t = s\right\} \end{aligned}}$$

Or, without the expectation operator:

$$V^\pi(s) = \sum_a \pi(s,a) \sum_{s'} P_{ss'}^a \left[R_{ss'}^a + \gamma V^\pi(s')\right]$$

Optimal Value Functions

- For finite MDPs, policies can be partially ordered:

$$\pi \geq \pi' \quad \text{if and only if} \quad V^{\pi}(s) \geq V^{\pi'}(s) \;\; \text{for all } s \in S$$

- There is always at least one (and possibly many) policies that is (are) better than or equal to all the others. This is an optimal policy. We denote them all $\pi *$.

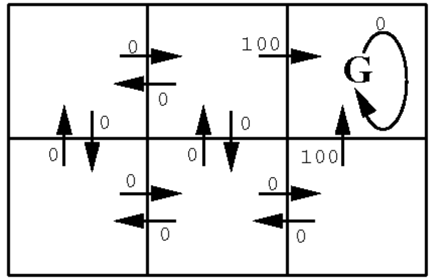
- Optimal policies share the same optimal state-value function:

$$V^{*}(s) = \max_{\pi} V^{\pi}(s) \quad \text{for all } \; s \in S$$

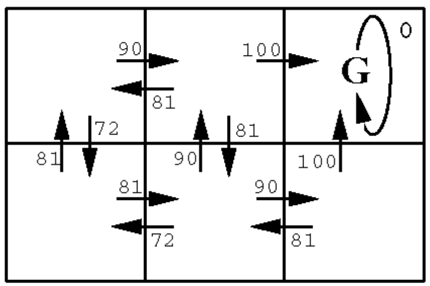
- Optimal policies also share the same optimal action-value function:

$$Q^{*}(s,a) = \max_{\pi} Q^{\pi}(s,a) \;\; \text{for all } \; s \in S \text{ and } a \in A(s)$$

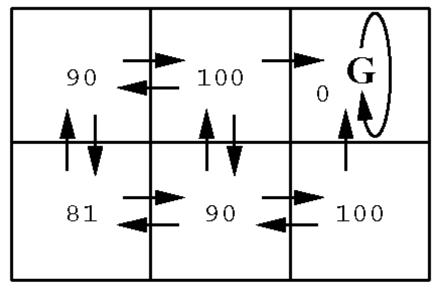
This is the expected cumulative reward for taking action a in state s and thereafter following an optimal policy.



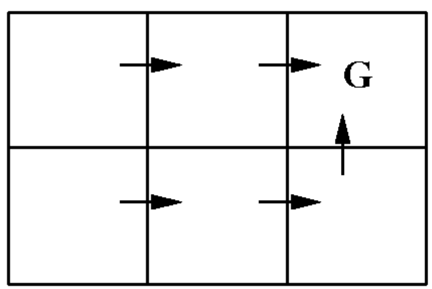
$r(s,a)$ (immediate reward) values



$Q(s,a)$ values



$V^*(s)$ values



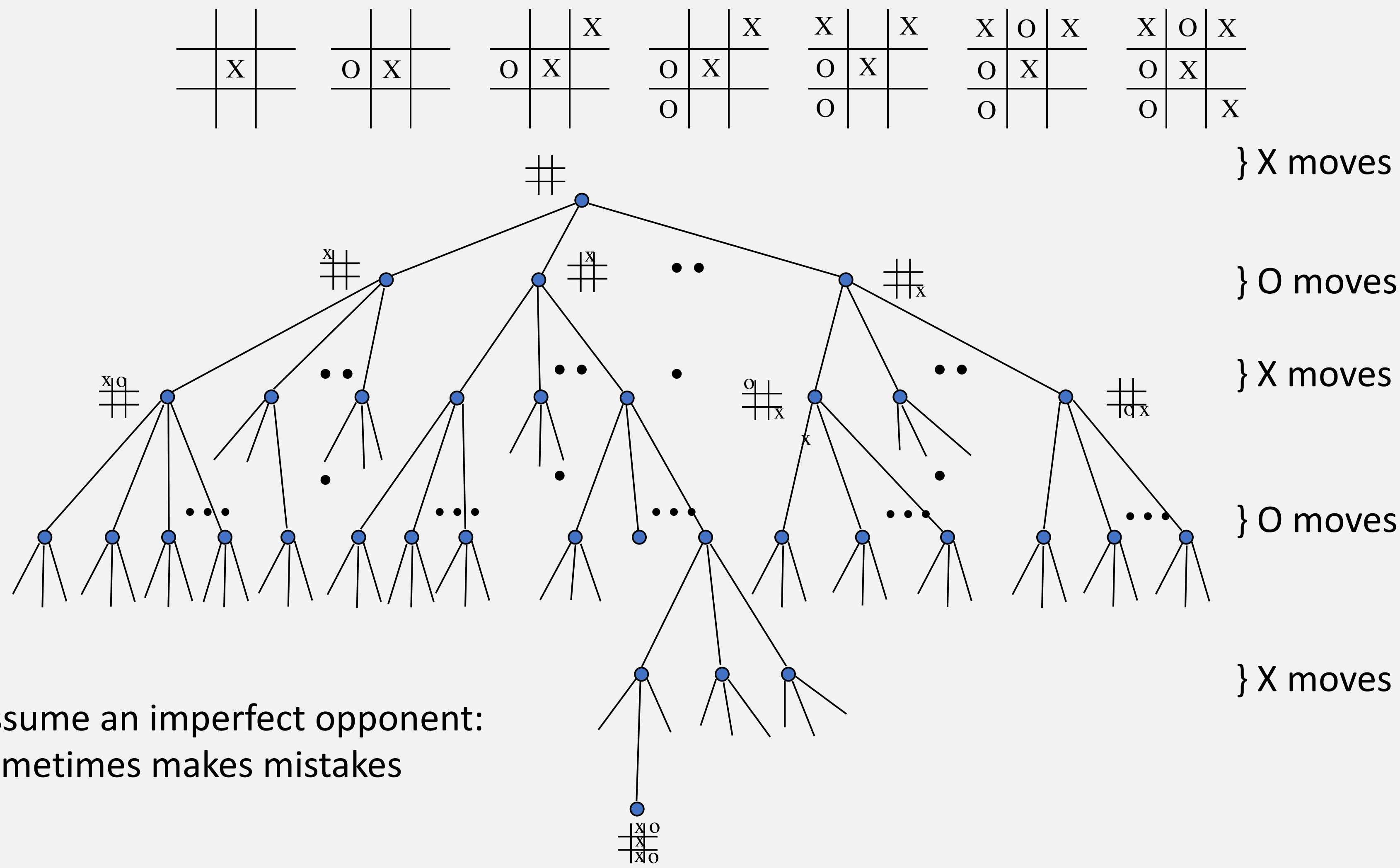
One optimal policy

Why Optimal State-Value Functions are Useful

Any policy that is greedy with respect to V^* is an optimal policy.

Therefore, given V^*, one-step-ahead search produces the long-term optimal actions.

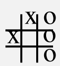
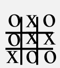
Example: Tic-Tac-Toe



Assume an imperfect opponent: sometimes makes mistakes

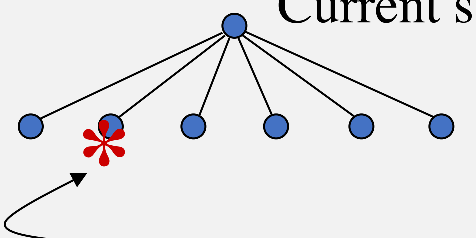
A Simple RL Approach to Tic-Tac-Toe

Make a table with one entry per state

State	V(s) – estimated probability of winning	
	0.5	
	0.5	
⋮	⋮	
	1	win
⋮	⋮	
	0	loss
⋮	⋮	
	0.5	draw

Play lots of games.
To pick our moves, look ahead one step

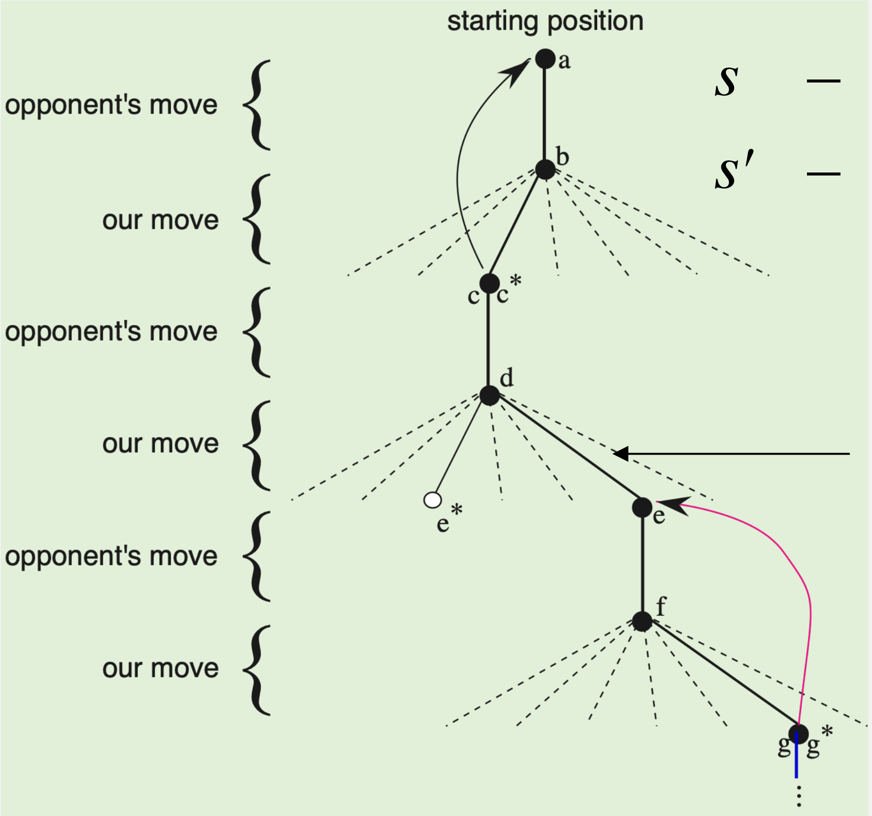
Current state



Possible next states

Pick the next state with the highest estimated prob. of winning — the largest V(s) – a greedy move;
Occasionally pick a move at random – an exploratory move.

Learning Rule for Tic-Tac-Toe



s — the state before our greedy move

s' — the state after our greedy move

"Exploratory" move

We increment each $V(s)$ toward $V(s')$ – a ***backup***:

$$V(s) \leftarrow V(s) + \alpha\left[V(s') - V(s)\right]$$

Why is Tic-Tac-Toe Too Easy?

- Number of states is small and finite
- One-step look-ahead is always possible
- State completely observable

What About Optimal Action-Value Functions?

Given Q^* the agent does not even have to do a one-step-ahead search:

$$\pi^*(s) = \arg\max_{a \in A(s)} Q^*(s,a)$$

Simpler Setting: The *n*-Armed Bandit Problem

- Choose repeatedly from one of *n* actions; each choice is called a play
- After each play a_t you get a reward r_t where

$$E\langle r_t \mid a_t \rangle = Q^*(a_t)$$

- Distribution of r_t depends only on a_t
- No dependence on state
- Objective is to maximize the reward in the long term, e.g., over 1000 plays

The Exploration – Exploitation Dilemma

- Suppose you form action value estimates

$$Q_t(a) \approx Q^*(a)$$

- The greedy action at *t* is

$$a_t^* = \arg\max_a Q_t(a)$$

$$a_t = a_t^* \Rightarrow \text{exploitation}$$

$$a_t \neq a_t^* \Rightarrow \text{exploration}$$

- You can't exploit all the time; you can't explore all the time
- You can never stop exploring; but you could reduce exploring

Action-Value Methods

- Adapt action-value estimates and nothing else.
- Suppose by the t-th play, action a had been chosen k_a times, producing rewards $r_1, r_2, \ldots, r_{k_a}$, then

$$Q_t(a) = \frac{r_1 + r_2 + \cdots r_{k_a}}{k_a}$$

$$\lim_{k_a \to \infty} Q_t(a) = Q^*(a)$$

ε-Greedy Action Selection

- Greedy

$$a_t = a_t^* = \arg\max_a Q_t(a)$$

- ε-Greedy

$$a_t = \begin{cases} a_t^* & \text{with probability } 1-\varepsilon \\ \text{random action with probability } \varepsilon \end{cases}$$

- Boltzmann

$$\Pr(\text{choosing action } a \text{ at time } t) = \frac{e^{Q_t(a)/\tau}}{\sum_{b=1}^{n} e^{Q_t(b)/\tau}}$$

where τ is *computational temperature*

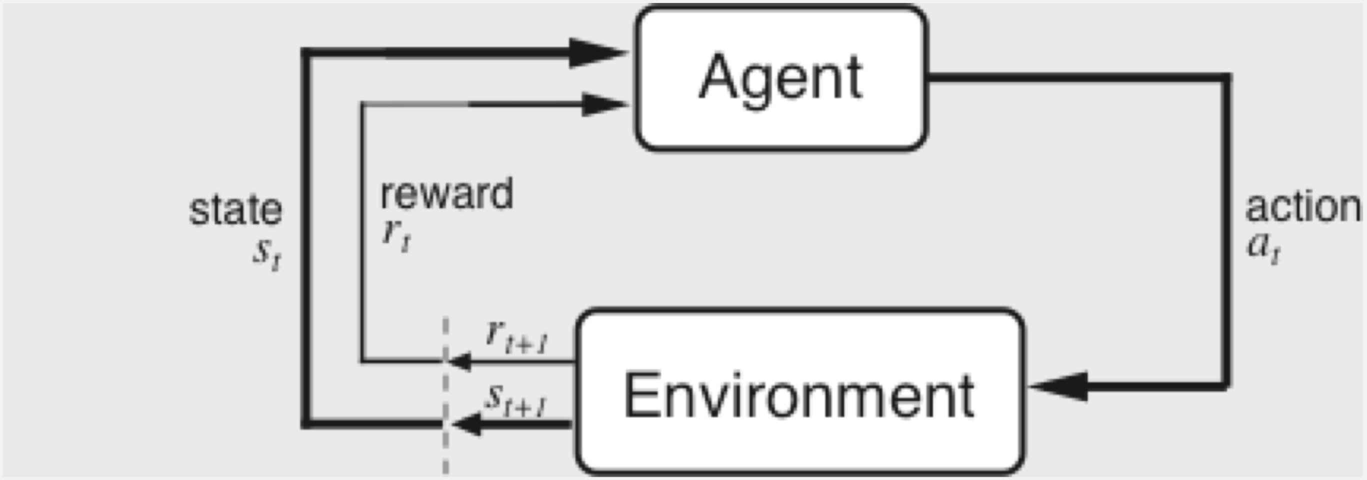
Solving the Bellman Optimality Equation

- Finding an optimal policy by solving the Bellman Optimality Equation requires:
  - accurate knowledge of environment dynamics;
  - enough space an time to do the computation;
  - the Markov Property.
- How much space and time do we need?
  - polynomial in number of states (via dynamic programming methods),
  - BUT, number of states is often huge
- We usually have to settle for approximations.
- Many RL methods can be understood as approximately solving the Bellman Optimality Equation.

Some Notable RL Applications

- TD-Gammon
  - Worlds best backgammon program
- Elevator scheduling
- Inventory Management
  - 10% – 15% improvement over the state-of-the art methods
- Dynamic Channel Assignment –
  - high performance assignment of radio channels to mobile telephone calls

Agents that Acts Rationally in the presence of delayed rewards



Agent and environment interact at discrete time steps: $t = 0, 1, 2, \ldots$

Agent observes state at step t: $s_t \in S$

produces action at step t: $a_t \in A(s_t)$

gets resulting reward: $r_{t+1} \in \Re$

and resulting next state: s_{t+1}

$$\ldots \; \text{---} \; s_t \; \xrightarrow[a_t]{} \; r_{t+1} \; s_{t+1} \; \xrightarrow[a_{t+1}]{} \; r_{t+2} \; s_{t+2} \; \xrightarrow[a_{t+2}]{} \; r_{t+3} \; s_{t+3} \; \xrightarrow[a_{t+3}]{} \ldots$$

The Agent Learns a Policy

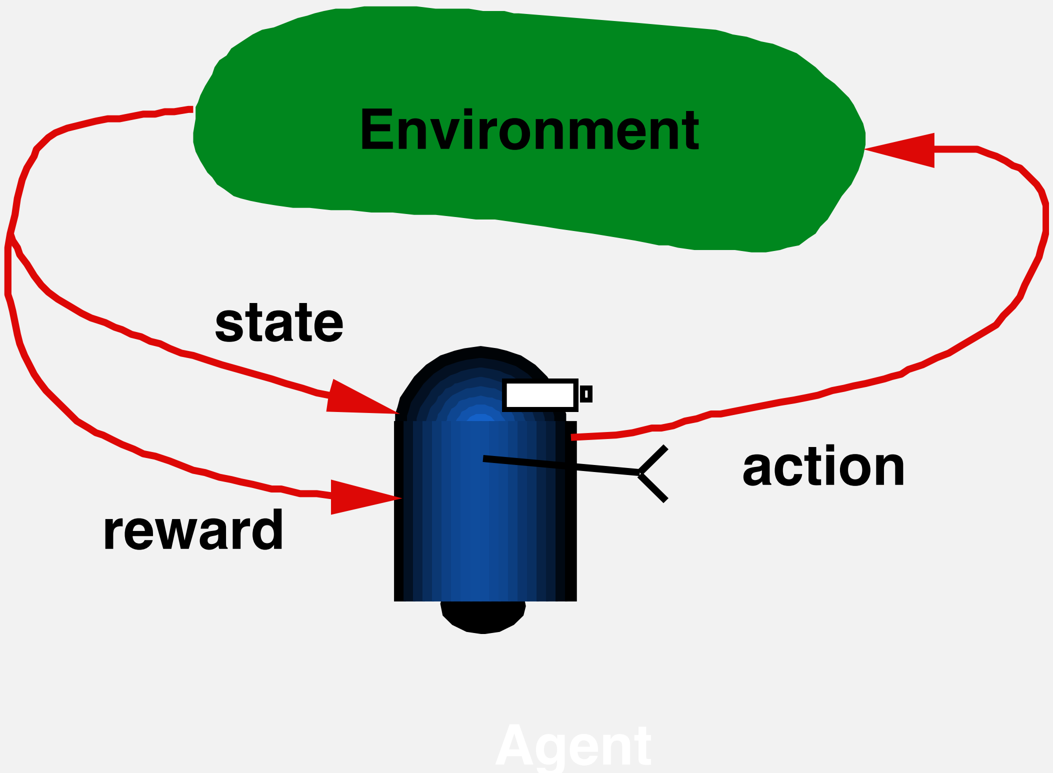
Policy at step t, π_t:

a mapping from states to action probabilities

$\pi_t(s,a) =$ probability that $a_t = a$ when $s_t = s$

- Reinforcement learning methods specify how the agent changes its policy as a result of its experience.
- Roughly, the agent's goal is to get as much reward as it can over the long run.

Reinforcement learning



Markov Decision Processes

- Assume
  - Finite set of states S
  - Finite set of actions A
- At each discrete time
  - The agent observes state $s_t \in S$ and chooses action $a_t \in A$, receives immediate reward r_t
  - Environment state changes to s_{t+1}
- Markov assumption: $s_{t+1} = \delta(s_t, a_t)$ and $r_t = r(s_t, a_t)$
  - i.e., r_t and s_{t+1} depend only on current state and action
  - functions δ and r may be nondeterministic
  - functions δ and r may not necessarily be known to the agent – reinforcement learning

Agent's learning task

- Execute actions in environment, observe results, and
- Learn action policy $\pi : S \rightarrow A$ that maximizes

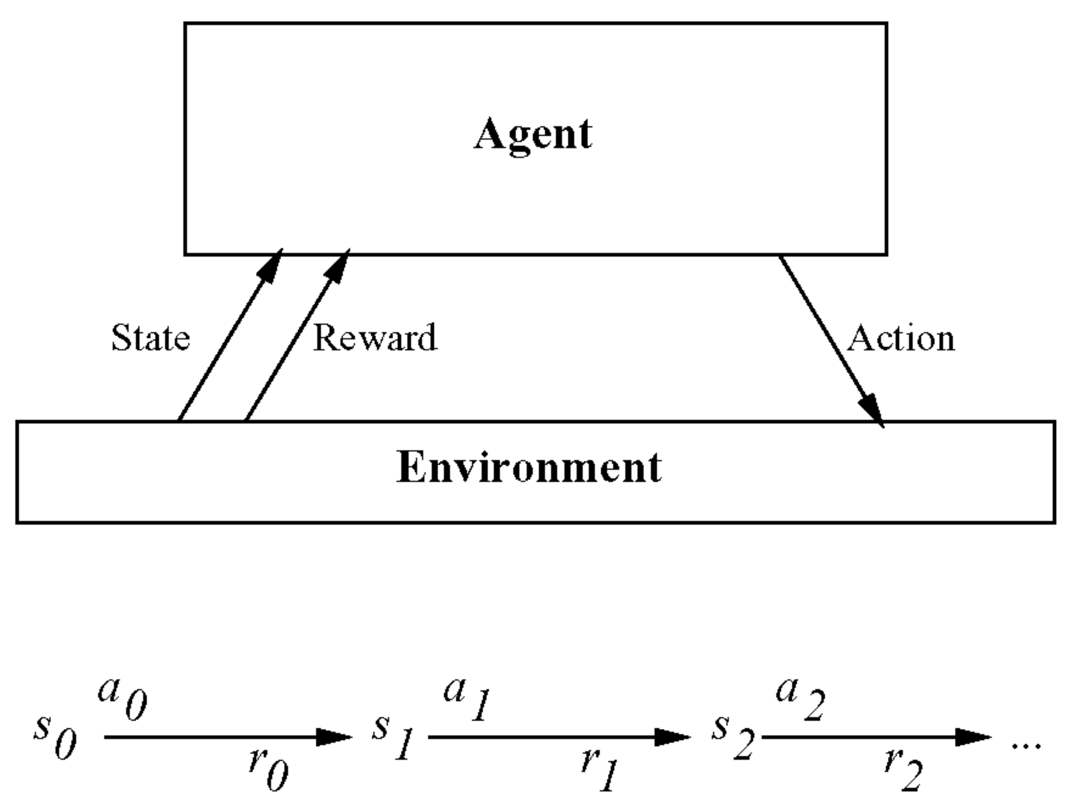
$$E\,[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots\,]$$

from any starting state in S

- Here $0 \leq \gamma < 1$ is the discount factor for future rewards

- Target function is to be learned is $\pi : S \rightarrow A$
- But we have no training examples of form $\langle s, a \rangle$
- Training examples are of form $\langle \langle s, a \rangle, r \rangle$

Reinforcement learning problem



- Goal: learn to choose actions that maximize

  $r_0 + \gamma r_1 + \gamma^2 r_2 + \ldots$, where $0 \leq \gamma < 1$

Value function

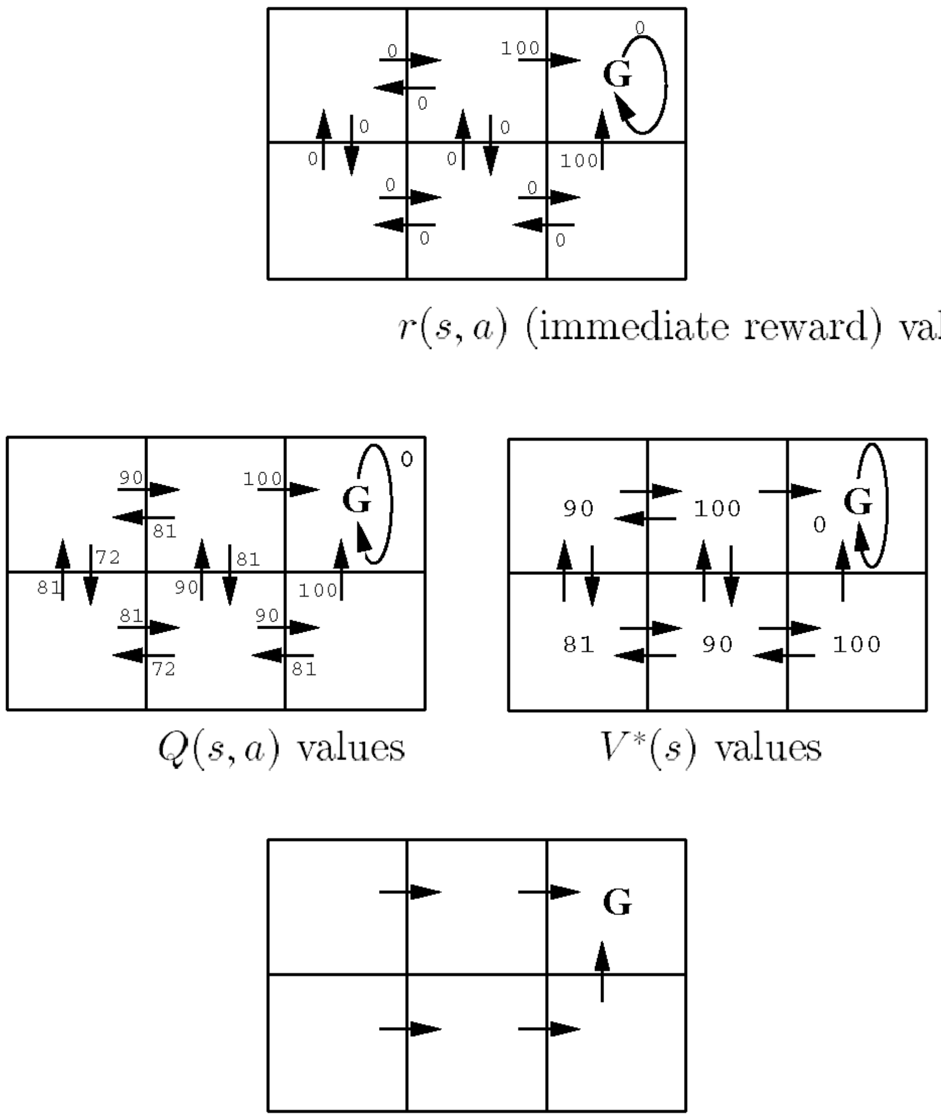
- To begin with, consider deterministic worlds...
- For each possible policy π the agent might adopt, we can define an evaluation function over states

$$V^{\pi}(s) \equiv r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots$$
$$\equiv \sum_{i=0}^{\infty} \gamma^i r_{t+i}$$

where $r_t, r_{t+1}, \ldots$ are generated by following policy π starting at state s

- Restated, the task is to learn the optimal policy π^*

$$\pi^* \equiv \arg\max_{\pi} V^{\pi}(s), (\forall s)$$



$r(s, a)$ (immediate reward) values

$Q(s, a)$ values

$V^*(s)$ values

One optimal policy

What to learn

- We might try to have agent learn the evaluation function V^{π^*} (which we write as V^*)
- It could then do a look-ahead search to choose best action from any state s because

$$\pi^*(s) \equiv \underset{a}{\arg\max}[r(s,a) + \gamma V^*(\delta(s,a))]$$

A problem:

- This works if agent knows $\delta : S \times A \rightarrow S$, and $r : S \times A \rightarrow \Re$
- But when it doesn't, it can't choose actions in this way

Action-Value function – Q function

- Define a new function very similar to V*

$$Q(s,a) \equiv r(s,a) + \gamma V^*(\delta(s,a))$$

- If agent learns Q, it can choose optimal action even without knowing δ!

$$\pi^*(s) \equiv \arg\max_{\pi}[r(s,a) + \gamma V^*(\delta(s,a))]$$

$$\pi^*(s) \equiv \arg\max_{\pi} Q(s,a)$$

- Q is the evaluation function the agent will learn

Training rule to learn Q

- Note Q and V* are closely related:

$$V^*(s) = \max_{a'} Q(s,a'))$$

- Which allows us to write Q recursively as

$$\begin{aligned} Q(s_1,a_1) &= r(s_1,a_1) + \gamma V^*(\delta(s_1,a_1)) \\ &= r(s_1,a_1) + \gamma \max_{a'} Q(\delta(s_{t+1},a')) \end{aligned}$$

- Let $\hat{Q}$ denote learner's current approximation to Q. Consider training rule

$$\hat{Q}(s,a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s',a')$$

- where s' is the state resulting from applying action a in state s.

Q-Learning

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

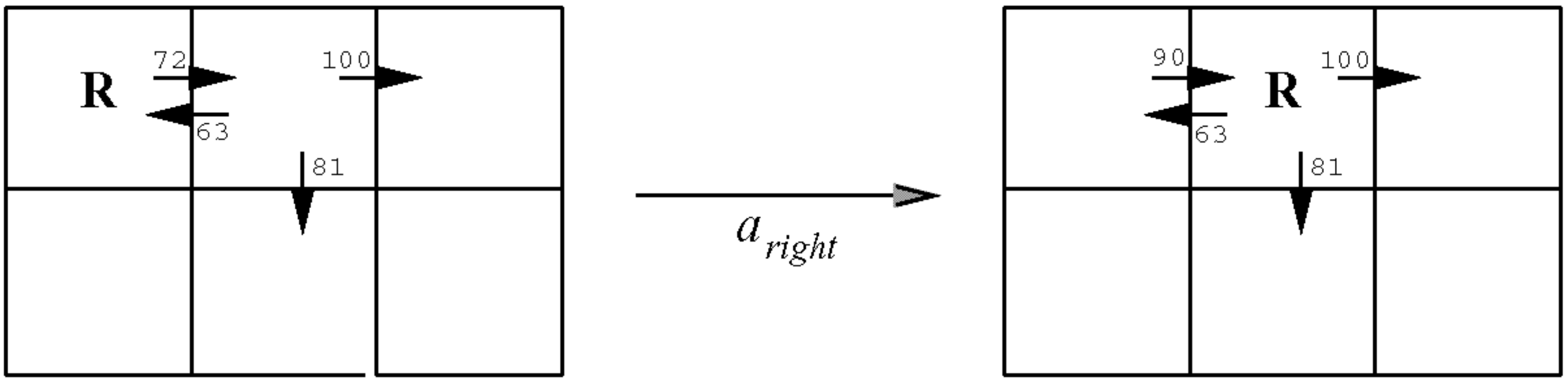
Initialize $Q(s, a)$ arbitrarily
Repeat (for each episode):
  Initialize s
  Repeat (for each step of episode):
    Choose a from s using policy derived from Q (e.g., ϵ-greedy)
    Take action a, observe r, s'
    $Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a)\right]$
    $s \leftarrow s'$;
  until s is terminal

Q Learning for Deterministic Worlds

- For each s, a initialize table entry $\hat{Q}(s,a) \leftarrow 0$
- Do forever:
  - Observe current state s
  - Select an action a and execute it
  - Receive immediate reward r
  - Observe the new state s'
  - Update the table entry for $\hat{Q}(s,a)$ as follows:

$$\hat{Q}(s) \leftarrow r + \gamma \max_{a'} \hat{Q}(s',a')$$

Updating Q



initial state: s_1 next state: s_2

$$\hat{Q}(s_1, a_{right}) \leftarrow r + \gamma \max_{a'} \hat{Q}(s_2', a')$$
$$\leftarrow 0 + 0.9 \max\{63, 81, 100\}$$
$$\leftarrow 90$$

Notice if rewards non-negative, then

$$(\forall s, a, n) \quad \hat{Q}_{n+1}(s,a) \geq \hat{Q}_n(s,a)$$

and

$$(\forall s, a, n) \quad 0 \leq \hat{Q}_n(s,a) \leq Q(s,a)$$

Convergence theorem

- Theorem: $\hat{Q}$ converges to Q.
- Consider case of deterministic world, with bounded immediate rewards, where each ⟨*s, a*⟩ visited infinitely often.
- Proof: Define an interval during which each ⟨*s, a*⟩ is visited at least once. During each full interval the largest error in table is reduced by factor of γ.
- Let $\hat{Q}_n$ be the $\hat{Q}$ table after n updates, and Δ_n be the maximum error in $\hat{Q}_n$:

$$\Delta_n = \max_{s,a} | \hat{Q}_n(s,a) - Q(s,a) |$$

Convergence theorem

- For any table entry $\hat{Q}_n(s,a)$ updated on iteration $n + 1$, the error in the revised estimate $\hat{Q}_{n+1}(s,a)$s

$$
\begin{aligned}
|\hat{Q}_{n+1}(s,a) - Q(s,a)| &= |(r + \gamma \max_{a'} \hat{Q}_n(s',a')) - (r + \gamma \max_{a'} Q(s',a'))| \\
&= \gamma \, |\max_{a'} \hat{Q}_n(s',a') - \max_{a'} Q(s',a')|
\end{aligned}
$$

Convergence theorem

$$
\begin{aligned}
|\hat{Q}_{n+1}(s,a) - Q(s,a)| &= |(r + \gamma \max_{a'} \hat{Q}_n(s',a')) - (r + \gamma \max_{a'} Q(s',a'))| \\
&= \gamma \,|\max_{a'} \hat{Q}_n(s',a') - \max_{a'} Q(s',a')| \\
&\leq \gamma \max_{a'} |\hat{Q}_n(s',a') - Q(s',a')| \\
&\leq \gamma \max_{s'',a'} |\hat{Q}_n(s'',a') - Q(s'',a')| \\
|\hat{Q}_{n+1}(s,a) - Q(s,a)| &= \gamma \Delta_n
\end{aligned}
$$

Note we used general fact that:

$$
|\max_a f_1(a) - \max_a f_2(a)| \leq \max_a |f_1(a) - f_2(a)|
$$

Non-deterministic case

- What if reward and next state are non-deterministic?
- We redefine V and Q by taking expected values.

$$V^{\pi}(s) \equiv E[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots]$$
$$\equiv E\left[\sum_{i=0}^{\infty} \gamma^i r_{t+i}\right]$$

$$Q(s,a) \equiv E[r(s,a) + \gamma V^*(\delta(s,a))]$$

Nondeterministic case

Q learning generalizes to nondeterministic worlds Alter training rule to

$$\hat{Q}_n(s,a) \leftarrow (1-\alpha_n)\hat{Q}_{n-1}(s,a) + \alpha_n[r + \max_{a'} \hat{Q}_{n-1}(s',a')]$$

where

$$\alpha_n = \frac{1}{1 + visits_n(s,a)}$$

Convergence of $\hat{Q}$ to Q can be proved [Watkins and Dayan, 1992]

Sarsa

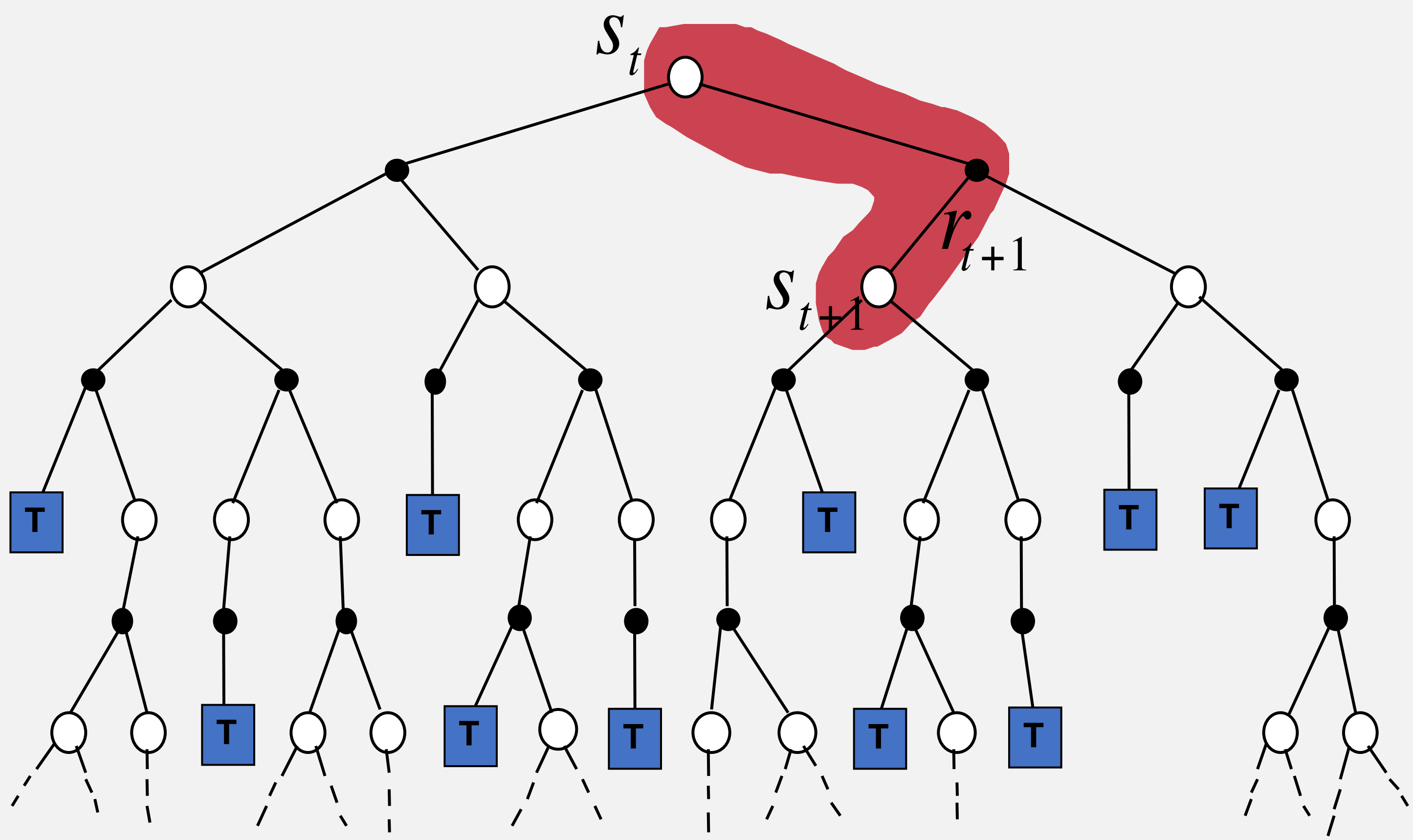
Always update the policy to be greedy with respect to the current estimate

Initialize $Q(s, a)$ arbitrarily
Repeat (for each episode):
  Initialize s
  Choose a from s using policy derived from Q (e.g., ϵ-greedy)
  Repeat (for each step of episode):
    Take action a, observe r, s'
    Choose a' from s' using policy derived from Q (e.g., ϵ-greedy)
    $Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma Q(s', a') - Q(s, a)\right]$
    $s \leftarrow s'; a \leftarrow a';$
  until s is terminal

Temporal Difference Learning

- Temporal Difference (TD) learning methods
  - Can be used when accurate models of the environment are unavailable – neither state transition function nor reward function are known
  - Can be extended to work with implicit representations of action-value functions
  - Are among the most useful reinforcement learning methods

Q-learning: The simplest TD Method TD(0)



Example – TD-Gammon

- Learn to play Backgammon (Tesauro, 1995)
- Immediate reward:
- +100 if win
- -100 if lose
- 0 for all other states

- Trained by playing 1.5 million games against itself.
- Now comparable to the best human player.

Temporal difference learning

Q learning: reduce discrepancy between successive Q estimates

One step time difference:

$$Q^{(1)}(s_t, a_t) \equiv r_t + \gamma \max_a \hat{Q}(s_{t+1}, a)$$

Why not two steps?

$$Q^{(2)}(s_t, a_t) \equiv r_t + \gamma r_{t+1} + \gamma^2 \max_a \hat{Q}(s_{t+2}, a)$$

Or n?

$$Q^{(n)}(s_t, a_t) \equiv r_t + \gamma r_{t+1} + \cdots + \gamma^{(n-1)} r_{t+n-1} + \gamma^n \max_a \hat{Q}(s_{t+n}, a)$$

Blend all of these:

$$Q^{\lambda}(s_t, a_t) \equiv (1-\lambda)[Q^{(1)}(s_t, a_t) + \lambda Q^{(2)}(s_t, a_t) + \lambda^2 Q^{(3)}(s_t, a_t)$$

Temporal difference learning

$$Q^{\lambda}(s_t, a_t) \equiv (1-\lambda)[Q^{(1)}(s_t, a_t) + \lambda Q^{(2)}(s_t, a_t) + \lambda^2 Q^{(3)}(s_t, a_t)$$

Equivalent expression:

$$Q^{\lambda}(s_t, a_t) = r_t + \gamma[(1-\lambda)\max_a \hat{Q}(s_t, a_t) + \lambda Q^{\lambda}(s_{t+1}, a_{t+1})]$$

- TD(λ) algorithm uses above training rule
- Sometimes converges faster than Q learning
- converges for learning V^* for any $0 \le \lambda \le 1$ (Dayan, 1992)
- Tesauro's TD-Gammon uses this algorithm

Advantages of TD Learning

- TD methods do not require a model of the environment, only experience
- TD methods can be fully incremental
  - You can learn before knowing the final outcome
    - Less memory
    - Less peak computation
  - You can learn without the final outcome
    - From incomplete sequences
- TD converges (under reasonable assumptions)

Optimality of TD(0)

- Batch Updating: train completely on a finite amount of data, e.g., train repeatedly on 10 episodes until convergence.
- Compute updates according to TD(0), but only update estimates after each complete pass through the data.

For any finite Markov prediction task, under batch updating, TD(0) converges for sufficiently small α.

Handling Large State Spaces

- Replace $\hat{Q}$ table with neural net or other function approximator
- Virtually any function approximator would work provided it can be updated in an online fashion

Recall Q-learning Algorithm

Initialize $Q(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\left[R + \gamma \max_a Q(S', a) - Q(S, A)\right]$
    $S \leftarrow S'$
  until S is terminal

Q-function Approximation

- Basic idea is to use a neural network to approximate the Q function.
- Define a set of features over state-action pairs: $f_1(s,a)$, …, $f_n(s,a)$
  - State-action pairs with similar feature values will be treated similarly
  - More complex functions require more complex features

$$\hat{Q}_\theta(s,a) = \theta_0 + \theta_1 f_1(s,a) + \theta_2 f_2(s,a) + \ldots + \theta_n f_n(s,a)$$

- We can generalize Q-learning to update the parameters of the Q-function approximation

Learning state-action values

- Training examples of the form:

$$< \ (s_t, a_t), \ v_t >$$

- The general gradient-descent rule:

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha \left[v_t - Q_t(s_t, a_t) \right] \nabla_{\vec{\theta}} Q(s_t, a_t)$$

Gradient Descent

$$\overleftarrow{\theta}_{t+1} = \overleftarrow{\theta}_t - \frac{1}{2}\eta\nabla_{\overleftarrow{\theta}}\sum_s\left[r_t + \gamma max_a\hat{Q}(\delta(s_t, a_t), a) - \hat{Q}(s_t, a_t)\right]^2$$

$$= \overleftarrow{\theta}_t + \frac{1}{2}\eta\left[r_t + \gamma max_a\hat{Q}(\delta(s_t, a_t), a) - \hat{Q}(s_t, a_t)\right]\nabla_{\overleftarrow{\theta}}\hat{Q}(s_t, a_t)$$

Suppose

$$\hat{Q}_\theta(s,a) = \theta_0 + \theta_1 f_1(s,a) + \theta_2 f_2(s,a) + \ldots + \theta_n f_n(s,a)$$

Then

$$= \overleftarrow{\theta}_t + \frac{1}{2}\eta\left[r_t + \gamma max_a\hat{Q}(\delta(s_t, a_t), a) - \hat{Q}(s_t, a_t)\right]\left[1, f_1(s,a), f_2(s,a), \cdots f_n(s,a)\right]$$

Problem: Samples are correlated, not IID!

Deep Q-Networks (DQN)

- Basic idea is to use a function approximator $Q(s, a; \theta)$ to approximate the action-value function in Q-learning
- Deep Q-Networks use a neural network, the Q-network, to approximate the Q function
- Discrete and finite set of actions A

Q-Networks

- Core idea: We want the neural network to learn a non-linear feature representation that approximates the Q table
- The neural network has an output unit for each possible action, which gives the Q-value estimate for that action in the given state
- The neural network is trained using mini-batch stochastic gradient updates and experience replay

online Q iteration algorithm:

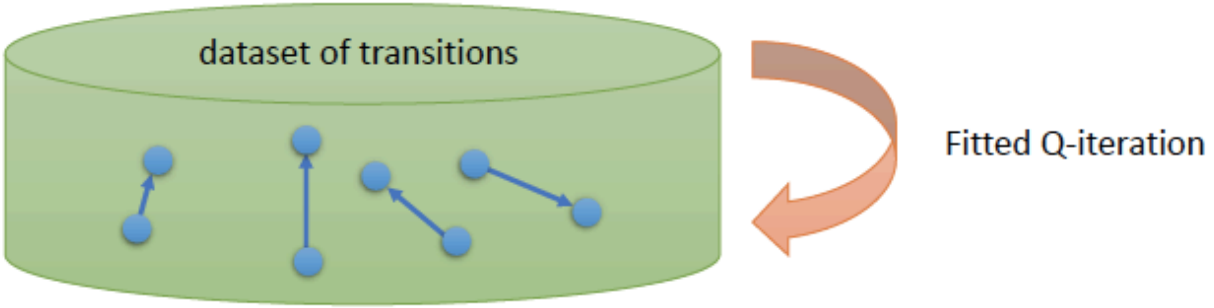
special case with K = 1, and one gradient step

1. take some action $\mathbf{a}_i$ and observe $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$
2. $\phi \leftarrow \phi - \alpha \frac{dQ_\phi}{d\phi}(\mathbf{s}_i, \mathbf{a}_i)(Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - [r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)])$

full fitted Q-iteration algorithm:

1. ~~collect dataset $\{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)\}$ using some policy~~ — any policy will work! (with broad support)
2. set $\mathbf{y}_i \leftarrow r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'_i} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)$ — just load data from a buffer here
3. set $\phi \leftarrow \arg\min_\phi \frac{1}{2} \sum_i \|Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - \mathbf{y}_i\|^2$ — still use one gradient step

$K\times$



Another solution: replay buffers

Q-learning with a replay buffer:

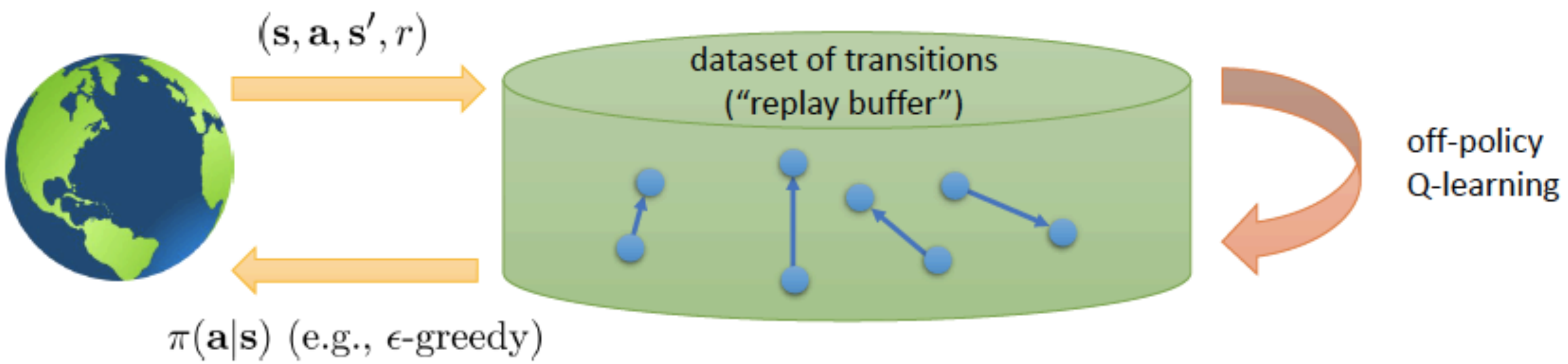
1. sample a batch $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$ from $\mathcal{B}$
2. $\phi \leftarrow \phi - \alpha \underline{\sum_i} \frac{dQ_\phi}{d\phi}(\mathbf{s}_i, \mathbf{a}_i)(Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - [r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)])$

+ samples are no longer correlated

+ multiple samples in the batch (low-variance gradient)

but where does the data come from?

need to periodically feed the replay buffer…



Putting it together

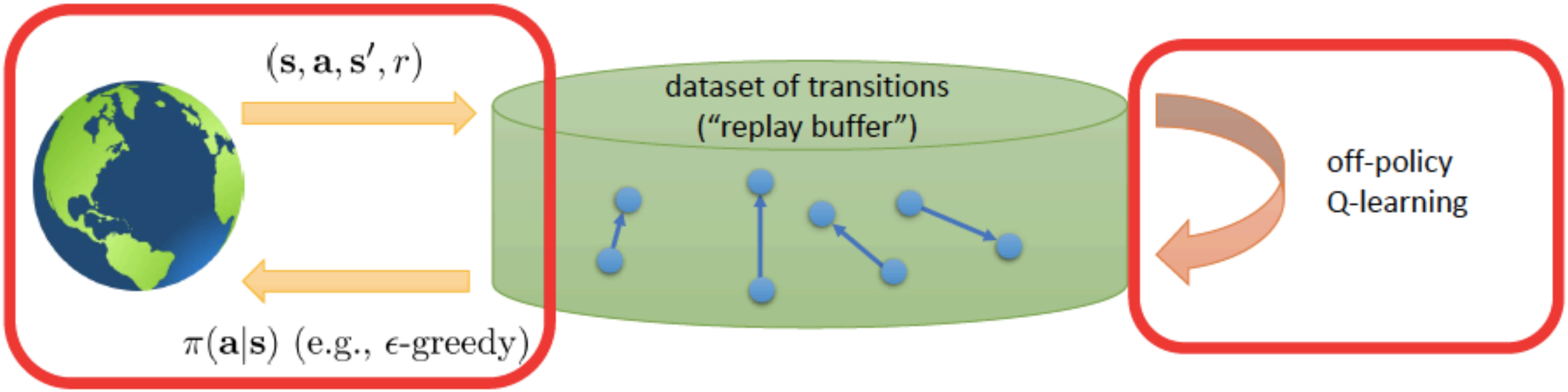
full Q-learning with replay buffer:

1. collect dataset $\{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)\}$ using some policy, add it to $\mathcal{B}$

$K\times$

2. sample a batch $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$ from $\mathcal{B}$
3. $\phi \leftarrow \phi - \alpha \sum_i \frac{dQ_\phi}{d\phi}(\mathbf{s}_i, \mathbf{a}_i)(Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - [r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)])$

K = 1 is common, though larger K more efficient



What's wrong?

online Q iteration algorithm:

1. take some action $\mathbf{a}_i$ and observe $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$
2. $\mathbf{y}_i = r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)$
3. $\phi \leftarrow \phi - \alpha \frac{dQ_\phi}{d\phi}(\mathbf{s}_i, \mathbf{a}_i)(Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - \mathbf{y}_i)$

~~these are correlated!~~

use replay buffer

Q-learning is *not* gradient descent!

$$\phi \leftarrow \phi - \alpha \frac{dQ_\phi}{d\phi}(\mathbf{s}_i, \mathbf{a}_i)(Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - [r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)])$$

no gradient through target value

This is still a problem!

Q-Learning and Regression

full Q-learning with replay buffer:

1. collect dataset $\{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)\}$ using some policy, add it to $\mathcal{B}$
2. sample a batch $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$ from $\mathcal{B}$
3. $\phi \leftarrow \phi - \alpha \sum_i \underline{\frac{dQ_\phi}{d\phi}(\mathbf{s}_i, \mathbf{a}_i)(Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - [r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)])}$

$K\times$

one gradient step, moving target

full fitted Q-iteration algorithm:

1. collect dataset $\{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)\}$ using some policy
2. set $\mathbf{y}_i \leftarrow r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'_i} Q_\phi(\mathbf{s}'_i, \mathbf{a}'_i)$
3. set $\underline{\phi \leftarrow \arg\min_\phi \frac{1}{2} \sum_i \|Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - \mathbf{y}_i\|^2}$

$K\times$

perfectly well-defined, stable regression

Q-Learning with target networks

Q-learning with replay buffer and target network:

1. save target network parameters: $\phi' \leftarrow \phi$
   2. collect dataset $\{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)\}$ using some policy, add it to $\mathcal{B}$
      3. sample a batch $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$ from $\mathcal{B}$
      4. $\phi \leftarrow \phi - \alpha \sum_i \frac{dQ_\phi}{d\phi}(\mathbf{s}_i, \mathbf{a}_i)(Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - \underline{[r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'} Q_{\phi'}(\mathbf{s}'_i, \mathbf{a}'_i)]})$

$N\times$ $K\times$

targets don't change in inner loop!

supervised regression

"Classic" deep Q-learning algorithm (DQN)

Q-learning with replay buffer and target network:

1. save target network parameters: $\phi' \leftarrow \phi$
   2. collect dataset $\{(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)\}$ using some policy, add it to $\mathcal{B}$
      3. sample a batch $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$ from $\mathcal{B}$
      4. $\phi \leftarrow \phi - \alpha \sum_i \frac{dQ_\phi}{d\phi}(\mathbf{s}_i, \mathbf{a}_i)(Q_\phi(\mathbf{s}_i, \mathbf{a}_i) - [r(\mathbf{s}_i, \mathbf{a}_i) + \gamma \max_{\mathbf{a}'} Q_{\phi'}(\mathbf{s}'_i, \mathbf{a}'_i)])$

$N\times$ $K\times$

"classic" deep Q-learning algorithm:

1. take some action $\mathbf{a}_i$ and observe $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$, add it to $\mathcal{B}$
2. sample mini-batch $\{\mathbf{s}_j, \mathbf{a}_j, \mathbf{s}'_j, r_j\}$ from $\mathcal{B}$ uniformly
3. compute $y_j = r_j + \gamma \max_{\mathbf{a}'_j} Q_{\phi'}(\mathbf{s}'_j, \mathbf{a}'_j)$ using *target* network $Q_{\phi'}$
4. $\phi \leftarrow \phi - \alpha \sum_j \frac{dQ_\phi}{d\phi}(\mathbf{s}_j, \mathbf{a}_j)(Q_\phi(\mathbf{s}_j, \mathbf{a}_j) - y_j)$
5. update ϕ': copy ϕ every N steps

$K = 1$

DQN Algorithm

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation $\boxed{3}$
  end for
end for

Q-learning suggested readings

- Classic papers
  - Watkins. (1989). Learning from delayed rewards: introduces Q-learning
  - Riedmiller. (2005). Neural fitted Q-iteration: batch-mode Q-learning with neural networks
- Deep reinforcement learning Q-learning papers
  - Lange, Riedmiller. (2010). Deep auto-encoder neural networks in reinforcement learning: early image-based Q-learning method using autoencoders to construct embeddings
  - Mnih et al. (2013). Human-level control through deep reinforcement learning: Q-learning with convolutional networks for playing Atari.
  - Van Hasselt, Guez, Silver. (2015). Deep reinforcement learning with double Q-learning: a very effective trick to improve performance of deep Q-learning.
  - Lillicrap et al. (2016). Continuous control with deep reinforcement learning: continuous Q-learning with actor network for approximate maximization.
  - Gu, Lillicrap, Stuskever, L. (2016). Continuous deep Q-learning with model-based acceleration: continuous Q-learning with action-quadratic value functions.
  - Wang, Schaul, Hessel, van Hasselt, Lanctot, de Freitas (2016). Dueling network architectures for deep reinforcement learning: separates value and advantage estimation in Q-function.

Not covered

- Using hierarchical state and action representations
- Coping with partial observability
- Coping with extremely large state spaces
- Neural basis of reinforcement learning