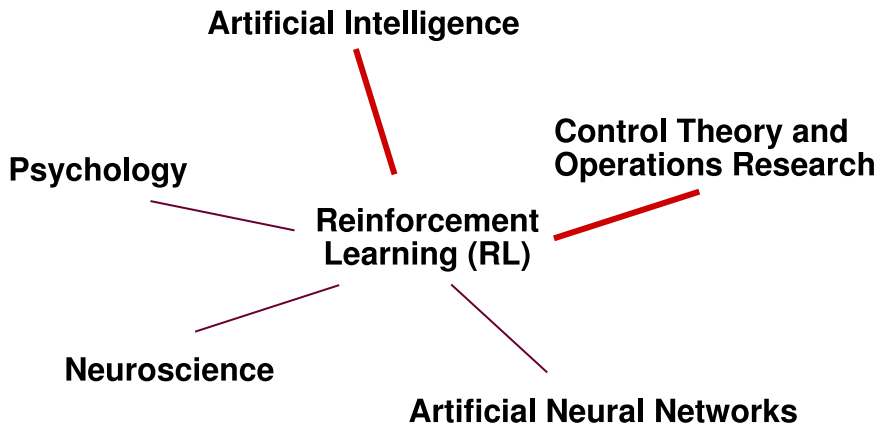
Reinforcement Learning

- Mainly based on "Reinforcement Learning – An Introduction" by Richard Sutton and Andrew Barto
- Slides are mainly based on the course material provided by the same authors

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

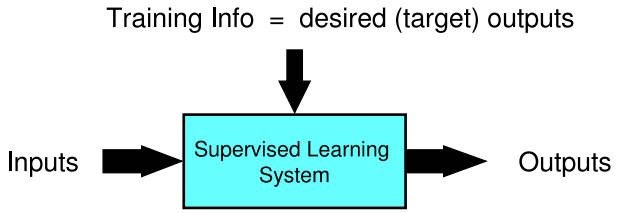
Learning from Experience Plays a Role in …



What is Reinforcement Learning?

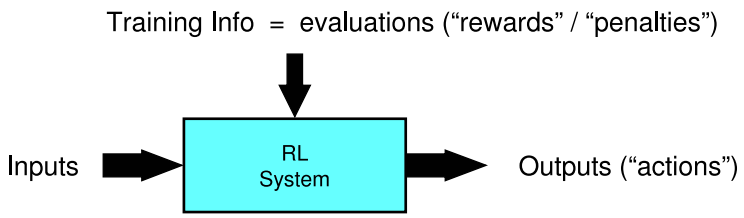
- Learning from interaction
- Goal-oriented learning
- Learning about, from, and while interacting with an external environment
- Learning what to do—how to map situations to actions—so as to maximize a numerical reward signal

Supervised Learning



Error = (target output – actual output)

Reinforcement Learning



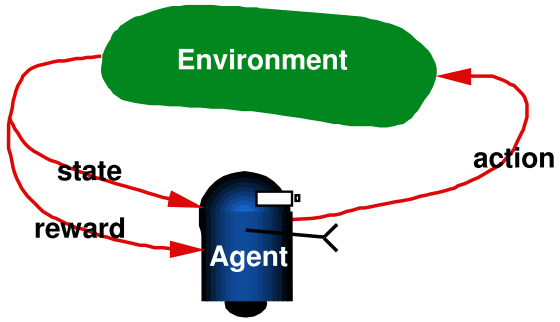
Objective: get as much reward as possible

Key Features of RL

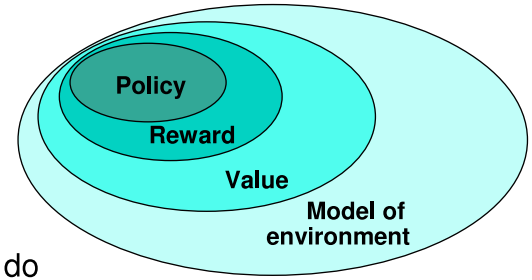
- Learner is not told which actions to take
- Trial-and-Error search
- Possibility of delayed reward (sacrifice short-term gains for greater long-term gains)
- The need to ***explore*** and ***exploit***
- Considers the whole problem of a goal-directed agent interacting with an uncertain environment

Complete Agent

- Temporally situated
- Continual learning and planning
- Object is to ***affect*** the environment
- Environment is stochastic and uncertain

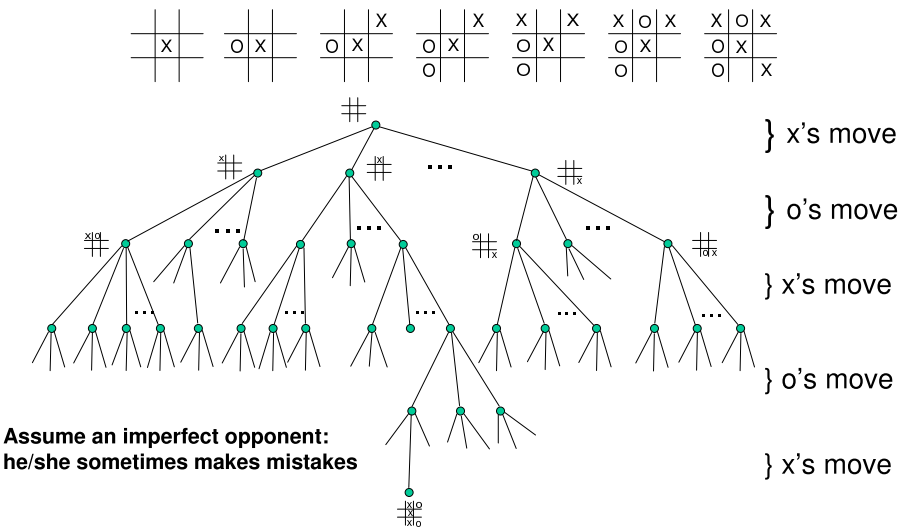


Elements of RL



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

An Extended Example: Tic-Tac-Toe



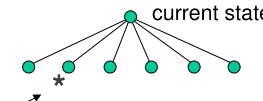
Assume an imperfect opponent:
he/she sometimes makes mistakes

An RL Approach to Tic-Tac-Toe

1. Make a table with one entry per state:

State	$V(s)$ – estimated probability of winning	
	.5	?
	.5	?
	1	win
	0	loss
	0	draw

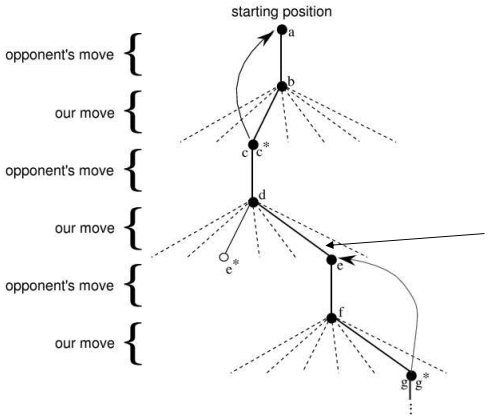
2. Now play lots of games. To pick our moves, look ahead one step:



Just pick the next state with the highest estimated prob. of winning — the largest $V(s)$; a ***greedy*** move.

But 10% of the time pick a move at random; an ***exploratory move.***

RL Learning Rule for Tic-Tac-Toe



"Exploratory" move

s – *the state before our greedy move*

s' – *the state after our greedy move*

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

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

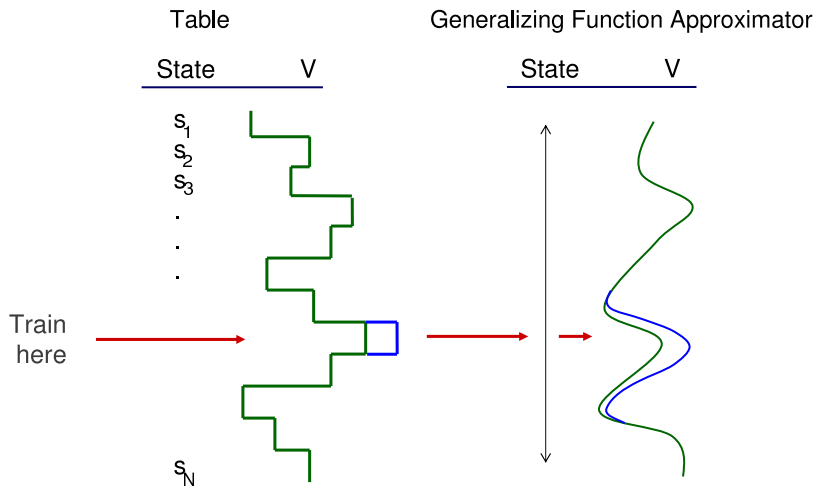
α *a small positive fraction, e.g.,* $\alpha = .1$

the ***step - size parameter***

How can we improve this T.T.T. player?

- Take advantage of symmetries
  - representation/generalization
  - How might this backfire?
- Do we need "random" moves? Why?
  - Do we always need a full 10%?
- Can we learn from "random" moves?
- Can we learn offline?
  - Pre-training from self play?
  - Using learned models of opponent?
- . . .

e.g. Generalization



How is Tic-Tac-Toe Too Easy?

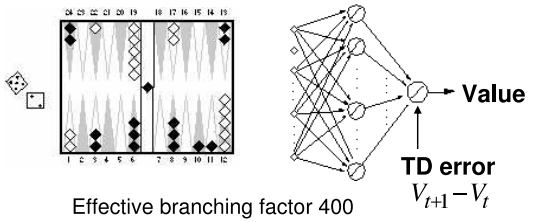
- Finite, small number of states
- One-step look-ahead is always possible
- State completely observable
- …

Some Notable RL Applications

- **TD-Gammon**: Tesauro
  - world's best backgammon program
- **Elevator Control**: Crites & Barto
  - high performance down-peak elevator controller
- **Dynamic Channel Assignment**: Singh & Bertsekas, Nie & Haykin
  - high performance assignment of radio channels to mobile telephone calls
- …

TD-Gammon

Tesauro, 1992–1995



Effective branching factor 400

Action selection by 2–3 ply search

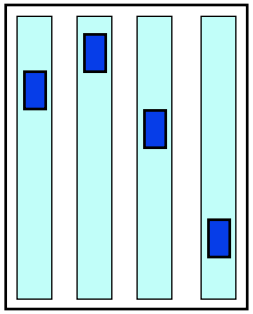
Start with a random network
Play very many games against self
Learn a value function from this simulated experience

This produces arguably the best player in the world

Elevator Dispatching

Crites and Barto, 1996

10 floors, 4 elevator cars



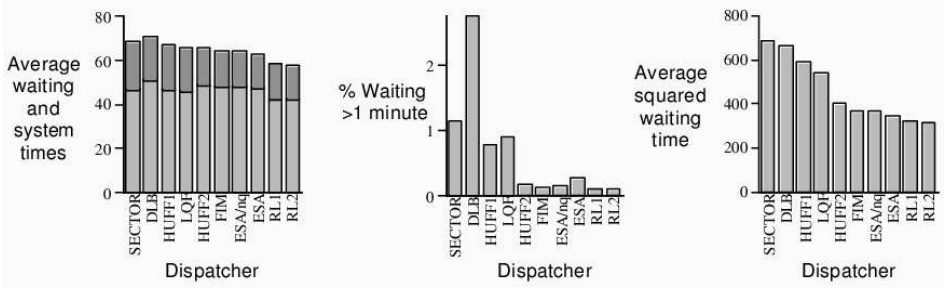
STATES: button states; positions, directions, and motion states of cars; passengers in cars & in halls

ACTIONS: stop at, or go by, next floor

REWARDS: roughly, –1 per time step for each person waiting

Conservatively about 10^{22} states

Performance Comparison



Evaluative Feedback

- **Evaluating** actions vs. **instructing** by giving correct actions
- Pure evaluative feedback depends totally on the action taken. Pure instructive feedback depends not at all on the action taken.
- Supervised learning is instructive; optimization is evaluative
- **Associative** vs. **Nonassociative**:
  - Associative: inputs mapped to outputs; learn the best output **for each** input
  - Nonassociative: "learn" (find) one best output
- *n*-armed bandit (at least how we treat it) is:
  - Nonassociative
  - Evaluative feedback

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

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

These are unknown **action values**
Distribution of r_t depends only on a_t

- Objective is to maximize the reward in the long term, e.g., over 1000 plays

To solve the *n*-armed bandit problem, you must **explore** a variety of actions and then **exploit** the best of them.

The Exploration/Exploitation Dilemma

- Suppose you form estimates

$$Q_t(a) \approx Q^*(a)$$ **action value estimates**

- 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 should always reduce exploring

Action-Value Methods

- Methods that adapt action-value estimates and nothing else, e.g.: 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

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

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

ε-Greedy Action Selection

- Greedy action selection:

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

- ε-Greedy:

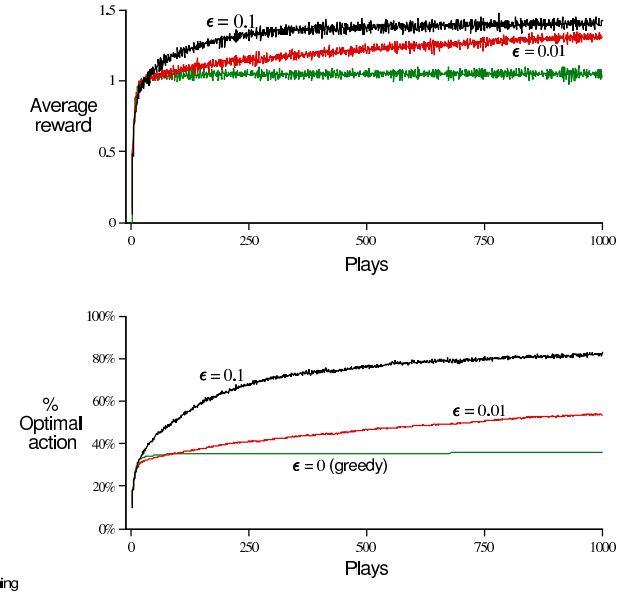
$$a_t = \begin{cases} a_t^* & \text{with probability } 1-\varepsilon \\ \text{random action} & \text{with probability } \varepsilon \end{cases}$$

... the simplest way to try to balance exploration and exploitation

10-Armed Testbed

- n = 10 possible actions
- Each $Q^*(a)$ is chosen randomly from a normal distribution: $N(0,1)$
- each r_t is also normal: $N(Q^*(a_t),1)$
- 1000 plays
- repeat the whole thing 2000 times and average the results
- *Evaluative versus instructive feedback*

ε-Greedy Methods on the 10-Armed Testbed



Softmax Action Selection

- Softmax action selection methods grade action probs. by estimated values.
- The most common softmax uses a Gibbs, or Boltzmann, distribution:

Choose action a on play t with probability

$$\frac{e^{Q_t(a)/\tau}}{\sum_{b=1}^{n} e^{Q_t(b)/\tau}},$$

where τ is the "computational temperature"

Evaluation Versus Instruction

- Suppose there are K possible actions and you select action number k.
- Evaluative feedback would give you a single score f, say 7.2.
- Instructive information, on the other hand, would say that action k', which is eventually different from action k, have actually been correct.
- Obviously, instructive feedback is much more informative, (even if it is noisy).

Binary Bandit Tasks

Suppose you have just **two** actions: $a_t = 1$ or $a_t = 2$

and just **two** rewards: $r_t = success$ or $r_t = failure$

Then you might infer a **target** or **desired action**:

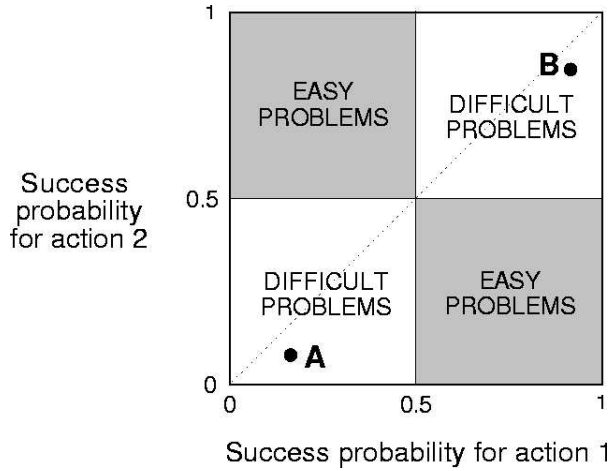
$$d_t = \begin{cases} a_t & \text{if } success \\ \text{the other action} & \text{if } failure \end{cases}$$

and then always play the action that was most often the target

Call this the **supervised algorithm**.
It works fine on deterministic tasks but is suboptimal if the rewards are stochastic.

Contingency Space

The space of all possible binary bandit tasks:



Linear Learning Automata

Let $\pi_t(a) = \Pr\{a_t = a\}$ be the only adapted parameter

L_{R-I} (Linear, reward - inaction)

On *success* : $\pi_{t+1}(a_t) = \pi_t(a_t) + \alpha(1 - \pi_t(a_t)) \qquad 0 < \alpha < 1$

(the other action probs. are adjusted to still sum to 1)

On *failure* : no change

L_{R-P} (Linear, reward - penalty)

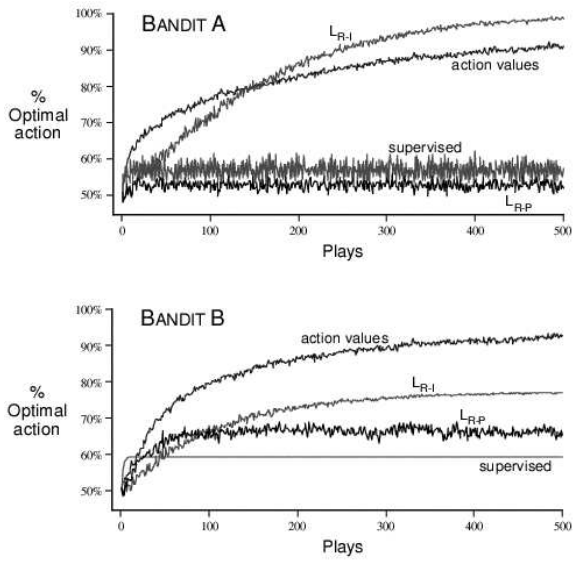
On *success* : $\pi_{t+1}(a_t) = \pi_t(a_t) + \alpha(1 - \pi_t(a_t)) \qquad 0 < \alpha < 1$

(the other action probs. are adjusted to still sum to 1)

On *failure* : $\pi_{t+1}(a_t) = \pi_t(a_t) + \alpha(0 - \pi_t(a_t)) \qquad 0 < \alpha < 1$

For two actions, a stochastic, incremental version of the supervised algorithm

Performance on Binary Bandit Tasks A and B



Incremental Implementation

Recall the sample average estimation method:

The average of the first k rewards is (dropping the dependence on a): $Q_k = \dfrac{r_1 + r_2 + \cdots r_k}{k}$

Can we do this incrementally (without storing all the rewards)?

We could keep a running sum and count, or, equivalently:

$$Q_{k+1} = Q_k + \frac{1}{k+1}\left[r_{k+1} - Q_k\right]$$

This is a common form for update rules:

NewEstimate = *OldEstimate* + *StepSize*[*Target* – *OldEstimate*]

Computation

$$
\begin{aligned}
Q_{k+1} &= \frac{1}{k+1}\sum_{i=1}^{k+1} r_i \\
&= \frac{1}{k+1}\left(r_{k+1} + \sum_{i=1}^{k} r_i\right) \\
&= \frac{1}{k+1}\left(r_{k+1} + kQ_k + Q_k - Q_k\right) \\
&= Q_k + \frac{1}{k+1}\left[r_{k+1} - Q_k\right]
\end{aligned}
$$

Stepsize constant or changing with time

Tracking a Non-stationary Problem

Choosing Q_k to be a sample average is appropriate in a stationary problem, i.e., when none of the $Q^*(a)$ change over time,

But not in a non-stationary problem.

Better in the non-stationary case is:

$$Q_{k+1} = Q_k + \alpha\left[r_{k+1} - Q_k\right]$$

for *constant* α, $0 < \alpha \le 1$

$$= (1-\alpha)^k Q_0 + \sum_{i=1}^{k} \alpha(1-\alpha)^{k-i} r_i$$

exponential, recency-weighted average

Computation

Use

$$Q_{k+1} = Q_k + \alpha[r_{k+1} - Q_k]$$

Then

$$
\begin{aligned}
Q_k &= Q_{k-1} + \alpha[r_k - Q_{k-1}] \\
&= \alpha r_k + (1-\alpha)Q_{k-1} \\
&= \alpha r_k + (1-\alpha)\alpha r_{k-1} + (1-\alpha)^2 Q_{k-2} \\
&= (1-\alpha)^k Q_0 + \sum_{i=1}^{k} \alpha(1-\alpha)^{k-i} r_i
\end{aligned}
$$

In general : convergence if

$$\sum_{k=1}^{\infty} \alpha_k(a) = \infty \text{ and } \sum_{k=1}^{\infty} \alpha_k^2(a) < \infty$$

satisfied for $\alpha_k = \frac{1}{k}$ but not for fixed α

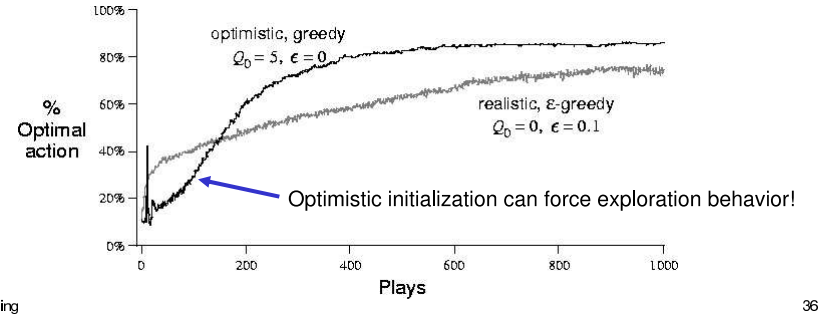
Notes:

1. $\sum_{i=1}^{k} \alpha(1-\alpha)^{k-i} = 1$

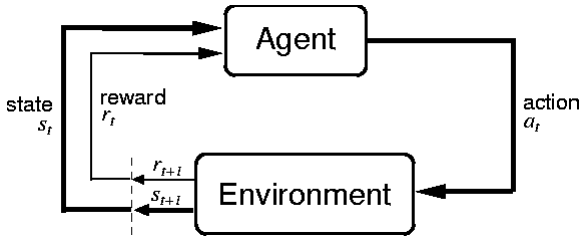
2. Step size parameter after the k-th application of action a

Optimistic Initial Values

- All methods so far depend on $Q_0(a)$, i.e., they are **biased.**
- Suppose instead we initialize the action values **optimistically,** i.e., on the 10-armed testbed, use $Q_0(a) = 5$ for all a.



The Agent-Environment Interface



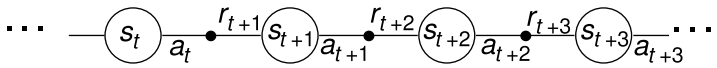
Agent and environment interact at discrete time steps : $t = 0,1,2,\ldots$

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}



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 experience.
- Roughly, the agent's goal is to get as much reward as it can over the long run.

Getting the Degree of Abstraction Right

- Time steps need not refer to fixed intervals of real time.
- Actions can be low level (e.g., voltages to motors), or high level (e.g., accept a job offer), "mental" (e.g., shift in focus of attention), etc.
- States can low-level "sensations", or they can be abstract, symbolic, based on memory, or subjective (e.g., the state of being "surprised" or "lost").
- An RL agent is not like a whole animal or robot, which consist of many RL agents as well as other components.
- The environment is not necessarily unknown to the agent, only incompletely controllable.
- Reward computation is in the agent's environment because the agent cannot change it arbitrarily.

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 must be outside the agent's direct control—thus outside the agent.
- The agent must be able to measure success:
  - explicitly;
  - frequently during its lifespan.

Returns

Suppose the sequence of rewards after step t is:

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

What do we want to maximize?

In general, we want to maximize the **expeted 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.

Returns for Continuing Tasks

Continuing tasks: interaction does not have natural episodes.

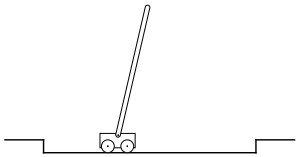
Discounted return:

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

where $\gamma, 0 \le \gamma \le 1,$ is the **discount rate**.

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

An Example



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:

reward $= +1$ for each step before failure

$\Rightarrow$ return $=$ number of steps before failure

As a **continuing task** with discounted return:

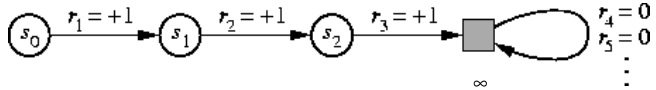
reward $= -1$ upon failure; 0 otherwise

$\Rightarrow$ return $= -\gamma^k$, for k steps before failure

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

A Unified Notation

- In episodic tasks, we number the time steps of each episode starting from zero.
- We usually do not have distinguish between episodes, so we write s_t instead of $s_{t,j}$ for the state at step t of episode j.
- Think of each episode as ending in an absorbing state that always produces reward of zero:



- We can cover all cases by writing $R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$,

where γ can be 1 only if a zero reward absorbing state is always reached.

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, i.e., it should have the **Markov Property**:

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

for all s', r, and histories s_t, a_t, s_{t-1}, a_{t-1}, …, r_1, s_0, a_0.

Markov Decision Processes

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

$$P^a_{ss'} = \Pr\{s_{t+1}=s' \mid s_t=s, a_t=a\} \quad \text{for all } s, s' \in S,\ a \in A(s).$$

  - **reward probabilities**:

$$R^a_{ss'} = E\{r_{t+1} \mid s_t=s, a_t=a, s_{t+1}=s'\} \quad \text{for all } s, s' \in S,\ a \in A(s).$$

An Example Finite MDP

Recycling Robot

- At each step, robot has to decide whether it should (1) actively search for a can, (2) wait for someone to bring it a can, or (3) 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

Recycling Robot MDP

$S = \{\text{high}, \text{low}\}$

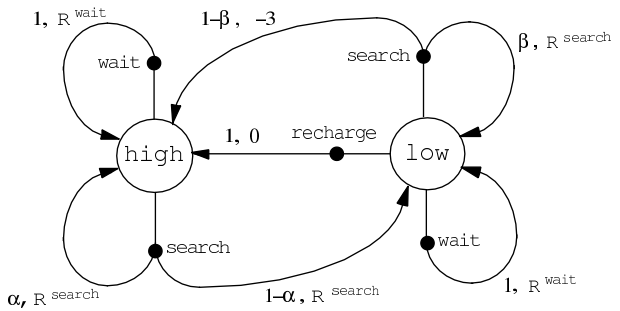
$A(\text{high}) = \{\text{search}, \text{wait}\}$

$A(\text{low}) = \{\text{search}, \text{wait}, \text{recharge}\}$

$R^{\text{search}} = \textit{expected no. of cans while searching}$

$R^{\text{wait}} = \textit{expected no. of cans while waiting}$

$R^{\text{search}} > R^{\text{wait}}$



Transition Table

Table 3.1 Transition probabilities and expected rewards for the finite MDP of the recycling robot example.

s	s'	a	$\mathcal{P}^a_{ss'}$	$\mathcal{R}^a_{ss'}$
high	high	search	α	$\mathcal{R}^{\text{search}}$
high	low	search	$1-\alpha$	$\mathcal{R}^{\text{search}}$
low	high	search	$1-\beta$	-3
low	low	search	β	$\mathcal{R}^{\text{search}}$
high	high	wait	1	$\mathcal{R}^{\text{wait}}$
high	low	wait	0	$\mathcal{R}^{\text{wait}}$
low	high	wait	0	$\mathcal{R}^{\text{wait}}$
low	low	wait	1	$\mathcal{R}^{\text{wait}}$
low	high	recharge	1	0
low	low	recharge	0	0

Note: There is a row for each possible combination of current state, s, next state, s', and action possible in the current state, $a \in \mathcal{A}(s)$.

Value Functions

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

State - value function for policy π :

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

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

Action - value function for policy π :

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

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:

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

Or, without the expectation operator:

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

Derivation

Derivation

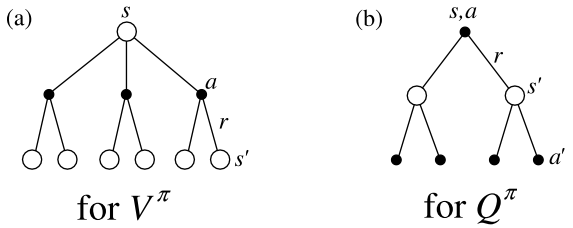
$$
\begin{aligned}
V^\pi(s) &= E_\pi\{R_t \mid s_t = s\} \\
&= E_\pi\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \,\middle|\, s_t = s\right\} \\
&= E_\pi\left\{r_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \,\middle|\, s_t = s\right\} \\
&= \sum_a \pi(s,a) \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma E_\pi\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+2} \,\middle|\, s_{t+1} = s'\right\}\right] \\
&= \sum_a \pi(s,a) \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V^\pi(s')\right],
\end{aligned}
$$

More on the Bellman Equation

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

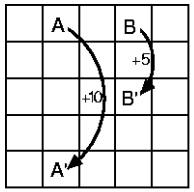
This is a set of equations (in fact, linear), one for each state. The value function for π is its unique solution.

Backup diagrams:



Grid World

- Actions: `north`, `south`, `east`, `west`; deterministic.
- If would take agent off the grid: no move but reward = –1
- Other actions produce reward = 0, except actions that move agent out of special states A and B as shown.



(a)

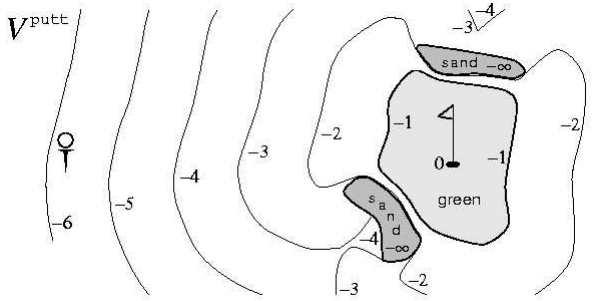
3.3	8.8	4.4	5.3	1.5
1.5	3.0	2.3	1.9	0.5
0.1	0.7	0.7	0.4	-0.4
-1.0	-0.4	-0.4	-0.6	-1.2
-1.9	-1.3	-1.2	-1.4	-2.0

(b)

State-value function for equiprobable random policy; $\gamma = 0.9$

Golf

- State is ball location
- Reward of -1 for each stroke until the ball is in the hole
- Value of a state?
- Actions:
  - `putt` (use putter)
  - `driver` (use driver)
- `putt` succeeds anywhere on the green



Optimal Value Functions

- For finite MDPs, policies can be **partially ordered**:

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

- There is always at least one (and possibly many) policies that is better than or equal to all the others. This is an **optimal policy**. We denote them all π^*.
- Optimal policies share the same **optimal state-value function**:

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

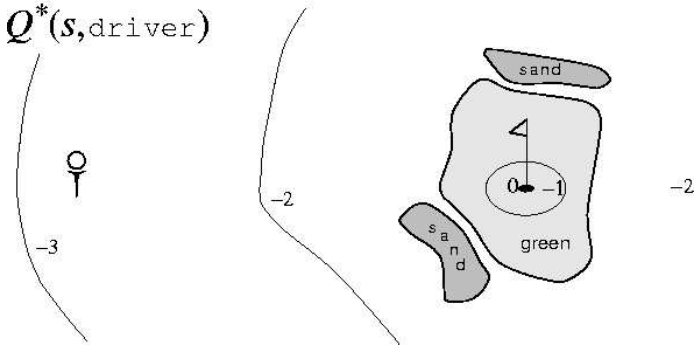
- Optimal policies also share the same **optimal action-value function**:

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

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

Optimal Value Function for Golf

- We can hit the ball farther with `driver` than with `putter`, but with less accuracy
- $Q^*(s,$`driver`$)$ gives the value of using `driver` first, then using whichever actions are best

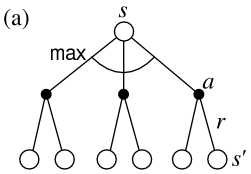


Bellman Optimality Equation for V^*

The value of a state under an optimal policy must equal the expected return for the best action from that state:

$$\begin{aligned} V^*(s) &= \max_{a \in A(s)} Q^{\pi^*}(s,a) \\ &= \max_{a \in A(s)} E\{r_{t+1} + \gamma V^*(s_{t+1}) \mid s_t = s, a_t = a\} \\ &= \max_{a \in A(s)} \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V^*(s') \right] \end{aligned}$$

The relevant backup diagram:

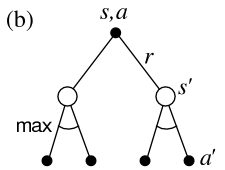


V^* is the unique solution of this system of nonlinear equations.

Bellman Optimality Equation for Q^*

$$\begin{aligned} Q^*(s,a) &= E\left\{ r_{t+1} + \gamma \max_{a'} Q^*(s_{t+1}, a') \mid s_t = s, a_t = a \right\} \\ &= \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma \max_{a'} Q^*(s', a') \right] \end{aligned}$$

The relevant backup diagram:



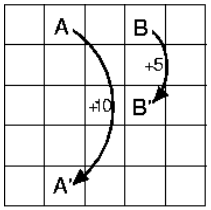
Q^* is the unique solution of this system of nonlinear equations.

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.

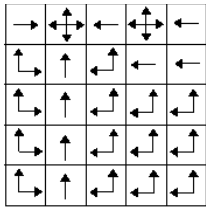
E.g., back to the grid world:



a) gridworld

22.0	24.4	22.0	19.4	17.5
19.8	22.0	19.8	17.8	16.0
17.8	19.8	17.8	16.0	14.4
16.0	17.8	16.0	14.4	13.0
14.4	16.0	14.4	13.0	11.7

b) V^*



c) π^*

What About Optimal Action-Value Functions?

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

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

Solving the Bellman Optimality Equation

- Finding an optimal policy by solving the Bellman Optimality Equation requires the following:
  - accurate knowledge of environment dynamics;
  - we have enough space and 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; see later),
  - BUT, number of states is often huge (e.g., backgammon has about 10^{20} states).
- We usually have to settle for approximations.
- Many RL methods can be understood as approximately solving the Bellman Optimality Equation.

A Summary

- Agent-environment interaction
  - States
  - Actions
  - Rewards
- Policy: stochastic rule for selecting actions
- Return: the function of future rewards the agent tries to maximize
- Episodic and continuing tasks
- Markov Property
- Markov Decision Process
  - Transition probabilities
  - Expected rewards
- Value functions
  - State-value function for a policy
  - Action-value function for a policy
  - Optimal state-value function
  - Optimal action-value function
- Optimal value functions
- Optimal policies
- Bellman Equations
- The need for approximation

Dynamic Programming

Objectives of the next slides:

- Overview of a collection of classical solution methods for MDPs known as dynamic programming (DP)
- Show how DP can be used to compute value functions, and hence, optimal policies
- Discuss efficiency and utility of DP

Policy Evaluation

Policy Evaluation: for a given policy π, compute the state-value function V^π

Recall: State value function for policy π:

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

Bellman equation for V^*:

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

A system of $|S|$ simultaneous linear equations

Iterative Methods

$$V_0 \to V_1 \to \cdots \to V_k \to V_{k+1} \to \cdots \to V^\pi$$

a "sweep"

A sweep consists of applying a **backup operation** to each state.

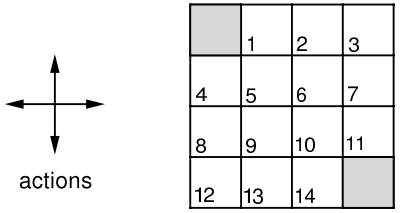
A **full policy evaluation backup**:

$$V_{k+1}(s) \leftarrow \sum_a \pi(s,a) \sum_{s'} P^a_{ss'}\left[R^a_{ss'} + \gamma V_k(s')\right]$$

Iterative Policy Evaluation

Input π, the policy to be evaluated
Initialize $V(s) = 0$, for all $s \in \mathcal{S}^+$
Repeat
  $\Delta \leftarrow 0$
  For each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \sum_a \pi(s,a) \sum_{s'} \mathcal{P}^a_{ss'}\left[\mathcal{R}^a_{ss'} + \gamma V(s')\right]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)
Output $V \approx V^\pi$

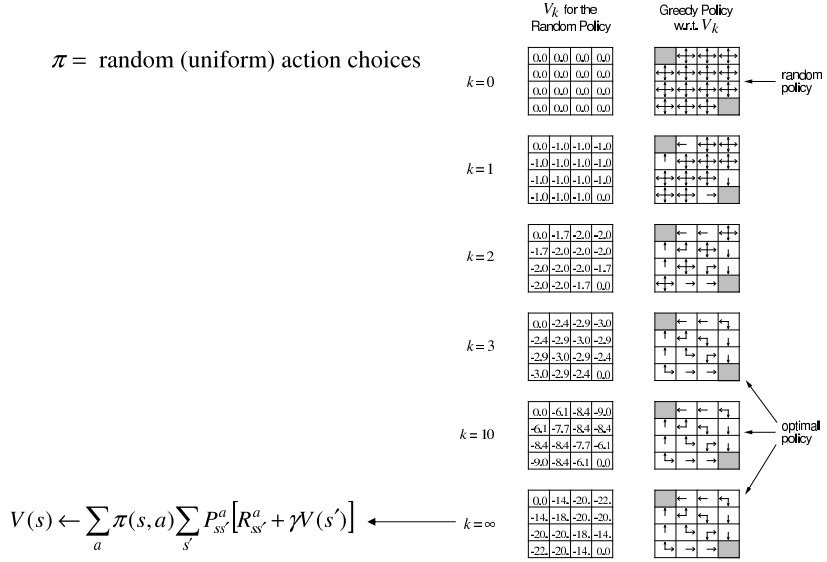
A Small Gridworld



$r = -1$
on all transitions

- An undiscounted episodic task
- Nonterminal states: 1, 2, . . ., 14;
- One terminal state (shown twice as shaded squares)
- Actions that would take agent off the grid leave state unchanged
- Reward is –1 until the terminal state is reached

Iterative Policy Evaluation for the Small Gridworld



Policy Improvement

Suppose we have computed V^* for a **deterministic** policy π.

For a given state s, would it be better to do an action $a \neq \pi(s)$?

The value of doing a in state s is:

$$Q^\pi(s,a) = E_\pi\{r_{t+1} + \gamma V^\pi(s_{t+1}) \mid s_t = s, a_t = a\}$$
$$= \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V^\pi(s') \right]$$

It is better to switch to action a for state s if and only if
$Q^\pi(s,a) > V^\pi(s)$

The Policy Improvement Theorem

Let π and π' be two policies such that for all $s \in S$:

$$Q^\pi(s, \pi'(s)) \geq V^\pi(s)$$

Then π' is a better policy than π, i.e. for all $s \in S$:

$$V^{\pi'}(s) \geq V^\pi(s)$$

Proof sketch

$$
\begin{aligned}
V^{\pi}(s) &\le Q^{\pi}(s, \pi'(s)) \\
&= E_{\pi'}\{r_{t+1} + \gamma V^{\pi}(s_{t+1}) \mid s_t = s\} \\
&\le E_{\pi'}\{r_{t+1} + \gamma Q^{\pi}(s_{t+1}, \pi'(s_{t+1})) \mid s_t = s\} \\
&= E_{\pi'}\{r_{t+1} + \gamma E_{\pi'}\{r_{t+2} + \gamma V^{\pi}(s_{t+2})\} \mid s_t = s\} \\
&= E_{\pi'}\{r_{t+1} + \gamma r_{t+2} + \gamma^2 V^{\pi}(s_{t+2}) \mid s_t = s\} \\
&\le E_{\pi'}\{r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 V^{\pi}(s_{t+3}) \mid s_t = s\} \\
&\vdots \\
&\le E_{\pi'}\{r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} + \cdots \mid s_t = s\} \\
&= V^{\pi'}(s).
\end{aligned}
$$

Policy Improvement Cont.

Do this for all states to get a new policy π' that is **greedy** with respect to V^{π} :

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

Then $V^{\pi'} \ge V^{\pi}$

Policy Improvement Cont.

What if $V^{\pi'} = V^{\pi}$?

i.e., for all $s \in S$, $V^{\pi'}(s) = \max_a \sum_{s'} P^a_{ss'} \left[R^a_{ss'} + \gamma V^{\pi}(s') \right]$?

But this is the Bellman Optimality Equation.

So $V^{\pi'} = V^*$ and both π and π' are optimal policies.

Policy Iteration

$$
\pi_0 \rightarrow V^{\pi_0} \rightarrow \pi_1 \rightarrow V^{\pi_1} \rightarrow \cdots \pi^* \rightarrow V^* \rightarrow \pi^*
$$

policy evaluation

policy improvement
"greedification"

Policy Iteration

1. Initialization
   $V(s) \in \Re$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Repeat
      $\Delta \leftarrow 0$
      For each $s \in \mathcal{S}$:
         $v \leftarrow V(s)$
         $V(s) \leftarrow \sum_{s'} \mathcal{P}_{ss'}^{\pi(s)} \left[\mathcal{R}_{ss'}^{\pi(s)} + \gamma V(s')\right]$
         $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number)

3. Policy Improvement
   policy-stable $\leftarrow$ true
   For each $s \in \mathcal{S}$:
      $b \leftarrow \pi(s)$
      $\pi(s) \leftarrow \arg\max_a \sum_{s'} \mathcal{P}_{ss'}^{a} \left[\mathcal{R}_{ss'}^{a} + \gamma V(s')\right]$
      If $b \neq \pi(s)$, then *policy-stable* $\leftarrow$ false
   If *policy-stable*, then stop; else go to 2

Value Iteration

- **Drawback to policy iteration** is that **each iteration involves a policy evaluation**, which itself may require multiple sweeps.
- **Convergence of V^{π} occurs only in the limit** so that we in principle have to wait until convergence.
- As we have seen, the **optimal policy is often obtained long before V^{π} has converged.**
- Fortunately, the **policy evaluation** step **can be truncated in several ways without losing the convergence guarantees** of policy iteration.
- **Value iteration** is to **stop policy evaluation after just one sweep.**

Value Iteration

Recall the **full policy evaluation backup**:

$$V_{k+1}(s) \leftarrow \sum_{a} \pi(s,a) \sum_{s'} P_{ss'}^{a} \left[R_{ss'}^{a} + \gamma V_k(s')\right]$$

Here is the **full value iteration backup**:

$$V_{k+1}(s) \leftarrow \max_{a} \sum_{s'} P_{ss'}^{a} \left[R_{ss'}^{a} + \gamma V_k(s')\right]$$

Combination of policy improvement and truncated policy evaluation.

Value Iteration Cont.

Initialize V arbitrarily, e.g., $V(s) = 0$, for all $s \in \mathcal{S}^{+}$

Repeat
   $\Delta \leftarrow 0$
   For each $s \in \mathcal{S}$:
      $v \leftarrow V(s)$
      $V(s) \leftarrow \max_a \sum_{s'} \mathcal{P}_{ss'}^{a} \left[\mathcal{R}_{ss'}^{a} + \gamma V(s')\right]$
      $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)

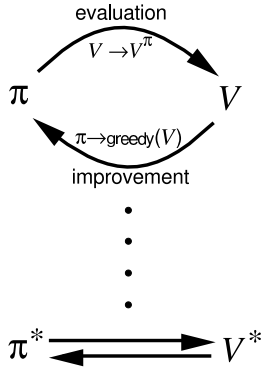
Output a deterministic policy, π, such that
   $\pi(s) = \arg\max_a \sum_{s'} \mathcal{P}_{ss'}^{a} \left[\mathcal{R}_{ss'}^{a} + \gamma V(s')\right]$

Asynchronous DP

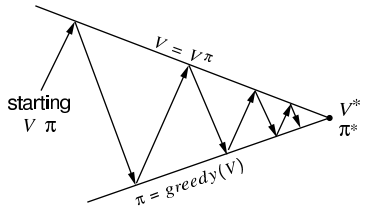
- All the DP methods described so far require exhaustive sweeps of the entire state set.
- Asynchronous DP does not use sweeps. Instead it works like this:
  - Repeat until convergence criterion is met: Pick a state at random and apply the appropriate backup
- Still needs lots of computation, but does not get locked into hopelessly long sweeps
- Can you select states to backup intelligently? YES: an agent's experience can act as a guide.

Generalized Policy Iteration

Generalized Policy Iteration (GPI):
any interaction of policy evaluation and policy improvement, independent of their granularity.



A geometric metaphor for convergence of GPI:



Efficiency of DP

- To find an optimal policy is polynomial in the number of states…
- BUT, the number of states is often astronomical, e.g., often growing exponentially with the number of state variables (what Bellman called "the curse of dimensionality").
- In practice, classical DP can be applied to problems with a few millions of states.
- Asynchronous DP can be applied to larger problems, and appropriate for parallel computation.
- It is surprisingly easy to come up with MDPs for which DP methods are not practical.

Summary

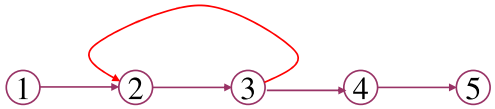
- Policy evaluation: backups without a max
- Policy improvement: form a greedy policy, if only locally
- Policy iteration: alternate the above two processes
- Value iteration: backups with a max
- Full backups (to be contrasted later with sample backups)
- Generalized Policy Iteration (GPI)
- Asynchronous DP: a way to avoid exhaustive sweeps
- **Bootstrapping**: updating estimates based on other estimates

Monte Carlo Methods

- Monte Carlo methods learn from *complete* sample returns
  - Only defined for episodic tasks
- Monte Carlo methods learn directly from experience
  - *On-line:* No model necessary and still attains optimality
  - *Simulated:* No need for a *full* model

Monte Carlo Policy Evaluation

- *Goal:* learn $V^{\pi}(s)$
- *Given:* some number of episodes under π 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

First-visit Monte Carlo policy evaluation

Initialize:

$\pi \leftarrow$ policy to be evaluated

$V \leftarrow$ an arbitrary state-value function

$Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Repeat forever:

(a) Generate an episode using π

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

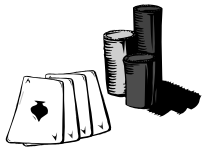
$R \leftarrow$ return following the first occurrence of s

Append R to $Returns(s)$

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

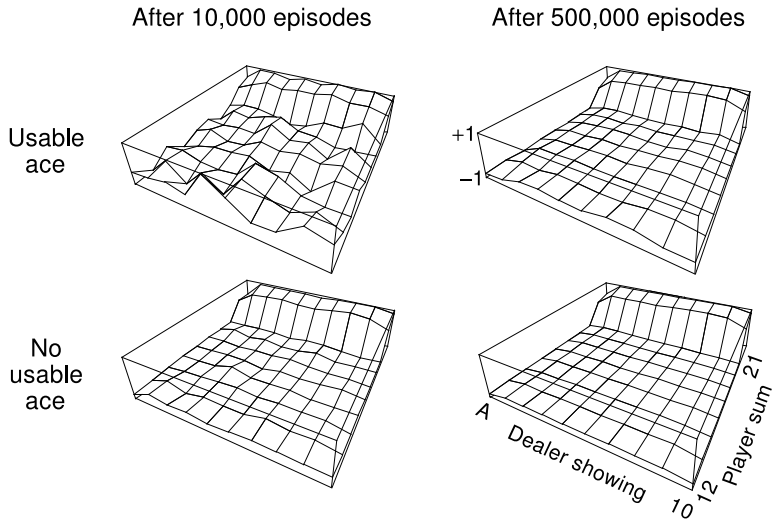
Blackjack example

- *Object:* Have your card sum be greater than the dealers without exceeding 21.
- *States* (200 of them):
  - current sum (12-21)
  - dealer's showing card (ace-10)
  - do I have a useable ace?



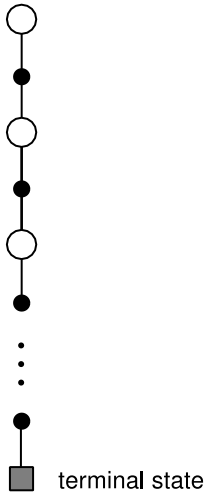
- *Reward:* +1 for winning, 0 for a draw, -1 for losing
- *Actions:* stick (stop receiving cards), hit (receive another card)
- *Policy:* Stick if my sum is 20 or 21, else hit

Blackjack value functions



Backup diagram for Monte Carlo

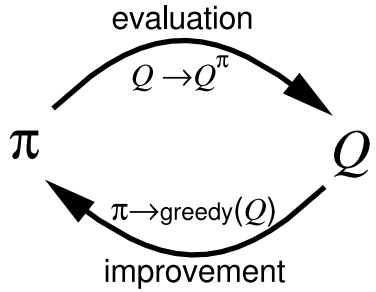
- Entire episode included
- Only one choice at each state (unlike DP)
- MC does not bootstrap
- Time required to estimate one state does not depend on the total number of states



Monte Carlo Estimation of Action Values (Q)

- Monte Carlo is most useful when a model is not available
  - We want to learn Q^*
- $Q^\pi(s,a)$ - average return starting from state s and action a following π
- Also converges asymptotically *if* every state-action pair is visited
- *Exploring starts:* Every state-action pair has a non-zero probability of being the starting pair

Monte Carlo Control



- MC policy iteration: Policy evaluation using MC methods followed by policy improvement
- Policy improvement step: greedify with respect to value (or action-value) function

Monte Carlo Exploring Starts

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
$Q(s,a) \leftarrow$ arbitrary
$\pi(s) \leftarrow$ arbitrary
$Returns(s,a) \leftarrow$ empty list

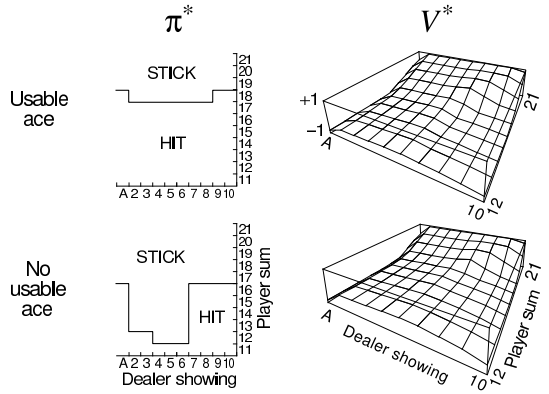
Repeat forever:
(a) Generate an episode using exploring starts and π
(b) For each pair s, a appearing in the episode:
$R \leftarrow$ return following the first occurrence of s, a
Append R to $Returns(s,a)$
$Q(s,a) \leftarrow$ average$(Returns(s,a))$
(c) For each s in the episode:
$\pi(s) \leftarrow \arg\max_a Q(s,a)$

Fixed point is optimal policy π^*

Proof is open question

Blackjack Example Continued

- Exploring starts
- Initial policy as described before



On-policy Monte Carlo Control

- *On-policy:* learn about policy currently executing
- How do we get rid of exploring starts?
  - Need *soft* policies: $\pi(s,a) > 0$ for all s and a
  - e.g. ε-soft policy:

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

non-max greedy

- Similar to GPI: move policy *towards* greedy policy (i.e. ε-soft)
- Converges to best ε-soft policy

On-policy MC Control

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
$Q(s,a) \leftarrow$ arbitrary
$Returns(s,a) \leftarrow$ empty list
$\pi \leftarrow$ an arbitrary ϵ-soft policy

Repeat forever:
(a) Generate an episode using π
(b) For each pair s, a appearing in the episode:
$R \leftarrow$ return following the first occurrence of s, a
Append R to $Returns(s,a)$
$Q(s,a) \leftarrow$ average$(Returns(s,a))$
(c) For each s in the episode:
$a^* \leftarrow \arg\max_a Q(s,a)$
For all $a \in \mathcal{A}(s)$:

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

Off-policy Monte Carlo control

- Behavior policy generates behavior in environment
- Estimation policy is policy being learned about
- Weight returns from behavior policy by their relative probability of occurring under the behavior and estimation policy

Off-policy MC control

Initialize, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$:
$Q(s,a) \leftarrow$ arbitrary
$N(s,a) \leftarrow 0$; Numerator and
$D(s,a) \leftarrow 0$; Denominator of $Q(s,a)$
$\pi \leftarrow$ an arbitrary deterministic policy

Repeat forever:
(a) Select a policy π' and use it to generate an episode:
$s_0, a_0, r_1, s_1, a_1, r_2, \ldots, s_{T-1}, a_{T-1}, r_T, s_T$
(b) $\tau \leftarrow$ latest time at which $a_\tau \neq \pi(s_\tau)$
(c) For each pair s, a appearing in the episode at time τ or later:
$t \leftarrow$ the time of first occurrence of s, a such that $t \geq \tau$
$w \leftarrow \prod_{k=t+1}^{T-1} \frac{1}{\pi'(s_k,a_k)}$
$N(s,a) \leftarrow N(s,a) + wR_t$
$D(s,a) \leftarrow D(s,a) + w$
$Q(s,a) \leftarrow \frac{N(s,a)}{D(s,a)}$
(d) For each $s \in \mathcal{S}$:
$\pi(s) \leftarrow \arg\max_a Q(s,a)$

Summary about Monte Carlo Techniques

- MC has several advantages over DP:
  - Can learn directly from interaction with environment
  - No need for full models
  - No need to learn about ALL states
  - Less harm by Markovian violations
  - MC methods provide an alternate policy evaluation process
- One issue to watch for: maintaining sufficient exploration
  - exploring starts, soft policies
- No bootstrapping (as opposed to DP)

Temporal Difference Learning

Objectives of the following slides:

- Introduce Temporal Difference (TD) learning
- Focus first on policy evaluation, or prediction, methods
- Then extend to control methods

TD Prediction

Policy Evaluation (the prediction problem):
for a given policy π, compute the state-value function V^π

Recall: Simple every-visit Monte Carlo method:

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

target: the actual return after time t

The simplest TD method, TD(0):

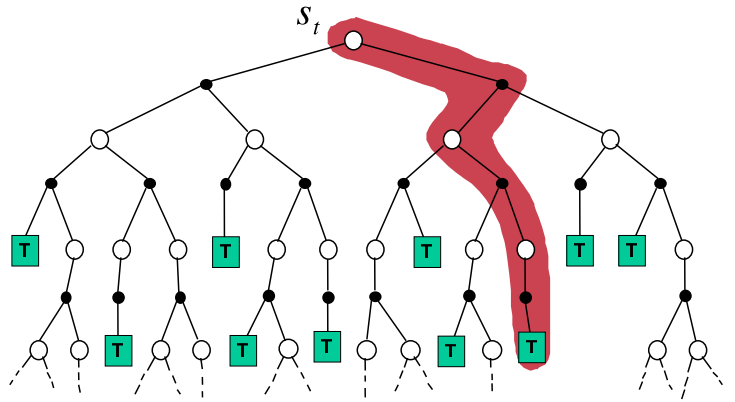
$$V(s_t) \leftarrow V(s_t) + \alpha\left[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)\right]$$

target: an estimate of the return

Simple Monte Carlo

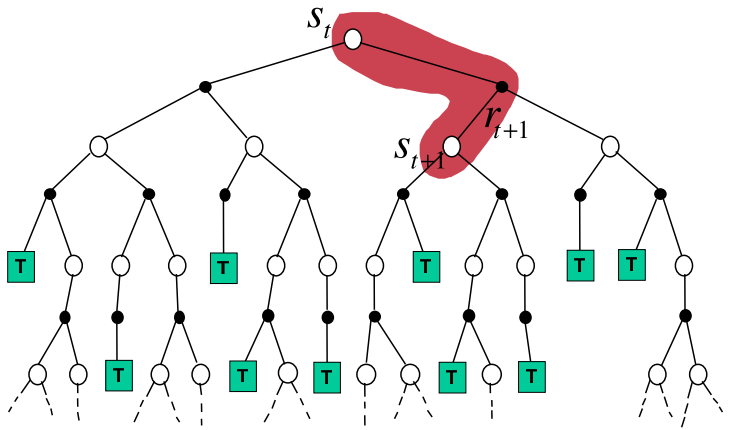
$$V(s_t) \leftarrow V(s_t) + \alpha\left[R_t - V(s_t)\right]$$

where R_t is the actual return following state s_t.



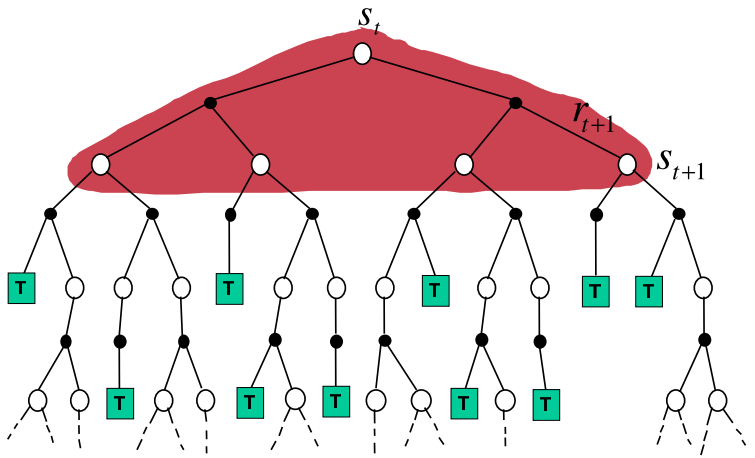
Simplest TD Method

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



cf. Dynamic Programming

$$V(s_t) \leftarrow E_\pi\left\{r_{t+1} + \gamma V(s_t)\right\}$$



TD Bootstraps and Samples

- Bootstrapping: update involves an estimate
  - MC does not bootstrap
  - DP bootstraps
  - TD bootstraps
- Sampling: update does not involve an expected value
  - MC samples
  - DP does not sample
  - TD samples

A Comparison of DP, MC, and TD

	bootstraps	samples
DP	+	-
MC	-	+
TD	+	+

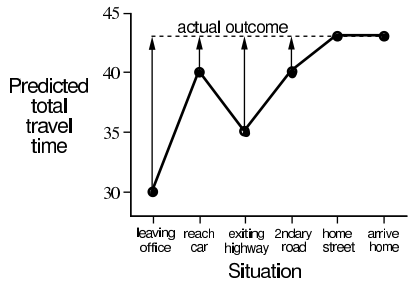
Example: Driving Home

State	Elapsed Time (minutes)	Predicted Time to Go	Predicted Total Time
leaving office	0	30	30
reach car, raining	5	35	40
exit highway	20	15	35
behind truck	30	10	40
home street	40	3	43
arrive home	43	0	43

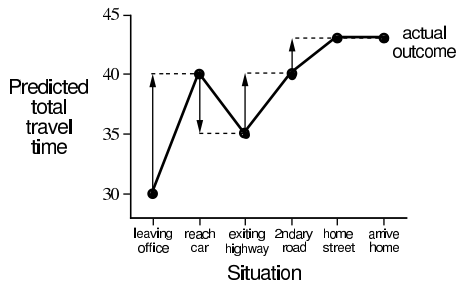
- Value of each state: expected time to go

Driving Home

Changes recommended by Monte Carlo methods (α=1)



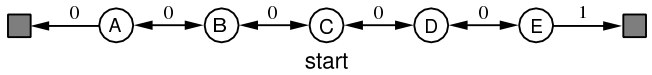
Changes recommended by TD methods (α=1)

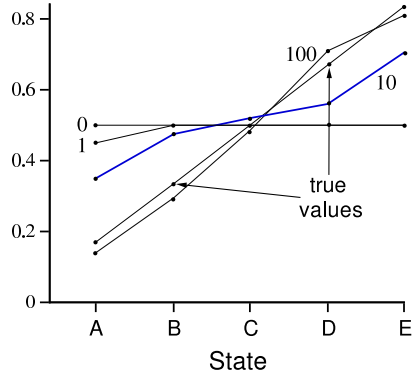


Advantages of TD Learning

- TD methods do not require a model of the environment, only experience
- TD, but not MC, 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
- Both MC and TD converge (under certain assumptions), but which is faster?

Random Walk Example

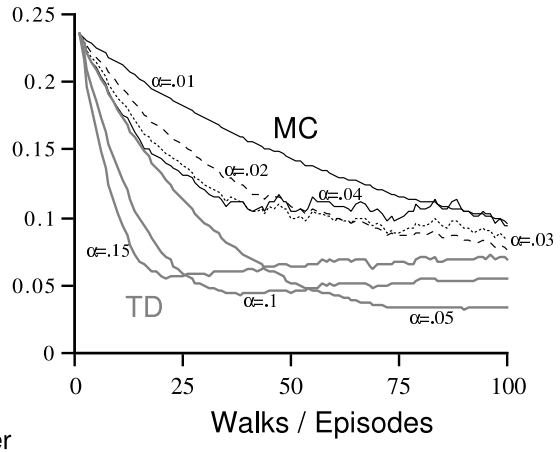




Estimated value

Values learned by TD(0) after various numbers of episodes

TD and MC on the Random Walk



RMS error, averaged over states

Data averaged over 100 sequences of episodes

Optimality of TD(0)

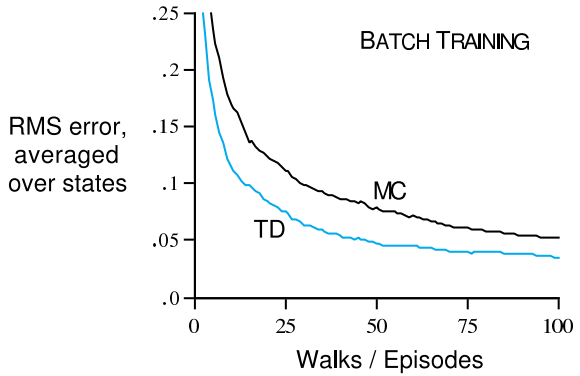
Batch Updating: train completely on a finite amount of data, e.g., train repeatedly on 10 episodes until convergence.

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.

Constant-α MC also converges under these conditions, but to a difference answer!

Random Walk under Batch Updating



After each new episode, all previous episodes were treated as a batch, and algorithm was trained until convergence. All repeated 100 times.

You are the Predictor

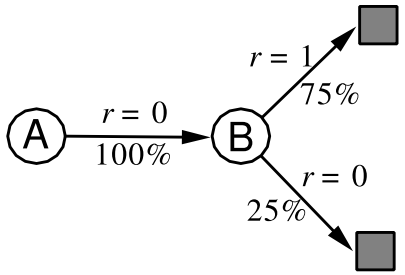
Suppose you observe the following 8 episodes:

A, 0, B, 0
B, 1
B, 1
B, 1
B, 1
B, 1
B, 1
B, 0

$V(\mathrm{A})?$

$V(\mathrm{B})?$

You are the Predictor



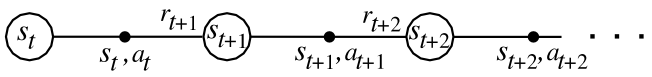
$V(\mathrm{A})?$

You are the Predictor

- The prediction that best matches the training data is V(A)=0
  - This minimizes the mean-square-error on the training set
  - This is what a batch Monte Carlo method gets
- If we consider the sequentiality of the problem, then we would set V(A)=.75
  - This is correct for the maximum likelihood estimate of a Markov model generating the data
  - i.e, if we do a best fit Markov model, and assume it is exactly correct, and then compute what it predicts
  - This is called the certainty-equivalence estimate
  - This is what TD(0) gets

Learning an Action-Value Function

Estimate Q^π for the current behavior policy π.



After every transition from a nonterminal state s_t, do this:

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

If s_{t+1} is terminal, then $Q(s_{t+1}, a_{t+1}) = 0$.

Sarsa: On-Policy TD Control

Turn this into a control method by always updating 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

Q-Learning: Off-Policy TD Control

One-step Q-learning:

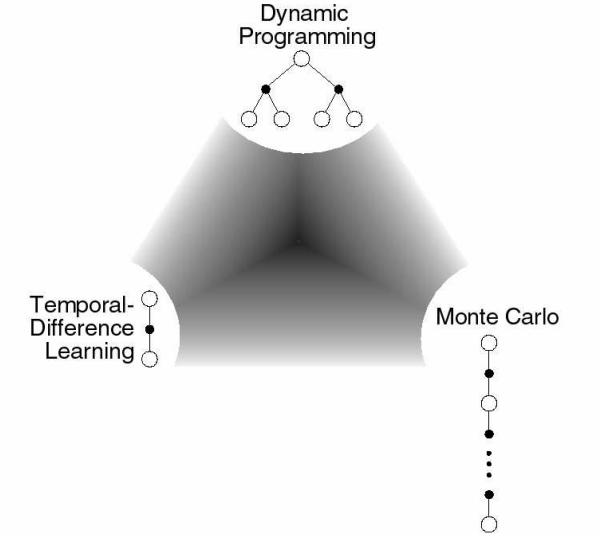
$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)\right]$$

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

Summary

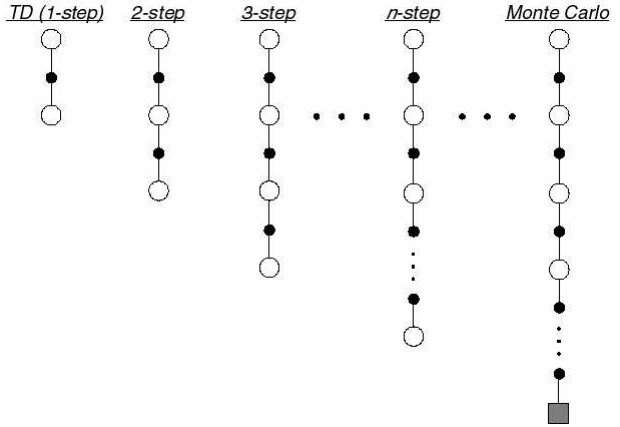
- TD prediction
- Introduced *one-step tabular model-free TD methods*
- Extend prediction to control by employing some form of GPI
  - On-policy control: Sarsa
  - Off-policy control: Q-learning (and also R-learning)
- These methods bootstrap and sample, combining aspects of DP and MC methods

Eligibility Traces



N-step TD Prediction

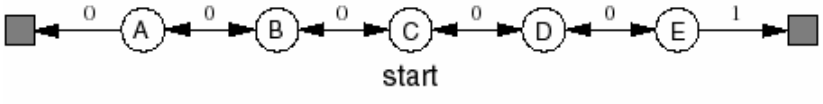
- Idea: Look farther into the future when you do TD backup (1, 2, 3, …, n steps)



Mathematics of N-step TD Prediction

- Monte Carlo: $R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots + \gamma^{T-t-1} r_T$
- TD: $R_t^{(1)} = r_{t+1} + \gamma V_t(s_{t+1})$
  - Use V to estimate remaining return
- n-step TD:
  - 2 step return: $R_t^{(2)} = r_{t+1} + \gamma r_{t+2} + \gamma^2 V_t(s_{t+2})$
  - n-step return: $R_t^{(n)} = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots + \gamma^{n-1} r_{t+n} + \gamma^n V_t(s_{t+n})$
- Backup (online or offline): $\Delta V_t(s_t) = \alpha \left[R_t^{(n)} - V_t(s_t) \right]$

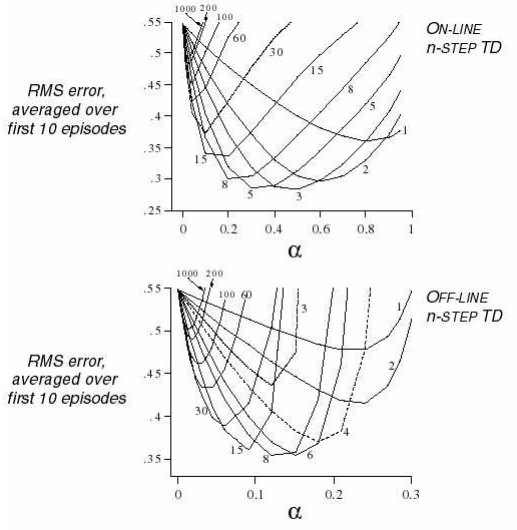
Random Walk Examples



- How does 2-step TD work here?
- How about 3-step TD?

A Larger Example

- Task: 19 state random walk
- Do you think there is an optimal n (for everything)?

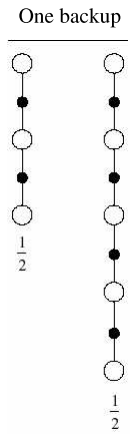


Averaging N-step Returns

- n-step methods were introduced to help with TD(λ) understanding
- Idea: backup an average of several returns
  - e.g. backup half of 2-step and half of 4-step

$$R_t^{avg} = \frac{1}{2} R_t^{(2)} + \frac{1}{2} R_t^{(4)}$$

- Called a complex backup
  - Draw each component
  - Label with the weights for that component



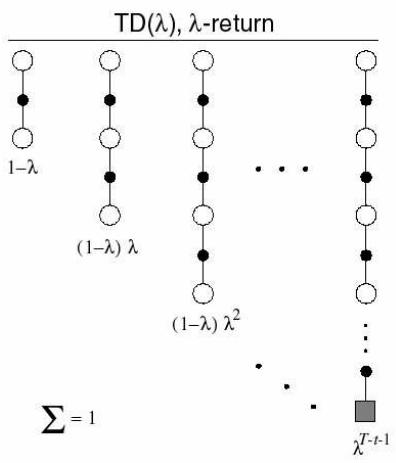
Forward View of TD(λ)

- TD(λ) is a method for averaging all n-step backups
  - weight by λ^{n-1} (time since visitation)
  - λ-return:

$$R_t^{\lambda} = (1-\lambda)\sum_{n=1}^{\infty} \lambda^{n-1} R_t^{(n)}$$

- Backup using λ-return:

$$\Delta V_t(s_t) = \alpha\left[R_t^{\lambda} - V_t(s_t)\right]$$



Relation to TD(0) and MC

- λ-return can be rewritten as:

$$R_t^{\lambda} = \underbrace{(1-\lambda)\sum_{n=1}^{T-t-1} \lambda^{n-1} R_t^{(n)}}_{\text{Until termination}} + \underbrace{\lambda^{T-t-1} R_t}_{\text{After termination}}$$

- If λ = 1, you get MC:

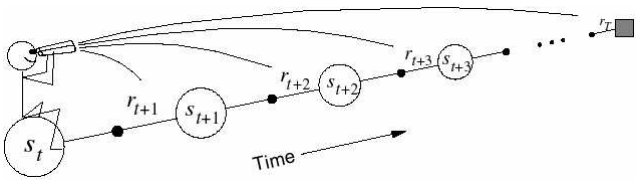
$$R_t^{\lambda} = (1-1)\sum_{n=1}^{T-t-1} 1^{n-1} R_t^{(n)} + 1^{T-t-1} R_t = R_t$$

- If λ = 0, you get TD(0)

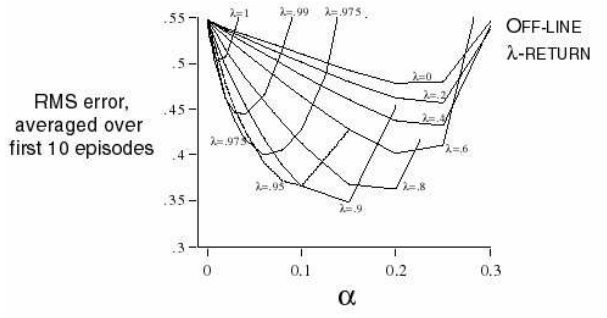
$$R_t^{\lambda} = (1-0)\sum_{n=1}^{T-t-1} 0^{n-1} R_t^{(n)} + 0^{T-t-1} R_t = R_t^{(1)}$$

Forward View of TD(λ) II

- Look forward from each state to determine update from future states and rewards:



λ-return on the Random Walk



- Same 19 state random walk as before
- Why do you think intermediate values of λ are best?

On-line Tabular TD(λ)

Initialize $V(s)$ arbitrarily and $e(s) = 0$, for all $s \in S$
Repeat (for each episode):
  Initialize s
  Repeat (for each step of episode):
    $a \leftarrow$ action given by π for s
    Take action a, observe reward, r, and next state s'
    $\delta \leftarrow r + \gamma V(s') - V(s)$
    $e(s) \leftarrow e(s) + 1$
    For all s:
      $V(s) \leftarrow V(s) + \alpha\delta e(s)$
      $e(s) \leftarrow \gamma\lambda e(s)$
    $s \leftarrow s'$
  Until s is terminal

Sarsa(λ) Algorithm

Initialize $Q(s,a)$ arbitrarily and $e(s,a) = 0$, for all s, a
Repeat (for each episode):
  Initialize s, a
  Repeat (for each step of episode):
    Take action a, observe r, s'
    Choose a' from s' using policy derived from Q (e.g. ?-greedy)
    $\delta \leftarrow r + \gamma Q(s', a') - Q(s, a)$
    $e(s,a) \leftarrow e(s,a) + 1$
    For all s,a:
      $Q(s,a) \leftarrow Q(s,a) + \alpha\delta e(s,a)$
      $e(s,a) \leftarrow \gamma\lambda e(s,a)$
    $s \leftarrow s'; a \leftarrow a'$
  Until s is terminal

Summary

- Provides efficient, incremental way to combine MC and TD
  - Includes advantages of MC (can deal with lack of Markov property)
  - Includes advantages of TD (using TD error, bootstrapping)
- Can significantly speed-up learning
- Does have a cost in computation

Conclusions

- Provides efficient, incremental way to combine MC and TD
  - Includes advantages of MC (can deal with lack of Markov property)
  - Includes advantages of TD (using TD error, bootstrapping)
- Can significantly speed-up learning
- Does have a cost in computation

Three Common Ideas

- Estimation of value functions
- Backing up values along real or simulated trajectories
- Generalized Policy Iteration: maintain an approximate optimal value function and approximate optimal policy, use each to improve the other

Backup Dimensions