An Introduction to Reinforcement Learning and Multi-arm Bandits

Explore-Exploit Dilemma

B. Ravindran

Reconfigurable and Intelligent Systems (RISE) Group

Department of Computer Science and Engineering

Indian Institute of Technology Madras

Learning to Control

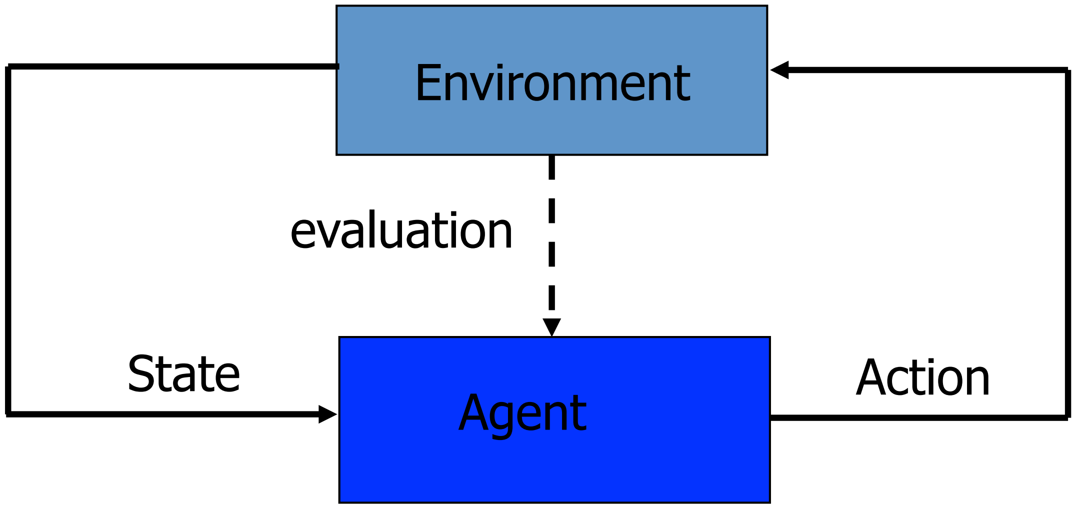
- Familiar models of machine learning
  - Supervised: Classification, Regression, etc.
  - Unsupervised: Clustering, Frequent patterns, etc.

- How did you learn to cycle?
  - Neither of the above
  - Trial and error!
  - Falling down hurts!

Reinforcement Learning

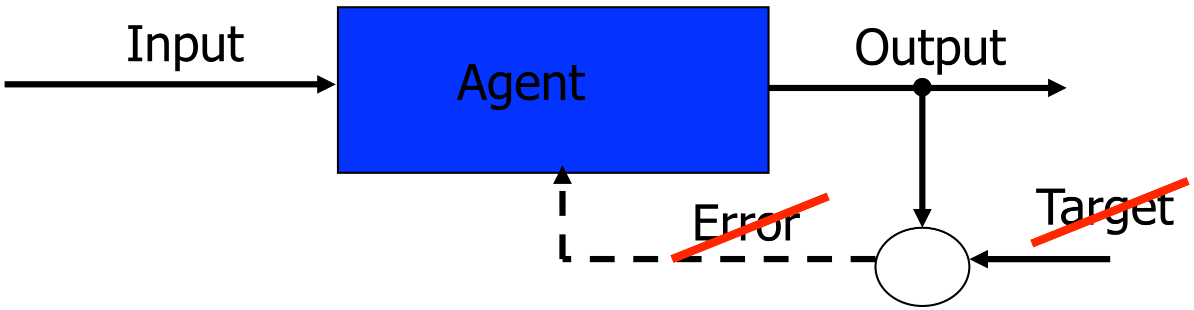
- A trial-and-error learning paradigm
  - Rewards and Punishments
- Not just an algorithm but a new paradigm in itself
- Learn about a system through interaction
- Inspired by behavioural psychology!

RL Framework



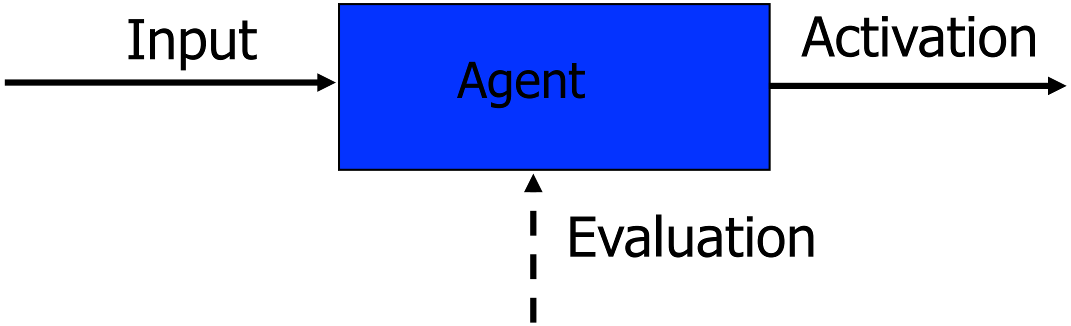
- Learn from close interaction
- Stochastic environment
- Noisy delayed scalar evaluation
- Maximize a measure of long term performance

Not Supervised Learning!



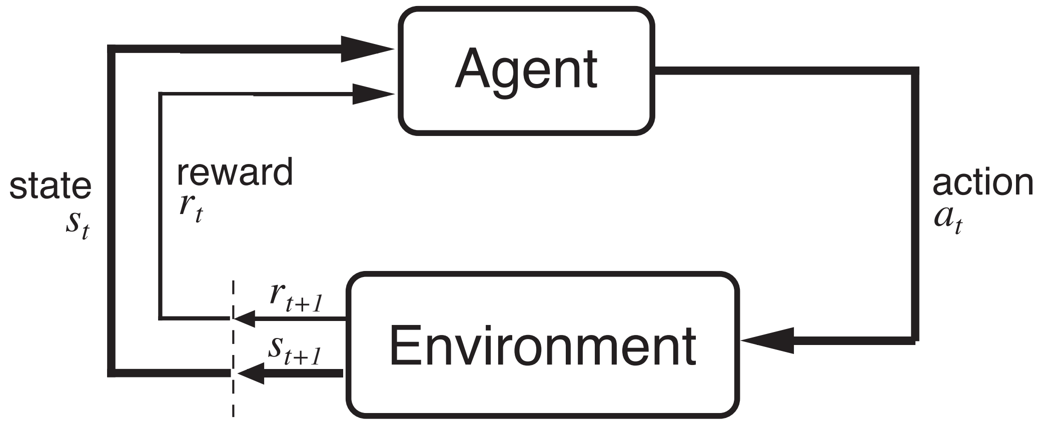
- Very sparse “supervision”
- No target output provided
- No error gradient information available
- Action chooses next state
- Explore to estimate gradient – Trail and error learning

Not Unsupervised Learning



- Sparse "supervision" available
- Pattern detection not primary goal

The Agent-Environment Interface



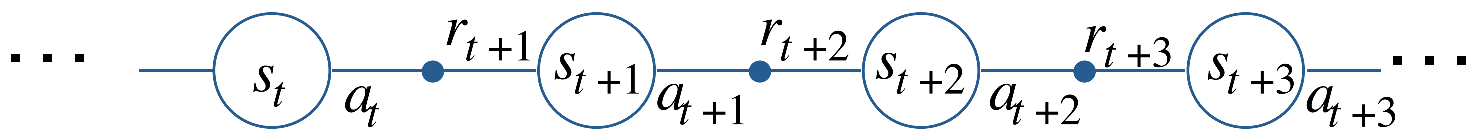
Agent and environment interact at discrete time steps: $t = 0, 1, 2, \ldots$

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.

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 :

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

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.

Returns for Continuing Tasks

Continuing tasks: interaction does not have natural episodes.

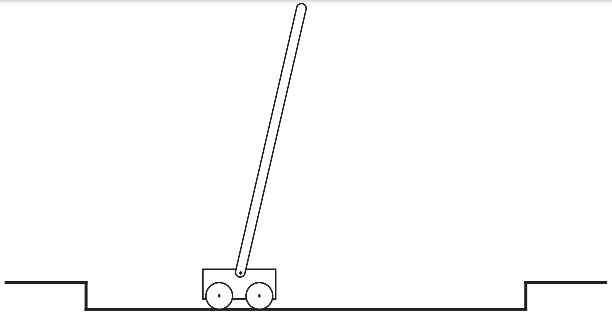
Discounted return:

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

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

$$\text{shortsighted } \; 0 \leftarrow \gamma \rightarrow 1 \; \text{ farsighted}$$

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:

$$\text{reward} = +1 \text{ for each step before failure}$$

$$\Rightarrow \quad \text{return} = \text{number of steps before failure}$$

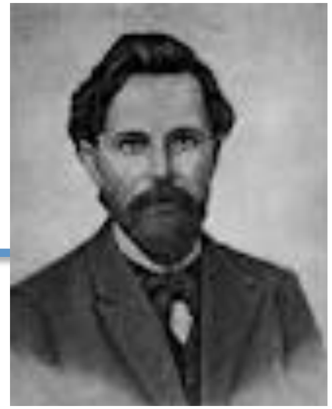
As a **continuing task** with discounted return:

$$\text{reward} = -1 \text{ upon failure; } 0 \text{ otherwise}$$

$$\Rightarrow \quad \text{return} = -\gamma^k, \text{ for } k \text{ steps before failure}$$

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

The Markov Property



- "the state" at step *t*, means 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**:

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

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

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: $M = \langle S, A, P, R \rangle$
  - state and action sets
  - one-step "dynamics" defined by transition probabilities:

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

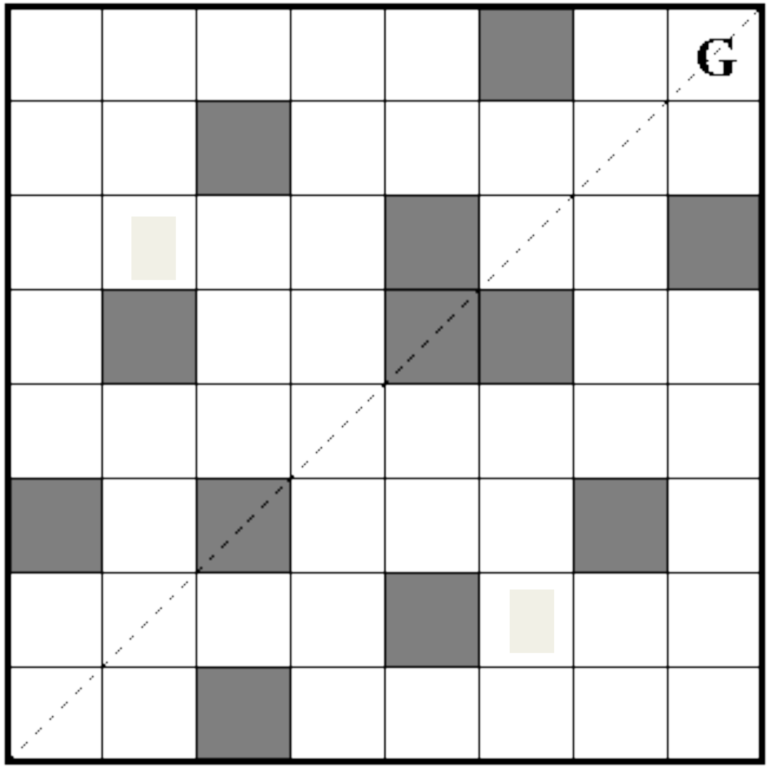
  - reward expectations:

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

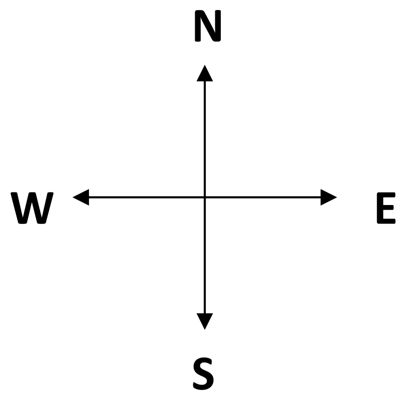
Markov Decision Processes

- MDP, *M*, is the tuple: $M = \langle S, A, \Psi, P, R \rangle$
  - *S* : set of states.
  - *A* : set of actions.
  - $\Psi \subseteq S \times A$: set of admissible state-action pairs.
  - $P : \Psi \times S \rightarrow [0,1]$: probability of transition.
  - $R : \Psi \rightarrow \Re$: expected reward.
- Policy $\pi \; : \; S \rightarrow A$ (can be stochastic)
- Maximize total expected reward.

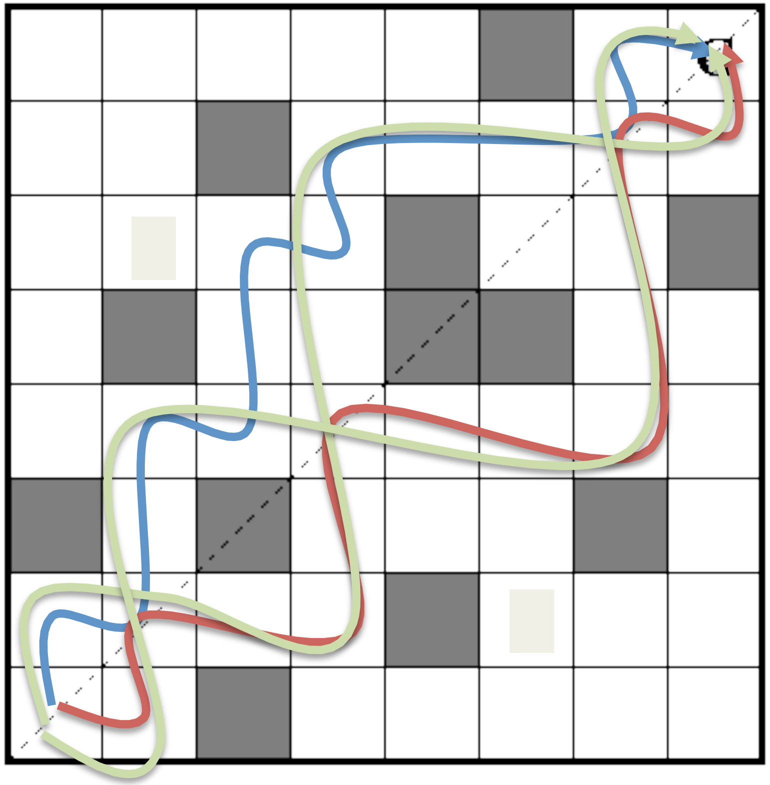
Example



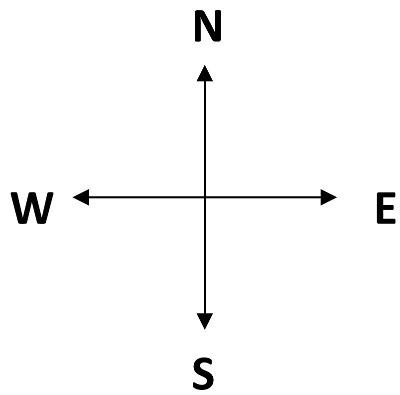
$$M = \langle S, A, \Psi, P, R \rangle$$



Optimal Policies



$$M = \langle S, A, \Psi, P, R \rangle$$



Solution Methods

- Temporal Difference Methods
  - TD(λ)
  - Q-learning
  - SARSA
  - Actor-Critic
- Policy Search
  - Policy Gradient Methods
  - Evolutionary algorithms
- Stochastic Dynamic Programming

Applications of RL

- Optimal Control
  - Robot Navigation
  - Helicopters!
  - Chemical Plants
- Combinatorial Optimization
  - Elevator Dispatching
  - VLSI placement and routing
  - Job-shop scheduling
  - Routing algorithms
  - Call admission control
- More
  - Intelligent Tutoring Systems
- Computational Neuroscience
  - Primary mechanism of learning
- Psychology
  - Behavioral and operant conditioning
  - Decision making
- Operations Research
  - Approximate Dynamic Programming
- More
  - Game Playing
  - Dialogue systems

Reinforcement Learning
Lecture 8

Gillian Hayes

1st February 2007

Algorithms for Solving RL: Monte Carlo Methods

- What are they?
- Monte Carlo Policy Evaluation
- First-visit policy evaluation
- Estimating Q-values
- On-policy methods
- Off-policy methods

Monte Carlo Methods

- **Learn** value functions
- **Discover** optimal policies
- Don't require environmental knowledge: $P^a_{ss'}$, $R^a_{ss'}$,
  cf. Dynamic Programming
- Experience : sample sequences of states, actions, rewards s, a, r
  : real experience, simulated experience
- Attains optimal behaviour

How Does Monte Carlo Do This?

- Divide experience into episodes
  - all episodes must terminate

    e.g. noughts-and-crosses, card games

- Keep estimates of value functions, policies

- Change estimates/policies at end of each episode

$\Rightarrow$ Keep track of $s_1, a_1, r_1, s_2, a_2, r_2, \ldots s_{T-1}, a_{T-1}, r_{T-1}, s_T$

$s_T =$ terminating state

- Incremental episode-by-episode

  NOT step-by-step cf. DP

- Average **complete** returns – NOT partial returns

Returns

- Return at time t: $R_t = r_{t+1} + r_{t+2} + \ldots r_{T-1} + r_T$ for each episode
  r_T is a terminating state
- Average the returns over many episodes starting from some state s.

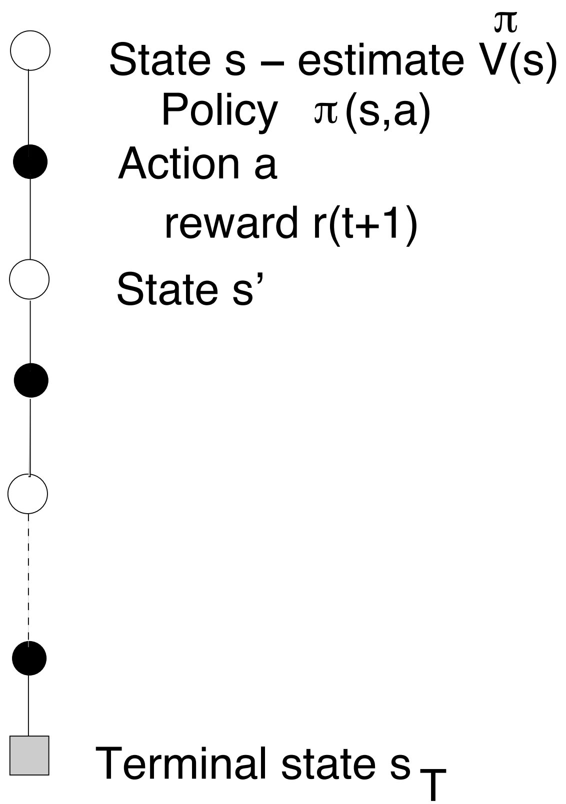
This gives the value function $V^\pi(s)$ for that state for policy π since the state value $V^\pi(s)$ is the expected cumulative future discounted reward starting in s and following policy π.

Monte Carlo Learning of V^π

MC methods estimate from experience: generate many "plays" from s, observe total reward on each play, average over many plays

1. Initialise
   - $\pi =$ arbitrary policy to be evaluated
   - $V =$ arbitrary value function
   - $Returns(s)$ an empty list, one for each state s
2. Repeat till values converge
   - Generate an episode using π
   - For each state appearing in the episode
     - $R =$ return following first occurrence of s
     - Append R to $Returns(s)$
     - $V(s) =$ average $Returns(s)$

Backup Diagram for MC



One Episode – full episode needed before back-up.
cf DP which backs up after one move
Monte Carlo does **not** bootstrap but
Monte Carlo does sample

- Play many games
- Average returns (first-visit MC) following each state
- $\Rightarrow$ True state-value functions
    * Easier than DP $\Rightarrow$ That needs $P^a_{ss'}, R^a_{ss'}$
    * Easier to generate episodes than calculate probabilities

Policy Iteration (Reminder)

– Policy evaluation: Estimate V^π or Q^π for fixed policy π

– Policy improvement: Get a policy better than π

Iterate until optimal policy/value function is reached

So we can do Monte Carlo as the Policy Evaluation step of Policy Iteration because it computes the value function for a given policy. (There are other algorithms we can use.)

First-visit MC vs. Every-visit MC

In each episode observe return following **first** visit to state s

Number of first visits to s must $\rightarrow \infty$

Converges to $V^{\pi}(s)$

cf. Every-visit MC

Calculate V as the average over return following **every** visit to state s in a set of episodes

Good Properties of MC

Estimates of V for each state are independent
– no bootstrapping

Compute time to calculate changes (i.e. V of each state) is independent of number of states

If values of only a few states needed, generate episodes from these states $\Rightarrow$ can ignore other states

Can learn from actual/simulated experience

Don't need $P^a_{ss'}$, $R^a_{ss'}$,

Estimating Q-Values

$Q^\pi(s, a)$ – similarly to V

Update by averaging returns following first visit to that state-action pair

Problem

If π deterministic, some/many (s, a) never visited

MUST EXPLORE!

So...

* Exploring starts: start every episode at a different (s, a) pair

* Or always use ϵ-greedy or ϵ-soft policies
  – stochastic, where $\pi(s, a) > 0$

Optimal Policies – Control Problem

Policy Iteration on Q

$$\pi_0 \rightarrow_{PE} Q^{\pi^0} \rightarrow_{PI} \pi_1 \rightarrow_{PE} Q^{\pi^1} \rightarrow_{PI} \pi_2 \ldots \rightarrow_{PI} \pi^* \rightarrow_{PE} Q^*$$

- Policy Improvement: Make π greedy w.r.t. current Q
- Policy Evaluation: As before, with ∞ episodes

Or episode-by-episode iteration. After an episode:

- policy evaluation (back-up)
- improve policy at states in episode
- eventually converges to optimal values and policy

Can use exploring starts: MCES – Monte Carlo Exploring Starts to ensure coverage of state/action space

Algorithm: see e.g. S+B Fig. 5.4

Monte Carlo: Estimating $Q^\pi(s, a)$

- If π deterministic, some (s, a) not visited $\Rightarrow$ can't improve their Q estimates

  MUST MAINTAIN EXPLORATION!

- Use exploring starts $\longrightarrow$ optimal policy

- Use an ϵ-soft policy
  ON-POLICY CONTROL $\longrightarrow$ ϵ-greedy policy
  OFF-POLICY CONTROL $\longrightarrow$ optimal policy

On-Policy Control

Evaluate and improve the policy used to generate behaviour

Use a soft policy:

$\pi(s,a) > 0 \;\; \forall s, \forall a$ GENERAL SOFT POLICY DEFINITION

$\pi(s,a) = \frac{\epsilon}{|A|}$ if a not greedy ϵ-GREEDY

$\quad\quad = 1 - \epsilon + \frac{\epsilon}{|A|}$ if a greedy

$\pi(s,a) \geq \frac{\epsilon}{|A|} \;\; \forall s, \forall a$ ϵ-SOFT

POLICY ITERATION

$Evaluation$: as before $Improvement$: move towards ϵ-greedy policy (not greedy)

Avoids need for exploring starts

ϵ-greedy is "closer" to greedy than other ϵ-soft policies

Off-Policy Control

- Behaviour policy π' generates moves
- But in off-policy control we learn an Estimation policy π. How?

We need to:

- compute the weighted average of returns from behaviour policy
- the weighting factors are the probability of them being in estimation policy,
- i.e. weight each return by relative probability of being generated by π and π'

In detail...

Reinforcement Learning
Lecture 10

Gillian Hayes

8th February 2007

School of informatics

Algorithms for Solving RL: Temporal Difference Learning (TD)

- Incremental Monte Carlo Algorithm
- TD Prediction
- TD vs MC vs DP
- TD for control: SARSA and Q-learning

Incremental Monte Carlo Algorithm

Our first-visit MC algorithm had the steps:

R is the return following our first visit to s
Append R to $Returns(s)$
$V(s) = \text{average}(Returns(s))$

We can implement this incrementally:

$$V(s) = V(s) + \frac{1}{n(s)}[R - V(s)]$$

where $n(s)$ is the number of first visits to s

We can also formulate a constant-α Monte Carlo update:

$$V(s) = V(s) + \alpha[R - V(s)]$$

useful when tracking a non-stationary problem (why?).

Model-Based vs Model-Free Learning

- In RL we're generally trying to learn an optimal policy

- If a model is available, $P^a_{ss'}$, $R^a_{ss'}$, we can calculate optimal policy via dynamic programming

- If no model, either:

  learn model and then derive optimal policy
  (model-based methods) or

  learn optimal policy without learning model
  (model-free methods)

- Temporal difference (TD) learning is a model-free, bootstrapping method based on sampling the state-action space

Temporal Difference Prediction

Policy Evaluation is often referred to as the Prediction Problem: we are trying to predict how much return we'll get from being in state s and following policy π by learning the state-value function V^π.

Monte-Carlo update:

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

Target: actual return from s_t to end of episode

Simplest temporal difference update TD(0):

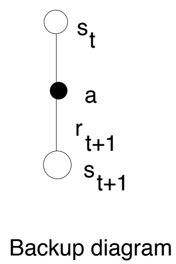
$$V(s_t) \rightarrow V(s_t) + \alpha[\underbrace{r_{t+1} + \gamma V(s_{t+1})} - V(s_t)]$$

Target: estimate of the return

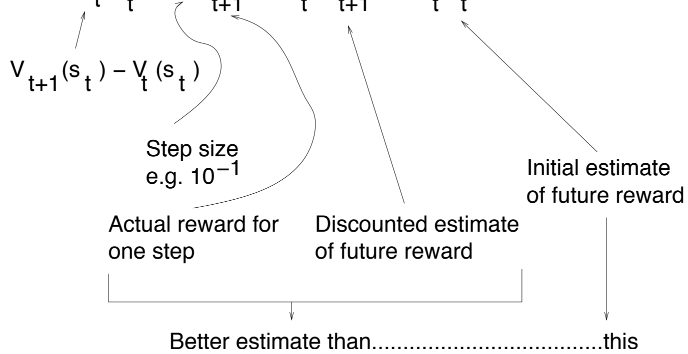
Both have the same form

Temporal Difference Learning

- Doesn't need a model $P^a_{ss'}$, $R^a_{ss'}$
- Learns directly from experience
- Updates estimates of $V(s)$ based on what happens after visiting state s



$$\Delta V_t(s_t) = \alpha [r_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t)]$$



TD(0) update:

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

cf Dynamic Programming update:

$$\begin{aligned} V^\pi(s) &= E_\pi\{r_{t+1} + \gamma V^\pi(s_{t+1}) \mid s_t = s\} \\ &= \sum_a \pi(s, a,) \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V^\pi(s')] \end{aligned}$$

Advantages of TD Learning Methods

- Don't need a model of the environment
- On-line and incremental so can be fast

  don't need to wait till the end of the episode so need less memory, computation
- Updates are based on actual experience (r_{t+1})
- Converges to $V^{\pi}(s)$ – but must decrease step size α as learning continues
- Compare backup diagrams of TD, MC and DP

Bootstrapping, Sampling

TD **bootstraps**: it updates its estimates of V based on other estimates of V

DP also bootstraps

MC does not bootstrap: estimates of complete returns are made at the end of the episode

TD **samples**: its updates are based on one path through the state space

MC also samples

DP does not sample: its updates are based on all actions and all states that can be reached from the updating state

Examples: see e.g. random walk example S+B sect. 6.2

MC vs TD updating: see e.g. S+B sect. 6.3

Difference Between TD and MC Estimates

See S+B Example 6.4:

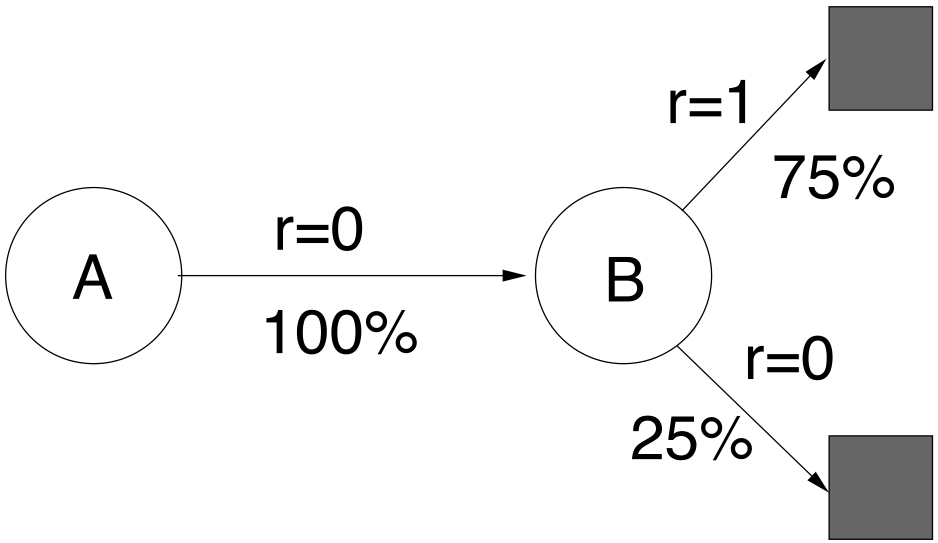
Suppose you observe the following 8 episodes:

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

First episode starts in state A, transitions to B getting a reward of 0, and terminates with a reward of 0. Second episode starts in state B and terminates with a reward of 1, etc.

What are the best values for the estimates V(A) and V(B)?

Modelling the Underlying Markov Process



V(A) = ?

TD and MC Estimates

- Batch Monte Carlo (updating after all these episodes are done) gets V(A) = 0.
  - This best matches the training data
  - It minimises the mean-square error on the training set

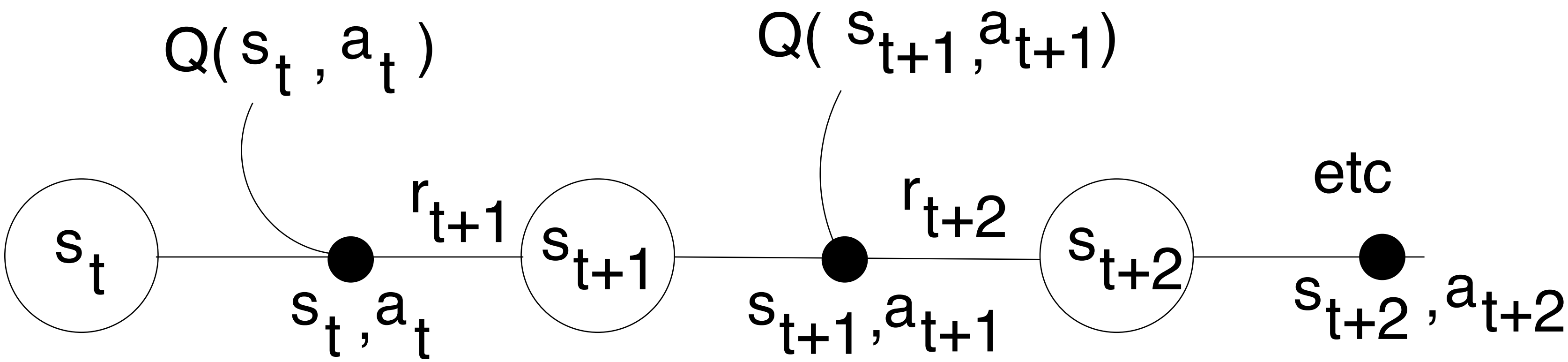
- Consider sequentiality, i.e. A goes to B goes to terminating state; then V(A) = 0.75.
  - This is what TD(0) gets
  - Expect that this will produce better estimate of future data even though MC gives the best estimate on the present data

- Is correct for the maximum-likelihood estimate of the model of the Markov process that generates the data, i.e. the best-fit Markov model based on the observed transitions
- Assume this model is correct; estimate the value function – "certainty-equivalence estimate"

TD(0) tends to converge faster because it's moving towards a "better" estimate.

TD for Control: Learning Q-Values

Learn action values $Q^\pi(s,a)$ for the policy π



SARSA update rule:

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

• Choose a behaviour policy π and estimate the Q-values (Q^π) using the SARSA update rule. Change π towards greediness wrt Q^π.

• Use ϵ-greedy or ϵ-soft policies.

• Converges with probability 1 to optimal policy and Q-values if visit all state-action pairs infinitely many times and policy converges to greedy policy, e.g. by arranging for ϵ to tend towards 0.

Remember: learning optimal Q-values is useful since it tells us immediately which is(are) the optimal action(s) – have the highest Q-value

SARSA Algorithm

- Initialise $Q(s, a)$
- Repeat `many times`
  - Pick s, a
  - Repeat `each step to goal`
    - Do a, observe r, s'
    - Choose a' based on $Q(s', a')$ ϵ`-greedy`
    - $Q(s, a) = Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)]$
    - $s = s'$, $a = a'$
  - Until s terminal (where $Q(s', a') = 0$)

Use with policy iteration, i.e. change policy each time to be greedy wrt current estimate of Q

Example: windy gridworld, S+B sect. 6.4

Q-Learning

SARSA is an example of **on-policy** learning. Why?

Q-LEARNING is an example of **off-policy** learning

Update rule:

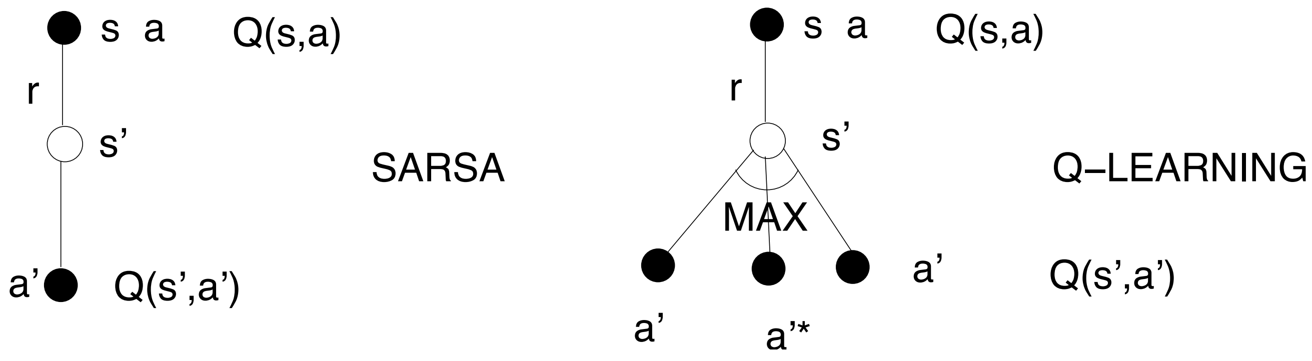
$$\Delta Q_t(s_t, a_t) \quad = \quad \alpha[r_{t+1} + \gamma \max_a Q_t(s_{t+1}, a) - Q_t(s_t, a_t)]$$

Always update using *maximum* Q value available from next state: then $Q \Rightarrow Q*$, optimal action-value function

Q-Learning Algorithm

- Initialise $Q(s, a)$
- Repeat `many times`
  - Pick s `start state`
  - Repeat `each step to goal`
    * Choose a based on $Q(s, a)$ ϵ`-greedy`
    * Do a, observe r, s'
    * $Q(s, a) = Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$
    * $s = s'$
  - Until s terminal

Backup Diagrams for SARSA and Q-Learning



SARSA backs up using the action a' actually chosen by the behaviour policy.

Q-LEARNING backs up using the Q-value of the action a'^* that is the *best* next action, i.e. the one with the highest Q value, $Q(s', a'^*)$. The action actually chosen by the behaviour policy *and followed* is not necessarily a'^*

Example: The cliff S+B sect. 6.5

Q-Learning vs SARSA

QL: $Q(s,a) = Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$ off-policy

SARSA: $Q(s,a) = Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)]$ on-policy

In the cliff-walking task:

QL: learns optimal policy along edge

SARSA: learns a safe non-optimal policy away from edge

ϵ-greedy algorithm

For $\epsilon \neq 0$ **SARSA** performs better online. Why?

For $\epsilon \to 0$ gradually, both converge to optimal.