Course # 01 - Fundamentals of RL

RL Agent → learn by exploration

RL → problem of learning is modelled as exploration and learning

→ Can use supervised and unsupervised learning within RL

Todo → Read Ch # 2-2.7 ✓

k-armed Bandit Problem

→ We have an agent who chooses between k actions and receives a reward based on the action it chooses.

k → actions

- value of taking each action is called action-values.

→ In the action-value function, the value is the expected reward and is defined as:

$$q_*(a) \doteq \mathbb{E}[R_t \mid A_t = a] \quad \forall a \in \{1, \dots, k\}$$

⇒ value of selecting an action as the expected reward we receive when taking bad action.

$$q_*(a) = \sum_r p(r|a) r$$

↳ probability of observing the reward r

→ can be extended to continuous case by integrating instead of summation.

→ The goal is to maximize the expected reward.

$$\Rightarrow \boxed{\underset{a}{\text{argmax}}\ q_*(a)}$$

Learning Action Values

- Estimate action values using the sample-average method

The value of an action is the expected reward when that action is taken.

$$q_*(a) = \mathbb{E}[R_t \mid A_t = a]$$

⇒ $q_*(a)$ is not known to the agent, so we estimate it.

Sample-Average Method

$$Q_t(a) \doteq \frac{\text{sum of rewards when } a \text{ (action } a\text{) taken prior to } t}{\text{number of times } a \text{ taken prior to } t}$$

$$Q_t(a) = \frac{\sum_{i=1}^{t-1} R_i}{t-1}$$

Action Selection

Greedy action selection → select action which has currently largest estimated value.

⇒ Selecting a greedy action means, the agent is exploiting the current knowledge.

$a_g = \text{argmax}(Q(a))$ → greedy action selection

⇒ Alternatively, the agent might choose to explore by sacrificing the immediate reward (greedy approach) in order to gain more information on other actions.

- The agent cannot choose to do both explore and exploit at the same time, this is one of the fundamental problems of RL.
  - ↳ The exploration - exploitation dilemma

Estimating Action Values Incrementally

The SAM can be written in recursive manner.

$$Q_{n+1} = \frac{1}{n}\sum_{i=1}^{n} R_i \rightarrow \text{Sample Average Method (SAM)}$$

$$= \frac{1}{n}\left(R_n + \sum_{i=1}^{n-1} R_i\right) \rightarrow \text{sum till prev. value}$$

$$Q_{n+1} = \frac{1}{n}\left(R_n + (n-1)\frac{1}{n-1}\sum_{i=1}^{n-1} R_i\right)$$

As,

$$Q_n = \frac{1}{n-1}\sum_{i=1}^{n-1} R_i$$

$$Q_{n+1} = \frac{1}{n}(R_n + (n-1)Q_n)$$

$$= \frac{1}{n}(R_n + nQ_n - Q_n) \rightarrow \text{incremental rule update}$$

$$Q_{n+1} = Q_n + \frac{1}{n}(R_n - Q_n)$$

New Estimate ← Old Estimate + StepSize [Target − Old Estimate]

↳ $\frac{1}{n}$ $(R_n - Q_n)$ → error

$\Rightarrow Q_{n+1} = Q_n + \alpha_n(R_n - Q_n)$

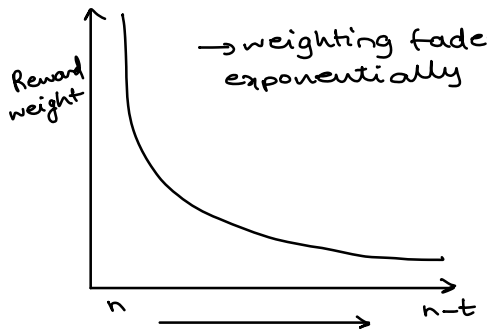
$\alpha_n \rightarrow [0,1]$

$\alpha_n = \frac{1}{n}$ → sample average case

Non-stationary bandit problem

→ like the normal bandit problem, but the distribution of reward changes with time.

- One solution is to have a constant step size so that the most recent reward are given higher weights.



→ weighting fades exponentially

Decaying Past Rewards

$$Q_{n+1} = Q_n + \alpha(R_n - Q_n)$$
$$= Q_n + \alpha R_n - \alpha Q_n$$
$$= \alpha R_n + (1-\alpha)Q_n$$
$$Q_{n+1} = \alpha R_n + (1-\alpha)[\alpha R_{n-1} + (1-\alpha)Q_{n-1}]$$
$$= \alpha R_n + (1-\alpha)\alpha R_{n-1} + (1-\alpha)^2 Q_{n-1}$$

⇒ shows relationship b/w R_n and R_{n-1}

⇒ can unroll this further to get to Q_1.

$$= \alpha R_n + (1-\alpha)\alpha R_{n-1} + (1-\alpha)^2 \alpha R_{n-2} + \cdots + (1-\alpha)^{n-1}\alpha R_1 + (1-\alpha)^n Q_1$$

$$= \underbrace{(1-\alpha)^n Q_1}_{\text{initial value of } Q} + \underbrace{\sum_{i=1}^{n} \alpha(1-\alpha)^{n-i} R_i}_{\text{weighted sum of rewards over time}}$$

Exploration and Exploitation

→ exploits the agent's current knowledge

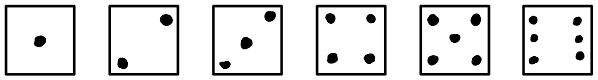
- improve the knowledge for long-term benefit
- exploits the knowledge for short term benefit

→ This forms a dilemma, not knowing what to choose. Cannot choose both.

↳ One very simple solution is to choose randomly.

Epsilon-Greedy Action Selection

↳ Solves the problem by choosing exploitation most of the time with a little chance of exploring.



Epsilon → the prob. of choosing

- ✓ $\varepsilon = 0$ → only greedy w/o exploration
- ✓ $\varepsilon = 0.01$ → explore 1% of the time.

⇒ Roll the dice, if it falls on one → explore
↳ greedy action otherwise

→ In this case, epsilon = 1/6

$$A_t \leftarrow \begin{cases} \underset{a}{\text{argmax}}\ Q_t(a) & \text{greedy action with prob. } 1-\varepsilon \text{ (exploit)} \\ a \sim \text{Uniform}(\{a_1 \ldots a_K\}) & \text{with prob } \varepsilon \text{ (explore)} \end{cases}$$

↳ random action

Optimistic Initial Values → better balances b/w exploration & exploitation.

Doctor - Medicine Example

- A reward of 1 if the treatment succeeds otherwise 0.

Update rule ⇒ $Q_{n+1} \leftarrow Q_n + \alpha(R_n - Q_n)$

Let $\alpha = 0.5$

$Q_1(P) = 2.0$, $Q_1(Y) = 2.0$

$Q_1(B) = 2.0$ → optimistic initial values

	P	Y	B
Patient 1	+1 $Q_2 = 1.5$		
Patient 2		+0 $Q_3(Y) = 1$	
Patient 3			+1 $Q_4(B) = 1.5$
Patient 4			+1 $Q_5(B) = 1.75$
Patient 5	+0 $Q_6(P) = .75$		
Patient 6			+0 $Q_7(B) = 0.625$
Patient 7		+1 $Q_8(Y) = 1.0$	
Patient 8		+1 $Q_9(Y) = 1.0$	
Patient 9		+0 $Q_{10}(Y) = 0.5$	
Patient 10	+0 $Q_{11}(P) = .375$		

← medicines to test

$q^*(P) = 0.25$

$q^*(Y) = 0.75$

$q^*(B) = 0.5$

⇒ $Q(P) > q^*(P)$ → explore

After the first patient comes in, the doctor prescribes the medicine P, then the Q value updates as follows.

The patient feels better so $R_n = 1$

$Q_n = 2.0$

$Q_{n+1} = 2 + 0.5(1-2)$

$= 2 + 0.5(-1)$

$Q_{n+1} = 2 - 0.5 = 1.5$

So, $Q_2 = 1.5$

So, $Q_{11}(P) = 0.375$, $Q_{11}(Y) = 0.5$, $Q_{11}(B) = 0.625$

we can say that optimistic initial values encourage exploitation early in learning.

Optimistic Initial Values have limitations (OIV)

- Exploits only early on
- Non-stationary values → actions change over time
- we may not know what the OIV should be because we may not know max. reward in practice.
- ↳ could be that agent following OIV already settled on an option & now unaware a different action is better.

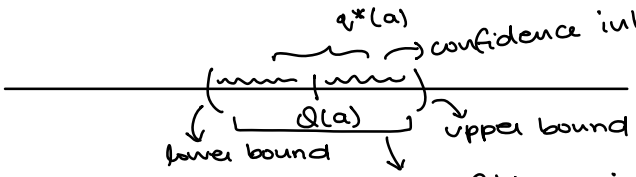
Upper-Confidence Bound (UCB) Action Selection

→ method to balance exploration and exploitation.

- Since we are estimating our action values from sampled rewards, there is inherent uncertainity in accuracy of our estimate.
- we explore to reduce this uncertainity so that we can take better decisions in the future.

UCB → uses the uncertainity in estimates to drive exploration.

Uncertainity in Estimates

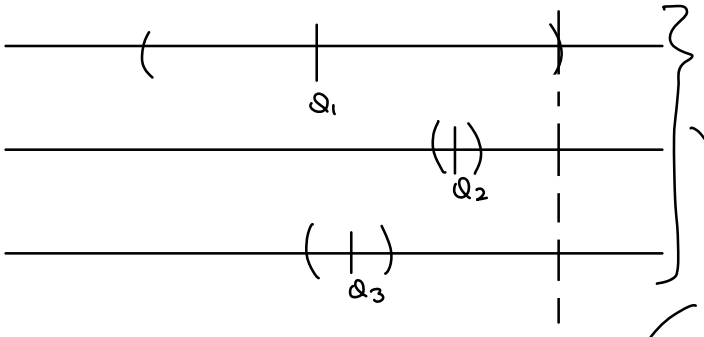


confidence interval representing uncertainity.

- If the interval is small we are certain that $q^*(a)$ is very near our estimate.
- If the interval is large we are uncertain that $q^*(a)$ is near.

UCB → Optimism in Face of Uncertainity

If we are uncertain about something, optimistically assume that it is good.



3 actions with associated uncertainities.

The agent has no idea which to pick, so it optimistically picks the one with highest upper bound.

- This is good because it could either be good if we get good reward or we get to learn about the action and the uncertainity improves (reduces)

$$A_t \doteq \operatorname{argmax} \left[Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}} \right]$$

- argmax → select the one with the highest combination.
- $Q_t(a)$ → estimated value (exploit)
- $c\sqrt{\frac{\ln t}{N_t(a)}}$ → upper conf. bound (explore)

c → user chosen parameter

$\ln t$ → ln timesteps

$N_t(a)$ → times action a is taken

Week # 02

Markov Decision Processes (MDPs)

In k-armed Bandit problem → agent is presented with same situation each time and same action is always optimal.

- In many problems, different situation calls for different responses.
- The actions chosen now affect the reward we get into the future.

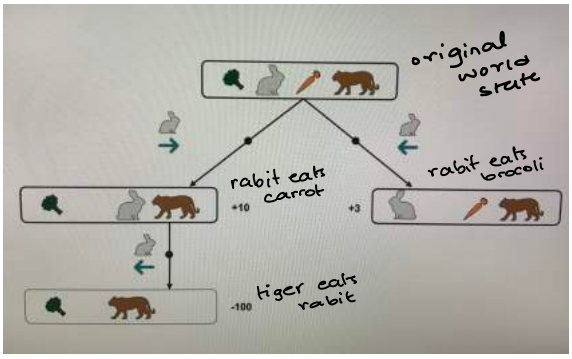
MDP → captures these two aspects of the real world problems.

Difference b/w Bandits & MDPs

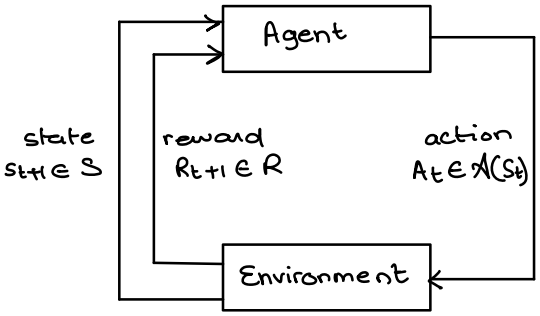
Brocolli, +3 A rabit, carrot +10
↓
prefers carrot
So rabit goes right
In another situation
Carrot, Rabit, Brocolli
Here the rabit goes left

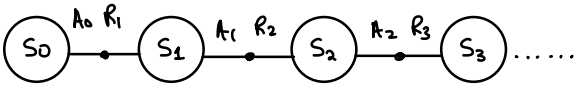
✓ The k-armed bandit doesn't account for the fact that different situations call for different actions.

✓ A k-armed bandit agent is only concerned about the immediate reward, does not consider the long term impact of the actions.

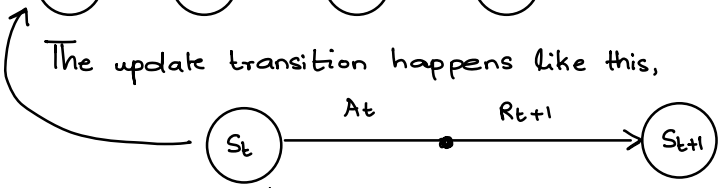


Formalism



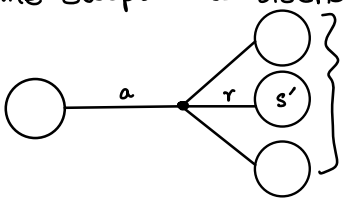


The update transition happens like this,



The dynamics of MDPs

The output is stochastic, so the language is prob.



set of states

$\Rightarrow p(s', r \mid s, a)$ ← the transition dynamic fⁿ.

$p \rightarrow$	the joint prob. of next state s' and reward r
$s' \rightarrow$	the next state
$r \rightarrow$	the reward
$s \rightarrow$	the current state
$a \rightarrow$	the action taken

Assumption: the actions and rewards are finite.

Since p is a prob. distribution, it must be non-negative and sum over all possible states and rewards must be 1.

$$p: S \times R \times S \times A \rightarrow [0,1]$$

$$\sum_{s' \in S} \sum_{r \in R} p(s', r \mid s, a) = 1 \quad \forall s \in S,\ a \in A(s)$$

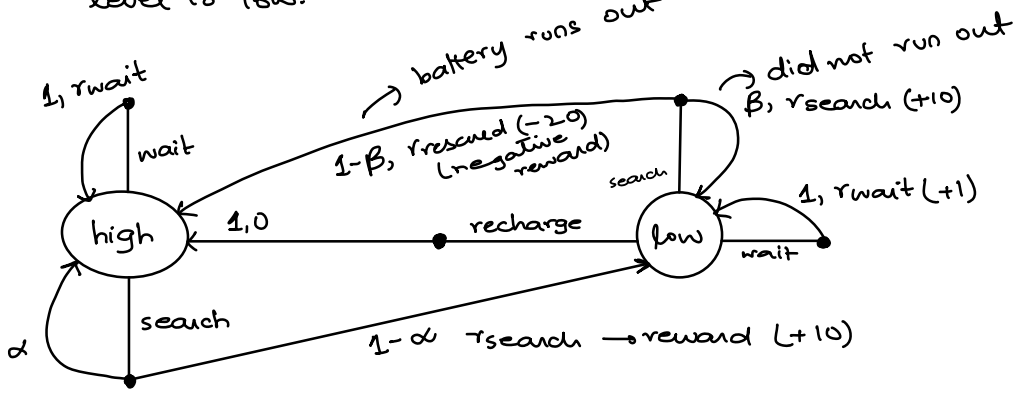
- Future state and reward only depend on current state & action. → Markov Property

Examples of MDPs

1) Recycling Robot → detects the cans and collect them (as many as it can)

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

→ search with high energy may drop level to low. → wait does not drain battery



MDP → pretty general and can range from low level to high level.

2) Pick & Place Task

State: latest readings of joint angles and velocities
actions: the amount of voltage applied to each motor
reward: +100 successful object placed
−1 unit energy consumed

- Agents have long-term goals
  ⇒ what looks good in short term might not be best in long term.

Goal: Formal definition

$G_t \doteq R_{t+1} + R_{t+2} + \dots$

↳ random variable because dynamics of MDP can be stochastic

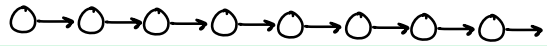
Randomness in individual rewards and state transitions therefore maximize the expected return

$$E[G_t] = E[R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T]$$

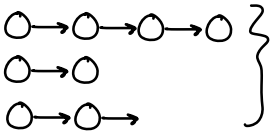
↳ the sum should be finite

↳ final time step

Episodic Tasks



When the interaction ends, it breaks into chunks called episodes



episodes

- Each episode begins independently of how the previous one ended.
- At termination, agent is reset to start state.
- Every episode has a final stage called terminal stage.

↓ Episodic tasks

Example: Episodic Task

A game of chess ends with check mate, draw or resignation

A single episode → a single game

Reward Hypothesis

↳ Give a man a fish, eat for a day → (GOFAI) old AI

Teach a man to fish, eat for life time (supervised learning)

Give a man a taste of fish, he'll figure out the details even if they change (RL)

Intelligent behavior arises from the actions of an individual seeking to maximized its received reward signals in a complex and changing world.

How to maximize rewards?

Goals as Rewards → set reward

↓ one way

1 for goal, 0 otherwise → goal-reward rep.

-1 for not goal, 0 goal reached → action-penalty rep.

↓

⇒ difficult in long term goals

→ easy when common currency

stock market ($)

solar panel position (energy)

Continuing Tasks

Episodic tasks	Continuing tasks
• Interaction breaks naturally into episodes • Each episode ends in a terminal state • Episodes are independent $G_t \doteq R_{t+1} + R_{t+2} + \cdots + R_T$	• Cannot break into episodes the interaction goes on continually • No terminal stage Example → smart thermostat (never stops interacting with the env.) state = temp, situations, time of day actions = on, off → (off: heater) reward = −1 for manual adj., 0 otherwise To avoid −ve reward, thermostat learns to anticipate user pref. $G_t \doteq R_{t+1} + R_{t+2} + \cdots \infty$? → modify To make sure G_t is finite discount the rewards in future by $\gamma \Rightarrow 0 \leq \gamma \leq 1$ (↳ atleast) $G_t \doteq R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{k-1} R_{t+k} + \cdots$ $= \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$ ← guaranteed to be finite as long as $\gamma \in [0,1]$

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

- Assume R_{max} is the reward agent can receive at any time stamp

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \leq \sum_{k=0}^{\infty} \gamma^k R_{max} \quad (\text{const.})$$

(upper bound)

$$= R_{max} \sum_{k=0}^{\infty} \gamma^k$$

→ geometric series

converges to $\frac{1}{1-\gamma}$

$$= R_{max} \times \frac{1}{1-\gamma}$$

→ finite ∴ upper bound

Effect of γ on agent behavior

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{k-1} R_{t+k} + \cdots$$

$\gamma = 0$

$$G_t \doteq R_{t+1} + (0) R_{t+2} + (0) R_{t+3} + \cdots$$

$G_t \doteq R_{t+1}$ → reward at just next step

⇒ Agent is short sighted ∴ only cares about immediate rewards.

$\gamma \to 1$

⇒ Agent takes into account the rewards in future

Recursive Nature of Returns

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{k-1} R_{t+k} + \cdots$$

$$\doteq R_{t+1} + \gamma (R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + \cdots) \longrightarrow G_{t+1}$$

$G_t = R_{t+1} + G_{t+1}$ → recursive equation.

Week # 3

Specifying Policies

Policy → distribution over actions for each state

Deterministic Policy Notation

$\pi(s) = a$ (policy) → In the simplest case policy maps each state to an action

action a selected in state s by policy π

- agent can select same action in multiple states
- some actions might not be selected in any state

In general, policy assigns probability to each action in each state

Stochastic Policy Notation

$\pi(a|s)$ → multiple actions maybe selected with non-zero prob.

$$\Rightarrow \sum_{a \in A(s)} \pi(a|s) = 1 \; ; \; \pi(a|s) \geq 0$$

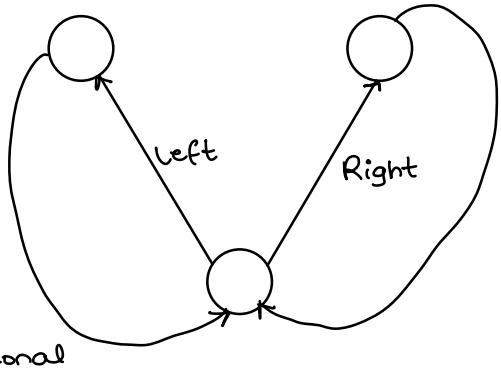
- It is important that policy depends only on the current state.

Example

- Consider two policies

1) L → 50% prob
   R → 50% prob

2) Alternate L & R
   (LRLR,... → not valid because conditional on prev. action.)



- A policy maps current state to a set of prob. for taking each action.

Value Functions (State, Action)

State value Functions

$v(s) \doteq E[G_t | S_t = s]$ →

Future reward the agent can according to π expect to receive starting from a particular state.

Expected return from a given state.

$v_\pi(s) \doteq E_\pi[G_t | S_t = s]$ (expectation computed w.r.t π (policy))

→ action selection

Action-value Functions

$q_\pi(s,a) \doteq \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a]$

↳ defines what happens when agent first selects a particular action

- Action-value of a state is the expected return if agent selects action a and then follows policy π.

Value Functions → predict rewards into the future

↳ they allow an agent to query the quality of its current situation instead of waiting to observe the long-term outcome.

- Two-fold benefit
  - ↳ the return is not immediately available
  - ↳ the return may be random due to stochasticity in both policy & environment dynamics.
- summarizes all the possible futures by averaging over returns
- help us judge quality of different policies

Example

→ consider an agent playing the game of chess. Chess is an episodic MDP. The state is given by all the positions on the board. (all)

→ the actions are the legal moves

→ termination occurs when game ends in either a win, lose or draw.

→ reward → 1 for win, 0 for all moves ⇒ does not tell us how well the agent is playing during the match.

- value func. tells us much more.
- The state value is equal to the expected sum of future rewards.

Since only non-zero reward is win, then $P(\text{win}) = V_\pi(S)$ state value is prob. of winning following current policy π.

- The opponent's move is the part of the state transition.

Example → simple continuing MDP

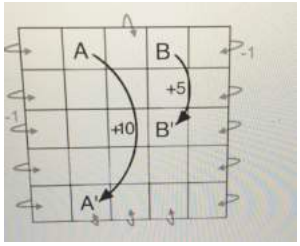
↳ states are defined by the locations on the grid

→ actions move the agent up, down, left or right.

→ bump generates → −1

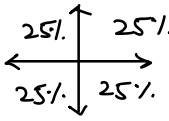
→ agent cannot move off the grid

Special states
- → A → every action in state A → +10
- → B → every action in state B → +5



⇒ Specify the policy before specifying the value function.

* Uniform Random Policy →



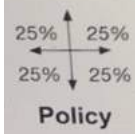
→ policy

- Since the task is continuing, we need gamma ($\gamma = 0.9$)

only +ve reward in MDP → filled value from value function.

	A			B
	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

$\gamma = 0.9$



highest but less than 10 even though every a = 10. Because every transition moves agent closer to lower wall.

because move to middle and unlikely bump.

these are very low because the agent is likely to bump into the wall before reaching A and B. the distant states

↳ State-value function → expected return from a given state under a specific policy

↳ Action-value function → expected return from a given state after taking a specific action & following a policy.

- RL → memorized content sensitive search – Barto

Bellman Equation Derivation

↳ formalize the connection b/w value of state & its successors

State-value function Bellman Equation

$$v_\pi(s) \doteq E_\pi[G_t \mid S_t = s] \quad -①$$

↳ relationship b/w value of state & value of its possible successor states

expected return starting from state s.

Return → discounted sum of future rewards → recall

$$\Rightarrow G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

from ①

$$v_\pi(s) \doteq E_\pi\left[R_{t+1} + \gamma G_{t+1} \mid S_t = s\right]$$

$$= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r \mid s, a)\left[r + \gamma E_\pi[G_{t+1} \mid S_{t+1} = s']\right]$$

↳ action choice only depends on current state

↳ next state & reward depend only on current state & action

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r \mid s, a)\left[r + \gamma v_\pi(s')\right]$$ → Bellman eq. of State-value fn

Action-value function Bellman Equation

$$q_\pi(s,a) \doteq E_\pi[G_t | S_t = s, A_t = a]$$

$$= \sum_{s'} \sum_{r} p(s', r | s, a)\left[r + \gamma E_\pi[G_{t+1} | S_{t+1} = s']\right]$$

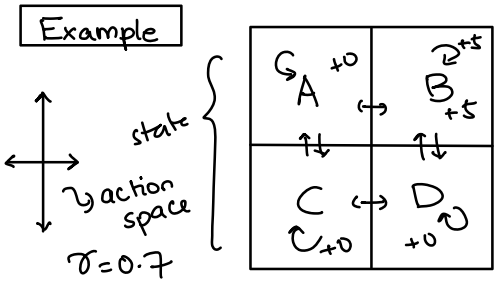
$$= \sum_{s'} \sum_{r} p(s', r | s, a)\left[r + \gamma \sum_{a'} \pi(a'|s') E_\pi[G_{t+1} | S_{t+1} = s', A_{t+1} = a']\right]$$

$$= \sum_{s'} \sum_{r} p(s', r | s, a)\left[r + \gamma \sum_{a'} \pi(a'|s') q_\pi(s', a')\right]$$

↳ Bellman eq. for action-value fn.

Why Bellman Equations?

Example



π = uniform random policy → 25% ↑ ↓ → ←

- How to work out the value of each state under the π?

$$V_\pi(A) \doteq E_\pi[G_t | S_t = A]$$
$$V_\pi(B) \doteq E_\pi[G_t | S_t = B]$$
$$V_\pi(C) \doteq E_\pi[G_t | S_t = C]$$
$$V_\pi(D) \doteq E_\pi[G_t | S_t = D]$$

Recall, value fn ($V_\pi(s)$)
→ expected return under policy pi. Average over return obtained by each sequence of actions an agent could possibly choose infinitely many futures.

Bellman State value fn

$$V_\pi(s) \doteq \sum_a \pi(a|s) \sum_r \sum_{s'} p(s', a' | s, a)\left[r + \gamma V_\pi(s')\right]$$

$$V_\pi(A) = \sum_a \pi(a|A)(r + 0.7 V_\pi(s'))$$

$$V_\pi(A) = \frac{1}{4}(5 + 0.7 V_\pi(B)) + \frac{1}{4}(0 + 0.7 V_\pi(C)) + \frac{1}{4} 0.7 V_\pi(A) - \frac{1}{4} 0.7 V_\pi(A)$$

(right; down; up; left)

$$V_\pi(A) = \frac{1}{4}(5 + 0.7 V_\pi(B)) + \frac{1}{4} 0.7 V_\pi(C) + \frac{1}{2} 0.7 V_\pi(A) \quad \text{—(i)}$$

Similarly,

$$V_\pi(B) = \frac{1}{2}(5 + 0.7 V_\pi(B)) + \frac{1}{4} 0.7 V_\pi(A) + \frac{1}{4} 0.7 V_\pi(D) \quad \text{—(ii)}$$

$$V_\pi(D) = \frac{1}{4}(5 + 0.7 V_\pi(B)) + \frac{1}{4} 0.7 V_\pi(C) + \frac{1}{2} 0.7 V_\pi(D) \quad \text{—(iii)}$$

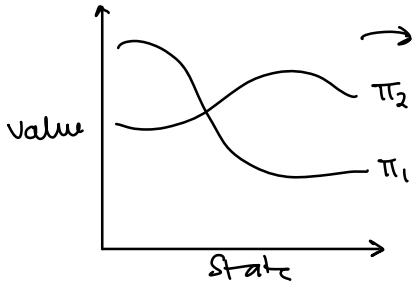
$$V_\pi(C) = \frac{1}{4} 0.7 V_\pi(A) + \frac{1}{4} 0.7 V_\pi(D) + \frac{1}{2} 0.7 V_\pi(C) \quad \text{—(iv)}$$

⇒ Solving we get, $V_\pi(A) = 4.2$, $V_\pi(B) = 6.1$, $V_\pi(C) = 2.2$, $V_\pi(D) = 4.2$

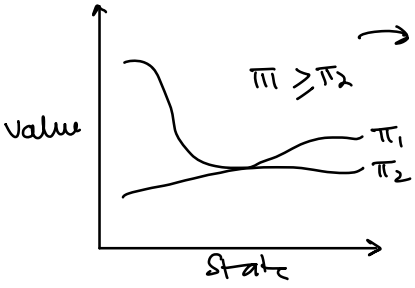
- This approach works for small games, MDPs, but not for large systems since constructing linear eq. for each state is not possible. Consider chess ⇒ 10^{45} states approx.

Optimal Policy → policy with highest value in all states

- How is one policy better than other policy?



→ we cannot say $\pi_2 > \pi_1$ because atleast equal value or greater is required in all states.

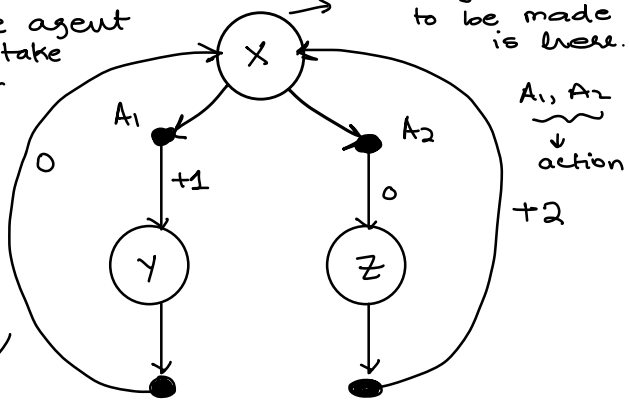


→ we can say that π_1 is better because π_1 is as as good as or better than π_2.

- An optimal policy π_* is a policy which is as good as or better than all other policies.
- An optimal policy will have highest value in every state.
- There is always an optimal policy.
- Policy specifies how an agent behaves. Given this way of behaving, we then aim to find the value function.

Example

- the agent can take either A_1 or A_2



the only decision to be made is here.

A_1, A_2 ↓ action

$\pi_1(X) = A_1, \quad \pi_2(X) = A_2$

- two deterministic MDPs

⇒ optimal policy → the value of X is highest

answer → γ

lets $\gamma = 0$

$v_{\pi_1}(X) = 1$ (value only on immediate reward)

$v_{\pi_2}(X) = 0$

$\pi_1 \rightarrow$ optimal

$\gamma = 0.9$

$v_{\pi_1}(X) = 1 + (0.9)(0) + (0.9)^2(1) + \dots$

$v_{\pi_1}(X) = \sum_{k=0}^{\infty} (0.9)^{2k}$

* Two choice MDP

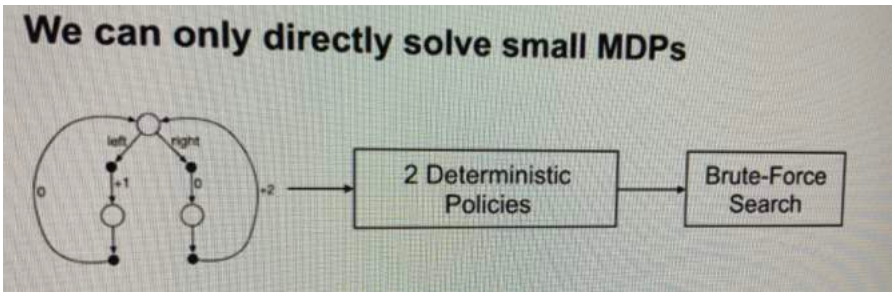
$v_{\pi_1}(X) = \frac{1}{1-0.9^2} = 5.3$

$v_{\pi_2}(X) = 0 + (0.9)(2) + (0.9)^2 (0) + \dots$

$v_{\pi_2}(X) = \sum_{k=0}^{\infty} (0.9)^{2k+1} \times 2$

$v_{\pi_2}(X) = \frac{0.9}{1-0.9^2} \times 2 \simeq 9.5$

$\pi_2 \rightarrow$ optimal in this case.



- Finding optimal policy was easy.
- Only 2 choice and we could brute force.
- Not easy generally

A General MDP → $|A|^{|S|}$ Deterministic Policies ~~→~~ Brute Force Search

↳ too huge

- Can be solved using Bellman equations.

Bellman optimality equations

Why do we need both (state value (V) func., action-state value (Q) func.)?

- $V_\pi(s)$ → expresses the expected value of following policy π forever when the agent starts following it from state s.
- $Q_\pi(a,s)$ → expresses the expected value of first taking action a from state s and then following policy π forever.

Optimal Value Functions

Recall ⇒ $\pi_1 \geq \pi_2$ iff $V_{\pi_1}(s) \geq V_{\pi_2}(s)$ $\forall s \in S$

V_* $\quad V_{\pi_*}(s) \doteq E_{\pi_*}[G_t \mid S_t = s] = \max_\pi V_\pi(s)$ $\forall s \in S$

All optimal policies have optimal state value func. V_*

↳ state-action value func. Q_*

q_* $\quad q_{\pi_*}(a,s) \doteq E_{\pi_*}[G_t \mid S_t = s, A_t = a] = \max_\pi q_\pi(a,s)$ $\forall s \in S, a \in A$

Bellman eq. for state-value func.

$$V_\pi(s) = \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r \mid s, a)\left[r + \gamma V_\pi(s')\right]$$

↳ holds for any policy

$$V_*(s) = \sum_a \pi_*(a|s) \sum_{s'} \sum_r p(s', r \mid s, a)\left[r + \gamma V_*(s)\right]$$ → Bellman eq. for V_*

$$V_*(s) = \max_a \sum_{s'} \sum_r p(s', r \mid s, a)\left[r + \gamma V_*(s')\right]$$ → Bellman Optimality Eq. for V_*

- A deterministic optimal policy always exists. The optimal deterministic policy selects an optimal action in every state. Such a policy will assign prob. 1 for an action that achieves highest value and prob. 0 for all other actions.

Bellman eq. for state-action value func.

$$q_\pi(s,a) = \sum_{s'}\sum_{r} p(s',r|s,a)\left[r + \gamma \sum \pi(a'|s')\, q_\pi(s',a')\right]$$

$$q_*(s,a) = \sum_{s'}\sum_{r} p(s',r|s,a)\left[r + \gamma \sum \underbrace{\pi_*(a'|s')}_{\text{optimal policy}} q_*(s',a')\right]$$

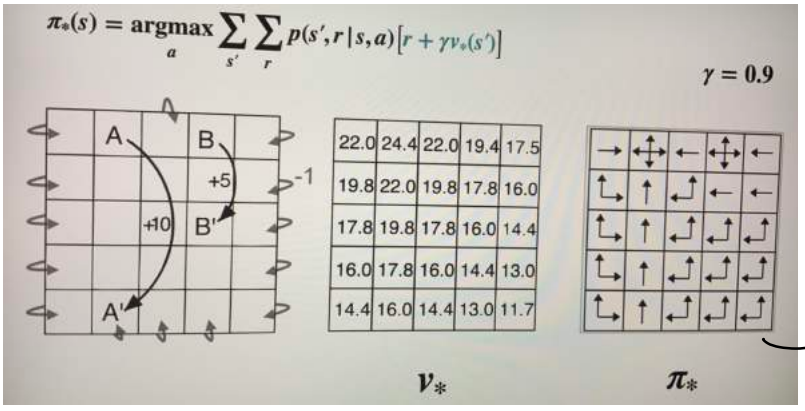
$$q_*(s,a) = \sum_{s'}\sum_{r} p(s',r|s,a)\left[r + \gamma \max_{a'} q_*(s',a')\right] \rightarrow \text{Bellman Optimality Eq. for } q_*$$

Using Optimal Value Func. to Get Optimal Policy

$$v_*(s) = \max_a \sum_{s'}\sum_{r} p(s',r|s,a)\left[r + \gamma v_*(s')\right]$$

$$\pi_*(s) = \operatorname*{argmax}_a \sum_{s'}\sum_{r} p(s',r|s,a)\left[r + \gamma v_*(s')\right]$$

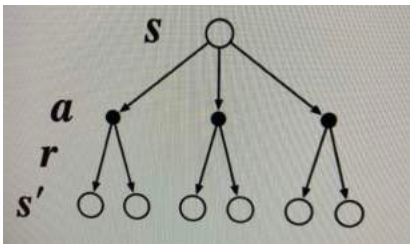
- For any state, look at each available action and evaluate
- For some action, this will obtain maximum. A deterministic policy which selects max. action for each state, will necessarily be optimal, since it obtains the highest possible value.



→ optimal policy filled in using $\pi_*(s)$ and solving for $[r + \gamma v_*(s')]$

- In general, the dynamics func. p can be stochastic and it might not be simple.

- Using v_* to get to π_*, requires one step look ahead →



- However, if we have access to q_*, it is even easier to come up with the optimal policy.

$$\pi_*(s) = \operatorname*{argmax}_a q_*(s,a)$$

- we do not need one-step look ahead.
- we only have to select any action a, that maximizes q_* of a.
- The state-action value func. caches the results of one-step look ahead for each action.

✓ A policy depends only on the current state. The state should provide the agent with all the information it needs to make good decision.

Week # 4

Two distinct tasks → policy evaluation and control

⇒ Policy Evaluation ⇒ the task of working out the value function for a specific policy.

⇒ Control ⇒ the task of finding a policy to obtain as much as reward as possible. Finding a policy which maximizes the value function.

↳ the ultimate goal of reinforcement learning

- The task of policy evaluation is usually the necessary first step.

Policy Evaluation

$\pi \rightarrow v_\pi$ Recall that: $v_\pi(s) \doteq \mathbb{E}_\pi[G_t \mid S_t = s]$

$$G_t \doteq \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r \mid s, a)\left[r + \gamma v_\pi(s')\right] \rightarrow \text{one per state}$$

$\pi, p, \gamma \rightarrow$ [Linear System Solver] $\rightarrow v_\pi$

↳ can be solved using LA.

However, in practice, use DP.

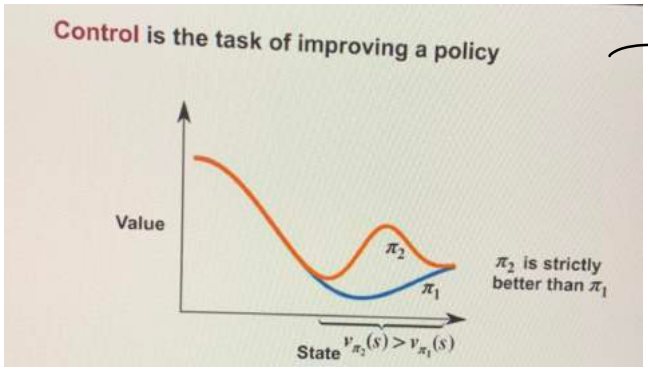
$\pi, p, \gamma \rightarrow$ [Dynamic Prog] $\rightarrow v_\pi$

(p: dynamics of env.)

↳ suitable for general MDPs.

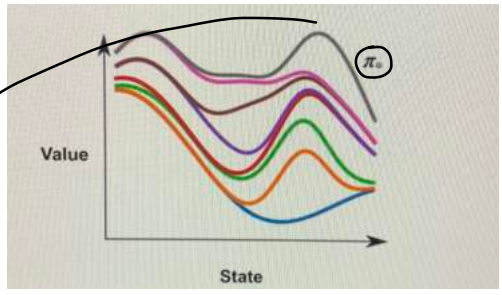
Control

→ task of improving policy



→ the goal of control task is to modify policy to produce a new one which is strictly better.

- We can try to improve a policy repeatedly to obtain a sequence of better & better policies.



- the iterative method can no longer find policy better than the current policy, so currently policy is optimal policy π_*.

$\pi, p, \gamma \longrightarrow$ [Dynamic Prog.] $\longrightarrow v_\pi$ Policy evaluation

$p, \gamma \longrightarrow$ [Dynamic Prog.] $\longrightarrow \pi_*$ Control

Iterative Policy Evaluation

- DP algorithms are obtained by turning Bellman equations into update rules.
  - the first of these algorithms.

Recall that Bellman eq. for $v_\pi(s)$ is recursive.

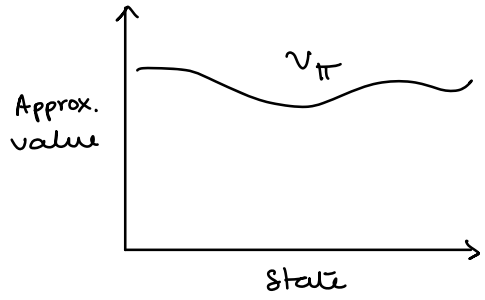
$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r | s, a)\left[r + \gamma v_\pi(s')\right]$$

↓ use this update rule

$$v_{k+1}(s) \leftarrow \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r | s, a)\left[r + \gamma v_k(s')\right]$$

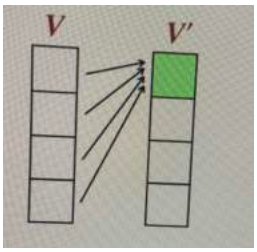
- Begin with arbitrary initialization and then use the update rule to better approx. the value function.
- Each iteration applies the update to every state s in S (state space) — known as the sweep.
- can be used to iteratively refine our estimate of value function.
- Produces a sequence of better and better approximations to value functions.

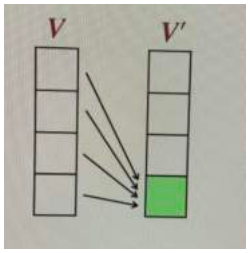
- If the update leaves the value function approximation unchanged, $v_{k+1} = v_k \ \forall s \in S$, then $v_k = v_\pi$ and the value func. is found. v_π is the unique sol. to the Bellman equation.

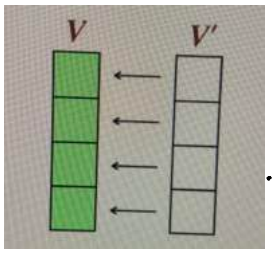


For any v_0 $\lim_{k \to \infty} v_k = v_\pi$

- To implement this, we store two arrays V and V'. Each has one entry for every state.
- $V \rightarrow$ stores current approx. value func.
- $V' \rightarrow$ stores the updated values.
- This way we can compute new values from the old one state at a time. without old values being changed.
- After the full sweep, all new values are written into V and do next iteration.

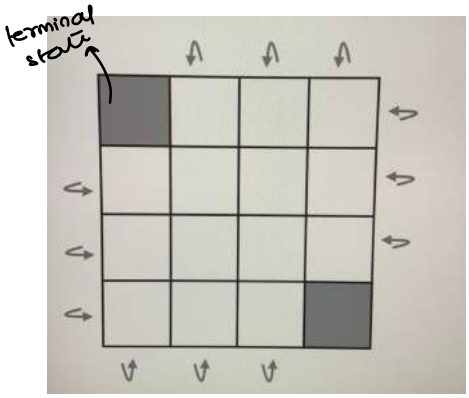






- One array version can also be used and some updates will use new values instead of old ones. This method also converges and might also converge faster: gets to use updated values sooner.

Example → episodic MDP, grid world



$R = -1$ (for every transition)

$\gamma = 1$ (episodic problem)

↳ undiscounted case

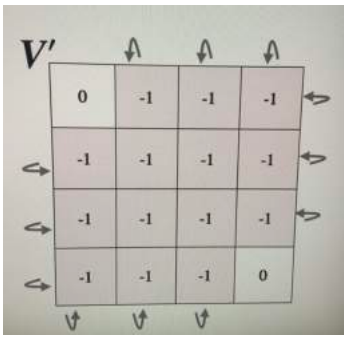
↔↕ } deterministic actions

Policy → uniform random

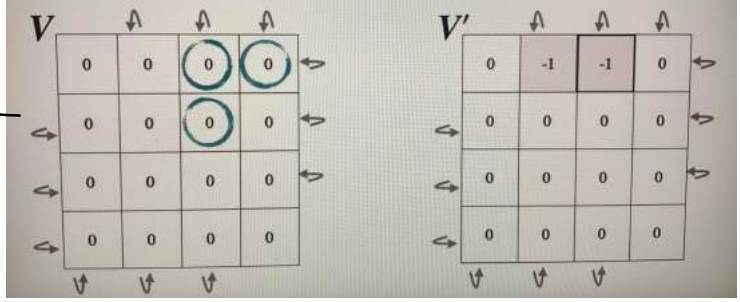
$$V'(s) \leftarrow \sum_a \pi(a|s) \sum_{s'} \sum_r p(s',r|s,a)[r+\gamma V(s')]$$

(deterministic)

$0.25*(-1+0)+0.25*(-1+0)+0.25*(-1+0)+0.25*(-1+0) = -1$

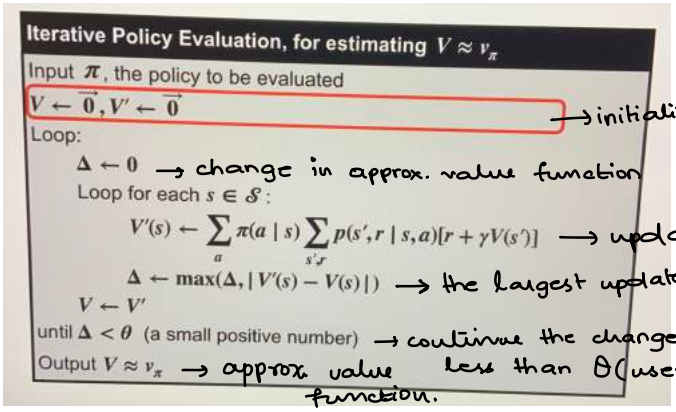


after full sweep



$v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_\pi$

Algorithm



→ initialization

→ change in approx. value function

→ update rule

→ the largest update in given state

→ continue the change is less than θ (user specified constant)

→ approx value function.

Policy Improvement

- Control task

Recall that: (greedy action)

$$\pi_*(s) = \arg\max_a \sum_{s',r} p(s',r|s,a)[r+\gamma v_*(s')]$$

- Given v_*, we can find the optimal policy by choosing the greedy action.
- The greedy action maximizes the Bellman optimality eq. in each state.

- Instead of optimal value function, select an action which is greedy w.r.t the v_π of an arbitrary policy π.

$$\pi(s) = \arg\max_a \sum_{s',r} p(s',r|s,a)[r+\gamma v_\pi(s')] \quad \forall s \in S$$

- This new policy must be different than π. If the greedification doesn't change Pi, then Pi was already greedy w.r.t its own value function.
- $V_\pi \rightarrow$ obeys the Bellman optimality eq; $\rightarrow \pi$ is already optimal
- The new policy obtained must be strict improvement over π unless π was already optimal.

Policy Improvement Theorem

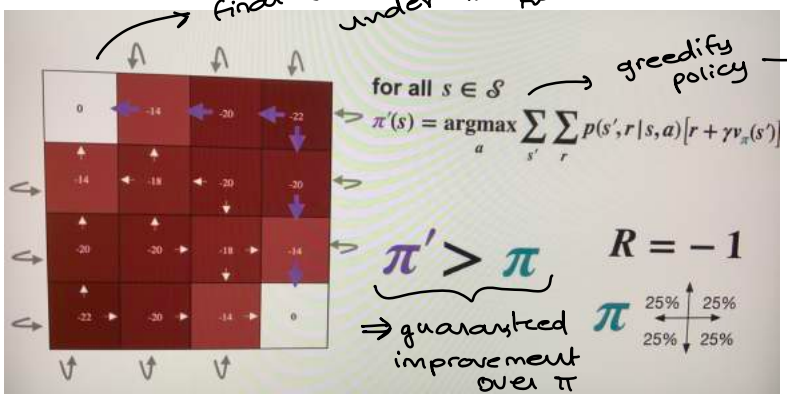
$\rightarrow q_\pi(s, \pi'(s)) \geq q_\pi(s, \pi(s)) \quad \forall s \in S \rightarrow$ tells value of state taking action a and then following policy π.
$\hookrightarrow \pi' \geq \pi$

- action taken according to π' and then follow π
- If this action has higher value than action under π, then $\pi' \geq \pi$.

act according to π' in current state, π for all others.

$\Rightarrow q_\pi(s, \pi'(s)) > q_\pi(s, \pi(s))$ for atleast one $s \in S \rightarrow \pi' > \pi$

Example



→ in each state select the action that leads to next state with highest value.
- In this, least negative value.

- Although the value func. was not optimal, the greedy policy (π') w.r.t V_π is optimal. This is not true so easily always.
- More generally, the policy improvement theorem only guarantees that the new policy is an improvement on the original.

Policy Improvement Theorem ⇒ greedified policy is a strict improvement unless the original policy was already optimal.

Policy Iteration

→ find an optimal policy by iteratively evaluating and proving a sequence of policies.

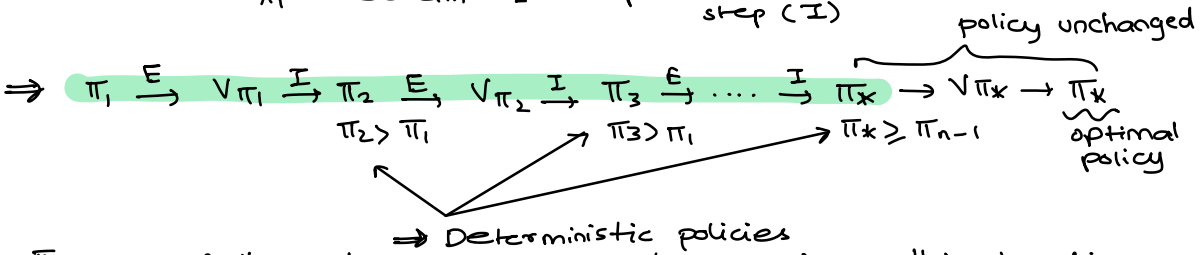
→ Policy Iteration Algorithm → finds the optimal policy.

→ Dance of policy & value → how policy iteration reaches the optimal policy by alternating between evaluating a policy and improving it.

- Apply policy iteration to compute optimal policies and optimal value functions.

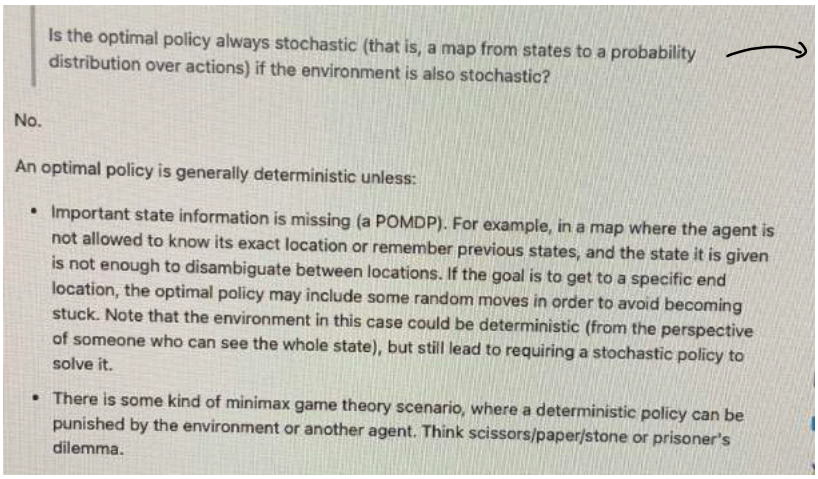
Policy Iteration

- Lets say that we begin with policy π_1. We can evaluate π_1 using iterative policy evaluation to obtain V_{π_1} → evaluation step (E)
- Using the results of policy improvement theorem, we can greedify w.r.t V_{π_1} to obtain π_2 → improvement step (I)



- There are finite number of deterministic policies, so this iterative improvement must always reach optimal policy.

Q- When/how to define policy as deterministic or stochastic?


Is the optimal policy always stochastic (that is, a map from states to a probability distribution over actions) if the environment is also stochastic?

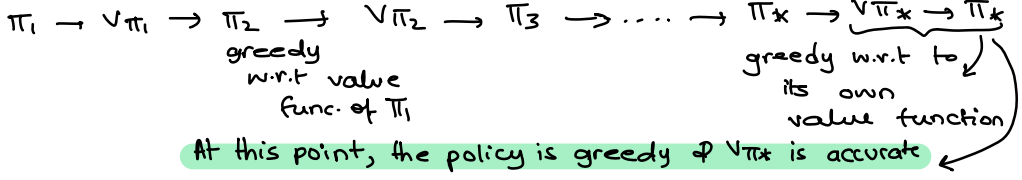
No.

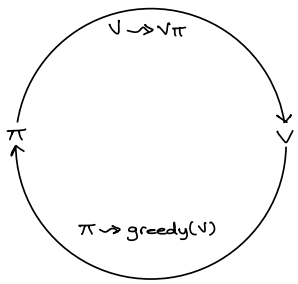
An optimal policy is generally deterministic unless:

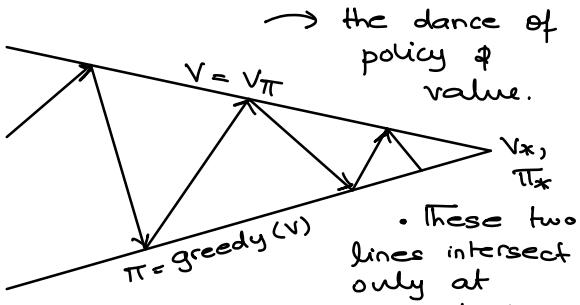
- Important state information is missing (a POMDP). For example, in a map where the agent is not allowed to know its exact location or remember previous states, and the state it is given is not enough to disambiguate between locations. If the goal is to get to a specific end location, the optimal policy may include some random moves in order to avoid becoming stuck. Note that the environment in this case could be deterministic (from the perspective of someone who can see the whole state), but still lead to requiring a stochastic policy to solve it.
- There is some kind of minimax game theory scenario, where a deterministic policy can be punished by the environment or another agent. Think scissors/paper/stone or prisoner's dilemma.

(SC) (Neil Slater)

- This method of finding the optimal policy is called policy iteration.
- Policy iteration consists of two distinct steps repeated over and over → eval, improvement.







→ the dance of policy & value.

- These two lines intersect only at optimal policy & value fn.

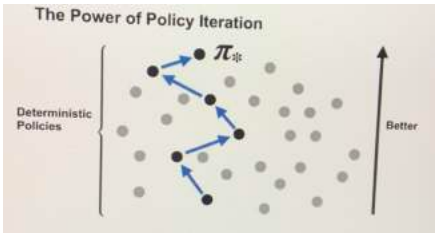
Algorithm

Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$ → initialize any way

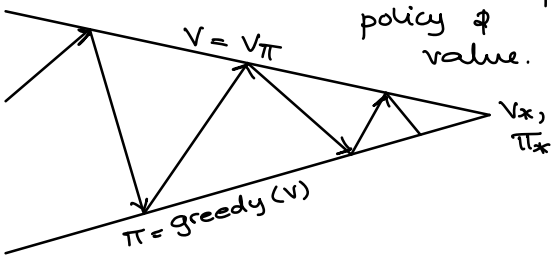
2. Policy Evaluation
   Loop:
   $\Delta \leftarrow 0$
   Loop for each $s \in \mathcal{S}$:
   $v \leftarrow V(s)$
   $V(s) \leftarrow \sum_{s',r} p(s', r|s, \pi(s))[r + \gamma V(s')]$
   $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)
   } evaluate

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
   old-action $\leftarrow \pi(s)$
   $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
   If *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   } improve → greedify
   If *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2

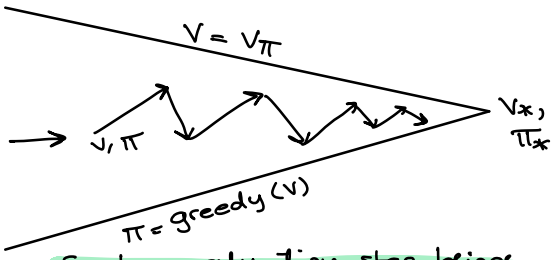


- Policy iteration cuts through the search space.

Generalized Policy Iteration



→ the dance of policy & value.



- Each evaluation step brings us little closer to the value of the current policy.
- Each policy improvement, makes policy little more greedy but not totally greedy.

1) Value Iteration

- We still sweep over all the states, and greedify w.r.t the current value function. However, we do not run policy evaluation to completion.
- Perform one sweep over all the states. After that greedify again.

Update Rule becomes:

$$V(s) \leftarrow \max_a \underbrace{\sum_{s',r} p(s', r|s, a)}_{\text{update according to action that maximizes current value estimate.}} [r + \gamma V(s')]$$

- Value iteration sweeps the entire state space on each iteration just like policy iteration.
- Methods that perform systematic sweeps → synchronous
  ↳ problematic if state space is large

- Asynchronous DP → update value of states in any order. They do not perform systematic sweeps. They might update a given state many times before another is updated even once.
  ↳ to guarantee convergence → the asynchronous algorithms must update all states.

- Value iteration → allows to combine policy evaluation and improvement in a single step.

Value Iteration, for estimating $\pi \approx \pi_*$

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
  $\Delta \leftarrow 0$
  Loop for each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \max_a \sum_{s',r} p(s',r|s,a)[r + \gamma V(s')]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Output a deterministic policy, $\pi \approx \pi_*$, such that
$\pi(s) = \text{argmax}_a \sum_{s',r} p(s',r|s,a)[r + \gamma V(s')]$ } improvement

Efficiency of Dynamic Programming

- The value of each state can be treated as totally independent estimation problem.

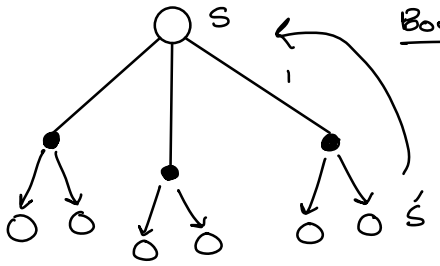
$$v_\pi(s) \doteq E_\pi[G_t | S_t = s]$$

- Gather a large number of returns under π and take their average.
  ↳ Monte-Carlo method
- We may need large number of returns from each state.

- Since we have to get many returns to make sure average converges, this is not efficient.

- The DP allows the use of other value estimates we have already computed.

⇒ The process of using value estimates of successor states to improve current value estimate is called bootstrapping.



Bootstrapping

→ much more efficient than MC that computes each state/value independently.

Brute-force Search

- Policy iteration computes optimal policies. Brute-force search is alternative.
- Evaluate every possible deterministic policy one at a time.
- Pick the one with highest value.
- Since finite policies (deterministic) and optimal policy always exists, the approach finds the answer.

- However, the number the policies can be huge.
- A deterministic policy consists of one action choice per state.
- So the total number of policies is exponential.
  ↳ $|A|^{|S|}$

- Generally, solving MDP gets harder as number of states grow.

Curse of Dimensionality

- The size of state space grows exponentially as number of relevant features increase.

Course # 02 - Sample-based Learning Methods

- Sample-based learning → Learn solely from trial and error don't necessarily model, can just learn from data.

Monte-Carlo → used broadly for any method that relies on repeated random sampling.

- In RL, MC methods allow us to estimate values directly from experience, from sequences of states, actions and rewards.

- Learning from experience ⇒ agent can accurately estimate a value function without prior knowledge of the environment dynamics.

⇒ To use a pure DP approach → the agent needs to know the environment transition probilities. However, in some problems, we do not know the transition probs.
↳ Computation is tedious and error prone.

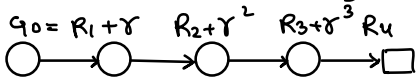
↙ MC methods don't sweep through all possibilities.
Instead they estimate values by averaging over a large number of random samples.

In RL, the goal is to learn value function.

$$v_\pi(s) \doteq E_\pi[G_t | S_t = s]$$

↳ represent expected return

- MC method for learning a value function would first observe multiple returns from the same state. Then they average those observed returns to estimate the expected return from that state.
- The more returns the agent observes from the state, the more likely it is the sample average is close to the state value.
- These returns could only be observed at the end of an episode.

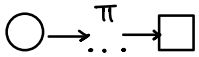


↳ So, MC only for episodic tasks.
⇒ the tasks that have a terminal state.

- MC methods look a bit like bandits.

⇒ In bandits, the value of an arm is estimated using the average payoff sampled by pulling that arm (taking that action).

⇒ MC methods consider policies instead of arms.



- The value of state S under a given policy is estimated using the average return sampled by following that policy from S to termination.

⇒ Repeat for many episodes and learn good estimate for value func.

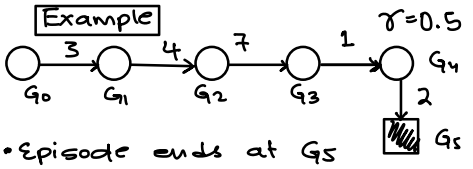
MC prediction, for estimating $V \approx v_\pi$

Input: a policy π to be evaluated
Initialize:
  $V(s) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$
  $Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$
Loop forever (for each episode):
  Generate an episode following π: $S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$
  $G \leftarrow 0$
  Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
    $G \leftarrow \gamma G + R_{t+1}$
    Append G to $Returns(S_t)$
    $V(S_t) \leftarrow$ average$(Returns(S_t))$

→ Repeat for many episodes and learn good estimate for value func.

→ working backwards to avoid duplicate computations

Example



- Episode ends at G_5

$G_5 = 0$

$G_0 = R_1 + \gamma G_1$
$G_1 = R_2 + \gamma G_2$
$G_3 = R_3 + \gamma G_3$
$\vdots$
$G_5 = 0$

each return is included for prev. time's return so, we can avoid duplicates by computing backward.

$$G_0 = R_1 + \gamma R_2 + \gamma^2 R_3 + \gamma^3 R_4 + \gamma^4 R_5 = 7$$
$$G_1 = R_2 + \gamma R_3 + \gamma^2 R_4 + \gamma^3 R_5 = 8$$
$$G_2 = R_3 + \gamma R_4 + \gamma^2 R_5 = 8$$
$$G_3 = R_4 + \gamma R_5 = 2$$
$$G_4 = R_5 = 2$$
$$G_5 = 0$$

$G_5 = 0$
$G_4 = R_5 + \gamma G_5 = 2$
$G_3 = R_4 + \gamma G_4 = 2$
$G_2 = R_3 + \gamma G_3 = 8$
$G_1 = R_2 + \gamma G_2 = 8$
$G_0 = R_1 + \gamma G_1 = 3$

- We can avoid keeping all the sampled returns in a list, by using this incremental update rule.

NewEstimate ← OldEstimate + StepSize [Target − OldEstimate]

Monte-Carlo for Prediction

- Formulating Black Jack as an undiscounted MDP where each game of back jack corresponds to an episode.

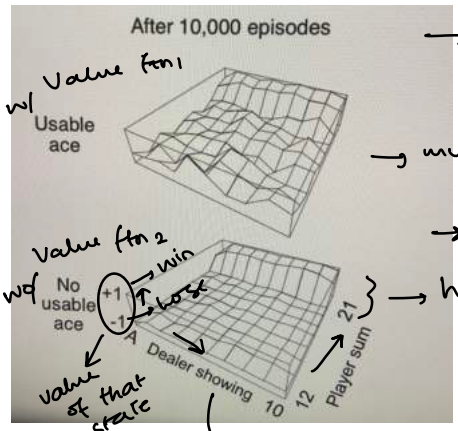
Rewards: −1 for loss, 0 for draw, 1 for win (given at end of game)
Actions: Hit or Stick
States:
- Whether the player has usable Ace (Yes or No) → 11 Ace
- Sum of cards
- Card the dealer shows

→ 200 states total
→ Cards are dealt from deck with replacement. → there's no point in keeping track of cards that have been dealt and state respect's Markov property.
→ Policy: Stops requesting cards when player sum is 20 or 21.

S (Usable Ace, Sum, Dealer)	Returns(S)	V(S)
A = (NoAce, 20, 10)	Returns(A) = [1]	1
B = (NoAce, 13, 10) ← first state	Returns[B] = [1]	1 ← average

first episode

Last non-terminal state (second state) since first episode ended with two states.



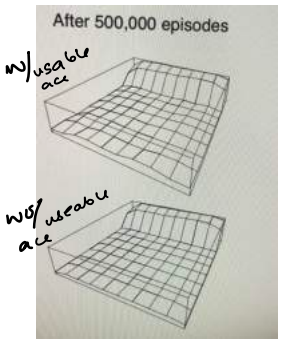
→ value function after 10,000 episodes.

→ much rough because very few games have usable Ace.

→ both plots have similar shape.

→ higher here because agent sticks at 20 or 21 and is likely to win.
- whereas agent hits at 19 and lower and is likely to lose.

→ looks like the card the dealer is showing has little to no impact



→ the estimates have nearly converged to state values.
→ the plots are much smoother now.
→ same pattern is seen as before.

Implications of MC Methods

- We do not keep a large model of the environment since MC methods learn directly from experience.
- MC methods estimate value of an individual state independent of values of other states.
- In DP, the value of each state depends on other states.
- The computation needed to update the value of each state doesn't depend on size of MDP, depends on the length of episode.

Monte-Carlo for Action Values

→ estimate action-value func. using MC.
→ Almost the same as state-value.
→ we learn state values by sampling average returns from that state.

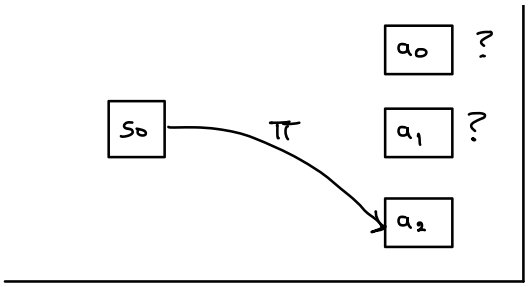
For action values, recall that → $q_\pi(s,a) \doteq E_\pi[G_t \mid S_t = s, A_t = a]$

- We collect returns following a policy from state-action pair and then take their average.

$\underset{a}{\text{argmax}}\, q_\pi(s,a)$ → allow us to compare different actions in the same state and we can switch to a better action if one is available.
→ only possible if we have estimate of values of other actions.

⇒ Imagine an action that is never selected by the policy. The agent will never observe returns corresponding to that action. We won't be able to form an accurate MC estimate.

→ The agent must try all the actions in each state in order to learn their values. ⇒ Problem of Maintaining Exploration in RL

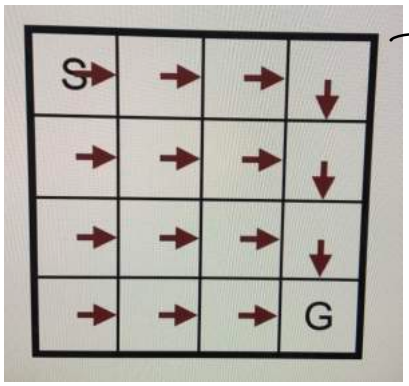


s_0, a_0 $s_1, a_1, s_2, a_2, \ldots$
random from π & p

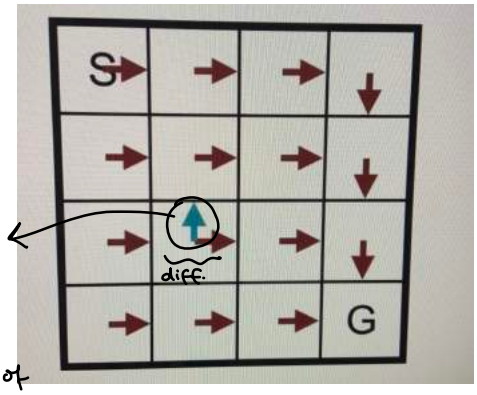
Exploring Starts

⇒ One way to maintain exploration is called exploring starts.

⇒ Guarantee that episode starts in every state, action pair. After that the agent simply follows its own policy.

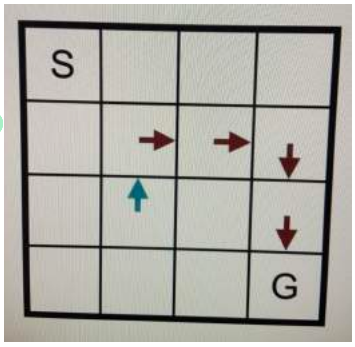


→ policy shown in red.



→ exploring starts will randomly sample a state and action at the start of an episode, shown in blue.

- After the initial action, the agent will follow the policy (red arrows) until the episode ends.
- We must be able to set the start state this way to evaluate the deterministic policy.
- May not always be possible.



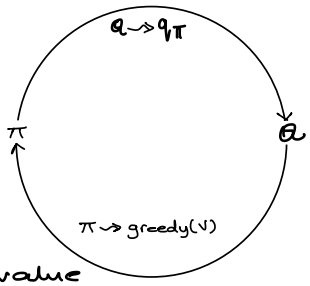
→ Other exploration strategies like epsilon-greedy, can be used to evaluate stohastic policies.

MC for Generalized Policy Iteration (GPI)

- includes a policy evaluation and policy evaluation.

$\pi_0 \rightarrow \pi_1 \rightarrow \pi_2 \ldots$

- GPI → produce sequences of policies that are atleast as good as the policies before them.



- In GPI framework, the value estimates only improve a little, not all the way to the correct action values.
- For convergence, the estimates should continue to improve.

- For the policy improvement step, we can make the policy greedy w.r.t the agent's current action-value estimates.

$\pi_{k+1}(s) \doteq \underset{a}{\text{argmax}}\, q_{\pi_k}(s,a)$

- For the policy evaluation, MC prediction
  ↳ MC method to estimate action values

→ MC control methods combine policy improvement and policy eval on an episode-by-episode basis.

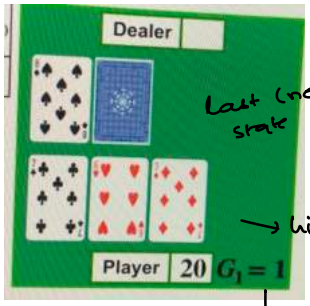
Monte Carlo ES (Exploring Starts), for estimating $\pi \approx \pi_*$

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

Loop forever (for each episode):
Choose $S_0 \in \mathcal{S}$, $A_0 \in \mathcal{A}(S_0)$ randomly such that all pairs have probability > 0 → random state, action pair
Generate an episode from S_0, A_0, following π: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$ → generate episode following policy π
$G \leftarrow 0$
Loop for each step of episode, $t = T-1, T-2, \ldots, 0$: → backward loop
$G \leftarrow \gamma G + R_{t+1}$
Append G to $Returns(S_t, A_t)$
$Q(S_t, A_t) \leftarrow$ average$(Returns(S_t, A_t))$
$\pi(S_t) \leftarrow \text{argmax}_a Q(S_t, a)$ → policy improvement

Example Applying MC GPI to black jack.

↓ randomly choose ∈ {hit, stick}



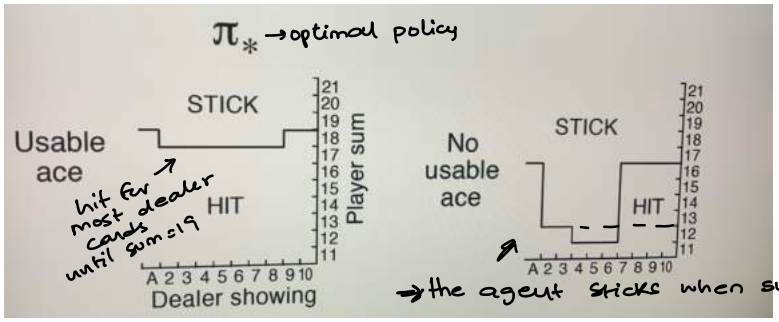
→ hit chosen randomly
drawn 7
total = 20
now according to old stick
dealer then hit and went over
so we win, reward of 1.

S, A	Returns (S, A)	Q(S, A)		π(S)
		Hit	Stick	
X, Stick	Returns (X, Stick) = [1]	0	1	Stick → greedy over actions
Y, hit	Returns (Y, hit) = [1]	1	0	hit

work backwards ↓

X = (No Ace, 13, 8) → 1st
Y = (No Ace, 20, 8) → Last
} episode

- Repeat over many episodes.



- Without a useable use, the policy depends on cards dealer is showing.
→ the agent sticks when sum is 13 or greater & dealer has low card.

- With useable ace, the agent has lot more flexibility in calculating sum of its cards. So policy is aggressive.

Epsilon-soft Policies

- Exploring-starts ⇒ must be able to start from every possible state-action pair. Otherwise the agent may not explore enough and could converge to sub-optimal solution.
- It is difficult to start in all possible states → Self driving car.

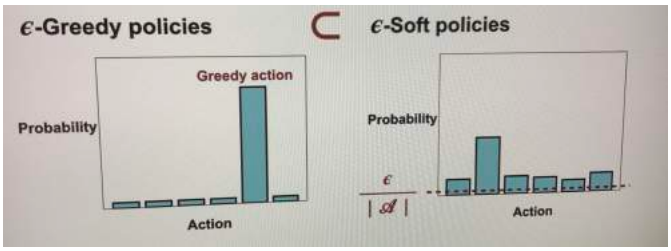
ε-greedy exploration ⇒ can be used with MC.

↳ policies are stohastic policies. They usually take greedy action, but occassionally take random action.

ε-greedy ⊂ ε-soft ↳ larger set of policies.

- ε-soft policies take each action with epsilon/|A|
- uniform random → ε-soft policy.

↳ $\frac{\varepsilon}{|A|}$



- ε-soft policies force the agent to continually explore and that means we can drop the exploring starts requirement from the MC control algo.
- ε-soft policy assigns non-zero prob. to each action in every state because of this ε-soft agents continue to visit all state-action pairs indefinitely.

- ε-soft policies are always stohastic. Return the prob. of taking action in each state.
- All actions have prob. of atleast $\varepsilon/|A|$. They will eventually try all the actions.

⇒ If our policy always gives atleast ε prob. to each action, its impossible to converge to a deterministic optimal policy.

- Exploring starts can be used to find optimal policy but ε-soft can be only used to find optimal ε-soft policy, which may not be optimal always.

⇒ policy with highest value in each state out of all ε-soft policies.

⇒ optimal ε-soft policy performs reasonably worse than optimal policy in general, but performs reasonably well and allows to get rid of exploring starts.

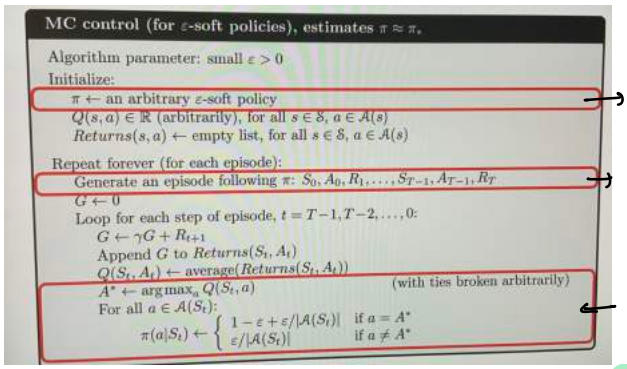
- Improvement step is

$$A^* \leftarrow \arg\max_a Q(S_t, a)$$

$$a \in A(S_t):$$

$$\pi(a|S_t) \leftarrow \begin{cases} 1-\varepsilon+\frac{\varepsilon}{|A|} & \text{if } a = A^* \\ \frac{\varepsilon}{|A|} & \text{if } a \neq A^* \end{cases}$$

ε-greedy w.r.t action-value estimate



MC control (for ε-soft policies), estimates $\pi \approx \pi_*$

Algorithm parameter: small $\varepsilon > 0$

Initialize:

$\pi \leftarrow$ an arbitrary ε-soft policy

$Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

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

Repeat forever (for each episode):

Generate an episode following π: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:

$G \leftarrow \gamma G + R_{t+1}$

Append G to $Returns(S_t, A_t)$

$Q(S_t, A_t) \leftarrow$ average$(Returns(S_t, A_t))$

$A^* \leftarrow \arg\max_a Q(S_t, a)$ (with ties broken arbitrarily)

For all $a \in \mathcal{A}(S_t)$:

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

→ the initial policy must be ε-soft e.g. uniform random

→ generates episode following its ε-soft policy.

improvement of policy

← New policy is ε-greedy w.r.t to action value estimates.

Optimal ε-soft policy = ε-greedy

Off-policy learning for Prediction

→ A way of learning value functions.

→ help deal with exploration problem.

- On-policy: improve and evaluate the policy being used to select actions.

  Policy evaluation ⇒ learning the value function.

  Control ⇒ learning the optimal policy

- Off-policy: improve and evaluate a different policy from the one used to select actions.

  ↳ the policy agent is learning is off the policy that is used for action selection.

  Target Policy → the policy the agent is learning $\pi(a|s)$

  → value function that the agent is learning is based on the target policy.

  → optimal policy is the target policy.

  Behavior policy → the policy the agent is using to select actions because it defines the agent's behavior. $b(a|s)$

  - selects the action for the agent.
  - generally an exploratory policy.

Decoupling behavior policy from target policy because it provides another strategy for continual exploration.

- If agent behaves according to the target policy it might only experience a small number of states.
- If agent can behave according to a policy that encourages exploration, it can experience a much larger number of states.

⇒ The behavior policy must cover the target policy ⇒ $\pi(a|s) > 0$ where $b(a|s) > 0$

On policy: $\pi(a|s) = b(a|s)$

- prob. of selecting action a given state s

Importance Sampling → allows off-policy learning.

Sample: $x \sim b$ → sampled from prob. b
(x: random variable)

Estimate: $E_\pi[X]$
↳ target distribution π

- Because x is drawn from b, cannot simply use the sample average.

$$E_\pi(x) \doteq \sum_{x \in X} x\,\pi(x)$$

$$= \sum_{x \in X} x\,\pi(x)\frac{b(x)}{b(x)}$$ → prob. of observed outcome x under b.

$$= \sum_{x \in X} x\frac{\pi(x)}{b(x)}\,b(x)$$

$\frac{\pi(x)}{b(x)}$: ratio b/w π & b
↓
importance sampling ratio

$$= \sum_{x \in X} x\,p(x)\,b(x)$$

$p(x)$: rho of x (importance sampling ratio)

Let $xp(x)$ be a new random variable, then

$$E_\pi[X] = E_b[X\,p(X)] = \sum_{x \in X} x\,p(x)\,b(x)$$

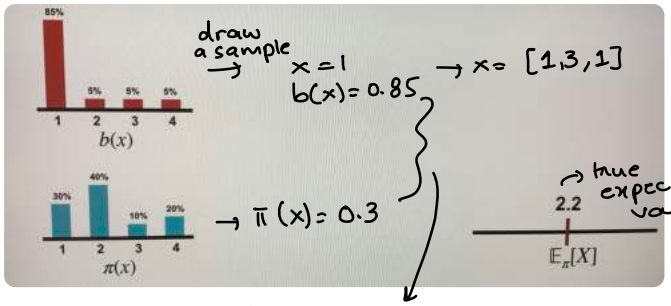
Expectation from data,

$$E[X] = \frac{1}{n}\sum_{i=1}^{n} x_i$$ → weighted sample average

$$E_\pi[X] = \frac{1}{n}\sum_{i=1}^{n} x_i\,p(x_i)$$

$x_i \sim b$

Example



$$E_\pi[X] = \frac{1}{n}\sum_{i=1}^{n} x_i\,p(x_i) \Rightarrow$$

1) $\dfrac{1 \times \frac{.3}{.85}}{1} = 0.35$

2) $x = 3,\ b(x) = 0.05$
$\pi(x) = 0.1$

$$= \frac{\left(1 \times \frac{.3}{.85}\right) + \left(3 \times \frac{0.3}{0.05}\right)}{2} = 3.18$$

3) $x = 1$
$b(x) = 0.85$
$\pi(x) = 0.3$

$$= \frac{\left(1 \times \frac{.3}{.85}\right) + \left(3 \times \frac{.1}{.05}\right) + \left(1 \times \frac{.3}{.85}\right)}{3}$$

$$= 2.24 \approx 2.2$$

↓
true expected value.

Off-Policy Prediction with Monte Carlo

$$V_\pi(s) \doteq E_\pi[G_t \mid S_t = s]$$

~~average(~~ Returns[0], Returns[1], Returns[2]) ← a or b → behavior policy

↓ action

We have to correct each return.

$$V_\pi(s) \doteq E_\pi[G_t \mid S_t = s]$$

↝ average(ρ_0 Returns[0], ρ_1 Returns[1], ρ_2 Returns[2])

$$V_\pi(s) \doteq E_b[\rho G_t \mid S_t = s]$$

$$\rho = \frac{P(\text{trajectory under } \pi)}{P(\text{trajectory under } b)}$$

Off-policy Trajectories

$$P(A_t, S_{t+1}, A_{t+1}, \ldots, S_T \mid S_t, A_{t:T})$$

⇒ given that agent is in some state S_t what is the prob. that it takes action A_t and ends up in S_{t+1} state then it takes action A_{t+1} and ends up in S_{t+2} and so on until termination at time T.

- Actions are sampled according to behavior b.

Because of Markov property, we can break this distribution into smaller chunks.

$$\underbrace{b(A_t \mid S_t)\, p(S_{t+1} \mid S_t, A_t)}\, b(A_{t+1} \mid S_{t+1})\, p(S_{t+2} \mid S_{t+1}, A_{t+1}) \ldots p(S_T \mid S_{T-1}, A_{T-1})$$

⇒ prob that agent selects action A_t in state S_t time the prob. that the environment transitions into environment S_{t+1}.

can be re-written as product

$$P(\text{trajectory under } b) \Rightarrow \prod_{k=t}^{T-1} b(A_k \mid S_k)\, p(S_{k+1} \mid S_k, A_k)$$

$$\Rightarrow \rho_{t:T-1} \doteq \frac{P(\text{traj. under } \pi)}{P(\text{traj. under } b)}$$

$$\doteq \prod_{k=t}^{T-1} \frac{\pi(A_k \mid S_k)\, p(S_{k+1} \mid S_k, A_k)}{b(A_k \mid S_k)\, p(S_{k+1} \mid S_k, A_k)}$$

$$\rho_{t:T-1} \doteq \prod_{k=t}^{T-1} \frac{\pi(A_k \mid S_k)}{b(A_k \mid S_k)}$$

Off-Policy Value

$$E_b[\rho_{t:T-1} G_t \mid S_t = s] \doteq v_\pi(s)$$

Off-policy every-visit MC prediction, for estimating $V \approx v_\pi$

Input: a policy π to be evaluated
Initialize:
  $V(s) \in \mathbb{R}$, arbitrarily, for all $s \in S$
  $Returns(s) \leftarrow$ an empty list, for all $s \in S$
Loop forever (for each episode):
  Generate an episode following b: $S_0, A_0, R_1, S_1 \ldots, S_{T-1}, A_{T-1}, R_T$
  $G \leftarrow 0 \quad W \leftarrow 1$
  Loop for each step of episode, $t = T-1, T-2, \ldots, 0$
    $G \leftarrow \gamma W G + R_{t+1}$
    Append G **to** $Returns(S_t)$
    $V(S_t) \leftarrow$ **average**$(Returns(S_t))$
    $W \leftarrow W \frac{\pi(A_t \mid S_t)}{b(A_t \mid S_t)}$

→ behavior policy b

→ corrected by W

} accumulated product of importance sampling ratio on each time step of episode.

Computing $\rho_{t:T-1}$ Incrementally

$$\rho_{t:T-1} \doteq \prod_{k=t}^{T-1} \frac{\pi(A_k \mid S_k)}{b(A_k \mid S_k)}$$

$$= \rho_t \rho_{t+1} \rho_{t+2} \cdots \rho_{T-2} \rho_{T-1}$$

← MC algo loops over time steps backwards.

$W_1 \leftarrow \rho_{T-1}$
$W_2 \leftarrow \rho_{T-1} \rho_{T-2}$
$W_3 \leftarrow \rho_{T-1} \rho_{T-2} \rho_{T-1}$

- Each time step adds one additional term

$W_{t+1} \leftarrow W_t \rho_t$ ⇒ compute recursively without having to store all past values of ρ.

Batch Reinforcement Learning

↳ counterfactual RL

- Learn a good policy that will perform well in the future

$$\Rightarrow \text{argmax} \int_{s \in S_0} \hat{V}^\pi(s, D)\, ds$$

Covariance Shift Challenge ⇒ we have different policies, those are generally going to be taking different actions and those different actions and those different actions are gonna lead to different state distributions.

- Importance sampling ⇒ allows to use data generated by a different policy and re-weigh it so that we get correct expectation.

$$V^{\pi}(s) = \sum_{\tau} p(\tau|\pi, s)\, R(\tau)$$

$$= \sum_{\tau} p(\tau|\pi_b, s)\, \frac{p(\tau|\pi, s)}{p(\tau|\pi_b, s)}\, R_{\tau}$$

$$\approx \sum_{i=1, \tau_i \sim \pi_b}^{N} \frac{p(\tau_i|\pi, s)}{p(\tau_i|\pi_b, s)}\, R_{\tau} \quad \rightarrow \text{dynamics model of the world}$$

$$= \sum_{i=1, \tau_i \sim \pi_b}^{N} R_{\tau_i} \prod_{t=1}^{H_i} \frac{p(s_{i,t+1}|s_{it}, a_{it})\, p(a_{it}|\pi, s_{it})}{p(s_{i,t+1}|s_{it}, a_{it})\, p(a_{it}|\pi_b, s_{it})}$$

$$= \sum_{i=1, \tau_i \sim \pi_b}^{N} R_{\tau_i} \prod_{t=1}^{H_i} \frac{p(a_{it}|\pi, s_{it})}{p(a_{it}|\pi_b, s_{it})}$$

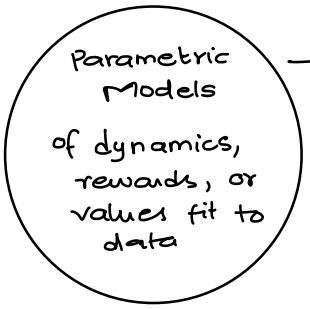
↓ doesn't require us to know the dynamics model of the world.

D: Dataset of n traj.s τ, $\tau \sim \pi_b$
π: Policy mapping $s \rightarrow a$
S_0: set of initial states
$\hat{V}^{\pi}(s, D)$: Estimate $V(s)$ w/ dataset D

Challenges of importance sampling for policy eval.

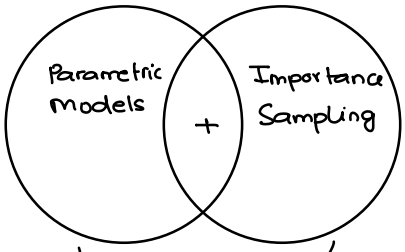
- IS provides us with an unbiased estimator but it can be very high variance.
  → sometimes variance can scale exponentially with the horizon.
- If we have very little data or if horizon is very long, then the estimators would be poor approximators of how well policy might do in future.

Alternative ⇒ use parametric model



→ + Low variance
− bias (unless realizable)

Doubly Robust



+ Smaller variance than importance sampling
+ Unbiased if either model realizable or behavior policy known

⇒ combine model based estimators with importance sampling in order to get lower variance & lower bias.

Week #3

Temporal Difference Learning

→ In prediction problem, we learn a value function that estimates the returns starting from a given a state.

$$v_\pi(s) \doteq E_\pi[G_t \mid S_t = s]$$ → to compute returns, we have to take samples of full trajectories.

$$V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)]$$ → small modification

↳ update rule

- can form MC estimate without storing lists of returns
- Not learning during the episode

Recall: Bootstrapping (discounted return)

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \cdots$$

$$G_t \doteq R_{t+1} + \gamma G_{t+1}$$

$$v_\pi(s) = E_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s]$$

$$\Rightarrow v_\pi(s) = R_{t+1} + \gamma v_\pi(S_{t+1})$$

Temporal Difference

- We want to update towards the return but we don't want to wait for it.

$$V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)]$$. TD Error

$$V(S_t) \leftarrow V(S_t) + \alpha[\underbrace{R_{t+1} + \gamma V(S_{t+1})}_{\text{t-target}} - V(S_t)]$$

$$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$$

↳ TD Error

1-step TD

$S_t \leftarrow S_{t+1}$ (store state from previous timestep to make TD update)

$S, A, R, S, A, R \ldots$

- From the state at time t, we can take an action and observe the next state at time $t+1$.
- Only then we can update value of previous state.

$S, A, R, S, A, R \ldots$

Tabular TD(0) Algorithm

Tabular TD(0) for estimating v_π

Input: the policy π to be evaluated
Algorithm parameter: step size $\alpha \in (0, 1]$
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$ → initial estimate of value-fln

Loop for each episode:
  Initialize S → every episode begins in some initial state S
  Loop for each step of episode:
    $A \leftarrow$ action given by π for S
    Take action A, observe R, S'
    $V(S) \leftarrow V(S) + \alpha[R + \gamma V(S') - V(S)]$ → on each step
    $S \leftarrow S'$ → keep track of previous state
  until S is terminal

- TD → specialized for prediction learning
- Methods that scale with computation are future of AI.

⇒ weak general-purpose methods are better than strong methods (utilizing human insight)

↳ Supervised learning and model-free RL methods are only weakly scalable.

Prediction Learning → try to predict what will happen next, make a prediction, wait and see what happens and when you find out what happens, you learn.

↳ unsupervised supervised learning
- We have a target (just by waiting)
- Yet no human labelling is needed
- scalable model-free learning

The one-step Trap

- Thinking that one-step predictions are sufficient
- At each step predict the state and observations one step later.
- Long term prediction can then be made by simulation.

In theory this works, but not in practice.
- → Making long-term predictions by simulation is exponentially complex
- → amplifies even small errors in one-step predictions.

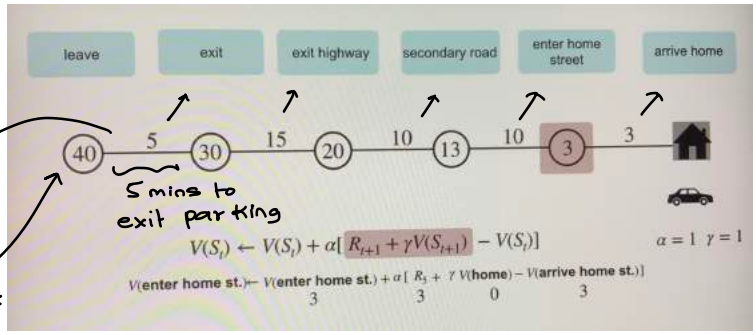
TD → bootstraps like DP and learns from experience like MC methods

Example

$V(leave) \leftarrow V(leave) + \alpha[R_1 + \gamma V(exit) - V(leave)]$

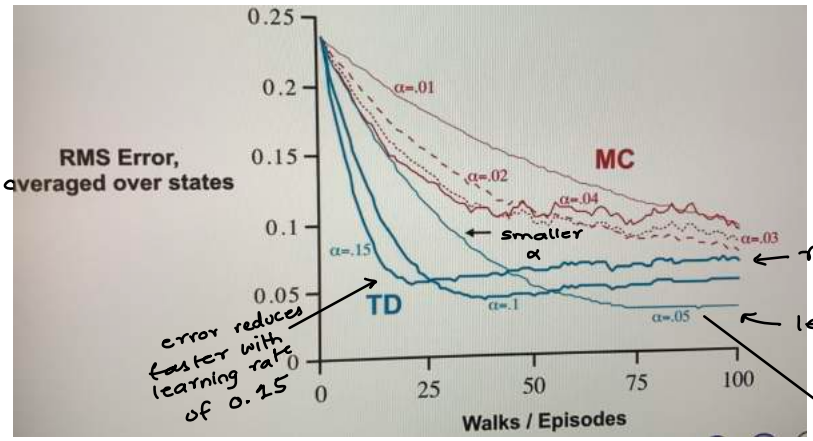
$40 \leftarrow 30 + [5 + 35 - 30]$

update estimate



TD → requires no model of environment, can update on every step

Comparing TD and Monte Carlo



← results in final higher error

← learns slowly but final error is the smallest.

converges faster to the lower final error (TD)

Week #4

Generalized Policy Iteration with TD: Sarsa

- Use TD in policy evaluation in GPI.

GPI

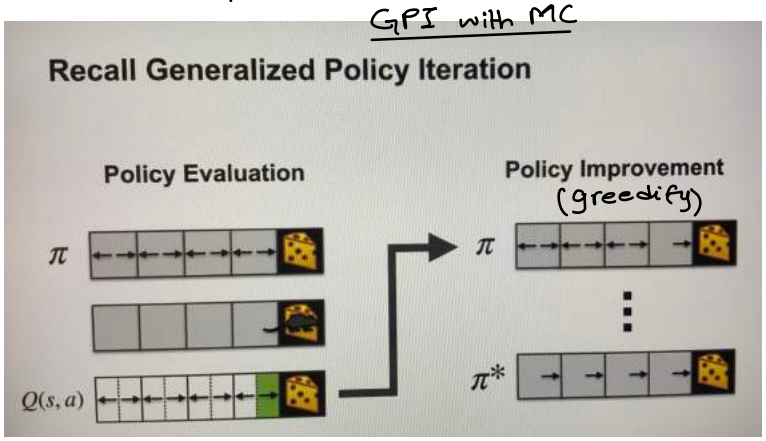
policy evaluation ↘ ↙ policy improvement

SARSA → A_{t+1}

S_t A_t R_{t+1} S_{t+1}

- Take an action A_t in initial state S_t and get a reward R_{t+1} and go to next state S_{t+1}.
- However, before update, SARSA requires that next action in the new state is also committed to.

GPI with MC

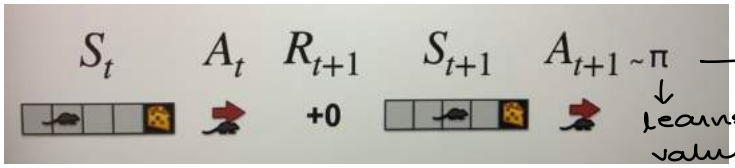


- GPI with Monte Carlo (MC) does not perform a full policy evaluation step before improvement.
- Rather it evaluates & improves after each episode.

SARSA Algorithm

state-action values

$\Rightarrow Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left(R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\right)$



→ policy eval.
↓
learns action values for a specific fixed policy

→ However, with GPI, this can turn into a control algorithm.

$S_t \; A_t \; R_{t+1} \; S_{t+1} \; A_{t+1} \sim$ e-greedy

$\Rightarrow$ improve policy every time step, rather than after an episode or after convergence.

Off-Policy TD Control: Q-Learning

$\Rightarrow$ What is Q-learning?

↓ update rule

$$Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \underbrace{\max_a Q(S',a)}_{\text{different from SARSA}} - Q(S,A)\right]$$

SARSA: $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\right]$

$$q_\pi(s,a) = \sum_{s',r} p(s',r \mid s,a)\left(r + \gamma \sum \pi(a' \mid s')\, q_\pi(s',a')\right)$$

↓ Bellman state-action value equation

Q-learning: $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left[R_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a') - Q(S_t, A_t)\right]$

$$q^*_\pi(s,a) = \sum_{s',r} p(s',r \mid s,a)\left(r + \gamma \max_{a'} q_\pi(s',a')\right)$$

↓ Bellman-optimality equation

- Doesn't require switching b/w policy evaluation & control

SARSA ~ policy iteration

Q-learning ~ value iteration
↳ converges as long as the agent continues to explore and samples all areas of the state-action space.

- Q-learning takes the max over the next action values, so only changes when the agent learns that one action is better than the other.

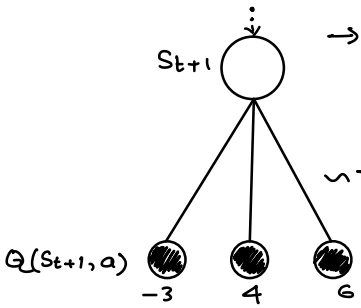
- SARSA uses the estimate of next action value in its target. This changes every time the agent takes exploratory action.

How is Q-learning off-policy?

Q-learning → doesnot use importance sampling.

↳ because agent estimating action values with a known policy.

↳ It doesnot need importance sampling ratios to correct for difference in action selection.

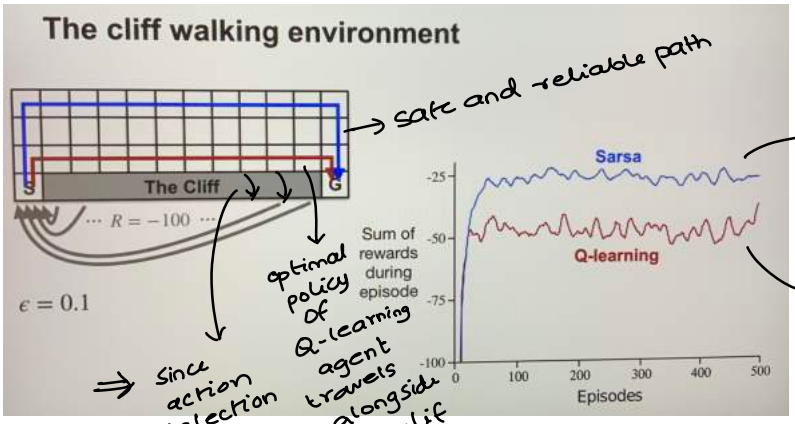


→ the agent's target policy is the prob. of taking each action in the given state.

→ the agent can calculate the expected return under its target policy from any given states, in particular the next state S_{t+1}.

↳ $\pi(a', S_{t+1})$

$$\sum_{a'} \pi(a'|S_{t+1}) Q(S_{t+1}, a') = E_\pi[G_{t+1}|S_{t+1}] = \max_{a'} Q(S_{t+1}, a') = 6$$



→ safe and reliable path

→ longer path but reliable
• wrong action doesn't throw off the cliff.

→ shorter and more direct path, but occassionally falls off the cliff.

optimal policy of Q-learning agent travels alongside the cliff

⇒ since action selection is ε-greedy, sometimes takes the wrong action & falls off the grid.

Expected SARSA → more robust to large step sizes, can quickly learn (in some cases) than SARSA

- Explicitly computing the expectation over next actions is the main idea behind expected SARSA.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left(R_{t+1} + \gamma \sum_{a'} \pi(a'|S_{t+1}) Q(S_{t+1}, a') - Q(S_t, A_t)\right)$$

- Expected SARSA has more stable update target than SARSA and have lower variance.

→ Computing the average over next actions become more expensive as the number of actions increases

→ When there are many actions, computing the average might take a long-long time, especially since average has to be computed every episode.

Generality of Expected SARSA → off-policy

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left(R_{t+1} + \gamma \sum_{a'} \pi(a'|S_{t+1}) Q(S_{t+1}, a') - Q(S_t, A_t)\right)$$

$$\sum_{a'} \pi(a'|S_{t+1}) Q(S_{t+1}, a') = \max_{a'} Q(S_{t+1}, a')$$

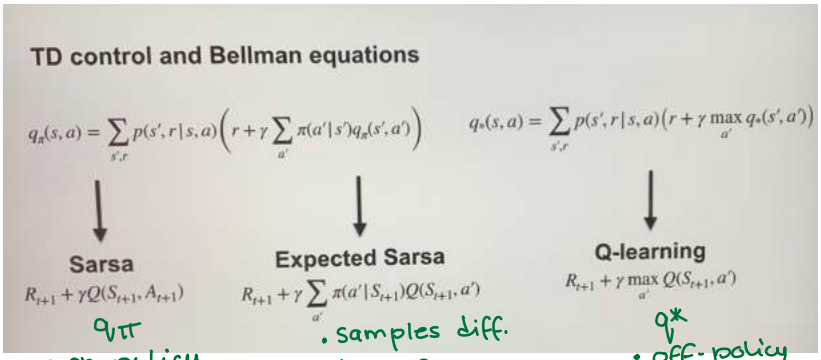
greedy

$A_{t+1} \sim \pi(a'|S_{t+1}) \neq b(a'|S_{t+1})$

→ expectation over actions is computed independently of the action actually selected in the next state.

- Q-learning is a special case of expected SARSA.

- Expected SARSA where target policy is greedy with respect to its action-values is exactly Q-learning.



q_π
- on-policy
- learns action-values for policy it is currently following

- samples diff. than SARSA
- takes expectation
- both an off-policy & on-policy algorithm that learns action values for any policy.

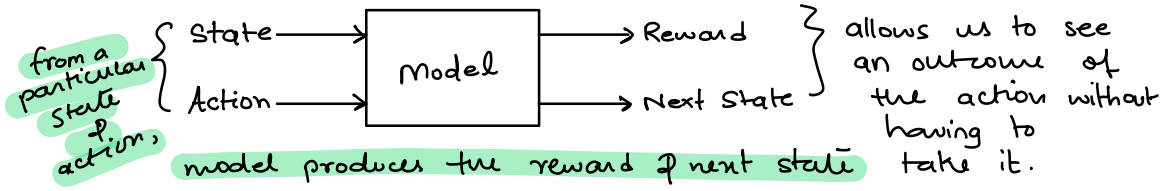
q_*
- off-policy
- learns optimal action-values

↳ On-policy control methods account for their own exploration.

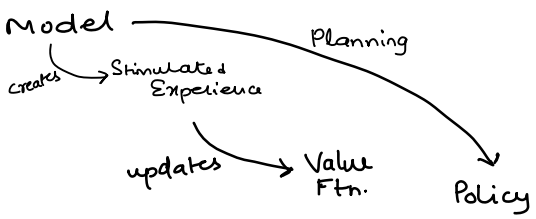
Week # 5

TD → learns only from sampled experience

Models → store the knowledge about the dynamics



Planning → the process of using a model to improve a policy.



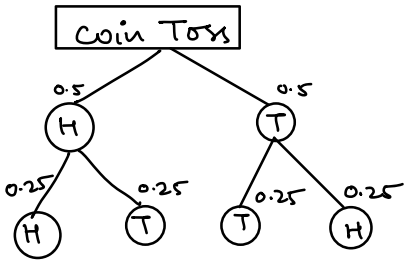
- Simulating experience improves the sample efficiency.
- The addition of simulated experience, means the agent needs fewer interactions with the world to come up with the same policy.

Sample Model → computationally inexp.

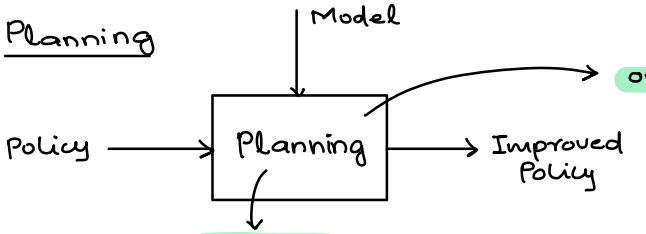
- Produces an actual outcome drawn from some underlying probabilities.
- Procedurally generate samples, without explicitly storing the probability of each outcome.

Distribution Model

- Completely specifies the likelihood or probability of every outcome.
- Contain more information.
- Difficult to specify.
- Enumerates every sequence and assigns a probability to each seq.
- can be used to compute the exact expected outcome.



Planning



→ takes a model as input and outputs an improved policy.

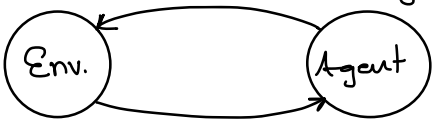
one possible approach to planning is to first sample experience from the model.

- use this generated experience to perform updates to the value function as if these interactions actually occurred in the world.

Random-sample One-Step Tabular Q-learning

1. Sampling (State, action pair) $S \to S$, $A \to A$ → Model → S', R } Sample model of trans. dynamics available
2. Q-Learning Update $Q(S,A) \leftarrow Q(S,A) + \alpha \left[R + \gamma \max_a Q(S',a) - Q(S,A) \right]$
3. Greedy Improv. $\pi(s) = \operatorname{argmax}_a Q(s,a)$ ↳ updated action values

- Planning only uses simulated or imagined experience.

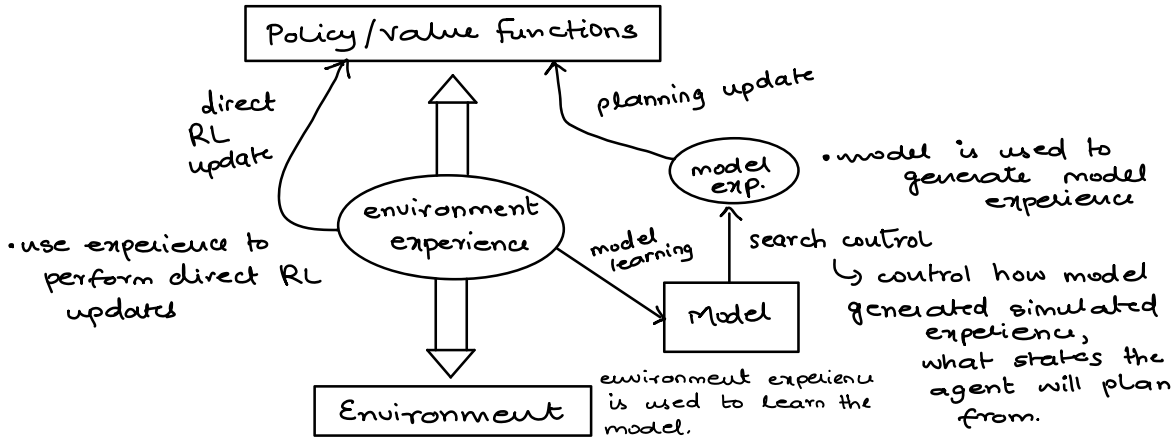


- All updates can be done without behaving in real world or parallelly w/ interaction

- Q-learning → performs updates using environment experience
- Q-planning → performs updates using simulated experience from the model.

Dyna Architecture

↳ combines direct RL update and planning updates.



Tabular DynaQ (Model based)

Tabular Dyna-Q

Initialize $Q(s,a)$ and $Model(s,a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$

Loop forever:

(a) $S \leftarrow$ current (nonterminal) state

(b) $A \leftarrow \varepsilon$-greedy(S,Q) → policy

(c) Take action A; observe resultant reward, R, and state, S'

(d) $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$

(e) $Model(S,A) \leftarrow R, S'$ (assuming deterministic environment)

(f) Loop repeat n times:

planning steps:

$S \leftarrow$ random previously observed state

$A \leftarrow$ random action previously taken in S

$R, S' \leftarrow Model(S,A)$

$Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$

(c)–(d): Q-learning

(e): → model learning step
• memorizes next state & reward for state action pair.

→ repeated several times

(f): search control ↳ (sec. 8.4)

S, A: model query

value update

- DynaQ performs many planning updates for each environment transition.

Inaccurate Models

- Models are inaccurate if the transitions they store are different from the transitions that happen in the environment.

Incomplete Models → sample the visited state-action pairs to plan

- At the start of learning, the agent has not tried most of the actions in almost all the states.
- The transitions associated with trying those actions in those states are missing from the model → such model is incomplete.

Changing Environment

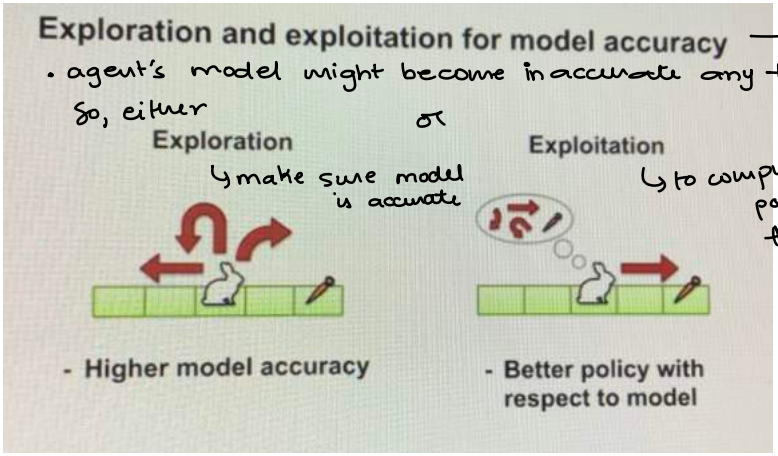
- Taking an action in a state could result in a different next state and reward than what the agent observed before the change.

Model inaccurate → what happens different from what the model says.

Planning with Inaccurate Models

- In case of incomplete model, as long as the model stores some transitions as the agent interacts, the model can be used for planning.

→ In case of changing environment, planning with wrong model could change value function or policy incorrectly.
→ Planning in such a case will make policy worse w.r.t the environment.



→ When the environment changes, the model is incorrect and remains so until the agent revisits the part of the environment that changed and updates the model.

Solution → encourage agent to re-visit the states periodically. Add a bonus to the reward used in planning.

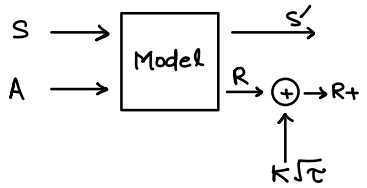
New reward = $r + K\sqrt{\tau}$

- r: actual reward
- K: small constant that controls the influence of bonus
- τ: the time since the state-action pair was last tried.
  - not updated in planning loop since that is not a real visit.

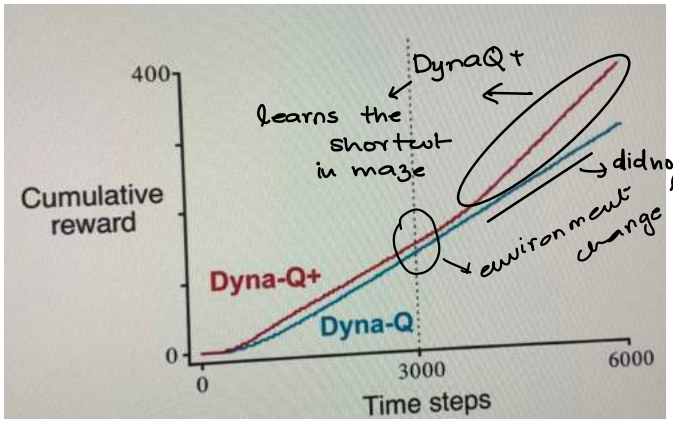
$K = 0$, ignore the bonus completely.

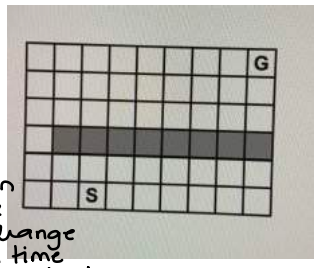
DynaQ+

- Add the bonus to the DynaQ's planning updates

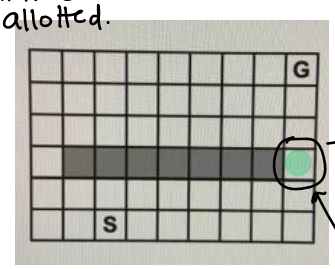


- If state-action pair not visited in a long time, the τ (tau) will be large. As tau grows, the bonus becomes bigger and bigger.
- Eventually, planning will change the policy to go directly to S due to large bonus, and model will be updated.





Base environment



changed environment

shortcut introduced