2 | DYNAMIC PROGRAMMING (DP)

2.1 Introduction

– DP and its Requirements

- DP is a method for solving complex problems by:
  ↳ breaking them into sub problems.
    ↳ solve sub prob → combine solution to sub prob.

- Requirements for DP are:
  → Optimal sub-structure : an opt solution can be decomposed into opt solution of sub-problems.
  → Overlapping subproblems : subproblems recur many times; solutions can be cached and reused.
  → MDPs satisfy both properties
    ↳ Bellman equation gives recursive decomposition.
    ↳ value function stores & reuses solution.

– Planning by DP

In planning someone tells us the structure of the MDP (the dynamics and rewards) and given perfect knowledge of how the environment works we want to solve the MDP.

Two special cases of planning can be solved in MDP :

- Plan to solve a prediction problem
  → Input : MDP (S, A, P, R, γ) and policy π
  ↳ Output : value-function v_π

  } Someone tells us the MDP and also gives a policy and asks you, how much reward would one get walking around in this world.

- Plan is used for control
  → Input : MDP (S, A, P, R, γ)
  ↳ Output : optimal value function v_*
  : optimal policy π_*

  } someone tells us the MDP, but now instead of giving a policy they want to know what's the best policy; amongst all policies what's the most reward that can be achieved from this MDP

2.2 Policy Evaluation

- Iterative Policy Evaluation

- Problem : evaluate a given policy π
- Solution : take bellman eq and turn into iterative update

↳ v_1 v_2 - - - v_π

We begin with initializing the value space randomly. At v_2 we would have plugged in our 1 step look ahead bellman eq., we iterate this many time and we eventually end up w/ true val function.

↳ The way this is done is using syncronous back-ups ie.

↳ @ each iteration we go a sweep across all states in MDP

@ each iteration $k+1$

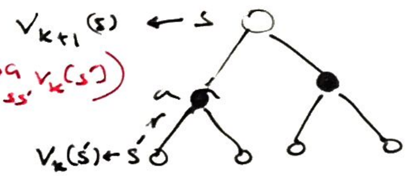
$\forall$ states $s \in S$

Update $v_{k+1}(s)$ from $v_k(s')$

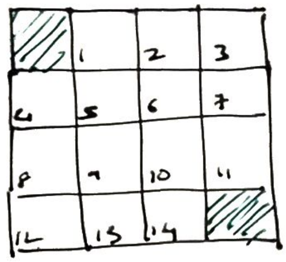
where s' is a successor state of s

→ $$v_{k+1}(s) = \sum_{a \in A} \pi(a|s)\left(R_s^a + \gamma \sum_{s' \in S} P^a_{ss'} v_k(s')\right)$$

$$v^{k+1} = R^\pi + \gamma P^\pi v^k$$



- Grid-world example



→ Undiscounted episodic MDP ($\gamma = 1$)

→ Non-terminal states 1... 14

→ One terminal state

→ Actions out of grid leave state unchanged

→ Reward = -1 until terminal state reached

→ Agent follows uniform random policy $\pi(n|\cdot) = \pi(e|\cdot) = \pi(s|\cdot) = \pi(w|\cdot) = 0.25$

$k = 0$

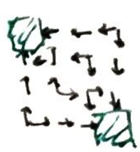
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

$k = 2$

0	-1.7	-2	-2
-1.7	-2	-2	-2
-2	-2	-2	-1.7
-2	-2	-1.7	0

~~Iten~~ $k = 10$

0	-6	-8	-9
-6	-7	-8	-8
-8	-8	-7	-6
-8	-9	-6	0



We notice with the gradual inc in iterations the policy gets better until optimality is reached.

2.3 Policy Iteration

For the previous setting we evaluated a given policy and here we'd want to find the best possible policy.

- Making Policy better
  - given policy π
    → evaluate policy π : $v_\pi(s) = E[R_{t+1} + \gamma R_{t+2} + \dots \mid S_t = s]$
    → Improve policy by acting greedily w.r.t v_π

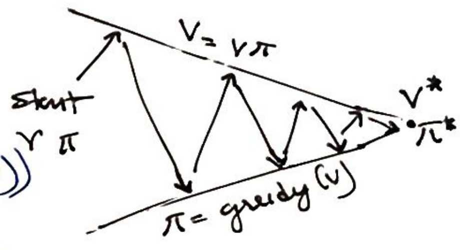
$$\pi' = \text{greedy}(v_\pi)$$

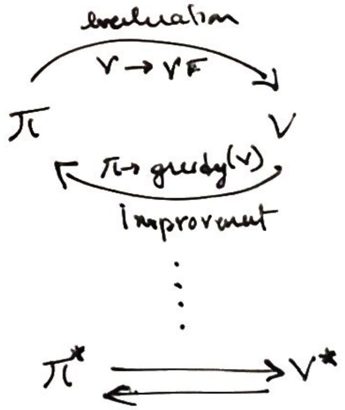
To explain further, we begin at state s, cal val function of states around & onto which an action can be effectuated and then greedily move in the direction w/ highest value. Again, evaluate the stat you land in and move again greedily.

→ In the small grid world problem this was enough but not in real world prob where you'll need more iterations of improvement / evaluation.

→ This process always converges to an optimal policy π^*

- Illustration
  - We start w/ some i/ps (random vals (0's)) and some policy
    ↳ policy eval ↗ and policy improve ↘
      ↳ start w/ policy and eval it (↗) then (↘) act greedily w.r.t to val function to give us the new policy ....





- Check rental car policy example in slides.

- Policy Improvement

- Consider a deterministic policy, $a = \pi(s)$: for example acting greedy always makes my policy deterministic.
- We can improve policy by acting greedily $\pi'(s) = \underset{a \in A}{\operatorname{argmax}}\, q_\pi(s,a)$
- If we act greedily ; the greedy policy at least improves the values we gna get over 1 step (immediate 1 step)

$$q_\pi(s, \pi'(s)) = \max_{a \in A} q_\pi(s,a) \geq q_\pi(s, \pi(s)) = v_\pi(s)$$

→ This is indirectly saying that if we pick this greedy policy the value of greedy policy is at least as much as the policy before we "greedified" it. So we don't make things worse ever, at least make it better.

- If improvements stop, we'd want to know if we achieved optimal

$$q_\pi(s, \pi'(s)) = \max_{a \in A} q_\pi(s,a) = q_\pi(s, \pi(s)) = v_\pi(s)$$

→ this is basically the bellman equation, ie if this is satisfied we have satisfied the bellman optimality equation $v_\pi(s) = \max_{a \in A} q_\pi(s,a)$

→ And if bellman opt eq satisfied then π must be the optimal policy $\therefore v_\pi(s) = v_*(s) \ \forall s \in S$

- Extensions to policy iterations

- Modified Policy Iteration

→ Does the policy eval need to converge to v_π?

→ Or should we introduce a stopping condition : when the state-values are only changed by a tiny amt after each iteration then one can just stop. (small grid world $k=3$ was sufficient for π_*)

→ Or simply stop after k iterations of iterative policy evaluation

→ Why not update policy every iteration? ie stop after k=1, calc values and ~~proceed~~ act greedy w.r.t to that value function and then proceed: → this is equivalent to value Iteration

2.4 Value Iteration

- Principle of Optimality (Recap)
  - Any opt policy can be subdivided into 2 comps:
    - → An optimal 1^{st} action A_*
    - → followed by following an opt policy from successor state
  - A policy $\pi(a|s)$ achieves the opt value from state s,
    $v_\pi(s) = v_*(s)$ iff → for any state s' reachable from s
    ↳ π achieves the opt val from state s',
    $v_\pi(s') = v_*(s')$

- Deterministic Value Iteration
  - If we know the solution to subproblems $v_*(s')$ (The opt values from s'). Use this info to build opt value func for prev step
  - All we need to do is; one step look ahead using bellman optimality eq $v_*(s) \leftarrow \max_{a \in A} R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_*(s')$
  - The idea of value iteration is to apply updates iteratively; ie instead of someone giving $v_*(s)$, we first initialize it randomly and then iteratively arrive at opt sol.
    ↳ start w/ final reward & work backwards.

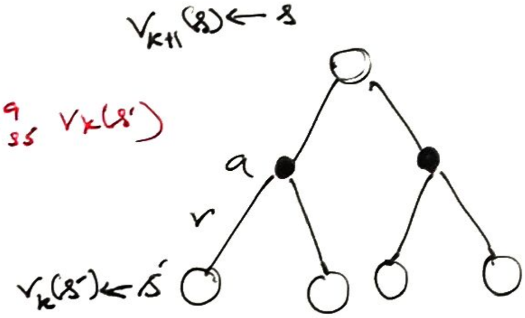
- Value Iteration
  - Problem : find optimal π
  - Solution : Iterative application of Bellman Optimality Backup ($v_1, v_2 --- v_*$)
  - Using synchronous backups
    → @ each iteration $k+1$ → for all states $s \in S$ → Update $v_{k+1}(s)$ from $v_k(s')$

- Unlike policy iteration, there's no explicit policy.
  - ↳ w/ policy iteration @ every step we constructed a val-function that was the val-function for a particular policy whereas w/ value iteration a given V_i may not correspond to any real policy.

-

$$V_{k+1}(s) = \max_{a \in A} \left(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a V_k(s') \right)$$

$$V_{k+1} = \max_{a \in A} R^a + \gamma P^a V_k$$



- Synchronous Dynamic Programming Algos

Problem	Bellman Equation	Algo
Prediction	Bellman Expectation Eq	Iterative Policy evaluation
Control	BEE + greedy policy improvement	Policy Iteration
Control	Bellman optimality Eq	Value Iteration.

- All problem are planning problems, in all cases the MDP is given and we're just trying to solve for it.
  - ↳ 2 types of planning problem: Prediction and Control
  - ~~(1) Prediction~~ :

- In case 1. we're trying to solve how much reward are you gna get from a particular policy; To figure that out we use the bellman expectation equation, which is taken and turned into an iterative update to arrive @ an Iterative policy eval algo.

- Talking about control, which shows how to get the most out of an MDP, how to max total reward, how find v_* and hence the optimal policy.

- In case 2, the 1st control approach was to again use the bellman exp eq and iterate over it to evaluate the policy, but then to alternate that process of evaluation w/ the process of policy improvement which gave us the policy iteration algo
- In case 3, we turn the B.O.eq to an iterative update to get the value iteration algo.
- All these algos are based on state-val func $v_\pi(s)$ or $v_*(s)$
- Complexity if we have n actions & m states : $O(mn^2)$ per iteration. Not too great for bigger probs
- Could also apply to action-val fun $q_\pi(s,a)$ or $q_*(s,a)$
  ↳ complexity $O(m^2n^2)$

Backups just mean getting v(s) by one-step look ahead.

- Asynchronous D.P

- DP methods so far use synchronous backups & ie all states are backed in parallel.
- Async DP back ups states individually in any order. For each selected state apply appropriate backup. Reduces computation and guarantee to converge if all states continue to be selected
- 3 ideas for Async DP
  → In place DP : stores only 1 copy of val-function

$$v(s) \leftarrow \max_{a \in A} \left(R^a_s + \gamma \sum_{s' \in S} P^a_{ss'} v(s') \right)$$

  → Prioritised sweeping : use Bellman error to guide state selection

$$\left| \max_{a \in A} \left(R^a_s + \gamma \sum_{s' \in S} P^a_{ss'} v(s') \right) - v_k(s) \right|$$

@ iteration k+1

↳ Back up state w/ largest bellman error ↳ update bellman error of affected states after each backup. Can be implemented using priority queue

  → Real time DP $v(S_t) \leftarrow \max_{a \in A} \left(R^a_{S_t} + \gamma \sum_{s' \in S} P^a_{S_t s'} v(s') \right)$