Ch-4, Dynamic Programming

→ Dynamic Programming (DP) – collection of algs that can be used to compute optimal policies given a perfect model of the environment as a MDP

→ Assume - env is a finite MDP - assume that its state, action and reward sets S, A and R are finite and that its dynamics are given by set of prob $p(s', r|s, a)$ for all $s \in S$, $a \in A(s)$, $r \in R$ and $s' \in S^+$ (S^+ is S plus a terminal state if prob is episodic)

→ Although DP ideas can be applied to prob with continuous state and action spaces, exact Sol^ are only possible in special cases.
A common way – obtain approx sol^ for tasks with continuous states and actions is to quantize the state and actions and then apply finite-state DP methods.

→ Key idea of DP – use of value functions to organize and structure the search for good policies

We can easily obtain optimal policy once we have found the optimal value functions v_* or q_* which satisfy Bellman Optimality eq^:

$$v_*(s) = \max_a \mathbb{E}\left\{ R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a \right\}$$

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

$$q_*(s, a) = \mathbb{E}\left\{ R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \mid S_t = s, A_t = a \right\}$$

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

DP algs are derived from the Bellman eq by turning them into update rules for iteratively improving approximations of the value functions

Policy Evaluation (Prediction)

→ Policy evaluation is the process of computing the state value function v^{π} for a given policy π
This is also known as prediction problem

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

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

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

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

where $\pi(a|s)$ - prob to take action a in state s under policy π
$p(s', r|s, a)$ - prob of transitioning to state s' and receiving reward r after taking action a in state s
γ - discount factor

- The existence and uniqueness of v_π are guaranteed as long as either $\gamma < 1$ or eventual termination is guaranteed from all states under policy π

→ To compute v_π we can use an iterative approach known as iterative policy evaluation. Starting with an initial approximation v_0 (which can be arbitrary except for terminal state, if any, which must be given a value of 0), use

repeatedly apply the following update rule to each state s:

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

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

This process is repeated until the value function converges to a fixed pt, which is guaranteed under certain condition (eg if $\gamma < 1$ or if eventual termination is guaranteed from all states under policy π)

→ To produce each successive approx, v_{k+1} from v_k, iterative policy evaluation applies the same operation to each state s: it replaces old value of successor states of s with a new a new value obtained from the old values of the successor states of s, and expected immediate rewards along all the one-step transitions possible under the policy being evaluated — Expected update.

In this eq^n, the inner summation computes the expected return for each action a, considering all next possible states s' and rewards r. The outer summation then averages these expected returns according to policy π which specifies the prob of taking each action a in state s.

This process is applied to each state s in state space, and the result is a new approximate v_{k+1} of the value function. By repeating this process repeatedly, value function converges to true value function v^π for given policy π.

The term "expected update" emphasis that this process is

based on expectation over all possible next states rather than on a single sampled next state — key characteristic of DP algo

→ Iterative Policy Evaluation : for estimating $V \approx V_\pi$

Input π, the policy to be evaluated
Algo parameter : a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in S^+$, arbitrarily except $V(\text{terminal}) = 0$

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

Policy Improvement :-

→ Reason for computing the value function for a policy — help find better policies.

→ value function of a policy (V_π) : This is the expected return when starting in state s and following policy π thereafter

→ $q_\pi(s,a) \doteq \mathbb{E}[R_{t+1} + \gamma V_\pi(S_{t+1}) \mid S_t = s, A_t = a]$

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

Policy Improvement – Computation of an improved policy given the value function for that policy

PAGE EDGE

Date : / /

This is the expected return when starting in state s, and following policy π thereafter.

→ Policy improvement : Suppose we have a deterministic policy π and we want to know if we should change the policy to always choose an action a in state s. We compare $q_\pi(s,a)$ with $v_\pi(s)$:

- If $q_\pi(s,a) > v_\pi(s)$, it means that taking action a in state s and then following π from s. Here we should change the policy to always choose action a in state s.

→ Policy Improvement Theorem :

If for all state s, $q_\pi(s, \pi'(s)) \geq v_\pi(s)$ where π' - new policy then π' is as good or better than π. That is, $v_{\pi'}(s) \geq v_\pi(s)$ for all s.
If there is a strict inequality for any state, then π' is strictly better than π.

If $q_\pi(s,a) > v_\pi(s)$ then the changed policy is indeed better than π.

→ Greedy Policy Improvement :

To improve a policy, we can construct a new policy π' that is greedy wrt value function of the original policy π. For each state s, π' chooses the action that maximizes $q_\pi(s,a)$. This new policy π' is better or as good as the original policy π.

$$\pi'(s) \doteq \arg\max_a q_\pi(s,a)$$

$$\pi'(s) = \underset{a}{\operatorname{argmax}}\, E\left[R_{t+1} + \gamma V_\pi(S_{t+1}) \mid S_t = s, A_t = a\right]$$

$$= \underset{a}{\operatorname{argmax}} \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma V_\pi(s')\right]$$

where argmax - value of a at which expression that follows is maximized

Greedy policy takes the action that looks best in short term. By constructing the greedy policy meeting the conditions of the policy improvement theorem, so we know that it is as good as, or better than, the original policy. The process of making a new policy that improves on an original policy, by making it greedy wrt value function of the original policy is called policy improvement.

Note: Policy improvement thus must give us a strictly better policy except when the original policy is already optimal.

→ Convergence to Optimal Policy: If the greedy policy π' is as good as, but not better than, the old policy π, then both π and π' must be optimal policies. If not, the process of policy improvement with π' can be repeated to get even better policy and the process will eventually converge to policy which is optimal.

Summary: Policy improvement is a process of using the value function of a policy to construct a better policy. By repeatedly applying policy improvement, we can iteratively improve policies until we reach an optimal policy.

PAGE EDGE
Date : / /

Policy Iteration :-

→ Once a policy π has been improved using v_π to yield a better policy π' we can then compute $v_{\pi'}$ and improve it again to yield an even better π''.

$$\pi_0 \xrightarrow{E} v_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} v_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \cdots \xrightarrow{I} \pi_* \xrightarrow{E} v_*$$

Where $\xrightarrow{E}$ denotes policy evaluation
$\xrightarrow{I}$ denotes policy improvement

- Each policy is guaranteed to be a strict improvement over the previous one (unless it is already optimal).
- Because a finite MDP has only a finite no of policies, this process must converge to an optimal policy and optimal value function in a finite number of iterations.

This way of finding optimal policy — Policy Iteration

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

1) Initialization :

$V(s) \in \mathbb{R}$ and $\pi(s) \in A(s)$ arbitrarily for all $s \in S$

2) Policy Evaluation :

Loop:
$\Delta \leftarrow 0$
Loop for each $s \in S$:

$v \leftarrow V(s)$

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

$\Delta \leftarrow \max(\Delta, |v - V(s)|)$

until $\Delta < \theta$ (a small +ve no. determining the accuracy of estimation)

3) Policy Improvement:

policy-stable ← true

For each $s \in S$:

old-action ← $\pi(s)$

$\pi(s) \leftarrow \text{argmax}_a \sum_{s',r} p(s', r | s, a) [r + V(s')]$

If old action $\neq \pi(s)$, then policy-stable ← false

If policy-stable, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2

Note: Starting each policy evaluation with the value function for the previous policy typically results in faster convergence, as the value function often changes little from one policy to the next.

- Policy Improvement theorem assures us that these policies are better than the original random policy.

Value Iteration :-

→ Drawback of policy iteration: Each of its iteration involves policy evaluation, which may itself be a protracted iterative computation requiring multiple sweeps through

PAGE EDG3
Date : / /

the state set.

→ Value Iteration - variant of policy iteration

→ value iteration simplifies this by truncating the policy evaluation step to just one sweep through the state set, effectively combining policy evaluation and policy improvement into a single step.

Update rule for value iteration is :

$$v_{k+1}(s) \doteq \max_a E\left[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s, A_t = a\right]$$

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

This rule is similar to the policy evaluation update, but it includes a maximization over all actions a, which incorporates policy improvement directly into update.

→ In practice, value iteration is terminated when the change in the value function is small enough, indicating that it has approximately converged to v^*. The algo then derives an optimal policy from the final value function by selecting, for each state, the action that maximizes the expected return according to the value function.

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

Algorithm parameter : a small threshold $\theta > 0$ determining accuracy of estimation

Initialize $V(s)$ for all $s \in S^+$, arbitrarily except

that $V(\text{terminal}) = 0$

loop:

$\Delta \leftarrow 0$

loop for each $s \in S$:

$v \leftarrow V(s)$

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

$\Delta \leftarrow \max(\Delta, |v - V(s)|)$

until $\Delta < \theta$

Output a deterministic policy, $\pi \approx \pi_*$ such that

$\pi(s) = \text{argmax}_a \sum_{s',r} p(s', r | s, a)[r + \gamma V(s')]$

Note: Value iteration is obtained simply by turning the Bellman optimality eq^n into an update rule.

→ Value iteration is often faster than policy iteration bcoz it avoids the separate, potentially lengthy policy evaluation step. It effectively combines one sweep of policy evaluation and one sweep of policy improvement in each iteration. Moreover, variations of value iteration can be created by adjusting the num of policy evaluation sweeps b/w each policy improvement sweep, allowing balance b/w a computational efficiency and convergence speed.

Imagine you are playing a game, finding a best way from Start → Find in a maze with lot of paths

1) Policy Evaluation (Checking Your Path): You pick a path and walk - get some pts. Track of pts you get

PAGE EDGE
Date : / /

for the whole path.

2) Policy Improvement (Finding a better path): If you find a step where changing a direction gets you more pts, you change your path.

3) Policy Iteration (Keep Improving): Keep doing the above 2 steps over at over. You keep doing this till you don't find any more ways to get more pts.

4) Value Iteration (Shortcut): Instead of checking the whole path, you pick the best move at every step. You make the best move at we end up with best path faster.

Asynchronous Dynamic Programming :-

→ Drawback of traditional DP method — involve operations over the entire set of MDP which can be computationally infeasible for large-scale problem.

→ Asynchronous DP addresses this issue by updating the value of states in any order, without the need for systematic sweeps.

→ Key Characteristics of Asynchronous DP algorithm :

1) In-place updates : value of state are updated in place using most recent value of other states that are available.

2) Flexibility in state selection : Some states might be updated multiple times before others are updated once.

3) Convergence requirement: To ensure convergence all states must continue to be updated; no state can be permanently ignored. The sequence of state updates can be deterministic or stochastic, that is random.

4) Asynchronous value Iteration: An example of an asynchronous DP algorithm is a version of value iteration that updates the value of only one state at each step. Asymptotic convergence to the optimal value function v^* is guaranteed under certain conditions.

5) Efficiency: Improve the rate of progress by selecting states for updates in a way that allows value info to propagate efficiently. This can include updating states more frequently if they are more relevant to optimal policies or skipping updates for states that are less relevant.

6) Real time iteration: Asynchronous algo can be interleaved with real time iteration, allowing an agent to use the latest value of policy info for decision-making while experiencing the environment. Updates can be applied to states as the agent visits them, focusing computational resources on the most relevant parts of state space.

→ Asynchronous DP - offer more flexible & potentially more efficient approach to solve large-scale MDPs.

Generalized Policy Iteration :-

→ Policy Iteration consists of 2 simultaneous, interacting processes:
- Policy evaluation - making value function consistent with current policy.
- Policy improvement - making policy greedy wrt current value function.

PAGE EDGE
Date : / /

→ Policy iteration – these 2 processes alternate.

→ In value iteration – for eg: only a single iteration of policy evaluation is performed in b/w each policy improvement

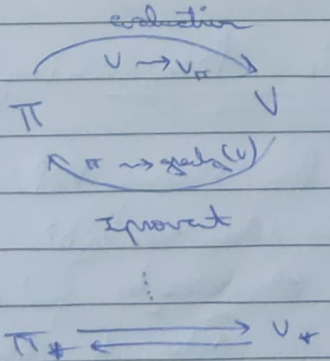
→ In asynchronous DP – the evaluation and improvement are interleaved at ~~each~~ an even finer grain

→ Generalized Policy Iteration (GPI) – encompasses a wide range of RL methods

- It describes the iteration b/w 2 processes: policy evaluation and policy improvement. These processes work together to find the optimal value function and policy even though they might operate at different granularities and in various ways across diff algo.

→ Almost all RL methods are GPI

All have identifiable policies and value functions, with ~~the~~ policy always being improved wrt to value function and value function being driven towards value function for policy



If both the evaluation process and improvement process stabilize, then value function and policy optimal

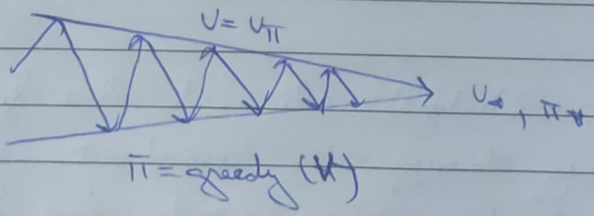
VI - highlights the fundamental iteration that underlying improving them, which is at heart of doing optimal in RL

The value function stabilizes only when it is consistent with the current policy and the policy stabilizes only when it is greedy with respect to current value function.

Thus both processes stabilize only when a policy has been found that is greedy with respect to its own evaluation function

→ Evaluation and Improvement process in GPI can be viewed as both competing and cooperating.
They compete - pull in opposite direction. Making the policy greedy with value function typically makes value function incorrect for changed policy, and making value function consistent with policy typically causes that policy no longer to be greedy

In long run, however the process interact to find optimal value function and a optimal policy



Efficiency of Dynamic Programming:

→ DP is powerful method for solving MDP.

→ Efficiency: In worst case, time taken by DP to find an optimal policy is polynomial in the number of states and actions. This is exponentially faster than a direct search in policy space, which would require examining each of k^n possible deterministic policies (n - no of states, k - no of action)

PAGE EDGE
Date : / /

→ Handling large State Spaces: Although the curse of dimensionality (exponential growth of of state space with the num of state variables) poses challenge, DP is comparatively better suited to handle large state than methods like direct search or linear programg.

→ Faster Convergence with good Initialization: Policy iteration and value iteration converg much faster than their theoretical worst-case run times. Starting with good initial value functions or policies can further speed up convergence.

→ Asynchronous Methods: For large state spaces, asynchronous DP preferred. These methods update the value of state in any order and do not require updation of memory for every state in each sweep.

→ Bootstrapping:

- special prop of DP methods
- all of them update estimate of the value of state based on the estimates of the values of successor states.
- That is they update estimates based on part of the estimate
- May RL prform bootstrapping even though they do not require as DP requires it (DP requires complete and accurate model of env).