Dynamic Programming

Reinforcement Learning : Module 04

Policy Evaluation (Prediction), Policy Improvement, Policy Iteration, Value Iteration, Asynchronous Dynamic Programming, Generalized Policy Iteration

The slides contain

- Excerpts from the book Reinforcement Learning : An Introduction, 2nd edition, Richard S. Sutton and Andrew G. Barto
- Content acquired using ChatGPT, Gemini and CoPilot

the "policy" is a strategy that the agent follows to decide which action to take in each state of the environment. An "optimal policy" is the best possible strategy that maximizes the agent's long-term rewards. The goal of RL algorithms is to find or learn this optimal policy.

Dynamic Programming

Perfect Model of the Environment: An environment in RL is typically represented as a Markov Decision Process (MDP) which consists of states, actions, transition probabilities, and rewards. Having a perfect model means that the agent knows exactly how the environment behaves. It knows the probabilities of transitioning from one state to another when taking specific actions, as well as the rewards associated with each state-action pair.

Markov Decision Process (MDP), which consists of states, actions, transition probabilities, and rewards. The objective is to find an optimal policy that dictates which action to take in each state to maximize the cumulative reward over time.

- The term dynamic programming (DP) refers to a collection of algorithms that can be used to compute optimal policies given a perfect model of the environment as a Markov decision process (MDP).
- Classical DP algorithms are of limited utility in reinforcement learning both because of their assumption of a perfect model and because of their great computational expense, but they are still important theoretically.
- The methods - to be discussed - can be viewed as attempts to achieve much the same effect as DP, only with less computation and without assuming a perfect model of the environment.

while classical DP remains important theoretically in RL, practical considerations have led to the development of alternative methods that can achieve similar objectives with reduced computational requirements and without assuming perfect knowledge of the environment.

value functions are a powerful tool within DP for finding good policies in RL. They act as a guide, helping the agent navigate the environment and make informed decisions that maximize its long-term rewards.

Dynamic Programming

- The key idea of DP, and of reinforcement learning generally, is the use of value functions to organize and structure the search for good policies.
- We can easily obtain optimal policies once we have found the optimal value functions, v_* or q_*, which satisfy the Bellman optimality equations:

$$\begin{aligned} v_*(s) &= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \max_a \sum_{s',r} p(s', r \mid s, a) \Big[r + \gamma v_*(s') \Big], \text{ or} \end{aligned} \tag{4.1}$$

$$\begin{aligned} q_*(s,a) &= \mathbb{E}\Big[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \;\Big|\; S_t = s, A_t = a \Big] \\ &= \sum_{s',r} p(s', r \mid s, a) \Big[r + \gamma \max_{a'} q_*(s', a') \Big], \end{aligned} \tag{4.2}$$

$$\text{for all } s \in \mathcal{S},\ a \in \mathcal{A}(s), \text{ and } s' \in \mathcal{S}^+.$$

DP algorithms in reinforcement learning are derived by translating Bellman equations into update rules for improving the agent's estimates of value functions. These update rules guide the agent in iteratively refining its approximations of the true value functions, even in situations where exact calculations are not feasible due to the complexity of the environment.

Dynamic Programming

- (As we shall see)
- DP algorithms are obtained by turning Bellman equations such as 4.1 and 4.2 into assignments, that is, into update rules for improving approximations of the desired value functions.
  - **Update rules :** Imagine these update rules as instructions for progressively improving the agent's estimates of the value functions (v or q).
  - **Approximations and Improvement**: In real-world applications, the environment might be too complex to calculate the exact value functions directly. So, DP algorithms rely on approximations. These approximations are initially rough estimates of the true value functions.

The Update Process

- The update rules based on the Bellman equations iteratively improve these approximations.
- In each iteration, the agent considers the rewards received, the value of the next state (according to the current approximation), and updates the estimated value of the current state.
- This process continues until the approximations converge to a good estimate of the actual value functions.
- **Analogy** : Imagine passing through a maze blindfolded.

The Blindfolded Maze Analogy:
The maze analogy is a great way to visualize the update process. Imagine yourself blindfolded in a maze.
You start by exploring the maze randomly, assigning values to different sections based on how good or bad they seem (e.g., dead ends = bad, open paths = good).
As you explore further, you encounter rewards (reaching a cheese hidden somewhere) and penalties (bumping into walls).
With each step, you update your mental map of the maze, assigning higher values to paths that lead to rewards and lower values to dead ends.
Over time, your mental map (approximation) becomes more accurate, reflecting the true layout and value of the maze.

Policy Evaluation (Prediction)

Revision : What is Policy?

- Policy defines the strategy an agent uses to navigate its environment and make decisions.
- **Function of a Policy**:
  - The policy acts as a mapping function. It takes the current state of the environment (a set of features representing the situation) as input and outputs an action for the agent to take.
  - Ideally, the policy guides the agent towards actions that maximize its long-term reward.
- **Types of Policies**:
  - **Deterministic Policies**: These policies always recommend the same action for a given state. Imagine a robot following a pre-programmed path in a factory.
  - **Stochastic Policies**: These policies assign a probability distribution over possible actions for each state. The agent randomly chooses an action based on these probabilities. This can be useful for exploration in unknown environments or dealing with uncertainty.

Policy Evaluation (Prediction)

- Let us consider how to compute the state-value
  arbitrary policy π.
- This is called policy evaluation in the DP literatu
  - Also referred as the prediction problem.

Policy Evaluation (Prediction)

- For all s ∈ ***S***,

$$
\begin{aligned}
v_\pi(s) &\doteq \mathbb{E}_\pi[G_t \mid S_t = s] \\
&= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\
&= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s] && (4.3) \\
&= \sum_a \pi(a|s) \sum_{s',r} p(s', r \mid s, a) \Big[r + \gamma v_\pi(s') \Big] && (4.4)
\end{aligned}
$$

where

- G_t is expected return, sum of the sequence of rewards received after time step t.
- π (a|s) is the probability of taking action a in state s under policy π, and the expectations are subscripted by π to indicate that they are conditional on π being followed

Iterative Policy Evaluation

- Consider a sequence of approximate valu
each mapping $\mathbf{S}^+$ to **R** (the real numbers).
- The initial approximation, v_0, is chosen ar
terminal state, if any, may be given value 0), and each successive approximation is obtained by using the Bellman equation for v_π (4.4) as an update rule:

$$\begin{aligned} v_{k+1}(s) &\doteq \mathbb{E}_\pi[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r \mid s, a) \Big[r + \gamma v_k(s') \Big] \end{aligned} \qquad (4.5)$$

- This algorithm is called *iterative policy evaluation.*

https://www.analyticsvidhya.com/blog/2018/09/reinforcement-learning-model-based-planning-dynamic-programming

Iterative Policy Evaluation

- To produce each successive approximation, v_{k+1} from v_k, iterative policy evaluation applies the same operation to each state s: it replaces the old value of s with a new value obtained from the old values of the successor states of s, and the expected immediate rewards, along all the one-step transitions possible under the policy being evaluated.
- We call this kind of operation an expected update.
- Each iteration of iterative policy evaluation updates the value of every state once to produce the new approximate value function v_{k+1}

Iterative Policy Evaluation

- There are several different kinds of expected updates, depending on whether a state (as here) or a state–action pair is being updated, and depending on the precise way the estimated values of the successor states are combined.
- All the updates done in DP algorithms are called expected updates because they are based on an expectation over all possible next states rather than on a sample next state.

Developing a computer program

- To write a sequential computer program to implement iterative policy evaluation as given by (4.5) you would have to use two arrays, one for the old values, $v_k(s)$, and one for the new values, $v_{k+1}(s)$.
- With two arrays, the new values can be computed one by one from the old values without the old values being changed.
- Of course it is easier to use one array and update the values “in place,” that is, with each new value immediately overwriting the old one.
- Then, depending on the order in which the states are updated, sometimes new values are used instead of old ones on the right-hand side of (4.5).

Developing a computer program

- This in-place algorithm also converges to v_π; in fact, it usually converges faster than the two-array version, because it uses new data as soon as they are available.
- We think of the updates as being done in a sweep through the state space.
- For the in-place algorithm, the order in which states have their values updated during the sweep has a significant influence on the rate of convergence.
- We usually have the in-place version in mind when we think of DP algorithms.

Pseudocode

Iterative Policy Evaluation, for estimating $V \approx v_\pi$

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

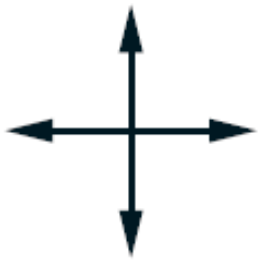
Loop:
  $\Delta \leftarrow 0$
  Loop for each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)\left[r + \gamma V(s')\right]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Pseudocode

- A complete in-place version of iterative policy evaluation is shown in pseudocode in the box.
- Note how it handles termination.
- Formally, iterative policy evaluation converges only in the limit, but in practice it must be halted short of this.
- The pseudocode tests the quantity $\max_{s \in \mathbf{s}} |v_{k+1}(s) - v_k(s)|$ after each sweep and stops when it is sufficiently small.

Example 4.1

- Consider the 4 X 4 gridworld shown below.

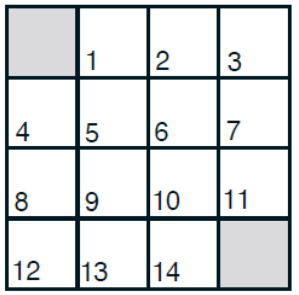


actions

	1	2	3
4	5	6	7
8	9	10	11
12	13	14	

$$R_t = -1$$
on all transitions

Example 4.1

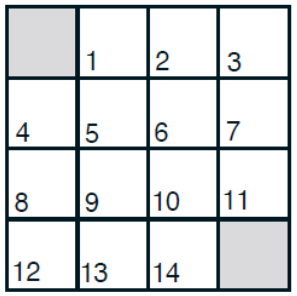


$R_t = -1$
on all transitions

The nonterminal states are $\mathcal{S} = \{1, 2, \ldots, 14\}$. There are four actions possible in each state, $\mathcal{A} = \{\texttt{up}, \texttt{down}, \texttt{right}, \texttt{left}\}$, which deterministically cause the corresponding state transitions, except that actions that would take the agent off the grid in fact leave the state unchanged. Thus, for instance, $p(6, -1|5, \texttt{right}) = 1$, $p(7, -1|7, \texttt{right}) = 1$, and $p(10, r|5, \texttt{right}) = 0$ for all $r \in \mathcal{R}$. This is an undiscounted, episodic task. The reward is -1 on all transitions until the terminal state is reached. The terminal state is shaded in the figure (although it is shown in two places, it is formally one state). The expected reward function is thus $r(s, a, s') = -1$ for all states s, s' and actions a. Suppose the agent follows the equiprobable random policy (all actions equally likely). The left side of Figure 4.1 shows the sequence of value functions $\{v_k\}$ computed by iterative policy evaluation.

Example 4.1

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



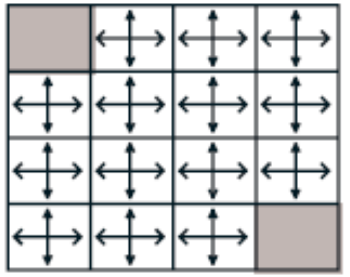
$R_t = -1$
on all transitions

$k = 0$

v_k for the random policy

0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0

greedy policy w.r.t. v_k



← random policy

Calculate $V_1(1)$

Action order assumed is : Left, Up, Right, Down

$V_1(1)$

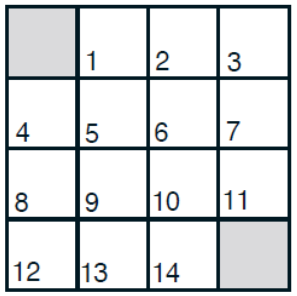
$= [0.25*(-1+1*v_0(0)] + [0.25*(-1+1*v_0(1)] + [0.25*(-1+1*v_0(2)] + [0.25*(-1+1*v_0(5)]$

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

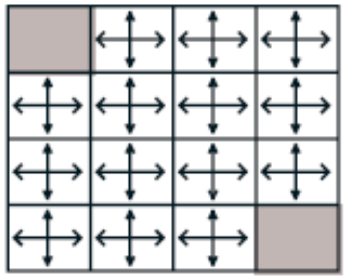
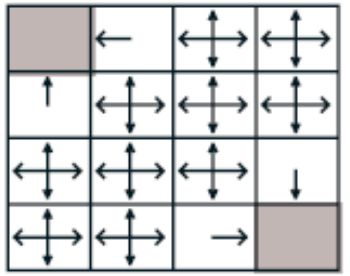
$= -1$

Example 4.1

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



$R_t = -1$
on all transitions

	v_k for the random policy	greedy policy w.r.t. v_k
$k = 0$	0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0	 ← random policy
$k = 1$	0.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 -1.0 0.0	

Calculate $V_2(1)$

Action order assumed is : Left, Up, Right, Down

$V_2(1)$

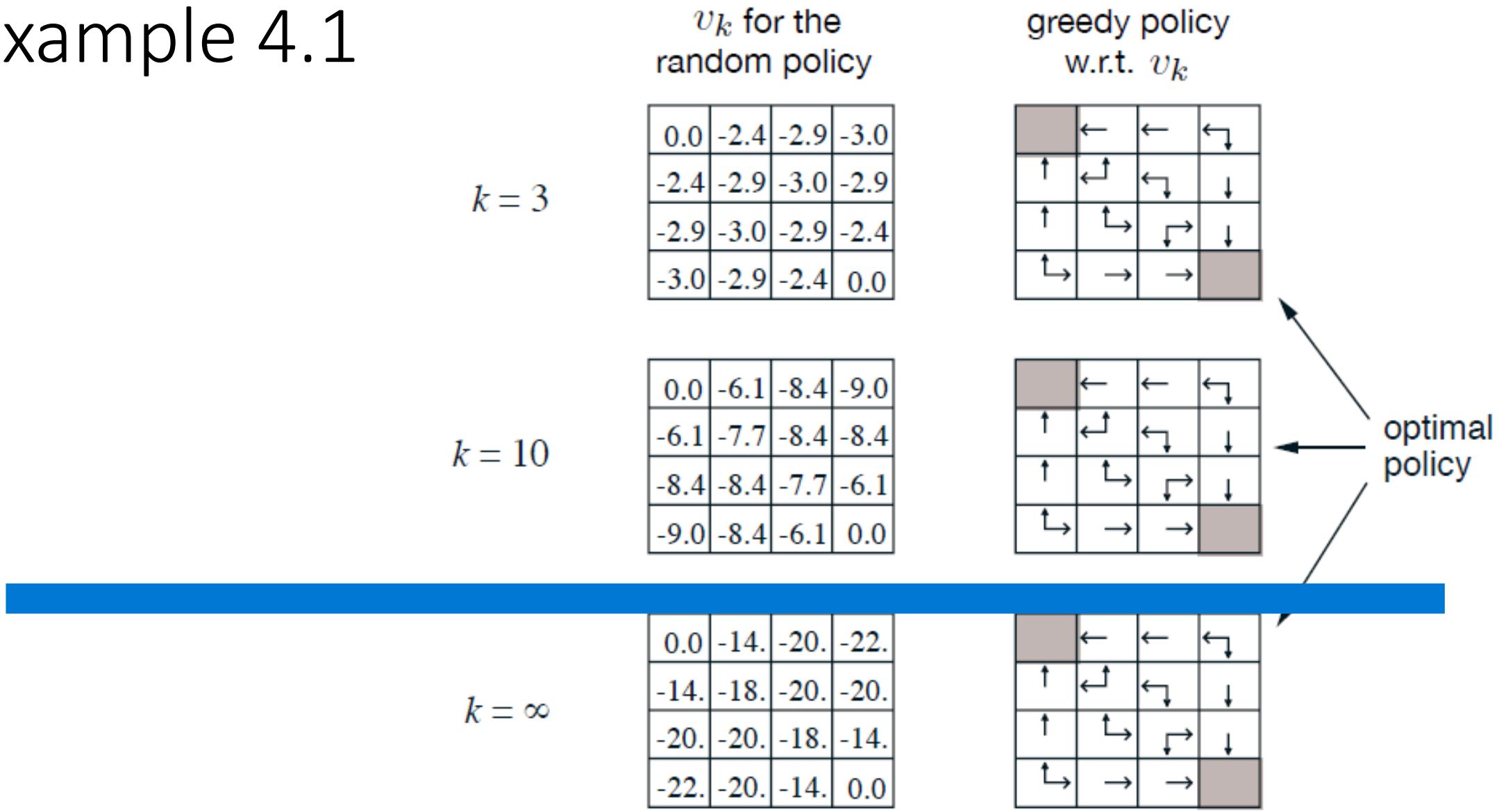
= [0.25*(-1+1*$v_1(0)$] + [0.25*(-1+1*$v_1(1)$] + [0.25*(-1+1*$v_1(2)$] + [0.25*(-1+1*$v_1(5)$]

= [0.25* (-1+0)] + [0.25* (-1+-1)] + [0.25* (-1+-1)] + [0.25* (-1+-1)]

= -0.25 + (-0.5) + (-0.5) + (-0.5) + (-0.5)

= -1.75

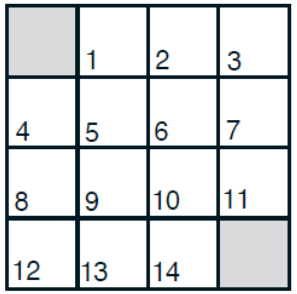
Example 4.1


v_k for the random policy
greedy policy w.r.t. v_k
$k = 3$
0.0 -2.4 -2.9 -3.0
-2.4 -2.9 -3.0 -2.9
-2.9 -3.0 -2.9 -2.4
-3.0 -2.9 -2.4 0.0
$k = 10$
0.0 -6.1 -8.4 -9.0
-6.1 -7.7 -8.4 -8.4
-8.4 -8.4 -7.7 -6.1
-9.0 -8.4 -6.1 0.0
$k = \infty$
0.0 -14. -20. -22.
-14. -18. -20. -20.
-20. -20. -18. -14.
-22. -20. -14. 0.0
optimal policy

Example 4.1

- The final estimate is in fact v_π, which in this case gives for each state the negation of the expected number of steps from that state until termination.

Exercise



$R_t = -1$
on all transitions

$k = \infty$

0.0	-14.	-20.	-22.
-14.	-18.	-20.	-20.
-20.	-20.	-18.	-14.
-22.	-20.	-14.	0.0

- In Example 4.1, if π is the equiprobable random policy,
  - What is $q_\pi(11, \text{down})$?
  - What is $q_\pi(7, \text{down})$?

Answer

$q_\pi(s, a) = r + \gamma v_\pi(s')$

where reward is always -1 and rewards are not discounted.

$q_\pi(s, a) = -1 + v_\pi(s')$

$q_\pi(11, down) = -1 + v_\pi(15) = -1 + 0 = -1$

$q_\pi(7, down) = -1 + v_\pi(11) = -1 + (-14) = -15$ ($v_\pi(11)$ is looked up from figure 4.1)

Exercise

Exercise 4.2 In Example 4.1, suppose a new state 15 is added to the gridworld just below state 13, and its actions, `left`, `up`, `right`, and `down`, take the agent to states 12, 13, 14, and 15, respectively. Assume that the transitions *from* the original states are unchanged. What, then, is $v_\pi(15)$ for the equiprobable random policy? Now suppose the dynamics of state 13 are also changed, such that action `down` from state 13 takes the agent to the new state 15. What is $v_\pi(15)$ for the equiprobable random policy in this case? □

Answer

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

where actions are equaprobable, reward is always -1 and rewards are not discounted.

In first case, the new state is not reachable from state 13.

$v_\pi(s) = \sum_a 0.25[(-1) + v_\pi(s')]$

where $v(12) = -22, v(13) = -20, v(14) = -14, v(9) = -20$:

$v_\pi(15) = (-1) + 0.25 * [v_\pi(12) + v_\pi(13) + v_\pi(14) + v_\pi(15)]$

$v_\pi(15) = -15 + 0.25 v_\pi(15) = -20$

In second case, the new state is reachable from state 13. Now value of 13 depends on value of 15, and value of 15 depends on value of 13. We have two equations with two unknowns which can be solved.

$v_\pi(15)$ is :

$v_\pi(15) = (-1) + 0.25 * [v_\pi(12) + v_\pi(13) + v_\pi(14) + v_\pi(15)]$

$v_\pi(15) = -1 + 0.25 * [-36 + v_\pi(13) + v_\pi(15)] = -1 - 9 + \frac{v_\pi(13)}{4} + \frac{v_\pi(15)}{4}$

$v_\pi(15) = \frac{-40 + v_\pi(13)}{3}$

$v_\pi(13)$ is :

$v_\pi(13) = (-1) + 0.25 * [v_\pi(12) + v_\pi(9) + v_\pi(14) + v_\pi(15)]$

$v_\pi(13) = -1 + 0.25 * [-56 + v_\pi(15)] = -1 - 14 + \frac{v_\pi(15)}{4} = -15 + \frac{v_\pi(15)}{4}$

One can plug $v_\pi(13)$ to obtain $v_\pi(15)$:

$v_\pi(15) = \frac{-40 + -15 + \frac{v_\pi(15)}{4}}{3} = -20$

Exercise

- Assume a game to be played as follows.
  1. There are 2 options for the player: *play* or *quit*
  2. If quit is chosen then 10 points are granted to the player and the game ends.
  3. If play is chosen then 4 points are granted to the player and the game continues with the rolling of a 6 faced die.
     i. If 1 or 2 occurs on the die then the game ends, with no additional points awarded to player.
     ii. If 3 to 6 occurs on the die then then follow step 1.
- Question : Design this game as MDP.

Exercise

Exercise 4.3 What are the equations analogous to (4.3), (4.4), and (4.5) for the action-value function q_π and its successive approximation by a sequence of functions $q_0, q_1, q_2, \ldots$?

$$
\begin{aligned}
q_\pi(s,a) &\doteq \mathbb{E}_\pi\big[G_t \,|\, S_t = s,\, A_t = a\big] \\
&= \mathbb{E}_\pi\big[R_{t+1} + \gamma G_{t+1} \,|\, S_t = s,\, A_t = a\big] \\
&= \mathbb{E}_\pi\Big[R_{t+1} + \gamma \sum_{s',a'} q_\pi(s',a') \,|\, S_t = s,\, A_t = a\Big] \\
&= \sum_{s',r} p(s',r|s,a)\Big[r + \gamma \sum_{a'} \pi(a'|s') q_\pi(s',a')\Big]
\end{aligned}
$$

$$
\begin{aligned}
q_{k+1}(s,a) &\doteq \mathbb{E}_\pi\big[R_{t+1} + \gamma G_{t+1} \,|\, S_t = s,\, A_t = a\big] \\
&= \sum_{s',r} p(s',r|s,a)\Big[r + \gamma \sum_{a'} \pi(a'|s') q_k(s',a')\Big]
\end{aligned}
$$

Policy Improvement

The reason for computing the value function for a policy is to help find better policies.

What is Policy Improvement

- The process of creating a new policy that outperforms the original policy.
- Achieved by evaluating the current policy and identifying actions that lead to higher rewards in specific states.
- The new policy prioritizes these actions, leading to potentially better outcomes.

a deterministic policy is a strategy where the agent, given a specific state, always takes the same action. eg , robot navigating a maze

Policy Improvement : Introduction

- Suppose we have determined the value function v_π for an arbitrary deterministic policy π.
- For some state s we would like to know whether or not we should change the policy to deterministically choose an action $a \neq \pi(s)$.
- We know how good it is to follow the current policy from s—that is $v_\pi(s)$—but would it be better or worse to change to the new policy *?*
- One way to answer this question is to consider selecting *a* in s and thereafter following the existing policy, π.
- The value of this way of behaving is

$$\begin{aligned} q_\pi(s,a) &\doteq \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \sum_{s',r} p(s',r|s,a)\Big[r + \gamma v_\pi(s')\Big]. \end{aligned} \tag{4.6}$$

Policy Improvement : Introduction

$$
\begin{aligned}
q_\pi(s,a) &\doteq \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = a] \\
&= \sum_{s',r} p(s', r \mid s, a)\Big[r + \gamma v_\pi(s')\Big].
\end{aligned}
\tag{4.6}
$$

- The key criterion is whether this is greater than or less than v_π (s).
- If it is greater
  - that is, if it is better to select *a* once in s and thereafter follow π than it would be to follow π all the time
  - then one would expect it to be better still to select *a* every time s is encountered, and that the new policy would in fact be a better one overall.

Policy Improvement Theorem

Let π and π' be any pair of deterministic policies such that, for all $s \in \mathcal{S}$,

$$q_\pi(s, \pi'(s)) \geq v_\pi(s). \tag{4.7}$$

Then the policy π' must be as good as, or better than, π. That is, it must obtain greater or equal expected return from all states $s \in \mathcal{S}$:

$$v_{\pi'}(s) \geq v_\pi(s$$

Policy Improvement Theorem : Proof

Starting from (4.7), we keep expanding the q_π side with (4.6) and reapplying (4.7) until we get $v_{\pi'}(s)$:

$$
\begin{aligned}
v_\pi(s) &\leq q_\pi(s, \pi'(s)) \\
&= \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = \pi'(s)] && \text{(by (4.6))} \\
&= \mathbb{E}_{\pi'}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s] \\
&\leq \mathbb{E}_{\pi'}[R_{t+1} + \gamma q_\pi(S_{t+1}, \pi'(S_{t+1})) \mid S_t = s] && \text{(by (4.7))} \\
&= \mathbb{E}_{\pi'}[R_{t+1} + \gamma \mathbb{E}_{\pi'}[R_{t+2} + \gamma v_\pi(S_{t+2}) | S_{t+1}, A_{t+1} = \pi'(S_{t+1})] \mid S_t = s] \\
&= \mathbb{E}_{\pi'}\big[R_{t+1} + \gamma R_{t+2} + \gamma^2 v_\pi(S_{t+2}) \;\big|\; S_t = s\big] \\
&\leq \mathbb{E}_{\pi'}\big[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 v_\pi(S_{t+3}) \;\big|\; S_t = s\big] \\
&\vdots \\
&\leq \mathbb{E}_{\pi'}\big[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \cdots \;\big|\; S_t = s\big] \\
&= v_{\pi'}(s).
\end{aligned}
$$

Policy

- We saw
  evaluat
- It is a na
  possible
  according to $q_\pi(s, a)$. In other words, to consider the new greedy policy, π', given by

$$\begin{aligned}\pi'(s) &\doteq \arg\max_a q_\pi(s,a) \\ &= \arg\max_a \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t = a] \qquad (4.9) \\ &= \arg\max_a \sum_{s',r} p(s', r \mid s, a)\Big[r + \gamma v_\pi(s')\Big],\end{aligned}$$

Policy Improvement

- The greedy policy takes the action that looks best in the short term—after one step of lookahead—according to v_π.
- By construction, the greedy policy meets the conditions of the policy improvement theorem (4.7), 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 with respect to the value function of the original policy, is called *policy improvement*.

Policy Improvement

Suppose the new greedy policy, π', is as good as, but not better than, the old policy π. Then $v_\pi = v_{\pi'}$, and from (4.9) it follows that for all $s \in \mathcal{S}$:

$$\begin{aligned} v_{\pi'}(s) &= \max_a \mathbb{E}[R_{t+1} + \gamma v_{\pi'}(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \max_a \sum_{s',r} p(s', r \mid s, a) \Big[r + \gamma v_{\pi'}(s') \Big]. \end{aligned}$$

But this is the same as the Bellman optimality equation (4.1), and therefore, $v_{\pi'}$ must be v_*, and both π and π' must be optimal policies. Policy improvement thus must give us a strictly better policy except when the original policy is already optimal.

Policy Iteration

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''. We can thus obtain a sequence of monotonically improving policies and value functions:

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

where $\xrightarrow{\text{E}}$ denotes a policy *evaluation* and $\xrightarrow{\text{I}}$ denotes a 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 number of policies, this process must converge to an optimal policy and optimal value function in a finite number of iterations.

This way of finding an optimal policy is called ***policy iteration*.**

Policy Iteration 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 $s \in \mathcal{S}$

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

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

Example 4.2 : Jack’s Car Rental

- Students are recommended to study Jack’s Car Rental problem discussed on Page 81 of the book.

Exercise 4.4

- The policy iteration algorithm on page 80 has a subtle bug in that it may never terminate if the policy continually switches between two or more policies that are equally good.
- This is ok for pedagogy, but not for actual use.
- Modify the pseudocode so that convergence is guaranteed.

Solution

In the step 3. Policy Improvement, it said:

If old-action $\neq \pi(s)$ *, then .....*

It is a bug and one way to fix it is to say the following instead:

If old-action $\notin \{a_i\}$*, which is the all equi-best solutions from* $\pi(s)$*, ...*

Exercise 4.5

- How would policy iteration be defined for action values?
- Give a complete algorithm for computing q_*, analogous to that on page 80 for computing v_*.
- Please pay special attention to this exercise, because the ideas involved will be used throughout the rest of the book.

Solution

1. Initialization

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

2. Policy Evaluation

Loop:

$\Delta \leftarrow 0$

Loop for each $s \in S$ and $a \in A$:

$q = Q(s,a)$

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

$\Delta \leftarrow \max(\Delta, |q - Q(s,a)|)$

until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

3. Policy Improvement

policy-stable $\leftarrow$ *true*

For each $s \in S$ and $a \in A$:

old-action $\leftarrow \pi(s)$

$\pi(s) \leftarrow \arg\max_a Q(s,a)$

If *old-action* $\notin \{a_i\}$, which is the set of equi-best solutions from $\pi(s)$

Then *policy-stable* $\leftarrow$ *false*

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

Value Iteration

Value Iteration

- One drawback to policy iteration is that each of its iterations involves policy evaluation, which may itself be a protracted iterative computation requiring multiple sweeps through the state set.
- If policy evaluation is done iteratively, then convergence exactly to v_π occurs only in the limit.
- Must we wait for exact convergence, or can we stop short of that?
- The example in Figure 4.1 certainly suggests that it may be possible to truncate policy evaluation.
  - In that example, policy evaluation iterations beyond the first three have no effect on the corresponding greedy policy.

Value Iteration

- In fact, the policy evaluation step of policy iteration can be truncated in several ways without losing the convergence guarantees of policy iteration.
- One important special case is when policy evaluation is stopped after just one sweep (one update of each state).
- This algorithm is called *value iteration*.

Value Iteration

- It can be written as a particularly simple update operation that combines the policy improvement and truncated policy evaluation steps:

$$\begin{aligned} v_{k+1}(s) &\doteq \max_a \mathbb{E}[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \max_a \sum_{s',r} p(s', r \mid s, a) \Big[r + \gamma v_k(s') \Big], \end{aligned} \tag{4.10}$$

- Note that value iteration is obtained simply by turning the Bellman optimality equation into an update rule.
- Also note how the value iteration update is identical to the policy evaluation update (4.5) except that it requires the maximum to be taken over all actions

Value Iteration

- Let us consider how value iteration terminates.
- Like policy evaluation, value iteration formally requires an infinite number of iterations to converge exactly to v_*.
- In practice, we stop once the value function changes by only a small amount in a sweep.
- The box below shows a complete algorithm with this kind of termination condition.

Value Iteration

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

Output a deterministic policy, $\pi \approx \pi_*$, such that
$\pi(s) = \arg\max_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V(s')\big]$

Example 4.3 : Gambler’s Problem

- Students are recommended to study Gambler’s problem discussed on Page 84 of the book.

Asynchronous Dynamic Programming

Traditional DP algorithms require iterating through the entire state space of the MDP (Markov Decision Process) during each update. This is called a "sweep."
For problems with massive state spaces (like backgammon with its 10^20 states), a single sweep can be incredibly slow or even impossible.

Asynchronous Dynamic Programming

- A major drawback to the DP methods that we have discussed so far is that they involve operations over the entire state set of the MDP, that is, they require sweeps of the state set.
- If the state set is very large, then even a single sweep can be prohibitively expensive.
- For example, the game of backgammon has over 10^{20} states. Even if we could perform the value iteration update on a million states per second, it would take over a thousand years to complete a single sweep.

Asynchronous Dynamic Programming

- Asynchronous Dynamic Programming (ADP) refers to a class of algorithms used to solve Markov Decision Processes (MDPs) by asynchronously updating the value function or policy of states.
- Unlike traditional dynamic programming algorithms, such as value iteration and policy iteration, which update all states in a synchronized manner, ADP updates states individually and in any order.
- This asynchronous updating process offers several advantages in reinforcement learning settings:

Advantages of ADP

- **Efficiency**:
  - ADP can be more computationally efficient than synchronous dynamic programming algorithms, especially in large-scale problems where updating all states simultaneously may be computationally prohibitive.
  - By focusing computational resources on states that are most in need of updating, ADP can converge to an optimal solution more quickly.
- **Scalability**:
  - Asynchronous updates allow for more scalable solutions to reinforcement learning problems, as they enable the algorithm to handle large state spaces more efficiently.
  - By updating states independently, ADP can be applied to problems with millions or even billions of states.

Advantages of ADP

- **Exploration-Exploitation Trade-off:**
  - ADP can help balance exploration and exploitation in reinforcement learning. By updating states asynchronously, the algorithm can focus on exploring regions of the state space that are less explored or have higher uncertainty, while exploiting known information in other regions.
- **Incremental Updates**:
  - ADP typically performs incremental updates to the value function or policy, updating only the affected states based on changes in neighboring states.
  - This incremental update process can lead to faster convergence and reduced computational overhead compared to recomputing the entire value function or policy in each iteration.

Here's an analogy: Imagine exploring a maze. Traditional DP would require remapping the entire maze after each step. Incremental updates in ADP are like only updating the immediate area around your current position as you explore, making the process more efficient.

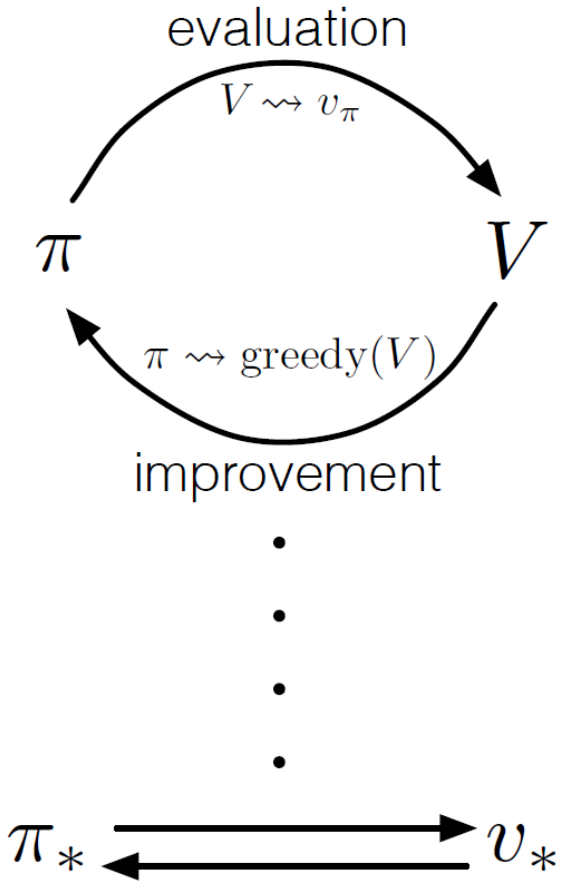
Asynchronous Dynamic Programming

- Popular ADP algorithms in reinforcement learning include asynchronous value iteration, asynchronous policy iteration, and various variants of asynchronous Q-learning.
- These algorithms have been successfully applied to a wide range of reinforcement learning tasks, including robotic control, game playing, and autonomous decision-making

Generalized Policy Iteration

Generalized Policy Iteration

- The term *generalized policy iteration* (GPI) refers to the general idea of letting policy-evaluation and policy improvement processes interact, independent of the granularity and other details of the two processes.
- Almost all reinforcement learning methods are well described as GPI.
- That is, all have identifiable policies and value functions, with the policy always being improved with respect to the value function and the value function always being driven toward the value function for the policy, as suggested by the diagram to the right.



Generalized Policy Iteration

- If both the evaluation process and the improvement process stabilize, that is, no longer produce changes, then the value function and policy must be optimal.
- 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 the current value function.
- Thus, both processes stabilize only when a policy has been found that is greedy with respect to its own evaluation function.
- This implies that the Bellman optimality equation (4.1) holds, and thus that the policy and the value function are optimal.

the evaluation and improvement processes can indeed be seen as both competing and cooperating, and this duality is crucial for the learning process. Let's break down how they compete and cooperate:
Competing Nature :Pulling in Opposing Directions: Policy improvement typically aims to make the policy greedy with respect to the current value function. This means selecting actions that maximize the estimated value. However, when the policy becomes more greedy, it may explore fewer options, leading to potentially inaccurate value estimates for unexplored states or actions.

Generalized Policy Iteration

Impact on Value Function Accuracy: Conversely, updating the value function to be consistent with the current policy might cause it to no longer accurately represent the true values. For example, if the policy becomes more exploratory, the value function might need to adjust to reflect this increased uncertainty, which can lead to inaccuracies in estimating the expected returns.

- The evaluation and improvement processes in GPI can be viewed as both competing and cooperating.
- They compete in the sense that they pull in opposing directions.
- Making the policy greedy with respect to the value function typically makes the value function incorrect for the changed policy, and making the value function consistent with the policy typically causes that policy no longer to be greedy.
- In the long run, however, these two processes interact to find a single joint solution: the optimal value function and an optimal policy.

Cooperating Nature:

Iterative Improvement: Despite the competing objectives, the evaluation and improvement processes cooperate in the iterative refinement of the policy and value function. Each cycle of GPI allows for incremental progress towards better policies and more accurate value estimates.
Feedback Loop: The competing nature of evaluation and improvement creates a feedback loop. As the policy becomes more greedy, the value function needs to adjust to provide accurate estimates for the updated policy. Similarly, updates to the value function influence subsequent policy improvements. This iterative feedback loop drives the learning process towards convergence to optimal policies and value functions.

The flexibility of generalized policy iteration (GPI) makes it incredibly versatile and applicable to a wide range of reinforcement learning problems. Here's how its flexibility manifests:
Asynchronous Updates:
GPI allows for asynchronous updates, meaning that policy evaluation and improvement can occur independently and at different rates. This flexibility is beneficial in dynamic environments where the optimal policy may change over time or where computational resources are limited. Asynchronous updates enable the agent to continuously adapt its policy based on new information without being constrained by a fixed update schedule.

Generalized Policy Iteration

Interleaving Policy Evaluation and Improvement:
GPI does not prescribe a strict order or timing for policy evaluation and improvement. Instead, it allows these processes to be interleaved, meaning that the agent can alternate between evaluating the current policy and making improvements based on the evaluation results. This flexibility enables efficient use of computational resources and can lead to faster convergence to optimal policies.

- Generalized Policy Iteration (GPI) is crucial in reinforcement learning due to its ability to iteratively refine both the policy and the value function.
- By continuously evaluating and improving the agent's policy, GPI enables adaptive learning and optimal decision-making in complex environments.
- Its flexibility allows for asynchronous updates, interleaving policy evaluation and improvement, and accommodating various algorithms, making it applicable to a wide range of reinforcement learning problems.
- GPI provides a unified framework that balances exploration and exploitation, leading to faster convergence and more efficient learning.

Accommodating Various Algorithms:

GPI is compatible with a wide range of reinforcement learning algorithms, including both model-free and model-based approaches. Whether the problem requires value iteration, policy iteration, Q-learning, SARSA, actor-critic methods, or other algorithms, GPI provides a framework for integrating these techniques within the iterative cycle of policy evaluation and improvement. This versatility allows researchers and practitioners to choose the most appropriate algorithm based on the specific characteristics of the problem at hand.

Generalized Policy Iteration (GPI) is a fundamental concept in the field of reinforcement learning that emphasizes the interaction between policy evaluation and policy improvement, regardless of the specific details of each process. Here's a breakdown of what this means:

Policy Evaluation: Policy evaluation is the process of determining the quality of a given policy in terms of its expected long-term reward. This typically involves estimating the value function or the state-action value function under the current policy. The value function represents the expected return that an agent can achieve from a given state (or state-action pair) following the current policy.

Policy Improvement: Policy improvement is the process of enhancing the current policy to obtain a better one. This is usually achieved by selecting actions that are expected to lead to higher rewards based on the information gained from policy evaluation. The goal is to iteratively refine the policy to maximize the cumulative reward over time.

Now, in GPI, these two processes interact in a cyclic manner:

Interaction: The key idea is that policy evaluation and policy improvement are not treated as separate, independent phases. Instead, they continually influence each other in an iterative loop. After evaluating the current policy, insights gained from the evaluation are used to improve the policy. Then, the updated policy is evaluated again, leading to further improvements. This cyclic interaction allows for a continuous refinement of the policy over time.

Independence of Granularity: GPI emphasizes that the specific details and granularity of policy evaluation and policy improvement methods can vary. For instance, policy evaluation could involve methods like dynamic programming, Monte Carlo simulation, or temporal difference learning. Similarly, policy improvement can utilize techniques such as greedy policy updates, policy gradient methods, or exploration strategies like epsilon-greedy. GPI focuses on the overarching principle of iterative refinement, rather than prescribing specific algorithms or approaches.

By allowing policy evaluation and policy improvement to interact in this flexible manner, GPI provides a framework for systematically learning and improving policies in reinforcement learning problems. It emphasizes the importance of continuous feedback and adaptation in the learning process.

End

The statement you provided succinctly captures the essence of generalized policy iteration (GPI) and its applicability to most reinforcement learning methods. Let's break down why this is the case:

Identifiable Policies and Value Functions: In reinforcement learning, an agent interacts with an environment by taking actions and receiving rewards. The agent's behavior is governed by a policy, which specifies the actions it should take in different states. Additionally, the value function estimates the expected return or utility associated with being in a particular state and following a particular policy. Both policies and value functions are central concepts in reinforcement learning, providing a framework for decision-making and evaluation.

Policy Improvement with Respect to Value Function: The GPI framework emphasizes the iterative improvement of policies based on value function estimates. When the value function is updated, the policy can be improved by selecting actions that are expected to lead to higher value states. This process of policy improvement with respect to the value function drives the agent towards making better decisions over time.

Value Function Driven Toward the Value Function for the Policy: Conversely, the value function is also updated to better approximate the expected return under the current policy. By continually refining the value function, it becomes more accurate in estimating the long-term rewards associated with different states and actions. This, in turn, guides the policy improvement process, as the policy seeks to maximize the expected rewards estimated by the value function.

By maintaining this cyclic interaction between policy improvement and value function estimation, reinforcement learning methods ensure that policies and value functions converge towards optimal solutions, where the policy selects actions that maximize expected rewards and the value function accurately represents these expected rewards. Therefore, most reinforcement learning methods can indeed be described within the framework of GPI, highlighting its broad applicability and effectiveness in learning optimal policies in diverse environments.