Dynamic Programming

Reinforcement Learning

2019

Sepehr Maleki

University of Lincoln
School of Engineering

Policy Evaluation

- For an arbitrary policy π, the state-value function is:

$$
\begin{aligned}
v_\pi(s) &= \mathbb{E}_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ...|S_t = s] \\
&= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s] \\
&= \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)[r + \gamma v_\pi(s')] \ .
\end{aligned}
$$

- In the DP literature, this is called policy evaluation.

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

- If the environment's dynamics are completely known, then $v_\pi(s)$ describes $|\mathcal{S}|$ linear equations in $|\mathcal{S}|$ unknowns.

(S. Maleki 2019)

An Iterative Approach to Policy Evaluation

- Consider a sequence of approximate value functions $v_0, v_1, v_2, ...$, each mapping $\mathcal{S}^+$ to $\mathbb{R}$ ($\mathcal{S}^+$ is $\mathcal{S}$ plus a terminal state for an episodic problem).
- The initial approximation v_0, is chosen arbitrarily (except that the terminal state, if any, which is 0).
- The successive approximation is obtained by using the Bellman equation for v_π:

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

- The sequence $\{v_k\}$ will converge to v_π as $k \to \infty$.
- This algorithm is called the *iterative policy evaluation*.

(S. Maleki 2019)

Iterative Policy Evaluation Algorithm

Input π, the policy to be evaluated ;

Initialise an array $V(s) = 0$, for all $s \in \mathcal{S}^+$;

Repeat

$$\begin{aligned}
&\Delta \leftarrow 0 \\
&\texttt{For each } s \in \mathcal{S}^+: \\
&\quad v \leftarrow V(s) \\
&\quad V(s) \leftarrow \textstyle\sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')] \\
&\quad \Delta \leftarrow \max(\Delta, |v - V(s)|)
\end{aligned}$$

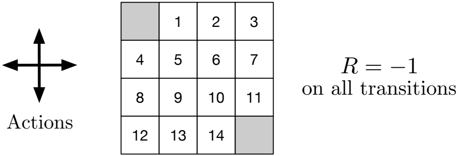
until $\Delta < \theta$ (a small positive number)

Output $V \approx v_\pi$

(S. Maleki 2019)

Example: Gridworld

Consider the 4×4 gridworld shown below:



- Non-terminal states: $\mathcal{S} = \{1, 2, ..., 14\}$.
- Four actions possible in each state: $\mathcal{A} = \{$`up, down, right, left`$\}$.
- Actions taking the agent off the grid leave the state unchanged.
- The episodic task is undiscounted.
- All actions are equally likely.

(S. Maleki 2019)

Example: Gridworld

v_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

v_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

v_2

0.0	-1.8	-2.0	-2.0
-1.8	-2.0	-2.0	-2.0
-2.0	-2.0	-2.0	-1.8
-2.0	-2.0	-1.8	0.0

v_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

v_4

0.0	-3.1	-3.8	-4.0
-3.1	-3.7	-3.9	-3.8
-3.8	-3.9	-3.7	-3.1
-4.0	-3.8	-3.1	0.0

v_{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

v_π

0.0	-14.0	-20.0	-22.0
-14.0	-18.0	-20.0	-20.0
-20.0	-20.0	-18.0	-14.0
-22.0	-20.0	-14.0	0.0

(S. Maleki 2019)

Policy Improvement

Problem statement: For some state s, we want to determine whether or not we should change the policy.

- We know how good it is to follow the current policy from s (that is $v_\pi(s)$).
- The action-value function is given by:

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

- Let π and π' be any pair of deterministic policies. The the policy π' is as good as or better than π if for all states $s \in \mathcal{S}$:

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

- In other words, π' is a better policy than π if for all states:

$$q_\pi(s, \pi'(s)) \geq v_\pi(s)$$

(S. Maleki 2019)

Proof For The Policy Improvement

$$
\begin{aligned}
v_\pi(s) &\leq q_\pi(s, \pi'(s)) \\
&= \mathbb{E}'_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s] \\
&\leq \mathbb{E}_{\pi'}[R_{t+1} + \gamma q_\pi(S_{t+1}, \pi'(S_{t+1}) | S_t = s] \\
&= \mathbb{E}_{\pi'}[R_{t+1} + \gamma \mathbb{E}_{\pi'}[R_{t+2} + \gamma v_\pi(S_{t+2}) | S_t = s] \\
&= \mathbb{E}_{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2 v_\pi(S_{t+2}) | S_t = s] \\
&\leq \mathbb{E}_{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 v_\pi(S_{t+3}) | S_t = s] \\
&\vdots \\
&\leq \mathbb{E}'_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + ... | S_t = s] \\
&= v'_\pi(s) \ .
\end{aligned}
$$

(S. Maleki 2019)

Greedy Policy

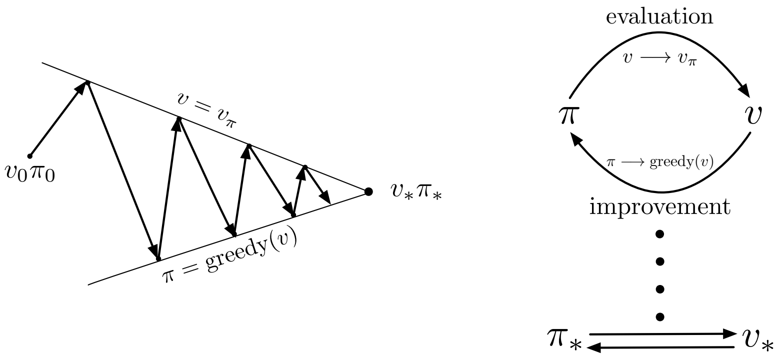
The greedy policy takes the action that looks best in the short term (after one step of lookahead) according to v_π.

$$
\begin{aligned}
\pi'(s) &= \operatorname*{argmax}_a q_\pi(s,a) \\
&= \operatorname*{argmax}_a \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = a] \\
&= \operatorname*{argmax}_a \sum_{s',r} p(s',r|s,a)[r + \gamma v_\pi(s')]
\end{aligned}
$$

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*.

(S. Maleki 2019)

Generalised Policy Iteration



- **Policy Evaluation**: Estimate v_π
- **Policy Improvement**: Generate $\pi' \geq \pi$

(S. Maleki 2019)

Policy Iteration

- We can iteratively compute value functions and improve the policy to a better one:

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

- Each policy is guaranteed to be a strict improvement over the previous one (unless already optimal).
- Because a finite MDP has only a finite number of policies, this process must converge to an optimal policy in a finite number of iterations.

(S. Maleki 2019)

Policy Iteration Algorithm

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

2. Policy Evaluation
   Repeat

   $\Delta \leftarrow 0$

   For each $s \in \mathcal{S}^+$:

   $v \leftarrow V(s)$

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

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

   until $\Delta < \theta$ (a small positive number)

3. Policy Improvement
   policy-stable $\longleftarrow$ $true$

   For each $2 \in \mathcal{S}$:

   $a \longleftarrow \pi(s)$

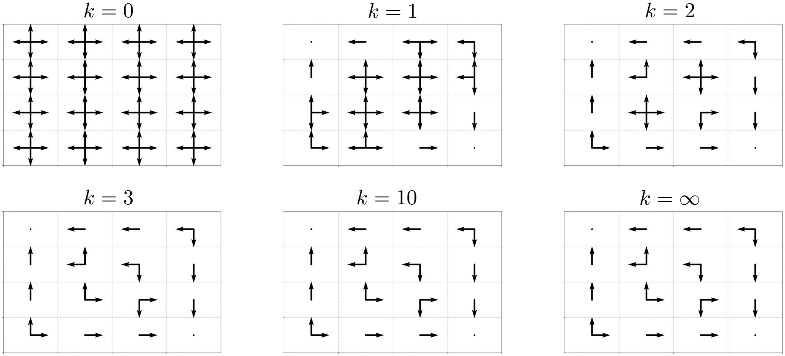
   $\pi(s) \longleftarrow \text{argmax}_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$

   If $a \neq \pi(s)$m then *policy-stable* $\longleftarrow$ $false$

   If *policy-stable*, then stop and return V and π; else, go to 2.

(S. Maleki 2019)

Example: Gridworld



(S. Maleki 2019)

Asynchronous Dynamic Programming

- A major drawback to the DP methods discussed so far, is that they involve operations over the entire state set of the MDP.
- If the state set is very large, then even a single sweep can be prohibitively expensive.
- For example, it would take over a thousand years to complete a single sweep in the game of backgammon which has over 10^{20} states.
- Asynchronous DP algorithms back up states individually, in any order using whatever values of other states happen to be available.
- To converge correctly, an asynchronous algorithm must continue to backup the values of all the states after some point.

(S. Maleki 2019)

Example: Jack's Car Rental

- States: Two locations, maximum of 20 cars at each.
- Actions: Move up to 5 cats between locations overnight (at $\$2$ per car).
- Reward: $\$10$ for each car rented (must be available).
- Transitions: Cars returned and requested randomly:
  - Poisson distribution, n returns/requests with probability $\frac{\lambda^n}{n!}e^{-\lambda}$.
  - 1st location: average requests $= 3$, average returns $= 3$.
  - 2nd location: average requests $= 4$, average returns $= 2$.
- $\gamma = 0.9$.

(S. Maleki 2019)

In-Place Dynamic Programming

- Synchronous value iteration stores two copies of value function for all $s \in \mathcal{S}$:

$$v_{new}(s) \longleftarrow \max_{a \in \mathcal{A}} \left(\sum_{s',r} p(s', r|s, a)[r + \gamma v_{old}(s')] \right)$$

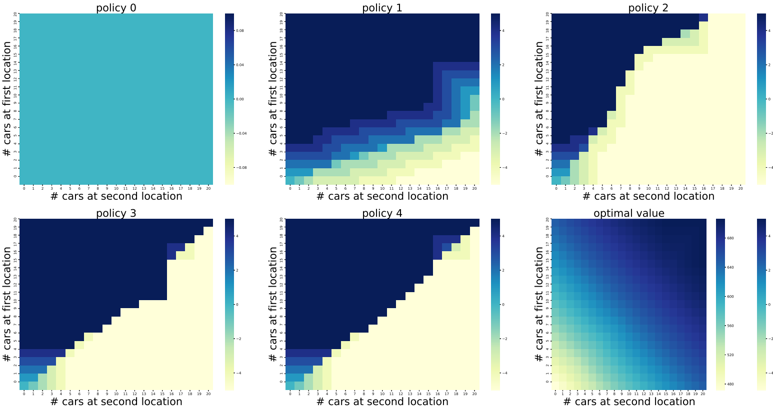
$$v_{old} \longleftarrow v_{new}$$

- In-place value iteration only stores one copy of the value function for all $s \in \mathcal{S}$:

$$v(s) \longleftarrow \max_{a \in \mathcal{A}} \left(\sum_{s',r} p(s', r|s, a)[r + \gamma v(s')] \right)$$

(S. Maleki 2019)

In-Place Dynamic Programming



(S. Maleki 2019)

Prioritised Sweeping

- Back up the state with the largest remaining Bellman error:

$$\left| \max_{a \in \mathcal{A}} \left(\sum_{s',r} p(s', r|s, a)[r + \gamma v(s')] \right) - v(s) \right| .$$

- Update Bellman error of affected states after each backup.
- Requires knowledge of reverse dynamics.
- Can be implemented efficiently by maintaining a priority queue.

(S. Maleki 2019)

Efficiency of Dynamic Programming

- DP may not be practical for very large problems, but are actually quite efficient comparatively.
- The time DP methods take to find an optimal policy is polynomial in the number of states and actions.
- If n and m denote the number of states and actions, a DP method is guaranteed to find an optimal policy in polynomial time even though the total number of (deterministic) policies is m^n.

(S. Maleki 2019)