Chapter 4 Dynamic Programming

Created	@November 6, 2023 10:48 AM
Tags	

Written By YEE

4.1 Policy Evaluation

我們可以以policy π 計算state-value function v_π 。在Dynamic Programming(動態規劃)的情況下我們可以將這個步驟稱之為 **Policy Evaluation**。

回顧在第三章的 $v_\pi(s)$

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s] \tag{4.3}$$

$$= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')] \tag{4.4}$$

如果environment的動態機制(environment's dynamic)為已知，則上述公式是一個長為 $|\mathcal{S}|$的 線性方程組

有著 $|\mathcal{S}|$個未知數。如果數量太多我們可以使用 **iteration solution method**(迭代法)去求得近似值。從 $v_0, v_1, v_2, ...$不斷迭代。 v_0 初始值可以是任意數，每次有效的迭代會以Bellman-equation for v_π做update rule更新。

$$v_{k+1}(s) = \mathbb{E}_\pi[R_{t+1} + \gamma v_k(S_{t+1})|S_t = s]$$

$$= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_k(s')] \tag{4.5}$$

若 $k \to \infty$ 這個 v_k 最後會收束到 v_π， 這個公式稱為**iterative policy evaluation**。

為了取得 v_k 的資訊應用在 v_{k+1} ，下一代會用取得的 s 取代上一次的 s 然後獲得新的立即獲得獎勵的期望值，我們稱上述過程為**一次的 Expected Update**，每輪迭代會將舊的

policy v_k 更新成 v_{k+1}

Expected Update有各種方法，有更新 state 的或是更新 state-action的，這些更新都是用DP動態規劃的方法做更新的

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

已知 / 參數設定：

π：待評估的 policy

θ： 代表的預測準確度下限，到達一定準確度我們就停止迭代， $\theta > 0$

$V(s)$： 初始化state value function。 For all $s \in \mathcal{S}^+$，除了 $V(terminal) = 0$ 以外，可以將其他 $V(s)$初始化為任意實數。

注意：並沒有包含更新policy部分，僅有評估部分

Code :

Loop:

$\Delta \leftarrow 0$ (誤差)

Loop for each $s \in \mathcal{S}$:

$v \leftarrow V(s)$ (舊policy state value)

$V(s) \leftarrow= \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)[r + \gamma V(s')]$ (新policy state value使用下一個State的機率分布 *)

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

until $\Delta < \theta$ (直到誤差達到一定準確度之上)

Policy Improvement

假設 π 和 π' 是deterministic policy 在所有的 state $s \in \mathcal{S}$ 中

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

那麼 policy π' 一定會比 π 更好。

$$v_{\pi'}(s) \geq v_\pi(s) \tag{4.8}$$

且若 π' 在某一個任意state大於 π (4.7)，則在該 state 下 (4.8) 也會是嚴格大於 (strict inequality) 的。

proof of the policy improvement:

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

我們目前知道，給定policy跟這個policy 的 value-function，我們可以計算policy在單一個state下的改變。自然我們會考慮到所有改變過的state並從裡面的 $q_\pi(s, a)$中選擇最優解。換句話說，利用貪婪(greedy) policy π'如下

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

$\arg\max_a$意義是遵循數值大的action走(數值打平時選擇任意action)。 Greedy policy 會向前一步看哪一步是最好的。Greedy policy 根據 (4.7) 所構築的概念(By construction)下，我們知道他一定至少會等於或是優於原policy，這個根據舊 policy 的 value function 使用 greedy 的方式使新的 policy 改善舊的 policy 稱之為 **policy improvement**.

現在我們假設這個新 policy π' 跟舊 policy π 一樣好，沒有超越舊的policy，也可以表示為 $v_\pi = v_{\pi'}$，從 (4.9) 我們得知所有的 $s \in \mathcal{S}$:

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

這個跟bellman optimality equation完全一樣，所以 $v_{\pi'}$ 一定就是 v_*，兩個 π, π' 皆必定為 optimal policy。

4.3 Policy Iteration

每當 policy π 每次以 v_π 改良後獲得 policy π' ，我們就能計算 v'_π 然後可以改良 policy 到 π'' 我們可以繼續改良下去取得下式。

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

$\xrightarrow{\text{E}}$表示 policy evaluation， $\xrightarrow{\text{I}}$表示 policy improvement，因為 MDP 只有有限種 policy ，這個會嚴格改良(strict improvement)的 policy 必定會收束在最優 policy 跟最優 value function 在有限的迭代下，這個在尋找最佳 policy 的過程可以被稱為 **policy iteration**.

$\text{Policy Iteration(using iterative policy evaluation) for estimating :}$

$\text{Step 1. Initialization}$

$V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrary for all $s \in \mathcal{S}$ 一開始先使用任意的 $V(s)$和 $\pi(s)$

$\text{Step 2. Policy Evaluation}$

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 $\Delta < \theta$ (一個測試準確度的常數，通常很小)

Step 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 $policy\ stable \leftarrow false$

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

EX 4.4

當不只一個policy同樣好時，就有可能會發生永久迴圈的情況，以此我們需要更改

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

改為

If $old\ action \neq \{a_i\}$, then $policy\ stable \leftarrow false$

$\{a_i\}$為多個同樣好之policy。

EX 4.5

改寫上述 Policy iteration 成 action value

Step 1. Initialization

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

Step 2. Policy Evaluation

Loop :

$\Delta \leftarrow 0$

Loop for each $s \in \mathcal{S}$:

$q \leftarrow Q(s,a)$

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

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

until $\Delta < \theta$ (一個測試準確度的常數，通常很小)

Step 3. Policy Improvement :

$policy\ stable \leftarrow\ true$

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

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

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

If $old\ action \neq \{a_i\}$, then $policy\ stable \leftarrow false$

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

EX 4.6

當我要在這個policy要加入 $\epsilon\text{-}soft$，或是說選擇各其他的action在各個state是 $\frac{\epsilon}{|A(s)|}$，我們需要怎麼改寫上述policy？

Step 1: π 需要加入 $\epsilon\text{-}soft$ 的method的因素下去做定義，而且 ϵ 需要被初始化。

Step 2: 任何 $\epsilon\text{-}soft$ method 都不該受 θ 的限制

Step 3: 只有在 policy 不去 explore 的情況下才去判斷說是否已經 policy-stable

4.4 Value Iteration

policy iteration 有一個很明顯的缺點，就是要做 policy evaluation前都需要**多次**掃過所有的 State，往往會拖延(protract)一代的計算時間長度。我們需要等待他收束才能知道其 policy

Value Iteration的概念是 policy improvement跟部份的policy evaluation steps。

Value Iteration:

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

Value Iteration是直接把 Bellman optimality equation當作update rule

你應該也注意到了他是用所有action-value的maxima去做計算。

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

Algorithm parameter: $\theta > 0$ (準確度)，初始化 $V(s)$ for all $s \in \mathcal{S}^+$

為任意數除了 $V(terminal) = 0$

Loop :

$\quad \Delta \leftarrow 0$

$\quad$ Loop for each $s \in \mathcal{S}$:

$\quad\quad v \leftarrow V(s)$

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

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

$\quad$ until $\Delta < \theta$ (一個測試準確度的常數，通常很小)

$\quad$ Output deterministic policy $\pi \approx \pi_*$ s.t.

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

你有發現 policy直接消失了嗎？Value iteration的概念就是我們不初始化policy了，我們的policy就是直接在 $V(s)$ 中找出 $\arg\max$ action並朝著那邊走就好。

4.5 Asynchronous(非同步) Dynamic Programming

DP有一個非常大的缺點就是他需要MDP 掃過子集中所有的state，換句話說， state 子集過於龐大會需要大量的時間成本

。

Asynchronous DP演算法是一種in place(原地演算法)，並不會依照順序更新value of state，他們會使用**現有的資訊去更新state-value**，會發生的情況是**每個state的更新次數不同**，然而某些state終究還是不能置之不理。不過好處是彈性變高了，雖然這樣並不會減少計算量，但是我們不用每次掃完state才更新資料了。它的彈性可以加速更新的進程，因為有些state也許不需要那麼常更新而其他state需要。我們有時甚至可以完全跳過他的更新因為他並不重要。有些是Chapter 8 會談到的東西。

Asynchronous DP可以較容易的整合進即時更新的資料，我們可以在正在進行MDP的當下更新資料。即時更新的資料可以讓AI做即時的判斷。

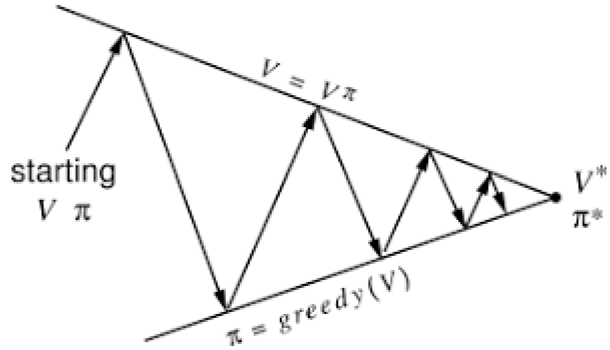
4.6 Generalized Policy Iteration

Policy iteration是由兩個互動的流程所組成，一個是現有policy(policy evaluation)，一個是遵從現在value function 的 greedy policy(policy improvement)。這兩個流程會輪流進行，在其中一個結束後另一個就會立刻開始進行。

在 Asynchronous DP這個流程是交錯的，”顆粒感”會更細(finer grain)，只要Async-DP有照顧到所有的state，他終究也會收束到最佳policy。

我們使用 **Generalized Policy Iteration** (GPI)這個術語**描述兩個流程輪流進行及互動，各自獨立顆粒度(independent of the granularity)或是其他關於這兩個流程的細節**。**幾乎所有的強化學習都可以被稱之為GPI**，任何有關明確的policy跟value functions，並且policy 隨著 value function 更新且value function 是被 policy 推進。若是兩個流程都趨於穩定的話，那我們可以說他們已經是最佳化了。Value function會穩定化主要原因為對當下policy的嚴格遵守greedy的，因此流程都會以穩定化只有在policy也同樣遵從value function的情況下會發生，也隱約證明了 (4.1) 下 Bellman Optimality Equation。

你也可以想像說這兩個流程的互動為區間限制及最終目標，以2D的形式來表示就如同下圖



每次過程會驅使value function或是policy朝著目標移動，這兩個流程聯動的過程會將其帶向optimality(最佳化)。

4.7 Efficiency of DP

DP也許沒辦法解決非常大的問題，但是相較於MDP，DP的效率是很好的，DP在最糟的情況下找出最優解的時間為state及action組成的多項式(polynominal in the number of states and actions)。

DP經常受到高維度的限制，其增長的計算量為指數型的上升，例如下五子棋時，state集合會以枝葉行向外擴散開，其state子集總數會以指數的方式擴散，但是這並不代表是DP的問題，事實上這是問題來自於題目自身。事實上DP已經比直接搜尋(direct search) 和線性規劃(linear programming)來的快了。

現今DP已經可以利用現今電腦運算速度優勢去計算MDP有百萬個state的情況了，policy iteration跟value iteration都有被廣泛地利用。紙面上說，這些方法若是在有好的初始值情況下，都遠遠比最糟情況(worst-case scenario)所花的時間來的快。

在巨大數量的State 時 Asynchronous DP 大部分情況下是比一般DP適合的，在甚至某些問題下利用電腦強大的計算及記憶力都是無解的，但是我們仍然可以利用一些方法去間接得到我們想要的結果。Asynchronous DP解決問題的速度還是比一般DP來的快上許多。