03. 다이나믹 프로그래밍으로 계획

김호철

Contents

---

1 소개

- **계획(Planning)과 학습(Learning)** : 계획(Planning)문제는 환경의 모델을 알고 있을때(Model-based) 최적의 정책을 찾는 문제, 학습(Learning)이란 환경의 모델을 모르지만 상호작용을 통해서 문제를 푸는 것 → 다이나믹 프로그래밍은 계획(Planning) 문제

1.1 다이나믹 프로그래밍

- **다이나믹** : 연속적으로 단계별로 발생되는 문제, **프로그래밍** : 최적화해서 문제를 푼다.
- 주어진 문제를 세부 문제로 분해하고, 각 세부 문제의 해를 결합해 전체 문제에 대한 최적해를 구한다.
- 복잡한 문제를 풀기 위해 하위 문제들로 나누어 해결한 뒤, 합쳐서 큰 문제를 해결하는 방식
- 겹치는 하위 문제를 배열에 저장하여 사용하는 것이 다른 알고리즘과 다름

1.2 다이나믹 프로그래밍 요구사항 2가지

- Optimal substructure : 전체의 큰 문제는 작은 문제들로 나눌 수 있어야 함
- Overlapping subproblems : 작은 문제들은 다시 나타나고, 해는 저장해뒀다가 다시 사용
- MDP는 2가지를 만족 : 벨만 방정식에서 재귀적 분해는 작은 문제들로 나뉠 수 있고, 가치 함수(value function)는 해를 저장하고 다시 사용한다.

1.3 다이나믹 프로그래밍으로 플래이닝

- MDP에 대한 모든 지식을 알고 있다고 가정하고, 계획(Planning)하는 데 사용된다
- 예측(Prediction) : 정책이 주어졌을때 기대되는 총 보상을 예측하는 것. 정책 평가(추정)
  - 입력은 MDP $< S, A, P, R, \gamma >$와 정책 π 이거나,
  - 혹은 MRP $< S, P^{\pi}, R^{\pi}, \gamma >$ 이다.
  - 출력은 **가치 함수** v_{π} 이다.
- 제어(Control) : 어떤 상태가 주어지면 기대되는 총 보상을 최대화하는 정책을 찾는 것. 어떠한 정책이 주어지면 제어를 통해서 최적 정책을 찾는다. 정책 향상(최적화)
  - 입력은 MDP $< S, A, P, R, \gamma >$
  - 출력은 **최적 가치 함수** v_*와 **최적 정책** π_* 이다.
- 주어진 정책에 대해 가치함수를 학습하는 것이 예측이고, 예측과 함께 정책을 발전시켜 최적 정책을 학습하는 것이 제어

1.4 이터레이션

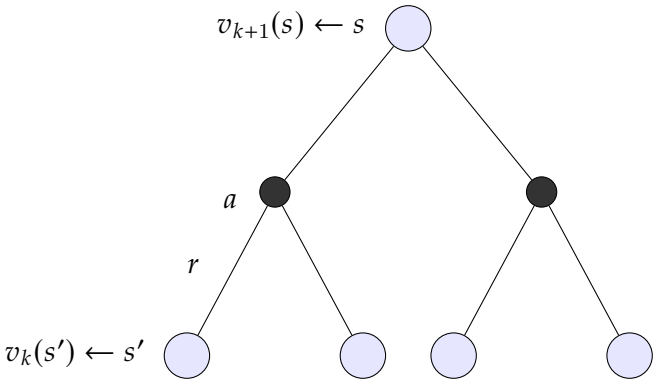
- 한번의 수행으로 해를 찾을 수 없고, 반복적으로 수행하면 최적의 해가 찾아지는 것이 벨만 최적 방정식인데, 이때의 반복을 **이터레이션**이라 한다.
- **정책 이터레이션** : 명시적인 정책이 액션을 직접 결정한다.(정책 평가+정책 개선)
- **가치 이터레이션** : 가치 함수를 보고 액션이 결정된다.

2 정책 평가(Policy Evaluation)

2.1 반복적(Iterative) 정책 평가

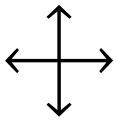
- 문제 : 주어진 정책 π를 평가
- 솔루션 : **벨만 기대 방정식**을 반복적으로 적용
- 백업(Backup) : 나눠진 하부 문제의 해를 저장하는 것
- 동기 백업과 비동기 백업이 있는데, 동기 백업은 한꺼번에 모든 상태를 업데이트 하는 것이다.

- 동기 백업을 사용
  - 각 반복 $k+1$에서, 모든 상태 $s \in S$에 대해
  - s'가 s의 성공 상태일 때, $v_k(s')$로부터 $v_{k+1}(s)$를 업데이트



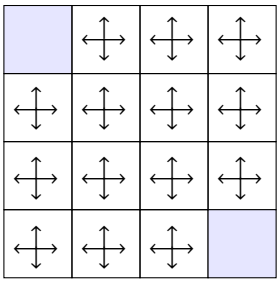
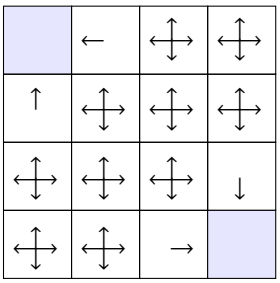
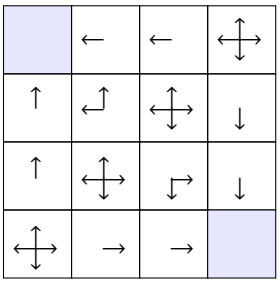
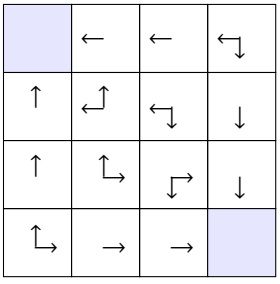
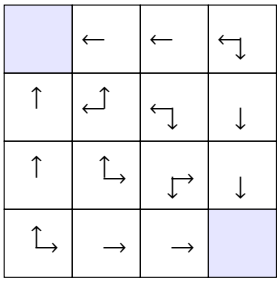
$$v_{k+1}(s) = \sum_{a \in A} \pi(a|s)(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_\pi(s')) \tag{1}$$
$$v^{k+1} = R^\pi + \gamma P^\pi v^k$$

2.2 그리드월드 예제



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

- 할인되지 않는 에피소딕 MDP(할인계수=1)
- 1부터 14번 상태는 종료되지 않는다.
- 한번의 종료 상태(그림자 칸 2개)
- 그리드를 벗어나려는 액션은 상태를 변경하지 않음
- 종료에 도달할 때 까지 각 액션마다 보상은 -1
- 에이전트는 균등(4방향 확률이 각 0.25)한 랜덤 정책을 따른다.

	랜덤 정책으로 구한 v_k	v_k에 대한 Greedy 정책
$k = 0$	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	
$k = 1$	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	
$k = 2$	0.0 -1.7 -2.0 -2.0 -1.7 -2.0 -2.0 -2.0 -2.0 -2.0 -2.0 -1.7 -2.0 -2.0 -1.7 0.0	
$k = 3$	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	 최적 정책
$k = 10$	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	 최적 정책

3 정책 이터레이션(Policy Iteration)

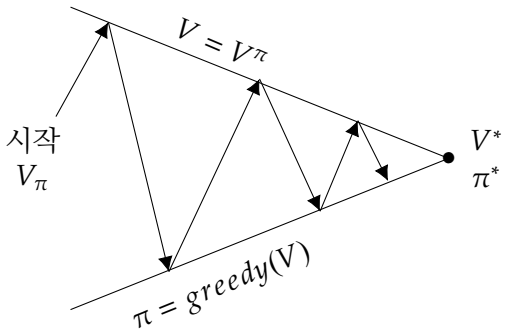
- 주어진 정책 π
  - 정책 π를 평가(Evaluate)

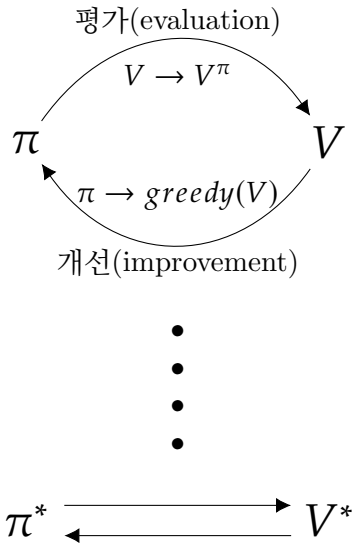
$$v_\pi = \mathbb{E}[R_{t+1} + \gamma R_{t+2} + ...|S_t = s] \tag{2}$$

  - v_π에 대해 탐욕(Greedy) 정책으로 개선(Improve) - 그리드월드에서 4방향 0.25 확률에서, 왼쪽 방향만 1의 확률로 개선

$$\pi' = greedy(v_\pi) \tag{3}$$

- 그리드월드에서 개선된 정책은 최적 정책이 된다. $\pi' = \pi^*$
- 일반적으로는 이보다 많은 개선(improvement)/평가(evaluation) 반복(iteration)이 필요
- 그러나 이 정책 반복 프로세스는 항상 최적(π^*)에 수렴한다.





정책 평가 : 반복적인 정책 평가로 v_π를 추정

정책 개선 : 탐욕적 정책 개선으로 $\pi' \geq \pi$를 생성

4 가치 이터레이션(Value Iteration)

4.1 MDP에서 가치 이터레이션

- 어떤 최적 정책은 (1)최적 첫 번째 액션 A_*와 (2)뒤따르는 후속 상태 S'의 최적 정책으로 구성된다.

정리(최적의 원칙)

정책 $\pi(a|s)$는 다음과 같은 경우에만 상태 s에서 최적의 값인 $v_\pi(s) = v_*(s)$를 달성한다.

- s가 도달할 수 있는 모든 상태 s'들에 대해서,

- π가 상태 s'들에서 최적의 값을 얻는다. $v_\pi(s') = v_*(s')$

- 하위 문제 $v_*(s')$에 대한 해를 알고 있는 경우에는
- 솔루션 $v_*(s)$는 한 단계 미리보기(lookahead)로 찾을 수 있다.

$$v_*(s) \leftarrow \max_{a \in A} R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_*(s') \tag{4}$$

- 가치 이터레이션의 아이디어는 이러한 업데이트를 반복적으로 사용하는 것이다.
- 직관(Intuition) : 최종 보상부터 시작하여 거꾸로 작업
- 반복적이고 확률적인 MDP들에 여전히 잘 동작한다.
- 최단 경로 찾기 문제

g			

문제영역

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

V_1

0	-1	-1	-1
-1	-1	-1	-1
-1	-1	-1	-1
-1	-1	-1	-1

V_2

0	-1	-2	-2
-1	-2	-2	-2
-2	-2	-2	-2
-2	-2	-2	-2

V_3

0	-1	-2	-3
-1	-2	-3	-3
-2	-3	-3	-3
-3	-3	-3	-3

V_4

0	-1	-2	-3
-1	-2	-3	-4
-2	-3	-4	-4
-3	-4	-4	-4

V_5

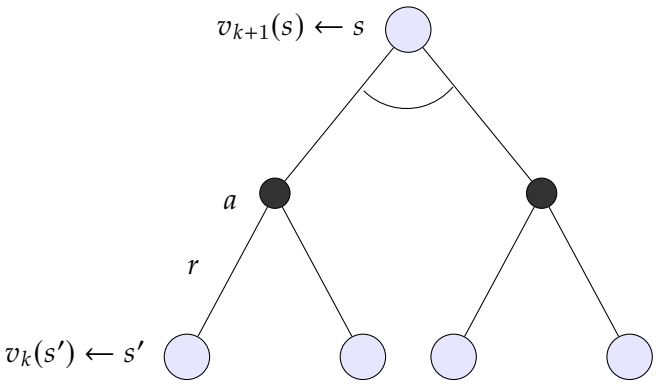
0	-1	-2	-3
-1	-2	-3	-4
-2	-3	-4	-5
-3	-4	-5	-5

V_6

0	-1	-2	-3
-1	-2	-3	-4
-2	-3	-4	-5
-3	-4	-5	-6

V_7

- 문제 : 최적 정책 π를 찾기
- 솔루션 : **벨만 최적** 백업을 반복적 적용
- $v_1 \rightarrow v_2 \rightarrow ... \rightarrow v_*$
- 동기적 백업을 사용
  - 각 반복 k+1 에서,
  - 모든 상태 $s \in S$에 대해서,
  - $v_k(s')$를 $v_{k+1}(s)$로 업데이트
- 정책 이터레이션과 달리 **명시적 정책(Policy)**이 없다.
- 중간 가치 함수는 어떤 정책과도 일치하지 않을 수 있다.



$$
\begin{aligned}
v_{k+1}(s) &= \max\left(\mathcal{R}_s^a + \gamma \sum_{s' \in S} \mathcal{P}_{ss'}^a v_k(s')\right) \\
v_{k+1} &= \max_{a \in A} \mathcal{R}^a + \gamma \mathcal{P}^a v_k
\end{aligned}
\tag{5}
$$

4.2 DP 알고리즘 정리

문제	벨만 방적식	알고리즘
예측(Prediction)	벨만 기대 방정식	이터레이티브 정책 평가
제어(Control)	벨만 기대 방정식 + 탐욕 정책 개선	정책 이터레이션
제어(Control)	벨만 최적 방정식	가치 이터레이션

- 예측은 주어진 정책에 대해 가치함수를 학습하는 것, 제어는 최적 가치 함수와 최적 정책을 학습하는 것
- 상태-가치(state-value) 함수 $v_\pi(s)$나 $v_*(s)$ 기반으로, 복잡도는 m개의 액션, n개의 상태에서 $O(mn^2)$
- 액션-가치(action-value) 함수 $q_\pi(s)$나 $q_*(s)$에도 적용할 수 있으며, 복잡도는 $O(m^2n^2)$ 으로 비교적 비용이 매우 크다.

5 다이나믹 프로그래밍의 확장

5.1 비동기 다이나믹 프로그래밍

- 비동기적(Asynchronous) 동적 프로그래밍 : 지금까지의 DP는 모두 동기적 백업을 사용한 것(모든 상태를 병렬적으로 백업). 비동기 DP는 상태들을 순서에 상관없이 개별적으로 백업한다. 선택된 상태에 대해 적절히 백업하면, 계산을 줄일수 있고, 모든 상태들이 지속적으로 선택되면 수렴이 보장된다.
- In-place dynamic programming : 동기적 가치 함수는 두개의 가치 함수 복사본(평가 완료된 이전 상태와 평가중인 현재 상태의 복사본)이 저장 되지만, In-place 가치 함수는 하나의 복사본만 저장.

- Prioritised sweeping : 벨만 오류(업데이트의 폭)가 컸던 상태들에 대해서 우선 순위를 주어, 순서적으로 업데이트 하는 방식
- Real-time dynamic programming : 상태의 영역이 매우 넓고 방문하는 곳이 한정적일때, 에이전트들을 환경에 방문하게 하고 난 뒤, 방문한 상태들을 먼저 업데이트 하게 하는 방식.

5.2 전체 너비와 샘플 백업

- 전체 너비 백업(Full-Width Backups) : 동기나 비동기 모두 모든 상태와 모든 액션들에 대해 가치함수나 정책을 모두 백업(저장)한다. DP는 중간 크기(수백만) 상태의 문제에 적합. 큰 문제의 경우 차원의 저주(상태가 하나 추가 되면 복잡도가 지수적으로 증가)
- 샘플 백업 : 보상 함수 R과 상태 전이 확률 P를 샘플링하여 사용한다. 장점은 모델 프리로 MDP에 대한 사전 지식이 필요하지 않음. 샘플링을 통해 차원의 저주를 해결, 백업 비용이 상태의 수에 상관없이 일정

5.3 근사 다이나믹 프로그래밍(Approximate Dynamic Programming)

- 가치 함수를 모두 저장해 놓지 않고, 근사(Approximate)하여 사용
- 근사 함수 $\hat{v}(s, \mathbf{w})$는 상태를 입력하면, 가중치 w로 계산하여, 가치 함수나 정책을 출력한다.