8. Planning and Learning with Tabular Methods

휘유우. 그래도 재밌다.
제목이 좀 뭐지 스쳐가는 건가 싶은데 중요한 것들이 많이 나온다. 재밌겠지?
이번 챕터에선 environment 의 model 을 필요로 하는 methods 들, 그러니까 dynamic programming 이나 heuristic search 같은 것들 말이지.
걔들과, model 없이도 쓸 수 있는 methods 들, 그러니까 Monte Carlo 나 temporal-difference methods 들의 unified view 를 만들어보자.
이들은 각각 model-based 와 model-free 로 불린다.
Model-based methods 는 planning 을 주로 사용한다.
Model-free methods 는 learning 을 주로 사용한다.
두 methods 는 차이도 분명 있지만 또 비슷하기도 하다.
특히, 두 methods 의 정수는 value functions 의 계산이다.
더욱이, 모든 methods 는 future events 를 내다보고, backed-up value 를 계산하고, 또 이걸 approximate value function 에 대한 update target 으로 쓰는 데 기초하고 있다.
책 초반에 Monte Carlo 와 temporal-difference methods 를 distinct alternatives 로 소개했고 그들을 어떻게 n-step methods 로 통합하는지 봤다.
이 챕터에서는 비슷하게 model-based 와 model-free methods 를 통합한다.
이전엔 distinct 했으니, 이제 섞어보자.

8.1 Models and Planning

environment 의 model 이란 말이지. agent 가 actions 을 하면 environment 가 어떻게 반응할지 예측하는 데 쓸 수 있는 모든 걸 말해.
주어진 state 와 action 에서, model 은 resultant next state와 next reward의 예측을 뱉어내.
model 이 stochastic 이면, 가능한 next states 와 next rewards 가 일정 발생 확률을 가질거야.
어떤 모델은 모든 possibilities 와 그 probabilities 의 description 을 produce 한다.
이런 걸 distribution models 이라 부른다.
다른 모델은 확률에 따라 샘플링 된 possibilities 중 하나만 produce 한다.
이런 걸 sample models 라고 부르지.
예를 들어, 12개 주사위의 합을 modeling 하는 걸 생각해보자.
distribution model 은 모든 possible sums 와 그 발생 확률을 줄 거고
sample model 은 이 확률 분포에 따라 나온 individual sum 을 줄 거야.
dynamic programming 에서 가정하는 모델은 distribution model 이야.
MDP 의 dynamics 의 estimates, $p(s', r \mid s, a)$ 말이다.
챕터 5의 blackjack 예시에서 사용된 모델의 종류는 sample model 이다.
distribution models 이 sample models 보다 강력하다. 걔들은 항상 samples 을 produce 하는 데 쓰일 수 있으니까.
하지만 많은 applications 에서 distribution models 보다 sample models 을 얻기가 훨씬 쉽다.
dozen dice 는 이의 간단한 예시다.
주사위 굴려서 sum 을 return 하는 프로그램 싸는 건 쉬울 거야.
하지만 모든 possible sums 과 그들의 확률을 구하는 건 아주 어렵고 에러가 나기 쉬운 일이지.

models 은 experience 를 mimic or simulate 하는 데 쓸 수도 있어.

starting state 와 action 이 주어졌을 때,
sample model 은 possible transition 을 produce 하고
distribution model 은 그들의 발생 확률로 weight 된 all possible transitions 을 generate 한다.

starting state 와 policy 가 주어졌을 때,
sample model 은 entire episode 를 produce하고
distribution model 은 all possible episodes 와 그 확률을 generate 한다.
두 cases 모두, model 이 environment 를 simulate 하고 simulated experience 를 produce 하는 데 쓰였다고 한다.

planning 이라는 단어는 다양하게 쓰인다.
우린 이걸 다음과 같이 쓰겠다.
planning:
model 을 input 으로 넣고 modeled environment 와 상호작용하는 policy 를
produce 하거나 improve 하는 모든 computational process.
결국 model로 policy 만드는 애?

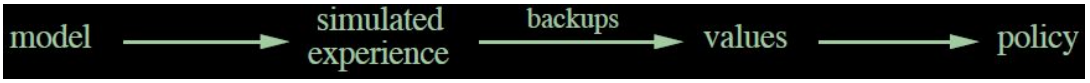
인공지능에서, 이 정의에 따르면 planning 에 접근하는 두 가지 방법이 있다.

1. state-space planning: 이 책에서 사용할 접근법을 포함하는 앤데, 주로 optimal policy 나 goal 로 가는 optimal path 를 위한 state space를 찾는 search 라고 볼 수 있다.
   actions 은 state 간 transitions 을 일으키고, value functions 은 states 마다 계산된다.

2. plan-space planning: plans 의 공간을 search 하는 것. operator 가 한 plan 을 다른 plan 으로 변환시키고, value functions은(있다면) plans 의 공간에 따라 정의된다.
   evolutionary methods 와 partial-order planning 을 포함한다.
   얘들은 stochastic sequential decision problems 에 부적합. 그니까 강화학습엔 안맞겠지.

우리가 이 챕터에서 소개하는 통합된 관점은,
모든 state-space planning methods 가 이 책에서 나온 learning methods 에도 있는 공통의 구조를 공유하고 있다는 거다.
챕터 나머지 부분에서 이런 관점을 develop 할 거다.
하지만 두 가지 basic ideas 가 있다.

1. all state-space planning methods 가 policy 를 개선하기 위한 key intermediate step 으로 value functions 을 계산하는 걸 포함한다.
2. simulated experience 에 적용된 updates 나 backup operations 으로 value functions 을 계산한다.



이런 구조.

Dynamic programming methods 는 이 구조에 딱 맞다.
states 의 공간을 sweep 하고 possible transitions 의 분포를 각 state 에 대해 생성한다.
각 distribution 으로 backed-up value (update target) 를 계산하고 state 의 estimated value 를 update 한다.

이 챕터에서 우리는 여러 다른 state-space planning methods 도 이 구조에 부합한다고 주장할 거야. 개별 방법은 걔들이 update 하는 종류, 수행 순서, backed-up 정보가 유지되는 기간만 달라.

이 관점에서 planning methods 를 보는 건 learning methods 와의 관계를 강조한다.
learning 과 planning methods 의 정수는 backing-up update operations 으로 value functions 의 estimation 이다.
차이점은
planning 이 model 에 의해 생성된 simulated experience 를 쓰는 데 반해,
learning methods 는 environment 에 의해 생성된 real experience 를 사용한다는 것이다.
당연히 이 차이는 다른 차이로 이끈다.
예를 들어 performance 를 평가하는 방법이나, experience 가 flexibly 생성되는 방법들.
하지만 common structure 는 많은 ideas 와 algorithms 들이 planning 과 learning 을 왔다갔다할 수 있다는 뜻이다.
특히, 많은 경우에 learning algorithm 이 planning method 의 주요 update step 을 대체할 수 있다.
learning methods 는 input 으로 experience 만 필요로 하고, 많은 경우에 real experience 뿐만 아니라 simulated experience 에도 적용될 수 있다.
아래 박스는 one-step tabular Q-learning 과 sample model 에서의 random samples 에 기반한 planning method 의 간단한 예시를 보여준다.

Random-sample one-step tabular Q-planning

Loop forever:

1. Select a state, $S \in \mathcal{S}$, and an action, $A \in \mathcal{A}(s)$, at random
2. Send S, A to a sample model, and obtain
   a sample next reward, R, and a sample next state, S'
3. Apply one-step tabular Q-learning to S, A, R, S':
   $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$

이 방법, 그러니까 random-sample one-step tabular Q-planning 은,
model 에 대한 optimal policy 로 converge 한다.
real environment 에 대해 one-step tabular Q-learning 이 optimal policy 로 converge 하는 같은 조건 하에서.
각 state-action pair 가 step 1 에서 무한번 선택돼야하고 α 가 시간에 따라 적절히 감소되어야 해.

planning 과 learning methods 를 통합하는 관점에 더해,
이 챕터의 두 번째 주제는 small, incremental steps 에서 planning 의 장점이다.
이러면 planning 이 계산 낭비 거의 없이 언제든지 중단하거나 방향을 바꿀 수 있다.
요게 acting 이랑 model의 learning과 planning 을 효율적으로 섞어주는 핵심이더라.

very small steps 에서 planning 은 문제가 정확히 풀기 어려울만큼 너무 크다면,
순수 planning 문제에도 가장 효율적인 접근법일 수 있다.

8.2 Dyna: Integrating Planning, Acting, and Learning

planning 이 on-line 으로 되면, environment 와 상호작용하는 동안,
많은 흥미로운 이슈들이 발생한다.

interaction 에서 얻은 새로운 정보는 model 을 바꿀거고 따라서 planning 과 interact 한다.
현재, 또는 가까운 미래에 기대되는 states 나 decisions 에 어떤 방식으로든 planning process를 customize 하는 게 좋을 수 있다.
만약 decision making 과 model learning 이 둘 다 computation-intensive 프로세스면, 사용가능한 computational resources 을 둘로 나눠서 써야할 수 있다.
이런 이슈들을 탐색하기 위해,
이 섹션에서 Dyna-Q를 소개한다.
on-line planning agent 에서 필요한 major functions 을 통합한 간단한 아키텍쳐다.
Dyna-Q에서 나오는 각 함수는 간단하고 trivial 한 form이다.
이어지는 sections에서 각 함수를 achieve 하는 몇 가지 alternate 한 방법과 그들 사이의 trade-offs 에 대해 설명한다.
지금은, idea만 보여주고 너의 직관을 자극하려고 해.

planning agent 내에서, real experience 는 최소 두 가지 역할을 해.
model 을 improve 하는 데 쓰인다.(real environment 에 더 accurately match 시키려고)
value function과 policy 를 직접적으로 improve 하는 데 쓰인다.(우리가 이때까지 봤던 강화학습 methods 이용해서)
앞의 경우를 model-learning 이라고 하고
뒤의 경우를 direct reinforcement learning (direct RL) 이라고 한다.
experience, model, values, policy 사이의 possible relationships 은 Fig 8.1 에 요약돼 있다.

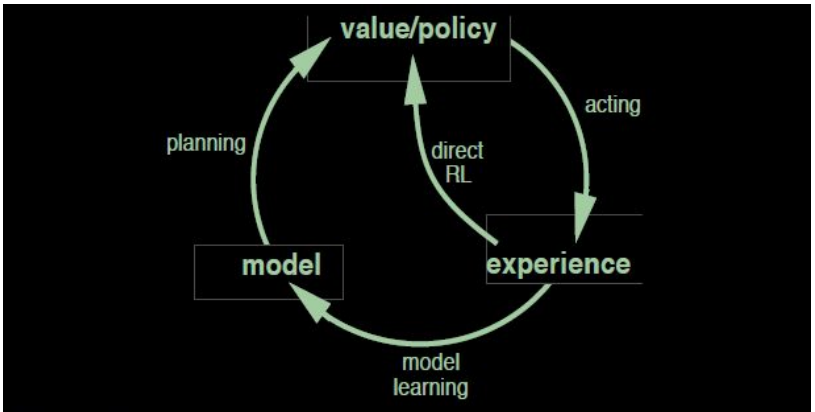


Figure 8.1: Relationships among learning, planning, and acting.

각 화살표는 influence와 예상되는 improvement 의 relationship 을 보여준다.
experience 가 어떻게 value functions 과 policies 를 improve 하는지 봐.
directly 하게도 하고, model 을 통해 indirect 하게 하기도 해.
indirect 하게 하는 게 planning 에 관련된 indirect reinforcement learning 이다.

direct, indirect methods 둘 다 자진모리 장단이 있다.
indirect methods 가 제한된 양의 experience 를 최대한 활용한다. 그래서 더 적은 environmental interactions 으로 better policy 를 달성해.
반면, direct methods 는 훨씬 더 간단하고 model 의 design 의 bias 에 영향을 받지 않는다.

indirect methods 가 direct 보다 항상 낫다고 주장하는 사람도 있고
direct methods 가 사람과 동물이 학습하는 방법이라 주장하는 사람도 있다.

우리 관점은 두 측면의 유사성에 좀 더 초점을 맞춰 통찰력을 얻어보자는 거다.
예를 들어, dynamic programming 과 temporal-difference methods 사이의 깊은 관련성을 강조해 왔지. 하나는 planning 이고 model-free learning 인데도 말이야.

Dyna-Q 는 Fig 8.1 의 모든 프로세스를 담고 있어.
planning, acting, model-learning, direct RL.
전부 연속적으로 일어난다.
planning method 는 random-sample one-step tabular Q-planning method 이다.
direct RL method 는 one-step tabular Q-learning 이다.
model-learning method 도 table-based 이고 environment 가 deterministic 함을 가정한다.
각 transition $S_t, A_t \rightarrow R_{t+1}, S_{t+1}$ 이후, model 은 S_t, A_t 에 대한 table entry 에 R_{t+1}, S_{t+1} 이 deterministically 하게 따르는 prediction 을 기록한다.
그래서 만약 모델이 이전에 경험한 state-action pair 로 쿼리를 날리면,
마지막으로 관측된 next state 와 next reward 를 prediction 으로 리턴한다.
planning 동안, Q-planning 알고리즘은 이전에 경험했던 state-action pairs 에서만 랜덤샘플링 한다. (Step 1에서)
그래서 model 은 정보가 없는 pair 는 쿼리되지 않는다.

Dyna agents 의 전체적인 아키텍처는 Fig 8.2 에 있다.
Dyna-Q 알고리즘이 한 예고.

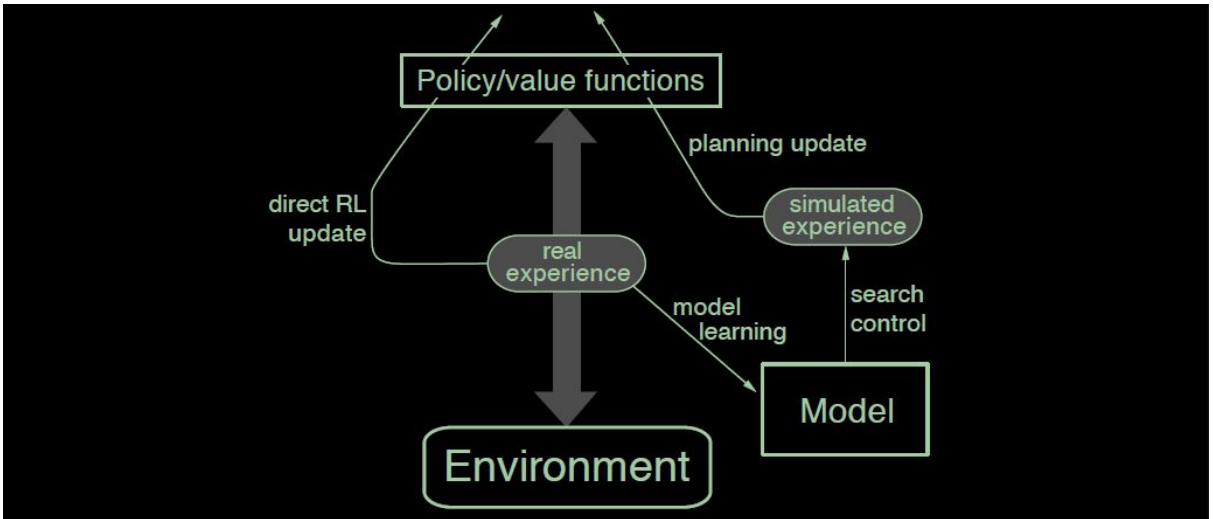


Figure 8.2: The general Dyna Architecture. Real experience, passing back and forth between the environment and the policy, affects policy and value functions in much the same way as does simulated experience generated by the model of the environment.

Fig 8.2: 일반적인 Dyna Architecture. Real experience 가 environment와 policy 사이를 오가며 policy 와 value functions 에 영향을 미친다. environment의 model 에 의해 생성된 simulated experience 와 거의 동일한 방식으로.

central column 은 agent 와 environment 사이의 기본적인 interaction 을 나타낸다. real experience의 궤적을 만들어낸다.
그림의 왼쪽 화살표는 direct RL operating 을 나타낸다. real experience 로부터 value function 과 policy 를 improve 한다.
오른쪽은 model-based processes 이다.
model 은 real experience 로부터 학습하고 simulated experience를 낳는다.
search control 이라는 용어를 사용하는데, 이건 무슨 뜻이냐면,
프로세스인데, model 로 생성한 simulated experience 에 대한 starting states 와 actions 을 고르는 프로세스를 말해.

마지막으로, 마치 실제로 일어난 것처럼 simulated experiences 에 강화학습 방법을 적용시켜 planning 이 달성된다.
통상적으로, Dyna-Q에서처럼, 같은 강화학습 방법이
real experience 를 통한 learning 과
simulated experience 로부터의 planning
모두에 쓰인다.
그래서 강화학습 방법은 learning 과 planning 의 “final common path” 이다.
learning 과 planning 은 깊이 integrated 돼 있다. 거의 같은 machinery 를 공유한다는 면에서.
experience 의 source만 다르고.

개념적으로, planning, acting, model-learning, direct RL 은
Dyna-Q에서 동시에 병렬적으로 발생한다.
하지만 구체적으로 serial computer 에 적용하기 위해선 time step 내에서 발생하는 순서를 fully specify 한다.
Dyna-Q 에서, acting, model-learning, direct RL 프로세스들은 계산량이 적고, 시간상으로 조금만 소모한다고 가정한다.
각 step에서 remaining time 은 planning process 에 사용할 수 있다.
그건 계산량이 많나봐. computation-intensive 래.
acting, model-learning, direct RL 이후, 각 step 에서
Q-planning algorithm 의 n iterations (Steps 1-3) 을 완료하는 데 시간이 있다고 가정하자.
아래 박스의 Dyna-Q 알고리즘의 수도코드에서,
Model(s, a) 는 (s, a) 에 대한 predicted next state and reward의 내용을 나타낸다.
Direct RL, model-learning, planning 은 d, e, f 에서 각각 적용된다.
e 와 f 가 빠지면, one-step tabular Q-learning 이 된다.

Tabular Dyna-Q

Initialize $Q(s,a)$ and $Model(s,a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$
Loop forever:
- (a) $S \leftarrow$ current (nonterminal) state
- (b) $A \leftarrow \epsilon$-greedy(S,Q)
- (c) Take action A; observe resultant reward, R, and state, S'
- (d) $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
- (e) $Model(S,A) \leftarrow R, S'$ (assuming deterministic environment)
- (f) Loop repeat n times:
  - $S \leftarrow$ random previously observed state
  - $A \leftarrow$ random action previously taken in S
  - $R, S' \leftarrow Model(S,A)$
  - $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$

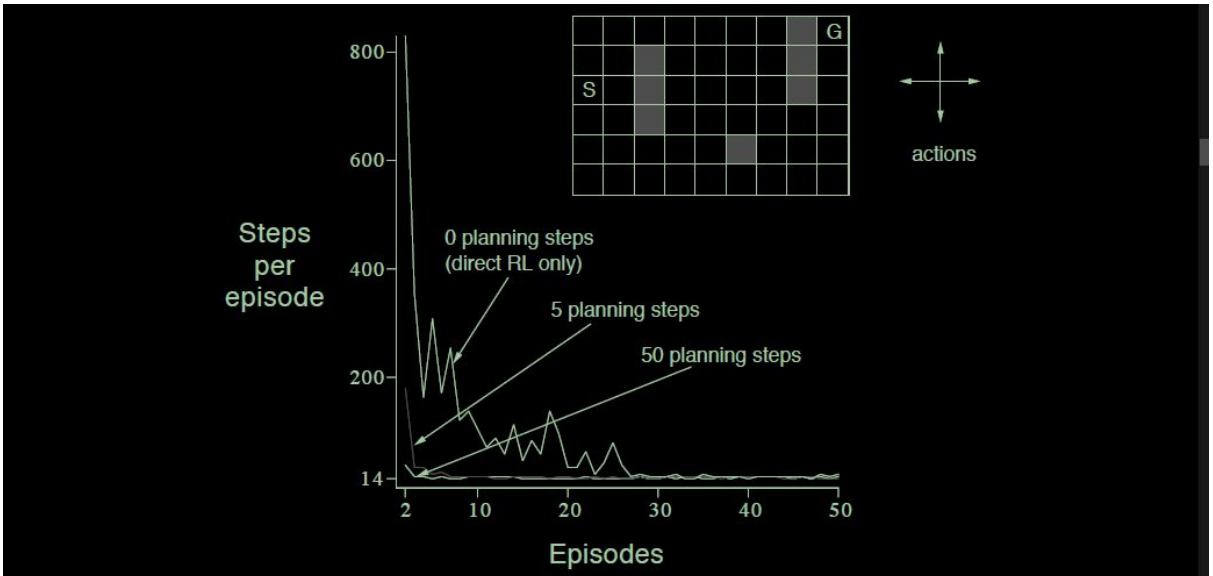


Figure 8.3: A simple maze (inset) and the average learning curves for Dyna-Q agents varying in their number of planning steps (n) per real step. The task is to travel from S to G as quickly as possible.

(Fig 8.3) real step 당 planning steps (n) 수에 따라 달라지는 Dyna-Q agents 의 average learning curve.

(Example 8.1) Dyna Maze
Fig 8.3 을 보라. 47 states 각각에서 상하좌우 네 가지 actions 이 있고 deterministically 하게 이웃 states 로 간다. 장애물에 부딪히거나 밖으로 나가면 그 자리에 머무른다.
모든 transitions 에서 reward는 0이고 goal 에서 +1.
G 에 도달하면 S에서 새로운 episode 를 시작한다.
γ = 0.95 인 discounted episodic task 이다.

Fig 8.3 의 main part 는 Dyna-Q 가 적용됐을 때의 average learning curve 다.
initial action values 는 0이고, step-size parameter α = 0.1, exploration parameter ε = 0.1 이다.
actions 은 greedy 하게 선택하게. tie 면 random 하게 break.
agents 는 real step 당 수행하는 planning steps n 수를 다양하게 했다.
각 n에 대해, curves 는 각 episode에서 goal 에 도달하기까지 agent 가 밟은 steps 수인데, 30번의 실험 반복을 평균했다.
각 반복에서 random number generator 의 initial seed 는 고정.
이것때문에 첫 번째 episode 는 n 의 모든 값에 대해 정확히 같았다.
첫 번째 episode 이후, n의 모든 값에 대해 성능은 향상됐다.
더 큰 값에 대해 훨씬 더 빨랐다?
n = 0 인 nonplanning agent 는 direct RL(one-step tabular Q-learning) 만 쓴다.
이게 지금까지 중에 가장 느린 agent 이다. α, ε 가 여기에 최적화 돼 있어도 말이다.
nonplanning agent 는 (ε-) optimal performance 까지 가기에 25 episodes 가 걸렸고
n=5 인 agent 는 5 episodes, n=50 일 땐 3 episodes 만 걸렸다.

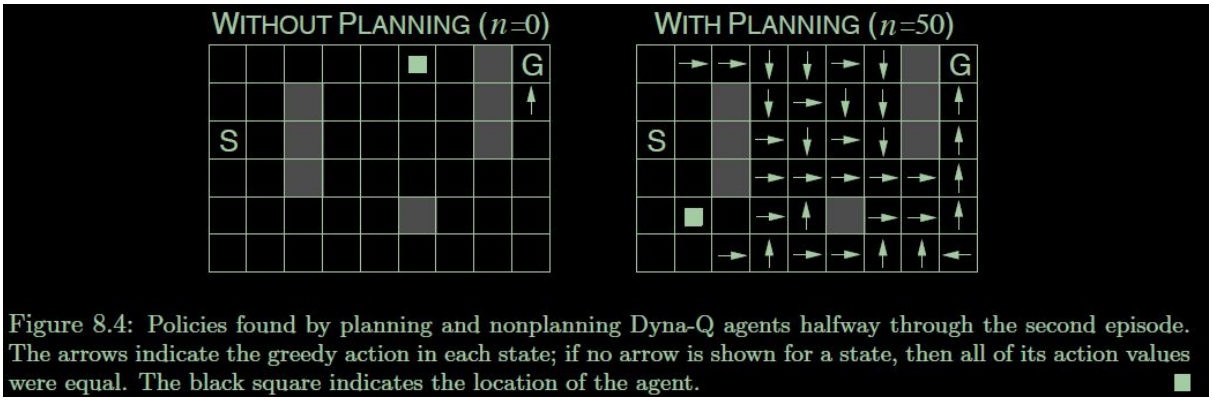


Figure 8.4: Policies found by planning and nonplanning Dyna-Q agents halfway through the second episode. The arrows indicate the greedy action in each state; if no arrow is shown for a state, then all of its action values were equal. The black square indicates the location of the agent.

Figure 8.4: 두 번째 episode 중간에 n=0, n=50 인 agent 가 찾은 policies 다.
화살표는 각 state 에서 greedy action 을 가리킨다. 화살표가 없으면 모든 action values 가 같다. 네모는 agent 위치.

Fig 8.4는 왜 planning agents 가 solution 을 훨씬 더 빨리 찾는지 보여준다.
planning 없이는, 각 episode 에서 policy에 하나의 additional step 만 추가한다.
그래서 지금까지 마지막 스텝 하나만 배운거.
planning 을 쓰면, 첫 번째 episode 에선 한 step 만 배우지만,
두 번째 episode 동안 policy 가 엄청 학습돼서 episode 가 끝날 때쯤엔 start state 까지 거의 연결된다.
이 policy 는 agent 가 start state 근처에서 돌아다니는 동안 planning process 로 만들어진다.
세 번째 episode 의 끝에는 완전한 optimal policy 가 찾아질 거고 perfect performance 가 나온다.

Dyna-Q에서, learning 과 planning 은 정확히 같은 알고리즘으로 완성된다.
learning 은 real experience 로, planning 은 simulated experience로.
planning이 incrementally 하게 진행되기 때문에 planning 과 acting 을 섞는 게 trivial하다?
둘 다 할 수 있는 최대한 빨리 진행한다.
agent 는 항상 reactive, deliberative 하다.
latest sensory 정보에 즉시 반응하지만, 항상 background 에서 planning 한다.
model-learning process도 background 에서 진행 중이다.
새로운 정보가 얻어지면, model 은 reality 에 더 잘 맞게 update 된다.
model 이 바뀜에 따라, ongoing planning process는 새로운 model 에 맞춰 다르게 행동하는 방법을 gradually 계산할 거다.

(Exercise 8.1)

8.3 When the Model Is Wrong

maze example 에서, model 내에서의 변화는 상대적으로 modest 했다.
model 은 empty 로 시작해서 정확히 맞는 정보로만 채워졌다.
일반적으로, 그렇게 운이 좋길 바랄 수 없다.
models 은 정확하지 않을거다.
environment 는 stochastic 이고 제한된 수의 samples 만이 관측됐으니까.
불완전게 일반화된 function approximation 을 사용해 학습됐기 때문일 수도 있고.

아니면 environment 가 바뀌고 새로운 행동이 아직 관측되지 않았기 때문일 수도 있다.
model 이 incorrect 할 때, planning process 는 suboptimal policy 를 계산할 가능성이 높다.

어떤 경우엔, planning 으로 계산된 suboptimal policy 가 modeling error 를 빨리 발견하고 고친다. 이건 model 이 optimistic 일 때 발생하는 편이다.
optimistic 이 무슨 의미로 한 말이냐면,
실제로 가능한 것보다 더 큰 reward나 더 좋은 state transitions 을 predicting 한다는 의미로 한 말이다. 그럼 일어날 수 없는 일? 수준? 을 상정한다는 말인가?
planned policy 는 이런 기회를 exploit 하려고 시도하고,
그렇게 함으로써 그들이 존재하지 않는다는 걸 알게 된다.

(Example 8.2) Blocking Maze
Fig 8.5 에서 이 상대적으로 minor 한 종류의 modeling error 와 거기서 recovery 하는 것을 보여주고 있다.
처음에, 좌상단 그림을 보면 장애물의 오른쪽에 길이 있는거 보이지?
1000 steps 후에, short path 가 막히고 장애물 왼쪽에 longer path 가 생겨.
그래프를 보면 cumulative reward 가 있어.
Dyna-Q 와 Dyna-Q+ 의 그래프지.
1000 steps 내에서 short path 잘 찾았고,
environment 가 바뀌면, 그래프가 flat 하지.
장애물 뒤에서 방황하면서 reward 를 얻지 못하고 있어.
하지만 이내 new opening 과 new optimal behavior 를 찾을 수 있었지.

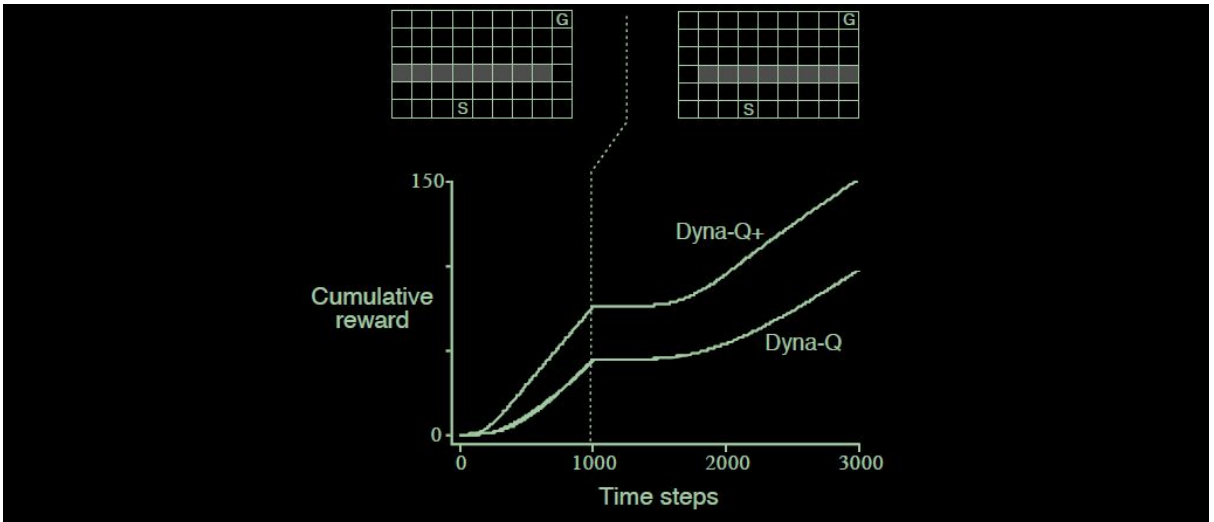


Figure 8.5: Average performance of Dyna agents on a blocking task. The left environment was used for the first 1000 steps, the right environment for the rest. Dyna-Q+ is Dyna-Q with an exploration bonus that encourages exploration.

environment 가 이전보다 더 좋아지면 더 큰 어려움이 발생한다.
하지만 이전의 correct policy 는 improvement 를 reveal 하지 않는다? 뭔 말?
이런 경우들에서 modeling error 는 긴 시간동안 detect 되지 못할거다.
다음 예에서 보자.

(Example 8.3) Shortcut Maze
environment 가 바껴서 생기는 이런 문제들이 Fig 8.6에 나와있다.
3000 steps 후에 왼쪽에서 오른쪽으로 바뀐다.

shorter path 가 만들어지는거지. longer path 방해 없이.
그래프를 보면 regular Dyna-Q는 shortcut 으로 안바꿔.
아예 존재 자체도 모를걸.
model 은 저기 shortcut 이 없으니, 더 planned 될수록 오른쪽으로 갈 일이 없어지지.
ε-greedy policy 를 쓴다해도, agent 가 shortcut 을 발견하기 위해 그렇게 많이 exploratory actions 을 할 가능성은 없지.

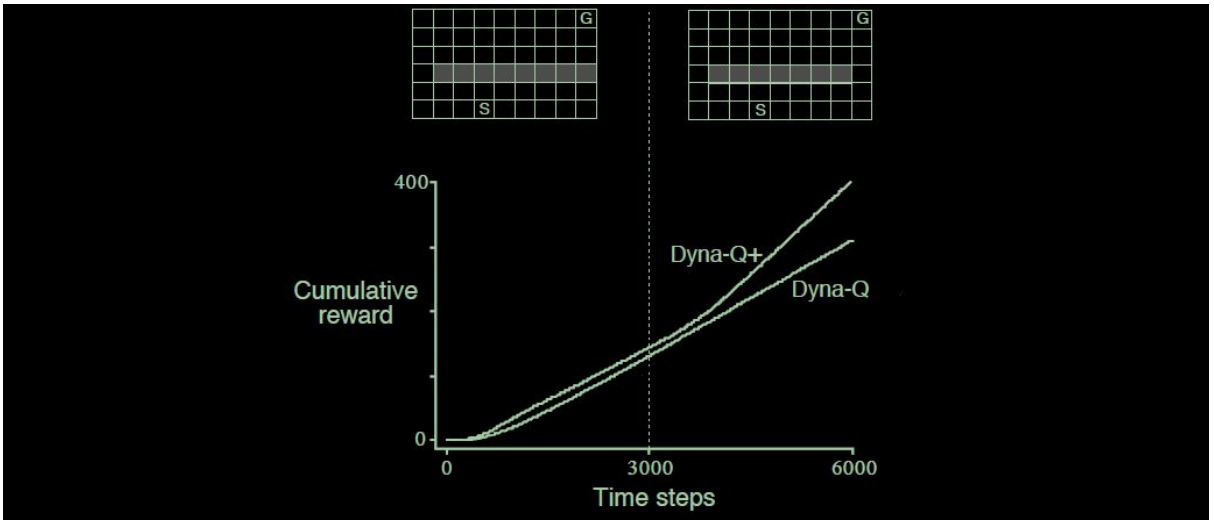


Figure 8.6: Average performance of Dyna agents on a shortcut task. The left environment was used for the first 3000 steps, the right environment for the rest.

여기서 일반적인 문제는 exploration 과 exploitation conflict 의 또다른 버전이라는 거다.
planning context 에서, exploration 은 model 을 improve 하는 actions 을 시도하는 거다.
반면 exploitation 은 현재 model 에서 optimal way 에 따라 행동하는 걸 말해.
우린 agent 가 explore 해서 environment 의 변화를 감지하길 원하지만,
그렇다고 성능이 너무 떨어져서도 안 돼.
이전에 봤던 exploration/exploitation conflict 에서처럼, 완벽하고 실용적인 solution 은 없지만 간단한 heuristic 은 종종 효과가 있어.

shortcut maze 를 푼 Dyna-Q+ agent 는 그런 heuristic 중 하나를 썼어.
이 agent 는 environment 와의 real interaction 에서 마지막으로 try 된 이후에
얼마나 많은 time steps 이 지나갔는지
각 state-action pair 를 추적한다.
더 많은 시간이 지났을수록, 이 pair 의 dynamics 가 바꼈을 확률, model 이 incorrect 할 확률이 커진다.
long-untried actions 를 test 하는 behavior 를 encourage 하기 위해,
이런 actions 이 포함된 simulated experiences 에 special “bonus reward” 를 준다.
특히, 만약 transition 의 modeled reward 가 r 이고, transition 이 τ time steps 동안 try 돼지 않았다면, planning updates 는 transition 이 $r + \kappa\sqrt{\tau}$ (for small κ) 의 reward 를 produce 하는 것처럼 수행된다.
이게 agent 가 모든 accessible state transitions 를 계속 test 하도록 해주고
이런 tests 를 수행하기 위해 긴 sequence의 actions 을 찾도록 해주기도 한다.
당연히 모든 이 testing 은 cost 가 들지만
이런 종류의 computational curiosity 는 추가 exploration 의 가치가 있다.

(Exercise 8.2)

(Exercise 8.3)
(Exercise 8.4)

8.4 Prioritized Sweeping

이전 sections 의 Dyna agents 에선,
simulated transitions 은 이전에 경험했던 모든 pairs 에서 uniformly random 하게 선택된 state-action pairs 로 시작된다.
하지만 uniform selection 은 대개 best 가 아니지.
simulated transitions 과 updates 가 특정 state-action pairs 에 초점을 맞추면 planning 은 훨씬 더 효율적일 수 있다.
예를 들어, Fig 8.4 에서 두 번째 episode 동안 무슨 일이 있었는지 생각해 봐라.
두 번째 episode 의 시작에서, goal 로 이끄는 state-action pair 만 positive value 를 가졌다.
다른 pairs 의 values 는 여전히 0이다.
이건 거의 모든 transitions 에 따라 updates 를 수행하는 건 의미가 없다는 걸 뜻한다.
agent 를 zero-valued state 에서 다른 state 로 데려가기 때문에 업데이트가 효과가 없다.
goal 바로 직전의 state 로의 transition 이나 goal 로부터의 update 만이 어떤 value 든 바꿀거다.
simulated transitions 이 uniformly generate 되면, 이런 useful update 중 하나에 우연히 걸리기 전에 많은 wasteful updates 가 있을거야.
planning 이 진행될수록, useful updates 의 영역이 커진다.
하지만 most good 을 할 수 있는 곳에 집중할 때보다 훨씬 덜 효율적이다.
우리의 목표인 훨씬 더 큰 문제에서는, states 의 수가 엄청 커서 unfocused search 는 아주 비효율적이다.

이 예시는 goal states 에서 backward 로 보면 search 가 유용하게 focus를 맞출 수 있음을 보여준다.
당연히, 우린 “goal state” 라는 idea 에 특정 methods 를 사용하고 싶진 않다.
우린 general reward functions 에 적합한 methods 를 원한다.
goal states 는 단지 special case일 뿐이다. 직관을 자극하는거지.
일반적으로, goal states 뿐만 아니라 value 가 바뀐 모든 state 에서 work back 하길 바란다.

maze 예제에서 goal 을 발견하기 전 상황과 같이,
주어진 model 에서 values 가 initially correct 하다 가정하자.
이제 agent 가 environment 에서 변화를 발견하고 한 state 의 estimated value 를 바꿨다 가정하자. up 이든 down 이든.
통상적으로, 이건 다른 많은 states 의 values 도 바껴야 한다는 걸 의미한다.
하지만 useful 한 one-step updates 는 value 가 바뀐 state 로 가는 actions 이다.
이들 actions 의 values 가 update 되면, predecessor states 의 values 가 차례로 바뀔 수 있다.
그렇게 되면, 거기로 가는 actions 이 update 되어야 하고,
그러면 그들의 predecessor states 가 바뀔 수 있겠지.

이런 방식으로 value 가 바뀐 임의의 states 에서 work backward 할 수 있다.
useful updates 를 하거나 propagation 을 끝내거나 하는거지.
이 일반적인 idea를 planning computations의 *backward focusing* 이라고 부른다.

useful updates 의 경계가 backward 로 propagates 함에 따라, 종종 빠르게 커지면서

usefully update 될 수 있는 많은 state-action pairs 를 생성한다.
하지만 이런 애들이 다 같이 useful 한 건 아니다.
어떤 states 의 values 는 많이 바꼈을 수도 있고 다른 애들은 거의 안바꼈을 수 있어.
많이 바뀐 애들의 predecessor pairs 는 또한 많이 바뀔 가능성이 높다.

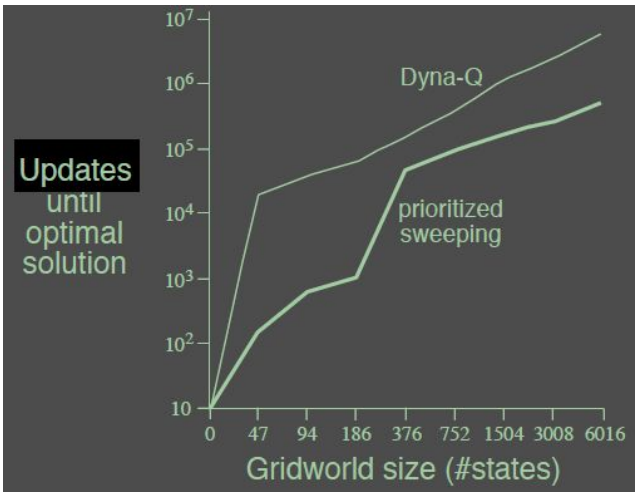
stochastic environment 에서,
estimate 된 transition probabilities 의 variation 은
changes 의 크기의 variation 과
어느 pairs 가 update 되어야 하는지의 urgency 에 대한 variation 에 또한 기여한다.
그들의 urgency의 measure 에 따라 updates 를 prioritize 하고
priority에 따라 수행하는 건 자연스럽지.
이게 *prioritized sweeping* 에 담긴 idea다.

queue 는, update 될 때 estimated value 가 크게 바뀌지 않는 모든 state-action pair 에 대해 유지된다.
change 의 size 에 따라 prioritize 된다.
queue 의 top pair 가 update 되면, 그 predecessor pairs 각각에 미칠 effect가 계산된다.
effect 가 어떤 threshold보다 크면,
pair 는 새로운 priority 로 queue 에 들어간다.
만약 queue 에 pair 의 이전 entry 가 있을 때 insertion 하면 priority 가 높은 entry 만 queue 에 남는다.
이런 방식으로 changes 의 effects 가 backward 로 전파된다. quiescence(중단) 될 때까지.
deterministic environments 에서의 full algorithm 은 아래 박스.

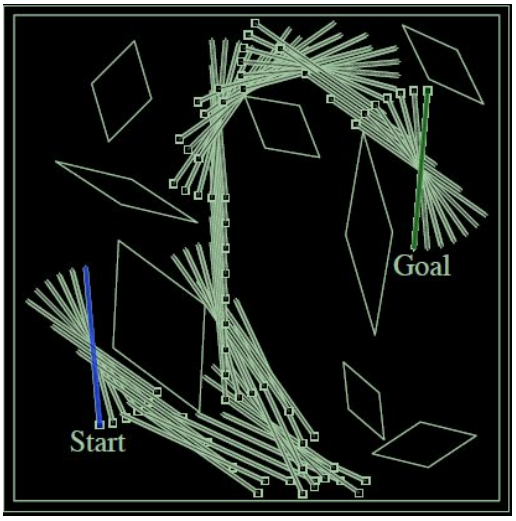
Prioritized sweeping for a deterministic environment

Initialize $Q(s,a)$, $Model(s,a)$, for all s, a, and $PQueue$ to empty
Loop forever:
- (a) $S \leftarrow$ current (nonterminal) state
- (b) $A \leftarrow policy(S,Q)$
- (c) Take action A; observe resultant reward, R, and state, S'
- (d) $Model(S,A) \leftarrow R, S'$
- (e) $P \leftarrow |R + \gamma \max_a Q(S',a) - Q(S,A)|$.
- (f) if $P > \theta$, then insert S, A into $PQueue$ with priority P
- (g) Loop repeat n times, while $PQueue$ is not empty:
  - $S, A \leftarrow first(PQueue)$
  - $R, S' \leftarrow Model(S,A)$
  - $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$
  - Loop for all $\bar{S}, \bar{A}$ predicted to lead to S:
    - $\bar{R} \leftarrow$ predicted reward for $\bar{S}, \bar{A}, S$
    - $P \leftarrow |\bar{R} + \gamma \max_a Q(S,a) - Q(\bar{S},\bar{A})|$.
    - if $P > \theta$ then insert $\bar{S}, \bar{A}$ into $PQueue$ with priority P

(Example 8.4) Prioritized Sweeping on Mazes
Prioritized Sweeping 은 미로 문제에서 optimal solutions 이 발견되는 속도를 5-10 배까지 증가시킨다. PS 가 Dyna-Q보다 좋네. Dyna-Q는 unprioritized.
두 systems 다 environmental interaction 당 최대 n = 5 인 updates 를 했다.



(Example 8.5) Rod Maneuvering
제한된 사각형 공간에 이상하게 배치된 장애물들이 있다.
여기서 최소한의 step으로 막대를 goal 로 이동시킨다.
막대는 막대의 긴 축 방향으로, 또는 거기에 수직으로 움직일 수 있고,
막대 가운데를 중심으로 회전할 수도 있다.
각 step 에서 움직일 수 있는 거리는 work space 의 1/20 정도고
rotation 은 10도.
변환은 deterministic 하고 20 x 20 positions 으로 quantize 된다.



prioritized sweeping 으로 찾은 shortest solution.
이 문제는 deterministic 이지만, 4 actions 과 14,400 potential states 가 있다.
너무 커서 unprioritized methods 로는 풀기 어렵다.

prioritized sweeping 에서 stochastic environments 로의 확장은 straightforward 하다.
model 은 각 state-action pair 를 경험한 횟수와 다음 states 의 수를 기록함으로써 유지된다.
그러면 각 pair 를 sample update 가 아니라 expected update 로 update 하는 게 자연스럽다.
모든 possible next states 와 그 발생 확률을 고려하는 expected update로 말이다.

prioritized sweeping 은 planning efficiency를 향상시키기 위해 계산을 나누는 한 방법일 뿐이다.

prioritized sweeping 의 한계 중 하나는 expected updates 를 쓴다는 거다.
그럼 stochastic environments 에서는 low-probability transitions 에 많은 계산 낭비를 할 수 있지.
다음 section 에서 보여줄 것처럼,
sample updates 는 많은 경우에
sampling 에 의한 variance 에도 불구하고, 더 적은 계산으로 true value function 에 더 가깝다.
sample updates 는 전체 backing-up computation 을 individual transitions 에 대응하는 조각으로 작게 쪼개서 가장 큰 영향을 미칠 조각에 초점을 맞출 수 있기 때문에 이길 수 있다.
이 idea 는 sutton 이 도입한 small backups 라는 논리적 한계에 도달했다?
얘들은 sample update 처럼 single transition 에 따른 updates 이다.
하지만 expected update 에서처럼 sampling 없이 transition 의 확률에 기반한다.
small updates 가 수행되는 순서를 잘 선택하면
prioritized sweeping 으로 가능한 것 이상으로 planning efficiency 를 크게 향상시킬 수 있다.

이 챕터에서 우리는 모든 종류의 state-space planning 이 value updates 의 sequences 로 볼 수 있다는 걸 제안했다. update 의 유형, expected 인지 sample 인지, large or small 인지, update 의 수행 순서 이런 것들만 다를 뿐이다.
이 section 에서 backward focusing 을 강조했지만 이건 단지 한 전략일 뿐이다.
예를 들어, 현재 policy 하에서 자주 방문하는 states 에서 얼마나 쉽게 갈 수 있는지에 따라 states 에 초점을 맞출 수도 있어.
이걸 forward focusing 이라고 부를 수도 있겠지.
다음 몇 sections 에서 extreme form 으로 한 번 보자.

8.5 Expected vs. Sample Updates

이전 sections 의 예시는 learning 과 planning methods 들을 합치는 가능성의 범위에 대한 idea 를 줬다.
이 챕터의 나머지 부분에서, idea와 관련된 요소 중 일부를 분석한다.
expected 와 sample updates 의 상대적 이점부터 시작한다.

이 책의 많은 부분이 다양한 value-function updates 들에 대한 내용이었다.
one-step updates 를 보자면,
그들은 주로 3가지 binary dimensions 에 따라 다르다.
첫 두 dimensions은
state values 를 update 하는지 actions values를 update 하는지,
그리고 optimal policy 에 대한 value 를 estimate 하는지 주어진 임의의 policy 에 대해 value를 estimate 하는지다.
이들 두 dimensions 은 네 가지 value functions q_*, v_*, q_π, v_π 를 approximate 하는 데
네 가지 classes 의 updates 를 만든다.
다른 binary dimension 은 updates 가
expected updates(일어날 수 있는 모든 events 를 고려하느냐)인지
sample updates(일어날 수 있는 single sample을 고려하느냐)인지가 있다.

이들 세 binary dimensions 은 8가지 cases를 만들어내고
이 중 일곱은 특정 알고리즘에 대응한다.
이들 one-step updates 전부가 planning methods 에 쓰일 수 있다.
Dyna-Q agents 는 q_* sample updates 를 쓴다.

하지만 q_* expected updates 도 쓸 수 있고
expected 나 sample q_π updates 도 가능하다.
Dyna-AC system 은 learning policy structure 와 함께 v_π sample updates 를 사용한다.
stochastic 문제에선, prioritized sweeping 은 항상 expected updates 중 하나를 써서 수행한다.

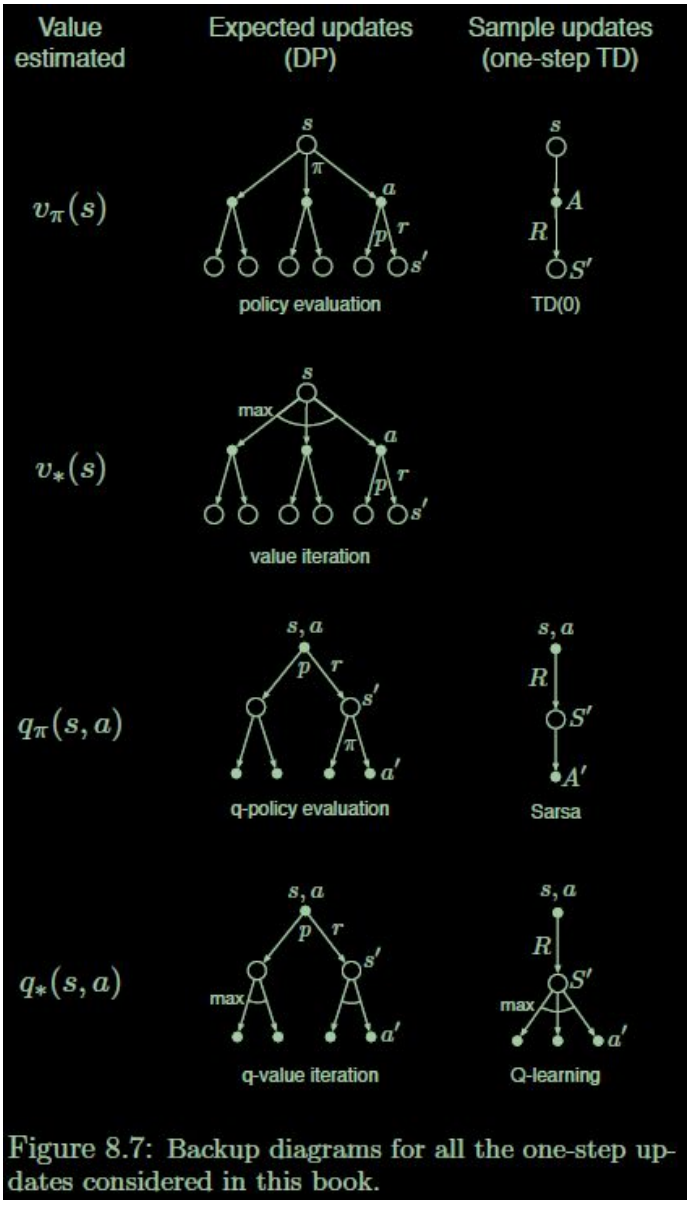


Figure 8.7: Backup diagrams for all the one-step updates considered in this book.

챕터 6에서 one-step sample updates 를 소개했을 때,
expected updates 의 대체인 것처럼 얘기했었지.
distribution model 없이는 expected updates 는 불가능하지만 sample updates 는 가능하다.
environment 의 sample transitions 이나 sample model 에서 얻은 sample transitions 을 써서 말이다.
그런 관점에서 암시하는 바는,
가능하다면 expected updates 가 sample updates 보다 선호된다는 거다.
근데 진짤까? 밀당고수?
expected updates 는 분명 더 좋은 estimate 를 낸다.

sampling error 에 의해 uncorrupt 니까.
하지만 더 많은 계산을 필요로 한다. 그리고 계산은 종종 planning 에서 limiting resource다.
planning 을 위한 expected 와 sample updates의 상대적 장점을 적절히 평가하려면
그들의 서로 다른 computational requirements 를 조절해야한다.

구체적으로,
q_*를 approximate 하는 expected와 sample updates,
discrete states 와 actions 의 special case,
approximate value function Q의 table-lookup representation,
estimated dynamics 형태의 model $\hat{p}(s', r|s, a)$
를 생각해보자.
state-action pair s, a 에 대한 expected update는:

$$Q(s,a) \leftarrow \sum_{s',r} \hat{p}(s',r|s,a)\Big[r + \gamma \max_{a'} Q(s',a')\Big]. \tag{8.1}$$

대응하는 sample updates 는, (model로부터) 주어진 sample next state and reward R', S 에서,
Q-learning-like update이다:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\Big[R + \gamma \max_{a'} Q(S',a') - Q(s,a)\Big], \tag{8.2}$$

α 는 usual positive step-size parameter다.

이들 expected 와 sample updates 의 차이는 environment 가 stochastic 일 경우
significant하다.
구체적으로, 주어진 state와 action 에서, 다양한 확률로 많은 가능한 next states가
발생할 수 있을 때.
만약 하나의 next state 만 가능하다면, 위에서 주어진 expected 와 sample updates 는
identical 하다. (α=1 일 때)
many possible next states면, 큰 차이가 있을거야.
expected update 가 더 좋은 점? 계산이 정확해서 좋아.
successor states의 $Q(s', a')$ 의 정확성에 의해서만 정확성이 제한되는 새로운 $Q(s, a)$ 를
생성한다.
sample update 는 게다가 sampling error 의 영향을 받지.
반면, sample update는 계산적으로 싸.
모든 possible next states 가 아니라 only one next state 만 고려하니까.
실제로는, update operations 에 필요한 계산은 보통 Q가 evaluate 되는 state-action pairs 의
수가 좌우해.
특정 starting pair s, a 에 대해, b가 *branching factor*
(i.e., $\hat{p}(s'|s,a) > 0$ 인 possible next states 의 수) 라고 하자.
그러면 이 pair 의 expected update 는 sample update 보다 대략 b배 계산이 더 필요하다.

만약 expected update 를 완료할 시간이 충분하면,
resulting estimate 가 일반적으로 b sample updates 의 그것보다 낫다.
sampling error 가 없으니까.
하지만 시간이 부족하면, sample updates 가 항상 preferable 하다.
그들은 value estimate 에서 b개 미만의 updates 로 value estimate를 최소한 얼마만큼은
개선하기 때문이다.
state-action pairs 가 많은 큰 문제에선, 보통 후자쪽 상황이다.

state-action pairs 가 많을 땐 걔들 모두의 expected updates 는 시간이 매우 많이 걸릴거야.
그 전에 많은 state-action pairs 에서 조금의 sample updates 를 하는 게
조금의 pairs 에서 expected updates 하는 것보다 훨씬 나을 수 있다.
주어진 computational effort 의 단위에서,
expected updates 몇 번 하는 게 낫니,
sample updates 를 b배 더 많이 하는 게 낫니?

Fig 8.8 은 이 질문에 대답해준다.

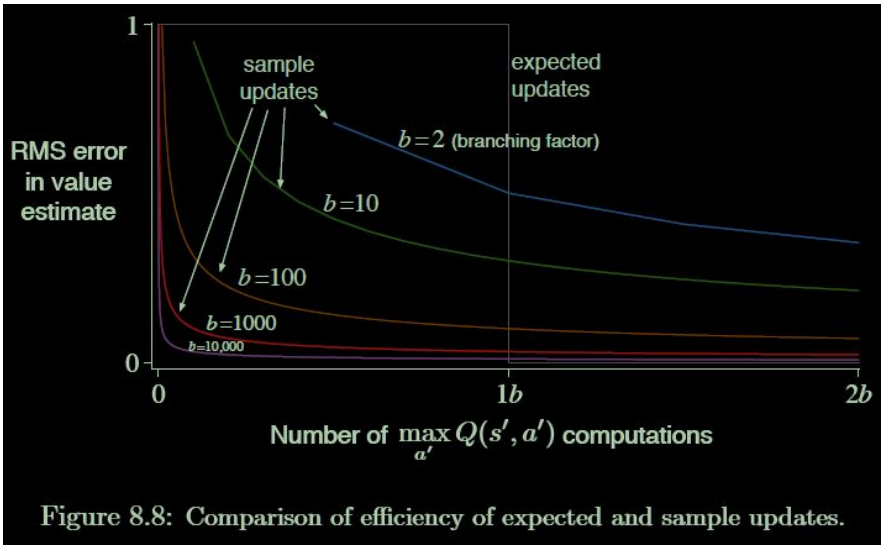


Figure 8.8: Comparison of efficiency of expected and sample updates.

b를 달리하여 expected 와 sample updates 의 computation time 의 함수로 estimation error 을 나타낸다.
모든 b successor states 가 equally likely 이고 initial estimate 의 error 가 1 인 case이다.
next states 의 values 는 맞다고 가정해서 expected update 는 1b 다 했을 때 error 가 0으로 줄어든다.
이 경우에, sample updates는 error 를 $\sqrt{\{ (b\text{-}1) / bt \}}$에 따라 줄인다.
t는 수행된 sample updates 의 수. sample averages 가정. 그러니까 $\alpha = 1/t$
key observation 은, b가 적당히 큰 경우 b updates 의 작은 부분으로도 error 가 엄청 감소한다는 거다.
이런 cases 에서, 많은 state-action pairs 가 values 를 크게 개선할 수 있다.
expected update 의 효과의 몇 퍼센트 이내로.
single state-action pair 가 expected update 를 수행하는 동시에.

Fig 8.8 에서 알 수 있는 sample updates의 장점은 아마 real effect 의 underestimate 이다.
실제 문제에서, successor states 의 values 는 그들 스스로가 update 되는 estimates 일 거다.
두 번째 장점은,
estimates 가 더 정확해 지는 걸 더 빨리 함으로써,
sample updates 는 successor states 로부터 backed up 된 values 가 더 정확해질거라는 것.
이런 결과로
large stochastic branching factors 와 정확하게 계산되기에 너무 많은 states 가 있는 문제에서
sample updates 가 expected updates 보다 superior 하기 쉽다.