6. Temporal-Difference Learning

강화학습에서 central 하고 novel 한 idea 가 뭔지 물으신다면,
그건 바로 temporal-difference (TD) learning 일 것이야.
TD learning 은 Monte Carlo ideas 와 dynamic programming (DP) ideas 의 combination 이야.
MC methods 처럼, TD methods 도 environment's dynamics 의 model 없이 raw experience로부터 바로 학습할 수 있다.
DP처럼, TD methods 도 부분적으로 다른 학습된 estimates 에 기반해 estimates 를 update 한다. final outcome 을 기다리지 않고.(bootstrap한다?)

TD와 DP, MC 들의 관계는 강화학습 이론에서 반복되는 주제다.
이 챕터가 그 모험의 시작이지.
그 탐험이 끝나기 전에, 이 ideas 와 methods 가 서로 섞이고 조합될 수 있는 다양한 방법들을 볼 거야.
특히, 챕터 7 에서는 n-step algorithms 을 소개할 건데, 그건 TD 와 MC methods 를 잇는 bridge 역할을 할 거야.
챕터 12 에서는 TD(λ) 알고리즘을 소개한다. 그들을 매끄럽게 통합하는 애야.

하던 것처럼, policy evaluation 또는 prediction problem 에 초점을 맞추며 시작한다.
주어진 policy π 에서 value function v_π 를 estimating 하는 문제지.
control problem(finding an optimal policy) 에서는, DP, TD, MC methods 가 전부 generalized policy iteration(GPI) 의 변형을 사용한다.
methods 에서의 차이는 주로 prediction problem 에 대한 접근 방식에서의 차이다.

6.1 TD Prediction

TD와 MC methods 모두 experience 를 쓴다. prediction problem 을 풀기 위해.
policy π 를 따르는 주어진 experience 에서, 두 methods 다 해당 experience 에서 발생한 nonterminal state S_t 에 대한 v_π의 estimate V 를 update 한다.
Roughly speaking, MC methods 는 visit 이후 return 을 알 수 있을 때까지 기다린다. 이후 그 return 을 $V(S_t)$ 에 대한 target 으로 쓴다.
nonstationary environment 에 적합한 simple every-visit MC methods 는

$$V(S_t) \leftarrow V(S_t) + \alpha\Big[G_t - V(S_t)\Big], \tag{6.1}$$

이다. 여기서 G_t 는 time t 이후의 actual return 이고, α 는 constant step-size parameter 이다.
이 method 를 constant-α MC 라고 부른다.
MC methods 가 $V(S_t)$ 에 대한 increment를 determine하려면 episode 가 끝날 때까지 기다려야 한다. (그래야 G_t 를 알 수 있음)
반면, TD methods 는 next time step 까지만 기다리면 된다.
t+1 시점에서, 그들은 즉각적으로 target 을 형성하고 observed reward R_{t+1} 과 estimate $V(S_{t+1})$ 을 이용한 useful update 를 한다.
가장 간단한 TD method 는 update

$$V(S_t) \leftarrow V(S_t) + \alpha\Big[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\Big] \tag{6.2}$$

를 S_{t+1} 로의 transition 과 R_{t+1} 을 받자마자 한다.

사실, MC update 에 대한 target 은 G_t 이다.
반면, TD update 에 대한 target 은 $R_{t+1} + \gamma V(S_{t+1})$ 이다.
이 TD method 를 TD(0) 이라 부른다. 또는 one-step TD 라 부름.
왜냐면 이건 TD(λ) 와 n-step TD methods 의 특수한 경우이기 때문이다.
아래 박스는 TD(0) 를 procedural form 으로 나타낸 것.

Tabular TD(0) for estimating v_π

Input: the policy π to be evaluated
Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

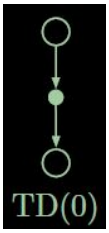
Loop for each episode:
  Initialize S
  Loop for each step of episode:
    $A \leftarrow$ action given by π for S
    Take action A, observe R, S'
    $V(S) \leftarrow V(S) + \alpha[R + \gamma V(S') - V(S)]$
    $S \leftarrow S'$
  until S is terminal

TD(0) 가 어느 정도 existing estimate 에 기반해 update 하기 때문에, 이걸 DP 처럼 bootstrapping method 라고 한다.
챕터 3에서

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi[G_t \mid S_t = s] && (6.3) \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] && (\text{from } (3.9)) \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s]\,. && (6.4) \end{aligned}$$

요거 봤지?
Roughly speaking, MC methods 는 (6.3) 의 estimate 를 target 으로 쓴다.
반면, DP methods 는 (6.4) 의 estimate 를 target 으로 쓴다.
Monte Carlo target 은 estimate 이다. 왜냐면 (6.3) 에서 expected value 는 not known 이기 때문이다. sample return 이 real expected return 대신 쓰인다.
DP target 은 estimate 인데, environment 의 model 에 의해 완전히 제공된다고 가정하는 expected values 때문이 아니라, $v_\pi(S_{t+1})$ 을 모르고 대신 current estimate $V(S_{t+1})$ 를쓰기 때문이다.
TD target 은 두 이유 모두로 estimate 이다. (6.4)에서 expected values 를 샘플링하고, true v_π 대신 current estimate V 를 쓰는 것.
그래서 TD methods 는 Monte Carlo 의 sampling 을 DP의 bootstrapping 과 결합한다.
우리가 볼 수 있듯이, 주의와 상상력으로, 이게 MC 와 DP methods 둘 모두의 장점을 얻는 데 좋다는 건지 싫다는건지.. this can take us a long way toward obtaining the advantages of both MC and DP.

이건 tabular TD(0) 의 backup diagram 이다.



위쪽 state node 의 value estimate 는 update 되는 게, 바로 다음 이어지는 state 로의 transition 의 one sample 에 근거해서 update 한다.
TD 와 MC updates 를 sample updates 라고 했었지?
왜냐면 얘들은 sample successor state(or state-action pair) 를 미리 내다보고,
backed-up value 를 계산하는 과정에서 successor 의 value 와 reward를 사용한 다음,
그 후, 그에 따라 original state (or state-action pair) 의 value를 update 하는 것을 포함하기 때문이다.
Sample updates 는 DP methods 의 expected updates 와 다르다.
얘들은 complete distribution of all possible successors 가 아니라
single sample successor 에 기반한다.

마지막으로, TD(0) update 안의 괄호 안의 quantity 는 error 같은 거다.
S_t 의 estimated value 와 better estimate인 $R_{t+1} + \gamma V(S_{t+1})$ 의 차이를 측정하는.
이 quantity 를 TD error 라고 부르고, 강화학습 전반에 걸쳐 다양한 형태로 나온다.

$$\delta_t \doteq R_{t+1} + \gamma V(S_{t+1}) - V(S_t). \tag{6.5}$$

각 time 에서 TD error 가 그 time 에서 만들어진 estimate 의 error 이다.
TD error 가 다음 state 와 다음 reward 에 depend 하고 있기 때문에 one time step later 하기 전까진 available 하지 않다.
즉, δ_t가 error in $V(S_t)$ 이고 t+1 time 에서 available 하다.

array V 가 episode 동안 변하지 않으면 (MC methods 에서 변하지 않는 것처럼),
Monte Carlo error 는 TD errors 의 합으로 쓸 수 있다.

$$\begin{aligned} G_t - V(S_t) &= R_{t+1} + \gamma G_{t+1} - V(S_t) + \gamma V(S_{t+1}) - \gamma V(S_{t+1}) && \text{(from (3.9))} \\ &= \delta_t + \gamma\big(G_{t+1} - V(S_{t+1})\big) \\ &= \delta_t + \gamma\delta_{t+1} + \gamma^2\big(G_{t+2} - V(S_{t+2})\big) \\ &= \delta_t + \gamma\delta_{t+1} + \gamma^2\delta_{t+2} + \cdots + \gamma^{T-t-1}\delta_{T-1} + \gamma^{T-t}\big(G_T - V(S_T)\big) \\ &= \delta_t + \gamma\delta_{t+1} + \gamma^2\delta_{t+2} + \cdots + \gamma^{T-t-1}\delta_{T-1} + \gamma^{T-t}\big(0 - 0\big) \\ &= \sum_{k=t}^{T-1} \gamma^{k-t}\delta_k. && (6.6) \end{aligned}$$

이 identity 는 정확하지 않다. V가 episode 동안 update 되면.(TD(0) 에선 update 되지)
하지만 step size 가 작으면 approximately hold 할 수 있다.
이 identity 의 일반화는 TD learning 의 알고리즘과 이론에서 중요한 역할을 한다.

(Exercise 6.1)
V 가 episode 동안 바뀌면, (6.6) 은 only holds approximately 한다.
두 가지 면의 차이는 뭣일까?
V_t 는 TD error (6.5)와 TD update (6.2) 안에 time t에서 쓰인 state values 의 array 를 나타낸다. 위의 derivation 을 다시 해봐. TD errors 의 합에 뭔가 더 추가되면 MC error 가 돼.

(Example 6.1) Driving Home
집에 올 때 얼마나 걸리나 예측해본다.
회사를 나올 때 시간, 요일, 날씨 같은 연관이 있을 것 같은 애들을 다 적는다.

이번 금요일에 6시 정각에 퇴근하면서 집까지 30분 걸릴 거라고 estimate 했다고 해보자.
차까지 가니 6:05. 그러더니 이제 비가 오네. traffic 이 비오면 더 느려지니까 35분 걸린다고 다시 estimate. 총 40분 걸리는거지.
15분 후에 highway 를 끝냄. secondary road 로 나가자 total 을 35분으로 다시 estimate.
하지만, 이때 느린 트럭 뒤에 잡힘. 도로가 좁아서 지나치기도 힘들고.
6:40 니가 사는 side street 를 만날 때까지 따라갈 수 밖에 없었어.
3분 후에 집에 도착.
states, times, predictions 의 sequence 가 다음과 같다.

State	*Elapsed Time (minutes)*	*Predicted Time to Go*	*Predicted Total Time*
leaving office, friday at 6	0	30	30
reach car, raining	5	35	40
exiting highway	20	15	35
2ndary road, behind truck	30	10	40
entering home street	40	3	43
arrive home	43	0	43

여기서 rewards 는 각 leg of journey 의 elapsed times 이다. (이게 만약 travel time 을 minimizing 하는 control 문제였다면, rewards 를 elapsed time 의 negative 로 했을 거다. 하지만 여기선 prediction (policy evaluation)만 다루니 positive numbers 로 간단히 다룰 수 있다.)
discounting 은 안 하고 (γ = 1), 그래서 각 state 의 return 은 그 state 에서 실제 time to go 이다.
각 state 의 value는 expected time to go.
Predicted time to go 는 각 state에서 current estimated value 이다.
MC methods 의 operation 을 보는 간단한 방법은 마지막 컬럼인 predicted total time을 그려보는 거다. 아래 그림 (Fig 6.1) 왼쪽처럼.

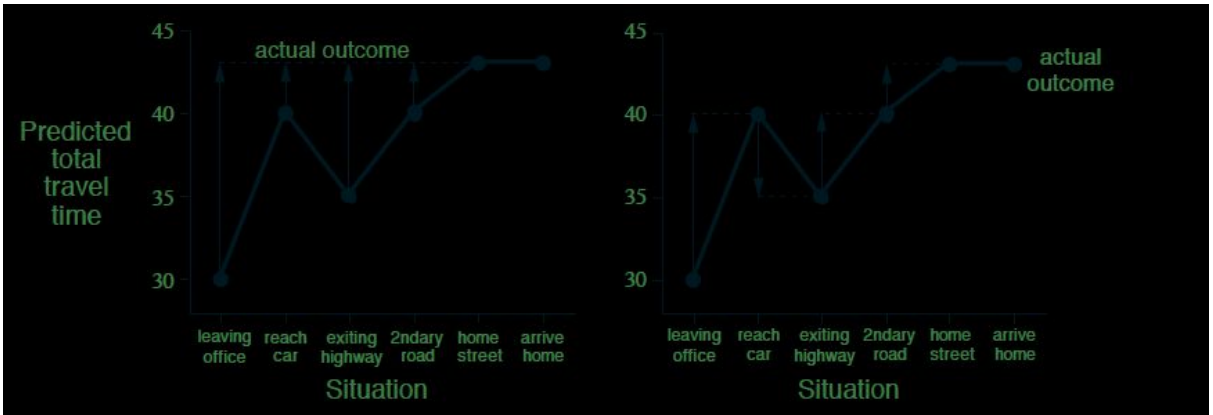


Figure 6.1: Changes recommended in the driving home example by Monte Carlo methods (left) and TD methods (right).

화살표는 predictions 의 변화를 보여준다. 어떤 prediction? constant-α MC method (6.1) 에 의해 추천받은. α = 1일 때.
얘들은 정확히 각 state 의 estimated value (predicted time to go) 와 actual return (actual time to go) 사이의 errors 다.
예를 들어, highway 를 벗어났을 때 집까지 15분 걸릴 거라고 생각했지만 사실 23분 걸렸지.

식 6.1 을 여기서 적용하고 highway 를 벗어난 뒤 time to go 의 estimate 에서의 increment를 결정한다.
error, G_t - $V(S_t)$ 는 여기서 8분.
step-size parameter α 가 1/2이라고 해보자.
그러면 predicted time to go 는 이 experience 의 결과로 4분식 상향 조정된다.
이건 아마 이 case 에서 너무 커. 트럭은 아마 그냥 unlucky break? 였지.
어떤 event 에서도, 변화는 off-line 에서만 일어날 수 있다. 그러니까, 집에 도착한 이후에.
이 시점에서만 actual returns 을 알 수 있어.
학습을 시작하기 전에 final outcome을 알 때까지 기다리는 게 꼭 필요할까?
다른 날을 생각해보자.
마찬가지로 회사를 나서며 30분 걸린다고 예상했어.
근데 massive traffic jam에 갇혀버렸지.
회사를 떠난지 25분이 지났음에도 여전히 highway 에서 범퍼를 맞대고있어.
집까지 25분 추가로 걸린다고 다시 estimate. 총 50분.
처음 30분 estimate 가 너무 optimistic했네.
initial state 에서의 estimate 를 올리는데 집까지 도착하는 걸 기다려야할까?
MC 에 따르면 그래야하지. true return을 아직 모르니까.

TD에 따르면 즉각적으로 학습할거야. initial estimate를 30에서 50으로 바로 바꾸는거지.
사실, 각 estimate는 바로 다음 estimate로 이동할거야.
첫 날로 돌아가서, (Fig 6.1)의 오른쪽 그림은 TD rule (6.2) 에 의해 recommended 된 predictions의 변화를 보여준다. (α = 1 일 때)
각 error는 prediction 시간의 변화, 즉 temporal differences in predictions 에 비례한다.

traffic 에 갇혀있으면서 뭔가 할 일을 주는 것 뿐 아니라, 몇 가지 계산적 이유가 있다.
current predictions에 근거해 학습하는 게 actual return 을 알 때까지 termination을 기다리는 것보다 더 좋은 이유가 있다 이말이야.
다음 섹션에서 얘기하자?

(Exercise 6.2)
이거 보면 왜 TD가 MC보다 보통은 더 효율적인지 직관을 기를 수 있을거야.
위에서 봤던 driving home 예제에서 TD 와 MC methods를 어떻게 다뤘는지 생각해봐.
TD update 가 MC update보다 평균적으로 더 낫다는 시나리오가 나오겠냐?
past experience 와 current state를 얘기해보렴. 예시 시나리오를 말해봣.
힌트: experience가 아주 많다고 해봐. 근데 새 빌딩과 새 주차장으로 옮긴거지. highway는 같고. 이제 너는 새 빌딩에서의 predictions을 학습하기 시작해. 이 경우에 왜 TD updates가 훨씬 좋을 가능성이 높은지 보이니? 적어도 initially 할 때 말이야. original task 에서도 비슷한 종류의 일이 일어날까?

6.2 Advantages of TD Prediction Methods
TD methods 는 다른 estimates에 부분적으로 근거해 estimates 를 update한다.
guess 로 guess 를 학습하는 거지. bootstrap.
이게 좋은 일일까?
TD 가 MC, DP 보다 좋은 건 뭐지?

확실히, environment, reward, next-state 확률분포의 model 이 필요없다는 건 TD 가 DP에 비해 갖는 장점임.

TD가 MC에 비해 갖는 다음 장점은, 자연스럽게 on-lilne, fully incremental fashion 으로 구현된다는 점이다.
MC로는 episode의 끝까지 기다려야한다. 그래야 return을 알 수 있으니까.
TD는 한 time step 만 기다리면 된다.
요게 생각보다 critical하다.
어떤 applications은 아주 긴 episodes를 갖고 있어서 episode가 끝날 때까지 모든 학습을 기다리는 건 매우 느리다.
다른 applications은 continuing tasks 에 episodes 가 아예 없지.
마지막으로, 어떤 MC methods 는 실험적 actions 이 취해진 episodes를 무시하거나 discount 해야한다. 이건 학습을 엄청 느리게 만든다.
TD methods는 이런 문제에 훨씬 덜 예민하다. 왜냐면 이후의 actions 이 뭐가 취해지든 상관없이 각 transitions으로부터 학습하기 때문.

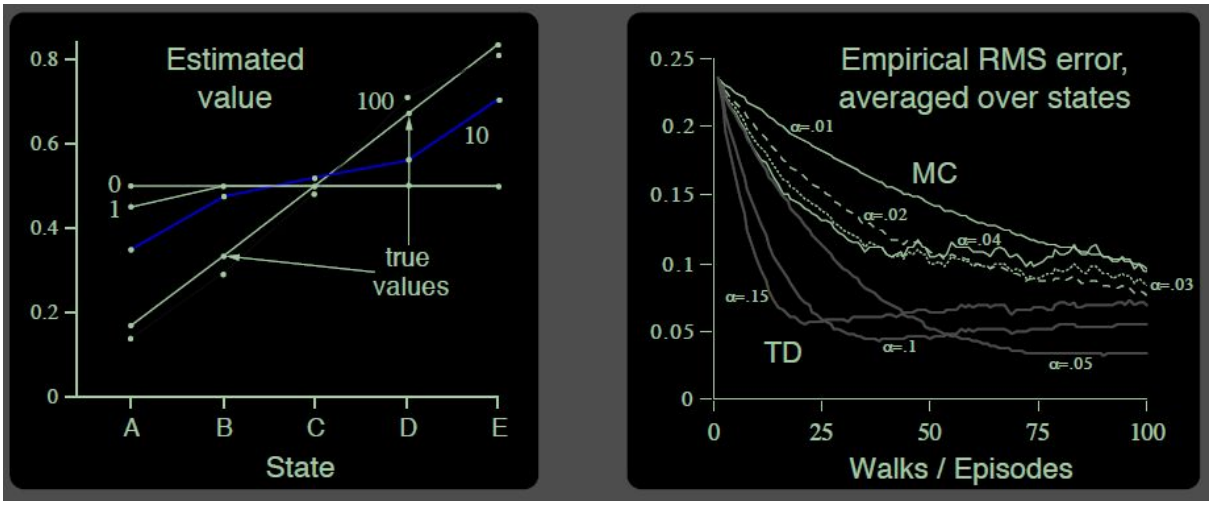
하지만 TD methods가 좋냐?
next 로부터 one guess를 학습하기엔 확실히 편리하다. actual outcome 을 기다리지 않고.
하지만 correct answer로의 convergence를 보장할 수 있을까?
다행히 답은 yes다.
for any fixed policy π, TD(0) 는 v_π 로 수렴하는 게 증명돼 있다. constant step-size parameter 의 평균이 충분히 작은 경우에?
그리고 확률 1로 step-size parameter 가 줄어들면 수렴하는건가? usual stochastic approximation conditions (2.7) 에 따라?
대부분의 convergence proofs 는 (6.2)처럼 알고리즘의 table-based case에 적용된다.
하지만 일부는 general linear function approximation 에도 적용된다.
이 결과들은 챕터 9 에서 더 general setting 에서 다룬다.

TD와 MC methods 가 correct predictions 에 asymptotically 수렴하면, 다음 질문은 당연히, 뭐가 먼저 도달할까 이다. 다른 말로, 뭐가 더 빨리 학습하냐.
뭐가 limited data 에 더 효율적으로 쓰일까.
이건 현 시점까지 증명 안 된 open question 이다.
사실 이런 질문이 적합한 방식인지도 분명하지 않다.
하지만 실제로는 TD 가 constant-α MC 보다 보통 더 빠르다. stochastic tasks 에서.
Example 6.2 에 나와 있는 것처럼.

(Example 6.2) Random Walk
여기선 경험적으로 비교한다. 뭘? TD(0) 와 constant-α MC 가 다음과 같은 Markov reward process 에 적용됐을 때 prediction abilities 를 비교해보자.

Markov reward process (MRP) 는 Markov decision process without actions 이다.
prediction problem 에 초점을 맞출 때 MRP 를 주로 쓸 거야.

environment 로 인한 dynamics 와 agent 로 인한 dynamics 를 구분할 필요가 없을 때 말이야.
이 MRP 에서, 모든 episodes 는 center state C 에서 시작한다.
그 후 각 step 마다 왼쪽이나 오른쪽 한 state 로 proceed. 왼오는 같은 확률로.
episodes 는 좌우 끝에 도달하면 terminate.
오른쪽에서 terminate 하면 +1 reward.
다른 모든 rewards 는 0이다.
예를 들어, 요런 state-and-reward sequence 가 episode 에 포함될 수 있다.
C, 0, B, 0, C, 0, D, 0, E, 1.
이 task 는 undiscounted 니까, center state 의 true value 는 v_π(C) = 0.5 이다.
A 에서 E 까지 true values 는 ⅙, 2/6, 3/6, 4/6, ⅚ 이다.



왼쪽 그래프는 TD(0) 의 single run 에서 다양한 수의 episodes 이후 학습한 values 이다.
100 episodes 이후 estimates 가 true values 에 가장 가깝다. constant step-size parameter α=0.1 일 때.
values 는 가장 최근 episodes 의 outcomes 에 대응해 무한정 변동한다.

오른쪽 그래프는 다양한 α 에 따른 두 methods 의 learning curves 이다.
performance measure는 RMSE 이고, 다섯 states 를 평균내고, 그 후 averaged over 100 runs 했다.
모든 cases 에서 approximate value function 은 모든 s 에 대해 intermediate value $V(s)$ = 0.5 로 initialized 됐다.
TD method 는 일관적으로 MC method 보다 낫다. 이 task 에선.

(Exercise 6.3)

6.3 Optimality of TD(0)

finite amount 의 experience 만이 가능하다고 해보자. 10 episodes나 100 time steps 쯤?
이 경우에, incremental learning methods 의 일반적인 approach 는 method 가 답에 수렴할 때까지 반복적으로 experience 를 present 하는거야.

주어진 approximate value function 에서, (6.1)과 (6.2) 에 의해 specified 된 increments 는 nonterminal state 가 visit 된 매 time step t 마다 계산된다. 하지만 value function 은 한 번만 바뀐다. 모든 increments 의 합으로.
그러면 모든 가능한 experience가 새로운 value function으로 다시 process 된다. 전체적으로 새로운 increment를 만들기 위해. and so on. value function이 converge 할 때까지.
이걸 batch updating 이라고 부른다. updates가 training data의 전체 batch 를 각각 학습하고 나서야 update 가 이루어지기 때문.

batch updating 에서는, TD(0) 가 step-size parameter α에 독립적으로 single answer 에 deterministically converge 한다. α 가 충분히 작을 때.
constant-α MC methods 도 같은 조건 하에서 deterministically converge 한다. 하지만 다른 answer 로 converge.
이 두 answers 를 이해하는 건 두 methods 의 차이를 이해하는 데 도움을 줄 거야.
normal updating 하에서, methods는 각 batch answers 로 all the way 이동하지 않는다.
하지만 어떤 면에선 그 방향으로 한 걸음씩 내딛는다.
일반적인 상황, 모든 가능한 tasks에서 두 answers 를 이해하길 시도하기 전에 몇 개 예를 보자.

(Example 6.3) Random walk under batch updating
TD(0) 와 constant-α MC 의 batch-updating 버젼을 random walk prediction 에 적용한 예를 볼까
각 new episode 이후, 그때까지 나온 모든 episode는 batch 로 취급한다.
그들은 반복적으로 알고리즘에 presented 됐다. value function 이 converge 될 정도로 충분히 작은 α로.
이후, 결과로 나온 value function 은 v_π 와 비교되고, 5개 states 의 average RMSE (전체 experiment의 100 independent repetitions) 는 Fig 6.2 에 learning curves 를 얻기 위해 그려져 있다.
batch TD method 가 batch MC method 보다 consistently better 인 게 보이니.

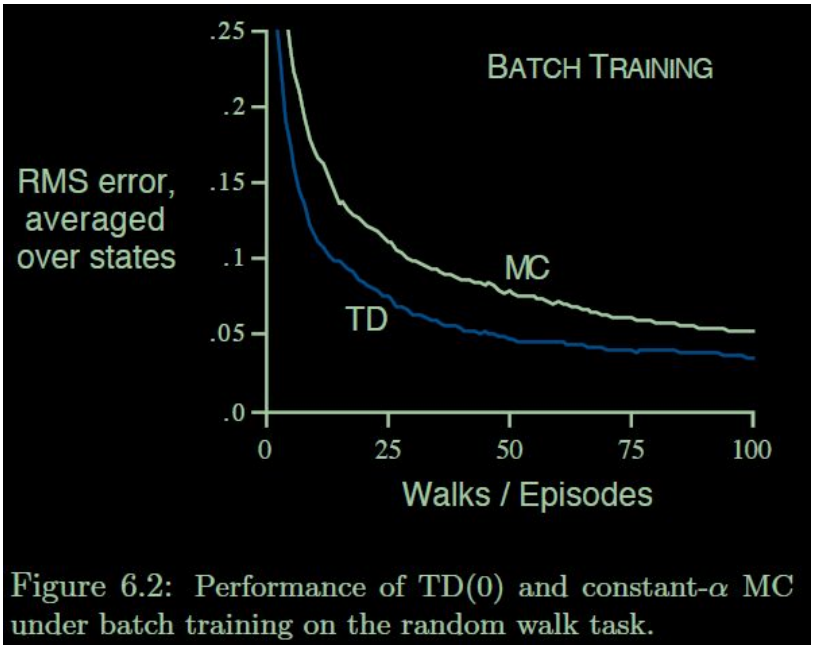


Figure 6.2: Performance of TD(0) and constant-α MC under batch training on the random walk task.

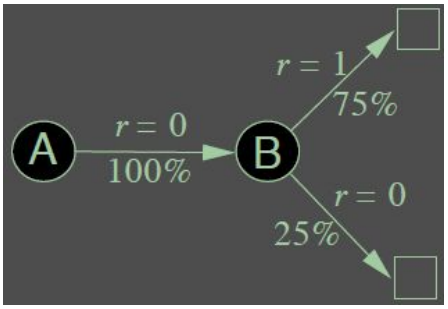
batch training 하에서, constant-α MC 는 각 state s 를 방문한 뒤 경험된 actual returns 의 sample averages 인 $V(s)$ 로 수렴한다.
얘들 $V(s)$ 는 optimal estimates 다. 어떤 면에서? training set 에서 actual returns 의 MSE 를 minimize 하는 면에서.
이런 측면에선 놀라운 거야. batch TD method 가 RMSE 에 따라서는 더 좋을 수 있었다는 게 말이야.
이 optimal method 보다 어떻게 batch TD 가 더 좋을 수 있었을까?
답은, MC method 는 limited way 에서만 optimal 하다는 것. 그리고 TD 는 returns 를 predicting 하는 데 더 관련있는 쪽으로 optimal 하다는 것.
하지만 먼저 다른 예시로 다른 종류의 optimality 에 대한 직관을 길러보자.
예시 6.4 를 볼까?

(Example 6.4) You are the Predictor
이름이 거창하고만.
니가 unknown Markov reward process 의 returns 을 predict 하는 역할을 한다고 해보자.
다음 여덟 episodes 를 observe 했다 쳐.

A, 0, B, 0	B, 1
B, 1	B, 1
B, 1	B, 1
B, 1	B, 0

첫 번째 episode 가 state A 에서 시작해 B 로 transitioned 했고 reward 를 0 받았고, B 에서 terminated 하면서 reward 가 0 이었다는 뜻.
다른 일곱 episodes 는 더 짧지. B 에서 시작해서 바로 terminate.
주어진 이 batch of data 에서, optimal predictions 이 뭐라고 할래? 그러니까 estimates V(A)와 V(B) 의 best values 가 뭐냔 말이지.
아마 V(B) 의 optimal value 가 ¾ 라는 건 다들 동의할거야.
왜냐면 state B 에서 8번 중 6번 return 1 받고 바로 종료됐고, 다른 두 번은 return 0으로 종료됐으니까.

그럼 estimate V(A) 의 optimal value 는?
두 가지 reasonable answers 를 주겠어.
하나는, 100% 로 A에서 reward 0 을 받고 B로 갔지. B의 value는 ¾ 이라고 이미 결정했으니까, A도 ¾ 이어야 해.
이 답을 보는 한 가지 방법은, 첫 번째로 Markov process 를 모델링 하고 (이 경우엔 요 그림처럼)



그리고는 모델에 대한 정확한 estimates 를 계산하는 것에 기반한다는 것이다.
이 경우엔 실제로 V(A) = ¾ 이 되지.

이건 batch TD(0) 가 주는 답이기도 해.

다른 reasonable answer 는, 우리가 A를 한 번 봤고 return 이 0이었어. 그래서 V(A) 를 0이라고 estimate 하는거지.
이게 batch Monte Carlo methods 가 주는 답이야.
training data 에 대한 minimum squared error 를 주는 답이기도 해.
사실 data 에 대해서 zero error 를 주지.
하지만 우린 여전히 first answer 가 더 낫길 바라.
process 가 Markov 면, first answer 가 future data 에 더 적은 error 를 발생시키길 원하지.
Monte Carlo answer 가 existing data 에 더 낫다고 하더라도.
∎

Example 6.4 는 batch TD(0) 와 batch Monte Carlo methods 로 찾은 estimates 간의 일반적인 차이를 보여준다.
Batch Monte Carlo methods 는 항상 training set 에 대한 MSE 를 minimize 하는 estimates 를 찾고, batch TD(0) 는 항상 Markov process 의 maximum-likelihood model 에 대해 정확히 맞도록 하는 estimates 를 찾는다.
일반적으로, parameter 의 maximum-likelihood estimate 는 data를 생성하는 확률이 가장 큰 parameter value 이다.
이 경우엔, maximum-likelihood estimate 는 관측된 episodes 로부터 명백한 방식으로 형성된 Markov process 의 model 이다. i에서 j로의 estimated transition probability 는 i에서 j로 갔던 관측된 transitions 의 부분이고, associated expected reward 는 그 transitions 에서 관측된 rewards 의 평균이다.
주어진 이 모델로, value function 을 estimate 할 수 있다. model 이 정확히 맞으면 이것도 정확히 맞을 거다.
이걸 certainty-equivalence estimate 라고 한다. 왜냐면 underlying process 의 estimate가 approximated 되기보다 known with certainty 라고 가정하는 것과 equivalent 하기 때문이다.
일반적으로, batch TD(0) 는 certainty-equivalence estimate 로 수렴한다.

이게 TD 가 왜 MC 보다 더 빨리 수렴하는지 설명하는 데 도움을 줘.
batch form 에선, TD(0) 가 MC methods 보다 빠르다. 왜냐면 true certainty-equivalence estimate 를 계산하기 때문.
이게 Fig 6.2 의 random walk task 의 batch results 에서 본 TD(0) 의 장점을 설명해준다.
certainty-equivalence estimate 에 대한 관계는 nonbatch TD(0) 의 속도 장점을 부분적으로 설명해주기도 한다. (Example 6.2, 100, right graph)
nonbatch methods 가 certainty-equivalence 나 minimum squared-error estimates 를 달성하지 못한다고 해도, 그들은 이런 방향으로 움직인다고 이해할 수 있다.
Nonbatch TD(0) 가 constant-α MC 보다 빠를 수도 있다. 더 좋은 estimate 로 가고 있으니까. 완전히 거기 가진 못한다고 해도.
현재 on-line TD 와 Monte Carlo methods 의 상대적 효율성에 대해선 더 말할 수 있는 건 없다.

마지막으로, certainty-equivalence estimate 가 어떤 면으론 optimal solution 이라 해도, directly 계산하는 게 거의 never feasible 해. (feasible solution을 '가능한 해'라고 하네. 속에 optimal solution을 포함하고 있고. naver 에서.)

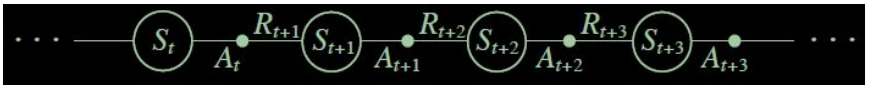
n = $|\mathcal{S}|$ 가 states 수면, 프로세스의 maximum-likelihood estimate 를 그냥 형성하는 게 order of n^2 memory 를 필요로 할 수 있고, 대응하는 value function을 계산하는 건 order of n3 계산 steps 을 필요로 한다. 전통적 방법으로 했을 때.
이런 관점에서, TD methods 가 order n 을 넘지 않는 memory를 사용하여 같은 solution을 approximate 할 수 있고 training set 에 반복된 계산을 할 수 있다는 것은 아주 striking.
state spaces 가 큰 tasks 의 경우, TD methods 가 certainty-equivalence solution 을 approximating 하는 유일한 feasible way 일 수 있다.

(Exercise 6.7) TD(0) update 의 off-policy version 을 design 해봐. arbitrary target policy π 와 behavior policy b 를 커버하는 와 함께 쓰일 수 있는 TD(0) update. 각 step t 에서 importance sampling ratio $\varrho_{t:t}$ (5.3) 을 쓰는 TD(0) update

6.4 Sarsa: On-policy TD Control

이제 TD prediction methods 를 control problem 에 쓰는 걸 보자.
여느때와 같이, generalized policy iteration(GPI) 의 패턴을 따르고, 이번에만 TD methods 를 evaluation 이나 prediction 파트에 쓴다.
As with Monte Carlo methods, exploration 과 exploitation 을 trade off 할 필요가 있다.
그리고 또다시 접근법은 두 main classes 로 나뉜다. on-policy 와 off-policy.
이 section에선 on-policy TD control method 를 보여줄게.

첫 번째 step 은 action-value function 을 학습하는 거다. state-value function 을 학습하기 보단.
특히, on-policy method 에선 current behavior policy π 와 모든 states s, actions a 에 대한 $q_\pi(s, a)$ 를 estimate 해야한다.
이건 위에서 얘기했던, v_π 를 학습할 때처럼 본질적으로 같은 TD method 를 사용해서 수행할 수 있다.
episode 가 states 와 state-action pairs 의 alternating sequence(교대로 나온다는 말인듯)로 이루어져 있는거 기억하지?



이전 section 에선 state 에서 state 로 가는 transitions 을 보고 states 의 values 를 학습했지.
이번에는 state-action pair 에서 state-action pair 로 가는 transitions 과 state-action pairs 의 values 를 학습하는 걸 보자.
공식적으로 이런 cases 들은 identical 하다.
모두 reward process 가 있는 Markov chains 이다.
TD(0) 하에서 state values 의 수렴을 보장하는 theorems 또한 대응하는 <actions values 에 대한 알고리즘>에 적용한다:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\Big]. \tag{6.7}$$

이 update 는 nonterminal state S_t 에서의 모든 transition 이후에 이루어진다.
S_{t+1} 이 terminal 이면, $Q(S_{t+1}, A_{t+1})$ 는 0으로 정의한다.

이 rule 은 한 state-action pair 에서 다음으로의 transition 을 구성하는 $(S_t, A_t, R_t, S_{t+1}, A_{t+1})$ events 의 quintuple 의 모든 element 를 다 사용한다.
이 quintuple 이 알고리즘에 Sarsa 라는 이름을 줬다.
Sarsa 의 backup diagram 은 이거다.

(Exercise 6.8) (6.6)의 action-value 버전이 TD error

$$\delta_t = R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)$$

의 action-value form 을 만족한다? 는 걸 보여줘. step 간의 변화에서 values 는 바뀌지 않는다고 또다시 가정하고.

∎

Sarsa prediction method에 기초한 on-policy control algorithm 을 디자인하는 건 쉽다.
모든 on-policy methods 에서처럼, 우린 계속 behavior policy π 에 대한 q_π 를 estimate 할 거고, 동시에, q_π에 대한 greediness 쪽으로 π 를 바꿀 거다.
Sarsa control algorithm 의 일반적인 폼은 아래 박스와 같다.

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

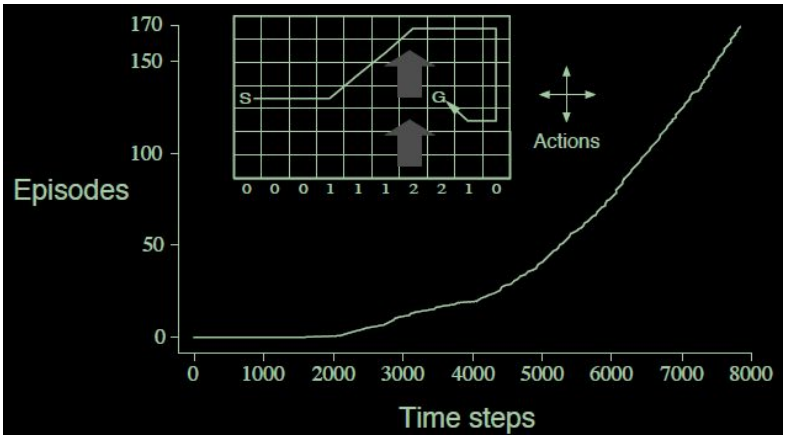
Loop for each episode:
  Initialize S
  Choose A from S using policy derived from Q (e.g., ϵ-greedy)
  Loop for each step of episode:
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ϵ-greedy)
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma Q(S', A') - Q(S, A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

박스 설명:
알고리즘 파라미터가 step size α, ε 가 있다.
모든 $s \in \mathcal{S}^+$, $a \in \mathcal{A}(s)$ 에 대해서 $Q(s, a)$를 arbitrarily initialize. Q(terminal,.) = 0 빼고.
각 episode 별로 Loop:
    S를 initialize.
    Q로부터 이끌어낸(?) policy 를 써서 S 에서 A를 고른다. (ε-greedy 같은 거)
        episode 의 각 step 별로 Loop:
            action A 를 하고 R, S' 을 observe.
            Q로부터 이끌어낸(?) policy 를 써서 S' 에서 A′ 을 고른다.
            $Q(S, A) \leftarrow Q(S, A) + \alpha[\, R + \gamma Q(S', A') - Q(S, A)\,]$
            $S \leftarrow S'$; $A \leftarrow A'$;
        S 가 terminal 일 때까지

∎

Sarsa algorithm 의 convergence properties 는 policy가 Q 에 의존하고 있는 특성에 달려 있다.
예를 들어, ε-greedy 나 ε-soft policies 를 쓸 수 있겠지.
Sarsa 는 확률 1로 optimal policy 와 action-value function 으로 수렴한다. 모든 state-action pairs 가 무한 번 visit 되고 극한에서 policy 가 greedy policy 로 수렴하는 한.(arranged 될 수 있다. 예를 들어 ε-greedy policies 에서 ε = 1/t 로 세팅해서?)

(Example 6.5) Windy Gridworld



위 Fig(6.3)에 나오는 건 start 와 goal states 가 있는 평범한 gridworld 이다.
하지만 다른 점이 하나 있지.
가운데 위로 가는 crosswind 가 있어.
actions 은 4개 - up, down, right, left.
하지만 가운데 영역에선 다음 states 가 바람 때문에 위로 가. 힘은 column 마다 다르고.
예를 들어, goal 바로 오른쪽 cell 이면, left 라는 action 을 했을 때 goal 바로 위로 가게 되는거야.
이걸 goal 에 도달하기 전까지 constant rewards -1 을 받는 undiscounted episodic task 로 취급하자.
(Fig 6.3) 의 그래프는 ε-greedy Sarsa 를 이 task 에 적용한 결과이다. ε = 0.1, α = 0.5, 모든 s, a 에 대해 $Q(s, a)$ = 0 이다.
그래프의 올라가는 경사는 시간에 따라 더 빠르게 goal 에 도달하는 걸 나타낸다.
8000 time steps 이 되면, greedy policy 가 optimal 이 된 지 오래다? 궤적이 그림으로 나와 있고. 계속된 ε-greedy exploration 은 average episode length 가 약 17 steps 로 유지시켜준다. 15가 minimum인데.
Monte Carlo methods 는 이 task 에서 쉽게 쓸 수 없다. termination 이 모든 policies 에서 보장되지 않기 때문에.
만약 policy 가, agent 가 같은 state에 머물도록 찾아졌다면, 다음 episode는 끝나지 않을 거다.
Sarsa 와 같은 step-by-step learning methods 는 이런 문제가 없다. 왜냐면 episode 동안 그런 policies 가 좋지 않다는 걸 빠르게 학습하고 다른 걸로 전환하기 때문이다.
∎

(Exercise 6.9) Windy Gridworld with King's Moves

windy gridworld task 를 8개 가능한 actions 이 있다고 가정하고 다시 풀어봐라. diagonal moves 포함해서. 얼마나 좋아졌니? 바람에 의한 거 말고는 움직이지 않는 9번째 actions 을 추가하면 더 좋아질까?

(Exercise 6.10)Stochastic Wind

6.5 Q-learning: Off-policy TD Control

강화학습에서 초기 breakthroughs 는 off-policy TD control algorithm 이다.
그것이 Q-learning.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\Big]. \tag{6.8}$$

이렇게 정의된다.
이 case에서, 학습된 action-value function Q 는 따르는 policy 에 상관없이 optimal action-value function q_* 를 directly approximates 한다.
이게 알고리즘의 분석을 엄청 간단하게 만들어주고 early convergence proof 를 가능하게 했다.
policy 는 어떤 state-action pairs 가 visit 되고 update 되는지 결정하는 데 여전히 영향력이 있다.
하지만 correct convergence 를 위해 필요한 건, 모든 pairs 가 계속 update 된다는 것이다.
챕터 5에서 봤듯이, 이건 최소한의 요구 조건이다. general case 에서 optimal behavior 를 찾는 걸 보장하는 어떤 method 든 require 한다는 면에서.
이 가정과 step-size parameters 의 sequence 에 대한 usual stochastic approximation conditions 의 variant(변형) 하에서, Q 는 확률 1로 q_* 에 수렴하는 게 증명 돼 있다.

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
    $S \leftarrow S'$
  until S is terminal

박스 설명:
위에서 봤던 거에서 update 공식만 달라져있다. 어떻게? 다음 state 에서 Q를 최대로 하는 a를 고르는거?

Q-learning 의 backup diagram 은 어떻게 될까?
rule (6.8) 은 state-action pair 를 update 한다. 그래서 top node = the root of the update 는 small, filled action node 여야 한다.
update 또한 next state 에서 가능한 모든 actions 를 maximizing 하는 actions nodes로부터 나온다.

그래서 backup diagram 의 bottom nodes 는 이 모든 actions nodes 여야 한다.
마지막으로, 이 “next action” nodes 들에서 maximum 을 취하는 걸 arc 로 표현했던 걸 기억하렴.
이제 diagram 을 추측할 수 있겠니?
만약 그렇다면, 답을 보기 전에 한 번 그려보렴.

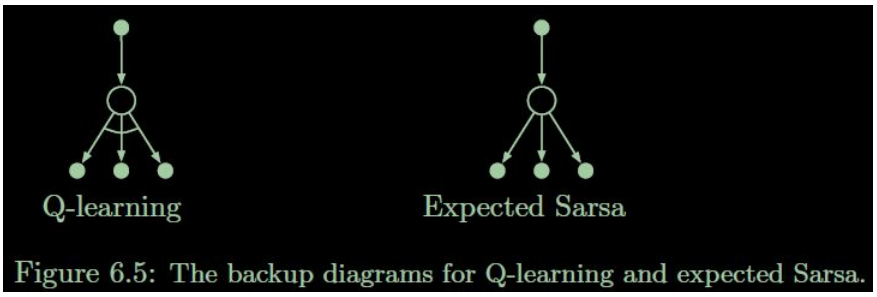


Figure 6.5: The backup diagrams for Q-learning and expected Sarsa.

(Example 6.6) Cliff Walking
이 gridworld example 은 Sarsa 와 Q-learning 을 비교해 본다.
on-policy (Sarsa) 와 off-policy (Q-learning) methods 의 차이점을 부각시켜서.

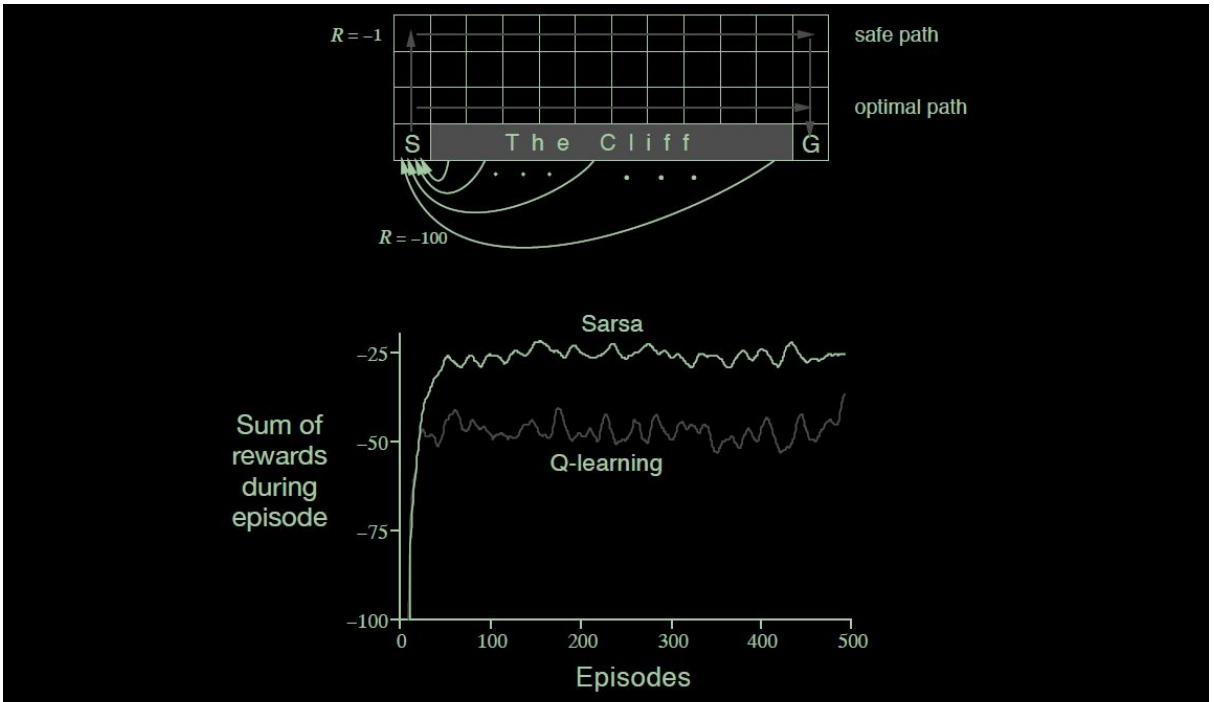


Figure 6.4: The cliff-walking task. The results are from a single run, but smoothed by averaging the reward sums from 10 successive episodes.

(Fig 6.4) 의 위 gridworld 를 볼까.
이건 standard undiscounted, episodic task 야. start 와 goal states 가 있고 상하좌우가 있지.
Reward 는 모든 transitions 에서 -1 이야. Cliff 로 가는 거 빼고.
여기로 들어가면 -100 reward 를 받고 agent 는 start 로 바로 돌아간다.
아래 파트는 Sarsa 와 Q-learning methods 의 performance야. ε = 0.1 인 ε-greedy action selection 을 하고.
initial transient 이후, Q-learning 은 optimal policy의 values 를 학습한다. cliff 의 가장자리를 따라가는거지.
하지만, 가끔 cliff 로 떨어져. ε-greedy action selection 때문에.

Sarsa는 action selection 까지 고려해서 멀지만 더 안전한 위쪽 길을 학습해.
Q-learning 이 실제 optimal policy 의 values 를 학습하긴 하지만
on-line performance는 돌아가는 policy 를 학습하는 Sarsa 보다 안 좋아.
당연히, ε 가 서서히 감소한다면, 두 methods 다 점근적으로 optimal policy 로 수렴할 거야.

(Exercise 6.11)
Q-learning 이 왜 off-policy control method니?

(Exercise 6.12)
action selection 이 greedy 라고 하자. 그러면 Q-learning 이 Sarsa 랑 같은가?
정확히 같은 action selections 과 weight updates 를 할까?

6.6 Expected Sarsa
Q-learning 에서, next state-action pairs의 maximum 대신 expected value를 쓰는 learning algorithm 을 생각해보자. current policy 하에서 각 action이 얼마나 likely 한지 고려하는겨.
그러니까,

$$\begin{aligned} Q(S_t, A_t) &\leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \mathbb{E}[Q(S_{t+1}, A_{t+1}) \mid S_{t+1}] - Q(S_t, A_t)\Big] \\ &\leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q(S_{t+1}, a) - Q(S_t, A_t)\Big], \end{aligned} \tag{6.9}$$

이런 update rule 을 가진 알고리즘을 생각하는거야. 나머지는 Q-learning의 schema를 따르고.
next state, S_{t+1} 이 주어졌을 때, 이 알고리즘은 deterministically 하게 움직인다. Sarsa 가 expectation 으로 움직이는 것과 같은 방향으로.
그래서 Expected Sarsa 라고 부른다.
backup diagram 은 Fig 6.5에 있지.

Expected Sarsa 가 Sarsa보다 계산적으로 더 복잡하지만,
대신 A_{t+1}의 random selection으로 인한 variance를 eliminate 한다.
같은 양의 experience가 주어졌을 때, Sarsa보다 이게 조금 더 낫길 기대할거다.
(Fig 6.6)은 cliff-walking task 에 Expected Sarsa를 적용했을 때의 결과를 Sarsa와 Q-learning 과 비교하며 요약했다.
게다가, Expected Sarsa 는 Sarsa 에 비해 광범위한 step-size parameter α 에서 많은 성능 향상을 보여준다.
cliff walking 에서 state transitions 은 모두 deterministic 이고 모든 randomness 는 policy 로부터 나온다.
그런 cases 에서, Expected Sarsa 는 asymptotic performance의 degradation 없이 안전하게 α=1 로 설정할 수 있다.
반면 Sarsa 는 작은 α 값으로만 long run 에서 좋은 성능을 발휘한다. 근데 그건 short-term 에서 성능이 구리지.
이거나 다른 예시에서, Expected Sarsa 가 Sarsa 보다 일관적으로 empirical(실증적인) 장점이 있다.

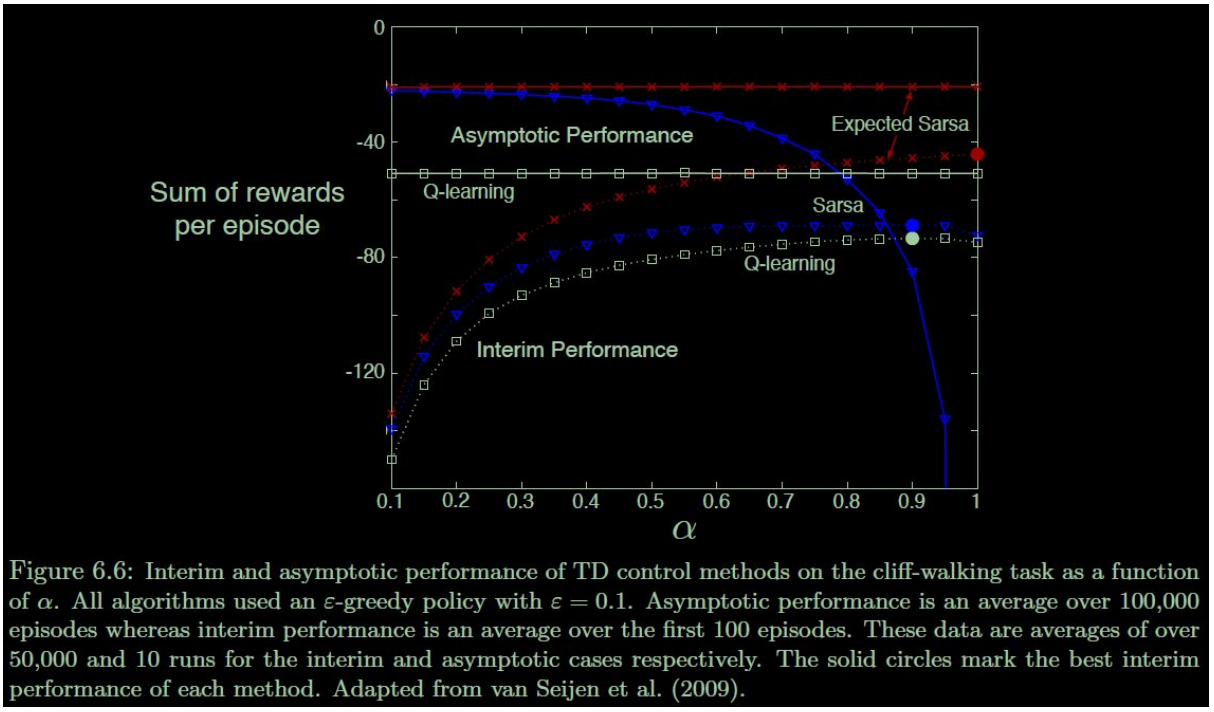


Figure 6.6: Interim and asymptotic performance of TD control methods on the cliff-walking task as a function of α. All algorithms used an ε-greedy policy with $\varepsilon = 0.1$. Asymptotic performance is an average over 100,000 episodes whereas interim performance is an average over the first 100 episodes. These data are averages of over 50,000 and 10 runs for the interim and asymptotic cases respectively. The solid circles mark the best interim performance of each method. Adapted from van Seijen et al. (2009).

(Fig 6.6): TD control methods 를 cliff-walking 에 적용한 중간 & 점근적 성능. α의 함수.
모든 알고리즘이 ε = 0.1 의 ε-greedy policy 를 썼다.
점근적 성능은 100,000 episodes 의 평균이고, interim performance는 첫 100 episodes 의 성능이다. 이 데이터는 50,000 interim & 10 asymptotic runs 을 각각 평균했다.
동그라미는 각 method 에서 best interim performance이다.
■

이 cliff walking 결과에서, Expected Sarsa 는 on-policy 를 썼다.
하지만 일반적으로 target policy π 와 다른 policy 를 쓸 거다. behavior 를 generate 하기 위해.
그러면 off-policy 알고리즘이 되는거지.
예를 들어, π가 greedy policy 이고 behavior 는 더 exploratory 하다고 하자.
그러면 Expected Sarsa 가 정확히 Q-learning 이다.
이런 면에서 Expected Sarsa 는 Q-learning을 포함하고 일반화한다. Sarsa를 안정적으로 향상시키면서.
계산상의 비용 조금 추가하는 거 외로는 Expected Sarsa가 다른 잘 알려진 TD control algorithms 둘을 완전히 dominate 한다.

6.7 Maximization Bias and Double Learning

우리가 지금까지 봤던 모든 control algorithms 은 그들의 target policies 구조에 maximization 을 포함하고 있다.
예를 들어, Q-learning 에서 target policy 는 주어진 current action values 에서 greedy policy이고, max 로 정의되지.
Sarsa 에선, policy 는 주로 ε-greedy 이고, 마찬가지로 maximization operations 을 포함한다.
이런 알고리즘에선, estimated values 에 대해 maximum 이 maximum value 의 estimate 로 implicitly 쓰이고 있다. 근데 이게 positive bias 로 이어질 수 있다네?
왜인지 보기 위해, single state s 를 생각해보자.

actions a 가 많은데 그 true values 인 $q(s, a)$ 가 전부 0이야.
근데 그 estimated values $Q(s, a)$ 는 uncertain 이라 0 위아래로 분포돼 있어.
true values 의 maximum 은 0인데, estimates 의 maximum 은 positive야.
positive bias 지.
이걸 maximization bias 라 불러.

(Example 6.7) Maximization Bias Example
(Fig 6.7) 의 작은 MDP 는 maximization bias 가 어떻게 TD control algorithms 의 성능을 해치는지 보여주는 간단한 예시야.
MDP 에 두 개의 non-terminal states A 와 B가 있어.
episodes 는 항상 A에서 시작하고, left & right 두 가지 actions 중 하나를 선택해.
right action 은 곧장 terminal state 로 가고 reward 와 return 은 0이야.
left action 은 B로 가고 reward 0 이야.
근데 거기서 여러 가지 가능한 actions 이 있고, 그걸 하면 종료되면서 reward를 받는데, 그 reward는 N(-0.1, 1) 에서 나와.
그래서 left 로 시작하는 어떤 trajectory도 expected return 이 -0.1 이고,
A 에서 left 를 하는 건 항상 mistake 다.
그럼에도 불구하고, 우리의 control methods는 left 를 좋아할 거야.
maximization bias 는 B가 positive value 를 갖는 것처럼 보이게 하기 때문이지.
Fig 6.7을 보면, ε-greedy action selection 을 쓰는 Q-learning 이 초반에 left action 을 강하게 선호하도록 학습해.
asymptote 에서도 Q-learning 은 5% 이상 더 left action 을 취한다.
ε = 0.1, α = 0.1, γ =1 인 parameter setting 에서 optimal 인 때보다.

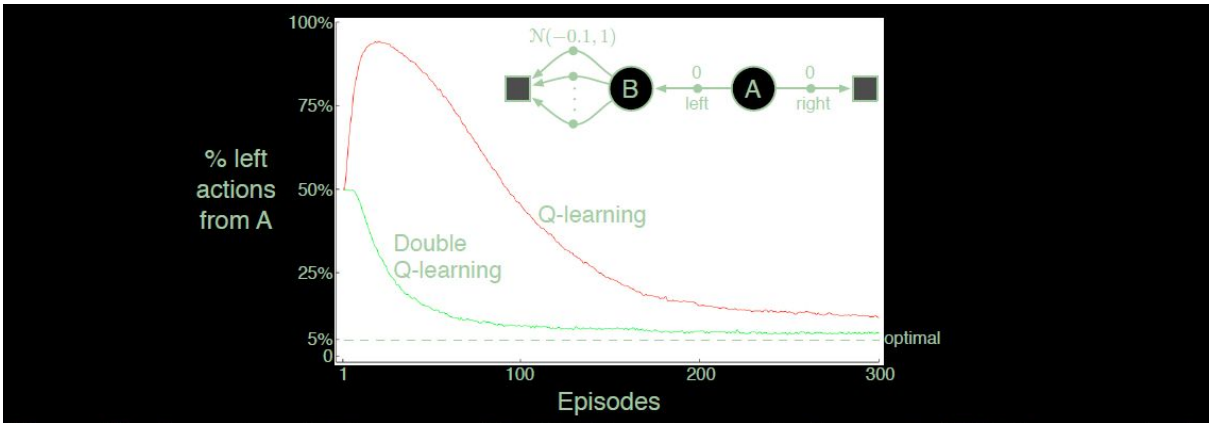


Figure 6.7: Comparison of Q-learning and Double Q-learning on a simple episodic MDP (shown inset). Q-learning initially learns to take the left action much more often than the right action, and always takes it significantly more often than the 5% minimum probability enforced by ε-greedy action selection with $\varepsilon = 0.1$. In contrast, Double Q-learning is essentially unaffected by maximization bias. These data are averaged over 10,000 runs. The initial action-value estimates were zero. Any ties in ε-greedy action selection were broken randomly.

(Fig 6.7) Q-learning 과 Double Q-learning 비교. simple episodic MDP 에서.
Q-learning 은 초반에 left action 을 right보다 훨씬 더 많이 하도록 학습한다.
그리고 항상 ε = 0.1 인 ε-greedy action selection 을 시행했을 때의 5% minimum 확률보다 훨씬 더 자주 left 를 고른다.
Double Q-learning에선 본질적으로 maximization bias 에 의해 영향을 받지 않는다.
이 data는 10,000 runs 에서 평균낸 거다.

initial action-value estimates 는 0이다.
∎

maximization bias 를 피하는 알고리즘이 있을까?
각 action 의 모든 plays 에서 받은 rewards 의 sample averages로 얻은 noisy estimates, 각 many actions 의 value에 대한 noisy estimates가 있는 bandit case 를 생각해볼까?
위에서 논의했듯, true values 의 maximum 에 대한 estimate 로 estimates의 maximum 을 쓰면 positive maximization bias 가 있을 거야.
문제를 보는 한 가지 방법은,
maximizing action 을 determine 하는 것과, 그 value를 estimate 하는 데에
같은 samples(plays) 을 쓰기 때문이라는 거야.
plays 를 two sets 로 나누고 두 independent estimates를 학습하는 데 썼다고 가정해보자.
걔들을 $Q_1(a)$, $Q_2(a)$ 로 부르고. 각각 true value $q(a)$ 의 estimate 이다.
그러면 한 estimate 를 쓸 수 있을거야. Q_1 이라고 할까? 이걸로 maximizing action A^* = $\text{argmax}_a Q_1(a)$ 를 determine 하는 데 쓰는거지.
Q_2 는 그 value의 estimate 를 제공하는 데 쓰고. $Q_2(A^*) = Q_2(\text{argmax}_a Q_2(a))$.
이 estimate 는 unbiased 일 거야.
$\mathbb{E}[Q_2(A^*)] = q(A^*)$ 인 측면에서.
두 estimates 의 역할을 바꿔서 두 번째 unbiased estimate Q_1 ($\text{argmax}_a Q_2(a)$)를 산출할 수도 있다.
이게 double learning 의 idea.
우리가 두 estimates 를 학습했어도, 각 play에서 한 estimate 만 update 된다.
double learning 은 memory 가 두 배 더 필요하지만, step 별로 계산량이 늘어나진 않는다.

double learning 의 idea 는 full MDPs 에 대한 알고리즘으로 자연스럽게 확장된다.
예를 들어, Q-learning 과 유사한 double learning 알고리즘, Double Q-learning 은
각 step마다 동전을 던져 time steps 을 둘로 나눈다.
앞면이면, update가

$$Q_1(S_t, A_t) \leftarrow Q_1(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma Q_2\big(S_{t+1}, \operatorname*{arg\,max}_a Q_1(S_{t+1}, a)\big) - Q_1(S_t, A_t)\Big]. \qquad (6.10)$$

이렇게 되고,
뒷면이면 Q_1, Q_2 를 바꿔서 같은 update 를 한다. 그래서 Q_2 가 update 되지.
두 approximate value functions 을 완전히 symmetrically 다룬다.
behavior policy 는 두 action-value estimates 모두 사용할 수 있다.
예를 들어, Double Q-learning 에 대한 ε-greedy policy 는 두 action-value estimates 의 평균이나 합에 기반할 수 있다.

Double Q-learning 의 전체 알고리즘은 아래 박스.
Fig 6.7에 나온 결과도 이 알고리즘을 썼다.
그 예제에서 double learning 은 maximization bias 에 의한 harm 을 없애는 것처럼 보였지.
물론 Sarsa 나 Expected Sarsa 도 double versions이 있다.

Double Q-learning, for estimating $Q_1 \approx Q_2 \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q_1(s, a)$ and $Q_2(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q.(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using the policy ε-greedy in $Q_1 + Q_2$
    Take action A, observe R, S'
    With 0.5 probabilility:
      $Q_1(S, A) \leftarrow Q_1(S, A) + \alpha\Big(R + \gamma Q_2\big(S', \arg\max_a Q_1(S', a)\big) - Q_1(S, A)\Big)$
    else:
      $Q_2(S, A) \leftarrow Q_2(S, A) + \alpha\Big(R + \gamma Q_1\big(S', \arg\max_a Q_2(S', a)\big) - Q_2(S, A)\Big)$
    $S \leftarrow S'$
  until S is terminal

박스:
$Q_1 \approx Q_2 \approx q_*$ 를 estimating 하는 Double Q-learning.
모든 $s \in \mathcal{S}^+$, $a \in \mathcal{A}(s)$ 에 대해 $Q_1(s, a)$, $Q_2(s, a)$ 를 arbitrarily initialize. 단, Q.(terminal, .) = 0
각 episode 마다 Loop:
  S 를 initialize.
  episode의 각 step 마다 Loop:
    S에서 A를 뽑는데, $Q_1 + Q_2$ 의 ε-greedy policy 를 이용해서.
    action A 하고 R, S' 를 observe.
    0.5 확률로:
      $Q_1(S, A) \leftarrow Q_1(S, A) + \alpha\ (\ R + \gamma Q_2\ (\ S', \text{argmax}_a\ Q_1(S', A)) - Q_1(S, A)\ )$
    else:
      $Q_2(S, A) \leftarrow Q_2(S, A) + \alpha\ (\ R + \gamma Q_1\ (\ S', \text{argmax}_a\ Q_2(S', A)) - Q_2(S, A)\ )$
    $S \leftarrow S'$
  S 가 terminal 될 때까지.

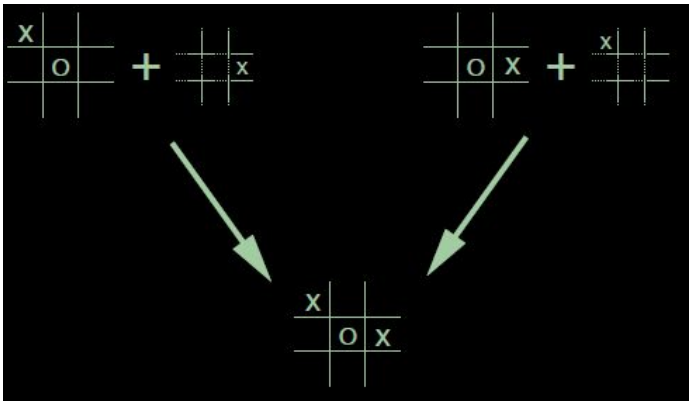
(Exercise 6.13)
ε-greedy target policy 를 쓰는 Double Expected Sarsa 의 update equations 은 뭘까?

6.8 Games, Afterstates, and Other Special Cases

이 책에서 a wide class of tasks 에 uniform approach 하는 걸 보여줬어.
근데 당연히 exceptional tasks 도 있겠지? specialized way 에서 better treated 되는?
예를 들어, 우리 general approach 는 action-value function 을 학습하는 걸 포함하지.
하지만 챕터 1에서 state-value function 과 같은 걸 학습하는 틱택토 를 play 하는 걸 학습하는 TD method 를 보여줬어.
그 예시를 잘 보면, 학습된 function 이 일반적인 의미에서 action-value function 도 아니고 state-value function 도 아니라는 걸 알 수 있다.
기존의 state-value function 은 agent 가 action 을 선택하는 옵션이 있는 states 를 evaluate 한다.
하지만 틱택토에서 쓰인 state-value function 은 agent 가 움직인 후 board positions 을 evaluate 한다.
이 afterstates 와 여기서의 value functions 을 afterstates value functions 라고 한다.

afterstates 는 environment 의 dynamics 의 initial part 에 대한 knowledge를 갖고 있을 때 유용하다. full dynamics 까진 필요없고.
예를 들어, games 에서 우린 보통 우리 moves 의 즉각적인 효과를 알고 있다.
우린 각 가능한 chess move 로 position이 어떻게 될 지 알지만 상대가 어떻게 reply 할지는 모른다.
afterstate value functions 은 이런 종류의 knowledge 의 장점을 이용하는 자연스러운 방법이다. 그로 인해 더 효과적인 learning method 를 생성한다.

afterstates 의 관점에서 알고리즘을 디자인하는 게 더 효과적인 이유는 틱택토를 보면 명백하다.
기존의 action-value function 은 positions & moves 를 value 의 estimate 로 매핑한다.
하지만 많은 position-move pairs 가 같은 결과의 position 을 생성한다. 아래 예시처럼.



그런 cases 에서 position-move pairs 는 서로 다르지만, 같은 afterposition 을 생성한다.
그래서 같은 value 를 가져야 한다.
기존의 action-value function 은 두 pairs 를 각각 assess(평가)해야 할 거다.
반면 afterstate value function 은 둘을 즉시 동등하게 평가할 거다.
왼쪽 position-move pair 에 대한 어떤 learning 도 오른쪽 pair 로 즉시 transfer 할 거다.

afterstates 는 games 뿐 아니라 많은 tasks 에서 발생한다.
예를 들어, queuing(대기열) tasks 에서 customers 를 servers 에 배정하거나, rejecting 하거나, discarding information 을 하는 actions 이 있다.
그런 cases 에서는, 사실 완전히 알고 있는, 즉각적인 effects 의 관점에서 actions 을 정의한다.

모든 specialized problems 와 그에 대응하는 specialized learning algorithms 을 다 묘사할 순 없다.
하지만, 이 책에서 develop 된 principles 은 widely apply 돼야한다.
예를 들어, afterstate methods 는, policy 와 (afterstate) value function 이 본질적으로 같은 방법으로 상호작용하면서 generalized policy iteration 의 관점에서 여전히 적절하게 묘사된다.
많은 cases 에서, 지속적인 exploration 에 대한 need 를 managing 하기 위해 on-policy 와 off-policy 를 선택해야 한다.

(Exercise 6.14)
Jack’s Car Rental (ex 4.2) 를 afterstates 의 관점으로 reformulate 해보렴.

이런 게 왜 convergence 를 빠르게 할 것 같을까?

6.9 Summary

이 챕터에선 새로운 종류의 learning method, temporal-difference (TD) learning 을 배웠다.
그리고 이게 강화학습에 어떻게 적용될 수 있는지도 봤다.
평소와 같이, 전체 문제를 prediction & control problem 으로 나눴다.
TD methods 는 prediction problem 을 푸는 데 Monte Carlo methods 의 대안이다.
두 cases 모두에서, control problem 으로의 확장은 dynamic programming 에서 abstract 한 GPI 의 idea 에 의해 이루어진다.
이건 approximate policy 와 value functions 이 둘 다 optimal values 로 가도록 상호작용해야한다는 idea 이다.

GPI 를 구성하는 두 프로세스 중 하나가 value function이 current policy 에 대한 returns 을 정확하게 predict 하도록 한다. 이게 prediction problem 이다.
다른 프로세스는 current value function에 대해 policy 가 locally improve (예를 들어 ε-greedy 하도록) 하게 한다.
첫 프로세스가 experience 에 기반했을 때, 충분한 exploration 을 유지하는 데에서 복잡성이 발생한다.
이 복잡성을 on-policy 로 처리하는지, off-policy 로 처리하는지에 따라 TD control methods 를 분류할 수 있다.
Sarsa 는 on-policy method 이고, Q-learning 은 off-policy method 이다.
Expected Sarsa 도 off-policy method 이다.
TD methods 를 control 로 확장하는 actor-critic methods 도 있는데 챕터13에서 얘기해줄게.

이 챕터에서 소개한 methods 들이 가장 널리 쓰이는 강화학습 방법들이다.
아마 그들의 simplicity 때문이리라.
최소한의 계산량으로, environment 와의 상호작용으로 생성된 experience에 on-line 을 적용할 수 있다.
작은 컴퓨터 프로그램으로 구현될 수 있는 single equations 으로 거의 완전히 표현될 수 있다.
다음 몇 챕터에서 이런 알고리즘들을 조금 더 복잡하게, 그리고 파워풀하게 만들어서 확장할거다.
모든 새 알고리즘들은 여기서 소개된 essence 들을 갖고 있을거야.
상대적으로 적은 계산으로 on-line 으로 experience 를 처리할 수 있을 거고,
TD errors 에 의해 driven 될 거다.

이 챕터에서 소개된 TD methods 의 special cases 는 one-step, tabular, model-free TD methods 라고 불려야 한다.
다음 두 챕터에서 n-step forms(a link to Monte Carlo methods) 로 확장하고, environment 의 model을 포함하는 forms(a link to planning and dynamic programming)로 확장한다.
책은 두 번째 파트에서 tables 대신 다양한 형태의 function approximation 으로 확장한다.(a link to deep learning and artificial neural networks)

마지막으로, 이 챕터에서 TD methods 를 완전히 강화학습 문제 내에서 논의했다.

하지만 TD methods 는 이것보다 더 일반적이지.
그들은 dynamical systems 에 대해 long-term predictions 을 하는 걸 학습하는 일반적인 methods 다.
예를 들어, TD methods 는 financial data, life spans, election outcomes, weather patterns, animal behavior, demands on power stations, customer purchase 를 예측하는 데 관련이 있을 수 있다.
TD methods 가 강화학습에서의 사용과는 독립적으로, 순수하게 prediction methods 로 분석됐을 때만 이론적 특성이 잘 이해된다.
그렇다 하더라도, TD learning methods 의 다른 잠재적인 applications 은 아직 다 탐구되지 않았다.

$\gamma r\varepsilon\varrho\lambda\alpha\delta$
$\pi(a|s)$
$v_\pi(s)$ $Q(s, a)$
$q_\pi(s, a)$
$ACSQRTVWGtabpnij\boldsymbol{h}$
$\mathcal{ASP}$
$\ni\in\geq\leq\neq\Leftrightarrow\longleftarrow$ $s\in\mathcal{S}$, $a\in\mathcal{A}(s)$
$\blacksquare'$
$\varrho_{r.T(t)\text{-}1}$