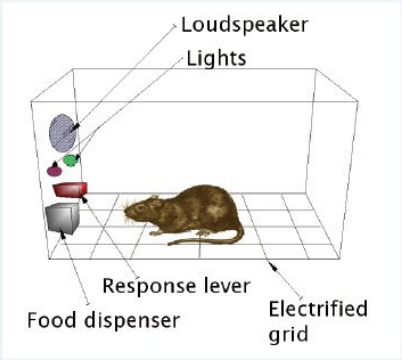
강화 학습과 게임 지능

Preview

- 환경과 상호작용하며 살아가는 인간과 동물
  - 환경의 상태를 보고 자신에게 유리한 행동을 결정하고 실행
  - 행동에 따른 결과가 좋으면 기억했다가 반복하고, 결과가 나쁘면 회피
    - 예) 자전거 타기
    - 예) 스키너의 행동심리학 실험
    - 예) 바둑, 비디오 게임, ...
  - 행동 → 상태 변화 → 보상의 학습 사이클

(a) 어린이의 자전거 배우기



(b) 스키너 상자(출처: 위키백과)

(c) 학습 사이클

그림 9-1 '강화'를 통한 학습 과정

Preview

- 컴퓨터로 이런 방식의 학습을 할 수 있을까?
  - 지금까지 공부한 지도 학습(다층 퍼셉트론, 컨볼루션 신경망, 순환 신경망)은 부적절함
  - 강화 학습이 대안
  - 강화 학습의 성공 사례
    - 알파고
    - 인간 성능을 능가하는 인공지능 비디오 게임
    - 알파스타(인공지능 스타크래프트)

9.1 강화 학습의 원리와 응용

- 강화 학습의 성공 사례
  - 이세돌을 이긴 알파고
  - 스타크래프트 0.2% 수준의 플레이어가 된 알파스타
  - 로봇이나 자율 주행차 제어 등

- 이 절에서는 실제 문제를 가지고 강화 학습 공부를 시작
  - 가장 단순한 문제인 다중 손잡이 밴딧 문제를 다룸
  - OpenAI 재단이 제공하는 gym 라이브러리 소개(gym은 다양한 문제를 제공)

9.1.1 다중 손잡이 밴딧 문제

다중 손잡이 밴딧 문제

- $1을 넣고 여러 손잡이 중 하나를 골라 당기면 $1을 잃거나 땀
- 손잡이마다 승률이 정해져 있는데 사용자는 확률을 모름
- 행동의 집합={손잡이1, 손잡이2, 손잡이3, 손잡이4, 손잡이5}
- 보상의 집합={-1,1}

그림 9.2 다중 손잡이 밴딧

- 행동 → 상태 변화 → 보상의 학습 사이클에서 상태가 없는 단순한 문제
  - 행동 → 보상 사이클

9.1.1 다중 손잡이 밴딧 문제

- 탐험형 정책과 탐사형 정책
  - 양 극단
    - 처음부터 끝까지 무작위로 선택하는 극단적인 탐험형 정책exploration policy
    - 몇 번 시도해보고 이후에는 그때까지 승률이 가장 높은 손잡이만 당기는 극단적인 탐사형 정책exploitation policy
  - 둘 사이의 균형이 중요함
    - 현재까지 높은 확률을 보인 손잡이를 더 자주 당기지만 일정한 비율로 다른 손잡이도 시도하는 정책
- 에피소드가 충분히 긴 경우는 해법이 단순
  - 강화 학습에서는 게임을 시작하여 마칠 때까지 기록을 에피소드episode라 부름
  - 충분히 긴 에피소드로부터 승률을 계산하여 이후에는 승률이 가장 높은 손잡이만 당김
    - 에피소드가 충분히 길다면 '최적 정책 ' 을 알아낼 수 있음
  - 하지만 세상은 그리 단순하지 않음
    - 주인이 수시로 확률을 바꾼다거나 돈과 시간이 충분하지 않은 경우가 태반

9.1.1 다중 손잡이 밴딧 문제

- 랜덤 정책을 쓰는 알고리즘을 구현한 [프로그램 9-1(a)]

프로그램 9-1(a) 다중 손잡이 밴딧 문제를 위한 랜덤 정책

```
01  import numpy as np
02
03  # 다중 손잡이 밴딧 문제 설정
04  arms_profit=[0.4, 0.12, 0.52, 0.6, 0.25]
05  n_arms=len(arms_profit)
06
07  n_trial=10000                    # 손잡이를 당기는 횟수(에피소드 길이)
08
09  # 손잡이 당기는 행위를 시뮬레이션하는 함수(handle은 손잡이 번호)
10  def pull_bandit(handle):
11      q=np.random.random()
12      if q<arms_profit[handle]:
13          return 1
14      else:
15          return -1
16
```

손잡이의 승률(플레이어에게는 감춰져 있음)

9.1.1 다중 손잡이 밴딧 문제

```
17  # 랜덤 정책을 모방하는 함수
18  def random_exploration():
19      episode=[]
20      num=np.zeros(n_arms)        # 손잡이별로 당긴 횟수
21      wins=np.zeros(n_arms)       # 손잡이별로 승리 횟수
22      for i in range(n_trial):
23          h=np.random.randint(0,n_arms)
24          reward=pull_bandit(h)
25          episode.append([h,reward])
26          num[h]+=1
27          wins[h]+=1 if reward==1 else 0
28      return episode, (num,wins)
29
30  e,r=random_exploration()
31
32  print("손잡이별 승리 확률:", ["%6.4f"% (r[1][i]/r[0][i]) if r[0][i]>0 else 0.0 for i
    in range(n_arms)])
33  print("손잡이별 수익($):",["%d"% (2*r[1][i]-r[0][i]) for i in range(n_arms)])
34  print("순 수익($):",sum(np.asarray(e)[:,1]))
```

```
손잡이별 승리 확률: ['0.4095', '0.0925', '0.5347', '0.6021', '0.2525']
손잡이별 수익($): ['-42', '-141', '14', '39', '-100']
순 수익($): -230
```

프로그램이 추정한 승률
최적 정책은 [0,0,0,1,0]

230달러를 잃음

9.1.1 다중 손잡이 밴딧 문제

ε-탐욕 알고리즘

- 탐욕 알고리즘greedy algorithm
  - 과거와 미래를 전혀 고려하지 않고 현재 순간의 정보만 가지고 현재 최고 유리한 선택을 하는 알고리즘 방법론
  - 탐사형에 치우친 알고리즘
- ε-탐욕 알고리즘은 기본적으로 탐욕 알고리즘인데, ε 비율만큼 탐험을 적용하여 탐사와 탐험의 균형을 추구

9.1.1 다중 손잡이 밴딧 문제

- ε-탐욕을 구현한 [프로그램 9-1(b)]

프로그램 9-1(b) 다중 손잡이 밴딧 문제를 위한 ε-탐욕 알고리즘

```
35
36  # ε-탐욕을 구현하는 함수
37  def epsilon_greedy(eps):
38      episode=[]
39      num=np.zeros(n_arms)          # 손잡이별로 당긴 횟수
40      wins=np.zeros(n_arms)         # 손잡이별로 승리 횟수
41      for i in range(n_trial):
42          r=np.random.random()
43          if(r<eps or sum(wins)==0):  # 확률 eps로 임의 선택
44              h=np.random.randint(0,n_arms)
45          else:
46              prob=np.asarray([wins[i]/num[i] if num[i]>0 else 0.0 for i in range(n_arms)])
47              prob=prob/sum(prob)
48              h=np.random.choice(range(n_arms),p=prob)
49          reward=pull_bandit(h)
50          episode.append([h,reward])
51           num[h]+=1
52           wins[h]+=1 if reward==1 else 0
53      return episode, (num,wins)
54
55  e,r=epsilon_greedy(0.1)
```

랜덤 정책으로 손잡이 선택

이때까지 추정한 확률에 따라 손잡이 선택

9.1.1 다중 손잡이 밴딧 문제

```
56
57   print("손잡이별 승리 확률:", ["%6.4f"% (r[1][i]/r[0][i]) if r[0]
     [i]>0 else 0.0 for i in range(n_arms)])
58   print("손잡이별 수익($):",["%d"% (2*r[1][i]-r[0][i]) for i in range(n_arms)])
59   print("순 수익($):",sum(np.asarray(e)[:,1]))
```

```
손잡이별 승리 확률: ['0.4423', '0.0667', '0.5142', '0.5952', '0.2222']
손잡이별 수익($): ['-24', '-26', '8', '64', '-80']
순 수익($): -58
```

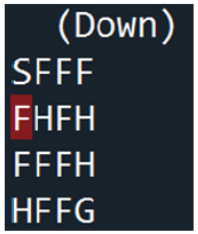
랜덤 정책보다 높은 수익

9.1.1 다중 손잡이 밴딧 문제

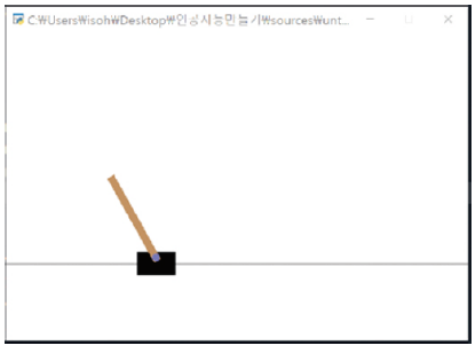
- 몬테카를로 방법
  - 현실 세계의 현상 또는 수학적 현상을 난수를 생성하여 시뮬레이션 하는 기법
  - [프로그램 9-1(a)]의 random_exploration 함수와 [프로그램 9-1(b)]의 epsilon_greedy 함수는 몬테카를로 방법임
  - 인공지능은 다양한 목적으로 몬테카를로 방법 활용. 예) 11.6절의 몬테카를로 트리 탐색

9.1.2 OpenAI의 gym 라이브러리

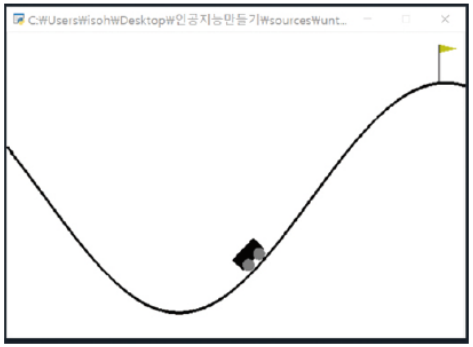
- gym 라이브러리(https://gym.openai.com)
  - OpenAI 재단이 만들어 배포하는 라이브러리로서 여러 강화 학습 문제를 제공



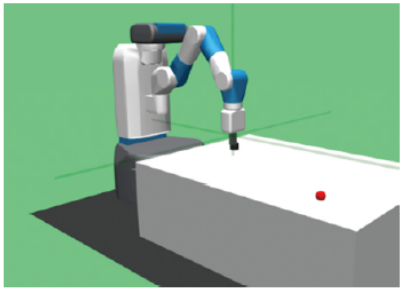
(a) FrozenLake 문제



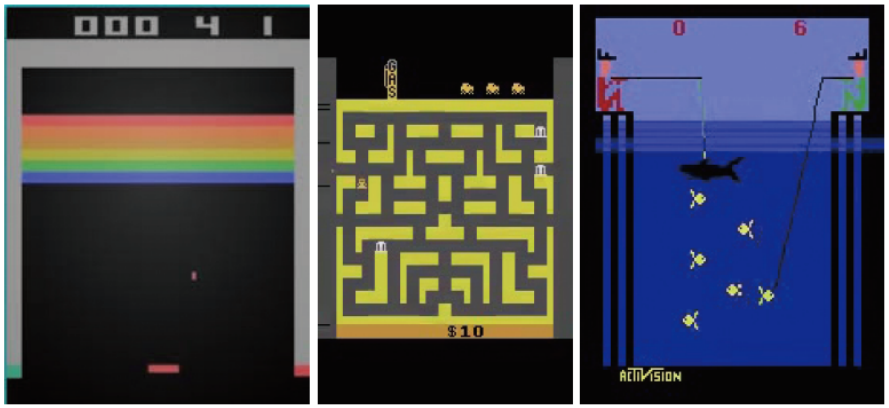
(b) CartPole 문제



(c) MountainCar 문제



(d) FetchSlide 문제



(e) 아타리 게임

그림 9-3 gym 라이브러리가 제공하는 여러 가지 문제

9.1.2 OpenAI의 gym 라이브러리

FrozenLake 문제

- 단순하지만 강화 학습의 개념을 설명하는데 유익한 문제
- S에서 시작하여 G에 도착하면 이기는 게임
- F는 얼어 있어 밟고 지날 수 있으나 H는 구멍이라 빠지면 짐(H와 F는 감추어져 있음)
- 현재 있는 곳이 상태(16가지 상태 {0,1,...,15}), 좌우상하 이동이 행동(4가지 행동 {Left, Down, Right, Up})

(a) FrozenLake-v0 환경 (b) 16개의 상태 (c) 네 가지 행동

그림 9-4 FrozenLake-v0 환경의 상태와 행동

9.1.2 OpenAI의 gym 라이브러리

- FrozenLake 문제를 살펴보기 위한 [프로그램 9-2]

프로그램 9-2 gym 라이브러리의 FrozenLake 문제

```
01  import gym
02
03  # 환경 불러오기
04  env=gym.make("FrozenLake-v0",is_slippery=False)
05  print(env.observation_space)
06  print(env.action_space)
07
08  n_trial=20
09
10  # 에피소드 수집
11  env.reset()
12  episode=[]
13  for i in range(n_trial):
14      action=env.action_space.sample()        # 행동을 취함(랜덤 선택)
15      obs,reward,done,info=env.step(action)       # 보상을 받고 상태가 바뀜
16      episode.append([action,reward,obs])
17      env.render()
18      if done:
19          break
20
21  print(episode)
22  env.close()
```

step 함수는 매개변수 action이 지정한 행동을 수행하고
(새로운 상태,보상,에피소드 끝 여부,부가정보)를 반환

9.1.2 OpenAI의 gym 라이브러리

```
Discrete(16)
Discrete(4)
```


```
[[2, 0.0, 1], [0, 0.0, 0], [1, 0.0, 4], [2, 0.0, 5]]
```

실행 결과의 해석
(Right→Left→Down→Right 순으로 행동을 취하고 H에 빠져 게임에 짐)

그림 9-5 [프로그램 9-2]를 실행하여 얻은 에피소드 사례(난수를 사용하므로 실행할 때마다 결과 다름)

9.1.3 계산 모형

- 마르코프 결정 프로세스(MDP Markov decision process)
  - 상태의 종류, 행동의 종류, 보상의 종류를 지정하고, 행동을 취했을 때 발생하는 상태 변환을 지배하는 규칙을 정의

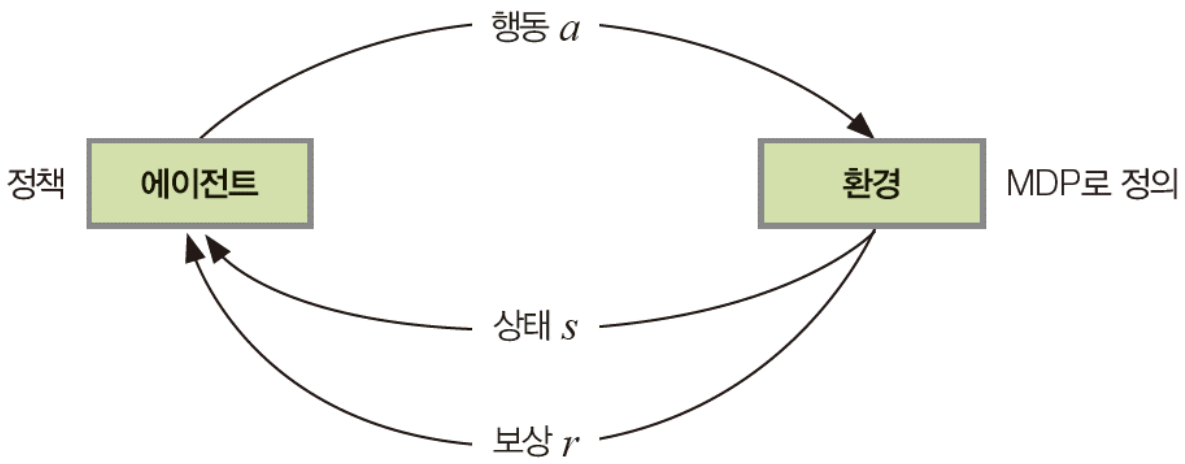


그림 9-6 강화 학습의 계산 모형

9.1.3 계산 모형

- 상태와 환경, 보상

$$\begin{cases} \text{상태 집합: } \mathcal{S}=\{s_1,s_2,\cdots,s_l\} \\ \text{행동 집합: } \mathcal{A}=\{a_1,a_2,\cdots,a_m\} \\ \text{보상 집합: } \mathcal{R}=\{r_1,r_2,\cdots,r_n\} \end{cases} \quad (9.1)$$

  - 보상이 주어지는 시점
    - 즉시 보상: 예) 다중 손잡이 밴딧
    - 지연된 보상: 예) FrozenLake, 바둑, 장기, 비디오 게임 등(대부분 문제가 지연 보상)

9.1.3 계산 모형

[예제 9-1] 상태와 행동, 보상

[그림 9-2]의 다중 손잡이 밴딧 문제를 식 (9.1)에 대입하면 다음과 같다. 앞에서 설명한 바와 같이 상태는 없다. $\varnothing$는 공집합을 뜻한다. 손잡이가 5개인 기계이므로 손잡이 5개가 각각 행동에 해당한다. 프로그래밍할 때는 0, 1, 2, 3, 4로 구분하면 된다. 보상은 돈을 따거나($1), 잃는 것($-1)이다.

$$\text{다중 손잡이 밴딧} \begin{cases} \text{상태 집합: } \mathcal{S}=\varnothing \\ \text{행동 집합: } \mathcal{A}=\{\text{손잡이}0, \text{손잡이}1, \text{손잡이}2, \text{손잡이}3, \text{손잡이}4\} \\ \text{보상 집합: } \mathcal{R}=\{1,-1\} \ \text{(즉시 보상)} \end{cases}$$

[그림 9-4]의 FrozenLake 문제를 식 (9.1)에 대입하면 다음과 같다. 각각의 칸이 상태에 해당하여 16개 상태가 있다. 상하좌우로 이동할 수 있으므로 4개의 행동이 있다. 보상은 목표 지점에 도달하면 1이고 나머지는 0이다.

$$\text{FrozenLake} \begin{cases} \text{상태 집합: } \mathcal{S}=\{0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15\} \\ \text{행동 집합: } \mathcal{A}=\{0(\text{Left}), 1(\text{Down}), 2(\text{Right}), 3(\text{Up})\} \\ \text{보상 집합: } \mathcal{R}=\{0,1\} \ \text{(지연된 보상)} \end{cases}$$

9.1.3 계산 모형

상태 전이

- 결정론적 환경(100% 확률로 새로운 상태가 정해지는 환경)
  - 예, FrozenLake 문제에서 Right를 선택하면 100%확률로 오른쪽으로 이동

$$P(s'=2, r=0 | s=1, a=2) = 1.0$$

상태 1에서 (s=1) 행동 2를 취하면 (a=2)
새로운 상태 2로 전환하고 (s'=2) 보상이 0일 (r=0) 확률이 1

- 스토캐스틱 환경(확률 분포에 따라 새로운 상태가 다르게 정해지는 환경)
  - [프로그램 9-2]의 04행에서 is_slippery를 True로 설정하면 스토케스틱 환경이 됨
    - Right를 선택해도 일정한 확률로 오른쪽으로 이동하지 않을 수 있음
    - 실제 세계에서 발생하는 바람 또는 얼음 위의 미끄러짐을 흉내 냄

$$\text{상태 전이 확률 분포: } P(s', r | s, a), \forall s \in \mathcal{S}, \forall a \in \mathcal{A}, \forall r \in \mathcal{R} \qquad (9.2)$$

- 바둑이나 장기, 비디오 게임 등은 모두 결정론적 환경
- 이 책은 결정론적 환경만 다룸

9.2 최적 정책과 가치 함수

- 9.1절은 강화 학습이 풀어야 하는 문제를 설명
- 이제는 문제를 어떻게 풀 것인지에 대한 설명

9.2.1 강화 학습과 지도 학습의 비교

- 여러 측면에서 다름

당장은 손해가 나더라도 전체 과정에 걸쳐 누적 보상액을 최대화해야 함(예, 바둑에서 작은 집을 희생하여 최종적으로 더 많은 집을 확보)

표 9-1 강화 학습과 지도 학습 비교

	지도 학습(신경망)	강화 학습
문제(데이터)	훈련 집합 X(특징 벡터)와 Y(레이블)	환경(식 (9.2)의 상태 전이 확률 분포) 또는 환경에서 수집한 데이터(에피소드)
최적화 목표	신경망 출력 **o**와 레이블 **y**의 오차인 $\|\|\mathbf{o}-\mathbf{y}\|\|$ 최소화	누적 보상 최대화
학습 알고리즘이 알아내야 하는 것	오차를 최소화하는 신경망의 가중치	누적 보상을 최대화하는 최적 정책
품질을 평가하는 함수	손실 함수	가치 함수
학습 알고리즘	스토케스틱 그레이디언트 하강법 (SGD)	동적 프로그래밍, Sarsa, Q 러닝, DQN 등

9.2.2 최적 정책

- 학습 알고리즘이 해야 할 일
  - '누적 보상'을 최대화하는 '최적 정책'을 알아내야 함
- 최적 정책이란?
  - 예) [그림 9-2]의 다중 손잡이 밴딧에서 승률이 가장 높은 4번 손잡이를 당기는 정책

$$\text{다중 손잡이 밴딧([그림 9-2])}: P(a=0|.)=0, P(a=1|.)=0, P(a=2|.)=0, P(a=3|.)=1, P(a=4|.)=0$$

  - 예) [그림 9-4]의 Fronzenlake에서 상태 4에서 행동 1을 취해 안전한 길을 찾음

$$\text{FrozenLake([그림 9-4])}: \cdots, \cdots, P(a=1|s=4)=1, \cdots, \cdots, \cdots)$$

9.2.2 최적 정책

- 확률 분포로 표현되는 정책

$$\text{정책} : \pi(a|s)=P(a|s), \forall s \in \mathcal{S}, \forall a \in \mathcal{A} \qquad (9.3)$$

  - P(a=i|s=j)는 상태 변수 s가 j라는 값일 때 행동 변수 a가 i라는 값을 취할 확률

[예제 9-2] 최적 정책

[그림 9-2]의 다중 손잡이 밴딧의 경우 승률이 최대인 손잡이만 당겨야 누적 보상을 최대화할 수 있기 때문에 최대 승률인 손잡이의 확률을 1로 설정한 확률 분포가 최적 정책이다. $\hat{\pi}$은 가능한 모든 정책 중에서 가장 좋은 정책, 즉 최적 정책을 뜻한다.

$$\hat{\pi} : P(a=0|.)=0, P(a=1|.)=0, P(a=2|.)=0, P(a=3|.)=1, P(a=4|.)=0$$

9.2.2 최적 정책

[그림 9-4]의 FrozenLake에서 최적 정책은 다음과 같다. 맨 위 행은 0번 칸(출발점)에서 행동을 선택할 확률로, 1(Down)과 2(Right)에 0.5씩 배분하였다. 따라서 여러 번 실행하면 오른쪽으로 가는 에피소드와 아래쪽으로 가는 에피소드가 반반씩 생성된다. 1(Down)에만 1을 배정하는 정책도 최적인데, 이렇게 하면 항상 밑으로 내려가는 에피소드만 생성된다. 누적 보상을 최대화한다는 목표로 보면 둘 다 최적 정책이다.

두 번째 행은 칸 1에서의 확률인데, P(2|1)=1이므로 항상 오른쪽으로 간다. 마지막 행은 14번 칸의 확률인데, P(2|14)=1이므로 항상 오른쪽으로 이동해 목표 지점에 도달한다.

$$\hat{\pi} = \begin{cases} P(0|0)=0, P(1|0)=0.5, P(2|0)=0.5, P(3|0)=0 \\ P(0|1)=0, P(1|1)=0, P(2|1)=1, P(3|1)=0 \\ P(0|2)=0, P(1|2)=1, P(2|2)=0, P(3|2)=0 \\ P(0|3)=1, P(1|3)=0, P(2|3)=0, P(3|3)=0 \\ P(0|4)=0, P(1|4)=1, P(2|4)=0, P(3|4)=0 \\ \vdots \\ P(0|14)=0, P(1|14)=0, P(2|14)=1, P(3|14)=0 \end{cases}$$

생략한 상태에 대한 확률을 쓰는 일은 연습문제로 남겨둔다.

9.2.3 가치 함수로 찾는 최적 정책

- 최적 정책을 찾는 학습 알고리즘의 필요성
  - [예제 9-2]에서는 최적 정책을 쉽게 찾았는데 문제가 단순하기 때문에 가능
  - 실용적인 문제에서는 학습 알고리즘이 필요(가치 함수를 통해 최적 정책을 찾는 전략)
- 가치 함수
  - 정책의 품질을 평가하는 함수

$$\text{가치 함수: } v_\pi(s), \forall s \in \mathcal{S} \qquad (9.4)$$

  - 학습 알고리즘이 해야 할 일은 $v_\pi(s)$를 최대화하는 최적 정책 $\hat{\pi}$를 찾기

$$\hat{\pi} = \underset{\pi}{\operatorname{argmax}}\, v_\pi(s), \forall s \in \mathcal{S} \qquad (9.5)$$

  - 식 (9.5)는 할 일을 정의. 이제부터 가치 함수를 계산하는 방법을 찾아야 함

9.2.3 가치 함수로 찾는 최적 정책

가치 함수의 계산

- 식 (9.6)은 가치 함수 계산을 위한 실마리 제공

$$v_\pi(s)=E(\mathbb{r}|s)=\Sigma_{s\text{에서 출발하는 모든 에피소드 }z}P(z)\mathbb{r}(z),\forall s\in\mathcal{S} \qquad (9.6)$$

[예제 9-3] 가치 함수의 계산

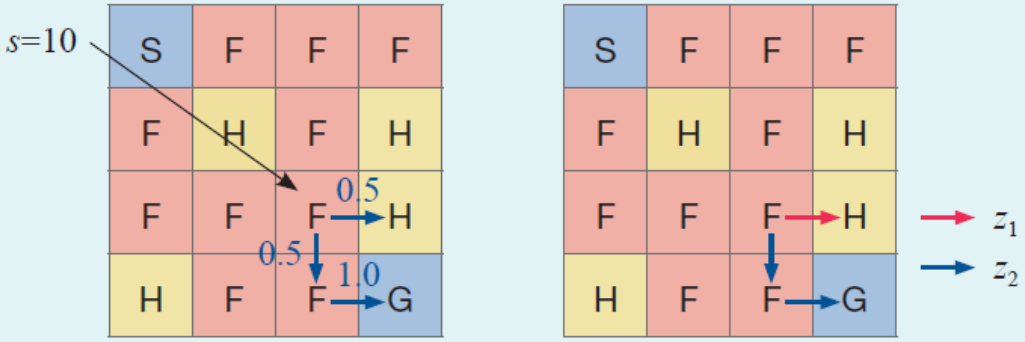
모든 칸에서 1(Down)과 2(Right) 행동을 0.5 확률로 선택하는 정책을 π_1이라 하고, 경계에 있는 칸에서는 바깥으로 나가는 이동을 허용하지 않는다고 가정하자. 따라서 칸 14에서는 $P(2|14)=1$이다.

$$\pi_1=\begin{cases} P(0|0)=0,P(1|0)=0.5,P(2|0)=0.5,P(3|0)=0 \\ P(0|1)=0,P(1|1)=0.5,P(2|1)=0.5,P(3|1)=0 \\ \vdots \\ P(0|10)=0,P(1|10)=0.5,P(2|10)=0.5,P(3|10)=0 \\ \vdots \\ P(0|14)=0,P(1|14)=0,P(2|14)=1,P(3|14)=0) \end{cases}$$

9.2.3 가치 함수로 찾는 최적 정책

[그림 9-7(a)]는 상태 s=10에 대해 가치 함수 $v_{\pi_1}(s=10)$을 계산하는 데 필요한 확률을 보여준다. [그림 9-7(b)]는 상태 s=10에서 가능한 에피소드 z_1과 z_2를 보여준다. 상태 s=10에서 가치 함수 값을 다음 공식으로 계산할 수 있다.

$$v_{\pi_1}(s=10)=v_{\pi_1}(10)=\sum_{z\in\{z_1,z_2\}}P(z)\mathbb{r}(z)=P(z_1)\mathbb{r}(z_1)+P(z_2)\mathbb{r}(z_2)$$



(a) 정책 π_1에 따른 확률 (b) 가능한 모든 에피소드

그림 9-7 상태 s = 10에서 가치 함수 계산

이제 $P(z_1)$과 $\mathbb{r}(z_1)$, $P(z_2)$와 $\mathbb{r}(z_2)$를 알면 $v_{\pi_1}(10)$을 계산할 수 있다. $P(z_1)$은 칸 10에서 행동 2를 취하여 칸 11로 갈 확률인데 정책 π_1에 따르면 $P(2|10)=0.5$이므로 $P(z_1)=0.5$이다. z_1은 구멍에 빠지게 되므로 $\mathbb{r}(z_1)$은 0이다. 비슷하게 $P(z_2)=0.5*1$이고 $\mathbb{r}(z_2)$는 목표 지점에 도달했으므로 1이다. 이 값들을 대입하면 다음과 같이 $v_{\pi_1}(10)$는 0.5이다.

$$v_{\pi_1}(10)=P(z_1)\mathbb{r}(z_1)+P(z_2)\mathbb{r}(z_2)=0.5*0+0.5*1.0*1=0.5$$

9.2.3 가치 함수로 찾는 최적 정책

최적 정책을 찾는 전략

- 신경망 학습과 마찬가지로 π_1으로 초기화한 다음, π_1을 π_2로 개선하고, π_2를 π_3로 개선하고, π_3를 π_4로 개선하고, ..., 결국 최적 정책 $\hat{\pi}$으로 수렴
- 예, [예제 9-3]의 π_1을 π_2로 바꾸면

$$\pi_2 = \begin{cases} \vdots \\ P(0|10)=0, P(1|10)=1, P(2|10)=0, P(3|10)=0 \\ \vdots \end{cases}$$

  - $v_{\pi 2}(10)=1$로서 $v_{\pi 1}(10)=0.5$보다 좋음

$$v_{\pi_2}(10)=P(z_2)\mathbb{r}(z_2)=1.0*1.0*1=1$$

9.2.4 가치 함수 계산을 위한 벨만 방정식

- 식 (9.6) 계산의 어려움
  - [예제 9-3]은 매우 작은 문제이므로 쉽게 계산
  - 실제 세계에는 1000*1000과 같은 아주 큰 격자를 가진 FozenLake 문제나 바둑과 같이 상태가 무한대에 가까운 문제
  - 식 (9.6)은 가치 함수를 계산하는 알고리즘을 설계하는데 쓸 중간 표현

- 벨만 기대 방정식
  - 상태는 서로 밀접한 관련성을 가짐
    - 예) FrozenLake([그림 9-8])의 경우 어떤 상태에서 이동할 수 있는 상태는 기껏 4개
  - 이런 특성을 이용한 벨만 기대 방정식Bellman expectation equation
    - 오른쪽 변에 자기 자신을 포함하는 순환식 형태
    - $\mathcal{A}(s)$는 상태 s에서 취할 수 있는 행동의 집합, P(a|s)는 상태 s에서 행동 a를 취할 확률, r은 상태 s에서 행동 a를 취했을 때 받는 보상, s′는 상태 s에서 a를 취했을 때 다음 상태

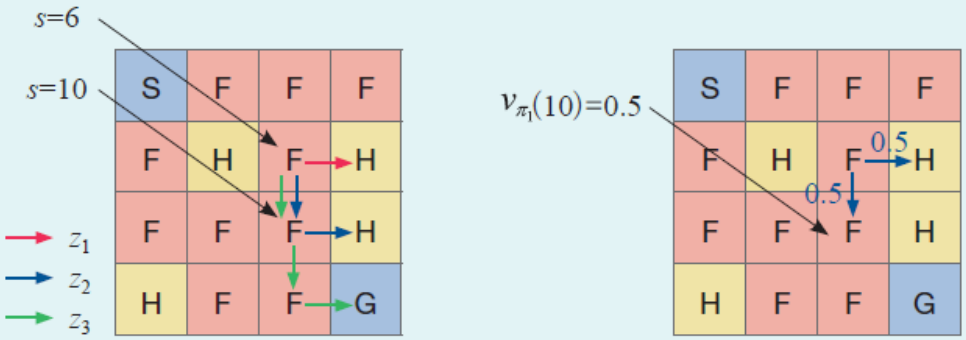
$$v_\pi(s) = \sum_{a \in \mathcal{A}(s)} P(a|s)\left(r + v_\pi(s')\right), \ \forall s \in \mathcal{S} \qquad (9.7)$$ 벨만 기대 방정식

9.2.4 가치 함수 계산을 위한 벨만 방정식

[예제 9-4] 식 (9.7)의 벨만 기대 방정식 적용

상태 s=6에서 가치 함수를 계산한다. [그림 9-8(a)]는 s=6에서 발생할 수 있는 에피소드 3개를 보여준다. 식 (9.6)을 사용한다면 [그림 9-7(a)]에 있는 에피소드 3개를 가지고 아래 식과 같이 가치 함수를 계산할 수 있다. 계산 결과 $v_{\pi_1}(6)$=0.25를 얻었다.

$$v_{\pi_1}(6)=P(z_1)\mathbb{r}(z_1)+P(z_2)\mathbb{r}(z_2)+P(z_3)\mathbb{r}(z_3)$$
$$=0.5\times 0+0.5\times 0.5\times 0+0.5\times 0.5\times 1.0\times 1=0.25$$



(a) s=6에서 발생하는 3개 에피소드 (b) 식 (9.7)을 이용해 s=6에서 가치 함수 계산

그림 9-8 벨만 기대 방정식을 이용해 s = 6에서 가치 함수 계산

이제 똑똑한 식 (9.7)을 사용하여 계산하자. 둘째 줄에서 순환하는 항을 파란색으로 표시해 강조하였다. 셋째 줄에서는 [예제 9-3]에서 계산해 놓은 $v_{\pi_1}(10)$=0.5를 대입하면 0.25를 얻는다. 모든 에피소드를 나열하는 식 (9.6)의 계산 결과와 같다.

$$v_{\pi_1}(6)=\sum_{a\in\{2(Right),1(Down)\}}P(a|6)\left(r+v_{\pi}(s')\right)$$
$$=P(2|6)\left(0+v_{\pi_1}(7)\right)+P(1|6)\left(0+v_{\pi_1}(10)\right)$$
$$=0.5(0+0)+0.5(0+0.5)=0.25$$

9.2.4 가치 함수 계산을 위한 벨만 방정식

할인율을 적용한 누적 보상액

- 누적 보상액$_{\text{accumulating reward}}$의 수학적 정의
  - 누적 보상액 $\mathbb{r}(z)$는 현재 순간 t에서 시작하여 에피소드가 끝날 때까지 발생한 보상의 총합
  - γ는 할인율$_{\text{discount rate}}$(미래에 발생하는 보상의 가치를 일정 비율로 삭감하는 역할)

$$\left.\begin{aligned} &\mathbb{r}(z) = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} + \cdots + \gamma^{T-(t+1)} r_T \\ &\text{이때 } z = (s_t, r_t) \xrightarrow{a_t} (s_{t+1}, r_{t+1}) \xrightarrow{a_{t+1}} (s_{t+2}, r_{t+2}) \xrightarrow{a_{t+2}} \cdots\cdots \xrightarrow{a_{T-1}} (s_T, r_T) \end{aligned}\right\} \quad (9.8)$$

- 할인율을 적용한 벨만 기대 방정식
  - 앞으로 다룰 동적 프로그래밍, Sarsa, Q 러닝은 식 (9.9)를 푸는 알고리즘

$$v_\pi(s) = \sum_{a \in \mathcal{A}(s)} P(a \mid s)\left(r + \gamma v_\pi(s')\right), \ \forall s \in \mathcal{S} \quad (9.9)$$

9.2.4 가치 함수 계산을 위한 벨만 방정식

- 상태 가치 함수와 행동 가치 함수
  - 식 (9.9)는 상태 가치 함수state value function(상태 s의 함수)
  - 식 (9.10)은 행동 가치 함수action value function(상태 s와 행동 a의 함수)

$$q_\pi(s,a) = r + \gamma v_\pi(s') = r + \gamma \sum_{a' \in \mathcal{A}(s')} P(a' \mid s') q_\pi(s', a') \qquad (9.10)$$

9.3 동적 프로그래밍

- 알고리즘 과목에서 배우는 동적 프로그래밍
  - 문제를 가장 작은 단위까지 분해한 다음 가장 작은 문제부터 해결하기 시작하여 점점 큰 문제로 진행하는 상향식 문제해결 방법론
  - 예, 행렬 곱셈 순서 문제: n개 행렬을 곱할 때 어떤 순서로 곱해야 연산량이 가장 적은지 알아내는 문제

9.3.1 정책 반복 알고리즘

정책 반복 알고리즘

- 03행이 가장 많은 시간 소요
- 계산 시간이 과도하여 잘 사용되지 않음

[알고리즘 9-1] 정책 반복

입력: 상태 전이 확률 분포 $P(s',r|s,a), \forall s \in \mathcal{S}, \forall a \in \mathcal{A}, \forall r \in \mathcal{R}$ (식 (9.2))

출력: 최적 정책 $\hat{\pi}$와 최적 가치 함수 $\hat{v}$

01. 랜덤 정책으로 π를 초기화한다.
02. repeat
03. 가치 함수 v_π를 계산한다. // 식 (9.9)를 적용하여 평가
04. v_π로 π를 개선하여 π'라 한다. // 개선
05. $\pi = \pi'$
06. until (멈춤 조건)
07. $\hat{\pi} = \pi, \hat{v} = v_\pi$

9.3.2 가치 반복 알고리즘

가치 반복 알고리즘

- 식 (9.12)의 벨만 최적 방정식을 이용함

$$v(s) = \max_{a \in \mathcal{A}(s)} \left(r + \gamma v(s') \right), \forall s \in \mathcal{S} \qquad (9.12)$$

[알고리즘 9-2] 가치 반복

입력: 상태 전이 확률 분포 $P(s', r|s, a), \forall s \in \mathcal{S}, \forall a \in \mathcal{A}, \forall r \in \mathcal{R}$ (식 (9.2))

출력: 최적 가치 함수 $\hat{v}$

01. 모든 상태 $s \in \mathcal{S}$에 대해 $v(s) = 0$으로 초기화한다.
02. repeat
03. for $(s \in \mathcal{S})$ // 모든 상태에 대해
04. $v'(s) = \max_{a \in \mathcal{A}(s)} \left(r + \gamma v(s') \right)$ // 식 (9.12)의 벨만 최적 방정식 적용
05. $v = v'$
06. until (멈춤 조건)
07. $\hat{v} = v$

- 동적 프로그래밍은 부트스트랩 방식
  - 모든 상태가 부정확한 값으로 출발하여 이웃 상태와 정보를 주고받으며 점점 수렴해가는 방식
  - 예) FrozenLake에서는 목적지에 인접한 상태부터 정확해져서 점점 멀리 확산

9.3.2 가치 반복 알고리즘

- 동적 프로그래밍의 한계
  - 마르코프 결정 프로세스를 알아야만 적용 가능
  - 크기가 작은 문제에 국한하여 적용 가능(표를 사용하기 때문)

9.4 학습 기반 알고리즘

- 훈련 데이터를 활용한 '학습' 기반 알고리즘
  - 동적 프로그래밍은 상태 전이 확률을 완벽히 알아야만 적용 가능
  - 상태 전이 확률이 없는 경우, 환경을 시뮬레이션하여 데이터, 즉 에피소드를 수집
    - 예) 프로 기사가 둔 바둑 기보 또는 프로그램과 프로그램을 대결시키는 셀프 플레이로 데이터 수집

TIP https://www.gokgs.com에 접속하면 기보를 얻을 수 있다. 알파고는 이 기보를 사용해 학습하였다.

9.4.1 몬테카를로 방법

에피소드에서 데이터 수집

- 에피소드의 표현

$$e=[s_0, r_0]a_0[s_1, r_1]a_1[s_2, r_2]a_2\cdots[s_t, r_t]a_t\cdots[s_T, r_T] \qquad (9.13)$$

- 예) FrozenLake에서 수집한 에피소드

e_1=[0, 0]1[4, 0]2[5, 0]

e_2=[0, 0]2[1, 0]2[2, 0]2[3, 0]0[2, 0]1[6, 0]1[10, 0]1[14, 0]2[15, 1]

- 상태를 중심으로 에피소드를 잘라 데이터 수집

$$\begin{cases} Z(0)=\{[0,0]1[4,0]2[5,0],[0,0]2[1,0]2[2,0]2[3,0]0[2,0]1[6,0]1[10,0]1[14,0]2[15,1]\} \\ Z(1)=\{[1,0]2[2,0]2[3,0]0[2,0]1[6,0]1[10,0]1[14,0]2[15,1]\} \\ Z(2)=\{[2,0]2[3,0]0[2,0]1[6,0]1[10,0]1[14,0]2[15,1],[2,0]1[6,0]1[10,0]1[14,0]2[15,1]\} \\ Z(3)=\{[3,0]0[2,0]1[6,0]1[10,0]1[14,0]2[15,1]\} \\ Z(4)=\{[4,0]2[5,0]\} \\ \vdots \end{cases}$$

9.4.1 몬테카를로 방법

- 데이터를 가지고 가치 함수를 계산하는 몬테카를로 방법

$$v_\pi(s) = \frac{1}{|Z(s)|} \sum_{z \in Z(s)} \mathbb{r}(z), \forall s \in \mathcal{S} \qquad (9.14)$$

$$q_\pi(s,a) = \frac{1}{|Z(s,a)|} \sum_{z \in Z(s,a)} \mathbb{r}(z), \forall s \in \mathcal{S}, \forall a \in \mathcal{A} \qquad (9.15)$$

  - 몬테카를로 방법은 데이터를 활용하기 때문에 학습 기반인데, 이웃 상태의 정보를 고려하지 않기 때문에 부트스트랩이 아님
  - 동적 프로그래밍은 데이터를 활용하지 않기 때문에 학습 기반이 아닌데, 이웃 상태를 고려하므로 부트스트랩 방식임
  - 학습 방법과 부트스트랩의 장점을 결합하여 더 좋은 알고리즘을 만들 수 있을까?

9.4.2 시간차 학습 알고리즘: Sarsa와 Q 러닝

- 시간차 학습time-difference(TD) learning의 핵심 아이디어
  - 식 (9.14)에서는 Z에 있는 모든 샘플이 종료 순간 T까지 도달해야 함. 종료 순간까지 가봐야 누적 보상액을 알 수 있기 때문
  - 종료 순간까지 가보지 않고 단지 이웃 상태로 전환하여 가치 함수를 개선할 수는 없나?
- 시간차 학습을 위한 공식 유도
  - $e = [s_0, r_0]a_0 \cdots [s_t, r_t]a_t[s_{t+1}, r_{t+1}]a_{t+1} \cdots [s_T, r_T]$로부터 t 순간에 추출한 샘플 $z = [s_t, r_t]a_t[s_{t+1}, r_{t+1}]a_{t+1} \cdots [s_T, r_T]$를 처리한다고 가정하고 식 (9.14)를 다시 쓰면,

$$v(s_t) = \frac{1}{|Z(s_t)|} \sum_{z \in Z(s_t)} \mathbb{r}(z)$$

    - z가 k번째 추가된다면,

$$v^{new}(s_t) = \frac{v^{old}(s_t) * (k-1) + \mathbb{r}(z)}{k} = v^{old}(s_t) + \frac{1}{k}\left(\mathbb{r}(z) - v^{old}(s_t)\right)$$

    - 위 첨자를 제거하고 1/k를 ρ로 대치하면,

$$v(s_t) = v_\pi(s_t) + \rho\left(\mathbb{r}(z) - v(s_t)\right)$$

9.4.2 시간차 학습 알고리즘: Sarsa와 Q 러닝

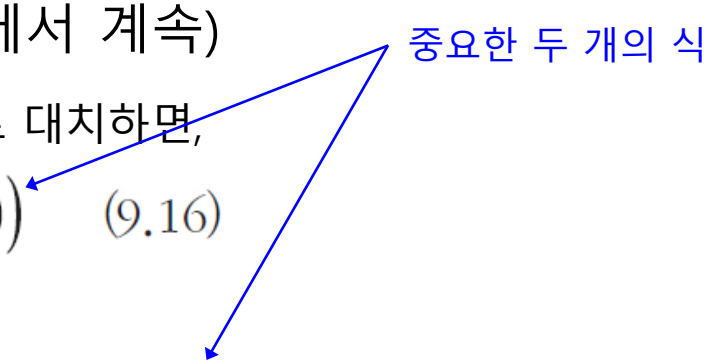
- 시간차 학습을 위한 공식 유도(...앞에서 계속)
  - $\mathbb{r}(z)$를 식 (9.9)에 있는 항 $r_{t+1} + \gamma v_\pi(s_{t+1})$로 대치하면,

$$v(s_t) = v(s_t) + \rho\left(\left(r_{t+1} + \gamma v(s_{t+1})\right) - v(s_t)\right) \qquad (9.16)$$

  - 행동 가치 함수로 쓰면,

$$q(s_t, a_t) = q(s_t, a_t) + \rho\left(\left(r_{t+1} + \gamma q(s_{t+1}, a_{t+1})\right) - q(s_t, a_t)\right) \qquad (9.17)$$

  - 시간차 학습
    - 식 (9.16)과 식 (9.17)은 에피소드를 사용하므로 학습 기반이고 s_t와 s_{t+1}이 같이 등장하므로 부트스트랩 방식임
    - 학습 기반의 장점과 부트스트랩의 장점을 결합한 혁신적인 알고리즘
    - Sarsa와 Q 러닝의 두 가지 알고리즘이 있음



9.4.2 시간차 학습 알고리즘: Sarsa와 Q 러닝

- Sarsa
  - 04, 06, 07행이 s-a-r-s′-a′ 고리를 만들고, 08행은 sarsa 고리를 식 (9.17)에 대입하여 가치 함수를 개선함
  - 켜진 정책$_{\text{on-policy}}$ 방식(행동 a′를 결정할 마땅한 방법이 없어 07행에서 현재 가치 함수 q에 의존)

[알고리즘 9-3] Sarsa

입력: 매개변수 ρ, γ

출력: 최적 행동 가치 함수 $\hat{q}$

01. 모든 상태 $s \in \mathcal{S}$와 모든 행동 $a \in \mathcal{A}$에 대해 $q(s,a)=0$
02. repeat
03. s=초기 상태 // 새로운 에피소드 시작
04. $a = \underset{a}{\text{argmax}}\, q(s,a)$에 따라 상태 s에서 행동 a 결정. 필요하면 ε-탐욕 적용
05. repeat
06. a를 실행하여 보상 r과 다음 상태 s'를 얻는다.
07. $a' = \underset{a}{\text{argmax}}\, q(s',a)$에 따라 상태 s'에서 행동 a'를 결정. 필요하면 ε-탐욕 적용
08. $q(s,a) = q(s,a) + \rho(r + \gamma q(s',a') - q(s,a))$ // 식 (9.17) 적용
09. $s=s'$, $a=a'$
10. until(s가 종료 상태)
11. until(멈춤 조건)
12. $\hat{q} = q$

9.4.2 시간차 학습 알고리즘: Sarsa와 Q 러닝

Q 러닝

- 가치 함수 q에 의존하는 대신 max 연산자를 사용함

$$q(s_t,a_t)=q(s_t,a_t)+\rho\left(\left(r_{t+1}+\gamma\max_a q(s_{t+1},a)\right)-q(s_t,a_t)\right) \quad (9.18)$$

- 꺼진 정책$_{\text{on-policy}}$ 방식

[알고리즘 9-4] Q 러닝

입력: 매개변수 ρ,γ

출력: 최적 행동 가치 함수 $\hat{q}$

```
모든 상태 s∈S와 모든 행동 a∈A에 대해 q(s,a)=0
repeat
    s=초기 상태  // 새로운 에피소드 시작
    repeat
        a = argmax_a q(s,a)에 따라 상태 s에서 행동 a 생성. 필요하면 ε-탐욕 적용
        a를 실행하여 보상 r과 다음 상태 s'를 얻는다.
        q(s,a)=q(s,a)+ρ(r+γ max_a' q(s',a')-q(s,a))        // 식 (9.18) 적용
        s=s'
    until(s가 종료 상태)
until(멈춤 조건)
q̂ = q
```

9.4.3 Q 러닝 프로그래밍

- FronzenLake 문제에 Q 러닝을 적용한 [프로그램 9-3]

프로그램 9-3 Q 러닝 알고리즘을 이용한 FrozenLake 문제해결

```
01  import gym
02  import numpy as np
03
04  env=gym.make('FrozenLake-v0',is_slippery=False)          # 환경 생성
05  Q=np.zeros([env.observation_space.n,env.action_space.n])  # Q 배열 초기화
06
07  rho=0.8                                                  # 학습률
08  lamda=0.99                                               # 할인율
09
10  n_episode=2000
11  length_episode=100
12
13  # 최적 행동 가치 함수 찾기
14  for i in range(n_episode):
15      s=env.reset()                                        # 새로운 에피소드 시작
16      for j in range(length_episode):
17          argmaxs=np.argwhere(Q[s,:]==np.amax(Q[s,:])).flatten().tolist()
18          a=np.random.choice(argmaxs)
19          s1,r,done,_=env.step(a)
20          Q[s,a]=Q[s,a]+rho*(r+lamda*np.max(Q[s1,:])-Q[s,a])    # 식 (9.18)
21          s=s1
22          if done:
23              break
24
25  np.set_printoptions(precision=2)
26  print(Q)
```

gym을 에피소드 발생기로 사용

상태의 개수*행동의 개수 크기의 배열 생성

상태 s에 대해 최대값을 갖는 행동을 모두 찾아 리스트에 저장

식 (9.18) 적용

9.4.3 Q 러닝 프로그래밍

```
[[0.    0.95  0.    0.  ]
 [0.    0.    0.    0.  ]
 [0.    0.    0.    0.  ]
 [0.    0.    0.    0.  ]
 [0.    0.96  0.    0.  ]
 [0.    0.    0.    0.  ]
 [0.    0.    0.    0.  ]
 [0.    0.    0.    0.  ]
 [0.    0.    0.97  0.  ]
 [0.    0.98  0.    0.  ]
 [0.    0.63  0.    0.  ]
 [0.    0.    0.    0.  ]
 [0.    0.    0.    0.  ]
 [0.    0.    0.99  0.  ]
 [0.    0.    1.    0.  ]
 [0.    0.    0.    0.  ]]
```

[프로그램 9-3]의 실행 결과를 보기 좋게 새로 그리면

[프로그램 9-4]의 실행 결과를 보기 좋게 새로 그리면

.0 .0 S .0 .95	.0 .0 F .0 .0	.0 .0 F .0 .0	.0 .0 F .0 .0
.0 .0 F .0 .96	.0 .0 H .0 .0	.0 .0 F .0 .0	.0 .0 H .0 .0
.0 .0 F .97 .0	.0 .0 F .0 .98	.0 .0 F .0 .0	.0 .0 H .0 .0
.0 .0 H .0 .0	.0 .0 F .99 .0	.0 .0 F 1. .0	.0 .0 G .0 .0

(a) [프로그램 9-3]으로 찾은 최적 행동 가치 함수

.94 .94 S .95 .93	.95 .94 F .96 .0	.96 .95 F .95 .97	.95 .96 F .94 .0
.94 .0 F .0 .86	.0 .0 H .0 .0	.96 .0 F .0 .98	.0 .0 H .0 .0
.0 .81 F .97 .0	.0 .87 F .98 .77	.97 .97 F .0 .99	.0 .0 H .0 .0
.0 .0 H .0 .0	.97 .0 F .98 .0	.98 .96 F 1. .99	.0 .0 G .0 .0

(b) [프로그램 9-4]로 찾은 최적 행동 가치 함수(ε-탐욕 적용)

그림 9-9 Q 러닝으로 FrozenLake-v0 문제해결

9.4.3 Q 러닝 프로그래밍

ε-탐욕 적용

- [프로그램 9-3]은 탐사형으로 동작함(한 개 경로만 찾음)
- ε-탐욕을 적용하여 탐사형과 탐험형의 균형을 추구하는 [프로그램 9-4]

프로그램 9-4 Q 러닝 알고리즘을 이용한 FrozenLake 문제해결(ε-탐욕 적용)

```
01  import gym
02  import numpy as np
03
04  env=gym.make('FrozenLake-v0',is_slippery=False)          # 환경 생성
05  Q=np.zeros([env.observation_space.n,env.action_space.n]) # Q 배열 초기화
06
07  rho=0.90                                                 # 학습률
08  lamda=0.99                                               # 할인율
09  eps=1.0                                                  # 엡시론
10  eps_decay=0.999                                          # 삭감 비율
11
12  n_episode=3000
13  length_episode=100
14
```

9.4.3 Q 러닝 프로그래밍

```
15  # 최적 행동 가치 함수 찾기(탐사와 탐험의 균형 추구)
16  for i in range(n_episode):
17      s=env.reset()                          # 새로운 에피소드 시작
18      for j in range(length_episode):        # 시간이 지남에 따라 탐험형 정도를 줄임
19          r=np.random.random()
20          eps=max(0.01,eps*eps_decay)        # 엡시론을 조금씩 줄여나감
21          if(r<eps):                         # 랜덤 정책
22              a=np.random.randint(0,env.action_space.n)
23          else:
24              argmaxs=np.argwhere(Q[s,:]==np.amax(Q[s,:])).flatten().tolist()
25              a=np.random.choice(argmaxs)
26          s1,r,done,_=env.step(a)
27          Q[s,a]=Q[s,a]+rho*(r+lamda*np.max(Q[s1,:])-Q[s,a])
28          s=s1
29          if done:
30              break
31
32  np.set_printoptions(precision=2)
33  print(Q)
```

9.4.3 Q 러닝 프로그래밍

```
[[0.94  0.93  0.95  0.94]
 [0.94  0.    0.96  0.95]
 [0.95  0.97  0.95  0.96]
 [0.96  0.    0.94  0.95]
 [0.    0.86  0.    0.94]
 [0.    0.    0.    0.  ]
 [0.    0.98  0.    0.96]
 [0.    0.    0.    0.  ]
 [0.81  0.    0.97  0.  ]
 [0.87  0.77  0.98  0.  ]
 [0.97  0.99  0.    0.97]
 [0.    0.    0.    0.  ]
 [0.    0.    0.    0.  ]
 [0.    0.    0.98  0.97]
 [0.96  0.99  1.    0.98]
 [0.    0.    0.    0.  ]]
```

[그림 9-9(b)] 참조

9.4.3 Q 러닝 프로그래밍

- 참조표 방식의 한계
  - 상태 가치 함수를 표현하는 v 또는 행동 가치 함수를 표현하는 q는 각각 1차원과 2차원 배열. 이것을 사용하는 알고리즘을 참조표$_{\text{lookup table}}$ 방식이라 부름
  - 상태의 개수가 방대한 경우 참조표 방식은 비현실적임
    - 예) 바둑 $3^{361} \fallingdotseq 1.74*10^{172}$개의 상태

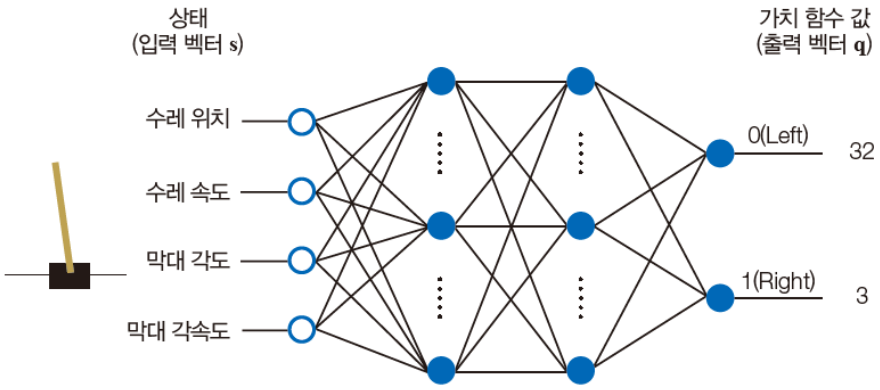
9.5 DQN

- 참조표 없이 가치 함수를 계산하는 방법은?
  - 신경망이 대안(신경망은 입력과 출력을 연결해주는 일종의 함수이므로 가치 함수를 표현할 수 있음)
  - 가장 중요한 문제는 학습에 사용할 훈련 집합을 수집하는 방법을 찾는 것(강화 학습에서 '특징 벡터-레이블'의 샘플 수집이 가능한가? 레이블은 누가 어떻게 달아주나?)
  - 므니 등은 에피소드에서 샘플을 추출하고 레이블을 자동으로 달아주는 획기적인 알고리즘 개발. 비디오 게임에 적용하여 사람 수준의 프로그램을 만듦

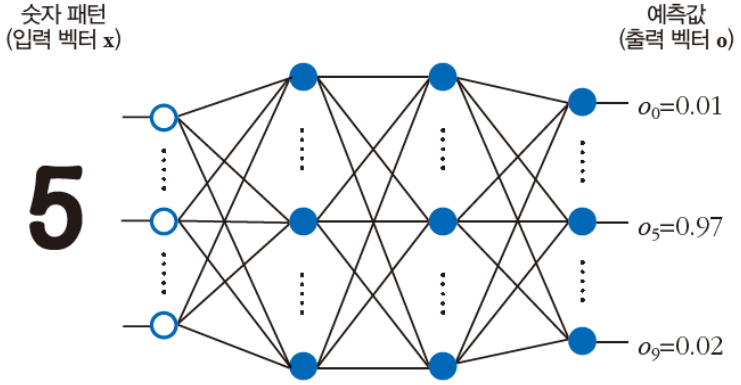
9.5.1 DQN의 원리

- DQN
  - 딥러닝과 Q 러닝을 결합하여 뛰어난 성능을 달성하는 신경망 모델
  - [그림 5-8]의 깊은 다층 퍼셉트론 또는 [그림 6-10]의 컨볼루션 신경망에 Q 러닝이 사용하는 식 (9.18)을 결합

- 가치 함수를 추정해주는 신경망([그림 9-10(a)]
  - 입력은 상태 s이고 출력은 가치 함수의 값 q(s,a)
  - 최적의 신경망이 있다면 모든 게임을 이길 수 있음(상태 s에서 q(s,a)를 계산하고 가장 큰 값을 가진 a를 선택하면 됨)
  - 어떻게 이런 신경망을 만들까?



(a) 강화 학습에서 신경망의 입출력(CartPole 예)



(b) 지도 학습에서 신경망의 입출력

그림 9-10 강화 학습과 지도 학습에서 신경망이 하는 일

9.5.1 DQN의 원리

훈련 집합 수집 방법

- Q 러닝에서 생성되는 에피소드로부터 샘플을 수집
  - 에피소드에서 t 순간은 $[s_t, r_t]a_t[s_{t+1}, r_{t+1}]$
  - 식 (9.18)을 간결하게 다시 쓰면,

$$q(s,a) = q(s,a) + \rho\left(\left(r' + \gamma \max_{a'} q(s',a')\right) - q(s,a)\right) \qquad (9.19)$$

- [그림 9-11]은 신경망 입장에서 식 (9.19)를 해석
  - 현재 q(s,a)를 신경망의 추정치, 식 (9.19)의 오른쪽 변을 레이블로 간주

현재 추정치(신경망의 출력) 개선된 추정치(레이블)

$$q(s,a) \qquad q(s,a) + \rho\left(\left(r' + \gamma \max_{a'} q(s',a')\right) - q(s,a)\right)$$

reward[i] np.amax(target1[i])

그림 9-11 개선된 추정치를 레이블로 간주하여 훈련 샘플 수집(파란색은 [프로그램 9-5] 43행의 항)

9.5.1 DQN의 원리

- 리플레이 메모리
  - 샘플 간의 상관 관계가 높은 문제
    - t가 1에서 2로, 2에서 3으로, 3에서 4로 … 바뀔 때 샘플을 수집하므로 샘플이 유사한 문제 발생
  - 리플레이 메모리로 해결
    - 추출한 샘플로 바로 학습하는 대신 리플레이 메모리에 저장해 둠
    - 실제 학습은 리플레이 메모리에서 램덤하게 샘플을 추출하여 미니배치를 구성하여 수행
    - 리플레이 메모리가 꽉 차면 가장 오래된 것을 삭제하고 추가 → 큐로 구현

9.5.2 CartPole 문제

CartPole 문제

- 수레를 좌우로 이동하여 수직 막대를 오래 유지하는 게임

CartPole의 상태: $s=(x,\bar{x},\theta,\bar{\theta})$

$$=(\text{수레 위치},\text{수레 속도},\text{막대 각도},\text{막대 각속도}) \quad (9.20)$$

  - CartPole-v0는 200 단위시간을 넘기면 이김. 단위시간을 넘길 때마다 보상 1을 받음
  - 막대가 12도를 넘거나 위치가 2.4를 넘으면 실패로 간주하고 에피소드가 끝남
  - reset하면 [-0.05,0.05] 사이에서 초기 위치를 잡고 시작

- [그림 9-10(a)]는 CartPole 문제를 위한 신경망 구조

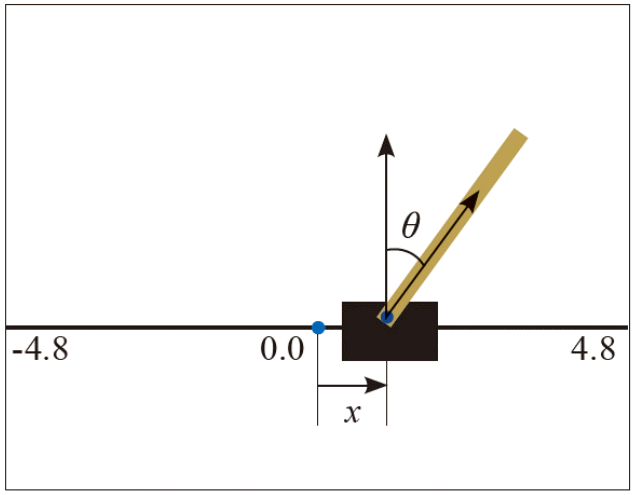


그림 9-12 CartPole 문제에서 상태 표현

9.5.3 DQN 프로그래밍

- DQN을 학습하는 [프로그램 9-5]

프로그램 9-5 DQN을 이용한 CartPole 문제해결(ε-탐욕 적용)

```
01  import numpy as np
02  import random
03  import gym
04  import tensorflow as tf
05  from tensorflow.keras.models import Sequential
06  from tensorflow.keras.layers import Dense
07  from tensorflow.keras.optimizers import Adam
08  from collections import deque
09
10  # 하이퍼 매개변수 설정
11  rho=0.9        # 학습률
12  lamda=0.99     # 할인율
13  eps=0.9
14  eps_decay=0.999
15  batch_siz=64
16  n_episode=100
17
```

9.5.3 DQN 프로그래밍

```
18  # 신경망을 설계해주는 함수
19  def deep_network():
20      mlp=Sequential()
21      mlp.add(Dense(32,input_dim=env.observation_space.shape[0],activation='relu'))
22      mlp.add(Dense(32,activation='relu'))
23      mlp.add(Dense(env.action_space.n,activation='linear'))
24      mlp.compile(loss='mse',optimizer='Adam')
25      return mlp
26
27  # DQN 학습
28  def model_learning():
29      mini_batch=np.asarray(random.sample(D,batch_siz))
30      state=np.asarray([mini_batch[i,0] for i in range(batch_siz)])
31      action=mini_batch[:,1]
32      reward=mini_batch[:,2]
33      state1=np.asarray([mini_batch[i,3] for i in range(batch_siz)])
34      done=mini_batch[:,4]
35
36      target=model.predict(state)
37      target1=model.predict(state1)
38
39      for i in range(batch_siz):
40          if done[i]:
41              target[i][action[i]]=reward[i]
42          else:
43              target[i][action[i]]+=rho*((reward[i]+lamda*np.amax(target1[i]))
                -target[i][action[i]]) # Q 러닝(식 (9.19))
44      model.fit(state,target,batch_size=batch_siz,epochs=1,verbose=0)
45
```

입력 벡터(식 (9.20)를 보면 4개의 요소)

출력 벡터

누적 보상을 출력하기 위해 linear 사용

[그림 9-10(a)]에 해당하는 신경망 설계

DQN의 핵심 아이디어를 코딩

29~34행은 리플레이 메모리 D에서 미니배치를 샘플링. 미니배치에서 현재 상태 state, 행동 action, 보상 reward, 다음 상태 state1, 에피소드 끝 여부 done을 추출

현재 상태 state에 대한 신경망 예측 결과를 target에 저장

다음 상태 state1에 대한 신경망 예측 결과를 target1에 저장

39~43행은 에피소드 끝이면 reward를 target에 저장 그렇지 않으면 식 (9.19)를 target에 저장

state를 입력, target을 출력으로 설정하고 학습 수행

9.5.3 DQN 프로그래밍

46행부터 메인

```
46  env=gym.make("CartPole-v0")
47
48  model=deep_network()    # 신경망 생성
49  D=deque(maxlen=2000)    # 리플레이 메모리 초기화
50  scores=[]
51  max_steps=env.spec.max_episode_steps
52
53  # 신경망 학습
54  for i in range(n_episode):
55      s=env.reset()
56      long_reward=0
57
58      while True:
59          r=np.random.random()
60          eps=max(0.01,eps*eps_decay)
61          if(r<eps):
62              a=np.random.randint(0,env.action_space.n)    # 랜덤 정책
63          else:
64              q=model.predict(np.reshape(s,[1,4]))         # 신경망이 예측한 행동
65              a=np.argmax(q[0])
66          s1,r,done,_=env.step(a)
67          if done and long_reward<max_steps-1:             # 실패
68              r=-100
69
70          D.append((s,a,r,s1,done))
71
72          if len(D)>batch_siz*3:
73              model_learning()
74
```

55~56행은 새로운 에피소드 시작

59~82행은 에피소드 하나를 처리

59~65행은 ε-탐욕 적용하여 행동 a를 정함

66행은 행동 a를 취하여 다음 상태 s1, 보상 r, 에피소드 끝 여부 done을 결정

67~68행은 에피소드가 중간에 끝난 경우 처리

70행은 샘플을 리플레이 메모리에 저장

72~73행은 모델을 학습(리플레이 메모리가 아직 충분치 않은 경우에는 생략)

9.5.3 DQN 프로그래밍

```
75          s=s1
76          long_reward+=r
77
78          if done:
79              long_reward=long_reward if long_reward==max_steps else long_reward+100
80              print(i,"번째 에피소드의 점수:",long_reward)
81              scores.append(long_reward)
82              break
83
84      if i>10 and np.mean(scores[-5:])>(0.95*max_steps):
85          break
86
87  # 신경망 저장
88  model.save("./cartpole_by_DQN.h5")
89  env.close()
90
91  import matplotlib.pyplot as plt
92
93  plt.plot(range(1,len(scores)+1),scores)
94  plt.title('DQN scores for CartPole-v0')
95  plt.ylabel('Score')
96  plt.xlabel('Episode')
97  plt.grid()
98  plt.show()
```

75~76행은 다음 반복을 준비

78~82행은 에피소드가 끝난 경우를 처리

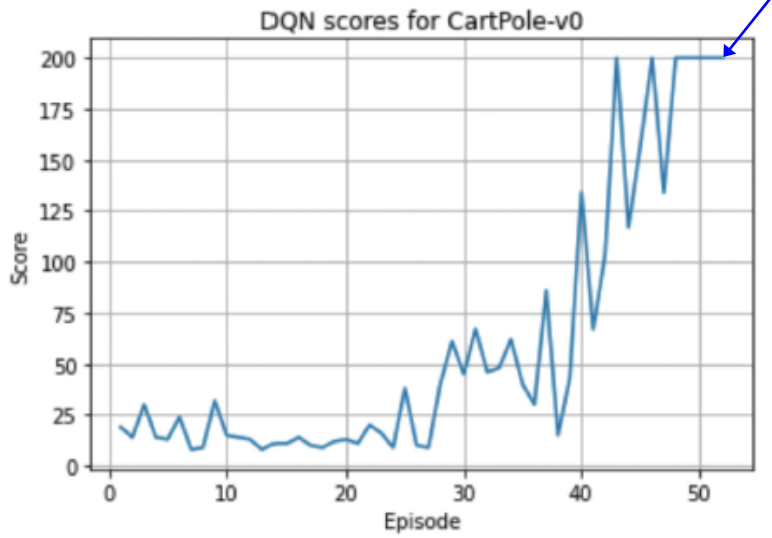
84~85행은 멈춤 조건에 도달하지 못했어도 수렴했다 판단되면 멈춤(최근 5개 에피소드가 최대 보상액의 95%를 넘으면)

88~89행은 나중에 사용할 목적으로 학습된 모델을 저장

9.5.3 DQN 프로그래밍

```
0 번째 에피소드의 점수: 14.0
1 번째 에피소드의 점수: 19.0
2 번째 에피소드의 점수: 32.0
3 번째 에피소드의 점수: 16.0
…
48 번째 에피소드의 점수: 152.0
49 번째 에피소드의 점수: 176.0
50 번째 에피소드의 점수: 200.0
51 번째 에피소드의 점수: 200.0
52 번째 에피소드의 점수: 200.0
53 번째 에피소드의 점수: 200.0
```

84행의 수렴 조건을 만족하여 끝냄



9.5.3 DQN 프로그래밍

- 학습된 DQN으로 게임 플레이

프로그램 9-6 학습된 신경망으로 CartPole 플레이

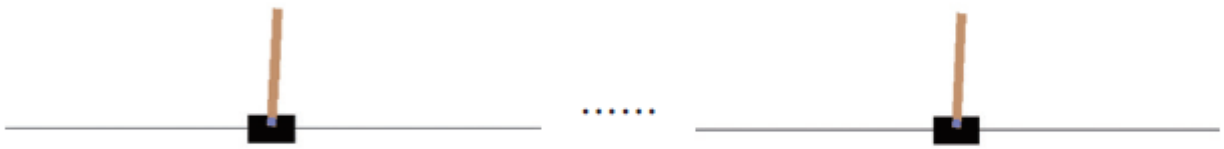
```
01  from tensorflow.keras.models import Sequential
02  from tensorflow.keras.models import load_model
03  import numpy as np
04  import gym
05  import time
06
07  # 신경망 불러옴
08  model=load_model('./cartpole_by_DQN.h5')
09
10  env=gym.make("CartPole-v0")
11  long_reward=0
12
13  # CartPole 플레이
14  s=env.reset()
15  while True:
16      q=model.predict(np.reshape(s,[1,4]))  # 신경망이 예측한 행동
17      a=np.argmax(q[0])
18      s1,r,done,_=env.step(a)
19      s=s1
20      long_reward+=r
21
22      env.render()
23      time.sleep(0.02)
24
25      if done:
26          print("에피소드의 점수:",long_reward)
27          break
28
29  env.close()
```

모델을 불러옴

15~27행은 신경망으로 행동을 예측하고 step 함수로 행동을 실행하는 일을 반복함

시간 지연을 두어 애니메이션 속도 조절

9.5.3 DQN 프로그래밍



에피소드의 점수: 200.0

9.6 강화 학습과 강한 인공지능

- 강화 학습은 강한 인공지능에 한 발 가까움
  - 비지도 학습(레이블을 자동으로 붙임)
  - 자율 플레이로 학습
  - 같은 신경망 구조로 서로 다른 게임을 높은 수준으로 플레이(과업 간 일반화 능력 향상)

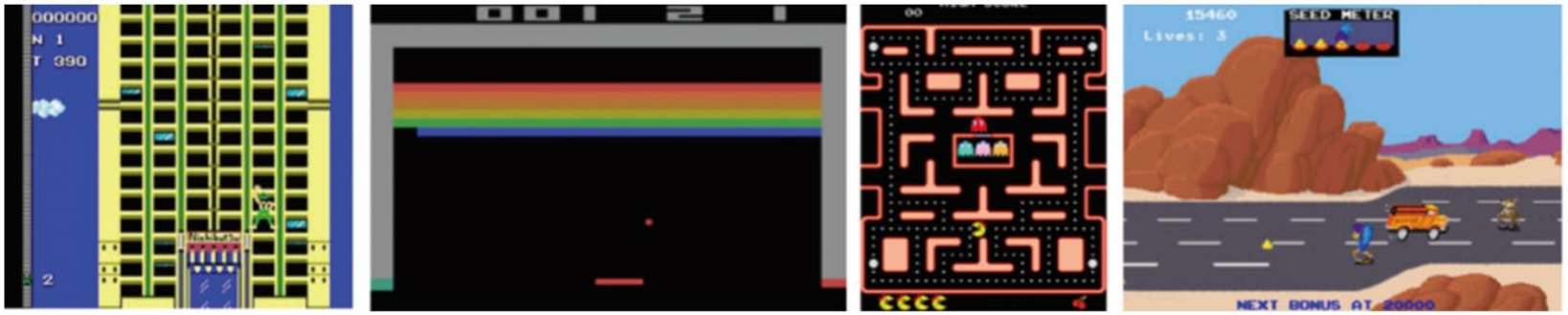


그림 9-13 여러 가지 아타리 비디오 게임

- 상대적인 평가에 불과하고 여전히 강한 인공지능은 멀리 있음