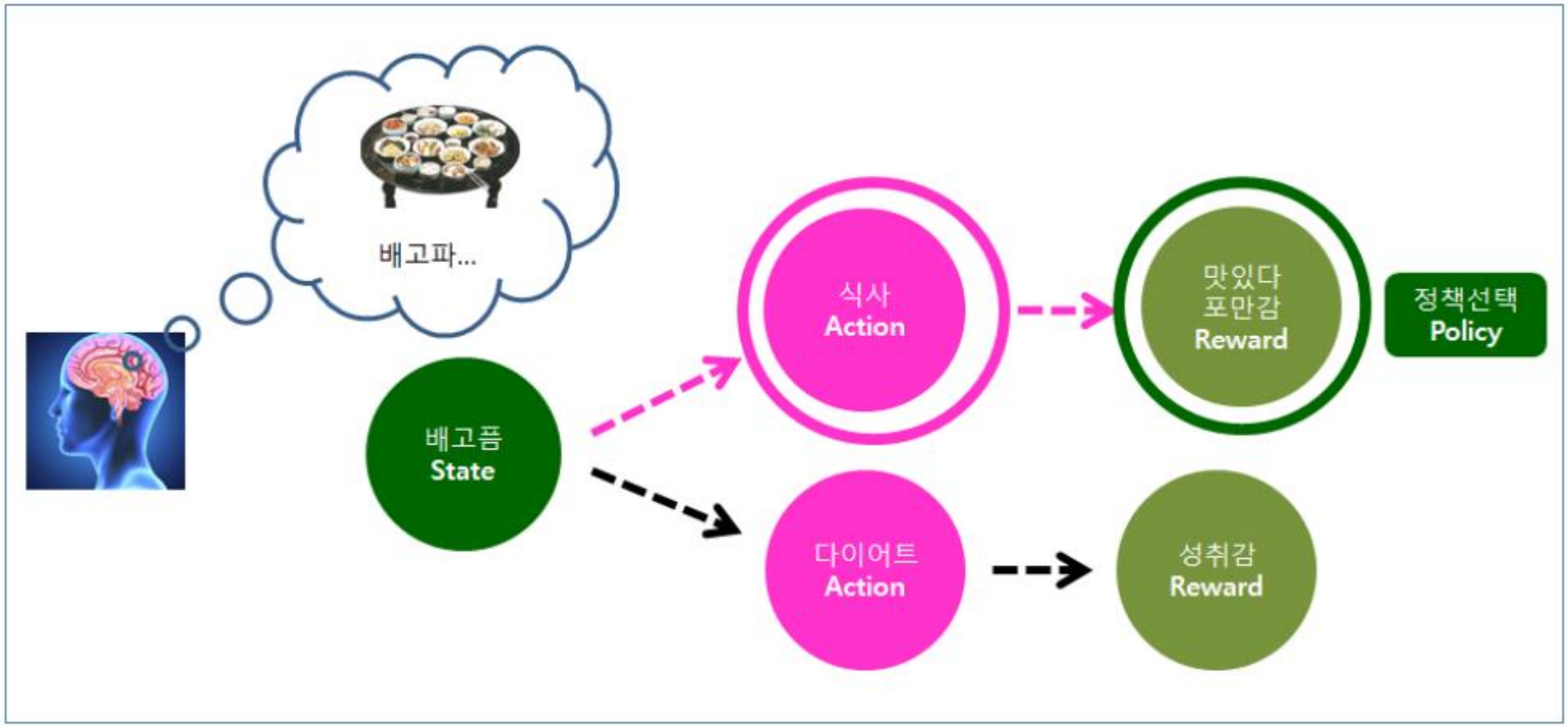
Deep 강화학습

Deep RL


배고파...
배고픔
State
식사
Action
다이어트
Action
맛있다
포만감
Reward
성취감
Reward
정책선택
Policy

우리는 제일 먼저 어떤 욕망을 느낀다. 식사하고 싶다? 운동하고 싶다? 영화보고 싶다? 의 형태인데 우리는 이것에 따라서 어떤 걸맞는 행동을 하게 되고 결과를 얻는다. 이때 얻는 결과는 보상이 될 수도 있고 벌점이 될 수도 있다. 이런 일련의 path들이 모이면 경험이 된다. 우리가 어떤 행동을 하는 이유는 심리적인 이유이고 그 중에서 가장 큰 부분을 차지하는 것은 욕망이

나 가치관이지만 이것을 AI에서 구체적으로 모델링하기는 어려우므로 이것을 '**상태**'라는 용어로 표현한다.

'배고픔'의 상태는 어떤 행동을 하도록 촉구하고 선택할 수 있는 몇 개의 행동리스트가 어떤 확률로 존재한다. 여기에서는 '식사'와 '다이어트'라는 2가지 **행동**을 취할 수 있다고 가정하자. (사실 두가지 말고 다른 선택할 행동은 없지만) 그동안의 경험을 볼 때에 두가지 행동이 가져올 결과는 상상이 가능하다. 이것은 뇌 속에 기억으로 저장이 되어 있는데 각각 '포만감'과 '성취감'으로 기억이 된다고 하자. 이것이 우리에게 가져다 주는 보상이 어느 정도인지는 사람마다 틀리지만 AI는 이것을 수치로 나타낼 수 있다. 이것을 '**보상**'이라고 한다. 그리고 우리는 '**보상**'이 가장 큰 path를 선택하게 되는데 이것을 '**정책**'이라고 한다.

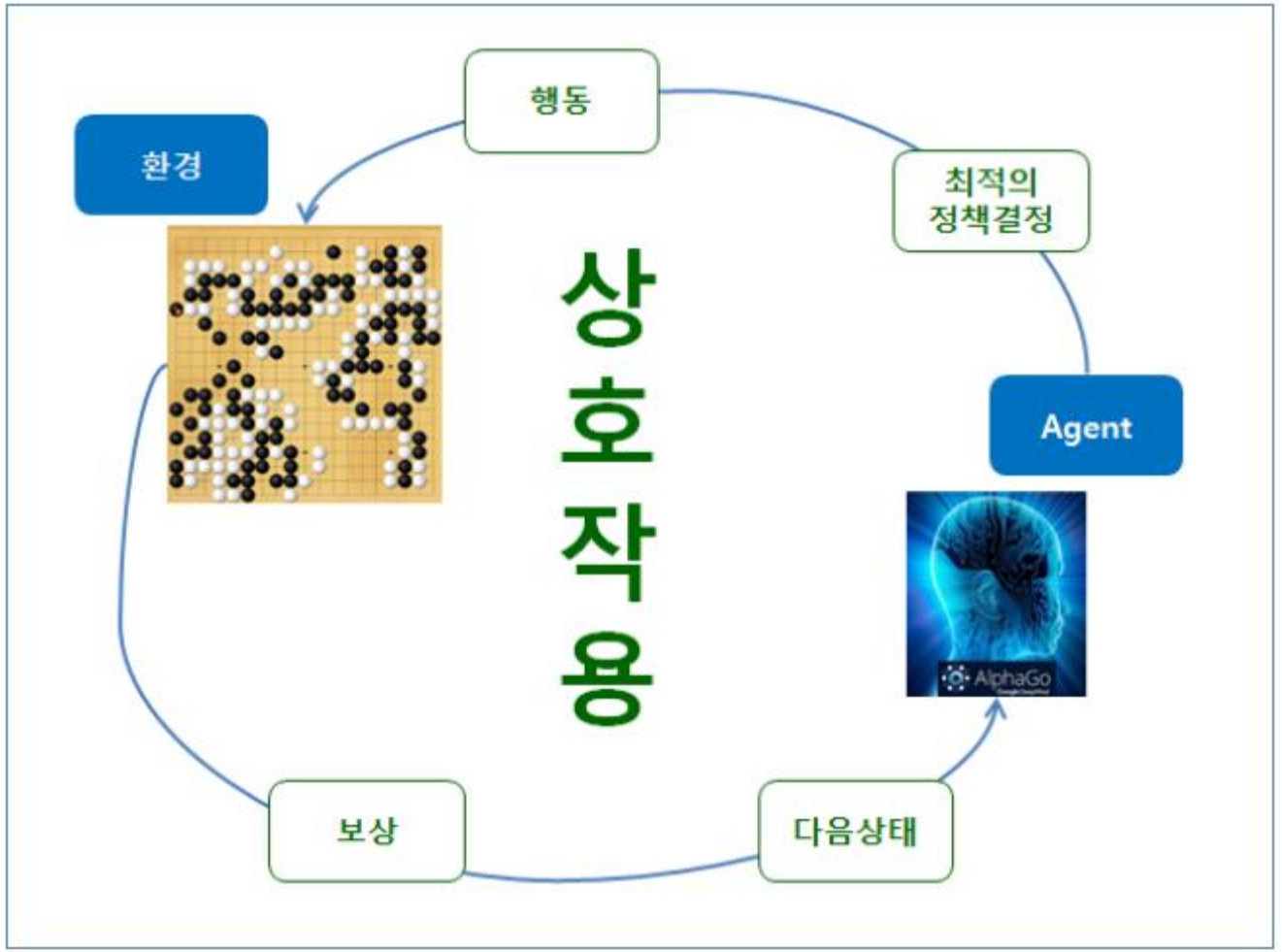
사람도 어릴 때에는 경험이 없어서 이것저것 시도를 해보면서 시행착오를 통하여 경험을 쌓다가 나이가 들면서 시행착오보다는 기존의 경험에 의존하여 행동을 선택하려는 보수적인 특징을 띄게 되는데 강화학습도 경험이 없는 초반에는 **탐험**(exploration)의 비중을 크게 두었다가 점차 시간이 흐르면서 그동안의 경험데이터를 기반으로 **정책**을 선택하려는 greedy한 경향이 커진다.

다음은 **보상**에 대하여 생각을 하여 보자. 보상이란 단어의 속에는 시간적인 가치라는 의미가 포함되어 있다는 사실을 알 수가 있다. 즉, 불확실한 미래의 확률적인 변동을 가지는 가치를 현재의 가치로 환산한다고 정의할 수 있다. 보상에 대한 시간적인 확률분포가 완전히 random이라면 불확실성은 시간이 지남에 따라 커지기 때문에 가치는 시간에 지남에 따라 일정비율만큼 감소한다고 가정한다. 그 일정비율이 γ라고 하고 현재의 보상이 R_{t+1}이라고 한다면 모든 시간에서의 보상의 총 현재환산가치는 $R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \gamma^4 R_{t+5} + \cdots$ 와 같은 무한급수가 될 것이다. γ는 discount factor(감가율)이라고 하고 0에서 1사이의 값을 가진다. 강화학습의 문제에서는 이 보상함수를 함수형태로 정의를 해야 하는데 학습의 초반에는 agent가 캄캄한 상태공간을 헤매이면서 시행착오를 겪기 때문에 환경이 주는 보상이 어떤 값인지 전혀 알지 못한다. 이러한 보상함수를 잘 정의하는 것이 필요하고 강화학습의 목적은 보상을 최대화하는 **정책**을 발견하는 것이다. (딥러닝의 목적은 에러함수를 최소화하는 것이었다) 현재의 보상 R은 시간이 1만큼 흐른 후에 받을 보상으로 정의하므로 현재시간이 t이고 시간이 k만큼 지나면 그때 받을 보상은 $\gamma^{k-1} R_{t+k}$가 된다.

그렇다면 **상태**에 대하여 생각을 하여보자. 우리는 **상태**를 행동을 촉발시키는 원인이 되는 욕구와 같은 것이라고 정의를 했지만 이것은 너무 추상적이므로 강화학습에서는 agent의 다음의 행동의 예측에 영향을 주는 모든 형태의 정보를 '**상태**'로 정의를 한다. agent가 게임에서의 캐릭터라고 한다면 현재의 좌표값과 움직이는 속도와 느끼는 중력 등이 될 수가 있고 백화점에서 TV를 사는 고객이라면 현재의 지갑의 상태나 고객정보가 될 수 있으며 agent의 고유정보로서 환경과 분리하여 생각을 해야 한다.

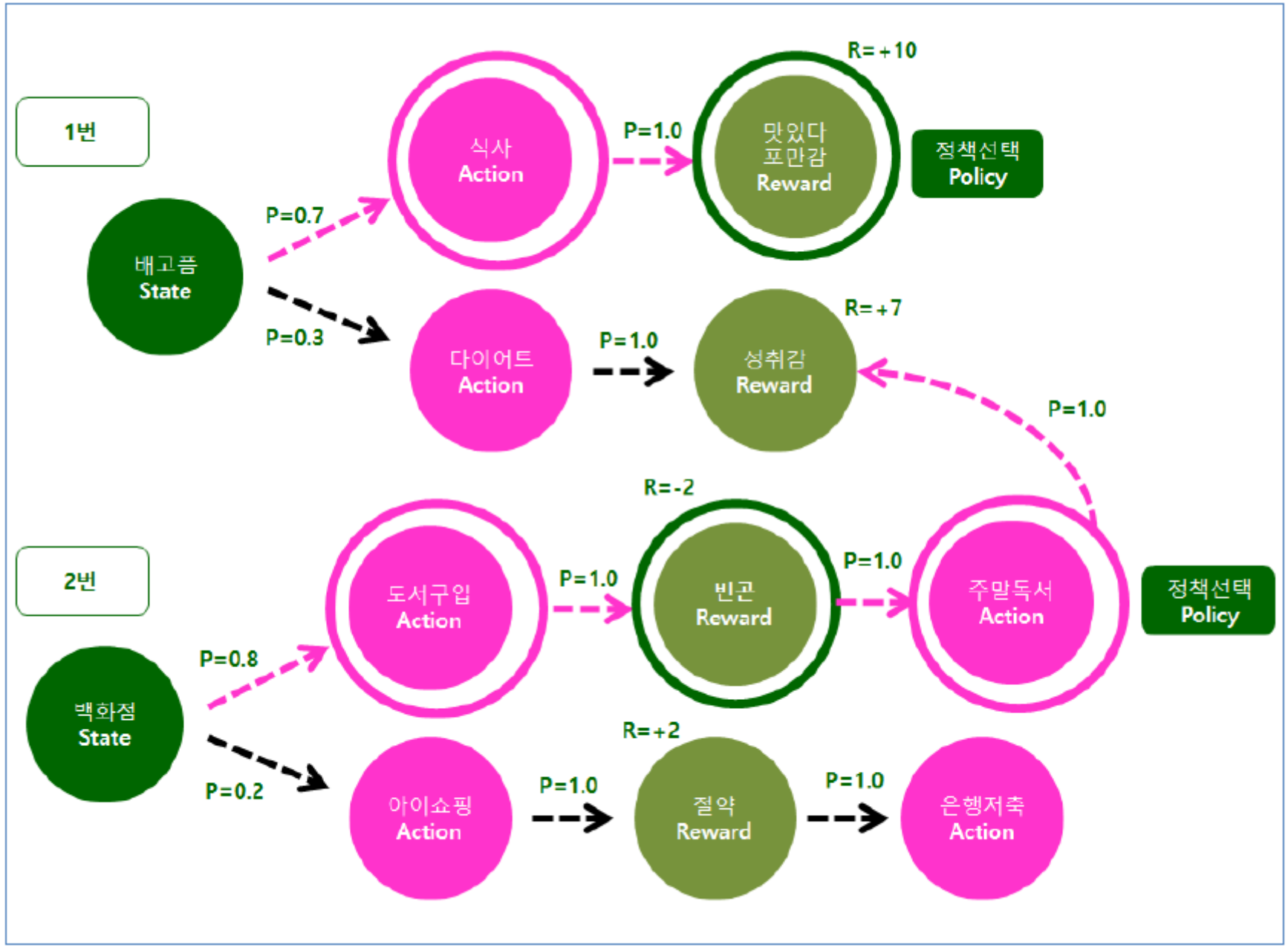
그렇다면 '**행동**'이란 무엇일까? 행동은 상태와 밀접한 관계에 있으며 그것은 행동에 따라서 다음 상태가 바뀌기 때문이다. 행동은 agent가 상태공간을 움직이는 가능한 모든 경우로 정의한다. 게임에서의 행동은 단순하여 보통 '좌','우'나 '상','하','좌','우'를 의미하고 이에 따라서 agent는 때로는 무한에 가까운 상태공간을 여행하면서 상태가 수시로 바뀐다. 백화점에서 물건을 사는 고객이라면 '산다','안산다'의 행동이 있을 수가 있으며 이에 따라서 상태는 '지갑이 가벼워짐','저축액수가 늘어남'등의 상태가 되고 그 다음의 행동은 '아낀 돈으로 외식을 하러 간다'나 또는 '은행에 통장을 개설한다'등이 될 가능성이 높아진다. 그 행동은 또 다른 상태를 낳고 그 상태는 또 다른 행동을 낳게 된다. 이런 연쇄반응은 무한히 계속 되는 경우도 있거나 게임처럼 캐릭터가 dead상태가 될 때까지 유한한 시간동안 반복되는 경우도 있다.

이때 우리는 유한한 시간단계동안 얻을 수 있는 모든 Reward들의 총합이 가장 큰 바로 다음 단계의 최적의 행동을 선택해야 하는 목적을 갖게 된다. 이것을 모든 상태에 대하여 가장 최적의 행동을 구하게 되면 그것을 '**정책**'이라고 한다. 이렇게 보면 강화학습은 진정한 의미의 인공지능에 가깝다고 볼 수 있다. 아무런 데이터가 없는 상태에서 스스로 환경에 적응하면서 최선의 정책을 발견하기 때문이다. agent와 환경은 이렇게 다음과 같이 상호작용을 하면서 agent의 지능은 향상되어 간다.


행동
환경
최적의
정책결정
상호작용
Agent
AlphaGo
보상
다음상태

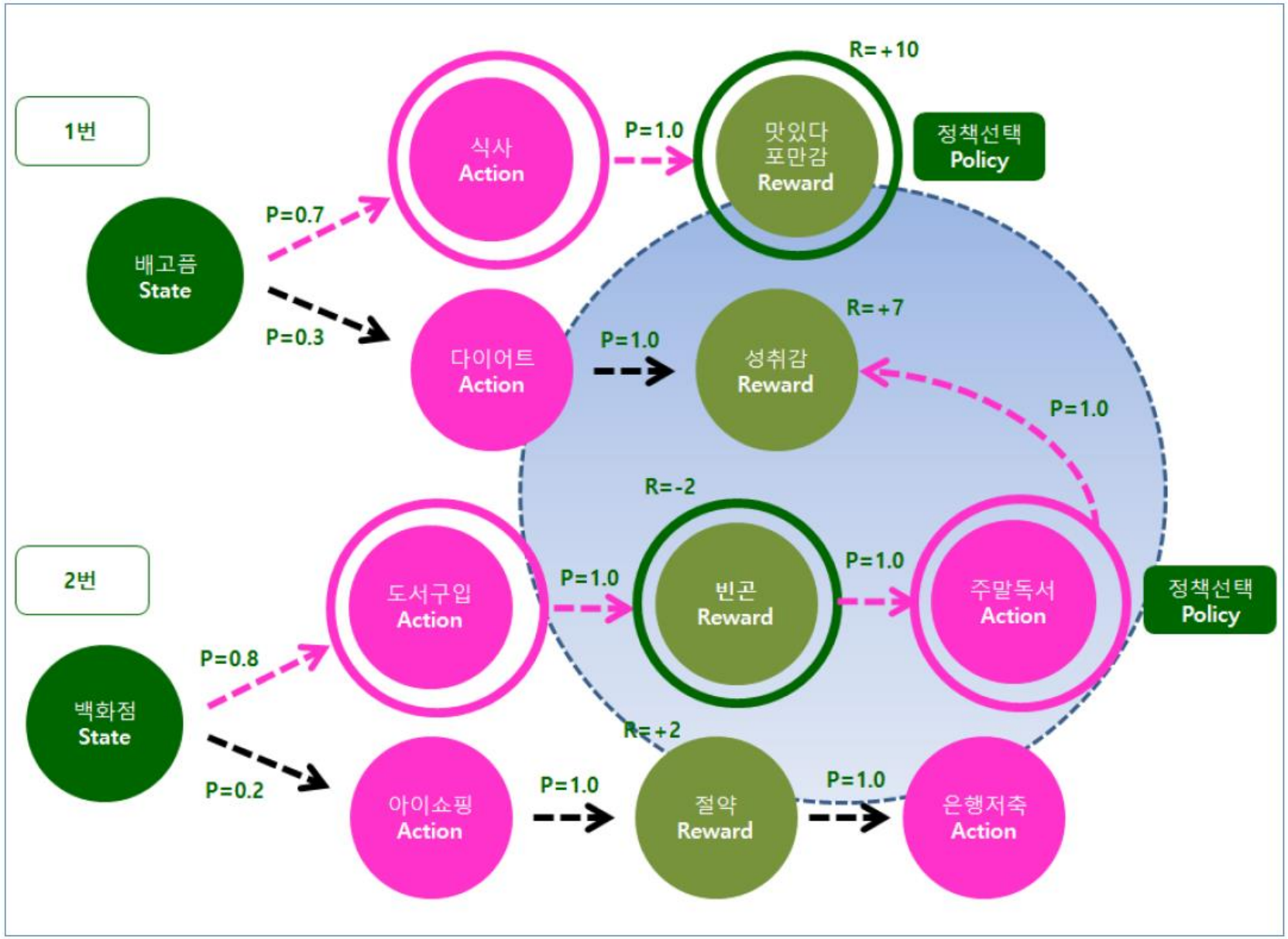
MDP(Markov Decision Process)

위의 그래프를 좀 더 확장시켜서 생각을 하여보자. 보통 행동은 사람마다 가치관이나 경험에 따라서 조금씩 바뀌기 마련이다. 기계도 학습을 하면서 가치관이나 경험이 바뀌면서 행동을 선택하는 기준도 매번 조금씩 틀려진다. 물론 기계는 가치관이라는 정신적인 개념은 없으므로 우리는 '**정책**'이란 개념을 사용한다. 정책은 앞에서 언급했다시피 모든 상태에서 가능한 행동규칙을 확률로 만든 테이블과 같은 것이다. 정책함수는 테이블이 아니라 확률분포가 될 수도 있다. 기계는 어떤 같은 상태에서 가능한 모든 행동리스트들 중에서 항상 똑같은 행동을 선택하지는 않고 어떤 확률로 분포를 가지게 되는데 다음과 같은 그림으로 이해를 하면 된다.


1번
배고픔
State
P=0.7
P=0.3
식사
Action
다이어트
Action
P=1.0
P=1.0
맛있다
포만감
Reward
성취감
Reward
R=+10
R=+7
정책선택
Policy
P=1.0
2번
백화점
State
P=0.8
P=0.2
도서구입
Action
아이쇼핑
Action
P=1.0
P=1.0
빈곤
Reward
절약
Reward
R=-2
R=+2
P=1.0
P=1.0
주말독서
Action
은행저축
Action
정책선택
Policy

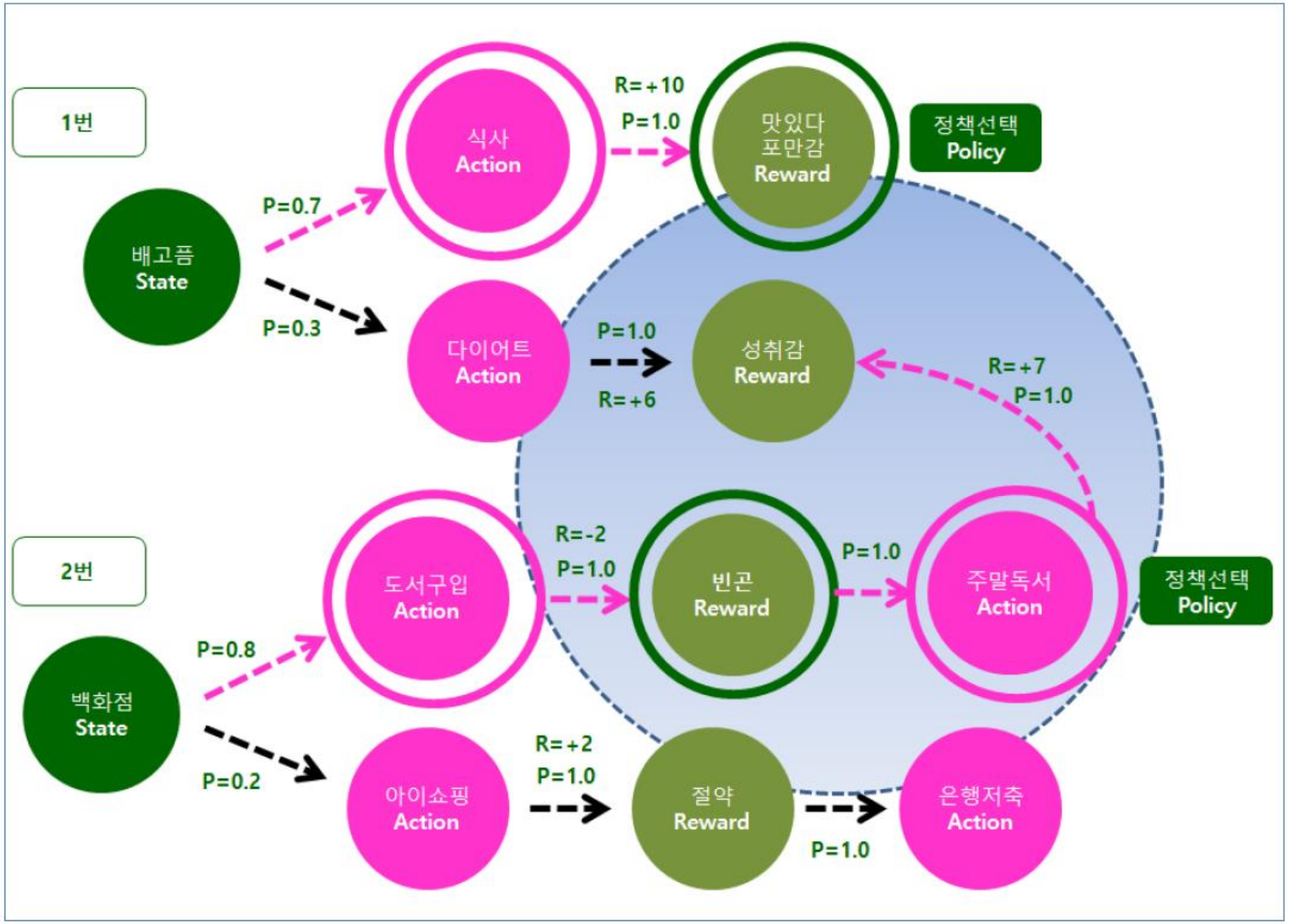
여기에서의 한 상태에서 다른 상태로 이전하는 확률은 **상태전이확률**이라고 하고 상태는 공유가 가능하다. 그리고 일부의 상태는 Reward를 갖는다. **마르코프 프로세스(MP)**가 조금 확장된 이런 그래프를 **마르코프 보상 프로세스(Markov Reward Process,MRP)**라고 한다. 이때 어떤 상태에서 어떤 확률로 어떤 행동을 취한다는 규칙들을 '**정책**'이라고 한다고 배웠다. 하지만 기계가 처음에는 환경이라는 상태공간을 돌아다니면서 경험 데이터를 수집을 하는데 이러한 기억이 거의 없는 상태에서 기억에만 의존하여 정책을 수립한다면 한정된 상태공간에서 한정된 행동만 취하게 되고 좋은 정책을 발견하지 못하게 된다.

초반에는 무작위로 돌아다니는 **탐험**에 많은 시간을 할애하도록 한다. 이것은 유전자 알고리즘에서 mutation에 해당하는 개념과 비슷하다. 어느 정도의 탐험에 의해서 기억 속에 경험데이터가 충분히 쌓인다면 기계는 학습을 할 때에 기억 속의 경험을 활용하며 **탐험**의 비중을 조금씩 줄여나가게 된다. ε이라는 비율로 탐험을 한다면 ε만큼은 무작위로 행동을 선택하고 1-ε만큼은 정책을 충실히 따른다. 그리고 위의 그래프처럼 상태와 행동은 공유가 가능하고 보상은 행동과 관계없이 상태에 종속이 된다.


1번
배고픔
State
P=0.7
P=0.3
식사
Action
다이어트
Action
P=1.0
맛있다
포만감
Reward
R=+10
정책선택
Policy
P=1.0
성취감
Reward
R=+7
P=1.0
2번
백화점
State
P=0.8
P=0.2
도서구입
Action
아이쇼핑
Action
P=1.0
빈곤
Reward
R=-2
P=1.0
주말독서
Action
정책선택
Policy
P=1.0
절약
Reward
R=+2
P=1.0
은행저축
Action

포커스가 된 부분에서 '성취감'이라는 상태에서의 Reward가 어떻게 결정되는지 보자. 두가지 행동의 경로에 의해서 보상이 결정되는데 하나는 '다이어트'이고 하나는 '주말독서'이다. 이것은 agent를 '성취감'이라는 상태로 이동시키고 보상점수 +7점을 준다. 하지만 서로 다른 행동에 의해서 얻어진 '성취감'이란 상태에서의 보상점수가 같을 수가 있을까?

다이어트의 성취감과 주말독서의 성취감의 점수를 다르게 하고 싶다면 이 모델로 설명하기가 힘들다는 사실을 알게 된다. 즉 Reward는 상태에만 종속되지 않는 상태와 행동의 함수이어야 한다. 보상이 달라지기 때문에 상태전이확률도 역시 상태만으로 종속되지 않고 상태와 행동에 의해서 종속된다.


1번
배고픔
State
P=0.7
P=0.3
식사
Action
다이어트
Action
R=+10
P=1.0
맛있다
포만감
Reward
정책선택
Policy
P=1.0
R=+6
성취감
Reward
R=+7
P=1.0
2번
백화점
State
P=0.8
P=0.2
도서구입
Action
아이쇼핑
Action
R=-2
P=1.0
R=+2
P=1.0
빈곤
Reward
절약
Reward
P=1.0
주말독서
Action
은행저축
Action
정책선택
Policy

이렇게 상태중심에서 상태와 행동중심으로 가치의 평가를 이루어지도록 그래프를 변경하였는데 이것을 MDP(Markov Decision Process)라고 한다. 즉, **마르코프 의사결정 프로세스**이다.

벨만 기대방정식과 Q함수

상태는 어떻게 표현할 것인가? 상태는 agent가 가지고 있는 정보 또는 고객이 가지고 있는 정보 또는 어떤 기계가 가지고 있는 정보인데 이 정보는 다음의 행동을 예측하여 정책을 수립하는데 도움이 되는 정보이어야 한다. 바둑에서는 돌의 정보이고 그것은 색깔 = {'검은색','하얀색'} 또는 좌표 = {(1,1),(1,2), … , (32,1),(32,2),(32,3),…}와 같은 집합으로 나타낼 수 있다. Invader와 같은 전자오락게임에서는 캐릭터의 정보로서 목숨의 개수 = {0,1,2,3} 또는 남은 적의 수 = {0,1,2,3,…}가 될 수도 있다. 상태를 s라고 한다면 시간에 대한 함수이므로 S = S(t)로 나타낸다. 백화점의 고객이라면 상태는 {고객의 직업,고객의 카드잔여한도,고객의 백화점 현재위치,고객의 결혼여부,…}이 될 수 있다. S(t)는 시간t에서의 agent의 상태정보이다.

행동은 어떻게 표현할까? 행동은 agent가 다음단계로 상태공간을 움직일 수 있는 가능한 모든 경우를 집합으로 나타낸 것이다. 예를 들어 벽돌깨기와 같은 게임에서는 행동의 경우는 {왼쪽, 오른쪽,발사,정지}로 표현이 가능하고 백화점의 고객의 경우에는 {구매,비구매}로 표현이 가능하다. 데이터를 하는 남녀커플의 경우는 {영화감상,레스토랑,노래방,…}로 나타낼 수도 있다. 행동도 역시 시간 t의 함수이므로 A = A(t)로 나타낼 수 있으며 A(t)는 시간 t에서의 agent의 행동을 의미한다.

이렇게 상태와 행동을 정의하게 되면 이것을 이용하여 **보상함수**를 정의할 수 있다. 현재시간 t에서의 상태와 행동을 각각 S(t),A(t)라고 한다면 다음단계인 t+1에서의 보상의 평균을 구할 수가 있는데 이것을 현재시각 t에서의 보상으로 정의한다. R_{t+1}은 정책에 의하여 확률적으로 변동이 되는 확률변수이기 때문에 기대값을 구하는 것이다. 보상이란 의미 속에 미래의 가치가 포함되어 있기 때문에 바로 다음 단계에 받을 보상의 평균으로 현재의 보상을 정의한다는 것은 이치에 맞아 보인다. R_{t+1}의 평균을 이용하여 R_t을 정의하고 R_{t+1}은 R_{t+2}의 평균을 이용하여 정의하며 R_{t+2}은 R_{t+1}의 평균을 이용하여 정의하여 … 이런 식으로 프로세스가 끝날 때까지 가게 되어 R_t의 정보 안에는 프로세스의 path의 정보를 잘 반영하는 값이 될 것이다. 이것을 수식으로 표현해보자.

$$R_s^a = \mathrm{E}[R_{t+1}|S(t) = s, A(t) = a]$$

$$P_{ss'}^{a} = \mathrm{P}[S_{t+1} = s' \mid S(t) = s, A(t) = a]$$

다음은 **정책**을 표현하여 보자. 정책은 현재의 상태에서 agent가 선택할 수 있는 행동들을 확률로 표현한 것이다. 물론 선택확률은 학습에 의해서 전체보상이 가장 큰 쪽이 크도록 학습이 된다. 이 선택확률은 $\pi(s,a)$로 나타내며 모든 행동에서 합산한 값은 당연히 1이 된다. 여기에서 π는 정책함수이다. 현재상태 s에서 행동 a를 선택할 확률은 $\pi(a|s)$이다.

$\sum_{a \in A} \pi(s, a) = 1$, A는 가능한 모든 행동공간

$$\pi(a|s) = P[A_t = a \,|S_t = s]$$

이것을 모든 상태에 대하여 정의한 것을 **정책(Policy)**이라고 한다. 우리의 목표는 최적의 정책을 구하는 것인데 최적의 정책은 모든 상태에 대하여 **가치함수**를 최대화하는 정책이다. 그렇다면 **가치함수**는 무엇일까?

우리는 확률적으로 k단계의 시각까지 앞으로 받을 보상의 합을 구할 수가 있을 것이고 이때 앞부분에서 언급했듯이 감가율을 고려하여 합산을 하게 되면 다음과 같다. (γ는 감가율(discount factor))

$$R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \gamma^4 R_{t+5} + \cdots = G_t$$

이것은 현재까지 업데이트된 정책에 의하여 어떤 상태에서 가능한 모든 행동들에 대한 있을 수 있는 모든 보상들의 합을 감가율을 고려하여 현재가치로 환산한 값이다. 이것은 확률이 포함되어 있으므로 G_t는 확률변수이고 따라서 기대값으로 나타내어야 한다.

$$E[G_t|S_t = s] = V_s = \text{현재상태 s에서의 가치함수}$$

즉, 가치함수는 보상값들의 시간적인 총합의 평균이라고 할 수 있다. 그리고 우리의 정책은 가치함수를 최대화하는 방향으로 학습되어야 한다. 우리는 모든 상태에서의 가치함수를 구한 후에 현재상태에서 가능한 행동에 의해서 가능한 상태로 갈 때에 가장 가치함수가 높은 상태로 가면 된다. 위의 수식은 다음과 같이 바뀐다.

$$R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \gamma^4 R_{t+5} + \cdots = G_t$$

$$= R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + \cdots) = R_{t+1} + \gamma G_{t+1}$$

따라서 가치함수는 다음과 같이도 쓸 수 있다. 아래에서 G_{t+1}은 확률변수이므로 기대값으로 표현되어 $E[G_{t+1}|S_t = s] = V_{s'}$로 바꾸어 쓸 수 있다. s'은 다음 단계의 상태이다.

$$E[G_t|S_t = s] = V_s = E[R_{t+1} + \gamma G_{t+1}|S_t = s]$$

$$= E[R_{t+1} + \gamma V_{s'}|S_t = s]$$

가치함수는 항상 정책에 근거하여 업데이트가 되어야 하므로 정책 π의 기호를 붙여서 다음과 같이 표현하면 이것은 **벨만 기대방정식**이 된다.

$$V_s^{\pi} = E_{\pi}[R_{t+1} + \gamma V_{s'}^{\pi} | S_t = s] = E_{\pi}[G_t | S_t = s]$$

우리가 위에서 정의했던 상태전이확률과 보상함수도 정책 π를 고려하여 다음과 같이 바꾸어서 쓸 수 있는데 현재상태 s에서 모든 가능한 행동집합 A를 고려하여 합한 값이다.

$$\sum_{a \in A} \pi(s, a) P^{a}_{ss'} = P^{\pi}_{ss'}$$

$$\sum_{a \in A} \pi(s, a) R^{a}_{s} = R^{\pi}_{s}$$

사실 우리가 구한 V_s^{π}은 상태가치함수이다. 즉, 상태에 대해서만 종속적인 가치함수이기 때문에 모든 상태에 대하여 가치를 구할 수 있어서 어떤 상태로 가야 하는지 명확한 기준을 제공하여 주기 때문에 편리하기는 하지만 이것도 역시 위에 발생한 어떤 문제를 내포하게 된다. 다시 말하여 상태뿐만 아니라 행동에 대하여도 보상이 다를 수가 있기 때문에 가치함수를 상태와 행동의 함수로 재정의를 해야 한다는 요구가 생기게 된다. 이러한 가치함수를 큐함수(Q function)라고 한다.

이것은 상태가치함수보다 더 세밀하게 가치를 구해준다. 즉 어떤 상태에서 어떤 행동에 대하여 가치가 얼마인지를 알게 하여 준다. 이것을 $Q^{\pi}(s,a)$라고 하며 이것을 모든 행동에 대하여 합산을 하면 행동에 대한 변수가 사라지면서 다음과 같이 상태가치함수가 된다.

$$V_s^{\pi} = \sum_{a \in A} \pi(a|s) Q^{\pi}(s,a)$$

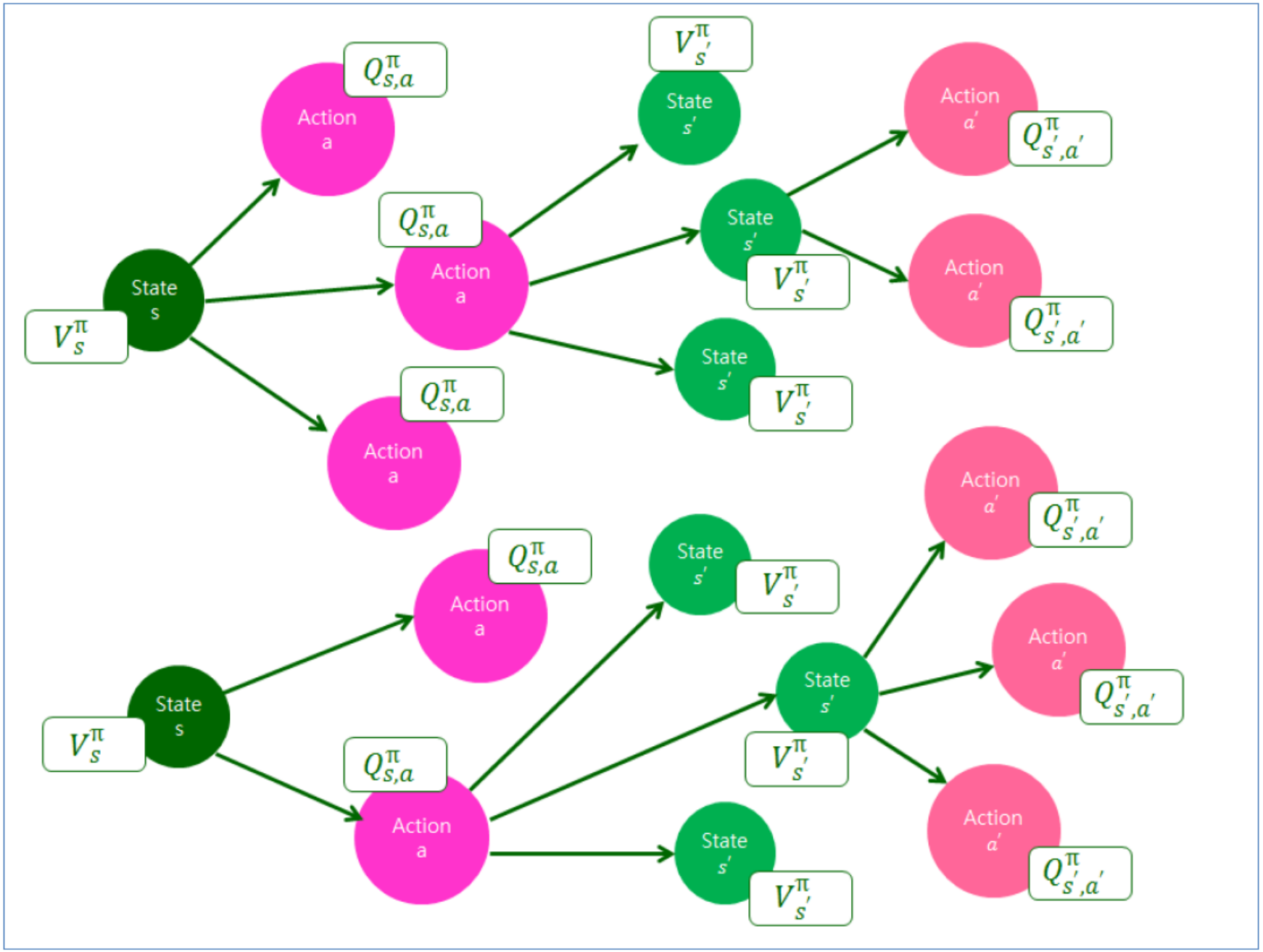
위의 식은 상태가치함수와 Q함수와의 관계를 나타내는 식이기도 하다. $Q^{\pi}(s,a)$는 위에서 언급한 행동을 고려한 상태전이확률과 보상함수를 이용하여 다음과 같이 정의할 수도 있다.

$$Q^{\pi}(s,a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a V_{s'}^{\pi}$$

마지막 2개의 식을 결합하면 다음과 같이 변환된다.

$$V_s^{\pi} = \sum_{a \in A} \pi(a|s) Q^{\pi}(s,a) = \sum_{a \in A} \pi(a|s) \{R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a V_{s'}^{\pi}\}$$

$$V_s^{\pi} = E_{\pi(a|s)}[Q^{\pi}(s,a)]$$


State
s
V_s^π
Action
a
$Q_{s,a}^\pi$
State
s'
$V_{s'}^\pi$
Action
a'
$Q_{s',a'}^\pi$

또한 여기에서 상태 s'에서 상태가치함수와 Q함수와의 관계를 나타내는 식인 다음의 2개의 식을 결합하여 본다.

$$V_{s'}^{\pi} = \sum_{a' \in A} \pi(a'|s') Q^{\pi}(s', a')$$

$$Q^{\pi}(s, a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a V_{s'}^{\pi}$$

$$Q^{\pi}(s,a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a V_{s'}^{\pi} = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \{ \sum_{a' \in A} \pi(a'|s') Q^{\pi}(s',a') \}$$

이 식의 양변에 모든 상태 s와 행동 a에 대하여 합산을 하여 행렬로 나타내면 다음과 같이 변한다. 마지막에 나온 식은 벨만 방정식을 풀어서 V^{π}를 구하는 수학적인 방법을 나타내는데 역행렬이기 때문에 상태공간이 어느 정도 작은 경우에만 직접 계산으로 풀 수가 있을 것이다.

$$V^{\pi} = R^{\pi} + \gamma P^{\pi} V^{\pi}$$

$$\left(\mathrm{I} - \gamma P^{\pi}\right) V^{\pi} = R^{\pi}$$

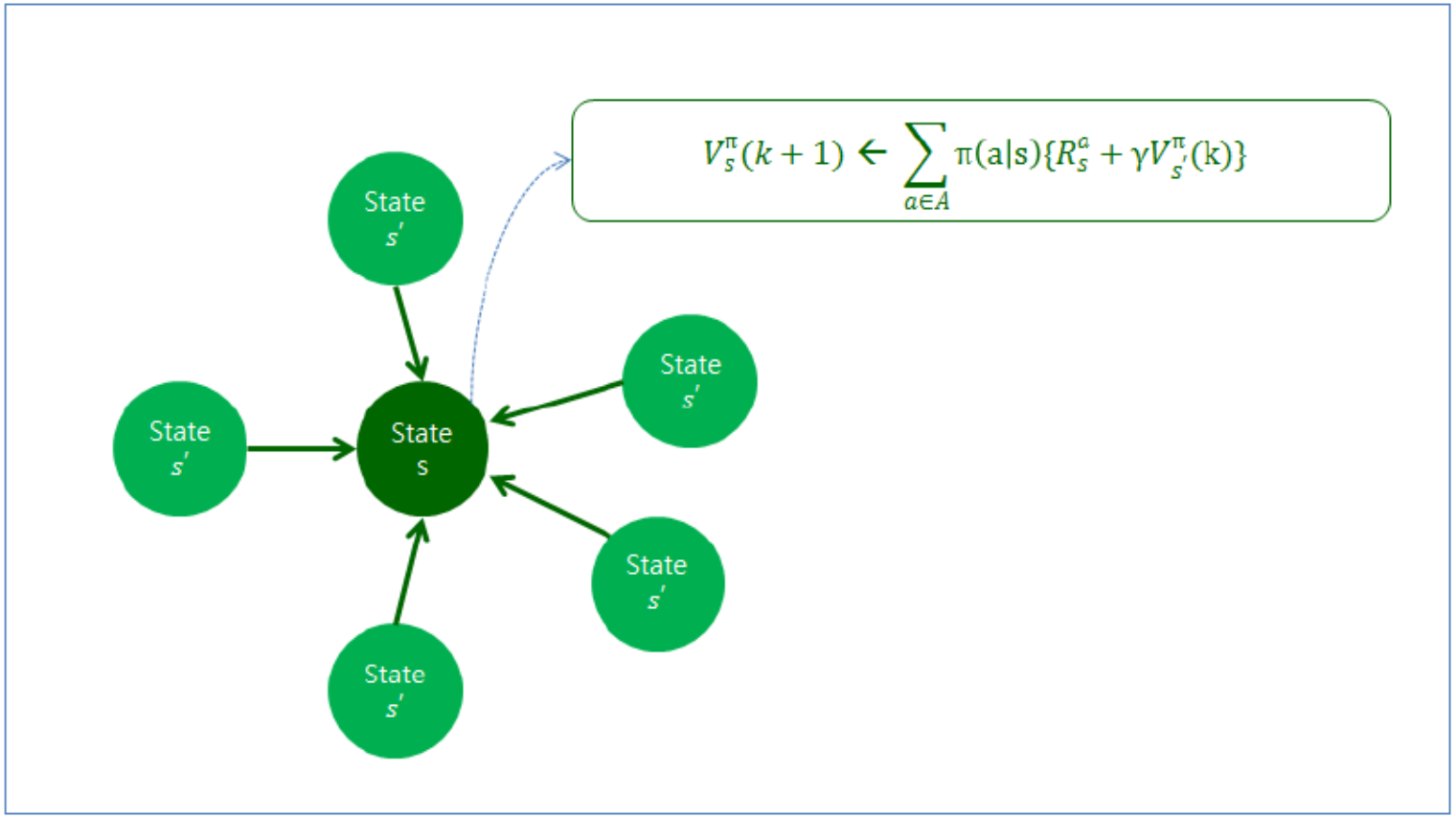
$$\therefore \; V^{\pi} = \left(\mathrm{I} - \gamma P^{\pi}\right)^{-1} R^{\pi}$$

Q함수의 벨만 기대방정식이며 $Q^{\pi}(s,a)$와 $Q^{\pi}(s',a')$의 관계를 나타내어 주고 있다.

$$Q^{\pi}(s,a) = E_{\pi}[R_{t+1} + \gamma Q^{\pi}(s',a') | S_t = s, A_t = a]$$

$$V_s^{\pi} = \sum_{a \in A} \pi(a|s) Q^{\pi}(s,a) = \sum_{a \in A} \pi(a|s) \{R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a V_{s'}^{\pi}\}$$

벨만의 기대 방정식은 다음 단계의 가치함수를 이용하여 현재의 가치함수를 계산하는 방식인데 초기에는 random값으로 시작하지만 차츰 반복하다보면 전체적으로 수렴하게 되는데 이 값이 우리가 구하고자 하는 가치함수이다. 다음처럼 s의 주변에 있는 다음 단계의 s'들로 s에 대한 가치함수 V를 수렴할 때까지 업그레이드 하는 방식이다.



하지만 위의 방식은 정책 π이 변하지 않기 때문에 참값이지만 최적 가치함수를 얻지는 못한다. 최적의 가치함수는 다음처럼 가치함수가 최대가 되는 어떤 정책 π에 대한 가치함수이다. 같은 방법으로 최적의 Q함수도 비슷하게 정의한다.

$$V_*(\mathrm{s}) = max_{\pi} V_s^{\pi}$$

$$Q_*(\mathrm{s},\mathrm{a}) = max_{\pi} Q_{s,a}^{\pi}$$

$$\pi_*(s,a) = \begin{cases} 1, \text{if } a = \ argmax_{a \in A} Q_*(s,a) \\ 0, \qquad \text{otherwise} \end{cases}$$

다음과 같이 최적의 Q함수를 알고 있다면 다음으로 최적의 가치함수를 구할 수가 있다. $Q_*(s,a)$ 이 최적이 아니라면 아무리 max값을 선택해도 $V_*(s)$는 최대가 되지 않는다.

$$V_*(s) = \ max_a[Q_*(s,a) \mid S_t = s, A_t = a]$$

다음은 **가치함수의 벨만 최적방정식**인데 기대방정식에서 최대가 되는 행동 a를 찾는 것이다. $V_*(s)$와 $V_{s'}^*$의 관계를 나타내는 식이기도 하다.

$$V_*(s) = max_a E\left[R_{t+1} + \gamma V_{s'}^* \mid S_t = s, A_t = a\right]$$

$$= max_a E[R_{t+1} + \gamma \sum_{s' \in S} P_{ss'}^a V_{s'}^* | S_t = s, A_t = a] (\text{if } P_{ss'}^a \neq 1)$$

위에서 유도했던 Q함수의 벨만 기대방정식에서 간단히 다음을 유추할 수 있다. 이것은 **Q함수의 벨만의 최적방정식**이라고 한다. 이것은 $Q^*(s,a)$와 $Q^*(s',a')$사이의 관계식이기도 하다.

$$Q^*(s,a) = E[R_{t+1} + \gamma max_{\mathbf{a}} Q^*(s',a') | S_t = s, A_t = a]$$

$$= E[R_{t+1} + \gamma \sum_{s' \in S} P^a_{ss'} max_{\mathbf{a}} Q^*(s',a') | S_t = s, A_t = a] \, (\text{if } P^a_{ss'} \neq 1)$$

동적 프로그래밍(Dynamic Programming)

동적 프로그래밍도 벨만이 생각한 방식이며 Value Iteration과 Policy Iteration의 방법을 쓰는데 각각 벨만의 최적방정식과 기대방정식을 이용하여 문제를 푸는 스타일을 말한다.

정책반복법부터 알아보자. 정책반복법은 정책을 평가하는 과정과 발전시키는 과정인 2가지로 나눌 수 있다. 정책의 평가는 이미 위에서 배웠던 벨만의 기대방정식을 이용한 가치함수의 참값을 구하는 과정이다. 다음과 같은 수식이다. V(1)부터 시작하여 무한히 반복하면 어떤 참값에 수렴하게 된다.

$$V_s^{\pi}(k+1) \leftarrow \sum_{a \in A} \pi(\mathrm{a}|\mathrm{s})\{R_s^a + \gamma V_s^{\pi}(\mathrm{k})\}$$

정책평가는 위의 과정을 충분히 하여 충분히 수렴한 값을 얻어내는 것이다. 즉, 모든 상태 s에 대하여 가치함수를 구하는 것이다. 그리고 다음 단계인 **정책발전**으로 넘어간다. 정책발전의 단계는 정책평가의 단계에서 구한 상태가치함수를 이용하여 최적의 행동을 구하는 것이다. 최적의 행동 선택은 그동안 열심히 배웠던 Q함수의 기대방정식인 다음을 쓸 수가 있다.

$$Q^{\pi}(s,a) = E_{\pi}[R_{t+1} + \gamma Q^{\pi}(s',a')|S_t = s, A_t = a]$$

하지만 기대값을 계산하기는 힘들기 때문에 다음과 같은 형태의 실용적인 Q함수를 사용한다. (위에서 이미 언급했다.)

$$Q^{\pi}(s,a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a V_{s'}^{\pi} = R_s^a + \gamma V_{s'}^{\pi} \ (\text{if } P_{ss'}^a = 1)$$

위의 식을 이용하여 초기의 무작위의 수로 세팅이 되어있던 정책 π를 다음과 같이 업데이트한다. 즉 현재의 상태에서 가장 큰 Q함수를 가지는 행동을 찾는 것이다. 만약 행동들 중에서 같은 Q함수를 가지는 것이 2개이상 있다면 같은 확률로 선택하도록 만들면 된다.

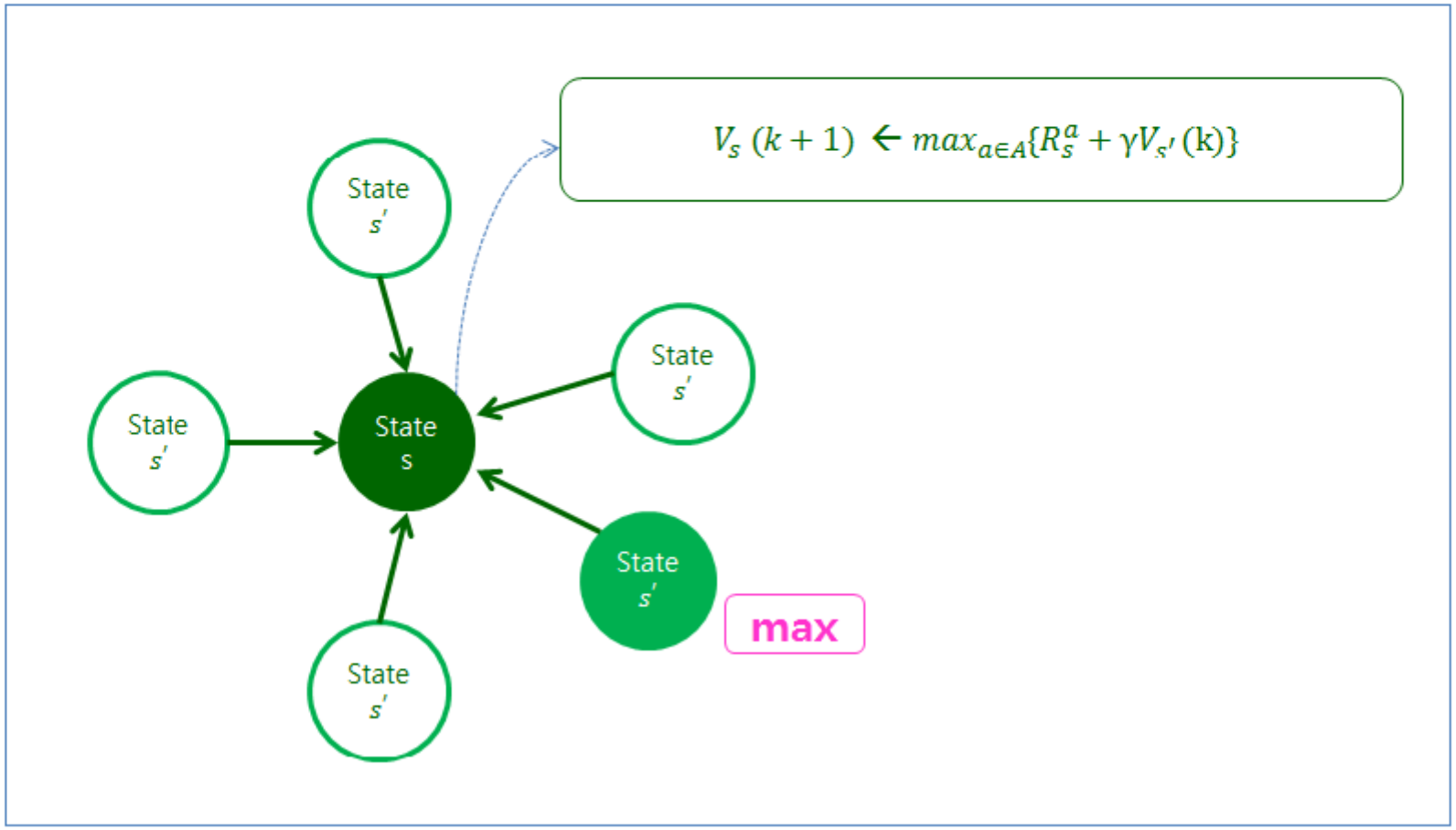
$$\pi(s) = argmax_{a \in A} Q_{\pi}(s, a)$$

정책평가는 k를 1단계로 하여 정책평가와 정책발전을 번갈아 가면서 볼 수 있도록 프로그래밍을 할 수도 있고 정책평가를 k를 크게 하여 어느 정도 충분히 수렴한 후에야 정책발전을 시도하게 만들 수도 있지만 이것은 프로그래머의 자유이다.

다음은 **가치반복법**을 알아보도록 하자. 이것은 정책발전은 생략하고 최적의 가치를 구하는데에 초점이 맞추어져 있다. 따라서 다음과 같은 벨만의 최적방정식을 사용한다. 최적방정식은 모든 정책 π를 고려하기 때문에 index에 π는 없다.

$$V_*(s) = max_a E\left[R_{t+1} + \gamma V_{s'}^* \,\middle|\, S_t = s, A_t = a\right]$$

정책반복법에서 가치를 구할 때에는 다음 단계의 상태 s'에서의 가치를 고려하여 기대값을 구했지만 **가치반복법**은 다음 단계의 s'에서의 가치를 고려하여 max값 하나만 뽑아서 업데이트를 하게 된다. 다음의 그림을 보면 이해가 된다.



가치반복법은 정책을 저장하는 변수는 따로 두지 않는다. 왜냐하면 상태마다 가치함수를 업데이트할 때에 π(a|s)이란 항이 필요가 없기 때문이다. (기대값이 아니라 최대값을 저장하기 때문이다.) 그리고 가치함수를 업데이트하고 난 후에 최적의 행동을 선택하는 부분은 **정책반복법**에서 사용했던 방법과 동일하게 다음과 같이 Q함수를 쓴다. (π는 사라진다.)

$$Q(s,a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a V_{s'} = R_s^a + \gamma V_{s'} \ (\text{if } P_{ss'}^a = 1)$$

$$selected\ action = argmax_{a \in A} \mathrm{Q(s,a)}$$

벨만이 만든 이 방식은 환경에 대한 완벽한 정보가 있어야 한다는 가정이 있기 때문에 현실문제를 푸는데 적합하지 않다는 단점이 발견되었다. 게다가 상태공간의 크기가 늘어남에 따라 계산복잡도가 기하급수적으로 증가하는 것도 큰 문제가 되었다. 그럼 이 문제를 해결하는 몇가지 다른 방법을 공부해보자.

몬테카를로 방법(Monte-Carlo Prediction)

위에서 언급한 다음의 수식을 보자. 이것은 벨만의 기대방정식으로 가치함수를 업데이트하는 방법을 알려준다.

$$V_s^{\pi} = \sum_{a \in A} \pi(a|s) Q^{\pi}(s,a) = \sum_{a \in A} \pi(a|s) \{ R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a V_{s'}^{\pi} \}$$

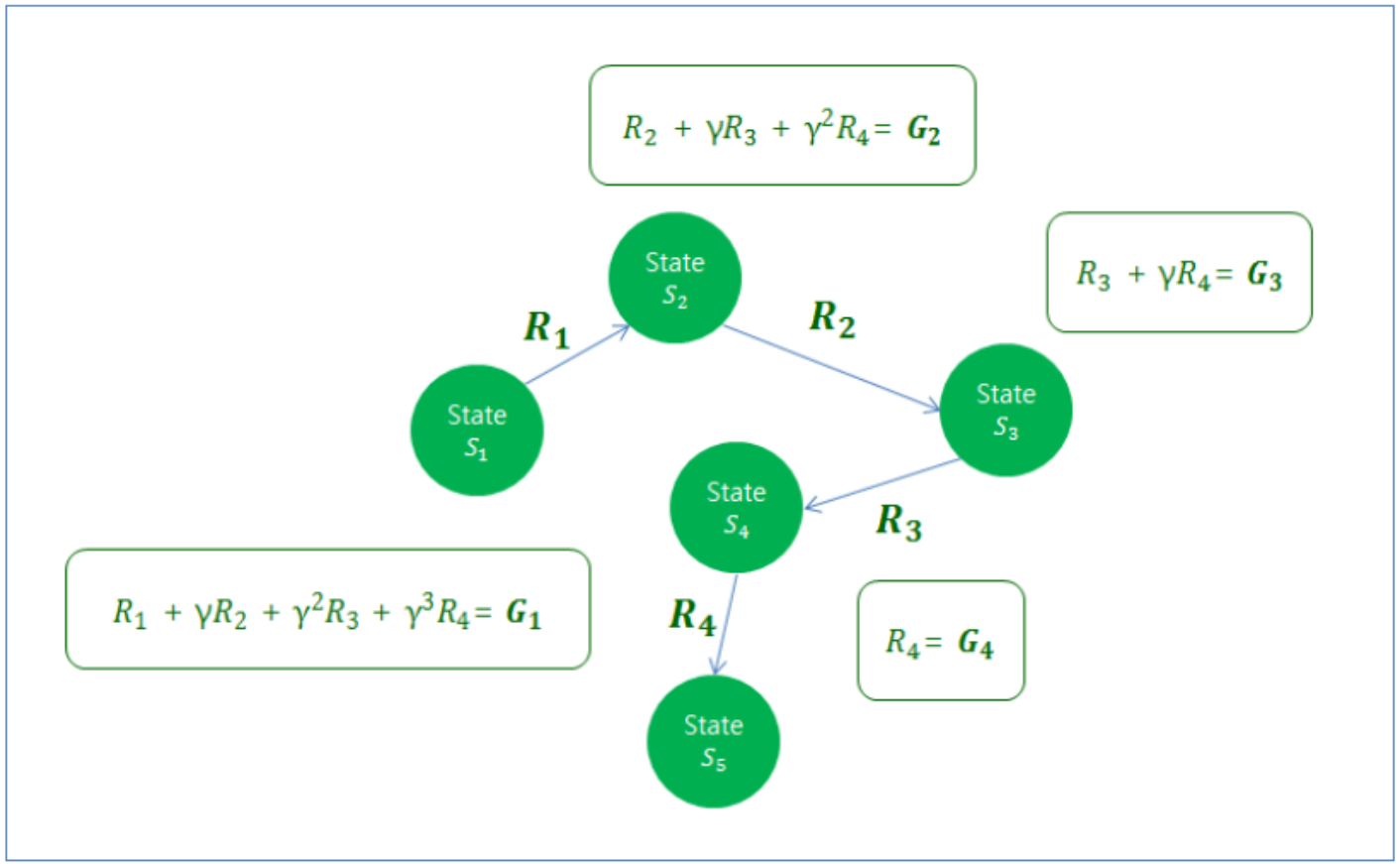
문제는 우리가 처음부터 R_s^a와 $P_{ss'}^a$을 알 길이 없다는 것이다. 이것들은 환경이 주는 변수로서 우리가 알아내야 하는 값이라는 점이 문제가 된다. 다시 말하여 우리는 환경의 모델을 몰라도 가치함수를 구할 수 있는 다른 방법을 찾아야 한다.

여기에서 쓸 수 있는 방법은 몬테카를로 방법이다. 우리는 어느 정도의 충분한 시간단계까지의 agent가 탐험을 하도록 시켜서 다음의 값을 얻을 수가 있다. G_t는 시간 t시점에서 예상한 전체시간에서의 Reward이다.

$$R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \gamma^{T-t+1} R_T = G_t$$

위의 수식을 이용하여 다음의 그림처럼 agent가 탐험을 하면서 여러 개의 sample G_t를 얻을 수가 있다. 여기에서는 G_1, G_2, G_3, G_4를 얻었다.

위의 수식을 이용하여 다음의 그림처럼 agent가 탐험을 하면서 여러 개의 sample G_t를 얻을 수가 있다. 여기에서는 G_1, G_2, G_3, G_4를 얻었다.



이렇게 한번의 path를 훑고 지나가면 한번의 에피소드가 완성이 되는데 이것을 여러 번 반복한다면 하나의 상태 s에서만 여러 개의 G_s를 얻을 수가 있다. 상태 s의 방문횟수를 N_s이라고 하고 $N_s \gg 1$ 이라면 이 값들의 평균값으로 상태 s에서의 가치함수 V_s로 추정을 해도 별 무리가 없다. 즉 이것은 다음과 같이 쓸 수가 있다.

$$V_s \approx \frac{1}{N_s} \sum_{k=1}^{N_s} G_k^s$$

위의 식은 다시 다음과 같이 전개할 수가 있다. n은 상태 s의 방문횟수이다.

$$V_{n+1} = \frac{1}{n}\sum_{k=1}^{n} G_k = \frac{1}{n}(G_n + \sum_{k=1}^{n-1} G_k)$$

$$= \frac{1}{n}(G_n + \frac{n-1}{n-1}\sum_{k=1}^{n-1} G_k) = \frac{1}{n}(G_n + V_n(n-1))$$

$$= V_n + \frac{1}{n}(G_n - V_n)$$

$$V_{new} \leftarrow V_{old} + \frac{1}{n}(G_n - V_{old})$$

위의 수식의 의미는 agent가 탐험을 하면서 G를 계속 얻으면서 V를 업데이트한다는 것이다. 이때 새로운 V는 새로 얻은 G와 이전의 V의 차이에 비례하는 값만큼 더하여 얻는다. $\frac{1}{n}$은 0과 1사의 값으로 바꾸어 쓸 수 있으며 이것은 학습률 η가 된다.

$$V_{new} \leftarrow V_{old} + \eta(G_n - V_{old})$$

한번의 에피소드가 진행될 때마다 agent가 방문한 모든 상태에 대한 V를 업데이트하고 또다시 에피소드를 반복한다. 이런 과정을 반복한다. 이것도 역시 단점이 존재하는데 에피소드 하나가 끝나야 한번 업데이트를 하는 구조이기 때문에 에피소드 길이가 긴 경우에는 시간이 오래 걸린다는 단점이 있다.

시간차 예측방법(Temporal-Difference Prediction,TD)

시간차 예측방법은 에피소드가 끝날 때까지 기다리지 않고 한 단계의 시간 뒤의 sample로 가치 함수의 기대값을 구하는 방법이다.

$$V_s^{\pi} = E_{\pi}[R_{t+1} + \gamma V_{s'}^{\pi} | S_t = s] = E_{\pi}[G_t | S_t = s]$$

위의 기대값을 몬테카를로 방식으로 구하고 싶다면 아주 많은 sample $(R_{t+1} + \gamma V_{s'}^{\pi})$의 값을 얻어서 평균을 내면 된다. s'은 바로 다음 단계의 상태을 의미하므로 한 단계만 전진하여 경험치를 얻으면 되기 때문에 에피소드의 길이와 상관없이 실시간으로 바로 업데이트가 가능하다. 따라서 바로 전 절에서 설명한 몬테카를로 방식에서 G_t대신에 $R + \gamma V_{s'}$를 쓰면 된다. $R + \gamma V_{s'} - V_s$을 시간차 에러라고 부른다.

$$V_{new} \leftarrow V_s + \eta(R + \gamma V_{s'} - V_s)$$

시간차 예측방법은 sample이 충분할 때에 참가치함수에 수렴하며 몬테카를로 예측보다 더 빠른 것으로 알려져있다.

살사(SARSA)와 Q러닝(Q-Learning)

여기서 살사는 스포츠댄스가 아니다. 앞에서 설명한 시간차예측방법을 살짝 변형시킨 것이다. 위의 시간차 예측방법에서 가치함수대신에 Q함수를 적용하여 보자. 우리가 얻는 가치는 상태뿐만 아니라 행동에도 영향을 미치기 때문이라고 설명을 했었다. Q함수의 업데이트의 식은 다음과 같다.

$$Q_{new} \leftarrow Q_{s,a} + \eta(R + \gamma Q_{s',a'} - Q_{s,a})$$

위의 식을 알기 위해서는 a, a'도 같이 알아야 한다. 즉, 모두 s, a, R, s', a'을 알아야 한다. 그래서 SARSA라는 이름이 붙여졌는데 a'을 얻기 위해서는 어쩔 수 없이 s'에서 한 단계 더 나아가야 한다. 이렇게 상태와 행동마다 Q함수값을 저장하는 테이블을 만들어 놓고 현재상태에서 가장 높은 Q함수값을 갖는 행동을 선택하게 되는데 ε의 비율만큼 이것을 무시하고 무작위의 행동을 선택하도록 만든다. 이것은 '**탐험**'을 통하여 좀 더 좋은 해를 구하기 위한 방법이다. 어느 정도 경험이 쌓이면 ε의 수치를 점점 줄여야 좋은 효과를 볼 수 있다. 그렇지 않으면 좋은 해를 이미 구하고도 계속 무작위 탐험으로 시간을 낭비할 수도 있기 때문이다. 이것을 ε-greedy정책이라고 한다.

살사에서 시뮬레이션을 하다보면 ε-greedy정책이 오히려 독이 되는 경우를 발견할 수가 있는데 초기에 경험이 없는 상태에서 무작위로 행동을 하다가 잘못된 학습으로 Q함수을 업데이트하는 경우가 생겨 오랫동안 잘못된 해에 갇혀버리는 경우가 생길 수가 있다. 거기에서 빠져 나오려면 Q함수를 무시하고 ε의 비율로 무작위 시도를 오랫동안 시도를 해야 한다. 이 단점을 극복한 것이 바로 다음에 설명할 Q-Learning이다.

위의 문제를 해결하는 방법은 의외로 간단하다. 상태 s에서 a를 구할 때에는 ε-greedy정책을 쓰는 것은 살사와 동일하지만 그 다음의 상태 s'에서는 이것을 허용하지 않고 가장 큰 Q함수값을 가지는 a'을 선택한다. 그렇게 하여 얻은 s, a, R, s', a'을 가지고 Q함수를 업데이트한다. 즉 수식은 다음과 같이 된다.

$$Q_{new} \leftarrow Q_{s,a} + \eta(R + \gamma max_{a'} Q_{s',a'} - Q_{s,a})$$

실제로 agent가 행동을 할 때에는 ε-greedy정책을 하지만 Q함수를 업데이트할 경우에는 a'의 경우는 허용하지 않게 되는데 이렇게 agent의 행동정책과 학습정책이 다른 경우를 off-Policy라고 한다. 살사는 on-Policy에 해당한다.

Deep SARSA

지금까지는 일반적인 강화학습에 관한 내용이라면 지금부터는 신경망을 들어가는 경우이다. 지금까지 배운 내용으로도 어느 정도의 성과 있는 문제를 풀 수는 있지만 방대한 상태정보를 가지고 변화하는 환경과 같은 복잡한 문제는 풀기가 힘들다. 지금까지는 Q함수가 어떤 상태와 행동에서 가치가 얼마인지를 테이블형태로 만들어 놓고 agent의 행동기준을 정했지만 그렇게 단순하게는 이런 복잡하고 어려운 문제를 풀 수가 없다. 이때 필요한 것이 딥러닝이다. 우리가 자동운행시스템을 만든다고 가정하고 현재의 자동차의 속도와 장애물들의 상대속도와 위치 뿐만 아니라 신호등이나 교통표지판의 인식등 수많은 상태들에 따라 가장 올바른 행동을 실시간으로 결정해야 한다면 지금까지의 방법으로는 불가능하다.

딥러닝을 이용하여 모든 상태에 대한 Q함수를 한꺼번에 구한다면 cost function을 먼저 정의를 해야 한다. 위의 SARSA에서 이미 다음과 수식을 사용했었다.

$$Q_{new} \leftarrow Q_{s,a} + \eta(R + \gamma Q_{s',a'} - Q_{s,a})$$

따라서 $R + \gamma Q_{s',a'}$가 목표값이고 $Q_{s,a}$가 예측값이라고 생각한다면 이것은 error가 되며 이것을 cost function으로 정의하면 된다.

Error function = $(R + \gamma Q_{s',a'} - Q_{s,a})^2$

나머지는 SARSA와 동일하다. 즉, s, a, R, s', a'을 먼저 추출한 후에 딥러닝에 입력값을 s로 하고 cost function은 위처럼 정의해주고 딥러닝 모델에 넣으면 자동으로 상태에 맞는 최적의 Q함수값을 output으로 나오게 되는데 output node의 개수는 Action의 경우의 수만큼 설정하고 input node의 개수는 State의 개수만큼 넣으면 된다. 단, 여기에서 활성화함수는 선형함수를 써야 한다.

Monte-Carlo Policy Gradient

이번에는 Deep SARSA와 비슷하지만 output값이 해당 행동의 선택에 대한 확률로 나타나도록 해보자. 즉 이것은 정책의 정의 $\pi(a|s)$와 같다. 이때 활성화함수는 softmax와 같은 함수를 쓰는 것이 좋을 것이다. 신경망의 가중치를 W라고 한다면 목표함수는 어떤 정책 π와 상태 s에 대한 가치함수 $V_\pi(s)$라고 하면 된다. $V_\pi(s)$이 최대가 되도록 W를 갱신시키는 방법은 무엇일까? 다음과 같은 Policy Gradient라고 불리우는 식을 사용한다.

$$W_{t+1} \leftarrow W_t + \eta \nabla_w V_\pi(s)$$

위와 같은 gradient ascent를 쓰면 되는데 1부에서 배워서 이미 알고 있지만 ∇_w은 w에 대한 미분연산자이다. 하지만 우리가 원하는 것은 모든 상태 s와 행동 a에 대하여 $V_\pi(s)$이 최대가 되는 것이므로 다음과 같이 $\sum$기호를 써야 한다. 여기에서 $d_\pi(s)$는 agent가 상태 s에 있을 확률분포이다. (상태분포라고 한다.) 또한 우리가 구하고자 하는 것은 정책신경망이므로 π와 Q함수는 W에 종속이 되기 때문에 각각 π_w, Q_w라고 표기한다. 우리는 앞부분에서 가치함수와 Q함수의 관계식을 살펴보았는데 다음과 같았다.

$$V_s^{\pi} = \sum_{a \in A} \pi(a|s) Q^{\pi}(s,a)$$

그리고 우리는 모든 상태 s에 대하여 V_s^{π}가 최대가 되는 가중치 W를 찾는 것이 목적이므로 다음과 같이 정의가 되어야 한다.

$$\nabla_w V = \sum_s d_\pi(s) \nabla_w V_s^\pi$$

V_s^π을 치환을 하면 다음과 같이 된다.

$$\nabla_w V = \sum_s d_\pi(s) \sum_{a \in A} \nabla_w \pi_w(a|s) Q_\pi(s,a)$$

위의 수식의 의미는 모든 상태 s와 모든 행동 a에 대하여 $Q_w(s,a)$에 대한 gradient로 정의한 것이다. 위의 수식은 다음과 같이 바꾸어 전개가 가능하다.

위의 수식의 의미는 모든 상태 s와 모든 행동 a에 대하여 $Q_w(s,a)$에 대한 gradient로 정의한 것이다. 위의 수식은 다음과 같이 바꾸어 전개가 가능하다.

$$\nabla_w V = \sum_s d_\pi(s) \sum_{a \in A} \frac{\pi_w(a|s)}{\pi_w(a|s)} \nabla_w \pi_w(a|s) Q_\pi(s,a)$$

$$= \sum_s d_\pi(s) \sum_{a \in A} \pi_w(a|s) \frac{\nabla_w \pi_w(a|s)}{\pi_w(a|s)} Q_\pi(s,a)$$

$\frac{\nabla_w \pi_w(a|s)}{\pi_w(a|s)}$은 log함수의 미분형태이므로 $\nabla_w(\log \pi_w(a|s))$이 되므로 다음과 같이 쓸 수 있다.

$$= \sum_s d_\pi(s) \sum_{a \in A} \pi_w(a|s) \nabla_w(\log \pi_w(a|s)) Q_\pi(s,a)$$

여기에서 $\sum_s d_\pi(s) \sum_{a \in A} \pi_w(a|s)..$ 이 부분이 사전확률*조건부확률의 곱의 형태이므로 모든 상태 s와 모든 상태 a에서의 s와 a의 결합확률분포와 같다. 즉, $\sum_{s,a} p(s,a)..$ 의 형태이므로 위의 수식은 $\nabla_w(\log \pi_w(a|s))Q_w(s,a)$의 평균임을 알 수가 있다. 다음과 같이 바꿔보자.

$$\nabla_w V = \mathrm{E}[\nabla_w(\log \pi_w(a|s))Q_\pi(s,a)]$$

기대값이기 때문에 몬테카를로 방식으로 sampling하여 평균값을 구하면 된다.

$$W_{t+1} \leftarrow W_t + \eta \nabla_w V_\pi(s)$$

$$\leftarrow W_t + \eta\{\nabla_w(\log \pi_w(a|s))Q_\pi(s,a)\}$$

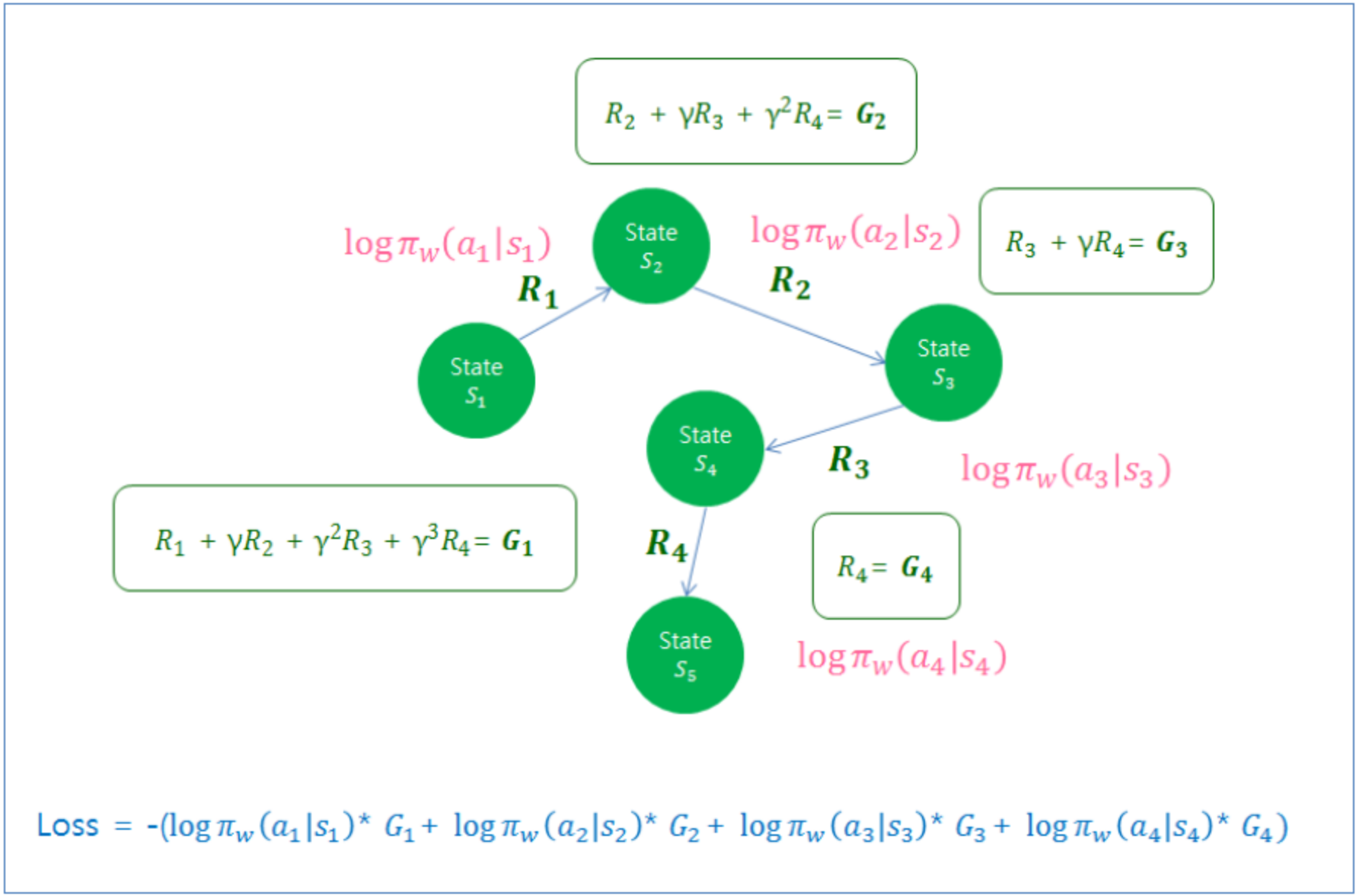
결국 $\nabla_w(\log \pi_w(a|s))Q_\pi(s,a)$의 값을 구하여 가중치를 계속 업데이트하면 된다. 하지만 우리는 여기에서 Q함수나 가치함수를 사용하지 않기로 했기 때문에 $Q_\pi(s,a)$을 G_t로 대체한다. 그리고 G_t는 몬테카를로 방식으로 배웠듯이 탐험을 통하여 sampling하여 얻어낸다. 이렇게 정책신경망을 구하기 위하여 몬테카를로 방식으로 Policy gradient를 구한다고 하여 지금과 같은 방식을 이 절의 제목처럼 Monte-Carlo Policy Gradient방식이라고 한다.

$$W_{t+1} \leftarrow W_t + \eta\{\nabla_w(\log \pi_w(a|s))G_t\}$$

여기에서 $\nabla_w(\log \pi_w(a|s))G_t$이 어떤 의미를 가지고 있는지 생각해보자. G_t는 W와 무관하고 상태 s에서 구한 값임에 명심하자.

$$\nabla_w(\log \pi_w(a|s))G_t = \nabla_w(\log \pi_w(a|s)\, G_t)$$

$\pi_w(a|s)$은 신경망이 구한 output값이며 여기에 target값인 action은 one hot vector라고 하자. 그러면 $-\sum_i \text{action}_i * \log \pi_w(a_i|s)$은 cross entropy가 된다. cross entropy는 오류함수이며 이것이 최소가 되어야 한다. 이것은 $-\log \pi_w(a|s)$와 동일하게 되고 거기에 따른 보상인 G_t와 $+\log \pi_w(a|s)$을 곱하면 이 값은 최대가 되어야 한다. 따라서 Loss function으로 $-\log \pi_w(a|s)\, G_t$로 설정하고 정책신경망을 업데이트하면 된다. 다음의 그림을 보면 이해가 더 빠를 것이다. 한번의 에피소드로 다음과 같은 agent가 탐색을 하였다면 gradient를 구하기 위하여 다음과 같이 계산한다. 4번의 화살표가 있다면 4개의 $\log \pi_w(a|s)$와 4개의 G_t가 생길 것이다. 이것을 모두 각각 곱한 후 더한 값에 (-1)을 곱하면 신경망에 설정해야 하는 loss함수가 된다.



$$\text{Loss} = -(\log \pi_w(a_1|s_1)^* G_1 + \log \pi_w(a_2|s_2)^* G_2 + \log \pi_w(a_3|s_3)^* G_3 + \log \pi_w(a_4|s_4)^* G_4)$$

이렇게 신경망으로 정책함수를 설계하는 방식을 Policy gradient라고 하며 특히 위처럼 G_t를 사용하는 방식을 Reinforce알고리즘이라고 한다.

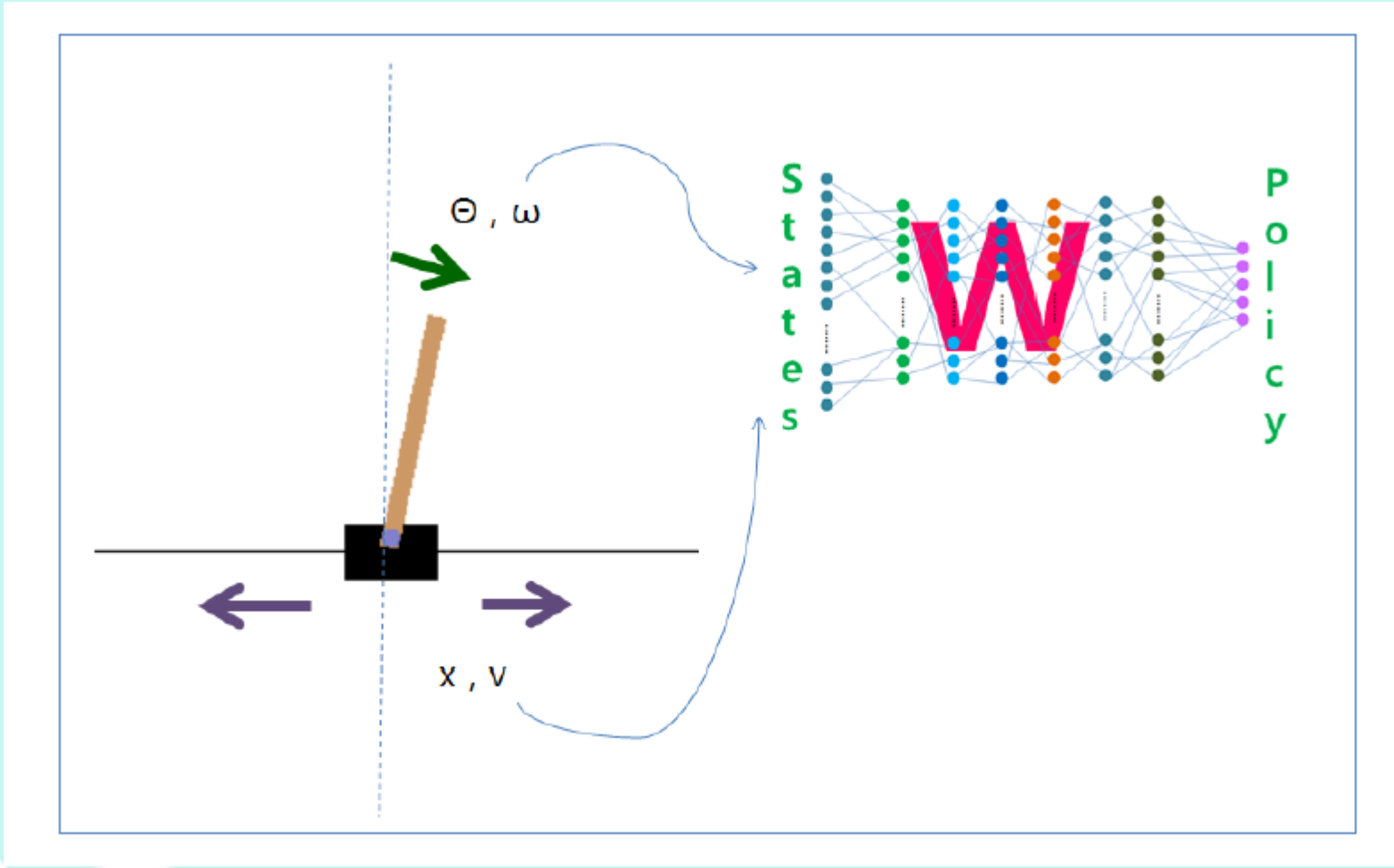
DQN

In [1]:

```
import gym

env = gym.make('CartPole-v0')

for i_episode in range(20):
    observation = env.reset()

    for t in range(100):
        env.render()
        print(observation)
        action = env.action_space.sample()
        observation, reward, done, info = env.step(action)

        if done:
            print("Episode finished after {} timesteps".format(t+1))
            break
```

```
[ 0.09874684  0.93287511 -0.07522848 -1.44894202]
[ 0.11740434  1.12883694 -0.10420732 -1.76414973]
[ 0.13998108  1.32497147 -0.13949031 -2.0873371 ]
[ 0.16648051  1.52119949 -0.18123705 -2.41969747]
Episode finished after 13 timesteps
[ 0.04599577 -0.04725685  0.00777614 -0.04485642]
[ 0.04505064  0.14775273  0.00687901 -0.3350758 ]
```

위의 python코드는 다음과 같은 화면을 띄우는데 내부적으로 강화학습 알고리즘이 돌아가고 있으며 여기에서는 DQN으로 어떻게 설계할 수 있는지 보도록 하자. agent의 상태는 그림에서와 같이 현재위치 x,속도 v,막대의 각도 θ,막대의 각속도 ω로 보면 될 것 같으며 이것은 신경망의 입력값으로 들어 갈 것이고 output값은 당연히 카트의 효율적인 움직임이다. Action이 일어날 경우의 수는 3가지가 되며 {왼쪽,정지,오른쪽}이 전부일 것이다. 이 3가지에 대한 정책이나 Q함수값이 출력이 되도록 신경망을 구성하면 된다.


Θ , ω
x , v
States
W
Policy

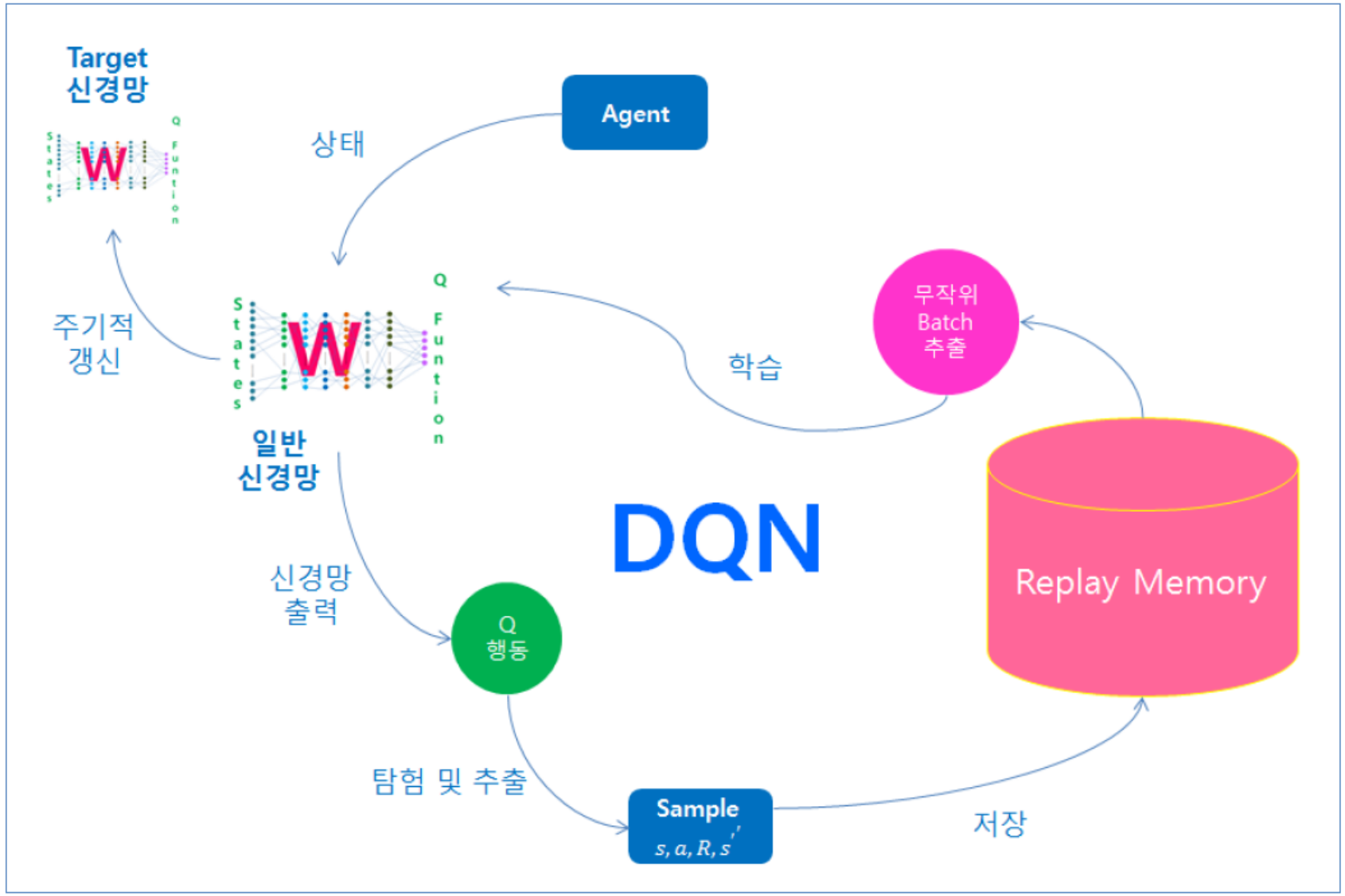
DQN은 딥마인드에서 2013년에 발표된 알고리즘으로 딥살사를 조금 개선시킨 것이다. 이미 알고 있겠지만 딥살사는 Q함수를 근사시키는 신경망을 사용한다. 이때 신경망의 학습은 sample이 생길때마다 진행지만 DQN은 학습데이터를 한꺼번에 메모리에 저장을 놓았다가 에피소드가 끝나면 한꺼번에 넣어서 훈련시킨다. 이런 방법을 경험 리플레이(Experience Replay)라고 하며 이때 사용하는 메모리를 Replay Memory라고 한다. 또 한가지 다른 점은 신경망을 학습할 때에 쓰이는 오류함수의 정의인데 Q러닝에서의 방법을 그대로 가져온다. 다음은 Q러닝에서 적용했던 Error함수이다.

$$Q_{new} \leftarrow Q_{s,a} + \eta(R + \gamma max_{a'} Q_{s',a'} - Q_{s,a})$$

여기에서 Q함수는 신경망의 출력값이므로 가중치 W의 함수가 된다. 그래서 다음과 같이 변형된다. 신경망의 에러함수는 다음과 같이 MSE의 형태가 된다.

$$Error\ Funtion = (R + \gamma Q_{s',a',W} - Q_{s,a,W})^2$$

여기에서 또 한가지 다른 점을 적용을 해야 하는데 바로 신경망을 2개로 만든다는 점이다. 하나의 신경망은 일반적으로 agent가 행동을 하거나 학습을 하기 위하여 얻어야 하는 Q함수를 위한 신경망이고 또 하나는 **target신경망**으로 부르며 위의 수식에서 $Q_{s',a',W}$을 얻기 위하여 사용한다. 즉, 다음 상태 s'에 해당하는 Q함수를 얻기 위하여 사용한다. 그 이유는 일반적인 신경망은 time step마다 학습을 하기 때문에 위의 에러함수은 학습할 때마다 값이 바뀌면서 변동이 심하여 신경망이 불안정하게 된다. 더불어서 agent의 행동도 불안정하게 된다. 이것을 방지하기 위하여 target에 해당하는 $R + \gamma Q_{s',a',W}$의 부분은 target신경망을 사용하여 얻어내며 target신경망은 에피소드가 끝날 때마다 일반신경망과 동기화를 시켜준다. (target신경망 ← 일반신경망) 일반신경망의 업데이트는 메모리에 충분한 sample이 쌓이게 되면 time step마다 무작위로 batch를 추출하여 훈련하여 학습시킨다. 그림으로 그리면 다음과 같은 모양이 된다.


Target
신경망
States
W
Q Function
Agent
상태
주기적
갱신
States
W
Q Function
일반
신경망
무작위
Batch
추출
학습
DQN
Replay Memory
신경망
출력
Q
행동
탐험 및 추출
Sample
s, a, R, s'
저장

실제로 게임에서 강화학습을 적용할 때에는 상태값과 보상값을 어떻게 설계할 것인지 에피소드의 단위가 되는 이벤트(캐릭터가 죽을 때까지? 또는 성공할 때까지?)는 어떻게 정할 것인지를 결정해야 하며 성공과 실패의 정의도 결정해야 한다. 예를 들면 Cart Pole의 경우에는 막대기가 쓰러지면 실패이고 어느 시간 동안 쓰러지지 않으면 성공으로 하고 2가지의 이벤트가 일어난 경우에 에피소드가 종료된다는 규칙을 세우는 것으로 알고 있다. 사실 이 모든 환경에 관한 문제는 openAI의 gym에서 이미 설정이 되어 있는 경우가 많다. DQN에서는 위에서 정의한 Error함수를 쓰지 않고 target신경망의 가중치를 W'이라고 한다면 다음처럼 정의를 하게 된다.

$$Error\ Funtion = (R + \gamma Q_{s',a',W'} - Q_{s,a,W})^2$$

Advantage Actor-Crtic(A2C)

앞에서 배웠던 Policy Gradient역시 단점을 가지고 있다. 한번의 에피소드가 끝날 때까지 기다렸다가 에러함수가 하나 만들어지고 신경망이 학습을 하기 때문에 오차의 편차가 심하고 학습속도도 느리다. 리처드 서튼이 만든 Actor-Critic알고리즘은 time step마다 신경망을 업데이트하며 이런 단점을 극복하고 개선해였다. Policy Gradient에서 정책신경망의 갱신은 다음과 같다는 사실을 배웠었다.

$$W_{t+1} \leftarrow W_t + \eta\{\nabla_w(\log \pi_w(a|s))Q_\pi(s,a)\}$$

$$W_{t+1} \leftarrow W_t + \eta\{\nabla_w(\log \pi_w(a|s))G_t\}$$

$Q_\pi(s,a)$대신 G_t를 쓰는 것이 Reinforce 알고리즘이라고 배웠다. 그리고 Policy Gradient의 목적은 정책신경망으로 정책함수를 근사시키는 것이라고 하였다. 여기서 한걸음만 더 나아가 보자. 우리가 다루기 힘들어 했던 $Q_\pi(s,a)$도 역시 신경망으로 예측을 하여 보자. 이것을 가치신경망이라고 하자. 이것은 앞에서 배웠던 **정책반복법**에서 두가지 전략인 **정책평가**와 **정책발전**을 생각나게 만든다. 즉 가치신경망이 정책평가의 역할을 한다면 정책신경망은 정책발전의 역할을 한다고 볼 수도 있다. 이렇게 하여 위의 수식을 다시 고쳐서 쓸 필요성이 있는데 그 이유는 $Q_\pi(s,a)$는 가치신경망에 의해서 결정되므로 가치신경망의 가중치 행렬 W'으로 index를 바꿔주어야 한다. 따라서 업데이트 수식은 다음과 같이 된다.

$$W_{t+1} \leftarrow W_t + \cap\{\nabla_w(\log \pi_w(a|s))Q_{W'}(s,a)\}$$

Actor-Crtic은 이렇게 두 개의 신경망을 동시에 가지고 있게 된다. 즉, W의 가중치를 가지는 정책신경망과 W'의 가중치를 가지는 가치신경망이다. Actor가 행동을 결정하는 정책신경망을 의미하고 Critic은 평가를 하는 책임을 가지고 있는 가치신경망을 의미한다. 그리고 $-\log \pi_w(a|s)$은 정책신경망의 error function이 되는 cross entropy임을 이미 알고 있다. 2개의 신경망 모두 input으로 status가 들어가며 output로는 정책신경망은 정책 π,가치신경망은 Q함수를 출력한다. Actor-Crtic에서의 오류함수는 $-(\log \pi_w(a|s))Q_{W'}(s,a)$이 된다. 이것을 줄이는 방법을 생각해야 한다.

이제는 $Q_{W'}(s,a)$에 대해서 생각을 해보자. 이미 입력값으로 상태가 정해진 경우이므로 상태 s는 고려할 필요가 없고 행동 a에 따라서 Q값이 좌우될 것이다. 이미 정해진 상태에서 Action에 의해서 얼마나 손익을 보았는지 계산을 해보려면 $Q_{W'}(s,a) - V(s)$를 하면 될 것이다. 이것을 advantage함수라고 하는데 실제의 변화량을 의미하므로 이것으로 $Q_{W'}(s,a)$을 대체해도 상관은 없을 것이다.

$$\text{Advantage(s,a)} = Q_{W'}(s,a) - V(s) = \text{A(s,a)}$$

앞부분에서 Q함수를 공부하면서 다음과 같이 가치함수와의 관계가 표현된다는 것을 배웠었다.

$$Q^{\pi}(s,a) = R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a V_{s'}^{\pi} = R_s^a + \gamma V_{s'}^{\pi} \text{ (if } P_{ss'}^a = 1)$$

이것을 위의 식에 대입하여 보자.

$$A(s,a) = Q_{W'}(s,a) - V(s) = R_s^a + \gamma V_{s'} - V(s)$$

이것은 앞의 TD알고리즘에서 배웠던 시간차 에러의 수식과 같다. 이것을 δ_v라고 하고 Actor-Crtic의 update식에 대입하면 다음과 같이 된다.

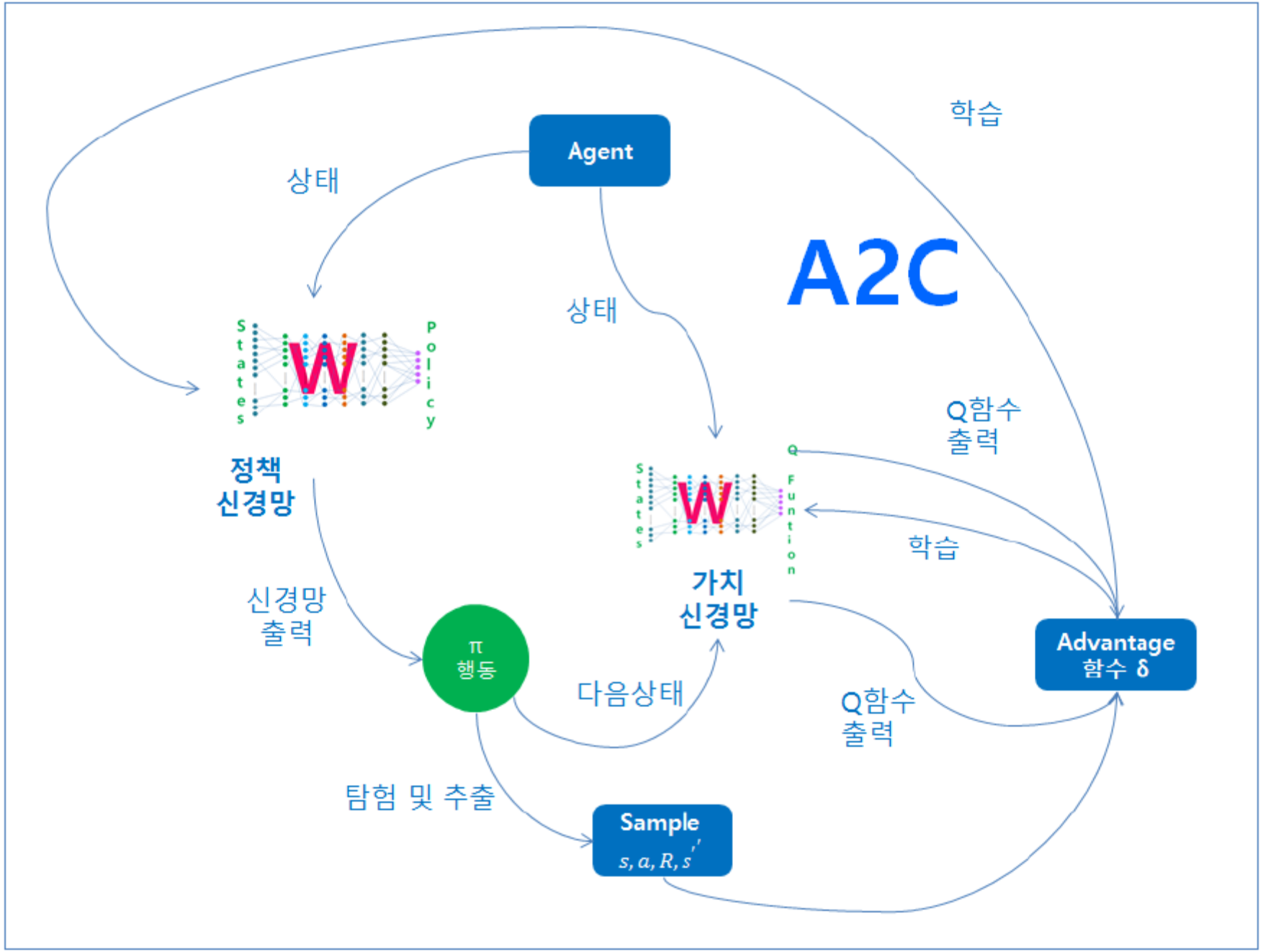
$$W_{t+1} \leftarrow W_t + \eta\{\nabla_w(\log \pi_w(a|s))\delta_v\}$$

δ_v를 오류로 하여 먼저 가치신경망을 업데이트하면 최적의 Q를 출력하는 신경망이 만들어지는데

이때 오류함수는 MSE는 다음과 같이 정의한다.

$$\text{MSE} = \delta_v^{\,2} = (R_s^a + \gamma V_{s'} - V(s))^2$$

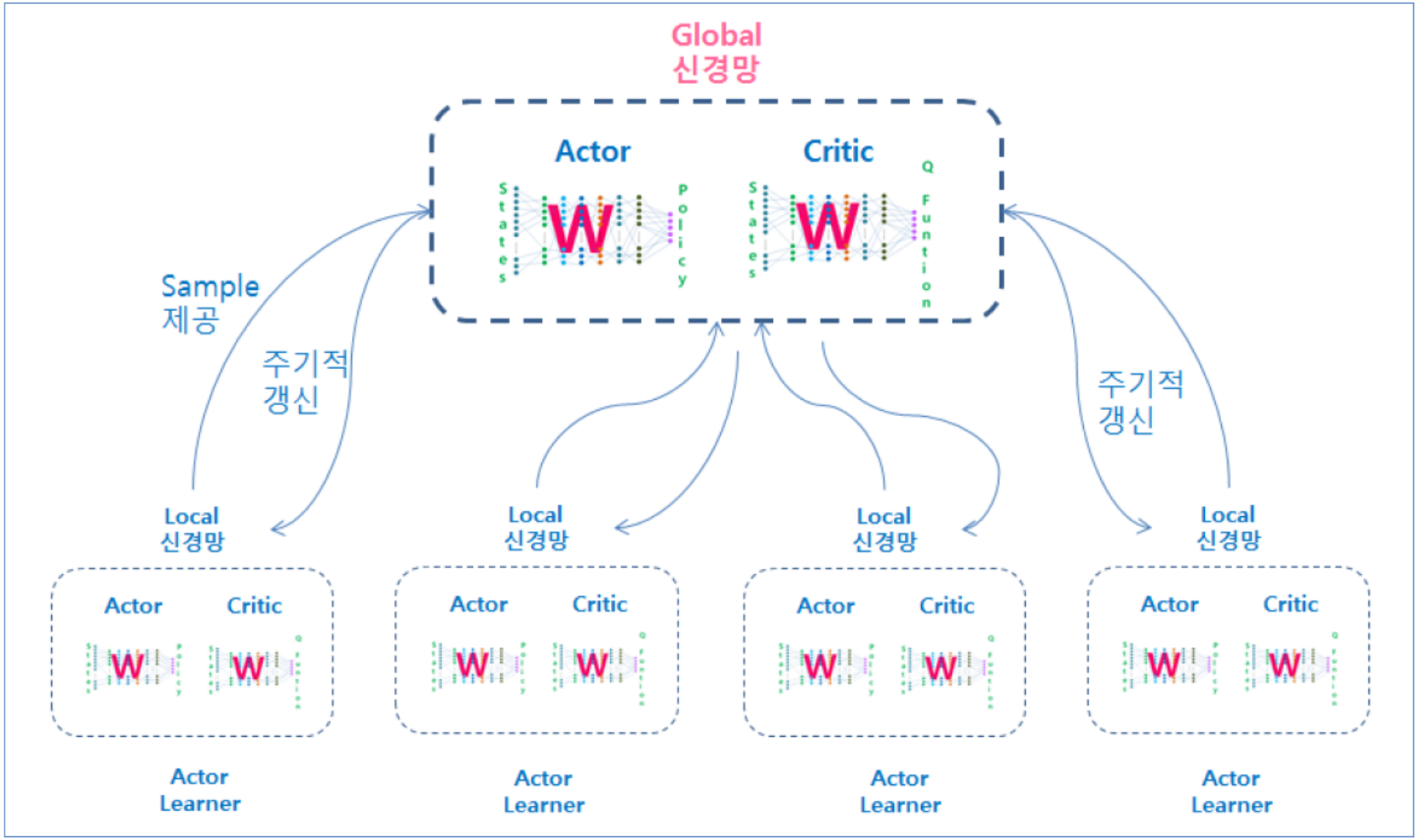
동시에 시간차 에러 δ_v를 가지고 Actor-Critic의 오류함수인 $(\log \pi_w(a|s))\delta_v$을 이용하여 정책신경망을 학습하면 된다. 2개의 신경망의 훈련순서는 의미가 없으며 동시에 이루어진다. Actor-Critic이 Advantage함수를 사용한다고 하여 A2C(Advantage Actor-Critic)라고도 한다. 다음은 지금까지의 flow를 알기 쉽게 정리한 그림이다.


학습
Agent
상태
A2C
상태
States
W
Policy
정책
신경망
Q함수
출력
States
W
Q Function
가치
신경망
학습
신경망
출력
π
행동
다음상태
Q함수
출력
Advantage
함수 δ
탐험 및 추출
Sample
s, a, R, s'

신경망의 입력값에 들어가는 status가 꼭 어떤 값들의 집합일 필요는 없다. 알파고처럼 바둑판모양이나 게임화면자체를 통째로 상태값으로 입력이 가능하다. output값은 보통 똑같이 Q함수값이나 정책함수를 출력한다. 입력정보는 정지화면이나 연속화면 뿐만 아니라 동영상도 가능하다. 한가지 중요한 점은 정책신경망에서의 활성화함수는 softmax와 같은 함수를 쓰지만 Q함수를 출력하는 가치신경망에서는 확률값이 아니기 때문에 활성화함수로 linear함수를 쓴다는 사실이다.

Asynchronous Advantage Actor-Crtic(A3C)

DQN의 Replay Memory방식은 매우 큰 규모의 데이터를 필요로 하는 딥러닝에서는 메모리문제로 학습속도의 저하의 원인이 된다. 딥마인드는 이것의 단점의 극복을 하면서 sample간의 상관관계를 훨씬 더 저하시키는 A3C라는 알고리즘을 논문으로 발표했다.
이것은 Actor-Learner라고 하는 수많은 서로 다른 local신경망이 서로 다른 환경에서 게임을 하면서 얻어낸 sample데이터를 주기적으로 global신경망에 업데이트를 하도록 한다. 여기에서 얻어낸 sample들은 서로 영향을 미치지 않으므로 양질의 데이터이며 잘못된 정책에 빠지는 경우가 생기지 않을 뿐더러 메모리 문제에서도 벗어나도록 해준다. 각각의 local신경망을 생성하기 위하여 멀티프로세싱이나 멀티스레딩이라는 프로그래밍 기법을 쓰기도 한다. 다음의 그림을 보면 이해가 쉽다.


Global
신경망
Actor
Critic
States
Policy
Q
Function
Sample
제공
주기적
갱신
주기적
갱신
Local
신경망
Local
신경망
Local
신경망
Local
신경망
Actor
Critic
Actor
Learner
Actor
Learner
Actor
Learner
Actor
Learner

Actor Learner가 부르는 각각의 local신경망은 옆에 있는 local신경망을 전혀 신경 쓰지 않고 탐험을 하면서 sample을 만들어 낸다. 한 에피소드가 끝나거나 어느 정도의 time step이 지나면 이 sample들을 Global신경망이 훈련을 하여 A2C방식으로 update한다. update하자마자 다시 해당하는 local신경망을 update한다. 이 과정들이 수많은 local신경망들이 병렬로 계속 반복된다. 이것이 비동기로 진행하는 A2C이기 때문에 Asynchronous라는 말이 붙어서 A3C알고리즘이라고 부른다. A2C와 다른 점은 A2C처럼 Actor와 Critic을 신경망을 따로 구성한 것이 아니고 공용으로 쓰이며 output layer와 활성화함수만 다르게 한다.

즉 Actor는 정책신경망이므로 행동의 경우의 수 만큼 노드를 구성하고 확률을 출력해야 하므로 softmax와 같은 함수를 쓰는 반면에 Critic은 평가를 하는 가치신경망이므로 Q함수를 출력해야 하기 때문에 linear함수를 활성화함수로 쓰고 단 하나의 값만 출력하도록 노드의 개수가 1개이다.

1-step 시간차에러

$$A(s,a) = R_s^a + \gamma V_{s'} - V(s)$$

multi-step 시간차에러

$$A(s,a) = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \gamma^4 V_{t+4} - V(s)$$

Boltzmann 선택

'**탐험**'의 비중을 조정하기 위하여 ε-greedy방식을 사용하기도 하지만 볼츠만 선택이란 방법도 효율적이다. 마치 에너지함수와 같은 공식으로 되어 있는데 정책의 선택확률이 Q함수값에 비례하는 정도 β를 조정하여 탐험의 비중을 정한다. 다음과 같은 수식을 활용한다.

$$\pi(a|s) \propto e^{\beta Q(s,a)}$$

또는 π의 (-1)*Entropy를 오류함수에 더하는 방법도 사용된다. Entropy는 행동에 대한 확률이 모두 같을 때에 최대가 되므로 그럴수록 '탐험'을 증가시키는 효과가 있기 때문이다. ('탐험'이란 모든 행동에 골고루 기회를 주는 정도라고도 정의가 가능하다. 반대로 greedy정책은 한 행동만을 선택하는 경향을 가진다.)

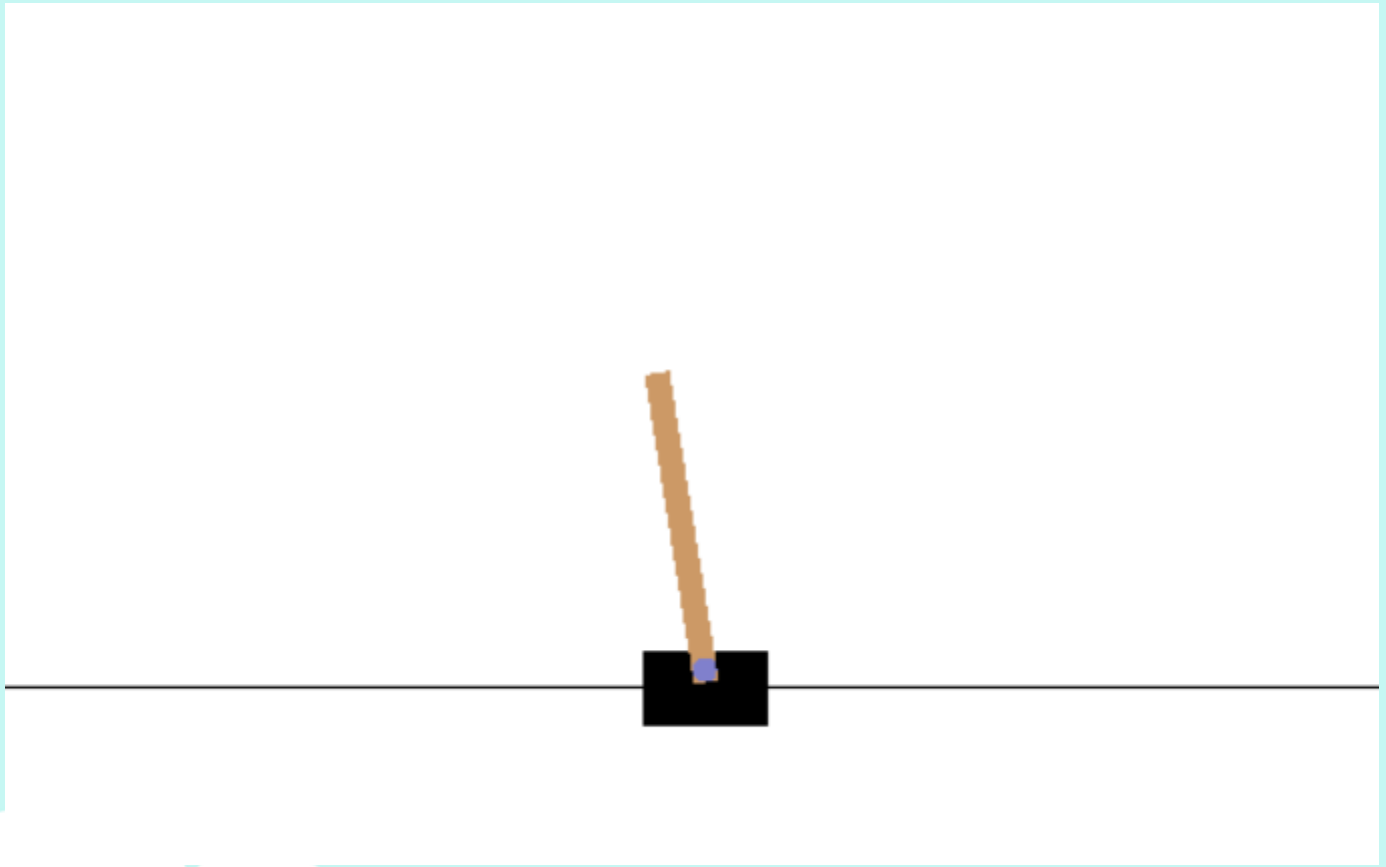
알파고의 경우에는 강화학습을 하기 전에 기계학습으로 프로기사들이 할 수 있는 있는 action 361개를 추출하였다. 위에서 배운 바와 같이 정책신경망과 가치신경망을 사용하였으며 Policy Gradient방식을 사용하기도 하였다고 한다. 이것으로 Deep RL의 설명을 모두 마친다.

In [1]:

```
import gym

env = gym.make('CartPole-v0')

for i_episode in range(20):
    observation = env.reset()

    for t in range(100):
        env.render()
        print(observation)
        action = env.action_space.sample()
        observation, reward, done, info = env.step(action)

        if done:
            print("Episode finished after {} timesteps".format(t+1))
            break
```

```
[-0.00486743 -0.04015013 -0.04691254  0.04406436]
[-0.00567043 -0.23456907 -0.04603125  0.3215847 ]
[-0.01036181 -0.03882287 -0.03959956  0.014748  ]
[-0.01113827 -0.2333552  -0.0393046   0.29467851]
[-0.01580537 -0.03769557 -0.03341103 -0.01013683]
```

```
[-0.04012713  0.40491376  0.02014392 -0.52610737]
[-0.03202886  0.59974655  0.00962177 -0.81237535]
[-0.02003393  0.40449413 -0.00662574 -0.51668153]
[-0.01194405  0.59970875 -0.01695937 -0.81144499]
[ 5.01295854e-05  7.95058872e-01 -3.31882694e-02 -1.10941385e+00]
[ 0.01595131  0.60038838 -0.05537655 -0.82732447]
[ 0.02795907  0.40606558 -0.07192304 -0.55255947]
[ 0.03608039  0.60211998 -0.08297423 -0.86700791]
[ 0.04812279  0.40821922 -0.10031438 -0.60152385]
[ 0.05628717  0.214633   -0.11234486 -0.34204733]
[ 0.06057983  0.41115909 -0.11918581 -0.66793805]
[ 0.06880301  0.6077191  -0.13254457 -0.99564414]
[ 0.08095739  0.41459484 -0.15245745 -0.74735274]
[ 0.08924929  0.22186812 -0.16740451 -0.50626491]
[ 0.09368665  0.02945162 -0.1775298  -0.27066233]
[ 0.09427569  0.22660392 -0.18294305 -0.61366284]
[ 0.09880776  0.034446   -0.19521631 -0.38372164]
[ 0.09949668  0.23172603 -0.20289074 -0.73105064]
Episode finished after 20 timesteps
```

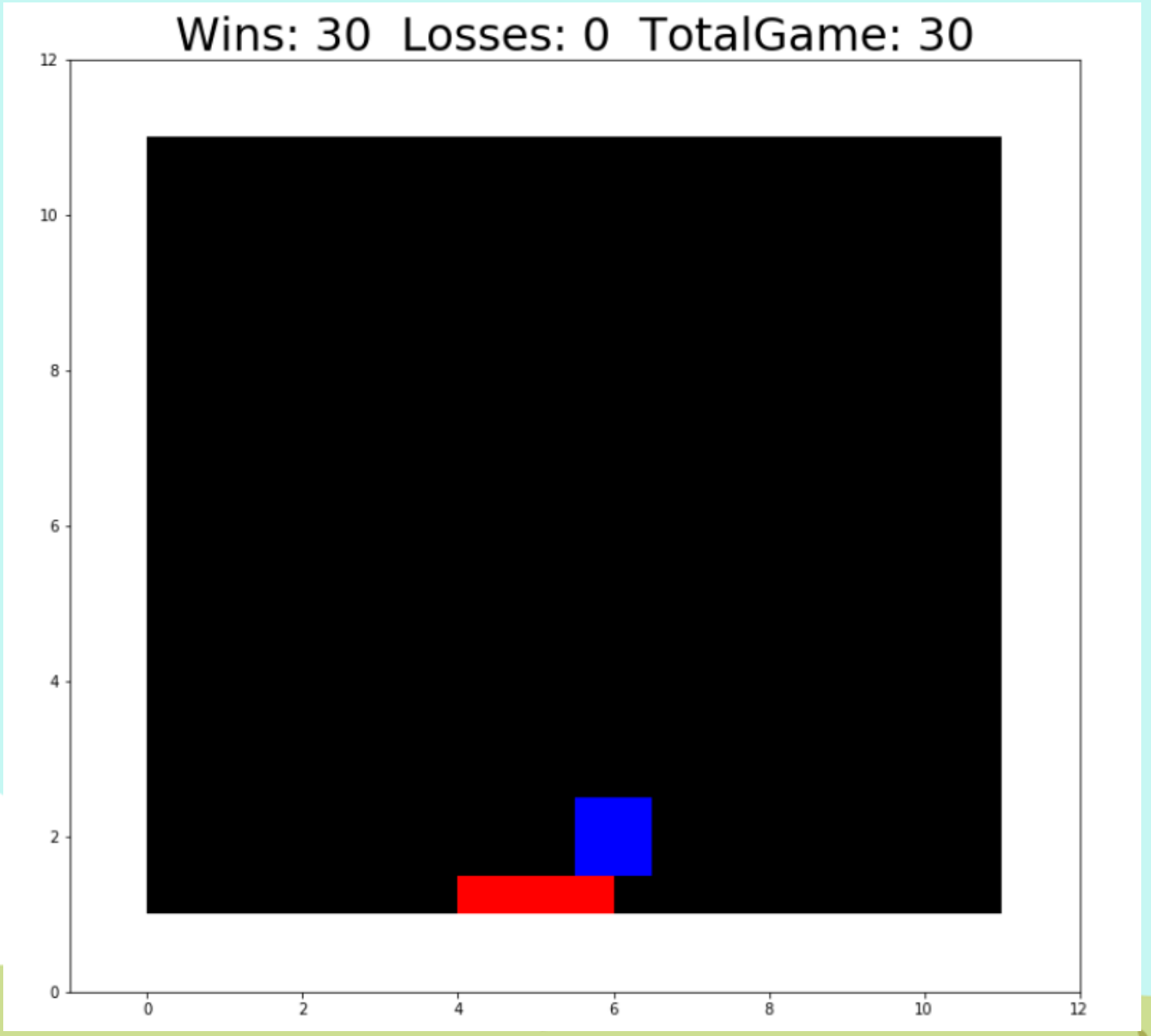


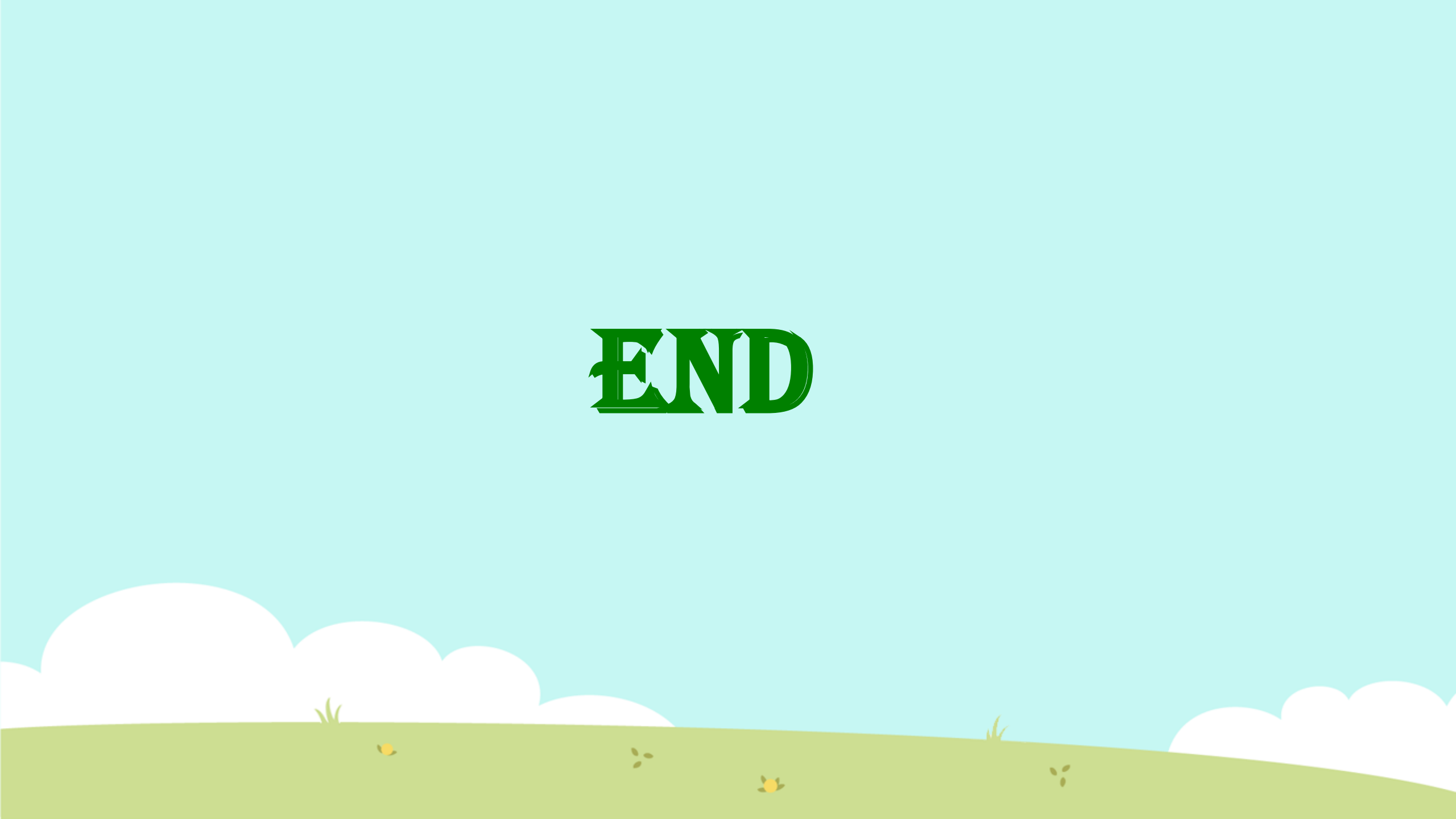
```
# 설정값들을 정의합니다.
gridSize = 10
maxGames = 30
env = CatchEnvironment(gridSize)
winCount = 0
loseCount = 0
numberOfGames = 0

# 화면을 그리기 위한 설정들을 정의합니다.
ground = 1
plot = pl.figure(figsize=(12,12))
axis = plot.add_subplot(111, aspect='equal')
axis.set_xlim([-1, 12])
axis.set_ylim([0, 12])

# 파라미터를 불러오기 위한 tf.train.Saver() class를 선언합니다.
saver = tf.train.Saver()
```

```
# 현재 상태를 그리기 위한 drawState 함수를 정의합니다.
def drawState(fruitRow, fruitColumn, basket, gridSize):
  # 과일이 몇번째 세로축에 있는지 정의합니다.
  fruitX = fruitColumn
  # 과일이 몇번째 가로축에 있는지 정의합니다.
  fruitY = (gridSize - fruitRow + 1)
  # 승리 횟수, 패배 횟수, 전체 게임 횟수를 화면 상단에 출력합니다.
  statusTitle = "Wins: " + str(winCount) + "  Losses: " + str(loseCount) + "  TotalGame: " + str(numberOfGames)
  axis.set_title(statusTitle, fontsize=30)
  for p in [
    # 배경의 위치를 지정합니다.
    patches.Rectangle(
        ((ground - 1), (ground)), 11, 10,
        facecolor="#000000"       # Black
    ),
    # 바구니의 위치를 지정합니다.
    patches.Rectangle(
        (basket - 1, ground), 2, 0.5,
        facecolor="#FF0000"     # Red
    ),
    # 과일의 위치를 지정합니다.
    patches.Rectangle(
        (fruitX - 0.5, fruitY - 0.5), 1, 1,
        facecolor="#0000FF"       # Blue
    ),
    ]:
      axis.add_patch(p)
  display.clear_output(wait=True)
  display.display(pl.gcf())
```

```
with tf.Session() as sess:

  # 저장된 파라미터를 불러옵니다.
  saver.restore(sess, os.getcwd()+"/model.ckpt")
  print('저장된 파라미터를 불러왔습니다!')

  # maxGames 횟수만큼 게임을 플레이합니다.
  while (numberOfGames < maxGames):
    numberOfGames = numberOfGames + 1

    # 최초의 상태를 정의합니다.
    isGameOver = False
    fruitRow, fruitColumn, basket = env.reset()
    currentState = env.observe()
    drawState(fruitRow, fruitColumn, basket, gridSize)

    while (isGameOver != True):
      # 현재 상태를 DQN의 입력값으로 넣고 구한 Q값중 가장 큰 Q값을 갖는 행동을 취합니다.
      q = sess.run(y_pred, feed_dict={x: currentState})
      action = q.argmax()

      # 행동을 취하고 다음 상태로 넘어갑니다.
      nextState, reward, gameOver, stateInfo = env.act(action)
      fruitRow = stateInfo[0]
      fruitColumn = stateInfo[1]
      basket = stateInfo[2]
```

```
        # 과일을 받아내면 winCount를 1 늘리고 과일을 받아내지 못하면 loseCount를 1 늘립니다.
        if (reward == 1):
          winCount = winCount + 1
        elif (reward == -1):
          loseCount = loseCount + 1

        currentState = nextState
        isGameOver = gameOver
        drawState(fruitRow, fruitColumn, basket, gridSize)
        # 다음 행동을 취하기 전에 0.05초의 일시정지를 줍니다.
        time.sleep(0.05)

# 최종 출력결과 이미지를 하나로 정리합니다.
display.clear_output(wait=True)
```


Wins: 30 Losses: 0 TotalGame: 30
12
10
8
6
4
2
0
0
2
4
6
8
10
12


END