Reinforcement Learning

CS231n Lecture 14

ToBigs 14기 이정은

목차

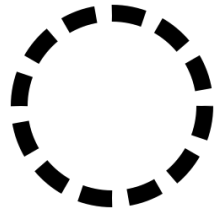
Reinforcement Learning

목적 : 최대 reward을 받을 수 있는 action을 취하도록 학습시키는 것

= 최적의 정책 함수를 찾는다

Reinforcement Learning

environment
강화학습 전체



state 상태

agent가 action을
선택하기 위해 받는 정보

agent 주체

environment에서
action을 취하는 주체

action 행동

agent가 특정 state에서
취할 수 있는 행동

reward 보상

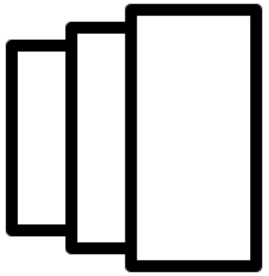
agent가 어떤 action을
취하면 주는 보상

목적 : 최대 **reward**을 받을 수 있는 **action**을 취하도록 학습시키는 것

Reinforcement Learning

Policy 정책

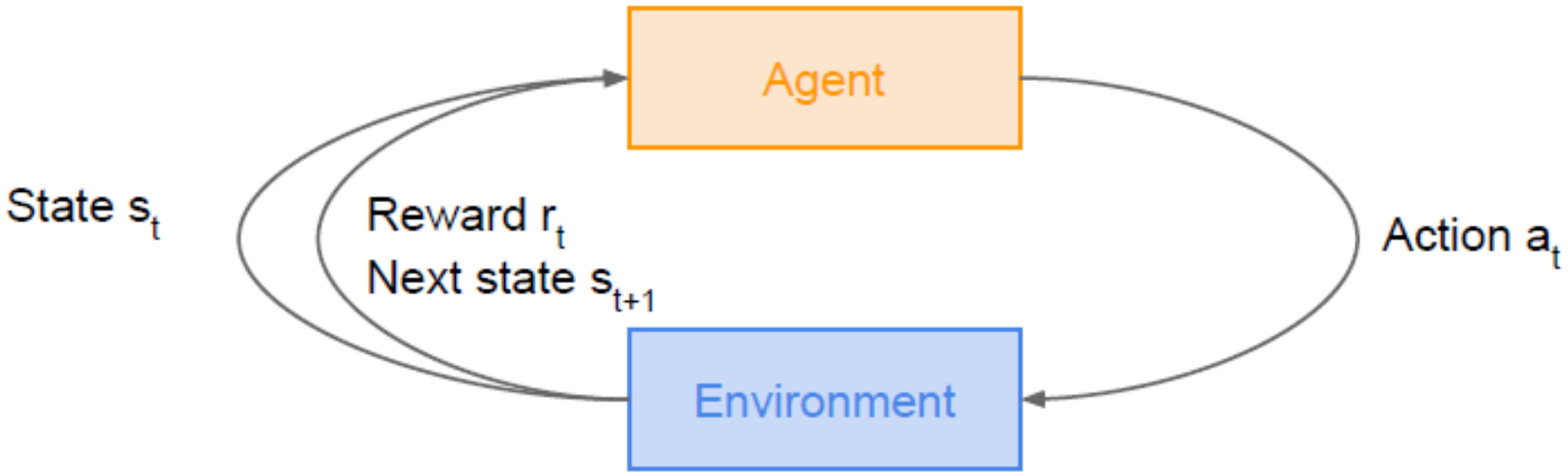
특정 state에서 action을 골라주는 함수



Episode 에피소드

시작부터 종료까지 agent가 거친 (state, action, reward)의 sequence

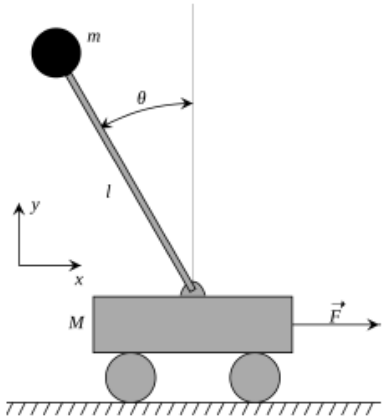
Reinforcement Learning



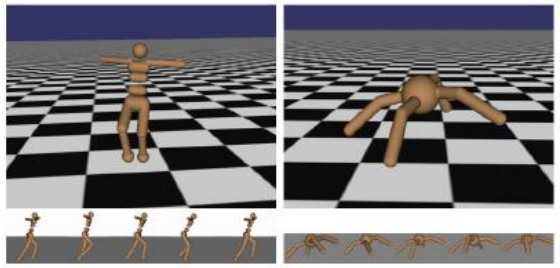
Reinforcement Learning

Example

Cart-Pole Problem

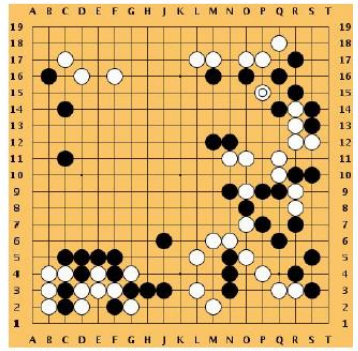


Robot Locomotion



Atari Games

Go



Reinforcement Learning

Markov Decision Process

(MDP, 마르코프 의사결정 과정) : 강화학습 문제를 수식화

Markov Property (마르코프 특징)

현재가 주어졌을 때, 과거와 미래는 독립
-> t+1번째 state는 오로지 t번째의 state와 action에만 의존

Difined by $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathbb{P}, \gamma)$

S : state의 집합

A : action의 집합

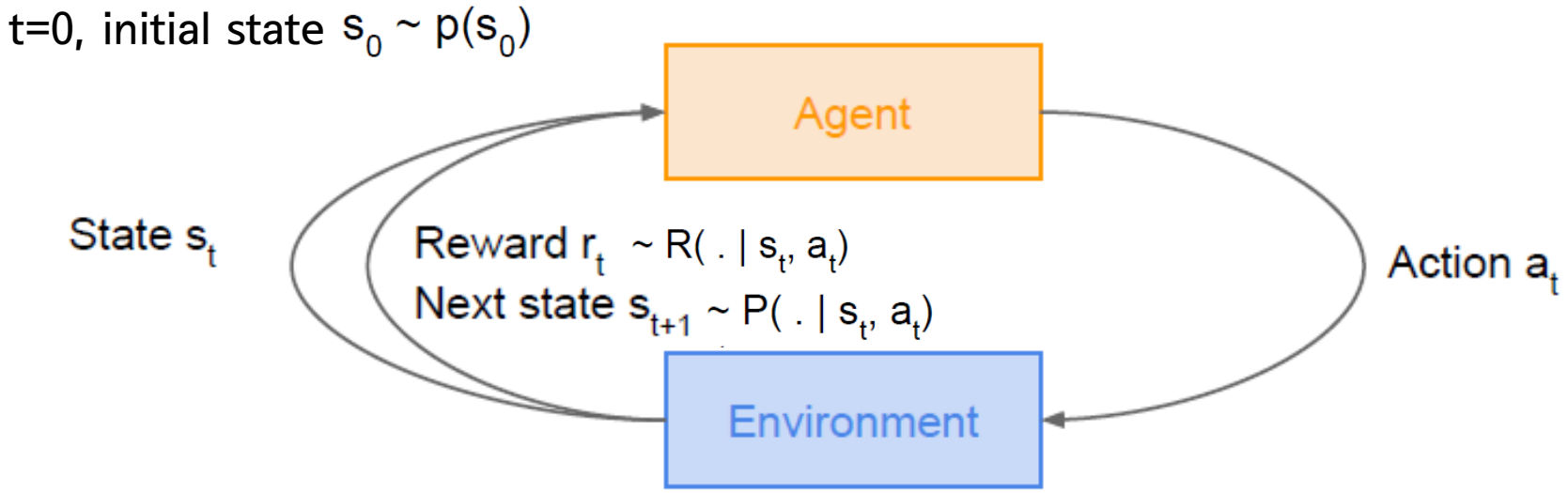
R : (state, action)로부터 오는 reward function

P : transition probability, 특정 state에서 다음 번에 도달할(전이될) state들의 확률 분포

r : discount factor, 미래의 가치를 현재 가치로 환산 (같은 reward라면 빨리 얻는 것이 좋다)

Reinforcement Learning

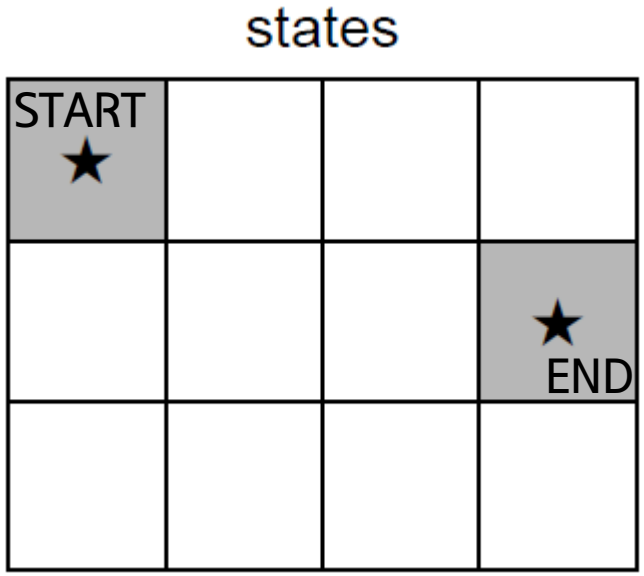
Markov Decision Process



목적 : discount된 누적 reward $\sum_{t \geq 0} \gamma^t r_t$가 최대가 되는 **policy π^*** 구하기

Markov Decision Process

Grid World



state : 격자

agent : 우주선

action : 상/하/좌/우 움직이기

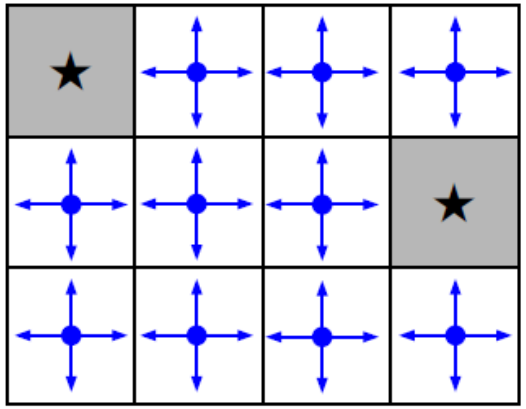
reward : negative reward (-1)

objective : START에서 END state까지 최소한의 action으로 도달

Markov Decision Process

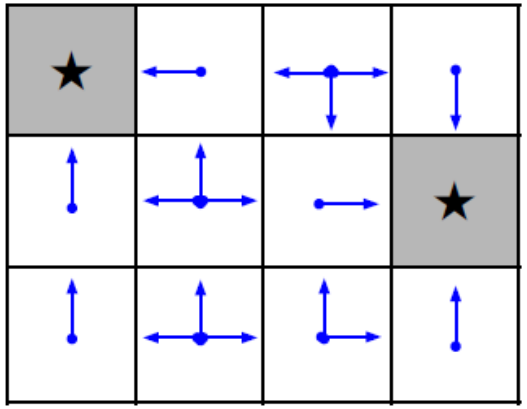
Grid World

Stochastic Policy



Random Policy

모든 방향(action)이
같은 확률을 가지는 경우



Optimal Policy

학습을 통해 얻게 된 policy
종료 state에 가깝게 이동할 수 있는 경
우에 따라 다른 확률을 가짐

Markov Decision Process

Optimal Policy π^*

= reward의 합을 최대로 만드는 policy

Q. initial state, transition probability 등에서 발생하는 randomness는 어떻게 해야하나?

A. **reward의 합의 기댓값을 최대화하기**

Formally: $\pi^* = \arg\max_{\pi} \mathbb{E}\left[\sum_{t \geq 0} \gamma^t r_t | \pi\right]$ with $s_0 \sim p(s_0), a_t \sim \pi(\cdot|s_t), s_{t+1} \sim p(\cdot|s_t, a_t)$

Optimal Policy π^*

Value Function & Q-value Function

각 episode마다 policy에 따라 수행하면, sample trajectories를 얻는다 $(s_0, a_0, r_0, s_1, a_1, r_1, \cdots)$

Value Function 현재 속해있는 state가 얼마나 좋을까?

state s와 policy π가 주어졌을 때, 누적된 보상의 기댓값

$$V^{\pi}(s) = \mathbb{E}\left[\sum_{t \geq 0} \gamma^t r_t | s_0 = s, \pi\right]$$

Q-value Function 현재 state에서 어떤 action을 취해야 좋은가?

state s와 action a, policy π가 주어졌을 때, 받을 수 있는 누적된 보상의 기댓값

$$Q^{\pi}(s, a) = \mathbb{E}\left[\sum_{t \geq 0} \gamma^t r_t | s_0 = s, a_0 = a, \pi\right]$$

Markov Decision Process

Bellman equation

MDP가 주어졌을 때,

최적의 정책 함수(Q-value Function Q*)를 찾는 기본적인 방법

Q-value Function $Q^*(s,a) = \max_{\pi} \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t | s_0 = s, a_0 = a, \pi\right]$

위 수식이 Bellman equation을 따르면,

Optimal Q-value Function $Q^*(s,a) = \mathbb{E}_{s'\sim\mathcal{E}}\left[r + \gamma \max_{a'} Q^*(s',a') | s, a\right]$

Optimal Policy π^*

Bellman equation

$$Q^*(s,a) = \mathbb{E}_{s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q^*(s',a') | s,a \right]$$

Q*(s,a) : 현재 (state, action)

Q*(s',a') : 다음 (state, action)

Q* = 어떤 action을 취했을 때 미래에 받을 reward의 최대치

-> Q*(s',a')에 대한 Q*(s,a)를 안다면,

optimal policy는 $r + \gamma Q^*(s',a')$의 기댓값을 최대화하는 action을 취하는 것

Optimal Policy를 푸는 방법

-> **Value iteration / Policy iteration**

Bellman equation

Value Iteration

Bellman equation을 통해 각 step마다 Q를 최적화

$$Q_{i+1}(s,a) = \mathbb{E}\left[r + \gamma \max_{a'} Q_i(s',a') | s,a\right]$$

i->∞, Q_i는 Q*로 수렴

문제 : Not scalable

-> 반복적인 업데이트를 위해 모든 Q(s,a)를 계산해야 하는데,

기본적으로 전체 state 공간은 매우 크므로 계산하는 것이 불가능

해결 : Q(s,a)를 근사시켜 추정 ex. neural network

Bellman equation

Q-Learning

action-value function 추정을 위해 함수 근사를 이용

$Q(s, a; \theta) \approx Q^*(s, a)$ weight θ : function parameters

-> 해당 함수 추정을 위해서 deep neural network 사용

= deep Q-Learning

$$Q^*(s, a) = \mathbb{E}_{s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q^*(s', a') | s, a \right]$$

앞에서 말했듯이, Q-function은 Bellman Equation을 따르므로

Bellman equation

Q-Learning

$$Q^*(s,a) = \mathbb{E}_{s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q^*(s',a') | s,a \right]$$

Forward Pass

Loss function: $L_i(\theta_i) = \mathbb{E}_{s,a \sim \rho(\cdot)} \left[(y_i - Q(s,a;\theta_i))^2 \right]$

where $y_i = \mathbb{E}_{s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) | s,a \right]$

학습시킨 Q-function 값과

Bellman equation의 차이가 최소가 되도록

Backward Pass

Gradient update (with respect to Q-function parameters θ):

계산한 Loss를 기반으로 θ를 업데이트

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a \sim \rho(\cdot);s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) - Q(s,a;\theta_i)) \nabla_{\theta_i} Q(s,a;\theta_i) \right]$$

Q-Learning

Playing Atari Games

목적 : 게임에서 높은 점수 받기

state : raw pixel inputs

action : 상/하/좌/우 움직이기

reward : 각 step에서 얻는 점수

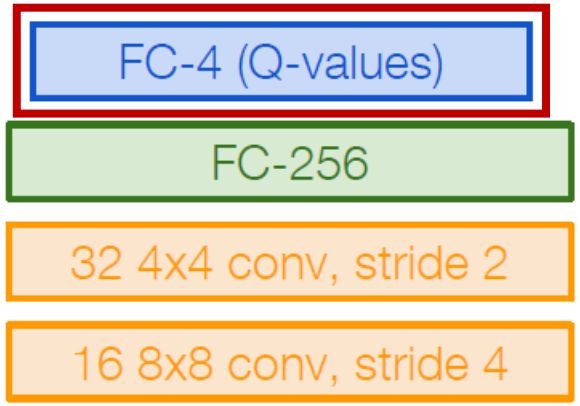
Playing Atari Games

Q-network Architecture

$Q(s, a; \theta)$: weight θ가 있는 neural network

output vector는 각 state가 주어졌을 때, action에 대한 Q-value값

해당 게임에서 action은 (상/하/좌/우) 4가지 이므로 output도 4차원 FC-4



input으로 state를 넣으면,

모든 Q-value를 한번의 forward pass로

계산할 수 있어 효율적이다

Current state s_t: 84x84x4 stack of last 4 frames

(after RGB->grayscale conversion, downsampling, and cropping)

Training Q-network

Loss Function

근사시킨 함수도 Bellman equation을 만족해야 하므로

Q-value가 target value(y_i)와 가까워지도록 반복적인 학습을 시켜야 한다

Forward Loss $L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot)}\left[(y_i - Q(s,a;\theta_i))^2\right]$

Training Q-network

Experience Replay

Q-network를 학습시킬 때 발생하는 문제

하나의 batch에서 연속적인 샘플들로 학습하면,

1) 모든 샘플들이 상관관계를 가져 비효율적

2) 파라미터가 다음 샘플까지 결정하게 되는, bad feedback loops 발생

(어떤 action을 취할지에 대한 policy를 결정한다 = 다음 샘플들도 결정하게 된다)

-> 해결하기 위해 Experience Replay 사용

Q-network

Experience Replay

Replay Memory

(s_t, a_t, r_t, s_{t+1})로 구성된 transition table

-> episode를 play하면서 지속적으로 업데이트

연속적인 샘플들 대신 Replay Memory에서 랜덤하게 샘플링된 미니배치를 사용하여 Q-network를 학습시킴

-> 상관관계 문제 해결

하나의 샘플이 여러 번 뽑혀 multiple weight update이 가능

-> 데이터의 효율 증가

Playing Atari Games

Deep Q-Learning with Experience Replay

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N

Initialize action-value function Q with random weights

for episode $= 1, M$ **do**

  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$

  for $t = 1, T$ **do**

    With probability ϵ select a random action a_t

    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$

    Execute action a_t in emulator and observe reward r_t and image x_{t+1}

    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$

    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$

    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$

    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3

  end for

end for

1) memory capacity N 지정 & Q-network weight 임의로 초기화

2) M개의 episode 플레이

3) 각 episode마다 state를 초기화

4) timestep t만큼 수행

5) 적은 확률로 policy를 따르지 않고 action을 취하고,
나머지는 policy에 따라 action을 취함 (다양한 state 샘플링을 위해)

6) action에 따른 reward와 next state를 얻음

7) transition을 replay memory에 저장

8) experience reaplay를 사용하여 replay memory를 업데이트,
임의의 미니배치를 샘플링하여 Q-learning 학습에 사용

Q-learning

Policy Gradients

Q-network의 문제

Q-network의 매 과정마다 (s,a)를 학습하고 Q-value를 추정하는 과정이 매우 complicated

-> Q-value 추정하는 과정없이 optimal policy를 찾을 수 있을까?

-> Policy Graidents 사용

Q-learning

Policy Gradients

$$\Pi = \{\pi_\theta, \theta \in \mathbb{R}^m\}$$

Policy들은 weight인 θ에 의해 매개변수화 된다

$$J(\theta) = \mathbb{E}\left[\sum_{t\geq 0} \gamma^t r_t | \pi_\theta\right]$$

각 Policy를 정의하면,

J(θ) = 미래에 받을 reward들의 누적 합의 기댓값

Optimal Policy

$$\theta^* = \arg\max_\theta J(\theta)$$

policy는 θ에 의해 정해지므로 Optimal policy를 찾기 위해 θ*를 찾기

= J(θ)을 최대로 하는 policy parameter θ 찾기

-> **Gradient Ascent** on policy parameter를 통해 업데이트하며 θ 찾기

Policy Gradients

Reinforce Algorithm

r(τ) trajectory $\tau = (s_0, a_0, r_0, s_1, \ldots)$의 reward

reward 기댓값 $J(\theta) = \mathbb{E}_{\tau \sim p(\tau;\theta)}[r(\tau)] = \int_\tau r(\tau) p(\tau;\theta) \mathrm{d}\tau$

Gradient Ascent 수행 (미분)

$$\nabla_\theta J(\theta) = \int_\tau r(\tau) \nabla_\theta p(\tau;\theta) \mathrm{d}\tau$$

-> Intractable, p가 θ에 종속되어 있는 상태에서 기댓값 안에 gradient가 존재하면 문제가 될 수 있다

$$\nabla_\theta p(\tau;\theta) = \frac{p(\tau;\theta) \nabla_\theta p(\tau;\theta)}{p(\tau;\theta)} = p(\tau;\theta) \nabla_\theta \log p(\tau;\theta)$$

위 아래에 얘를 곱함

$$\nabla_\theta J(\theta) = \int_\tau \left(r(\tau) \nabla_\theta \log p(\tau;\theta)\right) p(\tau;\theta) \mathrm{d}\tau = \mathbb{E}_{\tau \sim p(\tau;\theta)}[r(\tau) \nabla_\theta \log p(\tau;\theta)]$$

Monto Carlo sampling으로 추정이 가능

Policy Gradients

Reinforce Algorithm

어떤 trajectory에 대한 확률 $p(\tau;\theta) = \prod_{t\geq 0} p(s_{t+1}|s_t, a_t)\pi_\theta(a_t|s_t)$

$$\log p(\tau;\theta) = \sum_{t\geq 0} \log p(s_{t+1}|s_t, a_t) + \log \pi_\theta(a_t|s_t)$$

미분 과정에서 첫 항은 transition probability와 무관하므로, 사라짐

$$\nabla_\theta \log p(\tau;\theta) = \sum_{t\geq 0} \nabla_\theta \log \pi_\theta(a_t|s_t)$$

= gradient를 계산할 때, transition probability는 필요하지 않다

따라서 tranjectory r를 샘플링할 때, J(θ)를 추정할 수 있다

$$\nabla_\theta J(\theta) \approx \sum_{t\geq 0} r(\tau)\nabla_\theta \log \pi_\theta(a_t|s_t)$$

Policy Gradients

Gradient estimator

$$\nabla_\theta J(\theta) \approx \sum_{t \geq 0} r(\tau) \nabla_\theta \log \pi_\theta(a_t|s_t)$$

$r(\tau)$ (경로로부터 얻은 reward)높으면, 해당 actions의 확률을 높임 = 좋은 action

낮으면, 해당 actions의 확률을 낮춤 = 좋지 못한 action

해당 trajectory의 좋고/나쁨은 알 수 있지만, 그 중 어떤 action이 좋고/나빴는지는 알 수 없다

-> 좋은 estimator를 위해 샘플링을 충분히 하는 방법밖에 없음

장점 : gradient를 잘 계산하면 loss function을 낮추고, θ에 대한 local optimum을 구할 수 있다

단점 : credit assignment로 인한 high variance를 가진다

Policy Gradients

Variance reduction

Policy Gradient에서 샘플링을 더 적게 하면서, estimator의 성능을 높일 수 있는 방법

Gradient estimator $\nabla_\theta J(\theta) \approx \sum_{t \geq 0} r(\tau) \nabla_\theta \log \pi_\theta(a_t|s_t)$

First Idea

전체 state가 아닌 해당 state로부터 받을 미래의 reward만을 고려하여

어떤 action을 취할 확률을 키우는 방법

= 어떤 action이 발생시키는 미래의 reward가 얼마나 큰지 고려하겠다는 의미

$$\nabla_\theta J(\theta) \approx \sum_{t \geq 0} \left(\sum_{t' \geq t} r_{t'} \right) \nabla_\theta \log \pi_\theta(a_t|s_t)$$

Second Idea

지연된 reward에 대해 discount factor r을 적용

= 해당 action과 가까운 곳에 더 가중치를 주고, 나중에 수행하는 action에 대해서는 가중치를 낮춤

$$\nabla_\theta J(\theta) \approx \sum_{t \geq 0} \left(\sum_{t' \geq t} \gamma^{t'-t} r_{t'} \right) \nabla_\theta \log \pi_\theta(a_t|s_t)$$

Variance reduction

Baseline

문제 : 경로에서부터 계산한 값을 그대로 사용하는 것

-> 의미가 없는 값 일수도 있기 때문

-> 우리가 받은 reward가 예상보다 좋은지 아닌지를 판단하는 것이 더 중요

해결 : Baseline 사용

baseline : 해당 state에서 우리가 얼만큼의 보상을 원하는지에 대한 기준의 역할

-> 미래에 얻을 reward의 합을 특정 기준이 되는 값(baseline)에서 빼주는 형태로 사용

$$\nabla_\theta J(\theta) \approx \sum_{t \geq 0} \left(\sum_{t' \geq t} \gamma^{t'-t} r_{t'} - b(s_t) \right) \nabla_\theta \log \pi_\theta(a_t|s_t)$$

Variance reduction

Baseline

선택 방법

1) moving average

에피소드를 수행하는 학습 과정에서 지금까지 봤던 모든 경로들에 대해 reward가 어땠는지에 관한 평균을 계산

= 현재 reward가 상대적으로 좋은지 나쁜지를 비교 = Vanilla Reinforce

2) Q-value & value function을 이용

특정 state에서 특정 action을 취했을 때,

Q-value가 얻을 수 있는 미래에 받는 누적 reward 합의 기댓값이 value function보다 더 큰 경우

= 우리가 선택한 action이 다른 action보다 좋은 action이라는 의미

= 반대로 음수이거나 값이 더 작으면 선택한 action이 안좋은 action

-> 얘를 통해 esimator를 얻음 $\nabla_\theta J(\theta) \approx \sum_{t \geq 0} (Q^{\pi_\theta}(s_t, a_t) - V^{\pi_\theta}(s_t)) \nabla_\theta \log \pi_\theta(a_t|s_t)$

Reinforce Algorithm

Actor-Critic Algorithm

Q. 지금까지는 Q-value function과 value function을 구하지 않았다. 그럼 학습을 어떻게 하지?

A. Q-Learning을 통해 학습시킨다

- actor가 action을 선택하면, critic은 action이 얼마나 좋은지 어떻게 조정해야 하는지 알려줌
- policy에 의해 생성된 (s,a)만 학습하면 되므로 critic의 일이 완화
- Experience Replay도 사용 가능
- advantage function을 통해 해당 actions들이 기대보다 좋았는지 판단 $A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s)$

Reinforce Algorithm

Actor-Critic Algorithm

Initialize policy parameters θ, critic parameters ϕ
For iteration=1, 2 … **do**
    Sample m trajectories under the current policy
    $\Delta\theta \leftarrow 0$
    For i=1, …, m **do**
        For t=1, … , T **do**

$$A_t = \sum_{t' \geq t} \gamma^{t'-t} r_t^i - V_\phi(s_t^i)$$

$$\Delta\theta \leftarrow \Delta\theta + A_t \nabla_\theta \log(a_t^i | s_t^i)$$

$$\Delta\phi \leftarrow \sum_i \sum_t \nabla_\phi ||A_t^i||^2$$

$$\theta \leftarrow \alpha\Delta\theta$$

$$\phi \leftarrow \beta\Delta\phi$$

End for

1) 초기화

2) 매 iteration마다 현재 policy를 기반으로 m개의 trajectories 샘플링

3) 각 경로마다 advantage function를 계산
   이를 이용해 gradient estimator를 계산하고 전부 누적

4) ∅ 학습을 위해 value function 계산

5) θ ∅ gradient를 업데이트하며 학습 진행

Reinforce in action

Recurrent Attention Model (RAM)

목표 : Image Classification

모델 : Policy gradient를 이용한 Hard Attention Model

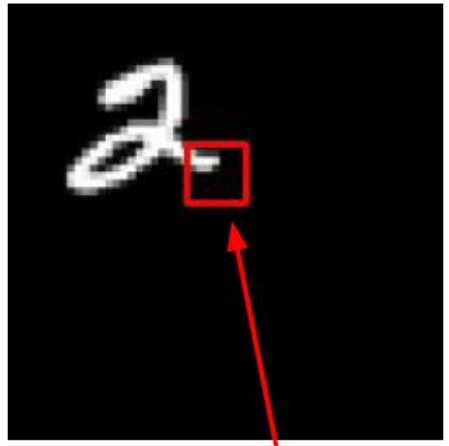
-> 일련의 glimpses(이미지 전체가 아닌 지역적인 부분)만 가지고 이미지를 분류

Q. 왜 Reinforcement Learning으로?

A. glimpses를 뽑아내는 것이 미분 불가능한 연산이기 때문에

state modeling을 위해 RNN을 사용

policy parameter를 이용하여 다음 action을 선택



state : glimpses

action : glimpse의 중심좌표 (x,y)

reward : classification을 제대로 하면 1, 아니면 0

Reinforce in action

Recurrent Attention Model (RAM)

output

: 다음 glimpse의 중심 좌표

time step은 주로 6-8의 고정된 값을 사용