김범수

Policy Gradient

Policy Gradient

By POSCAT

Contents

Introduction

- 지난 시간에는 action-value function, Q(s, a)를 neural network에 학습시켜보았습니다. 이 때 Neural network는 parameter의 집합이라고 할 수 있고, 이 parameter의 집합을 θ라 표현하면, 우리는 실제 action-value function을 θ라는 parameter로 근사하여 표현했다고 할 수 있습니다. 이를 식으로 아래와 같이 표현할 수 있습니다.

$$Q_\theta(s,a) \approx Q^\pi(s,a)$$

- 이 때 policy는 action-value function을 이용해서 바로 도출하였습니다. 우리는 e-greedy 방법을 이용해서 policy를 바로 도출했습니다.
- 이번 시간에는 policy를 parameter로 표현하여 학습하는 방법을 배워볼 것입니다.

$$\pi_\theta(s,a) = \mathbb{P}[a|s,\theta]$$

Introduction

- 장점
- action space의 차원이 높거나, action space가 continuous한 경우에도 효율적으로 학습할 수 있습니다.
- Deterministic policy 뿐만이 아니라 stochastic policy도 학습할 수 있습니다.
- 가위바위보를 deterministic policy로 할 경우(가위, 바위, 보 중 하나만 내는 경우), 상대에게 파악 당해 질 위험이 높습니다. 따라서 세 동작을 1/3 확률로 하는 stochastic policy가 최선의 방법입니다. (N)

- 단점
- global optimum이 아니라 local optimum으로 수렴하는 경향을 가지고 있습니다.

Introduction

- 우리의 목표는 policy를 $\pi_\theta(s,a)$로 표현할 때, 가장 좋은 θ를 찾는 것이 우리 목표입니다.
- 먼저 π_θ가 얼마나 좋은 지를 어떻게 확인할 수 있는 지를 알아봅시다.
- 먼저 stationary distribution $d^{\pi_\theta}(s)$을 정의합니다. stationary distribution은 게임을 무한히 진행했을 때, 현재 state가 s에 있을 확률입니다.

$$d^{\pi_\theta}(s) = \lim_{t\to\infty} \mathbb{P}[S_t = s | s_0, \pi_\theta]$$

- 이 때 π_θ가 얼마나 좋은 지를 나타내는 reward function은 아래와 같이 표현됩니다.

$$J(\theta) = \sum_{s\in S} d^{\pi_\theta}(s) V^{\pi}(s) = \sum_{s\in S} d^{\pi_\theta}(s) \sum_{a\in A} \pi_\theta(a|s) Q^{\pi_\theta}(s,a)$$

- 즉 (내가 state s 에 있을 확률) * (state s에서 내가 받을 수 있는 total reward)의 합으로 정의됩니다.

Policy gradient

- 따라서 policy를 학습시키는 문제는 $J(\theta)$를 최대화하는 문제로 생각할 수 있습니다. 우리는 $J(\theta)$를 최대화하는 θ를 찾을 것입니다.
- 먼저 우리는 $\nabla_\theta J(\theta)$, 즉 reward function을 θ로 미분한 값을 구해야 합니다. 이 값을 구한다면,

$$\Delta\theta = \alpha\nabla_\theta J(\theta)$$

  로 θ를 학습시킬 수 있습니다.

- $\nabla_\theta J(\theta)$는 $\nabla_\theta J(\theta) = \begin{pmatrix} \frac{\partial J(\theta)}{\partial \theta_1} \\ \frac{\partial J(\theta)}{\partial \theta_2} \\ \vdots \\ \frac{\partial J(\theta)}{\partial \theta_n} \end{pmatrix}$ 로 정의할 수 있습니다.

Policy gradient

- $\frac{\partial J(\theta)}{\partial \theta_k}$를 계산하는 방법은 다양한 방법이 있습니다. 그 중 가장 간단한 방법은

$$\frac{\partial J(\theta)}{\partial \theta_k} \approx \frac{J(\theta + \epsilon u_k) - J(\theta)}{\epsilon}$$

로 근사하여 계산하는 방법입니다. 이 때 u_k는 k번째 원소는 1이고, 나머지 원소는 0인 행렬입니다. 즉 θ_k값을 아주 조금 키워서 $J(\theta)$가 커진다면, θ_k을 키우고, 그렇지 않다면, θ_k를 줄이는 방식으로 학습을 진행하는 것입니다.

- 이 방법은 policy gradient의 초기에 사용되었지만, 현재에 와서는 거의 사용되지 않는 방법입니다.

Policy gradient theorem

- $J(\theta) = \sum_{s \in S} d^{\pi_\theta}(s) \sum_{a \in A} \pi_\theta(a|s) Q^{\pi_\theta}(s,a)$로 정의하였습니다. 이 식을 이용해 $\nabla_\theta J(\theta)$를 계산해봅시다.
- $\nabla_\theta J(\theta) = \nabla_\theta \sum_{s \in S} d^{\pi_\theta}(s) \sum_{a \in A} \pi_\theta(a|s) Q^{\pi_\theta}(s,a)$ 로 표현됩니다. 이 때 policy gradient theorem에 의하면,

$$\nabla_\theta \sum_{s \in S} d^{\pi_\theta}(s) \sum_{a \in A} \pi_\theta(a|s) Q^{\pi_\theta}(s,a) = \sum_{s \in S} d^{\pi_\theta}(s) \sum_{a \in A} \nabla_\theta \pi_\theta(a|s) Q^{\pi_\theta}(s,a)$$

로 표현할 수 있습니다. 이 식을 정리해봅시다.

Policy gradient theorem

$$\nabla_\theta J(\theta) = \sum_{s \in S} d^{\pi_\theta}(s) \sum_{a \in A} \nabla_\theta \pi_\theta(a|s) Q^{\pi_\theta}(s,a)$$

$$= \sum_{s \in S} d^{\pi_\theta}(s) \sum_{a \in A} \pi_\theta(a|s) \frac{\nabla_\theta \pi_\theta(a|s)}{\pi_\theta(a|s)} Q^{\pi_\theta}(s,a)$$

$$= \mathbb{E}_\pi \left[\frac{\nabla_\theta \pi_\theta(a|s)}{\pi_\theta(a|s)} Q^{\pi_\theta}(s,a) \right] = \mathbb{E}_\pi [\nabla_\theta \ln \pi_\theta(a|s) Q^{\pi_\theta}(s,a)]$$

- 이 때, $\mathbb{E}_\pi$는 policy π_θ를 따를 때의 state distribution과 action distribution상태에서의 평균값을 의미합니다. 즉, 실제로 게임을 진행하는 상태에서의 평균값을 의미합니다. 따라서 이 식을 바탕으로 다양한 policy gradient algorithm이 탄생하게 됩니다.

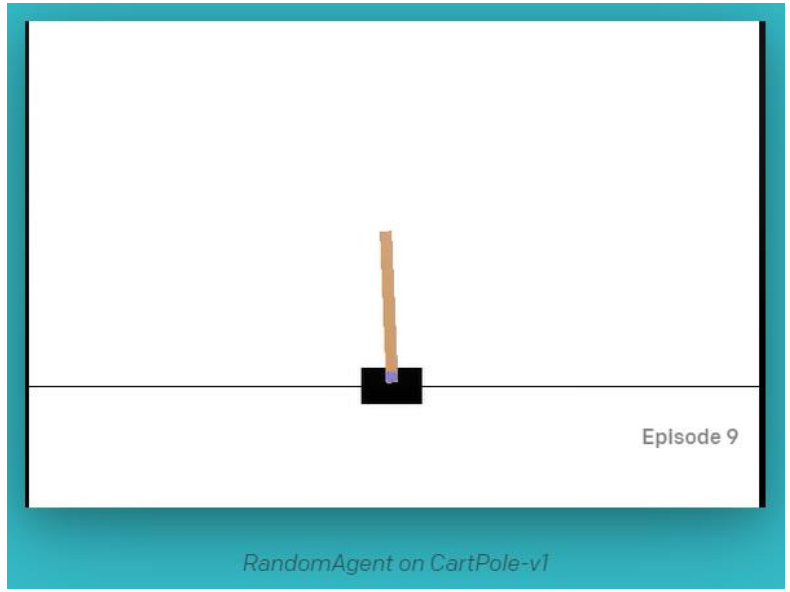
REINFORCE

- 이번에는 가장 기초적인 policy gradient theorem을 배워봅니다.
- $\nabla_\theta J(\theta) = \mathbb{E}_\pi\left[Q^{\pi_\theta}(s,a)\nabla_\theta \ln \pi_\theta(a|s)\right] = \mathbb{E}_\pi[G_t \nabla_\theta \ln \pi_\theta(a|s)]\ (\because Q^{\pi_\theta}(s,a) = \mathbb{E}_\pi[\mathrm{G_t}|\mathrm{S_t} = \mathrm{s}, \mathrm{A_t} = \mathrm{a}])$
- 위 식을 이용해서 학습을 진행할 수 있습니다. 즉, 실제로 게임을 진행하고, 그 결과를 바탕으로 학습하는 기법입니다. 이 기법을 REINFORCE라고 합니다.
- REINFORCE의 절차는 매우 직관적으로 이루어집니다.
- 1. θ를 랜덤한 값으로 초기화한다.
  2. policy π_θ를 바탕으로 게임을 진행한다. $(s_1, a_1, r_2, s_2, \dots, s_T)$
  3. 모든 t = 1, 2⋯ T-1에 대해
     3.1. G_t를 계산한다.
     3.2. 이를 바탕으로, θ를 update한다.

$$\theta \leftarrow \theta + \alpha G_t \nabla_\theta \ln \pi_\theta(a_t|s_t)$$

연습문제

- 이번 연습문제에서는 gym의 cartpole 문제를 풀어봅니다.
- cartpole 문제는 카트를 좌우로 움직여서 막대가 기울어지지 않도록 하는 문제입니다.
- 막대가 12도 이상 기울어지거나, 카트가 화면 밖으로 나가면 종료됩니다.
- 게임이 끝날 때까지 1의 보상을 주며, 최근 100 에피소드에서 평균 보상이 195점 이상일 경우, 성공으로 간주합니다.
- state는 총 4개의 실수로 구성되어 있습니다.
  카트의 위치와 속도, 막대기의 각도와 각속도
- action은 카트에 왼쪽 또는 오른쪽에 힘을 가하는 2가지 행동이 가능합니다.
  0 = 왼쪽으로 카트를 미는 action
  1 = 오른쪽으로 카트를 미는 action



연습문제

- 우리가 설계할 Neural Network는 state로 4개의 실수를 입력 받아 action 0을 취할 확률과 1을 취할 확률, 2개의 실수를 반환해야 합니다. 당연하지만, 0을 취할 확률과 1을 취할 확률의 합은 1이 되어야 합니다.

- 먼저 Neural Network에서 얻은 두 실수를 a, b라 하면, 0을 취할 확률을 $\frac{a}{a+b}$1을 취할 확률을 $\frac{b}{a+b}$로 계산하여 취급하는 방법이 가능합니다. 하지만 a, b가 음수이면 안된다는 단점이 존재하며,a 또는 b가 0이라면 0 또는 1이라는 action을 전혀 취하지 않아 exploration에 문제가 발생합니다. 또한 a = b = 0이 될 경우 계산이 불가능합니다.

- 이러한 단점을 보완하기 위해, softmax 함수를 이용해서 확률을 계산합니다. softmax 함수란, n개의 action에 대해 반환한 값이 $(x_1, x_2, \dots, x_n)$일 때, $softmax(x_i) = \frac{\exp(x_i)}{\sum \exp(x_j)}$로 정의됩니다.

- pytorch에서는 torch.nn.Softmax 함수로 구현되어 있습니다.

연습문제

- action-value를 학습시키면, policy가 급격하게 변하게 됩니다. action-value가 조금만 변하더라도, policy는 a를 선택한다에서 b를 선택한다로 바뀔 수 있으므로, 학습 결과로 인해 policy가 급격히 변할 수 있다는 단점이 있었습니다. 하지만, REINFORCE처럼 policy를 학습시키는 알고리즘은 policy 자체를 천천히 변화시키므로, 이러한 단점이 없어집니다.
- REINFORCE 알고리즘은 학습에 total reward를 사용하기 때문에 variance가 매우 크다는 단점이 있습니다. 따라서 실제로 REINFORCE 알고리즘을 구현해보면, variance로 인해 잘 수렴하지 않는 것을 볼 수 있습니다. 따라서 다음 시간에는 variance를 줄이기 위해 action-value와 policy를 모두 학습시키는 알고리즘인 actor-critic에 대해 배워보겠습니다.