

Fundamental of Reinforcement Learning

Leewoongwon

Published
with GitBook



차례

Preface	1.1
Cover	1.2
1. Introduction	1.3
What is Reinforcement Learning	1.3.1
History	1.3.2
Example	1.3.3
2. Markov Decision Process	1.4
MDP	1.4.1
Value Function	1.4.2
3. Bellman Equation	1.5
Bellman Expectation Equation	1.5.1
Bellman Optimality Equation	1.5.2
4. Dynamic Programming	1.6
Policy Iteration	1.6.1
Value Iteration	1.6.2
Example	1.6.3
5. Monte-Carlo Learning	1.7
Monte-Carlo Prediction	1.7.1
Monte-Carlo Control	1.7.2
6. Temporal Difference Learning	1.8
TD Prediction	1.8.1
TD Control	1.8.2
Eligibility Traces	1.8.3
7. Off-Policy Control	1.9
Importance Sampling	1.9.1
Q Learning	1.9.2
8. Value Function Approximation	1.10
Value function Approximation	1.10.1
Stochastic Gradient Descent	1.10.2
Learning with Function Approximator	1.10.3

9. DQN(Deep Q - Networks)	1.11
Neural Network	1.11.1
Deep Q Networks	1.11.2
10. Policy Gradient	1.12
Policy Gradient	1.12.1
Finite Difference Policy Gradient	1.12.2
Monte-Carlo Policy Gradient : REINFORCE	1.12.3
Actor-Critic Policy Gradient	1.12.4

Fundamental of Reinforcement Learning

Author : Woong won, Lee

- RLCode(Reinforcement Learning Code)
- 강남다이나믹스(nonlinear, underactuated robotics), 모두의연구소
- OpenRL(Reinforcement Learning), 모두의연구소
- DCULab(Drone Control and Utiliaztion Laboratory), 모두의연구소
- Mechanical Engineering, Yonsei University

How to start

2016년 초부터 모두의 연구소의 자율주행 드론 연구실 DCULab의 연구실장을 맡아서 드론을 연구를 해오고 있었습니다. 그러던 중에 "드론의 제어에 사용되는 PID 계수를 자동으로 맞춰주는 방법이 없을까?"라는 생각이 들었고 찾다보니 Reinforcement learning을 접하게 되었습니다. 마침 모두의 연구소에서 강화학습 스터디가 시작되었고 그 때부터 David Silver교수님의 강의를 듣기 시작했습니다

Reinforcement Learning

환경과의 상호작용을 통한 학습 방법 중에서 computational하게 접근하는 것이 machine learning입니다. 강화학습은 machine learning의 범주 안에 있는 학습 방법 중의 하나입니다. 그 방법 중에서 아이가 걷는 것을 배우는 것처럼 어떻게 행동할 줄 모르지만 환경과 상호작용하면서 걷는 법을 알아가는 것과 같은 학습 방법을 강화학습이라고 합니다.

Purpose of book

강화학습 스터디를 하면서 "혼자 공부하기는 정말 어렵겠다"라는 생각을 많이 했습니다. 또한 강화학습과 관련된 전문 자료들을 꽤 있지만 입문하는 사람들을 위한 쉬운 강의나 자료가 잘 없고 한국어로는 거의 전무한 상황입니다. 따라서 전공자가 아니면 쉽게 처음 시작을 하지 못하는 어려움이

있습니다. 따라서 저 스스로는 공부했던 내용들을 정리하면서도 처음 시작하는 분들에게 도움이 되고자 이 책을 쓰게 되었습니다.

Lectures about RL

강화학습 스터디는 Deepmind의 David Silver교수님의 강의를 듣는 것으로 진행했었는데 강화학습을 처음 시작할 때 듣기에 좋은 강의입니다. 교재는 Sutton교수님의 *Introduction to Reinforcement Learning*이라는 책을 사용였고 책과 강의의 링크는 다음과 같습니다.

<https://www.youtube.com/watch?v=2pWv7GOvuf0>

<http://www.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html>

Silver 교수님 강의 말고도 Udacity강의도 있습니다. 저희 스터디도 처음에는 유다시티 강의로 시작했지만 대부분 많은 사람들이 Sutton교수님을 강화학습의 아버지로 생각하고 있기 때문에 그 책을 바탕으로 한 David Silver 교수님의 강의를 듣기로 정했습니다. 이 강의는 David Silver가 UCL의 교수일 때 Computer Science 학생들을 대상으로 한 강의입니다. 개인적으로는 Silver교수님 강의가 학문적인 바탕을 탄탄히 깔아주는 것 같습니다. 다만 정식 녹화와 녹음이 아니고 노트북을 사용했기 때문에 자료가 잘 안 보이고 소리가 잘 안 들리는 단점이 있습니다. Udacity 강화학습 강의 링크는 다음과 같습니다.

<https://classroom.udacity.com/courses/ud600/lessons/4676850295/concepts/467334481109>

23

UC Berkeley의 intro to AI 강의에서도 강화학습에 관련된 내용이 나옵니다.

http://ai.berkeley.edu/lecture_slides.html

Think

이 영화는 2015년 "Chappie"라는 영화입니다. 이 로봇은 완전한 인공지능으로 가치관도 학습해나갑니다. 자신이 놓인 환경에 따라 추구하는 것이 달라지는 것입니다. 영화에서는 Chappie(로봇이름)가 갱과 함께 생활하고 사람들로부터 학대를 받으면서 그릇된 가치관을 배워나가는데 그것을 통해서 미래의 인공지능에 대한 경각심을 일으키는 것 같습니다. 현재의 인공지능 기술은 발전해나가는 과정에 있지만 인공지능을 옳바르게 사용해야한다는 생각을 가지고 연구해야 할 것 같습니다. 그러한 관점을 가지고 강화학습을 공부해나간다면 더 재미있을 것 같습니다.



<http://1u88jj3r4db2x4txp44yqfj1.wpengine.netdna-cdn.com/wp-content/uploads/2015/02/Chappie.jpg>

Writing a guide book for Reinforcement Learning

정말 감사하게도 많은 분들이 이 책을 봐주셨고 그로인해 강화학습을 공부하고자 하는 분들에게 도움을 드릴 수 있음을 알게 되었습니다. 따라서 현재 출판사와 계약해서 코드와 함께 강화학습 기본을 공부할 수 있는 책을 집필하고 있습니다. 책에 들어갈 예제 코드들은 깃헙에 공개되어 있습니다. 자유롭게 테스트 해보시고 코멘트 주세요!

<https://github.com/ricode/reinforcement-learning>

How to contact

오타나 틀린 내용, 잘 이해가 안 가는 부분, 추가 설명이 필요한 부분, 또는 궁금한 것들을 discussion에 올려주셔도 되고 제 메일로 보내주셔도 좋습니다!

Email : dnddnjs11@naver.com

Fundamental of Reinforcement Learning

First edition

Woongwon Lee

2016

Chapter 1 : Introduction

강화학습이 무엇인지에 대해 그 정의를 살펴보고 어디서 유래했는지를 알아보는 챕터입니다. 또한 앞으로 공부할 강화학습에 대해 유명한 예제를 살펴봄으로서 전체적인 insight를 얻을 수 있습니다.

1.1 What is Reinforcement Learning?

1.2 History of Reinforcement Learning

1.3 Example of Reinforcement Learning

1.1 What is Reinforcement Learning

강화학습이라는 이름만 봐서는 무엇인가를 강화하는 인공지능인 것 같은데 정확히 무엇을 말하는지 모르는 분들이 계실 겁니다.

https://ko.wikipedia.org/wiki/%EA%B0%95%ED%99%94_%ED%95%99%EC%8A%B5

위키피디아에서 강화 학습을 다음과 같이 설명하고 있습니다.

강화 학습(Reinforcement learning)은 기계학습이 다루는 문제 중에서 다음과 같이 기술 되는 것을 다룬다. 어떤 환경을 탐색하는 에이전트가 현재의 상태를 인식하여 어떤 행동을 취한다. 그러면 그 에이전트는 환경으로부터 포상을 얻게 된다. 포상은 양수와 음수 둘 다 가능하다. 강화 학습의 알고리즘은 그 에이전트가 앞으로 누적될 포상을 최대화하는 일련의 행동으로 정의되는 정책을 찾는 방법이다.

학습이라는 것에 대해 생각할 때 우리가 제일 처음 생각해낼 수 있는 것은 환경과의 상호작용을 통한 학습입니다. 그러한 개념은 거의 모든 인공지능의 이론의 기본 바탕이 되어있습니다. 그러한 학습을 computational하게 접근하는 것을 machine learning이라고 합니다.

강화학습은 machine learning의 범주 안에 있는 학습 방법 중의 하나입니다. 그 방법 중에서 아이가 걷는 것을 배우는 것처럼 어떻게 행동할 줄 모르지만, 환경과 상호작용하면서 걷는 법을 알아가는 것과 같은 학습 방법을 강화학습이라고 합니다. 제가 공부하면서도 느낀 것은 강화학습으로 학습하는 방법은 사람이 평소에 어떤 것을 배워나가고 일상 속에서 행동하는 방법과 상당히 유사하다는 것입니다.

사람이 처음 자전거를 배울 때를 생각해봅시다. 어릴 때 부모님이 자전거를 가르쳐주실 때

"이 자전거는 ~한 시스템이고 각 부분은 ~한 dynamics를 가지고 있어서 만약 10도 정도 기울었을 때는 핸들을 반대로 ~한 각속도로 틀어줘야 한다. 근데 너의 몸무게가 얼마나 되지?"

이렇게 가르쳐주시지 않았을 겁니다. 아마도 아무것도 모르고 자전거에 올라서 타보면서 어떻게 자전거를 타는 지 배우는데 이렇게 하면 넘어지고 이렇게 하면 똑바로 간다는 것을 학습했을 것입니다.

dynamics를 모르고 학습하는 것이기 때문에 다음 영상과 같이 단순히 핸들 방향만 바꾼 건데 전혀 자전거를 탈 수 없게 됩니다.

<https://youtu.be/EqXL7xC-4Y4>



강화학습도 이와 마찬가지로 agent가 아무것도 모르고 환경 속으로 들어가서 경험을 통해 학습하는 것입니다. 그냥 간단히 생각하기에 "컴퓨터니까 다 계산해서 게임을 하거나 로봇을 움직이거나 하면 안 돼?"라고 생각할 수도 있겠지만 그렇지 않습니다.

경우의 수가 적은 게임의 경우에는 모든 것을 계산할 수 있겠지만, 바둑 같은 경우나 혹은 실재의 세상에서 모든 것을 계산하는 것은 불가능한 일입니다. 하지만 모든 사람은 프로바둑 기사가 될 수는 없어도 바둑을 두다 보면 어느정도는 바둑을 둘 수 있게 됩니다. 인공지능이 사람의 그러한 학습 방법을 모티브 삼아서 학습하는 것입니다.

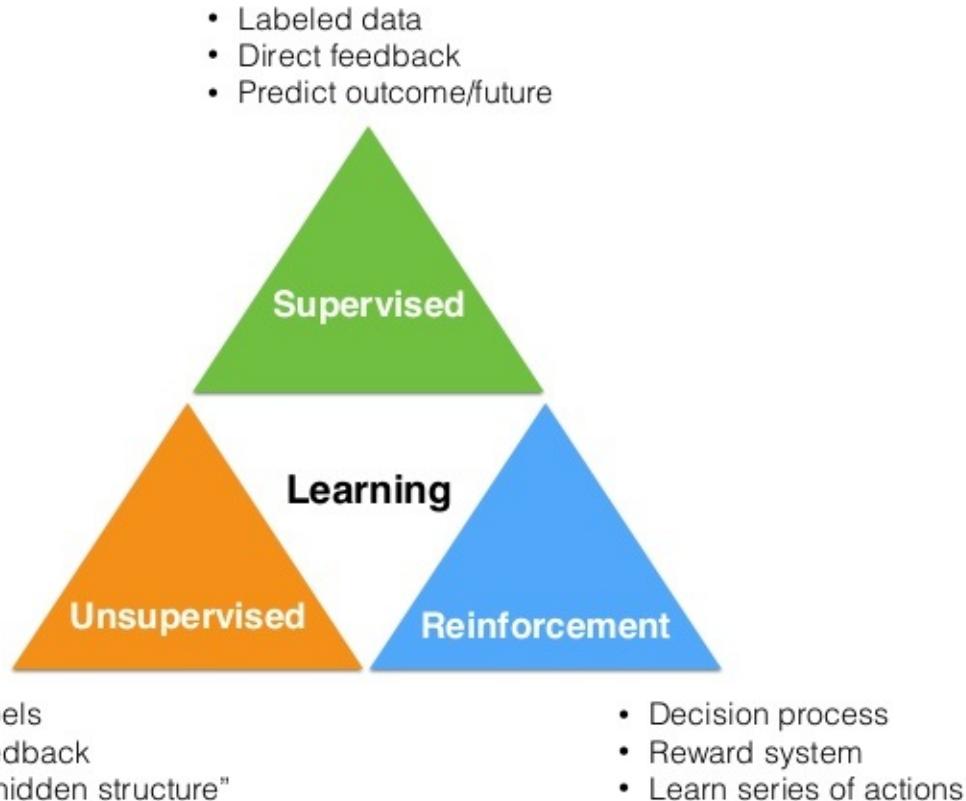
Sutton교수님의 책에서 다음과 같은 문장이 있습니다.

Reinforcement learning is defined not by characterizing learning methods, but by characterizing a learning problem.

특이하게도 강화학습은 학습하는 "방식"으로 정의되는 것이 아니고 강화학습 "문제"인가로 정의되어 집니다. 그렇다면 어떤 문제가 강화학습 문제일까요?

강화학습의 정의에 대해서 살펴보기 전에 강화학습을 포함하는 더 넓은 분야인 machine learning의 범주에 대해서 간단히 살펴볼 필요가 있습니다. machine learning은 크게 세 가지로 나눠질 수 있습니다.

1. Supervised Learning
2. Unsupervised Learning
3. Reinforcement Learning



- Supervised Learning은 지도학습으로서 "정답"을 알 수 있어서 바로바로 피드백을 받으면서 학습하는 것을 말합니다.
- Unsupervised Learning은 비지도 학습으로서 정답이 없는 "분류"와 같은 문제를 푸는 것을 말합니다.
- Reinforcement Learning은 강화학습으로서 정답은 모르지만, 자신이 한 행동에 대한 "보상"을 알 수 있어서 그로부터 학습하는 것을 말합니다. 뒤에서 말하겠지만, 강화학습은 MDP로 표현되어지는 문제를 푸는 것을 말합니다.

강화학습 문제의 예는 다음과 같습니다.

1. Fly stunt manoeuvres in a helicopter
2. Defeat the world champion at Backgammon
3. Manage an investment portfolio
4. Control a power station
5. Make a humanoid robot walk
6. Play many different Atari games better than humans

헬리콥터를 멋지게 날게 할 수도 있고 특정 게임의 세계 챔피언을 이길 수도 있습니다(체스, Backgammon, 바둑!). 그뿐만 아니라 로봇이 걷게 할 수도 있고 사업에서도 매니지먼트를 할 수도 있습니다. 아직은 발전해야 할 것이 참 많지만, 가능성은 무궁무진한 흥미로운 분야인 것 같습니다.

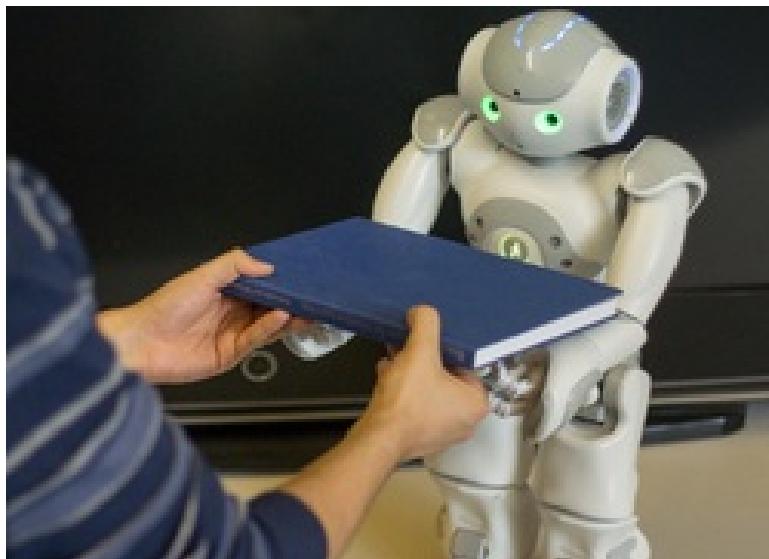
이러한 강화학습의 가장 중요한 두 가지 특징은 아래와 같습니다.

1. Trial and Error

2. Delayed Reward

- 첫 번째는 환경과의 상호작용으로 학습하는 것과 깊은 관련이 있습니다. Trial and Error, 즉, 해보지 않고 예측하고 움직이는 것이 아니고 해보면서 자신을 조정해나가는 것입니다.

아래와 같이 좋은 행동을 했을 경우에 좋은 반응이 환경으로부터 오게 됩니다. 이 반응을 "Reward"라고 하는데 이것에 관해서는 Chapter 2에서 다루도록 하겠습니다. 사람이 그러하듯 이 인공지능 또한 상을 많이 받기 위해 노력할 것입니다. "어떻게 상을 더 많이 받을 것이냐?"는 강화학습의 핵심 쟁점중의 하나입니다.



- 두 번째는 강화학습이 다루는 문제에 "시간"이라는 개념이 포함되어 있다는 것과 관련이 있습니다. 강화학습은 시간의 순서가 있는 문제를 풀기 때문에 지금 한 행동으로 인한 환경의 반응이 늦어질 수가 있는데(혹은 다른 행동과 합해져서 더 좋은 환경의 반응을 받아낼 수도 있습니다), 이럴 경우에 환경이 반응할 때까지 여러 가지 다른 행동들을 시간의 순서대로 했기 때문에 어떤 행동이 좋은 행동이었는지 판단하기 어려운 점이 있습니다. 이 점은 강화학습의 중요한 문제로서 계속 머리속에 넣어둘 필요가 있습니다.

다시 위의 위키피디아 강화학습의 정의로 돌아가 보겠습니다.

강화 학습(Reinforcement Learning)은 기계학습이 다루는 문제 중에서 다음과 같이 기술되는 것을 다룬다. 어떤 환경을 탐색하는 에이전트가 현재의 상태를 인식하여 어떤 행동을 취한다. 그러면 그 에이전트는 환경으로부터 포상을 얻게 된다. 포상은 양수와 음수 둘 다 가능하다. 강화 학습의 알고리즘은 그 에이전트가 앞으로 누적될 포상을 최대화 하는 일련의 행동으로 정의되는 정책을 찾는 방법이다.

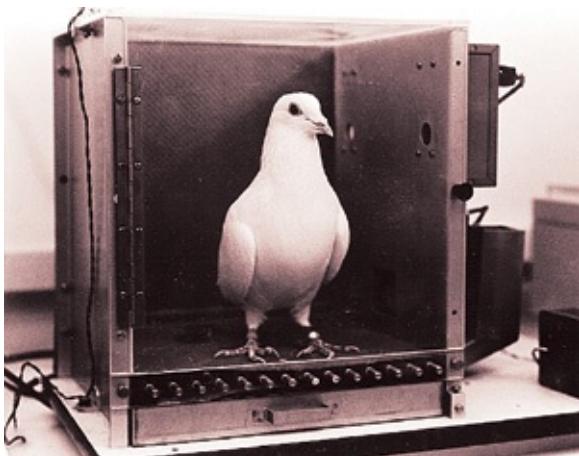
에이전트(Agent), 상태(State), 행동(Action), 포상(Reward), 정책(Policy) 이러한 단어가 윗글에서 나오는데 강화학습 문제를 정의하고 풀어나가는데 필수적인 이 개념들은 다음 글에서 다루도록 하겠습니다.

History

강화학습의 시작은 크게 두 가지로 볼 수 있습니다.

- Trial and error
- Optimal control

Trial and error는 1.1에서도 강화학습의 중요한 특징이라고 언급했는데 바로 그 부분은 동물의 행동에 관한 심리학 연구에서 출발하였습니다. 심리학에서는 "강화"라는 개념은 상당히 보편적으로 알려져 있는 개념으로서 동물이나 인간이 행동 결과에 따라 행동을 변화시키고 발전시킨다는 이론입니다. 스키너라는 심리학자의 "스키너 상자 실험"이라는 것이 있습니다. 상자 안에 비둘기를 집어넣고 다음과 같이 실험을 하였습니다. <http://blog.naver.com/PostView.nhn?blogId=babomaum&logNo=120014011465>



스키너는 굽긴 비둘기를 방음이 잘 된 상자(스키너 상자)속에 넣는데, 이 상자의 한쪽 벽에 원판이 장치되어 있다. 비둘기가 원판을 쪼면 먹이통에 이'T는 먹이가 떨어지게 설치를 해두었다. 이 스키너 상자에 비둘기를 넣으면 비둘기는 새로운 환경을 탐색하면서 여러 가지 반응을 나타낸다. 처음에 비둘기는 여러 가지 행동을 할 것이고, 그러다 우연히 원판을 쪼개 될 것이다. 그러면 먹이가 자동적으로 주어진다. 따라서 이런 과정이 몇 차례 계속되면 비둘기는 드디어 여러 반응을 생략하고 즉각적으로 원판을 쪼는 반응을 계속할 것이다.

[출처] [스키너] 스키너의 강화이론|작성자 하루하루

이와 같이 자신이 한 행동에 따른 보상으로 인해 더 좋은 보상을 받는 행동을 하도록 학습이 되는 것을 볼 수 있는데 이러한 학습방법이 강화학습의 모티프가 된 것입니다.

Optimal control이라는 말은 1950년대부터 사용되기 시작했는데 어떠한 비용함수의 비용을 최소화하도록 컨트롤러를 디자인하는 것을 말합니다. 뒤에서 배울 Bellman이 "Bellman equation"이라는 방정식을 만들어서 이를 통해서 optimal control 문제를 풀었고 그러한 방법은 Dynamic

Programming이라고 불립니다. 또한 Bellman은 MDP(Markov Decision Process)라는 수학적 모델을 만들어서 강화학습의 기초를 놓게 됩니다. 저도 Dynamic Programming과 MDP에 대해서 더 알고 싶어서 아래의 강의와 책을 참고할 예정입니다.

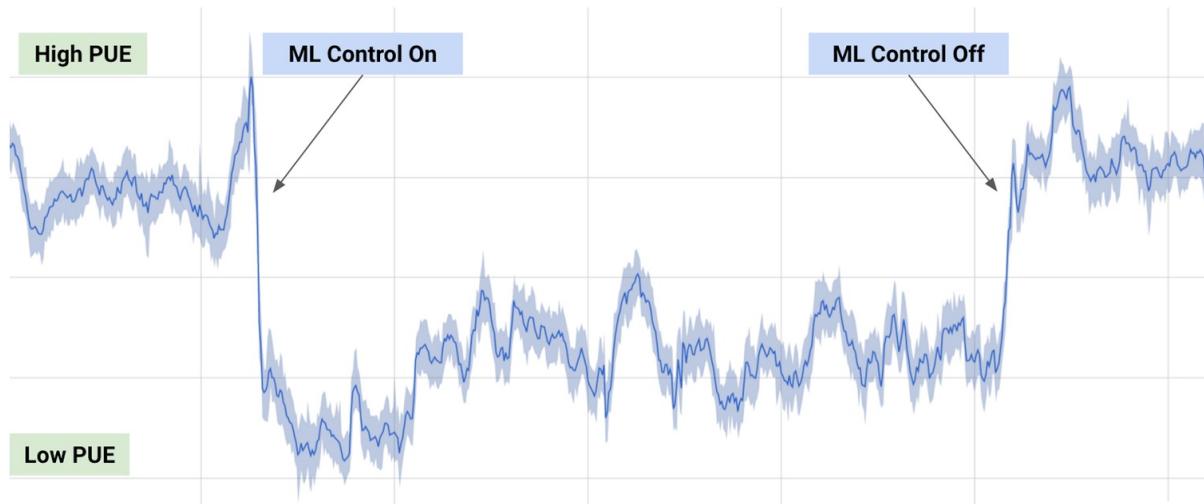
- Dynamic Programming 강의 : [Approximate Dynamic Programming Lectures by D. P. Bertsekas](#)
- MDP 책 : [Markov Decision Processes: Discrete Stochastic Dynamic Programming](#)

이러한 두 가지가 합쳐지면서 "강화학습"이 탄생하게 되었고 그 뒤로 Temporal difference Learning, Q Learning으로 발전해오다가 최근에 딥러닝과의 조합으로 엄청난 성과를 내고 있습니다. 다들 잘 아는 알파고는 강화학습 + 딥러닝으로서 Policy gradient with Monte-carlo Tree Search 알고리즘을 사용하였습니다.

로봇에 강화학습을 적용시켜 학습시키는 연구 또한 부지런히 수행되어 왔는데 앞으로 로봇의 발전에 큰 역할을 할 것으로 기대가 됩니다. 또한 이번에 구글이 강화학습을 사용해 전기료를 줄였다는 기사처럼 정말 많은 분야에 적용되어서 성능을 개선할 수 있다는 기대를 많이 받고 있습니다.

아래는 딥마인드가 구글 데이터센터의 전기 사용량을 줄인 기사입니다.

- [DeepMind AI Reduces Google Data Centre Cooling Bill by 40%](#)



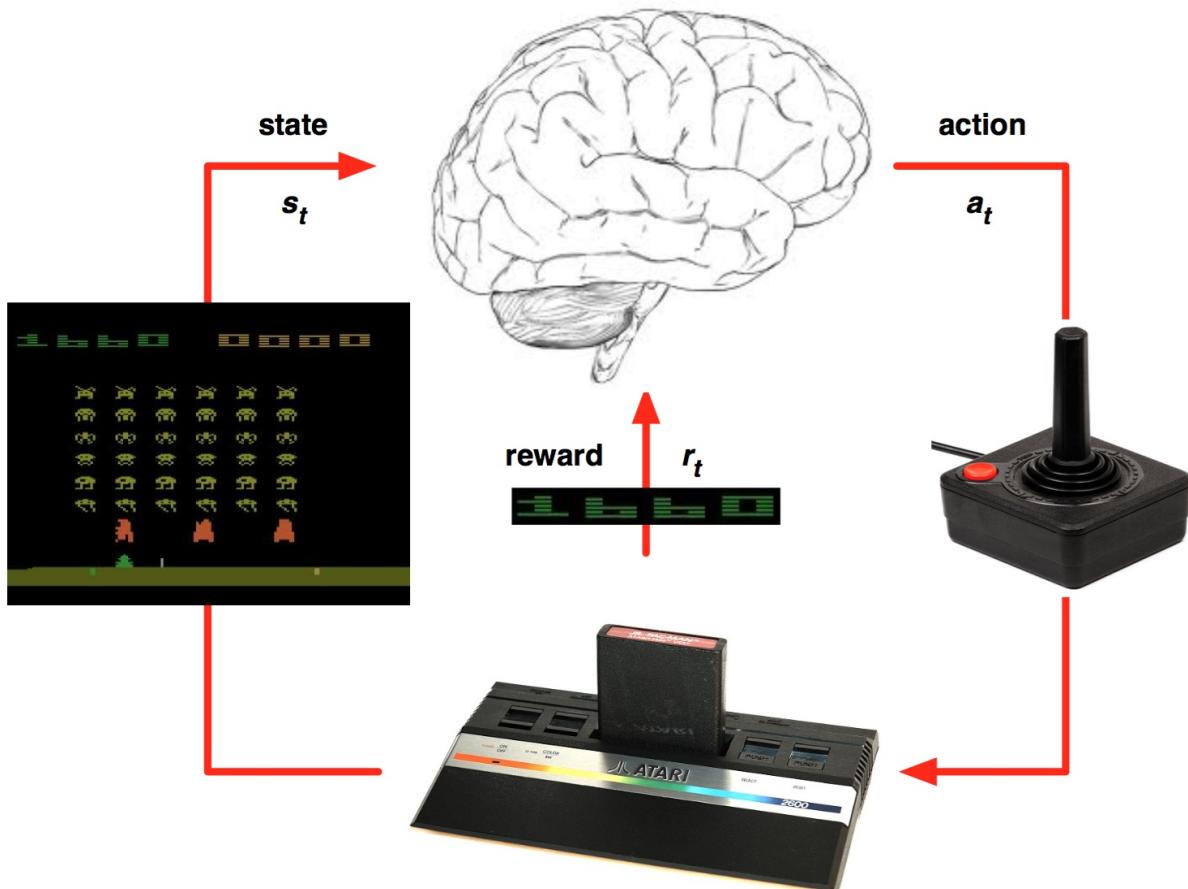
Example

어떠한 학문을 공부하는데 있어서 기초부터 차근차근 배워나가는 것도 좋지만 그렇게 공부하다 보면 지금 무엇을 하고 있는지 모를 때가 많습니다. 따라서 중간중간에 실습이나 어떠한 예제를 들어서 생각을 해보거나 때로는 단계를 깅충 뛰어서 진짜 문제에 어떻게 적용되는지 맛만 보는 것 또한 도움이 되기도 합니다.

Playing atari with deep reinforcement learning이라는 논문이 있습니다. 강화학습 + 딥러닝으로 atari라는 고전 게임을 학습시킨 것입니다.

Atari게임 중에서도 Breakout이라는 벽돌깨기 게임을 컴퓨터가 플레이하는 것이 상당히 인상적이었습니다. 아래는 그 동영상입니다. 잘 보시면 단순히 벽돌만 깨는 것이 아니고 한 쪽을 터널로 뚫어서 여러개의 벽돌을 한꺼번에 깨버리는 전략을 쓰는 것도 볼 수 있습니다.

<https://youtu.be/iqXKQf2BOSE>

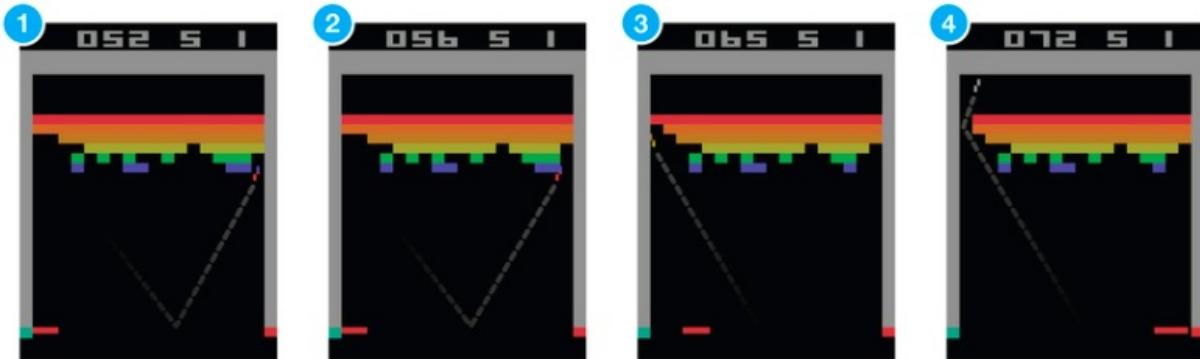


위 그림이 어떻게 학습을 하는지에 대해서 간략하게 보여주는 그림입니다. 강화학습의 "학습" 대상은 agent입니다. 위에서 나오는 state, reward, action은 다음 챕터에서 자세히 다루도록 하겠습니다. 사람의 뇌에 해당하는 이 agent는 처음에 랜덤하게 움직이게 됩니다. 랜덤하게 움직이다가 우

연히 공을 치게 되고 게임 점수가 올라가게 되면 이 agent는 "아! 이 행동을 하면 점수가 올라가는구나!!"라는 식으로 판단을 해서 점수를 올라가게 했던 행동을 더 하려고 자신을 학습시킵니다.

하지만 agent는 단순히 즉각적인 점수만을 높이려는 것이 아니고 시간이 지날수록 한 episode 동안 받는 점수를 최대화시키려고 합니다. 따라서 agent는 일련의 연속된 행동 즉 "정책(Policy)"이 필요하게 됩니다. 사실은 높은 점수라는 것은 어떠한 한 행동의 결과라기보다는 각 상황에 맞는 적절한 행동들의 좋은 조합이라고 할 수 있습니다. 따라서 높은 점수를 얻는 것이 목표인 agent는 이러한 자신의 일련의 행동들의 정책 혹은 전략을 좀 더 높은 점수를 받는 쪽으로 변화시킵니다.

어떤 행동이 좋은 행동이고 어떤 조합이 좋은 조합인지는 Trial and error로서 이것저것 시도해보면서 차츰차츰 알아가게 됩니다. 이렇게 경험을 통해 학습이 되는 방식도 여러 가지가 있습니다. 상황에 맞는 적절한 방법을 선택하는 것 또한 중요한데 그건 실제로 학습을 시켜보면서 경험을 하면서 얻어지는 것 같습니다. 그렇게 이것 저것 시도해보다가 어느 순간 터널이 뚫려서 한 번에 엄청난 점수를 얻는 사건이 발생합니다. 이제 agent는 알게 되는 것입니다. "터널을 뚫어서 점수를 얻는 것 이 더 많은 점수를 얻을 수 있다"라는 것을 말입니다.



<https://www.nervanasys.com/demystifying-deep-reinforcement-learning/>

이 논문의 주목할 점은 다음과 같습니다.

1. input data로 raw pixel를 받아온 점 --> CNN과의 연결
2. 같은 agent로 여러 개의 게임에 적용되어서 학습이 된다는 점
3. Deep neural network를 function approximator로 사용
4. Experience Replay
5. Target networks

최근에 Deep Reinforcement Learning이 대세가 되었는데 위에서처럼 Reinforcement Learning에 Deep Learning을 합한 것을 말합니다. 처음에 강화학습을 배우기 시작할 때는 gridworld같은 작은 환경에서 이것저것 해보겠지만 위와 같이 실제 게임을 학습하거나 할 때에는 데이터의 숫자가 너무 많기 때문에 제대로 학습이 되지 않습니다. 따라서 그 데이터의 숫자를 다 일일이 저장해서 그 것으로 행동을 하는 것이 아니고 함수의 형태로 만들어서 정확하지 않더라도 효율적으로 학습을 할 수 있게 하는 방법을 사용합니다. 많은 공학의 문제에서 사용하는 "approximation"입니다. 이 Approximation 툴 중에서 Deep Neural Network는 최근에 엄청난 성능을 자랑하는 툴입니다.

이렇게 유명한 Breakout 게임 학습에 대해서 간단히 설명하였습니다. 중간에 여러 가지 모르는 개념들이 많이 나왔을 텐데 앞으로 자세히 하나하나 다루도록 하겠습니다. 이 예제를 통해 강화학습이 게임에서 어떻게 학습시키는지 적절히 이해를 하면 됩니다.

Chapter 2 : Markov Decision Process

문제를 잘 정의하면 문제의 절반은 풀었다고도 할 수 있습니다. 그만큼 문제를 정의하는 것이 중요 한데 강화학습은 MDP로 표현되는 문제를 푸는 알고리즘의 집합입니다. MDP는 무엇이고 그 MDP와 강화학습이 어떻게 연결되어지는 것일까요?

1. MDP

2. Value Function

Markov Decision Process

David Silver 강의에서는 MDP를 배우기 전에 Markov하다는 말의 정의와 Markov Chain, Markov Reward Process를 배웁니다. Markov는 1800년대의 러시아 수학자의 이름입니다. 이 분의 이름이 하나의 형용사가 되었는데 그 의미는 다음과 같습니다.

Consider how a general environment might respond at time $t + 1$ to the action taken at time t . In the most general, causal case this response may depend on everything that has happened earlier. In this case the dynamics can be defined only by specifying the complete probability distribution:

$$\Pr\{R_{t+1} = r, S_{t+1} = s' \mid S_0, A_0, R_1, \dots, S_{t-1}, A_{t-1}, R_t, S_t, A_t\}, \quad (3.4)$$

for all r , s' , and all possible values of the past events: $S_0, A_0, R_1, \dots, S_{t-1}, A_{t-1}, R_t, S_t, A_t$. If the state signal has the *Markov property*, on the other hand, then the environment's response at $t + 1$ depends only on the state and action representations at t , in which case the environment's dynamics can be defined by specifying only

$$\Pr\{R_{t+1} = r, S_{t+1} = s' \mid S_t, A_t\}, \quad (3.5)$$

for all r , s' , S_t , and A_t . In other words, a state signal has the Markov property, and is a Markov state, if and only if (3.5) is equal to (3.4) for all s' , r , and histories, $S_0, A_0, R_1, \dots, S_{t-1}, A_{t-1}, R_t, S_t, A_t$. In this case, the environment and task as a whole are also said to have the Markov property.

뒤에서 state와 value에 대해서 설명하겠습니다.

위의 첫 식처럼 처음 어떠한 상태로부터 시작해서 현재 상태까지 올 확률이 바로 전 상태에서 현재 상태까지 올 확률과 같을 때, 두 번째 식처럼 표현이 될 수 있고 state는 Markov하다고 일컬어질 수 있습니다.

스타크래프트같은 게임이라고 생각하면 게임 중간 어떤 상황은 이전의 모든 상황들에 영향을 받아서 지금의 상황이 된 것이기 때문에 사실은 지금 상황에 이전 상황에 대한 정보들이 모두 담겨있다고 가정할수도 있습니다. 강화학습이 기본적으로 MDP로 정의되는 문제를 풀기때문에 state는 Markov라고 가정하고 접근합니다. 하지만 절대적인 것은 아니며 Non-Markovian MDP도 있으며 그러한 문제를 풀기위한 강화학습들도 있지만 상대적으로 연구가 덜 되었으며 처음에 접하기에는 적합하지 않습니다. 강화학습에서는 value라는 어떠한 가치가 현재의 state의 함수로 표현되고 이 state가 Markov하다고 가정됩니다.

다음은 UC Berkeley의 intro to AI 강의의 slide에서 가져온 그림입니다.

http://ai.berkeley.edu/lecture_slides.html

Markov Decision Processes



Instructors: Dan Klein and Pieter Abbeel
University of California, Berkeley

위 그림에서 로봇이 세상을 바라보고 이해하는 방식이 MDP가 됩니다. MDP란 Markov Decision Process의 약자로서 state, action, state transition probability matrix, reward, discount factor로 이루어져 있습니다. 로봇이 있는 위치가 state, 앞뒤좌우로 이동하는 것이 action, 저 멀리 보이는 빛나는 보석이 reward입니다. 한 마디로 문제의 정의입니다. 이제 이 로봇은 보석을 얻기 위해 어떻게 해야할지를 학습하게 될 것 입니다. 그 전에 MDP에 대해서 더 살펴볼 필요가 있습니다.

다시 제가 들었던 Silver교수님의 강의에서 말하는 MDP의 정의를 살펴보겠습니다. 밑의 그림은 <http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html>

에 있는 2장 자료에서 가져왔습니다.

A Markov decision process (MDP) is a Markov reward process with decisions. It is an *environment* in which all states are Markov.

Definition

A *Markov Decision Process* is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$

- \mathcal{S} is a finite set of states
- \mathcal{A} is a finite set of actions
- \mathcal{P} is a state transition probability matrix,
 $\mathcal{P}_{ss'}^{\textcolor{red}{a}} = \mathbb{P}[S_{t+1} = s' | S_t = s, A_t = \textcolor{red}{a}]$
- \mathcal{R} is a reward function, $\mathcal{R}_s^{\textcolor{red}{a}} = \mathbb{E}[R_{t+1} | S_t = s, A_t = \textcolor{red}{a}]$
- γ is a discount factor $\gamma \in [0, 1]$.

State

간단히 설명을 하자면 state는 agent가 인식하는 자신의 상태입니다. 사람으로 치자면 눈이라는 관측도구를 통해서 "나는 방에 있어"라고 인식하는 과정에서 "방"이 state가 됩니다.

이 이외에도 state는 생각보다 많은 것들이 될 수 있는데 달리는 차 같은 경우에는 "차는 Jeep이고 사람은 4명 탔으며 현재 100km/h로 달리고 있다"라는 것이 state가 될 수 있습니다. OpenAI에도 있는 atari game 같은 경우에는 게임화면 자체, 즉 pixel이 agent가 인식하는 state가 됩니다. 또 Cartpole에서는 cart의 x위치와 속도, pole의 각도와 각속도가 state가 됩니다. 즉, 문제는 정의하기 나름입니다. 실제로 어떠한 문제를 강화학습으로 풀 수도 있고 다른 machine learning 기법으로 풀 수도 있기 때문에 강화학습을 적용시키기 전에 왜 강화학습을 써야하고 다른 머신러닝 기법에 비해서 나은 점이 무엇인가를 따져보고 사용해야할 것 같습니다. 강화학습은 "시간"이라는 개념이 있는 문제를 푸는 인공지능 기법입니다. 이는 결국 강화학습의 목표가 Policy(일련의 행동들)된다는 의미를 함포합니다.

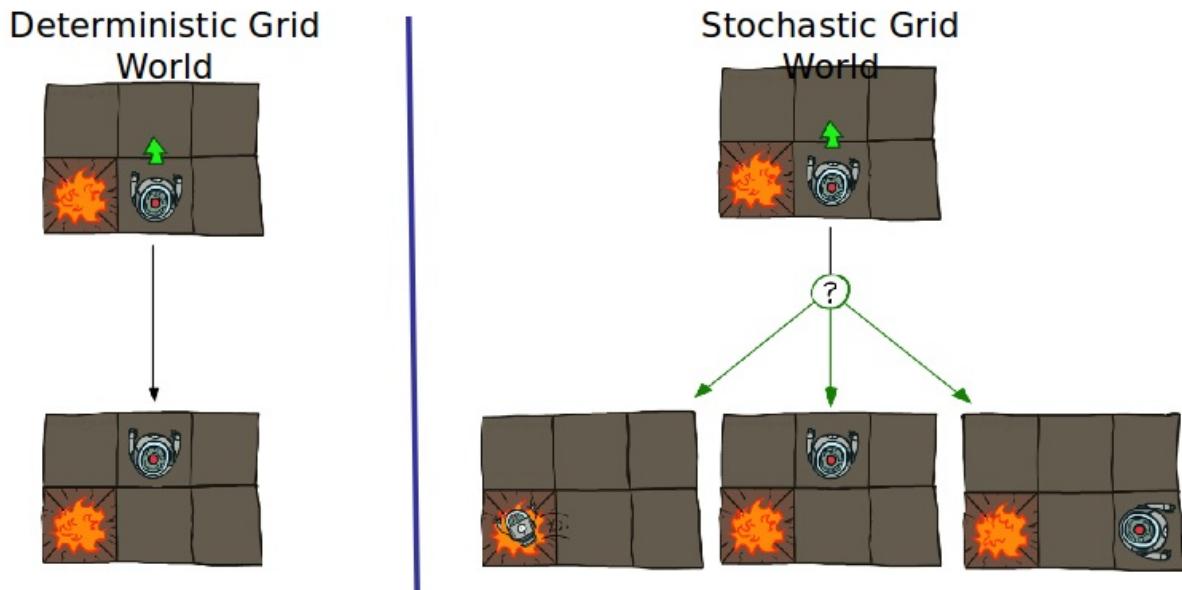
Action

Agent의 역할은 무엇일까요? environment에서 특정 state에 갔을 때 action을 지시하는 것입니다. robot이 왼쪽으로 갈지, 오른쪽으로 갈지를 결정해주는 역할입니다. 그래서 사람의 뇌라고 생각하면 이해가 쉽습니다. "오른쪽으로 간다", "왼쪽으로 간다"라는 것이 action이 되고 agent가 그 action을 취했을 경우에 실제로 오른쪽이나 왼쪽으로 움직이게 됩니다. 또한 agent는 action을 취함으로서 자신의 state를 변화시킬 수 있습니다. 로봇에서는 흔히 Controller라 부르는 개념입니다.

State transition probability matrix

robot이 움직인다고 생각해봅시다. robot이 왼쪽으로 움직이면 위치가 변하듯이, action을 취하면 environment상의 agent의 state가 변하는 데 그것 또한 environment가 agent에게 알려줍니다. 정확히 말하면 agent가 observe하는 것입니다. 대신에 어떠한 외부요인에 의해 (ex 바람이 분다던지) robot이 왼쪽으로 가려했지만 오른쪽으로 가는 경우가 발생할 수 있습니다. 다음 그림을 참고하면 개념이 좀 더 잘 와닿으실 겁니다. 로봇은 앞으로 간다고 갔지만 왼쪽으로 가서 불에 빠질 수도 있고 오른쪽으로 갈 수도 있다는 것입니다. 그 확률을 표현하는 것이 "state transition

probability matrix"입니다. 이렇게 어떠한 action을 취했을 경우 state가 deterministic하게 딱 정해지는 것이 아니고 확률적으로 정해지게 되는데 일종의 noise라고 생각하셔도 될 것 같습니다.



정의는 다음과 같습니다. s 라는 state에서 a 라는 행동을 취할 때 s' 에 도착할 확률을 이야기합니다.

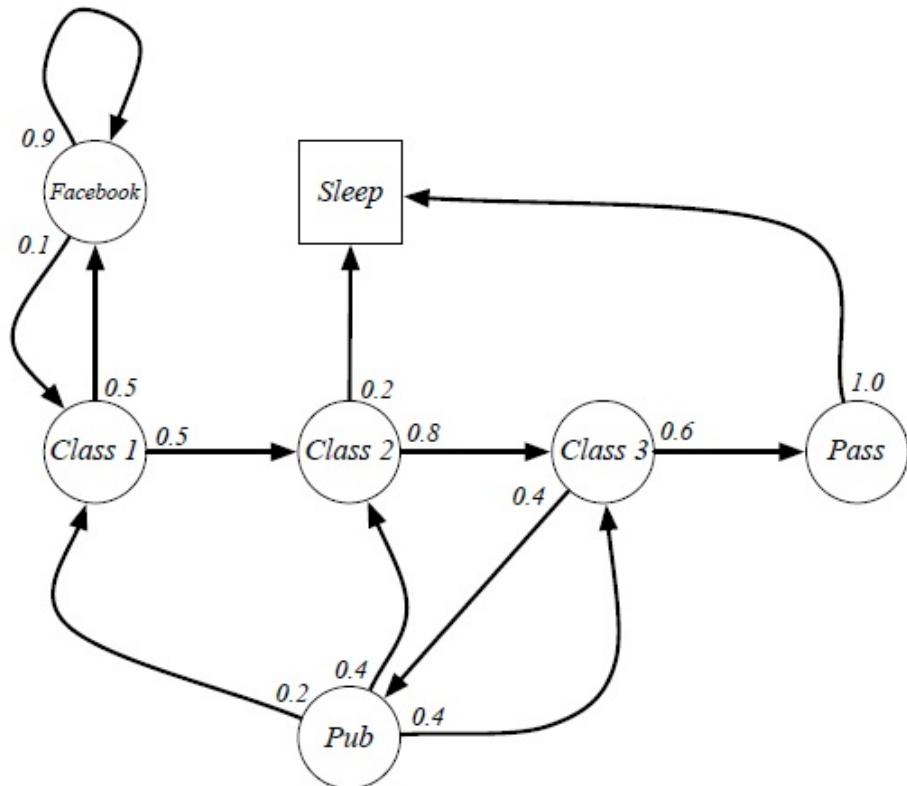
$$\mathcal{P}_{ss'}^a = \mathbb{P}[S_{t+1} = s' | S_t = s, A_t = a]$$

Markov Chain

MDP에서 action과 reward가 없다고 가정해봅시다. 그렇다면 state와 state끼리의 transition matrix를 생각해볼 수 있을 것 입니다. Silver교수님은 Markov Chain에 대해서 다음과 같이 설명 하셨습니다. 학생들의 상태를 state로 잡고 각 state끼리의 transition probability를 정의할 수 있습니다. 이것만으로서는 무엇인가를 학습시킬 수 없지만 MDP를 배우기전에 배우는 기본 개념으로서 유용한 것 같고 후에 policy gradient에서 다루는 stationary distribution을 이해하는데 도움이 될 것 같습니다. 이 Markov Chain에서는 무한대로 시간이 흐르면 모두 Sleep으로 수렴할 것이고 더이상 변화가 없기 때문에 stationary distribution이라고 말합니다. 현재는 어떤 state에서 state

로 가는 확률이 표시되어 있지만 MDP에서는 action을 할 확률과 action을 해서 어떤 state로 갈 확률이 주어지게 됩니다.

Example: Student Markov Chain



Reward

agent가 action을 취하면 그에 따른 reward를 "environment"가 agent에게 알려줍니다. 그 reward는 atari game에서는 "score", 바둑의 경우에는 승패(알파고가 학습하는 방법), trajectory control의 경우에는 "의도한 궤도에 얼마나 가깝게 움직였나"가 됩니다. 정의는 다음과 같습니다. s 라는 state에 있을 때 a 라는 action을 취했을 때 얻을 수 있는 reward입니다. 강화학습에서는 정답이나 사전에 환경에 대한 지식이 없이 이 reward를 통해서 agent가 학습하게 됩니다.

$$\mathcal{R}_s^a = \mathbb{E}[R_{t+1} \mid S_t = s, A_t = a]$$

이 reward를 immediate reward라고 하는데 agent는 단순히 즉각적으로 나오는 reward만 보는 것 이 아니라 이후로 얻는 reward들까지 고려합니다.

Discount Factor

reward의 정의에 따라 각 state에서 어떠한 action을 취하면 reward를 받게 되는데 이때 단순히 받았던 reward들을 더하면 다음과 같은 문제가 발생합니다.

- 어떠한 agent는 각 time-step마다 0.1씩 reward를 받고 다른 agent는 1씩 받았을 경우에 시간이 무한대로 흘러간다면 0.1씩 계속 더해도 무한대이고 1씩 계속 더해도 무한대입니다. 수학에서 무한대는 크기 비교를 할 수 없습니다.
- 다음 두 가지 경우를 구분 할 수가 없습니다. agent가 episode를 시작하자마자 1 받았을 경우와 끝날 때 1을 받았을 경우를 둘 다 전체 reward를 1을 받았기 때문에 두 상황중에 어떤 경우가 더 나은 건지를 판단할 수 없습니다.

따라서 **discount factor**라는 개념이 등장하게 됩니다. 사람의 입장에서 생각해보면 당장 지금 배고픈 것을 채우는 것이 내일 배고픈 것을 채우는 것보다 중요하다 생각하고 행동하는 것처럼 discount factor를 통해서 시간에 따라서 reward의 가치가 달라지는 것을 표현하는 것입니다. discount factor는 0에서 1 사이의 값입니다. 다음 그림을 보면 이해가 쉽습니다.

Discounting

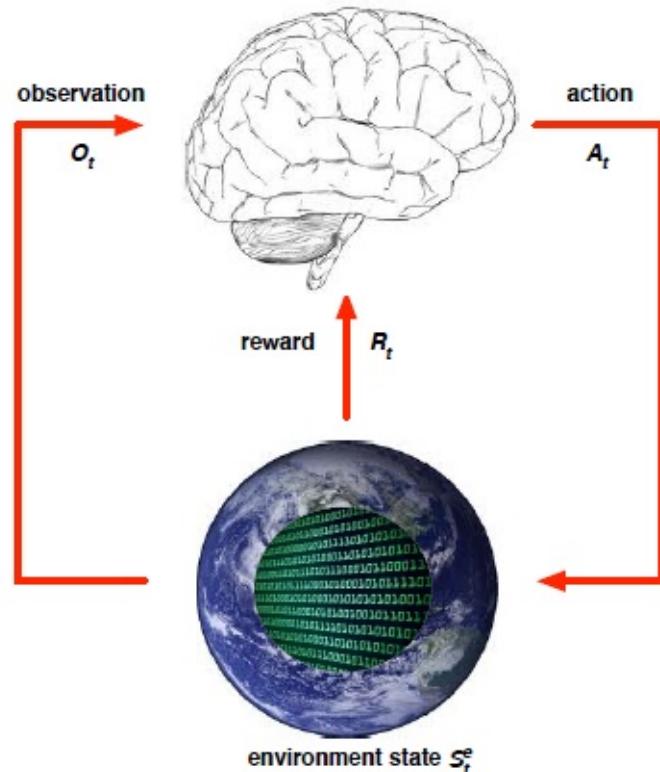
- It's reasonable to maximize the sum of rewards
- It's also reasonable to prefer rewards now to rewards later
- One solution: values of rewards decay exponentially



제가 이해하기로는 discount factor가 0이면 상당히 근시안적인 것이고 discount factor가 1이면 상당히 미래지향적인 것이라서 사실은 사람이 어떤 행동을 결정할 때 미래를 생각하며 결정하긴 하지만 모든 미래에 일어날 일을 다 고려하지는 않습니다. 따라서 dicount factor는 보통 0에서 1사이의 값을 사용합니다.

Agent-Environment Interface

이렇듯 agent는 action을 취하고 state를 옮기고 reward를 받고 하면서 environment와 상호작용



을 하는데 그 그림은 다음과 같습니다.

agent가 observation을 통해서 자신의 state를 알게되면 그 state에 맞는 action을 취하게 됩니다. 학습을 하지 않은 초기에는 random action을 취합니다. 그러면 environment가 agent에게 reward와 다음 state를 알려주게 됩니다. 시뮬레이터나 게임이 environment가 될 수도 있고 실제 세상이 environment가 될 수도 있습니다.

Policy

뜻 그대로 풀이하자면 "정책"입니다. 위에서 말했듯이 agent는 어떤 state에 도착하면 action을 결정하는 데 어떤 state에서 어떤 action을 할지를 policy라고 합니다. 결국에 강화학습의 목적은 optimal policy (accumulative reward = return 을 최대화하는 policy)를 찾는 것입니다(이것이 잘못된 관념이라고 얘기하는데 실제로 Policy Gradient는 suboptimal에 빠질수 있지만 강화학습입니다). policy의 정의는 다음과 같습니다. state s에서 action a를 할 확률을 이야기합니다.

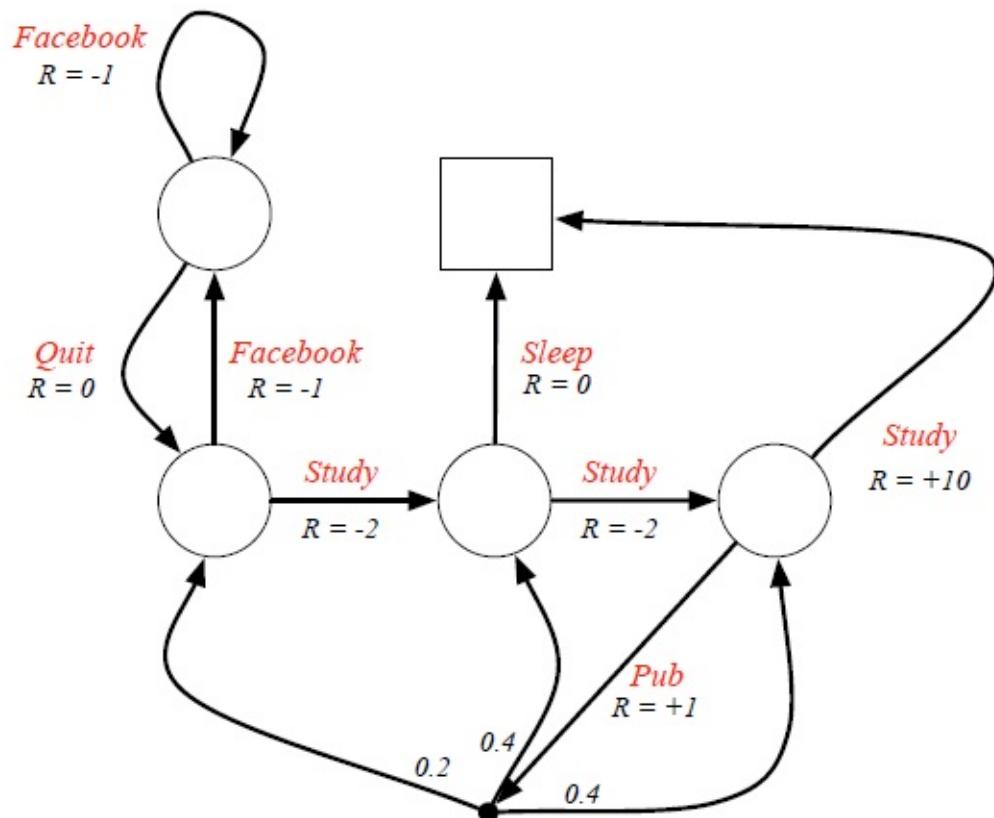
Definition

A **policy π** is a distribution over actions given states,

$$\pi(a|s) = \mathbb{P}[A_t = a | S_t = s]$$

MDP Graph

Markow Decision Process는 다음과 같이 그래프로 나타낼 수 있습니다. 이번에도 student의 수업을 듣는 것을 예로 들어서 Silver 교수님이 설명했습니다.



이와 같이

MDP의 graph는 state 사이의 transition 대신에 action을 통한 state의 transition과 reward로서 표현되게 됩니다.

Value Function

State-value function

agent가 state 1에 있다고 가정해봅시다. 거기서부터 쭉 action을 취해가면서 이동할테고 그에 따라서 reward를 받는 것들을 기억할 것입니다. 끝이 있는 episode라고 가정했을 때 episode가 끝났을 때 state 1에서부터 받았던 reward를 다 더할 수 있을 겁니다. 밑은 Silver 교수님 강의 chapter 2에서의 return의 정의입니다.

Definition

The *return* G_t is the total discounted reward from time-step t .

$$G_t = R_{t+1} + \gamma R_{t+2} + \dots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

Return에 대한 식은 위와 같고 이 return의 expectation이 state-value function입니다. expectation은 기대값으로서 평균은 expectation안에 포함되는 개념입니다. 주사위를 던 질 경우 얼마나 나올지에 대한 기대값으로도 expectation을 사용합니다. 아래와 같이 표현됩니다.

The *value function* $v(s)$ gives the long-term value of state s

Definition

The state value function $v(s)$ of an MRP is the expected return starting from state s

$$v(s) = \mathbb{E}[G_t | S_t = s]$$

즉, 어떤 상태 S 의 가치입니다. 그렇다면 어떻게 value function을 구할 수 있을까요? agent가 다음으로 갈 수 있는 state들의 가치를 보고서 높은 가치의 state로 이동하게 되는데 따라서 다음으로 갈 수 있는 state들의 value function이 상당히 중요하고 어떻게 효율적이고 정확한 value function을 구할 지가 중요한 문제가 됩니다.

value function을 구하는 하나의 방법을 예를 들어보겠습니다. Value function은 return(실제 경험을 통해서 받은 reward의 discounted amount)의 expectation이기 때문에 마치 주사위를 던져 보듯이 던져 보면서 expectation 값을 구할 수 있습니다. 계속 그 state로부터 시작되거나 그 state를 지나가는 episode를 try해보면서 얻어진 reward들에 대한 data들로 그 value function에 점점 다

가갈 수 있는데 사실 주사위도 무한번 던져야 1/3이라는 true expectation값을 가지듯이 value function 또한 무한히 try를 해봐야 true value function을 찾을 수 있을 것 입니다. 그렇다면 어떻게 적당한 선을 찾아서 "이 정도는 true 값이라고 하자"라고 결정을 내릴까요? 이 또한 생각해봐야 할 점인 것 같습니다.

Sample **returns** for Student MRP:

Starting from $S_1 = C1$ with $\gamma = \frac{1}{2}$

$$G_1 = R_2 + \gamma R_3 + \dots + \gamma^{T-2} R_T$$

C1 C2 C3 Pass Sleep	$v_1 = -2 - 2 * \frac{1}{2} - 2 * \frac{1}{4} + 10 * \frac{1}{8}$	=	-2.25
C1 FB FB C1 C2 Sleep	$v_1 = -2 - 1 * \frac{1}{2} - 1 * \frac{1}{4} - 2 * \frac{1}{8} - 2 * \frac{1}{16}$	=	-3.125
C1 C2 C3 Pub C2 C3 Pass Sleep	$v_1 = -2 - 2 * \frac{1}{2} - 2 * \frac{1}{4} + 1 * \frac{1}{8} - 2 * \frac{1}{16} \dots$	=	-3.41
C1 FB FB C1 C2 C3 Pub C1 ...	$v_1 = -2 - 1 * \frac{1}{2} - 1 * \frac{1}{4} - 2 * \frac{1}{8} - 2 * \frac{1}{16} \dots$	=	-3.20
FB FB FB C1 C2 C3 Pub C2 Sleep			

전에 배웠던 Policy에 대해서도 생각해봅시다. 위 정의와 예시에는 policy에 대한 고려가 되지 않았는데 만약에 agent가 어떤 행동 정책(어떤 경향성을 띠는 행동 방식)을 한다고 가정을 해봅시다. Random으로 움직일 때와 비교를 해보면 각 state들은 완전 다른 경험을하게 되었을 것 입니다. 음악을 좋아하는 아이와 과학을 좋아하는 아이가 완전 다른 인생을 사는 것과 같습니다.

그렇다면 value function을 계산하는데 있어서(아이가 인생의 가치를 어떻게 판단하나) agent의 일련의 행동방식은 꼭 고려를 해주어야 합니다. 사실은 위와 같이 계산을 할 경우 저절로 그 경험 안에 policy가 녹아들어가게 됩니다. 또한 각 policy마다 value function이 달라질 수 있으므로 그 value function을 최대로 하는 policy를 찾을 수 있게 되는 것입니다. policy에 대한 value function은 다음과 같습니다.

Definition

The **state-value function** $v_\pi(s)$ of an MDP is **the expected return starting from state s , and then following policy π**

$$v_\pi(s) = \mathbb{E}_\pi [G_t | S_t = s]$$

대부분 강화학습의 알고리즘에서는 value function을 얼마나 잘 계산하느냐가 중요한 역할을 하고 있습니다. "잘"이라는 의미에는 bias되지 않고 variance가 낮으며 true값에 가까우며 효율적으로 빠른 시간안에 수렴하는 것을 의미합니다.

Action-value function

MDP에서 action이란 무엇인지에 대해서 정의하였습니다. action이란 어떤 state에서 할 수 있는 행동들을 말하는데 보통 모든 state에서 가능한 행동은 모두 같습니다. 위에서 정의를 내린 value function에 대해서 생각을 해보면 사실 그 state의 가치라는 것은 그 state에서 어떤 action을 했는지에 따라 달라지는 reward들에 대한 정보를 포함하고 있습니다.

또한 agent 입장에서 다음 행동을 다음으로 가능한 state들의 value function으로 판단하는데 그러면 다음 state들에 대한 정보를 다 알아야하고 그 state로 가려면 어떻게 해야하는지(예를 들면 화살을 쓸 때 바람이 부니까 조금 오른쪽으로 쏜다라던지)도 알아야합니다.

따라서 state에 대한 value function 말고 action에 대한 value function을 구할 수 있는데 그것이 바로 action value function입니다. Action value function을 사용하면 value function과는 달리 단지 어떤 행동을 할지 action value function의 값을 보고 판단하면 되기 때문에 다음 state들의 value function을 알고 어떤 행동을 했을 때 거기에 가게 될 확률도 알아야하는 일이 사라집니다. action-value function에 대한 정의는 다음과 같습니다.

Definition

The *action-value function* $q_\pi(s, a)$ is the expected return starting from state s , taking action a , and then following policy π

$$q_\pi(s, a) = \mathbb{E}_\pi [G_t \mid S_t = s, A_t = a]$$

어떤 state s 에서 action a 를 취할 경우의 받을 return에 대한 기대값으로서 어떤 행동을 했을 때 얼마나 좋을 것인가에 대한 값입니다. 위에서 언급했던 이유로 앞으로 Value function이 아닌 action-value function을 사용할 겁니다. Action-value function은 다른 말로 Q-value로서 q-learning이나 deep q-network 같은 곳에 사용되는 q라는 것이 이것을 의미합니다.

Chapter 3 : Bellman Equation

앞서 배운 MDP의 value function들 사이에는 어떠한 연관성이 있는데 그것을 식으로 나타낼 수 있습니다. 그것을 Bellman Equation이라 하며 다음 챕터의 Dynamic Programming의 기반이 됩니다.

1. Bellman Expectation Equation

2. Bellman Optimality Equation

Bellman Expectation Equation

이전 Chapter에서 MDP(Markov Decision Process)와 Value function에 대해서 살펴보았습니다. Agent는 이 value function을 가지고 자신의 행동을 선택하게 됩니다. value function에는 두 가지가 있는데 다음과 같이 정의됩니다.

Definition

The state-value function $v_\pi(s)$ of an MDP is the expected return starting from state s , and then following policy π

$$v_\pi(s) = \mathbb{E}_\pi [G_t \mid S_t = s]$$

Definition

The action-value function $q_\pi(s, a)$ is the expected return starting from state s , taking action a , and then following policy π

$$q_\pi(s, a) = \mathbb{E}_\pi [G_t \mid S_t = s, A_t = a]$$

Bellman equation for Value function

Value function의 정의에서 value function은 다음과 같이 풀어써볼 수 있습니다. Return의 정의에 따라 풀어쓴 다음에 discount ratio로 묶어주면 아래와 같은 식이 됩니다. 아래와 같이 다음 state와 현재 state의 value function 사이의 관계를 식으로 나타낸 것을 Bellman equation이라고 합니다.

The **value function** can be decomposed into two parts:

- immediate reward R_{t+1}
- discounted value of successor state $\gamma v(S_{t+1})$

$$\begin{aligned}
 v(s) &= \mathbb{E}[G_t \mid S_t = s] \\
 &= \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots \mid S_t = s] \\
 &= \mathbb{E}[R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \dots) \mid S_t = s] \\
 &= \mathbb{E}[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\
 &= \mathbb{E}[R_{t+1} + \gamma v(S_{t+1}) \mid S_t = s]
 \end{aligned}$$

policy를 포함한 value function과 action value function도 Bellman equation의 형태로 아래와 같이 표현할 수 있습니다.

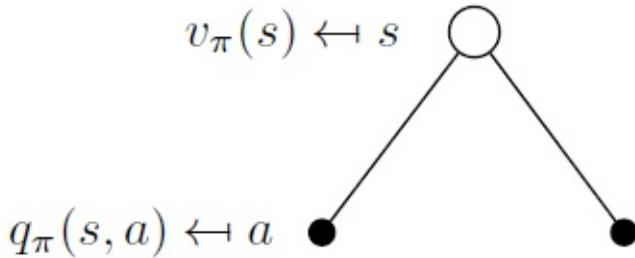
The **state-value function** can again be decomposed into immediate reward plus discounted value of successor state.

$$v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s]$$

The **action-value function** can similarly be decomposed,

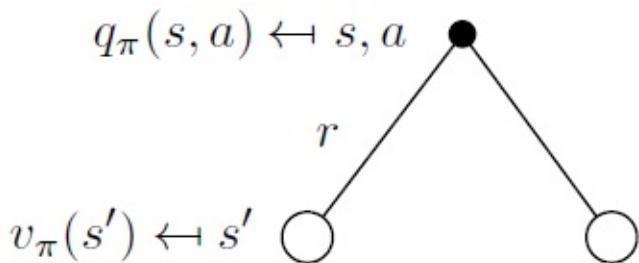
$$q_\pi(s, a) = \mathbb{E}_\pi[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) \mid S_t = s, A_t = a]$$

하지만 위와 같이 expectation으로 표현하는 것은 조금은 추상적이며 직관적이지 않을 수도 있습니다. 같은 식을 다른 방법으로 표현할 수도 있습니다. 현재 state의 value function과 next state의 value function의 상관관계의 식을 구하려면 그 사이에 있는 state-action pair(어떤 state에서 어떤 행동을 한 상태를 하나의 state같이 생각하는 개념)에 대해서 그 관계를 나눠볼 필요가 있습니다.



$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) q_\pi(s, a)$$

첫 번째로는 위와 같이 state와 state-action과 관계를 생각해 볼 수 있습니다. 흰 점은 state를 나타내고 검은 점은 state에서 action을 선택한 상황을 의미합니다. 간단히 표현하고자 그림에서는 가지를 2개씩만 쳤는데 이보다 더 많을 수도 적을 수도 있습니다. state에서 뻗어나가는 가지는 각각의 action의 개수만큼입니다. 이때 V와 q의 관계는 policy로서 위 식처럼 표현이 가능합니다. 각 action을 할 확률과 그 action을 해서 받는 expected return을 곱한 것을 더하면 현재 state의 value function이 된다는 것입니다.

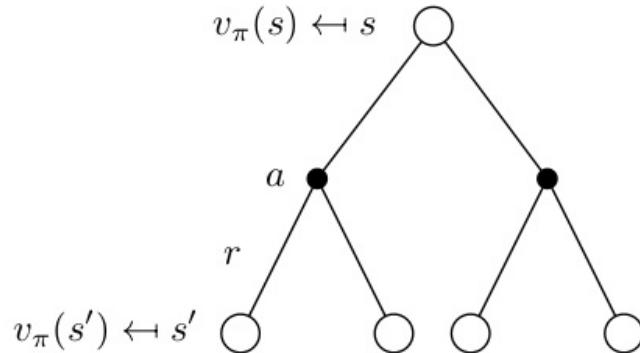


$$q_\pi(s, a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s')$$

위 그림보다 추가된 것은 r인데 reward를 의미합니다. reward의 정의에 state와 action이 모두 조건으로 들어가기 때문에 그림에서 검은 점 밑에 표시된 것을 볼 수 있습니다. 또한 action에서 퍼져나가는 가지는 만약 deterministic한 환경이라면 하나의 가지일 것이나 앞에서도 말했듯이 외부적인 요인에 의해서(바람이 분다던지) 같은 state에서 같은 action을 해줘도 다른 state로 갈 수도 있

습니다. 그러한 확률을 전에 언급했듯이 state transition probability matrix라고 합니다. action-value function은 immediate reward에다가 action을 취해 각 state로 갈 확률 곱하기 그 위치에서의 value function을(한 step이 지났으므로 discounted됩니다) 더한 것으로 표현할 수 있습니다.

이 두 도표를 합치면 아래와 같이 되고 식으로는 다음과 같이 표현할 수 있습니다.

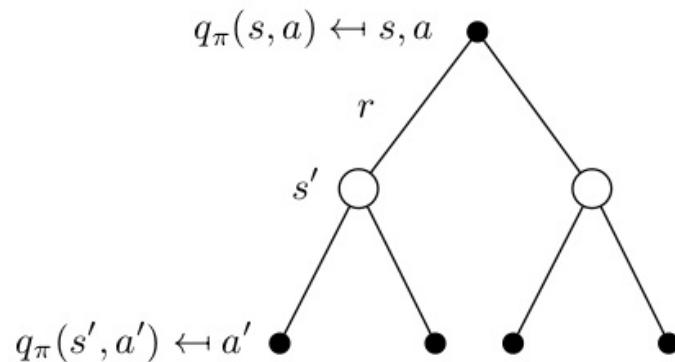


$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s') \right)$$

실제 강화학습으로 무엇인가를 학습시킬 때 reward와 state transition probability는 미리 알 수가 없습니다. 경험을 통해서 알아가는 것 입니다. 이러한 정보를 다 알면 MDP를 모두 안다고 표현하며 이러한 정보들이 MDP의 model이 됩니다. 강화학습의 큰 특징은 바로 MDP의 model를 몰라도 학습할 수 있다는 것 입니다. 따라서 reward function과 state transition probability를 모르고 학습하는 강화학습에서는 Bellman equation으로는 구할 수가 없습니다.

Bellman equation for Q-function

같은 식을 action value function에 대해서 작성하고 그림을 보면 다음과 같습니다.



$$q_{\pi}(s, a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \sum_{a' \in \mathcal{A}} \pi(a'|s') q_{\pi}(s', a')$$

Continuous-time optimization problem

보통 우리가 부르는 Bellman equation은 뒤에서 언급될 dynamic programming 같이 discrete한 time에서의 최적화 문제에 적용되는 식을 의미합니다. 하지만 시간이 연속적인 문제의 경우에는 Hamilton–Jacobi–Bellman equation이라고 부르는 식이 따로 있습니다. 관련 내용은 아래 홈페이지를 참고해주시길 바랍니다.

https://en.wikipedia.org/wiki/Bellman_equation

Bellman Optimality Equation

앞에서 현재 state의 value function과 next state의 value function 사이의 관계식인 Bellman equation에 대해서 살펴보았습니다. 하지만 왼쪽 목차를 보면 Bellman equation이 Bellman expectation equation과 Bellman optimality equation으로 구분됩니다.

Bellman expectation equation

다시 이전의 Bellman expectation equation을 살펴보겠습니다.

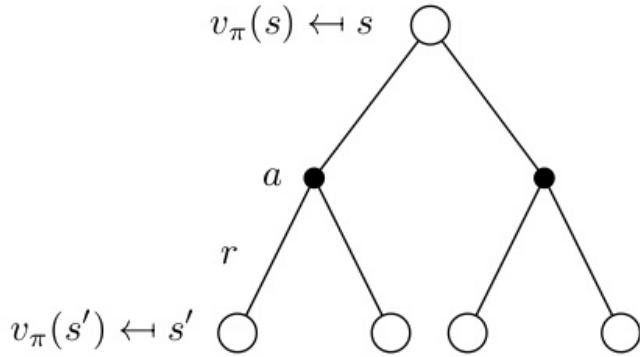
The state-value function can again be decomposed into immediate reward plus discounted value of successor state,

$$v_{\pi}(s) = \mathbb{E}_{\pi} [R_{t+1} + \gamma v_{\pi}(S_{t+1}) \mid S_t = s]$$

The action-value function can similarly be decomposed,

$$q_{\pi}(s, a) = \mathbb{E}_{\pi} [R_{t+1} + \gamma q_{\pi}(S_{t+1}, A_{t+1}) \mid S_t = s, A_t = a]$$

위 두 식은 expectation의 형태로 표현된 Bellman equation입니다. 따라서 이 식을 Bellman expectation equation이라고 부릅니다. 보통은 따로 expectation이라는 단어를 쓰지 않고 Bellman equation이라고 합니다. 위 식은 하지만 수학에서의 등호라기보다는 코딩에서 쓰이는 대입의 의미의 등호에 가깝습니다. 여기서 Backup의 개념이 나오게 됩니다.



$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s') \right)$$

Backup이라는 개념은 이렇습니다. 위에서 언급했듯이 코딩에서의 등호의 의미에 가까우며 오른쪽의 식을 왼쪽에 대입한다는 개념입니다. 즉 위 diagram에서 보면 미래의 값들(next state-value function)으로 현재의 value function을 구한다는 것이 Back-up입니다. 이 Back-up은 one step backup이 있고 multi step backup이 있습니다. 또한 Full-width backup(가능한 모든 다음 state의 value function을 사용하여 backup하는 것)과 sample backup(실제의 경험을 통해서 backup)이 있습니다. 뒤에서 다시 언급하겠지만 Full-width backup은 dynamic programming이고 sample backup은 reinforcement learning입니다.

Optimal value function

Bellman optimality equation을 보기 전에 optimal value function에 대해서 살펴보도록 하겠습니다. 강화학습의 목적이 accumulative future reward를 최대로 하는 policy를 찾는 것이라 했었습니다. optimal state-value function이란 현재 state에서 policy에 따라서 앞으로 받을 reward들이 달라지는데 그 중에서 앞으로 가장 많은 reward를 받을 policy를 따랐을 때의 value function입니다. optimal action-value function도 마찬가지로 현재 (s, a) 에서 얻을 수 있는 최대의 value function입니다.

Definition

The *optimal state-value function* $v_*(s)$ is the maximum value function over all policies

$$v_*(s) = \max_{\pi} v_{\pi}(s)$$

The *optimal action-value function* $q_*(s, a)$ is the maximum action-value function over all policies

$$q_*(s, a) = \max_{\pi} q_{\pi}(s, a)$$

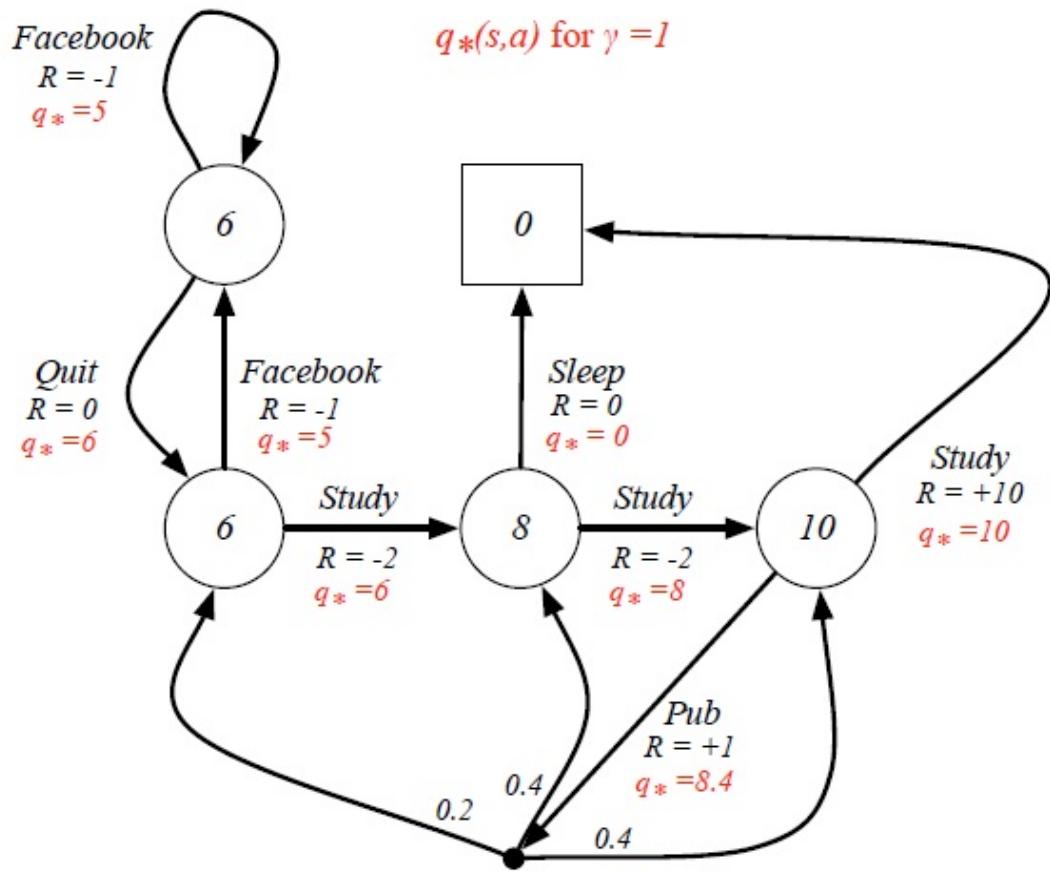
- The optimal value function specifies the best possible performance in the MDP.
- An MDP is “solved” when we know the optimal value fn.

즉, 현재 environment에서 취할 수 있는 가장 높은 값의 reward 총합입니다. 위의 두 식 중에서 두 번째 식, 즉 optimal action-value function의 값을 안다면 단순히 q값이 높은 action을 선택해주면 되므로 이 최적화 문제는 풀렸다라고 볼 수 있습니다. 강화학습 뿐만 아니라 Dynamic programming에서도 목표가 되는 optimal policy는 다음과 같습니다. optimal policy는 (s, a) 에서 action-value function이 가장 높은 action만을 고르기 때문에 deterministic합니다.

An *optimal policy* can be found by maximising over $q_*(s, a)$,

$$\pi_*(a|s) = \begin{cases} 1 & \text{if } a = \underset{a \in \mathcal{A}}{\operatorname{argmax}} q_*(s, a) \\ 0 & \text{otherwise} \end{cases}$$

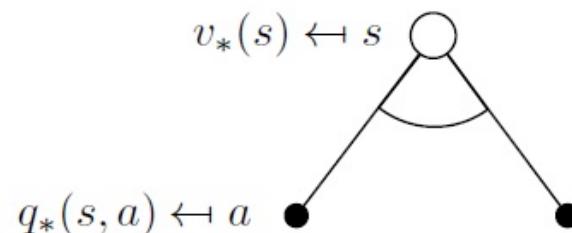
- There is always a deterministic optimal policy for any MDP
- If we know $q_*(s, a)$, we immediately have the optimal policy



Bellman Optimality Equation

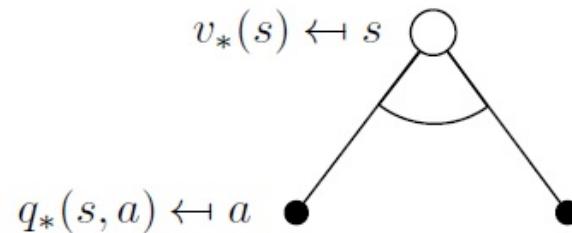
Bellman optimality equation는 위의 optimal value function 사이의 관계를 나타내주는 식입니다. 이전 backup diagram과 다른 점은 아래에는 호의 모양으로 표시된 "max"입니다.

The optimal value functions are recursively related by the Bellman optimality equations:

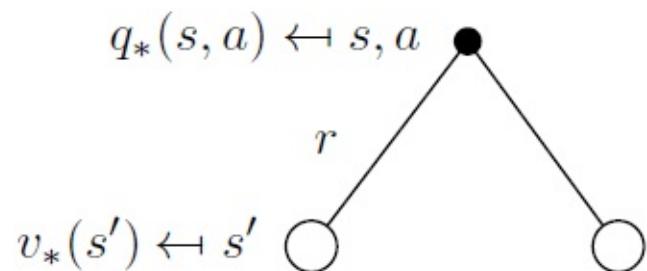


$$v_*(s) = \max_a q_*(s, a)$$

The optimal value functions are recursively related by the Bellman optimality equations:



$$v_*(s) = \max_a q_*(s, a)$$



$$q_*(s, a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s')$$

Chapter 4 : Dynamic Programming

1. Policy Iteration

2. Value Iteration

3. Example

Policy Iteration

앞으로 순서대로 Dynamic programming, Monte-carlo methods, Temporal difference methods를 살펴볼 것입니다. 세 방법에 대해서 Sutton교수님은 책에서 다음과 같이 설명하고 있습니다.

dynamic programming, Monte Carlo methods, and temporal-difference learning. Each class of methods has its strengths and weaknesses. Dynamic programming methods are well developed mathematically, but require a complete and accurate model of the environment. Monte Carlo methods don't require a model and are conceptually simple, but are not suited for step-by-step incremental computation. Finally, temporal-difference methods require no model and are fully incremental, but are more complex to analyze. The methods also differ in several ways with respect to their efficiency and speed of convergence.

또한 Dynamic Programming을 다음과 같이 정의하고 있습니다.

The term dynamic programming (DP) refers to a collection of algorithms that can be used to compute optimal policies given a perfect model of the environment as a Markov decision process (MDP)

Planning vs Learning

본격적으로 Dynamic Programming에 대해서 설명하기 전에 Planning과 Learning의 차이를 먼저 보도록 하겠습니다. 간단하게 말하자면 Planning이란 environment의 model을 알고서 문제를 푸는 것이고, Learning이란 environment의 model을 모르지만 상호작용을 통해서 문제를 푸는 것을 말합니다. Dynamic Programming은 Planning으로서 Environment의 model(reward, state transition matrix)에 대해서 안다는 전제로 문제를 푸는 방법(Bellman equation을 사용해서)을 말합니다. 강화학습과는 planning과 learning으로서 그 분류가 다르지만 강화학습이 이

DP(Dynamic programming)에 기반을 두고서 발전했기 때문에 DP를 이해하는 것이 상당히 중요합니다.

Two fundamental problems in sequential decision making

- Reinforcement Learning:
 - The environment is initially unknown
 - The agent interacts with the environment
 - The agent improves its policy
- Planning:
 - A model of the environment is known
 - The agent performs computations with its model (without any external interaction)
 - The agent improves its policy
 - a.k.a. deliberation, reasoning, introspection, pondering, thought, search

Prediction & Control

Dynamic Programming은 다음 두 step으로 나뉩니다. (1) Prediction (2)Control입니다. 이름은 중요하지 않고 이를 통해서 DP가 어떻게 작동하는지 대충 감을 잡으면 좋을 것 같습니다. 즉 현재 optimal하지 않는 어떤 policy에 대해서 value function을 구하고(prediction) 현재의 value function을 토대로 더 나은 policy를 구하고 이와 같은 과정을 반복하여 optimal policy를 구하는 것입니다.

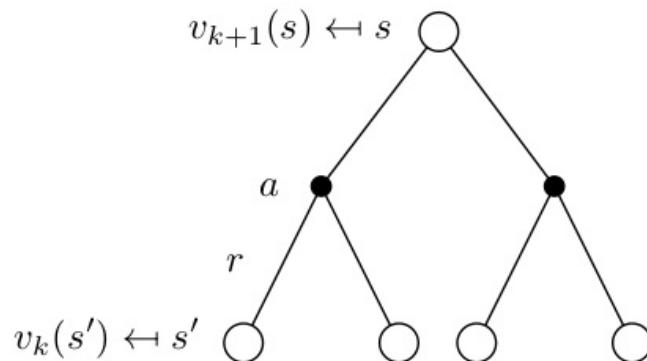
- For prediction:
 - Input: MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$ and policy π
 - or: MRP $\langle \mathcal{S}, \mathcal{P}^\pi, \mathcal{R}^\pi, \gamma \rangle$
 - Output: value function v_π
- Or for control:
 - Input: MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$
 - Output: optimal value function v_*
 - and: optimal policy π_*

Policy evaluation

- Problem: evaluate a given policy π
- Solution: iterative application of Bellman expectation backup
- $v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_\pi$
- Using *synchronous backups*,
 - At each iteration $k + 1$
 - For all states $s \in \mathcal{S}$
 - Update $v_{k+1}(s)$ from $v_k(s')$
 - where s' is a successor state of s

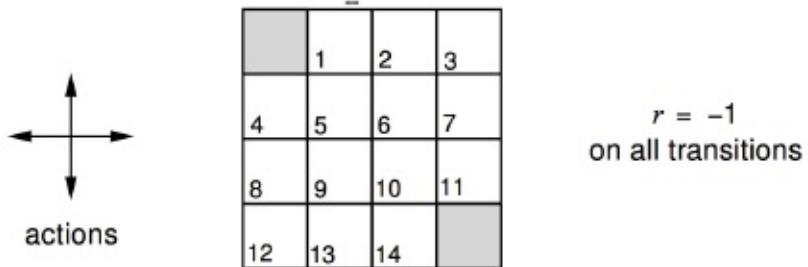
Policy evaluation은 prediction 문제를 푸는 것으로서 현재 주어진 policy에 대한 true value function을 구하는 것이고 Bellman equation을 사용합니다. 현재 Policy를 가지고 true value function을 구하는 것은 one step backup으로 구합니다.

이전의 Bellman equation의 그림과 다른 점은 value function에 k라는 iteration 숫자가 붙은 것입니다.



$$v_{k+1}(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_k(s') \right)$$

현재 상태의 value function을 update하는데 reward와 next state들의 value function을 사용하는 것입니다. 전체 MDP의 모든 state에 대해서 동시에 한 번씩 Bellman equation을 계산해서 update함으로서 k가 하나씩 올라가게 됩니다. 차례 차례 state 별로 구하는 것이 아니고 한 번에 계산해서 한 번에 value function을 update합니다. 이 Policy evaluation 과정을 예를 통해서 설명해 보겠습니다.



- Undiscounted episodic MDP ($\gamma = 1$)
- Nonterminal states 1, ..., 14
- One terminal state (shown twice as shaded squares)
- Actions leading out of the grid leave state unchanged
- Reward is -1 until the terminal state is reached
- Agent follows uniform random policy

$$\pi(n|\cdot) = \pi(e|\cdot) = \pi(s|\cdot) = \pi(w|\cdot) = 0.25$$

4X4 gridworld example입니다. state는 회색으로 표시된 terminal state가 두 개 있고 nonterminal state가 14개 있습니다. action은 상,하,좌,우 네 개를 취할 수 있고 time step이 지날 때마다 -1 의 reward를 받습니다. 따라서 agent는 reward를 최대로 하는 것이 목표이기 때문에 terminal state로 가능한 한 빨리 가려고 할 것입니다. 그러한 policy를 계산해내는 것이 DP이고 DP는 evaluation과 improve 두 단계로 나눠지게 됩니다. evaluation은 말 그대로 현재의 policy가 얼마나 좋은가를 판단하는 것이고 판단 기준은 그 policy를 따라가게 될 경우 받게 될 value function입니다. 처음 policy는 uniform random policy로서 모든 state에서 똑같은 확률로 상하좌우로 움직이는 policy입니다. 이 policy를 따라갈 경우 얻게 될 value function을 계산해보도록 하겠습니다.

v_k for the
Random Policy

 $k = 0$

0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0

 $k = 3$

0.0	-2.4	-2.9	-3.0
-2.4	-2.9	-3.0	-2.9
-2.9	-3.0	-2.9	-2.4
-3.0	-2.9	-2.4	0.0

 $k = 1$

0.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	0.0

 $k = 10$

0.0	-6.1	-8.4	-9.0
-6.1	-7.7	-8.4	-8.4
-8.4	-8.4	-7.7	-6.1
-9.0	-8.4	-6.1	0.0

 $k = 2$

0.0	-1.7	-2.0	-2.0
-1.7	-2.0	-2.0	-2.0
-2.0	-2.0	-2.0	-1.7
-2.0	-2.0	-1.7	0.0

 $k = \infty$

0.0	-14.	-20.	-22.
-14.	-18.	-20.	-20.
-20.	-20.	-18.	-14.
-22.	-20.	-14.	0.0

한 스텝씩 다음 식을 통해서 value function을 update합니다.

$$v_{k+1}(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in S} \mathcal{P}_{ss'}^a v_k(s') \right)$$

$k=1$ 을 보면 nonterminal state의 $V = 4 \times 0.25(-1 + 0) = -1$ 이 됩니다.

$k=2$ 에서 (1,2)state를 보면 $v = 1 \times 0.25(-1 + 0) + 3 \times 0.25(-1 + -1) = -1.7$ (소수 1째 자리까지만 표현)

(1,2)state에서 위로 action을 취하면 벽에 부딪히는데 그러면 자신의 state로 돌아오게 됩니다. 따라서 up, right, down의 action에 의해서 agent는 각각 -1의 value function을 가진 state에 도달합니다.

이런 식으로 무한대까지 계산하게 되면 현재 random policy에 대한 true value function을 구할 수 있고 이러한 과정을 policy evaluation이라고 합니다.

Policy iteration

이런 식으로 무한대까지 계산하게 되면 현재 random policy에 대한 true value function을 구할 수 있고 이러한 과정을 policy evaluation이라고 합니다.

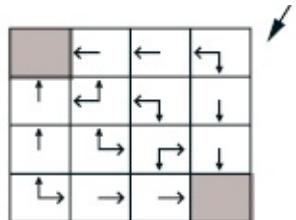
Policy iteration

해당 policy에 대한 참 값을 얻었으면 이제 policy를 더 나은 policy로 update해줘야 합니다. 그래야 점점 optimal policy에 가까워질 것입니다. 그러한 과정을 Policy improvement라고 합니다. improve하는 방법으로는 greedy improvement가 있습니다. 다음 식에서 보듯이 간단히 다음 state중에서 가장 높은 value function을 가진 state로 가는 것입니다. 즉, max를 취하는 것입니다.

Improve the policy by acting greedily with respect to v_π

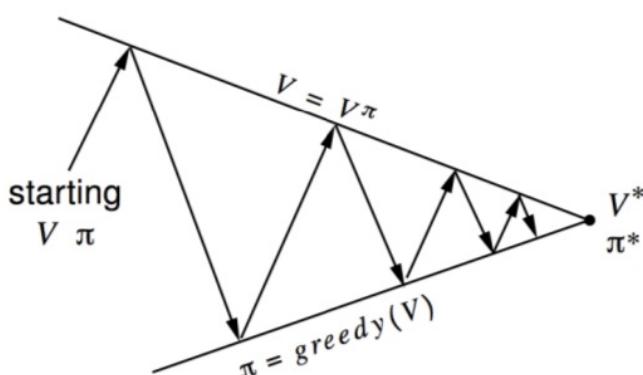
$$\pi' = \text{greedy}(v_\pi)$$

위와 같이 evaluation을 통해 구한 value function을 토대로 한 번 improve를 하게 되면 다음과 같



이 됩니다.

상당히 작은 gridworld이기 때문에 evaluation 한 번 improve 한 번 하면 optimal policy(위 그림)을 구할 수가 있는데 보통은 이러한 과정을 계속 반복 해줘야 optimal policy를 구할 수 있습니다. 이러한 반복되는 과정을 Policy Iteration이라고 합니다. 다음과 같이 그림으로 나타낼 수 있습니다.

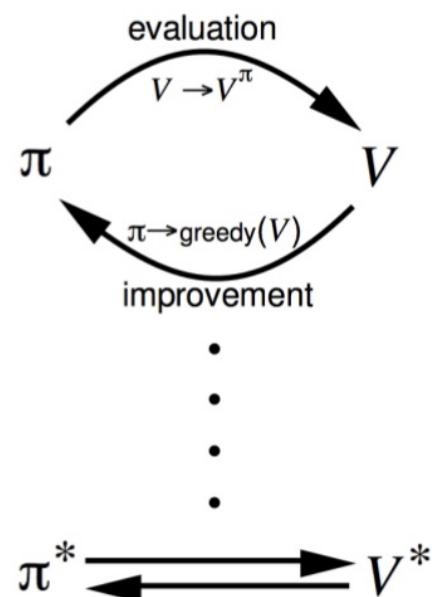


Policy evaluation Estimate v_π

Iterative policy evaluation

Policy improvement Generate $\pi' \geq \pi$

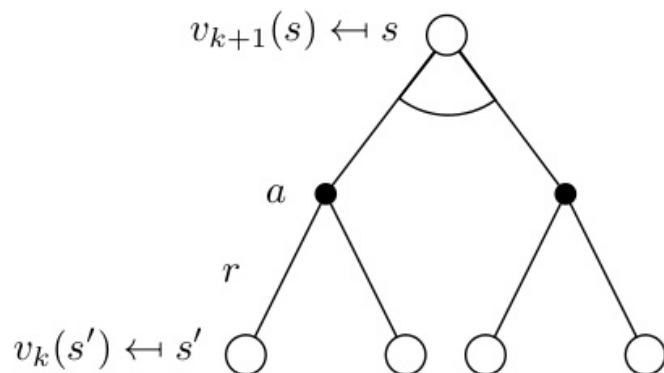
Greedy policy improvement



Value Iteration

Value Iteration

Value Iteration이 Policy Iteration과 다른 점은 Bellman Expectation Equation이 아니고 Bellman Optimality Equation을 사용한다는 것입니다. Bellman Optimality Equation은 optimal value function들 사이의 관계식입니다. 단순히 이 관계식을 iterative하게 변환시켜주면 됩니다. Policy Iteration의 경우에는 evaluation 할 때 수많은 계산을 해줘야하는 단점이 있었는데 그 evaluation 을 단 한 번만 하는 것이 value iteration입니다. 따라서 현재 value function을 계산하고 update할 때 max를 취함으로서 greedy하게 improve하는 효과를 줍니다. 따라서 한 번의 evaluation + improvement = value iteration이 됩니다.



$$v_{k+1}(s) = \max_{a \in \mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_k(s') \right)$$

value iteration의 예를 살펴봅시다. 아래 그림은 terminal state가 하나인 gridworld이고 time step 이 지나갈 때마다 -1의 reward를 받는 environment입니다. 아래 그림은 value iteration의 iteration숫자가 하나씩 늘어갈 때 각 state의 value function을 표시한 것 입니다. 변하지 않는 듯이 보이는 state들이 있지만 사실 같은 값으로 계속 update되고 있는 것입니다. V2의 (2,1)state의 value function -1을 보면

$$V2 = \max(-1 + V(\text{주변})) = -1$$

max를 취하고 terminal state가 계속 0의 value function을 가지기 때문에 (2,1) state는 계속 같은 값으로 업데이트가 됩니다. 그 이후로도 이와 같이 iteration을 진행하면 optimal value function을 구할 수 있고 그로 인해 optimal policy도 구할 수 있습니다.

g				

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

0	-1	-1	-1
-1	-1	-1	-1
-1	-1	-1	-1
-1	-1	-1	-1

0	-1	-2	-2
-1	-2	-2	-2
-2	-2	-2	-2
-2	-2	-2	-2

Problem V_1 V_2 V_3

0	-1	-2	-3
-1	-2	-3	-3
-2	-3	-3	-3
-3	-3	-3	-3

0	-1	-2	-3
-1	-2	-3	-4
-2	-3	-4	-4
-3	-4	-4	-4

0	-1	-2	-3
-1	-2	-3	-4
-2	-3	-4	-5
-3	-4	-5	-5

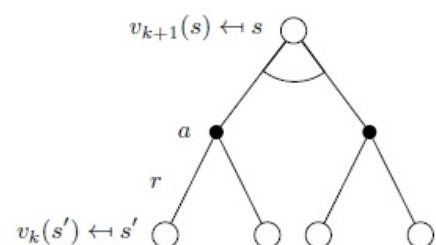
0	-1	-2	-3
-1	-2	-3	-4
-2	-3	-4	-5
-3	-4	-5	-6

V_4 V_5 V_6 V_7

Sample Backup

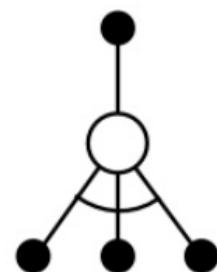
이렇게 Dynamic Programming에 대해서 살펴보았습니다. 처음에 언급했다시피 DP는 MDP에 대한 정보를 다 가지고 있어야 optimal policy를 구할 수 있습니다. 또한 DP는 full-width backup(한 번 update할 때 가능한 모든 successor state의 value function을 통해 update하는 방법)을 사용하고 있기 때문에 단 한 번의 backup을 하는 데도 많은 계산을 해야합니다. 또한 위와 같은 작은 gridworld 같은 경우는 괜찮지만 state 숫자가 늘어날수록 계산량이 기하급수적으로 증가하기 때문에 MDP가 상당히 크거나 MDP에 대해서 다 알지 못할 때는 DP를 적용시킬 수 없습니다.

- DP uses *full-width backups*
- For each backup (sync or async)
 - Every successor state and action is considered
 - Using knowledge of the MDP transitions and reward function
- DP is effective for medium-sized problems (millions of states)
- For large problems DP suffers Bellman's *curse of dimensionality*
 - Number of states $n = |\mathcal{S}|$ grows exponentially with number of state variables
- Even one backup can be too expensive



이때 등장하는 개념이 sample back-up입니다. 즉, 모든 가능한 successor state와 action을 고려하는 것이 아니고 Sampling을 통해서 한 길만 가보고 그 정보를 토대로 value function을 업데이트한다는 것입니다. 이렇게 할 경우 계산이 효율적이라는 장점도 있지만 "model-free"가 가능하다는 특징이 있습니다. 즉, DP의 방법대로 optimal한 해를 찾으려면 매 iteration마다 Reward function과 state transition matrix를 알아야 하는데 sample backup의 경우에는 아래 그림과 같이 $\langle S, A, R, S' \rangle$ 을 training set으로 실제 나온 reward와 sample transition으로서 그 두 개를 대체하게 됩니다. 따라서 MDP라는 model을 몰라도 optimal policy를 구할 수 있게 되고 "Learning"이 되는 것입니다. sample이라면 개념이 좀 어렵게 다가올 수도 있는데 처음에 강화학습이 trial and error로 학습한다 했던 것을 떠올려서 sample이 하나의 try라고 생각하면 이해가 잘 될 것 같습니다. 즉, 머리로 다 계산하고 있는 것이 아니고 일단 가보면서 겪는 experience로 문제를 풀겠다는 것입니다. DP를 sampling을 통해서 풀면서부터 "**Reinforcement Learning**"이 시작됩니다.

- In subsequent lectures we will consider *sample backups*
- Using sample rewards and sample transitions
 $\langle S, A, R, S' \rangle$
- Instead of reward function \mathcal{R} and transition dynamics \mathcal{P}
- Advantages:
 - Model-free: no advance knowledge of MDP required
 - Breaks the curse of dimensionality through sampling
 - Cost of backup is constant, independent of $n = |\mathcal{S}|$



Example

이번 chapter에서는 Dynamic programming에 대해서 살펴보았습니다. 앞으로 강화학습을 이해하는데 DP가 상당히 중요하고 실제로 코드를 한 번 봐보고 싶은 분들이 계실 것 같아서 예제를 보여드리려고 합니다.

UC Berkely AI lecture : CS188

<http://ai.berkeley.edu/home.html>

UC Berkely에서도 강화학습 관련하여 좋은 강의가 있다고 Introduction에서 말했었는데 위 사이트가 강의 사이트입니다. 여기서 저희는 Pacman Project 메뉴에서 P3 : Reinforcement Learning을 보도록 하겠습니다.

그 중에서도 저희가 볼 부분은 Question 1 (6 points): Value Iteration입니다. 이 프로젝트의 코드는 사이트에서 zip으로 다운받으실 수 있습니다. 실제 수업에서 프로젝트로 하는 것인데 다운받은 코드 중에서 학생들이 agent들의 코드를 짜는 프로젝트입니다. 저희는 value Iteration을 하는 agent를 만들면 되는데

`valueIterationAgents.py` : A value iteration agent for solving known MDPs.

이라는 파일의 내부를 코딩하면 됩니다. 물론 다른 파일들과 연결되어 있으므로 코드를 짜려면 다른 zip안에 있는 파일들도 봐야합니다. 코드는 파이썬으로 되어 있습니다.

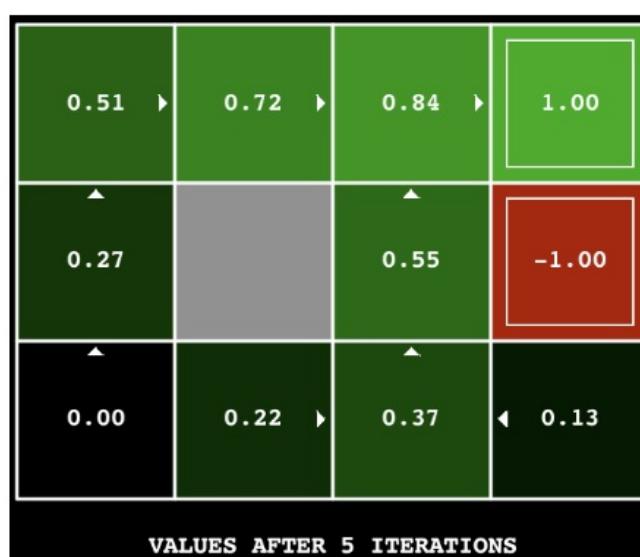
풀고 싶은 MDP는 아래 그림과 같습니다. (내용은 agent를 test하는 방법입니다) terminal state가 두 개가 있는데 오른쪽 맨 위의 칸은 1점으로 끝나는 state이고 아래는 -1점으로 끝나는 state입니다. 나머지 state에서 reward는 0입니다. 또한 stochastic하게 앞으로 가려할 때 좌, 우로 각각 10%의 확률로 가게됩니다. discount factor는 0.9입니다.

The following command loads your `ValueIterationAgent`, which will compute a policy and execute it 10 times. Press a key to cycle through values, Q-values, and the simulation. You should find that the value of the start state ($V(\text{start})$), which you can read off of the GUI) and the empirical resulting average reward (printed after the 10 rounds of execution finish) are quite close.

```
python gridworld.py -a value -i 100 -k 10
```

Hint: On the default BookGrid, running value iteration for 5 iterations should give you this output:

```
python gridworld.py -a value -i 5
```



valueIterationAgents.py라는 파일 내부를 보면 아래와 같이 Your Code Here라고 되어있는 부분들이 있는데 이러한 부분들을 채우는 것이 숙제입니다.

```
self.values = util.Counter() # A Counter is a dict with default 0

# Write value iteration code here
"*** YOUR CODE HERE ***"

def getValue(self, state):
    """
    Return the value of the state (computed in __init__).
    """
    return self.values[state]

def computeQValueFromValues(self, state, action):
    """
    Compute the Q-value of action in state from the
    value function stored in self.values.
    """
    "*** YOUR CODE HERE ***"
    util.raiseNotDefined()

def computeActionFromValues(self, state):
    """
    The policy is the best action in the given state
    according to the values currently stored in self.values.

    You may break ties any way you see fit. Note that if
    there are no legal actions, which is the case at the
    terminal state, you should return None.
    """
    "*** YOUR CODE HERE ***"
    util.raiseNotDefined()

def getPolicy(self, state):
    return self.computeActionFromValues(state)
```

이 사이트에서는 답이 나와있지 않으나 opensource인 만큼 이 문제를 풀어놓은 분들도 계십니다. 처음부터 어떻게 코딩을 해야할 지 모르겠다는 분들은 참고하시면 좋을 것 같습니다. 함께 공부하는 분들이 계시다면 함께 풀어봐도 재밌을 것 같습니다.

<https://github.com/mkapnick/pacman/blob/master/valueIterationAgents.py>

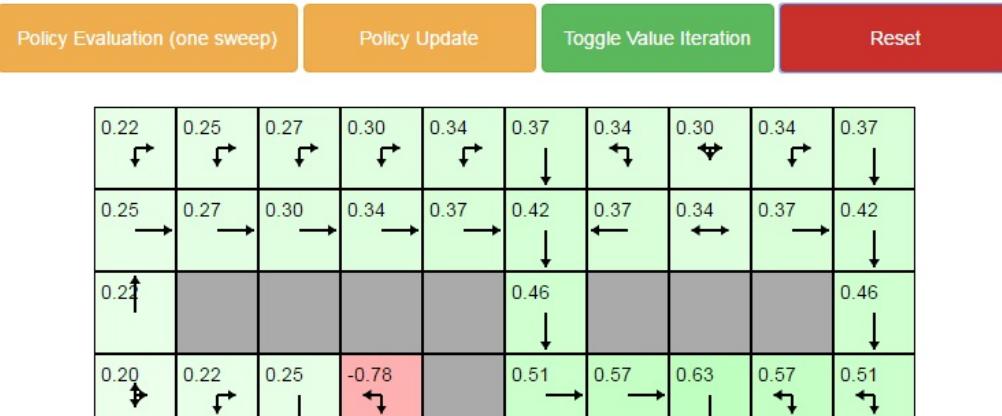
REINFORCEjs : Gridworld DP (Karpathy)

CS231n Convolutional Neural Network 강의로 유명한 Karpathy가 만든 사이트로서 자바스크립트로 위 예제보다는 더 큰 gridworld에서 dynamic programming의 과정을 볼 수 있도록 해놓았습니다. http://cs.stanford.edu/people/karpathy/reinforcejs/gridworld_dp.html

REINFORCEjs

About GridWorld: DP GridWorld: TD PuckWorld: DQN WaterWorld: DQN

GridWorld: Dynamic Programming Demo



Setting of Example

This is a toy environment called Gridworld that is often used as a toy model in the Reinforcement Learning literature. In this particular case:

State space: GridWorld has $10 \times 10 = 100$ distinct states. The start state is the top left cell. The gray cells are walls and cannot be moved to. Actions: The agent can choose from up to 4 actions to move around. In this example

Environment Dynamics: GridWorld is deterministic, leading to the same new state given each state and action

Rewards: The agent receives +1 reward when it is in the center square (the one that shows R 1.0), and -1 reward in a few states (R -1.0 is shown for these). The state with +1.0 reward is the goal state and resets the agent back to start.

In other words, this is a deterministic, finite Markov Decision Process (MDP) and as always the goal is to find an agent policy (shown here by arrows) that maximizes the future discounted reward. My favorite part is letting Value iteration converge, then change the cell rewards and watch the policy adjust.

Interface. The color of the cells (initially all white) shows the current estimate of the Value (discounted reward) of that state, with the current policy. Note that you can select any cell and change its reward with the Cell reward slider.

여기에서 살펴볼 것은 Policy Evaluation 코드와 Policy Update 코드입니다.

- Policy Evaluation

```

evaluatePolicy: function() {
    // perform a synchronous update of the value function
    var Vnew = zeros(this.ns); // initialize new value function array for each state
    for(var s=0;s < this.ns;s++) {
        var v = 0.0;
        var poss = this.env.allowedActions(s); // fetch all possible actions
        for(var i=0,n=poss.length;i < n;i++) {
            var a = poss[i];
            var prob = this.P[a*this.ns+s]; // probability of taking action under current
            policy
            var ns = this.env.nextStateDistribution(s,a); // look up the next state
            var rs = this.env.reward(s,a,ns); // get reward for s->a->ns transition
            v += prob * (rs + this.gamma * this.V[ns]);
        }
        Vnew[s] = v;
    }
    this.V = Vnew; // swap
},

```

각 state마다 가능한 action에 대해서 policy evaluation를 진행합니다. 각 state의 update될 값들을 행렬로 저장해놓고 있다가 모든 state에 대한 update값이 다 계산이 되면 한 번에 swap하는 형태입니다.

- Policy Update

```

updatePolicy: function() {
  // update policy to be greedy w.r.t. learned value function
  // iterate over all states...
  for(var s=0;s < this.ns;s++) {
    var poss = this.env.allowedActions(s);
    // compute value of taking each allowed action
    var vmax, nmax;
    var vs = [];
    for(var i=0,n=poss.length;i < n;i++) {
      var a = poss[i];
      // compute the value of taking action a
      var ns = this.env.nextStateDistribution(s,a);
      var rs = this.env.reward(s,a,ns);
      var v = rs + this.gamma * this.V[ns];
      // bookeeping: store it and maintain max
      vs.push(v);
      if(i === 0 || v > vmax) { vmax = v; nmax = 1; }
      else if(v === vmax) { nmax += 1; }
    }
    // update policy smoothly across all argmaxy actions
    for(var i=0,n=poss.length;i < n;i++) {
      var a = poss[i];
      this.P[a*this.ns+s] = (vs[i] === vmax) ? 1.0/nmax : 0.0;
    }
  },
},

```

Greedy action improvement로서 각 state에서 value function이 가장 큰 다음 state(value function 더하기 reward)를 선택하는 policy를 생성합니다.

이 두가지를 반복함으로서 Policy Iteration 알고리즘이 되는 것입니다.

Monte-Carlo Methods

1. Monte-Carlo prediciton

2. Monte-Carlo Control

Monte-Carlo Prediction

1. Model-Free

이전 챕터에서 Dynamic Programming에 대해서 배워보았습니다. Dynamic programming은 Bellman Equation을 통해서 optimal한 해를 찾아내는 방법으로서 MDP에 대한 모든 정보를 가진 상태에서 문제를 풀어나가는 방법을 이야기합니다.

특히 Environment의 model인 "Reward function"과 "state transition probabilities"를 알아야하기 때문에 Model-based한 방법이라고 할 수 있습니다. 이러한 방법에는 아래와 같은 문제점이 있습니다.

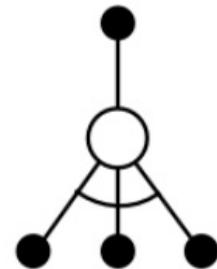
- (1) Full-width Backup --> expensive computation
- (2) Full knowledge about Environment

이러한 방식으로는 바둑같은 경우의 수가 많은 문제를 풀 수가 없고 실제 세상에 적용시킬 수가 없습니다. 사실 위와 같이 학문적으로 접근하지 않더라도 이러한 방법이 사람이 배우는 방법과 많이 다르다는 것을 알 수 있습니다. 이전에도 언급했었지만 사람은 모든 것을 다 안 후에 움직이지 않습니다. 만져보면서, 밟아보면서 조금씩 배워나갑니다. 이전에도 말했듯이 Trial and error를 통해서 학습하는 것이 강화학습의 큰 특징입니다.

따라서 DP처럼 full-width backup을 하는 것이 아니라 실제로 경험한 정보들로서 update를 하는 sample backup을하게 됩니다. 이렇게 실제로 경험한 정보들을 사용함으로서 처음부터 environment에 대해서 모든 것을 알 필요가 없습니다. Environment의 model을 모르고 학습하기

때문에 **Model-free**라는 말이 붙게 됩니다.

- In subsequent lectures we will consider *sample backups*
- Using sample rewards and sample transitions
 $\langle S, A, R, S' \rangle$
- Instead of reward function \mathcal{R} and transition dynamics \mathcal{P}
- Advantages:
 - Model-free: no advance knowledge of MDP required
 - Breaks the curse of dimensionality through sampling
 - Cost of backup is constant, independent of $n = |\mathcal{S}|$



현재의 policy를 바탕으로 움직여보면서 sampling을 통해 value function을 update하는 것을 **model-free prediction**이라 하고 policy를 update까지 하게 된다면 **model-free control**이라고 합니다.

이렇게 Sampling을 통해서 학습하는 model-free 방법에는 다음 두 가지가 있습니다.

- (1) Monte-Carlo
- (2) Temporal Difference

Monte-Carlo는 episode마다 update하는 방법이고 Temporal Difference는 time step마다 update하는 방법입니다. 이번 chapter에서는 Monte-Carlo Learning을 살펴보도록 하겠습니다.

2. Monte-Carlo

Monte-Carlo라는 말에 대해서 Sutton 교수님은 책에서 다음과 같이 이야기합니다.

The term "Monte Carlo" is often used more broadly for any estimation method whose operation involves a significant random component. Here we use it specifically for methods based on averaging complete returns

Monte-Carlo 단어 자체는 무엇인가를 random하게 측정하는 것을 뜻하는 말이라고 합니다. 강화 학습에서는 "averaging complete returns"하는 방법을 의미한다고 합니다. 이것은 무엇을 의미할까요?

Monte-Carlo와 Temporal Difference로 갈리는 것은 value function을 estimation하는 방법에 따라서입니다. value function이라는 것은 expected accumulative future reward로서 지금 이 state에서 시작해서 미래까지 받을 기대되는 reward의 총합입니다. 이것을 DP가 아니라면 어떻게 측정할 수 있을까요?

가장 기본적인 생각은 episode를 끝까지 가본 후에 받은 reward들로 각 state의 value function들을 거꾸로 계산해보는 것입니다. 따라서 MC(Monte-Carlo)는 끝나지 않는 episode에서는 사용할 수 없는 방법입니다. initial state S₁에서부터 시작해서 terminal state S_T까지 현재 policy를 따라서 움직이게 된다면 한 time step마다 reward를 받게 될 텐데 그 reward들을 기억해두었다가 S_T가 되면 뒤돌아보면서 각 state의 value function을 계산하게 됩니다. 아래 recall that the return이라고 되어있는데 제가 말한 것과 같은 말입니다. 순간 순간 받았던 reward들을 시간 순서대로 discount시켜서 sample return을 구할 수 있습니다.

- Goal: learn v_{π} from episodes of experience under policy π

$$S_1, A_1, R_2, \dots, S_k \sim \pi$$

- Recall that the *return* is the total discounted reward:

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

- Recall that the value function is the expected return:

$$v_{\pi}(s) = \mathbb{E}_{\pi} [G_t \mid S_t = s]$$

- Monte-Carlo policy evaluation uses empirical mean return instead of expected return

3. First-Visit MC vs Every-Visit MC

위에서는 한 에피소드가 끝나면 어떻게 하는지에 대해서 말했습니다. 하지만 multiple episode를 진행할 경우에는 한 episode마다 얻었던 return을 어떻게 계산해야 할까요? MC에서는 단순히 평균을 취해줍니다. 한 episode에서 어떤 state에 대해 return을 계산해놨는데 다른 episode에서도 그 state를 지나가서 다시 새로운 return을 얻었을 경우에 그 두개의 return을 평균을 취해주는 것이 있고 그 return들이 쌓이면 쌓일수록 true value function에 가까워지게 됩니다.

한 가지 고민해야 할 점이 있습니다. 만약에 한 episode 내에서 어떠한 state를 두 번 방문한다면 어떻게 해야 할까요? 이 때 어떻게 하냐에 따라서 두 가지로 나뉘게 됩니다.

- First-visit Monte-Carlo Policy evaluation
- Every-visit Monte-Carlo Policy evaluation

말 그대로 First-visit은 처음 방문한 state만 인정하는 것이고(두 번째 그 state 방문에 대해서는 return을 계산하지 않는) every-visit은 방문할 때마다 따로 따로 return을 계산하는 방법입니다. 두 방법은 모두 무한대로 갔을 때 true value function으로 수렴합니다. 하지만 First-visit이 좀 더 널리 오랫동안 연구되어 왔으므로 여기서는 First-visit MC에 대해서 다루도록 하겠습니다. 아래는 First-Visit Monte-Carlo Policy Evaluation에 대한 Silver 교수님 수업의 자료입니다.

First-Visit Monte-Carlo Policy Evaluation

- To evaluate state s
- The **first** time-step t that state s is visited in an episode,
- Increment counter $N(s) \leftarrow N(s) + 1$
- Increment total return $S(s) \leftarrow S(s) + G_t$
- Value is estimated by mean return $V(s) = S(s)/N(s)$
- By law of large numbers, $V(s) \rightarrow v_\pi(s)$ as $N(s) \rightarrow \infty$

4. Incremental Mean

위의 평균을 취하는 식을 좀 더 발전시켜보면 다음과 같습니다. 저희가 학습하는 방법은 여러개를 모아놓고 한 번에 평균을 취하는 것이 아니고 하나 하나 더해가며 평균을 계산하기 때문에 아래와 같은 Incremental Mean의 식으로 표현할 수 있습니다.

The mean μ_1, μ_2, \dots of a sequence x_1, x_2, \dots can be computed incrementally,

$$\begin{aligned}\mu_k &= \frac{1}{k} \sum_{j=1}^k x_j \\ &= \frac{1}{k} \left(x_k + \sum_{j=1}^{k-1} x_j \right) \\ &= \frac{1}{k} (x_k + (k-1)\mu_{k-1}) \\ &= \mu_{k-1} + \frac{1}{k} (x_k - \mu_{k-1})\end{aligned}$$

이 Incremental Mean을 위의 First-visit MC에 적용시키면 아래와 같습니다. 같은 식을 다르게 표현한 것입니다. 이 때, 분수로 가있는 $N(S_t)$ 가 점점 무한대로 가게 되는데 이를 알파로 고정시켜놓으면 효과적으로 평균을 취할 수 있게 됩니다. 맨 처음 정보들에 대해서 가중치를 덜 주는 형태라고 보시면 될 것 같습니다. (Complementary filter에 대해서 아시는 분은 이해가 쉬울 것 같습니다) 이와 같이 하는 이유는 강화학습이 stationary problem이 아니기 때문입니다. 매 episode마다 새로운 policy를 사용하기 때문에(아직 policy의 update에 대해서는 이야기하지 않았습니다) non-stationary problem이므로 update하는 상수를 일정하게 고정하는 것입니다.

- Update $V(s)$ incrementally after episode $S_1, A_1, R_2, \dots, S_T$
- For each state S_t with return G_t

$$\begin{aligned}N(S_t) &\leftarrow N(S_t) + 1 \\ V(S_t) &\leftarrow V(S_t) + \frac{1}{N(S_t)} (G_t - V(S_t))\end{aligned}$$

- In non-stationary problems, it can be useful to track a running mean, i.e. forget old episodes.

$$V(S_t) \leftarrow V(S_t) + \alpha (G_t - V(S_t))$$

5. Backup Diagram

이러한 MC의 backup과정을 그림으로 나타내면 아래와 같습니다.

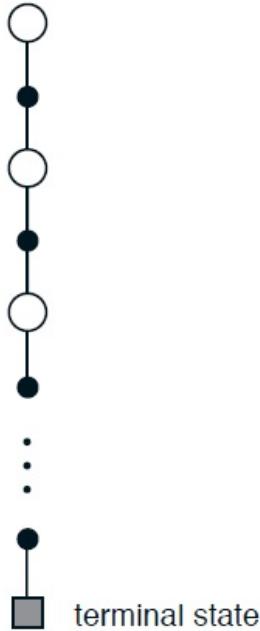


Figure 5.3: The backup diagram for Monte Carlo estimation of v_π .

DP의 backup diagram에서는 one step만 표시한 것에 비해서 MC에서는 terminal state까지 쭉 이어집니다. 또한 DP에서는 one-step backup에서 그 다음으로 가능한 모든 state들로 가지가 뻗었었는데 MC에서는 sampling을 하기 때문에 하나의 가지로 terminal state까지 가게됩니다.

Monte-Carlo는 처음에 random process를 포함한 방법이라고 말했었는데 episode마다 update하기 때문에 처음 시작이 어디었냐에 따라서 또한 같은 state에서 왼쪽으로 가느냐, 오른 쪽으로 가느냐에 따라서 전혀 다른 experience가 됩니다. 이러한 random한 요소를 포함하고 있어서 MC는 variance가 높습니다. 대신에 random인만큼 어딘가에 치우치는 경향은 적어서 bias는 낮은 편입니다.

6. Example

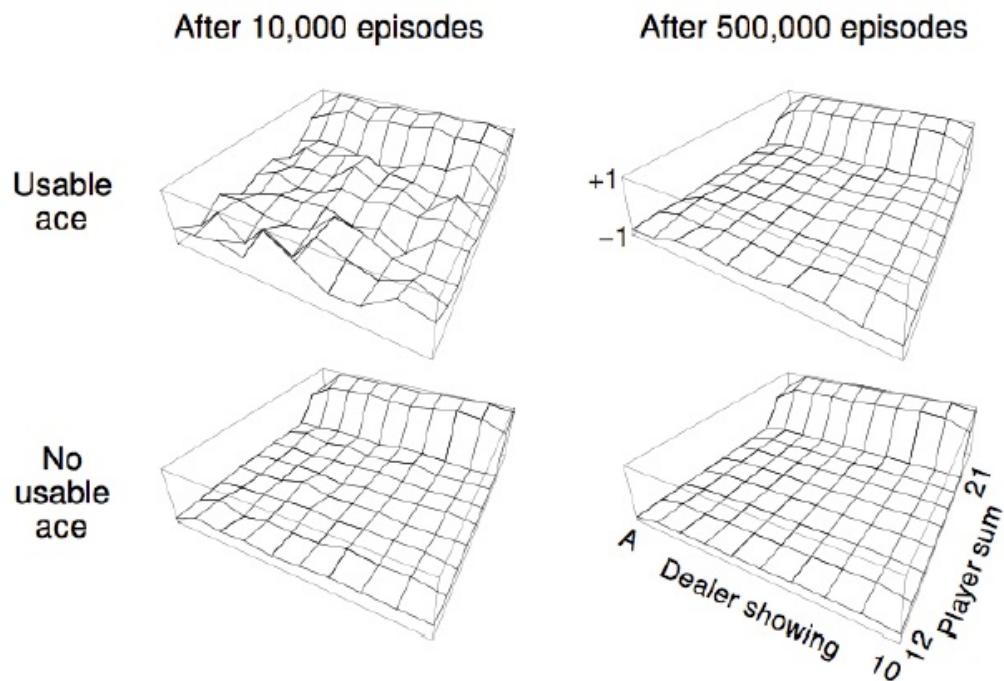
Silver교수님 강의에서는 Blackjack 게임을 Monte-Carlo policy evaluation의 예제로 설명합니다. 저에게는 블랙잭 자체의 룰을 몰라서 이 예제가 어렵게 다가왔기 때문에 예제 자체에 대해서는 설명하지 않겠습니다. 게임의 설정은 아래와 같습니다.

Blackjack Example

- States (200 of them):
 - Current sum (12-21)
 - Dealer's showing card (ace-10)
 - Do I have a "useable" ace? (yes-no)
- Action **stick**: Stop receiving cards (and terminate)
- Action **twist**: Take another card (no replacement)
- Reward for **stick**:
 - +1 if sum of cards > sum of dealer cards
 - 0 if sum of cards = sum of dealer cards
 - -1 if sum of cards < sum of dealer cards
- Reward for **twist**:
 - -1 if sum of cards > 21 (and terminate)
 - 0 otherwise
- Transitions: automatically **twist** if sum of cards < 12



이와 같은 게임에서 policy를 정해놓고 계속 computer로 시뮬레이션을 돌리면서 얻은 reward들로 sample return을 구하고 그 return들을 incrementally mean을 취하면 아래와 같은 그래프로 표현할 수 있습니다.



Policy: **stick** if sum of cards ≥ 20 , otherwise **twist**

Ace의 설정에 따라서 위 아래로 나뉘는 데 위의 두 개의 그래프만 보도록 하겠습니다. policy는 그래프 아래에 설정되어 있습니다. state는 x,y 2차원으로 표현되며 z축은 각 state의 value function

이 됩니다. episode를 지남에 따라 점차 어떠한 모양으로 수렴하는 것을 볼 수 있습니다. 500,000 번쯤 게임을 play했을 경우에 거의 수렴한 것을 볼 수 있습니다. 앞으로 할 Control에서는 이 value function을 토대로 policy를 update하게 됩니다. 그래서 어떤 것이 좋은 policy인지 비교가 가능해집니다.

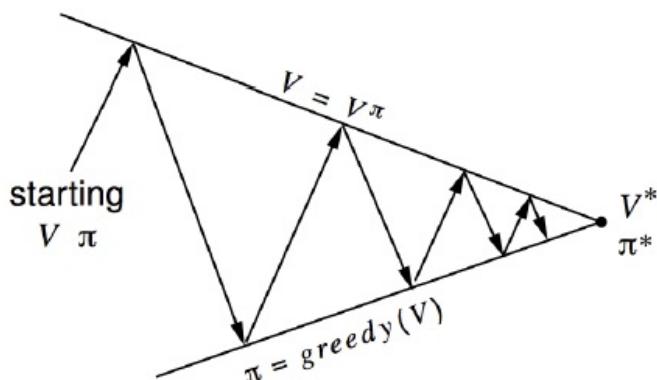
Monte-Carlo Control

1. Monte-Carlo Policy Iteration

이전에는 Monte-Carlo Policy Evaluation = Prediction을 보았습니다. Dynamic Programming 때도 evaluation + Improvement = Policy Iteration이 되었듯이 MC에서도 Monte-Carlo Policy Evaluation + Policy Improvement를 하면 Monte-Carlo Policy Iteration이 됩니다.

다시 DP의 Policy Iteration을 생각해봅시다. 현재 policy를 토대로 Value function을 iterative하게 계산해서 policy를 evaluation(true value function에 수렴할 때까지) 그 value function을 토대로 greedy하게 policy를 improve하고 그러한 과정을 optimal policy를 얻을 때까지 반복하였습니다.

Generalised Policy Iteration (Refresher)

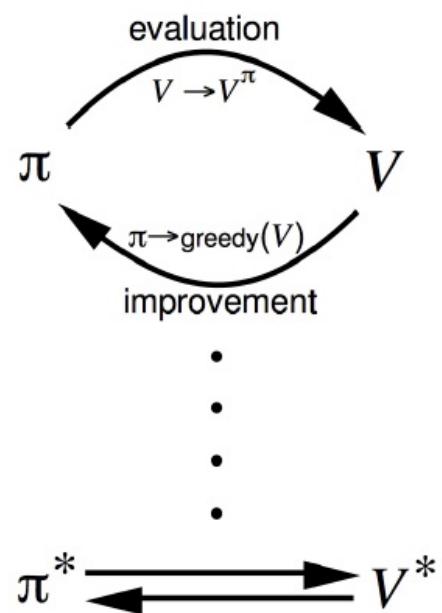


Policy evaluation Estimate v_π

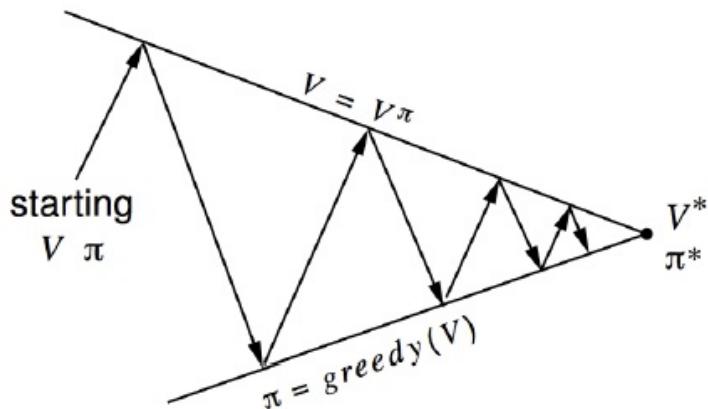
e.g. Iterative policy evaluation

Policy improvement Generate $\pi' \geq \pi$

e.g. Greedy policy improvement



여기에서 Policy evaluation만 Monte-Carlo policy evaluation으로 바꾸어주면 Monte-Carlo Policy Iteration이 됩니다.



Policy evaluation Monte-Carlo policy evaluation, $V = v_\pi$?

Policy improvement Greedy policy improvement?

2. Monte-Carlo Control

하지만 Monte-Carlo Policy Iteration에는 세 가지 문제점이 있습니다.

- Value function
- Exploration
- Policy Iteration

(1) Value function

현재 MC로서 Policy를 evaluation하는데 Value function을 사용하고 있습니다. 하지만 value function을 사용하면 policy를 improve(greedy)할 때 문제가 발생합니다. 원래 MC를 했던 이유는 Model-free를 하기 위해서였는데 value function으로 policy를 improve하려면 MDP의 model을 알아야 합니다. 아래와 같이 다음 policy를 계산하려면 reward와 transition probability를 알아야 할

수 있습니다. 따라서 value function 대신에 action value function을 사용합니다. 그러면 이러한 문제없이 model-free가 될 수 있습니다.

- Greedy policy improvement over $V(s)$ requires model of MDP

$$\pi'(s) = \operatorname{argmax}_{a \in \mathcal{A}} \mathcal{R}_s^a + \mathcal{P}_{ss'}^a V(s')$$

- Greedy policy improvement over $Q(s, a)$ is model-free

$$\pi'(s) = \operatorname{argmax}_{a \in \mathcal{A}} Q(s, a)$$

(2) Exploration

현재는 policy improve는 greedy policy improvement를 사용하고 있습니다. 하지만 계속 현재 상황에서 최고의 것만 보고 판단을 할 경우에는 optimal로 가는 것이 아니고 local optimum에 빠져버릴 수가 있습니다. 충분히 exploration을 하지 않았기 때문에 global optimum에 가지 못했던 것입니다. 현재 action a 가 가장 높은 value function을 가진다고 측정이 되어서 action을 a 만 하면 사실은 b 가 더 높은 value function을 가질 수도 있는 가능성을 배제해버리게 됩니다. 마치 대학교나 성적만 보고 사람을 뽑아쓰는 것과 같은 실수일지도 모릅니다. 따라서 그에 따른 대안으로서 일정 확률로 현재 상태에서 가장 높은 가치를 가지지 않은 다른 action을 하도록 합니다. 그 일정 확률을 epsilon이라하며 그러한 improve방법을 epsilon greedy policy improvement라고 합니다. 아래와 같이 선택할 수 있는 action이 m 개 있을 경우에 greedy action(가장 action value function이 높은 action)과 다른 action들을 아래와 같은 확률로 나눠서 선택합니다. 이로서 부족했던 exploration을 할 수 있게 된 것입니다.

- Simplest idea for ensuring continual exploration
- All m actions are tried with non-zero probability
- With probability $1 - \epsilon$ choose the greedy action
- With probability ϵ choose an action at random

$$\pi(a|s) = \begin{cases} \epsilon/m + 1 - \epsilon & \text{if } a^* = \underset{a \in \mathcal{A}}{\operatorname{argmax}} Q(s, a) \\ \epsilon/m & \text{otherwise} \end{cases}$$

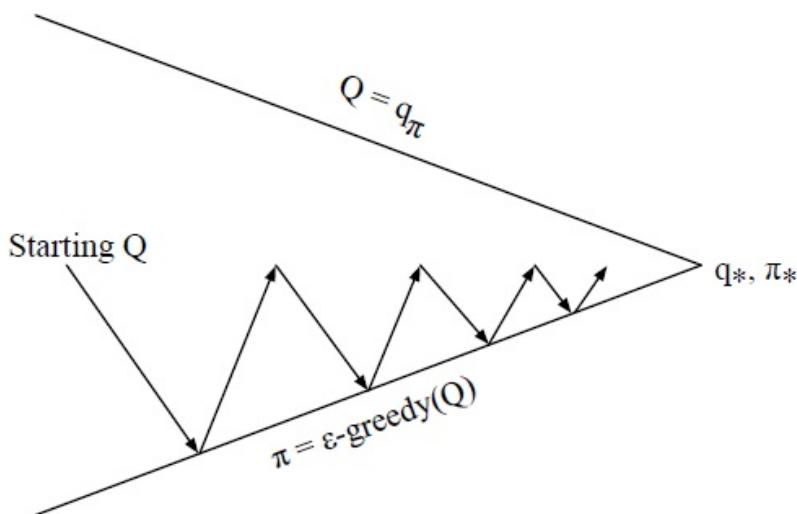


(3) Policy Iteration

Policy Iteration에서는 evaluation 과정이 true value function으로 수렴할 때까지 해야하는데 그렇게 하지 않고 한 번 evaluation한 다음에 policy improve를 해도 optimal로 간다고 말했습니다. 그것이 Value iteration이었는데 Monte-Carlo에서도 마찬가지로 이 evaluation 과정을 줄임으로서

Monte-Carlo policy iteration에서 Monte-Carlo Control이 됩니다. 결국 Monte-Carlo Control은 다음과 같습니다.

Monte-Carlo Control



Every episode:

Policy evaluation Monte-Carlo policy evaluation, $Q \approx q_\pi$

Policy improvement ϵ -greedy policy improvement

3. GLIE

GLIE란 Greedy in the Limit with Infinite Exploration을 뜻하고 sutton교수님 책에는 안 나왔지만 Silver교수님 강의에서 나왔던 내용입니다. 학습을 해나감에 따라 충분한 탐험을 했다면 greedy policy에 수렴하는 것을 말합니다. 하지만 epsilon greedy policy로서는 greedy하게 하나의 action 만 선택하지 않는데 이럴 경우는 GLIE하지 않습니다. 보통 learning을 통해서 배우려는 optimal policy는 greedy policy입니다. 따라서 exploration문제 때문에 사용하는 epsilon greedy에서

epsilon이 시간에 따라서 0으로 수렴한다면 epsilon greedy 또한 GLIE가 될 수 있습니다. 후에는 이러한 문제를 off-policy control로서 Q-learning을 쓰면서 해결하게 됩니다.

Definition

Greedy in the Limit with Infinite Exploration (GLIE)

- All state-action pairs are explored infinitely many times,

$$\lim_{k \rightarrow \infty} N_k(s, a) = \infty$$

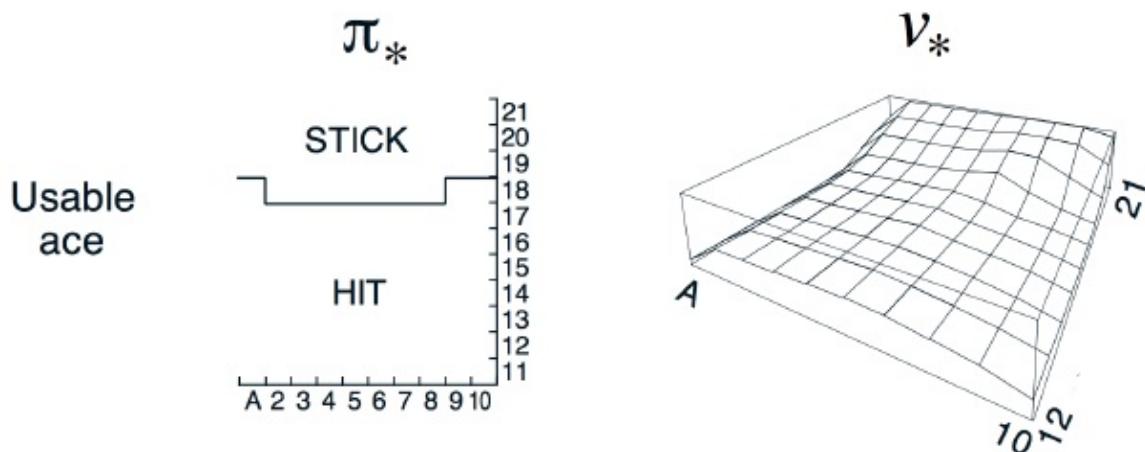
- The policy converges on a greedy policy,

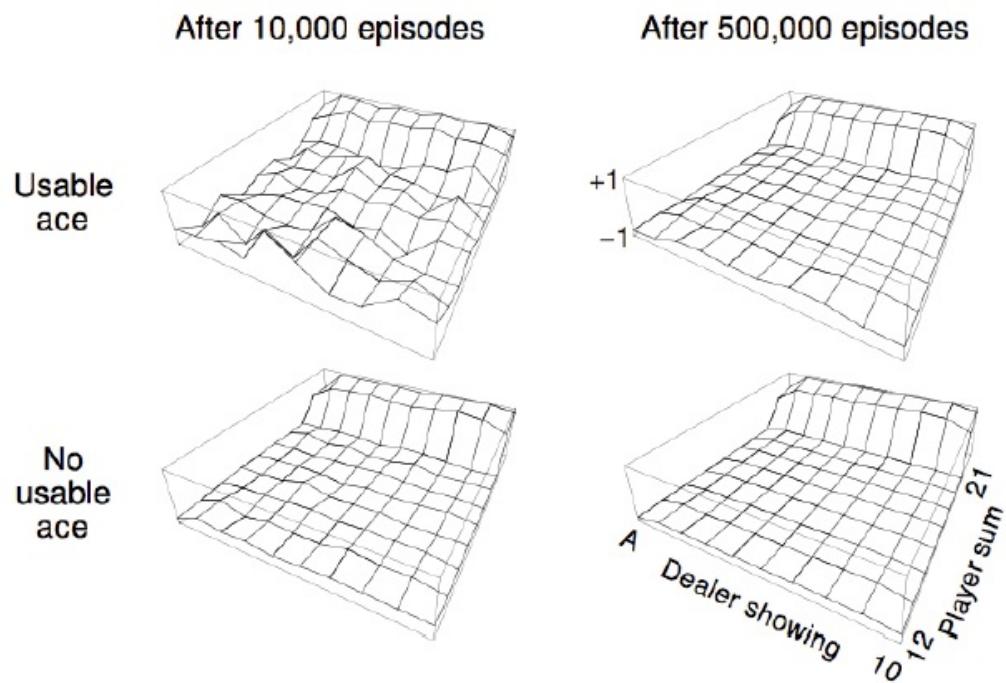
$$\lim_{k \rightarrow \infty} \pi_k(a|s) = \mathbf{1}(a = \operatorname{argmax}_{a' \in \mathcal{A}} Q_k(s, a'))$$

- For example, ϵ -greedy is GLIE if ϵ reduces to zero at $\epsilon_k \equiv \frac{1}{k}$

4. Blackjack

Monte-Carlo prediction에서 봤던 블랙잭 예제를 policy evaluation + policy improvement를 해서 다시 계산해보면, 즉 agent를 학습시켜보면 아래와 같이 됩니다. 그 아래의 Monte-Carlo Policy evaluation과 비교를 하면 전체적인 value function이 발전되었다는 것을 알 수 있고 이 policy가 optimal policy가 됩니다.





Policy: **stick** if sum of cards ≥ 20 , otherwise **twist**

Temporal Difference Methods

- 1. TD Prediction**
- 2. TD Control**
- 3. Eligibility Traces**

TD Prediction

1. Temporal Difference

이전 chapter에서 배웠던 Monte-Carlo Control은 Model-Free Control입니다. Model-Free라는 점에서 강화학습이지만 단점이 있습니다. 바로 online으로 바로바로 학습할 수가 없고 꼭 끝나는 episode여야 한다는 단점이 있습니다. 끝나지 않더라도 episode가 길 경우에는(예를 들어 atari 게임이 아니라 starcraft 같은 게임) 학습하기 어려운 단점이 있습니다. 따라서 자연스럽게도 꼭 episode가 끝나지 않더라도 DP처럼 time step마다 학습할 수 있지 않나?라는 생각을 하게 됩니다. 이게 바로 Temporal Difference이며 Sutton 교수님 책에서는 아래와 같이 소개하고 있습니다.

If one had to identify one idea as central and novel to reinforcement learning, it would undoubtedly be temporal-difference (TD) learning. TD learning is a combination of Monte Carlo ideas and dynamic programming (DP) ideas. Like Monte Carlo methods, TD methods can learn directly from raw experience without a model of the environment's dynamics. Like DP, TD methods update estimates based in part on other learned estimates, without waiting for a final outcome (they bootstrap)

Temporal difference(TD)는 MC와 DP를 섞은 것으로서 MC처럼 raw experience로부터 학습할 수 있지만 DP처럼 time step마다 학습할 수 있는 방법입니다. 마지막에 "bootstrap"이라고 하는데 이 말은 무엇을 뜻할까요?

대학생활을 예로 들어보겠습니다. MC는 대학교에 들어와서 졸업을 한 다음에 그 동안을 돌아보며 "이건 더 했어야했고 술은 덜 마셔야했다"라고 생각하며 다시 대학교를 들어가서 대학생활을 하면서 졸업할 때까지 똑같이 살다가 다시 졸업하고 자신을 돌아보는 반면에 TD같은 경우는 같은 대학교를 다니고 있는 2학년 선배가 1학년 선배를 이끌어주는 것을 말합니다.

사실은 둘 다 대학교를 졸업을 해보지 않은 상태에서 (잘 모르는 상황에서)이끌어 주는 것이지만 대학교를 다니면서 바로 바로 자신을 고쳐나가기 때문에 어쩌면 더 옳은 방법일지도 모릅니다. TD는 따라서 현재의 value function을 계산하는데 앞선(앞선이라고 표현하기에는 좀 애매한 부분이 있지만) 주변의 state들의 value function을 사용합니다. 이 것은 이전에 배웠던 Bellman Equation이며 따라서 Bellman equation 자체가 Bootstrap하는 것이라고 볼 수 있습니다.

2. TD(0)

TD는 Monte-Carlo + DP라고 말했습니다. 이전에 봤던 Monte-Carlo prediction에서 incremental mean을 보면 아래와 같이 return을 사용해서 update합니다. TD에서는 이 G_t 를

$\$ \$ \{ R \} \{ t+1 \} + \gamma V(\{ S \} \{ t+1 \}) \$ \$$ 로 바꿔서 아래과 같은 식이 됩니다. Temporal Difference learning 방법에도 여러가지가 있는데 그 중에서 가장 간단한 방법은 TD(0)이고 방금 말한 방법이 바로 TD(0)입니다. $\$ \$ \{ R \} \{ t+1 \} + \gamma V(\{ S \} \{ t+1 \}) \$ \$$ 를 TD target이라고 부르고 그 타겟과 현재의 value function과의 차이를 TD error라고 부릅니다.

- Goal: learn v_π online from experience under policy π
- Incremental every-visit Monte-Carlo
 - Update value $V(S_t)$ toward *actual return* G_t
$$V(S_t) \leftarrow V(S_t) + \alpha (G_t - V(S_t))$$
- Simplest temporal-difference learning algorithm: TD(0)
 - Update value $V(S_t)$ toward *estimated return* $R_{t+1} + \gamma V(S_{t+1})$
$$V(S_t) \leftarrow V(S_t) + \alpha (R_{t+1} + \gamma V(S_{t+1}) - V(S_t))$$

- $R_{t+1} + \gamma V(S_{t+1})$ is called the *TD target*
- $\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$ is called the *TD error*

TD(0)의 알고리즘을 살펴보고 backup diagram을 보면 아래와 같습니다.

```

Input: the policy  $\pi$  to be evaluated
Initialize  $V(s)$  arbitrarily (e.g.,  $V(s) = 0, \forall s \in \mathcal{S}^+$ )
Repeat (for each episode):
    Initialize  $S$ 
    Repeat (for each step of episode):
         $A \leftarrow$  action given by  $\pi$  for  $S$ 
        Take action  $A$ ; observe reward,  $R$ , and next state,  $S'$ 
         $V(S) \leftarrow V(S) + \alpha[R + \gamma V(S') - V(S)]$ 
         $S \leftarrow S'$ 
    until  $S$  is terminal
  
```

Figure 6.1: Tabular TD(0) for estimating v_π .



Figure 6.2: The backup diagram for TD(0).

3. MC vs TD

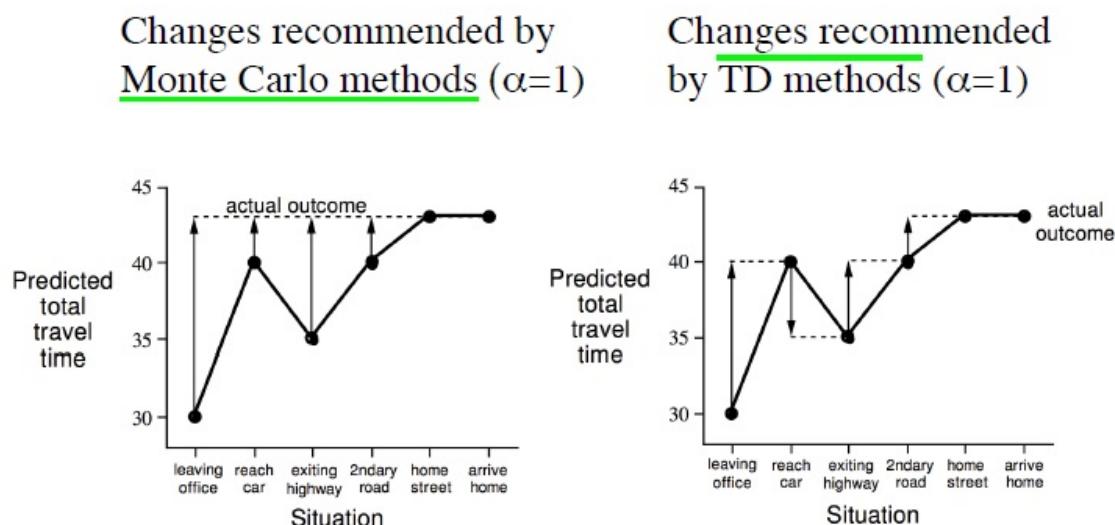
MC와 TD를 비교한 예제는 다음과 같습니다. 직장에서 집까지 가는 episode에 대한 prediction에 대한 예제인데(Control은 아닙니다) 직장에서 출발하고, 차가 막히고, 비가 오고, 고속도로를 나가고 집에 도착하고 이런 상태들을 "state"로 잡고 각각의 state에 있을 때 앞으로 얼마나 걸릴지에 대해 agent가 predict한 것에 대한 비교입니다. 그 state에서 앞으로 집으로 가기까지 얼마나 걸릴지가 value function이라 하면 각각의 상황에서 agent는 value function을 predict합니다. 그리고 state가 지날때마다 실제로 흐른 시간이 reward가 될 것 입니다.

Driving Home Example

State	Elapsed Time (minutes)	Predicted Time to Go	Predicted Total Time
leaving office	0	30	30
reach car, raining	5	35	40
exit highway	20	15	35
behind truck	30	10	40
home street	40	3	43
arrive home	43	0	43

MC의 경우에는 다 도착한 다음에 각각의 state에서 예측했던 value function과 실제로 받은 return을 비교해서 update를 하게됩니다. 하지만 TD에서는 한 스텝 진행을 하면 아직 도착을 하지 않아서 얼마가 걸릴지는 정확히 모르지만 한 스텝 동안 지났던 시간을 토대로 value function을 update합니다. 따라서 실제로 도착을 하지 않아도, final outcome을 모르더라고 학습할 수 있는 것이 TD의 장점이며 매 step마다 학습할 수 있다는 것도 장점입니다.

Driving Home Example: MC vs. TD



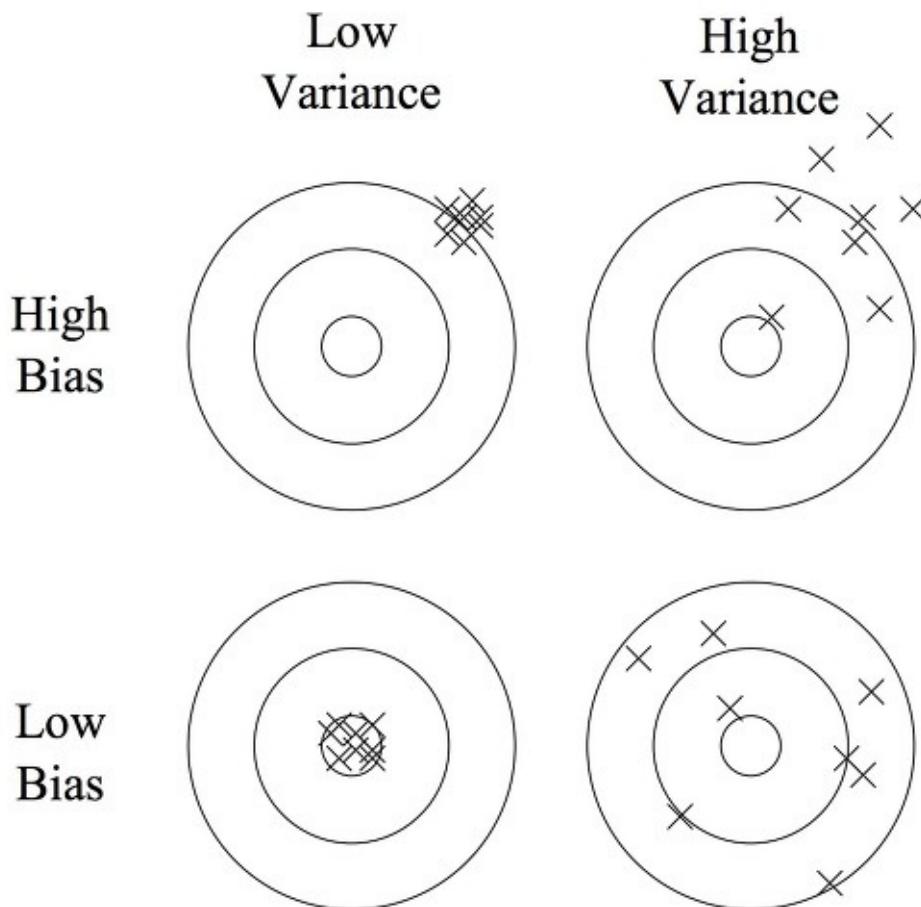
- TD can learn before knowing the final outcome
 - TD can learn online after every step
 - MC must wait until end of episode before return is known
- TD can learn without the final outcome
 - TD can learn from incomplete sequences
 - MC can only learn from complete sequences
 - TD works in continuing (non-terminating) environments
 - MC only works for episodic (terminating) environments

Bias/Variance Trade-Off

또 한가지 중요한 차이점은 바로 bias와 variance입니다. 이 두 개념이 무엇일까요?

<http://bywords.tistory.com/entry/%EB%B2%88%EC%97%AD-%EC%9C%A0%EC%B9%98%EC%9B%90%EC%83%9D%EB%8F%84-%EC%9D%B4%ED%95%B4%ED%95%A0-%EC%88%98-%EC%9E%88%EB%8A%94-biasvariance-tradeoff>

아래와 같이 중앙이 있다면 중앙으로부터 전체적으로 많이 벗어나게 되면 bias가 높다, 혹은 biased됐다라고 하고 전체적으로 많이 퍼져있으면(가운데로부터 벗어난 것이랑 관계없이) variance가 높다고 합니다.



둘 다 낮으면 좋겠지만 보통은 Trade-Off관계에 있어서 하나가 낮아지면 하나가 높아지는 관계에 있습니다. TD는 bias가 높고 MC는 variance가 높습니다. 둘 다 학습에 방해가 되는 요소입니다.

TD는 한 episode안에서 계속 업데이트를 하는데 보통은 그 전의 상태가 그 후에 상태에 영향을 많이 주기 때문에 학습이 한 쪽으로 치우쳐지게 됩니다. 계속 같은 분야 사람들과 이야기를 하면 생각의 폭이 좁아지는 것과 비슷하다 할까요 혹은 한 사람의 조언만 들으면 안되는 이유가 그 사람은 그 사람의 인생만 살아봤기 때문에 한 쪽으로 치우쳐질 즉, bias가 높을 수 있으니 여러 사람의 의견을 들어보는 것이 좋습니다.

하지만 너무 여러 사람의 의견을 듣다보면 이도 저도 아니게되서 결정을 못하게 되곤 합니다. 이러한 저희의 경험들이 사실 bias/variance trade-off와 관련이 되어있습니다. MC가 variance가 높은 이유는 앞에서도 설명했었지만 에피소드마다 학습하기 때문에 처음에 어떻게 했느냐에 따라 전혀 다른 experience를 가질 수가 있기 때문입니다.

- TD target $R_{t+1} + \gamma V(S_{t+1})$ is *biased* estimate of $v_\pi(S_t)$
- TD target is much lower variance than the return:
 - Return depends on *many* random actions, transitions, rewards
 - TD target depends on *one* random action, transition, reward

앞으로도 Bais와 Variance는 머리속에 기억해두어야 할 게 대부분의 강화학습을 발전시키려는 노력들이 근본적인 다른 알고리즘을 선택하는 것에도 있지만 그 알고리즘의 bais와 variance를 낮추려는 것에 집중된 경향이 있기 때문입니다.

TD Control

1. Sarsa

TD(0)의 알고리즘은 다음과 같습니다.

```

Input: the policy  $\pi$  to be evaluated
Initialize  $V(s)$  arbitrarily (e.g.,  $V(s) = 0, \forall s \in \mathcal{S}^+$ )
Repeat (for each episode):
    Initialize  $S$ 
    Repeat (for each step of episode):
         $A \leftarrow$  action given by  $\pi$  for  $S$ 
        Take action  $A$ ; observe reward,  $R$ , and next state,  $S'$ 
         $V(S) \leftarrow V(S) + \alpha[R + \gamma V(S') - V(S)]$ 
         $S \leftarrow S'$ 
    until  $S$  is terminal

```

Figure 6.1: Tabular TD(0) for estimating v_π .



Figure 6.2: The backup diagram for TD(0).

하지만 model-free control이 되기 위해서는 action-value function을 사용해야한다고 말했었습니다. 따라서 위 TD(0)의 식에서 value function을 action value function으로 바꾸어주면 Sarsa가 됩니다. Sarsa는 아래 backup diagram에서 따온 이름으로 아래 update식을 보면 현재 state-action pair에서 다음 state와 다음 action까지를 보고 update하기 때문에 붙은 이름입니다. TD(0)를 이해했다면 크게 어려운 점이 없는 부분입니다.

TD(0)의 알고리즘은 다음과 같습니다.

```

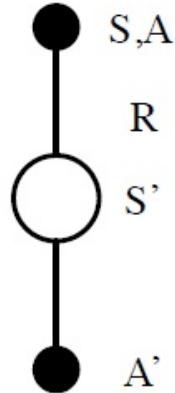
Input: the policy  $\pi$  to be evaluated
Initialize  $V(s)$  arbitrarily (e.g.,  $V(s) = 0, \forall s \in \mathcal{S}^+$ )
Repeat (for each episode):
    Initialize  $S$ 
    Repeat (for each step of episode):
         $A \leftarrow$  action given by  $\pi$  for  $S$ 
        Take action  $A$ ; observe reward,  $R$ , and next state,  $S'$ 
         $V(S) \leftarrow V(S) + \alpha[R + \gamma V(S') - V(S)]$ 
         $S \leftarrow S'$ 
    until  $S$  is terminal
  
```

Figure 6.1: Tabular TD(0) for estimating v_π .



Figure 6.2: The backup diagram for TD(0).

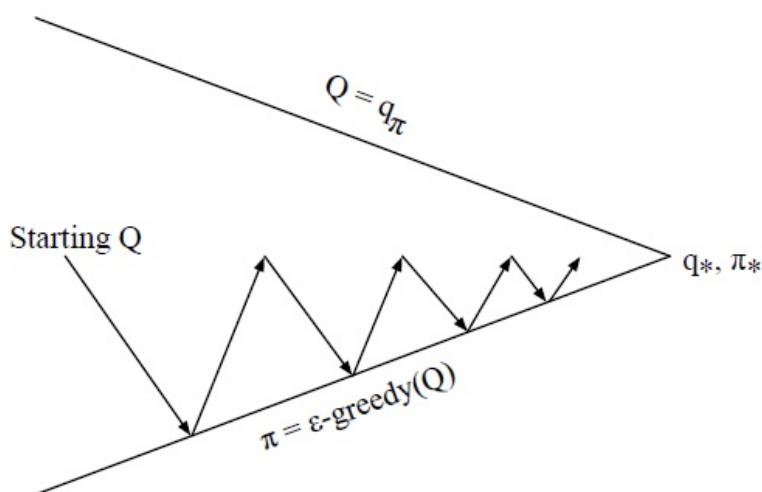
하지만 model-free control이 되기 위해서는 action-value function을 사용해야한다고 말했었습니다. 따라서 위 TD(0)의 식에서 value function을 action value function으로 바꾸어주면 Sarsa가 됩니다. Sarsa는 아래 backup diagram에서 따온 이름으로 아래 update식을 보면 현재 state-action pair에서 다음 state와 다음 action까지를 보고 update하기 때문에 붙은 이름입니다. TD(0)를 이해했다면 크게 어려운 점이 없는 부분입니다.



$$Q(S, A) \leftarrow Q(S, A) + \alpha (R + \gamma Q(S', A') - Q(S, A))$$

Sarsa는 따라서 TD(0)를 가지고 action-value function으로 바꾸고 ϵ -greedy policy improvement를 한 것 입니다.

On-Policy Control With Sarsa



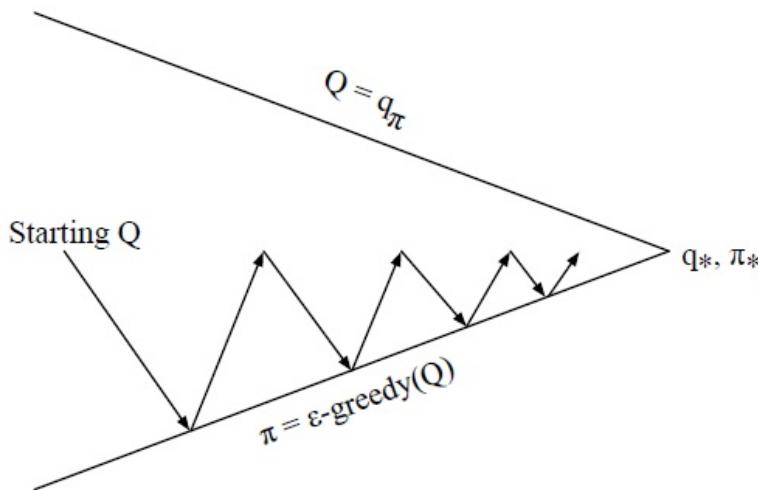
Every time-step:

Policy evaluation **Sarsa**, $Q \approx q_\pi$

Policy improvement ϵ -greedy policy improvement

Sarsa의 algorithm을 보면 다음과 같습니다. on-policy TD control algorithm으로서 매 time-step마다 현재의 Q value를 immediate reward와 다음 action의 Q value를 가지고 update합니다. policy는 따로 정의되지는 않고 이 Q value를 보고 ϵ -greedy하게 움직이는 것 자체가 policy입니다.

On-Policy Control With Sarsa



Every time-step:

Policy evaluation **Sarsa**, $Q \approx q_\pi$

Policy improvement ϵ -greedy policy improvement

Sarsa의 algorithm을 보면 다음과 같습니다. on-policy TD control algorithm으로서 매 time-step마다 현재의 Q value를 immediate reward와 다음 action의 Q value를 가지고 update합니다. policy는 따로 정의되지는 않고 이 Q value를 보고 ϵ -greedy하게 움직이는 것 자체가 policy입니다.

```

Initialize  $Q(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$ , arbitrarily, and  $Q(\text{terminal-state}, \cdot) = 0$ 
Repeat (for each episode):
    Initialize  $S$ 
    Choose  $A$  from  $S$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)
    Repeat (for each step of episode):
        Take action  $A$ , observe  $R, S'$ 
        Choose  $A'$  from  $S'$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)
         $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma Q(S', A') - Q(S, A)]$ 
         $S \leftarrow S'; A \leftarrow A'$ ;
    until  $S$  is terminal
  
```

Figure 6.9: Sarsa: An on-policy TD control algorithm.

Eligibility Traces

n-step TD나 TD(λ), eligibility trace는 최근 강화학습에서 많이 다루는 알고리즘들은 아닙니다. 하지만 최근 Asynchronous methods for deep reinforcement learning 논문에서 n-step을 사용한 일은 있었습니다. 하지만 제가 생각하기에 크게 중요한 부분은 아니고 "어떠한 문제점이 있어서 개선시키려고 했던 과정들이구나"라고 생각하면 편할 것 같습니다.

1. n-step TD

TD learning은 Monte-Carlo learning에 비해서 매 time-step마다 학습을 할 수 있다는 장점이 있습니다. 다시 한 번 TD Control인 Sarsa를 봄겠습니다. 아래의 update식을 보면 결국 update 한 번 할 때 R하나의 정보밖에 알 수 없어서 학습하는데 오래걸리기도 하고 bias가 높기 때문에 사실은 TD와 MC의 둘의 장점을 다 살릴 수 있다면 좋습니다.

```

Initialize  $Q(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$ , arbitrarily, and  $Q(\text{terminal-state}, \cdot) = 0$ 
Repeat (for each episode):
    Initialize  $S$ 
    Choose  $A$  from  $S$  using policy derived from  $Q$  (e.g.,  $\varepsilon$ -greedy)
    Repeat (for each step of episode):
        Take action  $A$ , observe  $R, S'$ 
        Choose  $A'$  from  $S'$  using policy derived from  $Q$  (e.g.,  $\varepsilon$ -greedy)
         $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma Q(S', A') - Q(S, A)]$ 
         $S \leftarrow S'; A \leftarrow A'$ ;
    until  $S$  is terminal

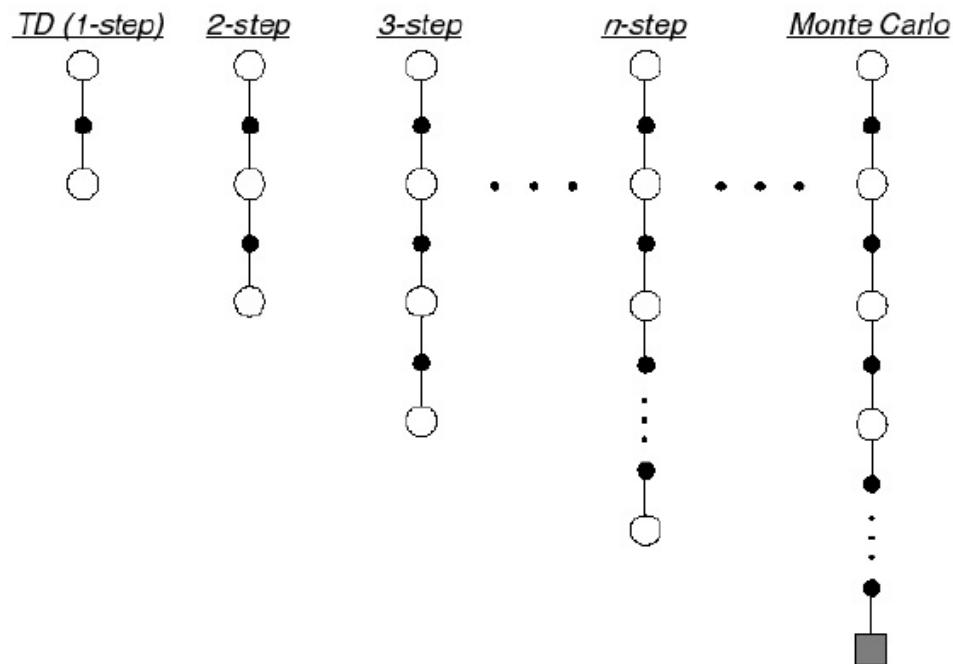
```

Figure 6.9: Sarsa: An on-policy TD control algorithm.

따라서 update를 할 때 one step만 보고서 update를 하는 것이 아니고 n-step을 움직인 다음에 update를 하자라고 생각하게 됩니다. 만약 지금이 t라고 하면 $t+n$ 이 되면 그 때까지 모았던 reward들로서 n-step return을 계산하고 그 사이의 방문했던 state들의 value function을 update하는 것입니다. 그것이 바로 n-step TD입니다. 몇 개의 time-step마다 update를 할 건지는 자신이 결정하면 됩니다. 이 n-step이 terminal state까지 간다면 그게 바로 Monte-Carlo이 됩니다. 따라서 둘의 장점을 다 취하기 위해서 그 사이의 적당한 n-step을 선택해주는 것이 좋습니다.

n-Step Prediction

- Let TD target look n steps into the future



n-Step Return

- Consider the following n -step returns for $n = 1, 2, \infty$:

$$\begin{aligned}
 n = 1 \quad (\text{TD}) \quad G_t^{(1)} &= R_{t+1} + \gamma V(S_{t+1}) \\
 n = 2 \quad &G_t^{(2)} = R_{t+1} + \gamma R_{t+2} + \gamma^2 V(S_{t+2}) \\
 &\vdots \qquad \vdots \\
 n = \infty \quad (\text{MC}) \quad G_t^{(\infty)} &= R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{T-1} R_T
 \end{aligned}$$

- Define the n -step return

$$G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{n-1} R_{t+n} + \gamma^n V(S_{t+n})$$

- n -step temporal-difference learning

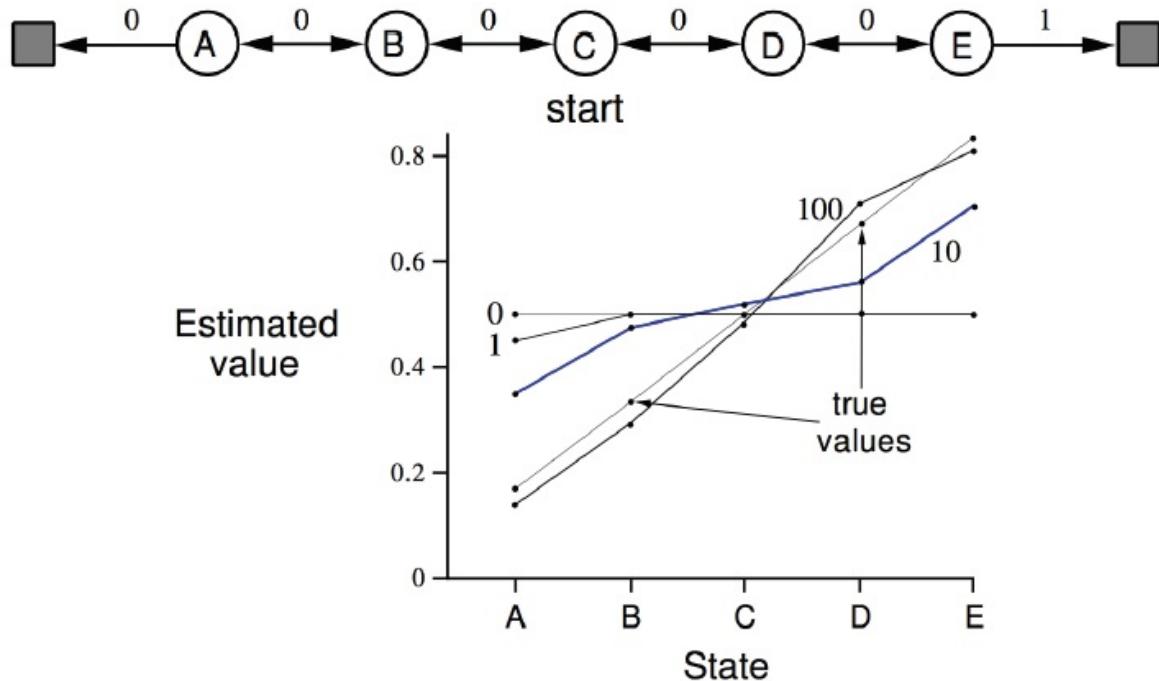
$$V(S_t) \leftarrow V(S_t) + \alpha (G_t^{(n)} - V(S_t))$$



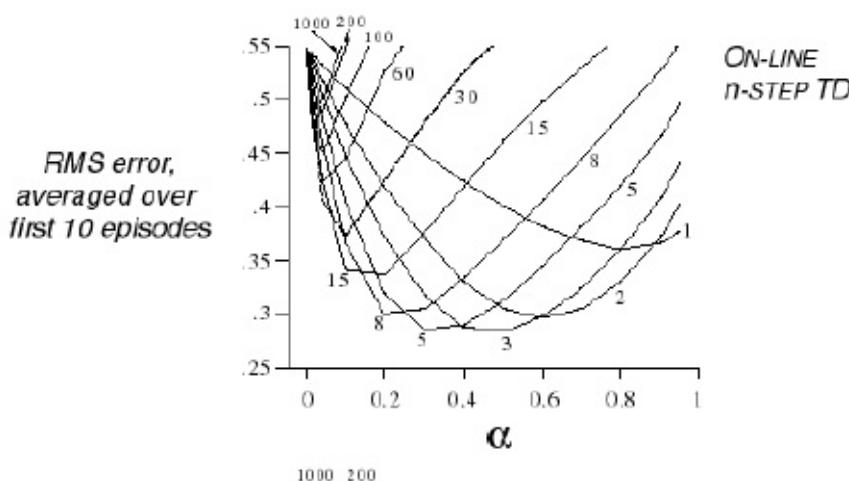
하지만 어떻게 적당한 n -step이라는 것을 알 수 있고 그 기준이 무엇일까요??

2. Forward-View of TD(λ)

Random walker라는 prediction example입니다. C state에서 시작해서 왼쪽 오른쪽으로 random하게 움직이는 policy를 가지고 있습니다. 이 문제에 대한 true value function 아래 그라프에 그려진 것처럼 되어있고 저 value function을 experience를 통해서 찾아내야합니다.

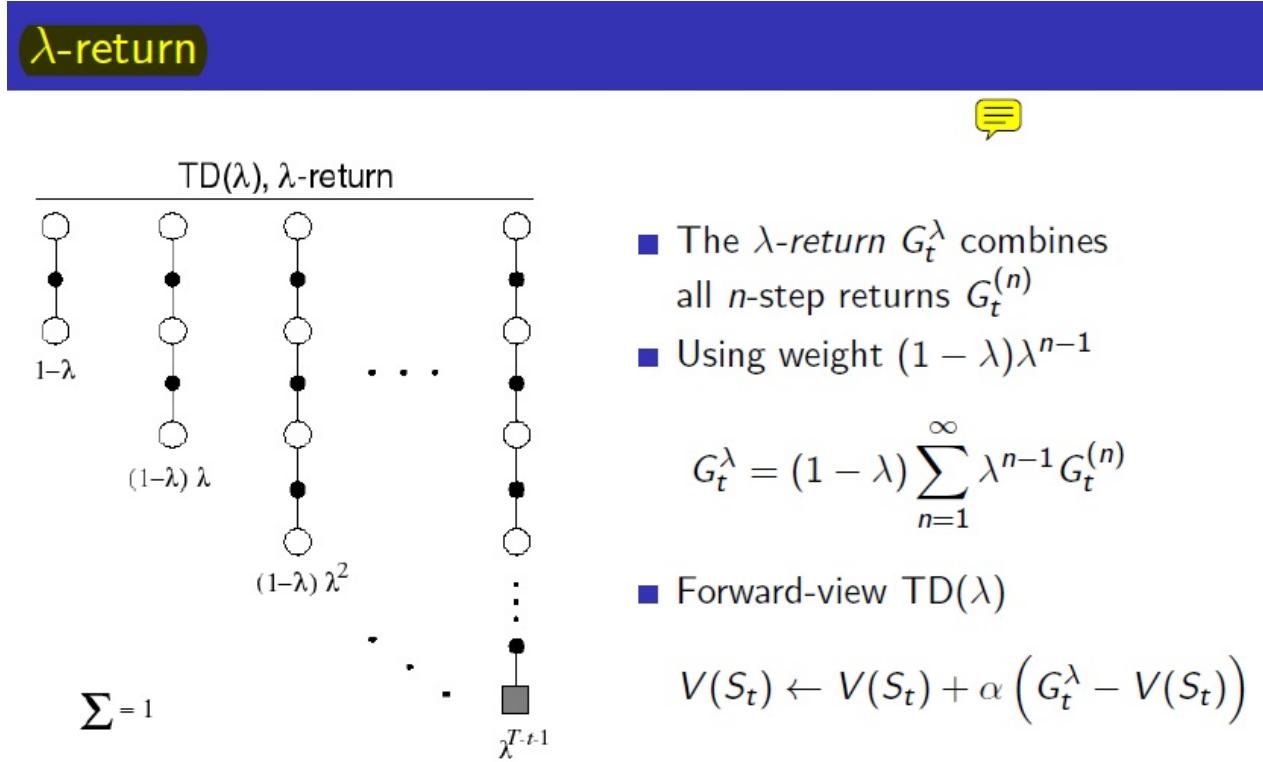


이 문제에 n-step TD prediction을 적용시켜볼 경우에 n 으로 적당한 숫자가 얼마인지 판별하기 쉽지 않습니다. 아래 그래프는 x축은 α , y축은 true value function과의 error입니다. 아래 그래프와 같이 α 의 값에 따라서 어떤 n-step이 학습에 좋은지는 달라지기 때문에 사실은 여러 n-step TD를 합할 수 있다면 각 n 의 장점을 다 취할 수 있을 것입니다.

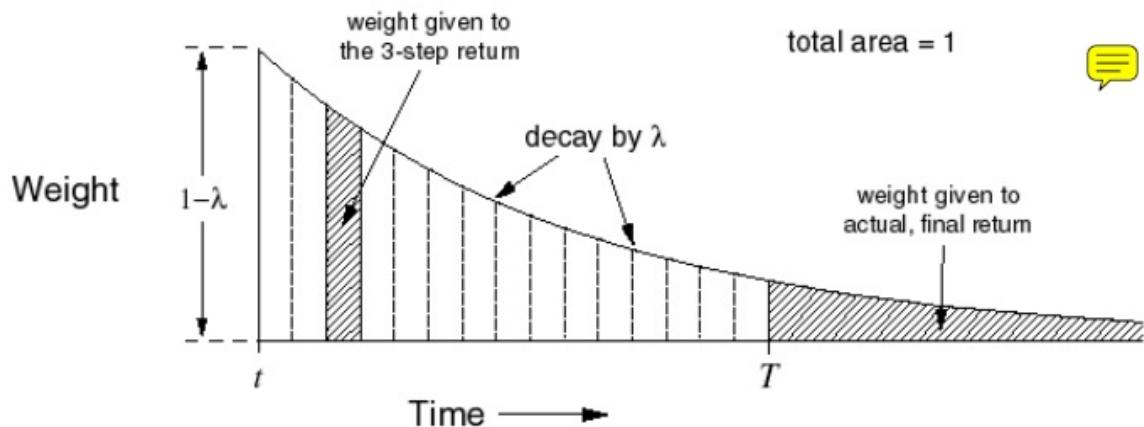


따라서 이 모든 n-step return을 모두 더해서 사용하기로 하였습니다. 하지만 단순히 더해서 평균을 구하는 방식이 아니고 아래와 같이 λ 라는 weight를 사용해서 geometrically weighted

sum을 이용합니다. 이렇게 하면 모든 n-step을 다 포함하면서 더하면 1이 나옵니다. 이것을 통해서 구한 λ -return을 원래 MC의 return 자리에 넣어주면 forward-view TD(λ)가 됩니다.

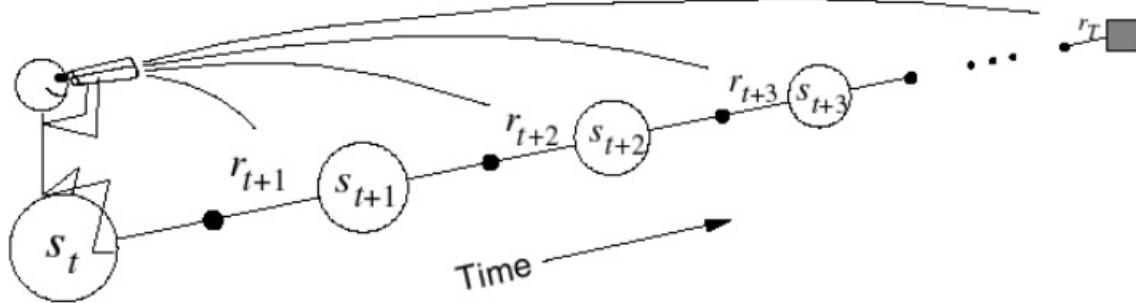


이것을 그림으로 나타내자면 다음과 같습니다.



$$G_t^\lambda = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_t^{(n)}$$

다시 정리를 해보자면 TD는 time-step마다 학습할 수 있는 장점은 있었지만 또한 bias가 높고 학습정보가 별로 없기 때문에 TD와 MC의 장점을 둘 다 살리기 위한 방법으로 n-step TD가 있었습니다. 하지만 각 n-step이 상황마다 다른 장점이 있어서 이 모든 장점을 포함하기 위해서 λ -return이라는 weight를 도입해서 λ -return을 계산해서 사용하는 것이 forward-view TD(λ -return)입니다. 하지만 이 방법에도 단점이 있습니다. 바로 MC와 똑같이 episode 까 끝나야 update를 할 수 있다는 것입니다.(모든 n-step을 포함하니까)



- Update value function towards the λ -return
- Forward-view looks into the future to compute G_t^λ
- Like MC, can only be computed from complete episodes

3. Backward-View of TD(λ)

따라서 본래 TD의 장점이었던 time-step마다 update할 수 있다는 장점이 사라졌습니다. MC의 장점은 살리면서도 바로 바로 update할 수 있는 방법이 없을까요? 여기서 바로 eligibility trace라는 개념이 나옵니다. 아래 그림과 같이 과거에 있었던 일들 중에서 현재 내가 받은 reward에 기여한 것이 무엇일까?라는 credit assignment 문제에서 "얼마나 최근에 일어났던 일이었나"와 "얼마나 자주 발생했었나"라는 것을 기준으로 과거의 일들을 기억해놓고 현재 받은 reward를 과거의 state들로 분배해주게 됩니다.

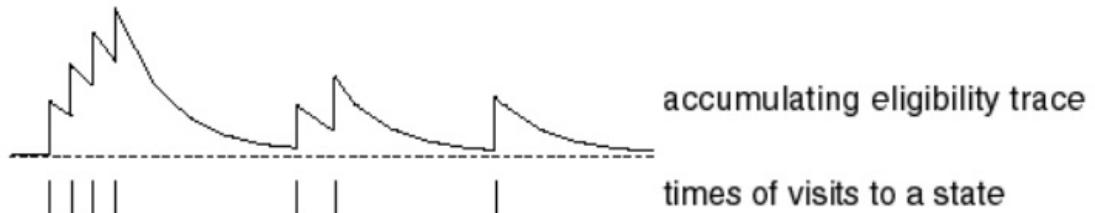
Eligibility Traces



- Credit assignment problem: did bell or light cause shock?
- **Frequency heuristic**: assign credit to most frequent states
- **Recency heuristic**: assign credit to most recent states
- Eligibility traces combine both heuristics

$$E_0(s) = 0$$

$$E_t(s) = \gamma \lambda E_{t-1}(s) + \mathbf{1}(S_t = s)$$



즉 TD(0)처럼 현재 update할 δ 를 계산하면 현재의 value function만 update하는 것이

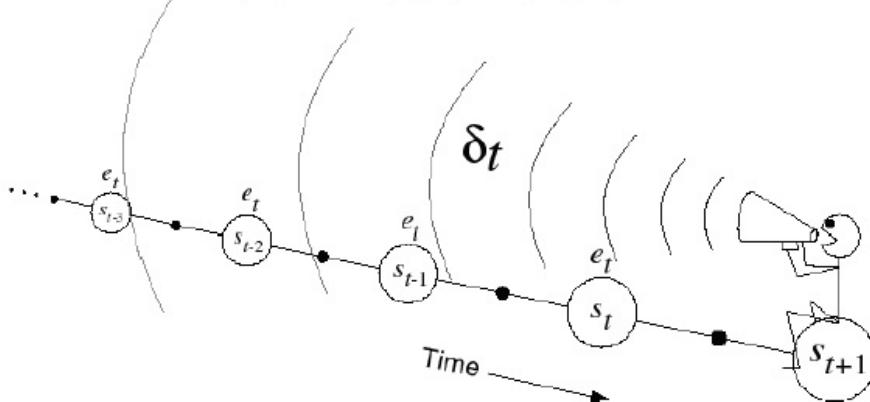
아니라 과거에 지나왔던 모든 state에 eligibility trace를 기억해두었다가 그 만큼 자신을 update하게 됩니다. 따라서 아래 그림과 같이 현재의 경험을 통해 한 번에 과거의 모든 state들의 value function을 update하게 되는 것입니다. 현재의 경험이 과거의 value function에 얼마나 영향을 주고 싶은가는 λ 를 통해 조절할 수 있습니다. 이러한 update 방식을 backward-view TD(λ)라고 합니다.

Backward View TD(λ)

- Keep an eligibility trace for every state s
- Update value $V(s)$ for every state s
- In proportion to TD-error δ_t and eligibility trace $E_t(s)$

$$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$$

$$V(s) \leftarrow V(s) + \alpha \delta_t E_t(s)$$



4. Sarsa(λ)

위에서는 TD prediction만 다루었습니다. 여기서는 TD Control에 대해서 다루도록 하겠습니다. Sarsa에도 n-step Sarsa가 있고 forward-view Sarsa(λ)가 있고 backward-view Sarsa(λ)가 있습니다. 각각은 다음과 같습니다. 설명은 생략하겠습니다.

n-Step Sarsa

- Consider the following n -step returns for $n = 1, 2, \infty$:

$$\begin{aligned}
 n = 1 & \quad (\text{Sarsa}) \quad q_t^{(1)} = R_{t+1} + \gamma Q(S_{t+1}) \\
 n = 2 & \quad q_t^{(2)} = R_{t+1} + \gamma R_{t+2} + \gamma^2 Q(S_{t+2}) \\
 & \vdots \quad \vdots \\
 n = \infty & \quad (\text{MC}) \quad q_t^{(\infty)} = R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{T-1} R_T
 \end{aligned}$$

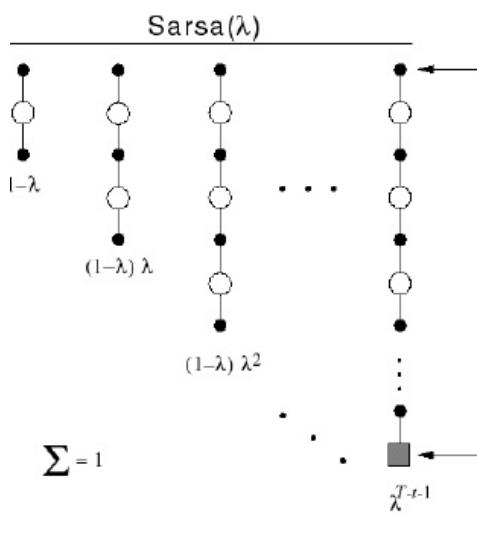
- Define the n -step Q-return

$$q_t^{(n)} = R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{n-1} R_{t+n} + \gamma^n Q(S_{t+n})$$

- n-step Sarsa updates $Q(s, a)$ towards the n -step Q-return

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left(q_t^{(n)} - Q(S_t, A_t) \right)$$

Forward View Sarsa(λ)



- The q^λ return combines all n -step Q-returns $q_t^{(n)}$
- Using weight $(1 - \lambda)\lambda^{n-1}$

$$q_t^\lambda = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} q_t^{(n)}$$

- Forward-view Sarsa(λ)

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left(q_t^\lambda - Q(S_t, A_t) \right)$$



Backward View Sarsa(λ)

- Just like TD(λ), we use eligibility traces in an online algorithm
- But Sarsa(λ) has one eligibility trace for each state-action pair

$$E_0(s, a) = 0$$

$$E_t(s, a) = \gamma \lambda E_{t-1}(s, a) + \mathbf{1}(S_t = s, A_t = a)$$

- $Q(s, a)$ is updated for every state s and action a
- In proportion to TD-error δ_t and eligibility trace $E_t(s, a)$

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

$$Q(s, a) \leftarrow Q(s, a) + \alpha \delta_t E_t(s, a)$$

backward-view Sarsa(\$\$\lambda\$\$)를 algorithm으로 표현하자면 아래과 같습니다.

Initialize $Q(s, a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$

Repeat (for each episode):

$$E(s, a) = 0, \text{ for all } s \in \mathcal{S}, a \in \mathcal{A}(s)$$

Initialize S, A

Repeat (for each step of episode):

Take action A , observe R, S'

Choose A' from S' using policy derived from Q (e.g., ε -greedy)

$$\delta \leftarrow R + \gamma Q(S', A') - Q(S, A)$$

$$E(S, A) \leftarrow E(S, A) + 1$$

For all $s \in \mathcal{S}, a \in \mathcal{A}(s)$:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \delta E(s, a)$$

$$E(s, a) \leftarrow \gamma \lambda E(s, a)$$

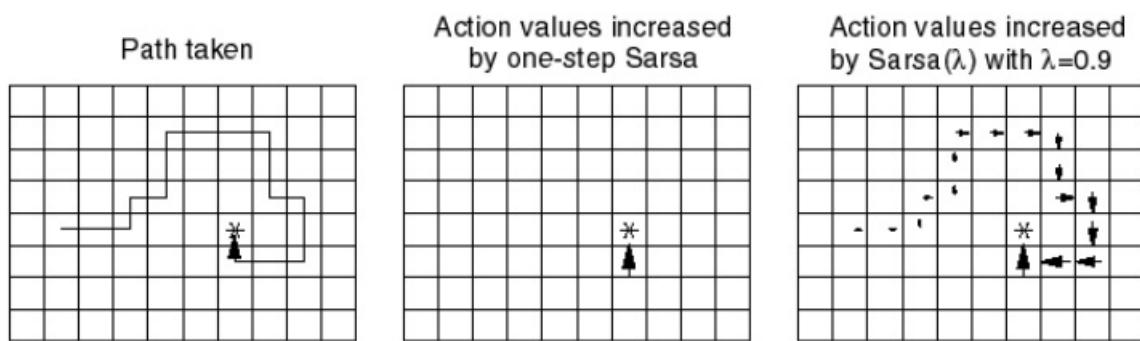
$$S \leftarrow S'; A \leftarrow A'$$

until S is terminal

또한 Eligibility trace를 이론으로만 봐서 이해가 안가는 분들이 계실텐데 아래 그림을 보면 이해가 쉽습니다. Path taken을 보면 지금까지 지나왔던 곳에 대해서 각각의 eligibility trace를 기억해놨다가 어느 지금에 이르러서 reward를 얻어서 update를 하려하면 자동으로 과거에 지나왔던 모든 state의 value function이 함께 update가 됩니다.

또한 Eligibility trace를 이론으로만 봐서 이해가 안가는 분들이 계실텐데 아래 그림을 보면 이해가 쉽습니다. Path taken을 보면 지금까지 지나왔던 곳에 대해서 각 각의 eligibility trace를 기억해놨다가 어느 지금에 이르러서 reward를 얻어서 update를 하려하면 자동으로 과거에 지나왔던 모든 state의 value function이 함께 update가 됩니다.

Sarsa(λ) Gridworld Example



Q learning

1. Importance Sampling

2. Q Learning

Importance Sampling

지금까지 Monte-Carlo Control과 Temporal-Difference Control 을 살펴보았습니다. 사실은 두 방법이 다 on-policy reinforcement learning입니다. 여기서 새로운 개념을 하나 알고 갈 필요가 있습니다.

1. On-Policy vs Off-Policy

다시 Sarsa의 알고리즘을 살펴보겠습니다. 아래와 같이 Sarsa에서는

- Choose A from S using Policy derived from Q
- Choose A' from S' using Policy derived from Q

action을 선택하는 것이 두 부분이 있습니다. 보면 둘 다 공통적으로 "using Policy derived from Q"가 사용된다는 것을 알 수 있습니다.

```

Initialize  $Q(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$ , arbitrarily, and  $Q(\text{terminal-state}, \cdot) = 0$ 
Repeat (for each episode):
    Initialize  $S$ 
    Choose  $A$  from  $S$  using policy derived from  $Q$  (e.g.,  $\varepsilon$ -greedy)
    Repeat (for each step of episode):
        Take action  $A$ , observe  $R, S'$ 
        Choose  $A'$  from  $S'$  using policy derived from  $Q$  (e.g.,  $\varepsilon$ -greedy)
         $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma Q(S', A') - Q(S, A)]$ 
         $S \leftarrow S'; A \leftarrow A'$ ;
    until  $S$  is terminal

```

Figure 6.9: Sarsa: An on-policy TD control algorithm.

이 말은 즉 현재 policy로 움직이면서 그 policy를 평가한다는 것입니다. 위 알고리즘에서 보면 에이전트가 실제로 움직인 Q function을 현재의 Q function을 업데이트합니다. 따라서 현재 policy위에서 control(prediction + policy improvement)을 하기 때문에 on-policy라고 생각해도 좋습니다.

하지만 on-policy 는 한계가 있습니다. 바로 탐험의 문제입니다. 현재 알고 있는 정보에 대해 greedy로 policy를 정해버리면 optimal에 가지 못 할 확률이 커지기 때문에 에이전트는 항상 탐험이 필요합니다. 따라서 on-policy처럼 움직이는 policy와 학습하는 policy가 같은 것이 아니고 이 두개의 policy를 분리시킨 것이 off-policy입니다. Silver는 수업에서 Off-policy를 다음과 같이 정의합니다.

Evaluate target policy $\pi(a|s)$ to compute $V(\pi(s))$ or $q(\pi(s,a))$ While following behaviour policy $\mu(s|a)$

- Evaluate target policy $\pi(a|s)$ to compute $v_\pi(s)$ or $q_\pi(s, a)$
- While following behaviour policy $\mu(a|s)$

$$\{S_1, A_1, R_2, \dots, S_T\} \sim \mu$$

- Why is this important?
- Learn from observing humans or other agents
- Re-use experience generated from old policies $\pi_1, \pi_2, \dots, \pi_{t-1}$
- Learn about *optimal* policy while following exploratory policy
- Learn about *multiple* policies while following *one* policy

Off-policy는 다음과 같은 장점이 있습니다.

- 다른 agent나 사람을 관찰하고 그로부터 학습할 수 있다
- 이전의 policy들을 재활용하여 학습할 수 있다.
- 탐험을 계속 하면서도 optimal한 policy를 학습할 수 있다.(Q-learning)
- 하나의 policy를 따르면서 여러개의 policy를 학습할 수 있다.

2. Importance sampling

위에서 Off-policy learning이 어떤 것인지 배웠습니다. 하지만 다른 policy로부터 현재 policy를 학습할 수 있다는 근거가 무엇일까요? "importance sampling"이라는 개념은 원래 통계학에서 사용하던 개념으로 아래와 특정한 분포의 값들을 추정하는 기법중의 하나입니다.

https://en.wikipedia.org/wiki/Importance_sampling

In statistics, importance sampling is a general technique for estimating properties of a particular distribution, while only having samples generated from a different distribution than the distribution of interest

어떤 값을 추정하는데 가장 기본적인 방법은 그냥 random하게 찍어보는 것입니다. 이미 저희가 배웠다시피 이러한 process 표현하는 말은 "monte-carlo"로서 Monte-Carlo estimation이라고 합니다. 하지만 너무 광범위하게 탐색하기도 하고 어떠한 중요한 부분을 알아서 그 위주로 탐색을 하면 더 빠르고 효율적으로 값을 추정할 수 있고 그러한 아이디어가 바로 "Importance Sampling"입니다.

다. 아래는 Importance Sampling에 대해서 설명해놓은 강의입니다. 원래 통계학의 개념이기 때문에 저에게는 생소해서 더 여려웠던 부분인 것 같습니다. <https://www.youtube.com/watch?v=S3LAOZxGcnk>

(ML 17.5) Importance sampling - introduction

Importance Sampling approx. $Ef(x) \approx \frac{1}{n} \sum_{i=1}^n f(x_i)$

(p density) $x_i \sim q$ $X \sim p, X_i \sim p$ q

$$Ef(x) = \int f(x) p(x) dx = \left(\int \left(f(x) \frac{p(x)}{q(x)} \right) q(x) dx \right) \approx \frac{1}{n} \sum_{i=1}^n f(x_i) \frac{p(x_i)}{q(x_i)}$$

$\forall q$ (pdf) s.t. $q(x) = 0 \Rightarrow p(x) = 0$

$Ef(x) \approx \frac{1}{n} \sum_{i=1}^n f(x_i) \omega(x_i)$ where $x_i \sim q$

"Importance weight" $\omega(x) = \frac{p(x)}{q(x)}$

▶ ▶ ⏴ 7:03 / 13:42

P와 q라는 다른 distribution이 있을 때 q라는 distribution에서 실제로 진행을 함에도 p로 추정하는 것처럼 할 수 있다는 것입니다. 강화학습에서도 policy가 다르면 state의 distribution은 달라지게 되어 있습니다. 따라서 다른 distribution을 통해 추정할 수 있다는 개념을 그대로 가져와서 다른 policy를 통해서 얻어진 sample을 이용하여 Q 값을 추정할 수 있다는 것입니다. 일종의 trick이라고 할 수 있을 것 같습니다.

위의 내용을 David Silver교수님은 아래와 같이 설명하십니다. $f(X)$ 라는 함수를 value function이라고 생각하고 강화학습에서는 이 value function = expected future reward를 계속 추정해나가는 데 $P(X)$ 라는 현재 policy로 형성된 distribution으로부터 학습을 하고 있었습니다. 하지만 다른 Q라는 distribution을 따르면서도 똑같이 학습할 수 있는데 단, 아래와 같이 간단히 식을 변형시켜주면 됩니다. $Q(X)$ 를 곱해주고 나눠주면 됩니다.

Importance Sampling

- Estimate the expectation of a different distribution

$$\begin{aligned}\mathbb{E}_{X \sim P}[f(X)] &= \sum P(X)f(X) \\ &= \sum Q(X) \frac{P(X)}{Q(X)} f(X) \\ &= \mathbb{E}_{X \sim Q} \left[\frac{P(X)}{Q(X)} f(X) \right]\end{aligned}$$

Off-Policy 또한 MC와 TD로 갈립니다. Off-policy MC는 아래와 같습니다. 에피소드가 끝나고 return을 계산할 때 아래와 같이 식을 변형시켜줍니다. 각 스텝에 reward를 받게 된 것은 μ 라는 policy를 따라서 얻었던 것이므로 매 step마다 $\pi(A_t | S_t)$ 를 해줘야합니다. 따라서 Monte-Carlo에 Off-policy를 적용시키는 것은 그리 좋은 아이디어가 아닙니다.

Importance Sampling for Off-Policy Monte-Carlo

- Use returns generated from μ to evaluate π
- Weight return G_t according to similarity between policies
- Multiply importance sampling corrections along whole episode

$$G_t^{\pi/\mu} = \frac{\pi(A_t | S_t)}{\mu(A_t | S_t)} \frac{\pi(A_{t+1} | S_{t+1})}{\mu(A_{t+1} | S_{t+1})} \dots \frac{\pi(A_T | S_T)}{\mu(A_T | S_T)} G_t$$

- Update value towards *corrected* return

$$V(S_t) \leftarrow V(S_t) + \alpha \left(G_t^{\pi/\mu} - V(S_t) \right)$$

- Cannot use if μ is zero when π is non-zero
- Importance sampling can dramatically increase variance

Off-Policy TD에서는 MC 때와는 달리 Importance Sampling을 1-step만 진행하면 됩니다.

MC때와 비교하면 Variance가 낮아지기는 했지만 여전히 원래 TD에 비하면 Importance sampling때문에 높은 variance를 가지고 있습니다. Off-policy learning을 할 때 Importance sampling말고 다른 방법을 생각할 필요가 있습니다. 바로 여기서 유명한 Q learning알고리즘이 나오게 됩니다.

Importance Sampling for Off-Policy TD

- Use TD targets generated from μ to evaluate π
- Weight TD target $R + \gamma V(S')$ by importance sampling
- Only need a single importance sampling correction

$$V(S_t) \leftarrow V(S_t) + \alpha \left(\frac{\pi(A_t|S_t)}{\mu(A_t|S_t)} (R_{t+1} + \gamma V(S_{t+1})) - V(S_t) \right)$$

- Much lower variance than Monte-Carlo importance sampling
- Policies only need to be similar over a single step

Q Learning

1. Q-Learning

Off-Policy Learning 알고리즘 중에서 Off-policy MC와 Off-policy TD가 있지만 Importance sampling 문제 때문에 새로운 방법이 필요하다고 말했었습니다. Off-Policy learning을 하는데 가장 좋은 알고리즘은 Q Learning입니다.

방법은 다음과 같습니다. 현재 state S 에서 action을 선택하는 것은 behaviour policy를 따라서 선택합니다. TD에서 update할 때는 one-step을 bootstrap하는데 이 때 다음 state의 action을 선택하는 데는 behaviour policy와는 다른 policy(alternative policy)를 사용하면 Importance Sampling이 필요하지 않습니다. 이전의 Off-Policy에서는 Value function을 사용했었는데 여기서는 action-value function을 사용함으로서 다음 action까지 선택을 해야하는데 그 때 다른 policy를 사용한다는 것입니다.

Q-Learning

- We now consider off-policy learning of action-values $Q(s, a)$
- No importance sampling is required
- Next action is chosen using behaviour policy $A_{t+1} \sim \mu(\cdot | S_t)$
- But we consider alternative successor action $A' \sim \pi(\cdot | S_t)$
- And update $Q(S_t, A_t)$ towards value of alternative action

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha (R_{t+1} + \gamma Q(S_{t+1}, A') - Q(S_t, A_t))$$

2. Off-Policy Control with Q-Learning

이 Q learning 알고리즘 중에서 가장 유명한 것이 아래입니다.

- Behaviour policy로는 ϵ -greedy w.r.t. $Q(s, a)$
- Target policy(alternative policy)로는 greedy w.r.t. $Q(s, a)$

를 택한 알고리즘입니다. 이전에 Off-policy의 장점이 exploratory policy를 따르면서도 optimal policy를 학습할 수 있다고 했는데 그게 바로 이 알고리즘입니다. greedy한 policy로 학습을 진행하면 수렴을 빨리 하는데 충분히 탐험을 하지 않았기 때문에 local에 빠지기 쉽습니다. 그래서 탐험을 위해서 ϵ -greedy policy를 사용하면 탐험을 계속하는데 이렇게 학습하면 수렴속도가 느려져서 학습속도가 느려지게 됩니다. 이를 해결하기 위한 방법이 ϵ -decay를 시간에 따라 decay시키는 방법과 아래와 같이 Q learning을 사용하는 것입니다.

Off-Policy Control with Q-Learning

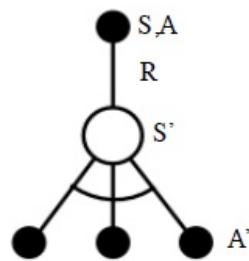
- We now allow both behaviour and target policies to **improve**
- The target policy π is **greedy** w.r.t. $Q(s, a)$

$$\pi(S_{t+1}) = \operatorname{argmax}_{a'} Q(S_{t+1}, a')$$

- The behaviour policy μ is e.g. **ϵ -greedy** w.r.t. $Q(s, a)$
- The Q-learning target then simplifies:

$$\begin{aligned} & R_{t+1} + \gamma Q(S_{t+1}, A') \\ &= R_{t+1} + \gamma Q(S_{t+1}, \operatorname{argmax}_{a'} Q(S_{t+1}, a')) \\ &= R_{t+1} + \max_{a'} \gamma Q(S_{t+1}, a') \end{aligned}$$

Q-Learning Control Algorithm



$$Q(S, A) \leftarrow Q(S, A) + \alpha \left(R + \gamma \max_{a'} Q(S', a') - Q(S, A) \right)$$

Theorem

Q-learning control converges to the optimal action-value function,
 $Q(s, a) \rightarrow q_*(s, a)$

알고리즘은 아래와 같습니다.

Q-Learning Algorithm for Off-Policy Control

```

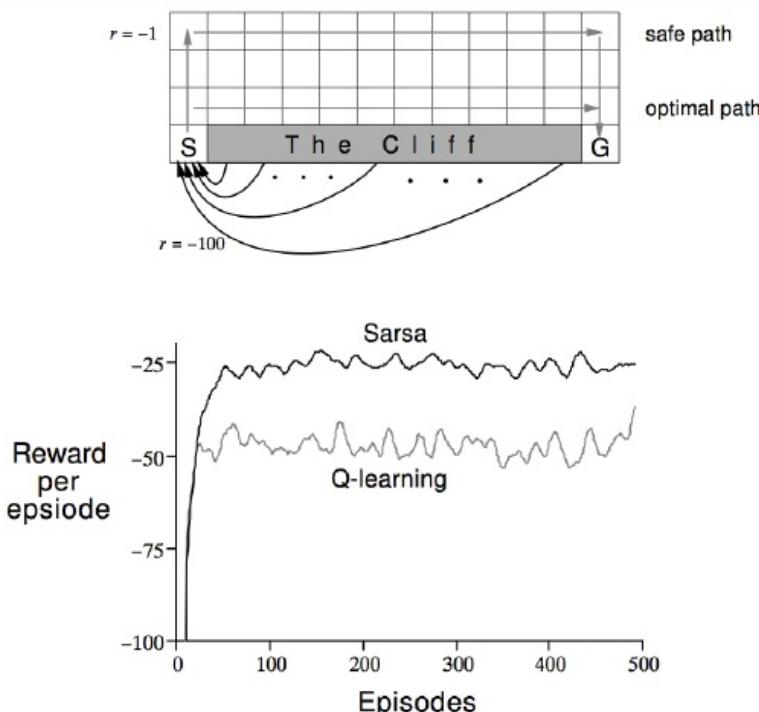
Initialize  $Q(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$ , arbitrarily, and  $Q(\text{terminal-state}, \cdot) = 0$ 
Repeat (for each episode):
    Initialize  $S$ 
    Repeat (for each step of episode):
        Choose  $A$  from  $S$  using policy derived from  $Q$  (e.g.,  $\varepsilon$ -greedy)
        Take action  $A$ , observe  $R, S'$ 
         $Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$ 
         $S \leftarrow S'$ ;
    until  $S$  is terminal
  
```

3.Sarsa vs Q-learning

3. Sarsa vs Q-learning

이렇게 Q-learning에 대해서 살펴봐도 직관적으로 Q-learning이 어떤 방식으로 작동하는지 잘 와닿지 않을 것입니다. Q-learning을 이해하려면 SARSA와 비교해보는 것이 좋습니다. Sutton이 이 두 가지를 비교할 수 있는 예제를 제시했습니다."Cliff Walking"이라는 예제입니다.

Cliff Walking Example



이 예제에서 목표는 S라는 start state에서 시작해서 Goal까지 가는 optimal path를 찾는 것입니다. 그림에 나와있는 Cliff에 빠져버리면 -100의 reward를 받고 time-step마다 reward를 -1씩 받는 문제라서 절벽에 빠지지 않고 goal까지 가능한 한 빠르게 도착하는 것이 목표입니다.

눈으로 딱 봐도 그림에 있는 optimal path가 답이라고 생각합니다. SARSA와 Q-learning 모두 다른 ϵ -greedy한 policy로 움직입니다. 따라서 더러는 Cliff에 빠져버리기도 합니다. 차이는 SARSA는 on-policy라서 그렇게 Cliff에 빠져버리는 결과로 인해 그 주변의 상태들의 value를 낮다고 판단합니다. 하지만 Q-learning의 경우에는 비록 ϵ -greedy로 인해 Cliff에 빠져버릴지라도 자신이 직접 체험한 그 결과가 아니라 greedy한 policy로 인한 Q function을 이용해서 업데이트합니다. 따라서 Cliff 근처의 길도 Q-learning은 optimal path라고 판단할 수 있어서 이 문제의 경우 SARSA보다는 Q-learning이 적합하다고 할 수 있습니다.

SARSA에서 탐험을 위해서 ϵ -greedy를 사용했지만 결국은 그로인해서 정작 에이전트가 optimal로 수렴하지 못하는 현상들이 발생한 것입니다. 따라서 Q-learning의 등장 이후로는 많은 문제에서 Q-learning이 더 효율적으로 문제를 풀었기 때문에 강화학습에서 Q-learning은 기

Value Function Approximation

1. Value Function Approximation

2. Stochastic Gradient Descent

3. Learning with Function Approximator

Value function Approximation

1. Tabular Methods

지금까지 살펴본 강화학습은 action value function을 Table로 만들어서 푸는 Tabular Methods입니다. 이에 대해서 Sutton교수님은 책에서 다음과 같이 이야기합니다. 즉, 현재의 방법은 state나 action이 작을 경우에만 적용 가능하다는 것입니다. 이 Table이 점점 더 커지면 이 값들을 다 기억 할 메모리도 문제지만 학습에 너무 많은 시간이 소요되기 때문에 사실상 학습이 불가능합니다. 앞에서 다뤘던 예제들도 다 gridworld같이 작은 예제였다는 것을 알 수 있습니다.

We have so far assumed that our estimates of value functions are represented as a table with one entry for each state or for each state{action pair. This is a particularly clear and instructive case, but of course it is limited to tasks with small numbers of states and actions. The problem is not just the memory needed for large tables, but the time and data needed to fill them accurately. In other words, the key issue is that of generalization

즉 아래와 같은 문제는 어떻게 풀거냐는 것입니다. 현재의 방식은 실용적이지 못하기 때문에 "Generalization"이 가능해지려면 새로운 idea가 필요합니다. 특히나 강화학습을 실제 세상에 적용시키고 싶다면 실제 세상은 continuous state space이므로 사실상 state가 무한대이기 때문에 새로운 방법이 있다면 로봇이 강화학습으로 학습하기는 어려울 것입니다.

Large-Scale Reinforcement Learning

Reinforcement learning can be used to solve *large* problems, e.g.

- Backgammon: 10^{20} states
- Computer Go: 10^{170} states
- Helicopter: continuous state space

How can we scale up the model-free methods for *prediction* and *control* from the last two lectures?

2. Parameterizing value function

이에 대한 해답은 아래와 같습니다. table로 작성하는 것이 아니고 w 라는 새로운 변수를 사용해서 value function을 함수화하는 것입니다.

- Solution for large MDPs:

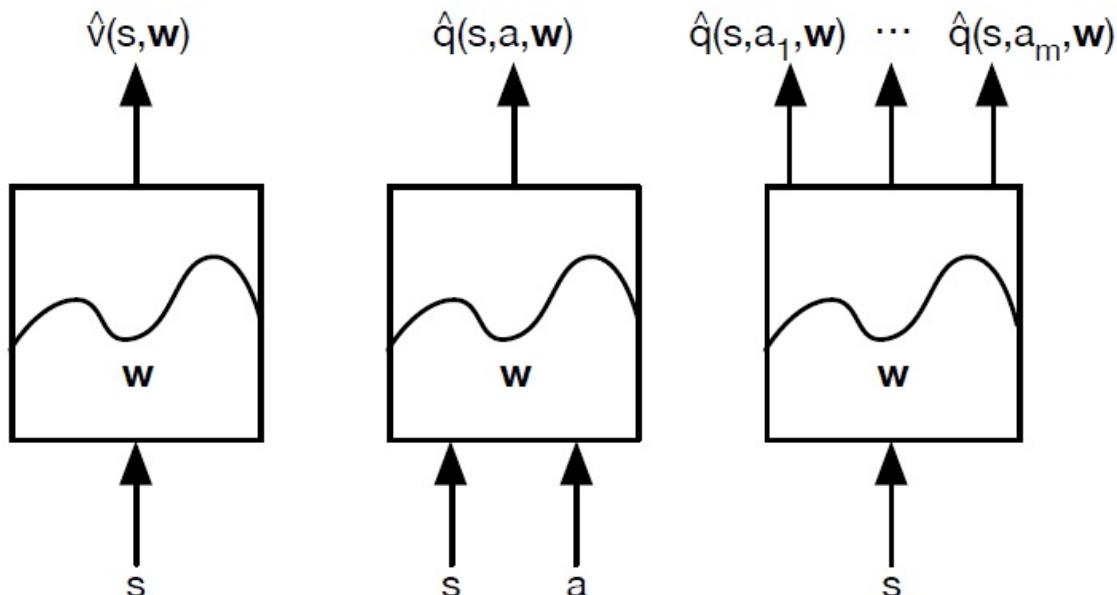
- Estimate value function with *function approximation*

$$\hat{v}(s, w) \approx v_\pi(s)$$

or $\hat{q}(s, a, w) \approx q_\pi(s, a)$

- Generalise from seen states to unseen states
- Update parameter w using MC or TD learning

그림으로 표현하자면 아래와 같습니다. state가 함수의 input으로 들어가면 w 라는 parameter로 조정되는 함수가 action value function을 output으로 내보내는 것입니다.



예를 들면 다음과 같습니다. $y=x^2$ 라는 함수가 있을 때 $(1,1), (2,4), (3,9)$ 라는 식으로 점을 다 찍어서 이 함수를 표현할 수도 있고 $y=ax^2+bx+c$ 에서 $a=1, b=c=0$ 이라고 표현할 수도 있을 것입니다. 후자가 더 효율적으로 표현할 수 있고 사실 점으로 찍지 않은 곳에 대해서는 아무 정보도 알 수 없는데 함수의 형태로 표현하면 모든 state에 대한 value function을 알 수 있습니다.

이제는 학습을 통해서 Q function을 update하는 것이 아니고 w라는 parameter를 업데이트하게 됩니다. 이러한 function approximation 방법에는 여러가지가 있습니다.

We consider **differentiable** function approximators, e.g.

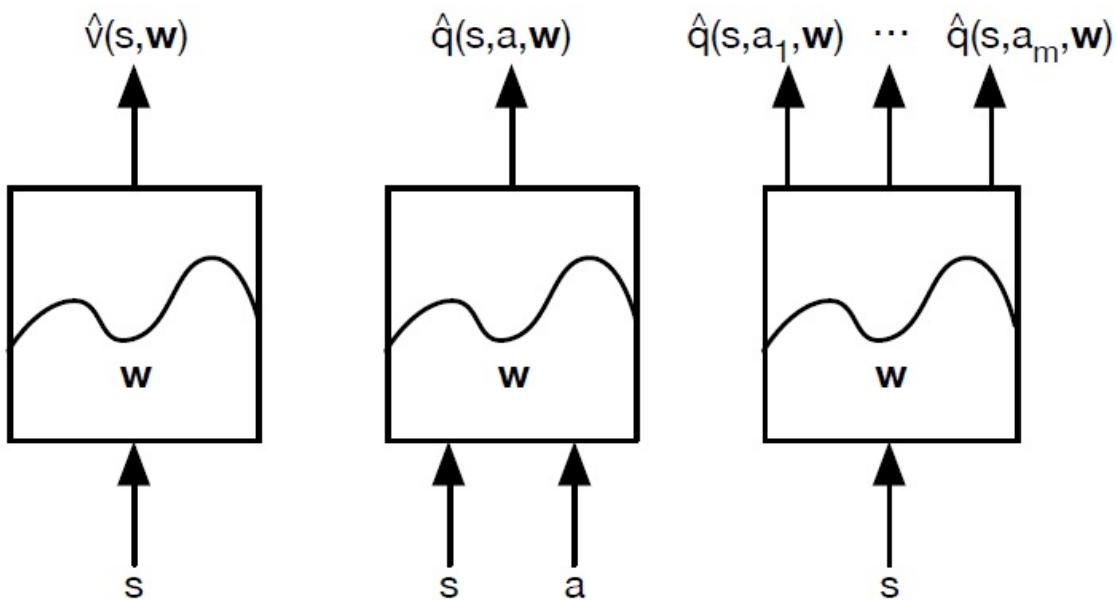
- Linear combinations of features
- Neural network
- Decision tree
- Nearest neighbour
- Fourier / wavelet bases
- ...

이 중에서 저희는 위의 두 가지를 볼텐데 (1) Linear combinations of features (2) Neural network 를 볼 것입니다. (2)은 특히 최근에 딥러닝의 발전으로 각광받는 방법입니다. 최근의 강화학습은 다 딥러닝을 approximator로 사용하기 때문에 보통 deep reinforcement learning이라고 부릅니다.

Stochastic Gradient Descent

1. Gradient Descent

이전까지 다루었던 Tabular Reinforcement learning의 단점 때문에 function approximator를 도입하게 되었고 value function을 w 라는 parameter를 통해서 approximate하였습니다. 또한 이제 학습이라는 것은 이 parameter를 update하는 것이라고 했었습니다.



그렇다면 parameter를 update를 하는 것은 어떻게 할 수 있을까요? 이전에 machine learning에 대해서 접해본 분이면 잘 아는 Stochastic Gradient Descent 방법을 활용하여 value function의 parameter를 update하게 됩니다.

이 방법은 간단하면서도 간단하기 때문에 프로그램 상으로 강력한 update 방법입니다. 하지만 시작 점에서 조금만 달라져도 다른 극점에 도착할 수 있으며 도착한 극점은 global optimum이 아닐 수도 있습니다. 이러한 단점을 극복하는 방법은 여러가지가 있지만 처음 parameter를 update하는 것을 배우는 입장에서는 간단한 개념을 알고 나중에 활용할 때 그러한 기법들을 도입하면 될 것 같습니다.

개념은 다음과 같습니다. w 로 표현하는 함수, 여기서는 $J(w)$ 로 표현한 함수는 어떠한 update의 목표로서 보통은 내가 원하는 대상과 자신의 error로 설정해서 그 error를 최소화하는 것을 목표로 합니다. update를 하려면 어느 방향으로 가야 그 error가 줄어드는지 알아야 하는데 그것을 함수의 미분(gradient)을 취해서 알 수 있습니다. gradient 자체는 경사이기 때문에 곡면에서 보자면 위로

올라가는 방향이므로 -를 곱해서 그 반대 방향으로 내려감으로서(descent) 조금씩 error를 줄여나가는 것입니다.

Gradient Descent

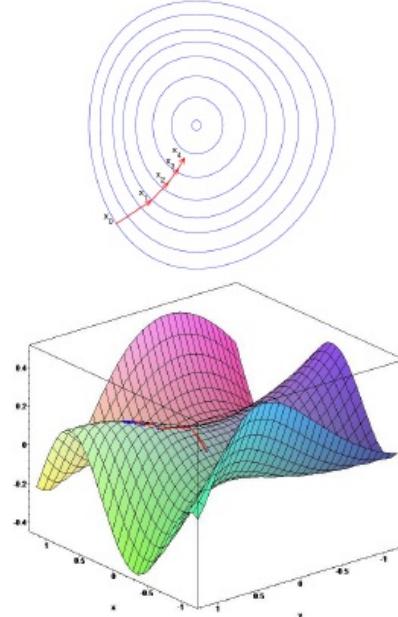
- Let $J(\mathbf{w})$ be a differentiable function of parameter vector \mathbf{w}
- Define the *gradient* of $J(\mathbf{w})$ to be

$$\nabla_{\mathbf{w}} J(\mathbf{w}) = \begin{pmatrix} \frac{\partial J(\mathbf{w})}{\partial w_1} \\ \vdots \\ \frac{\partial J(\mathbf{w})}{\partial w_n} \end{pmatrix}$$

- To find a local minimum of $J(\mathbf{w})$
- Adjust \mathbf{w} in direction of -ve gradient

$$\Delta \mathbf{w} = -\frac{1}{2} \alpha \nabla_{\mathbf{w}} J(\mathbf{w})$$

where α is a step-size parameter

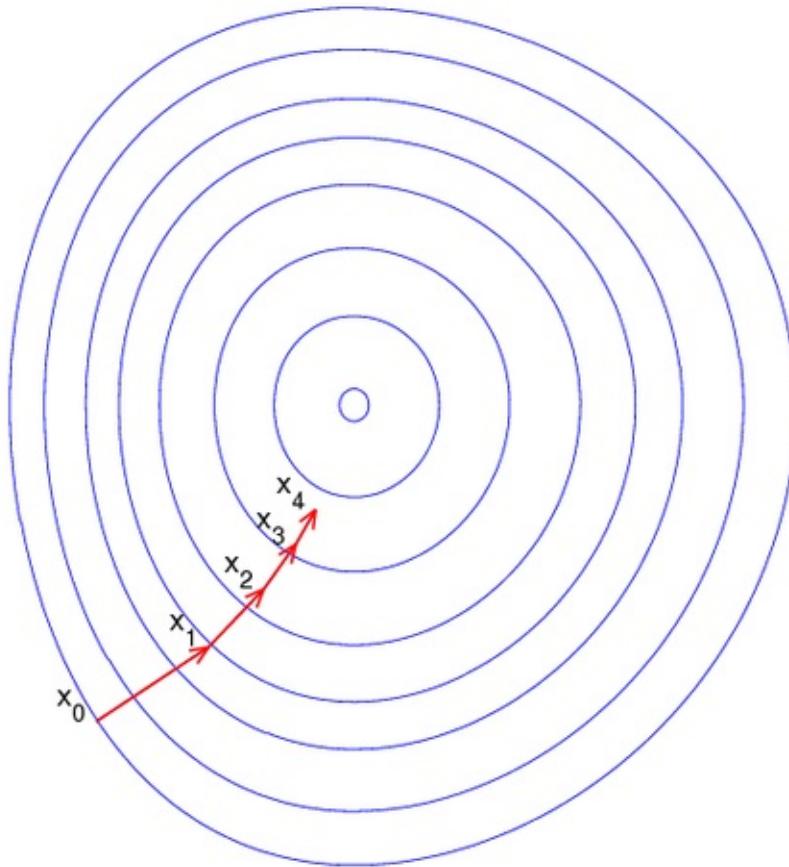


2. Example

gradient descent에 대한 위키페이지는 다음과 같습니다.

https://ko.wikipedia.org/wiki/%EA%B2%BD%EC%82%AC_%ED%95%98%EA%B0%95%EB%B2%95

아래의 그림을 어떠한 함수의 parameter space라고 하면 \mathbf{x}_0 부터 시작해서 함수의 gradient를 따라서 점진적으로 가운데의 극점으로 내려가게 됩니다. 언덕에서 공을 굴려서 아래로 떨어지고 있다고 생각하면 자연스로 경사를 따라서 내려가는 형태와 비슷합니다.



실제로 어떤 함수의 극점을 구하는 것을 방정식의 해를 구하는 방법이 아닌 numerical하게 풀어보는 예는 아래와 같습니다. 간단히 Stochastic gradient descent의 방법을 보여주는 코드도 있습니다.

이하의 파이썬 언어로 작상한 경사 하강법 알고리즘은 $f(x)=x^4-3x^3+2$ 함수의 극값을 미분값인 $f'(x)=4x^3-9x^2$ 를 통해 찾는 예를 보여준다.^[2]

```
# From calculation, we expect that the local minimum occurs at x=9/4
```

```
x_old = 0
x_new = 6 # The algorithm starts at x=6
eps = 0.01 # step size
precision = 0.00001
```

```
def f_prime(x):
    return 4 * x**3 - 9 * x**2

while abs(x_new - x_old) > precision:
    x_old = x_new
    x_new = x_old - eps * f_prime(x_old)
print "Local minimum occurs at ", x_new
```

Gradient Descent on RL

Gradient의 개념을 살펴보았습니다. 이 개념을 강화학습에 적용시켜보도록 하겠습니다. 강화학습에서는 $J(w)$ 를 어떻게 정의할까요? 바로 true value function과 approximate value $\hat{v}(s, w)$ (s, w)와의 error로 잡습니다.

Value Function Approx. By Stochastic Gradient Descent

- Goal: find parameter vector w minimising mean-squared error between approximate value fn $\hat{v}(s, w)$ and true value fn $v_\pi(s)$

$$J(w) = \mathbb{E}_\pi [(v_\pi(S) - \hat{v}(S, w))^2]$$

- Gradient descent finds a local minimum

$$\begin{aligned}\Delta w &= -\frac{1}{2} \alpha \nabla_w J(w) \\ &= \alpha \mathbb{E}_\pi [(v_\pi(S) - \hat{v}(S, w)) \nabla_w \hat{v}(S, w)]\end{aligned}$$

- Stochastic gradient descent *samples* the gradient

$$\Delta w = \alpha (v_\pi(S) - \hat{v}(S, w)) \nabla_w \hat{v}(S, w)$$

- Expected update is equal to full gradient update

Gradient Descent 방법도 (1) Stochastic Gradient Descent(SGD)와 (2) Batch 방법으로 나눌 수 있는데 위와 같이 모든 state에서 true value function과의 error를 한 번에 함수로 잡아서 업데이트하는 방식은 Batch의 방식을 활용한 것으로서 step by step으로 업데이트하는 것이 아니고 한꺼번에 업데이트하는 것입니다. Mean-squared error를 gradient방식에 집어넣어서 gradient를 취해 보면 아래와 같습니다.

- Gradient descent finds a local minimum

$$\begin{aligned}\Delta w &= -\frac{1}{2} \alpha \nabla_w J(w) \\ &= \alpha \mathbb{E}_\pi [(v_\pi(S) - \hat{v}(S, w)) \nabla_w \hat{v}(S, w)]\end{aligned}$$

하지만 DP에서 강화학습으로 넘어갈 때처럼 expectation을 없애고 sampling으로 대체하면 아래와 같아집니다.

- Stochastic gradient descent *samples* the gradient

$$\Delta \mathbf{w} = \alpha(v_\pi(S) - \hat{v}(S, \mathbf{w})) \nabla_{\mathbf{w}} \hat{v}(S, \mathbf{w})$$

이전에 MC와 TD Learning에서 했듯이 True value function 부분을 여러가지로 대체할 수 있습니다. Sample Return을 사용할 수도 있고 TD target을 사용할 수도 있습니다.

- Have assumed true value function $v_\pi(s)$ given by supervisor
- But in RL there is no supervisor, only rewards
- In practice, we substitute a *target* for $v_\pi(s)$
 - For MC, the target is the return G_t

$$\Delta \mathbf{w} = \alpha(G_t - \hat{v}(S_t, \mathbf{w})) \nabla_{\mathbf{w}} \hat{v}(S_t, \mathbf{w})$$

- For TD(0), the target is the TD target $R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w})$

$$\Delta \mathbf{w} = \alpha(R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w})) \nabla_{\mathbf{w}} \hat{v}(S_t, \mathbf{w})$$

- For TD(λ), the target is the λ -return G_t^λ

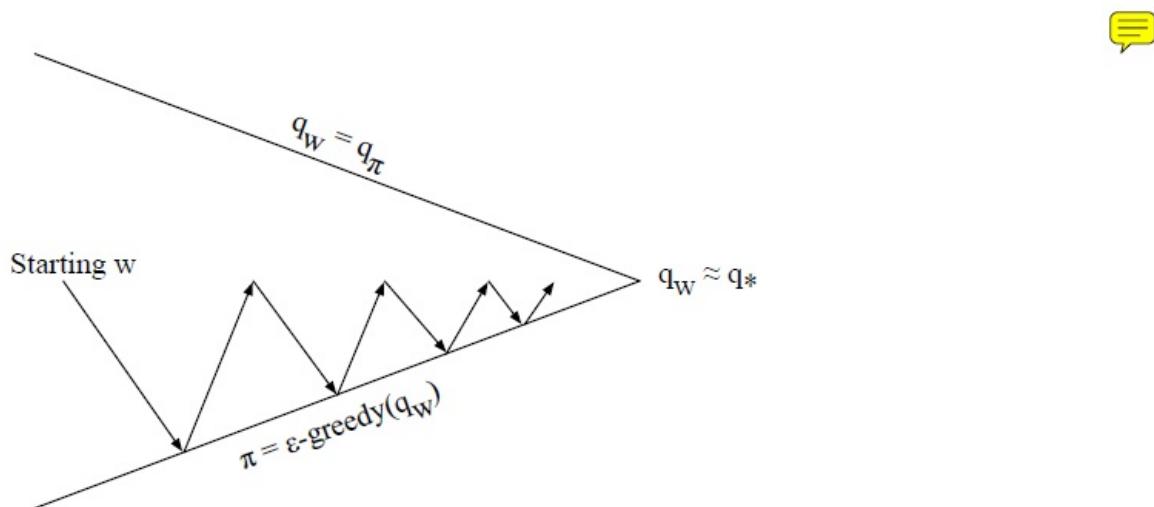
$$\Delta \mathbf{w} = \alpha(G_t^\lambda - \hat{v}(S_t, \mathbf{w})) \nabla_{\mathbf{w}} \hat{v}(S_t, \mathbf{w})$$

Learning with Function Approximator

1. Action-value function approximation

앞에서는 value function을 사용했지만 model free가 되려면 action value function을 사용해야 합니다. 그러한 알고리즘을 그림으로 표현하자면 아래와 같습니다. policy evaluation은 parameter의 update로 진행하며 policy improvement는 그렇게 update된 action value function에 ϵ -greedy한 action을 취함으로서 improve가 됩니다.

Control with Value Function Approximation



Policy evaluation **Approximate** policy evaluation, $\hat{q}(\cdot, \cdot, \mathbf{w}) \approx q_\pi$

Policy improvement ϵ -greedy policy improvement

앞에서 value function으로 했던 내용을 반복하면 아래와 같습니다.

Action-Value Function Approximation

- Approximate the action-value function

$$\hat{q}(S, A, \mathbf{w}) \approx q_\pi(S, A)$$

- Minimise mean-squared error between approximate action-value fn $\hat{q}(S, A, \mathbf{w})$ and true action-value fn $q_\pi(S, A)$

$$J(\mathbf{w}) = \mathbb{E}_\pi [(q_\pi(S, A) - \hat{q}(S, A, \mathbf{w}))^2]$$

- Use stochastic gradient descent to find a local minimum

$$\begin{aligned} -\frac{1}{2} \nabla_{\mathbf{w}} J(\mathbf{w}) &= (q_\pi(S, A) - \hat{q}(S, A, \mathbf{w})) \nabla_{\mathbf{w}} \hat{q}(S, A, \mathbf{w}) \\ \Delta \mathbf{w} &= \alpha(q_\pi(S, A) - \hat{q}(S, A, \mathbf{w})) \nabla_{\mathbf{w}} \hat{q}(S, A, \mathbf{w}) \end{aligned}$$

True value function을 대체하는 것도 아래와 같습니다.

- Like prediction, we must substitute a target for $q_\pi(S, A)$
 - For MC, the target is the return G_t

$$\Delta \mathbf{w} = \alpha(G_t - \hat{q}(S_t, A_t, \mathbf{w})) \nabla_{\mathbf{w}} \hat{q}(S_t, A_t, \mathbf{w})$$

- For TD(0), the target is the TD target $R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$

$$\Delta \mathbf{w} = \alpha(R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}) - \hat{q}(S_t, A_t, \mathbf{w})) \nabla_{\mathbf{w}} \hat{q}(S_t, A_t, \mathbf{w})$$

- For forward-view TD(λ), target is the action-value λ -return

$$\Delta \mathbf{w} = \alpha(q_t^\lambda - \hat{q}(S_t, A_t, \mathbf{w})) \nabla_{\mathbf{w}} \hat{q}(S_t, A_t, \mathbf{w})$$

- For backward-view TD(λ), equivalent update is

$$\delta_t = R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}) - \hat{q}(S_t, A_t, \mathbf{w})$$

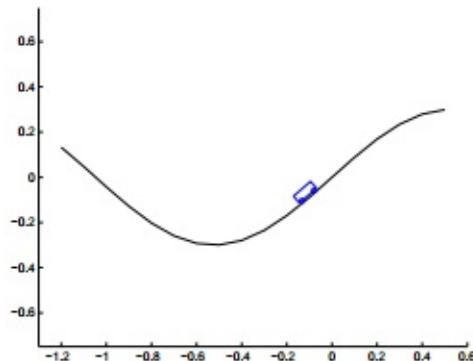
$$E_t = \gamma \lambda E_{t-1} + \nabla_{\mathbf{w}} \hat{q}(S_t, A_t, \mathbf{w})$$

$$\Delta \mathbf{w} = \alpha \delta_t E_t$$

2. Example

Appximation을 통한 강화학습의 예로 cartpole, acrobot과 같이 classical control로 유명한 Mountain Car를 사용 하였습니다. 이 예제의 목표는 Goal에 올라가는 것입니다.

<https://see.stanford.edu/materials/aimlcs229/problemset4.pdf> 문제의 정의는 다음과 같습니다. 정상을 제외한 모든 곳은 time step마다 reward를 -1씩 받게 됩니다. 따라서 최대한 빠른 시간 내에 goal에 올라가는 것을 agent는 목표로 하게 됩니다. 또한 바로 uphill을 할 추력은 차에게 없다고 가정을 한다면 차가 왔다 갔다하면서 중력으로 가속시켜서 올라가야하기 때문에 문제는 어려워집니다. 이러한 문제는 강화학습은 가볍게 풀어줍니다.

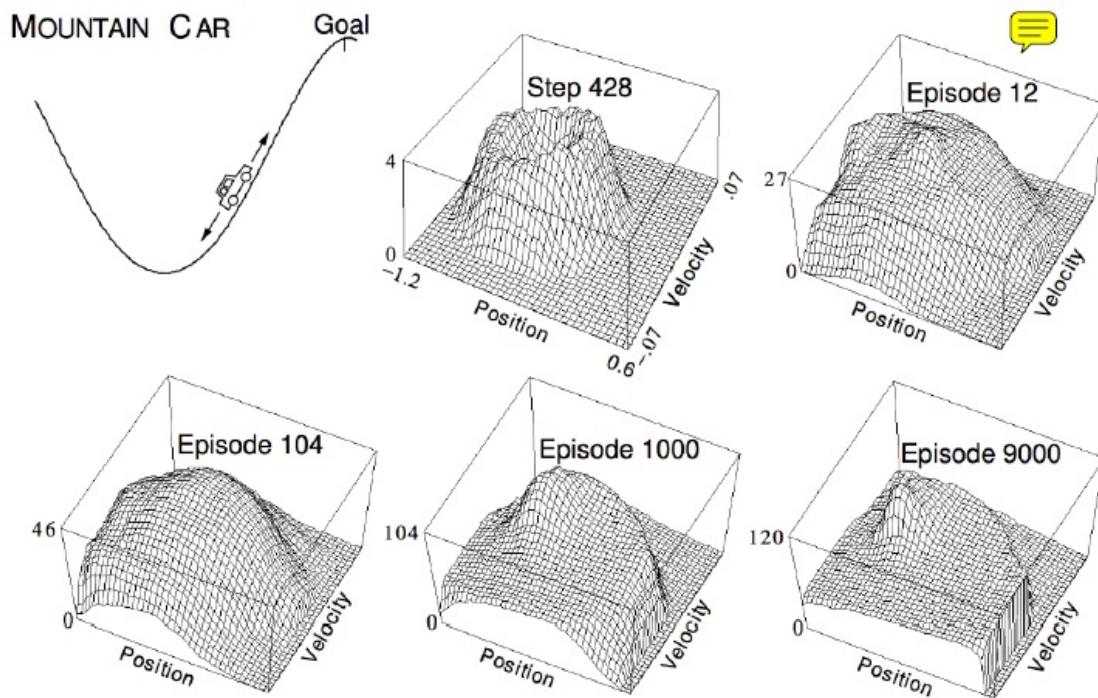


All states except those at the top of the hill have a constant reward $R(s) = -1$, while the goal state at the hilltop has reward $R(s) = 0$; thus an optimal agent will try to get to the top of the hill as fast as possible (when the car reaches the top of the hill, the episode is over, and the car is reset to its initial position). However, when starting at the bottom of the hill, the car does not have enough power to reach the top by driving forward, so it must first acceleraterate accelerate backwards, building up enough momentum to reach the top of the hill. This strategy of moving away from the goal in order to reach the goal makes the problem difficult for many classical control algorithms.

많은 control문제에서 그려듯이 state는 위치와 속도로 정의됩니다. 두 개의 component를 가지기 때문에 state-space는 2차원이 됩니다. 강화학습을 통해 optima policy를 알아내려면 각 state의 value function을 알아야하는데 state가 continuouns하기 때문에 기존의 table방법으로는 풀 수가 없는 문제입니다. 따라서 value function approximator를 통해서 모든 state의 value function을

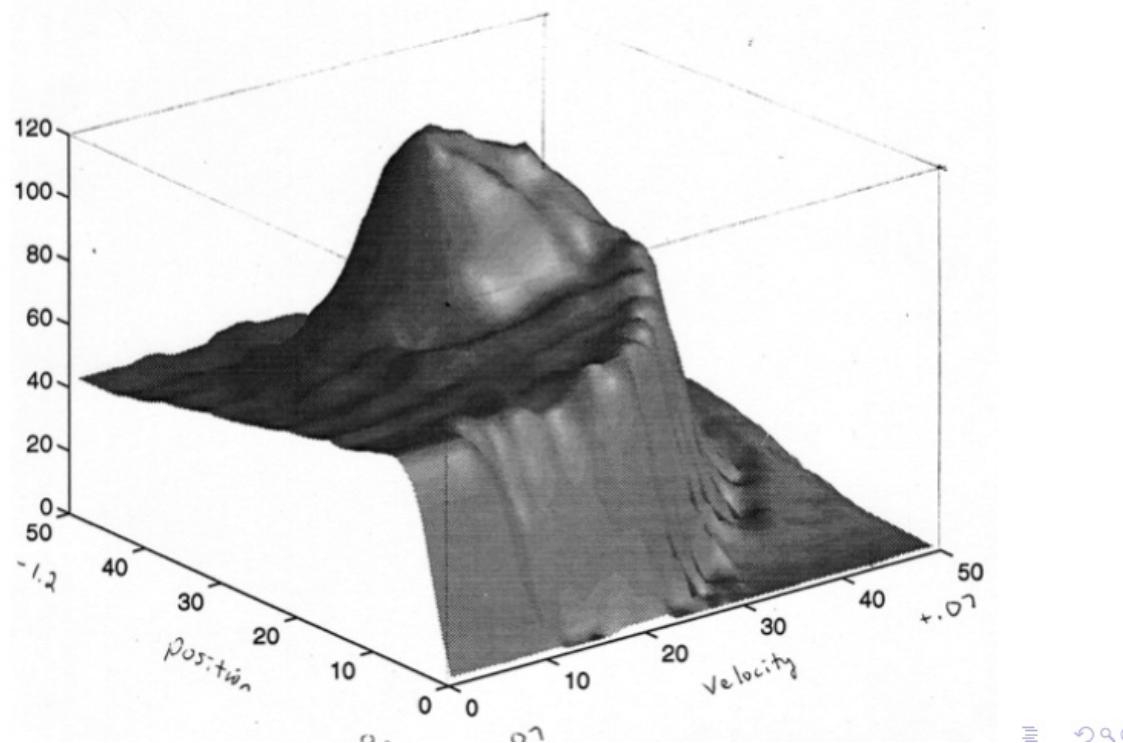
함수화해서 표현할 수 있고 sampling을 통해서 experience를 통해서 value function을 학습해나갑니다. 아래는 그 과정을 보여주는 그래프입니다.

Linear Sarsa with Coarse Coding in Mountain Car



학습이 완료된 value function은 아래와 같이 표현할 수 있고 agent가 하는 일은 각 state에서 가장 높은 value function을 가진 state로 계속 이동하는 것입니다.

Linear Sarsa with Radial Basis Functions in Mountain Car



3. Batch Methods

지금까지 SGD(Stochastic Gradient Descent)을 통해서 parameter를 update하는 방법을 사용했습니다. 하지만 이 방법은 아래와 같은 문제가 있습니다.

Batch Reinforcement Learning

- Gradient descent is simple and appealing
- But it is *not* sample efficient
- Batch methods seek to find the best fitting value function
- Given the agent's experience ("training data")

SGD처럼 차근차근 gradient를 따라서 parameter를 update하는 것이 아니고 training data(agent

가 경험한 것)들을 모아서 한꺼번에 update하는 것이 "Batch Methods"입니다. 하지만 Batch 방법은 한 번에 업데이트하는 만큼 그 많은 데이터들에 가장 잘 맞는 value function을 찾기가 어렵기 때문에 SGD와 Batch 방법의 중간을 사용하는 경우도 많습니다. 예를 들면, step-by-step으로 업데이트하는 것이 아니고 100개의 데이터가 모일 때까지 기다렸다가 100번에 한 번씩 업데이트하는 "mini-batch" 방법도 있습니다.

위에서 말하는 SGD의 문제점인 experience data를 한 번만 사용하는 것이 비효율적이다라고 말하는 점에 대해서는 한 번만 사용하지 않고 여러번 사용하는 것으로 문제를 해결할 수 있습니다. 하지만 어떤 방법으로 여러번 experience data를 활용할 것인가에 대해서 experience replay가 그 답을 말해줍니다.

4. Experience Replay

Experience Replay는 아래와 같습니다. 뒤에서 설명하겠지만 Deepmind에서 Atari Game에 사용했던 알고리즘이고 아래와 같습니다. replay memory라는 것을 만들어 놓고서 agent가 경험했던 것들을 $\$(s_t, a_t, r_{t+1}, s_{t+1})\$$ 로 time-step마다 끊어서 저장해놓습니다. action-value function의 parameter를 update하는 것은 time-step마다 하지만 하나의 transition에 대해서만 하는 것이 아니고 모아놓았던 transition을 replay memory에서 100개면 100개 200개면 200개씩 꺼내서 그 mini-batch에 대해서 update를 진행합니다.

Experience Replay in Deep Q-Networks (DQN)

DQN uses **experience replay** and **fixed Q-targets**



- Take action a_t according to ϵ -greedy policy
- Store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in replay memory \mathcal{D}
- Sample random mini-batch of transitions (s, a, r, s') from \mathcal{D}
- Compute Q-learning targets w.r.t. old, fixed parameters w^-
- Optimise MSE between Q-network and Q-learning targets

$$\mathcal{L}_i(w_i) = \mathbb{E}_{s,a,r,s' \sim \mathcal{D}_i} \left[\left(r + \gamma \max_{a'} Q(s', a'; w_i^-) - Q(s, a; w_i) \right)^2 \right]$$

- Using variant of stochastic gradient descent

이렇게 할 경우에 sample efficient할 수도 있지만 또한 episode 내에서 스텝 바이 스텝으로 업데이트를 하면 그 데이터들 사이의 correlation 때문에 학습이 잘 안되는 문제도 해결할 수 있습니다.

DQN

1. Neural Network

2. Deep Q Networks

Neural Network

1. What is DQN

강화학습에서 agent는 environment를 MDP를 통해서 이해를 하는데 table 형태로 학습을 모든 state에 대한 action-value function의 값을 저장하고 update시켜나가는 식으로 하면 학습이 상당히 느려집니다. 따라서 approximation을 하게되고 그 approximation방법 중에서 nonlinear function approximator로 deep neural network가 있습니다. 따라서 action-value function(q-value)를 approximate하는 방법으로 deep neural network를 택한 reinforcement learning방법이 Deep Reinforcement Learning(deepRL)입니다. 또한 action value function뿐만 아니라 policy 자체를 approximate할 수도 있는데 그 approximator로 DNN을 사용해도 DeepRL이 됩니다.

action value function을 approximate하는 deep neural networks를 Deep Q-Networks(DQN)이라고 하는데 그렇다면 DQN으로 어떻게 학습할까요? DQN이라는 개념은 DeepMind의 "Playing Atari with Deep Reinforcement Learning"라는 논문에 소개되어있습니다.

<https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf>

2. Artificial Neural Networks (ANN)

<http://sanghyukchun.github.io/74/> deepRL을 하려면 딥러닝의 기본적인 개념에 대해서 알 필요가 있습니다. 위 블로그에 게시된 내용을 바탕으로 관련내용을 요약해봤습니다. 저 또한 딥러닝은 처음 접하는 것이라 앞으로도 공부가 필요할 것 같습니다. 강화학습이 사람의 행동방식을 모방했다라고 한다면, artificial neural networks(줄여서 neural networks)는 사람의 뇌의 구조를 모방했습니다. 인공지능이 사람의 뇌를 모방하게 된 것에는 컴퓨터가 계산과 같은 일에는 사람보다 뛰어난 performance를 내지만 개와 고양이를 구별하는 사람이라면 누구나 간단하게 하는 일은 컴퓨터는 하지 못했기 때문입니다. 따라서 이미 뇌의 구조에 대해서는 수많은 뉴런들과 시냅스로 구성되어 있다는 것을 알고 그것을 수학적 모델로 만들어서 컴퓨터의 알고리즘에 적용시키는 방법을 택한 것입니다.

neural networks의 수학적 모델에 대해서 간단히 살펴보겠습니다. 그 전에 사람의 뉴런의 구조를 보면

<http://arxiv.org/pdf/cs/0308031.pdf>

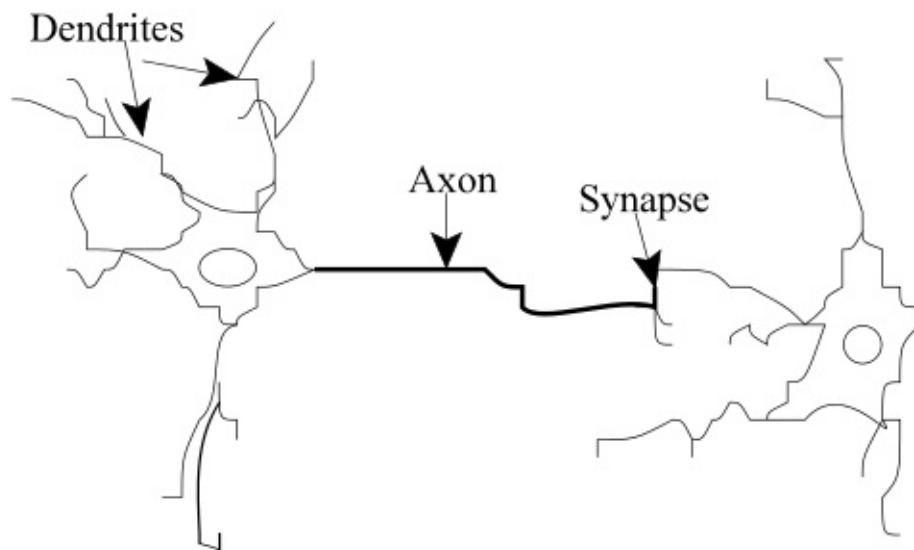


Figure 1. Natural neurons (artist's conception).

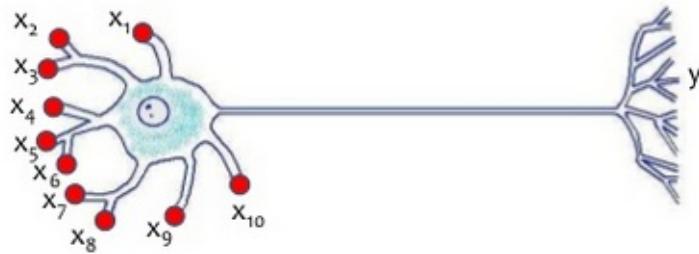
각 Neuron들은 synapse를 통해서 signal을 받습니다. 만약 signal이 어떤 특정한 threshold를 넘어간다면 neuron이 activate되고 그 뉴런은 axon을 통해서 signal을 다른 synapse로 보냅니다. 이러한 구조를 뉴런의 모양을 빼고 process위주로 다시 표현을 해보면 다음과 같습니다.

<http://www.slideshare.net/imanog/artificial-neural-network-48027460>

Modeling Neurons (Computer Science)



Modeling Neurons



Net input signal is a linear combination of input signals x_i .

같습니다. **Each Output is a function of the net input signal.**

이 그림에서와 같이 뉴런의 시냅스가 10개라고 가정해보면 이 시냅스들을 통해서 10개의 다른 input들이 들어오게 됩니다. 뉴런의 process에 들어가는 값은 이 10개의 input들의 linear combination입니다. 이 process를 거친 y 값은 다시 다른 뉴런들의 시냅스로 input으로 들어가게 됩니다. 이러한 사람의 뉴런의 구조를 모방해서 인공신경망을 구성하면, 각 neuron들은 node가 되고 synapse를 통해서 들어오는 signal은 input이 되고 각각 다른 synapse를 통해서 들어오는 signal들의 중요도가 다를 수 있으므로 weight를 곱해줘서 들어오게 됩니다. 이 signal들이 weight와 곱해진 것이 위에서 언급했던 net input signal입니다. 그 net input signal을 식으로 표현해보자면 다음과 같습니다.

Net input signal received through synaptic junctions is

$$\text{net} = b + \sum w_i x_i = b + W^T X$$

Weight vector: $W = [w_1 \ w_2 \ \dots \ w_m]^T$

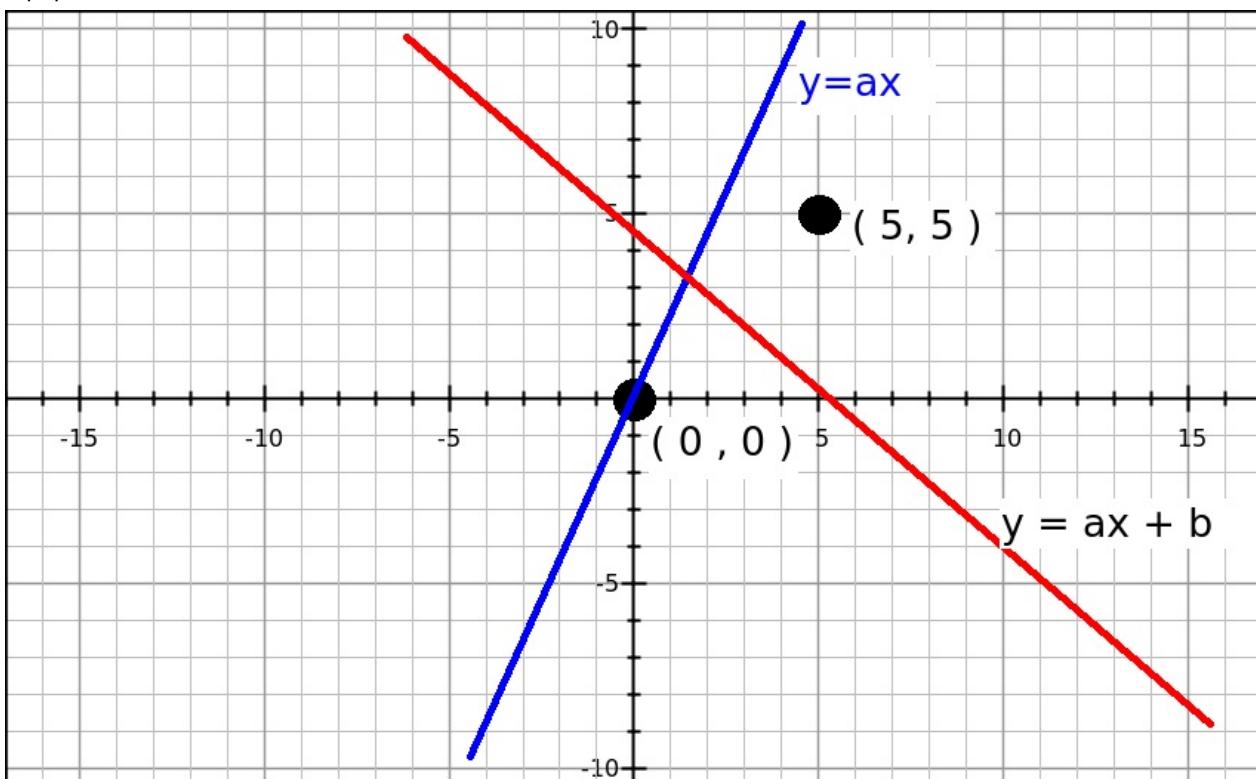
Input vector: $X = [x_1 \ x_2 \ \dots \ x_m]^T$

Each output is a function of the net stimulus signal (f is called the activation function)

$$y = f(\text{net}) = f(b + W^T X)$$

시냅스로 들어오는 각각의 input을 vector로 표현하고 그 input에 각각 곱해지는 weight 또한 그에 따라 vector로 만들어서 두 vector를 곱해서 input과 weight의 linear combination을 만들어줍니다. 여기서 새로운 개념이 나타나는데 b로 써지는 bias입니다.

Bias가 linear combination에 더해져서 net input signal로 들어가는 이유는 간단하게 말하자면 다음과 같습니다. 좌표평면에서 (0,0)과 (5,5)을 어떠한 선을 기준으로 구분하고 싶다고 생각해봅시다(예를 들면, 고양이과 개를 구분하는 문제라고 할 수 있습니다). bias가 없는 $y = ax$ 같은 함수의 경우에는 두 점을 구분할 수 있는 방법이 없습니다. 하지만 $y = ax + b$ 는 이 두 점을 구분할 수 있습니다.



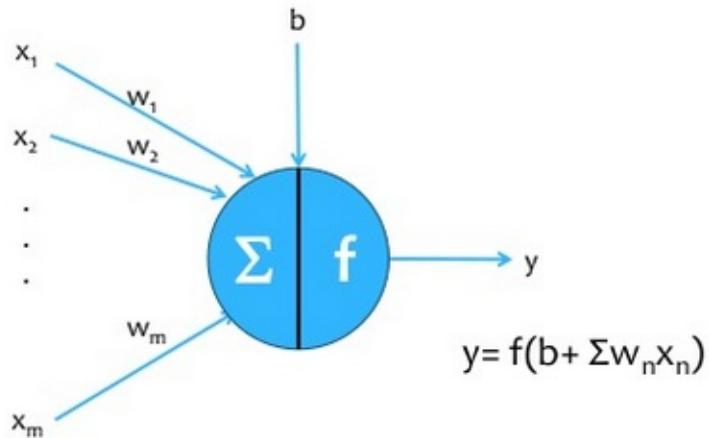
또한 다른 식으로 bias의 필요성을 설명하자면 다음과 같습니다.

<http://stackoverflow.com/questions/2480650/role-of-bias-in-neural-networks>

Modification of neuron WEIGHTS alone only serves to manipulate the shape/curvature of your transfer function, and not its equilibrium/zero crossing point. The introduction of BIAS neurons allows you to shift the transfer function curve horizontally (left/right) along the input axis while leaving the shape/curvature unaltered. This will allow the network to produce arbitrary outputs different from the defaults and hence you can customize/shift the input-to-output mapping to suit your particular needs.

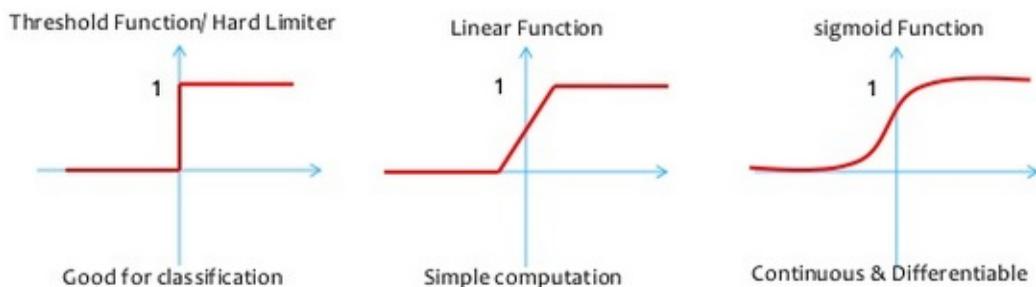
input signal들과 weight가 곱해지고 bias가 더해진 net input signal이 node를 activate시키는데 그 형식을 function으로 정의할 수 있습니다. 그러한 함수를 activation function이라 합니다. 이러한 개념들을 모두 합해서 그림으로 나타낸 artificial neuron은 다음과 같습니다.

General Model for Neurons



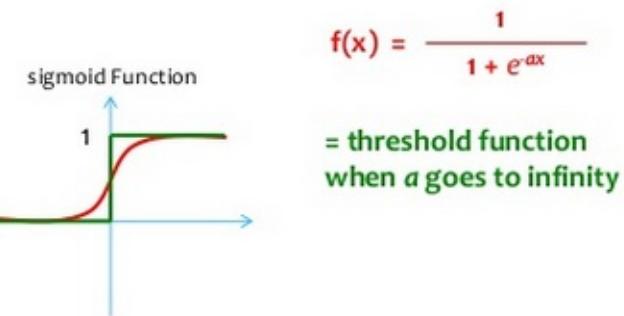
f 라고 표현되어 있는 activation function의 가장 간단한 형태는 들어온 input들의 합이 어떤 Threshold보다 높으면 1이 나오고 낮으면 0이 나오는 형태일 것입니다. 하지만 이런 형태의 activation function의 경우에는 미분이 불가능하고 따라서 gradient descent를 못 쓰기 때문에 그 이외의 미분가능 함수를 사용합니다. gradient descent에 대해서는 뒤에서 설명하겠습니다. 밑의 사진은 activation function의 예시입니다.

Activation functions



위에서 말한 가장 간단한 activation function의 형태는 첫번째 그림과 같습니다. 위에서 언급했듯 이 이 함수 대신에 미분가능한 함수를 사용하게 되었고 그 중에 대표적인 함수가 세번째 그래프인 sigmoid Function입니다. sigmoid function이란 무엇일까요? 식으로 다음과 같이 표현됩니다.

Sigmoid Function

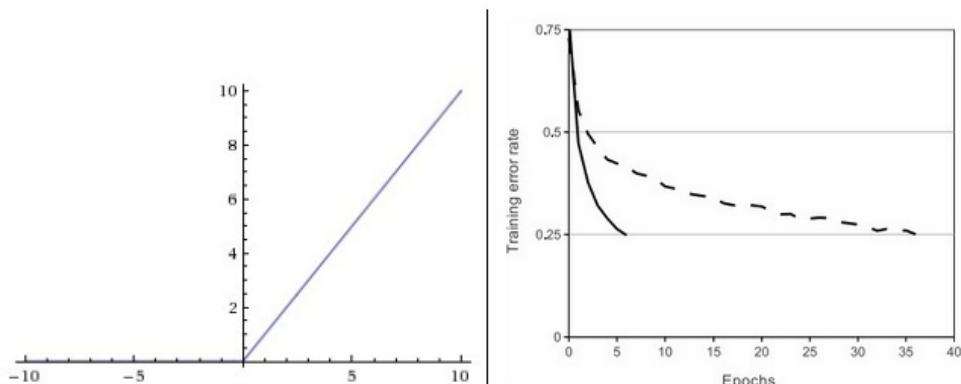


activation function의 예시에는 sigmoid 말고도 세 가지 다른 함수들이 있는데 이 함수들은 다 non-linear합니다. 그 이유는 activation function이 linear 할 경우에는 아무리 많은 neuron layer를 쓰는다 하더라도 그것이 결국 하나의 layer로 표현되기 때문입니다.

- sigmoid function
- tanh function
- absolute function
- ReLU function

가장 실용적인 activation function은 ReLU function이라고 합니다. 저희 또한 ReLU function을 activation function으로 사용했습니다. ReLU란 어떤 함수일까요?

<http://cs231n.github.io/neural-networks-1/>

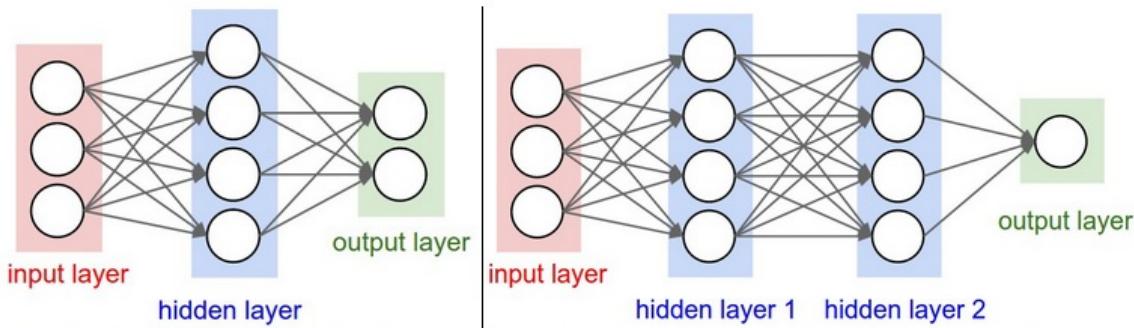


Left: Rectified Linear Unit (ReLU) activation function, which is zero when $x < 0$ and then linear with slope 1 when $x > 0$. Right: A plot from Krizhevsky et al. (pdf) paper indicating the 6x improvement in convergence with the ReLU unit compared to the tanh unit.

ReLU. The Rectified Linear Unit has become very popular in the last few years. It computes the function $f(x) = \max(0, x)$. In other words, the activation is simply thresholded at zero (see image above on the left).

위의 왼쪽 그림처럼 x 가 0보다 작거나 같을때는 y 가 0이 나오고 x 가 0보다 클때는 x 가 나오는 함수를 ReLU(The Rectified Linear Unit)이라는 함수입니다. 위 글에 써져있듯이 최근 몇 년동안 유명해지고 있는 함수입니다. 사실 딥러닝이 최근에 갑자기 급부상한 이유는 엄청 혁신적인 변화가 있었던 것이 아니고 activation함수를 sigmoid에서 ReLU로 바꾸는 등의 작은 변화들의 영향이 크다고 볼 수 있습니다.

sigmoid함수에 비해서 ReLU함수는 어떠한 장점이 있을까요? 위 그림에서 보듯이 ReLU의 직선적인 형태와 sigmoid함수처럼 수렴하는 형태가 아닌 점이 ReLU의 stochastic gradient descent가 더 잘 수렴하게 해줍니다. 또한 상대적으로 sigmoid함수에 비해서 계산량이 줄게 됩니다. 장점이 있으면 단점도 있는 법입니다. 단점은 다음과 같습니다. Learning rate에 따라서 중간에 최대 40%정도의 network가 "die"할 수 있다고 합니다. 단, learning rate를 잘 조절하면 이 문제는 그렇게 크지 않습니다.



Left: A 2-layer Neural Network (one hidden layer of 4 neurons (or units) and one output layer with 2 neurons), and three inputs.
Right: A 3-layer neural network with three inputs, two hidden layers of 4 neurons each and one output layer. Notice that in both cases there are connections (synapses) between neurons across layers, but not within a layer.

앞에서 살펴본 artificial neuron들을 network로 표현하면 위와 같습니다. 사실은 사람 뇌의 뉴런들은 이보다 상당히 더 복잡하게 연결되어 있지만 머신러닝에서 사용하는 neural network는 훨씬 간단한 형태입니다. 위에서 보이는 동그라미들은 뉴런에 해당하는 node들입니다. 이 node들은 각 층으로 분류될 수 있고 같은 층안에서는 연결되어 있지 않습니다. 정보의 방향은 왼쪽에서 오른쪽으로 흘러가는데 그렇지 않은 network도 있습니다(RNN). 보통은 node들이 fully-connected되어 있어서 한 node에서 나온 output들은 다음 층의 모든 node에 input으로 들어가게 됩니다. 왼쪽과 오른쪽은 둘 다 neural network이지만 차이는 오른쪽의 network는 hidden layer가 2층인 것을 알 수 있고 hidden layer가 2층 이상인 neural network를 deep neural network라고 부릅니다.

3. SGD(Stochastic Gradient Descent) and Back-Propagation

(1) SGD

지금까지는 deep neural network가 무엇인지에 대해서 살펴보았습니다. 다시 이 글의 처음으로 돌아가서 DQN이란 action-value function을 deep neural network로 approximation한 것을 말합니다. 강화학습의 목표는 optimal policy를 구하는 것이고 각 state에서 optimal한 action value function을 알고 있으면 q값이 큰 action을 취하면 되는 것이므로 결국은 q-value를 구하면 강화학습 문제를 풀게됩니다. 이 q-value는 DNN(deep neural networks)를 통해서 나오게 되는데 결국 DNN을 학습시키는 것이 목표가 되게 됩니다.

따라서 approximation하지 않았을 때와 다른 것은 q-table을 만들어서 각각의 q-value를 update하는 것이 아니고 DNN안의 weight와 bias를 update하게 됩니다. 그렇다면 어떻게 update할까요?

이 때 이전에 배웠던 Stochastic Gradient Descent가 사용됩니다. 정리하자면 gradient descent라는 것은 w 를 parameter로 가지는 J 라는 objective function을 minimize하는 방법중의 하나로서 w 에 대한 J 의 gradient의 반대방향으로 w 를 update하는 방식을 말합니다.

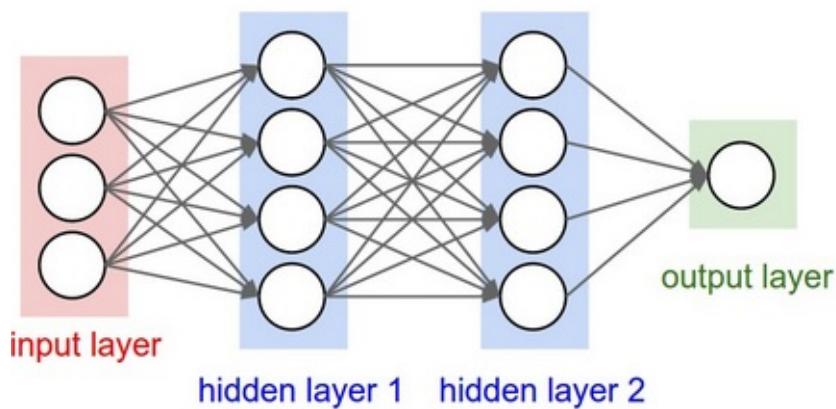
$$\begin{aligned}\Delta w &= -\frac{1}{2}\alpha \nabla_w J(w) \\ &= \alpha \mathbb{E}_\pi [(v_\pi(S) - \hat{v}(S, w)) \nabla_w \hat{v}(S, w)]\end{aligned}$$

이런식으로 update를 하게되는데 모든 데이터에 대해서 gradient를 구해서 한 번 update하는 것이 아니고 sampling을 통해서 순차적으로 update하겠다는 gradient descent방법이 stochastic gradient descent입니다. 아래 페이지를 참고해보면 그렇게 할 경우 수렴하는 속도가 훨씬 빠르며 online으로도 학습할 수 있다는 장점이 있습니다. 또한 하나 중요한 점은 gradient descent방법은 local optimum으로 갈 수 있다는 단점이 있습니다.

<http://sebastianruder.com/optimizing-gradient-descent/>

(2) Back-Propagation

이 gradient를 구했다면 DNN의 안에 있는 parameter들을 어떻게 update할까요? 다시 DNN안에서 data가 전달되어가는 과정을 생각해봅시다. input이 들어가면 layer들을 거쳐가며 output layer에 도달한 data가 output이 되어서 나오게 됩니다.



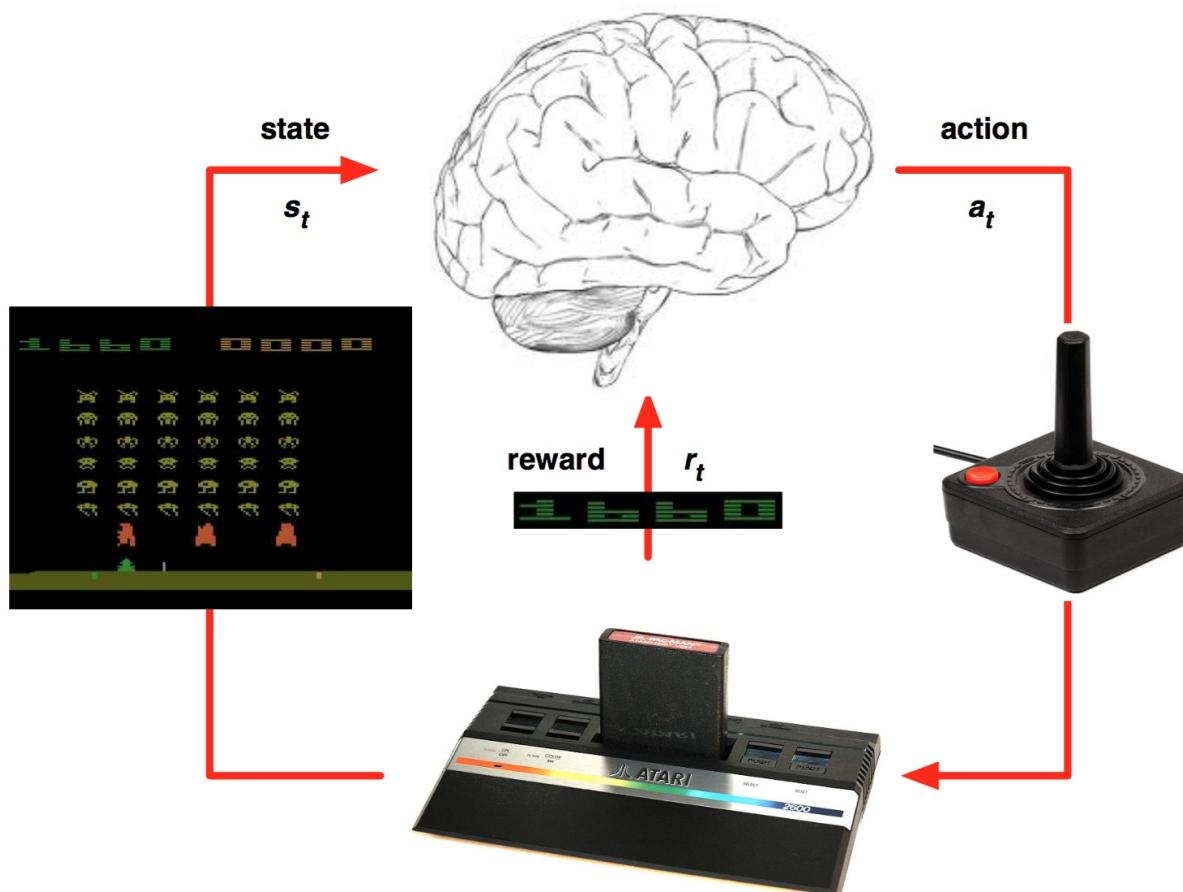
parameter를 SGD로 update할 때는 그 반대 방향으로 가게 됩니다. 따라서 그 이름이 Back-Propagation이라는 이름이 붙습니다. Tensorflow를 사용할 경우에는 그러한 식들이 library화 되어서 여기서 다룰 내용은 아닌 것 같습니다.

Deep Q Networks

첫 번째 chapter에서 Atari game의 학습에 대해서 소개했었습니다. 이 예제는 Playing atari with deep reinforcement learning이라는 논문에서 나온 것으로 링크는 아래와 같습니다. 강화학습 + 딥러닝으로 atari라는 고전 게임을 학습시킴으로 deep reinforcement learning의 시대를 열어주었습니다. <https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf>

이 논문의 abstract는 다음과 같습니다.

We present the first deep learning model to successfully learn control policies directly from high-dimensional sensory input using reinforcement learning. The model is a convolutional neural network, trained with a variant of Q-learning, whose input is raw pixels and whose output is a value function estimating future rewards. We apply our method to seven Atari 2600 games from the Arcade Learning Environment, with no adjustment of the architecture or learning algorithm. We find that it outperforms all previous approaches on six of the games and surpasses a human expert on three of them.



이 논문의 주목할 점은 다음과 같습니다.

1. input data로 raw pixel를 받아온 점
2. 같은 agent로 여러 개의 게임에 적용되어서 학습이 된다는 점
3. Convolutional neural network를 function approximator로 사용
4. Experience Replay

Deep Q Network라는 개념이 여기서 처음 소개되었는데 아래와 같이 action value function을 approximate하는 model로 deep learning의 model을 도입했는데 그 중에서 convolutional network를 도입해서 network를 훈련시키는 것이 DQN이라고 소개하고 있습니다.

We refer to convolutional networks trained with our approach as Deep Q-Networks (DQN).

Convolutional neural network(CNN)은 최근의 딥러닝 열풍을 몰고온 장본인으로서 이미지를 학습시키는 데 최적화된 Neural Network 모델입니다. 이 모델을 사용하면 화면 게임 픽셀 데이터 그 자체로 학습을 시킬 수 있습니다. 그렇기 때문에 따로 게임마다 agent 설정을 달리해주지 않아도 여러 게임에 대해 한 agent로 학습시킬 수 있는 것입니다.

Neural Network에 들어가는 input data에 대해서는 다음과 같이 언급하고 있습니다.

Working directly with raw Atari frames, which are 210X160 pixel images with a 128 color palette, can be computationally demanding, so we apply a basic preprocessing step aimed at reducing the input dimensionality. The raw frames are preprocessed by first converting their RGB representation to gray-scale and down-sampling it to a 110X84 image. The final input representation is obtained by cropping an 84X84 region of the image that roughly captures the playing area

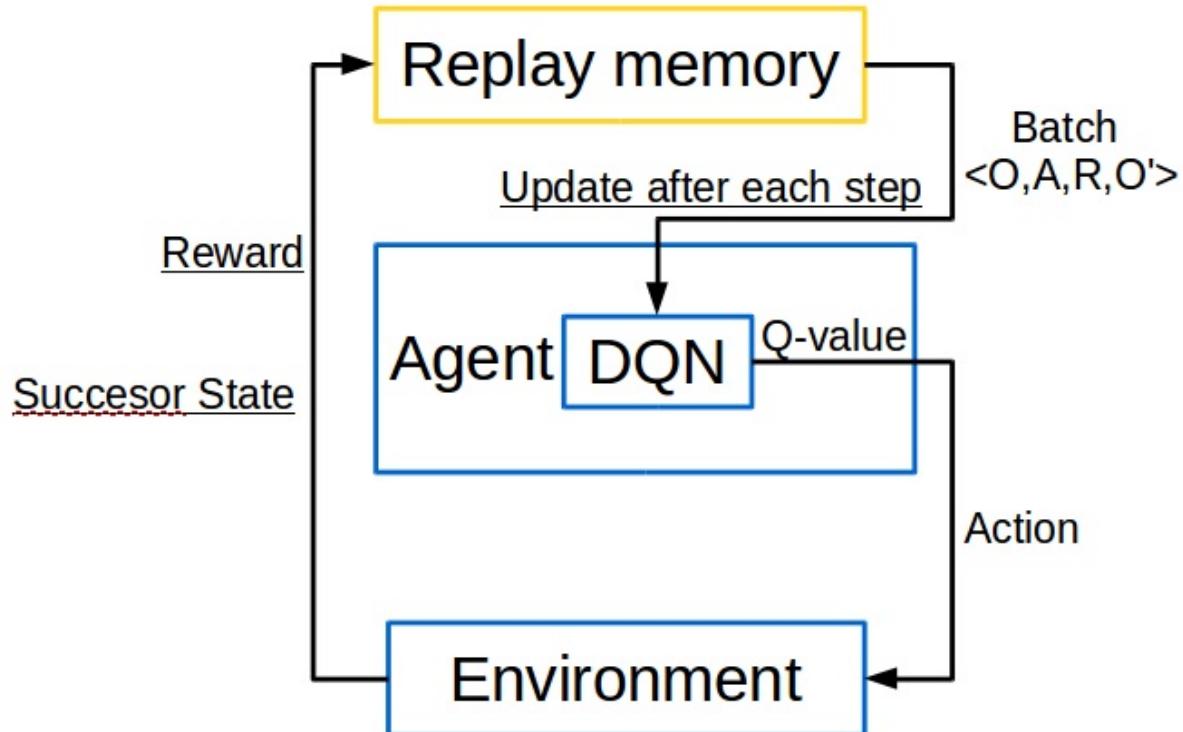
이 단계를 "Preprocessing"이라고 합니다. CNN이 학습할 수 있는 형태로 게임의 화면을 변화시켜 주는 것으로서 일단 색을 없애고 이미지의 크기를 줄이고 위아래의 불필요한 정보를 없애주며 정사각형의 이미지로 만들어주는 과정입니다. 이러한 이미지를 4개씩 묶어서 CNN으로 집어넣게 됩니다.



<https://www.nervanasys.com/demystifying-deep-reinforcement-learning/>

state가 갑자기 pixel data되어서 헷갈릴 수도 있다. 하지만 agent의 입장에서는 단지 data의 형태가 바뀌었을 뿐이고 화면을 하나의 상태로 인식해서 그 상태에서 어떤 행동을 했을 때의 reward를 기억하고 있는 것입니다.

이 알고리즘을 그림으로 나타내보자면 아래와 같습니다. chapter 8에서 언급했던 experience replay를 사용하고 있습니다. transition data들을 replay memory에 넣어 놓고 매 time step마다 mini-batch를 랜덤으로 memory에서 꺼내서 update를 합니다. learning 알고리즘으로는 q-learning을 사용하고 있습니다.



알고리즘은 아래와 같습니다.

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  $\epsilon$  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
  
```

replay memory는 N 개의 episode를 기억하고 있을 수 있는데 N 개가 넘어가면 오래된 episode부터 뺍니다.

episode마다 어떻게 update할까요? loss function을 정의하고 그 gradient를 따라서 업데이트합니다. mini-batch data에 대해서 bootstrap으로 q-learning이 했던 것처럼 $r + \gamma \max_{a'} Q(s', a')$ 를 현재 Q 가 update가 되어야 할 target으로 잡고 그 error를 quadratic하게 잡고서 gradient를 취하면 아래와 같습니다.

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

where $y_i = \mathbb{E}_{s' \sim \mathcal{E}} [r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) | s, a]$ is the target for iteration i and $\rho(s, a)$ is a probability distribution over sequences s and actions a that we refer to as the *behaviour distribution*. The parameters from the previous iteration θ_{i-1} are held fixed when optimising the loss function $L_i(\theta_i)$. Note that the targets depend on the network weights; this is in contrast with the targets used for supervised learning, which are fixed before learning begins. Differentiating the loss function with respect to the weights we arrive at the following gradient,

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s, a \sim \rho(\cdot); s' \sim \mathcal{E}} \left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i) \right) \nabla_{\theta_i} Q(s, a; \theta_i) \right]. \quad (3)$$

chapter8에서 배웠던 내용 활용한 것으로서 달라지는 것은 $\nabla_{\theta_i} Q(s, a; \theta_i)$ 를 어떻게 구하냐입니다. 사실은 이 부분은 딥러닝에 대해서 깊게 들어가야하는 부분인데 tensorflow 같은 라이브러리들이 잘 되어있어서 함수를 호출하면 알아서 계산해줍니다.

10. Policy Gradient

1. Numerical Methods

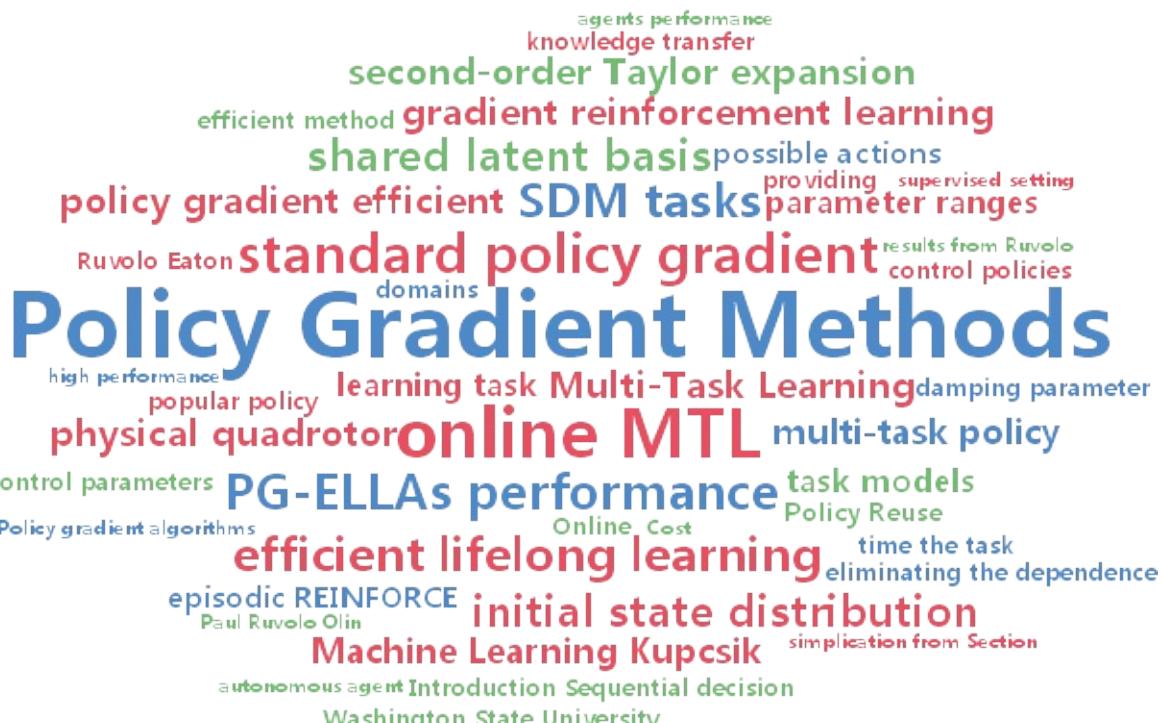
2. Monte-Carlo Policy Gradient

3. Actor-Critic Policy Gradient

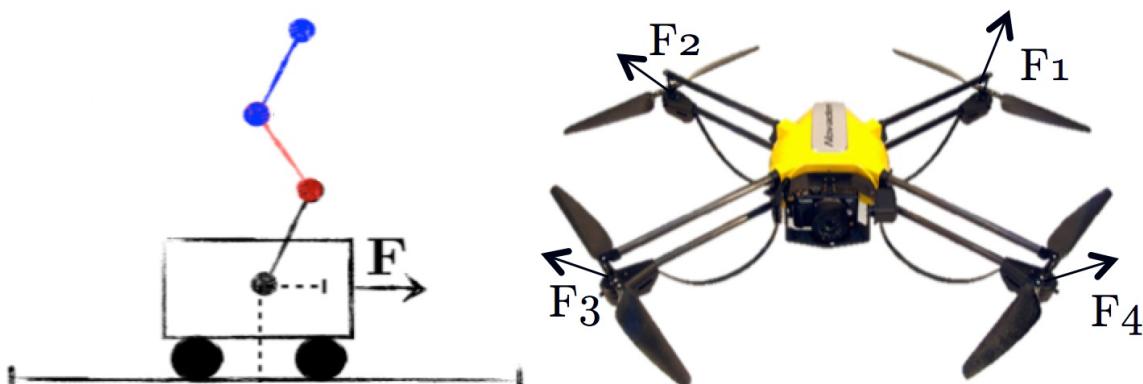
Policy Gradient

1. Policy Gradient

현재 강화학습에서 가장 "hot"하다고 볼 수 있는 방법이 Policy Gradient입니다. 강화학습의 기본 교재인 Sutton 교수님의 책에는 Policy Gradient가 몇 장 안나와 있어서 Silver교수님 RLcourse 7 번째 Policy Gradient강의를 통해 그 개념을 접하고 이해하시는 것을 추천드립니다. (2nd edition에서는 이 부분이 추가되었습니다.) <http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html>



AlphaGo의 알고리즘도 "Policy Gradient with Monte-Carlo Tree Search"라고 합니다. 또한 실재로봇이나 헬리콥터, 드론같은 적용에 적합한 방법이라고 합니다. 처음에 제가 강화학습에 관심을 가지게 된 계기도 드론의 자율주행이었기 때문에 Policy Gradient는 상당히 흥미롭게 다가왔습니다.



Multi-Task Policy Gradient Methods for Control

2. Value-based RL VS Policy-based RL

지금까지 저희가 다루었던 방법들은 모두 "Value-based" 강화학습입니다. 즉, Q라는 action-value function에 초점을 맞추어서 Q function을 구하고 그것을 토대로 policy를 구하는 방식입니다. 이전에 했던 DQN 또한 Value-based RL으로서 DNN을 이용해 Q-function을 approximate하고 policy는 그것을 통해 만들어졌습니다.

* **Value-based reinforcement learning** vs Policy-based reinforcement learning

- In the last lecture we approximated the value or action-value function using parameters θ ,

$$\boxed{V_\theta(s) \approx V^\pi(s)} \\ Q_\theta(s, a) \approx Q^\pi(s, a)}$$

- A policy was generated directly from the value function
 - e.g. using ϵ -greedy

그와 달리 Policy-based RL은 Policy 자체를 approximate해서 function approximator에서 나오는 것이 value function이 아니고 policy 자체가 나옵니다. Policy 자체를 parameterize하는 것입니다. 어떻게 보면 evolutionary 알고리즘의 개념에 더 가깝다고 할 수도 있습니다. 하지만 지금까지 저희가 살펴봤듯이 evolutionary 알고리즘과 달리 강화학습은 환경과의 상호작용이 있습니다.

* Value-based reinforcement learning vs **Policy-based reinforcement learning**

- In this lecture we will directly parametrise the policy

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

- We will focus again on model-free reinforcement learning

왜 이렇게 할까요?? Policy Gradient의 장점과 단점은 다음과 같습니다.

- 기존의 방법의 비해서 수렴이 더 잘되며 가능한 action이 여러개이거나(high-dimension) action 자체가 연속적인 경우에 효과적입니다. 즉, 실재의 로봇 control에 적합합니다.

- 또한 기존의 방법은 반드시 하나의 optimal한 action으로 수렴하는데 policy gradient에서는 stochastic한 policy를 배울 수 있습니다.(예를 들면 가위바위보)

하지만 local optimum에 빠질 수 있으며 policy의 evaluate하는 과정이 비효율적이고 variance가 높습니다.

Advantages:

- Better convergence properties
- Effective in high-dimensional or continuous action spaces 
- Can learn stochastic policies

Disadvantages:

- Typically converge to a local rather than global optimum
- Evaluating a policy is typically inefficient and high variance

기존 방법의 문제를 살펴보도록 하겠습니다. Value-based RL 방식에는 두 가지 문제가 있습니다.

- **Unstable**

Greedy vs Incremental

- **Greedy** updates

$$\theta_{\pi'} = \arg \max_{\theta} \mathbb{E}_{\pi_\theta}[Q^\pi(s, a)]$$

- $V^{\pi_0} \xrightarrow[\text{small change}]{} \pi_1 \xrightarrow[\text{large change}]{} V^{\pi_1} \xrightarrow[\text{large change}]{} \pi_2 \xrightarrow[\text{large change}]{} \dots$
- Potentially **unstable** learning process with **large policy jumps**
- **Policy Gradient** updates

$$\theta_{\pi'} = \theta_\pi + \alpha \left. \frac{dJ(\theta)}{d\theta} \right|_{\theta=\theta^\pi}$$

- $V^{\pi_0} \xrightarrow[\text{small change}]{} \pi_1 \xrightarrow[\text{small change}]{} V^{\pi_1} \xrightarrow[\text{small change}]{} \pi_2 \xrightarrow[\text{small change}]{} \dots$
- **Stable** learning process with **smooth policy improvement**

Value-based RL에서는 Value function을 바탕으로 policy 계산하므로 Value function이 약간만 달라져도 Policy 자체는 왼쪽으로 가다가 오른쪽으로 간다던지하는 크게 변화합니다. 그러한 현상들이 전체적인 알고리즘의 수렴에 불안정성을 더해줍니다. 하지만 Policy 자체가 함수화 되어버리면 학습을 하면서 조금씩 변하는 value function으로 인해서 policy 또한 조금씩 변하게 되어서 안정적이고 부드럽게 수렴하게 됩니다.

- **Stochastic Policy**

때로는 Stochastic Policy가 Optimal Policy일 수 있습니다. 가위바위보 게임은 동등하게 가위와 바위와 보를 1/3씩 내는 것이 Optimal한 Policy입니다. value-based RL에서는 Q function을 토대로 하나의 action만 선택하는 optimal policy를 학습하기 때문에 이러한 문제에는 적용시킬수가 없습니다.

Example: Rock-Paper-Scissors



Sometimes you need stochastic policy!!!

3. Policy Objective Function

이제 기존의 방법처럼 action value function을 approximate하지 않고 policy를 바로 approximate 할 것 입니다. 학습은 policy를 approximate한 parameter들을 update해나가는 것 입니다. 이 parameter를 update하려면 기준이 필요한 데 DQN에서는 TD error를 사용했었습니다. 하지만 Policy Gradient에서는 Objective Function이라는 것을 정의합니다. 정의하는 방법에는 세 가지가 있습니다. state value, average value, average reward per time-step입니다. 똑같은 state에서 시작하는 게임에서는 처음 시작 state의 value function이 강화학습이 최대로 하고자 하는 목표가 됩니다. 두 번째는 잘 사용하지 않고 세 번째는 각 time step마다 받는 reward들의 expectation값을 사용합니다. 사실은 time-step마다 받은 reward들을 discount시키지 않고 stationary

distribution을 사용해서 어떤 행동이 좋았나에 대한 credit assignment 문제를 풀고있지 않나 생각됩니다.

* How to optimize policy?

Policy Objective Functions

- Goal: given policy $\pi_\theta(s, a)$ with parameters θ , find best θ
- But how do we measure the quality of a policy π_θ ?
- In episodic environments we can use the start value

$$J_1(\theta) = V^{\pi_\theta}(s_1) = \mathbb{E}_{\pi_\theta} [v_1]$$

- In continuing environments we can use the average value

$$J_{avV}(\theta) = \sum_s d^{\pi_\theta}(s) V^{\pi_\theta}(s)$$

- Or the average reward per time-step

$$J_{avR}(\theta) = \sum_s d^{\pi_\theta}(s) \sum_a \pi_\theta(s, a) \mathcal{R}_s^a$$

- where $d^{\pi_\theta}(s)$ is stationary distribution of Markov chain for π_θ

Stationary distribution은 처음 접하는 개념일겁니다. sutton 교수님의 policy gradient 논문에서는 stationary distribution을 다음과 같이 정의하고 있습니다.

<https://webdocs.cs.ualberta.ca/~sutton/papers/SMSM-NIPS99.pdf>

*Stationary distribution??

With function approximation, two ways of formulating the agent's objective are useful. One is the average reward formulation, in which policies are ranked according to their long-term expected reward per step, $\rho(\pi)$:

$$\rho(\pi) = \lim_{n \rightarrow \infty} \frac{1}{n} E \{ r_1 + r_2 + \dots + r_n \mid \pi \} = \sum_s d^\pi(s) \sum_a \pi(s, a) \mathcal{R}_s^a,$$

where $d^\pi(s) = \lim_{t \rightarrow \infty} Pr \{ s_t = s \mid s_0, \pi \}$ is the stationary distribution of states under π , which we assume exists and is independent of s_0 for all policies. In the average reward formulation, the value of a state-action pair given a policy is defined as

$$Q^\pi(s, a) = \sum_{t=1}^{\infty} E \{ r_t - \rho(\pi) \mid s_0 = s, a_0 = a, \pi \}, \quad \forall s \in \mathcal{S}, a \in \mathcal{A}.$$

The second formulation we cover is that in which there is a designated start state s_0 , and we care only about the long-term reward obtained from it. We will give our results only once, but they will apply to this formulation as well under the definitions

$$\rho(\pi) = E \left\{ \sum_{t=1}^{\infty} \gamma^{t-1} r_t \mid s_0, \pi \right\} \quad \text{and} \quad Q^\pi(s, a) = E \left\{ \sum_{k=1}^{\infty} \gamma^{k-1} r_{t+k} \mid s_t = s, a_t = a, \pi \right\}.$$

where $\gamma \in [0, 1]$ is a discount rate ($\gamma = 1$ is allowed only in episodic tasks). In this formulation, we define $d^\pi(s)$ as a discounted weighting of states encountered starting at s_0 and then following π : $d^\pi(s) = \sum_{t=0}^{\infty} \gamma^t Pr \{ s_t = s \mid s_0, \pi \}$.

1. average-reward formulation

2. Start state formulation

개인적으로는 위에서 언급했다시피 각 state에 머무르는 비율로 이해하고 있습니다. 이러한 stationary distribution이 어떻게 구현되었나 궁금하기도 합니다.

Policy Gradient에서 목표는 이 Objective Function을 최대화시키는 Policy의 Parameter Vector을 찾아내는 것입니다. 그렇다면 어떻게 찾아낼까요? 바로 Gradient Descent입니다. 그래서 Policy Gradient라고 불리는 것입니다. 다음에서는 Objective Function의 Gradient를 어떻게 구하는지에 대해서 보겠습니다.

■ Find θ that maximises $J(\theta)$

$$J_1(\theta) = V^{\pi_\theta}(s_1) = \mathbb{E}_{\pi_\theta} [v_1]$$

$$J_{avR}(\theta) = \sum_s d^{\pi_\theta}(s) \sum_a \pi_\theta(s, a) \mathcal{R}_s^a$$

4. How to get gradient of objective function

Objective Function의 Gradient를 구하는 방법이 핵심인데 세 가지 방법이 있습니다.

- Finite Difference Policy Gradient
- Monte-Carlo Policy Gradient
- Actor-Critic Policy Gradient

하나씩 차근 차근 살펴보도록 하겠습니다.

Finite Difference Policy Gradient

1. Finite Difference Policy Gradient

이 방법은 수치적인 방법으로서 가장 간단하게 objective function의 gradient를 구할 수 있는 방법입니다. 만약 Parameter vector가 5개의 dimension로 이루어져 있다고 한다면 각 parameter를 ϵ 만큼 변화시켜보고 5개의 parameter에 대한 Gradient를 각각 구하는 것입니다. parameter space가 작을 때는 간단하지만 늘어날수록 비효율적이고 노이지한 방법입니다. policy 가 미분 가능하지 않더라도 작동한다는 장점이 있어서 초기 policy gradient에서 사용되던 방법입니다.

* Numerical Method

- To evaluate policy gradient of $\pi_\theta(s, a)$
- For each dimension $k \in [1, n]$
 - Estimate k th partial derivative of objective function w.r.t. θ
 - By perturbing θ by small amount ϵ in k th dimension

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

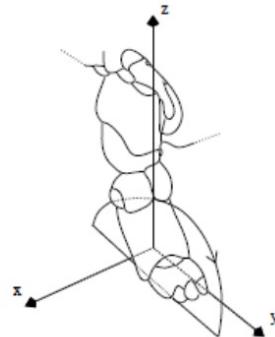
where u_k is unit vector with 1 in k th component, 0 elsewhere

- Uses n evaluations to compute policy gradient in n dimensions
- Simple, noisy, inefficient - but sometimes effective
- Works for arbitrary policies, even if policy is not differentiable

2. Example : Training AIBO

강화학습으로 로봇을 학습시키는 논문중의 Finite Difference Policy Gradient를 사용해서 Sony의 AIBO를 학습시킨 2004년 논문이 있습니다. Silver교수님 수업에서도 아래 그림과 같이 소개되었습니다.

Training AIBO to Walk by Finite Difference Policy Gradient



- Goal: learn a fast AIBO walk (useful for Robocup)
- AIBO walk policy is controlled by 12 numbers (elliptical loci)
- Adapt these parameters by finite difference policy gradient
- Evaluate performance of policy by field traversal time

<http://www.sony-aibo.com/> AIBO는 아래와 같이 네 발 달린 강아지모양의 로봇입니다. 이 로봇에는 기본적인 걸음걸이가 있는데 그것이 느려서 더 빠르게 걸음걸이(gait)를 튜닝하는데 강화학습을 사용한 것입니다. 다리의 궤적 자체를 parameterize했기 때문에 Policy gradient 방법이라고 볼 수 있습니다. 다리의 궤적 자체를 policy로 보는 것입니다. 따라서 가장 빠른 걸음걸이를 얻기 위해 12개의 component로 구성되어 있는 parameter vector를 학습시키는 것이 목표입니다. 여기서 objective function은 속도가 됩니다.(따라서 사실 앞에서 언급했던 objective function에 대한 식들은 여기서는 사용하지 않는다는 것)

A photograph of a white Sony AIBO robot sitting on a light-colored surface. The robot has black accents around its eyes and on its front paws.

SONY AIBO JUST BEAUTIFUL

Aibo a unique companion capable of expressing emotion and communicating with other electronic devices at home and beyond. Effortlessly, AIBO will use its Artificial Intelligence to blend into your lifestyle, occasionally requesting you to tickle it under the chin or to stroke its back.

[LEARN MORE >](#)

Gradient를 구하는 방법은 아래와 같습니다.

- (1) Parameter vector π (directly represent the policy of Aibo)

(2) To estimate gradient numerically generate t randomly generated policies

즉, 12개의 parameter를 기준의 parameter보다 미세한 양을 랜덤하게 변화시킨 t 개의 policy를 생성하는 것입니다. 그러면 변화가 된 t 개의 policy의 objective function(속도)측정합니다.. 12개 parameter 각각에 대해 average score를 계산해서 update를 하면 됩니다.

Our approach starts from an initial parameter vector $\pi = \{\theta_1, \dots, \theta_N\}$ (where $N = 12$ in our case) and proceeds to estimate the partial derivative of π 's objective function with respect to each parameter. We do so by first evaluating t randomly generated policies $\{R_1, R_2, \dots, R_t\}$ near π , such that each $R_i = \{\theta_1 + \Delta_1, \dots, \theta_N + \Delta_N\}$ and each Δ_j is chosen randomly to be either $+\epsilon_j$, 0, or $-\epsilon_j$. Each ϵ_j is a fixed value that is small relative to θ_j . As described below, the evaluation of each policy generates a score that is a measure of the speed of the gait described by that policy.

After evaluating the speed of each R_i , we estimate the partial derivative in each of the N dimensions. We do this by grouping each R_i into one of three sets for each dimension n :

$$R_i \in \begin{cases} S_{+\epsilon,n} & \text{if the } n\text{th parameter of } R_i \text{ is } \theta_n + \epsilon_n \\ S_{+0,n} & \text{if the } n\text{th parameter of } R_i \text{ is } \theta_n + 0 \\ S_{-\epsilon,n} & \text{if the } n\text{th parameter of } R_i \text{ is } \theta_n - \epsilon_n \end{cases}$$

We then compute an average score $Avg_{+\epsilon,n}$, $Avg_{+0,n}$, and $Avg_{-\epsilon,n}$ for $S_{+\epsilon,n}$, $S_{+0,n}$, and $S_{-\epsilon,n}$, respectively. These three averages give us an estimate of the benefit of altering the n th parameter by $+\epsilon_n$, 0, or $-\epsilon_n$. Note that in expectation,

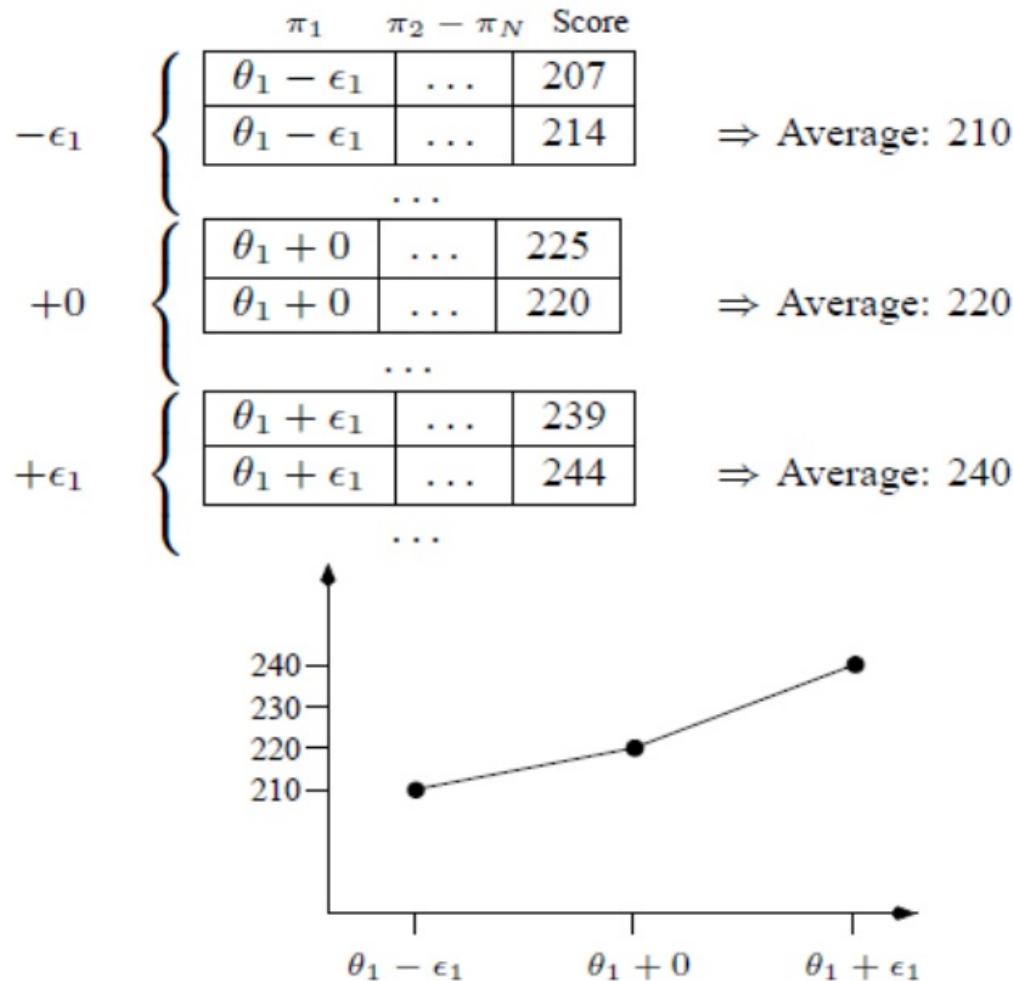


Fig. 3. An example of the process for estimating the gradient in one dimension. Each R_i is grouped into one of three categories, depending on the value of the first parameter (π_1) of each R_i . The averages for these categories can be used to estimate the value of the objective function at and around θ_1 .

로봇의 걸음걸이 자체를 함수화할 수 없어서 미분 불가능한 문제를 numerical하게 하나하나 해보면서 풀었던 예시였습니다. 최근에 와서는 잘 사용하지 않는 방법입니다.

Monte-Carlo Policy Gradient : REINFORCE

앞에서 살펴봤던 Finite Difference Policy gradient는 numerical한 방법이고 앞으로 살펴볼 Monte-Carlo Policy Gradient와 Actor-Critic은 analytical하게 gradient를 계산하는 방법입니다. analytical하게 gradient를 계산한다는 것은 objective function에 직접 gradient를 취해준다는 것입니다. 이때, Policy는 미분가능하다고 가정합니다.

이 때 gradient를 계산하는 것을 episode마다 해주면 MC Policy Gradient이고 time step마다 계산할 수 있으면 Actor-critic입니다. 밑에서 보면 score function이라는 것이 나옵니다. score function이란 무엇일까요?

1. Score Function

analytical하게 gradient를 계산하기 위해서 Objective function에 직접 gradient를 취하면 다음과 같습니다. objective function으로 average reward formulation을 사용하였습니다. gradient를 θ 에 대해서 취하기 때문에 objective function의 식 중에서 policy에만 gradient를 취하면 되서 안쪽으로 들어가게 됩니다.

$$\begin{aligned} J(\theta) &= \mathbb{E}_{\pi_\theta} [r] \\ &= \sum_{s \in \mathcal{S}} d(s) \sum_{a \in \mathcal{A}} \pi_\theta(s, a) \mathcal{R}_{s,a} \\ \nabla_\theta J(\theta) &= \sum_{s \in \mathcal{S}} d(s) \sum_{a \in \mathcal{A}} \pi_\theta(s, a) \nabla_\theta \log \pi_\theta(s, a) \mathcal{R}_{s,a} \\ &= \mathbb{E}_{\pi_\theta} [\nabla_\theta \log \pi_\theta(s, a) r] \end{aligned}$$

하지만 gradient가 안쪽으로 들어가면서 log가 갑자기 나오는데 그 이유는 다음과 같습니다. ∇_θ

π^θ 에 $\log \pi^\theta$ 를 곱하고 나누면 아래와 같이 \log 의 미분의 형태가 되기 때문에 $\log \pi^\theta$ 의 gradient를 $\log \pi^\theta$ 의 gradient로 바꿀 수가 있습니다.

$$\frac{d(\log x)}{dx} = \frac{1}{x}$$

$$\frac{d(\log f(x))}{dx} = \frac{f'(x)}{f(x)}$$

$$\begin{aligned}\nabla_\theta \pi_\theta(s, a) &= \pi_\theta(s, a) \frac{\nabla_\theta \pi_\theta(s, a)}{\pi_\theta(s, a)} \\ &= \pi_\theta(s, a) \nabla_\theta \log \pi_\theta(s, a)\end{aligned}$$

왜 이렇게 하는 것일까요? 만약에 \log 의 형태로 바꾸지 않았다고 생각하면 식은 다음과 같이 됩니다.

$$\sum_{s \in S} d(s) \sum_{a \in A} \{ \nabla_\theta \pi_\theta(s, a) \quad R_{s,a} \}$$

이렇게 되면 $\log \pi^\theta$ 가 사라졌기 때문에 expectation을 취할 수가 없습니다. 결국은 expectation으로 묶어서 그 안을 sampling하게 되어야 강화학습이 될텐데 따라서 expectation을

취하기 위해서 policy를 나눴다가 곱하는 것입니다. 그래서 score function은 아래와 같이 정의가 됩니다.

- We now compute the policy gradient *analytically*
- Assume policy π_θ is differentiable whenever it is non-zero
- and we know the gradient $\nabla_\theta \pi_\theta(s, a)$
- Likelihood ratios exploit the following identity

$$\begin{aligned}\nabla_\theta \pi_\theta(s, a) &= \pi_\theta(s, a) \frac{\nabla_\theta \pi_\theta(s, a)}{\pi_\theta(s, a)} \\ &= \pi_\theta(s, a) \nabla_\theta \log \pi_\theta(s, a)\end{aligned}$$

- The **score function** is $\nabla_\theta \log \pi_\theta(s, a)$

$$\mathbb{E}_{\pi_\theta} [\nabla_\theta \log \pi_\theta(s, a)r]$$

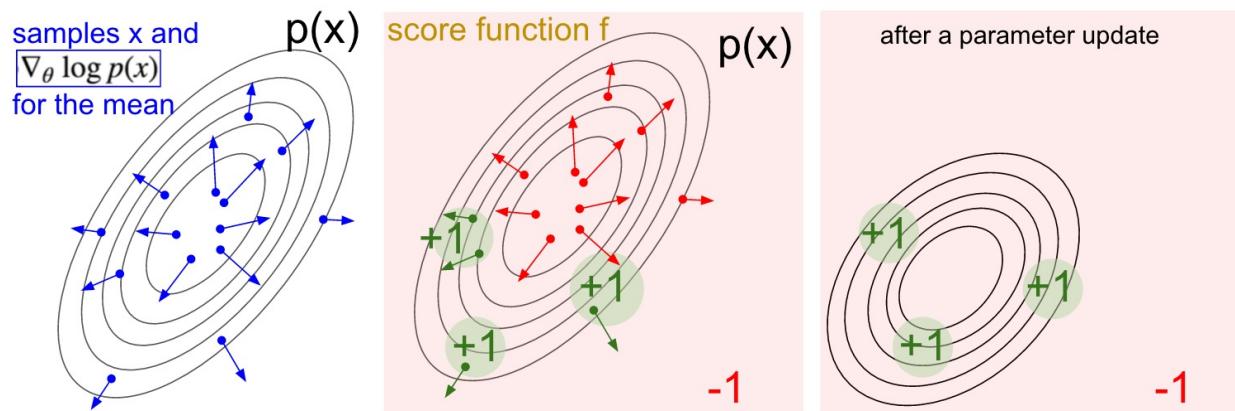
objective function의 gradient는 다음과 같습니다.

2. Policy Gradient Theorem

$$\mathbb{E}_{\pi_\theta} [\nabla_\theta \log \pi_\theta(s, a)r]$$

이 식의 의미는 다음과 같습니다. $p(x)$ 는 policy라고 보시면 되는데 $\nabla_\theta \log p(x)$ 는 이 policy를 표현하는 parameter space에서의 gradient입니다. 이 때 여기에 scalar인 reward r 을 곱해줌으로서 어떤 방향으로 policy를 업데이트해야 하는지를 결정합니다. 따라서 그 방향으로 parameter space에서의 policy가 이동하게 됩니다.

<http://karpathy.github.io/2016/05/31/rl/>



이 때, policy가 어디로 얼마나 update 될 것인지의 척도가 되는 scalar function으로 immediate reward만 사용하면 그 순간에 잘했나, 잘 못했나의 정보밖에 모르기 때문에 제대로 학습이 되지 않을 가능성이 높습니다. 이 immediate reward 대신에 자신이 한 행동에 대한 long-term reward인 action-value function을 사용하겠다는 것이 Policy Gradient Theorem입니다. 따라서 아래의 마지막 식을 보면 대신에 Q function이 들어간 것을 볼 수 있습니다. 한 순간 순간의 reward를 보는 것 이 아니라 지금까지 강화학습이 그래왔듯이 long-term value를 보겠다는 것입니다. 이 Theorem은 Sutton 교수님의 "Policy Gradient Methods for Reinforcement Learning with Function Approximation" 논문에 증명되어 있습니다.

Policy Gradient Theorem

- The policy gradient theorem generalises the likelihood ratio approach to multi-step MDPs
- Replaces instantaneous reward r with long-term value $Q^\pi(s, a)$
- Policy gradient theorem applies to start state objective, average reward and average value objective

Theorem

For any differentiable policy $\pi_\theta(s, a)$, for any of the policy objective functions $J = J_1, J_{avR}$, or $\frac{1}{1-\gamma} J_{avV}$, the policy gradient is

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta} [\nabla_\theta \log \pi_\theta(s, a) Q^{\pi_\theta}(s, a)]$$

3. Stochastic Policy

위의 gradient를 통해서 policy의 parameter들을 업데이트할 것입니다. 하지만 그 전의 stochastic한 policy를 어떻게 표현할 수 있을까요? 보통 딥러닝에서 output node에서 많이 사용되는 nonlinear 함수인 Sigmoid 함수와 Softmax 함수를 많이 사용합니다.

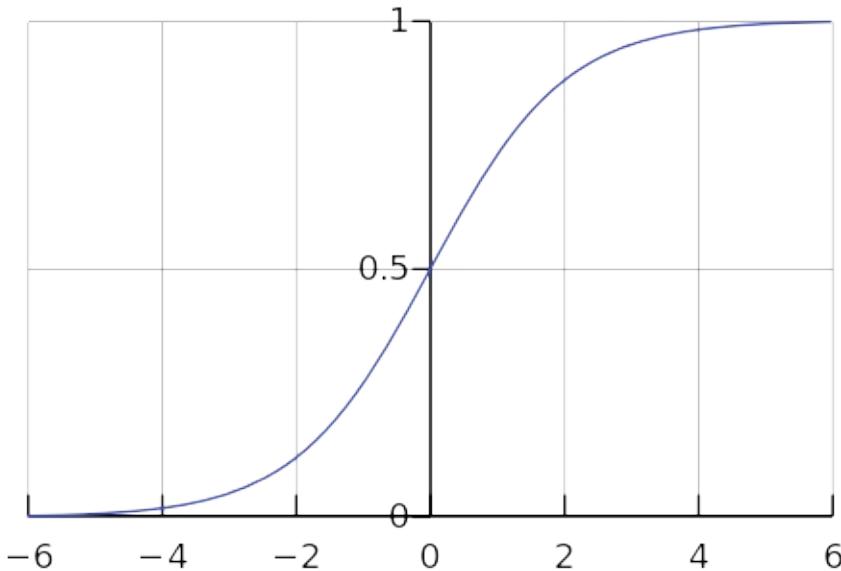
- **Sigmoid**

Sigmoid 함수는 다음과 같이 표현됩니다. https://en.wikipedia.org/wiki/Sigmoid_function

$$S(t) = \frac{1}{1 + e^{-t}}$$

이 함수를 그래프로 나타내면 다음과 같습니다. 이 함수는 output이 0~1 사이의 값으로 나오는 함수입니다. 따라서 stochastic 즉 확률을 나타내는 데에는 좋습니다.

discrete action space의 경우 agent가 왼쪽과 오른쪽으로 갈 수 있다면(action = left, right) 이 함수에서 나오는 값이 "1에 가깝다면 왼쪽으로 갈 확률이 높고 0에 가깝다면 오른쪽으로 갈 확률이 높다"라는 식으로 설정하여 stochastic한 policy를 표현할 수 있습니다. 또는 continuous action space일 경우에는 다른 형태로 표현할 수도 있습니다. 만약 어떤 로봇의 controller에 0부터 100까지 control input을 줄 수 있다면 sigmoid함수를 통해 0이 나오면 control input은 0, 1이 output으로 나오면 control input은 100을 주는 식으로 설정하면 continuous action또한 표현할 수 있습니다.



- **Softmax**

만약 discrete action space에서 action이 3개 이상이 되면 sigmoid함수로 표현하기가 애매해집니다. 이럴 때는 Softmax함수를 쓰는 것이 좋습니다. Softmax함수는 다음과 같이 표현할 수 있습니다. https://en.wikipedia.org/wiki/Softmax_function

Reinforcement learning [edit]

In the field of reinforcement learning, a softmax function can be used to convert values into action probabilities. The function commonly used is:^[3]

$$P_t(a) = \frac{\exp(q_t(a)/\tau)}{\sum_{i=1}^n \exp(q_t(i)/\tau)},$$

where the action value $q_t(a)$ corresponds to the expected reward of following action a and τ is called a temperature parameter (in allusion to statistical mechanics). For high temperatures ($\tau \rightarrow \infty$), all actions have nearly the same probability and the lower the temperature, the more expected rewards affect the probability. For a low temperature ($\tau \rightarrow 0^+$), the probability of the action with the highest expected reward tends to 1.

action이 $i=1$ 부터 n 까지 있을 때 각각의 action probability를 위의 함수로 표현할 수 있습니다. agent가 두 가지 행동을 한 번에 할 수는 없으므로 위 식이 좋은 건 모든 action probability를 더하면 1이 된다는 점입니다.

이렇게 stochastic한 policy를 어떻게 표현하는지, sigmoid와 softmax에 대해서 간단히 설명했는데 사실 이론보다는 실제로 코드로 구현할 때 해보면 더 잘 이해가 될 것 같습니다.

4. Monte-Carlo Policy Gradient

여기까지 policy gradient를 통해서 학습을 할 준비는 끝났습니다. objective function을 정의했고 policy를 parameter를 통해서 나타났을 때 그 parameter를 update하기 위해서 objective function의 gradient를 구해야 했습니다. objective function의 gradient는 아래와 같이 정의됩니다. 하지만 action value function의 값을 어떻게 알 수 있을까요? 이전에 모든 state에 대해 action value function을 알기 어려워서 approximation을 했었는데 policy 자체를 update하려니 기준이 필요하고 그러다보니 action value function을 사용해야하는데 사실 이 값을 알 방법이 애매합니다.

Policy Gradient Theorem

- The policy gradient theorem generalises the likelihood ratio approach to multi-step MDPs
- Replaces instantaneous reward r with long-term value $Q^{\pi}(s, a)$
- Policy gradient theorem applies to start state objective, average reward and average value objective

Theorem

For any differentiable policy $\pi_{\theta}(s, a)$, for any of the policy objective functions $J = J_1, J_{avR}$, or $\frac{1}{1-\gamma} J_{avV}$, the policy gradient is

$$\nabla_{\theta} J(\theta) = \mathbb{E}_{\pi_{\theta}} [\nabla_{\theta} \log \pi_{\theta}(s, a) Q^{\pi_{\theta}}(s, a)]$$

하지만 알 수 있는 방법이 있는데 바로 Monte-Carlo 방법입니다. episode를 가보고 받았는 reward들을 기억해놓고 episode가 끝난 다음에 각 state에 대한 return을 계산하면 됩니다. return 자체가

action value function의 unbiased estimation입니다. 이러한 알고리즘은 REINFORCE라고 하며 아래와 같습니다.

Monte-Carlo Policy Gradient (REINFORCE)

- Update parameters by stochastic gradient ascent
- Using policy gradient theorem
- Using return v_t as an unbiased sample of $Q^{\pi_\theta}(s_t, a_t)$

$$\Delta\theta_t = \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) v_t$$

function REINFORCE

```

Initialise  $\theta$  arbitrarily
for each episode  $\{s_1, a_1, r_2, \dots, s_{T-1}, a_{T-1}, r_T\} \sim \pi_\theta$  do
    for  $t = 1$  to  $T - 1$  do
         $\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) v_t$ 
    end for
end for
return  $\theta$ 
end function

```

loop문을 보시면 학습, 즉 parameter의 update가 episode마다 일어나고 있음을 알 수 있습니다. 이 때 parameter를 regression방법이 아니고 stochastic gradient descent방법을 사용해서 한 스텝씩 update합니다.

Actor-Critic Policy Gradient

Monte-Carlo Policy Gradient 알고리즘을 다시 살펴보겠습니다.

Monte-Carlo Policy Gradient (REINFORCE)

- Update parameters by stochastic gradient ascent
- Using policy gradient theorem
- Using return v_t as an unbiased sample of $Q^{\pi_\theta}(s_t, a_t)$

$$\Delta\theta_t = \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) v_t$$

function REINFORCE

```

Initialise  $\theta$  arbitrarily
for each episode  $\{s_1, a_1, r_2, \dots, s_{T-1}, a_{T-1}, r_T\} \sim \pi_\theta$  do
    for  $t = 1$  to  $T - 1$  do
         $\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) v_t$ 
    end for
end for
return  $\theta$ 
end function

```

REINFORCE 알고리즘에서는 Return을 사용하기 때문에 Monte-Carlo 고유의 문제인 high variance의 문제가 있습니다. 또한 episode 자체가 길수도 있기 때문에 학습하는 시간이 오래 걸릴 수도 있습니다. 따라서 다음과 같은 아이디어를 낼 수 있을 것입니다. parameter를 하나 더 사용해서 action value function도 approximation하는 것입니다.

1. Actor & Critic

그러한 알고리즘을 actor-critic이라고 부르고 아래 그림을 통해 설명하도록 하겠습니다. Critic은 action value function을 approximate하는 w 를 update하고 Actor는 policy를 approximate하는 θ 를 update합니다. 따라서 w 와 θ 라는 두 개의 weight parameter를 사용해야 합니다.

- Monte-Carlo policy gradient still has high variance
- We use a **critic** to estimate the action-value function,

$$Q_w(s, a) \approx Q^{\pi_\theta}(s, a)$$

- Actor-critic algorithms maintain two sets of parameters
 - Critic** Updates action-value function parameters w
 - Actor** Updates policy parameters θ , in direction suggested by critic
- Actor-critic algorithms follow an *approximate* policy gradient

$$\nabla_{\theta} J(\theta) \approx \mathbb{E}_{\pi_{\theta}} [\nabla_{\theta} \log \pi_{\theta}(s, a) Q_w(s, a)]$$

$$\Delta \theta = \alpha \nabla_{\theta} \log \pi_{\theta}(s, a) Q_w(s, a)$$

이 Critic은 action value function을 통해 현재의 Policy를 평가하는 역할을 합니다. action을 해보고 그 action의 action value function이 높았으면 그 action을 할 확률을 높이도록 policy의 parameter를 update하는데 그 판단척도가 되는 action value function또한 처음에는 잘 모르기 때문에 학습을 해줘야하고 그래서 critic이 필요합니다.

action value function을 update하는 것은 chapter 8에서 봤던 것처럼 TD(0)를 사용하여 update 합니다. 아래는 action value function을 linear하게 approximation했을 경우입니다. DNN을 사용할 때는 이전에 배웠던 방법으로 바꿔서 사용하면 됩니다. TD(0)를 사용한 Actor-Critic알고리즘을 아래와 같습니다. Monte-Carlo PG때와는 달리 매 time step마다 update를 하는 것을 볼 수 있습니다. 또한 update할 때는 policy의 parameter와 action value function의 parameter를 동시에 update해줍니다.

Action-Value Actor-Critic

- Simple actor-critic algorithm based on action-value critic
- Using linear value fn approx. $Q_w(s, a) = \phi(s, a)^T w$
- Critic Updates w by linear TD(0)
- Actor Updates θ by policy gradient

```

function QAC
    Initialise  $s, \theta$ 
    Sample  $a \sim \pi_\theta$ 
    for each step do
        Sample reward  $r = \mathcal{R}_s^a$ ; sample transition  $s' \sim \mathcal{P}_{s,a}$ .
        Sample action  $a' \sim \pi_\theta(s', a')$ 
         $\delta = r + \gamma Q_w(s', a') - Q_w(s, a)$ 
         $\theta = \theta + \alpha \nabla_\theta \log \pi_\theta(s, a) Q_w(s, a)$ 
         $w \leftarrow w + \beta \delta \phi(s, a)$ 
         $a \leftarrow a', s \leftarrow s'$ 
    end for
end function

```

2. Baseline

여기까지 기본적인 Policy Gradient의 개념에 대해서 살펴보았는데 Actor Critic 말고 다르게 Variance 문제를 해결하는 것이 Baseline입니다. Q function 이후로 사용하고 있지 않은 State value function을 일종의 평균으로 사용해서 현재의 행동이 평균적으로 얻을 수 있는 value보다 얼마나 더 좋나라는 것을 계산하도록 해서 variance를 줄이는 것입니다. 즉, 지금까지 해왔던 것보다 좋으면 그 방향으로 update를 하고, 아니면 그 반대방향으로 가겠다는 것입니다.

Reducing Variance Using a Baseline

- We subtract a baseline function $B(s)$ from the policy gradient
- This can reduce variance, without changing expectation

$$\begin{aligned}\mathbb{E}_{\pi_\theta} [\nabla_\theta \log \pi_\theta(s, a) B(s)] &= \sum_{s \in \mathcal{S}} d^{\pi_\theta}(s) \sum_a \nabla_\theta \pi_\theta(s, a) B(s) \\ &= \sum_{s \in \mathcal{S}} d^{\pi_\theta} B(s) \nabla_\theta \sum_{a \in \mathcal{A}} \pi_\theta(s, a) \\ &= 0\end{aligned}$$

- A good baseline is the state value function $B(s) = V^{\pi_\theta}(s)$
- So we can rewrite the policy gradient using the **advantage function** $A^{\pi_\theta}(s, a)$

$$\begin{aligned}A^{\pi_\theta}(s, a) &= Q^{\pi_\theta}(s, a) - V^{\pi_\theta}(s) \\ \nabla_\theta J(\theta) &= \mathbb{E}_{\pi_\theta} [\nabla_\theta \log \pi_\theta(s, a) A^{\pi_\theta}(s, a)]\end{aligned}$$

이러한 advantage function의 사용은 variance를 상당히 개선시킬 수 있습니다. 하지만 아래와 같이 value function과 action value function을 둘 다 approximation해줘야 한다는 단점이 있습니다.

- The advantage function can significantly reduce variance of policy gradient
- So the critic should really estimate the advantage function
- For example, by estimating both $V^{\pi_\theta}(s)$ and $Q^{\pi_\theta}(s, a)$
- Using two function approximators and two parameter vectors,

$$\begin{aligned}V_v(s) &\approx V^{\pi_\theta}(s) \\ Q_w(s, a) &\approx Q^{\pi_\theta}(s, a) \\ A(s, a) &= Q_w(s, a) - V_v(s)\end{aligned}$$

- And updating *both* value functions by e.g. TD learning

하지만 다시 action value function이 immediate reward + value function이라는 것을 생각하면 아래와 같이 결국 value function 하나만 approximate해도 되서 critic에 parameter를 두 개 사용하는 비효율성을 개선할 수 있습니다.

- For the true value function $V^{\pi_\theta}(s)$, the TD error δ^{π_θ}

$$\delta^{\pi_\theta} = r + \gamma V^{\pi_\theta}(s') - V^{\pi_\theta}(s)$$

- is an unbiased estimate of the advantage function

$$\begin{aligned}\mathbb{E}_{\pi_\theta} [\delta^{\pi_\theta} | s, a] &= \mathbb{E}_{\pi_\theta} [r + \gamma V^{\pi_\theta}(s') | s, a] - V^{\pi_\theta}(s) \\ &= Q^{\pi_\theta}(s, a) - V^{\pi_\theta}(s) \\ &= A^{\pi_\theta}(s, a)\end{aligned}$$

- So we can use the TD error to compute the policy gradient

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta} [\nabla_\theta \log \pi_\theta(s, a) \delta^{\pi_\theta}]$$

- In practice we can use an approximate TD error

$$\delta_v = r + \gamma V_v(s') - V_v(s)$$

- This approach only requires one set of critic parameters v

지금까지는 evaluton으로 TD(0)를 사용했지만 이전에 배웠듯이 이 자리는 $TD(\lambda)$ 가 들어갈 수도 있고 eligibility trace가 들어갈 수도 있습니다. 위 방법은 variance가 낮은 대신에 one step만의 정보로 update하므로 bias가 높습니다. 이 문제에 대한 대책으로 TD와 MC사이의 방법인 $TD(\lambda)$ 를 사용할 수도 있다는 것입니다.