1.1.

안녕하세요? 인공지능의 기초 과목을 수강하게 될 수강생 여러분, 환영합니다.

저는 이 과목을 여러분께 가르쳐드릴 서울대학교 컴퓨터공학부의 김건희라고 합니다.

저는 서울대학교에서 인공지능 분야들인 컴퓨터비전, 자연어처리, 기계학습 등을 가르치고 있습니다.

근래에 ‘인공지능’이라는 단어는 매우 흔하게 쓰이고 있습니다. 그래서 일반인조차도 인공지능이란 단어를 여기저기서 쉽게 들을 수 있을 정도로 많이 사용되고 있는데요.

그에 비해서 인공지능의 정확한 정의나 혹은 그 기초적인 내용에 대해서 모르는 분들이 많이 계실 거라고 생각이 됩니다.

그래서 본 강좌를 통해서 여러분이 인공지능의 기초 개념을 확립할 수 있도록 많은 도움을 드리도록 최선을 다하겠습니다.

우선적으로 1주차에서 처음으로 말씀드릴 내용은 인공지능의 정의 및 소개에 대해서 말씀드리고자 합니다.

그러면 인공지능의 정의, 그러니까 과연 인공지능이라는 게 뭘까요?

사실 인공지능을 정의하는 건 쉽지 않은 일입니다. 왜냐하면 인공지능과 관련된 용어들 하나하나를 보면 쉽게 정의하기가 어려운 용어들이 참으로 많습니다.

단적인 예로 인공지능에서 intelligence, 그러니까 지능이란 부분을 어떻게 정의할까? 그것조차도 학계에서 하나의 완성된 정의가 존재하지 않을 정도입니다.

그래서 저는 여러분이 쉽게 혹은 재미있게 인공지능의 정의를 한번 생각해볼 기회를 드리고자

기존 인공지능의 여러 유명한 분들께서 만드신 정의, 말씀하신 정의들을 하나둘씩 소개시켜드리고, 한번 여러분도 같이 생각해보시면 좋을 것 같습니다.

첫 번째 위에 보이는 정의를 보시면, 핵심 단어라고 할 수 있는 게 ‘컴퓨터가 생각을 한다.’ 그런 단어와 ‘machines with minds. 마음을 가진 기계다.’ 이런 식으로 정의가 되어 있습니다.

mind, 마음이라는 용어 자체도 그렇게 쉽게 정의할 수 있는 내용이 아니죠.

그런데 여기서 중요한 거는 컴퓨터 그리고 우리의 intelligence와 상관이 깊은 mind, 이런 단어들이 공통적으로 나타남을 볼 수가 있습니다.

그 밑에 있는 또 다른 정의를 보면, 마찬가지로 우리 정신을 구성하는 여러 기제들이 있는데 그거에 대한 연구를 하는 학문입니다.

그런데 중요한 건 computational model을 만든다. 그러니까 계산 가능한, 컴퓨터에서 실행 가능한 정신적인 기제를 구현하는 학문이라고 보실 수 있습니다.

우리는 여러 가지 정신적인 기제들이 있는데, 회상을 한다든지 기억을 한다든지 혹은 추론을 한다든지.

이런 저희가 머리를 활용해서 할 수 있는 수많은 능력을 computational model로 구현하는 학문이라고 보실 수가 있겠습니다.

마찬가지로 또 하나의 정의를 계속 보여드리고 있는데요.

공통적으로 보시면 intelligence라는 단어와 함께 computational processes 혹은 computer, 이런 식으로 공통적으로 나오고 있음을 보실 수가 있습니다.

그리고 두 번째 밑에 있는 정의를 좀 보시면, 결국에 사람이 잘하는 일들을 컴퓨터가 할 수 있도록 하는 학문이라고 정의가 되어 있고요. 중요한 단어 중에 하나가 at the moment입니다.

그러니까 현재로서는 사람이 잘하는 일을 하는 게 목적일 수는 있지만, 인공지능이 계속 발전되면 그건 너무 당연한 일이 되고,

사람이 할 수 있는 능력을 훨씬 뛰어넘는 일들을 해야지만 인공지능이라고 불릴 수 있는 시대가 올 수도 있겠습니다.

마찬가지로 여러 가지 정의가 있고 그다음에 맨 마지막에 밑에 보이는 정의를 보면, 어떻게 보면 컴퓨터공학에서 가장 근래에 많이 쓰이는 정의라고 보시면 되겠습니다.

그러니까 computer science, 컴퓨터공학의 한 분야이고요. 특히 지능적인 행동을 자동화하는 것과 관련된 여러 연구 분야라고 이해하시면 되겠습니다.

그래서 지금까지 인공지능의 몇몇 유명한 정의에 대해서 말씀드렸는데요. 좀 더 깊게 하나하나씩 말씀을 드리도록 하겠습니다.

그래서 인공지능이 뭔지 정의하기 위해서는 일단 지능이 무엇인지를 알아야 하겠죠. 그래서 이 슬라이드에서는 과연 지능이 무엇일까? 그거에 대해서 말씀을 드리고 있는데요.

첫 번째 질문을 보시면 과연 지능에 대해서 전체를 아우를 수 있는 정의라는 게 과연 존재하느냐? 그런 질문입니다.

좀 전에도 말씀드렸다시피 지능에 대한 정의가 좀 논쟁적입니다. 그러니까 학계에서도 ‘지능이 뭐다.’ 이렇게 명확하게 정의를 내린다는 게 쉽지 않은 일이라는 걸 말씀드리고요.

따라서 지능이란 단어를 정의하기 힘들다면 당연히 인공지능을 정의하기는 더 힘들다고 할 수 있겠죠.

그런데 두 번째 bullet에 표현돼 있는 문장을 보시면, 이건 일반적인 영어사전에서 지능이란 단어를 찾았을 때 얻을 수 있는 지능에 대한 설명입니다.

그러니까 지식이나 여러 기술들을 습득하고 그걸 적용하는 능력이다. 그렇게 지능이라고 정의가 되어 있습니다.

사실 이 정의 자체는 매우 훌륭한 정의죠. 그렇긴 하지만 과연 이 하나의 문장으로 지능이라는 것을 완벽히 정의할 수 있느냐? 그런 것은 쉽지 않다는 얘기입니다.

그래도 하나 그나마 좀 좋다고 할 수 있는, 희망적이라고 할 수 있는 내용은 무엇이냐 하면,

이렇게 지능이 무엇인지를 하나의 완결된 문장으로 정의하기는 매우 어렵지만, 지능을 갖추기 위해서는 어떤 요소들이 있어야 되는지를 나열하는 것은 상대적으로 쉽습니다.

이런 holistic definition, 그러니까 전체를 아우르는 정의보다는 ‘지능을 갖추려면 이러이러한 능력이 있어야 된다.’ 그런 식으로 정의를 하는 건 상대적으로 쉽겠죠.

그래서 무엇을 이해하는 능력이나 추론을 하는 능력, 문제를 해결하는 능력, 혹은 학습을 하고, 상식을 가지고 있고, 일반화를 하고

이런 수많은 것을 갖추어야지만 지능적이라고 얘기하기는 상대적으로 쉽다는 얘기입니다.

그런데 그렇다고 해서 과연 어떤 요소들을 꼭 갖추어야지만 지능이라고 할 수 있는지 혹은 지능이 있다고 하면 어떤 요소를 꼭 갖추어야 되는지는 여전히 불분명하다고 할 수 있습니다.

일반적으로 지능이라는 얘기를 들어보면 사람들은 사람의 지능을 먼저 생각하게 됩니다.

하지만 우리가 쉽게 볼 수 있는 개미, 벌 이런 곤충들조차도 지능을 가지고 있고요. 그리고 또 그런 곤충들은 개개인의 지능이 있지만 그게 군집으로 모임으로써 군집지능을 보일 수가 있습니다.

그래서 지능이란 단어를 다양한 레벨로 정의할 수가 있고, 그런 측면에서 지능의 정의가 참으로 어렵다고 말씀드릴 수가 있겠습니다.

그러면 이제 좀 학문적인 얘기를 말씀드리도록 하겠습니다.

이제 인공지능이라는, 그러니까 ‘지능을 인공적으로 만들겠다. 컴퓨터가 가질 수 있는 지능을 만들겠다.’ 과학자들이 이런 목표를 가지고 있었는데요.

그런 목표를 어떻게 달성할지, 그러니까 인공지능을 어떤 방식으로 달성할지에 대해서 크게 보면 4가지 관점이 이제 존재합니다.

맨 위에 보시다시피 4가지 관점이 존재하는데, 보시면 아시겠지만 왼쪽과 오른쪽은 차이가 humanly라는 단어가 왼쪽에 있고요. 오른쪽에는 rationally라는 단어가 있습니다.

그리고 위, 아래를 보시면 Thinking이라는 단어가 있고, Acting이라는 단어가 있습니다. 그러니까 인공지능을 사람이 어떻게 생각하는지를 기준으로 구현할 수도 있는 거고요.

혹은 대각선, 반대쪽을 보시면 지능이 이성적으로 행동할 수 있는 걸 구현하는 방식으로 가겠다. 그렇게 보실 수도 있겠습니다.

그래서 각각에 대해서 깊이 말씀을 드리도록 하겠습니다. 첫 번째로 humanly와 rationally를 비교해볼 수가 있겠는데요.

humanly는 말 그대로 ‘사람 같은’입니다. 그리고 rationally는 ‘이성적’이라는 건데요.

그러니까 우리가 만약 인공지능을 사람처럼 구현하겠다. 그렇게 해보실 수도 있고, 혹은 이성적인 agent를 하나 만들겠다. 그렇게 구현할 수 있다는 겁니다.

그래서 사람 같은 경우에는 사람이 가지고 있는 인지 모델을 이해하는 게 필요할 수가 있겠고요.

rationally라는 건 말 그대로 ‘이성적인’ 그러니까 여러분이 경제학에서 배우시는, 경제학에서는 기본적으로 모든 경제참여주체가 이성적인 행동, 판단을 한다고 가정하고 모든 모델을 설계하는데요.

그런 관점에서 보면 이성적이라는 건 각각의 경제 주체들이 자기가 도달하고자 하는 utility, performance measure가 존재하게 되고요.

그런 performance measure을 최대화할 수 있는 식으로 행동을 한다. 그렇게 모델링 할 수 있겠죠.

그렇지만 사람은 항상 이성적인 행동만 하는 게 아니라, 사람은 매우 많은 경우에 비이성적인 행동도 함께 합니다.

그래서 만약에 여러분이 인공지능을 rationally 관점에서 구현한다고 하면, 이제 말 그대로 ‘이성적인 agent를 만들겠다.’ 그렇게 인공지능을 바라보시는 것이고요.

humanly라고 본다면 ‘그런 걸 떠나서 사람처럼 행동하고 사람처럼 생각할 수 있는 인공지능을 만들겠다.’ 그렇게 보시면 되겠습니다.

그래서 humanly를 좀 더 살펴보면, 사람은 놀라운 능력들이 있는데요. 이게 항상 이성적인 것과 연결되지는 않습니다.

하지만 사람들은 상식을 가지고 있습니다. 상식은 누가 여러분에게 명시적으로 가르쳐주지 않은 내용임에도 불구하고 여러분이 그냥 아는 것들이 있습니다.

그리고 사회적인 행동을 보이고요. 사회적인 행동이라는 게 꼭 이성에 따라서 하는 건 아닐 수도 있고요.

그다음에 사람 같은 경우에 전문가 지식들이 있고, 여러 가지 문제해결 능력이 있습니다. 퍼즐을 푼다든지 게임을 한다든지 하는 능력이 있겠죠. 그런 것들은 humanly 관점에서 보실 수가 있겠습니다.

그럼 이번 슬라이드에서는 위와 아래의 차이에 대해서 말씀을 드리겠습니다. 위는 Thinking이었고요. 아래는 Acting이라고 되어 있는데요.

그것은 어떤 관점이냐 하면, 인공지능을 구현하는 데 있어서 사람이 어떻게 생각하는지를 이해하고 그것대로 구현하는 방법이 위에 보이는 관점이고요.

Acting이라는 건 사람이 어떻게 지능적인 행동을 하는지 혹은 어떤 생각을 통해서 그런 지능적인 행동을 하는지는 관심이 없고,

사람이 하는 지능적인 행동 자체만을 모사하자. 그런 관점이라고 보시면 되겠습니다.

그래서 지능이라는 건 아까 말씀드린 대로 정의하는 게 매우 어려운데, 지능적인 행동이라는 건 상대적으로 정의하기가 쉽습니다.

하나의 예로 요즘 여러분이 많이 사용하고 있는 기계번역을 예로 들면, 여러분이 기계번역을 어떻게 하는지,

여러분 머릿속에서 영어 문장을 봤을 때 그걸 한국어 문장으로 어떻게 번역하는지 그 과정을 알아내는 건 매우 힘들지만,

여러분이 어떤 함수, 프로그램을 정의해서 ‘이 프로그램은 영어 문장에 들어오면 무조건 한국어 문장이 나오는 프로그램이다.’ 이런 식으로만 정의하면 상대적으로 쉽겠죠.

그러니까 현재 기계번역기들을 보면, 여러분도 깜짝 놀랄 정도로 성능이 매우 좋습니다. 그런데 놀라운 점은 기계번역기들은 언어를 전혀 이해하지 못합니다.

그럼에도 불구하고 기계번역기가 아주 놀라운 성능을 보이는 이유는 영어 문장과 그에 상응하는 한국어 문장 쌍을 1억 개 정도 모아서, 1억 개가 더 될 수 있죠.

하여튼 가능한 한 최대한 많이 모은 다음에 걔네들 사이의 확률적, 통계적인 상관관계만을 배우는 것입니다.

그러니까 ‘영어 단어가 이런 식으로 나오면 통계적으로 한국어 단어가 이런 식으로 나오더라.’

그런 정보들을 예제로부터 학습을 해서 그대로 보일 뿐이지, 영어가 주어졌을 때 그 영어 문장을 완벽히 이해하고, 그 이해한 것을 바탕으로 한국어로 번역하는 식으로 진행되지 않는다는 것입니다.

그래서 중요한 점은 Thinking humanly 관점에서 구현하는 건 매우 어렵지만, Acting humanly 관점에서 인공지능을 정의하는 건 상대적으로 쉽습니다.

왜냐하면 그 안에 어떤 일들이 벌어지는지 우리는 관심을 가질 필요가 없고, 인터페이스에서 우리가 영어 문장을 주면 한국어 문장을 내놓는 아주 지능적인 행동만 제대로 할 수 있다면

그 관점에서 이제 인공지능을 구현할 수 있기 때문에 훨씬 상대적으로 쉽다고 할 수 있습니다.

대표적인 예가 Turing test. 여러분이 어떤 컴퓨터가 인공지능이 있는지 없는지를 판단하는 가장 유명한 테스트 중에 하나가 Turing test라는 걸 많이들 아실 텐데요.

그것도 Thinking humanly 관점이 아니라 Acting humanly 관점에서 인공지능을 정의하고 구현했다고 보시면 되겠습니다.

그래서 Turing test에 대해서 깊게 말씀을 드리면, 아시다시피 Alan Turing이라는 영국의 암호학자겸 어떻게 보면 컴퓨터공학의 아버지 같은 아주 위대한 분이라고 할 수 있습니다.

그분이 처음에 인공지능이라는 걸 정의하려고 노력을 많이 했는데, 지금 좀 전에 말씀드렸다시피 인공지능이라는 것의 정의가 매우 어렵다는 걸 이제 이분도 느끼신 거죠.

그래서 지능이라는 게 너무 정의하기 어려우니까 지능적인 행동 관점에서 인공지능을 한번 정의해보자. 그렇게 시작됐다고 보시면 되겠습니다.

아시다시피 Turing test에서 조사관이 한 명 있습니다. 그리고 닫힌 두 개의 방이 있고요. 이 두 개의 방에 컴퓨터가 있는지 사람이 있는지 조사관은 모릅니다.

그래서 제가 만약에 조사관이 돼서 양쪽 방에 질문을 했을 때 그쪽으로부터 답을 얻을 수가 있겠죠.

그런데 제가 만약에 그 두 답을 듣고도 누가 컴퓨터고 누가 사람인지 구별할 수 없다면, 그럼 그 컴퓨터는 Turing test를 통과한 거고,

그렇다면 그 Turing test는 인공지능을 갖추었다고 얘기할 수 있다고 정의가 된 것입니다.

좀 전에 말씀드린 바와 같이 여기서는 가려진 문 뒤에 어떤 entity가 있고,

그 entity로부터 대답을 얻어서 판단을 하는 것이기 때문에 이 컴퓨터가 사람처럼 행동만 하면 그 안에 어떻게 구현이 되어 있든지 간에

사람처럼 생각을 하든 사람처럼 생각을 하지 않든 지능적인 행동만 보이면 인공지능을 갖추었다고 정의할 수 있다. 이렇게 정의됐다고 보실 수가 있겠습니다.

그래서 여기 보시면 몇 가지 Turing test의 예들이 있는데요. 첫 번째 예는 ‘Forth Bridge를 주제로 해서 시를 하나 써 달라.’

만약에 제가 이런 질문을 했다고 했을 때 컴퓨터가 ‘나는 빼 달라. 나는 시를 잘 못 쓴다.’ 이런 식으로 대답을 한다면, 사람이라고 생각할 수도 있겠죠.

혹은 밑에 있는 예제처럼 ‘두 개의 큰 수를 더하라’라고 명령을 주었는데 30초 동안 쉬었다가 답을 내놓는다고 하면, 마찬가지로 사람과 구별할 수 없는 행동을 했다고 할 수 있겠죠.

그러니까 이 예를 보시면 알 수 있다시피 Turing test는 Acting humanly 관점에서 인공지능을 구현했다고 보시면 되겠습니다.

사실 컴퓨터에게 있어서 두 개의 큰 수를 더하라는 건 매우 쉬운 일임에도 불구하고 여기서 보시다시피 일부러 30초 쉬고 답을 얘기하는데요.

그것은 어디까지나 사람을 모방하기 위해서, 사람과 똑같은 행동을 하기 위해서 그런 기제들이 들어가 있다고 보실 수 있겠습니다.

그래서 Turing test에 대해서 좀 정리를 해보자면, Turing test는 Acting humanly 관점에서 정의가 됐다고 보실 수 있고요.

Acting rationally 관점에서 정의됐다고 할 수는 없습니다. 왜냐하면 좀 전에 말씀드렸다시피 ‘큰 숫자 두 개를 더해라.’

그 목적을 최대한 잘 달성하는 건 정확하고 빠르게 푸는 게 가장 성능이 높다고 할 수 있겠죠.

그러면 당연히 그 컴퓨터가 rationally, 이성적으로 행동한다면 그 두 개의 performance measure를 최대화할 수 있도록 정확하고 빠르게 풀어야 되는데, 그렇지 않다는 걸 보실 수가 있겠죠.

그래서 Turing test라는 건 Acting humanly 관점에서 인공지능을 정의했다고 보시면 되겠습니다.

그래서 Turing test를 좀 자세히 보시면, Turing test는 인공지능을 갖춘 컴퓨터의 행동을 보는 것입니다.

그래서 어떻게 보면 상대적으로 인공지능 정의와 판단이 되게 간단해졌고 그리고 또 객관적이라고 보실 수 있겠습니다.

그리고 좀 전에도 말씀드렸다시피 Turing test라는 게 인공지능을 정의하기 위한 매우 훌륭한 테스트임은 분명하지만,

Turing test를 통과했다고 해서 무조건 얘가 지능을 갖춘 컴퓨터라고 말하기는 매우 어려운 그런 정의라고 할 수 있고요.

그래서 Turing test를 통과하기 위해서는 여러 능력을 갖추어야 됩니다. 첫 번째는 여러 지식을 컴퓨터가 내부적으로 표현할 수 있어야 되고요.

그다음에 내가 가지고 있는 지식들을 바탕으로 내 지식 안에 없는 질문이 들어왔다고 했을 때는 추론을 통해서 그에 알맞은 대답을 만들어야 되고요.

또 학습을 통해서 새로운 지식을 계속 습득해야 되고 그리고 조사관이 얘기하는 질문을 이해하거나 혹은 그 질문을 이해해서 그걸 말로 표현할 수 있는 능력이 필요합니다.

그리고 원래의 Turing test 정의에서는 image, 그러니까 사진, 시각 정보에 대한 질의응답은 포함되어 있지 않지만, 그것까지 확장해볼 수 있겠죠.

그렇다고 하면 시각적인 이해능력이라든지 혹은 Robotics와 같이 제가 환경을 움직일 수 있는 능력까지도 필요하다고 할 수 있겠습니다.

그래서 지금 말씀드린 이런 component들이 다른 한편으로 보면 인공지능에서 매우 중요하게 다루는 주제들이라고 보실 수 있겠습니다.

몇 가지 중요한 토픽에 대해서 이 슬라이드에서 말씀을 드리도록 하겠습니다.

여기서 말씀드릴 주제가 인공지능의 주제를 모두 아우른다고 말씀드릴 수는 없고요. 그 대표적인 것만 보여드리고 있다고 생각하시면 되겠습니다.

크게 보시면 Sensing, Thinking, Acting 과정이 있습니다. Sensing이라는 건 말 그대로 감각을 이용해서 환경을 이해하고 환경을 측정해내는 거고요.

생각은 여러 가지 사고과정을 말하고요. Acting은 제가 생각하고 사고한 내용을 표현하는 방법이라고 할 수 있겠죠.

Sensing 관련돼서는 Computer vision. Computer vision은 시각지능에 대한 것입니다. 사람이 눈으로 할 수 있는 엄청나게 다양한 일들을 기계도 할 수 있게 만들겠다.

그게 Computer vision의 목적이라고 할 수 있는데, 그게 Sensing의 중요한 영역 중에 하나고요.

그리고 음성을 인식한다든지 자연어, 사람이 사용하는 언어를 이해하는 것들이 Sensing 영역에 있다고 할 수 있겠고요.

또 Thinking 영역에는 좀 전에 말씀드린 지식을 표현하는 것, 문제의 해결능력, 계획 그리고 제가 예전의 경험으로부터 새로운 지식을 얻어내는 학습과정 이런 것들이 포함된다고 할 수 있겠습니다.

Acting은 말 그대로 환경을 향해 표현하는 것입니다. 그래서 Robotics도 이제 Acting 영역이라고 할 수 있겠고요. 혹은 음성 발화를 하는 것도 Acting의 영역이라고 할 수 있겠습니다.

그래서 보시면 아시겠지만 인공지능이라는 분야는 매우 큰 분야입니다.

인공지능의 요소라고 지금 나열돼 있는 분야들이 각각 하나하나가 아주 큰 연구 분야이고, 각각은 대학교 같은 데서는 한 학기에 걸쳐 가르쳐야 될 정도로 매우 방대한 분량입니다.

그래서 인공지능이라는 학문은 쉽게 사람들이 사용을 하지만, 너무 큰 범위이고 정의도 하기 쉽지만은 않은 학문 분야라고 생각하시면 되겠고요.

그래서 되도록이면 여러분도 ‘인공지능’이라는 단어보다는 좀 더 정확한 단어를 사용하시는 게 좋다. 그렇게 추천을 드리고 싶습니다.

한 가지 예를 들어서 이 방에 만약 요크셔테리어가 있다고 해보세요.

그럴 경우에 어떤 사람은 저 요크셔테리어를 보고 “여기 네 발 달린 짐승이 있네.” 이렇게 얘기하는 사람이 있고,

또 한 사람은 “저기 요크셔테리어가 있다.” 이렇게 얘기하는 사람이 있다고 하면, 여러분도 다 인정을 하시겠지만 두 번째 사람이 더 전문적이라고 할 수 있겠죠.

인공지능이라는 단어가 요즘 많이 오용되고 있는 혹은 너무 필요 이상으로 자주 사용되는 경향이 좀 있는데,

많은 경우에 우리가 인공지능이라고 얘기하는 대다수의 것들이 음성인식을 말하는 경우가 꽤 많습니다.

그럴 경우에는 ‘인공지능이다.’ 이런 용어보다는 ‘음성인식을 하는 기계’ 이런 식으로 얘기하시는 게 더 전문적이고 더 정확하다고 말씀드릴 수가 있겠습니다.

그러면 몇 가지 중요한 토픽에 대해서 아주 간략하게만 말씀을 드리도록 하겠습니다.

인공지능의 큰 분야 중에 하나가 탐색에 대한 것입니다. 탐색은 특히 게임을 분석하는, 게임 인공지능에 아주 많이 사용되는 것이고요.

얼마 전에 여러분이 많이 경험해보셨다시피 바둑도 이런 탐색 과정을 통해서 지능적인 게임 플레잉 능력을 보였다고 보시면 되겠습니다.

사실 바둑은 매우 복잡한 문제이고요. 그래서 이 슬라이드에서는 그것보다 훨씬 쉬운 Tic-tac-toe라는 게임에 대해서 말씀을 드리는데요.

Tic-tac-toe라는 건 3×3 그리드가 있고요. 거기서 ○를 칠하는 사람과 X를 칠하는 사람, 두 개의 플레이어로 나눕니다.

그래서 X라고 하는 사람이든 ○라고 하는 사람이든 누군가가 먼저 가로로 혹은 세로로 혹은 대각선으로 모두 X 혹은 모두 ○를 칠할 수 있다고 하면 그 사람이 이기는 게임이라고 할 수 있겠습니다.

그래서 그런 게임을 할 수 있는 인공지능을 탐색 과정을 통해서 구현을 한다고 하면,

처음에 3×3 그리드가 주어졌을 때 X라는 사람이 먼저 게임을 플레이한다고 했을 때 X가 할 수 있는 경우의 수가 9가지가 있겠죠.

그럼 각각의 경우에 대해서 또 그다음에 ○라는 사람이 칠할 수 있는 영역, 가능성이 8개가 있겠죠.

그런 식으로 모든 경우에 대해서 다 검색을, 탐색을 해서 ‘내가 어디에 뒀을 때 이길 확률이 제일 높다.’ 그걸 판단해서 그다음에 제가 두어야 될 수를 결정할 수 있겠죠.

그래서 기본적으로 알파고도 이와 같은 아이디어로 구현이 됐다고 보실 수가 있습니다.

물론 그 경우에는 탐색 공간이 너무나도 크기 때문에 다른 여러 가지 딥러닝 기술이 활용되긴 했지만, 기본적으로는 탐색을 통해서 게임의 인공지능을 구현했다고 보실 수가 있겠습니다.

그다음에 좀 전에 말씀드린 지식 표현이라든지 추론 과정 혹은 계획 과정,

그다음에 여러분도 마찬가지겠지만 여러분이 어떤 계획을 할 때 많은 경우에는 여러분이 미처 고려하지 못한 불확실성을 통해서 여러분의 계획이 많이 틀어지는 걸 아실 수가 있을 텐데요.

인공지능에서도 마찬가지입니다. 그래서 이런 불확실성을 어떻게 인공지능에서 처리해야 될지도 매우 중요한 인공지능의 분야라고 할 수 있겠습니다. 또 학습도 마찬가지고요.

그다음에 또 하나 재미있는 것은 multi-agent systems라고 해서 예를 들면 주식시장이라고 하면 ‘주식시장은 여러 이벤트를 통해서 다음의 주가를 예측한다.’ 이렇게 바라볼 수도 있지만,

주식 투자를 하는 수많은 회사들과 개인들이 서로 어떻게 보면 하나의 매우 큰 게임을 하고 있다고 보실 수가 있습니다.

그런 식으로 주식시장을 어떻게 잘 예측할까를 game theory 관점에서도 보실 수가 있는데요. 그런 것도 매우 중요한 인공지능의 분야라고 할 수 있겠습니다.

그러면 여러분이 인공지능 관련해서 많이 듣게 되는 것들에 대해서 깊게 말씀을 드리도록 하겠습니다.

여러분이 많이 듣고 있는 얘기 중에 하나가 ‘강 인공지능’ 혹은 ‘약 인공지능’이라는 단어일 것입니다. 그래서 그 둘의 차이가 무엇인지 간단히 살펴보도록 하겠습니다.

우선 약 인공지능(Weak AI)에 대해서 말씀을 드리도록 하겠고요. 이것은 강 인공지능의 반대말이라는 건 쉽게 아실 수 있겠고요.

그래서 약 인공지능은 applied AI 혹은 narrow AI라고 합니다. ‘약 인공지능을 갖춘 기계다.’ 이것은 사람이 어떤 task를 명확히 정의해놓고 얘가 어떤 행동을 해야 될지 미리 다 프로그래밍 합니다.

단지 약 인공지능 기계들은 주어진 상황에서 어떤 행동을 해야 될지 스스로 결정해야 되는 거겠죠.

물론 어떤 환경이고, 어떤 task고, 어떤 action이 가능한지는 사람이 다 프로그래밍을 하지만,

그 주어진 상황에서 어떤 판단을 하고 행동을 해야 될지는 인공지능 스스로 판단할 수 있도록 하는 게 약 인공지능이라고 할 수 있고요.

그래서 지금 여러분이 보는 인공지능은 다 약 인공지능이라고 보실 수가 있겠습니다.

여러분이 알파고가 매우 지능적인 프로그램이라는 건 아시겠지만, 그렇다고 해도 이건 여전히 약 인공지능 영역에 있습니다.

왜냐하면 수많은 프로그래머들이 ‘바둑판은 19×19의 그리드 안에서 검은 돌과 흰 돌이 번갈아가면서 두고, 어떤 경우에는 이기고 어떤 경우에는 진다.’

그런 것들을 다 프로그래밍 해놓은 상태에서 각 주어진 상황에서 어떤 수를 둬야 될지를 학습한 기계라고 할 수 있기 때문에

바둑을 위해서 개발된 알파고가 갑자기 장기를 둔다거나 하는 건 도저히 상상할 수 없는 일이겠죠. 그래서 알파고는 약 인공지능이라고 할 수 있겠고요.

그리고 또 공장에서 조립하는 로봇들도 약 인공지능의 영역이라고 보실 수 있겠습니다.

그에 반해서 강 인공지능은 사람의 지능에 도달하거나 혹은 그걸 뛰어넘는 인공지능이라고 볼 수 있겠고요.

강 인공지능에서는 더 이상 사람을 모방하지 않고 그 자신의 지능을 갖추게 됩니다. 다른 말로는 General AI, 일반적인 인공지능이라고 얘기를 합니다.

그래서 많은 경우에 사람이 가지고 있는 특성들, 의식이라든지 예를 들면 자유의지, 자각 이런 것들을 갖추어야지만 강 인공지능이라고 할 수 있겠죠.

그래서 현실 세계에서는 강 인공지능은 거의 없고요. 강 인공지능을 여러분이 볼 수 있는 건 수많은 영화에서나 가능하다고 할 수 있습니다.

그 수많은 SF 영화를 보시면, 항상 기계 혹은 안드로이드가 고민하는 게 ‘나는 누구인가?’ 이런 것들로 고민하는 것들을 많이 볼 수가 있는데요.

예를 들면 「공각기동대」라는 영화를 보시면, 「공각기공대」의 주인공 로봇이 결국에 하는 일은 경찰이 하는 일입니다.

그래서 경찰로봇을 만든 건데, 그러면 경찰로봇을 만들겠다고 하면 경찰이 해야 되는 일들을 잘 정의해서 그것만 잘할 수 있도록 구현해서 경찰로봇을 만들면 되겠죠.

그런데 그 경찰로봇에게 자의식을 주고 혹은 그 경찰로봇이 ‘내가 누구인가?’ 이런 걸 생각할 기능을 줄 필요는 전혀 없다고 생각하실 수가 있겠습니다.

사실 현실 세계에서는 강 인공지능을 꼭 구현해야 된다는 필요성은 상대적으로 적고요. 대신에 약 인공지능이라고 한다면, 영화에서 약 인공지능만 나온다고 하면 되게 재미가 없겠죠.

그래서 좀 보수적으로 보는 게 좋을 수도 있을 것 같습니다.

많은 경우에 현실 세계에서 인공지능을 갖춘 기계라고 선전하는 것들을 많이 보면, 강 인공지능을 갖춘 것처럼 사람을 호도하는 경우도 매우 많은데요.

많은 경우에 그건 다 정해진 시나리오에 대해서 일일이 사람들이 프로그래밍을 해놓은 것이고, 실제로 강 인공지능은 우리가 쉽게 볼 수 없다고 이해하시면 되겠습니다.

그다음에 또 여러분이 많이 듣는 단어 중에 하나가 Singularity입니다. 특이점이라는 단어고요.

이 특이점이라는 건 그래프에서 보시면 아시겠지만, 사람의 지능은 점점 진화를 통해서 높아지고 있습니다. 그렇지만 진화되는 속도는 매우 낮다고 할 수 있겠죠.

그에 반해서 인공지능의 속도는 매우 빠르기 때문에 어떤 지점이 있을 것이다. Singularity, 특이점이라는 지점이 있을 것이다.

그 지점에서는 인공지능이 사람의 지능을 뛰어넘을 것이다. 이렇게 보실 수가 있겠습니다.

한 가지 단적인 예를 들어 지능이 서로 다른 두 개의 존재가 있다고 하면, 재미있는 것은 지능이 낮은 존재는 지능이 높은 존재가 하는 행동이나 말을 전혀 이해하지 못한다는 것입니다.

예를 들면 이 강의실에 고양이가 한 마리 있다고 하면, 고양이는 저희가 하는 말, 행동을 전혀 이해하지 못합니다. 왜냐하면 고양이는 저희보다 지능이 매우 낮기 때문에.

그래서 특이점이 넘어가면 인공지능이 하는 행동을 사람이 고양이가 된 듯이 전혀 이해를 못하는 현상이 일어나지 않을까? 그게 많은 분이 가지고 있는 특이점에 대한 두려움이라고 할 수 있습니다.

약간이나마 저희가 특이점을 경험한 거라고 할 수 있다면, 알파고 때를 생각해보시면 되겠죠.

알파고 때 처음에는 많은 전문가들이 이세돌 9단이 쉽게 알파고를 이길 것이라고 예측했었고요.

그래서 처음에 알파고가 사람이 생각하는 것과 이상한 수를 두면, ‘쟤가 실수를 했구나.’ 이런 식으로 판단했는데, ‘알고 봤더니 그게 더 나은 수였다.’ 이런 게 그 후에 밝혀졌죠.

그게 어떻게 보면 바둑이라는 특별한 게임에 대해서 저희가 Singularity, 특이점을 겪었다고 이해하실 수도 있겠습니다.

그래서 많은 전문가들이 과연 특이점이 올 것이냐, 안 올 것이냐? 그런 것들에 대해서 많은 논쟁을 하고 있는데요.

제 개인적인 생각으로 narrow AI 관점. 우리가 어떤 task를 매우 잘 정의하고 그 task만 하는 인공지능을 만들었다. 그렇게 했다면 우리는 Singularity, 특이점을 겪을 것입니다.

그러나 일반적인 지능 혹은 각 인공지능 관점에서의 Singularity는 아마 쉽게 오지 않을 거라고 예측해볼 수가 있겠죠.

그래서 많은 분들이 Singularity라는 단어가 어디에서 왔는지 궁금해 하실 수도 있어서 제가 재미있는 그림을 하나 소개시켜드리고자 합니다.

Singularity라는 이름이 어디서 왔느냐 하면, 물리학의 블랙홀에서 왔습니다. 그래서 블랙홀이 있으면, 블랙홀은 아시다시피 빛조차도 빠져나올 수 없는 영역이죠.

그래서 블랙홀을 중심으로 해서 그 주변에 event horizon이라고 있습니다.

event horizon 안에 있으면 빛조차도 블랙홀 안으로 빨려들어 가는 거고요. 그리고 event horizon 밖에 있으면 이제 거기서부터 탈출을 할 수 있는 거죠.

그러니까 event horizon 안쪽은 빛조차도 나올 수가 없고, 따라서 관측도 불가능하기 때문에 이 안은 미지의 영역이라는 것입니다.

그래서 여기서부터 Singularity라는 단어가 유래했다고 말씀드릴 수 있겠습니다.

그래서 지금까지 인공지능의 소개에 대해서 간단히 말씀을 드렸습니다.

말씀드린 바와 같이 인공지능이 워낙 넓은 분야고 정의하기 어려운 측면이 있기 때문에 그렇게 짧은 시간 안에 소개시켜드리는 게 쉬운 일은 아니지만,

여러분이 오늘 강의를 통해서 좀 더 관심을 가지고 이것저것 많이 살펴보시면 그 인공지능에 대한 이해를 좀 더 넓힐 수 있다고 생각을 합니다.

그러면 다음 강좌에서는 인공지능의 간단한 역사에 대해서 소개시켜드리도록 하겠습니다.

1.2.

그럼 인공지능 발전 역사에 대해서 간단히 되짚어보는 시간을 갖도록 하겠습니다.

사실 사람의 지능을 갖춘 기계를 만들겠다는 인간의 꿈은 아주 오래 된 꿈이긴 하지만,

여기서 지금 소개시켜드리는 몇 가지 예제들은 그 중에서도 재미있고 여러분이 아시면 좋을 법한 것들만 선택해서 간단히 말씀드리도록 하겠습니다.

첫 번째로 소개해드리고 싶은 건 Mechanical Turk이라는 기계입니다. 여기서 보시면 아시겠지만, 기본적으로 이 기계는 체스를 두는 게임이고요.

여기서 중요한 단어가 하나 있습니다. fake라는 단어죠. 이건 ‘사기’라는 뜻입니다.

실제로는 사람이 그 안에 들어가서 체스를 뒀는데, 이걸 발명한 사람들은 ‘사람을 뛰어넘는 체스 기계가 나왔다.’ 이렇게 홍보를 하고 알려진 것이죠.

그래서 여기서도 보시다시피 장장 84년 동안 이게 체스를 두는 인공지능 기계로서 활약을 하게 됐고요. 그래서 나폴레옹이라든지 유명한 사람들을 실제로 이긴 Mechanical Turk이 되겠습니다.

사실 Mechanical Turk은 아는 분만 알고 잘 모르는 분들이 대다수인 기계였는데, 근래에 Amazon Mechanical Turk이라는 플랫폼이 나왔는데, 그거와 함께 다시 조명을 받게 되었습니다.

Amazon Mechanical Turk은 일종의 크라우드소싱을 위한 플랫폼인데요. 그러니까 제가 예를 들어 아주 간단한 작업을 여러 개 하고 싶다.

그런데 그게 사람을 통해서 하고 싶다고 했을 때 제가 만약에 그 task를 Amazon Mechanical Turk에 올리면 수많은 사람이 제가 올려놓은 task를 보고 이제 수행을 하는 것입니다.

그러면 task를 하나 수행할 때마다 제가 예를 들어서 10센트를 주겠다.

그런 식으로 해놓으면 여기저기서 수많은 사람이 그 크라우드소싱 task에 참여를 하는 그런 플랫폼으로 아주 유명한 서비스가 되겠습니다.

그거와 더불어 다시 ‘Mechanical Turk의 이름이 어디서 유래됐느냐?’ 그런 식으로 많은 사람들의 관심을 얻게 되었습니다.

그리고 인공지능의 역사에서 빼놓을 수 없는 게 좀 전에 소개시켜드린 Turing test였죠.

Turing test는 어떻게 하면 인공지능을 정의할 수 있을까? 기계가 지능을 갖추었다는 걸 어떻게 정의할 수 있을까? 그걸 정의하는 과정에서 나온 아이디어고요.

그리고 실제로 Turing test는 인공지능 테스트의 아주 대표로서 많은 일반인들도 알고 있는 내용입니다.

얼마 전에 「이미테이션 게임」이라는 Alan Turing의 일대기를 다룬 영화가 개봉된 적이 있었습니다.

여기서 Alan Turing은 주로 암호학자, 그러니까 2차 세계대전 과정에서 독일군의 암호를 푼 사람으로서, 그런 수학자로서 알려졌는데요.

물론 암호학자로서 아주 뛰어난 분인 동시에 컴퓨터공학의 founding father로서의 위치도 같이 가지고 있습니다.

실제로 Turing test를 Alan Turing은 ‘imitation game’이라고 이름을 붙였습니다. 그런데 이후에 ‘imitation game’이라는 테스트가 ‘Turing test’로 이름이 바뀌어서 사람들에게 알려진 것이죠.

그리고 인공지능에서 아주 중요한 task 중에 하나가 음성인식입니다. 그래서 음성인식을 개발하기 위한 노력은 아주 많이 있었고요.

그래서 그 중에 대표적인 게 Audrey이라는, 그러니까 미국의 전화국 회사죠? Bell Lab에서 개발한 Audrey이라는 시스템이 있고요.

이것의 원래 목적은 사람들이 전화를 통해서 전화번호를 얘기할 때 그걸 어떻게 하면 자동인식을 할 수 있을까? 그 작업을 하기 위해서 개발된 시스템이고요.

62년도에는 Shoebox라는 음성인식 시스템이 16단어를 인식할 수 있었다.

그리고 이후에는 CMU. CMU는 Carnegie Mellon University라고 미국의 인공지능으로 유명한 대학입니다.

거기서 만든 Harpy system 같은 경우에 1000개 정도의 단어를 인식할 수 있는 시스템을 만들었다고 알려져 있습니다.

그리고 1956년에 Dartmouth 대학에서 열린 workshop이 AI 역사에서 아주 중요한 위치를 차지하고 있습니다.

이 workshop이 중요한 이유는 어떻게 보면 인공지능이라는 학문이 처음으로 태동하게 된 일종의 학술회의였다고 보시면 되겠습니다.

첫 번째 보시면 Marvin Minsky, John McCarthy 이런 이름이 쭉 나오는 걸 보실 수 있을 텐데요.

일반인 분들은 이 이름이 생소하실 수 있겠지만, 각각 이분들이 인공지능 역사에서 아주 핵심적인 역할을 하신 분입니다. 이 중에서는 노벨상을 수상하신 분들도 계시고요.

그래서 이 workshop에서 ‘이제 인공지능을 한번 연구하자. 그리고 인공지능을 연구하기 위해서는 이러이러한 분야를 정의해서 한번 해보자.’

그런 식으로 어떻게 보면 인공지능이라는 학문 자체가 처음으로 시작된 workshop이라고 이해하시면 되겠습니다.

그리고 인공지능 연구자들 사이에서 되게 오랫동안 관심 있게 연구된 주제가 체스 게임입니다. 59년도에 Arthur Samuel이라는 분이 alpha-beta pruning이라는 알고리즘을 제안하였습니다.

이건 기본적으로 체스를 tree search 형태로 문제를 해결해나가는데, tree search를 어떻게 하면 효율적으로 할까? 그거에 대한 알고리즘이었고요.

이분의 알고리즘을 통해서 개발된 체스 시스템 같은 경우 그 당시에 잘한다고 할 수 있는 아마추어 레벨까지는 도달했다고 보실 수가 있고요.

그리고 97년도에 IBM에서 DeepBlue라는 체스 프로그램을 만들었고요. 이게 그 당시에 인간 체스 챔피언을 이긴 걸로 유명해졌죠. 이거에 대해서는 이후에 다시 다루도록 하겠고요.

그리고 2007년도에 Schaeffer 교수라는 분이 Checkers was solved라는 페이퍼를 출판하였는데요.

이 페이퍼가 유명해진 이유는 체스라는 게임을 수학적으로 완전히 분석을 한 첫 번째 논문이었다는 의미에서 많은 각광을 받았습니다.

그러니까 체스라는 게임이 우리가 잘 모르는 블랙박스 같은 게임이 아니라, 수학적으로 완전히 분석 가능한 게임이라는 게 밝혀졌다고 보시면 되겠고요.

그리고 또 하나 재미있는 인공지능 역사에서 중요한 milestone 중에 하나라고 한다면, Shakey라는 로봇이라고 할 수 있습니다.

이 로봇은 어떻게 보면 최초의 이동로봇이라고 할 수 있겠습니다. 그러니까 실내를 카메라를 통해 인식을 하면서 스스로 이동할 수 있는 첫 번째 이동로봇이었다고 보시면 되겠고요.

여기 유튜브 링크가 걸려 있는데, 66년이면 매우 예전 일인데 그때 만든 로봇이 어떤 식으로 동작했는지 여러분도 보실 수가 있을 것입니다.

그리고 아까 체스 게임을 말씀드렸을 때 잠깐 언급한 DeepBlue라는 프로그램입니다. 이 프로그램은 1997년도에 IBM을 통해서 개발이 되었고요.

아까 말씀드린 CMU라는 대학에서 학생들이 ChipTest라는 체스 프로그램을 개발했고, 그다음 이후에 이게 Deep Thought라는 프로그램으로 진화가 되었습니다.

그런데 이 학생들이 졸업을 하고 IBM에 취직한 이후로 DeepBlue라는 시스템을 개발한 것이고, 그 당시에 인간 챔피언이었던 Kasparov를 3.5 대 2.5로 이기게 되었습니다.

여기서 0.5가 붙은 이유는 체스 같은 경우에는 무승부가 가능하기 때문입니다. 그러니까 세 번 이기고 한 번 비겼다고 이해하시면 되겠습니다.

그런데 그 당시에 인간 챔피언 같은 경우에는 IBM에서 만든 컴퓨터라는 게 뒤에 사람이 실제로 있는데 없는 척하는 걸 수도 있다고 의심을 해서

cheating을 했다고 고소를 하는 웃지 못 할 상황도 있었고요.

또 아주 중요한 milestone 중에 하나는 Honda사에서 개발한 Asimo라는 로봇입니다. 이 Asimo라는 로봇은 초등학교 어린 아이 정도 크기의 휴머노이드 로봇이고요.

처음 나왔을 당시에 아주 사람과 매우 유사한 방식으로 걸을 수 있는 휴머노이드 로봇이라고 각광을 받았습니다.

그 해 여러 번 version up이 돼서 계단을 올라간다든지 혹은 한 발로 뛴다든지 이런 것까지 가능하게 되었고요.

근래에는 보스턴다이내믹스라는 회사에서 여러 가지 휴머노이드가 나왔고요. 아마 여러분도 많이 보셨지만, 공중제비를 도는 휴머노이드 로봇도 현재 개발된 상태입니다.

그리고 또 하나 중요한 건 DARPA Urban Challenge라는 것인데요. 이건 뭐냐 하면, DARPA라는 곳은 미국 국방부의 funding agency입니다.

그러니까 연구 펀드를 여기저기에 배분해주는 조직인데요. 그 조직에서 하나의 challenge를 열었고요.

그 challenge에서 우승한 사람한테는 아주 큰 상금을 주겠다. 그렇게 해서 시작된 challenge입니다.

실제로 DARPA challenge는 2, 3년 전에 있었는데요. 그 당시에는 자율주행 로봇에 대한 것이었습니다.

사막 같은 사람이 없는 곳에서 아주 매우 긴 거리를 자동차가 스스로 운전할 수 있을까? 그걸 평가한 challenge가 있었고요. 그거에 대한 후속으로 Urban Challenge가 정의가 된 것인데요.

Urban Challenge는 사막을 자기가 스스로 가는 게 아니라 실제 도시처럼 만들어놓은 환경에서 자율주행 자동차가 얼마큼 잘 이동할 수 있는지를 경진대회 형태로 참여를 한 것이죠.

그래서 그 당시에 CMU와 미국의 GM이 함께 만든 Tartan Racing이라는 팀이 우승을 해서 1등 상금을 받았고, 그 당시에 200만 달러의 상금을 받게 되었습니다.

그런데 이게 AI 역사에서 매우 중요한 페이지를 장식하는 이유가 근래에 자율주행 자동차라는 화두가 여기저기서 많이 얘기되고 있는데, 그거의 시초가 된 challenge라고 할 수 있기 때문입니다.

그래서 실제로 구글 같은 데서 처음에 자율주행 자동차를 개발한 분들은 다 학생 때 혹은 그 이전에 DARPA Urban Challenge에 참여한 분들인데,

여기서 우승하고, 여기서 좋은 성적을 거둔 분들이 모여서 그런 breakthrough를 일으키게 된 것이죠.

이러한 연구로부터 시작해서 산업계 전체 혹은 인간 사회, 문화적으로 큰 임팩트를 준 아주 대표적인 사건이라고 할 수 있겠습니다.

그리고 또 하나 재미있는 인공지능 역사에서 중요한 순간 중에 하나는 IBM이 만든 Watson이라는 시스템입니다.

이건 미국의 퀴즈쇼 중에 Jeopardy라는 퀴즈쇼가 있습니다. 거기서 인간 우승자들과 함께 한 대회에서 그 인간 우승자들을 이겨서 아주 유명해진 시스템이라고 할 수 있고요.

한편으로 보면 되게 뛰어나지만 어떻게 보면 ‘이건 인공지능이 더 잘할 수밖에 없지 않느냐.’ 이런 생각이 들기도 합니다.

예를 들면 여러분이 퀴즈쇼에 참여를 했는데, 여러분에게 구글 검색을 할 수 있는 기회를 준다면 아마 여러분을 이길 수 있는 인간은 없을 것입니다.

그런 식으로 본다면 기계가 어떻게 보면 단답식의 질의응답 같은 경우라고 한다면, 지식을 외우고 있는지 안 외우고 있는지 평가하는 단답식의 문제라고 한다면,

어떻게 보면 기계가 더 잘할 수밖에 없지 않느냐. 이런 생각이 한편으로는 들기도 합니다.

그리고 마지막으로 소개시켜드리고 싶은 얘기는 많은 분들이 다 알고 계셔서 더 이상 부연할 필요도 없는 알파고입니다.

알파고는 Gooogle DeepMind라는 회사에서 개발이 된 것이고요. DeepMind는 영국 런던에 위치한, 원래 처음에는 벤처기업, 스타트업이었는데 후에 구글에 의해서 인수가 된 기업입니다.

그래서 이 DeepMind에서 첫 번째 바둑 프로그램을 만든 건데요.

첫 번째 바둑 프로그램 중에 인간의 professional, 그러니까 프로 기사를 이긴 그리고 여러 핸디캡 없이 19×19 보드, 그러니까 실제 크기의 바둑판에서 이긴 첫 번째 프로그램으로 알려져 있습니다.

많이들 보셨겠지만 2016년도에 이세돌 9단을 4:1로, 처음에 예상한 것과 달리 되게 손쉽게 이겼고요. 1년 후에 Go Summit이라고 중국에서 펼쳐진 대회였고요.

이세돌 9단이랑 상대한 알파고 같은 경우에 사람의 기보로부터 배운 학습을 통해서 그런 놀라운 성능을 거두었기 때문에

많은 분들에게서 ‘여러 사람의 기보를 다 외워서 하는 거 아니냐.’ 이런 의심을 많이 받았는데,

커제를 상대한 Summit에서 나온 알파고 같은 경우에 인간의 기보를 전혀 사용하지 않고 스스로 게임을 많이 둬가면서 학습한 알고리즘으로 유명해졌습니다.

그래서 이름도 AlphaGo Zero라는 식으로 붙여졌고요. 그러니까 인간의 기보를 전혀 사용하지 않았다는 뜻으로.

여기서 커제 9단을 매우 쉽게 이겼습니다. 3:0으로 이겼고요.

현재로서는 은퇴를 해서 이제 더 이상 알파고와 게임을 할 수는 없지만, 어쨌든 알파고가 특히 한국 사회에 끼친 영향은 매우 크다고 할 수 있겠습니다.

어떻게 보면 지금 많이 얘기하는 인공지능, 4차 산업혁명도 알파고 쇼크에서 많이 오지 않았나 하는 생각이 듭니다.

지금까지 간단하게 인공지능 역사에 대해서 말씀을 드렸고요. 지금 말씀드린 거 외에도 아주 많은 중요한 사건들이 있었습니다.

그리고 오늘 강의를 끝내기 전에 혹시 여러분 중에 본 강의 외에 더 많은 인공지능에 대한 정보를 얻고 싶다 하시는 분들이 계실 수가 있어서 하나의 좋은 책을 소개시켜드리고자 합니다.

이건 「Artificial Intelligence(인공지능)」이라는 제목으로 나온 책이고요. 현재 3판까지 개정이 되었고요.

이 책의 저자는 두 교수입니다. UC Berkeley의 교수이고, 한 분 같은 경우에는 교수를 그만두시고 구글에 몸을 담고 계신 분이 만드신 책이고요.

이 책에 대한 홈페이지는 여기 소개시켜드린 대로 되어 있고요. 여러분이 쉽게 구글링을 통해서 pdf 파일을 찾으실 수 있어서 아마 여러분이 관심 있다면 충분히 읽어보실 만합니다.

대신이 책이 매우 두껍기 때문에 처음 읽어보실 때 너무 낙담하지 마시고 한 페이지, 한 페이지 읽어보시면 인공지능에 많은 흥미를 얻게 되시지 않을까 생각을 합니다.

그러면 오늘의 강의는 여기서 마치도록 하겠습니다.

2.1.

안녕하세요? 서울대학교 컴퓨터공학부와 김건희입니다. 인공지능의 기초, 이번 수업에서는 ‘문제해결 및 탐색전략’이라는 주제로 말씀드리도록 하겠습니다.

인공지능에서 문제를 어떻게 정의하고, 그 문제를 어떻게 풀어내는지를 다루고요. 그리고 많은 경우에 인공지능의 문제라는 건 탐색을 통해서 해결을 하게 되어 있는데요.

그래서 어떤 탐색 아이디어를 가지고 문제를 풀게 되는지에 대해서 자세히 말씀드리도록 하겠습니다.

인공지능의 문제를 소개시켜드리기 위해서 간단한 예제를 먼저 말씀드리도록 하겠습니다. 이 예제는 Romania 예제라고 많이 알려진 문제고요.

간단히 문제에 대해서 말씀을 드리면, 제가 Romania에서 휴가를 보내고 내일 Romania를 떠나게 되었습니다.

그런데 Bucharest라는 도시로 돌아가서 비행기를 타야 되는데요.

여기서 문제는 내가 현재 Arad라는 도시에 있었을 때 어떻게 하면 Bucharest까지 갈 수 있는지 그 문제를 예제로 한번 같이 풀어보도록 하겠습니다.

문제를 정의하기 위해서는 일단 초기 상태와 목표를 정의해야 되는데, 이 문제의 경우에는 Arad라는 도시가 초기 상태가 될 테고요. 그다음에 Bucharest라는 곳이 최종 goal state가 될 것입니다.

그리고 여기서 state라고 말씀을 드리는 건 각 도시들을 state로 정의할 수 있겠고요.

action이라는 건 그 state에서 어떤 state로 가는, 그러니까 운전을 통해서 도시 사이를 이동하는 걸 action이라고 정의할 수 있겠죠.

그래서 결국에 이 문제라는 건 제가 Arad라는 state에서 시작해서 제가 여러 개의 action을 통해 도시를 옮겨 다니면서

결국에는 Bucharest까지 가는 전체 저의 action, sequence를 solution, 해라고 정의를 할 수 있겠습니다.

그래서 결국에는 이런 주어진 문제를 해결하는 여러 solution이 가능합니다. 이 예제에서는 Arad에서 Bucharest까지 가는 수많은 경로가 존재하게 될 테고요.

그 경로 하나하나가 이 문제를 해결하는 해가 될 것입니다. 하지만 많은 경우에 이 수많은 해들 중에서 최고의 해를 찾는 게 목적이 되는 경우가 많고요.

최고의 해라는 건 정의하기에 따라 다르겠지만, 일반적으로 생각할 수 있는 건 최단거리를 이동해서 Bucharest까지 도달하는 경로를 최고의 해라고 정의할 수 있겠습니다.

인공지능에서 문제를 정의하기 위해서는 크게 4가지 구성요소가 필요합니다. 그 구성요소에 대해서 하나하나씩 말씀을 드리도록 하겠습니다.

좀 전에 말씀드렸다시피 상태를 나타내는 state와 action, 이 두 가지가 가장 핵심이 되는 요소라는 건 말씀을 드렸는데요.

그거로부터 훨씬 더 정교화된 4개의 구성요소를 말씀드리도록 하겠습니다.

첫 번째 요소는 초기상태입니다. 그러니까 이 문제를 푸는 데 있어서 첫 상태를 얘기한다고 할 수 있겠고요.

Romania Example에서는 Arad라는 도시에 있는 그 상태가 초기상태라고 쉽게 정의를 할 수 있겠습니다.

두 번째로는 가능한 action들, 가능한 행동들을 정의하는 거라고 할 수 있겠는데요. 여기서 보면 Successor라는 함수를 정의하였습니다.

이 Successor라는 함수는 x라는 input을 받아서 action과 successor라는 pair가 이제 output으로 나오는 함수라고 정의를 할 수 있겠고요.

좀 전에 말씀드린 Romania 예제에서 보실 수 있다시피 현재 제가 Arad라는 state에 있다고 했을 때 Arad라는 state에서 할 수 있는 action들이 여러 도시로 가는 게 하나하나가 action이 되겠죠.

그래서 제가 만약에 Zerind라는 도시로 가는 action을 취한다고 하면, 그거의 결과로 Zerind라는 successor state로 도달하게 되는 것이죠.

그래서 action이라는 건 결국에 어떤 상태에서 다른 상태로 이동하기 위한 행동이라고 정의를 할 수가 있겠습니다.

세 번째 구성요소는 Goal test입니다. 그러니까 쉽게 얘기해서 현재 상태가 우리가 문제를 풀고자 하는 목적, goal에 도달했는지 아닌지를 판단해주는 일종의 테스트라고 보시면 되겠습니다.

그래서 좀 전에 말씀드린 Romania 예제에서는 Bucharest까지 가는, 그러니까 Bucharest라는 state 자체가 goal이 되겠죠.

그래서 만약에 제가 Bucharest라는 상태에 현재 도달했다. 그러면 Goal test를 통과했다고 할 수가 있겠죠.

그런데 문제에 따라서는 이런 식으로 하나의 state를 goal로 정의할 수도 있지만, 많은 경우에 state를 하나로 정의하기보다는 조건으로 정의하는 게 더 좋은 경우도 있습니다.

예를 들어서 여러분이 체스 게임을 문제로 정의를 해서 푼다고 했을 때 체스에서는 결국 상대방이 왕을 잡는 것이 목적이 될 테고요. 그게 goal이 될 테고요.

그래서 상대방의 왕을 잡는 경우는 아주 많기 때문에 그 state를 일일이 나열하는 건 매우 힘들고요.

단지 어떤 경우에 왕을 잡게 되는지 그 조건을 기술한 함수로서 Goal test를 수행할 수가 있겠습니다. 이런 식으로 명시적으로 Goal test를 정의할 수도 있고요. 비명시적으로도 가능합니다.

네 번째 구성요소는 경로 cost입니다. Path cost라고 정의를 하는데요.

이건 뭐냐 하면, 제가 어떤 state에서 어떤 action을 취해서 또 다른 state로 간다. 이걸 하나의 경로라고 할 수 있는데, 이 경로를 수행하는 데 있어서의 비용을 Path cost로 정의할 수 있겠습니다.

그래서 여기서는 Step cost라는 걸 하나 정의했는데요. 이 Step cost라는 건 3개의 input을 받습니다.

그러니까 Step cost는 x라는 state에서 a라는 action을 취해서 y라는 state로 갔을 때의 cost를 c(x, a, y)로 정의할 수 있겠죠.

그리고 이런 Step cost는 많은 경우에 0보다 크거나 같은 식으로 정의를 합니다.

그러니까 제가 어떤 action을 하는 경우에 비용이 발생하는 경우가 많고요. 최상의 경우에는 비용이 하나도 없는 경우도 있겠죠.

그렇지만 비용이 마이너스가 되는 경우는, 물론 그것도 여러분이 정의하기에 따라서 가능할 수는 있지만 일반적으로 ‘Path cost는 0보다 크거나 같다.’ 그런 식으로 정의를 하게 됩니다.

그래서 좀 전에 말씀드린 Romania 예제에서는 어떤 도시에서 어떤 action을 취해서 다른 도시로 갔다고 했을 때

그 도시들 사이의 거리가 Path cost로 주어지는 경우가 Step cost로 주어지고, 그래서 총 action 결과로서의 Path에 대해서 비용을 정의할 수 있겠죠.

solution이라는 건 Initial state에서 수많은 action을 통해서 goal state로 가게 될 텐데, 그 action들의 연속을 인공지능 문제에서의 해라고 정의할 수 있겠습니다.

인공지능 문제를 정의하는 데 있어서 중요한 것 중에 하나가 state를 어떻게 정의하느냐? 이것입니다.

실제로 우리가 살고 있는 세상은 생각보다 매우 복잡합니다. 그러니까 제가 만약에 Arad라는 도시에서 Bucharest까지 가는 방식은 매우 다양하고요.

그게 단순히 action 몇 개로만 할 수 있는 일도 아닙니다. 그렇지만 많은 경우에 실제 문제를 모델링을 통해서 일종의 abstraction, 추상화 과정을 통해서 먼저를 정의하게 됩니다.

추상화라는 건 여러분이 실제 문제를 모델링 하는 데 있어서 실제 문제보다 이해하기 쉽게 문제를 변환하는 과정이라고 할 수가 있겠습니다.

한 가지 예로 좀 전에 Romania 예제에서 각 도시를 state라고 정의했는데, 실제로 제가 Arad라는 도시에 있다는 것 자체는 수많은 경우가 있고, 수많은 복잡한 상황이 있을 텐데,

그냥 그 모든 상황을 하나의 state로 abstraction, 그러니까 추상화를 했다고 보실 수가 있겠죠.

그리고 action도 마찬가지로 추상화를 통해서 우리가 좀 전에 문제를 정의했다고 할 수 있는데요.

우리가 특정 도시에서 다른 도시로 간다는 action이라는 건 매우 복잡한 과정이 될 텐데, 그 복잡한 과정을 그냥 하나의 action으로 추상화했다고 보실 수가 있겠습니다.

마찬가지로 이런 추상화를 통해서 문제를 정의하게 되면 그 solution 자체도 어떻게 보면 매우 추상화된 해라고 할 수 있겠죠.

어떻게 보면 도시의 연속이 일종의 solution이 될 텐데, 그게 그렇게 간단한 문제는 아니고 현실에서는 그게 매우 복잡한 상황이지만

우리가 추상화를 통해서 해와 문제를 매우 간단하게 그리고 이해하기 쉽게 바꿨다고 보시면 되겠습니다.

그래서 결국에는 abstraction이라는 건 우리가 원래 문제, 그러니까 Arad에서 Bucharest까지 가는 원래 문제를 추상화를 통해서 좀 더 이해하기 쉽게 만드는 과정이라고 보시면 되겠습니다.

그래서 다른 문제에 대해서도 이와 마찬가지로 4개의 중요한 구성요소로 표현을 할 수 있습니다. 이건 많이 아시다시피 8-puzzle이라고 알려진 문제인데요.

여기 보시면 가운데 빈칸이 존재하죠. 빈칸이 존재한다는 건 우리가 이 빈칸의 주변에 있는 4개의 말 중에 하나를 이 가운데 영역으로 움직일 수 있다는 걸 의미합니다.

우리가 어떤 번호를 선택해서 그걸 빈 공간으로 움직이는 것 자체가 action이라고 정의할 수 있겠죠.

그리고 state라는 건 여기서는 어떻게 정의를 했느냐 하면, 1~8의 숫자가 배열된 상태를 하나의 state로 정의를 하였습니다.

그래서 Initial state는 왼쪽에 보이다시피 여러 숫자가 무작위로 배열된 상태고요.

우리가 도달하고자 하는 Goal state는 1번부터 8번까지 아주 정렬이 잘된 상태로 바꾸는 것이 Goal state라고 할 수 있겠죠.

그리고 여기서 중요한 Path cost는 뭐냐 하면, 많은 경우에 우리가 최적의 solution을 얻는다는 건 최단시간 안에 이 퍼즐을 풀어내는 거기 때문에

그렇다고 하면 말 하나를 움직이는 게 똑같은 시간이 걸린다고 가정하면 하나의 움직임을 1이라는 Path cost로 정의할 수 있겠죠.

만약에 이런 Path cost로 정의를 하고 이 Path cost대로 문제를 푼다면, 우리가 최소한으로 빈칸을 움직여서 Goal state로 도달하는 해를 구한다고 보실 수가 있겠죠.

이거는 Bucharest 예제와 또 다른 인공지능에서의 문제를 이런 식으로 정의할 수 있다는 것이고요. 이외에도 수많은 문제를 지금 말씀드린 4가지 요소로 여러분이 한번 정의를 해볼 수가 있겠죠.

그러면 많은 경우에 이와 같은 해를 구하는 과정을 Tree Search라는 알고리즘을 통해서 해결을 하게 됩니다. Tree라는 건 아시다시피 일종의 나무처럼 생긴 데이터 구조를 말하는데요.

어떤 특정 노드가 있고, 그 노드에 여러 가지 가능성이 있는 노드들을 연결하고, 그 각각의 노드에 대해서 또 가능성이 있는 state를 나열합니다.

그러다 보면 그게 Tree와 같은 모양으로 갖추어졌다고 해서 Tree Search Algorithms이라고 말을 합니다. 그래서 결국에 Initial state가 Tree의 루트, 그러니까 꼭대기가 되고요.

거기서 갈 수 있는 다양한 state가 그 밑의 자식 노드가 될 테고요. 그 자식 노드로부터 또 갈 수 있는 state가 여러 개 있을 텐데요.

거기서부터 또 노드들을 펼쳐나가고 펼쳐나가다 보면, 결국 우리가 하나의 Tree Search를 통해서 Bucharest라는 state까지 가는 그런 solution을 찾을 수 있겠죠.

그런 식으로 Tree라는 데이터 구조를 이용해서 문제를 해결하는 과정을 Tree Search Algorithms이라고 정의를 할 수 있겠습니다.

그래서 현재 state부터 시작해서 그 state로부터 도달할 수 있는 state를 찾아내고, 그 state로 가서 또 그 state에서 갈 수 있는 여러 state를 펼치는 식으로 하나하나씩 state를 펼쳐나가면서,

그러니까 Tree를 점점 펼쳐나가면서 solution을 찾는 과정을 진행하게 될 것입니다.

이건 좀 전에 말씀드린 Romania 예제로부터 Tree Search를 어떻게 적용할 수 있는지를 예제로 보여드린 것입니다.

좀 전에 말씀드렸다시피 우리가 현재 Arad라는 도시에 있습니다. 그리고 city map, 그러니까 지도를 통해서 보시면 아시겠지만 Arad에서 도달할 수 있는 도시가 총 3군데가 있습니다.

그렇다면 Arad를 Tree의 루트라고 했을 때 거기서부터 펼쳐질 수 있는 노드는 이와 같이 3가지 노드가 가능합니다.

그러면 이 각각의 3가지 노드에 대해서 마찬가지로 인접한 노드들을 하나하나씩 찾아내면 거기서부터 또 도달할 수 있는 노드를 Tree 형태로 이와 같이 찾아갈 수 있을 것입니다.

그래서 이 과정을 계속 반복하다 보면 결국 Bucharest라는 노드가 나올 텐데, 그렇다면 그 루트부터 그 state까지의 Path를 만든다고 하면 그 Path 하나가 solution이 되겠죠.

그런 식으로 해서 Tree Search Algorithms을 통해서 우리가 정의한 인공지능 문제를 해결하게 됩니다.

그래서 지금까지 말씀드린 내용을 정리하자면, state space, 그러니까 모든 가능한 state 공간을 다 펼치고 펼치다 보면 그게 하나의 나무, Tree라는 구조를 가지게 된다는 말씀을 드린 거고요.

그다음에 저희의 Initial state, 그러니까 초기조건, 초기상태라는 건 Tree라는 데이터 구조에서 루트, 그러니까 제일 꼭대기죠? 제일 꼭대기에 위치하게 되고요.

각 노드는 여기서 보시다시피 도시 이름들이 적혀 있는데, 각 노드는 state를 나타냅니다. 이 Tree상에서의 action이라는 건 각각의 edge들을 action이라고 정의할 수 있겠죠.

그러니까 Arad라는 state에서는 3개의 action이 가능한 겁니다.

Sibiu라는 도시로 갈 수 있는 action, Timisoara라는 도시로 갈 수 있는 action, Zerind라는 도시로 갈 수 있는 action. 이렇게 3개가 존재하는 것이고요.

각 action은 결국 일종의 Tree에서의 branch, 가지를 생성하게 됩니다. 그래서 결국에 저희가 Arad부터 시작해서 3개를 펼치고요.

그럼 이제 3개 중에서 우리가 그다음으로 펼쳐야 될 노드가 뭔지를 결정해야겠죠.

만약에 우리가 Sibiu라는 도시부터 탐색을 해나가겠다. 그렇게 하면 Sibiu에서 도달할 수 있는 state를 펴나가게 되는 것이죠.

그런 식으로 탐색이라는 건 결국 state를 펼치고, 그 중에 어느 state부터 펼칠지를 결정해서 그 state로 가서 또 펼치는 과정을 반복함으로써 Goal state를 찾는 과정이라고 보실 수 있겠습니다.

그러면 여기서 보시면 좀 전에 말씀드렸다시피 탐색을 하기 위해서는 전략이 필요합니다.

그러니까 첫 문장에 나와 있다시피 전략이라는 건 무엇이냐 하면, 우리가 각각의 노드를 펼쳐나가야 되는데 펼쳐나갈 때 어떤 순서로 펼쳐나갈 것이냐? 그걸 정의한다고 할 수 있겠습니다.

앞 슬라이드에서 보시면, 저희가 Arad에서 3개의 노드를 펼쳤고요. 그다음에 이 3개 중에서 어떤 노드부터 펼칠 것이냐?

만약 Sibiu라는 도시를 선택하면 다시 4개의 노드가 나오는데, 그러면 우리가 아직 펴보지 않은 노드가 6개 존재하게 되겠죠.

그러면 이 6개 중에 또 뭘 펼칠 거냐? 그런 전략을 탐색 전략이라고 얘기합니다.

그러면 여러분이 여러 아이디어를 사용해서 전략을 정의할 수 있습니다. ‘어떤 순서대로 내가 펼칠 것이다.’ 그런 전략을 정의할 수 있는데요.

그 수많은 다른 전략 중에서 뭐가 더 좋은지를 판단하기 위해서는 어떤 구성요소가 필요하겠죠. 그러니까 어떤 기준으로 이 전략을 판단하겠다.

일반적으로 크게 4가지 기준으로 전략을 판단합니다. 첫 번째는 Completeness, 그러니까 완전성입니다. 내가 만약에 이 전략을 따른다면 반드시 solution을 찾을 수 있느냐?

만약에 Completeness에 대한 답이 Yes라고 하면 이 전략을 사용했을 때 만약에 해가 존재한다면 반드시 내가 찾을 수 있다. 이렇게 얘기할 수 있겠죠.

그래서 전략들 중에 Completeness를 만족하는 전략이 더 좋다고 할 수 있습니다. 왜냐하면, 내가 이 전략을 선택하는 순간 내가 해를 찾을 수 있다는 게 보증이 되기 때문입니다.

두 번째 요소는 Time complexity입니다. 이건 시간 복잡도라는 건데요.

우리가 노드 하나하나를 goal인지 아닌지 판단하는 데 똑같은 시간이 걸린다면, 결국 Time complexity라는 건 우리가 얼마큼의 노드를 펼쳐야지만 goal을 찾을 수 있느냐를 나타낸다고 할 수 있겠죠.

그게 기준이라고 한다면 당연히 Time complexity가 낮은 전략을 쓰는 게 좋을 것입니다.

Space complexity는 뭐냐 하면, 여러분이 노드를 하나씩 펼쳐나갈 텐데 여러분이 펼쳐만 놓고 그 노드를 자세히 탐색을 안 했다면 그건 메모리상에 저장을 해놓아야 됩니다.

왜냐하면 나중에 그 노드를 펼쳐봐야 되기 때문에. 그런 식으로 내가 어떤 전략을 썼을 때 메모리 공간이 얼마나 필요하냐? 이걸 Space complexity라고 정의를 하고요.

그리고 전략들 중에 당연히 Space complexity가 낮은 전략이 더 좋다고 할 수 있겠죠.

Optimality는 뭐냐? Completeness보다 좀 더 나아간 기준이라고 볼 수 있는데요. 내가 항상 최적의 solution을 찾을 수 있느냐?

그러니까 내가 Arad까지 갈 수 있는 수많은 Path들이 있는데, 내가 이 전략을 쓴다면 최적으로 갈 수 있는 solution을 내가 찾을 수 있느냐? 그거에 대한 기준이라고 할 수 있습니다.

그래서 결국에 Optimality도 Yes가 되면 제일 좋다고 할 수 있겠죠. 그런데 여러분이 생각을 해보시면 아실 수 있겠지만, 모든 전략 중에서 서로 trade off가 있습니다.

어떤 전략은 어떤 측면에서는 더 좋고요. 또 다른 전략은 그 측면에서는 안 좋지만 또 다른 측면에서는 좋고요.

그래서 많은 경우에 여러분이 문제에 따라서 적절한 전략을 선택하는 것이 매우 중요합니다.

그래서 탐색 전략에 대해서 좀 자세히 다루려고 합니다. 탐색 전략 중에서 가장 먼저 생각해볼 수 있는 건 Uninformed search라는 전략입니다.

이건 좀 전에 말씀드린 문제 정의의 4가지 구성요소에서 주어진 정보만을 활용해서 탐색을 하는 전략들을 Uninformed search라고 합니다.

다음 시간에는 Heuristic을 이용한 search에 대해서도 얘기할 예정인데, 이번 시간에는 Uninformed search에 집중해서 말씀드리고자 하고요.

그리고 Uninformed search의 대표적인 5개는 지금 보여드린 이와 같습니다.

그런데 여기서 보시다시피 5개의 서로 다른 대표적인 Uninformed search 전략들이 있긴 하지만, 얘네들도 결국에 크게 보면 두 가지 탐색 전략으로부터 유래됐다고 할 수 있습니다.

그 두 가지 탐색 전략이란 Breadth-first search와 Depth-first search를 얘기합니다.

Breadth-first search라는 건 많은 경우에 BFS라고 약자로 쓰기도 하고요. 한국말로는 ‘너비 우선 탐색’이라고 얘기합니다.

Depth-first search는 DFS라고 많이 요약을 해서 얘기를 하고요. 한국어로는 ‘깊이 우선 탐색’이라고 얘기합니다.

그래서 이 뒤에 나온 Depth-limited search라든지 Iterative deepening search는 다 Depth-first search를 기반으로 해서 그걸 좀 더 개선한 알고리즘이라고 할 수 있겠고요.

Uniform-cost search도 Breadth-first search를 좀 더 변형한 탐색 전략이라고 이해하시면 되겠습니다.

사실 이 5개에 대해서 다 다루는 게 아주 재미있긴 하지만, 본 강의에서는 가장 기본이 되는 Breadth-first search와 Depth-first search에 대해서 깊이 다루도록 하겠습니다.

그럼 우선적으로 Breadth-first search, 그러니까 너비우선탐색이라는 전략에 대해서 소개를 시켜드리도록 하겠습니다.

기본 아이디어는 이와 같습니다. 만약에 주어진 노드가 있고 아직 펼쳐지지 않은 노드들이 있다고 하면, 가장 얕은 아직 펼쳐지지 않은 노드부터 펼치자. 그런 전략이라고 할 수 있겠습니다.

그리고 이런 경우에 Breadth-first search는 FIFO라는 데이터 구조로 구현이 가능한데요. FIFO는 무슨 뜻이냐 하면, First In First Out입니다.

그래서 ‘처음 들어간 게 먼저 나온다. 먼저 들어간 게 먼저 나온다.’ 이런 철학을 가진 데이터 구조라고 할 수 있습니다.

여기서 Fringe라는 단어는 뭐냐 하면, 아까 말씀드린 대로 일단 노드가 나오긴 했지만 아직 펼쳐지지 않은 노드들을 Fringe라고 얘기합니다.

그래서 과연 이 Breadth-first search가 어떻게 동작이 되고, 이게 어떻게 FIFO라는 queue를 통해서 구현이 되는지 하나하나씩 말씀을 드리도록 하겠습니다.

그러면 이제 루트부터 하나씩 가보도록 하죠. 여기서 루트 노드라는 건 결국 Initial state가 되겠죠. Initial state가 여기서는 A라는 게 되겠죠.

그러면 Initial state를 일단 FIFO, 그러니까 First In First Out이라는 queue에 넣었습니다. 그러면 여기 보시다시피 A라는 노드가 queue 안에 존재하게 되겠죠.

그러면 그다음 스텝으로 넘어가도록 하겠습니다. 여기서 지금 A라는 노드가 queue 사이에 있기 때문에 A라는 걸 하나 빼는 것이죠.

A를 빼고 A에서 갈 수 있는 게 여기 보시다시피 B와 C라는 노드가 있습니다.

그러면 A라는 노드를 queue에서 뺀 다음에 ‘얘가 goal이냐 아니냐?’를 판단하고 ‘goal이 아니다.’, ‘그렇다면 여기서 갈 수 있는 state가 뭐냐?’ 그걸 찾은 다음에 그 각각을 queue에 넣습니다.

그러니까 지금 A라는 Initial state로 queue에 넣었다가 그걸 빼서 goal인지 판단하고 펼치는 순간 이제 A는 탐색이 다 됐기 때문에 queue에서 버려지게 됩니다.

그래서 A는 버려지고 B와 C라는 노드가 queue상에 존재하게 되겠죠.

그러면 First In First Out이기 때문에 여기서 B와 C 중에 앞에 있는 것이 무엇이죠? 앞에 있는 게 B입니다. 그래서 B를 그다음에 빼내봅니다.

그래서 B를 빼내서 ‘그럼 B가 지금 Goal state냐?’, ‘아니다.’, ‘그러면 B에서 갈 수 있는 노드가 어떤 것들에 있느냐?’ 이 Example에서는 D와 E가 있겠죠.

그러면 B를 판단해서 봤더니 D와 E라는 노드가 새로 생겨나게 되고요. D라는 노드를 queue에 넣고 E라는 노드를 queue에 넣고 이제 B라는 노드는 버려지게 되겠죠.

그럼 결과적으로 queue에는 C, D, E가 남아있게 되죠. 그런데 여기 queue에 C, D, E라는 세 가지 요소가 있는데, 이 중에서 가장 먼저 들어온 게 무엇이죠?

가장 먼저 들어온 게 C였죠. B가 들어온 다음에 C가 들어왔고, 그다음에 B를 판단한 다음에 D와 E가 queue 사이에 들어왔습니다.

그렇다면 가장 먼저 들어온 애가 C이기 때문에 C를 마찬가지로 판단을 하게 되겠죠.

그래서 C를 queue에서 빼낸 다음에 ‘이게 골이냐 아니냐?’를 판단하고, 만약에 ‘아니다.’라고 하면 ‘C에서 도달할 수 있는 state가 뭐냐?’ 봤더니 F랑 G다. 그러면 이 F랑 G가 queue에 들어가게 되겠죠.

그러면 C는 버려지고 D, F는 뒤에 붙어 있었고, F랑 G가 들어가니까 결국에는 C, E, F, G라는 요소들이 queue에 존재하게 되겠죠.

이 상황에서 다음에 또 search를 해나간다고 하면 어떤 노드부터 evaluation이 될까요? 그건 당연히 D부터 나오겠죠.

그래서 좀 전에 말씀드린 대로 D라는 노드를 또 queue에서 제일 앞에 있는 노드이기 때문에 빼낸 다음에

그걸 또 마찬가지로 ‘goal이냐 아니냐?’ 만약에 goal이면 ‘solution을 찾았다.’ 이렇게 결론을 내릴 수가 있고요. goal이 아니다. 그러면 D에서 펼쳐질 수 있는 게 뭐냐?

그 노드들을 찾은 다음에 그걸 다시 queue의 뒷부분에 넣어주면, 이런 식으로 계속 반복을 해주면 결국에 Breadth-first search를 수행할 수 있게 됩니다.

지금 말씀드린 내용을 보시면, FIFO라는 데이터 구조, 그러니까 First In First Out을 통해서 저희가 탐색 전략을 구현했는데요.

이 전략대로 하다 보니까 A를 먼저 탐색하고, 그다음 B 탐색하고, 그다음에 B로부터 D, E가 펼쳐져 있긴 하지만 D, E, C 중에서 그다음에 C를 먼저 펼치게 됩니다.

그러니까 이 FIFO queue대로 전략을 구현하다 보면, 자연스럽게 같은 depth, 같은 깊이에 있는 노드를 일단 다 펼쳐보고

goal인지 아닌지 판단을 한 다음에 다음 depth로, 다음 깊이로 넘어가는 걸 여러분이 아실 수가 있겠죠.

그렇기 때문에 Breadth-first search, 그러니까 너비우선탐색이라는 이름이 붙게 된 것입니다.

루트를 검색하고요. 그다음에 depth1. 그러니까 루트에서 갈 수 있는 노드들이 depth1에 위치하겠죠.

그 노드들을 다 goal인지 아닌지 판단하고 펼치고 난 다음에야 depth2, 두 번째 깊이에 있는 노드들을 하나하나씩 펼쳐나가게 되겠죠.

그래서 이걸 Breadth-first search라고 하고, Breadth-first search는 고민할 거 없이 여러분이 FIFO라는 data structure, 데이터 구조를 활용하면 아주 쉽게 구현할 수 있다는 것입니다.

그러면 지금까지 너비우선탐색에 대해서 말씀을 드렸고요. 이제는 깊이우선탐색에 대해서 말씀을 드리도록 하겠습니다.

깊이우선탐색은 이름에서도 아실 수 있다시피 가장 깊은 아직 펼쳐지지 않은 노드들부터 하나하나씩 체크하고 펼쳐나가겠다. 그런 아이디어입니다.

Breadth-first search 같은 경우에 가장 낮은 shallow unexpanded node를 찾았다고 하면, 여기에서는 deepest unexpanded node를 찾는다고 보시면 되겠고요.

구현은 어떻게 하느냐? LIFO라는 data structure를 이용하면 됩니다. LIFO는 Last In First Out입니다.

그러니까 가장 늦게 들어간 애가 가장 먼저 나오는 식으로 데이터 구조가 구성되어 있다고 보시면 되겠습니다.

여기서도 마찬가지로 아주 간단한 예제와 함께 깊이우선탐색에 대해서 말씀을 드리고자 합니다.

시작점은 Initial state죠? 여기서는 Initial state가 A이기 때문에 A가 LIFO, 그러니까 Last In First Out queue에 이제 들어가게 됩니다.

그래서 이렇게 넣어놓은 상태에서 queue 안에 있는, 가장 앞에 있는 요소를 끄집어냅니다. 그럼 A를 끄집어내서 ‘A가 goal이냐 아니냐?’ 판단을 하고요.

goal이 아니라고 하면 얘가 갈 수 있는, 여기서부터 펼쳐질 수 있는 노드가 뭐냐? 그랬더니 B와 C다. 그렇게 이제 판단이 되고요.

그런데 B, C가 함께 같이 펴진 거잖아요. A로부터 B, C가 같이 펴졌는데, 이해를 좀 쉽게 하기 위해서 이 둘 중에 어떤 걸 먼저 넣을까 할 때 C부터 넣고 B를 넣었다고 한번 가정을 해보죠.

그러면 이와 같이 되겠죠. C를 넣은 다음에 B를 넣었습니다. 아까 같은 경우에는 뒤를 통해서 B, C가 들어갔다고 하면, 여기서는 앞을 통해서 B, C가 들어갔다고 할 수 있겠죠.

왜냐하면 Last In First Out이기 때문에 가장 늦게 들어간 애가 앞에 위치해야 됩니다.

그런데 여기서는 A에서 펼쳐질 수 있는 게 B, C였는데, 저희가 편의상 둘 중에 C부터 넣고 그다음에 B를 넣었다고 가정을 해보죠.

그런 식으로 했다고 하면 지금 B와 C가 queue상에 존재합니다. 그러면 C부터 넣은 다음에 B를 넣기 때문에 Last In이 B입니다. 그러면 Last In이 First Out 해야 되기 때문에 B가 이제 나옵니다.

그럼 B가 나와서 마찬가지로 evaluation 과정을 거칩니다. 그래서 B가 goal이냐 아니냐? 아니다. 그럼 B에서 도달할 수 있는 state가 뭐냐? D와 E다.

그런데 좀 전에도 말씀드렸다시피 D, E 중에 E부터 한번 넣죠. E부터 넣고 D를 넣었다고 가정을 해보시죠.

그럼 결국에는 B랑 C가 있었는데 B가 빠져나왔고요. 거기서부터 D와 E가 나왔는데 E를 넣고 D를 넣었습니다.

아까도 말씀드렸다시피 여기서는 들어가는 게 앞을 통해서 들어갑니다. 그래서 C가 있었는데 E를 넣고 D를 넣었습니다.

그럼 여기서 가장 늦게 들어간 게 D고요. 그렇기 때문에 D가 가장 먼저 나와야 됩니다. 그리고 queue상에서 가장 앞에 위치하게 되겠고요.

그래서 D도 마찬가지 과정을 거칩니다. D를 다시 빼낸 다음에 D가 goal이냐 아니냐? 아니다. 여기서 펼쳐질 수 있는 게 뭐냐? H랑 I다.

그러면 좀 전에 말씀드렸다시피 I를 넣고 H를 넣고요. 그러면 H부터 나와서 evaluation을 수행하게 됩니다.

결국에 H, I, E, C가 queue상에 존재하게 되죠. 이 상황에서 또 이 과정을 반복합니다. 그러면 H가 나오고요. H가 나왔는데 이게 goal이냐 아니냐? 그래서 아니라고 판단이 됐고요.

그런데 여기서 보시다시피 H에서는 더 이상 펼쳐질 수 있는 노드가 없습니다. 그러면 H가 나와서 goal이냐 아니냐? 아니다. 그러면 이제 더 이상 넣는 거 없이 I, E, C가 되는 것이죠.

이제 LIFO에는 I, E, C가 되고요. 그럼 여기서 가장 앞에 있는 게 I이기 때문에 I가 또 나옵니다. 그러면 I가 나와서 goal이냐 아니냐? 아니다. 또 펼쳐질 수 있는 게 없다.

그럼 이제 결국에 E와 C가 남고요. E와 C가 남은 상태에서 또 마찬가지로 E를 빼내고 E가 goal이냐 아니냐? 아니다.

그런데 여기서는 이제 또 펼쳐질 수 있는 게 있죠. J와 K가 펼쳐질 수 있는데, 이 상태에서 C만 남아있는 queue에서 K를 앞으로 넣고요. 그다음에 J를 앞으로 넣는다고 하면 J, K, C가 되겠고요.

또 이 과정을 계속 반복하게 됩니다. 그래서 J를 판단하고요. J가 goal이 아니었고, K도 goal이 아니었고요. 그러면 이제 또 C가 되고요.

이런 식으로 하면 B로부터 펼쳐질 수 있는 건 이미 다 검색이 됐는데, C쪽은 전혀 검색이 안 된 것입니다.

그러니까 Depth-first search는 어떤 특정한 Path를 정해서 끝까지 가보는 것입니다. 끝까지 가서 goal이 나올 때까지 혹은 더 이상 펼쳐지는 게 없을 때까지 가보고요.

만약에 goal이 아니었다고 판단하면 다시 올라와서 다시 깊이로 끝까지 가고요. 이 과정을 반복을 하게 됩니다. 그래서 뒤에 있는 부분은 여러분이 쉽게 응용을 할 수 있겠죠.

그래서 C로부터 펼쳐질 수 있는 노드가 F와 G가 되고요. 그게 LIFO queue에 저장이 되고요. F부터 시작해서 앞서 말씀드린 과정을 반복하게 됩니다.

F가 goal인지 판단을 하고, 그다음에 펼치고요. 그다음에 L, M이 새로 나왔고요. L이 goal인지 아닌지 판단을 하고, 아니라고 하면 M에 대해서 판단을 하고요.

그래서 이 과정을 계속해서 반복을 하다 보면, 결국에는 모든 search space를 다 찾을 수 있게 되겠죠.

좀 전에 말씀드린 Breadth-first search랑 Depth-first search를 비교해보시면, 너비우선탐색은 어떻게 보면 Initial state에서 도달할 수 있는 가장 가까운 영역부터 샅샅이 다 찾는 거고요.

그리고 깊이우선탐색 같은 경우에는 어떻게 보면 하나의 Path를 정해서 그 Path로 끝까지 한번 가보는 것이죠. goal이 됐든 goal이 되지 않았든.

그거에 대해서 판단을 하고, 그다음에 거기서 goal을 찾으면 다행인데 만약 goal을 못 찾았다. 그럼 다시 위로 올라와서 찾는 이런 과정을 반복하게 되는 것입니다.

그러면 마지막으로 BFS(너비우선탐색)과 DFS(깊이우선탐색)을 요약해서 말씀드리도록 하겠습니다.

아까도 말씀드렸다시피 어떤 탐색 전략을 판단하는 기준은 크게 많이 사용되는 게 4가지라고 말씀을 드렸고요.

이 4가지 특성을 하나로 요약한 표를 여러분이 보시는 것입니다. 그래서 완전성이 있느냐 했을 때 BFS는 Yes입니다.

그러니까 BFS를 쓰면 시간이 얼마 걸리든지 간에 이 전략을 계속 쓰다 보면 결국에 해를 찾을 수 있다는 거고요.

DFS는 애석하게도 No입니다. 그러니까 좀 전에 보셨다시피 만약에 우리가 왼쪽 Path로 계속 끝까지 가게 되는데,

재수가 없는 경우에는 어떤 state를 갔다가 다시 돌아오는 경우도 있습니다. 계속 사이클을 그리면서 돌아오는 경우가 있는데요.

실제로는 오른쪽 Path를 통해서 가야지 goal이 있는데 왼쪽 Path를 취했고, 그런데 계속 가다 보면 그리고 거기서 어떤 루프에 빠진다고 하면 결국에 Completeness를 만족하지 않게 되겠죠.

그다음에 여기 보시면 Time complexity와 Space complexity가 있는데요. 되게 복잡한 notation으로 되어 있습니다.

알파벳 대문자로 O라고 되어 있고요. 그 안에 어떤 기호로 수식이 표현되어 있는데요.

이 O라는 건 복잡도를 나타내는 notation 중에 하나입니다. 그래서 Big-O notation이라고 일반적으로 얘기하는 방식이고요.

이건 여러분이 만약에 알고리즘에 대해서 공부를 하셨다면 배우셨을 거라고 생각이 되고요.

혹시 알고리즘을 배우시지 않았더라면 지금은 그냥 한번 넘어가시고 나중에 혹시 알고리즘을 알게 되면 다시 한 번 보시는 게 좋지 않을까 생각이 됩니다.

그래서 여기 여러 가지 기호가 있는데, b는 한 노드에서 펼쳐질 수 있는 최대의 숫자입니다. 그러니까 최대로 펼쳐질 수 있는 branch 수고요.

그다음에 d라는 건 goal이 있는 depth입니다. m이라는 건 search를 하게 될 depth입니다.

그래서 이렇게 보시면 좀 복잡하게 느껴지실 수도 있는데, 간단히만 말씀을 드리면 Breadth-first search는 Completeness에서는 장점이 있긴 하지만 알고리즘 자체가 복잡합니다.

그러니까 시간도 더 많이 걸리고, 메모리 공간도 훨씬 많이 차지하게 되죠.

그런데 Depth-first search는 해를 찾을 수 있다는 걸 guarantee, 보장은 못해 주지만, 한결 한결 특히 space complexity 관점에서는 훨씬 효율적인 알고리즘이라고 보시면 되겠습니다.

그리고 마지막 Optimality라는 요소가 있는데요. Optimality는 BFS 같은 경우에는 Yes지만 DFS 같은 경우에는 No입니다.

사실 이 4가지 기준에 대해서 이 결과가 어떻게 나왔는지 한번 유도해보는 것도 재미있는 문제이긴 한데,

시간관계상 여기서 다루지는 않습니다만 여러분이 혹시 관심이 있으시다면 웹에서 충분히 어떻게 유도하는지 과정을 쉽게 공부하실 수 있기 때문에 여러분에게 숙제로 남겨두도록 하겠습니다.

그런데 마지막으로 좀 전에도 시작하기 전에 말씀드렸다시피 오늘 강좌에서는 Uninformed search 전략에 대해서만 말씀을 드렸습니다.

그런데 이 Uninformed search 말고 여러 가지 다른 정보를 활용하는 Heuristic search라는 게 있는데요.

Heuristic search을 활용하면 지금 말씀드린 Uninformed search보다는 훨씬 더 쉽게 solution을 찾을 수가 있습니다. 그래서 다음 시간에는 그거에 대해서 주로 다룰 예정이고요.

그리고 그렇다 하더라도 지금 말씀드린 Breadth-first search, Depth-first search는 Heuristic search에서도 계속 사용될 거기 때문에

기본 개념을 이해하는 측면에서는 반드시 알아두어야 할 탐색 전략이라고 할 수 있겠습니다.

그러면 오늘의 강의는 여기서 마치도록 하겠고요. 다음 강좌를 통해서 다시 만나 뵙도록 하겠습니다.

3.1.

안녕하세요? 서울대학교 컴퓨터공학과의 김건희입니다. 오늘은 인공지능의 기초 휴리스틱 탐색과 지역 탐색에 대한 주제로 말씀을 드리도록 하겠습니다.

지난 시간에 저희가 Depth-first search랑 Breath-first search라는 탐색 전략에 대해서 배웠습니다.

그 둘을 소개하면서 제가 그 두 가지는 Uninformed Search Strategies라고 말씀을 드렸는데요.

Uninformed라는 얘기는 문제를 정의하는 데 필요한 4가지 요소, 그 4가지 요소 외에 다른 정보는 전혀 주어지지 않은 상태에서 탐색을 수행하는 과정입니다.

이번 주차에서 다룰 내용은 그와는 반대로 문제정의 외에 다른 추가적인 정보가 있을 때 어떻게 그것을 활용하여 탐색을 더 용이하게 하느냐, 그거에 대해서 다루도록 하겠습니다.

이런 부가적인 정보를 많은 경우에 휴리스틱이라고 얘기를 합니다.

그래서 휴리스틱을 사용하는 Informed Search Strategies에서 기본 아이디어는 무엇이냐 하면, 우리가 노드를 펼쳐놓고 그 노드에 대해서 깊이 탐색을 하기 전에,

그 노드가 얼마나 우리 문제를 푸는 데 있어서 좋은 노드인지, 바람직한 노드인지를 판단해줄 수 있는 평가함수를 정의하자는 것입니다.

그래서 그 평가함수에 따라서 평가함수 값이 높은 노드부터 일단 더 깊이 펼치고 더 탐색을 해나가자. 이게 Informed Search의 기본 아이디어라고 할 수가 있겠습니다.

그래서 실제로 저희가 Fringe라는 용어를 사용했었죠. 그러니까 펼쳐지긴 했지만 아직 깊이 탐색하지 않은 노드들이 있을 때

그 노드들을 단순히 가장 얕은 노드부터 펼칠지, 혹은 가장 깊은 노드부터 펼칠지, 이런 식의 정해진 룰대로 하는 것이 아니라,

Evaluation function을 보고 가장 바람직한 노드부터 하나둘씩 펼쳐나가는 그런 탐색 전략이라고 말씀드릴 수 있겠습니다.

그래서 Informed search strategy 중에서 오늘은 특히 Greedy best-first search랑 A\* search라는 가장 대표적인 방법에 대해서 말씀을 드리도록 하겠습니다.

그럼 우선 Greedy best-first search부터 말씀을 드리도록 하겠습니다.

이 문제는 지난 시간에 저희가 함께 다루었던 루마니아 문제입니다.

저는 Arad라는 도시에 있는데 내일까지 Bucharest에 가고자 하고요. 그리고 되도록 이면 최단 거리를 통해서 원하는 목적지에 도달하고자 합니다.

그런데 예를 들어서 저희의 추가적인 정보로 예를 들어 각 도시에서 Bucharest까지의 직선거리를 알 수 있다고 한번 가정을 해보죠.

직선거리라는 것은, 즉 문제정의에 꼭 있어야 되는 정보는 아니지만, 우리가 추가적으로 가지고 있는 정보입니다.

그럼 이 정보를 활용해서 저희가 더 나은 탐색 방법을 제안할 수 있다는 것입니다.

그래서 일단 Greedy best-first search에서는 저희가 조금 전에 말씀드린 평가함수를 휴리스틱 함수의 값을 그대로 활용을 합니다.

그러니까 조금 전의 예제 같은 경우에 저희가 휴리스틱 한 정보로서 각 도시에서 Bucharest까지의 직선거리를 알 수 있다고 했기 때문에,

도시를 하나하나씩 탐색해나갈 때 되도록이면 Bucharest까지의 거리가 가장 짧은 노드부터 하나씩 더 깊이 펼쳐나가는 게 낫겠죠.

그래서 저희는 Evaluation function으로 Heuristic Function을 쓰고, Heuristic Function은 직선거리이고, 그것을 기반으로 해서 하나하나씩 노드를 탐색하는 과정을 한번 적용을 해보도록 하겠습니다.

그러면 예제와 함께 다시 보도록 하죠. 저희가 지금 Arad라는 도시에 있습니다. Arad는 Bucharest까지 366㎞ 밖에 있는 도시죠.

지난번처럼 Arad에서 우리가 직접적으로 갈 수 있는 도시가 세 군데였습니다. Sibiu, Timisoara, Zerind이 3가지 도시가 있는데요.

그 중에서 휴리스틱으로 저희가 직선거리를 사용하기로 했지 않습니까? 그래서 그 평가함수 값으로서 휴리스틱 값이 직선거리를 활용을 하고자 합니다.

그래서 이 세 도시를 보시면 Bucharest까지의 직선거리가 가장 짧은 곳이 Sibiu입니다. 그래서 Sibiu를 더 깊이 탐색을 해나가게 되는 것이죠.

그러면 여기에서 보시다시피 Sibiu에서 바로 갈 수 있는 인접 도시들을 하나씩 나열을 했습니다. 그러면 4가지 도시가 도달 가능한 곳인데요.

그중에서 또 Heuristic Function을 활용을 해서 최단거리가 가장 가까운 도시가 어디인지 살펴보니, Fagaras라는 도시가 가장 가까운 것으로 밝혀졌습니다.

그러면 지금 여기에 아직 깊이 탐색하지 않는 노드가 총 6개가 존재하게 되는데, 이 6개 중에 Bucharest까지의 거리가 가장 짧은 Fagaras, 그 도시부터 더 탐색을 해나가게 되는 것입니다.

그럼 Fagaras의 주변 도시를 찾아보면, Sibiu라는 도시와 운이 좋게도 저희의 목적지인 Bucharest가 Fagaras에서 바로 도달할 수 있는 곳이었습니다.

그래서 이런 식으로 search를 해나가면 훨씬 더 빠르게 저희가 원하는 goal까지 도달할 수 있겠죠.

만약에 여러분이 Breadth-first search를 적용한다고 하면 Timisoara나 Zerind 같은 도시를 계속 탐색을 해나가야 될 텐데, 그럴 필요가 없었고요.

만약에 여러분이 Depth-first search를 적용했다고 하면 Arad에서 또다시 search해 나간다든지 그런 비효율적인 탐색 과정이 필요한데,

저희가 휴리스틱 함수를 활용을 해서 그 휴리스틱 함수 기반으로 가장 좋다고 생각되는 도시를 하나하나씩 펼쳐나감으로써 훨씬 더 목적지까지 빠르게 탐색해나갈 수 있는 것입니다.

그래서 이와 같은 방법이 Greedy best-first search라고 보실 수가 있겠습니다.

그럼 이제 첫 번째 알고리즘에 대해서 말씀을 드렸고요. 그다음 Informed Search Strategies의 가장 많이 사용되는 것 중에 하나가 A\* 알고리즘이라고 합니다.

그래서 A\* Search에 대해서 좀 더 깊이 말씀을 드리도록 하겠습니다. A\* Search는 Evaluation function을 정할 때 추가적으로 한 가지 정보를 더 활용을 합니다.

여기에서는 g(n)이라고 되어 있는데, g(n)이라는 건 n 노드까지 오는 데 걸린 시간이었습니다.

그래서 best-first search에서는 f(n)은 h(n)과 동일하다고 활용을 했는데, 여기에서는 g(n)이라는 정보까지 더한 값을 평가 함수로 활용하겠다는 것입니다.

그래서 여기에서 g(n)이라는 건 조금 전에도 말씀드렸다시피 노드 n까지 오는 데 걸린 cost, 비용이었고요. h(n)은 현재 노드 n에서 목적지까지 실어야 할 비용입니다.

그런데 여기에서 중요한 거는 estimated cost라는 거죠. 그러니까 우리가 실제로 현재 노드 n에서 goal까지 얼마의 비용으로 가는지는 탐색을 해봐야 아는 것이고요.

여기에서 h(n)이라는 것은 우리가 조금 전에 말씀드린 Heuristic Function을 이용을 해서 어느 정도 정확한 값은 아니지만 가늠할 수 있는 값으로 사용을 한 것입니다.

그래서 조금 전의 예제에서는 직선거리를 활용을 했었죠.

그래서 결국에는 특정 노드가 얼마나 좋은지, 안 좋은지, 그 노드를 펼쳐야 될지, 말아야 될지를 결정하는 평가 함수로 g(n)과 h(n)을 합한 값을 활용을 하게 됩니다.

그래서 아이디어를 보시면 기본적으로 현재 노드들 중에서 goal까지 도달하는 데 가장 빠른 노드를 펼치는 동시에 그리고 또 하나 추가된 내용은

그 노드까지 가는 데 있어서 이미 비싼 비용을 치렀다고 하면, 그 노드를 더 이상 펼치지 말고, 비용이 더 작은 노드를 중점적으로 활용하자는 것입니다.

그러니까 어떤 노드까지 오는 데 걸린 cost, 그 노드로부터 goal까지 가는 데 걸린 예측된 cost값을 더한 그 값을 활용을 해서 이제 노드를 펼쳐나가겠다.

그게 A\* 알고리즘의 기본 아이디어라고 할 수 있겠습니다.

그래서 여기에서 Heuristic Function에 대해서 다시 한 번 말씀을 드리는데, Heuristic Function은 어떻게 보면 우리가 실제로 계산해야 될 값은 아니지만,

그것을 가늠한, estimated한 값이라고 보시면 되겠습니다. 그래서 실제 여러분이 Heuristic Function을 어떻게 정할까가 실제 탐색 문제에서 되게 어려운 문제이긴 한데요.

Heuristic Function을 좋은 거를 만약에 쓴다고 하면, 그 Heuristic Function을 활용하면 더 빨리 탐색을 할 수 있겠죠.

그렇지만 Heuristic Function 중에 별로 안 좋은 휴리스틱을 쓰면 그만큼 더 안 좋아질 테고요.

많은 경우에 여기에는 트레이드오프 관계가 있습니다.

그러니까 우리가 Heuristic Function을 만들려고 하면 실제 문제를 더 잘 이해하고, 더 많은 정보를 활용해서 그래야만 더 좋은 휴리스틱 함수를 만들 수가 있을 텐데요.

그렇다고 하면 휴리스틱 함수를 만드는 데 걸리는 노력, 그리고 그 함수를 썼을 때 optimal 값을 더 빠르고 정확하게 찾을 수 있는 노력, 그 둘 사이에는 트레이드오프 관계가 있다는 것입니다.

그래서 많은 경우에 휴리스틱 함수는 문제에 대한 전문가가 정확히 문제를 풀기에 앞서서

이 문제를 푸는 데 있어서 이런, 이런 정보가 유용할 것이라는 어떻게 보면 guess, 예측을 기반으로 해서 정의를 하게 되는데,

그거는 좋은 휴리스틱 함수를 정의하는 노력, 그리고 그 함수를 활용을 해서 goal을 찾는 노력 간 트레이드오프 관계가 있다. 이렇게 보시면 되겠습니다.

그러면 A\* 알고리즘에 대해서 말씀을 드리도록 하겠습니다.

여기 보시다시피 앞에 있는 숫자는 f(n)입니다. evaluation 함수고요. 첫 번째 함수는 g(n)이죠. 그러니까 이 노드까지 오는 데 걸린 시간이 cost입니다.

그리고 두 번째 더하는 숫자는 h(n)입니다. 이거는 우리 예제에서 Arad에서 Bucharest까지 직선거리였죠.

이 경우에는 처음에 초기노드, Arad가 초기노드이기 때문에 g(n) 값은 여기에서 0이 됩니다. 그러면 Arad 주변에 갈 수 있는 도시가 조금 전에 보셨다시피 3개의 도시가 있었죠.

예전에는 어떻게 보면 각 도시에서 goal까지의 최단거리를 가지는 도시를 먼저 살펴봤다고 한다면,

여기에서는 그 도시까지 가는 데 걸린 cost, 그리고 그 도시에서 goal까지의 최단거리, 그거를 활용하게 됩니다.

그래서 보시면 Sibiu가 g(n)이랑 h(n)의 합에 가장 적기 때문에 Sibiu라는 노드부터 탐색을 진행하게 됩니다.

그럼 Sibiu 주변에 4개의 도시가 있었고요. 그러면 처음 Fringe에는 아직 탐색하지 않은 노드가 총 6개가 있습니다. 6개가 있는데 마찬가지로 각각에 대해서 f(n)을 계산을 한 것입니다.

그래서 각 노드 밑에 있는 숫자 중에 첫 번째 숫자는 초기 state에서 현재 state까지 오는 데 걸린 cost고요.

두 번째는 그 노드에서 goal까지 거리입니다. 그래서 이 둘을 다 합해 보니까 가장 cost, f(n)이 작은 도시가 Vilcea입니다. 이 경우에는 총 f(n)이 413이라는 값을 가지게 되었습니다.

그래서 6개 주의 가장 작은 값을 가지는 노드입니다. 그래서 이 노드 주변에 있는 도시들을 다시 한 번 살펴봅니다.

살펴보면 이제 또 3개의 도시가 나오는데요. 그 각각의 3개의 도시에 대해서 f(n)을 다시 다 구해봅니다. 그러면 아직 탐색하지 않은 노드가 총 8개가 있는데, 그 8개 중에 마찬가지로 f값이 제일 적은 도시를 선택을 하게 됩니다.

이 경우에는 Fagaras라는 도시가 f값이 가장 작기 때문에 이 노드에 대해서 또 탐색을 시작합니다.

그런데 조금 전에 예제에서 보셨다시피 Fagaras에서 Bucharest까지 갈 수 있는 path가 존재했었죠.

그래서 앞에 Greedy best-first search에서는 Arad에서 Sibiu, Fagaras, Bucharest까지 가는 것을 최단거리로 예측을 했는데요.

마찬가지로 그 노드를 저희가 발견을 했습니다. 그렇지만 A\* Search 에서는 여기에서 그냥 끝내는 게 아니라, 혹시라도 더 좋은 솔루션이 있는지 찾는 과정을 한 번 더 거치게 됩니다.

그러니까 지금 말씀드린 이 path를 통해서 Bucharest까지 가는 데 걸린 f(n)은 450이었습니다.

그러면 예를 들어서 옆에 있는 Sibiu라는 도시로 가는 데 값을 한번 비교해보죠. 이 값은 591입니다. 따라서 Sibiu는 더 이상 탐색할 필요가 없습니다.

왜냐하면 여기에서도 탐색해봐야 지금까지 찾은 솔루션 450이라는 값보다는 항상 클 거기 때문에 Sibiu는 더 이상 탐색을 하지 않아도 좋습니다.

Arad도 646인데 450보다 큰 수이기 때문에 서치를 하지 않아도 되고요. 마찬가지로 450이라는 값보다 더 큰 값을 가지는 노드는 더 이상 찾을 필요가 없습니다.

하지만 여기 Arad, Sibiu, R.V, pitesti로 가는 path를 한번 보죠. 여기에서는 f값이 417입니다. 417은 어쨌든 450보다 작은 숫자죠.

그래서 이 노드를 따라 가면 혹시라도 더 짧은 거리를 통해서 Bucharest까지 갈 수 있는 여지가 있기 때문에 이 노드에서는 한 번 더 탐색을 수행하게 됩니다.

그래서 Pitesti 주변에 도시를 하나씩 펼쳐 보는데, 그 인접 도시로서 Bucharest까지 존재를 했죠. 그런데 인접 도시로 가는 이 path를 통해서 Bucharest까지 가는 함수 f 값은 418입니다.

이 418이라는 값은 우리가 그 전에 찾은 Arad, Sibiu, Fagaras, Bucharest path에 비해서 더 작은 값을 가지게 됩니다.

따라서 어떻게 보면 A\* 알고리즘을 통해서 우리가 기존에 찾은 솔루션보다 더 나은 솔루션을 찾았다. 이렇게 보실 수가 있습니다.

그래서 이게 어떻게 보면 A\* Search와 Greedy best-first search와의 차이점인데요. A\* 알고리즘을 쓰면 많은 경우에 더 optimal한 솔루션을 찾을 수 있다는 장점이 있습니다.

그러면 여러분이 자연스럽게 궁금해질 질문 중 하나가 이것입니다.

언제 A\* 알고리즘을 사용했을 때 최적의 솔루션을 찾을 수 있는지 그게 아마 궁금하실 거고요.

그것에 대해서 얘기하기 위해서 하나의 정의를 소개시켜드리고자 합니다. 휴리스틱 함수에 대한 정의인데요. 휴리스틱 함수가 admissible하다고 얘기하는 경우가 있습니다.

이거는 어떤 경우냐 하면, 우리가 사용하는 휴리스틱 함수 값이 실제 그 노드에서 goal까지의 true cost보다 항상 작거나 같을 경우를 우리가 admissible이라고 합니다.

그러니까 우리가 항상 under estimate 한다는 거죠. 실제 값보다.

우리의 휴리스틱 함수가 실제 cost보다 항상 under estimate, 실제 값보다 항상 작거나 같게 예측을 하면 그 휴리스틱 함수를 admissible하다고 얘기를 합니다.

우리가 조금 전에 본 예제에서 휴리스틱 함수로서 어떤 도시와 goal까지의 직선거리를 활용했습니다. 그런데 아시다시피 직선거리는 언제나 실제거리보다는 작겠죠.

실제거리는 도시와 도시 사이에 나 있는 도로를 따라서만 가야 되기 때문에 항상 직선거리에 비해서는 실제거리가 더 길게 됩니다.

그렇기 때문에 지금 저희가 본 루마니아 예제에서의 휴리스틱 함수는 admissible하다고 할 수 있겠죠.

그래서 만약에 여러분의 휴리스틱 함수가 admissible하다고 했을 때 A\* 알고리즘을 활용을 한다면 언제나 최적의 솔루션을 찾을 수가 있습니다.

이게 되게 중요한 결과라고 할 수가 있겠죠. 그러니까 여러분이 휴리스틱 함수를 적용을 하면 많은 경우에 더 나은 솔루션을 더 빠르게 찾을 수가 있고요.

그리고 만약에 그 휴리스틱 함수가 admissible하다는 그 조건을 만족을 한다면 여러분은 A\* 알고리즘으로 항상 최적의 솔루션을 구할 수가 있습니다.

그러면 다음 슬라이드에서는 왜 admissible 하면 A\* 알고리즘이 optimal한 솔루션을 주는지 그거에 대해서 간단한 증명을 살펴보도록 하겠습니다.

이거는 어떻게 보면 개념적인 증명이라고 할 수가 있겠고요. 여기는 3가지 정의가 일단 있습니다.

첫 번째, G2라는 건 G2라는 그런 예도 나와 있다시피 suboptimal goal이다. suboptimal goal라는 거는 G2가 goal이긴 한데, 최적의 goal은 아니라는 것입니다.

그러니까 start부터 G2까지 간다고 하면, 이 path로 간다고 하면, 여러분이 Bucharest라는 도시에 갈 수 있긴 하지만, 그런데 이게 최적의 path가 아닌 경우를 G2라고 합니다.

그리고 여기에서 G라고 하는 노드는, 이거는 최적의 path를 통해서 goal을 도달한 경우라고 볼 수가 있겠습니다.

그리고 여기 n이라는 노드가 있는데, n이라는 노드는 아직 펼쳐지지 않은 노드입니다.

fringe에 있어서 아직 펼쳐지지 않은 노드인데, n을 만약에 G까지 가는 optimal path에 속해져 있는 노드라고 한번 가정을 해보죠.

그러니까 n은 아직 펼쳐지지 않았지만 얘는 optimal path상에 존재하는 노드다. 이렇게 한번 가정을 해보죠.

그러면 일단 여러 가지 스텝을 통해서 증명을 해보도록 하겠습니다. 첫 번째 증명은, 첫 번째 결론은 f(G2)가 g(G2)랑 똑같다는 뜻입니다.

왜냐하면 G2는 아까 말씀드렸다시피 goal이기 때문에 G2에서의 Heuristic Function의 값은 0입니다.

그러니까 G2에서 goal까지의 estimate distance는 G2가 goal이기 때문에 당연히 0이 되어야 하겠죠. 그래서 G2에서는 f값과 g값이 동일하게 됩니다.

그러면 이제 g(G2)랑 g(G)를 한번 비교를 해보죠. 여기에서는 항상 G2의 g값이 G에서의 g값보다 크게 됩니다.

왜냐하면 조금 전에 우리가 말씀을 드렸다시피 G2는 suboptimal goal이고, G는 optimal goal입니다.

그런데 이제 g라는 함수가 시작점에서 그 노드까지 가는 데 걸린 cost이기 때문에 당연히 suboptimal goal의 G값보다는 실제 optimal의 G값이 훨씬 크게 되죠.

그래서 두 번째 조건도 우리가 쉽게 알아낼 수가 있습니다. 그리고 세 번째 결과는 G에서의 f 값과 g 값은 동일하다는 것입니다.

왜냐하면 조금 전에 말씀드린 대로 g는 goal이기 때문에 여기에서의 휴리스틱 함수값은 0이 됩니다. 따라서 이거를 얻을 수가 있겠고요.

그러면 위에 밝혀진 3가지 룰을 통해서 우리가 f(G2)는 f(G)보다 크다는 것을 알 수가 있겠습니다.

왜냐하면 f값이라는 것은 g랑 h의 합이었죠. 따라서 G2에서의 g값이, G에서의 g값보다 크고, h값은 둘 다 0이기 때문에 당연히 f값을 비교해보면 G2에서의 f값이 G에서의 f값보다 크게 됩니다.

그래서 지금 말씀드린 정의에 의해서 하나하나씩 밟아가면서 결국에는 f(G2)가 f(G)보다 크다는 결론을 도출할 수가 있습니다.

그럼 이제 계속 이 유도를 한번 진행해나가도록 해보죠. 여기에서 첫 번째 수식은 전 슬라이드의 결론을 그냥 단순히 카피를 한 것입니다.

그리고 두 번째 수식의 경우에는 저희가 h가 admissible하다고 가정을 했기 때문에 h(n)은 실제 n에서의 true cost보다는 항상 작거나 같게 됩니다.

그리고 세 번째 수식은 두 번째 수식으로부터 바로 올 수가 있는데, 두 번째 수식에서 양쪽에 g(n)이라는 값을 동시에 더하게 됩니다.

그래서 부등호의 경우에는 똑같은 수를 양쪽에 더하면 그 부등호 방향은 바뀌지 않기 때문에 세 번째 결론도 쉽게 얻을 수가 있고요.

그다음에 마지막 결론은 저희가 말씀드렸다시피 n이라는 노드는 시작 노드에서 g라는 goal 노드까지 가는 중간에 있는 노드이기 때문에 f(n)은 항상 f(G)보다 작거나 같아야 합니다.

그러니까 cost는 노드를 가면 갈수록 항상 커지거나 같아야만 하기 때문에 이와 같은 결론을 얻을 수가 있겠죠.

그래서 결국에는 첫 번째 수식과 네 번째 맨 마지막 수식을 한번 활용을 해보죠.

그래서 f(G2)가 f(G)보다 항상 큽니다. 그런데 동시에 f(n)은 f(G)보다 항상 작거나 같기 때문에 결국에는 f(G2)는 f(n)보다 항상 크다는 것을 알 수가 있습니다.

그러니까 A\* 알고리즘을 우리가 활용한다면 f값을 이용해서 저희가 어떤 노드를 펼칠지를 결정을 하는데,

여기에서 알 수 있는 사실은 G2를 n 대신에 펼치는 일은 절대 일어나지 않는다는 것입니다.

A\* 알고리즘은 f 값을 활용을 해서 f 값이 작은 노드부터 하나씩 펼쳐지게 되는데,

f(G2) 그러니까 suboptimal한 노드는 n, 그러니까 optimal한 path 상에 있는 노드 대신에 펼쳐지는 경우가 절대 없다는 것입니다.

그래서 이것을 계속 반복을 하다 보면 결국에는 저희가 goal, 노드 g까지 도달하는 데 있어서 노드 G2를 펼치는 일은 절대 일어나지 않는다.

이것을 저희가 증명을 통해서 확인할 수 있었습니다. 그래서 본 수업에서는 저희가 Informed Search 알고리즘에 대해서 말씀을 드렸고요.

이 경우에는 저희가 문제를 정의하는 데 필요한 4가지 요소 외에 다른 정보가 주어졌다고 가정을 하고요. 그리고 그 정보를 Heuristic Function을 통해서 얻게 됩니다.

그리고 Heuristic Function은 트레이드 오프 관계가 있어서 좋은 Heuristic Function을 가지고 있으면 더 나은 솔루션을 쉽게 찾을 수 있지만,

그런데 좋은 Heuristic Function을 찾는다는 것 자체, 그것을 정의한다는 것 자체가 더 많은 노력이 필요하겠죠. 그래서 그 사이에는 트레이드 오프 관계가 있다는 것이고요.

그리고 A\* 알고리즘은 기본 아이디어는 그 노드까지 오는 데 걸린 cost,

그리고 그 노드에서 goal까지 가는 데 예측된 휴리스틱 함수에 cost가 합쳐서 그 합이 최소가 되는 노드부터 하나씩 펼쳐나간다는 것이고요.

그래서 A\* Search가 더 나은 알고리즘, 그러니까 Greedy best-first search보다는 더 낫다는 것을 여러분이 쉽게 이해를 하실 거고요.

그리고 A\* 알고리즘 같은 경우에는 저희 Heuristic Function이 admissible하다.

이런 조건을 만족한다면 항상 A\* 알고리즘은 optimal solution을 저희가 Tree search를 통해서 찾을 수 있다. 그것을 증명을 하였습니다.

그러면 오늘 수업은 이렇게 요약을 할 수 있을 것 같고요. 그러면 다음 시간에 다시 만나 뵙도록 하겠습니다.

3.2.

이제는 지역 탐색이라고 불리는 테크닉에 대해서 함께 알아보도록 하겠습니다. 지역 탐색은 영어로는 Local Search라고 하고요. 실제로는 Local Search 방법이 매우 많이 활용이 되고 있습니다.

여러분이 요즘 많이 듣고 있는 딥러닝이라는 기술도 딥러닝 모델을 학습하는 과정에서 Local Search 방법을 아주 적극적으로 활용을 하고 있고요.

그래서 실제로는 아주 많이 사용 방법이다. 이렇게 이해하시면 되겠고요. 그래서 Local Search를 언제 활용하느냐?

이런 경우를 한번 보시면 많은 경우에 목적까지의 path가 중요하지는 않고, 그냥 목적을 달성하는 그 자체가 중요한 경우가 매우 많습니다.

물론 지금까지 우리가 살펴본 루마니아의 예제에서는 최단거리로 가는 게 당연히 좋긴 하겠지만,

예를 들어서 여러분이 체스 게임을 탐색 문제로 한번 정의를 해본다고 하면 물론 체스도 마찬가지로 최단수를 둬서 상대방을 제압하는 게 좋긴 하겠지만,

많은 경우에는 이기는 것, 그러니까 상대방의 왕을 잡는 것. 그거 자체가 목적이 되겠죠.

그런 경우에는 path cost를 고려한다기보다는 goal에 도달하는 그것에 더 집중을 한 그런 문제라고 볼 수 있겠습니다.

그래서 저희가 이 예제에서는 다음 슬라이드에 이제 소개가 될 텐데, n-queens 문제라는 게 있습니다.

그래서 자세한 내용은 다음 슬라이드에서 더 자세히 말씀을 드리도록 하겠고요.

그래서 지금 말씀 드린 대로 goal 자체를 찾는 게 중요한 경우, 그 경우에는 지역 탐색 방법을 많이 활용을 한다는 거고요.

기본 아이디어는 이와 같습니다.

제가 현재 어떤 state에 있다고 할 때 내가 어떤 액션을 취해서 바로 갈 수 있는 인접한 state들 중에 지금 내 state보다 낮다고 판단되는 데가 있으면 그거를 그냥 취하겠다는 겁니다.

그러니까 여러분이 문제를 풀 때 goal까지의 거리, 아주 global한 reasoning을 통해서 문제를 해결하겠다는 게 아니라,

그냥 매 state에서 ‘지금 보다 더 나은 state로만 가 보자.’ 그런 식으로 하나하나씩 가는 방법이 Local Search Algorithms이라고 할 수 있겠습니다.

그렇게 함으로써 여러 가지 장점이 있는데 그중에 가장 대표적인 2개를 살펴보면, 일단은 메모리를 매우 적게 필요로 한다는 것이죠.

그러니까 현재 state만 기억을 하고 있고요. 그리고 현재 state에서 주변 state 정도로 탐색을 하면 되기 때문에 메모리가 훨씬 적습니다.

현재 state에서 2번 스텝을 갔을 때 어떻게 될 것이냐? 이런 것까지 고려하지 않고 그냥 바로 내가 갈 수 있는 인접한 노드들만 탐색을 해나가겠다. 그게 기본 아이디어라고 할 수 있고요.

그리고 실제로 저희 문제들 중에서 Search space 혹은 State space가 엄청나게 큰 문제들이 많이 있는데요.

그래서 그런 문제를 전역적으로 global하게 분석을 하기 위해서 정말 많은 시간 계산이 필요할 텐데,

그런 과정을 다 없앰에도 불구하고 많은 경우에 꽤 괜찮은 솔루션을 얻게 된다. 그런 실질적인 장점이 있다고 할 수 있겠습니다.

그래서 좀 전에 말씀드린 n-queens 문제에 대해서 좀 더 깊이 말씀드리도록 하겠습니다.

여러분이 체스의 룰을 아실 수 있고, 모르실 수도 있지만, 그중에 queen(왕비)라는 말은 가로 혹은 세로로 어디든지 이동할 수 있고요. 또 대각선으로도 움직일 수가 있습니다.

그래서 그런 queen이 n개가 있다고 가정을 해보시면 되겠고요. 그래서 현재 판은 n × n 보드가 있습니다. 이 example에서는 가로 넷, 세로 넷, 그래서 4 × 4 보드라고 할 수가 있겠죠.

4 × 4 보드 안에 queen, 그러니까 왕비 말을 4개를 놓습니다. n개를 놓는 거죠.

우리의 목적은 무엇이냐 하면 queen이 서로 공격하지 않는 마지막 goal을 도달해보자. 그게 n-queens 문제의 목적이라고 할 수 있겠습니다.

여기에서 보시면 첫 번째 example에서 보시면 이 queen과 이 queen은 서로 대각선에 있기 때문에 얘네들은 서로 공격을 하게 되죠.

이런 게 하나도 없는 상태로 말을 놓아 보자. 이게 n-queens 문제라고 정의할 수 있겠습니다.

그래서 만약에 이런 문제 같은 경우에 조금 전에 말씀드렸다시피 물론 최단거리로 이런 배열을 찾는 게 중요하다고 할 수 있지만,

많은 경우에는 배열을 찾는 것 자체가 되게 도전적이고, 그것을 도달하는 것 자체가 어렵기 때문에, 그거 자체가 목적인 경우가 매우 많다고 할 수 있습니다.

이 경우에는 Local Search, 지역탐색 방법을 활용해서 해를 구할 수가 있습니다. 그래서 첫 번째, 지역탐색 아이디어 중에서 Hill-Climbing Search라는 거를 말씀드리도록 하겠습니다.

그러니까 지역탐색 방법은 여러 가지가 있는데요. 그중에서 가장 간단한 Hill-Climbing Search라고 불리는 방법에 대해서 말씀을 드리도록 하겠습니다.

제목에서 알 수 있다시피 이거는 일종의 언덕을 올라가는 그런 과정을 모사를 해서 만들어진 탐색 방법이라고 할 수가 있겠고요.

첫 번째 문장은 Hill-Climbing Search의 아이디어를 비유와 함께 설명한 것입니다. 그러니까 여러분이 에베레스트 산을 올라가는 것입니다.

현재 매우 깊은 안개 안에 있는 것입니다. 그러니까 여러분이 멀리 볼 수는 없고요. 바로 주변, 발 앞에서 앞뒤로 1m 정도만 볼 수 있는 상황이죠.

그런 상황이고, local하게만 볼 수가 있고요. 여러분이 global하게는 볼 수 없는 상황을 얘기를 하고 있습니다.

그리고 또 하나 중요한 가정 중에 하나는 기억상실증에 걸린 거죠.

그러니까 여러분이 내가 이 위치를 기존에 왔었는지, 안 왔었는지는 모르고 제가 그 state에서 주변만 보고 그다음 state로 가고 그거를 반복한다는 것입니다.

그래서 만약에 재수가 없는 경우에는 제가 있었던 곳으로 다시 돌아오는 그런 경우도 있겠지만, 그거까지 고려를 하지 않는다는 것입니다.

그래서 지역탐색의 기본 철학 중 하나가 말씀드렸다시피 메모리를 최대한 적게 활용하자는 게 목적이었기 때문에

우리가 지금까지 온 모든 path에 대해서 다 기억을 하는 거는 그것조차도 어려운 문제가 꽤 많다는 것이죠.

그래서 여러분이 에베레스트 산을 완전히 시야가 확보되지 않은 상태에서 올라간다고 하면 그냥 여러분 주변에서 경사가 제일 높은 곳으로 한발, 한발씩 올라갈 수밖에 없겠죠.

그거와 같이 지역탐색을 수행하는 과정을 Hill-Climbing Search라고 얘기를 합니다.

여기에서는 또 다른 얘기로는 Steepest ascent라는 얘기를 합니다. 그러니까 가장 급한 쪽으로 올라가겠다. 그렇게 이해를 하시면 되겠고요.

여러분이 만약에 Optimization, 최적화 문제를 푸는데, 여러분이 최적화의 목적함수를 최대화하고 싶다. Maximization 하고 싶다.

그런 경우에는 여러분 주변에 있는 neighbor들 중에 그 목적함수 값이 가장 높은 노드로 가게 되겠죠. 그러니까 그와 같이 반복하는 방법을 Steepest ascent라고 합니다.

그리고 Greedy local search라고 불리는데, 여기에서 Greedy라는 얘기는 여러분이 문제를 길게 탐색하지 않고

바로 내가 지금 취할 수 있는 행동들 중에 가장 좋은 행동을 취하는 방법을 Greedy search라고 하는데,

여기 설명한대로 goal까지 어떻게 갈지, 그리고 내가 이 액션을 취하고 이후에 무슨 일이 벌어질지는 전혀 고려하지 않고 그냥 단순히 가장 좋은 이웃으로만 가겠다. 이와 같이 불리게 됩니다.

그래서 Hill-Climbing Search는 개념 자체는 매우 간단하지만, 한 가지 치명적인 단점이 있습니다. 우리가 만약에 Maximization 하는 문제를 푼다고 했을 때, local maxima에 빠질 수 있다는 것입니다.

여기에서 local maxima는 지역적으로 최댓값을 가지는 영역이라고 보시면 됩니다. 여기에 지금 우리의 목적함수에 대한 그림이 있는데요.

여기 global optimum, global maximum이라고 불리는 포인트가 어떻게 보면 우리 문제에서 최적의 솔루션이라고 할 수가 있죠.

예를 들어서 저희가 지금 보여진 현재 위치에서 Steepest ascent 혹은 Hill-Climbing 아이디어를 활용한다고 그러면

계속 왼쪽, 오른쪽을 봤을 때 값을 증가시키는 방향인 이쪽으로 계속 올라가다가 결국에는 Local Search에 머무르게 되겠죠.

그래서 많은 경우에 Hill-Climbing Search를 할 때는 여러분이 initial point를 어디로 잡느냐에 따라서 결과가 매우 달라집니다.

여러분이 만약에 이 포인트에서 initial point를 찾고 그다음에 Hill-Climbing을 한다고 하면 여러분이 global maximum에 도달할 수 있겠죠.

그 외에 shoulder라는 영역은 평평한 영역인데, 여기 왼쪽은 더 낮은 곳이고, 오른쪽은 더 높은 곳이고요.

flat local maximum이라고 불리는 부분은 local maximum이긴 한데, flat한, 평평한 영역을 얘기를 합니다.

그래서 Hill-Climbing Search는 되게 간단하고 많은 경우에 좋은 솔루션을 주긴 하지만 이런 local maximum, local optimum에 빠지는 문제가 생길 수도 있다고 이해하시면 되겠습니다.

그래서 Hill-Climbing Search를 n-queens 문제에 대해서 한번 적용해보도록 하죠. 그래서 여기에서는 더 어려운 8-queens Problem입니다. 그러니까 총 8 × 8의 그리드가 있고요.

그리고 총 8개의 여왕 말이 배열이 되어 있습니다.

그래서 저희의 목적은 뭐냐? 저희의 목적은 얘네들이 서로 공격하지 않는 그런 배열을 찾는 게 저희의 목적이고요. 그런 목적을 달성하기 위해서 우리가 h라는 변수를 정의하고요.

h는 여왕들 중에 서로 공격하는 여왕의 쌍을 보여주고 있습니다.

그래서 우리가 global minimum이다. global optimal point다. 그 영역은 목적을 만족하는, 그러니까 h가 0이 되는 상황이겠죠.

서로 공격하는 여왕 쌍이 하나도 없는 경우, 그게 우리가 원하는 global minimum이라고 할 수가 있겠습니다.

그런데 예를 들어서 우리가 처음에 문제 시작하기에 앞서서 이렇게 initial state로 queen이 random하게 배치가 되어서 이와 같이 주어졌다고 한번 가정을 해보죠.

이 상황에서는 h가 지금 17입니다. 그래서 이 한 쌍이 지금 서로 공격하고 있고요. 이 한 쌍이 공격하고 있고, 얘와 얘가 서로 공격하고 있고.

이런 식으로 여러분이 하나하나씩 세 나가다 보면 총 17개의 쌍이 서로 공격하고 있다. 이렇게 보실 수가 있습니다.

그래서 우리가 이 주어진 상황에서 서로 공격하는 쌍이 하나도 없는 상태까지 한번 탐색해보자. 이게 현재 문제라고 할 수가 있고요.

Hill-Climbing에서의 기본 아이디어는 지금 저희가 이렇게 8개의 말이 있는데, 이 8개의 말 중에 하나를 움직였을 때 h 값 가장 많이 떨어지는 그 말을 한번 찾아보자는 것입니다.

그러니까 여러분이 8개의 조합을 다 고려하는 게 아니라, 그냥 이 주어진 상황에서 내가 8개 중에 어떤 애를 어디로 옮겼을 때 가장 h값이 크게 줄어드느냐?

그런 식으로 하나하나씩 찾아가는 과정입니다.

그래서 여러분이 이 과정을 계속 반복을 하다 보면 아까 주어진 initial state에서는 최대로 도달할 수 있는 경우가 h가 1인 경우입니다.

그러니까 perfect solution, 그러니까 서로 공격하는 퀸이 하나도 없는 그 상황까지 도달할 수가 없고요.

여러분이 local search를 하다 보면 가장 최선으로 도달할 수 있는 상황이 이와 같은 상황입니다. 그래서 여기에서는 총 한 쌍의 서로 공격하는 queen이 있고요.

8개 중에 어떻게 움직이든 간에 h값을 줄일 수 있는 배열은 존재하지 않게 됩니다. 그래서 이게 장점과 단점을 같이 보여주는 예제라고 할 수 있습니다.

장점은 아주 간단하고요. 단점은 여러분이 어디에서 시작하는지에 따라서 최적의 솔루션에 도달하지 못하는 경우도 있다. 그렇게 이해를 하시면 되겠습니다.

이 단점을 해결하기 위해서 나온 local search 방법 하나가 Simulated Annealing Search입니다. 그러니까 Annealing이라는 거는 ‘담금질’이라고 해석이 되는데요.

많은 경우에 금속을 강도를 높이기 위해서 가열을 했다가 급격히 식혔다, 가열을 했다가 급격히 식혔다 이 과정을 반복하게 되는데,

거기에서부터 아이디어를 얻어서 제한된 방법이다. 이렇게 보시면 되겠습니다.

그래서 key idea는 뭐냐 하면, 우리가 Hill-Climbing 방법에서는 제가 각 스텝마다 움직일 수 있는 액션들 중에서 무조건 제 objective를 최대화하는 그런 모션을 했다고 한다면,

여기에서는 중간 중간 현재 상태보다 좋지는 않지만 안 좋은 move를 허락을 하는데,

그 안 좋은 move를 허락하는 빈도를 처음에는 크게 줬다가 가면 갈수록 그거를 줄여주는 그런 방식이라고 보시면 되겠습니다.

그래서 조금 전에도 살펴보셨다시피 Hill-Climbing Algorithms 같은 경우에는 결코 downhill을 하지 않죠. 얘는 항상 올라가기만 합니다.

그렇지만 Simulated Annealing에서는 가끔 내려가기도 한다는 것이죠. 그래서 이런 식으로 무조건 올라가기만 한다면 local maximum에 빠지는 문제가 있었고요.

그래서 Simulated Annealing의 기본 아이디어는 Hill-Climbing을 따르긴 하지만 동시에 random walk, 그러니까 가끔 더 좋은 솔루션이 아님에도 불구하고 무작위로 움직이겠다는 얘기입니다.

그래서 이런 식으로 무작위성을 넣는 게 많은 경우에 local maxima를 피할 수가 있다. 이렇게 이해하시면 되겠고요.

그래서 이거에 대한 Intuition을 말씀드리기 위해서 gradient descent, 그러니까 아까는 gradient ascent였죠.

가장 좋은 방향으로 올라가는 예제였는데, 여기에서는 우리 목적 함수가 최소화하는 경우입니다. 그래서 가장 급하게 내려가는 그런 예를 한번 보도록 하죠.

여러분이 위에서 볼을 떨어뜨렸는데, 면에 여러 가지 범프가 있어서 얘가 잴 수가 없으면 local minimum에 빠지게 되는 거죠.

지역적으로 움푹 들어간 곳에 빠지게 되는 것입니다. 그런 상황을 한번 가정해봤을 때 여러분이 random하게 전체 판을 이렇게 흔들 수 있다고 가정을 해보시죠.

그러면 local minimum에 빠져 있는 공이 튕겨져 나올 수가 있을 텐데요. 그런 상황을 유사하게 생각해보실 수가 있을 것입니다.

그래서 공을 떨어트리는데 우리가 판을 가끔 흔들 수가 있고요. 그런데 처음에는 많이 흔들다가 가면 갈수록 점점 흔드는 빈도를 줄여나가자. 그게 기본 아이디어라고 할 수가 있고요.

그래서 조금 전에 말씀드린 담금질 아이디어랑 유사성을 보자면 처음 시점은 온도가 높은 시점이고요. 시간이 많이 지난 시점은 온도가 낮은 시점으로 이렇게 가정을 합니다.

그래서 처음에는 온도가 높았다가, 그래서 온도가 높았을 때는 이런 randomness가 더 많이 들어가 있다가, 온도는 시간이 지나면 지날수록 저희가 계속 낮춰줍니다.

낮춰주면서 온도에 따라서 randomness가 결정이 되는 것이죠.

그래서 뒤에 아주 간단한 Algorithms이 주어져 있고요. 하나하나씩 보도록 하겠습니다.

여기에서의 기본 아이디어는 뭐냐 하면 가끔 best move를 선택하는 대신에 random move를 가끔 선택을 하게 되고요.

그리고 만약에 그 move가 현재 상황보다 낫다고 하면 그 move를 통해서 이동한 state를 받아들이고요.

만약에 그게 안 좋다고 그러면, randomness를 통해서 받아들이든지, 안 받아들이든지 그것을 결정하게 됩니다.

그래서 이 Algorithms을 하나씩 보시면 input으로서는 문제와 schedule이 주어져 있고요.

여기에서 보여드린 Algorithms은 pseudo code라고 해서 아주 엄밀하게 만든, 그러니까 Simulated Annealing을 구현하기 위해서 아주 엄밀하게 설계된 Algorithms이 아니라, 개념 위주로 보여드린 거다.

이렇게 보시면 되겠습니다. 그래서 문제가 주어졌고, scheduling 아이디어가 주어진 것입니다.

이 schedule이라는 건 제가 계속 iteration을 통해서 탐색을 해나갈 텐데, 그 iteration마다 온도를 어떻게 줄여나갈지 그거에 대한 함수다. 이렇게 보시면 되겠습니다.

그래서 local variables은 현재 노드, 다음 노드 그리고 온도를 나타내는 t가 주어져 있고요. t는 조금 전에도 말씀드렸다시피 시간이 가면, iteration이 진행되면 진행될수록 점점 낮아지는 값입니다.

그래서 결국에 중요한 for roop 안을 한 번 보도록 하죠. 여기에서 t라는 거는 iteration을 나타내는 값이고요. iteration을 계속 반복하게 됩니다.

그래서 조금 전에 말씀드린 대로 현재의 t값, 그러니까 온도 값을 어떻게 결정하느냐는 temperature의 schedule에 따라서 결정이 됩니다.

그래서 조금 전에도 말씀드렸다시피 schedule이라는 함수를 여러분이 input으로서 주어져야 되는데 그러니까 얼마나 급하게 온도를 낮춰갈지는 여러분이 결정해야 될 문제입니다.

그래서 만약에 t가 0까지 가면 이제 더 이상 탐색을 그만 하고 끝내는 것입니다. 그러니까 이 scheduling 과정에서 몇 번의 iteration을 돌고 탐색을 그만 둘지를 결정을 해야 된다는 것이고요.

그리고 현재 state에서 random하게 다음 state를 결정을 합니다. 그리고 그다음 state의 값과 현재 state의 값의 차이를 delta E라고 정의를 했습니다.

그래서 만약에 delta E가 0보다 크다. 그러면 무슨 뜻입니까? 제가 random으로 next state를 정했는데, 거기에서의 값이 현재의 값보다 더 크다는 거고, 그거는 더 좋다는 거죠.

그런 경우에는 무조건 accept를 합니다. 그러니까 delta E가 0보다 클 때는 제가 이런 식으로 random으로 선택한 다음 state를 현재 state로 만들어버린다.

그러니까 그쪽으로 이동하겠다. 그 얘기고요.

delta E가 0보다 작은 경우가 문제겠죠. 그러니까 delta E가 작다는 거는 현재보다 random으로 선택한 다음 state가 더 안 좋다는 겁니다.

그렇다고 해서 무조건 버리는 게 아니라, 특정 확률을 가지고 받아들일지, 안 받아들일지를 결정을 하겠다는 겁니다.

그래서 이런 exponential 함수를 통해서 정의를 하는데요. 이 경우에는 언제 더 받아들일 확률이 높아지느냐? 그래서 delta E가 클 경우에는 항상 받아들이는데, 중요한 거는 delta E가 작은 경우죠.

그러니까 제가 다음 next state로 옮겼을 때 목적함수의 값으로 떨어지는 경우입니다. 이 경우에는 주어진 확률 분포 값으로 선택을 하게 되는데요.

여기 보시면 exponential 함수 위에 delta E 나누기 t라는 식으로 정의가 되어 있습니다.

그런데 이 수식의 의미를 말씀을 드려 보면 이게 언제 받아들일 확률이 줄어드는지 하면 delta E의 절대 값이 클 경우, 혹은 temperature가 낮은 경우입니다.

delta E의 절대 값이 크다는 거는 무슨 얘기냐 하면, 지금 여기 else문에 들어왔다는 거는 delta E가 항상 마이너스라는 거죠.

마이너스인 상황에서 절대 값이 크다는 거는 내가 그다음 state로 갔을 때 value의 감소가 아주 크다는 겁니다.

그러니까 value의 감소가 크면 클수록 받아들일 확률을 줄여주겠다는 거고요. 또 하나 중요한 parameter는 온도 t인데요. t가 낮을수록 받아들일 확률이 적어집니다.

t가 낮아진다는 거는 iteration이 계속 진행이 돼서 점점 t가 낮아지는 상황이기 때문에, 이 수식을 통해서 처음에는 bad move임에도 받아들일 확률이 높다가,

가면 갈수록 bad move는 받아들일 확률이 떨어지는 그거를 이런 식으로 구현했다고 보실 수가 있겠습니다.

그래서 Simulated Annealing에 대해서 개략적인 Algorithms을 말씀을 드렸는데요. 이 기본 Algorithms을 통해서 다양한 변화들이 있습니다.

Algorithms마다 조금 조금씩 다르긴 하지만 전체 철학은 이대로 따라간다고 보시면 되겠고요.

Local Search, 지역탐색이라는 Algorithms은 어떻게 보면 하나의 큰 Algorithms 그룹을 말씀을 드리는 거고,

이번 시간에는 Hill-Climbing 방법과 Simulated Annealing 방법을 말씀드렸지만, 그 외에도 훨씬 다양한 Algorithms이 있습니다.

유전자 Algorithms이라든지, 그 외에 여러 Algorithms이 존재하게 되고요.

혹시 관심 있으신 분은 이것을 기반으로 해서 좀 더 찾아보시고 공부해보시면 더 유익한 인공지능 공부가 되지 않을까 이렇게 생각을 합니다. 그러면 오늘의 강연은 여기에서 마치도록 하겠습니다.

4.1.

안녕하세요? 서울대학교 컴퓨터공학부의 김건희입니다. 인공지능의 기초 오늘은 강화 학습에 대해서 말씀을 드리도록 하겠습니다.

근래에 많은 분들이 강화 학습에 아주 관심이 많다고 생각하는데요.

가장 큰 이유가 얼마 전 알파고 혁명 때 알파고의 알고리즘이 강화 학습을 기반으로 만들어졌다는 게 알려지면서 많은 분들이 관심을 갖게 되었고요.

그 외에 게임을 하는 인공지능, 그러니까 여러분이 들어보셨겠지만 스타크래프트라는 게임 인공지능을 만드는 연구가 많이 진행되고 있다고 알고 계신데요.

그것도 강화 학습을 기반으로 해서 만들어지고 있다. 이렇게 이해하시면 되겠습니다.

본 강의를 소개시켜드리기에 앞서서 말씀을 드리고자 하는 내용은 여러분이 만약에 강화학습을 정말 잘 알고 싶다고 하면 지금 보여드리는 강의를 한번 들어보시는 게 좋을 수 있습니다.

David Silver라는 분이 university of college london에서 강의한 내용이고요.

David Silver는 아시는 분은 아시겠지만 그때 알파고 때 핵심, 그러니까 전체를 설계한 분으로 소개됐고 같이 참여도 하신 분으로 알려져 있습니다.

그분이 영국 대학교에서 강의를 열었는데 그 강의한 내용을 촬영한 내용입니다. 그래서 여러분이 동영상도 볼 수 있고요. 그리고 lecture note도 같이 다운을 받을 수 있습니다.

그런데 주의하셔야 할 게 강화 학습이라는 주제 자체가 좀 어려운 주제이기 때문에 대학원 정도 수준의 일반인들이 들으시기에는 어려울 수 있다는 게 단점이 될 수 있고요.

그리고 영어로 진행된 강의이기 때문에 영어에도 익숙하셔야 들을 수 있는 좋은 강의입니다.

하지만 전 세계에서 열리는 수많은 강의들 중에 가장 뛰어난 강의라고 꼽히고 있기 때문에 혹시 강화 학습을 깊게 알아보고 싶으신 분들은 꼭 한번 들어보시는 걸 추천해 드리고자 합니다.

그리고 조금 전에 말씀드린 강의도 마찬가지로 이 책을 기반으로 하고 있습니다. 「Reinforcement Learning: An Introduction」이고요.

이 책은 아주 자주 계속해서 업데이트되는 책이고요. 여러분이 서점을 통해서 구입이 가능하고요.

그리고 저자들이 온라인을 통해서 드래프트를 올린 것도 여러분이 받아볼 수 있기 때문에 미완성 책의 경우에는 여러분이 PDF를 받아서 볼 수 있습니다.

그래서 이 책도 마찬가지로 여러분이 강화 학습을 깊게 공부하고 싶다. 이러신 분들은 꼭 한번 찾아보시는 게 좋을 것 같습니다.

그러면 이제 강화 학습에 대해서 말씀을 드리도록 하겠습니다. 강화 학습은 인공지능의 아주 핵심 분야 중의 하나고요. 그러니까 인공지능 에이전트를 학습시키는 과정 중의 하나입니다.

그런데 어떻게 학습하느냐고 하면 이 에이전트 보고 어떤 행동을 하게 한 다음에 그 행동이 잘 했으면 보상을 하고요. 그다음에 잘 못 했으면 벌을 주는 그런 식으로 학습하는 과정입니다.

그러니까 어떤 에이전트에게 ‘뭘 어떻게 하라.’ 이거를 직접적으로 가르쳐주는 게 아니라 그냥 스스로 할 수 있게 한 다음에 그 결과를 놓고 reward 혹은 punishment를 주는 그런 경우라고 할 수 있습니다.

그래서 사람에 대한 학습을 예로 들자면 어린 아이가 빨간불일 때는 차도를 건너면 안 되고 파란불일 때 건너야 하는데 엄마가 애한테 ‘파란불일 때 건너고 빨간불일 때는 건너지 마.’

이런 식으로 가르쳐줄 수도 있지만 또 다른 방식으로는 애가 자기 멋대로 하도록 놔두는 것이죠.

그래서 애가 만약에 파란불일 때 건너면 보상을 해주고요. 빨간불일 때 건너면 벌을 주고, 그걸 반복하다보면 애가 스스로 ‘빨간불일 때는 건너지 말아야겠구나.’ 이걸 학습할 수 있도록 하는 것이죠.

한편으로 보면 비효율적인 학습방법이라고 할 수도 있습니다. 왜냐하면 직접적으로 알려주는 게 아니라

얘 보고 Trial&Error를 하도록 반복한 다음에 저희는 보상과 punishment로 에이전트를 학습하는 거기 때문에

많은 경우에 강화 학습을 통해서 학습을 하면 그 에이전트를 쉽게 학습할 수 없다는 단점은 있습니다.

대신에 강화 학습은 언제 활용이 되느냐면 저희가 그때마다 룰을 일일이 가르쳐주는 게 매우 어려운 환경에서는 강화 학습이 유리할 수 있습니다.

그러니까 게임 같은 인공지능에서는 저희가 룰을 몇 개는 만들어서 알려줄 수 있지만 많은 경우에는 그때마다 임기응변으로 해야 할 상황이 많고요.

그렇기 때문에 그런 환경에서는 저희가 어떻게 일일이 ‘어떻게 해라. 어떻게 해라.’ 학습을 시켜주기보다는 스스로 수많은 게임을 통해서 학습하도록 가르친다고 볼 수 있고요.

마찬가지로 사람도 게임을 잘하게 되는 과정을 보면 이런 강화 학습의 아이디어로 점점 잘해진다고 볼 수도 있습니다. 그래서 학습을 어떻게 하느냐? 보상 함수를 통해서 학습을 하고요.

그다음에 fills in the details. 그러니까 자세한 내용은 에이전트가 스스로 배워야 하는 겁니다. 제가 일부러 가르쳐주는 건 아니고요.

그리고 Feedback, 그러니까 보상 혹은 벌이라는 게 내가 어떤 행동을 했을 때 바로바로 주어질 수도 있지만 많은 경우에는 한참 후에 주어질 수 있습니다.

바둑 같은 예죠. 바둑도 한 번 수를 둘 때마다 저희가 보상을 해주는 게 아니라 끝까지 가보고 게임을 끝까지 해봐야지 ‘졌구나.’ 혹은 ‘이겼구나.’ 이를 통해서 강화 학습을 적용할 수 있겠죠.

그리고 타이밍이라는 게 굉장히 중요합니다.

그러니까 처음에는 강화 학습 에이전트가 거의 아는 게 없다가 얘가 점점 알아가는 게 많아질 텐데 그런 상황에서는 어떤 순서로 얘를 학습시키느냐도 매우 중요합니다.

그래서 non i.i.d라는 거는 i.i.d는 independent identical distribute이란 뜻인데, 그러니까 저희의 학습 데이터가 서로 다 독립 사건이 아니라는 거고요.

그리고 그 학습 데이터가 똑같은 확률분포에서 주어지지 않았다. 그렇게 볼 수 있습니다.

그리고 에이전트가 특정 스텝에서 어떤 액션을 취했느냐가 다음에 얻을 데이터가 어떤 형태로 올 것인지 결정하는 경우가 많습니다.

그래서 일단 학습을 시키는, 그러니까 선생님 입장에서는 시간을 많이 save할 수 있지만

대신에 배우는 자가 더 많은 Trial&Error를 통해서 스스로 배워야 하는 그런 장단점이 있는 학습방식이라고 보시면 되겠습니다.

이 강화 학습 부분은 좀 전에 말씀드렸다시피 advanced, 고급 과정이라고 볼 수 있습니다.

그래서 여러분이 강화 학습 강의를 잘 이해하고 싶은 경우에는 확률통계라든지 여러 가지 선형대수에 대한 기본적인 지식을 선수과목으로 갖출 필요가 있고요.

혹시라도 하나하나 설명해 나가는 과정에서 이해가 안 되는 부분이 있다고 하면 찾아보면서 진행하셔야 하지 않을까 생각합니다.

그래서 기본적인 강화 학습 모델에 대해서 도식화하면 이것과 같습니다. 여기 보시다시피 에이전트가 있고요. 에이전트는 그 주변 환경이 있습니다. 환경이 있고요.

이 주변 환경과의 인터랙션을 통해서 에이전트가 점점 학습해 나가는 과정이라고 보시면 되겠고요.

그래서 강화 학습의 목적이라고 하면 에이전트를 제어하는 정책을 찾는 건데 어떤 정책을 찾느냐?

에이전트가 최대의 보상을 얻을 수 있는 control policy, 행동 혹은 제어 정책을 찾는 것이라고 보시면 되겠고요.

policy라는 단어에 대한 구체적인 정의는 다음 슬라이드에서 다시 말씀드리도록 하겠습니다.

그래서 에이전트 관점에서는 현재 에이전트가 어떤 상태에 있다가 그다음에 주어진 상태에서 내가 취할 수 있는 액션 셋이 여러 개 있겠죠.

액션 셋에 액션이 여러 개가 있을 텐데, 그 중에 하나의 액션을 선택해서 수행을 합니다. 그러면 그 액션을 통해서 이 에이전트의 state는 다음 state로 변경이 되고요.

다음 state로 변경이 됨과 동시에 지금 직전에 한 행동에 대한 reward가 주어집니다.

그러니까 St라는 상태에서 At라는 액션을 취했을 때 St+1이라는 state로 바뀌면서 reward가 얼마나 주어지느냐? 그거에 대한 내용이라고 보시면 되겠고요.

이 과정을 반복함으로써, reward를 계속 관측함으로써 ‘내가 어떤 상태에서는 어떤 액션을 취해야 되겠구나.’ 그걸 스스로 배울 수 있도록 한다고 보시면 되겠습니다.

여기서 가장 중요한 값 중의 하나가 reward입니다. 그러니까 결국에는 에이전트를 학습시키는 게 이 reward라는 scaler, 그러니까 하나의 값을 통해서 컨트롤하게 되기 때문에

이 reward를 어떻게 정의하고 어떻게 주느냐? 이게 참 쉽지 않은 문제고요. 실제로 강화 학습을 적용할 때도 reward를 어떻게 정의하느냐? 그게 참 어려운 일입니다.

그래서 하나하나 중요한 정의에 대해서 말씀을 드리도록 하겠습니다. 에이전트 같은 경우에는 이 3가지를 다 가지고 있는 경우도 있고 일부만 가지고 있는 경우도 있습니다.

그래서 하나하나씩 보면 policy. policy라는 건 뭐냐면 에이전트의 behavior function이다. 그러니까 에이전트가 어떤 상태에서 어떤 액션을 할지 정해주는 함수라고 보시면 되겠고요.

value function이라는 건 뭐냐면 내가 어떤 state에 갔을 때 그 state가 얼마나 좋은지, 그러니까 내가 reward를 계속 maximizing하는 방향으로 행동을 할 텐데

내가 어떤 state에서 어떤 state로 갈 때 그 state가 얼마나 좋은지 나타내는 함수를 value function이라고 하고요.

그다음에 model이라고 하는 건 에이전트가 환경을 어떻게 표현하고 있는지 그거에 대한 모든 것을 모델이라고 통칭합니다.

그러니까 소개시켜드리는 단어 하나하나가 다른 콘텍스트에서는 다른 의미로 쓰이는데 적어도 강화 학습에서는 이런 의미로 쓰인다는 것을 여러분이 확실히 이해하셔야 할 필요가 있고요.

그래서 policy, value function, model 이게 강화 학습에서 제일 중요한 컨퍼런트이고요. 그래서 이 3개의 단어는 강의 내내 계속 나올 것입니다.

이것에 대해서는 확실히 이해할 필요가 있고 지금까지는 개념적으로 말씀드렸다고 하면 하나하나씩 구체적인 정의에 대해서 말씀을 드리도록 하겠습니다.

그래서 policy는 함수입니다. 함수인데 state로부터 액션으로의 함수입니다. 그래서 크게 보면 Deterministic policy가 있고, Stochastic policy가 있습니다.

그러니까 Deterministic이라는 건 뭐냐면 내가 어떤 상태에서 어떤 액션을 할지 정의해놓은 겁니다. 하나하나씩 정의한 거고요.

Stochastic이라는 거는 어떤 state에서 어떤 액션을 취할지 확률분포로 주어지는 겁니다. 그러니까 어떤 액션을 취할 확률이 높긴 하지만 항상 그 액션을 취하는 건 아니다.

그렇게 이해하면 되겠습니다. 그리고 policy는 통상적으로 π라는 notation으로 많이 쓰고요. 그래서 π는 함수인데 s를 받아서 a를 아웃풋으로 내는 함수다. 그렇게 이해하시면 되겠고요.

Stochastic policy 같은 경우에는 좀 전에 말씀드렸다시피 어떤 state에서 어떤 액션을 취할지가 확률분포로 주어지는 거고요.

확률분포이기 때문에 확률분포를 값 하나로 요약하는 게 expectation입니다. 그래서 expectation은 기댓값이라고 정의가 되는데요.

그래서 혹시라도 확률분포에 대해서 약간 배경지식이 적다고 하면 expectation의 의미에 대해서 한번 알아보시는 게, 찾아보시고 이 강의를 다시 듣는 게 좋을 수도 있습니다.

간단히 말씀드리면 어떤 state에서 어떤 액션을 취할지에 대한 확률분포라고 이해하시면 되겠습니다.

그다음에 중요한 정의 중의 하나입니다. value function인데요. 가치 함수라고 한국말로 얘기하고요. 가치 함수는 미래의 보상에 대한 예측입니다. 예측 값이고요.

일단은 이런 식으로 notation을 씁니다. V라는 notation을 주로 쓰고요, value function을 위해서.

이 value function이라는 건 어떻게 정의가 되냐면 만약에 policy가 주어졌을 때 내가 만약에 π라는 policy를 따라서 행동한다고 했을 때 S라는 state의 value가 얼마냐?

이거를 정의한다고 보시면 되겠습니다. 그래서 나중에 가면 value function이 크게 2종류가 있습니다. V로 쓰는 경우와 Q로 쓰는 경우가 있는데 V는 state에 대한 함수고요.

Q는 state와 액션에 대한 함수입니다. 그래서 Q는 나중에 보시면 되겠고요. V부터 이해하시면 된다고 보시면 되겠습니다. 그래서 V는 어떤 policy를 따른다고 했을 때 어떤 state의 value를 나타낸다고.

그런데 저희가 이렇게 π라고 서브스크립트, 그러니까 아래첨자로 써주는 이유는 만약에 policy가 바뀌면 어떤 state의 값이 자동으로 바뀌게 됩니다.

그래서 아래첨자로 π를 썼다는 건 얘가 π라는 policy를 따라갈 때의 value function이 이렇게 주어진다고 보시면 되겠고요. 그래서 정의대로 보시면 제가 S라는 state에 있습니다.

그러니까 타임 스텝 t에 제가 S라는 state에 있을 때 얘가 계속 액션 그리고 state 변화를 반복하게 되겠죠. 반복하게 되는데 일종의 기댓값입니다.

기댓값인데 미래에 대한 reward의 기댓값이고요. 현재 state가 S일 때 내가 미래에 취할 수 있는 reward에 대한 기댓값이라고 보시면 되겠습니다.

그리고 여기 보시다시피 Rt+1은 t+1 스텝에서의 reward, Rt+2는 t+2 스텝에서의 reward, 이렇게 보시면 되겠고 이 γ라는 건 discount function이라고 합니다.

그러니까 먼 미래에 대한 reward는 γ만큼 discount를 해준다는 거죠. 그래서 γ는 많은 경우에 1보다 작은 값을 줍니다.

그래서 만약에 γ²이 되면 훨씬 더 작은 값이 되고, γ³이 되면 훨씬 더 작은 값이 되고요. 그래서 멀면 멀수록 그거에 대한 reward는 discount한 값이라고 보시면 되겠습니다.

그리고 model이라고 말씀드린 게 있었죠. model은 크게 환경에 대한 representation이라고 아까 정의를 했는데 크게 보면 2가지 구성요소로 이루어져 있습니다.

하나는 P이고요, 하나는 R입니다. P라는 건 뭐냐 하면 마찬가지로 제가 S라는 state에 있습니다. S라는 state에 있을 때 a라는 액션을 취하면 S`이라는 state로 갈 확률을 나타냅니다.

그러니까 오른쪽에 기술했다시피 제가 t 스텝의 S라는 state에 있고 그때 내가 만약에 a라는 액션을 취했을 때 그다음 state로 S`으로 갈 확률, 그거를 P라는 notation으로 정의하게 되고요.

그리고 R이라는 건 뭐냐면 제가 S라는 state에 있을 때 a라는 액션을 취하면 그때 얻는 reward다. 그래서 예를 들면 제가 t 스텝에 a라는 액션을 취했습니다.

그리고 t state에서 S라는 state에 있었습니다. 그때 a라는 액션을 취했을 때 그다음 state로 가면서 그때 얻는 reward에 대한 확률분포라고 보시면 되겠습니다.

그래서 강화 학습에서 모델이라고 하면 P랑 R을 지칭한다. 이렇게 보시면 되겠습니다.

그래서 P는 state transition, 그리고 R은 reward에 대한 확률분포다. 이렇게 이해하시면 되겠습니다.

그래서 예제와 함께 이해를 해보도록 하죠. 그래서 이거는 기본적으로 미로에 있습니다. 미로에 있는데 start부터 시작해서 goal까지 가는 게 에이전트의 목적입니다.

그런데 저희가 기대하는 거는 최단거리로 goal까지 가는 걸 기대하는 것입니다. 그럴 경우에는 저희가 reward를 어떻게 주느냐면 얘가 한번 타임 스텝을 할 때마다 reward를 -1로 줍니다.

그러니까 이거는 무슨 뜻이냐 하면 내가 쓸데없는 행동을 하면 할수록 -1을 더 많이 받기 때문에 이 reward 함수를 활용한다면 우리가 최단거리로 goal까지 갈 수 있겠죠.

그리고 액션이라는 건 N, E, S, W. 그러니까 내가 북쪽으로 갈 수도 있고 동쪽으로 갈 수도 있고 서쪽으로 갈 수도 있고 W로 갈 수 있고,

그러니까 각각의 location에서 내가 취할 수 있는 액션은 총4가지입니다. 그리고 여기서 state는 어떻게 정의했냐면 에이전트가 현재 어느 위치에 있는지를 state로 정했다고 보시면 되겠습니다.

그래서 initial state는 스타트 포인트가 되겠고요. goal state는 goal 포지션이 되겠죠.

그래서 이런 식으로 에이전트를 학습하다 보면 결국에는 최단거리로 스타트부터 goal까지 가는 그런 에이전트가 학습되겠죠.

그래서 policy의 한 예제입니다. 여기서는 Deterministic policy를 예제로 보여드리고 있고요.

아까 policy가 뭐라고 말씀드렸죠? policy라는 건 내가 어떤 state에서 어떤 액션을 취할지 나타내는 함수라고 말씀드렸습니다. 그래서 notation으로는 π를 썼고요.

그래서 왼쪽 그림을 보시면 각각의 state마다 화살표가 있습니다.

그래서 화살표의 이름이 뭐냐면 이 에이전트가 만약에 이 위치에 있다고 하면 이 state에 있을 때는 북으로 가는 액션을 취해라. 그게 policy입니다.

그래서 여기 지금 왼쪽 그림에서 주어져 있다시피 모든 state에서 어떤 액션을 취해야 하는지 화살표로 표현한 것이죠. 그래서 이거는 하나의 policy의 예제라고 이해하시면 되겠습니다.

그리고 value function은 아까도 말씀드렸다시피 내가 어떤 policy를 취하고 있을 때 어떤 state의 값입니다. 그러니까 그 state가 얼마나 가치가 있는지 나타내는 값이죠.

그래서 오른쪽의 그림을 보시면 value function의 한 예를 보여주고 있습니다. 그러니까 각 state마다 그 state의 value가 얼마인지 나타내고 있는 것이죠.

그래서 많은 경우에 value function이랑 policy랑 되게 관련이 깊습니다. 만약에 여러분이 이 위치에 있다고 한번 가정을 해보죠.

이 위치에 있다고 했을 때 왼쪽 state의 value는 -16이고요. 아래쪽 state는 -16이고요. 위쪽에 있는 state는 -14입니다. 이럴 경우에는 어떤 policy를 취하게 될까요?

당연히 위쪽으로 가는 policy를 취하겠죠. 왜냐하면 그쪽으로 가는 게 그 state의 value가 가장 최대이기 때문입니다.

그래서 이 value function이라는 건 각각의 state가 얼마나 value가 있는지, 얼마나 가치가 있는지를 나타내고요.

이 예제에서 볼 수 있다시피 goal이랑 가까우면 가까울수록 value값이 큽니다.

그러니까 goal 바로 옆에 있는 state 같은 경우에 -1이이죠. 그리고 start에 가까울수록 value function값이 작습니다.

그러니까 이런 식으로 value function이 주어져 있다고 하면 에이전트는 value function이 커지는 방향으로 계속 이동을 하게 될 테고요.

그러다 보면 value function만으로도 에이전트를 제어할 수 있는 상황이 오게 됩니다.

그래서 나중에도 계속 말씀드리겠지만 value function이랑 policy랑은 되게 밀접한 관계가 있다. 이렇게 보시면 되겠고요.

많은 경우에 하나만 알고 있어도 다른 하나를 유도할 수 있고요. 그에 대해서는 좀 더 자세히 다루도록 하겠습니다.

그리고 model에 대한 내용이죠. model은 아까도 말씀드렸다시피 크게 P와 R이라는 2개의 정보가 있다고 말씀드렸습니다.

P는 state transition, 그러니까 S라는 state에 있을 때 a라는 액션을 취하면 S`이라는 state로 갈 확률을 나타내는 거고요.

reward라는 건 S라는 state에서 a라는 액션을 취했을 때 reward, 그거를 나타낸다고 할 수 있겠죠.

Grid layout이라는 게 여기서 transition model을 정의하고 있다고 보시면 되는데요.

그러니까 여기 흰 색으로 표현됐다는 건 뭐냐면 여기서 이쪽으로도 갈 수 있고 이쪽으로도 갈 수 있다는 것입니다.

그런데 여기서 북쪽으로는 못 가겠죠. 북쪽은 검은색이고 또 마찬가지로 왼쪽으로도 못 가겠죠. 왼쪽도 마찬가지로 검은색이니까.

그러니까 지금 이렇게 흰색으로 표현됐다는 것 자체가 implicit하게 transition model을 정의하고 있다. 이렇게 보시면 되겠고요.

그다음에 여기 지금 숫자들이 이렇게 -1로 채워져 있는데 이 -1이라는 게 reward function, 그러니까 보상 함수라고 이해하시면 되겠습니다.

그래서 저희가 문제 시작할 때 우리가 한번 move할 때마다 reward가 -1씩 주어진다고 했는데, 그거를 표현하면 이런 식으로 주어진다.

그래서 Maze Example에 대해서 저희가 문제 정의를 말씀드렸고 그다음에 policy의 예 하나 보여드렸고, value의 예, 그다음에 model의 예를 하나씩 보여드렸다고 이해하시면 되겠습니다.

그리고 중요한 개념 중의 하나가 Exploration and Exploitation입니다. 그러니까 탐험을 하는 것과 제가 알고 있는 정보를 활용하는 것이죠.

그래서 시작할 때 reinforcement learning, 강화학습을 시작할 때 말씀드렸다시피 그거는 기본적으로 Trial&Error를 반복함으로써 에이전트가 학습하는 과정을 보여드리고 있는데요.

많은 경우에는 제가 경험이 쌓이면 쌓일수록 제 경험상 ‘이런 액션을 취했을 때 reward가 많았으니까 이런 reward를 취해야지.’

이런 식으로 Exploitation, 알고 있는 지식을 활용하는 경우죠. 그런데 그것만 활용해서는 안 됩니다.

왜냐하면 제가 처음에 두 번째로 좋은 액션을 취해서 reward 1을 받았는데 원래 처음에 제일 좋은 액션을 취했으면 reward를 예를 들어서 10을 받을 수 있었는데 제가 미처 그걸 안 했다.

그럴 경우에는 제가 처음에 그걸 안 했으면 계속 그걸 안 하게 되는 그런 현상이 벌어질 수도 있지 않습니까?

그래서 많은 경우에는 Exploitation, 그러니까 내가 알고 있는 정보를 활용해서 reward를 maximizing하는 액션을 하는 것도 중요하지만,

일종의 random action을 취해서 ‘내가 이런 상황에서 어떤 액션을 취하면 이런 reward를 받는구나.’ 그런 경험을 쌓는 과정도 필요합니다.

그래서 강화 학습일 경우에는 Exploration이랑 Exploitation을 반복하고 이 둘 사이에 트레이드오프 관계가 존재한다. 이렇게 이해하시면 되겠고요.

이걸 우리가 일상생활의 예제로 한번 본다고 하면 우리가 식당을 결정할 때도 우리가 알고 있는 가장 맛있는 식당을 가는 경우도 있지만

우리가 가보지는 않았지만 한번 새로운 레스토랑을 시도해볼 수도 있겠죠. 그렇게 새로 시도해보는 걸 Exploration이라고 할 수 있고,

그리고 내가 현재까지 알고 있는 지식을 바탕으로 해서 제일 좋다고 생각되는 레스토랑을 가는 그 과정을 Exploitation 혹은 활용, 이용이라고 볼 수 있습니다.

그리고 또 하나 중요한 2가지 term입니다. 이거는 Prediction이랑 Control이라는 term인데요. Prediction은 말 그대로 일종의 예측이고, 한번 예측을 해본다. 이렇게 해석할 수 있고요.

Control은 ‘제어한다’ 이렇게 볼 수 있는데 reinforcement learning에서의 Prediction은 무슨 뜻이냐 하면 policy가 주어졌을 때 value function을 구하는 것, 그 작업을 Prediction이라고 합니다.

그러니까 어떤 policy π이라는 게 주어졌을 때 가치 함수, 그러니까 Vπ(S)를 구하는 것이죠. π가 주어졌을 때 Vπ(S)를 구하는 과정을 Prediction이라고 합니다.

그리고 Control이라는 건 뭐냐면 문제가 주어졌을 때 최적의 policy, 그러니까 reward를 최대한으로 할 수 있는 policy를 찾는 과정, 그걸 Control이라고 합니다.

그래서 Control은 제어다. 그거는 많은 분들이 아실 테고 제어는 다른 데서는 다른 의미로 쓰일 텐데 reinforcement learning에서는 이런 식으로 정의가 되었다고 이해할 필요가 있습니다.

그러면 Prediction과 policy의 예제를 한번 보여드리도록 하겠습니다. 여기서의 예제는 에이전트가 이 Grid 상 어딘가에 위치할 수 있습니다.

어딘가 위치할 수 있는데 만약에 에이전트가 A라는 위치에 도달했다고 해보십시오. A라는 위치에 도달하면 reward로 10을 받고, A`이라는 위치로 이동하게 됩니다.

그리고 만약에 에이전트가 B라는 위치에 있다고 했을 때는 reward를 5 받고요. B`이라는 데로 이동을 하게 됩니다.

예를 들어서 policy를 uniform random policy로 정했습니다.

uniform random은 제가 어떤 state든지 간에 북으로 갈 확률 그리고 남으로 갈 확률 그리고 동, 서로 갈 확률이 다 0.25로 동일하다는 것입니다.

그러니까 간단히 얘기하면 내가 어떤 state든지 간에 무작위로 위아래 왼쪽, 오른쪽을 간다는 것입니다.

그런 policy를 따라서 제가 따라간다고 했을 때 그때의 value function은 오른쪽과 같이 구할 수 있습니다. 이거는 어떻게 구하는지 다음 주에 배울 예정이고요.

현재로서는 그냥 policy가 문제가 주어지고 policy가 주어지면 이런 value function을 구할 수 있다는 것을 이해하시면 되겠습니다.

그래서 여기서 보시면 value function이 가장 높은 state가 여기 8,8입니다. 왜냐하면 내가 이쪽으로 가면 reward를 10으로 받고 A`으로 이동을 하게 되죠.

그래서 이 state, A라고 마크된 state로 가는 value function이 제일 높습니다. 그런데 8,8이라는 숫자가 어떻게 나왔냐? 이거는 다음 시간에 배우도록 하겠습니다.

그리고 이거는 policy, 이 슬라이드에 대해서는 policy에 대한 예제입니다. 그래서 여기 \*라고 쓰여 있는 거는 optimal이라는 뜻입니다.

그래서 수많은 value function 중에서 optimal, 최적의 value function을 V\*라고 하고요.

π\*라는 건 뭐냐면 내가 취할 수 있는 수많은 policy 중에서 reward를 가장 maximizing하는 policy를 나타냅니다.

그러니까 여기 policy대로 따르자면 제가 만약에 이 state에 있다고 하면 오른쪽으로 가는 policy를 취하는 게 최적이라는 거죠. 왜냐?

이 state로 가면 +10이라는 reward를 받으면서 이 state로 이동을 하게 되죠. 그리고 내가 만약에 이 state에 있다고 하면 이렇게 가도 좋고 이렇게 가도 좋다는 겁니다.

그래서 만약에 내가 이쪽으로 이동했다고 하면 여기서는 반드시 북쪽으로 가는 게 좋고요. 그래서 오른쪽 그림은 policy에 대한 예제를 보여주고 있다고 이해하시면 되겠습니다.

그래서 마찬가지로 이런 optimal policy는 어떻게 구하느냐? 이거는 마찬가지로 다음 시간에 다룰 내용이고요.

일단 여기서는 이해하셔야 할 게 강화 학습에서 Prediction이 무엇을 의미하는지, Control이 무엇을 의미하는지 그거에 대한 각각의 예제를 보여드렸다고 이해하시면 되겠습니다.

그럼 지금까지 말씀드린 강화 학습의 기초 정의에 대해서는 여기까지 말씀드리고 다음 시간에 이어서 하도록 하겠습니다.

4.2.

이제 강화 학습을 모델링하는 데에 있어서 가장 기본적인 모델인 마르코프 과정에 대해서 말씀드리도록 하겠습니다.

기본적으로 강화 학습은 Markov Decision Processes라고 마르코프 결정 과정이라는 모델을 기반으로 formulation이 됩니다.

그러니까 Markov Decision Processes까지 이해하는 데 있어서 가장 핵심이 되는, 그런 기본이 되는 마르코프 과정에 대해서 함께 알아보도록 하겠습니다.

아주 간단한 경우를 한번 보죠. 그러니까 하나의 결정을 하는 간단한 모델입니다.

제가 만약에 0이라는 state에 있고요. 0이라는 state에서 A랑 B라는 여러 개의 액션이 있습니다. 여기서는 2개죠. 2개의 액션이 있고 A라는 액션을 취하면 1이라는 state로 가고요.

B라는 액션으로 취하면 2라는 state로 갑니다. 그래서 reward는 각각 1이라는 reward와 2라는 reward를 받는 것이고요.

그래서 강화 학습의 목적인 ‘내가 이 reward를 maximizing, 최대화하는 선택을 policy, 선택을 찾겠다.’

이 경우라고 하면 그냥 0이라는 state에 있을 때 제가 B라는 액션을 취하면 된다. 이렇게 쉽게 이해할 수 있겠죠.

하지만 많은 경우에 이렇게 하나의 선택으로만 우리의 태스크가 모델링되는 경우는 없을 것입니다.

그래서 실제로는 아주 복잡한 여러 개의 state가 있고 또 이 여러 개 state 사이에서 내가 어떤 액션을 취하느냐에 따라서 어느 state로 갈지가 결정되고 또 reward도 어떻게 받고,

이런 수많은 훨씬 복잡한 과정을 겪게 되는데요. 이걸 전체적으로 포괄하는 모델을 Markov Decision Processes라고 합니다.

그래서 마르코프 결정 과정 이렇게 한국말로 말할 수 있고요. 그래서 MDP의 구체적인 정의는 다음 시간에 할 예정이고요.

MDP의 정의하는 데 있어서 가장 기본이 되는 마르코프 프로세스부터 하나씩 밟아 나가도록 하겠습니다.

가장 간단한 모델입니다. 마르코프 프로세스부터 시작하도록 하죠. 마르코프 프로세스는 크게 2가지 구성요소로 되어 있습니다.

2가지 구성요소로 되어 있고 S는 여기서 보시다시피 state의 집합입니다. state의 집합이고요. P라고 되어 있는 거는 저희가 이전 시간에 model이라고 얘기한 것 중의 하나였죠.

P는 뭐였냐 하면 state transition function입니다. P라는 함수는 S를 정의역으로 가지고 있고 S를 공역 혹은 치역으로 가지고 있습니다.

그러니까 state를 받아서 또 다른 state로 맵핑해주는 함수라고 이해하시면 되겠고요. 많은 경우에 이거는 확률분포 형태로 주어지게 됩니다.

그래서 내가 현재 state에서 다음 state로 갈 수 있는 확률을 P라는 매트릭스로 표현할 수 있겠죠. 그래서 만약에 우리가 n개의 state가 있다고 가정을 해보죠.

우리가 총 가능한 state가 n개다. 그러면 P라는 transition function은 n×n 매트릭스, n×n의 행렬로 표현할 수 있습니다.

그런데 이 위에서 보시면 memoryless random process라는 정의가 있습니다.

memoryless라는 거는 뒤에 나오겠지만 히스토리, 그러니까 히스토리가 현재 state 하나로 다 설명이 된다는 뜻이고요.

그러니까 현재 state만 알면 그 이전의 히스토리는 몰라도 된다. 그런 뜻입니다.

그래서 그런 의미로 memoryless라는 단어가 붙었고요. random process라는 거는 지금도 말씀드렸다시피 우리의 state transition이 어떤 확률분포를 따라서 가게 된다.

그거를 말씀드린다고 보시면 되겠습니다. 그래서 마르코프 프로세스의 정의를 보시면 두 번째 줄을 보시면 되겠죠. 그래서 random state의 연속이고요.

대신에 중요한 건 Markov property를 가정하고 state transition을 진행한다고 보시면 되겠습니다. 그래서 과연 Markov property가 뭐냐? 그거에 대한 정의가 바로 밑에 주어져 있는데요.

state라는 건 그 히스토리로부터 주어진 모든 정보를 다 가지고 있다. 이런 뜻입니다. 그래서 확률분포 수식으로 나타내면 이와 같이 표현할 수 있는데요.

그러니까 현재 state가 주어졌을 때 그다음 state, 어디로 갈지에 대한 확률분포라는 것과 기존에 내가 어느 state를 방문했는지 모든 히스토리가 주어졌을 때

다음 state에 어디로 갈지의 확률이 같다는 것입니다.

그러니까 현재 state까지 오는데 어떤 히스토리를 겪었든지 간에 현재 state만 알면 다음 state가 어디로 갈지에 대한 확률분포가 정해진다는 것입니다.

또 다른 얘기로 말씀을 드리면 현재 state가 주어지면 이 현재 state의 미래 일과 과거의 일이 서로 독립이다. 이렇게 이해할 수도 있습니다.

그래서 이거는 예제입니다. 하나의 예제이고요. 여기서 보시면 동그라미로 표현된 게 state이고요. 네모로 표현되어 있는 state가 있는데 이거는 터미널, 종료 state입니다.

그러니까 여기에 도달하는 순간 더 이상 갈 수 있는 곳이 없죠. 그래서 하나의 예제를 들어보시면 제가 만약에 C1이라는 state에 있다고 하면

0.5의 확률로 C2라는 state로 가거나 혹은 0.5의 확률로 FB이라는 state로 가고요.

FB state에 있었으면 0.9의 확률로 다시 FB을 하거나 혹은 0.1의 확률로 C1으로 다시 돌아가거나, 이와 같은 구조로 되어 있다고 보시면 되겠습니다.

그래서 이와 같은 모델이 주어졌을 때 state transition function은 밑에 주어진 대로 P라고 주어질 수 있겠죠. 그래서 아까 말씀드렸다시피 P는 n×n 행렬로 표현됐다고 말씀을 드렸잖아요.

그래서 이 그림을 보시면 지금 state가 C1부터 C3, Pass, Pub, FB, Sleep 이런 총 7개의 state로 구성되어 있고요.

그래서 그 7개의 state에 대한 state transition function은 7×7 행렬로 주어지게 되고요.

일단 이 state transition function을 구체적으로 보면 내가 현재 C1이라는 state에 있다고 하면 C2로 transition할 확률이 0.5이고요. FB으로 transition할 확률이 0.5입니다.

여기 빈칸으로 되어 있는 부분은 그냥 다 0이라고 보시면 대요. 그래서 C1에서 C1로 가는 확률은 0이다. 이런 식으로 볼 수 있고요.

C2도 마찬가지로 내가 C2에 있었을 때 C3로 갈 확률이 0.8이고요. Sleep이라는 state로 갈 확률은 0.2이다.

여기서 재미있는 특징 중의 하나는 각 행에 대해서 합을 구해보면 다 1이라는 걸 알 수 있습니다.

왜냐하면 내가 한 state에서 모든 state에 대해서 어떻게 갈지에 대한 확률분포의 합은 1이 되어야겠죠.

여기 위에 보시면 에피소드라는 얘기가 있습니다.

에피소드는 뭐냐면 제가 이니셜 state로부터 시작해서 주어진 state transition 함수를 이용해서 하나의 경험을 샘플링 하는 거다. 이렇게 보시면 됩니다.

그러니까 제가 C1부터 시작해서 0.5의 확률로 FB, 혹은 C2로 가는데 제가 동전 던지기 같은 걸 해서 C2로 갔고요. 그다음에 C3로 갔고 Pass하고 Sleep으로 가겠죠.

그래서 결국에는 시작 state는 C1이고, 종료 state는 항상 Sleep입니다. Sleep인데 제가 주어진 P라는 확률분포에 따라서 random 하게 한번 가보는 것이죠.

가봐서 그 하나하나를 에피소드라고 합니다. 그러니까 쉽게 얘기하면 여러분한테 P가 주어진다고 하면 여러분이 예를 들어서 100개의 에피소드를 샘플링할 수 있는 겁니다.

그러니까 주어진 확률분포대로 이렇게 가보고 저렇게 가보고 해서 터미널 로드까지 갈 때 그 전 과정 하나를 에피소드라고 하는 것이죠.

그래서 실제로 강화 학습에서 게임을 하는 인공지능을 만든다고 할 때도 실제로 게임을 여러 판을 하게 되죠. 그 여러 판이라는 게 결국 하나하나가 에피소드가 될 거고요.

그 에피소드로부터 P를 예측할 수도 있고 이런 policy도 구할 수 있고 그런 식으로 된다고 보시면 되겠습니다. 그래서 P가 주어졌을 때 에피소드를 샘플링할 수도 있고요.

혹은 여러분의 에피소드를 아주 많이 얻은 다음에, 시뮬레이션을 통해 얻은 다음에 그것으로부터 P를 예측할 수도 있습니다.

그러면 지금까지 Markov process에 대해서 얘기했고요. 그거보다 한 단계 더 나간 Markov reward process라는 것에 대해서 말씀을 드리도록 하겠습니다.

여기서 보시면 빨간색으로 표시된 컨퍼런트들이 있는데 빨간색으로 표시된 건 지금까지 말씀드린 Markov process에서 추가된 컨퍼런트를 보여드리고 있습니다.

그래서 state S이랑 transition function은 Markov process와 동일하고요. 그 외에 추가적으로 reward function과 discount Factor라는 게 주어집니다.

discount Factor는 하나의 스케일러 값이고 이거는 0부터 1 사이의 값을 가집니다. 그리고 reward 함수는 뭐냐면 정의역은 state이고 공역, 치역은 실수입니다.

그러니까 어떤 state가 주어졌을 때 그 state의 어떤 reward 값을 맵핑해주는 함수라고 보시면 되겠습니다.

그래서 구체적인 정의가 오른쪽에 나타나 있는데, 내가 S라는 state에 있을 때 어떤 reward를 받을 기댓값을 이런 식으로 표현할 수 있습니다.

그래서 좀 전에 Student MRP, 그러니까 Markov process Example을 보여드렸는데 그거를 MRP Example로 확장한 것입니다. 그러니까 이거 하나는 MRP의 예제라고 보시면 되겠고요.

그래서 여기서 보셨다시피 우리가 새로 추가되어야 할 거는 R과 γ입니다.

그래서 R이라는 건 state 각각에 대해서 어떤 실수 값을 주어야 하기 때문에 이렇게 빨간색으로 보셨다시피 각 state마다 reward 값을 assign을 한 것이죠.

그리고 엄밀하게 여기 discount Factor에 대해서 주어지지는 않았지만 예를 들어서 discount Factor γ를 1로 세팅했다.

그런데 이 Example 자체가 하나의 MRP Example이라고 이해하시면 되겠습니다.

이제 Markov process에서는 state와 state transition밖에 없었는데 여기 Markov reward process에서는 reward라는 새로운 개념이 추가가 됐죠.

그렇기 때문에 우리가 return이라는 개념과 value function이라는 개념을 말씀드릴 수 있습니다.

그래서 return이라는 건 뭐냐면 total discount reward from time step t입니다. 그래서 현재 time step t로부터 해서 제가 미래에 얻을 수 있는 reward를 다 합한 것입니다.

중요한 거는 discount가 된다는 것이죠. 그러니까 한 step 미래의 reward에 대해서는 γ만큼, 두 step 후의 reward에 대해서는 γ²만큼,

그러니까 그만큼 discount가 된 reward를 정의하고 그 모든 걸 다 더한 값이 return이라고 보시면 되겠습니다.

그래서 immediate reward, 그러니까 당장 얻을 수 있는 reward가 나중에 delayed reward보다는 훨씬 더 높은 가치를 줘야 한다는 거고요.

그래서 여러분이 만약에 γ를 0으로 줬다는 건 여러분이 근시안적인 모델을 만들었다는 거죠. 이렇게 학습을 하다 보면 이 에이전트는 지금 당장의 reward만 따라가도록 학습이 될 거고요.

만약에 γ를 1로 줬다. 이거는 지금 당장의 reward랑 아주 먼 미래의 reward랑 똑같이 고려하겠다는 거기 때문에 훨씬 멀리 보는 에이전트를 학습할 수 있게 되겠죠.

그래서 이 γ에 대한 얘기는 다음 슬라이드에서 다시 한 번 말씀드리도록 하겠습니다.

그다음에 reward라는 개념이 새로 들어왔기 때문에 이제 value function을 정의할 수 있습니다. value function은 어떻게 정의하느냐면 현재 state의 value는 어떻게 정의하느냐?

내가 만약에 t라는 step에 s라는 state에 있을 때 내가 return을 Gt라고 했을 때 얻을 수 있는 return의 expectation 값이다. 이렇게 엄밀하게 정의하게 됩니다.

그래서 여기 보시다시피 expected return starting from state s 이런 식으로 정의할 수 있습니다.

그러면 discount Factor에 대해서 자세히 말씀드리도록 하겠습니다. 여러분이 ‘이게 왜 필요하냐?’ 이렇게 생각할 수 있는데요. 한 가지 예제와 함께 왜 discount가 필요한지 말씀드리도록 하겠습니다.

지금 여기 하나의 Markov reward process의 예제가 있습니다. 여기서 중요한 점은 뭐냐면 state1과 state2를 보면 자기 자신으로 돌아오면서 reward를 1로 받는 그런 edge가 존재한다는 것이죠.

그래서 만약에 여러분이 S1, 1이라는 state에 있을 때의 value를 구해보면 이게 무한대가 되는 문제가 있습니다.

왜냐하면 내가 S1에 있을 때 A를 취하고, A를 취하고, A를 취하면 계속 state1에 머무르면서 계속 reward를 취할 수 있겠죠.

그래서 discount Factor가 없다면 이런 infinity한, 그러니까 value function의 값이 무한대가 되는 이런 안 좋은 상황이 벌어지게 됩니다.

여기 Q라는 value function이 나와 있는데요. 현재로는 Q라는 value function이 뭔지에 대해서는 이해하실 필요가 없고요.

이게 나중에 배울 value function을 구하는 equation인데, 우리가 나중에 배울 value function equation을 적용해보면 S1에서의 value가 무한대가 된다.

일단 현재로서는 그렇게만 이해하시면 되겠습니다.

그래서 만약에 여러분의 모델에서 cycle edge가 존재하고요. 이 cycle edge의 reward가 0이 아니라고 하면 무한대가 되는 안 좋은 현상이 일어납니다.

그래서 만약에 discount Factor가 없는 경우에 이런 상황이 벌어진다는 것이고요.

그런데 만약에 discount Factor가 있다. 그렇다고 하면 나중에 설명을 하겠지만 S1의 value function 값이 1+γ, Q가 되고

이거는 다시 1+γ1+γ²Q, 그리고 이거는 다시 1+γ+γ²+γ³Q 이런 식으로 표현이 됩니다.

그래서 γ가 만약에 1보다 작은 수가 된다면 이게 무한한 텀을 가진 summation이라고 하더라도 특정 값에 bound가 되죠. 그러니까 극한 값을 취하면 특정 값으로 수렴하게 됩니다.

그래서 이런 무한대가 되는 걸 막을 수가 있다는 장점이 있고요. 그리고 이게 상식적, 물리적으로 생각해도 discount Factor는 매우 타당합니다.

여러 가지 방식으로 해석할 수 있겠죠. 그러니까 수학적으로도 γ가 있으면 안정화되는 효과뿐만 아니라 여러 가지로 해석할 수 있는데

이거는 한마디로 보면 시간이 지나면 지날수록 에이전트가 살아있을 확률이 점점 줄어들 텐데 그거에 대한 표현일 수도 있고요.

그다음에 내가 현재 봤을 때 내가 두 step 후에 얼마만큼의 reward를 받을 수 있을 거라고 생각하지만 세상에는 불확실성이 존재하고 그만큼 discount를 해줘야 한다. 그렇게 말씀드릴 수 있고요.

또 하나는 당연하게도 당장의 reward는 아주 먼 미래의 reward보다는 더 가치 있게 평가를 받아야 하고요.

그리고 마지막으로 수학적으로도 γ가 있느냐, 없느냐에 따라서 무한대가 발생할 수 있느냐, 없느냐? 이런 여러 가지 경우가 가능하다는 것입니다.

그래서 이거는 MRP에 대한 예제를 value function을 어떻게 구하는지 예제를 들어본 것입니다.

그래서 여기서는 저희가 starting 포인트로 C1. starting state는 항상 C1이고, discount Factor는 0.5입니다.

그래서 value function을 어떻게 구하는지의 하나의 예를 보여준 건데 아까도 말씀드렸다시피 우리가 random process가 하나 주어지면 여러 개의 에피소드를 샘플링할 수 있습니다.

예를 들어서 우리가 100개의 에피소드를 샘플링했다고 해보죠. 그러면 첫 번째 걸 보면 C1에 있을 때 reward가 -2였죠. 그다음에 C2일 때 reward가 -2입니다.

-2인데 discount를 해줘야 해서 2분의 1을 곱해야 되겠죠. 그다음에 C3로 갔으면 다시 마이너스입니다. 그런데 γ가 한 번 더 곱해지니까 4분의 1이 되죠.

그런 식으로 하다 보면 첫 번째 에피소드에 대해서 C1이라는 state에 있었을 때의 값은 -2.25가 됩니다.

그래서 이런 간단한 산술을 여러분이 모든 에피소드에 대해서 다 구할 수 있고요. 아까도 말씀드렸다시피 G1,

그러니까 state1에서의 return이라는 건 다음 step의 reward로부터 해서 계속 γ만큼 discount하면서 끝까지 갔을 때의 reward의 총 합이라고 말씀드렸고,

value function은 그거에 대한 expectation이라고 말씀드렸죠. 그래서 이런 에피소드를 만약에 여러분이 100개를 샘플링했다고 하고

그 에피소드 100개에 대해서 V1값을 다 구한 다음에 걔들의 평균을 내주면 V of C1, 그러니까 C1일 때의 value function값을 구할 수 있습니다.

그래서 오늘은 Markov process와 Markov reward process의 개념과 예제를 말씀드렸고요.

그다음에 Markov reward process에서 중요한 정의인 return, G라고 표현을 하고요. 그다음에 value function, V를 소개시켜드렸습니다.

그러면 다음 시간에는 우리가 정작 배워야 할 Markov Decision Processes와 거기서부터 어떻게 Prediction 혹은 Control을 수행하는지에 대해서 간단히 배워보도록 하겠습니다.

그러면 다음 시간에 다시 뵙도록 하겠습니다.

5.1.

안녕하세요? 서울대학교 컴퓨터공학부의 김건희입니다.

인공지능의 기초, 오늘은 마르코프 결정 과정과 다이내믹 프로그램을 이용해서 결정 과정을 푸는 방식에 대해 배워보도록 하겠습니다.

지난 시간에 저희가 강화 학습의 기초를 배웠고요.

그다음에 강화 학습을 모델링하기 위한 가장 기본적인 마르코프 결정 과정을 배우기 위해서 마르코프 과정이랑 마르코프 리워드 과정에 대해서 저희가 학습을 하였습니다.

지난 시간에 배운 내용을 바탕으로 해서 거기에서도 한 단계 나아가서 우리가 정작 배우고자 하는 마르코프 결정 과정에 대해서 오늘 다루도록 하겠습니다.

그리고 지난 시간에도 잠깐 말씀드렸다시피 reinforcement learning, 그러니까 강화 학습이라는 부분이 인공지능에서 좀 어려운 부분이기도 하고

그리고 많은 경우에 대학원에서 다뤄지기 때문에 쉽지 않게 느껴질 수도 있으실 것 같습니다.

그리고 확률통계에 대한 기본적인 지식을 필요로 하기 때문에 좀 어렵게 느껴지실 수 있는데요.

일단 지난 시간과 이번 시간에 배운 걸 기초로 해서 좀 더 공부를 해나가시고 그리고 지난 시간에 추천해드린 강의나 책들을 함께 보신다면 더 나은 이해를 얻으실 수 있지 않을까 생각합니다.

그러면 오늘 강의에 대해서 구체적으로 나아가 보도록 하겠습니다.

그래서 지난 시간에는 마르코프 프로세스에 대해서 일단 배웠고요. 그다음에 마르코프 리워드 프로세스에 대해서 배웠습니다.

그래서 그 두 개를 바탕으로 오늘은 Markov Decision Process를 소개시켜드리도록 하겠습니다.

지난번과 마찬가지로 여기에 빨간색으로 마크 되어 있는 부분은 방금 전, 지난 시간에 배운 마르코프 리워드 프로세스에서 새롭게 추가된 컴포넌트를 보여드리고 있습니다.

기본적으로는 다 똑같은데 action set이라고 할 수 있는 A가 상대적으로 추가됐다고 이해하시면 되겠습니다.

A라는 건 집합이고요. 에이전트가 취할 수 있는 action들의 집합이라고 이해하시면 되겠습니다.

그런데 action set이 새롭게 들어오면서 기존에 저희가 가지고 있던 Transition function과 Reward function도 변화를 해야 될 필요가 있습니다.

왜냐하면, 기존에는 Transition function의 경우에 state에서 어떤 state로 가는 그런 확률분포를 표현했다고 하면, 지금은 어떤 state에서 어떤 action을 취했을 때 어떤 state로 가는.

그게 state transition으로 정의가 되기 때문에 정의역이 조금씩 바뀌었다고 보시면 되겠죠. 정의역에 지금 action set에 대한 곱집합으로 표현이 됐다.

그러니까 state function을 P를 예로 들면, 어떤 state에서, 그러니까 어떤 state라는 건 집합 S의 한 원소고요.

A라는 것도 마찬가지로 집합 A의 한 원소. 그게 주어졌을 때 어떤 state로 갈지에 대한 확률분포다.

그래서 t step의 S라는 state에 있고, 내가 그 step에서 A라는 action을 취했을 때 다음 step에 S‘으로 갈 확률을 P가 나타내게 되는 것이죠.

Reward도 마찬가지로 A라는 term이 더 추가되는 겁니다.

왜냐하면 어떤 state에서 어떤 Reward를 바로 받는 게 아니라, 어떤 state에서 어떤 action을 취했을 때 그다음 step에서 어떤 Reward를 받느냐. 그런 식으로 확장이 된다고 보시면 되겠습니다.

그래서 마르코프 리워드 프로세스랑 매우 비슷한데 A가 추가됐고, A가 추가됐음으로써 P와 R이 조금씩 바뀌었다고 이해하시면 되겠습니다.

그럼 이제 action set이라는 것을 새롭게 정의했기 때문에 이제야 policy라는 걸 좀 더 깊이 정의할 수 있게 되었습니다.

그러니까 마르코프 리워드 프로세스에서는 action set을 아직 정의하지 않았기 때문에 policy는 정의하지 않고 value function만 정의를 했는데,

여기서는 policy까지 정의가 가능해지게 된 것입니다.

그래서 introduction에서 말씀드렸다시피 policy의 정의는 이와 같습니다. 그래서 어떤 state에서 어떤 action을 취할 확률분포를 policy로 정의하게 됩니다.

그래서 이 정의에서 보시다시피 policy라는 건 history에 의존하는 게 아니라, 현재 state에만 의존합니다.

왜냐하면, 기본적으로 이건 마르코프 프로세스의 일종이기 때문에 마르코프 assumption을 계속 사용하게 됩니다.

그러니까 현재 state만 알면 그 history는 더 이상 알 필요가 없다. 그 바탕으로 이제 정의가 됐다고 보시면 되겠고요.

그다음에 여기서 일단 가정하는 건 policy가 stationary, 그러니까 시간에 따라서 변화하는지 않는다고 보시면 되겠습니다.

내가 10 step의 state에 갔을 때랑 100 step의 state에 갔을 때 만약에 걔가 똑같은 state라고 한다면, 걔에 대한 action의 확률분포는 동일하게 주어졌다고 이해하시면 되겠습니다.

그러면 이제 policy라는 정의를 새롭게 말씀드렸고요.

value function은 마르코프 리워드 프로세스에서 이미 얘기를 했는데, 여기에서는 value function을 좀 더 한 단계 업그레이드해야 될 필요가 있습니다.

여기서 변화된 부분은 마찬가지로 빨간색으로 표현이 됐는데, 가장 중요한 변화는 policy가 아래첨자로 추가됐다는 것입니다.

그러니까 제가 소개 때 말씀드렸다시피 value function이라는 건 policy가 주어졌을 때 결정이 되는 것이고요. policy가 바뀌면 value function도 바뀝니다.

그래서 어떤 policy 하에서 value function가 정의가 됐는지 나타내기 위해서 아래첨자로 π를 썼고요.

그러니까 마르코프 리워드 프로세스에서는 아래첨자 π가 없었는데, 그 이유는 마르코프 리워드 프로세스에서는 action set이 없었고요.

따라서 policy에 대해서는 아직 정의를 안 했기 때문에 V를 그냥 V(s) 이런 식으로 표현을 했죠. 그러나 value function이 이런 식으로 바뀌었다는 게 하나의 중요한 변화고요.

또 하나의 변화는 또 다른 형태의 value function이 정의가 되었습니다. 또 다른 형태의 value function은 주로 Q로 표현을 합니다.

V는 s에 대한 함수입니다. s만 주어지면 V가 결정이 되는데, Q는 s와 A가 주어졌을 때의 value를 나타냅니다.

그래서 이 둘의 차이를 구별하기 위해서 V는 state value function이라고 하고요. 그리고 Q는 state action value function 혹은 action value function이라고 얘기를 합니다.

그래서 정의는 기본적으로 똑같습니다. value function도 state가 주어졌을 때 return에 대한 expectation이었는데요.

Q에 대해서 한 가지 차이가 있다면 S가 주어졌고, 그때 어떤 action을 취했을 때 expected return이다. 그렇게 보시면 되겠습니다.

그래서 taking action a, 이 부분이 하나 추가됐다고 이해하시면 되겠습니다.

그래서 우리가 정작 다루는 Markov Decision Process에서는 V랑 Q라는 value function을 동시에 사용하게 된다고 이해하도록 하겠습니다.

계속 우리가 하나의 example로부터 시작을 해서 하나하나씩 개선을 해나가면서 example을 보여드리고 있습니다.

현재 이 example에서는 우리가 policy를 Random policy 정의를 했습니다.

Random policy라는 건 뭐냐 하면, 어떤 state에 주어졌을 때 내가 취할 수 있는 action이 두 개라고 하면 이 두 개의 action은 0.5로 동일하다. 그렇게 보시면 되겠습니다.

여기서 지금 나타내는 건 Vπ(s)입니다. 그러니까 만약에 Random policy를 사용할 때 value function은 지금 모형에서 빨간색으로 이렇게 표현이 되어 있고요.

그리고 이 빨간색으로 주어진 값을 어떻게 구할지는 뒤에 다시 다루도록 하겠습니다.

그리고 이 모델에서 한 가지 바뀐 것, 지난 example에서 바뀐 것을 보시면, Pub라는 게 그 전 example에서는 하나의 state로 정의가 됐는데, 여기는 action으로 정의가 됐습니다.

그래서 예를 들어서 내가 만약에 이 state, 그러니까 C3라는 state에 있을 때 Pub라는 action을 취하면 1이라는 Reward를 받고 그다음에 0.4의 확률로는 C2로 가고요.

0.2의 확률로는 C1으로 가고, 0.4의 확률로 다시 C로 돌아온다. 이걸 나타내고 있습니다.

그래서 검은색 동그라미로 쓰여 있는 부분을 action이라고 보시면 됩니다. 어떤 action을 취했을 때, 내가 Pub라는 action을 취했을 때 3개의 state로 갈 확률이 이렇게 주어진 것이죠.

그래서 우리가 state transition probability를, 앞에서 예를 들어서 ss’a라고 rotation을 썼는데, s라는 state에서 A라는 s‘이라는 state로 갈 확률. 그런 식으로 이해하시면 되겠습니다.

여기서 C3에서 Pub라는 action을 취했을 때 주어진 확률로 C1, C2, C3로 옮길 수가 있다고 이해하시면 되겠습니다.

그러면 이제 MDP에서 가장 중요한 Equation입니다. Bellman Equation에 대해서 말씀을 드리도록 하겠습니다.

MDP에서 Bellman Equation은 크게 두 가지가 있습니다. 하나는 Bellman Expectation Equation이고, 하나는 Bellman Optimality Equation입니다.

그래서 저희 강의에서는 Bellman Expectation Equation에 대해서만 다루고요. Bellman Optimality Equation은 제가 소개해드린 책에서는 소개가 되어 있는데요.

그 개념은 여기서부터 상대적으로 쉽게 이해가 될 수 있기 때문에 본 강연에서는 Bellman Expectation Equation에 대해서 주로 다루도록 하겠습니다.

기본 아이디어는 뭐냐 하면, 우리가 정의한 value function에 대해서 현재 state의 value function과 그다음 state의 value function의 상관관계를 정의한 것입니다.

그래서 내가 어떤 state에서 Reward를 받고 다음 state로 갔을 때 현재 state에서의 value function과

Reward를 받고 다음 state로 갔을 때 그 다음 state에서의 value function의 관계를 배운다고 보시면 되겠습니다.

수식을 보시면 이 첫 번째 수식은 value function의 정의입니다. 이건 앞서 소개해드린 정의에서 가져온 거고요.

그다음 수식은 Gt, 그러니까 앞서 소개해드린 return의 정의대로 표현을 한 것입니다. 그래서 첫 두 수식은 앞서 설명 드린 수식으로부터 왔다고 보시면 되겠고요.

그다음에 이 γ가 있는 term들부터 해서 γ를 앞으로 빼고 다시 표현을 한 겁니다. 그러면 이 뒤에 건 Rt+2+γRt+3 이런 식으로 끝까지 가겠죠.

그러면 이 안은 뭡니까? 이 안은 우리가 기존에 얘기한 return의 정의랑 동일하죠. 그런데 단지 앞에서는 Gt였는데 여기서는 Gt+1으로 바뀌어야 되겠죠. 그러니까 한 step 이후니까.

그래서 결국에는 E에 Rt+1+γGt+1’|St=S 이런 식으로 표현이 됩니다.

그러면 이 부분을 보시죠. 이 뒷부분을 보시면 이건 다시 value function에 대한 정의입니다. 이 부분을 value function으로 표현을 하면 결국에는 이와 같이 표현을 할 수 있습니다.

그래서 여기서 중요한 룰을 사용한 거는 Expectation의 Expectation은 Expectation 한 번 취한 것과 같습니다.

그래서 실제로는 여기도 한 번 Expectation 취한 거랑 똑같기 때문에 그 상황에서 이 Expectation Gt+1을 Vπ(St+1)으로 바꿨다고 이해하시면 되겠습니다.

그래서 결과적으로 첫 번째 수식과 마지막 수식을 비교한다면, value function이라는 건 현재 step의 return의 Expected value인데,

그건 Immediate reward에 그다음 state의 value function을 γ만큼 discount한 것과의 합으로 이렇게 표현할 수가 있다. 그래서 이 첫 번째 설명이랑 동일하게 볼 수가 있겠죠.

value function은 두 부분으로 나눠지는데, Immediate reward, 그러니까 당장 받을 수 있는 Reward에 대한 Expectation

그리고 다음 state로 갔을 때의 value function에 대한 Expectation. 그 두 개로 표현이 된다고 이해하시면 되겠습니다.

그래서 앞서 말씀드린 state value function에 대한 Bellman Expectation Equation이라고 불리는 새로운 정의가 이렇게 주어졌고요.

이거와 똑같은 방식으로 해서 Q에 대해서도 마찬가지로 이와 같이 정의를 할 수 있습니다.

Q에 대해서 정의하는 것의 차이라고 한다면, Q는 주어지는 given part에 action이 같이 주어져야 된다는 거고요. 그 외에는 다 동일합니다.

Immediate reward+γ 그다음에 다음 state, 다음 action에서의 Q값을 더한 것과 같다고 할 수 있겠죠.

유도는 직접적으로 안 하지만 여러분이 똑같은 방식으로 유도를 할 수 있고요.

결국에 V는 state에 대한 함수이고, Q는 state와 action에 대한 함수이기 때문에

그 부분만 바꿔주시면 결국에는 state value function, action value function에 대한 Bellman Expectation Equation을 유도할 수 있다는 것입니다.

이 두 equation이 저희 MDP 공부하는 데 있어서 가장 중요한 공식이라고 이해하고 넘어가도록 하겠습니다.

그런데 중요한 점은 V와 Q와의 관계를 밝히고자 하는 것입니다. 일단 MDP에서는 두 가지 step으로 나뉩니다.

두 가지 step이라는 건 뭐냐 하면, 제가 만약에 어떤 state에 있을 때 내가 어떤 action을 취하면, 그 action에 따라서, 확률분포에 따라서 state가 결정되죠.

어떻게 보면 state에서 action으로 가는 한 가지 과정이 있고, action에서 다시 state로 가는 두 가지 과정이 있습니다.

그래서 이 두 가지 과정을 통해서 V와 Q와의 관계도 알고, 이제 어떻게 하면 이 equation을 V로만 표현하고 Q로만 표현할 수 있는지에 대해서 말씀을 드리도록 하겠습니다.

지금 value function에 대한 내용입니다. value function은 어떻게 정의가 될 수 있느냐? 이 policy가 주어집니다. policy는 뭐라고 말씀을 드렸죠?

policy라는 건 내가 S라는 state에 있을 때 A라는 action을 취할 확률이죠. 그러니까 상식적으로, 수학적으로 생각해보면 내가 S라는 state에 있을 때 value라는 건 어떻게 구하느냐?

어떻게 구하느냐 하면, 내가 S라는 state에서 취할 수 있는 action이 여러 개가 있어요.

그런데 이 action에 대한 확률이 있을 거고 이 action에 대한 확률이 있고 여러 가지 action에 대한 확률이 있습니다.

그러면 내가 만약에 이 확률대로 이 action을 취했을 때의 value, 이 확률대로 이 action을 취했을 때의 value.

그러니까 확률에 대해서 weighted sum을 해줘야겠죠. 그러면 현재 state의 value를 구할 수가 있겠죠.

그래서 이게 Expectation의 정의랑 연결이 돼 있습니다. Expectation이라는 건 뭐냐 하면, 가장 간단히 생각하면 평균이고요.

그런데 평균이 가능한 이유는 만약에 n개의 데이터가 있는데 얘네들이 다 똑같은 weight를 가지고 있다고 하면, 여러분이 Expectation 구한다고 하면 그냥 평균을 구하면 됩니다.

그런데 예를 들어서 내가 데이터마다 다른 확률분포를 줬다.

그런 경우에는 이 확률에 대해서 그 값이랑 곱하고, 이 확률에 대해서 그 값이랑 곱한 다음에 다 더해야지만 Expectation을 구할 수가 있습니다.

그럼 간단히 예제를 들어보면, 우리가 어떤 한 학급에서 시험을 봤다고 가정해보죠. 시험을 보면 만약에 한 강의실에 50명의 학생이 있다고 하면, 50명에 대한 성적 분포가 나올 것입니다.

distribution이 나올 텐데, 여러분이 그 학급이 얼마나 잘하는지 못하는지를 하나의 숫자로 표현해본다고 해보세요.

하나의 숫자로 표현하는 가장 좋은 방법이 뭘까요? 그건 50명 학생의 성적을 평균 내는 것이죠. 그래서 50명에 대한 성적 분포가 있는데, 그걸 하나의 숫자로 요약한 거고요.

그렇게 하나의 숫자로 요약을 하면 이 반과 저 반과 또 다른 반 사이에서 누가 더 잘했는지 하나의 숫자로 이제 비교가 되는 것이죠.

그런데 저희가 평균을 구한다는 건 똑같은 weight로 값을 구했다는 겁니다.

그런데 제가 만약에 너는 0.2의 weight를 주고 너는 0.1의 weight를 줬다고 하면,

그 weight값이랑 그 학생의 성적이랑 곱하고 또 이 weight랑 그 학생의 성적을 곱하는 식으로 다 더해야지만 Expectation을 구할 수가 있다는 거죠.

그러니까 우리가 일반적으로 평균이라고 하면 weight라는 걸 그냥 1/50으로 다 똑같이 줬다고 보시면 되고요.

만약에 학생들마다 서로 다른 weight를 주게 되면 이런 식으로 표현을 하게 됩니다.

V라는 state S에 대한 value function은 내가 그 state에서 취하게 될 모든 action에 대해서 확률분포를 다 구한 다음에 그 확률분포대로 Q값을 곱했다고 보면 됩니다.

그래서 이건 Expectation을 구했다고 보시면 되겠고요. 그래서 결국에는 state value function을 구하고자 하려면 Q값을 알아야 되는 것입니다.

만약에 우리가 각 state와 action 사이의 Q값을 알면, V값을 자동적으로 구할 수 있고요. 그럼 이제 Q값을 한번 어떻게 구할지 보죠.

현재 내가 어떤 state에서 어떤 action을 취했을 때 Q값이라는 건 어떻게 정의가 되느냐 하면, 이 그림을 자세히 이해해보면 검은색 동그라미로 표현된 부분이 action입니다.

그래서 실제로는 이 그림이 여기 이렇게 붙어 있는 거라고 보시면 되겠고요. 여기는 또 다른 게 붙어 있다고 보시면 되겠습니다.

그래서 현재 S와 A가 주어졌을 때의 value function을 어떻게 구하느냐 하면, 내가 state S에서 A를 취했을 때의 Reward 더하기, 그다음에 이건 무엇이죠?

이건 state transition probability죠? 그러니까 S라는 state에서 내가 A라는 action을 취했을 때 S‘으로 갈 확률입니다.

그래서 결국에 이 부분도 마찬가지입니다. 이 부분도 결국에는 Expectation 값을 구한 거예요.

그러니까 내가 S라는 state에서 A라는 action을 취했을 때 이 state로 갈 수도 있고, 이 state로 갈 수도 있고, 이 state로 갈 수도 있습니다.

그 state 각각은 Pss’이라는 값으로 확률분포가 주어졌고요. 여기로 갔을 때 이 state의 value값, 이 state의 value값, 이 state의 value의 확률대로 weighted sum을 한 Expectation이 된다.

그런데 이건 다음 step이기 때문에 γ만큼 discount를 해야 되고요.

이 Reward는 우리가 어떤 state에서 어떤 action을 취하고, 그 결과로 state를 transition하면서 R을 받기 때문에 이런 식으로 표현이 됩니다.

그러니까 어떻게 보면 우리가 state에서 action을 취해서 다른 state로 가는 과정을 두 step으로 나눈 것이고요.

그래서 V는 Q로 정의가 되고, Q는 다시 그 다음 step의 state의 V값으로 정의가 됩니다. 그래서 결국에는 이 두 가지 equation이 가장 핵심이 되는 equation이라고 보시면 되겠습니다.

그러면 여기서 한 가지 ‘마음에 안 든다.’ 이렇게 생각할 수 있는 게 뭐냐 하면, V를 구하려면 Q를 구해야 되잖아요. 그리고 Q를 구하려면 V를 구해야 된다.

그런데 많은 경우에 둘 다 계속 구하고 싶지 않은 경우도 많습니다. V만 가지고 있거나 혹은 Q만 가지고 싶은 경우가 있습니다.

그래서 결국에 이 두 equation을 합쳐서 V는 V에 대한 equation으로만 표현하고, Q는 Q에 대한 equation으로만 표현을 해보자. 그게 다음 슬라이드에 나온 내용입니다.

그래서 첫 번째 equation은 뭐냐 하면, 앞에 equation에서 원래 여기가 Q값이었죠? 그런데 그 Q값 대신에 그 앞에 equation에 Q에 대한 equation을 그냥 넣은 것입니다.

이것도 마찬가지로 여기가 실제로는 다음 state에 대한 value였는데, 그걸 이 equation대로 적용한 것이죠.

그런데 여기서 주의해야 될 점은 여기 현재 state s에 대한 게 아니라 다음 state s에 대한 equation으로 써야 된다.

지금 말씀드린 두 equation을 V를 기준으로 정리를 한 번 것, Q를 기준으로 정리를 한 번 한 것, 이렇게 두 개로 나눠져 있다고 보시면 되겠습니다.

이렇게 하나의 equation으로 보면 되게 길고 복잡해 보이실 수 있는데, 앞 슬라이드에서 말씀드린 대로 두 가지 step으로 나누면 상대적으로 쉽게 이해할 수 있습니다.

그러니까 처음에 V라는 현재 state의 value function 값은 검은색 동그라미에 붙어 있는 Q(s, a) 값으로 표현이 되고요.

그러니까 내가 현재 state에서 취할 수 있는 action이 여러 개가 있고, 각각에 대해서 Q값이 존재하고, 그 Q값에 대한 확률분포도 존재하죠. 그렇게 일단 표현을 할 수 있고요.

또 Q는 내가 어떤 state에서 어떤 action을 취하면 Reward를 받고 그다음에 P라는 state transition matrix에 의해서 다음 state로 갈 텐데, 그 과정을 앞 슬라이드에서 설명을 드린 거죠.

그래서 이 둘을 합치면 V에 대해서 정리한 것, Q에 대해서 정리한 것. 이 두 가지로 이해하실 수 있겠습니다.

그래서 결국에 저희가 실제로 쓰게 될 equation은 여러분이 만약 V를 쓰게 된다면 첫 번째 equation을 주로 쓰고, Q를 쓴다고 하면 두 번째 equation을 주로 쓴다고 이해하시면 되겠습니다.

그래서 이건 Bellman Equation의 예제를 한번 보여드리는 것입니다. 어떻게 구할 수 있는지를 예제로 보여드리는 거고요.

앞서 말씀드렸다시피 현재 우리가 관심 있는 state는 이겁니다. 그러니까 이게 C3라는 state인데, 이 C3라는 state에서의 value function을 한번 구해보자.

그래서 equation은 Bellman Equation에 따라서 이와 같이 주어졌죠. 그다음에 중요한 것은 이 policy죠.

그런데 우리가 그 전 example에서 Random policy를 정의했습니다. Random policy라는 건 뭡니까?

내가 이 state에서 Study라는 action을 취할 확률이랑 Pub를 취할 확률이랑 똑같다는 것입니다.

그래서 얘는 0.5로 고정이 됩니다. 그러면 여기 보시다시피 내가 모든 가능한 A에 대해서 다 summation을 해야 됩니다.

그래서 먼저 Study라는 action을 취했을 때 한번 보죠. 그러면 내가 만약에 Study라는 action을 취했으면, 이 π값이 얼마가 되죠? 말씀드린 대로 0.5가 되고요. 뒤에 있습니다. 0.5가 되고요.

그다음에 Reward가 어떻게 되죠? Reward가 10입니다. 그래서 이 부분이 10이 되고요.

그다음에 내가 C3라는 state에서 Study라는 action을 취할 때 갈 수 있는 state라는 건 하나밖에 존재하지 않습니다. 이 Sleep이라는 state 하나밖에 존재하지 않죠.

그래서 하나에 대해서만 고르라고 하면 되는데, 그런데 Sleep이라는 state의 value값이 뭐가 됩니까? 0이 됩니다.

그래서 여러분이 Study를 처음 고려했을 때 이 equation을 따라가면 이게 0.5가 되고, 이게 10이 되고요. 그다음에 이 V가 0이 되기 때문에 결국에는 0.5×10이 된다고 이해하시면 되겠고요.

제가 이 Study부터 한 이유는 이 앞에 게 좀 복잡하기 때문에 간단한 것부터 말씀드린 것입니다.

그러면 또 다른 action인 Pub에 대해서 보도록 하죠. 여러분이 Pub라는 action을 취할 확률은 몇이죠? 0.5로 동일합니다. 그래서 이건 0.5가 되겠고요.

그다음에 내가 C3에서 Pub라는 action을 취했을 때 Immediate reward는 1입니다.

그다음에 이 부분을 구해야 되는데, 저희가 γ, discount factor을 계산을 쉽게 하기 위해서 일단 1이라고 가정을 했습니다. 그러면 얘는 1이니까 일단 고려를 안 해도 되겠고요.

그리고 이 상황에서 제가 C3에서 Pub를 한 다음에 갈 수 있는 state가 몇 개 있습니까? 지금 3개 있죠. 그래서 이 S라는 게 C1, C2, C3입니다. 그래서 C1, C2, C3에 대해서 다 계산을 해야 돼요.

그래서 C1일 확률이 뭡니까? C1일 확률은 얘가 0.2죠. 그때 0.2라는 확률로 갔을 때의 value function은 뭡니까? -1.4입니다. 그래서 0.2×(-1.3)이 있고요.

그다음에 C2. C2의 확률은 0.4입니다. 그걸 따라갔을 때 value는 2.7이었죠. 그래서 0.4×2.7.

그다음에 마지막으로 C3죠. C3일 때는 0.4의 확률이었고, C3의 value값은 7.4였습니다.

그래서 총 현재 C3라는 state에서의 value값은 이제 주어진 equation에 따라서 이와 같이 주어집니다. 그래서 이걸 여러분이 계산해보면 7.4라는 값이 나오게 됩니다.

그래서 지금까지 말씀드린 내용은 저희가 마르코프 프로세스부터 시작해서 정작 알아야 될 Markov Decision Process까지 정의를 다 말씀드렸고요.

그때의 value function 정의, V랑 Q가 있고 그다음에 policy에 의한 정의를 말씀드렸고요.

그리고 그게 주어진 상태에서 MDP에서 가장 중요한 equation이라고 말씀을 드린 Bellman Expectation Equation을 소개시켜드렸고요.

Bellman Expectation Equation을 정의하려면, V를 정의하려면 Q가 필요하고요. Q를 정의하려면 V가 필요했고요.

그래서 여러분이 이 두 수식을 이용해서 V로만 표현하고 싶다, Q로만 표현하고 싶다. 이렇게 양방향으로 할 수가 있고요.

그다음에 지금 방금 value function을 어떻게 구할지에 대한 예제를 소개시켜드렸습니다.

지금까지는 어떻게 보면 여러분이 이 예제에서는 value값이 이렇게 주어져 있기 때문에 한 가지 step에 대해서 example로 계산을 해볼 수가 있는데,

이런 게 전혀 주어지지 않은 상태에서 처음부터 어떻게 계산하는지 그 과정은 다음 시간에 다뤄보도록 하겠습니다.

그래서 현재 예제 같은 경우에는 V가 이렇게 주어진 상태에서 한 step에 대한 value function 값을 구하는 걸 예제로 보여드렸고요.

처음부터 value값이 state마다 주어지지 않았을 때 어떻게 구할 수 있는지를 전체 equation를 통해서 표현해볼 수 있습니다.

우리가 Bellman Equation을 좀 전 예제에서는 하나의 state에 대해서 따로 따로 표현했다고 하면, 만약 n개의 state가 있다고 하면 V를 n×1 벡터로 표현할 수 있고요.

Reward도 마찬가지로 n×1 벡터로 표현할 수 있고, P도 policy가 정해진 상태에서는 n×n matrix로 표현할 수 있고요.

이와 같은 식으로 지금 앞서 말씀드린 수식을 행렬 형태로 표현한다면 이와 같이 표현할 수 있습니다.

그래서 보시다시피 V라는, value function이라는 게 왼쪽에도 존재하고 오른쪽에도 존재합니다.

그래서 구하는 방식은 두 가지 방식이 있는데, 여러분이 value function을 하나 initialization을 한 다음에, R이라는 건 주어진 거고요. P라는 것도 주어진 겁니다.

그래서 MDP 정의를 보시면 R과 P가 주어져야 되는 값입니다. 그래서 P와 R이 주어졌다고 하면 Random initialization한 V로부터 새로운 V를 구할 수 있고요.

그 V를 다시 우항에 넣어서 γP를 곱하고 다시 R을 더해서 또 다른 V를 곱하고. 이 과정을 반복을 하면 여러분이 수렴된 V를 구할 수 있습니다. 그게 한 가지 방법이고요.

또 하나의 방법이라고 한다면 이와 같이 수식 형태로 표현이 됐기 때문에 여기서 이 부분을 좌항으로 넘깁니다.

그러면 실제로는 이게 matrix multiplication이기 때문에 이건 identity matrix, 그러니까 n×n에 identity matrix를 곱한 거로 표현할 수 있고요.

이걸 좌항으로 넘겼다고 하면 V에 대해서 이렇게 표현을 할 수 있고요. 그다음에 나누기. matrix에서 나누기 개념을 하는 게 역행렬이잖아요.

그래서 그 부분에 대해서 역행렬을 양쪽에 곱해주면 왼쪽에 있는 애는 사라지고 이제 오른쪽에 역행렬이 들어가게 됩니다.

그런데 이 방법은 추천해줄 수는 없는 방법입니다. 왜냐하면 matrix, 행렬이 n×n 행렬이라고 하면, 이 n×n 행렬의 역행렬을 구하는 복잡도가 n의 3승입니다.

그러니까 만약에 state가 10개 정도 있다고 하면 상관이 없는데, state가 100개, 1,000개 이런 식으로 급격히 많아진다고 했을 때 n의 3승의 복잡도라는 건 매우 오랜 시간이 필요한 복잡도입니다.

그래서 이렇게 역행렬을 구해서 풀지 말고 좀 더 효율적으로 풀 수 있는 방법이 여러 가지가 있는데, 하나는 Dynamic programming이고요. 이건 다음 시간을 통해서 저희가 배울 예정이고요.

그다음에 Monte-Carlo evaluation, Temporal-Difference learning라는 건 좀 더 advanced, 고급 테크닉인데요.

아쉽게도 본 강의에서는 다루지 않고요. 여러분이 관심 있다면 책이나 강화학습에 초점을 맞춘 강연을 통해서 확인해보실 수가 있겠습니다.

아주 중요한 개념으로서 Optimal Value Function에 대해서 소개시켜 드리고자 합니다.

좀 전에 말씀드렸다시피 value function을 구하기 위해서는 policy가 주어져 있어야 합니다. 그래서 저희가 value function 아래첨자로 π를 추가했었는데요.

그래서 Optimal Value Function이라는 건 뭐냐 하면, 주어진 문제에서 모든 가능한 policy를 다 고려했을 때 가장 최대가 되는 value function이 Optimal Value Function입니다.

그래서 Optimal이라는 뜻으로 별표를 아래첨자로 씁니다. 이건 policy마다 다른 value function이 존재하는데, policy들 중에서 최적의 policy가 되는 것이죠.

그러니까 value function을 최대화하는 policy가 주어졌을 때의 value function이 V\*(s)라고 주어지고요.

Q도 마찬가지로 똑같이 정의를 할 수 있습니다.

수많은 policy에 대해서 Q value function의 값이 다 달라질 텐데, 그 중에서 최적의 policy를 취했을 때 Q가 maximize되는 때의 Optimal action value function을 이와 같이 정의합니다.

한 가지 MDP에서 중요한 이론 중에 하나는 어떤 MDP라도 여러분의 MDP를 정의하면 항상 optimal policy라는 건 존재한다는 것이고요. 그래서 optimal policy는 항상 존재한다.

그리고 optimal policy를 따랐을 때의 value function이랑 value function의 최대, 그러니까 최적의 value function은 같다는 거고요.

그리고 Q에 대해서도 마찬가지입니다. 여러분이 optimal policy를 찾은 다음에 그때 Q를 구해도 되고요.

아니면 Q값들 중에 최대가 되는 최적의 Q를 구해도 됩니다. 그렇게 구한 다음에 optimal policy를 찾을 수도 있습니다.

그래서 이 수식의 의미는 지금 말씀드린 대로 optimal value function을 찾은 다음에 거기서부터 best policy, 최적의 policy를 찾아도 되고요.

혹은 최적의 policy를 찾아서 그거에 대한 value function을 구하면 역시 마찬가지로 V\*(s)를 구할 수 있고 얘네들은 항상 같다는 것이죠.

그래서 optimal policy를 찾는 과정은 실제로는 간단합니다. 간단한 이유가 optimal value function을 찾으면 optimal policy는 쉽게 찾을 수 있다는 것입니다.

그래서 많은 경우에 policy를 다루지 않고 그냥 value function만 다루는 경우가 있습니다.

value function만 다뤄서 최적의 value function을 찾으면 거기서부터 최적의 policy를 찾는 건 너무 쉬운 일이기 때문에.

만약에 최적의 value function이 주어졌을 때 어떻게 최적의 policy를 찾느냐? 그건 너무나도 간단한데, Q를 한번 예로 들어보죠.

최적의 Q라는 건 뭐냐 하면, state가 주어지고 action이 주어졌을 때의 value를 정의한 함수입니다.

그러니까 어떤 state가 있었을 때 그 state에서 취할 수 있는 action이 이렇게 여러 개 있다고 했을 때 그냥 Q\*(s, a).

첫 번째 action에 대한 Q값, 두 번째 action에 대한 Q값, 세 번째 action에 대한 Q값을 다 따져봐서 그 중에 최대가 되는 action을 취하면 최적의 policy가 되는 거죠.

그러니까 Q값을 구해놓으면 그 state에서 수많은 action에 대한 Q값들을 다 비교해서 그 Q값들 중에 가장 큰 Q값을 가지는 action을 항상 취하도록 하는 게 optimal policy이 된다는 겁니다.

그래서 많은 경우에 value function부터 구하고 그다음에 policy를 구하는 식으로 진행이 됩니다.

이건 optimal policy에 대한 예제입니다. 여기 보시면 빨간색으로 되어 있는 화살표가 optimal policy라는 얘기입니다.

예를 들면 제가 만약에 C3라는 state에 있어요. 그럴 경우에 제가 취할 수 있는 action은 Pub가 있고요. 제가 취할 수 있는 action은 Study가 있습니다.

그런데 제가 만약에 C3에 있을 때 Study를 취하면 이게 값이 주어진 대로 10이라고 한번 가정을 해보고요.

그리고 예를 들어서 C3에 있을 때 Pub라는 action을 취하면 여기 주어진 대로 8.4라는 값을 가지게 되죠.

그래서 이 C3라는 state에 있어서 optimal policy라는 건 C에 오면 Study를 취해라. Study라는 action을 해라. 그게 optimal policy입니다.

그래서 결국에는 각 state마다 여러 개의 화살표가 있지만, 빨간색으로 된 화살표는 그 state에 있으면 그 action을 취하는 게 optimal이다. 그런 의미로 이해하실 수 있겠습니다.

오늘은 MDP의 완벽한 정의에 대해서 말씀을 드렸고요. 그 MDP로 오면서 value function이 V랑 Q라는 애가 존재한다. 그리고 policy라는 걸 정의할 수도 있다. 그거에 대해서 말씀을 드렸고요.

그다음에 Bellman Expectation Equation이 MDP에서 제일 중요한 equation이고요. 그거에 대해서 소개시켜드렸고요.

그다음에는 optimal policy는 어떻게 구하느냐. optimal value function을 구하면 된다. optimal value function을 구하면 쉽게 optimal policy를 구할 수 있다.

그리고 예제는 이와 같다고 말씀을 드리면서 오늘 강의는 여기서 마치도록 하겠습니다. 그럼 다음 시간에 다시 뵙도록 하겠습니다.

5.2.

지난 강연에서는 저희가 Bellman Expectation Equation에 대해서 소개를 드렸고요. Bellman Expectation Equation이라는 건 결국에 V 혹은 Q라는 value function을 구하는 과정이었습니다.

그래서 V가 어떻게 표현이 되는지, Q가 어떻게 표현이 되는지를 말씀드렸고요. 그리고 V나 Q를 구하기 위해서 행렬의 역행렬을 곱하는 방법에 대해서 말씀드렸습니다.

그때 제가 그걸 좀 더 효율적으로 구할 수 있는 방법이 있고, 그때 Dynamic programming, 그러니까 동적 프로그래밍을 이용하는 방법이 있다고 말씀을 드렸는데요.

본 강의에서는 어떻게 그 value function을 동적 프로그램을 통해서 구하는지에 대해서 설명을 드리도록 하겠습니다.

Dynamic programming이라는 건 여러분이 만약에 알고리즘 수업을 들었다면 반드시 꼭 배워야 되는 부분 중에 하나고요.

본 강연에서는 여러분이 Dynamic programming에 대한 사전지식이 있다고 가정하고 말씀을 드리도록 하겠습니다.

Dynamic programming은 단순히 지금 말씀드리는 강화학습에서만 쓰이는 건 아니고 되게 다양한 분야에서 해를 구하는 알고리즘으로 사용이 되고요.

기본적으로 다음 두 가지 특성을 만족하는 경우에 Dynamic programming을 사용할 수 있습니다.

그러니까 주어진 문제가 있다고 하면 그 주어진 문제를 여러 개의 sub-problem, 더 작은 문제로 나눌 수가 있고요.

그리고 중요한 건 그 나눠진 작은 문제에 대해서 저희가 솔루션을 구하면, 그 솔루션이라는 게 원래 복잡한 문제를 푸는 솔루션을 구하는 데 사용이 된다는 것입니다.

그러니까 원래 문제를 계속 작은 문제로 나눠가고요. 그래서 충분히 작아진 문제에 대해서 솔루션을 구해서

그 솔루션을 활용해 그것보다 더 높은, 더 어려운 문제에 대한 솔루션을 구하고 구하는 이런 과정이 가능한 경우에 Dynamic programming을 적용하게 됩니다.

말씀드린 MDP에서는 이 두 가지 특성을 다 만족하기 때문에 Dynamic programming을 활용해서 Bellman Expectation Equation을 풀 수가 있다고 말씀을 드리겠습니다.

그리고 이건 예전에 말씀드린 것에 대한 리마인드입니다. 다시 한 번 말씀드리고자 하는 내용인데요.

Prediction이랑 Control에 대한 정의를 그때 말씀드리면서 강화학습에서 Prediction이라는 건 policy가 주어졌을 때 value function 구하는 거라고 말씀을 드렸고요.

그래서 좀 더 구체적으로 지금 MDP setting에서 말씀을 드리면, MDP가 주어지고 policy가 주어졌을 때 value function을 구하는 과정이 Prediction이 됩니다.

Control은 뭐냐? Control은 optimal policy를 구하는 과정이라고 했잖아요. 그래서 마찬가지로 이 MDP setting에서는 MDP가 주어졌을 때 optimal policy를 구하는 과정이라고 보시면 되겠습니다.

그런데 전 수업 때 잠깐 말씀드렸다시피 optimal policy이라는 건 optimal value function을 구하면 되게 쉽게 구할 수가 있습니다.

그래서 많은 경우에 Control 문제라고 하면 optimal policy를 구하는 문제이긴 한데, 그냥 optimal value function을 구하는 문제를 Control이라고 부르기도 합니다.

그리고 여기서 MDP가 주어졌다는 얘기가 있는데 그건 무슨 얘기냐 하면, S도 알고, A도 알고, P, R도 알고, γ도 알고 있다. 이게 다 주어졌다고 가정을 한 상태입니다.

그래서 이 5개의 component가 다 주어졌을 때 policy가 추가적으로 주어지면 value function을 구하는 문제. 그게 Prediction이고요.

MDP만 주어졌을 때 optimal policy를 구하는 과정이 Control이다.

그래서 Iterative Policy Evaluation을 통해서 Prediction을 해결한다. 이걸 다음 슬라이드부터 말씀을 드리도록 하겠습니다.

Iterative Policy Evaluation이라는 건 뭐냐 하면, policy가 주어졌을 때 이걸 evaluation 한다. 그러니까 전 슬라이드에서 말씀드렸다시피 Vπ를 구하는 과정입니다.

Vπ를 어떻게 구하느냐 하면, 다음과 같이 구합니다.

그래서 처음에 여러분이 V를 initialize한 다음에 그걸 계속 matrix multiplication으로 주어진 수식대로 반복을 하면, 결국에 V가 수렴이 되는데요.

그 V가 수렴이 된 결과를 policy에 대한 evaluation이 끝났다고 보실 수가 있습니다.

그러니까 예를 들어서 만약에 state가 10개다. 그러면 V를 1로만 이루어진 10×1 벡터로 정의를 합니다.

그리고 MDP가 주어졌다고 얘기했잖아요. MDP가 주어졌다는 얘기는 R이 주어졌고, P도 주어졌고, γ도 주어졌습니다.

그러니까 1, 1, 1, 1부터 시작해서 matrix multiplication을 하고, 그 각각의 요소에 γ를 곱하고, 그걸 Reward랑 더하면 다음 Iteration의 V값을 구하죠.

그 V값을 활용해서 우항에 다시 넣는 것입니다. 그래서 다시 γ 곱하고, P 곱하고, R 더하고.

이 과정을 계속 반복해서 이게 결국이 수렴이 되는데, 그 수렴된 값이 policy가 주어지고 MDP가 주어졌을 때 value function 값이라고 보시면 되겠고요.

이거는 일종의 리마인드인데, Bellman Expectation Equation을 아래와 같이 처음에 정의했고 그거에 대한 matrix 형태는 위와 같다고 말씀드렸는데요. 그거에 대해 다시 한 번 보여드리고 있습니다.

그래서 지금 말씀드린 과정에 대해서 한번 예제와 함께 보도록 하겠습니다. 지금 Problem setting은 어떻게 되어 있느냐 하면, 여러분이 취할 수 있는 action은 이와 같고요.

위로 가든지, 오른쪽으로 가든지, 왼쪽으로 가든지, 아래쪽으로 가든지.

그리고 일단 계산을 쉽게 하기 위해서 γ는 1로 주어졌다고 가정을 해보고요. 그리고 Terminal state, 종료되는 state는 왼쪽, 오른쪽의 회색. square가 Terminal state고요.

그리고 만약에 여러분이 1이라는 state에 있을 때 ‘북쪽으로 가라’라는 action을 취하면 아무데도 안 가고 그냥 1에 머물러 있습니다. 그래서 세 번째 얘기가 그 말씀을 드리는 거고요.

그리고 Reward는 어떻게 정의하느냐 하면, 갈 때마다 -1, move를 한 번 할 때마다 -1로 주어졌습니다.

그러니까 이 Reward로 주어졌다고 하면 최단거리로 이 Terminal state로 가는 policy가 학습이 되겠죠.

그리고 마지막으로 policy를 정해줘야 하죠. 지금까지는 MDP를 정의했고, 마지막으로 policy를 정의해줘야지만 그때의 value function을 구할 수 있는데요.

우리가 여기서는 Random policy를 가정했습니다. 그러니까 모든 state, 어떤 state든지 간에 북으로 가거나, 동으로 가거나, 남쪽으로 가거나, 서쪽으로 가는 확률이 다 0.25로 동일하다.

이런 문제 세팅이 주어졌다고 한번 가정을 해보고요. 이때의 value function을 구하는 것입니다. 이게 우리의 목적이고요.

value function을 구한다는 건 이 수많은 1~14의 state들이 있는데, 거기에서의 value값을 구하는 것입니다.

다음 슬라이드에 말씀드릴 Iterative Policy Estimation을 통해서 value값을 구하는 과정입니다.

그래서 소개시켜드린 대로 다음 step의 value function이라는 건 Reward+γ+state transition probability P+V. 이런 식으로 표현이 됐죠.

처음에 Iteration 0에서는 V를 모두 다 0으로 initialize합니다. 그런 다음에 P를 곱하고 γ를 곱한 다음에 다시 R을 더하면 이와 주어지고요.

이렇게 새롭게 구한 V를 가지고 다시 equation에 넣어서 P를 곱하고 γ를 곱하고 다시 R을 더하면 그다음 Iteration 2에서의 value function이 나옵니다.

그래서 이 과정을 계속 반복하다가 수렴될 때까지 진행을 하다 보면, 결국에 마지막으로 얻는 이 value function 값이 저희가 구하고자 하는 Vπ(s).

그러니까 Random policy를 우리가 따라간다고 했을 때 각 state의 value function 값을 이와 같이 구할 수가 있습니다.

그래서 좀 전에 말씀드린 내용은 evaluation, 그러니까 V를 구하는 과정에 대해서 말씀을 드렸고요.

여기 그림을 보시면 오른쪽에 Greedy policy에 대해서 소개가 되어 있는 것이 있습니다. 그런데 현재까지 구해진 value function을 기반으로 해서 Greedy policy를 구한 것입니다.

그래서 Iteration 0일 때를 보시면, 이 state를 보시면 오른쪽도 0.0, 아래도 0.0, 왼쪽도 0.0이기 때문에 이 state에서는 어디를 가든지 간에 다 동일합니다. 그래서 모든 방향으로 화살표가 있고요.

그리고 다음 step에서 V를 새롭게 구하면 이런 state에서는 마찬가지로 위, 아래, 왼쪽, 오른쪽이 다 -1이기 때문에 다 똑같은 확률로 구해지지만,

이 state를 기준으로 본다면 왼쪽은 0이고, 오른쪽, 아래쪽은 -1이기 때문에 여기서는 왼쪽으로 가는 optimal policy로 업데이트가 되죠.

저희가 계속 V를 구해가면서 그때마다 각 state에서 value function이 최대가 되는 방향으로 화살표를 긋는다면, Greedy policy를 계속 구할 수가 있고요.

그런데 재미있는 사실은 뭐냐 하면, value function을 구하려면 Iteration을 되게 많이 돌렸어야 되는데,

Greedy policy 같은 경우에는 V3, 그러니까 step3까지만 V를 구하면 optimal policy, 그러니까 수렴된 Greedy policy를 구할 수 있다는 사실입니다.

그래서 여기서 재미있는 건 뭐냐 하면, 여러분이 만약 policy만 구하고 싶은 경우에는 더 적은 Iteration만으로도 구해낼 수 있는 경우가 꽤 있다.

그리고 만약에 evaluation하고 싶다고 하면 충분히 Iteration을 돌려서 V가 더 이상 바뀌지 않을 때까지 수행을 해야 된다. 이 예제와 함께 볼 수 있었습니다.

그러면 여기서 또 하나 재미있는 내용은 Policy Improvement에 대한 내용입니다.

그러니까 좀 전에 말씀드렸다시피 policy가 주어졌을 때 evaluation하는, 그러니까 Vπ를 구하는 과정은 전 슬라이드에서 했습니다.

그런데 만약에 π가 주어졌을 때, policy가 주어졌을 때 Vπ를 구한 다음에 그 구해진 Vπ를 가지고 혹시라도 policy를 Improve할 수 있지 않을까? 그거에 대한 질문입니다.

그러니까 그 전 예제를 보면 우리가 처음에 가정한 policy라는 건 Random policy였습니다. 그러니까 어느 state든지 간에 Random으로 움직여라.

그런데 그 전 example에서 보셨다시피 오른쪽 그림에 보여드린 내용은 우리가 V를 계속 구함에 따라서 optimal policy도 계속 업데이트 된다는 걸 아실 수 있겠죠.

그래서 어떻게 하면 우리가 안 좋은 policy로부터 시작해서 더 나은 policy를 구할 수 있을까? 그 과정이 전 예제에서 힌트로 주어져 있다고 보시면 되겠습니다.

실제로는 어떻게 하느냐? policy를 계속 개선하려면 어떻게 하느냐? 이와 같이 하시면 됩니다.

여러분이 일단 policy를 하나 정합니다. 그러니까 그 전 슬라이드처럼 여러분이 특별한 아이디어가 없으면 그냥 Random policy처럼.

‘각 state에서 Random으로 어떤 action을 취해라.’ 이런 식으로 policy를 정합니다. 그렇게 정한 다음에 여러분이 그 policy에 대한 value function을 구합니다.

그다음에 구해진 value function에 대해서 policy를 Improve 하는데, policy를 개선하는데 어떻게 하느냐 하면, Greedy 아이디어를 통해서 policy를 한번 업데이트 해봅니다.

그러니까 Greedy라는 건 뭐냐 하면, 각 state에서 주변에 내가 취할 수 있는 action을 쭉 봐서 value function이 최대가 되는 action을 취하도록 그런 식으로 policy를 바꾸는 겁니다.

그렇게 바꾼 다음에 그러면 새로운 policy가 생겨났죠? 그럼 그 새로운 policy를 가지고 다시 Vπ를 구합니다. 그래서 value function을 다시 구합니다.

그래서 그 value function에 대고 Greedy하게 action을 취해나가면 또 새로운 policy가 나오겠죠. 이 과정을 계속 반복하다 보면 결국에 optimal policy로 수렴이 된다. 그게 밝혀진 사실입니다.

그래서 그걸 도식화하면 다음과 같습니다. 여러분이 처음에 policy를 initialization을 해요. 그리고 마찬가지로 value function도 initialize를 합니다.

value function은 좀 전에 예제에서는 다 0으로 initialize 했죠. 그다음에 Iterative Policy Evaluation을 통하면 새로운 value function을 구할 수 있었죠.

그 value function에 대고 Greedy한 action을 취하는 policy로 새롭게 업데이트를 합니다.

그래서 그 policy가 업데이트 되면, 그 policy를 기반으로 다시 Iterative Policy Evaluation을 통해서 V값을 구하고요.

이 과정을 계속 반복하다 보면, 여러분이 V\*, 그러니까 optimal한 value function과 optimal한 policy를 구할 수 있다. 이게 밝혀진 사실입니다.

그래서 Policy Iteration이라는 방법은 여러분이 optimal policy를 구하는 과정 중에 하나고요. 좀 시간이 걸리고 Iterative한 방법이긴 하지만, V랑 π를 initialize해서 일단 V 구하고,

그 V 구한 걸 기반으로 Policy Improvement 하고, Improve된 policy로부터 value function을 다시 구하고. 이 과정을 반복하면 결국에 수렴이 된다.

그래서 MDP에서는 이와 같이 여러분이 원하는 value function도 언제든지 구할 수 있고요. 원하는 policy도 언제든지 구할 수가 있습니다.

그래서 지금까지 말씀드린 내용에 대해서 요약을 하자면 다음과 같습니다. 크게 보면 문제가 두 개 있었죠? Prediction이 있었고 Control이 있었습니다.

Prediction은 policy가 주어졌을 때 그리고 MDP가 주어졌을 때 value function을 구하는 것이고요. Control이라는 건 MDP가 주어졌을 때 optimal policy를 구하는 것입니다.

그래서 좀 전에 말씀드린 내용으로 보시면 Bellman Expectation을 활용하고 그다음에 Iterative Policy Evaluation이라는 방법을 활용하면 여러분이 Prediction을 풀 수 있었습니다.

그리고 Control은 어떻게 푸느냐? Iterative Policy Evaluation 해서 value function이 구해지면 Greedy Policy Improvement를 하고요.

Greedy Policy Improvement를 하면 그거에 대해서 다시 Iterative Policy Evaluation을 하고요.

이 두 과정을 반복하는 것이고, 이 두 과정을 반복하는 방법을 Policy Iteration이라고 한다. 이렇게 보시면 되겠습니다.

지금까지 설명 드린 내용을 보시면, 제가 V를 활용해서 V 구하는 방법과 그다음에 그 구해진 V로부터 optimal policy 구하는 과정, 이런 과정을 설명 드렸는데요.

그런데 여러분이 똑같은 과정을 Q에 대해서도 할 수가 있습니다.

Q를 initialization 한 다음에 그 Q를 가지고 Iterative Policy Evaluation으로 주어진 policy에 대한 Q를 구하고, 그 Q에 대해서 다시 Greedy Policy Improvement를 하는 식으로 해도 됩니다.

그런데 좀 전에 보여드린 대로 V를 주로 다루는 이유는 뭐냐 하면, V를 다룰 때 Complexity가 더 적기 때문입니다.

왜냐하면, action이 총 m개가 있고 state가 총 n가 개가 있으면, 만약 V를 활용하면 복잡도가 m×n제곱입니다.

그런데 여러분이 Q를 활용하면 m제곱×n제곱입니다. 그러니까 V를 활용하는 게 더 효율적으로 구할 수 있다는 거고요.

상식적으로 생각하면 Q는 각 s에 대해서, 각 a에 대해서 다 정의를 해줘야 되죠.

V값은 어떻게 보면 state가 n개라고 하면 n×1 벡터로 표현할 수가 있지만, Q는 어떻게 됩니까? state가 n개고, action이 m개라고 하면 n×m matrix로 표현이 되죠.

그러니까 Q를 다룬다는 건 더 많은 메모리가 필요하고, 더 많은 computation이 필요하다고 이해하시면 되겠습니다.

지금까지는 MDP가 주어졌을 때 Dynamic programming 아이디어를 활용해서 저희의 주요한 두 가지 문제,

그러니까 value function 구하는 문제, policy 구하는 문제를 어떻게 할 수 있는지에 대해서 소개를 시켜드렸고요.

물론 이거 외에 Prediction을 하고 Control 하는 방법이 되게 다양하게 많이 존재하는데, 안타깝게도 본 강의에서는 일단 여기서 마치는 걸로 하고요.

혹시라도 강화학습에 더 관심이 많으시다면, 강화학습에 더 초점에 맞춰진 수업이나 책 그리고 강연을 참조하시기 부탁드립니다. 그럼 오늘의 강연은 여기서 마치도록 하겠습니다.

6.1.

안녕하십니까? 여러분과 함께 인공지능의 기초를 함께 배워보고 있는 서울대학교 컴퓨터공학부의 김건희입니다.

오늘은 인공지능에서 매우 중요한 주제 중 하나인 게임 이론에 대해서 함께 배워보도록 하겠습니다.

그럼 우선 게임 이론에서 말하는 게임이 어떤 의미인지에 대해서 함께 알아보도록 하겠습니다.

기본적으로 게임은 여러 명의 player가 함께 참여하고 있다고 가정을 하고 있습니다.

그리고 게임에 참여한 player들은 자기들이 취할 수 있는 action들 중에 하나를 선택해서 하게 되어 있고요.

그런 action을 통해서 어떤 보상을 얻는 상황을 게임 이론에서 말하는 게임이라고 할 수 있겠습니다.

그래서 일반적으로 게임에서는 여러 player의 결정 과정이 결국에 게임의 보상을 결정하는 상황이라고 보실 수 있겠고요.

그래서 우리가 일반적으로 얘기하는 게임과는 꼭 맞지 않을 수도 있습니다.

그래서 여기서 중요한 게임의 requirement, 그러니까 꼭 필요한 구성요소들을 알아보면, 가장 중요한 내용은 이 게임에 참여하는 player가 최소 2명 이상이다.

그러니까 여러 명의 agent 혹은 player가 함께 같은 상황에서 자신의 보상을 최대화하는 decision making을 계속 하는 과정이라고 보시면 되겠고요.

그리고 여러 player들은 각각 자기 자신에게는 여러 action들 중에 하나를 선택할 수 있다. 이런 action set이 다 같이 주어지게 됩니다.

그리고 플레어어들은 자기가 어떤 action을 선택했는지에 따라서 보상을 얻게 되고요.

그래서 지난 시간에 배운 강화학습이랑 어떻게 보면 시나리오가 매우 비슷하다고 할 수 있는데요.

하나의 가장 큰 차이는 게임에서는 player가 여럿이다. 그리고 이 player들이 서로 경쟁 관계에 있는 경우가 많다. 이렇게 이해하시면 좋겠습니다.

Game Theory에서 주로 사용하는 가정에 대해서 말씀을 드리도록 하겠습니다. 여기서는 3가지 가정에 대해서 말씀을 드리고 있는데요.

첫 번째 중요한 가정은 게임에 참여하는 모든 player들이 다 이성적이라는 것입니다. 이성적이라는 건 무슨 뜻이냐 하면, 각 player들은 자신만의 효용함수, utility가 존재하게 되고요.

그리고 이 player들은 자신의 utility를 최대화하기 위해 그것을 목적으로 선택을 한다. 그런 게 아주 중요한 가정이고요.

그다음에 세 번째 가정은 여러분이 생각하실 때 아주 직관적이지 않을 수는 있는데, 하나의 또 다른 중요한 가정은 모든 player들은 자신과 함께 게임하고 있는 다른 player들이 다 이성적이다.

그러니까 각각이 다 자신의 효용을 최대화하기 위해서 노력하고 있다. 이걸 모두 다 알고 있다는 사실입니다.

이 가정이 소개된 가장 큰 이유는 Game Theory 분석을 좀 더 쉽게 하기 위해서입니다.

만약에 예를 들어 어떤 player가 무작위로 자신의 action을 선택한다. 그런 player들이 많이 존재한다고 하면 더 분석하기가 어려운 측면이 있습니다.

물론 실제 우리 환경에서는 가끔 예측할 수 없는 행동을 함으로써 상대방을 교란시키는 경우가 매우 많고요.

그래서 Game Theory도 당연히 그런 특별한 상황에 대해서도 formulation을 하고 해결을 해나가야겠지만,

우선 저희가 기본적으로 배우는 게임에서는 모든 player들은 이성적이고 또 다른 player들이 다 이성적이라는 걸 모두 다 알고 있다. 그런 가정에서 한번 얘기를 계속해 나가도록 하겠습니다.

그리고 밑에는 게임의 대표적인 예들을 몇 가지 보여드리고 있는데요. 첫 번째로 보여드리는 건 체스 게임입니다.

체스 게임의 경우에는 흰 말과 검은 말을 두는 두 명의 player가 존재하고요. 이 각각의 player들은 게임을 이기기 위해서 최선의 노력을 할 것입니다.

그리고 각 player들은 상대방도 마찬가지로 이기기 위해서 최선의 선택을 계속할 것이다. 이걸 서로 알고 있다는 것이죠.

그리고 우리가 즐겨 하는 부르마블 같은 보드게임도 대표적인 Game Theory의 게임이라고 할 수 있겠습니다.

여기에서도 마찬가지로 여러 명의 player가 존재하고, 각각은 자신이 이기기 위해서 최선의 선택을 반복하게 됩니다.

그리고 정치적인 이슈일 수도 있지만, 나라들끼리 경쟁하는 모습, 특히 핵무기 개발을 위해서 나라들끼리 서로 경쟁하는 모습 역시도 게임 이론으로 설명할 수가 있습니다.

그래서 실제로 경제학과나 정치학과에서도 게임 이론을 활용하여 어떤 정치, 사회적 현상을 분석하기도 합니다.

그리고 마지막으로 우리에게 친근하지만 미처 생각하지 못한 게임의 예라고 한다면, 투표가 될 수 있겠죠.

그러니까 우리 모두에게는 한 표를 행사할 수 있는 action이 있고요. 그리고 입후보자들 중에서 저희가 한 명, 그러니까 저희의 효용을 최대화하는 한 명을 선택하게 됩니다.

그 과정도 Game Theory에서 게임의 예로 설명 드릴 수가 있겠습니다.

그러면 이제 Game Theory를 좀 더 수학적으로 표현하기 위해서 한 가지 예와 함께 시작해보도록 하겠습니다.

이 예는 죄수의 딜레마라고 많이 알려져 있고요. 아마 예능 등을 통해서 소개가 많이 되어 많은 분들이 알고 계시리라고 생각을 합니다.

우선 두 명의 죄수가 있습니다. 그리고 이 두 명의 죄수는 각각 조사를 받는데요.

이 두 명의 죄수는 두 가지의 action이 가능합니다. 조용히 있거나 혹은 상대방을 밀고하거나. 이 두 가지 경우가 가능하고요.

만약에 첫 번째 죄수가 상대방을 밀고하지 않고 조용히 있다. 그리고 P2라고 되어 있는 다른 죄수가 역시 마찬가지로 조용히 있다. 그 경우에는 두 명 다 1년 동안 감옥에 살게 됩니다.

만약에 두 명이 서로 밀고를 하는 경우에는 둘 다 3년형을 받게 됩니다.

그리고 또 재미있는 건 만약에 첫 번째 죄수가 상대방을 밀고하지 않고 조용히 있고 두 번째 죄수가 밀고를 한다면, 두 번째 죄수는 방면이 되는 거고요.

첫 번째 죄수는 대신 5년의 형을 받게 됩니다.

이 게임 전체를 보는 우리의 입장에서 죄수 두 명을 위한 최선의 선택은 둘 다 가만히 있는, 침묵을 지키는 것이 최선의 선택이라고 얘기할 수 있겠지만,

각 player 입장에서는 상대방이 어떤 action을 하든지 간에 가장 좋은 선택이라는 것은 밀고를 하는 것입니다.

상대방이 어떤 action을 하든지 간에 내가 취하는 최고의 action을 dominant한 action이다. 혹은 dominant한 policy 혹은 strategy라고 얘기합니다.

제가 만약에 Silent라고 action을 선택하면, 물론 상대방이 Silent도 역시 선택을 하면 최선이겠지만, 상대방이 Implicate, 그러니까 밀고하는 경우에 제가 -5, 그러니까 5년형을 받게 되죠.

상대방의 형에 상관없이 가장 좋은 선택이라고 하는 건 서로 밀고하는 이 마지막 경우라고 할 수 있겠습니다.

전체를 다 볼 수 있다면 서로 Silent, Silent가 제일 좋지만, 각각의 입장에서 최적의 선택이라는 건 밀고하게 되는 것이죠.

이런 딜레마 혹은 아이러니를 죄수의 딜레마에서는 표현하고 있습니다.

그래서 이건 공공의 비극이라고 일컬어지는 사회 현상이랑도 좀 관련이 있습니다.

그러니까 사회 전체적으로 볼 때는 내가 조금의 희생을 감수하는 것이 사회 전체 공공의 이익에는 최선이지만,

많은 경우에 제가 상대방을 속이거나 혹은 사소하게 법규를 어길 경우에 저한테 이득이 되는 경우가 많습니다.

그런 경우에는 저는 이기적으로 행동할 수밖에 없고요. 실제로 사회에서는 그런 현상이 많이 일어나게 됩니다.

그래서 사람들의 공공을 생각하기 이전에 자신의 이익을 먼저 생각함으로써 사회 전체의 효용이 떨어지는 그런 상황을 공공의 비극이라고 얘기하는데요.

죄수의 딜레마가 이런 공공의 비극과 아주 밀접한 관련이 있다고 할 수가 있죠.

그리고 중요한 건 만약에 player들이 서로 협력을 한다면 더 나은 효용을 얻을 수 있지만, 만약에 그게 보장이 되지 않는다. 혹은 상대방을 믿지 않는다.

그런 경우에는 밀고를 하게 되는 게 dominant한 전략이 될 수 있다. 좀 전의 예에서 함께 보셨습니다.

죄수의 딜레마의 예제로 핵무기의 경쟁, 그것도 죄수의 딜레마로 볼 수 있겠죠.

그러니까 서로 협동을 한다면 무기를 감축하는 것이 더 나은 선택이겠지만, 서로를 믿지 않기 때문에 계속 무기를 만들게 되는 그런 비극이 있을 수밖에 없고요.

또 얼마 전에 미국이 기후협약을 탈퇴하겠다고 그런 선언을 하기도 했었는데요.

공공의 이익, 지구 전체의 이익을 위해서는 서로 협력을 해서 이산화탄소의 배출을 줄이는 게 공공을 위해서, 모두를 위해서는 좋지만,

개인의 이득을 위해서는 많은 경우에 협약을 탈퇴하는 게 좋을 수도 있다. 그렇게 보실 수가 있겠습니다.

그럼 이제 지금까지 말씀드린 게임을 좀 더 정교화된 형태로 표현하는 방법을 배워보도록 하겠습니다.

그러니까 좀 전에 소개해드린 죄수의 딜레마에서는 여러 가지 문장을 통해서 죄수의 딜레마라는 게임을 설명 드렸는데요.

그렇게 하지 말고 수학적으로 한번 모든 게임을 동일 하게 표현을 해보자. 그게 정규 형태라고 보시면 되겠습니다.

게임을 정규 형태로 표현하기 위해서는 3개의 컴포넌트가 필요합니다. 이 3개의 컴포넌트에 대해서 정의를 해주셔야 되고요.

여기서 N이라는 건 참여자의 수입니다. 그러니까 player의 개수라고 할 수 있고요. 그리고 각각의 player는 i라는 인덱스를 사용해서 나타내도록 하겠습니다.

그리고 두 번째 중요한 요소는 action set입니다. 그러니까 action의 집합이라고 할 수 있죠.

여기 보시면 A라는 action set은 A1, 그러니까 player1이 취할 수 있는 action set 곱하기, 여기서 곱하기는 곱집합을 얘기하는 것이죠.

그래서 N명의 player가 지금 있으니까 A1부터 AN까지의 곱집합으로 표현을 하게 됩니다. 그걸 action set, A라고 표현을 하고요.

마지막으로 u는 utility function, 효용함수입니다. 여기서 효용함수는 정의역이 A이고 공역은 혹은 치역은 실수인데 N차원의 실수입니다.

왜냐하면, 여기 player가 N명 있기 때문에 만약에 player1부터 playerN까지 이런 action을 취했다고 했을 때, 그때 각각의 효용을 나타낸다고 보시면 되겠습니다.

u는 어떻게 보면 Output은 N dimenSional vector이기 때문에 player 각각에 대해서 u를 기술하기 위해서는 이와 같이 사용합니다.

아래첨자 i를 사용해서 이 아래첨자 i는 player i가 action A를 선택했을 때 player1, i가 얻는 효용을 이와 같이 표현하게 됩니다.

그래서 이런 아래첨자가 있다면 ui는 실수값 스케일러, 그러니까 실수값 하나이겠지만 일반적인 u는 N dimensional vector입니다.

왜냐? u라는 건 각각 모든 player, N명의 player의 효용을 모두 다 나타내는 벡터이기 때문입니다. 그래서 이와 같이 표현하는 방식을 게임의 정규 형태라고 합니다.

그러면 우리가 좀 전에 살펴봤던 죄수의 딜레마를 한번 정규 형태로 표현해보죠. 중요한 건 N이 무엇인지, A가 무엇인지, u가 무엇인지 각각 정의해야 합니다.

좀 전에 살펴봤다시피 죄수의 딜레마에서 player의 수는 2가 됩니다. 그래서 N은 2로 세팅을 할 수 있고요.

그리고 A라는 집합은 player1이 취할 수 있는 action의 집합, 그것의 곱집합으로 또 player2가 취할 수 있는 action의 집합. 이렇게 표현이 되겠죠.

그래서 player1은 Silent 혹은 Implicate, 이 두 가지 action이 있고요. 마찬가지로 player2도 조용히 있거나 혹은 상대방을 밀고하는 이 두 가지 action이 있겠죠.

utility는 말씀드렸다시피 각각의 action에 대한 효용을 매핑해주는 함수라고 이해하시면 되겠고요.

오른쪽에는 죄수의 딜레마를 설명하면서 utility에 대한 정의를 이렇게 드렸는데요.

여기서 보시면 player1이 Silent하고 player2가 Silent라는 action을 취하면 player1은 -1이라는 utility값을 얻고, player2도 마찬가지로 -1이라는 utility값을 가지죠.

그래서 이걸 바탕으로 해서 u라는 함수를 정해볼 수가 있습니다. 일단 u는 정의역이 A였죠. 그리고 A는 {S, I}와 {S, I}, 또 다른 {S, I}의 곱집합입니다.

그래서 이거의 가능한 요소는 (S, S), (S, I), (I, S), (I, I)가 되겠죠. 그러니까 두 개의 집합의 곱집합이기 때문에 각각의 A라는 요소는 action의 쌍이 됩니다.

그런데 action의 쌍 중에 모든 가능한 action의 쌍을 다 기술하고, 그 각각에 대해서 utility가 얼마인지 정의해야 될 필요가 있겠죠.

그래서 오른쪽에 보여드린 행렬을 통해서 만약에 player1가 S라는 action을 취하고 player2가 S라는 action을 취하면, 각각 -1, -1이라는 utility를 얻고요.

player1이 S, player2가 I라는 action을 취하면, player1가 5년 동안 감옥에 있어야 하고, player2는 무죄방면이 되는 것이죠. 그런 식으로 각각을 행렬을 통해서 정의할 수 있고요.

좀 전에 말씀드렸다시피 utility function의 공역 혹은 치역은 N dimenSional vector라고 말씀드렸는데, 여기서 보시다시피 각각의 Output은 2×1 벡터임을 알 수 있겠죠.

그다음에는 이제 또 다른 중요한 게임의 예에 대해서 하나 말씀을 드리도록 하겠습니다.

우리가 일반적으로 많이 얘기하는 Zero-Sum Game에 대한 것입니다. Zero-Sum Game은 부가 새로 생겨나지도 않고 없어지지도 않는다는 뜻입니다.

그러니까 어떤 사람이 1이라는 utility를 얻으면 다른 사람은, 반드시 누군가는 1이라는 utility를 잃어야 합니다. 왜냐하면, 새로운 부는 절대 생겨나지 않기 때문입니다.

만약에 여러분이 Zero-Sum Game을 2 player, 그러니까 player가 딱 2명만 있는 Zero-Sum Game으로 한정을 한다면 이와 같이 표현할 수 있습니다.

그러니까 player1가 a라는 action을 취했을 때 얻는 이득은 항상 player2가 a라는 action을 취했을 때 얻는 이득의 마이너스다.

그러니까 이 둘을 합치면 항상 0이 된다는 뜻입니다. 왜냐하면 한 사람이 얻으면 한 사람이 반드시 잃어야 되고요.

그래서 이 효용의 합, 모든 player의 효용의 합은 항상 0이어야 됩니다. 그러니까 어떤 action이든지 간에 모든 action에 대해서 항상 이게 만족이 된다면 Zero-Sum Game이라고 얘기합니다.

게임의 가정에 대해서 말씀을 드릴 때 모든 player는 자신의 효용을 최대화하기 위한 action을 취한다고 말씀을 드렸잖아요.

그래서 player1 같은 경우에는 u1(a)를 최대화하기 위해서 노력을 할 것입니다. 그리고 player2는 자신의 utility를 최대화하기 위해서 노력을 하겠죠.

그런데 자신의 utility라는 건 뭡니까? Zero-Sum Game에서는 -(상대방의 utility)입니다.

그래서 player2가 자신의 utility를 최대화하기 위해서 행동한다는 건 곧 상대방의 utility를 최소화하기 위해서 선택을 한다. 이렇게 이해하시면 되겠습니다. 재미있는 내용이죠.

player1은 u1(a)를 최대화하기 위해서 노력을 하고, player2는 똑같은 값을 최소화하기 위해서 노력을 한다고 보시면 되겠습니다.

가장 간단한 Zero-Sum Game의 예를 보시면, 가위바위보를 보실 수 있겠죠. 우리가 가위바위보에서는 똑같은 action을 취하게 되면 둘 다 얻는 utility는 0입니다. 서로 비겼기 때문에.

그런데 제가 만약 보자기를 냈고 상대방이 바위를 냈다고 하면, 나는 이기게 돼서 +1이라는 utility가 있고, 상대방은 지게 되므로 ­1이라는 utility를 갖게 됩니다.

그래서 Zero-Sum Game에서의 특징은 뭐냐 하면, 행렬 각각의 합을 구해보시면 다 0이 된다는 특징이 있습니다.

모든 경우, 그러니까 player1과 player2가 취할 수 있는 모든 action의 combination에 대해서 이 둘의 utility의 합을 구해보면 항상 0이 된다. 이렇게 보시면 되겠습니다.

하지만 죄수의 딜레마에서는 이 특성을 만족하지 않습니다. 그래서 죄수의 딜레마는 Zero-Sum Game은 아니다.

그러니까 전 슬라이드에서 각 element에 있는 두 숫자를 여러분이 더해보시면 각각이 0이 되지 않는다는 걸 쉽게 확인하실 수 있습니다.

그럼 Zero-Sum Game에 대해서 말씀을 드렸고요.

그다음에는 게임 이론에서 가장 중요한 개념 중에 하나인 Nash Equilibrium에 대해 설명을 하기 위해서 하나둘씩 새로운 개념에 대해서 정의를 해나가도록 하겠습니다.

우선 첫 번째로 정의해야 될 것은 전략입니다. 게임에서 player의 전략을 얘기합니다.

그래서 player i의 전략은 Si라고 표현을 합니다. 그 Si는 뭐냐 하면, i가 취할 수 있는 action에 대한 확률분포 값입니다.

그러니까 내가 어떤 action을 선호한다고 하면, 그 action에 대한 확률분포 값은 높을 테고, 어떤 action을 덜 선호한다고 하면 그 action 값은 낮은 수가 되겠죠.

그래서 확률분포라는 건 중요한 게 각각의 action에 대한 전략의 값은 항상 0보다 크거나 같아야 하겠죠. 왜냐하면 확률이기 때문에.

그리고 모든 action에 대해서 Si값을 다 합치면 총합은 1이 되어야 합니다. 그래서 Si(ai) 이런 식으로 표현을 하면, player i가 ai를 선택할 확률을 나타내게 되죠.

그러면 전략이라는 게 각각의 player에 대해서 정의가 되고요. 이건 action set에 대한 확률분포라고 말씀을 드렸고요.

그러면 지금 게임에는 N명의 player가 있다고 가정을 하면, 전략 프로파일을 우리가 정의할 수 있습니다.

이건 N명의 player 각각에 대한 전략들을 하나로 모은 것입니다. 그래서 S1부터 SN까지 모두를 아우르는 정의로 strategy profile이라는 용어를 사용하게 됩니다.

그리고 또 밑에는 중요한 정의가 하나 나와 있는데요. support라는 정의입니다. support는 뭐냐 하면, 확률론에서 나온 정의고요.

확률론에서 support라는 것의 정의는 뭐냐 하면, 이와 같은 확률분포가 있을 때 확률의 값이 0보다 큰 영역입니다.

그래서 이 부분은 support라고 할 수 있고요. 예를 들어서 이 영역에서는 확률값이 0이다. 그러면 이 부분은 support가 아닙니다.

그래서 support라는 건 확률값이 0보다 큰 영역을 지칭하는 거고요.

당연히 그 개념을 전략에 적용해보면, action들 중에 그 확률이 0이 아닌 action들을 support. 이렇게 얘기할 수 있습니다.

그리고 마지막으로 중요한 정의가 어떤 전략이 순수하다. 혹은 혼합돼 있다. 이런 정의인데요. 영어로는 pure, mixed.

pure strategy 혹은 mixed strategy profile 이런 식으로 얘기하는데, 어떤 전략이 순수하다는 건 뭐냐 하면, 모든 player들이 항상 똑같은 전략을 쓰는 경우.

그러니까 모든 player의 Si가 시간에 따라 변하지 않고 고정돼 있는 경우를 pure하다고 얘기하고요.

그리고 mixed, 혼합돼 있다는 뜻은 Si, 그러니까 어떤 action을 취하는 확률분포 상에서 우리가 randomness를 추가적으로 넣는 경우.

그러니까 어떤 player가 선택을 하는 데 있어서 Si를 따라서만 선택을 하는 게 아니라, 가끔 그걸 벗어나는 무작위성을 같이 가지고 있는 경우에 혼합된 전략이라고 얘기를 하게 됩니다.

지금 계속 정의가 나와서 어려우실 수도 있는데, 하나하나씩 밟아나가면서 계속 정의를 따라오시기를 바랍니다.

하나하나 정의할 때마다 그 전에 정의한 내용을 활용하고 활용하는 식으로 되어 있기 때문에 여러분이 그 전에 정의된 내용을 충분히 이해하셔야지 그다음 내용도 쉽게 이해하실 수 있습니다.

전략에 대한 profile을 얘기도 했고, 그다음에 정의해야 될 중요한 내용은 최선의 응답, 최고의 응답, Best response라는 겁니다.

이건 정의가 어떻게 되느냐 하면, 저를 제외한 모든 다른 player들의 전략이 주어졌다고 했을 때 내가 취할 수 있는 전략 중에 최선의 전략을 Best response라고 합니다.

그러니까 나를 제외한 모든 player들, 그 모든 player들의 전략이 주어졌다고 했을 때 내가 취할 수 있는 전략 중에 최선의 전략을 최선의 응답이라고 얘기합니다.

그래서 그걸 수학적으로 표현하면 이와 같습니다. 만약에 저를 제외한 모든 player의 정의를 이와 같이 표현할 수 있겠죠. S￢i 이런 식으로 한번 정의를 해볼 수 있습니다.

이건 strategy profile인데, 여기서 보시면 아시겠지만 Si가 빠져 있습니다. 그러니까 player i 입장에서는 나를 제외한 모든 다른 player들의 strategy profile이라고 보실 수 있겠습니다.

그래서 i번째 player에 대한 Best response라는 건 결국에 나를 제외한 모든 이들의 전략이 주어졌다고 했을 때 효용이, 그러니까 내가 취할 수 있는 최선의 선택이라는 것은 결국 모든 가능한 전략,

내가 취할 수 있는 전략 중에서 만약에 그 전략을 넣었을 때 얻을 수 있는 utility가 모든 다른 전략에 비해서 더 높은 경우를 최선의 전략.

여기서는 Si\* 이런 식으로 표현을 했죠. 그러니까 u(Si\*, S￢i)는 모든 Si에서의 utility보다 항상 높거나 같아야 합니다. 그래서 이와 같이 정의된 전략을 Best response라고 정의할 수 있게 됩니다.

그러면 모든 정의가 끝났고, Nash Equilibrium, 그러니까 Nash 평형점에 대해 말씀을 드리도록 하겠습니다.

어떤 profile, 그러니까 전략 profile S가 Nash Equilibrium에 있다.

이건 무슨 뜻이냐 하면, 모든 player들의 전략이 그 player를 제외한 나머지 전략이 주어졌을 때 최선의 전략이라는 뜻입니다.

그러니까 이쪽에 지금 Best response에 대한 정의가 되어 있는데요.

이 Best response가 첫 번째 player부터 시작해서 모든 N번째 player까지 다 Best response로만 이루어진 전략 profile을 Nash Equilibrium이라고 말씀드릴 수 있겠습니다.

그래서 Nash Equilibrium이라는 건 주어진 상태에서 모든 agent가 내가 어떤 전략을 바꿈으로써 얻을 수 있는 이득이 전혀 없다는 겁니다.

그러니까 나도 최선을 다하고 있고 남도 최선을 다하고 있고요. 그래서 평형 상태에서 내가 어떠한 전략을 조금이라도 바꾸면 나는 더 이상 얻는 게 없고 잃는 것만 있는 경우.

그런 상황이 모든 player에게 다 적용되는 그런 평형점을 Nash Equilibrium이라고 얘기합니다. 많은 경우에 게임에서 Nash Equilibrium은 여러 포인트가 존재하게 됩니다.

그러면 저희가 지금 Zero-Sum Game에 대해서 좀 전에 말씀을 드렸고요. 이 Zero-Sum Game에서 과연 Nash Equilibrium은 언제 나타나게 되는지에 대해서 말씀을 드리도록 하겠습니다.

그러면 간단히 하기 위해서 Zero-Sum Game 중에서 player가 2명만 있는 경우를 일단 표현해보도록 하겠습니다.

Zero-Sum Game에서 예제로 보여드렸던 게 가위바위보 게임이었는데, 가위바위보 게임에서 matrix, 행렬을 보시면 각 요소에 두 가지 숫자가 있고요. 이 두 가지 숫자의 합은 0이라고 말씀드렸습니다.

그렇다면 각 요소에 있는 첫 번째 숫자만을 취해서 일반적인 행렬을 만들 수가 있겠죠.

여기에 주어진 숫자는 player1이 두 번째 action을 취했을 때, player2가 첫 번째 action을 취했을 때 player1 입장에서의 utility값입니다.

그러면 player2 입장에서의 utility는 어떻게 될까요? 그 값에 마이너스를 붙이면 되겠죠.

왜냐하면 지금 Zero-Sum Game이기 때문에 여기 보셨다시피 player1이 1이라는 utility를 얻었다고 하면, 똑같은 상황에서 player2는 ­1을 얻어야만 하죠.

그래서 이걸 일반화하도록 하겠습니다. 그러면 행의 숫자가 m개가 있고 열의 숫자가 n개가 있는 행렬을 Two-player Zero-Sum Game을 위해서 정의를 할 수 있겠죠.

그러면 여기서 각각의 행이 의미하는 건 player1이 취할 수 있는 action을 나타냅니다.

제가 만약에 세 번째 행을 취했다. 그건 내가 취할 수 있는 m개의 action들 중에 세 번째 action을 취했다는 얘기고요.

만약에 네 번째 열을 선택했다. 그건 player2가 네 번째 action을 취했다고 보시면 됩니다.

그러면 3, 4에 있는 행렬값은 뭘 의미하게 될까요? 그건 player1이 얻는 utility입니다.

그러면 player2가 그 상황에서 얻을 수 있는 utility는 얼마나 될까요? 그것은 그 값에 마이너스를 붙인 값이 되겠죠.

그래서 좀 전에 말씀드렸다시피 Xij. i번째 행의 j번째 열의 요소는 효용입니다. player1 입장에서의 효용이고요.

그건 언제의 효용이냐? player1이 i번째 action을 취했을 때 그리고 player2가 j번째 action을 취했을 때입니다.

그래서 Two-player Zero-Sum Game 같은 경우에 이와 같이 행렬 형태로 전체 utility를 간략하게 표현할 수가 있겠습니다.

그러면 Two-player Zero-Sum Game에서의 Nash Equilibrium이 언제 나타나는지를 확인하기 위해서 Pure Maxmin Strategies에 대해서 소개를 시켜드리도록 하겠습니다.

여기서 Pure라는 건 좀 전에 말씀드렸다시피 randomness가 전혀 주어지지 않는다는 겁니다. 그 상황에서 player1이 입장에서 maxmin strategy가 무엇인지 한번 다뤄보도록 하겠습니다.

maxmin이라는 건 뭐냐 하면, 이름에서 아실 수 있다시피 player1 입장에서는 player2가 어떤 행동을 할지 제어할 수 없습니다.

대신 중요한 가정 중에 하나가 상대방은 자신에게 가장 유리한 action을 취할 것이다. 그걸 모든 player가 알고 있다고 말씀을 드렸죠.

그러면 player1 입장에서는 ‘내가 지금 이 상황에서 어떤 action을 취하는 게 최선일까?’라고 본다면, 결국에 각 행이 내가 취할 수 있는 action이고요.

내가 만약에 첫 번째 action을 취했을 때 얻을 수 있는 utility값이 첫 번째 행에 주어지겠죠.

그런데 가장 중요한 가정은 상대방은 상대방에게 가장 유리한 선택을 할 것이다. 그런 가정이 있기 때문에 그럼 상대방 입장에서는 첫 번째 행 중에서 x값이 가장 낮은 열을 선택하게 될 것입니다.

왜냐하면, 그게 player1의 utility를 최소화하는 거고요. player1의 utility를 최소화하는 게 player2의 utility를 maximize하는 거기 때문입니다.

그래서 player1 입장에서 최선의 action은 뭐냐 하면, 각 행마다 최솟값을 얻습니다. 최솟값이라는 건 내가 i번째 action을 취했을 때 player2가 최선의 action을 취한 상황입니다.

그래서 각 행의 최솟값을 보고 그 중에 최댓값이 되는 행을 선택하는 게 player1 입장에서는 최고의 선택이 되겠죠. 그래서 그게 Minmax 전략이라고 말씀을 드릴 수 있습니다.

그럼 다시 말씀드리면, 각 행의 최솟값을 다 구하고요. 그 최솟값들 중에 최댓값이 되는 행을 선택해라. 그런 것입니다.

첫 번째 행의 minimum 값, 두 번째 행의 minimum 값, 세 번째 행의 minimum 값을 각각 구한 다음에 이 전체에 대해서 maximum 값을 취하도록 해라.

그게 player1 입장에서는 최선의 전략이 될 테고요. 그걸 Minmax strategy이라고 할 수 있겠죠.

그래서 player2도 마찬가지입니다. player도 내 utility를 최선으로 하기 위한 선택을 할 거고요. 그 경우는 어떤 경우냐 하면, player1이 이성적이므로 내가 모든 열을 봅니다.

여기서 얻을 수 있는 최악의 경우, 최악의 payoff를 찾은 다음에 그것들 중에서 가장 좋은 열을 선택하게 됩니다.

그런데 중요한 건 지금 이 행렬은 player1의 utility 입장에서 기술이 되었습니다. 그래서 player2 입장에서는 실제로 다 마이너스를 붙여야지만 그게 player2 입장에서의 utility가 되겠죠.

그래서 똑같은 아이디어를 적용한다고 하면 각 열에서 maximum 값을 찾습니다. 그것은 player1 입장에서의 maximum이기 때문에 player2 입장에서는 최악의 경우죠.

그래서 max를 취한 다음에 각 열의 max들 중에서 min을 취하게 됩니다. 그래서 결국 player2 입장에서는 minmax 전략이 최선의 전략이 됩니다.

요약을 하자면 player1이나 player2나 기본 전략은 똑같습니다.

그런데 단지 이 행렬이 player1 입장의 utility로 기술되었기 때문에 player1 입장에서의 최선은 max가 되고, player 입장의 최선은 minimum이 됩니다.

그래서 min과 max가 이렇게 바뀌는 상황이 벌어지게 되는 것이죠.

그럼 예제와 함께 보도록 하겠습니다. player1 입장에서 한번 보도록 하죠. player1 입장에서 내가 만약에 action1을 선택했다고 해보세요.

그러면 player2가 첫 번째 action을 취해주면 내가 2라는 utility를 얻고요. 두 번째 action을 취하면 0을 얻고, 세 번째 action을 취하면 1을 얻게 됩니다.

그런데 상대방은 분명 자신의 최선을 위해서 선택할 거기 때문에 두 번째 action을 선택하겠죠.

내가 만약에 첫 번째 action을 취하면 상대방은 두 번째 action을 취해서 내 utility를 최소화하겠죠. 내 utility를 최소화한다는 건 상대방 utility를 최대화하는 거죠.

그래서 말씀드린 대로 내가 첫 번째 action을 취했을 때 최악의 경우는 minimum 값인 2입니다.

그리고 내가 두 번째 action을 취했을 때 최악의 경우는 ­3이고요. 그리고 내가 세 번째 action을 취했을 때 최악의 경우는 ­2입니다.

그래서 maxmin에서 말씀드렸다시피 각 행에서 minimum 값을 얻은 다음에 그 중에 최댓값을 얻는 거죠.

그러면 0, ­3, ­2 중에 최댓값은 0이 되겠죠. 그래서 player1 입장에서는 첫 번째 action을 취하는 게 좋습니다.

만약에 player1이 두 번째 action을 취했다고 해보세요. 그러면 player2가 첫 번째 action을 취해주면 제가 4라는 아주 좋은 payoff를 얻게 되지만,

그런데 상대방이 바보가 아니고 상대방은 되게 이성적인 player이기 때문에 당연히 두 번째 action을 취함으로써 제 payoff를 줄이고 자신의 payoff를 올리겠죠.

그렇기 때문에 제가 두 번째 action을 취하면 4를 보고 ‘내가 4를 얻을 수 있는데...’ 그러면서 두 번째 action을 취하면 절대 안 되고 이건 피해야 된다는 것이죠.

그래서 player1 입장에서는 첫 번째 action을 취하게 됩니다. 그게 maxmin 전략의 핵심이라고 할 수 있고요.

player2 입장에서는 마찬가지입니다. player2 입장에서는 제가 행 대신에 열을 선택하게 되죠.

그런데 말씀드린 대로 제가 첫 번째 action을 취하게 되면 최선으로 얻는 게 몇이 되겠습니까? 최선으로 얻는 게 -1이죠.

그러니까 좀 전에 말씀드렸다시피 이 숫자들은 다 player1 입장의 utility이기 때문에 player2 입장의 utility라는 건 다 마이너스가 붙어야 됩니다.

그래서 결국에 player2 입장에서 첫 번째 action을 취했을 때에 최악의 경우는 4고요. 두 번째 action을 취했을 때 최악의 경우는 0이고, 세 번째 action을 취했을 때 최악의 경우는 2 값이 되겠죠.

그래서 이 max들 중에서 min을 취하는 것은 바로 두 번째 action을 취해서 0을 얻는 게 최선이 됩니다. 그래서 일반적으로 maxmin solution이라는 건 이와 같이 정의되고요.

그래서 player1 입장에서는 최선의 전략이 a1이라고 했습니다. 그러니까 첫 번째 행을 선택하는 경우. 그리고 player2 입장에서는 최선의 action이라는 건 두 번째를 취하는 것입니다.

그럴 경우에 player1이 첫 번째 action을 취하고, player2가 두 번째 action을 취했을 때의 value는 0이 되겠죠. 그래서 여기서 말하는 v1과 v2는 v랑 동일하게 되는데, 여기서 v는 0이 됩니다.

그래서 이런 player1과 player2가 maxmin 전략을 통해서 선택했을 action에서 얻을 수 있는 value라는 걸 이 게임의 가치라고 정의를 합니다.

그리고 v라는 값과 더불어 player1의 action과 player2의 action을 모아서 maxmin solution 혹은 minmax solution이라고 합니다.

그래서 결국에는 Two-player Zero-Sum Game에서 행렬이 주어지면 player1 입장에서 maxmin을 하고, player2 입장에서 minmax를 함으로써

player1과 player2의 action이 정해지고 그때의 값도 정해지죠. 그래서 그걸 게임의 솔루션이라고도 얘기하고요. 그리고 v값은 특별히 그 게임의 가치라고 말씀을 드릴 수 있겠습니다.

그래서 이거의 의미는 무엇이냐? 이렇게 판단을 해보면, a1, b2, 그러니까 player1은 첫 번째 행, player2는 두 번째 열을 선택하는 게 maxmin solution이었고요. 그때의 v값은 0이었습니다.

그것의 의미는 뭐냐 하면, P1 입장에서는 P2가 뭘 하든지 간에, 어떤 전략을 취하든지 간에 내가 첫 번째 행을 선택하면 적어도 0은 얻을 수 있다. 이게 의미라고 할 수 있고요.

그리고 P2 입장에서도 마찬가지입니다. P2 입장에서도 P1이 어떤 전략을 선택하든지 간에 내가 두 번째 열, b2를 선택하면 내 입장에서는 적어도 -v는 얻는다.

그러니까 말씀드렸다시피 여기에 지금 주어진 v값은 player2 입장에서는 마이너스를 붙여야 자신의 utility가 되기 때문에 ­v가 됩니다.

그래서 maxmin solution은 많은 경우에 게임의 saddle point라고 합니다. saddle이라는 건 말 안장이죠.

말 안장의 특징이 뭐냐 하면, 한쪽 방향으로는 이렇게 생겼습니다. 그런데 반대쪽 방향으로는 이렇게 생겼죠.

그래서 saddle point라는 건 뭐냐 하면, 어떤 방향으로는 minimum이 되고 어떤 방향으로는 maximum이 되는 그 포인트를 saddle point라고 하고요.

지금 말씀드린 대로 maxmin solution 같은 경우에 row의 minimum이고, 열의 maximum이기 때문에 이거의 게임의 saddle point라고 얘기하게 됩니다.

그래서 지금까지 maxmin solution에 대해서 얘기했는데, 왜 이 얘기를 하느냐? 제가 처음에 이 논의를 시작할 때 Nash Equilibrium 말씀을 드렸는데요.

Two-player Zero-Sum Game에서 우리가 maxmin solution을 구했다. 그래서 ai와 bj를 구하면 그것은 곧 Nash Equilibrium point가 됩니다.

그래서 이건 증명된 사실이기 때문에 Two-player Zero-Sum Game에서는 여러분이 maxmin solution을 구하게 되면, 이건 항상 Nash Equilibrium이 된다고 이해하시면 되겠고요.

여기 보시면 iff가 되어 있습니다. 그건 무슨 뜻이냐 하면, if and only if입니다. 여러분이 그냥 equivalent, 동지다라고 이해하시면 됩니다.

만약 a, b가 Zero-Sum Game의 solution이면 a, b는 Nash Equilibrium이다. 그것도 만족이 되고요.

만약 a, b가 Nash Equilibrium이면 그 a, b는 Zero-Sum Game의 maxmin solution이다. 그것도 만족이 되는 겁니다.

그래서 필요충분조건이라고 보시면 되고, 그건 결국에 서로 equivalent, 동일하다고 보시면 되겠습니다.

지금까지 maxmin 전략에 대해 말씀드린 이유는 설명 드린 대로 각각 player 입장에서는 상대방이 어떤 전략을 취하든지 간에 내가 이 action을 취하면 적어도 이 값 이상은 guarantee가 된다.

그렇기 때문에 상대방이 이성적이라고 하면, 상대방이 자신의 이익을 위해서 최선을 한다고 하면, 이 선택을 하는 게 제일 낫겠죠.

그래서 player1도 그런 선택을 하고, player2도 그런 선택을 한다면, 그때가 바로 Nash Equilibrium이 된다. 그렇게 결론을 짓도록 하겠습니다.

그러면 지금까지 Two-player Game에 대해 말씀을 드렸습니다. 그런데 만약에 이걸 Two-player가 아니라, 3명의 player, 4명의 player 이런 식으로 확장을 하게 되면 이게 갑자기 어려워집니다.

이론을 전문적으로 공부하시는 분들만 이해하고 해석할 수 있을 정도로 갑자기 어려워지기 때문에 더 이상은 나아가지 않고,

그냥 여기서 Two-player Zero-Sum Game에 대해서 Nash Equilibrium은 이렇게 찾을 수 있다고 말씀을 드리고요.

Multi-player Game에서는 어떻게 되느냐? 여기에 혹시 관심 있으시면 아주 advanced된 정보가 필요하다고 이해하시면 되겠습니다.

그래서 Multi-player Game 같은 경우에는 몇몇 player들이 서로 협동을 할 수도 있고 서로 경쟁을 할 수도 있겠죠.

그리고 Multi-player에서는 Nash Equilibrium에 대해서 다룬다는 게 갑자기 너무 어려워지기 때문에 자세히 유도를 하거나 디테일하게 다루지 않지만 몇 가지를 말씀드리면,

만약에 player들 사이에 협력 혹은 합종연횡이 허락되지 않는다고 하면, Multi-player Game에서의 Nash Equilibrium은 찾을 수가 있다고 알려져 있고요.

그다음에 player들 사이에 협력을 허락을 한다. 그러니까 몇몇 player들끼리는 자기네들만 짜서 ‘최선의 노력을 하자.’ 이런 식으로 되는 게임에 대해서는

Nash Equilibrium이 어디서 존재하는지 수학적으로 증명하고 찾아내는 게 매우 어렵다고 이해하시면 되겠고요.

그래서 N명의 player가 있는 게임에서 그걸 일반화해서 N명이 있을 때 Nash Equilibrium이 언제 나타나느냐? 그거에 대한 명확한 이론은 아직까지 개발이 되지 못했다고 보시면 되겠습니다.

어떻게 보면 허무한 결론일 수 있는데, 저희가 게임 이론에 대해서 얘기했고 그다음에 Zero-Sum Game 얘기를 했고

그다음에 Two-player Zero-Sum Game에 대해서는 ‘Nash Equilibrium을 이럴 때 찾을 수 있다.’고 말씀을 드렸는데, Multi-player가 되면 갑자기 어려워진다고 이해하시면 되겠고요.

만약에 그럼 Zero-Sum Game이 아닌 경우에는 어떻게 되느냐? 그 경우는 게임별로 여러분이 특정 게임에 대해서 Nash Equilibrium을 찾을 수 있긴 하지만,

모든 Non-Zero-Sum Game에 대해서 통일해서 얘기할 수 있는 솔루션을 찾는다는 건 매우 어려운 일이 됩니다.

여기도 마찬가지로 만약에 player들이 서로 경쟁만 허락할 것인지 혹은 서로 연합까지도 허락할 것인지에 따라서도 분석이 매우 달라지고 어려워지게 되겠죠.

그래서 Non-Zero-Sum Game에서는 많은 경우에 우리가 관심 있는 특별한 게임에 대해서만 다루고요.

그래서 그거에 대해서 기술하고, 그때의 Nash Equilibrium이 어디인지 찾는 식으로 주로 진행하게 됩니다.

지금부터는 Zero-Sum Game이 아닌 게임들 중에서 몇 가지 잘 알려지고 유명한 것들에 대해서 하나씩 살펴보도록 하겠습니다.

첫 번째는 Stag Hunt라는 게임입니다. Stag Hunt는 수사슴을 사냥하는 게임이죠. 그건 다른 말로 assurance game, coordination game, trust dilemma라고 합니다.

그리고 이 게임은 정치학자로 혹은 철학자로 알려진 장 자크 루소도 이 게임에 대해서 기술한 적이 있고요.

게임의 시나리오는 이와 같습니다. 두 명의 사냥꾼이 있고요. 이 두 명의 사냥꾼은 사슴을 잡든지 아니면 수토끼를 잡든지 선택을 할 수 있습니다.

그런데 사슴은 항상 두 명의 사냥꾼이 ‘나는 사슴을 잡을 거야.’ 그래야지만 잡을 수 있습니다. 그러니까 한 명만 선택해서는 절대 잡을 수 없고요.

대신에 수토끼는 한 명이 만약에 선택을 하면 그 사람이 잡을 수가 있습니다. 하지만 내가 만약에 수토끼를 잡으면, 수토끼를 통해서 얻는 utility, 이득은 사슴을 잡았을 때 이득보다 훨씬 낮게 되죠.

일반적으로는 왼쪽에 보이는 행렬대로 표현을 하게 됩니다. 만약에 player1도 수사슴을 잡고 player도 수사슴을 잡으면 둘 다 a, a라는 utility값을 얻고요.

그다음에 player1이 수사슴을 잡고 player가 2가 수토끼를 잡으면 각각은 c, b라는 utility를 얻게 됩니다. 그런데 이 경우에는 당연히 a가 b보다 훨씬 커야 되겠고요. c는 b보다 작게 되겠죠.

그리고 둘 다 Hare, Hare, 그러니까 수토끼를 잡는다고 하면 둘 다 d만큼의 utility를 얻습니다.

이 경우는 여러분이 Stag Hunt 게임을 어떻게 정의하느냐에 따라 다른데, b와 d를 똑같이 줄 수도 있고 다르게 줄 수도 있습니다.

그래서 오른쪽에 예제가 하나 있는데요. 이 경우에는 두 사냥꾼이 만약 둘 다 수사슴을 잡겠다고 하면 둘 다 2라는 utility를 얻고요.

그다음 만약에 player1은 수사슴을 잡는데 player는 수토끼를 잡겠다. 그러면 player2는 1이라는 이득을 얻습니다. 그런데 player1은 아무것도 못 잡게 되기 때문에 0을 얻게 되죠.

그래서 이와 같이 기술을 한 게 Stag Hunt의 한 예라고 보실 수 있겠습니다.

두 번째 Non-Zero-Sum Game의 예제에 대해서 말씀을 드리도록 하겠습니다. 이건 성 대결에 대한 게임입니다.

여러분, 한 쌍의 부부가 있다고 가정을 해보시고요. 남편의 경우에는 풋볼 게임을 보러 가는 걸 원합니다. 그리고 부인의 경우에는 발레를 보러 가는 걸 원합니다.

이 경우에 부부가 서로 얘기하지 않는 상황에서 각각은 어떤 선택을 하는 게 좋을까? 그거에 대한 게임이라고 보시면 되겠습니다.

남편 입장에서는 당연히 풋볼을 보러 가고 싶긴 하지만, Wife가 풋볼을 싫어하고 또 발레를 선택할 수도 있다. 이렇게 예상을 한다면 발레를 선택할 수도 있겠죠.

그래서 내가 Husband 입장에서도 발레를 선택하고 아내 입장에서도 발레를 선택하면, 남편은 2라는 utility를 얻고 아내는 3이라는 utility를 얻습니다.

여기서 남편의 utility가 더 적은 이유는 남편은 원래 풋볼을 더 좋아하는 상황이겠죠. 그런데 그렇게 즐기지 않는 발레를 보러 가게 됐지만 그래도 아내가 좋아하니까 2라는 utility를 얻게 됩니다.

그런데 만약에 남편이 상대방을 위해서 발레를 선택하고 Wife는 또 상대방을 위해서 풋볼을 선택한다면,

둘은 같이 보러 가지도 못할 뿐더러 좋아하는 걸 보지 못하기 때문에 둘 다 0이라는 상황을 얻게 되고요.

그리고 만약에 남편이 풋볼을 선택하고 Wife가 풋볼을 선택하는 경우는 마찬가지로 발레, 발레를 선택한 경우와 utility값이 Husband, Wife 둘 다 조금씩 바뀌었다는 걸 알 수 있겠죠.

이렇게 성 대결 게임을 정의할 수도 있고요. 또 한 방법으로는 이 부분, 그러니까 남편이 풋볼을 선택하고 와이프가 발레를 선택할 경우에 대해서 조금의 utility를 주는 경우입니다.

여기서 조금의 utility를 주는 경우는 물론 남편 입장에서 아내와 함께 하지는 못하지만 그래도 자기가 좋아하는 풋볼을 보고, 와이프도 마찬가지로 자기가 좋아하는 발레를 보니까

그래도 남편이 발레 보러 가고 와이프가 풋볼 보러 가는 것보다는 더 낫지 않겠느냐? 그런 개념에서 여기서 1, 1이라는 값을 assign 했습니다.

그래서 크게 보면 이게 성 대결에 대한 게임이지만, 이렇게 디테일한 건 상황에 따라서 다른 payoff를 여러분이 정의할 수 있겠습니다.

그다음에 마지막으로 살펴볼 Non-Zero-Sum Game의 예는 많은 분들이 아시겠지만 치킨 게임입니다.

치킨 게임은 많은 경우에 매와 비둘기의 게임 혹은 snowdrift, 그러니까 눈바람이 불어서 눈 언덕이 되는 그런 게임입니다.

그래서 이 게임의 핵심은 뭐냐 하면, 두 player 중에 한 명이 겁쟁이가 되면 혹은 한 명이 양보를 하면 둘 다 benefit을 얻는 경우입니다.

여기서 한번 예를 보도록 하죠. 두 마리의 닭이 있는 경우에 나도 직진을 하고 상대방도 직진을 하면 사고가 나게 되죠. 그 경우에는 둘 다 치명적인 마이너스의 payoff, utility를 얻게 됩니다.

그렇지만 내가 직진을 하고 상대방이 피해주면, 나는 1이라는 utility를 얻고 상대방은 -1을 얻습니다.

상대방은 어쨌든 피했기 때문에 겁쟁이라고 놀림을 받을 수 있으니까 ­1이라는 작은 값을 주게 되죠.

여기서의 핵심은 뭐냐 하면, 둘 다 Straight를 취하게 되면 둘 다 망하는 거고요.

둘 중에 한 명이 양보를 하면 약간의 damage는 있긴 하지만 그래도 큰 damage는 없는 상황을 보실 수 있겠습니다.

그래서 여러분이 Nash Equilibrium이 언제 일어나는지를 보면, 둘이 서로 다른 action을 취할 때 나타난다. 이건 여러분이 Nash Equilibrium 정의를 적용해보면 쉽게 확인해볼 수 있습니다.

그래서 치킨 게임의 경우에는 Anti-coordinate game이라고도 불리고요. 그러니까 둘이 서로 다른 전략을 취해야지만 서로 이득이 되는 경우입니다.

이건 치킨 게임의 많은 경우에 갈등이 서로 증폭되는 과정을 표현하는 모델로 많이 사용이 됩니다.

그리고 Hawk-dove game은 마찬가지로 매와 비둘기의 게임인데, 우리가 위협을 그냥 보여주기만 하는 걸 Dove라고 하고요. 그리고 실제로 공격하는 걸 Hawk라고 합니다.

이것도 마찬가지로 둘 다 Hawk를 하면 큰 문제가 되는 거고요. 한 명이라도 Dove를 하면 그나마 나은 경우라고 보실 수 있겠습니다.

가장 좋은 방법은 상대방을 아주 위협해서 상대방이 Straight를 하지 못하게 하는 것도 방법이고요. 혹은 그 전에 서로 약속을 함으로써 이 상황을 벗어날 수가 있습니다.

현실에서는 이렇다는 걸 많이는 알고 있지만, 우리가 종종 여러 명의 player들이 서로 양보를 하지 않음으로써 점점 갈등이 증폭되는 경우도 실제로 많이 보게 됩니다.

지금까지 게임 이론의 기초에 대해서 말씀을 드렸습니다.

Zero-Sum Game에 대해서 말씀을 드렸고, Two-player Zero-Sum Game에 대해서 말씀을 드렸고, Zero-Sum Game이 아닌 몇몇 예제를 말씀드렸습니다.

게임 이론이라는 게 참 재미있는 분야이기도 하고요. 우리 현실이랑도 많이 맞닿아 있는 분야이기도 하고요.

그런데 이론적인 측면에서만 본다면 되게 과정이 많은, Two-player Zero-Sum Game에 대해서만 수학적으로 잘 분석이 되어 있고 아직 많은 부분은 밝혀지지 않은 상태로 남아 있다.

한편으로 보면 지금 우리가 다루고 있는 게임 이론에서의 게임은 실제 게임보다는 훨씬 단순화시킨 게임임에도 불구하고 거기서 어떤 수학적인 증명을 얻는다는 게 매우 어렵다는 걸 봤는데요.

실제 현상에서 이런 여러 player들이 서로 경쟁하거나 합종연횡 하는 경우에 어떻게 분석을 할까? 그건 아주 도전적인 문제로 남아 있습니다.

게임 이론은 인공지능뿐만 아니라 시작할 때 말씀드렸다시피 경제학이나 정치학에서도 아주 중요한 모델로 활용이 되기 때문에 관심 있으신 분들은 좀 더 공부해보시는 것도 추천해 드립니다.

그럼 오늘의 강의는 여기서 마치도록 하겠습니다.

7.1.

안녕하십니까? 서울대학교 컴퓨터공학부의 김건희입니다. 오늘은 인공지능의 기초 자연어 처리에 대해서 말씀을 드리도록 하겠습니다.

자연어 처리라는 분야는 인공지능 중에 언어지능에 대한 내용입니다. 그러니까 사람이 사용하는 언어를 기계도 이해하고 같이 활용하고 사용할 수 있도록 하는 학문이라고 보시면 되겠고요.

사실 자연어 처리를 인공지능의 하위 분야라고 단순히 말하기에는 자연어 처리라는 분야 자체가 요즘은 아주 큰 분야가 되고 있습니다.

그래서 다른 인공지능의 분야 그 이상이 될 수 있을 정도로 자연어 처리 그 자체로도 아주 큰 분야라고 이해하시면 되겠고요.

그래서 오늘 강연은 한 시간 안에 자연어 처리에 대해서 완벽히 말씀드리는 건 좀 어려울 것 같고,

자연어 처리라는 작업이 무엇인지 그리고 여러분이 이걸 한번 알아보시면서 흥미가 생긴다면 더 깊이 알아보는 기회가 될 수 있도록 활용하시면 좋을 것 같습니다.

첫 번째로 Reference, 그러니까 여러분이 참고할 만한 자료에 대해서 말씀을 드리도록 하겠습니다.

해외의 여러 대학들 중에 스탠포드대학이라는 곳이 자연어 처리 분야에서 선두주자 중 하나입니다.

그 중에서도 특히 Jurafsky 교수와 Manning 교수가 함께 가르친 자연어 처리 클래스가 있는데요. 그거에 대한 슬라이드는 온라인상에서 쉽게 찾아볼 수 있습니다.

그래서 만약에 여러분이 이 분야에 좀 더 관심이 있다면 이 강연을 한번 찾아보시는 것도 좋을 것 같고요. 이거 외에도 아주 다양한 강의가 온라인상에 존재합니다.

그리고 책으로는 이 두 가지 책이 가장 유명하다고 말씀드릴 수 있겠고요. 좀 전에 설명 드린 교수님 두 분께서 직접 저술하신 책입니다.

한편으로 보면 책이 처음 나온 지는 좀 시간이 지났기 때문에 근래에 딥러닝 기반으로 자연어 처리를 많이 수행하는데,

그것에 대한 부분은 누락되어 있을 수도 있긴 하지만 자연어 처리의 기초를 다지는 데는 아주 뛰어난 책입니다.

그리고 최근 딥러닝과 함께 자연어 처리가 어떻게 발전됐는지 여러 논문이나 그 외에 여러 강연 자료를 통해서 참조하시면 더 좋을 것 같은 생각이 듭니다.

그러면 자연어 처리에 대한 여러 가지 기술들에 대해서 아주 간략하게 소개를 시켜드리도록 하겠습니다.

왼쪽부터 오른쪽까지 쉬운, 그나마 많이 풀린 문제부터 아주 어려운 문제까지 여러 가지가 기술되어 있는데요.

많이 풀린 문제는 Spam detection이라고 스팸메일을 걸러주는 기능입니다. 물론 이 기능이 아직도 완벽하다고 할 수 없지만, 그래도 많은 부분 잘 구현이 되어 있고요.

그다음에 두 번째로 나와 있는 게 Part-of-speech, POS라고 줄여서 얘기하는데요. 그건 각 단어의 품사를 알아내는 것입니다. 이것에 대한 구체적인 내용은 뒤에 다시 소개시켜드리도록 하겠습니다.

그리고 그 밑에는 NER이라고 약자로 많이 쓰는데요. 명사들 중에 고유명사들이 있습니다. 그러니까 사람 이름, 단체 이름을 문장 안에서 찾아내는 게 NER이라고 얘기를 많이 하고 있습니다.

그리고 어떤 문장이 주어졌을 때 문장에 대한 감정이 좋은 감정인지 나쁜 감정인지를 알아내는 걸 Sentiment analysis라고 얘기하고요.

이건 아주 완벽히 풀리진 않았지만 그래도 많은 progress를 보이는 작업이라고 할 수 있겠고요.

그다음에 Coreference resolution은 지시대명사입니다. 그러니까 ‘그는’ 혹은 ‘이것을’ 이런 식의 지시대명사가 앞에 문장이나 단어들 중에 뭘 의미하는지를 찾아내는 과정이고요.

그다음에 WSD(Word sense disambiguation)라는 건 뭐냐 하면, 어떤 단어가 어떤 의미로 쓰였는지를 찾아내는 과정입니다.

이 예에서는 mouse가 컴퓨터 부품으로서의 mouse인지 아니면 쥐를 나타내는지를 얘기하고 있고요.

Parsing이라는 건 문장이 주어졌을 때 이 문장의 구조를 알아내는 것입니다. 그래서 주어, 동사가 있고, 목적어는 이 동사에 종속되어 있고. 이런 것들을 찾아내는 과정이고요.

그다음에 번역. 그리고 정보를 뽑아내는, 그러니까 예를 들어서 어떤 회사에 대한 보고서가 있다고 했을 때 자연어로 쓰인 보고서를 테이블 형태,

그러니까 회사의 위치, 회사의 revenue 이런 식으로 구조화된 정보를 뽑아내는 걸 Information extraction이라고 하고요.

그리고 여전히 어려운 건 질의응답에 대한 것과 그다음에 Paraphrase.

그러니까 어떤 문장을 써놓고 그 문장이랑 같은 뜻이지만 다른 형태로 표현하는 그 과정도 기계가 쉽게 하지 못하는 작업 중에 하나입니다.

그리고 아주 긴 문장을 한 단어로 요약한다든지 혹은 한 문장으로 요약하는 Summarization task.

그리고 근래에 인공지능 스피커들이 많이 나오긴 했지만, 여전히 대화를 할 수 있는 기계는 완벽하다고 말하기 어렵습니다.

그래서 이 중에서 모든 걸 다 자세히 말씀드리지는 않겠지만, 이 여러 task들 중에 몇몇은 여러분에게 좀 더 자세히 이번 시간에 말씀드리도록 하겠고요.

보시면 아시겠지만 워낙 다양한 분야가 있기 때문에 각각의 분야를 아주 깊이 아는 건 많은 시간과 노력이 필요하고요.

여러분이 혹시 관심 있는 분야가 있다면 한번 깊이 알아보시는 것도 좋을 것 같습니다.

그러면 자연어 처리가 왜 이렇게 어려운 문제가 되는지에 대해서 말씀을 드리도록 하겠습니다.

사실 사람이 하는 언어라는 게 많은 경우에 모호성이 있습니다. 우리가 대충 얘기해도 사람은 되게 잘 이해하는 경우가 있는데요.

우리가 대충 얘기한다는 게 정확하게 말을 표현하지는 않지만, 사람은 어떻게 되는지는 설명하기가 어렵지만, 잘 아는 사람이 되게 모호하게 얘기하더라도 그 의미를 완전히 파악하기도 하고요.

그리고 또 다른 이슈는 우리가 한국어를 모국어로 쓰고 있긴 하지만, 우리가 사용하는 언어들 중에 비문도 아주 많고요.

그러니까 문법적으로 완벽하지 않은 문장도 많고, 단어를 잘못 쓰는 경우도 있습니다.

그렇지만 그런 잘못 쓰는 상황에서도 사람들은 ‘이 context에서 이 사람이 이 용어를 잘못 썼지만, 이런 뜻으로 했겠거니.’ 하고 되게 잘 이해하는 그런 특성이 있습니다.

그래서 모호성이라는 게 이럴 때는 이렇고 저럴 때는 저런 문제이기 때문에 그걸 일일이 기계에게 가르쳐주는 건 매우 어렵습니다.

그래서 여기 보여드린 문장들은 하나의 문장으로서 이런 뜻이 될 수도 있고 저런 뜻이 될 수도 있는 그런 여러 가지 예제를 보여드리고 있고요.

하나씩 여러분이 보시면 되겠지만 제가 두 번째 문장을 예로 설명 드리도록 하겠습니다.

동사가 무엇이냐에 따라서 이 문장의 뜻이 완전히 바뀝니다. 여러분이 만약에 Strike를 동사로 본다면 ‘선생님이 놀고 있는 아이를 때렸다.’ 이렇게 해석할 수 있고요.

여러분이 만약에 Idle을 동사로 본다면 Teacher Strikes, 그러니까 ‘선생님들이 파업을 해서 아이들이 놀고 있다.’ 이런 식으로도 해석이 가능합니다.

이건 영어를 예로 들어서 말씀을 드렸는데요. 이건 꼭 영어만의 문제는 아니고 모든 언어에서 이런 모호성이 아주 많이 보이는 문장을 실제로 많이 보게 됩니다.

그리고 또 어려운 이유는 몇 가지 말씀드리면, 우선 영어 기준으로 말씀드리도록 하겠습니다.

첫 번째 어려운 이유는 모국어가 영어인 사람이라고 하더라도 많은 경우에 비문을 쓰기도 하고요.

혹은 요즘과 같은 소셜 네트워크 환경에서 일부러 문법적으로 맞지 않는 문장을 쓰는 경우도 많습니다.

그리고 이런 경우가 매우 빨리 변하기 때문에 이걸 다 해결한다는 게 쉽지 않다는 것이고요.

그리고 또 하나의 어려운 점은 분절을 하는 데 있어서 어려움이 있는데요. 여기서 보면 ‘the New York-New Heaven Railroad’ 이런 예제가 있습니다.

그런데 여러분이 만약에 New York이 도시 이름이고 New Heaven도 도시 이름이라는 걸 알면 두 번째처럼 분절을 할 수 있지만,

이런 식으로 단어가 연속으로 되어 있을 때 어디서 끊어 읽어야 되는지, 어느 단어와 어느 단어를 합쳐야 되는지 쉽지 않은 경우가 꽤 많습니다.

그리고 세 번째 어려운 점은 관용어구라고 할 수 있겠죠.

첫 번째 예로 보면 dark horse. 그래서 이 단어가 있었을 때 그냥 ‘검은 말’ 이런 식으로만 해석을 한다면, 이거의 실제 의미를 제대로 전달할 수 없겠죠.

그다음에 또 하나의 어려운 점은 신조어입니다. 그러니까 언어라는 게 매우 빠르게 변하고 있고 새로운 단어가 계속 나타나게 됩니다.

unfriend. 사실 friend는 ‘친구’라는 명사인데, 소셜 네트워크에서 friend가 일종의 동사처럼 ‘친구 맺다’로 사용이 되고, 당연히 친구를 끊는 걸 unfriend라고 하는 것이죠.

그런 unfriend라는 단어는 예전에는 없었는데 사회가 발전하면서 계속 생겨나게 되고요. 그런 단어는 매우 많습니다.

그리고 또 하나의 어려운 점은 여러분이 자연어 처리를 잘하기 위해서 언어만 이해해서는 안 되고 지식이 있어야 됩니다.

예를 들어서 이런 두 가지 문장이 있을 때 걸 제대로 해석하기 위해서는 Mary가 누구인지, Sue가 누구인지 알아야 됩니다.

그리고 많은 경우에 드라마 제목이 나올 수도 있고요. 여러분이 그 드라마를 모른다고 하면 역시 마찬가지로 그 자연어를 이해할 수 있다고 얘기할 수 없겠죠.

그리고 또 하나는 어떤 entity, 그러니까 어떤 요소를 얘기하는 데 있어서 좀 어려운 경우도 있습니다.

A Bug’s Life. 이런 건 3개의 단어 자체가 영화 하나를 나타내는 명사입니다. 그런데 그걸 만약에 모른다면 되게 어려울 수도 있겠죠.

그리고 Let It Be도 어떻게 보면 song title입니다. 그러니까 음악 노래의 제목인데, 이걸 모른다고 하면 마찬가지로 문장을 제대로 이해하는 게 매우 어렵겠죠.

그래서 이 모두를 아우르는 게 또 하나의 중요한 특징은 이 모든 게 되게 빨리 변한다는 것입니다.

그래서 영어를 잘못 쓰는 경우도 standard, 그러니까 표준에 맞지 않는 영어를 쓰는 방식도 매우 빠르게 변하고요.

그다음에 관용어도 빠르게 변하고요. 그리고 World knowledge, 지식체계도 빨리 변하기 때문에 그런 이유에서 사람의 자연어를 이해한다는 건 매우 어려운 일이 됩니다.

대신에 우리가 일상적으로 쓰는 말을 기계가 이해할 수 있도록 한다는 것이 대신에 인공지능의 다른 분야보다 더 재미있게 하는 요소가 될 수도 있을 것입니다.

그러면 지금부터는 자연어 처리의 대표적인 task, 작업들에 대해서 하나둘씩 말씀을 드리도록 하겠습니다.

첫 번째는 질의응답에 대한 것이고요. IBM이 Jeopardy라는 미국의 퀴즈쇼에 Watson이라는 기계로 참여해서 인간 챔피언들을 이긴 사건이 2011년도에 있었죠.

그때의 예를 한번 들어보도록 하겠습니다. 이건 실제 Jeopardy에서 나온 문제인데요. 여기 보시면 문제가 되게 어렵습니다.

문제의 특징이 뭐냐 하면, 여러분이 하나의 지식만 알면 안 되고요. 여러 개의 지식, 2개 이상의 지식을 동시에 알고 있어서 거기서부터 추론을 해야지만 답을 할 수 있는 것입니다.

WILLIAM WILKINSON, 이분은 한국 분들이면 당연히 잘 모르실 텐데 영국의 외교관이고요.

영국의 외교관이 이런 책을 썼습니다. 책 제목은 「왈라키아와 몰도비아 공국에 대한 설명」 이런 식의 책을 썼습니다.

그런데 이 책은 이 저자의 가장 유명한 소설에서 아이디어를 받았다. 그러니까 영감을 얻었다고 보시면 되겠습니다.

그래서 답은 뭐냐 하면, 오른쪽에 나와 있다시피 Bram Stoker입니다. 아시다시피 Bram Stoker는 「드라큘라」라는 소설의 작가고요.

여기서 보시다시피 왈라키아, 몰도비아가 루마니아 근처를 지칭하고 있고, 그게 「드라큘라」라는 소설의 배경이라는 걸 아셔야 됩니다.

그러니까 이 책에 대한 정보도 알아야 되고, 「드라큘라」의 저자도 알아야 되고, 여러 가지 지식을 가지고 있어야지만 풀 수 있는 문제입니다. 어떻게 보면 되게 어려운 문제겠죠.

그래서 질의응답은 여러 가지 경우가 있는데요. Factoid questions, 그러니까 사실관계에 대한 질문입니다.

조금 전에 보셨다시피 사실관계에 대한 질문은 많은 경우에 단답형으로 알 수가 있겠죠.

그러니까 ‘누가 뭘 했냐.’ 여기서 보시다시피 ‘칼로리가 몇이냐’ 이런 식의 질문은 단답형 하나로, 많은 경우에 명사 하나 혹은 명사인데 단어 여러 개로 답을 할 수 있겠죠.

그렇지 Factoid questions보다 더 어려운 건 narrative, 기술하는 문제입니다. 그러니까 ‘00를 설명해보시오.’ 그리고 질문 중에 ‘왜 이렇게 했습니까?’

그건 단순히 단어 한두 개로 표현할 수 있는 계제가 아니기 때문에 이런 경우에 매우 어렵습니다.

그래서 현실적으로 현재 나와 있는 질의응답 시스템 같은 경우에 첫 번째 Factoid questions에 대한 질의응답 시스템은 많이 존재하지만,

narrative questions에 대해서는 아직 응답을 제대로 할 수 있는 기계학습 혹은 자연어 처리 알고리즘이 많이 존재하지 않는다고 이해하시면 되겠습니다.

그리고 또 하나 재미있는 작업은 Information Extraction입니다. 이건 근래에 메일 시스템에서 어느 정도 구현이 된 것인데요.

제가 만약에 특정 교수 분과 미팅에 대한 메일을 주고받는다고 했을 때

이 중에서 장소와 시간에 대한 정보를 자동으로 뽑아서 개인 달력의 entry로 자동으로 만들어주는 그런 예가 Information Extraction의 예입니다.

그래서 이벤트 제목, 날짜, 언제부터 언제까지 그리고 위치가 어디인지. 이런 것들을 이메일 텍스트만 보고 자동으로 생성해주는 작업을 Information Extraction이라고 볼 수가 있습니다.

그리고 이건 아주 관련이 깊은 건데, 관계를 뽑아내는 겁니다. Information 중에서 특별히 관계에 대한 것입니다.

하나의 예를 보시면 이건 일반 사람이 쓴 회사에 대한 보고서입니다. 그런데 그냥 사람이 이해할 수 있는 자연어로 기술이 되어 있죠. 글로부터 이런 여러 가지 관계 정보를 알아내는 것입니다.

그러니까 IBM과 Location과의 관계. IBM은 어디에 있느냐? 뉴욕에 있고, IBM이랑 Founding Date와의 관계. 이런 것들을 알아내는 것이죠.

결국에 이건 세 가지 요소로 표현이 됩니다. A, B 그리고 A, B와의 관계로 표현이 되겠죠.

그래서 IBM과 1911은 설립연도라는 관계로 맺어진다. 이런 식으로 이런 관계들을 자연어에서 자동으로 찾아내는 task를 Relation Extraction이라고 할 수 있겠죠.

이건 위키피디아 페이지를 보고 기계가 자동으로 수많은 관계를 알아내는 게 과연 가능할까? 이것에 대한 내용입니다.

Standford는 The Leland Standford Junior University의 약자. 얘네 둘은 똑같은 거고, 캘리포니아에 있고.

이런 것들을 어떻게 하면 이런 긴 문장에서 자동으로 알아낼 수 있을까. 이게 Relation Extraction 문제라고 보시면 되겠습니다.

그리고 자연어에서 정보를 뽑아내는 것과 감정에 대한 분석을 동시에 수행하는 예제가 이렇게 있습니다.

예를 들면 쇼핑 사이트에 가보시면 어떤 카메라에 대해서 여러 가지 리뷰가 있을 텐데요.

그런데 리뷰가 많은 경우에 ‘카메라가 좋다, 나쁘다.’ 이 얘기뿐만 아니라 기능별로, 그러니까 특성별로 다 되게 얘기합니다. ‘값은 싸지만 기능은 별로다.’ 이런 식으로 얘기할 수 있습니다.

그래서 사람이 쓴 자연어 문장으로부터 Attributes, 어떤 속성별로 좋은 평인지 나쁜 평인지를 찾아내고요. 그리고 거기에서의 키워드를 이렇게 뽑아낼 수 있는지.

그게 Information Extraction과 Sentiment Analysis를 같이 결합한 문제라고 보실 수 있겠고요.

감정 분석은 되게 다양한 예제가 있습니다. 단순히 어떤 영화 리뷰가 positive한지 negative한지, 긍정적인지 부정적인지만 알아내는 게 목적일 수도 있고요.

또 밑에는 재미있는 연구 결과인데요.

소비자 확신, 그러니까 소비자가 얼마나 소비를 할 건지에 대한 설문조사를 원래 전통적으로 해야 되지만, 트위터의 사람들의 감정을 통해서 예측할 수도 있다. 이걸 보여준 내용이고요.

그러니까 트위터 상에 존재하는 수많은 트윗들을 기계가 자동적으로 분석해서 실제 설문을 하지 않고 과연 소비자들이 어떤 생각을 하고 있는지 알아내는 그런 작업이라고 볼 수 있겠습니다.

아시다시피 웬만한 쇼핑 사이트를 가보시면 사람들이 자유롭게 쓴 리뷰 글로부터,

그러니까 각 속성별로 이 사람이 어떤 스코어를 주고 있는지 그리고 그 스코어를 주면서 키워드들은 무엇인지를 뽑아내는 기능이 현재 일부 구현이 되어 있습니다.

그래서 Sentiment Analysis를 자세히 보면, 이 Sentiment Analysis는 여러 가지 다른 이름을 가지고 있습니다.

Opinion mining, 그러니까 의견을 mine 한다. 혹은 extraction이라고도 얘기하고요. 그다음에 주관적인 분석이라고도 얘기합니다. 그래서 다양한 이름으로 불리고요.

그래서 Sentiment Analysis는 그냥 하나의 task가 아니라 되게 다양한 task가 있는데, 단순히 긍정/부정, 그 둘만 찾아내는 게 목적일 수도 있고요.

혹은 별점으로 해서 1점부터 5점까지 과연 몇 점이냐? 그걸 알아내는 것도 있을 수 있겠고요.

그다음에는 어떤 감정의 대상이 뭐고, 감정의 근원이 뭐고, 여러 Attribute별로 어떤 복잡한 반응을 가지고 있는지를 다 찾아내는 건 훨씬 어려운 task가 되겠죠.

그래서 Sentiment Analysis도 어느 정도 많이 진행되긴 했지만, 완벽히 되고 있다고 보기는 어렵습니다.

그래서 왜 어려운지 몇 가지 예제와 함께 보시면, Sentiment를 알아내기 위해서는 되게 미묘한 점이 있습니다.

첫 번째 예제를 보시면, 이건 Dorothy Parker라는 평론가가 Katherine Hepburn이라는 배우에 대해서 평을 한 것입니다.

그런데 어떻게 평을 했느냐 하면, ‘그녀는 감정의 표현이 A부터 B까지 되게 화려했다.’ 이런 식의 평을 했습니다.

그런데 언뜻 들으면 되게 좋은 뜻인 것 같지만 A에서 B까지 아주 감정을 풍부하게 표현했다. 그건 결국에 비꼬는 내용입니다.

그러니까 가수한테는 ‘성량이 되게 풍부해서 도부터 솔까지 자유롭게 노래를 불렀다.’ 이런 식으로 좀 비꼬는 내용이죠. 이건 사람조차도 positive인지 negative인지 알기도 어렵고요.

그리고 많은 경우에 사람들이 결국에 부정적인 얘기를 하고 싶은데 처음에 그냥 좋은 얘기를 이렇게 나열하는 것이죠. ‘영화 괜찮고 여러 가지가 좋았는데 견딜 수가 없었다.’ 이런 식으로 쓰는 경우.

그래서 단순히 어떤 단어가 쓰였는지만 보면, 좋은 단어가 많이 쓰였기 때문에 이게 positive라고 생각할 수 있지만 실제로는 그렇지 않다는 거고요.

그리고 순서에 따라서도 영향이 있는데, ‘키아누 리브스는 평소처럼 특별한 게 없었고, 놀랍게도 매우 뛰어난 배우인 이 사람이 별로 안 좋았다. 그래서 놀랐다.’

그거는 뭐냐 하면 되게 잘해야 되는 배우가 별로였다는 얘기죠.

그래서 결국에는 negative한 평인데, 이것도 마찬가지로 긍정적인 단어와 부정적인 단어의 순서에 따라서 의미가 많이 바뀔 수 있다. 이런 몇 가지 예제를 보여주고 있습니다.

그래서 ‘자연어 처리가 쉽지만은 않구나.’ 여러분이 많이 느끼시지 않을까 생각합니다.

그리고 Named Entity Recognition, NER이라고 말씀을 드렸죠? 여기서의 목적은 뭐냐? 목적은 뭐냐 하면, 주어진 문장에서 여기서 보신 예제와 같이 고유명사를 찾아내는 것입니다.

그리고 그 고유명사가 사람인지, 날짜인지, 위치인지 혹은 단체인지를 알아내는 것이죠.

Labor라고 하면 그냥 ‘노동’ 이런 식으로 여러분이 이해할 수도 있지만, 이 context상에서는 ‘정당’이라는 뜻이죠. 그런 식의 주변 context를 봐서 이게 과연 고유명사인지 알아내는 거고요.

고유명사인지 알아내는 게 왜 중요하냐 하면, 여러분이 뽑혀진 고유명사에 따라서 문서로 정리, indexing을 할 수도 있고요.

그다음에 여러분이 어떤 회사와 혹은 product, 상품에 대한 감정을 하려면 이런 것들을 기본적으로 다 찾아내야 되고요.

그다음에 Information Extraction에서도 필요하고, 질의응답에서도 필요합니다. 어떻게 보면 가장 기본적인 task 중에 하나겠죠.

그리고 또 하나 중요한 문제 중에 하나가 Part-of-Speech Tagging입니다. 이건 좀 전에 말씀드렸다시피 품사에 대한 얘기입니다.

그래서 문장이 주어지면 각각의 문장을 구성하는 단어들의 품사를 알아내는 것이고요.

그래서 아시다시피 명사는 고유명사도 있고 일반명사도 있고요. 그리고 동사 같은 경우에는 일반적인 동사도 있지만 이렇게 can, had 이런 보조동사도 있고요.

이런 것들을 알아내는 과정이라고 보시면 되겠습니다.

그래서 Plays, well, with, others라는 문장이 주어지면 과연 Plays가 뭔지 well이 뭔지를 알아내는 거고요.

밑에 보시면 어렵게 생긴 이 단어가 각각의 태그입니다. VBZ가 뭘 나타내는지는 이 밑에 주어진 웹사이트에 가셔서 확인하셔야 되는데요.

간단히만 말씀드리면 동사고요. 그런데 동사로만 끝나는 게 아니라 3인칭 단수 현재형 동사다. 그런 식으로 구체적으로 알아내는 과정이죠.

그래서 이 태그의 의미가 안 와 닿으실 수도 있는데, 그런 분들은 이 웹사이트에서 각각의 태그가 의미하는 게 무엇인지.

실제로는 품사라는 게 종류가 매우 많습니다. 여기서 보시다시피 Plays가 이런 의미일 수도 있고 이런 의미일 수도 있다.

그러니까 복수형 명사일 수도 있고 좀 전에 말씀드렸다시피 3인칭 현재형 동사일 수도 있고요. 그런 것 중에서 동사였다. 이런 걸 지금 표현해주고 있고요.

그래서 Part-of-Speech Tagging이 왜 어렵냐 하면, 영어 같은 경우에 거의 모든 단어가 하나 이상의 품사를 가지고 있습니다.

그래서 that을 예로 들면, 이건 접속사일 수도 있고 지시대명사일 수도 있고 부사일 수도 있습니다.

그렇기 때문에 이게 어떻게 쓰이느냐? 주변 단어랑 같이 봐서 결정을 해야 되기 때문에 그렇게 쉽지만은 않은 문제라고 보시면 되겠습니다.

그리고 또 하나 task를 소개시켜드리면, 구조에 대한 Parsing입니다. 이건 문장이 주어지면 문법적인 구조를 알아내는 과정이라고 보시면 되겠습니다.

하나의 예를 보여드리도록 하겠습니다. ‘I saw a girl with a telescope.’ 이런 문장이 주어졌을 때 이건 두 가지로 해석이 됩니다.

만약에 with a telescope가 saw랑 연결이 됐다고 해보세요. 그 경우에는 ‘나는 저 소녀를 망원경으로 봤다.’라는 뜻입니다.

그런데 만약에 telescope가 girl에 연결돼 있다고 하면 ‘나는 망원경을 가지고 있는 소녀를 봤다.’ 이런 뜻이 됩니다.

그래서 문장 구조를 알아낸다는 게 문장의 뜻을 정확히 이해하는 데 매우 중요한 task라고 이해하시면 되겠고요.

그래서 이 문장 구조를 어떻게 나타내느냐? 그걸 크게 보면 두 가지 관점에서 볼 수 있는데요.

한 가지 관점은 구성 요소로 나누는 것입니다. 문장이 있으면 with a telescope는 구조구, 전치사구입니다.

그래서 문장이 주어지면 그걸 여러 개의 구로 나누고, 다시 그 구는 또 여러 개의 구성요소로 나누고.

이게 Constituency라고 보이는 view고요. Dependency는 뭐냐 하면, 단어 사이의 종속관계입니다.

그래서 I, 주어는 saw라는 동사에 종속이 되어 있고, a는 girl이라는 명사에 종속되어 있습니다. 이런 종속관계를 알아내는 관점으로 문장 구조를 해석한다고 보실 수도 있습니다.

그래서 이 두 가지는 결국에 같은 얘기인데, 이렇게도 해석을 할 수 있고 저렇게도 해석할 수 있다고 이해하시면 되겠고요.

좀 더 자세히 보면 첫 번째는 구성요소별로 Parsing, 그러니까 이해를 하는 거죠. 구조를 이해하는 건데요.

그건 어떤 구가 어떤 구성, 하나로서의 구성 요소인지를 어떻게 판단하느냐? 그걸 두 가지 특성을 통해서 판단할 수 있다는 예제를 보여드리고 있습니다.

그러니까 여기 보시면 about drugs라는 구가 있고요. to the children이라는 구가 있습니다.

그런데 이 두 개의 문장을 보시면 이 두 개의 구는 하나가 앞에 나와도 되고 하나가 뒤에 나와도 됩니다. 그러니까 영어 문법 상 둘이 다 맞는 경우죠.

그런 식으로 어떤 구성요소가 움직일 수 있다, 분배가 될 수 있다고 하면 ‘얘는 구성요소다.’ 이렇게 정의할 수도 있다는 거고요.

또 하나는 치환 관점에서 볼 수 있습니다. 어떤 구성요소가 다른 구성요소로 계속 치환이 가능하다. 그러면 어떻게 보면 그게 하나의 구성요소로 Parsing이 돼야 한다고 이해할 수도 있겠죠.

그리고 Dependency 같은 경우에 마찬가지로 각각의 단어가 어디에 종속되어 있느냐?

그래서 on이라는 전치사가 동사에 연결이 되는지 혹은 그 앞에 있는 명사에 연결이 되는지에 따라서 의미가 다 바뀌기 때문에

그런 것들의 종속관계를 파악하는 것도 아주 중요한 Parsing의 view라고 보시면 되겠습니다.

그리고 그다음 task입니다. 그다음 task는 단어의 의미에 대한 task고요. 단어의 기본 구성, 그러니까 단어의 요소는 여러 가지 의미를 가지고 있습니다.

많은 경우에 영어 같은 경우는 한 단어가 하나의 뜻만 가지는 경우는 거의 없습니다.

bank를 한번 예로 들어보죠. bank가 ‘은행’일 수도 있지만 ‘강둑’이라는 뜻으로 쓰일 수도 있습니다.

그러니까 결국에 word meaning, sense는 언어의 뜻을 얘기하는 거고, 이게 과연 어떤 언어의 뜻으로 활용됐는지를 얘기하는 거고요.

그래서 첫 번째로 살펴볼 게 Homonymy입니다. 이건 동형이의어라고 말씀드릴 수 있는데요. bank가 대표적인 예입니다.

여기서는 bank라는 하나의 단어가 두 가지 서로 다른 뜻으로 해석된다고 보기보다는

bank가 원래 이런 근원의 단어가 있었고 이런 근원의 단어가 있었는데, 이 둘이 어떻게 보니까 알파벳이 같더라. 이렇게 보시는 게 더 타당합니다.

그러니까 단어의 유래도 완전히 다르고 뜻도 완전히 다른, 그러니까 형태는 똑같지만 뜻은 완전히 다른 그런 것이 Homonymy가 되겠고요.

이건 자연어 처리 분석에 대해서 여러 가지 어려운 점이 있습니다. 그래서 bat care라고 하면 bat이 ‘방망이’를 얘기하는 건지 혹은 ‘박쥐’를 얘기하는 건지 되게 어렵겠죠.

그래서 그거에 따라서 번역도 되게 어려워질 수 있겠고요. 그러니까 bat를 동물로 번역할지 혹은 ‘방망이’로 번역할지.

그리고 또 하나의 예는 Text-to-Speech, 그러니까 단어로부터 말을 만들어낼 때도 어렵습니다.

똑같은 bass지만 [베이스]라고 얘기해야 될 필요가 있고요. 혹은 [바스]라고 얘기해야 될 때가 있습니다.

그러니까 지금 말씀드린 내용은 이 단어들은 하나의 단어가 여러 뜻을 가지고 있다고 해석하기보다는

독립된 단어들이 존재했는데 이 독립된 단어들이 어떻게 보니까 알파벳이 같더라. 그걸 Homonymy라고 한다고 보시면 되겠습니다.

그리고 Polysemy는 말 그대로 단어가 여러 뜻을 갖는 경우입니다. 하나의 단어가 여러 뜻을 갖는 경우.

좀 전에도 말씀드렸다시피 영어 단어 중에 딱 하나의 뜻만 가지는 경우는 거의 없습니다. 모든 단어가 여러 가지 뜻을 가지고 있죠.

그냥 단적인 예로 보시면, 영어에서 그런 경우가 좀 많은데요. Jane Austen이라고 쓰면 이게 저자를 얘기할 수도 있고 혹은 그 저자의 작품을 얘기할 수도 있습니다.

예를 들면 저희가 어떤 지명을 얘기할 때 말 그대로 지명일 수도 있고 그 지명을 가진 프로 팀을 얘기하는 걸 수도 있고요.

그런 식으로 우리가 일반적으로 사전을 찾았을 때 여러 뜻이 있는 경우를 제외하고 일반적인 단어임에도 불구하고 여러 뜻으로 쓰이는 경우가 되게 많다는 겁니다.

그리고 Synonyms이라는 건 동의어입니다. 그러니까 같은 뜻을 가진 다른 말이죠. couch랑 sofa는 동의어가 되겠죠. 이런 식으로 되어 있는 걸 찾아내는 게 쉽지 않은 문제라는 얘기고요.

그런데 또 한편으로는 뉘앙스라는 게 달라질 수 있습니다. 두 단어가 동의어이긴 한데 이럴 때는 동의어인데 또 다른 경우에는 동의어를 안 쓰고 뉘앙스가 좀 달라지는 경우가 있습니다.

그래서 첫 번째 예제를 보시면, big이랑 large가 많은 경우에 같이 쓰일 수 있습니다. How big, How large is that plane? 이렇게 쓰일 때 big이랑 large는 똑같은 거죠.

똑같은 경우지만 만약에 sister랑 같이 사용을 하면, 그 sister가 언니인지 아니면 나보다 그냥 큰 자매인지에 따라서 달라지죠.

그래서 똑같은 동의어라도 이럴 때는 동의어가 되고 이럴 때는 동의어가 아닐 수도 있다. 그게 분석이 어려운 이유 중에 하나가 되겠고요.

또 다른 것은 반대말입니다. 그러니까 정확히 반대의 의미를 가지는 경우죠. long/short, fast/slow 이런 것들이 다 반대어가 되겠고요.

그런데 또 재미있는 특징은 문법적으로는 얘네들이 똑같습니다. 그래서 long을 써야 될 때 short를 쓰면 그게 문제가 전혀 되지 않죠.

그런데 문법적으로 완전히 같은데 단어만 반대인 경우가 많다. 그게 재미있는 현상이라고 할 수 있겠습니다.

그리고 또 다른 예제는 기계번역입니다. 기계번역은 되게 오랫동안 자연어 처리에서 연구된 분야인데, 근래에 기계번역의 아주 놀라운 발전이 있었습니다. 그리고 그 발전은 딥러닝에 기반했고요.

사람 입장에서는 기계번역이라는 게 되게 어렵습니다. 그래서 여러분이 영어로 작문을 한다든지 영어 문장을 한글로 쓴다든지 그게 되게 어려운 작업인데, 상대적으로 기계는 쉽게 해결하고 있고요.

요즈음 여러분이 많이 쓰시는 구글 번역기 같은 경우에는 구글 번역기의 모델 이름이 GNMT라고 불립니다.

그러니까 Google Neural Machine Translation system이라고 불리고요. 이건 2016년도 9월에 발표가 되었고요.

Neural이라는 단어가 붙은 이유는 이게 기본적으로 깊은 신경망, 그러니까 근래에 사람들이 얘기하는 딥러닝 모델에 기반하고 있다고 보시면 되겠습니다.

여기 예제는 중국어 문장이 들어왔을 때 기존 방법으로 하면 파랑색처럼 보이는데,

자신들이 제안한 방법으로 했더니 녹색 예제처럼 나왔고, 실제 사람보고 번역하라고 시켰더니 구글 번역기 성능이 더 좋았다. 이걸 예제로 보여드린 거고요.

그리고 뒤에 그림은 이 사람들이 구글에서 제안한 GNMT 모델의 개략적인 모습입니다.

여기서는 너무 깊은 얘기이긴 하지만 기본적으로 Sequence-to-Sequence라는 모델을 기반으로 했고,

attention mechanism을 활용해서 아주 놀라운 성능의 기계 번역기가 나오기 시작했다고 이해하시면 되겠습니다.

혹시 이 알고리즘에 관심이 많으시면 이걸 구현한, 그러니까 이 시스템이 논문으로 출판이 되었고 그 논문을 기반으로 해서 많은 사람이 구현을 했습니다.

그래서 구현된 코드를 받아서 여러분이 한번 테스트를 해보실 수도 있습니다. 그러기 위해서는 딥러닝에 대한 공부가 필요하다고 할 수 있겠죠.

그리고 또 다른 아직도 어려운 문제 중에 하나가 요약입니다. 문서의 요약입니다.

그러니까 주어진 문서가 있을 때 제 의도대로, 유저가 원하는 방향대로 짧게 만들어주는 게 Text Summarization이고요.

많은 경우에 Single-document summarization이 필요한 경우가 있습니다. 아주 긴 기사가 있을 때 그걸 한 문장으로 요약한다든지. 그게 Single-document summarization이라고 할 수 있고요.

그리고 또 이것보다 더 어려운 문제는 Multiple-document summarization입니다. 그러니까 여러 문서를 참조해서 요약을 하는 것이죠.

이건 일종의 여러분이 리포트를 쓰는 상황을 보시면 되겠습니다.

리포트 주제가 ‘간디의 일생에 대해서 한번 써라. 요약을 해봐라.’ 이런 식의 문제가 주어졌다고 하면,

여러분이 하나의 텍스트 소스만 보는 게 아니라 여러 텍스트를 봐서 그분의 일대기를 쓰게 되겠죠. 그런 경우에는 훨씬 어렵다고 할 수 있겠고요.

그래서 사실에 대한 요약뿐만 아니라 많은 경우에는 스토리 진행 상태에 대한 요약이 필요할 수 있습니다.

예를 들면 북한과 남한이 여러 평화를 위한 일들을 진행하고 있는데, 올해 초부터 나온 기사부터 시작해서 지금까지 나온 기사를 스토리별로 요약을 하자.

그동안 무슨 일이 있었는지. 그런 것도 Multiple-document summarization의 예제라고 할 수 있겠습니다.

실제로 Text Summarization은 여러 가지 형태로 분류할 수 있겠고요.

Generic, 일반적인 요약은 그냥 말 그대로 document가 주어지면 그걸 대표하는 문장 하나 혹은 5개 미만의 문장으로 요약하는 것이 될 수 있고요.

그다음에 Query-focused summarization은 유저가 어떤 질의를 주면 그거에 기반해서 요약하는 것입니다.

영화를 요약할 때도 인물 중심으로 요약할 수도 있고 아니면 특정 사건 중심으로 요약할 수도 있고 아니면 인물 간의 관계 혹은 특정 인물 관점에서의 요약.

이런 식으로 여러 가지 summarization을 할 수 있겠죠. 이 summarization이랑 질의응답 task랑은 아주 밀접한 관계가 있는 경우가 많습니다.

그리고 이 두 가지는 summarization을 어떻게 수행하느냐에 따라 두 가지 방식으로 나뉩니다.

Extractive 방식이라는 건 뭐냐 하면, 우리가 어떤 문서를 summarization 할 때 주어진 문서 중에 특정 구 혹은 단어를 뽑아가면서 요약하는 거고요.

Abstractive summarization은 전체 글을 다 이해한 후에 그 글을 하나로 표현하는, 요약하는 방법을 Abstractive summarization이라고 할 수 있겠죠.

이 두 가지 summarization task 중에서 두 번째 것이 더 어렵습니다.

그러니까 첫 번째 거는 각 문장의 중요도를 여러분이 파악한 다음에 가장 중요하다고 생각되는 문장을 뽑아내는 식으로 요약할 수도 있겠지만,

Abstractive summarization 방법은 기본적으로 글을 다 이해하고, 그 글에서 사용된 단어뿐만 아니라 새로운 표현 단어를 활용해서 요약하는 과정을 Abstractive summarization이라고 합니다.

여러분이 실제 환경에서 Text Summarization이 활용되는 예제를 보실 수 있는데요.

Snippets라고 해서 여러분이 만약 구글 검색을 하다 보면, 주어진 키워드에 대해서 여러 가지 document가 검색됩니다.

그런데 각각의 문서 링크 외에 그 밑에 조그맣게 문서의 주어진 query에 대한 정보가 어떻게 요약되어 있는지 어떻게 나타나 있는지를 짧게 요약한 부분이 많이 존재하는데요.

그런 것도 다 자동적으로, 그러니까 원래 텍스트를 요약해서 return 하는 경우라고 할 수 있겠죠.

많은 경우에 어떤 문서로 가면 그 문서에 두꺼운 글씨로 쓰여 있는 경우라든지 혹은 제목, 소제목 혹은 그런 게 전혀 없다고 하면 첫 번째 문장 혹은 마지막 문장에 좋은 요약이 되는 경우가 많죠.

그런 여러 가지 룰들을 활용해서 특정 웹페이지를 요약한 걸 보여드리고 있다고 보시면 되겠습니다.

오늘 이 시간에는 자연어 처리 기초에 대해서 말씀드리고, 그게 왜 어려운지 말씀드렸고 그리고 자연어 처리에 여러 task들이 있다고 하나하나씩 말씀을 드렸습니다.

어떻게 보면 넓게 소개한 강좌라고 할 수 있고요. 지금 하나하나씩 소제목으로 달려 있는 task, 작업들은 그 자체로도 나의 소분야이고요.

그리고 그 분야를 해결하기 위해서 정말 수많은 논문들이 출판되어 있고 책들이 존재하게 됩니다.

오늘 말씀드린 내용은 여러분에게 자연어 처리라는 학문을 소개시켜드리고 여러분의 흥미를 끌게 하기 위해서 구성이 되었고요.

혹시라도 자연어 처리에 더 관심이 많으시다면, 자연어 처리에 특화된 수업이라든지 혹은 그 자료를 더 공부하시면 더 많은 이해를 얻을 수 있을 것 같습니다.

그러면 오늘 강의는 여기서 마치도록 하겠습니다.

8.1.

안녕하세요? 서울대학교 컴퓨터공학부의 김건희입니다. 오늘은 인공지능의 기초 마지막 시간이고요. 오늘의 주제는 Computer Vision입니다.

사실 처음에도 말씀드렸다시피 인공지능은 매우 큰 분야이기 때문에 8개의 강좌로 충분히 다 말씀드리기에는 시간적인 한계가 있긴 합니다.

그래도 저 개인적인 입장에서는 중요하다고 생각되는 주제들 위주로 말씀드렸고요.

각각이 말씀드린 대로 하나의 매우 큰 분야이기 때문에 이 강좌를 통해서 여러분이 인공지능을 단순히 일반 사람들이 이해하듯 막연하게 생각하지 말고

‘인공지능이 이러이러한 분야가 있고, 이런 것들의 소개 강좌를 들어봤더니 더 관심이 생기더라. 그래서 더 알아보고 싶다.’ 이런 기회가 됐으면 합니다.

마지막 차수의 주제는 Computer Vision입니다. Computer Vision은 인공지능 중에서 시각 지능에 대한 내용입니다.

그러니까 사람은 눈으로 사물을 이해하고 그리고 우리가 원하는 곳까지 안전하게 이동할 수가 있는데요.

‘그와 같은 능력을 기계도 한번 가질 수 있도록 해보자’ 이게 Computer Vision의 가장 중요한 목적이라고 할 수 있겠고요.

그리고 저 개인적으로도 인공지능 분야 중에서 Computer Vision에 가장 많은 연구를 하고 있기 때문에 저 개인적으로도 매우 친숙한 분야이기도 합니다.

그럼 Computer Vision에 대해서 소개를 시켜드리도록 하겠습니다.

그러면 첫 번째로 과연 Computer Vision이 무엇인지에 대해서 알아보도록 하겠습니다.

사실 Computer Vision의 정의는 여러 가지가 가능할 텐데요. 제가 개인적으로 가장 만족스러운 정의를 여러분께 소개시켜드리고자 합니다.

여기 보시다시피 Computer Vision이란 결국에 우리가 사람으로서 눈을 가지고 할 수 있는 정말 많은 일이 있는데, 그 일들을 기계 혹은 컴퓨터에게 가르쳐주는 학문이라고 보시면 되겠습니다.

우리가 시각지능을 갖추어서 여러 가지 일을 하는데, 그 시각지능의 수많은 작업들을 기계도 역시 할 수 있도록 가르치는 것이 Computer Vision의 목적 그리고 그 정의라고 이해하시면 되겠습니다.

Computer Vision이란 학문 자체가 사람의 시각 능력을 모사하기 위해서 시작됐다고 할 수 있기 때문에 과연 사람의 비전, 사람의 시각 능력이 어떤지 한번 살펴보도록 하죠.

여기 몇 가지 리스트 업을 했다시피 우리는 시각 능력을 가짐으로써 정말 많은 그리고 놀라운 일들을 할 수 있습니다.

우리가 주변의 사람도 알아볼 수가 있고요. 그리고 여러 가지 물체도 인식할 수 있고요. 또 그 물체들 중에 내 거, 남의 거 이런 식으로 속성까지도 다 알 수 있습니다.

또 하나 중요한 특징 중에 하나는 제가 걸어서 강의실까지 오는 것도 제 눈이 있어서 가능한 일이었고요.

그래서 여러분이 시각 능력을 갖춤으로써 주변 환경에 수많은 장애물들이 있지만, 거기에 부딪히지 않고 안전하게 여러분이 원하는 곳으로 이동할 수도 있습니다.

그리고 세 번째 내용은 어떻게 보면 말로 설명하기 힘든 우리의 놀라운 능력 중에 하나라고 할 수 있는데요.

우리는 주변 환경을 보고 분위기도 이해할 수 있습니다. 그러니까 지금 내가 어떤 말을 하고 싶을 때 지금 내가 말을 해도 되는 분위기인지 아닌지도 시각 능력을 통해서 할 수가 있습니다.

그리고 또 하나 놀라운 능력은 여러분에게 사진 한 장만 준다고 하더라도 여러분은 그 사진으로부터 정말 긴 이야기를 만들어낼 수 있습니다.

이 모든 것이 다 우리가 시각 능력을 갖추어서 사진을 보고 사진을 다 이해할 수 있기 때문에 가능합니다.

하지만 사람의 시각 능력은 완벽하다고 할 수 없습니다. 그래서 몇 가지 단점에 대해서 말씀드리면, 첫 번째 단점은 우리는 착시현상을 겪는다는 것입니다.

그거에 대한 몇 가지 예는 다음 슬라이드에서 말씀드리도록 하겠습니다.

그리고 사람은 어떤 사진을 볼 때 전체에서 가장 중요한 내용을 빠르게 알아내는 능력은 있지만, 세부 디테일을 자세히 보거나 혹은 그걸 기억하는 능력은 매우 떨어집니다.

그리고 우리는 주변 상황을 어떻게 보면 내 식대로 보고, 내 식대로 이해하고, 내 식대로 설명을 한다고 볼 수 있고요.

그래서 여러분이 수많은 분들에게 사진 한 장을 주고 그것에 대해서 한번 기술해보라고 요청을 한다면, 아마 100명이 다 다른 문장을 써낼 것입니다.

그리고 또 하나의 단점이라고 할 수 있는 건 우리는 주변 환경을 잘 이해하긴 하지만, 주변 환경을 정확하게 이해하지는 못합니다.

그러니까 제가 어느 방에 들어가서 그 방의 사이즈가 과연 천장이 1.5m인지 1.6m인지 이런 건 전혀 알 수가 없고요. 그리고 어떤 물건이 주어져도 그거의 정확한 크기, 길이는 잘 알기가 어렵습니다.

그리고 또 하나는 우리의 기억력이라는 게 매우 제한이 돼 있어서 아마 여러분에게 한 10장의 사진을 보여드린다고 하면,

첫 번째 사진이 무엇이었는지를 아주 자세히 기억하는 건 매우 어려운 일이 될 것입니다.

그래서 지금 말씀드린 Human Vision의 단점이라고 보여드린 내용에 대해서는 오히려 Computer Vision이 훨씬 더 잘하는 분야입니다.

Computer Vision은 착시현상도 겪지 않고요. 그리고 Computer Vision은 정확하게 환경을 측정할 수도 있는 능력을 가지고 있습니다.

한편으로 보면 인공지능의 좋은 메시지라고 할 수 있는데요.

인공지능이 단순히 사람을 대체하기 위해서 개발된다기보다는 기계가 잘할 수 있는 분야와 사람이 잘할 수 있는 분야는 많은 경우에 다릅니다.

기계와 사람이 결국에는 공존하는 그런 사회가 인공지능의 목적이라고 이해하실 수 있겠습니다.

그럼 Illusion의 몇 가지 예에 대해서 한번 보여드리도록 하겠습니다.

여러분, 왼쪽의 파란색 정육면체를 계속 보고 있으면 두 가지 정육면체를 보게 됩니다.

첫 번째 정육면체는 지금 보여드린 이 면이 앞으로 보이는 정육면체이고, 또 하나의 예는 이 두 번째 면이 앞면으로 보이는 정육면체입니다.

그래서 이 그림을 계속 유심히 보고 있으면 처음에는 한쪽 면을 정면으로 보고 있다가 어느 순간에 다른 쪽 면을 정면으로 보는 직육면체로 이해가 되고요.

그게 갑자기 왔다 갔다 한다. 그런 착시현상을 여기서 보여주고 있습니다.

또 다른 착시현상은 매우 잘 알려진 착시현상입니다. 여기 보시면 체커 보드가 있고요. 그 위에 원 기둥이 있고요. 그리고 빛 반대쪽으로 그림자가 있습니다.

그래서 여러분이 A라는 영역과 B라는 영역을 보시면, B라는 영역이 훨씬 밝아 보일 것입니다.

하지만 우리가 A, B를 연결하는 회색 바(bar)를 그리게 된다면 A와 B라는 영역이 똑같은 밝기의 영역이라는 걸 여러분이 아실 수 있습니다.

그런데 이런 착시현상이 왜 일어날까요? 사람의 뇌는 차이를 인식하도록 학습되었습니다. 그러니까 어떤 주어진 영역의 밝기가 정확히 어느 정도의 밝기인지를 인식하도록 설계되어 있지 않고요.

이 영역의 밝기가 주변에 비해서 얼마나 밝은지 혹은 얼마나 어두운지를 잘 인식하도록 학습이 되었습니다.

그래서 A 같은 경우에는 주변에 밝은 영역이 있기 때문에 얘가 어두워 보이는 거고, B 같은 경우에 주변이 어두운 영역이기 때문에 더 어둡게 보이는 것입니다.

그러니까 A와 B의 절대적인 밝기는 사람은 잘 특정해내지 못하고 인지해내지 못한다. 이게 이 착시현상에서 보여주는 결론입니다.

Computer Vision에 대해서 본격적으로 말씀을 드리고자 하는데요. Computer Vision이 참 어렵습니다.

왜 어려운지 말씀드리면, 여러분이 왼쪽을 보면 이 사진을 보는 순간 아주 짧은 찰나에 ‘여기 사람 눈, 코, 입이 있고, 사람 얼굴이 있고, 머리가 있고, 어깨도 있구나.

그리고 이 사람은 내가 아는 사람이구나. 아인슈타인이라는 물리학자구나.’ 이걸 정말 보는 순간 알게 됩니다.

그런데 이 사진이라는 게 컴퓨터에는 어떻게 저장이 되어 있느냐 하면, 디지털 이미지 형태로 저장되어 있고요.

디지털 이미지는 해상도라는 게 있습니다. 만약에 이 이미지의 해상도가 600×400 해상도 이미지라고 하면, 그거는 곧 600개의 행과 400개의 열로 이루어진 그리드고요.

그리고 그 각각을 픽셀이라고 부르는데, 그 픽셀은 0~255 사이의 정수 값으로 주어져 있습니다.

그래서 여기서는 gray scale, 흑백 이미지이기 때문에 각 픽셀마다 하나의 숫자가 주어져 있고요.

컬러 이미지 같은 경우에 각 픽셀마다 RGB, 그러니까 빨강, 녹색, 파랑색 3개의 채널에 대해서 각각 0~255 사이의 정수 값으로 구성되어 있다고 보시면 되겠습니다.

사람 입장에서는 이게 너무 당연한 이미지인데, 컴퓨터가 결국에 보는 건 엄청나게 큰 숫자 array일 뿐입니다.

그 숫자 array로부터 ‘이 부분에 눈이 있고, 이 부분에 코가 있고, 이 전체가 사람 얼굴이다.’

이걸 기계한테 가르쳐준다는 게 쉽지만은 않은 일이라는 걸 여러분도 어느 정도 짐작하실 수 있을 것 같습니다.

Computer Vision이 결국에 무엇인지를 요약하는 슬라이드입니다. 그래서 여러분에게 몇 가지 Computer Vision의 정의를 여기에 나열했는데요.

첫 번째 정의는 제가 맨 첫 슬라이드에 소개시켜드린 그 정의입니다. ‘사람으로서 우리가 눈으로 할 수 있는 모든 능력을 기계에게 가르쳐주자.’ 그게 목적이고요.

또 하나의 정의라고 하면 사진을 지능적으로 해석할 수 있는 능력을 Computer Vision이라고 볼 수도 있습니다.

여기서 imagery라고 쓴 이유는 image의 통칭, 여러 종류의 이미지를 다 합쳐서 말씀드리는 것입니다.

우리가 일반적으로 보는 RGB 이미지뿐만 아니라, depth를 측정하는, 거리를 측정하는 이미지도 있을 수 있고요. 여러 형태. 레이저 스캐너로부터 얻어진 이미지들도 있을 수 있고요.

그러니까 그런 모든 이미지 형태에 대해서 지능적으로 이해하는 학문이라고 보실 수가 있고요.

그리고 세 번째 정의는 인공지능이랑 연결돼 있는 정의입니다. 그러니까 대뇌에서 시각 부분을 담당하는 부분을 Visual Cortex라고 합니다.

그러니까 Visual Cortex, 우리 대뇌에서 시각 영역을 담당하는 Visual Cortex을 인공적으로 한번 만들어보자. 그런 학문이라고도 정의할 수 있겠습니다.

그리고 네 번째 정의에서 optics라는 건 광학이고요. 광학이라는 건 빛에 대한 학문을 연구하는 것입니다.

그래서 우리가 실제로 어떤 object, 물체를 보는 것은 조명이 있고, 광원이 존재하고, 그 광원으로부터 빛의 입자가 나가서 물체 표면이 반사가 되고,

그 반사된 빛을 우리 망막이 측정해내기 때문에 빛을 보는 것입니다. 그러니까 이 과정 자체가 광학이라고 할 수 있겠죠.

그런데 Inverse optics라는 건 광학의 반대 방향이라는 겁니다.

그러니까 빛과 물체가 있어서 이미지를 얻는 게 광학이라고 한다면, 우리는 이미지가 주어졌을 때 빛이 어디에 존재하고 환경이 어떻게 구성돼 있는지,

그러니까 반대 방향을 알아내는 과정이라고 보시면 되겠습니다.

여러분이 어떤 정의를 좋아하시든지 간에 Computer Vision은 참 어려운 학문이고요. 대신에 우리가 실제로 보는 것을 다루는 학문이기 때문에 재미는 매우 높다고 할 수 있겠습니다.

그래서 Computer Vision에 대한 여러 가지 작업들에 대해서 말씀을 드리도록 하겠습니다. Computer Vision 작업들은 크게 보면 3가지 레벨로 표현이 가능한데요.

Low-level Computer Vision, Mid-level Computer Vision. 이런 식으로 Computer Vision 연구하시는 분들은 얘기를 합니다.

그래서 Low-level Computer Vision이라는 건 주어진 input도 이미지고 output도 이미지인 경우.

Mid-level 컴퓨터 Vision task라는 건 이미지가 주어졌을 때 output으로 이미지를 얻어내는 게 아니라, 어떤 특정 정보를 나타내는 feature, 특징점을 알아내는 걸 Mid-level 비전이라고 하고요.

High-level은 이미지로부터 어떤 의미를 알아내는 걸 High-level 비전이라고 합니다. 그럼 다음 슬라이드부터 하나씩 예제와 함께 설명해드리도록 하겠습니다.

Low-level 비전의 예입니다. 좀 전에 말씀드렸다시피 Low-level Vision task는 input으로 이미지가 주어졌을 때 output은 또 다른 이미지, output이 됩니다.

첫 번째 예제는 Deblurring입니다. blurring이라는 건 흐릿하다는 뜻이고요. 그러니까 Deblurring이라는 건 흐릿한 부분을 없애서 선명한 이미지로 바꿔준다는 것입니다.

실제 우리가 왜 blurring된 사진을 찍느냐 하면, 저희가 어떤 카메라 셔터를 누르는 순간 우리는 인지하지 못하지만 카메라가 아주 작게 혹은 빠르게 흔들리게 됩니다.

그러면 CCD 카메라의 한 픽셀에 환경의 하나의 포인트가 매핑이 되는 게 아니라 여러 포인트가 CCD 카메라의 한 픽셀에 기록이 되고요.

그러니까 어떻게 보면 각 픽셀마다 환경의 여러 포인트에 대한 평균 정보가 기록됐기 때문에 Deblurring이 발생한다고 이해하시면 되겠고요.

Edge detection에서 Edge라는 것은 이미지를 보시면 어떤 특정 영역은 밝기가 급격히 변하는 부분이 있습니다.

그러니까 이 사진에서는 모자를 따라서 배경과 모자와 경계 부분에 아주 급격하게 밝기가 변하게 되는데, 밝기 변화를 찾아내는 걸 Edge detection이라고 하고요.

그리고 Super-resolution은 뭐냐 하면, 저해상도 이미지를 고해상도 이미지로 바꾸는 것입니다.

여러분이 600×300 이미지를 단순히 확대만 하면 되게 부자연스러운 사진이 나올 것입니다. 좀 더 선명하면서도 자세한 사진을 얻는 것이 Super-resolution이고요.

마지막 예제로 Colorization은 흑백 이미지가 주어지면 그걸 컬러 이미지로 바꿔주는 것입니다.

그래서 이 네 가지 task를 보면 왼쪽이 input이고 오른쪽이 output입니다. 그러니까 이미지가 들어가서 우리가 원하는 더 나은 이미지를 만들어내는 과정이라고 이해하실 수 있겠습니다.

이제 Mid-level Vision task에 대해서 알아보도록 하죠. Mid-level task라는 건 input은 이미지고 output은 단순히 이미지가 아니라 어떤 특징을 가지고 있는 feature가 되겠습니다.

첫 번째 예제는 Boundary detection입니다. Boundary detection이라는 건 경계를 찾아내는 거고요.

좀 전에 살펴본 Edge detection랑 차이를 보시면, Edge는 말 그대로 사진의 밝기가 급격히 변하는 부분이라고 하면, Boundary는 object의 경계라고 할 수 있겠죠.

그래서 도마뱀 표면을 보면 여러 무늬가 있고요. 그 무늬별로 Edge는 detection이 되겠지만, Boundary 같은 경우에 무늬에 대해서는 Boundary가 나오면 안 되겠죠. 그래서 그런 차이가 있고요.

Segmentation은 이미지를 동일한 영역으로 분할해주는 과정을 얘기합니다. 사람 바지 혹은 머리별로 나눠주는 것이고요.

Shape-from-shading은 뭐냐 하면, 사람은 그림자가 보이는 정보로부터 3차원 정보를 알아냅니다.

지금 보여드린 예제를 보시면 오른쪽은 볼록하다고 느끼고 왼쪽은 오목하다고 느낍니다.

그런데 실제로 이건 평면의 사진일 뿐인데 사람은 shading, 그러니까 그림자로부터 shape, 모양을 알아낼 수 있다.

모양 자체가 단순히 이미지가 아니라 그 이미지보다 한 단계 더 나간 feature라고 보실 수 있고요.

또 다른 task는 Alignment입니다. 그러니까 지금 두 이미지에 똑같이 나비가 있지만 나비가 다른 방향에서 찍힌 사진입니다.

어떤 영역이 어떤 영역과 서로 매칭이 되는지를 Align 하는 과정을 Alignment라고 하고요. 그건 단순히 이미지가 output이 아니라, 매칭된 포인트들이 feature로 나오게 됩니다.

그리고 High-level Vision task는 이미지가 들어오면 그거에 대한 의미를 파악하는 것입니다.

첫 번째 보여드린 예제는 Image classification이고요. 첫 번째 이미지가 주어지면 ‘이게 진드기구나.’ 진드기라는 의미로 해석을 해내는 것이죠.

Object detection이라는 건 무슨 물체가 있는지 단순히 이미지를 분류하는 게 아니라, 우선 우리가 관심 있는 물체가 이미지 상 어디에 존재하는지 사각형으로 영역을 지정하고요.

그 영역에 어떤 object가 있는지를 찾아내는 과정입니다. 그러니까 위치까지도 같이 알아야 될 때는 더 어려운 문제죠.

그리고 Image captioning은 근래에 많이 연구되는 분야인데, 이미지가 주어지면 그걸 설명하는 문장을 만드는 것입니다.

Image classification은 이미지를 명사 하나로 변환한다고 하면, Image captioning은 이미지 하나를 문장 하나로 번역을 한다고 보시면 되겠고요.

Pose detection은 우리가 많은 경우에 사진에서 제일 중요한 건 사람입니다. 여러분도 사진 찍을 때 인물 사진을 많이 찍게 되는데요.

주어진 사진에서 사람이 있는 경우에 그 사람이 어떤 자세로 있는지 알아내는 것을 Pose detection이라고 합니다.

지금까지 Low-level부터 High-level 비전까지 여러 예제를 보여드렸는데, 꼭 이거에만 국한되는 게 아니라 아주 다양한 문제가 존재합니다.

그러면 Computer Vision이랑 아주 연관된 분야인 Image Processing, 그러니까 영상처리와의 비교를 말씀드리도록 하겠습니다.

영상처리는 이미지에서 이미지로 변환하는 과정을 주로 다룹니다. 좀 전에 살펴드린 대로 input도 이미지고 output도 이미지이기 때문에 Low-level Vision task와 아주 관련이 깊습니다.

대표적인 작업으로는 이미지 압축, 복원, 개선하는 그런 것들이 있겠고요.

Computer Vision 같은 경우에 Image Processing의 테크닉들을 아주 많이 사용하긴 하지만,

많은 경우에 이미지 분석에 대한 output이 이미지가 아니라 좀 더 그것보다 High-level의 지능적인 분석을 많이 다루게 됩니다. 그런 측면에서 두 학문에 차이가 있다고 보실 수 있겠습니다.

이 그림은 Computer Vision이라는 작업이 아주 Low-level부터 High-level까지 다양한 task로 구성되어 있다고 보실 수가 있고요.

가장 낮은 영역에서는 Image Formation, 그러니까 카메라가 어떻게 동작하는지부터 시작해서 Low-level Vision task.

컬러라든지 혹은 동영상일 경우에 어떤 픽셀이 다음 프레임에서 어디로 움직였는지 그런 모션 정보까지 분석하는 것부터 시작해서 점점 Mid-level task, High-level task까지 있고요.

High-level task가 끝나면 다른 인공지능의 작업들과 연결 관계를 가지게 됩니다.

그래서 아주 낮은 레벨의 광학부터 시작해서 의미론, 다른 인공지능과의 연관관계. 그렇게 아주 다양하게 여러 hierarchy를 가지고 Computer Vision task를 정의할 수 있다고 보실 수 있겠습니다.

그리고 또 하나 중요한 Computer Vision의 특징은 아주 다학제적인 분야라는 것입니다.

여러분이 컴퓨터공학을 전공해도 Computer Vision을 할 수 있지만, 여러분이 Neruo-Science를 전공한다고 하더라도 Human Vision 관점에서 Computer Vision을 해석할 수 있고요.

그리고 여러분이 로봇을 연구한다고 하면 로봇의 많은 경우에 Computer Vision과 함께 문제를 풀어야 되는 경우도 많고요.

그리고 광학을 한다면 물리학도 연관이 되고요. 그리고 요즘 많이들 관심 있어 하시는 기계학습 영역과도 아주 밀접한 관계가 있습니다.

그래서 근래에 Computer Vision 연구는 기계학습 기반으로 많이 이루어지고 있습니다.

이건 어떻게 보면 여러분에게 좋은 뉴스입니다. 여러분이 어떤 Background를 가지고 있든지 간에 여러분 관점에서 컴퓨터 비전을 해보실 수가 있다고 보실 수 있겠습니다.

그러면 지금부터는 Computer Vision의 소개를 마치고요. Computer Vision이 왜 어려운지.

제가 누누이 강조한 내용이 ‘Computer Vision이 재미는 있지만 어렵다.’ 이렇게 말씀을 드렸는데, 왜 어려운지에 대해서 하나하나씩 말씀을 드리도록 하겠습니다.

첫 번째 어려운 이유는 우리 환경은 3차원인데 사진이라는 건 3차원의 환경을 2차원으로, 어떻게 보면 정보의 손실을 감수하고 표현을 변화한 것입니다.

그래서 하나의 3차원 물체가 있을 때 여러분이 어느 방향에서 보느냐에 따라서 사진의 결과가 매우 달라집니다.

이건 재미있는 예로 미켈란질로의 유명한 작품인데, 심지어 사람에게 조차도 이 작품을 어떻게 찍었느냐에 따라서 이게 과연 똑같은 작품일까 의심이 될 정도로 매우 다르게 보입니다.

좀 전에 말씀드렸다시피 우리 환경은 3차원인데 그걸 사진으로 찍는 순간 2차원으로 변경함으로써 여러 가지 실제에서는 일어날 수 없는 현상이 사진 속에 이루어지게 됩니다.

위에서 보시면 어떤 이미지 상의 한 포인트는 카메라 센터로부터 그 이미지 plane의 포인트까지 직선을 그었을 때

그 직선에 해당하는 모든 포인트들이 그 이미지의 픽셀로 매핑이 될 수 있습니다.

그래서 occlusion, 가림 현상을 보시면, 이 사진에서 보시면 뒤에 건물 앞에 사람이 있음으로 해서 이 건물이 가려지게 되는데요.

그건 카메라 센터로부터 이미지 플레인까지 직선을 그었을 때 가장 먼저 만나는 환경 상의 지점이 사람의 다리였기 때문입니다.

그래서 제가 직선을 그으면 사람 다리를 거쳐서 건물까지 이어지지만, 그런데 사람 다리가 앞에 있음으로 해서 뒷부분이 보이지 않게 되는 것이죠.

이 사진 같은 경우에 마치 사람이 피사의 사탑을 발로 차는 것처럼 보이지만, 왜 이렇게 보이게 되는지 여러분이 약간이나마 이해하실 수 있을 것 같습니다.

그리고 오른쪽 이미지를 보시면, 사람은 매우 작고 호박이 엄청나게 큽니다. 그러면 사람은 이 사진을 보는 순간 ‘이렇게 큰 호박이 있구나.’ 이렇게 생각하지 않고요.

‘카메라의 아주 가까운 부분에 호박이 있고 사람은 멀리 있구나.’ 그러니까 멀리 있는 object는 조그맣게 보이고 가까이 있는 object는 크게 보입니다.

이것도 마찬가지로 3차원의 환경이 2차원으로 projection 되면서 벌어지는 현상이라고 보시면 되겠습니다.

두 번째로 어려운 점은 아까도 언뜻 말씀을 드렸지만, 우리가 어떤 물체를 본다는 것은 환경의 조명, 그러니까 광원이 존재했고요.

그 광원으로부터 나온 빛이 그 물체의 표면에서 반사를 일으켜 그 반사된 빛을 우리가 보기 때문입니다. 그래서 광원이 전혀 없으면 아무것도 우리는 볼 수 없게 되겠죠.

그래서 똑같은 사람이라고 하더라도 광원이 어디에 있느냐에 따라서 매우 다르게 보입니다. 이게 또 두 번째 Computer Vision 비전의 어려운 점이라고 할 수 있겠죠.

사진을 어떻게 보면 빛의 마술이라는 식으로 많이들 말씀하시는데, 그래서 이런 빛과 물체와의 interaction 때문에 아주 많은, 아주 이상한 여러 현상이 일어나게 됩니다.

왼쪽 그림을 보시면 이 부분은 모두 균일한 아스팔트라는 건 사람이 다 알고 있습니다. 그런데 이 부분과 이 부분을 비교해보시면 이 부분은 되게 밝고 이 부분은 되게 어둡습니다.

똑같은 영역임에도 불구하고 매우 다르게 보이는 거죠. 사람은 여기 나무가 보이지 않아도 ‘이게 그림자구나.’ 이걸 알 수 있지만,

기계는 이 사진을 보는 순간 ‘나무가 없는데 어떻게 그림자부터 보냐.’ 이렇게 이해할 수도 있다는 겁니다.

그리고 오른쪽 사진을 보면 이게 역광이 된 사진이기 때문에 역광에서 보면 물체는 항상 어둡게 보이고요.

왼쪽 그림과 마찬가지로 균일한 풀밭인데 어떤 부분은 어둡게, 어떤 부분은 밝게 보이게 되죠.

그리고 Illumination, 조명 조건으로 인해서 Computer Vision이 어려운 또 하나의 예는 이와 같습니다. 반사를 통해서 우리가 보는 물체와 현실이 다르다는 것이죠.

왼쪽 그림을 보시면, 이건 창문입니다. 여러분은 구름도 보실 수 있고, 나무도 보시고, 강도 보십니다.

이건 환경이 창문에 반사가 된 것인데, 기계 입장에서는 ‘여기에 나무가 있구나.’ 이렇게 이해합니다.

그러니까 창문을 보지 못하고 ‘창문에 비친 모습이다.’라고 이해하기보다는 ‘여기 나무가 있구나.’ 이렇게 생각하기 쉽다는 거고요.

그리고 오른쪽은 비가 오면 땅바닥에 중간 중간 물웅덩이가 있을 수 있고요. 그걸 통해서 환경이 반사가 될 수 있습니다.

그런데 여러분이 예를 들어서 자율주행차를 만드는데 자율주행 자동차의 사진기로부터 이런 사진을 얻었다. 그러면 갑자기 땅바닥에 나무가 보이는 겁니다.

그러면 자율주행 자동차가 갑자기 멈추게 될 텐데, 그럼 ‘이건 나무가 아니라 여기 물이 있었고 거기에 비친 거다.’ 이런 것들을 일일이 가르쳐준다는 게 참 어려운 일이라는 것입니다.

그리고 또 다른 여러 가지 물리적 광학 현상에 의해서 사진 해석이 어려운 점이 많은데요. 세 번째는 굴절입니다.

우리는 이 사진을 보는 순간 수영장 밑에 있는 타일은 정확한 사이즈로 규격화돼 있고 명확한 사각형이 균일하게 있다고 알지만, 이 사진은 모든 게 다 찌그러져 보이는 것이죠.

그리고 오른쪽 같은 경우에 사람의 경우에는 ‘이게 나뭇잎이고, 나뭇잎 뒤에 나뭇잎이 있다.’ 이런 것들을 알게 되겠지만,

반투명한 특성이 어디까지가 하나의 풀인지, 하나의 이파리인지 알아내는 게 매우 어렵게 됩니다.

그리고 그 외에 되게 다양한 자연현상이 있죠. 그러니까 안개가 낀다든지 비가 온다든지 혹은 눈이 내린다든지. 이 모든 게 다 연구 과제고요.

그래서 자율주행 자동차를 만들었는데 갑자기 눈이 많이 내려서 시야 확보가 안 될 때 어떻게 할 것이냐? 그런 어려움이 있고요.

그리고 실제로 비 오는 날에 찍은 이미지가 주어졌을 때 그 비 모습을 다 지워주는 기술조차도 제안이 되었습니다.

Computer Vision이라는 게 참 다양한 현상이 있고, 각각에 대해서 해결하기 위한 정말 다양한 연구가 진행되고 있다고 이해하실 수 있겠습니다.

그리고 세 번째 Challenge입니다. 세 번째 도전인데, 세 번째 도전인은 Occlusion, 그러니까 가림 현상이 있다는 것이죠.

좀 전에 왜 가림현상이 일어나는지 3차원의 세계를 2차원으로 projection 하면서 나타나는 현상이라고 말씀을 드렸는데요.

이 사진을 보는 순간 말 두 마리를 갈라져 있다고 생각하시는 분은 없을 겁니다. 사람에 의해서 말이 반으로 나눠진 것처럼 보인다.

이건 사진을 보는 순간 다 알 수 있는데, 이것조차도 컴퓨터에게 일일이 가르쳐 준다는 게 참 어렵다는 것이죠.

Occlusion이 왜 중요한지 단적으로 보여주는 사진입니다. 우리가 자율주행 자동차를 만든다고 할 때 보행자를 검출하는 것과 주변에 차를 인지하는 게 매우 중요한 작업입니다.

그런데 우리가 현실에서 사람이 돌아다니는 거리를 본다고 하면, 처음 머리부터 발끝까지 하나의 가려짐도 없이 완벽한 사람을 보는 경우는 매우 드물고요.

대다수의 사람은 어딘가가 가려져 있는 그런 사람들입니다. 그래서 이런 사진이 주어졌을 때 사람이 과연 몇 명 있느냐?

혹은 오른쪽 사진이 주어졌을 때 자동차가 몇 대 있느냐? 이걸 찾아낸다는 게 그렇게 쉬운 일은 아니겠죠.

그리고 이 부분이 아주 재미있는 부분인데, 사람은 아주 매우 작은 이 영역만 보고도 ‘거기에 차가 있구나.’를 알게 되는데,

만약에 기계가 그 부분만 봐서 이게 자동차처럼 생겼는지 아닌지로만 판단을 한다면 자동차라고 알 수가 없겠죠.

이게 자동차인지 알려면 도로라는 context가 있고, 버스는 되게 크고 그리고 버스의 이 정도 거리 뒤에 있는 뭔가의 물체는 자동차일 확률이 높구나.

이런 아주 복잡한 reasoning을 통해서 사람은 거기에 자동차가 있다는 걸 알게 됩니다.

네 번째 어려운 점은 Scale입니다. Scale이라는 건 똑같은 object라고 하더라도 이 이미지 상에서는 되게 크게 보일 수도 있고 아주 죽게 보일 수도 있다는 것입니다.

이 두 가지 object가 똑같은 object인지 알기가 어려울 수 있다는 의미고요. 예제와 함께 보도록 하겠습니다.

왼쪽 그림을 보시면 이건 일반적인 도로의 사진입니다. 그러면 저희가 이 도로의 사진 중에 두 가지 영역을 보도록 하죠. 이 두 가지 영역입니다.

우리가 이 사진을 보는 순간 이 영역에 사람이 있다는 건 아주 쉽게 알 수 있습니다. 그런데 저희가 만약에 사진만 놓고 봤을 때 이 영역을 잘라서 크게 확대하면 이와 같이 보입니다.

그래서 오른쪽 그림만 보고 이게 사람이구나, 아니구나 판단하는 건 너무 어려운 일이고요. 대신에 이렇게 작은 물체가 이미지 상에 존재하면 우리는 상대적으로 사람임을 쉽게 알 수 있습니다.

우리가 왜 사람인지 쉽게 알 수 있을까요? 우리는 context라는 정보를 활용합니다.

context라는 건 이 영역에 어떤 물체가 있는지를 판단하는 데 있어서 그 부분의 픽셀 값만 보는 게 아니라 주변 영역을 다 본다는 것이죠.

그래서 여기에 지금 땅바닥이 존재하고요. ground 플레인이 존재하고, 이 영역은 ground 플레인에 붙어 있다. 그러니까 걸어가는 사람일 것이다.

그리고 이 옆에 자동차가 하나 있는데 자동차와 사람 간에 상대적인 크기를 봤을 때 이 정도로 보이는 건 사람일 확률이 높고,

마찬가지로 옆에 큰 건물이 있는데 건물과 사람과의 크기 비율이 이 정도면 이 부분은 불명확하긴 하지만 사람이구나. 이런 걸 쉽게 알 수 있다는 겁니다.

그런데 기계에게 이런 것들을 일일이 가르쳐 주는 건 매우 힘들죠.

그러니까 여기에 사람이 있다는 걸 알기 위해서는 어디에 ground 플레인이 존재하고, 이 옆에 있는 게 자동차라는 걸 알아야 합니다.

그래서 많은 경우에 chicken and egg problem이 필요합니다.

그러니까 사람인지를 알려면 주변을 알아야 되고, 주변을 알려면 사람이 옆으로 걸어 다니는 상황이라는 걸 아는 게 매우 많이 도움이 되고요.

실제로 보행자 검출을 자율주행 자동차에서 한다고 할 때 사람이 명확하게 잘 보이는 건 거의 98% 이상으로 아주 정확하게 검출을 할 수 있습니다.

그런데 대다수의 실패 케이스는 언제냐 하면, 좀 전에 말씀드린 대로 occlusion이 매우 심하게 된 경우 혹은 매우 작게 보이는 경우입니다.

그리고 다섯 번째 어려운 점은 물체라는 게 하나의 형태만 가지고 있는 게 아니라 여러 가지 형태를 가지고 있다는 겁니다. 움직일 수 있고, 어렵게 deform이 가능하다는 거고요.

이 그림을 보시면, 지금 여기에 아주 많은 수의 말이 있습니다. 그런데 이 말들 중에 똑같이 보이는 말은 하나도 없고요. 똑같은 자세를 가진 말도 하나도 없습니다.

그러니까 강체라고 하죠? 자동차같이 딱딱한 물체 같은 경우에 shape, 그 모양이 정해져 있긴 하지만, 말 같은 경우에 관절이 있는 물체입니다.

그래서 관절이 있는 물체 같은 경우에 자기가 자기의 shape을 마음대로 바꿀 수가 있고요. 그리고 표면이 물렁물렁하기 때문에 deformation에 의해서 변화가 됩니다.

그리고 여섯 번째 Challenge는 Background, 그러니까 배경의 난잡함, 어려움 때문입니다. Background Clutter라고 많이 얘기하는데요.

여러분이 만약에 이미지 분류 작업을 한다고 할 때 무슨 쇼핑 사이트에서 상품 이미지, 깨끗이 찍힌 이미지를 분류하는 건 상대적으로 쉽지만,

만약에 그 물체가 일상 환경에 있었을 때 그걸 분류하는 건 어렵습니다.

왜냐하면, 일상 환경에서는 우리가 관심 있는 물체뿐만 아니라 여러 가지 다른 물체가 있기 때문에 그것 때문에 혼동을 얻게 된다는 것이죠.

그래서 지금 계속 예제와 함께 말씀을 드렸는데, 자율주행 자동차 예제를 한번 보도록 하죠.

이 사진은 대만의 한 거리에서 찍힌 사진입니다. 그리고 현재 자율주행 자동차가 가장 어려워하는 부분 중에 하나가 사인, 그러니까 신호등을 본다든지 아니면 교통 표지판을 보는 것입니다.

실제로 자동차에서 사진을 찍으면 신호등이 되게 조그맣게 보이고요. 여러 가지 사인 같은 것도 다 조그맣게 보입니다.

이 경우를 예로 들면, 지금 신호등이 파란 불이기 때문에 자동차가 갈 수 있는데, 보시다시피 신호등은 매우 조그맣게 보이고요.

그리고 배경에 수많은 Clutter들이 존재하기 때문에 과연 어디에 신호등이 있는지를 찾아내는 게 매우 어렵습니다.

그리고 이 특별한 예제에서는 간판 색깔로 빨간색이 많이 사용되고 있어서 만약 특정 영역을 신호등으로 잘못 인식하고 ‘여기 빨간 불이 있네?’라고 한다면, 자동차가 전혀 움직이지 못하겠죠.

그런 식으로 Background Clutter가 실제로 환경을 인식하는 데 있어서 매우 어려운 점이 됩니다.

그리고 마지막 Challenge에 대해서 말씀드리도록 하겠습니다. 이거는 무슨 뜻이냐 하면, 물체 내부의 Variation, 변화가 매우 크다는 것입니다.

여기에 지금 6가지의 사진이 있는데요. 이 6가지는 다 의자에 대한 사진입니다. 사람은 이것들을 보고 다 의자라고 생각하지만, 각각의 이미지, 의자의 겉모습은 매우 다릅니다.

사실 의자라는 물체를 정의하기 위해서는 ‘앉는다.’는 개념이 필요합니다.

그러니까 우리가 물체를 보고 ‘이건 의자다.’라고 생각하는 가장 큰 이유는 이 물체가 앉을 수 있느냐 없느냐를 따집니다.

그래서 이 사진을 보고 우리는 겉모습을 통해서 의자라는 걸 알아내는 게 아니라, 기능을 찾아보고 그걸 기반으로 ‘의자구나’ 이렇게 인식을 하게 됩니다.

그래서 이런 건 의자뿐만 아니라 되게 다양한 상황에서도 보일 수 있는데요.

여러분이 예를 들어 개와 고양이를 분류하고 싶다고 했을 때 개처럼 생긴 고양이도 되게 많고요. 그리고 고앙이처럼 생긴 개도 많습니다.

그래서 어떤 두 이미지는 둘 다 개인데 매우 다르게 생겼고, 이 개 이미지랑 고양이 이미지는 클래스가 다름에도 불구하고 되게 똑같이 보이는 그런 경우가 매우 많이 발생하게 됩니다.

지금까지 Computer Vision이 왜 어려운지 7개 Challenge에 대해서 하나하나씩 살펴봤고요. 어떻게 보면 너무 비관적인 말씀만 드린 것 같긴 한데요.

이러이러한 어려움이 있긴 하지만 근래에는 아주 많은 Computer Vision의 진보가 있어서 여러 분야에서 점점 Computer Vision의 활용이 커지고 있습니다.

그럼에도 불구하고 아직 텍스트 인식만큼 기계가 자동으로 사진을 인식하거나 동영상을 인식하는 상황은 이루어지지 않았다고 보시면 좋겠고요.

그래서 Computer Vision분야에서 해야 될 일이 매우 많다고 이해하시면 되겠습니다.

시간이 갈수록 Computer Vision이란 학문의 중요성은 점점 커지고 있습니다.

한 가지 예를 들어보면, 여러분이 요즘 인터넷을 통해서 가장 많이 하는 활동을 보시면 많은 경우에 사진을 보거나 동영상을 보는 경우입니다.

그러니까 인터넷 트래픽 관점에서 보면, 95% 이상, 압도적인 트래픽이 동영상을 전송하거나 사진을 전송하는 경우죠.

결국에 온라인상에 Computer Vision으로 해석해야지만 되는 정보량이 점점 많아지고 있습니다. 그러니까 텍스트 정보의 증가 못지않게 시각 정보의 양이 아주 급격히 늘어나고 있고요.

현재 기술로는 많은 시각 정보가 제대로 해석되지 않고 그냥 버려져 있는 혹은 그 시각 정보 주변에 쓰여 있는 텍스트 태그를 기반으로 해서 검색을 한다든지 이해를 하는 그런 경우가 많은데요.

인공지능 기술이 계속 발전하다 보면, 사진과 동영상을 기계가 바로 이해해서 그거로부터 사람처럼 행동할 수 있는 날이 머지않을 거라고 생각합니다.

그래서 지금 8개의 강의를 통해서 인공지능의 기초에 대해서 말씀을 드렸습니다. 마지막으로 Computer Vision에 대해서 말씀을 드렸고요.

어떻게 보면 소개만 말씀을 드렸고, ‘이러이러한 점에서 어렵다.’는 점만 말씀을 드렸는데요. 여러분이 이 강의를 통해서 흥미를 가지시고 ‘이런 문제는 풀 수 있겠구나.’

그리고 기사에서 보듯이 인공지능이라는 게 다 되는, 금방이라도 될 것 같은 쉬운 문제만은 아니고 아직도 해결해야 될 문제가 많다. 그렇기 때문에 더 공부를 해야겠다.

이런 생각을 가지게 되셨으면 아주 좋을 것 같습니다. 그러면 마지막 강의를 여기서 마치도록 하겠습니다. 그동안 경청해 주셔서 감사합니다.