Monte Carlo Tree Search

저번시간에 배웠던 내용 :

Monte Carlo method는 결과가 무엇인지는 알고 있지만, 정답을 구하는 과정을 정확히 모르는 경우 사용 가능한 방법입니다.

사용 방법은

1. 엄청나게 많은 무작위 시뮬레이션을 시행한다.
2. 시뮬레이션의 결과를 분석해 정답을 유추한다.

정도로 요약이 가능합니다. 오늘 학습할 Monte Carlo Tree Search역시 simulation을 통해 Tree에서의 진행 방향을 정하기 때문에 이름에 Monte Carlo가 들어가 있습니다.

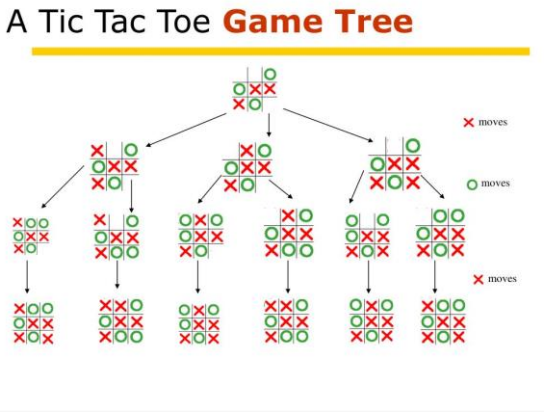
오늘의 내용 :

IBM이 만들었던 Deep Blue는 1997년 세계 체스 챔피언을 상대로 승리를 거두면서 게임의 분야에서도 인공지능이 활용 가능함을 보여줬습니다.

이때 Deep Blue는 **min-max** 알고리즘을 사용했는데 대략 설명하자면 다음과 같습니다.

(각 노드는 게임의 특정 상태를 나타냅니다.)

1. 경기에 앞서 플레이 가능한 모든(혹은 대부분의) 경우의 수를 트리로 기록해 놓습니다.
2. 경기의 진행에 따라 하위 노드를 탐색하며 결과를 살펴봅니다.
3. 나의 승률이 가장 높은 방향으로 진행(play)합니다.
4. 게임의 승부가 날때까지 2-3을 반복하여 진행합니다.



[Tic Tac Toe 의 Game Tree – 각 노드는 게임의 state를 표현한다.]

위와 같은 방법은 성능적으로 강력하지만 이미 전체(또는 대부분의) tree의 모습을 알고 있기 때문에 가능한 알고리즘입니다. 다시 말하자면 엄청나게 많은 경우의 수(board state)를 가지고 있는 바둑 과 같은 게임에서 적용되려면

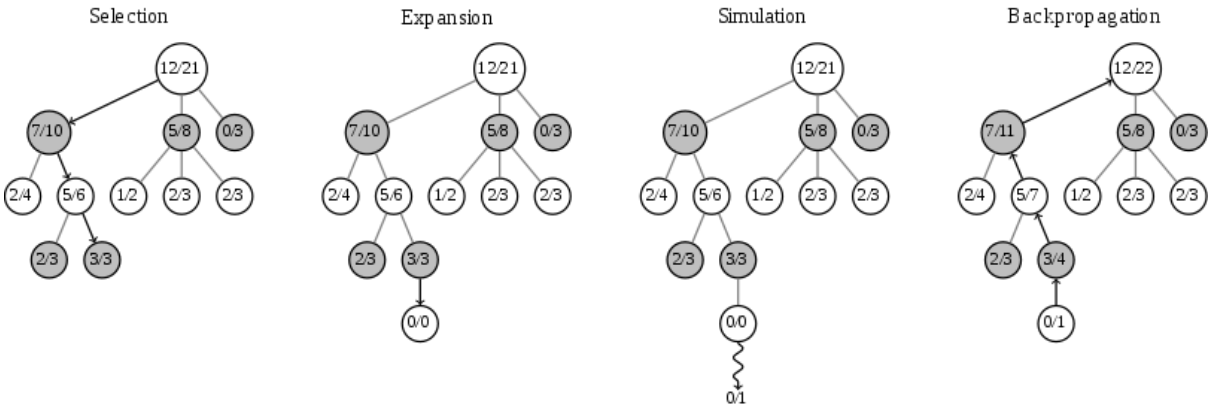
1. 게임의 모든 경우의 수를 기록한 tree를 저장할 엄청나게 많은 메모리가 필요합니다.
2. 게임이 엄청나게 길기 때문에 하위 노드들을 빠르게 비교하기 위해 엄청난 연산력이 필요합니다.

위와 같은 제한사항 때문에 컴퓨터가 바둑으로 인간을 이기는 것은 오랜 시간동안 불가능해 보였습니다. 하지만 2016년 Google의 deep mind에서 만든 AlphaGo가 이세돌 기사와의 대국에서 4승 1패의 성적을 거두게 됩니다. 1997년에 비해 컴퓨팅 능력이 좋아졌고 메모리의 단가도 낮아졌지만 바둑판에서 가능한 배치의 모든 경우의 수인

208,1681,9938,1979,9846,9947,8633,3448,6277,0286,5224,5388,4530,5484,2563,9456,8209,2741,9612,7380,1537,8525,6484,5169,8519,6439,0725,9916,0156,2812,8546,0898,8831,4427,1297,1531,9317,5577,3662,0397,2470,6484,0935가지 (대략 10의 171승 – 출처 : Counting Legal Positions in Go (tromp.github.io))

경우의 수를 모두 저장하고 연산에 고려하기에는 역부족입니다. (참고로 우주 전체 원자의 개수가 10의 80승 정도라고 추정되고 있습니다.) 이러한 제한을 돌파하기위해 AlphaGo가 사용한 알고리즘이 바로 Monte Carlo Tree Search입니다.

How Monte Carlo Tree Search Works.



Selection : 현제 게임의 status를 root로 하는 game tree에서 이미 계산되어 있는 child node들이 있을 경우 가장 높은 승률을 보여주는 child node로 진행합니다.

Expansion : leaf–node 에 도달하게 되면 새로운 child를 하나 더합니다.

Simulation : random한 play를 게임이 끝날 때가지 반복합니다. 새로운 child는 Expansion단계에서 생성한 것 이외에 추가로 생성하지 않습니다.

Backpropagation : Simulation단계에서 확인한 게임의 결과를 상위 node들에 update합니다.

Monte Carlo Tree Search각 단계의 의미 분석

Selection : 현제 게임의 status를 root로 하는 game tree에서 이미 계산되어 있는 child node들이 있을 경우 가장 높은 승률을 보여주는 child node로 진행합니다.

현제 게임의 status를 root로 하는 tree에서 분석을 실행하는 것은 현제에 이르기까지 과거의 play들을 잊는 것을 의미합니다. 이것은 어떤 경로를 통해 현제 status에 도달했는지는 앞으로의 결과에 영향을 끼치지 않는다는 의미가 있습니다. (요즘은 과거를 분석해 상대의 play style을 유추해서 거기에 맞는 play를 하는 알고리즘도 있다고 들었습니다.)

이미 계산되어 있는 child node들이 있을 경우 본 알고리즘은 사전에 계산된 tree없이 실시간으로 예측하는 것을 기본으로 합니다. 다시 말해 미리 전체 트리의 구조를 저장하지는 않지만 실시간으로 Simulation과 Expansion을 반복하며 얻게 되는 데이터들이 있기 때문에 어느정도 미리 계산된 node를 사용하게 됩니다.

가장 높은 승률을 보여주는 child node로 진행합니다. 정확히 설명하기 어렵지만 한술 국으로 통 전체의 맛을 알 수 있다 라는 말이 있습니다. (부대에서 신병이 사고를 치면 이런 말을 듣게 됩니다. 내가 이 친구 잘은 모르겠지만 앞으로 사고 더 칠 것 같다 라는 의미입니다.) 이와 같은 논리로 앞서 맛봤던 play의 결과들이 긍정적이라면 승률이 높을 것이다 라는 논리로 이전의 결과들이 좋았던 node를 더 탐사하게 됩니다.

Expansion : leaf–node 에 도달하게 되면 새로운 child를 하나 더합니다.

Simulation : random한 play를 게임이 끝날 때가지 반복합니다. 새로운 child는 Expansion단계에서 생성한 것 이외에 추가로 생성하지 않습니다.

leaf–node 에 도달하게 되면 새로운 child를 하나 더합니다. / 새로운 child는 Expansion단계에서 생성한 것 이외에 추가로 생성하지 않습니다.

Simulation에서 조차 모든 child를 더하게 된다면 2가지 문제가 생기게 됩니다.

1. Selection 단계에서 root(현제 상황)에 가까운 단계에서 여러 경우로 뻗어 나가지 않고 이미 파악되어 append되어있는 node들을 타고 내려가게 되어서 시아가 좁아지게 됩니다. 2에서 말하게 되지만 root에 가까운 노드가 더 중요합니다.
2. 중요하지 않은 말단의 node를 저장하는데 소중한 메모리들을 낭비하게 됩니다. 말단의 node들은 이미 게임의 결과가 대부분 결정 난 상태를 의미하기 때문에 그만큼 value 가 떨어지게 됩니다. 게임 초장에 승기를 잡으려면 root에 가까운 node들에 노력을 해야 합니다. root에서 먼 곳에 집중하는 것은 초반에 게임 다 던지고 나중에 어떻게 풀어보려는 행위와 같습니다.

Backpropagation : Simulation단계에서 확인한 게임의 결과를 상위 node들에 update합니다.

어떻게 사용하느냐?

While (게임 진행중) :

While (정해진 시간동안 or 정해진 횟수 만큼) : # Runtime에 학습

MCTS 반복(Selection -> Expansion -> Simulation -> Backpropagation -> Selection)

Root에서 봤을 때 가장 좋은 수로(child) 진행(play)

Root = Root의 Best child

놀랍게도 한술국으로 통 전체의 맛을 알 수 있다는 논리가 통합니다.

제가 짠 예제 코드의 경우 몇 가지 Error 가 있습니다.

1. 각 단계의 의미에 대한 이해가 부족한 상태로 코드를 짜다 보니 minmax 알고리즘과 비슷한 형태를 띄게 되어서 mcts는 Expansion에서만 child가 생겨야 하는데 Simulation에서도 child를 더하게 되는 오류를 범했습니다. (전체 tree를 파악해 나가는 과정이라고 이해했던 오류가 있었습니다.)
2. 현제 state의 상위 tree는 잊어야 메모리 활용성이 올라가는데 전체 tree를 기억하고 있는 문제가 있습니다. 메모리 활용성이 떨어집니다.

전체 tictactoe의 경우의 수가 255,168 가지 있지만 매 play마다 1000회 mcts시뮬레이션을 돌리고 그에 따라 착수하는 알고리즘으로도 거의 항상 무승부의 결과를 얻을 수 있습니다.

Tictactoe의 경우 선후공에 상관없이 가장 optimal한 play를 할 경우 항상 무승부가 가능합니다.

예제 코드는

Ubuntu 20-04LTS

Python 3.8.10 64-bit

환경에서 만들어졌습니다.

2021-08-15 강승목