인공지능 Connect4 과제 보고서

- Connect4 AI using MCTS & Reinforcement Learning

컴퓨터학과
2011280050
이세훈

목차

1. 서론

Connect4 게임은 6x7 게임판에서 1~7 사이 열을 선택해 놓을 수 있는 가장 아래 자리에 돌을 놓는 게임이다. 돌을 놓은 직후 돌들이 가로, 세로 대각선으로 4개 이상 이어진다면 승리한다.

이 게임 또한 게이머가 승리하는 상태와 과정을 완벽하게 알 수 있다면 반드시 승리할 수 있다. 하지만, 3^42개의 모든 상태에서 승리하는 과정을 확인하는 것은 불가능에 가깝다. 따라서, 현 상태에서 차례로 다음 상태를 추론해나가는 Backward Chaining 방식으로 인공지능을 구성하려고 노력했다.

개발 과정에서 시행착오와 발전 과정을 거쳐 최종적인 인공지능은 Monte Carlo Tree Search와 이를 바탕으로 한 Reinforcement Learning을 구현해보았다. 컴퓨터 성능으로 인해 많은 양을 학습하지는 못했고 처음부터 Genetic Search도 구현해보고 싶은 욕심이 들기도 했지만 일단 현 상태에서 마무리하였다.

2. Brute Method

제일 처음 접근 방식은 Depth1의 1~7까지 열을 무작위 입력을 통해 30번 경기한 후 승률을 비교해보는 방식이었다. 이후 Depth1의 최고승률에서 Depth2의 1~7열 무작위 입력을 통해 다시 한번 30번씩 경기 후 최저승률을 택한다. 이후 Depth3의 1~7열 무작위 입력 경기 후 Depth1값을 갱신해 Depth1에서 다시 승률을 비교해보는 방식을 생각했다. 총 100,000 경기를 바탕으로 마지막 Depth1의 최고 승률을 선택하는 방식이었다.

이 방식은 전체 코드에 흔적만 남아 있는데 문제점은 1) 처음 Depth1 탐색에서 높았던 승률을 계속해서 재탐색한다는 것이고 2) Depth3까지의 탐색으로는 신뢰성을 얻지 못했기 때문이다.

3. Pure Monte Carlo

이후 Min-Max 방식을 바탕으로 Min-Max의 Heuristic 값으로 Monte Carlo 방식을 사용하려고 했으나 그보다 먼저 단순한 Monte Carlo부터 구현하게 되었다.

```
class Agamotto(game):
    # search 해야할 현재 board
    def __init__(self, game):
        super(Agamotto, self).__init__()
        self.board = game.board
        self.cells = game.cells
        self.winner = game.winner
        self.row_board = game.row_board
        self.marker = game.marker
        # 왼쪽눈은 Tree input, 오른쪽눈은 Tree output
        self.EyeOfAgamotto_left = dict()
        self.EyeOfAgamotto_right = dict()
        self.battle_count = 0
        return
```

위 코드는 Algorithm을 구현하기 전에 Game으로부터 정보를 얻어오는 과정이다. 여기서 눈여겨 볼만한 코드는 EyeOfAgamotto_left와 EyeofAgamotto_right이다. Min_Max를 목표로 했기 때문에 Tree를 구성하기 위한 input, output을 위한 변수이지만 결국은 비효율적으로 사용 중이다. battle_count는 몇 번의 무작위 경기를 했는지 알려주는 변수이다.

```
def VictoryNumber(self):
    # EOA = self.EyesOfAgamotto
    self.FutureSight()
    print(f"닥터: {self.battle_count}개의 전투에서 승리할 수 있는 수는 이것뿐이었네")
    for key, value in self.EyeOfAgamotto_right.items():
        print(f"{key}: {value}", end=", ")
    print (max(self.EyeOfAgamotto_right.items(), key=itemgetter(1)))
    return str(max(self.EyeOfAgamotto_right.items(), key=itemgetter(1))[0])[0]
```

인공지능이 init 이후로 제일 처음 호출하는 함수인 VictoryNumber()는 모의 경기를 준비하는 FutureSight()를 호출한 뒤 Output이 저장되있는 EyeOfAgamotto_right에서 최고값을 string형식으로 리턴해준다.

```
def FutureSight(self):
    if self.cells == 42:
        self.EyeOfAgamotto_left = {str(i): 0 for i in [1, 2, 3, 5, 6, 7]}
    else:
        Battle_list = [j for j in range(1,8) if self.board[5][j-1] == " "]
        self.EyeOfAgamotto_left = {str(i): 0 for i in Battle_list}
    for key in self.EyeOfAgamotto_left.keys():
        self.EyeOfAgamotto_right[key] = self.FutureSight_Fight(key)
```

VictoryNumber()에서 호출된 FutureSight()는 현 게임 상태, 첫 턴인가 혹은 해당 열이 6행까지 놓인 상태인가에 따라 Input 열을 정리해 EyeOfAgamotto_left에 넣는다. 이후 EyeOfAgamotto_left의 key값을 FutureSight_Fight로 넘겨 무작위 전투를 시도한 뒤 그 return값을 EyeOfAgamotto_right에 저장한다.

```python
def FutureSight_Fight(self, battle_field_num):
    # 턴 확인 0이면 닥터 턴(max), 1이면 유저 턴(min)
    global_turn = len(battle_field_num)%2
    minmax_arr=[]
    for minmax in range(30):
        win = 0
        for winrate in range(30):
            orb = deepcopy(self)
            local_turn = orb.marker
            for battle_code in battle_field_num:
                orb.input(orb.validate_input_col(battle_code))
            while not orb.winner and orb.cells > 0:
                orb.input(orb.validate_input_col(str(random.randrange(1, 8))))
            # 닥처차례에 닥터가 승리했거나
            if orb.winner == local_turn and not global_turn:
                win += 1
            # 유저차례에 유저가 이겼거나 비겼으면
            elif global_turn and orb.winner == local_turn or orb.winner == 0:
                win += 1
        # print(f'{battle_field_num}전장에서 {win}번 이겼네')
        minmax_arr.append(win)
        self.battle_count += 900
    return (max(minmax_arr), min(minmax_arr))
```

FutureSight_Fight()가 제일 핵심적인 함수이다. 이 함수에서는 받은 String값은 게임에서 선택한 열의 순서이다. 예를 들어, 내가 1열을 선택하고 적이 3열, 내가 다시 3열을 선택한다면 '133'이라는 문장이다. 전체함수에서는 1~7밖에 사용하지 않더라도 depth를 조절할 수 있는 코드이다. 이 길이에 따라 누구의 턴인지 확인하고 30번 경기의 승률을 30번 구해서 그 중에 최고값과 최저값을 돌려준다.

처음에는 단순히 100번의 경기를 통한 승률 사이에서 1~7열을 선택했는데 그보다 30번의 승률 최고값 사이에서 1~7열을 선택하는 것이 무작위 경기에서 생길 수 있는 통계적 오류를 줄여주었다. 하지만 "최고값이 아니라 평균값이었다면 어땠을까"라는 의문점이 남는다

Pure Monte Carlo의 단점은 승리했다는 결과만 남는다는 점이다. 주석처리 한 print()를 보면 어떻게 어떤 결과를 얻었는지, 예를 들어 "'1234'에서 승리했습니다"는 알 수 있는데 다음번에 다시 호출될 때는 그런 정보가 남지 않는다. 결국, 매번 최대 6300회의 모의 경기를 거쳐야 한다.

4. Monte Carlo Tree Search(MCTS)

Pure Monte Carlo에서 문제점을 해결하기 위해 고안한 방법이 MCTS이다. MCTS는 Selection-Expansion-Simulation-Backpropagation 순서로 Tree를 만들어 나간다. 이 Tree를 바탕으로

의사결정을 하는 것인데 여기서 나는 Simulation과 Backpropagation만 진행해서 Tree를 만들었다.

```
class Agamotto(game):
    # search 해야할 현재 board
    def __init__(self, game, EOA, BC):
        super(Agamotto, self).__init__()
        self.board = game.board
        self.cells = game.cells
        self.winner = game.winner
        self.row_board = game.row_board
        self.marker = game.marker
        self.battle = game.battle
        self.EOA = EOA
        self.BC = BC
        return
```

MCTS의 init()이다. Pure Monte Carlo와 비슷하게 게임에 대한 정보를 얻어온다. 이후 EyesOfAgamotto(EOA)와 BC(Battle Count)를 Agamotto를 선언한 MainStream.py에서 얻어온다. EOA는 dictionay형 자료로 '12312'와 같은 입력순서를 key값으로 하고 승, 무, 패에 대한 정보를 Array형으로 가진다. BC는 단순히 모의 경기의 횟수를 기록하기 위한 정보이다.

```
def VictoryNumber(self):
    self.FutureSight()
    # ex) 1번 배틀이면 11,12,13,14,15,16,17중에 최고값 선택
    battle_winrate_list = []
    battle_list = [1, 2, 3, 5, 6, 7] if self.cells == 42 else [j for j in range(1,8) if self.board[5][j-1] == " "]
    for battle_field in battle_list:
        next_battle = self.battle + str(battle_field)
        try:
            winrate = self.EOA[next_battle][0] / sum(self.EOA[next_battle])
        except KeyError:
            return self.VictoryNumber()
        winrate = self.EOA[next_battle][0] / sum(self.EOA[next_battle])
        battle_winrate_list.append((battle_field, next_battle, winrate))
    print(battle_winrate_list)
    print(f"닥터: {self.BC}개의 전투에서 승리할 수 있는 수는 {str(max(battle_winrate_list, key=lambda x: x[2])[0])[0]}뿐이었네")
    return str(max(battle_winrate_list, key=lambda x: x[2])[0])[0]
```

MCTS의 VictoryNumber()는 Pure Monte Carlo의 FutureSight()와 VictoryNumber()를 합했다. 상황에 따른 Input을 구하고 Input에 따른 Tree를 탐색하고 계산해서 내보내는 역할을 하고 있다. 예를 들어 지금 '11234'라는 상태라면 '112341', '112342', '112343', '112344', '112345', '112346', '112347'의 값들을 가져와서 승률을 구해 비교해 최고값을 return한다. 또한, try~ except~ 부분이 로직

적으로 중요한 부분인데 만약 모의 경기를 통해 1~7 사이에 빼먹은 경기가 있다면 재귀적으로 본인을 리턴해서 탐색을 한 번 더 한다.

```
105     def FutureSight(self):
106         for sight in range(250):
107             orb = deepcopy(self)
108             # 승리자가 나올때 까지 무작위 input
109             while not orb.winner and orb.cells > 0:
110                 orb.input(orb.validate_input_col(str(random.randrange(1, 8))))
111             # local game(orb)에서 승무패에 따라 아가모토의눈에 [승, 무, 패]를 기록
112             if orb.winner == self.marker:
113                 # print(f'{orb.battle}전장에서 이겼네')
114                 for i in range(len(orb.battle)):
115                     win_record = self.EOA.get(orb.battle[0:i+1], [0, 0, 0])
116                     win_record[0] += 1
117                     self.EOA[orb.battle[0:i+1]] = win_record
118             elif orb.winner == 0:
119                 # print(f'{orb.battle}전장에서 비겼네')
120                 for i in range(len(orb.battle)):
121                     win_record = self.EOA.get(orb.battle[0:i+1], [0, 0, 0])
122                     win_record[1] += 1
123                     self.EOA[orb.battle[0:i+1]] = win_record
124             else:
125                 # print(f'{orb.battle}전장에서 졌네')
126                 for i in range(len(orb.battle)):
127                     win_record = self.EOA.get(orb.battle[0:i+1], [0, 0, 0])
128                     win_record[2] += 1
129                     self.EOA[orb.battle[0:i+1]] = win_record
130         self.BC += 250
```

Pure Monte Carlo에서 FutureSight_Fight()와 비슷한 역할을 하는 FutureSight()는 여기서도 핵심 Algorithm이다. 250번 모의 경기를 해서 그 결과들을 Tree에 저장한다. 예를 들어 '1123'에서 승리했다면 트리의 '1123', '112', '11', '1'에 승리 횟수를 더한다.

MCTS는 Pure Monte Carlo에서 아쉬웠던 모의 경기의 결과를 내버려 두지 않고 활용할 수 있다는 장점이 있다. 하지만 여전히 탐색량은 적지 않고 450번 모의 경기 기준 6턴 동안 예측한 방향으로 경기가 진행되어야 Pure Monte Carlo의 예측량과 비슷하다.

5. Monte Carlo Tree Search & Reinforcement Learning

MCTS에서의 단점인 예측량의 부족을 해결하기 위해서 Reinforcement Learning을 이용하기로 했다. Reinforcement Learning의 핵심은 이전의 행동이 긍정적 결과(포상)을 가져왔다면 다음번에도 그러한 행동을 하도록 유도하는 것이다. 즉, MCTS에서 모의 경기를 할 때마다 그 과정을 기록해서 다음번에 새로운 온전한 경기에서 사용한다.

```python
def VictoryNumber(self):
    # ex) 1번 배틀이면 11,12,13,14,15,16,17중에 최고값 선택
    battle_winrate_list = []
    battle_list = [1, 2, 3, 5, 6, 7] if self.cells == 42 else [j for j in range(1,8) if self.board[5][j-1] == " "]
    for battle_field in battle_list:
        next_battle = self.battle + str(battle_field)
        self.FutureSight(battle_field)
        while sum(self.EOA[next_battle]) < 30:
            self.FutureSight(battle_field)
        if self.marker == 'X':
            win = self.EOA[next_battle][0]
            lose = self.EOA[next_battle][2]
            draw = self.EOA[next_battle][1]
            winrate = win / (win+lose+draw)
        elif self.marker == "O":
            win = self.EOA[next_battle][2]
            lose = self.EOA[next_battle][0]
            draw = self.EOA[next_battle][1]
            winrate = win / (win + lose + draw)
        battle_winrate_list.append((battle_field, next_battle, winrate))
        print(f'Dr.Weird: {battle_field}를 선택한 {next_battle}전장의 전투는 {win+lose+draw}번, 승리 {win}, 패배 {lose}, 무승부 {draw}, 승률은 {winrate*100:0.2f}%네')
    vic_num = str(max(battle_winrate_list, key=lambda x: x[2])[0])[0]
    print(f"Dr.Weird: {self.BC}개의 전투에서 승리할 수 있는 수는 {vic_num}뿐이었네")
    return vic_num
```

init()은 MCTS의 것과 같은 것을 사용한다. VictoryNumber()는 이전처럼 Input을 조건에 따라 설정해준다. 이후 1~7열마다 최소 30번을 확인할 수 있도록 보장해주고 만약 30회를 넘었다면 한 번만 FutureSight()를 실행한 후 승률을 계산하여 최고값을 가진 열의 숫자를 return한다.

```python
def FutureSight(self, code):
    orb = deepcopy(self)
    str_len = len(orb.battle)
    orb.input(orb.validate_input_col(str(code)))
    # 승리자가 나올때 까지 무작위 input
    while not orb.winner and orb.cells > 0:
        orb.input(orb.validate_input_col(str(random.randrange(1, 8))))
    # local game(orb)에서 승무패에 따라 아가모토의눈에 [승, 무, 패]를 기록
    if orb.winner == 'X':
        # print(f'{orb.battle}전장에서 이겼네')
        for i in range(str_len, len(orb.battle)):
            win_record = self.EOA.get(orb.battle[0:i+1], [0, 0, 0])
            win_record[0] += 1
            self.EOA[orb.battle[0:i+1]] = win_record
    elif orb.winner == 0:
        # print(f'{orb.battle}전장에서 비겼네')
        for i in range(str_len, len(orb.battle)):
            win_record = self.EOA.get(orb.battle[0:i+1], [0, 0, 0])
            win_record[1] += 1
            self.EOA[orb.battle[0:i+1]] = win_record
    else:
        # print(f'{orb.battle}전장에서 졌네')
        for i in range(str_len, len(orb.battle)):
            win_record = self.EOA.get(orb.battle[0:i+1], [0, 0, 0])
            win_record[2] += 1
            self.EOA[orb.battle[0:i+1]] = win_record
    self.BC += 1
```

FutureSight()에서는 parameter로 받은 값을 입력한 상태에서 모의경기의 결과를 record에 기록한다. record에는 key값은 '123145'와 같이 입력한 순서를 나타내는 문자열이고, 값은 후턴 X 기준 승, 무, 패를 Array인 형태로 기록한다. 또한 각 state마다 예측의 독립성을 위해서 현재상태까지만 저장하기로 했다.

MCTS에 Reinforcement Learning을 적용한 것은 예측적으로 만족스러웠다. 다만, 한 번도 본 적 없는 상태에서 예측하는 속도가 느린 편이며 충분히 학습한 뒤에야 실제 사용이 가능하다. 또한

Random Walk를 기준으로 코드를 구성해 Search Space가 굉장히 많다. 이를 Random Status를 기준으로 했으면 속도가 좀 더 빨랐을 것이다.

6. 결론

결과적으로는 완벽하지는 않지만 충분히 많은 시뮬레이션을 통해 다음 수의 승률을 예측하고 이 데이터를 누적해서 지속적으로 승률을 유지할 수 있는 인공지능을 만들었다. 만드는 과정에서 시행착오도 겪었고 수업 시간에 아직 다루지 않은 Machine Learning도 공부하는 등 인공지능에 대해 흥미를 풀 수 있었다.