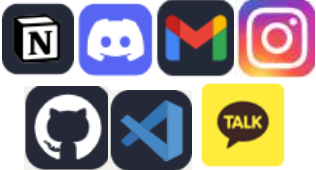


# PBL FINAL REPORT

## Target Game: Othello

Teammates: 김진원, 박유상, 박윤준, 양진우, 홍성필

문제 해결 기간	2024.5.15 ~ 2024.06.11	
소통 방식	회의 진행 시간	회의 장소
	매주 월, 수 1630 : 1730 (1hr) 수요일 1400 : 1500 (1hr)	월, 수: 4공학관 Smash Room
회의 진행 방식		사전에 회의 안건을 정하고 각자 충분한 자료 조사 및 생각을 정리해 온다. 팀장의 주도 하에 회의 진행.
회의 참석자 및 역할 분담		
양진우(팀장)	DQN+ResNet 코드 1차 구현. PBL FINAL REPORT 작성. PBL Analysis Report 작성	
김진원	Othello 자료 조사. Demo Video 초안	
박유상	DQN, ResNet 자료 조사	
박윤준	DQN+ResNet 코드 1차 구현. 회의록 정리 및 Notion 관리. Demo Video 편집	
홍성필	Othello Base Code 작성. DQN+ResNet 코드 2차 구현. Demo Video 편집	

## 1. Target Game: Othello

오델로(Othello)는 인공지능 연구 및 개발에서 자주 사용되는 보드 게임이다. 이 게임은 흑과 백의 돌을 사용하여 두 명의 플레이어가 번갈아 가며 놓으며, 상대방의 돌을 자신의 돌로 양쪽에서 감싸는 방식으로 진행된다. 8x8 크기의 보드에서 플레이 되며, 목표는 게임이 끝났을 때 보드 위에 가능한 많은 자신의 돌을 놓는 것이다.

오델로의 규칙은 비교적 간단하지만, 게임의 전략적 깊이는 매우 깊어 다양한 전술과 복잡한 의사 결정을 요구한다. 이러한 특성 때문에 오델로는 인공지능 연구에서 중요한 도전 과제가 된다. 특히, 오델로는 상태 공간이 크고, 최적의 수를 찾기 위한 계산량이 많아 인

공지능 알고리즘의 성능을 테스트하고 개선하는데 유용하다.

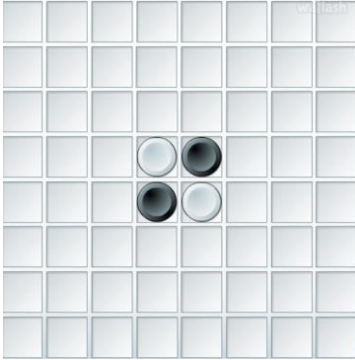
오델로는 미니맥스 알고리즘, 알파-베타 가지치기, 몬테카를로 트리 탐색(MCTS) 등 다양한 인공지능 기법이 적용된 사례가 많다. 이러한 연구를 통해 인공지능은 사람과의 대결에서 높은 수준의 성능을 보이며, 게임 이론 및 알고리즘 개발에 큰 기여를 하고 있다.

## 2. Basic Setting of Othello

오델로를 세팅하기 위해 사전에 정의할 것들이 있다. 액션 공간(Action Space)은 현재 상태에서 유효한 모든 위치로 구성이 된다. 이때 유효한 모든 위치는 최소한 개 이상의 상대방 돌을 뒤집을 수 있는 위치를 뜻한다. 관측 공간(Observation Space)은 8\*8 격자의

현재 상태로, 모든 검은색과 흰색 돌의 위치가 표시된다. 우리는 보드 상태를 총 3가지로 나누어 표현하기로 했다. 보드 상태는 다음과 같다.

[0: 빈칸, 1: 검은색 돌, 2: 흰색 돌]



[그림 1.] 오델로 기본 세팅

처음에는 (4,5)와 (5,4) 위치에 검은색 돌이 있고 (4,4)와 (5,5) 위치에 흰색 돌을 배치한다. 게임은 검은색 돌부터 시작한다. 오델로에서의 Reward(보상)은 2가지로 설정했다. 우선 기본 보상은 턴 종료 후 (자신 색의 돌 - 상대 색의 돌)이다. 그리고 게임 종료 보상으로는 승리한 쪽이 +100, 반대는 -100을 부여했다.

### 3. Code Setting Before Playing

```
1 import gym_games
2 import gymnasium as gym
3 from DeepQNet import DQN
4 env = gym.make('Othello-v0', render_mode='human')
```

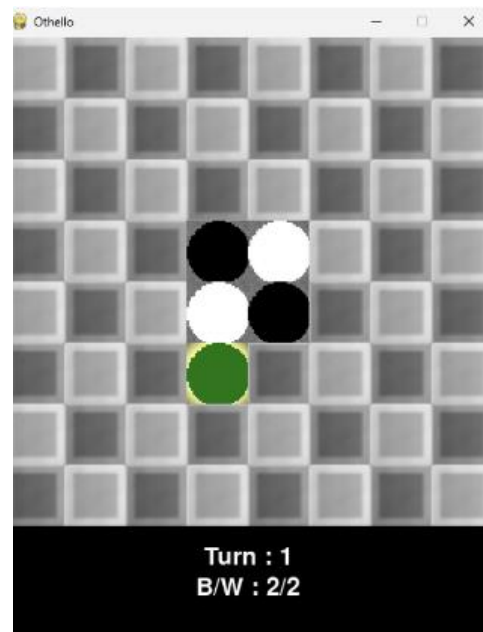
[그림 2.] Gym 환경 구축 코드

강화 학습을 위해 오델로 게임을 Gym 환경으로 구축하였다.

```
1 env.metadata['autoplay'] = False
2 while True:
3     print(info)
4     obs, reward, done, _, info = env.step(None)
```

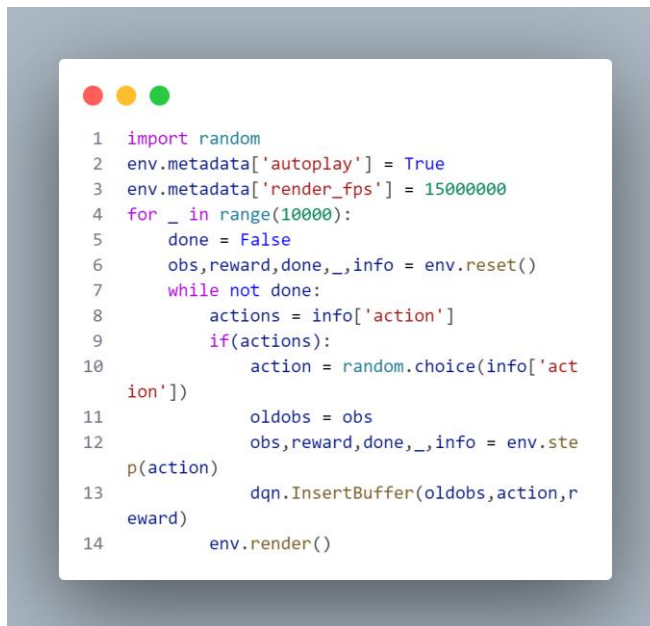
[그림 3.] 자동플레이 False/True 코드

Metadata의 autoplay를 False로 설정하면 직접 플레이가 가능하다. 이때 Action 값은 마우스를 통해 입력된다.



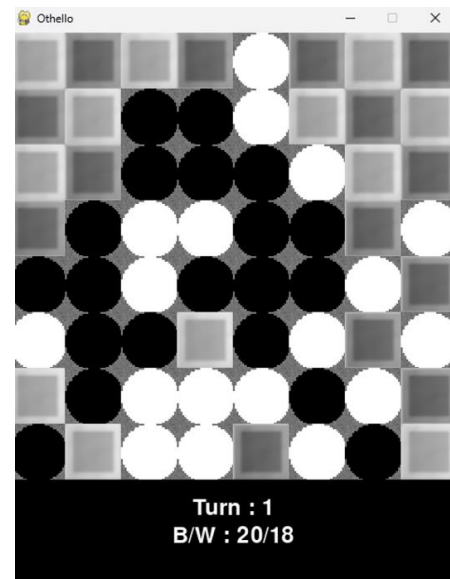
[그림 4.] 직접플레이로 게임 실행하는 모습

직접 플레이로 실행 시 돌을 놓을 수 있는 위치는 초록색으로 표시되는 것을 볼 수 있다. Autoplay를 True로 설정하면 자동으로 시뮬레이션을 생성한다.



[그림 5.] Metadata 코드 및 실행

Metadata의 'render\_fps' 조작을 통해 시뮬레이션의 빠르기를 조절할 수 있다.

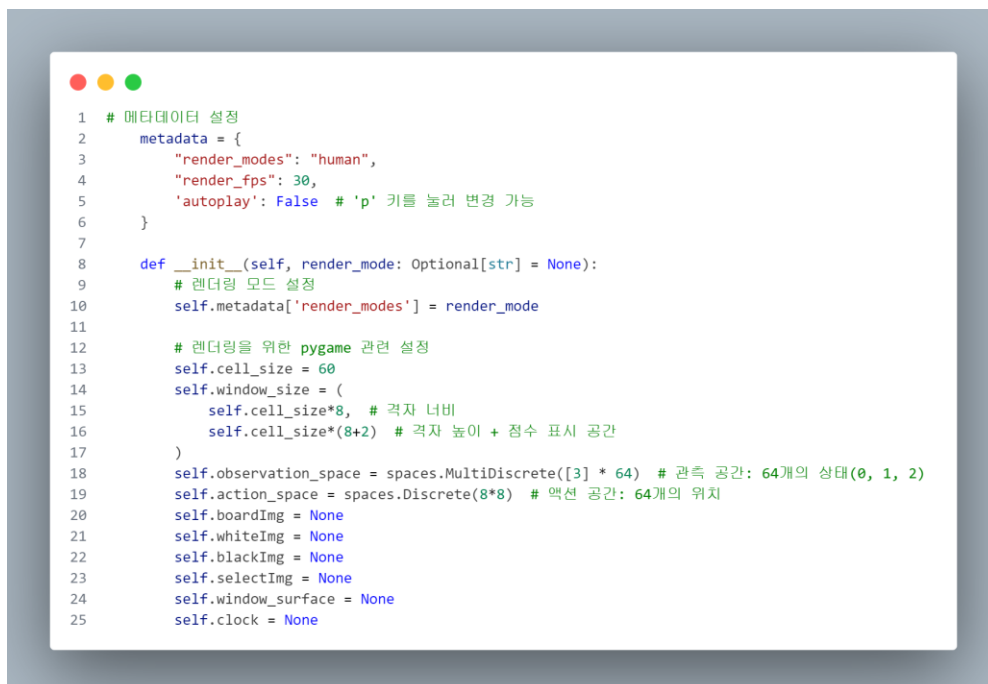


[그림 6.] 자동플레이로 게임 실행하는 모습

GUI에서 Turn 1은 검정 돌을, Turn 2는 흰색 돌의 순서를 의미한다. 아래의 B/W는 현재 보드판에 있는 돌의 개수를 의미한다.

## 4. Othello Base Code

인공지능으로 학습하는 오텔로 코드를 만들기 전에 우선 오텔로의 기본 코드를 구성해야 한다. 지금부터 소개하는 코드들은 class OthelloEnv(gym.Env)안에 있는 코드들이다.



[그림 7.] 초기화 메서드 & Metadata Setting 코드

위 코드는 Python의 Gym 환경 설정을 위해 사용되는 초기화 메서드와 메타데이터 설정에 대한 코드이다. Metadata에서는 2가지를 설정할 수 있다. autoplay에서는 자동 재생 모드 여부를, render\_fps에서는 autoplay가 True일 때 시뮬레이션의 FPS를 정하는 역할을 한다. 뒤의 표에서 초기화 메서드인 `__init__` 코드에 필요한 내용들을 정의한다.

render_mode	초기화 시 선택된 렌더링 모드를 설정
self.cell_size	각 셀의 크기를 픽셀 단위로 정의
self.window_size	게임 창의 크기를 정의. 이 크기는 격자의 너비와 높이 및 점수 표시 공간을 포함
self.observation_space	관측 공간을 정의. 64개의 셀 각각이 3가지 상태(0, 1, 2) 중 하나를 가질 수 있음을 나타냄
self.action_space	액션 공간을 정의. 64개의 위치 중 하나를 선택할 수 있음을 나타냄
self.boardImg, self.whitelmg, self.blacklmg, self.selectlmg	각각 게임 보드, 흰색 돌, 검은색 돌, 선택된 위치의 이미지를 저장할 변수
self.window_surface	Pygame을 사용하여 렌더링할 창의 표면을 저장할 변수
self.clock	게임의 프레임 속도를 제어하기 위한 Pygame 시계 객체를 저장할 변수

```

1 def reset(self, *, seed: Optional[int] = None, options: Optional[dict] = None):
2     # 환경 초기화
3     super().reset(seed=seed)
4     self.board = np.zeros(8*8, dtype=int) # 보드 초기화
5     self.board[27] = 1 # 초기 검은색 돌 위치
6     self.board[36] = 1
7     self.board[28] = 2 # 초기 흰색 돌 위치
8     self.board[35] = 2
9     self.curplayer = 1 # 현재 플레이어 (1: 검은색, 2: 흰색)
10    self.blackSum = 2 # 검은색 돌 개수
11    self.whiteSum = 2 # 흰색 돌 개수
12
13    # 렌더링 GUI 초기화
14    if self.metadata['render_modes'] == "human":
15        self._init_render_gui()
16    self.render()
17
18    # 초기 상태와 정보 반환
19    return (self.board, 0, False, False, {'autoplay': self.metadata['autoplay'], 'turn': self.curplayer, 'action': self.get_valid_actions(), 'blackSum': self.blackSum, 'whiteSum': self.whiteSum})

```

[그림 8.] 게임 초기 상태 설정 코드

Reset 함수에서는 게임을 초기 상태로 설정하는 역할을 한다. 환경 초기화 단계에서는 8\*8 보드를 0으로 초기화 한다. 초기 돌의 위치를 설정한 후, 현재 플레이어의 색을 검은색으로 설정한다. 초기 돌의 위치를 설정하였기에 검은색과 흰색 돌의 개수를 각각 2로 설정한다. 렌더링 모드가 human이라면 GUI 초기화를 수행하고 현재 상

태를 렌더링한다. 마지막에는 보드 상태, 점수, 종류 여부 등을 포함한 초기 상태와 추가 정보를 반환한다.

```
1 def render(self):
2     #렌더링 모드에 따라 렌더링 수행
3     if self.metadata['render_modes'] == "human":
4         return self._render_gui()
5
6 def _init_render_gui(self):
7     #pygame GUI 초기화
8     if self.window_surface is None:
9         pygame.init()
10        pygame.display.init()
11        pygame.display.set_caption("Othello")
12        self.window_surface = pygame.display.set_mode(self.window_size)
13        pygame.font.init()
14        self.font = pygame.font.Font(None, 36)
15        self.MouseX, self.MouseY = 0,0
16        if self.clock is None:
17            self.clock = pygame.time.Clock()
18        if self.boardImg is None:
19            self.boardImg = pygame.transform.scale(pygame.image.load(path.join(path.dirname(__file__), "img/board.png")), (self.cell_size*8,self.cell_size*8))
20        if self.whiteImg is None:
21            self.whiteImg = pygame.transform.scale(pygame.image.load(path.join(path.dirname(__file__), "img/white.png")), (self.cell_size,self.cell_size))
22        if self.blackImg is None:
23            self.blackImg = pygame.transform.scale(pygame.image.load(path.join(path.dirname(__file__), "img/black.png")), (self.cell_size,self.cell_size))
24        if self.selectImg is None:
25            self.selectImg = pygame.transform.scale(pygame.image.load(path.join(path.dirname(__file__), "img/select.png")), (self.cell_size,self.cell_size))
26
27 def _render_gui(self):
28     #pygame GUI 렌더링
29     self.window_surface.fill((0, 0, 0))
30     self.clock.tick(self.metadata["render_fps"])
31     self.window_surface.blit(self.boardImg, (0,0)) #보드 이미지 렌더링
32     for i, v in enumerate(self.board):
33         if(v !=0 ): #빈칸이 아닌경우
34             row, col = np.unravel_index(i, (8,8)) #1차원 인덱스를 2차원 좌표로 변환
35             pos = (col * self.cell_size, row * self.cell_size) #렌더링 위치 계산
36             if(v==1):
37                 self.window_surface.blit(self.blackImg, pos) #검은색 돌 렌더링
38             else:
39                 self.window_surface.blit(self.whiteImg, pos) #흰색 돌 렌더링
40
41     #현재 턴과 점수 렌더링
42     self._render_text(f"Turn : {self.Curplayer}",self.cell_size*8//2, self.cell_size*9-30)
43     self._render_text(f"B/W : {self.blackSum}/{self.whiteSum}",self.cell_size*8//2, self.cell_size*9)
44
45     if(not self.metadata['autoplay']):
46         #자동 재생 모드가 아닌 경우, 마우스 위치에 선택 이미지 렌더링
47         row = min(self.MouseY//self.cell_size, 8*8-1)
48         col = self.MouseX//self.cell_size
49         a = 8*row+col
50         if self.is_valid_action(a):
51             pos = (col * self.cell_size, row * self.cell_size)
52             self.window_surface.blit(self.selectImg, pos)
53
54     pygame.event.pump()
55     pygame.display.update()
56
57 def _render_text(self, text, x,y):
58     #텍스트 렌더링 헬퍼 함수
59     text_surface = self.font.render(text, True, (255, 255, 255))
60     self.window_surface.blit(text_surface, (x - text_surface.get_width() // 2, y - text_surface.get_height() // 2))
61
62 def _check_coordinates(self, pos):
63     if( 0< pos <8 *self.cell_size ):
64         return True
65
66 def close(self):
67     pygame.quit()
68     return super().close()
```

[그림 9.] Rendering 코드

그 후 렌더링 코드에 대한 내용이 나온다. 모델로 구현과는 직접적인 연관이 없기에 코드에 대한 설명은 생략하도록 한다.



```
1 def is_valid_action(self, a):
2     # 주어진 위치에 돌을 둘 수 있는지 확인하는 함수
3     if not ( 0 <= a < 8* 8):
4         return False
5     if self.board[a] != 0:
6         return False # 이미 돌이 있는 위치에는 둘 수 없음
7     # 상하좌우 및 대각선 방향에 적군 돌이 있는지 확인
8     directions = [(0, 1), (0, -1), (1, 0), (-1, 0), (1, 1), (-1, -1), (1, -1), (-1, 1)]
9     row, col = np.unravel_index(a, (8,8))
10    for dr, dc in directions:
11        r, c = row + dr, col + dc
12        find = False
13        while 0 <= r < 8 and 0 <= c < 8:
14            if self.board[ 8*r+c] == 0:
15                break #빈 칸을 만나면 탐색 종료
16            elif self.board[ 8*r+c] != self.Curplayer:
17                r, c = r + dr, c + dc
18                find = True # 상대방 돌을 만나면 계속 탐색
19            else:
20                if(find):
21                    return True #자신의 돌 만나고 중간에 상대 돌 있으면 유효 위치
22                else:
23                    break #자신 돌 만났지만 중간에 상대 돌 없다면 무효
24    return False
```

[그림 10.] 유효한 행동인지 판단하는 코드

is\_valid\_action 함수는 주어진 위치에 돌을 둘 수 있는지 확인하는 함수이다. 이 함수는 크게 4단계로 이루어져 있다.

## 1. 위치 유효성 검사


위치 'a'가 보드 범위 내에 있는지 확인한다. 범위를 벗어난다면 'False'를 반환한다. 또한 만약 해당 위치에 이미 돌이 위치해 있다면 'False'를 반환한다.

## 2. 돌 놓기 가능여부 확인

돌을 둘 수 있는지 확인하기 위해 상하좌우, 대각선을 탐색한다. 현재 위치를 (row, col)로 변환하고, 각 방향에 대해 탐색을 진행한다.

## 3. 방향 별 탐색

row, col을 (dr, dc) 형태로 바꾼 뒤 탐색을 진행한다. 첫 조건으로 적의 돌을 만나야 한다. 만나지 못한다면 그 방향으로의 탐색은 종료한다. 적 돌을 만나면 계속 탐색을 진행하여 자신의 돌을 만날 때까지 진행한다. 자신의 돌을 만나고 그 중간에 적 돌이 있었다면 True를 반환한다. 자신의 돌을 만나지 못하거나 적 돌을 만나지 못하면 해당 방향으로의 탐색은 종료하고 다른 방향을 탐색한다. 모든 방향을 탐색했지만 유효한 위치가 발견되지 않았다면 False를 반환한다.



```

1  def capture_action(self, a):
2      # 주어진 위치에 돌을 두고 상대방 돌을 뒤집는 함수
3      blackSumold = self.blackSum
4      whiteSumold = self.whiteSum
5      directions = [(0, 1), (0, -1), (1, 0), (-1, 0), (1, 1), (-1, -1), (1, -1), (-1, 1)]
6      row, col = np.unravel_index(a, (8, 8))
7      for dr, dc in directions:
8          r, c = row + dr, col + dc
9          find = False
10         while 0 <= r < 8 and 0 <= c < 8:
11             if self.board[8*r+c] == 0:
12                 break
13             elif self.board[8*r+c] != self.Curplayer:
14                 r, c = r + dr, c + dc
15                 find = True
16             else:
17                 if find:
18                     # 상대방 돌을 뒤집음
19                     r, c = r - dr, c - dc
20                     while (r, c) != (row, col):
21                         self.board[8*r+c] = self.Curplayer
22                         if self.Curplayer == 1:
23                             self.blackSum += 1
24                             self.whiteSum -= 1
25                         else:
26                             self.blackSum -= 1
27                             self.whiteSum += 1
28                     r, c = r - dr, c - dc
29                     break
30                 else:
31                     break

```

[그림 11.] 행동에 의해 돌 뒤집는 함수 코드

Capture\_action 함수 코드는 주어진 위치에 돌을 두고 상대방의 돌을 뒤집는 코드이다. 이 코드는 총 4가지 단계로 이루어져 있다.

## 1. 초기 상태 저장 및 위치 변환

돌을 뒤집기 전, 현재 검은색 돌과 흰색 돌의 개수를 저장한다. 주어진 위치 'a'를 (row, col)로 변환한 후 상하 좌우, 대각선 방향으로 탐색을 시작한다.

## 2. 방향 별 탐색 및 돌 뒤집기

각 방향에 대해 적 돌을 탐색한다. 처음에 적 돌을 만나지 않았다면 해당 방향 탐색은 종료한다. 적 돌을 만났다면 자신의 돌을 만날 때까지 계속해서 진행한다. 자신의 돌을 만났다면 다시 역방향으로 이동하면서 돌을 뒤집는다. 돌을 뒤집으며 돌의 개수를 업데이트 한다. 돌을 뒤집으며 현재 플레이어의 돌 개수는 증가시키고 상대 플레이어의 돌 개수는 감소시킨다.





```

1 # 현재 플레이어의 돌 추가
2     if self.Curplayer == 1:
3         if blackSumold != self.blackSum:
4             self.board[a] = self.Curplayer
5             self.blackSum += 1
6     else:
7         if whiteSumold != self.whiteSum:
8             self.board[a] = self.Curplayer
9             self.whiteSum += 1

```

[그림 12.] 돌 뒤집음에 따라 점수가 변하는 코드



```

1 def get_valid_actions(self):
2     # 현재 상태에서 유효한 모든 액션 반환
3     actions = []
4     for i in range(0, 8*8):
5         if self.is_valid_action(i):
6             actions.append(i)
7     return actions

```

[그림 13.] 현상태에서 가능한 행동들을 불러오는 코드



```

1 def step(self, a):
2     if ((self.blackDone) and (self.Curplayer == 1)) or ((self.whiteDone) and (self.Curplayer == 2)):
3         self.Curplayer = 3 - self.Curplayer
4         actions = self.get_valid_actions()
5         done = False
6         if(not actions):
7             done = True
8             if(self.Curplayer == 1):
9                 self.blackDone = True
10            else:
11                self.whiteDone = True
12            return ( self.board, 0, done , False, {'autoplay': self.metadata['autoplay'], 'turn':self.Curplayer,'action' : actions, 'blackSum':self.blackSum,'whiteSum':self.whiteSum} )
13
14 if(not self.metadata['autoplay']):
15     while True:
16         pygame.event.pump()
17         pygame.display.update()
18         self.MouseX , self.MouseY = pygame.mouse.get_pos()
19         self.render()
20         if(pygame.mouse.get_pressed()[0]):
21             row = min(self.MouseY//self.cell_size, 8*8-1)
22             col = self.MouseX//self.cell_size
23             actions = self.get_valid_actions()
24             a = 8*row+col
25             if (not actions) or (a in actions):
26                 break
27
28 actions = self.get_valid_actions()
29 if(a is None) or (not actions):
30     self.Curplayer = 3 - self.Curplayer
31     actions = self.get_valid_actions()
32     done = False
33     if(not actions):
34         done = True
35         if(self.Curplayer == 1):
36             self.blackDone = True
37         else:
38             self.whiteDone = True
39     return ( self.board, 0, done , False, {'autoplay': self.metadata['autoplay'], 'turn':self.Curplayer,'action' : actions, 'blackSum':self.blackSum,'whiteSum':self.whiteSum} )
40 elif(a in actions):
41     oldblacksum = self.blackSum
42     oldwhitesum = self.whiteSum
43     self.capture_action(a)
44     player = self.Curplayer
45     self.Curplayer = 3 - self.Curplayer
46     actions = self.get_valid_actions()
47     done = False
48     reward = 0
49     if(not actions):
50         done = True
51         if(self.Curplayer == 1):
52             self.blackDone = True
53         else:
54             self.whiteDone = True
55         if(self.blackSum == self.whiteSum):
56             reward = 0
57         elif ((player ==1) and (self.blackSum>self.whiteSum)) or ((player == 2) and (self.blackSum<self.whiteSum)):
58             reward = 100
59         else:
60             reward = -100
61     else:
62         if(player==1):
63             reward = (self.blackSum - oldblacksum) - (self.whiteSum - oldwhitesum)
64         else:
65             reward = (self.whiteSum - oldwhitesum) - (self.blackSum - oldblacksum)
66     return ( self.board, reward, done , False, {'autoplay': self.metadata['autoplay'], 'turn':self.Curplayer,'action' : actions, 'blackSum':self.blackSum,'whiteSum':self.whiteSum} )
67

```

[그림 14.] 게임 한 턴 진행 코드

Step함수 코드에서는 모델로 게임의 한 턴을 진행한다. 위에서 구한 유효한 액션들을 가져온다. 액션 'a'가 가



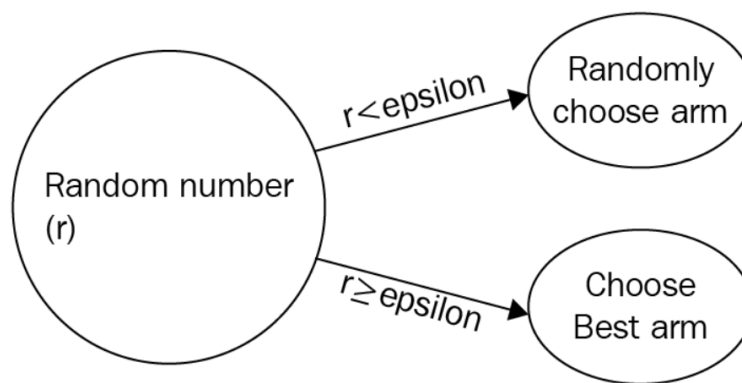
능한 경우, 실행하고 상대의 돌을 뒤집는다. 그 후 플레이어를 변경하여 새로운 유효한 액션 목록을 업데이트한다. 액션이 유효하지 않은 경우에는 -100의 보상을 반환한다. 더 이상 새로운 유효한 액션이 없다면 게임이 종료되었음을 나타낸다.

## 5. Learning Algorithms

기본 모델로 코드는 완성하였다. 이제 인공지능이 학습을 진행하는 코드를 구현해야 했다. 어떠한 과정을 통해 최종 알고리즘을 선택하였는지 설명하겠다.

### 5-1. Limitations on E-Greedy Algorithm

코드구상 초기에는 수업시간에 배운 내용들을 토대로 입실론 그리디 정책(E-Greedy Policy)을 활용하기로 했다. 입실론 그리디 정책은 일정 확률을 가지고 랜덤한 정책을 통해 탐색에 기회를 제공하는 정책이다. 코드를 구현하며 생각을 해본 결과, 만약 이미 많이 가본 경로에서 가치가 적다는 것이 완전히 확인이 된다면, 더 이상 그 경로를 탐색할 필요가 없을 것이라고 생각했다. 엡실론 그리디 정책에서는 이를 위해 간단하게, 입실론을 하이퍼 파라미터로 두어 시간이 지남에 따라 감소시킨다. 하지만 이는 애매한 결정이 되어 성능 하락의 원인이 될 수도 있다고 생각하였다. 그래서 우리는 몬테 카를로 트리 탐색을 활용하기로 하였다.



[그림 15.] E-Greedy Algorithm

### 5-2. Applications of Monte Carlo Tree Search

모델로의 경우 최대 60가지의 행동이 가능하며  $3^8(8 \times 8)$  개의 상태가 존재한다. 에피소드의 길이는 길지 않을 것으로 판단되나 상태 개수가 매우 많다. 고민을 해본 결과, 가장 좋은 방법은 각 상태의 방문 회수를 추적하고 상태의 가치를 종합적으로 고려하여 탐사와 활용의 밸런스를 맞추는 것이라고 생각했다. 전통적인 방법론으로 몬테 카를로 트리를 구축하는 것이 떠올랐다.

따라서 DQN 모델이 신경망 기반 가치 학습을 진행하되 Behavior Policy로 엡실론 그리디 정책이 아닌, 몬테 카를로 트리 탐색에서 사용하는 UCT (Upper Confidence Boundary of Tree) 전략을 코드에서 사용한다. 그리고 이를 위해서 메모리 공간을 희생하고 방대한 상태 방문 테이블을 구축하는 것을 목표로 하였다. 이제부터 소개하는 코드들은 DeepQResNet.py 안에 있는 내용들이다.



```
1  def GetCount(self, state):
2      hash_value = self._Gethash(state)
3      half_length = len(hash_value) // 2
4      first_half = hash_value[:half_length]
5      second_half = hash_value[half_length:]
6      row = int(first_half, 16) % 10
7      col = int(second_half, 16) % 10
8      if hash_value not in self.hashtable[row][col]:
9          return 0
10     else:
11         return self.hashtable[row][col][hash_value]
12
13  def flush(self):
14      buffer = self.replay_buffer
15      self.replay_buffer = []
16      return buffer
17
18  def _Gethash(self, state):
19      # 2차원 배열의 각 요소 값을 결합하여 하나의 문자열로 만듭니다.
20      combined_string = ''.join(map(str, state))
21      # 결합된 문자열에 대한 해시 값을 계산합니다.
22      hash_value = hashlib.md5(combined_string.encode()).hexdigest()
23      return hash_value
24
25  def _InsertHashTable(self, state):
26      hash_value = self._Gethash(state)
27      half_length = len(hash_value) // 2
28      first_half = hash_value[:half_length]
29      second_half = hash_value[half_length:]
30      row = int(first_half, 16) % 10
31      col = int(second_half, 16) % 10
32      if hash_value not in self.hashtable[row][col]:
33          self.hashtable[row][col][hash_value] = 1
34      else:
35          self.hashtable[row][col][hash_value] += 1
```

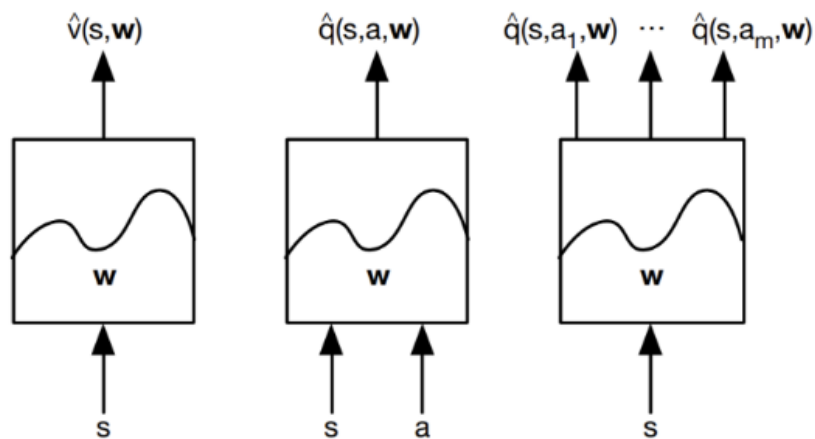
[그림 16.] 해시 테이블 및 값 계산 코드

Learning 알고리즘을 추가하기 전, 검색과 저장의 시간을 단축하기 위해 해시 테이블을 만들었다. 모델로의 상태는 2차원 배열임으로 먼저 이를 하나의 해시 값으로 변화시킨다. 그 후, 각각 절반의 비트 값을 다시 해시 값으로 변환한다. 결과적으로 각 상태에 대한 두 개의 해시 값이 존재함으로 이를 테이블 인덱스로 활용한다.

### 5-3. Using Deep-Q-Network (DQN)

DQN(Deep Q-Network)은 강화 학습 알고리즘의 종류로, 에이전트가 환경에서 행동을 수행하고 그 결과에 대한 보상을 기반으로 학습하는 방식이다. Q-learning 알고리즘을 기반으로 하며, 신경망을 사용하여 상태-가치 함수(State-Value Function)를 근사한다. 상태-가치 함수는 각 상태에서 각 행동을 취했을 때 예상되는 보상 값을 나타내는 함수이다. DQN 에이전트는 상태-가치 함수를 기반으로 가장 높은 보상을 얻을 수 있는 행동을 선택한다.

DQN은 복잡한 환경에서도 효과적으로 학습될 수 있다. 사전 지식이 없더라도 환경과의 상호 작용을 통해 직접 학습을 할 수 있다. 또한 여러 전략을 통해 효율적으로 환경을 탐색할 수 있기에 이 DQN을 모델로를 학습시키는 메인 프로세스로 선택하였다.

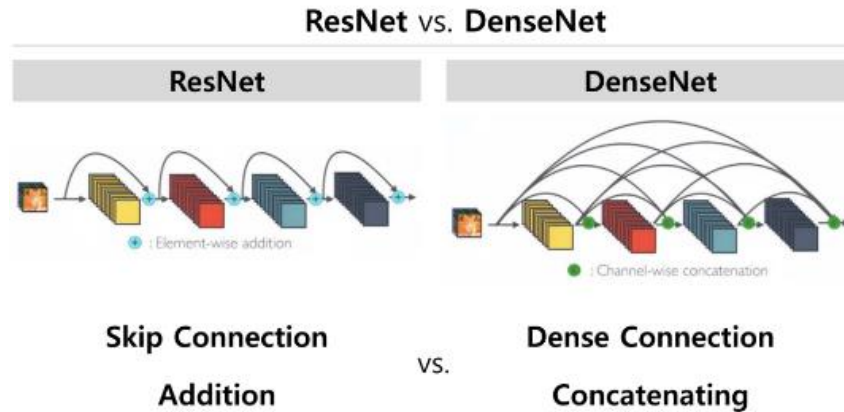


[그림 17.] Sample Image of DQN

### 5-4. Developing Deep-Q-Network (DQN)

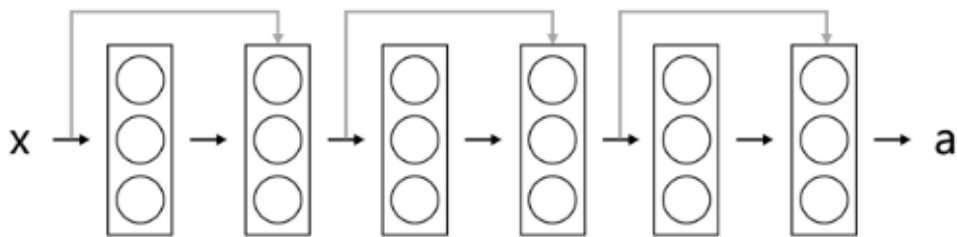
에피소드가 길수록 완전한 역전파가 이루어지기까지 오랜 시간 학습이 필요할 것이다. 이는 다시 말해 초기에는 활용보다 탐사가, 후반에는 탐사보다는 활용이 중점적으로 이루어질 것이라는 것을 의미할 것이다. 학습이 종료된 다면, 다양한 탐사가 이루어졌기 때문에 초반 정책에 강한 모델이 만들어질 것이다.

한편으로 역전파는 기울기 소실 문제를 갖고 있다. 가치 학습은 미래 보상에 대해 감가율을 적용하기 때문에 에피소드가 길면 길수록 현재 상태가 먼 미래에 얼마나 영향을 갖는지 학습하기가 어렵다. 신경망 구조에는 Skip Connection이라는 전략이 있다. 특정 레이어의 Output을 몇 단계 건너 뛰어 다음 레이어 Input에 더해주어 학습 성능을 높이는 전략이다. Cnn의 DenseNet의 경우에는, 아예 현재 레이어의 Output을 앞으로의 모든 레이어의 Input으로 넣어버린다.



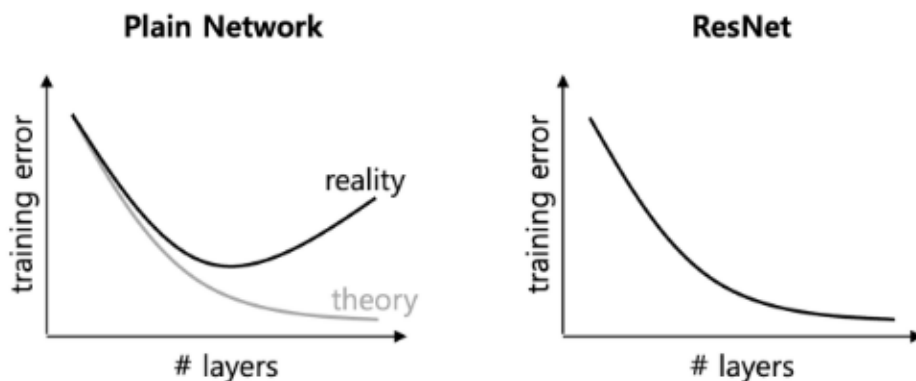
[그림 18.] ResNet 과 DenseNet 의 비교

ResNet(Residual Neural Network)은 딥러닝 모델의 학습 속도를 향상시키기 위해 제안된 신경망 구조이다. 일반적인 Convolution 신경망과 달리 잔여 연결(Residual Connection)을 사용하여 학습 과정에서 발생하는 정보 손실을 최소화한다. 잔여 연결은 이전 레이어의 출력을 직접 다음 레이어의 입력에 연결하는 방식이며, 이를 통해 모델이 더 깊은 구조를 유지하면서도 학습 효율성이 높아지게 된다



[그림 19.] ResNet 에서의 Residual Connection

이론상 신경망의 층이 깊어질수록 Training Error가 감소한다. 하지만 실제로는 Residual Block이 없는 "Plain Network"에서 신경망의 층이 깊어지면 Training Error가 감소하다가 다시 증가하는 경향을 보인다. 반면에 ResNet은 신경망의 층이 깊어지더라도 이론대로 Training Error가 계속 감소한다.



[그림 20.] Plain Network와 ResNet에서의 신경망의 층에 따른 Training Error 비교

모델로는 직선의 양 끝을 점령하는 것이 중요하다. 따라서 이러한 공간적 정보를 가진 잠재 변수를 추출하기 위해 CNN 기반 가치 신경망 설계가 좋을 것으로 보였다. 게임 종료 조건에 따라, 각 상태에서 받을 수 있는 보상은 현재 자신의 돌 색 개수이다. 하지만 정말 중요한 것은 보드의 ‘끝’을 차지하여 미래에 큰 보상을 얻을 수 있다는

것을 학습시켜야 했다. 우리는 이를 구현하기 위해 ResNet 아이디어를 사용하기로 해보았다.

**“우리는 DQN과 ResNet을 함께 넣어 모델로 학습시키는 모델을 구현하기로 하였다.”**

## 6. Deep Q\_ResNet Code

수업시간에 배운 내용을 활용하여 우선적으로 Deep Q-Learning을 구현하였다. 그 후 신경망에 Residual Block을 포함한 네트워크로 구성하였다. 우리가 구현한 Deep Q-ResNet의 핵심 기능들을 소개하겠다.

```
1 def _build_model(self):
2     state_input = layers.Input(shape=self.state_shape)
3     turn_input = layers.Input(shape=(1,), dtype='float32')
4     # turn_input을 3차원 텐서로 변환
5     turn_info = layers.RepeatVector(self.state_shape[0] * self.state_shape[1])(turn_input)
6     turn_info = layers.Reshape((self.state_shape[0], self.state_shape[1], 1))(turn_info)
7
8     # 상태 텐서와 턴 정보를 결합
9     combined_input = layers.Concatenate(axis=-1)([state_input, turn_info])
10    x = layers.Conv2D(64, kernel_size=7, strides=2, padding='same')(combined_input)
11    x = layers.BatchNormalization()(x)
12    x = layers.Activation('relu')(x)
13    x = layers.MaxPooling2D(pool_size=3, strides=2, padding='same')(x)
```

[그림 21.] DQN에서의 Input Layer 입력 과정

우선 상태와 차례 정보를 입력 받는다. 첫 Convolution Layer에서는 64개의 필터를 사용하여 7\*7 Kernel Size, Stride=2로 다운 샘플링한다. 그 후 Batch Normalization을 진행한다. 활성화 함수로는 ReLU를 사용한다. 3\*3 Pooling과 Stride=2를 통해 다운 샘플링을 진행한다.

```
1 # Residual block 1
2 for _ in range(3):
3     shortcut = x
4     x = layers.Conv2D(64, kernel_size=3, padding='same')(x)
5     x = layers.BatchNormalization()(x)
6     x = layers.Activation('relu')(x)
7     x = layers.Conv2D(64, kernel_size=3, padding='same')(x)
8     x = layers.BatchNormalization()(x)
9     x = layers.add([x, shortcut])
10    x = layers.Activation('relu')(x)
```

[그림 22.] 첫 번째 Residual Block 코드

계속해서 위 build\_model 함수 안에 있는 내용들이다. 첫 번째 Residual Block을 설정하는 과정이다. 첫 번째 컨볼루션 > 배치 정규화 > ReLU 활성화 > 두 번째 컨볼루션 > 배치 정규화 > 잔차 연결 > ReLU 활성화 과정을 거친다.



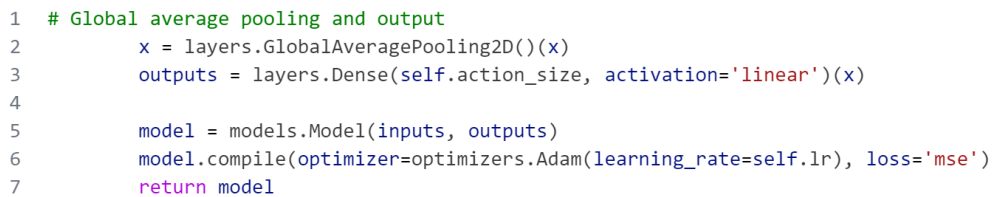
```

1 # Residual block 2
2     for i in range(4):
3         shortcut = x
4         if i == 0: # 첫 번째 블록에서만 채널 수가 변경되므로, 이를 맞추기 위해 1x1 Convolution 사용
5             shortcut = layers.Conv2D(128, kernel_size=1, padding='same')(shortcut)
6             shortcut = layers.BatchNormalization()(shortcut)
7
8         x = layers.Conv2D(128, kernel_size=3, padding='same')(x)
9         x = layers.BatchNormalization()(x)
10        x = layers.Activation('relu')(x)
11        x = layers.Conv2D(128, kernel_size=3, padding='same')(x)
12        x = layers.BatchNormalization()(x)
13        x = layers.add([x, shortcut])
14        x = layers.Activation('relu')(x)

```

[그림 23.] 두 번째 Residual Block 코드

첫번째 블록에서는 입력 채널과 출력 수가 다르기 때문에 1\*1 Convolution을 통해 채널 수를 맞춰준다. 그 후 과정은 동일하다. 첫 번째 컨볼루션 > 배치 정규화 > ReLU 활성화 > 두 번째 컨볼루션 > 배치 정규화 > 잔차 연결 > ReLU 활성화를 차례로 진행한다.



```

1 # Global average pooling and output
2     x = layers.GlobalAveragePooling2D()(x)
3     outputs = layers.Dense(self.action_size, activation='linear')(x)
4
5     model = models.Model(inputs, outputs)
6     model.compile(optimizer=optimizers.Adam(learning_rate=self.lr), loss='mse')
7     return model

```

[그림 24.] Outputs 변환 과정

GlobalAveragePooling2D를 통해 각 Feature Map의 평균을 계산하여, 입력을 하나의 벡터로 변환한다. 그 후 행동의 개수만큼 뉴런을 가지는 Dense Layer를 통해 Q 값을 출력한다. 이제 Inputs와 Outputs를 연결하여 모델을 생성한다. Adam 옵티마이저와 MSE 손실함수를 사용하여 모델 컴파일을 진행하였다. 그 후 수업시간에 배운 내용을 토대로 Deep Q-Learning을 진행하였다.



[그림 25.] Deep-Q-Learning 과정

우리는 Deep Q-Learning과 ResNet을 합친 우리의 모델을 DeepQResNet을 부르기로 하였다. 이렇게 DeepQResNet의 구성에 대해 알아보았다. 이젠 이 코드를 토대로 실제 학습을 진행해보자.

## 7. How to Train and Play Our Code

학습을 진행하며 모델로 게임을 플레이해보겠다.



[그림 26.] Gym 초기화 코드

우선 모델로 플레이를 하기 위해 기본 세팅을 불러온다. 이제 학습을 진행해야 한다. 보고서 5번 Learning Algorithms파트에서 핵심 전략으로 UCT를 사용하기로 했다. 코드를 구현하던 중 인공지능 자동 모드에서 흰 색, 검은색 돌을 각각 학습시키면 어떨까 하는 생각이 들었다. 따라서 이를 훈련하는 경쟁학습도 구현하였다. 최종적으로 기본 Greedy Policy Learning을 포함하여 훈련 과정을 총 3개 제시한다.

1. DQN + Greedy Policy Learning
2. DQN + UCT Policy Learning
3. (DQN + UCT Policy) + (DQN + UCT Policy) 경쟁 학습

1번 학습 진행 후 모델 저장. 모델 로드 후 플레이.

2번 학습 진행 후 모델 저장. 모델 로드 후 플레이

3번 학습 진행 후 모델 저장. 모델 로드 후 플레이

각각에 대해 학습을 진행한 후 플레이를 통해 성능을 비교해본다.



## 7.1 DQN & Greedy Policy Learning

```
1 env.metadata['autoplay'] = True
2 env.metadata['render_fps'] = 1000000
3
4
5 obs, reward, bdone, _, info = env.reset()
6 dqn = DQN(state_shape, env.action_space.n)
7 version = -1
8 #dqn.load( f"model/Greedy/BlackModel_{version}.weights.h5")
9 count = version + 1
10 def GetPolicy(bdone, wdone, turn, obs, actions):
11     if(not actions):
12         return None
13     if(turn==1):
14         if(bdone):
15             return None
16     else:
17         if(wdone):
18             return None
19     return dqn.EstimatePolicy(obs, turn, actions)
20 def InsertBuffer(turn, oldobs, action, reward, obs, done, actions):
21     dqn.InsertBuffer(oldobs, action, reward, obs, done, actions, turn)
22
23 for episode in range(1,500):
24     bdone = False
25     wdone = False
26     obs, reward, done, _, info = env.reset()
27     steps = 0
28     while (not bdone) or (not wdone):
29         actions = info['action']
30         turn = info['turn']
31         oldobs = obs
32         action = GetPolicy(bdone, wdone, turn, obs, actions)
33         obs, reward, done, _, info = env.step(action)
34         if done:
35             if info['turn'] == 2:
36                 wdone = True
37             else:
38                 bdone = True
39         if(action is not None):
40             InsertBuffer(turn, oldobs, action, reward, obs, done, info['action'])
41         steps += 1
42     loss = dqn.train()
43     if loss is not None:
44         print(f"Episode {episode + 1},=Loss: {loss}\n")
45     if(episode % 30 == 0):
46         print("update Target Model")
47         dqn.update_target_model()
48     if(episode % 100 == 0):
49         print('save \n')
50         dqn.save(f"model/Greedy/Model_{count}.weights.h5")
51         count+=1
```

Episode 4,=Loss: 5.406990051269531

Episode 6,=Loss: 5.819243907928467

Episode 8,=Loss: 6.143962860107422

Episode 10,=Loss: 5.630764007568359

Episode 12,=Loss: 5.221203804016113

Episode 14,=Loss: 5.108561038970947

Episode 16,=Loss: 5.383352756500244

Episode 19,=Loss: 5.265793800354004

Episode 21,=Loss: 5.028144836425781

Episode 23,=Loss: 4.845694541931152

Episode 25,=Loss: 4.684718608856201

Episode 27,=Loss: 4.551350116729736

Episode 29,=Loss: 4.439423561096191

...

Episode 49,=Loss: 4.533563137054443

Episode 51,=Loss: 4.471510410308838

[그림 27.] DQN & Greedy Policy Learning & Training Results

## 7.2 DQN & UCT Policy Learning

```
1 env.metadata['autoplay'] = True
2 env.metadata['render_fps'] = 150000
3 obs, reward, bdone, _, info = env.reset()
4 dqn = DQN(state_shape, env.action_space.n)
5
6 version = -1
7 #dqn.load( f'model/UCT/Model_{version}.weights.h5')
8 Count = version + 1
9 def GetPolicy(bdone, wdone, turn, env, obs, actions):
10     if(not actions):
11         return None
12     if(turn==1):
13         if(bdone):
14             return None
15     else:
16         if(wdone):
17             return None
18     return dqn.BehaviorPolicy(env, obs, turn, actions)
19 def InsertBuffer(turn, oldobs, action, reward, obs, done, actions):
20     dqn.InsertBuffer(oldobs, action, reward, obs, done, actions, turn)
21 for episode in range(1, 400):
22     bdone = False
23     wdone = False
24     obs, reward, done, _, info = env.reset()
25     steps = 0
26     while (not bdone) or (not wdone):
27         actions = info['action']
28         turn = info['turn']
29         oldobs = obs
30         action = GetPolicy(bdone, wdone, turn, env, obs, actions)
31         obs, reward, done, _, info = env.step(action)
32         if done:
33             if info['turn'] == 2:
34                 wdone = True
35             else:
36                 bdone = True
37         if(action is not None):
38             InsertBuffer(turn, oldobs, action, reward, obs, done, actions)
39         steps += 1
40     loss = dqn.train()
41     if loss is not None:
42         print(f"Episode {episode + 1}, Loss: {loss}\n")
43     if(episode % 30 == 0):
44         print("update Target Model")
45         dqn.update_target_model()
46     if(episode % 100 == 0):
47         print('save ')
48         dqn.save(f'model/UCT/Model_{Count}.weights.h5')
49         Count+=1
```

Episode 4, BlackLoss: 7.89846134185791

Episode 6, BlackLoss: 7.028895854949951

Episode 8, BlackLoss: 6.10620641708374

Episode 10, BlackLoss: 5.578690528869629

Episode 103, BlackLoss: 4.443989276885986

Episode 105, BlackLoss: 4.415994644165039

Episode 107, BlackLoss: 4.414387226104736

Episode 109, BlackLoss: 4.389989852905273

Episode 111, BlackLoss: 4.412949562072754

Episode 113, BlackLoss: 4.387295722961426

Episode 115, BlackLoss: 4.430821895599365

Episode 118, BlackLoss: 4.428666114807129

Episode 120, BlackLoss: 4.402266025543213

Episode 122, BlackLoss: 4.380159854888916

Episode 124, BlackLoss: 4.378911018371582

Episode 126, BlackLoss: 4.397068500518799

...

Episode 150, BlackLoss: 4.3450093269348145

Episode 152, BlackLoss: 4.359941482543945

[그림 28.] DQN & UCT Policy Learning & Training Results

## 7.3 AI Competing Training With DQN

```
1 env.metadata['autoplay'] = True
2 env.metadata['render_fps'] = 150000
3 obs, reward, bdone, _, info = env.reset()
4 state_shape = (8, 8, 1) # Adding the channel dimension
5 Blackdqn = DQN(state_shape, env.action_space.n)
6 Whitedqn = DQN(state_shape, env.action_space.n)
7
8 version = -1
9 # Blackdqn.load(f"model/UCT/Black/Model_{version}.weights.h5")
10 # Whitedqn.load(f"model/UCT/White/Model_{version}.weights.h5")
11 BlackmodelCount = version + 1
12 WhitemodelCount = version + 1
13 def GetPolicy(bdone, wdone, turn, env, obs, actions):
14     if(not actions):
15         return None
16     if(turn==1):
17         if(bdone):
18             return None
19         return Blackdqn.BehaviorPolicy(env, obs, turn, actions)
20     else:
21         if(wdone):
22             return None
23         return Whitedqn.BehaviorPolicy(env, obs, turn, actions)
24 def InsertBuffer(turn, oldobs, action, reward, obs, done, actions):
25     Blackdqn.InsertBuffer(oldobs, action, reward, obs, done, actions, turn)
26     Whitedqn.InsertBuffer(oldobs, action, reward, obs, done, actions, turn)
27 for episode in range(1, 10000):
28     bdone = False
29     wdone = False
30     obs, reward, done, _, info = env.reset()
31     steps = 0
32     while (not bdone) or (not wdone):
33         actions = info['action']
34         turn = info['turn']
35         oldobs = obs
36         action = GetPolicy(bdone, wdone, turn, env, obs, actions)
37         obs, reward, done, _, info = env.step(action)
38         if done:
39             if info['turn'] == 2:
40                 wdone = True
41             else:
42                 bdone = True
43             if(action is not None):
44                 InsertBuffer(turn, oldobs, action, reward, obs, done, actions)
45             steps += 1
46         blakloss = Blackdqn.train()
47         whiteloss = Whitedqn.train()
48         if blakloss is not None:
49             print(f"Episode {episode + 1}, BlackLoss: {blakloss}\n")
50             Blackdqn.update_target_model()
51         if whiteloss is not None:
52             print(f"Episode {episode + 1}, WhiteLoss: {whiteloss}\n")
53             Whitedqn.update_target_model()
54         if(episode % 30 == 0):
55             print("update Target Model")
56             Blackdqn.update_target_model()
57             Whitedqn.update_target_model()
58         if(episode % 100 == 0):
59             print('save ')
60             Whitedqn.save(f"model/BlackModel_{WhitemodelCount}.weights.h5")
61             Blackdqn.save(f"model/WhiteModel_{BlackmodelCount}.weights.h5")
62             BlackmodelCount+=1
63             WhitemodelCount+=1
```

Episode 8, BlackLoss: 1.1248804330825806

Episode 8, WhiteLoss: 1.2767577171325684

Episode 10, BlackLoss: 0.9338687658309937

Episode 10, WhiteLoss: 1.0718914270401

Episode 12, BlackLoss: 0.7854718565940857

Episode 12, WhiteLoss: 0.9076473116874695

Episode 14, BlackLoss: 0.6724572777748108

Episode 14, WhiteLoss: 0.7806628346443176

---

Episode 723, BlackLoss: 0.03028571978211403

Episode 723, WhiteLoss: 0.023801758885383606

Episode 725, BlackLoss: 0.030220387503504753

Episode 725, WhiteLoss: 0.023865556344389915

Episode 727, BlackLoss: 0.030160322785377502

Episode 727, WhiteLoss: 0.023975661024451256

Episode 729, BlackLoss: 0.03010265901684761

Episode 729, WhiteLoss: 0.024063169956207275

[그림 29.] AI Competing Training & Training Results

총 3번의 과정을 통해 학습은 모두 끝났다. test.ipynb의 마지막 코드를 실행하면 훈련이 된 AI를 상대로 게임을 할 수 있게 된다. BlackPlay의 True/False 값 변경을 통해 플레이어의 선공/후공의 여부도 결정할 수 있게 하였다.

## 8. Code Trouble & Solution

**Trouble 1** 코드에 DQN + ResNet을 추가하기 전 최종적으로 코드에 이상이 없는지 확인해보도록 했었다. 1인 모드로 모델을 플레이 하던 중 플레이어가 돌을 둘 곳이 없으면 게임이 종료하는 문제가 발생했다. 이를 어떻게 해결할지 고민하였다. OthelloEnv의 코드를 살펴본 후 step 함수를 수정하였다. 그 결과 “**Player1이 돌 곳이 없으면 그 상태에서 게임종료**”에서 “**Player1이 돌을 둘 곳이 없으면 상대방(Player2)턴으로 넘어가 상대가 게임을 종료할 때까지 턴을 둔다**” 로 구현하였다.

**Trouble 2** DQN을 구현하며 코드를 작성하였다. 하지만 Reward 값 출력 기능이 없었기에 실제로 학습이 진행되는지 여부를 판단할 수 없었다. Episode를 돌리며 Loss Function을 출력하게 함으로써 AI가 개선되고 있음을 확인하였다.

**Trouble 3** 코드를 구현 완료하고 학습을 시키는 도중에 Loss가 폭발하는 현상이 일어났다.

```
update Target Model
Episode 1593, BlackLoss: 3.4354095458984375

Episode 1596, WhiteLoss: 2312.12353515625
```

[그림 30.] Loss 급증하는 모습

학습 도중, Q\_Value도 증가하는 현상이 발생하였다.

```
q_values: array([ 0.07865731,  0.3346547 ,  0.29335064,  0.13002545,  0.2909158
q_values: array([-746.7987 ,  740.61926, -778.5077 , -769.3439 ,  733.5575
```

[그림 31.] 증가하는 Q\_value

이러한 문제가 왜 일어날까 고민을 하던 중 Reward값이 너무 커서 이런 오류가 날까 하는 생각이 들었다. 그래서 **Reward의 최댓값을 1로 변경**하였다.

```
1 for i in range(self.batch_size):
2     if done[i]:
3         target[i][actions[i]] = rewards[i] /100.0
4     else:
5         target[i][actions[i]] = rewards[i] /100.0 + self.gamma * np.amax(Qvalue[i][valid_actions[i]])
6     self._InsertHashTable(states[i])
```

[그림 32.] Reward의 최댓값 1로 변경

또한 모델 구조를 변경하였다.

## Before

```
1 def _build_model(self):
2     inputs = layers.Input(shape=self.state_shape)
3     x = layers.Conv2D(64, kernel_size=7, strides=2, padding='same')(inputs)
4     x = layers.BatchNormalization()(x)
5     x = layers.Activation('relu')(x)
6     x = layers.MaxPooling2D(pool_size=3, strides=2, padding='same')(x)
```

## After

```
1 def _build_model(self):
2     state_input = layers.Input(shape=self.state_shape)
3     turn_input = layers.Input(shape=(1,), dtype='float32')
4     # turn_input을 3차원 텐서로 변환
5     turn_info = layers.RepeatVector(self.state_shape[0] * self.state_shape[1])(turn_input)
6     turn_info = layers.Reshape((self.state_shape[0], self.state_shape[1], 1))(turn_info)
```

[그림 33.] DQN 기본 모델 구조 변경

기존 코드에서는 상태만 입력 받았다. 수정된 코드에서는 상태와 차례 정보를 같이 받는 것으로 변경하였다. 그 후 ResNet과정은 동일하다. 그 후 다시 학습을 진행하였다.

```
Episode 4, BlackLoss: 1.614793300628662 Episode 712, BlackLoss: 0.0305742509663105
Episode 4, WhiteLoss: 1.7577934265136719 Episode 712, WhiteLoss: 0.02350020967423916
Episode 6, BlackLoss: 1.349267601966858 Episode 714, BlackLoss: 0.030515674501657486
Episode 6, WhiteLoss: 1.512926697731018 Episode 714, WhiteLoss: 0.023533260449767111
Episode 716, BlackLoss: 0.030455823987722397
Episode 716, WhiteLoss: 0.023590989410877228
Episode 719, BlackLoss: 0.030405746772885321
Episode 719, WhiteLoss: 0.02366810478270054
Episode 721, BlackLoss: 0.030348286032676697
Episode 721, WhiteLoss: 0.02372787706553936
```

[그림 34.] 모델 구조 변경 후 학습 결과

초기 Loss 값에 비해 학습이 진행됨에 따라 Loss 값이 수렴되어 간다는 것을 볼 수 있었다.

## 9. Project Review

**Team:** 게임 모델을 만들고 학습하는 과정에서 많은 어려움이 있었다. 그렇지만 지속적인 회의와 아이디어들을 통해 해결해 나갈 수 있었다. 혼자가 아닌 여럿이 문제를 해결하니 더욱 효과적으로 프로젝트를 진행할 수 있었다. 프로젝트를 진행하며 아쉬웠던 점은 훈련에 시간이 오래 걸려 완벽하게 학습을 시키지 못한 것이다. 또한 구현한 모델로 규칙에서 현재 한쪽 플레이어가 둘 곳이 없으면 상대 플레이어가 게임이 종료될 때까지 계속 두게 되는데 이 규칙을 나중에 수정해보고 싶다. 학습을 시키면 Loss값 출력을 통해 훈련이 되고 있다는 것을 볼 수 있었다. 다만 팀의 모델로 실력이 좋지 못해 Random 모델을 상대했을 때와 Trained 모델을 상대했을 때의 차이를 체감할 수는 없던 점이 아쉬웠다.

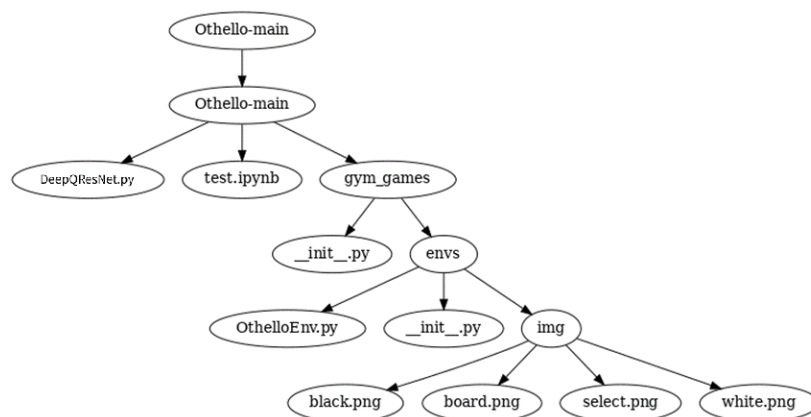
**박윤준:** 이번 프로젝트를 통해 ResNet을 공부하고 코드로 구현해보며 무조건 성능이 좋은(sota) 모델보다 내가 해결하고자 하는 문제에 맞는 Neural Network를 정해서 구현하는 것이 중요하다는 것을 깨달았습니다. 인공지능의 이해 수업때는 Kamisado 게임에 minimax 알고리즘을 적용했었는데 이번에는 Othello에 DQN+ResNet을 적용해보며 다양한 게임에서 여러 방법으로 AI Player를 구현할 수 있음을 배웠습니다.

**양진우:** 하나의 모델(게임)에 대하여 문제 분석 보고서를 쓰는 것을 시작으로, 심층적으로 분석해보고 실제로 구현해보는 것은 처음이었습니다. 단순한 모델을 구현하는 것에서 그치지 않고 팀만의 아이디어(ResNet+DQN)를 적용한 인공지능 학습 알고리즘을 탑재하는 색다른 경험이었습니다. 코드를 구현하고 보고서를 작성하며 한 학기 동안 배웠던 내용들을 다시 복습해볼 수 있었습니다.

**홍성필:** 강화학습으로 모델을 DQN으로 구현한 경험은 흥미롭고 도전적이었습니다. 코드를 작성하며 문제를 해결하고 성능을 향상시키는 과정에서 많은 것을 배울 수 있었습니다. 또한, 강화 학습에는 컴퓨팅 파워를 갖추는 것이 매우 중요하다는 것을 깨달았습니다. 프로젝트를 통해 어떻게 다양한 게임에 강화학습을 접목시킬 수 있을지 관심을 가지게 되었습니다.

## 10. Our Code

<https://github.com/tjdvlf2880/Othello>



[그림 35.] 코드 구성도

## 11. References

[1]. Andrew Ng - C4W2L03 Resnets

[2]. Andrew Ng - C4W2L04 Why ResNets Work

[3]. 202410HY23794 기계학습 RL03. Monte-Carlo RL (slides)

[4]. 202410HY23794 기계학습 RL05. Deep RL (slides) (UPDATED)

[5]. Sunghyun Kim, & Youngwan Cho (2020). An Artificial Intelligence Othello Game Agent Using CNN Based MCTS and Reinforcement Learning. Journal of Korean Institute of Intelligent Systems, 30(1), 40-46, 10.5391/JKIIS.2020.30.1.40

[6]. He, K., Zhang, X., Ren, S., & Sun, J. (2015). Deep residual learning for image recognition. *arXiv*.  
<https://doi.org/10.48550/arXiv.1512.03385>