Artificial Intelligence and Machine Learning Fundamentals

Activity 4: Connect Four

This section will practice using the **EasyAI** library and develop a heuristic. We will be using connect four game. The game board is seven cells wide and cells high. When you make a move, you can only select the column in which you drop your token.

Then gravity pulls the token down to the lowest possible empty cell. Your objective is to connect four of your own tokens horizontally, vertically, or diagonally, before your opponent does this, or you run out of empty spaces. The rules of the game can be found at: https://en.wikipedia.org/wiki/Connect_Four

1. Let's set up the TwoPlayersGame framework:

```
from easyAI import TwoPlayersGame
from easyAI.Player import Human_Player
class ConnectFour(TwoPlayersGame):
def __init__(self, players):
self.players = players
def possible_moves(self):
return []
def make_move(self, move):
return
def unmake_move(self, move):
# optional method (speeds up the AI)
return
def lose(self):
return False
def is_over(self):
return (self.possible_moves() == []) or self.lose()
def show(self):
print ('board')
def scoring(self):
return -100 if self.lose() else 0
if __name__ == "__main__":
from easyAI import AI_Player, Negamax
ai_algo = Negamax(6)
```

2. We can leave a few functions from the definition intact. We have to implement the following methods:

```
__init__
possible_moves
make_move
unmake_move (optional)
lose
show
```

3. We will reuse the basic scoring function from tic-tac-toe. Once you test out the game, you will see that the game is not unbeatable, but plays surprisingly well, even though we are only using basic heuristics.

4. Let's write the init method. We will define the board as a one-dimensional list, similar to the tic-tac-toe example. We could use a two-dimensional list too, but modeling will not get much easier or harder. Beyond making initializations like we did in the tic-tac-toe game, we will work a bit ahead. We will generate all of the possible winning combinations in the game and save them for future use:

```
def __init__(self, players):
self.players = players
# 0 1 2 3 4 5 6
# 7 8 9 10 11 12 13
# ...
# 35 36 37 38 39 40 41
self.board = [0 for i in range(42)]
self.nplayer = 1 # player 1 starts.
def generate_winning_tuples():
tuples = []
# horizontal
tuples += [
list(range(row*7+column, row*7+column+4, 1))
for row in range(6)
for column in range(4)]
# vertical
tuples += [
list(range(row*7+column, row*7+column+28, 7))
for row in range(3)
for column in range(7)
]
# diagonal forward
tuples += [
list(range(row*7+column, row*7+column+32, 8))
for row in range(3)
for column in range(4)
]
# diagonal backward
tuples += [
list(range(row*7+column, row*7+column+24, 6))
for row in range(3)
for column in range(3, 7, 1)
]
return tuples
self.tuples=generate_winning_tuples()
```

5. Let's handle the moves. The possible moves function is a simple enumeration. Notice we are using column indices from 1 to 7 in the move names, because it is more convenient to start column indexing with 1 in the human player interface than with zero. For each column, we check if there is an unoccupied field. If there is one, we will make the column a possible move.

```
def possible_moves(self):
return [column+1
for column in range(7)
if any([
self.board[column+row*7] == 0
```

```
for row in range(6)
])
]
```

6. Making a move is similar to the possible moves function. We check the column of the move, and find the first empty cell starting from the bottom. Once we find it, we occupy it. You can also read the implementation of the dual of the make_move function: unmake_move. In the unmake_move function, we check the column from top to down, and we remove the move at the first non-empty cell. Notice we rely on the internal representation of easyAi so that it does not undo moves that it had not made. Otherwise, this function would remove a token of the other player without checking whose token got removed.

```
def make_move(self, move):
column = int(move) - 1
for row in range(5, -1, -1):
index = column + row*7
if self.board[index] == 0:
self.board[index] = self.nplayer
return
def unmake_move(self, move):
# optional method (speeds up the AI)
column = int(move) - 1
for row in range(6):
index = column + row*7
if self.board[index] != 0:
self.board[index] = 0
return
```

7. As we already have the tuples that we have to check, we can mostly reuse the lose function from the tic-tac-toe example.

```
def lose(self):
return any([all([(self.board[c] == self.nopponent)
for c in line])
for line in self.tuples])
def is_over(self):
return (self.possible_moves() == []) or self.lose()
```

8. Our last task is the show method that prints the board. We will reuse the tic-tac-toe implementation, and just change the variables. def show(self):

```
print('\n'+'\n'.join([
' '.join([['.', 'O', 'X'][self.board[7*row+column]]
for column in range(7)]
)
for row in range(6)])
)
```

Now that all functions are complete, you can try out the example. Feel free to play a round or two against the opponent. You can see that the opponent is not perfect, but it plays reasonably well. If you have a strong computer, you can increase the parameter of the Negamax algorithm. I encourage you to come up with a better heuristic.