NEURAL NETWORKS

AND GAMES: OTHELLO

Samuel Effendy
Vira Kasprova

Othello Reinforcement Learning

Reinforcement learning has been shown to be an effective method to solve complex problems in the domain of board games, as it trains the agent to maximize its expected reward by learning the value of each action through entering specific states. In this paper, we explore its application on Othello, a two-player deterministic, zero-sum, perfect information game.

Theoretical Background

Othello.

Othello is a perfect information game because every player is perfectly informed of every event in the game, including the initialization of the board. It is also zero-sum, since every gained advantage of a player is equal to the gained disadvantage of their opponent, e.g. every tile gained by a player is lost to another player. Next, it is deterministic, because there are no random elements in its rules. This game is a perfect test bed for reinforcement algorithms like Q-learning, since there is a clear distinction between reward and consequences as well as a numeric value for each action to train the agent.

The board is initially empty but for the four center squares, which feature discs on the two squares on the major diagonal ([4][4] and [5][5]) and the two squares on the opposing diagonal, respectively [4][5] and [5][4].

The player who moves first is referred to as "Black" and the player who moves second is referred to as "White". A disc must be placed on an empty square in order for there to be a sequence of one or more discs belonging to the opponent, followed by one's own disc, in at least one direction (horizontally, vertically, or diagonally) from the square played on. In such a series, the opponent's discs are flipped and changed to one's own color.

Othello's state space has a maximum of 60 movements and a dimension of around 1028. Two players use 64 two-sided discs that are black on one side and white on the other to play Othello on an 8 by 8 board. The discs are placed on the board with the white side up by one player and the black side up by the other player.

Parameterization of Othello.

We chose to write our parameterization in Python due to its accessibility and library support for machine learning. To represent the board in our version, we will utilize an 8 by 8 character array. Instead of using black and white pieces, we will use Xs and Os. The lower left corner is at coordinate [0][0], while the upper right corner is at coordinate [7][7].

```
  def __init__(self,p1,p2):
    self.board = np.full((8,8),".")
    #set four entries to be two X's and two O's
    self.board[3][4] = "X";
    self.board[4][3] = "X";
    self.board[3][3] = "O";
    self.board[4][4] = "O";
    self.p1 = p1
    self.p2 = p2
    self.isEnd = False
    self.boardHash = None
    self.playerSymbol = "X" #first player symbol
```

Python code

Since the rules for movement in Othello are complicated, we broke the movements down into three functions. The validMove() checks if the move is valid or not. The move function takes as input the board, the XY coordinate to place the piece, and the piece we are placing (X or O). If a piece can be flipped in any of the eight directions and the space on the board is unoccupied, the move is valid. We check all adjacent cells to our current location by calling checkFlip() function for left, right, down, up, down-left, down-right, up-left, and up-right.

```
def validMove(boardList, x, y, piece):
  # Check that the coordinates are empty
  if boardList[x][y] != '.':
    return False
  # Figure out the character of the opponent's piece
  opponent = 'O'
  if piece == 'O':
    opponent = 'X'
  # If we can flip in any direction, it is valid
  # Check to the left
  if checkFlip(boardList, x - 1, y, -1, 0, piece, opponent):
    return True
  # Check to the right
  if checkFlip(boardList, x + 1, y, 1, 0, piece, opponent):
    return True
  # Check down
  if checkFlip(boardList, x, y - 1, 0, -1, piece, opponent):
    return True
  # Check up
  if checkFlip(boardList, x, y + 1, 0, 1, piece, opponent):
    return True
  # Check down-left
  if checkFlip(boardList, x - 1, y - 1, -1, -1, piece, opponent):
    return True
  # Check down-right
  if checkFlip(boardList, x + 1, y - 1, 1, -1, piece, opponent):
    return True
  # Check up-left
  if checkFlip(boardList, x - 1, y + 1, -1, 1, piece, opponent):
    return True
  # Check up-right
  if checkFlip(boardList, x + 1, y + 1, 1, 1, piece, opponent):
    return True

  return False # If we get here, we didn't find a valid flip direction
```

Python code

We use makeMove() function to make an actual move. We check all adjacent cells to our current location by calling checkFlip() function for left, right, down, up, down-left, down-right, up-left, and up-right. If the flip is possible we call flipPieces() fucntion to flip the value of the opponent piece to the player piece and vice versa.

```
def makeMove(self,boardList, x, y, piece): #this function is copied over to state class
    # Put the piece at x,y
    boardList[x][y] = piece
    # Figure out the character of the opponent's piece
    opponent = 'O'
    if piece == 'O':
      opponent = 'X'

    # Check to the left
    if checkFlip(boardList, x - 1, y, -1, 0, piece, opponent):
      flipPieces(boardList, x - 1, y, -1, 0, piece, opponent)
    # Check to the right
    if checkFlip(boardList, x + 1, y, 1, 0, piece, opponent):
      flipPieces(boardList, x + 1, y, 1, 0, piece, opponent)
    # Check down
    if checkFlip(boardList, x, y-1, 0, -1, piece, opponent):
      flipPieces(boardList, x, y-1, 0, -1, piece, opponent)
    # Check up
    if checkFlip(boardList, x, y + 1, 0, 1, piece, opponent):
      flipPieces(boardList, x, y + 1, 0, 1, piece, opponent)
    # Check down-left
    if checkFlip(boardList, x-1, y - 1, -1, -1, piece, opponent):
      flipPieces(boardList, x-1, y - 1, -1, -1, piece, opponent)
    # Check down-right
    if checkFlip(boardList, x + 1, y - 1, 1, -1, piece, opponent):
      flipPieces(boardList, x + 1, y - 1, 1, -1, piece, opponent)
    # Check up-left
    if checkFlip(boardList, x - 1, y + 1, -1, 1, piece, opponent):
      flipPieces(boardList, x - 1, y + 1, -1, 1, piece, opponent)
    # Check up-right
    if checkFlip(boardList, x + 1, y + 1, 1, 1, piece, opponent):
      flipPieces(boardList, x + 1, y + 1, 1, 1, piece, opponent)
```

Python code

This function has two helper functions. One function, checkFlip(), checks if an existing disk on the board can be flipped in one of eight directions (left, right, up, down, up-left, up-right, down-left, down-right), with the condition that an opponent disk is found between our current and next piece. It returns true or false.

```
def checkFlip(boardList, x, y, deltaX, deltaY, piece, opponent):
  if (x >= 0 and x < 8 and y >= 0 and y < 8):
    if(boardList[x][y] == opponent):
      x += deltaX
      y += deltaY
      while (x >= 0) and (x < 8) and (y >= 0) and (y < 8):
        if(boardList[x][y] == '.'):
          return False
        if(boardList[x][y] == piece):
          return True
  return False
```

Python code

This is then used in an if statement to decide whether or not the second helper function flipPieces() should be executed, with flipPieces() doing the actual flipping. It simply changes the value of the opponent piece to the player piece and vice versa.

```
def flipPieces(boardList, x, y, deltaX, deltaY, myPiece, opponentPiece):
  while boardList[x][y] == opponentPiece:
    boardList[x][y] = myPiece
    x += deltaX
    y += deltaY
```

Python code

The function that calculates the score is called score(). The function takes as input the board and the character (X or O). It loops through the board and counts the number of a particular character (X or O).

```
def score(boardList, piece):
  total = 0
  for x in range(8):
    for y in range (8):
      if boardList[x][y] == piece:
        total += 1
  return total
```

Python code

CHALLENGES

We had difficulties integrating Q-learning into our parameterized program. This is because our code was written to play Othello manually, without opponent AI. Furthermore, the structure of our code was divided into snippets containing separate functions, without grouping of similar functions under the agent and the state it inhabits. Upon consulting our advisor, we realized that we needed to have two classes and sort our functions beneath it.

Finding the next move of our game was challenging after modifying our code, since we weren't sure of which functions to implement inside a class, and which class. Eventually, we figured out that all three functions should be called in the state class, because we are checking for moves in each state that the agent enters.

```
def availablePositions(self):
    #this is the function that returns all possible moves from a particular board
    positions = []
    for i in range(8):
      for j in range(8):
        if validMove(self.board,i,j,self.playerSymbol):
          positions.append((i,j))
        print(validMove(self.board,i,j,self.playerSymbol), end = ' ')
    for i in positions:
     print(i, end = ' ')
    return positions
```

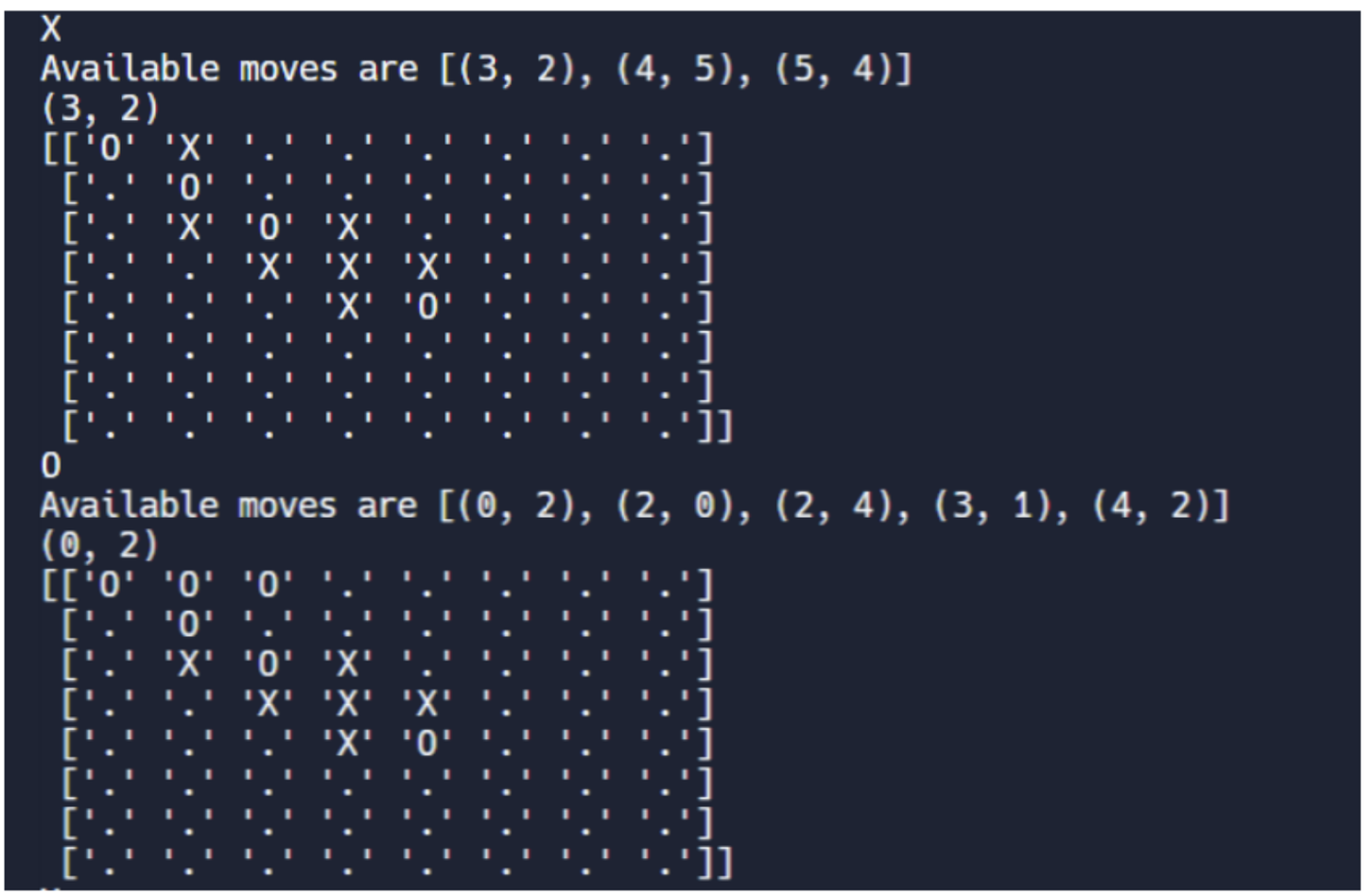
Python code

Checking if the game was finished was trivial in our original implementation, due to the rule that an Othello game ends if there are no empty tiles left, regardless of the scores of each player. That means that there could be a tie, and this was taken into consideration in our program. However, as we currently have no final snapshot of the game finished through implementation of Q-learning, we cannot say for sure.

METHOD

Initially, we were thinking of using an open-source version of the AlphaZero algorithm, but we chose Q-learning to train our program. In particular, we used a Tic-Tac-Toe with Q-learning examples provided by our advisors as a baseline. After doing our research, Q-learning is efficient because it doesn't require a model of the environment (model-free), saving us time in implementing our program.

Furthermore, Q-learning finds an optimal rule to maximize the reward for the agent in every state, by adding the maximum reward from future states to the reward it achieves in the current state. It is most suitable for board games like Othello, since the game has deterministic states and actions, and we can override old information with new information instantly (e.g. white replaces all black tiles in between, so history of black tiles is no longer saved.)



Game visualization and output

X and O are both played by the AI. Here, we see the AI choosing from available moves to either X or O for each turn.

Playing the game with two AI players using Q-learning requires us to automate checking for available positions, choosing the agent's actions, and updating and adding states. This is reflected in the function below.

```
  def play(self, rounds=10000):
    global player1wins
    global player2wins
    global ties
    ties = 0
    player1wins = 0
    player2wins = 0
    for i in range(rounds):
      if i % 1000 == 0:
        print("Round {}".format(i))
      while not self.isEnd:
        # Player 1
        positions = self.availablePositions()
        if len(positions) != 0:
          p1_action = self.p1.chooseAction(positions, self.board, self.playerSymbol)
          # take action and update board state
          self.updateState(p1_action)
          board_hash = self.getHash()
          self.p1.addState(board_hash)
        #Player 2
        self.playerSymbol = "O"
        positions = self.availablePositions()
        if len(positions) != 0:
          # Player 2
          p2_action = self.p2.chooseAction(positions, self.board, self.playerSymbol)
          self.updateState(p2_action)
          board_hash = self.getHash()
          self.p2.addState(board_hash)

        win = self.winner()
        if win is not None:
          #print("Winner is", self.defineWinner())
          self.defineWinner()
          self.showBoard()
          # ended with p1 either win or draw
          self.giveReward()
          self.p1.reset()
          self.p2.reset()
          self.reset()
          break
```

Python code

We wrote functions to test out simulations between random players, trained and random players, and between trained players. Then, we took the success rates of our simulations using the function below.

```
def percentage(numGames, player1wins, player2wins, ties):
  #global player1wins
  #global player2wins
  #global ties
  print("Player 1 won", player1wins, "times")
  print("Player 2 won", player2wins, "times")
  print("There were ties", ties, "times")
  print("Number of games:", numGames, "\n")

  percentOne = (player1wins / numGames) * 100
  percentTwo = (player2wins / numGames) * 100
  ties = (ties / numGames) * 100

  print("Player 1 percentage:", percentOne)
  print("Player 2 percentage:", percentTwo)
  print("Ties percentage:", ties)
```

Python code

```
Available moves are []
Winner is O!
[['O' 'O' 'O' 'O' 'O' 'O' 'O' 'O']
 ['O' 'O' 'O' 'O' 'O' 'O' 'X' 'O']
 ['O' 'O' 'O' 'O' 'O' 'X' 'O' 'O']
 ['O' 'O' 'O' 'X' 'X' 'O' 'X' 'O']
 ['O' 'O' 'X' 'O' 'X' 'X' 'X' 'O']
 ['O' 'O' 'O' 'O' 'X' 'O' 'X' 'O']
 ['O' 'O' 'O' 'O' 'O' 'X' 'O' 'O']
 ['O' 'O' 'O' 'O' 'O' 'O' 'O' 'O']]
Result of random AI vs random AI
Player 1 won 36 times
Player 2 won 63 times
There were ties 1 times
Number of games: 100

Player 1 percentage: 36.0
Player 2 percentage: 63.0
Ties percentage: 1.0
```

```python
def randomVRandom():

  numGames = 10
  st = State(p1,p2)
  st.play(numGames)
  print("Result of random AI vs random AI")
  percentage(numGames, player1wins, player2wins, ties)
  st.reset()
```

Python code

```
Available moves are []
Winner is X!
[['X' 'X' 'X' 'X' 'X' 'X' 'X' 'O']
 ['X' 'X' 'X' 'X' 'X' 'X' 'X' 'O']
 ['X' 'X' 'X' 'O' 'X' 'O' 'X' 'O']
 ['X' 'X' 'O' 'X' 'X' 'X' 'O' 'O']
 ['X' 'O' 'X' 'O' 'X' 'X' 'X' 'O']
 ['X' 'O' 'O' 'X' 'O' 'X' 'O' 'X']
 ['X' 'O' 'O' 'O' 'X' 'O' 'O' 'O']
 ['O' 'O' 'O' 'O' 'O' 'O' 'O' 'O']]
Result of random AI vs trained AI
Player 1 won 36 times
Player 2 won 64 times
There were ties 0 times
Number of games: 100

Player 1 percentage: 36.0
Player 2 percentage: 64.0
Ties percentage: 0.0
```

```python
def randomVTrained():

  numGames = 10
  st = State(p1,p2)
  st.play(numGames)


  p1.savePolicy()
  st.play(numGames)
  print("Result of random AI vs trained AI")
  percentage(numGames, player1wins, player2wins, ties)

  st.reset()
```

Python code

```
Available moves are []
Winner is O!
[['O' 'O' 'O' 'O' 'O' 'O' 'O' 'O']
 ['O' 'O' 'O' 'O' 'O' 'O' 'O' 'O']
 ['O' 'O' 'O' 'O' 'O' 'O' 'O' 'O']
 ['O' 'O' 'O' 'O' 'X' 'O' 'O' 'O']
 ['O' 'O' 'O' 'X' 'O' 'O' 'O' 'O']
 ['O' 'O' 'O' 'O' 'X' 'O' 'X' 'X']
 ['O' 'O' 'O' 'O' 'O' 'O' 'O' 'X']
 ['X' '.' 'X' 'X' 'X' 'X' 'X' 'X']]
Result of trained AI vs trained AI
Player 1 won 33 times
Player 2 won 67 times
There were ties 0 times
Number of games: 100

Player 1 percentage: 33.0
Player 2 percentage: 67.0
Ties percentage: 0.0
```

```python
def trainedVTrained():

  numGames = 10
  st = State(p1,p2)
  st.play(numGames)

  p1.savePolicy()
  p2.savePolicy()
  p1.loadPolicy("policy_p1")
  p2.loadPolicy("policy_p2")


  st.play(numGames)
  print("Result of trained AI vs trained AI")
  percentage(numGames, player1wins, player2wins, ties)
  st.reset()
```

Python code

What are some further directions you could take the program next? How could you further improve your trained player?

We might attempt to decrease the amount of time required to train the algorithm. We completed step 1 by creating a functional prototype, regardless of its performance or the problem's complexity. Next, we might take measures to reduce memory use and evaluate alternative network configurations and technical options to improve precision. In addition, we might attempt to apply several methods and compare them based on their time complexity and precision.

Next, we may create more effective graphical user interfaces to facilitate the audience's data reading. Lastly, we might potentially use several ML techniques that were utilized by other research teams, such as the minimax algorithm.