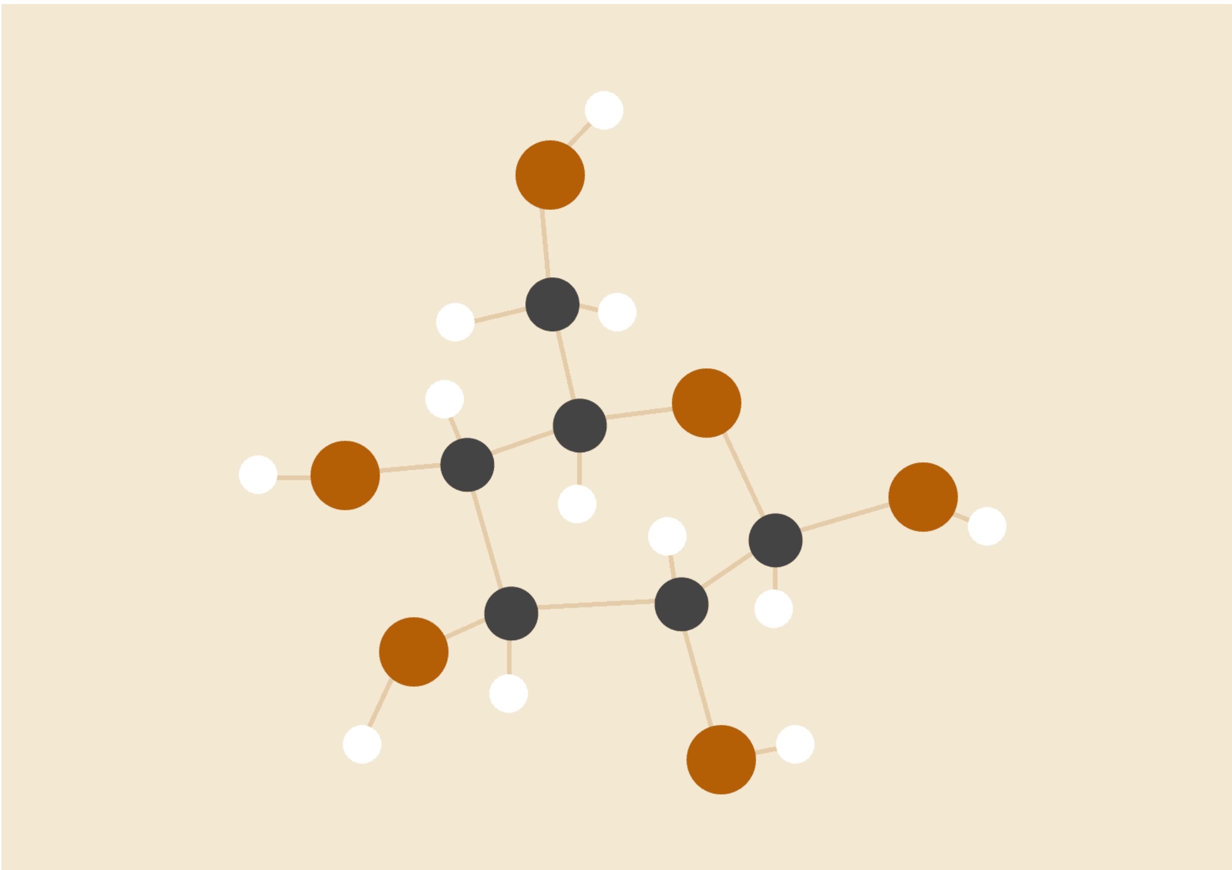
AI Assignment II



Name: Bingqi Xia

Student Id: 22300549

Implementation

Tic-Tac-Toe

My program represents the tic-tac-toe board as a list of 9 strings, which is filled with " " at the start of the game, and the two players fill the board with 'X' and 'O' respectively. Where X is the player who starts first. (The game interaction logic is based on open-source code [1])

The game supports alternating between player_x and player_o until a win or the board is filled (a draw).

Algorithms Implementation

I have implemented defaultPlayer, MiniMaxPlayer, and QlearningPlayer to play against each other.

Default Opponent

DefaultPlayer is an almost completely random strategy agent that only selects winnable positions for states that are about to win the game, and blocks when the opponent is about to win. In all other cases, it randomly selects an empty position on the board.

MiniMax

MiniMaxPlayer starts each turn by calculating the minimax value for all available positions, choosing the position that maximizes the value.

Q-Learning

In tic-tac-toe, each square of the board has three states - empty, player x has placed a piece, and player O has placed a piece - so there are a total of 3^9=19683 states of the board. In Q-learning, each state corresponds to one of the states of the board, so the state space is of size 19683.

Actions: Players can place their pieces in spaces, so actions include all space positions on the board. Reward functions: In the implementation, my rewards are: +1 for winning, -1 for losing, and 0 for a draw: both players get a reward. where 1 and -1 can be any positive and negative numbers, thus allowing the agent to learn to win and avoid losing.

For the hyperparameters, regarding Epsilon, which represents the probability used for exploration, I set it to 0.5 in the early training to allow the intelligence to explore the state space more and learn more information. As the experiment progressed I adjusted the Epsilon to 0.2 to allow the intelligence to make more decisions based on the learned Q tables. When evaluating performance, I set the epsilon to 0. This is to allow the intelligence to rely solely on the trained model to make decisions. For Alpha, the higher the alpha, the faster the learning speed but the more likely to overfit. For Gamma, I set it to 0.9 as I prefer the intelligence to be more long-term reward oriented.

Connect4

For training time reasons I have implemented a connect4 game with a 5*6-sized board.

Algorithms Implementation

I have implemented defaultPlayer, MiniMaxPlayer, and QlearningPlayer to play against each other.

MiniMax

In a Connect4 board of size 6*7, the size of the state space is 42 times 3. The search time with alpha-beta pruning was still unacceptable for connect4 due to the large state space, so I added a search depth limit, which was experimentally adjusted to 5. This means that the minimax will stop after at most 5 iterations.

Q-Learning

The time cost of full computation was too high for my computer, so I implemented a 5*6 size board. For the hyperparameters, I adjusted Epsilon to 0.5 to allow the intelligence to explore new states more due to the larger state space of connect4.

Performance Analysis

For both games, I played DefaultPlayer, MiniMaxPlayer, and QLearningPlayer against each other 100 times and differentiated for first and second-play situations.

Since training Q Learning is time-consuming, I went ahead and had Q Learning play against Default 100, 000 times, and the updated Q table was dumped into a file using Pickle. The time cost of training the tic-tac-toe QLearningPlayer is **34.5s**, and the time cost of training connect QLearningPlayer is **401.35s**.

The results are shown in the table below: (A VS B means A is the first player)

Tic-Tac-Toe

	Minimax Win	Q Learning Win	Default Win	Draw	time_cost
MiniMax VS Default	89	-	0	11	11.51s
Default VS MiniMax	21	-	0	79	11.6s
Q-Learning VS Default	-	16	24	60	0.03s
Default VS Q-Learning	-	13	59	28	0.03s
MiniMax VS Q-Learning	84	0	-	16	2.45s
Q-Learning VS MiniMax	51	43	-	6	1.19s

(For Q-Learning, there is an extra training time cost: 34.5s.)

Connect4

	Minimax Win	Q Learning Win	Default Win	Draw	time_cost
MiniMax VS Default	71	-	5	24	186.04s
Default VS MiniMax	57	-	41	2	158.82s
Q-Learning VS Default	-	11	81	8	0.38s
Default VS Q-Learning	-	13	82	5	0.35s
MiniMax VS Q-Learning	90	10	-	0	127.25s
Q-Learning VS MiniMax	66	27	-	7	113.07s

Results Analysis

Play with Default

Tic-Tac-Toe

a. **Win Rate**

In the MiniMax and Default games, the win, loss and draw rates when MiniMax was the first player were 89%, 0%, and 11%. When MiniMax is the second player, the win rate is 21%, and the other 79% draws.

When Q Learning was the first player, the defeat and draw rates were 16%, 24%, and 60% respectively, and when Q Learning was the second player, the defeat and draw rates were 13%, 59%, and 28%. Q learning has a lower winning percentage than MiniMax.

The reason for this may be:

Default, being an almost completely random opponent, takes a strategy that is not optimal in most cases, so MiniMax can guarantee not to lose by searching the game tree to calculate the win or loss rate of each possible choice in order to obtain the maximum win or minimum loss rate from it.

b. Time Cost

During the game, MiniMax spends much more time than Q learning due to the fact that each turn MiniMax has to search the game tree to calculate the win or loss rate of each possible choice. Q learning, on the other hand, is loaded and trained in advance and takes 34.5s for 100,000 training sessions For the simple state game, MiniMax takes less time.

Connect4

a. Win Rate

As we can see from the table above, Q learning has a much higher failure rate than MiniMax in Connect4 games. This is due to the huge state space in connect4 and the fact that Q learning is not fully trained.

b. Time Cost

MiniMax takes longer to search than Q learning because Connect4 has a greater search depth and MiniMax takes longer to search.

Overall

In the game against Default, the Minimax algorithm had a much higher overall win rate than Q-learning, but the Q-learning algorithm took less time to learn than MiniMax.

Play with Each Other

Tic-Tac-Toe

In Tic-Tac-Toe games, MiniMax won 84% of the time when it was the first player and only drew with Q-Learning in 16% of the cases.

When MiniMax was the second player, it still won more games than Q-Learning, but the gap narrowed.

Connect4

In connect4 games, MiniMax has a 90% win rate as the first player, and even with MiniMax as the second player, his win rate is much higher than Q learning.

Overall

When MiniMax plays with Q Learning, whether as the first player or the second player, MiniMax has a higher win rate than Q Learning.

Conclusion

The Minimax algorithm guarantees a relatively high win rate, but will take longer to search, especially if the search depth is large. Q learning takes a lot of time to train to get better results, and it has a much lower win rate than Minimax when training is incomplete.

Reference

[1] https://github.com/zkan/tictactoe

Appendix

I Tic-Tac-Toe

```
import random

class TicTacToe:

  def __init__(self, player_x, player_o):

      self.board = [' '] * 9

      self.player_x, self.player_o = player_x, player_o

      self.player_x_turn = True


  def draw_board(self):

      # reuse board

      print('     |     |     ')

      print('  %s  |  %s  |  %s  ' % (self.board[0],\

                                self.board[1],\

                                self.board[2]))

      print('_____|_____|_____')

      print('     |     |     ')

      print('  %s  |  %s  |  %s  ' % (self.board[3],\

                                self.board[4],\

                                self.board[5]))

      print('_____|_____|_____')

      print('     |     |     ')

      print('  %s  |  %s  |  %s  ' % (self.board[6],\

                                self.board[7],\

                                self.board[8]))

      print('     |     |     ')
```

```
    def is_board_full(self):
        return not any([space == ' ' for space in self.board])

    def play_game(self, train=True):
        if not train:
            print('\nNew game!')
            print('Play 1: X, Player 2: O')

        self.player_x.start_game()
        self.player_o.start_game()
        while True:
            if self.player_x_turn:
                player, char, other_player = self.player_x, 'X', self.player_o
            else:
                player, char, other_player = self.player_o, 'O', self.player_x

            if other_player.name == "human":
                self.draw_board()

            move = player.move(self.board)
            self.board[move - 1] = char

            if self.player_wins(char):
```

```
            player.reward(1, self.board)
            other_player.reward(-1, self.board)
            if not train:
                self.draw_board()
                print(char + ' wins!')
            if char == 'X':
                return 1
            else:
                return -1
        if self.is_board_full():
            player.reward(0.5, self.board)
            other_player.reward(0.5, self.board)
            if not train:
                self.draw_board()
                print('Draw!')
            return 0
        other_player.reward(0, self.board)
        self.player_x_turn = not self.player_x_turn
    def player_wins(self, char):
        winner_states = [(0, 1, 2), (3, 4, 5), (6, 7, 8),
                         (0, 3, 6), (1, 4, 7), (2, 5, 8),
                         (0, 4, 8), (2, 4, 6)]
        for i, j, k in winner_states:
            if char == self.board[i] == self.board[j] == self.board[k]:
```

```
                return True
        return False


class TicTacToePlayer(object):
    def __init__(self, is_first=True):
        self.name = 'human'
        self.player = 'X'
        self.opponent = 'O'
        self.is_first = is_first
        self.set_player()

    def start_game(self):
        pass

    def move(self, board):
        return int(input('Your move? '))

    def set_player(self):
        if not self.is_first:
            self.opponent = 'X'
            self.player = 'O'

    def available_moves(self, board):
```

```
        return [i + 1 for i in range(0, 9) if board[i] == ' ']

    def game_won(self, player, board):

        winning_states = [
            [0, 1, 2], [3, 4, 5], [6, 7, 8],
            [0, 3, 6], [1, 4, 7], [2, 5, 8],
            [0, 4, 8], [2, 4, 6]
        ]
        for positions in winning_states:
            if all(board[pos] == player for pos in positions):
                return True
        return False
    def reward(self, value, board):
        pass
```

I.I TicTacToe DefaultPlayer

```
from player.TicTacToePlayer import TicTacToePlayer
import random
class DefaultPlayer(TicTacToePlayer):
    def __init__(self, is_first):
        super().__init__(is_first)
        # self.name = 'default'
```

```
        self.name = 'default'

    def move(self, board):
        # Check if there's a winning move for the player and make it
        for move in self.available_moves(board):
            new_board = board[:move - 1] + list(self.player) + board[move:]
            if self.game_won(self.player, new_board):
                return move

        # if opponent will win in next turn, block it
        for move in self.available_moves(board):
            new_board = board[:move - 1] + list(self.opponent) + board[move:]
            if self.game_won(self.opponent, new_board):
                return move
        # else choose a random move
        return random.choice(self.available_moves(board))

class TicTacToePlayer(object):
    def __init__(self, is_first=True):
        self.name = 'human'
        self.player = 'X'
        self.opponent = 'O'
        self.is_first = is_first
        self.set_player()
```

```
def start_game(self):
    pass

def move(self, board):
    return int(input('Your move? '))

def set_player(self):
    if not self.is_first:
        self.opponent = 'X'
        self.player = 'O'

def available_moves(self, board):
    return [i + 1 for i in range(0, 9) if board[i] == ' ']

def game_won(self, player, board):

    winning_states = [
        [0, 1, 2], [3, 4, 5], [6, 7, 8],
        [0, 3, 6], [1, 4, 7], [2, 5, 8],
        [0, 4, 8], [2, 4, 6]
    ]

    for positions in winning_states:
```

```
            if all(board[pos] == player for pos in positions):
                return True
        return False


    def reward(self, value, board):
        pass
```

I.II TicTacToe MiniMaxPlayer

```
from player.TicTacToePlayer import TicTacToePlayer
import random
class MiniMaxPlayer(TicTacToePlayer):
    def __init__(self, is_first):
        super().__init__(is_first)
        self.name = 'minimax'
    def move(self, board):
        alpha = float('-inf')
        beta = float('inf')
        depth = 0


        best_value = float('-inf')
        best_move = None
        for move in self.available_moves(board):
            board[move - 1] = self.player
```

```
            value = self.min_value(board, alpha, beta, depth + 1)
            board[move - 1] = ' '
            if value > best_value:
                best_value = value
                best_move = move
        return best_move

    def is_game_over(self, board):
        for a, b, c in [(0, 1, 2), (3, 4, 5), (6, 7, 8),
                        (0, 3, 6), (1, 4, 7), (2, 5, 8),
                        (0, 4, 8), (2, 4, 6)]:

            if self.player == board[a] == board[b] == board[c]:
                return (True, 1)
            elif self.opponent == board[a] == board[b] == board[c]:
                return (True, -1)

        if not any([space == ' ' for space in board]):
            return (True, 0)

        return (False, 0)

    def max_value(self, board, alpha, beta, depth):
        in_terminal_state, utility_value = self.is_game_over(board)
```

```
        if in_terminal_state or depth >= 9:
            return utility_value

        value = float('-inf')
        for move in self.available_moves(board):
            board[move - 1] = self.player
            value = max(value, self.min_value(board, alpha, beta, depth + 1))
            board[move - 1] = ' '

            alpha = max(alpha, value)
            if beta <= alpha:
                break
        return value

    def min_value(self, board, alpha, beta, depth):
        in_terminal_state, utility_value = self.is_game_over(board)
        if in_terminal_state or depth >= 9:
            return utility_value
        value = float('inf')
        for move in self.available_moves(board):
            board[move - 1] = self.opponent
            value = min(value, self.max_value(board, alpha, beta, depth + 1))
            board[move - 1] = ' '
```

```
            beta = min(beta, value)
            if beta <= alpha:
                break
        return value
```

I.III TicTacToe QLearningPlayer

```
import os.path
import random
from player.TicTacToePlayer import TicTacToePlayer
import pickle
import os


class QLearningPlayer(TicTacToePlayer):
    def __init__(self, is_first, save_path):
        super().__init__(is_first)
        self.name = 'qlearning'
        self.q_table = {}
        self.epsilon = 0.2
        self.alpha = 0.4
        self.gamma = 0.9
        self.last_state = (' ',) * 9
        self.last_move = None
        self.save_path = save_path
```

```
        self.load_qtable()

    def move(self, board):
        actions = self.available_moves(board)
        if random.random() < self.epsilon:
            self.last_move = random.choice(actions)
            self.last_state = tuple(board)
            return self.last_move
        qs = [self.getQ(self.last_state, each) for each in actions]
        maxQ = max(qs)
        if qs.count(maxQ) > 1:
            best_options = [i for i in range(len(actions)) if qs[i] == maxQ]
            i = random.choice(best_options)
        else:
            i = qs.index(maxQ)
        self.last_move = actions[i]
        self.last_state = tuple(board)
        return self.last_move

    def start_game(self):
        self.last_state = (' ',) * 9
        self.last_move = None

    def getQ(self, state, action):
```

```
        if self.q_table.get((state, action)) is None:
            self.q_table[(state, action)] = 1.0

        return self.q_table.get((state, action))

    def save_qtable(self):
        filehandler = open(self.save_path, 'wb')
        pickle.dump(self.q_table, filehandler)

    def load_qtable(self):
        if os.path.exists(self.save_path):
            filehandler = open(self.save_path, 'rb')
            self.q_table = pickle.load(filehandler)

    def reward(self, value, board):
        if self.last_move:
            self.learn(self.last_state, self.last_move, value, tuple(board))

    def learn(self, state, action, reward, result_state):
        prev = self.getQ(state, action)
        maxqnew = max(
            [self.getQ(result_state, a) for a in self.available_moves(state)]
        )
        self.q_table[(state, action)] = prev + \
```

```
            self.alpha * ((reward + self.gamma * maxqnew) - prev)
```

II Connect4

```
import numpy as np


class Connect4(object):
   def __init__(self, player_x, player_o):
       self.row_size = 5
       self.column_size = 6
       self.window_length = 4

       self.player_x, self.player_o = player_x, player_o

       self.PLAYER_PIECE = 1
       self.OPPONENT_PIECE = -1
       self.EMPTY = 0

       self.player_x_turn = True

       self.board = self.create_board()

   def create_board(self):
       board = np.zeros((self.row_size, self.column_size))
```

```
        return board

    def set_move(self, row, col, piece):
        self.board[row][col] = piece

    def is_valid_location(self, col):
        return self.board[self.row_size - 1][col] == 0

    def get_available_row(self, col):
        # return first available row
        for r in range(self.row_size):
            if self.board[r][col] == 0:
                return r

    def draw_board(self):
        print(np.flip(self.board, 0))

    def player_wins(self, piece):
        # Check vertical
        for c in range(self.column_size):
            for r in range(self.row_size - 3):
                if self.board[r][c] == piece and self.board[r + 1][c] == piece and self.board[r +
2][c] == piece and \
                        self.board[r + 3][c] == piece:
```

```
                return True

        # Check positively sloped
        for c in range(self.column_size - 3):
            for r in range(self.row_size - 3):
                if self.board[r][c] == piece and self.board[r + 1][c + 1] == piece and \
                        self.board[r + 2][c + 2] == piece and self.board[r + 3][c + 3] == piece:
                    return True

        # Check negatively sloped
        for c in range(self.column_size - 3):
            for r in range(3, self.row_size):
                if self.board[r][c] == piece and self.board[r - 1][c + 1] == piece and \
                        self.board[r - 2][c + 2] == piece and self.board[r - 3][c + 3] == piece:
                    return True

        # Check horizontal locations for win
        for c in range(self.column_size - 3):
            for r in range(self.row_size):
                if self.board[r][c] == piece and self.board[r][c + 1] == piece and self.board[r][c +
2] == piece and \
                        self.board[r][c + 3] == piece:
                    return True
```

```
  def is_terminal_node(self):
    return self.player_wins(self.PLAYER_PIECE) or
self.player_wins(self.OPPONENT_PIECE) or len(
      self.get_valid_locations()) == 0


  def get_valid_locations(self):
    valid_locations = []
    for col in range(self.column_size):
      if self.is_valid_location(col):
        valid_locations.append(col)
    return valid_locations


  def play_game(self, train=True):
    if not train:
      print('\nNew game!')
      print('Play 1: X, Player 2: O')


    while True:
      if self.player_x_turn:
        player, piece, other_player = self.player_x, self.PLAYER_PIECE, self.player_o
      else:
        player, piece, other_player = self.player_o, self.OPPONENT_PIECE, self.player_x


      move = player.move(self.board)
```

```
        self.set_move(move[0], move[1], piece)

        if self.player_wins(piece):
            player.reward(1, self.board)
            other_player.reward(-1, self.board)
            if piece == self.PLAYER_PIECE:
                if not train:
                    print("X wins!")
                return 1
            elif piece == self.OPPONENT_PIECE:
                if not train:
                    print("O wins!")
                return -1

        if len(self.get_valid_locations()) == 0:
            player.reward(0.5, self.board)
            other_player.reward(0.5, self.board)
            if not train:
                print('Draw!')
            return 0
        self.player_x_turn = not self.player_x_turn
class Connect4Player(object):
    def __init__(self, is_first):
        self.name = 'human'
```

```
        self.player = 1
        self.opponent = -1
        self.is_first = is_first
        self.EMPTY = 0

        self.row_size = 5
        self.column_size = 6
        self.window_length = 4

        self.set_player()

    def start_game(self):
        pass

    def reward(self, value, board):
        pass

    def set_move(self, board, row, col, piece):
        board[row][col] = piece

    def move(self, board):
        return int(input('Your move? '))

    def is_valid_location(self, board, col):
```

```
        return board[self.row_size - 1][col] == self.EMPTY


    def get_available_row(self, board, col):
        for r in range(self.row_size):
            if board[r][col] == self.EMPTY:
                return r


    def get_valid_locations(self, board):
        valid_locations = []
        for col in range(self.column_size):
            if self.is_valid_location(board, col):
                valid_locations.append(col)
        return valid_locations


    def set_player(self):
        if not self.is_first:
            self.opponent = 1
            self.player = -1


    def available_moves(self, board):
        return [i + 1 for i in range(0, 9) if board[i] == ' ']


    def game_won(self, player, board):
```

```
    winning_states = [
        [0, 1, 2], [3, 4, 5], [6, 7, 8],
        [0, 3, 6], [1, 4, 7], [2, 5, 8],
        [0, 4, 8], [2, 4, 6]
    ]

    for positions in winning_states:
        if all(board[pos] == player for pos in positions):
            return True
    return False
```

II.I Connect4 DefaultPlayer

```
from player.Connect4Player import Connect4Player
import random
class DefaultPlayer(Connect4Player):
  def __init__(self, is_first):
    super().__init__(is_first)
    self.name = 'default'
    self.column_size = 6
    self.row_size = 5

  def available_moves(self, board):
    valid_locations = []
    for col in range(self.column_size):
```

```
            if self.is_valid_location(board, col):
                valid_locations.append(col)
        return valid_locations

    def is_valid_location(self, board, col):
        return board[self.row_size - 1][col] == 0

    def move(self, board):
        # Check if there's a winning move for the opponent and block it
        for col in self.available_moves(board):
            row = self.get_available_row(board, col)
            b_copy = board.copy()
            self.set_move(b_copy, row, col, self.opponent)
            if self.game_won(b_copy, self.opponent):
                return row, col

        # Check if there's a winning move for the player and make it
        for col in self.available_moves(board):
            row = self.get_available_row(board, col)
            b_copy = board.copy()
            self.set_move(b_copy, row, col, self.player)
            if self.game_won(b_copy, self.player):
                return row, col
```

```
        # If no winning move for the player or the opponent, choose a random move
        column = random.choice(self.available_moves(board))
        row = self.get_available_row(board, column)
        return row, column

    def reward(self, value, board):
        pass

    def game_won(self, board, piece):
        # Check horizontal locations for win
        for c in range(self.column_size - 3):
            for r in range(self.row_size):
                if board[r][c] == piece and board[r][c + 1] == piece and board[r][c + 2] == piece
and board[r][
                    c + 3] == piece:
                    return True

        # Check vertical locations for win
        for c in range(self.column_size):
            for r in range(self.row_size - 3):
                if board[r][c] == piece and board[r + 1][c] == piece and board[r + 2][c] == piece
and board[r + 3][
                    c] == piece:
                    return True
```

```
        # Check positively sloped diaganols
        for c in range(self.column_size - 3):
            for r in range(self.row_size - 3):
                if board[r][c] == piece and board[r + 1][c + 1] == piece and board[r + 2][c + 2] == 
piece and \
                        board[r + 3][c + 3] == piece:
                    return True

        # Check negatively sloped diaganols
        for c in range(self.column_size - 3):
            for r in range(3, self.row_size):
                if board[r][c] == piece and board[r - 1][c + 1] == piece and board[r - 2][c + 2] == 
piece and \
                        board[r - 3][c + 3] == piece:
                    return True
```

II.II Connect4 MiniMaxPlayer

```
import numpy as np
import random
from player.Connect4Player import Connect4Player



class MiniMaxPlayer(Connect4Player):
```

```
    def __init__(self, is_first):
        super().__init__(is_first)
        self.name = 'minimax'

    def player_wins(self, board, piece):
        # Check horizontal
        for c in range(self.column_size - 3):
            for r in range(self.row_size):
                if board[r][c] == piece and board[r][c + 1] == piece and board[r][c + 2] == piece \
                        and board[r][c + 3] == piece:
                    return True

        # Check vertical
        for c in range(self.column_size):
            for r in range(self.row_size - 3):
                if board[r][c] == piece and board[r + 1][c] == piece and board[r + 2][c] == piece
and \
                        board[r + 3][c] == piece:
                    return True

        # Check positively sloped
        for c in range(self.column_size - 3):
            for r in range(self.row_size - 3):
```

```
            if board[r][c] == piece and board[r + 1][c + 1] == piece and board[r + 2][c + 2] ==
piece and \
                board[r + 3][c + 3] == piece:
              return True

    # Check negatively sloped
    for c in range(self.column_size - 3):
      for r in range(3, self.row_size):
        if board[r][c] == piece and board[r - 1][c + 1] == piece and board[r - 2][c + 2] ==
piece and \
                board[r - 3][c + 3] == piece:
              return True

  def evaluate_window(self, window, piece):
    score = 0
    opp_piece = self.player
    if piece == self.player:
      opp_piece = self.opponent

    if window.count(piece) == 4:
      score += 100
    elif window.count(piece) == 3 and window.count(self.EMPTY) == 1:
      score += 5
    elif window.count(piece) == 2 and window.count(self.EMPTY) == 2:
```

```
            score += 2

        if window.count(opp_piece) == 3 and window.count(self.EMPTY) == 1:
            score -= 4

        return score

    def score_position(self, board, piece):
        score = 0

        ## Score center column
        center_array = [int(i) for i in list(board[:, self.column_size // 2])]
        center_count = center_array.count(piece)
        score += center_count * 3

        ## Score Horizontal
        for r in range(self.row_size):
            row_array = [int(i) for i in list(board[r, :])]
            for c in range(self.column_size - 3):
                window = row_array[c:c + self.window_length]
                score += self.evaluate_window(window, piece)

        ## Score Vertical
        for c in range(self.column_size):
```

```
        col_array = [int(i) for i in list(board[:, c])]

        for r in range(self.row_size - 3):

            window = col_array[r:r + self.window_length]

            score += self.evaluate_window(window, piece)


    ## Score posiive sloped diagonal

    for r in range(self.row_size - 3):

        for c in range(self.column_size - 3):

            window = [board[r + i][c + i] for i in range(self.window_length)]

            score += self.evaluate_window(window, piece)


    for r in range(self.row_size - 3):

        for c in range(self.column_size - 3):

            window = [board[r + 3 - i][c + i] for i in range(self.window_length)]

            score += self.evaluate_window(window, piece)


    return score




def is_terminal_node(self, board):

    return self.player_wins(board, self.player) or self.player_wins(board, self.opponent)
or len(

        self.get_valid_locations(board)) == 0
```

```
def minimax(self, board, depth, alpha, beta, isMax):
    valid_locations = self.get_valid_locations(board)
    is_terminal = self.is_terminal_node(board)
    if depth == 0 or is_terminal:
        if is_terminal:
            if self.player_wins(board, self.player):
                return (None, self.player)
            elif self.player_wins(board, self.opponent):
                return (None, self.opponent)
            else:  # draw
                return (None, 0)
        else:  # Depth is zero
            return (None, self.score_position(board, self.player))
    if isMax:
        value = float('-inf')
        column = random.choice(valid_locations)
        for col in valid_locations:
            row = self.get_available_row(board, col)
            b_copy = board.copy()
            self.set_move(b_copy, row, col, self.player)
            new_score = self.minimax(b_copy, depth - 1, alpha, beta, False)[1]
            if new_score > value:
                value = new_score
                column = col
```

```
            alpha = max(alpha, value)
            if alpha >= beta:
                break
        return column, value

    else:  # Minimizing player
        value = float('inf')
        column = random.choice(valid_locations)
        for col in valid_locations:
            row = self.get_available_row(board, col)
            b_copy = board.copy()
            self.set_move(b_copy, row, col, self.opponent)
            new_score = self.minimax(b_copy, depth - 1, alpha, beta, True)[1]
            if new_score < value:
                value = new_score
                column = col
            beta = min(beta, value)
            if alpha >= beta:
                break
        return column, value

def move(self, board):
    col, minimax_score = self.minimax(board, 5, float('-inf'), float('inf'), True)
```

```
        if self.is_valid_location(board, col):
            row = self.get_available_row(board, col)
            self.set_move(board, row, col, self.opponent)
            return row, col
```

II.III Connect4 QlearningPlayer

```
import numpy as np
import random
from player.Connect4Player import Connect4Player
import pickle
import os



class QLearningPlayer(Connect4Player):
    def __init__(self, is_first, save_path):
        super().__init__(is_first)
        self.name = 'qlearning'
        self.q_table = {}
        # learning rate
        self.alpha = 0.5
        self.epsilon = 0.1
        self.gamma = 0.9  # discount factor
        self.last_state = np.zeros((self.row_size, self.column_size))
        self.last_move = None
```

```
        self.save_path = save_path

    def save_qtable(self):
        filehandler = open(self.save_path, 'wb')
        pickle.dump(self.q_table, filehandler)

    def load_qtable(self):
        if os.path.exists(self.save_path):
            filehandler = open(self.save_path, 'rb')
            self.q_table = pickle.load(filehandler)

    def start_game(self):
        self.last_state = np.zeros((self.row_size, self.column_size))
        self.last_move = None

    def tuple_state(self, state):
        return tuple(map(tuple, state))

    def getQ(self, state, action):
        state = self.tuple_state(state)
        if self.q_table.get((state, action)) is None:
            self.q_table[(state, action)] = 1.0  # Initial all q as 1

        return self.q_table.get((state, action))
```

```
def move(self, board):
    actions = self.get_valid_locations(board)

    if random.random() < self.epsilon:  # To balance exploration and exploitation
        col = random.choice(actions)
        row = self.get_available_row(board, col)
        self.last_move = row, col
        self.last_state = self.tuple_state(board)
        return self.last_move

    qs = [self.getQ(self.last_state, each) for each in actions]
    maxQ = max(qs)

    if qs.count(maxQ) > 1:
        best_options = [i for i in range(len(actions)) if qs[i] == maxQ]
        i = random.choice(best_options)
    else:
        i = qs.index(maxQ)

    col = actions[i]
    row = self.get_available_row(board, col)
    self.last_move = row, col
    self.last_state = self.tuple_state(board)
```

```
        return self.last_move

    def reward(self, value, board):
        if self.last_move:
            self.learn(self.last_state, self.last_move, value, self.tuple_state(board))

    def learn(self, state, action, reward, result_state):
        prev = self.getQ(state, action)
        maxqnew = max(
            [self.getQ(result_state, a) for a in self.get_valid_locations(state)]
        )
        self.q_table[(state, action)] = prev + \
                                  self.alpha * ((reward + self.gamma * maxqnew) - prev)
```