

# Artificial Intelligence Assignment 2

Jiaming Deng 22302794

## 1. Tic Tac Toe

In this part, I mainly introduce the implementation of game Tic Tac Toe and the implementation methods of two algorithms, Minimax algorithm with alpha-beta pruning and tabular Q-learning Reinforcement Learning algorithm in the game.

### 1. Implementation of the Tic Tac Toe game

Firstly, we initialize the game as a 3x3 two-dimensional matrix, in which if the element with a value of 0 is a position that has not played chess, the two global variables HUMAN and COMP are used to identify two two players, because this assignment requires the use of two algorithms Play against each other, so these two global variables are also two algorithms. The three global variables WIN, LOSE, and DRAW are used to record the number of wins, losses, and draws of one of the players, and are used to compare the performance of the two algorithms.

```
HUMAN = 1
COMP = 2
board = [
    [0, 0, 0],
    [0, 0, 0],
    [0, 0, 0],
]
WIN = 0
LOSE = 0
DRAW = 0
```

Next, I implemented some game operation functions, in which the wins function inputs the state and the player, and judges whether all 3 points in the state are consistent to determine whether the player wins. The game\_over function determines whether the two players have won to determine whether the game is over. The empty\_cells function returns all points that no player has marked. The valid\_move function reads in two coordinates to determine whether the position has been moved. The set\_move function is used to mark where the player moves.

```

def wins(state, player):
    win_state = [
        [state[0][0], state[0][1], state[0][2]],
        [state[1][0], state[1][1], state[1][2]],
        [state[2][0], state[2][1], state[2][2]],
        [state[0][0], state[1][0], state[2][0]],
        [state[0][1], state[1][1], state[2][1]],
        [state[0][2], state[1][2], state[2][2]],
        [state[0][0], state[1][1], state[2][2]],
        [state[2][0], state[1][1], state[0][2]],
    ]
    if [player, player, player] in win_state:
        return True
    else:
        return False

def game_over(state):
    return wins(state, HUMAN) or wins(state, COMP)

def empty_cells(state):
    cells = []

    for x, row in enumerate(state):
        for y, cell in enumerate(row):
            if cell == 0:
                cells.append([x, y])

    return cells

def valid_move(x, y):
    if [x, y] in empty_cells(board):
        return True
    else:
        return False

```

## 2. Minimax algorithm

First, at the beginning of the algorithm, I judge whether the current game state is a win or a draw, or the depth of the algorithm is 0. I use the `is_terminal` function to determine if the game is over, and use the depth parameter to determine if the depth of the algorithm is 0. If the game is over, determine which player wins. If an algorithm wins, the update score is the victory score plus depth times 3. If the other algorithm wins, the updated score is the failure score minus the depth multiplied by 3. In case of a tie, the score is calculated according to the situation.

```

def minimax1(state, depth, alpha, beta, isCOMP):
    possibleMoves = empty_cells(state)
    if is_terminal(state) or depth == 0:
        if is_terminal(state):
            if wins(state, COMP):
                score = WIN_SCORE + depth * 3
                return (None, score)
            elif wins(state, HUMAN):
                score = LOSE_SCORE - depth * 3
                return (None, score)
            else:
                return (None, 0)
        else:
            return (None, score_position(board, COMP))

```

Next, get the positions of all possible moves in the current state through the function. If COMP is the current turn, the function will traverse all possible moves, perform a depth-first search under each move, and calculate the evaluation of the state corresponding to the move value. Here, the COMP player chooses the move with the highest evaluation value, because he wants to maximize his score, while assuming that the opponent will take the most unfavorable response for him. After traversing a node each time, the algorithm will update the value of alpha. If the value of alpha is greater than or equal to beta, it indicates that the optimal solution has been found. Similarly, if it is the turn of the HUMAN player, the function will also try to traverse all the movement modes, perform a depth-first search in each movement mode, and assume that the COMP player adopts the most unfavorable behavior for him, after traversing a node each time, the algorithm will update the value of beta. If the value of alpha is greater than or equal to beta, it indicates that the optimal solution has been found. Finally returns the selected positions and evaluated values.

```

if isCOMP:
    value = -math.inf
    row, col = random.choice(possibleMoves)

    for x, y in possibleMoves:
        new_board = np.copy(state)
        set_move(x, y, COMP)
        newScore = minimax1(new_board, depth - 1, alpha, beta, False)[1]
        if newScore > value:
            value = newScore
            row = x
            col = y
        alpha = max(alpha, value)
        if alpha >= beta:
            break

    return row, col, value

else:
    value = math.inf
    row, col = random.choice(possibleMoves)
    for x, y in possibleMoves:
        new_board = np.copy(state)
        set_move(new_board, col, HUMAN)
        newScore = minimax1(new_board, depth - 1, alpha, beta, True)[1]
        if newScore < value:
            value = newScore
            row = x
            col = y
        beta = min(beta, value)
        if alpha >= beta:
            break

    return row, col, value

```

### 3. Q-learning Reinforcement Learning algorithm

In the constructor of the Qlearn class, alpha is the learning rate, gamma is the discount factor, eps is the  $\epsilon$  value in the greedy algorithm, and eps\_decay is the decay factor of the  $\epsilon$  value. The position where the action may move. Initialize a 3x3 two-dimensional array to indicate the position where the initial state can move. Q is a dictionary that stores actions and states. If there is a next state for each update, use the Q value of the next state to update The Q value of the current state. rewards is to record the rewards obtained during training.

```

class QLearner:
    def __init__(self, alpha, gamma, eps, eps_decay=0.):
        self.alpha = alpha
        self.gamma = gamma
        self.eps = eps
        self.eps_decay = eps_decay
        self.actions = []
        for i in range(3):
            for j in range(3):
                self.actions.append((i, j))
        self.Q = {}
        for action in self.actions:
            self.Q[action] = collections.defaultdict(int)
        self.rewards = []

```

Next, in the `get_action` function, we will find out a possible movable position from the state each time, and call the random function to generate a random number from 0 to 1. If the random number is less than `eps`, we return a random action, otherwise I find the maximum value of all actions and states in the Q list, and return the action corresponding to this value, update `eps` to `1-eps_decay` and return this action. In the `get_action` function, we will find out a possible movable position from the state every time, and call the random function to generate a random number from 0 to 1. If the random number is less than `eps`, we return a random action, otherwise I find the maximum value of all actions and states in the Q list, and return the action corresponding to this value, update `eps` to `1-eps_decay` and return this action. The `save` function is used to save the trained parameters. The `update` function is used to update the Q list and the reward list according to the current and last state and action.

```

def get_action(self, s):
    possible_actions = [a for a in self.actions if s[a[0]*3 + a[1]] == '0']
    if random.random() < self.eps:
        action = possible_actions[random.randint(0, len(possible_actions)-1)]
    else:
        values = np.array([self.Q[a][s] for a in possible_actions])
        col_max = np.where(values == np.max(values))[0]
        if len(col_max) > 1:
            col = np.random.choice(col_max, 1)[0]
        else:
            col = col_max[0]
        action = possible_actions[col]

    self.eps *= (1.-self.eps_decay)
    return action

def save(self, path):
    if os.path.isfile(path):
        os.remove(path)
    with open(path, 'wb') as f:
        pickle.dump(self, f)

def update(self, s, s_, a, a_, r):
    if s_ is not None:
        possible_actions = []
        Qs = []
        for action in self.actions:
            if s_[action[0] * 3 + action[1]] == '0':
                possible_actions.append(action)
        for action in possible_actions:
            Qs.append(self.Q[action][s_])
        self.Q[a][s] += self.alpha*(r + self.gamma*np.max(Qs) - self.Q[a][s])
    else:
        self.Q[a][s] += self.alpha*(r - self.Q[a][s])
    self.rewards.append(r)

```

To train the Qlearn algorithm, I implemented a Teacher class. Because the rules of tic-tac-toe are relatively simple, there is a perfect way to play chess, according to certain rules, you can guarantee that you will not lose, and the worst is a draw. The movement algorithm of the Teacher class is implemented according to such rules. I implement the movement methods with different priorities respectively, and call these methods to move according to the priority. These functions are sorted according to priority: the position that will move is the position that directly wins the game, the position where teammates can win by taking one step, the position where there are two positions to move and win after moving, and the position where the opponent moves has two positions The location to move to win, the location in the center, the location in the four corners, the other locations.

```
class Teacher:
    def set_win(self, board, player=1):...

    def set_blockOpponentWin(self, board):...

    def set_twoThreatToWin(self, board):...

    def set_blockOpponentTwoThreatWin(self, board):...

    def set_center(self, board):
        if board[1][1] == 0:
            return 1, 1
        return None

    def set_corner(self, board):...

    def set_other(self, board):...

    def set_random(self, board):...

    def move(self, board):
        if random.random() > 0.8:
            return self.set_random(board)
        if self.set_win(board):
            return self.set_win(board)
        if self.set_blockOpponentWin(board):
            return self.set_blockOpponentWin(board)
        if self.set_blockOpponentTwoThreatWin(board):
            return self.set_blockOpponentTwoThreatWin(board)
        if self.set_center(board):
            return self.set_center(board)
        if self.set_corner(board):
            return self.set_corner(board)
        if self.set_other(board):
            return self.set_other(board)
        return self.set_random(board)
```

#### 4. Baseline algorithm

The baseline function is the default opponent that will play these games against my algorithm. Its design idea is to find all possible moving positions, traverse them and simulate moving it and judge whether it will win, move this position if it wins, and move to an empty position randomly if it does not win.

```

def baseline(player):
    depth = len(empty_cells(board))
    if depth == 0 or game_over(board):
        return

    cells = [...]
    x = -1
    y = -1
    for i in range(len(cells)):
        if 0 <= i <= 2:
            if cells[i][0] == cells[i][1] and cells[i][0] == player and cells[i][2] == 0:
                x = i
                y = 2
                break
            elif cells[i][0] == cells[i][2] and cells[i][0] == player and cells[i][1] == 0:
                x = i
                y = 1
                break
            elif cells[i][1] == cells[i][2] and cells[i][1] == player and cells[i][0] == 0:
                x = i
                y = 0
                break
        elif 3 <= i <= 5:
            pass
        elif i == 6:
            pass

    if x == -1 and y == -1:
        while True:
            x = choice([0, 1, 2])
            y = choice([0, 1, 2])
            if valid_move(x, y):
                break
    set_move(x, y, player)

```

## 2. Connect 4

In this part, I mainly introduce the implementation of game Connect 4 and the implementation methods of two algorithms, Minimax algorithm with alpha-beta pruning and tabular Q-learning Reinforcement Learning algorithm in the game.

### 1. Implementation of the Connect 4 game

Firstly, we initialize the game as a 6x7 two-dimensional matrix, in which if the element with a value of 0 is a position that has not played chess, the two global variables HUMAN and COMP are used to identify two players, because this assignment requires the use of two algorithms Play against each other, so these two global variables are also two algorithms. The three global variables WIN, LOSE, and DRAW are used to record the number of wins, losses, and draws of one of the players, and are used to compare the performance of the two algorithms.

```

board = [[0, 0, 0, 0, 0, 0, 0],
          [0, 0, 0, 0, 0, 0, 0],
          [0, 0, 0, 0, 0, 0, 0],
          [0, 0, 0, 0, 0, 0, 0],
          [0, 0, 0, 0, 0, 0, 0],
          [0, 0, 0, 0, 0, 0, 0]]

HUMAN = 1
COMP = 2
WIN = 0
LOSE = 0
DRAW = 0

```

Second, I implemented some game operation functions, among which the `set_move` function is used to add the chess played by the corresponding player in the specified column, the `checkWin` function is used to judge whether the specified player wins, and the `getValidColumns` function is used to obtain the current chessboard that can play chess column, the `is_terminal` function is used to judge whether the current chess game is over, and the `valid_move` function is used to judge whether the specified column can play chess.

By calling these functions, the process of a game is probably an algorithm to return the chess position, call the `set_move` function to play chess, and call the `checkWin` function to determine whether a player has won. If there is no call to `getValidColumns` to determine whether there is still a chess position, judge Is it a match. Other functions that are not called are needed during the execution of the algorithm.

```

def set_move(state, column, player):
    for i in range(6):...

def checkWin(state, player):...

def getValidColumns(state):
    validColumns = []
    for i in range(7):
        if state[0][i] == 0:
            validColumns.append(i)
    return validColumns

def is_terminal(state):
    return checkWin(state, HUMAN) or checkWin(state, COMP) or len(getValidColumns(state)) == 0

def valid_move(col):
    if board[0][col] == 0:
        return True
    else:
        return False

```

## 2. Minimax algorithm

First of all, at the beginning of the algorithm, I judge whether the current game state has a victory or a draw, or the depth of the algorithm is 0. I use the `is_terminal` function to judge whether the game is over, and use the depth parameter to judge whether the depth of the algorithm is 0. If the game is over, determine which player wins. If the an algorithm wins, the update score is the victory score plus the depth multiplied by 3. If the other algorithm wins, the updated score is the defeat score minus the depth multiplied by 3. If it is a tie, return according to the situation Calculated score.

```
def minimax(state, depth, alpha, beta, isCOMP):
    possibleMoves = getValidColumns(state)
    if is_terminal(state) or depth == 0:
        if is_terminal(state):
            if checkWin(state, COMP):
                score = WIN_SCORE + depth * 3
                return (None, score)
            elif checkWin(state, HUMAN):
                score = LOSE_SCORE - depth * 3
                return (None, score)
            else:
                return (None, 0)
        else:
            return (None, score_position(board, COMP))
```

Next, get the columns of all possible moves in the current state through the function. If COMP is the current turn, the function will traverse all possible moves, perform a depth-first search under each move, and calculate the evaluation of the state corresponding to the move value. Here, the COMP player chooses the move with the highest evaluation value, because he wants to maximize his score, while assuming that the opponent will take the most unfavorable response for him. After traversing a node each time, the algorithm will update the value of alpha. If the value of alpha is greater than or equal to beta, it indicates that the optimal solution has been found. Similarly, if it is the turn of the HUMAN player, the function will also try to traverse all the movement modes, perform a depth-first search in each movement mode, and assume that the COMP player adopts the most unfavorable behavior for him, after traversing a node each time , the algorithm will update the value of beta. If the value of alpha is greater than or equal to beta, it indicates that the optimal solution has been found. Finally returns the selected columns and evaluated values.



```

if isCOMP:
    value = -math.inf
    column = random.choice(possibleMoves)
    for col in possibleMoves:
        new_board = np.copy(state)
        set_move(new_board, col, COMP)
        newScore = minimax(new_board, depth - 1, alpha, beta, False)[1]
        if newScore > value:
            value = newScore
            column = col
        alpha = max(alpha, value)
        if alpha >= beta:
            break

    return column, value

else:
    value = math.inf
    column = random.choice(possibleMoves)
    for col in possibleMoves:
        new_board = np.copy(state)
        set_move(new_board, col, HUMAN)
        newScore = minimax(new_board, depth - 1, alpha, beta, True)[1]
        if newScore < value:
            value = newScore
            column = col
        beta = min(beta, value)
        if alpha >= beta:
            break

    return column, value

```

### 3. Q-learning Reinforcement Learning algorithm

Firstly, I define the Qlearn algorithm as a class. In the constructor, alpha is the learning rate, gamma is the discount factor, eps is the  $\epsilon$  value in the greedy algorithm, and eps\_decay is the decay factor of the  $\epsilon$  value. The position where actions may move, initialized to 7 numbers from 0 to 6, referring to columns 1 to 7, Q is a dictionary, storing actions and states, if the next state exists in each update, use the next state The Q value updates the Q value of the current state. rewards is to record the rewards obtained during training.

```

class QLearner:
    def __init__(self, alpha, gamma, eps, eps_decay=0.):
        self.alpha = alpha
        self.gamma = gamma
        self.eps = eps
        self.eps_decay = eps_decay
        self.actions = []

        for i in range(7):
            self.actions.append(i)
        self.Q = {}
        for action in self.actions:
            self.Q[action] = collections.defaultdict(int)
        self.rewards = []

```

Next, in the `get_action` function, we will find a possible movable position from the state every time, and generate a random number from 0 to 1 by calling the random function. If the random number is less than `eps`, we return a random action, otherwise I will be in the Q list Find the largest value among all actions and states, and return the action corresponding to this value, update `eps` to `1-eps_decay` and return this action. In the `get_action` function, we will find a possible movable position from the state every time, and generate a random number from 0 to 1 by calling the random function. If the random number is less than `eps`, we return a random action, otherwise I will be in the Q list Find the largest value among all actions and states, and return the action corresponding to this value, update `eps` to `1-eps_decay` and return this action. The `save` function is used to save the trained parameters. The `update` function is used to update the Q list and reward list according to the current and last status and action.

```
def get_action(self, s):...

def save(self, path):
    if os.path.isfile(path):
        os.remove(path)
    with open(path, 'wb') as f:
        pickle.dump(self, f)

def update(self, s, s_, a, a_, r):
    if s_ is not None:
        possible_actions = []
        Qs = []
        for action in self.actions:
            if s_[action] == '0':
                possible_actions.append(action)
        for action in possible_actions:
            Qs.append(self.Q[action][s_])
        self.Q[a][s] += self.alpha * (r + self.gamma * np.max(Qs) - self.Q[a][s])
    else:
        self.Q[a][s] += self.alpha * (r - self.Q[a][s])
    self.rewards.append(r)
```

In order to train the Qlearn algorithm, I implemented a Teacher class. Like the previous game, I hope to realize a perfect Teacher whose movement is always optimal, but it is difficult to realize such a Teacher in the Connect4 game, so I only designed three functions to realize the Teacher, including priority Take the step that can win, take the step that does not allow the opponent to win, and take a random step. Therefore, it is better to use the minimax algorithm to train the Qlearn algorithm.

```

class Teacher:
    def set_win(self, board, player=1):
        columns = getValidColumns(board)
        for column in columns:
            new_board = np.copy(board)
            set_move(new_board, column, player)
            if checkWin(board, player):
                return column

        return None

    def set_blockOpponentWin(self, board):
        return self.set_win(board, 2)

    def set_random(self, board):
        while True:
            col = random.randint(0, 6)
            if board[0][col] == 0:
                return col

    def move(self, board):
        if self.set_win(board):
            return self.set_win(board)
        elif self.set_blockOpponentWin(board):
            return self.set_blockOpponentWin(board)
        else:
            return self.set_random(board)

```

#### 4. Baseline algorithm

The baseline function is a default opponent which will play these games against my algorithms. Its design idea is to find all possible moving columns, traverse them and simulate moving it and judge whether it will win, if it wins, move this column, if not, move to a column randomly.

```

def baseline(player):
    if is_terminal(board):
        return
    column = -1
    valid_columns = getValidColumns(board)
    for valid_column in valid_columns:
        new_board = np.copy(board)
        set_move(new_board, valid_column, player)
        if checkWin(new_board, player):
            column = valid_column
            break
    if column == -1:
        while True:
            column = random.choice([0, 1, 2, 3, 4, 5, 6])
            if valid_move(column):
                break
    set_move(board, column, player)

```

### 3. Compare algorithms

The minimax algorithm and the default opponent algorithm do not require training, while the q-learn algorithm can be used after training or in games with other algorithms. Therefore, in the performance comparison of the q-learn algorithm, we use the q-learn mode training 5 thousand times, training 50 thousand times and training 500 thousand times to facilitate the comparison of q-learn algorithms with different training scales. Because the running time of the minimax algorithm is very long, in the algorithm performance comparison, the number of games is 20, 50 and 100 respectively.

#### 1. Tic Tac Toe

##### 1) minimax VS default opponent

iteration times	minimax win rate	default opponent win rate	draw rate
20	90%	0%	10%
50	82%	0%	18%
100	88%	0%	12%

##### 2) q-learn VS default opponent

###### Q-learn algorithm training 5k times

iteration times	q-learn win rate	default opponent win rate	draw rate
20	55%	45%	0%
50	44%	50%	6%
100	41%	49%	10%

###### Q-learn algorithm training 50k times

iteration times	q-learn win rate	default opponent win rate	draw rate
20	60%	30%	10%
50	62%	28%	10%
100	57%	28%	15%

###### Q-learn algorithm training 500k times

iteration times	q-learn win rate	default opponent win rate	draw rate
20	75%	5%	20%
50	76%	6%	18%
100	78%	6%	16%

### 3) q-learn VS minimax

Q-learn algorithm training 5k times

iteration times	q-learn win rate	minimax win rate	draw rate
20	55%	45%	10%
50	74%	10%	16%
100	81%	7%	12%

Q-learn algorithm training 50k times

iteration times	q-learn win rate	minimax win rate	draw rate
20	75%	25%	0%
50	84%	14%	2%
100	82%	15%	3%

Q-learn algorithm training 500k times

iteration times	q-learn win rate	minimax win rate	draw rate
20	90%	5%	5%
50	80%	12%	8%
100	85%	9%	6%

## 2. Connect 4

### 1) minimax VS default opponent

iteration times	minimax win rate	default opponent win rate	draw rate
20	90%	10%	0%
50	95%	5%	0%
100	100%	0%	0%

### 2) q-learn VS default opponent

Q-learn algorithm training 5k times

iteration times	q-learn win rate	default opponent win rate	draw rate
20	50%	40%	10%
50	64%	20%	16%
100	73%	12%	15%

Q-learn algorithm training 50k times

iteration times	q-learn win rate	default opponent win rate	draw rate
20	70%	30%	0%
50	82%	14%	4%
100	87%	13%	0%

Q-learn algorithm training 500k times

iteration times	q-learn win rate	default opponent win rate	draw rate
20	84%	16%	0%
50	86%	6%	8%
100	85%	8%	7%

3) q-learn VS minimax

Q-learn algorithm training 5k times

iteration times	q-learn win rate	minimax win rate	draw rate
20	45%	50%	5%
50	58%	36%	6%
100	70%	23%	7%

Q-learn algorithm training 50k times

iteration times	q-learn win rate	minimax win rate	draw rate
20	65%	30%	5%
50	74%	18%	8%
100	82%	12%	6%

Q-learn algorithm training 500k times

iteration times	q-learn win rate	minimax win rate	draw rate
20	75%	20%	5%
50	84%	12%	6%
100	91%	4%	5%

3. Analysing and discussing

1) In the game of tic-tac-toe, the minimax and q-learn algorithms play multiple games with the default opponent respectively, and the analysis results show that:

1. The worst result of the minimax algorithm is a tie, and the probability of losing the game is very small; even if the q-learn algorithm is trained 500 thousand times, there is still a probability of losing the game.

2. The winning rate of the minimax algorithm is in a stable range no matter how many games are played, about 80% to 90%; the q-learn algorithm has been trained from 5k times, 50k times to 500k times, and the winning rate has been increasing. As the number of battles increases, the winning rate also maintains an upward trend.

The reason is that minimax is a game theory algorithm, and it moves based on guessing that the opponent will make the most unfavorable action for itself, so it is not easy to lose the game. The q-learn algorithm is a reinforcement learning algorithm, which is used to learn an optimal strategy for an agent to take action in the environment. In each iteration, the agent observes the current state, and then selects an agent based on the Q value function of the current state. action. When the agent finishes performing this action, it receives a reward and enters a new state. In the new state, the agent again chooses an action according to the Q-value function, and iterates until it reaches the terminal state. Therefore, in the confrontation with the same opponent, it is easy to find a way to defeat the opponent through continuous learning, so the winning rate continues to increase.

2) In the Connect 4 game, the minimax and q-learn algorithms played multiple games with the default opponent respectively, and the analysis results show that:

1. The winning rate of the minimax algorithm is higher than that of the game of tic-tac-toe, but there is a certain probability of losing to the opponent, and the possibility of a draw is small.

2. The winning rate of the minimax algorithm is in a stable range no matter how many games are played, about 90% to 100%; the q-learn algorithm has been trained from 5k times, 50k times to 500k times, and the winning rate has been increasing. As the number of battles increases, the winning rate also maintains an upward trend.

Compared with tic-tac-toe, the performance of the q-learn algorithm is very similar, but the minimax algorithm has a higher probability of losing, because the rules of Connect 4 are more complicated, and the game is much more difficult than tic-tac-toe.

3) By analyzing the results of the minimax algorithm and the q-learn algorithm and the default opponent playing these two games, the following conclusions are drawn:

1. The minimax algorithm has a higher winning rate than the q-learn algorithm in the confrontation with the default opponent, but as the number of games between q-learn and the default opponent increases, the q-learn algorithm has a higher winning rate than the minimax algorithm.

2. The winning rate of the q-learn algorithm always increases in the confrontation with the default opponent, while the minimax winning rate does not change much, and the randomness of the change is relatively large, and there is no predictable improvement or decline.

4) By analyzing the results of minimax algorithm and q-learn algorithm playing tic-tac-toe game, the following conclusions are drawn.

1. In the first 20 games where the q-learn algorithm was trained 5k times, the winning rate of the q-learn algorithm was a little higher than that of minimax, but as the

number of games increased, the winning rate of the q-learn algorithm increased to 81%, while the minimax The win rate is only 7%, but the draw rate has improved somewhat.

2. As the number of q-learn algorithm training increases, in the first 20 games, the winning rate increases from 55% to 90%, while minimax decreases from 45% to 5%. The reason why the q-learn algorithm can achieve a higher winning rate than minimax after training 5k times is that the rules used by the Teacher class used to train the q-learn algorithm are the perfect rules of tic-tac-toe, so the q-learn algorithm learns and improves quickly. As the number of games increases, because the q-learn algorithm continues to learn how to fight against the opponent's strategy, the winning rate increases significantly. However, according to the minimax algorithm, it always predicts the most unfavorable characteristics of the opponent's actions, so the tie rate will increase.

5) By analyzing the results of playing the Connect 4 game with the minimax algorithm and the q-learn algorithm, the following conclusions are drawn.

1. In the first 20 games where the q-learn algorithm is trained 5k times, the winning rate of the q-learn algorithm is 45%, which is a little lower than the 55% of minimax, but as the number of games increases, the winning rate of the q-learn algorithm increases to 75%, while minimax's winning rate is only 20%, but the tie rate has improved somewhat.

2. As the number of q-learn algorithm training increases, in the first 20 games, the winning rate increases from 45% to 75%, while minimax decreases from 55% to 20%. Compared with tic-tac-toe, the q-learn algorithm has a lower winning rate than minimax when training 5k times, because the Teacher class used to train the q-learn algorithm is not perfect, so the learning efficiency of q-learn is not high. As the number of games increases, because the q-learn algorithm continues to learn how to fight against the opponent's strategy, the winning rate increases significantly. However, according to the minimax algorithm, it always predicts the most unfavorable characteristics of the opponent's actions, so the tie rate will increase.

6) By analyzing the results of playing these two games with the minimax algorithm and the q-learn algorithm, the following conclusions are drawn:

1. In the game of tic-tac-toe, when the q-learn algorithm is trained for 5k times, the winning rate of the first 20 rounds of minimax is lower than that of q-learn, but in the Connect 4 game, the winning rate of the first 20 rounds of minimax is higher than that of q-learn . But as the number of training increases and the number of games increases, the winning rate of q-learn is always higher than that of minimax.

2. As the number of games increases, the winning rate of the q-learn algorithm always increases in the confrontation with minimax, and the draw rate always increases.



## Appendix

### 1. Tic Tac Toe Minimax VS Baseline

```
1. import random
2. from math import inf as infinity
3. from random import choice
4.
5. HUMAN = -1
6. COMP = +1
7. board = [
8.     [0, 0, 0],
9.     [0, 0, 0],
10.    [0, 0, 0],
11.]
12. WIN = 0
13. LOSE = 0
14. DRAW = 0
15.
16.
17. def evaluate(state):
18.     if wins(state, COMP):
19.         score = +1
20.     elif wins(state, HUMAN):
21.         score = -1
22.     else:
23.         score = 0
24.
25.     return score
26.
27.
28. def wins(state, player):
29.     win_state = [
30.         [state[0][0], state[0][1], state[0][2]],
31.         [state[1][0], state[1][1], state[1][2]],
32.         [state[2][0], state[2][1], state[2][2]],
33.         [state[0][0], state[1][0], state[2][0]],
34.         [state[0][1], state[1][1], state[2][1]],
35.         [state[0][2], state[1][2], state[2][2]],
36.         [state[0][0], state[1][1], state[2][2]],
37.         [state[2][0], state[1][1], state[0][2]],
38.     ]
39.     if [player, player, player] in win_state:
40.         return True
41.     else:
```

```
42.         return False
43.
44.
45. def game_over(state):
46.     return wins(state, HUMAN) or wins(state, COMP)
47.
48.
49. def empty_cells(state):
50.     cells = []
51.
52.     for x, row in enumerate(state):
53.         for y, cell in enumerate(row):
54.             if cell == 0:
55.                 cells.append([x, y])
56.
57.     return cells
58.
59.
60. def valid_move(x, y):
61.     if [x, y] in empty_cells(board):
62.         return True
63.     else:
64.         return False
65.
66.
67. def set_move(x, y, player):
68.     if valid_move(x, y):
69.         board[x][y] = player
70.         return True
71.     else:
72.         return False
73.
74.
75. def minimax(state, depth, player):
76.     if player == COMP:
77.         best = [-1, -1, -infinity]
78.     else:
79.         best = [-1, -1, +infinity]
80.
81.     if depth == 0 or game_over(state):
82.         score = evaluate(state)
83.         return [-1, -1, score]
84.
```

```
85.     for cell in empty_cells(state):
86.         x, y = cell[0], cell[1]
87.         state[x][y] = player
88.         score = minimax(state, depth - 1, -player)
89.
90.         state[x][y] = 0
91.         score[0], score[1] = x, y
92.
93.         if player == COMP:
94.             if score[2] > best[2]:
95.                 best = score
96.             else:
97.                 if score[2] < best[2]:
98.                     best = score
99.         return best
100.
101.
102. def ai_turn(player):
103.     depth = len(empty_cells(board))
104.     if depth == 0 or game_over(board):
105.         return
106.     if depth == 9:
107.         x = choice([0, 1, 2])
108.         y = choice([0, 1, 2])
109.     else:
110.         move = minimax(board, depth, player)
111.         x, y = move[0], move[1]
112.         set_move(x, y, player)
113.
114.
115. def baseline(player):
116.     depth = len(empty_cells(board))
117.     if depth == 0 or game_over(board):
118.         return
119.
120.     cells = [
121.         [board[0][0], board[0][1], board[0][2]],
122.         [board[1][0], board[1][1], board[1][2]],
123.         [board[2][0], board[2][1], board[2][2]],
```

```
124.         [board[0][0], board[1][0], board[2][0]],
125.         [board[0][1], board[1][1], board[2][1]],
126.         [board[0][2], board[1][2], board[2][2]],
127.         [board[0][0], board[1][1], board[2][2]],
128.         [board[2][0], board[1][1], board[0][2]],
129.     ]
130.     x = -1
131.     y = -1
132.     for i in range(len(cells)):
133.         if 0 <= i <= 2:
134.             if cells[i][0] == cells[i][1] and ce
                lls[i][0] == player and cells[i][2] == 0:
135.                 x = i
136.                 y = 2
137.                 break
138.             elif cells[i][0] == cells[i][2] and
                cells[i][0] == player and cells[i][1] == 0:
139.                 x = i
140.                 y = 1
141.                 break
142.             elif cells[i][1] == cells[i][2] and
                cells[i][1] == player and cells[i][0] == 0:
143.                 x = i
144.                 y = 0
145.                 break
146.         elif 3 <= i <= 5:
147.             if cells[i][0] == cells[i][1] and ce
                lls[i][0] == player and cells[i][2] == 0:
148.                 x = 2
149.                 y = i - 3
150.                 break
151.             elif cells[i][0] == cells[i][2] and
                cells[i][0] == player and cells[i][1] == 0:
152.                 x = 1
153.                 y = i - 3
154.                 break
155.             elif cells[i][1] == cells[i][2] and
                cells[i][1] == player and cells[i][0] == 0:
156.                 x = 0
```

```

157.             y = i - 3
158.             break
159.         elif i == 6:
160.             if cells[i][0] == cells[i][1] and ce
                lls[i][0] == player and cells[i][2] == 0:
161.                 x = 2
162.                 y = 2
163.                 break
164.             elif cells[i][0] == cells[i][2] and
                cells[i][0] == player and cells[i][1] == 0:
165.                 x = 1
166.                 y = 1
167.                 break
168.             elif cells[i][1] == cells[i][2] and
                cells[i][1] == player and cells[i][0] == 0:
169.                 x = 0
170.                 y = 0
171.                 break
172.
173.         if x == -1 and y == -1:
174.             while True:
175.                 x = choice([0, 1, 2])
176.                 y = choice([0, 1, 2])
177.                 if valid_move(x, y):
178.                     break
179.             set_move(x, y, player)
180.
181.
182.     def changeWIN():
183.         global WIN
184.         WIN += 1
185.
186.
187.     def changeLOSE():
188.         global LOSE
189.         LOSE += 1
190.
191.
192.     def changeDRAW():
193.         global DRAW
194.         DRAW += 1
195.
196.
197.     def changeBOARD():

```

```
198.     global board
199.     board = [
200.         [0, 0, 0],
201.         [0, 0, 0],
202.         [0, 0, 0]
203.     ]
204.
205.
206.     def reset():
207.         global WIN, LOSE, DRAW, board
208.         WIN, LOSE, DRAW = 0, 0, 0
209.
210.
211.     def gameplay():
212.         while wins(board, HUMAN) == False and wins(board, COMP) == False and len(empty_cells(board)) > 0:
213.             ai_turn(HUMAN)
214.             baseline(COMP)
215.             if wins(board, HUMAN):
216.                 changeWIN()
217.                 break
218.             elif wins(board, COMP):
219.                 changeLOSE()
220.                 break
221.             elif len(empty_cells(board)) == 0:
222.                 changeDRAW()
223.                 break
224.
225.
226.     def minimaxVSbaseline(first_player):
227.         if first_player == HUMAN:
228.             baseline(HUMAN)
229.             while True:
230.                 # agent move
231.                 ai_turn(COMP)
232.                 if wins(board, COMP):
233.                     changeWIN()
234.                     break
235.                 elif len(empty_cells(board)) == 0:
236.                     changeDRAW()
237.                     break
238.                 # teacher move
239.                 baseline(HUMAN)
```

```

240.         if wins(board, HUMAN):
241.             changeLOSE()
242.             break
243.         elif len(empty_cells(board)) == 0:
244.             changeDRAW()
245.             break
246.
247.
248. def runMinimaxVSBaseline(iters):
249.     for i in range(iters):
250.         if random.random() < 0.5:
251.             minimaxVSbaseline(HUMAN)
252.         else:
253.             minimaxVSbaseline(COMP)
254.         changeBOARD()
255.         print("Minimax Win rate: " + str(WIN / iters
          * 100) + "%")
256.         print("Minimax Lose rate: " + str(LOSE / ite
          rs * 100) + "%")
257.         print("Minimax Draw rate: " + str(DRAW / ite
          rs * 100) + "%")
258.
259.
260. if __name__ == '__main__':
261.     print("Minimax VS Baseline 20 times")
262.     runMinimaxVSBaseline(20)
263.     print("")
264.     reset()
265.
266.     print("Minimax VS Baseline 50 times")
267.     runMinimaxVSBaseline(50)
268.     print("")
269.     reset()
270.
271.     print("Minimax VS Baseline 100 times")
272.     runMinimaxVSBaseline(100)
273.     print("")
274.     reset()

```

## 2. Tic Tac Toe q-learn VS baseline and q-learn VS minimax

```

1. import os
2. import pickle
3. import collections
4. import numpy as np

```

```

5. import random
6.
7.
8. class QLearner:
9.     def __init__(self, alpha, gamma, eps, eps_decay
        =0.):
10.         self.alpha = alpha
11.         self.gamma = gamma
12.         self.eps = eps
13.         self.eps_decay = eps_decay
14.         self.actions = []
15.         for i in range(3):
16.             for j in range(3):
17.                 self.actions.append((i, j))
18.         self.Q = {}
19.         for action in self.actions:
20.             self.Q[action] = collections.defaultdict(
                int)
21.         self.rewards = []
22.
23.     def get_action(self, s):
24.         possible_actions = [a for a in self.actions
            if s[a[0]*3 + a[1]] == '0']
25.         if random.random() < self.eps:
26.             action = possible_actions[random.randomin
                t(0, len(possible_actions)-1)]
27.         else:
28.             values = np.array([self.Q[a][s] for a i
                n possible_actions])
29.             col_max = np.where(values == np.max(val
                ues))[0]
30.             if len(col_max) > 1:
31.                 col = np.random.choice(col_max, 1)[
                    0]
32.             else:
33.                 col = col_max[0]
34.                 action = possible_actions[col]
35.
36.         self.eps *= (1.-self.eps_decay)
37.         return action
38.
39.     def save(self, path):
40.         if os.path.isfile(path):
41.             os.remove(path)

```





```

23.THREEINROW = 10
24.TWOINROW = 3
25.MIDDLE_COLUMN = 2
26.
27.AGING_PENALTY = 3
28.
29.OPP_FOURINROW = -1000000000000000000000000
30.OPP_THREEINROW = -12
31.OPP_TWOINROW = -4
32.GAME_COUNT = 0
33.
34.def evaluate(state):
35.    if wins(state, COMP):
36.        score = +1
37.    elif wins(state, HUMAN):
38.        score = -1
39.    else:
40.        score = 0
41.
42.    return score
43.
44.
45.def wins(state, player):
46.    win_state = [
47.        [state[0][0], state[0][1], state[0][2]],
48.        [state[1][0], state[1][1], state[1][2]],
49.        [state[2][0], state[2][1], state[2][2]],
50.        [state[0][0], state[1][0], state[2][0]],
51.        [state[0][1], state[1][1], state[2][1]],
52.        [state[0][2], state[1][2], state[2][2]],
53.        [state[0][0], state[1][1], state[2][2]],
54.        [state[2][0], state[1][1], state[0][2]],
55.    ]
56.    if [player, player, player] in win_state:
57.        return True
58.    else:
59.        return False
60.
61.
62.def game_over(state):
63.    return wins(state, HUMAN) or wins(state, COMP)
64.
65.

```

```

66. def empty_cells(state):
67.     cells = []
68.
69.     for x, row in enumerate(state):
70.         for y, cell in enumerate(row):
71.             if cell == 0:
72.                 cells.append([x, y])
73.
74.     return cells
75.
76.
77. def valid_move(x, y):
78.     if [x, y] in empty_cells(board):
79.         return True
80.     else:
81.         return False
82.
83.
84. def set_move(x, y, player):
85.     if valid_move(x, y):
86.         board[x][y] = player
87.         return True
88.     else:
89.         return False
90.
91.
92. def minimax(state, depth, player):
93.     if player == COMP:
94.         best = [-1, -1, -infinity]
95.     else:
96.         best = [-1, -1, +infinity]
97.
98.     if depth == 0 or game_over(state):
99.         score = evaluate(state)
100.        return [-1, -1, score]
101.
102.        for cell in empty_cells(state):
103.            x, y = cell[0], cell[1]
104.            state[x][y] = player
105.            score = minimax(state, depth - 1, -player)
106.            state[x][y] = 0
107.            score[0], score[1] = x, y
108.

```

```

109.         if player == COMP:
110.             if score[2] > best[2]:
111.                 best = score
112.         else:
113.             if score[2] < best[2]:
114.                 best = score
115.
116.     return best
117.
118.
119. def ai_turn(player):
120.     depth = len(empty_cells(board))
121.     if depth == 0 or game_over(board):
122.         return
123.     if depth == 9:
124.         x = choice([0, 1, 2])
125.         y = choice([0, 1, 2])
126.     else:
127.         move = minimax(board, depth, player)
128.         x, y = move[0], move[1]
129.     set_move(x, y, player)
130.
131.
132. def is_terminal(state):
133.     return wins(state, COMP) or wins(state, HUMAN) or len(empty_cells(state)) == 0
134.
135.
136. def minimax1(state, depth, alpha, beta, isCOMP):
137.     possibleMoves = empty_cells(state)
138.     if is_terminal(state) or depth == 0:
139.         if is_terminal(state):
140.             if wins(state, COMP):
141.                 score = WIN_SCORE + depth * 3
142.             return (None, score)
143.             elif wins(state, HUMAN):
144.                 score = LOSE_SCORE - depth * 3
145.             return (None, score)
146.         else:
147.             return (None, 0)
148.     else:
149.         return (None, score_position(board,
COMP))

```

```
150.
151.     if isCOMP:
152.         value = -math.inf
153.         row, col = random.choice(possibleMoves)
154.
155.         for x, y in possibleMoves:
156.             new_board = np.copy(state)
157.             set_move(x, y, COMP)
158.             newScore = minimax1(new_board, depth
- 1, alpha, beta, False)[1]
159.             if newScore > value:
160.                 value = newScore
161.                 row = x
162.                 col = y
163.                 alpha = max(alpha, value)
164.                 if alpha >= beta:
165.                     break
166.
167.         return row, col, value
168.
169.     else:
170.         value = math.inf
171.         row, col = random.choice(possibleMoves)
172.
173.         for x, y in possibleMoves:
174.             new_board = np.copy(state)
175.             set_move(new_board, col, HUMAN)
176.             newScore = minimax1(new_board, depth
- 1, alpha, beta, True)[1]
177.             if newScore < value:
178.                 value = newScore
179.                 row = x
180.                 col = y
181.                 beta = min(beta, value)
182.                 if alpha >= beta:
183.                     break
184.
185.         return row, col, value
186.
187. def score_position(state, player):
188.     score = 0
189.     # score center column
```

```

190.         center_array = [state[i][3] for i in range(6)
191. ]
192.         # for i in range(6):
193.         #     center_array.append(state[i][3])
194.         center_count = center_array.count(player)
195.         score += center_count * MIDDLE_COLUMN
196.
197.         # score horizontal
198.         for r in range(6):
199.             row_array = [state[r][i] for i in range(
200. 7)]
201.             for c in range(4):
202.                 window = row_array[c:c + WINDOW LENG
203. TH]
204.                 score += evaluate_window(window, pla
205. yer)
206.
207.         # score vertical
208.         for c in range(7):
209.             col_array = [state[i][c] for i in range(
210. 6)]
211.             for r in range(3):
212.                 window = col_array[r:r + WINDOW LENG
213. TH]
214.                 score += evaluate_window(window, pla
215. yer)
216.
217.         # score positive sloped diagonal
218.         for r in range(3):
219.             for c in range(4):
220.                 window = [state[r + i][c + i] for i
221. in range(4)]
222.                 score += evaluate_window(window, pla
223. yer)
224.
225.         # score negative sloped diagonal
226.         for r in range(3):
227.             for c in range(4):
228.                 window = [state[r + 3 - i][c + i] fo
229. r i in range(4)]
230.                 score += evaluate_window(window, pla
231. yer)
232.
233.         return score

```

```
223.
224.
225. def evaluate_window(window, player):
226.     score = 0
227.     opp_piece = HUMAN
228.     if player == HUMAN:
229.         opp_piece = COMP
230.
231.     if window.count(player) == 4:
232.         score += FOURINROW
233.         return score
234.     elif window.count(player) == 3 and window.co
unt(0) == 1:
235.         score += THREEINROW
236.     elif window.count(player) == 2 and window.co
unt(0) == 2:
237.         score += TWOINROW
238.
239.     if window.count(opp_piece) == 4:
240.         score -= FOURINROW
241.         return score
242.     elif window.count(opp_piece) == 3 and window.
count(0) == 1:
243.         score -= THREEINROW
244.     elif window.count(opp_piece) == 2 and window.
count(0) == 2:
245.         score -= TWOINROW
246.         return score
247.
248.
249.
250. def baseline(player):
251.     depth = len(empty_cells(board))
252.     if depth == 0 or game_over(board):
253.         return
254.
255.     cells = [
256.         [board[0][0], board[0][1], board[0][2]],
257.         [board[1][0], board[1][1], board[1][2]],
258.         [board[2][0], board[2][1], board[2][2]],
```

```
259.         [board[0][0], board[1][0], board[2][0]],
260.         [board[0][1], board[1][1], board[2][1]],
261.         [board[0][2], board[1][2], board[2][2]],
262.         [board[0][0], board[1][1], board[2][2]],
263.         [board[2][0], board[1][1], board[0][2]],
264.     ]
265.     x = -1
266.     y = -1
267.     for i in range(len(cells)):
268.         if 0 <= i <= 2:
269.             if cells[i][0] == cells[i][1] and ce
                lls[i][0] == player and cells[i][2] == 0:
270.                 x = i
271.                 y = 2
272.                 break
273.             elif cells[i][0] == cells[i][2] and
                cells[i][0] == player and cells[i][1] == 0:
274.                 x = i
275.                 y = 1
276.                 break
277.             elif cells[i][1] == cells[i][2] and
                cells[i][1] == player and cells[i][0] == 0:
278.                 x = i
279.                 y = 0
280.                 break
281.             elif 3 <= i <= 5:
282.                 if cells[i][0] == cells[i][1] and ce
                    lls[i][0] == player and cells[i][2] == 0:
283.                     x = 2
284.                     y = i - 3
285.                     break
286.                 elif cells[i][0] == cells[i][2] and
                    cells[i][0] == player and cells[i][1] == 0:
287.                     x = 1
288.                     y = i - 3
289.                     break
290.                 elif cells[i][1] == cells[i][2] and
                    cells[i][1] == player and cells[i][0] == 0:
291.                     x = 0
```



```
292.             y = i - 3
293.             break
294.         elif i == 6:
295.             if cells[i][0] == cells[i][1] and ce
                lls[i][0] == player and cells[i][2] == 0:
296.                 x = 2
297.                 y = 2
298.                 break
299.             elif cells[i][0] == cells[i][2] and
                cells[i][0] == player and cells[i][1] == 0:
300.                 x = 1
301.                 y = 1
302.                 break
303.             elif cells[i][1] == cells[i][2] and
                cells[i][1] == player and cells[i][0] == 0:
304.                 x = 0
305.                 y = 0
306.                 break
307.
308.         if x == -1 and y == -1:
309.             while True:
310.                 x = choice([0, 1, 2])
311.                 y = choice([0, 1, 2])
312.                 if valid_move(x, y):
313.                     break
314.             set_move(x, y, player)
315.
316.
317. def changeWIN():
318.     global WIN
319.     WIN += 1
320.
321.
322. def changeLOSE():
323.     global LOSE
324.     LOSE += 1
325.
326.
327. def changeDRAW():
328.     global DRAW
329.     DRAW += 1
330.
331.
332. def changeGAME_COUNT():
```

```
333.     global GAME_COUNT
334.     GAME_COUNT += 1
335.
336.
337. def changeBoard():
338.     global board
339.     board = [
340.         [0, 0, 0],
341.         [0, 0, 0],
342.         [0, 0, 0]
343.     ]
344.
345.
346. def reset():
347.     global WIN, LOSE, DRAW, board
348.     WIN, LOSE, DRAW = 0, 0, 0
349.
350.
351. def gameplay():
352.     while wins(board, HUMAN) == False and wins(board, COMP) == False and len(empty_cells(board)) > 0:
353.         ai_turn(HUMAN)
354.         baseline(COMP)
355.         if wins(board, HUMAN):
356.             changeWIN()
357.             break
358.         elif wins(board, COMP):
359.             changeLOSE()
360.             break
361.         elif len(empty_cells(board)) == 0:
362.             changeDRAW()
363.             break
364.
365.
366. def trainGamePlay(first_player, teacher, agent):
367.     if first_player == HUMAN:
368.         action = teacher.move(board)
369.         set_move(action[0], action[1], HUMAN)
370.
371.     prev_board = toString(board)
372.     prev_action = agent.get_action(prev_board)
373.
```

```

374.         while True:
375.             # agent move
376.             set_move(prev_action[0], prev_action[1],
COMP)
377.             if wins(board, COMP):
378.                 reward = 1
379.                 break
380.             elif len(empty_cells(board)) == 0:
381.                 reward = 0
382.                 break
383.             # teacher move
384.             action = teacher.move(board)
385.             set_move(action[0], action[1], HUMAN)
386.             if wins(board, HUMAN):
387.                 reward = -1
388.                 break
389.             elif len(empty_cells(board)) == 0:
390.                 reward = 0
391.                 break
392.             else:
393.                 reward = 0
394.                 new_board = toString(board)
395.                 new_action = agent.get_action(new_board)
396.                 agent.update(prev_board, new_board, prev
_action, new_action, reward)
397.                 prev_board = new_board
398.                 prev_action = new_action
399.
400.             agent.update(prev_board, None, prev_action,
None, reward)
401.
402.
403.     def teacherPlay(agent):
404.         teacher = Teacher()
405.         if random.random() < 0.5:
406.             trainGamePlay(HUMAN, teacher, agent)
407.         else:
408.             trainGamePlay(COMP, teacher, agent)
409.
410.
411.     def train(agent, iters):
412.         while GAME_COUNT < iters:
413.             teacherPlay(agent)

```

```

414.         changeGAME_COUNT()
415.         if GAME_COUNT % 1000 == 0:
416.             print("Games played: %i" % GAME_COUNT)
417.         changeBoard()
418.         agent.save('q_agent.pkl')
419.
420.
421.     def toString(board):
422.         ans = ''
423.         for row in board:
424.             for col in row:
425.                 ans += str(col)
426.         return ans
427.
428.     # qlearn is COMP
429.     def qlearnVSminimax(first_player, agent):
430.         if first_player == HUMAN:
431.             ai_turn(HUMAN)
432.             prev_board = toString(board)
433.             prev_action = agent.get_action(prev_board)
434.
435.             while True:
436.                 # agent move
437.                 set_move(prev_action[0], prev_action[1],
438.                           COMP)
439.
440.                 if wins(board, COMP):
441.                     changeWIN()
442.                     reward = 1
443.                     break
444.                 elif len(empty_cells(board)) == 0:
445.                     changeDRAW()
446.                     reward = 0
447.                     break
448.                 # teacher move
449.                 ai_turn(HUMAN)
450.                 if wins(board, HUMAN):
451.                     reward = -1
452.                     changeLOSE()
453.                     break
454.                 elif len(empty_cells(board)) == 0:
455.                     reward = 0
456.                     changeDRAW()
457.                     break

```

```

456.         else:
457.             reward = 0
458.             new_board = toString(board)
459.             new_action = agent.get_action(new_board)
460.             agent.update(prev_board, new_board, prev
                _action, new_action, reward)
461.             prev_board = new_board
462.             prev_action = new_action
463.
464.             agent.update(prev_board, None, prev_action,
                None, reward)
465.
466.
467. def runQlearnVSMinimax(agent, iters):
468.     while GAME_COUNT < iters:
469.         if random.random() < 0.5:
470.             qlearnVSminimax(HUMAN, agent)
471.         else:
472.             qlearnVSminimax(COMP, agent)
473.             changeGAME_COUNT()
474.             changeBoard()
475.             print("Qlearn Win rate: " + str(WIN / iters
                * 100) + "%")
476.             print("Qlearn Lose rate: " + str(LOSE / iter
                s * 100) + "%")
477.             print("Qlearn Draw rate: " + str(DRAW / iter
                s * 100) + "%")
478.
479.
480. def qlearnVSbaseline(first_player, agent):
481.     if first_player == HUMAN:
482.         baseline(HUMAN)
483.         prev_board = toString(board)
484.         prev_action = agent.get_action(prev_board)
485.
486.         while True:
487.             # agent move
488.             set_move(prev_action[0], prev_action[1],
                COMP)
489.             if wins(board, COMP):
490.                 changeWIN()
491.                 reward = 1
492.                 break

```

```

493.         elif len(empty_cells(board)) == 0:
494.             changeDRAW()
495.             reward = 0
496.             break
497.         # teacher move
498.         baseline(HUMAN)
499.         if wins(board, HUMAN):
500.             reward = -1
501.             changeLOSE()
502.             break
503.         elif len(empty_cells(board)) == 0:
504.             reward = 0
505.             changeDRAW()
506.             break
507.         else:
508.             reward = 0
509.             new_board = toString(board)
510.             new_action = agent.get_action(new_board)
511.             agent.update(prev_board, new_board, prev
                _action, new_action, reward)
512.             prev_board = new_board
513.             prev_action = new_action
514.
515.             agent.update(prev_board, None, prev_action,
                None, reward)
516.
517.
518. def runQlearnVSBaseline(agent, iters):
519.     while GAME_COUNT < iters:
520.         if random.random() < 0.5:
521.             qlearnVsbaseline(HUMAN, agent)
522.         else:
523.             qlearnVsbaseline(COMP, agent)
524.         changeGAME_COUNT()
525.         changeBoard()
526.         print("Qlearn Win rate: " + str(WIN / iters
            * 100) + "%")
527.         print("Qlearn Lose rate: " + str(LOSE / iter
            s * 100) + "%")
528.         print("Qlearn Draw rate: " + str(DRAW / iter
            s * 100) + "%")

```

```
1. import os
2. import pickle
3. from Agent import QLearner
4. from Game import train, runQlearnVSMinimax, runQlea
   rnVSBaseline
5.
6.
7. class PlayGame:
8.     def __init__(self, alpha=0.5, gamma=0.9, epsilo
       n=0.1):
9.         self.alpha = alpha
10.        self.gamma = gamma
11.        self.epsilon = epsilon
12.        self.qtable = {}
13.        if os.path.isfile('q_agent.pkl'):
14.            with open('q_agent.pkl', 'rb') as f:
15.                self.agent = pickle.load(f)
16.        else:
17.            self.agent = QLearner(alpha, gamma, eps
              ilon)
18.        self.games_played = 0
19.
20.    def teach(self, iters):
21.        train(self.agent, iters)
22.
23.    def playQlearnVSMinimax(self, iters):
24.        # print('Qlearn VS Minimax...')
25.        runQlearnVSMinimax(self.agent, iters)
26.
27.    def playQlearnVSBaseline(self, iters):
28.        # print('Qlearn VS Baseline...')
29.        runQlearnVSBaseline(self.agent, iters)
30.
31.
32. if __name__ == '__main__':
33.     game_thread = PlayGame()
34.     # game_thread.teach(500000)
35.
36.     print("q-learn VS minimax 20 times")
37.     game_thread.playQlearnVSMinimax(20)
38.     print("")
39.
40.     print("q-learn VS minimax 50 times")
41.     game_thread.playQlearnVSMinimax(50)
```

```
42.     print("")
43.
44.     print("q-learn VS minimax 100 times")
45.     game_thread.playQlearnVSMinimax(100)
46.     print("")
```

```
1. import random
2. import Game
3.
4.
5. class Teacher:
6.     def set_win(self, board, player=1):
7.         cells = Game.empty_cells(board)
8.         for cell in cells:
9.             if board[cell[0]][cell[1]] == 0:
10.                 board[cell[0]][cell[1]] = player
11.                 if Game.wins(board, player):
12.                     board[cell[0]][cell[1]] = 0
13.                     return cell[0], cell[1]
14.             else:
15.                 board[cell[0]][cell[1]] = 0
16.
17.     def set_blockOpponentWin(self, board):
18.         return self.set_win(board, 2)
19.
20.     def set_twoThreatToWin(self, board):
21.         if board[1][0] == 1 and board[0][1] == 1:
22.             if board[0][0] == 0 and board[2][0] ==
23. 0 and board[0][2] == 0:
24.                 return 0, 0
25.             elif board[1][1] == 0 and board[2][1] =
26. = 0 and board[1][2] == 0:
27.                 return 1, 1
28.             elif board[1][0] == 1 and board[2][1] == 1:
29.
30.                 if board[2][0] == 0 and board[0][0] ==
31. 0 and board[2][2] == 0:
32.                     return 2, 0
33.                 elif board[1][1] == 0 and board[0][1] =
34. = 0 and board[1][2] == 0:
35.                     return 1, 1
36.                 elif board[2][1] == 1 and board[1][2] == 1:
```



```

32.         if board[2][2] == 0 and board[2][0] ==
           0 and board[0][2] == 0:
33.             return 2, 2
34.         elif board[1][1] == 0 and board[1][0] =
           = 0 and board[0][1] == 0:
35.             return 1, 1
36.         elif board[1][2] == 1 and board[0][1] == 1:

37.         if board[0][2] == 0 and board[0][0] ==
           0 and board[2][2] == 0:
38.             return 0, 2
39.         elif board[1][1] == 0 and board[1][0] =
           = 0 and board[2][1] == 0:
40.             return 1, 1
41.
42.         elif board[0][0] == 1 and board[2][2] == 1:

43.         if board[1][0] == 0 and board[2][1] ==
           0 and board[2][0] == 0:
44.             return 2, 0
45.         elif board[0][1] == 0 and board[1][2] =
           = 0 and board[0][2] == 0:
46.             return 0, 2
47.         elif board[2][0] == 1 and board[0][2] == 1:

48.         if board[2][1] == 0 and board[1][2] ==
           0 and board[2][2] == 0:
49.             return 2, 2
50.         elif board[1][0] == 0 and board[0][1] =
           = 0 and board[0][0] == 0:
51.             return 0, 0
52.         return None
53.
54.     def set_blockOpponentTwoThreatWin(self, board):

55.         corners = [board[0][0], board[2][0], board[
           0][2], board[2][2]]
56.         if board[1][0] == 2 and board[0][1] == 2:
57.             if board[0][0] == 0 and board[2][0] ==
               0 and board[0][2] == 0:
58.                 return 0, 0
59.             elif board[1][1] == 0 and board[2][1] =
               = 0 and board[1][2] == 0:
60.                 return 1, 1

```

```

61.         elif board[1][0] == 2 and board[2][1] == 2:
62.             if board[2][0] == 0 and board[0][0] ==
0 and board[2][2] == 0:
63.                 return 2, 0
64.             elif board[1][1] == 0 and board[0][1] =
= 0 and board[1][2] == 0:
65.                 return 1, 1
66.             elif board[2][1] == 2 and board[1][2] == 2:
67.                 if board[2][2] == 0 and board[2][0] ==
0 and board[0][2] == 0:
68.                     return 2, 2
69.                 elif board[1][1] == 0 and board[1][0] =
= 0 and board[0][1] == 0:
70.                     return 1, 1
71.                 elif board[1][2] == 2 and board[0][1] == 2:
72.                     if board[0][2] == 0 and board[0][0] ==
0 and board[2][2] == 0:
73.                         return 0, 2
74.                     elif board[1][1] == 0 and board[1][0] =
= 0 and board[2][1] == 0:
75.                         return 1, 1
76.                 # if we have two corners, try to set the ce
nter
77.                 elif corners.count(0) == 1 and corners.coun
t(2) == 2:
78.                     return 1, 2
79.                 elif board[0][0] == 2 and board[2][2] == 2:
80.                     if board[1][0] == 0 and board[2][1] ==
0 and board[2][0] == 0:
81.                         return 2, 0
82.                     elif board[0][1] == 0 and board[2][1] =
= 0 and board[0][2] == 0:
83.                         return 0, 2
84.                     elif board[2][0] == 2 and board[0][2] == 2:
85.                         if board[2][1] == 0 and board[1][2] ==
0 and board[2][2] == 0:
86.                             return 2, 2
87.                         elif board[1][0] == 0 and board[0][1] =
= 0 and board[0][0] == 0:

```

```
88.         return 0, 0
89.     return None
90.
91.     def set_center(self, board):
92.         if board[1][1] == 0:
93.             return 1, 1
94.         return None
95.
96.     def set_corner(self, board):
97.         # pick opposite corner
98.         if board[0][0] == 2 and board[2][2] == 0:
99.             return 2, 2
100.        elif board[2][2] == 2 and board[0][0] ==
0:
101.            return 0, 0
102.        elif board[0][2] == 2 and board[2][0] ==
0:
103.            return 2, 0
104.        elif board[2][0] == 2 and board[0][2] ==
0:
105.            return 0, 2
106.
107.        if board[0][0] == 0:
108.            return 0, 0
109.        elif board[2][0] == 0:
110.            return 2, 0
111.        elif board[0][2] == 0:
112.            return 0, 2
113.        elif board[2][2] == 0:
114.            return 2, 2
115.        return None
116.
117.    def set_other(self, board):
118.        if board[1][0] == 0:
119.            return 1, 0
120.        elif board[2][1] == 0:
121.            return 2, 1
122.        elif board[1][2] == 0:
123.            return 1, 2
124.        elif board[0][1] == 0:
125.            return 0, 1
126.        return None
127.
128.    def set_random(self, board):
```

```

129.         while True:
130.             x = random.randint(0, 2)
131.             y = random.randint(0, 2)
132.             if board[x][y] == 0:
133.                 return x, y
134.
135.     def move(self, board):
136.         if random.random() > 0.8:
137.             return self.set_random(board)
138.         if self.set_win(board):
139.             return self.set_win(board)
140.         if self.set_blockOpponentWin(board):
141.             return self.set_blockOpponentWin(board)
142.         if self.set_blockOpponentTwoThreatWin(board):
143.             return self.set_blockOpponentTwoThreatWin(board)
144.         if self.set_center(board):
145.             return self.set_center(board)
146.         if self.set_corner(board):
147.             return self.set_corner(board)
148.         if self.set_other(board):
149.             return self.set_other(board)
150.         return self.set_random(board)

```

### 3. Connect 4

```

1. import os
2. import pickle
3. import collections
4. import numpy as np
5. import random
6.
7.
8. class QLearner:
9.     def __init__(self, alpha, gamma, eps, eps_decay=0.):
10.         self.alpha = alpha
11.         self.gamma = gamma
12.         self.eps = eps
13.         self.eps_decay = eps_decay
14.         self.actions = []
15.
16.         for i in range(7):

```

```

17.         self.actions.append(i)
18.         self.Q = {}
19.         for action in self.actions:
20.             self.Q[action] = collections.defaultdict(
                int)
21.         self.rewards = []
22.
23.     def get_action(self, s):
24.         possible_actions = [a for a in self.actions
                if s[a] == '0']
25.         if random.random() < self.eps:
26.             action = possible_actions[random.randint(
                0, len(possible_actions) - 1)]
27.         else:
28.             values = np.array([self.Q[a][s] for a in
                possible_actions])
29.             col_max = np.where(values == np.max(values)) [0]
30.             if len(col_max) > 1:
31.                 col = np.random.choice(col_max, 1) [
                0]
32.             else:
33.                 col = col_max[0]
34.             action = possible_actions[col]
35.
36.         self.eps *= (1. - self.eps_decay)
37.         return action
38.
39.     def save(self, path):
40.         if os.path.isfile(path):
41.             os.remove(path)
42.         with open(path, 'wb') as f:
43.             pickle.dump(self, f)
44.
45.     def update(self, s, s_, a, a_, r):
46.         if s_ is not None:
47.             possible_actions = []
48.             Qs = []
49.             for action in self.actions:
50.                 if s_[action] == '0':
51.                     possible_actions.append(action)
52.
53.             for action in possible_actions:
                Qs.append(self.Q[action][s_])

```



```

38. def set_move(state, column, player):
39.     for i in range(6):
40.         if state[i][column] != 0:
41.             state[i - 1][column] = player
42.             break
43.         elif i == 5:
44.             state[i][column] = player
45.
46.
47. def checkWin(state, player):
48.     for i in range(6):
49.         for j in range(4):
50.             if state[i][j] == state[i][j + 1] == state[i][j + 2] == state[i][j + 3] == player:
51.                 return True
52.
53.     for i in range(7):
54.         for j in range(3):
55.             if state[j][i] == state[j + 1][i] == state[j + 2][i] == state[j + 3][i] == player:
56.                 return True
57.
58.     for i in range(3):
59.         for j in range(4):
60.             if state[i][j] == state[i + 1][j + 1] == state[i + 2][j + 2] == state[i + 3][j + 3] == player:
61.                 return True
62.
63.     for i in range(3):
64.         for j in range(3, 7):
65.             if state[i][j] == state[i + 1][j - 1] == state[i + 2][j - 2] == state[i + 3][j - 3] == player:
66.                 return True
67.     return False
68.
69.
70. def getValidColumns(state):
71.     validColumns = []
72.     for i in range(7):
73.         if state[0][i] == 0:
74.             validColumns.append(i)
75.     return validColumns

```

```

76.
77.
78. def is_terminal(state):
79.     return checkWin(state, HUMAN) or checkWin(state,
80.         COMP) or len(getValidColumns(state)) == 0
81.
82. def valid_move(col):
83.     if board[0][col] == 0:
84.         return True
85.     else:
86.         return False
87.
88.
89. def minimax(state, depth, alpha, beta, isCOMP):
90.     possibleMoves = getValidColumns(state)
91.     if is_terminal(state) or depth == 0:
92.         if is_terminal(state):
93.             if checkWin(state, COMP):
94.                 score = WIN_SCORE + depth * 3
95.                 return (None, score)
96.             elif checkWin(state, HUMAN):
97.                 score = LOSE_SCORE - depth * 3
98.                 return (None, score)
99.             else:
100.                 return (None, 0)
101.         else:
102.             return (None, score_position(board,
103.                 COMP))
104.     if isCOMP:
105.         value = -math.inf
106.         column = random.choice(possibleMoves)
107.         for col in possibleMoves:
108.             new_board = np.copy(state)
109.             set_move(new_board, col, COMP)
110.             newScore = minimax(new_board, depth
111.                 - 1, alpha, beta, False)[1]
112.             if newScore > value:
113.                 value = newScore
114.                 column = col
115.             alpha = max(alpha, value)
116.             if alpha >= beta:
117.                 break

```



```

117.
118.         return column, value
119.
120.     else:
121.         value = math.inf
122.         column = random.choice(possibleMoves)
123.         for col in possibleMoves:
124.             new_board = np.copy(state)
125.             set_move(new_board, col, HUMAN)
126.             newScore = minimax(new_board, depth
- 1, alpha, beta, True)[1]
127.             if newScore < value:
128.                 value = newScore
129.                 column = col
130.                 beta = min(beta, value)
131.                 if alpha >= beta:
132.                     break
133.
134.         return column, value
135.
136.
137. def score_position(state, player):
138.     score = 0
139.     # score center column
140.     center_array = [state[i][3] for i in range(6)
141. ]
142.     # for i in range(6):
143.     #     center_array.append(state[i][3])
144.     center_count = center_array.count(player)
145.     score += center_count * MIDDLE_COLUMN
146.
147.     # score horizontal
148.     for r in range(6):
149.         row_array = [state[r][i] for i in range(
150. 7)]
151.         for c in range(4):
152.             window = row_array[c:c + WINDOW LENG
153. TH]
154.             score += evaluate_window(window, pla
155. yer)
156.
157.     # score vertical
158.     for c in range(7):

```

```

155.         col_array = [state[i][c] for i in range(
        6)]
156.         for r in range(3):
157.             window = col_array[r:r + WINDOW LENG
        TH]
158.             score += evaluate_window(window, pla
        yer)
159.
160.         # score positive sloped diagonal
161.         for r in range(3):
162.             for c in range(4):
163.                 window = [state[r + i][c + i] for i
        in range(4)]
164.                 score += evaluate_window(window, pla
        yer)
165.
166.         # score negative sloped diagonal
167.         for r in range(3):
168.             for c in range(4):
169.                 window = [state[r + 3 - i][c + i] fo
        r i in range(4)]
170.                 score += evaluate_window(window, pla
        yer)
171.
172.         return score
173.
174.
175. def evaluate_window(window, player):
176.     score = 0
177.     opp_piece = HUMAN
178.     if player == HUMAN:
179.         opp_piece = COMP
180.
181.     if window.count(player) == 4:
182.         score += FOURINROW
183.         return score
184.     elif window.count(player) == 3 and window.co
        unt(0) == 1:
185.         score += THREEINROW
186.     elif window.count(player) == 2 and window.co
        unt(0) == 2:
187.         score += TWOINROW
188.
189.     if window.count(opp_piece) == 4:

```

```
190.         score -= FOURINROW
191.         return score
192.         elif window.count(opp_piece) == 3 and window.
count(0) == 1:
193.             score -= THREEINROW
194.             elif window.count(opp_piece) == 2 and window.
count(0) == 2:
195.                 score -= TWOINROW
196.         return score
197.
198.
199. def changeWIN():
200.     global WIN
201.     WIN += 1
202.
203.
204. def changeLOSE():
205.     global LOSE
206.     LOSE += 1
207.
208.
209. def changeDRAW():
210.     global DRAW
211.     DRAW += 1
212.
213.
214. def changeBOARD():
215.     global board
216.     board = [[0, 0, 0, 0, 0, 0, 0],
217.              [0, 0, 0, 0, 0, 0, 0],
218.              [0, 0, 0, 0, 0, 0, 0],
219.              [0, 0, 0, 0, 0, 0, 0],
220.              [0, 0, 0, 0, 0, 0, 0],
221.              [0, 0, 0, 0, 0, 0, 0]]
222.
223.
224. def changeGAME_COUNT():
225.     global GAME_COUNT
226.     GAME_COUNT += 1
227.
228.
229. def reset():
230.     global WIN, LOSE, DRAW, GAME_COUNT
231.     WIN = 0
```

```

232.     LOSE = 0
233.     DRAW = 0
234.     GAME_COUNT = 0
235.
236.
237.     def baseline(player):
238.         if is_terminal(board):
239.             return
240.         column = -1
241.         valid_columns = getValidColumns(board)
242.         for valid_column in valid_columns:
243.             new_board = np.copy(board)
244.             set_move(new_board, valid_column, player)
245.
246.             if checkWin(new_board, player):
247.                 column = valid_column
248.                 break
249.         if column == -1:
250.             while True:
251.                 column = random.choice([0, 1, 2, 3,
252.                                         4, 5, 6])
253.                 if valid_move(column):
254.                     break
255.             set_move(board, column, player)
256.
257.     def testBaseline():
258.         itr = 0
259.         while True:
260.             itr += 1
261.             baseline(HUMAN)
262.             if is_terminal(board):
263.                 break
264.             baseline(COMP)
265.             if is_terminal(board):
266.                 break
267.
268.     def trainGamePlay(first_player, teacher, agent):
269.
270.         depth = 6
271.         round = 0
272.         if first_player == HUMAN:
273.             # action = teacher.move(board)

```

```

273.         action = minimax(board, 6, -math.inf, ma
           th.inf, True)[0]
274.         set_move(board, action, HUMAN)
275.
276.     prev_board = toString(board)
277.     prev_action = agent.get_action(prev_board)
278.
279.     while True:
280.         # agent move
281.         set_move(board, prev_action, COMP)
282.         if checkWin(board, COMP):
283.             reward = 1
284.             break
285.         elif len(getValidColumns(board)) == 0:
286.             reward = 0
287.             break
288.         # teacher move
289.         # action = teacher.move(board)
290.         # set_move(board, action, HUMAN)
291.         start_time = time.time()
292.         ai_col = minimax(board, depth - 1, -math.
           inf, math.inf, True)[0]
293.         end_time = time.time()
294.         set_move(board, ai_col, HUMAN)
295.         run_time = end_time - start_time
296.         print("Round " + str(round) + ": Time ta
           ken: " + str(run_time) + "s")
297.         print("")
298.         if checkWin(board, HUMAN):
299.             reward = -1
300.             break
301.         elif len(getValidColumns(board)) == 0:
302.             reward = 0
303.             break
304.         else:
305.             reward = 0
306.             new_board = toString(board)
307.             new_action = agent.get_action(new_board)
308.             agent.update(prev_board, new_board, prev
               _action, new_action, reward)
309.             prev_board = new_board
310.             prev_action = new_action
311.             if run_time > 1 and round > 3:

```

```

312.         depth -= 1
313.         round += 1
314.         agent.update(prev_board, None, prev_action,
315.             None, reward)
316.
317.     def teacherPlay(agent):
318.         teacher = Teacher()
319.         if random.random() < 0.5:
320.             trainGamePlay(HUMAN, teacher, agent)
321.         else:
322.             trainGamePlay(COMP, teacher, agent)
323.
324.
325.     def train(agent, iters):
326.         while GAME_COUNT < iters:
327.             teacherPlay(agent)
328.             changeGAME_COUNT()
329.             if GAME_COUNT % 1000 == 0:
330.                 print("Games played: %i" % GAME_COUNT)
331.                 changeBOARD()
332.                 agent.save('q_agent.pkl')
333.
334.
335.     # minimax as COMP and baseline as HUMAN
336.     def minimaxVSbaseline(first_player):
337.         depth = 6
338.         round = 0
339.         if first_player == HUMAN:
340.             baseline(HUMAN)
341.         while True:
342.             # agent move
343.             start_time = time.time()
344.             ai_col = minimax(board, depth - 1, -math.
345.                 inf, math.inf, True)[0]
346.             end_time = time.time()
347.             set_move(board, ai_col, COMP)
348.             run_time = end_time - start_time
349.             print("Round " + str(round) + ": Time taken: " + str(run_time) + "s")
350.             print("")
351.             if checkWin(board, COMP):
352.                 changeWIN()

```

```

352.         break
353.     elif len(getValidColumns(board)) == 0:
354.         changeDRAW()
355.         break
356.     # teacher move
357.     baseline(HUMAN)
358.     if checkWin(board, HUMAN):
359.         changeLOSE()
360.         break
361.     elif len(getValidColumns(board)) == 0:
362.         changeDRAW()
363.         break
364.
365.     # if run_time < 7.5 and round > 4:
366.     #     depth += 1
367.     # elif run_time > 12.5 and round > 4:
368.     #     depth -=
369.     if run_time > 3 and round > 4:
370.         depth -= 1
371.         round += 1
372.
373.
374. def runMinimaxVSBaseline(iters):
375.     for i in range(iters):
376.         if random.random() < 0.5:
377.             minimaxVSbaseline(HUMAN)
378.         else:
379.             minimaxVSbaseline(COMP)
380.         changeBOARD()
381.         print("Minimax Win rate: " + str(WIN / iters
          * 100) + "%")
382.         print("Minimax Lose rate: " + str(LOSE / ite
          rs * 100) + "%")
383.         print("Minimax Draw rate: " + str(DRAW / ite
          rs * 100) + "%")
384.         reset()
385.
386.
387. # qlearn as COMP and minimax as HUMAN
388. def qlearnVSminimax(first_player, agent):
389.     depth = 6
390.     round = 0
391.     if first_player == HUMAN:

```

```

392.         ai_col = minimax(board, depth - 1, -math.
    inf, math.inf, True)[0]
393.         set_move(board, ai_col, HUMAN)
394.         prev_board = toString(board)
395.         prev_action = agent.get_action(prev_board)
396.         while True:
397.             # agent move
398.             set_move(board, prev_action, COMP)
399.             if checkWin(board, COMP):
400.                 changeWIN()
401.                 reward = 1
402.                 break
403.             elif len(getValidColumns(board)) == 0:
404.                 changeDRAW()
405.                 reward = 0
406.                 break
407.             # teacher move
408.             start_time = time.time()
409.             ai_col = minimax(board, depth - 1, -math.
    inf, math.inf, True)[0]
410.             end_time = time.time()
411.             set_move(board, ai_col, HUMAN)
412.             run_time = end_time - start_time
413.             print("Round " + str(round) + ": Time ta
    ken: " + str(run_time) + "s")
414.             print("")
415.             if checkWin(board, HUMAN):
416.                 reward = -1
417.                 changeLOSE()
418.                 break
419.             elif len(getValidColumns(board)) == 0:
420.                 reward = 0
421.                 changeDRAW()
422.                 break
423.             else:
424.                 reward = 0
425.
426.             # if run_time < 7.5 and round > 4:
427.             #     depth += 1
428.             # elif run_time > 12.5 and round > 4:
429.             #     depth -= 1
430.             if run_time > 3 and round > 4:
431.                 depth -= 1
432.                 round += 1

```



```
433.
434.         new_board = toString(board)
435.         new_action = agent.get_action(new_board)

436.         agent.update(prev_board, new_board, prev
            _action, new_action, reward)
437.         prev_board = new_board
438.         prev_action = new_action
439.
440.         agent.update(prev_board, None, prev_action,
            None, reward)
441.
442.
443. def runQlearnVSMinimax(agent, iters):
444.     for i in range(iters):
445.         if random.random() < 0.5:
446.             qlearnVSminimax(HUMAN, agent)
447.         else:
448.             qlearnVSminimax(COMP, agent)
449.         changeBOARD()
450.         print("Qlearn Win rate: " + str(WIN / iters
            * 100) + "%")
451.         print("Qlearn Lose rate: " + str(LOSE / iter
            s * 100) + "%")
452.         print("Qlearn Draw rate: " + str(DRAW / iter
            s * 100) + "%")
453.         reset()
454.
455. def qlearnVSbaseline(first_player, agent):
456.     if first_player == HUMAN:
457.         baseline(HUMAN)
458.     prev_board = toString(board)
459.     prev_action = agent.get_action(prev_board)
460.
461.     while True:
462.         # agent move
463.         set_move(board, prev_action, COMP)
464.         if checkWin(board, COMP):
465.             changeWIN()
466.             reward = 1
467.             break
468.         elif len(getValidColumns(board)) == 0:
469.             changeDRAW()
470.             reward = 0
```

```

471.             break
472.         # teacher move
473.         baseline(HUMAN)
474.         if checkWin(board, HUMAN):
475.             reward = -1
476.             changeLOSE()
477.             break
478.         elif len(getValidColumns(board)) == 0:
479.             reward = 0
480.             changeDRAW()
481.             break
482.         else:
483.             reward = 0
484.             new_board = toString(board)
485.             new_action = agent.get_action(new_board)
486.             agent.update(prev_board, new_board, prev
                _action, new_action, reward)
487.             prev_board = new_board
488.             prev_action = new_action
489.
490.             agent.update(prev_board, None, prev_action,
                None, reward)
491.
492.
493. def runQlearnVSBaseline(agent, iters):
494.     while GAME_COUNT < iters:
495.         if random.random() < 0.5:
496.             qlearnVsbaseline(HUMAN, agent)
497.         else:
498.             qlearnVsbaseline(COMP, agent)
499.         changeGAME_COUNT()
500.         changeBOARD()
501.         print("Qlearn Win rate: " + str(WIN / iters
            * 100) + "%")
502.         print("Qlearn Lose rate: " + str(LOSE / iter
            s * 100) + "%")
503.         print("Qlearn Draw rate: " + str(DRAW / iter
            s * 100) + "%")
504.
505.
506. def toString(board):
507.     ans = ''
508.     for row in board:

```

```
509.         for col in row:
510.             ans += str(col)
511.     return ans
```

```
1. import os
2. import pickle
3. from Agent import QLearner
4. from Game import train, runQlearnVSMinimax, runQlearnVSBaseline, runMinimaxVSBaseline
5.
6.
7. class PlayGame:
8.     def __init__(self, alpha=0.5, gamma=0.9, epsilon=0.1):
9.         self.alpha = alpha
10.        self.gamma = gamma
11.        self.epsilon = epsilon
12.        self.qtable = {}
13.        if os.path.isfile('q_agent.pkl'):
14.            with open('q_agent.pkl', 'rb') as f:
15.                self.agent = pickle.load(f)
16.        else:
17.            self.agent = QLearner(alpha, gamma, epsilon)
18.        self.games_played = 0
19.
20.    def teach(self, iters):
21.        train(self.agent, iters)
22.
23.    def playQlearnVSMinimax(self, iters):
24.        print('Qlearn VS Minimax...')
25.        runQlearnVSMinimax(self.agent, iters)
26.
27.    def playQlearnVSBaseline(self, iters):
28.        print('Qlearn VS Baseline...')
29.        runQlearnVSBaseline(self.agent, iters)
30.
31.    def playMinimaxVSBaseline(self, iters):
32.        print('Minimax VS Baseline...')
33.        runMinimaxVSBaseline(iters)
34.
35.
36. if __name__ == '__main__':
```

```

37.     game_thread = PlayGame()
38.     game_thread.teach(5000)
39.     print("q-learn VS baseline 20 times")
40.     game_thread.playQlearnVSMinimax(20)
41.     print("")
42.
43.     print("q-learn VS baseline 50 times")
44.     game_thread.playQlearnVSMinimax(50)
45.     print("")
46.
47.     print("q-learn VS baseline 100 times")
48.     game_thread.playQlearnVSMinimax(100)
49.     print("")

```

```

1. import random
2. import numpy as np
3.
4. def getValidColumns(board):
5.     columns = []
6.     for i in range(7):
7.         if board[0][i] == 0:
8.             columns.append(i)
9.     return columns
10.
11.
12. def set_move(state, column, player):
13.     for i in range(6):
14.         if state[i][column] != 0:
15.             state[i - 1][column] = player
16.             break
17.     elif i == 5:
18.         state[i][column] = player
19.
20.
21. def checkWin(state, player):
22.     for i in range(6):
23.         for j in range(4):
24.             if state[i][j] == state[i][j + 1] == state[i][j + 2] == state[i][j + 3] == player:
25.                 return True
26.
27.     for i in range(7):
28.         for j in range(3):

```

```

29.         if state[j][i] == state[j + 1][i] == state[j + 2][i] == state[j + 3][i] == player:
30.             return True
31.
32.     for i in range(3):
33.         for j in range(4):
34.             if state[i][j] == state[i + 1][j + 1] == state[i + 2][j + 2] == state[i + 3][j + 3] == player:
35.                 return True
36.
37.     for i in range(3):
38.         for j in range(3, 7):
39.             if state[i][j] == state[i + 1][j - 1] == state[i + 2][j - 2] == state[i + 3][j - 3] == player:
40.                 return True
41.     return False
42.
43.
44. class Teacher:
45.     def set_win(self, board, player=1):
46.         columns = getValidColumns(board)
47.         for column in columns:
48.             new_board = np.copy(board)
49.             set_move(new_board, column, player)
50.             if checkWin(board, player):
51.                 return column
52.
53.         return None
54.
55.     def set_blockOpponentWin(self, board):
56.         return self.set_win(board, 2)
57.
58.     def set_random(self, board):
59.         while True:
60.             col = random.randint(0, 6)
61.             if board[0][col] == 0:
62.                 return col
63.
64.     def move(self, board):
65.         if self.set_win(board):
66.             return self.set_win(board)
67.         elif self.set_blockOpponentWin(board):

```

```
68.         return self.set_blockOpponentWin(board)
69.     else:
70.         return self.set_random(board)
```