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.

```python
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.

```python
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 ε value in the greedy algorithm, and eps_decay is the decay factor of the ε 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.

```python
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.

```python
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.

```python
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:...
        elif i == 6:...

    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.

```python
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.

```python
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.

```python
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.

```python
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 ε value in the greedy algorithm, and eps_decay is the decay factor of the ε 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.

```python
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.

```python
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.

```python
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.

```python
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

```
import random
from math import inf as infinity
from random import choice

HUMAN = -1
COMP = +1
board = [
    [0, 0, 0],
    [0, 0, 0],
    [0, 0, 0],
]
WIN = 0
LOSE = 0
DRAW = 0


def evaluate(state):
    if wins(state, COMP):
        score = +1
    elif wins(state, HUMAN):
        score = -1
    else:
        score = 0

    return score


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


def set_move(x, y, player):
    if valid_move(x, y):
        board[x][y] = player
        return True
    else:
        return False


def minimax(state, depth, player):
    if player == COMP:
        best = [-1, -1, -infinity]
    else:
        best = [-1, -1, +infinity]

    if depth == 0 or game_over(state):
        score = evaluate(state)
        return [-1, -1, score]

```

```
    for cell in empty_cells(state):
        x, y = cell[0], cell[1]
        state[x][y] = player
        score = minimax(state, depth - 1, -player)

        state[x][y] = 0
        score[0], score[1] = x, y

        if player == COMP:
            if score[2] > best[2]:
                best = score
        else:
            if score[2] < best[2]:
                best = score

    return best


def ai_turn(player):
    depth = len(empty_cells(board))
    if depth == 0 or game_over(board):
        return
    if depth == 9:
        x = choice([0, 1, 2])
        y = choice([0, 1, 2])
    else:
        move = minimax(board, depth, player)
        x, y = move[0], move[1]
    set_move(x, y, player)


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

    cells = [
        [board[0][0], board[0][1], board[0][2]],

        [board[1][0], board[1][1], board[1][2]],

        [board[2][0], board[2][1], board[2][2]],
```

```
            [board[0][0], board[1][0], board[2][0]],
            [board[0][1], board[1][1], board[2][1]],
            [board[0][2], board[1][2], board[2][2]],
            [board[0][0], board[1][1], board[2][2]],
            [board[2][0], board[1][1], board[0][2]],
    ]
    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:
            if cells[i][0] == cells[i][1] and cells[i][0] == player and cells[i][2] == 0:
                x = 2
                y = i - 3
                break
            elif cells[i][0] == cells[i][2] and cells[i][0] == player and cells[i][1] == 0:
                x = 1
                y = i - 3
                break
            elif cells[i][1] == cells[i][2] and cells[i][1] == player and cells[i][0] == 0:
                x = 0
```

```
                y = i - 3
                break
        elif i == 6:
            if cells[i][0] == cells[i][1] and cells[i][0] == player and cells[i][2] == 0:
                x = 2
                y = 2
                break
            elif cells[i][0] == cells[i][2] and cells[i][0] == player and cells[i][1] == 0:
                x = 1
                y = 1
                break
            elif cells[i][1] == cells[i][2] and cells[i][1] == player and cells[i][0] == 0:
                x = 0
                y = 0
                break

    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)


def changeWIN():
    global WIN
    WIN += 1


def changeLOSE():
    global LOSE
    LOSE += 1


def changeDRAW():
    global DRAW
    DRAW += 1


def changeBOARD():
```

```
    global board
    board = [
        [0, 0, 0],
        [0, 0, 0],
        [0, 0, 0]
    ]


def reset():
    global WIN, LOSE, DRAW, board
    WIN, LOSE, DRAW = 0, 0, 0


def gameplay():
    while wins(board, HUMAN) == False and wins(board, COMP) == False and len(empty_cells(board)) > 0:
        ai_turn(HUMAN)
        baseline(COMP)
        if wins(board, HUMAN):
            changeWIN()
            break
        elif wins(board, COMP):
            changeLOSE()
            break
        elif len(empty_cells(board)) == 0:
            changeDRAW()
            break


def minimaxVSbaseline(first_player):
    if first_player == HUMAN:
        baseline(HUMAN)
    while True:
        # agent move
        ai_turn(COMP)
        if wins(board, COMP):
            changeWIN()
            break
        elif len(empty_cells(board)) == 0:
            changeDRAW()
            break
        # teacher move
        baseline(HUMAN)
```

```
        if wins(board, HUMAN):
            changeLOSE()
            break
        elif len(empty_cells(board)) == 0:
            changeDRAW()
            break


def runMinimaxVSBaseline(iters):
    for i in range(iters):
        if random.random() < 0.5:
            minimaxVSbaseline(HUMAN)
        else:
            minimaxVSbaseline(COMP)
        changeBOARD()
    print("Minimax Win rate: " + str(WIN / iters * 100) + "%")
    print("Minimax Lose rate: " + str(LOSE / iters * 100) + "%")
    print("Minimax Draw rate: " + str(DRAW / iters * 100) + "%")


if __name__ == '__main__':
    print("Minimax VS Baseline 20 times")
    runMinimaxVSBaseline(20)
    print("")
    reset()

    print("Minimax VS Baseline 50 times")
    runMinimaxVSBaseline(50)
    print("")
    reset()

    print("Minimax VS Baseline 100 times")
    runMinimaxVSBaseline(100)
    print("")
    reset()
```

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

```
import os
import pickle
import collections
import numpy as np
```

```python
import random


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 = []

    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)
```

```
from random import choice
import numpy as np
from Teacher import Teacher
HUMAN = 1
COMP = 2
board = [
    [0, 0, 0],
    [0, 0, 0],
    [0, 0, 0],
]
WIN = 0
LOSE = 0
DRAW = 0
GAME_COUNT = 0

WIN_SCORE = 1000000000
LOSE_SCORE = -1000000000000000000000000

# scoreing constants
WINDOW_LENGTH = 4

FOURINROW = 10000
```

```
THREEINROW = 10
TWOINROW = 3
MIDDLE_COLUMN = 2

AGING_PENALTY = 3

OPP_FOURINROW = -1000000000000000000000000
OPP_THREEINROW = -12
OPP_TWOINROW = -4
GAME_COUNT = 0

def evaluate(state):
    if wins(state, COMP):
        score = +1
    elif wins(state, HUMAN):
        score = -1
    else:
        score = 0

    return score


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)


```

```python
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


def set_move(x, y, player):
    if valid_move(x, y):
        board[x][y] = player
        return True
    else:
        return False


def minimax(state, depth, player):
    if player == COMP:
        best = [-1, -1, -infinity]
    else:
        best = [-1, -1, +infinity]

    if depth == 0 or game_over(state):
        score = evaluate(state)
        return [-1, -1, score]

    for cell in empty_cells(state):
        x, y = cell[0], cell[1]
        state[x][y] = player
        score = minimax(state, depth - 1, -player)
        state[x][y] = 0
        score[0], score[1] = x, y

```

```
        if player == COMP:
            if score[2] > best[2]:
                best = score
        else:
            if score[2] < best[2]:
                best = score

    return best


def ai_turn(player):
    depth = len(empty_cells(board))
    if depth == 0 or game_over(board):
        return
    if depth == 9:
        x = choice([0, 1, 2])
        y = choice([0, 1, 2])
    else:
        move = minimax(board, depth, player)
        x, y = move[0], move[1]
    set_move(x, y, player)


def is_terminal(state):
    return wins(state, COMP) or wins(state, HUMAN) or len(empty_cells(state)) == 0


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))
```

```
    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


def score_position(state, player):
    score = 0
    # score center column
```

```
    center_array = [state[i][3] for i in range(6)]
    # for i in range(6):
    #     center_array.append(state[i][3])
    center_count = center_array.count(player)
    score += center_count * MIDDLE_COLUMN

    # score horizontal
    for r in range(6):
        row_array = [state[r][i] for i in range(7)]
        for c in range(4):
            window = row_array[c:c + WINDOW_LENGTH]
            score += evaluate_window(window, player)

    # score vertical
    for c in range(7):
        col_array = [state[i][c] for i in range(6)]
        for r in range(3):
            window = col_array[r:r + WINDOW_LENGTH]
            score += evaluate_window(window, player)

    # score positive sloped diagonal
    for r in range(3):
        for c in range(4):
            window = [state[r + i][c + i] for i in range(4)]
            score += evaluate_window(window, player)

    # score negative sloped diagonal
    for r in range(3):
        for c in range(4):
            window = [state[r + 3 - i][c + i] for i in range(4)]
            score += evaluate_window(window, player)

    return score
```

```


def evaluate_window(window, player):
    score = 0
    opp_piece = HUMAN
    if player == HUMAN:
        opp_piece = COMP

    if window.count(player) == 4:
        score += FOURINROW
        return score
    elif window.count(player) == 3 and window.count(0) == 1:
        score += THREEINROW
    elif window.count(player) == 2 and window.count(0) == 2:
        score += TWOINROW

    if window.count(opp_piece) == 4:
        score -= FOURINROW
        return score
    elif window.count(opp_piece) == 3 and window.count(0) == 1:
        score -= THREEINROW
    elif window.count(opp_piece) == 2 and window.count(0) == 2:
        score -= TWOINROW
    return score



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

    cells = [
            [board[0][0], board[0][1], board[0][2]],
            [board[1][0], board[1][1], board[1][2]],
            [board[2][0], board[2][1], board[2][2]],
```

```
        [board[0][0], board[1][0], board[2][0]],
        [board[0][1], board[1][1], board[2][1]],
        [board[0][2], board[1][2], board[2][2]],
        [board[0][0], board[1][1], board[2][2]],
        [board[2][0], board[1][1], board[0][2]],
    ]
    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:
            if cells[i][0] == cells[i][1] and cells[i][0] == player and cells[i][2] == 0:
                x = 2
                y = i - 3
                break
            elif cells[i][0] == cells[i][2] and cells[i][0] == player and cells[i][1] == 0:
                x = 1
                y = i - 3
                break
            elif cells[i][1] == cells[i][2] and cells[i][1] == player and cells[i][0] == 0:
                x = 0
```

```
                y = i - 3
                break
        elif i == 6:
            if cells[i][0] == cells[i][1] and cells[i][0] == player and cells[i][2] == 0:
                x = 2
                y = 2
                break
            elif cells[i][0] == cells[i][2] and cells[i][0] == player and cells[i][1] == 0:
                x = 1
                y = 1
                break
            elif cells[i][1] == cells[i][2] and cells[i][1] == player and cells[i][0] == 0:
                x = 0
                y = 0
                break

    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)


def changeWIN():
    global WIN
    WIN += 1


def changeLOSE():
    global LOSE
    LOSE += 1


def changeDRAW():
    global DRAW
    DRAW += 1


def changeGAME_COUNT():
```

```
    global GAME_COUNT
    GAME_COUNT += 1


def changeBoard():
    global board
    board = [
        [0, 0, 0],
        [0, 0, 0],
        [0, 0, 0]
    ]


def reset():
    global WIN, LOSE, DRAW, board
    WIN, LOSE, DRAW = 0, 0, 0


def gameplay():
    while wins(board, HUMAN) == False and wins(board, COMP) == False and len(empty_cells(board)) > 0:
        ai_turn(HUMAN)
        baseline(COMP)
        if wins(board, HUMAN):
            changeWIN()
            break
        elif wins(board, COMP):
            changeLOSE()
            break
        elif len(empty_cells(board)) == 0:
            changeDRAW()
            break


def trainGamePlay(first_player, teacher, agent):

    if first_player == HUMAN:
        action = teacher.move(board)
        set_move(action[0], action[1], HUMAN)

    prev_board = toString(board)
    prev_action = agent.get_action(prev_board)

```

```
    while True:
        # agent move
        set_move(prev_action[0], prev_action[1], COMP)
        if wins(board, COMP):
            reward = 1
            break
        elif len(empty_cells(board)) == 0:
            reward = 0
            break
        # teacher move
        action = teacher.move(board)
        set_move(action[0], action[1], HUMAN)
        if wins(board, HUMAN):
            reward = -1
            break
        elif len(empty_cells(board)) == 0:
            reward = 0
            break
        else:
            reward = 0
        new_board = toString(board)
        new_action = agent.get_action(new_board)

        agent.update(prev_board, new_board, prev_action, new_action, reward)
        prev_board = new_board
        prev_action = new_action

    agent.update(prev_board, None, prev_action, None, reward)


def teacherPlay(agent):
    teacher = Teacher()
    if random.random() < 0.5:
        trainGamePlay(HUMAN, teacher, agent)
    else:
        trainGamePlay(COMP, teacher, agent)


def train(agent, iters):
    while GAME_COUNT < iters:
        teacherPlay(agent)
```

```
        changeGAME_COUNT()
        if GAME_COUNT % 1000 == 0:
            print("Games played: %i" % GAME_COUNT)
        changeBoard()
    agent.save('q_agent.pkl')


def toString(board):
    ans = ''
    for row in board:
        for col in row:
            ans += str(col)
    return ans

# qlearn is COMP
def qlearnVSminimax(first_player, agent):
    if first_player == HUMAN:
        ai_turn(HUMAN)
    prev_board = toString(board)
    prev_action = agent.get_action(prev_board)

    while True:
        # agent move
        set_move(prev_action[0], prev_action[1], COMP)
        if wins(board, COMP):
            changeWIN()
            reward = 1
            break
        elif len(empty_cells(board)) == 0:
            changeDRAW()
            reward = 0
            break
        # teacher move
        ai_turn(HUMAN)
        if wins(board, HUMAN):
            reward = -1
            changeLOSE()
            break
        elif len(empty_cells(board)) == 0:
            reward = 0
            changeDRAW()
            break
```

```python
        else:
            reward = 0
        new_board = toString(board)
        new_action = agent.get_action(new_board)

        agent.update(prev_board, new_board, prev_action, new_action, reward)
        prev_board = new_board
        prev_action = new_action

    agent.update(prev_board, None, prev_action, None, reward)


def runQlearnVSMinimax(agent, iters):
    while GAME_COUNT < iters:
        if random.random() < 0.5:
            qlearnVSminimax(HUMAN, agent)
        else:
            qlearnVSminimax(COMP, agent)
        changeGAME_COUNT()
        changeBoard()
    print("Qlearn Win rate: " + str(WIN / iters * 100) + "%")
    print("Qlearn Lose rate: " + str(LOSE / iters * 100) + "%")
    print("Qlearn Draw rate: " + str(DRAW / iters * 100) + "%")


def qlearnVSbaseline(first_player, agent):
    if first_player == HUMAN:
        baseline(HUMAN)
    prev_board = toString(board)
    prev_action = agent.get_action(prev_board)

    while True:
        # agent move
        set_move(prev_action[0], prev_action[1], COMP)
        if wins(board, COMP):
            changeWIN()
            reward = 1
            break
```

```
        elif len(empty_cells(board)) == 0:
            changeDRAW()
            reward = 0
            break
        # teacher move
        baseline(HUMAN)
        if wins(board, HUMAN):
            reward = -1
            changeLOSE()
            break
        elif len(empty_cells(board)) == 0:
            reward = 0
            changeDRAW()
            break
        else:
            reward = 0
        new_board = toString(board)
         new_action = agent.get_action(new_board)

         agent.update(prev_board, new_board, prev_action, new_action, reward)
        prev_board = new_board
        prev_action = new_action

    agent.update(prev_board, None, prev_action, None, reward)


def runQlearnVSBaseline(agent, iters):
    while GAME_COUNT < iters:
        if random.random() < 0.5:
            qlearnVSbaseline(HUMAN, agent)
        else:
            qlearnVSbaseline(COMP, agent)
        changeGAME_COUNT()
        changeBoard()
    print("Qlearn Win rate: " + str(WIN / iters * 100) + "%")
    print("Qlearn Lose rate: " + str(LOSE / iters * 100) + "%")
    print("Qlearn Draw rate: " + str(DRAW / iters * 100) + "%")
```

```python
import os
import pickle
from Agent import QLearner
from Game import train, runQlearnVSMinimax, runQlearnVSBaseline


class PlayGame:
    def __init__(self, alpha=0.5, gamma=0.9, epsilon=0.1):
        self.alpha = alpha
        self.gamma = gamma
        self.epsilon = epsilon
        self.qtable = {}
        if os.path.isfile('q_agent.pkl'):
            with open('q_agent.pkl', 'rb') as f:
                self.agent = pickle.load(f)
        else:
            self.agent = QLearner(alpha, gamma, epsilon)
        self.games_played = 0

    def teach(self, iters):
        train(self.agent, iters)

    def playQlearnVSMinimax(self, iters):
        # print('Qlearn VS Minimax...')
        runQlearnVSMinimax(self.agent, iters)

    def playQlearnVSBaseline(self, iters):
        # print('Qlearn VS Baseline...')
        runQlearnVSBaseline(self.agent, iters)


if __name__ == '__main__':
    game_thread = PlayGame()
    # game_thread.teach(500000)

    print("q-learn VS minimax 20 times")
    game_thread.playQlearnVSMinimax(20)
    print("")

    print("q-learn VS minimax 50 times")
    game_thread.playQlearnVSMinimax(50)
```

```
    print("")

    print("q-learn VS minimax 100 times")
    game_thread.playQlearnVSMinimax(100)
    print("")
```

```
import random
import Game


class Teacher:
     def set_win(self, board, player=1):
         cells = Game.empty_cells(board)
         for cell in cells:
             if board[cell[0]][cell[1]] == 0:
                 board[cell[0]][cell[1]] = player
                 if Game.wins(board, player):
                     board[cell[0]][cell[1]] = 0
                     return cell[0], cell[1]
                 else:
                     board[cell[0]][cell[1]] = 0

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

     def set_twoThreatToWin(self, board):
         if board[1][0] == 1 and board[0][1] == 1:
             if board[0][0] == 0 and board[2][0] == 0 and board[0][2] == 0:
                 return 0, 0
             elif board[1][1] == 0 and board[2][1] == 0 and board[1][2] == 0:
                 return 1, 1
         elif board[1][0] == 1 and board[2][1] == 1:
             if board[2][0] == 0 and board[0][0] == 0 and board[2][2] == 0:
                 return 2, 0
             elif board[1][1] == 0 and board[0][1] == 0 and board[1][2] == 0:
                 return 1, 1
         elif board[2][1] == 1 and board[1][2] == 1:
```

```
            if board[2][2] == 0 and board[2][0] == 0 and board[0][2] == 0:
                return 2, 2
            elif board[1][1] == 0 and board[1][0] == 0 and board[0][1] == 0:
                return 1, 1
        elif board[1][2] == 1 and board[0][1] == 1:
            if board[0][2] == 0 and board[0][0] == 0 and board[2][2] == 0:
                return 0, 2
            elif board[1][1] == 0 and board[1][0] == 0 and board[2][1] == 0:
                return 1, 1

        elif board[0][0] == 1 and board[2][2] == 1:
            if board[1][0] == 0 and board[2][1] == 0 and board[2][0] == 0:
                return 2, 0
            elif board[0][1] == 0 and board[1][2] == 0 and board[0][2] == 0:
                return 0, 2
        elif board[2][0] == 1 and board[0][2] == 1:
            if board[2][1] == 0 and board[1][2] == 0 and board[2][2] == 0:
                return 2, 2
            elif board[1][0] == 0 and board[0][1] == 0 and board[0][0] == 0:
                return 0, 0
        return None

    def set_blockOpponentTwoThreatWin(self, board):
        corners = [board[0][0], board[2][0], board[0][2], board[2][2]]
        if board[1][0] == 2 and board[0][1] == 2:
            if board[0][0] == 0 and board[2][0] == 0 and board[0][2] == 0:
                return 0, 0
            elif board[1][1] == 0 and board[2][1] == 0 and board[1][2] == 0:
                return 1, 1
```

```
        elif board[1][0] == 2 and board[2][1] == 2:
            if board[2][0] == 0 and board[0][0] == 0 and board[2][2] == 0:
                return 2, 0
            elif board[1][1] == 0 and board[0][1] == 0 and board[1][2] == 0:
                return 1, 1
        elif board[2][1] == 2 and board[1][2] == 2:
            if board[2][2] == 0 and board[2][0] == 0 and board[0][2] == 0:
                return 2, 2
            elif board[1][1] == 0 and board[1][0] == 0 and board[0][1] == 0:
                return 1, 1
        elif board[1][2] == 2 and board[0][1] == 2:
            if board[0][2] == 0 and board[0][0] == 0 and board[2][2] == 0:
                return 0, 2
            elif board[1][1] == 0 and board[1][0] == 0 and board[2][1] == 0:
                return 1, 1
        # if we have two corners, try to set the center
        elif corners.count(0) == 1 and corners.count(2) == 2:
            return 1, 2
        elif board[0][0] == 2 and board[2][2] == 2:
            if board[1][0] == 0 and board[2][1] == 0 and board[2][0] == 0:
                return 2, 0
            elif board[0][1] == 0 and board[2][1] == 0 and board[0][2] == 0:
                return 0, 2
        elif board[2][0] == 2 and board[0][2] == 2:
            if board[2][1] == 0 and board[1][2] == 0 and board[2][2] == 0:
                return 2, 2
            elif board[1][0] == 0 and board[0][1] == 0 and board[0][0] == 0:
```

```
                return 0, 0
        return None

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

    def set_corner(self, board):
        # pick opposite corner
        if board[0][0] == 2 and board[2][2] == 0:
            return 2, 2
        elif board[2][2] == 2 and board[0][0] == 0:
            return 0, 0
        elif board[0][2] == 2 and board[2][0] == 0:
            return 2, 0
        elif board[2][0] == 2 and board[0][2] == 0:
            return 0, 2

        if board[0][0] == 0:
            return 0, 0
        elif board[2][0] == 0:
            return 2, 0
        elif board[0][2] == 0:
            return 0, 2
        elif board[2][2] == 0:
            return 2, 2
        return None

    def set_other(self, board):
        if board[1][0] == 0:
            return 1, 0
        elif board[2][1] == 0:
            return 2, 1
        elif board[1][2] == 0:
            return 1, 2
        elif board[0][1] == 0:
            return 0, 1
        return None

    def set_random(self, board):
```

```python
        while True:
            x = random.randint(0, 2)
            y = random.randint(0, 2)
            if board[x][y] == 0:
                return x, y

    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)
```

3. Connect 4

```python
import os
import pickle
import collections
import numpy as np
import random


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):
```

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

    def get_action(self, s):
        possible_actions = [a for a in self.actions if s[a] == '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':
                    possible_actions.append(action)

            for action in possible_actions:
                Qs.append(self.Q[action][s_])
```

```python
                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)
```

```python
import math
import random
import numpy as np
import time
from Teacher import Teacher

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

WIN_SCORE = 1000000000
LOSE_SCORE = -1000000000000000000000000

# scoreing constants
WINDOW_LENGTH = 4

FOURINROW = 10000
THREEINROW = 10
TWOINROW = 3
MIDDLE_COLUMN = 2

AGING_PENALTY = 3

OPP_FOURINROW = -1000000000000000000000000
OPP_THREEINROW = -12
OPP_TWOINROW = -4
GAME_COUNT = 0


```

```
def set_move(state, column, player):
    for i in range(6):
        if state[i][column] != 0:
            state[i - 1][column] = player
            break
        elif i == 5:
            state[i][column] = player


def checkWin(state, player):
    for i in range(6):
        for j in range(4):
            if state[i][j] == state[i][j + 1] == state[i][j + 2] == state[i][j + 3] == player:
                return True

    for i in range(7):
        for j in range(3):
            if state[j][i] == state[j + 1][i] == state[j + 2][i] == state[j + 3][i] == player:
                return True

    for i in range(3):
        for j in range(4):
            if state[i][j] == state[i + 1][j + 1] == state[i + 2][j + 2] == state[i + 3][j + 3] == player:
                return True

    for i in range(3):
        for j in range(3, 7):
            if state[i][j] == state[i + 1][j - 1] == state[i + 2][j - 2] == state[i + 3][j - 3] == player:
                return True
    return False


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


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))

    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


def score_position(state, player):
    score = 0
    # score center column
    center_array = [state[i][3] for i in range(6)]
    # for i in range(6):
    #     center_array.append(state[i][3])
    center_count = center_array.count(player)
    score += center_count * MIDDLE_COLUMN

    # score horizontal
    for r in range(6):
        row_array = [state[r][i] for i in range(7)]
        for c in range(4):
            window = row_array[c:c + WINDOW_LENGTH]
            score += evaluate_window(window, player)

    # score vertical
    for c in range(7):
```

```python
        col_array = [state[i][c] for i in range(6)]
        for r in range(3):
            window = col_array[r:r + WINDOW_LENGTH]
            score += evaluate_window(window, player)

    # score positive sloped diagonal
    for r in range(3):
        for c in range(4):
            window = [state[r + i][c + i] for i in range(4)]
            score += evaluate_window(window, player)

    # score negative sloped diagonal
    for r in range(3):
        for c in range(4):
            window = [state[r + 3 - i][c + i] for i in range(4)]
            score += evaluate_window(window, player)

    return score


def evaluate_window(window, player):
    score = 0
    opp_piece = HUMAN
    if player == HUMAN:
        opp_piece = COMP

    if window.count(player) == 4:
        score += FOURINROW
        return score
    elif window.count(player) == 3 and window.count(0) == 1:
        score += THREEINROW
    elif window.count(player) == 2 and window.count(0) == 2:
        score += TWOINROW

    if window.count(opp_piece) == 4:
```

```
        score -= FOURINROW
        return score
    elif window.count(opp_piece) == 3 and window.count(0) == 1:
        score -= THREEINROW
    elif window.count(opp_piece) == 2 and window.count(0) == 2:
        score -= TWOINROW
    return score


def changeWIN():
    global WIN
    WIN += 1


def changeLOSE():
    global LOSE
    LOSE += 1


def changeDRAW():
    global DRAW
    DRAW += 1


def changeBOARD():
    global board
    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]]


def changeGAME_COUNT():
    global GAME_COUNT
    GAME_COUNT += 1


def reset():
    global WIN, LOSE, DRAW, GAME_COUNT
    WIN = 0
```

```
    LOSE = 0
    DRAW = 0
    GAME_COUNT = 0


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)


def testBaseline():
    itr = 0
    while True:
        itr += 1
        baseline(HUMAN)
        if is_terminal(board):
            break
        baseline(COMP)
        if is_terminal(board):
            break


def trainGamePlay(first_player, teacher, agent):
    depth = 6
    round = 0
    if first_player == HUMAN:
        # action = teacher.move(board)
```

```
            action = minimax(board, 6, -math.inf, math.inf, True)[0]
            set_move(board, action, HUMAN)

        prev_board = toString(board)
        prev_action = agent.get_action(prev_board)

        while True:
            # agent move
            set_move(board, prev_action, COMP)
            if checkWin(board, COMP):
                reward = 1
                break
            elif len(getValidColumns(board)) == 0:
                reward = 0
                break
            # teacher move
            # action = teacher.move(board)
            # set_move(board, action, HUMAN)
            start_time = time.time()
            ai_col = minimax(board, depth - 1, -math.inf, math.inf, True)[0]
            end_time = time.time()
            set_move(board, ai_col, HUMAN)
            run_time = end_time - start_time
            print("Round " + str(round) + ": Time taken: " + str(run_time) + "s")
            print("")
            if checkWin(board, HUMAN):
                reward = -1
                break
            elif len(getValidColumns(board)) == 0:
                reward = 0
                break
            else:
                reward = 0
            new_board = toString(board)
            new_action = agent.get_action(new_board)

            agent.update(prev_board, new_board, prev_action, new_action, reward)
            prev_board = new_board
            prev_action = new_action
            if run_time > 1 and round > 3:
```

```
            depth -= 1
        round += 1
    agent.update(prev_board, None, prev_action, None, reward)


def teacherPlay(agent):
    teacher = Teacher()
    if random.random() < 0.5:
        trainGamePlay(HUMAN, teacher, agent)
    else:
        trainGamePlay(COMP, teacher, agent)


def train(agent, iters):
    while GAME_COUNT < iters:
        teacherPlay(agent)
        changeGAME_COUNT()
        if GAME_COUNT % 1000 == 0:
            print("Games played: %i" % GAME_COUNT)
        changeBOARD()
    agent.save('q_agent.pkl')


# minimax as COMP and baseline as HUMAN
def minimaxVSbaseline(first_player):
    depth = 6
    round = 0
    if first_player == HUMAN:
        baseline(HUMAN)
    while True:
        # agent move
        start_time = time.time()
        ai_col = minimax(board, depth - 1, -math.inf, math.inf, True)[0]
        end_time = time.time()
        set_move(board, ai_col, COMP)
        run_time = end_time - start_time
        print("Round " + str(round) + ": Time taken: " + str(run_time) + "s")
        print("")
        if checkWin(board, COMP):
            changeWIN()
```

```
            break
        elif len(getValidColumns(board)) == 0:
            changeDRAW()
            break
        # teacher move
        baseline(HUMAN)
        if checkWin(board, HUMAN):
            changeLOSE()
            break
        elif len(getValidColumns(board)) == 0:
            changeDRAW()
            break

        # if run_time < 7.5 and round > 4:
        #     depth += 1
        # elif run_time > 12.5 and round > 4:
        #     depth -=
        if run_time > 3 and round > 4:
            depth -= 1
        round += 1


def runMinimaxVSBaseline(iters):
    for i in range(iters):
        if random.random() < 0.5:
            minimaxVSbaseline(HUMAN)
        else:
            minimaxVSbaseline(COMP)
        changeBOARD()
    print("Minimax Win rate: " + str(WIN / iters * 100) + "%")
    print("Minimax Lose rate: " + str(LOSE / iters * 100) + "%")
    print("Minimax Draw rate: " + str(DRAW / iters * 100) + "%")
    reset()


# qlearn as COMP and minimax as HUMAN
def qlearnVSminimax(first_player, agent):
    depth = 6
    round = 0
    if first_player == HUMAN:
```

```
            ai_col = minimax(board, depth - 1, -math.inf, math.inf, True)[0]
            set_move(board, ai_col, HUMAN)
        prev_board = toString(board)
        prev_action = agent.get_action(prev_board)
        while True:
            # agent move
            set_move(board, prev_action, COMP)
            if checkWin(board, COMP):
                changeWIN()
                reward = 1
                break
            elif len(getValidColumns(board)) == 0:
                changeDRAW()
                reward = 0
                break
            # teacher move
            start_time = time.time()
            ai_col = minimax(board, depth - 1, -math.inf, math.inf, True)[0]
            end_time = time.time()
            set_move(board, ai_col, HUMAN)
            run_time = end_time - start_time
            print("Round " + str(round) + ": Time taken: " + str(run_time) + "s")
            print("")
            if checkWin(board, HUMAN):
                reward = -1
                changeLOSE()
                break
            elif len(getValidColumns(board)) == 0:
                reward = 0
                changeDRAW()
                break
            else:
                reward = 0

            # if run_time < 7.5 and round > 4:
            #     depth += 1
            # elif run_time > 12.5 and round > 4:
            #     depth -= 1
            if run_time > 3 and round > 4:
                depth -= 1
            round += 1
```

```python

        new_board = toString(board)
        new_action = agent.get_action(new_board)

        agent.update(prev_board, new_board, prev_action, new_action, reward)
        prev_board = new_board
        prev_action = new_action

    agent.update(prev_board, None, prev_action, None, reward)


def runQlearnVSMinimax(agent, iters):
    for i in range(iters):
        if random.random() < 0.5:
            qlearnVSminimax(HUMAN, agent)
        else:
            qlearnVSminimax(COMP, agent)
        changeBOARD()
    print("Qlearn Win rate: " + str(WIN / iters * 100) + "%")
    print("Qlearn Lose rate: " + str(LOSE / iters * 100) + "%")
    print("Qlearn Draw rate: " + str(DRAW / iters * 100) + "%")
    reset()

def qlearnVSbaseline(first_player, agent):
    if first_player == HUMAN:
        baseline(HUMAN)
    prev_board = toString(board)
    prev_action = agent.get_action(prev_board)

    while True:
        # agent move
        set_move(board, prev_action, COMP)
        if checkWin(board, COMP):
            changeWIN()
            reward = 1
            break
        elif len(getValidColumns(board)) == 0:
            changeDRAW()
            reward = 0
```

```
                break
            # teacher move
            baseline(HUMAN)
            if checkWin(board, HUMAN):
                reward = -1
                changeLOSE()
                break
            elif len(getValidColumns(board)) == 0:
                reward = 0
                changeDRAW()
                break
            else:
                reward = 0
            new_board = toString(board)
            new_action = agent.get_action(new_board)
            agent.update(prev_board, new_board, prev_action, new_action, reward)
            prev_board = new_board
            prev_action = new_action

        agent.update(prev_board, None, prev_action, None, reward)


def runQlearnVSBaseline(agent, iters):
    while GAME_COUNT < iters:
        if random.random() < 0.5:
            qlearnVSbaseline(HUMAN, agent)
        else:
            qlearnVSbaseline(COMP, agent)
        changeGAME_COUNT()
        changeBOARD()
    print("Qlearn Win rate: " + str(WIN / iters * 100) + "%")
    print("Qlearn Lose rate: " + str(LOSE / iters * 100) + "%")
    print("Qlearn Draw rate: " + str(DRAW / iters * 100) + "%")


def toString(board):
    ans = ''
    for row in board:
```

```
        for col in row:
            ans += str(col)
    return ans
```

```
import os
import pickle
from Agent import QLearner
from Game import train, runQlearnVSMinimax, runQlearnVSBaseline, runMinimaxVSBaseline


class PlayGame:
    def __init__(self, alpha=0.5, gamma=0.9, epsilon=0.1):
        self.alpha = alpha
        self.gamma = gamma
        self.epsilon = epsilon
        self.qtable = {}
        if os.path.isfile('q_agent.pkl'):
            with open('q_agent.pkl', 'rb') as f:
                self.agent = pickle.load(f)
        else:
            self.agent = QLearner(alpha, gamma, epsilon)
        self.games_played = 0

    def teach(self, iters):
        train(self.agent, iters)

    def playQlearnVSMinimax(self, iters):
        print('Qlearn VS Minimax...')
        runQlearnVSMinimax(self.agent, iters)

    def playQlearnVSBaseline(self, iters):
        print('Qlearn VS Baseline...')
        runQlearnVSBaseline(self.agent, iters)

    def playMinimaxVSBaseline(self, iters):
        print('Minimax VS Baseline...')
        runMinimaxVSBaseline(iters)


if __name__ == '__main__':
```

```
    game_thread = PlayGame()
    game_thread.teach(5000)
    print("q-learn VS baseline 20 times")
    game_thread.playQlearnVSMinimax(20)
    print("")

    print("q-learn VS baseline 50 times")
    game_thread.playQlearnVSMinimax(50)
    print("")

    print("q-learn VS baseline 100 times")
    game_thread.playQlearnVSMinimax(100)
    print("")
```

```
import random
import numpy as np

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


def set_move(state, column, player):
    for i in range(6):
        if state[i][column] != 0:
            state[i - 1][column] = player
            break
        elif i == 5:
            state[i][column] = player


def checkWin(state, player):
    for i in range(6):
        for j in range(4):
            if state[i][j] == state[i][j + 1] == state[i][j + 2] == state[i][j + 3] == player:
                return True

    for i in range(7):
        for j in range(3):
```

```
            if state[j][i] == state[j + 1][i] == state[j + 2][i] == state[j + 3][i] == player:
                return True

    for i in range(3):
        for j in range(4):
            if state[i][j] == state[i + 1][j + 1] == state[i + 2][j + 2] == state[i + 3][j + 3] == player:
                return True

    for i in range(3):
        for j in range(3, 7):
            if state[i][j] == state[i + 1][j - 1] == state[i + 2][j - 2] == state[i + 3][j - 3] == player:
                return True
    return False


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)
```