Artificial Intelligence Assignment 1

1 Game and experiment design

I implemented Tic Tac Toe and Connect 4 in python, using the pygame library to draw the boards and visually show how different algorithms work. The application allows for any combination of human and computer players on any position, and thus can even be used by two people playing together on the same device! When running the code, the main menu screen is opened, from which the two players can be selected. The options for each player are Human, Easy - which always plays random moves, Medium - which wins and blocks if it can and play random moves otherwise, Hard - Minimax, and Hard - Q-Learning. This menu also allows to modify the delay, the number of games, and the window size. The delay applies to the computer player, with the default option being 0.5 seconds between the time a move is computed and when the move is shown on screen. This allows a game of Human vs Computer to feel more natural, and allows to see the sequence of moves when two algorithms play against each other. By setting a number of games to be played, multiple games will be played automatically, with the board resetting itself after each game. This is especially useful for simulating multiple matches between algorithms. Finally, there are two Play buttons, one for Tic Tac Toe and one for Connect 4. The game window has a Clear option, which resets and clears the game board, a button with reopens the main menu, and a Quit button.

I will be comparing the outcomes of different algorithms playing against each other. Each algorithm will play against all the algorithms, including itself. For each pairing, the algorithms will play 100 matches as player 1 and 100 as player 2.

Two default algorithms have been implemented - the easy mode and the medium mode. The Easy algorithm always looks at the possible moves and picks one at random, otherwise disregarding the current game state. The Medium algorithm is just slightly more advanced - It first checks whether there is a winning move, and if it cannot win the game on its current move, it check if the opponent is about to win and tried to block it.

The Minimax algorithm is allowed to run all the way for Tic Tac Toe, with scores set to 1 and -1 for wins and losses respectively. For Connect 4, Minimax is limited to a depth of 5, and a simple heuristic is used to score each state - for any subset of 4 adjacent cells, the score increases by the number of pieces the player has in the subset, as long as the opponent has no pieces in the subset. The scores for wins and losses are set to 300 and -400 to ensure that no matter the state, a win or a loss is felt by the algorithm, and that the algorithm will not risk allowing the game to get close to a loss in hopes of achieving a win.

QLearning was trained for Tic Tac Toe using parameters alpha = 0.2, gamma = 0.9, and epsilon = 0.2. The learning rate alpha determines the percentage distribution of the previous, aggregate q-value and new value. Large values allow the algorithm to quickly internalise the rewards encountered by recomputing a new q-value that is based largely on the new value and only in a small percentage on the previous aggregate. However, this often results in high fluctuations of the q-values and difficulty in convergence. Conversely, low values can impede the algorithm from learning enough from a run, needing a very large number of epochs to converge. The value of 0.2 was picked to avoid oscillations in the q-values and to be able to reliably asses the training progress by testing the performance of the algorithm in between runs of the training algorithm, while keeping the required number of training epochs feasible. The discount factor gamma determines how valuable an reward is depending on how far ahead into the future it is. The value has to be low enough for the algorithm to prioritise achieving an immediate win or avoiding an immediate loss over avoiding a potential dangerous state in the far future or working towards a distant beneficial position. On the other hand, since the only rewards used when training this algorithm are the win and loss rewards, the algorithm has to be able to see far enough into the future for these rewards to trickle up into the early state q-values. A value of 0.9 was chosen for gamma. Finally, epsilon represents the exploration rate - on each move, there is a chance epsilon to experiment with a random move instead of making the current best move. A low value prevents the algorithm from finding new states and makes it reliant on the opponent making new moves. A high value doesn't allow the algorithm to properly develop its end game moves. A value of 0.2 was chosen to balance exploration and exploitation. The Tic Tac Toe player was trained against the Medium opponent. The Connect 4 player went through multiple training runs against the Easy and Medium opponents. After its performance started matching and surpassing these opponents, the player 1 and player 2 algorithms were trained against each other. Most of the runs used the 0.2, 0.9, 0.2 values for the alpha, gamma, epsilon parameters, but about a hundred thousand games were played with linearly decreasing values of epsilon between 0.6 and 0.

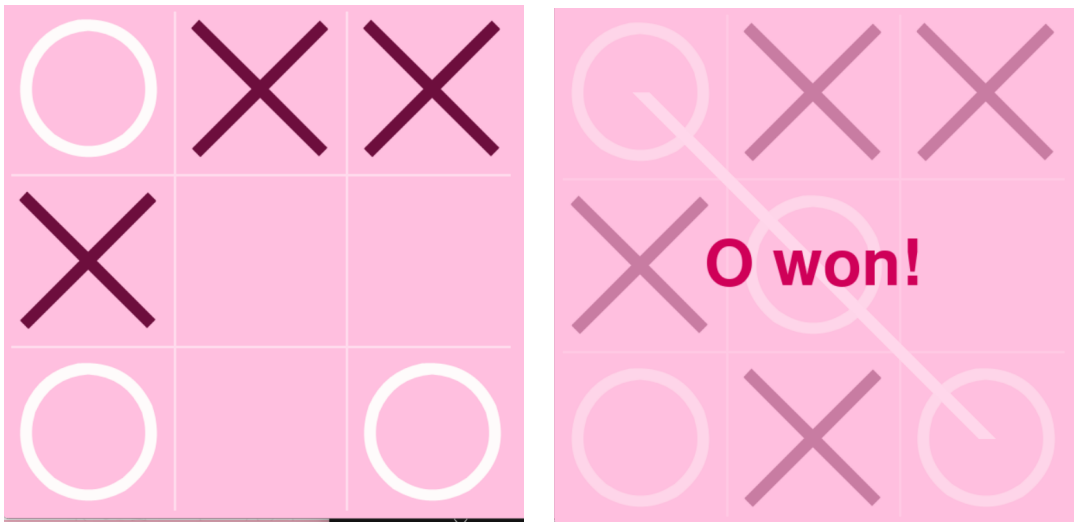
2 Results and discussion

2.1 Tic Tac Toe

Figure 1 shows the game during a game of Tic Tac Toe as well as the win screen. Figure 2 shows the outcomes of 100 games being played by each combination of players. The rows are player 1, and the columns are player 2. The dark pink encodes player 1 victories, light pink corresponds to player 2, and grey shows the draws. First, the two baseline algorithms - Easy and Medium - can be compared to see that in a totally random game, X will win 60% of the time, O 25%, and the game will end in a draw 15% of the time. When one of the players is allowed to seize winning opportunities and block the opponent, this player's chance of winning increase to 90% if it is player 1, and 70% if it is player 2. When both players can make winning and blocking moves, the game end in a draw half of the time, and player 1 is twice as likely to win. Normally, since it is able to block the opponent from winning, the Medium player should win or draw each game. However, both Tic Tac Toe and Connect 4 allow for "traps" to be laid. When a player manages to get multiple sets 2 pieces in a row, or 3 in the case of Connect 4, the opponent can only block one of them, resulting in a win for the player. The Medium algorithm cannot purposely prepare such a game state, nor can it avoid it. Thus, the results show that this type of state randomly occurs about half of the time, and twice as often for player 1 than it does for player 2.

A fully trained Minimax is guaranteed to win or draw every time. When playing as player 1, a is is able to win 100% of the games against the random opponent and 88% against the Medium opponent. These numbers decrease to 77% and 15% when playing as player 2. When playing against itself, Minimax always plays the same sequence of moves, which results in a draw. This makes sense, since the algorithm is guaranteed to either win or draw - the fact that the algorithm can only draw against itself acts as a sanity check. So far, the results consistently showed that beginning the game offers an advantage, however, Minimax shows that even though a win is easier as player 1, a draw can nevertheless be guaranteed.

When playing against the Medium opponent, the Q learning algorithm achieves a very similar performance to Minimax, winning 82% of games as player 1 and 14% as player 2, and only losing 2% of games in both cases. Its performance against the random opponent, however, is surprising, with the Easy player managing to beat Q Learning 13-19% of the time. It is likely that this is due to a training oversight - since the algorithm was trained against the Medium opponent, which would win when it could and block otherwise, the algorithm might not know what to do when it encounters a state that should have lead to a terminal state but instead the opponent allowed it to continue. It might also not know how to make the winning move in states where it didn't have to prepare a trap in order to be able to win. Since both Minimax and Q Learning have their own playbooks based on the best policy or the best q-values, they always play the same game against each other, which ends in a draw. Likewise, Q Learning always draws against itself.



(a) Ongoing (b) Finished

Figure 1: A match of Tic Tac Toe, shown both during the game and after the result

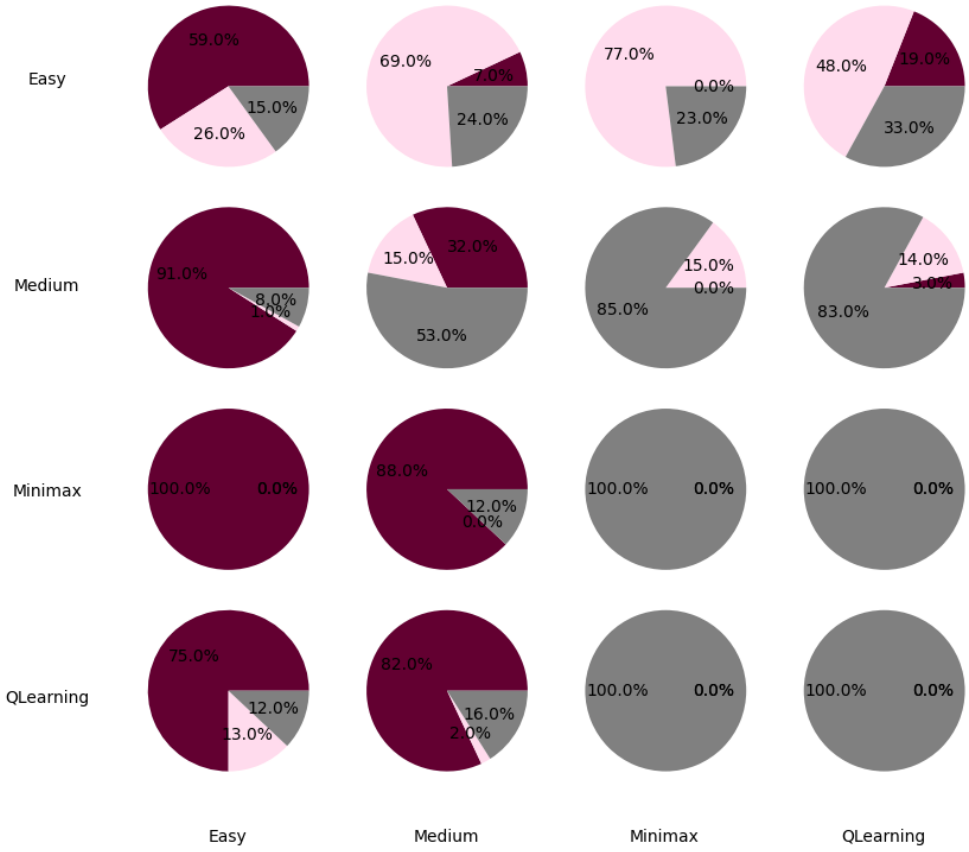


Figure 2: The outcomes of each pairing playing Tic Tac Toe.

2.2 Connect 4

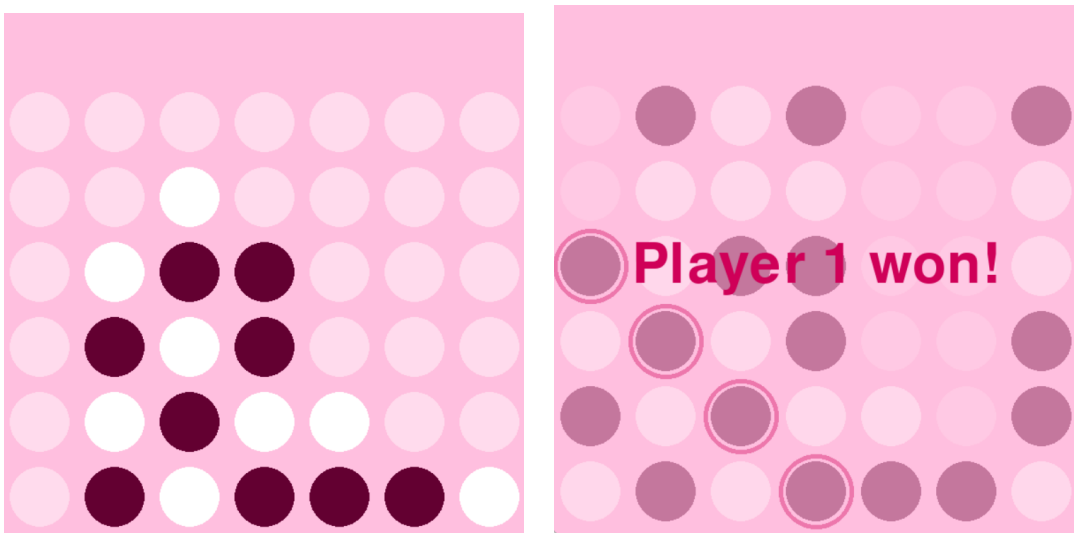
Figure 3 shows the connect 4 game that Minimax plays against itself during the match as well as after the end screen. The dark and light pieces correspond to the first and second players respectively. Figure 4 shows the match outcomes using the same format as figure 2. The baseline algorithms both have a winning rate of around 50% for each player when playing against themselves, with Medium winning slightly more often as player 1, and managing to fill up the board to a draw a few times. Since the Connect 4 board is larger, it offers more possibilities for configurations that Medium cannot block, such as 3 pieces in a horizontal row with free cells on each side. Thus, it is not surprising to see fewer draws. Moreover, since the board is so big, it is difficult to fill it up without ever achieving 4 pieces of the same colour in a row, which explains why Easy never draws. Otherwise, Medium has a huge advantage over the random Easy, beating it about 95% of the time.

Even though Minimax is not fully trained for Connect 4, it manages to beat the random opponent every time with 100% win rates as both player 1 and player 2. However, it loses to the Medium opponent 1% of the time regardless of who goes first,only achieving 99% win rates from both positions. When playing against itself, the game always ends in a win for player 1, with the result shown in figure 3.

Q Learning manages to beat Easy 94% of the time as player 1, which is much better that the performance Easy achieves against itself, but is actually just slightly worse than the Medium player's performance against Easy. Indeed, when playing against Medium, Q Learning only wine 40% of matches, even when playing as player 1. Its win rate as player 2 is worse but follows the same pattern - it performs better than Easy but wore than Medium when playing against either of them, With a 70% win rate against Easy - halfway between Easy and Medium's performance on the same position, and a 30% win rate against Medium - just slightly worse than Medium's win rate against itself, and just slightly wore than its own performance against medium with switched roles.

When playing against Minimax they always play the same games, both won by Minimax. This is especially

interesting since player 1 is supposed to be able to set itself on a guaranteed path to winning by picking the middle column for its first move. The fact that Q Learning cannot win or draw even when it has the first move shows that its training is not thorough enough to beat even a depth-limited Minimax. Finally, when playing against itself, player two wins - pointing to the fact that the previous result against Minimax is driven more by Q Learning's inability to secure a win than it is by Minimax's ability to seize the victory.



(a) Ongoing (b) Finished

Figure 3: A match of Connect 4, shown both during the game and after the result

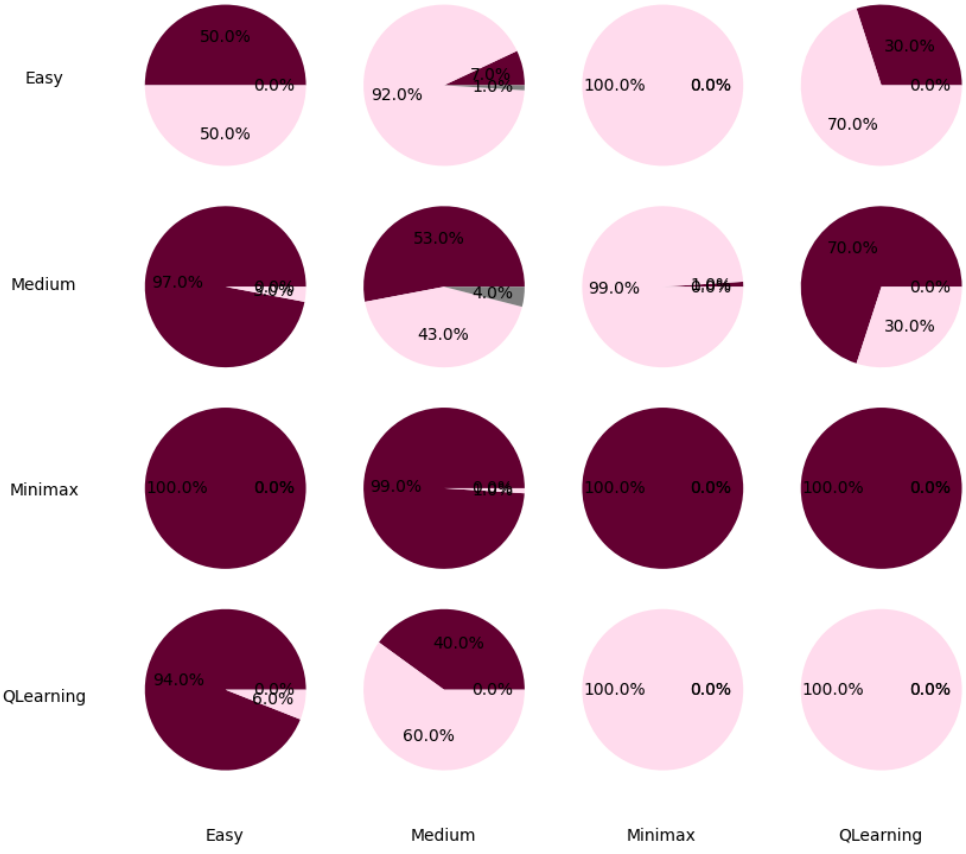


Figure 4: The outcomes of each pairing playing Connect 4. The rows are player 1, and the columns are player 2.

2.3 Comparison

Due to the fundamental differences between the games, the way Minimax and Q Learning are trained and performed also differes across games. Tic Tac Toe has a small enough state space for Minimax to fully explore every possibility and make a move in what feels to a human player like instant time. Q Learning can also achieve good performance with only a few thousand epochs in very little time. Connect 4, however, has a grid that's almost 5 times bigger, with the number of states increasing exponentially to over a trillion states. This number is somewhat reduced by removing impossible states - gaps under pieces or layers covering a 4 pieces of the same colour in a row. Even so, Minimax has to be run with a depth limit and a more specific reward heuristic.

On both Tic Tac Toe and Connect 4 Minimax is near unbeatable from both player positions. However, while it wins almost every game of Connect 4, including the one it plays against itself (it wins as player 1), it draws a very large share of Tic Tac Toe games, even against a completely random opponent. This can be attributed to the fact that Connect 4 offers a lot more possibilities to win, with players being able to secure wins depending on the first move, whereas achieving a draw at Tic Tac Toe is not difficult at all, with a draw being almost guaranteed as long as one does not make a mistake.

When playing Tic Tac Toe, Q Learning shows the peculiarity of performing worse as player 1 against a totally random opponent than it does against the Medium baseline opponent due to not being trained to encounter states where a game should have ended but due to a "mistake" of the opponent they continued. When playing Connect 4, this is no longer the case, since it was trained using multiple types of opponents, including the random player. However, unlike the Tic Tac Toe case, it ends up performing worse than the Medium baseline player when matched directly against it, in spite of having a much more extensive training. This is explained by the complexity of Connect 4, where learning to block and win is a lot easier by analysing the pieces on every move than by maintaining a dictionary of states and moves that can lead to wins and losses. In theory, Q Learning could learn to consistently beat the baseline player by laying out traps and precluding the opponent from doing the same, just like Minimax manages to do, but this would require significantly more training.

The limitations of Q Learning when playing a game with such an extensive potential state space can also be seen in its results against Minimax. While the game ends in a draw whenever the two algorithms play Tic Tac Toe, Minimax beats Q Learning very time at Connect 4. Not only that, but Q Learning even manages to lose against itself, with the game ending in a player 2 victory. This shows that even with a depth limit, Minimax player 2 manages to beat the odds, while Q Learning cannot fructify an advantageous starting position.

Appendix A - Easy

```
def easy(grid):
    legal_moves = []
    for r in range(len(grid)):
        for c in range(len(grid[0])):
            if grid[r][c] == -1:
                legal_moves.append([r,c])
    return sample(legal_moves, 1)[0]
```

Appendix B - Medium Tic Tac Toe

```
def medium_ttt(grid, player):
    opponent = 1 - player
    grid_flat = np.array(grid).flatten()
    player_wins = [player, player, -1], [player, -1, player], [-1, player, player]
    opponent_wins = [opponent, opponent, -1], [opponent, -1, opponent], [-1, opponent, opponent]
    lines = [[0, 1, 2], [3, 4, 5], [6, 7, 8], [0, 3, 6], [1, 4, 7], [2, 5, 8], [0, 4, 8], [2, 4, 6]]
    coords = [[0, 0], [0, 1], [0, 2], [1, 0], [1, 1], [1, 2], [2, 0], [2, 1], [2, 2]]
    # Check winning moves first
    for line in lines:
        l = [grid_flat[line[0]], grid_flat[line[1]], grid_flat[line[2]]]
        if l in player_wins:
            for position in line:
                if grid_flat[position] == -1:
                    return coords[position]
    # Check saving moves second
    for line in lines:
        l = [grid_flat[line[0]], grid_flat[line[1]], grid_flat[line[2]]]
        if l in opponent_wins:
            for position in line:
                if grid_flat[position] == -1:
                    return coords[position]
    legal_moves = [i for i in range(9) if grid_flat[i] == -1]
    try:
        return coords[sample(legal_moves, 1)[0]]
    except:
        return None
```

Appendix C - Medium Connect 4

```
def medium_c4(grid, player):
    opponent = 1 - player
    player_wins = [[player, player, player, -1], [player, player, -1, player],
    [player, -1, player, player],[-1, player, player, player]]
    opponent_wins = [[opponent, opponent, opponent, -1], [opponent, opponent, -1, opponent],
                     [opponent, -1, opponent, opponent], [-1, opponent, opponent, opponent]]
    for win_set in [player_wins, opponent_wins]:
        for i in range(6):
            for j in range(4):
                positions = [[i, j], [i, j + 1], [i, j + 2], [i, j + 3]]
                if [grid[i][j], grid[i][j + 1], grid[i][j + 2], grid[i][j + 3]] in win_set:
                    for m, move in enumerate([grid[i][j], grid[i][j + 1],
                    grid[i][j + 2], grid[i][j + 3]]):
```

```
                    if move == -1:
                        return positions[m]
        for j in range(7):
            for i in range(3):
                positions = [[i, j], [i + 1, j], [i + 2, j], [i + 3, j]]
                if [grid[i][j], grid[i + 1][j],
                grid[i + 2][j], grid[i + 3][j]] in win_set:
                    for m, move in enumerate([grid[i][j], grid[i + 1][j],
                    grid[i + 2][j], grid[i + 3][j]]):
                        if move == -1:
                            return positions[m]
        for i in range(3):
            for j in range(4):
                positions = [[i, j], [i + 1, j + 1], [i + 2, j + 2], [i + 3, j + 3]]
                if [grid[i][j], grid[i + 1][j + 1],
                grid[i + 2][j + 2], grid[i + 3][j + 3]] in win_set:
                    for m, move in enumerate([grid[i][j], grid[i + 1][j + 1],
                    grid[i + 2][j + 2], grid[i + 3][j + 3]]):
                        if move == -1:
                            return positions[m]
                positions = [[i, j + 3], [i + 1, j + 2], [i + 2, j + 1], [i + 3, j]]
                if [grid[i][j + 3], grid[i + 1][j + 2],
                grid[i + 2][j + 1], grid[i + 3][j]] in win_set:
                    for m, move in enumerate([grid[i][j + 3], grid[i + 1]
                    [j + 2], grid[i + 2][j + 1], grid[i + 3][j]]):
                        if move == -1:
                            return positions[m]

    legal_moves = []
    for j in range(7):
        for i in range(6):
            if grid[i][j] == -1:
                legal_moves.append([i, j])
    try:
        return sample(legal_moves, 1)[0]
    except:
        return None
```

Appendix D - Minimax Tic Tac Toe

```
def minimax_ttt(grid, turn, player, opponent, alpha, beta):
    legal_moves = []
    for j in range(3):
        for i in range(3):
            if grid[i][j] == -1:
                legal_moves.append([i, j])
    best_move = [-math.inf, None, None] if turn == player else [math.inf, None, None]
    if len(legal_moves) == 0:
        return evaluate_ttt(grid, player), None, None
    for move in legal_moves:
        grid_new = np.copy(grid)
        r, c = move
        grid_new[r][c] = turn
        payoff = evaluate_ttt(grid_new, player)
        if payoff == None:
            payoff = minimax_ttt(grid_new, 1 - turn, player, opponent, alpha, beta)[0]
```

```
        if turn == player:
            if payoff > best_move[0]:
                best_move = [payoff, r, c]
            alpha = max(alpha, best_move[0])
        else:
            if payoff < best_move[0]:
                best_move = [payoff, r, c]
            beta = min(beta, best_move[0])
        if beta <= alpha:
            break
    return best_move

    def evaluate_ttt(grid, player):
    grid_flat = np.array(grid).flatten()
    opponent = 1 - player
    if grid[2][0] == grid[1][1] == grid[0][2]:
        if grid[1][1] == player:
            return 1
        if grid[1][1] == opponent:
            return -1
    if grid[0][0] == grid[1][1] == grid[2][2]:
        if grid[1][1] in [0, 1]:
            if grid[1][1] == player:
                return 1
            if grid[1][1] == opponent:
                return -1
    for i in range(3):
        if grid[i][0] == grid[i][1] == grid[i][2]:
            if grid[i][1] == player:
                return 1
            if grid[i][1] == opponent:
                return -1
        elif grid[0][i] == grid[1][i] == grid[2][i]:
            if grid[1][i] == player:
                return 1
            if grid[1][i] == opponent:
                return -1
    if -1 not in grid_flat:
        return 0
    return None
```

Appendix E - Minimax Connect 4

```
def minimax_c4(grid, turn, player, opponent, alpha, beta, depth):
legal_moves = []
for i in range(6):
    for j in range(7):
        if grid[i][j] == -1:
            legal_moves.append([i, j])
best_move = [-math.inf, None, None] if turn == player else [math.inf, None, None]
payoff = evaluate_c4(grid, player)
if depth == 0 or abs(payoff) > 100 or len(legal_moves) == 0:
    return payoff, None, None
for move in legal_moves:
    grid_new = np.copy(grid)
    r, c = move
```

```
            grid_new[r][c] = turn
            if r > 0:
                grid_new[r - 1][c] = -1
            payoff = minimax_c4(grid_new, 1 - turn, player, opponent, alpha, beta, depth - 1)[0]
            if turn == player:
                if payoff > best_move[0]:
                    best_move = [payoff, r, c]
                alpha = max(alpha, best_move[0])
            else:
                if payoff < best_move[0]:
                    best_move = [payoff, r, c]
                beta = min(beta, best_move[0])
            if beta <= alpha:
                break
        return best_move


def evaluate_c4(grid, player):
    grid_flat = np.array(grid).flatten()
    opponent = 1 - player
    score = 0
    for i in range(6):
        for j in range(4):
            score = score + scoring([grid[i][j], grid[i][j + 1],
            grid[i][j + 2], grid[i][j + 3]], player)
    for j in range(7):
        for i in range(3):
            score = score + scoring([grid[i][j], grid[i + 1][j],
            grid[i + 2][j], grid[i + 3][j]], player)
    for i in range(3):
        for j in range(4):
            score = score + scoring([grid[i][j], grid[i + 1][j + 1],
            grid[i + 2][j + 2], grid[i + 3][j + 3]], player)
            score = score + scoring([grid[i][j + 3], grid[i + 1][j + 2],
            grid[i + 2][j + 1], grid[i + 3][j]], player)
    if -1 not in grid_flat:
        return 0
    return score

def scoring(points, player):
    count_player = sum(1 for p in points if p == player)
    count_opponent = sum(1 for p in points if p == 1 - player)
    if count_player == 4:
        return 300
    if count_opponent == 4:
        return -400
    if count_opponent == 0:
        return count_player
    if count_player == 0:
        return -count_opponent
    return 0
```

Appendix F - Q Learning Training Tic Tac Toe

```
def q_learning_train_ttt(player, alpha=0.2, gamma=0.9, epsilon=0.2):
    moves = [[i, j] for i in range(3) for j in range(3)]
```

```
    file = 'q' + str(player) + 'ttt.pickle'
    try:
        with open(file, 'rb') as f:
            q_table = pickle.load(f)
    except:
        q_table = {}
    for game in range(100):
        grid = [[-1 for _ in range(3)] for _ in range(3)]
        legal_moves = [i for i in range(9)]
        # If the opponent is X, it makes the first move
        if (player == 1):
            m_opponent = sample(legal_moves, 1)[0]
            grid[moves[m_opponent][0]][moves[m_opponent][1]] = 1 - player
            legal_moves.pop(legal_moves.index(m_opponent))
        q_move_ttt(q_table, grid, legal_moves, player, alpha, gamma, epsilon)
    with open(file, 'wb') as f:
        pickle.dump(q_table, f)


def q_move_ttt(q_table, grid, legal_moves, player, alpha, gamma, epsilon):
    state = grid_to_state(grid)
    moves = [[i, j] for i in range(3) for j in range(3)]
    reward = evaluate_ttt(grid, player)
    # Return the final state value if it is a leaf
    if reward is not None:
        return reward
    else:
        if state in q_table:
            # Try to pick the best move for the current state, pick a random move and
            add it if there are no other moves in the state
            try:
                m = max(q_table[state], key=q_table[state].get)
            except:
                m = sample(legal_moves, 1)[0]
        else:
            # Add state to the dictionary if it doesn't exist and pick a random move
            q_table[state] = {}
            m = sample(legal_moves, 1)[0]
        # With probability epsilon, pick a random move regardless
        if random.randint(1, 10) <= epsilon * 10:
            m = sample(legal_moves, 1)[0]
        # Adding the move to the table if it doesn't exist
        if m not in q_table[state]:
            q_table[state][m] = 0
        # Make the move
        new_grid = np.copy(grid)
        new_grid[moves[m][0]][moves[m][1]] = player
        legal_moves.pop(legal_moves.index(m))
        # If it's not a leaf after the new move, opponent moves too
        reward = evaluate_ttt(new_grid, player)
        if reward is None:
            # [r,c] = q_learning_ttt(new_grid, 1-player)
            # [p,r,c] = minimax_ttt(new_grid,1-player,1-player,player,-math.inf,math.inf)
            [r, c] = medium_ttt(new_grid, 1 - player)
            m_opponent = moves.index([r, c])
            new_grid[r][c] = 1 - player
            legal_moves.pop(legal_moves.index(m_opponent))
        # Updating q_table
```

```
        q_table[state][m] += alpha * (gamma * q_move_ttt(q_table, new_grid,
        legal_moves, player, alpha, gamma, epsilon) - q_table[state][m])
    # Returning the best value for the parent q-value
    return max(q_table[state].values())
```

Appendix G - Q Learning Training Connect 4

```
def q_learning_train_c4(player, alpha=0.2, gamma=0.9, epsilon=0.2, opponent = 'Random'):
    file = 'q' + str(player) + 'c4.pickle'
    file = 'q' + str(1-player) + 'c4.pickle'
    try:
        with open(file, 'rb') as f:
            q_table = pickle.load(f)
    except:
        q_table = {}
    try:
        with open(file, 'rb') as f:
            q_table_opponent = pickle.load(f)
    except:
        q_table_opponent = {}
    for game in range(10000):
        grid = [[-2 for _ in range(7)] for _ in range(5)]
        grid.append([-1 for _ in range(7)])
        legal_moves = [i for i in range(7)]
        # If the opponent is X, it makes the first move
        if (player == 1):
            m_opponent = sample(legal_moves, 1)[0]
            grid[5][m_opponent] = 0
            grid[4][m_opponent] = -1
        q_move_c4(q_table, q_table_opponent, grid, legal_moves, player, alpha,
        gamma, epsilon, opponent)
    with open(file, 'wb') as f:
        pickle.dump(q_table, f)


def q_move_c4(q_table, q_table_opponent, grid, legal_moves, player, alpha,
gamma, epsilon, opponent = 'Random'):
    state = grid_to_state(grid)
    reward = evaluate_c4_q(grid, player)
    moves = copy.deepcopy(legal_moves)
    # Return the final state value if it is a leaf
    if reward is not None:
        return reward
    else:
        if state in q_table:
            # Try to pick the best move for the current state, pick a random move and
            add it if there are no other moves in the state
            try:
                m = max(q_table[state], key=q_table[state].get)
            except:
                m = sample(moves, 1)[0]
        else:
            # Add state to the dictionary if it doesn't exist and pick a random move
            q_table[state] = {}
            m = sample(moves, 1)[0]
        # With probability epsilon, pick a random move regardless
```

```
        if random.randint(1, 10) <= epsilon * 10:
            m = sample(moves, 1)[0]
        # Adding the move to the table if it doesn't exist
        if m not in q_table[state]:
            q_table[state][m] = 0
        # Make the move
        new_grid = np.copy(grid)
        for row in range(5, -1, -1):
            if new_grid[row][m] == -1:
                new_grid[row][m] = player
                if row > 0:
                    new_grid[row - 1][m] = -1
                else:
                    moves.pop(moves.index(m))
                break

        # If it's not a leaf after the new move, opponent moves too
        reward = evaluate_c4_q(new_grid, player)
        if reward is None:
            #c = sample(legal_moves, 1)[0]
            c = medium_c4(new_grid, 1 - player)[1]  if opponent == 'Medium'
            else q_learning_c4(new_grid, 1 - player, q_table_opponent)
            if opponent == 'QLearning' else sample(moves, 1)[0]
            # [p,r,c] = minimax_c4(new_grid,1-player,1-player,player,-math.inf,math.inf,5)
            #[r, c] = easy_c4(new_grid, 1 - player)
            for row in range(5, -1, -1):
                if new_grid[row][c] == -1:
                    new_grid[row][c] = 1 - player
                    if row > 0:
                        new_grid[row - 1][c] = -1
                    else:
                        moves.pop(moves.index(c))
                    break
        # Updating q_table
        q_table[state][m] += alpha * (
                gamma * q_move_c4(q_table, q_table_opponent, new_grid, moves, player, alpha,
                gamma, epsilon, opponent) - q_table[state][m])
    # Returning the best value for the parent q-value
    return max(q_table[state].values())

def grid_to_state(grid):
    state = ""
    for row in grid:
        for cell in row:
            state += str(cell)
    return state
```

Appendix H - Q Learning Player Tic Tac Toe

```
def q_learning_ttt(grid, player):
    moves = [[i, j] for i in range(3) for j in range(3)]
    file = 'q' + str(player) + 'ttt.pickle'
    try:
        with open(file, 'rb') as f:
            q_table = pickle.load(f)
```

```
    except:
        print("This player is untrained.")
    state = grid_to_state(grid)
    try:
        best_move = max(q_table[state], key=q_table[state].get)
        return moves[best_move]
    except:
        # print("This game state has not been encountered yet. A random move will be played")
        legal_moves = []
        for i in range(3):
            for j in range(3):
                if grid[i][j] == -1:
                    legal_moves.append([i, j])
        return sample(legal_moves, 1)[0]
```

Appendix I - Q Learning Player Connect 4

```
def q_learning_c4(grid, player, q_table = None):
    moves = [[i, j] for i in range(3) for j in range(3)]
    file = 'q' + str(player) + 'c4.pickle'
    if q_table == None:
        try:
            with open(file, 'rb') as f:
                q_table = pickle.load(f)
        except:
            print("This player is untrained.")
    state = grid_to_state(grid)
    try:
        best_move = max(q_table[state], key=q_table[state].get)
        return best_move
    except:
        # print("This game state has not been encountered yet. A random move will be played")
        legal_moves = [i for i in range(7)]
        for j in range(7):
            if grid[0][j] in [0, 1]:
                legal_moves.pop(legal_moves.index(j))
        return sample(legal_moves, 1)[0]
```