CS7IS2: Artificial Intelligence Assignment 2

Submitted By

Pallavit Aggarwal - MSc Intelligent Systems
Student Number - 22333721
Email ID - aggarwpa@tcd.ie

Design Considerations

Q Learning:

Learning Rate:

The learning rate (also known as alpha) in Q-learning is a hyperparameter that determines how quickly the Q-values are updated during the learning process. A high learning rate means that the agent will learn from new experiences more quickly, while a low learning rate means the agent will learn more slowly.

A high learning rate can be beneficial in certain situations, especially in the initial stages of learning, as it allows the agent to explore and learn from new experiences quickly. However, a high learning rate can also cause the agent to overvalue recent experiences and become less stable, as it might overwrite previously learned knowledge too quickly. On the other hand, a low learning rate can make the agent's learning process more stable and less likely to oscillate. But it can also slow down the learning process, which might be a problem in cases where the agent needs to adapt quickly to new situations or where the environment is highly dynamic.

Gamma:

Gamma, or the discount factor, is a hyperparameter in Q-learning that determines the agent's preference for immediate rewards over future rewards. A low gamma value (closer to 0) means that the agent will have a strong preference for immediate rewards, whereas a high gamma value (closer to 1) means the agent will consider both immediate and future rewards more equally.

In the game of Connect 4, it is essential to consider future rewards, as each move can significantly affect the game's outcome.

A low gamma value impacts the Q-learning agent in the following ways:

Short-sightedness: With a low gamma, the agent focuses more on immediate rewards and becomes less concerned about the long-term consequences of its actions.

Faster convergence: Since the agent is less concerned about future rewards, it may converge to a suboptimal policy more quickly.

Less exploration: In some cases, a low gamma value can discourage the agent from exploring the environment effectively. If the agent is more focused on immediate rewards, it might not explore different action sequences that could potentially lead to higher long-term rewards.

Epsilon:

Epsilon (ε) is a hyperparameter in Q-learning that controls the exploration-exploitation trade-off. It determines the probability with which the agent takes a random action (exploration) instead of choosing the action with the highest estimated Q-value (exploitation).

Exploration: A higher epsilon value encourages more exploration, as the agent is more likely to take random actions. This can be beneficial early in training when the agent has limited knowledge about the environment and needs to gather information about different state-action pairs.

Exploitation: A lower epsilon value encourages more exploitation, as the agent is more likely to choose actions based on its current knowledge. As the agent gains more experience and develops a more accurate Q-value estimate, exploitation becomes more important to maximize the cumulative rewards.

Epsilon-decay strategies and Balancing exploration and exploitation: I n practice, it's common to use an epsilon-decay strategy, where the value of epsilon starts high (to encourage exploration) and gradually decreases over time (to encourage exploitation).

Incomplete learning: The Q-learning algorithm needs many episodes to learn the optimal strategy for a game like Connect Four. The training process might not be enough for the AI players to learn the perfect strategy. As a result, one player might still have some weaknesses that the other player can exploit, leading to a win.

Pessimistic and Optimistic Initialization:

Pessimistic: If we initialise all values in the Q Table with 0, the player will be very pessimistic, i.e. it will assume every move leads to a loss, and is likely to settle for anything that is better than losing. E.g. if the player achieves a draw before its first win, it will increase the values of the moves that lead to the draw while all other move values will still be 0. In subsequent games it will favour using those moves again over trying something new with the potential chance of actually winning.

Optimistic: On the other hand, if we initialise the values to 1, the player will be very optimistic and expect every move to lead to victory. The player is thus less likely to settle for a draw as best possible outcome and will actively explore other options first. The downside however is that it will learn quite slowly as it will exhaust every other option before settling on a strategy that leads to a draw.

(I tried first with Optimistic, but didn't really see any major difference in the game play, so I kept it initialized to default 0, which would be the pessimistic initialization).

On Connect 4:
In connect 4, the best move function is responsible for finding the optimal column for the AI player to make a move. Since the pieces always fall to the lowest unoccupied row in a column, we don't need to explicitly find the "best row" as it will be determined automatically once we choose the optimal column. When the AI agent selects a column and drops a piece in it, the get_next_open_row function is called to find the row where the piece will land. This function iterates through the rows in a column and returns the first unoccupied row it encounters. So, once we find the optimal column, the row is determined by the current state of the board and the rules of the game.

Minimax :

The MiniMax algorithm is a recursive algorithm used in decision-making and game theory. It delivers an optimal move for the player, considering that the competitor is also playing optimally. This algorithm is widely used for game playing in Artificial Intelligence, such as chess, tic-tac-toe, and myriad double players games. In this algorithm, two players play the game; one is called 'MAX', and the other is 'MIN.' The goal of players is to minimize the opponent's benefit and maximize self-benefit. The MiniMax algorithm conducts a depth-first search to explore the complete game tree and then proceeds down to the leaf node of the tree, then backtracks the tree using recursive calls.

Algorithm: At any given state, enumerate the possible child states. Now determine the value of these child states by invoking the minimizing player i.e. Ask the minimizing player what he would do in

each of these states and return the value he gets. Choose the action that leads to the highest value state. This is implemented in the maximize function.

Similarly for a minimizing player, enumerate the child states and determine the value of each state by invoking the maximizing player. Choose the action that leads to the lowest value state. This is implemented in the minimize function.

If at any stage, the child state is a terminal state, the value of the terminal state is simply returned.

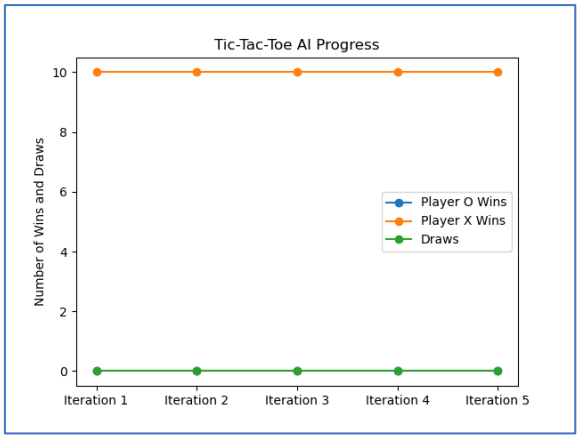
max_depth: The reason max_depth is useful in the minimax algorithm is that it helps to manage the complexity of the search. As the depth of the search increases, the number of possible game states that need to be evaluated grows exponentially. This can quickly become intractable for games with large game trees, like Connect 4, chess, or Go. By limiting the depth of the search, we can strike a balance between the quality of the decision-making and the amount of time spent computing the best move.

Comparison

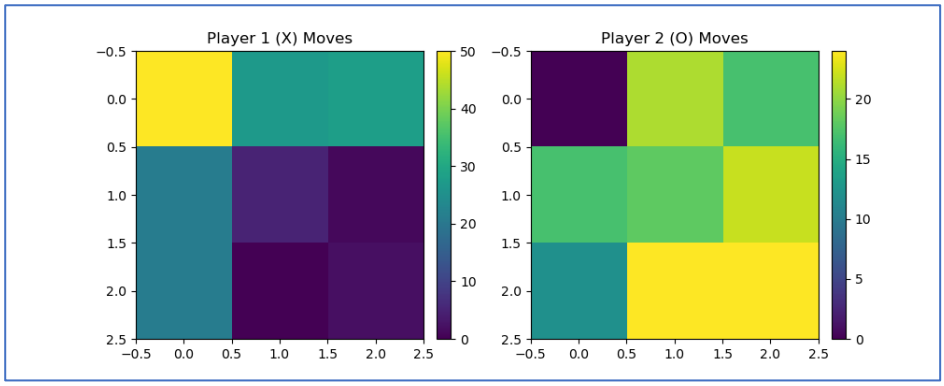
a) Tic Tac Toe

i) Minimax Agent

Case 1: X plays first ; X is minimax, O is Random Agent



The following game has been run for 5 iterations with 10 games per iteration. (Since minimax was performing better than q learning, I eventually realise that mentioning even few games against random is sufficient). We observe that X wins every game against the random agent with a "max_depth" set to 3.

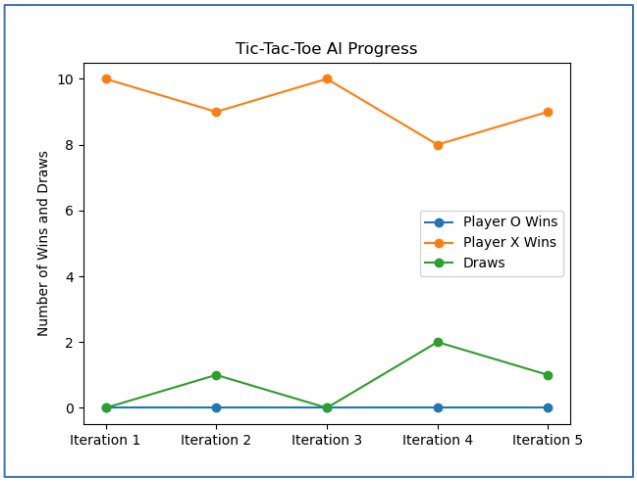


The heat map above shows the move selection done by minimax in Player 1 (X) Moves and by the random agent in Player 2 (O) moves. What is interesting to note is that, minimax doesn't consider blocking the centre space intuitively. Also, Minimax's preference is the top left corner,

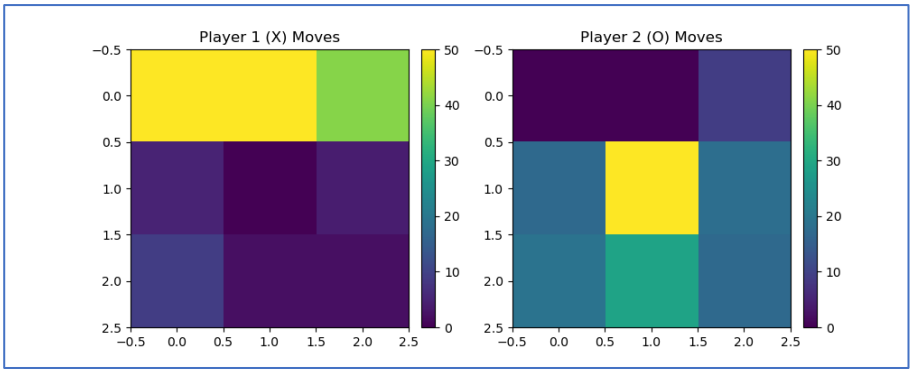
which is occupies almost in every game. Player O doesn't get to make a lot of moves, as the game is finished usually before O gets to finish the board to draw or last move.

Case 2: X plays first ; X is minimax, O is Random Agent and O blocks centre if available.

This case is important, because of the above noted tendency of minimax not occupying the centre. I wanted to see how it would perform, if the centre is blocked by the random agent.

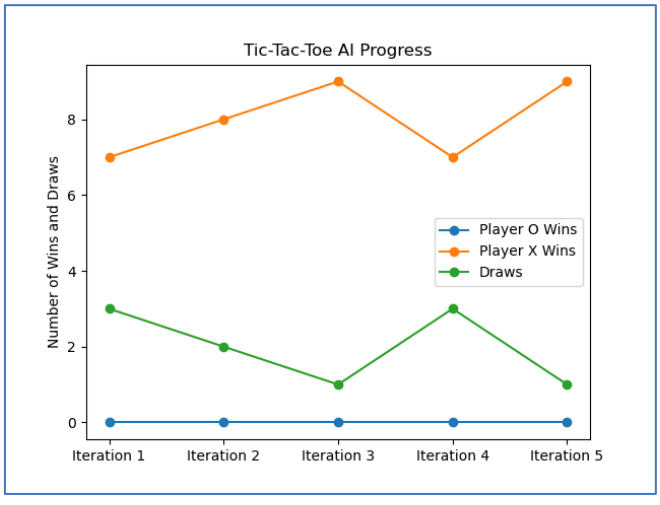


This has also been run for 5 iterations for 10 games in each iteration with a max depth of 3. Here we can see that the random agent is able to draw the minimax in iteration 2 and 4.

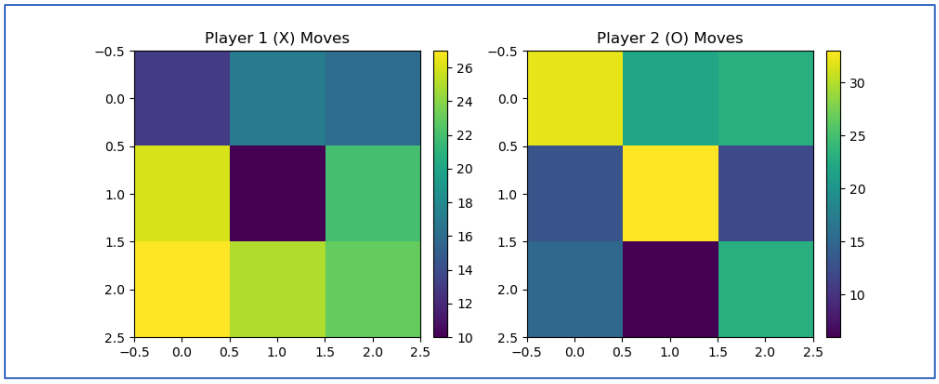


The heatmap above shows that Minimax still prefers to stay in the top line of the board, and doesn't increase its choice to occupy the centre position to block O.

Case 3: Case 1 variant where O plays first ; X is minimax, O is Random Agent

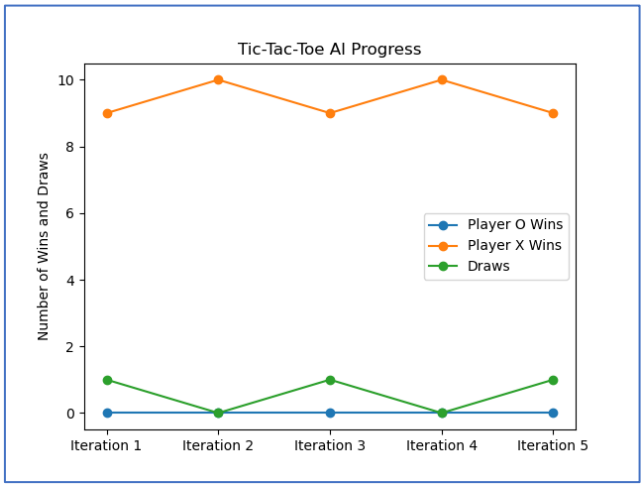


Playing first gives an advantage to O and this performance is almost comparable to the previous one with O blocking centre against minimax.

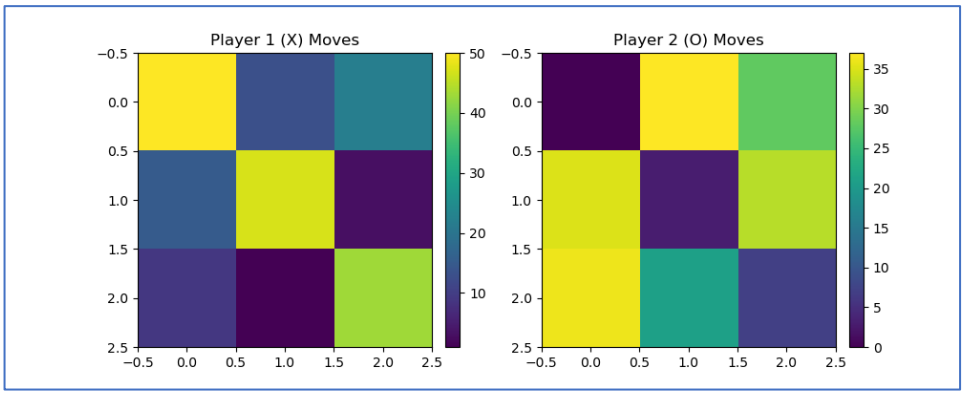


The heatmap here shows a different performance by the X playing as a minimax, to counter the first move played by O. O's preference for centre and top left is countered by bottle left and bottom region by the Minimax.

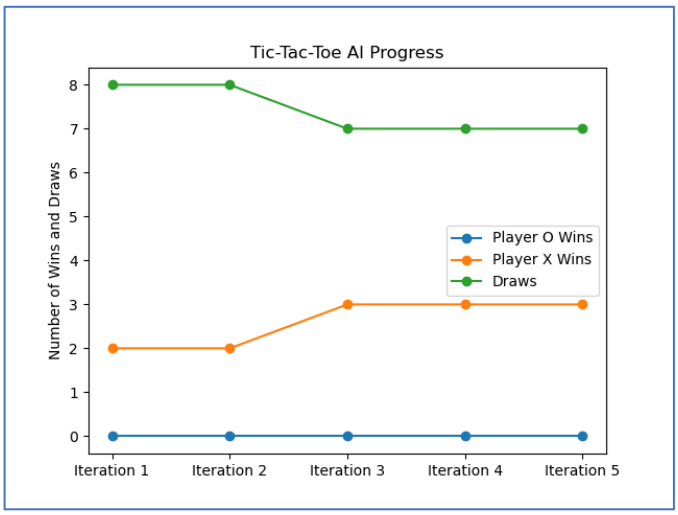
Case 4: X plays first ; X is minimax, O is Default Agent. A default agent can take a winning move or block an opponent's winning move.



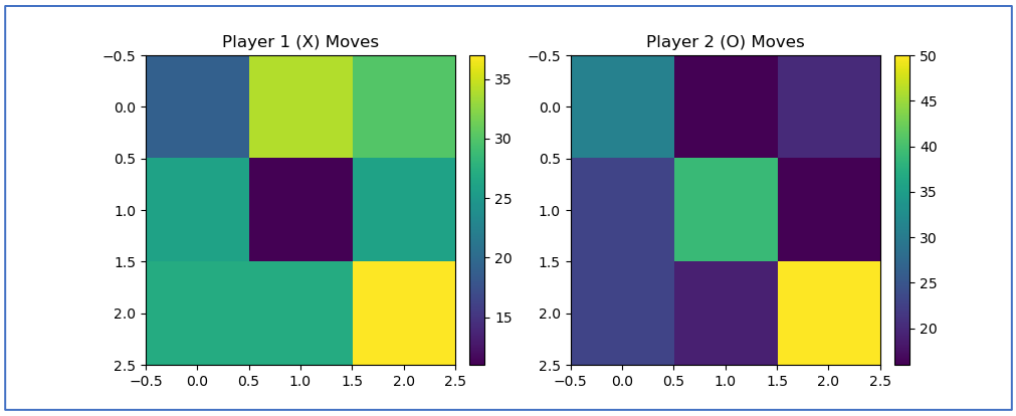
Here, minimax still plays optimally, but is countered by the default opponent leading to some draws. Comparing this with **Case 1,** the agent does play optimally.



Case 5: O plays first ; X is minimax, O is Default Agent. A default agent can take a winning move or block an opponent's winning move.



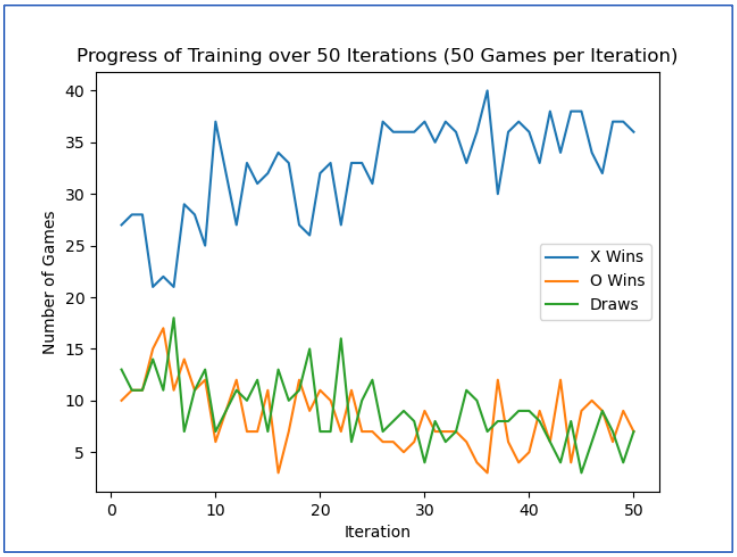
If O gets the lead, and O is the default opponent, then that skews the minimax heavily towards draws.



Above heatmap shows that When O leads as a default agent against Minimax, it's preference usually lies towards bottom left corner or the centre of the board which makes it optimal for the gameplay and leads to more draws as X tries to win the game.

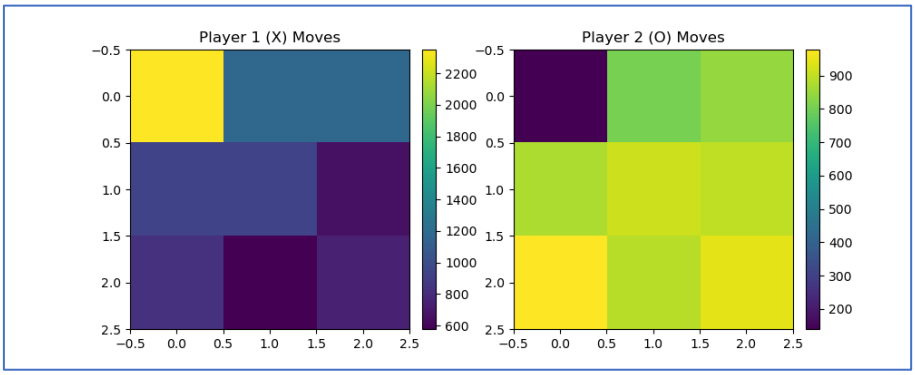
ii) Q Learning Agent

Case 1: X plays first ; X is Q learning agent and O is the random agent.



The parameters set here are epsilon=0.01, gamma=0.9 and alpha=0.9. Performance is mentioned over 50 iterations of 50 games in each iteration.

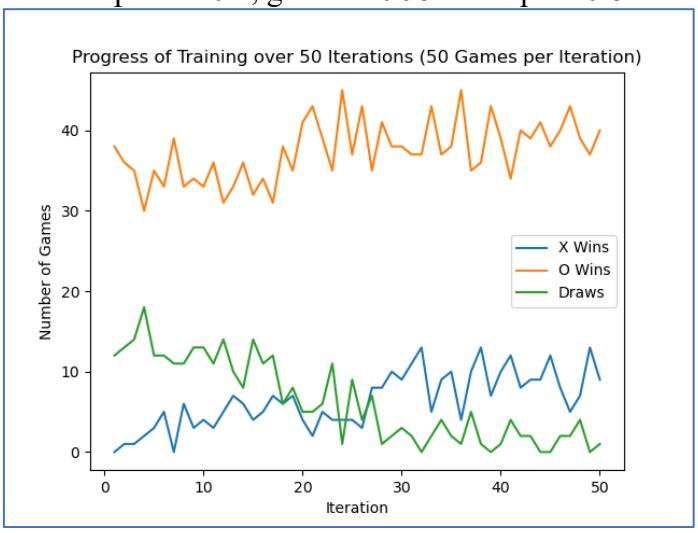
As we can see for a high learning rate alpha, and a discount factor of 0.9 leading to the algorithm prioritizing a longer game for a higher reward, and an epsilon of 0.01 to limit epsilon greediness and make the algorithm more exploitative, the algorithm gradually increases the wins and gets it to the higher 90s in percentage wins and prioritizes the draws the same as letting O win. This also has to do with the way the reward structure has been set.



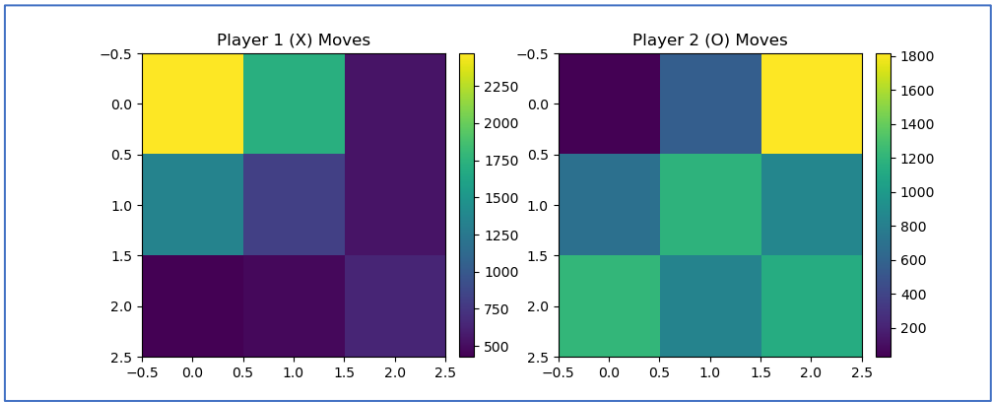
While a random R chooses almost any place with a high probability, the Q learning algorithm prefers the top left corner and the top horizontal line for most of the games.

Case 2: X plays first ; X is Q learning agent and O is the default agent. A default agent can take a winning move or block an opponent's winning move.

The parameters set here are epsilon=0.2, gamma=0.95 and alpha=0.8.



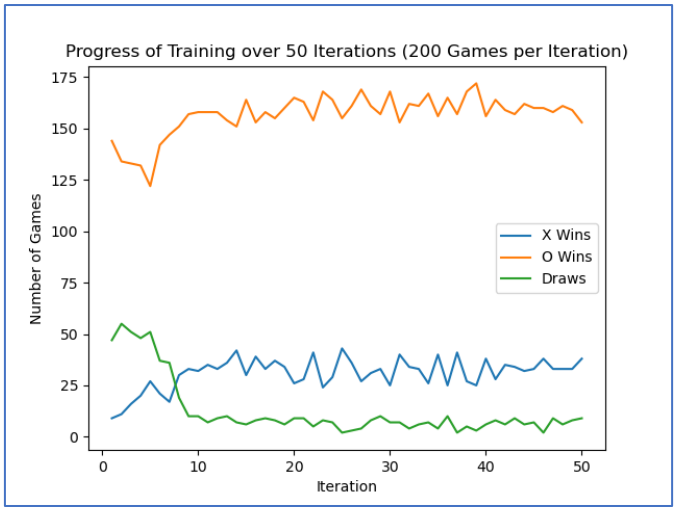
Here, the O default agent performs near optimally, by selecting any blocking moves, and choosing a win, otherwise playing random. This was interesting to observe because while the Q learning agent performed really well against the random agent, here it is able to bring down the draws and prioritize wins after the 25th iteration onwards.

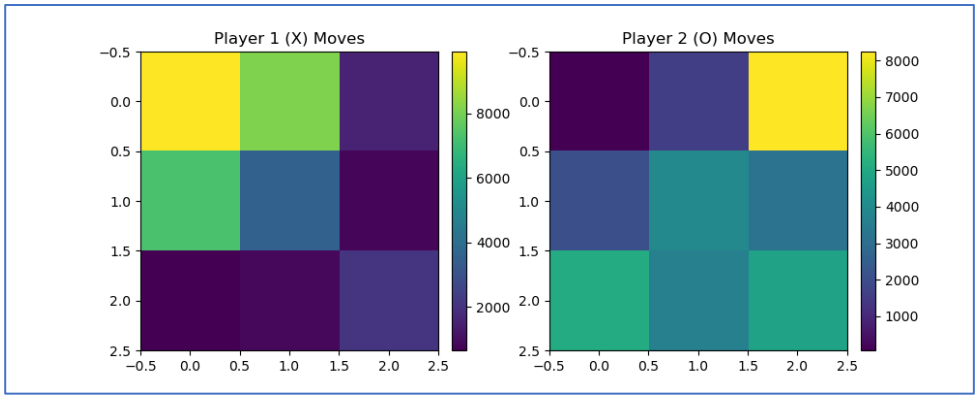


Case 2.1: X plays first ; X is Q learning agent and O is the default agent. A default agent can take a winning move or block an opponent's winning move.

The parameters set here are epsilon=0.01, gamma=0.95 and alpha=0.95.

In this case the agent plays for 50 iterations with 200 games in each iteration. While the agent learns to win and performs consistently better after the 10th iteration, it still doesn't defeat the default opponent.



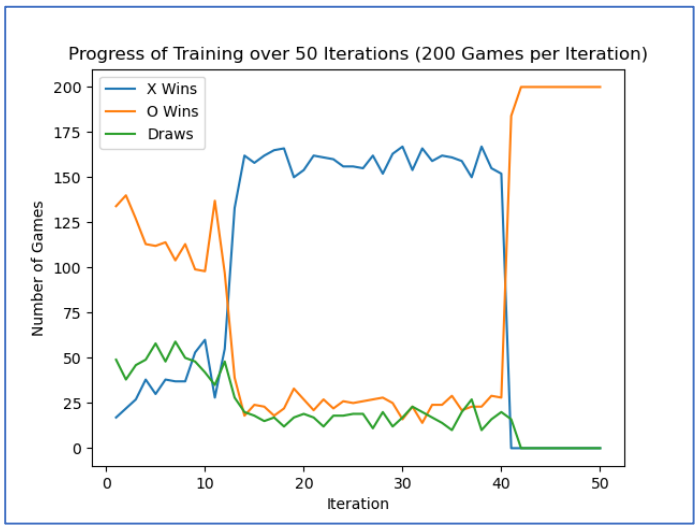


iii) Q Learning Agent vs Minimax Agent

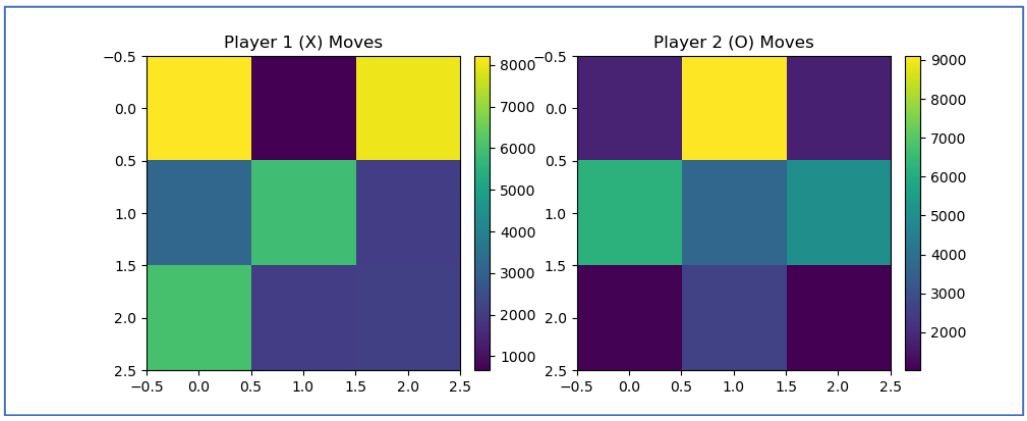
Case 1: X plays first ; X is Q Learning agent and O is Minimax agent

In this case since from observations it is clear that Minimax would perform optimally over 10000 games as compared to Q learning agent, I set the initial values of Alpha to 0.5 , Gamma to 0.5 and Epsilon to 0.5. This allows the Q learning agent to be more exploratory, stable and go for a balanced win rate strategy. Here the max depth is also set to 1. This makes the performance of Minimax a little random, short sighted and comparable to the performance of Q Learning agent for the first few iterations.

Then I apply the following rules: if(iteration>=10 and iteration<=30): set max_depth=2, epsilon=0.1, gamma=0.9, alpha=0.8. We can see that after the 10th iteration, this drastically improves the Q Learning agent.



Then, I apply the next rule which is if(iteration>=40): set max_depth=3, epsilon=0.001, gamma=0.9, alpha=0.9. Once the max depth reaches 3, the performance of Minimax defeats the Q Learning agent sharply.

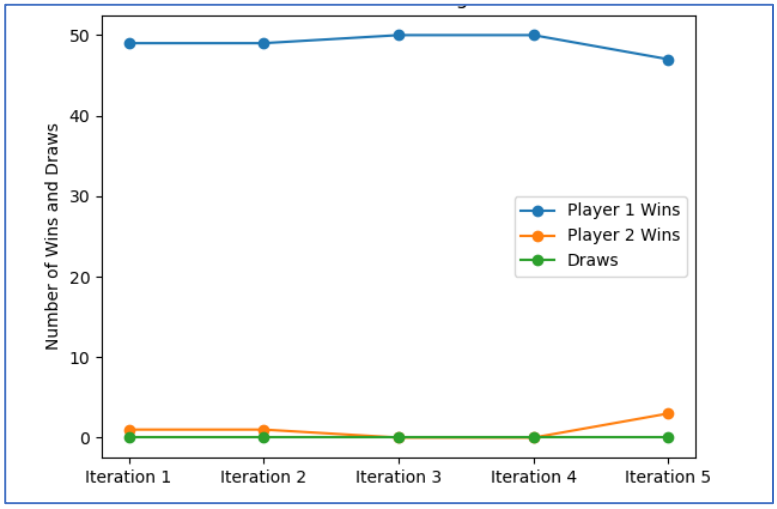


The heatmap for the above is interesting, as it shows that the Q Learning agent identifies the centre of the board as an important area and selects that more number of times than the minimax algorithm.

b) Connect 4

i) Minimax Agent

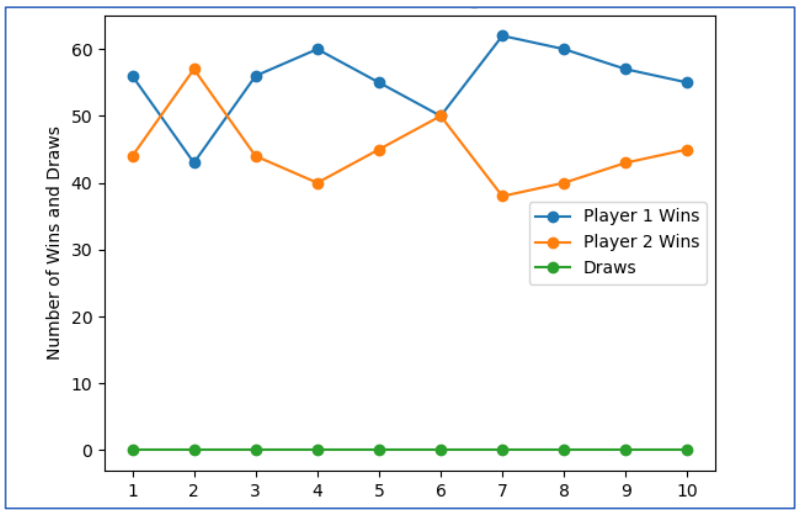
Case 1: Red plays first ; Red is minimax, Yellow is Random Agent
Since the number of game states that can explored in a game of connect 4 of size 6x7 can reach approx. 4.2 Trillion, max depth has to be set as a limiting parameter that cuts off the minimax function and returns the result till a particular depth only. I first run the test against the random agent with a max depth of 5.
Following is the result.



The minimax agent performs better than a random agent 99% of the times.

Case 2: Red plays first ; Red is minimax, Yellow is Default Agent. Max_Depth 5

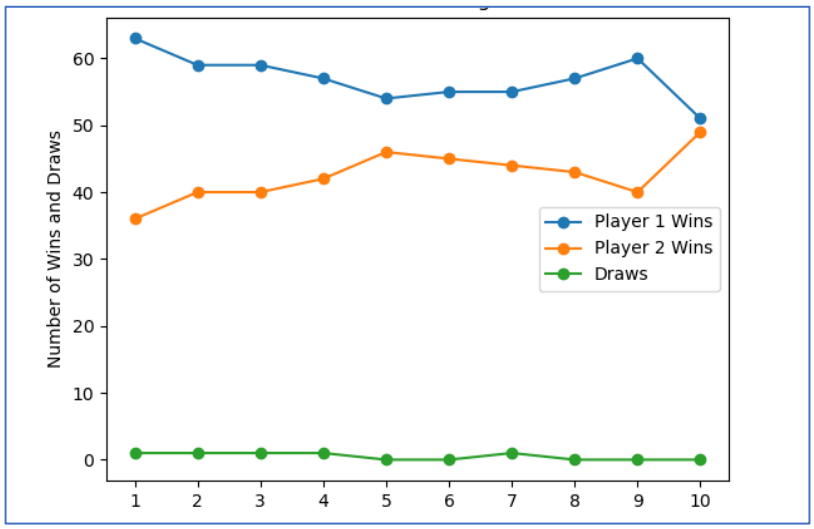
In this case the Yellow player which goes second is now a Default Agent. It can take a winning move or block the Red's winning move. While this may seem to be basic moves, this gives the ability to the Default agent to perform rather optimally. Below is the win & loss visualized between a Minimax as Player 1 and a default agent as Player 2. The max depth is set to 5 for below.



There are 10 iterations of 100 games each. As we can observe, the Player 1 now doesn't always win the game. This is because while blocking Player 1 , Player 2 is creating scenarios where it is able to win in the next set of consecutive moves.

Case 2.1: Red plays first ; Red is minimax, Yellow is Default Agent. Max_Depth 6

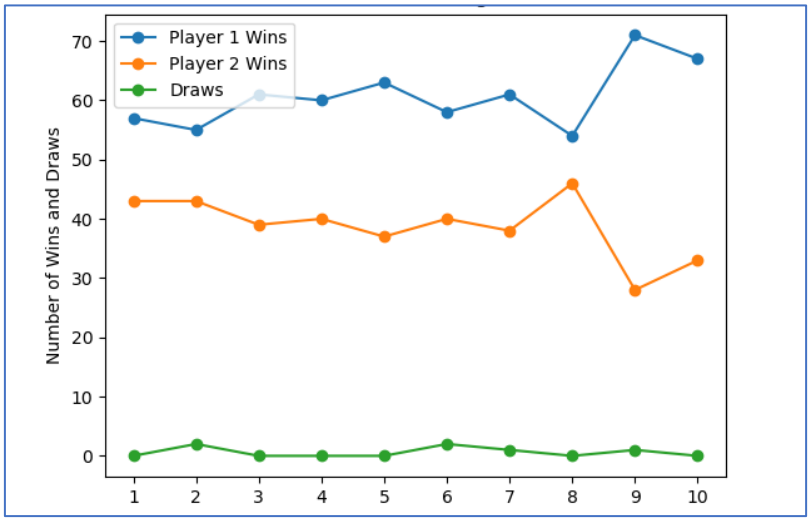
To explore more states, and improve Minimax's performance, I increase the depth. While this improves the performance of Minimax by 30% , it is still able to consistently outperform the default agent with a clear margin.



Case 2.2: Red plays first ; Red is minimax, Yellow is Default Agent. Changing Reward Structure

In this case I change the positive and the negative score received by the agent in case it finds itself in a terminal move state. This reward is much greater than the regular reward structure defined in evaluate_window. The evaluate_window function describes a reward structure for the normal moves, whereas the terminal move reward is highly greater now.

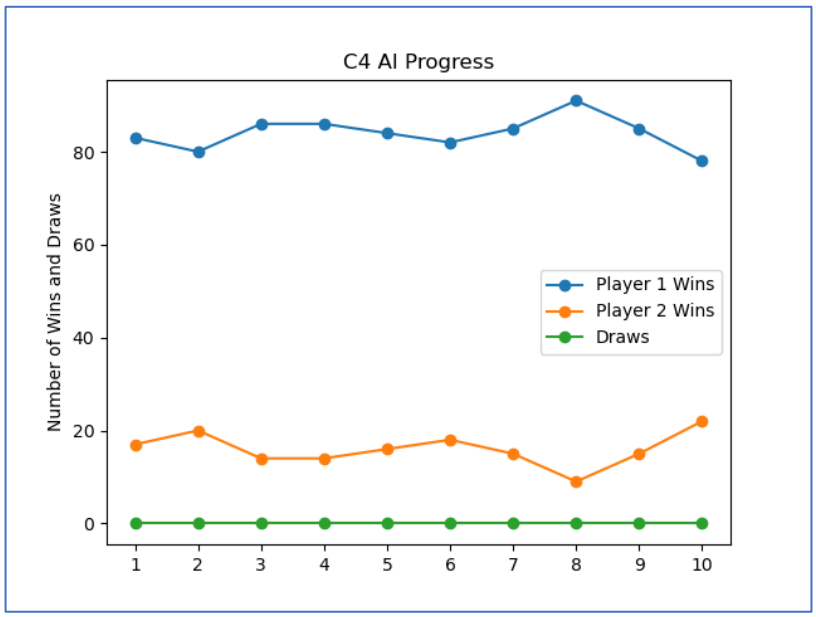
As we can observe, this helps the agent prioritize wins and make much better decisions. With a max_depth limited to 6, this agent with a skewed reward structure performs better than the previous one.



ii) Q Learning Agent

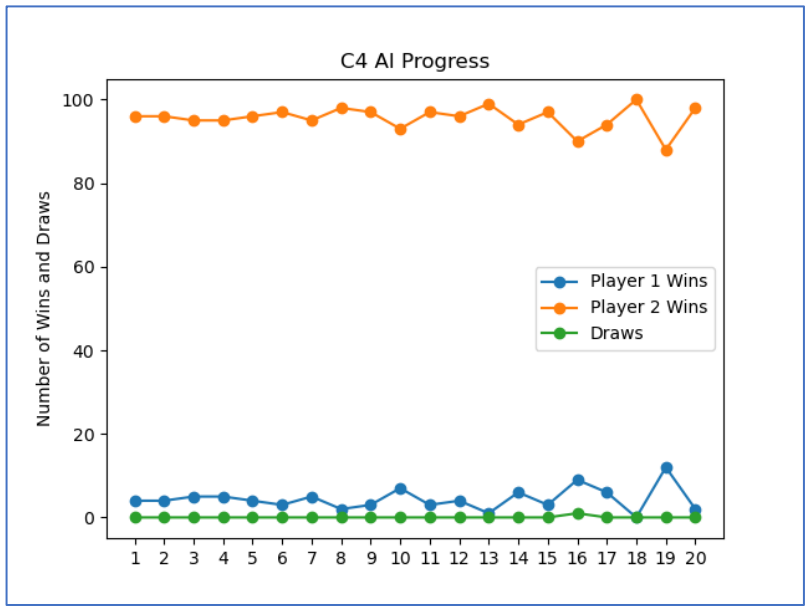
Case 3: Red plays first ; Red is Q learning agent, Yellow is Random Agent.

When Red which is player 1 is a Q Learning agent the game is very different from the one being played in case of tic-tac-toe. Here, keeping a low learning rate around 0.5, a low discount factor of 0.5 and an epsilon of 0.1 made the Agent perform much better than when the learning rate was >0.5. The agent also performed poorly if discount factor was kept below 0.5 or above it. Against a random agent, the Q learning agent is able to perform extremely well with a win rate of >80%.



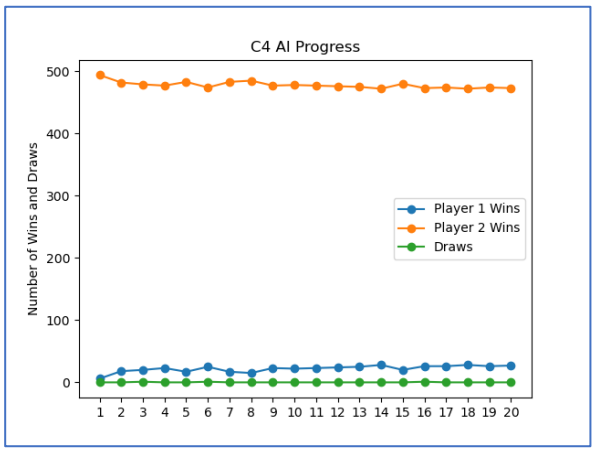
Case 4: Red plays first ; Red is Q learning agent, Yellow is Default Agent.

In case of Q Learning agent, things change drastically once the Agent is able to block the Q Learning agent or make a winning move for itself. I tried changing multiple values for the Q Learning agent but the Default agent does perform optimally. The Q Learning agent performs inconsistently and poorly over 20 iterations of 100 games per iterations against the default opponent.

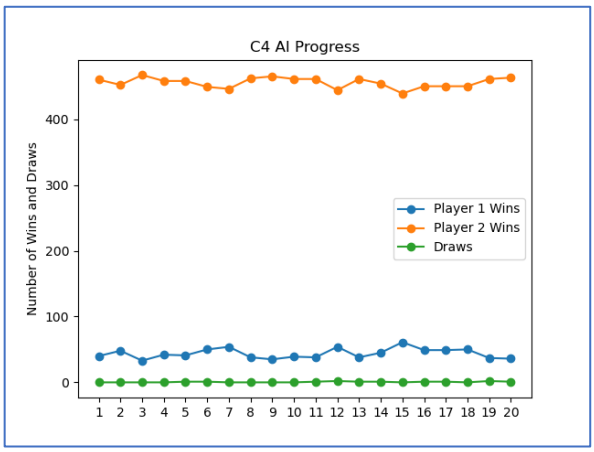


Case 4.1: Red plays first ; Red is Q learning agent, Yellow is Default Agent.

Since increasing the number of games is another valid option to see if that changes the Q Learning Agent's performance, I try that next. The Q learning agent still performs poorly. The agent plays 500 games over 20 iterations and shows a consistent improvement across the iterations, but still fails to beat a default opponent. This default opponent, plays near optimally for the case of connect 4, as it is searching for the next winning move and also blocking the opponent.



alpha = 0.5, gamma=0.5, epsilon=0.1

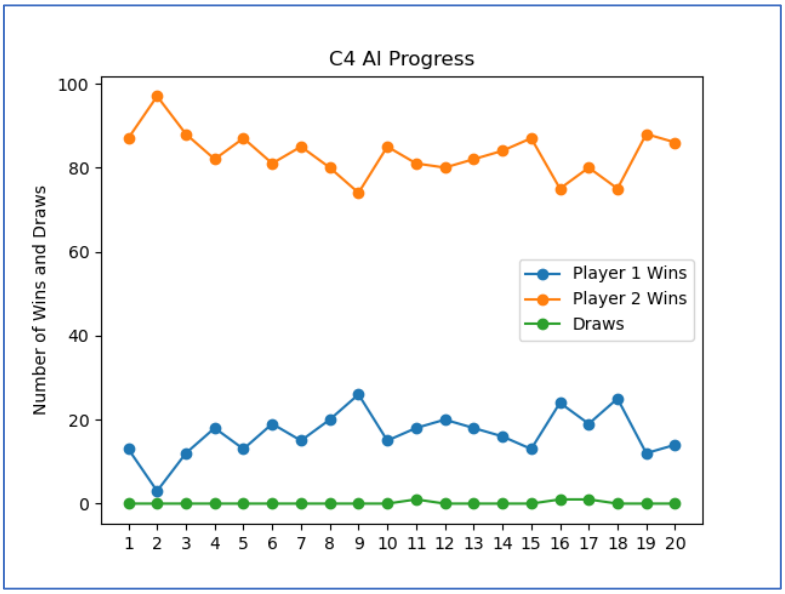


changing alpha=0.4/0.5/0.6, gamma=0.5/0.6/0.7
epsilon=0.1/0.01/0.001

To check another case, I tried changing the value of alpha, gamma and epsilon after certain iterations. While this did improve the performance of the Q learning agent (Right pic), it is still not able to match the default agent.

Case 4.2: Red plays first ; Red is Q learning agent, Yellow is Default Agent. Handicapped Default Agent

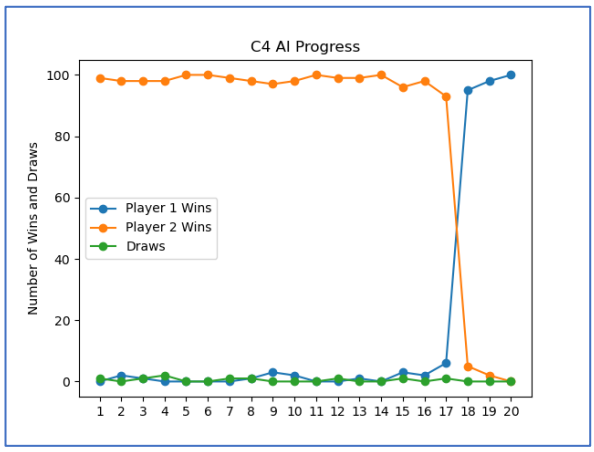
In this scenario I handicap the default agent by commenting the code where it finds a winning move. Now it only blocks the Red Q learning agent's winning move. I limit the number of games over each iteration in this case, but this shows that Q learning agent, is performing better but still unable to match the default agent over 2000 games of connect 4.

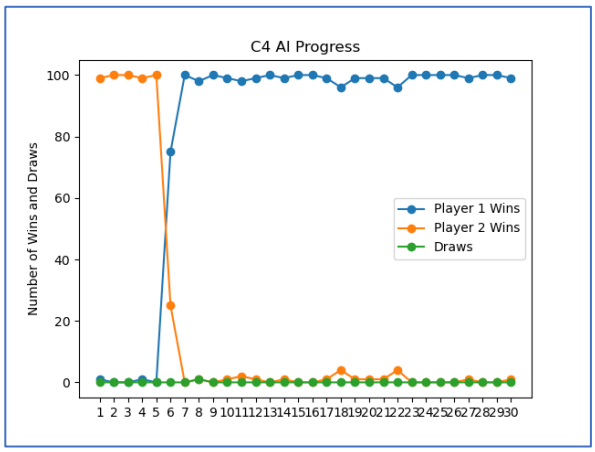


iii) Q Learning Agent vs Q Learning Agent

Case 5: Red plays first ; Red is Q learning agent, Yellow is Q Learning Agent.

Next, I try Q Learning Agent playing as Red vs Q learning Agent playing as Yellow. I run this simulation first with Alpha=0.9, Gamma=0.9 and Epsilon=0.01 to check if the agents can learn from each other's move. Following are the results.

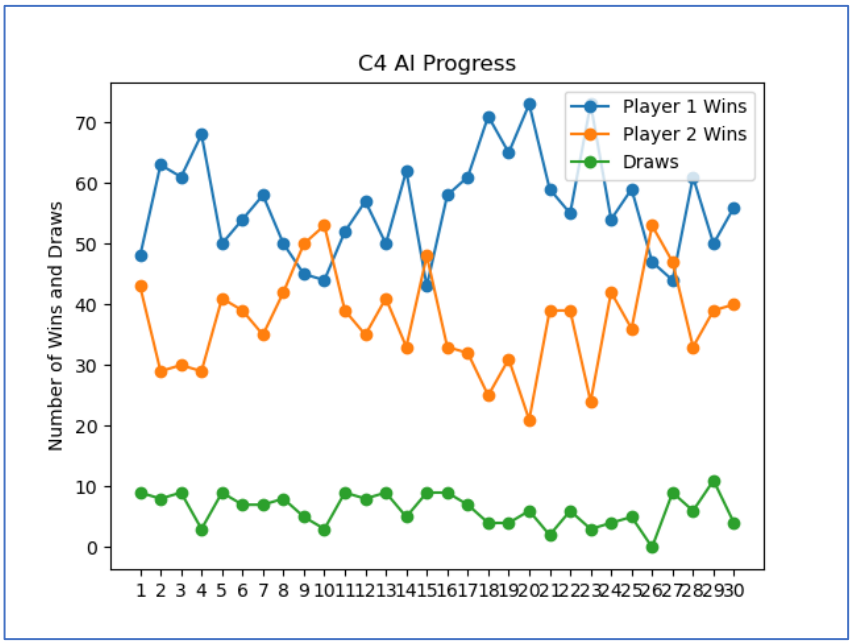




Since the first time it gave a spike for player 1 in the final iterations, I wanted to try it out another time to see if this could be replicated, but the next time, it favours Player 1 earlier on. On an average the Wins are random, and both the players make certain moves on the connect 4 and if the other agent is able to block it then it is ok, else it continues to make that move indefinitely, causing it to win all the rest of the games.

Case 5.2: Red plays first ; Red is Q learning agent, Yellow is Q Learning Agent.

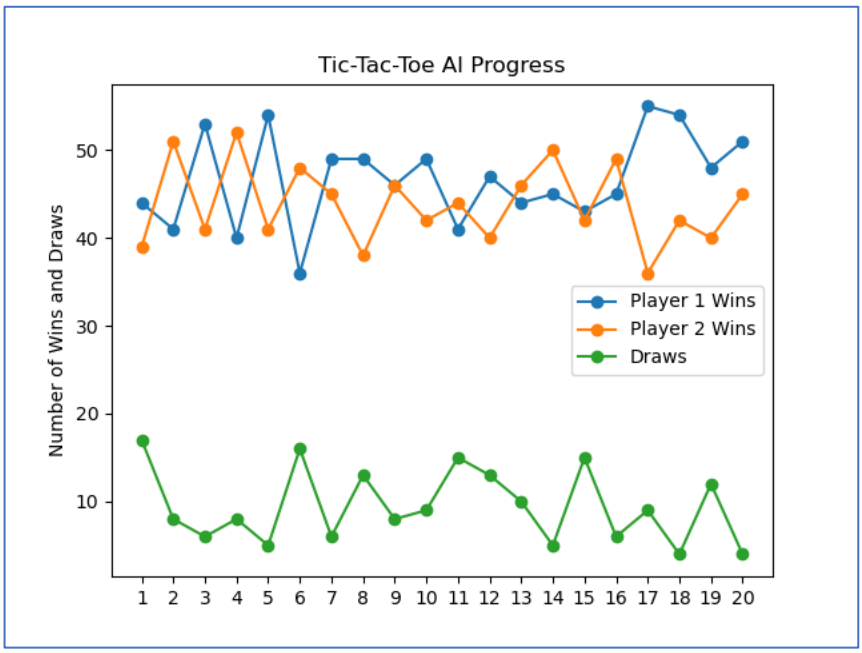
In this case I play Q agent vs Q Agent with values Alpha=0.5, Gamma=0.5 and Epsilon=0.1. With a balanced Learning rate and a balanced discounting factor and a relatively better exploratory value in terms of Epsilon, both the agents perform much better in terms of their gameplay and learning over 30 iterations of 100 games each. Player 1 (Red) since it starts, has an advantage over Player 2. Both the players are also now able to draw each other, while keeping a good game win-loss rate between each other.



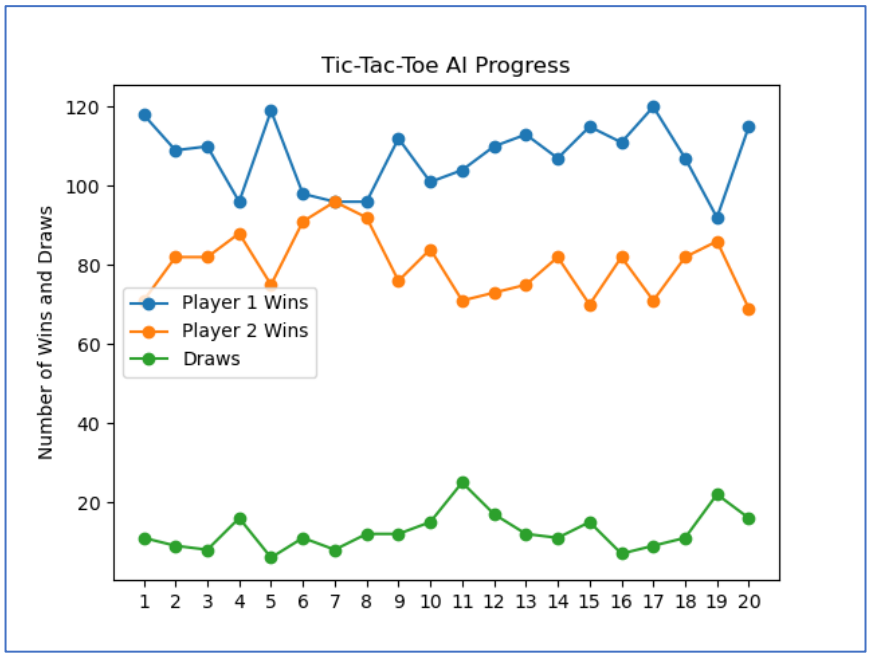
iv) Q Learning Agent vs Minimax

Case 6: Red plays first ; Red is Q learning agent, Yellow is Minimax

This case plays Connect 4 using Q Learning agent with Alpha=0.5, Gamma=0.5, Epsilon=0.1 and a Minimax Agent with max depth =5. It takes more time than usual to play 2000 games over 20 iterations of 100 games each. The Q learning agent is performing really well in this case as compared to the Minimax Agent.



In the below case it plays 200 games, with Alpha=0.5, Gamma=0.6, Epsilon=0.1 and a Max_depth of 6 for the Minimax Agent. In this scenario, A minimal change in the value of Gamma improves the performance of the Q Learning agent as compared to the minimax agent drastically even though minimax is now searching the tree structure till the depth of 6.



Conclusion

There are multiple take aways from these set of experiments.

- Minimax performs better than Q Learning agent in Tic Tac Toe.
- Q learning agent requires a lot of training to get the states and rewards established. The more states it explores the better it is. An augmented version of Q Learning that utilizes the next best action of Q learning along with some static rules like the default opponent's will definitely perform equivalent to or even better than Minimax.

- Minimax becomes comparable in performance to Q Learning when dealing with an exponential game states. In terms of very large game states , Q Learning is computationally much better to implement and faster to respond as compared to calculating depth first states in case of Minimax.

- Hyper-parameters like Epsilon, Gamma and Alpha are supposed to be differently tuned as per the game. For example in case of Connect 4, having a lower value for Gamma and Alpha proved to be better than having a higher value as in the case of tic tac toe. Even an explorative approach in case of connect 4 provided a much better performance from Q Learning agent as compared to the exploitative approach in tic-tac-toe.

- Changing the reward structure has the same effect as hyperparameter tuning. The reward structure defines the agents strategy on a longer run, and can affect the choices the agent makes in bigger games like connect 4.

- Q learning vs Q Learning led to a game play where both agents won equally on average and performed mostly equally. Whereas, Minimax vs Minimax mostly lead to draws and a flat gameplay. Minimax in tic-tac-toe is solved and optimal.

References

- Tic Tac Toe - vanilla repo used - https://github.com/daliborstakic/tictactoe-pygame

- Connect 4 - vanilla repo used - https://github.com/KeithGalli/Connect4-Python/blob/master/connect4.py

1) "Part 2 — The Min Max Algorithm" , https://medium.com/@carsten.friedrich/part-2-the-min-max-algorithm-ae1489509660
2) "Part 3 — Tabular Q Learning, a Tic Tac Toe player that gets better and better" , https://medium.com/@carsten.friedrich/part-3-tabular-q-learning-a-tic-tac-toe-player-that-gets-better-and-better-fa4da4b0892a
3) "An AI agent learns to play tic-tac-toe (part 3): training a Q-learning RL agent", https://towardsdatascience.com/an-ai-agent-learns-to-play-tic-tac-toe-part-3-training-a-q-learning-rl-agent-2871cef2faf0
4) "Why isn't my Q-Learning agent able to play tic-tac-toe?", https://ai.stackexchange.com/questions/10032/why-isnt-my-q-learning-agent-able-to-play-tic-tac-toe
5) "Does Q Learning learn from an opponent playing random moves?" , https://ai.stackexchange.com/questions/20878/does-q-learning-learn-from-an-opponent-playing-random-moves
6) "The Minimax Tic-Tac-Toe algorithm is impossible to beat", https://blogs.cornell.edu/info2040/2022/09/13/56590/#:~:text=The%20Minimax%20Tic%2DTac%2DToe,100%25%20chance%20of%20a%20draw
7) "Solving Tic-Tac-Toe with Minimax", https://www.govindgnair.com/post/solving-tic-tac-toe-with-minimax/
8) Blackboard Slides – Week 5 and 6

Appendix

Minimax TTT

```
def minimax_alpha_beta(game_array, depth, alpha, beta, maximizing_player):
    if depth == max_depth or has_won(game_array, 'x') or has_won(game_array, 'o') or
has_drawn(game_array):
        return evaluate_game_state(game_array)

    if maximizing_player:
        max_eval = -float("inf")
        for move in get_available_moves(game_array):
            i, j = move
            game_array[i][j] = (game_array[i][j][0], game_array[i][j][1], 'o', False)
            eval = minimax_alpha_beta(game_array, depth + 1, alpha, beta, False)
            game_array[i][j] = (game_array[i][j][0], game_array[i][j][1], "", True)
            max_eval = max(max_eval, eval)
            alpha = max(alpha, eval)
            if beta <= alpha:
                break
        return max_eval
    else:
        min_eval = float("inf")
        for move in get_available_moves(game_array):
            i, j = move
            game_array[i][j] = (game_array[i][j][0], game_array[i][j][1], 'x', False)
            eval = minimax_alpha_beta(game_array, depth + 1, alpha, beta, True)
            game_array[i][j] = (game_array[i][j][0], game_array[i][j][1], "", True)
            min_eval = min(min_eval, eval)
            beta = min(beta, eval)
            if beta <= alpha:
                break
        return min_eval
```

```
def minimax_opponent(game_array):
    best_move = None
    best_value = -float("inf")
    alpha = -float("inf")
    beta = float("inf")

    for move in get_available_moves(game_array):
        i, j = move
        game_array[i][j] = (game_array[i][j][0], game_array[i][j][1], 'o', False)
        move_value = minimax_alpha_beta(game_array, 0, alpha, beta, False)
        game_array[i][j] = (game_array[i][j][0], game_array[i][j][1], "", True)

        if move_value > best_value:
            best_value = move_value
            best_move = move

    return best_move
```

```
def evaluate_game_state(game_array):
    if has_won(game_array, 'o'):
```

```
        return 1
    elif has_won(game_array, 'x'):
        return -1
    else:
        return 0
```

Q Learning TTT

```
def update_q_table(prev_state, prev_action, reward, next_state):
    if prev_state not in Q_table:
        Q_table[prev_state] = [1 for _ in range(ROWS * ROWS)]

    if next_state not in Q_table:
        Q_table[next_state] = [1 for _ in range(ROWS * ROWS)]

    prev_q = Q_table[prev_state][prev_action[0] * ROWS + prev_action[1]]
    max_next_q = max(Q_table[next_state])

    Q_table[prev_state][prev_action[0] * ROWS + prev_action[1]] = prev_q + alpha * (reward +
gamma * max_next_q - prev_q)
```

```
def get_valid_actions(game_array):
    valid_actions = []
    for i in range(ROWS):
        for j in range(ROWS):
            if game_array[i][j][2] == '':
                valid_actions.append((i, j))
    return valid_actions

def choose_action(game_array, epsilon):
    global one, two, three
    state = state_to_str(game_array)
    if state not in Q_table:
        Q_table[state] = [1 for _ in range(ROWS * ROWS)]

    valid_actions = get_valid_actions(game_array)
    if not valid_actions:
        # No valid actions available, return a random action
        one+=1
        return random.choice([(i, j) for i in range(ROWS) for j in range(ROWS)])

    if np.random.uniform(0, 1) < epsilon:
        # Choose a random action with probability epsilon
        two+=1
        return valid_actions[np.random.randint(len(valid_actions))]
    else:
        # Choose the action with the highest Q-value
        three+=1
        q_values = Q_table[state]
        max_q = -float("inf")
        best_action = None
        for action in valid_actions:
            i, j = action
            action_q_value = q_values[i * ROWS + j]
            if action_q_value > max_q:
                max_q = action_q_value
```

```
                best_action = action
        return best_action
```

Minimax Connect 4:

```
def minimax(board, depth, alpha, beta, maximizing_player, piece):
    global eval_count
    eval_count += 1
    valid_locations = get_valid_locations(board)
    is_terminal = is_terminal_node(board)
    if depth == 0 or is_terminal:
        if is_terminal:
            if winning_move(board, 1):  # AI player wins
                return (None, 10000)
            elif winning_move(board, 2):  # Opponent player wins
                return (None, -10000)
            else:  # Game is over, no more valid moves (tie)
                return (None, 0)
        else:  # Depth is zero
            return (None, score_position(board, piece))
    if maximizing_player:
        value = -math.inf
        column = choice(valid_locations)
        for col in valid_locations:
            row = get_next_open_row(board, col)
            b_copy = board.copy()
            drop_piece(b_copy, row, col, piece)
            new_score = minimax(b_copy, depth - 1, alpha, beta, False, piece)[1]
            if new_score >= value:
                value = new_score
                column = col
            alpha = max(alpha, value)
            if alpha <= beta:
                break
        return column, value

    else:  # Minimizing player
        value = math.inf
        column = choice(valid_locations)
        for col in valid_locations:
            row = get_next_open_row(board, col)
            b_copy = board.copy()
            drop_piece(b_copy, row, col, 3 - piece)  # Drop opponent's piece
            new_score = minimax(b_copy, depth - 1, alpha, beta, True, piece)[1]
            if new_score <= value:
                value = new_score
                column = col
            beta = min(beta, value)
            if alpha <= beta:
                break
        return column, value
```

Q Learning Connect 4:

```
def q_learning_player(board, Q, epsilon):
    if np.random.random() < epsilon:
```

```
        valid_moves = [c for c in range(COLUMN_COUNT) if is_valid_location(board, c)]
        if len(valid_moves) == 0:
            return None
        return random.choice(valid_moves)

    state = get_board_state(board)
    if state not in Q:
        Q[state] = np.zeros(COLUMN_COUNT)

    valid_moves = [c for c in range(COLUMN_COUNT) if is_valid_location(board, c)]
    if len(valid_moves) == 0:
        return None

    q_values = np.array([Q[state][c] if c in valid_moves else -np.inf for c in
range(COLUMN_COUNT)])
    return np.argmax(q_values)
```

```
    # Update the Q-values based on the final reward
    if winning_move(board, 1):
        reward = 1
    elif winning_move(board, 2):
        reward = -1
    else:
        reward = 0
    update_q_table(Q, states, actions, reward)
```

```
def update_q_table(Q, states, actions, reward):
    for state, action in zip(reversed(states), reversed(actions)):
        if state not in Q:
            Q[state] = np.zeros(COLUMN_COUNT)
        next_state = states[-1] if len(states) > 1 else state
        Q[state][action] = Q[state][action] + ALPHA * (reward + GAMMA * np.max(Q[next_state])
- Q[state][action])
        reward = 0
```

-----END-----