ARK Task Round Documentation

Advance Tasks

Task 4.2 3D Tic Tac Toe Agent

I. Introduction

In this task we had to use minimax algorithm to create an agent to play the 2D tic tac toe game.

So, you had to design a tic tac toe game that would be player vs AI. To do this task I have used the help of minimax algorithm and called it recursively to find out the best possible move.

II. Problem statement

In the first part of the task, you had to create a program that plays 2D tic tac toe game with the user. The AI has to evaluate the best possible move at each step and play accordingly. For example, in chess you play with the laptop or you have chess engines like stockfish that help you analyse the game. Similarly, you had to do to play a 2D tic tac toe game.

In the second part of the question, you had to create a 3D tic tac toe game using gym tic tac toe environment. You had to use minimax algorithm and try to optimise it. You could also use genetic algorithms. The search space would be much larger our here.

III Related Work

Mostly I had tried to use the minimax algorithm, iterations, recursion to solve the 2D tic tac toe problem. I had broken down the code into functions to help check the number of moves on the board, if there were any empty boxes and to check if someone have won. There was another function to find the best possible move using minimax algorithm.

IV Initial Attempts

Initially what I had tried to do was implement the code using an already defined easy matrix, as in the one in which either 'X' or 'O' has already won or is in the position of just winning. Later on, I tried to implement it in smaller defined matrixes.

I really was not able to have a fully functional algorithm.

V. Final Approach

Here is the code for what I had tried to implement.

The code tells us the best possible move at that time and returns 0 if it is a draw, 10 if maximiser wins and –10 if minimizer wins.

The program has several functions to check if there are moves left on the board, to calculate the score, to evaluate who has won the game, to have a move to win, to stop the opponent from winning and to make the first move.

```
AI, opponent = 'x', 'o'

# This function returns true if there are moves
# remaining on the board. It returns false if
# there are no moves left to play.
def isMovesLeft(board) :

    for i in range(3) :
        for j in range(3) :
            if (board[i][j] == '_') :
                return True
    return False

#to make the first move on the board
def startMove(board):
```

```
    count=0
    q=0
    for i in range(3) :
        for j in range(3) :
            if (board[i][j] == '_') :
                count+=1
            if(board[i][j]=='o'):
                q+=1
    if(count==9):
        board[1][1]='X'
    if(count==8):
        if(q==1):
            if(board[1][1]=='_'):
                board[1][1]=='X'
    return board


#to try to make a move to win
def checkMoves(board):
    count=0
    a,b=0
    for i in range(3) :
        for j in range(3) :
            if(board[i][j] == 'x') :
                count+=1
                a=i
                b=j
    if(count)==1:
        if(a>0) and (b>0):
            if(board(a+1)(b)=='_'):
                board(a+1)(b)='x'
            if(board(a)(b+1)=='_'):
```

```
                board(a)(b+1)='x'
        if(a<2) and (b<2):
            if(board(a-1)(b)=='_'):
                board(a-1)(b)='x'
            if(board(a)(b-1)=='_'):
                board(a)(b-1)='x'
    return board


#to stop the opponent from winning
def stopopponent(board):
    for i in range(1) :
        for j in range(1) :
            if (board[i][j] == 'o') :
                if(board[i+1][j])=='o':
                    board[i+2][j]=='x'
                if(board[i][j+1])=='o':
                    board[i][j+2]=='x'
                if(board[i+1][j+1])=='o':
                    board[i+2][j+2]=='x'
    for i in range(1,2) :
        for j in range(1,2) :
            if (board[i][j] == 'o') :
                if(board[i+1][j])=='o':
                    board[i-1][j]=='x'
                if(board[i][j+1])=='o':
                    board[i][j-1]=='x'
                if(board[i+1][j+1])=='o':
                    board[i-1][j-1]=='x'
    return board
```

```
# This is the evaluation function
def evaluate(b) :

    # Checking for Rows for X or O victory.
    for row in range(3) :
        if (b[row][0] == b[row][1] and b[row][1] == b[row][2]) :
            if (b[row][0] == AI) :
                return 10
            elif (b[row][0] == opponent) :
                return -10

    # Checking for Columns for X or O victory.
    for col in range(3) :

        if (b[0][col] == b[1][col] and b[1][col] == b[2][col]) :

            if (b[0][col] == AI) :
                return 10
            elif (b[0][col] == opponent) :
                return -10
```

```
    # Checking for Diagonals for X or O victory.
    if (b[0][0] == b[1][1] and b[1][1] == b[2][2]) :

        if (b[0][0] == AI) :
            return 10
        elif (b[0][0] == opponent) :
            return -10

    if (b[0][2] == b[1][1] and b[1][1] == b[2][0]) :

        if (b[0][2] == AI) :
            return 10
        elif (b[0][2] == opponent) :
            return -10

    # Else if none of them have won then return 0
    return 0

# This is the minimax function. It considers all
# the possible ways the game can go and returns
# the value of the board
def minimax(board, depth, isMax) :
    score = evaluate(board)

    # If Maximizer has won the game return his/her
    # evaluated score
    if (score == 10) :
        return score

    # If Minimizer has won the game return his/her
    # evaluated score
```

```
if (score == -10) :
    return score

# If there are no more moves and no winner then
# it is a tie
if (isMovesLeft(board) == False) :
    return 0

# If this maximizer's move
if (isMax) :
    best = -1000

    # Traverse all cells
    for i in range(3) :
        for j in range(3) :

            # Check if cell is empty
            if (board[i][j]=='_') :

                # Make the move
                board[i][j] = AI

                # Call minimax recursively and choose
                # the maximum value
                best = max( best, minimax(board,
                                          depth + 1,
                                          not isMax) )

                # Undo the move
                board[i][j] = '_'
    return best
```

```
    # If this minimizer's move
    else :
        best = 1000

        # Traverse all cells
        for i in range(3) :
            for j in range(3) :

                # Check if cell is empty
                if (board[i][j] == '_') :

                    # Make the move
                    board[i][j] = opponent

                    # Call minimax recursively and choose
                    # the minimum value
                    best = min(best, minimax(board, depth + 1, not isMax))

                    # Undo the move
                    board[i][j] = '_'
        return best

# This will return the best possible move for the player
def findBestMove(board) :
    bestVal = -1000
    bestMove = (-1, -1)

    # Traverse all cells, evaluate minimax function for
    # all empty cells. And return the cell with optimal
    # value.
```

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

            # Check if cell is empty
            if (board[i][j] == '_') :

                # Make the move
                board[i][j] = AI

                # compute evaluation function for this
                # move.
                moveVal = minimax(board, 0, False)

                # Undo the move
                board[i][j] = '_'

                # If the value of the current move is
                # more than the best value, then update
                # best/
                if (moveVal > bestVal) :
                    bestMove = (i, j)
                    bestVal = moveVal

    print("The value of the best Move is :", bestVal)
    print()
    return bestMove
# Driver code
board = [
    [ 'x', 'o', 'x' ],
    [ 'o', 'o', '_' ],
    [ '_', '_', '_' ]
```

```
]

bestMove = findBestMove(board)
stopopponent(board)
checkMoves(board)
startMove(board)
board[bestMove[0]][bestMove[1]]='X'
print(board)
print("The Optimal Move is :")
print("ROW:", bestMove[0], " COL:", bestMove[1])
```

VI. Results and Observation

The algorithm that I have tried to use is not 100% accurate and still has to improve on. Like for example when you could use a trick like to get double wins in the next play whatever the opponent plays, is something that I could not implement. Also, if there are a smaller number of entries the code goes a bit haywy and does not display the correct output.

VII. Future Work

The code could be worked upon several areas like firstly upon how to play trick moves, or how to play your moves in such a way that you do not fall into an opponent's trap. It also be improved upon how to decide what to play as the first move depending upon what the minimizer plays as the first move.

Conclusion

The overall problem was about implementing minimax and genetic algorithm, using gym tic tac toe environment and about creating an algorithm to have a 2D tic tac toe and a 3D tic tac toe player vs AI.

We could use apply this in machine learning, like how different people play and apply logic. It could also be made advanced and used to have answers for solving different logical problems like about what to do when given a choice. Also, in algorithms like A* if we want to choose heuristics where a lot of choices are given, we could implement these programs. Like for example if a drone encounters an obstacle in mid-air, then this algorithm could be used to find the most optimised path, as in it fixes its destination and then implements backtracking to find the best possible path. Or in competitions between robots where who is able to optimize more wins.