Chess AI with MiniMax and Alpha/Beta pruning

Jordan Rudnick

for CS 490, Machine learning

Introduction

Chess is one of the oldest games still played by mankind, dating back to 6th century india. In spite of its age and due to it's complexity, chess has not yet been solved. In the last few years, we have gotten to the point where AI can outplay humans, but as an average player, I would love to be able to face an AI, but have no interest in trying to play the likes of Stockfish, alphazero, or deep blue.
I also find myself in the position of not having very much computing power, and certainly not aenough time to train a proper neural network through games, and feeding it games requires memory and runs the risk of making it too powerful for me to go up against.
The purpose of this project was to produce an AI that could play chess well, but not overhwelmingly so with as few resources as possible..

Initial requirements

Unlike most AI, the MINIMAx algorythm has very minimal requirements. It's only needs are the rules of the game, a list of legal moves, and some values by which to evaluate decisions. Someone was kind enough o make a chess library with the rules built in to the game, including a list of legal moves given a board position, leaving values. The standard ratio of values of chess pieces is that pawns are 1, knights and bishops are 3, rooks are 5, queens are 9, and kings don't normally have a score because why bother counting how far ahead or behind you are if you already won?
In this case, I took the normal ratios and multiplied them by 10, and valued the king as 1170, which is 3xs the value of all other pieces combined.

MiniMax algorithm

On it's own, the minimax algorithm is pretty simple, it assumes both players are ratinoal and allows the AI to look a certain number of turns ahead, for any given board position, the score is valued at the total value of white pieces – the total value of black pieces. It analyzes a decision tree and assumes white will attempt to have the highest score after the number of turns it looks ahead and black will want the lowest. Alone, the minimax algorithm gets expensive quickly and it's almost unplayable with just 4 moves, but that's where alpha/beta pruning comes in.

pruning

Without pruning, a minimax tree has 197 thousand openings to analyze for the first two turns, but by storing a single value at each level of depth, a lot of work can be avoided. The Alpha value is the best option for the maximizer(white), and beta is the best for black along a certain path from the root. If white sees a beta value on an unexplored branch, but it knows that the beta value of the other side of that branch is higher, exploring other options is pointless, because it already knows it's best choice, so rather than wasting it's time, it just moves on to either another sister node, or move back up to a sister of a higher level

code

```
def minimaxRoot(depth, board,isMaximizing):
    possibleMoves = board.legal_moves
    bestMove = -9999
    secondBest = -9999
    thirdBest = -9999
    bestMoveFinal = None
    for x in possibleMoves:
        move = chess.Move.from_uci(str(x))
        board.push(move)
        value = max(bestMove, minimax(depth - 1, board, not isM
        board.pop()
        if( value > bestMove):
            print("Best score: " ,str(bestMove))
            print("Best move: ",str(bestMoveFinal))
            print("Second best: ", str(secondBest))
            thirdBest = secondBest
            secondBest = bestMove
            bestMove = value
            bestMoveFinal = move
    return bestMoveFinal

def minimax(depth, board, is_maximizing):
    if(depth == 0):
        return -evaluation(board)
    possibleMoves = board.legal_moves
    if(is_maximizing):
        bestMove = -9999
        for x in possibleMoves:
            move = chess.Move.from_uci(str(x))
            board.push(move)
            bestMove = max(bestMove,minimax(depth - 1, board, n
            board.pop()
        return bestMove
    else:
        bestMove = 9999
        for x in possibleMoves:
            move = chess.Move.from_uci(str(x))
            board.push(move)
            bestMove = min(bestMove, minimax(depth - 1, board,
            board.pop()
        return bestMove

def evaluation(board):
    i = 0
    evaluation = 0
    x = True
    try:
        x = bool(board.piece_at(i).color)
    except AttributeError as e:
        x = x
    while i < 63:
        i += 1
        evaluation = evaluation + (getPieceValue(str(board.piec
    return evaluation


def getPieceValue(piece):
    if(piece == None):
        return 0
    value = 0
```