GOAL: To explore how machine learning works and the Markov decision process by creating an unbeatable checkers gaming agent (which I refer to as a bot, since it takes too long to say that). I originally was a little worried since checkers is such a simple game I would have a hard time getting the bot to a level that it could beat me. Luckily, I drastically overestimated my checkers-playing ability though.

RELEVANCE: With the explosion in AI technologies and potential for a true AGI being developed sometime in the next few decades, barring another AI Winter, it seems AI is fast becoming an integral part of every aspect of computer science. I think having a complex understanding of, and an ability to work with AI seems paramount to a successful career in any computer science related field. With that in mind, I wanted to start to understand this technology and figured the best place to start would be to start with machine learning. Inspired by IBM's Deep Blue defeating chess world champion Garry Kasparov in 1997, this project is the beginnings of an homage to that milestone.

HOW IT WORKS: To start, I figured the best way to go about it was to create a class for each object used in a game of checkers. So, I knew I'd need a class representation of the board, the pieces, the actual rules of the game, and the bot player to play against.

1. Board: keep track of the position of the pieces as the game was played
    a. Used a 2-D Array for the board. So, it's essentially a list of 8 elements, 1 for each row of a checkers board.

i. Each element of the list (or row) would be an additional list of 8 elements. Each element of this sub-list was representative of one of the 8 squares on that row.

ii. **[[0, P, 0, P, 0, P, 0, P],**
[P, 0, P, 0, P, 0, P, 0],
[0, P, 0, P, 0, P, 0, P],
[0, 0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0, 0],
[P, 0, P, 0, P, 0, P, 0],
[0, P, 0, P, 0, P, 0, P],
[P, 0, P, 0, P, 0, P, 0]]

iii. P = an instance of the Piece class and 0 = an empty square

iv. I ended up adding some additional attributes the board object tracks which are used when calculating what the best move for the bot to make will be based on the current board state such as...
    1. List of pieces still on the board for each color
    2. Number of captured pieces for each color
    3. Number of king pieces for each color

2. Piece: an object contained in the board array representing the pieces
    a. Each piece had instanced attributes for...
        i. Row: current row on the board the piece is located
        ii. Col: current column on the piece is located

iii. Color: tracked which team/color the piece belonged to
iv. King: Boolean tracking whether a piece had been kinged
    1. Initially, I forgot about this aspect of checkers and once I was reminded of them, they made things much more difficult…
v. X: x coordinate to draw the piece onto the surface
vi. Y: y coordinate to draw the piece onto the surface

3. Game: handles the actual game logic and rules of the game, as well as calling the appropriate methods drawing/updating the board and pieces as needed.
    a. Its instanced attributes are…
        i. Selected: stores a Piece object once it has been selected, which is then passed to a function finding all valid moves that piece could make based on the current board state. The valid moves are then drawn onto the board to show where the selected piece can be moved
        ii. Board: stores a Board object of the current board state, which is updated as moves are made
        iii. Turn: stores the color who's turn it is
        iv. Valid Moves: stores all valid moves a piece can make once one has been selected, as well as the pieces which would be jumped for each of those moves
        v. Turn-count: stores the total number of turns taken, and is used when reversing a previously taken move

vi. Board List: stores copies of the Board object at the start of each turn so that a previous move can be undone, thus returning to a previous board state

4. Bot: object representing the player a user is playing against.
    a. Its instanced attributes are...
        i. Color: stores the color the bot is playing as
        ii. Game: stores the game object the bot is playing in
        iii. Depth: stores the depth the recursive move selecting algorithm should check to. In other words, how many turns ahead the bot should evaluate moves.
            1. While increasing this number makes the bot much more difficult to defeat, each additional turn ahead it checks exponentially increases the number of board states that must be evaluated to determine the best move.
        iv. Max: Boolean value determining whether the bot should choose the move with the highest score (i.e. maximize its score) or choose the lower score (i.e. minimize its score)
            1. The algorithm used to find the best move is a minimax algorithm, which I'll explain in more detail further down.

Valid Moves Algorithm: This was the algorithm I used to find all possible valid moves for a particular piece.

Start by passing the selected piece's color, row and column as cur_row and start_col, using a step_size=1 and an empty list of jumped pieces.

1. Is the selected piece a king?

YES:

1. **Go to step 2, passing red as color.**
2. **Go to step 2, passing black as color.**

   ****both directions need to be checked****

NO:

Go to step 2, passing selected_piece's color as the color.

2. Is the passed color red or black?

RED: (moves from bottom to top, thus going from a higher index row to a lower index.)

next_row = cur_row - 1

cur_col_left = start_col - step_size

cur_col_right = start_col + step_size

BLACK: (moves from top to bottom, thus going from a lower index row to a higher index.)

next_row = cur_row +1

cur_col = start_col + step_size

cur_col = start_col - step_size

3. Is square at cur_col_left empty?

YES:

Is there a potentially jumped piece?

YES:

Add potentially jumped piece to list of jumped pieces since it is a valid jump. Add the move to list of valid moves with list of jumped pieces associated with it.

step_size += 1,

cur_row = next_row

repeat from step 1 using new step/row values to check for an additional jump (keeping the list of jumped pieces in case one is found)

NO:

Is list of jumped pieces empty?

YES:

Add move to list of valid moves with empty list of jumped pieces associated with it. (no pieces jumped if move selected)

NO:

There is no additional jump to make on this side.

NO:

Is there a potentially jumped piece?

YES:

There is no valid jump move nor any need to check for additional jumps on this side.

NO:

1. **store the piece occupying square as temp_piece**

2. **Is selected_piece color == temp_piece color?**

YES:

Then there is no valid move on this side and no need to check for additional jump moves.

NO:

1. ***store temp_piece as potentially jumped piece***
2. **step_size += 1**
3. **cur_row = next_row**
4. **Repeat from step 1 using new row/step/potentially jumped piece to check for a possible jump**

4. Is square at cur_col to the right empty?

YES:

Is there a potentially jumped piece?

YES:

Add potentially jumped piece to list of jumped pieces since it is a valid jump. Add the move to list of valid moves with list of jumped pieces associated with it.

step_size += 1,

cur_row = next_row

repeat from step 1 using new step/row values to check for an additional jump (keeping the list of jumped pieces in case 1 is found)

NO:

Is list of jumped pieces empty?

YES:

Add move to list of valid moves with empty list of jumped pieces associated with it. (no pieces jumped if move selected)

NO:

There is no additional jump to make on this side.

NO:

Is there a potentially jumped piece?

YES:

There is no valid jump move nor any need to check for additional jumps on this side.

NO:

1. ***store the piece occupying square as temp_piece.***
2. **Is selected_piece color == temp_piece color?**

YES:

There is no valid move on this side and no need to check for additional jump moves.

NO:

1. ***store temp_piece as potentially jumped piece***
2. **step_size += 1**
3. **cur_row = next_row**
5. **Repeat from step 1 using new row/step/potentially jumped piece to check for a possible jump**

RETURN LIST OF VALID MOVES AFTER ALL RECURSIVE CALLS COMPLETED

The basic process when the bot takes its turn is…

1. **For every piece that is the bot's color, a list is made of all the valid moves that piece could make.**
2. **For each of those moves, the resulting Board state is stored.**
3. **Then for each of those board states, for each piece on the board a list of all possible moves the opposing player could take in response is created.**
4. **For each move by each of those pieces, the resulting board states are again stored…**
    i. **This process is repeated until the pre-set depth is reached. (This is why each move that is looked ahead exponentially increases the amount of computational power required)**
5. **Each of the resulting board states at the set depth is passed to the evaluation method and given a score. The score is calculated separately for each color based on things such as the number of captured pieces each color has, number of kings, the position of each color's pieces, etc. Then red's score is subtracted from black's score to produce an overall score of the state the board would be in as a result of the potential moves creating it. Therefore, a high score is favorable to black, while a low score is favorable to red.**
    i. **Playing as black, the bot will always choose the board worth the most points and will assume that the opposing team will take the most optimal move and choose the board state worth the fewest points. So, for example if it is simulating red's turn when the max depth is reached, for each board state resulting from a potential move black might make, each potential move is assigned the lowest score of the possible outcomes from that potential move.**
    ii. **The bot then simulates its turn and compares each score assigned to each potential move and picks the highest score for each. It then again assumes that from those scores red will pick the lowest, and so on, until reaching the level containing the possible board states which could result from possible moves the bot could make during this turn.**
6. **The bot then compares the scores of the resulting board state for each potential move and selects the board state with the highest remaining score. Whatever potential move had led to that board state is determined to be optimal and is then chosen by the bot.**
7. **The bot then ends its turn and awaits the opposing player to take its turn, and the process starts all over again.**

Evaluation Method:

Score = 10*([# of Pieces captured by black] – [# of Pieces captured by red]) + 7*([# of black kings] – [# of red kings]) + [position of black's pieces on the board] – [position of red's pieces on the board]

Position Score is the sum of each individual piece's score for each color.

- **For pawns:**
  - **the further a piece gets towards the opposite end of the board, the greater the increase in score.**
- **For kings:**
  - **score is only increased if they are in the 4 center-most rows on the board.**

  My thought process was that since they can move in 4 directions, the closer to the center of the board they are the stronger the advantage they provide and the more effective they become at either preventing the opponent's pawns from also becoming kings or capturing opposing pieces. I had originally implemented this based on the viability of the valid moves available to each color based on the board state but it increased the time the bot spent evaluating the board and deciding on the optimal move so drastically that I felt that time would be better spent checking an additional move ahead, so I decided to try this instead. I was winning roughly 1 in 5 games once I made that change, but due to a coding mistake I made the positional scores weren't evaluated properly. Since discovering and fixing that bug, I haven't won against the bot a single time. I think there is still plenty of room to improve the evaluation method, but I'm not sure how to test if they are more effective than the current setup when I can't win against the bot to begin with…

Future development: There are so many ways this project can be improved as time progresses it would be impossible to list them all, however there are quite a few things I had initially hoped to include that there just wasn't time to complete.

- 'Pruning' of the decision tree through some method to make finding the optimal move more efficient. This could lead to various other improvements, such as…
    - The decision tree depth can then be increased without a drop in performance
    - The user would also be able to use the same algorithm to find their own optimal move to make based on the board state.
- Creation of another algorithm within the bot class that stores every decision during a game and could optimize the evaluation method based on previous games. This would allow it to 'learn' from any losses to ensure that the decisions which led to the loss are not repeated.
    - The bot is created in such a way that allows for it to play another instance of itself and play multiple games simultaneously, however without the ability to learn from previous games, each time I tried it both sides would eventually just move their pieces around the board preventing the other side from capturing them and so it was just an infinite game of checkers. I'd like to be able to run multiple game simulations simultaneously, allowing the bot to learn from the decisions it is making against itself in real time until it eventually reaches a point that the bot is able to defeat itself.
- The valid move finding algorithm itself feels very unoptimized, and while I have some other ideas on how this can be improved, time just did not allow me to get them written and tested before the deadline.
- The GUI can be drastically improved by adding…
    - The simplest thing I ran out of time before implementing is just a circle being drawn around the piece currently selected by the user indicating its selection.
    - Additional trackers besides just the number of captures, such as number of kings, the color's turn it is, the turn number, and other such QoL improvements.
    - At some point I'd also like to possibly even add a probability display which updates as the game unfolds displaying the probability of either side winning based on the current board state.
        - This would also be useful for learning reinforcement of the bot's decisions by rewarding moves which increase the probability and punishing moves that would decrease it.

LESSONS:

- The first lesson I learned was to make sure to have a clear understanding of the rules of an activity before attempting to design a system to automate it. I originally had not remembered that a checkers pawn was kinged when it reaches the opposite end of the board until a friend reminded me, and that once a piece was kinged it could then travel both forwards and backwards. This caused me to have to rewrite a lot of code that easily could have been avoided if I had spent any time researching the rules of checkers instead of just thinking I remembered everything.
- I always knew it was best practice to use relative dimensions for things rather than hardcoding them, but never really gave much thought to just how important it is until I went to add the GUI to the project. Even using relative measurements when coding this project, I still had to make adjustments to the math once the game board was no longer spanning the entire window and can only imagine how much extra work would have been required had I not.
- The most interesting things I learned though were related to attempting to think at a higher level about what makes one board state better than another and then how to express that mathematically. It's amazing that something which seems so simple actually has so many factors to consider when trying to find a way to objectively compare the qualities of one state to another.