Creating Artificial Intelligence to Play Chess

John Bannan & Arpita Das
Email: john.bannan@temple.edu
Email: arpita.das.cuet@gmail.com

A. ABSTRACT

This project is focused on creating an artificial intelligence (AI) system to play Chess using decision rule and deep neural network algorithm. Two approaches are used in this paper: one utilizing just the minimax algorithm and another using a Convolutional Neural Network (CNN) in conjugation with a minimax algorithm. The environment is created to play against AI itself and for humans too. The performance is tested against Stockfish to compare its moves. Our model could not beat Stockfish, however, was able to stick up to 20 moves against Stockfish. As CNN is introduced with mini-max, the moves got more tactical when tested for AI vs AI.

B. INTRODUCTION

The game of chess has been not only a popular game for centuries but an extensively studied topic, particularly in computer science. Humans have been seeking a way to automate or simulate human play of chess for a long time. For example, the Mechanical Turk developed by Wolfgang von Kempelen, which was an 18th century chess playing machine that wowed the crowds at the time. Although the Mechanical Turk was simply a trick that actually used a human to play the game, we have come far in developing ways to play chess using computers and machines. Deep Blue, developed by IBM, was able to defeat the world chess champion, Garry Kasaparov, stunning the world in a historical development for AI chess [15, 16]. But, while computers can beat humanity's top players, it should be noted that humans are still better at calculating moves more efficiently. Computers and AI chess algorthim's use brute force in order to find the optimal move, many moves which humans would never even consider and would consider as a poor move. Typically, chess AI's use a Minimax algorithm in their engines to some degree. In this report, we will discuss the algorithm to create a chess AI as well as its implementation and results. We have also attempted to use machine learning to aid in our chess AI.

C. BACKGROUND

It is important to understand the rules and guidelines of Chess before an AI or any algorithm can be implemented. The main goal of chess is to capture your opponent's King, thus ending the game in a "check-mate". Chess itself is played on a 8 x 8 grid (64 squares) based board with 16 white and black pieces, respectively. The chess pieces each consist of 6 different pieces: pawn, rook, knight, bishop, king, and queen [1]. The starting positions for each piece can be seen below in Figure 1. Each type of piece has its own rule set for how it can move and capture pieces. The pawn can only move one forward space at a time, except during the first turn it is moved where it can move 2 spaces, and can capture pieces that are diagonal to its space. Rooks move and capture pieces horizontally or vertically and can freely move backward or forward. Knights can freely move and capture pieces in an "L" shaped pattern. The bishop can move and capture pieces diagonally and can freely move backward or forward, thus the bishop is similar to the rook, but instead of moving laterally, it moves diagonally.. The king can move one space in any direction freely. The queen can move in any direction freely, making it like a king but with more movement [1, 2]. An example of each piece's movement is given below in Figure 2.

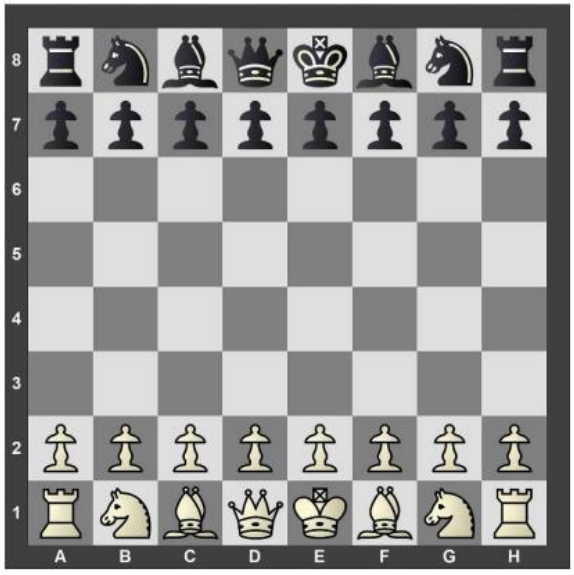


Figure 1. Initialization of Chess Board [1]

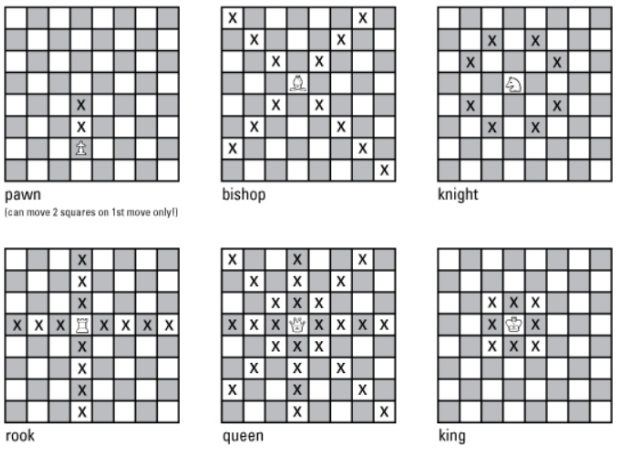


Figure 2. Piece Movement [2]

D. METHODOLOGY - MINIMAX ALGORITHM

I. Evaluation of Board and Evaluation Function

In order to implement chess into a software program, we have to first create a way to represent the board and board state in a numerical way. This also involves the logic and rules of chess as well. For our implementation the rules and logic given in [1] and in the background section have been given by the Python Chess library [3]. The Python Chess library will allow us to check the game state as well as give us vital piece information. However, the library does not give us a numerical way to evaluate the board state. For example, given a board state, it may be hard to determine in some cases, whether white or black has an

advantage. We can do this through Piece- Square Tables. A Piece Square Table is simply an 8 x 8 table or matrix of numerical values for each piece and position on the chess board. An example of a Piece square table can be seen below in Figure 3.

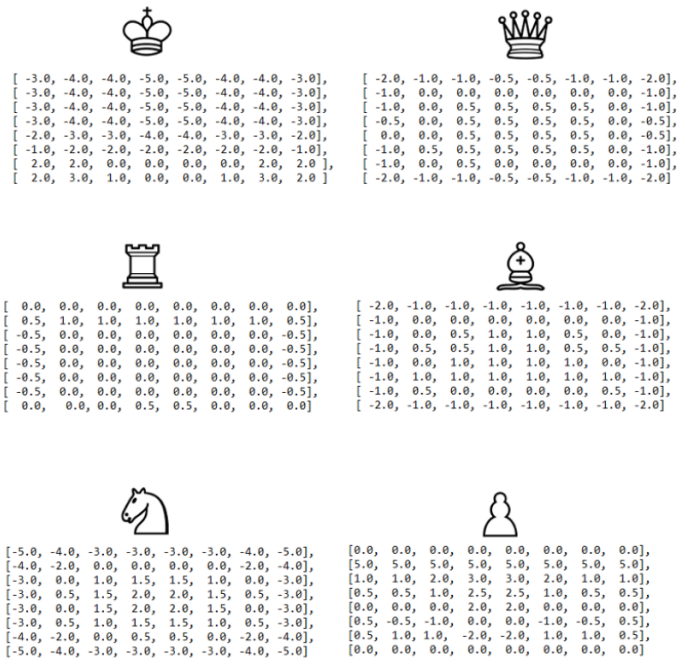


Figure 3. Piece Square Tables

If we look at figure 3, we can see that the knight, indicated by the horse symbol, has negative values in its table on the edges of the game board. This is because the knight is less mobile, or can move in less positions on the board while on the edge, as opposed to the middle. Another example is the king, whose first two rows are positive numbers, but anything beyond that is negative. This is due to the fact that the king generally does not leave the first two rows and other positions may be disadvantageous. The values assigned to the Piece Square Tables in Figure 3 and our implementation are based on mobility and similar logic as given by the previous two examples. However, not all Piece Square Tables are the same and in fact some programs even update the tables as games are played. For our purposes, the tables are static and do not change. Now that we have a way to evaluate pieces and their positions on the board, we have to evaluate the overall state of the board and which player has an advantage [4, 14].

An evaluation function is used to determine whether a player is in an advantageous position or not. In this case the higher the evaluation function value, the better the white player is doing, and vice versa for the black player. This function can be seen in Eqn 1.

$$Eval = Material\ Score + PawnScore + RookScore + KnightScore + BishopScore + QueenScore \quad \textbf{Eqn 1.}$$

Equation 1, returns the summation of the material score and the scores for each piece. The material score is given as the summation of each piece's weight multiplied by the difference between that piece for each player [5, 6]. The weights of the chess pieces are traditionally given values or a multiple of [1, 3, 3, 5, 9] for pawn, knight, bishop, rook, and queen respectively [6]. The king does not have a weight, because capturing the king will end the game and thus it is not considered in calculations. The scores of the pieces, for example the PawnScore, are given as the summation of said pieces' square table values at those locations in the games for both players. The evaluation function will return -9999, or 9999 when checkmate occurs

for a white or black win respectively.

II. Min Max Algorithm and Alpha Beta Pruning

Now that an evaluation function has been established, we can now use a minimax algorithm to determine movement selection for players and establish an AI. The minimax algorithm is an algorithm that is typically used in two player games such as checkers, chess, or Go. The algorithm seeks to minimize the loss of each player [6,7,8]. The minimax algorithm works backwards to determine the best choice for each player. An example can be seen below in Figure 4

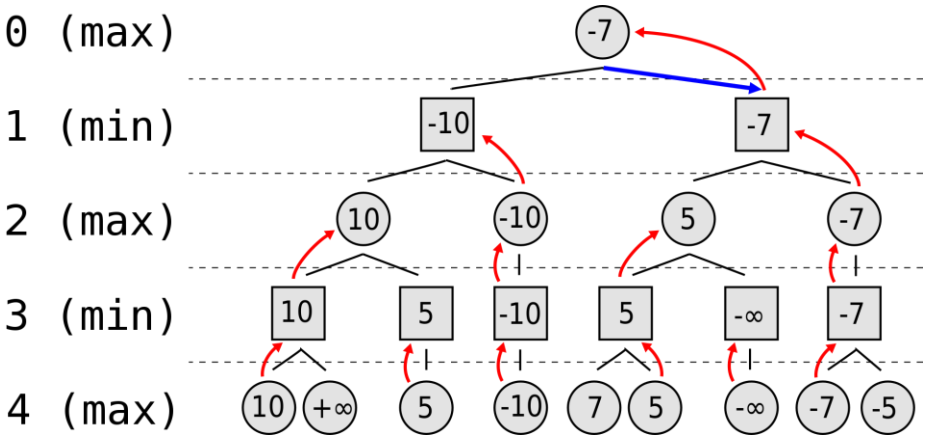


Figure 4. Example of Minimax Algorithm

In the above example, there are two players (one maximizes the score, the other minimizes the score) and each are allowed to see four moves into the future. The tree in Figure 4 represents the moves done by each player, in which -infinity results in the min player winning and nodes with + infinity show the max player winning. The other node values are determined by a heuristic function. Working backward from level three, we can see the minimum value of each node is selected from level 4. Then at level 2, the player will choose the maximum value, and so on until the starting node is reached. Once the starting node is reached, the maximizing player will pick the largest node, in this case -7. In chess, we use a negamax variation of the minimax algorithm, which just makes the loss and gain of each player a zero sum game. Thus a loss of -100 for white, is a gain of +100 for black, and thus since a gain for one player is a loss for another, only one evaluation value has to be stored and passed [10, 14]. In chess each node thus represents a move and how its move is determined to be advantageous or not is determined through the evaluation function.

For a game like chess, it becomes increasingly difficult to determine the best move to choose considering the immense amount of outcomes and choices given to a player. This would result in our decision tree for each move having many nodes, and even more if we want to consider further into the future, say 10 moves in the future for example. This in turn results in greater computational cost and time for our algorithm if we intend to search every node. We can get around this issue using Alpha Beta Pruning. Alpha Beta Pruning passes down pruning information in terms of values, α and β, that are used to prune nodes on the decision tree [11, 14, 16]. We use the α value to prune minimum nodes and α is set to the highest value for all of its max ancestors of the current node. Any node minimum who value is less than or equal to its α value does not need further exploration. The β value is used in a similar manner to prune maximum nodes. An example of Alpha Beta Pruning a decision tree can be seen below in Figure 5.

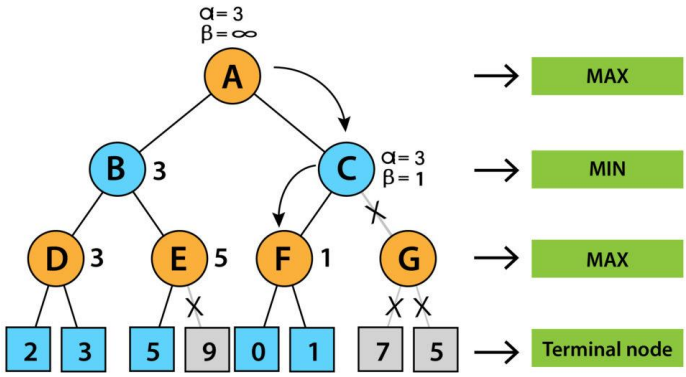


Figure 5. Alpha Beta Pruning

Thus by implementing Alpha Beta Pruning, we can cut out the need to compute every single node and only compute nodes that are not pruned, cutting our computational time and cost.

III. The Horizon Effect and Quiescence search

An issue many chess AI's have when making decisions, is the limitations of the computer to see N number of moves ahead (search depth). There are too many possibilities in chess that a computer can not see all possible moves and make a decision or at least in a computationally fast and efficient manner. Thus, the AI's decision to make an optimal move is only as optimal as the amount of data for N number of moves it can compute. The problem with having limited knowledge is that there may be an optimal move beyond the horizon of N number of moves. This issue is known as the Horizon Effect. An example of the Horizon Effect can be explained if you consider a computer which can only see 5 moves ahead in chess. Based on this knowledge, the AI decides to make a move between two choices: one of which will result in the loss of a knight, but avoids the loss of a queen, a highly valued piece and the other results in the direct loss of the queen [12, 16]. To the AI, the former is the optimal move, as it sacrifices a lower rank piece for a high valued piece. However, the issue becomes clear on the 6th move, which is beyond the horizon of the AI, and on the 6th move, the player loses the queen. Thus, in actuality the perceived optimal move has some disastrous consequences as now the player has lost both their knight and queen, and the latter choice is now the optimal move in hindsight. Thus, to avoid the issue of the horizon effect, we can implement an algorithm to our current chess AI called Quiescence search.

Quiescence search is an additional search to the current search depth, beyond the terminal nodes in order to get a more optimal decision or move. The criteria for the extra search can be different things depending on the game or application. For chess, the quiescence search typically only looks at moves which will result in violatale border moves that will significantly change the board state. Thus certain moves, like capturing a piece, can be considered a highly volatile move, and simply moving a piece without capture could be considered a stable or 'quiet' state. The quiescence search will keep searching additional volatile nodes until all nodes are at a quiet state and then end. Thus, all quiet positions are evaluated and accounted for when making an optimal decision [13, 16]. The quiescence search does take up a lot of computational time typically in chess AI's, and for games like Go, where the game is very unstable, it can take a significant amount of time. Now with our minimax algorithm with alpha beta pruning and quiescence search, our AI can make optimal decisions based on the board state.

IV. Convolutional Neural Network (CNN)

In addition to the traditional Chess algorithms we also tried to use a convolution neural network implementation. A CNN is a deep learning algorithm and is able to determine and assign importance to various aspects of an object. A CNN consists of a convolution layer, pooling layer, and a fully connected layer. Between each layer is an activation function. The convolution layer follows a convolution of the data, whether that be a 2D convolution of image or a multidimensional convolutions of a feature. The pooling layer then compresses the data, to only retain the most important information. In some pooling layers, the maximum value of a feature is taken. Finally, the fully connected layer is the final layer of a CNN and the data is then classified [18]. An example of a CNN algorithm can be seen below in Figure 6.

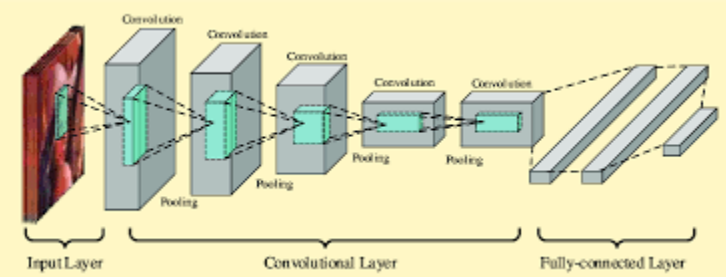


Figure 6. CNN Network

In python, we can use Keras library as a way to build our network. Once a CNN is developed, we can train it with data and use back propagation to develop the proper feature weights and return a trained model. The data set for our model can be created by creating random board states and associating them with a score, similar to our previously mentioned evaluation function. But in our case we use StockFish, an open source chess engine, to score our boards [17]. Thus we can use the data set to train our CNN and see how it performs.

E. IMPLEMENTATION

I. Non neural network approach

Consider black and white are two chess players. At the end of implementation, we would make AI to play against itself, to human (basically us) and at last to Stockfish for the best result check.

First, the board is evaluated using 9 parts: material, pawn structure, knight structure, bishop structure, rook structure, queen structure, king structure. For this purpose, first the last 8 parts scores were randomly initialized by keeping in mind some basic points like, the chances of the white's king crossing the centerline will be less than 20% and therefore we placed negative values in that matrix. After that, the material score is calculated by summing each piece's weights multiplied by the difference between the number of that piece between white and black.

```
wp = len(board.pieces(chess.PAWN, chess.WHITE))
bp = len(board.pieces(chess.PAWN, chess.BLACK))
wn = len(board.pieces(chess.KNIGHT, chess.WHITE))
bn = len(board.pieces(chess.KNIGHT, chess.BLACK))
wb = len(board.pieces(chess.BISHOP, chess.WHITE))
bb = len(board.pieces(chess.BISHOP, chess.BLACK))
wr = len(board.pieces(chess.ROOK, chess.WHITE))
br = len(board.pieces(chess.ROOK, chess.BLACK))
wq = len(board.pieces(chess.QUEEN, chess.WHITE))
bq = len(board.pieces(chess.QUEEN, chess.BLACK))
```

```
material = 100 * (wp - bp) + 320 * (wn - bn) + 330 * (wb - bb) + 500 * (wr - br) + 900 * (wq - bq)
```

Then, we calculated the evaluation function that is the sum of the material scores and the individual scores for white and black. When it's positive, we considered white is winning and for black, we negated it since an advantageous situation for white is equivalent to an unfavorable situation for black.

```
eval = material + pawnsq + knightsq + bishopsq + rooksq + queensq + kingsq
if board.turn:
    return eval
else:
    return -eval
```

Next, to score the game and to search for the moves, we used Negamax Implementation of the Minimax Algorithm.

```
def selectmove(depth):
    try:
        move = chess.polyglot.MemoryMappedReader("C:/Users/Arpita Das/CSNAP/books/varied.bin").weighted_choice(board).move()
        movehistory.append(move)
        return move
    except:
        bestMove = chess.Move.null()
        bestValue = -99999
        alpha = -100000
        beta = 100000
        for move in board.legal_moves:
            board.push(move)
            boardValue = -alphabeta(-beta, -alpha, depth-1)
            if boardValue > bestValue:
                bestValue = boardValue;
                bestMove = move
            if( boardValue > alpha ):
                alpha = boardValue
            board.pop()
        movehistory.append(bestMove)
        return bestMove
```

Before starting to play, the last step is to use α-β pruning to reduce execution time with quiescence search to evaluate the positions only where there are no winning tactical moves to be made.

```
def alphabeta(alpha, beta, depthleft):
    bestscore = -9999
    if (depthleft == 0):
        return quiesce(alpha, beta)
    for move in board.legal_moves:
        board.push(move)
        score = -alphabeta(-beta, -alpha, depthleft - 1)
        board.pop()
        if (score >= beta):
            return score
        if (score > bestscore):
            bestscore = score
        if (score > alpha):
            alpha = score
    return bestscore

def quiesce( alpha, beta ):
    stand_pat = evaluate_board()
    if( stand_pat >= beta ):
        return beta
    if( alpha < stand_pat ):
        alpha = stand_pat

    for move in board.legal_moves:
        if board.is_capture(move):
            board.push(move)
            score = -quiesce( -beta, -alpha )
            board.pop()

            if( score >= beta ):
                return beta
            if( score > alpha ):
                alpha = score
    return alpha
```

II. Neural network approach

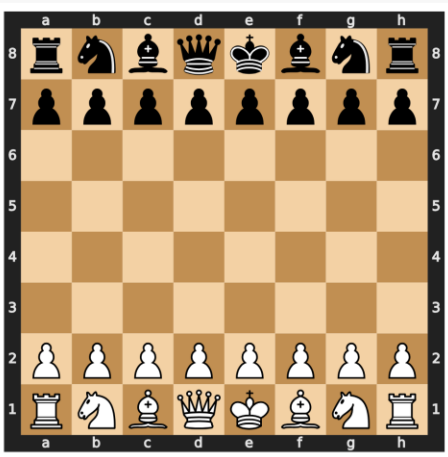
This approach also uses MINIMAX algorithm as a move search algorithm, however, this time we built the model for Chess environment using CNN and trained it.

In [22]:
```
import tensorflow.keras.models as models
import tensorflow.keras.layers as layers
import tensorflow.keras.utils as utils
import tensorflow.keras.optimizers as optimizers


def build_model(conv_size, conv_depth):
  board3d = layers.Input(shape=(14, 8, 8))

  # adding the convolutional layers
  x = board3d
  for _ in range(conv_depth):
    x = layers.Conv2D(filters=conv_size, kernel_size=3, padding='same', activation='relu')(x)
  x = layers.Flatten()(x)
  x = layers.Dense(64, 'relu')(x)
  x = layers.Dense(1, 'sigmoid')(x)

  return models.Model(inputs=board3d, outputs=x)
```

Now, we can start playing by getting moves from neural network using minimax.

In [22]:
```
# used for the minimax algorithm
def minimax_eval(board):
  board3d = split_dims(board)
  board3d = numpy.expand_dims(board3d, 0)
  return model(board3d)[0][0]


def minimax(board, depth, alpha, beta, maximizing_player):
  if depth == 0 or board.is_game_over():
    return minimax_eval(board)

  if maximizing_player:
    max_eval = -numpy.inf
    for move in board.legal_moves:
      board.push(move)
      eval = minimax(board, depth - 1, alpha, beta, False)
      board.pop()
      max_eval = max(max_eval, eval)
      alpha = max(alpha, eval)
      if beta <= alpha:
        break
    return max_eval
  else:
    min_eval = numpy.inf
    for move in board.legal_moves:
      board.push(move)
      eval = minimax(board, depth - 1, alpha, beta, True)
      board.pop()
      min_eval = min(min_eval, eval)
      beta = min(beta, eval)
      if beta <= alpha:
        break
    return min_eval


# this is the actual function that gets the move from the neural network
def get_ai_move(board, depth):
  max_move = None
  max_eval = -numpy.inf

  for move in board.legal_moves:
    board.push(move)
    eval = minimax(board, depth - 1, -numpy.inf, numpy.inf, False)
    board.pop()
    if eval > max_eval:
      max_eval = eval
      max_move = move

  return max_move
```

A snippet of the random Chessboard is given here:



F. Comparison of Results and analysis

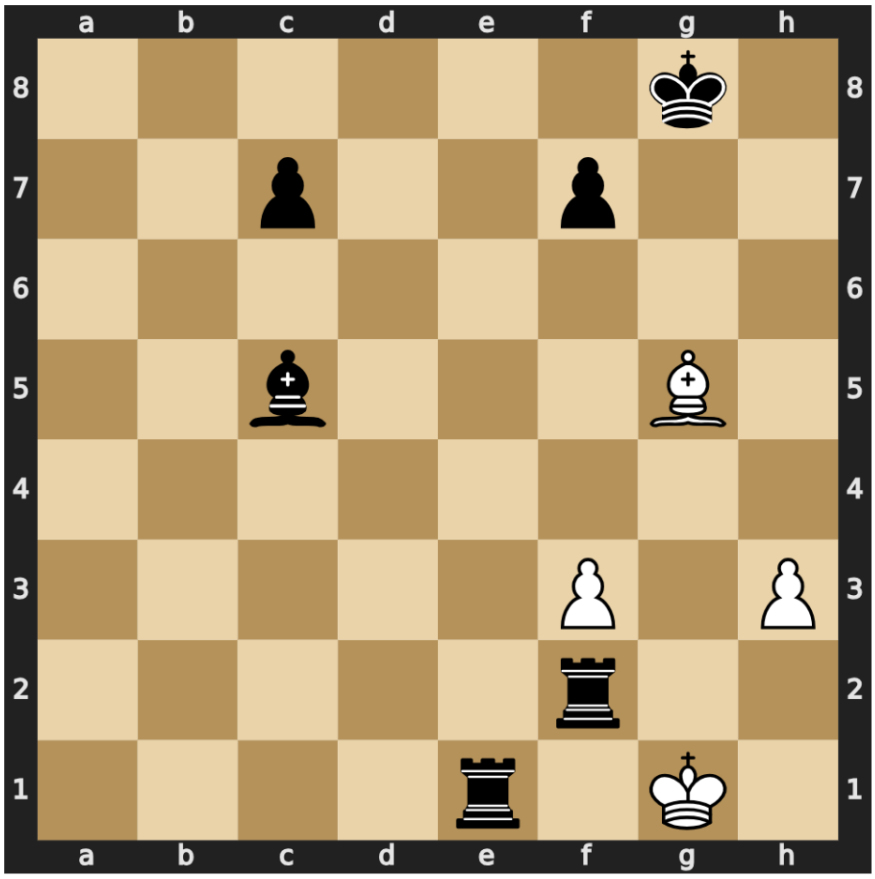
I. Non Neural network approach

1. Chess AI vs. Chess AI

1. Nf3 Nf6 2. Nc3 Nc6 3. e3 e6 4. d4 d5 5. Bb5 Bd6 6. Bxc6+ bxc6 7. Ne5 Bb7 8. Qf3 O-O 9. O-O Nd7 10. Nxd7 Qxd7 11. Rd1 c5 12. Rb1 Qc6 13. Ra1 Rfe8 14. Rd2 Rad8 15. Rd3 c4 16. Rd1 e5 17. dxe5 Rxe5 18. Rb1 Qc5 19. Ne2 Qb6 20. Nd4 Bc5 21. Qf4 Re4 22. Qg5 Ree8 23. Nf5 Qg6 24. e4 d4 25. e5 Be4 26. g4 Bxc2 27. Bf4 Qxg5 28. Bxg5 Rb8 29. Nxg7 Kxg7 30. Bf6+ Kg8 31. Rf1 Bxb1 32. Rxb1 c3 33. b3 c2 34. Re1 Bb4 35. Rf1 Bd2 36. e6 Rxe6 37. Bxd4 c1=Q 38. Rxc1 Bxc1 39. Bxa7 Ra8 40. Bd4 Rxa2 41. Bc5 Re1+ 42. Kg2 Bf4 43. b4 Rb2 44. h3 Bd2 45. Bd4 Rxb4 46. Bf6 h6 47. Bd8 Bf4 48. Bf6 Rc1 49. Bd8 Bd6 50. g5 hxg5 51. Bxg5 Rc2 52. Be3 Rh4 53. Bg5 Re4 54. Kf1 Bc5 55. f3 Rf2+ 56. Kg1 Re1# 0-1

```
[Event "AI vs AI"]
[Site "N/A"]
[Date "2023-12-19"]
[Round "1"]
[White "Ai"]
[Black "Ai"]
[Result "0-1"]
```

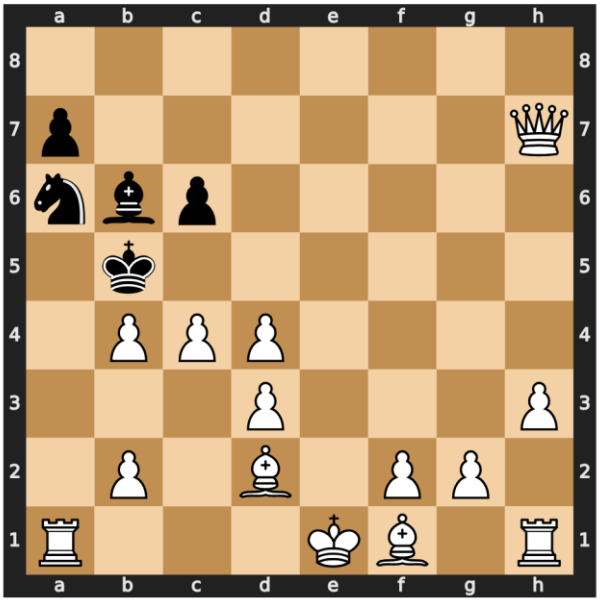
1. Nf3 Nf6 2. Nc3 Nc6 3. e3 e6 4. d4 d5 5. Bb5 Bd6 6.



In the first four moves, Chess AI (black) seems to have mirrored the moves from Chess AI (white) until in move 5, Chess AI (white) targets to trap the opponents' knight on c3, by placing bishop on b5, while in response Chess AI (black) brings bishop on d6. The game goes on in balance even after both the players lose their Queens in move 27 (white) and 28 (black). In move 29, white loses a precious knight to the black king, and for the first time the game sways in the favor of black. White further slides away from the game as it doesn't take the black pawn in c3 and instead moves to b3 from b2. This mistake eventually costs white the last Rook in move 38. In the next few moves white loses the game as the two black rooks and the black bishop dominate the white half.

2. Chess AI vs. Human (basically we played it)

```
[Event "AI vs Human"]
[Site "N/A"]
[Date "2023-12-07"]
[Round "1"]
[White "AI"]
[Black "Human"]
[Result "1-0"]
```



As amateurs, we (black) played with an aggressive mindset and Chess AI (white) took advantage of this. Without sufficient backup, in the first few moves we lost a couple of pawns in return for nothing. Chess AI (white) made excellent choices of moves and capitalized the lack of defense from our side (black) and eventually took the black queen. We made a comeback by taking both the white knights but lost a bishop to the white queen in the process. From there onwards, the white queen took both the black rooks, a knight one after another leaving black in a more helpless situation. Eventually, attacking with the bishop and the pawns, black won the match.

3. Chess AI vs. Stockfish

```
1. e4 Nf6 2. e5 Ne4 3. d3 Nc5 4. d4 Ne6 5. d5 Nc5 6. Nc3
d6 7. exd6 exd6 8. Nf3 Nbd7 9. b4 Qf6 10. Nd4 Qe7+ 11. Be2
Ne4 12. Nxe4 Qxe4 13. O-O Qxd5 14. Re1 Ne5 15. Bb5+ Kd8
16. c4 Nf3+ 17. Qxf3 Qe5 18. Rxe5 dxe5 19. Qxf7 Be7 20.
Qd5+ Bd7 21. Qxd7# 1-0
```

```
[Round "1"]
[White "Stockfish11"]
[Black "MyAi"]
[Result "1-0"]
```

Stockfish is a strong and open source chess engine that is readily available in our android. It is the most powerful chess-engine platform so the best way to check our coding output is to play it against Stockfish.

While in the first 6 moves, Stockfish (white) made way for an open attack by bringing the pawns at d2 and e2 forward, Chess AI (black) was fixated with the knight. Even in move 12, the game was balanced as both the players lost one of their knights. In move 18 Chess AI loses the queen thanks to strategic build-up from Stockfish and soon white conquers the game, as white queen corners black king by taking the bishop guarding the later.

II. Neural network approach

```
r . b . . . k .
p p . . . . p p
. . . K . . . .
. . . . . r . .
. . . . q . . .
. . . . . . . .
. P P . . . P P
. . . . . . . .

r . b . . . k .
p p . . . . p p
. . . K . . . .
. . . . q r . .
. . . . . . . .
. . . . . . . .
. P P . . . P P
. . . . . . . .
game_over
```

We made the codes such that moves are learnt from CNN as efficiently as possible. In this CNN approach, it was found that the move search became more efficient from AI for using CNN, although Stockfish won again.

G. CONCLUSION

In this project, we investigated the use of CNN based approach against non-neural based approach. We found that CNN based approach plays as efficiently as possible than the non-neural based approaches. AI never beat Stockfish given their move algorithm. Efficiency is always a major consideration when switching from a non-neural system to a machine learning approach, since the former systems are usually more efficient to evaluate than generic models. This is especially important for applications like a chess engine, where being able to search nodes quickly is strongly correlated with playing strength. Some earlier attempts at applying neural networks to chess have been thwarted by large performance penalties. For example, Thrun reported a difference of 2 orders of magnitude between his implementation of a neural-network based chess engine and a state of art chess engine of that time. [20] We found that our CNN based approach was at least two times more efficient than the non-neural based approaches with no performance penalty.

H. LIMITATIONS AND FUTURE WORK

One of the major limitations is, the Chess AI has always started its move with knights no matter who the opponent is. We think the quiescent search algorithm could be a reason for this. Another limitation is, the outcome of CNN was not prune to changes in hyperparameters like no of layers, activation function etc. Our initial choice for CNN was not only making the moves efficient but also reducing overall game time and especially beating Stockfish with different parameters which was not visible.

As a future work, we can investigate the use of deep reinforcement learning with automatic feature extraction. We can also try probabilistic approaches for the moves incorporating the reinforcement mechanism.

I. REFERENCES

[1] US Chess Rule Book Online, https://new.uschess.org/sites/default/files/media/documents/us-chess-rule-book-online-only-edition-chapters-1-2-10-11-9-1-20.pd .

[2] By: James Eade and Updated: 08-16-2023 From The Book: Chess For Dummies et al., “Chess Pieces and How They Move,” dummies, https://www.dummies.com/article/home-auto-hobbies/games/board-games/chess/knowing-the-moves-that-chess-pieces-can-make-186936/.

[3] “Chess: A chess library for python,” python, https://python-chess.readthedocs.io/en/latest/

[4] “Piece-square tables,” Piece-Square Tables - Chessprogramming wiki, https://www.chessprogramming.org/Piece-Square_Tables

[5] “Material,” Material - Chessprogramming wiki, https://www.chessprogramming.org/Material

[6] “Point value,” Point Value - Chessprogramming wiki, https://www.chessprogramming.org/Point_Value#:~:text=Common%20rule%20of%20thumb%20are,a%20Computer%20for%20Playing%20Chess%20.

[7] “Minimax,” Minimax - Chessprogramming wiki, https://www.chessprogramming.org/Minimax.

[8] “Artificial Intelligence: Mini-max algorithm - javatpoint,” www.javatpoint.com,

https://www.javatpoint.com/mini-max-algorithm-in-ai.

[9] “Searching state space,” AI - Search, https://cis.temple.edu/~pwang/5603-AI/Lecture/02-Searching.htm.

[10] “Negamax,” Wikipedia, https://en.wikipedia.org/wiki/Negamax.

[11]“Artificial Intelligence 3E,” 14.3 Solving Perfect Information Games ‣ Chapter 14 Multiagent Systems ‣ Artificial Intelligence: Foundations of Computational Agents, 3rd Edition‣ Chapter 14 Multiagent Systems ‣ Artificial Intelligence: Foundations of Computational Agents, 3rd Edition, https://artint.info/3e/html/ArtInt3e.Ch14.S3.html#SS1.p4.

[12]“Horizon effect,” Horizon Effect - Chessprogramming wiki, https://www.chessprogramming.org/Horizon_Effect.

[13]“Quiescence search,” Quiescence Search - Chessprogramming wiki, https://www.chessprogramming.org/Quiescence_Search.

[14] A. Gaikwad, “Let’s create a chess AI,” Medium, https://medium.com/dscvitpune/lets-create-a-chess-ai-8542a12afef.

[15] Neural Technologies, “Opening gambit – A history of chess AI and automation,” Neural Technologies, https://www.neuralt.com/opening-gambit-a-history-of-chess-ai-and-automation/

[16] M. Lai, “Giraffe: Using Deep Reinforcement Learning to Play Chess,” dissertation, 2015

[17] N. Puram, “Training a chess AI using tensorflow,” Medium, https://medium.com/@nihalpuram/training-a-chess-ai-using-tensorflow-e795e1377af2

[18] “How Convolutional Neural Networks Work,” Table of contents, https://e2eml.school/how_convolutional_neural_networks_work.html

[19] Deepanshi, “Beginners guide to Convolutional Neural Network with implementation in python,” Analytics Vidhya, https://www.analyticsvidhya.com/blog/2021/08/beginners-guide-to-convolutional-neural-network-with-implementation-in-python/

[20] Sebastian Thrun. Learning to play the game of chess. Advances in neural information pro- cessing systems, 7, 1995.