Danmarks
Tekniske
Universitet

02461 - Intelligente systemer

AUTHOR

Nicholas Erup Larsen (s224175)
Noah Ryu Nguyen (s224207)
Caroline Schubert Mortensen (s224173)

January 3, 2024

1 Abstract

by Nicholas Erup Larsen

Traditionally, chess AIs have used rule-based systems and search algorithms in order to become the reigning chess champions for 20 years, but these systems have hard-coded limitations to avoid searching through the absurdly high number of possible positions that exist for every move. 2017 marked a turning point for AI when AlphaZero beat those same machines using new self-play reinforcement learning techniques.

Surprisingly, there has been few attempts to recreate this success with other forms of neural networks. In this paper, we train a five-layer convolutional neural network (CNN) on 6059 different games of chess from professional players and Stockfish, totaling to 1 million positions, and use supervised learning to ideally beat an average person with little to no prior experience. In this context, our average person is being imitated by a 500 elo Stockfish.

Our results show a discrepancy between the results of our training loss data and the actual gameplay performance of our chess AI. This could suggest that convolutional neural networks might be an inadqeuate fit for this type of problem.

Contents

1 Abstract **i**

2 Introduction **1**

3 Methods **2**

4 Results **5**

5 Discussion **6**

6 References **8**

7 Appendiks **9**

2 Introduction

by Caroline Schubert Mortensen

This project will focus on the game called chess - a strategic and challenging two-man game, where logic and coherence are important in order to win over the opponent. Each player possesses 16 game pieces, each of which has different properties that affect how they can move on the game board. The goal of the game is to checkmate the opponent, which is done by attacking the opponent's king in a way that the attack cannot be parried.

In 1996, the computer Deep Blue beat the reigning world champion, Kasparov, in chess. An excellent programming presentation, which was based on the background of the fights of human grandmasters. Later AlphaZero entered the spotlight, which, unlike Deep Blue, was only fed the rules of chess, and which has subsequently experimented itself by playing against itself.

Over the years, more and more open source chess computers have appeared. One of the strongest chess program so far is called Stockfish, which has won the Top Chess Engine Championship over 10 times. Stockfish implements an advanced alpha-beta search, uses bitboards and compared to other engines, is characterized by its great search depth. Can we program a new chess engine? A chess engine based on games from Stockfish as well as professional chess players. A engine that can beat an average person with no or little experience?

To attempt this, we propose the development of a convolutional neural network-based chess engine, utilizing a combination of extensive game data and evaluations from the widely-used open-source engine, Stockfish. The implementation of this approach presents various challenges and complexities but is a necessary step toward achieving the ultimate goal of creating a highly effective chess engine. We hypothesize that the larger the amount of games our network trains on, where we use supervised learning, the better performance it will have in comparison to its previous generations.

3 Methods

by Noah Ryu Nguyen

Below is a template of our topology for the entire project.

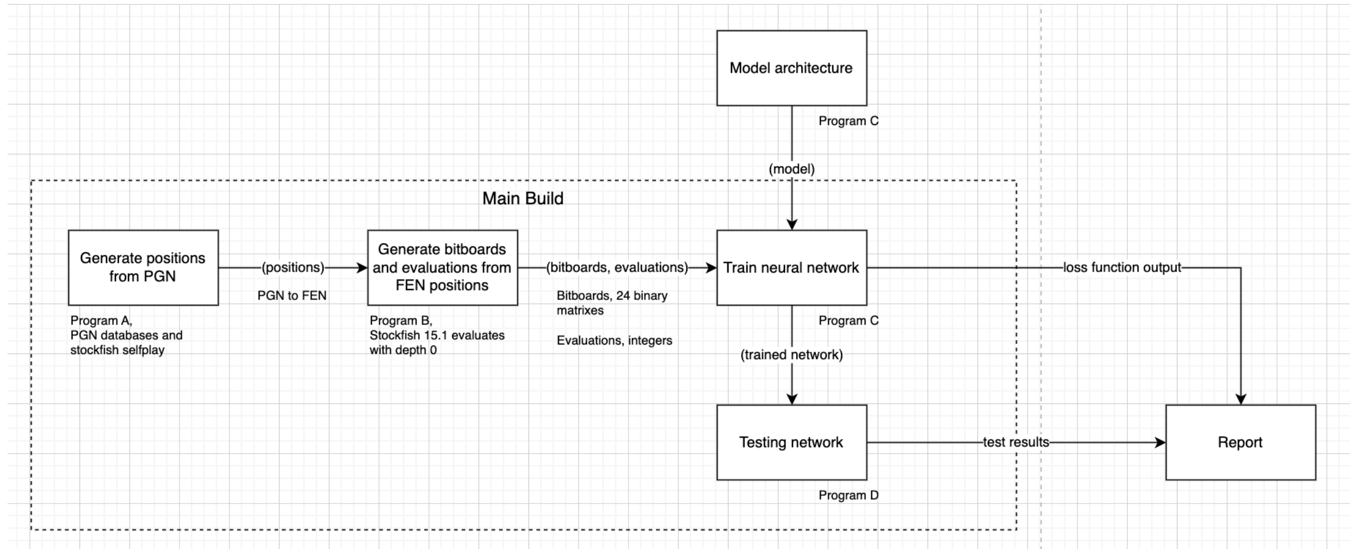


Figure 1: Topology / Main build for project

Program A

In our approach to creating a neural network for chess, we evaluated two options for obtaining training data. Our first option was to use the python-chess library, which offered a variety of features such as chess rules, moves generation, evaluations, and validations. Additionally, it allowed for direct communication with the Stockfish engine. However, we found that using Stockfish to self-play and record games was not a practical option as it required significant resources and resulted in limited game diversity. As a result, we had to find another approach to collecting our data.

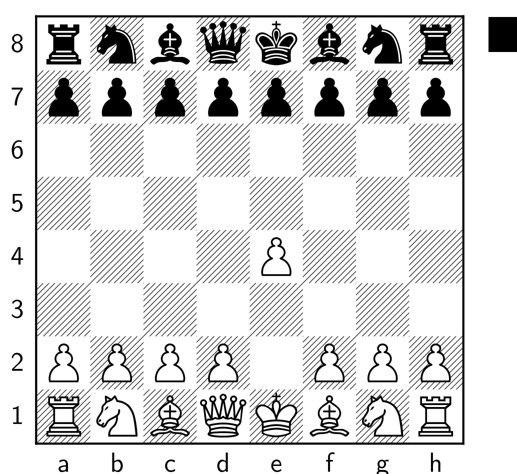
Our second approach to obtaining data for training the neural network was to download a large number of professional chess games in PGN format from databases. While this method was faster than our previous approach, it presented its own set of challenges. Professional chess games often end before a stalemate or checkmate occurs, with players resigning or agreeing to a draw. To address this, we decided to self-play additional games using the python-chess library when a game ended prematurely. We then created a Forsyth-Edwards Notation (FEN) for each position in all the games and combined them into a single file. This resulted in a total of 6059 games and 957210 unique board positions, each represented as a FEN-string. With this data, we were able to convert the positions into binary data and corresponding evaluations for use in training the neural network.

Program B

To effectively train a neural network on chess data, it is necessary to convert the visual representation of a chess position into numerical data that a computer can understand. One way to achieve this is through the use of bitboards. From each FEN position, we extracted 24 binary bitboard matrices that contain information about the position. These bitboards provide a compact and efficient way for a computer to understand the layout of the pieces on the board and their movements. The following image shows an example of a chessboard represented using bitboards.

---

1 e4



e4 is the most common opening move.

```
[[0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 1 0 0 0]
 [0 0 0 0 0 0 0 0]
 [1 1 1 1 0 1 1 1]
 [0 0 0 0 0 0 0 0]]
```

1. white pawns

```
[[0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 1 0 0 0 0 1 0]]
```

2. white knights

```
[[0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0]
 [0 0 1 0 0 1 0 0]]
```

3. white bishops

Additionally, we had 21 other bitboards - 1 for the white queen, 1 for the white king, 6 for black pieces, 1 for the turn, 4 for castling rights, 1 for en passant, 1 for fifty move repetition rule, 1 for threefold repetition rule, 1 for mobility and 1 for the mobility of player not on the turn. A total of 24 bitboards were generated from a single board position. (APPENDIX TO CODE) We used this information, along with the evaluation from Stockfish (a value between 0 and 1 indicating the strength of the position for black or white) as input for a neural network. The goal of the network was to mimic the evaluation function of Stockfish. All of the data were converted to tensors to manipulate and tune the values using PyTorch.

Program C

We use a convolutional neural network and find it interesting because it allows the engine to learn from experience and improve over time, rather than relying solely on rule-based systems and hard-coded chess knowledge like Stockfish. A CNN can be trained to recognize patterns that are not easily captured by traditional rule-based systems which can ideally lead to creativity and novel ideas. Once trained, the CNN can be used to evaluate positions and predict moves using a minimax algorithm, without relying on any hand-tuned rules.

Our model defines a convolutional neural network (CNN) for playing chess. The model takes the input of 24 bitboards and a corresponding stockfish evaluation and outputs a single value representing the predicted strength of the current chess position for the side to move. The CNN is trained to recognize patterns in the chess positions that are not easily captured by traditional rule-based systems. The model is initialized with three parameters: conv-size, conv-depth, and dropout-rate. The model architecture comprises convolutional layers, batch normalization, ReLU activation functions, dropout layers, dense layers, and a value head that predict the optimal chess position. Lastly, the model is moved to the GPU for faster processing.

For the training, the data is loaded from a .pt file and split into a training and validation set. The Adam optimizer is defined with a specified learning rate and the mean squared error loss function is defined. A counter is initialized to track the number of consecutive increases in validation loss, and a threshold is set for the number of consecutive increases before stopping the training. The training process consists of a number of full iterations (epochs) and in each iteration, the model is passed the training data and the gradients are calculated, then the optimizer updates the parameters of the model using backpropagation. After each epoch, the validation loss is calculated and recorded. The training ends when the validation loss has increased for a certain number of consecutive epochs or when the maximum number of iterations is reached.

Program D

The minimax algorithm is a decision-making algorithm that is commonly used in two-player games such as chess. It evaluates all possible moves of both players and selects the move that leads to the best outcome for the current player, assuming that the opposing player will also select the move that leads to the best outcome for them.

Alpha-beta pruning is a technique used to improve the performance of the minimax algorithm. It eliminates branches of the search tree that are unlikely to be selected, reducing the number of nodes that need to be evaluated and speeding up the search process. The algorithm uses the alpha and beta values to keep track of the best move that the current player can make and the best move that the opposing player can make, respectively. If beta <= alpha then the function breaks the loop since we don't need to keep checking the moves since the max player already found a better move.

Our function looks at whether the current player is trying to win (white) or prevent the opponent from winning (black). If the current player is white, it goes through all possible moves and makes each one on the board. It then calls itself with the new board, the same depth, and new values for alpha and beta. After trying all the moves, it chooses the one that leads to the best outcome for white. If the current player is black, it does the same thing but chooses the move that leads to the worst outcome for the white. All of our programs can be found in the appendix.

4 Results

by Nicholas Erup Larsen

Below is a table of how our model has performed versus three different opponents. Given the long computing time for moves beyond a depth of 3 and the static nature of its playstyle, we have limited the amount of games for each opponent to 10 per side. This also means a confidence interval seems meaningless to include. The results are to be interpreted as win/draw/loss for white.

Depth	White	Black	Win	Draw	Loss	CNN win %
1	CNN	Random	4	6	0	40 %
	Random	CNN	7	3	0	0 %
2	CNN	Random	5	5	0	50 %
	Random	CNN	4	5	1	10 %
3	CNN	Random	8	2	0	80 %
	Random	CNN	2	8	0	20 %
1-3	CNN	CNN	0	10	0	0 %
	CNN	CNN	0	10	0	0 %
1-3	CNN	Stockfish	0	0	10	0 %
	Stockfish	CNN	10	0	0	0 %

Figure 2: CNN performance on the chess board

In the results, Stockfish's parameters are set to mimic 100 elo. The reason for the static data in CNN vs itself and CNN vs Stockfish is, despite changing the depths of the minimax algorithm which does alter the model's playstyle slightly, it still plays the exact same moves invariably. Therefore, this outcome can be extrapolated beyond the range of 20 games except for randomly selected moves as the only opponent which forces our model to evaluate new positions.

Below is the training and validation loss function for the model used in the results above. The y-axis graphs the mean squared error between the prediction tensor vs evaluation tensor (loss function), and the x-axis graphs the number of iterations through the entire dataset.

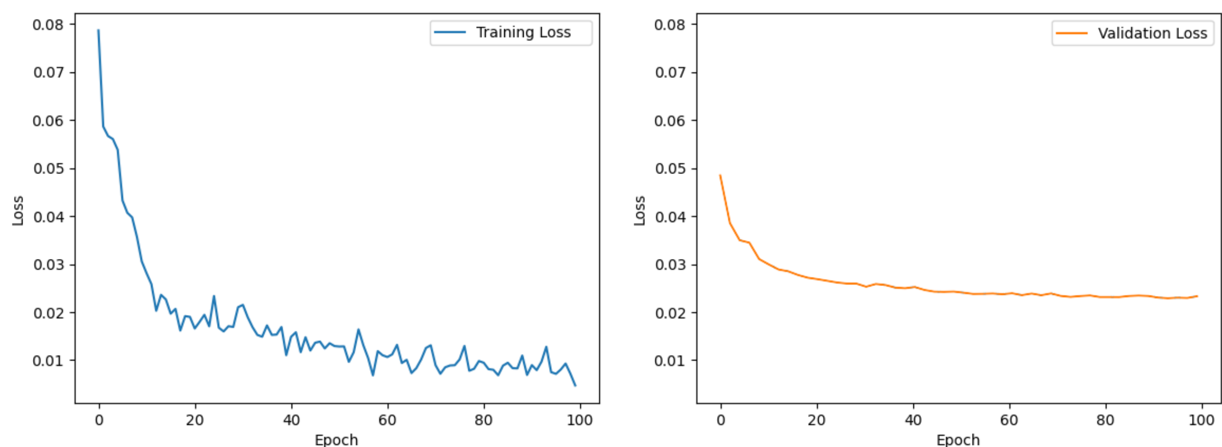


Figure 3: Graph of the training and validation loss data

5 Discussion

by Nicholas Erup Larsen, Noah Ryu Nguyen and Caroline Schubert Mortensen

As evident in the last section, the results did not meet our expectations. Our initial hypothesis was that the trained CNN would be able to beat a 500 elo rated player 10 out of 10 times, however, it only managed to achieve a positive score versus randomly generated moves. To our puzzlement, in spite of poor performance on the chess board, the data from our training and test loss showed desired developments and improved itself for every epoch. So although the model is learning, which is also reflected in the non-randomness of its moves, why is it not playing better?

At first, the model would play weird openings like **a4** which is notoriously one of the worst openings in the game. This happened due to a bug in the code of our minimax algorithm which expected values between $-\infty$ to ∞, however, our output evaluations from the CNN had values between 0 and 1. After fixing this, the model started playing more ordinary openings like **e4**.

Interestingly, with a validation mean squared error lower than 0.0025, the difference in centipawns (Stockfish's evaluation method) becomes less than 0.05.[7] With such a minimal difference in centipawns, in theory, our model should output the same evaluations as Stockfish with at least 1 decimal precision. One reason why this performance is not reflected on the chess board could be that our minimax algorithm, despite previous fixes, is not properly searching through the evaluations or has some other error. Another reason could be lack of generalization but this hypothesis does conflict with the graph from our validation set which does not seem to be overfitting. It is hard to pinpoint what exactly is working suboptimally but perhaps the observations we made from watching it play could provide further insight.

One frequent observation from games of multiple differently trained versions of our model, playing versus 100 elo imitated Stockfish, showed that the CNN would sacrifice its queen within the first couple of moves for nothing in return. A queen is widely regarded as the most valuable piece on the board (besides the king) so usually a queen sacrifice is used as a trap to lure the opponent into a checkmate. However, in this case it seemed like the model neither had any concept of the queen piece's value or devised a strategy to use it in a clever way.

We contemplated combatting this issue by manually assigning all the pieces to an appropriate value (usually pawn: 1, knight/bishop: 3, rook: 5, queen: 9) and use that as a bias for the network. But we came to the same conclusion as another paper has worded nicely; *"We noticed that adding the information about the value of the pieces does not provide any advantage to the ANNs. On the contrary both for the MLP and the CNN this penalizes their overall performances."*[7]

Another pattern we found is that the model struggles to checkmate in end-game positions. Even with clear winning positions and material advantage, it often ends up playing the same two moves endlessly until it draws because of the fifty-move rule, sometimes even while being able to checkmate. This, combined with the fact that unless challenged by new and unknown positions it will play the same moves every game, suggests that there's some rigid nature to CNNs that prevents them from learning the adaptive, generalized behaviour that makes players, and other AIs, excel at chess. A

2015 study came to a similar conclusion, describing the game of chess as too asymmetrically complex with all its intricacies and rules for a CNN to learn alone.[10]

To conclude, it is difficult to say what exactly went wrong and why our model did not perform as expected. It is possible our datasets contained errors, was too small or there were minor bugs in our neural network. CNNs are extremely sensitive to the hyperparameters such as the convolutional depth, the number of neurons, the number of layers and overall build size. It is a continuous and exasperating trade-off between generalization and complexity which always lead to either overfitting or underfitting. A CNN could be maximizing its performance during training, yet it would perform poorly on unseen data since it would not be able to adequately adapt and comprehend general structures. In other words, our network might be an expert at playing perfectly in games identitical or extremely similar to the data it was trained on but fails to evaluate moves properly in unrecognizable positions.

6 References

[1] Acquisition of Chess Knowledge in AlphaZero - arxiv.org

[2] AlphaZero: Shedding new light on chess, shogi, and Go - deepmind.com

[3] Bitboards - chessprogramming.org

[4] Chess Engine Design - hkopp.github.io

[5] Creating a Chess AI with TensorFlow - youtube.com

[6] How do modern chess engines work? - youtube.com

[7] Learning to Play Chess with Minimal Lookahead and Deep Value Neural Networks - researchgate.net, p.42 ll.20-21, p.39 ll.14-16

[8] Main Page - chessprogramming.org

[9] Neural network topology - lczero.org

[10] Predicting Moves in Chess using Convolutional Neural Networks

[11] Search: Games, Minimax, and Alpha-Beta - youtube.com

[12] Understanding AlphaZero Neural Network's SuperHuman Chess Ability - marktechpost.com

7 Appendiks

Program A

```python
import chess
import chess.engine
import chess.pgn
import numpy as np
import sys


try:
    pgn_path = sys.argv[1]
    fen_path = sys.argv[2]
    engine_path = sys.argv[3]
except IndexError:
    raise SystemExit(f"Usage: {sys.argv[0]} <pgn-file> <fen-output-file> <uci-engine-executable-pa

# Engine
engine = chess.engine.SimpleEngine.popen_uci(engine_path)
# Open the PGN file
pgn = open(pgn_path)

# Create a list to store the positions
positions = []
pgn_positions = 0
stockfish_positions = 0
game_count = 0


# Iterate through each game in the PGN file
while True:
    game = chess.pgn.read_game(pgn)
    if game is None:
        break

    # Get the moves of the game
    GameMoves = game.mainline_moves()

    # Set up the board for the game
    board = game.board()

    # Iterate through each position in the game
    for move in GameMoves:
        board.push(move)
        fen = board.fen()
        positions.append(fen)
        pgn_positions += 1
```

```python

    # Check if the game was resigned or drawn
    # if not board.is_checkmate() and not board.is_stalemate():
    # Use Stockfish to play the game from the last position
    while not board.is_game_over():
        result = engine.play(board, chess.engine.Limit(time=0.001))
        board.push(result.move)
        fen = board.fen()
        positions.append(fen)
        stockfish_positions += 1

    game_count += 1

    print(f"\rGames: {game_count:,}, PGN positions: {pgn_positions:,}, Stockfish positions: {stock

print()
print(f"PGN positions: {pgn_positions}")
print(f"Stockfish positions: {stockfish_positions}")
print(f"Total positions: {len(positions)}")


# Close the engine and the PGN file
engine.quit()
pgn.close()

# Save the file
np.savez_compressed(fen_path,positions=positions)
```

Program B

```python
import numpy as np
import chess
import chess.engine
import torch
import sys


# SETUP & INATIALIZE
squares_index = {
  'a': 0,
  'b': 1,
  'c': 2,
  'd': 3,
  'e': 4,
  'f': 5,
  'g': 6,
  'h': 7
```

```
18 }
19
20
21 # example: h3 -> 17
22 def square_to_index(square):
23   letter = chess.square_name(square)
24   return 8 - int(letter[1]), squares_index[letter[0]]
25
26
27 def split_dims(board):
28     # this is the 4d matrix
29     board4d = np.zeros((24, 8, 8), dtype=np.int8)
30
31     # here we add the pieces's view on the matrix
32     for piece in chess.PIECE_TYPES:
33         for square in board.pieces(piece, chess.WHITE):
34             idx = np.unravel_index(square, (8, 8))
35             board4d[piece - 1][7 - idx[0]][idx[1]] = 1
36         for square in board.pieces(piece, chess.BLACK):
37             idx = np.unravel_index(square, (8, 8))
38             board4d[piece + 5][7 - idx[0]][idx[1]] = 1
39
40     # add attacks and valid moves too
41     # so the network knows what is being attacked
42     aux = board.turn
43     board.turn = chess.WHITE
44     for move in board.legal_moves:
45         i, j = square_to_index(move.to_square)
46         board4d[12][i][j] = 1
47     board.turn = chess.BLACK
48     for move in board.legal_moves:
49         i, j = square_to_index(move.to_square)
50         board4d[13][i][j] = 1
51     board.turn = aux
52
53     # set the turn dimension
54     if board.turn == chess.WHITE:
55         board4d[14] = 1
56     else:
57         board4d[14] = 0
58
59     # add bitboard for en passant
60     if board.ep_square:
61         idx = np.unravel_index(board.ep_square, (8, 8))
62         board4d[15][idx[0]][idx[1]] = 1
63
64     # add bitboards for castling rights
65     if board.has_kingside_castling_rights(chess.WHITE):
66         board4d[16][7][7] = 1
67     if board.has_queenside_castling_rights(chess.WHITE):
```

```
68          board4d[17][7][0] = 1
69      if board.has_kingside_castling_rights(chess.BLACK):
70          board4d[18][0][7] = 1
71      if board.has_queenside_castling_rights(chess.BLACK):
72          board4d[19][0][0] = 1
73
74      # binary channel for repetition
75      repetitions = board.can_claim_fifty_moves()
76      if repetitions:
77          board4d[20][:][:] = 1
78
79      # binary channel for threefold repetition rule
80      repetitions = board.can_claim_draw()
81      if repetitions:
82          board4d[21][:][:] = 1
83
84      # add bitboard for mobility
85      for move in board.legal_moves:
86          i, j = square_to_index(move.from_square)
87          board4d[22][i][j] = 1
88
89      # add bitboard for mobility of player not on turn
90      aux = board.turn
91      board.turn = chess.WHITE if board.turn == chess.BLACK else chess.BLACK
92      for move in board.pseudo_legal_moves:
93          if board.is_legal(move):
94              i, j = square_to_index(move.from_square)
95              board4d[23][i][j] = 1
96      board.turn = aux
97
98      return board4d
99
100 try:
101     fen_path = sys.argv[1]
102     output_path = sys.argv[2]
103     engine_path = sys.argv[3]
104 except IndexError:
105     raise SystemExit(f"Usage: {sys.argv[0]} <fen-npz-file> <output-file> <uci-engine-executable-pa
106
107
108 # Load the NPZ file
109 positions = np.load(fen_path)["positions"]
110
111 counter = 0
112
113 with chess.engine.SimpleEngine.popen_uci(engine_path) as sf:
114     # Create a new list to store the scores
115     evaluations = []
116     positionsBitboard = []
117
```

```
118     # Iterate through the positions
119     for fen in positions:
120         # Create a board from the FEN string
121         board = chess.Board(fen)
122 
123         # Use the sf object to perform the analysis
124         result = sf.analyse(board, chess.engine.Limit(depth=1))
125         score = (result['score'].white().wdl(ply=1).expectation())
126 
127         if(not board.is_game_over()):
128             # push the principle varation's move on the board
129             board.push(result["pv"][0])
130 
131         # Add the score and positionsBitboard to the lists
132         evaluations.append(score)
133         positionsBitboard.append(split_dims(board))
134 
135         counter += 1
136         if (counter % 1000 == 0):
137             print(f"Evaluations: {len(evaluations)}")
138 
139 # Convert the numpy arrays to PyTorch tensors
140 evaluations = [val if val is not None else 0 for val in evaluations]
141 
142 
143 evaluations = np.array(evaluations)
144 positionsBitboard = np.array(positionsBitboard)
145 
146 positionsBitboard_tensor = torch.tensor(positionsBitboard, dtype=torch.float32)
147 evaluations_tensor = torch.tensor(evaluations, dtype=torch.float32).reshape(-1, 1)
148 
149 torch.save({'positionsBitboard': positionsBitboard_tensor, 'evaluations': evaluations_tensor}, out
```

Program C

```
1 import chess
2 import chess.engine
3 import torch
4 from torch.utils.data import DataLoader
5 import torch.nn as nn
6 import torch.optim as optim
7 import numpy as np
8 import matplotlib.pyplot as plt
9 import torch.cuda as cuda
10 from sklearn.model_selection import train_test_split
11 
12 
```

```
13 # SETUP & INATIALIZE
14 squares_index = {
15   'a': 0,
16   'b': 1,
17   'c': 2,
18   'd': 3,
19   'e': 4,
20   'f': 5,
21   'g': 6,
22   'h': 7
23 }
24
25
26 # example: h3 -> 17
27 def square_to_index(square):
28   letter = chess.square_name(square)
29   return 8 - int(letter[1]), squares_index[letter[0]]
30
31
32 def split_dims(board):
33     # this is the 4d matrix
34     board4d = np.zeros((24, 8, 8), dtype=np.int8)
35
36     # here we add the pieces's view on the matrix
37     for piece in chess.PIECE_TYPES:
38         for square in board.pieces(piece, chess.WHITE):
39             idx = np.unravel_index(square, (8, 8))
40             board4d[piece - 1][7 - idx[0]][idx[1]] = 1
41         for square in board.pieces(piece, chess.BLACK):
42             idx = np.unravel_index(square, (8, 8))
43             board4d[piece + 5][7 - idx[0]][idx[1]] = 1
44
45     # add attacks and valid moves too
46     # so the network knows what is being attacked
47     aux = board.turn
48     board.turn = chess.WHITE
49     for move in board.legal_moves:
50         i, j = square_to_index(move.to_square)
51         board4d[12][i][j] = 1
52     board.turn = chess.BLACK
53     for move in board.legal_moves:
54         i, j = square_to_index(move.to_square)
55         board4d[13][i][j] = 1
56     board.turn = aux
57
58     # set the turn dimension
59     if board.turn == chess.WHITE:
60         board4d[14] = 1
61     else:
62         board4d[14] = 0
```

```
63
64      # add bitboard for en passant
65      if board.ep_square:
66          idx = np.unravel_index(board.ep_square, (8, 8))
67          board4d[15][idx[0]][idx[1]] = 1
68
69      # add bitboards for castling rights
70      if board.has_kingside_castling_rights(chess.WHITE):
71          board4d[16][7][7] = 1
72      if board.has_queenside_castling_rights(chess.WHITE):
73          board4d[17][7][0] = 1
74      if board.has_kingside_castling_rights(chess.BLACK):
75          board4d[18][0][7] = 1
76      if board.has_queenside_castling_rights(chess.BLACK):
77          board4d[19][0][0] = 1
78
79      # binary channel for repetition
80      repetitions = board.can_claim_fifty_moves()
81      if repetitions:
82          board4d[20][:][:] = 1
83
84      # binary channel for threefold repetition rule
85      repetitions = board.can_claim_draw()
86      if repetitions:
87          board4d[21][:][:] = 1
88
89      # add bitboard for mobility
90      for move in board.legal_moves:
91          i, j = square_to_index(move.from_square)
92          board4d[22][i][j] = 1
93
94      # add bitboard for mobility of player not on turn
95      aux = board.turn
96      board.turn = chess.WHITE if board.turn == chess.BLACK else chess.BLACK
97      for move in board.pseudo_legal_moves:
98          if board.is_legal(move):
99              i, j = square_to_index(move.from_square)
100             board4d[23][i][j] = 1
101     board.turn = aux
102
103     return board4d
104
105 class build_model(nn.Module):
106     def __init__(self, conv_size, conv_depth, dropout_rate):
107         super(build_model, self).__init__()
108         self.board4d = nn.Sequential(
109             nn.Conv2d(24, conv_size, kernel_size=3, padding=1),
110             nn.BatchNorm2d(conv_size),
111             nn.ReLU(),
112             nn.Dropout2d(p=dropout_rate)
```

```
113         )
114         for _ in range(conv_depth - 1):
115             self.board4d.add_module('conv{}'.format(_), nn.Conv2d(conv_size, conv_size, kernel_siz
116             self.board4d.add_module('bn{}'.format(_), nn.BatchNorm2d(conv_size))
117             self.board4d.add_module('relu{}'.format(_), nn.ReLU())
118             self.board4d.add_module('dropout{}'.format(_), nn.Dropout2d(p=dropout_rate))
119 
120         self.flatten = nn.Flatten()
121         self.dense1 = nn.Linear(conv_size * 8 * 8, 256)
122         self.dense2 = nn.Linear(256, 256)
123         self.dense3 = nn.Linear(256, 64)
124         self.value_head = nn.Linear(64, 1)
125 
126     def forward(self, x):
127         x = self.board4d(x)
128         x = self.flatten(x)
129         x = self.dense1(x)
130         x = self.dense2(x)
131         x = self.dense3(x)
132         value = self.value_head(x)
133         return value
134 
135 model = build_model(128, 5, 0.20)
136 model.cuda()
137 
138 #Training!
139 # Assuming that "model" has already been defined using the build_model function
140 # Load the data from the .pt file
141 data = torch.load(r"FILEPATH.pt")
142 positionsBitboard = data["positionsBitboard"]
143 evaluations = data["evaluations"]
144 
145 dataset = torch.utils.data.TensorDataset(positionsBitboard, evaluations)
146 
147 dataloader = DataLoader(dataset, batch_size=128, shuffle=True)
148 
149 # Split the data into a training set and a validation set
150 train_data, val_data = train_test_split(dataset, test_size=0.2, random_state=42)
151 train_dataloader = DataLoader(train_data, batch_size=128, shuffle=True)
152 val_dataloader = DataLoader(val_data, batch_size=128, shuffle=False)
153 
154 # Define the Adam optimizer with the specified learning rate
155 optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
156 
157 # Define the mean squared error loss function
158 loss_fn = nn.MSELoss()
159 
160 # Initialize an empty list to store the losses
161 losses = []
162 
```

```
163 # Initialize an empty list to store the validation losses
164 val_losses = []
165
166 # Initialize a counter for the number of consecutive increases in validation loss
167 consec_increase = 0
168
169 # Threshold for the number of consecutive increases in validation loss
170 threshold = 5
171
172 # How many full iterations
173 num_epochs = 50
174
175 val_losses = [float('inf')]
176
177 for epoch in range(num_epochs):
178     for positionsBitboard, evaluations in dataloader:
179         # Clear the gradients
180         optimizer.zero_grad()
181
182         # Pass the data through the model
183         value = model(positionsBitboard.cuda())
184
185         # Calculate the loss
186         value_loss = loss_fn(value, evaluations.cuda())
187
188
189         # Perform backpropagation to update the model's parameters
190         loss = value_loss
191         loss.backward()
192         optimizer.step()
193
194         # Append the current loss to the list of losses
195         losses.append(loss.item())
196
197     # After each epoch, calculate the validation loss
198     with torch.no_grad():
199         val_loss = 0
200         for val_positionsBitboard, val_evaluations in val_dataloader:
201             val_loss += loss_fn(model(val_positionsBitboard.cuda()), val_evaluations.cuda())
202         val_loss /= len(val_dataloader)
203         val_losses.append(val_loss.item())
204
205     # Print the current epoch and the current loss
206     print("Epoch: {}/{}, Loss: {:.4f}".format(epoch+1, num_epochs, loss.item()))
207
208     # Check if the validation loss has increased
209     if len(val_losses)>1:
210         if val_losses[-1] > val_losses[-2]:
211             consec_increase += 1
212         else:
```

```
213             consec_increase = 0
214
215     # If the validation loss has increased for a certain number of consecutive epochs, stop the tr
216     if consec_increase >= threshold:
217         print("Validation loss has increased for {} consecutive epochs. Stopping training.".format
218         break
219
220     # Print the predicted evaluation and the real evaluation
221     print(f'Prediction: {value[0]}')
222     print(f'Evaluation: {evaluations[0]}')
223
224 losses = losses[128:]
225
226 plt.xlim(0, len(losses))
227 # Plot the loss over time with 30% opacity
228 plt.plot(losses, alpha=0.4)
229
230 # Plot the rolling average of the losses over time
231 rolling_window = 1280
232 losses_rolling = [np.mean(losses[i:i+rolling_window]) for i in range(1, len(losses)-rolling_window
233 plt.plot(losses_rolling, label='Training Loss')
234 plt.xlabel("Iterations")
235 plt.ylabel("Loss")
236 plt.show()
237
238 # save the model
239 torch.save(model.state_dict(), "NAME.pt")
```

Program D

```
1 import chess
2 import chess.engine
3 import torch
4 from torch.utils.data import DataLoader
5 import torch.nn as nn
6 import torch.optim as optim
7 import numpy as np
8 import matplotlib.pyplot as plt
9
10 # SETUP & INATIALIZE
11 squares_index = {
12   'a': 0,
13   'b': 1,
14   'c': 2,
15   'd': 3,
16   'e': 4,
17   'f': 5,
```

```
18   'g': 6,
19   'h': 7
20 }
21
22
23 # example: h3 -> 17
24 def square_to_index(square):
25   letter = chess.square_name(square)
26   return 8 - int(letter[1]), squares_index[letter[0]]
27
28
29 def split_dims(board):
30     # this is the 4d matrix
31     board4d = np.zeros((24, 8, 8), dtype=np.int8)
32
33     # here we add the pieces's view on the matrix
34     for piece in chess.PIECE_TYPES:
35         for square in board.pieces(piece, chess.WHITE):
36             idx = np.unravel_index(square, (8, 8))
37             board4d[piece - 1][7 - idx[0]][idx[1]] = 1
38         for square in board.pieces(piece, chess.BLACK):
39             idx = np.unravel_index(square, (8, 8))
40             board4d[piece + 5][7 - idx[0]][idx[1]] = 1
41
42     # add attacks and valid moves too
43     # so the network knows what is being attacked
44     aux = board.turn
45     board.turn = chess.WHITE
46     for move in board.legal_moves:
47         i, j = square_to_index(move.to_square)
48         board4d[12][i][j] = 1
49     board.turn = chess.BLACK
50     for move in board.legal_moves:
51         i, j = square_to_index(move.to_square)
52         board4d[13][i][j] = 1
53     board.turn = aux
54
55     # set the turn dimension
56     if board.turn == chess.WHITE:
57         board4d[14] = 1
58     else:
59         board4d[14] = 0
60
61     # add bitboard for en passant
62     if board.ep_square:
63         idx = np.unravel_index(board.ep_square, (8, 8))
64         board4d[15][idx[0]][idx[1]] = 1
65
66     # add bitboards for castling rights
67     if board.has_kingside_castling_rights(chess.WHITE):
```

```
        board4d[16][7][7] = 1
    if board.has_queenside_castling_rights(chess.WHITE):
        board4d[17][7][0] = 1
    if board.has_kingside_castling_rights(chess.BLACK):
        board4d[18][0][7] = 1
    if board.has_queenside_castling_rights(chess.BLACK):
        board4d[19][0][0] = 1

    # binary channel for repetition
    repetitions = board.can_claim_fifty_moves()
    if repetitions:
        board4d[20][:][:] = 1

    # binary channel for threefold repetition rule
    repetitions = board.can_claim_draw()
    if repetitions:
        board4d[21][:][:] = 1

    # add bitboard for mobility
    for move in board.legal_moves:
        i, j = square_to_index(move.from_square)
        board4d[22][i][j] = 1

    # add bitboard for mobility of player not on turn
    aux = board.turn
    board.turn = chess.WHITE if board.turn == chess.BLACK else chess.BLACK
    for move in board.pseudo_legal_moves:
        if board.is_legal(move):
            i, j = square_to_index(move.from_square)
            board4d[23][i][j] = 1
    board.turn = aux

    return board4d

class build_model(nn.Module):
    def __init__(self, conv_size, conv_depth, dropout_rate):
        super(build_model, self).__init__()
        self.board4d = nn.Sequential(
            nn.Conv2d(24, conv_size, kernel_size=3, padding=1),
            nn.BatchNorm2d(conv_size),
            nn.ReLU(),
            nn.Dropout2d(p=dropout_rate)
        )
        for _ in range(conv_depth - 1):
            self.board4d.add_module('conv{}'.format(_), nn.Conv2d(conv_size, conv_size, kernel_siz
            self.board4d.add_module('bn{}'.format(_), nn.BatchNorm2d(conv_size))
            self.board4d.add_module('relu{}'.format(_), nn.ReLU())
            self.board4d.add_module('dropout{}'.format(_), nn.Dropout2d(p=dropout_rate))

        self.flatten = nn.Flatten()
```

```
118         self.dense1 = nn.Linear(conv_size * 8 * 8, 256)
119         self.dense2 = nn.Linear(256, 256)
120         self.dense3 = nn.Linear(256, 64)
121         self.value_head = nn.Linear(64, 1)
122 
123     def forward(self, x):
124         x = self.board4d(x)
125         x = self.flatten(x)
126         x = self.dense1(x)
127         x = self.dense2(x)
128         x = self.dense3(x)
129         value = self.value_head(x)
130         return value
131 
132 
133 def minimax(board, depth, alpha, beta, maximizingPlayer, model):
134     if depth == 0 or board.is_game_over():
135         # use the trained model to predict the evaluation of the current position
136         input_data = split_dims(board)
137         input_data = torch.tensor(input_data, dtype=torch.float32)
138         input_data = input_data.unsqueeze(0)
139         with torch.no_grad():
140             evaluation = model(input_data).item()
141         return evaluation
142 
143     if maximizingPlayer:
144         bestValue = 0.0
145         for move in board.legal_moves:
146             board.push(move)
147             value = minimax(board, depth - 1, alpha, beta, not maximizingPlayer, model)
148             board.pop()
149             bestValue = max(bestValue, value)
150             alpha = max(alpha, bestValue)
151             if beta <= alpha:
152                 break
153         return bestValue
154     else:
155         bestValue = 1.0
156         for move in board.legal_moves:
157             board.push(move)
158             value = minimax(board, depth - 1, alpha, beta, True, model)
159             board.pop()
160             bestValue = min(bestValue, value)
161             beta = min(beta, bestValue)
162             if beta <= alpha:
163                 break
164         return bestValue
165 
166 def get_best_move(board, depth, model):
167     bestMove = chess.Move.null()
```

```
168     bestValue = float('-inf')
169     alpha = 0.0
170     beta = 1.0
171     for move in board.legal_moves:
172         board.push(move)
173         value = minimax(board, depth - 1, alpha, beta, False, model)
174         board.pop()
175         if value > bestValue:
176             bestValue = value
177             bestMove = move
178         alpha = max(alpha, value)
179     return bestMove
180 
181 
182 if __name__ == '__main__':
183     # create a chess board
184     board = chess.Board("r1bqkbnr/pp1p1ppp/2n1p3/2p5/2B1P3/5Q2/PPPP1PPP/RNB1K1NR w KQkq - 2 4")
185 
186     # set the search depth
187     depth = 3
188 
189     # load the trained model
190     model = build_model(128, 5, 0.20)
191     model.load_state_dict(torch.load("NAME.PT", map_location=torch.device('cpu')))
192     model.eval()
193 
194     # get the best move
195     best_move = get_best_move(board, depth, model)
196     print(best_move)
```