NEA

Creating A Chess Engine To Play Fischer Random Chess Using Machine Learning

Dougal Craig-Wood

Dougal Craig-Wood

Contents

Analysis

Requirements for the Project

- Playable Chess and Fischer Random Chess (human versus human)
- Playable Chess against computer engine
- Playable Chess against computer engine with a neural network (machine learning)

The Problem

The problem this project seeks to solve is the lack of dedicated Fischer Random Chess platforms with a competent computer opponent. This leaves us with the three major objectives that need to be solved in order to achieve this.

The first objective is to make Chess itself; this should have a board visualisation and be able to play human vs human. Alongside this, we want to be able to check the moves are valid for each piece and check for win, lose or draw. After this Fischer random chess commonly known as chess 960 should be able to be implemented by giving different boards to start the game with.

The second objective is to make a Chess engine (a computer player). This can be achieved by creating a list of valid moves for every position and evaluating each position numerically. For example, this could be evaluation by a pieces' points. Given a list of moves and an evaluation function we could use the Minimax algorithm to make a game tree and choose the best move.

The final objective is to use machine learning with a neural network to create an evaluation function that works better than the numerical evaluation. This would likely be achieved by playing it against itself or downloading games from the internet to train it on.

Background to project

The background to this project is that for me it is a chance to learn about machine learning and AI alongside a significant boost in my programming skills, such as learning how you can apply game theoretical algorithms.

The choice of chess is that it is a well-documented game alongside there being lots of theoretical and practical knowledge to help with implementation of the AI. Furthermore, the choice of Chess 960 is also to eliminate the need for specific opening strategies while making the game more random. This is as Chess 960 is where the backrow for each player is randomized in a specific manner causing many more combinations at the start of the game; eliminating the use of book openings therefore adding a variable element to chess.

End-Users and Supervisor

The end-users will be anyone who enjoys the game of chess, as the finished project should be a demonstration and insight into the workings of more complicated chess engines such as Stockfish.

For this project I selected a specific individual with lots of experience in the game of chess as the main client. Their background in chess but not in Chess 960 makes them an ideal client as they want to learn about the game as well and will have different needs about the project.

The supervisor of this project will be my father as he is well suited to be the supervisor as he has a significant background in computer science, such as knowing many key concepts to this project. As he is family, he is easy to access for advice on the project, while I have asked him to be critical rather than supportive.

Client interview

Q1: What level of AI would you like to play?

“I would like to play a level where it is an enjoyable level of chess that is challenging and difficult for me, but not at a level that plays perfectly and would beat me every time.”

Q2: Are there any specific User Interface components you would like?

“I would like to be able to drag pieces around the board and highlight which available squares I can move to with the piece I have selected.”

Q3: What performance or time frame for moves are you expecting?

“I like to play my games of chess within a 10-minute timer usually so I would expect the AI to play within these times. I would expect less than 20 seconds.”

Q4: Would you like game metrics and analysis?

“It would be interesting to see game metrics for example the board favour after the game has finished but I feel during the game it would be an interruption.”

Q5: How would you like errors in chess to be handled?

“I would like errors to not be possible, so the game would ignore it and reset the piece back to its original position.”

Q6: Any other comments?

“I would like to be able to play standard chess against the AI and also to be able to play against another controllable character if I want to play against one of my friends.”

Possible Problems

The first objective of building a Chess game seems straight forward however there are a lot of rules in chess that could complicate the process such as if a position arises three times in a game either player can claim a draw.

The second objective is more complicated however it is a well-studied area of computer science. A simple algorithm like the Minimax algorithm can use a lot of CPU to give satisfactory results and take a lot of time. Solutions to this may be optimising the algorithm using things such as Alpha-beta pruning or switching to a faster language such as C or even multi-threading. However, to simplify the third objective we want to make this as simple to understand as possible as well as half the point of making an evaluation function with machine learning is to speed up the process of numerical evaluation, therefore making optimisation of the numerical algorithm less important.

Problems with the third objective may arise if good data is not collected to run the machine learning on. Without a good set of data, the machine learning will be ineffective and so collecting thousands of pre-evaluated boards from websites such as Lichess will be necessary for training. Another problem with having lots of data is that it will take a lot of time to train and will require a powerful GPU or renting one.

This project will likely be done in only Python, and it will use many external source of data such as Chess games. As well as external Chess engines for testing and training.

Research

To do this project we needed to find out about specific Chess rules that need to be implemented, major algorithms and the general workings of neural networks and how they would suit this project.

- Creating chess itself:
  The goal is to create something like a chess module. It should have a list of functions such as:
  - Display board
  - Legal moves
  - Move
  - Checks for:
    - Check
    - Checkmate
    - Stalemate
    - Castling rights
    - Possible En Passants
    - Repetition rules

  Chess has many rules but importantly we need rules around each move which include the checks listed above as well as type of piece and position of piece that is being moved.

  Without displaying the board, the first goal would be to get a working chess game that would look like the figure below. Alongside this the graphical representation can also be referenced from website like Chess.com and Lichess.

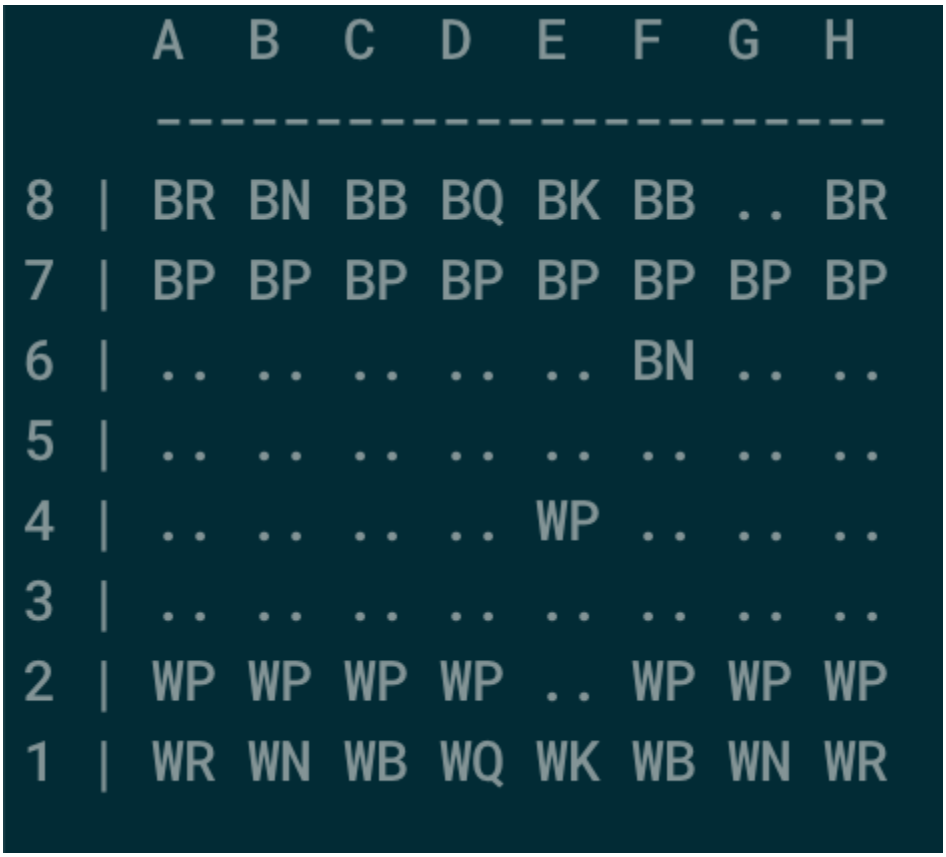


Figure 1/primitive chess board/ (Dirk Hoekstra, n.d.)

- Minimax algorithm
  The Minimax algorithm allows us to go through the game tree while maximizing our score (evaluation of the board in our favour) while assuming the opponent does the same.

  As chess is not a solved game, while both players play perfectly it can still be a win, lose or draw. This means while using Minimax with an infinite depth we are guaranteed to win. At each stage of the tree as seen below each side aims to increase their score as much as possible, and the goal is to get to positive infinity as that would be a win.

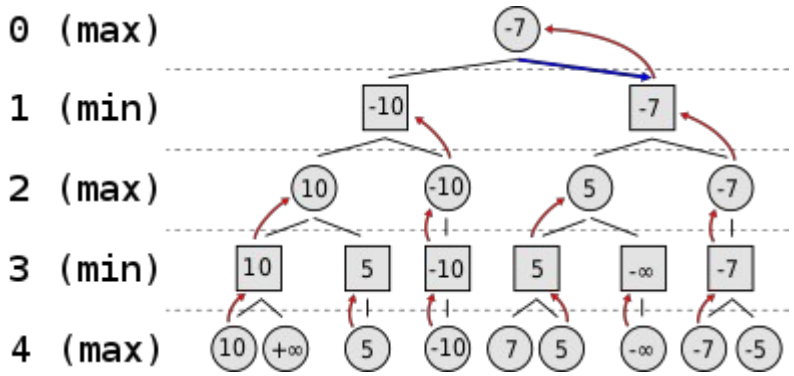


Figure 2/ Minimax Diagram/ (Minimax, n.d.)

- Alpha-beta pruning
  Alpha-beta pruning is a search algorithm used to minimize the amount of time the Minimax algorithm spends looking for good moves. Alpha-beta pruning decreases the number of nodes that are evaluated in the game tree when at least one possibility makes the move worse than another move than has previously been evaluated.
- Chess evaluation functions
  To create an evaluation function, we at least need to know the value of the pieces themselves and the material balance.
- Using Stockfish from Python
  This is simpler than it seems as there is a Stockfish module which can be used to get best moves or check a move is correct. This means Stockfish will be extremely easy to use to check the evaluation of the evaluation functions.

- Machine learning
  - Single output node in the output layer to represent the evaluation score
  - Representing input data
    Lichess' database which stores all the games played from each month, the games in the database are stored as a .pgn file which means Portable Game Notation. In the PGN file there is a list of all the moves played along with some general information. This means standard chess notation will have to be understood by the program.

```
[Event "F/S Return Match"]
[Site "Belgrade, Serbia JUG"]
[Date "1992.11.04"]
[Round "29"]
[White "Fischer, Robert J."]
[Black "Spassky, Boris V."]
[Result "1/2-1/2"]

1. e4 e5 2. Nf3 Nc6 3. Bb5 {This opening is called the Ruy Lopez.} 3... a6
4. Ba4 Nf6 5. O-O Be7 6. Re1 b5 7. Bb3 d6 8. c3 O-O 9. h3 Nb8 10. d4 Nbd7
11. c4 c6 12. cxb5 axb5 13. Nc3 Bb7 14. Bg5 b4 15. Nb1 h6 16. Bh4 c5 17. dxe5
Nxe4 18. Bxe7 Qxe7 19. exd6 Qf6 20. Nbd2 Nxd6 21. Nc4 Nxc4 22. Bxc4 Nb6
23. Ne5 Rae8 24. Bxf7+ Rxf7 25. Nxf7 Rxe1+ 26. Qxe1 Kxf7 27. Qe3 Qg5 28. Qxg5
hxg5 29. b3 Ke6 30. a3 Kd6 31. axb4 cxb4 32. Ra5 Nd5 33. f3 Bc8 34. Kf2 Bf5
35. Ra7 g6 36. Ra6+ Kc5 37. Ke1 Nf4 38. g3 Nxh3 39. Kd2 Kb5 40. Rd6 Kc5 41. Ra6
Nf2 42. g4 Bd3 43. Re6 1/2-1/2
```

Figure 3/Example PGN/ (Wikipedia, Portable Game Notation, n.d.)

- Structure of neural network
  The neural network will have an input of each possible board and one output between -1 and +1 representing the favourability of each player to win.

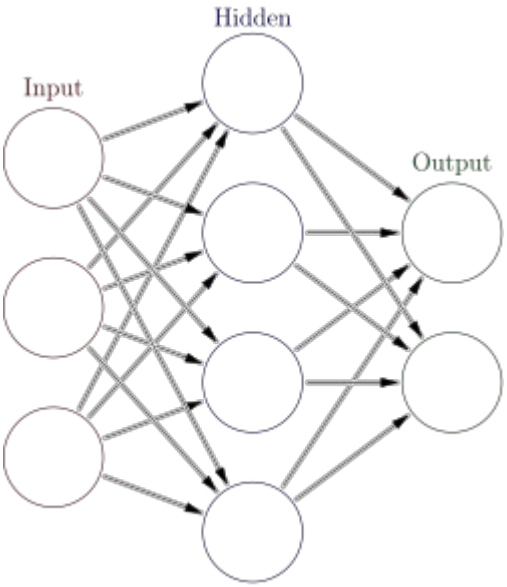


Figure 4/Neural network architecture/Wikipedia

- Map output to understandable data
  An output could be the evaluation of the board, which can be used to select the next best move after evaluating the possible boards.

Conclusion to Research

There are many complex components to the project but in conclusion I am most concerned about how I will make chess itself and then how I will interface my version of chess with Stockfish for final testing. This also relates to how I will train the AI and how I will process the data so that it can interact with my chess program too. Therefore, I think the main concern is interaction between the different parts of the project, especially the training data and the AI.

Database

A database can be implemented into this project through a book of openings and a collection of evaluated boards. Such as a database of PGN or FEN files collected from Lichess.

Proposed coding style

For Python we will use PEP 8 coding style to make the code consistent. We will use rules about:

- Indentation
- Maximum Line Length
- Imports
- Whitespace
- Comments
- Naming Conventions
- Function and Method Definitions
- Blank Lines
- Encoding
- Imports Formatting
- Maximum Imports Per Line
- Whitespace in Expressions and Statements

To check this form, we will use a PEP 8 linter to make sure these rules are followed. Alongside this, it will be written using the IDE Visual Studio Code, which is a popular professional IDE; this also has extensions such as PEP 8 built in.

Example of PEP 8 Code:

```python
import os  # STD lib imports first
import sys  # alphabetical

import some_third_party_lib  # 3rd party stuff next
import some_third_party_other_lib  # alphabetical

import local_stuff  # local stuff last
import more_local_stuff
import dont_import_two, modules_in_one_line  # IMPORTANT!
from pyflakes_cannot_handle import *  # and there are other reasons it should be avoided # noqa
# Using # noqa in the line above avoids flake8 warnings about line length!


_a_global_var = 2  # so it won't get imported by 'from foo import *'
_b_global_var = 3

A_CONSTANT = 'ugh.'


# 2 empty lines between top-level funcs + classes
def naming_convention():
    """Write docstrings for ALL public classes, funcs and methods.
    Functions use snake_case.
    """
    if x == 4:  # x is blue <== USEFUL 1-liner comment (2 spaces before #)
        x, y = y, x  # inverse x and y <== USELESS COMMENT (1 space after #)
    c = (a + b) * (a - b)  # operator spacing should improve readability.
    dict['key'] = dict[0] = {'x': 2, 'cat': 'not a dog'}


class NamingConvention(object):
    """First line of a docstring is short and next to the quotes.
    Class and exception names are CapWords.
    Closing quotes are on their own line
    """
```

Figure 5/pep 8 example/ (Bronosky, n.d.)

Pre-Modelling

System overview, concluded from research:

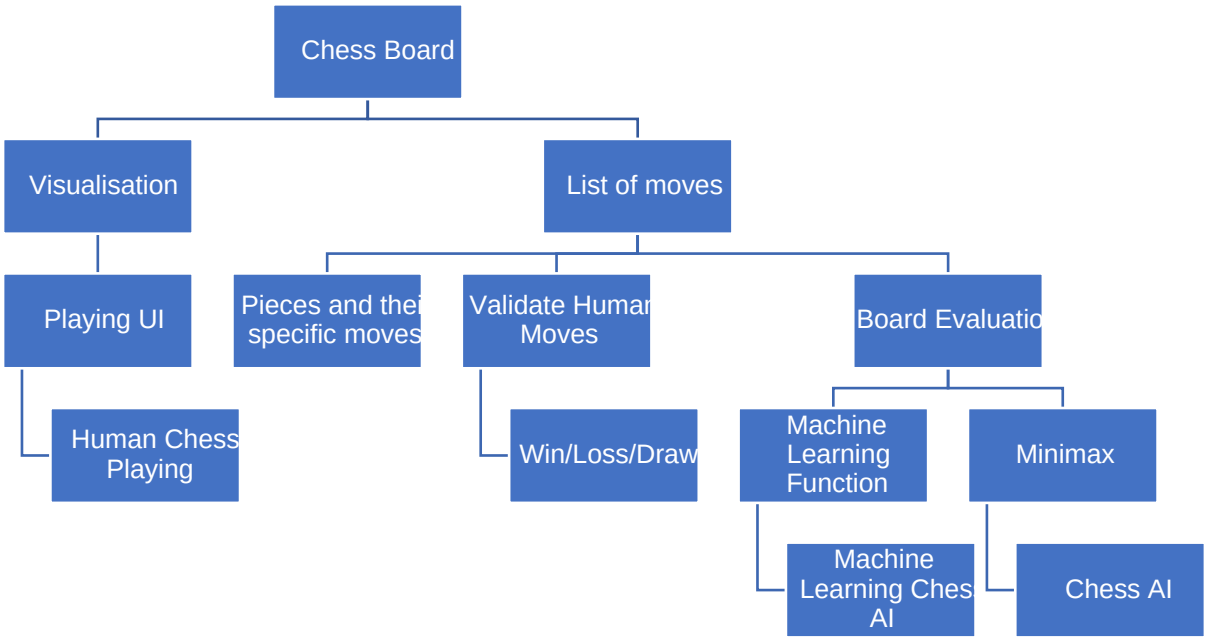


Figure 6/ System Overview Diagram

Possible Critical Path

The three objectives depend on each other and should be done in order, and so the first objective is the most important and should be prioritized first as well as having to be correct, in terms of development it is quite a simple project.

Possible Gantt Chart:

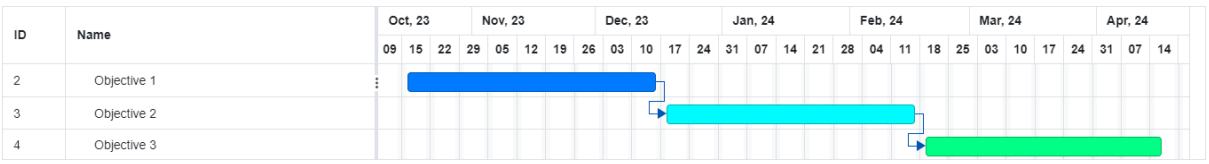


Figure 7/ (Timings are not to scale or accurate)

Objectives

1. Create a chess Board that can play chess between two human players, this should take about 8 weeks.

    1.1 Implement a text-based chess board that displays all pieces in a standard position, row and column numbers/ characters should also be displayed.

1.2 Create a move function which allows pieces to moved only into valid positions which accepts algebraic notation in the form square to square.
1.3 Add game checks and rules such as castling, En Passant, check, checkmate and drawing positions.
1.4 Implement a graphical representation of the current board that has correct piece and square colours, with piece images taken from the internet, which can also display chess 960.
1.5 Allow pieces in the GUI to be dragged and dropped into valid positions that are highlighted in a different colour to default squares.
1.6 Flip the board for current player so both players play upwards.
1.7 Represent board state such as win, loss, draw and stop the game at this point.

2. Create a playable engine with a simple Evaluation using minimax and Alpha-beta pruning, taking another 4 weeks.

2.1 Implement a simple evaluation function that can evaluate boards with a numeric value.
2.2 Using the simple evaluation, implement minimax from pseudocode making a best move function at a specified depth.
2.3 Adjust the GUI to allow different players so that the best move function can act as a different player.
2.4 Implement the alpha-beta pruning algorithm to replace minimax which will also result in a best move function.
2.5 Show the Alpha-beta pruning algorithm is more efficient than minimax.

3. Using the previous algorithms and creating a new evaluation function with machine learning, be able to beat stockfish on 2000 Elo. Including time to complete the project, this should take about 10 weeks to 15 weeks.

3.1 Find training data and convert it into a data type suitable for a neural network.
3.2 Create a simple neural network with a few layers and using a portion of the training data to prove the evaluation function will return somewhat correct evaluations.
3.3 Expand the layers of the neural network and train it using a larger amount of data to create an accurate evaluation function as compared to Stockfish.
3.4 Convert boards from internal representation to a format for input for the neural network to calculate evaluations while recursing through the game tree.
3.5 Create classes that can be passed into the GUI for each combination of evaluation type and tree searching technique.
3.6 Install Stockfish and convert internal notation to Stockfish notation in order to play many games against varying Stockfish levels to estimate the strength of play.

Design

Overview

The Project will be done in stages, first the implementation of a chess board, then using classes and implementation of multiple types of players: human (and random), minimax AI, and neural network AI. Then each class will be worked on individually in the order stated. With each objective corresponding to a type of player.

The design will be split into two sections, the main chess component, and the UI to go along with it. The idea of this is to create a working chess game before making it playable/user friendly, as this will assist with debugging and simplify the problem.

Some sections of the programming will be done using the methodology of test-driven development (TDD), this will be useful as it is an effective way of developing complex code with certainty that it doesn't contain bugs. The process starts by writing a test, writing code to pass the test then tidying up the code after, you can repeat this to build the required functionality one small step at a time. This process is shown in the Diagram below.

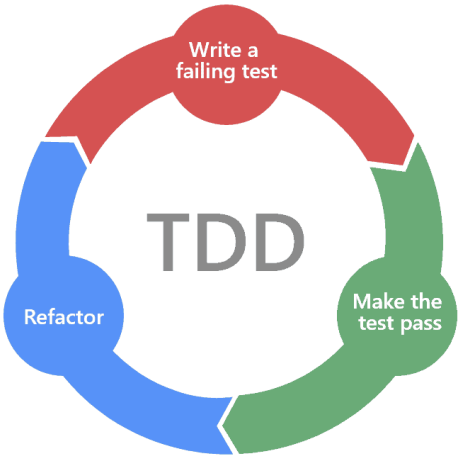


Figure 8/TDD/ (Why Test Driven Development, n.d.)

Furthermore, TDD is useful as to when the code will have to be inevitably optimised to run faster. I can rewrite the code while and if it contains bugs, it will not pass the test, so rewriting and optimising will be much easier.

Programming Paradigms

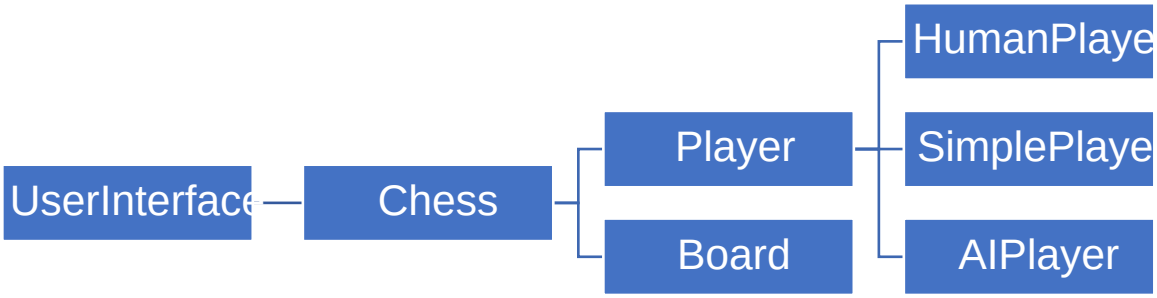
The project will be based on Object-Orientated Programming (OOP), this is to essentially split things that are separate such as the board and the type of player. This will allow an easily interchangeable game where you can play in the combination you want as well as neat programming due to the different objects being in different files.

Critical Path

Project does not have units of work to be done in parallel so the project will take as long as it takes to complete all the units/objectives, critical path is the same as mentioned in the analysis.

Structure/hierarchy chart

Each box represents a class that might be in the final code.



Design of Board

I have decided to keep the board as a one-dimensional array of 64 characters, laid out as seen below. This is opposed to bitboards where there would be a 64-digit integer for each piece-type and colour so there would be 12 different bitboards. This method is very memory efficient, but I decided to use an array as it much simpler to implement and easier to understand while programming the moves.

Finalised Layout of Board

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

a8	b8	c8	d8	e8	f8	g8	h8
a7	b7	c7	d7	e7	f7	g7	h7
a6	b6	c6	d6	e6	f6	g6	h6
a5	b5	c5	d5	e5	f5	g5	h5
a4	b4	c4	d4	e4	f4	g4	h4
a3	b3	c3	d3	e3	f3	g3	h3
a2	b2	c2	d2	e2	f2	g2	h2
a1	b1	c1	d1	e1	f1	g1	h1

Array index above and board square below.

Design of Piece Moves

Pieces move differently depending on what they are, each one will have a function which relies on the move function and a few checks such as if there is space to move to, if a take is valid and a few others.

As seen below, moves will be programmed in vectors, apart from the pieces that do “linear moves” like a bishop which can move any distance linearly. These pieces will also have to have checks on each piece that it passes through, also seen with the queen below.

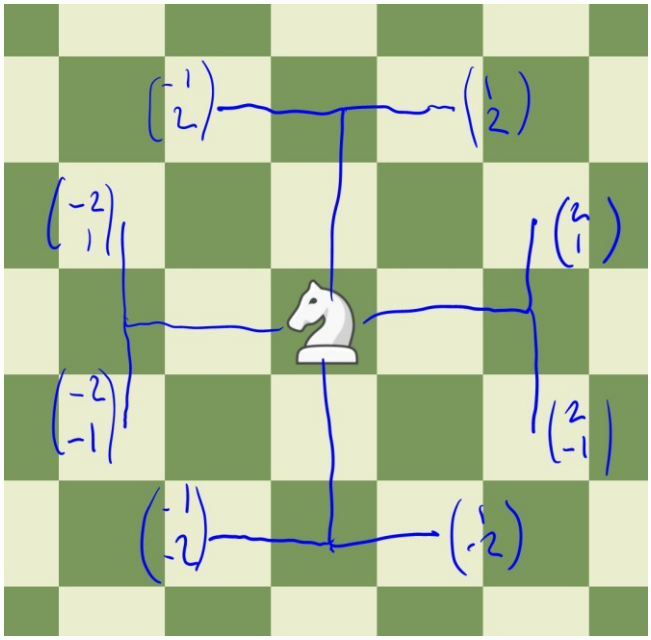


Figure 9/Knight and its vectors/Me

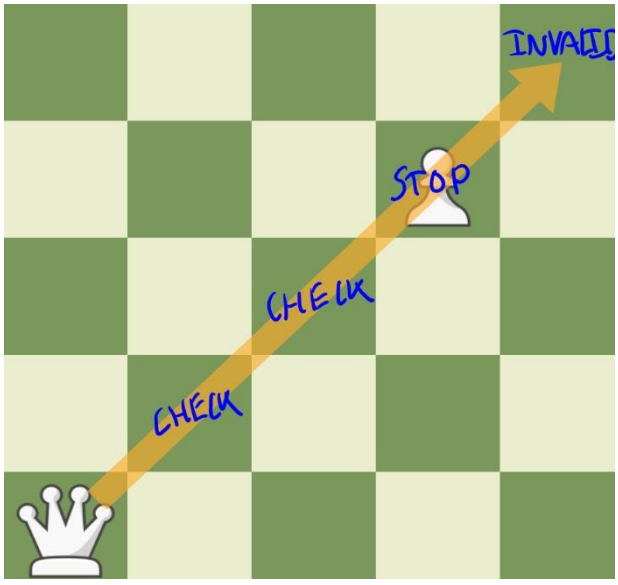


Figure 10/Queen - "linear moves"/Me

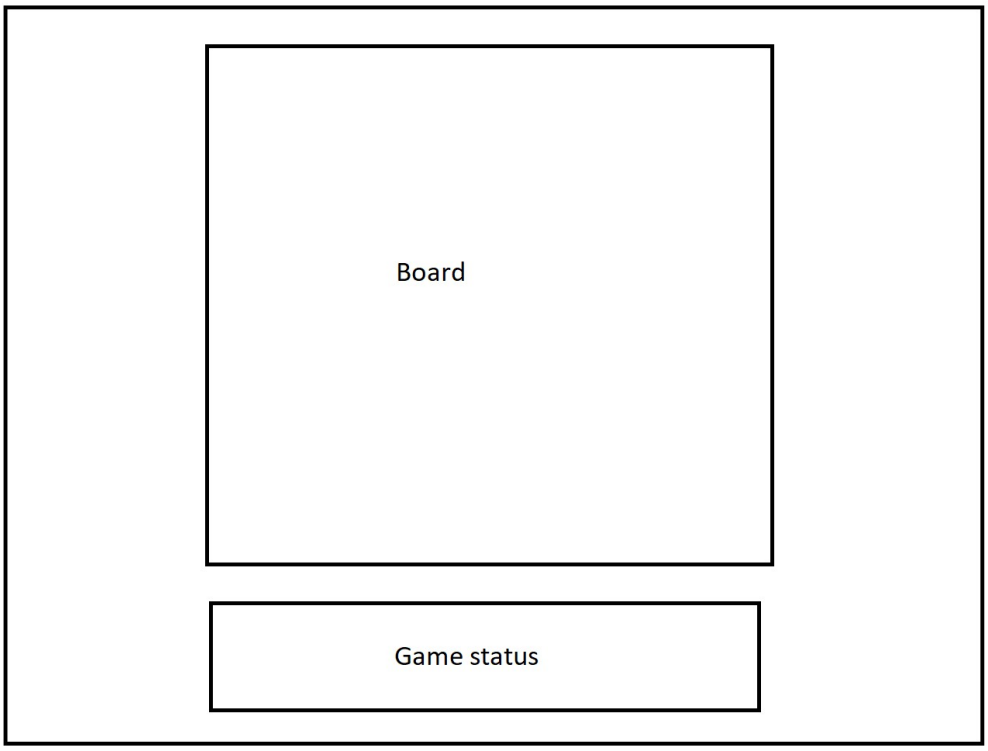
Design of Board Checks

As stated in the Analysis, there will need to be checks for specific board states such as:

- Checkmate
  A checkmate is where the king has no valid moves and is also in check. I think coding this will be as simple as using the Check function on all the possible places the king could move and where it is currently, if all return False, then it is checkmate.
- Stalemate
  A stalemate is where one side has no valid moves and is not in check. This will be as simple as using the Check function and the List all valid move function.
- Checks
  A Check will be when the position of the king is in the list of valid moves for the other side. After a Check, the King will have to move out of chess, this can also be done by moving a piece to block the check. A move into check will also be discounted as a valid move.
- En Passant
  En Passant is where a pawn moves forward 2 of the starting rows and there is an opposite-coloured pawn parallel with it, the opposite-coloured pawn can take the pawn that moved diagonally. As all pieces will be programmed to move forward once, then twice this will be checked on the first time it moves forward, if the take is possible at this point, the En Passant is possible.
- Castling Rules
  For castling there will need to be flags for whether the rook and king have moved, if these flags are false and there are no pieces in between the two, a castle is possible. These flags will have to be implemented into the Board class, not inside the piece's functions.
- Repetition Rules
  - Threefold repetition rule, this allows a draw when the same state is reached 3 times, these two can be solved with a counter and a list, storing the half move and the board state and then they can just be checked against. In actual chess a draw can be claimed meaning it's not necessary, but for sake of simplicity the draw will be forced if this happens (this is also what chess.com does).

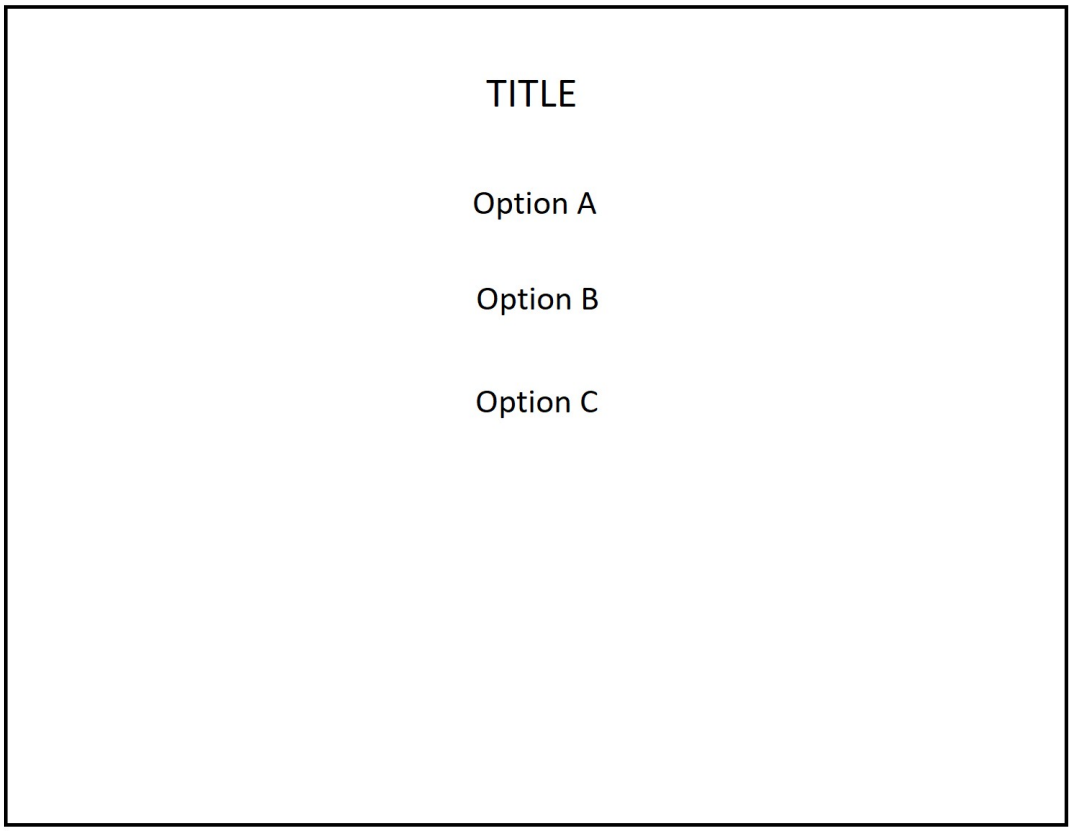
Design Of GUI

After implementing the board, to make the rest of the program more visual and easier to play, I will add a GUI. Firstly, we need to plot an 8x8 board then add the pieces which will likely be taken from the internet. Then a drag and drop algorithm will need to be implemented alongside a function that determines which square e.g. a2 you're dragging to and from, this will then return a new board to the backend. In its simplest form it will look like:



To plot the board, it should be as simple as running a loop through the board array, with some if statements saying whether it is which type of piece then plotting it from left to right as it reads the board. For this it'll do some co-ordinate maths with a scale of the screen and an offset to get the squares in the right place.

In the GUI the type of game and players will also be chosen with a click through menu. With the selection and below some options, which will look something like:



Titles and options:

Title	Options
Choose white player	Human, simple evaluation player, AI evaluation player
Choose black player	Human, simple evaluation player, AI evaluation player
(If computer player is picked) Choose depth	1,2,3
Chess 960	True or False

Another feature to the GUI must be to flip the board for a black player, this is important as playing chess the wrong way around with the pieces the wrong way around is very confusing. Alongside this, if it is a white computer player and a black human player, the board can stay flipped the whole time.

Design of Simple Computer Player

For each game in the game tree during the minimax algorithm, the board needs to be evaluated. Such evaluation function was formulated by Claude Shannon in 1949:

```
f(p) = 200(K-K')
       + 9(Q-Q')
       + 5(R-R')
       + 3(B-B' + N-N')
       + 1(P-P')
       - 0.5(D-D' + S-S' + I-I')
       + 0.1(M-M') + ...

KQRBNP = number of kings, queens, rooks, bishops, knights and pawns
D,S,I = doubled, blocked and isolated pawns
M = Mobility (the number of legal moves)
```

Figure 11/Shannon evaluation/ (Chess Programming Wiki, n.d.)

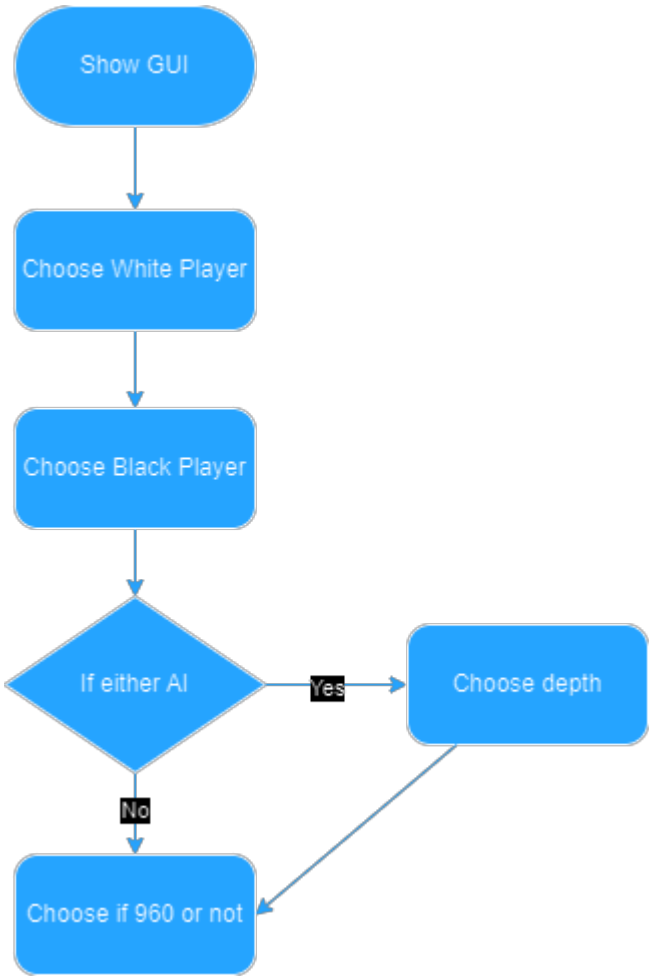
This evaluation gives a piece-wise evaluation consisting of a weight for each set of pieces and some other information about the game. In addition to this other information can be added such as Piece-Square tables, an example of this is centre distance where you want the pieces to control more of the centre of the board, so you assign higher weight on central pieces. Central distance defined in C:

```
const int arrCenterDistance[64] = { // char is sufficient as well, also unsigned
  3, 3, 3, 3, 3, 3, 3, 3,
  3, 2, 2, 2, 2, 2, 2, 3,
  3, 2, 1, 1, 1, 1, 2, 3,
  3, 2, 1, 0, 0, 1, 2, 3,
  3, 2, 1, 0, 0, 1, 2, 3,
  3, 2, 1, 1, 1, 1, 2, 3,
  3, 2, 2, 2, 2, 2, 2, 3,
  3, 3, 3, 3, 3, 3, 3, 3
};
```

Figure 12/Centre Distance/ (Chess Programming Wiki, n.d.)

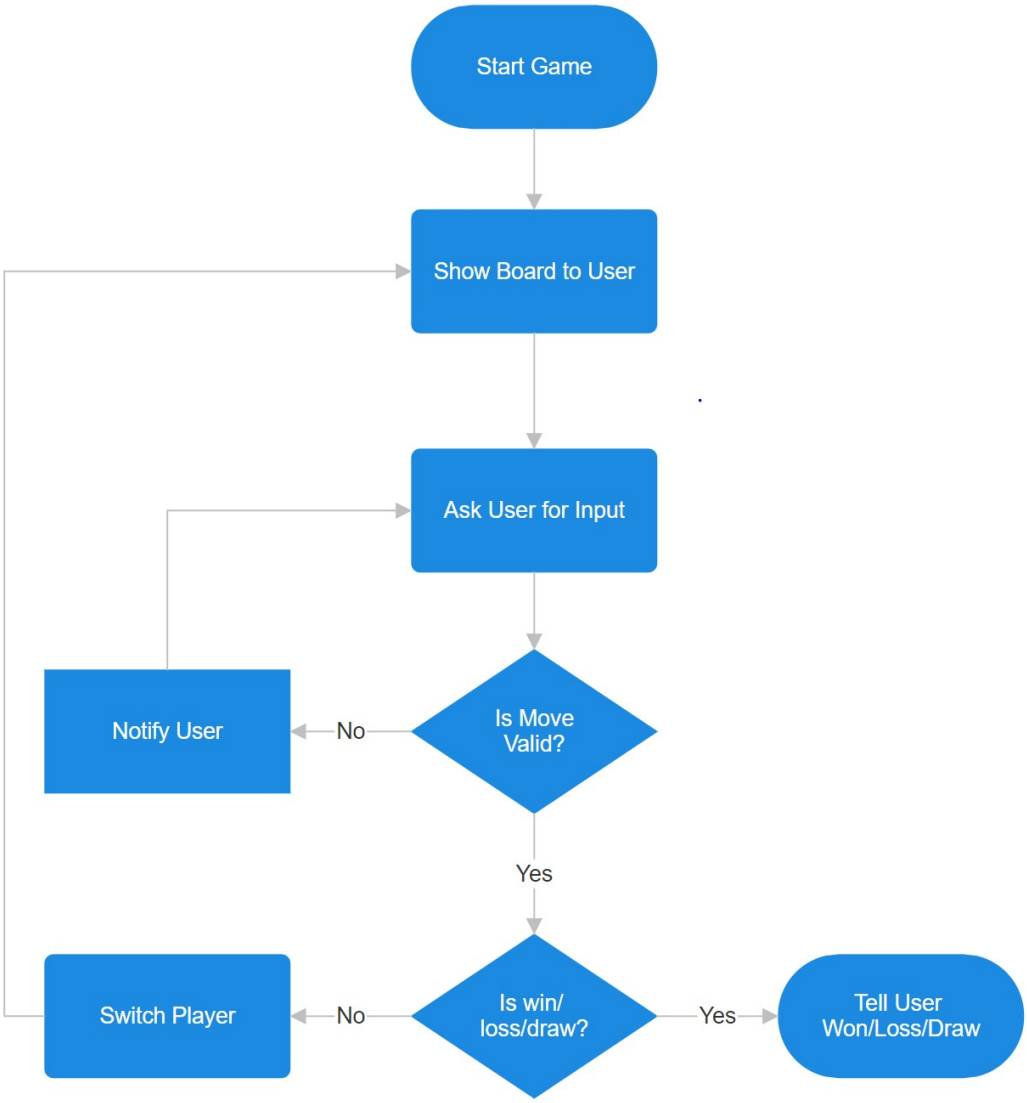
System Flowchart

Feature selection:



Game loop:

Starts game with selected features and players.



Algorithms

Some Important algorithms include:

- The algorithm for finding valid moves. Which will be created by finding all the pieces' moves and checking against this to see if the move the player wants to do is inside this list. e.g. For each white piece, generate a list of moves for that piece.
- Evaluation algorithms, which include stage 2 and 3, may include:
    - Minimax algorithm, to search through a game tree and find outcomes we want.
    - Alpha-Beta pruning algorithm to increase efficiency of the Minimax algorithm, optimising the tree searching.
    - Backpropagation and other algorithms for neural networks, which will be covered more in depth in the implementation section.

Data Structures

The board will be represented as an array of 64 characters, in each entry there will either be a "." character or a letter representing a piece, which will be capitalized if black and non-capitalized if white. This will be in algebraic notation, with placement 0 being a8, in the top left like a chess board,

and placement 63 being h8 in the bottom right. There will also be another list storing information as to which element is a white or black square as well as another 64-element array storing the names of each square in algebraic notation. These will be part of the Board class and will be called: Board, Colours and Notation.

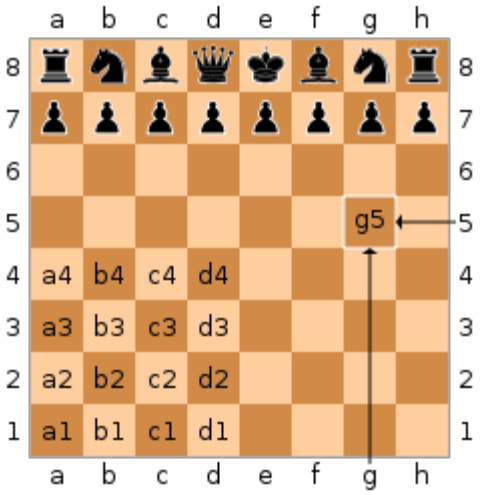


Figure 13/Algebraic Notation/ (Wikipedia, Algebraic Notation, n.d.)

Piece	White	Black
Pawn	p	P
Knight	n	N
Bishop	b	B
Rook	r	R
Queen	q	Q
King	k	K

Data Processing

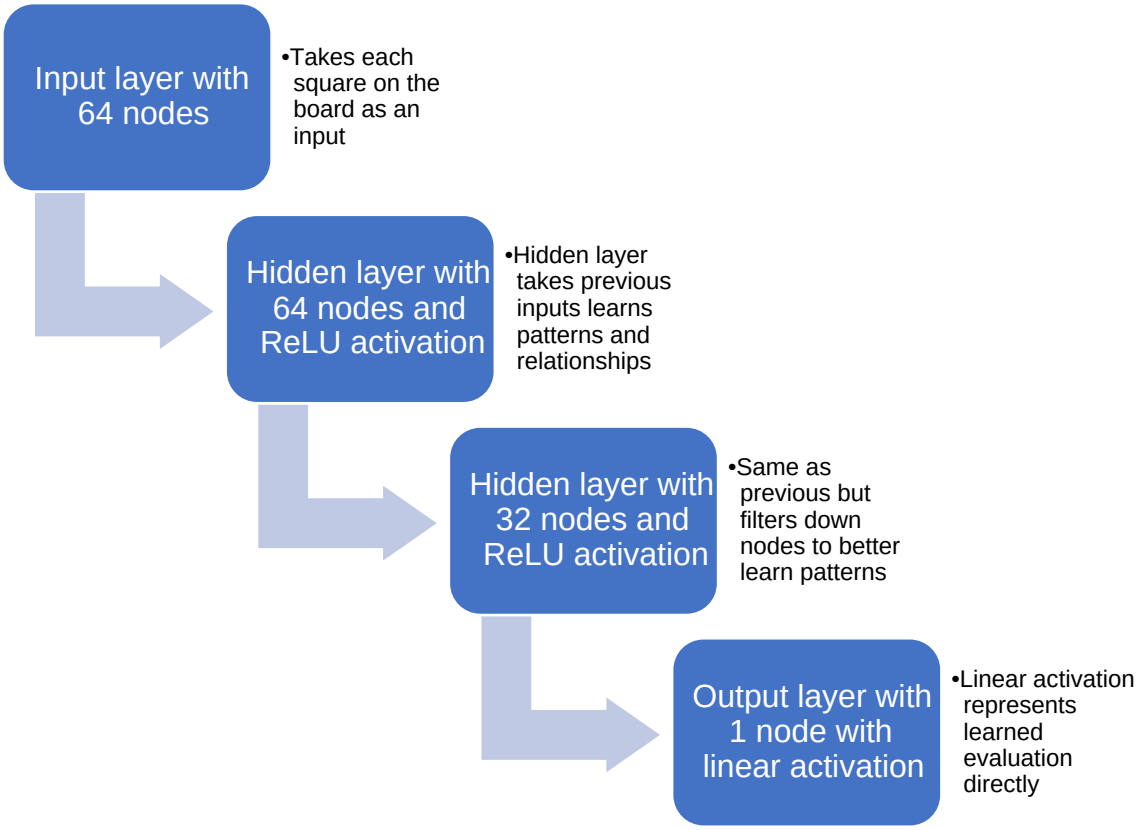
Data processing throughout the program will be simple besides converting my board notation into FEN notation and probably converting FEN notation into something that can be read by the training algorithm.

Besides this all the data from the Lichess database will also have to be processed to be used as training data.

Neural Network

Topology

An example topology may be:



Mathematical Functions

For each node in a layer (neuron) they will receive input from a previous layer, apply a weighted sum and bias then pass the result through an activation function. To do this, the parameters need to be trained with a loss function and backpropagation through training.

Weighted sum and bias:

Each neuron calculates a weighted sum of its inputs and adds a bias to stop it returning 0 if the inputs are 0. The weights and the bias let the neurons learn the transformation/ pattern on the inputs, like what a piece is. It is defined as:

$$z = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n + b$$

Where w is the weight, x is the input and b is the bias. The values of w and b are learned and optimised through training.

Activation functions:

As seen in the topology, the activation functions will be ReLU and linear. ReLU or rectified linear unit which is defined as:

$$ReLU(x) = \max(0, x)$$

Or graphically:

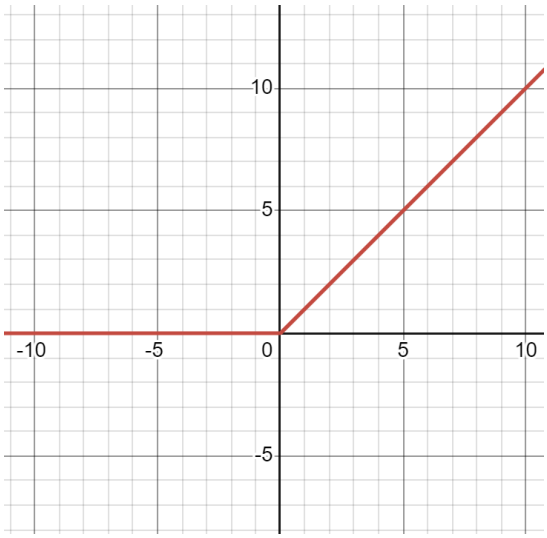


Figure 14/ReLU/ (Desmos, n.d.)

ReLU is used to introduce non-linearity which is needed for learning complex patterns as without this the model would act as a linear transformation like Model(x)=2x. ReLU is likely the most popular activation function due to its simplicity as well as it eliminates the vanishing gradient problem as it does not shrink large values as it has a gradient of 1 in positive regions and 0 in negative (also simple derivative is efficient for large networks).

Linear activation on the other hand is just a linear transformation and defined as:

$$f(x) = x$$

We use the linear activation in the chess model as we want it to return a float of the evaluation. Where this float is a continuous scalar value which directly represents the networks evaluation of the board. This is opposed to SoftMax which is also a very common output activation function, which converts values into probabilities of specific categories e.g. win/loss/draw.

Loss function:

The loss function measures how far the prediction of the model is to the actual training data values. This gives a loss which can be minimized so that the predictions are as accurate as possible. A common loss function is Mean Squared Error which is calculated with:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Where y and $\hat{y}$ are the actual value and predicted value. Mean Squared Error loss due to the squared nature penalizes big differences between the values.

Another loss function that may work better if there are outliers in the dataset is the Huber Loss function. This combines Mean Squared Error and Mean Absolute Error and helps to reduce the impact of large outliers without fully ignoring them. In the case of chess, the evaluations for checkmate might be very large and so this function could be very useful.

$$L_\delta(y, f(x)) = \begin{cases} \frac{1}{2}(y - f(x))^2 & \text{for} |y - f(x)| \le \delta, \\ \delta\,(|y - f(x)| - \frac{1}{2}\delta), & \text{otherwise.} \end{cases}$$

Figure 15/Huber loss/ (Wikipedia, n.d.)

Where y is the actual value and f(x) is the predicted value.

Backpropagation:

Backpropagation is a complicated process which is repeated for each training iteration known as an epoch and allows the network to adjust its parameters (the weights and biases) so that it returns a lower loss according to the loss function. It works through multiple steps:

1. Using the current parameters the network does a forward pass and computes the output.
2. Computes the loss using a specific loss function (compares output to desired output).
3. Calculates gradient of the loss for each parameter which gives an estimate for how much the parameter should change to achieve the desired output. This starts from the output layer and works backwards.
4. The weights and biases are updated according to an algorithm such as Gradient Descent or Root Mean Square Propagation.

After a specified number of epochs and training data this process should return a model with well-trained weights and biases allowing for an accurate prediction of the evaluation.

Hardware Selection

Most of the programming will be done on my school given Microsoft Surface Pro. However, this may not be powerful enough for training the neural network. For this Google Colab may be used to train the AI with access to Nvidia A100 graphics card that will allow millions of data points to be used for training.

Test Plan

All Classes will be tested with specific unit tests, alongside this we will have other important tests as listed below. In addition, tests will be shown in the testing video.

Tests for Objective 1

TEST NUMBER	OBJECTIVES	TEST	PURPOSE	EXPECTED OUTCOME
1	1.2,1.3	Passes all tests in Test Board file	Checks all functions and if they do what they should	All pass
2	1.1	Passes all tests in Test UI file	Checks all functions in UI	All pass
3	1.1	UI testing: -Valid board -Playable moves -Does not take invalid moves	Shows UI functionality	-Valid -displays moves -asks to redo moves
4	1.3,1.4,1.5,1.6,1.7	GUI testing: -Valid chess board -Selection menu, each option, check if 960 boards are valid -Shows valid moves, Cannot make invalid move -Flips board if human is playing on black -Shows if which side has win loss or drawn -Exits properly -En Passant	Tests all the functionality of the GUI	-Valid -All work (chess 960 valid) -All valid -Flips correctly -Correctly displays win/loss/draw -exits -Valid En Passant

Tests for Objective 2

TEST NUMBER	OBJECTIVES	TEST	PURPOSE	EXPECTED OUTCOME
1	2.1	Passes all tests in Test Simple Evaluation file	To show evaluation works as expected	All pass
2	2.2	Passes all tests in Test Minimax File	To show minimax works as intended with Simple Evaluation	All pass

3	2.4,2.5	Passes all tests in Test Alpha-beta Pruning File	To show alpha-beta pruning takes less time than minimax	All pass

Tests for Objective 3

TEST NUMBER	OBJECTIVES	TEST	PURPOSE	EXPECTED OUTCOME
1	3.1,3.2,3.3	Passes all tests in Test AI evaluation File	To show AI evaluation gives similar evaluations as stockfish	Within acceptable range of stockfish
2	3.5,3.6	With AI evaluation test Alpha-beta pruning and minimax Test files	To show it still works with the other form of evaluation	All pass

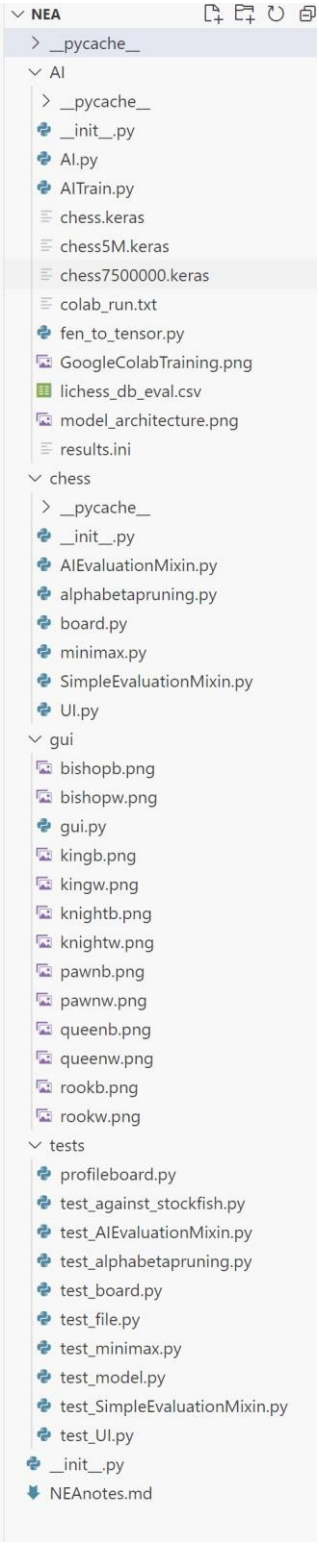
Final Tests

TEST NUMBER	OBJECTIVES	TEST	PURPOSE	EXPECTED OUTCOME
1	1,2,3,2.3	Test that each option in the selection menu is playable	Shows all parts o the project work	All works
2	3,3.6	For the AI evaluator at each depth play against stockfish at 500,1000,1500,2000 and 2500 rating	To give a rough estimate of Standard skill levels	-depth 1 beats ratings below 1000 -depth 2 beats ratings below 1500 -depth 3 beats rating below 2000

Implementation

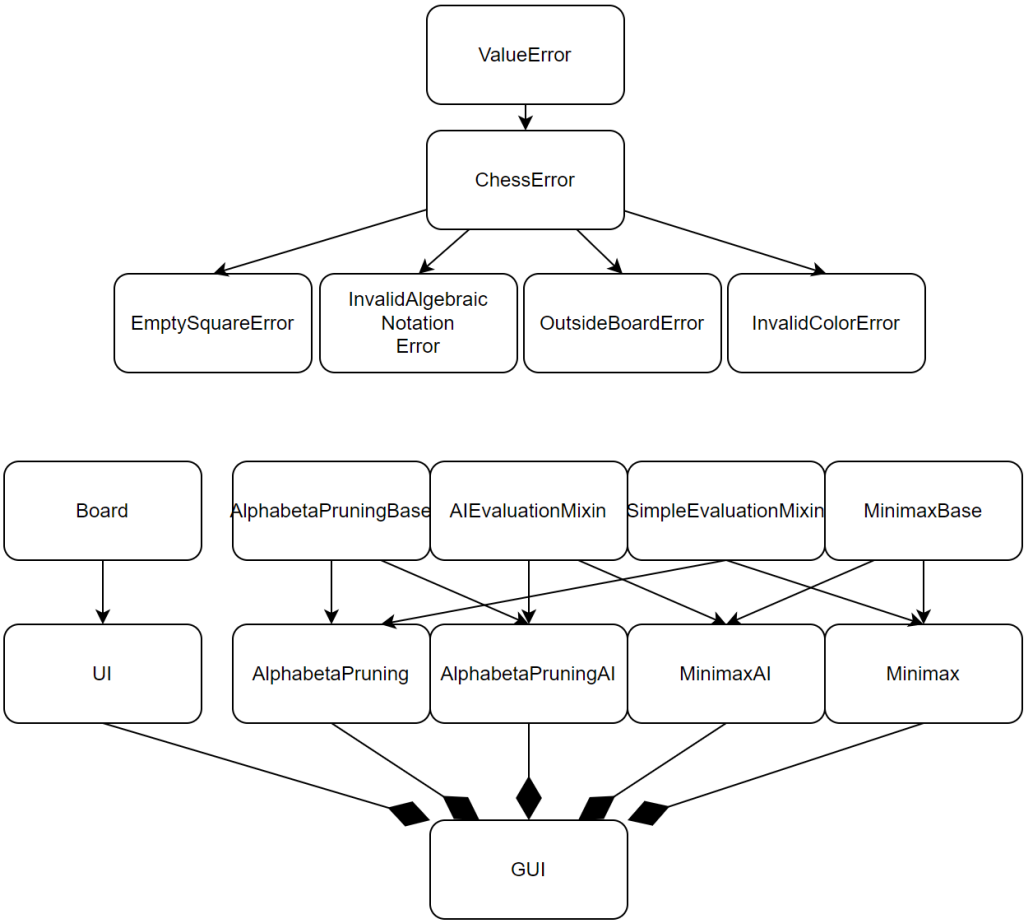
File Structure

Main chess files, the GUI, AI components and test files are in separate folders.



Post-modelling

Class hierarchy:



Skills

List skills with table and page numbers from specification.

Skills	Usage	Code Reference (Page number)
OOP 1.Classes 2.Inheritance 3.Composition 4.Polymorphism 5.Method types 6.Multiple Inheritance	1. Classes are used to encapsulate mos main functions and error classes: -all unit test files -mixins -minimax, alpha-beta pruning -GUI, UI, board -ChessError and its subclasses 2. -ChessError and subclasses -UI 3. GUI 4.Polymorphic call of best move in GUI (can be any of 4 different engines) 5. static methods 6. With mixins: -AlphabetaPruning, AlphabetaPruningAI	1. -93 -69,70 -71,72 -74,68,53 -53 2. -53 -68 3. 74 4. 74 (line 126) 5.53 (line 79) 6. 71,72 (end of file)

	-Minimax, MinimaxAI	
Mathematical Models 1.Recursive algorithms 2.Complex algorithms 3.Matrix operations/neural network	1. Minimax recurse through game tree and Randomize 960 recurses until a valid 960 board. 2. Alpha-beta pruning 3.Creating neural network model	1. 71, 53 (line 80) 2. 72 3. 81, 80
Data Structures 1.Lists and operations 2.Stacks 3.Queues 4.Graphs/trees and traversal 5.Multi-dimensional arrays 6.Files and text files 7. Dictionaries	1. Many uses throughout code, for example Board.enpassant_check which keeps list of squares to check for en passant. 2. Board.states has each game state pushed onto it. 3. Pygame events stored in queue so processed in order. 4. Minimax does recursive tree traversa 5. Tensor processing. 6. -piece images -lichess training data, csv file -model file 7. Converting between numeric board position and algebraic board position, reverse notation.	1. 53 (line 73) 2. 53 (line 72) 3. 74 (line 136) 4. 71 5. 84 6. -74 (line 16) -81 (line 9) -72 (line 102) -53

Stage One

Notable Board Class features

Board class initialisation:

```
44      #creates "slots" for class variables decreasing memory usage and increasing speed
45      #if changed, change copy method to include change
46      __slots__="board","turn","move_count","states","enpassant_check","unmoved","valid_moves_cache"
```

This creates slots for the class which only allocate memory to these class variables. This gives the class a limited size allowing each class to be stored with less memory, therefore increasing speed. If the slots are changed, the copy method must also copy the slot to keep the state of the class.

Chess960 Board initialisation:

```
    @staticmethod
    def randomize960():
        """
        randomize for backrows for chess 960
        """
        startposition=Board.startposition
        rows=[startposition[i:i+8] for i in range(0,len(startposition),8)]
        randomizedposition=[]
        randomrow=""
        for row_i,row in enumerate(rows):
            if row_i == 0:
                row="".join(random.sample(row,len(row)))
                randomrow=row
            if row_i==7:
                row=randomrow.lower()
            randomizedposition.append(row)
        startposition=("".join(randomizedposition))
        if Board.valid960(startposition):
            return startposition
        else:
            return Board.randomize960()

    @staticmethod
    def valid960(startposition):
        """
        validates if random boards obey chess 960 rules
        """
        top_row=[startposition[i:i+8] for i in range(0,len(startposition),8)][0]
        bishops_pos=[i for i,piece in enumerate(top_row) if piece=="B"]
        if len(bishops_pos)==2 and bishops_pos[0]%2==bishops_pos[1]%2:
            return False
        king_pos=top_row.index("K")
        rooks_pos=[i for i,piece in enumerate(top_row) if piece=="R"]
        if len(rooks_pos)==2 and not(rooks_pos[0]<king_pos<rooks_pos[1]):
            return False
        return True
```

These first method just randomizes the board from this “Board.startposition” which looks like:

```
    startposition=("RNBQKBNR"
                   "PPPPPPPP"
                   "........"
                   "........"
                   "........"
                   "........"
                   "pppppppp"
                   "rnbqkbnr")
```

Then checks to see if the randomized board obeys Chess 960 rules. These are notable methods as they are the only static methods. This is as they need to be used inside the board initialisation and therefore cannot belong to a specific instance of the class. Initialised here:

```
    def __init__(self, startboard=None,turn=WHITE,chess960=False):
        if startboard is None:
            startboard=Board.startposition
            if chess960:
                startboard=Board.randomize960()
```

Move method:

The move method is another special method in the class because it is the only method that can mutate the board state.

```
425     def move(self,src,dst):
426         """
427         Moves a piece from src to dst (in algebraic), and returns updated board, appends old board to states
428         """
429         pos_dst=self.algebraic_to_pos(dst)
430         pos_src=self.algebraic_to_pos(src)
431         piece = self.board[pos_src]
432         if self.board[pos_src] == ".":
433             raise InvalidMoveError(f"No piece at {src}")
434         if not(pos_dst in self.piece_moves(src)):
435             raise InvalidMoveError(f"Invalid move from {src} to {dst}")
436         if self.check_piece_color(piece)!=self.turn:
437             raise InvalidMoveError("Wrong colour piece")
438         # checked for all errors now we can change the state of board
```

Once all possible errors are checked, as in, if the source or destination of the piece are invalid or the destination is another one of your pieces, then we can mutate the board.

Another notable function is the copy function:

```
622     def copy(self):
623         """
624         return deep copy of this board without using copy.deepcopy as it's slow
625         """
626         copy=Board.__new__(Board) # makes uninitialised board
627         #shallow copy class variables
628         copy.board=self.board[:]
629         copy.turn=self.turn
630         copy.move_count=self.move_count
631         copy.states=self.states[:]
632         copy.enpassant_check=self.enpassant_check[:]
633         copy.unmoved=self.unmoved.copy()
634         copy.clear_caches()
635         return copy
```

As mentioned in the docstring, using the copy.deepcopy function is much simpler than this but it was much slower due to calling copy each time a move needs to be evaluated.

Notable GUI and UI Class features

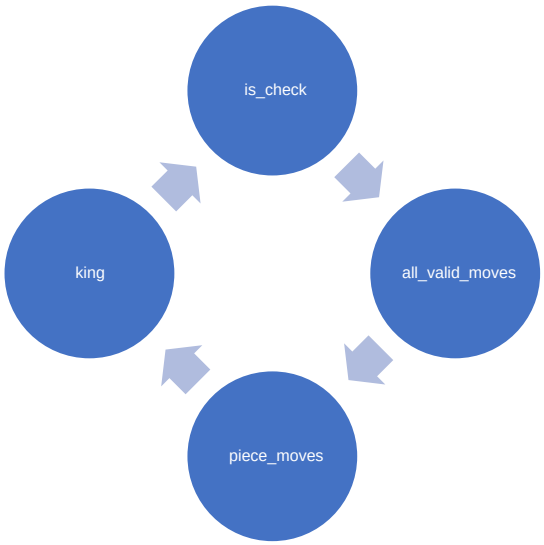
Challenges

While coding the check function I encountered multiple problems about the king position, the “is_check” function only worked when there was a king at the position specified. This means specific positions cannot be tested theoretically without inputting an actual board, this is especially important for the checkmate function. The simple fix to this was to give “is_check” an optional argument “color” this means the “is_check” function will work with either a position, colour or both but not neither, as seen below.

```
474     def is_check(self,king_pos,color=None):
475         """
476         Checks if the king is in check, returns Boolean, king_pos number postion, if no color specified (to be confirmed)
477         has to find out color from pos, if king not at pos and not specified color then error
478         """
```

```
#check if in check
if king_pos in self.all_valid_moves(check_turn,checkcheck=False):
    return True
return False
```

However as seen in the arguments of “all_valid_moves” a new optional argument “checkcheck” needed to be introduced. This was introduced as the code described works but causes “all_valid_moves” to call the “is_check” function on the opposite colour king, which would result in a recursion error which is described in the diagram below.



So, the solution to this was to introduce the “checkcheck” optional argument, with this and some Boolean expressions this error would not happen.

```
if self.move_okay(dst) and (checkcheck==False or self.is_check(dst,turn)==False):
    valid_moves.append(dst)
```

Figure 16/King "checkcheck" valid move function

However later this was changed to a nested “is_okay” function going through the valid moves with the “is_check” function.

```
def is_okay(src):
    """
    checks if square is blank and king wouldn't be in check if moved there
    """
    return self.board[src]=="." and (checkcheck==False or not self.is_check(src,turn))

for move in valid_moves:
    if checkcheck==True and self.is_check(move,turn):
        valid_moves.remove(move)
return valid_moves
```

As well as “is_check” making moves on a copy board.

```
copy=self.copy()
if check_turn==self.WHITE:
    copy.board[king_pos]="K"
else:
    copy.board[king_pos]="k"
if king_pos in copy.all_valid_moves(check_turn,checkcheck=False):
    return True
return False
```

Stage Two

Evaluation

As mentioned in design the evaluation of the board is based on Claude Shannon's evaluation and was implemented in the SimpleEvaluationMixin class containing the method "evaluate". The reason for the mixins was to promote the principle of composition over inheritance, as well as simplifying the code so that either mixin can be used in alpha-beta pruning or minimax.

Importantly, before working out an evaluation the board's state must be checked and accounted for.

```
winlossdraw=b.winlossdraw()
if winlossdraw=="white win":
    return 1000
elif winlossdraw=="black win":
    return -1000
elif winlossdraw=="draw":
    return 0
```

The weighting as ±1000 for a win is to encourage going for a win rather than a high value piece e.g. a queen when recursing through the game tree.

Minimax

The Minimax algorithm is relatively simple and translates well to my code using the pseudocode from Wikipedia.

```
function minimax(node, depth, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, minimax(child, depth − 1, FALSE))
        return value
    else (* minimizing player *)
        value := +∞
        for each child of node do
            value := min(value, minimax(child, depth − 1, TRUE))
        return value
```

Figure 17/Minimax Pseudocode/ (Wikipedia, n.d.)

This worked out to be:

```
class MinimaxBase():
    """
    Base class for minimax
    """
    def __init__(self,board,depth):
        self.board=board
        self.depth=depth

    def best_move(self):
        """
        returns best move at set depth
        """
        return self.best_move_for_level(self.board,self.depth)[1]

    def best_move_for_level(self,board,depth):
        """
        returns best move and evaluation, for current level and below until depth 0
        """
        movelist=board.valid_move_src_dst(board.turn)
        bestmovelist=[]
        for src,dst in movelist:
            copyboard=board.copy()
            copyboard.move(src,dst)
            if depth<=1:
                eval=self.evaluate(copyboard)
            else:
                eval,move=self.best_move_for_level(copyboard,depth-1)
            bestmovelist.append((eval,(src,dst)))
        # reverse if turn is white, sorts lowest first by default, lowest is best evaluation for black
        # maximises for white, minimises for black
        bestmovelist.sort(reverse=board.turn==board.WHITE)
        return bestmovelist[0]
```

Where there is no clause for maximising or minimising player as that is already built into the board. It is also implemented as two functions so the best move function can be used as a set depth best move search.

As this uses the self.evaluate function it is to be noted this is a base class allowing for either simple evaluation or an AI evaluation to be used.

```
class Minimax(MinimaxBase,SimpleEvaluationMixin):
    def __init__(self,board,depth):
        MinimaxBase.__init__(self,board,depth)
        SimpleEvaluationMixin.__init__(self)

class MinimaxAI(MinimaxBase,AIEvaluationMixin):
    def __init__(self,board,depth):
        MinimaxBase.__init__(self,board,depth)
        AIEvaluationMixin.__init__(self,"AI/chess5M.keras")
```

Alpha-Beta Pruning

Alpha-beta Pruning is slightly more complicated, and I polished it more as it is the main tree searching algorithm. It follows the same structure as minimax however I added some statistics to be displayed in the console too.

```python
def best_move(self):
    """
    returns best move at set depth
    """
    self.count=0
    start=time.time()
    eval=self.best_move_for_level(self.board,self.depth,-1000,1000)
    dt=time.time()-start
    print(f"Evaluations done: {self.count}, Best evaluation: {eval[0]}, Time taken: {round(dt,4)}")
    return eval[1]
```

The pseudocode also translated very well:

```
function alphabeta(node, depth, α, β, maximizingPlayer) is
    if depth == 0 or node is terminal then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, alphabeta(child, depth − 1, α, β, FALSE))
            if value > β then
                break (* β cutoff *)
            α := max(α, value)
        return value
    else
        value := +∞
        for each child of node do
            value := min(value, alphabeta(child, depth − 1, α, β, TRUE))
            if value < α then
                break (* α cutoff *)
            β := min(β, value)
        return value
```

Figure 18/alpha-beta pruning pseudocode/ (Wikipedia, n.d.)

```
    if board.turn==board.WHITE: #maximising player
        eval=-1000
        for src,dst in movelist:
            copyboard=board.copy()
            copyboard.move(src,dst)
            eval,move=self.best_move_for_level(copyboard,depth-1,alpha,beta)
            bestmovelist.append((eval,(src,dst)))
            if eval>beta:
                break #beta cut off
            alpha=max(alpha,eval)

    else: #minimising player
        eval=1000
        for src,dst in movelist:
            copyboard=board.copy()
            copyboard.move(src,dst)
            eval,move=self.best_move_for_level(copyboard,depth-1,alpha,beta)
            bestmovelist.append((eval,(src,dst)))
            if eval<alpha:
                break #alpha cut off
            beta=min(beta,eval)

bestmovelist.sort(reverse=board.turn==board.WHITE)
if len(bestmovelist)==0:
    self.count+=1
    return self.evaluate(board),None
return bestmovelist[0]
```

To add to the Alpha-beta pruning I experimented with multi-threading. I wanted to try using a new thread for each sub tree from the top level, however doing this meant it would not reduce the amount of searching alpha-beta pruning did. Although it worked much faster it did not reduce the amount of searching in larger depths and thus it was decided that it defeated the point of Alpha-beta pruning.

```
# multi threading experiment
if False:
# if depth==self.depth:
    with ThreadPool() as pool:
        def task(move):
            """
            uses a new thread for each sub tree of top level minimax tree
            """
            src,dst=move
            copyboard=board.copy()
            copyboard.move(src,dst)
            eval,move=self.best_move_for_level(copyboard,depth-1,alpha,beta)
            return eval,(src,dst)
        bestmovelist=list(pool.imap(task,movelist))
```

As with Minimax, Alpha-beta pruning was also separated into two other classes, with AlphabetaPruningAI being the focus of the project.

```
class AlphabetaPruning(AlphabetaPruningBase,SimpleEvaluationMixin):
    def __init__(self,board,depth):
        AlphabetaPruningBase.__init__(self,board,depth)
        SimpleEvaluationMixin.__init__(self)

class AlphabetaPruningAI(AlphabetaPruningBase,AIEvaluationMixin):
    def __init__(self,board,depth):
        AlphabetaPruningBase.__init__(self,board,depth)
        AIEvaluationMixin.__init__(self,"AI/chess5M.keras")
```

Challenges

At this point of the project, I started to realise how inefficient some of my algorithms were in the Board class. To help with speed I did some experiments such as the threading and improved many functions in the Board class such as the copy function which was mentioned in Stage One. Another experiment and proof of concept I did was to sort moves which would allow alpha-beta pruning to be more efficient while pruning.

```
def sort_moves(self,board,moves):
    """
    sort moves so takes are first to make alpha-beta pruning more efficient
    """
    notake=[]
    take=[]
    for src,dst in moves:
        if board.board[board.algebraic_to_pos(dst)]!=".":
            take.append((src,dst))
        else:
            notake.append((src,dst))
    return take+notake
```

This function sorts the moves so that Alpha-beta pruning will recurse through moves where there is a take first (more likely to have better evaluations). This is a small part of how much deeper searching chess engines such as Stockfish can search with a depth of 30 or more.

Another challenge which I faced during testing Alpha-beta pruning was due to the simple evaluation being too simple. This created a challenge in that many more of the evaluations were the same (as they are all integers) meaning the number of evaluations done could not be pruned as much as would be preferable. Despite this it was obvious Alpha-beta pruning was working, and it would work much better with a more complex evaluation.

Stage Three

Training

Data was collected from Index of /standard/ in the Lichess database, one file was selected. The data in these were filtered down to the lichess_db_eval.csv file.

The training was done in two parts: make something work, then make it work well, each done it separate files. Both files contained a way to load the data from the dataset in a csv. An example of a datapoint is seen in the second comment, with a FEN then an evaluation in centipawns.

```
# import data from csv
# example "7r/1p3k2/p1bPR3/5p2/2B2P1p/8/PP4P1/3K4 b - -",58
dataset="lichess_db_eval.csv"

def load_csv(data):
    """
    generator for reading lines out of large csv
    """
    with open(data,"r") as file:
        reader=csv.reader(file)
        for row in reader:
            board_fen,eval=row[0],row[1]
            yield (board_fen,eval)
```

This function must be a generator as there are millions of FEN notation in the csv and yielding the data allows it to be processed one at a time rather than all being loaded into memory. Then the data must be processed into the right training format for TensorFlow using the “fen_to_tensor” function. Where a tensor is a multi-dimensional matrix often used in machine learning. This function converts FEN notation into a tensor which is 8x8x13, consisting of 8x8x12 consisting of an 8x8 board for 12 pieces then a 13th row for the current side playing.

```
def fen_to_tensor(fen):
    """
    converts fen board notation to tensor of board and turn colour
    """
    board,turn,*_=fen.split() # ignore all except board and turn (simplicity)
    board_tensor = np.zeros((8,8,12))
    rows=board.split("/")
    for row_i,row in enumerate(rows):
        col_i=0
        for char in row:
            if char.isdigit():
                col_i+=int(char)
            else:
                piece=pieces[char]
                board_tensor[row_i,col_i,piece]=1 # add to tensor
                col_i+=1
    if turn=="w":
        turn_tensor=np.ones((8,8,1))
    else:
        turn_tensor=np.zeros((8,8,1))
    return np.concatenate([board_tensor,turn_tensor],axis=2) # axis 2 makes 8x8x12 into 8x8x13

i=0
for board_fen,eval in load_csv(dataset):
    x_train[i]=fen_to_tensor.fen_to_tensor(board_fen)
    y_train[i]=float(eval)/100 #convert from centipawns to pawns
    i+=1
    if i==size:
        break
```

This allows the data to be trained with a model. The simple model is as follows: an input layer for the 8x8x13 tensor; 2 convolution layers to apply filters which detect patterns; a flatten layer which decreases the dimension of the vector; then two dense layers using the ReLU activation function, fully connecting the layers and allowing it to learn patterns or relationships of the board and pieces; and finally the output layer which outputs a float of the evaluation.

```
def create_model():
    """
    return tensor flow model for chess evaluation
    """
    model=tf.keras.models.Sequential()
    #input
    model.add(tf.keras.layers.Input(shape=(8,8,13)))
    #convolution 1
    model.add(tf.keras.layers.Conv2D(32,(3,3),padding="same"))
    model.add(tf.keras.layers.BatchNormalization())
    model.add(tf.keras.layers.Activation("relu"))
    #convolution 2
    model.add(tf.keras.layers.Conv2D(64,(3,3),padding="same"))
    model.add(tf.keras.layers.BatchNormalization())
    model.add(tf.keras.layers.Activation("relu"))
    #flatten
    model.add(tf.keras.layers.Flatten())
    #dense 1
    model.add(tf.keras.layers.Dense(128,activation="relu"))
    model.add(tf.keras.layers.Dropout(0.5))
    #dense 2
    model.add(tf.keras.layers.Dense(64,activation="relu"))
    model.add(tf.keras.layers.Dropout(0.5))
    #output
    model.add(tf.keras.layers.Dense(1,activation="linear"))
    #compile
    model.compile(optimizer="adam",loss="mean_squared_error")
    return model
```

The more complex architecture is seen in the Neural Network Architecture section which was generated by “tf.keras.utils.plot_model” function. The additional layers are residual connections, global average pooling, and using the Huber loss function instead of mean squared error to account for outliers (for example a win or loss returns an evaluation of 1000).

Then it is trained with the training function, which takes how many datapoints, how many epochs and the batch size which affect how it trains and how long for.

```
def train(size=1000000,epochs=50,batch_size=32):
    """
    Create and train the model on size items from the dataset
    """
```

To note, in the dataset the amounts of black and white evaluations are not equal. So, to train the evaluation function to give even evaluations for black and white, the dataset needed to be filtered while training to an even amount of black and white evaluations. This was done as follows:

```
# balance the number of positive and negative evaluations
# to within 1%
if eval > 0:
    if positive > 1.01*negative:
        continue
    positive += 1
elif eval < 0:
    if negative > 1.01*positive:
        continue
    negative += 1
```

Then it was fitted with the data:

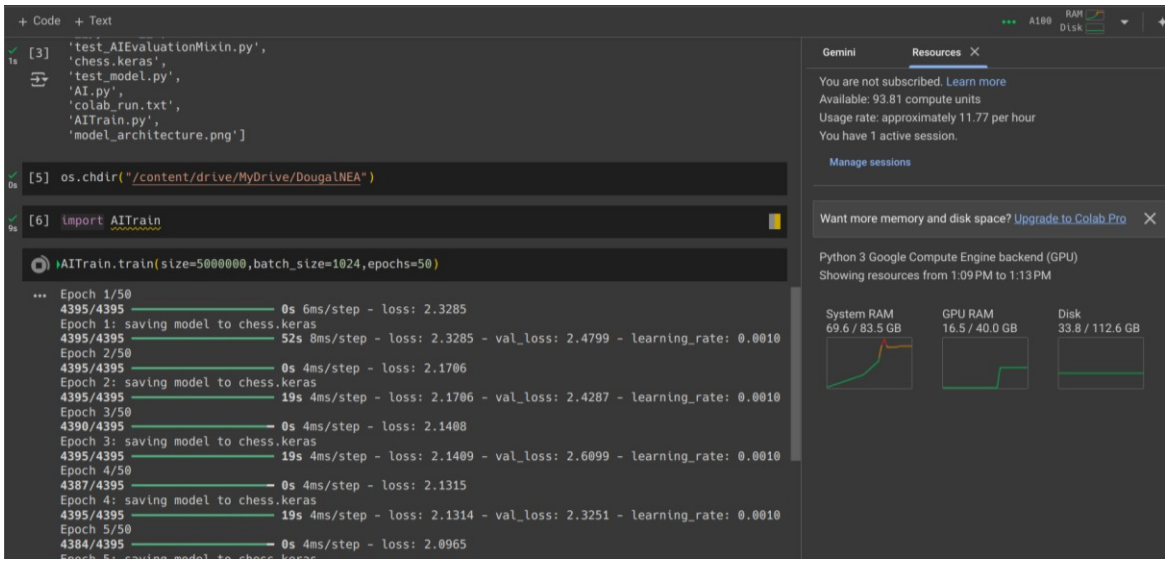
```
model.fit(x_train,
          y_train,
          epochs=epochs,
          batch_size=batch_size,
          validation_split=0.1,
          callbacks=[reduce_lr, save_model],
)
model.save("chess.keras")
```

I trained the model multiple times, depending on the amount of datapoints. Ending up with a 7.5M datapoint evaluation function on the simple neural network and a 5M on the complex network. The 7.5M is much worse even though it has more datapoints which is interesting and the 5M could be trained with much more data (up to 70M) but it was decided that this would take too long and would have very diminishing returns. The Epoch of training for 7.5M is as follows:

```
Epoch 50/50
105469/105469 ━━━━━━━━━━━━━━━━━━━━ 808s 8ms/step - loss: 127.4221 - val_loss: 135.1333
```

With a loss of 127 and a validation loss of 135 it is not overfitted, but it does not seem very accurate with such a high loss. However, this does not reflect the actual evaluations it returns so the loss was not a very helpful measurement.

For the more complex network, which was trained with Google Colab, as the error method was Huber Loss it seems to return a more accurate loss and validation loss.



In theory the validation loss should reflect the difference of evaluation between my evaluation function and Stockfish's. Also seen in the screenshot, each epoch takes much less time as it is being run on an A100 graphics card with 83.5GB of system RAM. This took 20 minutes rather than 10+ hours for the 7.5M datapoints run on my computer.

Neural Network Architecture

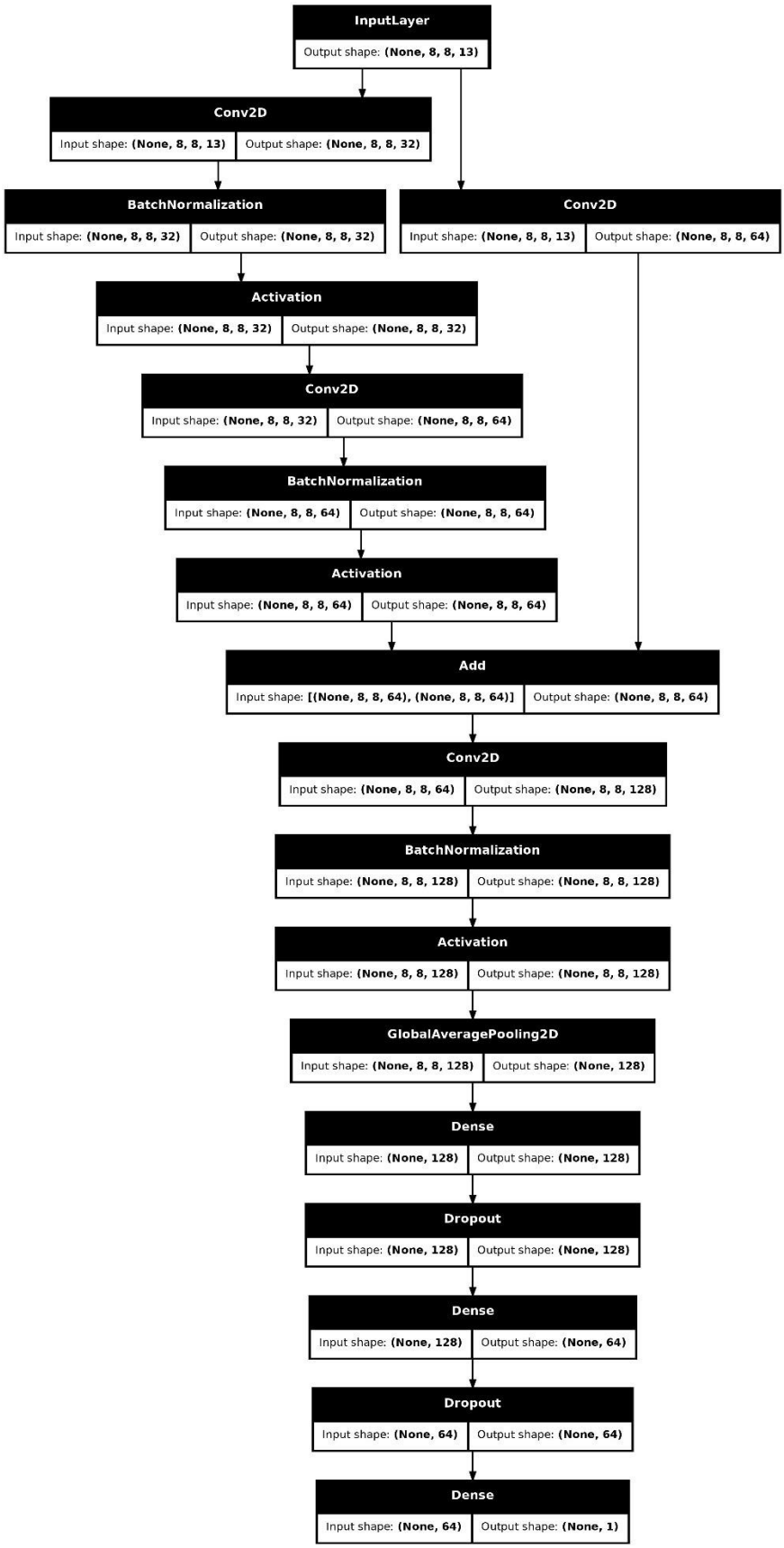


Figure 19/Final Architecture

Challenges

The main challenge during this process was optimising the neural network. Essentially this was just a trial-and-error process constantly iterating on what I thought would make it play better chess. I think the challenge with all activities involving neural networks is that they are quite magical, and it is hard to tell if for example, adding an extra dense layer is helping the network learn the patterns you want it to learn.

One thing to be noted about the final trained network is that it is only trained on 5 million boards rather than the full amount. If I did train it on the whole dataset, I would have to buy more Colab compute time and I decided that it plays well enough already. Although there could possibly be a difference in a couple hundred elo points if it had access to more board states.

To add to this training the model was a very challenging experience. On my main computer TensorFlow could not to find my GPU and I ended up training the first few iterations on my CPU. This led to using Google Colab, which was amazing, but it did cost a few pounds to buy compute time.

Testing

All unit tests found in appendix.

Test Video

Link:

https://youtu.be/rExKaI1SeAw

Results

Tests for Objective 1

Test 1:

```
PS C:\Users\23024\OneDrive - RGS\Python\NEA> & "C:/Program Files/Python/python.exe" "c:/Users/23024/OneDrive - RGS/Python/NEA/tests/test_board.py"

..............................
----------------------------------------------------------------------
Ran 30 tests in 0.036s

OK
PS C:\Users\23024\OneDrive - RGS\Python\NEA>
```

Test 2:

```
PS C:\Users\23024\OneDrive - RGS\Python\NEA> & "C:/Program Files/Python/python.exe" "c:/Users/23024/OneDrive - RGS/Python/NEA/tests/test_UI.py"
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
PS C:\Users\23024\OneDrive - RGS\Python\NEA>
```

Test 3:

Testing demonstrated in test video and below.

UI board is valid:

```
8   R N B Q K B N R
7   P P P P P P P P
6   . . . . . . . .
5   . . . . . . . .
4   . . . . . . . .
3   . . . . . . . .
2   p p p p p p p p
1   r n b q k b n r

    a b c d e f g h
```

Invalid moves return an error with the specified error:

```
Original position of piece:a2
Destination of piece:a5
InvalidMoveError Invalid move from a2 to a5
```

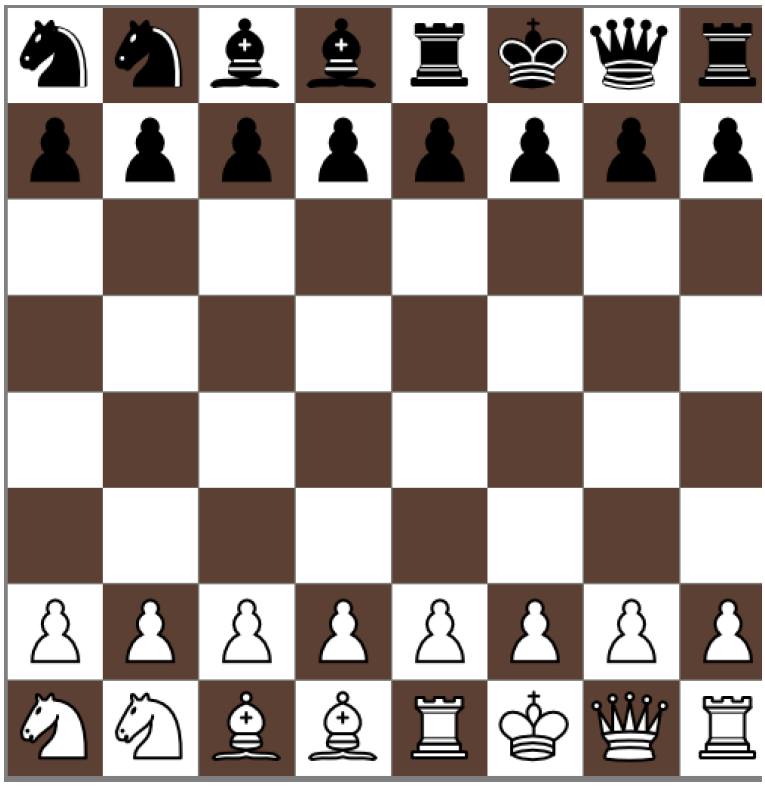
After an Error the player is queried for a move again:

```
InvalidAlgebraicNotationError Not valid algebraic notation: a0
Original position of piece:b4
Destination of piece:
```

Test 4:

GUI boards are valid:

And 960 board are valid:



This is valid as it satisfies the following:

1. The bishops must be placed on opposite-color squares.
2. The king must be placed on a square between the rooks.

Figure 20/chess 960 rules/ (Wikipedia, n.d.)

En Passant is also valid:

After the black pawn moves up two spaces, the white pawn now has the option to diagonally take.

Other testing demonstrated in test video.

Tests for Objective 2

Test 1:

```
PS C:\Users\23024\OneDrive - RGS\Python\NEA> & "C:/Program Files/Python/python.exe" "c:/Users/23024/OneDrive - RGS/Python/NEA/tests/test_SimpleEva
luationMixin.py"
..
----------------------------------------------------------------------
Ran 2 tests in 0.004s

OK
PS C:\Users\23024\OneDrive - RGS\Python\NEA>
```

Test 2:

```
PS C:\Users\23024\OneDrive - RGS\Python\NEA> & "C:/Program Files/Python/python.exe" "c:/Users/23024/OneDrive - RGS/Python/NEA/tests/test_minimax.p
y"
...
----------------------------------------------------------------------
Ran 3 tests in 3.914s

OK
PS C:\Users\23024\OneDrive - RGS\Python\NEA>
```

Test 3:

```
PS C:\Users\23024\OneDrive - RGS\Python\NEA> & "C:/Program Files/Python/python.exe" "c:/Users/23024/OneDrive - RGS/Python/NEA/tests/test_alphabeta
pruning.py"
Evaluations done: 5, Best evaluation: -7, Time taken: 0.0031
Evaluations done: 58, Best evaluation: -1, Time taken: 0.0498
Evaluations done: 817, Best evaluation: -200, Time taken: 0.676
.Evaluations done: 7, Best evaluation: 3, Time taken: 0.004
Evaluations done: 55, Best evaluation: 1, Time taken: 0.0548
Evaluations done: 1097, Best evaluation: 200, Time taken: 0.9045
..
----------------------------------------------------------------------
Ran 3 tests in 1.695s

OK
PS C:\Users\23024\OneDrive - RGS\Python\NEA>
```

Time difference between 3.914 and 1.695 on the same tests demonstrates the efficiency of alpha-beta pruning compared to minimax.

Tests for Objective 3

Test 1:

AI evaluation differs from Stockfish as expected but is in a very acceptable range in most cases.

```
PS C:\Users\23024\OneDrive - RGS\Python\NEA> & "C:/Program Files/Python/python.exe" "c:/Users/23024/OneDrive - RGS/Python/NEA/tests/test_AIEvaluationMixi
n.py"
-0.08483865112066269
14.211164474487305
14.750652313232422
..
----------------------------------------------------------------------
Ran 2 tests in 0.482s

OK
```

Test 2:

```
PS C:\Users\23024\OneDrive - RGS\Python\NEA> & "C:/Program Files/Python/python.exe" "c:/Users/23024/OneDrive - RGS/Python/NEA/tests/test_minimax.p
y"
...
----------------------------------------------------------------------
Ran 3 tests in 19.869s

OK
PS C:\Users\23024\OneDrive - RGS\Python\NEA>
```

```
PS C:\Users\23024\OneDrive - RGS\Python\NEA> & "C:/Program Files/Python/python.exe" "c:/Users/23024/OneDrive - RGS/Python/NEA/tests/test_alphabeta
pruning.py"
Evaluations done: 5, Best evaluation: -3.4507501125335693, Time taken: 0.2943
Evaluations done: 47, Best evaluation: 0.030200010165572166, Time taken: 0.4415
Evaluations done: 707, Best evaluation: -1000, Time taken: 2.7614
.Evaluations done: 7, Best evaluation: 11.199872970581055, Time taken: 0.1392
Evaluations done: 41, Best evaluation: 0.030200010165572166, Time taken: 0.2681
Evaluations done: 1097, Best evaluation: 200, Time taken: 1.3367
..
----------------------------------------------------------------------
Ran 3 tests in 6.408s

OK
PS C:\Users\23024\OneDrive - RGS\Python\NEA>
```

Final Tests

Test 1:

Each option shown making moves in the test video, with a focus on the AlphabetapruningAI player.

Test 2:

Highlighted in bold are the overall wins.

Depth	Stockfish Elo	Draws	Wins	Losses
1	500	0	1	**9**
	1000	0	0	**10**
	1500	0	0	**10**
	2000	0	0	**10**
	2500	0	0	**10**
2	500	2	**5**	3
	1000	0	**7**	3
	1500	2	0	**8**
	2000	2	1	**7**
	2500	0	1	**9**
3	500	1	**8**	1
	1000	1	**5**	4
	1500	0	**5**	5
	2000	2	2	**6**
	2500	2	0	**8**

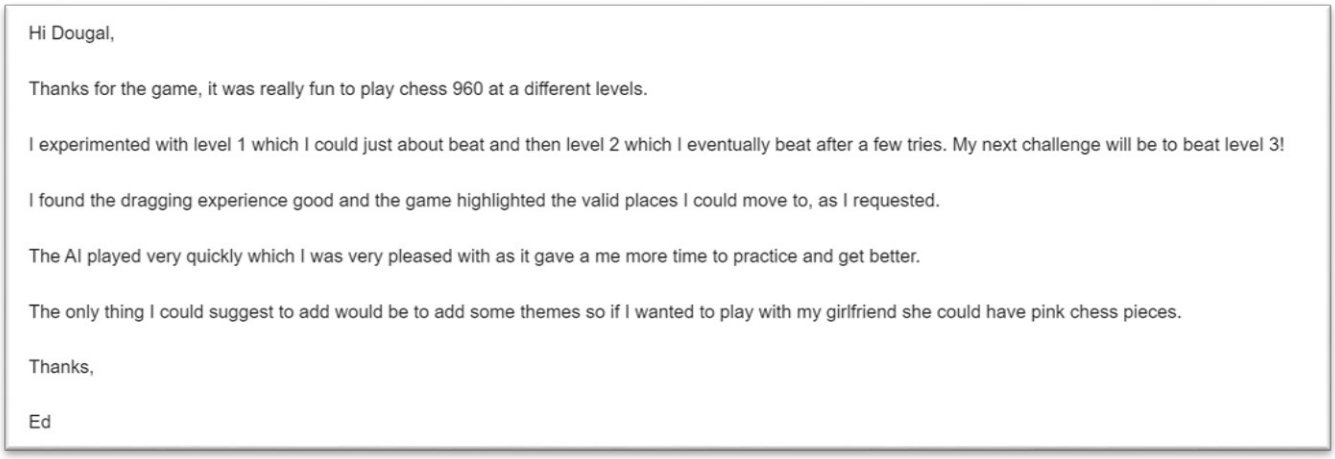
Elo estimate table:

Rating diff	Prob win
+800	0.99%
+750	1.32%
+700	1.75%
+650	2.32%
+600	3.07%
+550	4.05%
+500	5.32%
+450	6.98%
+400	9.09%
+350	11.77%
+300	15.10%
+250	19.17%
+200	24.03%
+150	29.66%
+100	35.99%
+50	42.85%
0	**50.00%**

Figure 21/Elo table difference/ (318chess, n.d.)

Evaluation

Client Feedback



Hi Dougal,

Thanks for the game, it was really fun to play chess 960 at a different levels.

I experimented with level 1 which I could just about beat and then level 2 which I eventually beat after a few tries. My next challenge will be to beat level 3!

I found the dragging experience good and the game highlighted the valid places I could move to, as I requested.

The AI played very quickly which I was very pleased with as it gave a me more time to practice and get better.

The only thing I could suggest to add would be to add some themes so if I wanted to play with my girlfriend she could have pink chess pieces.

Thanks,

Ed

Adding themes for the pieces and board is another future improvement that could be made. Apart from this, the project seems to satisfy the client and their original requests.

Review of Objectives and Client Requirements

SMART Objectives:

1. I successfully created a Playable chess Board with originally a text-based UI then an interactable GUI, this was created within around 8 weeks and other tweaks throughout the other objectives as needed.

1.1 Text-based chess board implemented and shown in testing.

1.2 Move function mutates the board and allows moves in algebraic notation and used in dragging and dropping pieces.

1.3 Game checks and rules added allowing a complete game of chess.

1.4 Graphical representation added which displays standard chess or chess 960.

1.5 Pieces in the GUI can be dragged and dropped only to valid spaces which are highlighted in yellow by the board.

1.6 The board flips for a black player, if it is a black player versus the engine, it stays permanently flipped.

1.7 Win, loss and draw are displayed on the GUI in text as well as which side is in check.

2. I successfully created playable Minimax and Alpha-beta pruning algorithms that work with a simple evaluation function and is twice as fast with Alpha-beta pruning. This took less than time than anticipated with it somewhat working in just a week.

2.1 Simple evaluation was implemented as a simplified version of Claude Shannon's evaluation.

2.2 Minimax was used to create a best move function which takes a depth and a board to search for.

2.3 GUI was adjusted to allow different players.

2.4 Alpha-beta pruning has also been implemented to replace minimax as a more efficient tree searching algorithm.

2.5 The alpha-beta pruning algorithm is about 2 times faster than minimax with the sorting moves function, and in higher depth searches it saves even more time.

3. Using Alpha-beta pruning and a newly trained evaluation function I was able to beat Stockfish at 2000 elo. This took around the expected time to be at a level which I deemed completed.

3.1 Training data was converted with the fen to tensor function, which converts a fen into a tensor containing the board and the current turn.

3.2 A simple neural network trained on about 10000 datapoints was trained and returned crude evaluations of the board.

3.3 A new neural network was created based on the previous and contains many more layers which returns a very similar evaluation to Stockfish in most cases.

3.4 Boards are converted with the board to fen method and then converted to a tensor, allowing the neural network to be played against.

3.5 Different types of players were created using mixins and the different types of evaluation function.

3.6 A new file “test against stockfish” was created to allow the final testing of the engine compared to Stockfish at various elos.

Client requirements:

1. Adjustable difficulties with the depth allow a challenging experience but not impossible to beat as specified.
2. The GUI has a smooth dragging experience and highlights squares as specified.
3. Even at the highest depth option the AI plays in below 20 seconds as specified.
4. Game metrics are printed in the console and so can be checked with an evaluation for analysis as specified.
5. Errors are handled intuitively, and the piece is reset back to its position as specified.
6. Using the selection menu, you can choose different players including the AI or another human as specified.

Limitations and Future Improvements

The limitation was that it was slow. At levels above three it took more than 30 seconds a turn which was too slow.

There are many solutions to this:

- Better ordering for moves in Alpha-beta pruning to increase effectiveness.
- Batching the AI evaluation as it uses the GPU much more effectively.
- Multithreading the subtrees (difficult in python) in game search.
- Reduce duplicated calculations when checking board states.
- More advanced board representation.
- Write in a faster language that makes multithreading possible.

A previously mentioned limitation is the training of the neural network. This could not be done on my own computer and therefore cost money. Further experimentation with the neural network would certainly bring better performance.

Other well-known chess techniques to improve the play which could be implemented are:

- Transposition table to store already evaluated positions to reduce computation.
- Iterative deepening and Quiescence search to search deeper than the depth specified.

Final Thoughts

I think this project has been interesting, challenging and has greatly improved my knowledge of programming, neural networks and chess. I am very happy with the performance even at a low depth of 3 where I estimate the Elo to be around 1800 in standard rating according to the table seen in testing. I was also very pleasantly surprised by how much better it played after adding machine learning. To play at that level with the simple evaluation it would need a much larger depth of possibly 10 or more.

Jumping into the deep end of python programming and machine learning through this project has been a rewarding experience which will be valuable for my future.

Appendix

Chess/board

```
"""
This is the implementation of the Chess Board
"""

import random

class ChessError(ValueError):
    """
    This is a superclass for all chess errors, differentiates them from
ValueErrors
    """

class EmptySquareError(ChessError):
    pass

class InvalidAlgebraicNotationError(ChessError):
    pass

class OutsideBoardError(ChessError):
    pass

class InvalidMoveError(ChessError):
    pass

class InvalidColorError(ChessError):
    pass

class Board:
    """
    This class represents a chess board and knows how to calculate valid moves
    """
    notation=["a8","b8","c8","d8","e8","f8","g8","h8",
              "a7","b7","c7","d7","e7","f7","g7","h7",
              "a6","b6","c6","d6","e6","f6","g6","h6",
              "a5","b5","c5","d5","e5","f5","g5","h5",
              "a4","b4","c4","d4","e4","f4","g4","h4",
              "a3","b3","c3","d3","e3","f3","g3","h3",
              "a2","b2","c2","d2","e2","f2","g2","h2",
              "a1","b1","c1","d1","e1","f1","g1","h1",]
    reverse_notation = {}
    for i, square in enumerate(notation):
        reverse_notation[square]=i
```

```
    #creates "slots" for class variables decreasing memory usage and
increasing speed
    #if changed, change copy method to include change
    __slots__="board","turn","move_count","states","enpassant_check","unmoved"
,"valid_moves_cache"

    startposition=("RNBQKBNR"
                   "PPPPPPPP"
                   "........"
                   "........"
                   "........"
                   "........"
                   "pppppppp"
                   "rnbqkbnr")

    valid_pieces={"p","P","n","N","b","B","r","R","q","Q","k","K","."}
    BLACK=0
    WHITE=1

    def __init__(self, startboard=None,turn=WHITE,chess960=False):
        if startboard is None:
            startboard=Board.startposition
            if chess960:
                startboard=Board.randomize960()
        self.board=[" "]*64
        assert len(startboard) == 64
        for i,piece in enumerate(startboard):
            assert piece in self.valid_pieces
            self.board[i]=piece
        self.turn=turn
        self.move_count=0
        self.states=[]
        self.enpassant_check=[]
        self.unmoved={0,7,4,56,63,60} # unmoved castles and kings positions
        if chess960:
            self.unmoved={self.find_piece("R")[0],self.find_piece("R")[1],self
.find_piece("K")[0],self.find_piece("r")[0],self.find_piece("r")[1],self.find_
piece("k")[0]}
        self.clear_caches()

    @staticmethod
    def randomize960():
        """
        randomize for backrows for chess 960
        """
        startposition=Board.startposition
```

```
        rows=[startposition[i:i+8] for i in range(0,len(startposition),8)]
        randomizedposition=[]
        randomrow=""
        for row_i,row in enumerate(rows):
            if row_i == 0:
                row="".join(random.sample(row,len(row)))
                randomrow=row
            if row_i==7:
                row=randomrow.lower()
            randomizedposition.append(row)
        startposition=("".join(randomizedposition))
        if Board.valid960(startposition):
            return startposition
        else:
            return Board.randomize960()

    @staticmethod
    def valid960(startposition):
        """
        validates if random boards obey chess 960 rules
        """
        top_row=[startposition[i:i+8] for i in
range(0,len(startposition),8)][0]
        bishops_pos=[i for i,piece in enumerate(top_row) if piece=="B"]
        if len(bishops_pos)==2 and bishops_pos[0]%2==bishops_pos[1]%2:
            return False
        king_pos=top_row.index("K")
        rooks_pos=[i for i,piece in enumerate(top_row) if piece=="R"]
        if len(rooks_pos)==2 and not(rooks_pos[0]<king_pos<rooks_pos[1]):
            return False
        return True

    def clear_caches(self):
        """
        clear caches after board state changes
        """
        self.valid_moves_cache={}

    def display(self):
        """
        Display the board in simple textual format with row and column numbers
        """
        for i,piece in enumerate(self.board):
            if i%8==0:
                print(f"{8-(i//8)}   ",end="")
            print(piece+" ",end="")
            if i%8==7:
```

```
                print()
        print()
        print("    a b c d e f g h")
        print(self.winlossdraw())

    def algebraic_to_pos(self,algebraic):
        """
        Convert algebraic notation to board position
        """
        try:
            pos=self.reverse_notation[algebraic]
        except KeyError:
            raise InvalidAlgebraicNotationError(f"Not valid algebraic
notation: {algebraic}")
        return pos

    def is_valid(self,src,dst):
        """
        Check if a move is valid from source (src) to destination (dst) in
algebraic format
        """
        # redo this function so move calls it and it returns true or false?
        src_pos=self.algebraic_to_pos(src)
        dst_pos=self.algebraic_to_pos(dst)
        piece=self.board[src_pos]
        if piece == ".":
            raise EmptySquareError("cannot move from an empty square")
        if piece not in self.valid_pieces:
            raise InvalidAlgebraicNotationError("Not valid piece notation")

    def check_piece_color(self,piece):
        """
        Check the colour of piece, returns WHITE, BLACK, None if neither.
        """
        if piece in "pnbrqk":
            return self.WHITE
        if piece in "PNBRQK":
            return self.BLACK
        if piece == ".":
            return None

    def move_pos(self,pos,dx,dy):
        """
        This takes the position and moves it by dx and dy, returns -1 if
outside board
        """
        x=pos%8
```

```
        y=pos//8
        x+=dx
        y+=dy
        dst_pos=x+y*8
        if x>7 or x<0 or y>7 or y<0:
            return None
        return dst_pos

    def move_okay(self,dst,turn=None):
        """
        Checks if a destination of a move is valid.
        """
        if dst is None:
            return False
        piece=self.board[dst]
        colour=self.check_piece_color(piece)
        if turn is None:
            if colour == self.turn:
                return False
        else:
            if colour==turn:
                return False
        return True

    def take_okay(self,dst,turn):
        """
        Checks if a piece is eligible to be taken
        """
        if dst is None:
            return False
        piece=self.board[dst]
        if piece == ".":
            return False
        return self.move_okay(dst,turn=turn) and
self.check_piece_color(piece)≠turn

    def pawn(self,src,turn):
        """"
        Moves for a pawn, src and dst as pos
        """
        if turn == self.WHITE:
            dy=-1
        else:
            dy=1
        valid_moves=[]
        #move one step
        dst=self.move_pos(src,0,dy)
```

```
        #pawns can only move into a free space
        if self.move_okay(dst,turn=turn) and self.board[dst]==".":
            valid_moves.append(dst)
        #two steps, can't move over pieces
        dst=self.move_pos(src,0,2*dy)
        if self.move_okay(dst,turn=turn) and self.board[dst]="." and
self.board[src+(dy*8)]==".":
            #pawns can only move twice if on home row
            if turn==self.BLACK and src//8==1:
                valid_moves.append(dst)
            elif turn==self.WHITE and src//8==6:
                valid_moves.append(dst)
        #diagonal take up and right
        dst=self.move_pos(src,1,dy)
        if self.take_okay(dst,turn):
            valid_moves.append(dst)
        #diagonal take up and left
        dst=self.move_pos(src,-1,dy)
        if self.take_okay(dst,turn):
            valid_moves.append(dst)
        #enpassant
        if src in self.enpassant_check:
            valid_moves.append(self.enpassant_check[2])
        return valid_moves

    def knight(self,src,turn):
        """
        Moves for a knight
        """
        valid_moves=[]
        #different possible vector directions it can move, see diagram
        directions=[(1,2),(-1,2),(2,1),(2,-1),(1,-2),(-1,-2),(-2,1),(-2,-1)]
        for x,y in directions:
            dst=self.move_pos(src,x,y)
            if self.move_okay(dst,turn):
                valid_moves.append(dst)
        return valid_moves

    def linear_move(self,src,dx,dy,turn):
        """"
        Moves where the piece can move as far as it wants: bishop, queen, rook
        """
        dst=src
        valid_moves=[]
        while True:
            dst=self.move_pos(dst,dx,dy)
            if self.move_okay(dst,turn=turn):
```

```
                valid_moves.append(dst)
            if dst==None:
                break
            if self.board[dst]!=".":
                break
        return valid_moves

    def bishop(self,src,turn):
        """
        Moves for bishop
        """
        #up and down is different on each colour
        if turn == self.WHITE:
            dy=-1
        else:
            dy=1
        valid_moves=[]
        #up right
        valid_moves.extend(self.linear_move(src,1,dy,turn))
        #up left
        valid_moves.extend(self.linear_move(src,-1,dy,turn))
        #down right
        valid_moves.extend(self.linear_move(src,1,-dy,turn))
        #down left
        valid_moves.extend(self.linear_move(src,-1,-dy,turn))
        return valid_moves

    def rook(self,src,turn):
        """
        Moves for a rook
        """
        if turn == self.WHITE:
            dy=-1
        else:
            dy=1
        valid_moves=[]
        #up
        valid_moves.extend(self.linear_move(src,1,0,turn))
        #right
        valid_moves.extend(self.linear_move(src,0,dy,turn))
        #down
        valid_moves.extend(self.linear_move(src,-1,0,turn))
        #left
        valid_moves.extend(self.linear_move(src,0,-dy,turn))
        return valid_moves

    def queen(self,src,turn):
```

```
        """
        Moves for a queen, equivalent to bishop and rook
        """
        valid_moves=[]
        #diagonals
        valid_moves.extend(self.bishop(src,turn))
        #straights
        valid_moves.extend(self.rook(src,turn))
        return valid_moves

    def find_rooks(self,king_pos):
        """
        finds rooks [left,right] compared to king
        """
        rooks=[None,None]
        if self.turn==self.WHITE:
            for i in range(56,64):
                if self.board[i]=="r":
                    rooks[0 if i<king_pos else 1]=i
        else:
            for i in range(8):
                if self.board[i]=="R":
                    rooks[0 if i<king_pos else 1]=i
        return rooks

    def king(self,src,turn,checkcheck=True):
        """
        Moves for a king
        """
        valid_moves=[]
        #8 directions, only move 1 space:
        directions=[(1,0),(0,1),(1,1),(-1,0),(-1,1),(0,-1),(1,-1),(-1,-1)]
        for x,y in directions:
            dst=self.move_pos(src,x,y)
            if self.move_okay(dst,turn=turn) and (checkcheck==False or
self.is_check(dst,turn)==False):
                valid_moves.append(dst)
        def is_okay(src):
            """
            checks if square is blank and king wouldn't be in check if moved
there
            """
            return self.board[src]=="." and (checkcheck==False or not
self.is_check(src,turn))
        #960 castling: a-side castle king to c-file rook to d-file, h-side
castle king to g-file rook to f-file
```

```
        if src in self.unmoved and (checkcheck==False or
self.is_check(src)==False):
            left_rook,right_rook=self.find_rooks(src)
            #a-side
            if left_rook is not None and left_rook in self.unmoved and
all(is_okay(i) for i in range(left_rook+1,src)):
                valid_moves.append(left_rook)
            #h-side
            if right_rook is not None and right_rook in self.unmoved and
all(is_okay(i) for i in range(src+1,right_rook)):
                valid_moves.append(right_rook)
        for move in valid_moves:
            if checkcheck==True and self.is_check(move,turn):
                valid_moves.remove(move)
        return valid_moves

    def piece_moves(self,src,checkcheck=True,turn=None):
        """
        Takes an algebraic src, finds the colour and the piece and runs the
function for it.
        """
        src=self.algebraic_to_pos(src)
        piece = self.board[src]
        valid_moves=[]
        if turn is None:
            turn=self.check_piece_color(piece)
        if piece == "p" or piece == "P":
            valid_moves=self.pawn(src,turn)
        elif piece == "b" or piece == "B":
            valid_moves=self.bishop(src,turn)
        elif piece == "n" or piece == "N":
            valid_moves=self.knight(src,turn)
        elif piece == "r" or piece == "R":
            valid_moves=self.rook(src,turn)
        elif piece == "q" or piece == "Q":
            valid_moves=self.queen(src,turn)
        elif piece == "k" or piece == "K":
            valid_moves=self.king(src,turn,checkcheck=checkcheck)
        else:
            raise EmptySquareError("src not a piece")
        if checkcheck:
            valid_moves=[move for move in valid_moves if
self.is_move_out_of_check(src,move)]
        return valid_moves


    def all_valid_moves(self,turn,checkcheck=True):
```

```
        """
        Iterate through all the number srcs and find run piece_moves for each
        """
        #get valid moves from cache if possible
        cache_key=(turn,checkcheck,tuple(self.board),tuple(self.enpassant_chec
k),frozenset(self.unmoved))
        cached_valid_moves=self.valid_moves_cache.get(cache_key)
        if cached_valid_moves is not None:
            return cached_valid_moves
        valid_moves=[]
        for i in self.notation:
            if
self.check_piece_color(self.board[self.algebraic_to_pos(i)])≠turn:
                continue
            try:
                new_moves=self.piece_moves(i,checkcheck=checkcheck,turn=turn)
                valid_moves.extend(new_moves)
            except EmptySquareError as e:
                print(e)
        self.valid_moves_cache[cache_key]=valid_moves # cache valid moves
        return valid_moves

    def total_moves(self):
        """
        Count how many total moves there are for the board for current side
        """
        return len(self.all_valid_moves(self.turn))

    def move(self,src,dst):
        """
        Moves a piece from src to dst (in algebraic), and returns updated
board, appends old board to states
        """
        pos_dst=self.algebraic_to_pos(dst)
        pos_src=self.algebraic_to_pos(src)
        piece = self.board[pos_src]
        if self.board[pos_src] == ".":
            raise InvalidMoveError(f"No piece at {src}")
        if not(pos_dst in self.piece_moves(src)):
            raise InvalidMoveError(f"Invalid move from {src} to {dst}")
        if self.check_piece_color(piece)≠self.turn:
            raise InvalidMoveError("Wrong colour piece")
        # checked for all errors now we can change the state of board
        castle=False
        if piece in ["k","K"]:
            if pos_src in self.unmoved and pos_dst in self.unmoved:
                castle=True
```

```
                self.board[pos_src],self.board[pos_dst]=".","."
                if piece=="k":
                    #a-side
                    if pos_dst<pos_src:
                        self.board[58],self.board[59]="k","r"
                    #h-side
                    else:
                        self.board[62],self.board[61]="k","r"
                else:
                    if pos_dst<pos_src:
                        self.board[2],self.board[3]="K","R"
                    #h-side
                    else:
                        self.board[6],self.board[5]="K","R"
        self.unmoved.discard(pos_src)
        self.move_count+=1
        self.states.append(self.board[:]) # copy the board state as a new
object
        if piece=="p" or piece=="P":
            self.enpassant(pos_dst)
        if len(self.enpassant_check)>0 and self.enpassant_check[2]==pos_dst:
            if piece=="p":
                self.board[pos_dst+8]="."
            elif piece=="P":
                self.board[pos_dst-8]="."
        if not castle:
            self.board[pos_src]="."
            self.board[pos_dst]=piece
        self.promotion(piece,pos_dst)
        #print(self.winlossdraw())
        self.turn=1-self.turn

    def is_check(self,king_pos,color=None):
        """
        Checks if the king is in check, returns Boolean, king_pos number
postion, if no color specified (to be confirmed)
        has to find out color from pos, if king not at pos and not specified
color then error
        """
        #finds colour thats being checked
        if color is None:
            if self.check_piece_color(self.board[king_pos]) == self.WHITE:
                check_turn=self.BLACK
            elif self.check_piece_color(self.board[king_pos]) is None:
                raise EmptySquareError(f"no king on specified king_pos:
{king_pos} and no color specified")
            else:
```

```
                check_turn=self.WHITE
            elif color is not None:
                if color==self.WHITE:
                    check_turn=self.BLACK
                elif color==self.BLACK:
                    check_turn=self.WHITE
                else:
                    raise InvalidColorError(f"Not valid color specified: {color}")
            #check if in check, uses copy of board to see if moved will it be in
check
            copy=self.copy()
            if check_turn==self.WHITE:
                copy.board[king_pos]="K"
            else:
                copy.board[king_pos]="k"
            if king_pos in copy.all_valid_moves(check_turn,checkcheck=False):
                return True
            return False

        def find_piece_index(self, pos):
            """
            finds index of piece on board given pos, pos in algebraic, returns
None if not found.
            """
            return self.reverse_notation.get(pos)

        def find_piece(self,piece):
            """
            finds a piece on the board, if multiple returns list, piece is in
letter form
            """
            return [i for i in range(64) if self.board[i]==piece]

        def is_stalemate(self,king_pos):
            """
            Check if stalemate for current, returns boolean, current color is the
one to be checked
            """
            if self.total_moves()==0 and self.is_check(king_pos)==False:
                return True
            return False

        def is_checkmate(self,king_pos):
            """
            Check if checkmate for current pos, returns boolean, current color is
the one to be checked,
            """
```

```
        #king has no moves AND other pieces have no VALID moves →
total_moves=0
        return self.total_moves()==0 and self.is_check(king_pos)

    def is_75_move_rule(self):
        """
        Checks for 75 move rule, returns boolean
        """
        #75 move rule, 150th move game stops
        if self.move_count==150:
            return True
        return False

    def is_threefold_repetition(self):
        """
        Checks for threefold repetition, forces draw, returns boolean
        """
        repcount=1
        for state in self.states:
            if state==self.board:
                repcount+=1
            if repcount⩾3:
                return True
        return False

    def contains(self,pos,piece):
        """
        if a specific position contains a specific piece, pos as number piece
as string
        """
        if pos is None:
            return False
        if self.board[pos]==piece:
            return True
        return False

    def enpassant(self,pos):
        """
        checks either side of a pawn thats moved up 2 for opposite color pawns
        if so stores opposite color pawn positions and valid move
        """
        opp1=self.move_pos(pos,1,0)
        opp2=self.move_pos(pos,-1,0) # adjacent squares
        self.enpassant_check=[]
        if self.turn==self.WHITE:
            if self.contains(opp1,"P") or self.contains(opp2,"P"):
                self.enpassant_check=[opp1,opp2,pos+8]
```

```
        elif self.turn==self.BLACK:
            if self.contains(opp1,"p") or self.contains(opp2,"p"):
                self.enpassant_check=[opp1,opp2,pos-8]

    def promotion(self,piece,dst):
        """
        if pawn on opposite end becomes queen for simplicity
        """
        if piece=="p" and dst//8==0:
            self.board[dst]="q"
        if piece=="P" and dst//8==7:
            self.board[dst]="Q"

    def winlossdraw(self):
        """
        runs all functions related to win/loss/draw, returns state in string
        """
        if self.turn==self.BLACK:
            king_pos=self.find_piece("K")[0]
            if self.is_checkmate(king_pos):
                return "white win"
            elif self.is_stalemate(king_pos):
                return "draw"
            elif self.is_75_move_rule():
                return "draw"
            elif self.is_threefold_repetition():
                return "draw"
            elif self.is_check(king_pos):
                return "black in check"
        else:
            king_pos=self.find_piece("k")[0]
            if self.is_checkmate(king_pos):
                return "black win"
            elif self.is_stalemate(king_pos):
                return "draw"
            elif self.is_75_move_rule():
                return "draw"
            elif self.is_threefold_repetition():
                return "draw"
            elif self.is_check(king_pos):
                return "white in check"
        return ""

    def game_over(self):
        """
        checks if game is over
        """
```

```
        return self.winlossdraw() in ["white win", "draw", "black win"]

    def copy(self):
        """
        return deep copy of this board without using copy.deepcopy as it's
slow
        """
        copy=Board.__new__(Board) # makes uninitialised board
        #shallow copy class variables
        copy.board=self.board[:]
        copy.turn=self.turn
        copy.move_count=self.move_count
        copy.states=self.states[:]
        copy.enpassant_check=self.enpassant_check[:]
        copy.unmoved=self.unmoved.copy()
        copy.clear_caches()
        return copy

    def is_move_out_of_check(self,src,dst):
        """
        does move on copy of board and sees if still in check
        """
        board=self.copy()
        board.board[src],board.board[dst]=".",board.board[src]
        if board.turn==board.BLACK:
            king_pos=board.find_piece("K")[0]
        else:
            king_pos=board.find_piece("k")[0]
        return not board.is_check(king_pos)

    def valid_move_src_dst(self,turn,checkcheck=True):
        srcdstlist=[]
        for i in self.notation:
            if
self.check_piece_color(self.board[self.algebraic_to_pos(i)])!=turn:
                continue
            try:
                new_moves=self.piece_moves(i,checkcheck=checkcheck,turn=turn)
                for move in new_moves:
                    srcdstlist.append((i,self.notation[move]))
            except EmptySquareError as e:
                print(e)
        return srcdstlist

    def board_to_fen(self):
        """
        takes current board and returns it in fen notation
```

```
        """
        fen=""
        for row in [self.board[i:i+8] for i in range(0, len(self.board), 8)]:
            space_count=0
            for char in row:
                if char==".":
                    space_count+=1
                else:
                    if space_count>0:
                        fen+=str(space_count)
                        space_count=0
                    if char in ["P","R","B","N","K","Q"]: # colours are other
way round
                        fen+=char.lower()
                    else:
                        fen+=char.upper()
            if space_count>0:
                fen+=str(space_count)
            fen+="/"
        #add turn
        fen = fen[:-1]
        if self.turn==self.WHITE:
            fen+=" w"
        else:
            fen+=" b"
        return fen
```

Chess/UI

```
"""
Implementation of UI
"""

from chess.board import *

class UI:
    """
    This class defines ways of displaying the board to the user
    """
    def __init__(self):
        self.board=Board()

    def player_input(self,input=input):
        src=input("Original position of piece:")
        dst=input("Destination of piece:")
        return src,dst

    def simple_display(self):
```

```
        self.board.display()

    def main_loop(self):
        """
        Runs the game loop
        """
        while not self.board.game_over():
            self.simple_display()
            okay=False
            while not okay:
                src,dst=self.player_input()
                try:
                    self.board.move(src,dst)
                    okay=True
                except ChessError as e:
                    print(e.__class__.__name__,e)
        self.simple_display()

if __name__=="__main__":
    a=UI()
    a.main_loop()
```

Chess/SimpleEvaluationMixin

```
class SimpleEvaluationMixin:
    """
    Simple evaluation based on Claude Shannon's evaluation
    """
    def evaluate(self,b):
        """
        returns number representing favour of board
        """
        winlossdraw=b.winlossdraw()
        if winlossdraw=="white win":
            return 1000
        elif winlossdraw=="black win":
            return -1000
        elif winlossdraw=="draw":
            return 0
        #find number of each pieces
        piecedict={}
        for i in range(64):
            piece=b.board[i]
            piecedict.setdefault(piece,0)
            piecedict[piece]+=1
        #return evaluation
```

```
        eval=200*(piecedict.get("k",0)-
piecedict.get("K",0))+9*(piecedict.get("q",0)-
piecedict.get("Q",0))+5*(piecedict.get("r",0)-
piecedict.get("R",0))+3*(piecedict.get("b",0)-piecedict.get("b",0) +
piecedict.get("n",0)-piecedict.get("N",0))+(piecedict.get("p",0)-
piecedict.get("P",0))
        return eval
```

Chess/AIEvaluationMixin

```
#suppress tensorflow messages
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
os.environ['TF_ENABLE_ONEDNN_OPTS'] = '0'

from AI.fen_to_tensor import fen_to_tensor
import tensorflow as tf
import numpy as np

class AIEvaluationMixin:
    """
    AI evaluation
    """
    def __init__(self,model):
        self.model=tf.keras.models.load_model(model)
        #model=tf.keras.models.load_model("chess7500000.keras")

    @tf.function
    def infer(self,x):
        """
        speed up model inference
        """
        return self.model(x)

    def evaluate(self,b):
        """
        returns number representing favour of board
        """
        winlossdraw=b.winlossdraw()
        if winlossdraw=="white win":
            return 1000
        elif winlossdraw=="black win":
            return -1000
        elif winlossdraw=="draw":
            return 0
```

```
        fen=b.board_to_fen()
        input_tensor=fen_to_tensor(fen)
        input_tensor=np.expand_dims(input_tensor, axis=0) # model expects
batch
        # batching experiment
        # copies=1
        # input_tensor=np.tile(input_tensor,(copies,1,1,1))
        eval=self.infer(input_tensor)
        eval=float(eval[0][0])
        return eval
```

Chess/minimax

```
from chess.SimpleEvaluationMixin import SimpleEvaluationMixin
from chess.AIEvaluationMixin import AIEvaluationMixin

class MinimaxBase():
    """
    Base class for minimax
    """
    def __init__(self,board,depth):
        self.board=board
        self.depth=depth

    def best_move(self):
        """
        returns best move at set depth
        """
        return self.best_move_for_level(self.board,self.depth)[1]

    def best_move_for_level(self,board,depth):
        """
        returns best move and evaluation, for current level and below until
depth 0
        """
        movelist=board.valid_move_src_dst(board.turn)
        bestmovelist=[]
        for src,dst in movelist:
            copyboard=board.copy()
            copyboard.move(src,dst)
            if depth<=1:
                eval=self.evaluate(copyboard)
            else:
                eval,move=self.best_move_for_level(copyboard,depth-1)
            bestmovelist.append((eval,(src,dst)))
```

```
        # reverse if turn is white, sorts lowest first by default, lowest is
best evaluation for black
        # maximises for white, minimises for black
        bestmovelist.sort(reverse=board.turn==board.WHITE)
        return bestmovelist[0]

class Minimax(MinimaxBase,SimpleEvaluationMixin):
    def __init__(self,board,depth):
        MinimaxBase.__init__(self,board,depth)
        SimpleEvaluationMixin.__init__(self)

class MinimaxAI(MinimaxBase,AIEvaluationMixin):
    def __init__(self,board,depth):
        MinimaxBase.__init__(self,board,depth)
        AIEvaluationMixin.__init__(self,"AI/chess5M.keras")
```

Chess/alphabetapruning

```
from chess.SimpleEvaluationMixin import SimpleEvaluationMixin
from chess.AIEvaluationMixin import AIEvaluationMixin
from multiprocessing.pool import ThreadPool
import time

class AlphabetaPruningBase():
    """
    Base class for AlphabetaPruning
    """
    def __init__(self,board,depth):
        self.board=board
        self.depth=depth
        self.count=0

    def best_move(self):
        """
        returns best move at set depth
        """
        self.count=0
        start=time.time()
        eval=self.best_move_for_level(self.board,self.depth,-1000,1000)
        dt=time.time()-start
        print(f"Evaluations done: {self.count}, Best evaluation: {eval[0]},
Time taken: {round(dt,4)}")
        return eval[1]

    def best_move_for_level(self,board,depth,alpha,beta):
        """
```

```
        returns best move and evaluation, for current level and below until
depth 0
        uses alpha-beta pruning to reduce searching, pseudo code wikipedia
        """
        if depth==0:
            self.count+=1
            return self.evaluate(board),None
        movelist=board.valid_move_src_dst(board.turn)
        movelist=self.sort_moves(board,movelist)
        bestmovelist=[]

        # multi threading experiment
        if False:
        # if depth==self.depth:
            with ThreadPool() as pool:
                def task(move):
                    """
                    uses a new thread for each sub tree of top level minimax
tree
                    """
                    src,dst=move
                    copyboard=board.copy()
                    copyboard.move(src,dst)
                    eval,move=self.best_move_for_level(copyboard,depth-
1,alpha,beta)
                    return eval,(src,dst)
                bestmovelist=list(pool.imap(task,movelist))
        else:
            if board.turn==board.WHITE: #maximising player
                eval=-1000
                for src,dst in movelist:
                    copyboard=board.copy()
                    copyboard.move(src,dst)
                    eval,move=self.best_move_for_level(copyboard,depth-
1,alpha,beta)
                    bestmovelist.append((eval,(src,dst)))
                    if eval>beta:
                        break #beta cut off
                    alpha=max(alpha,eval)

            else: #minimising player
                eval=1000
                for src,dst in movelist:
                    copyboard=board.copy()
                    copyboard.move(src,dst)
                    eval,move=self.best_move_for_level(copyboard,depth-
1,alpha,beta)
```

```
                    bestmovelist.append((eval,(src,dst)))
                    if eval<alpha:
                        break #alpha cut off
                    beta=min(beta,eval)

        bestmovelist.sort(reverse=board.turn==board.WHITE)
        if len(bestmovelist)==0:
            self.count+=1
            return self.evaluate(board),None
        return bestmovelist[0]

    def sort_moves(self,board,moves):
        """
        sort moves so takes are first to make alpha-beta pruning more
efficient
        """
        notake=[]
        take=[]
        for src,dst in moves:
            if board.board[board.algebraic_to_pos(dst)]!=".":
                take.append((src,dst))
            else:
                notake.append((src,dst))
        return take+notake

class AlphabetaPruning(AlphabetaPruningBase,SimpleEvaluationMixin):
    def __init__(self,board,depth):
        AlphabetaPruningBase.__init__(self,board,depth)
        SimpleEvaluationMixin.__init__(self)

class AlphabetaPruningAI(AlphabetaPruningBase,AIEvaluationMixin):
    def __init__(self,board,depth):
        AlphabetaPruningBase.__init__(self,board,depth)
        AIEvaluationMixin.__init__(self,"AI/chess5M.keras")
```

GUI/gui

```
import sys
import pygame
from chess import UI
from chess.minimax import Minimax,MinimaxAI
from chess.alphabetapruning import AlphabetaPruning,AlphabetaPruningAI
import time

class GUI:
    screen_width = 800
    screen_height = 600
```

```
    boardconstant = 0.85
    board_x=(screen_width-screen_height*boardconstant)/2
    board_y=screen_height*((1-boardconstant)/2)
    board_side=screen_height*boardconstant

    piecedict={
        "p":"gui/pawnw.png",
        "b":"gui/bishopw.png",
        "P":"gui/pawnb.png",
        "B":"gui/bishopb.png",
        "n":"gui/knightw.png",
        "N":"gui/knightb.png",
        "r":"gui/rookw.png",
        "R":"gui/rookb.png",
        "k":"gui/kingw.png",
        "K":"gui/kingb.png",
        "q":"gui/queenw.png",
        "Q":"gui/queenb.png"
    }

    pieceimages={}
    for key,file in piecedict.items():
        pieceimages[key]=pygame.image.load(file)

    def __init__(self):
        pygame.init()
        self.board=UI.Board(chess960=False)
        self.screen =
pygame.display.set_mode((self.screen_width,self.screen_height))
        pygame.display.set_caption("DOUGAL CHESS GUI")
        self.image_list=[]
        self.drag_pos=None
        self.square_list=[]
        self.valid_moves=[]
        self.font= pygame.font.SysFont("arialblack", 30)
        self.status=""
        self.players=[AlphabetaPruningAI,None] # [black,white]
        self.depth=2
        self.drag=False
        self.drag_image=None
        self.offset_x,self.offset_y=0,0
        self.run=True

    def main(self):
        """
        main loop, select menu then draws board and status then does player
and non-player moves
```

```
        """
        self.players[1]=self.choose_player("White")
        self.players[0]=self.choose_player("Black")
        if self.players[1] is not None or self.players[0] is not None:
            self.depth=self.menu("Depth?",[
                ["Depth 1",1],
                ["Depth 2",2],
                ["Depth 3",3],
            ])
        chess960=self.menu("Chess 960?",[
            ["Yes",True],
            ["No",False]
        ])
        self.board=UI.Board(chess960=chess960)
        self.draw_board()
        pygame.display.flip()
        while self.run:
            player=self.players[self.board.turn]
            is_human = player is None
            self.src,self.dst = None,None
            self.event_handler(is_human)
            if not self.board.game_over():
                self.make_move(player, is_human)
            self.draw_board()
            if self.drag:
                self.screen.blit(self.drag_image[1],self.drag_image[0])
            self.draw_text(self.status,0.95,(255,0,0))
            pygame.display.flip()
            time.sleep(1/90) # runs at limited fps
        pygame.quit()
        sys.exit()

    def choose_player(self,colour):
        """
        displays menu options for the player
        """
        return self.menu(f"Choose {colour} Player",[
            ["Human",None],
            ["Computer no AI", AlphabetaPruning],
            ["Computer with AI", AlphabetaPruningAI]
        ])

    def menu(self,heading,options):
        """
        displays menu for game options
        """
        rects=[]
```

```
        self.screen.fill((128,128,128))
        self.draw_text(heading,0.1,(0,0,0))
        i=0
        for option,value in options:
            i+=0.1
            text_rect=self.draw_text(option,0.2+i,(255,0,0))
            rects.append((text_rect,value))
        pygame.display.flip()
        while self.run:
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    self.run=False
                elif event.type==pygame.MOUSEBUTTONDOWN:
                    for rect,value in rects:
                        if rect.collidepoint(event.pos):
                            return value

    def make_move(self, player, is_human):
        """
        makes either player move or engine move
        """
        if not is_human:
            m=player(self.board,self.depth)
            self.src,self.dst=m.best_move()
        if self.src is not None:
            try:
                self.board.move(self.src,self.dst)
                self.status=self.board.winlossdraw()
                self.board.display()
            except UI.ChessError as e:
                print(e)
            self.src,self.dst=None,None

    def event_handler(self, is_human):
        """
        handles mouse events
        """
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                self.run=False
            elif is_human and event.type==pygame.MOUSEBUTTONDOWN:
                self.event_buttondown(event)
            elif is_human and event.type == pygame.MOUSEBUTTONUP:
                self.event_buttonup(event)
            elif is_human and event.type==pygame.MOUSEMOTION:
                self.event_mousemotion(event)
```

```
    def event_mousemotion(self, event):
        """
        handles mouse motion
        """
        if self.drag:
            self.drag_image[0].x=event.pos[0]-self.offset_x
            self.drag_image[0].y=event.pos[1]-self.offset_y

    def event_buttonup(self, event):
        """
        handles mouse button up
        """
        self.drag=False
        self.drag_image=None
        for rect,pos in self.square_list:
            if rect.collidepoint(event.pos) and self.drag_pos is not None:
                src_pos=self.drag_pos
                dst_pos=pos
                self.src=self.board.notation[src_pos]
                self.dst=self.board.notation[dst_pos]
        self.valid_moves=[]
        self.drag_pos=None

    def event_buttondown(self, event):
        """
        handles mouse button down
        """
        for rect,image,pos in self.image_list:
            if rect.collidepoint(event.pos):
                self.drag=True
                self.drag_image=(rect,image)
                self.offset_x=event.pos[0]-rect.x
                self.offset_y=event.pos[1]-rect.y
                self.drag_pos=pos
                if
self.board.check_piece_color(self.board.board[pos])==self.board.turn:
                    self.valid_moves=self.board.piece_moves(self.board.notatio
n[pos])
                else:
                    self.drag=False
                    self.drag_pos=None

    def draw_text(self,text,height_fraction,colour):
        """
        draws text to the screen at specificied height
        """
```

```
        text_surface = self.font.render(text, True, colour)  # Text,
antialiasing, color (black)
        text_rect = text_surface.get_rect()
        # Set the position (x, y) of the text rectangle
        text_rect.center = (self.screen_width // 2,
self.screen_height*height_fraction)  # Center the text
        self.screen.blit(text_surface, text_rect)
        return text_rect

    def draw_board(self):
        """
        draws the board onto the screen, flips screen if human black player
        """
        flip=False
        if self.players[self.board.turn] is None and
self.board.turn==self.board.BLACK:
            flip=True
        elif self.players[0] is None and self.players[1] is not None:
            flip=True
        self.image_list=[]
        self.square_list=[]
        self.screen.fill((128,128,128))
        gridconstant=self.board_side/8
        black=True
        for x in range(8):
            black=not black
            for y in range(8):
                if flip:
                    x=7-x
                    y=7-y
                color=(92,64,51)
                if not black:
                    color=(255,255,255)
                rect=pygame.Rect(self.board_x+x*gridconstant,self.board_y+y*gr
idconstant,gridconstant,gridconstant)
                pygame.draw.rect(self.screen,color,rect)
                if flip:
                    pos=(7-x)+(7-y)*8
                else:
                    pos=x+y*8
                self.square_list.append((rect,pos))
                if pos in self.valid_moves:
                    highlight=pygame.Surface((rect.width,rect.height),pygame.S
RCALPHA)
                    highlight.fill((100,100,0,200))
                    self.screen.blit(highlight,rect)
                i=self.board.board[pos]
```

```
                if pos!=self.drag_pos:
                    if i!=".":
                        self.screen.blit(self.pieceimages[i],(self.board_x+x*g
ridconstant,self.board_y+y*gridconstant))
                        self.image_list.append((rect,self.pieceimages[i],pos))
                if not flip:
                    black=not black

if __name__=="__main__":
    gui=GUI()
    gui.main()
```

AI/AI

```
import numpy as np
import fen_to_tensor
import tensorflow as tf
import csv

# import data from csv
# example "7r/1p3k2/p1bPR3/5p2/2B2P1p/8/PP4P1/3K4 b - -",58
dataset="lichess_db_eval.csv"

def load_csv(data):
    """
    generator for reading lines out of large csv
    """
    with open(data,"r") as file:
        reader=csv.reader(file)
        for row in reader:
            board_fen,eval=row[0],row[1]
            yield (board_fen,eval)

size=100000

#training input
x_train=np.empty((size,8,8,13))
#desired output
y_train=np.empty(size,dtype=np.float32)

i=0
for board_fen,eval in load_csv(dataset):
    x_train[i]=fen_to_tensor.fen_to_tensor(board_fen)
    y_train[i]=float(eval)/100 #convert from centipawns to pawns
    i+=1
    if i==size:
        break
```

```
# model

def create_model():
    """
    return tensor flow model for chess evaluation
    """
    model=tf.keras.models.Sequential()
    #input
    model.add(tf.keras.layers.Input(shape=(8,8,13)))
    #convolution 1
    model.add(tf.keras.layers.Conv2D(32,(3,3),padding="same"))
    model.add(tf.keras.layers.BatchNormalization())
    model.add(tf.keras.layers.Activation("relu"))
    #convolution 2
    model.add(tf.keras.layers.Conv2D(64,(3,3),padding="same"))
    model.add(tf.keras.layers.BatchNormalization())
    model.add(tf.keras.layers.Activation("relu"))
    #flatten
    model.add(tf.keras.layers.Flatten())
    #dense 1
    model.add(tf.keras.layers.Dense(128,activation="relu"))
    model.add(tf.keras.layers.Dropout(0.5))
    #dense 2
    model.add(tf.keras.layers.Dense(64,activation="relu"))
    model.add(tf.keras.layers.Dropout(0.5))
    #output
    model.add(tf.keras.layers.Dense(1,activation="linear"))
    #compile
    model.compile(optimizer="adam",loss="mean_squared_error")
    return model

model=create_model()
#tf.keras.utils.plot_model(model,show_shapes=True)

model.fit(x_train,y_train,epochs=3,batch_size=32,validation_split=0.1)
model.save("chess.keras")
```

AI/AITrain

```
import numpy as np
import fen_to_tensor
import tensorflow as tf
from tensorflow.keras import regularizers # type: ignore
import csv

# import data from csv
```

```
# example "7r/1p3k2/p1bPR3/5p2/2B2P1p/8/PP4P1/3K4 b - -",58
dataset="lichess_db_eval.csv"

def load_csv(data):
    """
    generator for reading lines out of large csv
    """
    with open(data,"r") as file:
        reader=csv.reader(file)
        for row in reader:
            board_fen,eval=row[0],row[1]
            yield (board_fen,eval)

def create_model():
    """
    Return a TensorFlow model for chess evaluation
    """
    input_layer = tf.keras.layers.Input(shape=(8, 8, 13))

    #convolutional block 1
    #correlates features in different places of the input producing 32x8x8
output layers
    x = tf.keras.layers.Conv2D(32, (5, 5), padding="same",
kernel_regularizer=regularizers.l2(0.001))(input_layer)
    x = tf.keras.layers.BatchNormalization()(x)
    x = tf.keras.layers.Activation("relu")(x)

    #convolutional block 2
    #correlate that into 64 higher level features 64x8x8
    x = tf.keras.layers.Conv2D(64, (3, 3), padding="same",
kernel_regularizer=regularizers.l2(0.001))(x)
    x = tf.keras.layers.BatchNormalization()(x)
    x = tf.keras.layers.Activation("relu")(x)

    #residual connection
    residual = tf.keras.layers.Conv2D(64, (1, 1), padding="same")(input_layer)
    x = tf.keras.layers.Add()([x, residual])

    #convolutional block 3
    #correlate into higher level features still 128x8x8
    x = tf.keras.layers.Conv2D(128, (3, 3), padding="same",
kernel_regularizer=regularizers.l2(0.001))(x)
    x = tf.keras.layers.BatchNormalization()(x)
    x = tf.keras.layers.Activation("relu")(x)

    #Global Average Pooling
    #collapse into 128 features
```

```
    x = tf.keras.layers.GlobalAveragePooling2D()(x)

    #dense layers
    #correlate the features
    x = tf.keras.layers.Dense(128, activation="relu",
kernel_regularizer=regularizers.l2(0.001))(x)
    x = tf.keras.layers.Dropout(0.3)(x)  # Reduced dropout rate

    x = tf.keras.layers.Dense(64, activation="relu",
kernel_regularizer=regularizers.l2(0.001))(x)
    x = tf.keras.layers.Dropout(0.3)(x)  # Reduced dropout rate

    #output layer
    #linear output layer to produce the evaluation in pawns directly
    output = tf.keras.layers.Dense(1, activation="linear")(x)

    #compile model
    model = tf.keras.models.Model(inputs=input_layer, outputs=output)
    #model.compile(optimizer="adam", loss="mean_squared_error")
    # Huber loss function is better at outliers which we have some of
    model.compile(optimizer="adam", loss=tf.keras.losses.Huber())

    return model


def train(size=1000000,epochs=50,batch_size=32):
    """
    Create and train the model on size items from the dataset
    """
    model=create_model()

    #create an image of the model architecture
    dot_img_file = 'model_architecture.png'
    tf.keras.utils.plot_model(model, to_file=dot_img_file, show_shapes=True)

    #reduce the learning rate if the validation loss plateaus
    reduce_lr = tf.keras.callbacks.ReduceLROnPlateau(monitor='val_loss',
factor=0.2, patience=5, min_lr=0.0001)

    #define the ModelCheckpoint callback to save the model as we progress
    save_model = tf.keras.callbacks.ModelCheckpoint(
        filepath='chess.keras',  # save model every epoch
        save_weights_only=False, # save the entire model (architecture +
weights + optimizer state)
        save_freq='epoch',       # save every epoch
        verbose=1                # verbosity mode, 1 = progress bar
    )
```

```
    #training input
    x_train=np.empty((size,8,8,13))
    #desired output
    y_train=np.empty(size,dtype=np.float32)

    i=0
    positive = 0
    negative = 0
    for board_fen,eval in load_csv(dataset):
        # evaluation in pawns, not centi-pawns
        eval = float(eval) / 100

        # balance the number of positive and negative evaluations
        # to within 1%
        if eval > 0:
            if positive > 1.01*negative:
                continue
            positive += 1
        elif eval < 0:
            if negative > 1.01*positive:
                continue
            negative += 1

        x_train[i]=fen_to_tensor.fen_to_tensor(board_fen)
        y_train[i]=eval
        i+=1
        if i==size:
            break

    model.fit(x_train,
              y_train,
              epochs=epochs,
              batch_size=batch_size,
              validation_split=0.1,
              callbacks=[reduce_lr, save_model],
    )
    model.save("chess.keras")

if __name__ == "__main__":
    train()
```

AI/fen_to_tensor

```
import numpy as np
```

```
# "rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1"

pieces={"r":0,"n":1,"b":2,"q":3,"k":4,"p":5,"R":6,"N":7,"B":8,"Q":9,"K":10,"P"
:11}

def fen_to_tensor(fen):
    """
    converts fen board notation to tensor of board and turn colour
    """
    board,turn,*_=fen.split() # ignore all except board and turn (simplicity)
    board_tensor = np.zeros((8,8,12))
    rows=board.split("/")
    for row_i,row in enumerate(rows):
        col_i=0
        for char in row:
            if char.isdigit():
                col_i+=int(char)
            else:
                piece=pieces[char]
                board_tensor[row_i,col_i,piece]=1 # add to tensor
                col_i+=1
    if turn=="w":
        turn_tensor=np.ones((8,8,1))
    else:
        turn_tensor=np.zeros((8,8,1))
    return np.concatenate([board_tensor,turn_tensor],axis=2) # axis 2 makes
8x8x12 into 8x8x13

#print(fen_to_tensor("rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0
1"))
```

Tests/profileboard

```
import cProfile
from chess import board
from chess.alphabetapruning import *

b=board.Board()
a=AlphabetaPruning(b,2)
cProfile.run("a.best_move()")
```

Tests/test_against_stockfish

```
from stockfish import Stockfish
from pprint import pprint

from chess.board import Board
```

```
from chess.minimax import Minimax,MinimaxAI
from chess.alphabetapruning import AlphabetaPruning,AlphabetaPruningAI

stockfish_path=r"C:\Users\dougal\Documents\stockfish\stockfish-windows-x86-
64.exe"
player=AlphabetaPruning


def play_game(my_depth,stockfish_elo,stats_dict):
    stockfish=Stockfish(stockfish_path)
    b=Board(chess960=False)

    stockfish.set_elo_rating(stockfish_elo)

    while True:
        status=b.winlossdraw()
        if status=="black win":
            stats_dict["stockfish win"]+=1
            break
        if status=="white win":
            stats_dict["me win"]+=1
            break
        if status=="draw":
            stats_dict["draw"]+=1
            break
        if b.turn==b.BLACK:
            move=stockfish.get_best_move()
            #print(f"stockfish best move {move}")
            # remove promoted piece from end
            if len(move) == 5:
                move = move[:4]
            assert len(move)==4, f"unkown move format {move}"
            # correcting algebraic notation for castling which is optimised
for 960
            if move=="e1g1":
                move="e1h1"
            elif move=="e1c1":
                move="e1a1"
            elif move=="e8g8":
                move="e8h8"
            elif move=="e8c8":
                move="e8a8"
            src,dst=move[:2],move[2:]
        else:
            src,dst=player(b,my_depth).best_move()
            #print(f"our best move {src} {dst}")
```

```
        #print(f"we move {src} {dst}")
        b.move(src,dst)
        # correcting algebraic notation for castling which is optimised for
960
        move=src+dst
        if move=="e1h1":
            move="e1g1"
        elif move=="e1a1":
            move="e1c1"
        elif move=="e8h8":
            move="e8g8"
        elif move=="e8a8":
            move="e8c8"
        assert stockfish.is_move_correct(move), f"stockfish think we aint
valid {move}"
        #print(f"stockfish moves {move}")
        stockfish.make_moves_from_current_position([move])

        b.display()
        # print(stockfish.get_board_visual())
        print(b.turn)
        print("----------------------------------------------------------------
------")

def main():
    final_stats={}
    # for depth in [1,2,3]:
    for depth in [3]:
        # for level in [500,1000,1500,2000,2500]:
        for level in [500]:
            stats_dict={"stockfish win": 0, "me win": 0, "draw": 0, "errors":
0}
            while stats_dict["stockfish win"]+stats_dict["me
win"]+stats_dict["draw"]<10:
            # while stats_dict["me win"]<1:
                try:
                    play_game(depth,level,stats_dict)
                except Exception as e:
                    print(f"Game failed with {repr(e)}")
                    stats_dict["errors"]+= 1
                pprint(stats_dict)
                print("================================================
===============")
            final_stats[depth,level]=stats_dict
            stats_dict={"stockfish win": 0, "me win": 0, "draw": 0, "errors":
0}
    pprint(final_stats)
```

```
if __name__ == "__main__":
    main()
```

Tests/test_AIEvaluationMixin

```
"""
SimpleEvaluationMixin unit test
"""

from chess.AIEvaluationMixin import *
from chess import board
import unittest

class test_AIEvaluationMixin(unittest.TestCase):
    def test_init(self):
        pass

    def test_evaluate(self):
        e=AIEvaluationMixin("AI/chess5M.keras")
        b=board.Board("RNBQKBNR"
                      "PPPPPPPP"
                      "........"
                      "........"
                      "........"
                      "........"
                      "pppppppp"
                      "rnbqkbnr")
        self.assertTrue(-0.5 < e.evaluate(b) < 0.5)

        b=board.Board("..BQKBNR"
                      "..PPPPPP"
                      "........"
                      "........"
                      "........"
                      "........"
                      "pppppppp"
                      "rnbqkbnr")
        self.assertAlmostEqual(6.8,e.evaluate(b),delta=1)

        b=board.Board(".R..R.K."
                      ".....PP."
                      "P..B.NQP"
                      ".PPp...."
                      "....p..."
```

```
                "..n..q.."
                "pp...ppp"
                ".r..r.k.")
        self.assertAlmostEqual(-3.5,e.evaluate(b),delta=1)

        b=board.Board("R.BQ.RK."
                "PPP...PP"
                "..N..N.."
                ".b.P...."
                "...p...."
                "..nb.n.."
                "ppp..ppp"
                "r..q.rk.")
        self.assertAlmostEqual(4.3,e.evaluate(b),delta=1)

        b=board.Board("........"
                "......K."
                ".....P.."
                ".....pP."
                "........"
                "...R...."
                ".r......"
                ".k......")
        self.assertAlmostEqual(-6.9,e.evaluate(b),delta=1)

        b=board.Board("........"
                "........"
                "...K...."
                "...P...."
                "...p.P.."
                ".....p.."
                ".....k.."
                "........")
        self.assertAlmostEqual(5.4,e.evaluate(b),delta=6)

        b=board.Board(".R..QRK."
                "PP....pp"
                "..B..n.."
                ".p.p...."
                "....P..."
                "..Nb.N.."
                "pp...pp."
                "r..q.rk.")
        self.assertAlmostEqual(3.2,e.evaluate(b),delta=1)

        b=board.Board("r..qk..r"
                "pp..bp.."
```

```
                    ".n..pnp."
                    ".....p.."
                    ".PpP.P.."
                    "..N.BN.."
                    "P..Q..PP"
                    "R..R..K.")
        self.assertAlmostEqual(0,e.evaluate(b),delta=1)

        b=board.Board("R...k..."
                    ".P..pp.."
                    "...p...."
                    "..r....."
                    ".....P.."
                    "...P...."
                    "p....pPP"
                    ".....K..")
        print(e.evaluate(b))
        self.assertAlmostEqual(0,e.evaluate(b),delta=1)

        b=board.Board("....K..."
                    ".P......"
                    "....p..."
                    "....p.p."
                    "P......."
                    "..r....."
                    "..k..pp."
                    ".......")
        print(e.evaluate(b))
        self.assertAlmostEqual(13.4,e.evaluate(b),delta=1)

        b=board.Board("....k..."
                    "....P..."
                    "p......P"
                    "..K....."
                    "......."
                    ".r......"
                    "......pp"
                    ".......")
        print(e.evaluate(b))
        self.assertAlmostEqual(14,e.evaluate(b),delta=1)


if __name__ == '__main__':
    unittest.main()
```

Tests/test_alphabetapruning

```
"""
alphabetapruning unit test
"""

from chess.alphabetapruning import *
from chess import board
import unittest

class test_alphabetapruning(unittest.TestCase):
    def test_init(self):
        pass

    def test_best_move_white(self):
        b=board.Board("k......K"
                ".......p"
                "........"
                "........"
                ".p..p..."
                "........"
                "...P...."
                "..p.....")
        # m=AlphabetaPruning(b,1)
        m=AlphabetaPruningAI(b,1)
        self.assertEqual(("c1","d2"),m.best_move())
        b=board.Board("k......K"
                ".......p"
                "........"
                "...P...."
                "....n..."
                "P......."
                ".p......"
                "........")
        #move differs depending on evaluation
        # m=AlphabetaPruning(b,2)
        # self.assertEqual(('e4', 'g5'),m.best_move())

        m=AlphabetaPruningAI(b,2)
        # depth 2 works, sees its about to be taken
        self.assertEqual(('e4', 'f6'),m.best_move())
        b=board.Board(".......K"
                ".......p"
                "....r..."
                "...r...."
                "........"
                "........"
                "........"
```

```
                "...k....")
        # m=AlphabetaPruning(b,3)
        m=AlphabetaPruningAI(b,3)
        #mate in 2
        self.assertEqual(("e6","e7"),m.best_move())

    def test_best_move_black(self):
        b=board.Board("k......K"
                "........"
                "........"
                "........"
                ".p..p..."
                "........"
                "...P...."
                "..p.....")
        b.turn=b.BLACK
        #move differs depending on evaluation
        # m=AlphabetaPruning(b,1)
        # self.assertEqual(('d2', 'c1'),m.best_move())

        m=AlphabetaPruningAI(b,1)
        self.assertEqual(('d2', 'd1'),m.best_move())
        b=board.Board("k......K"
                "........"
                ".P......"
                "p......."
                "....N..."
                "...p...."
                "........"
                "........")
        b.turn=b.BLACK
        #move differs depending on evaluation
        # m=AlphabetaPruning(b,2)
        # self.assertEqual(('e4', 'c3'),m.best_move())

        m=AlphabetaPruningAI(b,2)
        # depth 2 works, sees its about to be taken
        self.assertEqual(('b6', 'a5'),m.best_move())
        b=board.Board(".......K"
                "........"
                "........"
                "........"
                "........"
                "....RR.."
                "........"
                "k......")
        b.turn=b.BLACK
```

```
        # m=AlphabetaPruning(b,3)
        m=AlphabetaPruningAI(b,3)
        #mate in 2
        self.assertEqual(('e3', 'e2'),m.best_move())


if __name__ == '__main__':
    unittest.main()
```

Tests/test_board

```
"""
Unit Test for Board class
"""

import unittest
from chess.board import *

class TestBoard(unittest.TestCase):
    def test_init(self):
        b=Board()
        for i,piece in enumerate(Board.startposition):
            self.assertEqual(b.board[i], piece)

    def test_algebraic_to_pos(self):
        b=Board()
        self.assertEqual(0,b.algebraic_to_pos("a8"))
        self.assertEqual(20,b.algebraic_to_pos("e6"))
        self.assertRaises(InvalidAlgebraicNotationError,b.algebraic_to_pos,"k8
")

    def test_is_valid(self):
        b=Board()
        self.assertRaises(EmptySquareError,b.is_valid,"e6","e7")
        self.assertRaises(InvalidAlgebraicNotationError,b.is_valid,"k7","e4")
        self.assertRaises(InvalidAlgebraicNotationError,b.is_valid,"e4","k7")

    def test_check_piece_color(self):
        b=Board()
        self.assertEqual(1,b.check_piece_color("p"))
        self.assertEqual(0,b.check_piece_color("P"))
        self.assertEqual(None,b.check_piece_color("."))

    def test_move_pos(self):
        b=Board()
        self.assertEqual(7,b.move_pos(0,7,0))
        self.assertEqual(None,b.move_pos(0,-1,-1))
```

```
    def test_take_okay(self):
        b=Board()
        self.assertEqual(True,b.take_okay(10,b.WHITE))
        self.assertEqual(True,b.take_okay(51,b.BLACK))
        self.assertEqual(False,b.take_okay(35,b.WHITE))
        self.assertEqual(False,b.take_okay(None,b.WHITE))

    def test_pawn(self):
        b=Board()
        self.assertEqual([41,33],b.pawn(49,b.WHITE))
        self.assertEqual([19,27],b.pawn(11,b.BLACK))
        b=Board("RNBQKBNR"
                    "PPPPPP.P"
                    "...p..P."
                    "........"
                    "........"
                    "..p....."
                    "pp..pppp"
                    "rnbqkbnr")
        self.assertEqual([12,10],b.pawn(19,b.WHITE))
        self.assertEqual([34],b.pawn(42,b.WHITE))
        self.assertEqual(".",b.board[50])
        self.assertEqual([],b.pawn(11,b.BLACK))
        self.assertEqual([30],b.pawn(22,b.BLACK))

    def test_knight(self):
        b=Board()
        self.assertEqual([43,41],b.knight(58,b.turn))
        b=Board("........"
                "....N..."
                "........"
                "...n.N.."
                "........"
                "........"
                "........"
                "........")
        self.assertEqual([44, 42, 37, 21, 12, 10, 33, 17],b.knight(27,b.turn))
        b.turn=b.BLACK
        self.assertEqual([27, 22, 6, 18, 2],b.knight(12,b.turn))

    def test_linear_move(self):
        b=Board("........"
                "........"
                "........"
                "........"
                "........"
                "...P...."
```

```
                "........"
                "........"
                "b......")
        self.assertEqual([49,42,35],b.linear_move(56,1,-1,b.turn))

    def test_bishop(self):
        b=Board("........"
                "........"
                "........"
                "........"
                "...P...."
                "........"
                "...b...."
                "........")
        self.assertEqual([44, 37, 30, 23, 42, 33, 24, 60,
58],b.bishop(51,b.WHITE))

    def test_rook(self):
        b=Board("........"
                "........"
                "........"
                "........"
                "...P...."
                "........"
                "...r...."
                "........")
        self.assertEqual([52, 53, 54, 55, 43, 35, 50, 49, 48,
59],b.rook(51,b.WHITE))

    def test_queen(self):
        b=Board("........"
                "........"
                "........"
                "........"
                "...P...."
                "........"
                "...q...."
                "........")
        self.assertEqual([44, 37, 30, 23, 42, 33, 24, 60, 58,52, 53, 54, 55,
43, 35, 50, 49, 48, 59],b.queen(51,b.WHITE))

    def test_king(self):
        b=Board("........"
                "........"
                "........"
                "........"
                "........"
                "........"
```

```
            "........"
            "...k...."
            "........")
        self.assertEqual([52, 59, 60, 50, 58, 43, 44, 42],b.king(51,b.WHITE))
        b=Board("........"
            "........"
            "........"
            "........"
            "....R..."
            "........"
            "...k...."
            "........")
        self.assertEqual([59, 50, 58, 43, 42],b.king(51,b.WHITE))
        self.assertRaises(InvalidMoveError,b.move,"d2","e2")
        b=Board("........"
            "........"
            "........"
            "........"
            "........"
            "........"
            "...K...."
            "........")
        b.turn=b.BLACK
        self.assertEqual([52, 59, 60, 50, 58, 43, 44, 42],b.king(51,b.BLACK))
        b=Board("........"
            "........"
            "........"
            "........"
            "....r..."
            "........"
            "...K...."
            "........")
        b.turn=b.BLACK
        self.assertEqual([59, 50, 58, 43, 42],b.king(51,b.BLACK))
        self.assertRaises(InvalidMoveError,b.move,"d2","e2")
        b=Board("........"
            "........"
            "...Kp..."
            "........"
            "..p.r..."
            "........"
            "........"
            "........")
        b.turn=b.BLACK
        self.assertRaises(InvalidMoveError,b.move,"d6","e6")
        self.assertRaises(InvalidMoveError,b.move,"d6","d5")
        b=Board("....K..."
```

```
                "...p...."
                "........"
                "........"
                "........"
                "........"
                "........"
                "...q....")
        b.turn=b.BLACK
        self.assertRaises(InvalidMoveError,b.move,"e8","d7")


    def test_piece_moves(self):
        b=Board(".......K"
                "........"
                "........"
                "........"
                "...P...."
                "........"
                "........"
                "b......k")
        self.assertEqual([49,42,35],b.piece_moves("a1"))
        self.assertEqual([43],b.piece_moves("d4"))
        self.assertRaises(EmptySquareError,b.piece_moves,"b2")

    def test_all_valid_moves(self):
        b=Board(".......K"
                "........"
                "........"
                "........"
                "...P...."
                "........"
                "........"
                "b......k")
        self.assertEqual([49, 42, 35, 62, 55, 54],b.all_valid_moves(b.WHITE))
        self.assertEqual([15, 6, 14, 43],b.all_valid_moves(b.BLACK))

    def test_total_moves(self):
        b=Board()
        self.assertEqual(20,b.total_moves())

    def test_move(self):
        b=Board()
        b.move("a2","a4")
        self.assertEqual(list("RNBQKBNR"
                    "PPPPPPPP"
                    "........"
                    "........"
```

```
                "p......."
                "........"
                ".ppppppp"
                "rnbqkbnr"), b.board)
        self.assertRaises(InvalidMoveError,b.move,"a2","a4")
        self.assertRaises(InvalidMoveError,b.move,"h2","h5")

    def test_is_check(self):
        b=Board("....K..."
                "........"
                "........"
                "K...q..."
                "........"
                "........"
                "..Q....."
                "....k...")
        self.assertEqual(True,b.is_check(24))
        self.assertEqual(False,b.is_check(60))
        self.assertEqual(True,b.is_check(12,b.BLACK))
        self.assertEqual(False,b.is_check(11,b.BLACK))
        self.assertEqual(True,b.is_check(58,b.WHITE))
        self.assertEqual(True,b.is_check(4))
        self.assertRaises(EmptySquareError,b.is_check,56)
        b=Board("RNBQKBNR"
                "PPPPPPPP"
                "........"
                "....R..."
                "........"
                "........"
                "pppp.ppp"
                "rnbqkbnr")
        self.assertEqual(True,b.is_check(60))
        self.assertEqual(False,b.is_checkmate(60))
        b=Board()
        b.move("b1","c3")
        self.assertEqual(False,b.is_check(4))
        b.move("b8","c6")
        self.assertEqual(False,b.is_check(60))
        b.move("c3","b5")
        self.assertEqual(False,b.is_check(4))
        b.move("c6","b4")
        self.assertEqual(False,b.is_check(60))
        self.assertEqual([27, 20, 4, 16, 0],b.knight(10,b.turn))
        b.move("b5","c7")
        self.assertEqual(True,b.is_check(4))
        #b.move("b4","c2")
        #self.assertEqual(True,b.is_check(60))
```

```
    def test_find_piece_index(self):
        b=Board()
        self.assertEqual(60,b.find_piece_index("e1"))

    def test_find_piece(self):
        b=Board()
        self.assertEqual([60],b.find_piece("k"))
        self.assertEqual([8,9,10,11,12,13,14,15],b.find_piece("P"))

    def test_is_stalemate(self):
        b=Board("........"
                "...R...."
                ".R......"
                "....k..."
                "......R."
                ".....R.."
                "........"
                "........")
        self.assertEqual(True,b.is_stalemate(28))
        self.assertEqual(True,b.game_over())
        b=Board()
        b.turn=b.BLACK
        self.assertEqual(False,b.is_stalemate(4))

    def test_is_checkmate(self):
        b=Board("....r..K"
                "....r..."
                "........"
                "........"
                "........"
                "........"
                "........"
                "........")
        b.turn=b.BLACK
        self.assertEqual(True,b.is_checkmate(7))
        b=Board("RNB.KBNR"
                "PPPP.PPP"
                "....P..."
                "........"
                "......pQ"
                ".....p.."
                "ppppp..p"
                "rnbqkbnr")
        b.turn=b.WHITE
        self.assertEqual(0,len(b.all_valid_moves(b.turn,True)))
```

```
        b.turn=b.WHITE # checks if current side is checkmate
        self.assertEqual(True,b.is_checkmate(60))
        b=Board()
        self.assertEqual(False,b.is_checkmate(4))
        b=Board("....R..k"
                "....R..."
                "........"
                "........"
                "........"
                "........"
                "........"
                "........")
        self.assertEqual(True,b.is_checkmate(7))
        self.assertEqual(True,b.game_over())


    def test_is_75_move_rule(self):
        b=Board()
        b.move_count=149
        self.assertEqual(False,b.is_75_move_rule())
        b.move("a2","a4")
        self.assertEqual(True,b.is_75_move_rule())

    def test_is_threefold_repetition(self):
        b=Board("k......K"
                "........"
                "........"
                "........"
                "........"
                "........"
                "........"
                "........")
        b.move("a8","a7")
        self.assertEqual(False,b.is_threefold_repetition())
        b.turn=b.WHITE
        b.move("a7","a8")
        self.assertEqual(False,b.is_threefold_repetition())
        b.turn=b.WHITE
        b.move("a8","a7")
        self.assertEqual(False,b.is_threefold_repetition())
        b.turn=b.WHITE
        b.move("a7","a8")
        self.assertEqual(True,b.is_threefold_repetition())

    def test_enpassant(self):
        b=Board()
        b.move("a2","a3")
```

```
        b.move("e7","e5")
        b.move("a3","a4")
        b.move("e5","e4")
        b.piece_moves("d2")
        b.move("d2","d4")
        b.piece_moves("e4")
        self.assertEqual([36, 34, 43],b.enpassant_check)
        self.assertEqual([44,43],b.piece_moves("e4"))
        self.assertEqual([44,43],b.pawn(36,b.BLACK))
        self.assertEqual(b.board[35],"p")
        b.move("e4","d3")
        b=Board("RNB.KBNR"
                ".P......"
                "........"
                "........"
                "........"
                "........"
                "........"
                "rnbqkbnr")
        b.move("a1","a7")
        self.assertRaises(InvalidMoveError,b.move,"b7","a6")

    def test_castle(self):
        b=Board("R...K..R"
                "........"
                "........"
                "........"
                "........"
                "........"
                "p......."
                "r...kq..")
        b.move("e1","a1")
        self.assertEqual(False,b.is_check(58,b.WHITE))
        self.assertEqual(b.board[59],"r")
        self.assertEqual(b.board[58],"k")
        self.assertRaises(InvalidMoveError,b.move,"e8","h8")
        b.turn=b.WHITE
        b.move("f1","e2")
        self.assertEqual(True,b.is_check(4))
        self.assertRaises(InvalidMoveError,b.move,"e8","h8")
        b.turn=b.WHITE
        b.move("e2","d2")
        b.move("e8","h8")

    def test_winlossdraw(self):
        b=Board()
        self.assertEqual("",b.winlossdraw())
```

```
        b=Board("......k."
                "........"
                "........"
                "........"
                "...q...."
                "........"
                "........"
                "...K....")
        b.turn=b.BLACK
        self.assertEqual("black in check",b.winlossdraw()) # you are
'checking' the opposition
        b=Board("........"
                "...R...."
                ".R......"
                "....k..."
                "......R."
                ".....R.."
                "........"
                "K.......")
        self.assertEqual("draw",b.winlossdraw())
        b=Board("RNB.KBNR"
                "PPPP.PPP"
                "....P..."
                "........"
                "......pQ"
                ".....p.."
                "ppppp..p"
                "rnbqkbnr")
        self.assertEqual("black win",b.winlossdraw())
        b=Board("..RK.RBB"
                "P.PP.PPP"
                "..P....."
                "...N.P.."
                "........"
                "........"
                ".Q..pppp"
                "nk.Nnrbb")
        self.assertEqual("black win",b.winlossdraw())
        b=Board("RNBQKBNR"
                ".PPP.qPP"
                "........"
                "P...P..."
                "..b.p..."
                "........"
                "pppp.ppp"
                "rnb.k.nr")
        b.turn=b.BLACK
```

```
        self.assertEqual("white win",b.winlossdraw())
        b=Board("RNBQKBNR"
                ".PPP.PPP"
                "P......."
                "....P..q"
                "..b.p..."
                "........"
                "pppp.ppp"
                "rnb.k.nr")
        b.turn=b.BLACK
        b.move("a6","a5")
        b.move("h5","f7")
        self.assertEqual("white win",b.winlossdraw())

    def test_is_move_out_of_check(self):
        b=Board("...K...."
                "........"
                "........"
                "...QQ..."
                "........"
                "........"
                "...r...."
                "...k....")
        self.assertRaises(InvalidMoveError,b.move,"d1","e1")
        self.assertRaises(InvalidMoveError,b.move,"d2","e2")

    def test_valid_move_src_dst(self):
        b=Board("K......."
                "........"
                "........"
                "........"
                "....p..."
                "........"
                "........"
                "k.......")
        self.assertEqual([('e4', 'e5'),('a1', 'b1'), ('a1', 'a2'), ('a1',
'b2')],b.valid_move_src_dst(b.turn))
        self.assertEqual([('a8', 'b8'), ('a8', 'a7'), ('a8',
'b7')],b.valid_move_src_dst(1-b.turn))

    def test_fen_to_board(self):
        b=Board()
        self.assertEqual("rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR
w",b.board_to_fen())
        b=Board("RNB.KBNR"
                "PPPP.PPP"
                "....P..."
```

```
                "........"
                "......pQ"
                ".....p.."
                "pppp..p"
                "rnbqkbnr")
        b.turn=b.BLACK
        self.assertEqual("rnb1kbnr/pppp1ppp/4p3/8/6Pq/5P2/PPPPP2P/RNBQKBNR
b",b.board_to_fen())

if __name__ == '__main__':
    unittest.main()
```

Tests/test_file

```
from AI import fen_to_tensor

#print(fen_to_tensor.fen_to_tensor("rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBN
R w KQkq - 0 1"))

import numpy as np
import tensorflow as tf
from chess.board import *
import os

os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3' # suppress everything except errors

b=Board("RNBQKBNR"
        "PPPPPPPP"
        "........"
        "........"
        "........"
        "........"
        "pppppppp"
        "rnbqkbnr")
model=tf.keras.models.load_model("AI/chess5M.keras")
def eval(b):
    fen=b.board_to_fen()
    #print(fen)
    input_tensor=fen_to_tensor.fen_to_tensor(fen)
    #print(input_tensor)
    input_tensor=np.expand_dims(input_tensor, axis=0) # model expects batch
    eval=model.predict(input_tensor,verbose=0)
    print(eval[0][0])
    return eval[0][0]
eval(b)
b=Board("....K..."
        "....Q..."
```

```
        "........"
        "........"
        "........"
        "........"
        "pppppppp"
        "rnbqkbnr")
eval(b)
b=Board("RNBQKBNR"
        "PPPPPPPP"
        "........"
        "........"
        "........"
        "........"
        "pppppppp"
        "....k...")
eval(b)
b=Board("RNBQKBNR"
        "PPPPPPPP"
        "........"
        "........"
        "........"
        "........"
        "pppp.ppp"
        "....k...")
eval(b)
b=Board("RNBQKBNR"
        ".PPPPPPP"
        "........"
        "P......."
        "....p..."
        "........"
        "pppp.ppp"
        "rnbqkbnr")
eval(b)
b=Board("RNBQKBNR"
        "PPPP.PPP"
        "........"
        "....p..."
        "....p..."
        "........"
        "pppp.ppp"
        "rnbqkbnr")
eval(b)
```

Tests/test_minimax

```
"""
Minimax unit test
"""

from chess.minimax import *
from chess import board
import unittest

class test_minimax(unittest.TestCase):
    def test_init(self):
        pass

    def test_best_move_white(self):
        b=board.Board("k......K"
                "........"
                "........"
                "........"
                ".p..p..."
                "........"
                "...P...."
                "..p.....")
        # m=Minimax(b,1)
        m=MinimaxAI(b,1)

        self.assertEqual(("c1","d2"),m.best_move())
        b=board.Board("k......K"
                "........"
                "........"
                "...P...."
                "....n..."
                "P......."
                ".p......"
                "........")
        #move differs depending on evaluation
        # m=Minimax(b,2)
        # self.assertEqual(('e4', 'g5'),m.best_move())

        m=MinimaxAI(b,2)
        # depth 2 works, sees its about to be taken
        self.assertEqual(('e4', 'f6'),m.best_move())
        b=board.Board(".......K"
                "........"
                "....r..."
                "...r...."
                "........"
                "........"
```

```
                ........"
                ...k....")
        # m=Minimax(b,3)
        m=MinimaxAI(b,3)
        #mate in 2
        self.assertEqual(("e6","e7"),m.best_move())

    def test_best_move_black(self):
        b=board.Board("k......K"
                ........"
                ........"
                ........"
                .p..p..."
                ........"
                ...P...."
                ..p....."
        b.turn=b.BLACK

        #move differs depending on evaluation
        # m=Minimax(b,1)
        # self.assertEqual(('d2', 'c1'),m.best_move())

        m=MinimaxAI(b,1)
        self.assertEqual(('d2', 'd1'),m.best_move())

        b=board.Board("k......K"
                ........"
                .P......"
                p......."
                ....N..."
                ...p...."
                ........"
                ........")
        b.turn=b.BLACK
        #move differs depending on evaluation
        # m=Minimax(b,2)
        # self.assertEqual(('e4', 'c3'),m.best_move())

        m=MinimaxAI(b,2)
        # depth 2 works, sees its about to be taken
        self.assertEqual(('b6', 'a5'),m.best_move())
        b=board.Board(".......K"
                ........"
                ........"
                ........"
                ........"
                ....RR.."
```

```
                "........"
                "k......")
        b.turn=b.BLACK
        # m=Minimax(b,3)
        m=MinimaxAI(b,3)

        #mate in 2
        self.assertEqual(('e3', 'e2'),m.best_move())


if __name__ == '__main__':
    unittest.main()
```

Tests/test_model

```
"""
Test that the model evaluates to something like the training data
"""

import csv
from AI.fen_to_tensor import fen_to_tensor
import tensorflow as tf
import numpy as np

# import data from csv
# example "7r/1p3k2/p1bPR3/5p2/2B2P1p/8/PP4P1/3K4 b - -",58
dataset="AI/lichess_db_eval.csv"

def load_csv(data):
    """
    generator for reading lines out of large csv
    """
    with open(data,"r") as file:
        reader=csv.reader(file)
        for row in reader:
            board_fen,eval=row[0],row[1]
            yield (board_fen,eval)


def main():
    """
    Test the model against the test data
    """
    model=tf.keras.models.load_model("AI/chess5M.keras")

    size = 100
    i=0
```

```
    for board_fen,eval in load_csv(dataset):
        board_tensor=fen_to_tensor(board_fen)
        expected_eval=float(eval)/100 #convert from centipawns to pawns

        input_tensor=np.expand_dims(board_tensor, axis=0) # model expects
batch
        eval = model.predict(input_tensor,verbose=0)[0][0]
        print(f"{eval:6.3f} {expected_eval:6.3f} {board_fen}")
        i+=1
        if i==size:
            break

if __name__ == "__main__":
    main()
```

Tests/test_SimpleEvaluationMixin

```
"""
SimpleEvaluationMixin unit test
"""

from chess.SimpleEvaluationMixin import *
from chess import board
import unittest

class test_SimpleEvaluationMixin(unittest.TestCase):
    def test_init(self):
        pass

    def test_evaluate(self):
        e=SimpleEvaluationMixin()
        b=board.Board("k......K"
                "........"
                "........"
                "........"
                "........"
                "........"
                "........"
                "........")
        self.assertEqual(0,e.evaluate(b))
        b=board.Board("....r..K"
                "....r..."
                "........"
                "........"
                "........"
                "........"
```

```
                "........"
                "k......")
        b.turn=b.BLACK
        self.assertEqual(1000,e.evaluate(b))
        b=board.Board(".......K"
                "........"
                "........"
                "........"
                "........"
                "........"
                "..R....."
                "k.R.....")
        self.assertEqual(-1000,e.evaluate(b))
        b=board.Board("........"
                "...R...."
                ".R......"
                "....k..."
                "......R."
                ".....R.."
                "........"
                "K......")
        self.assertEqual(0,e.evaluate(b))
        b=board.Board("RNBQKBNR"
                "PPPPPPPP"
                "........"
                "....R..."
                "........"
                "........"
                "pppp.ppp"
                "rnbqkbnr")
        self.assertEqual(-6,e.evaluate(b))

if __name__ == '__main__':
    unittest.main()
```

Tests/test_UI

```
"""
UI unit test
"""

from chess.UI import *
import unittest

class test_UI(unittest.TestCase):
    def test_init(self):
        pass
```

```
    def test_player_input(self):
        u=UI()
        def input(prompt):
            return "a1"
        self.assertEqual(("a1","a1"),u.player_input(input=input))

if __name__ == '__main__':
    unittest.main()
```

References

318chess. (n.d.). *Elo Probability Table*. Retrieved from https://www.318chess.com/elo.html

Bronosky, R. (n.d.). *pep8_cheatsheet*. Retrieved from GitHub Gist: https://gist.github.com/RichardBronosky/454964087739a449da04

Chess Programming Wiki. (n.d.). *Center Distance*. Retrieved from Chess Programming Wiki: https://www.chessprogramming.org/Center_Distance

Chess Programming Wiki. (n.d.). *Evaluation*. Retrieved from Chess Programming Wiki: https://www.chessprogramming.org/Evaluation

Desmos. (n.d.). *Desmos Graphing Calculator*. Retrieved from Desmos: https://www.desmos.com/calculator

Dirk Hoekstra, P. M. (n.d.). *Chess AI*. Retrieved from github: https://github.com/Dirk94/ChessAI

Minimax. (n.d.). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Minimax

PEP 8. (n.d.). Retrieved from https://peps.python.org/pep-0008/

Why Test Driven Development. (n.d.). Retrieved from Marsner: https://marsner.com/blog/why-test-driven-development-tdd/

Wikipedia. (n.d.). *Algebraic Notation*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Algebraic_notation_(chess)

Wikipedia. (n.d.). *Alpha-Beta Pruning*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning#:~:text=Alpha%E2%80%93beta%20pruning%20is%20a,Connect%204%2C%20etc.).

Wikipedia. (n.d.). *Fischer Random Chess*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Fischer_random_chess

Wikipedia. (n.d.). *Huber Loss*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Huber_loss

Wikipedia. (n.d.). *Minimax*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Minimax

Wikipedia. (n.d.). *Portable Game Notation*. Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Portable_Game_Notation