

Table of Contents

[Analysis 4](#_Toc194315910)

[Research 4](#_Toc194315911)

[Interviews 5](#_Toc194315912)

[Objectives 6](#_Toc194315913)

[My view on the project 7](#_Toc194315914)

[Design and Technical solution 9](#_Toc194315915)

[Class diagram 9](#_Toc194315916)

[Program Flow 11](#_Toc194315917)

[UI, UX, and Human computer interaction 13](#_Toc194315918)

[Code, Class and object tables 15](#_Toc194315919)

[Main.py 15](#_Toc194315920)

[Game.py 22](#_Toc194315921)

[Board.py 27](#_Toc194315922)

[Square.py 44](#_Toc194315923)

[Piece.py 45](#_Toc194315924)

[Dragger.py 48](#_Toc194315925)

[AI\_player.py 50](#_Toc194315926)

[Move.py 54](#_Toc194315927)

[Helper files 56](#_Toc194315928)

[TensorFlow 60](#_Toc194315929)

[Training 66](#_Toc194315930)

[System requirements 70](#_Toc194315931)

[Call diagram 70](#_Toc194315932)

[Testing 72](#_Toc194315933)

[Objective 3 Piece Logic, motion and SFX 72](#_Toc194315934)

[Objectives 4 + 1: Neural Network creation and training 76](#_Toc194315935)

[Objectives 5 FEN input 84](#_Toc194315936)

[Objectives 6 UI and UX 85](#_Toc194315937)

[Evaluation 86](#_Toc194315938)

[end user feedback 89](#_Toc194315939)

[Positives 89](#_Toc194315940)

[Improvements: 90](#_Toc194315941)

[Overall 90](#_Toc194315942)

[Bibliography 91](#_Toc194315943)

# Analysis

For my NEA I will create an artificial intelligence to play the game of chess to an Elo of at least 1500. Any achievements above 2500 will be an outstanding for me and my stakeholders. I am building this project for a friend who needs a digital opponent who can adapt to her style of play in a way a human would. I feel that this project is of a suitable difficulty for me as it will challenge me to work independently and search for creative solutions to problems that arise. I will be able to make use of complex concepts such as matrices and JSON files, multiprocessing and threading in order to train a deep learning neural network that will play against the player.

To make the project easier in the long run I will use an Object-Oriented language / framework as the basis to my project. This will allow the project to have more synergy and makes debugging easier as the code will be split into separate files in which only relevant code will inhabit. The languages that could be used for a project like this are Python, Java, SwiftUI, C# / ++ as they either have simple ways to make a readable user interface or strong machine learning library that are industry standards. I have the most knowledge in python however if I find that another language is a better suit then the project will be coded in that language.

## Research

The most well know solution is [chess.com](https://www.chess.com/). a website with differing levels of opponents, online PvP, challenges and UI customisation. Despite this readily available resource, my friend has stated that it needs to run locally on her laptop and not require an internet connection. The UI must be simple, and the program should run a game immediately upon startup.

For local running stockfish Is available however this is too resource intensive for her laptop requiring more than 8GB of RAM to run locally. She has said this is a step in the right direction bus still not what she requires.

Her laptop has a chess game built into the operating system however she has stated that the user experience isn’t pleasing, and the game can run slowly on occasion and requires the device to be internet connected to use certain features such as more advanced opponents. She would like the game to be simple to use with a clear UI and be entirely offline.

The papers that I have read to research around machine learning, Artificial intelligence, neural networks and Deep learning are detailed in the bibliography. They provide context to how algorithms similar to mine such as stockfish and AlphaGo were built. They will form the foundation of my knowledge for AI / ML / DLNN during this project.

## Interviews

To fully understand what my client wanted I performed an interview with her so that I could grasp to what extent she requires from this project:

Q - What ability are you at chess?

A – “About 1800 to 1900 ELO”

Q – “What ability should the Opponent Play at roughly ”

A – “An opponent would be nice and should be around 2000 Elo however this is not a priority. It should be similar to my ability ”

Q – “What is a priority for you in this game”

A – “the game should not need an internet connection to run, and the UI should be easy to navigate”

Q – “Are different themes a priority for you?”

A – “ Yes, the game should have four ideally, but more would not be a miss”

Overall, the program should be fully offline game of chess with an algorithm that is of a 1500 Elo minimum, have variable themes that can be toggled by a key press and a simple and intuitive UI.

## Objectives

1. Learn how to code and deploy a deep learning Neural Network (NN).
   1. Are there different varieties or options for ML algorithms
   2. Understand how the activation functions of a network work
      1. What is an activation function
      2. How do they apply to a NN
      3. What is the math behind these functions
      4. Which is most applicable for my scenario
   3. TensorFlow Vs Pytorch
      1. What are they and their applications
      2. Which one is better suited for my project and uses
2. Discover the best language to code it in as well as frameworks .
   1. Languages could include Python, C, C++, Java
   2. Learn how to use the relevant libraries for the language.
   3. Is an API necessary for processing the data
3. Code a playable game of chess in which no errors occur that affect the gameplay.
   1. The game must allow all legal moves and en passent and castling.
   2. Different themes / colours for the board
   3. Sound effects to play on moves and capture of piece
   4. The game must show the available moves for a selected piece in a different colour e.g. red
      1. It must cope with any pieces that may be in the path of that piece and adjust its output path accordingly
   5. Pieces must only move in ways that are legal and cannot be placed anywhere that would violate chess logic
   6. Pieces work in tandem with one another and moving one will not break the logic of anther
4. Code a neural network that takes in an input and gives out the predictions for the best move
   1. Does it process the data prior to parsing it through the NN for efficiency
   2. Can it display the shape of the input data
   3. Can It load an existing model to make it better
      1. When loading a saved model does it (the saved model) initialise properly. Are all tensors loaded correctly and engaged before data is parsed through
   4. Does it make use of complex data structures like matrices or bit boards
   5. Can it be trained to become smarter
   6. does the model recognise errors in the layer formatting and can it correct them on the fly
   7. can it make predictions for the next best move
   8. can the prediction translate into a move on the board
5. Allow the game to accept industry standard game setup codes.
   1. Understand the industry codes for generating chess games (FEN).
      1. How do I integrate this into my game
   2. Have the game output a useable FEN string.
   3. Have the game accept a FEN string.
6. The user interface should be intuitive and easy to navigate
   1. Alternatively, it can boot straight into the game negating the need for a UI to navigate through
   2. The difference between efficient UI and an easy and intuitive UX
7. Understand how to create / maintain a repository through GitHub.
   1. understand common terminology for git.
   2. link my IDE to GitHub so that my friend can access the project as its being developed to give feedback if she wants to
   3. create and maintain a .md file containing my project details

## My view on the project

This project and objectives should challenge me to produce a high standard and quality of work in order. The project is different to anything I have attempted before but will allow me to pull on my current knowledge and experience whilst expanding it further in order to learn and develop as a person.

All changes and the full project are available in this GitHub repository:

https://github.com/English-Garfield/CompSciNEA\_IK/

My friend wants a Chess AI that she can play against. It should have and adjustable Elo along with being able to help her make the best next move should she want it to help her progress as a player. It also must have a user interface that is easy to navigate.

# Design and Technical solution

The program makes use of an OOP structure where different files hold associated classes that then get imported into the main loop and or other files when needed. This will keep my project organised and structured which will help me when debugging or making alterations. It will allow for a more efficient program as files are loaded as and when they are needed

The first problem is what language do I use to code this project as all would lend themselves to this project, Java is predominantly used for game development due to the Item oriented format that allows code sections to be reused. It will run on any machine, but a JVM (Java Virtual Machine) is recommended for execution and can make use of multithreaded execution which would benefit a Deep Learning Neural Network. However, Java is slow to execute, and UI looks old and clunky when compared to modern languages like JavaScript and Angular.JS

C++ is also used for game development due to its OOP structure and management of low-level functions allowing for efficient memory performance and mature collection of external libraries. However, a problem would be when the info points to an equivalent thing from two separate starting points. This will happen in my code so would need to be addressed from the start

Python has a large ecosystem of third-party modules and libraries such as PyGame and TensorFlow that will be vital to the project as they will make up its core functions. It is easy to learn is my primary language when coding and managing data structures. It is high level and abstracts lower-level detail but allows for data types to be dynamically typed and not explicitly declared as a type also allowing for support of OOP or procedural code. However, it is interpreted not compiled so can be slow to execute and has a GIL (Global Interpreter Lock). This is a mechanism in python that prevents multithreaded execution and parallelism and concurrency of certain applications which will affect the performance of my program but can be worked around

## Class diagram

Below is the structure I feel will allow the program to interface effectively with other aspects of the program. It is purposefully simple as I don’t know how each class will be constructed.

Figure 1 - System flowchart


## Program Flow

The program uses an object-oriented style as this allows for the program to interact with any element of itself easily. expanding on the Figure 1 diagram Figure 2 includes methods and how each class is called by other methods and objects. It is a more detailed design of the program and includes what I believe will make up the program.

It will act as a point of reference in case the classes don’t interface correctly, or I forget how they are constructed

A diagram of a computer

AI-generated content may be incorrect.

## UI, UX, and Human computer interaction

A screenshot of a chess game

AI-generated content may be incorrect.A screenshot of a game

AI-generated content may be incorrect.A screenshot of a computer screen

AI-generated content may be incorrect.A screenshot of a game

AI-generated content may be incorrect.A screenshot of a game

AI-generated content may be incorrect.A good user interface is important to the usability of the program. It must be simple to navigate and shouldn’t be resource intensive to process as the system resources will be needed for other calculations. The UI will be 800 x 800 pixels and look similar to the image to the left. The basic design makes it simple to render and push to the screen and can be drawn using in built PyGame functions such as .get\_rect(). The next screen displayed after pressing start will be the game itself a standard chess board and pieces with the player being white by default and the computer playing black. If the “Start” button is pressed, then the program will load the game loop or will exit out if “Quit” is pressed closing the UI and stopping the program.

The game state will resemble a chess board of 8x8 with four themes. Green, Blue, Brown and Grey the variety in colours will give her the ability to choose how it looks making it easier on her eyes. Each square will be outlined when hovered over to show where on the screen the cursor and what square is selected.

A screenshot of a game

AI-generated content may be incorrect.

In this image we can see both possible moves for the selected pawn and the highlighted area of the selected square. These features will allow a player to visualise next moves and how that affect the game state without making the move.

## Code, Class and object tables

My solution to the project is bellow which includes all classes used in the project and an explanation of how they work

### Main.py

This holds the main loop for the game and is the only thing to run continuously in the entire program.

|  |  |
| --- | --- |
| import pygame import sys from const import \* from game import Game from square import Square from move import Move from piece import Piece | These are the libraries needed for the object to function |
| # Initialize Pygame pygame.init() # Set up display screen = pygame.display.set\_mode((WIDTH, HEIGHT)) pygame.display.set\_caption("Main Screen")  # Button positions start\_button\_rect = pygame.Rect((WIDTH // 2 - BUTTON\_WIDTH // 2, 200), (BUTTON\_WIDTH, BUTTON\_HEIGHT)) quit\_button\_rect = pygame.Rect((WIDTH // 2 - BUTTON\_WIDTH // 2, 350), (BUTTON\_WIDTH, BUTTON\_HEIGHT)) | This initialises pygame and sets up the display and buttons |
| def \_\_init\_\_(self):  self.checkmate\_displayed = None  pygame.init()  self.screen = pygame.display.set\_mode((WIDTH, HEIGHT))  pygame.display.set\_caption('Chess')  self.game = Game()  self.running = True  self.in\_main\_menu = True  self.ai\_moving = False | Sets up the first instance of the game state and any variables that are needed in the main loop |
| def main\_screen(self):  while self.in\_main\_menu:  self.screen.fill(YELLOW)  pygame.draw.rect(self.screen, GREEN, start\_button\_rect)  pygame.draw.rect(self.screen, SALMON, quit\_button\_rect)   start\_text = FONT.render('Start Game', True, WHITE)  quit\_text = FONT.render('Quit', True, WHITE)   start\_text\_rect = start\_text.get\_rect(center=start\_button\_rect.center)  quit\_text\_rect = quit\_text.get\_rect(center=quit\_button\_rect.center)   self.screen.blit(start\_text, start\_text\_rect)  self.screen.blit(quit\_text, quit\_text\_rect)   pygame.display.update()   for event in pygame.event.get():  if event.type == pygame.QUIT:  self.in\_main\_menu = False  self.running = False  elif event.type == pygame.MOUSEBUTTONDOWN:  if start\_button\_rect.collidepoint(event.pos):  self.in\_main\_menu = False  elif quit\_button\_rect.collidepoint(event.pos):  pygame.quit()  sys.exit() | The first thing the player interacts with. A simple and intuitive UI asking them if they want to play or quit |
| def mainloop(self):  self.main\_screen()   while self.running:  self.screen.fill(DARK\_GREEN)  self.game.show\_bg(self.screen)  self.game.show\_last\_move(self.screen)  self.game.show\_moves(self.screen)  self.game.show\_pieces(self.screen)  self.game.show\_hover(self.screen)   if self.game.dragger.dragging:  self.game.dragger.update\_blit(self.screen)   # Display checkmate message if in checkmate state  if self.game.checkmate:  self.game.show\_checkmate\_message(self.screen)   if self.game.next\_player == 'black' and not self.ai\_moving:  self.ai\_moving = True # Set flag to prevent multiple AI moves  try:  # Get AI move  ai\_move = self.game.ai\_player.get\_move(self.game.board)   if ai\_move:  print(f"AI Move: {ai\_move}")   if isinstance(ai\_move, Move):  piece = ai\_move.piece  move = ai\_move   # Validate the move data  if isinstance(piece, Piece) and isinstance(move, Move):  # Check if it's a valid move  if self.game.board.valid\_move(piece, move):  # Check if a piece is captured  captured = self.game.board.squares[move.final.row][move.final.col].has\_piece()   # Perform the move  self.game.board.move(piece, move)   # Handle en passant  self.game.board.set\_true\_en\_passant(piece)   # Play sound if a piece is captured  if captured:  self.game.play\_sound(captured)  else:  self.game.play\_sound(False) # Play move sound   # Switch to next player  self.game.next\_turn()   print(f"AI moved {piece} to {move.final.row}, {move.final.col}")   # Update the display after the AI move  self.game.show\_bg(self.screen)  self.game.show\_last\_move(self.screen)  self.game.show\_pieces(self.screen)  pygame.display.update()   # Add a small delay to ensure the display is updated  pygame.time.delay(500)  else:  print(f"Invalid move: {move}")  # Try to calculate moves for this piece again  initial\_row, initial\_col = move.initial.row, move.initial.col  self.game.board.calc\_moves(piece, initial\_row, initial\_col, bool=True)   # Try again with recalculated moves  if self.game.board.valid\_move(piece, move):  # Check if a piece is captured  captured = self.game.board.squares[move.final.row][move.final.col].has\_piece()   # Perform the move  self.game.board.move(piece, move)   # Handle en passant  self.game.board.set\_true\_en\_passant(piece)   # Play sound if a piece is captured  if captured:  self.game.play\_sound(captured)  else:  self.game.play\_sound(False) # Play move sound   # Switch to next player  self.game.next\_turn()   print(f"AI moved {piece} to {move.final.row}, {move.final.col} after recalculating moves")   # Update the display after the AI move  self.game.show\_bg(self.screen)  self.game.show\_last\_move(self.screen)  self.game.show\_pieces(self.screen)  pygame.display.update()   # Add a small delay to ensure the display is updated  pygame.time.delay(500)  else:  print(f"Move still invalid after recalculating: {move}")  self.game.next\_turn() # Skip AI's turn if move is still invalid  else:  print("Invalid piece or move object returned by AI!")  self.game.next\_turn() # Skip AI's turn if move is invalid  else:  print(f"Unexpected AI move format: {ai\_move}")  self.game.next\_turn() # Skip AI's turn if move is invalid  else:  print("AI did not return a valid move. Checkmate!")  self.game.checkmate = True # Set checkmate flag  self.game.next\_turn() # Skip AI's turn if no move is returned  except Exception as e:  print(f"Error during AI move: {e}")  # If AI fails, switch to player's turn  self.game.next\_turn()   self.ai\_moving = False # Reset flag after AI move is complete   pygame.display.update()   for event in pygame.event.get():  if event.type == pygame.QUIT:  self.running = False   elif event.type == pygame.MOUSEBUTTONDOWN:  self.game.dragger.update\_mouse(event.pos)  clicked\_row = self.game.dragger.mouseY // SQSIZE  clicked\_col = self.game.dragger.mouseX // SQSIZE  if self.game.board.squares[clicked\_row][clicked\_col].has\_piece():  piece = self.game.board.squares[clicked\_row][clicked\_col].piece  if piece.color == self.game.next\_player:  self.game.board.calc\_moves(piece, clicked\_row, clicked\_col, bool=True)  self.game.dragger.save\_initial(event.pos)  self.game.dragger.drag\_piece(piece)  self.game.show\_bg(self.screen)  self.game.show\_last\_move(self.screen)  self.game.show\_moves(self.screen)  self.game.show\_pieces(self.screen)   elif event.type == pygame.MOUSEMOTION:  motion\_row = event.pos[1] // SQSIZE  motion\_col = event.pos[0] // SQSIZE  self.game.set\_hover(motion\_row, motion\_col)  if self.game.dragger.dragging:  self.game.dragger.update\_mouse(event.pos)  self.game.show\_bg(self.screen)  self.game.show\_last\_move(self.screen)  self.game.show\_moves(self.screen)  self.game.show\_pieces(self.screen)  self.game.show\_hover(self.screen)  self.game.dragger.update\_blit(self.screen)   elif event.type == pygame.MOUSEBUTTONUP:  if self.game.dragger.dragging:  self.game.dragger.update\_mouse(event.pos)  released\_row = self.game.dragger.mouseY // SQSIZE  released\_col = self.game.dragger.mouseX // SQSIZE  initial = Square(self.game.dragger.initial\_row, self.game.dragger.initial\_col)  final = Square(released\_row, released\_col)  move = Move(initial, final, self.game.dragger.piece) # Pass the piece   if self.game.board.valid\_move(self.game.dragger.piece, move):  captured = self.game.board.squares[released\_row][released\_col].has\_piece()  self.game.board.move(self.game.dragger.piece, move)  self.game.board.set\_true\_en\_passant(self.game.dragger.piece)  self.game.play\_sound(captured)  self.game.show\_bg(self.screen)  self.game.show\_last\_move(self.screen)  self.game.show\_pieces(self.screen)  self.game.next\_turn()   self.game.dragger.undrag\_piece()   elif event.type == pygame.KEYDOWN:  if event.key == pygame.K\_t:  self.game.change\_theme()   if event.key == pygame.K\_r:  self.game.reset()  self.game = Game()  # Ensure checkmate flag is reset  self.game.checkmate = False   pygame.display.update() | **Main Loop (mainloop method)**:  **Line 63**: Calls self.main\_screen() to display the main menu screen.  **Lines 65-103**: Main game loop that runs while self.running is True.  **Lines 66-71**: Updates the game display by showing the background, last move, possible moves, pieces, and hover effects.  **Lines 73-75**: Updates the display if a piece is being dragged.  **Lines 76-101**: Handles the AI move if the next player is 'black'. It retrieves the AI move, validates it, performs the move, and updates the display.  **Lines 105-166**: Event handling loop that processes user inputs.  **Lines 106-107**: Handles the quit event.  **Lines 109-123**: Handles mouse button down events to select and drag pieces.  **Lines 124-135**: Handles mouse motion events to update hover effects and dragging.  **Lines 137-156**: Handles mouse button up events to drop pieces and perform moves.  **Lines 158-165**: Handles key down events to change the theme or reset the game. |

### Game.py

This builds the board and associate logic to play chess. It creates an 8x8 grid with letters and numbers along the sides. It also lets the ai player interface with the system.

|  |  |
| --- | --- |
| import pygame from const import \* from board import Board from dragger import Dragger from config import Config from square import Square from ai\_player import AIPlayer from resource\_path import resource\_path | These are the other objects and libraries needed for this file to function |
| def \_\_init\_\_(self):  self.next\_player = 'white'  self.hovered\_sqr = None  self.board = Board()  self.dragger = Dragger()  self.config = Config()  self.ai\_player = AIPlayer()  self.checkmate = False | The constructor method that initialises the object and other objects called from within the program |
| def show\_bg(self, surface):  theme = self.config.theme   for row in range(ROWS):  for col in range(COLS):  # color  color = theme.bg.light if (row + col) % 2 == 0 else theme.bg.dark  # rect  rect = (col \* SQSIZE, row \* SQSIZE, SQSIZE, SQSIZE)  # blit  pygame.draw.rect(surface, color, rect)   # row coordinates  if col == 0:  # color  color = theme.bg.dark if row % 2 == 0 else theme.bg.light  # label  lbl = self.config.font.render(str(ROWS - row), 1, color)  lbl\_pos = (5, 5 + row \* SQSIZE)  # blit  surface.blit(lbl, lbl\_pos)   # col coordinates  if row == 7:  # color  color = theme.bg.dark if (row + col) % 2 == 0 else theme.bg.light  # label  lbl = self.config.font.render(Square.get\_alphacol(col), 1, color)  lbl\_pos = (col \* SQSIZE + SQSIZE - 20, HEIGHT - 20)  # blit  surface.blit(lbl, lbl\_pos) | **Retrieve the theme**: Gets the current theme from the config attribute.  **Loop through each square**: Iterates over each row and column of the board.  **Determine the colour**: Chooses the colour based on the sum of the row and column indices (alternating colours).  **Draw the square**: Draws a rectangle on the surface at the appropriate position with the chosen colour.  **Draw row coordinates**: Draws the row number on the left side if the column is 0.  **Draw column coordinates**: Draws the column letter at the bottom if the row is 7. |
| def show\_pieces(self, surface):  for row in range(ROWS):  for col in range(COLS):  # piece ?  if self.board.squares[row][col].has\_piece():  piece = self.board.squares[row][col].piece   # all pieces except dragger piece  if piece is not self.dragger.piece:  piece.set\_texture(size=80)  try:  img = pygame.image.load(piece.texture)  img\_center = col \* SQSIZE + SQSIZE // 2, row \* SQSIZE + SQSIZE // 2  piece.texture\_rect = img.get\_rect(center=img\_center)  surface.blit(img, piece.texture\_rect)  except Exception as e:  print(f"Error loading piece image: {e}")  # Continue without blitting if image can't be loaded | **Iterates through each square on the board and checks if there is a piece in that square and loads the pieces image in the appropriate location if so.**  **Contains a try except to handle the misallocation of the piece image files** |
| def show\_moves(self, surface):  theme = self.config.theme   if self.dragger.dragging:  piece = self.dragger.piece   # loop all valid moves  for move in piece.moves:  # color  color = theme.moves.light if (move.final.row + move.final.col) % 2 == 0 else theme.moves.dark  # rect  rect = (move.final.col \* SQSIZE, move.final.row \* SQSIZE, SQSIZE, SQSIZE)  # blit  pygame.draw.rect(surface, color, rect) | **This is responsible for displaying the possible moves for the selected piece. It iterates over valid moves and draws a square where one would be** |
| def show\_last\_move(self, surface):  theme = self.config.theme   if self.board.last\_move:  initial = self.board.last\_move.initial  final = self.board.last\_move.final   for pos in [initial, final]:  # color  color = theme.trace.light if (pos.row + pos.col) % 2 == 0 else theme.trace.dark  # rect  rect = (pos.col \* SQSIZE, pos.row \* SQSIZE, SQSIZE, SQSIZE)  # blit  pygame.draw.rect(surface, color, rect) | **Highlights the last move on the board and draws a lighter square around the square the piece moved from and to.** |
| def show\_hover(self, surface):  if self.hovered\_sqr:  # color  color = (180, 180, 180)  # rect  rect = (self.hovered\_sqr.col \* SQSIZE, self.hovered\_sqr.row \* SQSIZE, SQSIZE, SQSIZE)  # blit  pygame.draw.rect(surface, color, rect, width=3) | **This checks if there is a hovered square. If so, it highlights it grey, calculates the position and size based on the column and row the draws a rectangle with a boarder width of 3px** |
| def next\_turn(self):  self.next\_player = 'white' if self.next\_player == 'black' else 'black'  def set\_hover(self, row, col):  self.hovered\_sqr = self.board.squares[row][col]  def change\_theme(self):  self.config.change\_theme()  def play\_sound(self, captured=False):  if captured:  self.config.capture\_sound.play()  else:  self.config.move\_sound.play()  def reset(self):  self.\_\_init\_\_() | next\_turn: Switches the player from 'white' to 'black' or vice versa.  set\_hover: Sets the square that the mouse is currently hovering over.  change\_theme: Changes the game's visual theme.  play\_sound: Plays a sound when a move is made, with a different sound if a piece is captured.  reset: Resets the game by reinitializing the Game object. |
| def show\_checkmate\_message(self, surface):  if self.checkmate:  # Create a semi-transparent overlay  overlay = pygame.Surface((WIDTH, HEIGHT), pygame.SRCALPHA)  overlay.fill((0, 0, 0, 128)) # Black with 50% transparency  surface.blit(overlay, (0, 0))   # Render checkmate message  font = pygame.font.Font(None, 60)  checkmate\_text = font.render('CHECKMATE!', True, WHITE)  restart\_text = font.render('Press R to restart', True, WHITE)   # Position the text in the center of the screen  checkmate\_rect = checkmate\_text.get\_rect(center=(WIDTH//2, HEIGHT//2 - 30))  restart\_rect = restart\_text.get\_rect(center=(WIDTH//2, HEIGHT//2 + 30))   # Draw the text on the screen  surface.blit(checkmate\_text, checkmate\_rect)  surface.blit(restart\_text, restart\_rect) | Displays a checkmate message to the screen when there are no valid moves or the ai can see no more moves from the current position |

### Board.py

Stacks / queues are not explicitly used here however the recursive nature of calculating possible moves and validating them against potential checks implies a stack like behaviour. When a moved is explored it is ‘pushed’ onto the stack. And returns only if it is a valid state

(Green highlighting indicates A group algorithms)

|  |  |  |
| --- | --- | --- |
| from const import \* from square import Square from piece import \* from move import Move from sound import Sound import copy import os from resource\_path import resource\_path | | All imports from other classes and external libraries needed for the class to function |
| def \_\_init\_\_(self):  self.squares = [[0, 0, 0, 0, 0, 0, 0, 0] for col in range(COLS)]  self.last\_move = None  self.\_create()  self.\_add\_pieces('white')  self.\_add\_pieces('black') | | Constructor class that initialises an 8 by 8 array of 0’s that represents each piece’s position, the last move, and white + black pieces |
| def move(self, piece, move, testing=False):  initial = move.initial  final = move.final   en\_passant\_empty = self.squares[final.row][final.col].isempty()   # console board move update  self.squares[initial.row][initial.col].piece = None  self.squares[final.row][final.col].piece = piece   if isinstance(piece, Pawn):  # en passant capture  diff = final.col - initial.col  if diff != 0 and en\_passant\_empty:  # console board move update  self.squares[initial.row][initial.col + diff].piece = None  self.squares[final.row][final.col].piece = piece  if not testing:  sound = Sound(  resource\_path('assets/sounds/capture.wav'))  sound.play()   # pawn promotion  else:  self.check\_promotion(piece, final)   # king castling  if isinstance(piece, King):  if self.castling(initial, final) and not testing:  diff = final.col - initial.col  rook = piece.left\_rook if (diff < 0) else piece.right\_rook  self.move(rook, rook.moves[-1])   # move  piece.moved = True   # clear valid moves  piece.clear\_moves()   # set last move  self.last\_move = move  if isinstance(piece, Pawn):  # en passant capture  diff = final.col - initial.col  if diff != 0 and en\_passant\_empty:  # console board move update  self.squares[initial.row][initial.col + diff].piece = None  self.squares[final.row][final.col].piece = piece  if not testing:  sound = Sound(  os.path.join('assets/sounds/capture.wav'))  sound.play()   # pawn promotion  else:  self.check\_promotion(piece, final)   # king castling  if isinstance(piece, King):  if self.castling(initial, final) and not testing:  diff = final.col - initial.col  rook = piece.left\_rook if (diff < 0) else piece.right\_rook  self.move(rook, rook.moves[-1])   # move  piece.moved = True   # clear valid moves  piece.clear\_moves()   # set last move  self.last\_move = move | | **Extract Initial and Final Positions**: It gets the initial and final positions from the move object.  **Check for En Passant**: It checks if the final position is empty, which is relevant for the en passant move in chess.  **Update Board**: It updates the board by removing the piece from the initial position and placing it at the final position.  **En Passant Capture**: If the move is an en passant capture, it removes the captured pawn.  **Promotion**: It checks if the pawn should be promoted to a queen.  **Castling**: It handles castling by moving the rook as well.  **Mark Piece as Moved**: It marks the piece as having moved.  **Clear Valid Moves**: It clears the list of valid moves for the piece.  **Set Last Move**: It records the move as the last move made. |
| def valid\_move(self, piece, move):  return move in piece.moves  def check\_promotion(self, piece, final):  if final.row == 0 or final.row == 7:  self.squares[final.row][final.col].piece = Queen(piece.color)  def castling(self, initial, final):  return abs(initial.col - final.col) == 2 | | **valid\_move(self, piece, move)**:  This method checks if a given move is valid for a specific piece.  It returns True if the move is in the list of valid moves for the piece, otherwise False.  **check\_promotion(self, piece, final)**:  This method checks if a pawn has reached the opposite end of the board (promotion row).  If the pawn reaches the last row (0 for black, 7 for white), it is promoted to a Queen.  **castling(self, initial, final)**:  This method checks if a move is a castling move.  It returns True if the move is a castling move (the king moves two squares horizontally), otherwise False. |
| def set\_true\_en\_passant(self, piece):   if not isinstance(piece, Pawn):  return   for row in range(ROWS):  for col in range(COLS):  if isinstance(self.squares[row][col].piece, Pawn):  self.squares[row][col].piece.en\_passant = False   piece.en\_passant = True | | This function checks if a piece is in range for en passent to be applicable to it |
| def in\_check(self, piece, move):  temp\_piece = copy.deepcopy(piece)  temp\_board = copy.deepcopy(self)  temp\_board.move(temp\_piece, move, testing=True)   for row in range(ROWS):  for col in range(COLS):  if temp\_board.squares[row][col].has\_enemy\_piece(piece.color):  p = temp\_board.squares[row][col].piece  temp\_board.calc\_moves(p, row, col, bool=False)  form in p.moves:  if isinstance(m.final.piece, King):  print("Checkmate")  return True   return False | | Checks if a move puts an enemy king in check by looking at the pieces moved through a temporary state board |
| def calc\_moves(self, piece, row, col, bool=True):  """  Calculate all the possible (valid) moves of an specific piece on a specific position  """   def pawn\_moves():  # steps  steps = 1 if piece.moved else 2   # vertical moves  start = row + piece.dir  end = row + (piece.dir \* (1 + steps))  for possible\_move\_row in range(start, end, piece.dir):  if Square.in\_range(possible\_move\_row):  if self.squares[possible\_move\_row][col].isempty():  # create initial and final move squares  initial = Square(row, col)  final = Square(possible\_move\_row, col)  # create a new move  move = Move(initial, final)   # check potential checks  if bool:  if not self.in\_check(piece, move):  # append new move  piece.add\_move(move)  else:  # append new move  piece.add\_move(move)  # blocked  else:  break  # not in range  else:  break   # diagonal moves  possible\_move\_row = row + piece.dir  possible\_move\_cols = [col - 1, col + 1]  for possible\_move\_col in possible\_move\_cols:  if Square.in\_range(possible\_move\_row, possible\_move\_col):  if self.squares[possible\_move\_row][possible\_move\_col].has\_enemy\_piece(piece.color):  # create initial and final move squares  initial = Square(row, col)  final\_piece = self.squares[possible\_move\_row][possible\_move\_col].piece  final = Square(possible\_move\_row, possible\_move\_col, final\_piece)  # create a new move  move = Move(initial, final)   # check potencial checks  if bool:  if not self.in\_check(piece, move):  # append new move  piece.add\_move(move)  else:  # append new move  piece.add\_move(move)   # en passant moves  r = 3 if piece.color == 'white' else 4  fr = 2 if piece.color == 'white' else 5  # left en pessant  if Square.in\_range(col - 1) and row == r:  if self.squares[row][col - 1].has\_enemy\_piece(piece.color):  p = self.squares[row][col - 1].piece  if isinstance(p, Pawn):  if p.en\_passant:  # create initial and final move squares  initial = Square(row, col)  final = Square(fr, col - 1, p)  # create a new move  move = Move(initial, final)   # check potencial checks  if bool:  if not self.in\_check(piece, move):  # append new move  piece.add\_move(move)  else:  # append new move  piece.add\_move(move)   # right en pessant  if Square.in\_range(col + 1) and row == r:  if self.squares[row][col + 1].has\_enemy\_piece(piece.color):  p = self.squares[row][col + 1].piece  if isinstance(p, Pawn):  if p.en\_passant:  # create initial and final move squares  initial = Square(row, col)  final = Square(fr, col + 1, p)  # create a new move  move = Move(initial, final)   # check potencial checks  if bool:  if not self.in\_check(piece, move):  # append new move  piece.add\_move(move)  else:  # append new move  piece.add\_move(move)   def knight\_moves():  # 8 possible moves  possible\_moves = [  (row - 2, col + 1),  (row - 1, col + 2),  (row + 1, col + 2),  (row + 2, col + 1),  (row + 2, col - 1),  (row + 1, col - 2),  (row - 1, col - 2),  (row - 2, col - 1),  ]   for possible\_move in possible\_moves:  possible\_move\_row, possible\_move\_col = possible\_move   if Square.in\_range(possible\_move\_row, possible\_move\_col):  if self.squares[possible\_move\_row][possible\_move\_col].isempty\_or\_enemy(piece.color):  # create squares of the new move  initial = Square(row, col)  final\_piece = self.squares[possible\_move\_row][possible\_move\_col].piece  final = Square(possible\_move\_row, possible\_move\_col, final\_piece)  # create new move  move = Move(initial, final)   # check potencial checks  if bool:  if not self.in\_check(piece, move):  # append new move  piece.add\_move(move)  else:  break  else:  # append new move  piece.add\_move(move)   def straightline\_moves(incrs):  for incr in incrs:  row\_incr, col\_incr = incr  possible\_move\_row = row + row\_incr  possible\_move\_col = col + col\_incr   while True:  if Square.in\_range(possible\_move\_row, possible\_move\_col):  # create squares of the possible new move  initial = Square(row, col)  final\_piece = self.squares[possible\_move\_row][possible\_move\_col].piece  final = Square(possible\_move\_row, possible\_move\_col, final\_piece)  # create a possible new move  move = Move(initial, final)   # empty = continue looping  if self.squares[possible\_move\_row][possible\_move\_col].isempty():  # check potencial checks  if bool:  if not self.in\_check(piece, move):  # append new move  piece.add\_move(move)  else:  # append new move  piece.add\_move(move)   # has enemy piece = add move + break  elif self.squares[possible\_move\_row][possible\_move\_col].has\_enemy\_piece(piece.color):  # check potencial checks  if bool:  if not self.in\_check(piece, move):  # append new move  piece.add\_move(move)  else:  # append new move  piece.add\_move(move)  break   # has team piece = break  elif self.squares[possible\_move\_row][possible\_move\_col].has\_team\_piece(piece.color):  break   # not in range  else:  break   # incrementing incrs  possible\_move\_row = possible\_move\_row + row\_incr  possible\_move\_col = possible\_move\_col + col\_incr   def king\_moves():  adjs = [  (row - 1, col + 0), # up  (row - 1, col + 1), # up-right  (row + 0, col + 1), # right  (row + 1, col + 1), # down-right  (row + 1, col + 0), # down  (row + 1, col - 1), # down-left  (row + 0, col - 1), # left  (row - 1, col - 1), # up-left  ]   # normal moves  for possible\_move in adjs:  possible\_move\_row, possible\_move\_col = possible\_move   if Square.in\_range(possible\_move\_row, possible\_move\_col):  if self.squares[possible\_move\_row][possible\_move\_col].isempty\_or\_enemy(piece.color):  # create squares of the new move  initial = Square(row, col)  final = Square(possible\_move\_row, possible\_move\_col) # piece=piece  # create new move  move = Move(initial, final)  # check potencial checks  if bool:  if not self.in\_check(piece, move):  # append new move  piece.add\_move(move)  else:  break  else:  # append new move  piece.add\_move(move)   # castling moves  if not piece.moved:  # queen castling  left\_rook = self.squares[row][0].piece  if isinstance(left\_rook, Rook):  if not left\_rook.moved:  for c in range(1, 4):  # castling is not possible because there are pieces in between ?  if self.squares[row][c].has\_piece():  break   if c == 3:  # adds left rook to king  piece.left\_rook = left\_rook   # rook move  initial = Square(row, 0)  final = Square(row, 3)  moveR = Move(initial, final)   # king move  initial = Square(row, col)  final = Square(row, 2)  moveK = Move(initial, final)   # check potencial checks  if bool:  if not self.in\_check(piece, moveK) and not self.in\_check(left\_rook, moveR):  # append new move to rook  left\_rook.add\_move(moveR)  # append new move to king  piece.add\_move(moveK)  else:  # append new move to rook  left\_rook.add\_move(moveR)  # append new move king  piece.add\_move(moveK)   # king castling  right\_rook = self.squares[row][7].piece  if isinstance(right\_rook, Rook):  if not right\_rook.moved:  for c in range(5, 7):  # castling is not possible because there are pieces in between ?  if self.squares[row][c].has\_piece():  break   if c == 6:  # adds right rook to king  piece.right\_rook = right\_rook   # rook move  initial = Square(row, 7)  final = Square(row, 5)  moveR = Move(initial, final)   # king move  initial = Square(row, col)  final = Square(row, 6)  moveK = Move(initial, final)   # check potencial checks  if bool:  if not self.in\_check(piece, moveK) and not self.in\_check(right\_rook, moveR):  # append new move to rook  right\_rook.add\_move(moveR)  # append new move to king  piece.add\_move(moveK)  else:  # append new move to rook  right\_rook.add\_move(moveR)  # append new move king  piece.add\_move(moveK)   if isinstance(piece, Pawn):  pawn\_moves()   elif isinstance(piece, Knight):  knight\_moves()   elif isinstance(piece, Bishop):  straightline\_moves([  (-1, 1), # up-right  (-1, -1), # up-left  (1, 1), # down-right  (1, -1), # down-left  ])   elif isinstance(piece, Rook):  straightline\_moves([  (-1, 0), # up  (0, 1), # right  (1, 0), # down  (0, -1), # left  ])   elif isinstance(piece, Queen):  straightline\_moves([  (-1, 1), # up-right  (-1, -1), # up-left  (1, 1), # down-right  (1, -1), # down-left  (-1, 0), # up  (0, 1), # right  (1, 0), # down  (0, -1) # left  ])   elif isinstance(piece, King):  king\_moves() | | **calc\_moves Method**:  Tree/graph traversal  This method calculates all possible valid moves for a given piece at a specific position (row, col). It uses helper functions to handle the move calculations for different types of pieces (Pawn, Knight, Bishop, Rook, Queen, King).  **Helper Functions**:  **pawn\_moves**: Calculates vertical, diagonal, and en passant moves for pawns.  **knight\_moves**: Calculates the 8 possible L-shaped moves for knights.  **straightline\_moves**: Calculates moves for pieces that move in straight lines (Bishops, Rooks, Queens).  **king\_moves**: Calculates adjacent moves and castling moves for kings.  **Move Validation**:  Each helper function creates Move objects representing potential moves. The in\_check method is used to ensure that moves do not place the player's king in check. Valid moves are added to the piece's list of possible moves using the add\_move method.  **Piece-Specific Logic**:  **Pawns**: Handle initial double-step moves, diagonal captures, and en passant.  **Knights**: Handle L-shaped moves.  **Bishops, Rooks, Queens**: Use straightline\_moves to handle their respective movement patterns.  **Kings**: Handle single step moves in all directions and castling.  **Castling.** Special logic for castling moves is included in the king\_moves function, checking if the rooks and king have moved and if the path is clear.  Recursion is used on a temporary board state to check if a move result in check resulting in a recursive loop |
| def \_create(self):  for row in range(ROWS):  for col in range(COLS):  self.squares[row][col] = Square(row, col) | | The code defines the \_create method of the Board class. This method initializes the squares attribute of the Board object by creating an 8x8 grid of square objects. Each Square object is initialized with its row and column position. |
| def \_add\_pieces(self, color):  row\_pawn, row\_other = (6, 7) if color == 'white' else (1, 0)   # pawns  for col in range(COLS):  self.squares[row\_pawn][col] = Square(row\_pawn, col, Pawn(color))   # knights  self.squares[row\_other][1] = Square(row\_other, 1, Knight(color))  self.squares[row\_other][6] = Square(row\_other, 6, Knight(color))   # bishops  self.squares[row\_other][2] = Square(row\_other, 2, Bishop(color))  self.squares[row\_other][5] = Square(row\_other, 5, Bishop(color))   # rooks  self.squares[row\_other][0] = Square(row\_other, 0, Rook(color))  self.squares[row\_other][7] = Square(row\_other, 7, Rook(color))   # queen  self.squares[row\_other][3] = Square(row\_other, 3, Queen(color))   # king  self.squares[row\_other][4] = Square(row\_other, 4, King(color)) | **Determine Rows for Pieces.** row\_pawn, row\_other = (6, 7) if color == 'white' else (1, 0)  This sets the rows where pawns and other pieces will be placed based on the color.  **Add Pawns**:  for col in range(COLS):  self.squares[row\_pawn][col] = Square(row\_pawn, col, Pawn(color))  This loop places a pawn in each column of the specified row.  **Knights**:  self.squares[row\_other][1] = Square(row\_other, 1, Knight(color))  self.squares[row\_other][6] = Square(row\_other, 6, Knight(color))  These lines place knights in columns 1 and 6.  **Add Bishops**:  self.squares[row\_other][2] = Square(row\_other, 2, Bishop(color))  self.squares[row\_other][5] = Square(row\_other, 5, Bishop(color))  These lines place bishops in columns 2 and 5.  **Add Rooks**:  self.squares[row\_other][0] = Square(row\_other, 0, Rook(color))  self.squares[row\_other][7] = Square(row\_other, 7, Rook(color))  These lines place rooks in columns 0 and 7.  **Add Queen**:  self.squares[row\_other][3] = Square(row\_other, 3, Queen(color))  This line places the queen in column 3.  **Add King**:  self.squares[row\_other][4] = Square(row\_other, 4, King(color))  This line places the king in column 4.  All this is done dynamically based of the initial board set up | |
| def get\_king(self, color):  for row in range(ROWS):  for col in range(COLS):  piece = self.squares[row][col].piece  if isinstance(piece, King) and piece.color == color:  return piece  return None | | The get\_king method in the Board class searches for and returns the King piece of a specified color on the board. If no King of the specified color is found, it returns None. |

### Square.py

|  |  |
| --- | --- |
| class Square:   ALPHACOLS = {0: 'a', 1: 'b', 2: 'c', 3: 'd', 4: 'e', 5: 'f', 6: 'g', 7: 'h'}   def \_\_init\_\_(self, row, col, piece=None):  self.row = row  self.col = col  self.piece = piece  self.alphacol = self.ALPHACOLS[col]   def \_\_hash\_\_(self):  # Create a unique hash based on the row and column  return hash((self.row, self.col))   def \_\_eq\_\_(self, other):  return self.row == other.row and self.col == other.col   def has\_piece(self):  return self.piece is not None   def isempty(self):  return not self.has\_piece()   def has\_team\_piece(self, color):  return self.has\_piece() and self.piece.color == color   def has\_enemy\_piece(self, color):  return self.has\_piece() and self.piece.color != color   def isempty\_or\_enemy(self, color):  return self.isempty() or self.has\_enemy\_piece(color)   @staticmethod  def in\_range(\*args):  for arg in args:  if arg < 0 or arg > 7:  return False    return True   @staticmethod  def get\_alphacol(col):  ALPHACOLS = {0: 'a', 1: 'b', 2: 'c', 3: 'd', 4: 'e', 5: 'f', 6: 'g', 7: 'h'}  return ALPHACOLS[col] | The Square class represents a square on a chessboard. Here is a brief explanation of its components:   * **Class Variables:**   + ALPHACOLS: A dictionary mapping column index (0-7) to their corresponding alphabetical representations ('a'-'h'). * **Instance Variables:**   + row: The row index of the square.   + col: The column index of the square.   + piece: The piece currently on the square (if any).   + alphacol: The alphabetical representation of the column, derived from ALPHACOLS. * **Methods:**   + \_\_init\_\_: Initializes a square with a row, column, and optionally a piece.   + \_\_hash\_\_: Returns a unique hash based on the row and column.   + \_\_eq\_\_: Checks equality based on row and column.   + has\_piece: Returns True if there is a piece on the square.   + isempty: Returns True if the square is empty.   + has\_team\_piece: Checks if the square has a piece of the given color.   + has\_enemy\_piece: Checks if the square has an enemy piece of the given color.   + isempty\_or\_enemy: Checks if the square is empty or has an enemy piece.   + in\_range: Static method to check if given arguments are within the valid range (0-7).   + get\_alphacol: Static method to get the alphabetical column representation for a given column index |

### Piece.py

|  |  |  |
| --- | --- | --- |
| import os | This file just needs to control the file paths of certain assets so only needs the os module | |
| def \_\_init\_\_(self, name, color, value, texture=None, texture\_rect=None):  self.name = name  self.color = color  value\_sign = 1 if color == 'white' else -1  self.value = value \* value\_sign  self.moves = []  self.moved = False  self.texture = texture  self.set\_texture()  self.texture\_rect = texture\_rect | Constructor method of this class that establish the initial settings for each variable | |
| def set\_texture(self, size=80):  self.texture = os.path.join(  f'assets/images/imgs-{size}px/{self.color}\_{self.name}.png') | Establishes the location and size of the piece images in the projects file structure | |
| def add\_move(self, move):  self.moves.append(move)  def clear\_moves(self):  self.moves = [] | Adds and removes moves from the self.moves array | |
| class Pawn(Piece):   def \_\_init\_\_(self, color):  self.dir = -1 if color == 'white' else 1  self.en\_passant = False  super().\_\_init\_\_('pawn', color, 1.0)   class Knight(Piece):   def \_\_init\_\_(self, color):  super().\_\_init\_\_('knight', color, 3.0)   class Bishop(Piece):   def \_\_init\_\_(self, color):  super().\_\_init\_\_('bishop', color, 3.001)   class Rook(Piece):   def \_\_init\_\_(self, color):  super().\_\_init\_\_('rook', color, 5.0)   class Queen(Piece):   def \_\_init\_\_(self, color):  super().\_\_init\_\_('queen', color, 9.0)   class King(Piece):   def \_\_init\_\_(self, color):  self.left\_rook = None  self.right\_rook = None  super().\_\_init\_\_('king', color, 10000.0) | | This code defines several classes that represent different types of chess pieces. Each class inherits from the Piece class and initializes the specific attributes for that piece. Here is a brief explanation of each class:  **Pawn**:  Initializes the direction (dir) based on the color.  Sets the en\_passant attribute to False.  Calls the Piece constructor with the name 'pawn', the color, and a value of 1.0.  **Knight**:  Calls the Piece constructor with the name 'knight', the color, and a value of 3.0.  **Bishop**:  Calls the Piece constructor with the name 'bishop', the color, and a value of 3.001.  **Rook**:  Calls the Piece constructor with the name 'rook', the color, and a value of 5.0.  **Queen**:  Calls the Piece constructor with the name 'queen', the color, and a value of 9.0.  **King**:  Initializes the left\_rook and right\_rook attributes to None.  Calls the Piece constructor with the name 'king', the color, and a value of 10000.0. |

### Dragger.py

|  |  |
| --- | --- |
| import pygame  from const import \* | This class requires the PyGame module and access to the constant variables of the project |
| def \_\_init\_\_(self):  self.piece = None  self.dragging = False  self.mouseX = 0  self.mouseY = 0  self.initial\_row = 0  self.initial\_col = 0 | Constructor method of the class that sets the state of the variables before use |
| def update\_blit(self, surface):  # texture  self.piece.set\_texture(size=128)  texture = self.piece.texture  try:  # img  img = pygame.image.load(texture)  # rect  img\_center = (self.mouseX, self.mouseY)  self.piece.texture\_rect = img.get\_rect(center=img\_center)  # blit  surface.blit(img, self.piece.texture\_rect)  except Exception as e:  print(f"Error loading image in dragger: {e}")  # Continue without blitting if image can't be loaded  # other methods | Shows the piece moving as it is dragged by the curser  Also accounts for the piece files not being locatable |
| def update\_mouse(self, pos):  self.mouseX, self.mouseY = pos # (xcor, ycor) | Updates the mouse position in relation to the game window |
| def save\_initial(self, pos):  self.initial\_row = pos[1] // SQSIZE  self.initial\_col = pos[0] // SQSIZE | The code defines the save\_initial method of the Dragger class. This method takes a position (pos) as an argument and calculates the initial row and column based on the given position. The position is expected to be a tuple containing the x and y coordinates. The method uses integer division to convert the pixel coordinates to grid coordinates by dividing by SQSIZE, which is presumably a constant representing the size of each square in the grid. |
| def drag\_piece(self, piece):  self.piece = piece  self.dragging = True | Tells the program when a piece is in motion |
| def undrag\_piece(self):  self.piece = None  self.dragging = False | Reverts the above function back to the normal state |

### AI\_player.py

|  |  |
| --- | --- |
| import numpy as np import os import random from resource\_path import resource\_path  # Try different import strategies try:  import keras  KERAS\_AVAILABLE = True except ImportError:  KERAS\_AVAILABLE = False  print("Warning: Keras not available, will use random moves")  try:  import tensorflow as tf  TF\_AVAILABLE = True except ImportError:  TF\_AVAILABLE = False  print("Warning: TensorFlow not available, will use random moves")   from board import Board from move import Move from piece import Piece | Imports. The file needs the TensorFlow and NumPy external libraries and the board move and piece internal libraries  Tires to import some larger libraries but has contingencies if that is not possible |
| def \_\_init\_\_(self):  self.board = Board()  self.model = None   # Try different methods to load the model  model\_path = resource\_path('assets/chessModel.keras')  if os.path.exists(model\_path):  if KERAS\_AVAILABLE:  try:  self.model = keras.models.load\_model(model\_path)  print("Model loaded successfully with keras")  except Exception as e:  print(f"Failed to load model with keras: {e}")   if self.model is None and TF\_AVAILABLE:  try:  self.model = tf.keras.models.load\_model(model\_path)  print("Model loaded successfully with tensorflow")  except Exception as e:  print(f"Failed to load model with tensorflow: {e}")   if self.model is None:  print("Warning: Could not load AI model, will use random moves") | Loads the pretrained neural network from the file and initialises a board object to store its current state  Tries to load the model using keras and tensorflow for redundancy in the program exe file  If it cannot be loaded with either library the game is played with random moves for the black pieces |
| def get\_move(self, board):  # Get valid moves for the AI's pieces  valid\_moves = self.get\_valid\_moves(board)   # Debugging: Output the number of valid moves  print(f"Number of valid moves: {len(valid\_moves)}")   if not valid\_moves:  print("No valid moves available! Assuming checkmate.")  return None   # If model is not available, use random selection  if self.model is None:  print("Using random move selection (model not available)")  # Shuffle the valid moves to get a random one  random.shuffle(valid\_moves)   # Try each move until we find a valid one  for selected\_move in valid\_moves:  # Validate the selected move  if isinstance(selected\_move, Move) and isinstance(selected\_move.piece, Piece):  # Double-check that the move is valid  piece = selected\_move.piece  if board.valid\_move(piece, selected\_move):  print(f"Selected valid move: {selected\_move}")  return selected\_move   # If we couldn't find a valid move, try to create one  print("No valid moves found in the pre-calculated list, trying to create a new move")  for row in range(8):  for col in range(8):  piece = board.squares[row][col].piece  if piece and piece.color == 'black':  # Clear previous moves  piece.clear\_moves()  # Calculate new moves  board.calc\_moves(piece, row, col, bool=True)  # Find a valid move for this piece  if piece.moves:  selected\_move = piece.moves[0]  # Ensure the move has a valid piece  if not isinstance(selected\_move, Move) or not isinstance(selected\_move.piece, Piece):  print(f"Created move is invalid: {selected\_move}")  # Try to create a valid move manually  from square import Square  initial = Square(row, col, piece)  final = Square(selected\_move.final.row, selected\_move.final.col)  selected\_move = Move(initial, final, piece)  print(f"Created new move: {selected\_move}")  return selected\_move   # If we still can't find a valid move, assume checkmate  print("Could not find or create a valid move. Assuming checkmate.")  return None   try:  # Convert the board state to an input suitable for the AI model  board\_state = self.convert\_board\_to\_input(board)   # Debugging: Output the board state shape  print(f"Board state shape: {board\_state.shape}")   # Try to get prediction from model  prediction = self.model.predict(board\_state)   # Debugging: Output the prediction shape and content  print(f"Prediction shape: {prediction.shape}")  print(f"Prediction content: {prediction}")   # Select the best move based on the model's prediction  selected\_move = self.select\_best\_move(prediction, valid\_moves)   # Validate the selected move  if isinstance(selected\_move, Move) and isinstance(selected\_move.piece, Piece):  # Double-check that the move is valid  piece = selected\_move.piece  if board.valid\_move(piece, selected\_move):  print(f"Selected valid move from model: {selected\_move}")  return selected\_move  else:  print(f"Warning: Model selected an invalid move, using random move instead")  else:  print(f"Warning: Invalid move object from model, using random move instead")  # Try to find a valid move with the same coordinates  if hasattr(selected\_move, 'initial') and hasattr(selected\_move, 'final'):  for move in valid\_moves:  if (move.initial.row == selected\_move.initial.row and   move.initial.col == selected\_move.initial.col and   move.final.row == selected\_move.final.row and   move.final.col == selected\_move.final.col and  isinstance(move, Move) and   isinstance(move.piece, Piece)):  print(f"Found valid move with same coordinates: {move}")  return move   # Fall back to random selection if model selection fails  random.shuffle(valid\_moves)  for move in valid\_moves:  if isinstance(move, Move) and isinstance(move.piece, Piece) and board.valid\_move(move.piece, move):  return move   # If we still can't find a valid move, assume checkmate  print("Could not find a valid move after model prediction. Assuming checkmate.")  return None   except Exception as e:  print(f"Error during AI move selection: {e}")  print("Falling back to random move selection")   # Try each move until we find a valid one  random.shuffle(valid\_moves)  for move in valid\_moves:  if isinstance(move, Move) and isinstance(move.piece, Piece) and board.valid\_move(move.piece, move):  return move   # If we still can't find a valid move, assume checkmate  print("Could not find a valid move after exception. Assuming checkmate.")  return None | **Converts the board state into a format suitable for the model then Passes the board state through the model to get move predictions.** Itthen **Filters out only valid moves and selects the best move based on the model’s predictions. Returns the selected move.**  uses exception handling to account for inaccuracies in the DLNN or that the move in invalid |
| def convert\_board\_to\_input(self, board):  *"""Convert the board representation to a model-friendly format."""* board\_state = np.zeros((8, 8, 12)) # Assuming 12 piece types (6 per color)  for row in range(8):  for col in range(8):  piece = board.squares[row][col].piece  if piece:  piece\_type = self.get\_piece\_channel(piece)  board\_state[row][col][piece\_type] = 1  return np.expand\_dims(board\_state, axis=0) | Creates a **(8×8×12) tensor representation** of the chessboard where. Each square is represented by a **one-hot encoded vector. An** Example: If there's a white queen at (3, 4), the corresponding index is set to 1.  Board\_state = np.zeros((8, 8, 12)) creates a 3D array representing the chessboard state which is then fed into the NN as a complex matrix the number of dimensions is expanded upon the return of the data to the call |
| def get\_piece\_channel(self, piece):  *"""Map each piece to its respective channel."""* piece\_map = {  "pawn": 0, "rook": 1, "knight": 2,  "bishop": 3, "queen": 4, "king": 5  }  piece\_type = piece.name  if piece\_type not in piece\_map:  raise ValueError(f"Unknown piece type: {piece\_type}")  channel = piece\_map[piece\_type]  if piece.color == "black":  channel += 6 # Offset to differentiate black pieces  return channel | Assigns each piece type has a numerical channel for example  Pawn → 0 (White) | 6 (Black)  And are used to structure the 8x8x12 matrix |
| def get\_valid\_moves(self, board):  *"""Retrieve all valid moves for the AI's pieces."""* valid\_moves = []  for row in range(8):  for col in range(8):  piece = board.squares[row][col].piece  if piece and piece.color == 'black': # AI plays black  # Clear previous moves  piece.clear\_moves()  # Calculate new moves  board.calc\_moves(piece, row, col, bool=True)  # Add valid moves to the list  for move in piece.moves:  if board.valid\_move(piece, move):  valid\_moves.append(move)   # Debug output  print(f"Found {len(valid\_moves)} valid moves for AI")   if not valid\_moves:  raise ValueError("No valid moves found for the AI!")  return valid\_moves | Iterates through **all black pieces** and retrieves their **legal moves**. Uses board.calc\_moves(piece, row, col, bool=True)to calculate **valid moves**. **If no valid moves exist, the AI resigns**. |
| def select\_best\_move(self, prediction, valid\_moves):  *"""  Associate model predictions with valid moves and select the best move.  """* # Check if there are no valid moves  if not valid\_moves:  raise RuntimeError("No valid moves available for selection by AI!")   # Check if the prediction length matches the number of valid moves  if len(prediction[0]) != len(valid\_moves):  print(f"Warning: Model prediction output ({len(prediction[0])}) does not match the "  f"number of valid moves ({len(valid\_moves)}). Using random selection.")  # If there's a mismatch, just select a random move  return random.choice(valid\_moves)   # Pair valid moves with prediction scores  move\_scores = {move: prediction[0][idx] for idx, move in enumerate(valid\_moves)}   # Debugging: Output moves and their scores  for move, score in move\_scores.items():  print(f"Move: {move}, Score: {score}")   # Select the move with the highest associated prediction score  best\_move = max(move\_scores, key=move\_scores.get)   # Validate the selected move  if best\_move not in valid\_moves:  raise RuntimeError(f"Selected move {best\_move} is not in the list of valid moves!")   if not isinstance(best\_move, Move) or not isinstance(best\_move.piece, Piece):  print(f"Invalid move or piece detected in: {best\_move}")  # Try to find a valid move with the same coordinates  for move in valid\_moves:  if (move.initial.row == best\_move.initial.row and   move.initial.col == best\_move.initial.col and   move.final.row == best\_move.final.row and   move.final.col == best\_move.final.col and  isinstance(move, Move) and   isinstance(move.piece, Piece)):  print(f"Found valid move with same coordinates: {move}")  return move  # If we can't find a valid move with the same coordinates, raise an error  raise ValueError(f"Invalid move or piece detected in: {best\_move}")   return best\_move | First it prepares the compares model predictions with valid moves then selects the move with the highest rated prediction. If the number of legal moves and predicted moves does not match, then the program throws and error to ensure the selected move is valid  This is a complex user defined algorithm. It takes the NN predictions and associates the scores with valid moves then selects the ones with the highest score |

### Move.py

|  |  |
| --- | --- |
| def \_\_init\_\_(self, initial, final, piece=None):  # initial and final are squares  self.initial = initial  self.final = final  self.piece = piece  self.piece\_captured = None | Initialises 4 attributes   * initial – The starting square of the move. * final – The destination square of the move. * piece – The piece being moved (default is None). *  piece\_captured – Stores the piece that was captured during the move (initially None). |
| def \_\_repr\_\_(self):  return f"Move({self.initial}, {self.final}, {self.piece})" | Provides an official string representation for the move object in the form Move(A2, A4, Pawn) |
| def \_\_hash\_\_(self):  # Combines start and end positions to create a unique hash  return hash((self.initial, self.final)) | Allows the object to be used in hash-based collections (like sets and dictionaries). It uniquely identifies a move based on its initial and final positions.  Custom hashing function based off the initial and final squares allowing for the moves to be used as keys in a dictionary |
| def \_\_str\_\_(self):  s = ''  s += f'({self.initial.col}, {self.initial.row})'  s += f' -> ({self.final.col}, {self.final.row})'  return s | Returns a readable string representation of the move for example (0,1) → (0,3) |
| def \_\_eq\_\_(self, other):  return self.initial == other.initial and self.final == other.final | Defines equality for the move object. Two moves are considered equal if they have the same initial and final positions, regardless of the piece. |

### Helper files

|  |  |  |
| --- | --- | --- |
| File | Code | Explanation |
| Theme.py | from color import Color | The file requires the colour data to run |
| Theme.py | def \_\_init\_\_(self, light\_bg, dark\_bg,  light\_trace, dark\_trace,  light\_moves, dark\_moves):  self.bg = Color(light\_bg, dark\_bg)  self.trace = Color(light\_trace, dark\_trace)  self.moves = Color(light\_moves, dark\_moves) | Initialises 6 attributes related to the theme of the game   * light\_bg: Color for light squares (background). * dark\_bg: Color for dark squares (background). * light\_trace: Color for traces (highlighted paths) on light squares. * dark\_trace: Color for traces on dark squares. * light\_moves: Color for move indicators on light squares. * dark\_moves: Color for move indicators on dark squares. |
| Sound.py | import pygame | The game needs access to PyGame functions to run |
| Sound.py | def \_\_init\_\_(self, path):  self.path = path  try:  self.sound = pygame.mixer.Sound(path)  self.sound\_loaded = True  except FileNotFoundError:  print(f"Warning: Sound file '{path}' not found. Sound will be disabled.")  self.sound\_loaded = False | It initialises two attributes. One that holds the file path for the .wav file and another to load the sound  It disables sound if the path to the wav files cannot be found during execution |
| Sound.py | def play(self):  if hasattr(self, 'sound\_loaded') and self.sound\_loaded:  pygame.mixer.Sound.play(self.sound) | Plays the sound when called if the sound is loaded by the init |
| Const.py | # Screen dimensions WIDTH = 800 HEIGHT = 800  # Board dimensions ROWS = 8 COLS = 8 SQSIZE = WIDTH // COLS  # Colors WHITE = (255, 255, 255) BLACK = (0, 0, 0) GREEN = (137, 129, 33) DARK\_GREEN = (76, 75, 22) YELLOW = (230, 199, 103) SALMON = (248, 122, 83)  # Button dimensions BUTTON\_WIDTH = 200 BUTTON\_HEIGHT = 80  # Fonts FONT = pygame.font.Font(None, 40) | Hold s all the variables that won’t change whilst the program runs such as the screen dimensions and number of rows and columns.  It also holds the RGB values for any colours needed and the font used |
| Config.py – handles themes sounds fonts and SFX | import pygame import os  from sound import Sound from theme import Theme | It makes use of the PyGame and os external libraries and the theme and sound internal libraries. |
| Config.py | def \_\_init\_\_(self):  self.themes = []  self.\_add\_themes()  self.idx = 0  self.theme = self.themes[self.idx]  self.font = pygame.font.SysFont('monospace', 18, bold=True)  self.move\_sound = Sound(  resource\_path('assets/sounds/move.wav'))  self.capture\_sound = Sound(  resource\_path('assets/sounds/capture.wav')) | Initialises the attributes required for the config file |
| Config.py | def change\_theme(self):  self.idx += 1  self.idx %= len(self.themes)  self.theme = self.themes[self.idx] | This method **cycles through the themes** when called. It increments self.idx, looping back to 0 after reaching the last theme. It updates self.theme to the new theme. |
| Config.py | def \_add\_themes(self):  green = Theme((234, 235, 200), (119, 154, 88), (244, 247, 116), (172, 195, 51), '#C86464', '#C84646')  brown = Theme((235, 209, 166), (165, 117, 80), (245, 234, 100), (209, 185, 59), '#C86464', '#C84646')  blue = Theme((229, 228, 200), (60, 95, 135), (123, 187, 227), (43, 119, 191), '#C86464', '#C84646')  gray = Theme((120, 119, 118), (86, 85, 84), (99, 126, 143), (82, 102, 128), '#C86464', '#C84646')   self.themes = [green, brown, blue, gray] | Creates the four different board themes (green, brown, blue and grey). Each Theme object is initialised with colour tuples where the themes are store to self.themes |
| Color.py | def \_\_init\_\_(self, light, dark):  self.light = light  self.dark = (dark) | Sets a clear difference between light and dark pieces |
| Reasource\_path.py | def resource\_path(relative\_path):  *"""  Get the absolute path to a resource  """* try:  # PyInstaller creates a temp folder and stores path in \_MEIPASS  base\_path = sys.\_MEIPASS  except Exception:  # If not running as a PyInstaller executable, use the script's directory  base\_path = os.path.abspath(".")    return os.path.join(base\_path, relative\_path) | Gets the absolute path to a resource such as a .png file location |

## TensorFlow

This is perhaps the most challenging thing I’ve ever attempted. TensorFlow allows for the creation of a deep learning neural network (DLNN) that functions in theory like the human brain where data is input and passes through a selection of nodes and layers of differing size and complexity to work out the output. In my case the next best move. A DLNN is a form or Artificial Intelligence as it makes decisions based of weights and biases that are predetermined by me and its training. In theory these weights start of random and become more accurate as the model trains thus becoming better. The following functions make up the file:

|  |  |
| --- | --- |
| Method + calls | Purpose |
| def load\_pgn(file\_path):  with open(file\_path, 'r') as pgn\_file:  while True:  game = pgn.read\_game(pgn\_file)  if game is None:  break  yield game | Opens the .pgn files containing the chess games, checks if there is data. If there is it reads it else, it breaks |
| def board\_to\_matrix(board: Board):  matrix = np.zeros((8, 8, 12))  piece\_map = board.piece\_map()  for square, piece in piece\_map.items():  row, col = divmod(square, 8)  piece\_type = piece.piece\_type - 1  piece\_color = 0 if piece.color else 6  matrix[row, col, piece\_type + piece\_color] = 1  return matrix | Converts the chess boards in the files into an 8 by 8 by 12 matrices for the computer to interpret it this gets called during the function bellow |
| def create\_input\_for\_nn(games):  X = []  y = []  for game in games:  board = game.board()  for move in game.mainline\_moves():  X.append(board\_to\_matrix(board))  y.append(move.uci())  board.push(move)  return X, y | Prepares the data to be input into the DLNN by assigning it to an X and y array then appending the output of the board\_to\_matrix method to them |
| def encode\_moves(moves):  move\_to\_int = {move: idx for idx, move in enumerate(set(moves))}  return [move\_to\_int[move] for move in moves], move\_to\_int | takes a list of moves as input and encodes those moves into numerical values (integers) |
| def predict\_next\_move(board):  board\_matrix = board\_to\_matrix(board).reshape(1, 8, 8, 12)  predictions = model.predict(board\_matrix)[0]  legal\_moves = list(board.legal\_moves)  legal\_moves\_uci = [move.uci() for move in legal\_moves]  sorted\_indices = np.argsort(predictions)[::-1]  for move\_index in sorted\_indices:  move = int\_to\_move[move\_index]  if move in legal\_moves\_uci:  return move  return None | attempts to predict the best possible next move in the game based on the neural network model’s prediction and the current state of the game board. |

Checks if the device it is running on has access to a GPU for computation power

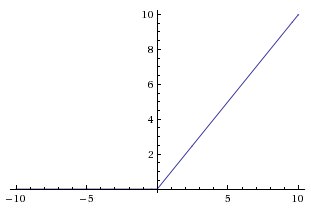
This is the logic for the DLNN that allows for its creation:

|  |
| --- |
| # Main execution print("Num GPUs Available: ", len(tf.config.experimental.list\_physical\_devices('GPU'))) print("Loading model...")  # Record time for loading model model\_load\_start = time.time()  files = [file for file in os.listdir('../assets/ChessData') if file.endswith('.pgn')] print(files) model\_load\_end = time.time() print(f"Model loading time: {(model\_load\_end - model\_load\_start) / 60:.2f} minutes") print("\nLoading games...")  LIMIT\_OF\_FILES = min(len(files), 10) file\_path = '../assets/ChessData' games = []  # Record time for loading games games\_load\_start = time.time() for i, file in enumerate(tqdm(files)):  games.extend(load\_pgn(f"{file\_path}/{file}"))  if i >= LIMIT\_OF\_FILES - 1:  break games\_load\_end = time.time() print(f"Games loading time: {(games\_load\_end - games\_load\_start) / 60:.2f} minutes")  print("\nGames loaded:", len(games)) print("Building and training neural network...")  # Limit the number of games processed LIMITED\_GAMES = games[:75000] # Adjust the limit as needed  # Record time for creating input input\_creation\_start = time.time() X, y = create\_input\_for\_nn(LIMITED\_GAMES)  y, move\_to\_int = encode\_moves(y) y = to\_categorical(y, num\_classes=len(move\_to\_int)) X = np.array(X) input\_creation\_end = time.time() print(f"Input creation time: {(input\_creation\_end - input\_creation\_start) / 60:.2f} minutes")  print("\nX shape:", X.shape) print("y shape:", y.shape)  # Record time for model building model\_build\_start = time.time() model = Sequential([  Conv2D(64, (3, 3), activation='relu', input\_shape=(8, 8, 12)),  Conv2D(128, (3, 3), activation='relu'),  Flatten(),  Dense(256, activation='relu'),  Dense(len(move\_to\_int), activation='softmax') ]) model.compile(optimizer=Adam(),  loss='categorical\_crossentropy',  metrics=['accuracy']) model.summary() model\_build\_end = time.time() print(f"Model building time: {(model\_build\_end - model\_build\_start) / 60:.2f} minutes")  # Record time for model training training\_start = time.time() model.fit(X, y, epochs=50,  validation\_split=0.1,  batch\_size=64) training\_end = time.time() print(f"Model training time: {(training\_end - training\_start) / 60:.2f} minutes")  # Record time for model saving model\_save\_start = time.time() model.save('../assets/chessModel.keras') model\_save\_end = time.time() print(f"Model saving time: {(model\_save\_end - model\_save\_start) / 60:.2f} minutes")  int\_to\_move = dict(zip(move\_to\_int.values(), move\_to\_int.keys()))  # Test prediction print("\nPredicting next move...") board = Board() print("Board before prediction:") print(board)  # Record time for prediction prediction\_start = time.time() next\_move = predict\_next\_move(board) prediction\_end = time.time() print(f"Prediction time: {(prediction\_end - prediction\_start) / 60:.2f} minutes")  board.push\_uci(next\_move)  print("\nPredicted move:", next\_move) print("Board after prediction:") print(board) print(str(pgn.Game.from\_board(board)))  Prepares the data for use and then constructs a sequential model with an input matrix, hidden layers (3), and an output node that make up the DLNN using the Adam optimiser calculating crossentropy accuracy the logging how long the whole operation took then starts to train the model of the pgn files in batches of 64 over 50 complete cycles through the models training data  # End time end\_time = time.time() total\_time = end\_time - start\_time print(f"\nTotal run time: {total\_time / 60:.2f} minutes") |

This code won’t be part of the release as it is only required on the back end to train the model. It starts by identifying the amount of GPU power available to the algorithm (in my case 1 physical GPU [apple m2 metal]) then logs the start time. The number of games that can be loaded is limited to 75000 or less to reduce ram pressure as greater than 75k uses roughly 60GB of RAM and virtual memory causing an SIGKILL 9 to be issued indicating the system has run out of memory and has self-terminated the program. When this happens all training progress and loaded files are lost and needs to be restarted and reloaded into ram.

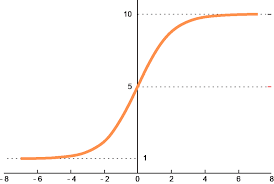
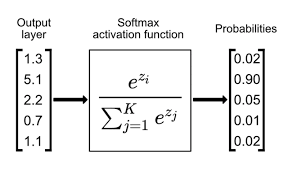
The model I used is sequential with 2 Conv2D layers that are then flattered before the data is passed through 2 Dense layers that are activated one with Relu and the other one with Softmax functions.

The sequential model is ideal is It takes one input and give one output which is what is require from the program. The one input is the chess data as a 3D matrix, and the output is the move with the best probability for success.

A 2D convolution (conv2D) creates a convolution kernel that is convolved with the layer input over a 2D spatial (or temporal) dimension (height and width) to produce a tensor of outputs that are activated by the relu function and takes an input shape of an 8x8x12 matrix which is the equivalent of 12 chess boards. The ReLU function (Rectified Linear Unit) where returns 0 if it receives a negative input but returns the value input if it is positive. The relu function can be graphed similar to the absolute value function. Using this prevents any moves being processed as negative which implies a 4th dimension to the chess game exists in some reality. This is beyond the scope of this project however, so the ReLU function is used.

These layers are then flattened which converts them form multidimensional into single dimension matrices allowing for the batch size and number of inputs to be kept the same

The dense layers implement the operation: output=activation(dot(input,kernel)+ bias) where activation is the element-wise activation function passed as the activation argument, kernel is a weights matrix created by the layer, and bias is a bias vector created by the layer (only applicable if use bias is True). I’m making use of it as it takes a 2D input with multiple dimensions (8x8 being the 2D input and x12 being the dimensions). One layer is activated by the relu function. The other makes use of SoftMax.

The SoftMax algorithm where is SoftMax, is the input vector, is a standard exponential function for input vectors, K is the number of classes in the multi class classifier and is the exponential function for the output vector. The function can be mapped and graphed as such. It is used to normalise the output network to a probability distribution over predicted output class.

When constructing a NN it is common to use SoftMax as the last layer as humans can read a probability better than the output of a hidden layer which is often a group of numbers. When making a prediction having the output as a decimal is useful as it allows for a human to interpret how much the DLNN has learned over a course of training.

## Training

To train the network the code above had to be completely reworked to make use of more optimised coding practices. First the matrix is converted to a bitboard. This is a 12x64 matrix where each plane represents a type of piece and where they are located. It is then reshaped to size 12x8x8 with 1s representing the position of the pieces. Moves are encoded into integers with the mapping saved to a JSON file. This is useful as it makes them suitable for the output labels in NN training. The file being saved allows for the encoding to be reused and understood maintaining consistency across separate passes.

When the inputs are created it is done so in parallel significantly speeding up the data preparation process with inputs mapped to X and outputs y datasets respectively.

The data is processed in batches which is essential as my computer cannot handle the scale of the dataset all at once, so it is given to it as and when needed to reduce memory pressure and increase training efficiency.

An existing neural network model is loaded. This is the one created previously. The output layer is adjusted if necessary and trains the model with the provided data. This function allows for incremental improvements and adjustments based on new or better-quality data. The code used for training is below.

The entire codebase makes use of complex user defined algorithms, complex OOP and Advanced matrix and list operations

|  |  |
| --- | --- |
| Code | Explanation |
| import os import chess import numpy as np import time import json import threading from chess import pgn, Board from keras.src.optimizers.schedules import ExponentialDecay from tqdm import tqdm from tensorflow.keras.models import load\_model, Model from tensorflow.keras.layers import Dense, BatchNormalization, Dropout from tensorflow.keras.optimizers import Adam from tensorflow.keras.utils import to\_categorical from tensorflow.keras.callbacks import ReduceLROnPlateau, EarlyStopping from concurrent.futures import ThreadPoolExecutor from queue import Queue | Imports optimised to reduce memory use by not holding large libraries such as TensorFlow when only a specific function of the library is needed |
| def board\_to\_bitboard(board: Board):  bitboards = np.zeros((12, 64), dtype=np.uint8)  piece\_types = {  chess.PAWN: 0, chess.KNIGHT: 1, chess.BISHOP: 2, chess.ROOK: 3, chess.QUEEN: 4, chess.KING: 5  }  for square in chess.SQUARES:  piece = board.piece\_at(square)  if piece:  piece\_index = piece\_types[piece.piece\_type] + (6 if piece.color == chess.BLACK else 0)  bitboards[piece\_index, square] = 1   return bitboards.reshape((12, 8, 8)) | This is the function tht generate the 12x64 matrix where each plane represents a type of piece and where they are located. It then reshaped to size 12x8x8 complex matrix with 1s representing the position of the pieces |
| def encode\_moves(moves, save\_path="move\_map.json"):  unique\_moves = sorted(set(moves))  num\_classes = len(unique\_moves)  move\_to\_int = {move: i for i, move in enumerate(unique\_moves)}   with open(save\_path, "w") as f:  json.dump({"move\_to\_int": move\_to\_int, "num\_classes": num\_classes}, f)   return np.array([move\_to\_int[move] for move in moves]), move\_to\_int, num\_classes | Moves are encoded into integers with the mapping saved to a JSON file. This is useful as it makes them suitable for the output labels in NN training. The file being save allows for the encoding to be reused and understood maintaining consistency across separate passes |
| def process\_game\_chunk(game\_chunk):  X, y = [], []  for game in game\_chunk:  board = game.board()  for move in game.mainline\_moves():  board\_matrix = board\_to\_bitboard(board) # Use bitboard  X.append(board\_matrix)  y.append(move.uci())   flipped\_board = board.mirror()  X.append(board\_to\_bitboard(flipped\_board))  y.append(move.uci()) # The move stays the same   board.push(move)  return X, y | For each game in the chunk the board state is s converted to a bitboard, and the moves recorded. It also processes the mirrored board state returning lists of board states and moves. |
| def load\_games\_threaded(file\_path, limit\_of\_files):  files = [f"{file\_path}/{file}" for file in os.listdir(file\_path) if file.endswith('.pgn')][:limit\_of\_files]  num\_threads = min(4, len(files))  chunks = np.array\_split(files, num\_threads)   queue = Queue()  threads = []   for chunk in chunks:  thread = threading.Thread(target=parallel\_load\_games, args=(chunk, queue))  threads.append(thread)  thread.start()   all\_games = []  for \_ in threads:  games = queue.get()  all\_games.extend(games)   for thread in threads:  thread.join()   return all\_games | Lists PGN files in a directory, splits them into chunks and starts a thread for each chunk in order to load games in parallel. It then collects the games from separate threads and combines them into one single list then returns it. |
| def data\_generator(X, y, batch\_size, num\_classes):  num\_samples = len(X)  while True:  for offset in range(0, num\_samples, batch\_size):  X\_batch = X[offset:offset + batch\_size]  y\_batch = y[offset:offset + batch\_size]  yield X\_batch, to\_categorical(y\_batch, num\_classes=num\_classes) | Yields the batches of board states and one-hot encoded moves from the input data |

## MacOS to Windows file conversion

Due to the program being developed in a MacOS or Unix environment any exe files produced by pyinstaller (an external library for turning projects into executables) would be in a .exec format however in order for my friend to test the project she needs it in a .exe format

In order to output an exe file from my computer I needed to make use of a Windows Virtual Machine (VM) and GitHub workflow. The GitHub workflow is used to ‘emulate’ a windows environment. This emulated environment would output a standard exe and not an exec that allows my friend to run it outside of an IDE. The workflow runs out a yaml file and is bellow

|  |
| --- |
| name: Build Windows EXE  on:  push:  branches: [ main ]  workflow\_dispatch:  jobs:  build:  runs-on: windows-latest   steps:  - name: Checkout code  uses: actions/checkout@v3   - name: Set up Python  uses: actions/setup-python@v4  with:  python-version: '3.11'   - name: Install PyInstaller and dependencies  run: |  pip install pyinstaller  pip install pygame board chess tqdm numpy tensorflow==2.12.0 keras==2.12.0   - name: Build EXE  run: pyinstaller --onefile --add-data "assets;assets" src/main.py   - name: Upload EXE artifact  uses: actions/upload-artifact@v4  with:  name: chess-exe  path: dist/main.exe |

This action runs once pushed to the repository and calls the latest windows iso file to build the running environment. Python and other dependencies are installed then the exe is built and dumped for me to share with my friend. The development environment is then collapsed and closed.

This code could be adjusted to fit any OS. To do this the section runs-on: insertOSName Would need to be changed to accommodate this.

## System requirements

the program does not require anything too modern however a modern laptop with at least 4GB of memory and some form of integrated GPU is recommended as it needs to load the DLNN player locally. If this process was done via an API, then the program can be run in a browser, but an internet connection would be required in order to play.

The program also has a requirements.txt that contains the libraires required in order to play and must be installed prior to running for the first time if the venv on the pc doesn’t already contain them . It can be done by executing the command

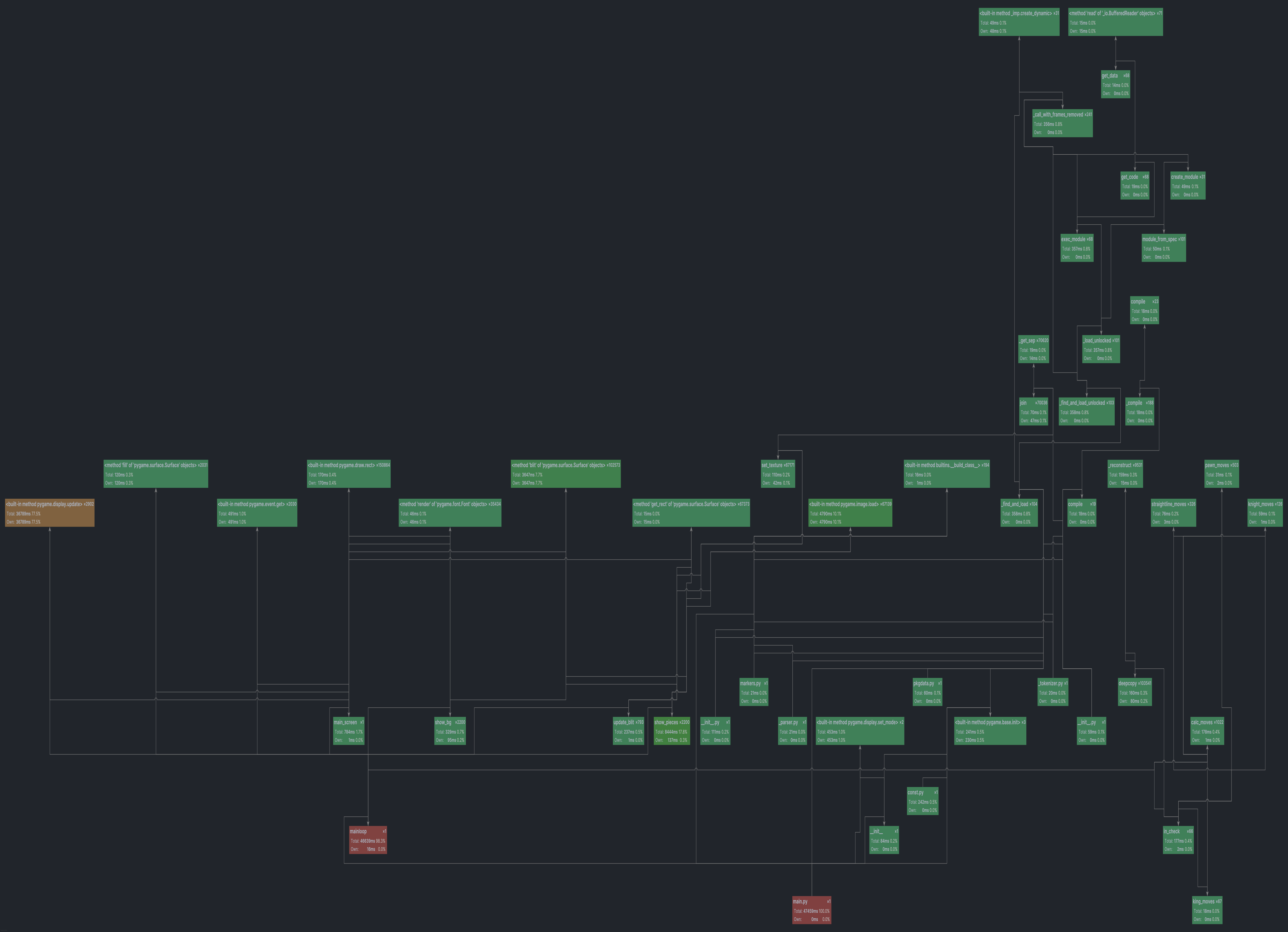
“pip3 install requiremnets.txt”

## Call diagram

A call diagram is a visual representation that shows how functions (or procedures, methods or subroutines) in a program call each other where nodes represent the individual functions, and the edges (arrows between functions) represent the relationships between them

A diagram like this is useful as it shows the structure of the codebase, where optimisations can be made to increase performance and refactoring.

The call diagram bellow was generated by my IDE PyCharm as part of an algorithm profile that included method lists and other statistics.



# Testing

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Objective | Test | Pass / Fail | Evidence | comments |
| Objective 3 Piece Logic, motion and SFX | | | | |
| 3.1 – The game must allow all legal moves and en passent and castling. | Can the code identify a legal move for all pieces on the board | P |  | The move loops through the move list and checks against the piece class to see if the move completed is a valid / legal one |
| 3.2 Different themes / colours for the board | Can the board change colour | P | A screenshot of a game  AI-generated content may be incorrect.A screenshot of a game  AI-generated content may be incorrect.A screenshot of a game  AI-generated content may be incorrect.A screenshot of a game  AI-generated content may be incorrect.A screenshot of a chess game  AI-generated content may be incorrect. | The board can cycle through 4 colour option on the press of the key ‘T’. it startsd on the classic green state |
| 3.3 Sound effects to play on moves and capture of piece and move of a piece | Does a sound clip play | P | <https://youtu.be/kH04EtOJhAs>  <https://youtu.be/kH04EtOJhAs>  two sound files |  |
| 3.4 and 3.4.1 – The game must show the available moves for a selected piece in a different colour e.g. red | Does a red box appear when a piece is in motion | P | <https://youtube.com/shorts/RYb2MdGHAkE?feature=share>   testing video | Black pieces cannot be shown without making a move as a later test prevents this from happening currently. |
| 3.5 Pieces must only move in ways that are legal and cannot be placed anywhere that would violate chess logic | All pieces on the board (pawn, rook, bishop, king, queen, knight) move as they are supposed to | P | Pawn moves including promotion for both colours, en passent, and optional double move from the start into only single moves  <https://youtube.com/shorts/RnNh4BiY0a4?feature=share>  Knight moves showing what happens if it is placed on a non-legal square. Shows that the piece can take others as well  <https://youtube.com/shorts/xt18zJc8QFk?feature=share>  All bishop moves including the pieces taking  <https://youtube.com/shorts/Jp8orj7FnNc?feature=share>  Rook moves up and down the rows including taking other piece  <https://youtube.com/shorts/IKHL1PERmEk?feature=share>  All queen moves in each possible direction including taking  <https://youtube.com/shorts/6_9g-4nvT2c?feature=share>  all king directions shown, taking not included as the king cannot be taken as a piece only put into check and checkmate.  <https://youtube.com/shorts/vQnH58baoEI?feature=share> |  |
| 3.6, Pieces work in tandem with one another and moving one will not break the logic of anther | A game of chess where moves are made randomly with no prior thought given to each move | P | Shows a series of random moves don’t affect the logic of any piece on the board  <https://youtube.com/shorts/V3AdUKTxHbg?feature=share> | This test also proved casteling works something that was not tested for in the section above |
| Objectives 4 + 1: Neural Network creation and training | | | | |
| 4.1 Does it process the data prior to parsing it through the NN for efficiency | Is it efficient in its memory usage and does it load what is necessary to run | P | loads the game files  Breaks the files down into threads that are then parsed into the CPU and GPU as and when they are needed to reduce memory pressure and load on the components  Returns a 32 bit float array over 1000 chunks |  |
| 4.2 can It display the shape of the input data | Does it write to the console | P |  | Shows all point for the NN to be trained off. |
| 4.3 can it load an existing model | Does it recognise a .keras file in the file structure, open it and train it condense | P | Loading existing model from ../assets/chessModel.keras  2025-03-06 09:53:54.313065: I metal\_plugin/src/device/metal\_device.cc:1154] Metal device set to: Apple M2  2025-03-06 09:53:54.313164: I metal\_plugin/src/device/metal\_device.cc:296] systemMemory: 16.00 GB  2025-03-06 09:53:54.313175: I metal\_plugin/src/device/metal\_device.cc:313] maxCacheSize: 5.33 GB  2025-03-06 09:53:54.313390: I tensorflow/core/common\_runtime/pluggable\_device/pluggable\_device\_factory.cc:306] Could not identify NUMA node of platform GPU ID 0, defaulting to 0. Your kernel may not have been built with NUMA support.  2025-03-06 09:53:54.313413: I tensorflow/core/common\_runtime/pluggable\_device/pluggable\_device\_factory.cc:272] Created TensorFlow device (/job:localhost/replica:0/task:0/device:GPU:0 with 0 MB memory) -> physical PluggableDevice (device: 0, name: METAL, pci bus id: <undefined>)  WARNING:absl:At this time, the v2.11+ optimizer `tf.keras.optimizers.Adam` runs slowly on M1/M2 Macs, please use the legacy Keras optimizer instead, located at `tf.keras.optimizers.legacy.Adam`. | Loads the model and displays the conditions under which it was made |
| 4.3.1, When loading a saved model does it (the saved model) initialise properly. Are all tensors loaded correctly and engaged before data is parsed through | Load the model over a randomly generated number of times (between 1 and 15) and log any issues | P | Tests to run 3   |  |  | | --- | --- | | Test 1 - Pass |  | | Test 2 – pass |  | | Test 3 - pass | A screenshot of a computer  AI-generated content may be incorrect. | | Random number generated using random number generator in google  URL - [Link](https://www.google.com/search?q=random+number+generator&sourceid=chrome&ie=UTF-8)  Number of tests to conduct - 4 |
| 4.4 does it make use of complex data structures such as a matrix or bitboard | What data structure is the data in | P |  | How the algorithm generates the data shape bellow.  The output is a 4D matrix. |
| 4.5 can it be trained to become smarter | How is it trained and does it successfully complete an epoch round | P |  | Outputs the amount of rounds (epochs) through the data set. The time taken in minutes, loss, accuracy (start and end) and the learning rate schedular  This log is generated after training |
| 4.6 does the model recognise errors in the layer formatting and can it correct them on the fly | Do the check sum layers in the code and NN .keras file work | P |  | Model self recognised an error in the layers and corrected it before training |
| 4.7 can the model make predictions for the next best move | Does it output a number to the console | P |  |  |
| 4.8 can the prediction translate into a move on the board | When I make a move as the white player can It move a black piece | P | <https://youtube.com/shorts/4KCpLrRwkqU> | When I make a move it can take and process that into a move for black |
| Objectives 5 FEN input | | | | |
| 5 allow the game to take FEN strings to build a game scenario off | Does an input box exist and can it correctly interpret FEN | F | This was never implemented due to how it interacted with the rest of the program causing other sections to crash when the string was being interpreted. |  |
| Objectives 6 UI and UX | | | | |
| 6 intuitive UI | Can the UI be interpreted by a random person with no prior knowledge of how to project is to function | P | Entering the game  <https://youtube.com/shorts/L-8ihiyuJ9A?feature=share>  Exiting the program  <https://youtube.com/shorts/mDDHjzK-wYU?feature=share> | Several of my testers were able to navigate the the game due to the limited options upon laoding the file due to the simplistic UI Desgin  These are the recordings from one of them |

# Evaluation

1. Learn how to code and deploy a deep learning Neural Network (NN).

This objective has been completed thoroughly with the result being a .keras file that holds the logic for a deep learning neural network that can predict a move base off my input to a 75% accuracy

* 1. Are there different varieties or options for ML algorithms

*I found two main libraries Pytorch and TensorFlow and evaluated the pros and cons of each coming to a reasoned verdict about the best option for my application. After creating my keras file I found a separate library in TensorFlow called TFlite that doesn’t use as much system resources however my current code is functional and changing this crucial element would take more time that I have to re implement.*

* 1. Understand how the activation functions of a network work
     1. What is an activation function
     2. How do they apply to a NN
     3. What is the math behind these functions
     4. Which is most applicable for my scenario

*This objective has been completed thoroughly. I analysed the available activation functions for my sequential model and understood how it would affect the output statistics of my model and its overall accuracy*

* 1. TensorFlow Vs Pytorch
     1. What are they and their applications
     2. Which one is better suited for my project and uses

*I fully compared both libraries and their application to my program.*

1. Discover the best language to code it in as well as frameworks .
   1. Languages could include Python, C, C++, Java
   2. Learn how to use the relevant libraries for the language.
   3. Is an API necessary for processing the data

*Overall python was the best High level language for my usage. I am already fluent in it, and it is the best language for ML / AI with the wide range of libraries that can be accessed for this purpose. PyGame is also useful as it is simple to use with documentation readily available.*

*No API was necessary as the clients’ requirements stated it had to be fully offline so an API would be useless as it wouldn’t fit the brief.*

1. Code a playable game of chess in which no errors occur that affect the gameplay.

*This objective is fully complete. The game allows for multiplayer (PvP) matches with the capability for a computer opponent to be implemented*

* 1. The game must allow all legal moves and en passent and castling.

*Objective complete there is code that enables these two features with appropriate names*

* 1. Different themes / colours for the board

*Objective complete, there a four separate board colours that the user can cycle through by pressing the ‘T’ key*

* 1. Sound effects to play on moves and capture of piece

*When a piece is captured or is place onto a new square a sound is played dependent on the case presented and if it was moved or captured*

* 1. The game must show the available moves for a selected piece in a different colour e.g. red
     1. It must cope with any pieces that may be in the path of that piece and adjust its output path accordingly

*Objective complete. The game accurately displays the legal moves for any piece on the board and stopes if an opposing piece is in the path of its possible moves*

* 1. Pieces must only move in ways that are legal and cannot be placed anywhere that would violate chess logic

*Objective complete, no logical issues arise when moving pieces and pieces cannot be moved in a what that violates the laws of chess*

* 1. Pieces work in tandem with one another and moving one will not break the logic of anther

*All pieces can exist on the same board with all they’re different move sets and logic peacefully coexisting in the same instance of the game.*

1. Code a neural network that takes in an input and gives out the predictions for the best move

*Objective complete. It tells the position of the next move and the statistics for the top 3 moves as a decimal (e.g. 0.67)*

* 1. Does it process the data prior to parsing it through the NN for efficiency

*Data is split between 1000 threads and then distributed to the 8 cores of my computer to reduce the pressure on my RAM CPU and GPU. This made the training of my model more efficient and faster cutting the time per 50 epochs from 5 days to 10 hours*

* 1. Can it display the shape of the input data

*Objective complete, outputs the x and y matrices to the console for review. Output in a developer friendly format*

* 1. Can It load an existing model to make it better
     1. When loading a saved model does it (the saved model) initialise properly. Are all tensors loaded correctly and engaged before data is parsed through

*Models are saved, loaded and initialised correctly post and prior to training and data parsing*

* 1. Does it make use of complex data structures like matrices or bit boards

*Objective complete. Data is passed through matrices, bitboards, arrays, queues and networks to correctly output the final statistics in the required format*

* 1. Can it be trained to become smarter

*Yes.*

* 1. does the model recognise errors in the layer formatting and can it correct them on the fly

*yes, it recognises if the model structure is wrong then reconstructs the nodes to a correct structure*

* 1. can it make predictions for the next best move

*Yes. It outputs a statistic, piece and moves in Forse-Edwardian notation.*

* 1. can the prediction translate into a move on the board

*No, it cannot yet interface with the program to move pieces around the board however it can predict the next best move*

1. Allow the game to accept industry standard game setup codes.

*Objective Failed. It wasn’t a requirement for the client so its lacking in deployment isn’t a concern to her*

* 1. Understand the industry codes for generating chess games (FEN).
     1. How do I integrate this into my game

*Failed*

* 1. Have the game output a useable FEN string.

*Failed*

* 1. Have the game accept a FEN string.

*Failed*

1. The user interface should be intuitive and easy to navigate

*Objective complete, two buttons, one launches the game the other quits the program*

* 1. Alternatively, it can boot straight into the game negating the need for a UI to navigate through

*Objective complete, it doesn’t need to as it passes through a start screen that gives the option to quit the program or a start a game*

* 1. The difference between efficient UI and an easy and intuitive UX

*Objective complete, Good UI and UX stem from simple and intuitive designing that makes the user navigation as simple as possible*

1. Understand how to create / maintain a repository through GitHub.

*Objective complete. Git repository can be viewed here (*[*https://github.com/English-Garfield/CompSciNEA\_IK*](https://github.com/English-Garfield/CompSciNEA_IK)*) with a release that works.*

* 1. understand common terminology for git.

*Objective complete, I understand commits, push, branch and force in relation to git*

* 1. link my IDE to GitHub so that my friend can access the project as its being developed to give feedback if she wants to

*objective complete, she was able to access the project during development at see if it was up to standard*

* 1. create and maintain a .md file containing my project details

*objective complete. The project contains a markdown (.md) file that explains the purpose of the project and the technologies used to develop it.*

## end user feedback

after allowing my client to use the program she gave me the following feedback

Positives:

She said the UI and UX was simplistic and intuitive and simple to use and interface with. She enjoyed changing the themes and didn’t need to be told the key to press (‘T’). she found the sound the pieces made satisfying and representative of what would be made is a physical piece is moved, and the capture sound gave the right feel for that action.

She found the AI to be of a reasonable level and challenging to play against that would push her to be a better player. In the future it could be adapted to learn from her moves or other games to become a better player in itself.

The one click startup enabled by an exe file was useful to her as she didn’t have to navigate an IDE each time she wants to play

### Improvements:

The AI made fundamental mistakes that a person of 2000 ELO wouldn’t make and could crash on occasion if the number of moves exceeded 100 a test case not accounted for in testing. This is due to the RAM and VRAM on the GPU holding too much data leading to the program being stopped.

She would have preferred a web UI rather than pygame built in GUI. Recently a library called StreamLit has been released that would have fit the use case perfectly. It is lightweight and would run smoother than the pygame built in GUI. Regarding UI design it could have been designed better. The design allowed for a simple UI however it could have been better looking. Again, the StreamLit library would fix these issues.

Regarding FEN input the lack of the feature was insignificant and didn’t affect the using experience however made for some annoyance if she wanted to take note of the game position and play it against a human rather than a computer.

Overall:

The product produced fits my use case and needs with any problems being minor or insignificant as the core program works as expected. A program that does what I asked and nothing else

# Bibliography

* PyGame
  + <https://www.pygame.org/docs/ref/display.html>
* TensorFlow website and resources and install
  + <https://www.tensorflow.org/>
  + <https://www.tensorflow.org/install>
  + <https://www.tensorflow.org/tutorials>
* Research Papers
  + Moroney, L(2020). AI and machine learning for coders. O’Reilly Media
  + Silver, D (2018). A general reinforcement learning algorithm that master’s chess, shogi, and go through self-play, Deep mind
  + Chollet, F (2018). Deep learning with python. Manning Publications
  + Klein D (2023). Neural Networks for chess. Klein D
* Assets
  + <https://commons.wikimedia.org/wiki/Category:PNG_chess_pieces/Standard_transparent>
  + <http://images.chesscomfiles.com/chess-themes/sounds/_MP3_/default/move-self.mp3>
  + <http://images.chesscomfiles.com/chess-themes/sounds/_MP3_/default/capture.mp3>