



GAME AND ENGINE FOR CHESS IN 3- DIMENSIONS

CENTRE NUMBER: 16437

CANDIDATE NUMBER: 8527

THOMAS BALE



CONTENTS

1.1.1 – Project proposal – Project Outline.....	5
1.2.1 – BackGround – What is 3D chess?.....	6
1.2.2 - BackGROUND – Rules of 3D chess	6
1.3.1 – IDentification of users – abouts the users.....	8
1.3.2 – Identification of Users – 1 st interview.....	8
1.4.1 – Prototype (rl) – Problem with research and follow up	10
1.4.2 - PROTOTYPE (rl) – showcasing the prototype	10
1.4.3 - PROTOTYPE (rl) – What I found after play	10
1.4.4 - PROTOTYPE (rl) – 2 nd interview with client after play.....	11
1.5.1 – Research – issue with research	12
1.5.2.1 – Research – Exisitng solutions – websites – chess.com.....	12
1.5.2.2 - Research – Exisitng solutions – websites – lichess.org	18
1.5.3.1 – Research – Exisiting Solutions (AI) – Stockfish	19
1.5.3.2 – Research – Exisiting Solutions (AI) – komodo	21
1.5.3.3 – Research – Exisiting Solutions (AI) – komodo's dragon	22
1.5.3.3 – Research – Exisiting Solutions (AI) – Deep blue.....	22
1.5.3.3.1 – Research – Exisiting Solutions (AI) – Alphazero – Deep Mind	23
1.5.3.3.2 – Research – Exisiting Solutions (AI) – Alphazero – LC0.....	23
1.5.4.1 - Research – Machine Learning – Q-learning and mdp	23
1.5.4.2 - Research – Machine Learning – Neural Networks	24
1.6.1 – Research – Practicality for AI – Reinforcement learning	26
1.6.2 – Research – Practicality for AI – Fully Custom Logic	26
1.6.3 – Research – Practicality for AI – Conclusion.....	27

1.7 – end goal objectives	27
1.8.1.1 – Computational Prototypes – UI - 2D	32
1.8.1.2 – Computational Prototypes – UI – 3D	32
1.8.2.1 – Computational Prototypes – AI – Customisable NN.....	35
2.1- Overall Design Discussion	43
2.2 – Logic Chart.....	51
2.3 - Class Diagram.....	53
2.4.1.1 - Algorithms – Neural Network - intro	54
2.4.1.2 - Algorithms – Neural Network – Training data.....	54
2.4.1.3 - Algorithms – Neural Network – Backpropogation algorithm.....	56
2.4.1.4 - Algorithms – Neural network – backpropogation learning	58
2.4.1.4 - Algorithms – Neural network – feedforward - algorithm.....	59
2.4.1.4 - Algorithms – Neural network – feedforward - data.....	60
2.4 - Algorithms – Legal Moves.....	61
2.5.1 – File Structure	66
2.5.2 - Organisation	71
2.6 – HCI.....	73
2.7 - Hardware selection	75
2.8 - Data Structures.....	75
Lists	75
Sorted Dictionaries.....	77
Dictionaries	80
Neural network implementation – main complexity of project	82
3.1.1.1 – Technical solution – algorithms – Back propogation - used	105
3.1.1.1 – Technical solution – algorithms – Back propogation - unused -1	107
3.1.1.1 – Technical solution – algorithms – Back propogation - unused -2.....	108
3.1.1.1 – Technical solution – algorithms – Back propogation - unused -3.....	109
3.1.2.1 – Technical solution – algorithms – Feed forward (MM) - Used.....	110
3.1.2.2 – Technical solution – algorithms – Feed forward (MM) – unUsed -1	111
3.1.2.2 – Technical solution – algorithms – Feed forward (MM) – unUsed -2	111
3.1.2.2 – Technical solution – algorithms – Feed forward (MM) – unUsed -3	112
3.1.3.1 – Technical solution – algorithms – Activation and cost function implementations.....	112
3.1.3.2 – Technical solution – algorithms – Training Data Creation and Handling	114
Data Creator	115
NN Interpreter	140
Outside AI Parent functions that are used.....	141
Board	141
NN Manager	142

About the AI – Complex User Defined code/System	142
3.1.4.1 – Technical solution – algorithms – AI – I	142
AI1 class	142
parent functions used	143
3.1.4.2 – Technical solution – algorithms – AI – 2	145
AI2 Class	145
Parent functions used	146
3.1.4.3 – Technical solution – algorithms – AI – 3	151
AI3 Class	151
Parent Functions USed	152
3.1.4.4 – Technical solution – algorithms – AI - 4	162
Parent Functions used	163
3.1.4.5 – Technical solution – algorithms – AI – 5	173
Parent Functions Used	173
3.1.4.1 – Technical solution – algorithms – AI – Specific function logic	181
3.1.4 – Technical soltuion – Algorithms – Legal MOves	233
Generate player moves	233
Pawn	234
Knight	235
Bishop	236
Rook	237
Commoner	237
Queen	239
King	240
making sure the king is not put into check or stays in check	242
Controller	242
ChessPlayer	244
Removing Illegal Check Moves	245
Chessgame controller	245
board	245
3.1.5 – Technical Solution – Algorithms – Merge SOrt	246
AI Manager	246
3.2.1 – Technical solution – 3 rd party code – Camera Free Look	248
3.3 – ALL COde (next headers represent class names)	251
Chess AI	251
AI1	251
AI2	252
Ai3	252

AI4.....	253
AI5.....	254
AI Manager.....	255
Neural Network.....	307
NN Manager.....	330
Chess Game.....	332
Pieces (folder for Piece scripts – not piece class).....	332
Pawn.....	332
Knight.....	333
Bishop.....	334
Rook.....	335
Commoner.....	336
Queen.....	337
King.....	338
Board.....	340
CHess Game COntroller	351
Board Layout	367
Chess Player.....	368
Piece (the piece class- parent to all piece classes).....	370
Pieces Creator.....	372
Sqaure Selector Creator.....	373
Data Structures.....	373
Piece Moves	373
Input System.....	373
Board Input Handler	373
collider input reciever	374
Debug Input Handler – unused during game.....	374
I Input handler (interface)	375
Input Reciever	375
UI input Handler	375
UI Input reciever.....	376
Non Game Utility.....	376
Training Data	376
Sound	401
BackGround Music	401
Sound manager.....	402
Tweeners	403
Instant tweener	403

I Object tweener (interface)	403
UI Controllers.....	403
Camera Free Look – 3 rd party code.....	403
Chess UI Manager.....	406
Utils	409
Board Button	409
Custom Exception – unused during gameplay.....	409
Material Setter.....	410
Official notation – unsued during gameplay (covered in board).....	410
random helper – unity random fails due to seed issues	411
Time helper.....	411
UI button	412
Preliminary.....	412
4.1 – The Board.....	417
4.2 – The Game.....	418
4.3 – Testing – UI.....	424
4.4 – Testing – AI	426
5.1.1 – Objectives – completeness.....	436
5.1.2 – Objectives – How It was Achieved	436
5.2 – IMPROVEMENT.....	436
5.3.1 – USER FEEDBACK – relating to 1 st and 2 nd interviews.....	437

I.0 ANALYSIS

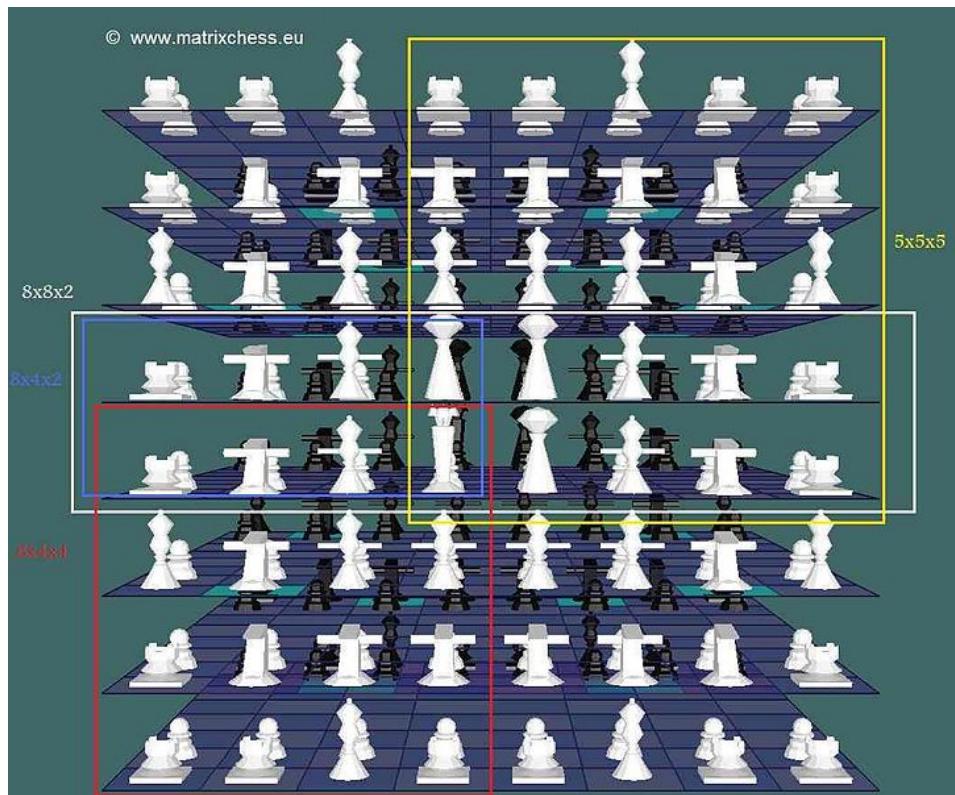
1.1.1 – PROJECT PROPOSAL – PROJECT OUTLINE

Oscar, an intelligent student at sixth form, is an avid chess player who likes to play many different chess variants. On top of this, he also enjoys playing chess on his computer and mobile device. His favourite variant is 'Strada chess' in which the chess pieces move through three stacked boards. What he enjoys most is the 3D aspect of the game as it allows a for a higher level of thinking.

There lie two problems for Oscar: he would like to play another version of chess where the game is more 3-dimensionally involved, whilst also being able to play the game without physical pieces or setup. Therefore, Oscar would like to have a game produced where he is able to play on an electronic device where the game works in the same degree as chess, but with altered rules.

As this would be a new game for a niche group other than Oscar he would also like to play against an AI as a real opponent may not be available to play when he would like. This AI will have to be tailor made to this specific game as a standard chess AI will not compute the necessary moves.

1.2.1 – BACKGROUND – WHAT IS 3D CHESS?



(Note – FIDE chess is the tournament standard ‘normal’ chess)

3D chess is played inside an 8x8x8 cube as opposed to the FIDE chess 8x8 board. One set of 32 pieces is used per layer (totalling to 256 pieces). The pieces also move differently from FIDE chess to allow them to move between ‘layers’. A representation can be seen above:

1.2.2 - BACKGROUND – RULES OF 3D CHESS

To fully understand how 3D chess works the rules that differ from FIDE chess are shown below (FIDE chess being the tournament standard chess).

- The playing field is an 8x8x8 cube instead of an 8x8 square.
- All squares in a single vertical stack are of the same colour: for example, a1 is a black square on each of the 8 8x8 levels.
- The standard algebraic notation is modified by the addition of the level number which is prefixed to the name of each square. Thus, the bottommost square a1 is 1a1, and a1 on the top level is 8a1.

- Eight standard chess sets are used. Each player has the standard Chess line-up on each of the eight levels.
- The Kings on 4e1 and 4e8 must be distinguished from the others in some obvious way. Rules about check and checkmate apply to only these two Kings; the other Kings are not royal, and in fact are called Commoners.
- Castling can involve Kings and Rooks from different levels.
 - It is obvious that from 4e1 the White King can Castle to 4g1 bringing the Rook from 4h1 to 4f1; after all, this is all on the same level, so it looks like FIDE Chess.
 - Equally obvious, "from 4e1 to 2g1 with 1h1", "from 4e1 to 6g1 with 7h1", "from 4e1 to 4c1 with 4a1", and "from 4e1 to 6c1 with 8a1".
 - However, "from 4e1 to 2c1 with 1a1" is a special rule: it really ought to be "with 0a1" but there is no level 0.
 - Vertical Castling, for example "from 4e1 to 6e1 with 8e1", is not allowed.
 - The notation for Castling "from 4e1 to 2g1 with 1h1" is "O-O(2g1)", in other words, we give the King's destination position as part of the move; this can be omitted if only one O-O is possible.
- The normal two-dimensional moves of the pieces are translated into three dimensions in the following manner:
 - A Rook makes a standard Rook move, or moves straight up and down, or rises or descends as it makes a Rook-move. For example, a Rook on 1a1 can (if nothing is in the way) move to 8a8, passing through 2a2, 3a3, 4a4, 5a5, 6a6, and 7a7.
 - A Bishop makes a standard Bishop move, or rises or descends; for example, from 1a1 to 8h8 via 2b2, 3c3, and so on.
 - The Queen moves like the FIDE king but the moves ray outwards in lines.
 - The King and the non-royal King which we call the Commoner move as the Queen, but just one square at a time.
- Pawns rules are as follows:
 - Pawns may only move straight forward on the same level, but when they capture, they may not stay on the same level. In other words, Pawns are move in one dimension but capture in three dimensions.
 - For example, from 4e4, a White Pawn could capture at 3d5, 5d5, 3f5, or 5f5. The normal two-dimensional Pawn captures at 4d5 and 4f5 are not available.

- Knight rules are as follows:
 - The Knight is completely and exclusively three-dimensional: it jumps two squares in one direction, and one square in each of the other directions.
 - From 4e4, a Knight can go to 24 different places: 3d6, 5d6; 3f6, 5f6; 3g5, 5g5; 3g3, 5g3; 3f2, 5f2; 3d2, 5d2; 3c3, 5c3; 3c5, 5c5; 6d5, 6f5, 6f3, 6d3; 2d5, 2f5, 2f3, 2d3.
 - Notice that the Knight cannot make its normal two-dimensional move because it must always change levels.

In summary, all piece rules from FIDE chess follow the same mathematical pattern, except for 3 dimensions instead of two on top of their classic 2d moves. The pawn and knight are exempt from this where they may not use their 2d moves, only 3d moves.

1.3.1 – IDENTIFICATION OF USERS – ABOUTS THE USERS

Although that this project is predominantly being designed for Oscar, there are other potential users. These would include other chess/ chess variant players, people who enjoy strategical videogames, those who may wish to use this as mind strengthening exercise for games, maths, logical thinking. The main benefitters inevitably will be those who will use this as they enjoy chess and hence this shall be mostly tailored to them (this is that planning goes under the assumption that the user has experience with FIDE chess).

As for accessibility, the compute power of the program must be considered. Although Oscar has a high-end computer system and up to date mobile, there may be users who do not. For this reason, the program must not be largely resource heavy and should run 'smooth' on a modest device for current times.

1.3.2 – IDENTIFICATION OF USERS – 1ST INTERVIEW

Q: What is your experience with chess?

A: I play regularly and have a 2300 chess rating (intermediate average is roughly 900). I have played a few chess variants but never 3d chess.

Takeaway: My client has a complex understanding of chess and will be able to contribute complex ideas and will be able to identify problems that I may not pick up on.

Q: With regards to the UI, what would you be expecting?

A: I would like the board to be able to be split up so that certain or individual layers can be seen separately. I would also like to be able to rotate the board and see which pieces have been captured. Additionally, a feature to see a piece's available moves would be nice.

Takeaway: Being able to visualise your moves is very important and as such having an easy to use, understand and visualise playing field should be a priority.

Q: What would you expect from the AI?

A: There should be different levels of AI with one that will lose to an average intelligence beginner but also one that would beat an experienced player. I would like the AI to be able to play as close to as a human would as possible.

Takeaway: A varying and competitive AI is important for play to be intriguing but also the AI should behave human-like to provide more realistic play.

Q: What time restrictions on time would you like to see?

A: Just as with FIDE chess I think there should be an untimed, standard, blitz and bullet options, however, at this time I am not sure what these exact times at this moment.

Takeaway: There should be multiplied time modes, however, until the game is played the specifics are uncertain.

Q: What game modes would you like?

A: Player vs AI, player vs player. Perhaps an AI vs AI to learn scenarios.

Takeaway: Having variation in what can be done inside the game is important for the user.

Q: How would you like games to be recorded/ displayed back to you?

A: Moves should be displayed on a sidebar as they happen.

Takeaway: A move sidebar feature

Q: How would you like this game to be available I.e. on a large store front, a website?

A: Preferably as a computer app as I feel a website would be too slow but would be fine if it is not slower. Perhaps also a phone application.

Takeaway: The user seems to be happy with most forms that it could be available in, however, they seem keen to play it on a computer.

Q: What hardware/software restrictions may affect the making of this project?

A: I think that this program should be able to be ran on a mid-range system (this may affect the AI). Like a phone.

Takeaway: The AI nor UI can be so complex that an average device will struggle to compute the game.

Q: What additional features above gameplay with an AI would you be expecting?

A: Player on player gameplay and AI vs AI gameply. I have no desire to play other people on a WAN as the games take too long and that is too much time commitment. Sounds would be good too.

Takeaway: The user just wants a way to play against other humans too on the same device. There should be piece move sound effects and background music.

Q: What/if any two player capabilities would you like?

A: Just standard play with different time limits

Takeaway: No extras needed.

1.4.1 – PROTOTYPE (RL) – PROBLEM WITH RESEARCH AND FOLLOW UP

Due to the nature of this project with this being the first product of its kind made there are no available versions for me to try and research. Therefore, for my primary research I create a real-life prototype of the game. Other research will focus on FIDE chess and can be seen in 1.4 Research.

1.4.2 - PROTOTYPE (RL) – SHOWCASING THE PROTOTYPE



As shown above the prototype was made from 8 8x8 board lined up to represent the cube.

1.4.3 - PROTOTYPE (RL) – WHAT I FOUND AFTER PLAY

I believe this was the most important section of analysis for my project as it greatly helped both myself and the client to gain a better understanding of the game.

The main issue that I found after playing was the time with games varying from 4-6 hours long during free play. This poses a question about whether this could be boring for users. A possible solution is to reduce the number of boards used in play. My main idea was to play on four boards with one checkable king as opposed to 8 (same rules can still apply otherwise).

Thinking time and possible moves was another possible problem that arose in my mind. Due to the increased size of play it is unreasonable for a player to look at all the moves available (in fact we only look at a minuscule proportion of possible moves) – this could create problems with the AI being playing at massive skill difference with only a 2-move look ahead. Aside from the AI players will have to be competent with their ability to think logically as the skill curve gradient for this game will be immense.

Check mating is extremely hard in this game due to the nature of the king's movement, namely: with pieces not being able to 'corner' the king so easily due the 3d nature of the board and, it is easy for the king to 'run away' through different levels. This does make the end game more intriguing for a human, however, coding an AI for this will be difficult as it is a very intuitive process. Something to consider would be to use deep Q reinforced learning or simply a neural network for the AI. Training time could be a limiting factor if I decide to take this route.

The last notable point is that board position is very different from FIDE chess. It is hard to determine where the 'powerful' positions on the boards are compared the FIDE chess where you can determine that the centre of the board is the most powerful. The again will bring problems when coding the AI – it will take a lot of playing from me to try and determine where these 'powerful' positions may be.

1.4.4 - PROTOTYPE (RL) – 2ND INTERVIEW WITH CLIENT AFTER PLAY

Q: How/would you change the overall time of the game?

A: Different time modes as mentioned before.

Takeaway: Time is an issue and I need to set objectives that allow for shorter gameplay.

Q: Do you think there should be a limitation on thinking time for the human?

A: Yes, of varying amounts of the user's choice.

Takeaway: There should be an ability to create different time mode modes with different starting times and added time.

Q: Do you think there should be a limitation on calculation time for the computer?

A: There should be different levels of AI that do fewer calculations.

Takeaway: Having varying AI levels is important not only for fun gameplay but also for competitive gameplay. (Quick calculating AIs and more complex AI)

Q: How strong should the AI be and what should it consider?

A: It should be strongly capable of beating the best of players but with abilities to reduce its own ability. It should prioritise piece taking as opposed to board position – the exception to this is where the piece will be taken by a lower value piece if moved to a location. Try to defend or move high value pieces

Takeaway: A strong AI is important to the user but equally important is one that can vary in ability. After understanding the core principles of the game, both I and the user have discovered that for 3D chess taking pieces early is essential for a game to conclude past as board position matter little compared to FIDE chess till near the end game. As such, this should be factored into the AI.

Q: How/would you change the end game and/or how the game can be concluded?

A: Time runout with piece points determining the winner or classic timed play.

Takeaway: This would allow for competitive gameplay with more positioning thinking without the dread of taking the game on for several hours.

Q: Are you happy with how the pieces move?

A: Yes, the 3d capabilities make it much better than Strada.

Takeaway: The rules should not be changed.

1.5.1 – RESEARCH – ISSUE WITH RESEARCH

As of the date of writing (26/05/22) there are no chess variant games that play through 3 dimensions provided through mainstream online storefronts. (Specific sites checked include steam, epic games, Microsoft store, apple store, google play store).

As a result of this I will conduct most of my research through FIDE chess, commenting on any changes necessary for it to be applicable for 3D chess

1.5.2.1 – RESEARCH – EXISTING SOLUTIONS – WEBSITES – CHESS.COM

With over 70 million users Chess.com is the most used chess app or website in the world for FIDE chess and is used by many grandmasters daily to train. Shown below is UI used for a match only as this is the only feature I am interested UI wise.

The chess pieces are shown in two dimensions which is something that I should consider alongside a 3D environment option. This can simply be implemented (from a 3d version) by unstacking the layers and changing the 3d objects for sprite images or simply flat objects. In general, I can implement both the 3d and 2d board through a 3d array of positions with piece titles (FIDE chess notation) in the 3d array. This may make it easier for users to visualise moves. As for sound, there is no background music but there is the option for sounds effects when pieces move, pieces capture, when check occurs and when checkmate occurs. The sounds are simplistic, and recognisable as they are essentially 'universal' in the online Chess world. Since they are copyright free I will most likely use these mp3s myself.

Also shown are the orange arrows. These are for pre-moves where a user (in this case using a right click) can plan moves ahead visually. This will be particularly useful in my project as planning is used to an even greater extent than in FIDE chess. I can implement this by arrows that are either object that can be enlarged and translated to show the respective move the user has 'pre-moved'.

Additionally, seen on the right, is a move log in which previous moves are shown to the user. This is something that my client stated they would be interested in. As an extension of this feature there is also



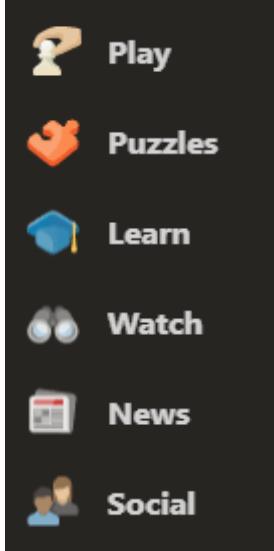
a 'move rating' for each move made where it shows how each move affected the position of both black and white. This may be something worth implementing in my game to aid learning, thought this will come at some computational expense. Implementing the past move list can be done simply by taking the name of the piece moved from the 3d array of positions and the starting and end position in said array, then displaying this to the user via text (and in this case a sprite for the respective piece).

One small detail to look at and compare to other FIDE chess games is the colour of the tiles. Whilst it may not seem important, when it comes to 3D chess being able to easily distinguish what tiles line up makes planning easier and hence planning time shorter. The green and cream colour scheme works well as both contrast each other and the black background. They also include an option to change the ‘theme’ of the board. This is not of the upmost importance, but it would be easy to implement by simple colour changes of the board and piece objects.



Aside from the UI there are other features that I investigated. Chess.com allows two ways to play using a chess engine, these are the analysis board and ‘playing against AI personalities’. The analysis board uses an AI engine, Stockfish -1.5.3.1, where you can play against yourself or another person on the same device while the AI analyses and feedbacks on the moves made. This can be seen above.

As for playing against personalities, Komodo – 1.5.3.2, is used where personalities can be replicas of how celebrities play, certain play styles i.e., aggressive, passive, or adaptive (to how you play) as well as being able to vary ability.



Both above would be something I am interested in looking into, practically it may be infeasible to implement the former exactly as the number of moves available in 3D is exponentially larger than in FIDE chess. This would mean to rank all possible moves and how they affect the future game would take too much computational power. I could still use this feature with an algorithm that utilises a heuristic approach with approximate worst, best and mean values or by rough standard deviations (make assumptions for N, mu and xi).

$$\sigma = \sqrt{\frac{\sum(x_i - \mu)^2}{N}}$$

σ : Population standard deviation

x : Datapoint value

μ : Population mean

N : Population size

Shown to the side are the other options that Chess.com. Most of these functions will not be useful for my project. Social may be of use if the game is to be played across a WAN. News will not be relevant as the news function here depicts new chess information. Watch allows a user to see AI vs AI or Grandmaster games, which is not necessary.

Puzzles are the other main feature it is not one my client said they would like to be included but should still be considered. Puzzles work by setting out scenarios that may occur in a game and you must reach the goal (usually checkmate) in the least moves possible. An example is showcased below (3rd picture shows what using the hint button will do. These are useful for brain training and learning specific situations better. Since the common checkmate patterns and such are unknown for 3d chess this feature does not seem applicable to my program.

The other section of interest for me is the learn function where words, pictures and videos are used to explain how to play the game for different levels i.e. basic, intermediate and advanced strategies and plays as well as instructions on how to play. There are also additional functions such as the 'classroom' where you can be taught be a real coach online, analysis of any game played by any two players, a



database of opening, insights into any players chess history and endgame techniques. This can all be seen below.

New to Chess (How to Play)



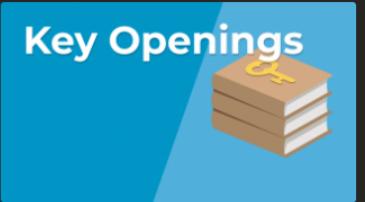
How to Move the Pieces

Get to know the chess pieces and how to set up the board.

Chess.com Team

8 Lessons Beginner

Advanced (Taking Control)



Key Openings

Understand what makes the most popular openings great

Chess.com Team

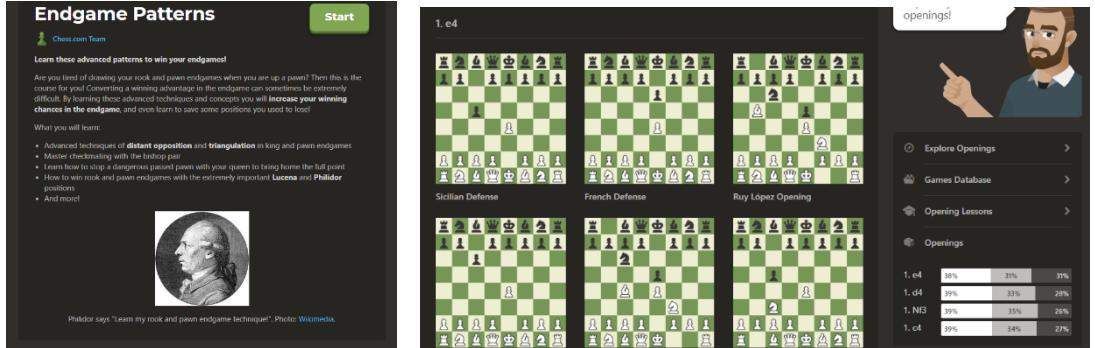
8 Lessons Advanced

Endgames

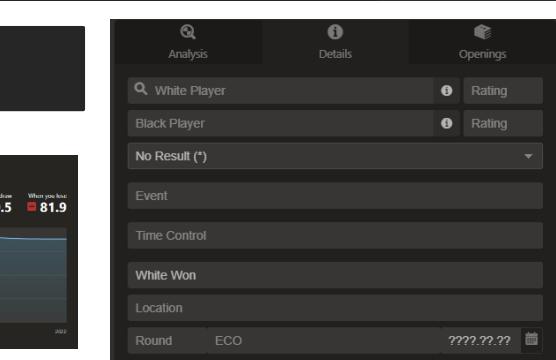


Themes Leaderboard

-  Checkmates
-  Pawn
-  Minor Piece
-  Rook
-  Queen
-  Imbalances



The interface shows various endgame patterns like Sicilian Defense, French Defense, Ruy Lopez Opening, Caro-Kann Defense, Italian Game, and Sicilian Defense: Closed, each with a small diagram of the board setup.



The analysis section includes a search bar for White Player (Hikaru) and Black Player (Rating), a chart showing Average accuracy over time (87.77%), and a list of recent games.

Move	White	Black	Result
1. e4	30%	31%	31%
1. d4	29%	33%	28%
1. Nf3	27%	33%	26%
1. c4	30%	34%	27%

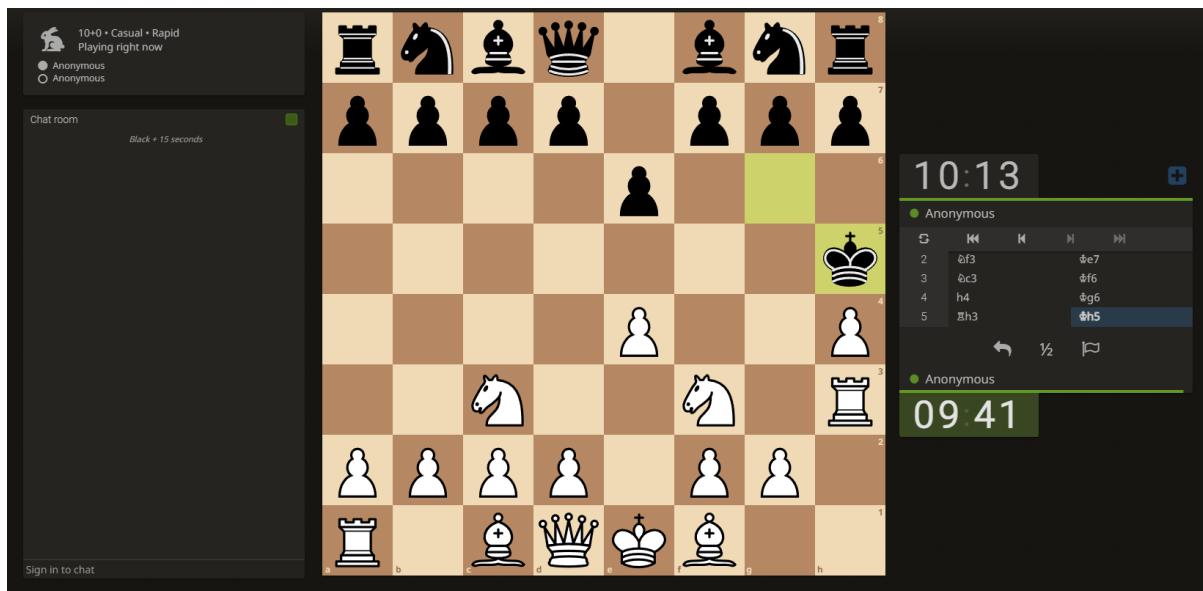
Lastly, Chess.com supports all FIDE chess time modes, however no option for custom time rules. This is something I would want to use in my game as time plays a very large factor with how intriguing the gameplay is.

17

1.5.2.2 - RESEARCH – EXISITNG SOLUTIONS – WEBSITES – LICHES.S.ORG

Lichess.org is a site the Oscar uses frequently. It has roughly 15 million users with grandmasters using it too, making it one of the most popular chess sites. It is worthy to note that through my research for this I will not be going to depth on any features that are the same as that in previous research.

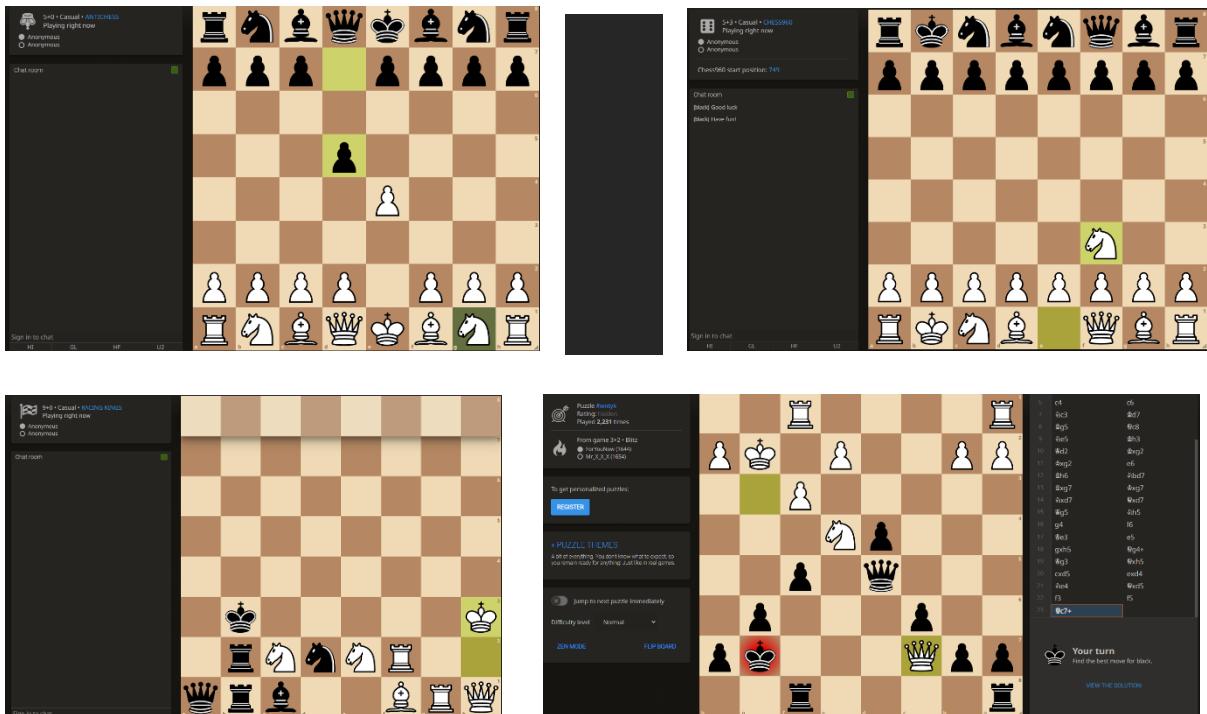
Shown below the UI when playing a game against another player.



The main and obvious difference here when compared to Chess.com is the chat function on the left-hand side. This, if play across a WAN is to be used (decided upon later), would be a useful function. There are some issues with live chats however regarding abuse. I am not, thought, too concerned with this as chess players are not typically people to abuse the opponent due the rules and etiquette associated with FIDE chess. For this reason, if I were to implement this function there would only have to be very simple forms of chat moderation. The other two noteworthy differences are the board colour and lack of sprites in the past move section. Although the 'classic' chess colours are used here I would still opt for the colours used in Chess.com as the default as I found it easier to plan moves with the higher contrast. I will also create other themes that the user can select with classic being one of them. As for the lack of sprites, the formal chess notation is necessary, however, I do feel the sprites quickly and easily identify the past moves. For my project I will implement both alongside each other with an option to hide the sprites.

For AI, lichess uses its own neural network chess engine called 'maia' which works off three levels (1,5,9). It was trained over 10 million games between over 1000 players. This is unusual to see from a chess site as typically chess sites used external chess engines. This could be something that I can implement with respect to my engine, though there may be an issue with an appropriately large set of training games.

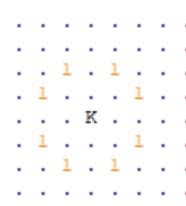
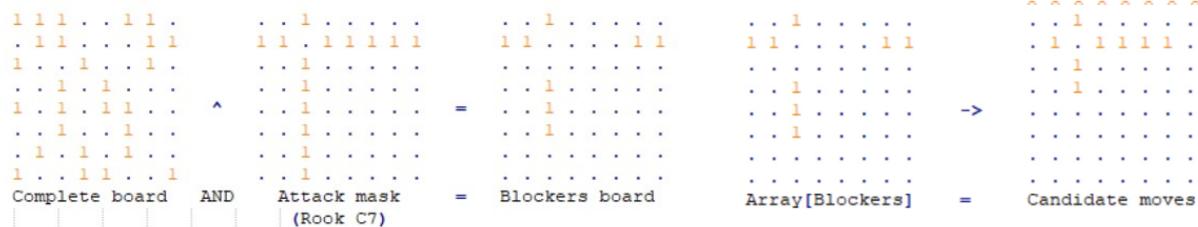
In addition to standard play lichess includes basic chess variants, such as: antichess (if you can take a piece, you must), chess960 (where pieces starting place is randomised) and king race (each player must try to get their own king to the 8th rank as fast as possible). These can both be seen below. Although my game is a chess variant it is impractical to implement this as my play space is a cube, unlike any other variant. Aside from this, Lichess offers the same sound, puzzle and learn options as Chess.com.



Regarding the puzzles, however, lichess I believe has a better design. Its puzzles can be derived from previously played games and tailored to how you play specifically (if you make a free account). Additionally, it shows the moves taken to reach the current board position which could also be helpful. The ability to create puzzles based on how a player plays is impractical for my solution as there is not a way for me to classify this. It is possible for lichess due the large database it has from all the games that have been played on their site. This makes pattern matching and finding common habits that can be recognised easy. This is not possible for me due the extensive game history needed and so I will not be implementing it.

1.5.3.1 – RESEARCH – EXISITING SOLUTIONS (AI) – STOCKFISH

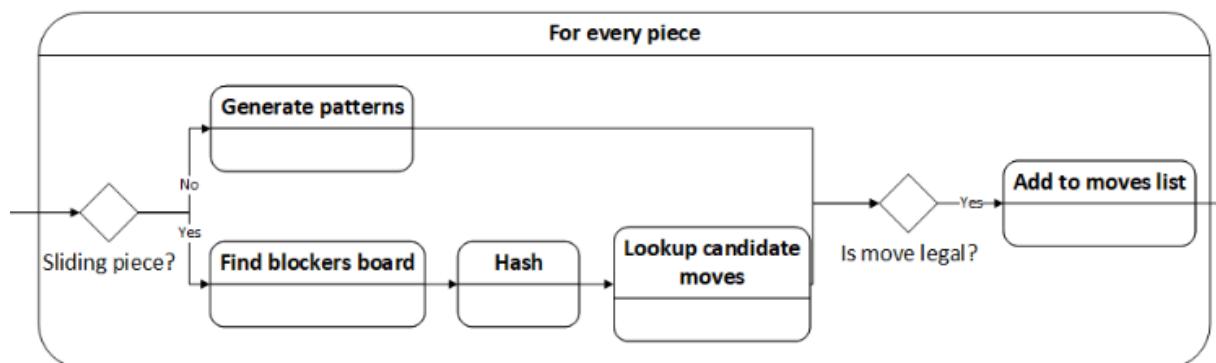
(Note ranks(A-H) are rows and files are columns (1-8)). Stockfish utilises bitboards to store the state that the board is in. The 64-bit variable can store the position of specific pieces in a game (as a board is 8x8 squares that can be occupied). A one at a specific position shows that the piece that the bitboard is representing is at that location. Bitboards start from the lowest left square and work across the board then up the ranks. (A bitboard for the starting position of pawns is shown to the side). Therefore, a move of 1 bit left would be left shift 1 and a move up one square would be a shift of 8 bits left. It is worthy to note bitboards here are stored using little-endian coding and hence empty squares are only not shown if they come after the lowest, leftist piece.



Constants are also set for pieces that have specific limited movement patterns; these are called candidate moves and are stored as bit shift operations. (Shown for a knight are shown above to the side). This representation then allows the AI to find all possible moves and starts to make a tree of pseudo-legal moves. These moves are then checked under various verifications. Firstly, the edge ranks and files are masked that the candidates that go outside the bounds of the board are determined illegal, then blocking pieces and discovered checks (king is under attack) eliminate other illegal candidates. The pseudo-legal moves can now be determined legal or illegal and the tree (for the moves with only pattern candidates) will be correct.

As for pieces that may ‘slide’ across the board, it is more complicated. The ‘rays’ that they may slide across are plotted onto a bitboard then an AND is applied to the attack rays board (Attack mask) and the complete board position (a combination of all piece bitboards) to produce a board where pieces interrupt the rays. To save computational power Stockfish indexes specific bitboard combinations and relates them to premade candidate moves. An array cannot be used, however, as this would be too large to store in memory with the board being 64 tiles (this means the candidate array will have 2^{64} elements). For this reason, a hash map is used, and the more manageable issue becomes finding an efficient hash function.

For this, magic boards are used where the bitboard of blockers is multiplied by a ‘magic number’ and then shifted by the number of bits gained to produce the hash key. This magic number is calculated by a function given the piece and its current space. All magic numbers are calculated when stockfish is first run, and it is actually done by brute force as opposed to a mathematical function. Shown below is an overview of the process discussed.



The implementation of this method of determining the board state may be harder in 3D due to the increased since and hence increased calculations involved. If I wanted to do it this way, however, I would have to change the bitboards to 512 bits as opposed to 64. Then, create more pattern bitboards and have a larger store of bitboards for sliding pieces. Additionally, I would need extra masking for the bitboards for the pattern moving pieces.

Once a tree of moves is created Stockfish must determine which move it would like to make, this is where the different levels come in to play. For Stockfish 12 and above it uses a NNUE (more on neural networks in 1.5.4) to determine the move. The inputs are Boolean as follow: (where C can be friendly or enemy) is piece A on square B with a C piece D on square E. This is iterated over for every possible piece combination on the board making up to over 40k inputs for each piece (each A – meaning without

iteration of A). Some of these can be eliminated, however, using a function that detects redundancy leaving only moves that correspond to certain criteria per piece. Stockfish also uses binary encoding reducing this down to under 800 and has the added benefit of incremental calculation through hidden layers. The advantage of an NNUE for me specifically is that it is optimised for CPU computation without the need for a GPU, allowing the AI to run on devices without a GPU. The features that benefit everyone, including myself, are the ability to use incremental calculation and the encoding for inputs.

The specific setup for Stockfish 12's neural network has three hidden layers and specific, heavy optimisation. The neural network is trained from a large data set of known moves created from Stockfish classic (below 12) where randomly generated positions are linked with the optimum subsequent moves by Stockfish classic. This is done with a very low-depth (low move lookahead – talked about later in this section) to create a large database in a reasonable time frame. Once trained to a reasonable standard, the neural network can be further trained used more complex, specific examples.

For Stockfish classic, an evaluation function is created from different chess concepts that eventually become weighted. The main concepts are material – consisting of material imbalance: total number of pieces for players, material advantage: the power of pieces in play and the power of combinations of pieces in play. The king – king safety: upcoming checks, current checks, protecting pieces, surrounding pieces. Space – positional advantage: having pieces attacking or occupying certain positions, occupying space: which squares are occupied by pieces. Threat – pieces under threat and protected, hanging pieces (under threat with no current protection), king threats, pawns or pawn line pushing, discovered threats, moves that will result in Exchange or threat, pieces not under threat but in a weak position. Strategy – pawn advantage: attacked pawns, pawns close to promotion, coupled/doubled/isolated/protected pawns, other piece advantage: blocked pieces, pieces in good positions for their move time, batteries of rooks and queens (same rank or file), enemy king under attack, rook on an open file, bishop on long open diagonals, bishop on long diagonals, bishop x-ray attacks.

There are two different ways the above must be weighted. Once for midgame and one for endgame as they are very different types of play. (There is no early play as this would simply be an opening). The weighting for each one is still evolving with chess programmers making pull requests to improve Stockfish's ELO rating. These two sections, however, are do not have a distinct start and stop point. Therefore, Stockfish interpolates these two weightings from 0-128 and with respect to the 50-move rule, that if in 50 moves no pawn has moved or piece been captured the game results in a draw. Once all this information has been weighted it is run for two looks ahead to discover any extra bonuses or penalties. Finally, the evaluation for the best move uses alpha-beta search (decreases the nodes needed to be looked at for minimax rule) which will determine the best move according to minimax rule using static evaluation (this is that to assign and sum values for aspects of the game - listed in the last paragraph). (Most chess engines use alpha-beta pruning.)

Despite Stockfish's (12) extreme optimisation of its neural network's inputs, the issues still arise in 3D chess of computational power and whether using this method would be viable. If this is an issue, I can adopt a similar but less complex scheme to Stockfish classic: creating a function from weighted parameters. In doing this I can chose more precisely how 'good' the engine is and how resource intensive it is.

1.5.3.2 – RESEARCH – EXISITING SOLUTIONS (AI) – KOMODO

Komodo and Stockfish are typically named the top two chess engines in the current day and are still ‘fighting it out’. Newer versions of Komodo are commercial (unlike stockfish), however, older versions are publicly available. Komodo uses Stockfish’s library to parse moves and positions and so will always be limited by Stockfish, however, this does mean it does have an advantage in that it doesn’t have to compute this. Komodo also has adjustable parameters for percentage of time spent on analysis and how many evaluations it makes before analysis stops.

Komodo works of programmatic chess where it will determine the best moves and most advantageous positions for both side and then use probability to choose the move to play. This is done by looking at techniques used and common chess knowledge it has learnt from previous matches. This and the fact that it attempts to use a small amount of memory creates an aggressive play style as aggressive players value simpler play, trying to shut down complex positions and in this forcing out mistakes from the opposition.

Komodo must be ‘trained’ in a specific way where it must draw and lose to weaker opponents to update its evaluation function. This is usually done through playing against other opponents. The reason that Komodo is good is that it is adaptive to new situations that arise on the board which is something than humans can do but other chess engines struggle to do.

This approach, in my opinion is not suited for this project. Komodo’s adaptive abilities make it significantly harder for a human to beat. If I were to implement this, it would mean creating a greater handicap for the engine against a player and the extra head room gained would not be needed as human 3d chess players would not surpass the limit without the addition of adaptive play.

1.5.3.3 – RESEARCH – EXISITING SOLUTIONS (AI) – KOMODO’S DRAGON

Dragon is a unique engine as it combines both programmatic techniques with the use of AI (in the form of an NNUE). In doing this dragon can analyse to a high level whilst also maintaining the adaptability that comes with being able to determine the probability of winning after a move.

Unfortunately, this is not publicly available and therefore cannot investigate the inner workings. It is worth noting that this type of complexity is unlikely to be needed as due to the size of the board in 3d chess computers will automatically be better elo for elo than in FIDE chess. I.e in a match of 3d chess between a 1500 elo chess player vs the compute power of a 1500 elo chess AI the AI has a much better chance. For this reason, the AI used for 3D chess requires less advanced chess techniques to beat a player.

1.5.3.3 – RESEARCH – EXISITING SOLUTIONS (AI) – DEEP BLUE

Deep blue was an extremely large parallel system designed to be able to create a graph/tree of all possible moves that can be made. Deep blue is a system with many components. I will be looking at deep blue for ideas to implement rather than a way to convert it for 3D chess.

After reading of the workings of deep blue, all that I saw that I would want to implement its idea of fast and slow evaluation, where in times where a simple approximation is only required the score of evaluation is based on a piece’s specific position. In slow evaluation, other key chess concepts can be involved. Not only is this an easy way to save compute power, but it also brought up another idea for me. That being different levels of AI can be created by simply removing more complex chess concepts

from my evaluation algorithm. This is a particularly good idea as different levels of AI will play like different levels of humans (as humans better at the game tend to know and understand more advanced chess concepts).

1.5.3.3.1 – RESEARCH – EXISITING SOLUTIONS (AI) – ALPHAZERO – DEEP MIND

Created by Deep mind, later brought by Google, AlphaZero is a generalised engine, not just for chess where it will take the rules of a game, train against itself and become better. It is an advancement on AlphaGo Zero, an AI to play Go given ‘zero’ extra information beyond the rules of the game. This makes AlphaZero vastly different to the likes of stockfish or Komodo that are built of human experience and more of an ‘evolutionary strategy’ where the algorithm and AI are adapted by humans to become better.

AlphaZero consists of a neural network and the MCTS (Monte Carlo Tree Search). It is worthy to note that these are now used in many other chess engines. The neural network will receive the layout of the last few boards and outputs, to the MCTS algorithm, the probability of winning with some ‘best moves’. The MCTS will then spend greater time ‘researching’ not only the best moves but also random moves in an exploration-exploitation strategy.

The genius behind AlphaZero is how it can be trained, and this is due to the MCTS still looking down paths that are given by the infant neural network. The results, percentages given by the neural network and the searching by the MCTS can then be made into new weights, which creates a better NN, creating a better MCTS. This creates a rapid feedback loop allowing the network to train extremely fast (as far as NN training goes).

1.5.3.3.2 – RESEARCH – EXISITING SOLUTIONS (AI) – ALPHAZERO – LC0

Leela Chess 0 or LC0 is a publicly available version of AlphaZero. It was written by an amateur programmer and trained over many system (and still currently being trained) by many people online who wish to. Although AlphaZero does not have an elo rating, LC0 is thought to be more powerful.

Unfortunately, it is unlikely I will ever manage to fully train a system like this to the level that LC0 and AlphaZero are capable of without the same public training scheme used for LC0. My AI would not need to be anywhere near as powerful, however, and so this could be something that I can implement in my project simply by changing the inputs nodes for ones capable of understanding a 3D board and use less previous games.

What makes this especially desirable over the previous engines is its ability to self-train. The desirability comes from the fact that I have no ‘model’ games that can be easily be given the other neural networks to train them. (More on NN in 1.5.4.1.)

1.5.4.1 - RESEARCH – MACHINE LEARNING – Q-LEARNING AND MDP

Q-Learning (modelled of MDP – Markov decision process) is a simple type of reinforcement learning and so I research for this project. It is one in which an agent takes actions in an environment for each time step and with that the environment may update to a new state and a reward may be given. As part of agents there are policies (π) which map certain states (of the environment) to probabilities of choosing a certain action from that state. Alongside this are the value functions: V value function– gives every state a value under π , Q value function– gives every action a value under π . These create q values which give the value of a state-action pair (making a particular action and a particular state). These

values are then optimised (where each state action pair yield the highest reward possible) using a reinforcement algorithm and when the optimal values are found for each state action pair (q^*) then the optimal policy for the agent to follow is also found. This reinforcement algorithm will consist usually by not always solely the following:

- Episodes - sections in which learning occurs and between episodes the environment is reset to a random state or set state
- Bellman Equation for optimality – how q values should change
- Cumulative reward and expected return – total reward for an episode and how much is expected under a policy
- Discount rate and discounted reward – gamma (how much lower future rewards are worth during calculation)
- Epsilon greedy strategy - an exploitation-exploration trade off
- Learning rate – how much old data is retained for q values

If I were to adopt this method of learning and simply try to use every single combination of how pieces can be arranged on a board as the possible states there would be an enormous state space, that combined with every possible action for those states creates an unfathomable number. Additionally, the reward system associated with MDP is not suited to this type of game as the reward would come after >100 moves.

1.5.4.2 - RESEARCH – MACHINE LEARNING – NEURAL NETWORKS

'Neural network' is a phrase that appeared numerous times in my research and appeared to be the key for how many of the chess engines worked. For this reason, I took a deeper look into them.

Neural networks work by connecting nodes to each other with weights and biases, where weights determine how much one neuron affects the one after it and biases are a threshold amount for a neuron to activate. This weight and bias layout allows neurons to be described as functions but also and more importantly allows them to compute anything. (This can be proven as specific weights and biases allow a NAND gate to be created – a universal gate.). In most cases ReLU, leaky ReLU, tanh or the sigmoid squishification function are used to modify the state of the neuron/preceptron to help with learning. The layout for a neural network consists of an input layer, hidden layers and an output layer and the activations are calculated by matrix multiplications or iterations.

What allows the neural networks to be so powerful is their ability to learn anything. (This can be proved as NAND gates can be produced by preceptrons which are universal gates.) This is done through minimising the cost function that is calculated by the mean square loss between the wanted result and the one obtained from the neural network. By finding the gradient of the cost function (through partial differentiation using the chain rule) local minima of the cost function can be found and at this point the obtained and wanted result should be extremely close. One important other addition that may be of use to the project is the use of mini batches for stochastic gradient descent, where smaller batches are trained on creating slightly more random but faster gradient descent (faster learning).

I also researched into the other ways that the 'standard' neural network can be improved:

- Convolutional neural network – convolutional layers are applied to the network
 - These layers apply a matrix of numbers across all inputs to the layer

- This process helps to identify specific patterns in the data
- Stopping overfitting to training data
 - Improving diversity of training data – hard to implement for this project
 - Skewing data - rotating morphing (not applicable for chess)
 - Gaining more raw data
 - Regularisation
 - L1 and L2 – both adaptation of cost functions to keep weights low
 - L1 – $(\lambda/n) * (\text{sum of modulus of weights})$ – means that weights are changed by a constant
 - L2 – $(\lambda/2n) * (\text{sum of square weights})$ – means that weights are changed proportionally to how large they are
 - Dropout
 - During training, half the neurons are removed randomly each epoch
 - Heuristically, this is similar to gaining an average over separately trained networks, making generalisation on the final network good.
- Improving the initial weight initialisation
 - When using sigmoid neurons if activations are near 0 or 1 the neurons become saturated and learn slowly, this is due to the nature of sigmoid function – tending towards 0 and 1 and the input tends to +- infinity
 - This problem can be somewhat tackled by initialising weights using random gaussians with mean 0 and low standard deviations (usually $1/\sqrt{\text{num inputs}}$)
- Changing the cost and activation functions
 - New cost functions
 - When using quadratic cost with sigmoid we find that when our output and desired output are far apart the learning starts slowly, speeds up then slows – this is because when finding the derivative of the cost function with respect to the activations there is a sigmoid prime term (this creates a leaning slow down as the nature of sigmoid prime is to be very low away from +-1)
 - This problem can be countered by using a cost function other than quadratic cost – namely cross entropy:

$$C = -\frac{1}{n} \sum_x \sum_j [y_j \ln a_j^L + (1 - y_j) \ln(1 - a_j^L)].$$

- When differentiating with respect to the activations or weights or biases the sigmoid prime term is cancelled allowing faster learning for 'bad' evaluation from the network. My proof can be seen below:
- Log likelihood – when used with SoftMax (talked about below) the differentiation provides similar results to cross entropy with sigmoid – this is

Cross - Entropy Cost function: $\sigma(z) = \frac{1}{1+e^{-z}}$ $\sigma'(z) = \sigma(z)(1-\sigma(z))$

$$C = -\frac{1}{n} \sum_x \sum_j [y_j \ln a_j^x + (1-y_j) \ln (1-a_j^x)]$$

$$\frac{\partial C}{\partial w_j} = -\frac{1}{n} \sum_x \sum_j \left[\frac{y_j}{\sigma(z_j)} - \frac{(1-y_j)}{1-\sigma(z_j)} \right] \cdot \frac{\partial \sigma(z_j)}{\partial w_j}$$

$$= -\frac{1}{n} \sum_x \sum_j \left[\frac{y_j}{\sigma(z_j)} - \frac{(1-y_j)}{1-\sigma(z_j)} \right] \sigma'(z_j) x_j^x$$

$$= \frac{1}{n} \sum_x \sum_j x_j^x (\sigma(z_j) - y_j)$$

σ' cancels \therefore no learning slow at high z

except the sigmoid function does not appear

- Different activation functions

- reLU – this follows $f(a) = \text{argmax}(0, a)$ – or that any values lower than 0 are 0 – it is unusual that a semi-linear function should work well for a neural network, however, in this special case it can speed up learning
- leaky reLU – this follows $f(a) = \text{argmax}(\text{const} \cdot a, a)$ where the constant is predetermined. This creates a small slope instead of all negative inputs becoming 0. This is useful as it fixes the dying reLU problem where neurons can get 'stuck' only outputting 0.
- Softmax – useful for classification (most likely not useful for this project) – it works by outputting a probability distribution and is defined by:

$$a_j^L = \frac{e^{z_j^L}}{\sum_k e^{z_k^L}}, \quad \sum_j a_j^L = \frac{\sum_j e^{z_j^L}}{\sum_k e^{z_k^L}} = 1.$$

Chess Engines that use NN are at 1.5.3.X || Discussion on if and how I will use a NN at 1.6.X

1.6.1 – RESEARCH – PRACTICALITY FOR AI – REINFORCEMENT LEARNING

If I use reinforcement learning for this level of project the training time would take an extremely long time and training data is hard to obtain as there is no 3d chess records. The best choice would be to have the AI play itself and learn that way. For this either an NNUE similar to that described for Stockfish or a deep q network would be preferable. Another option is mimic Komodo and to use a NN in conjunction with other moving parts. I have completely ruled out q learning from this project as the state-action space is far too large for 3d chess. Once trained, the network will most likely only have to perform one extra look ahead move.

1.6.2 – RESEARCH – PRACTICALITY FOR AI – FULLY CUSTOM LOGIC

This option would require more time playing to find out specific techniques and rules of thumb. The computer would then have to make more lookahead moves than if it was using a neural network as it

would not be so advanced. Heuristics can be used to reduce the number of paths looked down. These heuristics and techniques can only be achieved if there is enough playing, however.

1.6.3 – RESEARCH – PRACTICALITY FOR AI – CONCLUSION

At this time, there is no way of knowing whether tweaking the hyper variables of the neural network or playing until finding and powerful enough set of heuristics will be faster. For this reason, I have decided that starting to train neural networks will be the best. Alongside this though, I will continue to play the 2d representation I have made to still try to find a set of heuristics. This will allow me to first run a very computationally inexpensive set of instructions to remove moves and then run the remaining moves through the network. I will try using two agents to play against each other and try deep q learning too. I will judge which works best and adjust the networks according to my findings too (hyper variables, network layout, rewards type, input type). Whilst this may seem like decisions have no been made about how the AI will work, it is impossible to know what in the list above is best for a new project with testing first.

1.7 – END GOAL OBJECTIVES

1. The Board
 - a. There should be 8 boards stacked above on another.
 - i. Each board should have a full set of chess pieces.
 - ii. The pieces should be set up in the standard formation for chess.
 - iii. Each board should have a commoner in the king square, except for board 4
2. The Game
 - a. There should be player vs player game mode.
 - b. Pawns that have not yet moved can move two squares
 - c. There should be a ai vs ai mode
 - d. There should be a player vs ai mode.
 - e. For the three points above you should be able to play black or white
 - f. There should be a sound played for when pieces are moved.
 - g. There should be background music.
 - h. There should be the ability to limit the time one player can use over the whole game.
 - i. This must be able to be customised with:
 1. Total time for whole game per team. (Time must be same between teams)
 2. Time gained after making a move.
 - ii. Running out of time constitutes a loss.
 - i. Pieces must move as following:
 - i. Rook:
 1. Any amount of squares in the x or y or z dimension
 - ii. Bishop:
 1. Any amount of squares in the x and y
 2. Any amount of squares in the x and y and z
 - iii. Knight:
 1. Move 1 square in two of the (x,y,z) dimensions and move1 in the remaining 2
 - iv. Queen:

- 1. Move like the bishop
 - 2. Move like the rook
 - v. Commoner/King:
 - 1. Move 1 square in one or more of the (x,y,z) dimensions
 - j. The restrictions on piece moves are:
 - i. They may not exceed the limits of the board
 - ii. They may not leave their own king in check if it is currently so
 - iii. They may not put their own king into check
 - iv. They may not move onto a square that another piece of the same team occupies
 - v. For pieces that are not the knight: the path to the desired square cannot have pieces occupying any of the square.
 - k. No legal moves left for a team on their turn constitutes a loss if the king is in check.
 - l. If the team has no legal moves and the king is not in check then stalemate is achieved
 - m. When a piece is clicked it should display the moves it can make
 - i. Squares it can move to should be highlighted green
 - ii. Squares where it captures a piece should be highlighted red
 - n. The two checkable kings on layer 4 should be a different colour from each other and their pieces, namely blue and red.
3. User Interface
- a. Menu
 - i. The menu should contain buttons for all of the following:
 - 1. Access Settings
 - a. Opens the settings menu
 - 2. Player vs Player gameplay
 - a. Takes you to time settings and team choosing page
 - i. After time settings are chosen the boards and UI for gameplay are shown and the game begins
 - 3. Player vs AI gameplay
 - a. Takes you to the AI difficulty and team choosing page
 - i. After difficulty is chosen the board and UI gameplay are shown and the game begins
 - 4. AI vs AI gameplay
 - a. Takes you to the AI difficulty page
 - i. After difficulty is chosen the board and UI gameplay are shown and the game begins
 - 5. Load up a previously saved game
 - a. Previous save is loaded
 - i. board and ui gameplay are shown and the game begins with the position from the saved game
 - ii. the time should not be loaded – this is common practice in chess – adjournment
 - b. Game Information
 - i. The number of moves made should be displayed
 - ii. The official notation for each move should be displayed in a scrollable list

- iii. The number of each type of piece taken should be displayed
 - iv. When a team wins the game the menu screen will re appear and the winning team will be displayed at the top
 - c. Usability
 - i. There should be a button to toggle each layer between opaque and invisible
 - 1. The button will have a lower alpha (be more see-through) if the layer is toggled to be invisible
 - 2. The pieces on the respective layer should be kept as otherwise players cannot evaluate the position properly.
 - d. Settings
 - i. There should be a button in the corner of the screen to access settings at all times
 - 1. The button should be three stacked horizontal lines as this is easily recognisable as a menu/settings button
 - ii. There should be two mute buttons for sound effects and music
 - iii. There should be a save current game option which allows the current position to be stored
 - 1. Up to ten games can be stored
 - iv. There should be a return to menu option that aborts the current game and return the player to the menu screen
 - v. There should be a restart button which runs a new game with the previously applied settings
 - vi. There should be a return to game button which resumes the game that is being played.
4. AI
- a. There should be the following difficulties:
 - i. Beginner – This is designed to mimic someone who is new to the game of chess. To achieve this the below rules are set:
 - 1. It is given access to all possible moves
 - 2. Knows that taking pieces is good
 - 3. Knows that losing pieces is bad
 - 4. It has a 50% chance of having a 1 move look ahead
 - 5. 40% chance of two move look ahead
 - 6. 10% chance of making a random move
 - 7. It has a 10% blunder rate (making a purposely blunder) – this overrides all other logic
 - 8. It knows that checkmates are valuable but does not know checkmate patterns
 - 9. It knows that checks are valuable
 - 10. It knows the respective values of the pieces
 - 11. If it does not blunder it will make moves within the top 25% of its evaluation
 - 12. It does not know the principles of the opening
 - 13. It knows that its king is the most valued piece

ii. Novice – This is designed to mimic someone who plays chess to pass time and for fun but knows more about the game. To achieve this the below is set:

1. It is given access to all possible moves
2. Knows that taking pieces is good
3. Knows that losing pieces is bad
4. Will always look two moves ahead
5. It will blunder 5% of the time purposely – this overrides all other logic
6. It will make inaccuracies or mistakes 5% of the time purposely – this overrides all other logic
7. It will choose moves in the top 15% of its evaluation unless:
 - a. More than half of the top options result in negative 2 or more point of material
 - b. It will then choose randomly from the positively rated moves
8. It is aware that pieces in the centre four or protecting centre 4 squares of each layer are more valuable than others
9. It understands basic opening theory:
 - a. Get main pieces into play
 - b. Develop towards the centre (of the board and towards the enemy king)

iii. Amateur – This is designed to mimic someone who plays chess frequently and understands a lot about the workings of the game and a good proportion of theory. To achieve this the below rules are set:

1. It is given access to all possible moves
2. Knows that taking pieces is good
3. Knows that losing pieces is bad
4. Will always look two moves ahead
 - a. It will take three moves out of the top 15% of the evaluation and 50% of the time look to a lower depth – repeating this process twice (without the deeper search) (comparable to a sixth depth search – more human as humans make errors in calculations – introduces error more often than a sixth depth with error at the end)
 - b. It will then choose the best of these
5. Makes blunders 2% of the time and inaccuracies 5% of the time
6. Understands the importance of king safety
7. Has basic understanding for which squares are valuable for which pieces
 - a. Not the likes of doubled, isolated pawns
 - b. Knights near edge are bad
 - c. Rooks on open files are good
8. Understands the different stages of the game:
 - a. Opening
 - i. Develops major pieces towards the centre and towards enemy king
 - ii. Does not favour taking trading pieces

- b. Middle game
 - i. Making equal trades and keeping pieces in good positions
 - c. Endgame
 - i. Tries to create an active king and pass pawns
9. Its desire to trade pieces is proportional to the material advantage/disadvantage
- iv. Advanced – This is designed to mimic someone who takes chess seriously. Their blunders are infrequent and they have the ability to play more positionally. To achieve this the below rules are set:
1. It is given access to all possible moves
 2. Knows that taking pieces is good
 3. Knows that losing pieces is bad
 4. Will always look two moves ahead
 - a. It will take five moves out of the top 15% of the evaluation and look to a lower depth – until depth 6 reached
 - b. In addition to this the AI will add points based on the 'structural' NN which will assign points based the position of all pieces on the board at the end of the line.
 - c. It will then choose the best of these
 5. Makes blunders 0.75% of the time and inaccuracies 3% of the time
 6. Understands the importance of king safety
 7. Has basic understanding for which squares are valuable for which pieces
 - a. Including the pawn structure and position
 - i. Isolated and doubled pawns
 - b. All other positional work is included via the neural network
 8. Understands the different stages of the game:
 - a. Opening
 - i. Develops major pieces towards the centre and towards enemy king
 - ii. Does not favour taking trading pieces
 - b. Middle game
 - i. Making equal trades and keeping pieces in good positions
 - c. Endgame
 - i. Tries to create an active king and pass pawns

9. Its desire to trade pieces is proportional to the material advantage/disadvantage

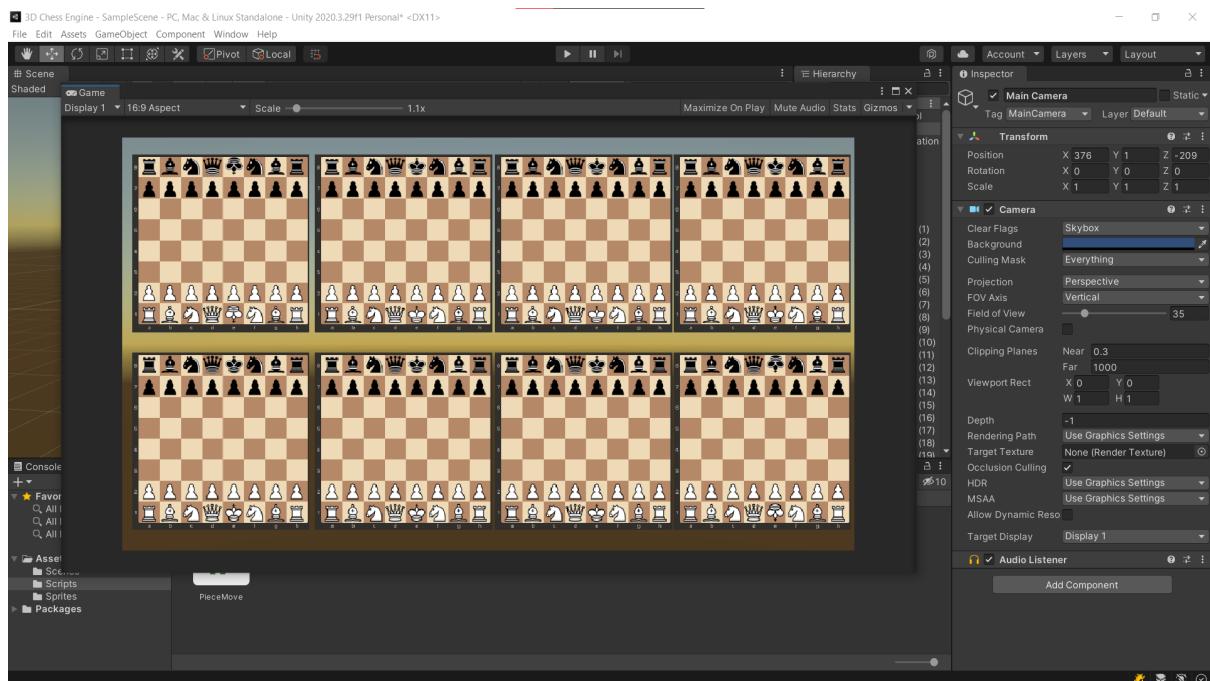
v. Master+ - mimics a GM or higher level. Achieved by:

 1. Acting as a 'Advanced' player except:
 - a. Blunder rate is 0
 - b. Inaccuracy rate is 0
 - c. Look ahead uses same approach as discussed above but taking the best line each time

As a general note for the AI. If one was comparing this to an AI used for standard chess, they may argue that it does not make sense to take moves in the top percentage of all moves as in chess if a person makes an 'blunder' (gives us a piece for free or allows checkmate etc) then even simple AIs should pick up on this. For my AI design however it is important to consider the difference that occurs due to the nature of 3d chess. It is very hard to keep track of the whole board and humans are much more likely to miss 'very obviously good moves' (i.e capturing a blundered piece or finding a checkmate). This is why I have gone with a probability amongst good moves as humans in 3d chess are likely to settle for a move they deem 'ok' or 'good enough'. This is something that was considered very carefully with the experience of playing and is not a mistake.

1.8.1.1 – COMPUTATIONAL PROTOTYPES – UI - 2D

Shown below is what one colour scheme of the 2d representation of the chess cube would look like. Note that checkable kings are denoted by being upside down kings.



Seen on the left is an example of a piece object. It shows that pieces can be moves by standard x-y transformations.

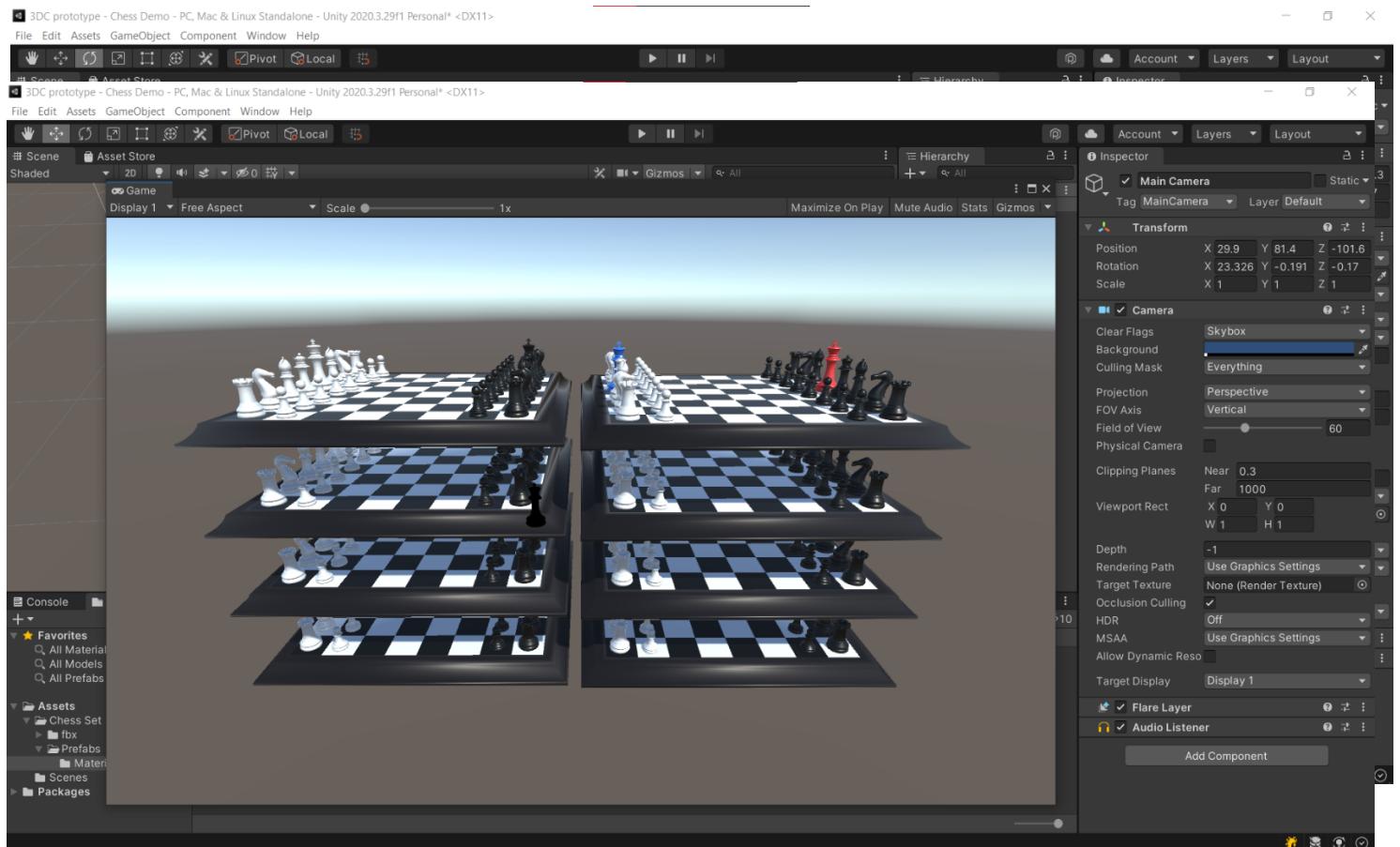


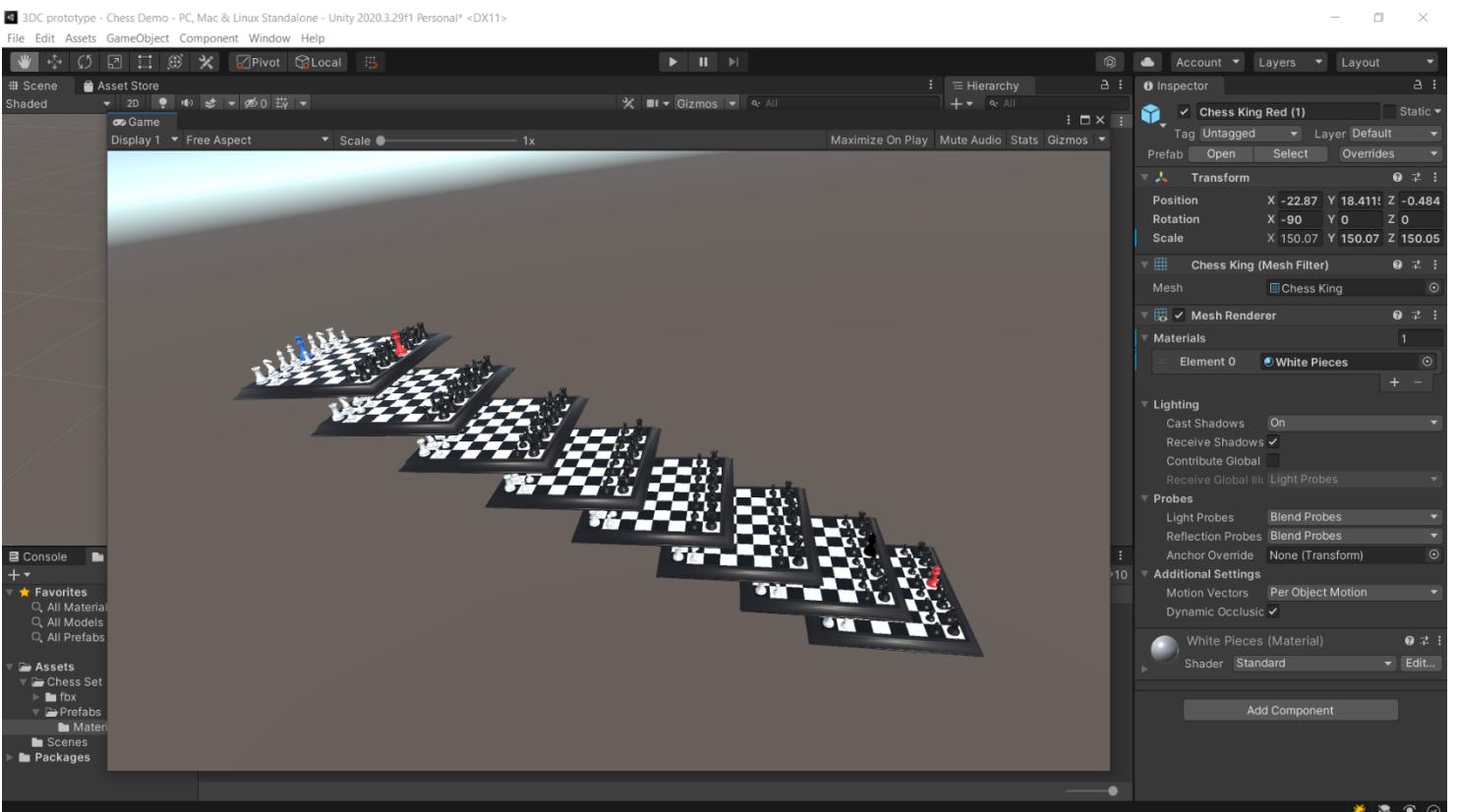
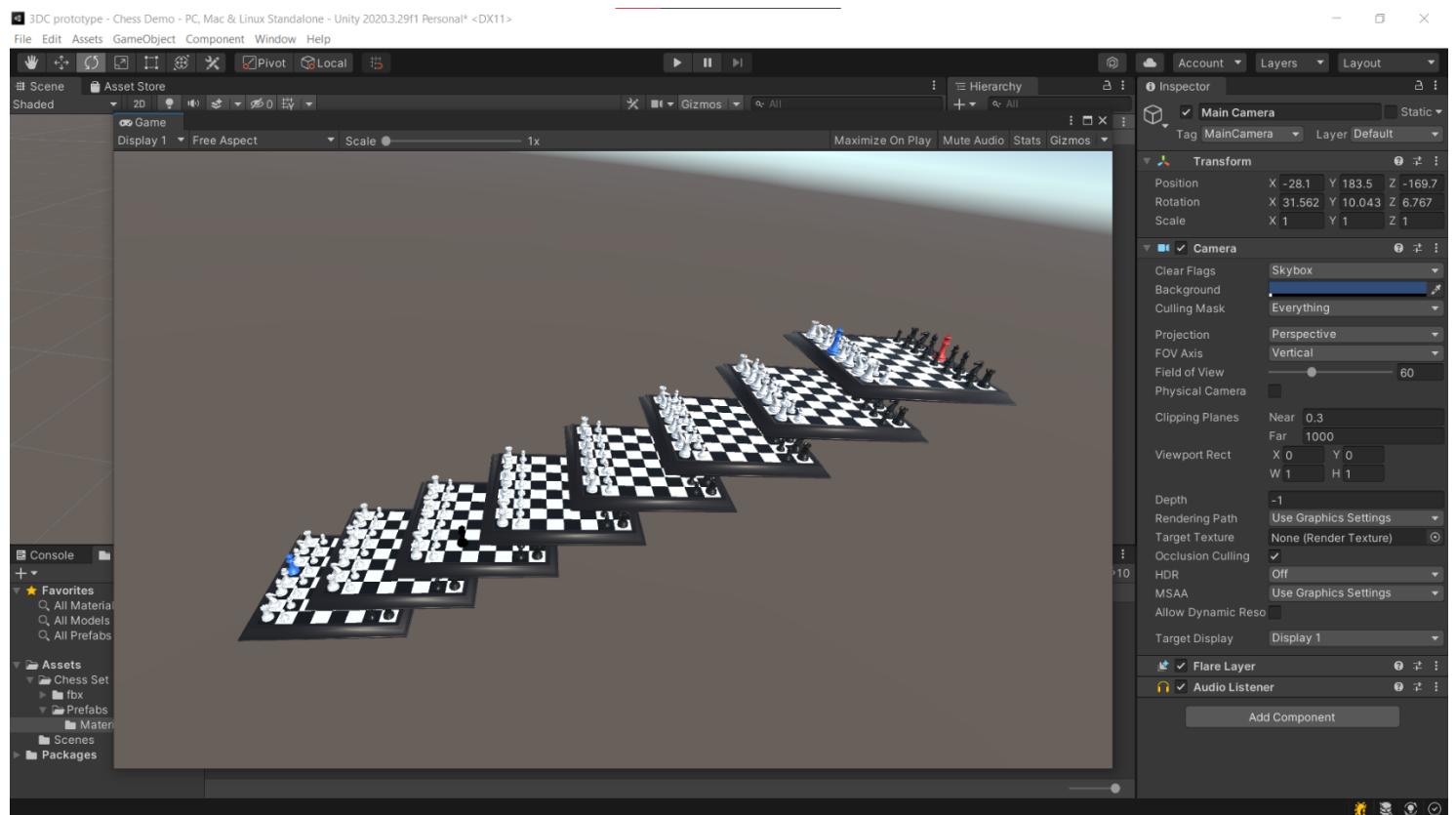
User Response: I like that you can see all boards clearly and am happy with how the checkable kings are shown. Although from looking at the checkable kings it is possible to tell which are the top and bottom boards, I would like the boards to be numbered so that it is quick and easy to tell when in gameplay.

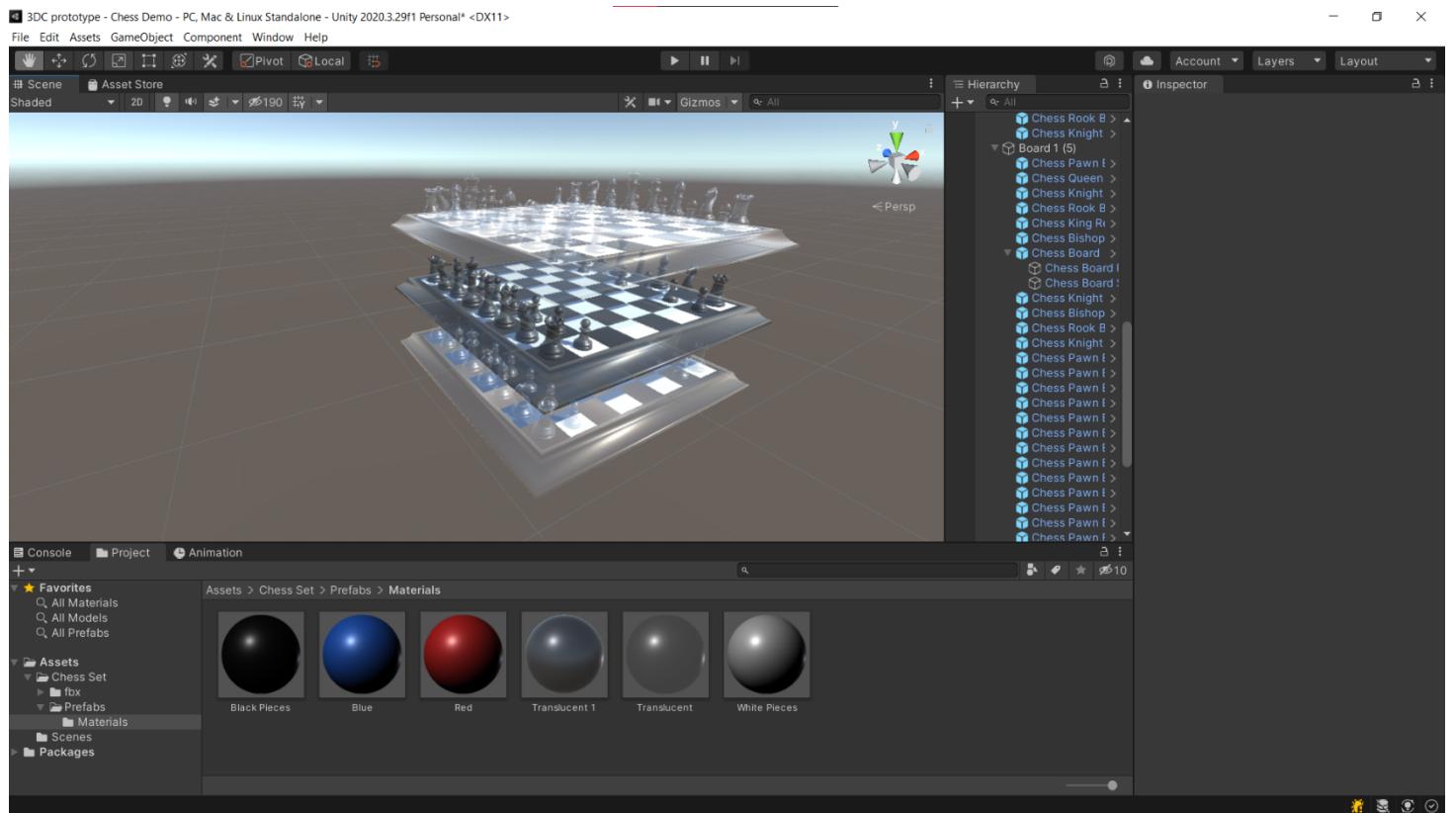
1.8.1.2 – COMPUTATIONAL PROTOTYPES – UI – 3D

Shown below are a few options for how the 3D layout may work in terms of the board:

- For all
 - Can have any layers translucent or hidden
 - Can Zoom into specific layer/s
 - Camera can be manipulated in 3-axes around the playing space
 - Can be split into different size stacks i.e. 2*4 height stacks (shown below) / 3*1 height stacks with other layers hidden – helps for visualising
- Staggered
 - Stack can be flipped to see pieces that would be hidden with stack only going one way







User Response: All the above are perfect. I think all will be good for different parts of gameplay and help people to visualise the game in the way they find the best. I think after seeing these than another helpful feature would be one to select certain board and be able to move the slightly as this would also help visualisation.

I.8.2.1 – COMPUTATIONAL PROTOTYPES – AI – CUSTOMISABLE NN

```

1  using System;
2
3  [Serializable]
4
5  class BackPropNN
6
7
8  {
9      public float[][][] NodeValues; // [layer in][number of node]
10     public float[][][] NodeBiases; // [layer in][number of node]
11     public float[][][] weights; // [layer in][node connected to][node connected from]
12
13     public float[][][] DesiredValues; // correct values for training
14     public float[][][] BiasNudges; // how much to nudge for cost
15     public float[][][] weightNudges; // "" ""
16
17     private const float ETA = 0.8f; // learning rate
18     private const float LAMBDA = 0.001f; // l2 regularisation
19     private const int MINI_BATCH = 100;
20     private const float SCALE = 0.01f; // leaky relu constant #
21     private int N;
22
23     private char a_type;
24     private char c_type;
25
26     private static Random rand = new Random();
27
28     public BackPropNN(int[] NNcomposure, char activation, char cost)
29     {
30         a_type = activation;
31         c_type = cost;
32
33         N = NNcomposure[0];
34
35         // structure format - {num input nodes, num hidden layer 1 nodes, num hidden layer 2 nodes, ..., num output nodes}
36         NodeValues = new float[NNcomposure.Length][];
37         NodeBiases = new float[NNcomposure.Length][];
38         weights = new float[NNcomposure.Length-1][][]; // no connection from output layer forwards
39
40         DesiredValues = new float[NNcomposure.Length][];
41         BiasNudges = new float[NNcomposure.Length][];
42         weightNudges = new float[NNcomposure.Length-1][][]; // "" ""
43
44         for (int i=0; i< NNcomposure.Length; i++) // adding the respective number of nodes for each layer
45         {
46             NodeValues[i] = new float[NNcomposure[i]];
47             NodeBiases[i] = new float[NNcomposure[i]];
48
49             DesiredValues[i] = new float[NNcomposure[i]];
50             BiasNudges[i] = new float[NNcomposure[i]];

```

```

51     }
52
53     for (int i = 0; i < NNcomposure.Length-1; i++) //adding the respective number of weights needed per layer
54     {
55         weights[i] = new float[NodeValues[i + 1].Length][]; // nodes to
56         weightNudges[i] = new float[NodeValues[i + 1].Length][]; //"" ""
57         for (int j=0; j < weights[i].Length; j++)
58         {
59             weights[i][j] = new float[DesiredValues[i].Length]; //nodes from
60             weights[i][j] = new float[DesiredValues[i].Length];//"" ""
61             for (int k=0; k < weights[i][j].Length; k++)
62             {
63                 // set every weight in the NN to a random value between 0 and 1
64                 // then multiply by the square root of two over the number of nodes in the layer
65                 // this improves the initial learning of the NN
66                 weights[i][j][k] = (float)(rand.NextDouble()) * MathF.Sqrt(2f / weights[i][j].Length);
67             }
68         }
69     }
70 }
71
72 1 reference
73 public float[] runNetwork(float[] inputs)
74 {
75     for (int i=0; i<NodeValues[0].Length;i++) // setting values
76     {
77         NodeValues[0][i] = inputs[i];
78     }
79
80     for (int i = 0; i < NodeValues.Length; i++)
81     {
82         for (int j = 0; j < NodeValues[i].Length; j++)
83         {
84             //calculating activations
85             NodeValues[i][j] = activation(SumForNode(NodeValues[i - 1], weights[i - 1][j]) + NodeBiases[i][j]); // sum of all weighted nodes before
86             DesiredValues[i][j] = NodeValues[i][j]; // useful for training purposes
87         }
88     }
89     return NodeValues[NodeValues.Length - 1];
90 }
91
92 1 reference
93 private static float SumForNode(float[] nodeValues, float[] weights)
94 {
95     float finalSum = 0;
96     for (int i = 0; i < nodeValues.Length; i++) // for each node in the layer
97     {
98         finalSum += nodeValues[i] * weights[i]; // multiply weight for the working on node by the last node activation
99     }
100    return finalSum;
101 }
102
103 0 references
104 public void Train(float[][] Tinputs, float[][] Toutputs)
105 {
106     int epoch = 0;
107     for (int i=0; i < Tinputs.Length; i++)
108     {
109         epoch++;
110         runNetwork(Tinputs[i]); // test the network for every set of trianing data given
111
112         for (int j=0; j < DesiredValues[DesiredValues.Length - 1].Length; j++)
113         {
114             DesiredValues[DesiredValues.Length-1][j] = Toutputs[i][j]; // adding the wanted outputs to the desired node values
115         }
116
117         for (int j=NodeValues.Length-1; j > 0; j--) // back prop up to but excluding the input layer
118         {
119             for (int k=0; k < NodeValues[j].Length; k++)
120             {
121                 var biasNudge = derivativeB(j,k);
122                 // chain rule diff for dc/db as dc/db = error delta = sigmod prime Zl * dc/dx where dc/dx = al - y
123                 BiasNudges[j][k] += biasNudge;
124                 for (int l=0; l< NodeValues[j - 1].Length; l++)
125                 {
126                     var weightNudge = derivativeW(j,l,biasNudge);
127                     weightNudges[j - 1][k][l] += weightNudge;
128
129                     var valueNudge = derivativeV(j, k, l, biasNudge); // again shown by diff - need to have wanted value for node behind to continue
130                     DesiredValues[j - 1][l] += valueNudge; // needed for calculating in previous layers
131                 }
132             }
133
134             if (epoch%MINI_BATCH == 0)
135             {
136                 for (int p = NodeValues.Length - 1; p > 0; p--) // for every layer bar the inputs
137                 {
138                     for (int j = 0; j < NodeValues[i].Length; j++) // for every node
139                     {
140                         NodeBiases[p][j] -= BiasNudges[p][j] * ETA/MINI_BATCH; // adjusting the biases
141                         BiasNudges[p][j] = 0; // resetting for more training
142
143                         DesiredValues[p][j] = 0;
144
145                         for (int k = 0; k < NodeValues[p - 1].Length; k++)
146                         {
147                             weights[p - 1][j][k] *= (1 - ETA)*LAMBDA/N;
148                             weights[p - 1][j][k] -= weightNudges[p-1][j][k] * ETA/MINI_BATCH;
149                             weightNudges[p - 1][j][k] = 0;
150                         }
151                     }
152                 }
153             }
154         }
155     }
156 }

```

```

151     }
152   }
153 }
154
155 ///////////////////////////////////////////////////////////////////
156
157 1 reference
158 private float activation(float z,float[] zs = null)
159 {
160   if (a_type == 's')
161   {
162     return sigmoid (z);
163   }
164   else if (a_type == 'm')
165   {
166     return softmax(zs,z);
167   }
168   else if (a_type == 'r')
169   {
170     return relu(z);
171   }
172   else if (a_type == 'l')
173   {
174     return leakyrelu(z);
175   }
176   else
177   {
178     return 0f;
179   }
180 }
181
182 1 reference
183 private float derivativeB(int j, int k)
184 {
185   if (c_type == 'q')
186   {
187     return MSL_SIG_B(j, k);
188   }
189   else if (c_type == 'c')
190   {
191     return CE_SIG_B(j, k);
192   }
193   else if (c_type == 'l')
194   {
195     return LL_SM_B(j,k);
196   }
197   else
198   {
199     return 0f;
200   }
201 }
202 1 reference
203 private float derivativeW(int j, int l, float bias, int k = 0)
204 {
205   if (c_type == 'q')
206   {
207     return MSL_SIG_W(j, k, l, bias);
208   }
209   else if (c_type == 'c')
210   {
211     return CE_SIG_W(j, k, bias);
212   }
213   else if (c_type == 'l')
214   {
215     return LL_SM_W(j, k,bias);
216   }
217   else
218   {
219     return 0f;
220   }
221 }
222 1 reference
223 private float derivativeV(int j, int k, int l, float bias)
224 {
225   if (c_type == 'q')
226   {
227     return MSL_SIG_V(j, k, l, bias);
228   }
229   else if (c_type == 'c')
230   {
231     return CE_SIG_V(bias);
232   }
233   else if (c_type == 'l')
234   {
235     return LL_SM_V(j, k);
236   }
237   else
238   {
239     return 0f;
240   }
241 }
242 ///////////////////////////////////////////////////////////////////
243
244 1 reference
245 private static float sigmoid(float input)
246 {
247   return 1f / (1f + (float)(Math.Exp(-input))); // sigmoid squishification function to get value 0-1
248 }
249 1 reference
250 private static float derivativeSigmoid(float input) // this is needed for the calculus involved in back prop
251 {

```

```

251     :     return input * (1 - input); // technically this should be sig(x) * (1-sig(x)) but delt with above
252 }
253
1 reference
254     private static float relu(float input) // perhaps try using this - reduces vanishing gradient - perhaps also try leaky relu
255     {
256         if (input < 0)
257         {
258             return 0;
259         }
260         return input;
261     }
262
1 reference
263     private static float leakyReLU(float input)
264     {
265         if (input < 0)
266         {
267             return input / SCALE;
268         }
269         else
270         {
271             return input;
272         }
273     }
274
0 references
275     private static float hardSigmoid(float input) // use for large data sets or lots of iterations
276     {
277         if (input < -2.5f)
278         {
279             return 0f;
280         }
281         if (input > 2.5f)
282         {
283             return 1f;
284         }
285         return 0.2f * input + 0.5f;
286     }
1 reference
287     private static float softmax(float[] layerInput, float input)
288     {
289         float sum = 0;
290         foreach( float num in layerInput)
291         {
292             sum += (float)(Math.Exp((num)));
293         }
294         return (float)(Math.Exp(input)/sum);
295     }
296
1 reference
297     private float MSL_SIG_W(int j, int k, int l, float biasNudge)
298     {
299         return NodeValues[j - 1][l] * biasNudge;
300     }

```

```

301
1 reference
302     private float MSL_SIG_B(int j, int k)
303     {
304         return derivativeSigmoid(NodeValues[j][k]) * (DesiredValues[j][k] - NodeValues[j][k]);
305     }
1 reference
306     private float MSL_SIG_V(int j, int k, int l, float biasNudge)
307     {
308         return weights[j - 1][k][l] * biasNudge;
309     }
310
1 reference
311     private float CE_SIG_W(int j, int k, float biasNudge)
312     {
313         float sum = 0;
314         foreach (float item in NodeValues[j - 1])
315         {
316             sum += item * biasNudge;
317         }
318         return sum / N;
319     }
320
1 reference
321     private float CE_SIG_B(int j, int k)
322     {
323         return (DesiredValues[j][k] - NodeValues[j][k]);
324     }
325
1 reference
326     private float CE_SIG_V(float biasNudge)
327     {
328         return biasNudge;
329     }
330
1 reference
331     private float LL_SM_W(int j, int k, float biasNudge)
332     {
333         float sum = 0;
334         foreach (float item in NodeValues[j - 1])
335         {
336             sum += item * biasNudge;
337         }
338         return sum / N;
339     }
340
1 reference
341     private float LL_SM_B(int j, int k)
342     {
343         return (DesiredValues[j][k] - NodeValues[j][k]);
344     }
345
1 reference
346     private float LL_SM_V(int j, int k)
347     {
348         return (DesiredValues[j][k] - NodeValues[j][k]);
349     }
350 }

```

Above is a customisable neural network that includes many of the features talked about in 1.6 above. Note that for efficiency when implementing this the 'activation' function will be replaced for the specific activation wanted. As the user is not knowledgeable in neural networks an interview seems inappropriate, so the below is testing for the neural network above. I have done this by adjusting the network to only use cross entropy and sigmoid neurons for efficiency. Shown below is code that I adjusted for efficiency.

```
0 references
public void Train(float[][] Tinputs, float[][] Toutputs)
{
    int epoch = 0;
    for (int i = 0; i < Tinputs.Length; i++)
    {
        epoch++;
        runNetwork(Tinputs[i]); // test the network for every set of training data given

        for (int j = 0; j < DesiredValues[DesiredValues.Length - 1].Length; j++)
        {
            DesiredValues[DesiredValues.Length - 1][j] = Toutputs[i][j]; // adding the wanted outputs to the desired node values
        }

        for (int j = NodeValues.Length - 1; j > 0; j--) // back prop up to but excluding the input layer
        {
            for (int k = 0; k < NodeValues[j].Length; k++)
            {
                var biasNudge = (DesiredValues[j][k] - NodeValues[j][k]);
                // chain rule diff for dc/db as dc/db = error delta = sigmod prime Zl * dc/dz where dc/dz = al - y
                BiasNudges[j][k] += biasNudge;
                for (int l = 0; l < NodeValues[j - 1][1]; l++)
                {
                    float sum = 0;
                    foreach (float item in NodeValues[j - 1])
                    {
                        sum += item * biasNudge;
                    }
                    var weightNudge = sum/N;
                    weightNudges[j - 1][k][l] += weightNudge;

                    var valueNudge = biasNudge; // again shown by diff - need to have wanted value for node behind to continue back prop
                    DesiredValues[j - 1][l] += valueNudge; // needed for calculating in previous layers
                }
            }
        }
    }
}

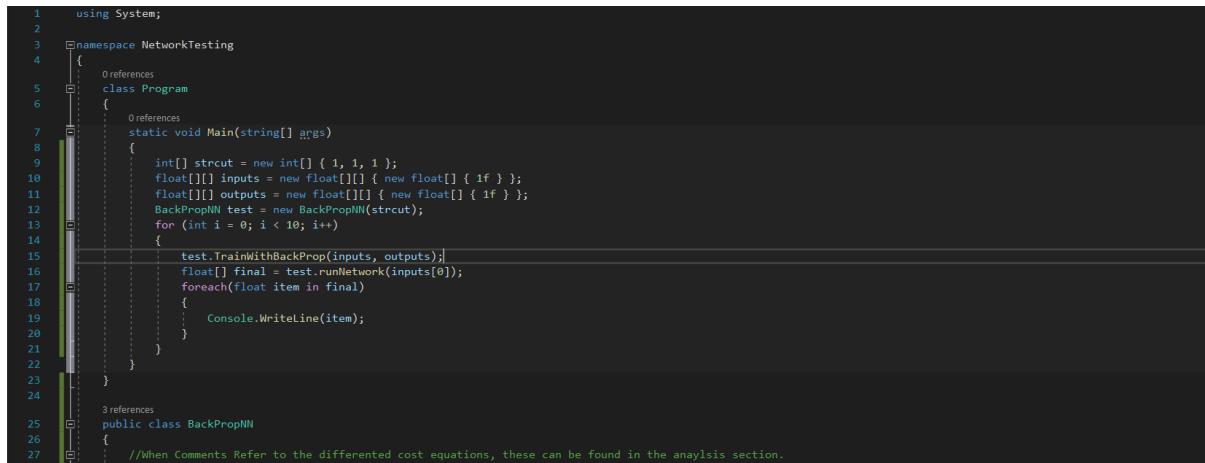
1 reference
public float[] runNetwork(float[] inputs)
{
    for (int i = 0; i < NodeValues[0].Length; i++) // setting values
    {
        NodeValues[0][i] = inputs[i];
    }

    for (int i = 0; i < NodeValues.Length; i++)
    {
        for (int j = 0; j < NodeValues[i].Length; j++)
        {
            //calculating activations
            NodeValues[i][j] = sigmoid(SumForNode(NodeValues[i - 1], weights[i - 1][j]) + NodeBiases[i][j]);
            DesiredValues[i][j] = NodeValues[i][j]; // useful for training purposes
        }
    }
    return NodeValues[NodeValues.Length - 1];
}
```

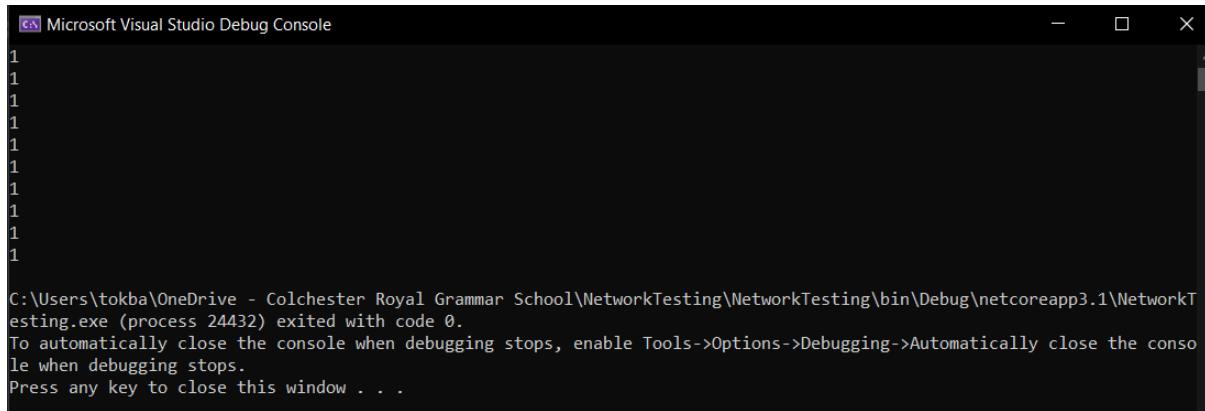
To test that is working I can run the following common testing technique.

I will test that for a single data point it can give us the desired result – I should see that the network overfits instantly.

This is only a very simply sanity check, more is discussed in design under backpropagation. This is because it should become apparent relatively quickly if there are any major issues with the networks ability to learn. Results below:



```
1  using System;
2
3  namespace NetworkTesting
4  {
5      References
6      class Program
7      {
8          References
9          static void Main(string[] args)
10         {
11             int[] strcut = new int[] { 1, 1, 1 };
12             float[][] inputs = new float[][] { new float[] { 1f } };
13             float[][] outputs = new float[][] { new float[] { 1f } };
14             BackPropNN test = new BackPropNN(strcut);
15             for (int i = 0; i < 10; i++)
16             {
17                 test.TrainWithBackProp(inputs, outputs);
18                 float[] final = test.runNetwork(inputs[0]);
19                 foreach (float item in final)
20                 {
21                     Console.WriteLine(item);
22                 }
23             }
24         }
25     References
26     public class BackPropNN
27     {
28         //When Comments Refer to the differentiated cost equations, these can be found in the analysis section.
29     }
30 }
```



```
1
1
1
1
1
1
1
1
1
1
C:\Users\tokba\OneDrive - Colchester Royal Grammar School\NetworkTesting\NetworkTesting\bin\Debug\netcoreapp3.1\NetworkTesting.exe (process 24432) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

We see that the above is successful with the network learning after the first pass (we do not see a random number first as learning occurs before the running of the network. For the next test these will be swapped so we expect to see a random number from the gaussian distribution on the first pass. Additionally, I will change the expected output for greater sanity checking.



```
1  using System;
2
3  namespace NetworkTesting
4  {
5      References
6      class Program
7      {
8          References
9          static void Main(string[] args)
10         {
11             int[] strcut = new int[] { 1, 1, 1 };
12             float[][] inputs = new float[][] { new float[] { 1f } };
13             float[][] outputs = new float[][] { new float[] { 0f } };
14             BackPropNN test = new BackPropNN(strcut);
15             for (int i = 0; i < 10; i++)
16             {
17                 float[] final = test.runNetwork(inputs[0]);
18                 foreach (float item in final)
19                 {
20                     Console.WriteLine(item);
21                 }
22             }
23         }
24     References
25     public class BackPropNN
26     {
27         //When Comments Refer to the differentiated cost equations, these can be found in the analysis section.
28     }
29 }
```

```

0.32981
0
0
0
0
0
0
0
0
0
0
0
0
0
C:\Users\tokba\OneDrive - Colchester Royal Grammar School\NetworkTesting\NetworkTesting\bin\Debug\netcoreapp3.1\NetworkTesting.exe (process 23604) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .

```

Below shows the hyperparameters used for this testing (they are set so that the network is as basic as possible. Also shown is the Gaussian distribution for the weights that give the initial randomness.

```

for (int i = 0; i < NNcomposure.Length; i++) //adding the respective number of nodes for each layer
{
    NodeValues[i] = new float[NNcomposure[i]];
    NodeBiases[i] = new float[NNcomposure[i]];

    DesiredValues[i] = new float[NNcomposure[i]];
    BiasNudges[i] = new float[NNcomposure[i]];
}

for (int i = 0; i < NNcomposure.Length - 1; i++) //adding the respective number of weights needed per layer
{
    weights[i] = new float[NodeValues[i + 1].Length]; // nodes to
    weightNudges[i] = new float[NodeValues[i + 1].Length[]]; //"" ""
    for (int j = 0; j < weights[i].Length; j++)
    {
        weights[i][j] = new float[DesiredValues[i].Length]; //nodes from
        weights[i][j] = new float[DesiredValues[i].Length];//"" ""
        for (int k = 0; k < weights[i][j].Length; k++)
        {
            // set every weight in the NN to a random value between 0 and 1
            // then multiply by the square root of two over the number of nodes in the layer
            // this distribution improves the initial learning of the NN
            weights[i][j][k] = (float)(rand.NextDouble()) * MathF.Sqrt(2f / weights[i][j].Length);
        }
    }
}

```

```

//When Comments Refer to the differented cost equations, these can be found in the analysis section.
//The cost function used in conjunction with tanh is -0.5 * ( (1-y)*log(1-a) + (1+y)*log(1+a) ) + log(2) // (cross-entropy-esque)
//this gives derives to functions that are proportional to the error
//namely, dc/dw = x(a-y) and dc/db or dv = a-y

public float[][] NodeValues; // [layer in][number of node]
public float[][] NodeBiases; // [layer in][number of node]
public float[][][] weights; // [layer in][node connected to][node connected from]

public float[][] DesiredValues; // correct values for training
public float[][] BiasNudges; // how much to nudge for cost
public float[][][] weightNudges; // "" ""

private const float ETA = 1f; // learning rate
private const float LAMBDA = 0f; // l2 regularisation
private const int MINI_BATCH = 1; // mini batch size for epoch based training
private const float SCALE = 0.01f; // leaky relu constant
private int N_x;

private char a_type;
private char c_type;

private static Random rand = new Random();

1 reference
public BackPropNN(int[] NNcomposure, char activation = 's', char cost = 'm')
{
    a_type = activation;
}

```

2.0 DESIGN

2.1- OVERALL DESIGN DISCUSSION

1. The Board
 - a. Each Board will be a 3rd party blender object set as its own game object that looks like a chess board
 - i. Each piece will be a 3rd party blender object set as its own custom object – all will inherit from a ‘piece’ class.
 - ii. When a game is started each type of piece will be instantiated for every occurrence it has over the boards and placed in the correct position upon its instantiation. This is done via a scriptable object ‘board layout’ which depicts what object should be placed and where.
 - iii. The pieces in the king square on board 4 will be a different colour from the rest and the other king
2. The Game
 - a. On start white can make a move by pressing on a piece to select it and then selecting the square for it to move to. If this move is legal the process will happen for black. If not whites turn will continue until a legal move is made. This process stops when either a player’s time runs out, checkmate is achieved or stalemate occurs from no legal moves.
 - b. There will be a Boolean for pawns to state whether they have moved or not – if they have not yet moved , they will have an extra move added to their move list that is to move two forward.
 - c. The AI of selected level will make a move for white then black until the game ends.
 - d. The player will make moves as in part A for their chosen side and the ai will make moves for the other team.
 - e. One of the attributes in the chess player class will be a Boolean enumerator for AI or Player.
 - f. There will be an audio source component for a ‘move sound’ object it will be activated by a function each time that a move is made.
 - g. There will be an audio source component for a ‘background sound’ object that will loop and start when the game starts through the inbuilt unity start function.
 - h. There will be two clocks and a clock function that pauses the clock for one side and starts the clock for the opponent each time a move is made.
 - i. This must be able to be customised with:
 1. Once the game mode has been chosen the time options as three boxes will appear and any valid time in hours and minutes can be entered into the first two. (Valid times are from 1 minute to 23 hours and 60 minutes)
 2. Gained time after move will be the other text box and can be any amount of time from 1 second to 60 seconds.
 - ii. If the time runs out for one team then the other team will win the game.

- i. Each Piece will have its own class that inherits from a piece class. The piece class determines where a piece can move with legal moves function. Each piece's own class will have an override function that will create the legal move for that piece type.
 - i. Rook:
 - 1. Has an array of vector3int that contains the direction vectors of + and - in each of the x,y and z dimensions
 - 2. The function will:
 - a. For every direction vector
 - i. Iterate over the size of the board.
 - 1. Get coordinates of the position the piece is currently in + the iteration number*direction vector
 - 2. If this is outside the bound of the board it will break from the iteration
 - 3. if iterate not broken It will get the piece on those coordinates
 - 4. If there is no piece, there it will call the try to add move function which will determine whether it puts the king into check or leaves it in check and if it does not then it will add the coords to the legal moves.
 - 5. If the piece is from the same team, it will break the iteration.
 - 6. If it is from the other team, it will call the try to add move function which will determine whether it puts the king into check or leaves it in check and if it does not then it will add the coords to the legal moves
 - ii. Bishop:
 - 1. Implemented the same as the rook except directions vectors are: +/- combinations for the x and y with each of (0,-,+z) directions.
 - iii. Knight:
 - 1. Implemented the same as the rook except:
 - a. It does not iterate over the board size
 - b. The direction vectors are 1 in two of (x,y,z) and 2 in the other of the three dimensions.
 - iv. Queen:
 - 1. Implemented the same as the rook but with the addition of the bishop vectors to the directions.
 - v. Commoner/King:
 - 1. Implemented the same as the rook except:
 - 2. It does not iterate over the boardsize and the direction vector are any combination of 1,0,-1 in the xyz dimensions, except 0,0,0
 - j. [All of the legal move points are covered in the piece move specifics above]

- k. Check can be checked by the attack on piece of type<T> function with T as the King piece. Which will take the coordinates of the square of the king and check to see if those coordinates are in any of the legal moves for the opposing team. No legal moves can be checked by the chessplayer class which keeps track of all legal moves each turn. Team wins is then displayed.
 - l. Check can be checked by the attack on piece of type<T> function with T as the King piece. No legal moves can be checked by the chessplayer class which keeps track of all legal moves each turn. (Repeated point as same implementation). Draw is then displayed.
 - m. When a piece is clicked it is received from the collider of the boards. Then the function get coords from click is will be called which will use the unity hight of boards and size of squares to work out the coordinates of the piece. (This is done by having an object in the bottom left of all of the board and registering how far in each dimension the click was from said object). The piece in that square can be found from get piece from coords which will be under the board class (this is as the board class will have 3d array of pieces that represent the board) and set as the 'selected piece'. The available moves from that piece can then be found from that piece's class's available moves function.
 - i. All available coords are converted to board positions and a green square unity object will be placed around those positions.
 - ii. Each square will be checked to see if it has a piece occupying it (you do not need to check the team as a piece cannot move to a square that contains a piece of the same team).
 - 1. If it does the material of the object will be changed to red.
 - ii. If the piece is clicked again, it will no longer be the 'selected piece'
 - iv. If a place is clicked and the coords given do not return and piece from get piece from coords (or it is out of bounds) then the piece is no longer the 'selected piece'
 - n. The material of the king pieces can be changed to blue and red materials.
3. User Interface
- a. Menu
 - i. The menu will be a UI panel containing.
 - 1. Access Settings
 - a. Sets the settings UI panel as active. (settings panel has higher priority (on top)).
 - 2. Player vs Player gameplay
 - a. Takes you to time settings and team choosing page which is a child panel with buttons for choosing team and text boxes for time
 - i. After time settings are chosen the boards and UI for gameplay are shown, the restart game function with the settings chosen is called and the menu panel is hidden
 - 3. Player vs AI gameplay

- a. Takes you to the AI difficulty and team choosing page which is a child panel where the difficulty is buttons and team choosing is buttons. This changes the AI settings and starting parameters from white chess player and black chess player objects
 - i. After difficulty and team is chosen the boards and UI for gameplay are shown, the restart game function with the settings chosen is called and the menu panel is hidden
- 4. AI vs AI gameplay
 - a. Takes you to the AI difficulty page which is a panel with difficulty buttons that change the AI settings
 - i. After difficulty is chosen the boards and UI for gameplay are shown, the restart game function with the settings chosen is called and the menu panel is hidden
 - 5. Saving a game will work by storing all of the moves made in a game. Each move is 6 numbers that represent what coords the piece started at and where they ended at.
 - a. Saved Game
 - i. When the game is loaded all moves are played up to that position.
 - ii. Time should not be loaded
- b. Game Information
 - i. The number of moves can be displayed on screen with a small text box in the corner of the screen with the moves made being stored as an int variable in the chess controller.
 - ii. The official notation function will take in the coordinates of the piece move, the piece type and the location to be moved to. It will condense and convert this to the correct format and display via a scrollable list on the side of the screen.
 - iii. On the opposing side of the screen each piece in both colours will be displayed as images. Next to each image will be a number that represents how many piece have been taken. It is set to 0 on game restart. Each time a piece is taken its team and type is taken note of and the corresponding number on screen will be increased.
 - iv. When a team wins the game the menu panel will be displayed however the title at the top will change to 'white wins', 'black wins' or 'draw'.
- c. Usability
 - i. There will be 8 buttons that correspond to the 8 layers. When one is clicked its respective layer will be have its material changed to 0 alpha and when clicked again changed to its original alpha.
 - 1. The button will have a lower alpha when a layer is invisible by changing the alpha of the button's image component.
 - 2. The pieces will stay as they are with this implementation

d. Settings

- i. The settings button can be implemented using a unity button
 - 1. The button's image can be set the iconic 'three lines' image
- ii. This can be done using two buttons that include mute image accompanied with text to the side saying 'sound effects' and 'background'
- iii. The save game implementation was discussed earlier
- iv. The menu panel will be re displayed if the 'return to menu' button is clicked
- v. This restart button will run the restart function in the chess game controller without changing any of the game settings.
- vi. The return button will hide the settings panel.

4. AI

a. There should be the following difficulties:

- i. Beginner – This is designed to mimic someone who is new to the game of chess. To achieve this the below rules are set:
 - 1. Can see all of the available moves under the activeplayer object (chessplayer class)
 - 2. Gives positive number reward is it calculates that an opponent's piece piece is taken.
 - 3. Gives negative number reward is it calculates that its own piece is taken.
 - 4. 1/2 random number decides whether it will stop after one move look ahead.
 - 5. If it does not stop it has a 4/5 random number decide to look ahead another move.
 - 6. If not, it make a random move
 - 7. Before all other calculation it will has a random 1/10 random number decide whether it will make a move in the bottom 10% of moves (with one move look ahead).
 - 8. Since it does not know checkmate patterns it simply assigns extra numerical reward if a move puts the king in check.
 - 9. assigns extra numerical reward if a move puts the king in check.
 - 10. Higher numerical rewards is given for the higher valued pieces (pawn:1, knight:3, bishop:3, rook:5, queen:9, commoner:4)
 - 11. If it does not blunder it will make moves within the top 25% of its evaluation
 - 12. Opens by the rules set above (no separate opening strategy)
 - 13. If opponent can achieve check it will assign lower numerical value.
- ii. Novice – This is designed to mimic someone who plays chess to pass time and for fun but knows more about the game. To achieve this the below rules are set:
 - 1. Can see all of the available moves under the activeplayer object (chessplayer class)
 - 2. Gives positive number reward is it calculates that an opponent's piece piece is taken.

3. Gives negative number reward if it calculates that its own piece is taken.
 4. Will always look two moves ahead
 5. It will make moves in the bottom 15% of its evaluation 15% of the time – this overrides all other logic
 6. It will make moves in the bottom 50-75% of its evaluation 5% of the time – this overrides all other logic except blundering.
 7. It will choose moves in the top 15% of its evaluation unless:
 - a. More than half of the top options result in negative 2 or more point of material
 - b. It will then choose randomly from the positively rated moves
 8. If a piece protects/attacks or occupies the centre of the board a small numerical value is added to the reward
 9. It understands basic opening theory:
 - a. Small extra numerical reward is given for moving piece off starting square.
 - b. Developing towards the centre is included in point 10.
- iii. Amateur – This is designed to mimic someone who plays chess frequently and understands a lot about the workings of the game and a good proportion of theory. To achieve this the below rules are set:
1. Can see all of the available moves under the activeplayer object (chessplayer class)
 2. Gives positive number reward if it calculates that an opponent's piece is taken.
 3. Gives negative number reward if it calculates that its own piece is taken.
 4. Will always look two moves ahead.
 - a. It will take three moves out of the top 5% of the evaluation and 50% of the time look to a lower depth – 3x more for the comparable to 6 depth search for specific lines. The percentages are done via a random number generator.
 - b. It will then choose the best of these
 - c. Makes moves in the bottom 15% of evaluation 2% of the time and moves in the 50-75% of its evaluation 5% of the time. The 'of the time' percentages are done via a random number generator.
 5. Assigns extra negative value if a move allows for more types of attack on its own king. Vice versa for the enemy's king.
 6. Has basic understanding for which squares are valuable for which pieces
 - a. If a line of moves results in isolated or doubled pawns it assigns additional negative reward to that line.
 - b. Small additional negative reward is given if a knight is on the 1st or last file

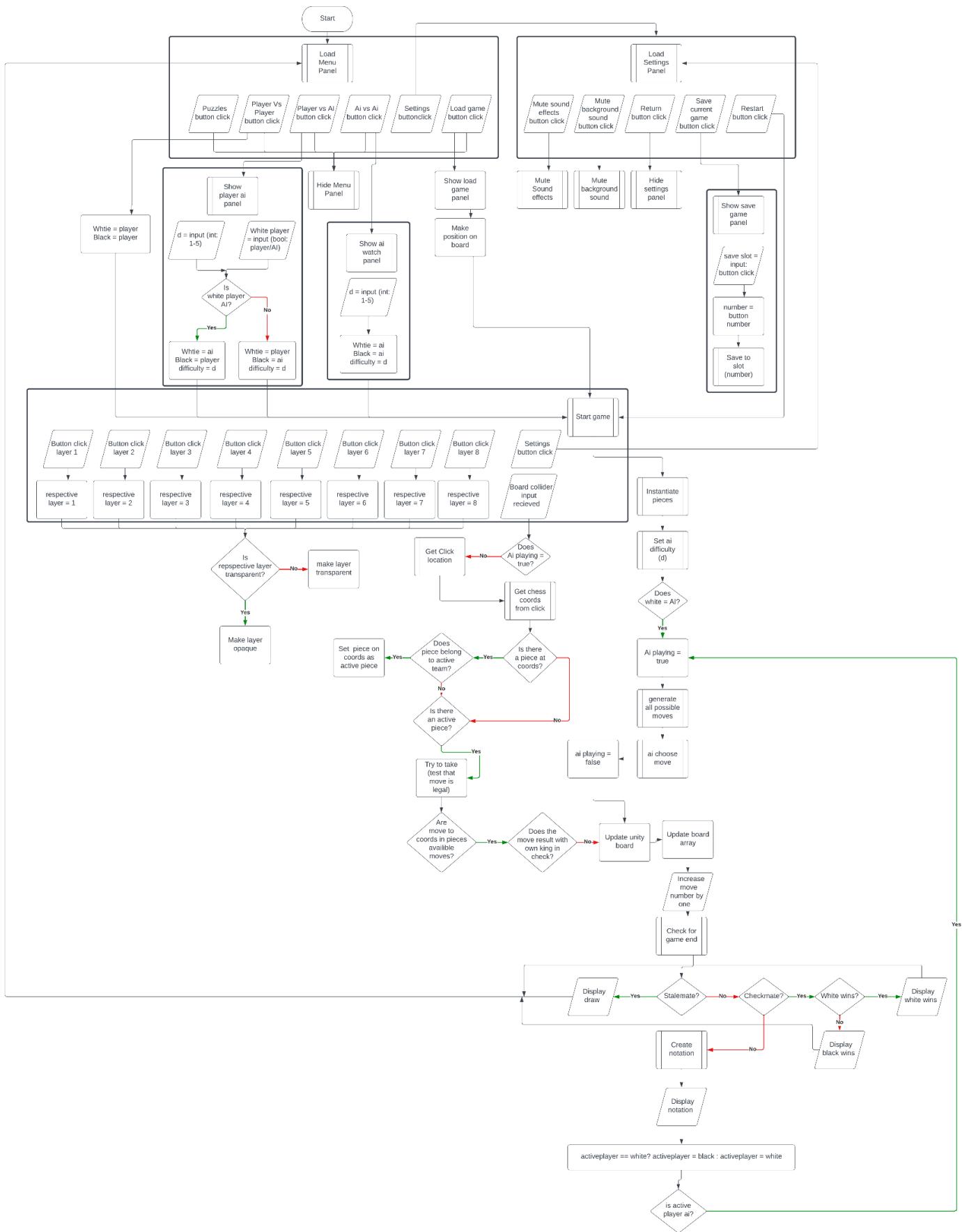
- c. Additional positive value is given if a rook is moved to a file that is empty.
7. Understands the different stages of the game:
- a. Opening – this occurs until more than 50 major pieces have been moved
 - i. Developing towards the centre is including in the point above and control over the centre
 - ii. During the opening stage it will apply small negative reward if pieces are traded (this stack for the number of pieces taken).
 - b. Middle game – occurs till 50 major pieces have been taken
 - i. Equal trades are now given positive reward.
 - c. Endgame – the last section of the game after the middle game ends
 - i. Small positive reward is given for getting the king towards the enemy's pawns that are closest the promotion. Small positive reward is given for moving a pawn forwards.
8. The more material that a team is down by the greater negative reward will be given for a line that includes one of its own pieces being taken. The inverse is true for a team that is up in material.
- iv. Advanced – This is designed to mimic someone who takes chess seriously. Their blunders are infrequent and they have the ability to play more positionally. To achieve this the below rules are set:
1. Can see all of the available moves under the activeplayer object (chessplayer class)
 2. Gives positive number reward if it calculates that an opponent's piece piece is taken.
 3. Gives negative number reward if it calculates that its own piece is taken.
 4. Will always look two moves ahead
 - a. It will take five moves out of the top 15% of the evaluation and look to a lower depth – 3x more for the comparable to 6 depth search for specific lines. The percentages are done via a random number generator
 - b. In addition to this the AI will add additional points based on the 'structural' neural network which will assign points based the position of all pieces on the board at the end of the line.
 - The network will use the backpropagation algorithm to run: the cost, activation functions, and hyper parameters such as the learning rate and minibatch size must be decided through extensive training with extensive training data.
- i. The training will work as follows:
1. I will create an algorithm that designs positions that favour one team by a certain

amount. This will be achieved by randomly assigning each team a random amount of the following (material must be same for both teams).

- a. Pawns protecting each other. +0.1
 - b. Isolated pawns -0.1
 - c. Doubled pawns -0.2
 - d. Knight on edge of board -0.2
 - e. Knight with all moves in bound of the board +0.2
 - f. Bishop pair +0.3
 - g. Bishop on same line as queen +0.1
 - h. Rook on open file +0.3
 - i. Any piece off starting square +0.1
 - j. Any piece attacking a square that touches the king +0.4
 - k. Any piece achieving a pin on king +0.1*the value of the pinned piece
 - l. Any piece achieving pin on the queen +0.4
 - m. Any piece controlling or on the centre 4 squares of any level +0.2
 - n. None of the above +0
2. The Board position is then passed to the network as the input and the value of all of whites points from above – blacks points from the above is the expected output from the neural network.
 - c. It will then choose the best of these lines
 5. Makes moves in the bottom 15% of evaluation 0.75% of the time and moves in the 50-75% of its evaluation 3% of the time – overrides everything else.
 6. King safety covered by neural network
 7. Valuable squares implementation covered by neural network
 8. Stages of the game implementation have already been discussed.
 9. Trading pieces desire implementation already discussed.
- v. Master+ - mimics a GM or higher level. Achieved by:
1. Acting as a 'Advanced' player except:
 - a. Best move in strategy played – no blunders
 - b. Best move in strategy played – no inaccuracies
 - c. The 6 move look ahead implementation can be done using the same functions as before but without taking top 15% and instead the best move

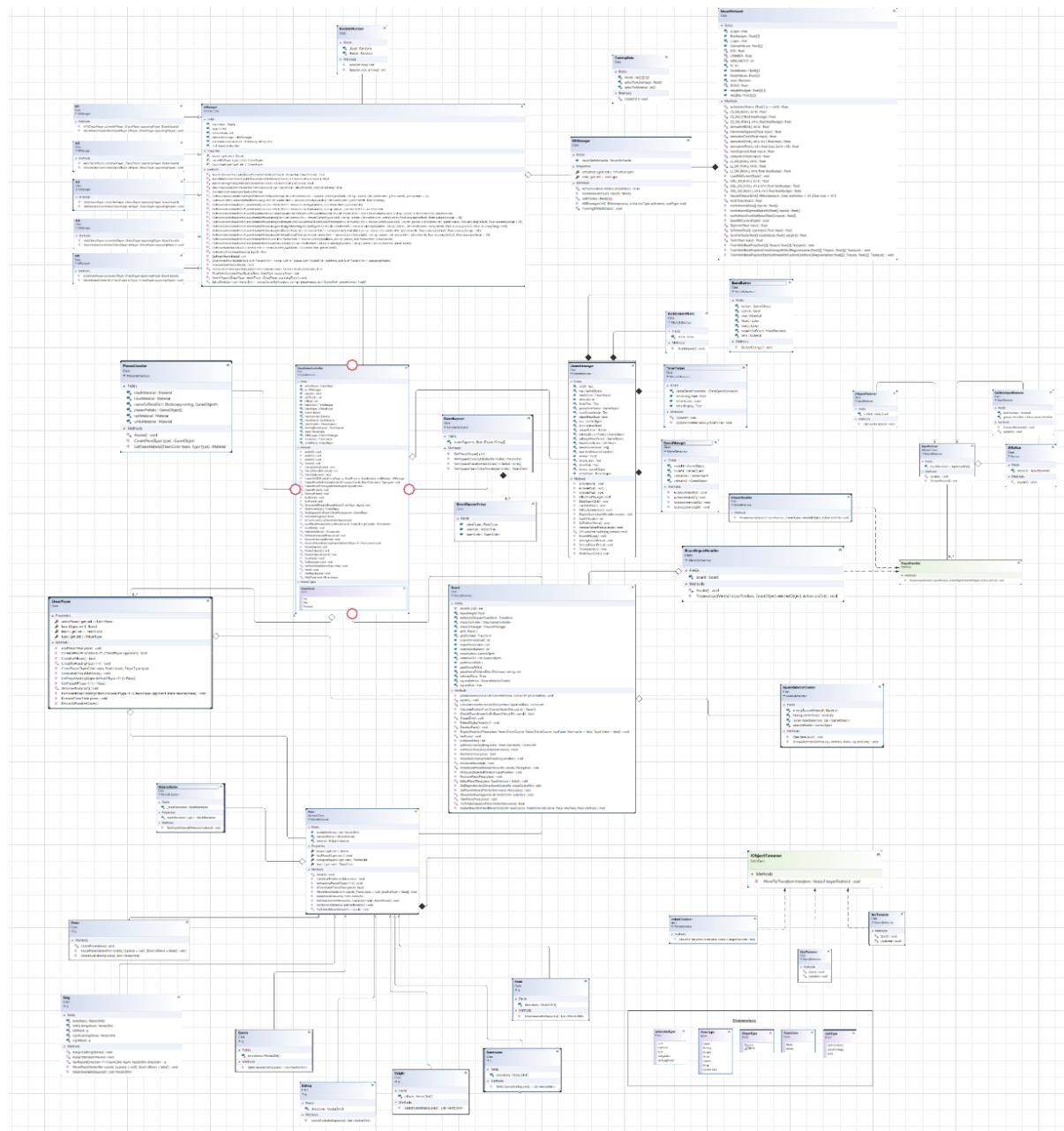
2.2 – LOGIC CHART

[note that boxed off areas means that inputs inside may only occur while the panel (or lack thereof for being in gameplay) are active]



2.3 - CLASS DIAGRAM

Below is an image of my UML Diagram: It is unreadable here due to the size. Provided below is also the link and QR Code to Access my UML diagram. Using the link/QR code you should be able to zoom in and out to see the full extent of the diagram and all the details such as functions, whether are private/public/protected, variables and their types. The symbols are as follows: blue bricks for varialbles/fields, pink cubes for functions/procedures, spanners for constructors, a lock for private, absence of a lock for public, a star for protected, italics for abstract, words following a colon give field or return type. You may also notice that the ability to move the camera does not appear here where it is integral to do so throughout the game. For this I am using third party code and the code does not interact with any of my code so has been excluded. Refer to 'CameraFreeLook' in the technical solution for this code.





https://lucid.app/lucidchart/00e80949-b8ce-4d34-9f60-323fef47eba1/edit?viewport_loc=-4079%2C-617%2C14400%2C7487%2CHWEP-vi-RSFO&invitationId=inv_bab933ee-3c0c-4431-a724-e8ff207fb1a2

(You may need a free LudidCharts Account to access)

2.4.1.1 - ALGORITHMS – NEURAL NETWORK - INTRO

The back propagation algorithm is the backbone of the machine learning implementation in this solution. Which, in turn is what allows the top AIs used in this program to not only act more like human but also become more advanced. The base for the algorithm that will be used is seen in the prototype in analysis. Here is a break down of the working and how it will be implemented in the final project.

Although this project is not strictly a multiclassification problem, in terms of the neural network topology it may be treated as one. This is because the purpose of the neural network is to evaluate the board positionally and in turn give an arbitrary numerical rating to the position. Therefore, the likes of a cross-entropy-esque cost function will be preferable here. As for the activation function this is something that will have to be decided upon during further neural network specific testing (to be spoken about later in this section). The likely candidates will be tanh and sigmoid as they couple well with the cross-entropy and likewise cost functions.

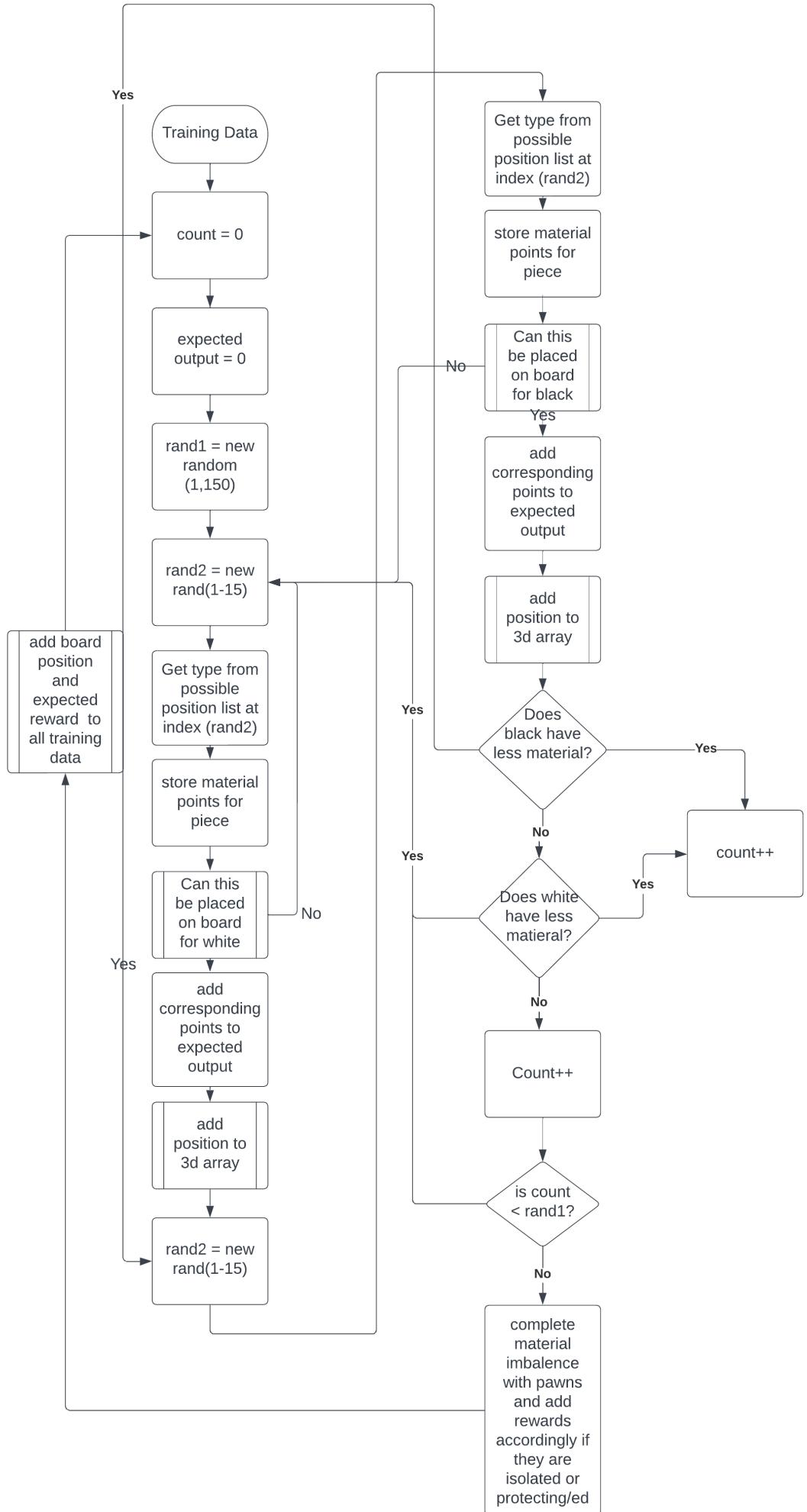
2.4.1.2 - ALGORITHMS – NEURAL NETWORK – TRAINING DATA

The creation of training data is the first part, and an outline of the algorithm can be seen above. It could be argued that instead of a neural network, this algorithm could be reversed engineered to give a structural evaluation. The goal however is for the neural network to find patterns in these positions deeper than with the basic rules outlined above. That is what makes the machine learning approach here more favourable.

In the algorithm above the “can this be placed on board for...” function checks whether for the position wanted whether it can be placed anywhere on the board. The easiest and most practical implementation of this is via brute force, as keeping track of unused squares for each situation is very more time consuming to implement and as this algorithm does not need to produce training data at immense speed it is more practical for me to have the algorithm set up this way. This is because keeping track of all unused squares for each square is much less memory efficient (with the enormous number of combinations) than brute force. This allows me to run the algorithm ‘on the side’ while I may use my computer simultaneously without the fear of running out of memory.

It i

is



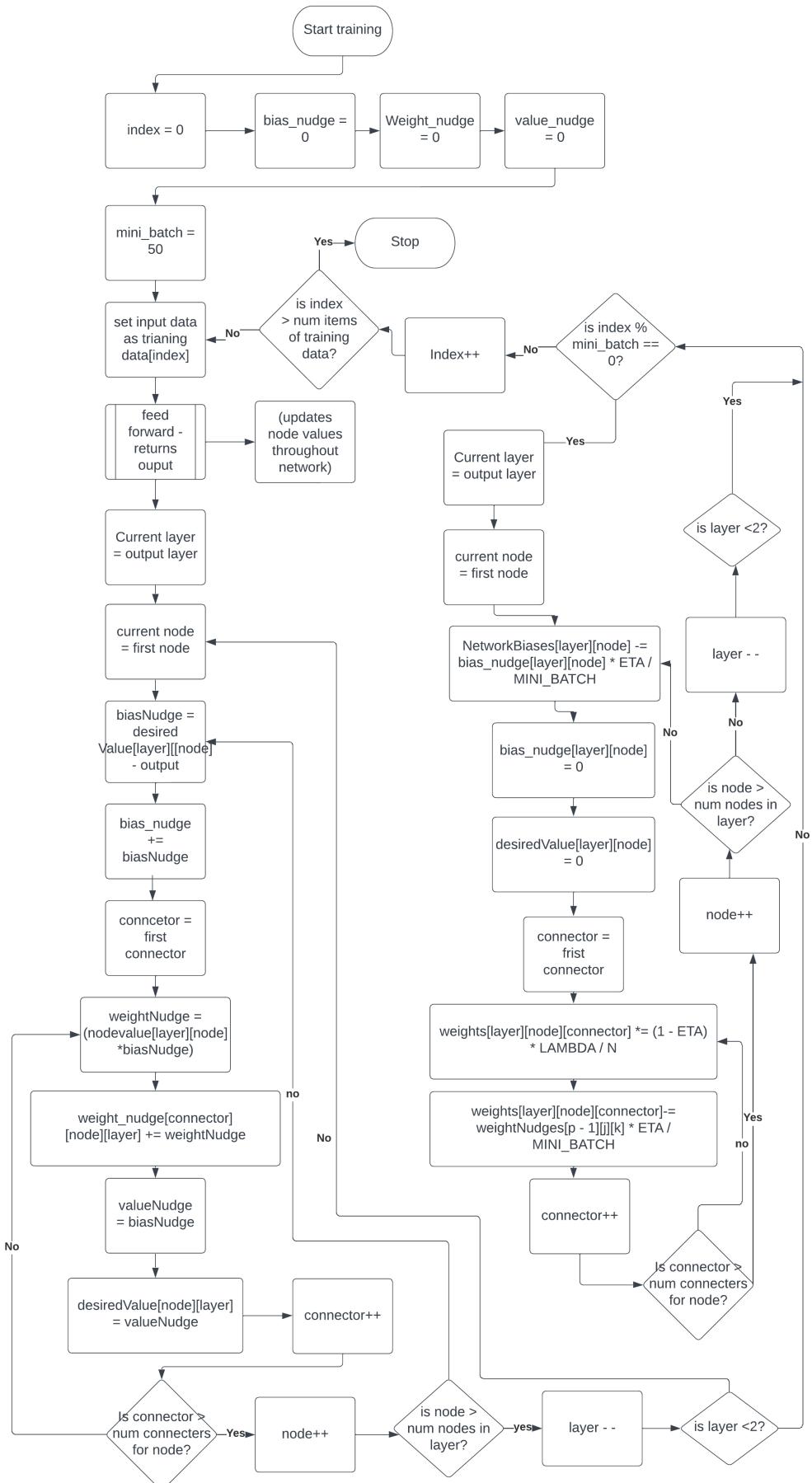
meant to run 'on the side' for as long as possible.

Three important notes for the implementation of the code (these can be seen as comments in the class):

- note that where try catches are used and they can cause a piece to not be placed even if it may be placed legally this is NOT a mistake - this is a more efficient way of creating adequate training data than checking for legal squares
- equally for training purposes since the training is solely for positional chess evaluation it does not matter if black and white pieces occupy the same square as the same principles still apply - this reduces the checking time by 2 fold
- between sessions of running this program the comments that include (piece name) are changed for different pieces in attempt to stop the network overfitting to that specific piece

2.4.1.3 - ALGORITHMS – NEURAL NETWORK – BACKPROPAGATION ALGORITHM

The second is the actual training of the neural network via the backpropagation algorithm using a stochastic gradient descent and batch hybrid (mini batch) which can be seen below. It is noteworthy that the process described wherein each node is the weighted sum of nodes behind is inherently a *matrix multiplication* and will be implemented as such. This will be my implementation of the backpropagation algorithm. The flowchart of the algorithm I will use can be seen below. The other things that must be considered which I mentioned briefly above are the hyper parameters and the activation functions. The two activation functions that I mentioned considering were the sigmoid and tanh. ReLU (or leaky ReLU) has been excluded here as it does not model a distribution and will not work with cross entropy. Since I wish for larger training steps (as this reduces time) I will opt for tanh initially. If after hyper parameter adjusting, I am unhappy with the learning speed I will inevitably have to consider other activation functions.



[Note the feedforward function shown is discussed below in feedforward] – constants referred to in the flowchart: learning rate (eta), l2 regularisation constant (lambda) can be changed in testing.

2.4.1.4 - ALGORITHMS – NEURAL NETWORK – BACKPROPOGATION LEARNING

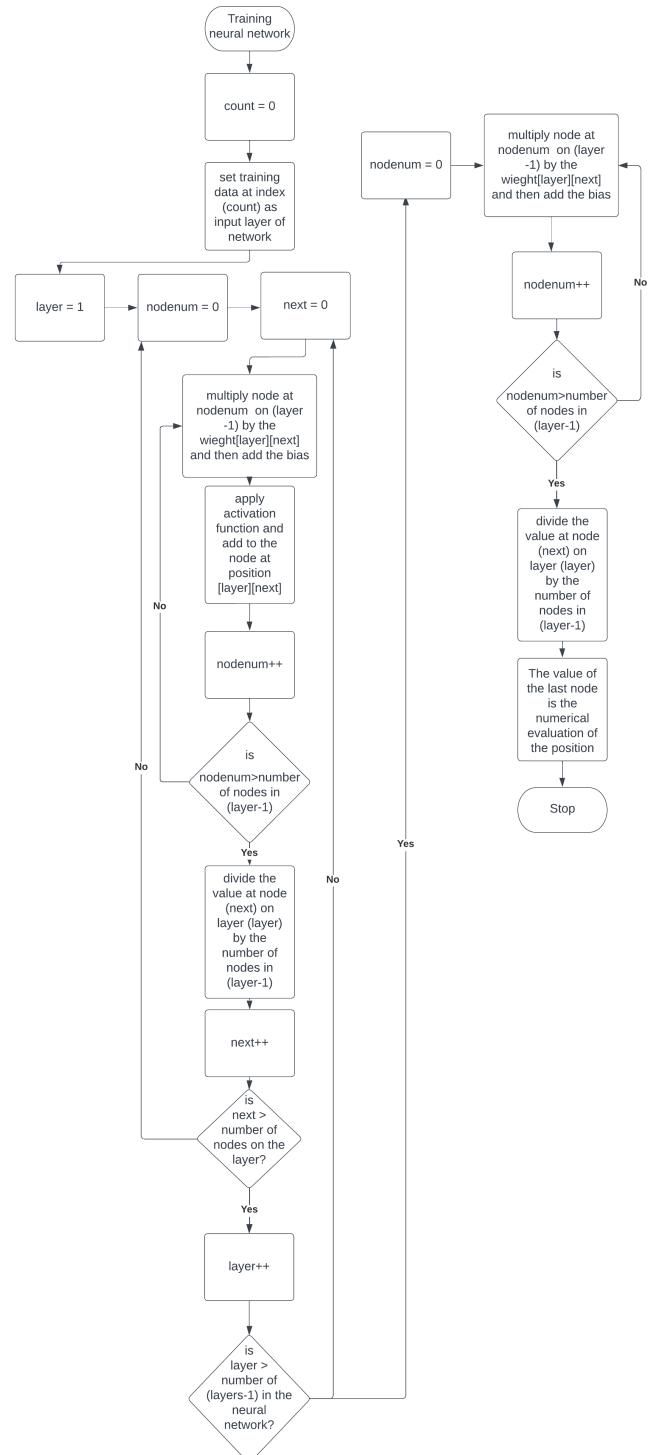
The learning rate of the network depends on many things, these are called hyper parameters. The ones in my network are the following: learning rate (eta), l2 regularisation constant (lambda), mini batch size, activation function, cost function. All of the above and their purpose are spoken about in analysis under neural networks. For every purpose there exists the best values/types for the fastest learning with low overfitting. Since the values can vary so largely here are the *heuristics* (since there is no set algorithm for this process) that I will follow to select my final values:

[Note that there is no set best algorithm for this but it will help me to maintain a good work flow and get the networks learning sufficiently fast in the time frame]

- The first stage is to get the network away from being a random noise generator. That is that the network should learn to perform better than random chance.
 - This will be done by a simple training data algorithm – to speed up the learning to try and obtain data about how the network is learning in less time.
 - For this the algorithm will simply produce positions where one team has protected pawns and the other does not
 - For this problem it seems sensible to remove all but one hidden layers to start with
 - These two points bring the network from a [1792,x,y,1] to a [256,128,1] ($256 \times 7 = 1792$ - each piece per square)
 - The heuristic goal here is that the one hidden layer will detect the patterns between the pawns before the output.
 - I will choose the frequency of testing to be close to 1 – tests once per second (this will likely be around 750 passes of training)
 - The testing will be done on roughly 250 unseen positions
 - The mini-batch wil stay fixed at 50 to start – due to this being a fully custom project and the user not wanting external libraries to be used, my linear alebra implementations run close the $O(n)$ of online learning, so large mini-batch numbers will lead so slow updates of the weights and biases.
 - I will start with orders of 10 changes to the lambda (starting at 0) and eta (learning rate and decay)
 - These will be tuned accordingly with how fast/ whether at all the network learns and how severe overfitting is.
 - Once adequate magnitudes are found for the above two hyper parameters:
 - I will test with implementing a learning rate schedule – this can simply be done by trying different mathematical operations on the learning after a certain number of epochs
 - Lastly, I will come back to the mini-batch number, most likely lowering the value.
- It is clear here that there is not much automation if any. The reason for neglecting methods such as grid search (systematically searching through hyper-parameter space) is due to the already existing harwardware limits that I have with needing to run training data algorithms whenever possible. It may seem sensible to stop these algorithms to instead run a grid search, however, it can be shown that under “*the same computational budget, random search finds*

better models by effectively searching a larger, less promising configuration space". [A paper on Random search for hyper-parameter optimization by James Bergstra and Yoshua Bengio @ <https://dl.acm.org/doi/10.5555/2503308.2188395>]. This implies that randomly searching will perform better and thus I can save the computational power for use else where whilst I manually tune.

2.4.1.4 - ALGORITHMS – NEURAL NETWORK – FEEDFORWARD - ALGORITHM



The feed forward algorithm is the piece of code that will be used by the AI in accompaniment with the final weights and biases from learning. The network will perform the algorithm below (the matrix multiplications) to arrive at the value the neural network has given for a position. It is note worthy that this algorithm is used in backpropagation and so the same algorithm and hence code can be used for the feedforward section in backpropagation.

2.4.1.4 - ALGORITHMS – NEURAL NETWORK – FEEDFORWARD - DATA

The final weights and biases (the state of the neural network) will be loaded in from the final epoch of training and are used whenever the network is requested to make an evaluation. This can be done with two functions in the neural network class that save the network's weights and biases. This implementation also allows saves to occur during training so that the state can be saved and reloaded for training at different times. Below is the c# code used to save and load the state. (Typically networks states are stored by serializing using a binary serializer, however, it is easier to use the same logic can be used for the save game and the artificial training data for me personally).

```
public void SaveNNCurrentState()
{
    string filePath = "Assets/NNState/W.txt";
    string allNums = "";

    foreach(float[][][] layer in weights)
    {
        allNums += "l";
        foreach(float[] from in layer)
        {
            allNums += "f";
            foreach(float weight in from)
            {
                allNums += "w";
                allNums += weight.ToString();
            }
        }
    }
    File.WriteAllText(filePath, allNums);

    filePath = "Assets/NNState/B.txt";
    allNums = "";

    foreach(float[] layer in NodeBiases)
    {
        allNums += "l";
        foreach (float bias in layer)
        {
            allNums += "b";
            allNums += bias.ToString();
        }
    }
    File.WriteAllText(filePath, allNums);
}

public void LoadNNCurrentState()
{
    string filePath = "Assets/NNState/W.txt";
    string allData = File.ReadAllText(filePath);
    int layer = -1;
    int from = -1;
    int to = -1;
```

```

        string weight = "";
        for (int i = 0; i < allData.Length; i++)
        {
            if(allData.Substring(i,1) == "l")
            {
                layer++;
            }
            else if (allData.Substring(i, 1) == "f")
            {
                from++;
            }
            else if (allData.Substring(i, 1) == "w")
            {
                to++;
            }
            else
            {
                weight += allData.Substring(i, 1);
            }
            weights[layer][from][to] = (float)(Convert.ToDouble(weight));
        }

        filePath = "Assets/NNState/B.txt";
        allData = File.ReadAllText(filePath);
        layer = -1;
        to = -1;
        string bias = "";
        for (int i = 0; i < allData.Length; i++)
        {
            if (allData.Substring(i, 1) == "l")
            {
                layer++;
            }
            else if (allData.Substring(i, 1) == "b")
            {
                to++;
            }
            else
            {
                weight += allData.Substring(i, 1);
            }
            NodeBiases[layer][to] = (float)(Convert.ToDouble(bias));
        }
    }
}

```

2.4 - ALGORITHMS – LEGAL MOVES

The implementation of this for when a piece moves occurs as follows (better described in words than via a flowchart as this will be implemented across various methods and classes including overriding dependant on the piece:

- At the start of a turn the get available moves list will be set to none
- For each piece in play you must calculate all legal moves (this is not inefficient as an AI must know this anyways and it is needed for displaying the available moves when a piece is clicked)
 - If the piece moves along lines
 - For every direction vector
 - Iterate over the size of the board.

- Get coordinates of the position the piece is currently in + the iteration number*direction vector
 - If this is outside the bound of the board it will break from the iteration
 - if iterate not broken It will get the piece on those coordinates
 - If there is no piece, there it will call the try to add move function which will determine whether it puts the king into check or leaves it in check and if it does not then it will add the coords to the legal moves.
 - If the piece is from the same team, it will break the iteration.
 - If it is from the other team, it will call the try to add move function which will determine whether it puts the king into check or leaves it in check and if it does not then it will add the coords to the legal moves
- If it does not
 - For every move vector
 - Get coordinates of the position the piece is currently in + the vector
 - If this is outside the bound of the board it will go to the next vector
 - If there is no piece, there it will call the try to add move function which will determine whether it puts the king into check or leaves it in check and if it does not then it will add the coords to the legal moves.
 - If the piece is from the same team, it will break the iteration.
 - If it is from the other team, it will call the try to add move function which will determine whether it puts the king into check or leaves it in check and if it does not then it will add the coords to the legal moves.
 -
- Will move put king into check implementation:
 - Foreach piece on the team
 - The function to remove moves that enable attack on the king from the available moves is called is called.
 - For all the coordinates in the piece's available moves
 - It will get the piece currently at that square
 - Replace that piece with the piece from the foreach
 - Generate all of the opponents moves for the new board position
 - See if the kings coordinates are in the players move list
 - If they are then remove the coordinates under question from the piece in question's move list
 - Replace the piece that was originally on the coordinates back and put the piece in question back to its original square.

Specifics on piece direction vectors are discussed in overall design discussion but also shown below.

```
public class Queen : Piece
{
    private Vector3Int[] directions = new Vector3Int[]
    {
        new Vector3Int(0,0,-1),
        new Vector3Int(-1,-1,-1),
        new Vector3Int(-1,0,-1),
        new Vector3Int(-1,1,-1),
        new Vector3Int(0,-1,-1),
        new Vector3Int(0,1,-1),
        new Vector3Int(1,-1,-1),
        new Vector3Int(1,0,-1),
        new Vector3Int(1,1,-1),

        new Vector3Int(-1,-1,1),
        new Vector3Int(-1,0,1),
        new Vector3Int(-1,1,1),
        new Vector3Int(0,-1,1),
        new Vector3Int(0,1,1),
        new Vector3Int(1,-1,1),
        new Vector3Int(1,0,1),
        new Vector3Int(1,1,1),
        new Vector3Int(0,0,1),

        new Vector3Int(-1,-1,0),
        new Vector3Int(-1,0,0),
        new Vector3Int(-1,1,0),
        new Vector3Int(0,-1,0),
        new Vector3Int(0,1,0),
        new Vector3Int(1,-1,0),
        new Vector3Int(1,0,0),
        new Vector3Int(1,1,0),
    };
}

public class Rook : Piece
{
    private Vector3Int[] directions = new Vector3Int[]
    {
        new Vector3Int(0,0,1),
        new Vector3Int(0,0,-1),
        new Vector3Int(0,1,0),
        new Vector3Int(0,-1,0),
        new Vector3Int(1,0,0),
        new Vector3Int(-1,0,0),
    };
}

public class Bishop : Piece
{
    private Vector3Int[] directions = new Vector3Int[]
    {
        new Vector3Int(1, 1,0),
        new Vector3Int(1, -1,0),
        new Vector3Int(-1, 1,0),
        new Vector3Int(-1,-1,0),
        new Vector3Int(-1,-1,1),
        new Vector3Int(-1,-1,-1),
        new Vector3Int(-1,1,1),
        new Vector3Int(1,-1,-1),
        new Vector3Int(1,-1,1),
        new Vector3Int(1,1,-1),
        new Vector3Int(1,1,1),
    };
}
```

```

public class Commoner : Piece
{
    private Vector3Int[] directions = new Vector3Int[]
    {
        new Vector3Int(0,0,-1),
        new Vector3Int(-1,-1,-1),
        new Vector3Int(-1,0,-1),
        new Vector3Int(-1,1,-1),
        new Vector3Int(0,-1,-1),
        new Vector3Int(0,1,-1),
        new Vector3Int(1,-1,-1),
        new Vector3Int(1,0,-1),
        new Vector3Int(1,1,-1),

        new Vector3Int(-1,-1,1),
        new Vector3Int(-1,0,1),
        new Vector3Int(-1,1,1),
        new Vector3Int(0,-1,1),
        new Vector3Int(0,1,1),
        new Vector3Int(1,-1,1),
        new Vector3Int(1,0,1),
        new Vector3Int(1,1,1),
        new Vector3Int(0,0,1),

        new Vector3Int(-1,-1,0),
        new Vector3Int(-1,0,0),
        new Vector3Int(-1,1,0),
        new Vector3Int(0,-1,0),
        new Vector3Int(0,1,0),
        new Vector3Int(1,-1,0),
        new Vector3Int(1,0,0),
        new Vector3Int(1,1,0),
    };
}

```

```

public class King : Piece
{
    Vector3Int[] directions = new Vector3Int[]
    {
        new Vector3Int(0,0,-1),
        new Vector3Int(-1,-1,-1),
        new Vector3Int(-1,0,-1),
        new Vector3Int(-1,1,-1),
        new Vector3Int(0,-1,-1),
        new Vector3Int(0,1,-1),
        new Vector3Int(1,-1,-1),
        new Vector3Int(1,0,-1),
        new Vector3Int(1,1,-1),

        new Vector3Int(-1,-1,1),
        new Vector3Int(-1,0,1),
        new Vector3Int(-1,1,1),
        new Vector3Int(0,-1,1),
        new Vector3Int(0,1,1),
        new Vector3Int(1,-1,1),
        new Vector3Int(1,0,1),
        new Vector3Int(1,1,1),
        new Vector3Int(0,0,1),

        new Vector3Int(-1,-1,0),
        new Vector3Int(-1,0,0),
        new Vector3Int(-1,1,0),
        new Vector3Int(0,-1,0),
        new Vector3Int(0,1,0),
        new Vector3Int(1,-1,0),
        new Vector3Int(1,0,0),
        new Vector3Int(1,1,0),
    };
}

```

```

public class Knight : Piece
{
    Vector3Int[] offsets = new Vector3Int[]
    {
        new Vector3Int(1,1,2),
        new Vector3Int(1,1,-2),
        new Vector3Int(1,-1,2),
        new Vector3Int(-1,1,2),
        new Vector3Int(-1,-1,2),
        new Vector3Int(-1,1,-2),
        new Vector3Int(1,-1,-2),
        new Vector3Int(-1,-1,-2),

        new Vector3Int(1,2,1),
        new Vector3Int(1,2,-1),
        new Vector3Int(1,-2,1),
        new Vector3Int(-1,2,1),
        new Vector3Int(-1,-2,1),
        new Vector3Int(-1,2,-1),
        new Vector3Int(1,-2,-1),
        new Vector3Int(-1,-2,-1),

        new Vector3Int(2,1,1),
        new Vector3Int(2,1,-1),
        new Vector3Int(2,-1,1),
        new Vector3Int(-2,1,1),
        new Vector3Int(-2,-1,1),
        new Vector3Int(-2,1,-1),
        new Vector3Int(2,-1,-1),
        new Vector3Int(-2,-1,-1),
    };
}

```

```

Vector3Int direction = team == TeamColor.White ? new Vector3Int(0,1,0) : new Vector3Int(0,-1,0);
float range = hasMoved ? 1 : 2;
for (int i = 1; i <= range; i++)
{
    Vector3Int nextCoords = occupiedSquare + direction * i;
    Piece piece = board.GetPieceOnSquare(nextCoords);
    if (!board.CheckIfCoordinatesAreOnBoard(nextCoords))
        break;
    if (piece == null)
        TryToAddMove(nextCoords);
    else
        break;
}

Vector3Int[] takeDirectionsWhite = new Vector3Int[]
{
    new Vector3Int(1,1,1),
    new Vector3Int(-1,1,1),
    new Vector3Int(1,1,-1),
    new Vector3Int(-1,1,-1)
};

Vector3Int[] takeDirectionsBlack = new Vector3Int[]
{
    new Vector3Int(1,-1,1),
    new Vector3Int(-1,-1,1),
    new Vector3Int(1,-1,-1),
    new Vector3Int(-1,-1,-1)
};

```

2.4 – ALGORITHMS – MERGESORT FOR LISTS

```
using System;
using System.Collections.Generic;
namespace ConsoleApp2
{
    class Program
    {
        static void Main(string[] args)
        {
            List<float> sort = new List<float> { -12, 0.12f, 2, 100000, -10000, .013414f, 234.6f };
            sort = MergeSort(sort);
            foreach (float item in sort)
            {
                Console.WriteLine(item);
            }
        }

        static List<float> MergeSort(List<float> nums)
        {
            int count = nums.Count;
            if (count <= 1)
            {
                return nums;
            }
            else
            {
                // splitting
                int middle = count / 2; // truncation - will be left mid if even number of array elements

                List<float> left = new List<float>();
                List<float> right = new List<float>();
                List<float> mergedList = new List<float>();

                for (int i = 0; i < middle; i++)
                {
                    left.Add(nums[i]);
                }
                for (int i = middle; i < count; i++)
                {
                    right.Add(nums[i]);
                }

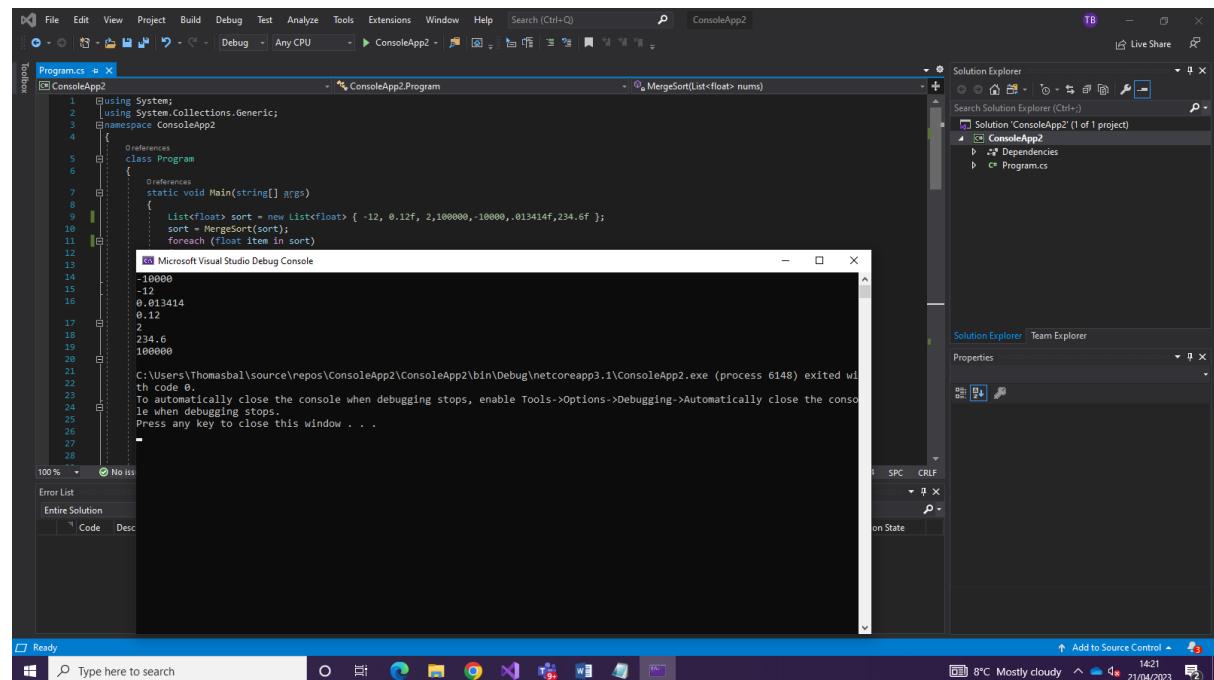
                // recursively split and sort for smaller and smaller list sizes until base case met
                left = MergeSort(left);
                right = MergeSort(right);

                // rebuilding back up bigger and bigger lists until back to base
                while (left.Count > 0 || right.Count > 0)
                {
                    if (right.Count > 0)
                    {
                        if (left.Count > 0)
                        {
                            if (left[0] < right[0])
                            {
                                mergedList.Add(left[0]); // add left if smaller than right at smallest index
                            }
                        }
                    }
                }
            }
        }
    }
}
```

```

        left.RemoveAt(0);
    }
    else
    {
        mergedList.Add(right[0]); //add right if smaller or
equal left at smallest index
        right.RemoveAt(0);
    }
    else
    {
        foreach (float item in right) // if no left remaining fill
list with right
        {
            mergedList.Add(item);
        }
        return mergedList;
    }
    else
    {
        foreach (float item in left)// if no right fill list with left
        {
            mergedList.Add(item);
        }
        return mergedList;
    }
}
return new List<float>() { -1f };
}
}
}

```

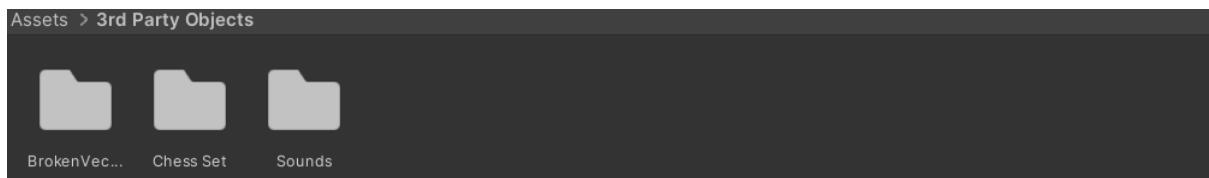


Screenshot shows merge sort working.

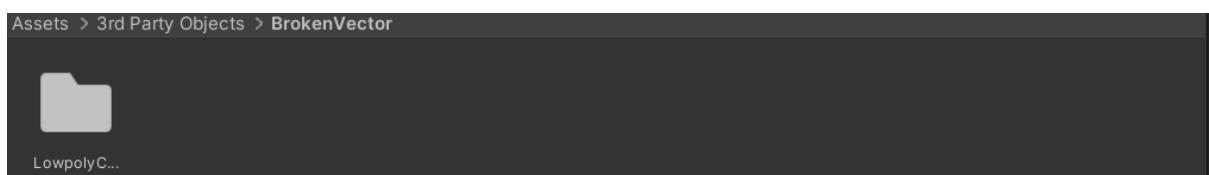
2.5.1 – FILE STRUCTURE



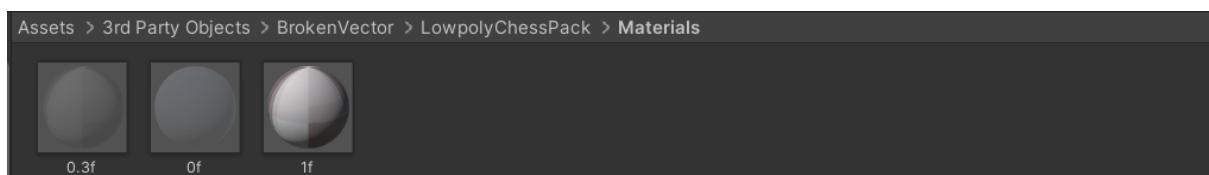
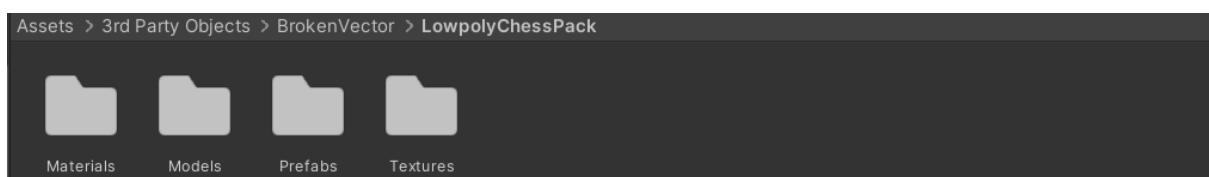
The folders above shows all of the folders for content that I have used to create this program.



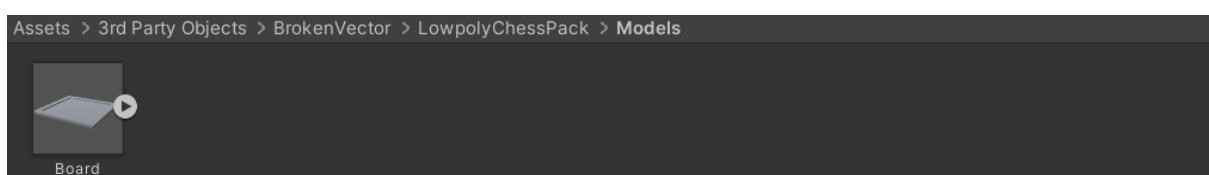
This stores all objects (excluding images) that I have not created myself.



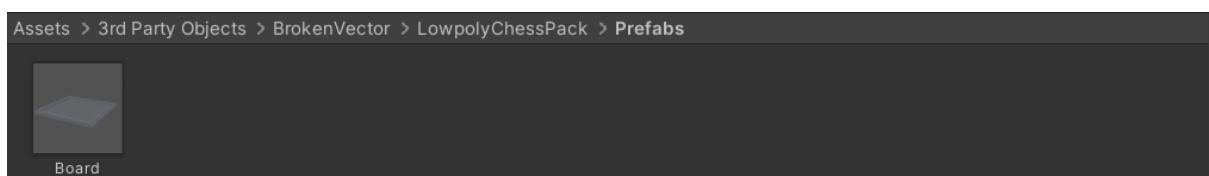
This is a low poly chess set where the chess board object used for each layer is included.



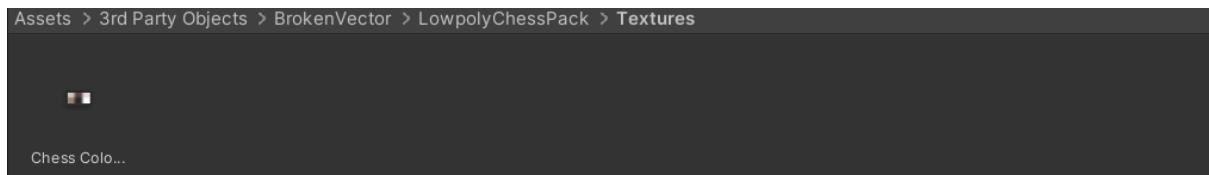
Originally there was only one chess set material, however to create the transparent boards I created more materials.



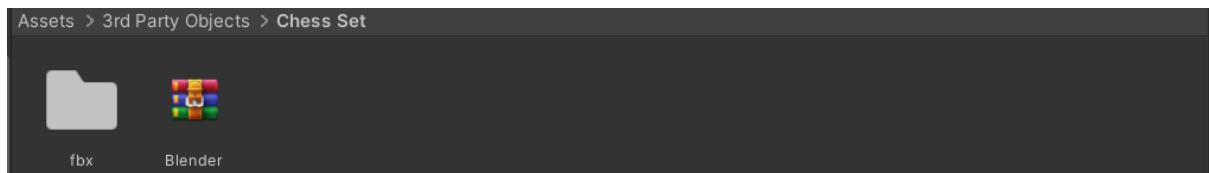
This is the model of the board used in the game.



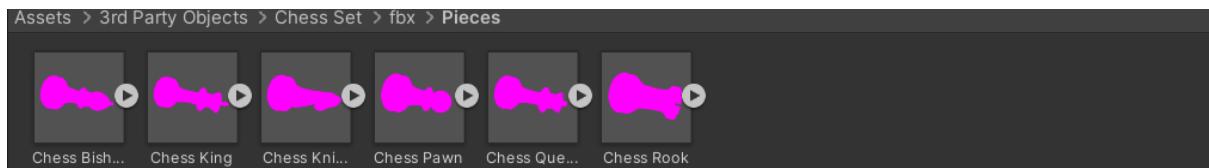
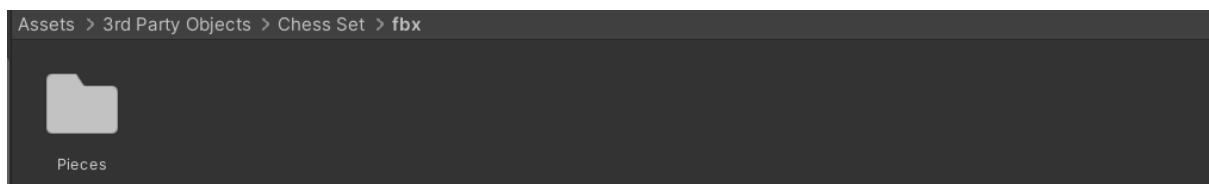
This is the corresponding board prefab.



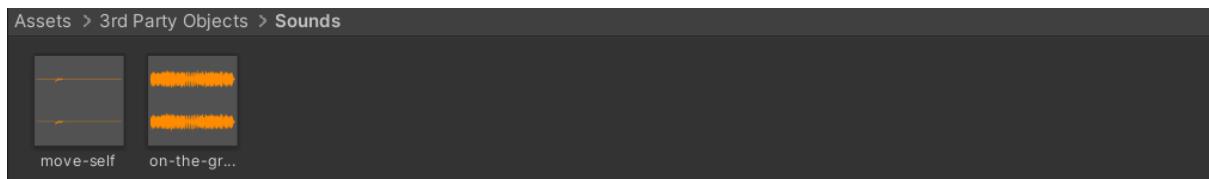
This creates the board pattern.



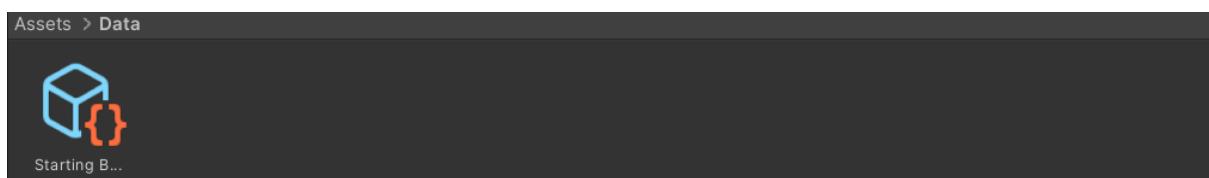
This includes the pieces used



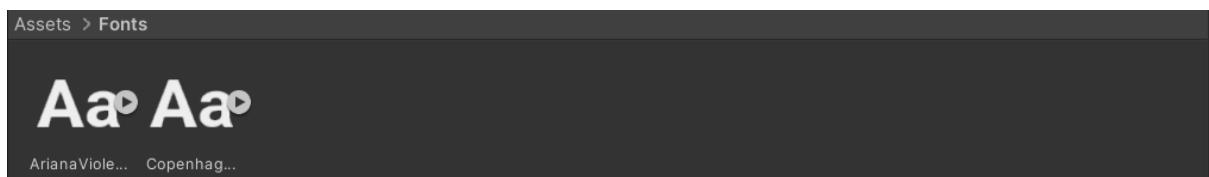
These are all the shapes of pieces needed (commoner and king the same) – purple as default material removed for my own (seen later).



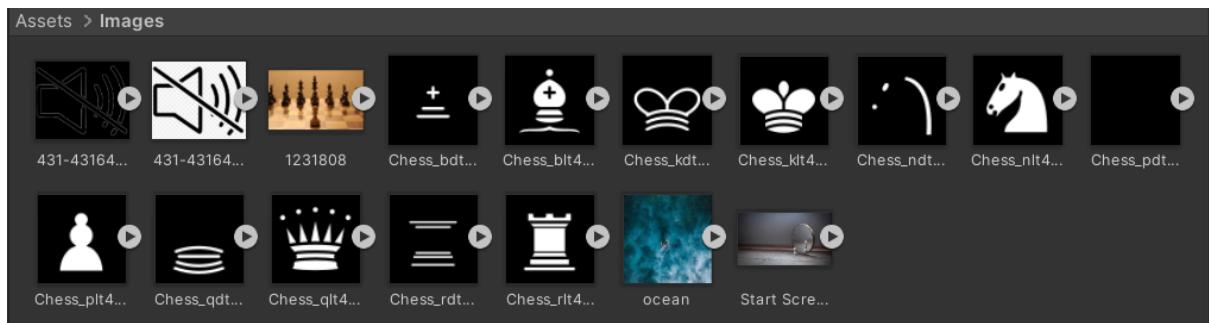
The two sounds that occurs in the game (background and piece move).



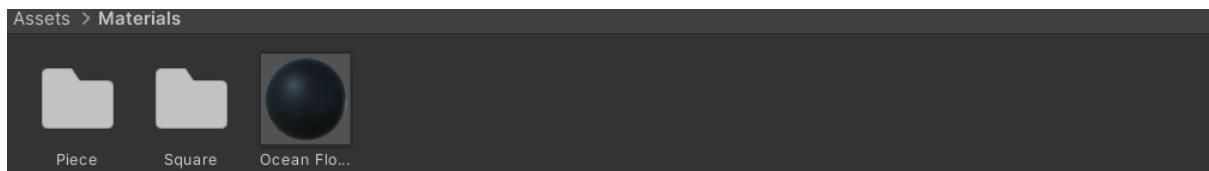
The scriptable object "starting board position" is held here.



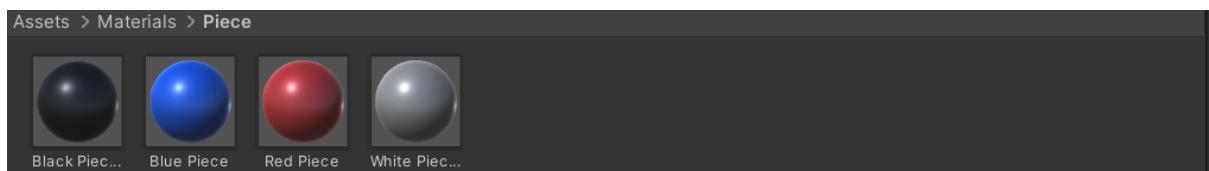
The two fonts used in the game



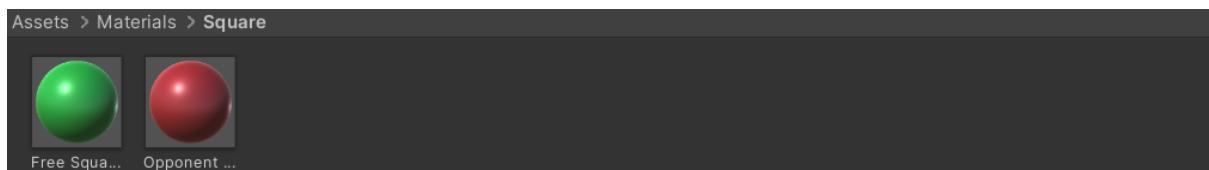
All images used throughout the game



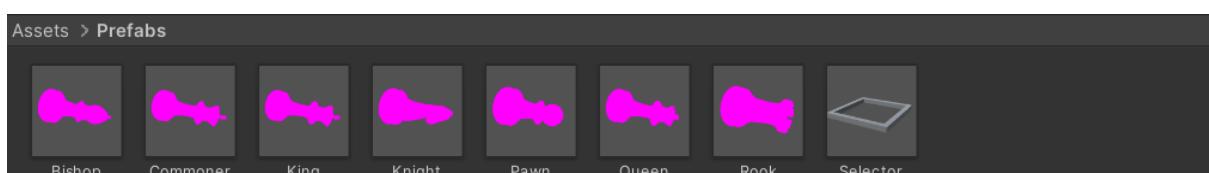
Ocean floor material used for the large base object for below the lowest board



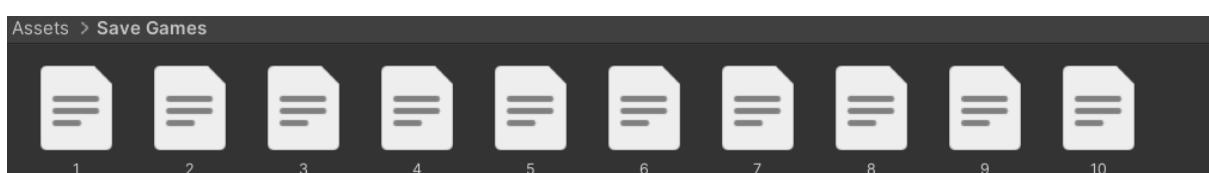
All piece materials



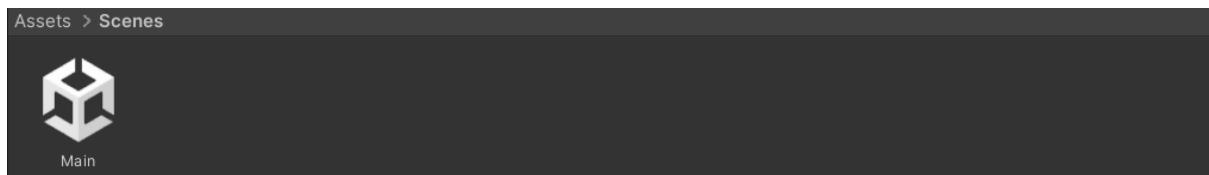
Two materials for available squares



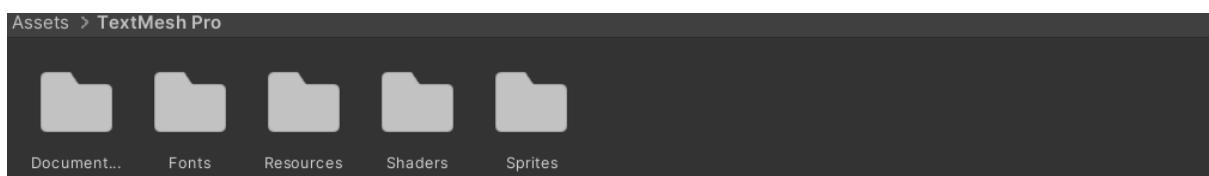
A copy of the third party prefabs in a suitable location + the object used to identify available squares.



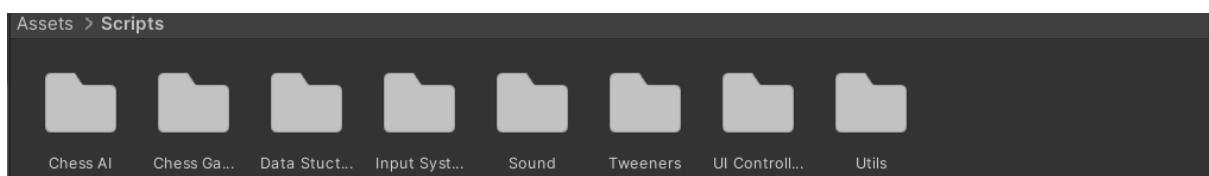
Holds all of the save game files



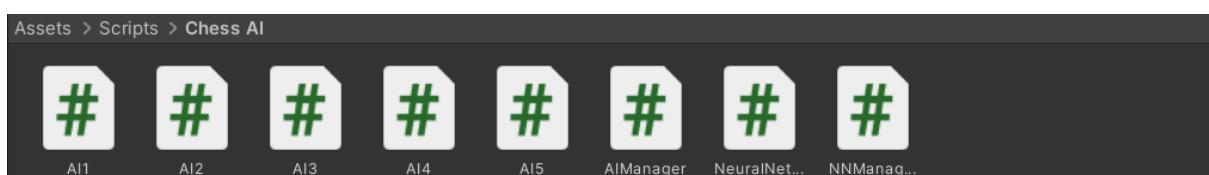
Holds the unity scene that all of the UI and game is in



Holds all of Unity's Text Mesh Pro files



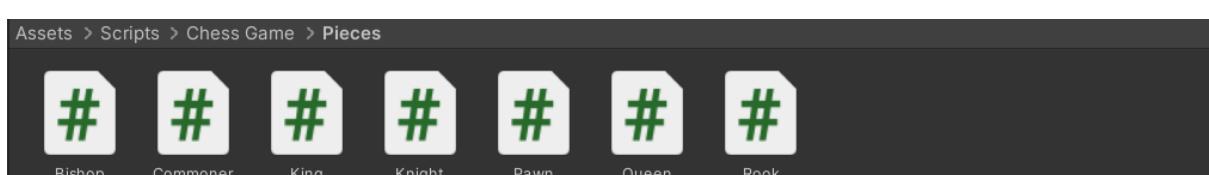
Holds all scripts – note that each script is one class



Holds all classes that control or are AI



All scripts that manage game rules and game play



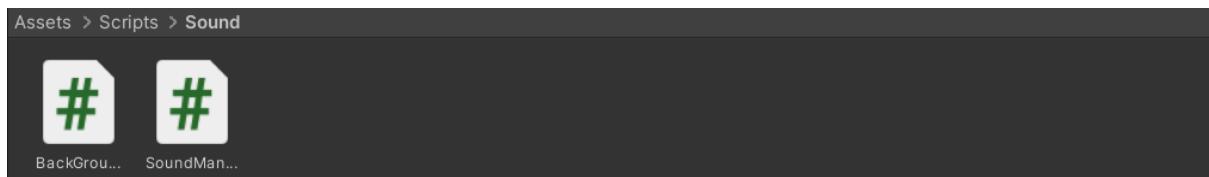
All piece scripts



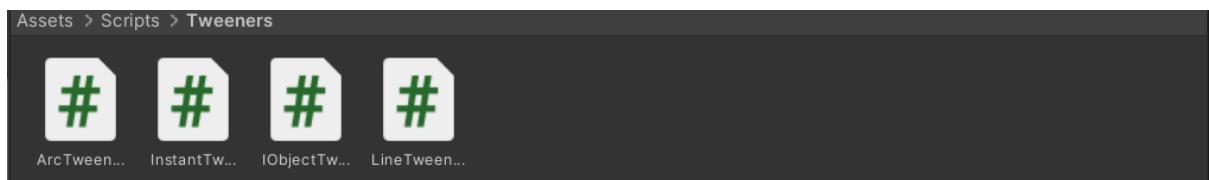
Holds data structures that I have defined



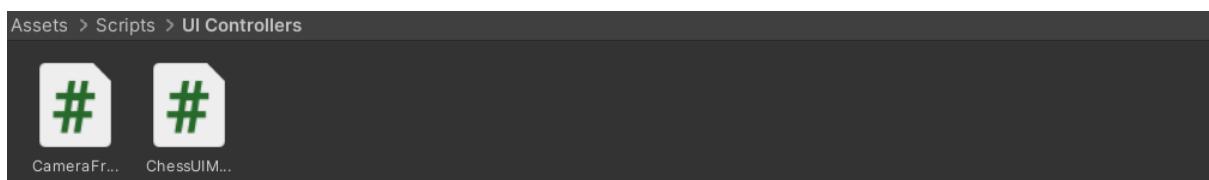
Holds all scripts that deal with the user inputs during gameplay (non UI)



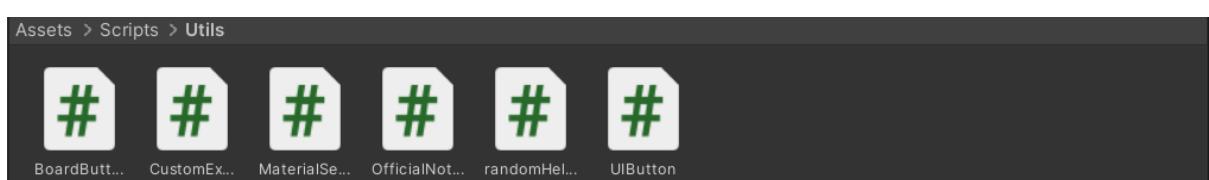
All scripts that deal with sound



All scripts that link and interface other scripts/classes

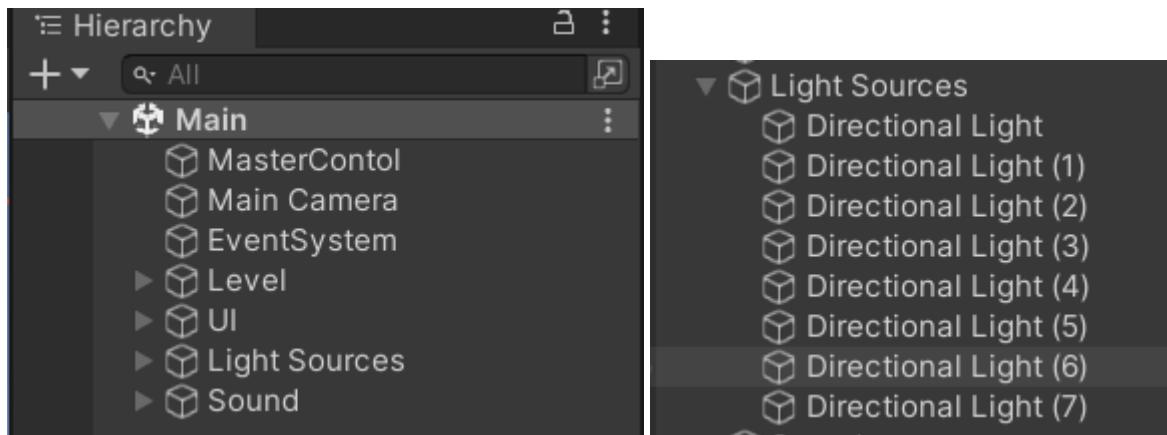


All scripts that control the UI or how the user perceives the game

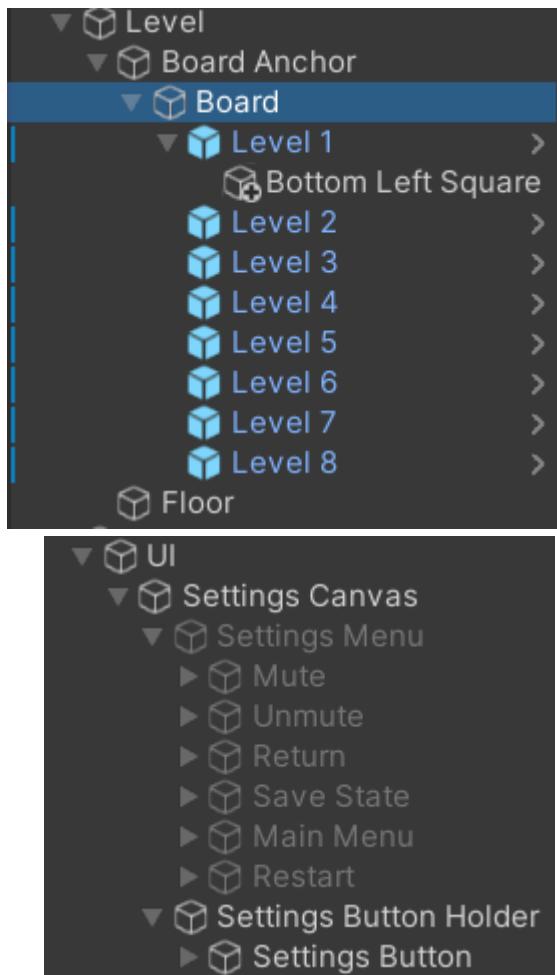


Any other script that gives utility to another script or the user.

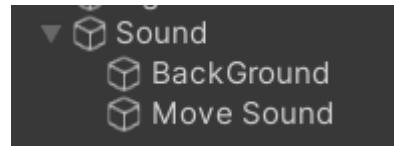
2.5.2 - ORGANISATION



Above is the overall structure of the game objects, with the main sections separated: Master control to hold controlling scripts, main camera, light sources and event system (unity objects) separated, the level (boards) separated, UI separated, and sound separated. Also shown is that light sources hold the source for each layer.

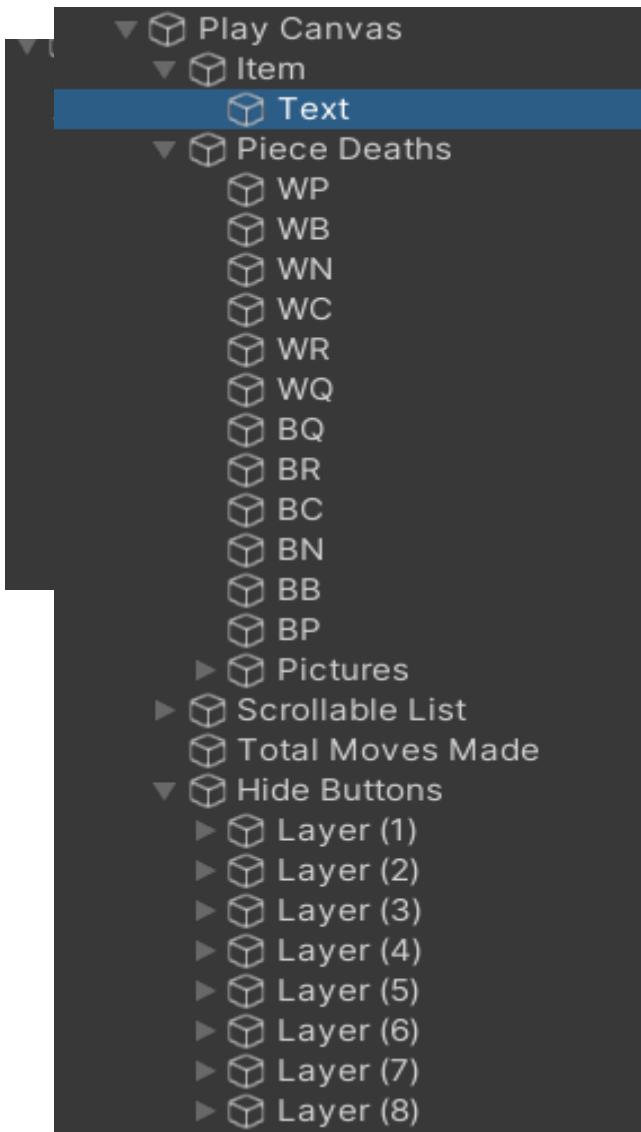


Level contains the board anchor which is an empty game object that holds the board object. The board object includes a collider component for the mouse clicks. Inside the board object are the individual board with an empty object "Bottom Left Square" set as a coordinate helper for identifying what square was clicked". Also included is floor which is the base of the game.



Simply holds two objects that contain audio source components for two types of sounds.

We see that settings canvas has two parts: the object holding the open button and a second section to all the main buttons to be hidden whilst the button stays displayed.



The menu section holds two main parts – the ‘home/gameover’ screen which contains all the initial options. The second part holds all of the intermediate screen which provide access to the games specific difficulty, team and time settings.

The Play canvas hold and ‘item’ which is duplicated to add piece move to the scrollable list. It holds an object for all of the piece death numbers and pictures and all of the buttons that enable layers to go transparent.

2.6 – HCI

The most important part of the HCI is how a user will play the game and interact with the board. The way that this will occur is by the user clicking on the piece or square they wish to select. The following classes, objects and functions deal with this interaction:

The landing Page:



Settings:

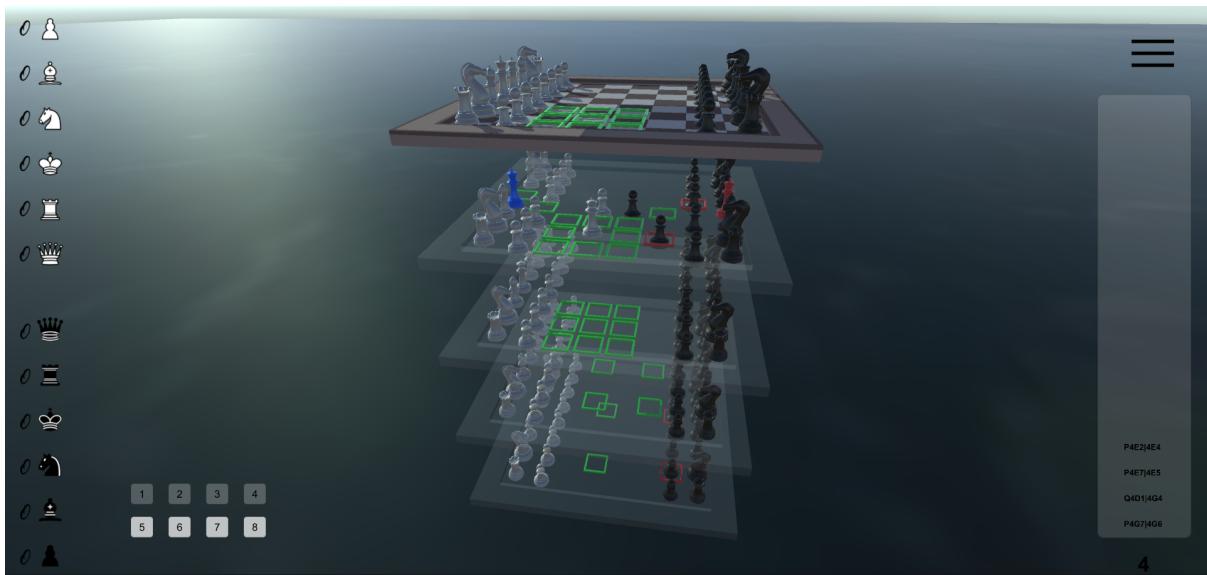


Playing:



Shown above is the scrollable move list on the right along with total number of moves. The available squares for a clicked piece and number of pieces taken on the left.

See-through boards:



2.7 - HARDWARE SELECTION

The strict hardware limitations:

- A mouse/trackpad or other pointer input device
- A computer that has an OS installed and can launch programs
- A monitor/tv or other display device capable of display of at least 420p at 30hz
- 1.5GB of free storage space
- 2GB of Ram at 2400MHz

Soft hardware limitations:

- A 4 core, 1.4Ghz CPU
- A display device capable of at least 720p resolution at 60hz
- 2.5GB of free storage space
- 4GB Ram at 2400MHz

Recommended lowest hardware:

- 4 or 6 cores and 3 or 2GHz respectively
- A display device capable to 1080p (full HD) resolution at 60hz
- 8GB Ram at 3000Mhz

Note the higher specification the machine the faster the AI will make moves.

2.8 - DATA STRUCTURES

A couple examples of each of the following used. They can be seen abundantly throughout the code for several purposes.

LISTS

QUEEN

```
public override List<Vector3Int> SelectAvailableSquares()
{
    availableMoves.Clear();
```

```

        float range = Board.BOARD_SIZE;
        foreach (var direction in directions)
        {
            for (int i = 1; i <= range; i++)
            {
                Vector3Int nextCoords = occupiedSquare + direction * i;
                q piece = board.GetPieceOnSquare(nextCoords);
                if (!board.CheckIfCoordinatesAreOnBoard(nextCoords))
                    break;
                if (piece == null)
                    TryToAddMove(nextCoords);
                else if (!piece.IsFromSameTeam(this))
                {
                    TryToAddMove(nextCoords);
                    break;
                }
                else if (piece.IsFromSameTeam(this))
                    break;
            }
        }
        return availableMoves;
    }

```

CHESS PLAYER

```

public (List<List<Vector3Int>>, List<q>, List<Vector3Int>) ReturnAllPossibleMoves()
{
    List<List<Vector3Int>> movesCurrentlyAvailable = new
List<List<Vector3Int>>();
    List<q> pieces = new List<q>();
    List<Vector3Int> pieceCoords = new List<Vector3Int>();
    for (int i = 0; i < activePieces.Count; i++)
    {
        if ((activePieces[i].availableMoves).Count != 0)
        {
            pieces.Add(activePieces[i]);
            pieceCoords.Add(activePieces[i].occupiedSquare);

movesCurrentlyAvailable.Add(activePieces[i].availableMoves);
        }
    }
    return (movesCurrentlyAvailable, pieces, pieceCoords);
}

public void RemoveMovesEnablingAttackOnPieceOfType<T>(ChessPlayer opponent, q
selectedPiece) where T : q
{
    List<Vector3Int> coordsToRemove = new List<Vector3Int>();
    coordsToRemove.Clear();
    if (selectedPiece)
    {
        foreach (var coords in selectedPiece.availableMoves)
        {
            q pieceOnCoords = board.GetPieceOnSquare(coords);
            board.UpdateBoardOnPieceMove(coords,
selectedPiece.occupiedSquare, selectedPiece, null);
            opponent.GenerateAllPossibleMoves();
            if (opponent.CheckIfIsAttackingPiece<T>())
                coordsToRemove.Add(coords);
            board.UpdateBoardOnPieceMove(selectedPiece.occupiedSquare,
coords, selectedPiece, pieceOnCoords);
        }
    }
}

```

```

        }
        foreach (var coords in coordsToRemove)
        {
            selectedPiece.availableMoves.Remove(coords);
        }
    }

}

```

SORTED DICTIONARIES

AI MANAGER

```

List<List<Vector3Int>> opposingMovesCurrentlyAvailable = new List<List<Vector3Int>>();
List<q> opposingPieces = new List<q>();
List<Vector3Int> opposingPieceCoords = new List<Vector3Int>();
q oldPiece;

SortedDictionary<float, int[]> positions = new SortedDictionary<float,
int[]>();
List<(Vector3Int, Vector3Int)> furtherPositions = new List<(Vector3Int,
Vector3Int)>();
float currentValue = 0;

float small = 0.00001f; // prevents same keys
for (int i = 0; i < movesCurrentlyAvailable.Count; i++)
{
    for (int j = 0; j < movesCurrentlyAvailable.ElementAt(i).Count; j++)
    {
        oldPiece =
board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
pieceCoords.ElementAt(i), pieces.ElementAt(i), null);
        (opposingMovesCurrentlyAvailable, opposingPieces, opposingPieceCoords)
= opposingPlayer.ReturnAllPossibleMoves();
        currentValue =
board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j)) == null ? 0
:
(pieceNameToValueDict[(board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).Ele
mentAt(j))).name] + board.materialImbalance * 0.05f);
        currentValue += IntermediateTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
        if(board.majorPiecesMoved < 50)
        {
            currentValue +=
BasicOpeningTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
        }
        else if(board.majorPiecesTaken < 50)
        {
            currentValue +=
BasicMiddleGameTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
        }
        else
        {

```

```

        currentValue +=
BasicEndGameTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
    }
    for (int k = 0; k < opposingMovesCurrentlyAvailable.Count; k++)
    {
        for (int l = 0; l <
opposingMovesCurrentlyAvailable.ElementAt(k).Count; l++)
        {
            currentValue -=
board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l)) ==
null ? 0 :
(pieceNameToValueDict[(board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementA
t(k).ElementAt(l))).name] + board.materialImbalance * 0.05f);
            currentValue -=
IntermediateTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(l));
            if (board.majorPiecesMoved < 50)
            {
                currentValue -=
BasicOpeningTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(l));
            }
            else if (board.majorPiecesTaken < 50)
            {
                currentValue -=
BasicMiddleGameTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(l));
            }
            else
            {
                currentValue -=
BasicEndGameTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(l));
            }
            while (positions.ContainsKey(currentValue)){currentValue -=
0.00001f;}positions.Add(currentValue, new int[] { i, j });
            small += 0.00001f;
        }
    }
}

board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
pieceCoords.ElementAt(i), oldPiece, pieces.ElementAt(i));
}
}
int numMoves = positions.Count-1;
if (numMoves > 0)
{
    if (accuracyRange == 0)
    {
        if (accuracyDefault > 0)
        {
            for (int a = 0; a < 3; a++)
            {
                int indecies = rnd.Next(Mathf.FloorToInt(positions.Count -
(positions.Count * accuracyDefault)), positions.Count);
                int i = positions.ElementAt(indecies - 1).Value[0];
                int j = positions.ElementAt(indecies - 1).Value[1];
                furtherPositions.Add((pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j)));
            }
        }
    }
}

```

```

        else
        {
            for (int a = 0; a < 3; a++)
            {
                int indecies = rnd.Next(0, Mathf.FloorToInt(positions.Count *
(-1 * accuracyDefault)));
                int i = positions.ElementAt(indecies - 1).Value[0];
                int j = positions.ElementAt(indecies - 1).Value[1];
                furtherPositions.Add((pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j)));
            }
        }
    }
    else
    {
        if (accuracyDefault > 0)
        {

            for (int a = 0; a < 3; a++)
            {
                int indecies = rnd.Next(Mathf.FloorToInt(positions.Count -
(positions.Count * accuracyRange)), Mathf.FloorToInt(positions.Count -
(positions.Count * accuracyDefault)));
                int i = positions.ElementAt(indecies - 1).Value[0];
                int j = positions.ElementAt(indecies - 1).Value[1];
                furtherPositions.Add((pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j)));
            }
        }
        else
        {
            for (int a = 0; a < 3; a++)
            {
                int indecies = rnd.Next(Mathf.FloorToInt(positions.Count * (-1
* accuracyRange)), Mathf.FloorToInt(positions.Count * (-1 * accuracyDefault)));
                int i = positions.ElementAt(indecies - 1).Value[0];
                int j = positions.ElementAt(indecies - 1).Value[1];
                furtherPositions.Add((pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j)));
            }
        }
    }
}

List<(float, Vector3Int, Vector3Int)> furtherPositionsWithEval = new
List<(float, Vector3Int, Vector3Int)>();
for (int a = 0; a < 3; a++)
{
    List<List<Vector3Int>> furtherTempMovesCurrentlyAvailable = new
List<List<Vector3Int>>();
    List<Vector3Int> miniFurtherTempMovesCurrentlyAvailable = new
List<Vector3Int>();
    List<q> furtherTempPieces = new List<q>();
    List<Vector3Int> furtherTempPieceCoords = new List<Vector3Int>();

miniFurtherTempMovesCurrentlyAvailable.Add(furtherPositions.ElementAt(a).Item2);
furtherTempMovesCurrentlyAvailable.Add(miniFurtherTempMovesCurrentlyAvailable);
furtherTempPieces.Add(board.GetPieceOnSquare(furtherPositions.ElementAt(a).Item1));

```

```

        furtherTempPieceCoords.Add(furtherPositions.ElementAt(a).Item1);

        (Vector3Int, Vector3Int) halfWayMoveChosen =
GetMoveMostValueFrom2LookAheadWithAccuracy(furtherTempMovesCurrentlyAvailable,
furtherTempPieces, furtherTempPieceCoords, 0.15f);

        miniFurtherTempMovesCurrentlyAvailable.Clear();
        furtherTempMovesCurrentlyAvailable.Clear();
        furtherTempPieces.Clear();
        furtherTempPieceCoords.Clear();
        miniFurtherTempMovesCurrentlyAvailable.Add(halfWayMoveChosen.Item2);

furtherTempMovesCurrentlyAvailable.Add(miniFurtherTempMovesCurrentlyAvailable);

furtherTempPieces.Add(board.GetPieceOnSquare(halfWayMoveChosen.Item1));
        furtherTempPieceCoords.Add(halfWayMoveChosen.Item1);

furtherPositionsWithEval.Add(GetMoveMostValueFrom2LookAheadWithAccuracyAndReturnEval(furtherTempMovesCurrentlyAvailable, furtherTempPieces, furtherTempPieceCoords, 0.15f));
    }
    float move1 = furtherPositionsWithEval.ElementAt(0).Item1;
    float move2 = furtherPositionsWithEval.ElementAt(1).Item1;
    float move3 = furtherPositionsWithEval.ElementAt(2).Item1;
    if (move1 > move2)
    {
        if (move1 > move3)
        {
            return (furtherPositionsWithEval.ElementAt(0).Item2,
furtherPositionsWithEval.ElementAt(0).Item3);
        }
        else
        {
            return (furtherPositionsWithEval.ElementAt(2).Item2,
furtherPositionsWithEval.ElementAt(2).Item3);
        }
    }
    else
    {
        if (move2 > move3)
        {
            return (furtherPositionsWithEval.ElementAt(1).Item2,
furtherPositionsWithEval.ElementAt(1).Item3);
        }
        else
        {
            return (furtherPositionsWithEval.ElementAt(2).Item2,
furtherPositionsWithEval.ElementAt(2).Item3);
        }
    }
}
else
{
    return GetMoveRandom(movesCurrentlyAvailable, pieceCoords);
}

}

```

DICTIONARIES

AIMANAGER

```

protected Dictionary<String, int> pieceNameToValueDict = new Dictionary<String, int>()
{
    {"King", 1 }, //Point for putting king in check
    {"Pawn", 1 },
    {"Bishop", 3 },
    {"Knight", 3 },
    {"Commoner", 3 },
    {"Rook", 5 },
    {"Queen", 9 },

    {"King(Clone)",1 },
    {"Pawn(Clone)", 1 },
    {"Bishop(Clone)", 3 },
    {"Knight(Clone)", 3 },
    {"Commoner(Clone)", 3 },
    {"Rook(Clone)", 5 },
    {"Queen(Clone)", 9 },
};

protected (Vector3Int, Vector3Int)
GetMoveFrom1LookAheadWithAccuracy(List<List<Vector3Int>> movesCurrentlyAvailable,
List<Vector3Int> pieceCoords, float accuracy)
{
    List<(float,int[])> positions = new List<(float, int[])>();
    float currentValue = 0;
    for (int i = 0; i < movesCurrentlyAvailable.Count; i++)
    {
        for (int j = 0; j < movesCurrentlyAvailable.ElementAt(i).Count; j++)
        {
            currentValue =
board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j)) == null ? 0
:
pieceNameToValueDict[(board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).Elem
entAt(j))).name];
            positions.Add((currentValue, new int[] { i, j }));
        }
    }
    positions = MergeSort(positions); // orders list by lowest evaluation
    int numMoves = positions.Count-1;
    if (numMoves > 0)
    {
        if (accuracy > 0)
        {
            int indecies = rnd.Next(Mathf.FloorToInt(numMoves - (numMoves *
accuracy)), numMoves);
            int i = positions.ElementAt(indecies).Item2[0];
            int j = positions.ElementAt(indecies).Item2[1];
            return (pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
        }
        else
        {
            int indecies = rnd.Next(0, Mathf.FloorToInt((numMoves)*(-
1*accuracy)));
            int i = positions.ElementAt(indecies).Item2[0];
            int j = positions.ElementAt(indecies).Item2[1];
            return (pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
        }
    }
    else
}

```

```

    {
        return GetMoveRandom(movesCurrentlyAvailable, pieceCoords);
    }

}

```

3.0 TECHNICAL SOLUTION

NEURAL NETWORK IMPLEMENTATION – MAIN COMPLEXITY OF PROJECT

[not all functions here are used in the final project but are part of the process]

```
using System;
```

```
using System.IO;
```

```
[Serializable]
```

```
public class NeuralNetwork
```

```
{
```

```
//When Comments Refer to the differented cost equations, these can be found in the analysis section.
```

```
//The cost function used in conjunction with tanh is -0.5 * ( (1-y)*log(1-a) + (1+y)*log(1+a) ) + log(2) //  
(cross-entropy-esque)
```

```
//this gives derives to functions that are proportional to the error
```

```
//namely, dc/dw = x(a-y) and dc/db or dv = a-y
```

```
public float[][] NodeValues; //[[layer in][number of node]
```

```
public float[][] NodeBiases; //[[layer in][number of node]
```

```
public float[][][] weights; // [layer in][node connected to][node connected from]
```

```
public float[][] DesiredValues; // correct values for training
```

```
public float[][] BiasNudges; // how much to nudge for cost
```

```
public float[][][] weightNudges; // "" ""
```

```
private const float ETA = 0.8f; // learning rate
```

```
private const float LAMBDA = 0.001f; //l2 regularisation
```

```

private const int MINI_BATCH = 100; // mini batch size for epoch based training

private const float SCALE = 0.01f; // leaky reLU constant

private int N;

private char a_type;

private char c_type;

private static Random rand = new Random();

public NeuralNetwork(int[] NNcomposure, char activation = 's', char cost = 'm')

{

    a_type = activation;

    c_type = cost;

    N = NNcomposure[0];

    // structure format - {num input nodes, num hidden layer 1 nodes, num hidden layer 2 node,..., num output nodes}

    NodeValues = new float[NNcomposure.Length][];

    NodeBiases = new float[NNcomposure.Length][];

    weights = new float[NNcomposure.Length - 1][][]; //no connection from output layer forwards

    DesiredValues = new float[NNcomposure.Length][];

    BiasNudges = new float[NNcomposure.Length][];

    weightNudges = new float[NNcomposure.Length - 1][][];// "" ""

    for (int i = 0; i < NNcomposure.Length; i++) //adding the respective number of nodes for each layer

    {

        NodeValues[i] = new float[NNcomposure[i]];

        NodeBiases[i] = new float[NNcomposure[i]];


```

```

DesiredValues[i] = new float[NNcomposure[i]];

BiasNudges[i] = new float[NNcomposure[i]];

}

for (int i = 0; i < NNcomposure.Length - 1; i++) //adding the respective number of weights needed per
layer
{
    weights[i] = new float[NodeValues[i + 1].Length](); // nodes to
    weightNudges[i] = new float[NodeValues[i + 1].Length](); //"" ""

    for (int j = 0; j < weights[i].Length; j++)
    {
        weights[i][j] = new float[DesiredValues[i].Length]; //nodes from
        weights[i][j] = new float[DesiredValues[i].Length];//"" ""

        for (int k = 0; k < weights[i][j].Length; k++)
        {
            // set every weight in the NN to a random value between 0 and 1
            // then multiply by the square root of two over the number of nodes in the layer
            // this distribution improves the initial learning of the NN
            weights[i][j][k] = (float)(rand.NextDouble()) * MathF.Sqrt(2f / weights[i][j].Length);
        }
    }
}

public float[] runNetwork(float[] inputs)
{
    for (int i = 0; i < NodeValues[0].Length; i++) // setting values
    {
        NodeValues[0][i] = inputs[i];
    }
}

```

```

    }

    for (int i = 0; i < NodeValues.Length; i++)
    {
        for (int j = 0; j < NodeValues[i].Length; j++)
        {
            //calculating activations

            NodeValues[i][j] = activation(SumForNode(NodeValues[i - 1], weights[i - 1][j]) + NodeBiases[i][j]);
            // sum of all weighted nodes before + the bias for the current node

            DesiredValues[i][j] = NodeValues[i][j]; // setup nodes before training

        }
    }

    return NodeValues[NodeValues.Length - 1];
}

public float[] runNetworkSigmoidSpecific(float[] inputs)
{
    for (int i = 0; i < NodeValues[0].Length; i++) // setting values
    {
        NodeValues[0][i] = inputs[i];

    }

    for (int i = 0; i < NodeValues.Length; i++)
    {
        for (int j = 0; j < NodeValues[i].Length; j++)
        {
            //calculating activations

            NodeValues[i][j] = Sigmoid(SumForNode(NodeValues[i - 1], weights[i - 1][j]) + NodeBiases[i][j]); //
            sum of all weighted nodes before + the bias for the current node
        }
    }
}

```

```

        DesiredValues[i][j] = NodeValues[i][j]; // setup nodes before training
    }

}

return NodeValues[NodeValues.Length - 1];

}

public float[] runNetworkTanhSoftmax(float[] inputs)
{
    for (int i = 0; i < NodeValues[0].Length; i++) // setting values
    {
        NodeValues[0][i] = inputs[i];
    }

    for (int i = 0; i < NodeValues.Length-1; i++)
    {
        for (int j = 0; j < NodeValues[i].Length-1; j++)
        {
            //calculating activations

            NodeValues[i][j] = Tanh(SumForNode(NodeValues[i - 1], weights[i - 1][j]) + NodeBiases[i][j]); // sum of all weighted nodes before + the bias for the current node

            DesiredValues[i][j] = NodeValues[i][j]; // setup nodes before training
        }
    }

    int finalLayer = NodeValues.Length-1;

    for (int i = 0; i < NodeValues[finalLayer].Length; i++)
    {
        NodeValues[finalLayer][i] = Softmax(NodeValues[finalLayer], SumForNode(NodeValues[finalLayer], weights[finalLayer - 1][i]) + NodeBiases[finalLayer][i]); // sum of all weighted nodes before + the bias for the current node

        DesiredValues[finalLayer][i] = NodeValues[finalLayer][i]; // setup nodes before training
    }
}

```

```

        }

        return NodeValues[NodeValues.Length - 1];

    }

    public float[] runNetworkTanh(float[] inputs)
    {
        for (int i = 0; i < NodeValues[0].Length; i++) // setting values

        {

            NodeValues[0][i] = inputs[i];

        }

        for (int i = 0; i < NodeValues.Length; i++)
        {

            for (int j = 0; j < NodeValues[i].Length; j++)
            {

                //calculating activations

                NodeValues[i][j] = Tanh(SumForNode(NodeValues[i - 1], weights[i - 1][j]) + NodeBiases[i][j]); // sum of all weighted nodes before + the bias for the current node

                DesiredValues[i][j] = NodeValues[i][j]; // setup nodes before training

            }

        }

        return NodeValues[NodeValues.Length - 1];
    }

    public void TrainWithBackProp(float[][] Tinputs, float[][] Toutputs)
    {
        int epoch = 0;

        for (int i = 0; i < Tinputs.Length; i++)
        {
            epoch++;
        }
    }
}

```

```

runNetwork(Tinputs[i]); // test the network for every set of trianing data given

for (int j = 0; j < DesiredValues[DesiredValues.Length - 1].Length; j++)
{
    DesiredValues[DesiredValues.Length - 1][j] = Toutputs[i][j]; // adding the wanted outputs to the
desired node values
}

for (int j = NodeValues.Length - 1; j > 0; j--) // back prop up to but excluding the input layer
{
    for (int k = 0; k < NodeValues[j].Length; k++)
    {
        var biasNudge = derivativeB(j, k);

        //// chain rule differentiation for dc/db first - easiest to start of this way as the weights annd value
differentiations include/are the bias differentials in some cases

        BiasNudges[j][k] += biasNudge;

        for (int l = 0; l < NodeValues[j - 1][l].Length; l++)
        {
            var weightNudge = derivativeW(j, l, biasNudge);

            weightNudges[j - 1][k][l] += weightNudge;

            var valueNudge = derivativeV(j, k, l, biasNudge); // again shown by diff - need to have wanted
value for node behind to continue back prop

            DesiredValues[j - 1][l] += valueNudge; // needed for calculating in previous layers
        }
    }
}

if (epoch % MINI_BATCH == 0)
{
    for (int p = NodeValues.Length - 1; p > 0; p--) // for every layer bar the inputs
}

```

```

{
    for (int j = 0; j < NodeValues[i].Length; j++) // for every node
    {
        NodeBiases[p][j] -= BiasNudges[p][j] * ETA / MINI_BATCH; // adjusting the biases
        BiasNudges[p][j] = 0; // resetting for more training

        DesiredValues[p][j] = 0;

        for (int k = 0; k < NodeValues[p - 1].Length; k++)
        {
            weights[p - 1][j][k] *= (1 - ETA) * LAMBDA / N; //adjust weights with accordance to L2
            regularisation

            weights[p - 1][j][k] -= weightNudges[p - 1][j][k] * ETA / MINI_BATCH; //continuation of
            weight adjustement equation

            weightNudges[p - 1][j][k] = 0; //reset
        }
    }
}

}

}

}

}

}

public void TrainWithBackPropAndCrossEntropyWithL2Regularisation(float[][] Tinputs, float[][] Toutputs)
{
    int epoch = 0;

    for (int i = 0; i < Tinputs.Length; i++)
    {
        epoch++;

        runNetwork(Tinputs[i]); // test the network for every set of trianing data given

        for (int j = 0; j < DesiredValues[DesiredValues.Length - 1].Length; j++)

```

```

{
    DesiredValues[DesiredValues.Length - 1][j] = Toutputs[i][j]; // adding the wanted outputs to the
desired node values

}

for (int j = NodeValues.Length - 1; j > 0; j--) // back prop up to but excluding the input layer
{
    for (int k = 0; k < NodeValues[j].Length; k++)
    {
        var biasNudge = DesiredValues[j][k] - NodeValues[j][k];

        // chain rule diff for dc/db as dc/db = error delta = sigmod prime Zl * dc/da where dc/da = al -y
        BiasNudges[j][k] += biasNudge;

        for (int l = 0; l < NodeValues[j - 1].Length; l++)
        {
            var weightNudge = (NodeValues[j - 1][l] * biasNudge)/N; // since the weight differential has a
terms that equal the bias in it the bias can be used here
            weightNudges[j - 1][k][l] += weightNudge;

            var valueNudge = biasNudge; // again shown by diff (they are the same) - need to have wanted
value for node behind to continue back prop

            DesiredValues[j - 1][l] += valueNudge; // needed for calculating in previous layers
        }
    }
}

if (epoch % MINI_BATCH == 0)
{
    for (int p = NodeValues.Length - 1; p > 0; p--) // for every layer bar the inputs
    {
        for (int j = 0; j < NodeValues[i].Length; j++) // for every node
        {
}

```

```

        NodeBiases[p][j] -= BiasNudges[p][j] * ETA / MINI_BATCH; // adjusting the biases
        BiasNudges[p][j] = 0; // resetting for more training

        DesiredValues[p][j] = 0;

        for (int k = 0; k < NodeValues[p - 1].Length; k++)
        {
            weights[p - 1][j][k] *= (1 - ETA) * LAMBDA / N; //adjust weights with accordance to l2
            regularisation
            weights[p - 1][j][k] -= weightNudges[p - 1][j][k] * ETA / MINI_BATCH; //continuation of
            weight adjustement equation
            weightNudges[p - 1][j][k] = 0; //reset
        }
    }
}

}

}

}

}

}

public void TrainWithBackPropAndTanhSoftmaxWithCustomCostAndL2Regularisation(float[][] Tinputs,
float[][] Toutputs)
{
    int epoch = 0;
    for (int i = 0; i < Tinputs.Length; i++)
    {
        epoch++;
        runNetworkTanhSoftmax(Tinputs[i]); // test the network for every set of trianing data given

        for (int j = 0; j < DesiredValues[DesiredValues.Length - 1].Length; j++)
        {
            DesiredValues[DesiredValues.Length - 1][j] += Toutputs[i][j]; // adding the wanted outputs to the
            desired node values
        }
    }
}

```

```
}
```

```
for (int j = NodeValues.Length - 1; j > 0; j--) // back prop up to but excluding the input layer
{
    for (int k = 0; k < NodeValues[j].Length; k++)
    {
        var biasNudge = DesiredValues[j][k] - NodeValues[j][k];
        // chain rule diff for dc/db as dc/db = error delta = sigmod prime ZI * dc/da where dc/da = al - y
        BiasNudges[j][k] += biasNudge;
    }
}
```

```
for (int l = 0; l < NodeValues[j - 1].Length; l++)
{
    var weightNudge = (NodeValues[j - 1][l] * biasNudge); // since the weight differential has a
    terms that equal the bias in it the bias can be used here
    weightNudges[j - 1][k][l] += weightNudge;
}
```

```
var valueNudge = biasNudge; // again shown by diff (they are the same) - need to have wanted
value for node behind to continue back prop
```

```
DesiredValues[j - 1][l] += valueNudge; // needed for calculating in previous layers
}
}
}
```

```
if (epoch % MINI_BATCH == 0)
```

```
{
```

```
for (int p = NodeValues.Length - 1; p > 0; p--) // for every layer bar the inputs
```

```
{
```

```
for (int j = 0; j < NodeValues[i].Length; j++) // for every node
```

```
{
```

```
NodeBiases[p][j] -= BiasNudges[p][j] * ETA / MINI_BATCH; // adjusting the biases
```

```
BiasNudges[p][j] = 0; // resetting for more training
```

```

DesiredValues[p][j] = 0;

for (int k = 0; k < NodeValues[p - 1].Length; k++)
{
    weights[p - 1][j][k] *= (1 - ETA) * LAMBDA / N; //adjust weights with accordance to l2
    regularisation
    weights[p - 1][j][k] -= weightNudges[p - 1][j][k] * ETA / MINI_BATCH; //continuation of
    weight adjustement equation

    weightNudges[p - 1][j][k] = 0; //reset
}
}

}

}

}

}

}

}

public void TrainWithBackPropAndTanhWithCustomCostAndL2Regularisation(float[][] Tinputs, float[][] Toutputs)
{
    int epoch = 0;
    for (int i = 0; i < Tinputs.Length; i++)
    {
        epoch++;
        runNetworkTanh(Tinputs[i]); // test the network for every set of trianing data given

        for (int j = 0; j < DesiredValues[DesiredValues.Length - 1].Length; j++)
        {
            DesiredValues[DesiredValues.Length - 1][j] = Toutputs[i][j]; // adding the wanted outputs to the
            desired node values
        }
    }

    for (int j = NodeValues.Length - 1; j > 0; j--) // back prop up to but excluding the input layer

```

```

{
    for (int k = 0; k < NodeValues[j].Length; k++)
    {
        var biasNudge = DesiredValues[j][k] - NodeValues[j][k];
        // chain rule diff for dc/db as dc/db = error delta = sigmod prime ZI * dc/da where dc/da = al - y
        BiasNudges[j][k] += biasNudge;
        for (int l = 0; l < NodeValues[j - 1][l]; l++)
        {
            var weightNudge = (NodeValues[j - 1][l] * biasNudge); // since the weight differential has a
            terms that equal the bias in it the bias can be used here
            weightNudges[j - 1][l][k] += weightNudge;
        }
        var valueNudge = biasNudge; // again shown by diff (they are the same) - need to have wanted
        value for node behind to continue back prop
        DesiredValues[j - 1][l] += valueNudge; // needed for calculating in previous layers
    }
}

if (epoch % MINI_BATCH == 0)
{
    for (int p = NodeValues.Length - 1; p > 0; p--) // for every layer bar the inputs
    {
        for (int j = 0; j < NodeValues[i].Length; j++) // for every node
        {
            NodeBiases[p][j] -= BiasNudges[p][j] * ETA / MINI_BATCH; // adjusting the biases
            BiasNudges[p][j] = 0; // resetting for more training
        }
        DesiredValues[p][j] = 0;
    }
    for (int k = 0; k < NodeValues[p - 1].Length; k++)
}

```

```

    {
        weights[p - l][j][k] *= (l - ETA) * LAMBDA / N; //adjust weights with accordance to l2
regularisation

        weights[p - l][j][k] -= weightNudges[p - l][j][k] * ETA / MINI_BATCH; //continuation of
weight adjustement equation

        weightNudges[p - l][j][k] = 0; //reset

    }
}

}

}

}

}

```

```

private static float SumForNode(float[] nodeValues, float[] weights)

{
    float finalSum = 0;

    for (int i = 0; i < nodeValues.Length; i++) // for each node in the layer

    {
        finalSum += nodeValues[i] * weights[i]; // multiply weight for the working on node by the last node
activation

    }

    return finalSum;
}

```

```

private float activation(float z, float[] zs = null) //nice way to chose between activation fucntion for small
incomplex data

```

```

{
    if (a_type == 's')
    {

```

```

    return Sigmoid(z);

}

else if (a_type == 'm')

{

    return Softmax(zs, z);

}

else if (a_type == 'r')

{

    return reLU(z);

}

else if (a_type == 'l')

{

    return LeakyreLU(z);

}

else

{

    return Of;

}

}

}

private float derivativeB(int j, int k) //nice way to chose between cost function for small incomplex data

{

    if (c_type == 'q')

    {

        return MSL_SIG_B(j, k);

    }

    else if (c_type == 'c')

    {

        return CE_SIG_B(j, k);

    }

}

```

```

else if (c_type == 'l')
{
    return LL_SM_B(j, k);
}
else
{
    return 0f;
}
}

```

private float derivativeW(int j, int l, float bias, int k = 0) //nice way to chose between cost function for small incomplex data

```

{
    if (c_type == 'q')
    {
        return MSL_SIG_W(j, k, l, bias);
    }
    else if (c_type == 'c')
    {
        return CE_SIG_W(j, k, bias);
    }
    else if (c_type == 'l')
    {
        return LL_SM_W(j, k, bias);
    }
    else
    {
        return 0f;
    }
}

```

```

private float derivativeV(int j, int k, int l, float bias) //nice way to chose between cost function for small
incomplex data

{
    if (c_type == 'q')
    {
        return MSL_SIG_V(j, k, l, bias);
    }
    else if (c_type == 'c')
    {
        return CE_SIG_V(bias);
    }
    else if (c_type == 'l')
    {
        return LL_SM_V(j, k);
    }
    else
    {
        return 0f;
    }
}

```

||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||

```

private static float Sigmoid(float input)
{
    return 1f / (1f + (float)(Math.Exp(-input))); // sigmoid squishification function to get value 0-1
}

```

```
private static float Tanh(float input)
```

```

{

    return 2f / (1f + (float)(Math.Exp(-2 * input))); // tanh gives fast learning than sigmoid and works well with
softmax

}

private static float derivativeTanh(float input)

{

    return 2f / (1f + (float)(Math.Exp(-2 * input))); // tanh gives fast learning than sigmoid and works well with
softmax

}

private static float DerivativeSigmoid(float input) // this is needed for the calculus involved in back prop

{

    return input * (1 - input); // technically this should be sig(x) * (1-sig(x)) but dealt with above

}

private static float reLU(float input) // perhaps try using this - reduces vanishing gradient - perhaps also try
leaky relu

{

    if (input < 0)

    {

        return 0;

    }

    return input;

}

private static float LeakyReLU(float input) //stop problems with dying weights due to small gradient with
negatives

{

    if (input < 0)

    {

        return input / SCALE;
}

```

```

    }
else
{
    return input;
}
}

```

private static float HardSigmoid(float input) // use for large data sets or lots of iterations

```

{
    if (input < -2.5f)
    {
        return 0f;
    }
    if (input > 2.5f)
    {
        return 1f;
    }
    return 0.2f * input + 0.5f;
}

```

private static float Softmax(float[] layerinput, float input) //for classification could be used for determining best moves from a small predefined set

```

{
    float sum = 0;
    foreach (float num in layerinput)
    {
        sum += (float)(Math.Exp((num)));
    }
    return (float)(Math.Exp(input) / sum);
}

```

```

private float MSL_SIG_W(int j, int k, int l, float biasNudge) //mean square loss with sigmoid weight change
{
    return NodeValues[j - 1][l] * biasNudge;
}

private float MSL_SIG_B(int j, int k) //mean square loss with sigmoid bias change
{
    return DerivativeSigmoid(NodeValues[j][k]) * (DesiredValues[j][k] - NodeValues[j][k]);
}

private float MSL_SIG_V(int j, int k, int l, float biasNudge) //mean square loss with sigmoid values change
{
    return weights[j - 1][k][l] * biasNudge;
}

private float CE_SIG_W(int j, int k, float biasNudge) //cross entropy with sigmoid weight change
{
    float sum = 0;
    foreach (float item in NodeValues[j - 1])
    {
        sum += item * biasNudge;
    }
    return sum / N;
}

private float CE_SIG_B(int j, int k) //cross entropy with sigmoid bias change
{
    return (DesiredValues[j][k] - NodeValues[j][k]);
}

private float CE_SIG_V(float biasNudge) //cross entropy with sigmoid values change

```

```

{

    return biasNudge;
}

private float LL_SM_W(int j, int k, float biasNudge) //Logarithmic loss with softmax weight change - classification uses only

{
    float sum = 0;
    foreach (float item in NodeValues[j - 1])
    {
        sum += item * biasNudge;
    }
    return sum / N;
}

private float LL_SM_B(int j, int k) //Logarithmic loss with softmax bias change - classification uses only

{
    return (DesiredValues[j][k] - NodeValues[j][k]);
}

private float LL_SM_V(int j, int k) //Logarithmic loss with softmax value change - classification uses only

{
    return (DesiredValues[j][k] - NodeValues[j][k]);
}

////////////////////////////////////////////////////////////////////////

public void SaveNNCurrentState()

{
    string filePath = "Assets/NNState/W.txt";
}

```

```

string allNums = "";

foreach(float[][] layer in weights)
{
    allNums += "l";
    foreach(float[] from in layer)
    {
        allNums += "f";
        foreach(float weight in from)
        {
            allNums += "w";
            allNums += weight.ToString();
        }
    }
}

File.WriteAllText(filePath, allNums);

filePath = "Assets/NNState/B.txt";
allNums = "";

foreach(float[] layer in NodeBiases)
{
    allNums += "l";
    foreach (float bias in layer)
    {
        allNums += "b";
        allNums += bias.ToString();
    }
}

File.WriteAllText(filePath, allNums);

```

```

    }

public void LoadNNCurrentState()
{
    string filePath = "Assets/NNState/W.txt";
    string allData = File.ReadAllText(filePath);
    int layer = -1;
    int from = -1;
    int to = -1;
    string weight = "";
    for (int i = 0; i < allData.Length; i++)
    {
        if(allData.Substring(i, 1) == "l")
        {
            layer++;
        }
        else if (allData.Substring(i, 1) == "f")
        {
            from++;
        }
        else if (allData.Substring(i, 1) == "w")
        {
            to++;
        }
        else
        {
            weight += allData.Substring(i, 1);
        }
        weights[layer][from][to] = (float)(Convert.ToDouble(weight));
    }
}

```

```

filePath = "Assets/NNState/B.txt";

allData = File.ReadAllText(filePath);

layer = -1;

to = -1;

string bias = "";

for (int i = 0; i < allData.Length; i++)

{

    if (allData.Substring(i, 1) == "l")

    {

        layer++;

    }

    else if (allData.Substring(i, 1) == "b")

    {

        to++;

    }

    else

    {

        weight += allData.Substring(i, 1);

    }

    NodeBiases[layer][to] = (float)(Convert.ToDouble(bias));

}

}

}

```

3.1.1.1 – TECHNICAL SOLUTION – ALGORITHMS – BACK PROPOGATION - USED

This shows the code specifically for the backpropogation used – this just shows the main logic of the code – the variables and other self defined functions can be found above in the full implementation of the neural network – this just helps to show where the main part of the implementation is. This is an example of a beyond a level algorithm that uses complex mathematical ideas. This implementation uses tanh (hyperbolic tan) as activation function with a custom cost deigned for tanh.

```

public void TrainWithBackPropAndTanhWithCustomCostAndL2Regularisation(float[][][] Tinputs, float[][][] Toutputs)
{

```

```

int epoch = 0;
for (int i = 0; i < Tinputs.Length; i++)
{
    epoch++;
    runNetworkTanh(Tinputs[i]); // test the network for every set of trianing
data given

    for (int j = 0; j < DesiredValues[DesiredValues.Length - 1].Length; j++)
    {
        DesiredValues[DesiredValues.Length - 1][j] = Toutputs[i][j]; // adding
the wanted outputs to the desired node values
    }

    for (int j = NodeValues.Length - 1; j > 0; j--) // back prop up to but
excluding the input layer
    {
        for (int k = 0; k < NodeValues[j].Length; k++)
        {
            var biasNudge = DesiredValues[j][k] - NodeValues[j][k];
            // chain rule diff for dc/db as dc/db = error delta = sigmod prime
Zl * dc/dal where dc/dal = al -y
            BiasNudges[j][k] += biasNudge;
            for (int l = 0; l < NodeValues[j - 1][l]; l++)
            {
                var weightNudge = (NodeValues[j - 1][l] * biasNudge); // since
the weight differential has a terms that equal the bias in it the bias can be used
here
                weightNudges[j - 1][k][l] += weightNudge;

                var valueNudge = biasNudge; // again shown by diff (they are
the same) - need to have wanted value for node behind to continue back prop
                DesiredValues[j - 1][l] += valueNudge; // needed for
calculating in previous layers
            }
        }
    }

    if (epoch % MINI_BATCH == 0)
    {
        for (int p = NodeValues.Length - 1; p > 0; p--) // for every layer bar
the inputs
        {
            for (int j = 0; j < NodeValues[i].Length; j++) // for every node
            {
                NodeBiases[p][j] -= BiasNudges[p][j] * ETA / MINI_BATCH; //
adjusting the biases
                BiasNudges[p][j] = 0; // resetting for more training

                DesiredValues[p][j] = 0;

                for (int k = 0; k < NodeValues[p - 1].Length; k++)
                {
                    weights[p - 1][j][k] *= (1 - ETA) * LAMBDA / N; //adjust
weights with accordance to l2 regularisation
                    weights[p - 1][j][k] -= weightNudges[p - 1][j][k] * ETA /
MINI_BATCH; //continuation of weight adjustement equation
                    weightNudges[p - 1][j][k] = 0; //reset
                }
            }
        }
    }
}

```

3.1.1.1 – TECHNICAL SOLUTION – ALGORITHMS – BACK PROPOGATION - UNUSED - I

[Tanh more efficient]

```

public void TrainWithBackPropAndCrossEntropyWithL2Regularisation(float[][][] Tinputs,
float[][][] Toutputs)
{
    int epoch = 0;
    for (int i = 0; i < Tinputs.Length; i++)
    {
        epoch++;
        runNetwork(Tinputs[i]); // test the network for every set of trianing
data given

        for (int j = 0; j < DesiredValues[DesiredValues.Length - 1].Length; j++)
        {
            DesiredValues[DesiredValues.Length - 1][j] = Toutputs[i][j]; // adding
the wanted outputs to the desired node values
        }

        for (int j = NodeValues.Length - 1; j > 0; j--) // back prop up to but
excluding the input layer
        {
            for (int k = 0; k < NodeValues[j].Length; k++)
            {
                var biasNudge = DesiredValues[j][k] - NodeValues[j][k];
                // chain rule diff for dc/db as dc/db = error delta = sigmod prime
Zl * dc/dy where dc/dy = al - y
                BiasNudges[j][k] += biasNudge;
                for (int l = 0; l < NodeValues[j - 1][l]; l++)
                {
                    var weightNudge = (NodeValues[j-1][l] * biasNudge)/N; // since
the weight differential has a terms that equal the bias in it the bias can be used
here
                    weightNudges[j - 1][k][l] += weightNudge;

                    var valueNudge = biasNudge; // again shown by diff (they are
the same) - need to have wanted value for node behind to continue back prop
                    DesiredValues[j - 1][l] += valueNudge; // needed for
calculating in previous layers
                }
            }
        }

        if (epoch % MINI_BATCH == 0)
        {
            for (int p = NodeValues.Length - 1; p > 0; p--) // for every layer bar
the inputs
            {
                for (int j = 0; j < NodeValues[i].Length; j++) // for every node
                {
                    NodeBiases[p][j] -= BiasNudges[p][j] * ETA / MINI_BATCH; //
adjusting the biases
                    BiasNudges[p][j] = 0; // resetting for more training

                    DesiredValues[p][j] = 0;

                    for (int k = 0; k < NodeValues[p - 1].Length; k++)
                    {
                        weights[p - 1][j][k] *= (1 - ETA) * LAMBDA / N; //adjust
weights with accordance to l2 regularisation
                }
            }
        }
    }
}

```

```

                weights[p - 1][j][k] -= weightNudges[p - 1][j][k] * ETA /
MINI_BATCH; //continuation of weight adjustement equation
                weightNudges[p - 1][j][k] = 0; //reset
            }
        }
    }
}
}
}
}
}
}
}
}
}
}
}
}
}
```

3.1.1.1 – TECHNICAL SOLUTION – ALGORITHMS – BACK PROPOGATION - UNUSED -2

[Tanh and cross entropy(and cross entropy esque) more efficient]

```

public void TrainWithBackProp(float[][][] Tinputs, float[][][] Toutputs)
{
    int epoch = 0;
    for (int i = 0; i < Tinputs.Length; i++)
    {
        epoch++;
        runNetwork(Tinputs[i]); // test the network for every set of trianing
        data given

        for (int j = 0; j < DesiredValues[DesiredValues.Length - 1].Length; j++)
        {
            DesiredValues[DesiredValues.Length - 1][j] = Toutputs[i][j]; // adding
            the wanted outputs to the desired node values
        }

        for (int j = NodeValues.Length - 1; j > 0; j--) // back prop up to but
        excluding the input layer
        {
            for (int k = 0; k < NodeValues[j].Length; k++)
            {
                var biasNudge = derivativeB(j, k);
                //// chain rule differentiation for dc/db first - easiest to start
                of this way as the weights annd value differentiations include/are the bias
                differentials in some cases
                BiasNudges[j][k] += biasNudge;
                for (int l = 0; l < NodeValues[j - 1][l]; l++)
                {
                    var weightNudge = derivativeW(j, l, biasNudge);
                    weightNudges[j - 1][k][l] += weightNudge;

                    var valueNudge = derivativeV(j, k, l, biasNudge); // again
                    shown by diff - need to have wanted value for node behind to continue back prop
                    DesiredValues[j - 1][l] += valueNudge; // needed for
                    calculating in previous layers
                }
            }
        }

        if (epoch % MINI_BATCH == 0)
        {
            for (int p = NodeValues.Length - 1; p > 0; p--) // for every layer bar
            the inputs
            {
                for (int j = 0; j < NodeValues[i].Length; j++) // for every node
                {
```

```
adjusting the biases           NodeBiases[p][j] -= BiasNudges[p][j] * ETA / MINI_BATCH; //  
BiasNudges[p][j] = 0; // resetting for more training  
  
DesiredValues[p][j] = 0;  
  
for (int k = 0; k < NodeValues[p - 1].Length; k++)  
{  
    weights[p - 1][j][k] *= (1 - ETA) * LAMBDA / N; //adjust  
weights with accordance to l2 regularisation  
    weights[p - 1][j][k] -= weightNudges[p - 1][j][k] * ETA /  
MINI_BATCH; //continuation of weight adjustement equation  
    weightNudges[p - 1][j][k] = 0; //reset  
}  
}  
}  
}  
}  
}
```

3.1.1.1 – TECHNICAL SOLUTION – ALGORITHMS – BACK PROPOGATION - UNUSED -3

[This setup would be preferable if this was a classification problem as I origionally thought it could be posed as – however having only one output node means that softmax is not the correct choice here]

```
public void TrainWithBackPropAndTanhSoftmaxWithCustomCostAndL2Regularisation(float[][][]  
Tinputs, float[][][] Toutputs)  
{  
    int epoch = 0;  
    for (int i = 0; i < Tinputs.Length; i++)  
    {  
        epoch++;  
        runNetworkTanhSoftmax(Tinputs[i]); // test the network for every set of  
        trianing data given  
  
        for (int j = 0; j < DesiredValues[DesiredValues.Length - 1].Length; j++)  
        {  
            DesiredValues[DesiredValues.Length - 1][j] = Toutputs[i][j]; // adding  
            the wanted outputs to the desired node values  
        }  
  
        for (int j = NodeValues.Length - 1; j > 0; j--) // back prop up to but  
        excluding the input layer  
        {  
            for (int k = 0; k < NodeValues[j].Length; k++)  
            {  
                var biasNudge = DesiredValues[j][k] - NodeValues[j][k];  
                // chain rule diff for dc/db as dc/db = error delta = sigmod prime  
                Zl * dc/dy where dc/dy = al - y  
                BiasNudges[j][k] += biasNudge;  
                for (int l = 0; l < NodeValues[j - 1][l]; l++)  
                {  
                    var weightNudge = (NodeValues[j - 1][l] * biasNudge); // since  
                    the weight differential has a terms that equal the bias in it the bias can be used  
                    here  
                    weightNudges[j - 1][k][l] += weightNudge;  
                }  
            }  
        }  
    }  
}
```

```

        var valueNudge = biasNudge; // again shown by diff (they are
the same) - need to have wanted value for node behind to continue back prop
                DesiredValues[j - 1][1] += valueNudge; // needed for
calculating in previous layers
            }
        }
    }

    if (epoch % MINI_BATCH == 0)
    {
        for (int p = NodeValues.Length - 1; p > 0; p--) // for every layer bar
the inputs
        {
            for (int j = 0; j < NodeValues[i].Length; j++) // for every node
            {
                NodeBiases[p][j] -= BiasNudges[p][j] * ETA / MINI_BATCH; //
adjusting the biases
                BiasNudges[p][j] = 0; // resetting for more training

                DesiredValues[p][j] = 0;

                for (int k = 0; k < NodeValues[p - 1].Length; k++)
                {
                    weights[p - 1][j][k] *= (1 - ETA) * LAMBDA / N; //adjust
weights with accordance to l2 regularisation
                    weights[p - 1][j][k] -= weightNudges[p - 1][j][k] * ETA /
MINI_BATCH; //continuation of weight adjustement equation
                    weightNudges[p - 1][j][k] = 0; //reset
                }
            }
        }
    }
}
}

```

3.1.2.1 – TECHNICAL SOLUTION – ALGORITHMS – FEED FORWARD (MM) - USED

Below is the specific code for the feed forward section of the neural network which displays the implementations of weights and biases.

```

public float[] runNetworkTanh(float[] inputs)
{
    for (int i = 0; i < NodeValues[0].Length; i++) // setting values
    {
        NodeValues[0][i] = inputs[i];
    }

    for (int i = 0; i < NodeValues.Length; i++)
    {
        for (int j = 0; j < NodeValues[i].Length; j++)
        {
            //calculating activations
            NodeValues[i][j] = Tanh(SumForNode(NodeValues[i - 1], weights[i -
1][j]) + NodeBiases[i][j]); // sum of all weighted nodes before + the bias for the
current node
            DesiredValues[i][j] = NodeValues[i][j]; // setup nodes before training
        }
    }
    return NodeValues[NodeValues.Length - 1];
}

```

3.1.2.2 – TECHNICAL SOLUTION – ALGORITHMS – FEED FORWARD (MM) – UNUSED -1

[Tanh and cross entropy(and cross entropy esque) more efficient]

```
public float[] runNetworkSigmoidSpecific(float[] inputs)
{
    for (int i = 0; i < NodeValues[0].Length; i++) // setting values
    {
        NodeValues[0][i] = inputs[i];
    }

    for (int i = 0; i < NodeValues.Length; i++)
    {
        for (int j = 0; j < NodeValues[i].Length; j++)
        {
            //calculating activations
            NodeValues[i][j] = Sigmoid(SumForNode(NodeValues[i - 1], weights[i - 1][j]) + NodeBiases[i][j]); // sum of all weighted nodes before + the bias for the current node
            DesiredValues[i][j] = NodeValues[i][j]; // setup nodes before training
        }
    }
    return NodeValues[NodeValues.Length - 1];
}
```

3.1.2.2 – TECHNICAL SOLUTION – ALGORITHMS – FEED FORWARD (MM) – UNUSED -2

[Not an efficient implementation using a function to call more functions with many if statements involved]

```
public float[] runNetwork(float[] inputs)
{
    for (int i = 0; i < NodeValues[0].Length; i++) // setting values
    {
        NodeValues[0][i] = inputs[i];
    }

    for (int i = 0; i < NodeValues.Length; i++)
    {
        for (int j = 0; j < NodeValues[i].Length; j++)
        {
            //calculating activations
            NodeValues[i][j] = activation(SumForNode(NodeValues[i - 1], weights[i - 1][j]) + NodeBiases[i][j]); // sum of all weighted nodes before + the bias for the current node
            DesiredValues[i][j] = NodeValues[i][j]; // setup nodes before training
        }
    }
    return NodeValues[NodeValues.Length - 1];
}
```

3.1.2.2 – TECHNICAL SOLUTION – ALGORITHMS – FEED FORWARD (MM) – UNUSED -3

[This setup would be preferable if this was a classification problem as I origioanlly thought it could be posed as – however having only one output node means that softmax is not the correct choice here]

```
public float[] runNetworkTanhSoftmax(float[] inputs)
{
    for (int i = 0; i < NodeValues[0].Length; i++) // setting values
    {
        NodeValues[0][i] = inputs[i];
    }

    for (int i = 0; i < NodeValues.Length-1; i++)
    {
        for (int j = 0; j < NodeValues[i].Length-1; j++)
        {
            //calculating activations
            NodeValues[i][j] = Tanh(SumForNode(NodeValues[i - 1], weights[i - 1][j]) + NodeBiases[i][j]); // sum of all weighted nodes before + the bias for the current node
            DesiredValues[i][j] = NodeValues[i][j]; // setup nodes before training
        }
    }
    int finalLayer = NodeValues.Length-1;
    for (int i = 0; i < NodeValues[finalLayer].Length; i++)
    {
        NodeValues[finalLayer][i] = Softmax(NodeValues[finalLayer],
SumForNode(NodeValues[finalLayer], weights[finalLayer - 1][i]) +
NodeBiases[finalLayer][i]); // sum of all weighted nodes before + the bias for the current node
        DesiredValues[finalLayer][i] = NodeValues[finalLayer][i]; // setup nodes before training
    }
    return NodeValues[NodeValues.Length - 1];
}
```

3.1.3.1 – TECHNICAL SOLUTION – ALGORITHMS – ACTIVATION AND COST FUNCTION IMPLEMENTATIONS

Below shows the simple implementations of the various mathematical functions used throughout the testing process.

```
private static float Sigmoid(float input)
{
    return 1f / (1f + (float)(Math.Exp(-input))); // sigmoid squishification
function to get value 0-1
}

private static float Tanh(float input)
{
    return 2f / (1f + (float)(Math.Exp(-2 * input))); // tanh gives fast learning
than sigmoid and works well with softmax
}

private static float derivativeTanh(float input)
{
```

```

        return 2f / (1f + (float)(Math.Exp(-2 * input))); // tanh gives fast learning
than sigmoid and works well with softmax
    }

    private static float DerivativeSigmoid(float input) // this is needed for the
calculus involved in back prop
{
    return input * (1 - input); // technically this should be sig(x) * (1-sig(x))
but delt with above
}

private static float reLU(float input) // perhaps try using this - reduces
vanishing gradient - perhaps also try leaky relu
{
    if (input < 0)
    {
        return 0;
    }
    return input;
}

private static float LeakyReLU(float input) //stop problems with dying weights due
to small graient with negatives
{
    if (input < 0)
    {
        return input / SCALE;
    }
    else
    {
        return input;
    }
}

private static float HardSigmoid(float input) // use for large data sets or lots
of iterations
{
    if (input < -2.5f)
    {
        return 0f;
    }
    if (input > 2.5f)
    {
        return 1f;
    }
    return 0.2f * input + 0.5f;
}

private static float Softmax(float[] layerinput, float input) //for classification
could be used for determining best moves from a small predefined set
{
    float sum = 0;
    foreach (float num in layerinput)
    {
        sum += (float)(Math.Exp((num)));
    }
    return (float)(Math.Exp(input) / sum);
}

private float MSL_SIG_W(int j, int k, int l, float biasNudge) //mean sqaure loss
with sigmoid weight change
{
    return NodeValues[j - 1][l] * biasNudge;
}

```

```

    }

    private float MSL_SIG_B(int j, int k) //mean sqaure loss with sigmoid bias change
    {
        return DerivativeSigmoid(NodeValues[j][k]) * (DesiredValues[j][k] -
NodeValues[j][k]);
    }
    private float MSL_SIG_V(int j, int k, int l, float biasNudge) //mean sqaure loss
with sigmoid values change
    {
        return weights[j - 1][k][l] * biasNudge;
    }

    private float CE_SIG_W(int j, int k, float biasNudge) //cross entropy with sigmoid
weight change
    {
        float sum = 0;
        foreach (float item in NodeValues[j - 1])
        {
            sum += item * biasNudge;
        }
        return sum / N;
    }

    private float CE_SIG_B(int j, int k) //cross entropy with sigmoid bias change
    {
        return (DesiredValues[j][k] - NodeValues[j][k]);
    }

    private float CE_SIG_V(float biasNudge) //cross entropy with sigmoid values change
    {
        return biasNudge;
    }

    private float LL_SM_W(int j, int k, float biasNudge) //Logarithmic loss with
softmax weight change - classification uses only
    {
        float sum = 0;
        foreach (float item in NodeValues[j - 1])
        {
            sum += item * biasNudge;
        }
        return sum / N;
    }

    private float LL_SM_B(int j, int k) //Logarithmic loss with softmax bias change -
classification uses only
    {
        return (DesiredValues[j][k] - NodeValues[j][k]);
    }

    private float LL_SM_V(int j, int k) //Logarithmic loss with softmax value change -
classification uses only
    {
        return (DesiredValues[j][k] - NodeValues[j][k]);
    }

```

3.1.3.2 – TECHNICAL SOLUTION – ALGORITHMS – TRAINING DATA CREATION AND HANDLING

[User defined code for training a neural network. Look at comments for more info on confusing parts of the code/ parts that may seem wrong]

DATA CREATOR

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System;
using System.IO;

public class TrainingData
{
    //note that where try catches are used and they can cause a piece to not be placed even if it may be placed legally this is NOT a mistake - this is a more efficient way of creating adequate trianing data than checking for legal squares

    //equally for trianing purposes since the training is soley for positional chess evaluation it does not matter is black and white pieces occupy the same sqaure as the same principles still apply - this reduces the checking time by 2 fold

    //between sessions of running this program the comments that include (piece name) are changed for differnt pieces in attempt to stop the network overfitting to that specific piece

    private int[][][] board = new int[2][][];
    private int[] selectToMaterial = new int[13] { 2, 1, 2, 3, 3, 3, 3, 5, 1, 3, -3, -12, 1 };
    private float[] selectToAdvantage = new float[13] { -0.1f, -0.1f, -0.2f, -0.2f, 0.3f, 0.1f, 0.3f, 0.1f, 0.4f, 0.3f, 0.4f, 0.2f };

    private void CreateTD()
    {
        string filePath = "Assets/Training Data/T.txt";
        string allData = "";
        float output = 0;
        int pieceNum = 0;
        int material = 0;
```

```

int x = 0;
int y = 0;
int z = 0;
int kingB = 0;
int kingW = 0;
bool placed = false;
System.Random rng = new System.Random();
while (true)
{
    for (int j = 0; j < 100; j++)
    {
        for (int i = 0; i < 8; i++)
        {
            for (int m = 0; m < 8; m++)
            {
                for (int k = 0; k < 8; k++)
                {
                    for (int l = 0; l < 7; l++)
                    {
                        board[0][i][m][k][l] = 0;
                        board[1][i][m][k][l] = 0;
                    }
                }
            }
        }
    }
    output = 0;
    material = 0;
    kingB = 0;
    x = rng.Next(0, 8);
}

```

```

y = rng.Next(0, 8);

z = rng.Next(0, 8);

board[1][x][y][z][0] = 1;

kingB = 100 * x + 10 * y + z;

kingW = 0;

x = rng.Next(0, 8);

y = rng.Next(0, 8);

z = rng.Next(0, 8);

board[0][x][y][z][0] = 1;

kingW = 100 * x + 10 * y + z;

for (int i = 0; i < rng.Next(1, 15); i++)

{

    placed = false;

    pieceNum = rng.Next(1, 15);

    x = rng.Next(0, 8);

    y = rng.Next(0, 8);

    z = rng.Next(0, 8);

    int count = 0;

    foreach (int piece in board[0][x][y][z])

    {

        if (piece != 0)

        {

            break;

        }

        count++;

    }

    if (count == 7)

    {

        if (pieceNum == 1) // protected pawns

        {


```

```

count = 0;

int one = 1;
int two = 1;

if(rng.Next(1, 3) == 1)
{
    one = -1;
}

if (rng.Next(1, 3) == 1)
{
    two = -1;
}

try
{
    foreach (int piece in board[0][x + one][y + 1][z + two])
    {
        if (piece != 0)
        {
            break;
        }
        count++;
    }

    if (count == 7)
    {
        placed = true;
        board[0][x][y][z][7] = 1;
        board[0][x+one][y+one][z+two][7] = 1;
    }
}
catch
{

```

```

        }

    }

else if (pieceNum == 2) // isolated pawn

{
    try

    {

        count = 0;

        foreach (int piece in board[0][x + 1][y - 1][z + 1])

        {

            if (piece != 0)

            {

                break;

            }

            count++;

        }

        if (count == 7)

        {

            count = 0;

            foreach (int piece in board[0][x - 1][y - 1][z + 1])

            {

                if (piece != 0)

                {

                    break;

                }

                count++;

            }

            if (count == 7)

            {

                count = 0;

```

```

foreach (int piece in board[0][x + 1][y - 1][z - 1])
{
    if (piece != 0)
    {
        break;
    }
    count++;
}

if (count == 7)
{
    count = 0;

    foreach (int piece in board[0][x - 1][y - 1][z - 1])
    {
        if (piece != 0)
        {
            break;
        }
        count++;
    }

    if (count != 7)
    {
        placed = true;
        board[0][x][y][z][7] = 1;
    }
}
}

}

catch
{

```

```

    }

}

else if (pieceNum == 3) //doubled pawns

{

for(int a = y; a < 8; a++)

{

if (board[0][x][a][z][7] == 1)

{

placed = true;

board[0][x][y][z][7] = 1;

break;

}

}

}

else if (pieceNum == 4) // knight on edge of the board[0]

{

if(x == 0 || x == 7 || y == 0 || y == 7 || z == 0 || z == 7)

{

placed = true;

board[0][x][y][z][6] = 0;

}

}

else if (pieceNum == 5) // knight with all moves

{

if (2 < x && x < 6 && 2 < y && y < 6 && 2 < z && z < 6)

{

placed = true;

board[0][x][y][z][6] = 0;
}
}

```

```

    }

}

else if (pieceNum == 6)//bishop pair

{
    int type = x % 2;

    for (int a = 0; a < 4; a+=2)

    {
        for (int b = 0; b < 4; b+=2)

        {
            for (int c = 0; c < 8; c++)

            {
                if (board[0][a + type][b + type][c][5] == 1)

                {
                    placed = true;

                    board[0][x][y][z][5] = 1;

                    break;
                }
            }
        }
    }

    if (placed)

    {
        break;
    }
}

if (placed)

{
    break;
}

}
}

else if (pieceNum == 7) //bishop and queen

```

```
{  
    for (int a = -7; a < 8; a++)  
    {  
        for (int b = -7; b < 8; b++)  
        {  
            try  
            {  
                if (board[0][a][b][z][2] == l)  
                {  
                    placed = true;  
                    board[0][x][y][z][4] = l;  
                    break;  
                }  
            }  
        }  
        catch  
        {  
  
        }  
    }  
    for (int c = -7; c < 8; c++)  
    {  
        try {  
            if (board[0][a][b][c][2] == l)  
            {  
                placed = true;  
                board[0][x][y][z][4] = l;  
                break;  
            }  
        }  
    }  
    catch {}  
}
```

```

        if(placed)
        {
            break;
        }

        if (placed) { break; }

    }

}

else if (pieceNum == 8) // rook on open file

{
    int counter = 0;

    for (int a = y; a < 8; a++)
    {
        for (int b = 0; b < 8; b++)
        {
            if(board[0][x][a][z][b] == 0)

            {
                counter++;
            }
        }
    }

    if(counter == 8)
    {
        placed = true;
        board[0][x][y][z][3] = 1;
    }
}

else if (pieceNum == 9) // off starting square (using bishop)

{
    if(x!=2 || (y != 0 && y!=7) || x!=5)
    {

```

```

        board[0][x][y][z][4] = 1;
    }
}

else if (pieceNum == 10) // attacking square that touches kingB (pawn)
{
    for (int a = 0; a < 7; a++)
    {
        int counter = 0;

        if (board[0][Mathf.FloorToInt(kingB / 100) - 2][kingB % 100 - kingB % 10 - 2][kingB % 10 - 2][a] == 0)
        {
            counter++;

        }

        if(counter == 7)
        {
            placed = true;

            board[0][Mathf.FloorToInt(kingB / 100)-2][kingB % 100 - kingB % 10 -2][kingB % 10- 2][6] = 1;
        }
    }
}

else if (pieceNum == 11) // pin on kingB (with pawn)
{
    int counter = 0;

    for (int a = 0; a < 7; a++)
    {

        if (board[0][Mathf.FloorToInt(kingB / 100) - 2][kingB % 100 - kingB % 10 - 2][kingB % 10 - 2][a] == 0 && board[1][Mathf.FloorToInt(kingB / 100) - 1][kingB % 100 - kingB % 10 - 1][kingB % 10 - 1][a] == 0)
    {

```

```

        counter++;
    }

}

if (counter == 7)
{
    placed = true;

    board[0][Mathf.FloorToInt(kingB / 100) - 2][kingB % 100 - kingB % 10 - 2][kingB % 10 -
2][6] = 1;

    board[1][Mathf.FloorToInt(kingB / 100) - 1][kingB % 100 - kingB % 10 - 1][kingB % 10 -
1][5] = 1;
}

else if (pieceNum == 12) // pin on queenB (pawn)
{
    int counter = 0;

    for (int a = 0; a < 7; a++)
    {
        if(board[1][x][y][z][a] == 0 && board[1][x-1][y-1][z-1][a] == 0 && board[0][x-2][y-2][z-
2][a] == 0)
        {

            counter++;
        }
    }

    if (counter == 7)
    {
        placed = true;

        board[1][x][y][z][2] = 1;
        board[1][x - 1][y - 1][z - 1][5] = 1;
        board[0][x - 2][y - 2][z - 2][6] = 1;
    }
}

```

```

    }

}

else // controlling centre squares (pawn)

{
    if((x == 5 || x == 4) && (y == 5 || y == 4))

    {
        board[0][x][y][z][6] = 1;

    }

}

if (placed)

{
    material += selectToMaterial[pieceNum - 1];

    output += selectToAdvantage[pieceNum - 1];

}

placed = false;

pieceNum = rng.Next(1, 15);

x = rng.Next(0, 8);

y = rng.Next(0, 8);

z = rng.Next(0, 8);

foreach (int piece in board[1][x][y][z])

{
    if (piece != 0)

    {
        break;

    }

    count++;

}

if (count == 7)

{

```

```

if (pieceNum == 1) // protected pawns
{
    count = 0;
    int one = 1;
    int two = 1;
    if (rng.Next(1, 3) == 1)
    {
        one = -1;
    }
    if (rng.Next(1, 3) == 1)
    {
        two = -1;
    }
    try
    {
        foreach (int piece in board[1][x + one][y + 1][z + two])
        {
            if (piece != 0)
            {
                break;
            }
            count++;
        }
        if (count == 7)
        {
            placed = true;
            board[1][x][y][z][7] = 1;
            board[1][x + one][y + one][z + two][7] = 1;
        }
    }
}

```

```

        catch

    {

}

else if (pieceNum == 2) // isolated pawn

{
    try

    {

        count = 0;

        foreach (int piece in board[1][x + 1][y - 1][z + 1])

        {

            if (piece != 0)

            {

                break;

            }

            count++;

        }

        if (count == 7)

        {

            count = 0;

            foreach (int piece in board[1][x - 1][y - 1][z + 1])

            {

                if (piece != 0)

                {

                    break;

                }

                count++;

            }

            if (count == 7)

```

```

{
    count = 0;

    foreach (int piece in board[!][x + !][y - !][z - !])

    {
        if (piece != 0)

        {
            break;
        }

        count++;
    }

    if (count == 7)

    {
        count = 0;

        foreach (int piece in board[!][x - !][y - !][z - !])

        {
            if (piece != 0)

            {
                break;
            }

            count++;
        }

        if (count != 7)

        {
            placed = true;

            board[!][x][y][z][7] = !;
        }
    }
}

```

```

        catch

    {

}

else if (pieceNum == 3) //doubled pawns

{

    for (int a = y; a < 8; a++)

    {

        if (board[l][x][a][z][7] == l)

        {

            placed = true;

            board[l][x][y][z][7] = l;

            break;

        }

    }

}

else if (pieceNum == 4) // knight on edge of the board[l

// ]

{

    if (x == 0 || x == 7 || y == 0 || y == 7 || z == 0 || z == 7)

    {

        placed = true;

        board[l][x][y][z][6] = 0;

    }

}

else if (pieceNum == 5) // knight with all moves

{

    if (2 < x && x < 6 && 2 < y && y < 6 && 2 < z && z < 6)

```

```

{
    placed = true;
    board[l][x][y][z][6] = 0;
}

}

else if (pieceNum == 6)//bishop pair

{
    int type = x % 2;

    for (int a = 0; a < 4; a += 2)

    {
        for (int b = 0; b < 4; b += 2)

        {
            for (int c = 0; c < 8; c++)

            {
                if (board[l][a + type][b + type][c][5] == l)

                {
                    placed = true;
                    board[l][x][y][z][5] = l;
                    break;
                }
            }
        }
    }

    if (placed)
    {
        break;
    }
}

if (placed)
{
    break;
}

```

```

    }

}

else if (pieceNum == 7) //bishop and queen

{
    for (int a = -7; a < 8; a++)

    {
        for (int b = -7; b < 8; b++)

        {

            try

            {

                if (board[1][a][b][z][2] == 1)

                {

                    placed = true;

                    board[1][x][y][z][4] = 1;

                    break;

                }

            }

            catch

            {

            }

        }

        for (int c = -7; c < 8; c++)

        {

            try

            {

                if (board[1][a][b][c][2] == 1)

                {

                    placed = true;

                    board[1][x][y][z][4] = 1;

                    break;

                }

            }

        }

    }

}

```

```

        }
    }

    catch {}

}

if (placed)
{ break; }

}

if (placed) { break; }

}

}

else if (pieceNum == 8) // rook on open file
{
    int counter = 0;

    for (int a = y; a < 8; a++)
    {
        for (int b = 0; b < 8; b++)
        {
            if (board[l][x][a][z][b] == 0)
            {
                counter++;
            }
        }
    }

    if (counter == 8)
    {
        placed = true;
        board[l][x][y][z][3] = l;
    }
}

```

```

else if (pieceNum == 9) // off starting square (using bishop)
{
    if (x != 2 || (y != 0 && y != 7) || x != 5)
    {
        board[1][x][y][z][4] = 1;
    }
}

else if (pieceNum == 10) // attacking sqaure that touches kingW (pawn)
{
    for (int a = 0; a < 7; a++)
    {
        int counter = 0;

        if (board[1][Mathf.FloorToInt(kingW / 100) - 2][kingW % 100 - kingW % 10 - 2][kingW % 10 - 2][a] == 0)
        {
            counter++;
        }

        if (counter == 7)
        {
            placed = true;

            board[1][Mathf.FloorToInt(kingW / 100) - 2][kingW % 100 - kingW % 10 - 2][kingW % 10 - 2][6] = 1;
        }
    }
}

else if (pieceNum == 11) // pin on kingW (with pawn)
{
    int counter = 0;

    for (int a = 0; a < 7; a++)
    {

```

```

        if (board[1][Mathf.FloorToInt(kingW / 100) - 2][kingW % 100 - kingW % 10 -
2][kingW % 10 - 2][a] == 0 && board[0][Mathf.FloorToInt(kingW / 100) - 1][kingW % 100 - kingW % 10 -
1][kingW % 10 - 1][a] == 0)

    {

        counter++;

    }

}

if (counter == 7)

{

    placed = true;

    board[1][Mathf.FloorToInt(kingW / 100) - 2][kingW % 100 - kingW % 10 - 2][kingW %
10 - 2][6] = 1;

    board[0][Mathf.FloorToInt(kingW / 100) - 1][kingW % 100 - kingW % 10 - 1][kingW %
10 - 1][5] = 1;

}

else if (pieceNum == 12) // pin on queenW (pawn)

{

    int counter = 0;

    for (int a = 0; a < 7; a++)

    {

        if (board[0][x][y][z][a] == 0 && board[0][x - 1][y - 1][z - 1][a] == 0 && board[1][x -
2][y - 2][z - 2][a] == 0)

        {

            counter++;

        }

    }

}

if (counter == 7)

{

```

```

placed = true;

board[0][x][y][z][2] = 1;

board[0][x - 1][y - 1][z - 1][5] = 1;

board[1][x - 2][y - 2][z - 2][6] = 1;

}

}

else // controlling centre squares (pawn)

{

if ((x == 5 || x == 4) && (y == 5 || y == 4))

{

board[1][x][y][z][6] = 1;

}

}

if (placed)

{

material -= selectToMaterial[pieceNum - 1];

output -= selectToAdvantage[pieceNum - 1];

}

}

}

int team = 0;

int num = 0;

if (material < 0)

{

team = 1;

material = -1 * material;

}

for (int a = 0; a < 8; a++)

{

```

```

for (int b = 0; b < 8; b++)
{
    for (int c = 0; c < 8; c++)
    {
        num = 0;
        for (int d = 0; d < 7; d++)
        {
            if(board[team][a][b][c][d] == 0)
            {
                num++;
            }
        }
        if (num == 7)
        {
            material--;
            board[team][a][b][c][6] = 1;
            //protected
            if(board[team][a + 1][b + 1][c + 1][6] == 1 || board[team][a - 1][b + 1][c + 1][6] == 1 ||
            board[team][a + 1][b + 1][c - 1][6] == 1 || board[team][a - 1][b + 1][c - 1][6] == 1)
            {
                output += team == 0 ? 0.1f : -0.1f;
            }
            else // isolated
            {
                output += team == 0 ? -0.1f : 0.1f;
            }
        }
        for (int i = b; i < 8; i++) // doubled
        {
            if(board[team][a][i][c][6] == 1)
            {

```

```
    output += team == 0 ? -0.2f : 0.2f;

}

}

if (material == 0)

{

    break;

}

}

if (material == 0)

{

    break;

}

}

if (material == 0)

{

    break;

}

}

for (int a = 0; a < 2; a++)

{

    for (int b = 0; b < 8; b++)

    {

        for (int c = 0; c < 8; c++)

        {

            for (int d = 0; d < 8; d++)

            {

                for (int e = 0; e < 7; e++)

                {
```

```

        allData += board[a][b][c][d][e];

    }

}

}

}

allData += output;

}

File.WriteAllText(filePath, allData);

}

}

}

```

NN INTERPRETER

Relevant parts of the NN manager class

```

public NeuralNetwork mainStateNetwork = new NeuralNetwork(new int[5]
{7168,3584,512,64,1}); // one input per piece per square per team, hidden layer in
// hopes it identifies black team negativity, hidden layer size of board, hidden layer 5x
// smaller, output

private void TrainingNNWithData()
{
    float[][][] tData = GetTData();
    float[][] TinputsWhole = tData[0];
    float[][] ToutputsWhole = tData[1];
    float[][] Tinputs = new float[100][];
    float[][] Toutputs = new float[100][];
    mainStateNetwork.LoadNNCurrentState();
    for (int i = 0; i < TinputsWhole.Length/100; i+=100)
    {
        for (int j = 0; j < 100; j++)
        {
            Tinputs[j] = TinputsWhole[i + j];
            Toutputs[j] = ToutputsWhole[i + j];
        }
    }

    mainStateNetwork.TrainWithBackPropAndTanhWithCustomCostAndL2Regularisation(Tinputs,
    Toutputs);
    mainStateNetwork.SaveNNCurrentState();
}
}

private float[][][] GetTData()
{
    string filePath = "Assets/Training Data/T.txt";
    string allData = File.ReadAllText(filePath);
    float[][][] tData = new float[2][][];

```

```

        float[] data = new float[7168];
        float[] output = new float[100000];
        int current = 0;
        for (int i = 0; i < allData.Length / 7169; i += 7169) //7 pieces per square
per team + one output
    {
        for (int j = 0; j < 7168; j++)
        {
            data[j] = Convert.ToInt32(allData[i + j]);
        }
        output[0] = Convert.ToInt32(allData[i + 7169]);
        tData[0][current] = data;
        tData[1][current] = output;
    }
    return tData;
}

```

OUTSIDE AI PARENT FUNCTIONS THAT ARE USED

These are functions that relate to how the AIs make move and may be helpful in better understanding the logic behind them:

BOARD

```

public Vector3Int getPieceCoords(string name, TeamColor team)
{
    for (int i = 0; i < BOARD_SIZE; i++)
    {
        for (int j = 0; j < BOARD_SIZE; j++)
        {
            for (int k = 0; k < BOARD_SIZE; k++)
            {
                if (grid[i, j, k] != null)
                {
                    if (grid[i, j, k].name == name && grid[i, j, k].team == team)
                    {
                        return new Vector3Int(i, j, k);
                    }
                }
            }
        }
    }
    return new Vector3Int(-1, -1, -1);
}

public void UpdateBoardOnPieceMove(Vector3Int newCoords, Vector3Int oldCoords, Piece
newPiece, Piece oldPiece)
{
    grid[oldCoords.x, oldCoords.y, oldCoords.z] = oldPiece;
    grid[newCoords.x, newCoords.y, newCoords.z] = newPiece;
}

public Piece GetPieceOnSquare(Vector3Int coords)
{
    if (CheckIfCoordinatesAreOnBoard(coords))
        return grid[coords.x, coords.y, coords.z];
    return null;
}

```

```
}
```

NN MANAGER

```
public float[] FeedForward(float[] inputs)
{
    return mainStateNetwork.runNetworkTanh(inputs); // as i know i will jsut be
using this combination of activation and cost easier to implement than many if
statements
}
```

ABOUT THE AI – COMPLEX USER DEFINED CODE/SYSTEM

All the code is user defined with all the ideas surrounding how to make the AI are original, made to play as close to a human as possible and optimised to give the fastest playing times i.e with deeper line analysis selection instead of traversing the entire move tree.

The AIs from three onwards, whilst not directly using any exiting algorithms such as alpha beta pruning, use the idea of pruning to look to a greater depth on relevant lines. This selection for pruning is covered by the sorted dictionaries, theory addition functions and the the piece to value dictionary. These in combination create line evalutions and irrelevant lines (good or bad – depening on whether the Ai is playing a good, bad, mediocre move) can be chosen or ‘pruned’. This adapted tree search and pruning on the possible moves is a complex user defined system which I have derived for chess and adapted to 3D chess. (In addition to this the neural network aid in structural play but this has already been discussed).

3.1.4.1 – TECHNICAL SOLUTION – ALGORITHMS – AI – I

AII CLASS

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class AII : AIManager
{
    public AII (ChessPlayer currentAiPlayer, ChessPlayer opposingPlayer, Board board)
    {
        this.currentAiPlayer = currentAiPlayer;
        this.board = board;
        this.opposingPlayer = opposingPlayer;
    }
    public override void MoveMakerController(ChessPlayer aiPlayer, ChessPlayer
opposingPlayer)
    {
        NewAiPlayers(aiPlayer, opposingPlayer);
        List<List<Vector3Int>> movesCurrentlyAvailable = new List<List<Vector3Int>>();
        List<Piece> pieces = new List<Piece>();
        List<Vector3Int> pieceCoords = new List<Vector3Int>();
        (movesCurrentlyAvailable, pieces, pieceCoords) =
currentAiPlayer.ReturnAllPossibleMoves();

        Vector3Int coordsOfPieceToMove;
        Vector3Int coordsToMoveTo;

        if(rnd.Next(1,11) == 5)
        {
            (coordsOfPieceToMove, coordsToMoveTo) =
GetMoveFrom1LookAheadWithAccuracy(movesCurrentlyAvailable, pieceCoords, -0.1f);
            Debug.Log("Blunder");
        }
    }
}
```

```

        else if (rnd.Next(1, 3) == 1)
    {
        (coordsOfPieceToMove, coordsToMoveTo) =
GetMoveFrom1LookAheadWithAccuracy(movesCurrentlyAvailable, pieceCoords, 0.25f);
    }
    else if(rnd.Next(1,6) == 5)
    {
        (coordsOfPieceToMove, coordsToMoveTo) =
GetMoveRandom(movesCurrentlyAvailable, pieceCoords);
        Debug.Log("Blunder");
    }
    else
    {
        (coordsOfPieceToMove, coordsToMoveTo) =
GetMoveMostValueFrom2LookAheadWithAccuracy(movesCurrentlyAvailable, pieces,
pieceCoords, 0.25f);
    }

    board.AI_makeMove(coordsOfPieceToMove, coordsToMoveTo);

}
}

```

PARENT FUNCTIONS USED

[Basic Ai functions]

```

protected (Vector3Int, Vector3Int)
GetMoveFrom1LookAheadWithAccuracy(List<List<Vector3Int>> movesCurrentlyAvailable,
List<Vector3Int> pieceCoords, float accuracy)
{
    List<(float,int[])> positions = new List<(float, int[])>();
    float currentValue = 0;
    for (int i = 0; i < movesCurrentlyAvailable.Count; i++)
    {
        for (int j = 0; j < movesCurrentlyAvailable.ElementAt(i).Count; j++)
        {
            currentValue =
board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j)) == null ? 0
:
pieceNameToValueDict[(board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j))).name];
            positions.Add((currentValue, new int[] { i, j }));
        }
    }
    positions = MergeSort(positions); // orders list by lowest evaluation
    int numMoves = positions.Count - 1;
    if (numMoves > 0)
    {
        if (accuracy > 0)
        {
            int indecies = rnd.Next(Mathf.FloorToInt(numMoves - (numMoves *
accuracy)), numMoves);
            int i = positions.ElementAt(indecies).Item2[0];
            int j = positions.ElementAt(indecies).Item2[1];
            return (pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
        }
        else
        {
            int indecies = rnd.Next(0, Mathf.FloorToInt((numMoves)*(-
1*accuracy)));
            int i = positions.ElementAt(indecies).Item2[0];

```

```

                int j = positions.ElementAt(indexes).Item2[1];
                return (pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
            }

        }
        else
        {
            return GetMoveRandom(movesCurrentlyAvailable, pieceCoords);
        }
    }

protected (Vector3Int, Vector3Int) GetMoveRandom(List<List<Vector3Int>>
movesCurrentlyAvailable, List<Vector3Int> pieceCoords)
{
    if(pieceCoords.Count > 0)
    {
        int val = Guid.NewGuid().GetHashCode() % pieceCoords.Count;
        int atIndex = val > 0 ? val : -val;
        val = Guid.NewGuid().GetHashCode() %
movesCurrentlyAvailable.ElementAt(atIndex).Count;
        int toIndex = val > 0 ? val : -val;
        return (pieceCoords.ElementAt(atIndex),
movesCurrentlyAvailable.ElementAt(atIndex).ElementAt(toIndex));
    }
    Debug.Log("error");
    return (Vector3Int.zero, Vector3Int.zero);
}

private List<(float, int[])> MergeSort(List<(float, int[])> nums)
{
    int count = nums.Count;
    if (count <= 1)
    {
        return nums;
    }
    else
    {
        // splitting
        int middle = count / 2; // truncation - will be left mid if even number of
array elements

        List<(float, int[])> left = new List<(float, int[])>();
        List<(float, int[])> right = new List<(float, int[])>();
        List<(float, int[])> mergedList = new List<(float, int[])>();

        for (int i = 0; i < middle; i++)
        {
            left.Add(nums[i]);
        }
        for (int i = middle; i < count; i++)
        {
            right.Add(nums[i]);
        }

        //recursively split and sort for smaller and smaller list sizes until base
case met
        left = MergeSort(left);
        right = MergeSort(right);
    }
}

```

```

        // rebuilding back up bigger and bigger lists until back to base function
call
    while (left.Count > 0 || right.Count > 0)
    {
        if (right.Count > 0)
        {
            if (left.Count > 0)
            {
                if (left[0].Item1 < right[0].Item1)
                {
                    mergedList.Add(left[0]); // add left if smaller than right
at smallest index
                    left.RemoveAt(0);
                }
                else
                {
                    mergedList.Add(right[0]); //add right if smaller or equal
left at smallest index
                    right.RemoveAt(0);
                }
            }
            else
            {
                foreach ((float, int[]) item in right) // if no left remaining
fill list with right
                {
                    mergedList.Add(item);
                }
                return mergedList;
            }
        }
        else
        {
            foreach ((float, int[]) item in left)// if no right fill list with
left
            {
                mergedList.Add(item);
            }
            return mergedList;
        }
    }
    return new List<(float, int[])>() { (-1f, new int[0]) };
}
}

```

3.1.4.2 – TECHNICAL SOLUTION – ALGORITHMS – AI – 2

AI2 CLASS

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class AI2 : AIManager
{
    public AI2 (ChessPlayer currentAiPlayer, ChessPlayer opposingPlayer, Board board)
    {
        this.currentAiPlayer = currentAiPlayer;
        this.board = board;
        this.opposingPlayer = opposingPlayer;
    }
}

```

```

    public override void MoveMakerController(ChessPlayer aiPlayer, ChessPlayer
opposingPlayer)
{
    NewAiPlayers(aiPlayer, opposingPlayer);
    List<List<Vector3Int>> movesCurrentlyAvailable = new List<List<Vector3Int>>();
    List<Piece> pieces = new List<Piece>();
    List<Vector3Int> pieceCoords = new List<Vector3Int>();
    (movesCurrentlyAvailable, pieces, pieceCoords) =
currentAiPlayer.ReturnAllPossibleMoves();

    Vector3Int coordsOfPieceToMove = Vector3Int.zero;
    Vector3Int coordsToMoveTo = Vector3Int.zero;
    int randomVal = rnd.Next(0, 100);

    if (randomVal < 15)//15%
    {
        (coordsOfPieceToMove, coordsToMoveTo) =
GetMoveMostValueFrom2LookAheadWithAccuracy(movesCurrentlyAvailable, pieces,
pieceCoords, -0.15f);
        Debug.Log("Blunder");
    }
    else if (randomVal < 20)//20%
    {
        (coordsOfPieceToMove, coordsToMoveTo) =
GetMoveMostValueFrom2LookAheadWithAccuracy(movesCurrentlyAvailable, pieces,
pieceCoords, -0.5f, -0.25f);
        Debug.Log("Innacuracy");
    }
    else
    {
        (coordsOfPieceToMove, coordsToMoveTo) =
GetMoveMostValueFrom2LookAheadWithAccuracyAndLargeMoveNegativityMitigation(movesCurren
tlyAvailable, pieces, pieceCoords, 0.15f);
    }

    board.AIMakeMove(coordsOfPieceToMove, coordsToMoveTo);
}
}

```

PARENT FUNCTIONS USED

[Shows much more complex ideas about how the AI should choose moves]

```

protected (Vector3Int, Vector3Int)
GetMoveMostValueFrom2LookAheadWithAccuracy(List<List<Vector3Int>>
movesCurrentlyAvailable, List<Piece> pieces, List<Vector3Int> pieceCoords, float
accuracyDefault, float accuracyRange = 0f)
{
    List<List<Vector3Int>> opposingMovesCurrentlyAvailable = new
List<List<Vector3Int>>();
    List<Piece> opposingPieces = new List<Piece>();
    List<Vector3Int> opposingPieceCoords = new List<Vector3Int>();
    Piece oldPiece;

    SortedDictionary<float, int[]> positions = new SortedDictionary<float,
int[]>();
    float currentValue = 0;
    float small = 0.00001f; // prevents same keys
    for (int i = 0; i < movesCurrentlyAvailable.Count; i++)
    {
        for (int j = 0; j < movesCurrentlyAvailable.ElementAt(i).Count; j++)
        {

```

```

        oldPiece =
board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
pieceCoords.ElementAt(i), pieces.ElementAt(i), null);
    (opposingMovesCurrentlyAvailable, opposingPieces, opposingPieceCoords)
= opposingPlayer.ReturnAllPossibleMoves();
    currentValue =
board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j)) == null ? 0
:
pieceNameToValueDict[(board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).Elem
entAt(j))).name];
    currentValue +=
BasicTheoryAddition(pieceCoords.ElementAt(i),movesCurrentlyAvailable.ElementAt(i),boar
d.GetPieceOnSquare(pieceCoords.ElementAt(i)) == null? true:
board.GetPieceOnSquare(pieceCoords.ElementAt(i)).team == TeamColor.White ?
true:false);
        for (int k = 0; k < opposingMovesCurrentlyAvailable.Count; k++)
{
    for (int l = 0; l <
opposingMovesCurrentlyAvailable.ElementAt(k).Count; l++)
    {
        currentValue -=
board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l)) ==
null ? 0 :
pieceNameToValueDict[(board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt
(k).ElementAt(l))).name];
        currentValue -=
BasicTheoryAddition(opposingPieceCoords.ElementAt(k),opposingMovesCurrentlyAvailable.E
lementAt(k), board.GetPieceOnSquare(pieceCoords.ElementAt(i)) == null ? true :
board.GetPieceOnSquare(pieceCoords.ElementAt(i)).team == TeamColor.White ? true :
false);
            while (positions.ContainsKey(currentValue)){currentValue ==
0.00001f;}positions.Add(currentValue, new int[] { i, j });
            small += 0.00001f;
        }
    }
}

board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
pieceCoords.ElementAt(i), oldPiece, pieces.ElementAt(i));
    }
}
int numMoves = positions.Count-1;

if (numMoves > 0)
{
    if (accuracyRange == 0)
    {
        if (accuracyDefault > 0)
        {
            int indecies = rnd.Next(Mathf.FloorToInt(positions.Count -
(positions.Count * accuracyDefault)), positions.Count);
            int i = positions.ElementAt(indecies - 1).Value[0];
            int j = positions.ElementAt(indecies - 1).Value[1];
            return (pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
        }
        else
        {
            int indecies = rnd.Next(0, Mathf.FloorToInt(positions.Count * (-1
* accuracyDefault)));
            int i = positions.ElementAt(indecies - 1).Value[0];

```

```

                int j = positions.ElementAt(indicies - 1).Value[1];
                return (pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
            }
        }
        else
        {
            if (accuracyDefault > 0)
            {
                int indicies = rnd.Next(Mathf.FloorToInt(positions.Count -
(positions.Count * accuracyRange)), Mathf.FloorToInt(positions.Count -
(positions.Count * accuracyDefault)));
                int i = positions.ElementAt(indicies - 1).Value[0];
                int j = positions.ElementAt(indicies - 1).Value[1];
                return (pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
            }
            else
            {
                int indicies = rnd.Next(Mathf.FloorToInt(positions.Count * (-1 *
accuracyRange)), Mathf.FloorToInt(positions.Count * (-1 * accuracyDefault)));
                int i = positions.ElementAt(indicies - 1).Value[0];
                int j = positions.ElementAt(indicies - 1).Value[1];
                return (pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
            }
        }
    }
    else
    {
        return GetMoveRandom(movesCurrentlyAvailable, pieceCoords);
    }
}

protected (Vector3Int, Vector3Int)
GetMoveMostValueFrom2LookAheadWithAccuracyAndLargeMoveNegativityMitigation(List<List<Vector3Int>> movesCurrentlyAvailable, List<Piece> pieces, List<Vector3Int> pieceCoords,
float accuracyDefault, float accuracyRange = 0f)
{
    List<List<Vector3Int>> opposingMovesCurrentlyAvailable = new
List<List<Vector3Int>>();
    List<Piece> opposingPieces = new List<Piece>();
    List<Vector3Int> opposingPieceCoords = new List<Vector3Int>();
    Piece oldPiece;

    SortedDictionary<float, int[]> positions = new SortedDictionary<float,
int[]>();
    float currentValue = 0;
    float small = 0.0001f; // prevents same keys
    for (int i = 0; i < movesCurrentlyAvailable.Count; i++)
    {
        for (int j = 0; j < movesCurrentlyAvailable.ElementAt(i).Count; j++)
        {
            oldPiece =
board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

            board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
pieceCoords.ElementAt(i), pieces.ElementAt(i), null);
            (opposingMovesCurrentlyAvailable, opposingPieces, opposingPieceCoords)
= opposingPlayer.ReturnAllPossibleMoves();

```

```

        currentValue =
board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j)) == null ? 0
:
pieceNameToValueDict[(board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j))).name];
        currentValue += BasicTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i), board.GetPieceOnSquare(pieceCoords.ElementAt(i)))
== null ? true : board.GetPieceOnSquare(pieceCoords.ElementAt(i)).team ==
TeamColor.White ? true : false);
        for (int k = 0; k < opposingMovesCurrentlyAvailable.Count; k++)
{
    for (int l = 0; l <
opposingMovesCurrentlyAvailable.ElementAt(k).Count; l++)
{
    currentValue -=
board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l)) ==
null ? 0 :
pieceNameToValueDict[(board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l))).name];
    currentValue -=
BasicTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves,
board.GetPieceOnSquare(opposingPieceCoords.ElementAt(k)) == null ? true :
board.GetPieceOnSquare(opposingPieceCoords.ElementAt(k)).team == TeamColor.White ?
false : true);
    while (positions.ContainsKey(currentValue)){currentValue ==
0.00001f;}positions.Add(currentValue, new int[] { i, j });
    small += 0.00001f;
}
}
}

board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
pieceCoords.ElementAt(i), oldPiece, pieces.ElementAt(i));
}
}
int numMoves = positions.Count-1;
Debug.Log(positions.ElementAt(numMoves).Key);
if (numMoves > 0)
{
    if (accuracyRange == 0)
    {
        if (accuracyDefault > 0)
        {
            if(positions.ElementAt(Mathf.FloorToInt(positions.Count -
Mathf.FloorToInt(positions.Count - (positions.Count * accuracyDefault))/2).Key < 0)
{
                // no need to create an additional list to only add values in
the accuracy range as the above takes care of this at the same time as the 50%
negative implementation discussed in 4.a.ii.7 in objectives
                // this is because we know that the first negative value must
lie within this range
                SortedDictionary<float, int[]> positivePositions = new
SortedDictionary<float, int[]>();
                foreach(KeyValuePair<float, int[]> item in positions)
{
                    if(item.Key >= 0)
{
                        positivePositions.Add(item.Key, item.Value);
}
}
int indecies = rnd.Next(0,positivePositions.Count);

```

```

                int i = positions.ElementAt(indicies - 1).Value[0];
                int j = positions.ElementAt(indicies - 1).Value[1];
                return (pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
            }
            else
            {
                int indicies = rnd.Next(Mathf.FloorToInt(positions.Count -
(positions.Count * accuracyDefault)), positions.Count);
                int i = positions.ElementAt(indicies - 1).Value[0];
                int j = positions.ElementAt(indicies - 1).Value[1];
                return (pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
            }
        }
        else
        {
            int indicies = rnd.Next(0, Mathf.FloorToInt(positions.Count * (-1 *
accuracyDefault)));
            int i = positions.ElementAt(indicies - 1).Value[0];
            int j = positions.ElementAt(indicies - 1).Value[1];
            return (pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
        }
    }
    else
    {
        if (accuracyDefault > 0)
        {
            int indicies = rnd.Next(Mathf.FloorToInt(positions.Count -
(positions.Count * accuracyRange)), Mathf.FloorToInt(positions.Count -
(positions.Count * accuracyDefault)));
            int i = positions.ElementAt(indicies - 1).Value[0];
            int j = positions.ElementAt(indicies - 1).Value[1];
            return (pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
        }
        else
        {
            int indicies = rnd.Next(Mathf.FloorToInt(positions.Count * (-1 *
accuracyRange)), Mathf.FloorToInt(positions.Count * (-1 * accuracyDefault)));
            int i = positions.ElementAt(indicies - 1).Value[0];
            int j = positions.ElementAt(indicies - 1).Value[1];
            return (pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
        }
    }
}
else
{
    return GetMoveRandom(movesCurrentlyAvailable, pieceCoords);
}
}

private float BasicTheoryAddition(Vector3Int toMoveCoords, List<Vector3Int>
attackCoords, bool whitePiece)
{
    float addition = 0f;
    if((toMoveCoords.x == 4 || toMoveCoords.x == 3)&&(toMoveCoords.y == 4 ||
toMoveCoords.y == 3))

```

```

    {
        addition += 0.4f;
    }
    if (whitePiece)
    {
        if(toMoveCoords.y != 0)
        {
            addition += 0.1f;
        }
    }
    else
    {
        if(toMoveCoords.y != 7)
        {
            addition += 0.1f;
        }
    }
    foreach(Vector3Int coords in attackCoords)
    {
        if ((coords.x == 4 || coords.x == 3) && (coords.y == 4 || coords.y == 3))
// middle
        {
            addition += 0.1f;
        }
    }
    return addition;
}

```

3.1.4.3 – TECHNICAL SOLUTION – ALGORITHMS – AI – 3

AI3 CLASS

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class AI3 : AIManager
{
    public AI3(ChessPlayer currentAiPlayer, ChessPlayer opposingPlayer, Board board)
    {
        this.currentAiPlayer = currentAiPlayer;
        this.board = board;
        this.opposingPlayer = opposingPlayer;
    }

    public override void MoveMakerController(ChessPlayer aiPlayer, ChessPlayer
opposingPlayer)
    {
        NewAiPlayers(aiPlayer, opposingPlayer);
        List<List<Vector3Int>> movesCurrentlyAvailable = new List<List<Vector3Int>>();
        List<Piece> pieces = new List<Piece>();
        List<Vector3Int> pieceCoords = new List<Vector3Int>();
        (movesCurrentlyAvailable, pieces, pieceCoords) =
currentAiPlayer.ReturnAllPossibleMoves();

        Vector3Int coordsOfPieceToMove = Vector3Int.zero;
        Vector3Int coordsToMoveTo = Vector3Int.zero;
        int randomVal = rnd.Next(0, 100);

        if (randomVal < 2)//2%
        {

```

```

        (coordsOfPieceToMove, coordsToMoveTo) =
GetMoveMostValueFrom2LookAheadWithAccuracy(movesCurrentlyAvailable, pieces,
pieceCoords, -0.15f);
        Debug.Log("Blunder");
    }
    else if (randomVal < 7)//2+5=7
    {
        (coordsOfPieceToMove, coordsToMoveTo) =
GetMoveMostValueFrom2LookAheadWithAccuracy(movesCurrentlyAvailable, pieces,
pieceCoords, -0.5f, -0.25f);
        Debug.Log("Innacuracy");
    }
    else
    {
        if(rnd.Next(2) == 1)
        {
            (coordsOfPieceToMove, coordsToMoveTo) =
GetMoveMostValueFrom2LookAheadWithAccuracyWithAdditionalTheory(movesCurrentlyAvailable
, pieces, pieceCoords, 0.15f);
        }
        else
        {
            (coordsOfPieceToMove, coordsToMoveTo) =
GetMoveMostValueFrom2LookAheadWithAccuracyAndDeeperLineEvaluationWithAdditionalTheory(
movesCurrentlyAvailable, pieces, pieceCoords, 0.15f);
        }
    }
}

board.AIMakeMove(coordsOfPieceToMove, coordsToMoveTo);

}
}

```

PARENT FUNCTIONS USED

[Shows powerful ideas that are associated with humans such as deeper line analysis and thinking positionally]

```

protected (Vector3Int, Vector3Int)
GetMoveMostValueFrom2LookAheadWithAccuracy(List<List<Vector3Int>>
movesCurrentlyAvailable, List<Piece> pieces, List<Vector3Int> pieceCoords, float
accuracyDefault, float accuracyRange = 0f)
{
    List<List<Vector3Int>> opposingMovesCurrentlyAvailable = new
List<List<Vector3Int>>();
    List<Piece> opposingPieces = new List<Piece>();
    List<Vector3Int> opposingPieceCoords = new List<Vector3Int>();
    Piece oldPiece;

    SortedDictionary<float, int[]> positions = new SortedDictionary<float,
int[]>();
    float currentValue = 0;
    float small = 0.00001f; // prevents same keys
    for (int i = 0; i < movesCurrentlyAvailable.Count; i++)
    {
        for (int j = 0; j < movesCurrentlyAvailable.ElementAt(i).Count; j++)
        {
            oldPiece =
board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
pieceCoords.ElementAt(i), pieces.ElementAt(i), null);
            (opposingMovesCurrentlyAvailable, opposingPieces, opposingPieceCoords)
= opposingPlayer.ReturnAllPossibleMoves();

```

```

        currentValue =
board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j)) == null ? 0
:
pieceNameToValueDict[(board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j))).name];
        currentValue +=

BasicTheoryAddition(pieceCoords.ElementAt(i),movesCurrentlyAvailable.ElementAt(i),board.GetPieceOnSquare(pieceCoords.ElementAt(i)) == null? true:
board.GetPieceOnSquare(pieceCoords.ElementAt(i)).team == TeamColor.White ?
true:false);
        for (int k = 0; k < opposingMovesCurrentlyAvailable.Count; k++)
{
    for (int l = 0; l <
opposingMovesCurrentlyAvailable.ElementAt(k).Count; l++)
    {
        currentValue -=
board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l)) ==
null ? 0 :
pieceNameToValueDict[(board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l))).name];
        currentValue -=

BasicTheoryAddition(opposingPieceCoords.ElementAt(k),opposingMovesCurrentlyAvailable.ElementAt(k), board.GetPieceOnSquare(pieceCoords.ElementAt(i)) == null ? true :
board.GetPieceOnSquare(pieceCoords.ElementAt(i)).team == TeamColor.White ? true :
false);
        while (positions.ContainsKey(currentValue)){currentValue -=
0.00001f;}positions.Add(currentValue, new int[] { i, j });
        small += 0.00001f;
    }
}
}

board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
pieceCoords.ElementAt(i), oldPiece, pieces.ElementAt(i));
}
}
int numMoves = positions.Count-1;

if (numMoves > 0)
{
    if (accuracyRange == 0)
    {
        if (accuracyDefault > 0)
        {
            int indecies = rnd.Next(Mathf.FloorToInt(positions.Count -
(positions.Count * accuracyDefault)), positions.Count);
            int i = positions.ElementAt(indecies - 1).Value[0];
            int j = positions.ElementAt(indecies - 1).Value[1];
            return (pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
        }
        else
        {
            int indecies = rnd.Next(0, Mathf.FloorToInt(positions.Count * (-1 *
accuracyDefault)));
            int i = positions.ElementAt(indecies - 1).Value[0];
            int j = positions.ElementAt(indecies - 1).Value[1];
            return (pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
        }
    }
    else
    {

```

```

        if (accuracyDefault > 0)
        {
            int indecies = rnd.Next(Mathf.FloorToInt(positions.Count -
(positions.Count * accuracyRange)), Mathf.FloorToInt(positions.Count -
(positions.Count * accuracyDefault)));
            int i = positions.ElementAt(indecies - 1).Value[0];
            int j = positions.ElementAt(indecies - 1).Value[1];
            return (pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
        }
        else
        {
            int indecies = rnd.Next(Mathf.FloorToInt(positions.Count * (-1 *
accuracyRange)), Mathf.FloorToInt(positions.Count * (-1 * accuracyDefault)));
            int i = positions.ElementAt(indecies - 1).Value[0];
            int j = positions.ElementAt(indecies - 1).Value[1];
            return (pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
        }
    }
    else
    {
        return GetMoveRandom(movesCurrentlyAvailable, pieceCoords);
    }
}

protected (Vector3Int, Vector3Int)
GetMoveMostValueFrom2LookAheadWithAccuracyAndDeeperLineEvaluationWithAdditionalTheory(
List<List<Vector3Int>> movesCurrentlyAvailable, List<Piece> pieces, List<Vector3Int>
pieceCoords, float accuracyDefault, float accuracyRange = 0f)
{
    List<List<Vector3Int>> opposingMovesCurrentlyAvailable = new
List<List<Vector3Int>>();
    List<Piece> opposingPieces = new List<Piece>();
    List<Vector3Int> opposingPieceCoords = new List<Vector3Int>();
    Piece oldPiece;

    SortedDictionary<float, int[]> positions = new SortedDictionary<float,
int[]>();
    List<(Vector3Int, Vector3Int)> furtherPositions = new List<(Vector3Int,
Vector3Int)>();
    float currentValue = 0;

    float small = 0.00001f; // prevents same keys
    for (int i = 0; i < movesCurrentlyAvailable.Count; i++)
    {
        for (int j = 0; j < movesCurrentlyAvailable.ElementAt(i).Count; j++)
        {
            oldPiece =
board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

            board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
pieceCoords.ElementAt(i), pieces.ElementAt(i), null);
            (opposingMovesCurrentlyAvailable, opposingPieces, opposingPieceCoords)
= opposingPlayer.ReturnAllPossibleMoves();
            currentValue =
board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j)) == null ? 0
:
(pieceNameToValueDict[(board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).Ele
mentAt(j))).name] + board.materialImbalance * 0.05f);
        }
    }
}

```

```

        currentValue += IntermediateTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
        if(board.majorPiecesMoved < 50)
        {
            currentValue +=
BasicOpeningTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
        }
        else if(board.majorPiecesTaken < 50)
        {
            currentValue +=
BasicMiddleGameTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
        }
        else
        {
            currentValue +=
BasicEndGameTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
        }
        for (int k = 0; k < opposingMovesCurrentlyAvailable.Count; k++)
{
    for (int l = 0; l <
opposingMovesCurrentlyAvailable.ElementAt(k).Count; l++)
    {
        currentValue -=
board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l)) ==
null ? 0 :
(pieceNameToValueDict[(board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementA
t(k).ElementAt(l))).name] + board.materialImbalance * 0.05f);
        currentValue -=
IntermediateTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(l));
        if (board.majorPiecesMoved < 50)
        {
            currentValue -=
BasicOpeningTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(l));
        }
        else if (board.majorPiecesTaken < 50)
        {
            currentValue -=
BasicMiddleGameTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(l));
        }
        else
        {
            currentValue -=
BasicEndGameTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(l));
        }
        while (positions.ContainsKey(currentValue)){currentValue -=
0.00001f;}positions.Add(currentValue, new int[] { i, j });
        small += 0.00001f;
    }
}
board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
pieceCoords.ElementAt(i), oldPiece, pieces.ElementAt(i));
}
}
int numMoves = positions.Count-1;

```

```

        if (numMoves > 0)
    {
        if (accuracyRange == 0)
        {
            if (accuracyDefault > 0)
            {
                for (int a = 0; a < 3; a++)
                {
                    int indecies = rnd.Next(Mathf.FloorToInt(positions.Count -
(positions.Count * accuracyDefault)), positions.Count);
                    int i = positions.ElementAt(indecies - 1).Value[0];
                    int j = positions.ElementAt(indecies - 1).Value[1];
                    furtherPositions.Add((pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j)));
                }
            }
            else
            {
                for (int a = 0; a < 3; a++)
                {
                    int indecies = rnd.Next(0, Mathf.FloorToInt(positions.Count *
(-1 * accuracyDefault)));
                    int i = positions.ElementAt(indecies - 1).Value[0];
                    int j = positions.ElementAt(indecies - 1).Value[1];
                    furtherPositions.Add((pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j)));
                }
            }
        }
        else
        {
            if (accuracyDefault > 0)
            {

                for (int a = 0; a < 3; a++)
                {
                    int indecies = rnd.Next(Mathf.FloorToInt(positions.Count -
(positions.Count * accuracyRange)), Mathf.FloorToInt(positions.Count -
(positions.Count * accuracyDefault)));
                    int i = positions.ElementAt(indecies - 1).Value[0];
                    int j = positions.ElementAt(indecies - 1).Value[1];
                    furtherPositions.Add((pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j)));
                }
            }
            else
            {
                for (int a = 0; a < 3; a++)
                {
                    int indecies = rnd.Next(Mathf.FloorToInt(positions.Count * (-1 *
accuracyRange)), Mathf.FloorToInt(positions.Count * (-1 * accuracyDefault)));
                    int i = positions.ElementAt(indecies - 1).Value[0];
                    int j = positions.ElementAt(indecies - 1).Value[1];
                    furtherPositions.Add((pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j)));
                }
            }
        }
    }
}

```

```

        List<(float, Vector3Int, Vector3Int)> furtherPositionsWithEval = new
List<(float, Vector3Int, Vector3Int)>();
        for (int a = 0; a < 3; a++)
{
    List<List<Vector3Int>> furtherTempMovesCurrentlyAvailable = new
List<List<Vector3Int>>();
    List<Vector3Int> miniFurtherTempMovesCurrentlyAvailable = new
List<Vector3Int>();
    List<Piece> furtherTempPieces = new List<Piece>();
    List<Vector3Int> furtherTempPieceCoords = new List<Vector3Int>();

miniFurtherTempMovesCurrentlyAvailable.Add(furtherPositions.ElementAt(a).Item2);

furtherTempMovesCurrentlyAvailable.Add(miniFurtherTempMovesCurrentlyAvailable);

furtherTempPieces.Add(board.GetPieceOnSquare(furtherPositions.ElementAt(a).Item1));
    furtherTempPieceCoords.Add(furtherPositions.ElementAt(a).Item1);

    (Vector3Int, Vector3Int) halfWayMoveChosen =
GetMoveMostValueFrom2LookAheadWithAccuracy(furtherTempMovesCurrentlyAvailable,
furtherTempPieces, furtherTempPieceCoords, 0.15f);

    miniFurtherTempMovesCurrentlyAvailable.Clear();
    furtherTempMovesCurrentlyAvailable.Clear();
    furtherTempPieces.Clear();
    furtherTempPieceCoords.Clear();
    miniFurtherTempMovesCurrentlyAvailable.Add(halfWayMoveChosen.Item2);

furtherTempMovesCurrentlyAvailable.Add(miniFurtherTempMovesCurrentlyAvailable);

furtherTempPieces.Add(board.GetPieceOnSquare(halfWayMoveChosen.Item1));
    furtherTempPieceCoords.Add(halfWayMoveChosen.Item1);

furtherPositionsWithEval.Add(GetMoveMostValueFrom2LookAheadWithAccuracyAndReturnEval(f
urtherTempMovesCurrentlyAvailable, furtherTempPieces, furtherTempPieceCoords, 0.15f));
}
float move1 = furtherPositionsWithEval.ElementAt(0).Item1;
float move2 = furtherPositionsWithEval.ElementAt(1).Item1;
float move3 = furtherPositionsWithEval.ElementAt(2).Item1;
if (move1 > move2)
{
    if (move1 > move3)
    {
        return (furtherPositionsWithEval.ElementAt(0).Item2,
furtherPositionsWithEval.ElementAt(0).Item3);
    }
    else
    {
        return (furtherPositionsWithEval.ElementAt(2).Item2,
furtherPositionsWithEval.ElementAt(2).Item3);
    }
}
else
{
    if (move2 > move3)
    {
        return (furtherPositionsWithEval.ElementAt(1).Item2,
furtherPositionsWithEval.ElementAt(1).Item3);
    }
    else

```

```

        {
            return (furtherPositionsWithEval.ElementAt(2).Item2,
furtherPositionsWithEval.ElementAt(2).Item3);
        }
    }
}
else
{
    return GetMoveRandom(movesCurrentlyAvailable, pieceCoords);
}

}

protected (float, Vector3Int, Vector3Int)
GetMoveMostValueFrom2LookAheadWithAccuracyAndReturnEval(List<List<Vector3Int>>
movesCurrentlyAvailable, List<Piece> pieces, List<Vector3Int> pieceCoords, float
accuracyDefault, float accuracyRange = 0f)
{
    List<List<Vector3Int>> opposingMovesCurrentlyAvailable = new
List<List<Vector3Int>>();
    List<Piece> opposingPieces = new List<Piece>();
    List<Vector3Int> opposingPieceCoords = new List<Vector3Int>();
    Piece oldPiece;

    SortedDictionary<float, int[]> positions = new SortedDictionary<float,
int[]>();
    float currentValue = 0;
    float small = 0.00001f; // prevents same keys
    for (int i = 0; i < movesCurrentlyAvailable.Count; i++)
    {
        for (int j = 0; j < movesCurrentlyAvailable.ElementAt(i).Count; j++)
        {
            oldPiece =
board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

            board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
pieceCoords.ElementAt(i), pieces.ElementAt(i), null);
            (opposingMovesCurrentlyAvailable, opposingPieces, opposingPieceCoords)
= opposingPlayer.ReturnAllPossibleMoves();
            currentValue =
            board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j)) == null ? 0
:
            pieceNameToValueDict[(board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).Elem
entAt(j))).name];
            currentValue += BasicTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i), board.GetPieceOnSquare(pieceCoords.ElementAt(i))
== null ? true : board.GetPieceOnSquare(pieceCoords.ElementAt(i)).team ==
TeamColor.White ? true : false);
            if (board.majorPiecesMoved < 50)
            {
                currentValue +=
BasicOpeningTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
            }
            else if (board.majorPiecesTaken < 50)
            {
                currentValue +=
BasicMiddleGameTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
            }
            else
            {

```

```

        currentValue +=
BasicEndGameTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
    }
    for (int k = 0; k < opposingMovesCurrentlyAvailable.Count; k++)
    {
        for (int l = 0; l <
opposingMovesCurrentlyAvailable.ElementAt(k).Count; l++)
        {
            currentValue -=
board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l)) ==
null ? 0 :
pieceNameToValueDict[(board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l))).name];
            currentValue -=
BasicTheoryAddition(opposingPieceCoords.ElementAt(k),
opposingMovesCurrentlyAvailable.ElementAt(k),
board.GetPieceOnSquare(pieceCoords.ElementAt(i)) == null ? true :
board.GetPieceOnSquare(pieceCoords.ElementAt(i)).team == TeamColor.White ? true :
false);
            if (board.majorPiecesMoved < 50)
            {
                currentValue -=
BasicOpeningTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(1));
            }
            else if (board.majorPiecesTaken < 50)
            {
                currentValue -=
BasicMiddleGameTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(1));
            }
            else
            {
                currentValue -=
BasicEndGameTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(1));
            }
            while (positions.ContainsKey(currentValue)) { currentValue -=
0.00001f; }
            positions.Add(currentValue, new int[] { i, j });
            small += 0.00001f;
        }
    }
}

board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
pieceCoords.ElementAt(i), oldPiece, pieces.ElementAt(i));
}
}
int numMoves = positions.Count-1;
if (numMoves > 0)
{
    if (accuracyRange == 0)
    {
        if (accuracyDefault > 0)
        {
            int indecies = rnd.Next(Mathf.FloorToInt(positions.Count -
(positions.Count * accuracyDefault)), positions.Count);
            int i = positions.ElementAt(indecies - 1).Value[0];
            int j = positions.ElementAt(indecies - 1).Value[1];
            return (positions.ElementAt(indecies).Key,
pieceCoords.ElementAt(i), movesCurrentlyAvailable.ElementAt(i).ElementAt(j)));
        }
    }
}

```

```

        }
        else
        {
            int indecies = rnd.Next(0, Mathf.FloorToInt(positions.Count * (-1
* accuracyDefault)));
            int i = positions.ElementAt(indecies - 1).Value[0];
            int j = positions.ElementAt(indecies - 1).Value[1];
            return (positions.ElementAt(indecies).Key,
pieceCoords.ElementAt(i), movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
        }
    }
    else
    {
        if (accuracyDefault > 0)
        {
            int indecies = rnd.Next(Mathf.FloorToInt(positions.Count -
(positions.Count * accuracyRange)), Mathf.FloorToInt(positions.Count -
(positions.Count * accuracyDefault)));
            int i = positions.ElementAt(indecies - 1).Value[0];
            int j = positions.ElementAt(indecies - 1).Value[1];
            return (positions.ElementAt(indecies).Key,
pieceCoords.ElementAt(i), movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
        }
        else
        {
            int indecies = rnd.Next(Mathf.FloorToInt(positions.Count * (-1 *
accuracyRange)), Mathf.FloorToInt(positions.Count * (-1 * accuracyDefault)));
            int i = positions.ElementAt(indecies - 1).Value[0];
            int j = positions.ElementAt(indecies - 1).Value[1];
            return (positions.ElementAt(indecies).Key,
pieceCoords.ElementAt(i), movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
        }
    }
}
else
{
    (Vector3Int, Vector3Int) tempHolder =
GetMoveRandom(movesCurrentlyAvailable, pieceCoords);
    return (-1f, tempHolder.Item1, tempHolder.Item2);
}

}

private float BasicOpeningTheoryAddition(Vector3Int coords, Vector3Int attackCoords)
{
    float addition = 0;
    Piece piece = board.GetPieceOnSquare(attackCoords);
    if (piece!= null)
    {
        addition -= pieceNameToValueDict[piece.name] *0.05f;
        if ((coords.x == 4 || coords.x == 3) && (coords.y == 4 || coords.y == 3))
// middle
        {
            addition += 1f;
        }
        piece = board.GetPieceOnSquare(coords);
        if (piece!= null && coords.z == board.getPieceCoords("King",
piece.team).z)
        {
            addition += 0.1f;
        }
    }
}

```

```

        return addition;
    }
    private float BasicMiddleGameTheoryAddition(Vector3Int toMoveCoords, Vector3Int
attackCoords)
    {
        float addition = 0;
        Piece piece = board.GetPieceOnSquare(attackCoords);
        if (piece != null)
        {
            addition += pieceNameToValueDict[piece.name] *0.05f;
        }
        return addition;
    }
    private float BasicEndGameTheoryAddition(Vector3Int toMoveCoords, Vector3Int
attackCoords)
    {
        float addition = 0;
        if (board.GetPieceOnSquare(toMoveCoords) != null)
        {
            if (board.GetPieceOnSquare(toMoveCoords).name == "Pawn" ||
board.GetPieceOnSquare(toMoveCoords).name == "Pawn(Clone)")
            {
                addition += 0.5f * attackCoords.y;
            }
            else if (board.GetPieceOnSquare(toMoveCoords).name == "King" ||
board.GetPieceOnSquare(toMoveCoords).name == "King(Clone)")
            {
                int numPiecesProtecting = 0;
                for (int i = -1; i < 2; i += 2)
                {
                    for (int j = -1; j < 2; j += 2)
                    {
                        for (int k = -1; k < 2; k += 2)
                        {
                            if (board.GetPieceOnSquare(new Vector3Int(attackCoords.x +
i, attackCoords.y + j, attackCoords.z + k)) != null)
                            {
                                numPiecesProtecting++;
                            }
                        }
                    }
                }
                addition += 0.05f * numPiecesProtecting;
            }
        }
        return addition;
    }

private float IntermediateTheoryAddition(Vector3Int toMoveCoords, Vector3Int
attackCoords)
{
    float addition = 0;
    Piece piece = board.GetPieceOnSquare(toMoveCoords);
    if (piece != null)
    {
        if (piece.name == "Knight" || piece.name == "Knight(Clone)")
        {
            if (attackCoords.x == 0 || attackCoords.x == 7 || attackCoords.y == 0
|| attackCoords.y == 7 || attackCoords.z == 0 || attackCoords.z == 7)
            {
                addition -= 0.3f;
            }
        }
    }
}

```

```

        }
    }
    else if (piece.name == "Rook" || piece.name == "Rook(Clone)")
    {
        if (attackCoords.x == 0 || attackCoords.x == 7 || attackCoords.y == 0
|| attackCoords.y == 7 || attackCoords.z == 0 || attackCoords.z == 7)
        {
            addition -= 0.3f;
        }
    }
}

return addition;
}

```

3.1.4.4 – TECHNICAL SOLUTION – ALGORITHMS – AI - 4

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class AI4 : AIManager
{
    public AI4 (ChessPlayer currentAiPlayer, ChessPlayer opposingPlayer, Board board)
    {
        this.currentAiPlayer = currentAiPlayer;
        this.board = board;
        this.opposingPlayer = opposingPlayer;
    }

    public override void MoveMakerController(ChessPlayer aiPlayer, ChessPlayer
opposingPlayer)
    {
        NewAiPlayers(aiPlayer, opposingPlayer);
        List<List<Vector3Int>> movesCurrentlyAvailable = new List<List<Vector3Int>>();
        List<Piece> pieces = new List<Piece>();
        List<Vector3Int> pieceCoords = new List<Vector3Int>();
        (movesCurrentlyAvailable, pieces, pieceCoords) =
currentAiPlayer.ReturnAllPossibleMoves();

        Vector3Int coordsOfPieceToMove = Vector3Int.zero;
        Vector3Int coordsToMoveTo = Vector3Int.zero;
        int randomVal = rnd.Next(0, 10000);

        if (randomVal < 75)//0.75%
        {
            (coordsOfPieceToMove, coordsToMoveTo) =
GetMoveMostValueFrom2LookAheadWithAccuracy(movesCurrentlyAvailable, pieces,
pieceCoords, -0.15f);
            Debug.Log("Blunder");
        }
        else if (randomVal < 300)//3%
        {
            (coordsOfPieceToMove, coordsToMoveTo) =
GetMoveMostValueFrom2LookAheadWithAccuracy(movesCurrentlyAvailable, pieces,
pieceCoords, -0.5f, -0.25f);
            Debug.Log("Innacuracy");
        }
        else
        {

```

```

        (coordsOfPieceToMove, coordsToMoveTo) =
GetMoveMostValueFrom2LookAheadAndDeeperLineEvaluationWithPositionalNeuralNetwork(moves
CurrentlyAvailable, pieces, pieceCoords);
    }

    board.AIMakeMove(coordsOfPieceToMove, coordsToMoveTo);

}
}

```

PARENT FUNCTIONS USED

```

protected (Vector3Int, Vector3Int)
GetMoveMostValueFrom2LookAheadWithAccuracy(List<List<Vector3Int>>
movesCurrentlyAvailable, List<Piece> pieces, List<Vector3Int> pieceCoords, float
accuracyDefault, float accuracyRange = 0f)
{
    List<List<Vector3Int>> opposingMovesCurrentlyAvailable = new
List<List<Vector3Int>>();
    List<Piece> opposingPieces = new List<Piece>();
    List<Vector3Int> opposingPieceCoords = new List<Vector3Int>();
    Piece oldPiece;

    SortedDictionary<float, int[]> positions = new SortedDictionary<float,
int[]>();
    float currentValue = 0;
    float small = 0.00001f; // prevents same keys
    for (int i = 0; i < movesCurrentlyAvailable.Count; i++)
    {
        for (int j = 0; j < movesCurrentlyAvailable.ElementAt(i).Count; j++)
        {
            oldPiece =
board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

            board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
pieceCoords.ElementAt(i), pieces.ElementAt(i), null);
                (opposingMovesCurrentlyAvailable, opposingPieces, opposingPieceCoords)
= opposingPlayer.ReturnAllPossibleMoves();
            currentValue =
board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j)) == null ? 0
:
pieceNameToValueDict[(board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).Elem
entAt(j))).name];
            currentValue +=
BasicTheoryAddition(pieceCoords.ElementAt(i), movesCurrentlyAvailable.ElementAt(i),boar
d.GetPieceOnSquare(pieceCoords.ElementAt(i)) == null? true:
board.GetPieceOnSquare(pieceCoords.ElementAt(i)).team == TeamColor.White ?
true:false);
                for (int k = 0; k < opposingMovesCurrentlyAvailable.Count; k++)
{
                    for (int l = 0; l <
opposingMovesCurrentlyAvailable.ElementAt(k).Count; l++)
{
                        currentValue -=
board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l)) ==
null ? 0 :
pieceNameToValueDict[(board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt
(k).ElementAt(l))).name];
                        currentValue -=
BasicTheoryAddition(opposingPieceCoords.ElementAt(k),opposingMovesCurrentlyAvailable.E
lementAt(k), board.GetPieceOnSquare(pieceCoords.ElementAt(i)) == null ? true :
board.GetPieceOnSquare(pieceCoords.ElementAt(i)).team == TeamColor.White ? true :
false);

```

```

                while (positions.ContainsKey(currentValue)){currentValue -=
0.00001f;}positions.Add(currentValue, new int[] { i, j });
                small += 0.00001f;
            }
        }

board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
pieceCoords.ElementAt(i), oldPiece, pieces.ElementAt(i));
    }
}

int numMoves = positions.Count-1;

if (numMoves > 0)
{
    if (accuracyRange == 0)
    {
        if (accuracyDefault > 0)
        {
            int indecies = rnd.Next(Mathf.FloorToInt(positions.Count -
(positions.Count * accuracyDefault)), positions.Count);
            int i = positions.ElementAt(indecies - 1).Value[0];
            int j = positions.ElementAt(indecies - 1).Value[1];
            return (pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
        }
        else
        {
            int indecies = rnd.Next(0, Mathf.FloorToInt(positions.Count * (-1 *
accuracyDefault)));
            int i = positions.ElementAt(indecies - 1).Value[0];
            int j = positions.ElementAt(indecies - 1).Value[1];
            return (pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
        }
    }
    else
    {
        if (accuracyDefault > 0)
        {
            int indecies = rnd.Next(Mathf.FloorToInt(positions.Count -
(positions.Count * accuracyRange)), Mathf.FloorToInt(positions.Count -
(positions.Count * accuracyDefault)));
            int i = positions.ElementAt(indecies - 1).Value[0];
            int j = positions.ElementAt(indecies - 1).Value[1];
            return (pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
        }
        else
        {
            int indecies = rnd.Next(Mathf.FloorToInt(positions.Count * (-1 *
accuracyRange)), Mathf.FloorToInt(positions.Count * (-1 * accuracyDefault)));
            int i = positions.ElementAt(indecies - 1).Value[0];
            int j = positions.ElementAt(indecies - 1).Value[1];
            return (pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
        }
    }
}
else
{
    return GetMoveRandom(movesCurrentlyAvailable, pieceCoords);
}

```

```

    }

    protected (Vector3Int, Vector3Int)
GetMoveMostValueFrom2LookAheadAndDeeperLineEvaluationWithPositionalNeuralNetwork(List<
List<Vector3Int>> movesCurrentlyAvailable, List<Piece> pieces, List<Vector3Int>
pieceCoords)
{
    List<List<Vector3Int>> opposingMovesCurrentlyAvailable = new
List<List<Vector3Int>>();
    List<Piece> opposingPieces = new List<Piece>();
    List<Vector3Int> opposingPieceCoords = new List<Vector3Int>();
    Piece oldPiece;
    Piece oldOPiece;

    SortedDictionary<float, int[]> positions = new SortedDictionary<float,
int[]>();
    List<(Vector3Int, Vector3Int)> furtherPositions = new List<(Vector3Int,
Vector3Int)>();
    float currentValue = 0;

    float small = 0.00001f; // prevents same keys
    for (int i = 0; i < movesCurrentlyAvailable.Count; i++)
    {
        for (int j = 0; j < movesCurrentlyAvailable.ElementAt(i).Count; j++)
        {
            oldPiece =
board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
pieceCoords.ElementAt(i), pieces.ElementAt(i), null);
            (opposingMovesCurrentlyAvailable, opposingPieces, opposingPieceCoords)
= opposingPlayer.ReturnAllPossibleMoves();
            currentValue =
board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j)) == null ? 0
:
(pieceNameToValueDict[(board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).El
ementAt(j))).name] + board.materialImbalance * 0.05f);
            currentValue += GetNNEvalFromBoardState(board.grid);
            if (board.majorPiecesMoved < 50)
            {
                currentValue +=
BasicOpeningTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
            }
            else if (board.majorPiecesTaken < 50)
            {
                currentValue +=
BasicMiddleGameTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
            }
            else
            {
                currentValue +=
BasicEndGameTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
            }
            for (int k = 0; k < opposingMovesCurrentlyAvailable.Count; k++)
            {
                for (int l = 0; l <
opposingMovesCurrentlyAvailable.ElementAt(k).Count; l++)
                {

```

```

                oldOPiece =
board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l));

board.UpdateBoardOnPieceMove(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l)
, opposingPieceCoords.ElementAt(k), opposingPieces.ElementAt(k), null);
        currentValue -=
board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l)) ==
null ? 0 :
pieceNameToValueDict[(board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l))).name];
        currentValue += GetNNEvalFromBoardState(board.grid);

board.UpdateBoardOnPieceMove(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l)
, opposingPieceCoords.ElementAt(k), oldOPiece, opposingPieces.ElementAt(k));
        if (board.majorPiecesMoved < 50)
{
        currentValue -=
BasicOpeningTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(l));
}
        else if (board.majorPiecesTaken < 50)
{
        currentValue -=
BasicMiddleGameTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(l));
}
        else
{
        currentValue -=
BasicEndGameTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(l));
}
        while (positions.ContainsKey(currentValue)) { currentValue -=
0.00001f; }
        positions.Add(currentValue, new int[] { i, j });
        small += 0.00001f;
    }
}

board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
pieceCoords.ElementAt(i), oldPiece, pieces.ElementAt(i));
}
int numMoves = positions.Count-1;
if (numMoves > 0)
{
    for (int a = 0; a < 5; a++)
{
    int i = positions.Last().Value[0];
    int j = positions.Last().Value[1];
    furtherPositions.Add((pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j)));
}
    List<float, Vector3Int, Vector3Int> furtherPositionsWithEval = new
List<float, Vector3Int, Vector3Int>();
    for (int a = 0; a < 5; a++)
{
    List<List<Vector3Int>> furtherTempMovesCurrentlyAvailable = new
List<List<Vector3Int>>();
        List<Vector3Int> miniFurtherTempMovesCurrentlyAvailable = new
List<Vector3Int>();
        List<Piece> furtherTempPieces = new List<Piece>();

```

```

        List<Vector3Int> furtherTempPieceCoords = new List<Vector3Int>();

miniFurtherTempMovesCurrentlyAvailable.Add(furtherPositions.ElementAt(a).Item2);

furtherTempMovesCurrentlyAvailable.Add(miniFurtherTempMovesCurrentlyAvailable);

furtherTempPieces.Add(board.GetPieceOnSquare(furtherPositions.ElementAt(a).Item1));
    furtherTempPieceCoords.Add(furtherPositions.ElementAt(a).Item1);

    (Vector3Int, Vector3Int) halfWayMoveChosen =
GetMoveMostValueFrom2LookAheadWithNN(furtherTempMovesCurrentlyAvailable,
furtherTempPieces, furtherTempPieceCoords);

        miniFurtherTempMovesCurrentlyAvailable.Clear();
        furtherTempMovesCurrentlyAvailable.Clear();
        furtherTempPieces.Clear();
        furtherTempPieceCoords.Clear();
        miniFurtherTempMovesCurrentlyAvailable.Add(halfWayMoveChosen.Item2);

furtherTempMovesCurrentlyAvailable.Add(miniFurtherTempMovesCurrentlyAvailable);

furtherTempPieces.Add(board.GetPieceOnSquare(halfWayMoveChosen.Item1));
    furtherTempPieceCoords.Add(halfWayMoveChosen.Item1);

furtherPositionsWithEval.Add(GetMoveMostValueFrom2LookAheadWithNNAndReturnEval(further
TempMovesCurrentlyAvailable, furtherTempPieces, furtherTempPieceCoords));
    }

    int maxIndex = 0;
    float max = 0;
    for (int i = 0; i < 5; i++)
    {
        if(furtherPositionsWithEval.ElementAt(i).Item1 > max)
        {
            maxIndex = i;
        }
    }
    return (furtherPositionsWithEval.ElementAt(maxIndex).Item2,
furtherPositionsWithEval.ElementAt(maxIndex).Item3);
}

else
{
    return GetMoveRandom(movesCurrentlyAvailable, pieceCoords);
}

}

protected (float,Vector3Int, Vector3Int)
GetMoveMostValueFrom2LookAheadWithNNAndReturnEval(List<List<Vector3Int>>
movesCurrentlyAvailable, List<Piece> pieces, List<Vector3Int> pieceCoords)
{
    List<List<Vector3Int>> opposingMovesCurrentlyAvailable = new
List<List<Vector3Int>>();
    List<Piece> opposingPieces = new List<Piece>();
    List<Vector3Int> opposingPieceCoords = new List<Vector3Int>();
    Piece oldPiece;
    Piece oldOPiece;

    SortedDictionary<float, int[]> positions = new SortedDictionary<float,
int[]>();
    float currentValue = 0;

```

```

        float small = 0.00001f; // prevents same keys
        for (int i = 0; i < movesCurrentlyAvailable.Count; i++)
        {
            for (int j = 0; j < movesCurrentlyAvailable.ElementAt(i).Count; j++)
            {
                oldPiece =
                    board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

                board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
                    pieceCoords.ElementAt(i), pieces.ElementAt(i), null);
                    (opposingMovesCurrentlyAvailable, opposingPieces, opposingPieceCoords)
                = opposingPlayer.ReturnAllPossibleMoves();
                currentValue =
                    board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j)) == null ? 0
                    :
                    pieceNameToValueDict[(board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j))).name];
                currentValue += GetNNEvalFromBoardState(board.grid);
                if (board.majorPiecesMoved < 50)
                {
                    currentValue +=
                    BasicOpeningTheoryAddition(pieceCoords.ElementAt(i),
                        movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
                }
                else if (board.majorPiecesTaken < 50)
                {
                    currentValue +=
                    BasicMiddleGameTheoryAddition(pieceCoords.ElementAt(i),
                        movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
                }
                else
                {
                    currentValue +=
                    BasicEndGameTheoryAddition(pieceCoords.ElementAt(i),
                        movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
                }
                for (int k = 0; k < opposingMovesCurrentlyAvailable.Count; k++)
                {
                    for (int l = 0; l <
                        opposingMovesCurrentlyAvailable.ElementAt(k).Count; l++)
                    {
                        oldOPiece =
                            board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l));

                        board.UpdateBoardOnPieceMove(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l),
                            opposingPieceCoords.ElementAt(k), opposingPieces.ElementAt(k), null);
                            currentValue -=
                            board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l)) ==
                            null ? 0 :
                            pieceNameToValueDict[(board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l))).name];
                            currentValue += GetNNEvalFromBoardState(board.grid);

                        board.UpdateBoardOnPieceMove(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l),
                            opposingPieceCoords.ElementAt(k), oldOPiece, opposingPieces.ElementAt(k));
                            if (board.majorPiecesMoved < 50)
                            {
                                currentValue -=
                                BasicOpeningTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
                                    opposingPieces.ElementAt(k).availableMoves.ElementAt(l));
                            }
                            else if (board.majorPiecesTaken < 50)

```

```

        {
            currentValue -=
BasicMiddleGameTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(l));
        }
        else
        {
            currentValue -=
BasicEndGameTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(l));
        }
        while (positions.ContainsKey(currentValue)){currentValue ==
0.00001f;}positions.Add(currentValue, new int[] { i, j });
            small += 0.00001f;
        }
    }

board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
pieceCoords.ElementAt(i), oldPiece, pieces.ElementAt(i));
}
}

int numMoves = positions.Count-1;
if (numMoves > 0)
{
    int i = positions.Last().Value[0];
    int j = positions.Last().Value[1];
    return (positions.Last().Key, pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
}
else
{
    (Vector3Int, Vector3Int) tempHolder =
GetMoveRandom(movesCurrentlyAvailable, pieceCoords);
    return (-1f, tempHolder.Item1,tempHolder.Item2);
}

}

protected (Vector3Int, Vector3Int)
GetMoveMostValueFrom2LookAheadWithNN(List<List<Vector3Int>> movesCurrentlyAvailable,
List<Piece> pieces, List<Vector3Int> pieceCoords)
{
    List<List<Vector3Int>> opposingMovesCurrentlyAvailable = new
List<List<Vector3Int>>();
    List<Piece> opposingPieces = new List<Piece>();
    List<Vector3Int> opposingPieceCoords = new List<Vector3Int>();
    Piece oldPiece;
    Piece oldOPiece;

    SortedDictionary<float, int[]> positions = new SortedDictionary<float,
int[]>();
    float currentValue = 0;
    float small = 0.00001f; // prevents same keys
    for (int i = 0; i < movesCurrentlyAvailable.Count; i++)
    {
        for (int j = 0; j < movesCurrentlyAvailable.ElementAt(i).Count; j++)
        {
            oldPiece =
board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
pieceCoords.ElementAt(i), pieces.ElementAt(i), null);

```

```

        (opposingMovesCurrentlyAvailable, opposingPieces, opposingPieceCoords)
= opposingPlayer.ReturnAllPossibleMoves();
        currentValue =
board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j)) == null ? 0
:
pieceNameToValueDict[(board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j))).name];
        currentValue += GetNNEvalFromBoardState(board.grid);
        if (board.majorPiecesMoved < 50)
{
        currentValue +=
BasicOpeningTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
}
        else if (board.majorPiecesTaken < 50)
{
        currentValue +=
BasicMiddleGameTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
}
        else
{
        currentValue +=
BasicEndGameTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
}
        for (int k = 0; k < opposingMovesCurrentlyAvailable.Count; k++)
{
        for (int l = 0; l <
opposingMovesCurrentlyAvailable.ElementAt(k).Count; l++)
{
        oldOPiece =
board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l));

board.UpdateBoardOnPieceMove(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(1),
, opposingPieceCoords.ElementAt(k), opposingPieces.ElementAt(k), null);
        currentValue -=
board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l)) ==
null ? 0 :
pieceNameToValueDict[(board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l))).name];
        currentValue += GetNNEvalFromBoardState(board.grid);

board.UpdateBoardOnPieceMove(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(1),
, opposingPieceCoords.ElementAt(k), oldOPiece, opposingPieces.ElementAt(k));
        if (board.majorPiecesMoved < 50)
{
        currentValue -=
BasicOpeningTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(1));
}
        else if (board.majorPiecesTaken < 50)
{
        currentValue -=
BasicMiddleGameTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(1));
}
        else
{
        currentValue -=
BasicEndGameTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(1));
}

```

```

        }
        while (positions.ContainsKey(currentValue)) { currentValue -=
0.00001f; }
            positions.Add(currentValue, new int[] { i, j });
            small += 0.00001f;
        }
    }

board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
pieceCoords.ElementAt(i), oldPiece, pieces.ElementAt(i));
}
int numMoves = positions.Count-1;
if (numMoves > 0)
{
    int i = positions.Last().Value[0];
    int j = positions.Last().Value[1];
    return (pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
}
else
{
    return GetMoveRandom(movesCurrentlyAvailable, pieceCoords);
}

}

private float GetNNEvalFromBoardState(Piece[,] grid)
{
    float[] inputs = new float[512*7*2];
    int counter = 0;
    for (int i = 0; i < 8; i++)
    {
        for (int j = 0; j < 8; j++)
        {
            for (int k = 0; k < 8; k++)
            {
                if (grid[i,j,k] != null)
                {
                    if(grid[i,j,k].team == TeamColor.White)
                    {
                        if (grid[i, j, k].name == "Pawn" || grid[i, j, k].name ==
"Pawn(Clone)")
                        {
                            inputs[counter] = 1;
                        }
                        else if (grid[i, j, k].name == "Knight" || grid[i, j,
k].name == "Knight(Clone)")
                        {
                            inputs[counter+1] = 1;
                        }
                        else if (grid[i, j, k].name == "Bishop" || grid[i, j,
k].name == "Bishop(Clone)")
                        {
                            inputs[counter+2] = 1;
                        }
                        else if (grid[i,j,k].name == "Rook" || grid[i, j, k].name ==
"Rook(Clone)")
                        {
                            inputs[counter+3] = 1;
                        }

```

```

        else if (grid[i,j,k].name == "Commoner" || grid[i, j,
k].name == "Commoner(Clone)")
{
    inputs[counter+4] = 1;
}
else if (grid[i, j, k].name == "King" || grid[i, j,
k].name == "King(Clone)")
{
    inputs[counter+5] = 1;
}
else if (grid[i, j, k].name == "Queen" || grid[i, j,
k].name == "Queen(Clone)")
{
    inputs[counter+6] = 1;
}
counter += 14;
}
else
{
    counter += 7;
if (grid[i, j, k].name == "Pawn" || grid[i, j, k].name ==
"Pawn(Clone)")
{
    inputs[counter] = 1;
}
else if (grid[i, j, k].name == "Knight" || grid[i, j,
k].name == "Knight(Clone)")
{
    inputs[counter + 1] = 1;
}
else if (grid[i, j, k].name == "Bishop" || grid[i, j,
k].name == "Bishop(Clone)")
{
    inputs[counter + 2] = 1;
}
else if (grid[i, j, k].name == "Rook" || grid[i, j,
k].name == "Rook(Clone)")
{
    inputs[counter + 3] = 1;
}
else if (grid[i, j, k].name == "Commoner" || grid[i, j,
k].name == "Commoner(Clone"))
{
    inputs[counter + 4] = 1;
}
else if (grid[i, j, k].name == "King" || grid[i, j,
k].name == "King(Clone)")
{
    inputs[counter + 5] = 1;
}
else if (grid[i, j, k].name == "Queen" || grid[i, j,
k].name == "Queen(Clone)")
{
    inputs[counter + 6] = 1;
}
counter += 7;
}
}
else
{
    counter += 14;
}

```

```

        }
    }
}
return networkManager.FeedForward(inputs)[0];//it is okay to do this as we
know there is only one output state (the eval)
}

```

3.1.4.5 – TECHNICAL SOLUTION – ALGORITHMS – AI – 5

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class AIS : AIManager
{
    public AIS (ChessPlayer currentAiPlayer, ChessPlayer opposingPlayer, Board board)
    {
        this.currentAiPlayer = currentAiPlayer;
        this.board = board;
        this.opposingPlayer = opposingPlayer;
    }

    public override void MoveMakerController(ChessPlayer aiPlayer, ChessPlayer
opposingPlayer)
    {
        NewAiPlayers(aiPlayer, opposingPlayer);
        List<List<Vector3Int>> movesCurrentlyAvailable = new List<List<Vector3Int>>();
        List<Piece> pieces = new List<Piece>();
        List<Vector3Int> pieceCoords = new List<Vector3Int>();
        (movesCurrentlyAvailable, pieces, pieceCoords) =
currentAiPlayer.ReturnAllPossibleMoves();

        Vector3Int coordsOfPieceToMove = Vector3Int.zero;
        Vector3Int coordsToMoveTo = Vector3Int.zero;
        (coordsOfPieceToMove, coordsToMoveTo) =
GetMoveMostValueFrom2LookAheadAndDeeperLineEvaluationWithPositionalNeuralNetwork(moves
CurrentlyAvailable, pieces, pieceCoords);

        board.AIMakeMove(coordsOfPieceToMove, coordsToMoveTo);
    }
}

```

PARENT FUNCTIONS USED

```

protected (Vector3Int, Vector3Int)
GetMoveMostValueFrom2LookAheadAndDeeperLineEvaluationWithPositionalNeuralNetwork(List<
List<Vector3Int>> movesCurrentlyAvailable, List<Piece> pieces, List<Vector3Int>
pieceCoords)
{
    List<List<Vector3Int>> opposingMovesCurrentlyAvailable = new
List<List<Vector3Int>>();
    List<Piece> opposingPieces = new List<Piece>();
    List<Vector3Int> opposingPieceCoords = new List<Vector3Int>();
    Piece oldPiece;
    Piece oldOPiece;

    SortedDictionary<float, int[]> positions = new SortedDictionary<float,
int[]>();
    List<(Vector3Int, Vector3Int)> furtherPositions = new List<(Vector3Int,
Vector3Int)>();
    float currentValue = 0;

```

```

        float small = 0.00001f; // prevents same keys
        for (int i = 0; i < movesCurrentlyAvailable.Count; i++)
        {
            for (int j = 0; j < movesCurrentlyAvailable.ElementAt(i).Count; j++)
            {
                oldPiece =
board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
pieceCoords.ElementAt(i), pieces.ElementAt(i), null);
            (opposingMovesCurrentlyAvailable, opposingPieces, opposingPieceCoords)
= opposingPlayer.ReturnAllPossibleMoves();
            currentValue =
board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j)) == null ? 0
:
(pieceNameToValueDict[(board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).Ele
mentAt(j))).name] + board.materialImbalance * 0.05f);
            currentValue += GetNNEvalFromBoardState(board.grid);
            if (board.majorPiecesMoved < 50)
            {
                currentValue +=
BasicOpeningTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
            }
            else if (board.majorPiecesTaken < 50)
            {
                currentValue +=
BasicMiddleGameTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
            }
            else
            {
                currentValue +=
BasicEndGameTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
            }
            for (int k = 0; k < opposingMovesCurrentlyAvailable.Count; k++)
            {
                for (int l = 0; l <
opposingMovesCurrentlyAvailable.ElementAt(k).Count; l++)
                {
                    oldOPiece =
board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l));

board.UpdateBoardOnPieceMove(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l)
, opposingPieceCoords.ElementAt(k), opposingPieces.ElementAt(k), null);
                    currentValue -=
board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l)) ==
null ? 0 :
pieceNameToValueDict[(board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt
(k).ElementAt(l))).name];
                    currentValue += GetNNEvalFromBoardState(board.grid);

board.UpdateBoardOnPieceMove(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l)
, opposingPieceCoords.ElementAt(k), oldOPiece, opposingPieces.ElementAt(k));
                    if (board.majorPiecesMoved < 50)
                    {
                        currentValue -=
BasicOpeningTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(l));
                    }
                }
            }
        }
    }
}

```

```

        else if (board.majorPiecesTaken < 50)
        {
            currentValue -=
BasicMiddleGameTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(1));
        }
        else
        {
            currentValue -=
BasicEndGameTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(1));
        }
        while (positions.ContainsKey(currentValue)) { currentValue -=
0.00001f; }
        positions.Add(currentValue, new int[] { i, j });
        small += 0.00001f;
    }
}

board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
pieceCoords.ElementAt(i), oldPiece, pieces.ElementAt(i));
}
}
int numMoves = positions.Count-1;
if (numMoves > 0)
{
    for (int a = 0; a < 5; a++)
    {
        int i = positions.Last().Value[0];
        int j = positions.Last().Value[1];
        furtherPositions.Add((pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j)));
    }
    List<float, Vector3Int, Vector3Int> furtherPositionsWithEval = new
List<float, Vector3Int, Vector3Int>();
    for (int a = 0; a < 5; a++)
    {
        List<List<Vector3Int>> furtherTempMovesCurrentlyAvailable = new
List<List<Vector3Int>>();
        List<Vector3Int> miniFurtherTempMovesCurrentlyAvailable = new
List<Vector3Int>();
        List<Piece> furtherTempPieces = new List<Piece>();
        List<Vector3Int> furtherTempPieceCoords = new List<Vector3Int>();

miniFurtherTempMovesCurrentlyAvailable.Add(furtherPositions.ElementAt(a).Item2);
furtherTempMovesCurrentlyAvailable.Add(miniFurtherTempMovesCurrentlyAvailable);

furtherTempPieces.Add(board.GetPieceOnSquare(furtherPositions.ElementAt(a).Item1));
        furtherTempPieceCoords.Add(furtherPositions.ElementAt(a).Item1);

        (Vector3Int, Vector3Int) halfWayMoveChosen =
GetMoveMostValueFrom2LookAheadWithNN(furtherTempMovesCurrentlyAvailable,
furtherTempPieces, furtherTempPieceCoords);

        miniFurtherTempMovesCurrentlyAvailable.Clear();
        furtherTempMovesCurrentlyAvailable.Clear();
        furtherTempPieces.Clear();
        furtherTempPieceCoords.Clear();
        miniFurtherTempMovesCurrentlyAvailable.Add(halfWayMoveChosen.Item2);

```

```

        furtherTempMovesCurrentlyAvailable.Add(miniFurtherTempMovesCurrentlyAvailable);

        furtherTempPieces.Add(board.GetPieceOnSquare(halfWayMoveChosen.Item1));
        furtherTempPieceCoords.Add(halfWayMoveChosen.Item1);

        furtherPositionsWithEval.Add(GetMoveMostValueFrom2LookAheadWithNNAndReturnEval(further
TempMovesCurrentlyAvailable, furtherTempPieces, furtherTempPieceCoords));
    }
    int maxIndex = 0;
    float max = 0;
    for (int i = 0; i < 5; i++)
    {
        if(furtherPositionsWithEval.ElementAt(i).Item1 > max)
        {
            maxIndex = i;
        }
    }
    return (furtherPositionsWithEval.ElementAt(maxIndex).Item2,
furtherPositionsWithEval.ElementAt(maxIndex).Item3);
}
else
{
    return GetMoveRandom(movesCurrentlyAvailable, pieceCoords);
}

}

protected (float,Vector3Int, Vector3Int)
GetMoveMostValueFrom2LookAheadWithNNAndReturnEval(List<List<Vector3Int>>
movesCurrentlyAvailable, List<Piece> pieces, List<Vector3Int> pieceCoords)
{
    List<List<Vector3Int>> opposingMovesCurrentlyAvailable = new
List<List<Vector3Int>>();
    List<Piece> opposingPieces = new List<Piece>();
    List<Vector3Int> opposingPieceCoords = new List<Vector3Int>();
    Piece oldPiece;
    Piece oldOPiece;

    SortedDictionary<float, int[]> positions = new SortedDictionary<float,
int[]>();
    float currentValue = 0;
    float small = 0.00001f; // prevents same keys
    for (int i = 0; i < movesCurrentlyAvailable.Count; i++)
    {
        for (int j = 0; j < movesCurrentlyAvailable.ElementAt(i).Count; j++)
        {
            oldPiece =
board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

            board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
pieceCoords.ElementAt(i), pieces.ElementAt(i), null);
            (opposingMovesCurrentlyAvailable, opposingPieces, opposingPieceCoords)
= opposingPlayer.ReturnAllPossibleMoves();
            currentValue =
board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j)) == null ? 0
:
pieceNameToValueDict[(board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).Elem
entAt(j))).name] ;
            currentValue += GetNNEvalFromBoardState(board.grid);
            if (board.majorPiecesMoved < 50)

```

```

        {
            currentValue +=
BasicOpeningTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
        }
        else if (board.majorPiecesTaken < 50)
{
    currentValue +=
BasicMiddleGameTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
}
else
{
    currentValue +=
BasicEndGameTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
}
for (int k = 0; k < opposingMovesCurrentlyAvailable.Count; k++)
{
    for (int l = 0; l <
opposingMovesCurrentlyAvailable.ElementAt(k).Count; l++)
    {
        oldOPiece =
board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l));

board.UpdateBoardOnPieceMove(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l),
, opposingPieceCoords.ElementAt(k), opposingPieces.ElementAt(k), null);
        currentValue -=
board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l)) ==
null ? 0 :
pieceNameToValueDict[(board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l))).name];
        currentValue += GetNNEvalFromBoardState(board.grid);

board.UpdateBoardOnPieceMove(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l),
, opposingPieceCoords.ElementAt(k), oldOPiece, opposingPieces.ElementAt(k));
        if (board.majorPiecesMoved < 50)
{
    currentValue -=
BasicOpeningTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(l));
}
        else if (board.majorPiecesTaken < 50)
{
    currentValue -=
BasicMiddleGameTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(l));
}
        else
{
    currentValue -=
BasicEndGameTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(l));
}
        while (positions.ContainsKey(currentValue)){currentValue ==
0.00001f;}positions.Add(currentValue, new int[] { i, j });
        small += 0.00001f;
    }
}
}

board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
pieceCoords.ElementAt(i), oldPiece, pieces.ElementAt(i));

```

```

        }
    }
    int numMoves = positions.Count-1;
    if (numMoves > 0)
    {
        int i = positions.Last().Value[0];
        int j = positions.Last().Value[1];
        return (positions.Last().Key, pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
    }
    else
    {
        (Vector3Int, Vector3Int) tempHolder =
GetMoveRandom(movesCurrentlyAvailable, pieceCoords);
        return (-1f, tempHolder.Item1,tempHolder.Item2);
    }
}

protected (Vector3Int, Vector3Int)
GetMoveMostValueFrom2LookAheadWithNN(List<List<Vector3Int>> movesCurrentlyAvailable,
List<Piece> pieces, List<Vector3Int> pieceCoords)
{
    List<List<Vector3Int>> opposingMovesCurrentlyAvailable = new
List<List<Vector3Int>>();
    List<Piece> opposingPieces = new List<Piece>();
    List<Vector3Int> opposingPieceCoords = new List<Vector3Int>();
    Piece oldPiece;
    Piece oldOPiece;

    SortedDictionary<float, int[]> positions = new SortedDictionary<float,
int[]>();
    float currentValue = 0;
    float small = 0.00001f; // prevents same keys
    for (int i = 0; i < movesCurrentlyAvailable.Count; i++)
    {
        for (int j = 0; j < movesCurrentlyAvailable.ElementAt(i).Count; j++)
        {
            oldPiece =
board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

            board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
pieceCoords.ElementAt(i), pieces.ElementAt(i), null);
            (opposingMovesCurrentlyAvailable, opposingPieces, opposingPieceCoords)
= opposingPlayer.ReturnAllPossibleMoves();
            currentValue =
board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j)) == null ? 0
:
pieceNameToValueDict[(board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).Elem
entAt(j))).name];
            currentValue += GetNNEvalFromBoardState(board.grid);
            if (board.majorPiecesMoved < 50)
            {
                currentValue +=
BasicOpeningTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
            }
            else if (board.majorPiecesTaken < 50)
            {
                currentValue +=
BasicMiddleGameTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
            }
        }
    }
}

```

```

        }
        else
        {
            currentValue +=
BasicEndGameTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
        }
        for (int k = 0; k < opposingMovesCurrentlyAvailable.Count; k++)
{
    for (int l = 0; l <
opposingMovesCurrentlyAvailable.ElementAt(k).Count; l++)
{
    oldOPiece =
board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l));

board.UpdateBoardOnPieceMove(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l)
, opposingPieceCoords.ElementAt(k), opposingPieces.ElementAt(k), null);
    currentValue -=
board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l)) ==
null ? 0 :
pieceNameToValueDict[(board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l))).name];
    currentValue += GetNNEvalFromBoardState(board.grid);

board.UpdateBoardOnPieceMove(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l)
, opposingPieceCoords.ElementAt(k), oldOPiece, opposingPieces.ElementAt(k));
    if (board.majorPiecesMoved < 50)
{
    currentValue -=
BasicOpeningTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(l));
}
    else if (board.majorPiecesTaken < 50)
{
    currentValue -=
BasicMiddleGameTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(l));
}
    else
{
    currentValue -=
BasicEndGameTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(l));
}
    while (positions.ContainsKey(currentValue)) { currentValue -=
0.00001f; }
    positions.Add(currentValue, new int[] { i, j });
    small += 0.00001f;
}
}
}

board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
pieceCoords.ElementAt(i), oldPiece, pieces.ElementAt(i));
}
}
int numMoves = positions.Count-1;
if (numMoves > 0)
{
    int i = positions.Last().Value[0];
    int j = positions.Last().Value[1];
    return (pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
}

```

```

        }
    else
    {
        return GetMoveRandom(movesCurrentlyAvailable, pieceCoords);
    }

}

private float GetNNEvalFromBoardState(Piece[,] grid)
{
    float[] inputs = new float[512*7*2];
    int counter = 0;
    for (int i = 0; i < 8; i++)
    {
        for (int j = 0; j < 8; j++)
        {
            for (int k = 0; k < 8; k++)
            {
                if (grid[i,j,k] != null)
                {
                    if(grid[i,j,k].team == TeamColor.White)
                    {
                        if (grid[i, j, k].name == "Pawn" || grid[i, j, k].name ==
"Pawn(Clone)")
                        {
                            inputs[counter] = 1;
                        }
                        else if (grid[i, j, k].name == "Knight" || grid[i, j,
k].name == "Knight(Clone)")
                        {
                            inputs[counter+1] = 1;
                        }
                        else if (grid[i, j, k].name == "Bishop" || grid[i, j,
k].name == "Bishop(Clone)")
                        {
                            inputs[counter+2] = 1;
                        }
                        else if (grid[i,j,k].name == "Rook" || grid[i, j, k].name ==
"Rook(Clone)")
                        {
                            inputs[counter+3] = 1;
                        }
                        else if (grid[i,j,k].name == "Commoner" || grid[i, j,
k].name == "Commoner(Clone)")
                        {
                            inputs[counter+4] = 1;
                        }
                        else if (grid[i, j, k].name == "King" || grid[i, j,
k].name == "King(Clone)")
                        {
                            inputs[counter+5] = 1;
                        }
                        else if (grid[i, j, k].name == "Queen" || grid[i, j,
k].name == "Queen(Clone)")
                        {
                            inputs[counter+6] = 1;
                        }
                        counter += 14;
                    }
                    else
                    {
                        counter += 7;
                    }
                }
            }
        }
    }
}

```

```

        if (grid[i, j, k].name == "Pawn" || grid[i, j, k].name ==
"Pawn(Clone)")
{
    inputs[counter] = 1;
}
else if (grid[i, j, k].name == "Knight" || grid[i, j,
k].name == "Knight(Clone)")
{
    inputs[counter + 1] = 1;
}
else if (grid[i, j, k].name == "Bishop" || grid[i, j,
k].name == "Bishop(Clone)")
{
    inputs[counter + 2] = 1;
}
else if (grid[i, j, k].name == "Rook" || grid[i, j,
k].name == "Rook(Clone)")
{
    inputs[counter + 3] = 1;
}
else if (grid[i, j, k].name == "Commoner" || grid[i, j,
k].name == "Commoner(Clone)")
{
    inputs[counter + 4] = 1;
}
else if (grid[i, j, k].name == "King" || grid[i, j,
k].name == "King(Clone)")
{
    inputs[counter + 5] = 1;
}
else if (grid[i, j, k].name == "Queen" || grid[i, j,
k].name == "Queen(Clone"))
{
    inputs[counter + 6] = 1;
}
counter += 7;
}

}
else
{
    counter += 14;
}
}
}
return networkManager.FeedForward(inputs)[0];//it is okay to do this as we
know there is only one output state (the eval)
}

```

3.1.4.1 – TECHNICAL SOLUTION – ALGORITHMS – AI – SPECIFIC FUNCTION LOGIC

This includes all of the functions used to calculate which moves are the best and which to be chosen – This is here to show functions that were once used and no longer are but were part of the testing process.

```
using System;
```

```
using System.Linq;
```

```
using System.Collections;
```

```

using System.Collections.Generic;
using UnityEngine;

public abstract class AIManager
{
    protected ChessPlayer currentAiPlayer {get; set;}
    protected ChessPlayer opposingPlayer { get; set; }
    protected Board board { get; set; }

    protected Dictionary<String, int> pieceNameToValueDict = new Dictionary<String, int>()
    {
        {"King", 1 }, //Point for putting king in check
        {"Pawn", 1 },
        {"Bishop", 3 },
        {"Knight", 3 },
        {"Commoner", 3 },
        {"Rook", 5 },
        {"Queen", 9 },

        {"King(Clone)",1 },
        {"Pawn(Clone)", 1 },
        {"Bishop(Clone)", 3 },
        {"Knight(Clone)", 3 },
        {"Commoner(Clone)", 3 },
        {"Rook(Clone)", 5 },
        {"Queen(Clone)", 9 },
    };
}

protected static int[] layers = new int[] {1,1,1,1 };

```

```

protected float[] inputState;

protected NNManager networkManager = new NNManager(layers,activationType.tanh,
costType.tanhCustom);

protected RandomNumber rnd = new RandomNumber();

protected int movesMade;

protected void NewAiPlayers (ChessPlayer newAiPlayer, ChessPlayer opposingPlayer)
{
    this.currentAiPlayer = newAiPlayer;
    this.opposingPlayer = opposingPlayer;
}

public abstract void MoveMakerController(ChessPlayer aiPlayer, ChessPlayer opposingPlayer);

protected (Vector3Int, Vector3Int) GetMoveRandom(List<List<Vector3Int>> movesCurrentlyAvailable,
List<Vector3Int> pieceCoords)
{
    int val = Guid.NewGuid().GetHashCode() % pieceCoords.Count;

    int atIndex = val > 0 ? val : -val;

    val = Guid.NewGuid().GetHashCode() % movesCurrentlyAvailable.ElementAt(atIndex).Count;

    int toIndex = val > 0 ? val : -val;

    return (pieceCoords.ElementAt(atIndex),
movesCurrentlyAvailable.ElementAt(atIndex).ElementAt(toIndex));
}

private float BasicTheoryAddition(Vector3Int toMoveCoords, List<Vector3Int> attackCoords, bool
whitePiece)
{
    float addition = 0f;

    if((toMoveCoords.x == 4 || toMoveCoords.x == 3)&&(toMoveCoords.y == 4 || toMoveCoords.y == 3))
    {

```

```

        addition += 0.4f;
    }
    if (whitePiece)
    {
        if(toMoveCoords.y != 0)
        {
            addition += 0.1f;
        }
    }
    else
    {
        if(toMoveCoords.y != 7)
        {
            addition += 0.1f;
        }
    }
}

foreach(Vector3Int coords in attackCoords)
{
    if ((coords.x == 4 || coords.x == 3) && (coords.y == 4 || coords.y == 3)) // middle
    {
        addition += 0.1f;
    }
}
return addition;
}

```

```

private float BasicOpeningTheoryAddition(Vector3Int coords, Vector3Int attackCoords)
{
    float addition = 0;

    Piece piece = board.GetPieceOnSquare(attackCoords);

    if (piece!= null)
    {
        addition -= pieceNameToValueDict[piece.name] *0.05f;

        if ((coords.x == 4 || coords.x == 3) && (coords.y == 4 || coords.y == 3)) // middle
        {
            addition += 1f;
        }
    }

    piece = board.GetPieceOnSquare(coords);
}

```

```

        if (piece != null && coords.z == board.getPieceCoords("King", piece.team).z)
    {
        addition += 0.1f;
    }
}

return addition;
}

private float BasicMiddleGameTheoryAddition(Vector3Int toMoveCoords, Vector3Int attackCoords)
{
    float addition = 0;

    Piece piece = board.GetPieceOnSquare(attackCoords);

    if (piece != null)
    {
        addition += pieceNameToValueDict[piece.name] *0.05f;
    }

    return addition;
}

private float BasicEndGameTheoryAddition(Vector3Int toMoveCoords, Vector3Int attackCoords)
{
    float addition = 0;

    if (board.GetPieceOnSquare(toMoveCoords) != null)
    {
        if (board.GetPieceOnSquare(toMoveCoords).name == "Pawn" ||
board.GetPieceOnSquare(toMoveCoords).name == "Pawn(Clone)")

        {
            addition += 0.5f * attackCoords.y;
        }

        else if (board.GetPieceOnSquare(toMoveCoords).name == "King" ||
board.GetPieceOnSquare(toMoveCoords).name == "King(Clone)")

        {
            int numPiecesProtecting = 0;

```

```

        for (int i = -1; i < 2; i += 2)
    {
        for (int j = -1; j < 2; j += 2)
        {
            for (int k = -1; k < 2; k += 2)
            {
                if (board.GetPieceOnSquare(new Vector3Int(attackCoords.x + i, attackCoords.y + j,
attackCoords.z + k)) != null)
                {
                    numPiecesProtecting++;
                }
            }
        }
    }

    addition += 0.05f * numPiecesProtecting;
}

}

return addition;
}

private float IntermediateTheoryAddition(Vector3Int toMoveCoords, Vector3Int attackCoords)
{
    float addition = 0;

    Piece piece = board.GetPieceOnSquare(toMoveCoords);

    if (piece != null)
    {
        if (piece.name == "Knight" || piece.name == "Knight(Clone)")

        {
            if (attackCoords.x == 0 || attackCoords.x == 7 || attackCoords.y == 0 || attackCoords.y == 7 ||
attackCoords.z == 0 || attackCoords.z == 7)

```

```

    {
        addition -= 0.3f;
    }
}

else if (piece.name == "Rook" || piece.name == "Rook(Clone)")

{
    if (attackCoords.x == 0 || attackCoords.x == 7 || attackCoords.y == 0 || attackCoords.y == 7 ||
attackCoords.z == 0 || attackCoords.z == 7)

    {
        addition -= 0.3f;
    }
}

}

return addition;
}

protected (Vector3Int, Vector3Int) GetMoveMostValueFromILookAhead(List<List<Vector3Int>>
movesCurrentlyAvailable, List<Vector3Int> pieceCoords)

{
    List<int[]> maxValuePositions = new List<int[]>();
    int maxValue = 0;
    int currentValue = 0;
    for (int i = 0; i < movesCurrentlyAvailable.Count; i++)
    {
        for (int j = 0; j < movesCurrentlyAvailable.ElementAt(i).Count; j++)
        {
            currentValue = board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j)) ==
null ? -1 : pieceNameToValueDict[(board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j))).name];
            if (currentValue > maxValue)

```

```

    {
        maxValuePositions.Clear();
        maxValuePositions.Add(new int[] { i, j });
    }

    if(currentValue == maxValue)
    {
        maxValuePositions.Add(new int[] { i, j });
    }
}

int numBestMoves = maxValuePositions.Count;

if (numBestMoves > 0)
{
    int indecies = rnd.Next(maxValuePositions.Count);

    return (pieceCoords.ElementAt(maxValuePositions.ElementAt(indecies)[0]),
movesCurrentlyAvailable.ElementAt(maxValuePositions.ElementAt(indecies)[0]).ElementAt(maxValuePositions.
ElementAt(indecies)[1]));
}

else
{
    return GetMoveRandom(movesCurrentlyAvailable, pieceCoords);
}
}

private List<(float, int[])> MergeSort(List<(float, int[])> nums)
{
    int count = nums.Count;
    if (count <= 1)
    {
        return nums;
    }
    else
    {
        // splitting
        int middle = count / 2; // truncation - will be left mid if even number of array elements

        List<(float, int[])> left = new List<(float, int[])>();

```

```

List<(float, int[])> right = new List<(float, int[])>();
List<(float, int[])> mergedList = new List<(float, int[])>();

for (int i = 0; i < middle; i++)
{
    left.Add(nums[i]);
}
for (int i = middle; i < count; i++)
{
    right.Add(nums[i]);
}

//recursively split and sort for smaller and smaller list sizes until base case met
left = MergeSort(left);
right = MergeSort(right);

// rebuilding back up bigger and bigger lists until back to base function call
while (left.Count > 0 || right.Count > 0)
{
    if (right.Count > 0)
    {
        if (left.Count > 0)
        {
            if (left[0].Item1 < right[0].Item1)
            {
                mergedList.Add(left[0]); // add left if smaller than right at smallest index
                left.RemoveAt(0);
            }
            else
            {
                mergedList.Add(right[0]); //add right if smaller or equal left at smallest index
                right.RemoveAt(0);
            }
        }
        else
        {
            foreach ((float, int[]) item in right) // if no left remaining fill list with right
            {
                mergedList.Add(item);
            }
            return mergedList;
        }
    }
    else
    {
        foreach ((float, int[]) item in left)// if no right fill list with left
        {
            mergedList.Add(item);
        }
        return mergedList;
    }
}
return new List<(float, int[])>() { (-1f, new int[0]) };
}
}

```

```

protected (Vector3Int, Vector3Int) GetMoveFrom1LookAheadWithAccuracy(List<List<Vector3Int>>
movesCurrentlyAvailable, List<Vector3Int> pieceCoords, float accuracy)
{
    List<(float,int[])> positions = new List<(float, int[])>();
    float currentValue = 0;
    for (int i = 0; i < movesCurrentlyAvailable.Count; i++)
    {
        for (int j = 0; j < movesCurrentlyAvailable.ElementAt(i).Count; j++)
        {
            currentValue = board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j)) ==
null ? 0 :
pieceNameToValueDict[(board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j))).name];
            positions.Add((currentValue, new int[] { i, j }));
        }
    }
    positions = MergeSort(positions); // orders list by lowest evaluation
    int numMoves = positions.Count - 1;
    if (numMoves > 0)
    {
        if (accuracy > 0)
        {
            int indecies = rnd.Next(Mathf.FloorToInt(numMoves - (numMoves * accuracy)), numMoves);
            int i = positions.ElementAt(indecies).Item2[0];
            int j = positions.ElementAt(indecies).Item2[1];
            return (pieceCoords.ElementAt(i), movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
        }
        else
        {
            int indecies = rnd.Next(0, Mathf.FloorToInt((numMoves)*(-1*accuracy)));
            int i = positions.ElementAt(indecies).Item2[0];
            int j = positions.ElementAt(indecies).Item2[1];
            return (pieceCoords.ElementAt(i), movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
        }
    }
    else
    {
        return GetMoveRandom(movesCurrentlyAvailable, pieceCoords);
    }
}
}

```

```

protected (Vector3Int, Vector3Int) GetMoveMostValueFrom2LookAhead(List<List<Vector3Int>>
movesCurrentlyAvailable, List<Piece> pieces, List<Vector3Int> pieceCoords)

```

```

{
    if(board.pastMovesBlack.Count > 3)

    {
        if (currentAiPlayer.team == TeamColor.White)
        {

```

```

        if (board.pastMovesWhite.ElementAt(board.pastMovesWhite.Count - 1) ==
board.pastMovesWhite.ElementAt(board.pastMovesWhite.Count - 3))

    {

        return GetMoveRandom(movesCurrentlyAvailable, pieceCoords);

    }

}

else

{

    if (board.pastMovesBlack.ElementAt(board.pastMovesBlack.Count - 1) ==
board.pastMovesBlack.ElementAt(board.pastMovesBlack.Count - 3))

    {

        return GetMoveRandom(movesCurrentlyAvailable, pieceCoords);

    }

}

}

List<List<Vector3Int>> opposingMovesCurrentlyAvailable = new List<List<Vector3Int>>();

List<Piece> opposingPieces = new List<Piece>();

List<Vector3Int> opposingPieceCoords = new List<Vector3Int>();

Piece oldPiece;

List<int[]> maxValuePositions = new List<int[]>();

int maxValue = 0;

int currentValue = 0;

for (int i = 0; i < movesCurrentlyAvailable.Count; i++)

{

    for (int j = 0; j < movesCurrentlyAvailable.ElementAt(i).Count; j++)

    {

        oldPiece = board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

        board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
pieceCoords.ElementAt(i), pieces.ElementAt(i), null );

        (opposingMovesCurrentlyAvailable, opposingPieces, opposingPieceCoords) =
opposingPlayer.ReturnAllPossibleMoves();
    }
}

```

```

        currentValue = board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j)) ==
    null ? 0 :
    pieceNameToValueDict[(board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j))).name];
}

for (int k = 0; k < opposingMovesCurrentlyAvailable.Count; k++)
{
    for (int l = 0; l < opposingMovesCurrentlyAvailable.ElementAt(k).Count; l++)
    {
        currentValue -=
    board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l)) == null ? 0 :
    pieceNameToValueDict[(board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l))).name];
    }

    if (currentValue > maxValue)
    {
        maxValuePositions.Clear();
        maxValuePositions.Add(new int[] { i, j });
    }

    if (currentValue == maxValue)
    {
        maxValuePositions.Add(new int[] { i, j });
    }
}

board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
pieceCoords.ElementAt(i), oldPiece, pieces.ElementAt(i));
}

int numBestMoves = maxValuePositions.Count;
if (numBestMoves > 0)
{
    int val = Guid.NewGuid().GetHashCode() % numBestMoves;
    val = val > 0 ? val : -val;
}

```

```

        return (pieceCoords.ElementAt(maxValuePositions.ElementAt(val)[0]),
movesCurrentlyAvailable.ElementAt(maxValuePositions.ElementAt(val)[0]).ElementAt(maxValuePositions.Elem
entAt(val)[1]));
    }

    else

    {

        return GetMoveRandom(movesCurrentlyAvailable, pieceCoords);

    }

}

protected (Vector3Int, Vector3Int)
GetMoveMostValueFrom2LookAheadWithAccuracy(List<List<Vector3Int>> movesCurrentlyAvailable,
List<Piece> pieces, List<Vector3Int> pieceCoords, float accuracyDefault, float accuracyRange = 0f)

{
    List<List<Vector3Int>> opposingMovesCurrentlyAvailable = new List<List<Vector3Int>>();
    List<Piece> opposingPieces = new List<Piece>();
    List<Vector3Int> opposingPieceCoords = new List<Vector3Int>();
    Piece oldPiece;

    SortedDictionary<float, int[]> positions = new SortedDictionary<float, int[]>();
    float currentValue = 0;
    float small = 0.00001f; // prevents same keys
    for (int i = 0; i < movesCurrentlyAvailable.Count; i++)
    {
        for (int j = 0; j < movesCurrentlyAvailable.ElementAt(i).Count; j++)
        {
            oldPiece = board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
            board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
pieceCoords.ElementAt(i), pieces.ElementAt(i), null);
            (opposingMovesCurrentlyAvailable, opposingPieces, opposingPieceCoords) =
opposingPlayer.ReturnAllPossibleMoves();
        }
    }
}

```

```

        currentValue = board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j)) ==
    null ? 0 :
    pieceNameToValueDict[(board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j))).name];
}

currentValue +=
BasicTheoryAddition(pieceCoords.ElementAt(i),movesCurrentlyAvailable.ElementAt(i),board.GetPieceOnSqua
re(pieceCoords.ElementAt(i)) == null? true: board.GetPieceOnSquare(pieceCoords.ElementAt(i)).team ==
TeamColor.White ? true:false);

for (int k = 0; k < opposingMovesCurrentlyAvailable.Count; k++)
{
    for (int l = 0; l < opposingMovesCurrentlyAvailable.ElementAt(k).Count; l++)
    {
        currentValue -=
board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l)) == null ? 0 :
pieceNameToValueDict[(board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt
(l))).name];

        currentValue -=
BasicTheoryAddition(opposingPieceCoords.ElementAt(k),opposingMovesCurrentlyAvailable.ElementAt(k),
board.GetPieceOnSquare(pieceCoords.ElementAt(i)) == null ? true :
board.GetPieceOnSquare(pieceCoords.ElementAt(i)).team == TeamColor.White ? true : false);

        while (positions.ContainsKey(currentValue)){currentValue -=
0.00001f;}positions.Add(currentValue, new int[] { i, j });

        small += 0.00001f;
    }
}
}

board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
pieceCoords.ElementAt(i), oldPiece, pieces.ElementAt(i));
}

}

int numMoves = positions.Count-1;

if (numMoves > 0)
{
    if (accuracyRange == 0)
    {
        if (accuracyDefault > 0)
        {

```

```

        int indecies = rnd.Next(Mathf.FloorToInt(positions.Count - (positions.Count * accuracyDefault)),
positions.Count);

        int i = positions.ElementAt(indecies - 1).Value[0];

        int j = positions.ElementAt(indecies - 1).Value[1];

        return (pieceCoords.ElementAt(i), movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

    }

    else

    {

        int indecies = rnd.Next(0, Mathf.FloorToInt(positions.Count * (-1 * accuracyDefault)));

        int i = positions.ElementAt(indecies - 1).Value[0];

        int j = positions.ElementAt(indecies - 1).Value[1];

        return (pieceCoords.ElementAt(i), movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

    }

}

else

{

    if (accuracyDefault > 0)

    {

        int indecies = rnd.Next(Mathf.FloorToInt(positions.Count - (positions.Count * accuracyRange)),
Mathf.FloorToInt(positions.Count - (positions.Count * accuracyDefault)));

        int i = positions.ElementAt(indecies - 1).Value[0];

        int j = positions.ElementAt(indecies - 1).Value[1];

        return (pieceCoords.ElementAt(i), movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

    }

    else

    {

        int indecies = rnd.Next(Mathf.FloorToInt(positions.Count * (-1 * accuracyRange)),
Mathf.FloorToInt(positions.Count * (-1 * accuracyDefault)));

        int i = positions.ElementAt(indecies - 1).Value[0];

        int j = positions.ElementAt(indecies - 1).Value[1];

        return (pieceCoords.ElementAt(i), movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

    }

}

```

```

        }

    }

}

else

{

    return GetMoveRandom(movesCurrentlyAvailable, pieceCoords);

}

}

protected (Vector3Int, Vector3Int) GetMoveMostValueFrom2LookAheadWithNN(List<List<Vector3Int>>
movesCurrentlyAvailable, List<Piece> pieces, List<Vector3Int> pieceCoords)

{
    List<List<Vector3Int>> opposingMovesCurrentlyAvailable = new List<List<Vector3Int>>();

    List<Piece> opposingPieces = new List<Piece>();

    List<Vector3Int> opposingPieceCoords = new List<Vector3Int>();

    Piece oldPiece;

    Piece oldOPiece;

    SortedDictionary<float, int[]> positions = new SortedDictionary<float, int[]>();

    float currentValue = 0;

    float small = 0.00001f; // prevents same keys

    for (int i = 0; i < movesCurrentlyAvailable.Count; i++)

    {

        for (int j = 0; j < movesCurrentlyAvailable.ElementAt(i).Count; j++)

        {

            oldPiece = board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

            board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
pieceCoords.ElementAt(i), pieces.ElementAt(i), null);

            (opposingMovesCurrentlyAvailable, opposingPieces, opposingPieceCoords) =
opposingPlayer.ReturnAllPossibleMoves();

```

```

        currentValue = board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j)) ==
null ? 0 :
pieceNameToValueDict[(board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j))).name
];

        currentValue += GetNNEvalFromBoardState(board.grid);

        if (board.majorPiecesMoved < 50)

        {

            currentValue += BasicOpeningTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

        }

        else if (board.majorPiecesTaken < 50)

        {

            currentValue += BasicMiddleGameTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

        }

        else

        {

            currentValue += BasicEndGameTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

        }

        for (int k = 0; k < opposingMovesCurrentlyAvailable.Count; k++)

        {

            for (int l = 0; l < opposingMovesCurrentlyAvailable.ElementAt(k).Count; l++)

            {

                oldOPiece =
board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l));

                board.UpdateBoardOnPieceMove(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l),
opposingPieceCoords.ElementAt(k), opposingPieces.ElementAt(k), null);

                currentValue -=
board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l)) == null ? 0 :
pieceNameToValueDict[(board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l))).name];

                currentValue += GetNNEvalFromBoardState(board.grid);

```

```

board.UpdateBoardOnPieceMove(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l),
opposingPieceCoords.ElementAt(k), oldOPiece, opposingPieces.ElementAt(k));

if (board.majorPiecesMoved < 50)

{

    currentValue -=
BasicOpeningTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(l));

}

else if (board.majorPiecesTaken < 50)

{

    currentValue -=
BasicMiddleGameTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(l));

}

else

{

    currentValue -=
BasicEndGameTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(l));

}

while (positions.ContainsKey(currentValue)) { currentValue -= 0.00001f; }

positions.Add(currentValue, new int[] { i, j });

small += 0.00001f;

}

}

board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
pieceCoords.ElementAt(i), oldPiece, pieces.ElementAt(i));

}

int numMoves = positions.Count-1;

if (numMoves > 0)

{

    int i = positions.Last().Value[0];

```

```

        int j = positions.Last().Value[1];

        return (pieceCoords.ElementAt(i), movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

    }

    else

    {

        return GetMoveRandom(movesCurrentlyAvailable, pieceCoords);

    }

}

protected (Vector3Int, Vector3Int)
GetMoveMostValueFrom2LookAheadWithAccuracyWithAdditionalTheory(List<List<Vector3Int>>
movesCurrentlyAvailable, List<Piece> pieces, List<Vector3Int> pieceCoords, float accuracyDefault, float
accuracyRange = 0f)

{

    List<List<Vector3Int>> opposingMovesCurrentlyAvailable = new List<List<Vector3Int>>();

    List<Piece> opposingPieces = new List<Piece>();

    List<Vector3Int> opposingPieceCoords = new List<Vector3Int>();

    Piece oldPiece;

    SortedDictionary<float, int[]> positions = new SortedDictionary<float, int[]>();

    float currentValue = 0;

    float small = 0.00001f; // prevents same keys

    for (int i = 0; i < movesCurrentlyAvailable.Count; i++)

    {

        for (int j = 0; j < movesCurrentlyAvailable.ElementAt(i).Count; j++)

        {

            oldPiece = board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

            board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
            pieceCoords.ElementAt(i), pieces.ElementAt(i), null);

            (opposingMovesCurrentlyAvailable, opposingPieces, opposingPieceCoords) =
            opposingPlayer.ReturnAllPossibleMoves();

```

```

        currentValue = board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j)) ==
null ? 0 :
(pieceNameToValueDict[(board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j))).name]
e] + board.materialImbalance*0.05f);

        currentValue += IntermediateTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

        if (board.majorPiecesMoved < 50)

        {

            currentValue += BasicOpeningTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

        }

        else if (board.majorPiecesTaken < 50)

        {

            currentValue += BasicMiddleGameTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

        }

        else

        {

            currentValue += BasicEndGameTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

        }

        for (int k = 0; k < opposingMovesCurrentlyAvailable.Count; k++)

        {

            for (int l = 0; l < opposingMovesCurrentlyAvailable.ElementAt(k).Count; l++)

            {

                currentValue -=

board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l)) == null ? 0 :
(pieceNameToValueDict[(board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l))).name]
t(l))] +board.materialImbalance * 0.05f;

                currentValue -= IntermediateTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(l));

                if (board.majorPiecesMoved < 50)

                {

```

```

        currentValue -=
BasicOpeningTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(l));

    }

    else if (board.majorPiecesTaken < 50)

    {

        currentValue -=
BasicMiddleGameTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(l));

    }

    else

    {

        currentValue -=
BasicEndGameTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(l));

    }

    while (positions.ContainsKey(currentValue)){currentValue -=
0.00001f;}positions.Add(currentValue, new int[] { i, j });

    small += 0.00001f;

}

}

board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
pieceCoords.ElementAt(i), oldPiece, pieces.ElementAt(i));

}

int numMoves = positions.Count-1;

if (numMoves > 0)

{

    if (accuracyRange == 0)

    {

        if (accuracyDefault > 0)

    {

```

```

        int indecies = rnd.Next(Mathf.FloorToInt(positions.Count - (positions.Count * accuracyDefault)),
positions.Count);

        int i = positions.ElementAt(indecies - 1).Value[0];

        int j = positions.ElementAt(indecies - 1).Value[1];

        return (pieceCoords.ElementAt(i), movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

    }

    else

    {

        int indecies = rnd.Next(0, Mathf.FloorToInt(positions.Count * (-1 * accuracyDefault)));

        int i = positions.ElementAt(indecies - 1).Value[0];

        int j = positions.ElementAt(indecies - 1).Value[1];

        return (pieceCoords.ElementAt(i), movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

    }

}

else

{

    if (accuracyDefault > 0)

    {

        int indecies = rnd.Next(Mathf.FloorToInt(positions.Count - (positions.Count * accuracyRange)),
Mathf.FloorToInt(positions.Count - (positions.Count * accuracyDefault)));

        int i = positions.ElementAt(indecies - 1).Value[0];

        int j = positions.ElementAt(indecies - 1).Value[1];

        return (pieceCoords.ElementAt(i), movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

    }

    else

    {

        int indecies = rnd.Next(Mathf.FloorToInt(positions.Count * (-1 * accuracyRange)),
Mathf.FloorToInt(positions.Count * (-1 * accuracyDefault)));

        int i = positions.ElementAt(indecies - 1).Value[0];

        int j = positions.ElementAt(indecies - 1).Value[1];

        return (pieceCoords.ElementAt(i), movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

    }

}

```

```

        }

    }

}

else

{

    return GetMoveRandom(movesCurrentlyAvailable, pieceCoords);

}

}

protected (float,Vector3Int, Vector3Int)
GetMoveMostValueFrom2LookAheadWithNNAndReturnEval(List<List<Vector3Int>>
movesCurrentlyAvailable, List<Piece> pieces, List<Vector3Int> pieceCoords)

{
    List<List<Vector3Int>> opposingMovesCurrentlyAvailable = new List<List<Vector3Int>>();
    List<Piece> opposingPieces = new List<Piece>();
    List<Vector3Int> opposingPieceCoords = new List<Vector3Int>();
    Piece oldPiece;
    Piece oldOPiece;

    SortedDictionary<float, int[]> positions = new SortedDictionary<float, int[]>();
    float currentValue = 0;
    float small = 0.00001f; // prevents same keys
    for (int i = 0; i < movesCurrentlyAvailable.Count; i++)
    {
        for (int j = 0; j < movesCurrentlyAvailable.ElementAt(i).Count; j++)
        {
            oldPiece = board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
            board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
                pieceCoords.ElementAt(i), pieces.ElementAt(i), null);
            (opposingMovesCurrentlyAvailable, opposingPieces, opposingPieceCoords) =
            opposingPlayer.ReturnAllPossibleMoves();
        }
    }
}

```

```

        currentValue = board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j)) ==
null ? 0 :
pieceNameToValueDict[(board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j))).name
];

        currentValue += GetNNEvalFromBoardState(board.grid);

        if (board.majorPiecesMoved < 50)

        {

            currentValue += BasicOpeningTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

        }

        else if (board.majorPiecesTaken < 50)

        {

            currentValue += BasicMiddleGameTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

        }

        else

        {

            currentValue += BasicEndGameTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

        }

        for (int k = 0; k < opposingMovesCurrentlyAvailable.Count; k++)

        {

            for (int l = 0; l < opposingMovesCurrentlyAvailable.ElementAt(k).Count; l++)

            {

                oldOPiece =
board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l));

                board.UpdateBoardOnPieceMove(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l),
opposingPieceCoords.ElementAt(k), opposingPieces.ElementAt(k), null);

                currentValue -=
board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l)) == null ? 0 :
pieceNameToValueDict[(board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l))).name];

                currentValue += GetNNEvalFromBoardState(board.grid);

```

```

board.UpdateBoardOnPieceMove(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l),
opposingPieceCoords.ElementAt(k), oldOPiece, opposingPieces.ElementAt(k));

if (board.majorPiecesMoved < 50)

{

    currentValue -=
BasicOpeningTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(l));

}

else if (board.majorPiecesTaken < 50)

{

    currentValue -=
BasicMiddleGameTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(l));

}

else

{

    currentValue -=
BasicEndGameTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(l));

}

while (positions.ContainsKey(currentValue)){currentValue -=
0.00001f;}positions.Add(currentValue, new int[] { i, j });

small += 0.00001f;

}

}

board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
pieceCoords.ElementAt(i), oldPiece, pieces.ElementAt(i));

}

int numMoves = positions.Count-1;

if (numMoves > 0)

{

int i = positions.Last().Value[0];

```

```

        int j = positions.Last().Value[1];

        return (positions.Last().Key, pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

    }

    else

    {

        (Vector3Int, Vector3Int) tempHolder = GetMoveRandom(movesCurrentlyAvailable, pieceCoords);

        return (-1f, tempHolder.Item1,tempHolder.Item2);

    }

}

protected (float, Vector3Int, Vector3Int)
GetMoveMostValueFrom2LookAheadWithAccuracyAndReturnEval(List<List<Vector3Int>>
movesCurrentlyAvailable, List<Piece> pieces, List<Vector3Int> pieceCoords, float accuracyDefault, float
accuracyRange = 0f)

{
    List<List<Vector3Int>> opposingMovesCurrentlyAvailable = new List<List<Vector3Int>>();

    List<Piece> opposingPieces = new List<Piece>();

    List<Vector3Int> opposingPieceCoords = new List<Vector3Int>();

    Piece oldPiece;

    SortedDictionary<float, int[]> positions = new SortedDictionary<float, int[]>();

    float currentValue = 0;

    float small = 0.00001f; // prevents same keys

    for (int i = 0; i < movesCurrentlyAvailable.Count; i++)

    {

        for (int j = 0; j < movesCurrentlyAvailable.ElementAt(i).Count; j++)

        {

            oldPiece = board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

            board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
pieceCoords.ElementAt(i), pieces.ElementAt(i), null);

        }

    }

}

```

```

        (opposingMovesCurrentlyAvailable, opposingPieces, opposingPieceCoords) =
opposingPlayer.ReturnAllPossibleMoves();

        currentValue = board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j)) ==
null ? 0 :
pieceNameToValueDict[(board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j))).name
];

        currentValue += BasicTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i), board.GetPieceOnSquare(pieceCoords.ElementAt(i)) == null ? true :
board.GetPieceOnSquare(pieceCoords.ElementAt(i)).team == TeamColor.White ? true : false);

        if (board.majorPiecesMoved < 50)

    {

        currentValue += BasicOpeningTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

    }

        else if (board.majorPiecesTaken < 50)

    {

        currentValue += BasicMiddleGameTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

    }

        else

    {

        currentValue += BasicEndGameTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

    }

        for (int k = 0; k < opposingMovesCurrentlyAvailable.Count; k++)

    {

        for (int l = 0; l < opposingMovesCurrentlyAvailable.ElementAt(k).Count; l++)

    {

        currentValue -=
board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l)) == null ? 0 :
pieceNameToValueDict[(board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l))).name];

        currentValue -= BasicTheoryAddition(opposingPieceCoords.ElementAt(k),
opposingMovesCurrentlyAvailable.ElementAt(k), board.GetPieceOnSquare(pieceCoords.ElementAt(i)) == null
? true : board.GetPieceOnSquare(pieceCoords.ElementAt(i)).team == TeamColor.White ? true : false);

        if (board.majorPiecesMoved < 50)

```

```

    {
        currentValue -=
BasicOpeningTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(l));

    }

    else if (board.majorPiecesTaken < 50)

    {

        currentValue -=
BasicMiddleGameTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(l));

    }

    else

    {

        currentValue -=
BasicEndGameTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(l));

    }

    while (positions.ContainsKey(currentValue)) { currentValue -= 0.00001f; }

    positions.Add(currentValue, new int[] { i, j });

    small += 0.00001f;

}

}

board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
pieceCoords.ElementAt(i), oldPiece, pieces.ElementAt(i));

}

int numMoves = positions.Count-1;

if (numMoves > 0)

{
    if (accuracyRange == 0)

    {
        if (accuracyDefault > 0)
    }
}

```

```

        int indecies = rnd.Next(Mathf.FloorToInt(positions.Count - (positions.Count * accuracyDefault)),
positions.Count);

        int i = positions.ElementAt(indecies - 1).Value[0];

        int j = positions.ElementAt(indecies - 1).Value[1];

        return (positions.ElementAt(indecies).Key, pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

    }

    else

    {

        int indecies = rnd.Next(0, Mathf.FloorToInt(positions.Count * (-1 * accuracyDefault)));

        int i = positions.ElementAt(indecies - 1).Value[0];

        int j = positions.ElementAt(indecies - 1).Value[1];

        return (positions.ElementAt(indecies).Key, pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

    }

}

else

{

    if (accuracyDefault > 0)

    {

        int indecies = rnd.Next(Mathf.FloorToInt(positions.Count - (positions.Count * accuracyRange)),
Mathf.FloorToInt(positions.Count - (positions.Count * accuracyDefault)));

        int i = positions.ElementAt(indecies - 1).Value[0];

        int j = positions.ElementAt(indecies - 1).Value[1];

        return (positions.ElementAt(indecies).Key, pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

    }

    else

    {

        int indecies = rnd.Next(Mathf.FloorToInt(positions.Count * (-1 * accuracyRange)),
Mathf.FloorToInt(positions.Count * (-1 * accuracyDefault)));

        int i = positions.ElementAt(indecies - 1).Value[0];

```

```

        int j = positions.ElementAt(indexes - 1).Value[1];

        return (positions.ElementAt(indexes).Key, pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

    }

}

else

{

    (Vector3Int, Vector3Int) tempHolder = GetMoveRandom(movesCurrentlyAvailable, pieceCoords);

    return (-1f, tempHolder.Item1, tempHolder.Item2);

}

}

protected (Vector3Int, Vector3Int)
GetMoveMostValueFrom2LookAheadWithAccuracyAndDeeperLineEvaluationWithAdditionalTheory(List<List
<Vector3Int>> movesCurrentlyAvailable, List<Piece> pieces, List<Vector3Int> pieceCoords, float
accuracyDefault, float accuracyRange = 0f)

{
    List<List<Vector3Int>> opposingMovesCurrentlyAvailable = new List<List<Vector3Int>>();

    List<Piece> opposingPieces = new List<Piece>();

    List<Vector3Int> opposingPieceCoords = new List<Vector3Int>();

    Piece oldPiece;

    SortedDictionary<float, int[]> positions = new SortedDictionary<float, int[]>();

    List<(Vector3Int, Vector3Int)> furtherPositions = new List<(Vector3Int, Vector3Int)>();

    float currentValue = 0;

    float small = 0.00001f; // prevents same keys

    for (int i = 0; i < movesCurrentlyAvailable.Count; i++)

    {

        for (int j = 0; j < movesCurrentlyAvailable.ElementAt(i).Count; j++)

        {

```

```

oldPiece = board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
pieceCoords.ElementAt(i), pieces.ElementAt(i), null);

(opposingMovesCurrentlyAvailable, opposingPieces, opposingPieceCoords) =
opposingPlayer.ReturnAllPossibleMoves();

currentValue = board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j)) ==
null ? 0 :
(pieceNameToValueDict[(board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j))).name]
+ board.materialImbalance * 0.05f);

currentValue += IntermediateTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

if(board.majorPiecesMoved < 50)

{

    currentValue += BasicOpeningTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

}

else if(board.majorPiecesTaken < 50)

{

    currentValue += BasicMiddleGameTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

}

else

{

    currentValue += BasicEndGameTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

}

for (int k = 0; k < opposingMovesCurrentlyAvailable.Count; k++)

{

    for (int l = 0; l < opposingMovesCurrentlyAvailable.ElementAt(k).Count; l++)

    {

        currentValue -=

board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l)) == null ? 0 :
(pieceNameToValueDict[(board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l))).name]
+ board.materialImbalance * 0.05f);

```

```

        currentValue -= IntermediateTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(l));

        if (board.majorPiecesMoved < 50)

        {

            currentValue -=
BasicOpeningTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(l));

        }

        else if (board.majorPiecesTaken < 50)

        {

            currentValue -=
BasicMiddleGameTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(l));

        }

        else

        {

            currentValue -=
BasicEndGameTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(l));

        }

        while (positions.ContainsKey(currentValue)){currentValue -=
0.00001f;}positions.Add(currentValue, new int[] { i, j });

        small += 0.00001f;

    }

}

board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
pieceCoords.ElementAt(i), oldPiece, pieces.ElementAt(i));

}

int numMoves = positions.Count-1;

if (numMoves > 0)

{

    if (accuracyRange == 0)

    {

```

```

if (accuracyDefault > 0)

{
    for (int a = 0; a < 3; a++)

    {

        int indecies = rnd.Next(Mathf.FloorToInt(positions.Count - (positions.Count *
accuracyDefault)), positions.Count);

        int i = positions.ElementAt(indecies - 1).Value[0];

        int j = positions.ElementAt(indecies - 1).Value[1];

        furtherPositions.Add((pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j)));

    }

}

else

{

    for (int a = 0; a < 3; a++)

    {

        int indecies = rnd.Next(0, Mathf.FloorToInt(positions.Count * (-1 * accuracyDefault)));

        int i = positions.ElementAt(indecies - 1).Value[0];

        int j = positions.ElementAt(indecies - 1).Value[1];

        furtherPositions.Add((pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j)));

    }

}

else

{

    if (accuracyDefault > 0)

    {

        for (int a = 0; a < 3; a++)

```

```

    {

        int indecies = rnd.Next(Mathf.FloorToInt(positions.Count - (positions.Count *
accuracyRange)), Mathf.FloorToInt(positions.Count - (positions.Count * accuracyDefault)));

        int i = positions.ElementAt(indecies - 1).Value[0];

        int j = positions.ElementAt(indecies - 1).Value[1];

        furtherPositions.Add((pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j)));

    }

}

else

{

    for (int a = 0; a < 3; a++)

    {

        int indecies = rnd.Next(Mathf.FloorToInt(positions.Count * (-1 * accuracyRange)),
Mathf.FloorToInt(positions.Count * (-1 * accuracyDefault)));

        int i = positions.ElementAt(indecies - 1).Value[0];

        int j = positions.ElementAt(indecies - 1).Value[1];

        furtherPositions.Add((pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j)));

    }

}

List<(float, Vector3Int, Vector3Int)> furtherPositionsWithEval = new List<(float, Vector3Int,
Vector3Int)>();

for (int a = 0; a < 3; a++)

{

    List<List<Vector3Int>> furtherTempMovesCurrentlyAvailable = new List<List<Vector3Int>>();

    List<Vector3Int> miniFurtherTempMovesCurrentlyAvailable = new List<Vector3Int>();

    List<Piece> furtherTempPieces = new List<Piece>();

    List<Vector3Int> furtherTempPieceCoords = new List<Vector3Int>();
}

```

```

miniFurtherTempMovesCurrentlyAvailable.Add(furtherPositions.ElementAt(a).Item2);

furtherTempMovesCurrentlyAvailable.Add(miniFurtherTempMovesCurrentlyAvailable);

furtherTempPieces.Add(board.GetPieceOnSquare(furtherPositions.ElementAt(a).Item1));

furtherTempPieceCoords.Add(furtherPositions.ElementAt(a).Item1);

(Ventor3Int, Ventor3Int) halfWayMoveChosen =
GetMoveMostValueFrom2LookAheadWithAccuracy(furtherTempMovesCurrentlyAvailable,
furtherTempPieces, furtherTempPieceCoords, 0.15f);

miniFurtherTempMovesCurrentlyAvailable.Clear();

furtherTempMovesCurrentlyAvailable.Clear();

furtherTempPieces.Clear();

furtherTempPieceCoords.Clear();

miniFurtherTempMovesCurrentlyAvailable.Add(halfWayMoveChosen.Item2);

furtherTempMovesCurrentlyAvailable.Add(miniFurtherTempMovesCurrentlyAvailable);

furtherTempPieces.Add(board.GetPieceOnSquare(halfWayMoveChosen.Item1));

furtherTempPieceCoords.Add(halfWayMoveChosen.Item1);

furtherPositionsWithEval.Add(GetMoveMostValueFrom2LookAheadWithAccuracyAndReturnEval(furtherTempMovesCurrentlyAvailable, furtherTempPieces, furtherTempPieceCoords, 0.15f));

}

float move1 = furtherPositionsWithEval.ElementAt(0).Item1;

float move2 = furtherPositionsWithEval.ElementAt(1).Item1;

float move3 = furtherPositionsWithEval.ElementAt(2).Item1;

if (move1 > move2)

{

    if (move1 > move3)

    {

        return (furtherPositionsWithEval.ElementAt(0).Item2,
furtherPositionsWithEval.ElementAt(0).Item3);
    }
}

```

```

        }

        else

        {

            return (furtherPositionsWithEval.ElementAt(2).Item2,
furtherPositionsWithEval.ElementAt(2).Item3);

        }

    }

    else

    {

        if (move2 > move3)

        {

            return (furtherPositionsWithEval.ElementAt(1).Item2,
furtherPositionsWithEval.ElementAt(1).Item3);

        }

        else

        {

            return (furtherPositionsWithEval.ElementAt(2).Item2,
furtherPositionsWithEval.ElementAt(2).Item3);

        }

    }

}

else

{

    return GetMoveRandom(movesCurrentlyAvailable, pieceCoords);

}

}

protected (Vector3Int, Vector3Int)
GetMoveMostValueFrom2LookAheadAndDeeperLineEvaluationWithPositionalNeuralNetwork(List<List<Vect
or3Int>> movesCurrentlyAvailable, List<Piece> pieces, List<Vector3Int> pieceCoords)

{

```

```

List<List<Vector3Int>> opposingMovesCurrentlyAvailable = new List<List<Vector3Int>>();
List<Piece> opposingPieces = new List<Piece>();
List<Vector3Int> opposingPieceCoords = new List<Vector3Int>();
Piece oldPiece;
Piece oldOPiece;

SortedDictionary<float, int[]> positions = new SortedDictionary<float, int[]>();
List<(Vector3Int, Vector3Int)> furtherPositions = new List<(Vector3Int, Vector3Int)>();
float currentValue = 0;

float small = 0.00001f; // prevents same keys

for (int i = 0; i < movesCurrentlyAvailable.Count; i++)
{
    for (int j = 0; j < movesCurrentlyAvailable.ElementAt(i).Count; j++)
    {
        oldPiece = board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
        board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
pieceCoords.ElementAt(i), pieces.ElementAt(i), null);

        (opposingMovesCurrentlyAvailable, opposingPieces, opposingPieceCoords) =
opposingPlayer.ReturnAllPossibleMoves();

        currentValue = board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j)) ==
null ? 0 :
(pieceNameToValueDict[(board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j))).name] + board.materialImbalance * 0.05f);

        currentValue += GetNNEvalFromBoardState(board.grid);

        if (board.majorPiecesMoved < 50)
        {

            currentValue += BasicOpeningTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
        }
        else if (board.majorPiecesTaken < 50)
        {
    
```

```

        currentValue += BasicMiddleGameTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

    }

    else

    {

        currentValue += BasicEndGameTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

    }

    for (int k = 0; k < opposingMovesCurrentlyAvailable.Count; k++)

    {

        for (int l = 0; l < opposingMovesCurrentlyAvailable.ElementAt(k).Count; l++)

        {

            oldOPiece =
board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l));

            board.UpdateBoardOnPieceMove(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l),
opposingPieceCoords.ElementAt(k), opposingPieces.ElementAt(k), null);

            currentValue -=
board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l)) == null ? 0 :
pieceNameToValueDict[(board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l))).name];

            currentValue += GetNNEvalFromBoardState(board.grid);

            board.UpdateBoardOnPieceMove(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l),
opposingPieceCoords.ElementAt(k), oldOPiece, opposingPieces.ElementAt(k));

            if (board.majorPiecesMoved < 50)

            {

                currentValue -=
BasicOpeningTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(l));

            }

            else if (board.majorPiecesTaken < 50)

            {

                currentValue -=
BasicMiddleGameTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(l));

```

```

        }

        else

        {

            currentValue -=
BasicEndGameTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(l));

        }

        while (positions.ContainsKey(currentValue)) { currentValue -= 0.00001f; }

        positions.Add(currentValue, new int[] { i, j });

        small += 0.00001f;

    }

}

board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
pieceCoords.ElementAt(i), oldPiece, pieces.ElementAt(i));

}

}

int numMoves = positions.Count - 1;

if (numMoves > 0)

{

    for (int a = 0; a < 5; a++)

    {

        int i = positions.Last().Value[0];

        int j = positions.Last().Value[1];

        furtherPositions.Add((pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j)));

    }

    List<(float, Vector3Int, Vector3Int)> furtherPositionsWithEval = new List<(float, Vector3Int,
Vector3Int)>();

    for (int a = 0; a < 5; a++)

    {

        List<List<Vector3Int>> furtherTempMovesCurrentlyAvailable = new List<List<Vector3Int>>();

        List<Vector3Int> miniFurtherTempMovesCurrentlyAvailable = new List<Vector3Int>();

```

```

List<Piece> furtherTempPieces = new List<Piece>();
List<Vector3Int> furtherTempPieceCoords = new List<Vector3Int>();

miniFurtherTempMovesCurrentlyAvailable.Add(furtherPositions.ElementAt(a).Item2);
furtherTempMovesCurrentlyAvailable.Add(miniFurtherTempMovesCurrentlyAvailable);
furtherTempPieces.Add(board.GetPieceOnSquare(furtherPositions.ElementAt(a).Item1));
furtherTempPieceCoords.Add(furtherPositions.ElementAt(a).Item1);

(Vector3Int, Vector3Int) halfWayMoveChosen =
GetMoveMostValueFrom2LookAheadWithNN(furtherTempMovesCurrentlyAvailable, furtherTempPieces,
furtherTempPieceCoords);

miniFurtherTempMovesCurrentlyAvailable.Clear();
furtherTempMovesCurrentlyAvailable.Clear();
furtherTempPieces.Clear();
furtherTempPieceCoords.Clear();
miniFurtherTempMovesCurrentlyAvailable.Add(halfWayMoveChosen.Item2);
furtherTempMovesCurrentlyAvailable.Add(miniFurtherTempMovesCurrentlyAvailable);
furtherTempPieces.Add(board.GetPieceOnSquare(halfWayMoveChosen.Item1));
furtherTempPieceCoords.Add(halfWayMoveChosen.Item1);

furtherPositionsWithEval.Add(GetMoveMostValueFrom2LookAheadWithNNAndReturnEval(furtherTempMovesCurrentlyAvailable, furtherTempPieces, furtherTempPieceCoords));

}

int maxIndex = 0;
float max = 0;
for (int i = 0; i < 5; i++)
{
    if(furtherPositionsWithEval.ElementAt(i).Item1 > max)
    {
        maxIndex = i;
    }
}

```

```

        }

    }

    return (furtherPositionsWithEval.ElementAt(maxIndex).Item2,
furtherPositionsWithEval.ElementAt(maxIndex).Item3);

}

else

{

    return GetMoveRandom(movesCurrentlyAvailable, pieceCoords);

}

}


```

```

private float GetNNEvalFromBoardState(Piece[,] grid)

{
    float[] inputs = new float[512*7*2];

    int counter = 0;

    for (int i = 0; i < 8; i++)
    {
        for (int j = 0; j < 8; j++)
        {
            for (int k = 0; k < 8; k++)
            {
                if (grid[i,j,k] != null)
                {
                    if(grid[i,j,k].team == TeamColor.White)
                    {
                        if (grid[i, j, k].name == "Pawn" || grid[i, j, k].name == "Pawn(Clone)")
                        {
                            inputs[counter] = 1;
                        }
                    }
                }
            }
        }
    }
}


```

```

else if (grid[i, j, k].name == "Knight" || grid[i, j, k].name == "Knight(Clone)")

{
    inputs[counter+1] = 1;

}

else if (grid[i, j, k].name == "Bishop" || grid[i, j, k].name == "Bishop(Clone)")

{
    inputs[counter+2] = 1;

}

else if (grid[i,j,k].name == "Rook" || grid[i, j, k].name == "Rook(Clone)")

{
    inputs[counter+3] = 1;

}

else if (grid[i,j,k].name == "Commoner" || grid[i, j, k].name == "Commoner(Clone)")

{
    inputs[counter+4] = 1;

}

else if (grid[i, j, k].name == "King" || grid[i, j, k].name == "King(Clone)")

{
    inputs[counter+5] = 1;

}

else if (grid[i, j, k].name == "Queen" || grid[i, j, k].name == "Queen(Clone)")

{
    inputs[counter+6] = 1;

}

counter += 14;

}

else

{
    counter += 7;

    if (grid[i, j, k].name == "Pawn" || grid[i, j, k].name == "Pawn(Clone)")

```

```

{
    inputs[counter] = l;
}

else if (grid[i, j, k].name == "Knight" || grid[i, j, k].name == "Knight(Clone)")

{
    inputs[counter + 1] = l;
}

else if (grid[i, j, k].name == "Bishop" || grid[i, j, k].name == "Bishop(Clone)")

{
    inputs[counter + 2] = l;
}

else if (grid[i, j, k].name == "Rook" || grid[i, j, k].name == "Rook(Clone)")

{
    inputs[counter + 3] = l;
}

else if (grid[i, j, k].name == "Commoner" || grid[i, j, k].name == "Commoner(Clone)")

{
    inputs[counter + 4] = l;
}

else if (grid[i, j, k].name == "King" || grid[i, j, k].name == "King(Clone)")

{
    inputs[counter + 5] = l;
}

else if (grid[i, j, k].name == "Queen" || grid[i, j, k].name == "Queen(Clone)")

{
    inputs[counter + 6] = l;
}

counter += 7;
}

```

```

        }

        else

        {

            counter += 14;

        }

    }

}

return networkManager.FeedForward(inputs)[0];//it is okay to do this as we know there is only one
output state (the eval)

}

protected (Vector3Int, Vector3Int)
GetMoveMostValueFrom2LookAheadWithAccuracyAndLargeMoveNegativityMitigation(List<List<Vector3Int>
> movesCurrentlyAvailable, List<Piece> pieces, List<Vector3Int> pieceCoords, float accuracyDefault, float
accuracyRange = 0f)

{

List<List<Vector3Int>> opposingMovesCurrentlyAvailable = new List<List<Vector3Int>>();

List<Piece> opposingPieces = new List<Piece>();

List<Vector3Int> opposingPieceCoords = new List<Vector3Int>();

Piece oldPiece;

SortedDictionary<float, int[]> positions = new SortedDictionary<float, int[]>();

float currentValue = 0;

float small = 0.00001f; // prevents same keys

for (int i = 0; i < movesCurrentlyAvailable.Count; i++)

{

    for (int j = 0; j < movesCurrentlyAvailable.ElementAt(i).Count; j++)

    {

        oldPiece = board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

        board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
pieceCoords.ElementAt(i), pieces.ElementAt(i), null);
    }
}

```

```

        (opposingMovesCurrentlyAvailable, opposingPieces, opposingPieceCoords) =
opposingPlayer.ReturnAllPossibleMoves();

        currentValue = board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j)) ==
null ? 0 :
pieceNameToValueDict[(board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j))).name];
];

        currentValue += BasicTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i), board.GetPieceOnSquare(pieceCoords.ElementAt(i)) == null ? true :
board.GetPieceOnSquare(pieceCoords.ElementAt(i)).team == TeamColor.White ? true : false);

        for (int k = 0; k < opposingMovesCurrentlyAvailable.Count; k++)
{
    for (int l = 0; l < opposingMovesCurrentlyAvailable.ElementAt(k).Count; l++)
{
    currentValue -=
board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l)) == null ? 0 :
pieceNameToValueDict[(board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l))).name];

    currentValue -= BasicTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves, board.GetPieceOnSquare(opposingPieceCoords.ElementAt(i)) ==
null ? true : board.GetPieceOnSquare(opposingPieceCoords.ElementAt(i)).team == TeamColor.White ?
false : true);

    while (positions.ContainsKey(currentValue)){currentValue -=
0.00001f;}positions.Add(currentValue, new int[] { i, j });

    small += 0.00001f;
}
}
}

        board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
pieceCoords.ElementAt(i), oldPiece, pieces.ElementAt(i));
}

}

int numMoves = positions.Count-l;

if (numMoves > 0)
{
    if (accuracyRange == 0)
{
    if (accuracyDefault > 0)

```

```

{
    if(positions.ElementAt(Mathf.FloorToInt(positions.Count - Mathf.FloorToInt(positions.Count -
(positions.Count * accuracyDefault))/2).Key < 0)

    {

        // no need to create an additional list to only add values in the accuracy range as the above
        takes care of this at the same time as the 50% negative implementation discussed in 4.a.ii.7 in objectives

        // this is because we know that the first negative value must lie within this range

        SortedDictionary<float, int[]> positivePositions = new SortedDictionary<float, int[]>();

        foreach(KeyValuePair<float, int[]> item in positions)

        {

            if(item.Key >= 0)

            {

                positivePositions.Add(item.Key,item.Value);

            }

        }

    }

    int indecies = rnd.Next(0,positivePositions.Count);

    int i = positions.ElementAt(indecies - 1).Value[0];

    int j = positions.ElementAt(indecies - 1).Value[1];

    return (pieceCoords.ElementAt(i), movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

}

else

{

    int indecies = rnd.Next(Mathf.FloorToInt(positions.Count - (positions.Count *
accuracyDefault)), positions.Count);

    int i = positions.ElementAt(indecies - 1).Value[0];

    int j = positions.ElementAt(indecies - 1).Value[1];

    return (pieceCoords.ElementAt(i), movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

}

}

```

```

        else
    {

        int indecies = rnd.Next(0, Mathf.FloorToInt(positions.Count * (-1 * accuracyDefault)));
        int i = positions.ElementAt(indecies - 1).Value[0];
        int j = positions.ElementAt(indecies - 1).Value[1];
        return (pieceCoords.ElementAt(i), movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
    }
}

else
{
    if (accuracyDefault > 0)

    {
        int indecies = rnd.Next(Mathf.FloorToInt(positions.Count - (positions.Count * accuracyRange)),
        Mathf.FloorToInt(positions.Count - (positions.Count * accuracyDefault)));
        int i = positions.ElementAt(indecies - 1).Value[0];
        int j = positions.ElementAt(indecies - 1).Value[1];
        return (pieceCoords.ElementAt(i), movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
    }
    else
    {
        int indecies = rnd.Next(Mathf.FloorToInt(positions.Count * (-1 * accuracyRange)),
        Mathf.FloorToInt(positions.Count * (-1 * accuracyDefault)));
        int i = positions.ElementAt(indecies - 1).Value[0];
        int j = positions.ElementAt(indecies - 1).Value[1];
        return (pieceCoords.ElementAt(i), movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
    }
}
else
{
    return GetMoveRandom(movesCurrentlyAvailable, pieceCoords);
}

```

```

    }

}

protected (Vector3Int, Vector3Int)
GetMoveByAlphaBetaPruningWithMovePriorityOrdering(List<List<Vector3Int>> movesCurrentlyAvailable,
List<Piece> pieces, List<Vector3Int> pieceCoords, int runNum=0)

{
    List<List<Vector3Int>> opposingMovesCurrentlyAvailable = new List<List<Vector3Int>>();
    List<Piece> opposingPieces = new List<Piece>();
    List<Vector3Int> opposingPieceCoords = new List<Vector3Int>();
    Piece oldPiece;

    List<int[]> maxValuePositions = new List<int[]>();
    int alpha = 0;
    int beta = 9;
    int currentValue = 0;
    for (int i = 0; i < movesCurrentlyAvailable.Count; i++)
    {
        for (int j = 0; j < movesCurrentlyAvailable.ElementAt(i).Count; j++)
        {
            oldPiece = board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
            board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
                pieceCoords.ElementAt(i), pieces.ElementAt(i), null);

            (opposingMovesCurrentlyAvailable, opposingPieces, opposingPieceCoords) =
                opposingPlayer.ReturnAllPossibleMoves();

            (opposingMovesCurrentlyAvailable, opposingPieces, opposingPieceCoords) =
                GiveMovesPriorityOrder(opposingMovesCurrentlyAvailable, opposingPieces, opposingPieceCoords,
                    movesCurrentlyAvailable);

            currentValue = board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j)) ==
                null ? 0 :
                pieceNameToValueDict[(board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j))).name];
        }
    }
}

```

```

if (pieceName == "Pawn" || pieceName == "Pawn(Clone)")
{
    currentValue += 9;
}

for (int k = 0; k < opposingMovesCurrentlyAvailable.Count; k++)
{
    for (int l = 0; l < opposingMovesCurrentlyAvailable.ElementAt(k).Count; l++)
    {
        currentValue -=
board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l)) == null ? 0 :
pieceNameToValueDict[(board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt
(l))).name];

        if (pieceName == "Pawn" || pieceName == "Pawn(Clone)")

        {
            currentValue -= 9;
        }

        if (currentValue < beta)
        {
            break;
        }

        if (currentValue > alpha)
        {
            maxValuePositions.Clear();

            maxValuePositions.Add(new int[] { i, j });

        }

        else if (currentValue == alpha)
        {
            maxValuePositions.Add(new int[] { i, j });

        }
    }
}

if (currentValue < beta)

```

```

    {
        break;
    }
}

board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
pieceCoords.ElementAt(i), oldPiece, pieces.ElementAt(i));

}
}

int numBestMoves = maxValuePositions.Count;

if (numBestMoves > 0)

{
    int val = Guid.NewGuid().GetHashCode() % numBestMoves;

    val = val > 0 ? val : -val;

    return (pieceCoords.ElementAt(maxValuePositions.ElementAt(val)[0]),
movesCurrentlyAvailable.ElementAt(maxValuePositions.ElementAt(val)[0]).ElementAt(maxValuePositions.ElementAt(val)[1]));
}

else

{
    return GetMoveRandom(movesCurrentlyAvailable, pieceCoords);
}

}

protected (List<List<Vector3Int>>, List<Piece>, List<Vector3Int>)
GiveMovesPriorityOrder(List<List<Vector3Int>> moves, List<Piece> pieces, List<Vector3Int> positions,
List<List<Vector3Int>> opposingAttacks)

{
    List<List<Vector3Int>> orderedMoves = new List<List<Vector3Int>>();
    List<Vector3Int> orderedPositions = new List<Vector3Int>();
    List<Piece> orderedPieces = new List<Piece>();

    Piece capturedPiece;
}

```

```

string capturedPieceName;
string pieceName;
int estimatedMoveReward = 0;
Vector3Int moveToSquare;
Dictionary<(Vector3Int, Piece, Vector3Int), int> findOrderDict = new Dictionary<(Vector3Int, Piece, Vector3Int), int>();
int totalMoves = 0;

for (int i = 0; i < moves.Count; i++)
{
    for (int j = 0; j < moves.ElementAt(i).Count; j++)
    {
        moveToSquare = moves.ElementAt(i).ElementAt(j);
        pieceName = pieces.ElementAt(i).name;
        capturedPiece = board.GetPieceOnSquare(moveToSquare);
        capturedPieceName = (capturedPiece == null ? "" : capturedPiece.name);
        estimatedMoveReward = 0;
        if (capturedPieceName != "")
        {
            estimatedMoveReward += 10 * pieceNameToValueDict[capturedPieceName] -
pieceNameToValueDict[pieceName];
        }

        if ((pieceName == "Pawn" || pieceName == "Pawn(Clone") && moveToSquare.y == 7)
        {
            estimatedMoveReward += 9;
        }

        for (int k = 0; k < opposingAttacks.Count; k++)
        {
            for (int l = 0; l < opposingAttacks.ElementAt(k).Count; l++)
            {

```

```

    {
        if (moveToSquare == opposingAttacks.ElementAt(k).ElementAt(l))
        {
            estimatedMoveReward -= pieceNameToValueDict[pieceName];
        }
    }

    totalMoves++;

    findOrderDict.Add((moveToSquare, pieces.ElementAt(i), positions.ElementAt(i)),
estimatedMoveReward);
}

}

for (int i = 18; i > -10; i--)
{
    var matchingKeys = findOrderDict.Where(kvp => kvp.Value == i).Select(kvp => kvp.Key);

    foreach((Vector3Int, Piece, Vector3Int) key in matchingKeys)
    {
        List<Vector3Int> move = new List<Vector3Int>() { key.Item1 };

        orderedMoves.Add(move);

        orderedPieces.Add(key.Item2);

        orderedPositions.Add(key.Item3);
    }
}

return (orderedMoves, orderedPieces, orderedPositions);
}

protected float[] SetupState(List<List<Vector3Int>> movesCurrentlyAvailable, List<Piece> pieceNames,
List<Vector3Int> pieceCoords)
{

```

```

        throw new NotImplementedException();

    }

protected (Vector3Int coordsOfPieceToMove, Vector3Int coordsToMoveTo) DecodeInputState(object
indexOfMove)
{
    throw new NotImplementedException();
}

public int GetNumMovesMade()
{
    return movesMade;
}

public void IncreaseNumMovesMade()
{
    movesMade++;
}

}

```

3.1.4 – TECHNICAL SOLUTION – ALGORITHMS – LEGAL MOVES

The legal moves implementation is split throughout the code as it makes it more efficient this way and is another example of a complex user defined system with optimisation techniques,

implementing when needed. These relevant parts are only called when needed and many parts are taken care of at the same time. The main example of this can be seen in chessgame controller where the available moves are created (for the likes of displaying moves on a piece click, checking for checkmate and for the AI to have a moves list), the illegal moves for king are removed, and checkmate is checked using the same functions. This can be done as first two can be used in a way which allows you to check checkmate, optimising the code. Another example is that whilst all illegal moves must be removed for the AI before the AI can calculate for a player this is not the case and instead illegal moves can be removed on the click of the piece, again reducing calculation. This can be seen in chessgame controller where remove checking moves is only called if AI is active and in board where on SelectPiece the illegal moves are removed.

GENERATE PLAYER MOVES

It is easiest to generate all moves for all pieces at once – following a few simple rules (out of bound, through a piece and how the piece moves in terms of vectors) – at the start of turns as this will be needed to display

available moves on the board and for the AI, but most importantly to check for checkmate and stalemate. For each piece here is the class – it gives the directional vectors and the create move functions.

PAWN

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Pawn : Piece
{
    public override List<Vector3Int> SelectAvailableSquares()
    {
        availableMoves.Clear();

        Vector3Int direction = team == TeamColor.White ? new Vector3Int(0,1,0) : new
Vector3Int (0,-1,0);
        float range = hasMoved ? 1 : 2;
        for (int i = 1; i <= range; i++)
        {
            Vector3Int nextCoords = occupiedSquare + direction * i;
            Piece piece = board.GetPieceOnSquare(nextCoords);
            if (!board.CheckIfCoordinatesAreOnBoard(nextCoords))
                break;
            if (piece == null)
                TryToAddMove(nextCoords);
            else
                break;
        }

        Vector3Int[] takeDirectionsWhite = new Vector3Int[]
        {
            new Vector3Int (1,1,1),
            new Vector3Int (-1,1,1),
            new Vector3Int (1,1,-1),
            new Vector3Int (-1,1,-1)
        };

        Vector3Int[] takeDirectionsBlack = new Vector3Int[]
        {
            new Vector3Int (1,-1,1),
            new Vector3Int (-1,-1,1),
            new Vector3Int (1,-1,-1),
            new Vector3Int (-1,-1,-1)
        };

        for (int i = 0; i < 4; i++)
        {
            Vector3Int nextCoords = team == TeamColor.White? (occupiedSquare +
takeDirectionsWhite[i]) : (occupiedSquare + takeDirectionsBlack[i]);
            Piece piece = board.GetPieceOnSquare(nextCoords);
            if (!board.CheckIfCoordinatesAreOnBoard(nextCoords))
                continue;
            if (piece != null && !piece.IsFromSameTeam(this))
            {
                TryToAddMove(nextCoords);
            }
        }
    return availableMoves;
}
```

```

        }

    public override void MovePiece(Vector3Int coords,Piece piece = null, bool isPawn =
false)
    {
        base.MovePiece(coords,null,true) ;
        CheckPromotion();
    }

    private void CheckPromotion()
    {
        int endOfBoardYCoord = team == TeamColor.White ? Board.BOARD_SIZE - 1 : 0;
        if (occupiedSquare.y == endOfBoardYCoord)
        {
            board.PromotePiece(this);
        }
    }
}

```

KNIGHT

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Knight : Piece
{
    Vector3Int[] offsets = new Vector3Int[]
    {
        new Vector3Int(1,1,2),
        new Vector3Int(1,1,-2),
        new Vector3Int(1,-1,2),
        new Vector3Int(-1,1,2),
        new Vector3Int(-1,-1,2),
        new Vector3Int(-1,1,-2),
        new Vector3Int(1,-1,-2),
        new Vector3Int(-1,-1,-2),

        new Vector3Int(1,2,1),
        new Vector3Int(1,2,-1),
        new Vector3Int(1,-2,1),
        new Vector3Int(-1,2,1),
        new Vector3Int(-1,-2,1),
        new Vector3Int(-1,2,-1),
        new Vector3Int(1,-2,-1),
        new Vector3Int(-1,-2,-1),

        new Vector3Int(2,1,1),
        new Vector3Int(2,1,-1),
        new Vector3Int(2,-1,1),
        new Vector3Int(-2,1,1),
        new Vector3Int(-2,-1,1),
        new Vector3Int(-2,1,-1),
        new Vector3Int(2,-1,-1),
        new Vector3Int(-2,-1,-1),
    };
}

public override List<Vector3Int> SelectAvailableSquares()
{
    availableMoves.Clear();
}

```

```

        for (int i = 0; i < offsets.Length; i++)
    {
        Vector3Int nextCoords = occupiedSquare + offsets[i];
        Piece piece = board.GetPieceOnSquare(nextCoords);
        if (!board.CheckIfCoordinatesAreOnBoard(nextCoords))
            continue;
        if (piece == null || !piece.IsFromSameTeam(this))
            TryToAddMove(nextCoords);
    }
    return availableMoves;
}
}

```

BISHOP

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Bishop : Piece
{
    private Vector3Int[] directions = new Vector3Int[]
    {
        new Vector3Int(1, 1, 0),
        new Vector3Int(1, -1, 0),
        new Vector3Int(-1, 1, 0),
        new Vector3Int(-1, -1, 0),
        new Vector3Int(-1, -1, -1),
        new Vector3Int(-1, -1, 1),
        new Vector3Int(-1, 1, -1),
        new Vector3Int(-1, 1, 1),
        new Vector3Int(1, -1, -1),
        new Vector3Int(1, -1, 1),
        new Vector3Int(1, 1, -1),
        new Vector3Int(1, 1, 1),
    };
    public override List<Vector3Int> SelectAvailableSquares()
    {
        availableMoves.Clear();
        float range = Board.BOARD_SIZE;
        foreach (var direction in directions)
        {
            for (int i = 1; i <= range; i++)
            {
                Vector3Int nextCoords = occupiedSquare + direction * i;
                Piece piece = board.GetPieceOnSquare(nextCoords);
                if (!board.CheckIfCoordinatesAreOnBoard(nextCoords))
                    break;
                if (piece == null)
                    TryToAddMove(nextCoords);
                else if (!piece.IsFromSameTeam(this))
                {
                    TryToAddMove(nextCoords);
                    break;
                }
                else if (piece.IsFromSameTeam(this))
                    break;
            }
        }
    }
}

```

```

        }
    }
    return availableMoves;
}
}

```

ROOK

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Rook : Piece
{
    private Vector3Int[] directions = new Vector3Int[]
    {
        new Vector3Int(0,0,1),
        new Vector3Int(0,0,-1),
        new Vector3Int(0,1,0),
        new Vector3Int(0,-1,0),
        new Vector3Int(1,0,0),
        new Vector3Int(-1,0,0),
    };
    public override List<Vector3Int> SelectAvailableSquares()
    {
        availableMoves.Clear();

        float range = Board.BOARD_SIZE;
        foreach (var direction in directions)
        {
            for (int i = 1; i <= range; i++)
            {
                Vector3Int nextCoords = occupiedSquare + direction * i;
                Piece piece = board.GetPieceOnSquare(nextCoords);
                if (!board.CheckIfCoordinatesAreOnBoard(nextCoords))
                    break;
                if (piece == null)
                    TryToAddMove(nextCoords);
                else if (!piece.IsFromSameTeam(this))
                {
                    TryToAddMove(nextCoords);
                    break;
                }
                else if (piece.IsFromSameTeam(this))
                    break;
            }
        }
        return availableMoves;
    }
}

```

COMMONER

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

```

```

public class Commoner : Piece
{
    private Vector3Int[] directions = new Vector3Int[]
    {
        new Vector3Int(0,0,-1),
        new Vector3Int(-1,-1,-1),
        new Vector3Int(-1,0,-1),
        new Vector3Int(-1,1,-1),
        new Vector3Int(0,-1,-1),
        new Vector3Int(0,1,-1),
        new Vector3Int(1,-1,-1),
        new Vector3Int(1,0,-1),
        new Vector3Int(1,1,-1),

        new Vector3Int(-1,-1,1),
        new Vector3Int(-1,0,1),
        new Vector3Int(-1,1,1),
        new Vector3Int(0,-1,1),
        new Vector3Int(0,1,1),
        new Vector3Int(1,-1,1),
        new Vector3Int(1,0,1),
        new Vector3Int(1,1,1),
        new Vector3Int(0,0,1),

        new Vector3Int(-1,-1,0),
        new Vector3Int(-1,0,0),
        new Vector3Int(-1,1,0),
        new Vector3Int(0,-1,0),
        new Vector3Int(0,1,0),
        new Vector3Int(1,-1,0),
        new Vector3Int(1,0,0),
        new Vector3Int(1,1,0),
    };
    public override List<Vector3Int> SelectAvailableSquares()
    {
        availableMoves.Clear();

        float range = 1;
        foreach (var direction in directions)
        {
            for (int i = 1; i <= range; i++)
            {
                Vector3Int nextCoords = occupiedSquare + direction * i;
                Piece piece = board.GetPieceOnSquare(nextCoords);
                if (!board.CheckIfCoordinatesAreOnBoard(nextCoords))
                    break;
                if (piece == null)
                    TryToAddMove(nextCoords);
                else if (!piece.IsFromSameTeam(this))
                {
                    TryToAddMove(nextCoords);
                    break;
                }
                else if (piece.IsFromSameTeam(this))
                    break;
            }
        }
        return availableMoves;
    }
}

```

QUEEN

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Queen : Piece
{
    private Vector3Int[] directions = new Vector3Int[]
    {
        new Vector3Int(0,0,-1),
        new Vector3Int(-1,-1,-1),
        new Vector3Int(-1,0,-1),
        new Vector3Int(-1,1,-1),
        new Vector3Int(0,-1,-1),
        new Vector3Int(0,1,-1),
        new Vector3Int(1,-1,-1),
        new Vector3Int(1,0,-1),
        new Vector3Int(1,1,-1),

        new Vector3Int(-1,-1,1),
        new Vector3Int(-1,0,1),
        new Vector3Int(-1,1,1),
        new Vector3Int(0,-1,1),
        new Vector3Int(0,1,1),
        new Vector3Int(1,-1,1),
        new Vector3Int(1,0,1),
        new Vector3Int(1,1,1),
        new Vector3Int(0,0,1),

        new Vector3Int(-1,-1,0),
        new Vector3Int(-1,0,0),
        new Vector3Int(-1,1,0),
        new Vector3Int(0,-1,0),
        new Vector3Int(0,1,0),
        new Vector3Int(1,-1,0),
        new Vector3Int(1,0,0),
        new Vector3Int(1,1,0),
    };
    public override List<Vector3Int> SelectAvailableSquares()
    {
        availableMoves.Clear();

        float range = Board.BOARD_SIZE;
        foreach (var direction in directions)
        {
            for (int i = 1; i <= range; i++)
            {
                Vector3Int nextCoords = occupiedSquare + direction * i;
                Piece piece = board.GetPieceOnSquare(nextCoords);
                if (!board.CheckIfCoordinatesAreOnBoard(nextCoords))
                    break;
                if (piece == null)
                    TryToAddMove(nextCoords);
                else if (!piece.IsFromSameTeam(this))
                {
                    TryToAddMove(nextCoords);
                    break;
                }
                else if (piece.IsFromSameTeam(this))

```

```

        break;
    }
}
return availableMoves;
}
}

```

KING

```

using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class King : Piece
{
    Vector3Int[] directions = new Vector3Int[]
    {
        new Vector3Int(0,0,-1),
        new Vector3Int(-1,-1,-1),
        new Vector3Int(-1,0,-1),
        new Vector3Int(-1,1,-1),
        new Vector3Int(0,-1,-1),
        new Vector3Int(0,1,-1),
        new Vector3Int(1,-1,-1),
        new Vector3Int(1,0,-1),
        new Vector3Int(1,1,-1),

        new Vector3Int(-1,-1,1),
        new Vector3Int(-1,0,1),
        new Vector3Int(-1,1,1),
        new Vector3Int(0,-1,1),
        new Vector3Int(0,1,1),
        new Vector3Int(1,-1,1),
        new Vector3Int(1,0,1),
        new Vector3Int(1,1,1),
        new Vector3Int(0,0,1),

        new Vector3Int(-1,-1,0),
        new Vector3Int(-1,0,0),
        new Vector3Int(-1,1,0),
        new Vector3Int(0,-1,0),
        new Vector3Int(0,1,0),
        new Vector3Int(1,-1,0),
        new Vector3Int(1,0,0),
        new Vector3Int(1,1,0),
    };

    private Piece leftRook;
    private Piece rightRook;

    private Vector3Int leftCastlingMove;
    private Vector3Int rightCastlingMove;

    public override List<Vector3Int> SelectAvailableSquares()
    {
        availableMoves.Clear();
        AssignStandardMoves();
        AssignCastlingMoves();
    }
}

```

```

        return availableMoves;
    }

    private void AssignCastlingMoves()
    {
        leftCastlingMove = new Vector3Int(-1, -1, 0);
        rightCastlingMove = new Vector3Int(-1, -1, 0);
        if (!hasMoved)
        {
            leftRook = GetPieceInDirection<Rook>(team, new Vector3Int(-1, 0, 0));
            if (leftRook && !leftRook.hasMoved)
            {
                leftCastlingMove = occupiedSquare + new Vector3Int(-1, 0, 0) * 2;
                availableMoves.Add(leftCastlingMove);
            }
            rightRook = GetPieceInDirection<Rook>(team, new Vector3Int(1, 0, 0));
            if (rightRook && !rightRook.hasMoved)
            {
                rightCastlingMove = occupiedSquare + new Vector3Int(1, 0, 0) * 2;
                availableMoves.Add(rightCastlingMove);
            }
        }
    }

    private Piece GetPieceInDirection<T>(TeamColor team, Vector3Int direction)
    {
        for (int i = 1; i <= Board.BOARD_SIZE; i++)
        {
            Vector3Int nextCoords = occupiedSquare + direction * i;
            Piece piece = board.GetPieceOnSquare(nextCoords);
            if (!board.CheckIfCoordinatesAreOnBoard(nextCoords))
                return null;
            if (piece != null)
            {
                if (piece.team != team || !(piece is T))
                    return null;
                else if (piece.team == team && piece is T)
                    return piece;
            }
        }
        return null;
    }

    private void AssignStandardMoves()
    {
        float range = 1;
        foreach (var direction in directions)
        {
            for (int i = 1; i <= range; i++)
            {
                Vector3Int nextCoords = occupiedSquare + direction * i;
                Piece piece = board.GetPieceOnSquare(nextCoords);
                if (!board.CheckIfCoordinatesAreOnBoard(nextCoords))
                    break;
                if (piece == null)
                    TryToAddMove(nextCoords);
                else if (!piece.IsFromSameTeam(this))
                {
                    TryToAddMove(nextCoords);
                    break;
                }
            }
        }
    }
}

```

```

        else if (piece.IsFromSameTeam(this))
            break;
    }
}

public override void MovePiece(Vector3Int coords, Piece piece = null, bool isPawn
= false)
{
    base.MovePiece(coords);
    if (coords == leftCastlingMove)
    {
        board.UpdateBoardOnPieceMove(coords + new Vector3Int(1,0,0),
leftRook.occupiedSquare, leftRook, null);
        leftRook.MovePiece(coords + new Vector3Int(1,0,0));
    }
    else if (coords == rightCastlingMove)
    {
        board.UpdateBoardOnPieceMove(coords + new Vector3Int(-1, 0, 0),
rightRook.occupiedSquare, rightRook, null);
        rightRook.MovePiece(coords + new Vector3Int(1, 0, 0));
    }
}
}

```

MAKING SURE THE KING IS NOT PUT INTO CHECK OR STAYS IN CHECK

This is done during the ‘check if game has ended phase’ as when checking for checkmate it is efficient to also remove the moves from players that allow illegal play involving checks. Here is the controller class function to call the end checking and then the subsequent functions in the chessplayer class that check for illegal moves regarding check.

Also shown here is All moves been generated for both teams – this was spoken about in the section before and is needed before we check for checkmate. We must do this so we know if the next team to play has any legal moves (check/stalemate) and need all of the moves of the current player can now make – this is not as we need to know the move directly, but we do need to know what squares the player is attacking (this allows us to know where discovered check and pins are – so that the correct moves can be removed from the next player to play).

CONTROLLER

```

public void EndTurn()
{
    if(activePlayer.team == TeamColor.White)
    {
        whiteClock.remainingTime += UIManager.timeInc;
        whiteClock.UpdateTimeRemaining(whiteClock.remainingTime);
        whiteClock.timerActive = false;
        blackClock.timerActive = true;
    }
    else
    {
        blackClock.remainingTime += UIManager.timeInc;
        blackClock.UpdateTimeRemaining(blackClock.remainingTime);
        blackClock.timerActive = false;
        whiteClock.timerActive = true;
    }
    moveSound.Play();
}

```

```

        ai.IncreaseNumMovesMade();
        UIManager.DisplayNumMovesMade(ai.GetNumMovesMade());
        GenerateAllPossiblePlayerMoves(activePlayer);
        GenerateAllPossiblePlayerMoves(GetOpponentToPlayer(activePlayer));
        int finNum = CheckIfGameIsFinished();
        if (finNum == 1)
        {
            EndGame();
        }
        else if(finNum == 2)
        {
            SetGameState(GameState.Finished);
            UIManager.OnGameFinished("Draw");
        }
        else
        {
            ChangeActiveTeam();

            if (activePlayer.type == PlayerType.AI)
            {
                RemoveCheckingMoves();
                AIactive = true;
                StartCoroutine(MakeAnAiMove());
                AIactive = false;
            }
            //RotateCameraInstant();
            //RotateCamera();
        }
    }

private int CheckIfGameIsFinished()
{
    Piece[] kingAttackingPieces =
activePlayer.GetPieceAttackingOppositePiceOfType<King>();
    if (kingAttackingPieces.Length > 0)
    {
        ChessPlayer oppositePlayer = GetOpponentToPlayer(activePlayer);
        Piece[] attackedKings = oppositePlayer.GetPiecesOfType<King>();
        foreach (var attackedKing in attackedKings)
        {

oppositePlayer.RemoveMovesEnablingAttackOnPieceOfType<King>(activePlayer,
attackedKing);
            int availableKingMoves = attackedKing.availableMoves.Count;
            if (availableKingMoves == 0)
            {
                bool canCoverKing =
oppositePlayer.CanHidePieceFromAttack<King>(activePlayer);
                if (!canCoverKing)
                    return 1; //mate
            }
        }
    }
    else
    {
        ChessPlayer oppositePlayer = GetOpponentToPlayer(activePlayer);
        Piece[] attackedKings = oppositePlayer.GetPiecesOfType<King>();
        foreach (var attackedKing in attackedKings)
        {
    
```

```

oppositePlayer.RemoveMovesEnablingAttackOnPieceOfType<King>(activePlayer,
attackedKing);
    foreach(Piece piece in oppositePlayer.activePieces)
    {
        if(piece.availableMoves.Count != 0)
        {
            return 0; //carry on
        }
    }
    return 2; //stale
}
return 0; // carry on
public void RemoveCheckingMoves()
{
    foreach(var active in activePlayer.activePieces)
    {

activePlayer.RemoveMovesEnablingAttackOnPieceOfType<King>(GetOpponentToPlayer(activePlayer), active);
    }
}

```

CHESSPLAYER

```

public Piece[] GetPieceAttackingOppositePiceOfType<T>() where T : Piece
{
    return activePieces.Where(p => p.IsAttackingPieceOfType<T>()).ToArray();
}

public Piece[] GetPiecesOfType<T>() where T : Piece
{
    return activePieces.Where(p => p is T).ToArray();
}

public void RemoveMovesEnablingAttackOnPieceOfType<T>(ChessPlayer opponent,
Piece selectedPiece) where T : Piece
{
    List<Vector3Int> coordsToRemove = new List<Vector3Int>();

    coordsToRemove.Clear();
    foreach (var coords in selectedPiece.availableMoves)
    {
        Piece pieceOnCoords = board.GetPieceOnSquare(coords);
        board.UpdateBoardOnPieceMove(coords,
selectedPiece.occupiedSquare, selectedPiece, null);
        opponent.GenerateAllPossibleMoves();
        if (opponent.CheckIfIsAttacingPiece<T>())
            coordsToRemove.Add(coords);
        board.UpdateBoardOnPieceMove(selectedPiece.occupiedSquare,
coords, selectedPiece, pieceOnCoords);
    }
    foreach (var coords in coordsToRemove)
    {
        selectedPiece.availableMoves.Remove(coords);
    }
}

```

```

internal bool CheckIfIsAttacingPiece<T>() where T : Piece
{
    foreach (var piece in activePieces)
    {
        if (board.HasPiece(piece) && piece.IsAttackingPieceOfType<T>())
            return true;
    }
    return false;
}

public bool CanHidePieceFromAttack<T>(ChessPlayer opponent) where T : Piece
{
    foreach (var piece in activePieces)
    {
        foreach (var coords in piece.availiableMoves)
        {
            Piece pieceOnCoords = board.GetPieceOnSquare(coords);
            board.UpdateBoardOnPieceMove(coords, piece.occupiedSquare,
piece, null);
            opponent.GenerateAllPossibleMoves();
            if (!opponent.CheckIfIsAttacingPiece<T>())
            {
                board.UpdateBoardOnPieceMove(piece.occupiedSquare,
coords, piece, pieceOnCoords);
                return true;
            }
            board.UpdateBoardOnPieceMove(piece.occupiedSquare, coords,
piece, pieceOnCoords);
        }
    }
    return false;
}
}

```

REMOVING ILLEGAL CHECK MOVES

[RemoveMovesEnablingAttackOnPieceOfType<T> can be found above in the chessplayer class]

CHESSGAME CONTROLLER

```

public void RemoveCheckingMoves()
{
    foreach(var active in activePlayer.activePieces)
    {

activePlayer.RemoveMovesEnablingAttackOnPieceOfType<King>(GetOpponentToPlayer(activePl
ayer), active);
    }
}

```

BOARD

```

private void SelectPiece(Piece piece, bool AImove = false)
{
    chessController.RemoveMovesEnablingAttackOnPieceOfType<King>(piece);
    selectedPiece = piece;
    List<Vector3Int> selection = selectedPiece.availiableMoves;
    if (!AImove)
    {
        ShowSelectionSquares(selection);
    }
}

```

```
}
```

3.1.5 – TECHNICAL SOLUTION – ALGORITHMS – MERGE SORT

Showing Merge sort, recursion and List Operations

AI MANAGER

```
private List<(float, int[])> MergeSort(List<(float, int[])> nums)
{
    int count = nums.Count;
    if (count <= 1)
    {
        return nums;
    }
    else
    {
        // splitting
        int middle = count / 2; // truncation - will be left mid if even
        number of array elements

        List<(float, int[])> left = new List<(float, int[])>();
        List<(float, int[])> right = new List<(float, int[])>();
        List<(float, int[])> mergedList = new List<(float, int[])>();

        for (int i = 0; i < middle; i++)
        {
            left.Add(nums[i]);
        }
        for (int i = middle; i < count; i++)
        {
            right.Add(nums[i]);
        }

        //recursively split and sort for smaller and smaller list sizes until
        base case met
        left = MergeSort(left);
        right = MergeSort(right);

        // rebuilding back up bigger and bigger lists until back to base
        function call
        while (left.Count > 0 || right.Count > 0)
        {
            if (right.Count > 0)
            {
                if (left.Count > 0)
                {
                    if (left[0].Item1 < right[0].Item1)
                    {
                        mergedList.Add(left[0]); // add left if smaller than
                    right at smallest index
                        left.RemoveAt(0);
                    }
                    else
                    {
                        mergedList.Add(right[0]); //add right if smaller or
                    equal left at smallest index
                        right.RemoveAt(0);
                    }
                }
            }
        }
    }
}
```

```

        }
        else
        {
            foreach ((float, int[]) item in right) // if no left
remaining fill list with right
            {
                mergedList.Add(item);
            }
            return mergedList;
        }
    }
else
{
    foreach ((float, int[]) item in left)// if no right fill list
with left
    {
        mergedList.Add(item);
    }
    return mergedList;
}
return new List<(float, int[])>() { (-1f, new int[0]) };
}

protected (Vector3Int, Vector3Int)
GetMoveFrom1LookAheadWithAccuracy(List<List<Vector3Int>> movesCurrentlyAvailable,
List<Vector3Int> pieceCoords, float accuracy)
{
    List<(float,int[])> positions = new List<(float, int[])>();
    float currentValue = 0;
    for (int i = 0; i < movesCurrentlyAvailable.Count; i++)
    {
        for (int j = 0; j < movesCurrentlyAvailable.ElementAt(i).Count; j++)
        {
            currentValue =
board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j)) == null ? 0
:
pieceNameToValueDict[(board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).Elem
entAt(j))).name];
            positions.Add((currentValue, new int[] { i, j }));
        }
    }
    positions = MergeSort(positions); // orders list by lowest evaluation
    int numMoves = positions.Count-1;
    if (numMoves > 0)
    {
        if (accuracy > 0)
        {
            int indecies = rnd.Next(Mathf.FloorToInt(numMoves - (numMoves *
accuracy)), numMoves);
            int i = positions.ElementAt(indecies).Item2[0];
            int j = positions.ElementAt(indecies).Item2[1];
            return (pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
        }
        else
        {
            int indecies = rnd.Next(0, Mathf.FloorToInt((numMoves)*(-
1*accuracy)));
            int i = positions.ElementAt(indecies).Item2[0];
            int j = positions.ElementAt(indecies).Item2[1];
        }
    }
}

```

```

        return (pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
    }

}
else
{
    return GetMoveRandom(movesCurrentlyAvailable, pieceCoords);
}

}

```

3.2.1 – TECHNICAL SOLUTION – 3RD PARTY CODE – CAMERA FREE LOOK

The code below is code that was not created by myself, but, also is not part of the complexity for the chess or ai solution. It allows for players to use the mouse to control the in game environment as if they were still in the unity scene editing.

```

using UnityEngine;
using System;

public class CameraFreeLook : MonoBehaviour
{
    [Header("Focus Object")]
    [SerializeField, Tooltip("Enable double-click to focus on objects?")]
    private bool doFocus = false;
    [SerializeField] private float focusLimit = 100f;
    [SerializeField] private float minFocusDistance = 5.0f;
    private float doubleClickTime = .15f;
    private float cooldown = 0;
    [Header("Undo Focus")]
    [SerializeField] private KeyCode firstUndoKey = KeyCode.LeftControl;
    [SerializeField] private KeyCode secondUndoKey = KeyCode.Z;

    [Header("Movement")]
    [SerializeField] private float moveSpeed = 1.0f;
    [SerializeField] private float rotationSpeed = 0.025f;
    [SerializeField] private float zoomSpeed = 10.0f;

    //Cache last pos and rot be able to undo last focus object action.
    Quaternion prevRot = new Quaternion();
    Vector3 prevPos = new Vector3();

    [Header("Axes Names")]
    [SerializeField, Tooltip("Vertical axis")] private string mouseY = "Mouse Y";
    [SerializeField, Tooltip("Horizontal axis")] private string mouseX = "Mouse X";
    [SerializeField, Tooltip("Zoom")] private string zoomAxis = "Mouse ScrollWheel";

    [Header("Move Keys")]
    [SerializeField] private KeyCode forwardKey = KeyCode.W;
    [SerializeField] private KeyCode backKey = KeyCode.S;
    [SerializeField] private KeyCode leftKey = KeyCode.A;
    [SerializeField] private KeyCode rightKey = KeyCode.D;
    [SerializeField] private KeyCode upKey = KeyCode.Q;
    [SerializeField] private KeyCode downKey = KeyCode.E;
    [SerializeField, Tooltip("Key to stop zoom")] private KeyCode stopZoom =
KeyCode.LeftControl;

    [Header("Anchored Movement"), Tooltip("By default in scene-view, this is done by
right-clicking for rotation or middle mouse clicking for up and down")]

```

```

[SerializeField] private KeyCode anchoredMoveKey = KeyCode.Mouse2;
[SerializeField] private KeyCode anchoredRotateKey = KeyCode.Mouse1;

private void Start()
{
    SavePosAndRot();
}

void Update()
{
    if (!doFocus)
        return;

    //Double click for focus
    if (cooldown > 0 && Input.GetKeyDown(KeyCode.Mouse0))
        FocusObject();
    if (Input.GetKeyDown(KeyCode.Mouse0))
        cooldown = doubleClickTime;

    //-----UNDO FOCUS-----
    if (Input.GetKey(firstUndoKey))
    {
        if (Input.GetKeyDown(secondUndoKey))
            GoBackToLastPosition();
    }

    cooldown -= Time.deltaTime;
}

private void LateUpdate()
{
    Vector3 move = Vector3.zero;

    //Move and rotate the camera

    if (Input.GetKey(forwardKey))
        move += Vector3.forward * moveSpeed;
    if (Input.GetKey(backKey))
        move += Vector3.back * moveSpeed;
    if (Input.GetKey(leftKey))
        move += Vector3.left * moveSpeed;
    if (Input.GetKey(rightKey))
        move += Vector3.right * moveSpeed;
    if (Input.GetKey(upKey))
        move += Vector3.up * moveSpeed;
    if (Input.GetKey(downKey))
        move += Vector3.down * moveSpeed;

    float mouseMoveY = Input.GetAxis(mouseY);
    float mouseMoveX = Input.GetAxis(mouseX);

    //Move the camera when anchored
    if (Input.GetKey(anchoredMoveKey))
    {
        move += Vector3.up * mouseMoveY * -moveSpeed;
        move += Vector3.right * mouseMoveX * -moveSpeed;
    }

    //Rotate the camera when anchored
    if (Input.GetKey(anchoredRotateKey))
    {

```

```

        transform.RotateAround(transform.position, transform.right, mouseMoveY * -
rotationSpeed);
        transform.RotateAround(transform.position, Vector3.up, mouseMoveX * -
rotationSpeed);
    }

    transform.Translate(move);

    //Scroll to zoom
    if (!Input.GetKeyDown(stopZoom))
    {
        float mouseScroll = Input.GetAxis(zoomAxis);
        transform.Translate(Vector3.forward * mouseScroll * zoomSpeed);
    }
}

private void FocusObject()
{
    //To be able to undo
    SavePosAndRot();

    //If we double-clicked an object in the scene, go to its position
    Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
    RaycastHit hit;

    if (Physics.Raycast(ray, out hit, focusLimit))
    {
        GameObject target = hit.collider.gameObject;
        Vector3 targetPos = target.transform.position;
        Vector3 targetSize = hit.collider.bounds.size;

        transform.position = targetPos + GetOffset(targetPos, targetSize);

        transform.LookAt(target.transform);
    }
}

private void SavePosAndRot()
{
    prevRot = transform.rotation;
    prevPos = transform.position;
}

private void GoBackToLastPosition()
{
    transform.position = prevPos;
    transform.rotation = prevRot;
}

private Vector3 GetOffset(Vector3 targetPos, Vector3 targetSize)
{
    Vector3 dirToTarget = targetPos - transform.position;

    float focusDistance = Mathf.Max(targetSize.x, targetSize.z);
    focusDistance = Mathf.Clamp(focusDistance, minFocusDistance, focusDistance);

    return -dirToTarget.normalized * focusDistance;
}
}

```

3.3 – ALL CODE (NEXT HEADERS REPRESENT CLASS NAMES)

CHESS AI

AI1

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class AI1 : AIManager
{
    public AI1 (ChessPlayer currentAiPlayer, ChessPlayer opposingPlayer, Board board)
    {
        this.currentAiPlayer = currentAiPlayer;
        this.board = board;
        this.opposingPlayer = opposingPlayer;
    }
    public override void MoveMakerController(ChessPlayer aiPlayer, ChessPlayer
opposingPlayer)
    {
        NewAiPlayers(aiPlayer, opposingPlayer);
        List<List<Vector3Int>> movesCurrentlyAvailable = new List<List<Vector3Int>>();
        List<Piece> pieces = new List<Piece>();
        List<Vector3Int> pieceCoords = new List<Vector3Int>();
        (movesCurrentlyAvailable, pieces, pieceCoords) =
currentAiPlayer.ReturnAllPossibleMoves();

        Vector3Int coordsOfPieceToMove;
        Vector3Int coordsToMoveTo;

        if(rnd.Next(1,11) == 5)
        {
            (coordsOfPieceToMove, coordsToMoveTo) =
GetMoveFrom1LookAheadWithAccuracy(movesCurrentlyAvailable, pieceCoords, -0.1f);
            Debug.Log("Blunder");
        }
        else if (rnd.Next(1, 3) == 1 )
        {
            (coordsOfPieceToMove, coordsToMoveTo) =
GetMoveFrom1LookAheadWithAccuracy(movesCurrentlyAvailable, pieceCoords, 0.25f);
        }
        else if(rnd.Next(1,6) == 5)
        {
            (coordsOfPieceToMove, coordsToMoveTo) =
GetMoveRandom(movesCurrentlyAvailable, pieceCoords);
            Debug.Log("Blunder");
        }
        else
        {
            (coordsOfPieceToMove, coordsToMoveTo) =
GetMoveMostValueFrom2LookAheadWithAccuracy(movesCurrentlyAvailable,pieces,
pieceCoords, 0.25f);
        }

        board.AIMakeMove(coordsOfPieceToMove, coordsToMoveTo);
    }
}
```

AI2

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class AI2 : AIManager
{
    public AI2 (ChessPlayer currentAiPlayer, ChessPlayer opposingPlayer, Board board)
    {
        this.currentAiPlayer = currentAiPlayer;
        this.board = board;
        this.opposingPlayer = opposingPlayer;
    }

    public override void MoveMakerController(ChessPlayer aiPlayer, ChessPlayer
opposingPlayer)
    {
        NewAiPlayers(aiPlayer, opposingPlayer);
        List<List<Vector3Int>> movesCurrentlyAvailable = new List<List<Vector3Int>>();
        List<Piece> pieces = new List<Piece>();
        List<Vector3Int> pieceCoords = new List<Vector3Int>();
        (movesCurrentlyAvailable, pieces, pieceCoords) =
currentAiPlayer.ReturnAllPossibleMoves();

        Vector3Int coordsOfPieceToMove = Vector3Int.zero;
        Vector3Int coordsToMoveTo = Vector3Int.zero;
        int randomVal = rnd.Next(0, 100);

        if (randomVal < 15)//15%
        {
            (coordsOfPieceToMove, coordsToMoveTo) =
GetMoveMostValueFrom2LookAheadWithAccuracy(movesCurrentlyAvailable, pieces,
pieceCoords, -0.15f);
            Debug.Log("Blunder");
        }
        else if (randomVal < 20)//20%
        {
            (coordsOfPieceToMove, coordsToMoveTo) =
GetMoveMostValueFrom2LookAheadWithAccuracy(movesCurrentlyAvailable, pieces,
pieceCoords, -0.5f, -0.25f);
            Debug.Log("Innacuracy");
        }
        else
        {
            (coordsOfPieceToMove, coordsToMoveTo) =
GetMoveMostValueFrom2LookAheadWithAccuracyAndLargeMoveNegativityMitigation(movesCurren
tlyAvailable, pieces, pieceCoords, 0.15f);
        }

        board.AIMakeMove(coordsOfPieceToMove, coordsToMoveTo);
    }
}
```

AI3

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class AI3 : AIManager
{
```

```

public AI3(ChessPlayer currentAiPlayer, ChessPlayer opposingPlayer, Board board)
{
    this.currentAiPlayer = currentAiPlayer;
    this.board = board;
    this.opposingPlayer = opposingPlayer;
}

public override void MoveMakerController(ChessPlayer aiPlayer, ChessPlayer
opposingPlayer)
{
    NewAiPlayers(aiPlayer, opposingPlayer);
    List<List<Vector3Int>> movesCurrentlyAvailable = new List<List<Vector3Int>>();
    List<Piece> pieces = new List<Piece>();
    List<Vector3Int> pieceCoords = new List<Vector3Int>();
    (movesCurrentlyAvailable, pieces, pieceCoords) =
currentAiPlayer.ReturnAllPossibleMoves();

    Vector3Int coordsOfPieceToMove = Vector3Int.zero;
    Vector3Int coordsToMoveTo = Vector3Int.zero;
    int randomVal = rnd.Next(0, 100);

    if (randomVal < 2)//2%
    {
        (coordsOfPieceToMove, coordsToMoveTo) =
GetMoveMostValueFrom2LookAheadWithAccuracy(movesCurrentlyAvailable, pieces,
pieceCoords, -0.15f);
        Debug.Log("Blunder");
    }
    else if (randomVal < 7)//2+5=7
    {
        (coordsOfPieceToMove, coordsToMoveTo) =
GetMoveMostValueFrom2LookAheadWithAccuracy(movesCurrentlyAvailable, pieces,
pieceCoords, -0.5f, -0.25f);
        Debug.Log("Innacuracy");
    }
    else
    {
        if(rnd.Next(2) == 1)
        {
            (coordsOfPieceToMove, coordsToMoveTo) =
GetMoveMostValueFrom2LookAheadWithAccuracyWithAdditionalTheory(movesCurrentlyAvailable
, pieces, pieceCoords, 0.15f);
        }
        else
        {
            (coordsOfPieceToMove, coordsToMoveTo) =
GetMoveMostValueFrom2LookAheadWithAccuracyAndDeeperLineEvaluationWithAdditionalTheory(
movesCurrentlyAvailable, pieces, pieceCoords, 0.15f);
        }
    }
    board.AIMakeMove(coordsOfPieceToMove, coordsToMoveTo);
}
}

```

AI4

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class AI4 : AIManager

```

```

{
    public AI4 (ChessPlayer currentAiPlayer, ChessPlayer opposingPlayer, Board board)
    {
        this.currentAiPlayer = currentAiPlayer;
        this.board = board;
        this.opposingPlayer = opposingPlayer;
    }

    public override void MoveMakerController(ChessPlayer aiPlayer, ChessPlayer
opposingPlayer)
    {
        NewAiPlayers(aiPlayer, opposingPlayer);
        List<List<Vector3Int>> movesCurrentlyAvailable = new List<List<Vector3Int>>();
        List<Piece> pieces = new List<Piece>();
        List<Vector3Int> pieceCoords = new List<Vector3Int>();
        (movesCurrentlyAvailable, pieces, pieceCoords) =
currentAiPlayer.ReturnAllPossibleMoves();

        Vector3Int coordsOfPieceToMove = Vector3Int.zero;
        Vector3Int coordsToMoveTo = Vector3Int.zero;
        int randomVal = rnd.Next(0, 10000);

        if (randomVal < 75)//0.75%
        {
            (coordsOfPieceToMove, coordsToMoveTo) =
GetMoveMostValueFrom2LookAheadWithAccuracy(movesCurrentlyAvailable, pieces,
pieceCoords, -0.15f);
            Debug.Log("Blunder");
        }
        else if (randomVal < 300)//3%
        {
            (coordsOfPieceToMove, coordsToMoveTo) =
GetMoveMostValueFrom2LookAheadWithAccuracy(movesCurrentlyAvailable, pieces,
pieceCoords, -0.5f, -0.25f);
            Debug.Log("Innacuracy");
        }
        else
        {
            (coordsOfPieceToMove, coordsToMoveTo) =
GetMoveMostValueFrom2LookAheadAndDeeperLineEvaluationWithPositionalNeuralNetwork(moves
CurrentlyAvailable, pieces, pieceCoords);
        }

        board.AIMakeMove(coordsOfPieceToMove, coordsToMoveTo);
    }
}

```

AI5

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class AI5 : AIManager
{
    public AI5 (ChessPlayer currentAiPlayer, ChessPlayer opposingPlayer, Board board)
    {
        this.currentAiPlayer = currentAiPlayer;
        this.board = board;
        this.opposingPlayer = opposingPlayer;
    }
}

```

```

    public override void MoveMakerController(ChessPlayer aiPlayer, ChessPlayer
opposingPlayer)
{
    NewAiPlayers(aiPlayer, opposingPlayer);
    List<List<Vector3Int>> movesCurrentlyAvailable = new List<List<Vector3Int>>();
    List<Piece> pieces = new List<Piece>();
    List<Vector3Int> pieceCoords = new List<Vector3Int>();
    (movesCurrentlyAvailable, pieces, pieceCoords) =
currentAiPlayer.ReturnAllPossibleMoves();

    Vector3Int coordsOfPieceToMove = Vector3Int.zero;
    Vector3Int coordsToMoveTo = Vector3Int.zero;
    (coordsOfPieceToMove, coordsToMoveTo) =
GetMoveMostValueFrom2LookAheadAndDeeperLineEvaluationWithPositionalNeuralNetwork(moves
CurrentlyAvailable, pieces, pieceCoords);

    board.AIMakeMove(coordsOfPieceToMove, coordsToMoveTo);
}
}

```

AI MANAGER

```

using System;
using System.Linq;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

```

```
public abstract class AIManager
```

```
{
    protected ChessPlayer currentAiPlayer {get; set;}
    protected ChessPlayer opposingPlayer { get; set; }
    protected Board board { get; set; }
```

```
protected Dictionary<String, int> pieceNameToValueDict = new Dictionary<String, int>()
```

```
{
    {"King", 1 }, //Point for putting king in check
    {"Pawn", 1 },
    {"Bishop", 3 },
    {"Knight", 3 },
```

```

        {"Commoner", 3 },
        {"Rook", 5 },
        {"Queen", 9 },

        {"King(Clone)", 1 },
        {"Pawn(Clone)", 1 },
        {"Bishop(Clone)", 3 },
        {"Knight(Clone)", 3 },
        {"Commoner(Clone)", 3 },
        {"Rook(Clone)", 5 },
        {"Queen(Clone)", 9 },
    };

```

```

protected static int[] layers = new int[] {1,1,1,1};

protected float[] inputState;

protected NNManager networkManager = new NNManager(layers,activationType.tanh,
costType.tanhCustom);

```

```

protected RandomNumber rnd = new RandomNumber();

protected int movesMade;

```

```

protected void NewAiPlayers (ChessPlayer newAiPlayer, ChessPlayer opposingPlayer)

{
    this.currentAiPlayer = newAiPlayer;
    this.opposingPlayer = opposingPlayer;
}

```

```
public abstract void MoveMakerController(ChessPlayer aiPlayer, ChessPlayer opposingPlayer);
```

```
protected (Vector3Int, Vector3Int) GetMoveRandom(List<List<Vector3Int>> movesCurrentlyAvailable,
List<Vector3Int> pieceCoords)
```

```

{
    int val = Guid.NewGuid().GetHashCode() % pieceCoords.Count;

    int atIndex = val > 0 ? val : -val;

    val = Guid.NewGuid().GetHashCode() % movesCurrentlyAvailable.ElementAt(atIndex).Count;

    int toIndex = val > 0 ? val : -val;

    return (pieceCoords.ElementAt(atIndex),
movesCurrentlyAvailable.ElementAt(atIndex).ElementAt(toIndex));
}

```

```

private float BasicTheoryAddition(Vector3Int toMoveCoords, List<Vector3Int> attackCoords, bool
whitePiece)

{
    float addition = 0f;

    if((toMoveCoords.x == 4 || toMoveCoords.x == 3)&&(toMoveCoords.y == 4 || toMoveCoords.y == 3))
    {
        addition += 0.4f;
    }
    if (whitePiece)
    {
        if(toMoveCoords.y != 0)
        {
            addition += 0.1f;
        }
    }
    else
    {
        if(toMoveCoords.y != 7)
        {
            addition += 0.1f;
        }
    }
}

foreach(Vector3Int coords in attackCoords)
{
    if ((coords.x == 4 || coords.x == 3) && (coords.y == 4 || coords.y == 3)) // middle
    {
        addition += 0.1f;
    }
}

return addition;

```

```

    }

private float BasicOpeningTheoryAddition(Vector3Int coords, Vector3Int attackCoords)
{
    float addition = 0;

    Piece piece = board.GetPieceOnSquare(attackCoords);

    if (piece != null)
    {
        addition -= pieceNameToValueDict[piece.name] *0.05f;

        if ((coords.x == 4 || coords.x == 3) && (coords.y == 4 || coords.y == 3)) // middle
        {
            addition += 1f;
        }
    }

    piece = board.GetPieceOnSquare(coords);

    if (piece != null && coords.z == board.getPieceCoords("King", piece.team).z)
    {
        addition += 0.1f;
    }
}

return addition;
}

private float BasicMiddleGameTheoryAddition(Vector3Int toMoveCoords, Vector3Int attackCoords)
{
    float addition = 0;

    Piece piece = board.GetPieceOnSquare(attackCoords);

    if (piece != null)
    {
        addition += pieceNameToValueDict[piece.name] *0.05f;
    }

    return addition;
}

```

```

    }

private float BasicEndGameTheoryAddition(Vector3Int toMoveCoords, Vector3Int attackCoords)
{
    float addition = 0;

    if (board.GetPieceOnSquare(toMoveCoords) != null)

    {
        if (board.GetPieceOnSquare(toMoveCoords).name == "Pawn" ||
            board.GetPieceOnSquare(toMoveCoords).name == "Pawn(Clone)")

        {
            addition += 0.5f * attackCoords.y;
        }

        else if (board.GetPieceOnSquare(toMoveCoords).name == "King" ||
            board.GetPieceOnSquare(toMoveCoords).name == "King(Clone)")

        {
            int numPiecesProtecting = 0;

            for (int i = -1; i < 2; i += 2)

            {
                for (int j = -1; j < 2; j += 2)

                {
                    for (int k = -1; k < 2; k += 2)

                    {
                        if (board.GetPieceOnSquare(new Vector3Int(attackCoords.x + i, attackCoords.y + j,
                            attackCoords.z + k)) != null)

                        {
                            numPiecesProtecting++;
                        }
                    }
                }
            }
            addition += 0.05f * numPiecesProtecting;
        }
    }
}

```

```

    }

    return addition;
}

private float IntermediateTheoryAddition(Vector3Int toMoveCoords, Vector3Int attackCoords)
{
    float addition = 0;

    Piece piece = board.GetPieceOnSquare(toMoveCoords);

    if (piece != null)
    {
        if (piece.name == "Knight" || piece.name == "Knight(Clone)")

        {
            if (attackCoords.x == 0 || attackCoords.x == 7 || attackCoords.y == 0 || attackCoords.y == 7 ||
attackCoords.z == 0 || attackCoords.z == 7)

            {
                addition -= 0.3f;
            }
        }
        else if (piece.name == "Rook" || piece.name == "Rook(Clone)")

        {
            if (attackCoords.x == 0 || attackCoords.x == 7 || attackCoords.y == 0 || attackCoords.y == 7 ||
attackCoords.z == 0 || attackCoords.z == 7)

            {
                addition -= 0.3f;
            }
        }
    }

    return addition;
}

```

```

protected (Vector3Int, Vector3Int) GetMoveMostValueFromILookAhead(List<List<Vector3Int>>
movesCurrentlyAvailable, List<Vector3Int> pieceCoords)

{
    List<int[]> maxValuePositions = new List<int[]>();

    int maxValue = 0;

    int currentValue = 0;

    for (int i = 0; i < movesCurrentlyAvailable.Count; i++)
    {
        for (int j = 0; j < movesCurrentlyAvailable.ElementAt(i).Count; j++)
        {
            currentValue = board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j)) ==
null ? -1 :
pieceNameToValueDict[(board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j))).name];
            if (currentValue > maxValue)
            {
                maxValuePositions.Clear();
                maxValuePositions.Add(new int[] { i, j });
            }
            if(currentValue == maxValue)
            {
                maxValuePositions.Add(new int[] { i, j });
            }
        }
    }

    int numBestMoves = maxValuePositions.Count;
    if (numBestMoves > 0)
    {
        int indecies = rnd.Next(maxValuePositions.Count);

```

```

        return (pieceCoords.ElementAt(maxValuePositions.ElementAt(indexes)[0]),
movesCurrentlyAvailable.ElementAt(maxValuePositions.ElementAt(indexes)[0]).ElementAt(maxValuePositions.
ElementAt(indexes)[1]));
    }

    else

    {

        return GetMoveRandom(movesCurrentlyAvailable, pieceCoords);

    }

}

private List<(float, int[])> MergeSort(List<(float, int[])> nums)
{
    int count = nums.Count;
    if (count <= 1)
    {
        return nums;
    }
    else
    {
        // splitting
        int middle = count / 2; // truncation - will be left mid if even number of array elements

        List<(float, int[])> left = new List<(float, int[])>();
        List<(float, int[])> right = new List<(float, int[])>();
        List<(float, int[])> mergedList = new List<(float, int[])>();

        for (int i = 0; i < middle; i++)
        {
            left.Add(nums[i]);
        }
        for (int i = middle; i < count; i++)
        {
            right.Add(nums[i]);
        }

        //recursively split and sort for smaller and smaller list sizes until base case met
        left = MergeSort(left);
        right = MergeSort(right);

        // rebuilding back up bigger and bigger lists until back to base function call
        while (left.Count > 0 || right.Count > 0)
        {
            if (right.Count > 0)
            {
                if (left.Count > 0)
                {
                    if (left[0].Item1 < right[0].Item1)
                    {
                        mergedList.Add(left[0]); // add left if smaller than right at smallest index
                        left.RemoveAt(0);
                    }
                }
            }
        }
    }
}

```

```

        else
        {
            mergedList.Add(right[0]); //add right if smaller or equal left at smallest index
            right.RemoveAt(0);
        }
    }
    else
    {
        foreach ((float, int[])) item in right) // if no left remaining fill list with right
        {
            mergedList.Add(item);
        }
        return mergedList;
    }
}
else
{
    foreach ((float, int[])) item in left)// if no right fill list with left
    {
        mergedList.Add(item);
    }
    return mergedList;
}
return new List<(float, int[])>() { (-1f, new int[0])};
}
}

protected (Vector3Int, Vector3Int) GetMoveFromILookAheadWithAccuracy(List<List<Vector3Int>>
movesCurrentlyAvailable, List<Vector3Int> pieceCoords, float accuracy)
{
    List<(float,int[])> positions = new List<(float, int[])>();
    float currentValue = 0;
    for (int i = 0; i < movesCurrentlyAvailable.Count; i++)
    {
        for (int j = 0; j < movesCurrentlyAvailable.ElementAt(i).Count; j++)
        {
            currentValue = board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j)) ==
null ? 0 :
pieceNameToValueDict[(board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j))).name];
            positions.Add((currentValue, new int[] { i, j }));
        }
    }
    positions = MergeSort(positions); // orders list by lowest evaluation
    int numMoves = positions.Count - 1;
    if (numMoves > 0)
    {
        if (accuracy > 0)
        {
            int indecies = rnd.Next(Mathf.FloorToInt(numMoves - (numMoves * accuracy)), numMoves);
            int i = positions.ElementAt(indecies).Item2[0];
            int j = positions.ElementAt(indecies).Item2[1];
            return (pieceCoords.ElementAt(i), movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
        }
    }
    else
    {
        int indecies = rnd.Next(0, Mathf.FloorToInt((numMoves)*(-1*accuracy)));
        int i = positions.ElementAt(indecies).Item2[0];
    }
}

```

```

        int j = positions.ElementAt(indexes).Item2[1];
        return (pieceCoords.ElementAt(i), movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
    }

}
else
{
    return GetMoveRandom(movesCurrentlyAvailable, pieceCoords);
}

}

protected (Vector3Int, Vector3Int) GetMoveMostValueFrom2LookAhead(List<List<Vector3Int>>
movesCurrentlyAvailable, List<Piece> pieces, List<Vector3Int> pieceCoords)
{
    if(board.pastMovesBlack.Count > 3)

    {
        if (currentAiPlayer.team == TeamColor.White)

        {
            if (board.pastMovesWhite.ElementAt(board.pastMovesWhite.Count - 1) ==
board.pastMovesWhite.ElementAt(board.pastMovesWhite.Count - 3))

            {
                return GetMoveRandom(movesCurrentlyAvailable, pieceCoords);
            }
        }
        else

        {
            if (board.pastMovesBlack.ElementAt(board.pastMovesBlack.Count - 1) ==
board.pastMovesBlack.ElementAt(board.pastMovesBlack.Count - 3))

            {
                return GetMoveRandom(movesCurrentlyAvailable, pieceCoords);
            }
        }
    }

    List<List<Vector3Int>> opposingMovesCurrentlyAvailable = new List<List<Vector3Int>>();
    List<Piece> opposingPieces = new List<Piece>();

```

```

List<Vector3Int> opposingPieceCoords = new List<Vector3Int>();
Piece oldPiece;

List<int[]> maxValuePositions = new List<int[]>();

int maxValue = 0;
int currentValue = 0;

for (int i = 0; i < movesCurrentlyAvailable.Count; i++)
{
    for (int j = 0; j < movesCurrentlyAvailable.ElementAt(i).Count; j++)
    {
        oldPiece = board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

        board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
pieceCoords.ElementAt(i),pieces.ElementAt(i), null );

        (opposingMovesCurrentlyAvailable, opposingPieces, opposingPieceCoords) =
opposingPlayer.ReturnAllPossibleMoves();

        currentValue = board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j)) ==
null ? 0 :
pieceNameToValueDict[(board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j))).name];
    }

    for (int k = 0; k < opposingMovesCurrentlyAvailable.Count; k++)
    {
        for (int l = 0; l < opposingMovesCurrentlyAvailable.ElementAt(k).Count; l++)
        {
            currentValue -=
board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l)) == null ? 0 :
pieceNameToValueDict[(board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l))).name];

            if (currentValue > maxValue)
            {
                maxValuePositions.Clear();
                maxValuePositions.Add(new int[] { i, j });
            }
            if (currentValue == maxValue)
        }
    }
}

```

```

    {
        maxValuePositions.Add(new int[] { i, j });
    }
}

}

board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
pieceCoords.ElementAt(i), oldPiece, pieces.ElementAt(i));

}

}

int numBestMoves = maxValuePositions.Count;

if (numBestMoves > 0)

{
    int val = Guid.NewGuid().GetHashCode() % numBestMoves;

    val = val > 0 ? val : -val;

    return (pieceCoords.ElementAt(maxValuePositions.ElementAt(val)[0]),
movesCurrentlyAvailable.ElementAt(maxValuePositions.ElementAt(val)[0]).ElementAt(maxValuePositions.ElementAt(val)[1]));
}

else

{
    return GetMoveRandom(movesCurrentlyAvailable, pieceCoords);
}

}

protected (Vector3Int, Vector3Int)
GetMoveMostValueFrom2LookAheadWithAccuracy(List<List<Vector3Int>> movesCurrentlyAvailable,
List<Piece> pieces, List<Vector3Int> pieceCoords, float accuracyDefault, float accuracyRange = 0f)

{
    List<List<Vector3Int>> opposingMovesCurrentlyAvailable = new List<List<Vector3Int>>();

    List<Piece> opposingPieces = new List<Piece>();

    List<Vector3Int> opposingPieceCoords = new List<Vector3Int>();

```

```

Piece oldPiece;

SortedDictionary<float, int[]> positions = new SortedDictionary<float, int[]>();

float currentValue = 0;

float small = 0.00001f; // prevents same keys

for (int i = 0; i < movesCurrentlyAvailable.Count; i++)

{

    for (int j = 0; j < movesCurrentlyAvailable.ElementAt(i).Count; j++)

    {

        oldPiece = board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

        board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),

pieceCoords.ElementAt(i), pieces.ElementAt(i), null);

        (opposingMovesCurrentlyAvailable, opposingPieces, opposingPieceCoords) = opposingPlayer.ReturnAllPossibleMoves();

        currentValue = board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j)) == null ? 0 :

pieceNameToValueDict[(board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j))).name];

        currentValue +=

BasicTheoryAddition(pieceCoords.ElementAt(i),movesCurrentlyAvailable.ElementAt(i),board.GetPieceOnSquare(pieceCoords.ElementAt(i)) == null? true: board.GetPieceOnSquare(pieceCoords.ElementAt(i)).team == TeamColor.White ? true:false);

        for (int k = 0; k < opposingMovesCurrentlyAvailable.Count; k++)

        {

            for (int l = 0; l < opposingMovesCurrentlyAvailable.ElementAt(k).Count; l++)

            {

                currentValue -=

board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l)) == null ? 0 :

pieceNameToValueDict[(board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l))).name];

                currentValue -=

BasicTheoryAddition(opposingPieceCoords.ElementAt(k),opposingMovesCurrentlyAvailable.ElementAt(k), board.GetPieceOnSquare(pieceCoords.ElementAt(i)) == null ? true :

board.GetPieceOnSquare(pieceCoords.ElementAt(i)).team == TeamColor.White ? true : false);

                while (positions.ContainsKey(currentValue)){currentValue -=

0.00001f;}positions.Add(currentValue, new int[] { i, j });

            }

        }

    }

}


```

```

        small += 0.00001f;

    }

}

board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
pieceCoords.ElementAt(i), oldPiece, pieces.ElementAt(i));

}

}

int numMoves = positions.Count - 1;

if (numMoves > 0)

{

    if (accuracyRange == 0)

    {

        if (accuracyDefault > 0)

        {

            int indecies = rnd.Next(Mathf.FloorToInt(positions.Count - (positions.Count * accuracyDefault)),
positions.Count);

            int i = positions.ElementAt(indecies - 1).Value[0];

            int j = positions.ElementAt(indecies - 1).Value[1];

            return (pieceCoords.ElementAt(i), movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

        }

        else

        {

            int indecies = rnd.Next(0, Mathf.FloorToInt(positions.Count * (-1 * accuracyDefault)));

            int i = positions.ElementAt(indecies - 1).Value[0];

            int j = positions.ElementAt(indecies - 1).Value[1];

            return (pieceCoords.ElementAt(i), movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

        }

    }

    else

    {

        if (accuracyDefault > 0)

```

```

    {

        int indecies = rnd.Next(Mathf.FloorToInt(positions.Count - (positions.Count * accuracyRange)),
        Mathf.FloorToInt(positions.Count - (positions.Count * accuracyDefault)));

        int i = positions.ElementAt(indecies - 1).Value[0];

        int j = positions.ElementAt(indecies - 1).Value[1];

        return (pieceCoords.ElementAt(i), movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

    }

    else

    {

        int indecies = rnd.Next(Mathf.FloorToInt(positions.Count * (-1 * accuracyRange)),
        Mathf.FloorToInt(positions.Count * (-1 * accuracyDefault))));

        int i = positions.ElementAt(indecies - 1).Value[0];

        int j = positions.ElementAt(indecies - 1).Value[1];

        return (pieceCoords.ElementAt(i), movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

    }

}

else

{

    return GetMoveRandom(movesCurrentlyAvailable, pieceCoords);

}

}

protected (Vector3Int, Vector3Int) GetMoveMostValueFrom2LookAheadWithNN(List<List<Vector3Int>>
movesCurrentlyAvailable, List<Piece> pieces, List<Vector3Int> pieceCoords)

{
    List<List<Vector3Int>> opposingMovesCurrentlyAvailable = new List<List<Vector3Int>>();

    List<Piece> opposingPieces = new List<Piece>();

    List<Vector3Int> opposingPieceCoords = new List<Vector3Int>();

    Piece oldPiece;
}

```

```

Piece oldOPiece;

SortedDictionary<float, int[]> positions = new SortedDictionary<float, int[]>();

float currentValue = 0;

float small = 0.00001f; // prevents same keys

for (int i = 0; i < movesCurrentlyAvailable.Count; i++)

{

    for (int j = 0; j < movesCurrentlyAvailable.ElementAt(i).Count; j++)

    {

        oldPiece = board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

        board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
pieceCoords.ElementAt(i), pieces.ElementAt(i), null);

        (opposingMovesCurrentlyAvailable, opposingPieces, opposingPieceCoords) =
opposingPlayer.ReturnAllPossibleMoves();

        currentValue = board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j)) ==
null ? 0 :
pieceNameToValueDict[(board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j))).name
];

        currentValue += GetNNEvalFromBoardState(board.grid);

        if (board.majorPiecesMoved < 50)

        {

            currentValue += BasicOpeningTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

        }

        else if (board.majorPiecesTaken < 50)

        {

            currentValue += BasicMiddleGameTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

        }

        else

        {

            currentValue += BasicEndGameTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

        }

    }

}

}

```

```

    }

    for (int k = 0; k < opposingMovesCurrentlyAvailable.Count; k++)
    {

        for (int l = 0; l < opposingMovesCurrentlyAvailable.ElementAt(k).Count; l++)
        {

            oldOPiece =
            board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l));

            board.UpdateBoardOnPieceMove(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l),
            opposingPieceCoords.ElementAt(k), opposingPieces.ElementAt(k), null);

            currentValue -=
            board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l)) == null ? 0 :
            pieceNameToValueDict[(board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l))).name];

            currentValue += GetNNEvalFromBoardState(board.grid);

            board.UpdateBoardOnPieceMove(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l),
            opposingPieceCoords.ElementAt(k), oldOPiece, opposingPieces.ElementAt(k));

            if (board.majorPiecesMoved < 50)

            {

                currentValue -=
                BasicOpeningTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
                opposingPieces.ElementAt(k).availableMoves.ElementAt(l));

            }

            else if (board.majorPiecesTaken < 50)

            {

                currentValue -=
                BasicMiddleGameTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
                opposingPieces.ElementAt(k).availableMoves.ElementAt(l));

            }

            else

            {

                currentValue -=
                BasicEndGameTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
                opposingPieces.ElementAt(k).availableMoves.ElementAt(l));

            }

        }
    }
}

```

```

        while (positions.ContainsKey(currentValue)) { currentValue -= 0.00001f; }

        positions.Add(currentValue, new int[] { i, j });

        small += 0.00001f;

    }

}

board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
pieceCoords.ElementAt(i), oldPiece, pieces.ElementAt(i));

}

int numMoves = positions.Count - 1;

if (numMoves > 0)

{

    int i = positions.Last().Value[0];

    int j = positions.Last().Value[1];

    return (pieceCoords.ElementAt(i), movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

}

else

{

    return GetMoveRandom(movesCurrentlyAvailable, pieceCoords);

}

}

protected (Vector3Int, Vector3Int)
GetMoveMostValueFrom2LookAheadWithAccuracyWithAdditionalTheory(List<List<Vector3Int>>
movesCurrentlyAvailable, List<Piece> pieces, List<Vector3Int> pieceCoords, float accuracyDefault, float
accuracyRange = 0f)

{
    List<List<Vector3Int>> opposingMovesCurrentlyAvailable = new List<List<Vector3Int>>();

    List<Piece> opposingPieces = new List<Piece>();

    List<Vector3Int> opposingPieceCoords = new List<Vector3Int>();

    Piece oldPiece;
}

```

```

SortedDictionary<float, int[]> positions = new SortedDictionary<float, int[]>();

float currentValue = 0;

float small = 0.00001f; // prevents same keys

for (int i = 0; i < movesCurrentlyAvailable.Count; i++)
{
    for (int j = 0; j < movesCurrentlyAvailable.ElementAt(i).Count; j++)
    {
        oldPiece = board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

        board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
pieceCoords.ElementAt(i), pieces.ElementAt(i), null);

        (opposingMovesCurrentlyAvailable, opposingPieces, opposingPieceCoords) =
opposingPlayer.ReturnAllPossibleMoves();

        currentValue = board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j)) ==
null ? 0 :
(pieceNameToValueDict[(board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j))).name] +
board.materialImbalance*0.05f);

        currentValue += IntermediateTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

        if (board.majorPiecesMoved < 50)
        {
            currentValue += BasicOpeningTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

        }
        else if (board.majorPiecesTaken < 50)
        {
            currentValue += BasicMiddleGameTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

        }
        else
        {
            currentValue += BasicEndGameTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
        }
    }
}

```

```

        for (int k = 0; k < opposingMovesCurrentlyAvailable.Count; k++)
    {
        for (int l = 0; l < opposingMovesCurrentlyAvailable.ElementAt(k).Count; l++)
    {
        currentValue -=
board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l)) == null ? 0 :
(pieceNameToValueDict[(board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementA
t(l))).name] +board.materialImbalance * 0.05f);

        currentValue -= IntermediateTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(l));

        if (board.majorPiecesMoved < 50)
    {
        currentValue -=
BasicOpeningTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(l));
    }
        else if (board.majorPiecesTaken < 50)
    {
        currentValue -=
BasicMiddleGameTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(l));
    }
        else
    {
        currentValue -=
BasicEndGameTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(l));
    }
}

while (positions.ContainsKey(currentValue)){currentValue -=
0.00001f;}positions.Add(currentValue, new int[] { i, j });

small += 0.00001f;
}
}

```

```

        board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
pieceCoords.ElementAt(i), oldPiece, pieces.ElementAt(i));

    }

}

int numMoves = positions.Count - 1;

if (numMoves > 0)

{

    if (accuracyRange == 0)

    {

        if (accuracyDefault > 0)

        {

            int indecies = rnd.Next(Mathf.FloorToInt(positions.Count - (positions.Count * accuracyDefault)),
positions.Count);

            int i = positions.ElementAt(indecies - 1).Value[0];

            int j = positions.ElementAt(indecies - 1).Value[1];

            return (pieceCoords.ElementAt(i), movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

        }

    }

    else

    {

        int indecies = rnd.Next(0, Mathf.FloorToInt(positions.Count * (-1 * accuracyDefault)));

        int i = positions.ElementAt(indecies - 1).Value[0];

        int j = positions.ElementAt(indecies - 1).Value[1];

        return (pieceCoords.ElementAt(i), movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

    }

}

else

{

    if (accuracyDefault > 0)

    {

        int indecies = rnd.Next(Mathf.FloorToInt(positions.Count - (positions.Count * accuracyRange)),
Mathf.FloorToInt(positions.Count - (positions.Count * accuracyDefault)));

```

```

        int i = positions.ElementAt(indexes - 1).Value[0];
        int j = positions.ElementAt(indexes - 1).Value[1];
        return (pieceCoords.ElementAt(i), movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
    }
    else
    {
        int index = rnd.Next(Mathf.FloorToInt(positions.Count * (-1 * accuracyRange)),
            Mathf.FloorToInt(positions.Count * (-1 * accuracyDefault)));
        int i = positions.ElementAt(index - 1).Value[0];
        int j = positions.ElementAt(index - 1).Value[1];
        return (pieceCoords.ElementAt(i), movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
    }
}
else
{
    return GetMoveRandom(movesCurrentlyAvailable, pieceCoords);
}
}

protected (float, Vector3Int, Vector3Int)
GetMoveMostValueFrom2LookAheadWithNNAndReturnEval(List<List<Vector3Int>>
movesCurrentlyAvailable, List<Piece> pieces, List<Vector3Int> pieceCoords)
{
    List<List<Vector3Int>> opposingMovesCurrentlyAvailable = new List<List<Vector3Int>>();
    List<Piece> opposingPieces = new List<Piece>();
    List<Vector3Int> opposingPieceCoords = new List<Vector3Int>();
    Piece oldPiece;
    Piece oldOPiece;
}

```

```

SortedDictionary<float, int[]> positions = new SortedDictionary<float, int[]>();
float currentValue = 0;
float small = 0.00001f; // prevents same keys
for (int i = 0; i < movesCurrentlyAvailable.Count; i++)
{
    for (int j = 0; j < movesCurrentlyAvailable.ElementAt(i).Count; j++)
    {
        oldPiece = board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
        board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
pieceCoords.ElementAt(i), pieces.ElementAt(i), null);
        (opposingMovesCurrentlyAvailable, opposingPieces, opposingPieceCoords) =
opposingPlayer.ReturnAllPossibleMoves();
        currentValue = board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j)) ==
null ? 0 :
pieceNameToValueDict[(board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j))).name];
        currentValue += GetNNEvalFromBoardState(board.grid);
        if (board.majorPiecesMoved < 50)
        {
            currentValue += BasicOpeningTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
        }
        else if (board.majorPiecesTaken < 50)
        {
            currentValue += BasicMiddleGameTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
        }
        else
        {
            currentValue += BasicEndGameTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
        }
    }
    for (int k = 0; k < opposingMovesCurrentlyAvailable.Count; k++)

```

```

{
    for (int l = 0; l < opposingMovesCurrentlyAvailable.ElementAt(k).Count; l++)
    {
        oldOPiece =
            board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l));
        board.UpdateBoardOnPieceMove(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l),
            opposingPieceCoords.ElementAt(k), opposingPieces.ElementAt(k), null);

        currentValue -=
            board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l)) == null ? 0 :
            pieceNameToValueDict[(board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l))).name];

        currentValue += GetNNEvalFromBoardState(board.grid);

        board.UpdateBoardOnPieceMove(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l),
            opposingPieceCoords.ElementAt(k), oldOPiece, opposingPieces.ElementAt(k));

        if (board.majorPiecesMoved < 50)
        {
            currentValue -=
                BasicOpeningTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
                    opposingPieces.ElementAt(k).availableMoves.ElementAt(l));
        }
        else if (board.majorPiecesTaken < 50)
        {
            currentValue -=
                BasicMiddleGameTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
                    opposingPieces.ElementAt(k).availableMoves.ElementAt(l));
        }
        else
        {
            currentValue -=
                BasicEndGameTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
                    opposingPieces.ElementAt(k).availableMoves.ElementAt(l));
        }
        while (positions.ContainsKey(currentValue)){currentValue -=
            0.00001f;}positions.Add(currentValue, new int[] { i, j });
    }
}

```

```

        small += 0.00001f;

    }

}

board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
pieceCoords.ElementAt(i), oldPiece, pieces.ElementAt(i));

}

}

int numMoves = positions.Count - 1;

if (numMoves > 0)

{

    int i = positions.Last().Value[0];

    int j = positions.Last().Value[1];

    return (positions.Last().Key, pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

}

else

{

    (Vector3Int, Vector3Int) tempHolder = GetMoveRandom(movesCurrentlyAvailable, pieceCoords);

    return (-1f, tempHolder.Item1, tempHolder.Item2);

}

}

protected (float, Vector3Int, Vector3Int)
GetMoveMostValueFrom2LookAheadWithAccuracyAndReturnEval(List<List<Vector3Int>>
movesCurrentlyAvailable, List<Piece> pieces, List<Vector3Int> pieceCoords, float accuracyDefault, float
accuracyRange = 0f)

{
    List<List<Vector3Int>> opposingMovesCurrentlyAvailable = new List<List<Vector3Int>>();

    List<Piece> opposingPieces = new List<Piece>();

    List<Vector3Int> opposingPieceCoords = new List<Vector3Int>();

    Piece oldPiece;
}

```

```

SortedDictionary<float, int[]> positions = new SortedDictionary<float, int[]>();

float currentValue = 0;

float small = 0.00001f; // prevents same keys

for (int i = 0; i < movesCurrentlyAvailable.Count; i++)
{
    for (int j = 0; j < movesCurrentlyAvailable.ElementAt(i).Count; j++)
    {
        oldPiece = board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

        board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
pieceCoords.ElementAt(i), pieces.ElementAt(i), null);

        (opposingMovesCurrentlyAvailable, opposingPieces, opposingPieceCoords) =
opposingPlayer.ReturnAllPossibleMoves();

        currentValue = board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j)) ==
null ? 0 :
pieceNameToValueDict[(board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j))).name
];

        currentValue += BasicTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i), board.GetPieceOnSquare(pieceCoords.ElementAt(i)) == null ? true :
board.GetPieceOnSquare(pieceCoords.ElementAt(i)).team == TeamColor.White ? true : false);

        if (board.majorPiecesMoved < 50)
        {
            currentValue += BasicOpeningTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
        }
        else if (board.majorPiecesTaken < 50)
        {
            currentValue += BasicMiddleGameTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
        }
        else
        {
            currentValue += BasicEndGameTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
        }
    }
}

```

```

    }

    for (int k = 0; k < opposingMovesCurrentlyAvailable.Count; k++)
    {

        for (int l = 0; l < opposingMovesCurrentlyAvailable.ElementAt(k).Count; l++)
        {

            currentValue -=
board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l)) == null ? 0 :
pieceNameToValueDict[(board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l))).name];

            currentValue -= BasicTheoryAddition(opposingPieceCoords.ElementAt(k),
opposingMovesCurrentlyAvailable.ElementAt(k), board.GetPieceOnSquare(pieceCoords.ElementAt(l))) == null
? true : board.GetPieceOnSquare(pieceCoords.ElementAt(l)).team == TeamColor.White ? true : false);

            if (board.majorPiecesMoved < 50)

            {

                currentValue -=
BasicOpeningTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(l));

            }

            else if (board.majorPiecesTaken < 50)

            {

                currentValue -=
BasicMiddleGameTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(l));

            }

            else

            {

                currentValue -=
BasicEndGameTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(l));

            }

            while (positions.ContainsKey(currentValue)) { currentValue -= 0.00001f; }

            positions.Add(currentValue, new int[] { i, j });

            small += 0.00001f;

        }

    }

}

```

```

        board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
pieceCoords.ElementAt(i), oldPiece, pieces.ElementAt(i));

    }

}

int numMoves = positions.Count - 1;

if (numMoves > 0)

{

    if (accuracyRange == 0)

    {

        if (accuracyDefault > 0)

        {

            int indecies = rnd.Next(Mathf.FloorToInt(positions.Count - (positions.Count * accuracyDefault)),
positions.Count);

            int i = positions.ElementAt(indecies - 1).Value[0];

            int j = positions.ElementAt(indecies - 1).Value[1];

            return (positions.ElementAt(indecies).Key, pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

        }

    }

    else

    {

        int indecies = rnd.Next(0, Mathf.FloorToInt(positions.Count * (-1 * accuracyDefault)));

        int i = positions.ElementAt(indecies - 1).Value[0];

        int j = positions.ElementAt(indecies - 1).Value[1];

        return (positions.ElementAt(indecies).Key, pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

    }

}

else

{

    if (accuracyDefault > 0)

    {

```

```

        int indecies = rnd.Next(Mathf.FloorToInt(positions.Count - (positions.Count * accuracyRange)),
Mathf.FloorToInt(positions.Count - (positions.Count * accuracyDefault)));

        int i = positions.ElementAt(indecies - 1).Value[0];

        int j = positions.ElementAt(indecies - 1).Value[1];

        return (positions.ElementAt(indecies).Key, pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

    }

    else

    {

        int indecies = rnd.Next(Mathf.FloorToInt(positions.Count * (-1 * accuracyRange)),
Mathf.FloorToInt(positions.Count * (-1 * accuracyDefault)));

        int i = positions.ElementAt(indecies - 1).Value[0];

        int j = positions.ElementAt(indecies - 1).Value[1];

        return (positions.ElementAt(indecies).Key, pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

    }

}

else

{

    (Vector3Int, Vector3Int) tempHolder = GetMoveRandom(movesCurrentlyAvailable, pieceCoords);

    return (-1f, tempHolder.Item1, tempHolder.Item2);

}

}

protected (Vector3Int, Vector3Int)
GetMoveMostValueFrom2LookAheadWithAccuracyAndDeeperLineEvaluationWithAdditionalTheory(List<List
<Vector3Int>> movesCurrentlyAvailable, List<Piece> pieces, List<Vector3Int> pieceCoords, float
accuracyDefault, float accuracyRange = 0f)

{
    List<List<Vector3Int>> opposingMovesCurrentlyAvailable = new List<List<Vector3Int>>();

    List<Piece> opposingPieces = new List<Piece>();

    List<Vector3Int> opposingPieceCoords = new List<Vector3Int>();
}

```

```

Piece oldPiece;

SortedDictionary<float, int[]> positions = new SortedDictionary<float, int[]>();
List<(Vector3Int, Vector3Int)> furtherPositions = new List<(Vector3Int, Vector3Int)>();
float currentValue = 0;

float small = 0.00001f; // prevents same keys

for (int i = 0; i < movesCurrentlyAvailable.Count; i++)
{
    for (int j = 0; j < movesCurrentlyAvailable.ElementAt(i).Count; j++)
    {
        oldPiece = board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

        board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
pieceCoords.ElementAt(i), pieces.ElementAt(i), null);

        (opposingMovesCurrentlyAvailable, opposingPieces, opposingPieceCoords) =
opposingPlayer.ReturnAllPossibleMoves();

        currentValue = board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j)) ==
null ? 0 :
(pieceNameToValueDict[(board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j))).nam
e] + board.materialImbalance * 0.05f);

        currentValue += IntermediateTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

        if(board.majorPiecesMoved < 50)
        {
            currentValue += BasicOpeningTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
        }
        else if(board.majorPiecesTaken < 50)
        {
            currentValue += BasicMiddleGameTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
        }
    }
}

```

```

{
    currentValue += BasicEndGameTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

}

for (int k = 0; k < opposingMovesCurrentlyAvailable.Count; k++)
{

    for (int l = 0; l < opposingMovesCurrentlyAvailable.ElementAt(k).Count; l++)
    {

        currentValue -=
board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l)) == null ? 0 :
(pieceNameToValueDict[(board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l)))].name] + board.materialImbalance * 0.05f;

        currentValue -= IntermediateTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(l));

        if (board.majorPiecesMoved < 50)

        {

            currentValue -=
BasicOpeningTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(l));

        }

        else if (board.majorPiecesTaken < 50)

        {

            currentValue -=
BasicMiddleGameTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(l));

        }

        else

        {

            currentValue -=
BasicEndGameTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(l));

        }

        while (positions.ContainsKey(currentValue)){currentValue -=
0.00001f;}positions.Add(currentValue, new int[] { i, j });

        small += 0.00001f;
    }
}

```

```

        }

    }

    board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
pieceCoords.ElementAt(i), oldPiece, pieces.ElementAt(i));

}

int numMoves = positions.Count - 1;

if (numMoves > 0)

{

    if (accuracyRange == 0)

    {

        if (accuracyDefault > 0)

        {

            for (int a = 0; a < 3; a++)

            {

                int indecies = rnd.Next(Mathf.FloorToInt(positions.Count - (positions.Count *
accuracyDefault)), positions.Count);

                int i = positions.ElementAt(indecies - 1).Value[0];

                int j = positions.ElementAt(indecies - 1).Value[1];

                furtherPositions.Add((pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j)));

            }

        }

    }

    else

    {

        for (int a = 0; a < 3; a++)

        {

            int indecies = rnd.Next(0, Mathf.FloorToInt(positions.Count * (-1 * accuracyDefault)));

            int i = positions.ElementAt(indecies - 1).Value[0];

            int j = positions.ElementAt(indecies - 1).Value[1];

            furtherPositions.Add((pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j)));

        }

    }

}

```

```

        }

    }

else
{
    if (accuracyDefault > 0)

    {

        for (int a = 0; a < 3; a++)
        {

            int indecies = rnd.Next(Mathf.FloorToInt(positions.Count - (positions.Count *
accuracyRange)), Mathf.FloorToInt(positions.Count - (positions.Count * accuracyDefault)));

            int i = positions.ElementAt(indecies - 1).Value[0];
            int j = positions.ElementAt(indecies - 1).Value[1];
            furtherPositions.Add((pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j)));
        }
    }

    else
    {
        for (int a = 0; a < 3; a++)
        {

            int indecies = rnd.Next(Mathf.FloorToInt(positions.Count * (-1 * accuracyRange)),
Mathf.FloorToInt(positions.Count * (-1 * accuracyDefault)));
            int i = positions.ElementAt(indecies - 1).Value[0];
            int j = positions.ElementAt(indecies - 1).Value[1];
            furtherPositions.Add((pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j)));
        }
    }
}

```

```

        }

    }

    List<(float, Vector3Int, Vector3Int)> furtherPositionsWithEval = new List<(float, Vector3Int,
Vector3Int)>();

    for (int a = 0; a < 3; a++)

    {

        List<List<Vector3Int>> furtherTempMovesCurrentlyAvailable = new List<List<Vector3Int>>();

        List<Vector3Int> miniFurtherTempMovesCurrentlyAvailable = new List<Vector3Int>();

        List<Piece> furtherTempPieces = new List<Piece>();

        List<Vector3Int> furtherTempPieceCoords = new List<Vector3Int>();




        miniFurtherTempMovesCurrentlyAvailable.Add(furtherPositions.ElementAt(a).Item2);

        furtherTempMovesCurrentlyAvailable.Add(miniFurtherTempMovesCurrentlyAvailable);

        furtherTempPieces.Add(board.GetPieceOnSquare(furtherPositions.ElementAt(a).Item1));

        furtherTempPieceCoords.Add(furtherPositions.ElementAt(a).Item1);




        (Vector3Int, Vector3Int) halfWayMoveChosen =  

GetMoveMostValueFrom2LookAheadWithAccuracy(furtherTempMovesCurrentlyAvailable,  

furtherTempPieces, furtherTempPieceCoords, 0.15f);






        miniFurtherTempMovesCurrentlyAvailable.Clear();

        furtherTempMovesCurrentlyAvailable.Clear();

        furtherTempPieces.Clear();

        furtherTempPieceCoords.Clear();

        miniFurtherTempMovesCurrentlyAvailable.Add(halfWayMoveChosen.Item2);

        furtherTempMovesCurrentlyAvailable.Add(miniFurtherTempMovesCurrentlyAvailable);

        furtherTempPieces.Add(board.GetPieceOnSquare(halfWayMoveChosen.Item1));

        furtherTempPieceCoords.Add(halfWayMoveChosen.Item1);




furtherPositionsWithEval.Add(GetMoveMostValueFrom2LookAheadWithAccuracyAndReturnEval(furtherTempMovesCurrentlyAvailable, furtherTempPieces, furtherTempPieceCoords, 0.15f));

```

```

    }

    float move1 = furtherPositionsWithEval.ElementAt(0).Item1;

    float move2 = furtherPositionsWithEval.ElementAt(1).Item1;

    float move3 = furtherPositionsWithEval.ElementAt(2).Item1;

    if (move1 > move2)

    {

        if (move1 > move3)

        {

            return (furtherPositionsWithEval.ElementAt(0).Item2,
furtherPositionsWithEval.ElementAt(0).Item3);

        }

        else

        {

            return (furtherPositionsWithEval.ElementAt(2).Item2,
furtherPositionsWithEval.ElementAt(2).Item3);

        }

    }

    else

    {

        if (move2 > move3)

        {

            return (furtherPositionsWithEval.ElementAt(1).Item2,
furtherPositionsWithEval.ElementAt(1).Item3);

        }

        else

        {

            return (furtherPositionsWithEval.ElementAt(2).Item2,
furtherPositionsWithEval.ElementAt(2).Item3);

        }

    }

}

else

```

```

    {

        return GetMoveRandom(movesCurrentlyAvailable, pieceCoords);

    }

}

protected (Vector3Int, Vector3Int)
GetMoveMostValueFrom2LookAheadAndDeeperLineEvaluationWithPositionalNeuralNetwork(List<List<Vector3Int>> movesCurrentlyAvailable, List<Piece> pieces, List<Vector3Int> pieceCoords)

{
    List<List<Vector3Int>> opposingMovesCurrentlyAvailable = new List<List<Vector3Int>>();
    List<Piece> opposingPieces = new List<Piece>();
    List<Vector3Int> opposingPieceCoords = new List<Vector3Int>();

    Piece oldPiece;
    Piece oldOPiece;

    SortedDictionary<float, int[]> positions = new SortedDictionary<float, int[]>();
    List<(Vector3Int, Vector3Int)> furtherPositions = new List<(Vector3Int, Vector3Int)>();
    float currentValue = 0;

    float small = 0.00001f; // prevents same keys

    for (int i = 0; i < movesCurrentlyAvailable.Count; i++)
    {
        for (int j = 0; j < movesCurrentlyAvailable.ElementAt(i).Count; j++)
        {
            oldPiece = board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
            board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
                pieceCoords.ElementAt(i), pieces.ElementAt(i), null);

            (opposingMovesCurrentlyAvailable, opposingPieces, opposingPieceCoords) =
                opposingPlayer.ReturnAllPossibleMoves();

            currentValue = board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j)) ==
                null ? 0 :

```

```

(pieceNameToValueDict[(board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j))).name]
e] + board.materialImbalance * 0.05f);

    currentValue += GetNNEvalFromBoardState(board.grid);

    if (board.majorPiecesMoved < 50)

    {

        currentValue += BasicOpeningTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

    }

    else if (board.majorPiecesTaken < 50)

    {

        currentValue += BasicMiddleGameTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

    }

    else

    {

        currentValue += BasicEndGameTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

    }

    for (int k = 0; k < opposingMovesCurrentlyAvailable.Count; k++)

    {

        for (int l = 0; l < opposingMovesCurrentlyAvailable.ElementAt(k).Count; l++)

        {

            oldOPiece =
board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l));

            board.UpdateBoardOnPieceMove(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l),
opposingPieceCoords.ElementAt(k), opposingPieces.ElementAt(k), null);

            currentValue -=
board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l)) == null ? 0 :
pieceNameToValueDict[(board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l))).name];

            currentValue += GetNNEvalFromBoardState(board.grid);

            board.UpdateBoardOnPieceMove(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l),
opposingPieceCoords.ElementAt(k), oldOPiece, opposingPieces.ElementAt(k));

```

```

if (board.majorPiecesMoved < 50)
{
    currentValue -=
BasicOpeningTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(l));

}
else if (board.majorPiecesTaken < 50)

{
    currentValue -=
BasicMiddleGameTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(l));

}
else
{
    currentValue -=
BasicEndGameTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves.ElementAt(l));

}

while (positions.ContainsKey(currentValue)) { currentValue -= 0.00001f; }

positions.Add(currentValue, new int[] { i, j });

small += 0.00001f;

}
}

board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
pieceCoords.ElementAt(i), oldPiece, pieces.ElementAt(i));

}

int numMoves = positions.Count - 1;

if (numMoves > 0)

{
    for (int a = 0; a < 5; a++)
    {
        int i = positions.Last().Value[0];

```

```

        int j = positions.Last().Value[1];

        furtherPositions.Add((pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i).ElementAt(j)));

    }

    List<(float, Vector3Int, Vector3Int)> furtherPositionsWithEval = new List<(float, Vector3Int,
Vector3Int)>();

    for (int a = 0; a < 5; a++)

    {

        List<List<Vector3Int>> furtherTempMovesCurrentlyAvailable = new List<List<Vector3Int>>();

        List<Vector3Int> miniFurtherTempMovesCurrentlyAvailable = new List<Vector3Int>();

        List<Piece> furtherTempPieces = new List<Piece>();

        List<Vector3Int> furtherTempPieceCoords = new List<Vector3Int>();




        miniFurtherTempMovesCurrentlyAvailable.Add(furtherPositions.ElementAt(a).Item2);

        furtherTempMovesCurrentlyAvailable.Add(miniFurtherTempMovesCurrentlyAvailable);

        furtherTempPieces.Add(board.GetPieceOnSquare(furtherPositions.ElementAt(a).Item1));

        furtherTempPieceCoords.Add(furtherPositions.ElementAt(a).Item1);





        (Vector3Int, Vector3Int) halfWayMoveChosen =
GetMoveMostValueFrom2LookAheadWithNN(furtherTempMovesCurrentlyAvailable, furtherTempPieces,
furtherTempPieceCoords);





        miniFurtherTempMovesCurrentlyAvailable.Clear();

        furtherTempMovesCurrentlyAvailable.Clear();

        furtherTempPieces.Clear();

        furtherTempPieceCoords.Clear();

        miniFurtherTempMovesCurrentlyAvailable.Add(halfWayMoveChosen.Item2);

        furtherTempMovesCurrentlyAvailable.Add(miniFurtherTempMovesCurrentlyAvailable);

        furtherTempPieces.Add(board.GetPieceOnSquare(halfWayMoveChosen.Item1));

        furtherTempPieceCoords.Add(halfWayMoveChosen.Item1);
    }
}

```

```

        furtherPositionsWithEval.Add(GetMoveMostValueFrom2LookAheadWithNNAndReturnEval(furtherTempMov
esCurrentlyAvailable, furtherTempPieces, furtherTempPieceCoords));

    }

    int maxIndex = 0;

    float max = 0;

    for (int i = 0; i < 5; i++)

    {

        if(furtherPositionsWithEval.ElementAt(i).Item1 > max)

        {

            maxIndex = i;

        }

    }

    return (furtherPositionsWithEval.ElementAt(maxIndex).Item2,
furtherPositionsWithEval.ElementAt(maxIndex).Item3);

}

else

{

    return GetMoveRandom(movesCurrentlyAvailable, pieceCoords);

}

}

```

```

private float GetNNEvalFromBoardState(Piece[,] grid)

{

    float[] inputs = new float[512*7*2];

    int counter = 0;

    for (int i = 0; i < 8; i++)

    {

        for (int j = 0; j < 8; j++)

        {

```

```

for (int k = 0; k < 8; k++)
{
    if (grid[i,j,k] != null)
    {
        if(grid[i,j,k].team == TeamColor.White)
        {
            if (grid[i, j, k].name == "Pawn" || grid[i, j, k].name == "Pawn(Clone)")
            {
                inputs[counter] = 1;
            }

            else if (grid[i, j, k].name == "Knight" || grid[i, j, k].name == "Knight(Clone)")
            {
                inputs[counter+1] = 1;
            }

            else if (grid[i, j, k].name == "Bishop" || grid[i, j, k].name == "Bishop(Clone)")
            {
                inputs[counter+2] = 1;
            }

            else if (grid[i,j,k].name == "Rook" || grid[i, j, k].name == "Rook(Clone)")
            {
                inputs[counter+3] = 1;
            }

            else if (grid[i,j,k].name == "Commoner" || grid[i, j, k].name == "Commoner(Clone)")
            {
                inputs[counter+4] = 1;
            }

            else if (grid[i, j, k].name == "King" || grid[i, j, k].name == "King(Clone)")
            {
                inputs[counter+5] = 1;
            }
        }
    }
}

```

```

else if (grid[i, j, k].name == "Queen" || grid[i, j, k].name == "Queen(Clone)")
{
    inputs[counter+6] = l;
}

counter += 14;

}

else

{
    counter += 7;

    if (grid[i, j, k].name == "Pawn" || grid[i, j, k].name == "Pawn(Clone)")

    {
        inputs[counter] = l;
    }

    else if (grid[i, j, k].name == "Knight" || grid[i, j, k].name == "Knight(Clone)")

    {
        inputs[counter + 1] = l;
    }

    else if (grid[i, j, k].name == "Bishop" || grid[i, j, k].name == "Bishop(Clone)")

    {
        inputs[counter + 2] = l;
    }

    else if (grid[i, j, k].name == "Rook" || grid[i, j, k].name == "Rook(Clone)")

    {
        inputs[counter + 3] = l;
    }

    else if (grid[i, j, k].name == "Commoner" || grid[i, j, k].name == "Commoner(Clone)")

    {
        inputs[counter + 4] = l;
    }

    else if (grid[i, j, k].name == "King" || grid[i, j, k].name == "King(Clone)")

```

```

    {
        inputs[counter + 5] = l;
    }

    else if (grid[i, j, k].name == "Queen" || grid[i, j, k].name == "Queen(Clone)")
    {
        inputs[counter + 6] = l;
    }

    counter += 7;

}

}

else
{
    counter += 14;
}

}

}

}

return networkManager.FeedForward(inputs)[0];//it is okay to do this as we know there is only one
output state (the eval)
}

protected (Vector3Int, Vector3Int)
GetMoveMostValueFrom2LookAheadWithAccuracyAndLargeMoveNegativityMitigation(List<List<Vector3Int>>
movesCurrentlyAvailable, List<Piece> pieces, List<Vector3Int> pieceCoords, float accuracyDefault, float
accuracyRange = 0f)
{
    List<List<Vector3Int>> opposingMovesCurrentlyAvailable = new List<List<Vector3Int>>();
    List<Piece> opposingPieces = new List<Piece>();
    List<Vector3Int> opposingPieceCoords = new List<Vector3Int>();
    Piece oldPiece;
}

```

```

SortedDictionary<float, int[]> positions = new SortedDictionary<float, int[]>();
float currentValue = 0;
float small = 0.00001f; // prevents same keys
for (int i = 0; i < movesCurrentlyAvailable.Count; i++)
{
    for (int j = 0; j < movesCurrentlyAvailable.ElementAt(i).Count; j++)
    {
        oldPiece = board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j));
        board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
pieceCoords.ElementAt(i), pieces.ElementAt(i), null);
        (opposingMovesCurrentlyAvailable, opposingPieces, opposingPieceCoords) =
opposingPlayer.ReturnAllPossibleMoves();
        currentValue = board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j)) ==
null ? 0 :
pieceNameToValueDict[(board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j))).name];
        currentValue += BasicTheoryAddition(pieceCoords.ElementAt(i),
movesCurrentlyAvailable.ElementAt(i), board.GetPieceOnSquare(pieceCoords.ElementAt(i))) == null ? true :
board.GetPieceOnSquare(pieceCoords.ElementAt(i)).team == TeamColor.White ? true : false);
        for (int k = 0; k < opposingMovesCurrentlyAvailable.Count; k++)
        {
            for (int l = 0; l < opposingMovesCurrentlyAvailable.ElementAt(k).Count; l++)
            {
                currentValue -=
board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l)) == null ? 0 :
pieceNameToValueDict[(board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l))).name];
                currentValue -= BasicTheoryAddition(opposingPieces.ElementAt(k).occupiedSquare,
opposingPieces.ElementAt(k).availableMoves, board.GetPieceOnSquare(opposingPieceCoords.ElementAt(i)))
== null ? true : board.GetPieceOnSquare(opposingPieceCoords.ElementAt(i)).team == TeamColor.White ?
false : true);
                while (positions.ContainsKey(currentValue)){currentValue -=
0.00001f;}positions.Add(currentValue, new int[] { i, j });
                small += 0.00001f;
            }
        }
    }
}

```

```

        board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
pieceCoords.ElementAt(i), oldPiece, pieces.ElementAt(i));

    }

}

int numMoves = positions.Count - 1;

if (numMoves > 0)

{

    if (accuracyRange == 0)

    {

        if (accuracyDefault > 0)

        {

            if(positions.ElementAt(Mathf.FloorToInt(positions.Count - Mathf.FloorToInt(positions.Count -
(positions.Count * accuracyDefault))/2).Key < 0)

            {

                // no need to create an additional list to only add values in the accuracy range as the above
takes care of this at the same time as the 50% negative implementation discussed in 4.a.ii.7 in objectives

                // this is because we know that the first negative value must lie within this range

                SortedDictionary<float, int[]> positivePositions = new SortedDictionary<float, int[]>();

                foreach(KeyValuePair<float, int[]> item in positions)

                {

                    if(item.Key >= 0)

                    {

                        positivePositions.Add(item.Key,item.Value);

                    }

                }

            }

            int indecies = rnd.Next(0,positivePositions.Count);

            int i = positions.ElementAt(indecies - 1).Value[0];

            int j = positions.ElementAt(indecies - 1).Value[1];

            return (pieceCoords.ElementAt(i), movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

        }

    }

}

```

```

        else

    {

        int indecies = rnd.Next(Mathf.FloorToInt(positions.Count - (positions.Count *
accuracyDefault)), positions.Count);

        int i = positions.ElementAt(indecies - 1).Value[0];

        int j = positions.ElementAt(indecies - 1).Value[1];

        return (pieceCoords.ElementAt(i), movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

    }

}

else

{

    int indecies = rnd.Next(0, Mathf.FloorToInt(positions.Count * (-1 * accuracyDefault)));

    int i = positions.ElementAt(indecies - 1).Value[0];

    int j = positions.ElementAt(indecies - 1).Value[1];

    return (pieceCoords.ElementAt(i), movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

}

else

{

    if (accuracyDefault > 0)

    {

        int indecies = rnd.Next(Mathf.FloorToInt(positions.Count - (positions.Count * accuracyRange)),
Mathf.FloorToInt(positions.Count - (positions.Count * accuracyDefault)));

        int i = positions.ElementAt(indecies - 1).Value[0];

        int j = positions.ElementAt(indecies - 1).Value[1];

        return (pieceCoords.ElementAt(i), movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

    }

}

```

```

        int indecies = rnd.Next(Mathf.FloorToInt(positions.Count * (-1 * accuracyRange)),
Mathf.FloorToInt(positions.Count * (-1 * accuracyDefault)));

        int i = positions.ElementAt(indecies - 1).Value[0];

        int j = positions.ElementAt(indecies - 1).Value[1];

        return (pieceCoords.ElementAt(i), movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

    }

}

else

{

    return GetMoveRandom(movesCurrentlyAvailable, pieceCoords);

}

}

protected (Vector3Int, Vector3Int)
GetMoveByAlphaBetaPruningWithMovePriorityOrdering(List<List<Vector3Int>> movesCurrentlyAvailable,
List<Piece> pieces, List<Vector3Int> pieceCoords, int runNum=0)

{
    List<List<Vector3Int>> opposingMovesCurrentlyAvailable = new List<List<Vector3Int>>();

    List<Piece> opposingPieces = new List<Piece>();

    List<Vector3Int> opposingPieceCoords = new List<Vector3Int>();

    Piece oldPiece;

    List<int[]> maxValuePositions = new List<int[]>();

    int alpha = 0;

    int beta = 9;

    int currentValue = 0;

    for (int i = 0; i < movesCurrentlyAvailable.Count; i++)

    {

        for (int j = 0; j < movesCurrentlyAvailable.ElementAt(i).Count; j++)

```

```

{
    oldPiece = board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j));

    board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
pieceCoords.ElementAt(i), pieces.ElementAt(i), null);

    (opposingMovesCurrentlyAvailable, opposingPieces, opposingPieceCoords) =
opposingPlayer.ReturnAllPossibleMoves();

    (opposingMovesCurrentlyAvailable, opposingPieces, opposingPieceCoords) =
GiveMovesPriorityOrder(opposingMovesCurrentlyAvailable, opposingPieces, opposingPieceCoords,
movesCurrentlyAvailable);

    currentValue = board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j)) ==
null ? 0 :
pieceNameToValueDict[(board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j))).name
];

    string pieceName = pieces.ElementAt(i).name;

    if (pieceName == "Pawn" || pieceName == "Pawn(Clone)")

    {
        currentValue += 9;
    }

    for (int k = 0; k < opposingMovesCurrentlyAvailable.Count; k++)

    {
        for (int l = 0; l < opposingMovesCurrentlyAvailable.ElementAt(k).Count; l++)

        {
            currentValue -=
board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l)) == null ? 0 :
pieceNameToValueDict[(board.GetPieceOnSquare(opposingMovesCurrentlyAvailable.ElementAt(k).ElementAt(l))).name];

            if (pieceName == "Pawn" || pieceName == "Pawn(Clone)")

            {
                currentValue -= 9;
            }

            if (currentValue < beta)

            {
                break;
            }
        }
    }
}

```

```

        if (currentValue > alpha)
    {
        maxValuePositions.Clear();
        maxValuePositions.Add(new int[] { i, j });
    }
    else if (currentValue == alpha)
    {
        maxValuePositions.Add(new int[] { i, j });
    }
    if (currentValue < beta)
    {
        break;
    }
}

board.UpdateBoardOnPieceMove(movesCurrentlyAvailable.ElementAt(i).ElementAt(j),
pieceCoords.ElementAt(i), oldPiece, pieces.ElementAt(i));
}

int numBestMoves = maxValuePositions.Count;
if (numBestMoves > 0)
{
    int val = Guid.NewGuid().GetHashCode() % numBestMoves;
    val = val > 0 ? val : -val;
    return (pieceCoords.ElementAt(maxValuePositions.ElementAt(val)[0]),
movesCurrentlyAvailable.ElementAt(maxValuePositions.ElementAt(val)[0]).ElementAt(maxValuePositions.ElementAt(val)[1]));
}
else
{
    return GetMoveRandom(movesCurrentlyAvailable, pieceCoords);
}

```

```

        }

    }

protected (List<List<Vector3Int>>, List<Piece>, List<Vector3Int>)
GiveMovesPriorityOrder(List<List<Vector3Int>> moves, List<Piece> pieces, List<Vector3Int> positions,
List<List<Vector3Int>> opposingAttacks)

{
    List<List<Vector3Int>> orderedMoves = new List<List<Vector3Int>>();
    List<Vector3Int> orderedPositions = new List<Vector3Int>();
    List<Piece> orderedPieces = new List<Piece>();

    Piece capturedPiece;
    string capturedPieceName;
    string pieceName;
    int estimatedMoveReward = 0;
    Vector3Int moveToSquare;

    Dictionary<(Vector3Int, Piece, Vector3Int), int> findOrderDict = new Dictionary<(Vector3Int, Piece,
    Vector3Int), int>();
    int totalMoves = 0;

    for (int i = 0; i < moves.Count; i++)
    {
        for (int j = 0; j < moves.ElementAt(i).Count; j++)
        {
            moveToSquare = moves.ElementAt(i).ElementAt(j);
            pieceName = pieces.ElementAt(i).name;
            capturedPiece = board.GetPieceOnSquare(moveToSquare);
            capturedPieceName = (capturedPiece == null ? "" : capturedPiece.name);
            estimatedMoveReward = 0;
            if (capturedPieceName != "")
            {

```

```

        estimatedMoveReward += 10 * pieceNameToValueDict[capturedPieceName] -
pieceNameToValueDict[pieceName];

    }

    if ((pieceName == "Pawn" || pieceName == "Pawn(Clone") && moveToSquare.y == 7)

    {

        estimatedMoveReward += 9;

    }

    for (int k = 0; k < opposingAttacks.Count; k++)

    {

        for (int l = 0; l < opposingAttacks.ElementAt(k).Count; l++)

        {

            if (moveToSquare == opposingAttacks.ElementAt(k).ElementAt(l))

            {

                estimatedMoveReward -= pieceNameToValueDict[pieceName];

            }

        }

    }

    totalMoves++;

    findOrderDict.Add((moveToSquare, pieces.ElementAt(i), positions.ElementAt(i)),
estimatedMoveReward);

}

}

for (int i = 18; i > -10; i--)

{

    var matchingKeys = findOrderDict.Where(kvp => kvp.Value == i).Select(kvp => kvp.Key);

    foreach((Vector3Int, Piece, Vector3Int) key in matchingKeys)

    {

        List<Vector3Int> move = new List<Vector3Int>() { key.Item1 };

        orderedMoves.Add(move);

    }

}

```

```

        orderedPieces.Add(key.Item2);

        orderedPositions.Add(key.Item3);

    }

}

return (orderedMoves, orderedPieces, orderedPositions);

}

protected float[] SetupState(List<List<Vector3Int>> movesCurrentlyAvailable, List<Piece> pieceNames,
List<Vector3Int> pieceCoords)

{
    throw new NotImplementedException();
}

protected (Vector3Int coordsOfPieceToMove, Vector3Int coordsToMoveTo) DecodeInputState(object
indexOfMove)
{
    throw new NotImplementedException();
}

public int GetNumMovesMade()

{
    return movesMade;
}

public void IncreaseNumMovesMade()

{
    movesMade++;
}

}

```

NEURAL NETWORK

```
using System;
using System.IO;

[Serializable]

public class NeuralNetwork
{
    //When Comments Refer to the differented cost equations, these can be found in the analysis section.

    //The cost function used in conjunction with tanh is -0.5 * ( (1-y)*log(1-a) + (1+y)*log(1+a) ) + log(2) //
    (cross-entropy-esque)

    //this gives derives to functions that are proportional to the error

    //namely, dc/dw = x(a-y) and dc/db or dv = a-y

    public float[][] NodeValues; //[[layer in]][number of node]
    public float[][] NodeBiases; //[[layer in]][number of node]
    public float[][][] weights; // [layer in][node connected to][node connected from]

    public float[][] DesiredValues; // correct values for training
    public float[][] BiasNudges; // how much to nudge for cost
    public float[][][] weightNudges; // "" ""

    private const float ETA = 0.8f; // learning rate
    private const float LAMBDA = 0.001f; //l2 regularisation
    private const int MINI_BATCH = 100; // mini batch size for epoch based training
    private const float SCALE = 0.01f; // leaky reLU constant
    private int N;

    private char a_type;
    private char c_type;
```

```

private static Random rand = new Random();

public NeuralNetwork(int[] NNcomposure, char activation = 's', char cost = 'm')
{
    a_type = activation;
    c_type = cost;

    N = NNcomposure[0];

    // structure format - {num input nodes, num hidden layer 1 nodes, num hidden layer 2 node,..., num
    output nodes}

    NodeValues = new float[NNcomposure.Length][];
    NodeBiases = new float[NNcomposure.Length][];
    weights = new float[NNcomposure.Length - 1][][]; //no connection from output layer forwards

    DesiredValues = new float[NNcomposure.Length][];
    BiasNudges = new float[NNcomposure.Length][];
    weightNudges = new float[NNcomposure.Length - 1][][];// "" ""

    for (int i = 0; i < NNcomposure.Length; i++) //adding the respective number of nodes for each layer
    {
        NodeValues[i] = new float[NNcomposure[i]];
        NodeBiases[i] = new float[NNcomposure[i]];

        DesiredValues[i] = new float[NNcomposure[i]];
        BiasNudges[i] = new float[NNcomposure[i]];
    }

    for (int i = 0; i < NNcomposure.Length - 1; i++) //adding the respective number of weights needed per
layer

```

```

{
    weights[i] = new float[NodeValues[i + 1].Length][]; // nodes to
    weightNudges[i] = new float[NodeValues[i + 1].Length][]; //"" ""
    for (int j = 0; j < weights[i].Length; j++)
    {
        weights[i][j] = new float[DesiredValues[i].Length]; //nodes from
        weights[i][j] = new float[DesiredValues[i].Length];//"" ""
        for (int k = 0; k < weights[i][j].Length; k++)
        {
            // set every weight in the NN to a random value between 0 and 1
            // then multiply by the square root of two over the number of nodes in the layer
            // this distribution improves the initial learning of the NN
            weights[i][j][k] = (float)(rand.NextDouble()) * MathF.Sqrt(2f / weights[i][j].Length);
        }
    }
}

public float[] runNetwork(float[] inputs)
{
    for (int i = 0; i < NodeValues[0].Length; i++) // setting values
    {
        NodeValues[0][i] = inputs[i];
    }

    for (int i = 0; i < NodeValues.Length; i++)
    {
        for (int j = 0; j < NodeValues[i].Length; j++)
        {

```

```

//calculating activations

    NodeValues[i][j] = activation(SumForNode(NodeValues[i - 1], weights[i - 1][j]) + NodeBiases[i][j]);
    // sum of all weighted nodes before + the bias for the current node

    DesiredValues[i][j] = NodeValues[i][j]; // setup nodes before training

}

}

return NodeValues[NodeValues.Length - 1];

}

public float[] runNetworkSigmoidSpecific(float[] inputs)

{
    for (int i = 0; i < NodeValues[0].Length; i++) // setting values

    {
        NodeValues[0][i] = inputs[i];
    }

    for (int i = 0; i < NodeValues.Length; i++)
    {
        for (int j = 0; j < NodeValues[i].Length; j++)
        {
            //calculating activations

            NodeValues[i][j] = Sigmoid(SumForNode(NodeValues[i - 1], weights[i - 1][j]) + NodeBiases[i][j]);
            // sum of all weighted nodes before + the bias for the current node

            DesiredValues[i][j] = NodeValues[i][j]; // setup nodes before training

        }
    }

    return NodeValues[NodeValues.Length - 1];
}

public float[] runNetworkTanhSoftmax(float[] inputs)

```

```

{
    for (int i = 0; i < NodeValues[0].Length; i++) // setting values
    {
        NodeValues[0][i] = inputs[i];
    }

    for (int i = 0; i < NodeValues.Length-1; i++)
    {
        for (int j = 0; j < NodeValues[i].Length-1; j++)
        {
            //calculating activations
            NodeValues[i][j] = Tanh(SumForNode(NodeValues[i - 1], weights[i - 1][j]) + NodeBiases[i][j]); // sum of all weighted nodes before + the bias for the current node
            DesiredValues[i][j] = NodeValues[i][j]; // setup nodes before training
        }
    }
    int finalLayer = NodeValues.Length-1;
    for (int i = 0; i < NodeValues[finalLayer].Length; i++)
    {
        NodeValues[finalLayer][i] = Softmax(NodeValues[finalLayer], SumForNode(NodeValues[finalLayer], weights[finalLayer - 1][i]) + NodeBiases[finalLayer][i]); // sum of all weighted nodes before + the bias for the current node
        DesiredValues[finalLayer][i] = NodeValues[finalLayer][i]; // setup nodes before training
    }
    return NodeValues[NodeValues.Length - 1];
}

public float[] runNetworkTanh(float[] inputs)
{
    for (int i = 0; i < NodeValues[0].Length; i++) // setting values
    {

```

```

        NodeValues[0][i] = inputs[i];

    }

    for (int i = 0; i < NodeValues.Length; i++)
    {
        for (int j = 0; j < NodeValues[i].Length; j++)
        {
            //calculating activations

            NodeValues[i][j] = Tanh(SumForNode(NodeValues[i - 1], weights[i - 1][j]) + NodeBiases[i][j]); // sum of all weighted nodes before + the bias for the current node

            DesiredValues[i][j] = NodeValues[i][j]; // setup nodes before training
        }
    }

    return NodeValues[NodeValues.Length - 1];
}

public void TrainWithBackProp(float[][] Tinputs, float[][] Toutputs)
{
    int epoch = 0;

    for (int i = 0; i < Tinputs.Length; i++)
    {
        epoch++;

        runNetwork(Tinputs[i]); // test the network for every set of trianing data given

        for (int j = 0; j < DesiredValues[DesiredValues.Length - 1].Length; j++)
        {
            DesiredValues[DesiredValues.Length - 1][j] = Toutputs[i][j]; // adding the wanted outputs to the desired node values
        }
    }
}

```

```

for (int j = NodeValues.Length - 1; j > 0; j--) // back prop up to but excluding the input layer
{
    for (int k = 0; k < NodeValues[j].Length; k++)
    {
        var biasNudge = derivativeB(j, k);

        //// chain rule differentiation for dc/db first - easiest to start of this way as the weights and value
        differentiations include/are the bias differentials in some cases

        BiasNudges[j][k] += biasNudge;

        for (int l = 0; l < NodeValues[j - 1][l]; l++)
        {
            var weightNudge = derivativeW(j, l, biasNudge);
            weightNudges[j - 1][k][l] += weightNudge;

            var valueNudge = derivativeV(j, k, l, biasNudge); // again shown by diff - need to have wanted
            value for node behind to continue back prop

            DesiredValues[j - 1][l] += valueNudge; // needed for calculating in previous layers
        }
    }
}

if (epoch % MINI_BATCH == 0)
{
    for (int p = NodeValues.Length - 1; p > 0; p--) // for every layer bar the inputs
    {
        for (int j = 0; j < NodeValues[i].Length; j++) // for every node
        {
            NodeBiases[p][j] -= BiasNudges[p][j] * ETA / MINI_BATCH; // adjusting the biases
            BiasNudges[p][j] = 0; // resetting for more training

            DesiredValues[p][j] = 0;
        }
    }
}

```

```

        for (int k = 0; k < NodeValues[p - 1].Length; k++)
    {
        weights[p - 1][j][k] *= (1 - ETA) * LAMBDA / N; //adjust weights with accordance to l2
        regularisation

        weights[p - 1][j][k] -= weightNudges[p - 1][j][k] * ETA / MINI_BATCH; //continuation of
        weight adjustement equation

        weightNudges[p - 1][j][k] = 0; //reset

    }
}

}
}
}
}

```

```

public void TrainWithBackPropAndCrossEntropyWithL2Regularisation(float[][] Tinputs, float[][] Toutputs)
{
    int epoch = 0;

    for (int i = 0; i < Tinputs.Length; i++)
    {
        epoch++;

        runNetwork(Tinputs[i]); // test the network for every set of trianing data given

        for (int j = 0; j < DesiredValues[DesiredValues.Length - 1].Length; j++)
        {
            DesiredValues[DesiredValues.Length - 1][j] = Toutputs[i][j]; // adding the wanted outputs to the
            desired node values
        }

        for (int j = NodeValues.Length - 1; j > 0; j--) // back prop up to but excluding the input layer
        {
            for (int k = 0; k < NodeValues[j].Length; k++)

```

```

{

    var biasNudge = DesiredValues[j][k] - NodeValues[j][k];

    // chain rule diff for dc/db as dc/db = error delta = sigmod prime ZI * dc/da where dc/da = al -y

    BiasNudges[j][k] += biasNudge;

    for (int l = 0; l < NodeValues[j - 1][l]; l++)

    {

        var weightNudge = (NodeValues[j-1][l] * biasNudge)/N; // since the weight differential has a
        terms that equal the bias in it the bias can be used here

        weightNudges[j - 1][k][l] += weightNudge;

        var valueNudge = biasNudge; // again shown by diff (they are the same) - need to have wanted
        value for node behind to continue back prop

        DesiredValues[j - 1][l] += valueNudge; // needed for calculating in previous layers

    }

}

}

if (epoch % MINI_BATCH == 0)

{

    for (int p = NodeValues.Length - 1; p > 0; p--) // for every layer bar the inputs

    {

        for (int j = 0; j < NodeValues[i].Length; j++) // for every node

        {

            NodeBiases[p][j] -= BiasNudges[p][j] * ETA / MINI_BATCH; // adjusting the biases

            BiasNudges[p][j] = 0; // resetting for more training

            DesiredValues[p][j] = 0;

            for (int k = 0; k < NodeValues[p - 1].Length; k++)

            {
}

```

```

        weights[p - 1][j][k] *= (1 - ETA) * LAMBDA / N; //adjust weights with accordance to l2
regularisation

        weights[p - 1][j][k] -= weightNudges[p - 1][j][k] * ETA / MINI_BATCH; //continuation of
weight adjustement equation

        weightNudges[p - 1][j][k] = 0; //reset

    }

}

}

}

}

}

```

```

public void TrainWithBackPropAndTanhSoftmaxWithCustomCostAndL2Regularisation(float[][] Tinputs,
float[][] Toutputs)

{
    int epoch = 0;

    for (int i = 0; i < Tinputs.Length; i++)
    {
        epoch++;

        runNetworkTanhSoftmax(Tinputs[i]); // test the network for every set of trianing data given

        for (int j = 0; j < DesiredValues[DesiredValues.Length - 1].Length; j++)
        {
            DesiredValues[DesiredValues.Length - 1][j] = Toutputs[i][j]; // adding the wanted outputs to the
desired node values
        }
    }

    for (int j = NodeValues.Length - 1; j > 0; j--) // back prop up to but excluding the input layer
    {
        for (int k = 0; k < NodeValues[j].Length; k++)
        {
            var biasNudge = DesiredValues[j][k] - NodeValues[j][k];

```

```

// chain rule diff for dc/db as dc/db = error delta = sigmod prime ZI * dc/da where dc/da = al - y

BiasNudges[j][k] += biasNudge;

for (int l = 0; l < NodeValues[j - 1][l]; l++)

{

    var weightNudge = (NodeValues[j - 1][l] * biasNudge); // since the weight differential has a
terms that equal the bias in it the bias can be used here

    weightNudges[j - 1][k][l] += weightNudge;

    var valueNudge = biasNudge; // again shown by diff (they are the same) - need to have wanted
value for node behind to continue back prop

    DesiredValues[j - 1][l] += valueNudge; // needed for calculating in previous layers

}

}

}

if (epoch % MINI_BATCH == 0)

{

    for (int p = NodeValues.Length - 1; p > 0; p--) // for every layer bar the inputs

    {

        for (int j = 0; j < NodeValues[i].Length; j++) // for every node

        {

            NodeBiases[p][j] -= BiasNudges[p][j] * ETA / MINI_BATCH; // adjusting the biases

            BiasNudges[p][j] = 0; // resetting for more training

            DesiredValues[p][j] = 0;

            for (int k = 0; k < NodeValues[p - 1].Length; k++)

            {

                weights[p - 1][j][k] *= (1 - ETA) * LAMBDA / N; //adjust weights with accordance to l2
regularisation

                weights[p - 1][j][k] -= weightNudges[p - 1][j][k] * ETA / MINI_BATCH; //continuation of
weight adjustement equation

```

```

        weightNudges[p - 1][j][k] = 0; //reset
    }
}
}
}
}

public void TrainWithBackPropAndTanhWithCustomCostAndL2Regularisation(float[][] Tinputs, float[][] Toutputs)
{
    int epoch = 0;
    for (int i = 0; i < Tinputs.Length; i++)
    {
        epoch++;
        runNetworkTanh(Tinputs[i]); // test the network for every set of trianing data given

        for (int j = 0; j < DesiredValues[DesiredValues.Length - 1].Length; j++)
        {
            DesiredValues[DesiredValues.Length - 1][j] = Toutputs[i][j]; // adding the wanted outputs to the
            desired node values
        }

        for (int j = NodeValues.Length - 1; j > 0; j--) // back prop up to but excluding the input layer
        {
            for (int k = 0; k < NodeValues[j].Length; k++)
            {
                var biasNudge = DesiredValues[j][k] - NodeValues[j][k];
                // chain rule diff for dc/db as dc/db = error delta = sigmod prime ZI * dc/da where dc/da = al -y
                BiasNudges[j][k] += biasNudge;
                for (int l = 0; l < NodeValues[j - 1][l]; l++)

```

```

    {

        var weightNudge = (NodeValues[j - 1][l] * biasNudge); // since the weight differential has a
        terms that equal the bias in it the bias can be used here

        weightNudges[j - 1][k][l] += weightNudge;

        var valueNudge = biasNudge; // again shown by diff (they are the same) - need to have wanted
        value for node behind to continue back prop

        DesiredValues[j - 1][l] += valueNudge; // needed for calculating in previous layers

    }

}

}

if (epoch % MINI_BATCH == 0)

{

    for (int p = NodeValues.Length - 1; p > 0; p--) // for every layer bar the inputs

    {

        for (int j = 0; j < NodeValues[i].Length; j++) // for every node

        {

            NodeBiases[p][j] -= BiasNudges[p][j] * ETA / MINI_BATCH; // adjusting the biases

            BiasNudges[p][j] = 0; // resetting for more training

            DesiredValues[p][j] = 0;

            for (int k = 0; k < NodeValues[p - 1].Length; k++)

            {

                weights[p - 1][j][k] *= (1 - ETA) * LAMBDA / N; //adjust weights with accordance to l2
                regularisation

                weights[p - 1][j][k] -= weightNudges[p - 1][j][k] * ETA / MINI_BATCH; //continuation of
                weight adjustement equation

                weightNudges[p - 1][j][k] = 0; //reset

            }

        }

    }

}

```

```
    }

}

}

}

private static float SumForNode(float[] nodeValues, float[] weights)

{
    float finalSum = 0;

    for (int i = 0; i < nodeValues.Length; i++) // for each node in the layer

    {
        finalSum += nodeValues[i] * weights[i]; // multiply weight for the working on node by the last node
activation
    }

    return finalSum;
}
```

A decorative horizontal border consisting of a repeating pattern of diagonal hatching.

```
private float activation(float z, float[] zs = null) //nice way to chose between activation fucntion for small  
incomplex data
```

```
{  
    if (a_type == 's')  
    {  
        return Sigmoid(z);  
    }  
    else if (a_type == 'm')  
    {  
        return Softmax(zs, z);  
    }  
    else if (a_type == 'r')  
}
```

```

{
    return reLU(z);
}
else if (a_type == 'l')
{
    return LeakyReLU(z);
}
else
{
    return 0f;
}
}

```

```

private float derivativeB(int j, int k) //nice way to chose between cost function for small incomplex data
{
    if (c_type == 'q')
    {
        return MSL_SIG_B(j, k);
    }
    else if (c_type == 'c')
    {
        return CE_SIG_B(j, k);
    }
    else if (c_type == 'l')
    {
        return LL_SM_B(j, k);
    }
    else
    {
        return 0f;
    }
}

```

```

    }

}

private float derivativeW(int j, int l, float bias, int k = 0) //nice way to chose between cost fucntion for small
incomplex data

{
    if (c_type == 'q')
    {
        return MSL_SIG_W(j, k, l, bias);
    }
    else if (c_type == 'c')
    {
        return CE_SIG_W(j, k, bias);
    }
    else if (c_type == 'l')
    {
        return LL_SM_W(j, k, bias);
    }
    else
    {
        return 0f;
    }
}

private float derivativeV(int j, int k, int l, float bias) //nice way to chose between cost fucntion for small
incomplex data

{
    if (c_type == 'q')
    {
        return MSL_SIG_V(j, k, l, bias);
    }
    else if (c_type == 'c')

```

```

    {
        return CE_SIG_V(bias);
    }
    else if (c_type == 'l')
    {
        return LL_SM_V(j, k);
    }
    else
    {
        return 0f;
    }
}

```

```

private static float Sigmoid(float input)
{
    return 1f / (1f + (float)(Math.Exp(-input))); // sigmoid squishification function to get value 0-1
}

private static float Tanh(float input)
{
    return 2f / (1f + (float)(Math.Exp(-2 * input))); // tanh gives fast learning than sigmoid and works well with
softmax
}

private static float derivativeTanh(float input)
{
    return 2f / (1f + (float)(Math.Exp(-2 * input))); // tanh gives fast learning than sigmoid and works well with
softmax
}
```

```
}
```

```
private static float DerivativeSigmoid(float input) // this is needed for the calculus involved in back prop
```

```
{
```

```
    return input * (1 - input); // technically this should be sig(x) * (1-sig(x)) but dealt with above
```

```
}
```

```
private static float reLU(float input) // perhaps try using this - reduces vanishing gradient - perhaps also try  
leaky relu
```

```
{
```

```
    if (input < 0)
```

```
{
```

```
        return 0;
```

```
}
```

```
    return input;
```

```
}
```

```
private static float LeakyReLU(float input) //stop problems with dying weights due to small gradient with  
negatives
```

```
{
```

```
    if (input < 0)
```

```
{
```

```
        return input / SCALE;
```

```
}
```

```
    else
```

```
{
```

```
        return input;
```

```
}
```

```
}
```

```
private static float HardSigmoid(float input) // use for large data sets or lots of iterations
```

```

{
    if (input < -2.5f)
    {
        return 0f;
    }
    if (input > 2.5f)
    {
        return 1f;
    }
    return 0.2f * input + 0.5f;
}

private static float Softmax(float[] layerinput, float input) //for classification could be used for determining best moves from a small predefined set
{
    float sum = 0;
    foreach (float num in layerinput)
    {
        sum += (float)(Math.Exp((num)));
    }
    return (float)(Math.Exp(input) / sum);
}

private float MSL_SIG_W(int j, int k, int l, float biasNudge) //mean square loss with sigmoid weight change
{
    return NodeValues[j - 1][l] * biasNudge;
}

private float MSL_SIG_B(int j, int k) //mean square loss with sigmoid bias change
{
    return DerivativeSigmoid(NodeValues[j][k]) * (DesiredValues[j][k] - NodeValues[j][k]);
}

```

```

}

private float MSL_SIG_V(int j, int k, int l, float biasNudge) //mean square loss with sigmoid values change
{
    return weights[j - 1][k][l] * biasNudge;
}

private float CE_SIG_W(int j, int k, float biasNudge) //cross entropy with sigmoid weight change
{
    float sum = 0;
    foreach (float item in NodeValues[j - 1])
    {
        sum += item * biasNudge;
    }
    return sum / N;
}

private float CE_SIG_B(int j, int k) //cross entropy with sigmoid bias change
{
    return (DesiredValues[j][k] - NodeValues[j][k]);
}

private float CE_SIG_V(float biasNudge) //cross entropy with sigmoid values change
{
    return biasNudge;
}

private float LL_SM_W(int j, int k, float biasNudge) //Logarithmic loss with softmax weight change - classification uses only
{
    float sum = 0;

```

```

foreach (float item in NodeValues[j - 1])
{
    sum += item * biasNudge;
}
return sum / N;
}

private float LL_SM_B(int j, int k) //Logarithmic loss with softmax bias change - classification uses only
{
    return (DesiredValues[j][k] - NodeValues[j][k]);
}

private float LL_SM_V(int j, int k) //Logarithmic loss with softmax value change - classification uses only
{
    return (DesiredValues[j][k] - NodeValues[j][k]);
}

////////////////////////////////////////////////////////////////////////

public void SaveNNCurrentState()
{
    string filePath = "Assets/NNState/W.txt";
    string allNums = "";

    foreach(float[][] layer in weights)
    {
        allNums += "l";
        foreach(float[] from in layer)
        {
            allNums += "f";
        }
    }
}

```

```

foreach(float weight in from)
{
    allNums += "w";
    allNums += weight.ToString();
}
}

File.WriteAllText(filePath, allNums);

filePath = "Assets/NNState/B.txt";
allNums = "";

foreach(float[] layer in NodeBiases)
{
    allNums += "l";
    foreach (float bias in layer)
    {
        allNums += "b";
        allNums += bias.ToString();
    }
}

File.WriteAllText(filePath, allNums);
}

public void LoadNNCurrentState()
{
    string filePath = "Assets/NNState/W.txt";
    string allData = File.ReadAllText(filePath);
    int layer = -1;
    int from = -1;
}

```

```

int to = -1;
string weight = "";
for (int i = 0; i < allData.Length; i++)
{
    if(allData.Substring(i,1) == "l")
    {
        layer++;
    }
    else if (allData.Substring(i, 1) == "f")
    {
        from++;
    }
    else if (allData.Substring(i, 1) == "w")
    {
        to++;
    }
    else
    {
        weight += allData.Substring(i, 1);
    }
    weights[layer][from][to] = (float)(Convert.ToDouble(weight));
}

filePath = "Assets/NNState/B.txt";
allData = File.ReadAllText(filePath);
layer = -1;
to = -1;
string bias = "";
for (int i = 0; i < allData.Length; i++)
{

```

```

        if (allData.Substring(i, l) == "l")
    {
        layer++;
    }

    else if (allData.Substring(i, l) == "b")
    {
        to++;
    }

    else
    {
        weight += allData.Substring(i, l);
    }
}

NodeBiases[layer][to] = (float)(Convert.ToDouble(bias));
}
}
}

```

NN MANAGER

```

using System;
using System.IO;
using System.Linq;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public enum activationType
{
    tanh, sigmoid, relu, leakyReLU, hardSigmoid
}

public enum costType
{
    tanhCustom, crossEntropy, MSE
}

public class NNManager
{
    public NeuralNetwork mainStateNetwork = new NeuralNetwork(new int[5]
{7168, 3584, 512, 64, 1}); // one input per piece per square per team, hidden layer in
// hopes it identifies black team negativity, hidden layer size of board, hidden layer 5x
// smaller, output

    public activationType activation { get; set; }
    public costType cost { get; set; }
}

```

```

public NNManager(int[] NNcomposure, activationType activation, costType cost)
{
    mainStateNetwork = new NeuralNetwork(NNcomposure);
    this.activation = activation;
    this.cost = cost;
    if (!ActivationAndCostAreCompatible())
    {
        throw new NeuralNetworkInstantiationException("The cost and activation
functions were invalid");
    }
}

private bool ActivationAndCostAreCompatible()
{
    if (cost == costType.crossEntropy)
    {
        if (activation == activationType.sigmoid || activation ==
activationType.hardSigmoid)
        {
            return true;
        }
    }
    else if(cost == costType.tanhCustom)
    {
        if(activation == activationType.tanh)
        {
            mainStateNetwork.LoadNNCurrentState();
            return true;
        }
    }
    else
    {
        return true;
    }
    return false;
}

public float[] FeedForward(float[] inputs)
{
    return mainStateNetwork.runNetworkTanh(inputs); // as i know i will jsut be
using this combination of activation and cost easier to implement than many if
statements
}

private void TrainingNNWithData()
{
    float[][][] tData = GetTData();
    float[][] TinputsWhole = tData[0];
    float[][] ToutputsWhole = tData[1];
    float[][][] Tinputs = new float[100][];
    float[][][] Toutputs = new float[100][];
    mainStateNetwork.LoadNNCurrentState();
    for (int i = 0; i < TinputsWhole.Length/100; i+=100)
    {
        for (int j = 0; j < 100; j++)
        {
            Tinputs[j] = TinputsWhole[i + j];
            Toutputs[j] = ToutputsWhole[i + j];
        }
    }
}

mainStateNetwork.TrainWithBackPropAndTanhWithCustomCostAndL2Regularisation(Tinputs,
Toutputs);

```

```

        mainStateNetwork.SaveNNCurrentState();
    }

private float[][][][] GetTData()
{
    string filePath = "Assets/Training Data/T.txt";
    string allData = File.ReadAllText(filePath);
    float[][][] tData = new float[2][][];
    float[] data = new float[7168];
    float[] output = new float[100000];
    int current = 0;
    for (int i = 0; i < allData.Length / 7169; i += 7169) //7 pieces per square
per team + one output
    {
        for (int j = 0; j < 7168; j++)
        {
            data[j] = Convert.ToInt32(allData[i + j]);
        }
        output[0] = Convert.ToInt32(allData[i + 7169]);
        tData[0][current] = data;
        tData[1][current] = output;
    }
    return tData;
}
}

```

CHESS GAME

PIECES (FOLDER FOR PIECE SCRIPTS – NOT PIECE CLASS)

PAWN

```

using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Pawn : Piece
{
    public override List<Vector3Int> SelectAvailableSquares()
    {
        availableMoves.Clear();

        Vector3Int direction = team == TeamColor.White ? new Vector3Int(0,1,0) : new
Vector3Int (0,-1,0);
        float range = hasMoved ? 1 : 2;
        for (int i = 1; i <= range; i++)
        {
            Vector3Int nextCoords = occupiedSquare + direction * i;
            Piece piece = board.GetPieceOnSquare(nextCoords);
            if (!board.CheckIfCoordinatesAreOnBoard(nextCoords))
                break;
            if (piece == null)
                TryToAddMove(nextCoords);
            else
                break;
        }
    }
}

```

```

        }

        Vector3Int[] takeDirectionsWhite = new Vector3Int[]
        {
            new Vector3Int (1,1,1),
            new Vector3Int (-1,1,1),
            new Vector3Int (1,1,-1),
            new Vector3Int (-1,1,-1)
        };

        Vector3Int[] takeDirectionsBlack = new Vector3Int[]
        {
            new Vector3Int (1,-1,1),
            new Vector3Int (-1,-1,1),
            new Vector3Int (1,-1,-1),
            new Vector3Int (-1,-1,-1)
        };

        for (int i = 0; i < 4; i++)
        {
            Vector3Int nextCoords = team == TeamColor.White? (occupiedSquare +
takeDirectionsWhite[i]) : (occupiedSquare + takeDirectionsBlack[i]);
            Piece piece = board.GetPieceOnSquare(nextCoords);
            if (!board.CheckIfCoordinatesAreOnBoard(nextCoords))
                continue;
            if (piece != null && !piece.IsFromSameTeam(this))
            {
                TryToAddMove(nextCoords);
            }
        }
        return availableMoves;
    }

    public override void MovePiece(Vector3Int coords,Piece piece = null, bool isPawn =
false)
    {
        base.MovePiece(coords,null,true) ;
        CheckPromotion();
    }

    private void CheckPromotion()
    {
        int endOfBoardYCoord = team == TeamColor.White ? Board.BOARD_SIZE - 1 : 0;
        if (occupiedSquare.y == endOfBoardYCoord)
        {
            board.PromotePiece(this);
        }
    }
}

```

KNIGHT

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Knight : Piece
{
    Vector3Int[] offsets = new Vector3Int[]
    {

```

```

        new Vector3Int(1,1,2),
        new Vector3Int(1,1,-2),
        new Vector3Int(1,-1,2),
        new Vector3Int(-1,1,2),
        new Vector3Int(-1,-1,2),
        new Vector3Int(-1,1,-2),
        new Vector3Int(1,-1,-2),
        new Vector3Int(-1,-1,-2),

        new Vector3Int(1,2,1),
        new Vector3Int(1,2,-1),
        new Vector3Int(1,-2,1),
        new Vector3Int(-1,2,1),
        new Vector3Int(-1,-2,1),
        new Vector3Int(-1,2,-1),
        new Vector3Int(1,-2,-1),
        new Vector3Int(-1,-2,-1),

        new Vector3Int(2,1,1),
        new Vector3Int(2,1,-1),
        new Vector3Int(2,-1,1),
        new Vector3Int(-2,1,1),
        new Vector3Int(-2,-1,1),
        new Vector3Int(-2,1,-1),
        new Vector3Int(2,-1,-1),
        new Vector3Int(-2,-1,-1),
    };

    public override List<Vector3Int> SelectAvailableSquares()
    {
        availableMoves.Clear();

        for (int i = 0; i < offsets.Length; i++)
        {
            Vector3Int nextCoords = occupiedSquare + offsets[i];
            Piece piece = board.GetPieceOnSquare(nextCoords);
            if (!board.CheckIfCoordinatesAreOnBoard(nextCoords))
                continue;
            if (piece == null || !piece.IsFromSameTeam(this))
                TryToAddMove(nextCoords);
        }
        return availableMoves;
    }
}

```

BISHOP

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Bishop : Piece
{
    private Vector3Int[] directions = new Vector3Int[]
    {
        new Vector3Int(1, 1,0),
        new Vector3Int(1, -1,0),
        new Vector3Int(-1, 1,0),

```

```

        new Vector3Int(-1,- 1,0),
        new Vector3Int(-1,- 1,-1),
        new Vector3Int(-1,- 1, 1),
        new Vector3Int(-1,1,-1),
        new Vector3Int(-1,1, 1),
        new Vector3Int(1,- 1,-1),
        new Vector3Int(1,- 1, 1),
        new Vector3Int(1,1,-1),
        new Vector3Int(1,1, 1),
    };
    public override List<Vector3Int> SelectAvailableSquares()
    {
        availableMoves.Clear();
        float range = Board.BOARD_SIZE;
        foreach (var direction in directions)
        {
            for (int i = 1; i <= range; i++)
            {
                Vector3Int nextCoords = occupiedSquare + direction * i;
                Piece piece = board.GetPieceOnSquare(nextCoords);
                if (!board.CheckIfCoordinatesAreOnBoard(nextCoords))
                    break;
                if (piece == null)
                    TryToAddMove(nextCoords);
                else if (!piece.IsFromSameTeam(this))
                {
                    TryToAddMove(nextCoords);
                    break;
                }
                else if (piece.IsFromSameTeam(this))
                    break;
            }
        }
        return availableMoves;
    }
}

```

ROOK

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Rook : Piece
{
    private Vector3Int[] directions = new Vector3Int[]
    {
        new Vector3Int(0,0,1),
        new Vector3Int(0,0,-1),
        new Vector3Int(0,1,0),
        new Vector3Int(0,-1,0),
        new Vector3Int(1,0,0),
        new Vector3Int(-1,0,0),
    };
    public override List<Vector3Int> SelectAvailableSquares()
    {
        availableMoves.Clear();

        float range = Board.BOARD_SIZE;
        foreach (var direction in directions)

```

```

    {
        for (int i = 1; i <= range; i++)
        {
            Vector3Int nextCoords = occupiedSquare + direction * i;
            Piece piece = board.GetPieceOnSquare(nextCoords);
            if (!board.CheckIfCoordinatesAreOnBoard(nextCoords))
                break;
            if (piece == null)
                TryToAddMove(nextCoords);
            else if (!piece.IsFromSameTeam(this))
            {
                TryToAddMove(nextCoords);
                break;
            }
            else if (piece.IsFromSameTeam(this))
                break;
        }
    }
    return availableMoves;
}
}

```

COMMONER

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Commoner : Piece
{
    private Vector3Int[] directions = new Vector3Int[]
    {
        new Vector3Int(0,0,-1),
        new Vector3Int(-1,-1,-1),
        new Vector3Int(-1,0,-1),
        new Vector3Int(-1,1,-1),
        new Vector3Int(0,-1,-1),
        new Vector3Int(0,1,-1),
        new Vector3Int(1,-1,-1),
        new Vector3Int(1,0,-1),
        new Vector3Int(1,1,-1),

        new Vector3Int(-1,-1,1),
        new Vector3Int(-1,0,1),
        new Vector3Int(-1,1,1),
        new Vector3Int(0,-1,1),
        new Vector3Int(0,1,1),
        new Vector3Int(1,-1,1),
        new Vector3Int(1,0,1),
        new Vector3Int(1,1,1),
        new Vector3Int(0,0,1),

        new Vector3Int(-1,-1,0),
        new Vector3Int(-1,0,0),
        new Vector3Int(-1,1,0),
        new Vector3Int(0,-1,0),
        new Vector3Int(0,1,0),
        new Vector3Int(1,-1,0),
    }
}

```

```

        new Vector3Int(1,0,0),
        new Vector3Int(1,1,0),
    };
    public override List<Vector3Int> SelectAvailableSquares()
    {
        availableMoves.Clear();

        float range = 1;
        foreach (var direction in directions)
        {
            for (int i = 1; i <= range; i++)
            {
                Vector3Int nextCoords = occupiedSquare + direction * i;
                Piece piece = board.GetPieceOnSquare(nextCoords);
                if (!board.CheckIfCoordinatesAreOnBoard(nextCoords))
                    break;
                if (piece == null)
                    TryToAddMove(nextCoords);
                else if (!piece.IsFromSameTeam(this))
                {
                    TryToAddMove(nextCoords);
                    break;
                }
                else if (piece.IsFromSameTeam(this))
                    break;
            }
        }
        return availableMoves;
    }
}

```

QUEEN

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Queen : Piece
{
    private Vector3Int[] directions = new Vector3Int[]
    {
        new Vector3Int(0,0,-1),
        new Vector3Int(-1,-1,-1),
        new Vector3Int(-1,0,-1),
        new Vector3Int(-1,1,-1),
        new Vector3Int(0,-1,-1),
        new Vector3Int(0,1,-1),
        new Vector3Int(1,-1,-1),
        new Vector3Int(1,0,-1),
        new Vector3Int(1,1,-1),

        new Vector3Int(-1,-1,1),
        new Vector3Int(-1,0,1),
        new Vector3Int(-1,1,1),
        new Vector3Int(0,-1,1),
        new Vector3Int(0,1,1),
        new Vector3Int(1,-1,1),
        new Vector3Int(1,0,1),

```

```

        new Vector3Int(1,1,1),
        new Vector3Int(0,0,1),

        new Vector3Int(-1,-1,0),
        new Vector3Int(-1,0,0),
        new Vector3Int(-1,1,0),
        new Vector3Int(0,-1,0),
        new Vector3Int(0,1,0),
        new Vector3Int(1,-1,0),
        new Vector3Int(1,0,0),
        new Vector3Int(1,1,0),
    };
    public override List<Vector3Int> SelectAvailableSquares()
    {
        availableMoves.Clear();

        float range = Board.BOARD_SIZE;
        foreach (var direction in directions)
        {
            for (int i = 1; i <= range; i++)
            {
                Vector3Int nextCoords = occupiedSquare + direction * i;
                Piece piece = board.GetPieceOnSquare(nextCoords);
                if (!board.CheckIfCoordinatesAreOnBoard(nextCoords))
                    break;
                if (piece == null)
                    TryToAddMove(nextCoords);
                else if (!piece.IsFromSameTeam(this))
                {
                    TryToAddMove(nextCoords);
                    break;
                }
                else if (piece.IsFromSameTeam(this))
                    break;
            }
        }
        return availableMoves;
    }
}

```

KING

```

using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class King : Piece
{
    Vector3Int[] directions = new Vector3Int[]
    {
        new Vector3Int(0,0,-1),
        new Vector3Int(-1,-1,-1),
        new Vector3Int(-1,0,-1),
        new Vector3Int(-1,1,-1),
        new Vector3Int(0,-1,-1),
        new Vector3Int(0,1,-1),
        new Vector3Int(1,-1,-1),
        new Vector3Int(1,0,-1),
        new Vector3Int(1,1,-1),
    };
}

```

```

        new Vector3Int(-1,-1,1),
        new Vector3Int(-1,0,1),
        new Vector3Int(-1,1,1),
        new Vector3Int(0,-1,1),
        new Vector3Int(0,1,1),
        new Vector3Int(1,-1,1),
        new Vector3Int(1,0,1),
        new Vector3Int(1,1,1),
        new Vector3Int(0,0,1),

        new Vector3Int(-1,-1,0),
        new Vector3Int(-1,0,0),
        new Vector3Int(-1,1,0),
        new Vector3Int(0,-1,0),
        new Vector3Int(0,1,0),
        new Vector3Int(1,-1,0),
        new Vector3Int(1,0,0),
        new Vector3Int(1,1,0),
    };

    private Piece leftRook;
    private Piece rightRook;

    private Vector3Int leftCastlingMove;
    private Vector3Int rightCastlingMove;

    public override List<Vector3Int> SelectAvailableSquares()
    {
        availableMoves.Clear();
        AssignStandardMoves();
        AssignCastlingMoves();
        return availableMoves;
    }

    private void AssignCastlingMoves()
    {
        leftCastlingMove = new Vector3Int(-1, -1, 0);
        rightCastlingMove = new Vector3Int(-1, -1, 0);
        if (!hasMoved)
        {
            leftRook = GetPieceInDirection<Rook>(team, new Vector3Int(-1, 0, 0));
            if (leftRook && !leftRook.hasMoved)
            {
                leftCastlingMove = occupiedSquare + new Vector3Int(-1, 0, 0) * 2;
                availableMoves.Add(leftCastlingMove);
            }
            rightRook = GetPieceInDirection<Rook>(team, new Vector3Int(1, 0, 0));
            if (rightRook && !rightRook.hasMoved)
            {
                rightCastlingMove = occupiedSquare + new Vector3Int(1, 0, 0) * 2;
                availableMoves.Add(rightCastlingMove);
            }
        }
    }

    private Piece GetPieceInDirection<T>(TeamColor team, Vector3Int direction)
    {
        for (int i = 1; i <= Board.BOARD_SIZE; i++)
        {

```

```

        Vector3Int nextCoords = occupiedSquare + direction * i;
        Piece piece = board.GetPieceOnSquare(nextCoords);
        if (!board.CheckIfCoordinatesAreOnBoard(nextCoords))
            return null;
        if (piece != null)
        {
            if (piece.team != team || !(piece is T))
                return null;
            else if (piece.team == team && piece is T)
                return piece;
        }
    }
    return null;
}

private void AssignStandardMoves()
{
    float range = 1;
    foreach (var direction in directions)
    {
        for (int i = 1; i <= range; i++)
        {
            Vector3Int nextCoords = occupiedSquare + direction * i;
            Piece piece = board.GetPieceOnSquare(nextCoords);
            if (!board.CheckIfCoordinatesAreOnBoard(nextCoords))
                break;
            if (piece == null)
                TryToAddMove(nextCoords);
            else if (!piece.IsFromSameTeam(this))
            {
                TryToAddMove(nextCoords);
                break;
            }
            else if (piece.IsFromSameTeam(this))
                break;
        }
    }
}

public override void MovePiece(Vector3Int coords, Piece piece = null, bool isPawn
= false)
{
    base.MovePiece(coords);
    if (coords == leftCastlingMove)
    {
        board.UpdateBoardOnPieceMove(coords + new Vector3Int(1,0,0),
leftRook.occupiedSquare, leftRook, null);
        leftRook.MovePiece(coords + new Vector3Int(1,0,0));
    }
    else if (coords == rightCastlingMove)
    {
        board.UpdateBoardOnPieceMove(coords + new Vector3Int(-1, 0, 0),
rightRook.occupiedSquare, rightRook, null);
        rightRook.MovePiece(coords + new Vector3Int(1, 0, 0));
    }
}
}

```

BOARD

using System;

```
using System.IO;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using UnityEditor;
using Unity;

[RequireComponent(typeof(SquareSelectorCreator))]

public class Board : MonoBehaviour
{
    public const int BOARD_SIZE = 8;

    [SerializeField] private Transform bottomLeftSquareTransform;

    [SerializeField] private float squareSize;
    [SerializeField] private float boardHeight;
    [SerializeField] private ChessUIManager chessUIManager;
    [SerializeField] private GameObject moveButton;
    [SerializeField] private Transform gridContent;

    public Piece[,] grid;

    private Piece selectedPiece;
    private ChessGameController chessController;
    private SquareSelectorCreator squareSelector;

    public List<(Vector3Int, Vector3Int)> pastMovesWhite = new List<(Vector3Int, Vector3Int)>();
    public List<(Vector3Int, Vector3Int)> pastMovesBlack = new List<(Vector3Int, Vector3Int)>();
```

```

private Dictionary<String, int> pieceNameToValueDict = new Dictionary<String, int>()
{
    {"Pawn", 1 },
    {"Bishop", 3 },
    {"Knight", 3 },
    {"Commoner", 3 },
    {"Rook", 5 },
    {"Queen", 9 },

    {"Pawn(Clone)", 1 },
    {"Bishop(Clone)", 3 },
    {"Knight(Clone)", 3 },
    {"Commoner(Clone)", 3 },
    {"Rook(Clone)", 5 },
    {"Queen(Clone)", 9 },
};


```

```
private List<GameObject> notationList = new List<GameObject>();
```

```

public int majorPiecesTaken;
public int majorPiecesMoved;
public int materialImbalance;
```

```

private void Awake()
{
    squareSelector = GetComponent<SquareSelectorCreator>();
    CreateGrid();
    moveButton.SetActive(false);
}
```

```

public void SetDependencies(ChessGameController chessController)
{
    this.chessController = chessController;
}

private void CreateGrid()
{
    grid = new Piece[BOARD_SIZE, BOARD_SIZE, BOARD_SIZE];
}

public Vector3 CalculatePositionFromCoords(Vector3Int coords)
{
    return bottomLeftSquareTransform.position + new Vector3(coords.x * squareSize,
    (coords.z*boardHeight) + 0.1f, coords.y * squareSize);
}

private Vector3Int CalculateCoordsFromPosition(Vector3 inputPosition)
{
    int x = Mathf.FloorToInt(transform.InverseTransformPoint(inputPosition).x / squareSize) + BOARD_SIZE
    / 2;

    int y = Mathf.FloorToInt(transform.InverseTransformPoint(inputPosition).z / squareSize) + BOARD_SIZE
    / 2;

    int z = (Mathf.FloorToInt(transform.InverseTransformPoint(inputPosition).y / boardHeight));

    return new Vector3Int(x, y,z);
}

public void OnSquareSelected(Vector3 inputPosition)
{
    Vector3Int coords = CalculateCoordsFromPosition(inputPosition);

    Piece piece = GetPieceOnSquare(coords);
}

```

```

if (selectedPiece)
{
    if (piece != null && selectedPiece == piece)
        DeselectPiece();

    else if (piece != null && selectedPiece != piece && chessController.IsTeamTurnActive(piece.team))
        SelectPiece(piece);

    else if (selectedPiece.CanMoveTo(coords))
        OnSelectedPieceMoved(coords, selectedPiece);
}

else
{
    if (piece != null && chessController.IsTeamTurnActive(piece.team))
        SelectPiece(piece);
}
}

public Vector3Int getPieceCoords(string name, TeamColor team)
{
    for (int i = 0; i < BOARD_SIZE; i++)
    {
        for (int j = 0; j < BOARD_SIZE; j++)
        {
            for (int k = 0; k < BOARD_SIZE; k++)
            {
                if (grid[i, j, k] != null)
                {
                    if (grid[i, j, k].name == name && grid[i, j, k].team == team)
                    {
                        return new Vector3Int(i, j, k);
                    }
                }
            }
        }
    }
}

```

```

        }

    }

}

return new Vector3Int(-1, -1, -1);
}

public void AIMakeMove(Vector3Int currentPosition, Vector3Int goToPosition)
{
    Piece piece = GetPieceOnSquare(currentPosition);
    SelectPiece(piece);
    OnSelectedPieceMoved(goToPosition, piece);
}

private void SelectPiece(Piece piece)
{
    chessController.RemoveMovesEnablingAttackOnPieceOfType<King>(piece);
    selectedPiece = piece;
    List<Vector3Int> selection = selectedPiece.availableMoves;
    ShowSelectionSquares(selection);
}

private void ShowSelectionSquares(List<Vector3Int> selection)
{
    Dictionary<Vector3, bool> squaresData = new Dictionary<Vector3, bool>();
    for (int i = 0; i < selection.Count; i++)

```

```

    {
        Vector3 position = CalculatePositionFromCoords(selection[i]);
        bool isSquareFree = GetPieceOnSquare(selection[i]) == null;
        squaresData.Add(position, isSquareFree);
    }
    squareSelector.ShowSelection(squaresData);
}

```

```

private void DeselectPiece()
{
    selectedPiece = null;
    squareSelector.ClearSelection();
}

private void OnSelectedPieceMoved(Vector3Int coords, Piece piece)
{
    ChessPlayer activePlayer = chessController.GetActivePlayer();
    if(activePlayer.team == TeamColor.White)
    {
        pastMovesWhite.Add((piece.occupiedSquare, coords));
    }
    else
    {
        pastMovesBlack.Add((piece.occupiedSquare, coords));
    }
    bool take = TryToTakeOppositePiece(coords);
    DisplayNotation(piece, piece.occupiedSquare, coords, take);
    UpdateBoardOnPieceMove(coords, piece.occupiedSquare, piece, null);
    if(piece.hasMoved == false)
    {

```

```

        if(!(piece.name == "Pawn" || piece.name == "Pawn(Clone)" || piece.name == "King" || piece.name == "King(Clone")))
    {
        majorPiecesMoved++;
    }
}

selectedPiece.MovePiece(coords, piece);
DeselectPiece();
EndTurn();
}

private void EndTurn()
{
    chessController.EndTurn();
}

public void UpdateBoardOnPieceMove(Vector3Int newCoords, Vector3Int oldCoords, Piece newPiece,
Piece oldPiece)
{
    grid[oldCoords.x, oldCoords.y, oldCoords.z] = oldPiece;
    grid[newCoords.x, newCoords.y, newCoords.z] = newPiece;
}

public Piece GetPieceOnSquare(Vector3Int coords)
{
    if (CheckIfCoordinatesAreOnBoard(coords))
        return grid[coords.x, coords.y, coords.z];
    return null;
}

public bool CheckIfCoordinatesAreOnBoard(Vector3Int coords)

```

```

{
    if (coords.x < 0 || coords.y < 0 || coords.z < 0 || coords.x >= BOARD_SIZE || coords.y >= BOARD_SIZE || coords.z >= BOARD_SIZE)
        return false;
    return true;
}

public bool HasPiece(Piece piece)
{
    for (int i = 0; i < BOARD_SIZE; i++)
    {
        for (int j = 0; j < BOARD_SIZE; j++)
        {
            for (int k = 0; k < BOARD_SIZE; k++)
            {
                if (grid[i, j, k] == piece)
                    return true;
            }
        }
    }
    return false;
}

public void SetPieceOnBoard(Vector3Int coords, Piece piece)
{
    if (CheckIfCoordinatesAreOnBoard(coords))
        grid[coords.x, coords.y, coords.z] = piece;
}

private bool TryToTakeOppositePiece(Vector3Int coords)

```

```

{
    Piece piece = GetPieceOnSquare(coords);

    if (piece != null && !selectedPiece.IsFromSameTeam(piece))

    {
        chessUIManager.IncreaseTakenPiece(piece);

        if (!(piece.name == "Pawn" || piece.name == "Pawn(Clone)" || piece.name == "King" || piece.name == "King(Clone")))
        {

            majorPiecesTaken++;

        }

        materialImbalance += piece.team == TeamColor.White ? pieceNameToValueDict[piece.name] : -1 * pieceNameToValueDict[piece.name];

        TakePiece(piece);

        return true;
    }

    return false;
}

private void TakePiece(Piece piece)

{
    if (piece)
    {

        grid[piece.occupiedSquare.x, piece.occupiedSquare.y, piece.occupiedSquare.z] = null;

        chessController.OnPieceRemoved(piece);

        Destroy(piece.gameObject);
    }
}

public int GetBoardSize()

{
    return BOARD_SIZE;
}

```

```

    }

public void PromotePiece(Piece piece)
{
    TakePiece(piece);
    chessController.CreatePieceAndInitialize(piece.occupiedSquare, piece.team, typeof(Queen));
}

internal void OnGameRestarted()
{
    selectedPiece = null;
    CreateGrid();
}

public void DisplayNotation(Piece piece, Vector3Int atCoords, Vector3Int toCoords, bool take, bool castle
= false, bool check=false)
{
    string checkS = "";
    string pieceChar = piece.name[0].ToString();
    if (piece.name == "Knight" || piece.name == "Knight(Clone)")
    {
        pieceChar = "N";
    }
    string start = ((atCoords.z+1).ToString() + (char)(65 + (atCoords.x)) + (atCoords.y+1).ToString());
    string finish = ((toCoords.z + 1).ToString() + (char)(65 + (toCoords.x)) + (toCoords.y + 1).ToString());
    string combiation = take ? "x" : "|";
    if (check) checkS = "+";
    string notation = pieceChar + start + combiation + finish + checkS;
    if (castle) notation = (atCoords.z.ToString() + "O-O" + toCoords.z.ToString() + checkS);
    InstantiateDisplayNotation(notation);
}

```

```

    }

public void InstantiateDisplayNotation(string notation)
{
    GameObject newMove = Instantiate(moveButton);
    newMove.GetComponentInChildren<Text>().text = notation;
    newMove.SetActive(true);
    newMove.transform.SetParent(gridContent);
    notationList.Add(newMove);
}

public void DeleteDisplayNotation()
{
    if(notationList.Count != 0)
    {
        foreach (GameObject item in notationList)
        {
            item.SetActive(false);
        }
    }
}
}

```

CHESS GAME CONTROLLER

```

using System;
using System.IO;
using System.Collections;
using System.Collections.Generic;
using System.Linq;

```

```
using UnityEngine;
using System.Threading.Tasks;

public enum PlayerType
{
    Person, AI
}

[RequireComponent(typeof(PiecesCreator))]
[RequireComponent(typeof(AIManager))]

public class ChessGameController : MonoBehaviour
{
    private enum GameState
    {
        Init, Play, Finished
    }

    [SerializeField] private BoardLayout startingBoardLayout;
    [SerializeField] private Board board;
    [SerializeField] private ChessUIManager UIManager;
    [SerializeField] private Camera mainCamera;
    [SerializeField] private AudioSource moveSound;

    [SerializeField] public TimerHelper whiteClock;
    [SerializeField] public TimerHelper blackClock;

    private PiecesCreator pieceCreator;
    private ChessPlayer whitePlayer;
    private ChessPlayer blackPlayer;
```

```

public ChessPlayer activePlayer;
private AIManager ai;

private GameState state;

public bool Alactive;
public int AINum = 0;
public int aiDifficulty;

private void Awake()
{
    SetDependencies();
}

private void SetDependencies()
{
    pieceCreator = GetComponent<PiecesCreator>();
}

public AIManager CreateAIOfDifficulty(ChessPlayer a, ChessPlayer o, Board board,int difficulty)
{
    if (difficulty == 1)
    {
        return new AI1(a, o, board);
    }
    else if (difficulty == 2)
    {
        return new AI2(a, o, board);
    }
}

```

```

else if (difficulty == 3)
{
    return new AI3(a, o, board);
}

else if (difficulty == 4)
{
    return new AI3(a, o, board);
}

else if (difficulty == 5)
{
    return new AI3(a, o, board);
}

else
{
    return new AI1(a, o, board);
}
}

private void CreatePlayers()
{
    UIManager.DifficultySelection();
    Debug.Log(UIManager.GiveDifficulty());
    if (UIManager.playerPlaysBlack == true)
    {
        Debug.Log("black plays");
        whitePlayer = new ChessPlayer(TeamColor.White, board, PlayerType.Person);
        ai = CreateAIOfDifficulty(whitePlayer, blackPlayer, board, UIManager.GiveDifficulty());
        PlayerType pType;
        pType = AINum > 0 ? PlayerType.AI : PlayerType.Person;
        whitePlayer = new ChessPlayer(TeamColor.White, board, pType);
    }
}

```

```

if (pType == PlayerType.AI)
{
    Alactive = true;
}

pType = AINum > 1 ? PlayerType.AI : PlayerType.Person;
blackPlayer = new ChessPlayer(TeamColor.Black, board, pType);
}

else
{
    whitePlayer = new ChessPlayer(TeamColor.White, board, PlayerType.Person);
    ai = CreateAIOfDifficulty(whitePlayer, blackPlayer, board, UIManager.GiveDifficulty());
    PlayerType pType;
    pType = AINum > 1 ? PlayerType.AI : PlayerType.Person;
    whitePlayer = new ChessPlayer(TeamColor.White, board, pType);
    if (pType == PlayerType.AI)
    {
        Alactive = true;
    }

    pType = AINum > 0 ? PlayerType.AI : PlayerType.Person;
    blackPlayer = new ChessPlayer(TeamColor.Black, board, pType);
}
}

private void Start()
{
    StartNewGame();
}

public void StartNewGame()
{

```

```

Alactive = false;

CreatePlayers();

SetGameState(GameState.Init);

board.SetDependencies(this);

CreatePiecesFromLayout(startingBoardLayout);

activePlayer = whitePlayer;

GenerateAllPossiblePlayerMoves(activePlayer);

SetGameState(GameState.Play);

board.DeleteDisplayNotation();

board.pastMovesBlack.Clear();

board.pastMovesWhite.Clear();

UIManager.blackClock.timerDisplay.text = "Waiting...";

if(Alactive)

{

    StartCoroutine(MakeAnAiMove());

}

}

private void SetGameState(GameState state)

{

    this.state = state;

}

internal bool IsGameInProgress()

{

    return state == GameState.Play;

}

private void CreatePiecesFromLayout(BoardLayout layout)

```

```

{
    for (int i = 0; i < layout.GetPiecesCount(); i++)
    {
        Vector3Int squareCoords = layout.GetSquareCoordsAtIndex(i);
        TeamColor team = layout.GetSquareTeamColorAtIndex(i);
        string typeName = layout.GetSquarePieceNameAtIndex(i);

        Type type = Type.GetType(typeName);
        CreatePieceAndInitialize(squareCoords, team, type);
    }
}

public void CreatePieceAndInitialize(Vector3Int squareCoords, TeamColor team, Type type)
{
    Piece newPiece = pieceCreator.CreatePiece(type).GetComponent<Piece>();
    newPiece.SetData(squareCoords, team, board);

    Material teamMaterial = pieceCreator.GetTeamMaterial(team, type);
    newPiece.SetMaterial(teamMaterial);

    if (newPiece.team == TeamColor.White) newPiece.transform.Rotate(0, 0, 180);

    board.SetPieceOnBoard(squareCoords, newPiece);
}

ChessPlayer currentPlayer = team == TeamColor.White ? whitePlayer : blackPlayer;
currentPlayer.AddPiece(newPiece);
}

```

```

private void GenerateAllPossiblePlayerMoves(ChessPlayer player)
{
    player.GenerateAllPossibleMoves();
}

public bool IsTeamTurnActive(TeamColor team)
{
    return activePlayer.team == team;
}

public void EndTurn()
{
    if(activePlayer.team == TeamColor.White)
    {
        whiteClock.remainingTime += UIManager.timelinc;
        whiteClock.UpdateTimeRemaining(whiteClock.remainingTime);
        whiteClock.timerActive = false;
        blackClock.timerActive = true;
    }
    else
    {
        blackClock.remainingTime += UIManager.timelinc;
        blackClock.UpdateTimeRemaining(blackClock.remainingTime);
        blackClock.timerActive = false;
        whiteClock.timerActive = true;
    }
    moveSound.Play();
    ai.IncreaseNumMovesMade();
    UIManager.DisplayNumMovesMade(ai.GetNumMovesMade());
    GenerateAllPossiblePlayerMoves(activePlayer);
}

```

```

GenerateAllPossiblePlayerMoves(GetOpponentToPlayer(activePlayer));

int finNum = CheckIfGameIsFinished();

if (finNum == 1)

{

    EndGame();

}

else if(finNum == 2)

{

    SetGameState(GameState.Finished);

    UIManager.OnGameFinished("Draw");

}

else

{

    ChangeActiveTeam();

}

if (activePlayer.type == PlayerType.AI)

{

    RemoveCheckingMoves();

    Alactive = true;

    StartCoroutine(MakeAnAiMove());

    Alactive = false;

}

//RotateCameraInInstant();

//RotateCamera();

}

private bool CheckStalemate()

{

```

```

if (activePlayer.CheckForMoves())
{
    return true;
}

return false;
}

private void RotateCamera()
{
    var smooth = 1f;
    Vector3 velocity = Vector3.zero;
    float smoothTime = 1f;

    Vector3 targetPosition = activePlayer == whitePlayer ? new Vector3(0, 10.5f, -5.25f) : new Vector3(0, 10.5f, 5.25f);

    Quaternion targetRotation = activePlayer == whitePlayer ? Quaternion.Euler(50, 0, 0) : Quaternion.Euler(130, 0, 180);

    mainCamera.transform.position = Vector3.SmoothDamp(transform.position, targetPosition, ref velocity, smoothTime);

    mainCamera.transform.rotation = Quaternion.RotateTowards(transform.rotation, targetRotation, Time.deltaTime * smooth);

    mainCamera.transform.position = Vector3.SmoothDamp(transform.position, targetPosition*2, ref velocity, smoothTime);
}

private void RotateCameraInstant()
{
    Vector3 targetPosition = activePlayer == whitePlayer ? new Vector3(0, 10.5f, -10.5f) : new Vector3(0, 10.5f, 10.5f);

    Quaternion targetRotation = activePlayer == whitePlayer ? Quaternion.Euler(50, 0, 0) : Quaternion.Euler(130, 0, 180);

    mainCamera.transform.rotation = targetRotation;

    mainCamera.transform.position = targetPosition;
}

```

```

private int CheckIfGamesFinished()
{
    Piece[] kingAttackingPieces = activePlayer.GetPieceAttingOppositePiceOfType<King>();
    if (kingAttackingPieces.Length > 0)
    {
        ChessPlayer oppositePlayer = GetOpponentToPlayer(activePlayer);
        Piece[] attackedKings = oppositePlayer.GetPiecesOfType<King>();

        foreach (var attackedKing in attackedKings)
        {
            oppositePlayer.RemoveMovesEnablingAttackOnPieceOfType<King>(activePlayer, attackedKing);
            int availableKingMoves = attackedKing.availableMoves.Count;
            if (availableKingMoves == 0)
            {
                bool canCoverKing = oppositePlayer.CanHidePieceFromAttack<King>(activePlayer);
                if (!canCoverKing)
                    return 1; //mate
            }
        }
    }
    else
    {
        ChessPlayer oppositePlayer = GetOpponentToPlayer(activePlayer);
        Piece[] attackedKings = oppositePlayer.GetPiecesOfType<King>();

        foreach (var attackedKing in attackedKings)
        {
            oppositePlayer.RemoveMovesEnablingAttackOnPieceOfType<King>(activePlayer, attackedKing);
            foreach (Piece piece in oppositePlayer.activePieces)
            {

```

```

        if(piece.availableMoves.Count != 0)

    {

        return 0; //carry on

    }

}

return 2; //stale

}

}

return 0; // carry on

}

public void RemoveCheckingMoves()

{

foreach(var active in activePlayer.activePieces)

{



activePlayer.RemoveMovesEnablingAttackOnPieceOfType<King>(GetOpponentToPlayer(activePlayer), active);

}

}

public void EndGame()

{

string display = activePlayer.team.ToString() + " Won!";

SetGameState(GameState.Finished);

UIManager.OnGameFinished(display);

}

public void RestartGame()

{

```

```

UIManager.ResetAllNums();
UIManager.AdjustForPlaying();
DestroyPieces();
board.OnGameRestarted();
whitePlayer.OnGameRestarted();
blackPlayer.OnGameRestarted();
StartNewGame();
}

private void DestroyPieces()
{
    whitePlayer.activePieces.ForEach(p => Destroy(p.gameObject));
    blackPlayer.activePieces.ForEach(p => Destroy(p.gameObject));
}

public void ChangeActiveTeam()
{
    activePlayer = activePlayer == whitePlayer ? blackPlayer : whitePlayer;
}

private ChessPlayer GetOpponentToPlayer(ChessPlayer player)
{
    return player == whitePlayer ? blackPlayer : whitePlayer;
}

public ChessPlayer GetActivePlayer()
{
    return activePlayer == whitePlayer ? whitePlayer : blackPlayer;
}

public void SaveState()

```

```

{
    Debug.Log("Saved");

    string filePath = "Assets/Save Games/Cheat.txt";
    string allMoves = "";
    Vector3Int atCoords;
    Vector3Int goCoords;

    for (int i = 0; i < board.pastMovesBlack.Count; i++)
    {
        (atCoords, goCoords) = board.pastMovesWhite.ElementAt(i);
        allMoves += (atCoords.x.ToString());
        allMoves += (atCoords.y.ToString());
        allMoves += (atCoords.z.ToString());
        allMoves += (goCoords.x.ToString());
        allMoves += (goCoords.y.ToString());
        allMoves += (goCoords.z.ToString());
        (atCoords, goCoords) = board.pastMovesBlack.ElementAt(i);
        allMoves += (atCoords.x.ToString());
        allMoves += (atCoords.y.ToString());
        allMoves += (atCoords.z.ToString());
        allMoves += (goCoords.x.ToString());
        allMoves += (goCoords.y.ToString());
        allMoves += (goCoords.z.ToString());
    }

    if( board.pastMovesWhite.Count > board.pastMovesBlack.Count)
    {
        (atCoords, goCoords) = board.pastMovesWhite.ElementAt(board.pastMovesWhite.Count - 1);
        allMoves += (atCoords.x.ToString());
        allMoves += (atCoords.y.ToString());
        allMoves += (atCoords.z.ToString());
        allMoves += (goCoords.x.ToString());
    }
}

```

```

        allMoves += (goCoords.y.ToString());
        allMoves += (goCoords.z.ToString());
    }

    File.WriteAllText(filePath, allMoves);
}

public void LoadState()
{
    AInactive = true;

    string filePath = "Assets/Save Games/Cheat.txt";
    string allMoves = File.ReadAllText(filePath);

    Vector3Int atCoords;
    Vector3Int goCoords;
    for (int i = 0; i < allMoves.Length/3; i+=2)
    {
        atCoords = new Vector3Int(Convert.ToInt32(allMoves.Substring(3 * i,1)),
        Convert.ToInt32(allMoves.Substring(3 * i + 1,1)), Convert.ToInt32(allMoves.Substring(3 * i+2,1)));
        goCoords = new Vector3Int(Convert.ToInt32(allMoves.Substring(3 * i+3,1)),
        Convert.ToInt32(allMoves.Substring(3 * i + 4,1)), Convert.ToInt32(allMoves.Substring(3 * i + 5,1)));
        StartCoroutine(LoadPiecePositions(atCoords, goCoords));
    }

    AInactive = false;
}

internal void OnPieceRemoved(Piece piece)
{
    ChessPlayer pieceOwner = (piece.team == TeamColor.White) ? whitePlayer : blackPlayer;
    pieceOwner.RemovePiece(piece);
}

internal void RemoveMovesEnablingAttackOnPieceOfType<T>(Piece piece) where T : Piece

```

```

{
    activePlayer.RemoveMovesEnablingAttackOnPieceOfType<T>(GetOpponentToPlayer(activePlayer),
piece);
}

IEnumerator MakeAnAiMove()
{
    yield return new WaitForSeconds(0.25f);

    ai.MoveMakerController(SetActivePlayer(),GetOpponentToPlayer(SetActivePlayer()));

}

IEnumerator LoadPiecePositions(Vector3Int atCoords, Vector3Int goCoords)
{
    yield return new WaitForSeconds(0.25f);

    board.AIMakeMove(atCoords, goCoords);

}

IEnumerator Wait(float time)
{
    yield return new WaitForSeconds(time);
}

public void addAI1()
{
    AINum = 1;
}

public void addAI2()
{
    AINum = 2;
}

```

```

public void addAI0()
{
    AINum = 0;
}

}

```

BOARD LAYOUT

```

using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using UnityEngine;

public enum TeamColor
{
    Black, White
}

public enum PieceType
{
    Pawn, Bishop, Knight, Rook, Queen, King, Commoner
}

[CreateAssetMenu(menuName = "Scriptable Objects/Board/Layout")]
public class BoardLayout : ScriptableObject
{
    [Serializable]
    private class BoardSquareSetup
    {
        public Vector3Int position;
        public PieceType pieceType;
        public TeamColor teamColor;
    }

    [SerializeField] private BoardSquareSetup[] boardSquares;

    public int GetPiecesCount()
    {
        return boardSquares.Length;
    }

    public Vector3Int GetSquareCoordsAtIndex(int index)
    {
        return new Vector3Int(boardSquares[index].position.x - 1,
boardSquares[index].position.y - 1, boardSquares[index].position.z -1);
    }

    public string GetSquarePieceNameAtIndex(int index)
    {
        return boardSquares[index].pieceType.ToString();
    }

    public TeamColor GetSquareTeamColorAtIndex(int index)
    {
        return boardSquares[index].teamColor;
    }
}

```

}

CHESS PLAYER

```
using System;
using System.Linq;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[Serializable]
public class ChessPlayer
{
    public TeamColor team { get; set; }
    public PlayerType type { get; set; }
    public Board board { get; set; }
    public List<Piece> activePieces { get; private set; }

    public ChessPlayer(TeamColor team, Board board, PlayerType type)
    {
        activePieces = new List<Piece>();
        this.board = board;
        this.team = team;
        this.type = type;
    }

    public void AddPiece(Piece piece)
    {
        if (!activePieces.Contains(piece))
            activePieces.Add(piece);
    }

    public void RemovePiece(Piece piece)
    {
        if (activePieces.Contains(piece))
            activePieces.Remove(piece);
    }

    public void GenerateAllPossibleMoves()
    {
        foreach (var piece in activePieces)
        {
            if(board.HasPiece(piece))
                piece.SelectAvailableSquares();
        }
    }

    public (List<List<Vector3Int>>, List<Piece>, List<Vector3Int>)
ReturnAllPossibleMoves()
    {
        List<List<Vector3Int>> movesCurrentlyAvailable = new
List<List<Vector3Int>>();
        List<Piece> pieces = new List<Piece>();
        List<Vector3Int> pieceCoords = new List<Vector3Int>();
        for (int i=0;i<activePieces.Count;i++)
        {
            if ((activePieces[i].SelectAvailableSquares()).Count != 0)
            {
                pieces.Add(activePieces[i]);
                pieceCoords.Add(activePieces[i].occupiedSquare);
            }
        }
    }
}
```

```

        movesCurrentlyAvailable.Add(activePieces[i].SelectAvailableSquares()));
    }
}
for (int i = 0; i < movesCurrentlyAvailable.Count; i++)
{
    for (int j = 0; j < movesCurrentlyAvailable.ElementAt(i).Count;
j++)
    {
        string nameOfPiece;
        if
(board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j)) != null)
        {
            nameOfPiece =
board.GetPieceOnSquare(movesCurrentlyAvailable.ElementAt(i).ElementAt(j)).name;
            if (nameOfPiece == "King" || nameOfPiece ==
"King(Clone)")
            {
                movesCurrentlyAvailable.ElementAt(i).RemoveAt(j);
            }
        }
    }
    return (movesCurrentlyAvailable, pieces, pieceCoords);
}

public Piece[] GetPieceAttackingOppositePieceOfType<T>() where T : Piece
{
    return activePieces.Where(p => p.IsAttackingPieceOfType<T>()).ToArray();
}

public Piece[] GetPiecesOfType<T>() where T : Piece
{
    return activePieces.Where(p => p is T).ToArray();
}

public void RemoveMovesEnablingAttackOnPieceOfType<T>(ChessPlayer opponent,
Piece selectedPiece) where T : Piece
{
    List<Vector3Int> coordsToRemove = new List<Vector3Int>();

    coordsToRemove.Clear();
    foreach (var coords in selectedPiece.availableMoves)
    {
        Piece pieceOnCoords = board.GetPieceOnSquare(coords);
        board.UpdateBoardOnPieceMove(coords,
selectedPiece.occupiedSquare, selectedPiece, null);
        opponent.GenerateAllPossibleMoves();
        if (opponent.CheckIfIsAttacingPiece<T>())
            coordsToRemove.Add(coords);
        board.UpdateBoardOnPieceMove(selectedPiece.occupiedSquare,
coords, selectedPiece, pieceOnCoords);
    }
    foreach (var coords in coordsToRemove)
    {
        selectedPiece.availableMoves.Remove(coords);
    }
}

internal bool CheckIfIsAttacingPiece<T>() where T : Piece

```

```

    {
        foreach (var piece in activePieces)
        {
            if (board.HasPiece(piece) && piece.IsAttackingPieceOfType<T>())
                return true;
        }
        return false;
    }

    public bool CanHidePieceFromAttack<T>(ChessPlayer opponent) where T : Piece
    {
        foreach (var piece in activePieces)
        {
            foreach (var coords in piece.availableMoves)
            {
                Piece pieceOnCoords = board.GetPieceOnSquare(coords);
                board.UpdateBoardOnPieceMove(coords, piece.occupiedSquare,
piece, null);
                opponent.GenerateAllPossibleMoves();
                if (!opponent.CheckIfIsAttacingPiece<T>())
                {
                    board.UpdateBoardOnPieceMove(piece.occupiedSquare,
coords, piece, pieceOnCoords);
                    return true;
                }
                board.UpdateBoardOnPieceMove(piece.occupiedSquare, coords,
piece, pieceOnCoords);
            }
        }
        return false;
    }

    internal void OnGameRestarted()
    {
        activePieces.Clear();
    }

    public bool CheckForMoves()
    {

        if(activePieces.Count != 0)
        {
            foreach(Piece piece in activePieces)
            {

                if (piece.availableMoves.Count != 0)
                {
                    return false;
                }
            }
        }
        return true;
    }
}

```

PIECE (THE PIECE CLASS- PARENT TO ALL PIECE CLASSES)

```

using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using UnityEngine;

```

```

[RequireComponent(typeof(MaterialSetter))]
[RequireComponent(typeof(IObjectTweener))]
public abstract class Piece : MonoBehaviour
{
    [SerializeField] private MaterialSetter materialSetter;
    public Board board { protected get; set; }
    public Vector3Int occupiedSquare { get; set; }
    public TeamColor team { get; set; }
    public bool hasMoved { get; private set; }
    public List<Vector3Int> availableMoves;

    private IObjectTweener tweener;

    public abstract List<Vector3Int> SelectAvailableSquares();

    private void Awake()
    {
        availableMoves = new List<Vector3Int>();
        tweener = GetComponent<IObjectTweener>();
        materialSetter = GetComponent<MaterialSetter>();
        hasMoved = false;
    }

    public void SetMaterial(Material selectedMaterial)
    {
        materialSetter.SetSingleMaterial(selectedMaterial);
    }

    public bool IsFromSameTeam(Piece piece)
    {
        return team == piece.team;
    }

    public bool CanMoveTo(Vector3Int coords)
    {
        return availableMoves.Contains(coords);
    }

    public virtual void MovePiece(Vector3Int coords, Piece piece = null, bool
isPawn = false)
    {
        Vector3 targetPosition = board.CalculatePositionFromCoords(coords);
        occupiedSquare = coords;
        hasMoved = true;
        tweener.MoveTo(transform, targetPosition);
    }

    protected void TryToAddMove(Vector3Int coords)
    {
        availableMoves.Add(coords);
    }

    public void SetData(Vector3Int coords, TeamColor team, Board board)
    {
        this.team = team;
        occupiedSquare = coords;
        this.board = board;
        transform.position = board.CalculatePositionFromCoords(coords);
    }
}

```

```

        public bool IsAttackingPieceOfType<T>() where T : Piece
    {
        foreach (var square in availableMoves)
        {
            if (board.GetPieceOnSquare(square) is T)
                return true;
        }
        return false;
    }
}

```

PIECES CREATOR

```

using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class PiecesCreator : MonoBehaviour
{
    [SerializeField] private GameObject[] piecesPrefabs;
    [SerializeField] private Material blackMaterial;
    [SerializeField] private Material whiteMaterial;
    [SerializeField] private Material blueMaterial;
    [SerializeField] private Material redMaterial;
    private Dictionary<string, GameObject> nameToPieceDict = new Dictionary<string, GameObject>();

    private void Awake()
    {
        foreach (var piece in piecesPrefabs)
        {
            nameToPieceDict.Add(piece.GetComponent<Piece>().GetType().ToString(),
piece);
        }
    }

    public GameObject CreatePiece(Type type)
    {
        GameObject prefab = nameToPieceDict[type.ToString()];
        if (prefab)
        {
            GameObject newPiece = Instantiate(prefab);
            return newPiece;
        }
        return null;
    }

    public Material GetTeamMaterial(TeamColor team, Type type)
    {
        Material colour = team == TeamColor.White ? whiteMaterial : blackMaterial;
        if (type.ToString() == "King")
        {
            colour = team == TeamColor.White ? blueMaterial : redMaterial;
        }
        return colour;
    }
}

```

SQAURE SELECTOR CREATOR

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class SquareSelectorCreator : MonoBehaviour
{
    [SerializeField] private Material freeSquareMaterial;
    [SerializeField] private Material enemySquareMaterial;
    [SerializeField] private GameObject selectorPrefab;
    private List<GameObject> instantiatedSelectors = new List<GameObject>();

    public void ShowSelection(Dictionary<Vector3, bool> squareData)
    {
        ClearSelection();
        foreach (var data in squareData)
        {
            GameObject selector = Instantiate(selectorPrefab, data.Key,
Quaternion.identity);
            instantiatedSelectors.Add(selector);
            foreach (var setter in
selector.GetComponentsInChildren<MaterialSetter>())
            {
                setter.SetSingleMaterial(data.Value ? freeSquareMaterial :
enemySquareMaterial);
            }
        }
    }

    public void ClearSelection()
    {
        for (int i = 0; i < instantiatedSelectors.Count; i++)
        {
            Destroy(instantiatedSelectors[i]);
        }
    }
}
```

DATA STRUCTURES

PIECE MOVES

Needed for old AI implementation – unused – can ignore

```
using UnityEngine;
struct PieceMoves
{
    public string PieceName;
    public Vector3Int CoordsOfMove;
}
```

INPUT SYSTEM

BOARD INPUT HANDLER

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
```

```

[RequireComponent(typeof(Board))]
public class BoardInputHandler : MonoBehaviour, IInputHandler
{
    private Board board;

    private void Awake()
    {
        board = GetComponent<Board>();
    }

    public void ProcessInput(Vector3 inputPosition, GameObject selectedObject, Action onClick)
    {
        board.OnSquareSelected(inputPosition);
    }
}

```

COLLIDER INPUT RECIEVER

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class ColliderInputReciever : InputReciever
{
    private ChessGameController gameController = new ChessGameController();

    private Vector3 clickPosition;
    void Update()
    {
        if (Input.GetMouseButtonDown(0))
        {
            RaycastHit hit;
            Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
            if (Physics.Raycast(ray, out hit))
            {
                clickPosition = hit.point;
                OnInputRecieved();
            }
        }
    }

    public override void OnInputRecieved()
    {
        foreach (var handler in inputHandlers)
        {
            if(gameController.AIactive == false)
            {
                handler.ProcessInput(clickPosition, null, null);
            }
            else
            {
                handler.ProcessInput(new Vector3(1000,1000,1000), null, null);
            }
        }
    }
}

```

DEBUG INPUT HANDLER – UNUSED DURING GAME

```
using System;
```

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class DebugInputHandler : MonoBehaviour, IInputHandler
{
    public void ProcessInput(Vector3 inputPosition, GameObject selectedObject, Action onClick)
    {
        Debug.Log(string.Format("Clicked object {0} in position {1} with callback {2}",
        selectedObject != null ? selectedObject.name.ToString() : "null",
        inputPosition,
        (onClick != null)));
    }
}

```

I INPUT HANDLER (INTERFACE)

```

using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public interface IInputHandler
{
    void ProcessInput(Vector3 inputPosition, GameObject selectedObject, Action onClick);
}

```

INPUT RECIEVER

```

using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public abstract class InputReciever : MonoBehaviour
{
    protected IInputHandler[] inputHandlers;

    public abstract void OnInputRecieved();

    private void Awake()
    {
        inputHandlers = GetComponents<IInputHandler>();
    }
}

```

UI INPUT HANDLER

```

using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class UIInputHandler : MonoBehaviour, IInputHandler
{

```

```

    public void ProcessInput(Vector3 inputPosition, GameObject selectedObject, Action
onClick)
{
    onClick?.Invoke();
}
}

```

UI INPUT RECIEVER

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Events;

public class UIInputReciever : InputReciever
{
    [SerializeField] UnityEvent onClick;

    public override void OnInputReceived()
    {
        foreach (var handler in inputHandlers)
        {
            handler.ProcessInput(Input.mousePosition, gameObject, () =>
onClick.Invoke());
        }
    }
}

```

NON GAME UTILITY

TRAINING DATA

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using System;
using System.IO;

```

```
public class TrainingData
```

```
{
```

```
//note that where try catches are used and they can cause a piece to not be placed even if it may be placed
legally this is NOT a mistake - this is a more efficient way of creating adequate trianing data than checkingB for
legal squares
```

```
//equally for trianing purposes since the training is soley for positional chess evaluation it does not matter is
black and white pieces occupy the same sqaure as the same principles still apply - this reduces the checking
time by 2 fold
```

```
//between sessions of running this program the comments that include (piece name) are changed for differnt
pieces in attempt to stop the network overfitting to that specific piece
```

```

private int[][][] board = new int[2][8][8];//team -white then balck,x,y,z,last is for pieces in order kingB,
commoner, queen, rook, bishop, knight, pawn

private int[] selectToMaterial = new int[13] { 2,1,2,3,3,3,3,5,1,3,-3,-12,1}; //negatives indicated pieces were
given to other team

private float[] selectToAdvantage = new float[13] { -0.1f, -0.1f,-0.2f,-
0.2f,0.2f,0.3f,0.1f,0.3f,0.1f,0.4f,0.3f,0.4f,0.2f};

```

```

private void CreateTD()
{
    string filePath = "Assets/Training Data/T.txt";
    string allData = "";
    float output = 0;
    int pieceNum = 0;
    int material = 0;
    int x = 0;
    int y = 0;
    int z = 0;
    int kingB = 0;
    int kingW = 0;
    bool placed = false;
    System.Random rng = new System.Random();
    while (true)
    {
        for (int j = 0; j < 100; j++)
        {
            for (int i = 0; i < 8; i++)
            {
                for (int m = 0; m < 8; m++)
                {
                    for (int k = 0; k < 8; k++)

```

```

{
    for (int l = 0; l < 7; l++)
    {
        board[0][i][m][k][l] = 0;
        board[l][i][m][k][l] = 0;
    }
}

output = 0;
material = 0;
kingB = 0;
x = rng.Next(0, 8);
y = rng.Next(0, 8);
z = rng.Next(0, 8);
board[l][x][y][z][0] = l;
kingB = 100 * x + 10 * y + z;
kingW = 0;
x = rng.Next(0, 8);
y = rng.Next(0, 8);
z = rng.Next(0, 8);
board[0][x][y][z][0] = l;
kingW = 100 * x + 10 * y + z;
for (int i = 0; i < rng.Next(1, 151); i++)
{
    placed = false;
    pieceNum = rng.Next(1, 15);
    x = rng.Next(0, 8);
    y = rng.Next(0, 8);
}

```

```

z = rng.Next(0, 8);

int count = 0;

foreach (int piece in board[0][x][y][z])

{
    if (piece != 0)

    {
        break;
    }

    count++;
}

if (count == 7)

{
    if (pieceNum == 1) // protected pawns

    {
        count = 0;

        int one = 1;

        int two = 1;

        if(rng.Next(1, 3) == 1)

        {
            one = -1;

        }

        if (rng.Next(1, 3) == 1)

        {
            two = -1;

        }

        try

        {

            foreach (int piece in board[0][x + one][y + 1][z + two])

            {

                if (piece != 0)

```

```

    {
        break;
    }

    count++;
}

if (count == 7)
{
    placed = true;

    board[0][x][y][z][7] = 1;
    board[0][x+one][y+one][z+two][7] = 1;
}

}

catch
{
}

}

else if (pieceNum == 2) // isolated pawn
{
    try
    {
        count = 0;

        foreach (int piece in board[0][x + 1][y - 1][z + 1])
        {
            if (piece != 0)
            {
                break;
            }

            count++;
        }
    }
}

```

```

if (count == 7)
{
    count = 0;

    foreach (int piece in board[0][x - 1][y - 1][z + 1])
    {
        if (piece != 0)
        {
            break;
        }

        count++;
    }

    if (count == 7)
    {
        count = 0;

        foreach (int piece in board[0][x + 1][y - 1][z - 1])
        {
            if (piece != 0)
            {
                break;
            }

            count++;
        }

        if (count == 7)
        {
            count = 0;

            foreach (int piece in board[0][x - 1][y - 1][z - 1])
            {
                if (piece != 0)
                {
                    break;
                }
            }
        }
    }
}

```

```

        }

        count++;

    }

    if (count != 7)

    {

        placed = true;

        board[0][x][y][z][7] = l;

    }

}

}

}

}

}

}

}

}

}

else if (pieceNum == 3) //doubled pawns

{

    for(int a = y; a < 8; a++)

    {

        if (board[0][x][a][z][7] == l)

        {

            placed = true;

            board[0][x][y][z][7] = l;

            break;

        }

    }

}

```

```

else if (pieceNum == 4) // knight on edge of the board[0]
{
    if(x == 0 || x == 7 || y == 0 || y == 7 || z == 0 || z == 7)
    {
        placed = true;
        board[0][x][y][z][6] = 0;
    }
}

else if (pieceNum == 5) // knight with all moves
{
    if (2 < x && x < 6 && 2 < y && y < 6 && 2 < z && z < 6)
    {
        placed = true;
        board[0][x][y][z][6] = 0;
    }
}

else if (pieceNum == 6)//bishop pair
{
    int type = x % 2;
    for (int a = 0; a < 4; a+=2)
    {
        for (int b = 0; b < 4; b+=2)
        {
            for (int c = 0; c < 8; c++)
            {
                if (board[0][a + type][b + type][c][5] == 1)
                {
                    placed = true;
                    board[0][x][y][z][5] = 1;
                    break;
                }
            }
        }
    }
}

```

```

        }

    }

    if (placed)

    {

        break;

    }

}

if (placed)

{

    break;

}

}

}

else if (pieceNum == 7) //bishop and queen

{

    for (int a = -7; a < 8; a++)

    {

        for (int b = -7; b < 8; b++)

        {

            try

            {

                if (board[0][a][b][z][2] == 1)

                {

                    placed = true;

                    board[0][x][y][z][4] = 1;

                    break;

                }

            }

        }

        catch

        {


```

```

    }

    for (int c = -7; c < 8; c++)
    {
        try {
            if (board[0][a][b][c][2] == 1)

            {
                placed = true;

                board[0][x][y][z][4] = 1;

                break;
            }
        }

        catch ({}}

    }

    if(placed)
    { break; }

}

if (placed) { break; }

}

}

}

else if (pieceNum == 8) // rook on open file

{
    int counter = 0;

    for (int a = y; a < 8; a++)
    {
        for (int b = 0; b < 8; b++)
        {

            if(board[0][x][a][z][b] == 0)

            {

```

```

        counter++;
    }

}

if(counter == 8)
{
    placed = true;
    board[0][x][y][z][3] = 1;
}

}

else if (pieceNum == 9) // off starting square (using bishop)
{
    if(x!=2 || (y != 0 && y!=7) || x!=5)
    {
        board[0][x][y][z][4] = 1;
    }
}

else if (pieceNum == 10) // attacking sqaure that touches kingB (pawn)
{
    for (int a = 0; a < 7; a++)
    {
        int counter = 0;
        if (board[0][Mathf.FloorToInt(kingB / 100) - 2][kingB % 100 - kingB % 10 - 2][kingB % 10 - 2][a] == 0)
        {
            counter++;
        }
        if(counter == 7)
        {
            placed = true;
        }
    }
}

```

```

        board[0][Mathf.FloorToInt(kingB / 100)-2][kingB % 100 - kingB % 10 -2][kingB % 10-
2][6] = 1;
    }

}

else if (pieceNum == 11) // pin on kingB (with pawn)
{
    int counter = 0;

    for (int a = 0; a < 7; a++)
    {

        if (board[0][Mathf.FloorToInt(kingB / 100) - 2][kingB % 100 - kingB % 10 - 2][kingB % 10
- 2][a] == 0 && board[1][Mathf.FloorToInt(kingB / 100) - 1][kingB % 100 - kingB % 10 - 1][kingB % 10 - 1][a]
== 0)
        {

            counter++;

        }
    }

    if (counter == 7)
    {
        placed = true;

        board[0][Mathf.FloorToInt(kingB / 100) - 2][kingB % 100 - kingB % 10 - 2][kingB % 10 -
2][6] = 1;

        board[1][Mathf.FloorToInt(kingB / 100) - 1][kingB % 100 - kingB % 10 - 1][kingB % 10 -
1][5] = 1;
    }
}

else if (pieceNum == 12) // pin on queenB (pawn)
{
    int counter = 0;
}

```

```

for (int a = 0; a < 7; a++)
{
    if(board[!][x][y][z][a] == 0 && board[!][x-1][y-1][z-1][a] == 0 && board[0][x-2][y-2][z-2][a] == 0)
    {
        counter++;
    }

}

if (counter == 7)
{
    placed = true;
    board[!][x][y][z][2] = 1;
    board[!][x - 1][y - 1][z - 1][5] = 1;
    board[0][x - 2][y - 2][z - 2][6] = 1;
}
}

else // controlling centre squares (pawn)
{
    if((x == 5 || x == 4) && (y == 5 || y == 4))
    {
        board[0][x][y][z][6] = 1;
    }
}

if (placed)
{
    material += selectToMaterial[pieceNum - 1];
    output += selectToAdvantage[pieceNum - 1];
}
placed = false;

```

```

pieceNum = rng.Next(1, 15);

x = rng.Next(0, 8);

y = rng.Next(0, 8);

z = rng.Next(0, 8);

foreach (int piece in board[1][x][y][z])
{
    if (piece != 0)
    {
        break;
    }
    count++;
}

if (count == 7)
{
    if (pieceNum == 1) // protected pawns
    {
        count = 0;

        int one = 1;

        int two = 1;

        if (rng.Next(1, 3) == 1)
        {
            one = -1;
        }

        if (rng.Next(1, 3) == 1)
        {
            two = -1;
        }

        try
        {

```

```

foreach (int piece in board[1][x + one][y + 1][z + two])
{
    if (piece != 0)
    {
        break;
    }
    count++;
}

if (count == 7)
{
    placed = true;
    board[1][x][y][z][7] = 1;
    board[1][x + one][y + one][z + two][7] = 1;
}

}
catch
{

}

else if (pieceNum == 2) // isolated pawn
{
    try
    {
        count = 0;

        foreach (int piece in board[1][x + 1][y - 1][z + 1])
        {
            if (piece != 0)
            {
                break;
            }
        }
    }
}

```

```

    }

    count++;

}

if (count == 7)

{

    count = 0;

    foreach (int piece in board[!][x - 1][y - 1][z + 1])

    {

        if (piece != 0)

        {

            break;

        }

        count++;

    }

    if (count == 7)

    {

        count = 0;

        foreach (int piece in board[!][x + 1][y - 1][z - 1])

        {

            if (piece != 0)

            {

                break;

            }

            count++;

        }

        if (count == 7)

        {

            count = 0;

            foreach (int piece in board[!][x - 1][y - 1][z - 1])

            {

```

```

        if (piece != 0)

    {

        break;

    }

    count++;

}

if (count != 7)

{

    placed = true;

    board[!][x][y][z][7] = !;

}

}

}

}

}

}

}

}

}

}

}

else if (pieceNum == 3) //doubled pawns

{



for (int a = y; a < 8; a++)

{

    if (board[!][x][a][z][7] == !)

    {

        placed = true;

        board[!][x][y][z][7] = !;

        break;

}

```

```

        }

    }

}

else if (pieceNum == 4) // knight on edge of the board[1

    //

{

    if (x == 0 || x == 7 || y == 0 || y == 7 || z == 0 || z == 7)

    {

        placed = true;

        board[1][x][y][z][6] = 0;

    }

}

else if (pieceNum == 5) // knight with all moves

{

    if (2 < x && x < 6 && 2 < y && y < 6 && 2 < z && z < 6)

    {

        placed = true;

        board[1][x][y][z][6] = 0;

    }

}

else if (pieceNum == 6)//bishop pair

{

    int type = x % 2;

    for (int a = 0; a < 4; a += 2)

    {

        for (int b = 0; b < 4; b += 2)

        {

            for (int c = 0; c < 8; c++)

            {

                if (board[1][a + type][b + type][c][5] == 1)

```

```

    {
        placed = true;
        board[!][x][y][z][5] = !;
        break;
    }
}

if (placed)
{
    break;
}

}

if (placed)
{
    break;
}

}

}

else if (pieceNum == 7) //bishop and queen
{
    for (int a = -7; a < 8; a++)
    {
        for (int b = -7; b < 8; b++)
        {
            try
            {
                if (board[!][a][b][z][2] == !)
                {
                    placed = true;
                    board[!][x][y][z][4] = !;
                    break;
                }
            }
        }
    }
}

```

```

        }

    }

    catch

    {

    }

    for (int c = -7; c < 8; c++)
    {
        try
        {
            if (board[l][a][b][c][2] == l)
            {
                placed = true;
                board[l][x][y][z][4] = l;
                break;
            }
        }
        catch {}
    }

    if (placed)
    {
        break;
    }

    if (placed) { break; }

}

}

else if (pieceNum == 8) // rook on open file
{
    int counter = 0;
    for (int a = y; a < 8; a++)

```

```

{
    for (int b = 0; b < 8; b++)
    {
        if (board[1][x][a][z][b] == 0)
        {
            counter++;
        }
    }

    if (counter == 8)
    {
        placed = true;
        board[1][x][y][z][3] = 1;
    }
}

else if (pieceNum == 9) // off starting square (using bishop)
{
    if (x != 2 || (y != 0 && y != 7) || x != 5)
    {
        board[1][x][y][z][4] = 1;
    }
}

else if (pieceNum == 10) // attacking square that touches kingW (pawn)
{
    for (int a = 0; a < 7; a++)
    {
        int counter = 0;

        if (board[1][Mathf.FloorToInt(kingW / 100) - 2][kingW % 100 - kingW % 10 - 2][kingW % 10 - 2][a] == 0)
        {
    }
}

```

```

        counter++;
    }

    if (counter == 7)
    {
        placed = true;

        board[1][Mathf.FloorToInt(kingW / 100) - 2][kingW % 100 - kingW % 10 -
2][kingW % 10 - 2][6] = 1;
    }

}

else if (pieceNum == 11) // pin on kingW (with pawn)
{
    int counter = 0;

    for (int a = 0; a < 7; a++)
    {

        if (board[1][Mathf.FloorToInt(kingW / 100) - 2][kingW % 100 - kingW % 10 -
2][kingW % 10 - 2][a] == 0 && board[0][Mathf.FloorToInt(kingW / 100) - 1][kingW % 100 - kingW % 10 -
1][kingW % 10 - 1][a] == 0)
        {

            counter++;
        }
    }

    if (counter == 7)
    {
        placed = true;

        board[1][Mathf.FloorToInt(kingW / 100) - 2][kingW % 100 - kingW % 10 - 2][kingW %
10 - 2][6] = 1;
    }

    board[0][Mathf.FloorToInt(kingW / 100) - 1][kingW % 100 - kingW % 10 - 1][kingW %
10 - 1][5] = 1;
}

```

```

    }

}

else if (pieceNum == 12) // pin on queenW (pawn)

{
    int counter = 0;

    for (int a = 0; a < 7; a++)

    {
        if (board[0][x][y][z][a] == 0 && board[0][x - 1][y - 1][z - 1][a] == 0 && board[1][x - 2][y - 2][z - 2][a] == 0)
            {

                counter++;

            }

    }

    if (counter == 7)

    {
        placed = true;

        board[0][x][y][z][2] = 1;

        board[0][x - 1][y - 1][z - 1][5] = 1;

        board[1][x - 2][y - 2][z - 2][6] = 1;

    }

}

else // controlling centre squares (pawn)

{
    if ((x == 5 || x == 4) && (y == 5 || y == 4))

    {
        board[1][x][y][z][6] = 1;

    }

}

if (placed)

```

```

    {
        material -= selectToMaterial[pieceNum - 1];
        output -= selectToAdvantage[pieceNum - 1];
    }
}

}

int team = 0;
int num = 0;

if (material < 0)
{
    team = 1;
    material = -1 * material;
}

for (int a = 0; a < 8; a++)
{
    for (int b = 0; b < 8; b++)
    {
        for (int c = 0; c < 8; c++)
        {
            num = 0;
            for (int d = 0; d < 7; d++)
            {
                if(board[team][a][b][c][d] == 0)
                {
                    num++;
                }
            }
            if (num == 7)
            {

```

```

material--;

board[team][a][b][c][6] = 1;

//protected

if(board[team][a + 1][b + 1][c + 1][6] == 1 || board[team][a - 1][b + 1][c + 1][6] == 1 ||
board[team][a + 1][b + 1][c - 1][6] == 1 || board[team][a - 1][b + 1][c - 1][6] == 1)

{

    output += team == 0 ? 0.1f : -0.1f;

}

else // isolated

{

    output += team == 0 ? -0.1f : 0.1f;

}

for (int i = b; i < 8; i++) // doubled

{

    if(board[team][a][i][c][6] == 1)

    {

        output += team == 0 ? -0.2f : 0.2f;

    }

}

if (material == 0)

{

    break;

}

}

}

if (material == 0)

{

    break;

}

```

```

        }

        if (material == 0)

        {

            break;

        }

    }

    for (int a = 0; a < 2; a++)

    {

        for (int b = 0; b < 8; b++)

        {

            for (int c = 0; c < 8; c++)

            {

                for (int d = 0; d < 8; d++)

                {

                    for (int e = 0; e < 7; e++)

                    {

                        allData += board[a][b][c][d][e];

                    }

                }

            }

        }

    }

    allData += output;

}

File.WriteAllText(filePath, allData);

}

}

}


```

SOUND

BACKGROUND MUSIC

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class BackGroundMusic : MonoBehaviour
{
    private bool mute = false;
    public void MuteSound()
    {
        if (mute == false)
        {
            AudioListener.volume = 0;
            mute = true;
        }
        else
        {
            AudioListener.volume = 0.3f;
            mute = false;
        }
    }
}

```

SOUND MANAGER

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class SoundManager : MonoBehaviour
{
    [SerializeField] private GameObject muteBG;
    [SerializeField] private GameObject unmuteBG;
    [SerializeField] private GameObject muteSF;
    [SerializeField] private GameObject unmuteSF;

    public void ActivateMuteBG()
    {
        muteBG.SetActive(true);
        unmuteBG.SetActive(false);
    }

    public void ActivateUnmuteBG()
    {
        muteBG.SetActive(false);
        unmuteBG.SetActive(true);
    }

    public void ActivateMuteSF()
    {
        muteSF.SetActive(true);
        unmuteSF.SetActive(false);
    }

    public void ActivateUnmuteSF()
    {
        muteSF.SetActive(false);
        unmuteSF.SetActive(true);
    }
}

```

TWEENERS

INSTANT TWEENER

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class InstantTweener : MonoBehaviour, IObjectTweener
{
    public void MoveTo(Transform transform, Vector3 targetPosition)
    {
        transform.position = targetPosition;
    }
}
```

I OBJECT TWEENER (INTERFACE)

```
using UnityEngine;

public interface IObjectTweener
{
    void MoveTo(Transform transform, Vector3 targetPosition);
}
```

UI CONTROLLERS

CAMERA FREE LOOK – 3RD PARTY CODE

```
using UnityEngine;
using System;

public class CameraFreeLook : MonoBehaviour
{
    [Header("Focus Object")]
    [SerializeField, Tooltip("Enable double-click to focus on objects?")]
    private bool doFocus = false;
    [SerializeField] private float focusLimit = 100f;
    [SerializeField] private float minFocusDistance = 5.0f;
    private float doubleClickTime = .15f;
    private float cooldown = 0;
    [Header("Undo Focus")]
    [SerializeField] private KeyCode firstUndoKey = KeyCode.LeftControl;
    [SerializeField] private KeyCode secondUndoKey = KeyCode.Z;

    [Header("Movement")]
    [SerializeField] private float moveSpeed = 1.0f;
    [SerializeField] private float rotationSpeed = 0.025f;
    [SerializeField] private float zoomSpeed = 10.0f;

    //Cache last pos and rot be able to undo last focus object action.
    Quaternion prevRot = new Quaternion();
    Vector3 prevPos = new Vector3();

    [Header("Axes Names")]
    [SerializeField, Tooltip("Vertical axis")] private string mouseY = "Mouse Y";
    [SerializeField, Tooltip("Horizontal axis")] private string mouseX = "Mouse X";
    [SerializeField, Tooltip("Zoom")] private string zoomAxis = "Mouse ScrollWheel";
```

```

[Header("Move Keys")]
[SerializeField] private KeyCode forwardKey = KeyCode.W;
[SerializeField] private KeyCode backKey = KeyCode.S;
[SerializeField] private KeyCode leftKey = KeyCode.A;
[SerializeField] private KeyCode rightKey = KeyCode.D;
[SerializeField] private KeyCode upKey = KeyCode.Q;
[SerializeField] private KeyCode downKey = KeyCode.E;
[SerializeField, Tooltip("Key to stop zoom")] private KeyCode stopZoom =
KeyCode.LeftControl;

[Header("Anchored Movement"), Tooltip("By default in scene-view, this is done by
right-clicking for rotation or middle mouse clicking for up and down")]
[SerializeField] private KeyCode anchoredMoveKey = KeyCode.Mouse2;

[SerializeField] private KeyCode anchoredRotateKey = KeyCode.Mouse1;

private void Start()
{
    SavePosAndRot();
}

void Update()
{
    if (!doFocus)
        return;

    //Double click for focus
    if (cooldown > 0 && Input.GetKeyDown(KeyCode.Mouse0))
        FocusObject();
    if (Input.GetKeyDown(KeyCode.Mouse0))
        cooldown = doubleClickTime;

    //-----UNDO FOCUS-----
    if (Input.GetKey(firstUndoKey))
    {
        if (Input.GetKeyDown(secondUndoKey))
            GoBackToLastPosition();
    }

    cooldown -= Time.deltaTime;
}

private void LateUpdate()
{
    Vector3 move = Vector3.zero;

    //Move and rotate the camera

    if (Input.GetKey(forwardKey))
        move += Vector3.forward * moveSpeed;
    if (Input.GetKey(backKey))
        move += Vector3.back * moveSpeed;
    if (Input.GetKey(leftKey))
        move += Vector3.left * moveSpeed;
    if (Input.GetKey(rightKey))
        move += Vector3.right * moveSpeed;
    if (Input.GetKey(upKey))
        move += Vector3.up * moveSpeed;
    if (Input.GetKey(downKey))
        move += Vector3.down * moveSpeed;

    float mouseMoveY = Input.GetAxis(mouseY);
}

```

```

float mouseMoveX = Input.GetAxis(mouseX);

//Move the camera when anchored
if (Input.GetKey(anchoredMoveKey))
{
    move += Vector3.up * mouseMoveY * -moveSpeed;
    move += Vector3.right * mouseMoveX * -moveSpeed;
}

//Rotate the camera when anchored
if (Input.GetKey(anchoredRotateKey))
{
    transform.RotateAround(transform.position, transform.right, mouseMoveY * -rotationSpeed);
    transform.RotateAround(transform.position, Vector3.up, mouseMoveX * rotationSpeed);
}

transform.Translate(move);

//Scroll to zoom
if (!Input.GetKeyDown(stopZoom))
{
    float mouseScroll = Input.GetAxis(zoomAxis);
    transform.Translate(Vector3.forward * mouseScroll * zoomSpeed);
}
}

private void FocusObject()
{
    //To be able to undo
    SavePosAndRot();

    //If we double-clicked an object in the scene, go to its position
    Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
    RaycastHit hit;

    if (Physics.Raycast(ray, out hit, focusLimit))
    {
        GameObject target = hit.collider.gameObject;
        Vector3 targetPos = target.transform.position;
        Vector3 targetSize = hit.collider.bounds.size;

        transform.position = targetPos + GetOffset(targetPos, targetSize);

        transform.LookAt(target.transform);
    }
}

private void SavePosAndRot()
{
    prevRot = transform.rotation;
    prevPos = transform.position;
}

private void GoBackToLastPosition()
{
    transform.position = prevPos;
    transform.rotation = prevRot;
}

private Vector3 GetOffset(Vector3 targetPos, Vector3 targetSize)

```

```

    {
        Vector3 dirToTarget = targetPos - transform.position;

        float focusDistance = Mathf.Max(targetSize.x, targetSize.z);
        focusDistance = Mathf.Clamp(focusDistance, minFocusDistance, focusDistance);

        return -dirToTarget.normalized * focusDistance;
    }
}

```

CHESS UI MANAGER

```

using System;
using System.Linq;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class ChessUIManager : MonoBehaviour
{

    [SerializeField] private GameObject gameOverParent;
    [SerializeField] private GameObject settingsButtonParent;
    [SerializeField] private GameObject settingsMenuParent;
    [SerializeField] private GameObject pvp;
    [SerializeField] private GameObject pva;
    [SerializeField] private GameObject ava;
    [SerializeField] private GameObject timers;

    [SerializeField] private Button restartButton;
    [SerializeField] private Text finishText;
    [SerializeField] private Text numMovesMade;

    [SerializeField] private Text timeMain;
    [SerializeField] private Text timeIncrem;
    [SerializeField] private Text aiDiff;
    public bool playerPlaysBlack = false;
    public int difficulty;

    [SerializeField] public TimerHelper whiteClock;
    [SerializeField] public TimerHelper blackClock;

    public float timeInc;

    [Header("THIS MUST HAVE SPECIFIC ORDER"), Tooltip("WP | WB | WN | WC | WR | WQ
    | BP | BB | BN | BC | BR | BQ ")]
    [SerializeField] private List<Text> takenPieceNums = new List<Text>();

    private int[] takenPieceValues = new int[12] { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 };

    private Dictionary<(TeamColor, string), int> teamAndPieceToIndexDict = new
    Dictionary<(TeamColor, string), int>()
    {
        {(TeamColor.White, "P"), 0},
        {(TeamColor.White, "B"), 1},
        {(TeamColor.White, "K"), 2},
        {(TeamColor.White, "C"), 3},
        {(TeamColor.White, "R"), 4},
        {(TeamColor.White, "Q"), 5},
        ...
    };
}

```

```

        {(TeamColor.Black, "P"), 6},
        {(TeamColor.Black, "B"), 7},
        {(TeamColor.Black, "K"), 8},
        {(TeamColor.Black, "C"), 9},
        {(TeamColor.Black, "R"), 10},
        {(TeamColor.Black, "Q"), 11 }

    };

    public void ConfirmPress()
    {
        pva.SetActive(false);
        pvp.SetActive(false);
        ava.SetActive(false);
        DifficultySelection();
    }

    public void WhiteTeamClick()
    {
        playerPlaysBlack = false;
    }

    public void BlackTeamClick()
    {
        playerPlaysBlack = true;
    }

    public void ActivatePvp()
    {
        pvp.SetActive(true);
    }

    public void ActivatePva()
    {
        pva.SetActive(true);
        gameOverParent.SetActive(false);
    }

    public void ActivateAva()
    {
        ava.SetActive(true);
        gameOverParent.SetActive(false);
    }

    public void AdjustForPlaying()
    {
        settingsMenuParent.SetActive(false);
        gameOverParent.SetActive(false);
        settingsButtonParent.SetActive(true);
    }

    internal void OnGameFinished(string winner)
    {
        gameOverParent.SetActive(true);
        settingsMenuParent.SetActive(false);
        settingsButtonParent.SetActive(true);
        finishText.text = winner;
    }

    public void SettingButtonPress()
    {
        gameOverParent.SetActive(false);
        settingsMenuParent.SetActive(true);
        settingsButtonParent.SetActive(false);
    }

```

```

        pva.SetActive(false);
        pvp.SetActive(false);
        ava.SetActive(false);
    }

    public void SettingReturnPress()
    {
        gameOverParent.SetActive(false);
        settingsMenuParent.SetActive(false);
        settingsButtonParent.SetActive(true);
        pva.SetActive(false);
        pvp.SetActive(false);
        ava.SetActive(false);
    }

    public void GoToMainMenu()
    {
        gameOverParent.SetActive(true);
        settingsMenuParent.SetActive(false);
        settingsButtonParent.SetActive(true);
        timers.SetActive(false);
        pva.SetActive(false);
        pvp.SetActive(false);
        ava.SetActive(false);
    }

    public void TimeSelection()
    {
        timers.SetActive(true);
        whiteClock.remainingTime = Convert.ToInt32(timeMain.text)*60;
        blackClock.remainingTime = Convert.ToInt32(timeMain.text)*60;
        blackClock.timerActive = false;
        timeInc = Convert.ToInt32(timeIncrem.text);
    }

    public void DifficultySelection()
    {
        try
        {
            if (Convert.ToInt32(aiDiff.text) < 6 &&
Convert.ToInt32(aiDiff.text) > 0)
            {
                difficulty = Convert.ToInt32(aiDiff.text);
            }
            else
            {
                difficulty = 1;
            }
        }
        catch
        {
            difficulty = 1;
        }
    }

    public int GiveDifficulty()
    {
        return difficulty;
    }

    public void IncreaseTakenPiece(Piece piece)
    {

```

```

        int index = teamAndPieceToIndexDict[(piece.team,
piece.name[0].ToString())];
        takenPieceValues[index]++;
        takenPieceNums.ElementAt(index).text =
(takenPieceValues[index]).ToString();
    }

    public void ResetAllNums()
{
    for (int i = 0; i < takenPieceNums.Count; i++)
    {
        takenPieceNums.ElementAt(i).text = "0";
        takenPieceValues[i] = 0;
        numMovesMade.text = "0";
    }
}

public void DisplayNumMovesMade(int moves)
{
    numMovesMade.text = moves.ToString();
}
}

```

UTILS

BOARD BUTTON

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class BoardButton : MonoBehaviour
{
    [SerializeField] MeshRenderer respectiveBoard;
    [SerializeField] Material zero;
    [SerializeField] Material max;
    [SerializeField] GameObject button;

    private bool current = true;
    private Color minC= new Color(1,1,1,0.25f);
    private Color MaxC = new Color(1, 1, 1, 0.75f);

    public void ButtonChange()
    {
        if (current)
        {
            respectiveBoard.material = zero;
            button.GetComponent<Image>().color = minC;
            current = false;
        }
        else
        {
            respectiveBoard.material = max;
            button.GetComponent<Image>().color = MaxC;
            current = true;
        }
    }
}

```

CUSTOM EXCEPTION – UNUSED DURING GAMEPLAY

```

using System;

[Serializable]

public class NeuralNetworkInstantiationFailed : Exception
{
    public NeuralNetworkInstantiationFailed(string reason) : base(reason)
    {
    }

}

}

```

MATERIAL SETTER

```

using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

[RequireComponent(typeof(MeshRenderer))]
public class MaterialSetter : MonoBehaviour
{
    [SerializeField] private MeshRenderer _meshRenderer;
    private MeshRenderer meshRenderer
    {
        get
        {
            if (_meshRenderer == null)
                _meshRenderer = GetComponent<MeshRenderer>();
            return _meshRenderer;
        }
    }

    public void SetSingleMaterial(Material material)
    {
        meshRenderer.material = material;
    }
}

```

OFFICIAL NOTATION – UNSUED DURING GAMEPLAY (COVERED IN BOARD)

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class OfficialNotation
{
    string moveNotation;

    public void CreateNotation(Piece piece, Vector3Int atCoords, Vector3Int toCoords,
bool take, bool castle = false)
    {
        string pieceChar = piece.name[0].ToString();
        Debug.Log(pieceChar);
        string start = (atCoords.z.ToString() + (char)(65 + (atCoords.y)) +
atCoords.x.ToString());
        string finish = (toCoords.z.ToString() + (char)(65 + (atCoords.y)) +
atCoords.x.ToString());
        string comblation = take ? "x" : " ";
        moveNotation = (pieceChar + start + comblation + finish);
    }
}

```

```

        if (castle) moveNotation = atCoords.z.ToString() + "0-0" +
    toCoords.z.ToString();
    }

    public void DisplayNotation()
    {
        Debug.Log(moveNotation);
    }
}

```

RANDOM HELPER – UNITY RANDOM FAILS DUE TO SEED ISSUES

The seed issues that arise from unity's random mean AI makes same moves

TIME HELPER

```

using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class TimerHelper : MonoBehaviour
{
    [SerializeField] private ChessGameController chessGameController;

    public float remainingTime;
    public bool timerActive;

    [SerializeField] public Text timerDisplay;

    void Update()
    {
        if (timerActive)
        {
            if (remainingTime > 0)
            {
                remainingTime -= Time.deltaTime;
                UpdateTimeRemaining(remainingTime);
            }
            else
            {
                chessGameController.ChangeActiveTeam();
                chessGameController.EndGame();
                remainingTime = 0;
                timerActive = false;
            }
        }
    }

    public void UpdateTimeRemaining(float time)
    {
        time++;

        float hours = Mathf.FloorToInt(time / 3600);
        float mins = Mathf.FloorToInt((time - hours * 3600) / 60);
        float secs = Mathf.FloorToInt(time % 60);

        timerDisplay.text = string.Format("{0:00} : {1:00}: {2:00}", hours, mins, secs);
    }
}

```

UI BUTTON

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

[RequireComponent(typeof(UIInputReciever))]
public class UIButton : Button
{
    private InputReciever reciever;
    protected override void Awake()
    {
        reciever = GetComponent<UIInputReciever>();
        onClick.AddListener(() => reciever.OnInputReceived());
    }
}
```

4.0 TESTING

PRELIMINARY

[Note that each of the testing objectives links directly to those in design and analysis]

[My microphone is not the best – in all testing videos there is speaking but you may have to turn your volume up to hear it]

You need to be signed into a google account to view the content in the videos.

Table of QR Codes:

Test	QR Code - Links
The Board	https://drive.google.com/file/d/1fNhfQDFRjIn7LR1m2qxq8_wufa6ohrOn/view?usp=share_link

	
The Game	https://drive.google.com/file/d/1P_QHI3i7d4_Z3KJNa_NvEuw3T7rGGZh6/view?usp=sharing 
UI	https://drive.google.com/file/d/1dIYP3whWLXkN6hpcErFNKTLKL2xOJw8Z/view?usp=sharing 
Ai1	https://drive.google.com/file/d/1AviV2v248TPKDxctyYy3FwVjcoFGoMfg/view?usp=sharing

	
Ai2-1	https://drive.google.com/file/d/1S9ijMHfFmvPohUn5oZJ6BBWpzO3tqm_n/view?usp=sharing 
Ai2-2	https://drive.google.com/file/d/1g0_UWFSLH6JwvRB8aN CX9XsOu9UYKg-E/view?usp=sharing 

Ai3-1	<p>https://drive.google.com/file/d/1xQ-0aF9L2FEhM-wmQHJz7WtWmFXBN3Vb/view?usp=sharing</p> 
Ai3-2	<p>https://drive.google.com/file/d/1bX1qqo6hNbZ35UJKWOxQqVVhQE8INcBo/view?usp=sharing</p> 
Ai4-1	<p>https://drive.google.com/file/d/1RiMu6ynkAT4-griymNg4189O4wqW62l0/view?usp=sharing</p> 

Ai4-2	https://drive.google.com/file/d/1kXEfn2JceSxZif9k4h4kiitzFf4n3Yok/view?usp=sharing 
Ai4-3	https://drive.google.com/file/d/11PUWT6HrZeCKzv3IzSQ6PcdtrauoYRyM/view?usp=sharing 
Ai5-1	https://drive.google.com/file/d/1fucjt-9S3sGcmIhgW9m6hUFV8rbZA8dw/view?usp=sharing 
Ai5-2	https://drive.google.com/file/d/1y-xgXZkNu-9ZOMsdpNKGKroFM8FeUbEg/view?usp=sharing

	
Ai5-3	https://drive.google.com/file/d/1krlhZIWJNknhkTIV1-n4sQKlod7NM1rG/view?usp=sharing 

4.1 – THE BOARD

Objective	Normal Data	Boundary Data	Erroneous Data	Pass/Fail
1,a - 8 Boards Stacked above one another	N/A	N/A	N/A	Pass
1.a.i - Each Board should have a full set of pieces	2 rooks, 2 knights, 2 bishops, a queen, a king (or commoner), 8 pawns in each colour	N/A	N/A	Pass

1.a.ii – The pieces should be set up in standard formation according to FIDE rules.	Top down the board should look as follows: 	N/A	N/A	
1.a.iii - Each board should have a commoner instead of a king except board 4	Board 4 has a different coloured piece in the 'king square'	N/A	N/A	

4.2 – THE GAME

Objective	Normal Data	Boundary Data	Erroneous Data	Pass/Fail
2.a – There should be a player vs player game mode	Both sides should be controlled by the user	N/A	N/A	
2.b – A pawn that has not moved can move 1 or two squares and one that has moved can only move one square	1- test for pawn that has not moved that it can move one square 2- test for pawn that has not moved that it can move two squares 3 - test for pawn that has moved	N/A	N/A	

	that it can only move one square			
2.c – There should be ai vs ai mode	Both sides should be controller by the AI 1 – see that AI makes moves for both teams 2 – see that user input cannot move pieces	N/A	N/A	
2.d – there should be player vs AI	User input should control one team and AI should control the other. 1 – see that the user can control the moves for one team 2 – see that the AI makes moves for the other team 3 – see that player cannot make moves for the same team that the AI is making moves for	N/A	N/A	
2.e – For 2.a,2.c and 2.d if there is user input allowed it should be able to be for either black or white	For 2.a this means user controls both (tested already) For 2.c it is irrelevant	N/A	N/A	

	For 2.d the user should be able to control white or black			
2.g - a sound should play when a piece is moved	Check that a sound is played when a piece is moved	N/A	N/A	
2.h – there should be background music	Check that throughout the game background music plays	N/A	N/A	
2.i.i – There should be the ability to limit time players can use over the whole of the game	1- test that overall time can be changed 2 – test that time gained after move can be changed 3 - test that when time runs out the player with no time loses	N/A	N/A	
2.j.i – rook should move as according to rules	1 – test I can move any amount of squares up 2 – test it can move any amount of squares along a file 3 – test is can move any amount of	N/A	N/A	

	squares along a rank			
2.j.ii – bishop should move according to rules	<p>1 – test it can move any amount of squares across the same number of files and ranks</p> <p>2 – test that it can move any amount of squares across the same number of files, ranks and levels</p>	N/A	N/A	
2.j.iii – the knight should move according to the rules	1 - it can move 2 squares in one direction and one square in the other two – test for all three permutations	N/A	N/A	
2.j.iv – the queen should move according to the rules	<p>Test that it moves according to the rook and the bishop</p> <p>1 – test it can move any amount of squares across the same number of files and ranks</p> <p>2 – test that it can move any amount of squares across the same number of files, ranks and levels</p>	N/A	N/A	

	<p>3 – test I can move any amount of squares up</p> <p>4 – test it can move any amount of squares along a file</p> <p>5 – test is can move any amount of squares along a rank</p>			
2.j.v – the commoner and king should move one square in each direction	Test that it can move a cube in around itself	N/A	N/A	
2.k.i – pieces cannot move out of the board	N/A	N/A	Try to move piece out of the bounds of the board	
2.k.ii – king may not stay in check	N/A	N/A	Try to move a piece that does not stop the king from being in check whilst in check	
2.k.iii – a piece may not move if it puts the king in check	N/A	N/A	Try to move a piece that creates check on the king	
2.k.iv – pieces of the same team may not occupy the same square	N/A	N/A	Try to move a piece onto a piece of the same team – where the move would usually be legal otherwise	

2.k.v -for all piece except the knight there may not be a piece in the path towards desired square	N/A	N/A	<p>Try moving a piece to square that has a piece in the way where it would be legal if said piece was not in the way.</p> <p>Test for</p> <ol style="list-style-type: none"> 1. Queen 2. Rook 3. Bishop 4. Pawn 5. Commoner 	
2.l – no legal moves left for a team is a loss if the king is in check	Achieve checkmate and see that title screen displays a win for the mating team	N/A	N/A	
2.m – no legal moves left for a team is a draw if the king is not in check	Achieve no legal moves for the enemy team without achieving checkmate and see that title screen displays draw	N/A	N/A	
2.n – when a piece is selected the available moves should be displayed	<p>Test that a legal move to a free square shows green</p> <p>Test that a legal move to where an enemy piece is shows red</p>	N/A	N/A	
2.0 – tested under 4.1 board	N/A	N/A	N/A	N/A

4.3 – TESTING – UI

Objective	Normal Data	Boundary Data	Erroneous Data	Pass/Fail
3.a.1	There should be a menu screen on launch	N/A	N/A	
3.a.i.1.a – settings button	Test that clicking the settings button opens the setting menu	N/A	N/A	
3.a.i.2 – player vs player	1. Clicking on player vs player will open the time page 2. after selecting time controls and pressing start the game starts. 3. game starts with the correct time controls	4.Entering no time starts a game still	N/A	
3.a.i.3 – player vs ai	1. Clicking on player vs ai will open the team, difficulty page 2. after difficulty, and team the game starts 3. game starts with correct controls		5 – entering no difficulty defaults to easiest 6 – entering no team defaults player as white or last chosen	
3.a.i.4 – ai vs ai	1. Clicking on ai vs ai will open the difficulty page 2. after selecting difficulty the game should start 3. game starts with correct controls	N/A	5 – entering no difficulty defaults to easiest	
3.a.i.6 – save game	Play three moves sand save the game	N/A	N/A	

	<p>1 – once save is selected the check that the same three move position is loaded</p> <p>2 - Repeat for different moves</p> <p>3 – check that no time is loaded</p>			
3.b.i – Display moves made	See that when a move is made the move counter increases	See that in a load game the number of moves made	N/A	
3.b.iii – number of taken pieces should be displayed	<p>Test that taking pieces increases the pieces taken counter for each team</p> <p>1 – take a white pawn</p> <p>2 – take a black queen</p>	<p>Test that taken moves stays constant for a loaded game</p> <p>1 - Save the game used for normal data testing</p> <p>2 – Load the game and see the taken pieces show</p>		
3.b.iv – winning team is displayed	Tested in 4.2.m	N/A	N/A	
3.c.i – transparent layers	<p>Test that each button makes its respective layer translucent and the button itself translucent.</p> <p>Test that pieces stay opaque and stationary</p>	N/A	N/A	
3.d.i – settings button	Click settings button and check it opens the settings	N/A	N/A	
3.d.ii – mute buttons	1 – press the sounds effect mute button	N/A	N/A	

	<p>2 – see that moving makes no sounds</p> <p>1 – press the mute background sound button</p> <p>2 – see that the background music stops playing</p>			
3.d.iii – Save game	[Tested in load game]	N/A	N/A	
3.d.iv – return to menu	Start player vs player game. Open settings. Select return to menu and see that the menu opens	N/A	N/A	
3.d.v – Restart button	Start a player vs Ai game. Make a move and wait for ai to move. Open Settings. Choose restart. Make and move and see ai makes a move	N/A	N/A	
3.d.vi – return to game	Start game. Open settings. Press return. See that game continues	N/A	N/A	

4.4 – TESTING – AI

Numbers shown in ' []' represent the objectives tested by each specific test

When I refer to objectives in testing videos I mean test numbers i.e when I say "testing objective 1 for AI 1" I mean that I am testing: (1- blunder a piece and check that the opponent ai takes it – testing [1][2][4][11]). Not that I am testing Objective one from the objective in design.

A note regarding the piece value tests – it is not mentioned in the videos but was brought up by the client as they thought there was a mistake. The best move being displayed by the Debug.Log() statement is taking the moves from white that give no additional theory and therefore we don't have to consider the sorts of development that the player can do following that may cancel out to leave us with smaller values, so this is not a mistake.

Another note regarding the implementation using percentage accuracy. It is briefly touched upon in the videos but to reiterate: the reason that the percentage accuracy implementation works here is due to the large state space. One might argue that if you compare this to a normal Chess Ai that this implementation could never work for the fact that in positions where the obvious move for any level of player is to capture a blundered piece or to recapture a piece you intended to trade that this

implementation will not do as such. This does work for 3D chess however as the computer has a very large advantage being able to see the whole board which the user cannot. The idea behind the percentage is to create a human like handicap for the computer where it cannot perceive the board well and does not have a good understanding of what is happening in every area of the boards.

One final note regarding the testing for Ai-4: I state that test 2 was passed I do not believe this was true however it was passed in the next video.

Objective	Normal Data	Boundary Data	Erroneous Data	Pass/Fail
4.1 – Ai1	<p><i>If blunder occurs retry test – (where ‘most’ and ‘some’ are written) - This will be indicated by a Debug.Log Statement</i></p> <p>1- blunder a piece and check that the opponent ai takes it – testing [1][2][4][11]</p> <p>2 – blunder a piece and check that the ai does not take some of the time – testing [4][6][7]</p> <p>3 – create a position where the ai can take a protected pawn with a higher valued piece and that it does not take – testing [5]</p> <p>4 – play an opening and show that ai does not play according to theory with pieces towards the centre – [12]</p> <p>6 – create position where player can make check the opponent and see that ai stops it some of the time – testing [14]</p> <p>7 – create position where ai can check the player’s king and see that it will then check some of the time – testing [9][8]</p> <p>8 – create a position where you can take the opponents piece</p>	N/A		

	<p>that is hanging and test that the ai protects some of the time – testing [3]</p> <p>9 – show that the AI understands the value of pieces [10]</p>			
4.2 – Ai 2	<p><i>If blunder occurs retry test – (where ‘most’ and ‘some’ are written) - This will be indicated by a Debug.Log Statement</i></p> <p>1- blunder a piece and check that the opponent ai takes it most of the time – testing [1][2][4]</p> <p>2 – blunder a piece and check that the ai does not take some of the time – testing [4][6][7]</p> <p>3 – create a position where the ai can take a protected pawn with a higher valued piece and that it does not take – testing [5]</p> <p>5 – create position where player can make check the opponent and see that ai stops it some of the time – testing [14]</p> <p>6 – create position where ai can check the player’s king and see that it will then check some of the time – testing [9][8]</p> <p>7 – create a position where you can take the opponents piece that is hanging and test that the ai protects most of the time – testing [3]</p>			

	<p>8 – show that the AI understands the value of pieces – [9b]</p> <p>13 – play 15 moves worth of opening phase and see the ai develops mostly new pieces towards the centre – testing [12][10]</p>			
4.3 – Ai 3	<p><i>If blunder occurs retry test – (where 'most' and 'some' are written) - This will be indicated by a Debug.Log Statement</i></p> <p>1- blunder a piece and check that the opponent ai takes it most of the time – testing [1][2][4]</p> <p>2 – blunder a piece and check that the ai does not take some of the time – testing [5]</p> <p>3 – create a position where the ai can take a protected pawn with a higher valued piece and that it does not take – testing [5]</p> <p>5 – create position where player can make check the opponent and see that ai stops it some of the time – testing [14]</p> <p>6 – create position where ai can check the player's king and see that it will then check some of the time – testing [9][8]</p> <p>7 – create a position where you can take the oponents piece that is hanging and test that the ai protects most of the time – testing [3]</p>			

	<p>8 – show that the AI understands the value of pieces – [9b]</p> <p>13 – play 15 moves worth of opening phase and see the ai develops mostly new pieces towards the centre – testing [9a]</p> <p>16 – test that some of the time the ai will make intelligent moves that require it look multiple moves ahead– testing [4a]</p> <p>17 – in a position where very little major pieces are left check the ai tries to move its king towards pawns of either team – testing [9c]</p> <p>18 – test that in a position where the ai is up material it will try to trade pieces if they can – testing [10]</p> <p>19 – test that in a position where the ai is down material it will not try to trade pieces if they can – testing [10]</p> <p>20 – test that if a rook can move to an open file safely (and it cannot obtain material in another way) the ai will chose to do so – testing [8c]</p> <p>21 – test that in a position where the ai should move it's knight it will decide to not move it to the edge of the board most of the time – testing [8b]</p> <p>22 – test that in a position where the ai will make doubled pawns or isolate a pawn it</p>		
--	--	--	--

	chooses not to most of the time – testing [8a]			
4.4 – Ai 4	<p><i>If blunder occurs retry test – (where 'most' and 'some' are written) - This will be indicated by a Debug.Log Statement</i></p> <p>1- blunder a piece and check that the opponent ai takes it most of the time – testing [1][2][4]</p> <p>2 – see that somewhere during testing the opponent either makes a blunder (test is set out like this due to the low blunder rate of the AI) – testing [5]</p> <p>3 – create a position where the ai can take a protected pawn with a higher valued piece and that it does not take – testing [5]</p> <p>5 – create position where player can make check the opponent and see that ai stops it some of the time – testing [14]</p> <p>6 – create position where ai can check the player's king and see that it will then check some of the time – testing [9][8]</p> <p>7 – create a position where you can take the opponents piece that is hanging and test that the ai protects most of the time – testing [3]</p> <p>8 – show that the AI understands the value of pieces – [9b]</p> <p>13 – play 15 moves worth of opening phase and see the the</p>			

	<p>ai develops mostly new pieces towards the centre – testing [9a]</p> <p>16 – test that some of the time the ai will make intelligent moves that require it look multiple moves ahead– testing [4a]</p> <p>17 – in a position where very little major pieces are left check the ai tries to move its king towards pawns of either team – testing [9c]</p> <p>18 – test that in a position where the ai is up material it will try to trade pieces if they can – testing [10]</p> <p>19 - test that in a position where the ai is down material it will not try to trade pieces if they can – testing [10]</p> <p>20 – test that a position where one of the following can occur: the ai tries to undergo the move if it is positive and tries to stop the move if it is negative (most of the time):</p> <p>Pawns protecting each other. +0.1</p> <p>Isolated pawns -0.1</p> <p>Doubled pawns -0.2</p> <p>Knight on edge of board -0.2</p> <p>Knight with all moves in bound of the board +0.2</p> <p>Bishop pair +0.3</p> <p>Bishop on same line as queen +0.1</p>		
--	--	--	--

	<p>Rook on open file +0.3</p> <p>Any piece off starting square (<i>tested under point 13 – except from checking moves and capturing moves all moves were off starting square move – check video one for evidence</i>) +0.1</p> <p>Any piece attacking a square that touches the king (<i>tested in first video – can be seen with the various check moves and especially the queen moves which stops the king from moving to desirable squares (out of the way of more checks)</i>) +0.4</p> <p>Any piece achieving a pin on king +0.1*the value of the pinned piece</p> <p>Any piece achieving pin on the queen +0.4</p> <p>Any piece controlling or on the centre 4 squares of any level (<i>tested under point 13 – except from checking moves and capturing moves all moves were pawn moves to the centre – check video one for evidence</i>) +0.2</p>			
4.5 – Ai 5	<p>1- blunder a piece and check that the opponent ai takes it all of the time – testing [1][2][4]</p> <p>3 – create a position where the ai can take a protected pawn with a higher valued piece and that it does not take – testing [5]</p> <p>5 – create position where player can make checkmate threat on the opponent and see</p>			

	<p>that ai stops it all of the time – testing [14]</p> <p>6 – create position where ai can check the player's king and see that it will then check all of the time where there is not a better move – testing [9][8]</p> <p>7 – create a position where you can take the oponents piece that is hanging and test that the ai protects (or offers a trade) all of the time – testing [3]</p> <p>8 – show that the AI understands the value of pieces – [9b]</p> <p>13 – play 15 moves worth of opening phase and see the the ai develops ally new pieces towards the centre – testing [9a]</p> <p>16 – test that some of the time the ai will make intelligent moves that require it look multiple moves ahead (this is done by create a position where the ai can make two moves in a row where the second move gives increased material) – testing [4a]</p> <p>17 – in a position where very little major pieces are left check the ai tries to move its king towards pawns of either team – testing [9c]</p> <p>18 – test that in a position where the ai is up material it will try to trade pieces if they can – testing [10]</p>		
--	---	--	--

	<p>19 - test that in a position where the ai is down material it will not try to trade pieces if they can – testing [10]</p> <p>20 – test that a position where one of the following can occur: the ai tries to undergo the move if it is positive and tries to stop the move if it is negative (all of the time):</p> <p>Pawns protecting each other. +0.1</p> <p>Isolated pawns -0.1</p> <p>Doubled pawns -0.2</p> <p>Knight on edge of board -0.2</p> <p>Knight with all moves in bound of the board +0.2</p> <p>Bishop pair +0.3</p> <p>Bishop on same line as queen +0.1</p> <p>Rook on open file +0.3</p> <p>Any piece off starting square (<i>tested under point 13 – except from checking moves and capturing moves all moves were off starting square move – check video one for evidence</i>) +0.1</p> <p>Any piece attacking a square that touches the king (<i>tested in first video – can been seen with the various check moves and especially the queen moves which stops the king from moving to desirable squares (out of the way of more checks)</i>) +0.4</p>		
--	--	--	--

	<p>Any piece achieving a pin on king +0.1*the value of the pinned piece</p> <p>Any piece achieving pin on the queen +0.4</p> <p>Any piece controlling or on the centre 4 squares of any level <i>(tested under point 13 – except from checking moves and capturing moves all moves were pawn moves to the centre – check video one for evidence)</i> +0.2</p>			
--	---	--	--	--

5.0 EVALUATION

5.1.1 – OBJECTIVES – COMPLETENESS

From the testing we can see that all the objectives were met, however, the testing regarding the AI was difficult to set out and conduct due to the nature of how AI needed to be created. Though it seems that the AI is working how I wish it to work and it under/outperforms myself as necessary there is no conclusive way to measure such objectives or a way to make the objectives in a way that allows conclusive testing.

5.1.2 – OBJECTIVES – HOW IT WAS ACHIEVED

The objectives were achieved through many hours of play and subsequent note making. Much of the thought process can be seen through the end of analysis and start of the design section. The AI was also made realistic (regarding the level of human player they are trying to imitate) by the same process of extensive play and note taking. The thought process can again be seen in the later stages of analysis and throughout design.

The implementation of the other parts was done in small stages using the structure chart shown in design to get the backbone of the project covered and filling in the specifics afterwards. The complex sections of code were tackled by generating flow charts and converting these into code. For the neural network specifically some research into multivariable calculus and the mathematics was needed to understand and code.

5.2 – IMPROVEMENT

Given more time there could have been some improvements. The neural network has had extensive training and has shown that it can pick up well on the patterns I had hoped. It does not learn fast however, due to its large nature (7168, 3584, 512, 64, 1). For this reason, the network most likely has a long time left of learning and given more time I could have attempted to make learning faster or train it for longer to achieve better results of the later AIs. This would future proof the application I have made, keeping the top AI very strong even though it does meet the current requirements.

Given more time some of the features from chess.com that are impractical because of the time needed to create such things are too time consuming. For example, the puzzles and lessons could have been made if had more time: the puzzles could have been made with an algorithm that works from checkmate backwards and uses the

Als to classify puzzle difficulty (this however would use up computational power that was being used to create training data and train the network). The lessons would require me to play much more of this game and then film the lessons.

5.3.1 – USER FEEDBACK – RELATING TO 1ST AND 2ND INTERVIEWS

Below are reminders of relevant original questions asked to the user and follow up questions asked after he played the final product.

[Old Questions in this colour]

Q: With regards to the UI, what would you be expecting?

A: I would like the board to be able to be split up so that certain or individual layers can be seen separately. I would also like to be able to rotate the board and see which pieces have been captured. Additionally, a feature to see a piece's available moves would be nice.

Takeaway: Being able to visualise your moves is very important and as such having an easy to use, understand and visualise playing field should be a priority.

Q: How do you feel with the UI and how it relates to what you were expecting?

A: I felt like the movement around the board was very easy to use and made the experience enjoyable. Being able to make the board transparent was surprising very helpful when considering tactics during the middle game. Displaying taken pieces on the side was good and important for knowing when to trade down and not, especially later in the game when things were harder to keep track of. My favourite UI feature by far was the ability to see where pieces can move and capture. For humans it makes it much more fun and easier to play a competitive game and reduces the stress I felt about keeping track of danger levels (*a side note from me – danger levels are where pieces are to do with how many pieces are attacking how many other pieces*).

Takeaway: It seems that the user is very happy with the outcome of the UI and has no complaints – this corresponds with what the completeness of the objectives suggest.

Q: What would you expect from the AI?

A: There should be different levels of AI with one that will lose to an average intelligence beginner but also one that would beat an experienced player. I would like the AI to be able to play as close to as a human would as possible.

Takeaway: A varying and competitive AI is important for play to be intriguing but also the AI should behave human-like to provide more realistic play.

Q: How do you feel with the AI and how it relates to what you were expecting?

A: Overall the variation in the AI is amazing. There's much more variety than I expected. I'm quite glad that I struggle with the 3rd AI as this means that there is huge headroom above me but also that the easy Als are accessible for beginners. Having played all 5 they did feel comparable to humans, of course due to the nature of probability it felt like sometimes the lower Als played a sequence of too good moves and the 4th and 3rd Als would make a couple horrendous blunders in a row. Overall, I wouldn't expect

humans to play much differently to AIs 1-4. It was good to see the positional play from AI 4. The jump from AI4 to 5 was extreme with AI5 dominating me the entire time. Overall, the AI was much better than anticipated.

Takeaway: It seems that the user is very happy with the outcome once again. There are more points to discuss here however, it seems that there could have been an AI between AI4 and 5 to cover this large gap. The neural network seems to be performing well if an experienced chess player can commend and its positional play.

Q: What time restrictions on time would you like to see?

A: Just as with FIDE chess I think there should be an untimed, standard, blitz and bullet options, however, at this time I am not sure what these exact times at this moment.

Takeaway: There should be multiplied time modes, however, until the game is played the specifics are uncertain.

Q: How do you feel about the fully variable time system

A: It is much better for users to find out what time is best for them than with set time rules that only apply to us. Personally I like 120m 1m inc.

Takeaway: I feel the same as the user with regard to other people playing this game. This is a success that meets the testing outcome.

Q: What game modes would you like?

A: Player vs AI, player vs player. Perhaps an AI vs AI to learn scenarios.

Takeaway: Having variation in what can be done inside the game is important for the user.

Q: How do you feel about the game modes?

A: They are all there and working so fine.

Takeaway: User's views align with testing.

Q: How would you like games to be recorded/ displayed back to you?

A: Moves should be displayed on a sidebar as they happen.

Takeaway: A move sidebar feature

Q: How do you feel about the sidebar?

A: It is good. The scrolling feature is a nice touch that I appreciate.

Takeaway: User's views align with testing.

Q: How would you like this game to be available I.e. on a large store front, a website?

A: Preferably as a computer app as I feel a website would be too slow but would be fine if it is not slower. Perhaps also a phone application.

Takeaway: The user seems to be happy with most forms that it could be available in, however, they seem keen to play it on a computer.

Q: Having played the game what do you think about how it should be available?

A: I think that computer only, as it is currently fine. It feels like it would be too finicky to play on a small handheld device.

Takeaway: User's views align with testing

Q: What hardware/software restrictions may affect the making of this project?

A: I think that this program should be able to be ran on a mid-range system (this may affect the AI). Like a phone.

Takeaway: The AI nor UI can be so complex that an average device will struggle to compute the game.

Q: Having played run the game on your computer how was the experience?

A: The AI took longer to move than when playing on your PC (*the device used for testing*) but still slower than stockfish 15 (*chess.com's top engine*) say. So pretty good.

Takeaway: The AI will take much longer to move on the baseline system stated in the design section, however it is still functional and faster than some normal Chess AIs (that are much more complex than my AIs however).

Q: What additional features above gameplay with an AI would you be expecting?

A: Player on player gameplay and AI vs AI gameply. I have no desire to play other people on a WAN as the games take too long and that is too much time commitment. Sounds would be good too.

Takeaway: The user just wants a way to play against other humans too on the same device. There should be piece move sound effects and background music.

Q: How did you like the ai vs ai and sound effects?

A: The ai moved too fast for me to keep track of real time but using save game to see what they were doing, and the scrollable list made watching what they were doing okay. The sound effects were good, the background music was obnoxious but I simply mute it.

Takeaway: A break between AI moves may be good for the AI vs AI gamemode. The mute button for the background music was a good idea. Maybe variety in background music choice would be something to consider.

Q: What/if any two player capabilities would you like?

A: Just standard play with different time limits

Takeaway: No extras needed.

Takeaway: Already covered.

Q: How/would you change the overall time of the game?

A: Different time modes as mentioned before.

Takeaway: Time is an issue and I need to set objectives that allow for shorter gameplay.

Takeaway: Already covered.

Q: Do you think there should be a limitation on thinking time for the human?

A: Yes, of varying amounts of the user's choice.

Takeaway: There should be an ability to create different time mode modes with different starting times and added time.

Takeaway: Already covered.

Q: Do you think there should be a limitation on calculation time for the computer?

A: There should be different levels of AI that do fewer calculations.

Takeaway: Having varying AI levels is important not only for fun gameplay but also for competitive gameplay. (Quick calculating AIs and more complex AI)

Takeaway: Already covered under first AI question.

Q: How strong should the AI be and what should it consider?

A: It should be strongly capable of beating the best of players but with abilities to reduce its own ability. It should prioritise piece taking as opposed to board position – the exception to this is where the piece will be taken by a lower value piece if moved to a location. Try to defend or move high value pieces

Takeaway: A strong AI is important to the user but equally important is one that can vary in ability. After understanding the core principles of the game, both I and the user have discovered that for 3D chess taking pieces early is essential for a game to conclude past as board position matter little compared to FIDE chess till near the end game. As such, this should be factored into the AI.

Takeaway: Already covered under first AI question.

Q: How/would you change the end game and/or how the game can be concluded?

A: Time runout with piece points determining the winner or classic timed play.

Takeaway: This would allow for competitive gameplay with more positioning thinking without the dread of taking the game on for several hours.

Takeaway: This was something the user discussed with me later during the design process when playing a late prototype. We decided not to change the rules from normal chess and keep the time pressure as it is part of the fun.

Q: Are you happy with how the pieces move?

A: Yes, the 3d capabilities make it much better than Strada.

Takeaway: The rules should not be changed.

Q: How do you feel about the new piece rules after lots of play?

A: The piece moves feel good and of course coincide with the new rules. It feels like the classic pieces' rules converted very well and it feels natural the way all the pieces move. It takes some getting used to but I like it.

Takeaway: User's views align with testing
