A Machine Learning Approach to Building a Chess Engine

Daniel Cloutier Sagar Patel

December 15, 2020

Contents

Abstract

In the current era, high volumes of data are being collected at an incredible velocity. Much of this data is embedded with valuable knowledge. [3] One example of the types of data that is collected is chess data. In fact, lichess.org, one of many websites where chess can be played online against opponents from around the world, contains a data dump of over one and a half billion games, seventy-eight million of them being played in November 2020 alone [10]. We intend to present an automatic learning method for chess, utilizing deep neural networks. We created our deep neural network using a combination of both unsupervised and supervised training. Employing unsupervised training, our engine pre-trains to identify and extract high-level features that exist in a dataset of chess board positions. The supervised training compares two chess positions and evaluates the more favourable board, from the white player's perspective. Therefore, we create a deep neural network that can understand chess without incorporating the rules of the game and using no prior manually extracted features. Instead, the system is trained from end to end on a large dataset of chess positions. The resulting deep neural network should allow for simulations that extend the possibilities of chess strategies, highly valuable to chess players at all levels.

1 Introduction

The game of chess is a very popular board game and has become a well-liked study for computer enthusiasts, especially in the domain of Artificial Intelligence, from the creation of Deep Blue in 1996 which defeated the world chess champion at the time, Garry Kasparov, up to modern data analysis techniques being tested on chess games as a show for it is possible uses [12][13][15]. Chess has become a good starting point for both more advanced software developers attempting to test something innovative or for newcomers trying to increase their knowledge.

One of the many reasons for the game's popularity is that chess data is collected at an incredibly high volume and velocity. Not only this, but the data collected is often readily available to the public. Databases such as chessbase and lichess contain thousands, if

not millions of games in a standardized notation format for chess called Portable Game Notation. This makes the game of chess a perfect starting point for new tools in Data Science to be tested.

Another of the many domains of study that has been recently using chess as a means of improving itself is that of Machine Learning. Namely, with the advent of algorithms such as AlphaZero [15], some new ideas have risen in an attempt to create algorithms that not only play chess better than humans, but even some that try and mimic human play [12].

Keeping all of this in mind, we wanted to take advantage of the very large amount of data stored by lichess in order to implement and perhaps try and reinvent some already existing ideas using this data that, as far as we could tell, has not been used much in recently published material.

2 Review of Research and Ideas

2.1 Chess Game Categorization

A possibility for using this data is to attempt to categorize chess data into different groups based on some sort of requirements that we could find. For example, perhaps we could group games based on similar openings [11], similar player ratings or perhaps try to find some different parameters to use to categorize them. One example comes by defining a distance between two games.

One such way is defined as follows. Defining a nine-dimensional space, consisting of the x-y location of a piece before it has been moved as well as after, a weight for the piece before it was moved as well as after (because of piece promotion) as well as the piece that was captured, as well as x-y locations for the piece that was captured. From here you can compare moves between games and calculate the distance between them to figure out what games are similar and what games are outliers [3]. This information could be easily stored although incredibly voluminous. Every move of the game would be stored as nine separate values. Figures 1 and 2 show how storing this information could look.

Figure 1: How storing the above information would look like

Move Number	Piece Before Move	Piece After Move	Piece Captured	Moved From	Moved To	Captured At
1	P	P	NULL	e2	e4	e4
2	P	P	NULL	c7	c5	C5
odd#	NULL	NULL	NULL	e1	e1	e1
even#	NULL	NULL	NULL	e8	e8	e8
11	K	K	NULL	e1	g1	g1
11	R	R	NULL	h1	f1	f1
13	P	Q	NULL	e7	e8	e8
21	P	P	P	d5	c6	c5
22	P	P	P	b7	c6	c6

Figure 2: Sample weight values for pieces

	K	Q	R	B	N	P	NULL
w()	12	9	5	3	2	1	0

From here on we can define the distance between two moves as a euclidean distance between two moves defined by this nine-dimensional space. To extend this further, we can define the distance between two whole games as a sum of the distance between corresponding moves. This would be defined as

$$dist(F, G) = \sum_{i=1}^{M} dist(F[i], G[i]) \tag{1}$$

where F[i] is the i-th move of game F, G[i] being the i-th move of game G and M is the greater of the maximum number of moves from either game [3]. Appending a NULL move as necessary.

This kind of categorization could be very useful for new and old players alike. To be able to choose a game and immediately be given results for similar games is a very powerful learning tool. There are often patterns that can emerge from chess positions which can likely be spotted by a thorough study of similar positions or games. Also, for an advanced player, studying games that lie outside of the norm can be a good way to surprise unexpecting opponents with new ideas.

The above is in fact a type of unsupervised machine learning. But what we can also do by using data mining techniques is create a supervised pattern matching algorithm to classify similar positions [3][17]. Once again, this idea can help categorize games into buckets which can then be looked at if there are certain themes that you know of which

appear in a certain group. Seeing other similar games could grant some insight into those types of positions, whether you want to learn the basics of them or want some fresh ideas to improve your play.

2.2 Predicting Game Results

Following from the previous idea, we could try to use the data in an attempt to predict the result of a game. The lichess database contains a lot of information that others might not. Based on our research, most uses of chess data was limited almost exclusively to Grandmaster games, in a specific time control, all as defined in the FIDE rulebook [6]. Lichess contains players of all levels, games are played in all sorts of time controls including fast one-minute "bullet" games, all the way up to thirty-minute slow games [10]. Using all of this information, along with what was defined previously and the fact that we know the result of every game in the database, we could try and define some way to predict whether a current game of chess, given the player ratings, other similar games, time control and so on, is going to be a win for white, black or a draw.

For example, say we could plot all games according to their similarity distance as defined above [3]. However, if we were to include things like player rating, opening used, time control and so on, we could plot games on multiple dimensions. From here, by looking at games similar to an input game and looking at their results, we could attempt to predict what the result of that game will be in its current state.

There's even more than we can do still. The lichess database starting April 2017 also includes the exact clock times on every single move played. Keeping this in mind means we can improve the above idea further. Some of these could be including the current amount of time each player has, trying to improve the way we define how similar two games are or even by defining a function for determining whether a board position is winning or losing from the white player's perspective assuming best play, not unlike what modern chess computers do [16].

Using the above information we could try and create an algorithm that would predict whether a game is winning or losing by looking at not an only perfect play, but also

factoring in things like time trouble and player strength. For a spectator or someone doing analysis, this could be much more insightful as to the dynamics of a position when looking at a game between two specific players instead of looking at a number that assumes the best play from both sides.

2.3 Creating a Chess Engine

Moving on with this idea raises the main point for this project. Could we use all of the previous information to create an algorithm that could play the game of chess at a human or superhuman level? This appears to be a common theme in the field of Data Science; how to use data to improve the way we look at a certain field. We could attempt to create our own heuristics for what is important and what is not for trying to consider a certain move, however, we decided on a different idea.

With a lot of superhuman chess engines like Komodo, Houdini and Stockfish, they have evaluation functions that are based on hard-coded values for things like pieces and piece placement, as well as the stage of the game. With this in mind, we considered using something similar along with the rest of the data that we had, but try and make the piece values more accurate via some other resources like regression analysis [13]. This paired with the rest of our data would hopefully allow us to more easily change the difficulty of our chess engine.

Keeping this in mind with the rest of our information, we considered some sort of artificial intelligence that could learn on its own instead of us giving the parameters. We are not chess professionals, but there are indeed ways of getting a computer to learn, especially with large volumes of data like we have at hand [1][7]. This would mean we could generate something useful despite the lack of knowledge in the field. Based on our research, this theme comes up often in Data Science, especially when it comes to knowledge discovery. If we give it the parameters that determine whether a position is winning or not, for the computer to make its move, it will only reflect our knowledge of the game. Of course, seeing as we are not expert chess players, this is not what we want. If we want to learn something new by creating this chess engine, we believed the best

approach would be to let the computer use this data to learn on its own.

3 Problem Statement

Now that we know what we want to do, how exactly do we approach this problem? Surely we are not the only ones that have thought of creating a chess engine that learns on its own. This of course is indeed the case. There have been some attempts at making a self-learning chess engine, however, these attempts mainly focus on learning from self-play; the computer knows the rules of chess exclusively and by playing against itself millions of times can learn on its own what wins and what does not [8][15]. This is not necessarily what we want. We already have the data we need, it is now just a matter of getting the computer to learn from it.

From this, we can form a formal problem definition: How do we use modern Data Science techniques to get a computer to learn how to play chess at a human or superhuman level? We will be looking primarily at creating a computer that can play chess in a more humanlike way.

4 Defining Tools for our Approach

4.1 Artificial Intelligence and Machine Learning

Using this problem definition we can start talking about what exactly it means for a computer to learn chess. According to the Oxford Dictionary, this falls into the definition of Artificial Intelligence. Essentially, our approach will be attempting to make a computer think for itself. Although it still is not in a very humanlike way; indeed, the computer is not thinking by our definition, it is merely doing math in the background and determining an answer based on that [1].

But of course, Artificial Intelligence covers many types of problems. This one in particular that we are trying to tackle falls very well into the more narrow scope of Machine Learning, which is more about writing code to make a computer learn [2]. This

is precisely what we want. But this means we need to learn more about Machine Learning.

There are various types of Machine Learning [1]. However what we want to do lends itself better to two specific ones, namely supervised and unsupervised machine learning. When talking about chess game categorization we already mentioned these terms with respect to how the computer categorized the items. Our approach is not as different as it may seem.

4.2 Supervised Learning

Now, what exactly do we mean by supervised machine learning. Fortunately, this term is quite literal. Supervised machine learning is a term used when we are talking about a computer algorithm that maps inputs to already known outputs [1]. Essentially, we define a set of inputs with mapped outputs that we know already. From here, we can define some algorithm by which the computer can learn on its own how to properly map each input to each corresponding output. Every time it gets an answer wrong, you warn it and it attempts to change the way it maps the inputs and if it gets an answer correctly it merely stays the way it is. In a sense you are supervising the computer as it learns, seeing as you already know the answer. What you are not doing, however, is defining *how* the computer reaches its answer. It does that on its own.

To give an example, this is not too dissimilar to how a child learns how to identify different everyday objects. The parent already knows the words (outputs) to all objects (inputs). What the parent does is supervise their child as they explore and try to label different objects with words. If the child correctly identifies an item in the house, they are rewarded and if they get it incorrectly, they are told what the object is called and hopefully they get it correctly next time. If you repeat this many times over, eventually the child learns how to identify every single object in the house.

In our case, we already know the result of every single game in the database we are using. This means that by giving each game a label for which side wins, we can attempt getting the computer to look at a specific position in the game and trying to tell whether it's winning for someone or a draw. Show the computer enough positions and it should

be able to look at any game and do the same.

4.3 Unsupervised Learning

Now we move on to a different kind of Machine Learning, namely the unsupervised kind. Thankfully after defining supervised learning, this one becomes a bit easier as it is quite literally a sort of the opposite. Where in supervised learning, you ensure that the computer correctly maps the inputs to the corresponding outputs, in an unsupervised Machine Learning approach, you do not know what the output is. Instead, the computer models the inputs on its own [1].

This is something we want to consider when looking at our data. Seeing as we want to use our approach in an attempt to discover something new, it would be helpful to take an unsupervised approach at some point during our implementation. Essentially, taking our data and getting the computer to model them as an output on its own could help us discover something new.

4.4 Neural Networks

Neural Networks are a very solid approach for us to accomplish everything we want. A Neural Network is fairly straightforward, you give it some content and you get an identification back. More specifically, you have some input nodes, which are just number values, which connect into more layers of nodes and so on, until you reach an output layer of nodes that correspond to different labels [18]. This is essentially perfect for us, especially in the context of supervised Machine Learning. We can model a network that, when trained, will learn how to identify whether a certain position is winning or losing from a certain perspective. This can be either using all of the parameters we have discussed or using only the information about a specific position, excluding factors like time control or player strength. This would likely be closer to an attempt at making the computer play strictly better, rather than more human.

Using Neural Networks, we can do quite a lot in fact. On top of the above, we can also train it to try and extract only important information about a certain board state [4], or

other parameters we give it. This would be done by using what is called an autoencoder.

In essence, an autoencoder is a type of unsupervised learning. Instead of the Neural Network trying to identify which label a certain input belongs to, we instead get the neural network to try and recreate the input as its output [9]. But how is that related to extracting important information?

Well, when you structure an autoencoder, you can create the network in such a way that you have fewer and fewer nodes as you go through, before finally re-expanding back up to the original size of the input. What this means is that all of the information from the input can still be extracted from fewer input nodes [4]. In essence, what you've done is compressed the original input down to a smaller size while still keeping enough important information to be able to extract the original input from it. So what you would do is train the network to encode, then decode the input, but only use the part the encodes.

If we wanted to reduce the size of our input layer, this is a great way of doing it. If we had as an example, two hundred inputs, we could train an autoencoder to reduce it to say, one hundred inputs, to then feed that into something that could try and label what we want.

4.5 Chess Data Structure

All of the above is fine, but now we need to determine what exactly our inputs will look like. We already know the objective; getting the computer to play moves by looking at a board state and determining whether it is a winning position or not, but how will we accomplish this. To accomplish this, we'll need to take a look at chess notation, namely Standard Algebraic Notation (SAN) Portable Game Notation (PGN) and Forsyth-Edwards Notation (FEN) [5].

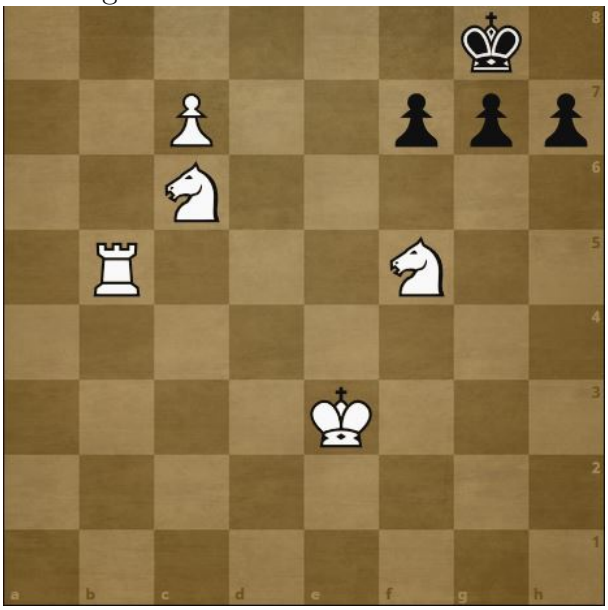
First off, let's explain SAN. What SAN allows a player to do is easily model any move that is played in a very simple format. If you name each column (also called "file") from left-right from the white players perspective with the letters a-h and all rows (also called "rank") from forward to back from the white players perspective with the numbers 1-8, you can mark each square with a letter-number pair (i.e. a4, for the leftmost square,

4 tiles away from the white players perspective). Each move is formatted as a series of characters, all in ASCII, as follows:

1. Piece moved, denoted as [K]ing, [Q]ueen, k[N]ight, [R]ook, [B]ishop or nothing if a pawn
2. Letter or number of original piece location if the same type of piece can make it to the same square
3. x if the move captures another piece
4. Location of the square the piece arrives at
5. =(piece) if the move is a pawn promotion, where (piece) is the piece it was promoted to
6. + if the move is check, # if the move is checkmate

This all seems complicated but allows you to accurately describe every possible move a player can make using a maximum of 7 characters. Let's look at the position in figure 3 as an example.

Figure 3: Random Chess Position



Some possible moves for the white player would be: Nfe7+, Nce7+, Rb8#, c8=Q#, c8=B, Nxg7 or Kf3. Notice that to move a knight to e7, you must specify which knight moves there, as it is ambiguous. Using this knowledge we can move on to describing Portable Game Notation (PGN).

Chess games nowadays are written down using PGN. The way PGN is formatted is by listing all of the moves played during a game using SAN, numbered by which turn it is, along with some added metadata like player names, dates, player ratings (ELO) and others. It is also all in ASCII, so it can be easily read by both humans and computer alike [5]. Having this is a powerful tool as we can now simulate every single game in our dataset, as well as capturing metadata like player ratings, opening used, the date the game was played, the time control used and so on. From here, we can determine some way to convert this information into something more easily parsed by a neural network, as in changing everything into a numeric vector that can be passed into the network, as well as formatting what our output vector will have to look like.

Now that we know we can model every possible move, we can move forward with modelling a board state using FEN. What FEN does is it encodes a board state using a single line of characters in the format: (board) (turn) (castling) (en-passant square) (half move clock) (full move number) [5]. One example would be something like this: rnbqkbnr/pppppppp/8/8/4P3/8/PPPP1PPP/RNBQKBNR b KQkq e3 0 1 which is the board after the single move 1. e4 is played. Using this, we can model every possible board state including whether a pawn can be taken by en-passant or not, whether players have castling rights as well as half-move clocks for determining whether the game is a draw. This format can easily be converted into a vector of integers that can be passed into a neural network for training, especially if paired with an output vector of whether that board resulted in a win or loss for the white player.

4.6 Search Trees

Seeing as we would like the create a chess engine, simply getting our neural network to predict whether a move is winning or not is not enough. Therefore, we must find some way

for our approach to also generate possible move candidates and determine which is the most viable option for a win, using its predictions. This means we will have to implement some sort of search tree algorithm. We can do it in the following way: modelling every possible move we can play, checking what the computer thinks is the best possible move and playing that one.

What we will be doing is implementing a version of the alpha-beta search. What this does is it generates every possible move from a given position and gives each node a value, often based on some function determining a value for the resulting position [14]. In our case, this will be the value our neural network generates when we pass positions into it. In our case, the white player will want to maximize the value and the black player will want to minimize it, in essence assuming best play from either player.

From here, to speed up our search, we can attempt pruning branches. This is done by always keeping the current maximal value (called alpha) and the current minimal value (called beta). When you reach a node where the white player is making a move, whose value is less than the current alpha, there is no point in searching further down this branch, as it is already worse than what we currently consider to be the best move. The reverse is done for the black player with the beta value [4][14]. Using this will cut off some branches of the tree, effectively reducing the number of branches we need to check. We can set this up for a certain depth of search as well.

What we will be doing is slightly different, but follows the same concept. We will be keeping alpha and beta as an entire board position, rather than a single value. Then when comparing, instead of using the neural network to grant a value to a position, we will instead pass into it two positions, from which it will decide the board state it considers to be better, from the white player's perspective [4].

5 Our Model

5.1 Inspiration

The primary inspiration for this work is that of the Deep Chess paper. Essentially, we will be creating a Neural Network whose structure is heavily inspired by the one described in this paper. We also take some inspiration from Maia Chess in terms of the goal of our project, as well as some from Leela Chess Zero and AlphaZero in the fact that they are also Neural Networks designed to play chess, though their goals are slightly different.

What we will be doing is parsing the chess data from the lichess database, which contains many millions of games, filtering games and positions we do not want, converting positions into input vectors as well as an expected output for which board is considered better, from the white players perspective. When the network is trained, we can then create an alpha-beta search that will use our network to determine which branches of the tree to keep looking at and which ones to discard when considering different possible moves.

5.2 Parsing Chess Data

The first thing we will have to do is parse the data from the lichess database to extract what we want from it. Seeing as our dataset is incredibly expansive and ranges from beginner players to Grandmaster level players from time controls as fast as one minute all the way up to slow games lasting over an hour, this means we're going to want to reduce the amount of variability in the data, especially since our network will only be taking in positions as inputs. Players playing at different time controls and different skill levels will heavily increase the variety of the data, which may make it hard for the algorithm to predict whether a position is favourable or not, especially as the other data will not be given to it. As such, we will be only considering specific board states, similarly to Deep Chess, along with some other data filtering which we have chosen to try and shrink our dataset, considering our lack of computational resources.

Because of this, we have made some limits. We will be limiting our data to:

1. Only games which ended decisively, as in no drawn games will be taken into our dataset.

2. Only games where both players have rated at least 1200 points in the lichess ranking system.

3. Only games where each player has a minimum of twenty minutes to make all of their moves, excluding time increment.

4. Only games which lasted at least ten moves.

We will also be picking our games at random from the lichess database. As it is partitioned by month, our program will select a random order for the months and start from the top, considering all of our conditions for what will be a valid game or not. When it finds a valid game, it will select a random move between move ten and the end of the game, play the game up to that point and store the resulting position as well as the result of the game and the player ratings. We do this until we reach ten thousand positions in which white won the game and ten thousand positions in which black won the game, for a total of twenty thousand games. This means that when we go to train our network, which will compare two different boards, one where white wins and one where black wins, we will have a total of $10^4 * 10^4 = 10^8$ or one hundred million different possibilities. This should be more than enough data to adequately train our network.

The above means our program will have to be able to parse through the PGN in the lichess database and convert it into an input vector. This input vector, we will define as sixty-nine different values, sixty-four for the different tiles of the board, four of them bits for castling rights and the last a bit for determining whose move it is. Each value for the board will contain a value between negative six and positive six, negative values for black pieces, positives values for white pieces and a zero for an empty tile. Although our autoencoder may suffer from the lack of total positions, we also lack the resources to recompute the PGN $\rightarrow$ Input Vector computation every time we want to train, as well as the resources to merely store many hundreds of thousands or even millions of board positions that we could generate from this database.

We will be parsing the PGN and converting each move into a new board position, beginning with the starting position for chess, all the way up to the final move that we have randomly selected from the selected game. The resulting position will then be parsed and converted into the input vector as described above. Note that this is different than the input vector described in Deep Chess. They used a bit string 773 characters long as their input vector. This is called a bit-board representation [4]. We do not use this representation due to a lack of computational resources. A network that large would likely take a significantly larger amount of resources to train than we currently have access to.

5.3 Autoencoder

Before we carry on with the network which compares two board positions, we must first talk about the autoencoder that we're going to be using. In the Deep Chess paper, they describe an autoencoder which has an input layer of 773 nodes, leading into hidden layers of 600 nodes $\rightarrow$ 400 nodes $\rightarrow$ 200 nodes $\rightarrow$ 100 nodes $\rightarrow$ 200 nodes $\rightarrow$ 400 nodes $\rightarrow$ 600 nodes with an output layer of 773 nodes [4]. We will be drawing inspiration from this structure with each layer using a ReLU activation function.

Our network will consist of input and output layers of 69 nodes, with hidden layers 40 nodes $\rightarrow$ 20 nodes $\rightarrow$ 10 nodes $\rightarrow$ 20 nodes $\rightarrow$ 40 nodes. We will be training this over our data set of twenty thousand board positions and using only the section ending with ten nodes as the encoding part of our deep neural network that will compare two board states.

This network will then be trained from this state with random samples of our training data to hopefully be able to regather the original input as its output. From then on, we will set up just the encoding layers and create two copies for the chess evaluator network which we discuss in the next section.

5.4 Chess Network

Using our implementation of earlier discussed Autocoders, we can convert given chess positions into a vector of values that represent the high-level features of that given position. In the Deep Chess paper, the researchers break up their neural network into two major components, the Deep Belief Network and Deep Chess. The Deep Belief Network is based on stacked autoencoders that are trained using layer-wise unsupervised training. The autoencoders discussed above are used to create 5 layers of nodes that begin with an input of 773 nodes and are encoded to 100 nodes [4]. Our structure is inspired by this network however, our input data consist of 69 nodes and is encoded down to 10 nodes. A major difference between both of our Deep Belief Networks that is the paper chooses to fix their weights after each autoencoder is trained and becomes the training weights of the next autoencoder. For instance, the weights acquired by training the autoencoder of 773 nodes to 600 nodes are then used to set the weights of the next autoencoder on the stack (600 nodes $\rightarrow$ 400 nodes $\rightarrow$ 600 nodes). In our approach, we initially trained each layer of our Deep Belief Network and calculated the weights after the complete stack had been trained. The calculated weights are then used as the input for our next major component.

The next component of our Network is also the core component of our method and is called DeepChess. The DeepChess component consists of various layers of stacked autoencoders that enables our network to reach our goal. In the Deep Chess paper, the DeepChess network is created using two disjoint copies of the autoencoder stack that combined creates the Deep Belief Network. This trained Deep Belief Network uses its calculated weights as the initial weights for the supervised network that is created by four fully connected layers. The size of these layers takes an input of 400 nodes that lead into hidden layers of 200 nodes $\rightarrow$ 100 nodes $\rightarrow$ 2 nodes. In our approach we use a very similar structure as described above however, due to our autoencoder consisting less cumulative nodes, we change our DeepChess layers to accept 40 nodes and contains layers of 40 nodes $\rightarrow$ 20 nodes $\rightarrow$ 10 nodes $\rightarrow$ 2 nodes. These layers are also similarly trained using the ReLU activation function.

This allows for the first autoencoder layers that represent our Deep Belief Network to function as high-level feature extractors from its input of a given chess position. Using two disjoint stacks of autoencoder allows us to take an input of two given chess positions and will extract the features of both locations. The combined weights of our Deep Belief Network are then inserted into our next four connected layers. These layers compare the calculated features of both positions and determine which one is the better option.

5.5 Training the Network

Unlike the earlier autoencoders that are trained using an unsupervised method. The DeepChess network using a supervised training method that began by creating 1,000,000 random input pairs. These pairs consist of one position selected from a set of random positions that resulted in a White win and one position also selected by random from a set that resulted in a black win. These pairs can be randomly ordered as such (W, L) or (L, W) and are saved into a large data set. This method allows for our training pairs to contain new distinct pairs of positions with a very low chance of overlapping of the same positions to occur. These positions are then entered into the DeepChess network to identify which positions are more likely to win from the white player's perspective while being checked with the correct solution of each pair.

5.6 Playing Moves

After we have a neural network that can take two positions as an input and give us an answer as to which board it believes is better for the white player, it will be time for us to put it to the test playing games and identifying positions. The way this will be done is using a search method, as discussed above, called alpha-beta pruning search.

In order to accomplish this, we will have to find a way to generate all possible moves from a certain position, generate the resulting board state from each of those moves and compare them to each other. However, keeping alpha-beta pruning in mind, the way we will actually do this is more as described in the Deep Chess paper. Usually, an evaluation function gives a value for the board state, which is then compared to an alpha value,

initialized to negative infinity. Alpha takes the maximum value between itself and the new board evaluation. We do the same thing with a beta value, which is initialized to positive infinity, and takes the minimum when comparing from black's perspective [4]. This approach allows us to ignore moves that are worse than the already known best position and not look further down certain branches more than we need to, optimizing our search to only branches that are promising.

We will be using a slightly different version however, which is described in the Deep Chess paper. Instead of saving the alpha and beta as values, we will instead store the board position instead. This way, we can compare the alpha board state directly to the current move we're looking at and then choose whichever board state is better for white to replace alpha. In beta's case, we do the same thing but look at which board state is worse for white.

6 Implementation

6.1 Python-Chess

While working on trying to parse PGNs we managed to find a very convenient library called python-chess. Essentially, this library very conveniently manages almost every single thing that you would need for chess. It contains boards, pieces, functions for generating moves, functions for generating FENs from PGNs, inputting moves into a board, move stacks and so on.

In order to save us some time, we decided to use this library. Seeing as we needed to generate moves, parse PGNs into FENs and so on, this library was a perfect fit. In fact, when it comes to PGNs, this library when parsing through files with multiple PGNs, can perfect distinguish between different games and even stores the metadata in its own dictionary object that can be indexed. This made generating our dataset, as seen in figure reffig:python-chess, a much simpler task than it would have been otherwise.

On top of this, we had to learn which way we wanted to get our data from the lichess database. Seeing as the lichess database had a list of all their download links, we were

Figure 4: Parsing PGN Data

```
for gamefile in filelist:
    if white_wins >= NUM_WINS and black_wins >= NUM_WINS:
        break

    with bz2.open(urllib.request.urlopen(gamefile), "rt") as reader:
        chessgame = None
        while True:
            if white_wins >= NUM_WINS and black_wins >= NUM_WINS:
                break

            try:
                chessgame = chess.pgn.read_game(reader)
            except EOFError:
                break
            if chessgame == None:
                break

            # Filtering data
            if chessgame.headers['Result'] == '1-0' and white_wins >= NUM_WINS:
                continue
            if chessgame.headers['Result'] == '0-1' and black_wins >= NUM_WINS:
                continue
            if chessgame.headers['Result'] == '1/2-1/2':
                continue
            if not chessgame.headers['WhiteElo'].isnumeric() or not chessgame.headers['BlackElo'].isnumeric():
                continue
            if int(chessgame.headers['WhiteElo']) < 2000 or int(chessgame.headers['BlackElo']) < 2000:
                continue
            if '+' not in chessgame.headers['TimeControl']:
                continue
            if int(chessgame.headers['TimeControl'].split('+')[0]) < 1200:
                continue

            # Make sure the game lasts at least 10 moves
            last_move = str(chessgame.end()).split('.')[0]
            if not last_move.isnumeric():
                continue
            if int(last_move) < 10:
                continue

            last_move = int(last_move)

            # Play moves past move 10
            board = chessgame.board()
            num_moves = random.randint(22, last_move * 2 + 2)
            i = 0
            for move in chessgame.mainline_moves():
                board.push(move)
                if i == num_moves:
                    break
                i += 1

            # Convert board to our representation
            board_position = fen_to_inputarray(board.fen())
            result = [int(res) for res in chessgame.headers['Result'].split('-')]
```

able to use this to our advantage so that we didn't have to fully download onto our machines all of the data for an entire month that we wanted to use. In fact, because of this convenience, we were able to randomize what data from what months was accessed when creating our dataset, to try and create a more varied dataset. A look at the code is seen in figure 5

6.2 Tensorflow 2.0 and Keras

Seeing as we would like the create a chess engine that would enable our goal, we decided to utilize Tensorflow by Google. Tensorflow is a very powerful and mature deep learning library that offers great visualization capabilities and offers many accelerated solutions however, we decided to take advantage of its effective training and inference features for

Figure 5: Collecting Lichess Game Data, Before Parsing

```
filelisturl = "https://database.lichess.org/standard/list.txt"
filelist = []

with urllib.request.urlopen(filelisturl) as fil:
    for line in fil:
        filelist.append(line.decode("utf-8").strip())

random.shuffle(filelist)

with open("test.csv", "w", newline="") as csvfile:
    writer = csv.writer(csvfile)
    writer.writerow(["result","white_elo","black_elo","board"])
    rows = []
    white_wins = 0
    black_wins = 0
```

deep neural networks when compared to many of its several competitors such as Pytorch.

However, while research we discovered Keras, an API that was tightly connected with the Tensorflow library and allows us to easily utilize convenient features while taking advantage of the deployment capabilities of the Tensorflow platform. Keras is a high-level API that consists of a highly productive interface that provides building blocks for developing machine learning solutions. Both of these libraries allows us to take advantage of their simplicity and powerful features to easily create our Autoencoders and Neural Network that were discussed above. Utilizing Keras we began creating the general layers and modals required to create our network. Below we create the basic building component of our autoencoder layers that accepts 69 nodes and consists of encoders layer 40 nodes $\rightarrow$ 20 nodes $\rightarrow$ 10 nodes 6.

Figure 6: Autoencoder in Keras

```
input_board = tf.keras.Input(shape=(69,))
encoder_layer_1 = tf.keras.layers.Dense(40, activation='relu')(input_board)
encoder_layer_2 = tf.keras.layers.Dense(20, activation='relu')(encoder_layer_1)
encoded = tf.keras.layers.Dense(10, activation='relu')(encoder_layer_2)
decoder_layer_1 = tf.keras.layers.Dense(20, activation='relu')(encoded)
decoder_layer_2 = tf.keras.layers.Dense(40, activation='relu')(decoder_layer_1)
decoded = tf.keras.layers.Dense(69, activation='relu')(decoder_layer_2)

autoencoder = tf.keras.Model(input_board, decoded)

return autoencoder
```

This code sample allows us to create two versions of our autoencoder stack to then

combine for our DeepChess network 7.

Figure 7: Deep Belief Network - Using two disjoint autoencoder stacks

```
autoencoder = tf.keras.models.load_model("saved_networks/autoencoder_model")
right_encoder = tf.keras.Model(inputs=autoencoder.input, outputs=autoencoder.layers[-4].output)
left_encoder= tf.keras.models.clone_model(right_encoder)
left_encoder.set_weights(right_encoder.get_weights())

for i, layer in enumerate(right_encoder.layers):
    layer._name = 'right_encoder_layer_' + str(i)
for i, layer in enumerate(left_encoder.layers):
    layer._name = 'left_encoder_layer_' + str(i)

right_encoder.compile(optimizer='adam', loss='binary_crossentropy')
left_encoder.compile(optimizer='adam', loss='binary_crossentropy')

print("Right Summary")
right_encoder.summary()
print("Left Summary")
left_encoder.summary()

combined = tf.keras.layers.concatenate([left_encoder.output, right_encoder.output])
```

Finally, we take our combined autoencoder layers and input them into our DeepChess layers that consist of four layers 8.

Figure 8: DeepChess

```
chess_network = tf.keras.layers.Dense(40, activation='relu')(combined)
chess_network = tf.keras.layers.Dense(20, activation='relu')(chess_network)
chess_network = tf.keras.layers.Dense(10, activation='relu')(chess_network)
chess_network = tf.keras.layers.Dense(2, activation='relu', name='dense_3')(chess_network)

chess_model = tf.keras.Model(inputs=[left_encoder.input, right_encoder.input],
outputs=chess_network)
```

6.3 Different Alpha-Beta Search

As mentioned above, we used an alpha-beta search method for our computer to pick which move to play. However, we made some changes as defined in the Deep Chess paper in order to take into account the fact that we don't have an evaluation function, but rather a network that determines which of two boards is better from the white player's perspective. The modified alpha-beta search function is shown in figure 9.

In our version, netpredict returns both boards that were compared in an array of length 2. However, the board that is better for white is always in the 0^{th} position and the board that's better for black is always in the 1^{st} position. Also, instead of trying to find some initial board state that is good to count as negative infinity for white and

Figure 9: New AlphaBeta Search

```
def alphabeta(board, depth, alpha, beta, max_player):
    if depth == 0 or board.legal_moves.count() == 0:
        return board

    if max_player:
        v = -1
        for move in board.legal_moves:
            cur = board.copy()
            cur.push(move)
            if v == -1:
                v = alphabeta(cur, depth-1, alpha, beta, False)
            if alpha == -1:
                alpha = v

            v = net_predict(v, alphabeta(cur, depth-1, alpha, beta, False))[0]
            alpha = net_predict(alpha, v)[0]
            if beta != 1:
                if net_predict(alpha, beta)[0] == alpha:
                    break
        return v
    else:
        v = 1
        for move in board.legal_moves:
            cur = board.copy()
            cur.push(move)
            if v == 1:
                v = alphabeta(cur, depth-1, alpha, beta, True)
            if beta == 1:
                beta = v

            v = net_predict(v, alphabeta(cur, depth-1, alpha, beta, True))[1]
            beta = net_predict(beta, v)[1]
            if alpha != -1:
                if net_predict(alpha, beta)[0] == alpha:
                    break
        return v
```

positive infinity for black, we instead pass them into the function as the integers -1 and 1. Then, we simply check to see if they've already been initialized before we compare them to another board state.

7 Conclusions

7.1 Training Results

Unforunately, due to a multitude of reasons that will be talked about in the next section, our neural network did not perform anywhere near as well as expected.

First of all, the final version of our autoencoder only achieved an accuracy of around 40-50%. Even after we tried to improve it, we often even made it significantly worse,

dropping as low as a 5% accuracy on both test and training data. This was very unfortunate, but we moved on to the deep chess network regardless, hoping that maybe even with this very low accuracy that the network would still be able to properly distinguish somewhat whether a position is winning or not.

Sadly, this was not the case. As much as our deepchess neural network does achieve around a 60-63% accuracy on both training and testing data, our later tests on playing moves were not very successful. Playing against very low ranked players proved that our engine essentially performed at the same level as if it were to pick a random move from the possibilities that were given to it. This meant our evaluation function, or even the autoencoder behind it, was very flawed.

In fact, when tested with some positions, our engine often played one of the worse possible moves in the position, sacrificing pieces for no gain, or playing moves that appear completely random. In fact, in figures 10 and 11, you can see the position before and after our engine makes a move, with its search depth limited to two.

Figure 10: Position Before Move

Figure 11: Position After Move

7.2 Possible Problems

While developing our Neural Network, we faced several issues that could potentially decrease our accuracy and efficiency. One of the issues we faced was an issue between the interaction of our training data and the ReLU activation function. The activation function is a linear function that will output the input directly if it is positive otherwise, it will output zero. We chose this activation function as it is quite easy to train and often achieves better performance. The issue became apparent after reviewing our original attempt at analyzing data from lichess chess games. In our original attempt, we were giving each while piece a specific value from 1 to 6 (1 - Pawn, 6 - King) while black pieces were the opposite value (-1 to -6). While training, our network would consider each black piece as a 0 value. We quickly noticed this issue and changed our training dataset to not contain any negative values so that our ReLU function could correctly evaluate the data.

As discussed above, while developing our autoencoder we decided to slightly deviate from the autoencoder described by the DeepChess paper by fixing our weights after we train all the layers that exist in our autoencoder stack. Meanwhile, the researchers in the DeepChess paper used the approach of fixing weights and using them as an input for each layer. We believe this may be a cause of lower efficiency results from our tests. By calculating the weights layer-wise we are extracting high-level features and

then further processing them by the next upcoming autoencoder. While our approach instead captures high-level features that were apparent throughout several autoencoder layer training. Currently, we are not sure which method is a favourable approach however we believe running experiments for both options can improve the overall accuracy of our autoencoders.

Due to the limitations of our systems and performance, we were only able to process 20,000 chess matches from lichess. This was not an issue when training the supervised model as we were randomly creating pairs of chess positons from White winning and Black winning datasets. However, this was a source of a bottleneck for our trained unsupervised layers. These layers were intended to learn about chess moves, the possibilities and the inherited rules by processing the 20,000 games as an input. We believe by increasing the data set of available chess matches would greatly increase our model's performance and accuracy due to the increase in training data. By adding several thousands of data, our model can learn a large variance of potential moves, learn about moves that were yet to be played on previously entered data and greatly improves its understanding of chess. The target goal we believe that can improve our model's performance would lie at a minimum of 2,000,000 games.

7.3 Possible Improvements

As we finished up with our experiments and began to receive feedback from our engine, we began to think of possible improvements that should improve the performance of our engine. Apart from the possible problems explain above, we decided these improvements are subjective and up to the experimenter. For instance, the chess engine can simply be improved by using better quality data. Some of the best moves are performed by the top-rated players as they are more likely to perform challenging and favourable movements hence by only using top-rated players as our training dataset, we are inevitably improving the capabilities of our engine. This however is a subjective improvement as the experimenter must decide which range of rating they consider "high level". Currently, our data acquisition function filters for player rating to be a minimum of 1200 for their game

to be including in the data set. By increasing this value we can filter for better games and player abilities however it may reduce the possible amount of data that the engine can train from. Similarly to the player rating, we can improve and personalize our engine by offering better filtering possibilities when training our engine. For instance, allowing the experimenter to exclude moves that were captured. These filters allow the engine to train according to the set requirements however this will be subjective and can alter the performance of the engine negatively if not done correctly. Lastly, to improve our engine, one can simply increase the input nodes given into our Deep Belief Network. Currently, we are using an input of 69 nodes. By increasing our input nodes we are adding more data points for each chess position. Hence we can check for further possibilities such as specific moves like the "en passant" or inserting player rating as a node to also compare the player's skill level while training from their moves. These filters allow for more data to be processed by our engine and result in increased training time however, it will improve the engine's ability to notice and choose more favourable positions.

7.4 Future Possibilities

1. With the improvements we can create something that can play at different skill levels

2. Whatever else we can think of

References

[1] Taiwo Oladipupo Ayodele. "Types of machine learning algorithms". In: *New advances in machine learning* 3 (2010), pp. 19–48.

[2] Mathias. Brandewinder. *Machine Learning Projects for .NET Developers*. eng. Berkeley, CA: Apress. ISBN: 9781430267669.

[3] J. A. Brown et al. "A Machine Learning Tool for Supporting Advanced Knowledge Discovery from Chess Game Data". In: *2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA)*. 2017, pp. 649–654. DOI: 10.1109/ICMLA.2017.00-87.

[4] Omid E. David, Nathan S. Netanyahu, and Lior Wolf. "DeepChess: End-to-End Deep Neural Network for Automatic Learning in Chess". In: *Lecture Notes in Computer Science* (2016), pp. 88–96. ISSN: 1611-3349. DOI: 10.1007/978-3-319-44781-0_11. URL: http://dx.doi.org/10.1007/978-3-319-44781-0_11.

[5] Steven J. Edwards. *Standard: Portable Game Notation Specification and Implementation Guide*. URL: https://ia802908.us.archive.org/26/items/pgn-standard-1994-03-12/PGN_standard_1994-03-12.txt.

[6] International Chess Federation. *FIDE Handbook*. URL: handbook.fide.com.

[7] Peter Harrington. *Machine Learning in Action*. USA: Manning Publications Co., 2012. ISBN: 1617290181.

[8] *Leela Chess Zero codebase*. URL: https://github.com/LeelaChessZero/lc0.

[9] Collins Achepsah Leke. *Deep Learning and Missing Data in Engineering Systems*. eng. 1st ed. 2019. Studies in Big Data, 48. Cham: Springer International Publishing. ISBN: 9783030011802.

[10] *lichess.org game database*. URL: https://database.lichess.org/. (accessed: 22.10.2020).

[11] A Matanovic. *Encyclopedia of Chess Openings (five volumes)*. Belgrad: Chess Informant, 1974-1979.

[12] Reid McIlroy-Young et al. "Aligning Superhuman AI with Human Behavior: Chess as a Model System". In: *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '20. Virtual Event, CA, USA: Association for Computing Machinery, 2020, pp. 1677–1687. ISBN: 9781450379984. DOI: 10.1145/3394486.3403219. URL: https://doi.org/10.1145/3394486.3403219.

[13] Vladimir Medvedev. *Point Value by Regression Analysis*. URL: https://www.chessprogramming.org/Point_Value_by_Regression_Analysis.

[14] Philipp Muens. *Game AIs with Minimax and Monte Carlo Tree Search*. URL: https://towardsdatascience.com/game-ais-with-minimax-and-monte-carlo-tree-search-af2a177361b0.

[15] David Silver et al. *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*. 2017. arXiv: 1712.01815 [cs.AI].

[16] *Stockfish Code Base*. URL: https://github.com/official-stockfish/Stockfish/blob/master/src/types.h.

[17] Yanmin Sun, Andrew K. C. Wong, and Yang Wang. "An Overview of Associative Classifiers". In: *Proceedings of the 2006 International Conference on Data Mining, DMIN 2006, Las Vegas, Nevada, USA, June 26-29, 2006*. Ed. by Sven F. Crone, Stefan Lessmann, and Robert Stahlbock. CSREA Press, 2006, pp. 138–143.

[18] Philippe De. Wilde. *Neural Network Models Theory and Projects*. eng. Second Edition. London: Springer London. ISBN: 9781846286148.