What is the optimal tradeoff between handcrafted features vs neural trained features when creating a chess playing agent?

Aron Klevansky
Student ID: 941190845
klevansky@campus.technion.ac.il

Niv Ostroff
Student ID: ???
???

Tamir Offen
Student ID: 211621479
tamiroffen@campus.technion.ac.il

October 2023

1 An Introduction to Chess Engines and Autonomous Chess Playing Agents

Chess is a fascinating game. One of the oldest games in the world (originating from the ancient game of chaturanga from nearly 1500 years ago), it has undergone many different revolutions in understanding: from the Romantic period of the late 18th century to the Scientific, Hypermodern, New Dynamism and finally Modern Era of today. This latest era, known as "The Modern Era", is characterised by the large scale integration of autonomous chess playing agents known as "Chess Engines", which has had a profound impact on the game. Most top chess players use these chess engines in order to evaluate chess positions for them, analyse critical lines for them and even generate new opening novelties that they can use against opponents! But where do these so called "Chess Engines" come from, and how do they work?

1.1 History of Chess Engines

Strangely enough, the first known account of an autonomous chess playing agent predates the invention of the computer. The "Mechanical Turk", was a life sized human-like robot invented by Wolfgang von Kempelen in the year 1770 - introduced to the public as "being able to play chess autonomously". The Mechanical Turk was able to beat human opponents at chess, and even solve the knight's tour chess puzzle. Unfortunately, this story was too good to be true. Many years later it was discovered that the Mechanical Turk was actually just one big hoax - there was actually a person hidden underneath the automaton who would control where the automaton moved its pieces. [13]

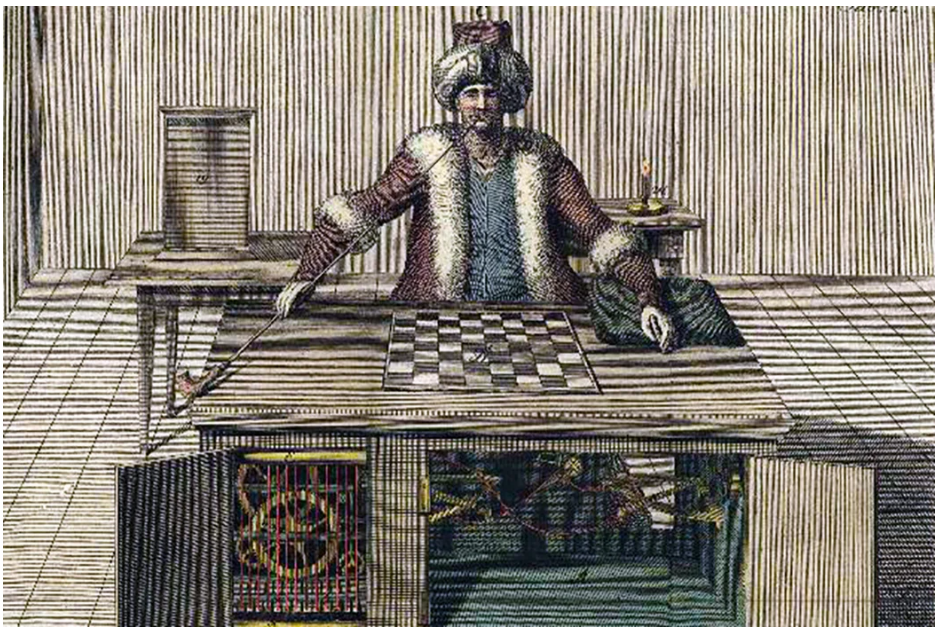


Figure 1: showing an illustration of the infamous "Mechanical Turk"

The first real case of a chess computer originated in 1912. Invented by Leonardo Torres Quevedo and known as "El Ajedrecista", the machine was able to checkmate in simple endgames of king and rook against bare king and even identify illegal moves! [5]

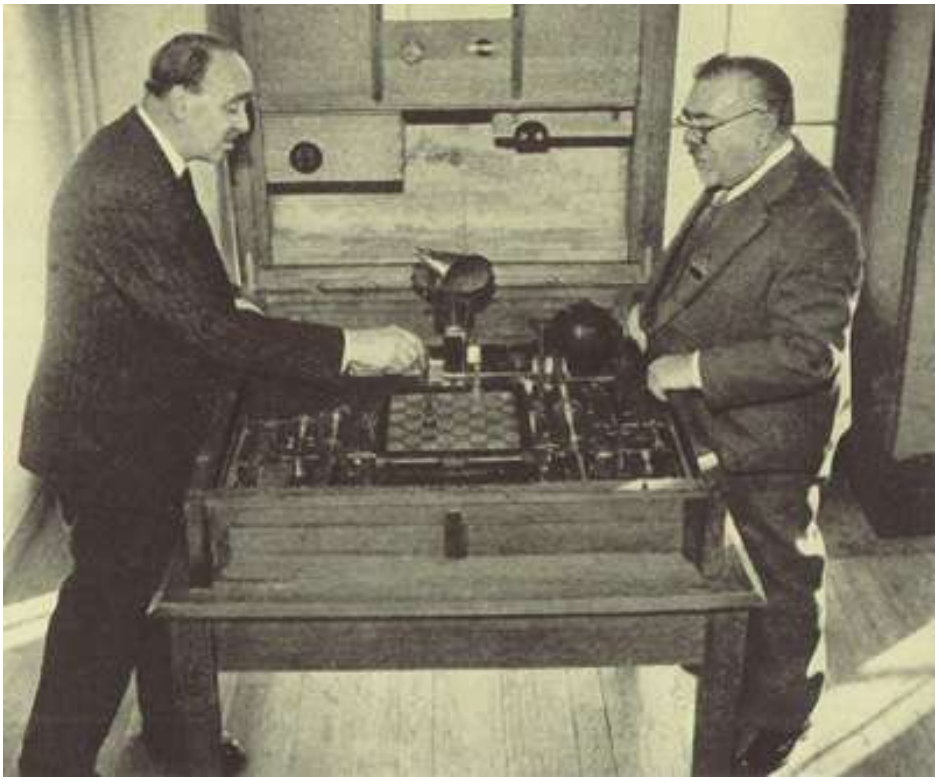


Figure 2: Leonardo Torres with the El Ajedrecista machine

Of course, the main field of developing chess engines more formally came into being in the 1940s through the works of Alan Turing (known as the founding father of the field of computing) and Claude Shannon (famous for his innovations in the field of information theory). Both Turing and Shannon wrote papers on developing chess

playing algorithms. Unfortunately, computing power in this era was not powerful enough to produce any agent of significance.

Slowly but surely, with major developments in the field of Computer Science both in terms of Hardware and Software, chess playing agents from the 1960s onwards started to experience a major increase in playing strength. During this time, one of the most popular algorithms being used (and still being used) is the Minimax algorithm. This algorithm (and its optimization using alpha-beta pruning) was proven by John von Neumann in 1928 and concentrates on the maximisation of one player's score and the minimization of the opposing player's score. Major advances to this initial algorithm took place in order to improve its efficiency (especially for the field of chess). Some of these advancements include move selection techniques, heuristic techniques and iterative deepening. Indeed, many grandmasters also became very involved in improving the quality of chess engines, including Chess World Champion Mikhail Botvinick (who himself held a degree in Electrical Engineering) and Grandmaster Walter Browne.

Of course, no discussion of the history of Chess Engines would be complete without mentioning the moment Autonomous Chess Playing Agents were decided to finally be better than the humans creating them. Of course, we are referring to the famous match in 1997 between then World Champion Garry Kasparov and the Chess Engine "Deep Blue" developed by IBM. After coming back from a 4-2 defeat in 1996, Deep Blue finally managed to beat Kasparov $3.5 - 2.5$! [12]

Figure 3: Kasparov making a move against IBM's Deep Blue in 1996

1.2 Current Innovations in the Field

Since the defeat of Kasparov, chess engines have become stronger and stronger and have reached a point where many are often used by top grandmasters in preparation for matches. Most of these chess engines have relied on some version of von Neumann's minimax algorithm with a and differ slightly according to the specific heuristic they use. Although these heuristics may differ in certain ways though, they have generally all been made with "hand crafted features" such as scores for things like king position, number of pawns, number of open files occupied by rooks,...etc.

However, with the recent boom in the field of deep learning due to the increased power offered by GPUs, there is a new approach to building chess engines using neural models, which is generating much hype in the field:

In 2017, building on their success from the development of AlphaGo, the team at DeepMind released a chess playing agent known as "AlphaZero" that had been trained with deep learning models (instead of using a heuristic that uses hand crafted features). In this way, the features of the heuristic are "learned" by the model. With this approach,

AlphaZero was able to beat the then current engine at the time (Stockfish) in a 100 game match, by winning 28 games and achieving a draw for the remaining 72!

Since then, most top chess programs such as Stockfish, Komodo and Leela all use neural networks in their programming - illustrating a major paradigm shift in the field. [15]

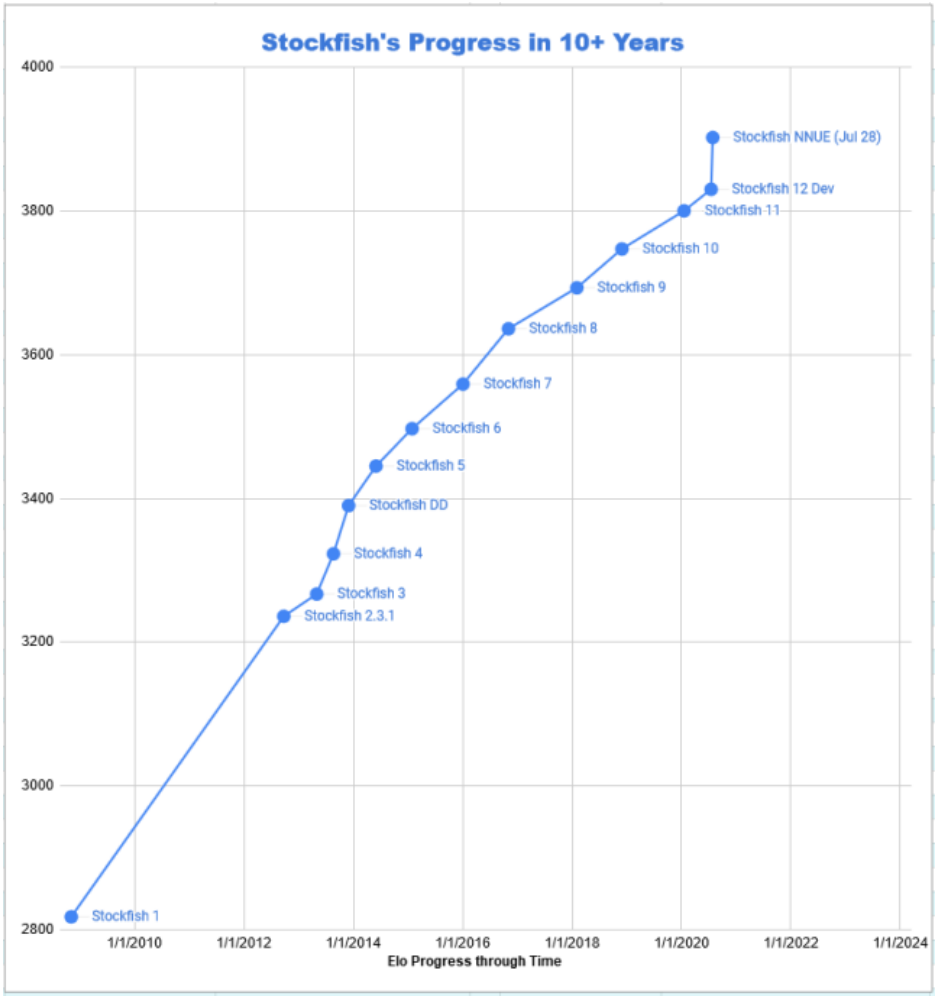


Figure 4: Stockfish ELO change over time. Note jump in 2020 with introduction of Neural Network!

2 An Introduction to our Chess Engine

As is seen from the previous section, there are two main types of chess playing agents: ones that use hand crafted features for their heuristic and those with heuristics that are trained neurally. The currently most successful chess engine in the world, Stockfish, uses both hand crafted features and a feed forward neural network in its implementation.

As such, we wanted to ask the question: What is the optimal tradeoff between handcrafted features vs neural trained features when creating a chess playing agent?

In order to answer this question, we have built a chess playing agent that works based on a tradeoff between a deep learning model and classical chess heuristics. The agent consists of two separate components:

1. A Convolutional Neural Network
2. A Minimax Algorithm whose heuristic is trained using genetic algorithms

In order to generate a final move, the chess playing agent makes use of a hyperparameter λ which provides a weighted tradeoff between these two different components of the chess engine. This tradeoff is shown in the equation below:

Move Played = $argmax_{move}$ (λ(list of moves chosen by CNN) + (1-λ)(list of moves chosen by heuristic))

3 Implementation of our Chess Engine

We will now provide high level discussions of each component comprising the chess playing agent:

3.1 Implementation of the Convolutional Neural Network

In this section, we will delve into the practical aspects of implementing a CNN based Chess engine model. Our primary objective is to design and develop a CNN based model that will input a chess board and current player, and output an intelligent and legal chess move. In order to achieve this, we will follow a structured approach based on various known ML and engineering techniques.

The choice of using a CNN based model for our chess engine is driven by their effectiveness in other fields, such as image recognition, which tells us that CNNs are capable of capturing intricate patterns and relationships within multi-channel input data. This characteristic makes CNNs particularly well suited for understanding the complex spatial arrangements on a chessboard.

We will begin with a relatively simple "proof of concept" regression task where we test our CNN architecture to predict Stockfish (SF) board evaluations. This initial step serves as a litmus test, allowing us to assess our CNN's capacity to evaluate the quality of a chessboard position.

After that, we will focus on the selection and fine tuning of several training hyperparameters. This process is essential in ensuring that our final CNN model optimally learns. We will discuss our approach to hyperparameter tuning and the rationale behind our choices.

Lastly, we will train our CNN model using the optimal training hyperparameters that we found. The training was done on a very large chess dataset. Throughout the training, we record various metrics, such as training loss and piece selection accuracy. At the conclusion of the training process, we will have developed a CNN based chess engine model, which we will analyze to assess its strengths and weaknesses.

3.1.1 Background on ANNs and CNNs

Artificial Neural Networks (ANNs) are computational models inspired by the structure and function of the human brain. They consist of interconnected nodes, or neurons, organized into layers, and they excel at learning and recognizing patterns in data. ANNs have become the basis of many deep learning models, which try to solve a wide range of problems from image recognition to natural language processing.

ANNs are made up of neurons, connections (weights), and layers. Neurons serve as the computational units within the network, performing various mathematical operations. Connections, characterized by weights, determine the strength of information transfer between neurons, allowing for the adjustment of network behavior during training. These neurons are organized into layers. [10]

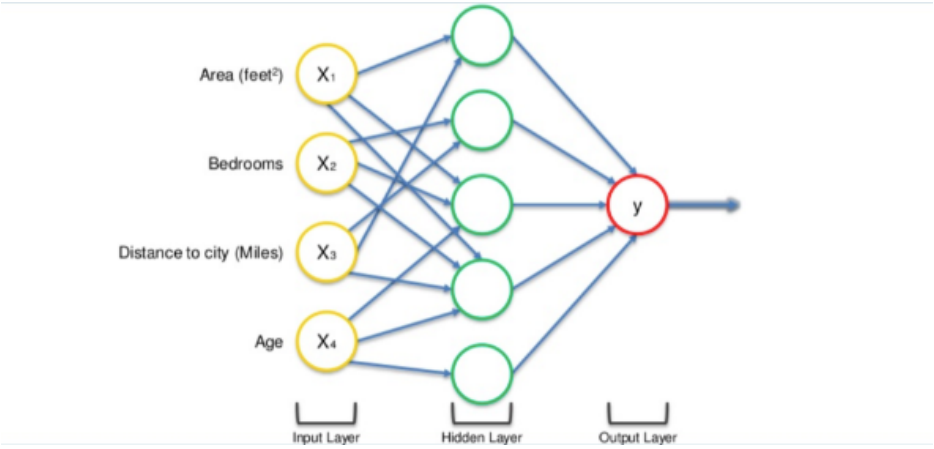


Figure 5: A neural network with a hidden layer

Activation functions are a fundamental component of neural networks by introducing non-linearity to the model. They are applied to the output of each neuron, influencing how the neuron is activated based on the input it receives. The choice of activation function depends on the nature of the problem and the network architecture. Experimentation with different activation functions is an essential part of optimizing the performance of a neural network for a specific task, as well as significantly affecting training performance. There are several common activation functions used in neural networks, each with its own characteristics: [1]

- Sigmoid Function: The sigmoid function transforms input values into a range between 0 and 1. It is particularly useful in binary classification problems where the network needs to produce probabilities, such as logistic regression.
- Hyperbolic Tangent (tanh) Function: The tanh function is similar to the sigmoid but transforms input values into a range between -1 and 1, offering zero-centered output. It is often used in hidden layers of neural networks.
- Rectified Linear Unit (ReLU): ReLU replaces negative inputs with zero while leaving positive inputs unchanged. Similar to a linear activation function, but negative values are transformed to 0.

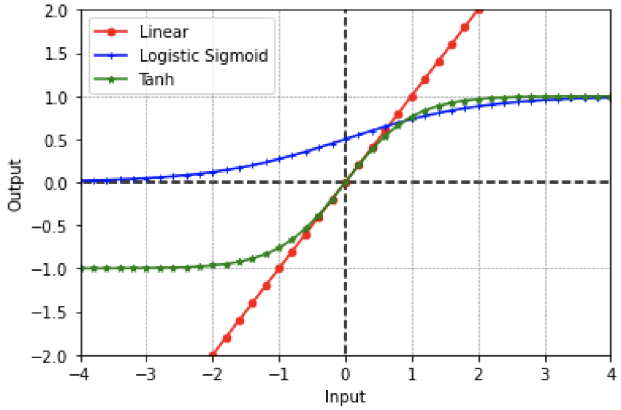


Figure 6: An illustration of Linear, Logistic Sigmoid and Tanh AFs.

While there are several other popular activation functions, such as Leaky ReLU and Exponential Linear Unit (ELU), it's important to note that these functions are often considered more domain specific, whereas the three listed above are regarded as fundamental activation functions.

The training of neural networks is a crucial phase where the model learns to perform its designated task and make predictions. It involves a series of iterative steps that gradually fine tune the network's parameters using labeled data. Each data point consists of input data ($\vec{x}$) and its corresponding output data (y_{true}), allowing the network

to learn by example. This is called supervised learning.

During training, the network's predictions (y_{pred}) are compared to the actual outputs in the training data, and any discrepancies are quantified using a loss/cost function. A loss function computes the difference between the predicted output and the true output for a given input ($\mathcal{L}(y_{pred}, y_{true})$). Usually, a lower loss is better than a higher loss, therefore the goal during training is to minimize the loss function. Common loss functions include mean squared error (MSE) for regression tasks and cross entropy (CE) for classification tasks:

$$\mathcal{L}_{\text{MSE}} = \frac{1}{n}\sum_{i=1}^{n}(y_{\text{pred}}^{(i)} - y_{\text{true}}^{(i)})^2 \qquad \mathcal{L}_{\text{CE}} = -\sum_{i=1}^{n} y_{\text{true}}^{(i)} \log(y_{\text{pred}}^{(i)})$$

To minimize the loss during training, optimization algorithms like gradient descent are employed. Gradient descent works by iteratively adjusting the network's weights and biases in the direction that reduces the loss. The gradients of the loss with respect to the network's parameters indicate how much each parameter should be updated.

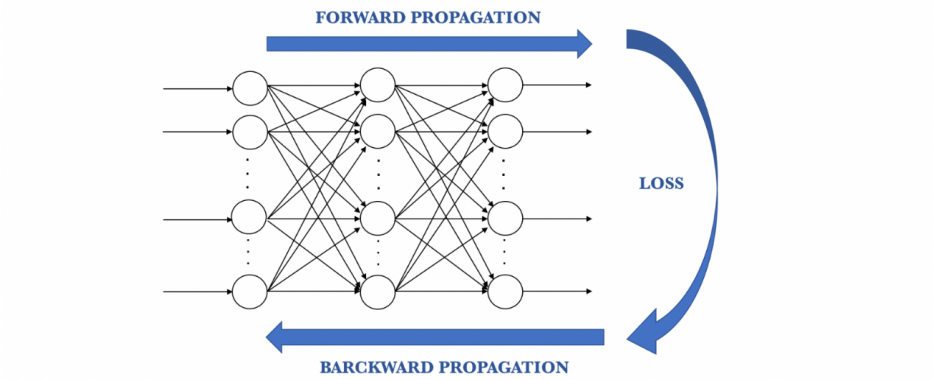


Figure 7: Training a Neural Network

The training process continues through multiple iterations, with the network progressively improving its ability to make accurate predictions. In theory, as the training proceeds, the network's parameters are adjusted and eventually converge towards values that enable it to generalize well to new, unseen data.

While training a deep neural network (DNN) follows a similar outline to training a shallow NN, there are several important factors to consider when working with DNNs:

- Gradients in shallow NNs remain stable, neither exploding nor vanishing. On the other hand, Gradients in deep NNs often suffer from instability, either exploding or vanishing, especially in the early layers of the network. Some of the ways we can combat this in DNNs is with appropriate weight initialization, using ReLU based AFs instead of sigmoid or tanh, and with tricks like BatchNorm layers or Skip Connections which we will explore later in the paper.
- Deep NNs need significantly more data to train on in order to generalize well in comparison to shallow NNs. Training DNNs can also take much longer than training shallow NNs. This can be attributed to DNNs having much more parameters to learn during training in comparison to shallow NNs.
- Deep NNs are more susceptible to overfitting than shallow NNs, because of their increased complexity, making them more capable of memorizing training data. Regularization techniques like dropout are effective when used on DNNs to combat overfitting.

In recent years, Convolutional Neural Networks (CNNs) have emerged as a powerful and versatile class of artificial neural networks. Originally designed for computer vision tasks, CNNs have found applications across various domains due to their remarkable ability to capture spatial patterns and hierarchical features from multi channel data. Unlike traditional feedforward neural networks, CNNs are structured to take advantage of the grid-like topology of data, making them particularly well suited for tasks involving images, sequences, and other structured data. They have played a pivotal role in revolutionizing the field of computer vision by enabling machines to automatically learn and recognize intricate patterns and features within images, such as edges, textures, and complex object

representations.

One of the key features of CNNs is their use of convolutional layers, which apply a series of learnable filters to input data in order to extract relevant features. These filters slide over the input, effectively scanning it for local patterns. This mechanism allows CNNs to automatically learn and emphasize meaningful patterns, making them highly effective in tasks like image classification, object detection, and segmentation [11].

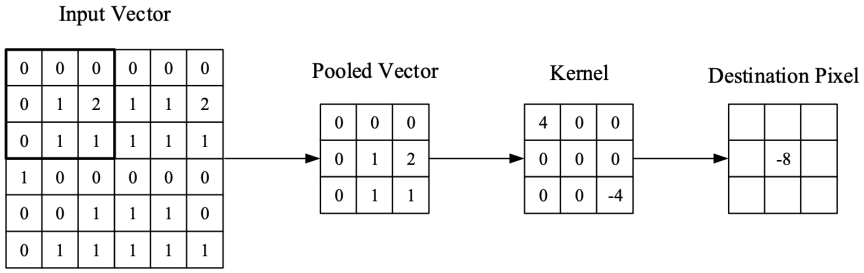


Figure 8: A visual representation of a convolutional layer. The center element of the kernel is placed over the input vector, of which is then calculated and replaced with a weighted sum of itself and any nearby pixels.

Very simple CNNs are comprised of three types of layers. These are convolutional layers, pooling layers and fully-connected layers. When these layers are stacked, a CNN architecture has been formed. The pooling layer will then simply perform down sampling along the spatial dimensionality of the given input, further reducing the number of parameters within that activation. The fully-connected layers will then perform the same duties found in standard ANNs and attempt to produce class scores from the activations, to be used for classification. It is also suggested that ReLU may be used between these layers, as to improve performance and introduce non linearity [11].

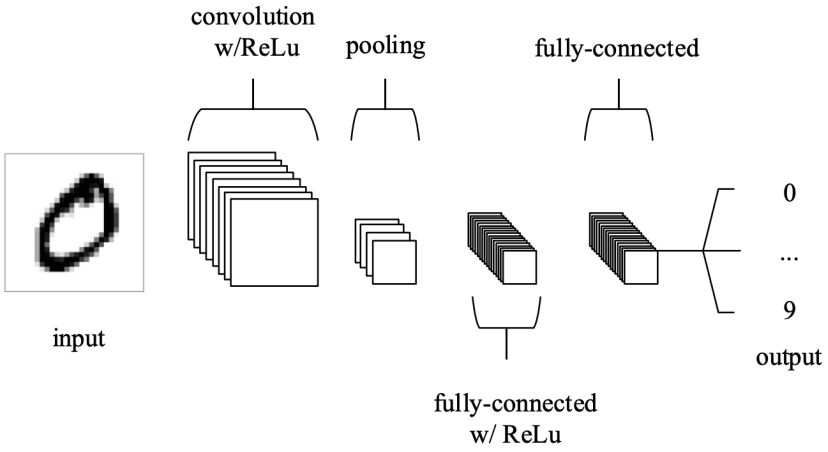


Figure 9: A simple CNN architecture example, comprised of just five layers

There are three main hyperparameters in a convolutional layer [4]:

1. *Depth*, **K** - The number of filters we would like to use, each learning to look for something different in the input. We will denote their width as **F**, also known as the spatial extent.

2. *Stride*, **S** - How many pixels we slide the filter. When the stride is 1 then we move the filters one pixel at a time. When the stride is 2 (or uncommonly 3 or more, though this is rare in practice) then the filters jump 2 pixels at a time as we slide them around. This will produce smaller output volumes spatially.

3. *Padding*, **P** - Sometimes it will be convenient to pad the input volume with zeros around the border. The size of this zero-padding is a hyperparameter. The nice feature of zero padding is that it will allow us to control the spatial size of the output volumes.

A conv layer that accepts a volume of size $\boldsymbol{D_1 \times W_1 \times H_1}$, will produce a volume of size $\boldsymbol{D_2 \times W_2 \times H_2}$ such that:

$$\boldsymbol{D_2 = K} \qquad \boldsymbol{W_2 = \frac{W_1 - F + 2P}{S} + 1} \qquad \boldsymbol{H_2 = \frac{H_1 - F + 2P}{S} + 1}$$

3.1.2 Why Use a CNN Model for Chess?

In 2014, Amos and Storkey achieved a notable 44.4% accuracy in predicting professional moves in the game of Go using CNNs [17]. Go is renowned for its reliance on complex reasoning and expert intuition. This result demonstrates that, given an appropriate architecture and a sufficiently large training set, a CNN can achieve reasonable performance in playing a complex game like Go.

Unlike Go, chess relies heavily on the interplay between the chess pieces. It involves short term plays that collectively shape long term strategies. This dynamic arises from the fact that the advantage of a specific chessboard position hinges on the interplay between the rules governing each piece. Consequently, mastering chess requires a deep understanding of how the precise positioning and interactions of the pieces on the board can yield a strategic advantage [14].

However, a strong chess playing agent does not have to play according to long term strategies but, rather, focus on determining the best move given a specific chessboard position. This task is well suited for a CNN because it can effectively utilize small, local features to predict a chess move. If we feed the data into a CNN in a way that the label of a training example (chessboard) represents a highly skilled move (such as one made by a Grandmaster), then the network can learn to recognize and reproduce these expert level moves. This perspective accurately reflects the character of chess moves in high level games, as nearly every move made by an expert level chess player can be considered a sensible choice, especially when averaged over the entire training set [14].

3.1.3 Datasets

There are two datasets that we used for this section (3.1) of the project:

1. Chess Evaluations [2] - Contains around 16 million chess positions with a Stockfish evaluation at depth 22 from white's point of view. This dataset was used to train our Chess Position Evaluation Network (*ChessPosEvalNet*), which we called our "proof of concept".
2. Chess Games [3] - Contains about 6.25 Million chess games played on lichess.org during July of 2016. Each games contains the list of moves ("AN") as well as the ranking of both white and black players (ELO). This dataset was used to train our Chess playing network (*ChessNet*).

Both of these datasets can be downloaded and explored using the `ChessDatasets.ipynb` notebook.

In order to effectively and efficiently use these datasets with our CNNs, we created custom PyTorch Dataset classes for them. This class enables convenient data access and indexing. For the Chess Evaluations dataset, we return a random board (x) with its corresponding SF score (y), regardless of what index was passed in. For the Chess Games dataset, we did something similar. We chose a random game, chose a random move played in that game, and returned a board up until *before* that move (x), along with that move (y), and the player color [9]. The main reason why for both dataset classes we chose to 'index' into the dataset by randomly choosing a game is for time complexity reasons and because the index of a certain game holds no effect on the training process of the CNNs. Finally, we used PyTorch's DataLoader class which is responsible for efficiently loading and batching data during training and evaluation.

The Dataset classes can be found in `data.py`.

3.1.4 Preprocessing

Our preprocessing steps were largely based on the steps taken by Barak Oshri and Nishith Khandwala [14], and by Moran Reznik [9].
First we preprocessed our datasets:

1. The Chess Evaluations dataset includes non numerical values in the 'Evaluation' column, which is a problem for our regression ChessPosEvalNet model. The reason why there are non numerical values is because SF evaluations take into account forced checkmates. For example, if white can force checkmate black in 5 moves, the SF evaluation of the position is not a huge number (which would signal white is winning by a lot), but

the string '#+5'. Therefore, we preprocessed the dataset to replace any white forced checkmates with +2500, and black forced checkmates with −2500.

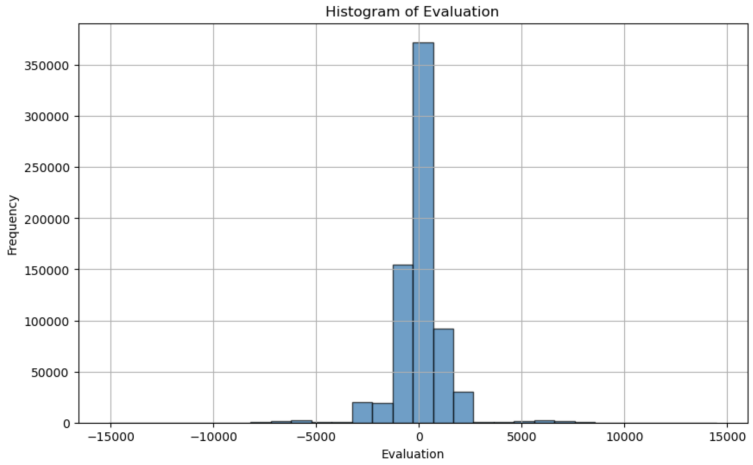


Figure 10: Histogram of 'Evaluation' in the Chess Evaluations dataset

By looking at figure 10 we can see that most positions have SF evaluations of between −2500 to +2500, which is why we chose these two numbers to replace forced checkmate.
We now noticed that the values of the evaluations are large, which might cause our loss function MSE to produce very large gradients. This is not optimal for learning. The data is distributed normally around 0, so we decided to use z-score normalization to keep this distribution.

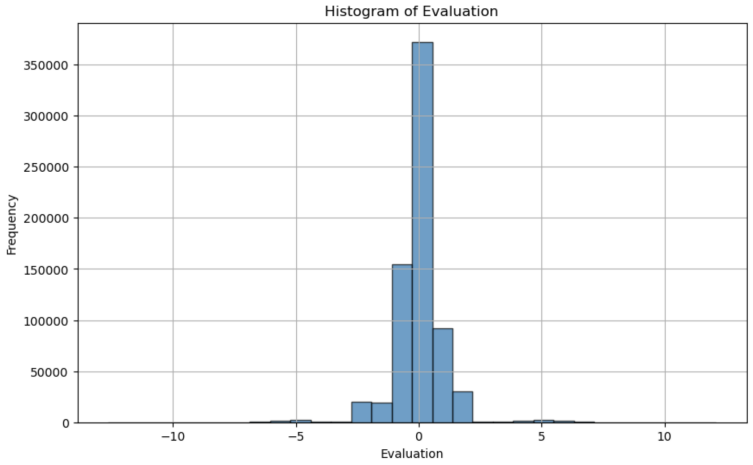


Figure 11: Histogram of 'Evaluation' in the Chess Evaluations dataset after z-score normalization

2. We preprocessed the Chess Games dataset to include only games of highly skilled player ($ELO > 2000$) and games that are 'not short' (20 moves or greater). Before preprocessing, the dataset had $6,256,184$ games, and after preprocessing we are left with $883,376$ games, which is still plenty of examples to train on.

A challenge we ran into was how to represent a chess board so that it would be effective for learning. The trivial approach is to assign a unique number to each type of piece, see figure 12. This is a bad idea because it does not capture the relationship of the pieces in a manner that is effective for a CNN to learn. For example, if we assigned $white\_pawn = 1$, $black\_pawn = 7$, does that mean that black pawns are worth more than a white pawns?

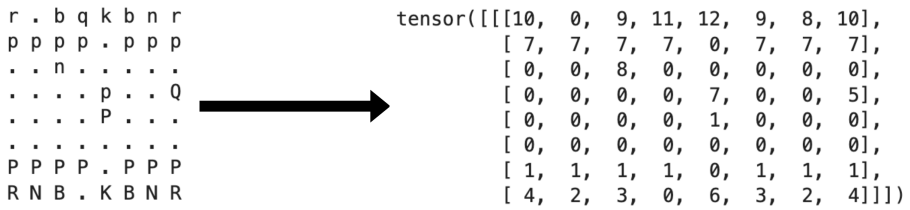


Figure 12: Unique piece type assignment board to tensor conversion example

An approach that would take advantage of CNN's ability to work with multi-channel data is to convert a chess board into 6 layers, one for each piece type, and assign $+1$ to white pieces and -1 to black pieces. See Figure 13 for an example. This approach offers several advantages over the trivial approach. Firstly, it provides a more detailed and informative representation of the chess position, allowing the model to capture intricate spatial relationships and piece type specific features. Secondly, the inclusion of white vs. black (+1 vs -1) enables the model to assess the relative strengths and weaknesses of each color. Lastly, this approach allows the model to train better by promoting better gradient flow during training.

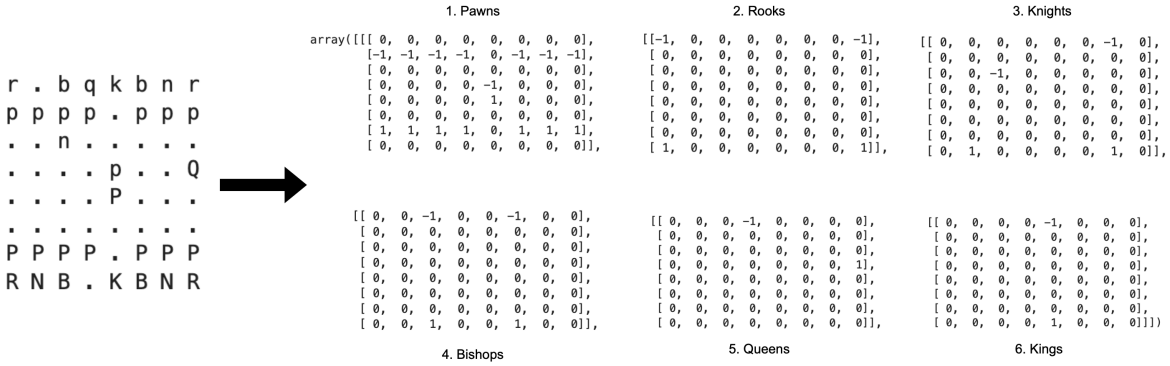


Figure 13: 6-layer representation of a chess board example

Continuing on from the last problem we faced, we needed to represent a chess move in a way that is convenient for learning. The trivial approach would be to represent each possible move as a class of its own, like in a traditional multi-class classification task. This is a bad idea because the space of possible moves is very large in Chess. The CNN would need to score a total of 4096 possible classes.

$$\textit{choose}(\text{piece to move}) \text{ and } \textit{choose}(\text{square to move to}) \Rightarrow (8 \times 8) \times (8 \times 8) = 4096 \text{ possibilities}$$

Even if we wanted to deal with the consequences of having over 4000 classes, this representation of a chess move would not help our CNN to learn how to play chess because the index of a move holds no information for learning.

A better, albeit still novel, approach is to divide the classification challenge into two parts:

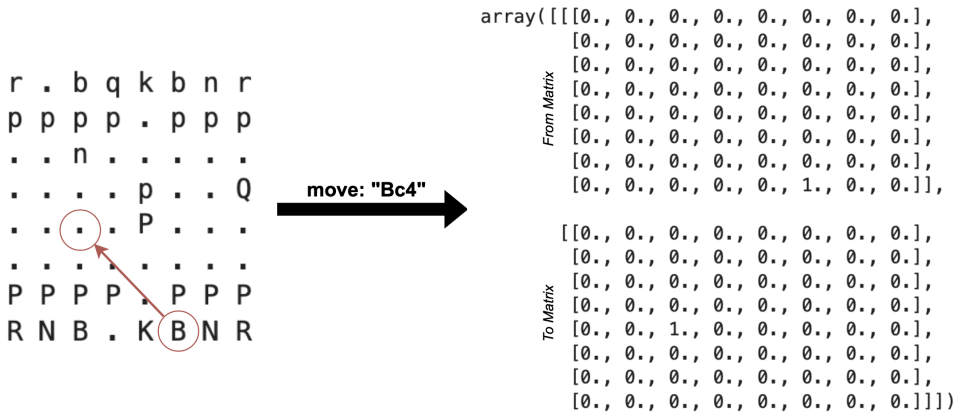


Figure 14: "Bc4" (Bishop to square c4) move representation using from and to matrices example

- From Matrix: Which square to move. Captures the notion of *"escape"* when a piece is under attack or the king needs to move while in check.
- To Matrix: Which square to move the piece to.

This new approach has several advantages in comparison to the trivial approach. Firstly, the class size is $(8 \times 8) \times 2 = 128$, which is a about a square root of the original class size of 4096. Secondly, human chess players often try to improve the position of their pieces, or improve their worst placed piece. This is knows as Makagonov's principle. In a similar way, the CNN identifies what piece to move (move matrix), which is often a poorly placed piece - and then tries to move it to a better square (to matrix).

The preprocessing code can be found in `preprocess.py`.

3.1.5 CNN Architecture

"All models are wrong, but some are useful" – George Box.

We employed a CNN model architecture developed by Moran Reznik [9] in our study. While Reznik did not provide an in depth rationale for the design choices behind this model, we identified specific factors that lead us to believe it is well suited for feature extraction from a chess board. The hyperparameters for this model are 'hidden_size' and 'hidden_layers', which are set by default to 200 and 4 respectively.

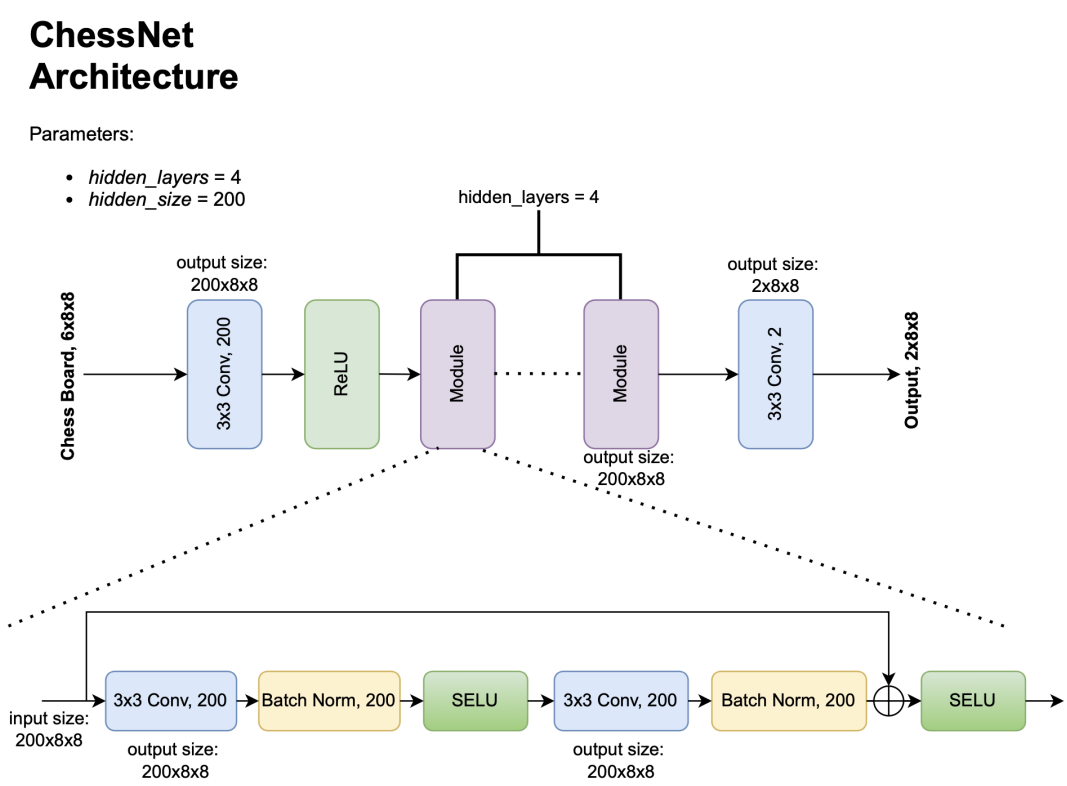


Figure 15: ChessNet Diagram with Default Parameters

The model begins by inputting a chess board representation of shape $6 \times 8 \times 8$. To extract meaningful features and patterns from this input, we employ a 3×3 convolutional layer equipped with 200 kernels. This choice is particularly effective because it aligns with the 8×8 grid structure of the chessboard. The 200 kernels in this layer significantly enhance the model's capacity to find patterns, thereby enabling it to grasp the wide array of intricate relationships within the chessboard representation. Following the convolutional layer, we introduce a ReLU activation layer. This introduces non linearity into the model, a standard practice in CNNs, allowing it to capture complex relationships and feature maps effectively.

To enhance feature extraction from the chessboard, our model employs a series of four specialized 'modules'. Each module takes an input denoted as $\mathcal{X}$, which has a shape of $200 \times 8 \times 8$, and produces an output, denoted as y, also with a shape of $200 \times 8 \times 8$. $\mathcal{X}$ is initially processed by a 3×3 convolutional layer with 200 kernels for similar reasons as above. Following this, the output passes through a batch normalization layer, which operates on 200 channels. Batch normalization is a component that helps stabilize the training process and accelerates convergence, allowing us to train deeper modules faster and with less compute power. It normalizes the activations of each channel across mini batches of data, contributing to more efficient learning. A SELU activation layer follows the batch normalization. SELU activations are known for their role in maintaining consistent mean and variance of activations during training, contributing to better convergence and generalization. Next, the output is passed through another convolutional layer followed by a batch norm layer further refining the learned features, but is also followed by a residual connection which helps prevent vanishing gradients in deep NNs. And finally, it is passed through another SELU layer to introduce non linearity as well as improve training.

The final layer of our architecture plays a critical role in producing the output format for chess moves. It takes a $200 \times 8 \times 8$ tensor, which has undergone feature extraction through the preceding modules, and processes it through a 3×3 convolutional layer with only 2 kernels. The choice of using a convolutional layer with just 2 kernels at this stage is intentional. We aim to generate a chess move prediction in a specific format, as outlined in Section 3.1.4 on preprocessing. In this format, each move is represented by a 'from matrix' and a 'to matrix,' both of which are of shape 8×8.

The CNN architecture code of both ChessNet and ChessPosEvalNet can be found in `complex_model.py`.

How do we use a trained ChessNet model to predict a move given a chessboard (`choose_move` function in `utils.py`)?

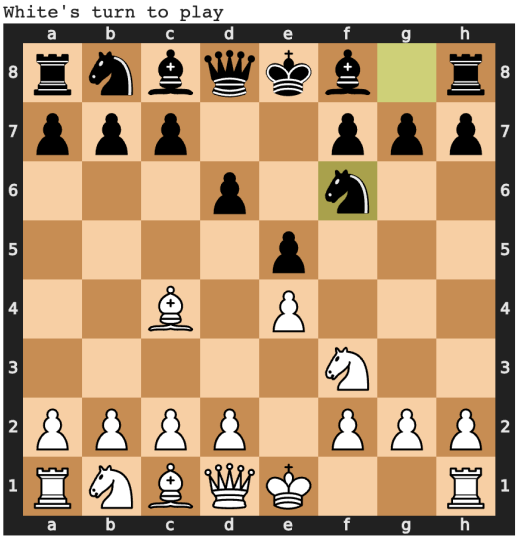
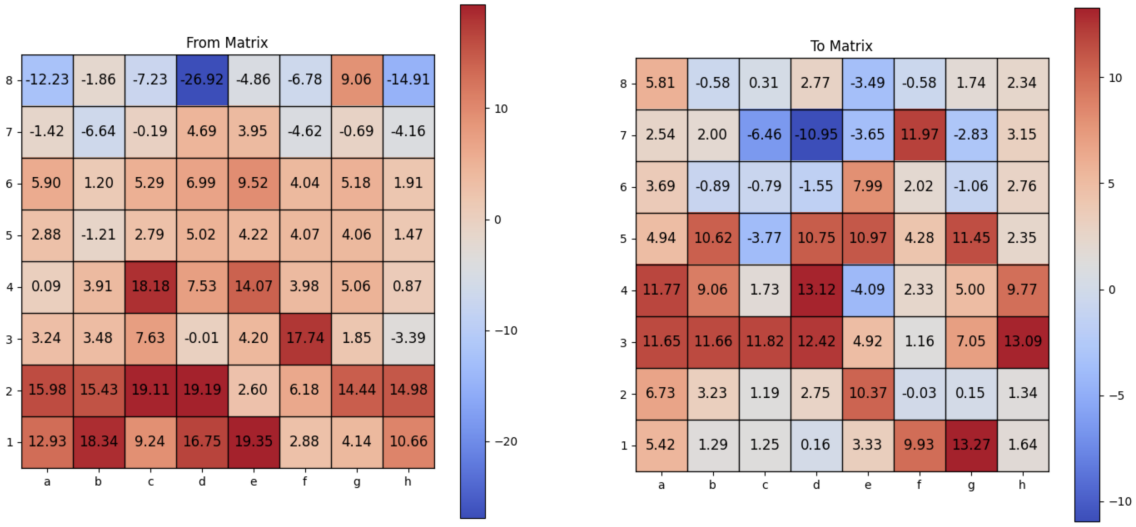


Figure 16: Example Board

1. Get the legal moves available to the current player. If no legal moves are available, then the board is in a terminal state.
2. Check if there is a k-step forced checkmate available to the current player. We chose k=1.
3. Pass the board into the trained ChessNet model. We get back *from* and *to* matrices.



4. Choose a *from square*, by making a probability distribution of the legal moves in the from matrix.

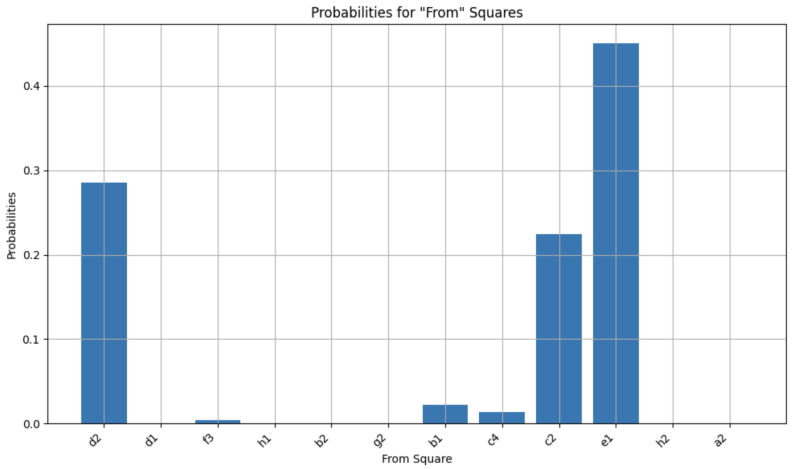


Figure 18: Calculated using `distribution_over_moves` func in `utils.py`

5. Aggregate the values in the *to matrix* corresponding to the chosen piece.

```
chosen piece: e1
vals in to matrix: [10.37, 9.93, 13.27]
```

Figure 19: The king on *e1* can move to [*e2*, *f1*, *g1*]

6. The chosen square to move 'chosen piece' to will be the square corresponding to the highest value in the to matrix from step 5.

```
model's chosen move: e1g1
```

Figure 20: square *g1* has the highest value (13.27)

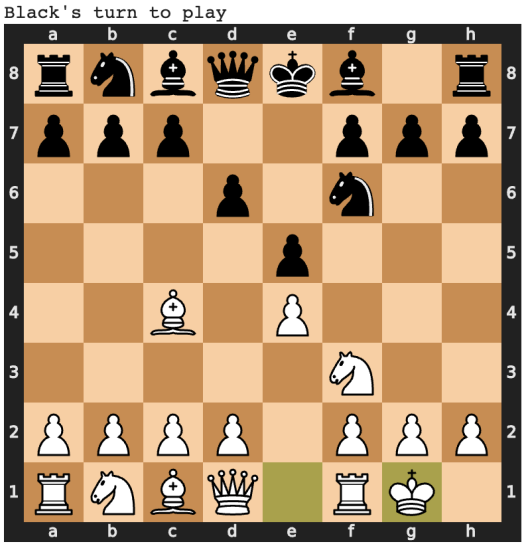


Figure 21: Board from ref. 16 after the model chose the move - *e1g1*

3.1.6 Proof of Concept: Chess Board SF Evaluation Model

Before training our ChessNet model to predict moves, we conducted a preliminary evaluation of our CNN architecture to assess its ability to extract relevant features from a chessboard.

We adapted the original CNN architecture of ChessNet by modifying the final layer to include a fully connected head, as depicted in Figure 22. This is a common practice in CNNs for regression tasks.

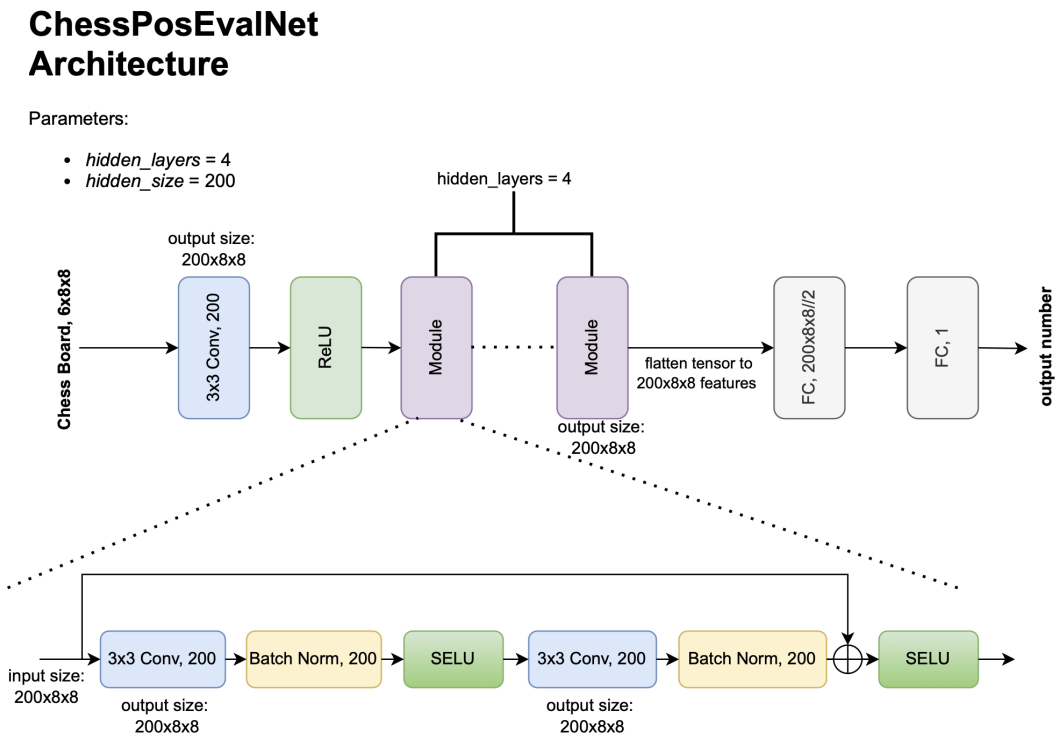
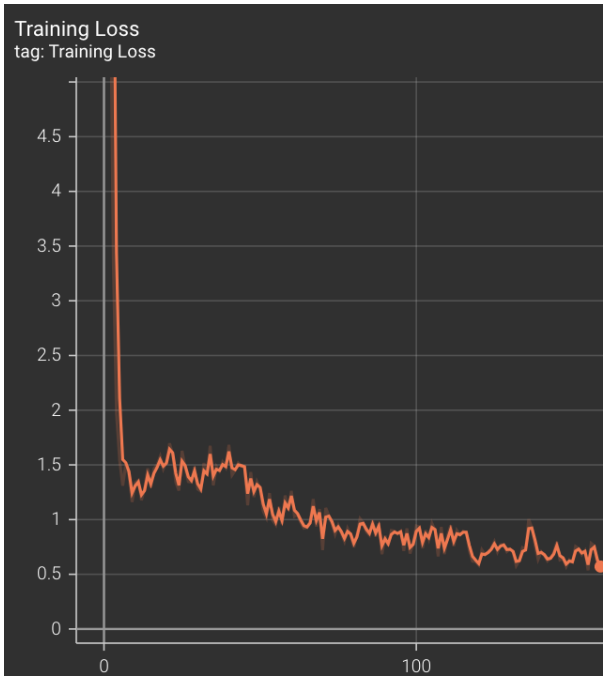
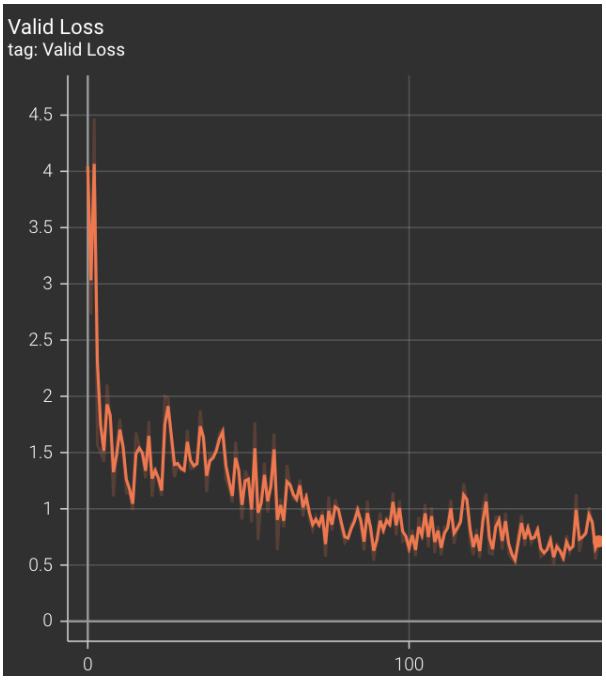


Figure 22: ChessPosEvalNet Diagram with Default Parameters

We trained our model on one NVIDIA TITAN Xp (12 gb) for 160 epochs, or about 8 hours on the Chess Evaluations dataset. We used the Adam optimizer for training due to its adaptive learning rate, efficiency in handling noisy gradients, robustness in deep networks, and effectiveness with sparse data [16]. Our dataloader had a batch size of 64 and we used the MSE loss between the model output and the true value.



(a) ChessPosEvalNet training loss vs epoch



(b) ChessPosEvalNet valid loss vs epoch

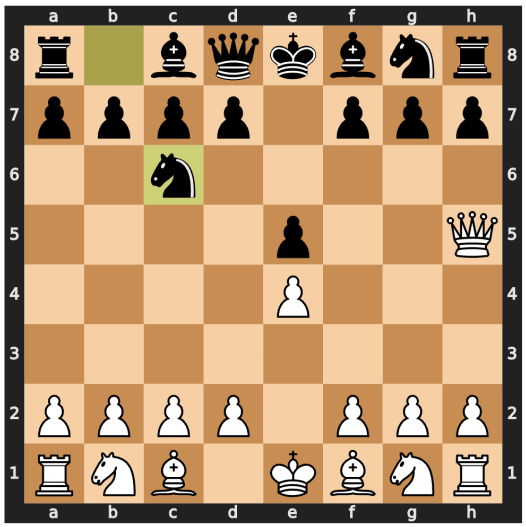
The model's MSE losses at the end of training:

$$\text{train loss} = 0.543 \quad \text{valid loss} = 0.736 \quad \text{test loss} = 0.620$$

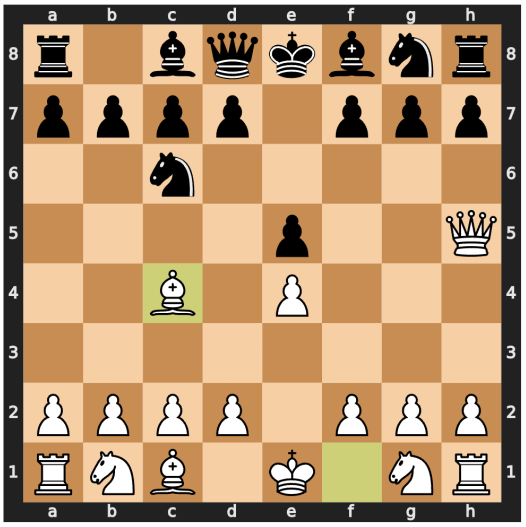
The training code of ChessPosEvalNet can be found in `train_board_eval.py`.

As we can see both the training and validation loss of our model have decreased substantially over time. This is a strong indicator that our model is learning to predict SF scores given a chessboard position. As training progresses, the model is becoming better at fitting the training examples and minimizing the error between its predictions and the actual target values, according to the MSE function. The decrease in validation loss is particularly significant. It indicates that our model is not just memorizing the training data but is also generalizing well to unseen data, which is the validation set that the model has not seen during training. This is a crucial aspect of our analysis of our model, as a model's ability to generalize is a key indicator of its performance on real world, unseen examples. If the validation loss continues to decrease or remains relatively stable as the training progresses, it suggests that our model is not overfitting the training data. Because the test loss is close to the train loss, we can see that the model did not overfit to the train data.

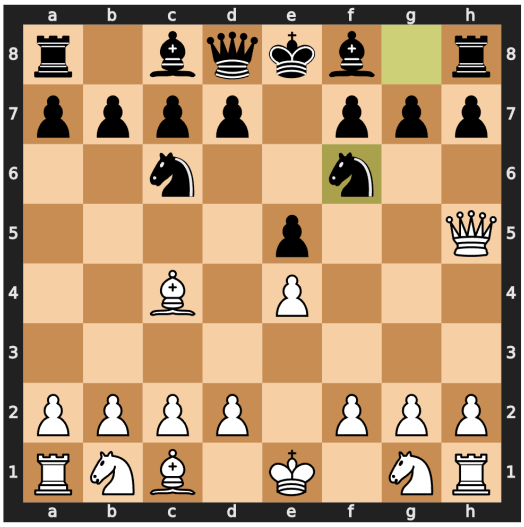
The following is a sequence of game moves. From (a) to (b), we can see that the model predicted an increase in white's strength, which makes sense because the bishop is set up in a position that can let the queen checkmate the black side. From (b) to (c), the black player did not defend against the checkmate, and the model predicted an even higher number in favor of white.



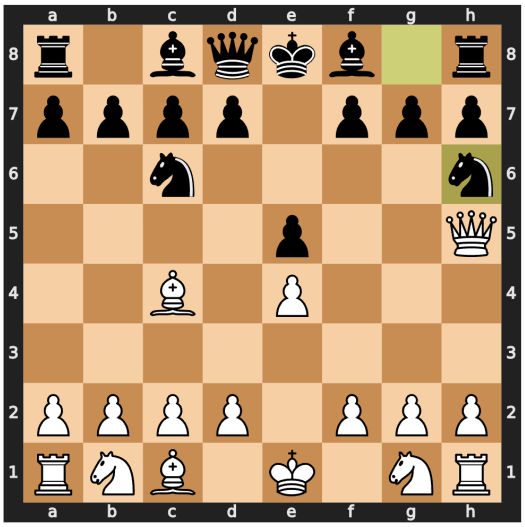
(a)



(b)



(c)



(d)

However, our model has faults. For example from (b) to (d), the black player moved his knight into a square that will guard against checkmate, but the model does not understand that and gave a higher strength score for white. Also in figure 25, we can see that our model is completely off when predicting the SF score of a starting chessboard. This is because during training, the model rarely sees a blank board, if ever. This can also be said for positions near the start of the game. From testing, the model does not predict accurately moves near the end of a game or near the beginning of game, but rather during the mid game, because the majority of training examples come from mid game.

However, our model exhibits certain limitations. For instance, in moves (b) to (d), the black player strategically positioned their knight to defend against potential checkmate threats. However, the model fails to recognize this defensive maneuver and incorrectly assigns a higher strength score to the white side. Furthermore, in Figure 25, our model demonstrates a notable discrepancy in predicting the SF score for a starting chessboard. This discrepancy arises because, during training, the model rarely encounters a blank board, if at all. A similar observation can be made for positions at the beginning and end of a game. Through testing, we have found that the model's predictions are less accurate in these early and late stages of a game. This is primarily because the majority of training examples are derived from positions occurring in the mid-game phase, where the model achieves a higher level of accuracy.

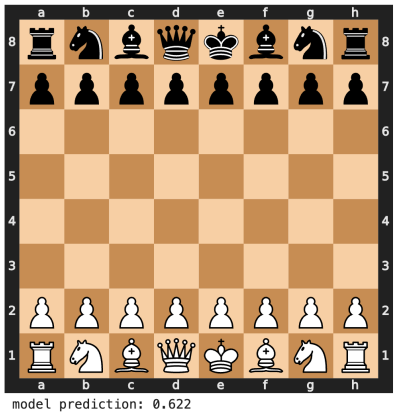


Figure 25: ChessPosEvalNet prediction for a starting chessboard

3.1.7 Hyperparameter Tuning

Having confirmed the effectiveness of our CNN model architecture in extracting critical chessboard features, our next step is to train our ChessNet move predictor model. However, before proceeding, we aim to determine the optimal training hyperparameters.

The hyperparameters that we tested were:

- batch size $\in [32, 512, 2048]$
- learning rate $\in [0.1, 0.001, 0.00001]$

We have defined three types of accuracies to assess and compare our models. Given a batch of data:

1. **From Accuracy** (from_accuracy): This metric evaluates the model's proficiency in correctly predicting the 'from' square of a chess move. It is calculated by dividing the number of correctly predicted 'from' squares by the total number of predictions where the model attempted to predict the 'from' square.
2. **Piece Accuracies** (piece_accuracies): These are individual accuracy scores for each type of chess piece (pawns, rooks, knights, bishops, queens, kings). For each piece type, it computes the ratio of correctly predicted moves (both 'from' and 'to' squares) involving that piece to the total number of moves for which the model correctly predicted the 'from' square of that piece type. If a particular piece type was not encountered during testing, its accuracy is marked as 'False'.
3. **Overall Piece Accuracy** (overall_piece_accuracy): This metric provides an overall evaluation of the model's performance in predicting moves across different chess piece types. It calculates a weighted average of piece accuracies, with the weights determined by the number of moves associated with each piece type.

```
avg loss for epoch 4 is 0.3153
epoch 4 accuracy on valid set:
    Testing piece accuracy of model, over 1 num of epochs
    from accuracy: 0.2389
    piece accuracy: [p,r,n,b,q,k]: [0.7090, 0.2406, 0.3897, 0.2601, 0.1536, 0.5123]
    overall piece accuracy: 0.3894
```

Figure 26: Example output of model performance summary for epoch 4, showing our 3 types of accuracies

The code that tests for these accuracies can be found in `test.py`.

Please note that individual piece accuracy may surpass the overall 'from' accuracy. As illustrated in Figure 26, while the 'from' accuracy is 0.24, the piece accuracy for Pawn type pieces reaches 0.7. This indicates that the model accurately predicts the 'from' square only 24% of the time. However, when the 'from' square involves a pawn, the model correctly predicts the 'to' square with a higher accuracy of 70%.

We conducted training sessions for nine models using two NVIDIA GeForce RTX 2080 Ti (12GB) and evaluated their convergence behavior, from accuracy, and overall piece accuracy. Each model underwent ten epochs of training, or about four hours of training time. Accuracy assessments were performed at the conclusion of each epoch using the validation dataset. The validation set did not impact the training of the models, instead, it was solely used for evaluating and determining the optimal hyperparameters.

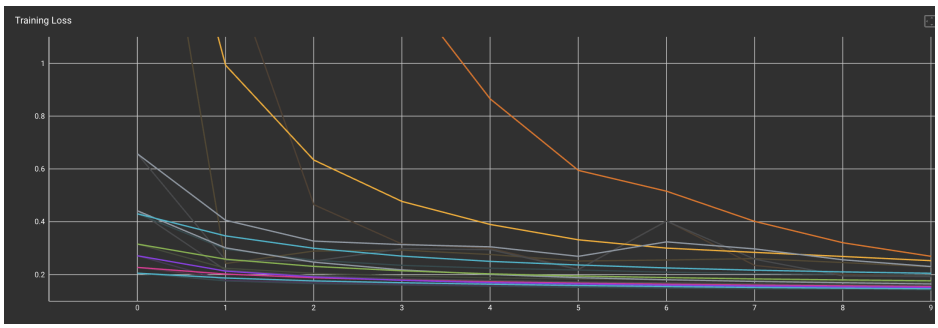


Figure 27: Training Loss vs. Epoch

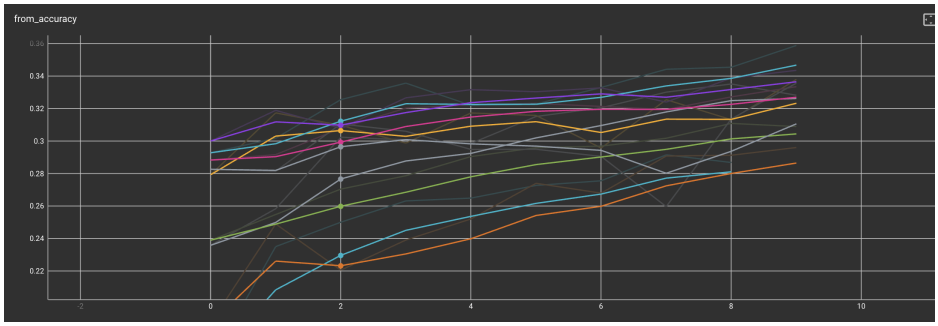


Figure 28: From Accuracy vs. Epoch

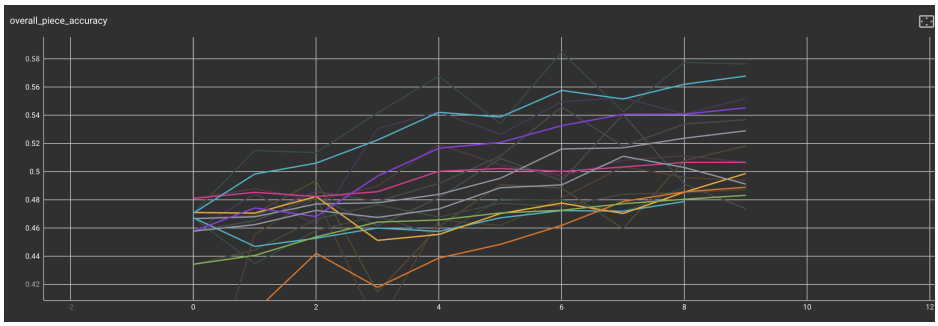


Figure 29: Overall Training Accuracy vs. Epoch

The hyperparameter tuning code can be found in `train_tune_complex.py`.

From the tuning results, we decided to go with the following training hyperparameters:

$$\text{batch size} = 32 \qquad \text{learning rate} = 0.001$$

We selected a batch size of 32 and a learning rate of 0.001 for training due to their ability to yield a low loss, high from accuracy, and an overall high training accuracy at the end of 10 epochs.

3.1.8 CNN Model Training and Results

TODO, talk about last paragraph on page 1 in ConvChess and the rest of that section. good transition to gen. algo. section.
We trained our ChessNet model on one NVIDIA GeForce RTX 2080 (8GB) for 75 epochs or about 24 hours, with a batch size of 32 and a learning rate of 0.001.

We used two specific loss functions to train our ChessNet model for move prediction, namely, the CE Loss for 'from' squares (loss_from) and the CE Loss for 'to' squares (loss_to). In the forward pass of the model, it predicts move information for both 'from' and 'to' squares simultaneously, as we can see in figure 15, the output is of shape $2 \times 8 \times 8$. To calculate these losses, we compare the model's predictions for 'from' squares and 'to' squares with the ground truth labels. loss_from measures the dissimilarity between the predicted 'from' square and the actual 'from' square in the ground truth. Similarly, loss_to measures the disparity between the predicted 'to' square and the true 'to' square. During training, we optimize our model by minimizing the sum of these two losses, which encourages the model to make accurate predictions for both 'from' and 'to' squares simultaneously. This dual loss approach ensures that our ChessNet learns to predict complete chess moves effectively.

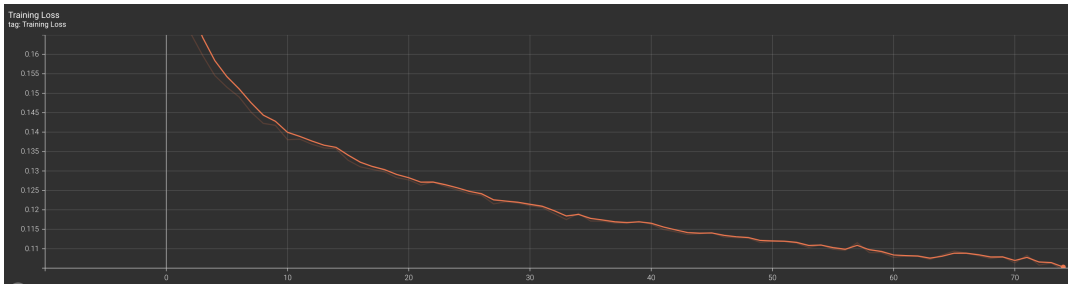


Figure 30: ChessNet Training - Loss vs. Epoch

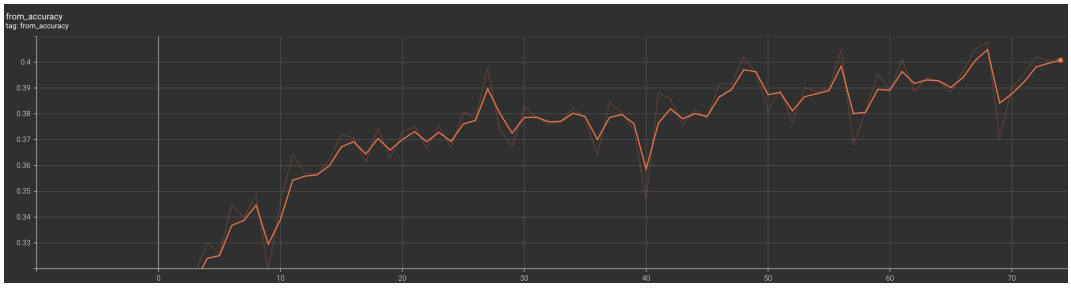


Figure 31: ChessNet Training - From Accuracy vs. Epoch

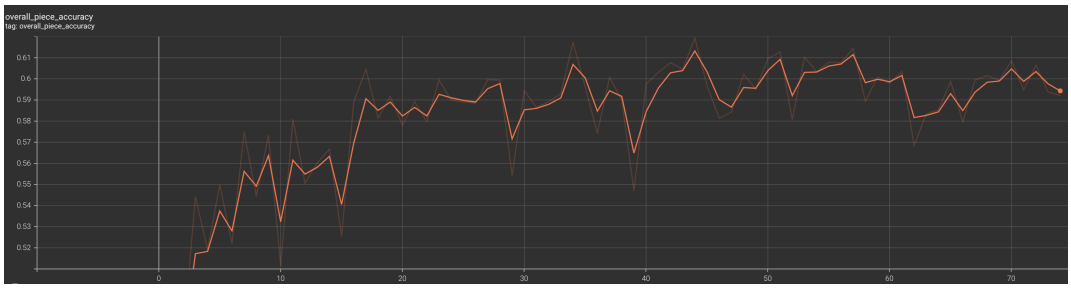


Figure 32: ChessNet Training - Overall Piece Accuracy vs. Epoch

Note: for figures 31 and 32, we tested the model's *from accuracy* and *overall piece accuracy* at the end of each epoch on a validation set, separate from the train set.

At the end of training, we tested our model's performance on a test set:

- from accuracy: 0.385
- overall piece accuracy: 0.589
  - pawns accuracy: 0.863
  - rooks accuracy: 0.376

- knights accuracy: 0.607
- bishops accuracy: 0.545
- queens accuracy: 0.316
- kings accuracy: 0.653

From the results above, we can see that the model did not overfit to the training data because the results it achieved on the test data are very close to its results on the train/valid datasets.

An evident observation from our results is the substantial variation in the model's accuracy based on the piece type. For long range pieces such as rooks and queens, the model succeeds in predicting the correct move only around 33 percent of the time. In contrast, for short range pieces like pawns, knights, and kings, the model demonstrates a significantly higher accuracy, predicting the correct move approximately 70 percent of the time. This divergence in accuracy aligns with our expectations, given that CNNs excel at detecting and learning local features. The superior performance with short-range pieces is likely attributed to the more constrained and local nature of their movement patterns, making them more predictable to CNNs.

It's essential to consider the context in which we evaluated our chess model. We compared move predictions with real-life moves, and the model achieved relatively high accuracy for certain piece types, demonstrating its ability to grasp elite player strategies. However, it's important to note that low accuracy or mispredictions by the engine doesn't necessarily indicate a bad move prediction. Efficiently measuring move quality would require building a mini-max tree and evaluating all options, which is impractical. Additionally, quantifying the 'closeness' of two moves remains challenging. Thus, our model's evaluation primarily relies on comparing its predictions with moves from real-life games.

TODO add examples of move predictions
TODO play it against stockfish (lowest version possible)

3.2 Implementation of the Genetically Trained Heuristic

This component of our chess engine is comprised of a genetically trained heuristic that is then employed by a Minimax algorithm in order to determine the best possible moves. The main inspiration for how we implemented this component is from a previous paper jointly written by Bar Ilan university researchers Moshe Koppel, Nathan Netanyahu, Tel Aviv PhD graduate Eli David and Jaap van Hendrik of Leiden university. [8]

The idea behind using a genetically trained heuristic is as follows:

1. The heuristic is comprised of many different hand crafted features. Each of these hand crafted features evaluates a different component of the chess position, such as king safety, number of isolated pawns, or even number of outpost squares for a knight.
2. The relative weighting of each of these features is crucial, and will ultimately determine the usefulness of the heuristic. For example, it is very important for a heuristic to value a queen over a pawn, or the chess engine will not perform well. Unfortunately, it is difficult to predict what exactly the relative weights of each of these various hand crafted features should be.
3. In order to determine the relative weights, we run a genetic algorithm that will give us the optimal weights, and therefore a very well tuned hand crafted heuristic.

3.2.1 Crafting the Evaluation Function

As such, before running the genetic algorithm in order to determine the optimal weights, we first designed a heuristic with hand crafted features. The choice of hand crafted features was made according to those commonly needed in order to implement primitive chess engines. In total, we ended up making 37 different features. Below is a table consisting of the name of each feature, as well as an explanation of to what it refers. Note that all of the features are evaluated from one side's perspective, either white or black (so for example Pawn Value evaluated from White's perspective refers to the total number of White pawns:

No	Feature Name	Description of Feature
1	Pawn Value	Counts the total number of pawns
2	Knight Value	Counts the total number of knights
3	Bishop Value	Counts the total number of bishops
4	Rook Value	Counts the total number of rooks
5	Queen Value	Counts the total number of queens
6	Passed Pawn Number	Counts the total number of pawns that have neither enemy pawns directly in front of nor and adjacent to them
7	Doubled Pawn Number	Counts the total number of times there are pawns of our side that occupy the same file
8	Isolated Pawn Number	Counts the total number of pawns that have no adjacent friendly pawns on either side
9	Backward Pawn Number	Counts the total number of backward pawns (pawns that are no longer defensible by own pawns as they are too advanced. And if pushed will be lost)
10	Passed Pawn Enemy King Distance	A metric of how far away the king is from a passed pawn (calculated using Manhatten distance).
11	Number of Center Pawns	Counts the total number of pawns found in the center.
12	Knight Mobility	Checks the total number of squares controlled by the knights
13	Knight Outpost Number	Counts the total number of knights on an outpost square. Outpost will be defined as a knight in the center or on the opponent's half of the board, defended by an own pawn, and no longer attackable by opponent pawns at all.
14	Bishop Mobility	Counts the total number of squares controlled by the bishops
15	Bishop Pair	Checks if we have two bishops
16	Rook Attack Weak Pawn Open Column	Counts the total number of rooks on the same line as a backward or isolated pawn of the enemy.
17	Rook Connected	Checks to see if the rooks are defending each other
18	Rook Mobility	Determines the total number of squares controlled by the rooks
19	Rook Open File	Counts the total number of rooks placed on files that do not contain any pawns (either ours or the opponents) on it
20	Rook Semi-Open File	Counts the total number of rooks placed on files that do not contain any pawns (only ours) on it
21	Rook Attack King Adj File	Counts the total number of rooks on the adjacent files of the enemy king
22	Rook Attack King File	Counts the total number of rooks on the same file as the enemy king
23	Rook Behind Passed Pawn	Counts the total number of our own rooks that are placed behind enemy passed pawns.
24	Queen Mobility	Counts the total number of squares controlled by the queen(s)
25	King Number Friendly Pawns	Counts the total number of pawns belonging to us that are found on the same file as the king
26	King Number Friendly Pawns Adjacent	Counts the total number of pawns belonging to us that are found on adjacent files to the king
27	King Friendly Pawn Advanced	Counts the total number of pawns in front of the king that are advanced by at least one
28	King Number Enemy Pawns	Counts the total number of enemy pawns that are on the same file as the king and are within four ranks of the king.
29	King Number Enemy Pawns Adjacent	Counts the total number of enemy pawns that are on adjacent files to the king and are within four ranks of the king.
30	Pressure against King	Calculates the distance of our king from the enemy pieces (distance is calculated using the Manhatten distance).
31	Checkmate	Checks if the position is checkmate
32	Bishop Piece Table	A feature that ascribes a set value to each square on the chess board. Squares that are generally more favourable for the bishop to be on are assigned a higher value.
33	Knight Piece Table	A feature that ascribes a set value to each square on the chess board. Squares that are generally more favourable for the knight to be on are assigned a higher value.
34	Rook Piece Table	A feature that ascribes a set value to each square on the chess board. Squares that are generally more favourable for the rook to be on are assigned a higher value.
35	Queen Piece Table	A feature that ascribes a set value to each square on the chess board. Squares that are generally more favourable for the queen to be on are assigned a higher value.
36	Pawn Piece Table	A feature that ascribes a set value to each square on the chess board. Squares that are generally more favourable for the pawn to be on are assigned a higher value.
37	King Piece Table	A feature that ascribes a set value to each square on the chess board. Squares that are generally more favourable for the king to be on are assigned a higher value. Note that in the middlegame, it is better to tuck the king away so that it is safe, while in the endgame it is important to centralize the king. As such, we assign different piece tables for the king according to the number of pieces still on the board.

Table 1: showing the descriptions of every parameter used in the evaluation function

3.2.2 Choosing a Dataset

In order to train the genetic algorithm, we needed to choose a dataset of chess games to work with. We chose to use the publicly available Lichess Elite database, which features all the games played on the website Lichess.com from December of 2022 where both players were above a 2000 Lichess rating threshold.

We then filtered the dataset: selecting only 1000 games where the Lichess rating of both players was greater than or equal to 2500 and white had won the game. From this, we took a single random position from each game and stored both the position and the move that was played by the top player in that position. This is what is what we will ultimately use as a measure of fitness for each one of the organisms - the total number of positions in which they are able to predict the same move as that made by the top player.

```
('rnbq1rk1/5ppp/pp2pn2/2b5/2B5/4PN2/PP1NQPPP/R1B2RK1 w - - 0 10', 'a3')
```

Figure 33: Random position represented in FEN format and the move chosen to be played by the top player in that position

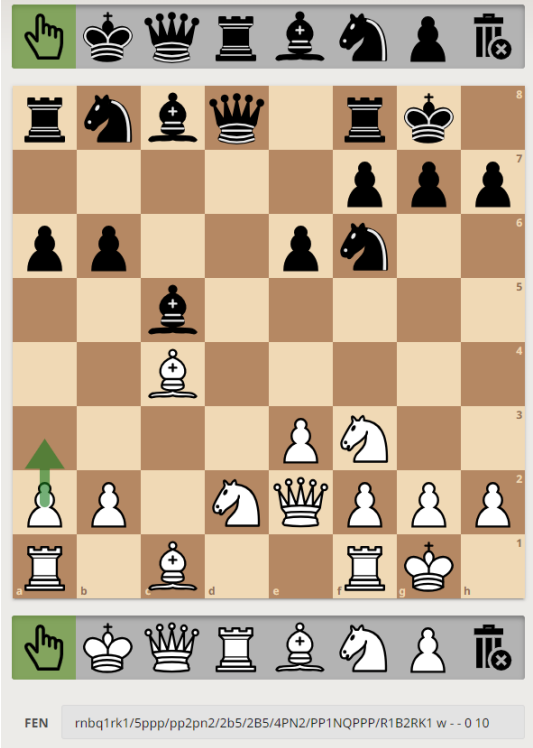


Figure 34: A visual representation of the Random position and the move chosen to be played by the top player in that position

3.2.3 Running the Genetic Algorithm

Now that we have both a dataset and evaluation function with hand crafted features, we were then able to apply the following genetic algorithm in order to determine the optimal weights for each of the features:

1. First we generate an initial population of 100 random binary strings of size 243 bits each.
2. Each of these 243 bit strings represents the "chromosome" of these organisms. Each of these chromosomes contains "genes" - bit substrings that when converted from a binary format to decimal represent the weighting of a specific parameter in the evaluation function for that organism. For example, the first 11 bits of every chromosome encode for the weighting of the "Knight Value" feature in the evaluation function of the organism.
3. As such, we now have 100 organisms that each will have their own evaluation function that behaves in its own way. For each organism, we now run its evaluation function on each of the training positions from our

dataset. We record the "fitness" of the organism (the total number of positions that the origanism guessed the correct move of).

4. Now that we have the fitness of all of the organisms, we can use this as a "selection pressure" for what organisms survive for the next generation. In order to determine the next population of organisms, we now apply the following mechanisms:

    - Elitism: The top ten percent of organisms (ranked according to their fitness level) are copied over to the next generation. This is the only mechanism that doesn't involve selecting parents to mate/mutate and generate offspring. In other words, the reproduction is asexual and not sexual.

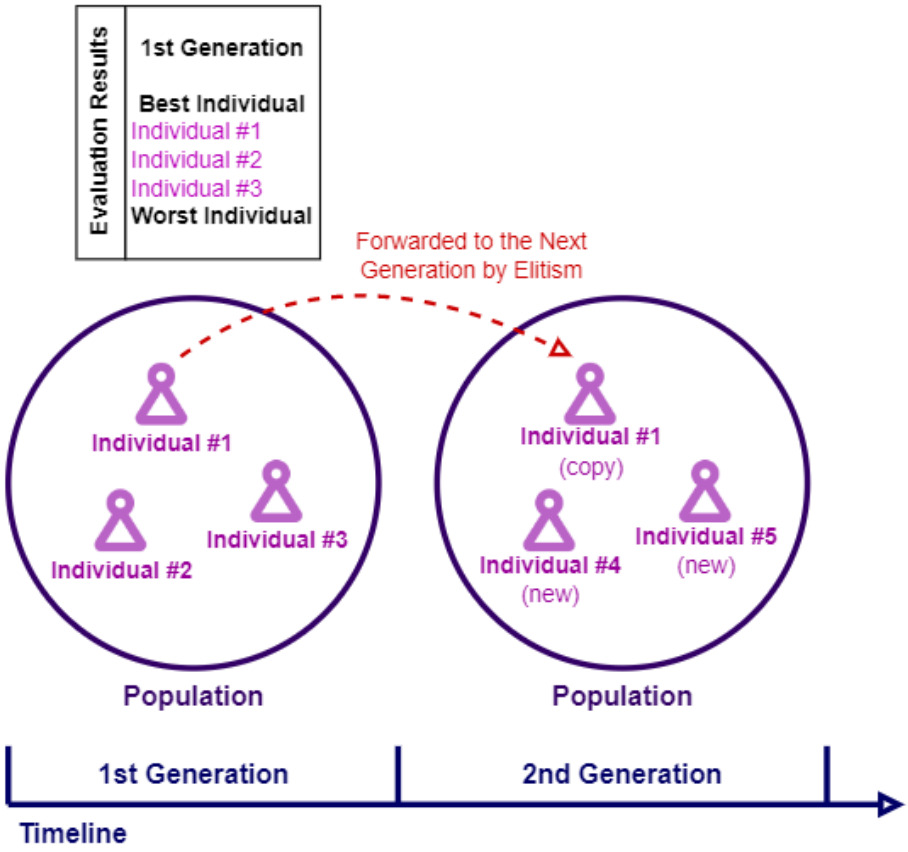
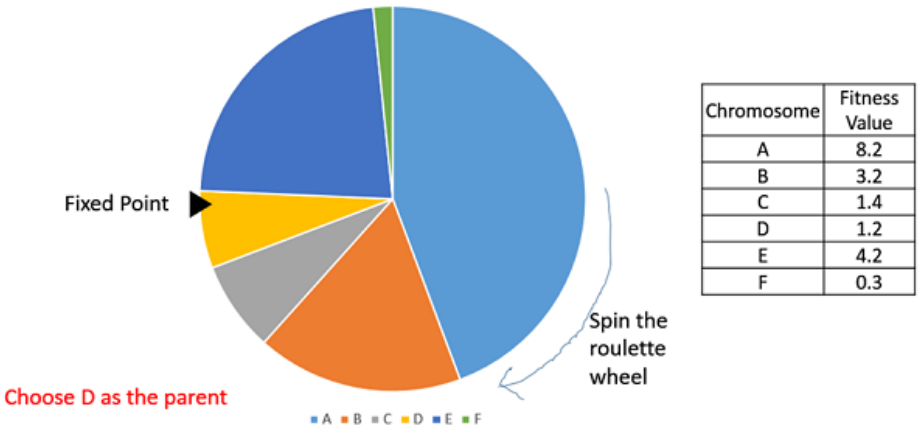


Figure 35: Simple example illustrating Elitism [6]

- Fitness Proportional Selection: In this method of parent selection, every individual can become a parent with a probability proportional to its fitness. In order to implement this, we used a method known as roulette wheel selection, wherein each of the organisms harbours a share of the wheel proportional to its fitness value. By "spinning" the wheel, we choose which parents to perform sexual reproduction. Naturally, parents with greater fitness values occupy more space on the wheel and so have a higher likelihood of being chosen. The algorithm describing this method is described below:
    - Calculate S: the sum of all of the fitnesses within the population.
    - Generate a random number between 0 and S.
    - Starting from the top of the population, keep adding fitnesses to the partial sum P, while P¡S.
    - The individual for which P exceeds S is the chosen individual. [7]



Chromosome	Fitness Value
A	8.2
B	3.2
C	1.4
D	1.2
E	4.2
F	0.3

Figure 36: Simple example illustrating Roulette Wheel Selection [7]

- Uniform Crossover: Once we have chosen two parents, we perform a form of crossover between the two chromosomes, wherein we make it a 50% probability for a given gene to be found in each of the two children organisms that will be part of the new population of organisms.

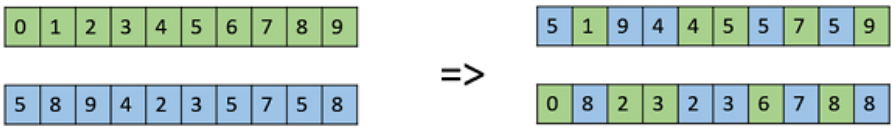


Figure 37: Simple example illustrating Uniform Crossover [7]

- Bit Flip Mutation: After performing uniform crossover, we employ a form of mutation whereby we cause each of the binary digits making up the child chromosome to potentially flip, (from a 0 to 1 or 1 to 0), with a probability proportional to the mutation rate (a hyperparameter that we defined in accordance to that chosen in the paper we based our work on [8].

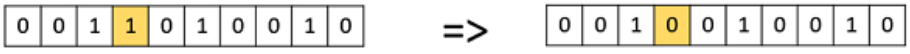


Figure 38: Simple example illustrating Bit Flip Mutation [7]

5. Thus, we perform the above mentioned process for a period of 200 generations, before selecting the organism from the final population with the greatest level of fitness. This is what we will designate to be the "optimally trained" organism.

3.2.4 Simulating an "Optimal Organism"

Before running the genetic algorithm, we decided to try and gain an understanding of whether the evaluation function that we had created was reasonable. As such, we got a Chess Candidate Master (Aron Klevansky) to evaluate what the weights of each of the parameters should be. The weights chosen by the Candidate Master are shown below:

No	Feature Name	Value Assigned by Candidate Master
1	Pawn Value	100
2	Knight Value	300
3	Bishop Value	300
4	Rook Value	500
5	Queen Value	900
6	Passed Pawn Number	63
7	Doubled Pawn Number	40
8	Isolated Pawn Number	30
9	Backward Pawn Number	50
10	Passed Pawn Enemy King Distance	0
11	Number of Center Pawns	20
12	Knight Mobility	13
13	Knight Outpost Number	63
14	Bishop Mobility	15
15	Bishop Pair	63
16	Rook Attack Weak Pawn Open Column	50
17	Rook Connected	20
18	Rook Mobility	7
19	Rook Open File	63
20	Rook Semi-Open File	30
21	Rook Attack King Adj File	0
22	Rook Attack King File	0
23	Rook Behind Passed Pawn	0
24	Queen Mobility	2
25	King Number Friendly Pawns	63
26	King Number Friendly Pawns Adjacent	63
27	King Friendly Pawn Advanced	50
28	King Number Enemy Pawns	30
29	King Number Enemy Pawns Adjacent	30
30	Pressure against King	0
31	Checkmate	4000
32	Bishop Piece Table	1
33	Knight Piece Table	1
34	Rook Piece Table	1
35	Queen Piece Table	1
36	Pawn Piece Table	1
37	King Piece Table	1

Table 2: showing the parameter values of the organism created by the Candidate Master

When running this handcrafted choice of weights against the dataset of random positions, we found that the organism guessed the correct result 200 (out of a total of 1000) times. This gives a success rate of 20%. As such, we will see if running the algorithm manages to converge to similar parameters as those chosen by the Candidate Master,and whether it manages to obtain higher levels of fitness.

3.3 Results after Training

We ran the genetic algorithm on a starting population of 75 organisms for a total of 150 generations. Additionally, we chose to use the following hyper-parameters:

1. crossover rate = 0.75
2. mutation rate = 0.005
3. elitism rate = 0.1

After 4 days of running on the Lambda Technion server, we obtained the following results:

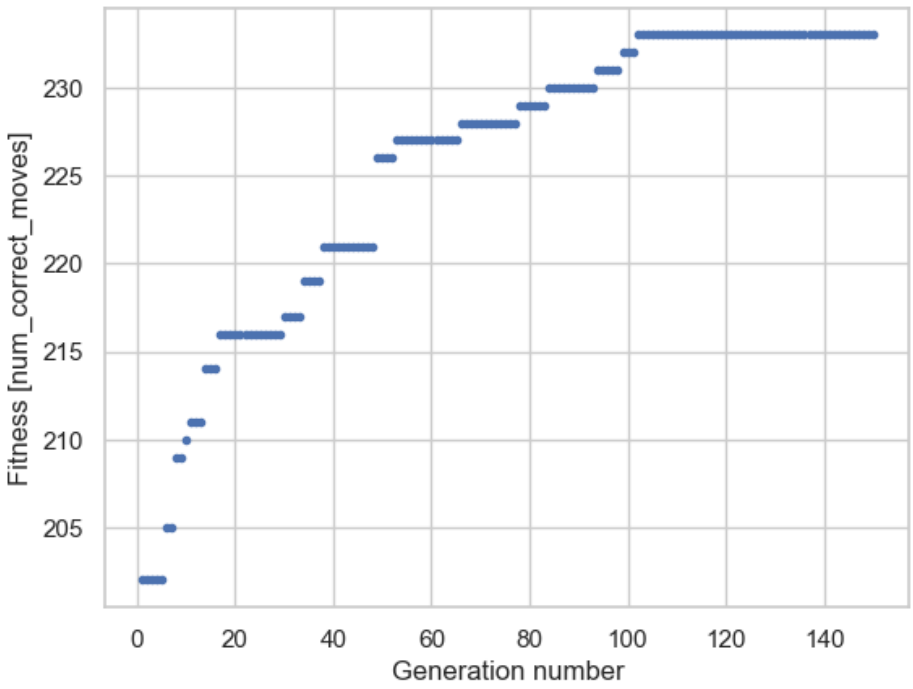


Figure 39: showing the fitness of the fittest organism in the population as a function of the generation number

From Figure 39, we see that initially the population improves in fitness rapidly, as seen from the steep gradient that the graph makes over the first 20 generations. However, eventually the graph stabilises at a threshold asymptotic value of 233 correct moves, from which it is difficult for it to improve further. In other words, it reaches a local maximum (as we expect to achieve from a genetic algorithm).

As such, we see that our algorithm achieves an accuracy of $\frac{233}{1000} = 0.233$. In other words, from a random set of 1000 chess positions played by chess players with Lichess ratings greater than 2500, the best trained organism guesses the correct move 25% of the time!

In order to illustrate how the population of organisms change in fitness over time, we present Figures 40, 41, 42 and 43 below:

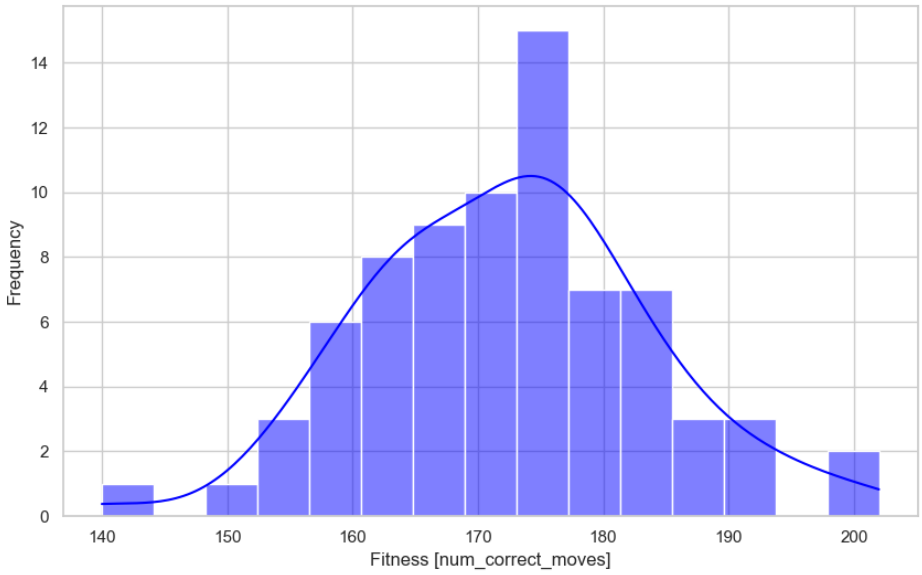


Figure 40: histogram of the population fitness of the 75 organisms after 1 generation

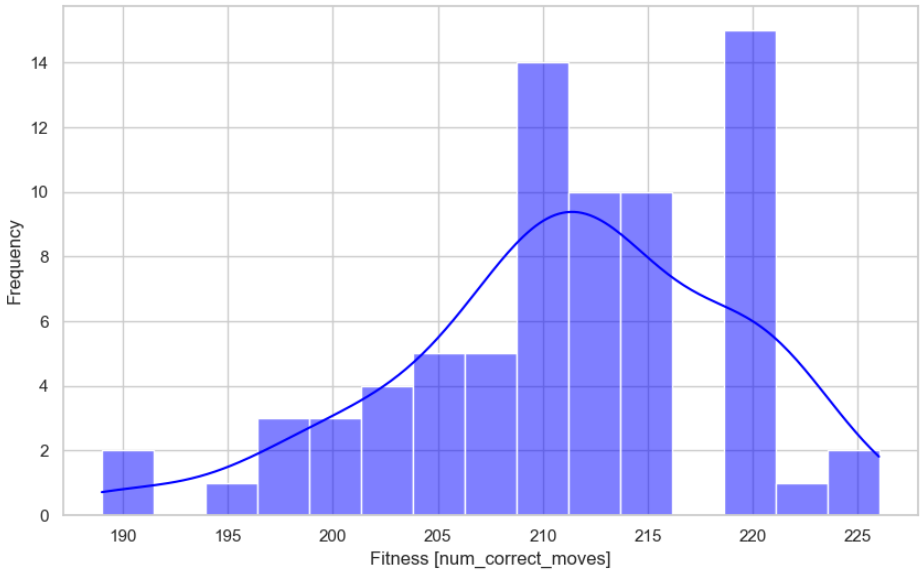


Figure 41: histogram of the population fitness of the 75 organisms after 50 generations

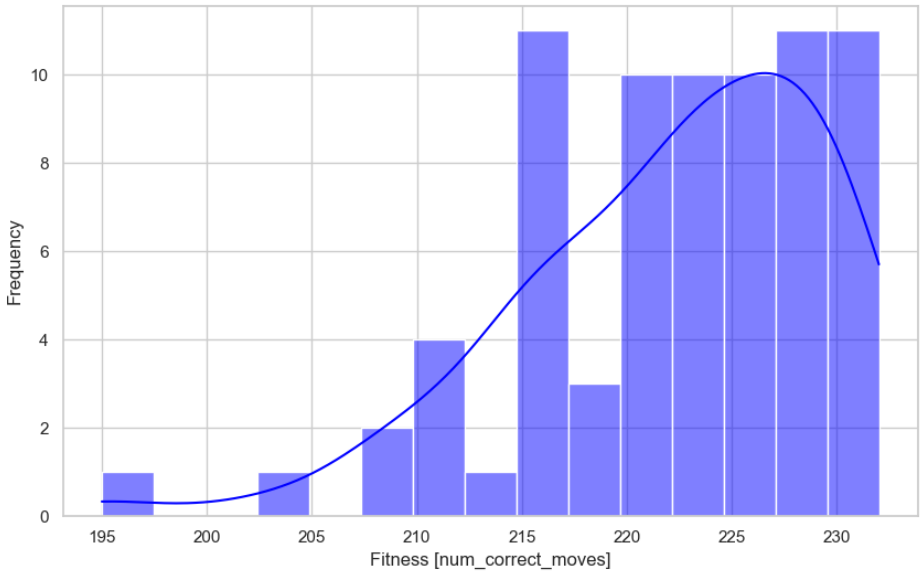


Figure 42: histogram of the population fitness of the 75 organisms after 100 generations

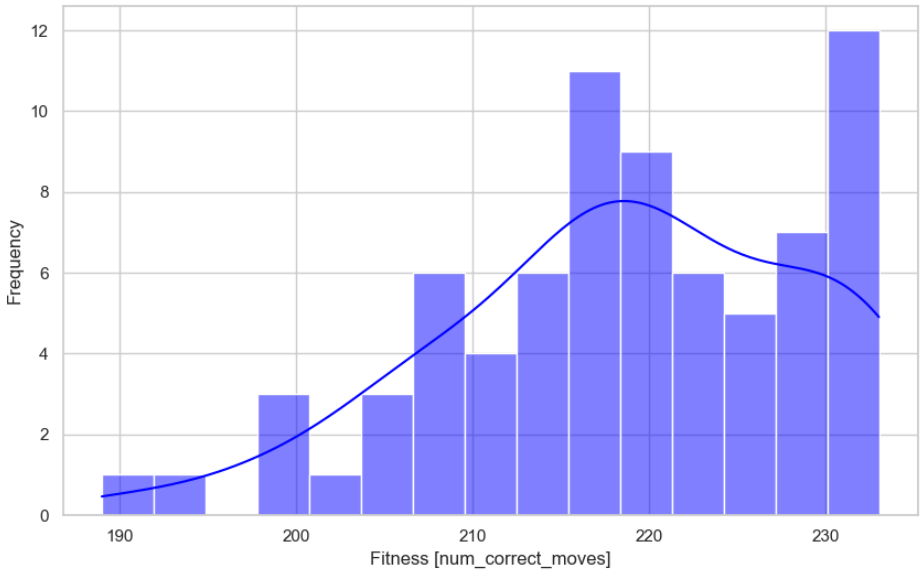


Figure 43: histogram of the population fitness of the 75 organisms after 150 generations

As such, we see that although the initial population is random (and therefore represents a normal distribution) as seen in Figure 40, over time the algorithm selects for fitter organisms as a selection pressure, thereby causing the distribution to gradually increase in average fitness level. Moreover, we see that the distribution becomes skewed to the right (organisms with the greatest fitness end up having the greatest frequency). This is most prominently observed in Figure 42. The reason for this skewing is mainly due to the elitism property of the genetic algorithm.

The following table is used to illustrate the parameter values of the best trained organism:

No	Feature Name	Value of Parameters of Best Trained Organism
1	Pawn Value	100
2	Knight Value	1339
3	Bishop Value	1834
4	Rook Value	2004
5	Queen Value	1968
6	Passed Pawn Number	12
7	Doubled Pawn Number	53
8	Isolated Pawn Number	36
9	Backward Pawn Number	8
10	Passed Pawn Enemy King Distance	5
11	Number of Center Pawns	20
12	Knight Mobility	13
13	Knight Outpost Number	9
14	Bishop Mobility	33
15	Bishop Pair	13
16	Rook Attack Weak Pawn Open Column	62
17	Rook Connected	19
18	Rook Mobility	34
19	Rook Open File	8
20	Rook Semi-Open File	30
21	Rook Attack King Adj File	24
22	Rook Attack King File	57
23	Rook Behind Passed Pawn	23
24	Queen Mobility	11
25	King Number Friendly Pawns	45
26	King Number Friendly Pawns Adjacent	2
27	King Friendly Pawn Advanced	30
28	King Number Enemy Pawns	3
29	King Number Enemy Pawns Adjacent	2
30	Pressure against King	38
31	Checkmate	15662
32	Bishop Piece Table	12
33	Knight Piece Table	26
34	Rook Piece Table	4
35	Queen Piece Table	4
36	Pawn Piece Table	41
37	King Piece Table	42

Table 3: showing the parameter values of the best trained organism

From Table 3, we see that as a result of the evolutionary process, the current organism has managed to pick up certain domain specific knowledge about the field of chess. For example, we see that the Checkmate parameter is 15662 - the highest valued parameter by a large extent. This illustrates that the organism understands checkmate to be the most important thing to achieve (more important than absolutely anything else). In addition, while not perfectly able to understand the specific values of the pieces, the organism does intuitively understand that rooks and queens are worth more than bishops and knights, which in turn are worth more than pawns. Moreover, the organism seems to have relatively high values for parameters such as "pressure against king", "king piece table" and "king number friendly pawns". This suggests that the organism highly prioritises king safety - which is of course one of the most important features in chess. From the obtained values, we suspect that given more training positions, increased generation number and increased population size, the organism's parameters would converge to results that are similar to those seen in the paper on genetic algorithms for evolving computer chess programs [8].

When comparing the values and fitness rates obtained by the best trained organism with those of the organism

handmade by the Candidate Master, we see that the Candidate Master's organism does much better than an average randomly generated organism (with a fitness of 20% of positions correct in comparison to the average 17% of positions correct).

However, it is difficult to compare the Candidate Master's organism with the best trained organism by directly comparing parameter values. This is because multiple parameters influence each other. For example, although the value of the queen may be greater in one organism, the value of mobility of the queen may be less. This then compensates for the "exaggeratedly" large value of the queen. As such, we believe the best way to learn about how well the best trained organism performs in comparison to the Candidate Master organism is to have them play against each other:

4 Experimental Methodology

TODO

5 Experimental Results and Analysis

TODO

6 Conclusion

TODO

References

[1] Activation functions in deep learning: A comprehensive survey and benchmark. https://arxiv.org/pdf/2109.14545.pdf.

[2] Chess evaluations dataset. https://www.kaggle.com/datasets/ronakbadhe/chess-evaluations.

[3] Chess games dataset. https://www.kaggle.com/datasets/arevel/chess-games.

[4] Cs231n convolutional neural networks for visual recognition. https://cs231n.github.io/convolutional-networks/.

[5] El ajedrecista. https://www.chessprogramming.org/El_Ajedrecista.

[6] Elitism. https://www.baeldung.com/cs/elitism-in-evolutionary-algorithms: :text=The

[7] Explanations of genetic algorithms. https://www.tutorialspoint.com/genetic$_a lgorithms/genetic_a lgorithms_p arent_s election.$

[8] Genetic algorithms for evolving computer chess programs. https://arxiv.org/pdf/1711.08337.pdf.

[9] How to build a 2000 elo chess ai with deep learning - youtube. https://youtu.be/aOwvRvTPQrs?si=ArJ03hf6-HwBj2fQ.

[10] Introduction to artificial neural networks. https://www.kdnuggets.com/2019/10/introduction-artificial-neural-networks.html.

[11] An introduction to convolutional neural networks. https://arxiv.org/abs/1511.08458.

[12] Kasparov vs deepblue. https://spectrum.ieee.org/how-ibms-deep-blue-beat-world-champion-chess-player-garry-kasparov.

[13] Mechanical turk. https://daily.jstor.org/amazons-mechanical-turk-has-reinvented-research/.

[14] Predicting moves in chess using convolutional neural networks. http://cs231n.stanford.edu/reports/2015/pdfs/ConvChess.pd

[15] Stockfish elo over time. https://www.chessprogramming.org/Stockfish.

[16] Survey of optimization methods from a machine learning perspective. https://arxiv.org/abs/1906.06821.

[17] Teaching deep convolutional neural networks to play go. https://arxiv.org/abs/1412.3409.