Determining the Optimal Tradeoff for building Chess Playing Agents: Genetically Tuned vs. Neural Features

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

Niv Ostroff
Student ID: 212732101
nivostroff@campus.technion.ac.il

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]

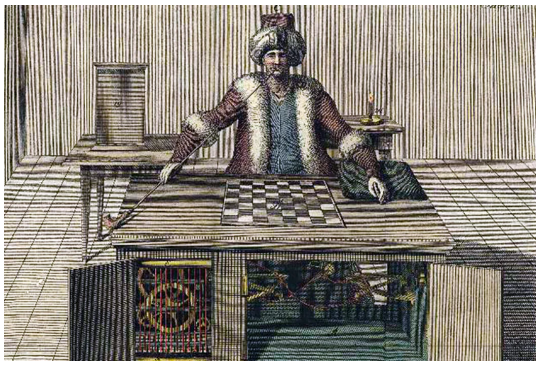


Figure 1: showing an illustration of the infamous "Mechanical Turk"

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]

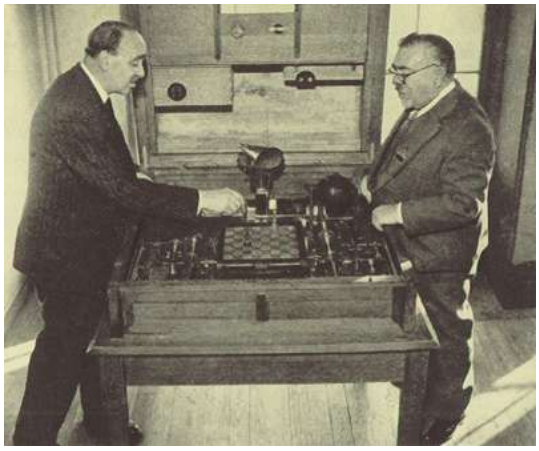


Figure 2: Leonardo Torres with the El Ajedrecista machine

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 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]

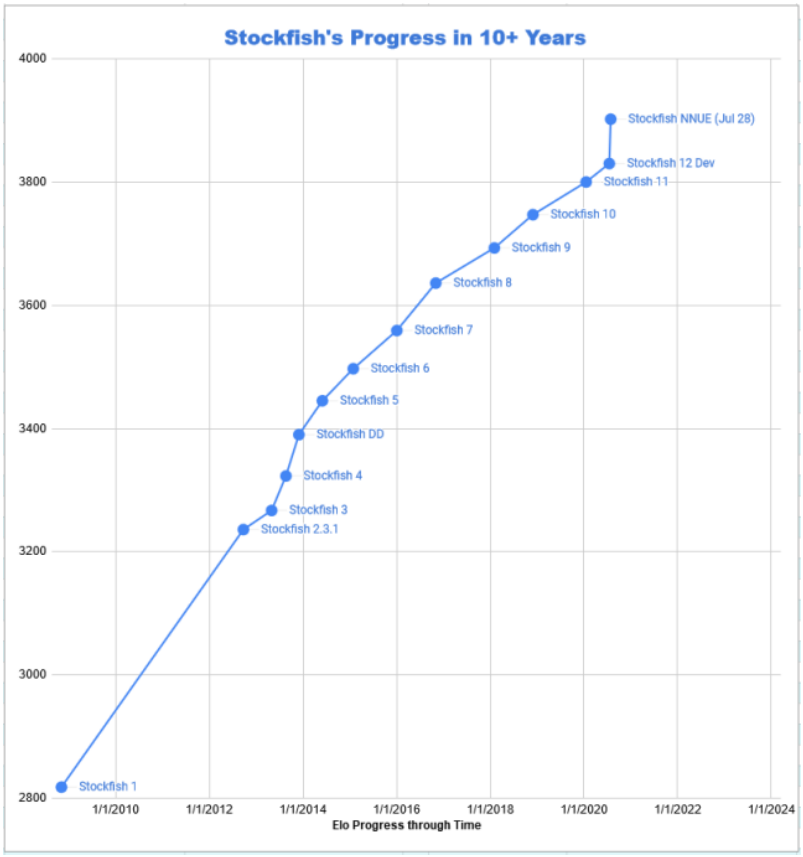


Figure 4: Stockfish ELO change over time. Note the jump in 2020 with the introduction of a Neural Network!

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 genetically tuned 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 trade-off between these two different components of the chess engine.

This trade-off works by adding a feature to the evaluation function that represents the CNN. Therefore, when iterating through all the legal moves, there is a positive score added to the final evaluation of the resulting position if the chosen move is the same as the move predicted by the CNN. This positive score is scaled by the value of lambda. If lambda is 1, the positive score added is so high that the evaluation function only takes into consideration the move suggested by the CNN. On the other hand, if the value of lambda is 0, then only the genetically tuned handcrafted features are considered in the evaluation process and the CNN is neglected.

By modifying the hyperparameter lambda to be some value between 0 and 1, we can determine the optimal trade-off between both the CNN and the genetically tuned handcrafted features.

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]

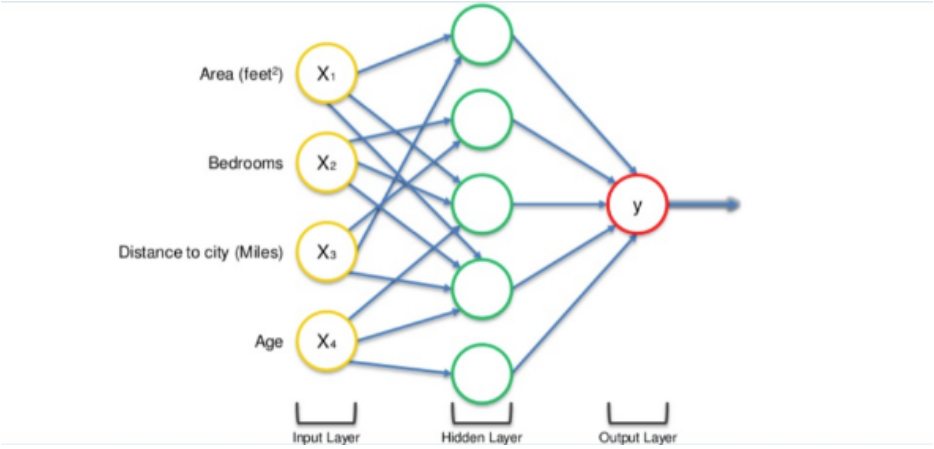


Figure 5: A neural network with a hidden layer

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.

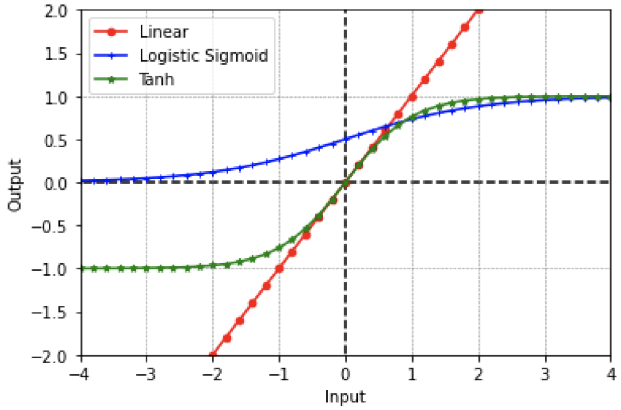


Figure 6: An illustration of Linear, Logistic Sigmoid and Tanh AFs.

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.

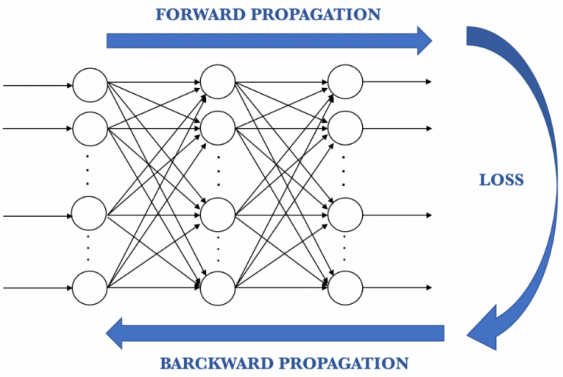


Figure 7: Training a Neural Network

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].

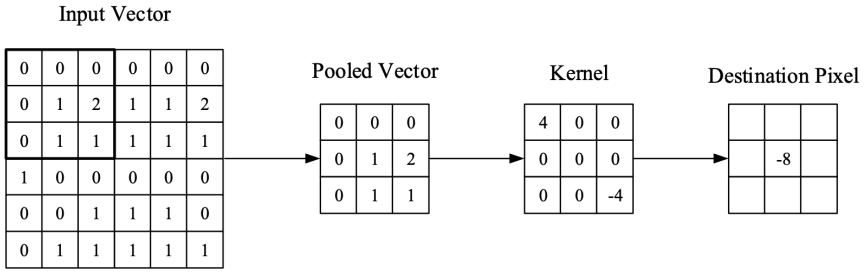


Figure 8: A visual representation of a convolutional layer. The center element of the kernel is placed over the input vector, of which is then calculated and replaced with a weighted sum of itself and any nearby pixels.

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].

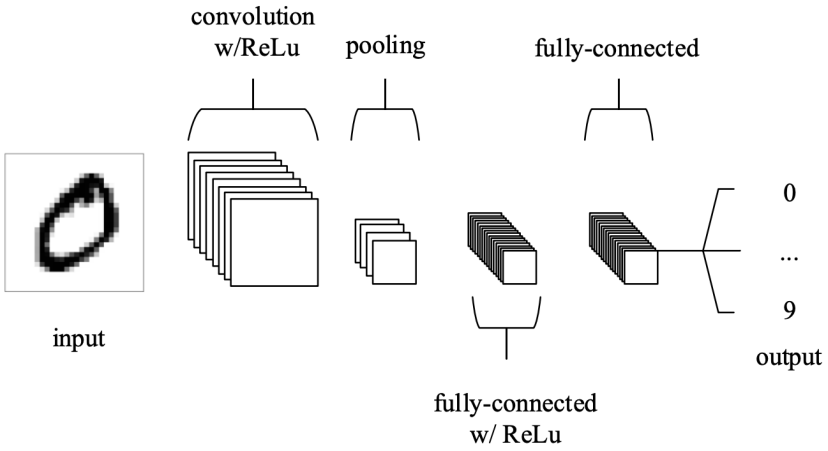


Figure 9: A simple CNN architecture example, comprised of just five layers

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

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

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.

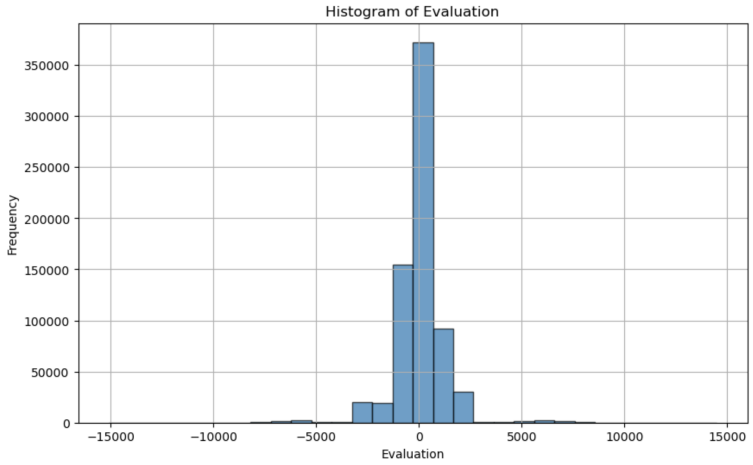


Figure 10: Histogram of 'Evaluation' in the Chess Evaluations dataset

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.

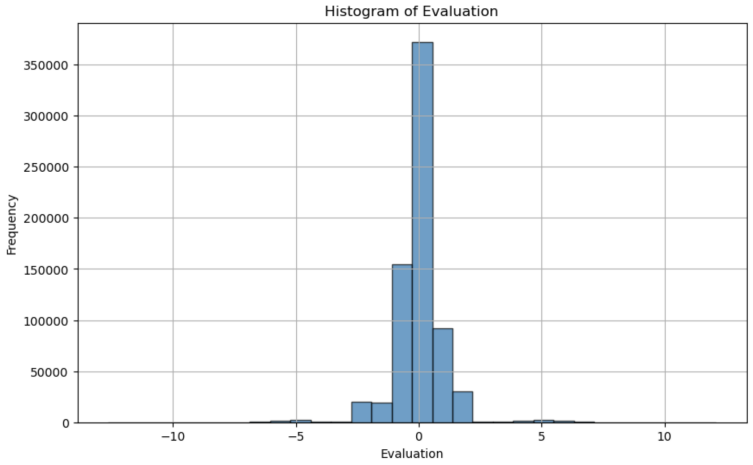


Figure 11: Histogram of 'Evaluation' in the Chess Evaluations dataset after z-score normalization

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?

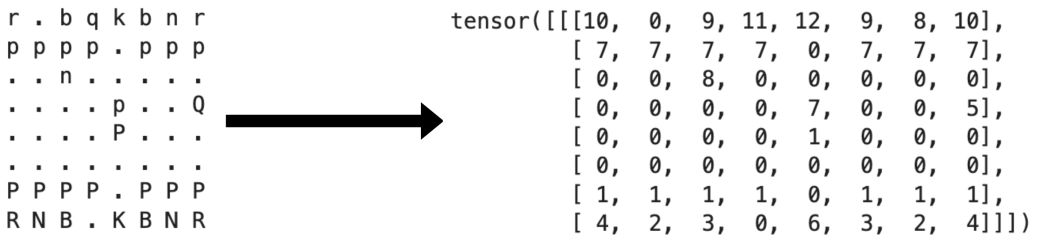


Figure 12: Unique piece type assignment board to tensor conversion example

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.

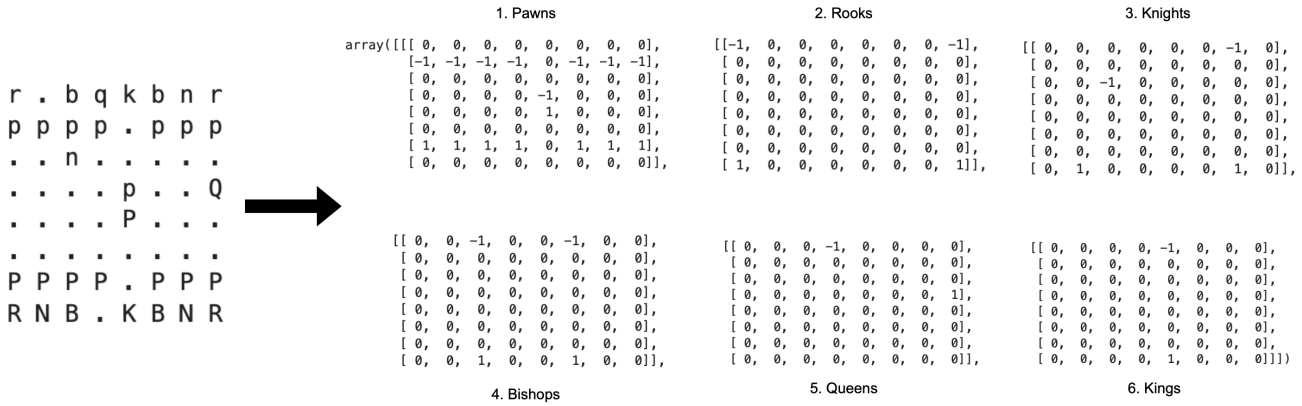


Figure 13: 6-layer representation of a chess board example

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:

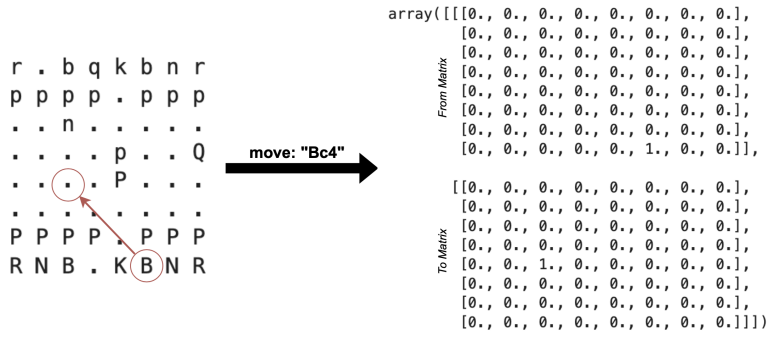


Figure 14: "Bc4" (Bishop to square c4) move representation using from and to matrices example

- 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

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.

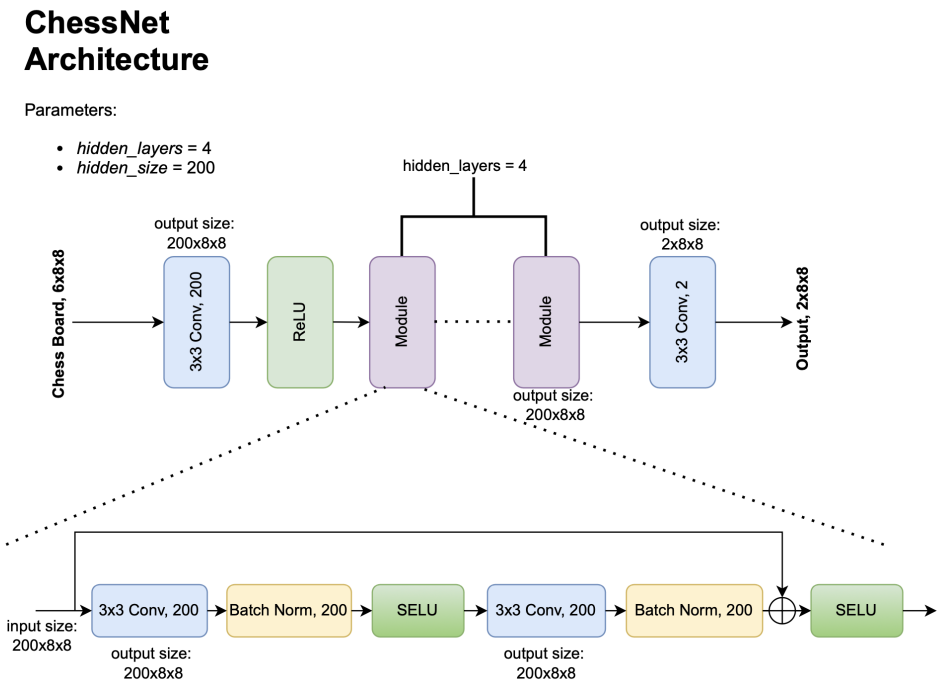


Figure 15: ChessNet Diagram with Default Parameters

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`)?

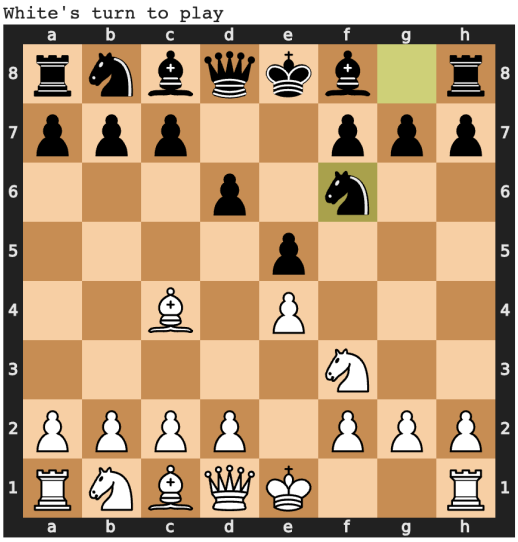
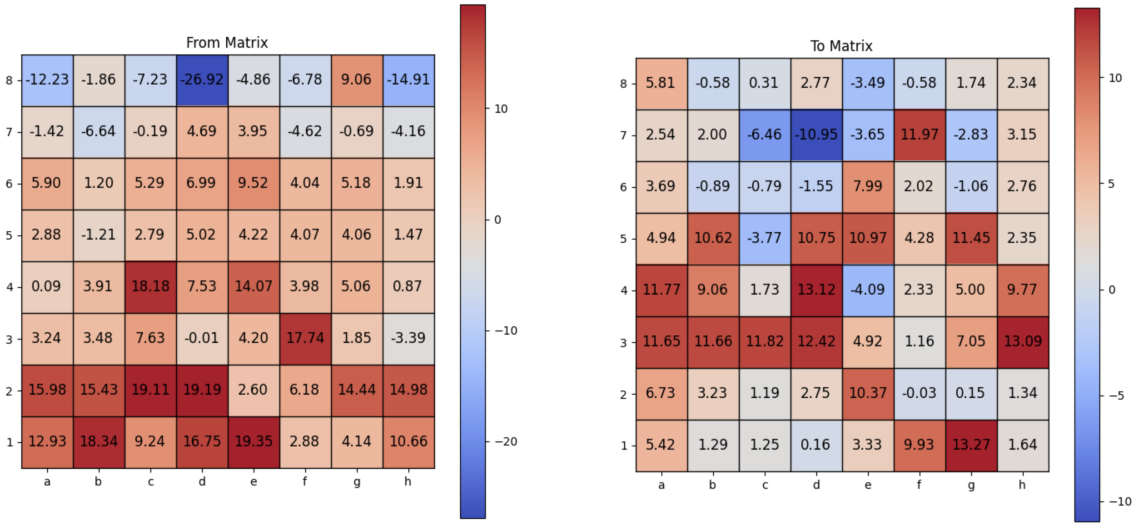


Figure 16: Example Board

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.

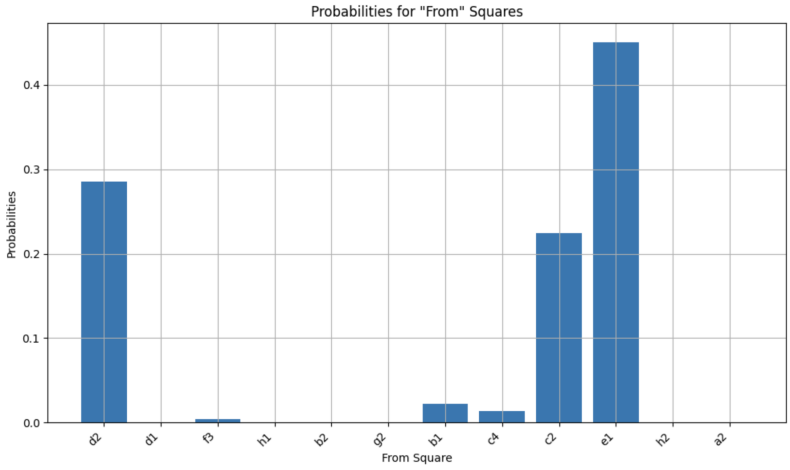


Figure 18: Calculated using `distribution_over_moves` func in `utils.py`

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)

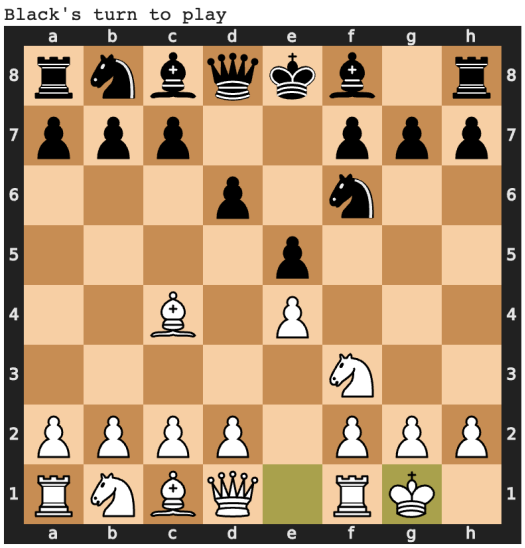


Figure 21: Board from ref. 16 after the model chose the move - *e1g1*

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.

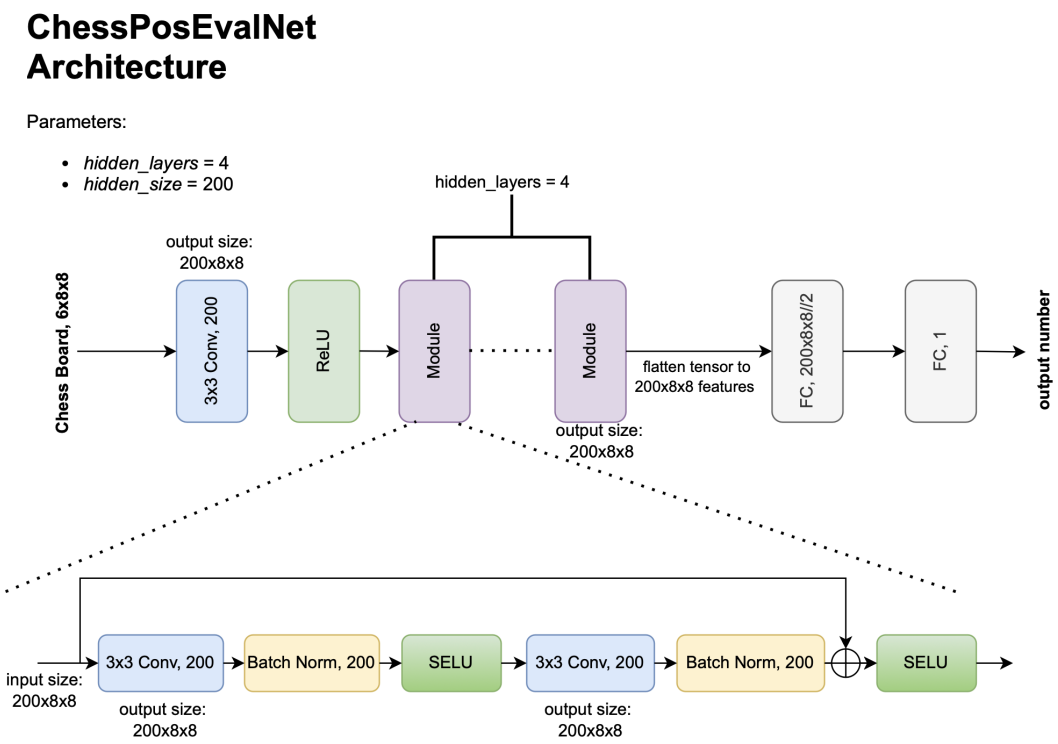
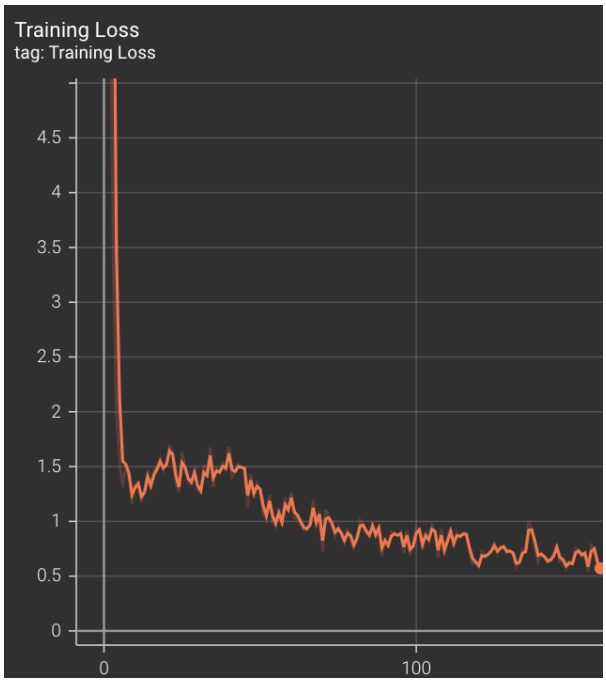


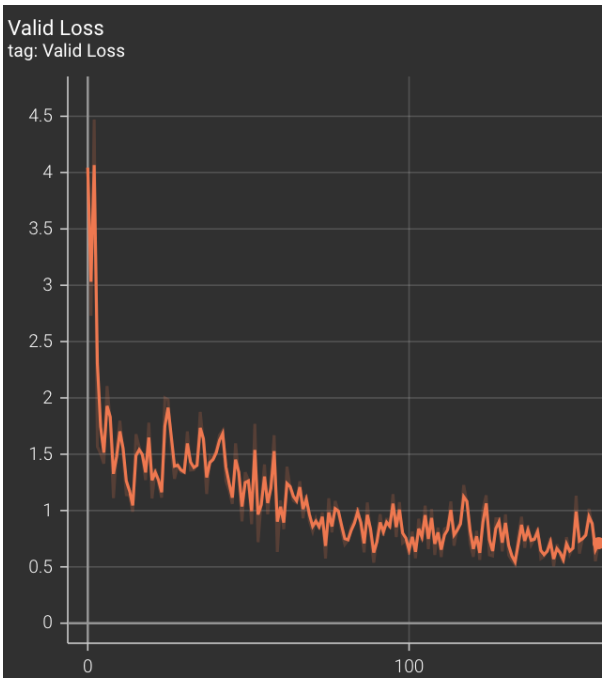
Figure 22: ChessPosEvalNet Diagram with Default Parameters

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

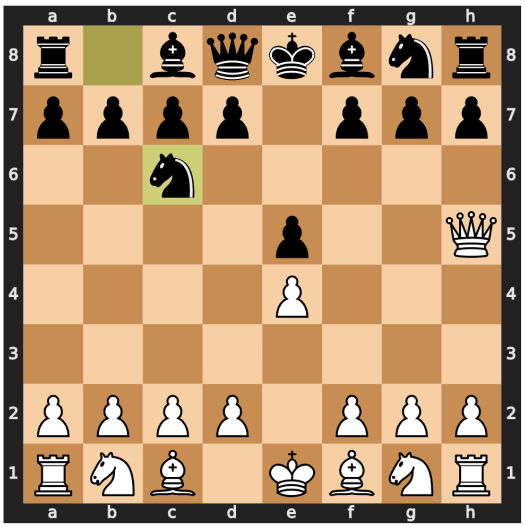
The model's MSE losses at the end of training:

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

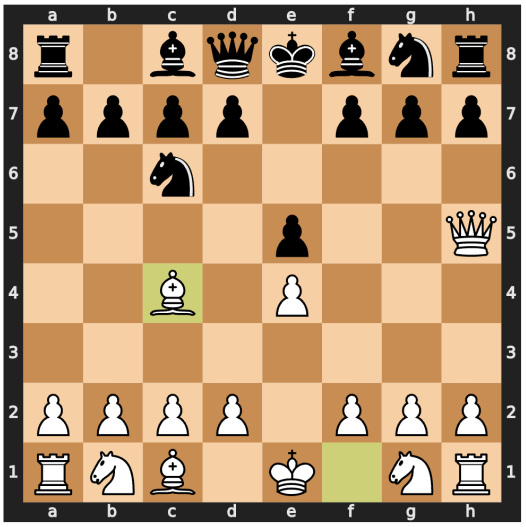
The training code of ChessPosEvalNet can be found in `train_board_eval.py`.

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.

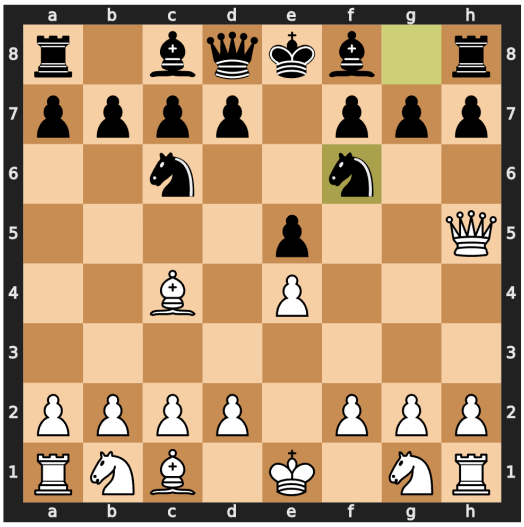
The following is a sequence of game moves. From (a) to (b), we can see that the model predicted an increase in white's strength, which makes sense because the bishop is set up in a position that can let the queen checkmate the black side. From (b) to (c), the black player did not defend against the checkmate,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



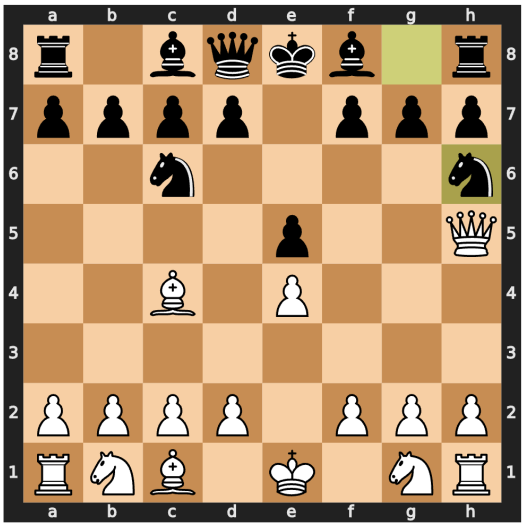
(a)



(b)



(c)



(d)

However, our model 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 stockfish 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.

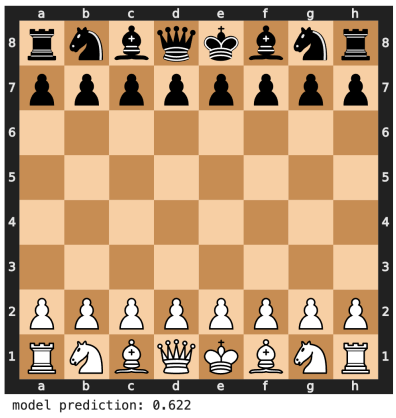


Figure 25: ChessPosEvalNet prediction for a starting chessboard

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.

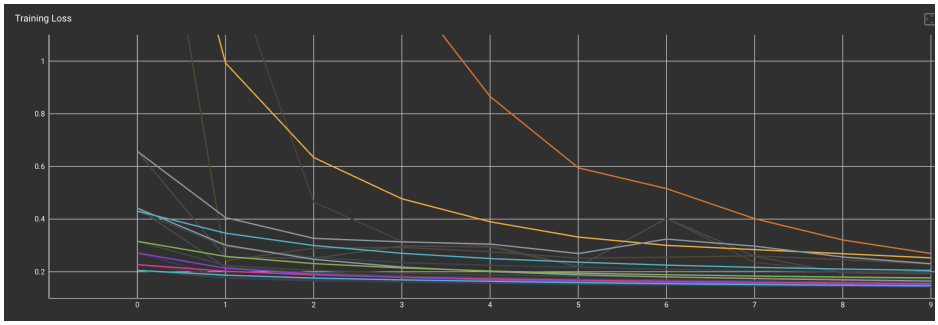


Figure 27: Training Loss vs. Epoch

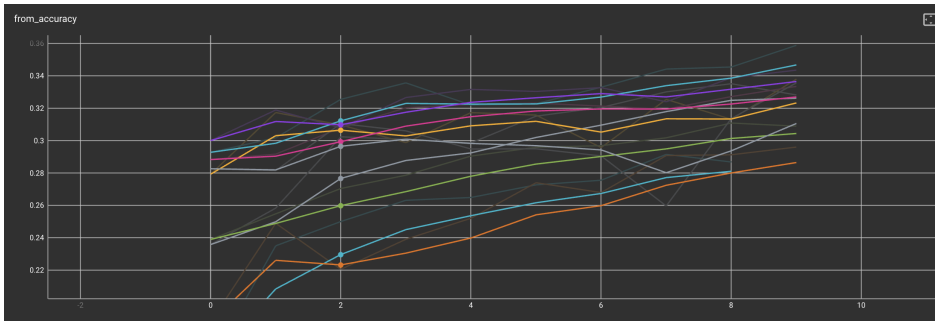


Figure 28: From Accuracy vs. Epoch

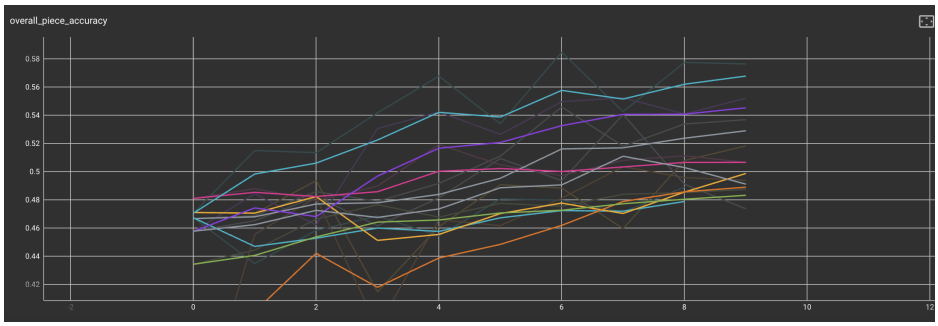


Figure 29: Overall Training Accuracy vs. Epoch

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 ChessNet's Results

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.

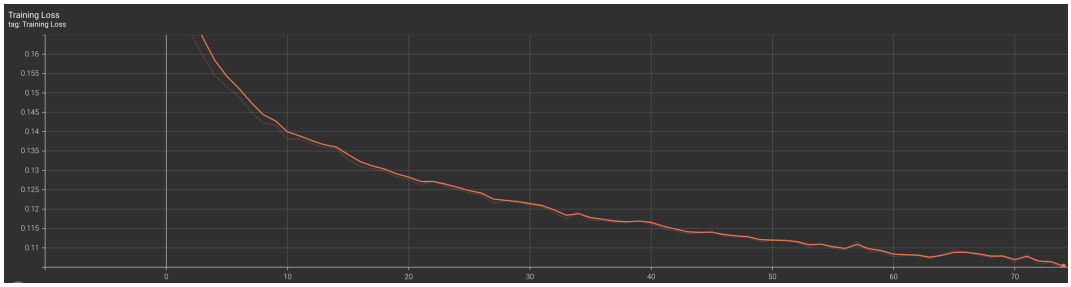


Figure 30: ChessNet Training - Loss vs. Epoch

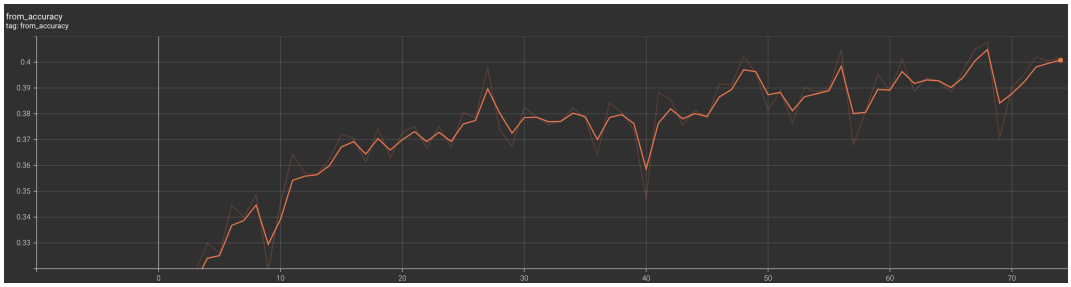


Figure 31: ChessNet Training - From Accuracy vs. Epoch

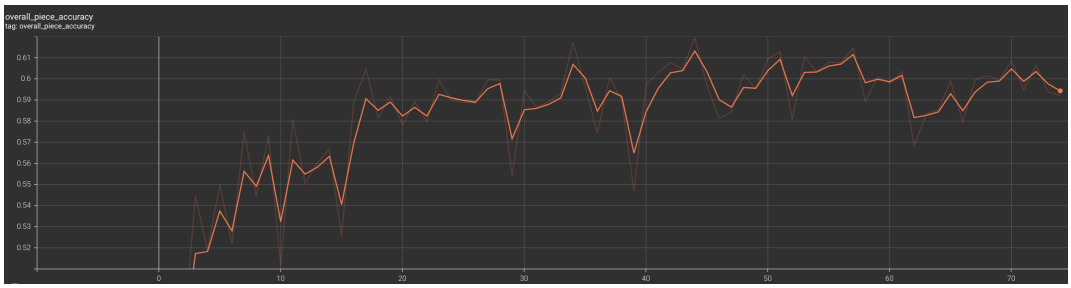


Figure 32: ChessNet Training - Overall Piece Accuracy vs. Epoch

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 the train, valid, and test sets: see figure 33. Results can be replicated using the `ChessNet_final_tests.ipynb` notebook. Note that because of the nature of our dataloaders, results cannot be 'perfectly' replicated, but running the accuracy tests over and over again yield numbers nearly identical. From the results, 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 [14].

Measurement	Accuracy
from	0.393
overall piece	0.584
pawns	0.877
rooks	0.344
knights	0.614
bishops	0.562
queens	0.301
kings	0.635

(a) Training Set

Measurement	Accuracy
from	0.392
overall piece	0.575
pawns	0.862
rooks	0.355
knights	0.600
bishops	0.524
queens	0.301
kings	0.649

(b) Validation Set

Measurement	Accuracy
from	0.387
overall piece	0.579
pawns	0.854
rooks	0.343
knights	0.604
bishops	0.550
queens	0.301
kings	0.654

(c) Test Set

Figure 33: ChessNet's Performance of the training, validation, and test set

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. Measuring move prediction quality would require simulating 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.

To better understand real world performance of ChessNet, we decided to play it against other chess bots on lichess. All games are available at ChessNet Games. We started off with a sanity test, postbotR which plays random moves. We let ChessNet and postbotR play 10 games, each bot choosing a random side every game. ChessNet won 10-0. This confirmed to us that ChessNet 'learned' to play chess, and is not predicting random moves. We moved on to letting ChessNet play against a bot with an ELO (competitive chess rating) equivalent to a beginner chess player. Because the choice of bots with an ELO that low is very limited on lichess, we went with an obscure bot that has a stockfish evaluation function but also cares about pushing the kind forward: king-forward-bot. That being said, the bot's ELO is about 1150 which is at the top end of a beginner player. The first thing we noticed about ChessNet is that its inference time is extremely quick, less than one second on average. This might signify that the ChessNet model is too simple. Although ChessNet lost almost every game against this bot, from the games we can see that ChessNet played fairly well, roughly matching the captured piece values throughout the game. We saw an obvious issue with ChessNet, it doesn't know how to play effectively during the late game. It continues to go for piece captures and ignores any mates or checkmates, unless it is one move away where it is programmed to go for that move. Another bot that we played ChessNet against was lisebot which has an ELO of about 800. Note that this bot stops playing the opponent has a significantly higher captured piece value. The last bot we let ChessNet play against was sargon-1ply, which is a re-implementation of Dan and Kathe Spracklen's 1978 Sargon chess engine. It has an ELO of an intermediate chess player and was able to very easily beat ChessNet in almost every game.

Table 1: ChessNet vs. Lichess Bots

Bot	ELO of Bot	ChessNet vs. Bot	Example Game	ChessNet vs. Bot Accuracy
postbotR	< 600	10-0	game	88% vs. 63%
lisebot	≈ 800	5-0	game	66% vs. 59%
king-forward-bot	≈ 1150	0.5 vs. 6.5	game	52% vs. 59%
sargon-1ply	≈ 1300	1.5 vs. 10.5	game	76% vs. 89%

In our analysis of ChessNet's performance against other bots, we've gained valuable insights into its capabilities and limitations. The most glaring issue is that ChessNet did not learn piece values very well. In chess, the relative value of pieces is well established, with queens considered more valuable than rooks, rooks more valuable than bishops and knights, and bishops/knights worth more than pawns. ChessNet often makes sub optimal exchanges, such as trading queens for rooks and other pieces for lower valued ones. This could be attributed to the network's depth and its inability to grasp these nuanced numerical relationships. Despite this, ChessNet demonstrates proficiency in starting games. According to the Stockfish analyzer on Lichess, ChessNet performs strongly in the opening moves, often matching up well against strong bot opponents. However, as games progress into the mid game, its performance tends to decline. The last major thing we noticed about ChessNet is its extremely fast inference time of less than

1 second per turn. While this might appear advantageous, it can place ChessNet at a significant disadvantage, particularly in long time controlled games. Opponent bots use their allotted time to analyze and make strategic moves, whereas ChessNet's rapid decisions are often not as well informed. To address these challenges, we believe that expanding the depth and width of ChessNet, which involves increasing *hidden_layers* and increasing *hidden_size*, could enhance its overall playing strength and allow for more competitive decision making across various game scenarios.

That being said, concepts such as defending or pawn chains are often times best expressed by heuristic methods and conditionals, such as "if pawn diagonally behind piece" or "if bishop on central diagonal". In other words, chess understanding is heavily characterized by domain knowledge. Thus, ChessNet's effectiveness as a chess agent can be significantly enhanced with the incorporation of an evaluation function. [14]

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 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 Manhattan 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 Manhattan 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 2: 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 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 34: Random position represented in FEN format and the move chosen to be played by the top player in that position

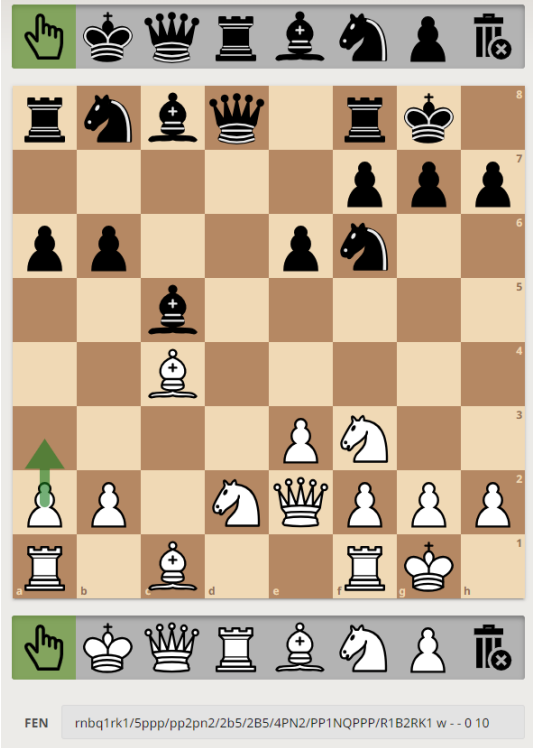


Figure 35: A visual representation of the Random position and the move chosen to be played by the top player in that position

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 75 random binary strings of size 244 bits each.
2. Each of these 244 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 75 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 organism 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.

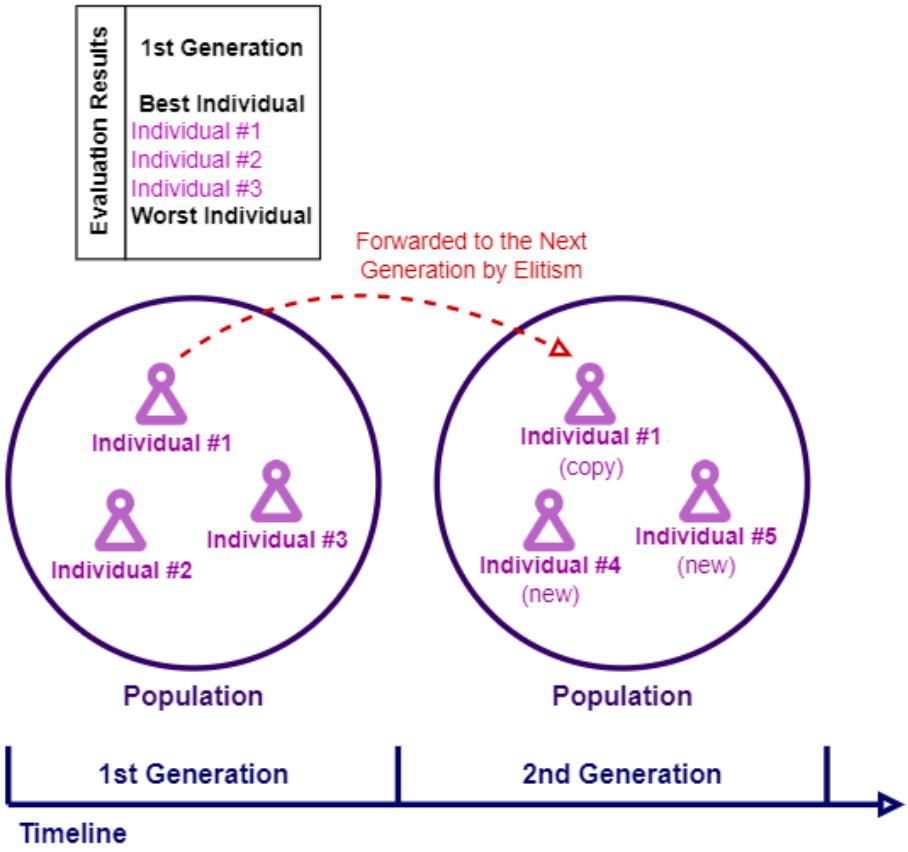
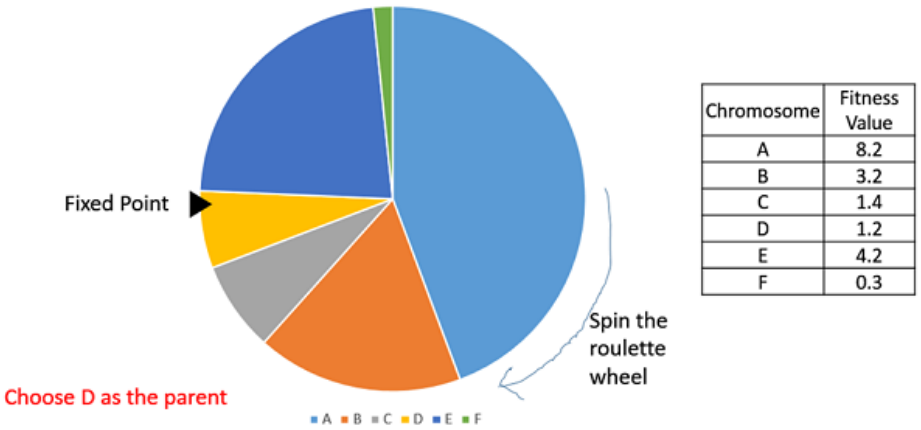


Figure 36: Simple example illustrating Elitism [6]

   - 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 will 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 $P \leq S$.
     - 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 37: 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.

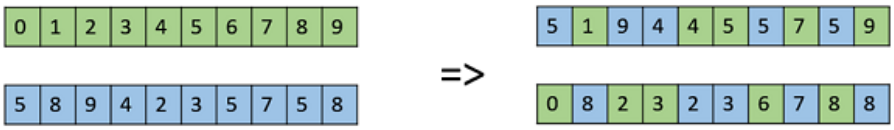


Figure 38: Simple example illustrating Uniform Crossover [7]

- 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]).

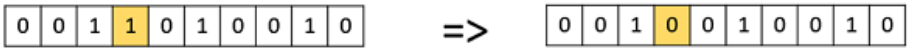


Figure 39: Simple example illustrating Bit Flip Mutation [7]

5. Thus, we perform the above mentioned process for a period of 150 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 Results after Genetic Algorithm 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:

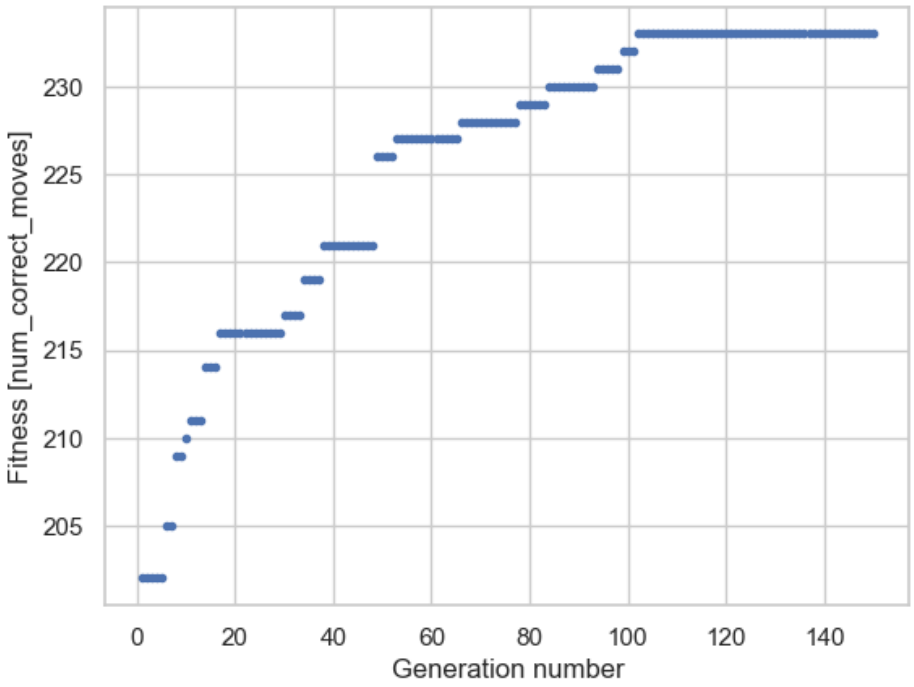


Figure 40: showing the fitness of the fittest organism in the population as a function of the generation number

From Figure 40, 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 41, 42, 43 and 44 below:

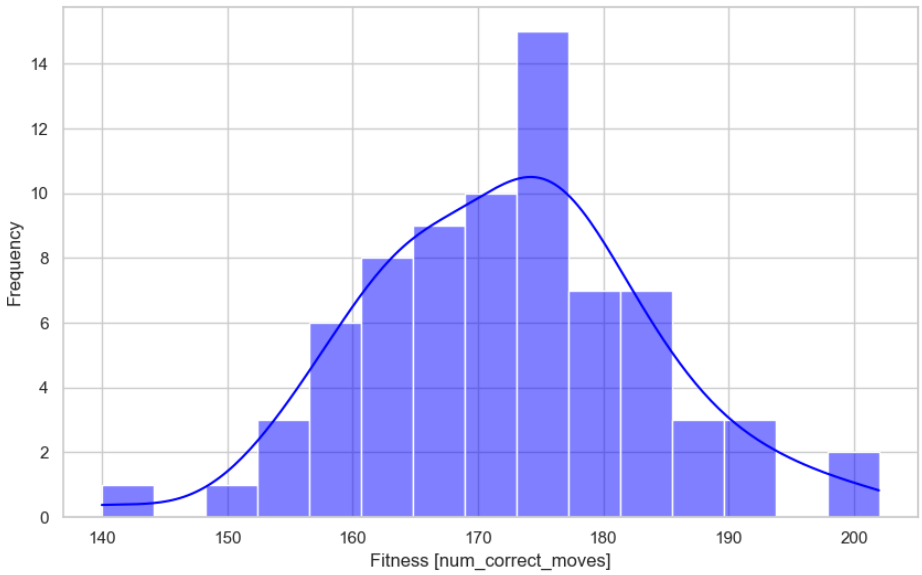


Figure 41: histogram of the population fitness of the 75 organisms after 1 generation

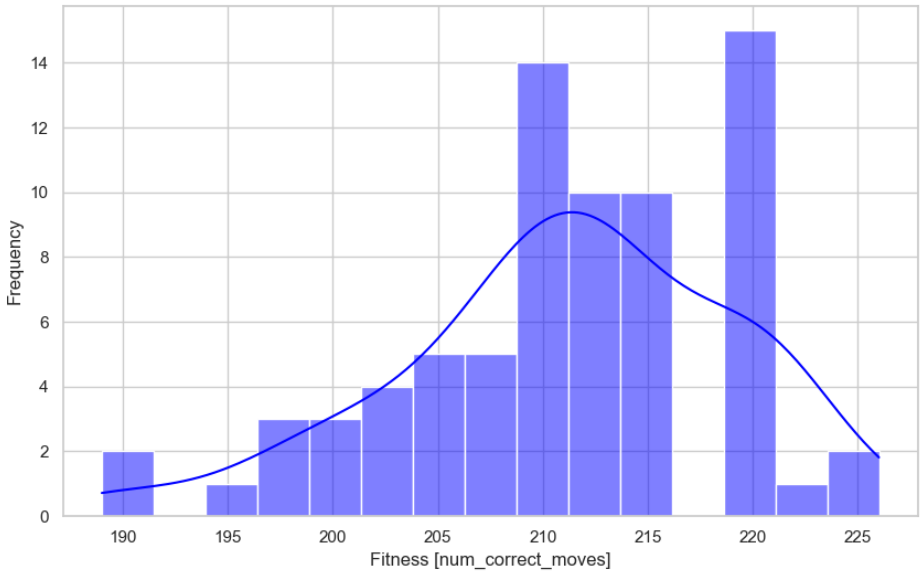


Figure 42: histogram of the population fitness of the 75 organisms after 50 generations

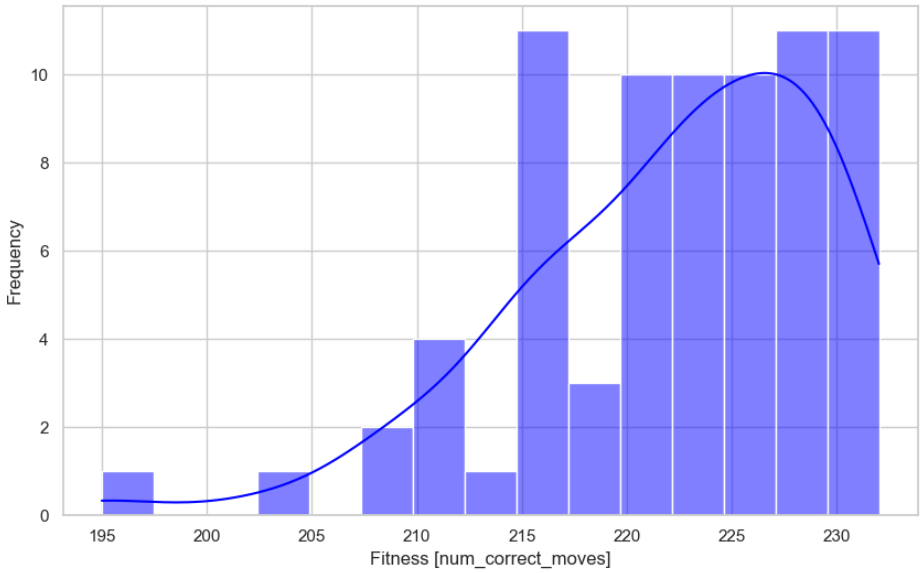


Figure 43: histogram of the population fitness of the 75 organisms after 100 generations

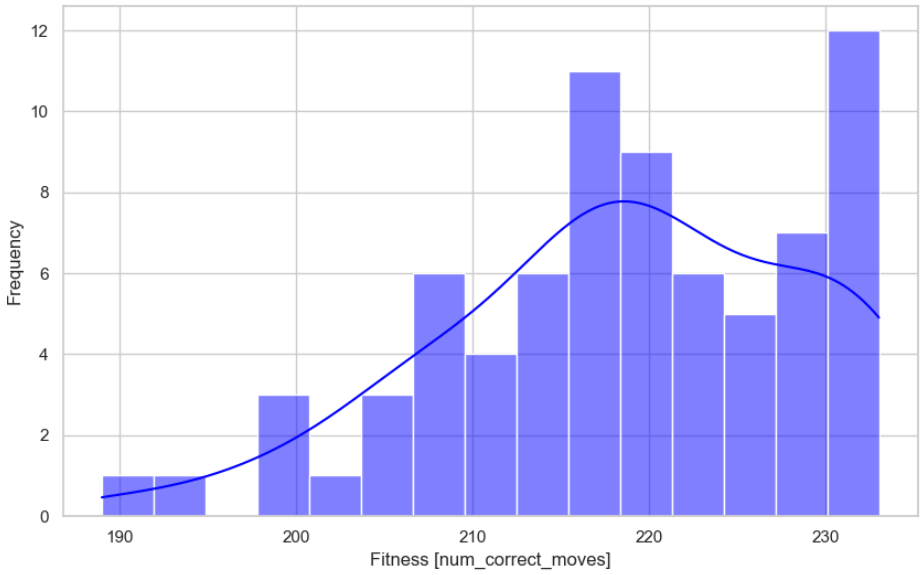


Figure 44: histogram of the population fitness of the 75 organisms after 150 generations

As such, we see that although the initial population is random (and therefore represents a normal distribution) as seen in Figure 41, 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 43. 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:

1. 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).

2. 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.

3. 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].

In order to gain a better sense of how the genetically trained organism was performing, we decided to create a chess engine bot on the internet platform Lichess that would run a Minimax program using our genetically trained heuristics. We decided to name our bot "aviTAL_BOTruCHESSky"as both an allusion to the chess world champion Mikhail Tal, and of course a direct pun on Avital Boruchovsky, Technion's resident grandmaster!

We used our bot (at a depth of 4) to play a number of classical and rapid games against the bots CHoMPBot , sargon-1ply (chosen because they are relatively weak) as well as the chess player Fang3416. The results are shown in the table below. Note that the accuracy of aviTAL_BOTruCHESSky is calculated using Lichess's accuracy metric - which makes use of Stockfish 16 (one of the strongest chess engines).

No	Name of Opponent Bot	Elo Rating of Bot	Colour of aviTAL	Accuracy of aviTAL	Result of game	Link to Game
1	CHoMPBot	1138	White	93%	1/2-1/2	https://lichess.org/kgPRtEfH4RrL
2	CHoMPBot	1138	Black	58%	1-0	https://lichess.org/DMg6qGt9Zhq7
3	sargon-1ply	1334	White	62%	0-1	https://lichess.org/AUBNwHXEbul4
4	sargon-1ply	1334	Black	87%	1/2-1/2	https://lichess.org/BN1FTxrQ5ojj
5	Fang3416	2150 (human player)	White	82%	0-1	https://lichess.org/ic7N16vM
6	Fang3416	2150 (human player)	Black	50%	1-0	https://lichess.org/LWvk2GdK/black#1

Table 4: showing the game results of the best trained organism after evolution

From these games, we were able to discern that while the genetically trained organism was able to learn concepts like the importance of development and king safety, it had a very poor understanding of material value. While this can partially be chalked up to the Minimax search being conducted at a low depth, it is clear that there are still issues regarding the relative value of the pieces. Here are a few examples:

Figure 45: Move 11 in the game Fang3416 vs aviTAL_BOTruCHESSky

In Figure 45, we see that the trained organism has castled and developed both its knight and queen - all generally good chess principles. However, it then decides to blunder the queen for a rook here with the move Qxf1+. This move was played most likely because the genetically trained organism learned the value of the queen (1968) to be approximately the same as that of a rook (2004). This is incorrect, as most conventional chess knowledge states that the value of the queen is almost twice the value of the rook!

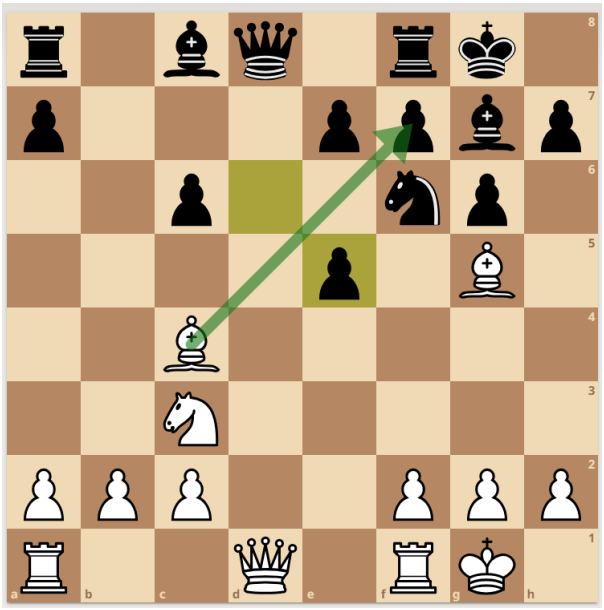


Figure 46: Move 11 in the game aviTAL_BOTruCHESSky vs Fang3416

In Figure 46, we see that the genetically trained organism has obtained a decent position. It has developed almost all its pieces and has taken care of the safety of the king by castling. However, it bizarrely decides to blunder with Bxf7+ here, giving up the bishop for a mere pawn! This pattern cropped up in a number of games, indicating that the heuristic overestimates the value of king safety, and underestimates the relative value of the pieces.

Figure 47: Move 11 in the game CHoMPBoT vs aviTAL_BOTruCHESSky

In Figure 47, our organism has a very nice position. All its pieces are developed and the opposite castling suggests the possibility of an upcoming pawn storm! Unfortunately, the organism commits the same error as that seen in Figure 46, except this time it blunders a queen and bishop for a measly rook and pawn. Once again, we see that the

organism underestimates the value of the pieces in favour of luring out the enemy king.

Figure 48: Move 7 in the game aviTAL_BOTruCHESSky vs sargon-1ply

In Figure 48, we see the organism blunder a knight with Ne2, favouring development of the pieces over material distribution. Like the previous organisms, this indicates flaws within the genetically tuned heuristics.

3.2.5 Running the Co-evolution Genetic Algorithm

The main problem with only having run the genetic algorithm once, is that like general hill climbing algorithms, only a local optimum is reached. Additionally, the set of random positions chosen introduces a sort of bias. For example, if the set of random positions contains more moves made by a rook than moves made by a queen, then it is possible that the value of the rook will evolve to be more valuable than the queen. In order to combat this problem, we did the following:

1. We ran the genetic algorithm 4 more times (which took about 4 days on the Lambda server for each iteration) in order to produce 5 distinct genetically trained organisms.

2. We then had every organism perform genetic crossover with every other organism. This produced an additional 10 new organisms.

3. These 10 new organisms along with the original 5 organisms that were each the product of the genetic algorithm then formed the initial population of the co-evolution.

4. In order to perform the co-evolution, we had every organism play on a depth of 3 against every other organism 4 times (twice with white and twice with black). In order to make sure that the results of the two white games and the two black games weren't duplicated, we made sure to use distinct chess positions. Figures 49 and 50 below illustrate the two initial starting positions chosen.

Figure 49: Starting position for co-evolution training: Initial setup

Figure 50: Starting position for co-evolution training: Sicillian Defense

If the organism won a game, they would win a point. If they lost a game, they would lose a point. A draw does not cause a change in point count.

It is important to note that many of the chess games would have gone on for a very long amount of time. Therefore, we decided to cap the total number of moves per game to 15 per side. If the game was not finished within 15 moves, the winner would be chosen as the person with a greater material count than their opponent. The other reason for doing this was in order to make the organisms care more about material - which was a common problem in the original genetically tuned organisms. In the same way as before, the winner gets a point added to their point count and the loser gets a point subtracted. Positions of equal material are taken to be a "draw" and no points are added or subtracted.

The point count that each organism maintained was taken to be its fitness value. From this, we were able to do the procedures of Elitism, Fitness Proportional Selection, and Uniform Crossover in a similar manner to that used previously.

When running the algorithm, we decided to use the following hyperparameters:

(a) crossover rate = 0.75

(b) mutation rate = 0

(c) elitism rate = 0.2

Note: The reason that the mutation rate is set to 0 and the Elitism rate was increased is because of the small population size.

Additionally, note that this is the reason why the procedure is known as co-evolution. Each of the organisms affects the evolution of each of the other organisms through the determination of its fitness value. This is in contrast to the previous form of evolution, where the fitness of each organism was independently correlated to the fitness of every other organism.

5. We thus performed the co-evolution procedure for 4 generations (which took a period of 4 days on the lambda server) and then took the organism in the final generation that had the highest fitness. This was our final "optimal genetically trained organism".

3.2.6 Results after Co-evolution

The following table is used to illustrate the parameter values of the best trained organism after the co-evolution process:

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	1458
5	Queen Value	2041
6	Passed Pawn Number	20
7	Doubled Pawn Number	53
8	Isolated Pawn Number	52
9	Backward Pawn Number	17
10	Passed Pawn Enemy King Distance	27
11	Number of Center Pawns	56
12	Knight Mobility	61
13	Knight Outpost Number	5
14	Bishop Mobility	3
15	Bishop Pair	57
16	Rook Attack Weak Pawn Open Column	9
17	Rook Connected	13
18	Rook Mobility	24
19	Rook Open File	43
20	Rook Semi-Open File	59
21	Rook Attack King Adj File	50
22	Rook Attack King File	29
23	Rook Behind Passed Pawn	40
24	Queen Mobility	2
25	King Number Friendly Pawns	8
26	King Number Friendly Pawns Adjacent	23
27	King Friendly Pawn Advanced	55
28	King Number Enemy Pawns	3
29	King Number Enemy Pawns Adjacent	23
30	Pressure against King	45
31	Checkmate	15708
32	Bishop Piece Table	23
33	Knight Piece Table	13
34	Rook Piece Table	2
35	Queen Piece Table	36
36	Pawn Piece Table	41
37	King Piece Table	4

Table 5: showing the parameter values of the best trained organism after co-evolution

We then updated our bot aviTAL_BOTruCHESSky to use the best trained organism after the co-evolution process. It then played a number of classical games (at a depth of 4) against the bots CHoMPBot and sargon-1ply. The results are as follows:

No	Name of Opponent Bot	Elo Rating of Bot	Colour of aviTAL	Accuracy of aviTAL	Result of game	Link to Game
1	CHoMPBot	1138	White	80%	1-0	https://lichess.org/obODY5gFUcCI
2	CHoMPBot	1138	Black	78%	0-1	https://lichess.org/0l2s2xkR825l
3	sargon-1ply	1334	White	42%	1/2-1/2	https://lichess.org/I6RG2gGdgQ74
4	sargon-1ply	1334	Black	71%	1/2-1/2	https://lichess.org/G8U9S8By/black#0

Table 6: showing the game results of the best trained organism after co-evolution

By comparing the results of Table 6 with the results of Table 4, we see that the co-evolution process indeed made the organism stronger as it is now able to convincingly beat the CHoMPBot, while it easily lost to it previously!

4 Experimental Methodology

We now wish to determine the optimal trade-off between the genetically tuned heuristics (after the co-evolution process) and the Convolutional Neural Network. In order to do this, we make use of the hyperparameter λ, in the same manner as that discussed in Section 2: "An Introduction to our Chess Engine". As such, we will define 11 different values of λ: 0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9 and 1.

Note that although λ is some value between 0 and 1, we will actually be scaling it by the factor 3000. In other words, when $\lambda = 1$, a score of 3000 will be added to the evaluation function.

The reason we chose 3000 is that every heuristic (except for the checkmate heuristic) can only influence a certain move to be chosen by adding or subtracting a score with a value less than 3000. For example, if a given move involves capturing the opponent's queen, then we gain the score of the value of the queen, as well as its mobility. The maximum mobility (or squares controlled) of a queen on an open board is 27. If we multiply this by the Queen Mobility parameter (2) and add the value of the queen, we only get 2095, which is less than 3000. In other words, if the value of λ is 1, then the weight of the CNN will be at least the value of a queen according to the evaluation function!

Additionally, we ignore the weight of the checkmate parameter as we should not have the move chosen by the CNN influence a decision if there is a move that causes checkmate on the board. The move that causes checkmate should be chosen every time.

Finally, it is important to note that we will be determining the optimal λ bot specifically for a depth of 1. The reason we chose to run at this depth is two fold. We partially chose it out of convenience, as the lack of search enables our code to run faster and so enables a greater quantity of experiments to be performed. Additionally, we wish to see how well our engine performs when it doesn't perform any search and only relies on heuristics (either those tuned by the genetic algorithm, deduced by the CNN or a hybrid of the two). As we can see from the chess bot sargon-1ply, these engines can perform relatively well even without search capabilities.

In order to determine which λ value is optimal (as well as the level of performance it achieves), we will use the following three methods:

1. We will make all of the λ bots play against one another and see which one performs the best.
2. We will test each of the λ bots on an external dataset and see which one performs the best.
3. We will take the best performing λ bot and play it against other Lichess bots to see how well it does.

Each of these methods has their own merits and shortcomings. For example, using an external dataset is good at picking up how often a bot chooses the correct move, but not how many times it blunders - as well as how bad the blunders are. In other words, it is not helpful to have a bot that plays a decent game of chess only to blunder a queen for a pawn after a certain number of moves. Conversely, a λ bot that performs well against the other λ bots may suggest that it does not blunder as much. However, it may not be as good at finding the best move in a given position. As such, we decided that using multiple metrics is useful in determining the optimal λ value.

As such, we will discuss the results of each of these different metrics in the following sections.

5 Experimental Results and Analysis

5.1 Making the λ bots compete against one another

We made each of the Lambda bots play one match against every other bot. Each match consisted of one λ bot playing ten games as white and then ten games as black (on a depth of 1) against another λ bot (also on a depth of 1). A win is worth one point, a draw half a point and a loss zero points. In order to make sure that the games weren't all the same, we used the following ten different starting positions (that are derived from typical chess openings).

(a) Starting Position

(b) Kings Pawn Opening

(c) French Defense

(d) Sicillian

(e) Caro-Kann

(f) Queen's Gambit

(g) Nimzo Indian Defense

(h) Nimzo-Larsen Attack

(i) Kings Indian Defense

(j) English Opening

The results from all the games are shown in the two tables below. Note that in Table 7 , each cell represents the number of points scored by the bot in the given row of the cell when it played as white against the bot found in the specific column of the cell. For example, the cell in the second row and third column has the number 7 in it. This means that the $\lambda = 0$ bot played against the $\lambda = 0.1$ bot and won 7 of its 10 games. Similarly, in Table 8, the row represents the bot that played as black and the number of points it scored against the column (which represents the bot that played as white).

λ value	$\lambda = 0$	$\lambda = 0.1$	$\lambda = 0.2$	$\lambda = 0.3$	$\lambda = 0.4$	$\lambda = 0.5$	$\lambda = 0.6$	$\lambda = 0.7$	$\lambda = 0.8$	$\lambda = 0.9$	$\lambda = 1$	Total Point Count
$\lambda = 0$	N/A	7	6.5	6.5	4.5	5.5	3	5.5	3.5	5.5	3.5	51
$\lambda = 0.1$	5	N/A	6.5	4	3.5	6	5.5	4	3	4.5	8	50
$\lambda = 0.2$	6.5	6	N/A	6	7	6	6	6	8.5	3.5	3	58.5
$\lambda = 0.3$	8	5.5	5.5	N/A	6	6	4.5	6	6	3.5	7.5	58.5
$\lambda = 0.4$	9	6.5	5.5	5.5	N/A	2.5	5	5	5	6.5	6.5	57
$\lambda = 0.5$	5.5	6.5	8	6	6	N/A	7	3.5	6	7.5	6	62
$\lambda = 0.6$	9	5	7.5	7.5	5.5	6	N/A	5.5	6.5	7.5	4	64
$\lambda = 0.7$	6.5	6	7	7	4.5	4	7.5	N/A	6	7.5	4.5	60.5
$\lambda = 0.8$	7	7	8	6	6	4.5	8	4.5	N/A	6	6.5	63.5
$\lambda = 0.9$	6.5	7	7	4.5	5	6.5	6	5.5	5.5	N/A	6	59.5
$\lambda = 1$	7	6.5	7.5	5.5	7	5	7.5	5	8.5	4.5	N/A	64

Table 7: showing the game results between each of the λ bots after 10 games at a depth of 1 (row is bot playing white, column is bot playing black, cell is number of points scored by white).

λ value	$\lambda = 0$	$\lambda = 0.1$	$\lambda = 0.2$	$\lambda = 0.3$	$\lambda = 0.4$	$\lambda = 0.5$	$\lambda = 0.6$	$\lambda = 0.7$	$\lambda = 0.8$	$\lambda = 0.9$	$\lambda = 1$	Total Point Count
$\lambda = 0$	N/A	5	3.5	2	1	4.5	1	3.5	3	3.5	3	30
$\lambda = 0.1$	3	N/A	4	4.5	3.5	3.5	5	4	3	3	3.5	37
$\lambda = 0.2$	3.5	3.5	N/A	4.5	4.5	2	2.5	3	2	3	2.5	31
$\lambda = 0.3$	3.5	6	4	N/A	4.5	4	2.5	3	2	3	2.5	31.5
$\lambda = 0.4$	5.5	6.5	3	4	N/A	4	4.5	5.5	4	5	3	45
$\lambda = 0.5$	4.5	4	4	4	7.5	N/A	4	6	5.5	3.5	5	48
$\lambda = 0.6$	7	4.5	4	5.5	5	3	N/A	2.5	2	4	2.5	40
$\lambda = 0.7$	4.5	6	4	4	5	6.5	4.5	N/A	5.5	4.5	5	49.5
$\lambda = 0.8$	6.5	7	1.5	4	5	4	3.5	4	N/A	4.5	1.5	41.5
$\lambda = 0.9$	4.5	5.5	6.5	6.5	3.5	2.5	2.5	2.5	4	N/A	5.5	43.5
$\lambda = 1$	6.5	2	7	2.5	3.5	4	6	5.5	3.5	4	N/A	44.5

Table 8: showing the game results between each of the λ bots after 10 games at a depth of 1 (row is bot playing black, column is bot playing white, cell is number of points scored by black).

From the results of these two tables, we are able to calculate the ranking of each of the λ bots according to how they did against each other.

Ranking	Value of λ	Total Point Count (out of 10)
1/2	0.7	110
1/2	0.5	110
3	1	108.5
4	0.8	105
5	0.6	104
6	0.9	103
7	0.4	102
8	0.2	95.5
9	0.3	89.5
10	0.1	87
11	0	81

Table 9: showing the final point score of each of the λ bots after playing one another.

As we can see, the best performing λ bots were those with λ values of 0.7 and 0.5. Additionally, the CNN performed much better than the genetically tuned heuristic, coming in with a Total Point Count of 108.5, compared to the low Total Point Count of only 81. Of course, this is also partially because we are only testing at a depth of 1. We would expect the genetically tuned heuristic to do better at higher depths, while the CNN would not benefit as much.

5.2 Testing the λ bots on an external dataset

In order to gain an additional perspective on what the best λ trade-off would be, we decided to run each of the bots at different depths on a dataset that was independent to those used for the training of both the genetically tuned heuristics and the convolutional neural network. In our case, we decided to use the publicly available Lichess Elite database, which features all the games played on the website Lichess.com from January of 2022 where both players were above a 2000 Lichess rating threshold.We then filtered the dataset: selecting only for games where the Lichess rating of both players was greater than or equal to 2500.

From this set of games, we then selected 5000 random White positions (where White won the game) and 5000 random Black positions (where Black won the game). Along with each random position, we also stored the move that the top player played in that position.

We then tested each of the λ models at a depth of 1 on this new,external dataset in order to determine the optimal λ trade-off. The results are shown in Table 10 :

Ranking	Value of λ	Number of correctly guessed moves (out of 10000)
1	0.6	2341
2	0.5	2320
3	0.7	2298
4	0.8	2282
5	0.4	2246
6	0.9	2215
7	1	2195
8	0.3	2163
9	0.2	2076
10	0.1	1996
11	0	1801

Table 10: showing the number of correctly guessed moves of each of the λ bots at a depth of 1.

From these results, we clearly see a trend of the λ bots obtaining a higher number of correct moves as λ increases from 0 to 0.6 and as λ decreases from 1 to 0.6. As such, we expect the best value of λ to be some value around approximately, 0.5-0.7 (given that 0.5, 0.6 and 0.7 are all the top performing λ values).

Indeed, from the previous section, we found that the best performing λ bots were $\lambda = 0.5$ and $\lambda = 0.7$. As such, overall we can deduce that $\lambda = 0.5$ is the optimal weighted trade-off between the CNN and the genetically tuned heuristics.

5.3 Observing the performance on Lichess of the best λ bot

Following the test results which indicated the optimal λ value to be 0.5, we updated the chess bot aviTAL_BOTruCHESSky to perform this weighted trade-off between the genetically tuned heuristics and the CNN.

We then made the bot play games at a depth of 3 (1 white and 1 black) against multiple Lichess bots. Each of the following games were played at a time control of 20 minutes per side and 30 second increment:

No	Name of Bot	Bot Elo Rating	Colour of aviTAL	Accuracy of aviTAL	Result of game	Link to Game
1	SxRandom	830	White	95%	1-0	https://lichess.org/H7hNIqJ7
2	SxRandom	830	Black	84%	0-1	https://lichess.org/ntcK0cj7
3	CHoMPBot	1138	White	88%	1-0	https://lichess.org/PKY7xSM98w5M
4	CHoMPBot	1138	Black	51 %	0-1	https://lichess.org/A4NSxGcL
5	sargon-1ply	1318	White	74%	1-0	https://lichess.org/8j2JnpTM
6	sargon-1ply	1318	Black	85%	1-0	https://lichess.org/WU76JOSg
7	sargon-2ply	1467	White	60%	0-1	https://lichess.org/pixaTsfe
8	sargon-2ply	1467	Black	45%	1-0	https://lichess.org/rG4c6jo6

Table 11: showing the game results of each of the best bot $\lambda = 0.5$ against multiple Lichess bots at a depth of 3.

From the results of the above table, we see that the bot is capable of playing at approximately a 1300 Elo level! This is seen from its convincing victories with both colours against SxRandom and CHoMPBot and its one victory against sargon-1ply with the white pieces.

Given that both the genetically tuned heuristics and the CNN could not defeat bots of the level of sargon-1ply when performing on their own, leads credence to the idea that doing a trade-off between neural and non-neuronal heuristics (such as ones seen in modern iterations of Stockfish) is a useful technique to improve the overall effectiveness of a chess engine.

6 Conclusion

In this paper, we were interested in investigating the effectiveness of different AI techniques in building chess engines. Primarily, we were interested in:

1. determining the effectiveness of deep learning techniques such as Convolutional Neural Networks
2. determining the effectiveness of handcrafted heuristics that are trained using genetic algorithms
3. The effect of combining these two methods, and the weighted trade-off between them.

From our research on Convolutional Neural networks, we found them to be very good at the beginning of chess games (as they memorise opening theory very well). However, they become less and less reliable the longer the game goes on. This is mainly due to the sheer volume of possible chess positions, and the inability of training data to cope with all possibilities. Additionally, we found that CNNs are very good at understanding good piece placement for short-range pieces such as knights and pawns, but they struggle for more long range pieces such as rooks and queens. This makes sense due to the small convolutional window that is generated through use of a CNN. Finally, we found that while the CNN was able to beat simple bots (such as SxRandom), it would often blunder away pieces meaning that on its own it was not good enough against stronger bots (such as CHoMPBot).

From our research on handcrafted algorithms and genetic heuristics, we found that while the evolutionary process used was helpful in establishing a basis of what the value of each of the parameters should be, the bias of the initial training sets of random positions meant that there were certain inconsistencies in the parameter values that arose. For example, the first organism that we trained believed that a rook was worth more than a queen because there

were more positions where the rook was the piece that moved than there were positions where it was the queen that had to move. This inconsistency in the value of material was solved through the process of co-evolution (a method that was inspired by the work previously conducted by Moshe Koppel, Nathan Netanyahu, Eli David and Jaap van Hendrik)[8]. By selecting for organisms that valued material more, we managed to achieve a final organism that had more reasonable parameter values and blundered less.

However, it would be unreasonable to discuss the genetically tuned heuristics without bringing up one of its major flaws: training and inference time. Due to the large quantity of handcrafted heuristics used, we found that training time often took multiple days for each organism. Additionally, this severely slowed down inference time, and prevented us from using much larger depths on the Minimax function, something that would have greatly improved the capability of our chess engine. We recommend that for further research to be carried out, it is probably a better idea to use less heuristics when constructing the evaluation function in order to save time for increased search capabilities through Minimax. Alternatively, it could also be useful to conduct more forms of pruning in the Minimax, other than the basic Alpha-Beta pruning that we conducted.

Finally, when looking at the optimal tradeoff between the CNN and the genetically tuned heuristics, we found that the best λ value was 0.5, dead in the centre between the genetic heuristic and the CNN! This λ value was based on our experiments conducted on an external dataset, as well as the sets of games we had between bots using different λ values. Indeed, we find the fact that the final model was able to beat bots such as sargon-1ply and CHoMPBot with both colours when both original models created were not able to achieve such a feat indicative of the potential that hybrid chess engine models have.

We hope that our work can inspire future chess engine builders, from mere hobbyists to those professionally dedicated to the field.

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%20main%20purpose%20of%20using,local%20or%20global%20optima%20result).

[7] Explanations of genetic algorithms. https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_parent_selection.htm.

[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.pdf.

[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.