Neural Network Chessbot

João Gaspar (114514), Gabriel Santos (113682), Panagiotis Leikos (128625)

***Abstract*—This project aims to build a neural network-powered chess bot capable of playing strong and human-like chess. We explore various deep learning architectures, develop different algorithms and techniques for data processing, integrate board rules and do several tests. The model combines pattern recognition from gameplay datasets with modern techniques such as legal move masking, probability sampling. The engine will be designed to evaluate board positions and suggest high-quality moves with human-style variation.**

***Index Terms*—Neural Network, Chess, Bot, Convolutional Neural Networks, Deep Learning, AlphaZero, Leela Chess, Bitboards**

I. State of the Art Review

Several approaches to creating neural network-based chess engines have been developed over recent years. Notably, projects like AlphaZero by DeepMind, Leela Chess Zero (LCZero), Maia Chess and Stockfish have demonstrated the effectiveness of deep learning for chess move prediction.

Stockfish is a Tree-search engine that is capable of evaluating millions of positions per second, through a NNUE (Efficiently Updatable Neural Network), using deep alpha-beta search, being considered the number one chess engine, capable of beating any human.

AlphaZero [1] uses reinforcement learning with a deep neural network and Monte Carlo Tree Search (MCTS), achieving superhuman performance. Leela Chess Zero [2] follows AlphaZero's principles but uses distributed training with community-contributed games. Maia Chess [3] focuses on human-like move prediction by training on specific Elo ranges.

These projects typically use convolutional neural networks (CNNs), bitboard representations, and policy-value heads. Inspired by these, we adopted a simplified CNN-based architecture with legal move masking, avoiding tree search to focus purely on prediction quality.

Additionally, we reviewed smaller-scale academic efforts, such as [4], where simpler models were applied using datasets with parsed PGN files. These efforts influenced our model design choices and pre-processing strategies.

II. Introduction

ARTIFICIAL Intelligence and machine learning chess models are becoming something really common to be explored, and in many different ways like explained earlier. There are several teams and companies already aiming to build the best chess bot ever, impossible to beat, even if 1000 grandmasters were in a room collectively debating the moves against the most advanced bot today, with no time restrictions, they would probably still lose, that's how far we've come. Of course there are also some small projects, made by university students or just small groups, aiming to create in creative ways, and try to innovate chess bot logic with their limited resources. Although we're not necessarily innovating, since this isn't something new, and all of our ideas have already been explored before, we're gonna do it our way, with our resources, with our brand and logic behind the model, mixing the knowledge we own, to the best of our abilities in this limited time we have to build this bot, and let's not forget the most important thing, even more than a perfect bot, results, statistics, learning, reflecting and making logical and justifiable decisions for our construction of this project.

III. Project Vision and Methodology

As explained earlier we're aiming to build a chess bot capable of playing human like, logical, and good moves. Just to change the approach and compare a little bit, what we're showing next, is our notebook of when we first started this project, the things we envisioned, the possibilities, a little bit of our ignorant and tireless selves hoping for a perfect scenario.

What style and approach can we take:
Predict the best move; Evaluate board positions; Plays like a human (imitate a playing style); Play to win (as strong as possible); Have in consideration book moves (Opening, rules, exceptions)
FEN strings PGN strings Tensor Bitboard / Tensor representation (Choose one of these) (Transform into a bit map)
Options for model architecture:
CNN (AlphaZero); Transformer-based models (Leela Chess); Reinforcement learning (Deeper things)
Frameworks:
TensorFlow; PyTorch
TO TRAIN:
Calculate valid positions; Mask illegal moves as negatives; Augment data by flipping boards
After training: Predict move probabilities; Mask illegal moves; Highest probability move / Sample from probabilities (Human like move)
Python-chess for integration
Post execution:
Reinforcement learning (Self play or MCTS); Model comprehension; Evaluate ELO
Tools: python-chess pgnparser or chess.pgn PyTorch or TensorFlow numpy, pandas
Prioritize higher time controls and higher elos
Neural network with MinMaxing and Alpha-beta pruning
If we want we can ignore the dataset
Another choice, would be for us to calculate moves, and then, let stockfish evaluate them But honestly that would be cheating

In terms of methodology almost nothing changed, we already had a pretty good idea of what we wanted and were expecting to be able to do, maybe due to previous knowledge and curiosity over chess engines, or just liking chess in general. As for the ideas presented earlier, everything written there is still a valid and logical consideration to take, and we don't regret the approach we took, since we already explained the most common **MinMaxing with AlphaBeta Pruning**

with Search trees and heuristics, was something that didn't excited us as much, even more, after having another AI project regarding those methodologies. In terms of project vision, maybe our expectations were a bit too high, but we certainly weren't disappointed, especially after the results we're going to present, and we really feel that it's unfortunate that we didn't have just 2 more weeks in order to perfectly make our model. Our workflow wasn't so organized but it was well structured, thus we knew what we had to do, kinda. We started by researching about models, datasets, etc, then we started taking care of the pre-processing and optimizing training times, after that we trained, corrected mistakes, optimized even more, and then refactored almost everything and changed architecture several times, but that's also part of the joy in this work, going from nothing working, to something really "cool". Also in terms of results that we're going to present, some of them were cut down and simplified, others were lost in panicking late nights, clueless about what we were doing wrong, others weren't considered relevant, and overall the kind of results and progress we have and can present is very redundant. Any extra information can be provided on our project's GitHub: https://github.com/GCapaross/ChessBot

Additionally, there are a few important considerations and reflections regarding the behavior and evaluation of our CNN-based chessbot model:

On the Use of F1 Score: While F1 score is a widely used metric in classification tasks, being the harmonic mean between precision and recall, it is not well-suited to our context [5], [6]. Our model performs multi-class classification across a very large output space (thousands of potential moves). In such cases, especially when only the top prediction is used, the F1 score does not capture model performance meaningfully. Instead, evaluation metrics like top-k accuracy (e.g., whether the correct move is in the top 5 predictions) or comparing predicted moves with a chess engine evaluation is more appropriate.

On Input Normalization: Input normalization is a common pre-processing step in many machine learning models; however, in our case, it would have been detrimental. Our board representation consists of binary bitboards, each plane indicating the presence or absence of a piece type in a square. Normalizing these binary inputs would disrupt this structure, leading to ambiguity and reducing the clarity of the model's input representation [7] - This study reinforces the idea that input normalization isn't directly adressed in their chessbot.

On False Positives: In traditional binary classification tasks, false positives are instances where the model predicts a positive class incorrectly. However, in our setting, where the model outputs a probability distribution over thousands of possible chess moves, this concept doesn't directly apply. There's no binary decision, and thus no strict "false positive" count. Instead, evaluation is better framed in terms of how high the correct move ranks in the predicted list (e.g., Top-1, Top-5 accuracy), or whether the model frequently suggests strong or weak moves compared to a chess engine baseline.

Repetitive Move Patterns: During experimentation, we observed that the model sometimes got stuck performing the same sequence of moves. This is a consequence of the model lacking a memory or planning mechanism; it operates solely based on the current board state without awareness of historical context or long-term strategy. This behavior reinforces the potential value of integrating reinforcement learning or tree-based search methods for planning.

Low Rate of Invalid Moves: One notable positive outcome is the low number of invalid moves generated by the model. The model can still output illegal moves, but they are filtered out, and we try to find the next most probable legal move to be played in that position. As a result, the model is generally robust in maintaining move legality, *it's important to notice that the way we trained it now reduced the amount of legal moves by the hundreds like it will be mentioned ahead, and it was self learned.* Illegal move verification was removed due to the increased times that it took to pre-process the data and train the model with. This happened because instead of loading 12 UINT's we were loading 75 (63+12).

Model's Understanding of the Rules: It is important to clarify that the model does not possess explicit rule-based understanding of chess. Rather, it learns legal move patterns implicitly from the training data and from the valid move masking during inference. While it may appear to follow the rules, its knowledge is probabilistic and derived from frequency patterns in data, not from a true grasp of the game's logic.

Inability to Execute Checkmates: The model lacks the capability to perform checkmate sequences. This is because mating combinations often require multi-move foresight, which is not captured in our single-move prediction CNN model. Without any form of depth search or planning mechanism, the model cannot evaluate future consequences of a move beyond its immediate context.

Pawn Bias: An interesting tendency we observed is the model's preference for pushing pawns, especially in the opening and mid game. This likely reflects the frequency of pawn moves in early-game data. However, without strategic evaluation or learning from consequences (as in reinforcement learning), the model may over-prioritize such moves even when they lead to positional weakness.

IV. Data description, visualization and pre-processing

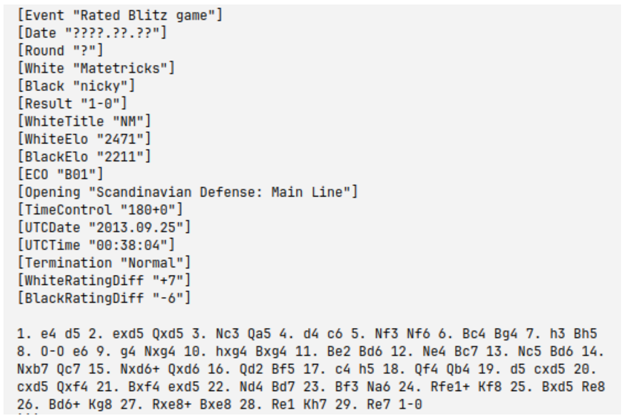
As mentioned before, we ought to create a chessbot that imitates the human playing style by teaching a Neural Network human moves extracted from real games played in Lichess, an internet chess platform that allows humans to find other opponents to play chess against, *basically it would get the highest probability move to be played by a human, in a certain chess position, due to it's human games training*. Therefore, from our dataset, we need the matches played and the elo rating of the players, to filter the games by high skill of play, which will allow us to extract board positions and moves played by a human, this to hopefully get a better and higher accuracy of play.

A. Dataset description and visualization

Initially we followed a dataset found on Kaggle: https://www.kaggle.com/datasets/arevel/chess-games, however we decided to shift to another dataset found on https://database.nikonoel.fr/, nonetheless both are based in the same initial dataset, games played on Lichess https://lichess.org/. This change sparked due to how we pre-processed the data. During the pre-processing, we extract games from specific ELO brackets, users with more than 1900 ELO, to diminish the number of bad moves represented, however, due to the low number of games in such scenario and the size of the dataset (6.5 millions games), the pre-processing became time consuming. Therefore, we decided to switch to a dataset that consisted of elite player games (games from users with at least 2200 ELO) that better matched our needs. With this change, our pre-processing pipeline was optimized, allowing us to extract 10 million moves made by black and another 10 million moves made by white in 45 minutes, where previously we were extracting around 2.5 million moves in around the same time frame due to the ELO constraints - (*This number is an approximation, having in mind that a game of chess has an average of 35 moves per side - 70 with both sides combined*).

This decision also prevented us from needing to clean our data, since that bullet games also existed in the previous dataset (*Games where each side just has 1 minute to play the entire game*) and other blitz games modes, where players have slightly more time, however in these quick format games, bad moves are commonly played, so we had two options, either we accepted these games or we had to filter them out, but, changing to the new dataset brought us automatically to this decision since this new dataset already filters those games out, so we end up with a higher quality dataset.

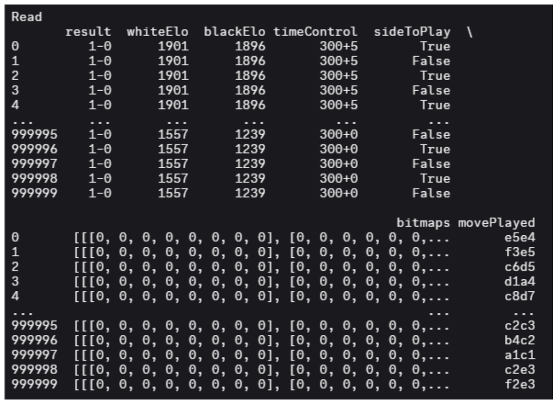
This dataset consists of a collection of games, each with information relevant to the played game and the game PGN format (moves made in the games):



```
[Event "Rated Blitz game"]
[Date "????.??.??"]
[Round "?"]
[White "Matetricks"]
[Black "nicky"]
[Result "1-0"]
[WhiteTitle "NM"]
[WhiteElo "2471"]
[BlackElo "2211"]
[ECO "B01"]
[Opening "Scandinavian Defense: Main Line"]
[TimeControl "180+0"]
[UTCDate "2013.09.25"]
[UTCTime "00:38:04"]
[Termination "Normal"]
[WhiteRatingDiff "+7"]
[BlackRatingDiff "-6"]

1. e4 d5 2. exd5 Qxd5 3. Nc3 Qa5 4. d4 c6 5. Nf3 Nf6 6. Bc4 Bg4 7. h3 Bh5
8. O-O e6 9. g4 Nxg4 10. hxg4 Bxg4 11. Be2 Bd6 12. Ne4 Bc7 13. Nc5 Bd6 14.
Nxb7 Qc7 15. Nxd6+ Qxd6 16. Qd2 Bf5 17. c4 h5 18. Qf4 Qb4 19. d5 cxd5 20.
cxd5 Qxf4 21. Bxf4 exd5 22. Nd4 Bd7 23. Bf3 Na6 24. Rfe1+ Kf8 25. Bxd5 Re8
26. Bd6+ Kg8 27. Rxe8+ Bxe8 28. Re1 Kh7 29. Re7 1-0
```

Fig. 1: First game present in the dataset found on https://database.nikonoel.fr/.



```
Read
        result  whiteElo  blackElo timeControl  sideToPlay  \
0          1-0      1901      1896       300+5        True
1          1-0      1901      1896       300+5       False
2          1-0      1901      1896       300+5        True
3          1-0      1901      1896       300+5       False
4          1-0      1901      1896       300+5        True
...        ...       ...       ...         ...         ...
999995     1-0      1557      1239       300+0       False
999996     1-0      1557      1239       300+0        True
999997     1-0      1557      1239       300+0       False
999998     1-0      1557      1239       300+0        True
999999     1-0      1557      1239       300+0       False

                                                   bitmaps movePlayed
0       [[[0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0,...       e5e4
1       [[[0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0,...       f3e5
2       [[[0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0,...       c6d5
3       [[[0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0,...       d1a4
4       [[[0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0,...       c8d7
...                                                    ...        ...
999995  [[[0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0,...       c2c3
999996  [[[0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0,...       b4c2
999997  [[[0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0,...       a1c1
999998  [[[0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0,...       c2e3
999999  [[[0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0,...       f2e3
```

Fig. 2: This is how we were previously filtering the data and part of how we received it from the CSV, also the way we processed it with bitmaps, whereas now, we get UINT directly and the move played

B. Pre-processing pipeline

We implemented a custom pre-processing pipeline to convert raw PGN chess game data into a format suitable for neural network training using the `python-chess` library [**?**]. Although the pre-processing pipeline had available game metadata such as player ELO, results, and time controls, we did not extract those features to pass to the training phase. Filtering by metadata was considered a downstream step for experimentation rather than a part of the core pipeline.

1) First approach: In our first approach, we cleaned the PGN files by removing annotations such as move comments, engine evaluations, and symbolic notations (e.g., `?!`), then tokenized the moves in Standard Algebraic Notation (SAN). We also chose to *skip the 4 initial moves*, in order to not prejudice the model for these due to the elevated amount of great openings that exists (This proved to maybe be an error afterwards, not only because the opening was always the same, but because it became completely irrational, at the same time we had the idea opened for a chess openings model, which going around again, it maybe didn't make sense, since openings are predefined responses or self righteous actions).

Each board position was encoded as a 13-channel tensor using bitboards:

- 12 channels represented the presence of each piece type (by color), example: White Queen was "Q":4, hence the 5th channel represents the white queen position on the board.
- 13th-channel that indicated the side to play.

Moves were also mapped to unique integer indices based on all possible `from-to` square combinations, serving as classification labels during training.

These channels, since they were bitmaps consisting of $(8 \times 8\ bits) \times 12\ channels$ bitmaps, were saved in twelve 64-bit unsigned integers and an integer 0 or 1 to represent the side to play in that state of the game, for memory efficiency purposes. *The* $8 \times 8 = 64$ *representing the board size and the 12 representing the amount of channels excluding side to play*.

To guide the model's predictions, we also computed valid move masks for each board state using the legal move generator from `python-chess`. These masks were stored in fixed-size $(63,)$ `uint64` arrays (covering 64×63 bits), marking legal moves and masking invalid ones. *64*63 representing a square in the chessboard for the start and the 63 the remaining squares you could move to, since you couldn't count a move as staying in the same spot.* However we decided to move away from this choice, because we thought it wasn't worth it, since our model learned on it's own to not make invalid moves, and if we did implement that specifically, it would make the training process much more time consuming.

2) Final approach - Changes made from the first approach: In our final approach, we decided to not clean the PGN data ourselves and decided to use a module also offered by `python-chess` that was capable of reading and processing PGN data.

Previously, we were skipping the first 4 moves in each side, in order to not punish the model for them, so that we wouldn't unfairly prejudice the model due to the elevated amount of great openings that exist. However, we decided to revert that decision to allow our model to, in a way, also learn openings instead of using, as proposed before, one from a book, since our objective was to make a: **Neural Network based Chess bot**, with no external help, *also because in the beginning we considered making a different model just to deal with the openings, but in reality it wouldn't really be a model because they are already predefined basically, and once you take a new approach you're leaving the opening principles and swapping to the other model, so it was more interesting for us to allow or train the model in such way, that it could follow those openings by learning through moves instead of having them directly fed into it.*

Other than that, we also disregarded creating an extra channel that held the side to move, thereafter we decided to create two models in order to slightly simplify our problem, one model that only played as white and one model that only played as black. There were two main reasons for us to choose this approach and train two models for each side to play (black and white). Initially we thought it would be way less confusing logically, and also for the model to learn just from one side to play, for example, pawn pushes and piece positioning, due to the bitmap being linear to each of them. Finally, the play styles also change a lot, hence white being the first side to move, it usually means it is the one that dictates the pace and style of the game, from passive to aggressive, from controlling to strategical, therefore it would be better for us, if the model could focus on only one of these ways.

Another critical move was played to better optimize the training of our model. Since the game of chess has **4032 possible moves (moving from one of 64 squares to one of the remaining 63 squares,** 64×63 **moves)**, we were representing each move in the outputs of our model, effectively having 4032 classes, which is not a good position for us to play, so we decided to create a class only for the moves that are really played in our dataset, reducing to ca. 1950 classes. This change allowed our model to reduce the loss slightly faster after each epoch.

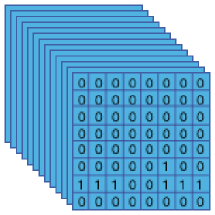


Fig. 3: Pre-processing result - 12 layers of 8x8 bitmaps, that represent the position of each piece by color

V. Machine Learning Algorithms

A. Possible choices

Based on the state of the art, we have various algorithms that could fit this problem the best. We could follow Stockfish's footsteps, by creating a tree-search that uses a hand-crafted heuristic function, we could follow Alpha-zero's footsteps that instead of an hand-crafted heuristic function, it uses a deep neural network to use as an heuristic function that is trained through reinforcement learning against itself, or we could follow Maia's steps, **which consists of a 6-layer CNN that learns from human played moves.** Since what we aim to achieve is the same objective as Maia's, we decided to follow their steps, creating a CNN that given a position, outputs the most probable human move for that ELO bracket. What further solidified our choice was, as mentioned before, a smaller-scale chessbot that has the exact same objective created by *Nikolái Skripkó*.

B. Chosen CNN Architecture

Following the steps of Nikolái Skripkó [4], we created a Neural Network that consists of:

- 12-Channels Convolutional layer - With a 3-wide kernel and a padding of 1 element, that outputs a 64 channel 8x8 board that passes through a batch normalization function followed by a ReLU activation function.
- 64-Channels Convolutional layer - With a 3-wide kernel and a padding of 1 element, that outputs a 128 channel 8x8 board that passes through a batch normalization function followed by a ReLU activation function and then the outputs are flattened to produce a vector of $8 \times 8 \times 128$ values.
- A fully connected layer with 256 nodes that use ReLU as an activation function.
- An output fully connected layer with `num_classes` nodes.

`num_classes` is a variable that represents the number of distinct moves found in the pre-processing of the dataset used. With this, we made a Neural Network that predicts the played move in the current position, so we are facing a classification problem and as such, during the training loop we decided to use a CrossEntropy Loss Function.

C. Hyper-parameters

In this architecture, we have several major hyper-parameters:

- Batch Size
- Learning Rate
- Convolutional Layer Kernel Size
- Convolutional Layer Kernel padding
- Number of Convolutional Layers
- Number of Channels in each Convolutional Layer

Due to the long training process we had to choose which hyper-parameters to test for and which parameters to keep mostly the same from the beginning.

So we decided to test how our model reacted to various combinations of Learning Rates and Batch Sizes.

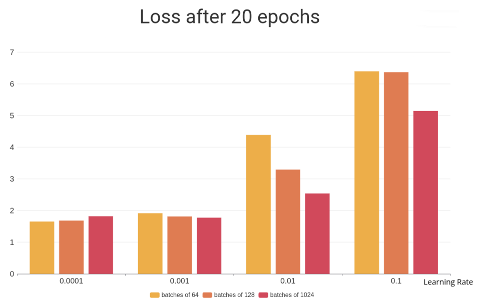


Fig. 4: Loss amounts after 20 epochs with different batch sizes

From figure 4, we saw that a larger learning rate was detrimental to our problem. Both a learning rate of 0.0001 and 0.001 had similar results for either batch size, however the minimum loss after 20 epochs was achieved with a learning rate of 0.0001 and a batch size of 64 elements. Therefore, we decided to use those values for their respective hyper-parameters in our model training loop.

It should be noted that, in order to speed up this evaluation, a smaller dataset, subset of the original, of 2.5M examples was used.

VI. Model Comparison and Performance Evaluation

A. *Previous Iteration problems*

On our first approach, we faced a lot of logic-based problems that we ended up discovering and fixing in this iteration. Even though our model got a loss value of around 1.0 in that approach, we were still seeing a poor performance when we played against it, which did not match our expectation since we followed the footsteps of Nikolái Skripkó [4] and Maia [3].

However, we have found those problems to lie in both our pre-processing and training phase. Firstly in the pre-processing, sometimes we weren't properly saving the move played and the game state, we were saving the game state after playing a certain move, and the move played, so we were effectively training our model to learn which move was played to reach a position, and not what move to play in a certain position. Furthermore, we found out that since we were using uint64 to represent 8×8 bitmaps, when we saved those values, we saved the bits in Little-Endian format, however, when we read them during the training phase, we were reading them in Big-Endian format, which resulted in every 8 bits being mirrored, such that the game was mirrored row-wise, while the played moves weren't affected, so, for example, moving from square 63 to square 38 lost sense in the loaded board.

Both of this problems combined, lead to our model not properly learning anything useful in the context of a chess game, however after these bigger problems being ironed-out, we quickly saw major improvements.

After testing for learning rate and batch size, we decided to also experiment with varying kernel size. From this experiment, there is a noticeable improvement in the minimum loss value achieved with a larger kernel size, as seen in 5. After 20 epochs, with a 3x3 kernel, the model achieved a loss of 1.6435, with a 5x5 kernel, it achieved a 1.5557, and with a 7x7 kernel, it got the lowest loss value at 1.4718. We believe that this effect occurs since that, in a chess game, a player has to keep track of the entire position instead of just the local positioning, and a kernel of 7x7 is better suited to do so instead of a smaller 3x3 kernel, however, we cannot verify this assumption. Nonetheless, in our final model we went ahead and still used a 3x3 kernel for our first model, and subsequently we also trained a model with a 7x7 kernel, but with less epoch's.

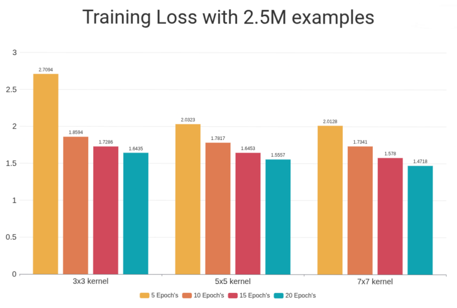


Fig. 5: Training loss relative to epochs amount and in different kernel formats

B. *Training Phase*

To create our model we ued PyTorch and to train it we used another dataset with 10 million examples. From that dataset, we also extracted 25k examples to be used as a validation dataset, however, these specific examples were never used to train the model. Nonetheless, since all chess games start in the same state, it is guaranteed that there should be positions in this subset that also appear in the training dataset, so there should be an overlap between the training dataset and the validation dataset.

Algorithm 1 Training Loop

```
1:  Procedure
2:  for $epoch \leftarrow 1$ to $NUM\_EPOCHS$ do
3:      for $batch \leftarrow dataloader.get_next_batch()$ do
4:          $bitmaps, expected \leftarrow batch$
5:          $prediction \leftarrow model.predict(bitmaps)$
6:          $loss \leftarrow CrossEntropy(predictions, expected)$
7:          $loss.backpropagation()$
8:      end for
9:      $validation\_dataset \leftarrow get\_validation\_dataset()$
10:     $loss, accuracy \leftarrow evaluate(validation\_dataset)$
11:     if $divmod(epoch, 5) = 0then$ then
12:         $save\_weights(model)$
13:     end if
14: end for=0
```

C. Training Results

By training a model with 10M examples after 80 epochs, using a 3x3 kernel, we acquired a model that reached 40.42% accuracy and 2.07 loss through the examples contained in the validation subset. In the training dataset, our model reached a loss of 1.73 and an accuracy of 47.08%. Accuracy is calculated by dividing the number of accurate predictions, which are defined as the class predicted of being the move played, by the total number of examples in the dataset.

$$Accuracy\,(\%) = \frac{correct\ predictions}{length\ of\ dataset} \times 100$$

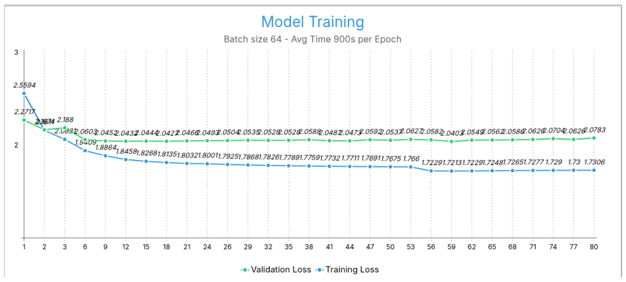


Fig. 6: Average batch loss per epoch

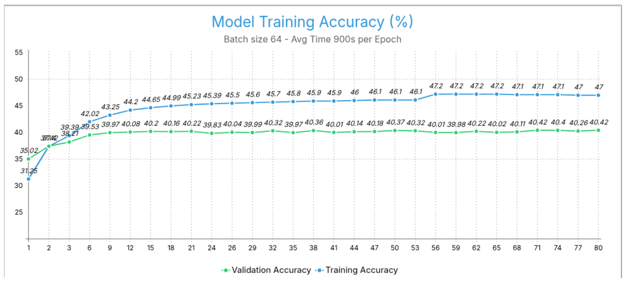


Fig. 7: Model accuracy during training - % of correctly classified moves in training

It should be noted that in the first epoch, our model was already at 35% accuracy, however it does start with 0%, but due to the sheer number of examples, the model is able to reach a "good" accuracy in the first epoch.

These results seem to indicate a bad model, since the accuracy is very low, however we believe that accuracy may not be a valid indicator of performance in this scenario, for the fact that in the same position many moves can be considered good/human moves, so our model could predict a move that is considered a human move but since that in that example we were expecting a specific move, we would end up considering it a bad prediction. Fair accuracy measure would be to have a list of moves made by a human in each position, effectively creating a map of position-moves played, however such a measure was deemed not feasible to achieve, due to the large memory requirement. Furthermore, in our training, we registered a minimum loss at **59 epochs with a loss value of 1.72131**, hence this would most probably be the Chessbot we would deploy, however the loss differences aren't significant, therefore if a higher validation/training accuracy is shown in other epochs we could also move to those model weights, however in our case, our model at epoch 59 really looks like it really is the best model.

By training a model with also 10M examples after 35 epochs, using a 7x7 kernel, we acquired a model that reached 41.75% accuracy and 2.0229 loss through the examples contained in the validation subset. In the training dataset, our model reached a loss of 1.6077 and an accuracy of 51.18%.

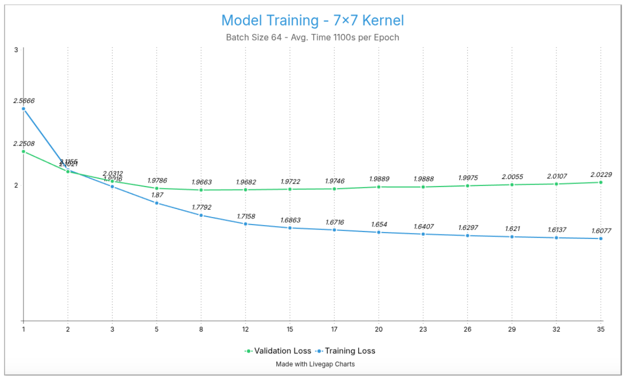


Fig. 8: Average batch loss per epoch

D. Test Games

The following evaluations were made by using https://chessigma.com and https://chess.com game evaluation tools. Both these tools employ Stockfish to give an evaluation of each position and moves made, by comparing the moves our bot made versus the best considered move that Stockfish achieves, furthermore it also uses Stockfish heuristic function to give a prediction of who has the better position. This allowed us to more accurately understand how well our model performed.

To test our model, we decided to also use https://chess.com available bots that attempts to play a game at a predefined ELO bracket, and we got the following results:

TABLE I: Semi-Slav Defense Accepted

Accuracy:	68.1%
Evaluated ELO:	900
Opponent ELO:	250 (Beginner level)
Opening:	12 book moves
Result:	Draw by repetition
Total Moves (per side):	58

TABLE II: King's Indian Defense: Bayonet Attack

Accuracy	66.4%
Evaluated ELO	950
Opponent ELO:	400
Opening:	4 book moves
Result:	Stalemate
Total Moves (per side):	111

TABLE III: Nimzo-Indian Defense: Gligorić, Bronstein Variation

Accuracy:	66.4%
Evaluated ELO:	1050
Opponent ELO:	700 (Played as 1500)
Opening:	4 book moves
Result:	White wins by checkmate
Total Moves (per side):	57

TABLE IV: Sicilian Defense: Nyezhmetdinov-Rossolimo, Fianchetto Variation

Accuracy	57.6%
Evaluated ELO	700
Opponent ELO	1000 (Intermediate level)
Opening:	9 book moves
Result:	White wins by checkmate
Total Moves (per side):	39

TABLE V: Sicilian Defense: Nyezhmetdinov-Rossolimo, Fianchetto Variation

Accuracy:	83.9%
Evaluated ELO:	1650
Opponent ELO:	2200 (Played as 2350, Expert level)
Opening:	9 book moves
Result:	White wins by checkmate
Total Moves (per side):	37

Sometimes the bot is getting a bit of rating in the end game, because it cannot find mate it just goes around getting good moves, but not forcing mate, even with mate in 1 on the board it might not find it, we noticed that the sacrifices it does sometimes are good, while others happen because it's expecting something in return but it doesn't play out well (Although now those scenarios are reduced). *The bot is playing better but obviously if we diverge from normal openings it's performance goes down, doesn't mean it won't win.* Also when analyzing games in chess.com the accuracy given is influenced by the rating of the bot chosen, so the real accuracy played is more accurate for a measurement than the accuracy given by chess.com.
In terms of testing, when it comes to the level of play in chess bots, it's noticeable with just one game, how smart the model actually is.

Now we're testing with a 7×7 kernel. Endgames with the 7×7 setup are still weak (but now we can find mates, and it's still much stronger, we say it's weak because it often ignores obvious plays, for some of less value but still valid), although early-game performance is nearly perfect. Overall, the bot didn't commit many blunders.

For this version of the bot, we applied batch normalization after each convolutional layer, and we added 3-element padding to ensure the output remains an 8×8 chessboard.
The bot delivered its first checkmate!

TABLE VI: Sicilian Defense: Najdorf Variation

Accuracy:	90.0%
Evaluated ELO:	2250
Opponent ELO:	250 (Played as 2150)
Opening:	6 book moves
Result:	Black wins by checkmate
Total Moves (per side):	64

TABLE VII: King's Indian Defense: Fianchetto, Karlsbad, Panno, Uhlmann-Szabó System

Accuracy:	94.3%
Evaluated ELO:	1600
Opponent ELO:	400 (Played as 1100)
Opening:	9 book moves
Result:	Black Wins by checkmate
Total Moves (per side):	27

TABLE VIII: Queen's Indian Defense: Fianchetto, Nimzowitsch Variation

Accuracy:	84.3%
Evaluated ELO:	1600
Opponent ELO:	700 (Played as 1150)
Opening:	5 book moves
Result:	Black Wins by checkmate
Total Moves (per side):	72

TABLE IX: Queen's Indian Defense: Fianchetto, Nimzowitsch Variation

Accuracy:	84.3%
Evaluated ELO:	1600
Opponent ELO:	1000 (Played as 1150)
Opening:	5 book moves
Result:	Black Wins by checkmate
Total Moves (per side):	72

TABLE X: Alapin Sicilian Defense

Accuracy	91.5%
Evaluated ELO	2450
Opponent ELO	1000 (Played 1900)
Opening:	4 book moves
Result:	Black wins by checkmate
Total Moves (per side):	50

TABLE XI: Sicilian Defense: Open, Najdorf, English Attack

Accuracy:	80.9%
Evaluated ELO:	1500
Opponent ELO:	2200 (Played as 2000, Expert level)
Opening:	8 book moves
Result:	White wins by checkmate
Total Moves (per side):	37

We also played against Nikolái Skripkó's bot, which was one of our biggest inspirations, the PGN's are all saved in the GitHub doc's folder https://github.com/GCapaross/ChessBot/blob/main/docs/PGNSagainstNikolai.txt. Regarding these games we can conclude that our chess bot is much better, we won 4/5 games and drew the last one. We had major winning positions in all of them. In comparison, our bot is able to checkmate while his, isn't. Our bot commits way less blunders, and makes preparation and tactical moves. The accuracies are all described in the document in GitHub.

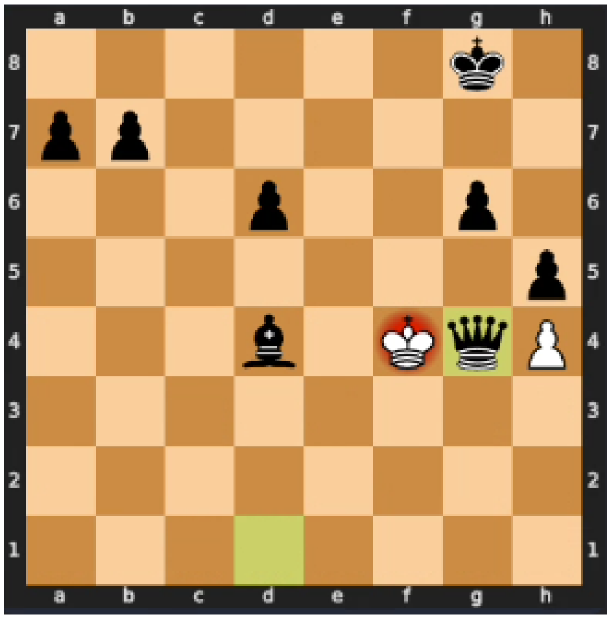


Fig. 9: Example of one of the games against Nikolai's model. Black (we) won by checkmate

We considered it wouldn't be relevant to test against higher level models because the games would be too short, or our moves would be way too induced by theirs, and therefore having only a restricted number of possibilities the accuracy would always increase (it would still be high but not really representative of the bot's actual play level). We understood that when testing with above 2500 elo, where after blundering once the game outcome is already determined. For example: Near the middle / end game against high elo Stockfish, we were short on pieces and every move we made would lead to a loss, and since we were already in a bad position, through Stockfish evaluation, even our bad moves were seen as just "ok", "good move", "best move" or "forced move", this to say, sacrificing a queen isn't seen as bad if you're already getting force mated in "x" moves.

These are just some examples of games beyond the other tests that were made, in conclusion, our bot has perfect opening principles and responses, it is good against low elo bots, but then it doesn't know how to win, if it does happen is due to luck. (This happens probably because of overfit, the bot gets too used to playing openings that always happen and mid games that always go through, but then doesn't really have have experience in endgames). Against intermediate opponents it's more of an equal game all throughout, but once again in the

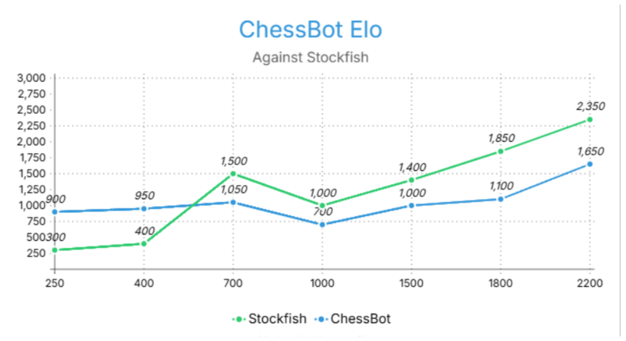


Fig. 10: Evaluation chart of the chess bot performance against Stockfish at different elos. The values in the x axis represent the difficulty set for Stockfish, and the lines represent the values at which they played at, evaluated by also Stockfish at elo 3800 (He compares the moves made with his own choices).

endgame blunders start to stack up. Against high elo players, our bot has much more accuracy and rating, we suppose it's because the moves are more normalized and logical, and therefore it follows more accurately the dataset in response. The average number of moves played was around 50 per side during these tests.

In comparison with our previous model, the improvements are substantial and not even comparable, despite not conducting direct matches between them. Some key differences include:

- The previous models would always play the same opening, regardless of the opponent's response.
- The old model frequently made illegal moves (reaching thousands in a single game), whereas the new model only made around five illegal moves per game on average.
- The previous bot rarely engaged in trades or captured pieces unless it was forced to.
- Special moves like *en passant* or castling would only be executed if they were among the few legal options available.
- The old bot's ELO rating never surpassed 100, and its accuracy, evaluated by Stockfish, never exceeded 20%.

E. *New Information and Insights Gathered*

While we will address future steps in a dedicated section, it's worth reflecting on how our approach might have changed if we were to start over with the knowledge we now possess. As a starter we would've been way more careful with mitigating pre-processing errors, a way to do that would've been if we had started with a simpler dataset from the beginning. What we got from this is that the way that information is given to the model really can affect how it plays out, and one slight logical change in bits, especially when it comes to something as specific as positions in chess games, where one bit might symbolize a whole different position. Of course here we're not accounting for architecture options since it wouldn't be fair. We learned a lot in terms of practical use of machine learning techniques, but also we shouldn't disregard the knowledge learned about python's libraries, which also gave us headaches sometimes when processing data, due to the logic being opposite to what we were expecting. We also developed more chess skills, beyond testing the bot with other engines online, we also tested it with ourselves, and in order to know if something is bad, we need to be slightly good at it.

F. Response to Professor's Feedback

During the second deliverable, the professor made 3 pertinent points / questions: 1. "Divide your big problem in several small problems, and define a strategy for each one. Some decisions may need justification." 2. "Explore state of the art implementation, and mitigate some of the difficulties based on that." 3. "Why not using the board images as input of the system?"

Regarding question 1, we did try to simplify, by making two main models, one that answers the question *"What's the best piece to move in this position?"* and another one that answers *"Whats the best square to move this piece to?"*, however we felt that making a model like that would not give the best results since, in our approach, we return a list of the most probable moves, at the cost of being more complex. Those two models ideologies would collide, for example: if the first model tells us to move the piece in the "e4" square and the second model tells us to move that same piece to the "e5" square, if that's an invalid move, should we go to the next most probable piece to move or should we find another most probable square to move to? This is a problem that cannot be easily solved, since if we go with either approach, the model can easily blunder, however, if we have a classic model that gives a list of probable moves, it allows the model to "rank" each move without getting stuck to a specific piece or without disregarding possible moves from that piece. Also, in a way, we simplified our problem by making a model for only black pieces and one for white pieces.

The second question was a hot topic inside the group, because there are two main ways people create chess bots, one is a tree search algorithm, and another is a transformer based deep learning architecture. The former one was disregarded for the sole reason that we wanted to explore new horizons, we could've made a tree search algorithm with a Neural network heuristic function. The latter was considered to be impossible in both the time frame we had and the capabilities we have in this area.

Finally, we believe that the proposed model would require a much higher degree of complexity. By following that path, we would pass images of a chess board to our model, hence effectively having something that has two functions: extracts positions from a picture and that calculates the best move to be played. By following our approach, we would only focus on the last problem: "Given a position, what's the best move to be played?".

VII. Next steps / Challenges and Considerations

For the future of this project, a few changes should be made for achieving better results. As noticed before, our model is great at openings, it's decent at midgame and it's underperforming in the endgames, therefore these two game scenarios need to be strengthened. Since we already have a model that can decently play, it would be a great candidate to be the initial model in a reinforcement learning strategy, in order to iron out the lacking parts and optimize the parts where he already shines. However following this approach would steer away from the initial objective, nonetheless it is a great next step.

As next steps, a tree-search algorithm path could also be followed, such that it would make the model truly shine especially in the areas where it lacks the most: the endgame, but like the previous strategy, it could divert from the initial objective. We could also try to add a second output and more layers for it, as a sort of evaluation like Stockfish does. Also the tree-search, minmaxing and alpha beta pruning would be really useful in endgames, since the plays are more simplified and there aren't as many options. *Chess is already solved with 7 pieces on the board, meaning with 7 pieces overall, all possibilities are analyzed by super computers, and all outcomes are viewed.*

In order to not diverge from the base choices, there are paths that could be followed to improve this Chessbot, such as trying new, more powerful, architectures, employing a bigger dataset and/or regulate the number examples and classes currently present in the dataset, ie. diminishing the number of opening states and increasing the number of middle-games/end-games, since we do believe that endgames are also lacking thanks to this dataset imbalance.

In terms of tests, it would also be useful for the next steps, to test and optimize certain scenarios and positions like endgames, knight and bishop checkmates and other very theoretical positions, something that could be solved with more training, better training, or another model.

VIII. Conclusion

Through this project, we demonstrated that a convolutional neural network trained on high-quality chess data can play reasonable chess, particularly in the opening and early midgame, and especially with low amount of computational hardware.

Beyond model performance, this project served as a comprehensive learning experience in several key areas. We gained practical experience in handling large datasets, implementing and optimizing pre-processing pipelines, and designing deep learning architectures suited for structured game data like chess. We also deepened our understanding of how neural networks can learn strategic patterns from raw data without explicit rule-based programming.

Additionally, we confronted the challenges of integrating domain-specific knowledge (like legal move enforcement and game rules) into a machine learning framework, and we learned the trade-offs between model complexity, interpretability, and training efficiency.

Most importantly, we learned to balance ambition with practicality. While our initial vision was broad and idealistic, we adapted to real-world constraints and iteratively improved both our pipeline and model. We now have a much clearer grasp of how chess engines function at different levels, from rule-based search trees to human-like pattern recognition, and we leave this project with valuable technical skills and a stronger intuition for how machine learning can be applied to complex decision-making tasks like chess.

References

[1] N. Cheerla, "AlphaZero Explained," [Online]. Available: https://nikcheerla.github.io/deeplearningschool/2018/01/01/AlphaZero-Explained/

[2] Leela Chess Zero, "Technical Explanation of Leela Chess Zero," [Online]. Available: https://lczero.org/dev/wiki/technical-explanation-of-leela-chess-zero/

[3] Maia Chess, "Maia Chess: A Human-Like Chess Engine," [Online]. Available: https://www.maiachess.com/

[4] Skripkon, "Chess Engine," GitHub repository, [Online]. Available: https://github.com/Skripkon/chess-engine

[5] McIlroy-Young, R., Sen, S., Kleinberg, J., and Anderson, A., "Aligning Superhuman AI with Human Behavior: Chess as a Model System," *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2020. Available: https://doi.org/10.1145/3394486.3403219

[6] Panchal, H., Mishra, S., and Shrivastava, V., "Chess Moves Prediction using Deep Learning Neural Networks," *2021 International Conference on Advances in Computing and Communications (ICACC)*, pp. 1–6, 2021. Available: https://doi.org/10.1109/ICACC-202152719.2021.9708405

[7] Pilueta, N., Grimaldo, H., Jardiniano, M., and Garcia, M., "Chessbot: A voice-controlled chess board with self-moving pieces," *AIP Conference Proceedings*, 2022. Available: https://doi.org/10.1063/5.0108986