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, integrate board rules, and enforce legal move constraints. 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. Introduction

ARTIFICIAL intelligence in chess has evolved rapidly, from classical search-based engines like Stockfish to modern neural models like AlphaZero. Our project focuses on creating a chess-playing neural network that leverages data-driven learning and game theory. It aims to understand board states and play competitively, while also imitating human-like decisions.

II. Project Vision and Methodology

We propose to achieve a chess-playing model that mimics human strategies by learning from datasets extracted from human-played games, to achieve this goal, we decided employing a neural network to be the root of this project, as such, the architecture of the model as a whole is considered a main task for the success of this project.

III. Architecture Options

Several architectures are under consideration in our design of a neural-network-enhanced chess engine. Each approach varies in how it interprets the game state and selects moves.

A. Tree-search

Tree-search algorithms form the backbone of classical chess engines like Stockfish. These engines evaluate possible future positions by traversing a game tree. The most commonly used techniques are [2]:

- **Minimax algorithm**: recursively explores all possible moves, assuming the opponent plays optimally.
- **Alpha-beta pruning**: optimizes the minimax algorithm by eliminating branches that cannot influence the final decision.

Tree-search can be combined with a neural network to evaluate positions (value) or suggest moves (policy), as done in AlphaZero and Leela Chess Zero. In such systems, the NN provides a heuristic evaluation of a board state, guiding the search process toward promising branches.

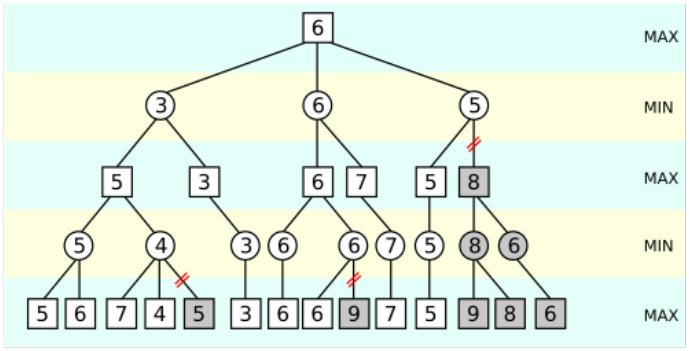

Fig. 1. Alpha Beta pruning with minimaxing

B. CNN-based Model

Convolutional Neural Networks (CNNs) treat the chessboard as a 2D grid, making them well-suited for recognizing spatial patterns. A board position is encoded as a tensor, where each channel may represent the presence of a specific piece or other contextual information (e.g., castling rights, move count). The CNN then processes these inputs to predict:

- A **policy vector**: representing probabilities over possible moves.
- A **value scalar**: estimating the win/loss probability from the given position.

This approach has been used in AlphaZero and in newer versions of Stockfish (since version 12), where CNNs augment traditional evaluation with learned positional understanding. The model's output can be used directly to select a move (highest probability) or as a guide for further tree search [3], [4].

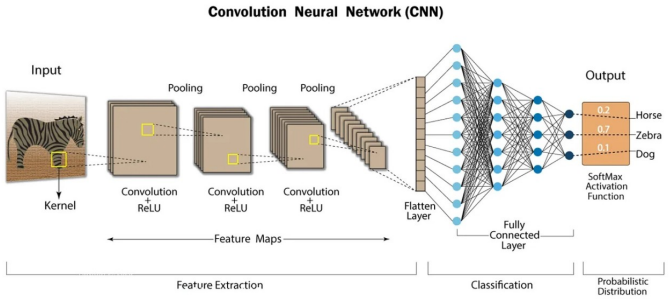

Fig. 2. CNN example - In our case we use boards instead of images, and they can be seen for example as a 8x8x13 image, meaning images with 64 pixels and 13 channels

C. Transformer-based Model

Transformers, originally developed for natural language processing, have been adapted to chess by encoding move

sequences as tokenized input. Instead of focusing solely on the current board state, transformers process the entire game history or long move sequences, capturing temporal dependencies and higher-level strategies [5]. Key components include:

- **Self-attention mechanisms**: allow the model to weigh the relevance of each past move.
- **Positional encoding**: preserves the order of moves in the sequence.

Transformers are more computationally expensive but are especially powerful at modeling openings and strategic plans. Leela Chess Zero and some recent research models (like GPT-Chess or ChessGPT) use transformers to predict the next best move in context [6].

D. Reinforcement Learning Agents

Reinforcement Learning (RL) allows the model to learn from self-play, using trial-and-error rather than supervised labels. The core components include:

- A **policy network**: decides which move to play in a given position.
- A **value network**: estimates the outcome from that position.
- A **reward signal**: assigned based on game outcomes (win = +1, loss = -1).

AlphaZero popularized this method by using deep reinforcement learning and Monte Carlo Tree Search (MCTS) to achieve superhuman performance in chess. Over time, the agent improves its play without needing human data, discovering novel strategies through exploration [3], [16], [17].

E. Move Selection Strategy

After the model outputs a policy vector, we consider two main strategies for selecting a move:

- **Highest probability move**: Selecting the move with the maximum probability yields the strongest play, useful for competitive or ”try-hard” modes.
- **Probabilistic sampling**: Sampling from the move distribution introduces human-like variety and imperfection, which can make the bot more fun and less predictable to play against.

We may dynamically switch between these strategies based on the use case or selected difficulty level.

F. Classical Elements

While our focus is on neural models, we also consider the integration of classical chess elements:

- **Minimax with Alpha-Beta Pruning**: Instead of relying on hand-crafted evaluation functions, we can use a neural network to evaluate board positions during minimax search, combining human-learned insight with deep search [7].
- **Opening Book Rules**: For the early game, we can join our model with well-established opening databases to avoid wasting compute on already-known optimal lines. These books can also help keep the bot within familiar human theory.

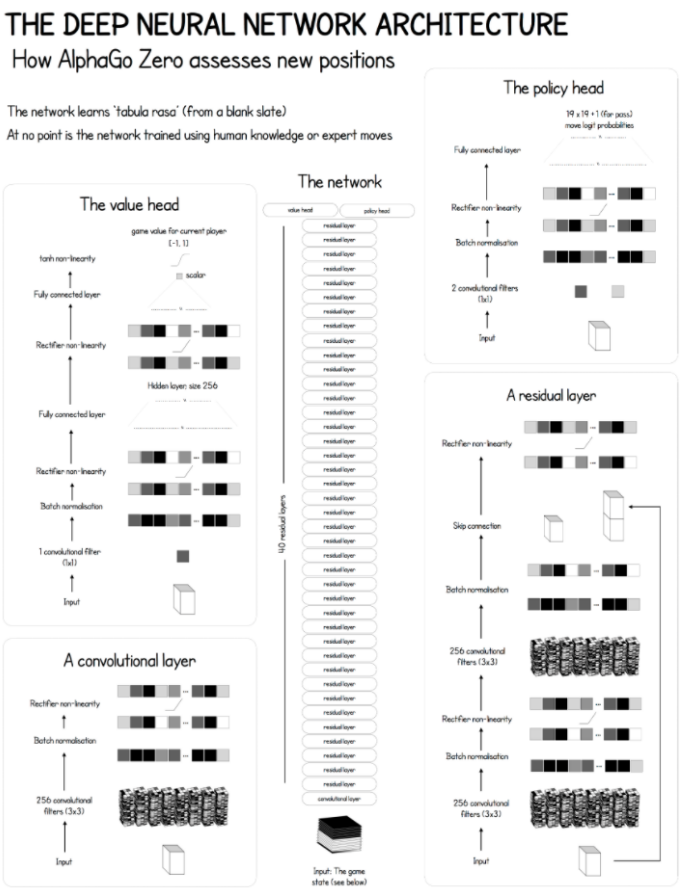

Fig. 3. How alpha zero accesses new positions

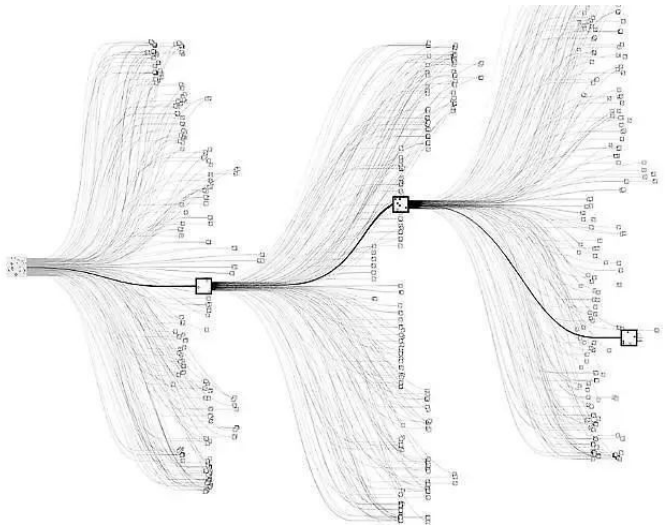

Fig. 4. Monte Carlo Tree Search

IV. Model Limitations and Feasibility

Although many of the architectures discussed above offer strong performance and are used by world-class engines, not all of them are equally viable for our setting. In this section, we evaluate the feasibility of each approach, considering the constraints of our project timeline, team size, and computational resources (Having in consideration we have to do the

best bot possible with the available resources).

A. Tree-search (Stockfish-like engines)

Tree-search engines such as Stockfish evaluate millions of positions per second using deep alpha-beta search. Training or tuning such an engine involves vast computational cost:

- Requires high-performance CPUs to evaluate billions of positions efficiently.
- Deep search trees need large amounts of RAM and parallelization for competitive speed.
- Combining tree-search with neural networks (as done in modern Stockfish) demands GPU acceleration and optimized codebases.

Conclusion: While powerful, developing a competitive search-based engine from scratch is infeasible in our timeframe without specialized hardware and a large engineering team.

B. CNN-based Model (Chosen Approach)

Convolutional Neural Networks offer a middle ground between performance and practicality:

- Efficient at capturing board patterns using relatively shallow architectures.
- Compatible with consumer-grade GPUs for training.
- Can be trained with supervised datasets (e.g., PGN files), reducing the need for self-play.
- Preprocessing and training can be modularized across a small team.

Conclusion: This is the approach we selected. It provides a good balance of learnability, performance, and feasibility, and is achievable within our team structure and hardware limitations.

C. Transformer-based Model

While transformer models have shown impressive results in chess-like tasks:

- They are significantly more computationally expensive than CNNs due to self-attention mechanisms.
- They require large training datasets and extensive GPU time, often on multi-GPU systems.
- Models like ChessGPT or Leela-style transformers are built and trained by large teams with considerable resources.

Conclusion: Too complex and resource-intensive for our scope. We chose not to pursue transformers despite their promising capabilities.

D. Reinforcement Learning Agents (AlphaZero-style)

RL-based agents are highly appealing conceptually but have major limitations:

- Require massive self-play generation — millions of games must be simulated.
- Monte Carlo Tree Search (MCTS) is GPU and CPU intensive.
- Google's AlphaZero ran on TPUs with massive compute budgets.

Conclusion: RL is beyond our computational capacity and time budget. It could be explored later as a secondary phase or with pre-trained models.

E. Additional Constraints

Across all approaches, we face the following common limitations:

- **Training Time**: We only have around 8 weeks for both development and training.
- **Hardware**: Our access is limited to personal laptops.
- **Storage**: Large models and datasets (especially self-play data) would require storage capacities we don't currently have access to.

Final Justification: Given these constraints, we chose to implement a CNN-based supervised model using available PGN data. This allows us to focus on model design, preprocessing pipelines, legal move handling, and evaluation, all within the boundaries of what is achievable by a student team in an academic semester. Of course this doesn't mean that this approach will be the one followed until the end.

This hybrid approach, blending neural predictions, classical logic, and probabilistic move selection, allows us to build a bot that is both strategic and capable of human-like behavior.

V. Data and Preprocessing

A. Training Dataset Sources

To train our model, we used the dataset available on https://www.kaggle.com/datasets/arevel/chess-games, composed of 6.2M games played on an online chess website called https://www.lichess.org. This dataset is composed of a csv file with several columns with information about the game, such the ELO (a rating system) of the players, the time control and, most importantly, the game encoded in PGN format. Later on we might consider other datasets, extracted directly from chess.com or other websites, also opening datasets, "x"-ELO datasets...

B. Preprocessing Pipeline

This section describes our custom preprocessing pipeline, which transforms raw PGN data into a machine-readable format suitable for neural network training. The goal is to convert human-readable chess games into structured tensors and valid move masks.

1) PGN Cleaning and Parsing: Raw PGN data is first cleaned to remove metadata and annotations such as:

- Move comments (e.g., {...}),
- Engine evaluations (e.g., `[%eval]`),
- Move evaluations like `?!`.

We then tokenize the cleaned PGN to extract individual moves in Standard Algebraic Notation (SAN), discarding move numbers and formatting artifacts.

2) Board Representation: Each board position is encoded into a 13-channel tensor using bitboards:

- 12 channels represent the presence of each piece type (6 white + 6 black) on all 64 squares.
- A 13th channel stores the side to play as a binary flag.

This allows our model to efficiently process spatial patterns using binary bit manipulations.

3) Move Encoding: We designed a custom mapping to transform each move into a unique integer index. This is done by encoding all possible `from-to` square pairs (excluding same-square transitions), resulting in a consistent integer label per move (e.g., `e2e4` becomes index i). This index is used as the classification label during training.

4) Valid Move Masking: For each board position, we compute the set of all legal moves using `python-chess`. These moves are stored in a fixed-size binary array of shape $(63,)$ of `uint64` elements, which can be seen as an array of 64×63 bits (the number of possible moves in a 64 square board), each bit corresponds to a possible move from the MOVE_DICTIONARY space. Illegal moves remain masked as zeroes, and this array is used to filter predictions during inference.

5) Game Metadata: The CSV output could also retain game-specific metadata such as:

- Result (`1-0`, `0-1`, `1/2-1/2`),
- White and Black ELO ratings,
- Time control (e.g., `600+5`).

which enables optional filtering by skill level or format in downstream model training. However, this filtering can also be easily be done in the pre-processing phase.

6) Batching and Writing: To manage memory and performance:

- The PGN file is read in batches (10,000 rows per iteration).
- Outputs are streamed directly into a CSV to avoid RAM overflow.
- Game loading and move processing is wrapped with exception handling to skip malformed entries.

Note: While we initially considered filtering games by ELO, game phase, or outcome, we leave these as optional post-processing steps for future iterations.

For PGN parsing and board evaluation, we use the `python-chess` library [15].

```
start = 1 # To ignore headers
total = 0
READ_SIZE = 10_000
RUN = True
i = 1
try:
    with open("./dataset/processed/results.csv", "w") as f:
        writer = csv.writer(f)
        while RUN:
            ti = time()
            games_df = pd.read_csv('./dataset/raw/chess_games.csv', skiprows=start, nrows=READ_SIZE, names=HEADERS)

            print(f"Reading {i}-th batch of games...")
            # Result, WhiteElo, BlackElo, TimeControl, BoardPositions: uint_64[], move_played
            for index, game in tqdm(games_df.iterrows()):
                board = chess.Board()
                # Get PGN
                Result = game["Result"]
                WhiteElo = game["WhiteElo"]
                BlackElo = game["BlackElo"]
                TimeControl = game["TimeControl"]
                moves_string = clean_pgn(game["AN"])

                bitmaps = []

                tokens = moves_string.replace("\n", " ").split()
                moves = [token for token in tokens if not token[0].isdigit() and '.' not in token]
                sideToPlay = True # True for white, False for Black
                numMoves = 0
                for move in moves:
                    try:
                        move: chess.Move = board.push_san(move)
                        numMoves += 1
                        # if numMoves < 10:
                        #     continue

                        bitmap = board_to_tensor(board, sideToPlay)
                        possible_moves = get_possible_moves(board)
                        writer.writerow([
                            bitmap.tolist(), # bitmaps
                            MOVE_DICTIONARY[move.uci()[:4]], # Played move
                            # 0 for invalid 1 for valid; To keep constant size
                            possible_moves.tolist(),
                        ])
                        sideToPlay = not sideToPlay
                    except Exception as e:
                        print(f"Skipping bad move: {moves} - {move} #{e}#")
                        break

            start += READ_SIZE
            tf = time()
            total += len(games_df)
            i+=1
            print(f"Read {len(games_df)} in {tf-ti} | TOTAL: {total}")

            if 50_000 <= total:
                RUN = False

except Exception as e:
    print(e.with_traceback())
```

Fig. 5. Pre-processing code - doesn't include auxiliary functions - full code in - https://github.com/GCapaross/ChessBot

```
Reading 1-th batch of games...
10000it [01:15, 132.83it/s]
Read 10000 in 75.35231304168701 | TOTAL: 10000
Reading 2-th batch of games...
10000it [01:14, 133.81it/s]
Read 10000 in 74.81473231315613 | TOTAL: 20000
Reading 3-th batch of games...
10000it [01:15, 132.25it/s]
Read 10000 in 75.76021242141724 | TOTAL: 30000
Reading 4-th batch of games...
10000it [01:15, 131.85it/s]
Read 10000 in 76.00544452667236 | TOTAL: 40000
Reading 5-th batch of games...
10000it [01:15, 133.10it/s]
Read 10000 in 75.25275897979736 | TOTAL: 50000
```

Fig. 6. Pre-processing run

VI. ChessBot Training Overview

This section outlines the architecture and training pipeline of a convolutional neural network designed to evaluate chess board positions and predict optimal moves. The system leverages deep learning techniques tailored to structured board representations, enabling piece and move prediction directly from board states.

A. Frameworks

We will use:

- `PyTorch` for model design and training
- `python-chess` for handling board states and legal moves during data pre-processing pipeline
- `numpy, polars` for data manipulation

B. Preprocessing and Data Encoding

Each chess position is encoded as a $13 \times 8 \times 8$ tensor, where the first 12 channels represent the presence of each piece type (6 for white, 6 for black), and the 13^{th} channel indicates the side to move, but we are also considering 4 more channels to represent castling rights, a key move in chess openings. Positions are loaded from a CSV file using a lazy loading approach via the `Polars` library, enabling efficient memory usage and the handling of large datasets (over a million examples). The preprocessing function converts each JSON-encoded board bitmap, represented as an array of unsigned 64-bit integers, into a structured binary tensor format using NumPy and bit unpacking, which is then passed into the model during training.

Fig. 7. Pre-processing pipeline

C. Model Architecture and Training Loop

Our model, `CompleteChessBotNetwork`, that is used in this training setup, consists of two convolutional layers followed by two fully connected layers. Weight initialization is done using *"Kaiming"* and *"Xavier"* strategies for better convergence. The training loop uses the Adam optimizer with cross-entropy loss, and includes gradient clipping to prevent instability. Model checkpoints are saved every 5 epochs, and the final model is serialized to disk.

Training runs on GPU if available and is designed to scale with RAM-efficient batch fetching. This modular design allows for plug-and-play improvements in input representation, output granularity, and reward shaping for future reinforcement learning integration.

VII. Model Training and Integration

Our model is a classification problem solved through a supervised learning strategy, so the model must output a move to be played in the current state of the board. Our training process will be based on 2 expectations, we expect the model to:

- **Predict the played move based on the current board and side-to-play** - this is penalized using cross-entropy loss as we use a classification model.
- **If the move is not the move played, it must predict a valid move** - although penalized in the previous step, we reduce the loss by a fixed quantity to encourage it to play legal moves even if its not the expected move

A. Legal Move Masking

During training and inference, the model can output illegal moves, whenever this happens we aim to penalize the model by increasing the loss, effectively attempting to teach our model the rules of the game, however if this fails when deploying the model, we have to use our last resource of ignoring illegal moves when fetching the move to play.

B. Training made until now

Based on our decisions, we started building and optimizing our models, which also has had an effect on the decisions made here.

```
True
Using device:  cuda
100%|██████████| 51773/51773 [02:51<00:00, 302.72it/s]
Epoch 1 - 3.5338 | Time: 171.02689218521118
100%|██████████| 51773/51773 [02:50<00:00, 302.84it/s]
Epoch 2 - 2.5336 | Time: 170.96192598342896
100%|██████████| 51773/51773 [02:49<00:00, 305.70it/s]
Epoch 3 - 2.2681 | Time: 169.36138129234314
100%|██████████| 51773/51773 [02:49<00:00, 305.55it/s]
Epoch 4 - 2.1291 | Time: 169.44144797325134
100%|██████████| 51773/51773 [02:49<00:00, 305.53it/s]
Epoch 5 - 2.0450 | Time: 169.45404505729675
100%|██████████| 51773/51773 [02:48<00:00, 306.97it/s]
Epoch 6 - 1.9900 | Time: 168.66061568260193
100%|██████████| 51773/51773 [02:48<00:00, 306.48it/s]
Epoch 7 - 1.9508 | Time: 168.92704105377197
100%|██████████| 51773/51773 [02:49<00:00, 305.15it/s]
Epoch 8 - 1.9209 | Time: 169.66611075401306
100%|██████████| 51773/51773 [02:48<00:00, 306.87it/s]
Epoch 9 - 1.8971 | Time: 168.71227622032166
100%|██████████| 51773/51773 [02:49<00:00, 304.84it/s]
Epoch 10 - 1.8778 | Time: 169.83908343315125
100%|██████████| 51773/51773 [02:52<00:00, 300.21it/s]
Epoch 11 - 1.8618 | Time: 172.45528602600098
100%|██████████| 51773/51773 [02:50<00:00, 304.23it/s]
Epoch 12 - 1.8483 | Time: 170.18055033683777
100%|██████████| 51773/51773 [02:51<00:00, 302.49it/s]
Epoch 13 - 1.8364 | Time: 171.15522742271423
100%|██████████| 51773/51773 [02:49<00:00, 305.03it/s]
Epoch 14 - 1.8263 | Time: 169.73297119140625
100%|██████████| 51773/51773 [02:49<00:00, 306.31it/s]
Epoch 15 - 1.8172 | Time: 169.0251874923706
100%|██████████| 51773/51773 [02:53<00:00, 298.29it/s]
Epoch 16 - 1.8092 | Time: 173.5646812915802
100%|██████████| 51773/51773 [02:45<00:00, 312.00it/s]
Epoch 17 - 1.8018 | Time: 165.93786597251892
100%|██████████| 51773/51773 [02:49<00:00, 304.64it/s]
Epoch 18 - 1.7951 | Time: 169.9509949684143
100%|██████████| 51773/51773 [02:50<00:00, 304.21it/s]
Epoch 19 - 1.7891 | Time: 170.1909146308899
100%|██████████| 51773/51773 [02:49<00:00, 304.95it/s]
Epoch 20 - 1.7836 | Time: 169.7792990207672
100%|██████████| 51773/51773 [02:54<00:00, 297.36it/s]
Epoch 21 - 1.7786 | Time: 174.10879158973694
100%|██████████| 51773/51773 [02:48<00:00, 306.99it/s]
Epoch 22 - 1.7739 | Time: 168.64929223060608
100%|██████████| 51773/51773 [04:43<00:00, 182.43it/s]
Epoch 23 - 1.7698 | Time: 283.80107975006104
100%|██████████| 51773/51773 [04:56<00:00, 174.41it/s]
Epoch 24 - 1.7657 | Time: 296.8620948791504
100%|██████████| 51773/51773 [04:51<00:00, 177.54it/s]
Epoch 25 - 1.7620 | Time: 291.61624121665955
100%|██████████| 51773/51773 [03:21<00:00, 256.58it/s]
Epoch 26 - 1.7585 | Time: 201.78068947792053
 90%|████████▉ | 46618/51773 [02:31<00:16, 307.36it/s]
```

Fig. 8. Training phase

C. Observed Issues and Future Fixes

During this training phase, we observed the following issues:

- **Overfitting**: Smaller subsets of data showed overfitting after 20 epochs. We mitigated this by extracting more examples from the original dataset.
- **Data bias**: The dataset is heavily skewed toward popular openings. We consider balancing or shuffling lines in future runs and even ignoring the opening phase of the games.
- **Limited endgame knowledge**: Since many games end early and endgames are way more chaotic, rare endgame positions are underrepresented.
- **High loss**: Throughout our testings, we never got lower than 1.5 average loss per epoch, which may happen because a lot of the same positions have several possible moves and each example only employs one of them. We

consider diminishing the loss whenever the chosen move is a valid move, even if it is not the expected move.

Next steps: Potential improvements include adding castling flags, move counters, and en passant rights into the tensor encoding. We also plan to run a small reinforcement learning loop on top of the trained model.

VIII. Evaluation and Self-Play

In the first phase, we will follow a data-set validation partitioning, ie. we will extract never seen positions from our dataset and pass it to our model and calculate the accuracy. After we validate our model through dataset validation, we can test its effectiveness using several metrics we have several metrics other than dataset evaluation.

A. Metrics

- Win/loss rate against baselines - Such as, using our model to play against players at a certain Elo rating.
- Elo estimation based on game results
- Quality of moves based on Stockfish evaluations

B. Reinforcement Learning

Reinforcement learning is also a path that could be followed to greatly increase our model accuracy and effectiveness against higher rated opponents, so this is a path to consider in future version, so they could include:

- Self-play training loop
- Monte Carlo Tree Search (MCTS) for lookahead

IX. Challenges and Considerations

- Masking illegal moves consistently
- Balancing performance and training time
- Avoiding overfitting to opening databases
- Choosing realistic sampling temperature

X. Next steps

We are currently taking some inspiration from the Maia5 model to improve our own system. While our CNN-based bot uses a 13-channel bitboard representation, Maia uses 17 input planes, including castling rights and other metadata. In future iterations, we plan to explore incorporating this additional information to potentially enhance the model's board evaluation capabilities.

Maia's architecture is composed of 6 blocks of 2 convolutional layers each, with 64 channels. While our architecture is different, we may experiment with similar modular depth to improve learning capacity. Another key difference lies in the training approach: Maia samples board states randomly (1 in 32 chance of sampling a specific move) from PGN files, likely to introduce randomness and reduce overfitting. In contrast, we currently store and train on every move of every game in a CSV format. We plan to revise our preprocessing pipeline to better balance data volume, training quality, and computational efficiency - potentially adopting ideas from Maia such as selective state sampling.

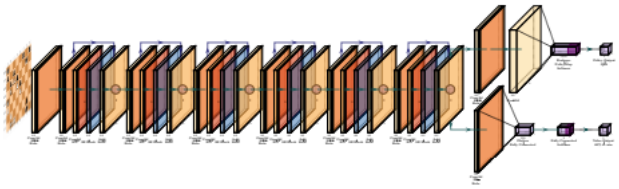

Fig. 9. Maia's architecture

Another planned improvement involves better masking of illegal moves. Right now, illegal outputs are not penalized yet, which can lead to a illegal moves being commonly predicted or inconsistent behavior. We'll improve this masking to ensure that the network learns legal move constraints more effectively.

Looking ahead, we aim to optimize the dataset itself by cleaning, augmenting, or sourcing new games where necessary (for example considering only high level games) - all while maintaining a focus on efficiency. We will also keep refining our pre-processing strategy to make it faster and more robust. Furthermore, we are considering future directions such as integrating reinforcement learning, developing our own heuristic evaluation (similar to Stockfish), or even combining both strategies like Maia does.

Finally, and most importantly, our goal is to iterate and improve the system without having to redo our base architecture.

XI. Conclusion

We aim to create a deep learning-based chess engine that not only plays well but also mimics human decision-making. Our hybrid approach allows us to combine strong theoretical understanding with modern deep learning tools. The next steps involve improving pre-processing by filtering the examples, training a baseline model, and integrating evaluation tools.

References

[1] D. Silver, J. Schrittwieser, K. Simonyan et al., "Mastering the game of chess without human knowledge," *Nature*, vol. 550, pp. 354–359, 2017.

[2] P. Behre, S. Tan, P. Varadharajan, and S. Chang, "Untitled," *International Journal on Natural Language Computing*, vol. 11, no. 5, 2022. Available: https://doi.org/10.5121/ijnlc.2022.115

[3] D. Klein, "Neural Networks for Chess," *ArXiv*, abs/2209.01506, 2022. Available: https://doi.org/10.48550/arXiv.2209.01506

[4] V. Chole, N. Gote, S. Ujwane, P. Hedaoo, and A. Umare, "Design and Development of Game Playing System in Chess using Machine Learning," 2021.

[5] D. Monroe and T. Team, "Mastering Chess with a Transformer Model," *ArXiv*, abs/2409.12272, 2024. Available: https://doi.org/10.48550/arXiv.2409.12272

[6] S. Toshniwal, S. Wiseman, K. Livescu, and K. Gimpel, "Chess as a Testbed for Language Model State Tracking," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 36, no. 10, pp. 11385–11393, 2021. Available: https://doi.org/10.1609/aaai.v36i10.21390

[7] O. Marckel, "Alpha-beta Pruning in Chess Engines," 2017.

[8] Leela Chess Zero. [Online]. Available: https://lczero.org

[9] Daniel Healey, "Building my own chess engine," [Online]. Available: https://healeycodes.com/building-my-own-chess-engine

[10] Chess Programming Wiki, [Online]. Available: https://www.chessprogramming.org/Main_Page

[11] python-chess Documentation, [Online]. Available: https://python-chess.readthedocs.io/en/latest/

[12] NanoChess, "Tiny Chess Programs," [Online]. Available: https://nanochess.org/chess3.html

[13] PGN Mentor Database, [Online]. Available: https://www.pgnmentor.com/files.html#openings
[14] J. Vermeer, "Chess Openings Visualized," [Online]. Available: https://jimmyvermeer.com/openings.html#A00
[15] `python-chess`: A chess library for Python. Available at: https://python-chess.readthedocs.io/en/latest/
[16] : Alpha Zero explained. Available at: https://nikcheerla.github.io/deeplearningschool/2018/01/01/AlphaZero-Explained/
[17] : Explanation available at: https://www.chessprogramming.org/AlphaZero