Alpha Vision +

Chess Board Computer Vision and Move Recommendations

Jim Yang (jimy@umich.edu), Mengqi Huang(mengqih@umich.edu), Josh Nankivel (nankivel@umich.edu)

Introduction

This project aims to create an end-to-end comprehensive chess analysis tool that integrates digital insights into over-the-board play in chess.

The first objective is to utilize Computer Vision to interpret 3-D images of chess boards, converting these into 2-D board states using Forsyth–Edwards Notation (FEN). Alternatively, if this is not feasible, then a proof of concept model predicting square occupancy given a photo of a chess board will be the MVP objective. At a high level, this will involve board localization to detect board corners, occupancy classification for all 64 squares of the chess board, and piece detection. In the ideal case, transfer learning will be applied to a short video of the specific board as last-shot training to provide the most accurate prediction given a subsequent photo at any point in the game.

The second objective of our project ventures into the realm of chess artificial intelligence by attempting to develop an implementation similar to AlphaZero. This involves exploring the potential of utilizing Monte Carlo Tree Search (MCTS) methodologies to iteratively enhance our algorithm's proficiency in chess. Although the objective is to have our engine find the best move in each position, the focus of this engine is not to surpass the best chess engines on the market, but rather to facilitate personal improvement and understanding in reinforcement learning. We aim to provide players with intelligent, contextually aware move recommendations that improve their strategic thinking and decision-making skills during their analysis and provide a platform to test their ideas.

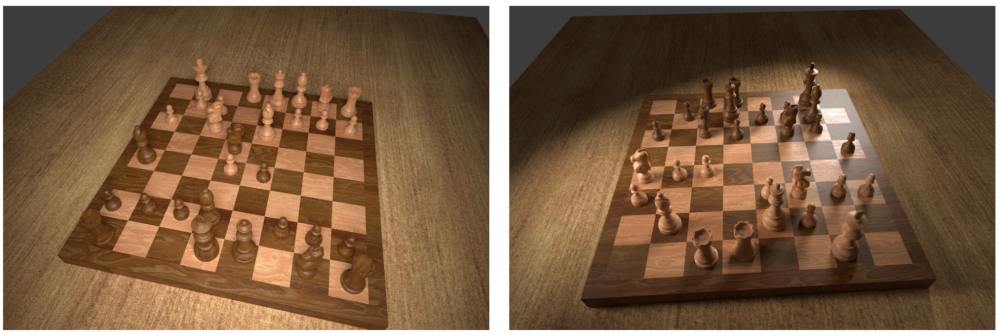
The third objective of this project is to enhance the analysis by dynamically matching current board states with an extensive database of master games. This system not only identifies similarities between positions but can also investigate the progression of games from various openings and middlegames. It may offer insights into how distinct positions develop from strategic decisions and pinpoint areas of strength and weakness for particular players. By observing the master's strategic play, actionable recommendations that transcend typical engine analysis may foster deeper strategic understanding and personal improvement for players.

The project intends to bridge the gap between offline chess and online analysis resources, providing users with gameplan inspiration through recognition of chess positions and advanced game analysis, offering a powerful resource for both enthusiasts and serious chess players.

Chess Vision

Data Sources

Wölflein and Arandjelović [5] and their chesscog [6] project generated a synthetic dataset using Blender of 3D renderings of board images and corresponding metadata using publicly available data of games by World Chess Champion Magnus Carlsen (Figure 1). Initially, the models trained on these data were used and intended to be extended via transfer learning with new real photographic datasets collected by our team (Figure 2). These were used for attempts 1 and 2. For attempt 3, we wrote code to synthesize our own 2D dataset by simulating chess games with random legal moves as input for occupancy detection.



(a) Camera flash (b) Spotlights

Figure 1. Sample of the synthetic dataset created by Wölflein and Arandjelović [5] and different lighting modes that were used to develop and test their models.

Figure 2. Self-sourced datasets, video frame extraction into separate images for last-shot transfer learning purposes in Attempts 1 and 2, and generated board images from the chess python library for Attempt 3.

Methods

Path to GitHub repository for chess vision section: https://github.com/nankivel/capstone/tree/main/chess_vision

Attempt 1 - 3D Piece Detection Transfer Learning with Chesscog

This portion of the project proved to be much more complex than originally anticipated. The chesscog[6] library was never successfully installed due to environment and dependency issues likely arising from incompatibility with the M1 Mac ARM64 architecture. Reverse engineering was attempted, and some parts of the pipeline were successfully made functional (Figure 3), but not the transfer learning portion. Rather than being able to spend much effort on training a new model or doing transfer learning, piecing together an environment and, in some cases, manually modifying the source code was necessary to inspect the pipeline and models and attempt to use portions of the functionality.

Figure 3. Failed attempt at prediction with pre-trained chesscog models

Figure 4. Failed board localization / corner detection

Attempt 2 - 3D Occupancy Detection with PyTorch

Later in the project, when it became clear the transfer learning approach in Attempt 1 was not going to work out in time, we attempted to leverage parts of Attempt 1 and focus instead of just Occupancy Detection. Classes ChessBoardDataset() and ChessBoardCNN() using PyTorch were written to facilitate the training of a Convolutional Neural Network without transfer learning. However, work continuing in parallel on Attempt 1 showed the board localization from chesscog failed (Figure 6) to generalize to our new test examples from the individual frames extracted from a video. This attempt was abandoned as a result.

Attempt 3 - 2D Occupancy Detection

We decided to finally focus on the MPV proposal, occupancy detection from a 2D board image. For this attempt, we used traditional computer vision techniques and leveraged functionality from opencv-python. We use the chess python library to simulate games with random legal moves and save the resulting board as a jpeg image after each move, saved with a filename that encodes the board state in modified FEN notation. The approach consists of the following steps:

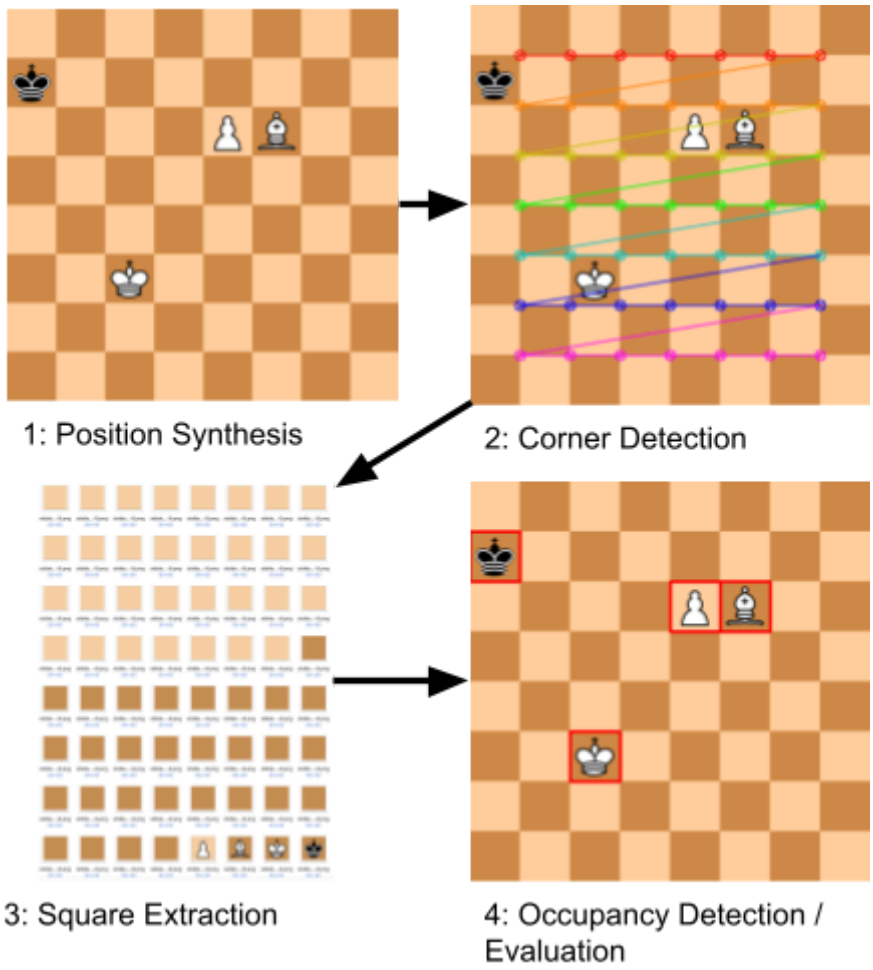


Figure 5. Process Overview

1. Simulate games with random legal moves
    a. Save with file names encoding FEN notation in **`data/generated/`**
    b. Write logs with moves from each simulated game for reference and inspection in **`data/logs`**
2. Detect inner corners in the image
    a. Save corner visualizations in **`data/corner_detection`**
3. Extract squares
    a. Extend range to cover outer rows
    b. Map extracted squares to A1:H8 positions
    c. Save extracted squares to **`data/square_extraction`**
4. Detect occupancy using the standard deviation of pixel values in each square
    a. Higher standard deviations are more likely to contain pieces
    b. Save occupancy visualizations to **`data/occupancy_detection`**
    c. Evaluate if the original FEN notation matches occupancy detection

Evaluation

While the transfer learning portion was not functional in attempt 1, the best available models from chesscog were evaluated on new images of different chess boards in various states. Each example had one or more errors in piece classification. The last example illustrates corner detection failure to generalize to chessboards with our own video-derived dataset, shifting many pieces an entire row up and causing more issues with piece classification as well. Corner detection failure also led us to abandon the Attempt 2 approach. Attempt 3 yields perfect performance for occupancy detection on the synthesized dataset, but would not generalize to other datasets.

Figure 6. Chesscog results without transfer learning on novel images not in the training dataset.

Discussion

Having never worked on a computer vision project before, our eyes were opened to the reality that this is a very difficult task, especially generalizing to unseen and unique chess boards and pieces. Others have taken to the time-consuming annotation of corners and pieces by users for each chess board as a way to avoid having to automatically detect corners. Even the project of occupancy detection on a 2D board is complex, and the 3D implications of real-world images, different lighting conditions, different chess boards and pieces, angles, occlusion of pieces by other pieces, etc. all add challenges to this problem.

The computer vision portion of this project was treated as a learning experience, mostly for computer vision and transfer learning. We completed several tutorials and examples of transfer learning to become familiar with the process, explored TorchViz to visualize the models, considered how they are connected, and what layers to unfreeze for transfer learning, etc. Much of traditional computer vision was also learned in dealing with different transformations on images, including grayscale, dilution, Gaussian blur, image masks, etc.

In the end, so much time was spent on the first 2 failed attempts that a robust set of results was not possible. Knowing what we know now, if we had started on the 3rd approach from the beginning, much more progress would have been possible, including better generalization of the approach to new unseen datasets. This works well (100% in all testing) for the specific use case of occupancy detection on output images from the chess python library, but would not generalize to other datasets.

The next steps for Attempt 3 include making it more robust to new data with different color profiles and varied pieces, likely through a more robust machine learning approach and adaptive thresholds for the masking of the images. This remains an interesting problem, especially the more ambitious formulation of the problem of trying to use 3D images on a novel chess set for transfer learning quickly in a real-world environment.

For the full scope of 3D piece detection from photographs, future work could include creating our own pre-trained models for board localization, occupancy detection, and piece classification, now that we better understand the challenges involved.

Chess Engine and Move Recommendations

Data Sources

Two sources:

1. Lichess Database [link]
    a. One hundred million chess games played on the lichess website in the month of January 2023 were downloaded. We further processed this dataset by only selecting games that contained evaluations (7%) and performing stratified sampling on games played between players of varying strengths to get a balanced dataset of 500,000 games. The board state changes after each move, and the states, along with the computer evaluation of each state, are stored as vectors.
    b. One million chess board positions were downloaded. The data is formatted in JSON, one position per line. Each position is represented by a FEN[1] string, along with a list of evaluations, ordered by the number of principal variations (PVs)[2]. 900,000 chess boards are used for training and 100,000 boards for validation. It is worth noting that the position FEN only contains pieces, active color, castling rights, and en passant square, i.e., it does not contain information about move counts.
2. Caissabase [link]
    Five million chess games, played between chess masters in official events, were downloaded. We further processed this dataset by only selecting games that featured at least one grandmaster (12%), which left us with 600,000 games. Each game contained metadata on game details, which were appropriately stored in a database, and chess moves, the arising positions of which were converted to vectors and also stored.

Outlier Removal: Most of the evaluation scores are within +/- 400; however, there are about 5% of samples with extreme high or low values. Some are more than 10x increases or decreases compared to the mean value. While these spikes are not necessarily wrong, they create issues for the subsequent normalization. To address this, we removed any observations with more than 2 standard deviations above or below the mean value (Figure 7).

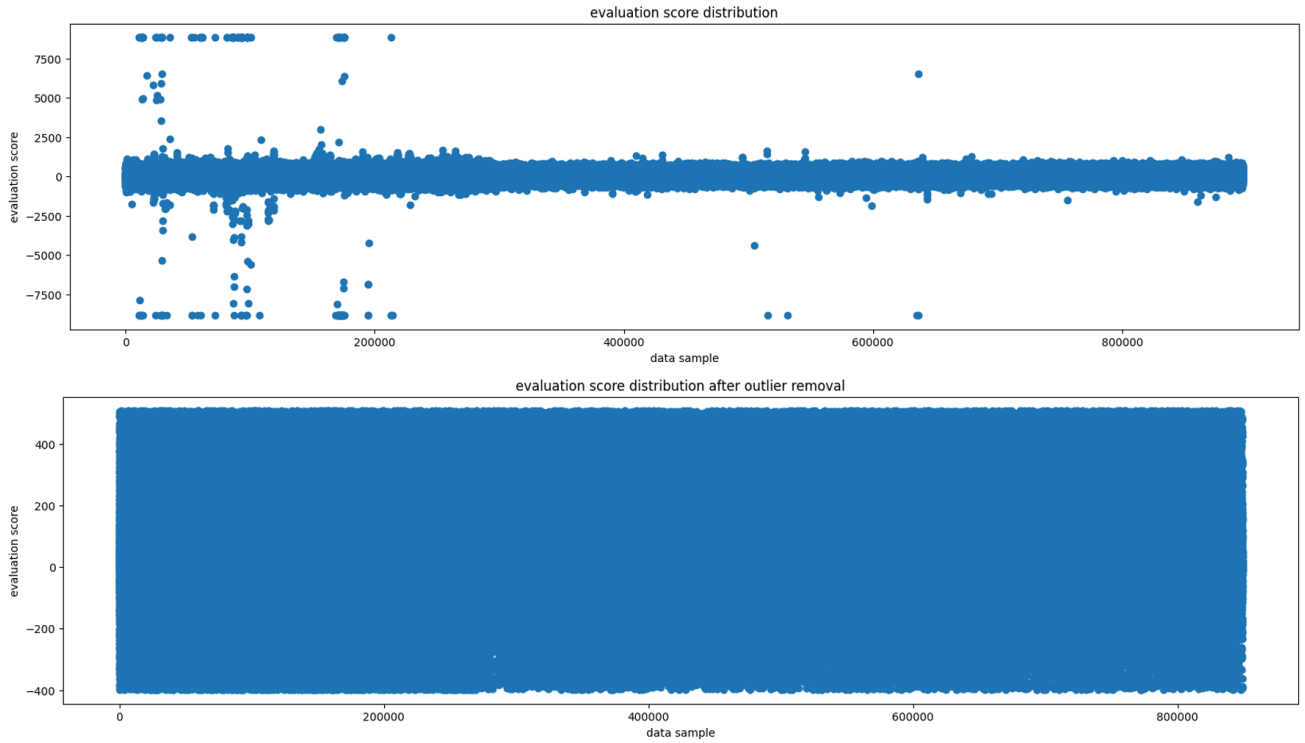


Figure 7. Raw data shows spikes in the board position evaluation scores (top), this is fixed after outlier removal (bottom)

Methods

Vectorization Techniques:

1. FEN Parsing and Board Representation: Every chess position can be described by FEN, which provides a textual representation of the game's state. This notation indicates which pieces are on which squares, along with additional game state information such as whose turn it is, castling rights, and en passant targets. Each position is translated into an 8x8 matrix where each cell contains a symbol representing either an empty square or a specific chess piece (e.g., 'Q' for Queen, 'p' for Pawn, etc.), colored either black or white. This matrix is the primary data structure from which further transformations are made.
2. Encoding into Binary Arrays: Each 8x8 matrix is then converted into a binary array. This conversion involves creating a binary representation for each type of piece and each color. For example, there would be separate binary layers for the white king, the black king, white queens, black queens, and so on for all piece types. This results in a stack of binary matrices, each of which has dimensions of 8x8, where the presence of a specific piece type and color on a square is marked as '1', and its absence as '0'.

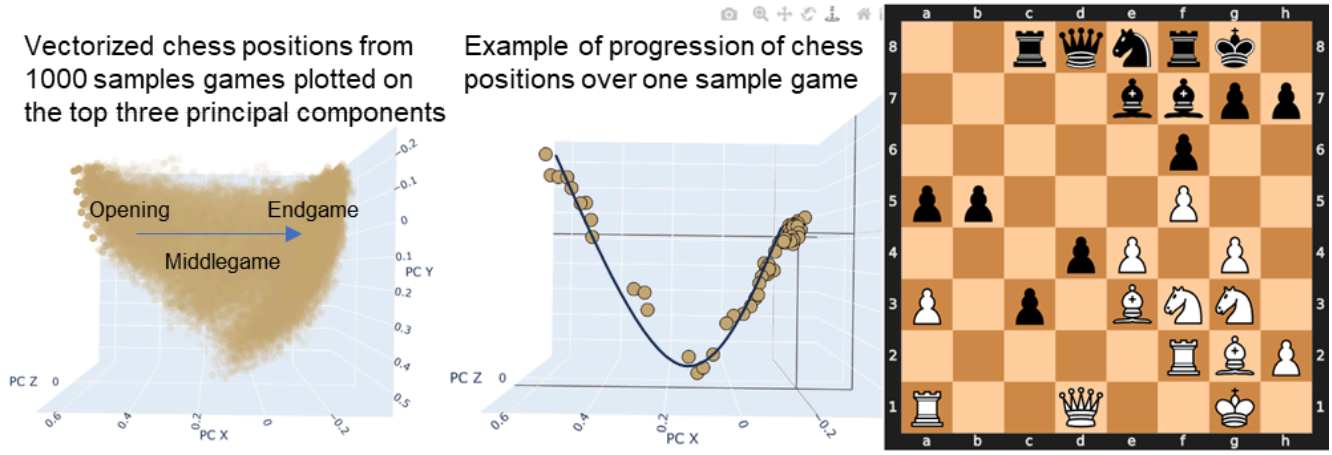


Figure 8. A visual representation of all chess positions from 1000 games; scatterplot and corresponding position of the values of the top three principle components, with game progression in reduced space for a selected position.

Semantic Vectorization:

1. Document-Term Matrix Construction: The next step is to construct a document-term matrix. In this context, each unique configuration of a chess board (each 8x8 matrix of pieces) is treated as a "document." Each square on the board that can contain any of the twelve different pieces (six types each for black and white) or be empty represents a unique "word" in our document. This matrix is high-dimensional and sparse. For each game position (document), the matrix columns correspond to the possible "words" (board squares with specific pieces), and the entries are binary, indicating whether that type of piece is present at that square in the given position.
2. Applying Latent Semantic Indexing (LSI): Using Latent Semantic Indexing (LSI), a technique borrowed from NLP/text analysis, we transform these high-dimensional data points—each position's unique configuration of pieces—into a more manageable form. This statistical method reduces the clutter of less informative details while preserving the most critical strategic themes inherent in the data. What results is a series of reduced-dimensional vectors, each encapsulating the essence of a chess position in a way that highlights its most strategically relevant features.

Dynamic Vectorization:

1. Position — Evaluation CNN Construction: The CNN is trained using values derived from evaluations by chess computers, which rate each chess position based on its likelihood of leading to a win or loss. These evaluations serve as a robust training set, guiding the CNN in recognizing patterns and configurations that historically correlate with successful outcomes. During training, the CNN employs multiple convolutional layers to process the input arrays, extracting features and learning to identify key indicators of a position's

strength. The intermediate output of this training, or what the CNN model sees, is a vector representation for each position, where the vector captures the essential characteristics as interpreted by the CNN.

2. Supervised Learning: A more complicated version of the above CNN model went into reinforcement learning. The residual convolutional network (ResNet) is our primary model architecture for supervised learning tasks in this project, inspired by the work presented in the ResNet paper[3]. ResNet introduced a key innovation through the use of residual blocks, which incorporate shortcut connections (skip connections) to mitigate the problem of vanishing gradients and facilitate the training of extremely deep networks. In our study, we also constructed networks where the residual blocks were replaced with regular convolutional blocks. This approach was taken to explore the feasibility of using a purely convolutional network architecture (PlainNet) for this specific problem domain and to compare its performance against the ResNet counterpart.

AlphaZero: Unlike supervised learning methods that require existing training data, AlphaZero[7] is a reinforcement learning approach. It uses a Monte-Carlo tree search (MCTS) algorithm to generate self-play games for training data. Each search consists of a series of simulated games of self-play that traverse a tree from root (input board state) to leaf. Each simulation proceeds by selecting, in each state s, a move with a low visit count, a high move probability, and a high value according to the current neural network. The self-play training data is then used to train the policy network with an objective function that maximizes the similarity between the network estimated policy and the actual policy distribution from the self-plays and minimizes the MSL between the estimated game value and the actual game outcome. Please refer to the appendix for more information. This was our main focus for the chess engine; however, we faced efficiency challenges during MCTS. We used 800 simulations for each MCTS step, according to the AlphaZero paper. This process on a CPU took, on average, 1.5 hours per game. With six games being played at the same time, it took a day to generate only about 100 games. It wasn't realistic to generate enough samples (AlphaZero was trained on 44 million games) that are suitable for deep network training. We took a shortcut to generate just 300 games per iteration. Due to insufficient data, the trained networks are not improving, despite the loss function continuing to decrease (Figure 9). In addition, at early iterations, most self-play games ended with a draw (value = 0), which didn't provide enough dynamic range for the target value. Due to these challenges, we unfortunately couldn't get a solid result. However, we gained precious learning experience through the interaction with this algorithm. This work refers to the implementation of the GitHub repository:
repository:https://github.com/geochri/AlphaZero_Chess

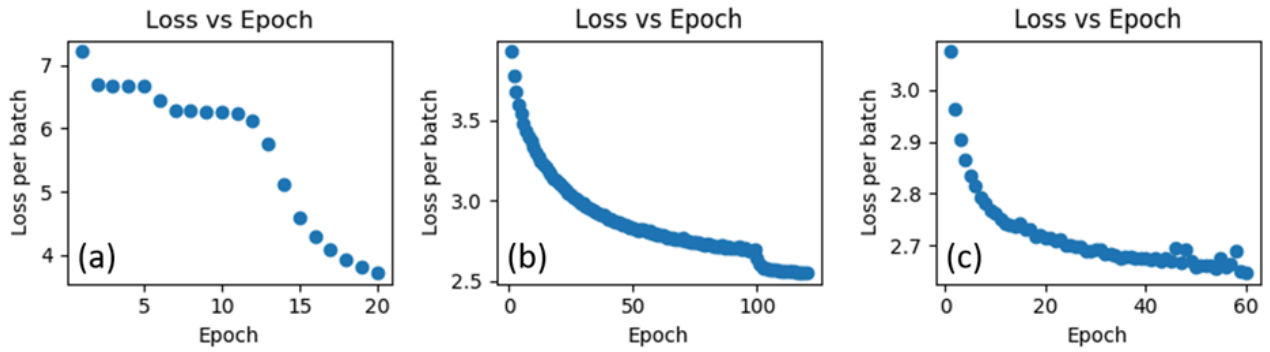


Figure 9. Even on 300 data samples for iteration, the policy network was still improving in terms of the decreasing loss values. (a) 1st iteration; (b) 2nd iteration; (c) 3rd iteration.

Recommendation System:

1. Database Management and Querying: Storing and managing the vast amount of chess data efficiently involves using PostgreSQL. The database is optimized for quick retrieval of vectorized positions, enabling real-time recommendations during gameplay. Queries are specifically designed to find the nearest neighbors in vector space for any given position, using cosine similarity measures.
2. Recommendation Engine: When a user submits a chess position query, the system first converts this position into its vector form using the same methods as the database entries. The query then performs a quick cosine similarity calculation (Figure 10) between the query vector and the vectors in the database.

The system sorts the results of the similarity calculations to identify the top matches—those positions most similar to the user's query. It then retrieves the full data for these top matches, including the FEN and associated game metadata, which are presented to the user as recommendations.

3. **Feedback Mechanism:** User feedback is captured to refine and personalize the recommendation process. Users can provide input on the usefulness of the recommendations they receive, which is intended to influence the system's future responses. This feedback could be managed by a scale system, which adjusts the weighting of semantic and dynamic contributions to the final recommendation score based on user feedback. This mechanism would allow the system to adapt over time, learning from user interactions to enhance the accuracy and relevance of its predictions.

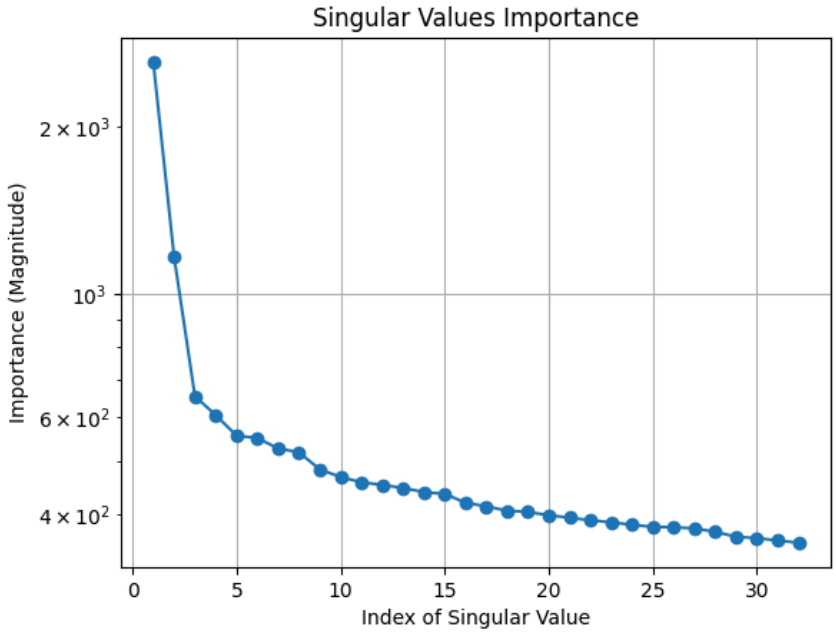


Figure 10. Computing the cosine similarity between two semantic vectors is efficient since, per the elbow method, we only need to calculate the inner product of the first 3 terms (or 5 or 8 depending on specific cases).

Evaluation

He et al.[3] demonstrated in their widely recognized work on deep learning architecture that ResNet effectively addresses the vanishing gradient problem commonly encountered in traditional deep networks, enabling networks to achieve superior performance even with increased depth. To assess this capability within the context of our chess engine problem, we implemented two ResNet models: ResNet7, consisting of seven residual blocks, and ResNet13, containing thirteen residual blocks. These models serve to investigate the impact of depth on performance and suitability for our specific application. The initial learning rate was set to 0.01 and was dynamically adjusted during training using the ReduceLROnPlateau scheduler, which reduced the learning rate by a factor of 0.2 whenever the training plateaued, with a patience of 5 epochs. Additionally, an early stopping mechanism was employed, requiring the validation loss to decrease by at least 5% per epoch compared to the average of the last five epochs. This training setup aimed to optimize model performance and prevent overfitting. The loss function is the mean squared error (MSE) between the predicted evaluation scores and the target scores.

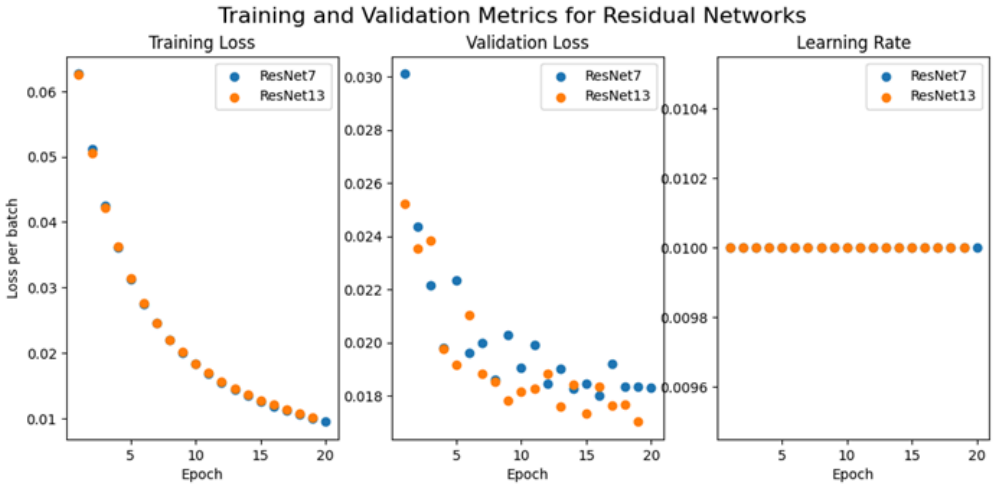


Figure 11. The training and validation loss per epoch during the ResNet training. ResNet13 shows smaller validation losses, suggesting that it generalizes better

Figure 11 shows the training process of both ResNet7 and ResNet13, demonstrating stable and consistent decreases in training loss over epochs. ResNet13 achieved approximately 10% lower validation loss compared to ResNet7, indicating better generalization performance. The training processes were terminated by the early stopping mechanism. While the ResNet13 model did not exhibit significantly better training performance compared to ResNet7, its validation results confirm that increasing the depth of the residual network does not degrade performance. It indicates that the additional layers and complexity did not introduce overfitting or compromise the model's ability to generalize. This finding supports the research finding that residual networks can effectively leverage increased depth to capture more complex patterns without sacrificing performance.

In contrast to the ResNet models, we also explored training models using plain convolutional blocks as a comparison. Initially, we attempted to train a model with seven convolutional blocks (PlainNet7), analogous to ResNet7, but encountered poor training performance. Ultimately, we found that a model with just one convolutional block (PlainNet1) could be trained; however, its performance was suboptimal. Figure 12 illustrates the training process of PlainNet1, showing a decreasing training loss over epochs but struggling with generalization, as indicated by oscillating and increasing validation loss in later epochs despite a reduced learning rate. This observation suggests that a purely convolutional architecture may not be suitable for the chess problem. We will dive deeper into this observation and its implications in the discussion section.

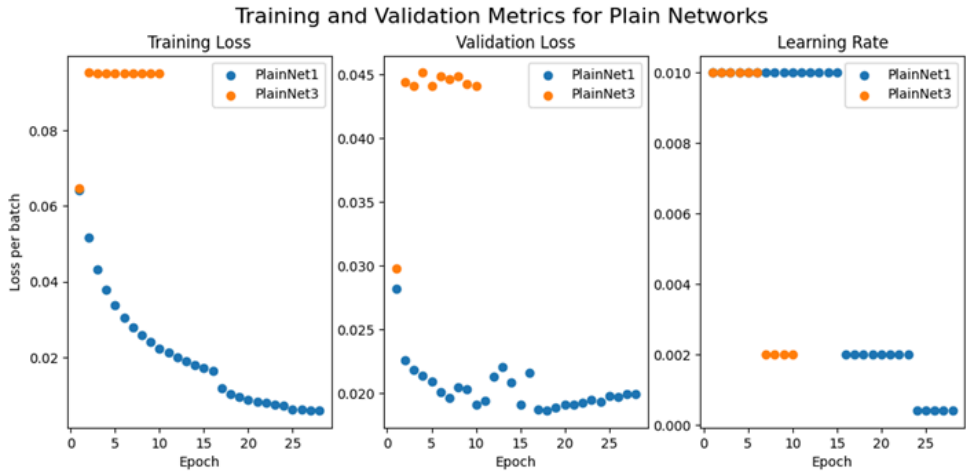


Figure 12. PlainNet3 shows increased and plateaued training and validation losses after the first epoch. Pure convolutional architecture is not suitable for representing chess board features. Skip connections in ResNet are critical.

Discussion

First, we emphasize the significance of exploratory data analysis, particularly in the context of outlier detection and removal. In this study, we observed the critical impact of outlier removal on model performance. Figure 13 illustrates the training processes of two ResNet7 models: one trained on raw, unprocessed data and another trained on data after outlier removal. The model trained on raw data, despite achieving a low loss, failed to effectively improve performance, and terminated training prematurely. This behavior can be attributed to outliers in the dataset, which caused most data samples' evaluation scores to shrink to near zero after normalization. Consequently, although the MSE loss appeared small, the differentiation space was severely constrained, rendering the training ineffective in capturing meaningful patterns.

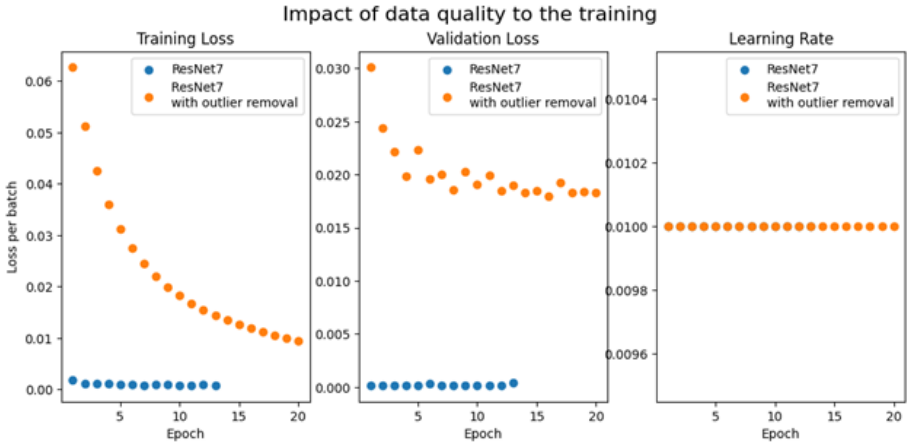


Figure 13. The conditioning of input data is critical. With outlier removal, the training and validation performance became much better. (constant learning rate for both ResNets)

Second, returning to our observations regarding PlainNets. Unlike typical image data that contains distinct visual features like edges, textures, and shapes, chess board configurations lack such explicit visual cues. For instance, the presence of a queen next to a pawn may not significantly differ from a rook next to a pawn in terms of chess gameplay, but convolutional layers could inadvertently attempt to capture visual features that don't directly translate to meaningful chess strategy. This confusion could worsen with deeper architectures, as more layers may generate increasingly complex but ultimately unhelpful features. In contrast, residual networks maintain the original features through skip connections, preventing the model from becoming easily confused or overwhelmed by irrelevant features. While additional layers in residual networks may not necessarily learn more useful information, they typically do not degrade performance.

Finally, we conducted performance evaluations by letting the trained models—ResNet7, ResNet13, and PlainNet1—compete against each other in chess games. Each pair of models played 30 games, with 15 games where one model played as the white player and the other as the black player. The first 10 moves of each game were played randomly to introduce variability and prevent models from falling into fixed patterns. Table 1 shows the game-play statistics. We can see ResNet13 shows superior performance to both ResNet7 and PlainNet1. ResNet7 and PlainNet1, however, are of similar power. Unfortunately, all three models are not good enough to compete with stockfish, even by limiting their power to only analyze up to a depth of 3. 50 games were played between ResNet13 and Stockfish, with 25 games where one model played as the white player and the other as the black player. Apart from evaluation accuracy, we believe the sophistication of the Minimax algorithm also plays a critical role. The minimax algorithm used by our trained models, although it has alpha-beta pruning, which improved the efficiency of the search, does not have domain-specific enhancements that stockfish implemented, such as move ordering, principal variation search (PVS), etc.

		Black Player		
	White Win – Draw – Black Win	ResNet7	ResNet13	PlainNet1
White Player	ResNet7	x	0-12-3	1-14-0
	ResNet13	4-11-0	x	4-11-0
	PlainNet1	1-13-1	1-9-5	x

Table 1. The arena game play results between the three models. ResNet13 is winning by a margin.

First, we emphasize the significance of exploratory data analysis, particularly in the context of outlier detection and removal. In this study, we observed the critical impact of outlier removal on model performance. Figure 13 illustrates the training processes of two ResNet7 models: one trained on raw, unprocessed data and another trained on data after outlier removal. The model trained on raw data, despite achieving a low loss, failed to effectively improve performance, and terminated training prematurely. This behavior can be attributed to outliers in the dataset, which caused most data samples' evaluation scores to shrink to near zero after normalization. Consequently, although the MSE loss appeared small, the differentiation space was severely constrained, causing the training to be ineffective in capturing meaningful patterns.

Figure 14 below shows two examples of a query position (top left) and three returned searches in two separate grids. The first query position (Sicilian Najdorf) was reached in the early middlegame, while the second occurred in the endgame. Three of the positions that were most similar to the first query were Werle - Burmakin (2015), Grivas - Sun (2018), and Ibrayev - Agmanov (2018), where a clear semblance can be seen between the positions, both dynamically and semantically. However, as the game progresses to the endgame phase, the exact pieces and their square placement become very concrete, yet the model doesn't seem to fully capture the transition of positional similarity from a more "dynamic" and active nature to a more "semantic" and rigid definition.

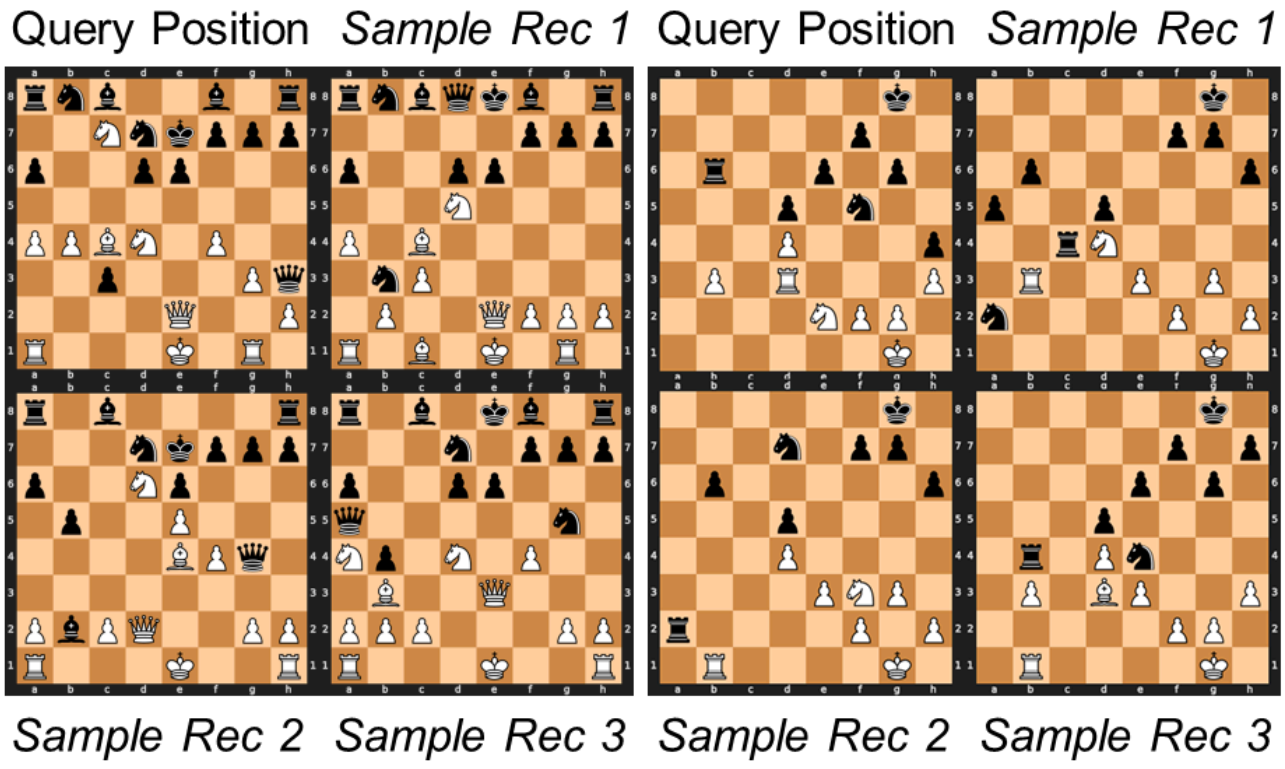


Figure 14. Query position and search results for positions with different levels of complexity.

References

[1] Edwards, S. J. (1994). Standard: Portable Game Notation Specification and Implementation Guide. https://ia802908.us.archive.org/26/items/pgn-standard-1994-03-12/PGN_standard_1994-03-12.txt

[2] https://www.chessprogramming.org/Principal_Variation

[3] He, K., Zhang, X., Ren, S., & Sun, J. (2015). Deep residual learning for image recognition. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 770-778*.

[4] Silver, D., et. al., (2018). Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. *Nature, 550(7676), 354-359*.

[5] Wölflein, G.; Arandjelović, O. Determining Chess Game State from an Image. J. Imaging 2021, 7, 94. https://doi.org/10.3390/jimaging7060094

[6] Wolflein, Georg. *Chesscog*. GitHub, latest commit June 3, 2023. Accessed 13 Apr. 2024. https://github.com/georg-wolflein/chesscog.

[7] Silver, D., et. al., (2017). Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. arXiv preprint arXiv:1712.01815.

[8] Yang, Z., Ying, L., Introduction to Reinforcement Learning, page 83-84

https://sites.google.com/view/intro-to-rl/home

Statement of Work

Jim Yang (Chess Consultant)	Mengqi Huang (Project Coordinator)	Josh Nankivel (Project Manager)
Chess Analytics Data Sourcing, Data manipulation, Board Vectorization, Master-Games Database Management, Model Building (2 agents), Data-Intensive Training, Report Writing, Version Control.	**Chess Analytics** Data Sourcing, Data manipulation, Board Vectorization, MCTS, Chess CNN Development and Training, Model Comparison, Stockfish Integration, Report Writing, Version Control.	**Computer Vision** Data Sourcing, Data Manipulation, Data Preprocessing, 3D Piece Detection, 3D and 2D Occupation Detection, Chesscog Communications, Report Writing, Version Control.

Appendixes

Future Work

Enhanced Model Validation through Advanced Clustering:

As part of future development, we aim to refine our models through advanced clustering techniques. By clustering similar positions and examining the consistency and relevance within these clusters, we can iteratively fine-tune the models to better understand what delineates similar from dissimilar chess positions. This method will not only increase the models' accuracy but also shed light on the underlying strategic and tactical themes that pervade expert chess play.

Efficient Querying with Indexing and Hashing:

To manage the vast dataset of 50 million chess positions more efficiently, future iterations of our system will incorporate enhanced indexing strategies and hashing techniques. Implementing approximate nearest neighbor (ANN) methods like Locality-Sensitive Hashing (LSH) or k-d trees will expedite the querying process. These methodologies allow for rapid approximation of nearest neighbors, significantly reducing computational overhead and enabling real-time analytical capabilities during gameplay.

Iterative Improvement through Co-Training:

Another ambitious goal is the implementation of co-training or ensemble learning techniques to enhance the synergy between our semantic and dynamic analysis models. This approach would entail alternately training each model using the output from the other as a provisional ground truth. Such a reciprocal training strategy is expected to harmonize the strengths of both models, refining their collective ability to evaluate and compare positional similarities effectively.

Recommendation Engine with ML Feedback Loop:

Incorporating a fully functional feedback mechanism will be crucial for continuously improving the accuracy and relevance of our recommendations. As users interact with the system and provide feedback on the utility of suggestions, this data will directly influence future recommendations. Over time, this feedback loop will enable the system to self-optimize, learning from a growing repository of user interactions to refine its recommendation algorithms. By adapting to user preferences and historical success rates, the system will become increasingly effective at providing personalized, strategic insights with respect to the phase of the game in the provided position.

Training data size, label, and pre-conditioning:

The current supervised learning-based chess engine used only one million chess positions as the training set due to the training time constraint; however, the amount of data may not be sufficient to best train a deep network as architected in our work. There are 22 million evaluated positions in the Lichess database. Future supervised learning work can benefit from having more training data. In addition, we may want to use the evaluations directly from the database, which are normally calculated to a depth of more than 30, instead of calculating our own, which was limited to a depth of 3 due to time constraints. A deeper evaluation would more accurately represent the position and thus provide better labels for the training. The current pre-conditioning treated positions that received high evaluations as outliers and got them removed. A better way, instead, may be to clip them.

Visualizations with TorchViz (from Chess Vision Attempt 1)

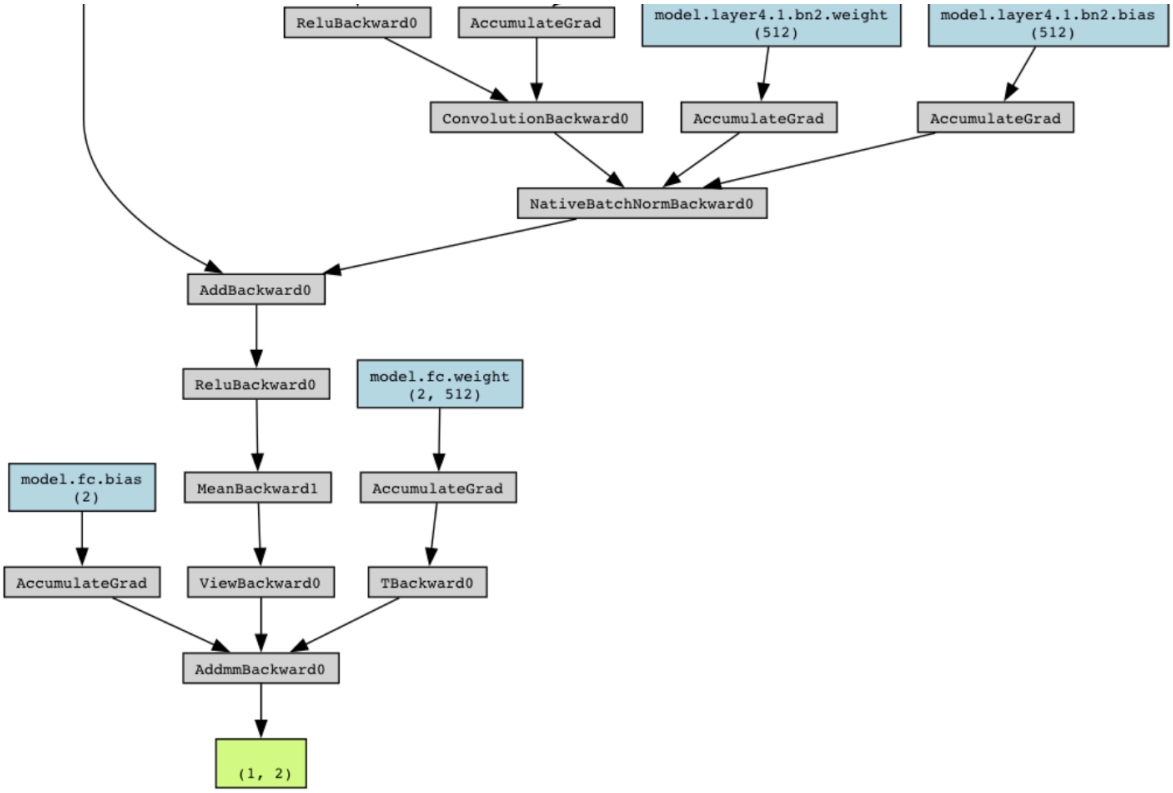


Figure A1 Section of the Resnet Occupancy Classifier visualized with torchviz

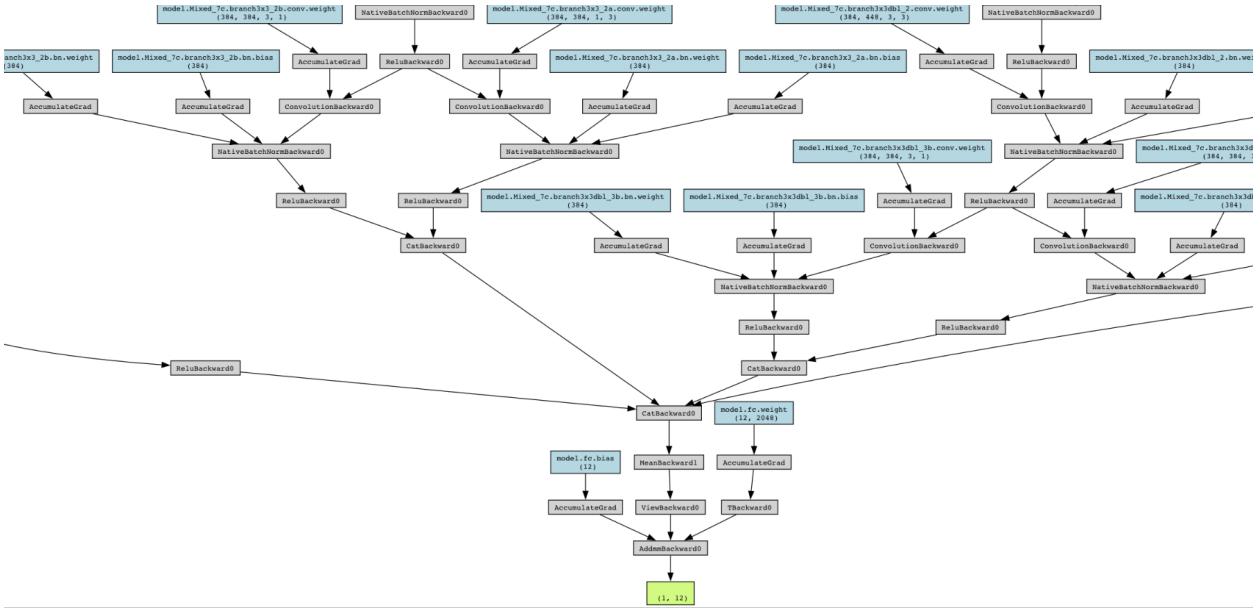


Figure A2 Section of the InceptionV3 Piece Classifier visualized with torchviz

Position Recommendation — Data Processing Details

Flipping chess boards depending on the turn:

In our chess analysis system, we incorporate a crucial step of orienting chess boards according to the player whose turn it is to move. This orientation process involves flipping the board so that the perspective is always from the side of the player who is about to make a move. The rationale behind this approach is to standardize the board's perspective across all data, ensuring that when we vectorize these positions, comparisons between them are consistent and meaningful, regardless of the original setup.

By aligning the board from the active player's viewpoint, we can more accurately capture and compare the structural and strategic similarities between different positions. This technique is especially beneficial when similar formations or strategies occur in disparate games or openings, as it allows the system to recognize these patterns without the confounding factor of orientation, thus enhancing the system's ability to make insightful recommendations based on positional similarity.

Incremental similarity computation:

In our current chess position analysis system, while we store the first 32 dimensions of each position vector derived from Latent Semantic Indexing (LSI) for comprehensive analysis potential, practical querying may selectively utilize only the top few singular values to enhance efficiency. This proposed method, inspired by principles similar to BallSearch, systematically discards a substantial fraction of the dataset that falls outside the proximity threshold set by the first few singular values.

By incrementally narrowing down the search space with each additional singular value, starting typically with the first one or two, this approach effectively isolates the most relevant positions. This tiered filtering technique not only expedites the querying process but also mirrors the strategy of BallSearch by focusing on progressively smaller "balls" or clusters of data points, optimizing both the speed and precision of database searches in large-scale datasets. The specifications of this implementation can be modified for different retrieval times and quality tradeoffs.

Move Recommendation — Data Processing Details

Data Processing: Each position can have multiple evaluations per chess position, but our supervised learning network requires a single evaluation score per position. Initially, we considered using the first evaluation score available. However, we opted instead to leverage the Stockfish API to generate a new evaluation score specifically for the white player's perspective, calculated to a depth of 3. This decision was made because evaluations from platforms like Lichess are typically conducted to a depth exceeding 20, which surpasses the efficient capabilities of our minimax algorithm given our computational constraints. The evaluation scores at different depths can vary significantly. We believe the variance in evaluation scores across different depths underscores the importance of maintaining consistency between the evaluation depth in our training data and the depth achievable during model evaluation and gameplay.

Board encoding: The input board position is represented using the Alpha Zero format[7], consisting of an 8 x 8 x 17 image stack. The first six planes encode binary features indicating the presence of white player's pieces, with each plane corresponding to a specific piece type (Pawn, Knight, Bishop, Rook, Queen, King). The subsequent six planes similarly encode the black player's pieces. The remaining 5 planes indicate the current player's color and the legality of castling (kingside or queenside) for both players. Notably, this implementation differs from Alpha Zero in several aspects: it does not include the encoding of previous board positions or information about repetitions, the total move count, or the no-progress move count. These omissions are due to the static nature of the input board state from the computer vision component of this project, which focuses solely on capturing the current board configuration without additional historical or game progression data. We acknowledge that the omission of certain game state details, such as tracking three-fold repetitions and the count of no-progress moves, may result in the model being unable to recognize specific draw conditions accurately and thus affect its performance.

Neural Network Architectures

ResNet: a single convolutional block followed by either 7 or 13 residual blocks, and then an output block[4].

The convolutional block contains the following modules:

1) A convolution of 256 filters of kernel size 3x3 with stride 1
2) Batch normalization
3) A ReLu nonlinearity

Each residual block contains the following modules:

1) A convolution of 256 filters of kernel size 3x3 with stride 1
2) Batch normalization
3) A ReLu nonlinearity
4) A convolution of 256 filters of kernel size 3x3 with stride 1
5) Batch normalization
6) A skip connection that adds the input to the block
7) A ReLu nonlinearity

The output block contains the following modules:

1) A convolution of 1 filter of kernel size 1x1 with stride 1
2) Batch normalization
3) A ReLu nonlinearity
4) A fully connected linear layer to a hidden layer of size 256
5) A rectifier nonlinearity
6) A fully connected linear layer to a scalar
7) A tanh nonlinearity outputting a scalar in the range [–1, 1]

The PlainNet shares the same building components as the ResNet, except the residual blocks are replaced by convolutional blocks that each contains the following modules:

1) A convolution of 256 filters of kernel size 3x3 with stride 1
2) Batch normalization
3) A ReLu nonlinearity
4) A convolution of 256 filters of kernel size 3x3 with stride 1
5) Batch normalization
6) A ReLu nonlinearity

AlphaZero

Board encoding[7]: The representation of the chess board is similar to the encoding used in our supervised learning tasks, with a key enhancement to accommodate a history of T=8 steps. This extended history enables the encoding of repetitive positions and includes essential features such as total move count and no-progress move count, crucial for determining draw conditions. Specifically, the first 6 planes represent binary features indicating the presence of white player's pieces, with each plane corresponding to a specific piece type (Pawn, Knight, Bishop, Rook, Queen, King). The subsequent 6 planes serve a similar purpose for the black player's pieces. Additionally, there are 2 planes dedicated to encoding repetition moves for both white and black players across the 8-step history. The remaining 7 planes encode the current player's color, legality of castling (kingside or queenside) for both players, total move count, and no-progress move count. In total, it is a 119-channel 8x8 image stack.

Action encoding[7]: The action policy is represented by an 8x8x73 stack of planes, designed to model a distribution of 4672 potential moves. Each 8x8 plane within this stack uses a one-hot encoding scheme to specify the square from which a piece will be moved.

- The first 56 planes are dedicated to encoding "queen moves," representing the possible movement distances (in the range of 1 to 7 squares) along eight different directions (N, NE, E, SE, S, SW, W, NW).
- The subsequent 8 planes represent potential knight moves, which involve fixed movement distances but across the same eight directional options.
- The final 9 planes encode possible underpromotions, excluding promotions to queens, which are represented within the "queen move" encoding.

Policy Network: In comparison to the ResNet architecture that was implemented in our supervised learning tasks, the AlphaZero network shares the same architecture as the ResNet that was implemented in our supervised learning tasks, with a difference that it has another policy head in the out block, which contains the following components:

1) A convolution of 256 filters of kernel size 1x1 with stride 1
2) Batch normalization
3) A fully connected linear layer to a vector of size 4672 (8x8x73) corresponding to the move space
4) Softmax to convert the move-scalar vector to a probability distribution

Let (s_(h,k),π_(h,k) (a),z_k) denote a sample from step h in kth MCTS self-play[8], where

- s_(h,k) is the board state
- π_(h,k) (a) is the action policy that is the probability of taking action a at state s_(h,k)
- z_k∈{-1,0,1} is the final outcome of the kth play. -1 means a loss, 0 means a draw, and 1 means a win

The loss function is as follows, which approaches the reinforcement learning problem with a supervised learning solution, aiming to maximize the similarity between the estimated policy p_θ and the sample policy π_(h,k) while minimizing the MSL between the estimated value v_θ (s_(h,k)) and the actual game outcome z_k:

$$L(\theta) = \sum_{b} \left(v_\theta\left(s_{h,k}\right) - z_k\right)^2 - \pi_{h,k} \cdot \log p_\theta\left(s_{h,k}\right) + c\|\theta\|^2$$

Monte-Carlo Tree Search: Monte Carlo Tree Search is a heuristic search algorithm that combines tree-based exploration with random sampling (Monte Carlo simulation) to efficiently explore and evaluate potential moves. In AlphaZero, this algorithm is used to generate training samples through self-plays, which are games in which the network plays against itself. For each step in a self-play game, for example, step h in game k, given the current board state s_(h,k) and the policy network θ, a number of simulations are carried out. These simulations form a new distribution of child node visits, within the legal move space, thus a new random policy π_(h,k). A training sample (s_(h,k),π_(h,k),?) is then generated, where "?" is the game result that is determined after the game ends.