Universiteit Leiden

Master Computer Science

Graph Neural Networks for Modelling Chess

Name:	Saleh Alwer
Student ID:	3305139
Date:	July 25, 2023
Specialisation:	Artificial Intelligence
1st supervisor:	Aske Plaat
2nd supervisor:	Walter Kosters

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

This research investigates the application of Graph Neural Networks (GNNs) in chess modeling, comparing their performance to traditional Residual Networks with array-based representations. We devised a graph-based representation of a chess board conducive to deep learning, harnessing nodes as squares and edges as moves between them, allowing a GNN to learn policy directly over legal actions. Through hyperparameter optimization, we determined an effective GNN architecture for chess analysis and modeling. In a comparative study involving the GNN and ResNet models, the GNN model demonstrated superior performance in supervised learning tasks in chess. This performance can be attributed to our graph-based approach which explicitly encodes legal moves. In contrast to conventional array-based deep learning methods, which map the board to the entire action space in chess, our GNN model fosters a more targeted approach, mapping contextual move representations directly to their values, thereby optimizing the utilization of learned patterns. The results indicate that GNNs, through their intrinsic ability to capture relational information between chess pieces, can provide a more effective mechanism for modeling the dynamics of a chess game. The fine-tuning of the model on specific players' games, despite being based on small datasets, demonstrated relatively good performance, signifying the versatility and wider applicability of GNNs for chess prediction tasks.

Contents

1 Introduction **1**
- 1.1 Problem Statement . . . 1
- 1.2 Reading Guide . . . 2

2 Related Work **2**
- 2.1 AI in Chess . . . 2
  - 2.1.1 Early AI in Chess . . . 2
  - 2.1.2 Modern AI in Chess . . . 3
  - 2.1.3 Significance and Future Directions . . . 3
- 2.2 Chess Board Representation . . . 4
  - 2.2.1 Array-based Representations . . . 4
  - 2.2.2 Graph-based Representations . . . 5
- 2.3 Graph Neural Networks . . . 6
- 2.4 Graph Attention Networks . . . 7

3 Data **8**
- 3.1 Grandmaster Games . . . 8
- 3.2 Random Positions . . . 8
- 3.3 Player Specific Data . . . 9

4 Methods **9**
- 4.1 Array-Based Baseline . . . 9
  - 4.1.1 Board . . . 10
  - 4.1.2 Policy . . . 10
  - 4.1.3 Model . . . 11
- 4.2 Graph Representation . . . 11
- 4.3 GNN Model . . . 12
  - 4.3.1 Architecture . . . 12
  - 4.3.2 Hyperparameters . . . 13
- 4.4 Training Loop . . . 14

5 Experiments & Results **15**
- 5.1 Setup . . . 15
- 5.2 Evaluation Metrics . . . 15
- 5.3 Hyper-parameter Optimisation (HPO) . . . 16
- 5.4 Modelling Stockfish . . . 18
- 5.5 Transfer Learning to Specific Human Players . . . 20

6 Discussion **21**

7 Conclusion **22**
- 7.1 Future Work . . . 22

References **24**

1 Introduction

The game of chess has long held a prominent place in the field of Computer Science. Its well-defined rules, perfect information structure, and vast decision tree make it an ideal testbed for the development and evaluation of artificial intelligence (AI) algorithms. Historically, the creation of chess-playing AI involved strategies such as brute force search methods and rule-based systems, designed to emulate the strategic thinking of human players.

A significant paradigm shift occurred with the advent of DeepMind's AlphaZero, which combined Monte Carlo Tree Search (MCTS) with a neural network in a reinforcement learning environment. This innovative approach has achieved unprecedented performance.

Despite these impressive advancements, there remains room for exploration and innovation. The neural network in AlphaZero represents the chess board as a multi-dimensional array akin to an image. This research proposes a novel approach to representing a chess board. Here, the chess board is seen as a graph structure where squares are nodes and potential moves form edges between these nodes. An example of this is seen in Figure 1. This perspective resonates with the intrinsic structure of chess, where pieces interact based on their positions and potential moves, thus forming a complex network of relationships naturally modelled as a graph. A primary advantage of this graph-based representation is that it inherently encodes possible moves as edges in the graph, allowing the GNN to directly infer a policy over the available actions. This stands in contrast to array-based representations, which require the architecture to explicitly encode the entire chess action space.

In this thesis, we explore the performance of Graph Neural Networks (GNNs) in capturing important aspects of the game such as value (the assessment of the current game state) and policy (the selection of the best move). GNNs are well-suited to process such graph-structured data, and we hypothesize that they could offer unique advantages in modelling chess. Unlike array-based networks, GNNs, which are designed to capture the relational information between nodes (or squares on the chess board), may provide a more nuanced understanding of the game state.

1.1 Problem Statement

This research seeks to investigate the potential of GNNs in modeling chess, a domain traditionally dominated by array-based neural networks. The primary question driving this research is:

> How effectively can GNNs, as compared to traditional methods like Residual Networks (ResNets) with array-based representations, capture both the value and policy of a given chess position?

In order to address this overarching question, three specific research questions are proposed:

1. What are the crucial elements of a chess board state that need to be incorporated into the graph representation, and what are the optimal methodologies for incorporating these elements effectively?

2. How does the performance of GNNs in modeling chess positions and playing styles compare to that of ResNets with array-based representations?

3. What is the most effective GNN architecture for chess analysis and modeling?

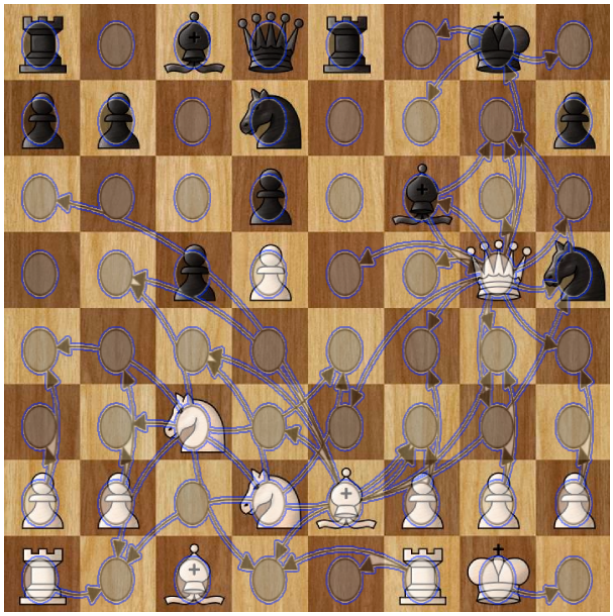


Figure 1: Graph representation of a chess board.

1.2 Reading Guide

We first go over the related work in Section 2. This includes previous research on neural networks in chess and an introduction to GNNs. The data used is desribed in Section 3. Section 4 includes the specifics for our implementation and justifications for all decisions made in relation to our model and the training process. Section 5 details experiments, evaluation metrics and the respective results. Section 6 discuss the results of the thesis and its limitations. Section 7 concludes this paper.

2 Related Work

This section reviews the existing literature and foundational concepts that underpin this research, covering the application of AI to chess, different chess board representations, and a brief introduction to GNNs.

2.1 AI in Chess

The application of AI to chess is a field with a rich history dating back to the mid-20th century. The emergence of computer chess, as an academic field, has its roots in the work of early computer scientists, such as Alan Turing and Claude Shannon, who were fascinated by the strategic complexity of the game and the potential for automating such a human-like cognitive process (Giannini and Bowen, 2017).

2.1.1 Early AI in Chess

One of the earliest strategies for AI in chess is the Minimax algorithm, proposed by John von Neumann. The idea behind the Minimax algorithm is quite straightforward: the algorithm makes the move that maximizes its minimum gain, as determined by a static evaluation

function written by chess experts. In other words, it assumes that the opponent will always make the move that is worst for it, and thus, it makes the move that will be best for it under this worst-case scenario (Kjeldsen, 2001).

However, the Minimax algorithm's brute-force approach to considering every possible move sequence becomes computationally infeasible as the game progresses. To mitigate this, the Alpha-Beta pruning algorithm was introduced as an optimization to the Minimax algorithm. Alpha-Beta pruning works by skipping branches of the game tree that do not need to be considered because there already exists a better move available. It maintains two values, alpha and beta, which represent the minimum score that the maximizing player is assured of and the maximum score that the minimizing player is assured of, respectively. Alpha-Beta pruning effectively reduces the complexity of the search tree and allows the alrogithm to search deeper in the same amount of time (Edwards and Hart, 1961).

The evolution of these early AI strategies led to remarkable milestones in computer chess. IBM's Deep Blue, utilizing a form of Minimax algorithm with Alpha-Beta pruning and advanced evaluation functions, struck a significant blow in 1997 by defeating reigning world champion Garry Kasparov (Campbell et al., 2002). Following this lineage, the open-source engine Stockfish stands as a testament to the continued relevance of Alpha-Beta pruning. Moreover, traditional chess engines also utilize an opening book and endgame tablebases to further enhance their performance. These tools provide pre-calculated sequences of moves and positions for the beginning and end phases of the game.

2.1.2 Modern AI in Chess

The modern era of computer chess was heralded by the introduction of machine learning-based models, most notably AlphaZero by DeepMind. Unlike its predecessors, AlphaZero utilized a combination of deep neural networks and MCTS to make its decisions. Through reinforcement learning, it learned to play chess from scratch by playing against itself and improving over time. The NN is trained to approximate the MCTS which searches the chess space using the NN. AlphaZero represents a significant departure from the traditional, heuristic-based approaches, with its capability to generate human-like strategies and tactics, while also often diverging from established chess theory (Silver et al., 2018).

Following the success of AlphaZero, several other models like LeelaChess and StockfishNNUE were developed (Pascutto and Linscott, 2020; Chess Programming Wiki contributors, 2020). Leela Chess Zero also employs MCTS for decision-making. However, an interesting fusion of traditional and contemporary approaches is seen in StockfishNNUE. This engine maintains the Minimax algorithm with Alpha-Beta pruning but replaces the heuristic-guided evaluation function with a neural network trained on Stockfish evaluations at a high depth.

2.1.3 Significance and Future Directions

The aforementioned advancements in computer chess have not only resulted in AI models that can consistently defeat human world champions, but they have also contributed significantly to our understanding of chess as a game. The analysis of games played by these AI models has often revealed new strategies and insights, influencing the way chess is played at the highest levels.

The AI models for chess are largely based on traditional board representations and tree-search algorithms. The application of novel deep learning techniques like GNNs to chess is an unexplored area. This forms the crux of our research.

2.2 Chess Board Representation

In this section, we delve into array-based and graph-based methods approaches for representing a chess board.

2.2.1 Array-based Representations

The representation of a chess board in a computer program is a critical aspect of chess AI. Traditional deep learning approaches generally represent the chess board as a matrix that can be processed by convolutional layers in neural networks.

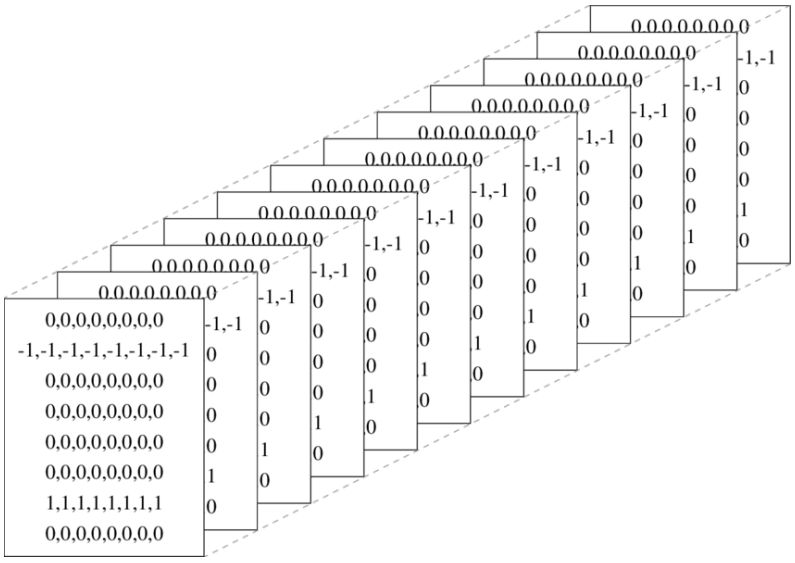


Figure 2: Visualisation of array-based representation of a chess board (Sabatelli, 2017).

AlphaZero captures the state of the chess game as a tensor of dimensions $N \times N \times (MT + L)$, where this representation caters to the intricate facets of chess (Silver et al., 2018). The first dimension, N, corresponds to the chessboard's size, which is generally 8 for a standard game, thus forming an $N \times N$ matrix. The second dimension comprises T sets of M planes, each of size $N \times N$. Each of these sets illustrates the board position at distinct time-steps, initialized to zero for steps less than one, and is oriented from the perspective of the current player. The planes encompass binary features, outlining the presence and type of the player's and opponent's pieces. The final dimension, L, introduces constant-valued input planes. These provide additional game-related information such as the player's color, the overall move count, and specific rules such as castling legality in chess. A visualization of this multidimensional chessboard representation is provided in Figure 2.

Similar array-based representations are utilized by LeelaChess (Pascutto and Linscott, 2020) and StockfishNNUE (Chess Programming Wiki contributors, 2020), albeit with some deviations from AlphaZero's format. Notably, these models do not include time-steps in their representations. This approach operates on the theory that the value and policy of a chess position are independent of the previous moves used to reach the position.

2.2.2 Graph-based Representations

Our research focuses on representing the chess board as a graph, which, while a less-traveled path, particularly for deep learning applications, holds potential for novel insights into chess dynamics.

Graph representations in chess have been explored, for instance, in knight's and rook's graphs. The knight's graph represents all legal moves of the knight chess piece on a chessboard, with each vertex of this graph corresponding to a square on the chessboard, and each edge connecting two squares that a knight can move between (Wikipedia contributors, 2023a). Similarly, the rook's graph represents all legal moves of the rook chess piece on a chessboard, with each vertex representing a square on the chessboard, and an edge connecting any two squares sharing a row or column (Wikipedia contributors, 2023b).

Another notable application of graph theory to chess game analysis can be seen in a project that used data from Lichess to analyze piece captures in over 20,000 matches (Sharp, 2020). The project employed the Neo4j graph database to map relationships between chess pieces, representing each piece as a node and each capture as an edge. Weighted degree centrality was used to measure piece importance, and the Louvain algorithm was used to identify communities of pieces that regularly capture each other.

A more comprehensive tool, ChessY (Rudolph-Lilith, 2019) offers a Mathematica toolbox for the generation, visualization, and analysis of positional chess graphs. It allows for a thorough analysis of chess games from a graph theory perspective. ChessY is built around three principal types of data objects: the chess position, nodes, and edges. The chess position object contains a list of pieces, their location on the chessboard, and supplementary information characterizing a given position. The nodes and edges are one-dimensional and two-dimensional lists, respectively, that describe the positional chess graph associated with a given position. A visualisation of this is seen in Figure 3.

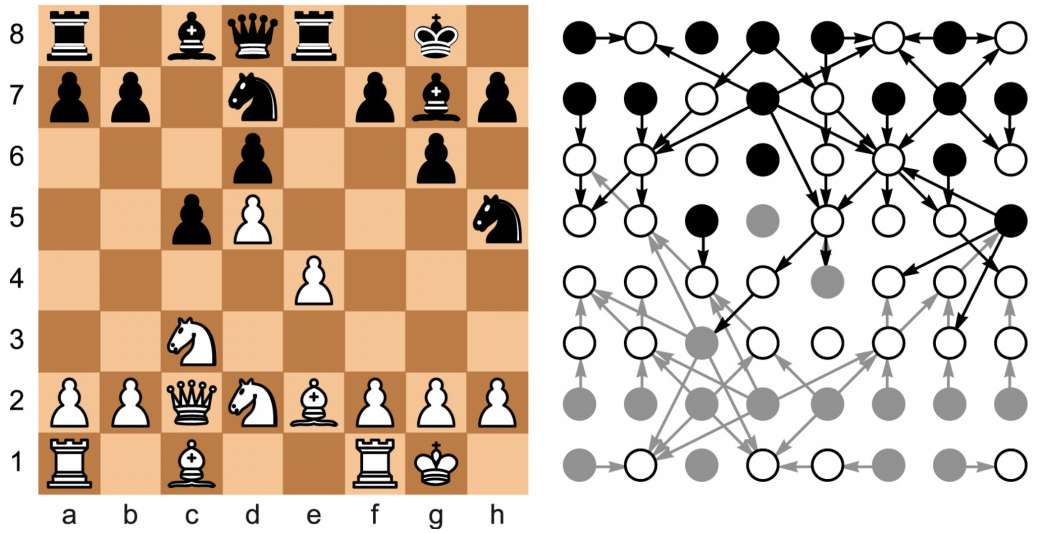


Figure 3: Visualisation of the graph representation of a chess board used by Rudolph-Lilith (2019).

ChessY was successfully utilized in a study of chess games between Grandmasters and computer players. The study aimed at identifying and characterizing strategical approaches employed in the gameplay of human players and computer chess algorithms. This analysis

found that both types of players benefited from retaining more and higher-value pieces, in conjunction with maintaining a high potential connectivity to squares on the board. However, the analysis also identified key differences between human and computer players, opening up interesting avenues for future research.

ChessY represents a significant advance in the application of graph theory to chess game analysis. Its tools for parsing game records and constructing positional chess graphs allow for a systematic and detailed exploration of chess games within a graph-theoretical context. It provides the foundation for exploring new approaches in chess game analysis and potentially developing more human-like computer chess algorithms. Its successful implementation serves as a compelling testament to the notion that valuable information about chess dynamics and strategies is indeed encoded within a graph-based representation of the game.

While the ChessY toolbox represents an important development, its scope is limited, and there remains a significant gap in the research landscape for a more expansive application of graph-based representations in chess, particularly in the area of deep learning.

2.3 Graph Neural Networks

GNNs are powerful deep learning models that cater specifically to graph-structured data. Their unique architecture allows them to harness the connectivity patterns inherent to such data, in contrast to traditional deep learning models which are primarily suited to grid-like data such as images or sequences.

In the domain of GNNs, numerous models and techniques have been proposed, all aimed at capitalizing on the structural features of graph data to improve performance. One key model in this sphere is the *Message Passing Neural Network* (MPNN) (Gilmer et al., 2017). The crux of MPNNs lies in the aggregation of messages propagated from neighboring nodes, a concept formalized in Equation 1.

$$\mathbf{x}_i' = \gamma_{\mathbf{\Theta}} \left(\mathbf{x}_i, \square_{j \in \mathcal{N}(i)} \, \phi_{\mathbf{\Theta}} \left(\mathbf{x}_i, \mathbf{x}_j, \mathbf{e}_{j,i} \right) \right) \tag{1}$$

Here, $\mathbf{x}_i$ signifies the node embedding of node i and $\mathbf{e}_{j,i}$ signifies the edge feature between node j and node i. The updated counterpart post message-passing is $\mathbf{x}_i'$. The aggregation function, symbolized by $\square$, is a permutation invariant operation (like sum, mean, or max) applied over the set of neighboring nodes of i, $\mathcal{N}_i$. The sum is often the preferred aggregation function due to its information-retention capability.

One pivotal architecture within GNNs is the *Graph Convolutional Network* (GCN) (Kipf and Welling, 2017). GCNs were introduced to execute convolution operations directly on graph data, thereby exploiting the inherent structure within the graph to enhance the model's performance. The primary operation in GCNs is depicted in Equation 2.

$$H^{(\ell+1)} = \sigma \left(\tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}} H^{(\ell)} W^{(\ell)} \right) \tag{2}$$

The notation $H^{(\ell)}$ denotes the matrix of activations (node features) at the ℓ^{th} layer of the GCN. As we move forward from one layer to the next (from ℓ to $\ell + 1$), these activations are updated according to Equation 2. The term $W^{(\ell)}$ represents the weight matrix at the ℓ^{th} layer. These weights determine the contribution of each node to the new features being calculated. The degree of a node in a graph is the number of edges connected to that node, $\tilde{D}$ is the degree matrix of the graph with added self-loops.The adjacency matrix with self-loops is represented by $\tilde{A}$. The product $\tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}}$ is a normalization of the adjacency matrix $\tilde{A}$ of the graph.

This normalized adjacency matrix essentially represents the structure of the graph and how the nodes are connected to each other. In the equation, $\tilde{D}$ is raised to the power of $-\frac{1}{2}$ on both sides of $\tilde{A}$. This means each element of the adjacency matrix is divided by the square root of the degrees of its corresponding nodes. This technique is used to avoid the scale of the output features being overly dependent on node degrees and to ensure the stability of the learning process. Lastly, the entire product is passed through the activation function σ. This summarizes how information from the ℓ^{th} layer of the GCN is transformed and propagated to the $(\ell+1)^{th}$ layer, taking into account the graph structure, the current node features, and the trainable weights. The updated node features $H^{(\ell+1)}$ can then be used in the next layer of the GCN, or as output for downstream tasks like node classification or graph regression, depending on the specific application.

GCNs do exhibit some limitations, particularly when it comes to weighting neighbor nodes. GCNs inherently assign equal weights to all neighbors during the aggregation step, which can be sub-optimal when the importance of neighbors varies. This serves as a key motivation for the emergence of *Graph Attention Networks* (GATs), which are detailed in the following sub-section.

2.4 Graph Attention Networks

In addressing the limitations associated with prior GNN models, Veličković et al. (2018) introduced GATs. The primary difference being the integration of attention mechanisms. The attention mechanism, inspired by Transformer models in Natural Language Processing, allows for adaptive weighting of neighboring nodes during the aggregation process.

In the operational mechanics of a GAT layer, each neighboring node i of a specific node forwards its attention coefficient vector $\overrightarrow{\alpha_{1i}}$. This vector carries individual attention coefficients for each attention head α^k_{1i}, analogous to the multiple filters used in traditional convolutional networks. The use of multiple attention heads enhances the model's ability to capture different types of relationships and patterns within the graph.

These attention coefficients are applied to scale the corresponding neighbor node's feature vectors $\overrightarrow{h_i}$, similar to the weights in the GCN model's convolution operation (Equation 2). These scaled features are then aggregated across the neighborhood, much like in the MPNN model (Equation 1), to compute the new feature vector of the node, $\overrightarrow{h'_1}$. A visualisation of the attention process is seen in Figure 4. This attention mechanism allows the GAT model to assign different importance to different nodes in a neighborhood, thereby capturing more nuanced structural information in the graph. Furthermore, unlike some previous models, the GAT architecture avoids costly matrix operations and can be parallelized across all nodes, offering significant computational advantages.

Building on the success of the original GAT, an enhanced version known as GATv2 (Brody et al., 2022) was proposed to address the issue of "static attention" in the original model. Unlike in the original GAT where the attention coefficients were computed before the aggregation operation, in GATv2, the order of operations is changed to allow the attention coefficients to be updated dynamically during the aggregation process. This modification enables GATv2 to capture more expressive attention dynamics, leading to superior performance across various benchmarks. Despite this added expressivity, GATv2 maintains the same parameter costs as the original GAT, making it a particularly attractive choice for complex tasks or challenging datasets in the GNN domain.

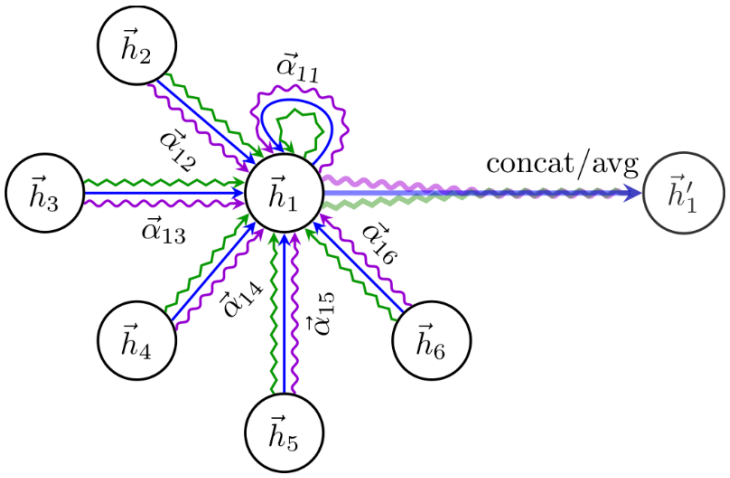


Figure 4: Visualisation of the attention mechanism used in GATs to update the node representation $\overrightarrow{h_1}$ to $\overrightarrow{h'_1}$ (Veličković et al., 2018).

3 Data

This section details the various sources and types of data utilized for this study, including games played by chess grandmasters, simulated random positions, and games played by specific individual players with varied skill levels

3.1 Grandmaster Games

The data we utilized for our experiments is a rich dataset consisting of 150,000 games played by chess grandmasters (Cuevas, 2021). The data is sourced from two established chess platforms: Chesstempo and PgnMentor. Both these sources offer a wide array of grandmaster games, providing a diverse dataset for our model to learn from.

Beyond the raw game data, we incorporated engine-derived evaluations to enrich the dataset. Specifically, we used the Stockfish chess engine to generate policy and value features for the game states at each position. Instead of relying on the moves made by the grandmasters, these features offer a more comprehensive understanding of the game state, thereby allowing a model to learn a policy modeled after one decision maker, rather than thousands. Moreover, these engine-derived evaluations serve as more consistent and objective labels for our learning task, reducing the potential biases inherent in human grandmaster moves.

At each position Stockfish is ran with a time limit of 0.01 seconds. This is around depth $= 10$ for non-endgame positions. We assign scores to the moves suggested by Stockfish using the formula $(n - m)/n$, where n is the total number of advantageous moves and m is the rank of a specific move, with all other moves assigned a score of zero. The board evaluation, originally expressed in centi-pawns, is normalized by dividing by 650 and capped at $+1$ and -1, respectively.

3.2 Random Positions

In this research, we also enrich our dataset by incorporating random positions derived from simulated chess games. In this procedure, we initiate a game and choose the next move either randomly or based on Stockfish's recommendation, with the choice governed by a certain

probability distribution. Specifically, Stockfish's moves are probabilistically selected to ensure that the generated games do not deviate too drastically from realistic game scenarios.

Furthermore, during this random walk, we save the game state at each position with a 20% probability. This method helps us capture a diverse and representative set of game positions. The randomness introduced by the process ensures a broad exploration of the game space, while the probabilistic selection of Stockfish's moves ensures that the positions are not entirely arbitrary and bear some relevance to practical gameplay.

The inclusion of these random positions in our training data allows us to achieve a more comprehensive coverage of the possible game states in chess. It helps ensure that our model is not overly specialized to the patterns present in grandmaster games and can generalize effectively across a wide array of game states. This diversification of the data is particularly crucial in the context of chess, where the number of possible game states is astronomically large. Thus, the inclusion of random positions in the dataset serves as a step towards the model's robustness and versatility in playing chess. The policy and values are retreived using Stockfish with a 0.01 second time limit, as desribed in the previous sub-section.

3.3 Player Specific Data

For player-specific learning, we use datasets derived from the games of three chess players of varying skill levels. Each dataset comprises only the positions and corresponding moves played by the specific player in each of their games, reducing the total position count by approximately half in comparison to considering all positions in a game.

The required game data was downloaded from OpeningTree (OpeningTree, 2023). The number of positions in each player's dataset is provided in Table 1. Average rated player data is from an online account of a 1500 Elo rated player's one-minute time-control online games. These are low quality moves. Anatoly Karpov and Alireza Firouzja are two of the best grandmasters in history who's data are from tournament games they have played with higher time controls. These are high quality moves.

Player	**Number of Games**	**Number of Positions**
GM Anatoly Karpov	2,104	134,624
GM Alireza Firouzja	3,313	156,508
Average rated player	8,160	325,972

Table 1: Number of positions in the player-specific datasets.

4 Methods

This section delves into the detailed methods and techniques adopted in our study, encompassing the array-based baseline approach, the graph-based representation of a chess board, the model proposed, and the training process.

4.1 Array-Based Baseline

To establish a comparative assessment of our GNN approach, we devised an array-based baseline for the task. This benchmark setup employs a three-dimensional array interpretation

of the chess board, coupled with a ResNet architecture, designed to predict game values and policy actions.

4.1.1 Board

The board's representation in our baseline method is implemented as a three-dimensional array of size $8 \times 8 \times 21$ that encapsulates the full state of a chess game. This array, offering a spatial and categorical description of the board, is structured to provide a holistic view of the game. The first 12 planes of array deals with piece and color. Each square of the chessboard is associated with a specific slice of this array, corresponding to a potential state of the square: it could be empty or occupied by any piece from either color.

In addition to the arrangement of pieces, 9 planes are padded to reflect: the color of the player to move, the castling rights for each player (4 planes), the number of full moves made, the repetition of positions, the half-move clock, and the presence of an en passant square.

This representation mirrors that of AlphaZero, albeit without the inclusion of time-steps. The decision not to include time-steps is primarily influenced by the approaches followed in prominent projects such as Leela Chess Zero and Stockfish NNUE. Both of these successful chess AIs forgo the inclusion of temporal features in their representation of the chess board, demonstrating that a strong model can be built without this additional layer of complexity. This reduction in complexity confers tangible computational advantages, leading to a decrease in processing time and resource consumption due to the smaller size of the array representation. Furthermore, the underlying rationale for excluding time-steps rests on the principle that the value and policy derived from a particular chess board configuration should be inherently independent of the sequence of moves taken to reach that position. The inclusion of time-steps in the AlphaZero architecture might have been necessary to facilitate the learning of MCTS in a reinforcement learning setting, where the knowledge of prior states can guide the exploration process. However, in our scenario, where the learning process is governed by supervised learning principles, the necessity of including time-steps diminishes. The exclusion of this temporal component, therefore, helps streamline the model's architecture without impinging upon its ability to accurately evaluate chess board positions and predict optimal moves.

4.1.2 Policy

For policy representation, we use a probabilistic approach similar to the one utilized by AlphaZero. Each move in a game of chess is described in two parts: the selection of a piece to move and the subsequent choice from among its legal moves. We capture this using an $8 \times 8 \times 73$ stack of planes, which encodes a probability distribution over the 4,672 possible moves. Each position in the 8×8 grid identifies the square from which a piece can be selected, followed by a set of planes that represent the possible moves for that piece.

The first 56 planes encode potential "queen moves" for any piece, capturing the number of squares [1..7] the piece can be moved in one of eight relative compass directions N, NE, E, SE, S, SW, W, NW. The next 8 planes cater to the unique knight moves. The final nine planes deal with the special case of underpromotions for pawn moves or captures in two possible diagonals, promoting to a knight, bishop, or rook.

4.1.3 Model

The network utilized to model the value and policy of an array representation of the chess board is essentially a deep Residual Network (He et al., 2015), based on the network used in AlphaZero. It is designed to process the input board state, run it through several layers of transformations, and produce a final output.

The input to the network is the board state represented as an array, specifically a tensor of dimensions $22 \times 8 \times 8$. This board state tensor undergoes a convolutional transformation and normalization in the initial Convolutional Block, followed by a ReLU activation. The result of this is a feature map that is passed onto subsequent layers. These are 19 residual blocks (ResBlocks). Each ResBlock applies two convolutional transformations on the input, each of which is followed by batch normalization and a ReLU activation. After the second transformation, the original input (the residual) is added back to the output feature map, promoting the network's ability to learn identity functions and mitigating the issue of vanishing gradients in deep networks.

Lastly, we have the output block, which is responsible for producing the final value and policy outputs. The value head applies a convolutional transformation followed by batch normalization, a ReLU activation, and two fully connected linear layers with tanh activation. It outputs a scalar value between -1 and 1, representing the predicted state value for the current player. The policy head applies a convolutional transformation, followed by batch normalization, ReLU activation, and a fully connected layer to generate a probability distribution over possible moves. A visualisation of this model is seen in Figure 5.

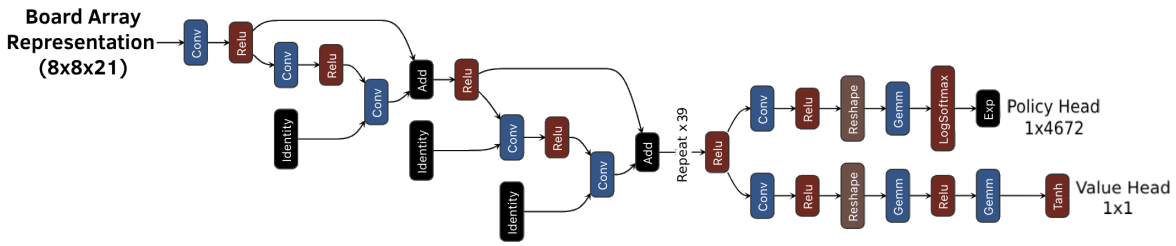


Figure 5: ResNet model with 60 million parameters which includes an initial convolutional block, 39 residual blocks, and an output block. The convolutional block contains a single convolutional layer followed by batch normalization and a ReLU activation function. Each Residual Block comprises two convolutional layers, each followed by batch normalization and a ReLU activation, with a skip connection that bypasses these operations. The output block consists of two heads, one for value estimation with a tanh activation function and another for policy prediction that uses a softmax function.

4.2 Graph Representation

The transformation of a chessboard into a graph-based representation is a two-fold process, focusing on the effective encoding of nodes and edges.

Each node corresponds to a square on the chessboard and carries a feature vector encoding information about the piece type, the color of the player to move, full move count, repetition of positions, half-move clock, and the presence of an en passant square. These node features are analogous to the array-based representation used by AlphaZero (Silver et al., 2018), which allows us to leverage their proven design.

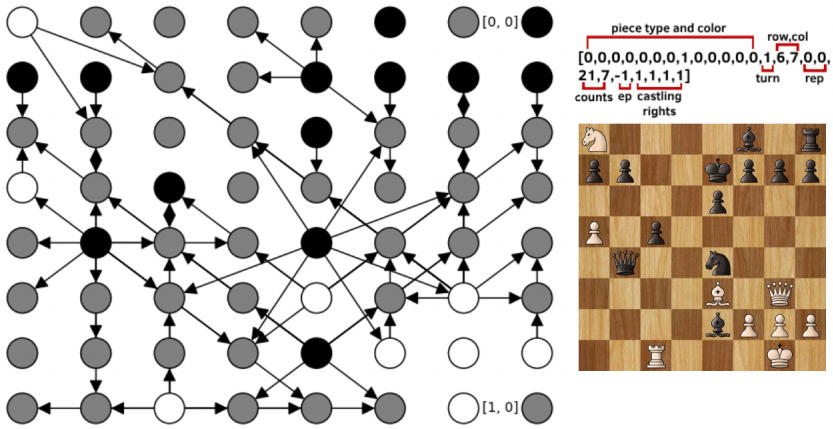


Figure 6: The graph representation of a chess board shown on the right. The vector on the top right shows a node feature vector (encoding of a black pawn, in a board with no castling rights, no en passant, white to move, move 21, 0 repetition and 7 half-move clock). The top right and bottom left show two edge feature vectors.

Specifically, each square is assigned a feature vector of size 20. If the square is unoccupied, the first entry of the vector is marked with a 1, and the rest are set to 0. For an occupied square, the vector entries corresponding to the piece type and color are flagged. The position entries carry information about the row and column of the square from the perspective of the current player. The remaining entries carry game-related information such as repetitions, move counts, and so on.

This encoding strategy aims to carry forward the insights provided by the AlphaZero design, ensuring that every node possesses a complete snapshot of its contextual information within the game. These local and global features enable the machine learning model to identify and learn important patterns and strategies from the game.

The edges of the graph encode the potential moves. Every legal move that a player can make from one square to another is represented as an edge in the graph. In addition to the current player's legal moves, we also account for the legal moves of the opponent by flipping the player's turn and generating their set of legal moves. This is in line with the design used by ChessY (Rudolph-Lilith, 2019), where the edges correspond to possible moves. This design offers an explicit illustration of the move dynamics of the chess game.

Furthermore, each edge carries a feature vector that encodes the player to whom the move belongs (black or white) and any promotions when necessary. An example of this is representation is seen in Figure 6.

4.3 GNN Model

This section outlines our proposed dual-headed GNN model that leverages GATs to effectively represent and predict chessboard states and potential moves.

4.3.1 Architecture

Our model architecture revolves around the core utilization of GATs, specifically tailored to serve our goal of predicting chess board policy and value. The model deploys two GATs, each embedding the input graph to obtain two distinct sets of embeddings for the nodes in our

graph. The first set of embeddings is used to capture the overall representation of the chess board, while the second set is dedicated to representing each individual move on the chess board.

To obtain a comprehensive representation of the chess board, we employ an attentional pooling mechanism on the first set of embeddings. Attentional pooling allows the model to weigh the importance of different regions of the chess board and dynamically adjust these weights based on the context.

The second set of embeddings, obtained from the other GAT, aims to represent each potential move in the chess game. The model obtains these embeddings by concatenating the embeddings of every source and target node for all edges in the graph. It is key to note that we only consider edges with edge features [0,]. This subset of edges signifies moves that are available for the current player, ensuring a focused, relevant output. In contrast, edges with features [1,] represent the opponent's potential moves, and are thus excluded from the move prediction distribution.

Once we have the chess board and move embeddings, we employ a self-attention mechanism to amalgamate these embeddings into a combined feature representation. In this process, the move embedding is concatenated with the pooled graph embedding from the first GAT and this serves as the query, key, and value for the self-attention mechanism. The result is a representation that effectively captures both the global state of the chess board and the specifics of individual moves.

To convert these representations into actionable outputs, we utilize a dual-headed approach, similar to the policy and value heads used in AlphaZero. The policy head downsamples the global move embeddings and outputs a probability distribution over the potential moves, while the value head evaluates the desirability of the current board state. This approach presents a significant advantage inherent to GNNs. Instead of encoding all potential moves on a chess board using a cumbersome matrix of size $8 \times 8 \times 73$, we can simply output a much more manageable distribution of size $n\_moves \times 1$. This distribution compactly encapsulates the likelihood of each possible move, thereby providing an elegant and computationally efficient solution. Figure 7 shows a representation of the described model. We also set up a comparative model that embeds the board using a GAT with high hidden embedding dimension and includes a value and policy head identical to the ResNet.

4.3.2 Hyperparameters

The architecture of our model hinges on several pivotal hyperparameters, as enumerated below:

1. *GNN Hidden Sizes and Number of Heads*: These parameters determine the dimensionality of the graph and edge embeddings output by the GATs and the number of parallel attention mechanisms, or "heads". The hidden sizes influence the model's ability to depict complex board states and diverse potential moves, while multiple heads enable capturing varied aspects of the board and move contexts.

2. *Number of GAT Layers*: This factor signifies the depth of each GAT in our model.

3. *Attention Embedding Size and Number of Heads*: The attention embedding size refers to the dimensionality of the combined graph and edge embeddings before applying self-attention. The number of self-attention heads is the count of individual attention mechanisms in this module. Together, these parameters govern the capability of the model to assimilate the graph and edge embeddings effectively.

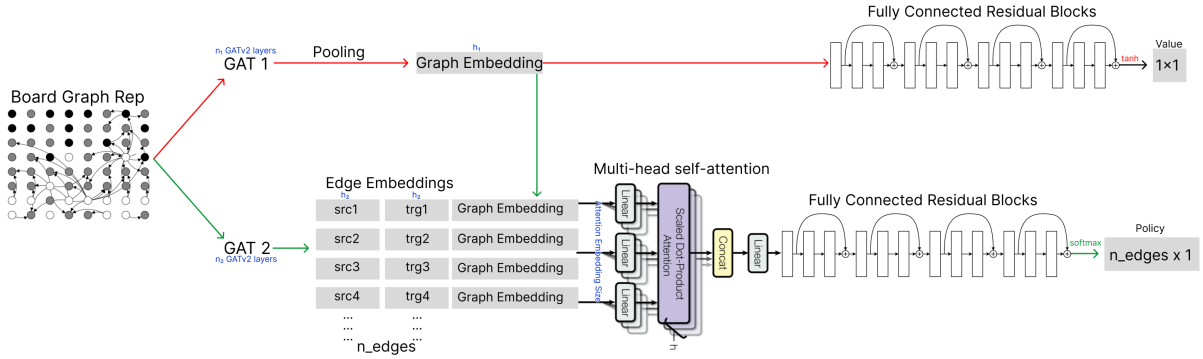


Figure 7: Overview of our GNN model. An input graph representation of a chessboard is processed by two distinct GNNs, each comprised of n_1 and n_2 GATv2 layers respectively, with hidden sizes denoted as h_1 and h_2. The first GNN output is pooled to give a graph embedding and this is forwarded to a value prediction head. The second GNN's embeddings are used to construct an edge-based representation with each edge corresponding to a potential move. The source and target node embeddings for edge n are represented by src_n and trg_n. These edge embeddings are then concatenated with the graph embedding from the first GNN to form enriched move representation. A self-attention mechanism is applied to the enriched move representations. The attention-modified embeddings are then processed by a policy head to yield a probability distribution over all possible chess moves.

4.4 Training Loop

The training loop is the main iterative part of our model's training process where the parameters of our model are updated incrementally with each pass of the data. Here, we delve into a detailed description of one cycle or loop in the training phase of our model.

We first collect a set of grandmaster-level chess games, specifically 5,000 games. This equates to approximately 400,000 distinct board positions. Alongside these games from grandmaster play, we also randomly generate 400,000 board positions following the procedure outlined in Section 3.2. This process is designed to enhance the model's exposure to a wider variety of board states, including scenarios which may not typically arise in high-level play. These randomly generated positions complement the grandmaster games to create a comprehensive training set that covers a wide spectrum of potential gameplay situations.

Once the data is collected, we proceed with the training of the network. Training occurs over one epoch, which involves one forward pass and one backward pass of all training examples through the network. During this phase, the network learns by updating its weights and biases to minimize the error in its predictions.

Our model employs the ADAM (Adaptive Moment Estimation) optimizer. The learning rate is set to $1 * 10^{-5}$, a relatively low rate that promotes stability and prevents the model from overshooting the optimal solution. Figure 8 shows a visualisation of this process.

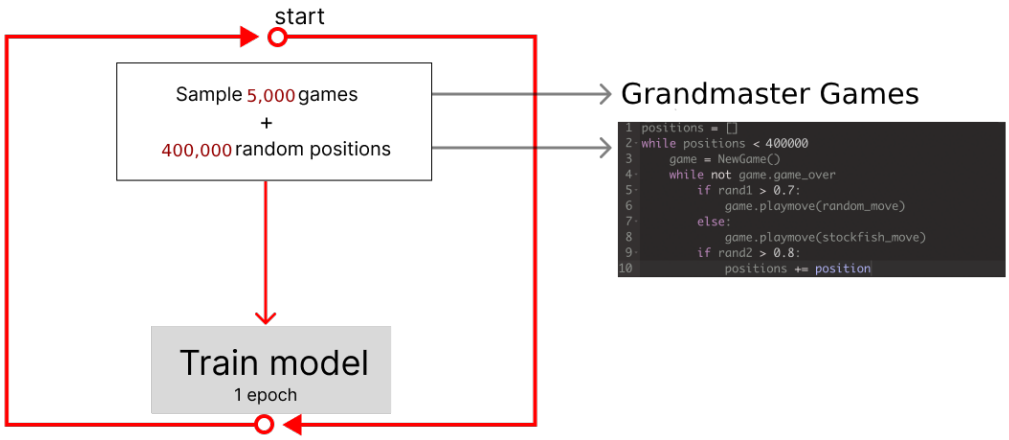


Figure 8: Schematic illustration of a single training loop in the model's learning process. The loop initiates with the collection of 5000 Grandmaster-level games and the generation of an additional 400,000 random board positions. The model then undergoes training for one epoch, using an ADAM optimizer to minimize a combined loss of Cross Entropy and Mean Squared Error.

The objective of our model is to concurrently minimize the Cross Entropy Loss and Mean Squared Error (MSE). The Cross Entropy Loss is used to optimize the policy head of the network, responsible for outputting a probability distribution over all legal moves from a given board position (in the case of our GNN with edge distribution output). This loss measures the dissimilarity between the predicted distribution and the true distribution. On the other hand, MSE is utilized to optimize the value head of the network that predicts the outcome of the game based on the current board position. The truth values for these predictions are sourced from Stockfish's evaluations (see Section 3).

5 Experiments & Results

In this section, we present comprehensive evaluations of our proposed model on various tasks. The main experiment being the comparison of our graph-based model's performance with baseline array representation models.

5.1 Setup

This work was performed using the ALICE compute resources provided by Leiden University. All experiments were run on a A100 GPU and 40GB of RAM. Implementations are in Python. This GitHub repository contains all code used in the methods and experiments.

5.2 Evaluation Metrics

The performance of our model is gauged on two main aspects: the prediction of the board position value, and the probability distribution of potential moves. We employed different evaluation metrics to assess these aspects thoroughly.

The score predictions are evaluated using MSE. The MSE is computed by averaging the squared differences between the values predicted by our model and the true values. Mathematically, for the i^{th} board position, if y_i represents the true value, $\hat{y}_i$ is the predicted value,

and N is the number of samples, the MSE is given by the formula:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 \tag{3}$$

On the other hand, the model's move prediction performance was evaluated using two interrelated metrics. Firstly, we used the Cross Entropy loss, which quantifies the dissimilarity between the model's predicted policy (probability distribution over possible moves) and the true policy. If $y_{o,c}$ is the true policy and $p_{o,c}$ is the predicted policy for class c in observation o, the Cross Entropy loss is calculated using the formula:

$$CE = -\sum_{c=1}^{N} y_{o,c}\log(p_{o,c}) \tag{4}$$

For evaluating move predictions, we also use accuracy measures. Firstly, $best\_move\_acc$ measures the frequency with which the model's top move choice (i.e., move with the highest predicted probability) aligns with the highest probability move in the true policy. It is given by:

$$best\_move\_acc = \frac{1}{N}\sum_{i=1}^{N} I(argmax(\hat{y}_i) = argmax(y_i)) \tag{5}$$

In addition, we calculated $move\_acc$, which quantifies the frequency with which the move with the highest predicted probability by our model is one of the moves from the true policy (i.e., a non-zero entry). It is represented as:

$$move\_acc = \frac{1}{N}\sum_{i=1}^{N} I(argmax(\hat{y}_i) \in M_i) \tag{6}$$

Here, $\hat{y}_i$ and y_i are the predicted and true policy vectors for the i^{th} position, respectively. Furthermore, $argmax$ represents the move with the highest probability in the respective policy vector, and M_i is the set of moves with non-zero probability in the true policy. The indicator function I equals 1 if the condition in the parentheses is true, and 0 otherwise. The accuracy measures, $best\_move\_acc$ and $move\_acc$, both provide insight into the practical applicability of the model's move predictions by showing how often the model's top choice aligns with expert moves.

5.3 Hyper-parameter Optimisation (HPO)

Our approach towards optimizing the hyperparameters for the GNN model involves using the Tree-structured Parzen Estimator (TPE) algorithm, provided by the HyperOpt Python library (Bergstra et al., 2015). This method offers an efficient way to explore the hyperparameter space by constructing a surrogate model that captures the relationship between hyperparameters and model performance.

We optimize over a predefined hyperparameter space (Table 2), which includes elements such as the number of layers in both the value and policy GATs, the dimensionality of graph and edge embeddings, the size of the attention embedding, the number of heads in the Graph Attention mechanism, and the total number of GAT layers.

Hyperparameter	Description	Possible Values
value_nlayers	Number of layers in the value head	1, 3, 5
pol_nlayers	Number of layers in the policy head	1, 3, 5
hidden_graph	Dimensionality of the first GAT (graph embedding)	256, 512, 1024, 2048
hidden_edge	Dimensionality of the second GAT (edge embedding)	256, 512, 1024, 2048
heads_GAT_graph	Number of heads in the first GAT (graph embedding)	16, 32, 64, 128, 256
heads_GAT_edge	Number of heads in the second GAT (edge embedding)	16, 32, 64, 128, 256
att_emb_size	Size of the global attention embedding in the policy head	128, 256, 512, 1024, 2048
GAheads	Number of heads in the global attention embedding in the policy head	16, 32, 64, 128
n_layers	Total number of layers in each GAT	1, 3, 5, 7, 9, 11

Table 2: Hyperparameters, their descriptions, and possible values.

The optimization process follows a comprehensive routine. For each configuration sampled by the TPE algorithm, the model is trained for 2 epochs using a sampled training dataset. Each sampled training dataset includes 5,000 human grandmaster games and 400,000 randomly generated positions. The validation set comprises a sampled 500 human grandmaster games and an additional 40,000 randomly generated positions. The selection criterion for a configuration is determined by its performance, quantified as the sum of the MSE and cross-entropy loss on the validation set. To ensure consistency in performance evaluation, the same training and validation sets are employed across all configuration tests. We run TPE with a budget of 50 configuration tests.

Figure 9 illustrates the results of the HPO process, highlighting the trade-offs between different parameters and the resultant model performance.

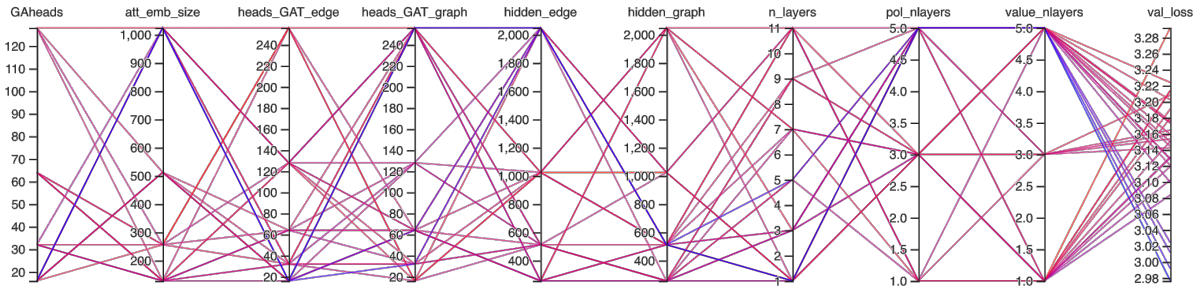


Figure 9: HPO Results. Each line is a configuration tested by TPE, colored by *val_loss* which is the summation of MSE and cross entropy on the validation set. The best configurations all used a higher hidden dimension for the edge GNN compared to the graph GNN. All also used the largest attention embedding size of 1024.

In our HPO process, the configurations that yielded superior performance showed a clear pattern: the hidden size of the node embeddings of the GNN used for move embeddings was consistently larger than that of the GNN used for graph embeddings. Additionally, the size of the global attention embedding (in the policy head) was on the upper end in these top-performing configurations, always with a low number of heads, indicating a potential correlation between larger attention embedding size and model efficacy.

Considering the setup of this experiment, it is important to note that it may exhibit a slight bias towards simpler, shallow models. This is primarily due to the limited size of the dataset and the small number of training epochs in the HPO setup, which may favor less complex models. In fact, three of the four best configurations — those with a loss below 3.03 — utilized just one layer in the GNNs.

The third-best configuration identified through the HPO process stands out as an optimal choice for our next experiments, which will involve more epochs of training with larger datasets.

This configuration is identical to the top-performing configuration in all aspects, except for the number of layers in the GNNs being five instead of one. The chosen configuration consists of:

- 16 heads in the global attention embedding and a size of 1024
- 256 heads in the first GAT (graph embedding) and 16 in the second GAT (edge embedding)
- Edge embedding dimension of 2048 and a graph embedding dimension of 512
- 5 GAT layers each
- 5 layers in each of the policy and value heads

This configuration corresponds to a model comprising a total of 218M parameters.

5.4 Modelling Stockfish

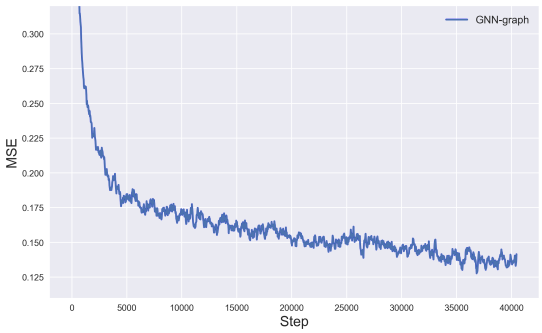
In this section we compare the use of a GNN with graph representations of the board to ResNet with array representations in the task of modelling chess boards to a policy and value. We experiment with three models: 1.*GNN-graph*, the GNN outlined in Section 4.3 with the configuration from the previous sub-section, 2.*ResNet*, the array representation baseline ResNet model and 3.*GNN-array*, a model that embeds the board using a GNN (embedding size 2048) that is passed to value and policy heads (identical to *ResNet*), operating similarly to *ResNet* but with a GNN to embed the board.

We follow the training loop mentioned in Figure 8. In total, each model will be trained on 135K games and 11M positions. The 135K games contain $\approx$ 11M positions. Since each loop samples 5,000 games, we need 27 loops to train on all our data. The last 15K games are used to comprise the test set. In observing the training process for the various models, distinctive performance patterns can be identified. The *GNN-graph* model, displayed in Figure 10, exhibits significant stability throughout the training process compared to *ResNet* and *GNN-array* models illustrated in Figure 11. This improved stability can likely be attributed to the graph-based approach of the *GNN-graph* model, which seems to provide a conducive environment for the learning process in chess prediction tasks.

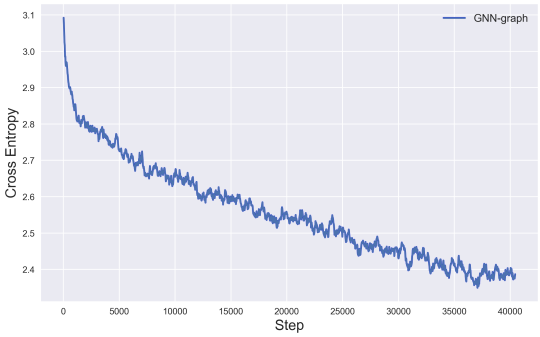
The GNN models, both *GNN-array* and *GNN-graph*, demonstrate superior MSE performance in comparison to *ResNet*. This superiority in performance underlines the potential of GNNs in effectively modeling chess games.

The *GNN-graph* model incorporates the policy along the edges of the graph, leading to a varying number of classes. As a result, a direct comparison of the Cross Entropy cannot be made between *GNN-graph* and the other models.

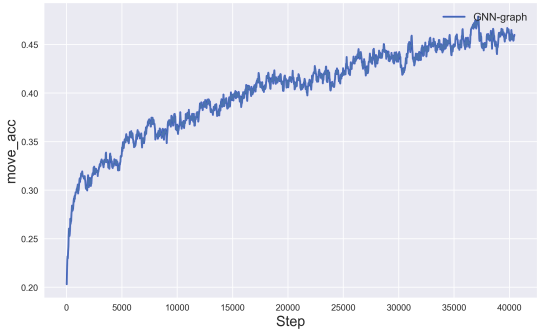
In Table 3, the test performance of the three models is presented. The test dataset comprises of 15,000 unseen grandmaster games and 1 million randomly generated positions. The table demonstrates the superiority of the *GNN-graph* model with respect to Move Accuracy, Best Move Accuracy, and MSE. The *GNN-graph* test results closely mirror those obtained during training, indicating a minimal degree of overfitting.



(a) MSE over training steps.

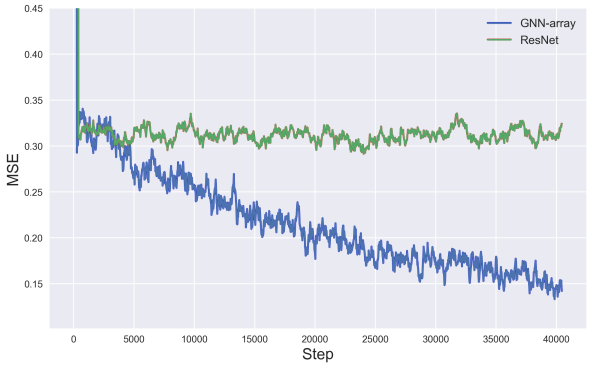


(b) Cross entropy over training steps.

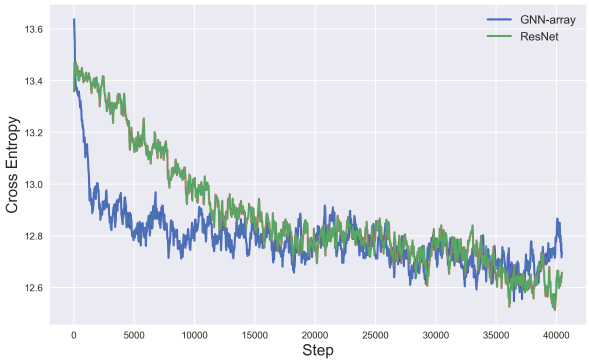


(c) Move accuracy over training steps.

Figure 10: Performance over training steps for the *GNN-graph* model with a batch size of 512. This model learns in a more stable manner and outperforms baseline models in terms of MSE and move accuracy.



(a) MSE over training steps.



(b) Cross entropy over training steps.

Figure 11: Performance over training steps for *ResNet* and *GNN-array* models with a batch size of 512. Learning is unstable, especially in later epochs for both networks. GNN-Array outperforms *ResNet*, especially in capturing the value (MSE).

Model	MSE	Move Acc	Best Move Acc
GNN-graph	**0.13**	**0.51**	**0.40**
GNN-array	0.15	0.26	0.12
ResNet	0.31	0.25	0.12

Table 3: Comparison of test performance across the *GNN-graph*, *GNN-array*, and *ResNet* models. *GNN-graph* performs best in all metrics.

To further test our models, we have conducted a series of matches where each model played against the others in a series of 1,000 games. The selection of moves during these matches was stochastic, leveraging the policy output from each model. Each move had a chance of being selected equivalent to its value in the softmax-transformed policy output. This method ensures a degree of variability in the gameplay and reduces the possibility of repeating the same sequences of moves. Ensuring non-repeating gameplay and an extensive coverage of potential scenarios, provides a robust evaluation of the models' abilities to handle different situations. The games' results are summarized in Table 4 which demonstrate superior performance of the *GNN-graph* model.

Match	Player 1 wins	Player 2 wins	Draws
GNN-graph vs *GNN-array*	509	3	488
GNN-graph vs *ResNet*	498	2	553
GNN-array vs ***ResNet***	66	78	856

Table 4: Results of 1,000 games played between pairs of models with moves chosen stochastically based on the softmax-transformed policy output from each model. The *GNN-graph* model consistently wins against the other models.

5.5 Transfer Learning to Specific Human Players

In this section, the pre-trained model from the previous experiment is loaded for a sequence of fine-tuning operations. The model is prepared for this phase by freezing all the parameters associated with the value prediction (the graph GNN and the value head). Additionally, the first half of the edge GNN is also frozen to conserve some aspects of the move representation derived during the pre-training phase.

The fine-tuning targets the style of specific human players. The respective datasets for these players, described in Section 3.3, are loaded and divided into training and test sets in an 80/20 proportion.

The semi-frozen model is then trained on the player-specific training set for 5 epochs, with a reduced learning rate of 5×10^{-6}. This lower learning rate is employed to prevent drastic changes to the pre-learned parameters, and to promote the subtle adjustments that are characteristic of fine-tuning processes.

The results of this evaluation on each of the human datasets are presented in Table 5. The table shows the performance before and after fine-tuning. Pre-trained performance is improved upon, especially in lower quality moves.

Player	Pretrained Accuracy		Finetuned Accuracy	
	Train	Test	Train	Test
Firouzja (2.1k games)	0.309	0.304	0.397	0.362
Karpov (3.3k games)	0.314	0.315	0.380	0.361
Average player (8.1k games)	0.347	0.351	0.517	0.498

Table 5: Comparison of model performance (accuracy) before and after fine-tuning on specific human player train and test datasets. The pretrained model (trained on Stockfish data from the previous section) improves its performance on player specific data after finetuning with little to no overfitting.

6 Discussion

We investigated the applicability of GNNs in chess AI. The results validate the use of graph representation for embedding a chess board to extract policy or value. The performance of GNN-based models was compared to existing state-of-the-art models, namely ResNets with array representations. Our findings demonstrated that the GNN model offered superior performance and facilitated more stable learning.

A key distinguishing feature of our proposed model was the representation of the policy along the graph's edges. This novel approach emerged as a more efficient method of learning the policy, outperforming the ResNet model, which employs an array policy output. Furthermore, in a head-to-head gameplay between the GNN model and the ResNet model, using their policy output for moves, the GNN model consistently emerged superior. It it pertinent to note that the *GNN-array* model, which also outputs an array policy similar to the ResNet model, was significantly outperformed by *GNN-graph*. This further underscores the efficacy of policy representation along the edges.

The embedding of a graph representation of a chess board encapsulates information for deep learning purposes than conventional array-based representations. This assertion is underscored by the comparative experiments involving the GNN and ResNet models, wherein the GNN models demonstrated a superior ability to accurately ascertain the value of a given state from its board embedding. This finding suggests that GNNs, through their inherent ability to capture relational information between chess pieces, provide a more effective mechanism for capturing the complex dynamics of a chess game compared to conventional methods.

In addition to developing the GNN model, this study also sought to optimise its hyperparameters to enhance its performance further. Our experiments revealed that the GAT responsible for producing the move embeddings should have a higher size than the GAT for the board embedding. Moreover, in the self-attention procedure in the policy head, a high attention embedding size (in our case, the highest option, 1024) proved beneficial.

To extend the applicability of our GNN model beyond the Stockfish games, we fine-tuned the model on a smaller dataset of games played by specific players. Despite the dataset's small size, the fine-tuned model performed on par with its performance on the Stockfish data, suggesting its potential for wider use. The model exhibited the best learning ability on a low quality moves, likely due to the repetitiveness and relatively simple structure of these games.

While our study did not implement training in a reinforcement learning environment like AlphaZero, primarily due to the computational intensity and resources required for MCTS data gathering, we still managed to train the models extensively. In terms of training iterations

on the model, our experiments handled a large dataset, whose size is comparable in order of magnitude to that used in AlphaZero. AlphaZero underwent training for approximately 700k steps with a batch size of 4096, which equates to around 2.9 billion positions. Conversely, our models were trained for 45k steps with a batch size of 512, amounting to around 25 million positions. This difference is roughly by a factor of 100x, yet considering the extreme scale of training that AlphaZero underwent, our experiment's training volume was significant. Therefore, despite the aforementioned constraints, the experiment's results hold validity and provide substantial insights into the applicability of GNNs in chess AI.

The results of our hyperparameter optimisation could potentially be biased due to computational resource constraints, only conducting two epochs of training per configuration on a relatively small dataset. Nevertheless, our findings offer substantial evidence to support the effectiveness of GNNs in chess AI, warranting further research to exploit their full potential.

7 Conclusion

The primary objectives of this research revolved around employing GNNs in the domain of chess modelling. The first goal was to devise a graph-based representation for a chess board conducive to deep learning. To this end, we combined previous studies on graph theory in chess with AlphaZero's board array representation, generating a set of node features, edge features, and an edge list that collectively embody a board. The resulting representation synergized effectively with GATs, as evidenced by our experiments.

Our second aim was to compare the performance of GNNs and ResNets in modelling the value and policy of a chess board. The outcomes of our comparative experiments demonstrated the superior performance of the GNN model over its ResNet counterpart. The cornerstone of this improvement was our innovative approach of outputting the policy along the edges of the graph.

The third objective was to determine the most effective GNN architecture for chess modelling. Although our study only experimented with GATs for embedding the board, the insights from the HPO results offer valuable guidance regarding the relative sizes of the GATs' embeddings and the depth of the model.

Furthermore, fine-tuning the GNN on specific players' games gave a relatively good performance. The potential applications of this could range from creating player-specific bots to aiding in the detection of cheating through anomaly identification in gameplay. The results reiterate the robust potential of GNNs for chess prediction tasks.
This research underscores the potential of GNNs in game modelling, particularly chess. It contributes to the current understanding of machine learning for game prediction and opens avenues for its practical applications.

7.1 Future Work

This research substantiates the effective application of GNNs, particularly GATs, within the context of chess modelling. As a next step, future investigations could extrapolate these techniques to chess and other strategic games. The logical continuation of this research is to implement a chess GNN model in a reinforcement learning environment analogous to AlphaZero. We were unable to do this due to its high computational demand. Training to the extent of AlphaZero is infeasible without the use of tremendous computation. An alternative

direction could be the exploration of a more efficient search algorithm, shifting the focus from the deep learning aspect.

Our current attention mechanism seems somewhat rudimentary, with the self-attention mechanism being preceded by the concatenation of all edge embeddings with the graph embedding. Future research could investigate a more sophisticated attention mechanism that maximize information extraction.

References

Bergstra, J., Komer, B., Eliasmith, C., Yamins, D., and Cox, D. D. (2015). Hyperopt: A Python library for model selection and hyperparameter optimization. *Computational Science & Discovery*, 8(1):Article 014008. https://dx.doi.org/10.1088/1749-4699/8/1/014008.

Brody, S., Alon, U., and Yahav, E. (2022). How Attentive are Graph Attention Networks? In *International Conference on Learning Representations*. https://openreview.net/forum?id=F72ximsx7C1.

Campbell, M., Hoane, A. J., and Hsu, F.-h. (2002). Deep Blue. *Artificial Intelligence*, 134(1–2):57–83. https://www.sciencedirect.com/science/article/pii/S0004370201001291.

Chess Programming Wiki contributors (2020). StockfishNNUE. https://www.chessprogramming.org/Stockfish_NNUE. [Online; accessed 7-June-2023].

Cuevas, R. L. (2021). GM Chess Games. https://www.kaggle.com/datasets/lazaro97/gm-chess-games. Accessed 10-May-2023.

Edwards, D. J. and Hart, T. P. (1961). The Alpha-Beta Heuristic. Technical Report AIM-030, MIT Artificial Intelligence Laboratory. http://hdl.handle.net/1721.1/6098.

Giannini, T. and Bowen, J. (2017). Life in Code and Digits: When Shannon Met Turing. In *Electronic Visualisation and the Arts (EVA 2017)*. https://www.scienceopen.com/hosted-document?doi=10.14236/ewic/EVA2017.9.

Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., and Dahl, G. E. (2017). Neural Message Passing for Quantum Chemistry. *arXiv preprint cs.LG*. http://arxiv.org/abs/1704.01212.

He, K., Zhang, X., Ren, S., and Sun, J. (2015). Deep Residual Learning for Image Recognition. *arXiv preprint CoRR*. http://arxiv.org/abs/1512.03385.

Kipf, T. N. and Welling, M. (2017). Semi-Supervised Classification with Graph Convolutional Networks. *arXiv preprint cs.LG*. http://arxiv.org/abs/1609.02907.

Kjeldsen, T. H. (2001). John von Neumann's Conception of the Minimax Theorem: A Journey Through Different Mathematical Contexts. *Archive for History of Exact Sciences*, 56(1):39–68. http://www.jstor.org/stable/41134130.

OpeningTree (2023). OpeningTree: Chess Opening Explorer. https://www.openingtree.com/. [Online; accessed 1-July-2023].

Pascutto, G. C. and Linscott, G. (2020). Leela Chess Zero. http://lczero.org/. [Online; accessed 7-June-2023].

Rudolph-Lilith, M. (2019). ChessY: A Mathematica toolbox for the generation, visualization and analysis of positional chess graphs. *SoftwareX*, 9:39–43. https://www.sciencedirect.com/science/article/pii/S2352711018301687.

Sabatelli, M. (2017). *Learning to Play Chess with Minimal Lookahead and Deep Value Neural Networks*. PhD thesis. https://doi.org/10.13140/RG.2.2.31956.71040.

Sharp, D. (2020). How to analyse chess games using graph networks. https://medium.com/applied-data-science/how-to-analyse-chess-games-using-graph-networks-38dd3b77d4be. [Online; accessed 13-March-2023].

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K., and Hassabis, D. (2018). A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362(6419):1140–1144. https://www.science.org/doi/abs/10.1126/science.aar6404.

Veličković, P., Cucurull, G., Casanova, A., Romero, A., Liò, P., and Bengio, Y. (2018). Graph Attention Networks. *arXiv preprint stat.ML*. http://arxiv.org/abs/1710.10903.

Wikipedia contributors (2023a). Knight's graph — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Knight%27s_graph&oldid=1143336799. [Online; accessed 8-June-2023].

Wikipedia contributors (2023b). Rook's graph — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Rook%27s_graph&oldid=1153169963. [Online; accessed 8-June-2023].