Hybrid Chess AI: Learning from Humans and Self-Play

Nakul Narang*, Farzan Mirza*, Amirreza Dast Parvadeh*
*College of Computing and Informatics, Drexel University, Philadelphia, PA, USA
Email: {nn474, fm474, ad4255}@drexel.edu

***Abstract*—This paper explores the development of chess playing agents using both imitation learning and deep reinforcement learning (DRL) techniques, leveraging an Lc0-style encoding for board states and actions. We detail the implementation of a neural network architecture inspired by AlphaZero and Lc0, capable of learning expert move distributions (policy) and game outcomes (value) through supervised training on PGN game data. Furthermore, we present a Deep Q-Network (DQN) agent designed to learn optimal policies through self-play, adapting the Lc0 encoding for its state and action spaces. Our contributions include a robust data pipeline for PGN processing, a custom neural network for chess, and the integration of these components into both imitation learning and DRL frameworks. Initial results demonstrate the feasibility of this approach for learning to play chess, highlighting the strengths and weaknesses of each methodology.**

***Index Terms*—Chess AI, Deep Learning, Imitation Learning, Deep Reinforcement Learning, DQN, Game AI, Neural Networks, Lc0.**

I. Introduction

The game of chess has long served as a challenging benchmark for artificial intelligence research. Traditional chess engines relied on extensive search algorithms and hand-crafted evaluation functions. However, the advent of deep learning has revolutionized the field, with models like AlphaZero demonstrating superhuman performance by learning solely through self-play, without any human-provided knowledge beyond the rules of the game [1]. This success is largely attributed to the use of deep neural networks for approximating both the policy and value functions, coupled with sophisticated search algorithms like Monte Carlo Tree Search (MCTS).

This project investigates two primary approaches to developing a chess AI: imitation learning and deep reinforcement learning. Imitation learning focuses on replicating expert behavior by training a neural network on a dataset of expert human games. Deep Q-Networks (DQN) aims to learn optimal policies through direct interaction with the environment (self-play). A crucial aspect of both approaches is the representation of the chess board state and the encoding of possible moves into a format suitable for neural networks. We adopt an Lc0-style encoding scheme, inspired by the highly successful open-source chess engine Leela Chess Zero, which uses a multi-plane representation for the board state and a flattened categorical representation for moves [2].

Our research contributions are threefold. First, we developed a robust data pipeline for extracting chess game states and expert moves from PGN files, incorporating historical board states and game outcomes as training targets. This pipeline includes Lc0-style encoding of board positions and moves, handling board symmetry for the current player's perspective . Second, we implemented a custom deep neural network (ChessNet) in PyTorch, featuring a residual block architecture similar to AlphaZero and Lc0, designed to simultaneously predict move probabilities (policy head) and game outcomes (value head) . Third, we integrated this ChessNet into an imitation learning framework, demonstrating its ability to learn from expert game data . Finally, we adapted the Lc0 encoding and ChessNet for a DQN-based reinforcement learning agent, capable of learning a chess policy through self-play. This includes designing a custom policy and Q-value estimation mechanism for the DQN agent .

II. Related Work

This section highlights the foundational deep learning techniques and specific encoding schemes that are referenced in this project for development.

A. Imitation Learning in Games

Imitation learning, also known as behavioral cloning, is a supervised learning paradigm where an agent learns to mimic the actions of an expert. For chess, this translates to feeding the network historical board states and having it predict the subsequent move made by a strong human or AI player. A relevant example of applying imitation learning to strategic board games can be found in [3]. While capable of quickly learning reasonable policies, imitation learning is inherently limited by the quality and diversity of the expert demonstrations, and cannot surpass the performance of the expert .

B. Deep Q-Networks (DQN) for Reinforcement Learning

Deep Q-Networks are a seminal algorithm in deep reinforcement learning that revolutionized learning in complex environments, particularly in Atari games [4], [5]. DQN extends traditional Q-learning by using a deep neural network to approximate the Q-function, $Q(s, a)$, which estimates the maximum expected future reward for taking action a in state s.

The key innovations of DQN include experience replay, where past transitions are stored in a replay buffer and randomly sampled to break correlations and improve data

efficiency [4], [5]. To stabilize the training process, DQN employs a separate "target network" to compute the target Q-values for the Bellman equation; this target network's weights are periodically updated from the main Q-network, rather than every training step, preventing oscillations and divergence [4], [5]. Furthermore, an epsilon-greedy exploration strategy is used to balance exploration and exploitation . Applying DQN to chess presents significant challenges due to the vast action space (4672 possible moves) and the long-term, sparse rewards.

C. Lc0-Style Encoding for Chess

A critical component of modern high-performing chess AI, particularly engines inspired by AlphaZero like Leela Chess Zero (Lc0), is the efficient and effective representation of chess board states and moves as input to neural networks [2]. This project directly adopts an "Lc0-style" encoding scheme for this purpose. The Lc0 encoding converts a complex chess board into a multi-plane (e.g., 119 planes in our implementation) 8x8 grid of numerical values, where different planes represent various aspects of the board, such as the position of different piece types (for both colors), castling rights, en passant opportunities, and even historical board states . This multi-plane representation allows convolutional neural networks to leverage their spatial processing capabilities effectively. Similarly, chess moves are translated into a flattened, categorical action space. Instead of representing moves as (from_square, to_square, promotion_piece), Lc0 encodes them into a single integer index by categorizing them into "queen-like" (moves along rays), "knight moves," and "pawn promotions" from each possible starting square . This results in a large but discrete action space (4672 possible encoded moves in our case) that is amenable to the output layer of a neural network and is particularly well-suited for algorithms like DQN that operate on discrete action spaces. This unified, symmetrical encoding, always from the perspective of the current player, streamlines the network's learning process.

III. Approach

Our approach involves two main phases: data preparation and agent training. Both rely on a consistent, Lc0-inspired representation of the chess board and moves.

A. Problem Formulation: Chess as a Sequential Decision-Making Problem

Chess is a sequential decision-making problem played in a two-player, zero-sum environment. For the purpose of training a single agent, we model the game as follows. A state $s \in S$ represents a complete snapshot of the chess game at a given moment, including the positions of all pieces on the 8x8 board, the current player to move, castling rights for both sides, the en passant target square, the half-move clock (for the 50-move rule), and the full-move number. To provide context for the neural network, our state representation also incorporates historical board positions (8 frames of history) to help identify patterns like repetitions . An action $a \in A$ corresponds to a legal chess move from the current state; the set of legal actions varies depending on the board configuration. Rewards are sparse and are received at the end of a game: +1 for a win, -1 for a loss, and 0 for a draw, from the perspective of the agent being trained. The primary goal is for a neural network to learn a **policy** (which move to make) and a **value function** (how good a position is) from these states and actions.

B. Lc0-Style Encoding for Network Input

Before being fed into a neural network, the raw chess board state and moves must be converted into numbers. We employ an Lc0-style encoding, which is highly efficient for convolutional neural networks.

1) Board State Encoding: The board state is transformed into a NumPy array with 119 planes of 8x8 spatial dimensions . This multi-plane representation captures comprehensive information about the board and game context. This includes 96 piece planes, where for each of the 8 most recent historical board states, there are 12 planes representing the presence or absence of each of the 6 piece types (Pawn, Knight, Bishop, Rook, Queen, King) for both the current player and the opponent . This historical context is crucial for detecting repetitions and understanding evolving positional trends. Additionally, 23 auxiliary planes provide non-spatial game state information for the *current* board position, such as side to move, en passant target square, castling rights (4 planes), half-move clock, full-move number, repetition flag, and additional standard Lc0 planes (e.g., zeros, ones, player color) . Crucially, for black's turn, the entire board representation is flipped 180 degrees, and piece planes and castling rights are symmetrically swapped. This ensures that the neural network always perceives the board from the current player's perspective, simplifying the learning task and allowing for shared weights .

2) Move Encoding: Legal chess moves are encoded into a single integer index within a large discrete action space of 4672 possible moves . This encoding scheme categorizes moves into three types: queen-like moves (0-55), which cover moves along horizontal, vertical, and diagonal lines, with 8 directions and up to 7 squares per direction ; knight moves (56-63), which are the 8 unique "L-shaped" moves a knight can make ; and pawn promotions (64-72), which handles pawn moves that result in a promotion, categorizing them by the type of pawn move (straight, diagonal left, diagonal right) and the piece promoted to (Queen, Rook, Bishop) . Knight promotions are typically mapped to Queen promotions in this scheme . The final encoded move index is calculated by combining the "from square" (0-63) with an "action plane" index (0-72) representing the type of move. Symmetry is also applied to the "from square" when it's black's turn to maintain consistency with the board state encoding .

C. Approach 1: Imitation Learning for Behavioral Cloning

Imitation learning aims to train a deep neural network to mimic expert chess behavior by learning directly from a dataset of expert games.

1) Data Collection and Preprocessing: We gather PGN (Portable Game Notation) files from blitz games played by Magnus Carlsen . The process_pgn_to_training_data_lc0 function parses these PGNs. For each move in a game, it extracts the Lc0-encoded board state just before the move (including 8 frames of history) , the Lc0-encoded expert move , and the final outcome of the game (win, loss, or draw), normalized to +1, -1, or 0 from the perspective of the current player at that specific board state . This creates a dataset of (state, expert_action, game_outcome) tuples, which serve as supervised training examples.

2) Network Architecture and Training: The core neural network architecture used for imitation learning is comprised of an input block, followed by five residual blocks, and two distinct output heads: a policy head and a value head . The input block consists of a convolutional layer that processes the 119 input planes from the Lc0-encoded state, followed by batch normalization and ReLU activation, expanding the feature space to 128 channels . Five `ResidualBlocks` further process these features, each containing two convolutional layers with batch normalization and ReLU activations, and a skip connection that adds the input of the block to its output, aiding in gradient flow and training deeper networks .

From the output of the residual blocks, the policy head uses a 1x1 convolutional layer to map the features to 73 action planes, followed by batch normalization and ReLU. This output is then flattened into a 4672-dimensional vector, representing the logits for each possible Lc0-encoded move . During inference, a softmax activation is applied to these logits to obtain a probability distribution over moves, allowing the agent to select the most likely expert move. The value head is a separate 1x1 convolutional layer that maps the features to a single channel, followed by batch normalization and ReLU. This output is flattened and passed through two fully connected layers (128 units, then 1 unit) . A tanh activation is applied to the final scalar output, scaling the value prediction between -1 (representing a loss for the current player) and +1 (representing a win for the current player) .

This network is trained in a supervised manner using a multi-task learning objective, minimizing a weighted sum of two loss components. The policy loss ($\mathcal{L}_{\text{policy}}$) is a cross-entropy loss between the predicted policy logits from the network's policy head and a one-hot encoded vector of the actual expert move ; the objective is to make the network output a high probability for the exact move chosen by the expert. The value loss ($\mathcal{L}_{\text{value}}$) is a Mean Squared Error (MSE) between the predicted value from the network's value head and the actual normalized game outcome ; the objective here is to make the network accurately predict the final result of the game from any given position. The total loss is $\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{policy}} + C \cdot \mathcal{L}_{\text{value}}$, where C is a weighting constant. During training, gradients are computed for this total loss and backpropagated through the network to update its weights using the Adam optimizer . This process encourages the network to simultaneously learn both where the expert would move and what the likely outcome of the game is from that position.

D. Approach 2: Deep Reinforcement Learning with DQN

For the reinforcement learning approach, we implement a Deep Q-Network (DQN) agent to learn optimal policies through self-play within the chess environment.

1) Experience Buffer Initialization and Game Interaction: The DQN agent learns from an experience replay buffer, which stores past transitions (state, action, reward, next state). Crucially, this buffer is initialized by pre-filling it with transitions extracted from Magnus Carlsen's games . This pre-filling aims to provide the agent with a strong initial understanding of chess dynamics and expert play, potentially accelerating the subsequent reinforcement learning phase and guiding initial exploration. After this initialization, the agent continuously generates new experiences through self-play. In each step of self-play, the agent interacts directly with a `chess.Board()` object. It observes the current Lc0-encoded board state, selects an action (an Lc0-encoded move), applies it to the board, and receives a reward (if the game ends) and the next board state. If an illegal move is attempted, a large negative reward is assigned, and the game terminates, penalizing invalid actions. Rewards are sparse: +1 for a win, -1 for a loss, and 0 for a draw, received only at the end of the game.

2) Network Architecture and Q-Value Estimation: The DQN agent utilizes a neural network based on the same core convolutional architecture as the imitation learning model. This network's architecture for DQN is adapted to output Q-values. The Q-network structure includes an input block and residual blocks similar to the imitation learning network . However, its output head is specifically configured to produce raw Q-values for each of the 4672 possible Lc0-encoded actions . That is, instead of producing logits for a softmax policy, it directly outputs a score for each action, representing its estimated long-term return. During training, the DQN agent employs an epsilon-greedy strategy. With probability ϵ, it explores by choosing a random legal move from the current board state; otherwise, it exploits by selecting the legal move with the highest predicted Q-value from its Q-network. During evaluation, ϵ is typically set to 0, and only the highest Q-value action is chosen.

3) Training and Objective Function: The DQN agent learns by sampling mini-batches of transitions from its experience replay buffer (which now contains both expert demonstrations and self-play experiences). The training objective for DQN is to minimize the difference between the predicted Q-value for a state-action pair and its "target Q-value." The loss function is typically Mean Squared Error (MSE):

$$\mathcal{L}_{\text{DQN}} = (Q(s, a; \theta) - Q_{\text{target}})^2 \tag{1}$$

where $Q(s, a; \theta)$ is the Q-value predicted by the main Q-network, and Q_{target} is calculated using the Bellman equation and a separate, periodically updated target network:

$$Q_{\text{target}} = r + \gamma \max_{a'} Q(s', a'; \theta_{\text{target}}) \tag{2}$$

Here, r is the immediate reward received from the environment, γ is the discount factor (determining the importance of future rewards), and $\max_{a'} Q(s', a'; \theta_{\text{target}})$ is the maximum Q-value for the next state s' and optimal action a' predicted

by the target network. Gradients are backpropagated through the main Q-network based on this loss, updating its weights θ to improve Q-value estimations. The target network's weights θ_{target} are periodically synchronized with the main network's weights, providing stability to the learning process. This iterative process allows the agent to gradually learn an optimal policy by refining its understanding of the long-term value of different actions in various chess states.

IV. Results/Evaluation

This section presents the evaluation of both the imitation learning and Deep Q-Network (DQN) agents using two criteria: performance on a held-out test set and gameplay results against baseline opponents (Stockfish Level 1 and a random agent). Gameplay outcomes are summarized in Table I.

A. Imitation Learning Evaluation

The imitation learning agent was trained using the Adam optimizer with a learning rate of 1×10^{-3}, a batch size of 128, and for 20 epochs. During training, the policy and value losses decreased consistently across epochs, indicating stable convergence. Figure 1 shows the policy and value loss curves, reflecting this steady decline.

On the held-out test set of 10,627 samples, the model achieved a final policy loss of 7.5671 and a value mean absolute error (MAE) of 0.2913. These values suggest that the network was able to learn a representation of expert play but struggled to generalize, likely due to the relatively small size and scope of the training data.

Gameplay evaluation revealed limited practical capability. The model lost all 100 games against Stockfish level 1. Against a random move agent, the model showed modest competence: 7 wins, 9 losses, and 84 draws. This indicates that the agent could exploit obvious mistakes but lacked deeper strategic planning.

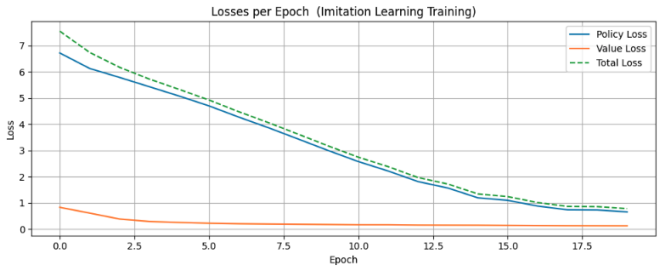

Fig. 1. Training Loss Curves for Imitation Learning: Policy Loss and Value Loss over 20 epochs.

B. Deep Q-Network (DQN) Evaluation

The DQN agent was trained using the Adam optimizer with a learning rate of 1×10^{-4}, a replay buffer size of 100,000, a discount factor $\gamma = 0.99$, and target network updates every 500 steps. Exploration was handled using an epsilon-greedy policy, with ϵ decaying linearly from 1.0 to 0.01 over 50,000 steps.

Unlike the imitation model, the DQN agent did not show consistent learning behavior. Figure 2 illustrates the training loss curve, which remained erratic and did not exhibit a downward trend. This instability suggests ineffective Q-value learning—likely due to the vast action space, sparse reward signals, or sampling inefficiencies from the replay buffer.

In gameplay evaluation, the DQN agent lost all 100 games against Stockfish level 1. Against the random agent, it managed 8 wins, 7 losses, and 85 draws. These results suggest that the agent may have learned some defensive or passive strategies but was not able to develop actionable gameplay against even basic opponents.

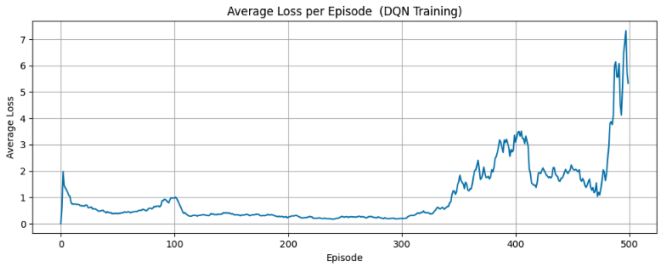

Fig. 2. Training Loss Curve for DQN Agent: TD Loss over training steps.

TABLE I
Gameplay Results Over 100 Games per Matchup

Model	vs. Stockfish (W/D/L)	vs. Random Agent (W/D/L)
Imitation Learning	0 / 0 / 100	7 / 84 / 9
DQN	0 / 0 / 100	8 / 85 / 7

V. Conclusions

This project explored two modern approaches to building chess-playing agents: supervised imitation learning and Deep Q-Network-based reinforcement learning, both using a shared Lc0-style encoding of board states and actions.

The **imitation learning agent** demonstrated clear learning behavior within the training set, with substantial improvements in policy and value predictions. However, the drop in test performance revealed overfitting and poor generalization—likely due to limited data variety. The **DQN agent**, on the other hand, exhibited **no learning progress**, as reflected by a stagnant loss curve and a complete failure in game evaluation, despite architecture parity and replay buffer initialization from expert games.

Critically, **both models failed to win any games against a Stockfish level 1 engine**, highlighting that neither approach reached a baseline level of functional gameplay. These results underscore that while architectural and encoding choices were technically sound, training efficacy was constrained by data scale, compute limitations, and algorithmic challenges.

A. Future Work

Several promising directions can be pursued to address the limitations identified in this project. A primary area for improvement lies in the training data used for the imitation

learning model. Relying on just 607 games from a single expert, even one as strong as Magnus Carlsen, restricts the model's exposure to the breadth of positions and strategies found in real-world chess. Expanding the dataset to include games from a diverse pool of players, across different time controls and play styles, would likely improve the model's ability to generalize and reduce the overfitting observed during testing.

On the reinforcement learning side, the DQN agent struggled to make meaningful learning progress, indicating a need to explore more suitable algorithms. Alternatives such as Proximal Policy Optimization (PPO), Advantage Actor-Critic (A2C), or AlphaZero-style Actor-Critic architectures paired with Monte Carlo Tree Search (MCTS) are known to perform better in environments like chess, which involve large action spaces and sparse reward signals. These methods could help overcome the training instability and exploration challenges that limited DQN's effectiveness in this setting.

In particular, integrating MCTS into the reinforcement learning framework represents a compelling next step. By combining learned policy networks to guide search and value networks to evaluate positions, MCTS can significantly improve decision-making—an approach proven successful by AlphaZero and Lc0. Incorporating this search-based mechanism could elevate both the tactical strength and strategic depth of future agents built on this foundation.

REFERENCES

[1] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, et al., "Mastering the game of Go without human knowledge," *Nature*, vol. 550, no. 7676, pp. 354–359, 2017.

[2] Leela Chess Zero. Lc0 Website. [Online]. Available: https://lczero.org/ (Accessed: Jun. 1, 2025).

[3] Z. Wang, et al., "Learning to Play Chess with Deep Residual Networks," *arXiv preprint arXiv:1810.02102*, 2018.

[4] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, et al., "Playing Atari with Deep Reinforcement Learning," *arXiv preprint arXiv:1312.5602*, 2013.

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, et al., "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.