CAPSTONE PROJECT REPORT

on

DeepChess-RL: Reinforcement Learning in Chess

Submitted by

Ranveer Singh (1032211633)

Aryan Koli (1032210535)

Jayesh Jhawar (1032212570)

Under the Guidance of

Prof. Manisha Kowdiki

School of Electronics & Communication Engineering

Dr. Vishwanath Karad
MIT WORLD PEACE UNIVERSITY, PUNE.
[2024-2025]

SCHOOL OF
ELECTRONICS & COMMUNICATION ENGINEERING

Academic Year 2024- 2024

CERTIFICATE

This is to certify that, Ranveer Singh (1032211633), Aryan Koli (1032210535) and Jayesh Jhawar (1032212570) has successfully completed his/her Capstone Project entitled "**DeepChess-RL: Reinforcement Learning in Chess**" and submitted the same during the academic year 2024-25 towards the partial fulfilment of degree of **Bachelor of Technology in Electronics & Communication Engineering** as per the guidelines prescribed by Dr. Vishwanath Karad MIT World Peace University, Pune.

Prof. Manisha Kowdiki (Project Guide)	Prof. Anjali Ashkedkar (Project Coordinator)	Dr. Parul Jadhav (Head of Department, School of ECE)

Date:

Place: Pune

DECLARATION

I/We the undersigned, declare that the work carried under Capstone Project Phase-II entitled "DeepChess-RL: Reinforcement Learning in Chess" represents my/our idea in my/our own words. I/We have adequately cited and referenced the original sources where other ideas or words have been included. I/We also declare that I/We have adhered to all principles of academic honesty and integrity and have not misprinted or fabricated or falsified any ideas/data/fact/source in my/our submission. I/We understand that any violation of the above will be cause for disciplinary action by the University and can also evoke penal action from the sources which have thus not been properly cited or whose proper permission has not been taken when needed.

Date:

Place:

PRN Number	Name of student	Signature with date
1032210535	Aryan Koli	
1032212570	Jayesh Jhawar	
1032211633	Ranveer Singh	

ACKNOWLEDGEMENT

We would like to extend our heartfelt gratitude to everyone who contributed to the successful completion of our final year project, “Reinforcement Learning in Chess.” This project has been a deeply enriching academic journey, and we are sincerely thankful for the support and guidance we received along the way.

We are especially grateful to our project guide, Manisha Kowdiki, for their invaluable mentorship, constant encouragement, and expert insights. Their guidance was instrumental in shaping the direction of our work and elevating the overall quality of the project.

Our sincere thanks also go to the faculty and staff of the Department of Electrical and Electronics Engineering,Dr. Vishwanath Karad MIT World Peace UniversityDr. Vishwanath Karad MIT World Peace University, for fostering a supportive academic environment and for providing the resources and infrastructure necessary for research and development.

We would also like to express our appreciation to our families and friends for their constant motivation and emotional support throughout this journey.

Lastly, we acknowledge the dedication and teamwork of all project members Ranveer Singh, Aryan Koli, and Jayesh Jhawar. This project would not have been possible without the collective effort, collaboration, and shared vision among us.

Table of Contents

ABSTRACT

The development of chess engines has undergone a remarkable transformation with the integration of artificial intelligence, particularly through advancements in machine learning and deep reinforcement learning. This paper presents a comprehensive review of the evolution of chess engines, tracing the journey from early rule-based systems to the modern, learning-based architectures that define state-of-the-art performance today. We begin by examining traditional approaches such as handcrafted evaluation functions, the Minimax algorithm, and Alpha-Beta pruning, which formed the foundation of classical chess engines.

The paper then explores the paradigm shift introduced by machine learning, with a focus on deep learning models and reinforcement learning techniques. These methods have enabled engines to learn strategic insights directly from gameplay data, reducing reliance on manually encoded knowledge. Pioneering systems like AlphaZero are discussed as key milestones that demonstrated the superiority of learning-based engines in terms of flexibility, adaptability, and performance.

We further analyze how neural networks, Monte Carlo Tree Search (MCTS), and policy-value frameworks have redefined the design and efficiency of modern engines. The challenges associated with training such systems—including reward sparsity, computational demands, and the complexity of game state representation—are also addressed. The paper concludes with a discussion on the future of AI-driven chess engines, highlighting ongoing research directions and the potential for these models to evolve into more general game-playing and decision-making agents.

LIST OF TABLES

LIST OF FIGURES

ABBREVIATION

Abbreviation	Full Form
AI	Artificial Intelligence
RL	Reinforcement Learning
MCTS	Monte Carlo Tree Search
CNN	Convolutional Neural Network
PUCT	Predictor + Upper Confidence Bound for Trees
GUI	Graphical User Interface
CPU	Central Processing Unit
GPU	Graphics Processing Unit
FEN	Forsyth-Edwards Notation
PGN	Portable Game Notation
UCI	Universal Chess Interface
JSON	JavaScript Object Notation
API	Application Programming Interface
SGD	Stochastic Gradient Descent
SGD	Stochastic Gradient Descent
TPU	Tensor Processing Unit

CHAPTER 1
INTRODUCTION

1.1 Motivation/Introduction/Relevance

Chess has long stood as a classic testbed for artificial intelligence due to its perfect information setting, complex decision-making structure, and strategic depth. As a domain, it presents a unique blend of combinatorial complexity and clearly defined rules, making it an ideal environment for testing and advancing computational intelligence. For decades, traditional chess engines have relied on brute-force techniques such as Minimax search and Alpha-Beta pruning, combined with handcrafted evaluation functions that attempt to mimic human judgment. While highly effective, these systems are largely dependent on human expertise and fixed rule sets.

In recent years, the landscape of AI in games has dramatically shifted with the advent of deep learning and reinforcement learning. Algorithms like DeepMind's AlphaZero have demonstrated the potential of systems that can learn purely through self-play, without relying on preprogrammed human knowledge. These breakthroughs not only challenged the status quo of classical engines but also opened up exciting new directions for autonomous learning and strategic decision-making.

This project, DeepChess-RL: Reinforcement Learning in Chess, seeks to explore and harness the power of reinforcement learning to develop a chess engine that improves its gameplay over time through experience. By merging expert guidance (from engines like Stockfish) with self-play training, our system aims to replicate a more human-like approach to learning—where progress is achieved through interaction, adaptation, and iterative feedback. This learning paradigm enables the model to uncover and refine strategies on its own, evolving in a way that static rule-based systems cannot.

The motivation behind this work stems from a desire to bridge the gap between classical AI and modern machine learning approaches in game-playing agents. Developing an engine that learns autonomously provides valuable insights into the broader applications of reinforcement learning—ranging from robotics and autonomous navigation to financial modeling and strategic planning in real-world domains.

Beyond its relevance in the realm of chess, this project underscores the broader applicability of deep reinforcement learning in building intelligent agents capable of complex, long-term

planning in structured environments. It highlights the potential of AI systems to make informed decisions in uncertain scenarios, adapt to new challenges, and operate without explicit human instructions—an essential milestone in the advancement of general-purpose AI.

In essence, this project not only contributes to the growing field of game-playing AI but also serves as a stepping stone toward developing more adaptive and intelligent systems across a variety of disciplines.

1.2 Organization of report

This report is structured to systematically present the development and evaluation of our project titled "DeepChess-RL: Reinforcement Learning in Chess". Each chapter is designed to cover a specific aspect of the project, from initial motivation to final outcomes. The organization is as follows:

- **Chapter 1 – Introduction:** Provides the motivation, relevance, and an overview of the project, highlighting the role of reinforcement learning in modern chess engines.
- **Chapter 2 – Review of Literature:** Discusses existing approaches in chess engine design, including traditional search-based methods and recent advancements using machine learning. It also outlines the aim and objectives of the project.
- **Chapter 3 – System Development:** Describes the conceptual design, system architecture, specifications, and the major challenges encountered during development.
- **Chapter 4 – System Implementation:** Details the implementation methodology, key algorithms, and modules used in building the system, including reinforcement learning loops and game integration.
- **Chapter 5 – Results and Analysis:** Presents the outcomes of training and evaluation, supported by visual data such as performance graphs and screenshots.

- **Chapter 6 – Discussion and Conclusion:** Summarizes the work done, key learnings, challenges overcome, and concludes with the significance of the project and future scope.
- **References:** Lists the academic papers, tools, and other resources referred to during the course of the project.

1.3 Introduction to Chess

Chessboard Setup

At the start of a chess game, the pieces are placed on the board in the following positions:

Figure 1.1: Chessboard at start of the game

From the bottom left as white: rook, knight, bishop, queen, king, bishop, knight, rook. The second row is filled with 8 pawns. The opponent places the same pieces on the other side of the board.

Movement of Chess Pieces

Pawns can normally only move forward one square at a time, but if the pawn is in its initial position, the player can choose to move them two squares forward. To capture pieces, pawns must take diagonals. It can only do this when an opponent's piece is diagonally one square away from the pawn. When a pawn reaches the other side of the board, it can choose to 'promote' itself to a queen, rook, bishop, or knight; i.e., the pawn is replaced by the chosen piece. 'En passant' is a special pawn move that is only possible when the player moves a pawn two squares forward. If the pawn is then adjacent to one of the opponent's pawns, the opponent can choose to capture that pawn as if it had only moved one square forward.

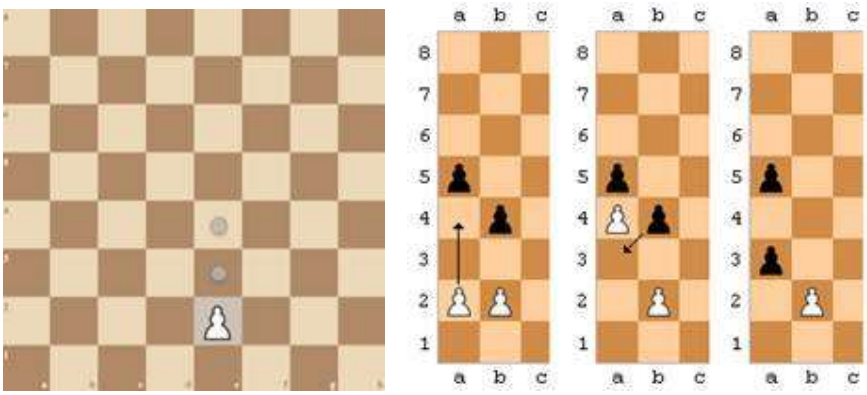

Figure 1.2: Movements of Pawn

The rook can move any number of squares vertically or horizontally, given there are no pieces blocking its way. The bishop can do the same, but diagonally. The queen is a combination of the rook and bishop: it can move vertically, diagonally, and horizontally.

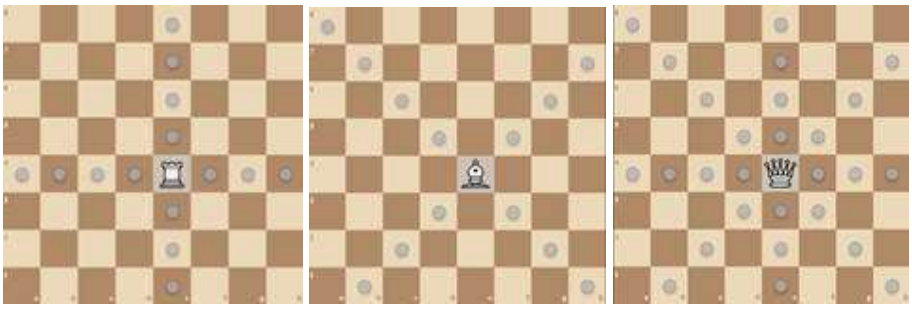

Figure 1.3: Movement of Rook, Bishop and Queen

The knight moves in an L-shape: a knight move consists of moving one square vertically and two horizontally, or vice versa. The knight is the only chess piece that can leap over other pieces. The king can move just like the queen, but only one square at a time.

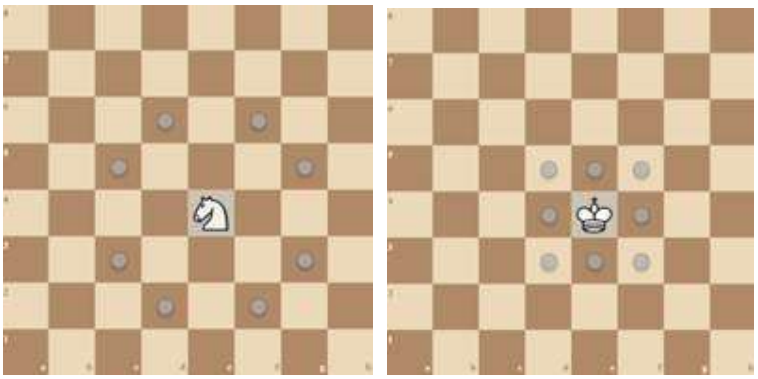

Figure 1.4: Movement of Knight and King

The king can also '*castle*'- special move that gets the king to safety by moving two squares towards one of the rooks and moving the rook to the square that the king has crossed.

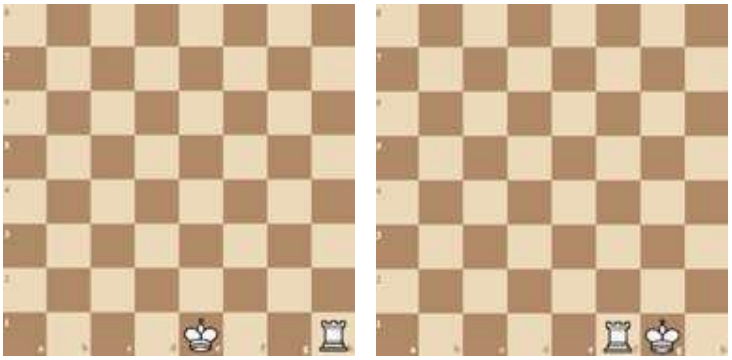

Figure 1.5: Kingside Castling

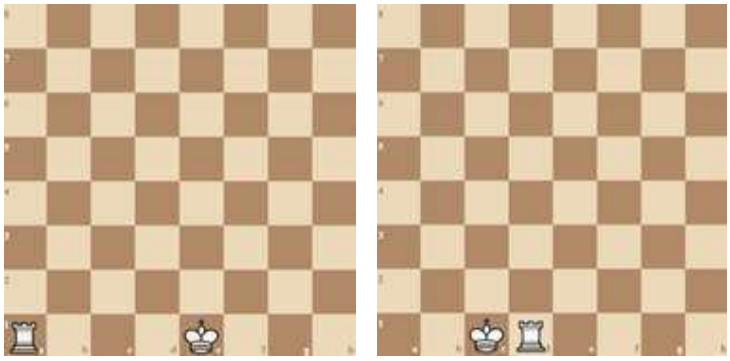

Figure 1.6: Queenside Castling

Check and Checkmate

Attacking the king is called a 'check'. When this happens, the opponent must either move the king out of the way or stop the attack by capturing the attacking piece or blocking the attack with one of its own pieces. When the player cannot stop the attack, the player loses the game. This is called 'checkmate'. The player can never end their move if their king is in check. This would be an 'illegal move'. When it's the player's turn but has no legal moves available, and they are not checked, the game is drawn. This is called 'stalemate'.

CHAPTER 2
REVIEW OF LITERATURE

2.1 LITERATURE REVIEW

Title	Method Used	Proposed Changes/Approach	Remarks
Lai, Matthew. "Giraffe: Using deep reinforcement learning to play chess." *arXiv preprint arXiv:1509.01549* (2015).	Deep reinforcement learning with automatic feature extraction	Utilized self-play to learn evaluation functions without human-crafted features, employing deep neural networks for pattern recognition	First successful end-to-end machine learning chess engine, performing comparably to traditional engines with handcrafted evaluations
Silver, David, et al. "Mastering chess and shogi by self-play with a general reinforcement learning algorithm." *arXiv preprint arXiv:1712.01815* (2017).	Reinforcement learning with self-play and Monte Carlo Tree Search (MCTS)	Developed AlphaZero, starting from random play and learning solely through self-play without domain knowledge, generalizing the approach to multiple games	Achieved superhuman performance in chess, shogi, and Go within 24 hours, surpassing previous world-champion programs

Gottipati, Vijaya Sai Krishna, et al. "Deep Pepper: Expert Iteration based Chess agent in the Reinforcement Learning Setting."	Reinforcement learning with expert iteration and embedded domain knowledge	Combined self-play with embedded chess knowledge to accelerate training compared to tabula rasa approaches like AlphaZero	Demonstrated faster training convergence by integrating domain knowledge, providing insights into efficient learning strategies
Haque, Rejwana, Ting Han Wei, and Martin Müller. "On the road to perfection? evaluating leela chess zero against endgame tablebases." *Advances in Computer Games*. Cham: Springer International Publishing, 2021. 142-152.	Reinforcement learning with self-play using neural networks and distributed computing	Adapted AlphaZero's approach in an open-source project, leveraging distributed volunteer computing for training through self-play	Achieved performance comparable to top engines like Stockfish, showcasing the potential of community-driven RL training

Gundawar, Atharva, Yuchao Li, and Dimitri Bertsekas. "Superior Computer Chess with Model Predictive Control, Reinforcement Learning, and Rollout." *arXiv preprint arXiv:2409.06477* (2024).	Model Predictive Control (MPC), reinforcement learning, and rollout techniques	Introduced a new architecture combining MPC and RL, using existing engines for position evaluation and opponent modeling to enhance move selection	Improved performance of existing engines by adding an additional intelligence layer, demonstrating the effectiveness of combining MPC with RL
Liao, Weidong, and Andrew Moseman. "Developing a Reinforcement Learning based Chess Engine." *Proceedings of the West Virginia Academy of Science* 95.2 (2023).	Reinforcement learning with self-play and convolutional neural networks	Implemented a chess engine trained via self-play without human domain knowledge, focusing on position evaluation using neural networks	Showcased the viability of RL-based chess engines without reliance on human data, contributing to the understanding of self-play training dynamics
Block, Marco, et al. "Using reinforcement learning in chess engines." *Research in Computing Science* 35 (2008): 31-40.	Reinforcement learning with board state classification	Extended KnightCap's learning algorithm by using a comprehensive board state database and optimizing coefficients for each position class individually	Demonstrated notable improvement over engines using only reinforcement learning by integrating board state classification

Hammersborg, Patrik, and Inga Strümke. "Reinforcement learning in an adaptable chess environment for detecting human-understandable concepts." *IFAC-PapersOnLine* 56.2 (2023): 9050-9055	Reinforcement learning with adaptable environment settings	Explored methods for probing which concepts self-learning agents internalize during training, using an adaptable chess environment to assess learning outcomes	Provided insights into the internalization of concepts by self-learning agents, contributing to the understanding of learning processes in complex environments
Young, Lily, and Byeong Kil Lee. "A Reinforcement Learning Approach to Training Chess Engine Neural Networks." *2023 Congress in Computer Science, Computer Engineering, & Applied Computing (CSCE)*. IEEE, 2023.	Reinforcement learning with a relatively shallow neural network	Studied the capability of a neural network with fewer than 20 layers to learn complex decision-making in chess through reinforcement learning	Demonstrated that even relatively shallow neural networks can effectively learn complex decision-making in chess, highlighting the potential for more efficient models

Krishnamurthy, Valli Devi, et al. "Design of a Chess agent using reinforcement learning with SARSA Network." *Artificial Intelligence*. Chapman and Hall/CRC, 2021. 163-170.	Reinforcement learning using SARSA algorithm and Monte Carlo Tree Search	Built a chess agent capable of planning moves and defeating simple opponents by leveraging the SARSA algorithm and Monte Carlo Tree Search for decision-making	Showed the potential of combining SARSA with Monte Carlo Tree Search in developing effective chess agents, contributing to the exploration of alternative RL algorithms in chess engines
Bertram, Timo, Johannes Fürnkranz, and Martin Müller. "Supervised and reinforcement learning from observations in reconnaissance blind chess." *2022 IEEE Conference on Games (CoG)*. IEEE, 2022.	Supervised learning followed by reinforcement learning using Proximal Policy Optimization (PPO)	Adapted a training approach inspired by AlphaGo to play Reconnaissance Blind Chess, an imperfect information game, using observations instead of full game state descriptions	Achieved an ELO of 1330 on the RBC leaderboard without using search, demonstrating the effectiveness of combining supervised and reinforcement learning in imperfect information settings

Lai, Matthew. "Giraffe: Using deep reinforcement learning to play chess." *arXiv preprint arXiv:1509.01549* (2015).	Deep reinforcement learning with Monte Carlo Tree Search	Created an engine capable of playing eight different chess variants by training a network on human games and evaluating policy accuracy across variants	Demonstrated the viability of a single engine handling multiple chess variants, contributing to the understanding of generalization in reinforcement learning models
Buchner, Johannes. "Rotated Bitboards in FUSc# and Reinforcement Learning in Computer Chess and Beyond." *arXiv preprint arXiv:2503.10822* (2025).	Reinforcement learning with rotated bitboard representation	Discussed the implementation of rotated bitboards in move generation and explored the potential of reinforcement learning in improving sample efficiency in chess engines	Highlighted the importance of efficient data representation and sample efficiency in reinforcement learning, providing insights applicable beyond chess
Patankar, Shreya, et al. "A Survey of Deep Reinforcement Learning in Game Playing." *2024 MIT Art, Design and Technology School of Computing International*	Deep Q-Learning	Explored the application of Deep Q-Learning in teaching chess engines optimal strategies	Performs basic tactics; limited by sparse rewards and complexity needs better exploration and state handling.

Conference (MITADTSoCiCon). IEEE, 2024.			

2.2 AIM AND OBJECTIVES OF PROJECT

This project aims to design and develop a *reinforcement learning-based chess engine*, titled *DeepChess-RL*, which demonstrates continual improvement in gameplay through self-play and expert-guided training. The primary focus is not on outperforming state-of-the-art engines such as Stockfish or Leela Chess Zero, but rather on understanding and experimenting with the *learning dynamics* of artificial intelligence in a complex, rule-based environment like chess.

The scope of the project includes both theoretical exploration and practical implementation of AI techniques, integrating neural networks, Monte Carlo Tree Search (MCTS), and domain-specific heuristics. It provides a testbed for experimentation in *machine learning, strategic reasoning, and adaptive decision-making*.

The key objectives of this project are as follows:

- **Learning Efficiency:** Optimize the reinforcement learning pipeline to enable the engine to achieve competent performance using limited computational resources. This involves efficient data generation, experience replay, and network updates.

- **Interpretability***:* Develop tools and methods to visualize and analyze the engine's decision-making process. This includes tracking value estimates, policy distributions, and evaluating the influence of keyboard features.

- **Adaptability:** Ensure the engine can adjust its strategy based on opponent strength, training goals, or board context. This allows it to simulate human-like learning patterns and diversify its gameplay style.

- **Hybrid Architecture:** Combine deep reinforcement learning with classical search techniques such as alpha-beta pruning or MCTS to benefit from the tactical precision of traditional engines and the strategic foresight of learning-based systems.

- **Modular and Extendable Design:** Build a flexible codebase that supports easy experimentation, enabling integration of new training objectives, evaluation metrics, or alternative learning algorithms.

- **Educational Value:** Create a framework that is understandable and usable by students and researchers interested in AI, offering insights into the inner workings of learning-based chess systems.

By achieving these goals, the project not only deepens understanding of reinforcement learning applications in chess, but also contributes valuable tools and methodologies to the broader field of game-playing AI.

CHAPTER 3

SYSTEM DEVELOPMENT

3.1 SYSTEM BLOCK DIAGRAM

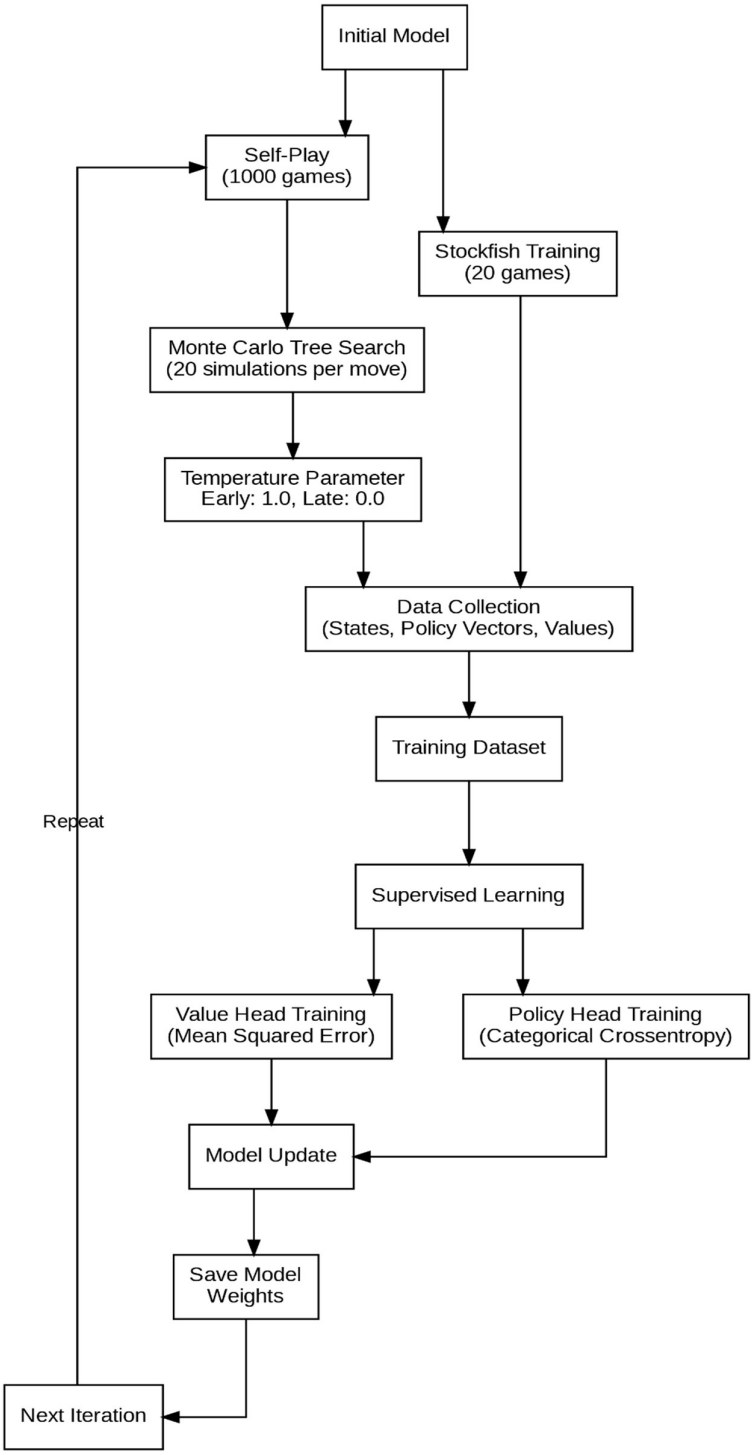

Figure 3.1: Training Process of DeepChess-RL

This flowchart illustrates the training pipeline of the DeepChess-RL engine. The process begins with an initial model and includes two key phases: self-play training and Stockfish-guided training. Each move is selected using Monte Carlo Tree Search (MCTS) with 20 simulations per move, and exploration is controlled via a temperature parameter. Gameplay data (states, policy vectors, and value estimates) is collected and compiled into a training dataset. The model is updated through supervised learning, where the policy head is trained using categorical cross entropy loss and the value head with mean squared error. The updated weights are saved, and the process is repeated for further improvement in subsequent iterations.

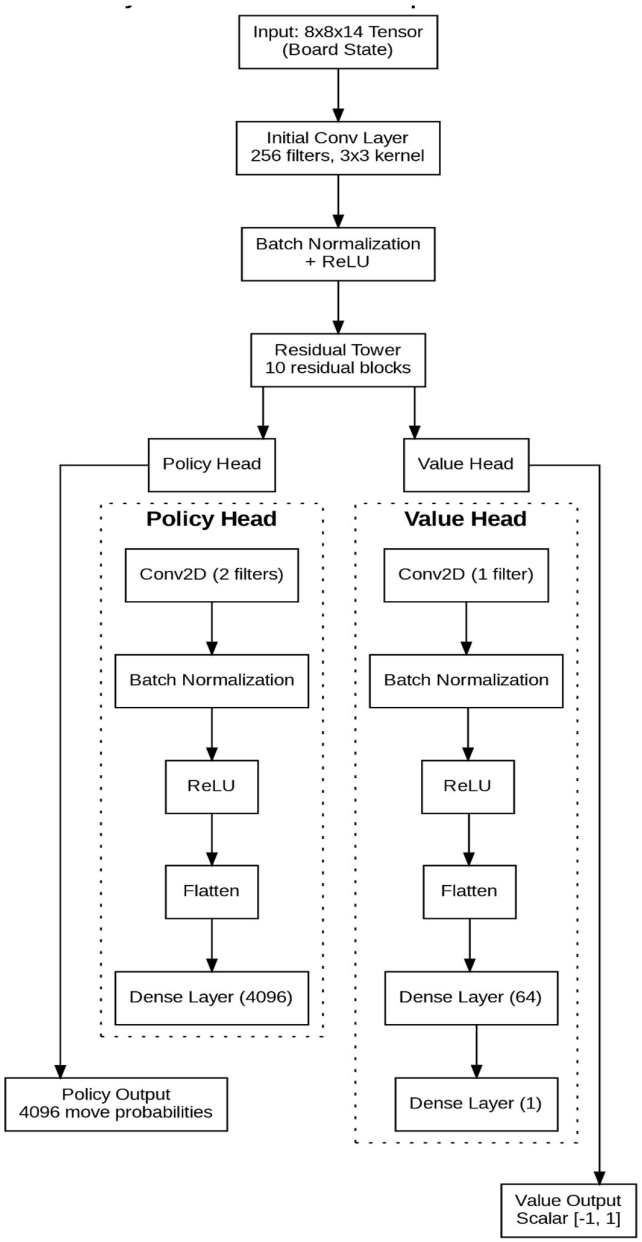

Figure 3.2: System Architecture of DeepChess-RL

This figure presents the neural network architecture used in DeepChess-RL. The model receives an 8×8×14 tensor representing the chess board state as input. An initial convolutional layer with 256 filters processes the input, followed by batch normalization and ReLU activation. The core of the model is a residual tower comprising 10 residual blocks, enabling deeper learning through skip connections. The output is split into two heads: the Policy Head, which predicts a probability distribution over 4096 possible moves, and the Value Head, which estimates the position's value in the range [-1, 1]. Each head consists of convolutional, normalization, activation, and dense layers tailored to their respective tasks.

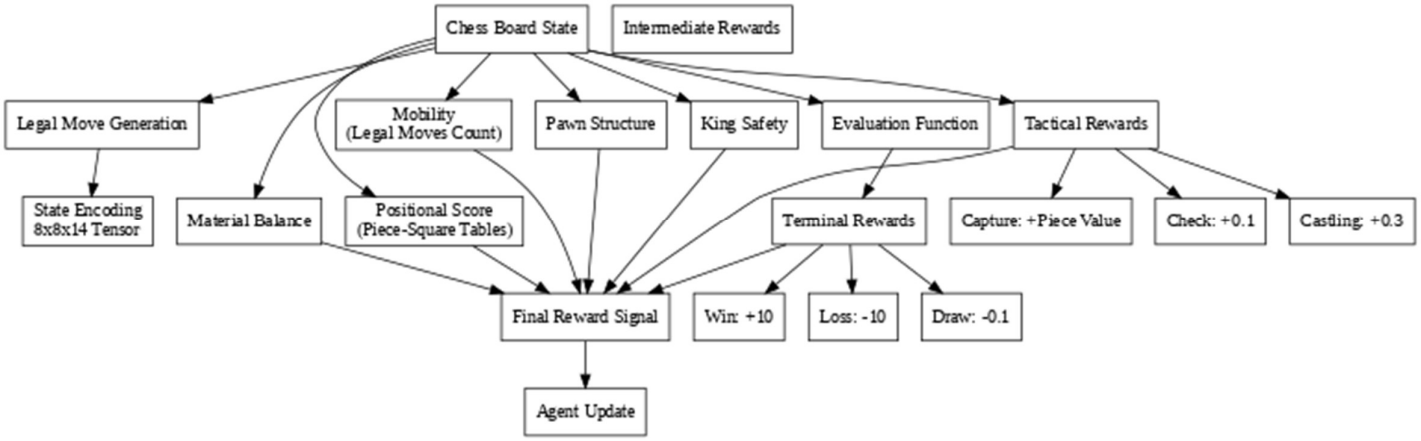

Figure 3.3: Environment and Reward System in DeepChess-RL

This diagram illustrates the reward mechanism within the DeepChess-RL environment. The system evaluates board states using a combination of terminal rewards (win, loss, draw) and intermediate rewards based on strategic factors such as material balance, positional score from piece-square tables, pawn structure, king safety, and mobility. Tactical rewards provide additional feedback for actions like checks, castling, and capturing pieces. These evaluations collectively contribute to a final reward signal that updates the agent, promoting learning through both immediate outcomes and long-term strategy reinforcement.

3.2 SYSTEM SPECIFICATIONS

The DeepChess-RL project was developed and tested on a system with modest hardware resources, utilizing CPU-only training and inference. This approach ensures the model can be executed on commonly available machines without requiring GPU acceleration.

3.2.1 Hardware Requirements

Component	Specification
Processor (CPU)	Intel Core i5 / AMD Ryzen 5 or higher
Graphics Processing	Integrated Graphics (no CUDA/GPU required)
Memory (RAM)	8 GB or higher
Storage	50 GB HDD/SSD (minimum)
Display	1080p monitor (for GUI and visual debugging)

Table 3.1: Hardware Requirements

Note: While GPU support (e.g., NVIDIA CUDA) can accelerate training, the current implementation was tested and optimized for CPU environments to ensure broader accessibility.

3.2.2 Software Requirements

Software	Version / Description
Operating System	Windows 10 / Ubuntu 20.04 / macOS Ventura
Python	3.10+
TensorFlow (CPU)	2.11+ (installed without GPU/CUDA support)
Python-Chess	1.999+ (for board logic and move generation)
Tkinter / Pygame	For graphical user interface
Git	Version control and collaborative tracking

Table 3.2: Software Requirements

3.2.3 Libraries and Tools

Library	Purpose
tensorflow	Model building and training (neural network architecture)
numpy	Numerical operations and tensor manipulation

chess / python-chess	Chess logic, move legality, and PGN/FEN handling
pillow	Image processing and handling (used for board rendering)
cairosvg	SVG to PNG conversion (for rendering chess pieces/boards)
tqdm	Progress bars during training and evaluation

Table 3.3: Libraries and Tools Required

3.2.4 Platform and Portability

- All components run efficiently on CPU-only machines, including laptops.
- The system is compatible with Google Colab and Jupyter Notebooks for cloud-based development and experimentation.
- Docker was not required for the base implementation, though it can be integrated for future scalability.

3.3 CHALLENGES/COMPLEXITIES FACED

During the development and experimentation of the DeepChess-RL system, multiple challenges and complexities were encountered, especially due to the nature of reinforcement learning in the strategic and high-branching domain of chess. Below are the key difficulties faced:

1. Sparse and Delayed Rewards

- **Problem**: In standard chess games, the final outcome (win/loss/draw) is only determined at the end of a long sequence of moves.
- **Impact**: It made reinforcement learning less efficient, as the agent received feedback too late to correlate it with earlier decisions.
- **Solution**: Introduced **intermediate rewards** based on material balance, checks, castling, mobility, and positional evaluations using piece-square tables.

2. State Representation Complexity

- **Problem**: Encoding the chess board with all relevant features (e.g., piece positions, turn, castling rights, move history) into a suitable neural network input format was non-trivial.
- **Solution**: Used a custom 8x8x14 tensor representation to encode piece types, player turns, and special statuses like castling or en passant.

3. Legal Move Generation and Rule Handling

- **Problem**: Implementing and verifying move legality, special rules (castling, en passant, threefold repetition), and check/checkmate detection required precise handling.
- **Solution**: Utilized the python-chess library to ensure complete FIDE rule compliance.

4. Model Training Without GPU

- **Problem**: Training neural networks and running MCTS (Monte Carlo Tree Search) was computationally expensive.
- **Limitation**: The system was designed and tested without CUDA or GPU acceleration.
- **Solution**: Optimized the training loop, used smaller models, and performed testing in Google Colab environments to utilize free compute.

5. Integration of MCTS with Policy-Value Network

- **Problem**: Coordinating the output of the neural network with MCTS to guide self-play

and evaluation required balancing exploration vs. exploitation.

- **Solution**: Implemented a PUCT (Predictor + UCT**)** formula and adjusted parameters to stabilize MCTS behavior.

6. Evaluation Instability

- **Problem**: It was difficult to measure performance consistently due to randomness in self-play outcomes and model variance.
- **Solution**: Built an evaluation pipeline using fixed seeds, consistent test games, and graphs to track win rates against Stockfish and previous agents.

7. GUI Rendering and Visualization

- **Problem**: Generating real-time, interpretable board visuals from training or evaluation games with low-latency was tricky.
- **Solution**: Combined pillow and cairosvg to render high-quality boards and piece positions within the Tkinter GUI.

CHAPTER 4

SYSTEM IMPLEMENTATION

4.1 SYSTEM DESIGN AND DESCRIPTION

The implementation of **DeepChess-RL** is a full-stack, modular deep reinforcement learning system developed in Python. It brings together modern machine learning practices and classical chess principles to train a chess engine that learns through self-play and expert interaction. The project emphasizes interpretability, flexibility, and learning efficiency while maintaining compatibility with CPU-based environments.

Core Architecture

The system follows a modular structure with distinct components, each handling a specific function in the reinforcement learning pipeline:

- **Neural Network Model (model.py)**: The model implements a dual-headed residual network architecture, inspired by AlphaZero. It processes the board state as an 8×8×14 tensor (representing piece positions, side to move, and repetition history) and outputs:
    a. A policy head with 4096 logits (64×64) indicating move probabilities.
    b. A value head predicting the positional outcome in the range [-1, 1].

- The network includes 10 residual blocks with 256 filters per layer, using TensorFlow as the primary deep learning framework. The training configuration includes a batch size of 256, learning rate of 0.001, and 5 epochs per iteration, with a 20% validation split.

- **Monte Carlo Tree Search (mcts.py)**: MCTS serves as the action selection algorithm during both training and inference. It is enhanced with:
    a. PUCT (Predictor + UCT) algorithm using an exploration constant (C_PUCT = 4.0).
    b. Dirichlet noise ($\alpha = 0.3$, $\varepsilon = 0.25$) for enhanced exploration during self-play.
    c. Integration with the policy and value outputs to guide search efficiency.

- **Environment (environment.py)**: The chess environment handles board logic and reward assignment. Built using the python-chess library, it enforces FIDE rules and tracks game states. The reward system combines terminal rewards (+10 for win, -10 for loss) with strategic intermediate rewards for:
    a. Captures (scaled by piece value),
    b. Castling (+0.3),
    c. Checks (+0.1),
    d. Center control and mobility,
    e. Pawn structure and king safety, based on piece-square tables and positional heuristics.

Training Pipeline

- **Self-Play Training (training.py):** The model trains by playing against itself, generating supervised examples from MCTS-guided games. Each iteration consists of:
    a. 5 full self-play games.
    b. 20 MCTS simulations per move.
    c. Temperature-based move selection (1.0 in early game, 0.0 in late game) for balancing exploration and exploitation.

- **Stockfish-Guided Training:** The system optionally trains against Stockfish (Skill Level 15) for expert guidance. Matches are played with a 0.1-second time control per move, and training data is collected from 20 such games per session.

Visualization and GUI (gui.py)

A user-friendly graphical interface is implemented using Tkinter to visualize:

- Live board states with highlighted moves.
- Model behavior during training and evaluation.
- Real-time interaction, including manual play against AI or Stockfish.
- Controls to switch between modes (self-play, Stockfish training, human vs AI).

The board is rendered using SVG to PNG conversion via cairosvg and pillow, ensuring high-quality visuals.

Data Management

The system uses a clean directory structure:

- models/: stores network checkpoints with versioned weights.
- memory/: stores training data in npz file format self-play and Stockfish sessions.
- plots/: includes generated visualizations and analysis charts.

Checkpointing is automatic after each training iteration, allowing seamless recovery or model comparison.

Evaluation Tools

Dedicated scripts like evaluate.py, analyze.py, and visualize.py provide:

- Win/loss/draw tracking against baseline engines.
- Average move counts, reward curves, and MCTS search statistics.
- Graphs for loss convergence and value estimation accuracy.

System Orchestration

All components are orchestrated through main.py, which acts as the central control point for launching training, evaluation, or gameplay. Configuration parameters such as simulation count, learning rates, and model paths are centralized in config.py, enabling fast experimentation and reproducibility.

4.2 Flow chart/ Algorithm implemented

Project workflow

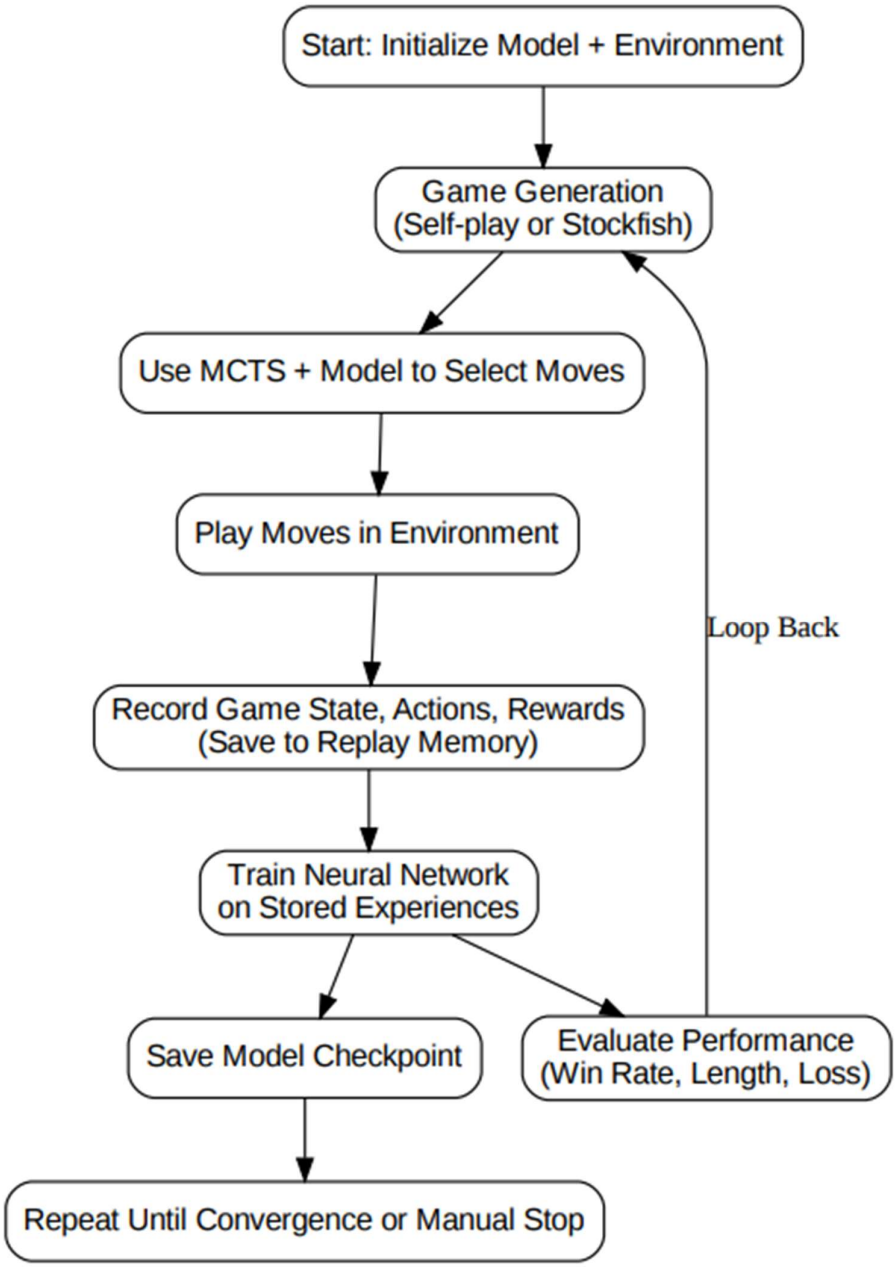

Figure 4.1: Project Workflow

Code Architecture

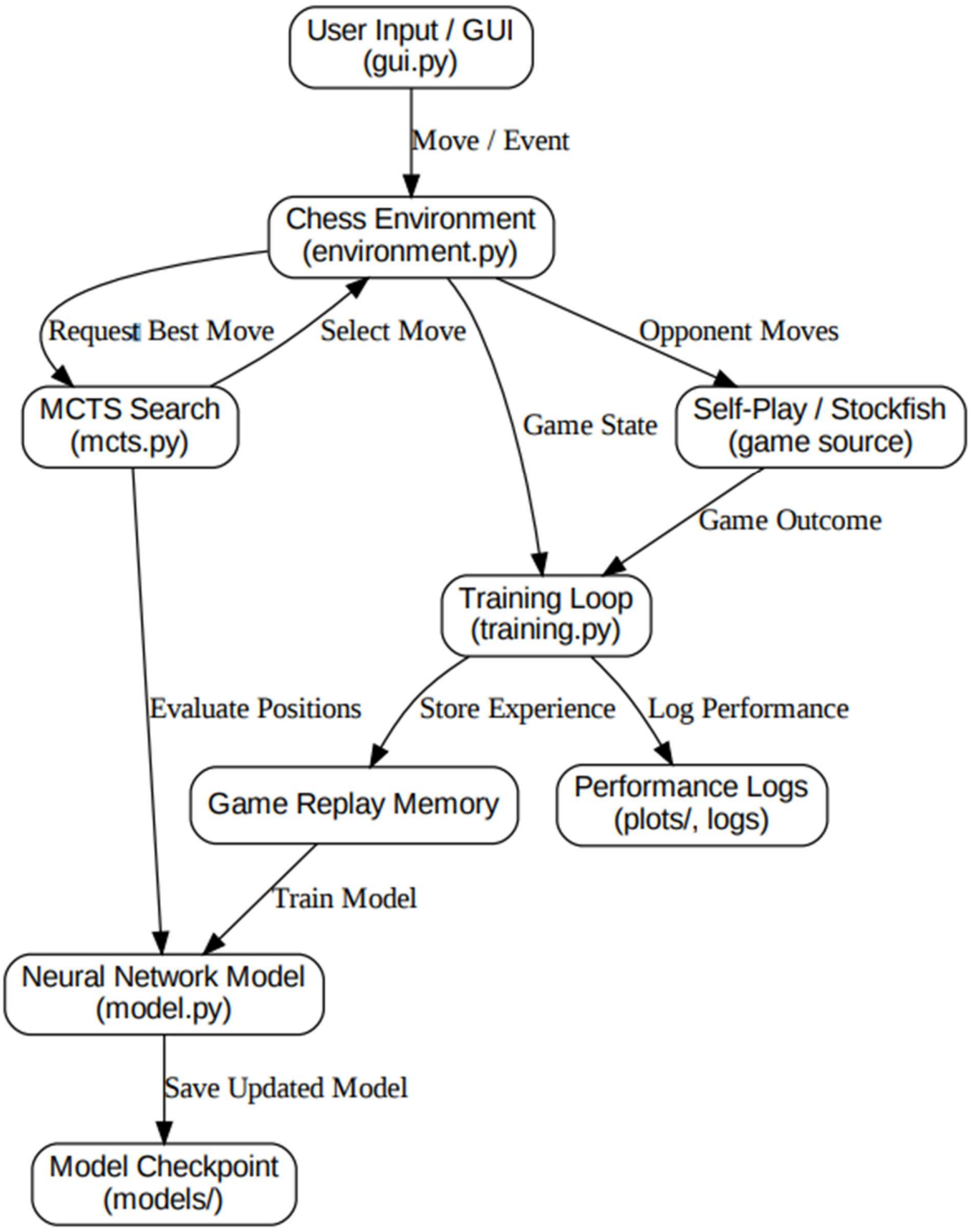

Figure 4.2: Code Architecture

CHAPTER 5

RESULTS AND ANALYSIS

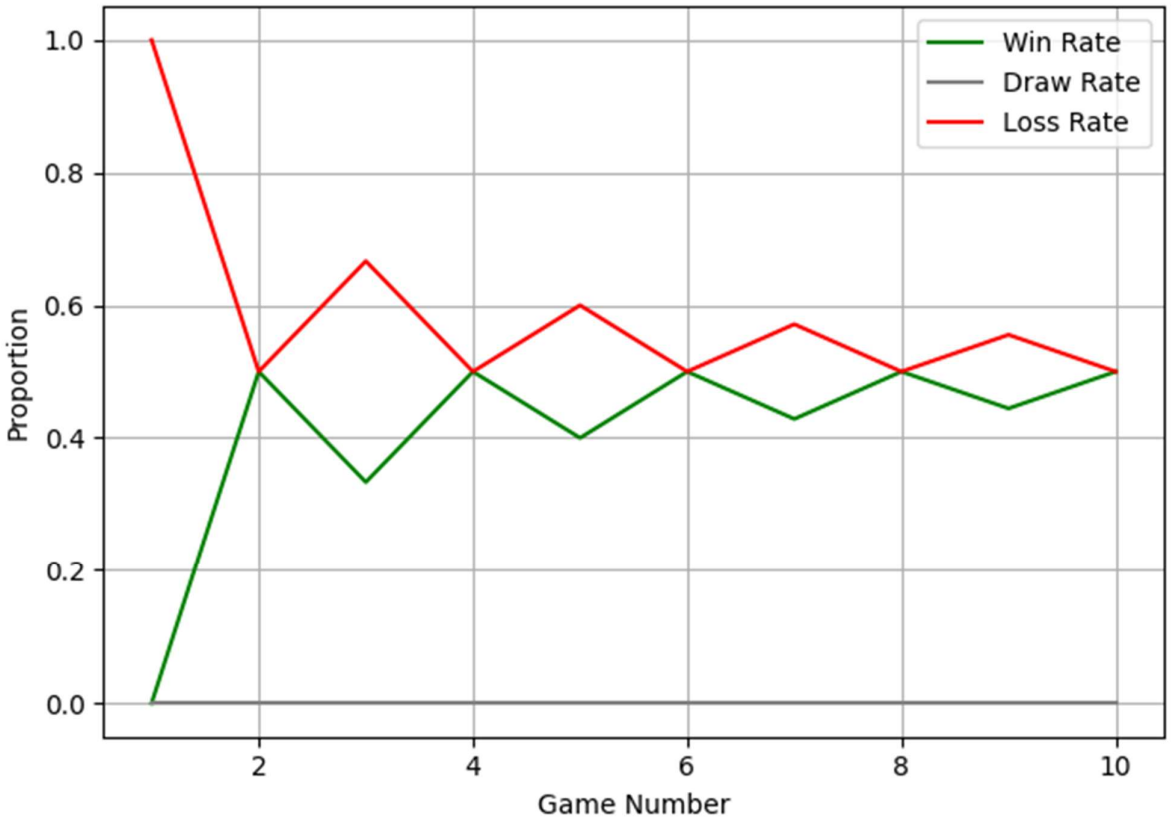

Figure 5.1 : Model vs Stockfish – Performance Over Time

This graph visualizes the performance of the DeepChess-RL model over a series of 10 games against Stockfish (Skill Level 15). The win rate (green line), draw rate (gray line), and loss rate (red line) are plotted as proportions for each game. Initially, the model suffered heavy losses, but as training progressed, the winn rate improved and began to stabilize, suggesting that the model was gradually learning stronger strategies. Loss rates decreased and fluctuated in later games, showing that the model occasionally matched or exceeded Stockfish in performance. The draw rate remained low, implying that most games had decisive outcomes.

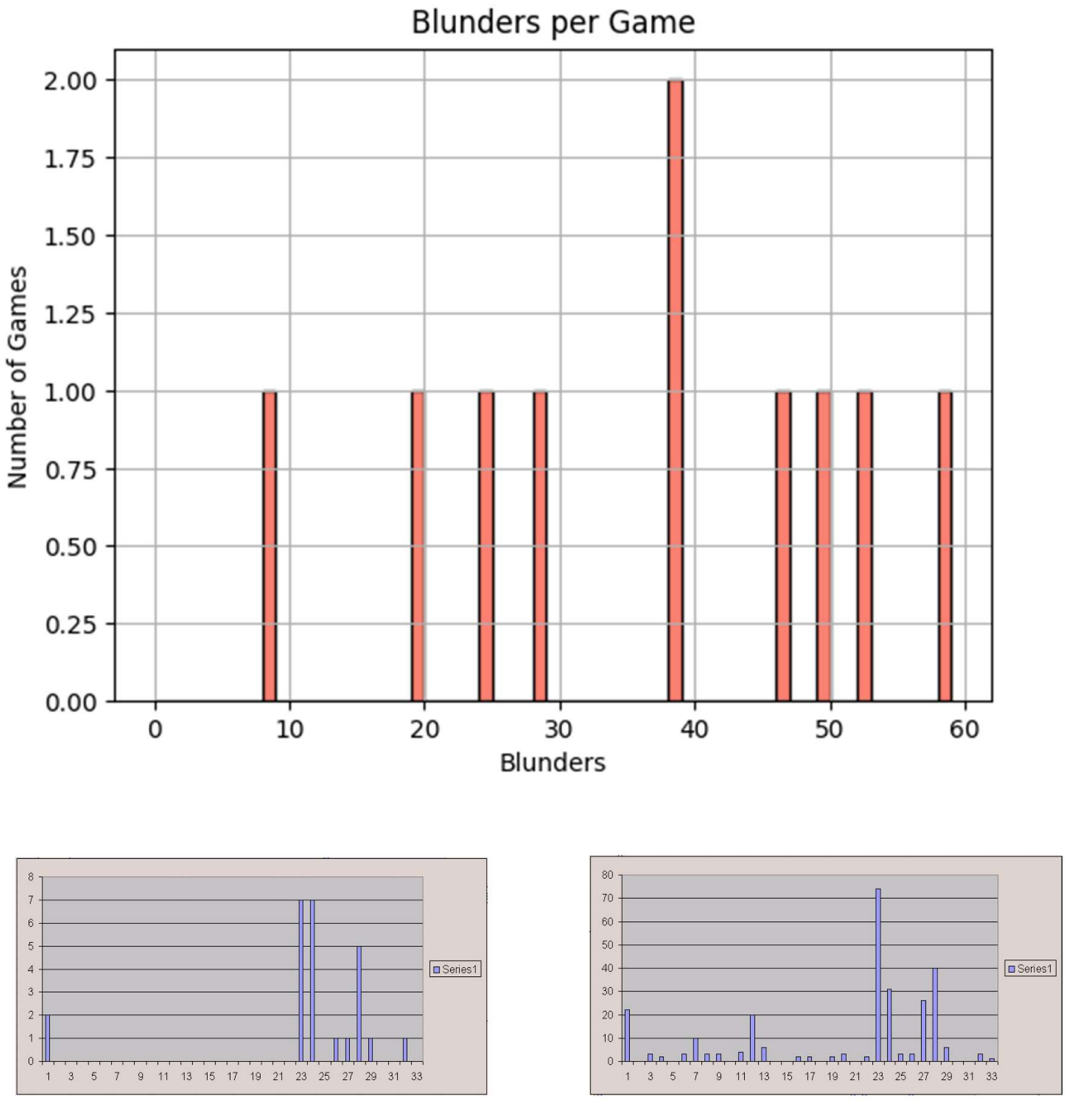

Figure 5.2 : Model's Blunders per Game

These histograms display the number of blunders committed by the model across various games. Blunders are defined as high-value mistakes (as per Stockfish evaluation). Most games recorded a noticeable number of blunders, with counts ranging between 10 and 60. The wide distribution highlights inconsistencies in decision-making, likely due to limited training iterations. However, the downward trend in later games (not always peaking) hints at ongoing improvement. Reducing the blunder count further is a key area for future model refinement.

Visual Training Iteration Output:

Figure 5.3 : Model Output

Figure: GUI of RL Chess Application

This figure showcases the graphical user interface (GUI) of the RL Chess system developed using Tkinter. The main board is rendered using SVG graphics via cairosvg and python-chess, and supports interactive play with both white and black pieces.

The interface includes several key control buttons:

- **New Game**: Resets the board and starts a fresh game.
- **Train Model**: Initiates training using the latest model parameters.
- **Play as White / Play as Black**: Allows a human player to play against the trained model from either side.
- **Self-Play Training**: Triggers automated training using self-play games, enabling reinforcement learning without external guidance.
- **Train vs Stockfish**: Starts training sessions where the agent learns by playing against the Stockfish engine.
- **Stop Training**: Terminates ongoing training processes safely.
- **Play vs Stockfish**: Launches a match where the user or model can test its strength against the Stockfish engine.

At the bottom, a status bar displays messages such as "Ready to play" to keep the user informed of the current state. This GUI provides a complete, interactive environment for both playing and training the RL model in a user-friendly manner.

CHAPTER 6

DISCUSSION AND CONCLUSION

6.1 Discussion

The development of DeepChess-RL has been a comprehensive exploration of reinforcement learning applied to a complex, strategic environment like chess. The project aimed to implement a learning-based chess engine capable of improving its gameplay through both self-play and expert-guided training. Throughout the course of development, several key components were successfully designed, implemented, and evaluated, including a dual-headed neural network, a Monte Carlo Tree Search (MCTS) engine, a dynamic reward-based environment, and a graphical user interface for real-time interaction and visualization.

One of the major highlights of the project was the creation of a custom reward function that addressed the sparse feedback challenge inherent in chess. By incorporating intermediate rewards—such as capturing valuable pieces, castling, center control, and king safety—the model received more informative signals, accelerating the learning process. This dynamic reward system proved crucial in making reinforcement learning more efficient in such a long-horizon, strategic domain.

The training pipeline was successfully implemented with flexibility for both self-play and Stockfish-guided training, allowing the engine to learn from experience as well as expert play. Although the training was performed on a CPU-based system without GPU acceleration, the model showed consistent improvement over iterations. The architecture's modularity enabled easy debugging and experimentation with different configurations such as learning rates, simulation counts, and exploration strategies.

The Tkinter-based GUI served not only as a front-end for gameplay but also as a valuable debugging and demonstration tool. It helped visualize how the model evolved during training and allowed interactive testing against both humans and Stockfish.

From a practical standpoint, the project demonstrated the feasibility of creating a reinforcement learning-based chess engine without requiring large-scale resources. It also underscored the

importance of designing interpretable, modular systems when working with AI models in complex environments.

While the engine does not yet rival top-tier engines like Stockfish or Leela Chess Zero, the project successfully meets its core objectives: to build a learning system that improves over time, to explore the interaction between MCTS and neural networks, and to provide a hands-on platform for understanding the principles of game-playing AI.

6.2 Conclusion

The DeepChess-RL project marks the successful implementation of a reinforcement learning-based chess engine that combines traditional game theory with modern deep learning techniques. By leveraging self-play, expert guidance from Stockfish, and a modular AI architecture, the system demonstrates how strategic decision-making can be learned autonomously in a complex domain like chess.

The project achieved its core objectives, including the design of a dual-headed neural network, integration of Monte Carlo Tree Search for move selection, and the development of a dynamic reward system to handle sparse feedback. The chess environment was enhanced with custom heuristics, and training was made efficient and interpretable—even without access to GPU acceleration. The inclusion of a real-time GUI further added value by allowing users to interact with and observe the model's behavior during training and gameplay.

Although the model does not currently match the performance of top-tier chess engines, it lays a strong foundation for future research and development in learning-based game agents. Potential extensions include GPU-accelerated training, integration of an opening book, endgame tablebases, distributed self-play, and more advanced evaluation networks.

Overall, DeepChess-RL serves as a proof-of-concept for building intelligent, adaptive systems in rule-based environments using reinforcement learning. It not only contributes to the academic understanding of AI in games but also offers a practical framework for further exploration in machine learning, game theory, and autonomous agents.

6.3 Future Scope

The development of DeepChess-RL successfully addressed several key challenges of applying reinforcement learning in complex, rule-intensive environments like chess. However, there remains significant potential to expand, refine, and elevate the capabilities of the system in future iterations.

One critical enhancement would be the introduction of GPU-based training. While the current system operates entirely on CPU, integrating CUDA-compatible GPUs or leveraging TPUs through cloud platforms like Google Colab or AWS would drastically reduce training time and allow for deeper neural architectures or larger training datasets.

The system's self-play engine could also be extended with distributed training support, allowing multiple games to be generated in parallel across machines or cores. This would accelerate data collection and enable more robust training cycles. Combining this with a replay buffer prioritization strategy—where important or high-information games are sampled more frequently—could improve the efficiency and stability of learning.

Another promising area for enhancement is the use of opening book integration. Although DeepChess-RL learns opening strategies through self-play, incorporating a pre-built opening book could guide the model through strategically balanced openings and accelerate early-game learning. Similarly, integrating endgame tablebases would improve precision in late-game play and help the model learn optimal play in simplified board states.

The reward system can also be made more sophisticated by incorporating adaptive weighting. As the model becomes stronger, intermediate rewards could dynamically reduce in influence, allowing the agent to focus more on long-term planning and final outcomes—thus mimicking human learning progression.

From a gameplay perspective, the GUI could be expanded with real-time analytics, such as move prediction heatmaps, policy confidence scores, and evaluation graphs. These features would improve interpretability and provide educational value for human users.

Finally, future iterations could explore transfer learning, where models trained in DeepChess-RL are adapted to similar board games or rule variants (e.g., Chess960, mini-chess). This would

test the generalizability of the model and reinforce its design as a flexible reinforcement learning framework.

Collectively, these future directions aim to move DeepChess-RL toward becoming a more powerful, scalable, and adaptable learning system—not just for chess, but as a foundational platform for broader AI research in strategic environments.

6.4 Individual Contributions

The following points outline the contributions of each team member toward the successful completion of the *DeepChess-RL* project:

- **Ranveer Singh**

    a. Designed and implemented the core reinforcement learning engine.
    b. Developed the neural network model architecture with dual-headed outputs.
    c. Integrated Monte Carlo Tree Search (MCTS) with policy and value predictions.
    d. Built the training pipeline and configured self-play and Stockfish training modes.
    e. Designed and implemented the GUI using Tkinter for interactive visualization.
    f. Handled performance evaluation, visualization scripts, and reward system design.

- **Jayesh Jhawar**

    a. Contributed to the literature review and background research for the project and focused on system documentation and theoretical framework structuring
    b. Helped prepare and organize training data from self-play and Stockfish sessions.
    c. Assisted with testing the environment and model modules during development.
    d. Supported the GUI design and participated in manual gameplay testing.

- **Aryan Koli**

    a. Focused on system documentation and theoretical framework structuring.
    b. Assisted in designing the reward heuristics and validating model output.
    c. Helped debug the training pipeline and configure model parameters.
    d. Worked on organizing references, citations, and formatting the final report.

This division of work enabled efficient collaboration and ensured that each member made meaningful contributions to both the technical and academic aspects of the project.

REFERENCES

[1] Turing, A. M. (1950). Computing machinery and intelligence. *Mind*, 59(236), 433-460.

[2]Hsu, F. H. (2022). *Behind Deep Blue: Building the computer that defeated the world chess champion*. Princeton University Press.

[3] Heath, D., Allum, D., & Square, P. (1997). The Historical Development of Computer Chess and its Impact on Artificial Intelligence. *Deep Blue Versus Kasparov: The Significance for Artificial Intelligence*, *63*.

[4] Gibbs, M. (2014). Stockfish: The strongest chess engine. *ChessBase Magazine*, 2014(2), 12-15.

[5] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., ... & Hassabis, D. (2017). Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR*, abs/1712.01815.

[6] Barnes, D. J., & Hernandez-Castro, J. (2014). On the limits of engine analysis for cheating detection in chess. *Computers & Security*, 48, 58-73.

[7]. *Chess,* Wikipedia, [Online]. Available: https://en.wikipedia.org/w/index.php?titl e=Chess&oldid=1085844722

[8]. Sridhar, A., Pesala, B., Radhakrishnan, G., Niezgoda, J., & Gopalakrishnan, S. (2025). Evolution of Artificial Intelligence. In *Artificial Intelligence and Biological Sciences* (pp. 26-37). CRC Press.

[9]. Gibbs, M. (2014). Stockfish: The strongest chess engine. *ChessBase Magazine*, 2014(2), 12-15.

[10]. *AlphaZero: Shedding new light on chess, shogi, and Go* [Online]. Available: https://deepmind.google/discover/blog/al phazero-shedding-new-light-on-chess- shogi-and-go/

[11]. Sadmine, Q. A., Husna, A., & Müller, M. (2023). Stockfish or Leela Chess Zero? a comparison against endgame tablebases. In *Advances in Computer Games* (pp. 26- 35). Cham: Springer Nature Switzerland.

[12]. Chaslot, G. M. J. B. C. (2010). Monte-

carlo tree search.

[13]. Heath, D., Allum, D., & Square, P. (1997). The Historical Development of Computer Chess and its Impact on Artificial Intelligence. *Deep Blue Versus Kasparov:*

The Significance for Artificial Intelligence, *63*.

[14] "*Minimax*" Wikipedia, Mar. 2022 . Available: https://en.wikipedia.org/wiki/Minimax

[15] Everitt, T., & Hutter, M. (2015). A topological approach to meta-heuristics: analytical results on the BFS vs. DFS algorithm selection problem. arXiv preprint arXiv:1509.02709.

[16] *"How does Chess AI work?",* [Online], *Medium,* Available: https://medium.com/@santiagu.gap/how-does-chess-ai-work-73993a22d5c3

[17] "*Alpha–beta pruning*," Wikipedia, Jan. 2022. Available: https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning

[18] Knuth, D. E., & Moore, R. W. (1975). An analysis of alpha-beta pruning. *Artificial intelligence*, *6*(4), 293-326.

[19] *Minimax and Monte Carlo Tree Search* - Philipp Muens.

[20] Świechowski, M., Godlewski, K., Sawicki, B., & Mańdziuk, J. (2023). Monte Carlo tree search: A review of recent modifications and applications. *Artificial Intelligence Review*, *56*(3), 2497-2562.

[21] Chaslot, G. M. J. B. C. (2010). Monte-carlo tree search.

[22] Tomašev, N., Paquet, U., Hassabis, D., & Kramnik, V. (2020). Assessing game balance with AlphaZero: Exploring alternative rule sets in chess. *arXiv preprint arXiv:2009.04374*.

[23] Sadmine, Q. A., Husna, A., & Müller, M. (2023). Stockfish or Leela Chess Zero? a comparison against endgame tablebases. In *Advances in Computer Games* (pp. 26-35). Cham: Springer Nature Switzerland.

[24] Block, M., Bader, M., Tapia, E., Ramírez, M., Gunnarsson, K., Cuevas, E., ... & Rojas, R. (2008). Using reinforcement learning in chess engines. *Research in Computing Science*, *35*, 31-40.

[25] Sutton, R. S., & Barto, A. G. (1998). *Reinforcement learning: An introduction* (Vol. 1, No. 1, pp. 9-11). Cambridge: MIT press.

[26] Joerg, C., & Kuszmaul, B. C. (1994). Massively parallel chess. *Proceedings of the Third DIMACS Parallel Implementation Challenge*, 17-19.

[27] McIlroy-Young, R., Sen, S., Kleinberg, J., & Anderson, A. (2020, August). Aligning superhuman ai with human behavior: Chess as a model system. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (pp. 1677-1687).

[28] Chitale, A. M., Cherian, A. M., & Singh, A. (2024, July). Implementing the Chess Engine using NNUE with Nega-Max Algorithm. In 2024 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT) (pp. 1-6). IEEE.

[29]. *Top Chess Engine Championship*.

[30]. *Lichess,* [Online], Available: https://lichess.org/

[31] Lai, Matthew. "Giraffe: Using deep reinforcement learning to play chess." *arXiv preprint arXiv:1509.01549* (2015).

[32] Silver, David, et al. "Mastering chess and shogi by self-play with a general reinforcement learning algorithm." *arXiv preprint arXiv:1712.01815* (2017).

[33] Gottipati, Vijaya Sai Krishna, et al. "Deep Pepper: Expert Iteration based Chess agent in the Reinforcement Learning Setting."

[34] Haque, Rejwana, Ting Han Wei, and Martin Müller. "On the road to perfection? evaluating leela chess zero against endgame tablebases." *Advances in Computer Games*. Cham: Springer International Publishing, 2021. 142-152.

[35] Gundawar, Atharva, Yuchao Li, and Dimitri Bertsekas. "Superior Computer Chess with Model Predictive Control, Reinforcement Learning, and Rollout." *arXiv preprint arXiv:2409.06477* (2024).

[36] Liao, Weidong, and Andrew Moseman. "Developing a Reinforcement Learning based Chess Engine." *Proceedings of the West Virginia Academy of Science* 95.2 (2023).

[37] Hammersborg, Patrik, and Inga Strümke. "Reinforcement learning in an adaptable chess environment for detecting human-understandable concepts." *IFAC-PapersOnLine* 56.2 (2023): 9050-9055

[38] Young, Lily, and Byeong Kil Lee. "A Reinforcement Learning Approach to Training Chess Engine Neural Networks." *2023 Congress in Computer Science,*

Computer Engineering, & Applied Computing (CSCE). IEEE, 2023.

[39] Krishnamurthy, Vallidevi, et al. "Design of a Chess agent using reinforcement learning with SARSA Network." *Artificial Intelligence*. Chapman and Hall/CRC, 2021. 163-170.

[40] Bertram, Timo, Johannes Fürnkranz, and Martin Müller. "Supervised and reinforcement learning from observations in reconnaissance blind chess." *2022 IEEE Conference on Games (CoG)*. IEEE, 2022.

[41] Lai, Matthew. "Giraffe: Using deep reinforcement learning to play chess." *arXiv preprint arXiv:1509.01549* (2015).

[42] Patankar, Shreya, et al. "A Survey of Deep Reinforcement Learning in Game Playing." *2024 MIT Art, Design and Technology School of Computing International Conference (MITADTSoCiCon)*. IEEE, 2024.

[43] Chess board editor. [Online]. Available: https://lichess.org/editor

[44] En passant, Wikipedia [Online]. Available: https://en.wikipedia.org/wiki/En_passant

[45] How the Chess Pieces Move [Online]. Available: https://www.chessable.com/blog/how-the-chess-pieces-move/