University of Sheffield

Comparative Analysis of Reinforcement Learning Techniques for Chess AI Development

Nishant Panchal

Supervisor: Eleni Vasilaki

A report submitted in fulfilment of the requirements
for the degree of MSc in Data Analytics

in the

Department of Computer Science

September 20, 2023

Declaration

All sentences or passages quoted in this report from other people's work have been specifically acknowledged by clear cross-referencing to author, work and page(s). Any illustrations that are not the work of the author of this report have been used with the explicit permission of the originator and are specifically acknowledged. I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this project and the degree examination as a whole.

Name: Nishant Panchal

Signature: Nishant

Date: 30/08/2023

Abstract

Reinforcement learning (RL) stands at the forefront of artificial intelligence, embodying the dream of machines that can learn to navigate complex tasks independently. Within this exciting frontier, this research provides a comprehensive analysis of reinforcement learning techniques—specifically Deep Q-Networks (DQN), Double Deep Q-Networks (DDQN), and Monte Carlo Tree Search (MCTS)—for advancing Chess Artificial Intelligence. Initial experiments demonstrated the strengths and weaknesses of these algorithms in self-play scenarios, offering insights into their tactical and strategic capabilities. Building upon this, the study investigated the efficacy of AlphaZero's pipeline, which, while potent, proved to be highly resource-intensive. A hybrid approach was then introduced, using the Computer Chess Rating Lists (CCRL) dataset's expert-guided learning and AlphaZero's self-improvement guided by MCTS algorithm. This combination aimed to increase performance and strategic depth while reducing computational burden and training time. The hybrid approach produced appealing results, striking a delicate balance between computational effectiveness and tactical expertise. These discoveries give a resource-effective yet reliable methodology for the building of chess engines through reinforcement learning, which contributes to the growing field of chess AI.

Acknowledgement

I would like to express my heartfelt gratitude to my supervisor, Prof. Dr. Eleni Vasilaki, for entrusting me with this intellectually stimulating project and for her unwavering guidance throughout the research journey. Her invaluable insights and expertise in the field have been instrumental in shaping this study into what it is today. Additionally, I am thankful for the encouragement and sound advice I've received throughout my academic career from my parents and friends, whose unconditional support has been a consistent source of strength in the completion of this project.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Chess is an ancient game that has captivated human minds for centuries. The beauty of chess lies in the endless creativity and sophisticated reasoning involved in each move. Chess was considered as an activity just like writing poems, painting etc. for many years due to these reasons. Every game of chess played is like a poem which tells a story of a war between the two armies. Behind the beauty of chess lies a captivating battle of tactics, presenting a formidable challenge that can be arduous to unravel and analyse for many individuals.

"The game of chess is one of the most widely-studied domain in the history of artificial intelligence"[20]. In the past few decades, the intersection of chess and artificial intelligence (AI) has enabled the investigation of uncharted domains of comprehension and strategic involvement. The development of robust chess engines and the seamless incorporation of artificial intelligence methodologies have undeniably revolutionised our methodology when it comes to engaging with the game.

Within the domain of chess and artificial intelligence, two notable encounters between IBM's Deep Blue chess engine[3] and the reigning world chess champion, Garry Kasparov, have served as significant milestones. The first encounter took place in 1996, followed by a highly anticipated rematch in 1997.

A turning point in the fusion of chess and artificial intelligence was marked by the legendary games between Deep Blue and Garry Kasparov in 1996 and 1997. Despite Kasparov's victory in 1996, Deep Blue's outstanding performance signalled the beginning of a new era in computer chess. Deep Blue improved for the much awaited rematch in 1997, adding sophisticated search algorithms and doing evaluations at a rate of millions of board states per second. Following these improvements, Deep Blue defeated Kasparov with a score of 3.5-2.5, garnering international attention and demonstrating the immense potential of artificial intelligence in simulating human-like strategic thinking.

The series of encounters conducted between Deep Blue and Kasparov in the years 1996 and 1997 exemplify noteworthy milestones within the domains of chess and artificial intelligence. This battle between human and a machine contributed significantly to the progress of computer chess and have served as a source of inspiration for subsequent scholarly investigations. Their work has effectively showcased the capacity of technology to both challenge and captivate human cognitive abilities. The confrontations between Deep Blue and Kasparov have become widely recognized and hold a lasting significance in the chronicles of chess.

Following the historic matches between Kasparov and Deep Blue, the field of computer chess has undergone remarkable transformation. Chess AI has reached new heights as a result of the quick development of hardware technology and the generation of complex algorithms. Today's advanced chess engines, which were formerly restricted to supercomputers, are available on personal mobile devices, democratising access to top-notch artificial opponents and analysis tools for chess fans around the world.

Concurrently, there has been a major shift in how chess AI is being developed. An integrated, hybrid methodology is replacing the era of reliance on manually created heuristics and brute-force search strategies. Chess AI development that incorporates machine learning methods, particularly reinforcement learning, holds the promise of allowing computers to comprehend and interact with the game in entirely new ways[20]. This ground-breaking fusion aims to uncover deeper, more nuanced levels of the strategic intricacy of the game, not just to increase the computer's chess playing prowess.

Given the astronomical number of possible chess games—estimated at around 10^{120} (Shannon's Number)[17]—a brute-force approach that evaluates all possible game trees is computationally infeasible. This complexity necessitates the use of more sophisticated approaches, such as Reinforcement Learning, which can navigate this vast search space more efficiently and effectively. Reinforcement Learning (RL) is a type of machine learning paradigm where an agent learns to behave in an environment by executing specific actions and observing the rewards of those behaviours. It is comparable to instructing a machine to learn from its mistakes, simulating how people or animals learn via trial and error. In simple terms RL replicates the way humans learn: through experience. An agent continuously interacts with its environment, takes decisions based on its present state, and learns from the results of those decisions in order to maximise some idea of cumulative reward.

1.1 Aims and Objectives

The primary aim of this study is to investigate the application of Reinforcement Learning (RL) techniques in the chess domain, leveraging algorithms such as Deep Q-Network (DQN), Double Deep Q-Network, and Monte Carlo Tree Search (MCTS) to enhance chess strategies through self-play. A key objective is to systematically assess the overall performance of these models across different scenarios. While the groundbreaking AlphaZero model serves as an important point of reference, this research aims to critically analyze its sophisticated fusion of neural networks and MCTS rather than attempt a full replication, given its extensive computational requirements. The **novelty** of this research is encapsulated in the introduction of the Hybrid approach: an initial supervised learning phase using the expert dataset, followed by a refined self-play phase with controlled exploration. This two-stage learning mechanism aims to incorporate elements of the AlphaZero pipeline, blending traditional chess wisdom with advanced RL techniques to improve performance while minimizing computational costs and time.

1.2 Overview of the Report

1.2.1 Chapter 2: Literature Review

This chapter explores Reinforcement Learning (RL) in depth, outlining its fundamental ideas and essential algorithms. The chapter also provides a thorough analysis of the development of chess engines and talks about the important algorithms they employ. Some basic chess terminologies are also discussed here. The aim is to define the theoretical background and historical development of AI in chess. In addition to traditional algorithms, the chapter also talks about the advancements in deep learning, such as Deep Neural Networks, Activation Functions, and the Backpropagation Algorithm. The overarching aim is to provide an integrated framework for understanding the multifaceted contributions of artificial intelligence to the field of chess

1.2.2 Chapter 3: Methodology

In Chapter 3, the investigation articulates a multi-faceted experimental methodology that spans from a simplified 4x4 chess environment to the conventional 8x8 board configuration. Techniques such as Deep Q-Networks (DQN), Double Deep Q-Networks and Monte Carlo Tree Search (MCTS) are meticulously analyzed and elucidated. Rather than attempting to replicate AlphaZero, given its computationally intensive nature, the chapter opts for a critical examination and evaluative discussion of its underlying methodology. The chapter also outlines the complete iterative pipeline employed in this study, and introduces the Hybrid approach which is basically combining Deep Reinforcement Learning with Supervised Learning.

1.2.3 Chapter 4: Results and Discussion

Chapter 4 focuses on presenting both quantitative and qualitative results from the various experiments conducted. The chapter provides a recapitulation of each experiment along with the discussion on the results. It offers a detailed analysis of the models' performance, and the comparison among different approaches. Each approach's effectiveness is critically appraised to give an in-depth understanding of their pros and cons.

1.2.4 Chapter 5: Conclusions and Future Works

The final chapter, Chapter 5, synthesizes the key findings of the study and draws conclusive statements about the potential and limitations of applying RL techniques to chess AI. It also articulates promising directions for future research, suggesting ways the work might be further extended or refined.

Chapter 2

Literature Survey

2.1 Reinforcement Learning

Reinforcement learning is a branch of machine learning that focuses on how an agent can learn to make optimal decisions based on the actions it takes in the given environment it interacts with. It encompasses the idea of learning through experiences, where the agent learns by receiving feedback in the form of rewards or punishments. Similar to how humans learn from their interactions with the world. When an infant engages in activities such as playing, moving its arms, or observing its surroundings, it has no explicit teacher. However, the child still has a direct, sensory and motor relationship with the environment, through which it learns and adapts[23]. Similarly a reinforcement learning agent learns by actively exploring various states, taking actions, and subsequently observing the resulting rewards or penalties achieved from taking those actions.

The learning process involves the agent's pursuit of a specific goal as stipulated by the user. Through exposure to multiple states and receiving feedback, the agent progressively enhances its decision-making abilities and acquires knowledge regarding appropriate actions to undertake in different situations. As an illustration, akin to how an infant acquires the knowledge of refraining from touching hot surfaces, a reinforcement learning agent possesses the capability to acquire the understanding of avoiding acts that result in unfavourable outcomes.

Reinforcement Learning (RL) stands in marked contrast to brute force approaches in terms of computational strategy[23]. While brute force approaches systematically explore all possible solutions to a problem, RL focuses on learning from interaction with an environment and the experiences gained from it to achieve a goal. RL is versatile and applicable in various domains. It can be utilized in scenarios such as a robot learning the most efficient path from one location to another or an AI playing board games and playing the optimal moves to win. By creating environments with clear rewards and penalties, a reinforcement learning agent can learn to interact with the environment in a manner that maximizes its performance in achieving the defined goal.

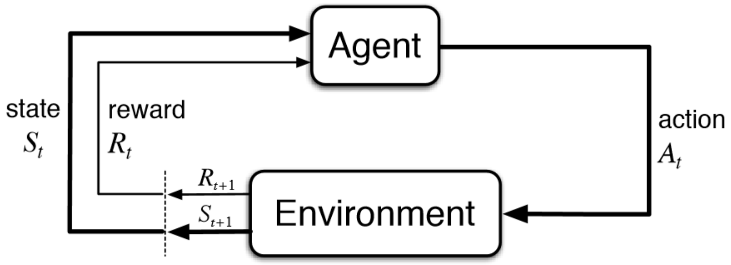

Figure 2.1: The reinforcement learning cycle

2.1.1 The Mathematics of the Markov Decision Processes

In a Reinforcement Learning (RL) context, a Markov Decision Process (MDP) is a mathematical framework used to describe an environment for RL[1]. As per the name suggests, these processes obey the Markov property, the current state of the environment depends on the previous state of the system. Here, the agent takes actions in states, receives rewards, and transitions between states according to certain probabilities. Let's define these components more formally:

- **States (S):** MDP assumes a finite set of states in which the system can exist. This set, although potentially large, is bounded.
- **Actions (A):** In each state, an agent can choose to perform one among a finite set of actions.
- **Rewards (R):** Upon performing an action in a given state, the agent receives a reward which is a random variable from a finite set, described by a probability distribution dependent on the current state-action pair.

The probability of transitioning from state s to s' and receiving reward r after taking action a is denoted as:

$$p(s', r|s, a) \tag{2.1}$$

This expression represents the probability of arriving at state s' with a reward r, given the current state s and action a. A fundamental property of this probability distribution is:

$$\sum_{s' \in S} \sum_{r \in R} p(s', r|s, a) = 1 \tag{2.2}$$

This property essentially encapsulates the entire dynamics of the system - ensuring that the transitions between states are described fully and coherently by this probability distribution.

To compute the **state transition probabilities** — the likelihood of transitioning from state s to s' when action a is taken — one sums over the possible rewards:

$$p(s'|s,a) = \sum_{r \in R} p(s', r|s,a) \tag{2.3}$$

Here, s' denotes the next state following the action a from the current state s.

A critical quantity in RL is the **expected reward** for a state-action pair, (s, a), which is defined as:

$$r(s,a) = \mathbb{E}[R_t|S_{t-1} = s, A_{t-1} = a] = \sum_{r \in R} r \sum_{s' \in S} p(s', r|s,a) \tag{2.4}$$

This is an expectation value, representing the average reward an agent expects to receive after taking action a in state s.

To illustrate this with a simple example, consider a fair coin flip game. A heads result earns the player one point, while tails result in a loss of one point. With equal probability for heads or tails, the expected reward for each flip is zero, reflecting the balanced nature of the game.

In more complex environments encountered in RL, this idea extends beyond single actions to consider the agent's trajectory through the state space of the environment, and the sequence of rewards it accumulates along its path. Thus, the MDP framework becomes a powerful tool for modeling and solving RL problems.

2.1.2 Returns and Reward Discounting

In reinforcement learning, the objective is to design an agent that acts so as to maximize the cumulative reward it receives over time. This naturally leads to the concept of *returns.* Formally, the return, denoted as G_t, is defined as the total sum of rewards the agent expects to receive over the course of an episode, starting from time t until a terminal time step T:

$$G_t = r_{t+1} + r_{t+2} + \ldots + r_T \tag{2.5}$$

However, in continuous tasks where an episode could potentially never end, this sum may grow indefinitely. To address this issue, *discounting* is introduced through a hyperparameter, denoted as γ (gamma), where $0 \leq \gamma \leq 1$. This parameter weights the sum of the rewards, applying a geometrically decreasing scale to future rewards:

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{2.6}$$

Here, as γ approaches 0, the agent is increasingly short-sighted, focusing on immediate rewards. As γ approaches 1, the agent becomes more far-sighted, emphasizing future rewards. This discount factor not only ensures that the sum of rewards remains finite but also allows the agent to account for the uncertainty associated with future rewards, given the potentially stochastic nature of the environment.

Remarkably, the formulation for returns exhibits a recursive property, useful for various RL algorithms:

$$G_t = r_{t+1} + \gamma G_{t+1} \tag{2.7}$$

This recursive relationship simplifies the calculation of returns, providing an efficient means to compute them. It is assumed that the return at the terminal state, G_T, is zero, consistent with the notion that no further rewards are received after the episode terminates.

2.1.3 Policies and Value Functions

While discounted returns provide a mathematical framework to define the objective that the agent seeks to maximize, they alone are insufficient to guide the agent's behavior. Without associating these returns with the states encountered during interaction with the environment, an agent lacks a mechanism to inform future decisions based on past experiences. This is where the concept of *value functions* comes into play.

A value function quantifies the expected future rewards an agent can expect to accumulate, starting from a particular state and acting according to a specific policy. In essence, it serves as a measure of the *quality* or *goodness* of different states, guiding the agent towards states that are expected to yield higher returns.

A *policy*, denoted by the Greek letter π, defines the agent's behavior. It is a mapping from each state $s \in S$ to a probability distribution over the available actions $a \in A$. Mathematically, $\pi(a|s)$ represents the probability of selecting action a given that the agent is in state s.

The state-value function for a policy π, denoted $v_\pi(s)$, is defined as the expected return when starting in state s and following policy π thereafter:

$$v_\pi(s) = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| S_t = s \right] \tag{2.8}$$

for all $s \in S$.

In addition to the state-value function, we also define the action-value function for a policy π, denoted $q_\pi(s, a)$, which represents the expected return after taking an action a in state s and thereafter following policy π:

$$q_\pi(s, a) = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \middle| S_t = s, A_t = a \right] \tag{2.9}$$

To make these definitions concrete, consider an example in which an agent's policy, based on its current state, prescribes certain probabilities for taking various actions. Each action, in turn, leads the agent to a new state and yields a corresponding reward. The value functions then quantify, in terms of expected future rewards, the desirability of states and actions under this policy. As the agent learns from its interaction with the environment, it seeks to update its policy in a manner that increases the probability of selecting actions leading to higher expected returns.

Learning in this context means adapting the policy based on the agent's ongoing interaction with the environment and the observed returns, with the aim of steering the agent towards actions that are expected to yield higher returns in the long run.

2.1.4 The Bellman Equation and Optimal Policies

The Bellman Equation, named after Richard Bellman, serves as a cornerstone in the domain of reinforcement learning by providing a recursive formula that links the value of a state to the values of its successor states. It's formulated for the state-value function under a policy π as:

$$V^{\pi}(s) = \sum_{a \in A} \pi(a|s) \sum_{s' \in S} \sum_{r \in R} p(s', r|s, a)[r + \gamma V^{\pi}(s')] \tag{2.10}$$

For the action-value function under a policy π, the equation is:

$$Q^{\pi}(s, a) = \sum_{s' \in S} \sum_{r \in R} p(s', r|s, a)[r + \gamma \sum_{a' \in A} \pi(a'|s') Q^{\pi}(s', a')] \tag{2.11}$$

Bellman's theorem posits that an optimal policy π^* is greedy with respect to the optimal value function V^* or action-value function Q^*. Formally,

$$\pi^*(s) = \arg\max_a Q^*(s, a)$$

This aligns with principles outlined in Chapter 3 of Sutton and Barto's book on Reinforcement Learning[23].

With the grounding in Bellman's Equation, we can discern that a policy π is considered optimal if it outperforms or equals all other policies π' for every state s[18], mathematically expressed as:

$$v_{\pi}(s) \geq v_{\pi'}(s), \forall s \in S. \tag{2.12}$$

An optimal policy π_*, thus, satisfies:

$$v_{\pi_*}(s) \geq v_{\pi}(s), \forall s \in S, \forall \pi. \tag{2.13}$$

The remainder of this paper will delve into algorithms and methodologies for efficiently computing or approximating these optimal policies.

2.1.5 Exploration vs Exploitation

In reinforcement learning scenarios, an agent must balance its actions between exploration (trying new, potentially rewarding decisions) and exploitation (making decisions based on current knowledge to maximize reward). Striking an optimal balance between these two components is paramount.

If the agent has complete knowledge of the environment, i.e., the transition probability $P_{a_{ss'}}$ can be calculated given state s and action a, the Bellman equation can be solved iteratively for a suitable discount factor γ. However, in unknown environments, substantial data must be collected to estimate $P_{a_{ss'}}$ before the action-value function q converges. Should q stabilize before the environment is fully explored, the model's performance may prove unsatisfactory, especially in situations with large action spaces.

To address the challenge of balancing exploration and exploitation, a variety of policies have been developed. One such policy is the ϵ-greedy[23] selection strategy. In this approach, rather than consistently selecting the action estimated to yield the best reward according to the q-function, the agent adopts a probabilistic mechanism. Specifically, with probability ϵ, the agent will randomly choose from among all available actions. This ensures that the agent performs sufficient exploration before the action-value function converges.

$$\pi(a|s) = \begin{cases} \epsilon/m + 1 - \epsilon & \text{if } a^* = \arg\max_{a \in A} q(s,a) \\ \epsilon/m & \text{otherwise} \end{cases} \tag{2.14}$$

where m represents the number of possible actions. Initially, this method may negatively impact the agent's performance in the initial episodes of training. However, in the long term, it broadens the agent's exposure to diverse states and actions, thereby fostering more robust learning.

Balancing the Trade-off The value of ϵ serves as the balance between exploration and exploitation. A higher ϵ value encourages more exploration at the cost of short-term performance, while a lower ϵ emphasizes exploitation, enabling the agent to more frequently select actions it currently estimates as optimal. It is crucial to tune ϵ as part of the training process, and often it is decreased over time, allowing for more exploration in the early stages of training and more exploitation as the agent becomes more competent.

Other Exploration Strategies For completeness, it's worth mentioning that there are alternative strategies to ϵ-greedy for balancing exploration and exploitation. One such method is the Upper Confidence Bound (UCB) algorithm, which selects actions based on both their estimated value and the uncertainty surrounding those estimates. Another approach is Thompson Sampling, which uses Bayesian inference to model the uncertainty in action values and make exploration decisions accordingly. Both methods aim to be more adaptive in their exploration strategies compared to ϵ-greedy.

2.1.6 Monte Carlo Methods

Monte Carlo (MC) methods, named after the famous casino in Monaco, represent a broad class of computational algorithms that harness the power of random sampling to produce numerical results. Originating from statistical physics and mathematical finance, they have permeated various domains including operations research, project management, and more recently, reinforcement learning (RL).

The essence of MC methods lies in their principle of averaging sample returns for value estimation. Given a particular state, its value is simply the mean of the returns that have followed all instances of that state. Their inherent stochasticity negates the need for a model of the environment, positioning them as *model-free* methods. When dissected, MC methods in RL can be categorized as:

- **First-Visit MC:** This method estimates the value for a state by only considering the first visit to that state in each episode.
- **Every-Visit MC:** In contrast, this approach accounts for all visits to a state in each episode for value estimation.

For MC methods to be effective in RL, they require constant exploration[5]. This ensures that every state-action pair has a non-zero chance of being selected, enabling the estimates for each state to converge. To facilitate this, the *-soft* policy is often adopted[5], in which every action has at least a probability of being selected.

The application of MC methods doesn't stop at value estimation. They can also play an instrumental role in refining policies. Once the value function has been computed through MC prediction methods, a superior policy can be derived by acting greedily concerning this value function.

In recent years, the integration of MC methods with modern deep learning paradigms has gained traction. One notable example is DeepMind's AlphaZero, which adeptly combined MC methods with deep neural networks and tree search techniques[20]. Specifically, AlphaZero employed the Monte Carlo Tree Search (MCTS) method. MCTS blends conventional tree search with MC sampling of the search space and has been used to master games like Chess, Go, and Shogi, often exceeding human performance levels[20]. With deep learning, the function approximation capabilities significantly enhance the MCTS, allowing it to efficiently traverse vast game trees and channel computational resources toward the most promising move sequences.

However, MC methods come with their set of constraints. Their episodic character mandates the conclusion of episodes to ascertain rewards. This poses challenges for tasks that are continuous and don't have clear episode boundaries. Nevertheless, these drawbacks can be somewhat mitigated by coupling MC methods with strategies like Temporal Difference (TD) learning[22] to devise hybrid algorithms.

2.2 Reinforcement Learning Algorithms

Reinforcement learning algorithms provide a formalism for learning optimal policies through interactions with the environment. This section presents three significant algorithms in the field: Deep Q Networks (DQN), Double Deep Q Networks (DDQN), and Monte Carlo Tree

Search (MCTS). Each of these algorithms offers unique advantages and challenges in solving reinforcement learning problems.

2.2.1 Deep Q Networks (DQN)

The Deep Q Network algorithm, commonly known as DQN, is a value-based reinforcement learning algorithm that combines traditional Q-Learning with the expressive power of deep neural networks[13][14]. It introduces mechanisms like experience replay and target networks to stabilize the training. The pseudocode for the DQN algorithm is presented below:

Algorithm 1 Deep Q Network

```
1: Initialize policy network $q(s, a)$ with random weights.
2: Set $\alpha, N, \epsilon, \gamma$.
3: Set $M, T$.
4: Set $q'(s, a) = q(s, a)$.
5: for episode = 1 to $N$ do
6:     $t = 0$.
7:     Get $s_0$ from environment.
8:     while episode not complete do
9:         Choose $a_t$ using $\epsilon$-greedy policy from $q(s, a)$.
10:        Execute $a_t$ to get $s_{t+1}, r_{t+1}$.
11:        Store $(s_t, a_t, s_{t+1}, r_{t+1})$ in replay pool.
12:        Sample $M$ transitions from pool.
13:        Calculate $q(s_k, a_k)$ from $q(s, a)$.
14:        if next state is complete then
15:            $y_k = r_{k+1}$.
16:        else
17:            $y_k = r_{k+1} + \gamma \max_a q'(s_{k+1}, a)$.
18:        end if
19:        Update $q(s, a)$ using $(y_k - q(s_k, a_k))^2$.
20:        if t mod $T$ is 0 then
21:            Update $q'$ from $q$.
22:        end if
23:        Increment $t$.
24:    end while
25: end for
```

2.2.2 Double Deep Q Networks (DDQN)

Building on the DQN framework, Double Deep Q Networks (DDQN) aim to solve the problem of overestimation bias in value function approximation. The DDQN algorithm uses two separate networks: a policy network for selecting actions and a target network for evaluating those actions[24]. This separation helps to stabilize learning. Below is the pseudocode for the DDQN algorithm:

Algorithm 2 Double Deep Q Network (DDQN)

```
1: Initialize policy network $q(s, a; \theta)$ with weights $\theta$.
2: Initialize target network $q(s, a; \theta')$ with weights $\theta' = \theta$.
3: Initialize replay buffer $\mathcal{R}$.
4: for episode = 1 to $N$ do
5:     Get initial state $s$.
6:     while episode not complete do
7:         Choose $a$ using $\epsilon$-greedy policy derived from $q$.
8:         Execute $a$ to get next state $s'$ and reward $r$.
9:         Store transition $(s, a, r, s')$ in $\mathcal{R}$.
10:        Sample random minibatch of transitions from $\mathcal{R}$.
11:        Set $y = r$ if the episode terminates at next step, else:
12:        $y = r + \gamma q(s', \text{argmax}_{a'} q(s', a'; \theta); \theta')$.
13:        Perform a gradient descent step on $(y - q(s, a; \theta))^2$ with respect to $\theta$.
14:        if time step mod $C$ is 0 then
15:            Update $\theta'$ with $\theta$.
16:        end if
17:        Move to the next state $s'$.
18:    end while
19: end for
```

2.2.3 Monte Carlo Tree Search (MCTS)

Monte Carlo Tree Search[8] is a model-free, online search algorithm used mostly in perfect-information games but can be applied to general reinforcement learning problems. Unlike DQN and DDQN, which are value-based methods, MCTS focuses on the exploration-exploitation trade-off by growing a search tree that represents different future possibilities. Here is the pseudocode for the MCTS algorithm:

2.3 Deep Learning

Deep learning is a subset of machine learning that employs deep neural networks to carry out tasks that would normally require human intelligence. Unlike traditional machine learning methods that plateau after a certain point with increasing data, deep learning algorithms improve their performance[10]. Below, we provide an overview of the foundational concepts of deep learning.

2.3.1 Neural Networks

A neural network consists of layers of interconnected nodes or ”neurons”. The simplest form, a single-layer perceptron, contains an input layer and an output layer. More sophisticated networks, like multi-layer perceptrons (MLPs), consist of one or more hidden layers between them. Each connection between nodes has an associated weight w, which is adjusted during

Algorithm 3 Monte Carlo Tree Search (MCTS)

```
1: Initialize root node $R$ with state $s_0$.
2: Set number of iterations $N$ and exploration factor $C$.
3: for iteration = 1 to $N$ do
4:     node ← $R$                                   ▷ Initialize at root
5:     path ← []                                    ▷ Initialize empty path
6:     while node is not terminal do
7:         if node is fully expanded then
8:             node ← SelectBestChild(node, C)
9:         else
10:            node ← Expand(node)
11:            Break
12:        end if
13:        Append node to path
14:    end while
15:    $\Delta$ ← Rollout(node)                     ▷ Simulate from node to terminal state
16:    for all node in path do
17:        Update $N$(node) and $Q$(node) with $\Delta$
18:    end for
19: end for
20: Return BestChild(R, 0)          ▷ Return the best child of root without exploration
```

training. Data inputs are fed into the input layer, and they undergo a series of transformations through the hidden layers until they reach the output layer.

2.3.2 Deep Neural Networks

Deep Neural Networks (DNNs) extend the concept of neural networks by adding more hidden layers, enabling the network to learn more complex and abstract features from the data[9]. The "depth" in deep learning refers to the number of layers through which the data is transformed. More layers allow for more complexity but also introduce challenges, such as the vanishing or exploding gradient problems, which can be mitigated through careful initialization of weights and the use of specific types of activation functions.

2.3.3 Activation Functions

Activation functions introduce non-linearity into the network, enabling it to learn from the error and make adjustments. The general form of the operation that occurs in a single neuron during the forward pass is:

$$f(x \cdot w + b) \tag{2.15}$$

Here, f is the activation function, x is the vector of input features, w is the vector of weights, and b is the bias term. The function f introduces non-linearity and transforms the weighted sum of inputs $x \cdot w + b$ into an output suitable for either the subsequent layers in the network or the final output layer.

Some common types of activation functions are:

- **Sigmoid (σ)**: Maps any input x into a value between 0 and 1.

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.16}$$

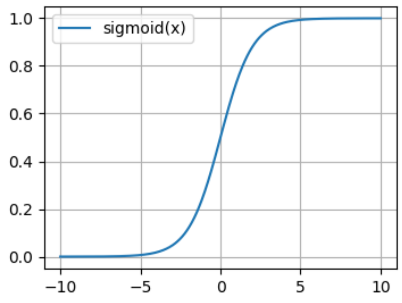

Figure 2.2: The Sigmoid function

- **ReLU (Rectified Linear Unit)**: Allows only positive values to pass through, thereby speeding up training time.

$$\text{ReLU}(x) = \max(0, x) \tag{2.17}$$

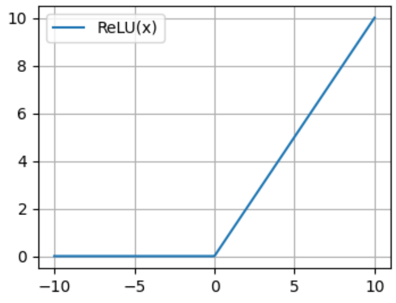

Figure 2.3: The ReLU function

- **Tanh**: Similar to Sigmoid but maps values to a range between -1 and 1.

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{2.18}$$

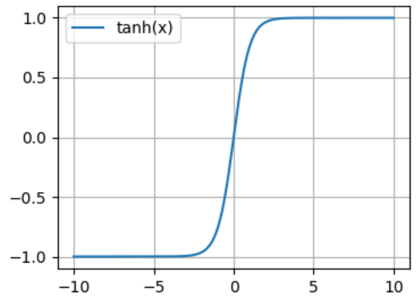

Figure 2.4: The TanH function

- **Softmax**: Often used in the output layer of a network for multi-class classification.

$$\text{Softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{2.19}$$

2.3.4 Backpropagation Algorithm

Backpropagation (backward propagation of errors) is a supervised learning algorithm used for minimizing the error in the neural network's predictions. During the forward pass, the network makes a prediction[16]. The error E is then calculated, usually as a mean squared error:

$$E = \frac{1}{2}\sum_k (y_k - t_k)^2 \tag{2.20}$$

Where y_k is the predicted output and t_k is the true label.

The backpropagation algorithm adjusts the network's weights w_{ij} in reverse order, starting from the output layer and moving toward the input layer. The aim is to minimize the error function by adjusting the weights via gradient descent:

$$w_{ij}^{new} = w_{ij}^{old} - \alpha \frac{\partial E}{\partial w_{ij}} \tag{2.21}$$

Here, α is the learning rate, which controls the step size in the gradient descent optimization.

The learning rate is an important hyperparameter in backpropagation; too high a learning rate may cause the model to converge too quickly and possibly overshoot the minimum cost, while too low a learning rate may cause the model to learn too slowly or not converge at all[16].

2.4 Evolution of Chess Engines

The evolution of chess engines is a testament to advancements in artificial intelligence, optimization algorithms, and machine learning. Over the years, various engines have surfaced, each influencing and, in many ways, revolutionizing how chess is perceived and played. This section delves into notable engines, charting their developments, methodologies, and impact on both the game and computational paradigms.

Deep Blue (IBM): IBM's Deep Blue etched its name in history following its landmark match against Garry Kasparov in 1997[17]. Though not based on machine learning in the contemporary sense, it employed a combination of brute-force search with an optimized evaluation function to dissect positions. Deep Blue's strategy was grounded in sheer computational prowess and intricate algorithms to probe game trees[6].

Komodo: Developed by the trio of Don Dailey, Larry Kaufman, and Mark Lefler, Komodo has become a quintessential symbol of combining positional insight with tactical strength in computer chess. Rather than relying solely on sheer computational brute force, Komodo incorporates a deep appreciation for chess's positional intricacies. Its evaluation function, influenced by Grandmaster Kaufman, often showcases moves emphasizing positional richness over mere tactical advantage. The resultant approach, especially in its advanced versions, mimics a human-like touch in the game. By versions like Komodo 10, it was not only clinching titles but also serving as a pivotal training tool, representing the synthesis of human expertise and computational prowess.

Stockfish: Stockfish stands out as an embodiment of advanced game theory application in chess. While its base lies in a powerful static evaluation function crafted over years, the true strength arises from its dynamic game analysis. Using the 'minimax' algorithm, Stockfish forecasts potential moves and counter-moves, creating a game tree of possible outcomes. To manage the exponential growth of this tree, it employs alpha-beta pruning, focusing only on the most relevant move sequences. This combination of a strong heuristic, predictive algorithms, and pruning techniques makes Stockfish a formidable chess engine. Its dominance in competitions like the Top Chess Engines Championship (TCEC) since 2010 attests to its prowess.

AlphaZero: Reinventing Chess Strategy with Deep Learning AlphaZero, a brainchild of DeepMind[20], represents a radical shift in computer chess paradigms. Eschewing extensive databases and hand-tailored evaluation functions, AlphaZero adopts deep neural networks coupled with reinforcement learning. With merely the rudimentary rules of chess as its foundation, it embarked on a self-play journey, playing millions of games against its own evolving versions.

Distinct from its counterparts, AlphaZero's forte isn't in brute force position evaluation. Instead of probing millions of positions each second, it delves deeper into fewer positions, guided by its neural network to produce evaluations reminiscent of human grandmaster insight. Its gameplay, characterized by aggressive and innovative strategies, has illuminated uncharted territories in the chess world, offering novel stratagems previously unexplored.

However, in a surprising move, Google's DeepMind eventually discontinued further development and public exhibitions of AlphaZero. While the exact reasons remain proprietary, this decision underscored the experimental nature of AlphaZero as a project, emphasizing its role in pushing the boundaries of AI research more than serving as a dedicated chess entity.

Leela Chess Zero (LCZero): A Community-driven AI Powerhouse Drawing inspiration from AlphaZero's methodologies, LCZero, colloquially known as Leela, incorporates neural network-based position evaluations. However, what sets LCZero apart is its origin—a collaborative, community-driven project.

Unlike the proprietary training of AlphaZero by DeepMind, LCZero's evolution benefits from a decentralized, community-driven approach. With contributors worldwide offering computational power, Leela's training set swells daily, making it a testament to the prowess of collective computing and the spirit of open-source collaboration.

Mirroring AlphaZero's intuitive gameplay, Leela offers profound strategic insights often punctuated with innovative creativity. Its evaluations often challenge longstanding chess conventions, thereby enriching the broader understanding of the game.

Key Machine Learning Mechanisms

Over the years, chess engines have incorporated several computational techniques to refine their gameplay. Of these techniques, three stand out as particularly influential: Monte Carlo Tree Search, Min-Max Algorithm, and Alpha Beta Pruning. Each of these methods offers a unique approach to evaluating game positions and determining the most promising moves.

Monte Carlo Tree Search (MCTS): MCTS, inspired by the Monte Carlo method's statistical sampling techniques, has emerged as a powerful decision-making tool for complex game scenarios like chess. By intertwining the randomness of Monte Carlo simulations with the structured nature of tree search, MCTS effectively evaluates a vast array of potential moves, considering both their immediate and long-term implications.

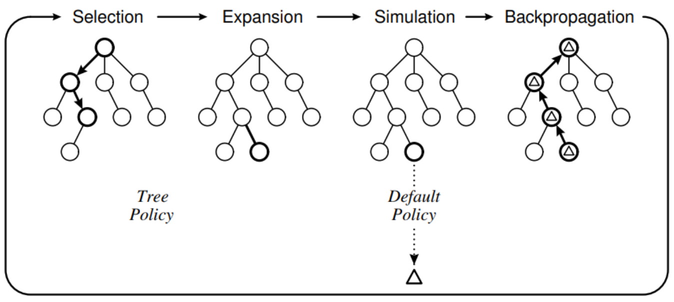

Figure 2.5: One iteration of the general MCTS approach[2].

1. **Selection:** Commencing at the search tree's root, the algorithm navigates through the tree by systematically selecting child nodes. This is done using a strategy that seeks

an equilibrium between exploring uncharted nodes and exploiting nodes known to be promising based on previous simulations.

2. **Expansion:** When an unexplored leaf node is encountered, it signals a potential game state that hasn't been simulated yet. Here, the tree expands by adding child nodes representing feasible moves from this game position.

3. **Simulation/Playout:** From this freshly minted node, the algorithm simulates a game by randomly executing moves until it reaches an end state. These simulations, though randomized, can be subtly guided by heuristics to mirror realistic gameplay better.

4. **Backpropagation:** Post simulation, the outcome—whether it's a win, loss, or draw—is propagated back to the root. Each node traversed during the selection phase is updated, refining their associated values and thereby influencing subsequent tree navigations.

Given the vast decision space in chess, MCTS's ability to balance between exploration and exploitation ensures it can efficiently home in on promising strategies without becoming mired in computational inefficiencies.

Min-Max Algorithm: The heart of many chess engines, the Minimax algorithm operates on a simple yet profound principle: assume both players will always make the best possible move. By diving deep into the game tree, it seeks to unearth the move that offers the maximum benefit while operating under the assumption that the opponent is striving to minimize this benefit. The algorithm's recursive nature ensures a comprehensive exploration of potential game states, allowing it to assess the merits and demerits of each move. However, its efficacy isn't just in its depth but also in its incorporation of intelligent shortcuts like alpha-beta pruning, which helps sidestep redundant evaluations.

Alpha Beta Pruning: While the Minimax algorithm is thorough, the sheer number of potential game states in chess means that computational efficiency is paramount. Enter alpha-beta pruning, a technique that adds a discerning eye to the Minimax method. By maintaining two threshold values—alpha (the best score for the maximizing player) and beta (the best score for the minimizing player)—this technique can swiftly identify and discard branches of the game tree that don't need evaluation. As the search progresses depth-first through the game tree, if it becomes evident that a particular branch will not influence the final decision, that branch is pruned, saving valuable computational resources. This is determined by continuously updating the alpha and beta values and comparing them to ascertain if a branch holds any promise. Through this intelligent pruning mechanism, chess engines can delve deeper into the game tree within a given timeframe, enhancing their gameplay by considering more distant consequences of current moves.

Deep Reinforcement Learning: Deep reinforcement learning is a sub field at the intersection of deep learning and reinforcement learning that has been particularly influential in the development of advanced chess engines[14]. Unlike traditional rule-based engines, deep reinforcement learning models enable the engine to learn optimal strategies through self-play or by studying large datasets of historical games[12]. By combining deep neural networks for complex feature extraction and reinforcement learning algorithms for optimal decision-making, these models can adapt and improve over time, discovering intricate strategies and tactics that might even be counter intuitive to human players[11]. The use of deep reinforcement learning in chess engines like AlphaZero has demonstrated superhuman performance[20], raising the bar for what is possible in automated chess play.

2.5 Basic Chess Terminologies

The game of chess, a two-player strategy board game, has its own set of unique terms and concepts that are critical to both playing the game and understanding the computational models that aim to solve it. Understanding these terms is crucial for following the advanced algorithms and methodologies discussed later in this paper. Below are some of the most important chess terminologies.

2.5.1 Elo Rating

The Elo rating system is a method for calculating the relative skill levels of players in two-player games like chess. Created by Arpad Elo, the system assigns a numerical rating to each player based on their performance against other rated players. Higher numbers indicate stronger players.

2.5.2 Checkmate

Checkmate is the condition in chess where a player's king is under threat of capture ('in check') and there is no legal move that the player can make to remove that threat. Achieving checkmate is the primary objective in chess, as it ends the game in favor of the player who delivers the checkmate.

2.5.3 Stalemate

Stalemate occurs when a player has no legal moves and their king is not in check. When a stalemate happens, the game ends in a draw. It's a highly relevant condition to consider, especially when designing algorithms for playing chess.

2.5.4 Threefold Repetition

The game is drawn if the same position occurs three times, not necessarily consecutively, with the same player to move each time. This rule is often used as a drawing mechanism in complex positions where neither player can make progress.

2.5.5 Pawn Promotion

Pawn promotion in chess occurs when a pawn reaches the opposite end of the board. Upon reaching this rank, the player must exchange the pawn for a queen, rook, knight, or bishop of the same color. Most board positions ask for a promotion to a queen, but there are situations where other pieces might be preferred. This choice is strategic, and the replacement isn't limited to previously captured pieces.

2.5.6 Opening Books

Opening books are databases of well-studied initial sequences of moves. These are often employed in computer chess engines to improve their performance by bypassing the computation of well-understood positions and directly accessing strong moves from the database.

2.5.7 Endgame Tablebases

Endgame tablebases are computer databases that contain pre-calculated exhaustive analysis for chess endgame positions. They provide perfect moves for any given board state in the endgame, allowing engines to play these positions perfectly.

2.5.8 Material Captured

In chess, each piece is assigned a specific value, representing its relative strength and importance on the board. The following are the standard values universally accepted across the chess community:

- Pawn: 1 point
- Knight: 3 points
- Bishop: 3 points
- Rook: 5 points
- Queen: 9 points

By evaluating the material captured, the evaluation of the given board state can be done. While 'pieces captured' refers to the sheer count of taken pieces, its strategic significance is limited. On the other hand, 'material captured' quantifies the value of these taken pieces, providing insight into their relative importance in the game. Within the scope of this research, analyzing the material a model captures provides insight into its strategic performance and decision-making abilities. It is important to note that king is not present in the list as it cannot be captured, it can only be placed in a checkmate or a stalemate.

Chapter 3

Methodology

The primary aim of this research is to harness the power of reinforcement learning to better understand and advance in the realm of chess. Reinforcement learning offers many intriguing algorithms such as Deep Q-learning and Monte Carlo Tree search which can help to face the challenging complexity of chess as given by the Shannon's number. In the context of chess which is full of intricate strategies and immense space of possible moves these algorithms hold enormous potential. In this chapter, we will delve deeper into the methodological intricacies of each phase of our research and go through the multiple experiments done for the research. We will go through the objective of each experiment, setup and the approach used to conduct these experiments. This section provides a general setting for the experiments. The choice of the model and the Reinforcement learning approach will also be discussed.

3.1 Chess Endgame via Deep Reinforcement Learning (DRL)

We kickoff the experiment phase of the research to solve the problem of finding the checkmate with a king and a queen. The objective of this phase of the research is to harness the prowess of a deep neural network combined with a strong Reinforcement Learning algorithm to master the endgame checkmating technique. A 4×4 chessboard environment is taken with three chess pieces: a King and a Queen for white and only a king for black. This sort of endgame often occurs in a game of chess and it can be very tricky to find the optimal solution under time pressure. From complete beginner to low ELO rated player, checkmating with a queen and king is a tedious task as many individual fall for a stalemate eventually drawing the game even with a queen up position. With the help of Deep Reinforcement learning we will try to solve this challenge and compare the algorithms. The primary aim is to train a neural network to deliver checkmate in the fewest possible moves.

3.1.1 Objective

The primary aim of this study is to train a neural network to deliver checkmate in the fewest possible moves. To achieve this, we employ two prominent reinforcement learning algorithms: Deep Q-Network (DQN) and its variant, Double Deep Q-Network (DDQN). Given the limited computational resources available for this research, we have elected to focus on chess endgames as a manageable yet insightful domain for preliminary investigation.

Endgames in chess involve fewer pieces and generally simpler board states, making it a tractable problem space for applying reinforcement learning techniques from scratch. It provides an ideal testbed for comparing the efficacy of DQN and DDQN algorithms without the computational burden that would come with simulating full-length chess games. This focused approach allows us to more effectively deduce which algorithm performs better in our environment and provides valuable insights into why one may be more suitable than the other for future studies on more complex game states.

3.1.2 Setup

A 4×4 chessboard environment is taken with White playing with a king and a queen and black is playing with only a king. Standard chess rules are applied and the initial setup of the board is always a legal one. A legal position is one in which the king is not already in a check or a stalemate so the model has to find checkmate from scratch. The game will end if no result is obtained in 20 moves.

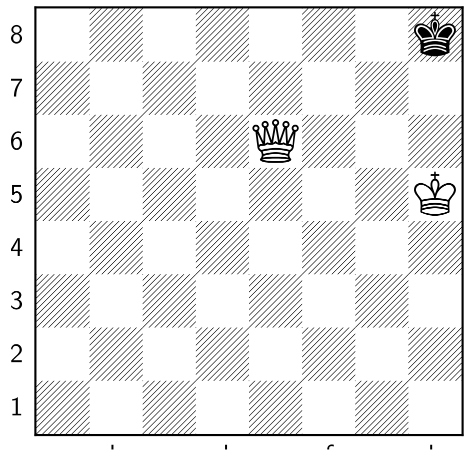

Figure 3.1: A sample endgame position

Fig 3.1 represents a sample endgame position of our chess environment. A thing to note here is that the playing field is reduced to a 4×4 board which resembles the environment used. This position is an interesting one as natural looking moves (defined as chess moves that appear to be strong or sensible choices at first glance) under time pressure can lead to instant stalemate, these scenarios happen often in real games where even with queen down position the opponent is reluctant to resign and wants to defeat someone by flagging or is hoping for a stalemate. Here white is to move and the moves such as King h6 or king g6 don't work in this position, they lead to stalemate/draw. Another natural looking move such as queen e5 check leads to a checkmate but it is also not the optimal solution here as it take more moves to checkmate.

3.1.3 Model Selection

The algorithms used for this experiment are Deep Q-learning (DQN) and Double Deep Q-learning (DDQN). Q-Learning is a model-free reinforcement learning algorithm, adept at determining the optimal action-selection policy for finite Markov decision processes[25]. These algorithms can efficiently handle the state representation of the given environment[13][14].

The architecture for our DQN is a fully connected neural network. This choice was influenced by the nature of the game environment and the requirement for a versatile, adaptable network structure. A single hidden layer of size 200 serves as a balance, offering the model the capacity to discern and learn intricate game strategies while maintaining computational efficiency. The ReLU activation function was chosen to introduce non-linearity, vital for approximating the non-linear Q-function associated with our chess scenario.

Adam, the optimizer of choice, is known for its adaptive learning rates, ensuring a harmonious balance between fast convergence and steady learning[7]. The hyperparameters such as the learning rate, discount factor, and epsilon decay were not arbitrarily chosen but were fine-tuned through a systematic process to best suit the specific requirements of the chess environment. Their adjustment, as elucidated in the experiment section, provides insights into the learning dynamics and adaptability of the model.

In essence, the algorithms and its architectural and hyper-parameter choices were consciously designed to ensure effective learning, optimal move selection, and robust adaptability to the intricacies of chess.

3.1.4 Comparative Metrics

For an objective assessment of our models, two main metrics are considered:

1. **Speed of Checkmate**: This metric measures the mean number of moves the model takes to deliver checkmate across a series of games. The aim here is not just victory, but efficiency, challenging the model to recognize and capitalize on strategic opportunities as quickly as possible. As in real life scenarios, it is very difficult to deliver checkmate under time pressure which leads to a draw or a losing result. This metric take care of this scenario, so the model can perform optimally under severe time pressure as well.

2. **Convergence to Optimal Reward**: This evaluates how many games the model must play before it consistently achieves the maximum possible reward (i.e., checkmate in the fewest moves). Rapid convergence is indicative of effective learning, while slower convergence may hint at the model's struggle to generalize its strategy across varied game states.

3.1.5 Evaluation Framework

Three distinct configurations of the DDQN model will be pitted against one another. Each will be assessed based on variations in key hyperparameters, reflecting different emphases on immediate vs. future rewards and exploration vs. exploitation balances. Subsequently, a standard DQN model, with default parameters, will be compared against the best-performing DDQN variant. This will provide insights into the relative advantages or shortcomings of each approach, in the context of achieving our primary objective of fastest checkmate.

3.1.6 Hyperparameter Selection and Training

The fine-tuning of hyperparameters was critical to the optimization of the model's performance. To rigorously evaluate the influence of various hyperparameter configurations, the training pipeline was executed in three distinct runs:

1. The initial experimentation phase indicated that a discount factor γ of 0.95 and a decay rate β of 10 yielded superior results. As such, these parameters were designated as the default settings for subsequent experimental phases.
2. The second run was conducted with a substantially reduced γ value of 0.05, to scrutinize the model's sensitivity to immediate rewards as opposed to long-term gains.
3. In the final run, the decay rate β was significantly increased to $10,000$, steering the agent toward reliance on established strategies at the expense of exploratory actions.

Motivation for Parameter Choice

The selection of hyperparameters were driven by both theoretical considerations and empirical validations. The discount factor γ of 0.95 was chosen based on its common use in reinforcement learning literature, signifying a balance between short-term and long-term rewards. Preliminary runs confirmed its efficacy in optimizing the model's performance within the constraints of the chess domain.

Conversely, the decay rate β was determined more experimentally. A value of 10 was initially used based on empirical results, showing a balanced approach between exploration and exploitation strategies for the agent. The extreme values of β in subsequent runs were intentionally chosen to study their impact on the model's reliance on known strategies versus exploration.

These parameter values were meticulously tested and validated, offering a robust setup for the broader scope of this research.

This study gained comparative insights into the performance of the Double Deep Q-Network (DDQN) across various hyperparameter settings, in comparison to the vanilla Deep Q-Network (DQN). For the second run of this experiment a comparison was conducted between the DQN and DDQN at default hyperparameter settings. This comparison served as a benchmark to evaluate the two classic Q-learning approaches the vanilla DQN and the DDQN.

The challenges faced were predominantly around hyperparameter tuning and striking an optimal balance between exploration and exploitation. The variations in results, based on different hyperparameter settings, underlined the importance of hyperparameter tuning.

3.2 Full-Game Implementation via Deep Reinforcement Learning (DRL)

This section explores the use of Deep Reinforcement Learning (DRL) in chess. The aim is to leverage DRL to transcend traditional search algorithms, enabling models to learn and adapt to complex game patterns. Despite the challenges brought on by the complexity of the game and the time-consuming nature of training, the experiment provides key insights regarding the viability of DRL in chess.

Two primary algorithms used are: Double Deep Q-Network (DDQN) and Monte Carlo Tree Search (MCTS). DDQN, first pioneered by DeepMind for Atari games, was selected due to its successful performance in initial 4x4 board experiments. It offers a stable and robust action-value function estimation, ideal for chess's complex state-action spaces. MCTS, a classical algorithm used in DeepMind's AlphaZero, serves as a comparative baseline. It balances exploration and exploitation effectively and has proven efficacy in games like chess.

The choice of these algorithms aims to evaluate the trade-offs between traditional search methods and DRL, offering a robust comparative framework for understanding each approach's merits and limitations in chess.

3.2.1 Objective

The primary objective of this experiment is to examine the viability of Deep Reinforcement Learning (DRL) as an approach for training chess engines. Given the extensive state-space and complicated strategic demands of chess, it is imperative to comprehend the capabilities and limitations of Deep Reinforcement Learning (DRL) in effectively capturing and exploiting these intricacies. Our goals include:

- Developing a chess engine using the DDQN algorithm to discern its capabilities in recognizing intricate game patterns and strategies.
- Comparing the performance of the trained models against a random player
- Comparing the performance of the DDQN-based engine against a traditional MCTS-based approach, to gauge the advantages and drawbacks of both methods in the context of chess.
- Drawing insights on the potential improvements and future directions for integrating DRL in board game AI, particularly for games as complex as chess.

With this experiment, we aim to bridge the gap between theory and practical implementation, providing a foundation for further exploration and improvements in the field of DRL for board games.

3.2.2 Setup for the DDQN Model

The Double Deep Q-Network (DDQN) serves as the core of our chess-playing engine. DDQN mitigates the overestimation bias inherent in traditional DQN by using two separate networks for approximating the Q-values[24]. The alignment between the complexities of chess and the DDQN model is crucial; hence, we outline the functionalities essential to the chess environment and their pivotal roles in facilitating DDQN's operation.

Environment

Our chess environment is structured to optimally accommodate the DDQN algorithm and features the following key functions:

1. **State Representation**: The state representation is designed to be directly ingestible by the neural network, capturing all pertinent information to facilitate rapid calculations.
2. **Action Execution**: The environment enables smooth execution of actions, translating them into corresponding state transitions and thus advancing the game.
3. **Rewarding Mechanism**: A well-defined reward structure accessible by the agent allows for immediate performance feedback, crucial for RL training.
4. **State Conversion**: The chessboard is translated into a tensor of shape $8 \times 8 \times 12$ through one-hot encoding. This facilitates seamless state updates.
5. **Action and Reward Logging**: The *step* function not only executes actions but also collates essential training data, including states, rewards, and terminal flags.
6. **Q-value Updates**: Unique to DDQN, our environment includes an internal mechanism for Q-value updating. The update follows the Double Q-Learning equation:

$$Q(s_t, a_t) = r(s, a) + \gamma Q'(s_{t+1}, \arg\max_a Q(s_{t+1}, a)) \tag{3.1}$$

   where $Q(s_t, a_t)$ and $Q'(s_{t+1}, a)$ are the Q-values predicted by the two different networks, $r(s, a)$ is the immediate reward, γ is the discount factor, and $\arg\max_a Q(s_{t+1}, a)$ selects the action that maximizes the Q-value in the next state s_{t+1}.

This structured and integrated environment ensures a robust platform for training the DDQN, harnessing its capabilities for mastering the game of chess.

Model Architecture

The core of our Double Deep Q-Network implementation lies within the Q-model class. This class is instrumental in facilitating interactions between the neural network and the chess environment. The architecture is similar to the one used in the deep mind paper. The key components and functionalities are :

1. **Network Architecture:** The neural network designed for our DDQN is a convolution network. It consists of three convolution layers, each activated by a ReLU (Rectified Linear Unit) function. The network culminates in a dense layer with 4096 neurons. This number is not arbitrary; it precisely maps to the potential moves in chess. Given 64 starting squares and 64 potential ending squares, we have a total of $64 \times 64 = 4096$ possible moves.

2. **Prediction Functionality:** This function commences by converting the chess board into a tensor. It then leverages the neural network to predict an output for the given input, which yields a probability distribution over all possible moves, ensuring their sum equals one. It is postulated that higher values in this distribution indicate the model's confidence in the corresponding moves being optimal. To ensure only valid moves are considered, a mask is applied to the distribution, effectively filtering out all illegal moves. Finally, a predefined dictionary translates the neural output into an actual chess move.

3. **Exploration Mechanism:** In the realm of reinforcement learning, exploration is paramount[15]. It ensures the agent doesn't fixate on local optima and seeks to understand the environment more holistically. The `explore` function embedded within our model facilitates this. It randomly selects legal moves to play initially. While this might introduce initial setbacks in the training process due to random actions, it holds the promise of escaping local optima and uncovering more effective strategies.

3.2.3 Setup for the Monte Carlo Tree search model

In this section, we outline the setup for the Monte Carlo Tree Search (MCTS) model. The Monte Carlo Tree Search (MCTS) algorithm is widely employed in the realm of game theory and puzzle-solving due to its notable efficacy in navigating intricate state spaces[4]. The selection of the Monte Carlo Tree Search (MCTS) algorithm for this research is predicated upon its remarkable capacity to effectively traverse highly intricate action spaces[20]. The objective of this arrangement is to assess the performance of Monte Carlo Tree Search (MCTS) in relation to the Double Deep Q-Network (DQN) approach in the context of training a chess engine.

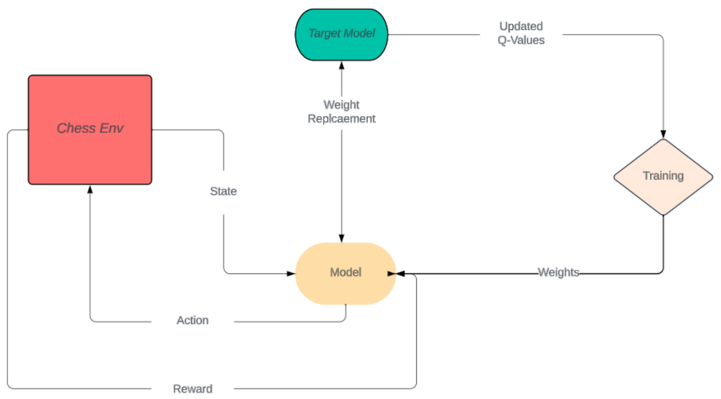

Figure 3.2: A visual representation of the DDQN program, mainly how the environment interacts between the model and the target model.

Environment

The chess environment functions as the fundamental framework of the Monte Carlo Tree Search (MCTS)-based chess engine, overseeing interactions and enabling strategic assessments within the game's context. The MCTS algorithm is effectively engaged through a collection of fundamental components that constitute this intricate ecosystem, thereby facilitating strategic exploration and intricate gameplay.

1. **Initialization and State Management:** The chess environment is defined at the start which is basically the state of the initial chess board. This current state of the environment is where all the interaction of the agent with the environment will occur and subsequently will lead to changes in the state of the board.

2. **Data Storage and Management:** As all the interaction occur in the state of the board, it will therefore be responsible to store the past board positions and the subsequent move outcomes. The MCTS will gather the data from here to make the next set of simulations.

3. **Action Execution:** Another important task of the environment is to execute a move on the chessboard.

4. **Reward Evaluation:** In reinforcement learning it is absolutely crucial to estimate the reward achieved from executing an action. In our scenario the MCTS provides a mechanism to assess the rewards generated from different actions.

5. **MCTS Exploration and Exploitation:**The Monte Carlo Tree Search (MCTS) algorithm plays a pivotal role in our system, as it effectively strikes a delicate balance between the exploration of uncharted paths and the exploitation of well-established, rewarding

paths. The search mechanism employed in the Monte Carlo Tree Search (MCTS) algorithm depends on the utilisation of the Upper Confidence Bound 1 (UCB1) formula to effectively determine the most suitable node for exploration in the search tree. The UCB1 criterion, which has been specifically designed for games, is expressed as:

$$UCB1(N, w, n) = \frac{w}{n} + C\sqrt{\frac{\ln N}{n}} \tag{3.2}$$

Where:

- $UCB1(N, w, n)$ denotes the Upper Confidence Bound used to determine the node's attractiveness for exploration.
- N is the total number of times the parent node has been visited.
- n is the frequency with which the current node has been visited.
- w stands for the total reward or the aggregate number of wins amassed from the current node.
- C is an exploration constant, steering the equilibrium between exploration and exploitation. In our implementation, this constant is dynamically adjusted using a decay rate, enhancing the algorithm's adaptiveness over time.

The underlying concept of this formula is combining the average reward of a node, denoted as $\frac{w}{n}$, with an exploration term, represented by $C\sqrt{\frac{\ln N}{n}}$. The combination of exploring interesting nodes and discovering possibly superior but less frequently visited nodes enhances the effectiveness of the search method resulting in a harmonious interplay.

The chess environment's synchronized interaction of these functions establishes a dynamic and responsive platform that enables the algorithm to explore strategic actions, learn from experiences and refining the overall strength to make good actions.

Model Architecture

The architecture of the neural network follows the AlphaZero model structure and comprises multiple components designed to function in a chess environment. The primary elements are:

1. **ConvBlock:** The initial layer is a convolutional block (ConvBlock) that accepts an input with 16 channels and performs convolution operations followed by batch normalization and ReLU activation. The number of filters in this block is defined as a parameter `num_filters`.

2. **Residual Blocks:** Following the ConvBlock, a series of residual blocks are employed to capture complex patterns and relations. Each block consists of two sets of convolution, batch normalization, and ReLU activation layers. The residual connection adds the

original input back to the output to aid in learning and mitigate the vanishing gradient problem.

3. **Value Head:** To estimate the value of the board position, the model includes a "Value Head" component. This section comprises a convolutional layer followed by batch normalization, ReLU activation, and two fully connected layers, culminating in a tanh activation to output values between -1 and 1.

4. **Policy Head:** To decide on the move to make, the model employs a "Policy Head." It consists of a convolutional layer, batch normalization, and ReLU activation, followed by a fully connected layer. The output dimension is 4608, corresponding to all possible chess moves.

5. **Loss Functions:** For training, two loss functions are used: Mean Squared Error (MSE) for the value head and Cross-Entropy Loss for the policy head. These are optimized during training to minimize the difference between the predicted and target values and policies, respectively.

The architecture allows for a variable number of residual blocks and filters, making it highly customizable. During inference, the network outputs both the board position value and a softmax probability distribution over possible moves, constrained by a mask representing legal moves. The algorithm and the architecture used is highly motivated from the deepmind paper.

This modular and extensible architecture allows the model to be finely tuned for different complexities and computational resources, while adequately balancing exploration and exploitation in the game of chess.

3.3 The AlphaZero Approach

This section delves into the exploration of implementing chess through the lens of the groundbreaking AlphaZero methodology. Rooted in the powerful amalgamation of deep learning and Monte Carlo Tree Search (MCTS), AlphaZero transcends established paradigms by autonomously mastering chess through self-driven discovery and strategy evolution[20]. This exploration ventures deep into the nuances and innovations that AlphaZero introduces within the intricate tapestry of chess. We will now discuss the important pillars of this approach.

3.3.1 Adapting AlphaZero's Methodology for Chess Engine Development

The AlphaZero paradigm combines Monte Carlo Tree Search (MCTS) with neural networks to achieve exceptional performance in complex board games, serving as an archetype for our own approach. This methodology seamlessly integrates MCTS-driven self-play for dataset generation, neural network training for enhanced decision-making, and iterative refinement through competitive self-evaluation.

- *MCTS-Guided Self-Play for Dataset Generation:* The cornerstone of AlphaZero's approach is the generation of high-quality game datasets through MCTS-guided self-play. In this phase, the MCTS algorithm uses prior probabilities provided by the current neural network to traverse the game tree, recording states, policies, and values along the way. This self-generated dataset serves as the raw material for subsequent neural network training.
- *Neural Network Training for Decision Refinement:* Utilizing the datasets acquired from MCTS-driven self-play, the neural network undergoes an intensive training regimen. This not only refines its decision-making capabilities but also improves its utility in guiding the MCTS in future self-play sessions. This symbiosis between the neural network and MCTS results in a powerful, iterative loop of constant improvement.
- *Iterative Model Evaluation and Refinement:* To maintain a state-of-the-art model, each freshly trained neural network is rigorously tested against its most successful predecessor in a series of matches where both sides employ MCTS. If the newer model wins at least 55% of the total games, it is declared the new baseline model for generating future datasets. This cyclical process ensures that the engine is continually refined, and it leverages computational resources for consistent and unyielding improvement.

Algorithm 4 AlphaZero Pipeline

```
1:  Initialize neural network NN
2:  Initialize previous best neural network BestNN with NN
3:  Initialize a repository for game datasets G
4:  while true do                                   ▷ Main iterative refinement loop
5:      G' ← MCTS self play                         ▷ Generate game datasets guided by BestNN
6:      Augment G with G'
7:      Train NN using the datasets from G          ▷ Neural network training
8:      winRate ← Evaluate(NN, BestNN)              ▷ Evaluate current NN against BestNN using MCTS
9:      if winRate  0.55 then
10:         BestNN ← NN                             ▷ Replace best model with the current model
11:     end if
12:     Optionally: Prune older datasets from G to manage memory
13: end while
14: return Final optimized neural network BestNN
```

3.4 Hybrid Learning Approach

In the pursuit of achieving a robust and efficient chess-playing model, a hybrid learning strategy can be adopted. This methodology combines the strengths of supervised learning, drawing knowledge from the rich CCRL dataset, combined with the dynamic exploration capabilities of the Monte Carlo Tree Search (MCTS).

As we studied in chapter 2, modern chess engines use handcrafted evaluation functions and deep search algorithms. Along with this they also rely on opening books and endgame table bases. Opening books grant engines a predefined set of opening moves, ensuring they embark on their games with theoretically sound footing. Meanwhile, endgame tablebases are vast repositories of endgame positions, allowing engines to play endgames with perfect precision when they fall within these predefined scenarios.

While these traditional methods have proven immensely effective, the advent of AI and neural networks in chess has spotlighted the potential of amalgamating supervised learning, which can rapidly harness the wisdom distilled in extensive game datasets, with reinforcement learning, which can fine-tune and evolve the model through self-play. This multi-pronged approach, leveraging both data-driven and algorithmic techniques, aims to foster a model that is not only well-informed but is also adaptable and innovative in its gameplay.

3.4.1 CCRL Dataset: Leveraging Expert Knowledge

The Computer Chess Rating Lists (CCRL) dataset is a compilation of games played by top-level chess engines. By training on these games, the model gains insights into expert-level strategies, tactics, and board evaluations. The neural network is exposed to diverse game situations, absorbing patterns that have proven to be successful in high-level play. The dataset consists of about 2.5 million top notch games. To comprehensively define a board state, it can be deconstructed into the following key aspects:

- **State**: This represents the current configuration of the board, detailing the position of each piece on the board, also whose turn it is to move.
- **Value**: The value offers an evaluation of the given state. Typically, it provides an estimation of the winning probability for the player, indicating the desirability of the current board configuration.
- **Policy**: This element furnishes a set of probable moves or actions that can be undertaken from the current state. It essentially maps out potential future trajectories for the game from the present state.

3.4.2 Monte Carlo Tree Search (MCTS): Exploration and Evaluation

Having a robust dataset is indispensable, but it's not the sole requirement for achieving mastery in a complex game of this magnitude. To make the model both time-efficient and astute, it is paramount to discern precisely which branches of the game tree should be dynamically expanded. Unlike traditional brute-force methods, MCTS selects these branches based on probabilistic evaluations. The salient phases of MCTS in the implementation include:

1. **Selection**: Navigate the game tree by selecting the move with the highest Upper Confidence bounds applied to Trees (UCT) value until a leaf node is reached. The UCT value is computed as given in Equation 3.2.

2. **Expand and Evaluate**: Evaluate the new board state, considering all legal moves unless the node is a checkmate or draw. The resulting evaluations are then assigned to the child nodes.

3. **Backpropagation**: After the evaluation, the algorithm traverses back through the visited nodes, updating their statistical values based on the leaf node's evaluation.

To optimize the search, the implementation of MCTS is run in parallel. However, naive parallelism can lead to redundant explorations. Thus, *Virtual Loss*, a technique that discourages multiple threads from exploring the same path concurrently is used. This technique will allow the model to search more board positions per second as the aim is to reduce any duplicate path searches.

3.4.3 Bringing it Together: The Hybrid Model

The hybrid approach aims to bridge the gap between pre-existing expert knowledge and on-the-fly decision-making. By pre-training on the CCRL dataset, the neural network acquires a foundational understanding of chess strategy. The goal is to incorporate the prowess of the MCTS algorithm which dynamically explores and evaluates board positions during gameplay. This symbiosis ensures that the model can not only replicate expert moves from the dataset but also innovate and adapt to novel game situations.

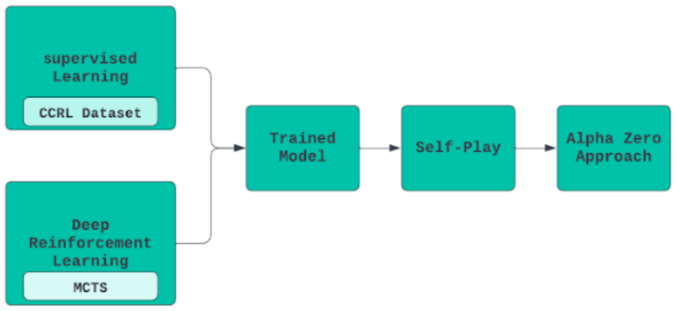

Figure 3.3: A visual representation of the Hybrid Approach

3.4.4 Evaluation Framework

The evaluation of the hybrid approach's performance is crucial for gauging its effectiveness and for validating its potential as a formidable chess engine. To maintain a focused and relevant evaluation, we exclusively employ head-to-head match-ups as the core framework for assessment:

1. **Head-to-Head Match-ups Against Established Engines**: The hybrid model is rigorously tested by pitting it against industry-leading chess engines like Komodo and

Stockfish. These engines, with estimated Elo ratings around 3200-2500, serve as the benchmark against which our hybrid model's capabilities are measured. Achieving favorable outcomes against such high-caliber engines would be indicative of the hybrid approach's prowess in creating a robust chess engine.

2. **Iteration-wise Self-Comparison**: Post-training on a new set of generated self-play datasets, the newly trained hybrid model is also matched against its previous iteration. This serves to measure the incremental gains achieved through additional training and allows for the identification of specific areas where the model has improved or may still need refinement.

3. **Exploring the Prowess of the Hybrid Approach with Human Dataset**: To delve deeper into the adaptability and effectiveness of the hybrid approach, a model is trained exclusively on a dataset comprising top human players, with an average Elo rating between 2500-2600. The model is then subjected to a refined self-play mechanism, in line with the hybrid approach, to further its training. The performance of this human-trained model is directly compared against the model trained on the CCRL dataset, which has a higher average Elo rating of 2900-3000. This experiment aims to evaluate the potential of the finely refined self-play mechanism in capturing the complex dynamics of chess, thereby shedding light on the true prowess of the hybrid approach.

The goal is to offer a thorough performance report of the hybrid model using these evaluation techniques, showing both its advantages and potential areas of improvement.

Chapter 4

Results and Discussion

4.1 Overview

This chapter focuses on the results achieved from the experiments described in the "Methodology" chapter. It aims to meticulously analyze and interpret the data, thereby providing a robust evaluation of the objectives laid out at the inception of this research.

Beginning with the simplistic 4x4 chessboard setting to assess the capabilities of Deep Q-Networks in determining the fastest route to checkmate, the chapter progresses to elaborate on more complex implementations. These include the deployment of Double Deep Q-Networks (DDQN) and Monte Carlo Tree Search (MCTS) on the standard 8x8 chessboard, followed by an exploration of the AlphaZero approach. Finally, a hybrid learning methodology incorporating a large CCRL dataset is examined to gauge its efficacy in expediting the training process and enhancing the performance of the chess-playing model.

Each section commences with a brief recapitulation of the objectives and experimental setup, followed by a structured presentation of the results. A comprehensive analysis ensues, aimed at dissecting the results to ascertain their relevance, implications, and any limitations therein. The chapter culminates in a comparative analysis across all experiments, focusing on performance metrics and computational resources, followed by a broader discussion that situates the findings within the larger academic landscape and suggests avenues for future research.

4.2 Chess Endgame via Deep Reinforcement Learning (DRL)

4.2.1 Objective

The principal objective of this section is to empirically evaluate the comparative effectiveness of Deep Q-Network (DQN) and Double Deep Q-Network (DDQN) algorithms in facilitating the fastest possible checkmate in a 4×4 chessboard endgame scenario. Through this endeavor, the aim is to unveil not only the optimal algorithm for achieving the fastest checkmate but also the underlying factors that contribute to the performance of each algorithm.

4.2.2 Hyper Parameter Tuning

Our experimental setup involved fine-tuning hyperparameters to examine the performance of Deep Q-Networks (DQN) and Double Deep Q-Networks (DDQN) in endgame scenarios. The hyperparameters included the discount factor of future rewards γ, the exploration rate ϵ, and the decay rate of the epsilon β, among others. Three separate runs were executed to investigate the behavior of the DDQN algorithm under different hyperparameter settings. These runs were configured to specifically assess the balance between exploration and exploitation, as well as immediate versus future rewards. The exploration rate is chosen as a default value of 0.1 and different values of β will be used for tuning.

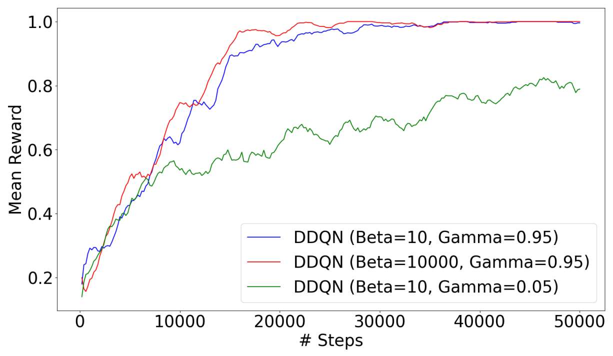

Figure 4.1: Performance evaluation of the DDQN algorithm based on mean reward over three different hyperparameter settings. The figure demonstrates the effects of different γ and β values on the algorithm's performance. The blue curve represents a model with a balanced γ of 0.95, showing consistent improvement. In contrast, the green curve for a myopic γ value of 0.05 stagnates at an average reward of 0.8, highlighting the negative impact of a short-sighted reward perspective. Similarly, Intriguingly, the red curve, trained with a higher β value, shows a mean reward almost identical to the blue curve. However, this setting results in a win rate of just 50%, suggesting that an excessively conservative strategy, while yielding comparable average rewards, does not translate into consistent wins.

4.2.3 Brief Recapitulation of Experimental Setup

The experimental setup involved a constrained 4×4 chessboard environment with White represented by a King and a Queen, and Black by only a King. Conforming to standard chess rules, each game commenced from a legal board position, ensuring that the Black King was not already in check. The environment operated under a move-limit of 20 moves per game to induce termination in the absence of a conclusive result.

This experimental design serves as a microcosm for understanding the applicability and effectiveness of reinforcement learning algorithms in complex game scenarios. It also sets the stage for our forthcoming detailed analysis on the efficiency, learning curves, and computational resources required by the DQN and DDQN algorithms in solving this particular endgame challenge.

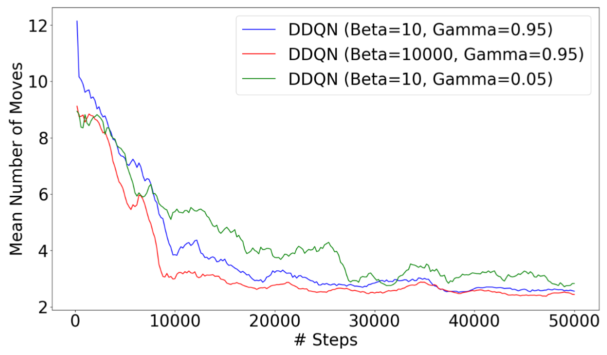

Figure 4.2: Performance evaluation of the DDQN algorithm based on the mean number of moves made under three different hyperparameter settings. The blue curve, representing the default DDQN with $\gamma = 0.95$, starts with a higher number of moves, suggesting an initial tendency for exploration and potentially evaluating strategies that require more moves but offer greater long-term rewards. The red curve with an increased β value shows the most efficient learning, achieving optimal results in the fewest training steps, possibly due to a greater focus on exploiting known successful strategies.

4.2.4 Impact of Hyperparameter Settings on DDQN

Figures 4.1 and 4.2 encapsulate the experimental outcomes for the DDQN algorithm under different settings of β and γ. It was observed that the default setting of hyperparameters ($\beta = 10$, $\gamma = 0.95$) allowed the network to converge to a global optimum after approximately 30,000 training steps. In stark contrast, reducing the γ parameter to 0.05 led to suboptimal performance, stagnating at an average reward of 0.8. Interestingly, the network trained with reduced exploration (increased β) displayed an initial surge in average rewards but failed to maintain this, resulting in only a 50% win rate eventually.

4.2.5 DDQN vs DQN

Theoretically DDQN has been shown to outperform DQN[24], through this experiment the aim was to prove this in the realm of chess. Utilizing the same default hyperparameter settings ($\beta = 10$, $\gamma = 0.95$) for both DQN and DDQN allowed for a controlled comparison, as depicted in Figures 4.3 and 4.4. DQN exhibited less effective learning dynamics, likely owing to an overestimation bias that led to unstable training and slower convergence. However, it showed promising trends in both average reward and move efficiency, indicating its potential to eventually master the optimal strategy.

4.2.6 Discussion

The key takeaways from these experiments are multifold. Firstly, the nuanced manipulation of hyperparameters like γ and β can have significant consequences on an agent's learning efficiency and the quality of the discovered strategies. A notable observation was that reducing γ significantly led to a less than optimal outcome. This can be attributed to the inherent

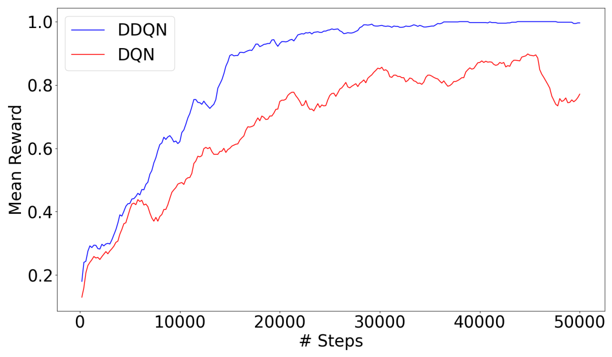

Figure 4.3: Comparative performance evaluation of DDQN and DQN based on mean reward at default hyperparameter settings. DDQN demonstrates superior performance, achieving a higher mean reward over the course of training. The increased stability and reduced overestimation bias is the most likely reason for this improved performance[24].

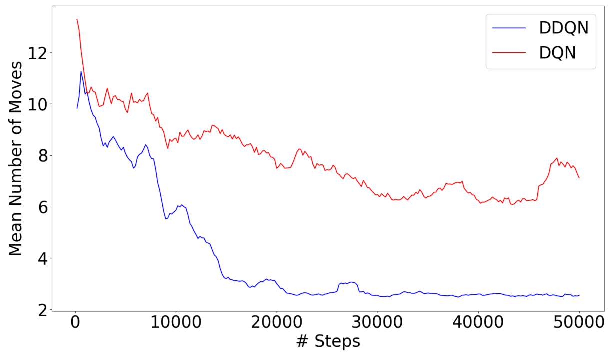

Figure 4.4: Comparative performance evaluation of DDQN and DQN based on the number of moves made at default hyperparameter settings. DDQN emerges as the more efficient algorithm, requiring fewer moves on average to achieve its goals. This efficiency can be attributed to the more accurate Q-value estimations provided by DDQN.

nature of chess, where short-term tactical gains often do not equate to long-term strategic advantages. In chess, optimal play often requires evaluating potential future rewards of different moves, rather than focusing solely on immediate gains. Consequently, a low γ value, which undermines the algorithm's capacity to account for future rewards, seems ill-suited for mastering a game as complex as chess.

Secondly, the DDQN algorithm's default settings appear to offer a balanced approach for the given task, efficiently leveraging both immediate and future rewards. Lastly, while DQN may not match DDQN's effectiveness in the short run, its positive trajectory suggests potential viability for endgame optimization given sufficient training time.

4.3 Full-Game Implementation via Deep Reinforcement Learning (DRL)

4.3.1 Objective and Brief Recap

The objective for this segment of the study was to assess the efficacy of Deep Reinforcement Learning (DRL) in chess. The algorithms used for this phase of study are Double Deep Q-Network (DDQN) and the traditional Monte Carlo Tree Search (MCTS). As chess is renowned for its vast state-space and intricate strategy requirements, understanding the strengths and weaknesses of DRL in capturing and leveraging these complexities is crucial.

The focus of this study is to scrutinize the performance of a chess engine built using DRL, examining its capabilities in capturing intricate game patterns and strategizing optimally in chess. To this end, the DRL models based on DDQN and MCTS will be tested and compared against each other. This comparative evaluation aims to deduce which model offers the most promising results and could thus be considered for the next set of experiments in the ongoing research.

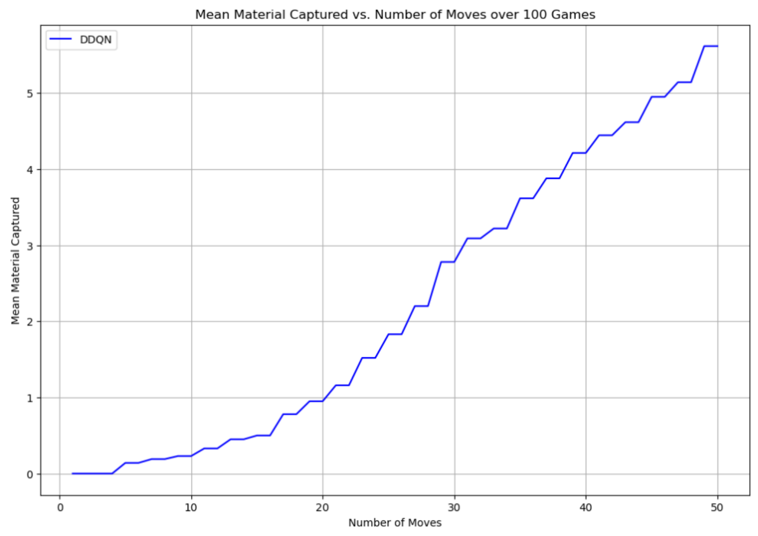

Figure 4.5: Mean material captured vs number of moves over 100 games for DDQN against a random opponent. Material capturing is quantified based on a pre-defined valuation of chess pieces given in chapter 2. The theoretical maximum material value that can be captured is 39, barring the opponent's king(Given we have 8 pawns, 2 bishops, 2 knights 2 rooks and a queen). In rare cases of pawn promotion the number can increase from 39. In the figure despite 50 moves being played, the DDQN agent captures an average material score of about 5.6, which is relatively low considering the random nature of the opponent's moves. This suggests that further optimization in the model's training or architecture is necessary, and additional training and computational resources may be required for performance improvement.

4.3.2 Hyperparameter Settings in DDQN Model

The DDQN model was trained with a carefully chosen set of hyperparameters to balance exploration and exploitation while also facilitating stable learning. Below are some of the critical hyperparameters:

- **Discount Factor (Gamma)**: Set to $\gamma = 0.99$ to heavily consider future rewards, which is vital for a strategic game like chess as discussed in the previous experimentation.
- **Epsilon Parameters**: Initially set at $\epsilon = 1.0$ and tapered down to a minimum of $\epsilon_{\min} = 0.1$ to balance exploration and exploitation.
- **Batch Size**: A batch size of 32 was selected for training the model.
- **Maximum Steps Per Episode**: Restricted to 100 steps per episode.
- **Optimizer**: Utilized the Adam optimizer with a learning rate of 0.00025 and a clip norm of 1.0.
- **Replay Buffer Size**: Set to 100,000 due to memory constraints, although larger sizes may be beneficial.
- **Update Frequencies**: The model was updated after every 4 actions, and the target network was updated every 100 actions.
- **Loss Function**: Employed the Huber loss for enhanced stability during training.

These hyperparameters were instrumental in shaping the learning process and performance of our DDQN model. Fine-tuning these could be a path for future research to optimize the model further.

Performance Evolution in DDQN The DDQN model's ability to capture opponent pieces, a critical metric in chess, was examined through a series of experiments. Figure 4.5 provides a focused view of this, where the DDQN model, in its initial stages, reaches a rather low mean material value of approximately 5.6 over 50 moves against a random opponent.

On progressing to Figure 4.6, the model shows some improvement but is far from optimized. At 100 training iterations, the mean material captured reached about 14.8, and at 500 iterations, it slightly improved to around 17.4 over 100 moves. These numerical insights corroborate that while the model is improving, its rate of improvement is moderate. The performance remains substantially below the theoretical maximum material value of 38 or the benchmarks set by more traditional algorithms like MCTS[4]. These observations underscore the DDQN model's current limitations in navigating chess's intricate state-space and complex strategic demands. While there is a discernible improvement in the material capturing ability, it remains significantly low. Given the modest rate of improvement, more extensive training could potentially enhance the model's performance. However, such an

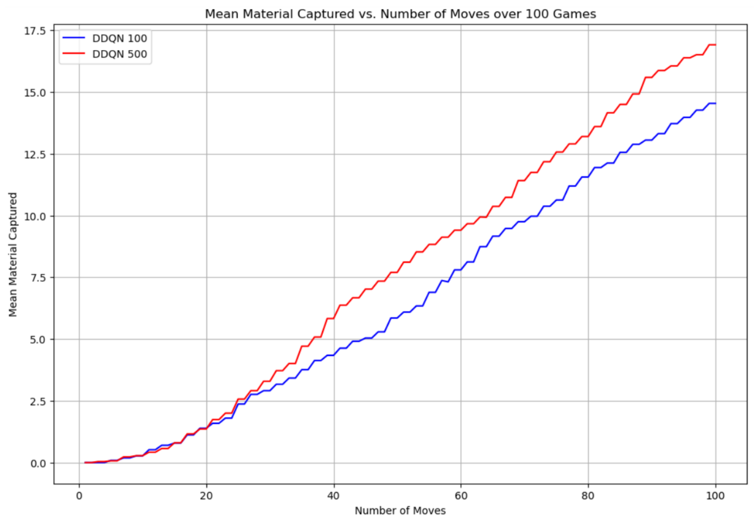

Figure 4.6: Comparison of mean material captured vs. number of moves over 100 games for DDQN at 100 and 500 training iterations against a random opponent. Despite increasing the training iterations from 100 to 500, the performance showed only a marginal improvement in mean material captured from about 14.8 to 17.4. This suggests that a substantially larger number of training iterations may be required to achieve significant performance gains.

approach would demand substantial computational resources and time for training, which is beyond the scope of this initial research study. In light of these findings, the research will now shift its focus toward an in-depth exploration of the Monte Carlo Tree Search (MCTS) algorithm.

4.3.3 Hyperparameter Settings in MCTS

In our implementation, various hyperparameters and settings were fine-tuned to optimize the MCTS model for computational efficiency and performance.

- **Starting Epsilon:** Set to 3 to control the level of exploration.
- **Minimum Epsilon:** Fixed at 1.5 to ensure a minimum level of exploration.
- **Game Progress Epsilon Decay:**

$$\epsilon = \max\left(\text{starting\_epsilon} - \text{game\_progress} \times 0.1, \text{min\_epsilon}\right) \tag{4.1}$$

  Here game progress is basically the number of move being played, the idea is to explore more in the initial stages of the game and to exploit more in the later stages. This is a common practice in RL for a chess environment.

- **Batch Size:** Set to 8 for mini-batch gradient descent.

- **Loss Function:** Mean Squared Error (MSE) is used for value loss, and a custom loss function for policy loss.
- **Optimizer:** Adam optimizer is employed for parameter updates.

These hyperparameters settings were carefully chosen to balance computational feasibility with the model's ability to effectively explore and exploit the game's state-space.

Table 4.1: Performance comparison of DDQN and MCTS models over 100 games against a random opponent.

Model	Wins	Losses	Draws
DDQN	2	2	96
MCTS	28	9	63

As illustrated by Table 4.1 and Figure 4.7, the Monte Carlo Tree Search (MCTS) algorithm shows a notably stronger performance in chess simulations compared to the DDQN model. Specifically, MCTS managed to secure 28 wins and only 9 losses, in stark contrast to the 2 wins and 2 loss exhibited by DDQN. This significant divergence is a testament to MCTS's more effective balance between exploration and exploitation, enabling it to execute well-informed and strategically superior moves. It should be noted that while the specific outcomes may vary slightly with each simulation run of 100 games, the overarching trends in performance remain consistent, reaffirming the relative robustness of MCTS compared to DDQN.

4.3.4 Discussion

The superiority of the Monte Carlo Tree Search (MCTS) algorithm over the DDQN model in chess simulations is evident from the results presented in Figure 4.7. Specifically, the MCTS algorithm outperforms the DDQN model in terms of the average amount of material captured over a span of 100 games against an opponent that makes random moves. The notable disparity can be ascribed to the advanced methodology employed by MCTS in effectively managing the trade-off between exploration and exploitation when making decisions. This fundamental characteristic significantly augments its capacity to make well-informed and strategically superior moves.

The adaptability of Monte Carlo Tree Search (MCTS) to intricate state-spaces and its capacity to contemplate a wider range of potential moves endow it with a competitive advantage, rendering it a more dependable contender for chess simulations within the confines of the present experimental settings. On the other hand, the DDQN model, although a worthy competitor, tends to falter in capturing material efficiently as observed in the figure. This suggests that while DDQN has its merits, MCTS offers a more optimized performance in the context of chess and could be the more suitable choice for further research and development.

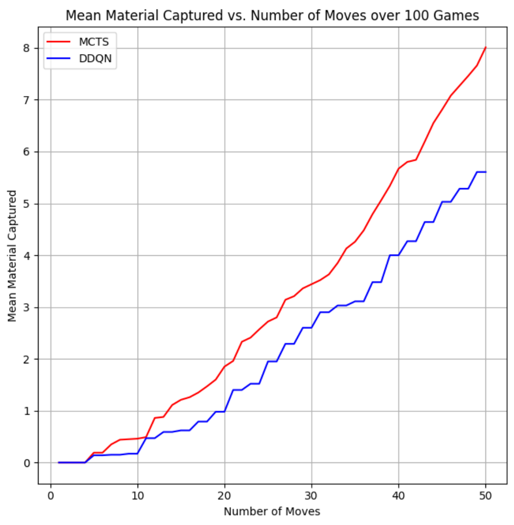

Figure 4.7: Comparison of Mean material captured vs Number of Moves over 100 games for MCTS and DDQN against a random opponent. MCTS outperforms DDQN, potentially due to its ability to perform deeper searches and make more optimal moves within the game tree.

4.4 The AlphaZero Approach

4.4.1 Objective and Brief Recap

The principal aim of this work was to replicate the groundbreaking methodologies put forth in the AlphaZero paper and to investigate its potential for Chess. To this end, models were trained with two different sets of iterations—Model 1 with 300 iterations and Model 2, which is essentially Model 1 fine-tuned with an additional 300 iterations of MCTS selfplay dataset.

Model	Wins	Losses	Draws
Model 1	4	8	88
Model 2	8	4	88

Table 4.2: Comparison of performance between Model 1 and Model 2 over 100 simulated games against each other. Model 2 emerged as victorious , moreover Model 2 was able to win more than 55% of the decisive games(8/12). Hence Model 2 will now replace Model 1 to generate selfplay games following the Alphazero pipeline.

4.4.2 Results

Initially, Model 1 underwent training on a 300-game self-play dataset. Subsequently, Model 2 was developed as an advanced iteration of Model 1 by further fine-tuning it through an additional 300 self-play games, ostensibly making it a more sophisticated version. Intriguingly, during head-to-head evaluations, Model 2 defied expectations by emerging as the victor, securing 8 wins as opposed to Model 1's 4. The majority of the games, accounting for 88%, culminated in a draw. This disparity in victories signals the incremental advantages gained through additional training.

4.4.3 Discussion

Pros The fact that Model 2, with extra training, outperformed Model 1 highlights the model's ability to improve and adapt with more training data and iterations. The AlphaZero framework recommends advancing to subsequent training iterations using the victorious model, in this case, Model 2, as the generator for new MCTS data sets. This iterative refinement substantially augments the model's capabilities, potentially resulting in dramatic performance improvements with each successive cycle.

Cons While the AlphaZero methodology offers a promising approach to game-playing AI, it comes with a considerable computational burden. For perspective, DeepMind's final iteration of AlphaZero was trained on an astonishing 44 million chess games only in 9 hours, a computational feat that is far beyond the reach of most individual researchers[20]. In contrast, in the experimentation done, training the model for just 100 games took approximately 2 hours. The additional training iterations for Model 2 did result in a performance boost but also increased the computational expenses substantially. Thus, while effective, this method may not be feasible for all scenarios, particularly those with limited computational resources.

Summary In summary, the AlphaZero approach has proven effective in developing a chess-playing agent that can adapt and improve with additional training. However, its computational expense remains a significant factor that must be considered for broader applicability.

4.5 Hybrid Learning Approach

4.5.1 Objective and Brief Recap

The overarching aim of melding the CCRL (Computer Chess Rating Lists) dataset with AlphaZero's approach is to capitalize on the virtues of both. The CCRL dataset, rich in expertly-curated chess games, can provide a strong foundation for the neural network. Simultaneously, AlphaZero's self-play mechanism and MCTS allow the model to evolve continually, fine-tuning its strategies to approach or surpass human-level performance. This amalgamation aims to minimize training time and computational resources while maximizing model performance and strategic depth.

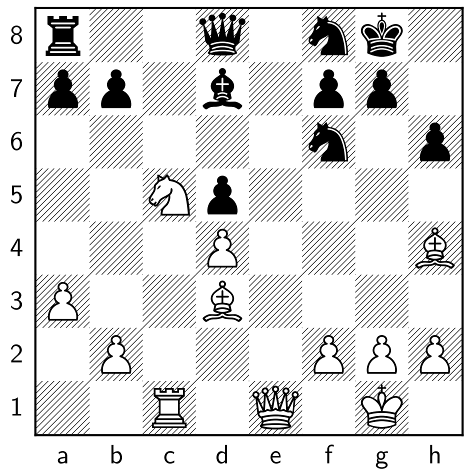

Figure 4.8: A pivotal middle-game position during a match between Model and Komodo24.

4.5.2 Results

Table 4.3 and 4.4 illustrates the performance of the hybrid model when pitted against various other high-level chess engines. These metrics highlight the competitive strength and robustness of our hybrid approach.

Color / ELO	Komodo25 (3200)	Komodo24 (2900)	Komodo23 (2700)
White	Draw	Win	Win
Black	Draw	Draw	Win

Table 4.3: Performance of the hybrid model against Komodo chess engines.

Color / ELO	Stockfish Level 8 (3000)	Stockfish Level 7 (2700)	Stockfish Level 6 (2500)
White	Draw	Win	Win
Black	Draw	Draw	Win

Table 4.4: Performance of the hybrid model against Stockfish chess engines.

As observed from the tables, the hybrid model demonstrates competitive performance, achieving a mix of wins and draws against chess engines with high ELO ratings. The model was specifically pitted against these opponents as these engines are easily available to play against and analyze the games on chess websites like chess.com and lichess.

4.5.3 Challenges and Limitations of the Hybrid Approach

One of the most challenging aspect of this research was to devise an optimal self-play mechanism for the Hybrid approach. It was absolutely essential to strike a perfect balance of exploration and exploitation otherwise the selfplay dataset may not be optimal for training as the games played will be of a mediocre quality. During the selfplay generation process on the

Comparison	Win	Loss	Draw
Hybrid Model vs Previous Iteration	5	2	3
Human-Inspired Hybrid Model vs CCRL Baseline Model	11	9	0

Table 4.5: Head-to-head match-up results comparing different training strategies. The first row (”Hybrid Model vs Previous Iteration”) showcases the performance of the model after a refinement cycle, using self-play and the Hybrid approach, pitted against its previous iteration. The second row (”Human-Inspired Hybrid Model vs CCRL Baseline Model”) illustrates the effectiveness of a model trained initially on human gameplay data which is basically around 2500 ELO (around 500 ELO points below CCRL data ratings), refined using the self-play mechanism of the Hybrid approach, and then tested against a model trained solely on the CCRL dataset. For self-play and model evaluation, an epsilon-greedy strategy was used with starting epsilon of 1.8 and a min epsilon of 0.5, the decay strategy used is defined by Equation 4.1. These settings were crucial for balanced exploration and exploitation. The results highlight the efficacy of our refined self-play mechanism, revealing the Hybrid Approach’s strength in mastering chess dynamics.

supervised train model(following the Hybrid approach), the games were constantly assessed for there quality using a stockfish engine manually. This was done to examine the trends of the game, choice of openings that model plays and key areas where the inaccuracy in move happen. These findings were then essential to setup the exploration hyperparameters and the epsilon decay methodology given by 4.1. As the Hybrid model incorporates a MCTS it has a limitation in the form of its move latency , the model takes about 120 seconds to make a move running at 1000 simulations whereas the other chess engines like the Stockfish and Komodo take 2-3 seconds.

4.5.4 Discussion

The hybrid model offered significant computational savings by initially training on the expert-curated dataset, thereby reducing the need for time-consuming self-play data generation. This accelerated the learning process, allowing the model to quickly reach a competent level of play. Implementing a self-play mechanism, especially after the application of supervised learning, that appropriately balances exploration and exploitation was a unique challenge. The design and fine-tuning of the decay mechanism for ϵ emerged as one of the most challenging aspects of this research. In terms of performance, the hybrid approach harnessed the dataset to provide a solid foundation in chess strategies and tactics. This was augmented by AlphaZero’s MCTS algorithm, guided by the UCB formula, which fine-tuned the model’s decisions between exploring new strategies and exploiting known ones. The result was a resource-efficient, strategically proficient model that combined the deep-rooted expertise from the supervised learning with the adaptive, self-improving capabilities of Reinforcement Learning. In summary, the hybrid model skillfully blends expert-guided learning with computational self-improvement, offering strategic depth in chess gameplay.

Chapter 5

Conclusions and Future work

The primary objective of this research was to explore the potential of applying Reinforcement Learning (RL) techniques to the domain of chess. The project traversed a journey that began with the application of training the model on endgames with limited complexity, scaling up to the full complexities of the actual game. Various RL algorithms and approaches such as Deep Q-Learning (DQN), Monte Carlo Tree Search (MCTS), and Double DQN (DDQN) were employed to train models capable of playing chess at different levels of expertise.

RL showed extreme potential in endgame scenarios, where both the DDQN and DQN model lived up to the task very well. The research conclusively demonstrated that the DDQN algorithm outperformed its DQN counterpart, aligning with theoretical expectations. This experiment also paved the path to selecting hyperparameters in a chess environment, basically enforcing the fact that future rewards must be considered for optimal success.

The limitations become apparent as the complexity of the game was increased to a full scale standard chess. It became evident that a mere few hundred training iterations were insufficient to capture the rich strategic and tactical complexities inherent in advanced chess play. The DDQN model which came out superior in the endgame scenario against DQN model struggled in this phase of the research. To address this, the Monte Carlo Tree Search (MCTS) algorithm was introduced, yielding noticeably improved performance.

The computational intensity of the AlphaZero approach posed significant challenges for this project's scope. To address this, a Hybrid Approach was introduced that seamlessly merged Deep Reinforcement Learning (DRL) with supervised learning techniques. The Hybrid approach performed exceptionally well, especially when pitted against other chess engines with max ELO rating of 3200. Some different training strategies were implemented to do experiments on the Hybrid model. The approach did present certain challenges and limitations, which will be addressed in the 'Future Work' section for further exploration and improvement. The approach presented in this paper is not limited to chess and can be seamlessly applied to other strategic games. Taking the game of "GO" for example. It has simple set of rules but has a more expansive game tree than that of chess[19][21]. With correct adaptation to the MCTS and the selfplay mechanism the methodology can be experimented. In summary, the insights garnered from this research have significantly enriched our understanding of how various reinforcement learning techniques can be effectively applied in the complex domain of chess. For a game as intricate and multi-dimensional as chess, relying on a singular algorithmic approach may fall short; a composite methodology offers a more robust path to mastery.

For future work, there are a few important things that can be tried to improve upon this research. First, we can try creating a model that utilizes both the DDQN and MCTS approach where the DDQN can further support the MCTS to traverse only the important nodes. This can reduce the models move latency.

Different supervised training techniques can be adopted for the Hybrid model. The training data used can be mixed up both in terms of volume and quality to examine the capability of the MCTS defined selfplay mechanism. As shown in chapter 4, a model trained on the human games dataset managed to go toe to toe with the standard CCRL datset model with the help of selplay refined training, more experiments can be done in this regard to harness the potential of RL.

There is always a scope of improvement in the Hybrid models self play mechanism. The mechanism can be further improved with some tuning in the exploration settings, this will result in a more solid foundation for the Hybrid model. The model's move latency, currently at 2 minutes for 1000 simulations, could potentially be reduced to 15 seconds with just 100 simulations, this can be done with more refined selfplay tuning after which the model will be able to deduce good moves from fewer simulations, provided that the self-play training is sufficiently optimized. Lastly, it was noticed that the Hybrid Model sometimes repeats moves in a winning position which eventually results in a 3 fold repetition, this is something that can be fixed by implementing a specific mechanism within the MCTS to detect and avoid such patterns.

Bibliography

[1] Bellman, R. A markovian decision process. *Journal of mathematics and mechanics* (1957), 679–684.

[2] Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., and Colton, S. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games 4*, 1 (2012), 1–43.

[3] Campbell, M., Hoane Jr, A. J., and Hsu, F.-h. Deep blue. *Artificial intelligence 134*, 1-2 (2002), 57–83.

[4] Coulom, R. Efficient selectivity and backup operators in monte-carlo tree search. In *International conference on computers and games* (2006), Springer, pp. 72–83.

[5] Hastings, W. K. Monte carlo sampling methods using markov chains and their applications.

[6] Hsu, F.-H. *Behind Deep Blue: Building the computer that defeated the world chess champion.* Princeton University Press, 2002.

[7] Kingma, D. P., and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[8] Kocsis, L., and Szepesvári, C. Bandit based monte-carlo planning. In *European conference on machine learning* (2006), Springer, pp. 282–293.

[9] Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems 25* (2012).

[10] Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. *Communications of the ACM 60*, 6 (2017), 84–90.

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

[12] Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971* (2015).

[13] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).

[14] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. Human-level control through deep reinforcement learning. *nature 518*, 7540 (2015), 529–533.

[15] Osband, I., Blundell, C., Pritzel, A., and Van Roy, B. Deep exploration via bootstrapped dqn. *Advances in neural information processing systems 29* (2016).

[16] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. Learning representations by back-propagating errors. *nature 323*, 6088 (1986), 533–536.

[17] Shannon, C. E. Xxii. programming a computer for playing chess. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science 41*, 314 (1950), 256–275.

[18] Silver, D. Reinforcement learning. *University Lecture, University College London* (2015).

[19] Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., et al. Mastering the game of go with deep neural networks and tree search. *nature 529*, 7587 (2016), 484–489.

[20] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815* (2017).

[21] Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., et al. Mastering the game of go without human knowledge. *nature 550*, 7676 (2017), 354–359.

[22] Sutton, R. S. Learning to predict by the methods of temporal differences. *Machine learning 3* (1988), 9–44.

[23] Sutton, R. S., and Barto, A. G. *Reinforcement Learning: An Introduction.* MIT Press, 2020.

[24] Van Hasselt, H., Guez, A., and Silver, D. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence* (2016), vol. 30.

[25] Watkins, C. J. C. H. Learning from delayed rewards.

Appendices

Appendix A

Programming Details

This section provides insight into the technical elements of the project. The project was primarily coded in python using PyTorch for neural networks. The code, along with all associated datasets is available on Google Drive at this link: https://drive.google.com/drive.

A.1 Chess Graphical User Interface (GUI)

A chess GUI was coded for this project. This GUI helped in conducting the games between the hybrid model and other chess engines as well as to play games against the trained model.

A.2 Self-Play and Model Evaluation

The self-play mechanism, which facilitates model training through controlled exploration using different parameters, was hand-coded to produce good quality games. This custom component handles move selection based on policy outputs. Custom-written code also assessed the model's evaluation metrics(material captured) as well as its ability to compete against various different trained models.

A.3 AlphaZero Pipeline

While the foundational MCTS and neural network architectures were derived from packages related to DeepMind's paper [20], these components underwent significant modifications and enhancements to better serve the goals of this research. The AlphaZero pipeline, which integrates these improved components, was coded to align closely with the specific objectives and requirements of this project.

A.4 Libraries and Packages

DQN and DDQN models were programmed using the Tianshou package. Pygame was used to develop the GUI and python-chess library for basic chess rules.