ENHANCED MONTE CARLO TREE SEARCH IN GAME-PLAYING AI: EVALUATING DEEPMIND'S ALGORITHMS

AMÉLIORATION DE LA RECHERCHE ARBORESCENTE MONTE CARLO DANS L'INTELLIGENCE ARTIFICIELLE POUR LES JEUX: ÉVALUATION DES ALGORITHMES DE DEEPMIND

A Thesis Submitted to the Division of Graduate Studies
of the Royal Military College of Canada
by

Karla Gonzalez, BSc

In Partial Fulfillment of the Requirements for the Degree of
Master of Science in Computer Science

August, 2023

To My Parents

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Dr. Francois Rivest, for his invaluable guidance and support throughout my thesis journey. His expertise and mentorship have been instrumental in shaping this research and my academic growth. Thank you to Dr. Haddad, Dr. de Boer, Dr. Sturgeon and Erin, your friendly faces, encouragement and advice really kept me going.

Además, estoy profundamente agradecida con mis padres, Carlos y Elsie, por su amor, creencia en mis habilidades y su ánimo y apoyo constante que me motivó a echarle ganas y seguir adelante.

A mi hermanita, Elsie. We will always be girls together. I can't imagine living in a world without you.

Finally I thank my God, my good Father, for guiding and blessing me throughout this journey. His unwavering love, mercy, and grace have provided me with strength, wisdom, and resilience to overcome challenges and pursue my academic endeavors. I will keep on trusting You for my future. Thank you, Lord.

Luke 1:37

Abstract

In the realm of Artificial Intelligence (AI), game-playing algorithms have reached remarkable heights of performance, exemplified by DeepMind's AlphaGo, AlphaZero, and MuZero algorithms. However, despite these achievements, there exists a need for a comprehensive evaluation and comparison of these algorithms, as well as an exploration of modifications to the underlying Monte Carlo Tree Search (MCTS) algorithm. This thesis represents a step forward in addressing these needs, aiming to shed light on the strengths and limitations of these state-of-the-art algorithms, the impact of modifications to MCTS, and their performance across diverse game environments.

The thesis first delves into the foundations of Reinforcement Learning (RL) and the MCTS algorithm, providing a solid understanding of the key concepts. It then proceeds to evaluate two modifications of the MCTS algorithm, Upper Confidence Bound for Trees (UCT) and Loss Avoidance. We found that both modifications, particularly the simultaneous implementation of both, improved not only MCTS in our selected environment, but also Alpha-Zero when used within its MCTS search.

Building upon this foundation, the thesis turns its attention to the imple-

mentation and evaluation of the DeepMind algorithms themselves: AlphaGo, AlphaZero, and MuZero. These algorithms are evaluated in a range of game environments, varying in complexity and size. By subjecting them to identical computational constraints and neural network architectures, we found that although Muzero is learning a model unlike it's sibling algorithms, overtime will outperform within fairly complex environments, such as Othello and Pinball.

Resume

Dans le domaine de l'intelligence artificielle, les algorithmes de jeu ont atteint des niveaux de performance remarquables, comme en témoignent les algorithmes AlphaGo, AlphaZero et MuZero de DeepMind. Cependant, malgré ces réalisations, il existe un besoin d'implémentation, d'évaluation et de comparaison complètes de ces algorithmes, ainsi que d'exploration des modifications apportées à l'algorithme sous-jacent de Recherche Arborescente Monte Carlo (MCTS). Cette thèse représente une avancée significative dans la réponse à ces besoins, visant à mettre en lumière les forces et les limites de ces algorithmes de pointe, l'impact des modifications apportées à l'algorithme MCTS, et leur performance dans divers environnements de jeu.

La thèse se penche d'abord sur les fondements de l'apprentissage par renforcement et de l'algorithme MCTS, ce qui permet de bien comprendre les concepts clés. Elle procède ensuite à l'évaluation de deux modifications de l'algorithme MCTS, Upper Confidence Bounds for Trees (UCT) et Loss Avoidance. Nous avons constaté que les deux modifications, en particulier l'implémentation simultanée des deux, ont amélioré non seulement MCTS dans notre environnement sélectionné, mais aussi AlphaZero lorsqu'il est utilisé

dans sa recherche MCTS.

Sur cette base, la thèse s'intéresse à la mise en œuvre et à l'évaluation des algorithmes DeepMind eux-mêmes : AlphaGo, AlphaZero et MuZero. Ces algorithmes sont évalués dans une série d'environnements de jeu, dont la complexité et la taille varient. En les soumettant à des contraintes de calcul et à des architectures de réseaux neuronaux identiques, nous avons constaté que, bien que Muzero apprenne un modèle différent de celui de ses autres algorithmes Alpha, les heures supplémentaires seront plus performantes dans des environnements assez complexes, comme Othello et Pinball.

Contents

List of Figures

List of Tables

Glossary

AI Artificial Intelligence. 1, 4, 19

CNN Convolutional Neural Network. 24

DNN Deep Neural Network. 2
DQN Deep Q-Network. 10, 71, 113
DRL Deep Reinforcement Learning. 1

LA Loss Avoidance. 4, 7, 32, 49

MB Model-Based. 18
MCTS Monte Carlo Tree Search. 2, 4, 19, 31
MDP Markov Decision Process. 12, 13
MF Model-Free. 18

NN Neural Network. 10, 16

PUCT Positional Upper Confidence Trees. 39

RL Reinforcement Learning. 4, 6

SL Supervised Learning. 42, 86

UCB Upper Confidence Bound. 46
UCT Upper Confidence Bound for Trees. 4, 7, 32, 46

1 Introduction

1.1 Motivation

In the world of Artificial Intelligence (AI), a new era has dawned, where machines have achieved what was once deemed unthinkable: defeating the greatest human minds in ancient and complex games. Over the years, AI has made significant strides in tackling games and achieving remarkable performance. One notable milestone was Deep Blue, a chess-playing computer program [1]. Deep Blue defeated the reigning world chess champion, Garry Kasparov, marking a breakthrough in AI's capabilities in strategic games. Building on this success, Allis, Uiterwijk, and van Rijswijck's work [2] explored searching for solutions in games and AI, laying the groundwork for more advanced algorithms. Mnih et al. [3] demonstrated a groundbreaking achievement in Deep Reinforcement Learning (DRL) with their paper on "Human-level control through deep reinforcement learning." Their work showcased how AI can learn from raw pixel inputs to achieve human-level performance in various challenging games. These developments illustrate the tremendous progress AI has made in understanding and mastering complex game environments,

paving the way for further advancements and real-world applications.

Enter the era of AlphaGo, AlphaZero, and MuZero, groundbreaking algorithms that have captivated the world with their remarkable achievements within these games [4]. Within this thesis we will refer to the entire group of them as the Alpha algorithms. These Alphas possess an unparalleled ability to learn and strategize, pushing the boundaries of what we believed machines were capable of. With a mix of Deep Neural Network (DNN)s , self-play, and powerful Monte Carlo Tree Search (MCTS) techniques, these algorithms have shattered long-standing records, revealing a glimpse of the astonishing potential of AI [5, 6, 7]. However, as we venture further into the realm of game-playing algorithms, it becomes clear that further exploration and evaluation are necessary to understand the full potential of these algorithms in different game environments.

The advancements achieved by AlphaGo and AlphaZero algorithms marked significant milestones, showcasing remarkable accomplishments in game-playing domains. Nonetheless, MuZero exhibited an unprecedented level of mastery, surpassing the performance of the aforementioned predecessors. The transformative impact of these algorithms has been widely acknowledged [8, 9], generating substantial interest research and application efforts [10, 11, 12, 13]. Here we list a few of the open research questions which arose with the publication of the Alpha's papers:

- Can the self-play training approach of AlphaZero and MuZero be generalized to domains beyond board games?
- How can the exploration-exploitation trade-off in the MCTS component of AlphaZero and MuZero be optimized for different types of environ-

ments?

- What is the impact of various hyperparameters, such as the number of simulations and the exploration parameter, on the performance of AlphaZero and MuZero?
- Can AlphaZero and MuZero be adapted to handle games or decision-making problems with continuous action spaces?
- How can the learned policies and value functions from AlphaZero and MuZero be transferred or generalized to new, unseen environments?
- What are the limitations of AlphaZero and MuZero in terms of scalability and computational requirements, and how can they be mitigated?
- Is the success of these algorithms able to be replicated without the support of detail from the original papers?
- Is the astounding performance demonstrated by these algorithms solely due to the extreme computational resources allotted to them or is there a scientifically observable and reproducible improvement attributed to another contribution?
- Can AlphaZero and MuZero be integrated with other RL algorithms or techniques to further enhance their performance?

These research questions highlight some of the theoretical aspects that researchers have explored or are yet to explore in relation to AlphaZero and MuZero, and do not cover the large group of practical applications of these algorithms which have been tackled by researchers [14, 11]. These inherent mysteries surrounding the algorithm, combined with its impressive performance, continue to motivate further exploration. By delving into these algorithms and their enigmatic nature, we aim to expand our understanding of

their mechanisms, paving the way for continued advancements in the field of AI.

Unfortunately, there are a number of obstacles still present for the accessibility, understanding, and usage of these algorithms. The propriety nature of the Alpha's limits their public accessibility, and their exceptional success can be attributed, in part, to the substantial computational resources they require, which are often unattainable for smaller research labs. As a result, different researchers have sought to overcome these barriers by developing various derived and pieced-together versions of these algorithms [15, 16]. These efforts aim to bridge the knowledge gaps and replicate the remarkable performance achieved by the original algorithms.

1.2 Problem Statement and Limitations

For this specific work, we cannot answer all the questions set forth but we do focus on two main components:

Firstly, MCTS stands as a pivotal component within the Alpha family of algorithms, playing a crucial role in their remarkable achievements. MCTS offers a systematic approach to search and decision-making in large and complex domains, enabling the algorithms to navigate expansive state spaces effectively. The inclusion of MCTS in these algorithms has yielded significant breakthroughs, propelling the boundaries of AI and game-playing capabilities. However, throughout the evaluation and evolution of the Alpha algorithms, the underlying MCTS algorithm has remained relatively unchanged. This presents a great opportunity to explore the impact of modifications to the

MCTS algorithm and investigate their effectiveness in improving the performance of these state-of-the-art game-playing agents. Additionally, there is a need for a comprehensive evaluation and comparison of these algorithms in different game environments, considering factors such as computational constraints and adaptability. The utilization of MCTS also raises intriguing scientific questions and challenges

We begin by exploring, the integration of the enhancements to the MCTS algorithm in both sections of the thesis provides an opportunity to explore the impact of these modifications on algorithmic performance. By evaluating the modified MCTS algorithm against the DeepMind algorithms, important questions arise:

- What is the impact of different modifications and variations of the MCTS component within the AlphaZero algorithm?
- How do these modifications affect the search efficiency and decision-making process?
- Can the modified MCTS algorithm achieve performance comparable to the implemented AlphaZero?

The exploration of these questions not only advances our understanding of the MCTS algorithm but also presents potential avenues for improving game-playing agents in terms of speed, efficiency, and decision quality.

Second, in this investigation, we aim to address a range of scientific questions to advance our understanding of the Alpha algorithms and their underlying principles. Specifically, we seek to explore:

- Is it feasible and possible to implement a framework which incorporates

the different Alphas algorithms such that we are able to conduct experiments with different size and complexity of environments?

- How do the Alpha algorithms perform when subjected to the same computational constraints?

By conducting a fair and rigorous comparison, this research seeks to unravel insights into the capabilities of these algorithms in different game contexts, shedding light on their general performance, adaptability and generalization abilities.

While this thesis makes its contributions to the field of game-playing AI algorithms and provides valuable insights into the capabilities and limitations of the DeepMind algorithms and MCTS modifications, there are certain limitations that need to be acknowledged. Firstly, the evaluation and comparison are primarily focused on zero-sum game environments, which may limit the generalizability of the findings to more complex and diverse scenarios. Additionally, due to computational constraints and training time, the algorithms' performance might not have reached their full potential. Furthermore, the choice of gaming environments might not fully represent the vast diversity and complexity of real-world problems, necessitating further exploration in more challenging domains. Moreover, while MuZero exhibits remarkable adaptability, its training regime might be challenging to replicate in resource-constrained scenarios, raising questions about its practical applicability in certain real-world applications. Despite these limitations, the thesis serves as a solid foundation for future research and opens avenues for further investigations in Reinforcement Learning (RL) and AI domains.

1.3 Solution Overview

The research begins by focusing on the evaluation and comparison of MCTS algorithms, including the original form and variations such as Upper Confidence Bound for Trees (UCT) and Loss Avoidance (LA)within MCTS the AlphaZero algorithm. These evaluations occur within well known board game environments, specifically Othello and Connect4. Which were chosen for their specific differences in difficulty, complexity and input dimensions. This work aims to pinpoint the strengths or weaknesses which arise from the variations within specific environments, while analyzing performance changes for each distinct modification.

Building upon the insights gained from the MCTS exploration, the thesis proceeds to re-implement and evaluate the Alpha algorithms in various game environments. Through a systematic comparison, the algorithms are evaluated in their original forms, considering their relative strengths and weaknesses. By subjecting them to the same computational constraints, the objective is to conduct a fair assessment is conducted to uncover their performance differences. This comparative analysis provides valuable insights into the effectiveness of each algorithm and their applicability in different strategic decision-making contexts.

1.4 Contribution

The contributions of this thesis are the following:

1. The implementation of the AlphaZero algorithm. While the original

paper provided limited details about the algorithm of the undisclosed code, we built upon existing research to create a modified version that can play on the environments of Connect4 and Othello.

2. Evaluate and compare the performance of the replicated algorithm in different environments, such as Othello and Connect4. Results suggest a simultaneous implementation of the enhancements produce a strong agent AlphaZero-ALL, which consistently outperforms the other implemented algorithms and baselines.
3. Evaluate the modifications to the MCTS algorithm proposed in previous research and analyze their potential impact on the AlphaZero's performance. Determined that AlphaZero-LA may be more suitable for certain games, and that tailoring the modifications to an algorithm can significantly influence its performance.
4. Replicated the other original Alphas algorithms (AlphaGo and Muzero) given the limited available information within the published papers within a variety of different environments: Connect4, Othello, Pong and Pinball.
5. A fair and consistent evaluation of individual algorithms, AlphaGo, AlphaZero and MuZero, agents under similarly controlled GPU and CPU computational resources.
6. Conducted evaluations to analyze and compare the performance and efficacy of the replicated algorithms in different game environments, particularly Othello, Connect4, Pong and Pinball. Determined that the learned model MCTS approach of Muzero outperforms the other Alphas and implemented benchmarks.

By addressing these scientific questions and providing comprehensive evaluations and comparisons, this thesis contributes to the field of ML and intelligent decision-making systems. The findings and insights gained from these investigations have the potential to inform the development of future game-playing agents, enhance their performance in diverse game environments, and pave the way for further research in strategic decision-making and AI algorithms.

1.5 Thesis Outline

This thesis is split into three main sections.

Chapter 2 serves as a comprehensive exposition of the essential background information required to comprehend the problems addressed in this thesis. This chapter offers a review of recent advancements in the pertinent domains of machine learning and RL. The literature review of this thesis is found in the different corresponding sections and is introduced as the related topics appear. This was done to avoid confusion and overload and provide an easy read.

In Chapter 3, we delve into the implementation, analysis, and comparison of various modifications to the MCTS algorithm within the context of the AlphaZero framework. We explore the integration of the traditional UCT algorithm as a baseline and within the AlphaZero algorithm, as well as the incorporation of LA strategies to enhance performance. Through a series of experiments on Connect4 and Othello, we evaluate and compare the performance of these modifications against random, greedy, and minimax algorithms. Our

findings sustain valuable insights into the potential of these enhancements and their impact on the performance of MCTS-based algorithms.

Chapter 4 contributes to the field by providing a comprehensive evaluation of the AlphaGo, AlphaZero, MuZero algorithms, and Deep Q-Network (DQN) baseline on diverse environments, including Connect4, Othello, Pong, and Pinball. By training all neural networks (NN) under the same computational constraints, we aim to conduct a fair evaluation. Our analysis elucidates the nuances and improvements of MuZero, demonstrating its superiority and highlighting its strengths beyond computational requirements.

In Chapter 5, we present a comprehensive summary of our work, encapsulating the key findings and contributions of our research. We discuss the implications and significance of our results in relation to the research objectives, providing insights into the broader implications of our study. Additionally, we examine the limitations of our experiments, addressing potential constraints, challenges, and constraints that may have influenced our findings. Additionally, we highlight potential future directions and avenues for further exploration, suggesting areas where our research can be extended or improved. This chapter serves as a thoughtful reflection on the outcomes of our research, offering valuable insights into the scope and boundaries of our study.

2 Background

In this chapter we will introduce the fundamental concepts needed to comprehensively understand this thesis. First, this section covers the field of RL and the an array of subtopics pertaining to model learning, and planning. These subfields of machine learning will provide the necessary insignt and the primary tools which are used throughout this thesis.

2.1 Reinforcement Learning

Reinforcement learning (RL) is a type of machine learning that enables agents to learn how to make decisions in complex and dynamic environments, such as video or board games, through trial and error [17]. The idea is to allow an agent to iteratively take actions within these environments, and allow it to learn through a scalar reward signal returned by the environment as a response to the actions the agent selected. These reward signals will act as a positive or negative "reinforcement" and provide associated feedback to the agent. From this feedback, an agent may learn a policy (a strategy or rule which maps states to actions which an agent may take in each state) from scratch. The goal of RL is to learn a policy that maximizes the expected cumulative reward

over time.

2.1.1 Markov Decision Process

In RL there is constant interaction between an agent and its environment. Markov Decision Process (MDP) are fundamental mathematical models used in the field of RL to represent sequential decision-making problems. Many sequential decision scenarios have the Markov property where an agent's decision only depends on the current state s. These decision scenarios can be defined by the five components $(\mathcal{S}, \mathcal{A}, P, R, \gamma)$ [17]:

$\mathcal{S}$: a finite set of states

$\mathcal{A}$: a finite set of actions

P : for each triplet of state, action, and next state (s, a, s'), the transition probability $P(s'|s, a)$ gives the probability that the system will transition to state s' if action a is taken in state s.

R : for each triplet of state, action, and next state (s, a, s'), a reward function, defined as $R(s, a, s') = \mathbb{E}[r_{t+1}|S_t = s, A_t = a, S_{t+1} = s']$

$\gamma \in [0,1]$: a discount factor of future rewards.

At each time step $t \in [0, 1, 2, ...]$, the agent is presented with a depiction of the current environment's state, represented as $s_t \in \mathcal{S}$. Based on this, it opts for an action, denoted as $a_t \in \mathcal{A}$. As an outcome of its chosen action and after the lapse of one-time step, the agent is given a real numerical reward, denoted as $r_{t+1} \in \mathbb{R}$. Subsequently, the agent transitions into an updated state, denoted as s_{t+1}. This process will continue indefinitely or until a terminal state is reached at time step T. Within this description we may also include a set

of at least one possible initial states $\mathcal{I} \subseteq \mathcal{S}$ and a set of zero or more terminal states $\mathcal{T} \subseteq \mathcal{S}$.

The critical property of MDPs is called the "Markov Property". It states that the system dynamics, P and R, at time step t will only depend on the current state s_t and action a_t which don't require any history or prior knowledge from the environment to be determined. This is expressed as such:

$$P(s'|s_t, a_t) = P(s'|s_t, a_t, s_{t-1}, a_{t-1}, s_{t-2}, a_{t-2}, ..., s_0, a_0) \tag{2.1}$$

In RL, the behaviour of an agent is defined by a policy, denoted as π, which can be either deterministic or probabilistic. This policy dictates the probability for the action, a_t, that an agent takes in its current state, s_t, by establishing a probability distribution over all feasible actions, defined as $\pi(a|s) = \mathbb{P}[A_t = a|S_t = s]$. Future rewards will then be discounted using γ such that the agent will value immediate rewards higher than those in future. The cumulative reward is typically defined as the sum of discounted rewards over a finite or infinite time horizon. We must distinguish this from the actual return, defined below:

$$G_t = \sum_{i=0}^{\infty} \gamma^i r_{t+i} \tag{2.2}$$

where R_t is the discounted return at time t, γ is the discount factor, and r_t is the reward at time t.

The discount factor plays a crucial role in determining the significance of future rewards compared to immediate rewards. The goal of RL achieved through MDPs is to learn the optimal policy π^* (if always followed) will give

the best outcome over any other policy π which could be possibly played.

$$\pi^* = \arg\max_{\pi} \mathbb{E}[\sum_{t=1}^{\infty} \gamma^t r_t | a_t \sim \pi(a_t|s_t)] \tag{2.3}$$

2.1.2 V-values and Q-values

In RL, reward measures how beneficial an action is in a given state. Often, we look at the total of future rewards, adjusted to be less valuable the further in the future they are, to help improve the agent's decisions or strategy. We refer to two crucial value functions associated with an MDP. The first, known as the state-value function $V_\pi(s)$, which represents the expected return when the agent starts from state s at time t and follows policy π. Its formal definition is:

$$V_\pi(s) := \mathbb{E}_\pi[G_t|s_t = s] \tag{2.4}$$

The second, important function we must refer to, is the action-value function:

$$Q_\pi(s, a) := \mathbb{E}_\pi[G_t|s_t = s, a_t = a] \tag{2.5}$$

While $V_\pi(s)$ represents the expected value gained by a policy after being in a state s, $Q_\pi(s, a)$ signifies the expected value a policy will gain after taking an action a in state s. The definitions differ only slightly, where $Q_\pi(s, a)$ is defined more specifically, over a specific action a.

We may represent $V^\pi(s)$ as the weighted sum of $Q_\pi(s, a)$ for all valid actions in the state s under policy π.

$$V^{\pi}(s) = \sum_{a \in \mathcal{A}} P_{\pi}(a|s) Q(s,a) \tag{2.6}$$

2.1.3 Value-Based and Policy Gradient Methods

There are two main subsections of RL, value based methods and policy gradient methods.

Value-based methods in RL aim to predict the scalar rewards that an agent receives after taking an action. The policy then guides the agent to select its next action based on the predicted highest expected reward or by selecting an action based on the predicted values. Learning to accurately predict these values, represented as Q or V, can be highly challenging.. Depending on the complexity of the environment, a large number of trajectories would need to be explored in order to be able to then generate accurate predictions. Value estimation is crucial for an agent to make decisions about which actions to take. It allows the agent to estimate the long-term return of a state or action, thereby guiding its decisions about which actions to take.

Typically, these algorithms are optimized as they minimize the discrepancy between the predicted Q-value $\hat{Q}^{\pi}(s,a)$ and the estimated action-discounted return $G^{\pi}(s,a)$ for the given policy. This is given by the following [18]:

$$L_t^{value} = (\hat{Q}^{\pi}(s_t, a_t) - G^{\pi}(s_t, a_t))^2 \tag{2.7}$$

The estimated return is obtained through a blend of sampled future rewards under policy π and, in certain cases, an estimation of Q_{π^*} at a future state. Among these techniques, Q-learning stands out as the most prevalent

and influential, shown in Equation 2.8 serving as the foundation upon which other methods are built.

$$Q_{\pi^*}(s_t, a_t) = \mathbb{E}[r_t + \gamma \max_{a'} Q_{\pi^*}(s_t, a')] \tag{2.8}$$

where r_t is the reward obtained by taking an action a_t in the state s_t.

Policy gradient methods are the second category of RL, and the one utilized time and time again in this thesis. These techniques directly enhance a stochastic policy by adjusting the action probabilities using the gradients of those probabilities concerning a loss function. [19]. In truth, policy gradient algorithms work to optimize the parameters θ, often represented by a Neural Network (NN), of a differentiable, stochastic policy π_θ. Aiming to maximize the expected return (discounted or not):

$$J(\pi_\theta) = \mathbb{E}[(G_t)] \tag{2.9}$$

We then want to optimize the policy using gradient descent method used, using repeated applications of stochastic ascent to approximate it:

$$\theta_{k+1} = \theta_k + \alpha \nabla_\theta J(\pi_\theta)|_{\theta_k} \tag{2.10}$$

and

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{s_t, a_t, s_{t+1\ldots} \sim \pi_\theta}[\sum_{t=}^{T} G^\pi(s_t, a_t) \nabla_\theta log \pi_\theta(a_t|s_t)] \tag{2.11}$$

With a specified learning rate α and the estimated discounted reward for the policy at each step of a trajectory $G^\pi(s_t, a_t)$, we can proceed.

The algorithms which optimize the policy this way are called policy gradient algorithms. Policy gradients can also be referred to as actor-critic methods, directly updating the policy controlling the actions the agent takes. Those will be covered in section 2.3.4.

2.1.4 On-Policy and Off-Policy

The training process chosen for each RL algorithm is an important way to categorize them, as this is how they interact with the data used to learn. Typically, RL algorithms undergo a process of cycling between gathering information from their environment and modifying their behavior or policy using the acquired data.

We split these algorithms into two main categories. Understanding their differences and how it affects them is a crucial part of the study of these algorithms. In on-policy algorithms, the policy being updated is the same policy that generates the data used for training. Conversely, off-policy algorithms update a policy that was not responsible for generating the data. For instance, Q-learning employs an ϵ-greedy policy during training, which includes random actions for exploration purposes. However, Q-learning uses an argmax operation to learn the optimal policy's value, thus considered an off-policy algorithm. Off-policy means you are learning/updating a policy that is different than the one you play.

Off-policy algorithms offer the advantage of incorporating exploration without directly impacting the policy itself, while on-policy algorithms need to integrate exploration into their policy directly. Various strategies, such as encouraging high entropy or adding noise to the policy, can be employed to

facilitate exploration in on-policy algorithms. Notably, exploration plays a vital role in the success of on-policy algorithms, and studies have emphasized its significance [20]. Moreover, recent empirical investigations have emphasized the significance of policy initialization in the on-policy algorithms' performance. These studies reveal that a poorly initialized policy can hamper exploration and hinder the learning progress [21].

2.1.5 Model-Based v. Model-Free

Model-Based and Model-Free RL are two fundamental approaches to solve sequential decision-making problems. In model-based RL, agents maintain an internal model of the environment to predict how the world evolves based on actions and states. This model allows them to plan ahead and simulate possible trajectories, enabling them to make informed decisions and optimize their policies. Notable examples include MCTS [22] and AlphaZero [6], which require a model, and DynaQ [23] and MuZero [24], which combine learning from real experience with learning from simulated experience using a learned model.

On the other hand, model-free RL algorithms directly learn from real experiences without constructing an explicit model of the environment. They rely solely on observed interactions with the world to optimize their policies. Notable model-free algorithms include Q-learning [19], DQN [25] and SARSA [19, 26], which iteratively update their value functions based on the observed rewards and transitions.

Both model-based and model-free RL have their unique strengths and weaknesses. Model-based methods tend to be more sample-efficient and can

make optimal decisions in fewer interactions with the environment. However, they might suffer from errors and inaccuracies in the learned model, leading to suboptimal performance in complex or uncertain environments. On the contrary, model-free methods do not require an accurate model, making them more robust in such scenarios. Nevertheless, they might need a significant number of interactions with the environment to converge to an optimal policy.

Recent research has focused on combining the strengths of both approaches in hybrid algorithms, such as DynaQ+ and Model-based Value Expansion. These methods aim to leverage the benefits of model-based planning while addressing the challenges of model inaccuracies and high computational costs. By finding the right balance between model-based and model-free learning, these hybrid approaches offer a promising direction to improve the efficiency and effectiveness of RL algorithms in various real-world applications.

2.1.6 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a powerful and widely used algorithm in the field of artificial intelligence AI, particularly in the context of games and decision-making problems. It belongs to the class of model-based RL algorithms, which leverage an internal model or simulation of the environment to guide decision-making. MCTS is known for its ability to efficiently explore large state spaces and find near-optimal solutions by combining tree search and Monte Carlo sampling techniques.

At its core, MCTS uses a search tree data structure to represent possible actions and states of the environment. The algorithm iteratively expands the tree by simulating episodes of play, known as rollouts or playouts, from the

current state to explore different trajectories. During each iteration, MCTS selects actions based on a balance between exploitation (choosing actions that seem to be promising based on the current knowledge) and exploration (trying out new actions to discover potentially better alternatives). The exploration can be guided by a prior probability distribution derived from the NN or other learned models.

2.2 Exploration vs. Exploitation Trade-off

Exploration versus exploitation is a fundamental challenge in RL algorithms. In RL, agents aim to maximize their long-term rewards by interacting with an environment. Exploration involves trying out different actions to gather information about the environment and discover potentially better strategies. On the other hand, exploitation focuses on exploiting the current knowledge to select actions that are believed to yield the highest immediate rewards based on past experiences. Striking the right balance between exploration and exploitation is crucial for finding optimal policies. Insufficient exploration may lead to suboptimal solutions, while excessive exploration can be inefficient and time-consuming.

RL algorithms employ various exploration strategies to strike this balance effectively. Methods like Epsilon-Greedy and Boltzmann Exploration are commonly used to encourage exploration during the initial stages of learning, while gradually shifting towards exploitation as the agent gains more knowledge about the environment. These strategies enable RL algorithms to navigate the complexities of real-world environments, adapt to changing conditions,

and deliver safe, efficient, and intelligent solutions across diverse domains.

In the context of game-playing AI, the exploration-exploitation trade-off is also a critical consideration. Game environments often present vast state-action spaces, making it challenging to discover optimal strategies. Here, MCTS emerges as a powerful approach to address this challenge.

MCTS is a popular method for making decisions in complex environments with large state spaces, commonly used in game-playing AI. It combines tree-based search and sampling-based rollouts to explore and evaluate potential actions effectively. During the search process, MCTS builds a search tree, with each node representing a state and each edge representing an action. The agent traverses this tree based on a tree policy, which balances the estimated value of a node with an exploration bonus that encourages the exploration of less visited nodes.

To further explore unexplored regions of the state-action space, MCTS employs a default policy. The default policy is a simple, heuristic-based policy that is used to simulate random rollouts from a given state. These random rollouts provide MCTS with additional insights into potential outcomes and contribute to the exploration process. The combination of the tree policy and the default policy allows MCTS to strike an effective balance between exploration and exploitation.

MCTS has demonstrated its effectiveness in various game-playing scenarios, where exploration is essential to explore diverse strategies, and exploitation is critical for exploiting the learned knowledge to make informed decisions. As the field of RL and game-playing AI continues to evolve, refining the exploration-exploitation trade-off in MCTS remains an active area of re-

search to enhance the algorithm's performance and adaptability across different game environments. Incorporating RL techniques, such as exploration through curiosity-driven learning, as explored by Pathak et al. [27], provides further insights into how to effectively balance exploration and exploitation in MCTS for various applications, ranging from autonomous driving to financial trading and robotics.

2.3 Deep Reinforcement Learning

Deep RL is a subfield of machine learning which utilizes NNs with multiple layers to automatically learn hierarchal representations of data, leading to powerful pattern reognition and planning capabilities.

2.3.1 Function Approximation

In the realm of RL, early algorithms were designed to operate in relatively simple environments, relying on tabular data structures to store and predict values for every possible action in every possible state. However, as the complexity of problems increases, such tabular approaches become impractical for high-dimensional scenarios due to their scalability limitations. To address this challenge, function approximation techniques have emerged as a powerful alternative. Function approximation involves replacing the tabular approach with a parameterized function capable of producing outputs for any valid state or action input without requiring storage for each possibility. Indeed, an important consideration is the possibility of replacing Q, V, and π tables with function approximation methods, such as DNNS. Utilizing

function approximation can offer more efficient and scalable representations of game states, empowering agents to handle larger and more complex environments while achieving enhanced performance. The function's parameters can be iteratively updated to enhance the policy, often utilizing techniques like policy gradient methods. By approximating the outputs of a vast (potentially infinite) table, function approximation methods are able to handle complex environments that are infeasible for tabular methods.In contrast to tabular methods that have independent outputs for each input, function approximators possess interconnectedness, enabling them to extrapolate to new states or state-action pairs that resemble existing ones. This property makes function approximation well-suited for navigating high-dimensional environments. While linear and kernel-based function approximators were once popular, modern RL predominantly relies on DNNs as the standard method for function approximation. [28, 5].

2.3.2 Neural Networks

Neural networks play a vital role in function approximation. They consist of layers of interconnected units that apply linear transformations to their inputs using adjustable weights. The forward pass involves passing the input through the network, where each unit's output is determined by a function with trainable parameters. The output is obtained by applying an element-wise activation function. Backpropagation, introduced by Rumelhart et al. [29], enables the training of NNs by computing partial derivatives of the error with respect to the network's weights. The weights are then updated using gradient descent. The performance of the model is evaluated using a cost

function that measures the error between the predicted output and the desired output. This error is backpropagated through the network to adjust the weights. Various techniques, such as stochastic updates or batch updates, example shuffling, input normalization, and nonlinear activation functions, have been explored to enhance the training process. These techniques contribute to improving the NN's ability to approximate complex functions and minimize the loss between predicted and desired outputs.

2.3.3 Convolutional Neural Networks

LeCun [30] observed that simple fully-connected feed-forward NNs could not maintain the topology of the input. As images have specific local structures and pixels have a high spatial correlation, LeCun et al. presented Convolutional Neural Networks to extract local features. CNNs differ from fully-connected feed-forward NNs in that the parameters that are trained are the parameters for the kernel in the convolution. A convolution is an operation defined as:

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau \tag{2.12}$$

The convolution of two functions f and g with respect to the variable t. The convolution operation combines the two functions by integrating the product of one function, f, and the time-reversed and shifted version of the other function, $g(t-\tau)$, over the entire range of τ.

Alternatively, if you're looking for the discrete form of convolution, it can be represented as:

$$(f * g)[n] = \sum_{k=0}^{T} f[k]g[n-k]$$

Here, the convolution of two discrete sequences f and g is computed by summing the element-wise product of one sequence, $f[k]$, and the time-shifted version of the other sequence, $g[n-k]$, over all possible values of k.

These equations capture the essence of the convolution operation in both continuous and discrete domains.

In the context of DeepMind algorithms, CNNs play a crucial role in processing and analyzing game-related data, particularly in visual domains such as Atari games. CNNs are employed to extract meaningful features and representations from raw pixel inputs, enabling the algorithms to understand and make decisions based on the game state.

In AlphaGo and AlphaZero, CNNs are used as policy networks to predict the probabilities of different moves or actions given a game state. These networks learn to encode the patterns and structures present in the game board, allowing the algorithms to make informed decisions based on the learned policy.

In MuZero, CNNs are utilized as dynamics models to predict the next state and next reward from the current state and action. These models help simulate future trajectories during the search process, aiding in decision-making and value estimation.

The design and architecture of the CNNs used in these algorithms are carefully crafted to capture and exploit the specific characteristics of the game domains. This includes considering factors such as input size, number of lay-

ers, kernel sizes, and activation functions. The CNNs are trained using large datasets and sophisticated optimization techniques to learn effective representations and parameters.

2.3.4 Actor Critic Methods

Actor-critic networks are a form of policy optimization algorithms. These methods involve the learning of a stochastic policy by the agent, referred to as the "actor", which is parameterized by a function such as a NN. Simultaneously, an accompanying "critic" function, often implemented as a separate NN, learns to estimate the state-value function. The critic's estimations serve to calculate advantage values used in training the actor function.

When training the actor and critic together, the critic can effectively maintain a current estimation of the state-value function, relying on the policy employed by the actor. This interaction between the actor and critic enables the improvement of the actor's decision-making abilities by leveraging the critic's value estimates. This approach is commonly used in policy gradient algorithms, including Vanilla Policy Gradient (VPG) and Trust Region Policy Optimization (TRPO) [31]. Additionally, Proximal Policy Optimization (PPO) [32], Advantage Actor-Critic (A2C), and Asynchronous Advantage Actor-Critic (A3C) [33]also perform gradient ascent to optimize performance. While PPO is often referred to as a policy gradient algorithm, it is more accurately an approximation or implementation of the fundamental policy gradient method. These methods effectively use a policy update based on the gradient of the performance objective with respect to the policy parameters. Further details and relevant references for these methods can be found in the literature.

While the actor and critic can be distinct functions, they may also share certain parameters, allowing for information transfer and enhancing the learning process. Parameter sharing facilitates the utilization of shared representations, potentially expediting convergence.

Actor-critic methods encompass the advantages of both value-based and policy-based approaches in RL. The actor learns the policy, while the critic provides valuable feedback on the desirability of different states. This combination leads to more efficient and effective learning, resulting in enhanced decision-making across a variety of domains.

AlphaZero can be loosely considered as an actor-critic method. While the original AlphaZero paper does not explicitly use the terminology of "actor-critic," it incorporates elements of both actor and critic in its approach.

In the AlphaZero architecture, the policy network can be seen as the actor, as it estimates the probabilities of different actions and guides the agent's decision-making process. The policy network selects actions based on a policy distribution, which is updated during training to improve the agent's policy.

On the other hand, the value network can be seen as the critic in AlphaZero. It estimates the value or expected outcome of a given game state, providing a measure of how favorable the state is for the agent. The value network is trained to minimize the mean-squared error between its predicted values and the actual outcomes of the games played.

Therefore, while not explicitly referred to as an "actor-critic" method in the original paper, the AlphaZero algorithm incorporates the key components and principles associated with actor-critic methods.

2.3.5 Self-Play Advantages

Self-play is a training technique used in RL, where an agent learns by playing against itself rather than relying on external supervision or expert knowledge.

Self-play is a learning approach in AI where an agent, such as a game-playing algorithm, improves its performance by playing against itself. It involves simulating a competitive environment within the AI system, where the agent acts as both players, taking turns to make moves and learn from the resulting game outcomes. The agent starts with minimal knowledge about the game and gradually refines its strategies through countless iterations of self-play. By continually challenging itself and exploring different actions, the agent learns from its successes and failures, adjusting its decision-making process based on the feedback obtained from its own simulated games. This self-reinforcing learning loop allows the AI agent to discover and adapt to effective strategies, ultimately achieving higher levels of performance without the need for external training data or human expertise. Self-play has proven to be a powerful technique, leading to remarkable successes in complex game-playing tasks and opening up new possibilities for AI research and applications.

The goal of self-play is to enable an agent to learn and improve through a process of self-exploration, discovering optimal strategies and refining its decision-making abilities. This approach addresses the challenge of training AI models in complex domains where expert knowledge may be limited or unavailable.

Both AlphaZero and MuZero leverage self-play as a key component of their training process. By playing against themselves, these algorithms can generate

a vast amount of training data, exploring a wide range of possible moves and strategies. This self-generated data serves as a diverse and representative training set, enabling the models to learn from their own experiences.

The use of self-play offers several advantages for AlphaZero and MuZero. Firstly, it provides an autonomous and adaptive learning environment. As the algorithms compete against previous versions of themselves, they face opponents of varying strengths, constantly challenging their own capabilities. This adaptive nature ensures ongoing improvement and exploration of new strategies.

Additionally, self-play allows AlphaZero and MuZero to train in a computationally efficient manner. Since they do not rely on external supervision or expert guidance, these algorithms can generate training examples rapidly and at scale. This efficiency enables them to engage in numerous iterations of self-play, accelerating the learning process and facilitating the development of robust strategies.

Moreover, self-play promotes the discovery of novel and innovative gameplay. By exploring a wide range of moves and strategies, AlphaZero and MuZero can uncover unconventional tactics that may not have been considered by human experts. This capacity for creative exploration allows these algorithms to achieve groundbreaking performance in complex game domains.

2.4 Conclusion

In summary, this chapter provided an overview of the diverse set of tools used in modern machine learning and AI for games. We explored the fundamental

concepts, algorithms, and techniques that form the basis of game-playing AI. While this chapter covered the essential foundations, each subsequent chapter will delve deeper into the relevant literature, exploring specific algorithms and their applications in various game environments.

3 Boosting AlphaZero through MCTS enhancements

3.1 Introduction

Monte Carlo Tree Search (MCTS) is a model-based method which aims to calculate optimal decisions in a chosen domain (oftentimes of large state and action space) by taking random sequences of decisions and building a search tree according to it's simulations [34]. Although it's introduction into the world of computing is relatively recent, its influence on AI methodologies has been significant, particularly in domains represented by decision trees, with notable impacts on board games and planning problems. Since the original MCTS algorithm was developed, there has been a tidal wave of time and energy dedicated to improving and perfecting the algorithm within various environments. More recent work has also coupled MCTS and other powerful look-ahead search algorithms with NNs, resulting in state-of-the-art game-playing agents and decision-making systems [35, 36, 5]. Unfortunately, these require the use of large pre-computed tables of known moves, professional

datasets of human expert games, significant and sometimes unobtainable computing resources and carefully crafted heuristic functions with a variety of edge cases.

In this chapter, we analyze and compare various proposed modifications to the MCTS algorithm, as well as their impact on performance when applied in conjunction with NNs within the AlphaZero algorithm. We implement the enhancements inividually within both MCTS and AlphaZero, thus producing MCTS, MCTS-UCT, MCTS-LA, AlphaZero, AlphaZero-UCT and AlphaZero-LA. Secondly, we then take it one step further and implement the enhancements concurrently, producing MCTS-ALL and AlphaZero-ALL. The first modification we explore is the implementation of the traditional Upper Confidence Bound for Trees (UCT) [22] algorithm within MCTS as a baseline and then within AlphaZero. UCT was designed to balance exploration and exploitation during the search process by taking into account both the average reward and uncertainty of a node in the search tree. This allows the algorithm to efficiently converge to a good solution without getting stuck in suboptimal paths. The second modification is Loss Avoidance (LA) [37, 38], which improves the performance of the algorithm by preventing it from making moves that are likely to result in a loss.

First, we will review the basic algorithms we will be working with in this thesis, MCTS and AlphaZero. Then we dive into a contextual literature review. Second, we dive into the two aforementioned enhancement strategies. Then, through a series of experiments, we will compare the performance of these modifications on Connect4 and Othello against random, greedy and minimax algorithms and identify the best-performing algorithm for each game.

Our findings provide valuable insights into the potential of incorporating modifications in MCTS-based algorithms to enhance their performance. These modifications can be applied to algorithms integrated with RL and NNs, like AlphaZero, as well as standalone MCTS algorithms. The implications of our study can guide future research in this field, paving the way for further advancements and improvements in the utilization of MCTS for solving complex problems.

3.2 Monte Carlo Tree Search

The complexity of modern strategy games poses the most significant challenge when developing AI for such domains. While expert-based techniques offer rapid progress in achieving a reasonable level of play, they fall short when confronted with complex or ambiguous situations that are difficult to encode using heuristics. Search techniques present an appealing solution to create adaptable and dynamic AI. However, traditional search methods like minimax, which explore all possible moves and evaluate resulting states, become impractical due to the vast number of moves or limitations imposed by heuristic quality in strategy games.

MCTS is a powerful search algorithm that has proven successful in a wide range of domains, including game playing, decision making, and optimization [34]. It combines heuristics and probabilistic techniques to explore and evaluate potential solutions in a systematic manner. MCTS has demonstrated remarkable performance in challenging environments with complex decision spaces, making it a valuable tool in various domains requiring intelligent

decision-making.

MCTS is an informed tree search technique as it uses gathered information from previous search iterations to guide the exploration of the tree in future iterations. Over the years, MCTS has been instrumental in achieving significant advancements in a variety of board games and computer games [39, 40], particularly due to its ability to manage complex search spaces by recursively applying Monte Carlo techniques during the search process. The robustness and adaptability of MCTS have solidified its position as a key technique despite the myriad of obstacles presented by these complex environments.

When applying tree search to address complex problems, it is possible for our algorithm to mistakenly identify a suboptimal action as the best choice. To mitigate this issue, the MCTS algorithm is employed, which continuously evaluates different actions and paths during the tree search process. By iteratively exploring the search space, MCTS progressively focuses on the most promising regions of the search tree, gradually refining its understanding of the problem domain. This adaptive exploration strategy allows MCTS to dynamically adjust its exploration factor and improve the quality of its decisions.

3.2.1 The Algorithm

Before, doing an actual move, MCTS will do numerous simulations in order to evaluate the part of the game tree and determine the best action to take. Each iteration and update of that tree, referred to as a tree-walk, involves the following four distinct phases, as seen in Figure 3.1:

Selection: The first phase will traverse the tree starting from the root node (current game state) using a *tree policy*. While a node s is found in the

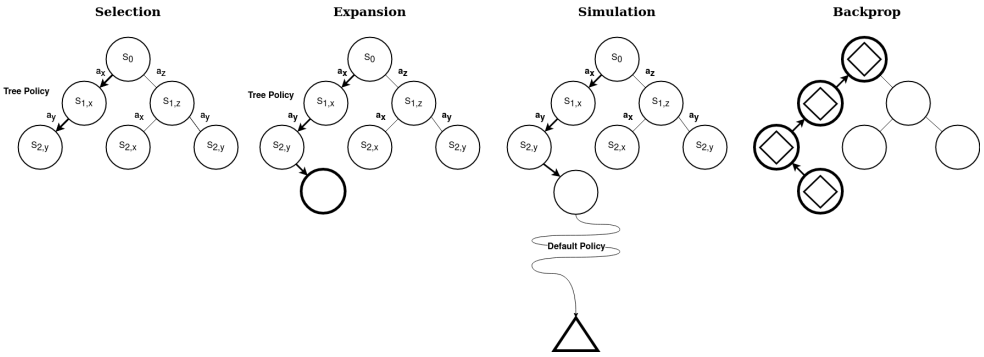

Figure 3.1: MCTS for one iteration. A single iteration refers to the process of selecting, simulating, propagating, and backpropagating a single search path in the search tree.

tree, the next edge (s, a) leading to a child node s' is chosen. The selection of the child nodes during the tree traversal is done uniformly at random. This means that each node has an equal probability of being selected. While this approach ensures that all possibilities are eventually explored, it lacks efficiency, as it doesn't prioritize nodes that seem more promising based on previous simulations.

Expansion: If the selected action a from the state s is not represented in the tree, then the algorithm is expanding a new edge (s, a) and a new child node s is added to the tree.

Simulation: For the rest of the simulated game, actions are selected using a *default policy*, typically a random play, until a terminal node is reached. The appropriate assignment of action selection probabilities plays a crucial role in determining the quality of gameplay. When all legal actions are chosen with equal probability, the resulting strategy tends to be weak and ineffective, leading to suboptimal performance of the Monte-Carlo program. Therefore,

ensuring a proper weighting of action selection probabilities is essential to achieve optimal gameplay and maximize the program's effectiveness. We can use different default policies, some which may rely on heuristic knowledge, to give larger weights to actions that look more promising and guide our agent towards a less unsuccessful level of execution.

Backprop: After reaching the end of the tree-walk, there is an associated reward computed for that terminal state. This reward value (and any acquired reward during the simulation process) is then used to update each tree node along the current path from this iteration.

This process is repeated multiple times to generate multiple simulations, which helps in estimating the values of different nodes. The outer loop, as seen in Figure 3.2 and 3.3 of MCTS continuously improves the decision-making capabilities, allowing the algorithm to adapt and learn more effective strategies as it gains more information from simulations. The game action finally executed by the program in the actual game, is the one corresponding to the child of the root node which has the best value.

Let us note that of the four aforementioned phases, there are two distinct sections which can be modified to affect performance. Firstly, we have *Tree Policy* defined as: the policy used to select or create a new leaf node from the previous leaf nodes already contained within the search tree (selection and expansion). Second, we find the *Default Policy* which is defined as: the policy used to play out through the domain from a given non-terminal node s to produce a reward value estimate. These will be the components where we propose to certain configurations and modifications in order to evaluate individual performance within a selection of different environments. We present

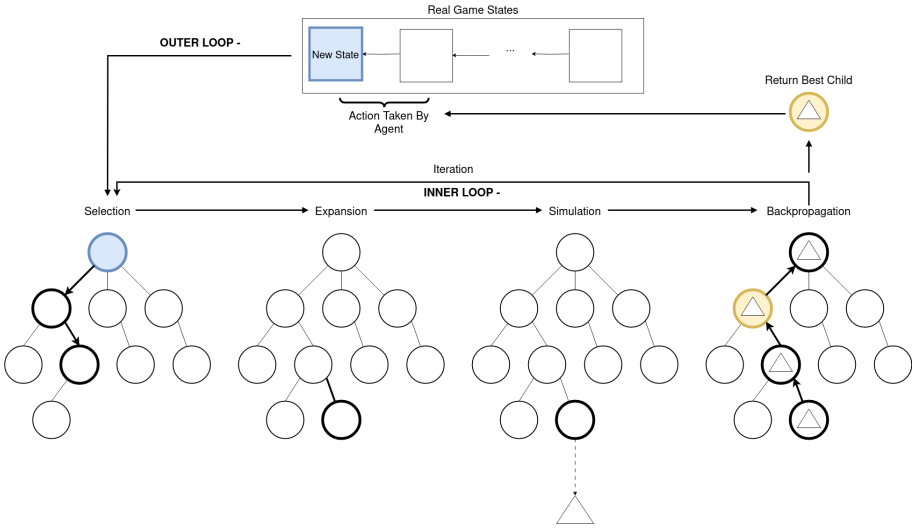

Figure 3.2: MCTS outer loop and inner loop interaction. The combination of these inner loops constitutes the overall MCTS process, which is often performed iteratively until a stopping criterion, such as a time limit or a certain number of iterations, is met.

a simple pseudocode to illustrate the process in Algorithm 1.

Algorithm 1 Vanilla MCTS approach

```
function : MCTSEARCH(s_0)
create root note v_0 with state s_0
while within computation budget do
    v_0 ← TREEPOLICY(v_0)
    Δ ← DEFAULTPOLICY(s(v_0))
    BACKUP(v_t, Δ)
end while
```

The policy update mechanism is how MCTS tackles the "exploration-exploitation" trade-off issue. It utilizes newly discovered actions to update its policy while simultaneously exploring the local space for alternative paths. Essentially, the algorithm will continue to explore and discover previously unseen nodes of the tree, updating its tree policy based on a more considerable

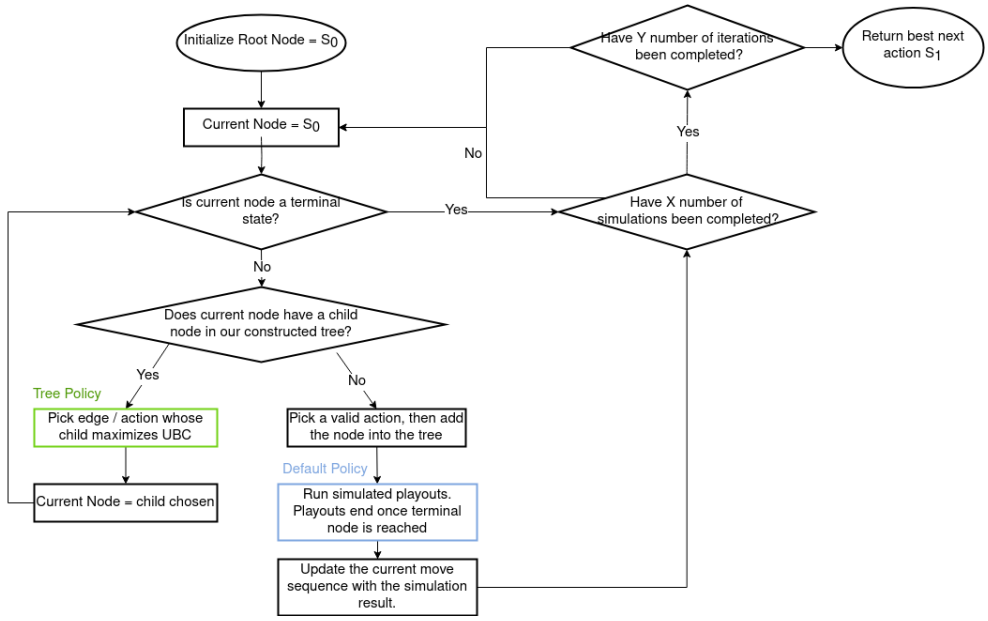

Figure 3.3: MCTS outer loop, performed to gradually build the search tree and improve the action selection based on the information gathered from previous iterations.

amount of data. As it does, it diminishes the probability of overlooking better paths toward a solution and, therefore, better policies. This can be classified as the "exploration" aspect of this approach.

Let it be noted that this version of tree search design also contains the "exploitation" side of the coin in order to make it efficient and balance out the issue mentioned above. Occurring in the second half of the algorithm, where it maintains a greedy policy as it sticks to the path with the greatest estimated value, enabling the tree to explore its depth alongside its breadth.

MCTS comes with its own share of drawbacks: within problems with large state spaces and correspondingly large action spaces, the tree will grow in depth and in breadth quickly due to a high branching factor after a few

iterations and may result in the algorithm failing to reach an optimal path, and therefore policy, within reasonable time and computational constraints. Additionally, with the vanilla approach, MCTS aims not to use any pragmatic heuristic function for positions in non-terminal states.

3.3 AlphaZero

AlphaZero is a powerful machine learning algorithm that learns complex board games like Chess, Go, and Shogi without prior knowledge. It combines DNNs, RL, and MCTS to play games. AlphaZero employs a variant of the Polynomial-UCT (PUCT) algorithm to balance exploration and exploitation during the MCTS process [6, 41].

There are three main processes for the AlphaZero algorithm: self-play, NN training and comparison between new and old network. During self-play, AlphaZero generates games where the NN, trained by the current MCTS policy, plays each move. This consists of a batch of episodes of self-play, which terminate once a terminal state s_T is reached (ie. when a winner, or none, has been established), when the game exceeds the maximum number of allocated moves, or when the search value returned by MCTS falls below a resignation threshold.

The NN used within AlphaZero is trained through a self-play RL algorithm that uses MCTS to play each move. First, the NN is initialized to random weights θ_0 . At each subsequent iteration, games of self-play are generated. Through each time-step t and each associated state s_t a MCTS search is executed using the previous iteration of the NN $f_{\theta-1}$ and outputs a vector of

search probabilities recommending moves to play π_t which it samples to play the next move.

During each iteration of MCTS, it visits a new game state and evaluates the network policy. AlphaZero employs a variant of the PUCT (Polynomial-UCT) algorithm [41] to strike a balance between exploration (i.e., visiting game states recommended by the initial policy) and exploitation (i.e., focusing on states with high expected rewards). This prior policy, learned from millions of games of self-play, provides a heuristic that guides the exploration towards more promising parts of the search space. This makes the search more efficient, as it can effectively leverage the knowledge acquired from past games to guide it.

Each tree edge stores a set of characteristics, $N(s,a), W(s,a), Q(s,a), P(s,a)$ where $N(s,a)$ is the visit count, $W(s,a)$ is the total action value, $Q(s,a)$ is the mean action value and $P(s,a)$ is the prior probability given by the network for selecting that edge. MCTS in AlphaZero follows the same general outline of the four aforementioned steps of the MCTS search mechanism with a few changes.

During the Selection phase, each simulation will traverse the tree by selecting the next existing edge with the following PUCT tree policy for action selection, shown in Equations 3.1 and 3.2:

$$PUCT(s,a) = c_p P(s,a) \frac{\sqrt{\sum_b N(s,b)}}{N(s,a)+1} \tag{3.1}$$

$$\pi_{PUCT}(s) = \arg\max_a (Q(s_t,a) + PUCT(s_t,a)) \tag{3.2}$$

which is proportional to the quotient of the prior probability and the visit count for that edge until a leaf node s' is encountered.

The Expansion phase is only comprised of the expanded leaf node being evaluated by the NN $f_\theta(s_L) = (p, v)$ and the prior probability vector is then stored in the outgoing edges from s. The leaf node is expanded and its node values are initialized to zero. Once they go through a backprop step, the visit counts and values are updated.

$$N(s_t, a_t) \leftarrow N(s_t, a_t) + 1$$

$$W(s_t, a_t) \leftarrow W(s_t, a_t) + v$$

$$Q(s_t, a_t) = \frac{W(s_t, a_t)}{N(s_t, a_t)}$$

The data from each time step (s_t, π_t, z_t) is stored according to the corresponding player at time t. Once the indicated amount of self-play games are completed, new network parameters θ_i are trained from the stored data, sampled uniformly from all the time-steps of the last batch of self-play. The NN aims to minimize the error between it's output v_t and the winner of the sampled self-play game z_t. This allows it to output similar move probabilities $p(s, a)$ to the search probabilities given by MCTS π_t. This is done by adjusting the NN parameters θ through gradient descent on the following loss, which combines the MSE and cross-entropy loss:

$$l = (z_t - v)^2 - \pi^T log\ p + c\|\theta\|^2 \tag{3.3}$$

where v is the output of the NN, z_t is the winner of the sampled game and

c is a parameter controlling the level of L2 weight regularization.

The primary motivation behind this work is that throughout the different generations of the Alpha algorithms, there have been substantial changes to different sections of the algorithm, including NN quantities and architecture, the removal of Supervised Learning (SL) techniques and even the shift from model-based to model-free learning. But oddly enough, the search algorithm has remained mostly the same throughout all the versions. Furthermore, there have been some significant advancements within the realm of MCTS which have yet to be applied and tested within noteworthy environments [34]. We aim to provide insight for the behaviour of the MCTS algorithm advancements within the RL approach of AlphaZero.

3.4 Literature Review

Numerous research endeavors have been undertaken to enhance the performance and efficiency of the AlphaZero algorithm. Some of these approaches involve modifications to the MCTS process. For instance, Anthony et al. [42] proposed an innovative approach that amalgamates imitation learning with tree search, creating a dynamic and iterative learning loop between an apprentice and an expert. Stankiewicz et al. [43] suggested specific enhancements to the MCTS algorithm, using two techniques, Last-Good-Reply (LGR) and N-grams, to enhance the play-out step of the MCTS algorithm. Additionally, the selection step was improved by initializing the visit and win counts of new nodes based on pattern knowledge. Other researchers have focused on exploring adjustments to the training process. This includes the work of

Seify et al. [44], who combined MCTS with DRL, featuring novel action value normalization.

Additionally some variations address specific challenges and improve its efficiency in various domains. One such approach is the Nested Monte Carlo Search (NMCS) [45], which employs multiple nested searches to increase the accuracy of the evaluations. By simulating shorter rollouts within a larger rollout, NMCS reduces variance and enhances the quality of evaluations. he Double Progressive Widening (DPW) algorithm [46] is another notable variation that extends the Progressive Widening technique. DPW incorporates two different widening mechanisms to efficiently explore the search space. It dynamically allocates child nodes to achieve better coverage of promising regions and, at the same time, ensures a balanced exploration of the entire game tree. The MCTS with Upper Confidence Bounds Applied to RAVE (UCT-RAVE) [47] is an extension of UCT that incorporates the Rapid Action Value Estimation (RAVE) heuristic. RAVE updates statistics using additional information from playouts, which includes the result of all moves played from a given state, even if they were not directly selected during the search.

To address the challenge of large action spaces, the MCTS with all Moves as First (MCTS-AMAF) [48] was introduced. It combines MCTS with the All Moves as First (AMAF) heuristic, which prioritizes moves that have not been visited in the current simulation. This approach enhances the exploration of the search space and allows for a more thorough exploration of possible actions. Other notable variations include the Clustered Monte Carlo Tree Search (CMCTS) [49], which groups similar game states to accelerate the search process, and the Parallel Monte Carlo Tree Search (PMCTS) [50], which

leverages parallel computing to speed up the search in complex games.

Furthermore, variations of the UCT algorithm [51] have been explored, which adaptively balance exploration and exploitation during tree search. Additionally, approaches like Loss Avoidance [42] have been proposed to mitigate overfitting during the learning process, ensuring more robust and generalized performance.

Moreover, the advent of MuZero [24], a successor to AlphaZero, further pushes the boundaries of self-learning algorithms by eliminating the need for a pre-existing model or knowledge of the game dynamics. This innovative approach enables MuZero to learn directly from raw input and achieve state-of-the-art performance across multiple domains. These varied modifications highlight the ongoing potential for improvements to the AlphaZero algorithm and its successors, such as MuZero. These algoriths, as part of the Alpha family, will be a main highlight within Chapter 4.

Soemers et al. [37] outline a variety of methodological enhancements made to the MCTS algorithm to adapt its implementation to improve it's performance within difficult game-playing environments. They introduce the N-gram Selection Technique, which is a playout policy update that biases the steps in favor of action sequences that have been successful in past simulations. One particularly notable concept is LA, in which the agent seeks to minimize losses by immediately pursuing superior options when faced with a loss.

These variations illustrate the versatility of the MCTS algorithm and its ability to adapt to different scenarios and challenges. The ongoing research and development of MCTS and its variations continue to expand the frontiers

of game-playing AI and decision-making algorithms, driving advancements in the field of AI and beyond.

3.5 Proposed Enhancements

The decision to test the existing algorithms, UCT and LA in AlphaZero stems from the motivation to explore and understand how these modifications can further enhance the performance and capabilities of the AlphaZero algorithm. UCT, being a widely adopted and successful variation of the MCTS algorithm, offers a compelling opportunity to investigate how adaptive exploration strategies can influence the search process, potentially leading to more efficient and effective decision-making. By incorporating UCT into AlphaZero, we aim to assess whether this variation can improve the algorithm's ability to balance exploration and exploitation, thereby leading to better policy generation and gameplay.

Similarly, the incorporation of LA in AlphaZero arises from the need to address potential overfitting during the learning process. By integrating the LA technique, we aim to mitigate the risks of excessively tailoring the algorithm's strategy to specific training data, ensuring more robust and generalized performance across diverse game scenarios. This approach is of particular interest as it has the potential to enhance the stability and reliability of AlphaZero's learning process, allowing it to adapt more effectively to previously unseen game situations.

In summary, the choice to test UCT and LA in AlphaZero is motivated by the desire to explore cutting-edge modifications to the MCTS algorithm

and address specific challenges in RL. By integrating these variations into AlphaZero, we seek to unlock further advancements in strategic decision-making systems, ultimately contributing to the broader field of AI and game-playing algorithms.

3.5.1 UCT Policy

The Upper Confidence Bound for Trees (UCT) algorithm [22], is a value-driven RL technique, with the goal of maximizing the reward from a set of options with uncertain outcomes. It primarily concentrates on the initial state of a game and the succeeding state tree, disregarding other game aspects. This is achieved by utilizing the principles of the multi-armed bandit problem-solving algorithm, specifically the Upper Confidence Bound (UCB) strategy, introduced by Auer et al. [52] to guide it's search, favouring actions with both high average reward and high uncertainty.

$$UCB = \hat{X}_s + 2\sqrt{\frac{lnN(s)}{N(s,a)}} \tag{3.4}$$

UCB, as shown in Equation 3.4, has the propitious property that it guarantees to converge to the minimax tree, meaning it is optimal. Once the suggested adjustments to this policy are implemented in order to apply it to rollout-based Monte-Carlo planning.

During the selection phase, the tree policy chooses the next node according to the statistics stored. The action-value function $Q_{UCT}(s,a)$, in Equation 3.5, is estimated by a search tree in a tabular representation $\mathcal{T} \subseteq \mathcal{S}\text{x}\mathcal{A}$, containing all the (state, action) visited pairs.

$$\hat{Q}_{UCT} = \left[Q_{UCT}(s,a) + c_{uct} \sqrt{\frac{logN(s)}{N(s,a)}} \right] \tag{3.5}$$

where $Q_{UCT}(s,a)$ is the average reward gathered over the previous tree-walks that have traversed that node s and edge (s,a) and therefore child node s'. $N(s,a)$ is the number of times where the action a has been selected from state s and $N(s)$ counts the total number of visits to a state s. The action selection is dictated by the Equation 3.6.

$$\pi_{UCT}(s) = \arg\max_{a} \hat{Q}_{UCT} \tag{3.6}$$

This formula encourages the exploitation of higher reward choices while the right-hand term encourages the exploration of less visited choices. Of particular significance is the second term, or the exploration term. As having children nodes which have not been explored will yield a dominantly large UCT value, therefore they are assigned the largest possible value. This allows each child node to be explored at least once. Allowing this approach to be a promising candidate in overcoming the exploration-exploitation conundrum.

We also explore a small variation of UCT [51] which incorporates a prior policy π_θ, shown in Equation 3.7, unlike the original UCT algorithm. This allows us to incorporate it within the AlphaZero algorithm. We briefly walk through the steps in Algorithm 2.

$$\hat{Q}_{UCT} = \left[Q_{UCT}(s,a) + c_{uct} \sqrt{\pi_\theta(a) \frac{logN(s)}{N(s,a)}} \right] \tag{3.7}$$

Algorithm 2 MCTS with UCT

```
function : MCTS_UCT(s_0)
create root note v_0 with state s_0
while within computation budget do
    v_0 ← UCT_TREEPOLICY(v_0)
    Δ ← DEFAULTPOLICY(s(v_0))
    BACKUP(v_t, Δ)
end while
return action of the best child of v_0

function : UCT_TREEPOLICY(v)
while v is non-terminal do
    if v is not fully expanded then
        return EXPAND(v)
    else
        v ← BESTCHILD(v)
    end if
end while
return v

function : BESTCHILD(v)
return π_UCT(s)

function : EXPAND(v)
select untried action a from A(s(v))
add child v' to v with s(v') = f(s(v), a)
return v'
```

3.5.2 Loss Avoidance

MCTS is often used for games with small state spaces and therefore finite win states. However, finding these states can be difficult due to the vastness of the game tree, which can pose a challenge even for powerful algorithms like MCTS. There are many games which have a high number of losing game states that are quickly discovered, this allows for the opportunity of avoiding them in future case.

Loss Avoidance (LA) [37] is a technique used in MCTS to bypass losses encountered during the search process. Instead of backpropagating losing results as in traditional MCTS, LA generates a state for every sibling of the last node in a simulation, and only the node with the highest evaluation is backpropagated. All generated nodes are still added to the tree and store their own evaluation in memory.

LA allows MCTS to maintain an optimistic view of the value of nodes by avoiding losses and continuing to explore promising nodes that may have otherwise been discarded. This approach differs from traditional MCTS, which backpropagates all outcomes equally. LA can be viewed as a modification to the criteria used to choose the next node to expand in the tree and the backpropagation method, favoring nodes that have not been explored or have been explored less. The modified formula for selecting child nodes during the selection step can be represented as in Equation 3.8:

$$\hat{Q}_{LA}(s,a) = \frac{Q_{LA}(s,a) * N(s,a) + (\sum_{i=1}^{T-t} \gamma^i L(s,a)_{t+i}) * \hat{N}(s,a)}{N(s,a) + \hat{N}(s,a)} \tag{3.8}$$

where $L(s,a)$ is the chosen reward for child node a from state s. Typically tailored per environment but generalized by setting it to an egregiously large negative value. We provide the LA algorithm we used in Algorithm 3.

Algorithm 3 MCTS with Loss Avoidance

```
function MCTS_LA(s_0)
    Create root node v_0 with state s_0
    while within computation budget do
        v_0 ← LA_TREEPOLICY(v_0)
        Δ ← DEFAULTPOLICY(s(v_0))
        BACKUP(v_t, Δ)
    end while
    return action of the best child of v_0
end function

function LA_TREEPOLICY(v)
    while v is non-terminal do
        if v is not fully expanded then
            return EXPAND(v)
        else
            v ← BESTCHILD_LA(v)
        end if
    end while
    return v
end function

function BESTCHILD_LA(v)
    return arg max_a Q̂_LA
end function

function EXPAND(v)
    Select untried action a from A(s(v))
    Add child v' to v with s(v') = f(s(v), a)
    return v'
end function
```

3.6 Methods

3.6.1 Environments

The choice to use Connect4 and Othello as the primary environments for our experiments was deliberate and strategic. These games possess specific characteristics that make them well-suited for evaluating the enhancements discussed in this paper. First, both Connect4 and Othello are played on squared-boards, providing a structured and well-defined state space that allows for systematic exploration and comparison of different algorithms and variations. Second, being zero-sum games, the objectives of Connect4 and Othello align with the core principles of RL, where agents aim to maximize their own rewards while minimizing the opponent's gains.

Furthermore, Connect4 and Othello are full-information games, meaning that all relevant information about the game state is readily available to the players. This aspect is essential for evaluating the effectiveness of the enhancements in decision-making accuracy and strategic planning. By having complete knowledge of the game board, the algorithms can make informed and strategic moves, allowing us to measure their performance more accurately.

Additionally, Connect4 and Othello are well-known and widely studied games, which have served as benchmarks for evaluating various AI algorithms. This prior research [53] provides a rich set of baseline results and reference points for comparison, enhancing the validity and significance of our experimental evaluations.

By utilizing these games, we could capitalize on the similarities between their game structures and network architechtures and those of AlphaZero

and MuZero, thus reducing the potential confounding factors introduced by varying input representations, network architectures, and other environment-related variables. This strategic choice allowed us to focus our investigation on the fundamental capabilities of the algorithms and better isolate their respective performances within the context of these specific games.

Connect4

The first game is Connect4. Connect4 is a popular children's game played on a 7x6 grid. It involves two players, red and black, competing on a vertical grid with six rows and seven columns. The game starts with red making the first move by dropping a disk, also known as a "stone," into one of the seven columns. The disk then falls into the lowest unoccupied cell in that column. The players take turns dropping disks into columns until one player manages to connect four disks either horizontally, vertically, or diagonally. This particular game has been extensively studied and proven to be a suitable task for learning algorithms, with traditional model-free approaches demonstrating exceptional performance in solving it. According to Numberphile, there are 4,531,985,219,092 ways to fill a Connect4 grid.

Connect Four is a well-studied game with a known outcome for perfect play [53]. It is a first-player win, where the first player will typically achieve a victory before the 41st move by making their first move in the centre column. However, starting in the columns adjacent to the center results in a theoretical draw. These well-established properties of Connect Four provide valuable insights into its strategic landscape and serve as a benchmark for evaluating algorithms and enhancements in this domain.

Agents have complete information about the Connect Four game and can observe each other's moves. The agent is provided with the following variables:

- Initial State: Specifies how the game is set up. Connect Four typically starts with an empty board where players take turns dropping their pieces into columns.
- Players: Represented as [1, -1]. The player pieces on the board and their positions aid in evaluating the state from the maximizing player's perspective.
- Action: Legal moves in the state space correspond to placing a piece in an empty column of the board.
- Result: Transition model that specifies the result of moves in the state space, including the updated board after each player's move.
- Terminal State: The game ends when a player successfully connects four of their pieces either horizontally, vertically, or diagonally, or when the board is completely filled with pieces.

Othello

The second game is Othello, also known as Reversi, a strategy board game played on an 8x8 board. The game is played by two players, each with their own set of discs of different colors, typically black and white. The game begins with four discs placed in the center of the board in a square pattern, with two discs of each color facing each other. The objective of the game is to have more discs of your color on the board than your opponent by the end of the game. Players take turns placing their discs on the board, with each placement potentially flipping some of the opponent's discs to their own color. A player

must place their disc adjacent to an opponent's disc so that it "sandwiches" one or more of the opponent's discs between two of their own. The sandwiched discs then flip to the player's color. If a player cannot make a legal move, their turn is skipped. The game reaches a terminal state when no other legal move is available or when both agents pass their turn, and the winner is the player with the most discs of their color on the board.

In the game, agents possess complete information about the game state and can observe each other's moves. The agent is given the following variables:

- Initial State: Describes the game setup, where we introduce diverse random board states to assess the search performance in start, middle, and end game scenarios.
- Players: Represented as [-1,1], indicating player pieces on the board and their positions, which help evaluate the state from the maximizing player's viewpoint.
- Action: Denotes the legal moves available in the state space.
- Result: The Transition Model determining the outcome of moves in the state space, including the available moves for the other player.
- Terminal State: The game concludes either when all empty spaces are filled or when a player has no more legal moves remaining.

3.6.2 Test Opponents

Test Opponents for Connect4

Greedy Agent: The Greedy agent in Connect4 selects moves that yield the highest immediate reward based on a straightforward heuristic evaluation of

the game board. It aims to maximize the number of pieces in a row to achieve victory.

Let N_{agent} and $N_{opponent}$ be the number of pieces owned by the agent and the opponent, respectively, after making a move. The heuristic value for Connect4 is given by Equation 3.9:

$$V_H = N_{agent} - N_{opponent} \tag{3.9}$$

Minimax Agent: The Minimax agent in Connect4 employs a depth-limited search tree with alpha-beta pruning to explore potential moves. The maximum search depth was set to 5 for a reasonable balance between computational resources and performance. The evaluation function considered factors such as the number of potential winning sequences, blocked sequences, and piece placements on the board. The evaluation function for Connect4 is found in Equation 3.10

$$V_E = w_1 \cdot (N_{agent} - N_{opponent}) + w_2 \cdot N_{agent_w} - w_3 N_{opponent_w} \tag{3.10}$$

Where w_1, w_2, and w_3 are weights assigned to each factor. N_{agent} is the number of pieces controlled by the agent. N_{opponent} is the number of pieces controlled by the opponent.N_{agent_w} and $N_{opponent_w}$ are the number of agent and opponent wins respectively.

Test Opponents for Othello

Greedy Agent: The Greedy agent in Othello selects moves that lead to the immediate highest reward, based on a simple heuristic evaluation of the game board. It prioritizes maximizing the number of pieces owned by the agent.

A simple heuristic called Coin Party [54] is simply the difference in coins between maximising and minimising player. Often used by Greedy agents in Othello is based on the difference in the number of pieces owned by the agent and the opponent after making a particular move. The heuristic value is the number of pieces gained (or lost) by the agent. Let N_{agent} and $N_{opponent}$ be the number of pieces controlled by the agent and the opponent, respectively, after making a move, then the equation is congruent to 3.9.

Minimax Agent: The Minimax agent in Othello utilizes a depth-limited search tree with alpha-beta pruning to explore possible moves. The maximum search depth was set to 10 for efficiency, and an evaluation function considered factors such as the difference in the number of pieces controlled by each player and the mobility of each player's pieces. The evaluation function for the Minimax agent in Othello considers several factors to assess the strength of a board position. One common evaluation function is based on the difference in the number of pieces controlled by the agent and the opponent. It can be defined as follows:

One common evaluation function for Othello used in the Minimax algorithm is based on the difference in the number of pieces controlled by the agent and the opponent, along with additional factors to consider the board's positional strength. The evaluation function is defined in Equation 3.11:

$$V_E = w_1 \cdot (N_{\text{agent}} - N_{\text{opponent}}) + w_2 \cdot M + w_3 \cdot CC + w_4 \cdot \text{Stability} \quad (3.11)$$

Where w_1, w_2, w_3, and w_4 are weights assigned to each factor with values 2,2,4,4 respectively. N_{agent} is the number of pieces controlled by the agent. N_{opponent} is the number of pieces controlled by the opponent. M represents the number of legal moves available to the agent. CC measures the agent's control over the corner positions, which are highly valuable in Othello. Stability evaluates the stability of the agent's pieces, ensuring that they are less prone to being flipped by the opponent.

This evaluation function provides a comprehensive measure of the board position's strength, combining various aspects of the game to guide the Minimax algorithm in making optimal decisions. By incorporating these specific details about the test opponents for the Connect4 and Othello environments, we aim to provide a clearer understanding of the evaluation process, emphasizing the differences between MCTS and Minimax agents in these specific games.

3.6.3 AlphaZero Network

Our implementation of AlphaZero for Connect4 and Othello consists of a DNN with an architecture broadly divided into three parts: an input layer, a body comprising of multiple residual blocks, and two separate heads for policy and value outputs.

Input Layer: Similarly to AlphaZero [6], the NN accepts as input an 6x7x3

image stack for Connect4 and an 8x8x9 image stack for Othello representing the board state. Each slide in this stack represents a specific aspect of the game state, such as the current board position, the player's pieces, the opponent's pieces. Othello includes all this information for three historic states, as there cannot be repeat moves in the game. Each cell in these planes holds binary values indicating the presence or absence of a piece. Please refer to Table 3.6.3 for comparison table.

Body: The body of the network, consists of 10 residual blocks. Each block contains a pair of ReLU activated, batch-normalized convolutional layers, following the ResNet architecture principles [55]. Each convolution operation engages 256 filters, each possessing a kernel size of 3×3 and a stride of 1.

Policy Head: The policy head is responsible for predicting a probability distribution over all possible moves from the current game state. It includes an additional ReLU activated, batch-normalized convolutional layer, succeeded by a final convolution. For Othello, the network predicts the likelihood of placing a disc on each of the 64 board positions plus one additional output for the "pass" action, amounting to 65 potential outputs. For Connect4 the policy head predicts a probability distribution over all possible moves. For Connect4, this equates to predicting the likelihood of dropping a disc into one of the seven columns. We then choose a move based off the softmax over the probability distribution of the columns.

Value Head: The value head predicts the expected outcome of the game from the current position, outputting a single scalar value within the range of -1 (expected loss) to 1 (expected win). It begins with a ReLU activated, batch-normalized convolution involving a solitary filter of kernel size 1×1

with stride 1. This is followed by a ReLU linear layer of size 256 and finally, a tanh activated linear layer of size 1.

The MCTS and AlphaZero experiments were meticulously implemented using the Python programming language and orchestrated on the robust Google Cloud platform. The self-play data, pivotal for enhancing the neural networks, was collected through uniform sampling from 500k games. This Python-based implementation, coupled with the computational resources offered by the Google Cloud, ensured the efficient and comprehensive evaluation of MCTS and AlphaZero variations across diverse game environments.

3.6.4 Experimental Setting

In this section, we aim to address a couple of questions: (1) How is the state-of-the-art hybrid algorithm AlphaZero affected once we introduce MCTS modifications and do these changes bring forward a more robust algorithm? (2) What changes among the MCTS algorithm subsections positively affect the performance the most?

To better understand which components of the the different modifications contribute the most to the performance gains, we implement both, MCTS, and Alpha-Zero, in four different versions: Standard, UCT, LA, and ALL. In this experimental setting, we aim to evaluate the performance of the MCTS-based algorithms, MCTS, MCTS-UCT and MCTS-LA, and MCTS-ALL, as well as the two AlphaZero algorithms, AlphaZero, AlphaZero-UCT and AlphaZero-LA, and AlphaZero-ALL against three baseline algorithms: Greedy and Random and Minimax. We also conduct a test over the same structure where we implement both of the MCTS adjustments simultaneously in the same algo-

Table 3.1: AlphaZero Network Architecture for Connect Four and Othello. Each residual block contains a pair of ReLU activated, batch-normalized convolutional layers, following the ResNet

Game	Layer	Input	Output	Details
Connect4	Input	6×7×3	6×7×3	Image stack representing board state
	Body	6×7×3	8×8×256	10 residual blocks, 256 filters, 3 × 3 kernel
	Policy Head	8×8×256	1×7	Conv layer, predicts move probabilities
	Value Head	8×8×256	1	1 convolutional layer, 256 filters, 1 × 1 kernel ReLU linear layer, tanh activated linear layer
Othello	Input	8×8×9	8×8×9	Image stack representing board state
	Body	8×8×9	8×8×256	10 residual blocks, 256 filters, 3 × 3 kernel
	Policy Head	8×8×256	1×65	Conv layer, predicts move probabilities
	Value Head	8×8×256	1	1 convolutional layer, 256 filters, 1 × 1 kernel ReLU linear layer, tanh activated linear layer

rithm; this produces two other game-playing agents which we label as MCTS-ALL and AlphaZero-ALL. We will evaluate the performance of these algorithms under two different search budgets, low (N_{sim}=10) and high (N_{sim} = 100), corresponding to the varying number of allowed simulations (tree-walks) before selecting the best action. Different experiments may be conducted to optimize the tree depth but one of our goals was to maintain a reasonable computation time. Lastly, if the same move was played more then three times or if the game length exceeded 100 for Othello and 40 for Connect4.

It's worth noting that our computational requirements were significantly lower compared to the AlphaZero paper [6], as well as different environments, where they test on chess, Shogi and Go. The original paper had 5,000 episode per iteration and 500 simulations per turn. An episode refers to a complete play-through of a game, starting from the initial state until the game terminates, either by a win, loss, or draw. An iteration is a unit of training in the AlphaZero algorithm, where the NN is updated based on the data collected from self-play episodes. A simulation, also known as a search, is a single traversal of the MCTS algorithm, which explores the game tree and estimates the value of different actions from a given game state.

3.7 Results

As shown on Table 3.2, our findings highlight the remarkable superiority of the AlphaZero-ALL algorithm, surpassing its counterparts in both the Othello and Connect4 games. This trend is evident across multiple games, even outperforming the standard AlphaZero algorithm. These results suggest that our

Table 3.2: Average win rates over 50k games of the different algorithms within the respective environments. AlphaZero results averaged over 5 networks. Random player start.

	Connect4			Othello		
v.	Greedy	Random	Minimax	Greedy	Random	Minimax
MCTS	52.3%	63.1%	43.3%	41.3%	57.9%	32.2%
MCTS-UCT	53.5%	70.4%	51.5%	43.6%	61.3%	54.4%
MCTS-LA	61.4%	66.5%	63.6%	55.8%	**70.7**%	64.2%
MCTS-ALL	**63.6**%	**72.5**%	**65.6**%	**61.5**%	67.1%	**65.4**%
AlphaZero	77.5%	79.5%	77.6%	73.5%	81.2%	75.3%
AlphaZero-UCT	68.7%	71.3%	76.6%	79.1%	81.1%	82.4%
AlphaZero-LA	72.5%	77.4%	70.9%	82.1%	83.8%	82.7%
AlphaZero-ALL	**81.6**%	**82.4**%	**78.8**%	**82.3**%	**84.3**%	**83.2**%

computationally conservative modifications to the MCTS algorithm have the potential to elevate performance across the same games when tested against various opponents. The average win rates were obtained over 50,000 games for each algorithm within the specific game environments, with AlphaZero results being averaged over 5 networks and the random player start configuration.

Within the MCTS family of algorithms implemented, we see that there is a pretty consistent and linear improvement from the MCTS to MCTS-ALL, and where MCTS-LA outperforms MCTS-UCT within both chosen environments. All except for MCTS-ALL within the Othello environment against the random opponent, where it performs at a 67.1% win rate while MCTS-LA score in Othello reaches 70.7%. It is also worth noting that againt Minimax and Random opponents, MCTS-ALL and MCTS-LA perform relatively similarly, we can tentatively state that the additional implementation of UCT within MCTS-ALL does not significantly contribute to its performance.

As we examine the progression of win rates across our algorithmic mod-

ifications, a consistent pattern emerges: an increase in wins for algorithms with additional implementations, as opposed to their original counterparts. However, this pattern does not extend uniformly to the AlphaZero family of algorithms. The AlphaZero-ALL algorithm shines with its highest win rates of 82.4% and 84.3% in the Othello and Connect4 environments, respectively, against a random opponent. Notably, it also maintains strong performance against other opponents, including greedy and minimax. Within the context of the selected parameters and network size, the AlphaZero-ALL modification consistently showcases superior performance compared to other variants, underscoring its adaptability and efficacy even under resource constraints. Intriguingly, in the Connect4 environment, the original AlphaZero implementation secures the second-best performance against every opponent. However, the performance ratios exhibit variations when we shift our focus to the medium environment of Othello.

To assess the impact of algorithm choice on the obtained means, a one-factor ANOVA was conducted for each problem. An ANOVA test is used to determine whether there are significant differences between the means of multiple groups. In this case we conduct an ANOVA test to compare the win rates of different algorithms (MCTS, MCTS-LA, MCTS-UCT, MCTS-ALL, AlphaZero, AlphaZero-UCT, AlphaZero-LA, and AlphaZero-ALL) against three opponents (Greedy, Random, and Minimax) based on 5 repetitions of 1k games for each combination. We begin by defining our null hypothesis and alternative hypothesis. Where our null hypothesis H_0 : There is no significant difference in the mean win rates among the different algorithm-opponent combinations. And out alternative hypothesis H_1: At least one of the algorithm-opponent

combinations has a different mean win rate. And lastly, our significance rate is $\alpha = 0.05$

For the Connect4 environment, the ANOVA detected a statistically significant difference in reward among at least two algorithms. We found a p-value of 2.31E-04 indicates that we can reject the null hypothesis and conclude that the four means are not all equal. However, we don't know which pairs of groups are significantly different. We also conducted a post-hoc Tukey's HSD test. Tukey's Honest Significant Difference (HSD) test is important because it helps identify specific pairwise differences between multiple groups, allowing for a deeper understanding of the significance of observed variations in statistical analyses such as ANOVA. We found that found that the difference in win rates between MCTS-ALL and AlphaZero-ALL was significant (p = 0.0345 , 95% C.I. = [50.86, 96.34]), as was the difference between AlphaZero and AlphaZero-ALL. No significant difference was found between MCTS-LA and MCTS-ALL (p = 0.108).

We repeat the same ANOVA test but within the Othello environment, though with the same comparative analysis of the algorithms. We obtain a p = 1.56E-03 value which also indicates that we are able to reject the null hypothesis and conclude that the means are not equal. The Othello Tukey test found that the difference in win rates between MCTS and MCTS-ALL was significant (p = 9.32E-03 , 95% C.I. = [35.52, 84.23]), as is the difference between MCTS-ALL and AlphaZero (p = 0.0239, 95% C.I. = [41.66, 85.45]). No significant difference was found between MCTS and MCTS-LA (p=0.299).

While these results offer valuable insights into algorithmic performance, it's essential to acknowledge that certain confounding variables might influence

our observations. One such variable could be the sensitivity of algorithmic performance to specific game environments, opponent strategies, or even the computational resources available. To mitigate these confounding effects, we ensure consistency by using identical computational constraints and neural network architectures across our experiments. This controlled environment helps us isolate the impact of algorithmic modifications and better comprehend their implications. It should be noted that there are a variety of ways of controlling variables for performance. While the approach chosen for this thesis is to ensure that the network architecture and MCTS algorithm used are as similar as possible throughout the experiments and different implemented algorithms, others may choose to simply optimize each algorithm within each domain and then set them against their opponents.

All the Alpha algorithms showcased in this section shared their networks. There were 5 different networks were trained and tested, each training with 5k episodes for Connect4 and 5k episodes for Othello. We played a total of 50k games for both environments. The plots in Images 3.4 to 3.7 depict the rewards averaged over the 5 random seeds.

We present Figure 3.8 and 3.9 which showcase different heatmaps depicting the learned starting positions of AlphaZero-ALL and MCTS-ALL, respectively for both players in Connect Four, averaged over an extensive dataset of 10k episodes per player starting position.

This visualization provides insights into the strategic landscape of the game, highlighting the preferred initial moves for each player and offering a comprehensive overview of the opening strategies. By analyzing this heatmap, we can discern the patterns and tendencies of optimal starting positions, en-

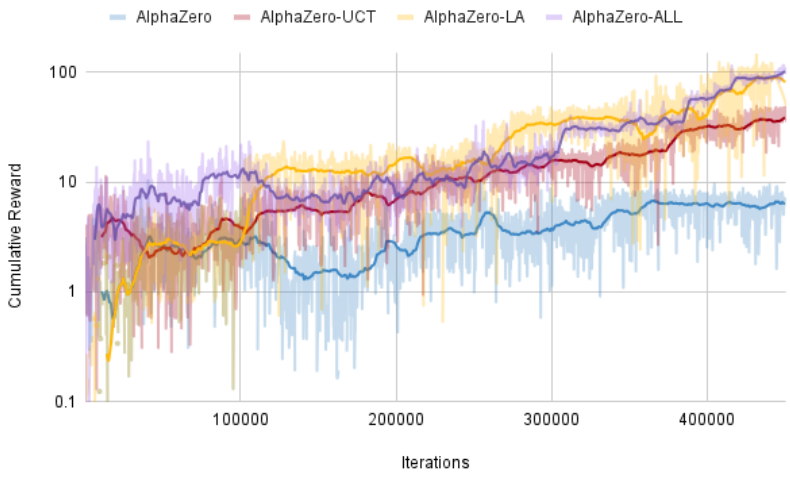

Figure 3.4: $N_{sim} = 10$ per iteration on Connect4. Averaged across 5 seeds.

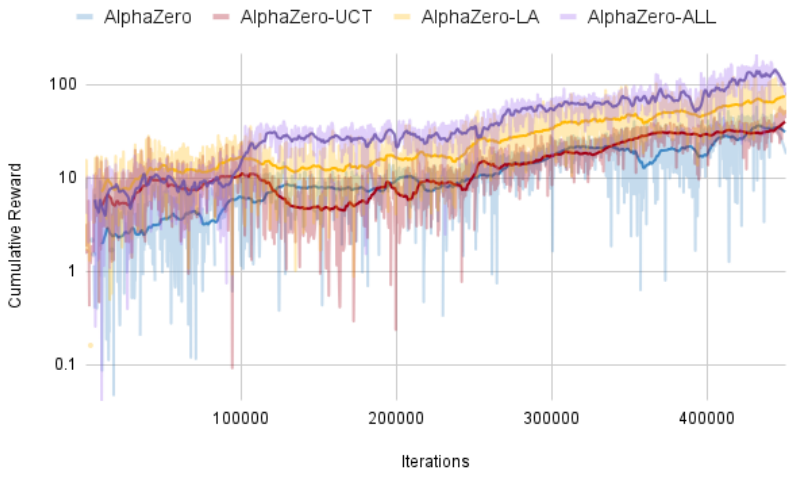

Figure 3.5: $N_{sim} = 100$ per simulation on Connect4. Averaged across 5 seeds.

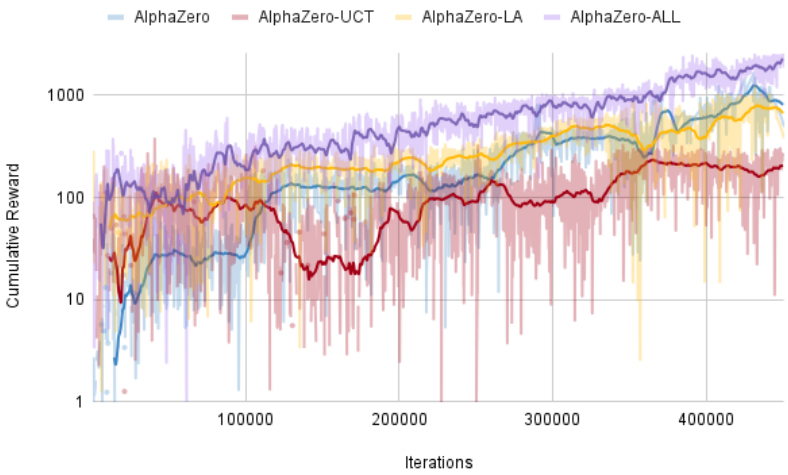

Figure 3.6: $N_{sim} = 10$ per iteration on Othello. Averaged across 5 seeds.

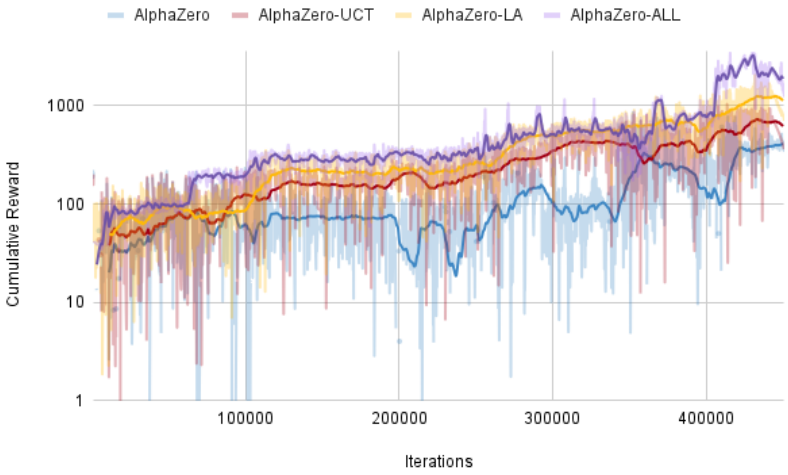

Figure 3.7: $N_{sim} = 100$ per iteration on Othello. Averaged across 5 seeds.

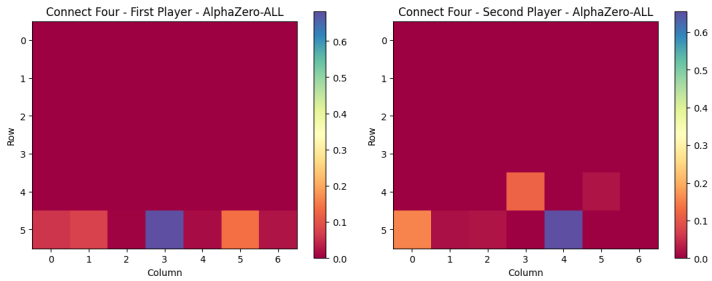

Figure 3.8: AlphaZero-ALL map depicting the average starting positions for first and second player averaged over 10k games in Connect Four.

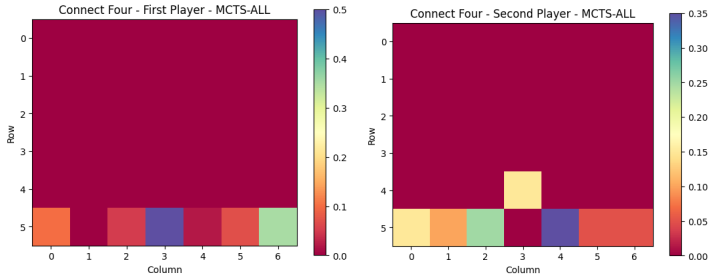

Figure 3.9: MCTS-ALL map depicting the average starting positions for first and second player averaged over 10k games in Connect Four.

abling a deeper understanding of the game's dynamics and strategic decision-making.

3.8 Conclusion

In conclusion, this study aimed to investigate the impact of various search modifications on the state-of-the-art hybrid algorithm AlphaZero and identify which changes within the MCTS algorithm can positively affect its performance the most. Utilizing Connect4 and Othello as test environments, the AlphaZero-ALL algorithm emerged as a favourable and strong performer,

showcasing the potential of specific modifications to enhance MCTS-based algorithms.

Our findings reveal that different algorithms may be more suitable for certain games, and that tailoring the modifications to an algorithm may influence its performance. The success of the AlphaZero-LA algorithm may be attributed to its dual implementation of UCT and LA, and it's ability to avert losses and maintain an optimistic view of node values, allowing for continued exploration of promising paths. Although there were improvements in the results between the algorithms with different enhancements, more studies should be done to conclude on the significance in performance difference between each variation.

Despite the promising results, it is essential to acknowledge the limitations of our study. The experiments were conducted on two specific games, Connect4 and Othello, which are squared-board, zero-sum, full-information games. These games share similarities with Go, the domain for which AlphaZero was initially designed. Therefore, our conclusions may not directly apply to other types of games, such as card games or chess, where different strategies and game dynamics may come into play.

Furthermore, the simulation budget used in our experiments was constrained to a specific value, limiting the depth and breadth of the search space explored by the algorithms. Increasing the simulation budget could potentially lead to different outcomes, providing further insights into the algorithms' behaviors and performances.

Despite these limitations, the results of this study shed light on the effectiveness of various modifications to the MCTS algorithm and their impact

on the performance of AlphaZero in Connect4 and Othello. By understanding the strengths and weaknesses of different algorithms, we can better tailor their configurations for specific game domains, paving the way for more efficient and effective AI agents in a wide range of applications. Future research could expand this investigation to other game domains and explore the impact of varying simulation budgets on algorithm performance, providing a more comprehensive understanding of the algorithmic enhancements.

4 Algorithm Comparison

4.1 Introduction

In this chapter, we delve into the implementation and evaluation of a comprehensive framework encompassing the Alpha algorithms, better knows as the three Deepmind cutting-edge algorithms: AlphaGo, AlphaZero, and MuZero. These Alpha algorithms, along with a baseline of Deep Q-Network (DQN), serve as the foundation for our study, allowing us to explore their capabilities and compare their performance on four diverse environments: Connect4, Othello, Pong, and Pinball. A notable aspect of our research is that all NNs were trained under the same computational resource and time constraints, ensuring a fair and unbiased evaluation.

While it is expected that MuZero, a state-of-the-art algorithm and youngest of the Alpha siblings, outperforms its counterparts, our aim is to determine whether its superiority is solely, or not, attributed to its intensive training regime or it's substantial allocation of resources (compared to it's other Alpha predecessors). Through our evaluation, we elucidate the nuances and improvements of MuZero compared to AlphaGo, AlphaZero, and DQN. By

analyzing their performance on different gaming environments, we uncover valuable insights into the strengths and weaknesses of each algorithm and provide a comprehensive assessment of their efficacy.

By conducting this implementation and evaluation, we contribute to the understanding of these remarkable algorithms and their potential applications and behaviour within a diverse set of problem domains. Our findings shed light on the implementation of the Alpha algorithms, as well as the advancements made by MuZero, while also highlighting the significance of well-balanced comparisons to ascertain its true value beyond its computational requirements. Through this chapter, we provide a foundation for further exploration of these algorithms and pave the way for future advancements in the field of RL.

In this chapter, we will first cover the essential related work of other researchers which span from the re-implementation to diverse testing within different environments. Their array of work has also focused on transferability, self-play, scaling and adversarial models. Next, we dive into the definition and a brief re-explanation of tree search within ML, as MCTS was the prime focus of the first section, yet still a critical part within the Alpha algorithms, which are now our focus within Chapter 4. Then, we dive into a structured and thorough presentation of AlphaGo, AlphaZero and MuZero. Lastly, conclude the thesis with an array of experiments over four critical environments; Connect4, Othello, Pong and Pinball. Our main contributions include, what we call the "re-implementation" of the Alpha algorithms. We build upon the previous work of Davaut and Hainaut who implemented Muzero [56]. Which encompasses the implementation of each of the algorithms using network structures and algorithms which are as similar as possible, aiming towards a fair

comparison when comparing the algorithms to their siblings. We run these experiments with the goal of assessing the performance and capabilities of the Alpha algorithms in diverse problem domains. Through comprehensive evaluations, we aim to understand the strengths and weaknesses of each algorithm and shed light on their adaptability and generalization abilities across different environments. Furthermore, we analyze the impact of the factor of training resources, with the aim of potentially observing how variations in the amount of computational power and data availability can influence the algorithms' learning rates and performance.

4.2 Literature Review

After the publication of AlphaZero, there was a huge surge of researchers scrambling to replicate, improve and test this algorithm. ELF OpenGo [57], an open-source reimplementation of AlphaZero, showcased convincing superhuman performance against top professionals in Go, shedding light on unresolved mysteries and facilitating future research. Similarly, A0C (Alpha Zero for Continuous action space) [58] presented a method to learn continuous policy networks, extending the success of iterated search and learning to single-player, continuous action space domains. Moreover, a critical examination of AlphaZero's limitations [59] highlighted the need for explanation techniques for human-friendly understanding of acquired game-playing knowledge and improvements in machine learning methods for more data-efficient learning in real-world domains. These subsequent algorithms serve as important stepping stones to further propel research in the field of AI and highlight the potential

of DRL in various domains.

Several papers have contributed to the advancement of AlphaZero, exploring its capabilities and enhancing its performance. A paper [60] focused on RL techniques applied to AlphaGo Zero, combining deep learning with MCTS to improve the algorithm's performance. They experimented with different RL strategies and shed light on the training process. To further enhance AlphaZero's performance, Huang, Li, and Wang (2019)[61] proposed Residual Monte Carlo Tree Search (RMCTS). By addressing the issue of inadequate exploration during the MCTS process, they introduced a residual network architecture that improved the algorithm's exploration capability. Experimental results demonstrated that RMCTS outperformed the original AlphaZero.

Schrittwieser et al. [62] presents the original algorithm with an extensive exploration of the MuZero algorithm. It showcases MuZero's exceptional performance across a diverse set of domains, including Atari 2600 games, Go, chess, and shogi. The authors highlight the versatility of MuZero by demonstrating its ability to achieve state-of-the-art results in complex environments without relying on any domain-specific knowledge or human data. The paper also delves into the inner workings of MuZero's learned model and planning algorithm, shedding light on its impressive adaptive learning and decision-making capabilities.

The underlying core of the Alpha algorithms - is their ability to learn and improve through self-play and the reliance on DNNs to guide decision-making—makes self-play a crucial aspect in the advancement of RL. Self-play enables an agent to continuously refine its strategies by competing against different versions of itself, thereby generating diverse and informative training

data. By engaging in self-play, the agent can explore various decision-making scenarios, learn from its successes and failures, and iteratively update its policies to achieve higher levels of performance. Moreover, the integration of DNN allows the agent to generalize its knowledge and make informed decisions based on learned patterns and representations. Through this review, we explore the significance of self-play in RL, examining how it fosters the development of intelligent agents capable of adaptive and strategic behavior in diverse environments. Furthermore, this paper [63] focused on improving self-play techniques in the AlphaZero algorithm. They proposed Adaptive Symmetric Self-Play (ASSP), a method that dynamically adjusts the exploration rate during self-play. This approach led to more balanced and effective training, resulting in improved performance across various game scenarios. This paper [64] introduces near-optimal self-play algorithms in two-player zero-sum games. In a tabular episodic Markov game with S states, A max-player actions, and B min-player actions, achieving a complexity $O(SAB)$ significant improvements in sample complexity. The proposed optimistic Nash Q-learning algorithm requires steps, while the new Nash algorithm has a sample complexity of $O(S(A + B))$, both closing the gap with previous lower bounds. Moreover, the paper provides a computational hardness result for the process of learning best responses against a fixed opponent in Markov games.

Furthermore, Browne et al.[65] explore a general framework that combines DRL and search for self-play in any two-player zero-sum game. It guarantees convergence to a Nash equilibrium and has demonstrated success in both perfect and imperfect-information games. In heads-up no-limit Texas hold'em poker, ReBeL achieves superhuman performance while requiring less domain

knowledge compared to previous poker AI systems. Additionally, in Wu et al. [66], they investigate the combination of self-play with population-based training, which involves training multiple agents simultaneously. Through extensive experiments, they demonstrate the benefits of self-play and population-based training in achieving improved performance and faster convergence in RL tasks. Sun et al. [67] delve into the adversarial self-play algorithm. They propose a framework where agents compete against each other through self-play to enhance their learning capabilities. By incorporating adversarial training techniques, the authors demonstrate the effectiveness of the approach in enhancing the agent's decision-making abilities and achieving better performance in RL tasks. Lastly, another paper written by Tian et al. [68] presents an approach to improving RL through self-play and the use of an opponent's model. They explore how self-play can be combined with the knowledge of an opponent's strategy to enhance the learning process. The authors highlight the importance of self-play in exploring different strategies and refining the agent's policy, ultimately leading to improved performance in RL tasks. A specific area of research focuses on the development of methods tailored for game AI competitions [69]. These competitions are held annually, providing a platform for AI players to compete against each other. The IEEE Conference on Games (CoG), previously known as Computational Intelligence and Games (CIG), is a prominent event in the academic community, featuring popular games such as Angry Birds and StarCraft, as well as broader disciplines like General Video Game Playing, MicroRTS, and Strategy Card Game. Open competitions encourage the exploration of novel methods, leading to remarkable results achieved through the skillful integration of multiple techniques.

Effective problem-specific enhancements address various aspects of the MCTS algorithm, contributing to improved performance in game AI competitions.

One instance of such a masterful combination is the algorithm designed to triumph in the Pac-man competition [70] named MAASTRICHT. The key enhancement focused on predicting the opponent's moves, lessening the number of states assessed. In the backpropagation phase, the node reward scheme incorporated the final score and simulation time. Interestingly, the Pac-Man winner also employed the MCTS algorithm. A series of refinements were outlined in that same paper. The majority pertain to heuristic modifications in the policy:

1. a tree with edges of varying lengths
2. preserving a history of favorable decisions to enhance the default policy,
3. policy alterations (distinct ways to eliminate some evidently incorrect decisions)

Another well-known challenging environment we have seen used for performance testing is the Mario arcade game [71] detail a series of tweaks and enhancements to the standard MCTS algorithm which yield some outstanding results. Unfortunately, the majority of these improvements can be categorized as the integration of domain knowledge into the general algorithm: MixMax (high rewards for effective actions), Macro Actions (dodge monsters in a series of moves), Partial Expansion (remove evident choices), and Hole Detection (additional heuristic to leap over a fatal trap).

In the field of model-based RL (MBRL), the question arises whether the objective of models should be to accurately simulate environment dynamics.

This study [72] challenges the notion of pursuing perfect accuracy in modeling and instead proposes focusing on the usefulness of models to the learner. Through experiments in a non-stationary environment, the researchers demonstrate that their meta-learning algorithm, which prioritizes model usefulness over accuracy, enables faster learning compared to using an accurate model built with domain-specific knowledge. These findings shed light on the potential benefits of redefining the goals of MBRL models, emphasizing their practical usefulness rather than striving for unattainable accuracy. Additionally, another study [73] investigates the role of episodic memory in model-based (MB) versus model-free (MF) learning. By examining the influence of episodic information on choices, the study aims to discern whether it affects decision-making through MB planning or MF evaluation. The findings reveal that subjects exhibited both incremental MF and MB strategies, alongside strong MB planning using individually cued episodes.

DREAMERv2 [74] is a recent paper that combines model-based and model-free learning to improve the efficiency of RL. DREAMERv2 uses a model-based approach to learn a predictive model of the environment, which is used to generate simulated experience. This simulated experience is then used to train a model-free policy. The combination of model-based and model-free learning allows DREAMERv2 to learn more efficiently than using either approach alone.

While there has been significant research and interest in replicating and improving these algorithms after the publication of AlphaZero, my work stands out in several ways. Firstly, I present a unique developmental framework that encompasses all three Alpha algorithms, facilitating a holistic view of their

capabilities and performance. The implementation of these algorithms simultaneously, with shared network architecture and MCTS algorithm, has not been widely attempted in previous research. Additionally, I conduct a comprehensive and systematic comparison of these algorithms across four diverse environments: Connect4, Othello, Pong, and Pinball. This allows for a fair and unbiased evaluation, ensuring that the algorithms' strengths and weaknesses are thoroughly examined in different problem domains. Moreover, my research aims to understand the impact of training resources on the performance of MuZero, investigating whether its superiority is solely attributed to its intensive training regime or if it can generalize effectively with fewer resources. This investigation is essential for gaining insights into the algorithm's adaptability and potential real-world applications. By exploring this novel approach, I contribute to the understanding of the interplay between these state-of-the-art algorithms, paving the way for potential synergistic benefits and new insights.

In summary, my thesis differentiates itself from previous research by presenting a reimplementation framework to facilitate testing, by conducting a comprehensive comparison of Alpha algorithms and analyzing the impact of training resources on MuZero. These contributions advance our understanding of RL algorithms, their capabilities, and potential applications, paving the way for further research and development in this exciting field.

4.3 Tree Search in Machine Learning

The field of ML has witnessed a significant surge in computer science research, resulting in numerous groundbreaking innovations. Its widespread popularity and versatile applications in addressing real-world problems have been rapidly expanding. ML encompasses a diverse range of techniques, with NNs being one of its prominent components.[75], decision trees [76], random forests[77], boosting algorithms[78], Bayesian approaches[79], support vector machines [80]. While traditionally associated with classical AI and search problems, MCTS can be viewed as a ML approach, specifically a form of RL. In recent times, MCTS has been increasingly combined with other ML models, resulting in remarkable achievements and successes in various domains. This integration of MCTS with ML highlights its adaptability and versatility in addressing complex challenges beyond its classical applications.

MCTS is a model-based search algorithm used to find optimal decisions in complex decision-making domains, particularly in games and planning problems. The main goal of MCTS is to efficiently explore the vast decision space and identify the most promising moves to make. Unlike traditional tree search algorithms, MCTS uses two essential components: the tree policy and the default policy. During the selection phase, the tree policy guides the search by determining which child nodes to explore based on their estimated potential for achieving high rewards. It aims to strike a balance between exploring new possibilities and exploiting known promising paths. On the other hand, the default policy is utilized during the simulation phase to play out the game from a given node to generate a terminal state. This simulation, also known

as a "rollout," helps estimate the value of the node by considering various game outcomes and approximate the potential outcomes of actions and build a search tree incrementally. It starts with a single node representing the current state of the game and iteratively expands the tree by selecting and simulating actions until a certain computational budget is reached.

As explained in Chapter 3, during each iteration, MCTS employs a four-step process: Selection, Expansion, Simulation, and Backpropagation. By repeating these iterations, MCTS efficiently allocates computational resources to promising moves, focusing on those with higher win rates and exploration potential. This property allows MCTS to handle games with large and complex decision spaces, making it a powerful and widely used algorithm in modern game-playing AI, most notably for us, in the Alpha algorithms. Its ability to combine exploration and exploitation effectively makes MCTS an essential tool in building intelligent agents capable of achieving superhuman performance in various domains.

4.4 General Outline

The family of game-playing algorithms comprising AlphaGo, AlphaZero, and MuZero share fundamental components that underpin their success in mastering complex games. These algorithms harness the power of DNNs, RL, and search techniques to achieve remarkable performance. At their core, all three algorithms utilize MCTS as a key component. MCTS facilitates exploration of the game tree, enabling the agents to simulate future game trajectories, evaluate potential moves, and refine their policies. This combination

of self-play training and MCTS provides a solid foundation for learning and decision-making in strategic games.

The MCTS procedure depicted in Figure 3.1 can be iteratively employed to play entire episodes or outer loops. This process entails conducting a search from the current state s_t of the environment, selecting an action a_{t+1} based on the statistics π_t obtained during the search, executing the chosen action to transition to the next state s_{t+1}, and recording the corresponding reward u_{t+1}. This cycle continues for multiple iterations, enabling the agent to learn and refine its decision-making abilities over time. This process is repeated until the environment reaches a terminal state s_T. By iteratively performing these steps, MCTS enables the agent to progressively improve its decision making capabilities within real states by exploring and evaluating different action sequences within the search tree (ie. latent states). The tree policy can be greedy or exploratory. If the latter then the sample action is proportional to the visit count and with a temperature parameter to control the degree of exploration, as shown in Equation 4.1.

$$\pi(a|s) = (\frac{N(s,a)}{\sum_b N(s,b)})^{1/t} \tag{4.1}$$

The architecture of Alpha algorithms revolves around two ML models, which represent the value and policy functions, respectively. In the context of NNs, these models are commonly referred to as "heads," as illustrated in Figure 4.1. During the training process, the NN is supplied with training examples in the format (s_t, π_t, z_t), where π_t corresponds to the MCTS estimate of the policy from state s_t, and z_t denotes the final outcome of the game from

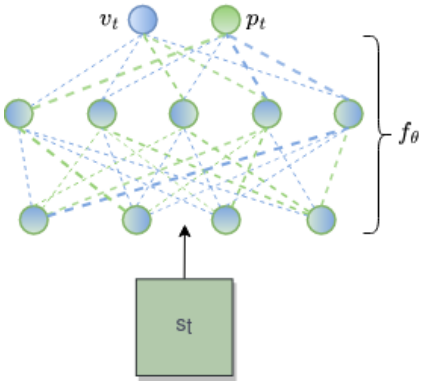

Figure 4.1: Network Training

the perspective of the player at that state. This combined approach, involving the value and policy functions, plays a crucial role in the success of Alpha algorithms.

The self-play algorithm can be seen as an approximate policy iteration scheme, where MCTS is utilized for both policy improvement and policy evaluation. Initially, a NN policy is used as a starting point. MCTS is then employed based on the NN's policy recommendations, resulting in a much stronger search policy. The refined search policy derived from MCTS is integrated back into the NN's function space to bolster the policy even further. Likewise, the results obtained from self-play games are also incorporated into the NN's function space to enhance its capabilities. This iterative process of refinement and evaluation through self-play enables the algorithm to converge towards more effective and resilient policies. The projection steps entail training the NN parameters to align with the search probabilities and self-play game outcomes, ensuring a continuous and adaptive enhancement in the agent's decision-making capabilities. By adjusting the NN's parameters based

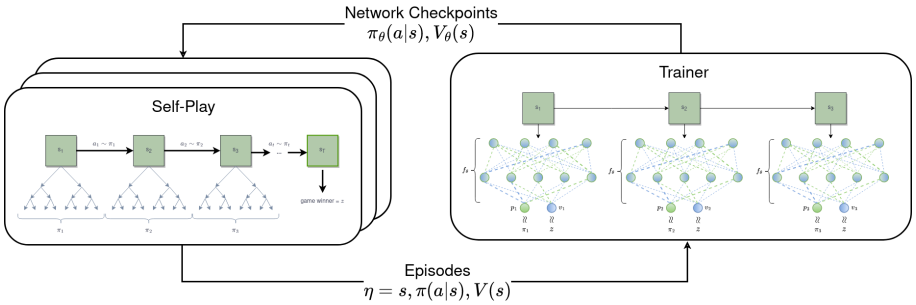

Figure 4.2: **General Main Training Loop:** Iterated tree search and function approximation.

on the search probabilities, the policy can be improved to reflect the stronger search policy. Additionally, aligning the NN with the self-play game outcomes enhances its ability to evaluate different states and make informed decisions during the RL process. Through this training procedure, the NN gradually adapts to capture the knowledge and strategies learned from the MCTS and self-play, leading to more effective and accurate policies. The general process is shown in Figure 4.2

In order to conduct the previously describe self-play training loop, the agents must train on generated data, as shown in Figure 4.3. By playing thousands of games against themselves, these algorithms generate a vast amount of training data, allowing them to refine their strategies and improve their decision-making abilities over time. In each position s_t, a MCTS π is executed (see Figure 4.4) using the latest NN f_θ. Actions are chosen based on the MCTS-generated search probabilities, with a_t drawn from π_t. The final state s_T is evaluated using the game rules to determine the game's winner z_t.

In summary, the primary training loop involves a learner that receives and stores the latest observations in a replay buffer. The learner utilizes these

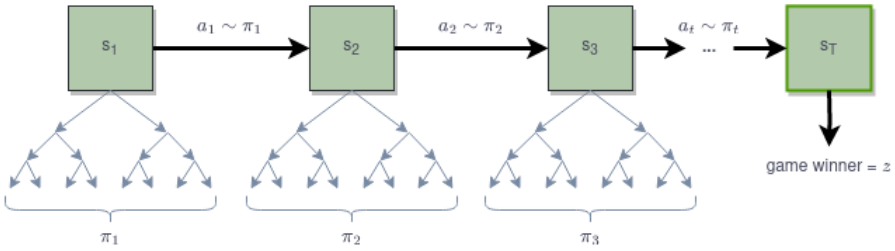

Figure 4.3: **Episode Generation:** An agent plays a real game $s_1, \ldots, s_T$ against itself.

trajectories to execute the training algorithm explained earlier. Additionally, multiple actors are employed, which periodically access the most recent network checkpoint from the learner. These actors utilize the network in MCTS to select actions and engage with the environment, generating new trajectories during the process.

AlphaGo used a combination of SL and RL with separate NNs for policy and value estimation. AlphaZero, in contrast, employed a single NN architecture for both policy and value estimation, using unsupervised learning through self-play. MuZero, being the most advanced, integrated a learned model for state transitions with value and policy networks to predict outcomes and suggest actions during MCTS. It could play games without prior knowledge of game rules, making it more versatile and adaptable to various domains. These differences in NN architectures and learning methods allowed each algorithm to achieve remarkable success in mastering complex games.

4.5 AlphaGo

The introduction of MCTS marked a significant leap in AI's ability to play what is considered to be a significantly complex board game, such as Go. MCTS emerged as a major breakthrough in computer Go, and the combination of MCTS with ML techniques constituted another significant advancement [81]. In 2015, AlphaGo, developed by Silver et al. [5] with the support of Google, achieved a groundbreaking milestone by defeating a professional human player, Fan Hui, on a standard board without a handicap. Subsequently, in 2016, AlphaGo demonstrated its exceptional capabilities by winning a momentous 4-1 victory against Lee Sedol [8], one of the most esteemed Go players in history. This accomplishment is widely regarded as a recent and significant AI achievement and has inspired a multitude of research efforts that combine MCTS with ML models.

AlphaGo combines several key elements in its algorithm. It utilizes a combination of DNNs and MCTS. The NNs are trained using a large dataset of expert human moves to learn patterns and strategies in the game. The MCTS algorithm explores potential moves by simulating numerous game sequences to evaluate their potential outcomes.

AlphaGo's training involved a two-step process: SL and RL. Initially, the NN was trained using expert human moves, learning to predict the best moves given a board position. This SL phase helped the system develop a strong foundation for gameplay.In the process of training the NN for AlphaGo, several key components are integrated. First, we train a fast rollout policy p_π and a SL policy network p_σ using a dataset of human expert moves, allowing them

to predict these moves accurately. The SL policy network then serves as the initial state for the RL policy network p_ρ, which is subsequently improved through policy gradient learning, aiming to outperform previous versions of the policy network. Their interaction is showcased in Figure4.5. The RL policy network is further used to generate a new dataset by engaging in self-play games. To complement the policy networks, a value network v_θ is trained using regression to predict the expected outcomes of the games, enabling the estimation of the agent's chances of winning or losing.

In the AlphaGo tree search process, the algorithm follows a series of steps to determine the best action to take. Initially, the edge with the highest action-value Q is selected, and an upper confidence bound U is applied, incorporating the stored prior probability P and the visit count N for that edge. The visit count N increases as the algorithm traverses through the search tree.

Once a leaf node is reached, it is expanded, and the associated position s is evaluated using the NN. The NN stores the vector of P values in the outgoing edges from s. After evaluation, the action-values Q are updated by taking the mean of all evaluations V in the subtree below that specific action.

This intricate combination of different components significantly contributes to the overall effectiveness and success of the AlphaGo algorithm, enabling it to make highly strategic and intelligent moves during gameplay. Finally, upon completion of the search, search probabilities π are returned, as seen in Figure 4.4

After that, RL was employed to further improve AlphaGo's performance. The system played numerous games against itself, utilizing the outcomes of these games to refine its NN and policy network. Reinforcement learning

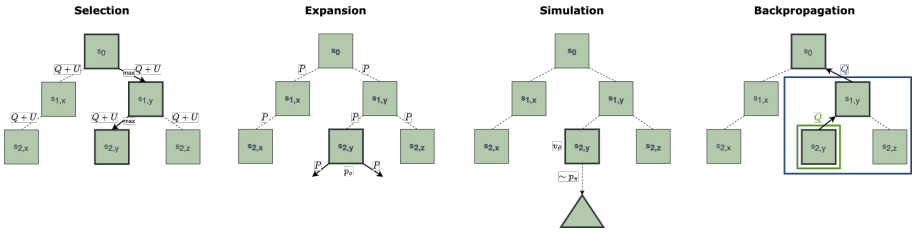

Figure 4.4: The four phases of MCTS within AlphaGo

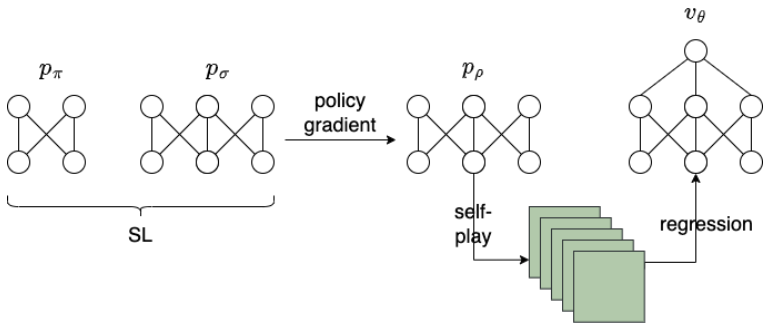

Figure 4.5: Relationship outline of distinct neural networks within AlphaGo

enabled the system to explore new strategies and optimize its gameplay. AlphaGo employs a sophisticated architecture consisting of three NNs during gameplay. The first NN serves as a policy network p_σ for real play, albeit with a slower computational speed than the second NN. However, it exhibits high accuracy (

sim57%) in predicting human moves. Its primary function is to generate a list of plausible moves, each assigned with corresponding probabilities. By leveraging these plausible moves, the MCTS is initialized. The slower speed of the first NN can be attributed to its larger size and the computation-intensive nature of the Go board's properties it considers, such as liberty counts, ataris, and ladder status.

In contrast, the second NN p_ρ with parameters ρ, is designed to be smaller and faster, sacrificing some accuracy (
sim24%) in predicting moves. This policy network operates without utilizing computed properties as inputs. Once the MCTS component reaches a leaf node in the current tree, p_ρ takes over. It simulates the remainder of the game by playing out positions using vaguely plausible moves and assigns scores to the resulting end positions.

The third NN acts as a value network v_θ, solely focusing on estimating the expected win margin for a given board position without engaging in actual gameplay simulations. Its purpose is to provide a value calculation for the provided game state without executing any moves. To arrive at an approximate result for a particular MCTS node, the results obtained from Monte Carlo playouts using p_ρ and the value calculations derived from the third NN are averaged. This averaged value is then recorded as the approximation for the respective MCTS state. Using the priors from p_σ and the accumulating results of MCTS, the values will continue to guide the actions the agent will take, creating a new path for further Monte Carlo exploration.

The impact of AlphaGo extends beyond the game of Go. Its advancements in DL and RL techniques have influenced various fields, including natural language processing, robotics, and scientific research [82, 83, 14]. Overall, AlphaGo demonstrated the potential of combining DNNs and MCTS in achieving high-level performance in complex games. It showcased the power of machine learning and pushed the boundaries of AI research.

4.5.1 AlphaGo Methods

In order to be able to evaluate this particular architecture on another environment other than Go, we require a dataset on which to train SL aspect. In this thesis we utilize separate human games dataset for AlphaGo, found in the following: Othello database, Connect4 database, Atari database, Atari database2. There are some limitations to consider when using these datasets, as they are either human expert games, complete set of possible moves or DQN generated and stored values. These values may not be optimal nor might they be plentiful enough to train a NN extensively. This is outside the scope of this thesis and we lean heavily on the concept of self-play and assume that the usage among the three Alpha algorithms is sufficient to replicate results as the original datasets are unavailable. To build the game dataset we use to train the network, we randomly sample 1k games from the different databases.

The selected environments are always two-player zero-sum games, ensuring that the original function with deterministic transitions and zero rewards (except at terminal states) can be effectively utilized. In this context, a policy represents a probability distribution over the legal actions, while the value function denotes the expected outcome when all actions for both players are chosen based on the policy. Recall that each zero-sum game has a unique optimal value which determines the outcome from a state obtained by choosing optimal policy play from both players, demonstrated in Equation 4.3.

$$v^p(s) = \mathbb{E}[z_t|s, a_{t,\dots,T} \sim p] \tag{4.2}$$

$$v^*(s) = \begin{cases} z_T & \text{if } s = s_T \\ \max_a -v^*(f(s,a)) & \text{else} \end{cases} \tag{4.3}$$

Where $v^p(s)$ is the expected value and $v^*(s)$ optimal value can be estimated via MCTS. It will estimate the optimal value of the interior nodes through double approximation, as in Equation 4.4 :

$$V_n(s) \approx v^p(s) \approx v^*(s) \tag{4.4}$$

In the initial approximation, n Monte Carlo simulations are employed to estimate the value function of a tree policy P_{tree}. Subsequently, the tree policy p is utilized, which selects actions based on a search control function $\text{argmax}_a(Q_n(s,a) + U(s,a))$, such as PUCT (Polynomial Upper Confidence Bound for Tree). This function aims to choose children with higher action values, where $Q_n(s,a) = V_n(f(s,a)) + U(s,a)$ and $U(s,a)$ plays a crucial role in encouraging exploration during the search process.

The AlphaGo program efficiently integrates large NNs into its search algorithm by implementing an asynchronous policy and value MCTS approach. The search tree consists of nodes (s) with edges (s,a) for all legal actions $a \in A(s)$. Each edge stores statistics and follow the selection algorithm covered in Chapter 3. Each edge stores the prior probability $(P(s,a))$, and combined

mean action value ($Q(s, a)$). The MCTS algorithm proceeds through four stages: Selection, Evaluation, Backup, and Expansion. The state-action pair (s') is enqueued for evaluation by the policy network. Using the SL policy network $p_\sigma(\cdot|s')$ with parameters σ, the prior probabilities are computed and then substituted for the existing placeholder prior probabilities $P(s', a) \leftarrow p_\sigma(a|s')$.

We trained the p_σ to classify positions learned through targets in the environments respective dataset. The data set was split into a test set and a training set (1:3). Each position consisted of a raw board description or raw Atari frames consisting of 210 $\times$ 160 pixel images with a 128 color palette. Fortunately, the selected datasets for Atari already underwent preprocessing steps to reduce input dimensionality. The initial RGB frames are converted to gray-scale and down-sampled to a 110×84 image. For the final input representation, an 84×84 region of the image, which captures the playing area s and the selected move a made by the human, is cropped. In each training step, a randomly selected mini-batch of 64 samples from the dataset, were sampled, and a stochastic gradient descent update was applied to maximize the log-likelihood of the action, as depicted in Equation 4.5.

$$\Delta\sigma = \frac{\alpha}{m} \sum_{k=1}^{m} \frac{\partial \log p_\sigma(a_k|s_k)}{\partial\sigma} \tag{4.5}$$

Gradients older than 100 steps were discarded. Similarly, the other policy network weights are initialized and every 100 iterations.

$$\Delta\rho \quad = \quad \frac{\alpha}{n} \sum_{i=1}^{n} \sum_{t=1}^{T^i} \frac{\partial \log p_\rho(a_t^i|s_t^i)}{\partial\sigma} (z_t^i - v(s_t^i)) \tag{4.6}$$

The training procedure for the value network follows the same approach as the SL policy network, with the only difference being the parameter update based on Mean Squared Error (MSE) between the predicted values and the actual rewards, as illustrated in Equation 4.7.

$$\Delta\theta = \frac{\alpha}{m}\sum_{k=1}^{m}(z^k - v_\theta(s^k))\frac{\partial \log p_\sigma(a_k|s_k)}{\partial\sigma} \tag{4.7}$$

The original AlphaGo has two seperate RL networks, but in order to maintain the objective of comparing the Alphas in the most equilateral testing environment, we implement the AlphaZero architecture where there is one network but two separate heads representing p and v. In the original implementation of AlphaGo, the state representation uses a few handcrafted feature planes. For the sake of this thesis we forgo these as the main goal is to create a framework which is easily adaptable to different environments.

4.6 AlphaZero

AlphaZero [62] is the third generation algorithm developed by DeepMind within this family. The main core and concepts remain the same. Recall the following details about how AlphaZero works. Essentially there are three stages which make up the trainign pipeline, all of which are executed in parallel:

Self-play games are continuously generated via MCTS and the data is collected to train the singular DNN. Following each training round, the newly trained model undergoes a comparison with the previous model. If the new model achieves victory over the previous one, the training process proceeds.

However, in case the new model does not outperform the previous one, the training outcome is discarded, and the process reverts to the previous step. In contrast, AlphaGo's training process involved multiple stages and a combination of SL from human expert games, RL, and self-play. Initially, AlphaGo was trained on a dataset of human expert moves to learn the basic patterns and strategies used by skilled players. Then, it employed RL by playing against various versions of itself to improve its gameplay through self-play. The training was iterative, with each new version of AlphaGo competing against the previous one, and only the superior models were retained for further training. The process continued for thousands of iterations until AlphaGo reached a high level of proficiency, eventually defeating world champion Go players.

While both AlphaZero and AlphaGo utilize self-play and NNs for training, AlphaZero's training approach is more streamlined and does not require any human expert guidance. It relies solely on self-play simulations and MCTS to explore game positions and generate data for training. This simplicity allows AlphaZero to achieve remarkable performance in various games without the need for extensive human expert knowledge, making it a more efficient and adaptable algorithm for mastering multiple board games.

4.6.1 AlphaZero Methods

The network architecture is as follows, in Figure 4.6: The input features undergo processing through a residual tower, which consists of a single convolutional block followed by either 19 or 39 residual blocks. The convolutional block applies a convolution operation with 256 filters of size 3x3 and stride 1, accompanied by batch normalization and a rectifier ReLU activation. The

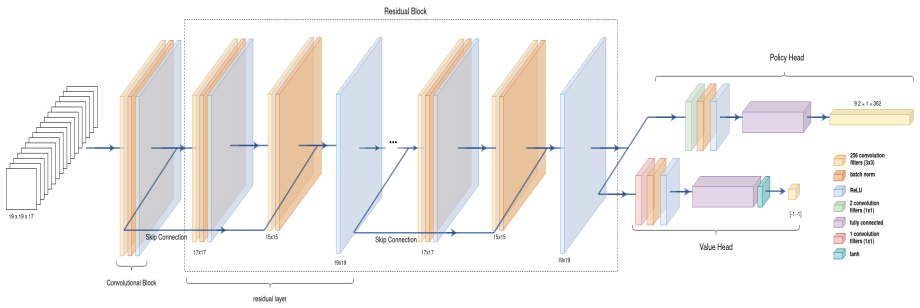

Figure 4.6: **Network Architecture:** illustrates the network architecture, consisting of a convolutional block, a residual tower with 19 or 39 blocks, policy and value heads, and specific modules for each component. All our networks only consist of 10 blocks.

residual blocks sequentially apply convolutions, batch normalization, ReLu, skip connections, and more convolutions. The policy head will output a vector of variable size depending on the chosen environment. This vector represents logit probabilities for all intersections and the pass move/no move if the game allows. The value head outputs a scalar value in the range of [-1, 1].

It's important to note that for this thesis, a modified version of the network was used, with a total of 10 residual blocks across all environments and algorithms to ensure a standarized implementation for the variety of algorithms. Additionally, the input dimensions and thus the layer dimensions all change depending on the environment chosen for the simulation.

The loss function used during the optimization process is defined by the Equation 3.3. This loss function guides the training process by quantifying the discrepancy between the predicted outputs of the NN and the desired outputs. The optimization algorithm aims to minimize this loss, resulting in improved performance and accuracy of the NN. The cross-entropy and MSE losses are

Table 4.1: Selected statistics of AlphaZero training

	Othello	Connect4	Pong	Pinball
Mini-batches	64	64	500	500
Training Games	1k	1k	50 k	50 k
MCTS Sims	100	100	100	100
α	{0.3, 0.15, 0.03}	{0.3, 0.15, 0.03}	{0.5}	{0.5}

weighted equally, assuming unit-scaled rewards between -1 and +1.

In AlphaZero, a single NN is continuously updated, eliminating the need to wait for iterations to complete. Self-play games are generated using the most recent parameters of this NN at all times. Checkpoints are created every 1k training steps, serving as snapshots of the NN's state and aiding in the generation of subsequent self-play game batches. These settings provide a balanced framework considering stability, regularization, and progress tracking.

4.7 MuZero

When faced with limited access to rules or a perfect simulator, planning for search requires essential elements: policy, value, and reward predictions. To achieve this, MuZero defines a representation function, denoted as h_θ, which projects the observation history to a latent state, s, used in the model. This hidden state enables us to make predictions about the value and policy using the prediction function. Additionally, there is the dynamics function g_θ, allowing us to simulate the model forward. It estimates the next state and the associated reward, aiding our search without a simulator. At each step of the search tree, we utilize the prediction function f_θ to roll it forward, guiding our search with the predicted values. By repeating this process and rolling

it out to generate trajectories, we interact with the environment, collecting observations, and running our search to obtain visit counts and distributions. These critical functions are depicted in Equations 4.8.

$$\text{representation:} \quad h_\theta(o) \rightarrow s^0 \tag{4.8}$$

$$\text{dynamics:} \quad g_\theta(s^{k-1}, a^k) \rightarrow s^k, r^k \tag{4.9}$$

$$\text{prediction:} \quad f_\theta(s^k) \rightarrow p^k, v^k \tag{4.10}$$

These trajectories (Equation 4.11), in turn, are used to train and update our model, as seen in Equation 4.13. We use prioritized replay to sample a trajectory and state from our replay buffer. At the sampled state, we align the representation function, mapping it to the latent space of our model's embedding. By rolling the model forward using the real trajectory $s_1, ..., s_T$, we ensure that the model remains grounded and can predict the quantities produced during the actual search. This approach strengthens our model's ability to make accurate predictions during imaginary states, contributing to the effectiveness of our planning process.

$$(s^0, ..., s^t, a^t, ..., a^{t+n}) \rightarrow (p_t^0, v_t^0, r_t^1, ..., p_t^{n+1}, v_t^{n+1}, r_t^{n+1}) \tag{4.11}$$

The objective is to discover the policy π that maximizes the value $v(s_0)$ from the initial state, where the value is defined as the finite-horizon cumulative return, in Equation 4.12:

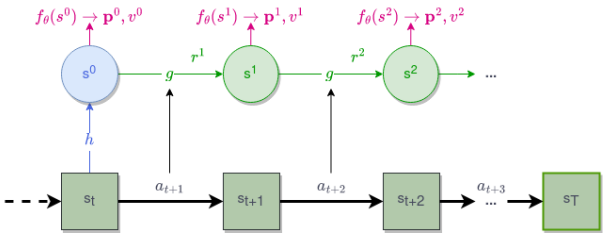

Figure 4.7: How MuZero trains its model.

$$v(s_0) = \mathbb{E}\left[\sum_{t=0}^{T} \gamma^t \cdot u_t \,\middle|\, s_0 = s\right] \tag{4.12}$$

Model:

$$\mu_\theta(o_1, ..., o_t, a_1, ..., a_k) = p^k, v^k, r^k \begin{cases} s^0 & = h_\theta(o_1, ..., o_t) \\ r^k, s^k & = g_\theta(s^{k-1}, a^k) \\ p^k, v^k & = f_\theta(s^k) \end{cases} \tag{4.13}$$

Note subscripts denote the time index in the real game environment, and superscripts index the timestep in the latent environment. Altogether, the entire computation graph implements the mapping, as illustrated in Figure 4.7:

$$s_t^0 = h_\theta(o_t, o_{t-1}, .., o_1) \tag{4.14}$$

$$f_\theta(s_t^0) = p_t^0, v_t^0 \tag{4.15}$$

$$g_\theta(s_t^0, a^1) = r_t^1, s_t^1 \tag{4.16}$$

Recall that t denotes the length of the sequence of observations (real states) on which we are training and t will also denote the current observation.

There are multiple things that need to be clarified here; we begin with the original notation of the dynamics function and note that it does not match with the loss function (starting at k=0 is infeasible), thus I suggest changing this to $g_\theta(s^k, a^{k+1}) \rightarrow s^{k+1}, r^{k+1}$ in order to accurately depict the flow of the dynamics function within the expansion of the tree. Additionally, we note that, even starting at the very first iteration of the loss sum, the smallest superscript obtained for the reward is 1. Hence, the first term of the published loss equation will never be fulfilled at k=0. Keeping to the notation of Figure 4.5, we also see that the target values u (the actual reward) for the imaginary reward r are always correlated to the next time step (t+1) and so this also serves as an argument for the, dare we call them, corrections to the original in Equation 4.16.

Using a single network with three distinct heads f, g, and h as functions, we leverage them to conduct a MCTS from state s_t in the latent space. Firstly, we encode the current state using h. Then, a variant of MCTS called PUCT [41] is employed, incorporating prior weights on available actions based on the policy network p_t^k [5]. Within the MCTS process, g and f govern the state transitions and policy/value predictions, respectively. As a result of the MCTS procedure, we obtain a policy $\pi_t = \pi(s_t)$ and a value estimate $v_t = v(s_t)$ for the root node, represented as $(\pi_t, v_t) \sim \text{MCTS}(s_0, \ldots, s_t | \mu_\theta)$. Subsequently, an action $a_t \sim \pi_t$ is chosen in the real environment, leading to a state transition, and the search process is repeated. This section is summarized in Equation block 4.17.

Search:

$$v_t, \pi_t = MCTS(s_t^0, \mu_\theta) \tag{4.17}$$

$$a_t \sim \pi_t \tag{4.18}$$

At each timestep t, an MCTS is executed, following the described procedure outlined in the section above. The search policy π_t guides the selection of the next action a_{t+1} based on the visit count of each action from the root node. The environment responds to the action by providing a new observation o_{t+1} and reward u_{t+1}. After an episode concludes, the trajectory data is stored in a replay buffer for later use. This process is depicted in Figure 4.9.

During the training process, a trajectory is randomly sampled from the replay buffer, and the sampled observations $o_1, \ldots, o_t$ undergo processing through the representation function h. Following this, the model, or latent states, are recurrently unrolled for K steps, as depicted in Figure 4.7. At each step k, the dynamics function g utilizes the previous hidden state s_{k-1} and the actual action a_{t+k} to generate the current hidden state s_k. By conducting joint training of the representation, dynamics, and prediction functions, the model's parameters are updated using backpropagation through time. The primary objective of this training process is to approximate the policy p_k as π_{t+k}, the value function v_k as z_{t+k}, and the reward r_k as u_{t+k}, where z_{t+k} corresponds to the sample return, either representing the final reward for board games or an n-step return for Atari. as seen in Equation 4.19.

$$z_t = \begin{cases} u_T & \text{for games} \\ u_{t+1} + \gamma u_{t+2} + ... + \gamma^{n-1} u_{t+n} + \gamma^n v_{t+n} & \text{for MDPs} \end{cases} \tag{4.19}$$

Losses:

$$l^P(\pi, p) = \pi^T logp \tag{4.20}$$

$$l^v(z, v) = \begin{cases} (z - v)^2 & \text{for games} \\ \phi(z)^T logv & \text{for MDPs} \end{cases} \tag{4.21}$$

$$l^r(u, r) = \begin{cases} 0 & \text{for games} \\ \phi(u)^T logr & \text{for MDPs} \end{cases} \tag{4.22}$$

$$l_t(\theta) = \sum_{k=0}^{K} l^P(\pi_{t+k}, p_t^k) + \sum_{k=0}^{K} (z_{t+k}, v_t^k) + \sum_{k=1}^{K} l^r(u_{t+k+1}, r_t^{k+1}) \tag{4.23}$$

The Atari environment presents unique challenges for the other two algorithms, AlphaGo and AlphaZero, due to its specific characteristics and requirements. One of the major difficulties arises from the need to clone the Atari state during the tree search process. Unlike board games like Go and Chess, where states are well-defined and can be easily represented, the Atari environment involves complex pixel-based inputs, making it harder to create a precise state representation. Cloning the Atari state accurately is essential for performing tree search effectively, as it forms the foundation for evaluating potential moves and planning future actions. The intricate nature of Atari states demands significant computational resources and poses a considerable

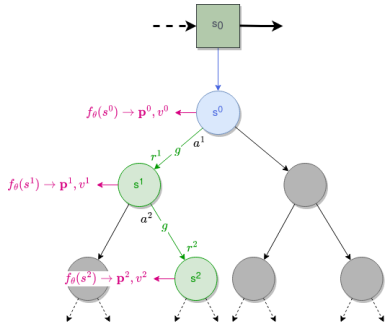

Figure 4.8: Demonstrating how MuZero utilizes it's learned model to plan. The learned model consists of three connected components for f_θ, g_θ and h_θ.

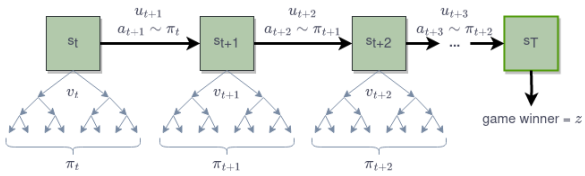

Figure 4.9: How MuZero acts in the environment.

technical hurdle for the other algorithms to adapt and perform optimally. This unique challenge highlights the need for tailored approaches and adaptations to leverage the strengths of AlphaGo and AlphaZero in the context of Atari games, making it an exciting area for further research and development.

4.7.1 MuZero Methods

To ensure coherence across the extensive MuZero model, all individual networks (h_θ, g_θ, and f_θ) shared a unified internal architecture. The encoding function h_θ solely processed present observations, independent of trajectories. Moreover, the dynamics functions integrated one-hot-encoded actions, combined with the current latent state, and then transmitted to g_θ. The action policy, value function, and reward prediction were represented as probability

Table 4.2: Latent roll-out depth (K), and latent dimensionality ($L = |s^k|$)

Hyperparameters	Pong	Pinball
L	4	4
K	5	10
t	0.5	0.5

distributions and trained using a cross-entropy loss, consistent with the original research [24]. For other unspecified hyperparameters, please refer to Table 4.2.

Ultimately, MuZero adopts dual normalization techniques to ensure stable learning. It leverages min-max normalization as the output activation for both h_θ and g_θ. Concretely, every abstract state undergoes adjustments according to the formula expressed in Equation 4.24:

$$s_{\text{scaled}} = \frac{s - \min(s)}{\max(s) - \min(s)} \tag{4.24}$$

This process aligns the range of s with that of the discrete actions, but it also introduces a noteworthy constraint, particularly when the latent space is compact. For instance, when $L = |s^k| = 4$, the min-max normalization enforces one element of $s \in [0, 1]$ For specific details on the latent space dimensionality parameters, refer to the information provided in Table 4.2.

Our MuZero implementation trained at $K = 5$ and 10 latent steps. Training proceeded for 100k mini batches of size 64 for the board game environments and 512 for the Atari environments.

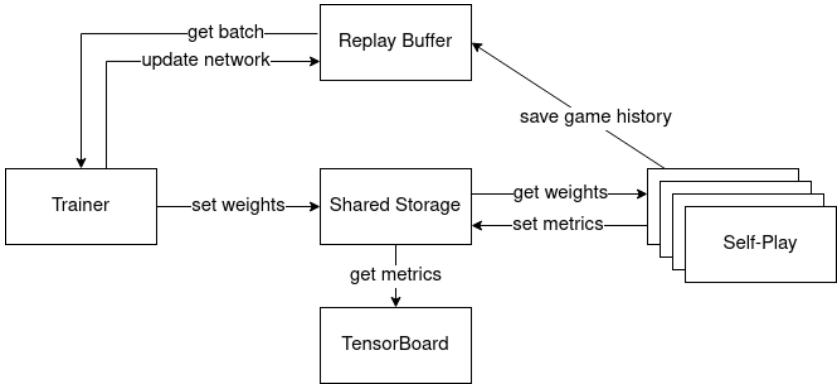

Figure 4.10: Alpha's Code Structure

4.8 The Alphas

We present a comprehensive reimplementation of AlphaGo, AlphaZero and MuZero, adhering as much to the original paper as feasibly possible [5, 6, 24]. Our implementations are written in Python, utilizing Tensorflow 2.0 [84], whose structure is shown in Figure 4.10.

In summary, the algorithms differ in their tree-policy, where AlphaGo utilizes prior probabilities from SL, AlphaZero relies on self-play probabilities, and MuZero employs learned model-based predictions. The default-policy varies as well, with AlphaGo using expert moves, AlphaZero employing a Monte Carlo policy, and MuZero learning its default policy. Furthermore, the simulation length is fixed for AlphaGo and AlphaZero but variable for MuZero, allowing it to adapt to different domains effectively. Additionally, AlphaGo uses separate NNs for policy and value estimation, while AlphaZero and MuZero employ a single network for both tasks, trained through self-play in the case of AlphaZero and integrated with a learned model for MuZero.

Table 4.3: Comparison of properties shared among the Alphas

	AlphaGo	AlphaZero	MuZero
Tree Policy	MCTS with prior probabilities from supervised learning	MCTS with prior probabilities from self-play	MCTS with learned model-based predictions
Default Policy	Monte Carlo Policy	Monte Carlo Policy	Learned along with the value and policy networks as part of the integrated learned model in MuZero
Neural Network Structure	Separate neural networks for policy and value estimation	Single neural network for both policy and value estimation, trained through self-play	Integrated learned model for state transitions, value, and policy networks, shared backbone for policy and value heads
Computational Resource	32 GPU 10 CPU	32 GPU 10 CPU	32 GPU 10 CPU
Training Time	150 hours	150 hours	150 hours

These distinctions highlight the evolution and adaptability of each algorithm, contributing to their unprecedented performance in mastering complex games.

4.8.1 Resource Requirements and Restraints

The Alphas have showcased remarkable accomplishments in the realm of game-playing AI. However, it is important to note that these achievements come at the cost of tremendous computational resource requirements. The training process of these algorithms involves intensive computation, which demands

Table 4.4: The Alpha's success is tremendous, but so it their resource requirements.

Model	Self-Play	Training	Frames
AlphaGo	-	2000 GPUs	100 M
AlphaZero	64 TPUs	8 TPUs	200 M
MuZero	1000 TPUs	16 TPUs	20 B

significant computational power, time, and resources. DeepMind's original implementations of these algorithms made use of large-scale distributed systems with multiple CPUs, GPUs and TPUs working in parallel. The training of NNs in MuZero, for instance, employed 16 TPU workers for training and 1k TPUs for self play. For AlphaZero, depending on the version could be anywhere from 176 GPUs to 48 TPUs. Please see Table 4.4 for standarized resource requirement for original algorithms. This resource-intensive training approach allowed the algorithms to learn and refine their strategies through an extensive exploration of game states. While these computational requirements pose challenges in terms of scalability and accessibility, they also highlight the significant strides made in game-playing AI and the immense computational power harnessed to achieve such groundbreaking results.

In order to establish a fair and level playing field for the evaluation and comparison of the Alpha family of algorithms in this thesis, we have imposed limitations on the computational resources allocated to each algorithm. By ensuring that all algorithms are trained with an equal amount of computational power, we aim to mitigate potential biases arising from disparate resource utilization. This approach enables a more objective assessment of the algorithmic improvements and advancements made in each iteration. Each algorithm's NNs, including their respective optimization processes, are implemented using

the same framework and executed on the same infrastructure. By enforcing uniformity in computational resources and time allocation, we reduce the impact of resource disparities on the performance outcomes. This rigorous and controlled experimental setup allows us to better isolate and evaluate the specific algorithmic enhancements and modifications, providing valuable insights into the strengths and weaknesses of each approach. Moreover, this approach aligns with scientific principles of fairness and repeatability, enabling robust comparisons and meaningful conclusions to be drawn from the experimental results.

Another aspect to consider is the computational time required for training and evaluation. While our study strives to maintain consistent computational constraints across the algorithms, it's important to acknowledge that the actual implementation and hardware dependencies can influence performance. Different implementations might exhibit varying levels of efficiency, potentially resulting in faster execution times for certain algorithms. However, the fact that the Alpha algorithms share the same framework somewhat mitigates these disparities, as the underlying components remain consistent.

Nonetheless, the intricacies of software optimization and hardware compatibility can introduce a layer of unpredictability. A seemingly minor tweak in code could significantly impact execution speed on a specific architecture. It's also worth noting that our evaluation doesn't delve into the realm of code optimization tailored to specific algorithms. While we attempt to control variables, the reality is that the AI landscape is nuanced, and even seemingly minute changes in implementation details could yield diverse outcomes.

Despite our efforts to measure CPU and GPU utilization, a more intricate

assessment reveals that MuZero's operational and computational complexity surpasses that of the other Alpha algorithms. This distinction arises from MuZero's utilization of a larger neural network architecture, enabling it to execute a greater number of simulations compared to its Alpha algorithm counterparts. Furthermore, it's worth noting that AlphaGo employs four distinct neural networks for various purposes, which adds another layer of complexity to its computational demands.

The inherent complexity of these algorithms invites the possibility of conducting a more in-depth analysis using Big-O complexity notation. This could provide insights into the algorithms' scalability and efficiency as their computational requirements grow with increasing problem sizes. However, a comprehensive Big-O complexity analysis might prove challenging due to the intricate interplay of neural network computations, MCTS operations, and parallel processing, all of which contribute to the overall operational complexity. Nevertheless, such analysis could potentially yield valuable insights into the algorithms' behavior as the problem scales, shedding light on their potential limitations and optimal utilization.

In conclusion, our study contributes valuable insights into the performance of Alpha algorithms within a controlled experimental setup. Yet, the broader context of AI research reminds us that findings can be influenced by a multitude of factors. The dynamic interplay between computational resources, code implementation, and the inherent complexities of game-playing domains underscores the need for comprehensive explorations in various scenarios to paint a complete picture of the capabilities and limitations of these cutting-edge algorithms.

The NNs utilized in AlphaGo, AlphaZero, and MuZero, represented as f_θ, underwent optimization using TensorFlow, a powerful deep learning framework, in Python, and executed on the Google Cloud platform. This endeavor required the collaborative effort of 32 GPU workers alongside 10 CPU parameter servers. Each worker performed training with a mini-batch size of 64, collectively amounting to a substantial batch size of 2,048. To ensure a well-rounded training dataset and promote efficient NN learning, we employed a sampling strategy that uniformly selected data from 500k games of self-play, contributing to the diverse training process.

4.9 Atari and Atari Learning Environment

Atari and the Arcade Learning Environment (ALE) have played a significant role in advancing the field of RL. Atari games serve as a benchmark for testing and evaluating the performance of various RL algorithms due to their complexity, diverse dynamics, and rich visual representations. The ALE, developed by Marc Bellemare and colleagues[85], provides a standardized platform for researchers to interact with Atari games, enabling fair comparisons and reproducibility.

In this thesis, we utilized the Atari games and the ALE as a testbed to evaluate the performance of the Alpha algorithms and compare it to the DQN baseline. By training and testing these algorithms on a range of Atari games, I aimed to investigate their abilities to learn and generalize across various game environments. The use of Atari and the ALE allowed me to assess the algorithms' performance in challenging and dynamic domains, providing insights

into their strengths, limitations, and potential for real-world applications.

Furthermore, the availability of a wide range of Atari games within the ALE allowed for comprehensive experimentation and analysis. I utilized game-specific performance metrics, such as average episode rewards and achieved scores, to quantify the algorithms' effectiveness and compare their performance across different games. This evaluation on the Atari platform served as a valuable means to assess the capabilities of the algorithms and understand their behavior in complex and dynamic environments.

Overall, Atari and the ALE have been instrumental in driving research and progress in RL. They provide a standardized and challenging testbed that enables researchers to develop, evaluate, and compare novel algorithms, ultimately contributing to the advancement of AI techniques in solving complex real-world problems.

Using Atari games in the MCTS algorithm presents a unique challenge since the states must be close enough to the original environment states in ordre to effectively simulate the game until the end. Unlike MuZero, which overcomes this issue through its self-play and simultaneous learning process, other algorithms like AlphaGo and AlphaZero face difficulties when dealing with large state spaces and complex game environments like Atari. This implementation challenge arises because in Atari games, reaching the terminal state requires navigating through numerous intermediate states, making the MCTS more computationally demanding.

MuZero, on the other hand, benefits from its ability to predict future states and rewards without having to fully simulate the game until the end. Through its NN architecture and Monte Carlo planning, MuZero can efficiently gen-

erate action sequences without relying on complete game simulations. This advantage allows MuZero to handle large and complex state spaces, making it particularly well-suited for games like Atari, where exhaustive simulations might be infeasible due to the vast number of possible states.

Highlighting the strength of MuZero in addressing this challenge sheds light on the benefits of its unique approach to self-play and learning. By directly predicting state transitions and rewards, MuZero avoids the need for exhaustive simulations, making it more adaptable to various game environments. This advantage could be crucial for game-playing AI algorithms, as it allows for efficient learning and decision-making even in complex and high-dimensional domains like Atari.

4.10 Experiments and Methods

4.10.1 Environments

We have already described in depth the Connect4 and Othello Environments in Section 3.6.1. Pong and Pinball were chosen as additional environments in our experiment due to their characteristics as zero-sum two-player games. This deliberate selection allowed us to explore the algorithms' performance in competitive settings where maximizing one player's reward directly impacts the other player's reward. By incorporating these games, we aimed to assess how the algorithms handle adversarial situations and strategic decision-making in a more dynamic and interactive environment. Moreover, these games presented distinct challenges compared to Connect4 and Othello, allowing us to examine the algorithms' adaptability across diverse domains and their potential for

broader applicability.

Pong is a classic Atari game that simulates a table tennis match.In Pong, players control paddles on opposite sides of the screen, aiming to hit the ball past their opponent's paddle and score points. Each time the ball successfully passes the opponent's paddle, a point is awarded to the player. The game continues until one player reaches a predetermined score or a time limit is reached. Pong was chosen as an environment for the algorithms because it is a simple yet competitive game that involves precise timing, strategic positioning, and quick decision-making. Its zero-sum nature, where one player's gain is directly at the expense of the other player, makes it a suitable choice for evaluating the effectiveness of self-play algorithms like MuZero and AlphaZero in learning optimal strategies through RL. The state space is an RGB image tensor of shape 210x160x3 then reduced to that of 84x84x4. The input action is an integer in [0,1,2,3,4,5], which denotes the following meanings: 0 : no action , 1 : fire , 2 : right , 3 : left, 4 : rightfire, 5 : leftfire.

Video Pinball, another Atari game, emulates the classic pinball arcade experience. The objective in Video Pinball is to keep a ball in play by controlling two paddles at the bottom of the screen and hitting targets on a pinball-like table. Players score points by hitting various objects and targets, such as bumpers, spinners, and ramps. The more accurately the ball is hit and the more targets are hit consecutively, the higher the score. Video Pinball was selected as an environment for the algorithms due to its rich dynamics, requiring players to consider timing, ball trajectory, and skillful paddle control. The scoring mechanics and the strategic decision-making involved in maximizing points make Video Pinball a challenging and interesting domain for exploring

the capabilities of self-play algorithms in learning optimal gameplay strategies. The state space in the Video Pinball environment is an RGB image tensor, typically with a shape of 210x160x3, which is then resized to 84x84x4 for processing. The input actions are integers ranging from 0 to 17, corresponding to different button combinations for controlling the flippers and launching the ball.

4.10.2 Deep-Q Learning - DQN

Deep Q-Network (DQN) [3] is a RL algorithm that belong to the Q-learning family. DQN introduced the use of DNNs to approximate the Q-values, enabling it to handle high-dimensional state spaces. It leverages an experience replay buffer to store and sample from past experiences, allowing for better data efficiency and reduced correlation between samples. This method uses a functon approximator to estimate the action-value function $Q(s, a; \theta) \approx Q^*(s, a)$. We train NN θ by minimizing the loss function in Equation 4.25 [17]:

$$L_i(\theta_i) = \mathbb{E}[(y_i - Q(s, a; \theta_i))^2] \tag{4.25}$$

where $y_i = \mathbb{E}[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}]$ is the target for iteration i and the parameters θ_{i-1} stay fixed when optimizing the loss. We then optimize the loss function by stochastic gradient descent. We note that this algorithm is model-free, as it learns via direct samples from the environment, without constructing a model. This method learns via a ϵ-greedy strategy $a = \max_a Q(s, a; \theta)$ where it selects random actions with probability ϵ. Refer to Table 4.5 for specific parameters used in the implementation.

Table 4.5: Hyperparameters for the DQN Algoritm

Hyperparameters	DQN
Replay Buffer Capacity	100k
Agent Update Freq.	500
Target Update Freq.	250
Max ϵ	1
Min ϵ	0.05
Learning Rate	1×10^{-3}
Batch Size	64

4.11 Results

Starting with the board game environments, Connect4 and Othello, we observed intriguing results as MuZero demonstrated superior performance compared to its Alpha counterparts. However, an interesting observation was that DQN achieved a remarkably similar performance to AlphaGo in Connect4. Moreover, we noticed a significant advantage for the first player in Connect4, as evidenced by the results in Table 4.6. In contrast, the Othello environment did not exhibit a significant advantage for the first player, as shown in Table 4.7.

To assess the capabilities of the Alpha algorithms in different environments, we conducted a thorough comparison against the benchmark performance of DQN. The diverse challenges presented by these environments did not appear to hinder MuZero's performance, nor significantly improve AlphaZero's. Figures 4.13 to 4.14 display the average training curves for each of the four algorithms on the four different environments. Interestingly, we observed consistent patterns, where the size and complexity of the environments did not seem to substantially affect the performance of the MuZero algorithm.

Table 4.6: Connect 4 Statistics. Won games over 5k games where agent initialized for both players equally.

Algorithm	# Player 1 wins	# Player 2 wins	# of tie games	Move Count per game
AlphaGo	3040	1905	55	22.102
AlphaZero	3325	1653	22	21.764
MuZero	3420	1580	0	18.44
DQN	2854	2086	60	23.252

On average, DQN (a model-free approach) performed less successfully than the Alpha algorithms, with AlphaGo and AlphaZero (both utilizing a model) following closely behind, while MuZero (learning the model) demonstrated the highest level of performance. This not only highlights MuZero's unique characteristic of not requiring a model, unlike other MCTS techniques, but also showcases its superiority over model-free techniques like DQN, making it an extremely efficient and effective choice.

Furthermore, we observed that MuZero effectively navigates the challenges presented by both dense reward environments, like the Atari games, where it quickly obtains gradients to structure the latent space, and sparse reward environments. Despite the larger and more complex environments, MuZero maintained its exceptional performance, further reinforcing its versatility and adaptability.

We approach the statistical analysis in a very similar way to which we approached the challenge in Chapter 3. For the Connect4 environment, we conduct an ANOVA test which detected a statistically significant difference in reward among at least two algorithms. We found a p-value of 0.039 which is close to our alpha parameter but still indicates that we can reject the null

Table 4.7: Othello Statistics. Won games over 5k games where agent initialized for both players equally.

Algorithm	# Player 1 wins	# Player 2 wins	# of tie games	Move Count \per Game
DQN	2869	2083	48	68.254
AlphaGo	2941	2023	46	65.397
AlphaZero	2097	1887	116	64.168
MuZero	3295	1693	12	58.621

Table 4.8: Average total reward over 50k episodes of Atari games

	Average Total Reward		Single Best Performing Episode	
	Pong	Video Pinball	Pong	Video Pinball
DQN	-3	76,421	10	144,253
AlphaGo	5	134,217	14	589,326
AlphaZero	11	221,579	21	501,978
MuZero	14	512,365	21	537,942

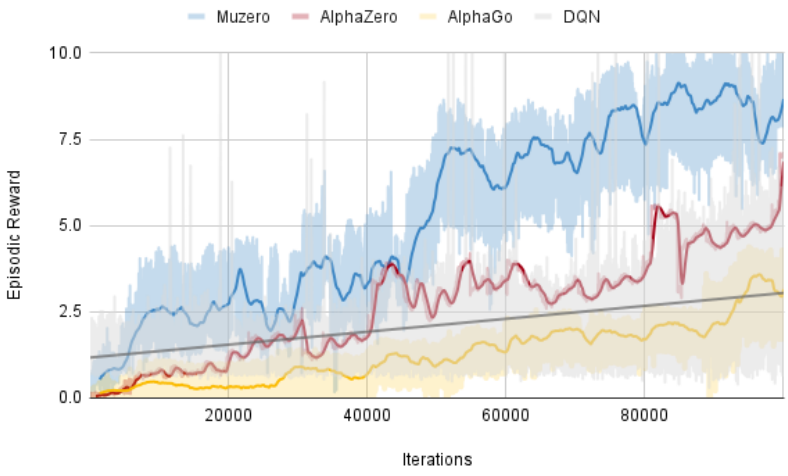

Figure 4.11: Evaluation of the Alphas throughout training in Connect4. DQN is shown as baseline. Averaged across 10 experiment replications

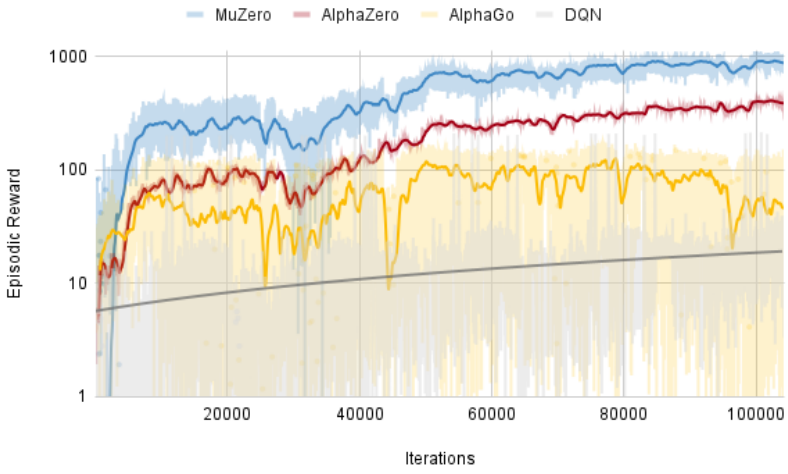

Figure 4.12: Evaluation of the Alphas throughout training in Othello. DQN is shown as baseline. Averaged across 10 experiment replications

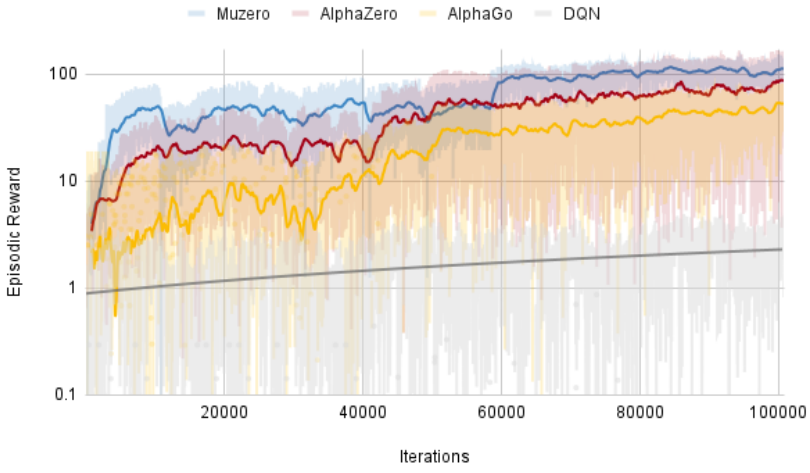

Figure 4.13: Evaluation of the Alphas throughout training in Pong. DQN is shown as baseline. Averaged across 10 experiment replications

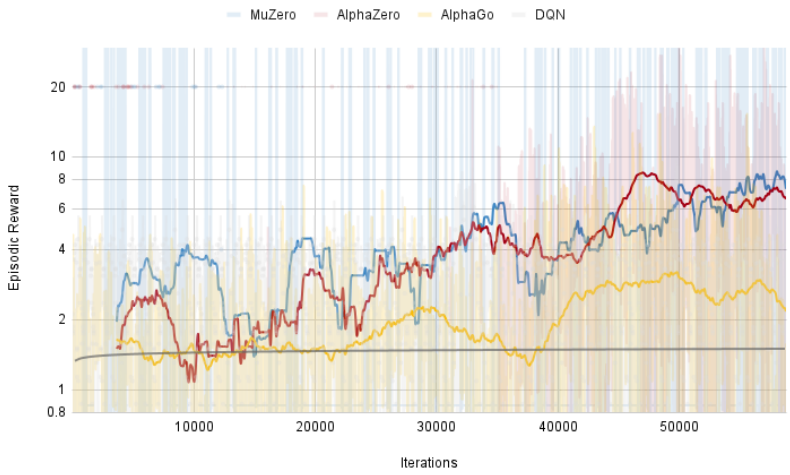

Figure 4.14: Evaluation of the Alphas throughout training in Pinball. DQN is shown as baseline. Averaged across 10 experiment replications

hypothesis and conclude that the four means are not all equal. We also conducted a post-hoc Tukey's HSD test and found that the difference in win rates between AlphaGo and AlphaZero was significant (p = 0.010 , 95% C.I. = [38.63,77.98]), as was the difference between AlphaZero and MuZero (p = 0.042, 95% C.I. = [67.84,97.28]). No significant difference was found between DQN and AlphaGo (p = 0.074).

We repeat the same ANOVA test for each of the remaining environments: Othello, Pong and Pinball. with and obtain a p = 2.88E-04, p = 0.039 and p = 5.64E-05 value which also indicates that we are able to reject the null hypothesis and conclude that the means are not equal for each of the environments. We also conducted a post-hoc Tukey's HSD test for each of these, which we will go through in order.

The Othello Tukey test found that the difference in win rates between

DQN and AlphaGo was significant (p = 0.012, 95% C.I. = [38.99 , 88.17]), as is the difference between AlphaGo and MuZero (p = 0.003, 95% C.I. = [45.18, 82.22]). No significant difference was found between AlphaZero and MuZero (p = 0.333). The Pong Tukey test found that the difference in win rates between all the pairs of algorithms. DQN and AlphaGo (p = 3.233E-04, 95% C.I. = [32.72 ,78.23]) and DQN and AlphaZero (p = 6.53E-05, 95% C.I. = [40.49, 93.91]) and DQN and Muzero (p= 5.63E-04, 95% C.I. [8.59, 96.36]) Followed by AlphaGo and AlphaZero (p= 4.43E-03, 95% C.I. [17.48, 66.43]), AlphaGo and MuZero (p= 1.33E-06, 95% C.I. [58.39, 74.38]) and lastly, AlphaZero and MuZero (p= 0.045, 95% C.I. [23.44, 57.99]) The last Tukey test we conducted was within the Pinball environment. We found that there was no significant difference between the win rates between AlphaZero and MuZero (p = 0.183). Every other algorithm pair comparisons and results were statistically different.

The results of our comparative evaluation reveal intriguing insights into the performance of the implemented algorithms on the Connect4, Othello, Pong, and video pinball environments. Notably, MuZero consistently outperforms all other algorithms, showcasing its remarkable capabilities in strategic decision-making and adaptive learning. However, our analysis goes beyond mere performance comparison and emphasizes the improvement and significance of MuZero, even when accounting for its intensive training. Our results suggest that MuZero's success stems not only from its resource-intensive nature but from its unique ability to generalize and plan without prior domain knowledge. These findings solidify the position of MuZero as a groundbreaking algorithm in the realm of RL and underscore its potential for real-world

applications. Furthermore, our results highlight the strengths and limitations of AlphaGo and AlphaZero providing valuable insights for future research and algorithmic developments in the field.

4.12 Limitations

Despite the comprehensive evaluation and insightful findings presented in this work, there are some limitations that should be acknowledged. Firstly, our study focused primarily on zero-sum game environments, and the performance of the Alpha algorithms in multi-agent settings remains an area for further investigation. Additionally, our evaluation was constrained by computational resources and training time, potentially limiting the algorithms' full potential. Moreover, the choice of gaming environments may not fully represent the diversity and complexity of real-world problems, warranting further exploration in more challenging domains. Lastly, while MuZero showcased remarkable adaptability, its training regime might be challenging to replicate in resource-constrained scenarios, raising questions about its practical applicability in certain real-world applications. Despite these limitations, this work provides a solid foundation for future research and highlights the impressive capabilities of the Alpha algorithms in RL.

In addition to the factors mentioned earlier, there are other variables that we did not explicitly control, which could potentially impact the results. One such factor is the difference in internal policies used by the Alpha algorithms during the training process. Each algorithm may adopt different exploration strategies and policies for guiding the search process, leading to variations

in their decision-making and learning behaviors. Furthermore, the specific training loops and hyperparameter settings used for each algorithm could also contribute to differences in their performance. The choice of learning rates, batch sizes, and other training hyperparameters can significantly influence the speed and stability of the learning process, potentially affecting the final results. While we made efforts to maintain consistency in the implementation and training settings, these variations remain potential sources of influence on the performance of the Alpha algorithms in our experiments. For example, there could be a set of hyperparameters where different algorithms will perform better and differently than our results. Acknowledging and understanding these uncontrolled factors helps to contextualize the results and encourages further investigation to identify their individual impacts.

4.13 Conclusion

In conclusion, our comprehensive evaluation of the Alpha algorithms, including AlphaGo, AlphaZero, and MuZero, alongside the baseline DQN, has revealed valuable insights into their performance on diverse gaming environments. Notably, MuZero demonstrated superior capabilities in strategic decision-making and adaptive learning, outperforming all other algorithms consistently. Importantly, we found that MuZero's success can be attributed not only to its resource-intensive training but also its unique ability to generalize and plan without prior domain knowledge. These findings solidify MuZero's position as a groundbreaking algorithm in RL, showcasing promising potential for real-world applications. Furthermore, our analysis shed light

on the strengths and limitations of AlphaGo, AlphaZero, and DQN, providing valuable guidance for future research and algorithmic developments in the field.

5 Conclusion and Future Work

5.1 Conclusions

In this comprehensive thesis, we delve into a thorough exploration and meticulous evaluation of various MCTS algorithm variations within the context of the AlphaZero paradigm. Furthermore, we undertake the endeavor of conducting a comparative analysis of diverse members within the Alpha algorithm family, all under the same computational resources.

Chapter 2 serves as the foundation, offering the essential background and fundamental concepts required for a coherent comprehension of the subsequent algorithms and discussions presented throughout this thesis. Proceeding to Chapter 3, we delve into a detailed evaluation of an array of MCTS variations, both within the original MCTS algorithm framework and integrated into the AlphaZero algorithm. We set all our eight implemented algorithms against three test opponents and obtained win rates for each match, averaged over multiple games. Notably, our findings highlight the emergence of the AlphaZero-ALL algorithm as a standout performer across a range of carefully chosen game environments.

In Chapter 4, our attention pivots towards an in-depth assessment of the fundamental Alpha algorithms. This evaluation takes place across a meticulously curated set of diverse game environments, with the aim of gaining comprehensive insights into the algorithms' performance across varying challenges and complexities. While all three Alpha algorithms demonstrate exceptional performance, the latest and most robust entrant, MuZero, consistently emerges as the front runner in every environment evaluated within this thesis. This resounding achievement extends to complex Atari games like Pong and Pinball, solidifying MuZero's status as the preeminent algorithm among its counterparts.

This work represents a commendable attempt at exploring and improving game-playing AI algorithms, particularly focusing on the Alpha algorithms and MCTS modifications for various environments. By comprehensively evaluating and comparing the DeepMind algorithms and modifications to the MCTS algorithm, the thesis advances our foundational understanding of their capabilities and limitations. The findings of this thesis hold promise for the continued development and refinement of strategic decision-making systems, propelling the field of AI into new realms of achievement.

5.2 Limitations

This thesis entails certain limitations that deserve consideration. Firstly, the choice of game environments for evaluation, namely Connect4 and Othello, alongside the absence of more diverse and complex real-world tasks like Atari games, might limit the generalization of results. The similarities of these

games to Go could potentially bias the algorithms' performance, favoring those designed explicitly for Go-like scenarios.

Moreover, the variations in network architectures and distinct MCTS implementations employed by each algorithm in Chapter 4 could introduce biases and impact their respective performances. Understanding the isolated effects of individual components becomes challenging due to these differences.

It is essential to note that the computational resources used during the original experiments were substantial, which, while necessary for achieving state-of-the-art performance, might not be readily available to all researchers and practitioners. Consequently, the practical applicability of the proposed algorithms in resource-constrained settings might be limited. In our work we simply limited the resources as a proposed mean to control one of the influencing variables.

Additionally, the evaluation metrics utilized in this thesis provide a comprehensive assessment of the algorithms' capabilities. However, there might be other relevant metrics or real-world benchmarks that could further enhance the evaluation and demonstrate the agents' strengths and weaknesses more effectively.

While our evaluation offers valuable insights into the performance of various algorithms within identical computational constraints, it's important to consider the limitations of our approach. By controlling factors like network size and computational resources, we aim to ensure a fair comparison among the algorithms. However, it's crucial to acknowledge that other variables, such as hyperparameters and neural network architectures, could impact the results. Confounding variables that we haven't accounted for might also play

a role in influencing the outcomes.

Furthermore, the results we present here provide only a limited glimpse into the capabilities and limitations of the Alpha algorithms. Each algorithm's performance might be contingent on specific game environments, opponents, and conditions, which our evaluation might not fully encapsulate. The dynamic nature of AI research and the complexities of game-playing domains make it challenging to draw definitive conclusions from a single study.

In summary, while our experiments shed light on the comparative performance of the Alpha algorithms, they represent only a single snapshot in the broader landscape of AI research. As the field continues to evolve, a comprehensive understanding of the algorithms' strengths and limitations will require rigorous exploration across a wider range of variables and conditions.

By acknowledging these limitations, this thesis offers valuable contributions to the field of AI and RL, showcasing the potential of deep learning and MCTS techniques in mastering complex game environments. Identifying these limitations also paves the way for future research to build upon this work, address the challenges, and explore opportunities for advancing AI algorithms beyond the scope of this thesis.

5.3 Future Work

5.3.1 MCTS in Stochastic Environments

Stochastic environments pose unique challenges for MCTS due to the inherent variability and uncertainty of outcomes. In these environments, the main challenge lies in accurately estimating the value of actions or states, as the

outcomes of simulations can be unpredictable and noisy. Such challenges can lead to suboptimal decisions and slower convergence to the optimal policy. However, several techniques have been developed to address these challenges and enhance the performance of MCTS in stochastic environments.

One of the primary challenges in stochastic environments is handling the variability of outcomes during simulations. To tackle this, rollout policies are employed to guide the simulation towards more promising paths. Rather than randomly sampling actions, rollout policies utilize learned models, such as NNs, to predict likely outcomes for each action. By incorporating these predictions, rollout policies improve the accuracy of value estimates, allowing for more informed decision-making.

Another challenge is the accurate estimation of values in the presence of noise and uncertainty. Importance sampling is a technique used to adjust the weight of each simulation based on the likelihood of being a reasonable estimate of the actual value. By considering the probabilities of simulated outcomes, importance sampling reduces the impact of noisy simulations and enhances the accuracy of value estimates.

The exploration-exploitation trade-off is crucial in stochastic environments. Progressive widening is a technique used to balance this trade-off based on the number of visits to each node. It progressively increases the exploration of new actions or states as the search progresses while favoring the most promising options based on accumulated information. This approach ensures a balanced exploration of the search space, leading to more effective decision-making.

Ensembling is a powerful technique that combines multiple estimators to improve the accuracy of value estimates. In the context of MCTS, ensembling

involves aggregating outputs from different MCTS algorithms with diverse exploration strategies. By leveraging the strengths of each algorithm, ensembling provides a more robust and accurate estimate of the optimal solution in stochastic environments.

To implement these techniques effectively, adaptations in the MCTS algorithm and exploration strategies are necessary. This may involve addressing computational resource requirements by utilizing parallelization techniques and optimizing the search algorithm to reduce computational complexity. Additionally, incorporating domain-specific knowledge and tuning hyperparameters can further enhance the performance of MCTS in stochastic environments.

In summary, stochastic environments present unique challenges for MCTS, as the variability and uncertainty of outcomes can hinder accurate value estimation and decision-making. However, through the utilization of rollout policies, importance sampling, progressive widening, and ensembling, the performance of MCTS can be significantly improved in these complex and uncertain domains. These techniques enable more effective exploration and exploitation of the search space, leading to better decision-making and faster convergence to optimal policies in a wide range of stochastic environments.

5.3.2 Future work - MCTS Impact

In terms of future work, one intriguing avenue for further exploration is to conduct experiments that minimize the influence of MCTS in the implemented algorithms. By significantly reducing the activity of MCTS or even removing it entirely, we can gain deeper insights into the specific impact of MCTS on the algorithms' performance. This experiment would enable us to isolate

and analyze the contributions of other components, such as the DNNs and the learned model, in decision-making and strategic planning. Understanding the relative importance of MCTS and its interplay with other elements could provide valuable knowledge for optimizing and fine-tuning these algorithms, potentially leading to more efficient and effective models. Such investigations into the role of MCTS would contribute to our broader understanding of RL techniques and pave the way for advancements in algorithm design and optimization.

However, it is important to note that minimizing the activity of MCTS in these algorithms also poses certain challenges and limitations. One significant consideration is the need for environment-specific tuning of the NNs utilized in these algorithms. Neural networks often require careful calibration and optimization to adapt to the nuances and complexities of each environment. By reducing the influence of MCTS, we risk diminishing the role of exploration and relying solely on the NNs' existing knowledge. This approach might lead to suboptimal performance or an inability to adapt effectively to diverse environments. Hence, striking a balance between the exploration capabilities of MCTS and the adaptability of NNs is crucial to ensure optimal performance across different domains. Finding the right trade-off between these elements remains a key challenge that future research must address to maximize the potential of these algorithms.

Bibliography

[1] M. Campbell, A. Hoane, and F. Hsu, "Deep blue," *Artificial Intelligence*, vol. 134, pp. 57–83, 2002.

[2] L. V. Allis, *Searching for Solutions in Games and Artificial Intelligence*. PhD thesis, Univ. Limburg, Maastricht, The Netherlands, 1994.

[3] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–533, 2015.

[4] M. Taves, "Google's alphago isn't taking over the world, yet," March 13 2016.

[5] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[6] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, *et al.*, "Mastering

the game of go without human knowledge," *Nature*, vol. 550, no. 7676, pp. 354–359, 2017.

[7] DeepMind, "The challenge match: Alphago vs. lee sedol." `https://www.deepmind.com/research/highlighted-research/alphago/the-challenge-match`, Year of Access. Accessed: [Insert Access Date].

[8] F.-Y. Wang, J. J. Zhang, X. Zheng, X. Wang, Y. Yuan, X. Dai, J. Zhang, and L. Yang, "Where does alphago go: From church-turing thesis to alphago thesis and beyond," *IEEE/CAA Journal of Automatica Sinica*, vol. 3, no. 2, pp. 113–120, 2016.

[9] S. D. Holcomb, W. K. Porter, S. V. Ault, G. Mao, and J. Wang, "Overview on deepmind and its alphago zero ai," in *Proceedings of the 2018 international conference on big data and education*, pp. 67–71, 2018.

[10] Y. Chen, A. Huang, Z. Wang, I. Antonoglou, J. Schrittwieser, D. Silver, and N. de Freitas, "Bayesian optimization in alphago," *arXiv preprint arXiv:1812.06855*, 2018.

[11] DeepMind, "Deepmind ai reduces google data centre cooling bill by 40%." `https://www.deepmind.com/blog/deepmind-ai-reduces-google-data-centre-cooling-bill-by-40`, 2023. Accessed: [Insert Access Date].

[12] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago, *et al.*, "Competition-level code generation with alphacode," *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022.

[13] J. Jumper, R. Evans, A. Pritzel, T. Green, M. Figurnov, O. Ronneberger, K. Tunyasuvunakool, R. Bates, A. Žídek, A. Potapenko, *et al.*, "Highly accurate protein structure prediction with alphafold," *Nature*, vol. 596, no. 7873, pp. 583–589, 2021.

[14] A. Senior, J. Jumper, R. Evans, T. Green, P. Ewels, J. Preece, D. Taylor, D. Fields, G. Prodromou, A. Antoniou, *et al.*, "Alphafold: Using ai for scientific discovery." DeepMind Blog, 2020.

[15] J. A. de Vries, K. S. Voskuil, T. M. Moerland, and A. Plaat, "Visualizing muzero models," *arXiv preprint arXiv:2102.12924*, 2021.

[16] S. Thakoor, S. Nair, and M. Jhunjhunwala, "Learning to play othello without human knowledge," 2016.

[17] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. MIT Press, 2018.

[18] O. Nachum, M. Norouzi, K. Xu, and D. Schuurmans, "Bridging the gap between value and policy based reinforcement learning," *CoRR*, vol. abs/1702.08892, 2017.

[19] R. S. Sutton and A. G. Barto, "Introduction to reinforcement learning," *MIT press*, 1998.

[20] I. Osband, C. Blundell, A. Pritzel, and B. Van Roy, "Deep exploration via bootstrapped dqn," *arXiv preprint arXiv:1602.04621*, 2016.

[21] M. Andrychowicz, A. Raichuk, P. Stańczyk, M. Orsini, S. Girgin, R. Marinier, L. Hussenot, M. Geist, O. Pietquin, M. Michalski, *et al.*, "What matters in on-policy reinforcement learning? a large-scale empirical study," *arXiv preprint arXiv:2006.05990*, 2020.

[22] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning," in *European Conference on Machine Learning*, pp. 282–293, 2006.

[23] R. S. Sutton, "Dyna, an integrated architecture for learning, planning, and reacting," *ACM SIGART Bulletin*, vol. 2, no. 4, pp. 160–163, 1991.

[24] J. Schrittwieser, I. Antonoglou, T. Hubert, K. Simonyan, L. Sifre, S. Schmitt, A. Guez, E. Lockhart, D. Hassabis, T. Graepel, *et al.*, "Mastering atari, go, chess and shogi by planning with a learned model," *Nature*, vol. 588, no. 7839, pp. 604–609, 2020.

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

[26] G. A. Rummery and M. Niranjan, "On-line q-learning using connectionist systems," tech. rep., Cambridge University Engineering Department, 1994.

[27] D. Pathak, P. Agrawal, A. A. Efros, and T. Darrell, "Curiosity-driven exploration by self-supervised prediction," in *International Conference on Machine Learning*, 2017.

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

[29] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[30] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[31] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *Proceedings of the 32nd International Conference on Machine Learning (ICML)*, pp. 1889–1897, 2015.

[32] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[33] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *Proceedings of the 33rd International Conference on Machine Learning (ICML)*, pp. 1928–1937, 2016.

[34] C. Browne, E. Powley, D. Whitehouse, S. Lucas, P. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of monte carlo tree search methods," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, no. 1, pp. 1–43, 2012.

[35] L. Wang, Y. Zhao, Y. Jinnai, Y. Tian, and R. Fonseca, "Alphax: exploring neural architectures with deep neural networks and monte carlo tree search," *arXiv preprint arXiv:1903.11059*, 2019.

[36] C. Gao, M. Müller, and R. Hayward, "Three-head neural network architecture for monte carlo tree search.," in *IJCAI*, pp. 3762–3768, 2018.

[37] D. J. N. J. Soemers, C. F. Sironi, T. Schuster, and M. H. M. Winands, "Enhancements for real-time monte-carlo tree search in general video

game playing," in *2016 IEEE Conference on Computational Intelligence and Games (CIG)*, pp. 1–8, IEEE, 2016.

[38] D. Perez, S. Samothrakis, J. Togelius, T. Schaul, S. M. Lucas, A. Couëtoux, J. Y. Lee, C.-W. Lim, and T. Thompson, "The 2014 general video game playing competition," *IEEE Transactions on Computational Intelligence and AI in Games*, 2016. To appear.

[39] Y. Tian, Q. Gong, W. Shang, Y. Wu, C. L. Zitnick, Y. Sun, and W. Chen, "Elf opengo: An analysis and open reimplementation of alphazero," in *International Conference on Machine Learning*, pp. 6247–6256, PMLR, 2019.

[40] G. Chaslot, M. Winands, J. van den Herik, and J. Uiterwijk, "Monte-carlo tree search: A new framework for game ai," in *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 4, pp. 185–190, 2008.

[41] C. D. Rosin, "Multi-armed bandits with episode context," in *28th International Conference on Machine Learning (ICML 2011)*, pp. 1057–1064, 2011.

[42] T. Anthony, Y. Tian, D. Barber, *et al.*, "Thinking fast and slow with deep learning and tree search," *arXiv preprint arXiv:1705.08439*, 2017.

[43] B. Stankiewicz, M. Szubert, T. Szczepanski, W. Jaśkowski, K. Krawiec, and Z. Michalewicz, "Monte-carlo tree search enhancements for othello," in *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*, pp. 270–277, IEEE, 2012.

[44] M. Seify, G. Chaslot, and O. Teytaud, "Single-player atari 2600 games with deep reinforcement learning and tree search," *arXiv preprint arXiv:2008.05041*, 2020.

[45] G. M.-B. Chaslot, M. H. Winands, J. V. D. Herik, *et al.*, "Nested monte carlo search," in *Computer Games Workshop*, vol. 8, p. 4, 2008.

[46] A. Couëtoux, T. Cazenave, and A. Saffidine, "Double progressive widening: learning a selection policy for guiding a monte carlo tree search," *arXiv preprint arXiv:1110.4650*, 2011.

[47] S. Gelly, L. Kocsis, D. Silver, and C. Szepesvári, "Uct with rave," *Handbook of Games and Economic Behavior*, vol. 3, pp. 45–96, 2011.

[48] G. Chaslot, M. H. Winands, J. V. D. Herik, J. W. Uiterwijk, and B. Bouzy, "Monte-carlo tree search: A new framework for game ai," *Proceedings of the fourth international conference on Computers and games*, pp. 216–227, 2008.

[49] M. Enzenberger, P. Kissmann, M. Müller, A. Saffidine, T. Schubert, and M. Tscherepanow, "Clustered monte-carlo tree search," in *Proceedings of the 3rd International Conference on Fun and Games*, pp. 27–38, 2010.

[50] G. Chaslot, S. Bakkes, I. Szita, and P. Spronck, "Parallel monte-carlo tree search," *New Mathematics and Natural Computation*, vol. 4, no. 3, pp. 343–357, 2008.

[51] J.-B. Grill, F. Altché, Y. Tang, T. Hubert, M. Valko, I. Antonoglou, and R. Munos, "Monte-carlo tree search as regularized policy optimization," in *International Conference on Machine Learning*, pp. 3769–3778, PMLR, 2020.

[52] P. Auer, N. Cesa-Bianchi, and P. Fischer, "Finite-time analysis of the multiarmed bandit problem," *Machine learning*, vol. 47, no. 2-3, pp. 235–256, 2002.

[53] L. V. Allis, "A knowledge-based approach of connect-four.," *J. Int. Comput. Games Assoc.*, vol. 11, no. 4, p. 165, 1988.

[54] V. Sannidhanam and M. Annamalai, "An analysis of heuristics in othello," 2015.

[55] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[56] H. Duvaud, "Muzero general: Open reimplementation of muzero."

[57] Y. Tian, J. Ma, Q. Gong, S. Sengupta, Z. Chen, J. Pinkerton, and L. Zitnick, "ELF OpenGo: an analysis and open reimplementation of AlphaZero," in *Proceedings of the 36th International Conference on Machine Learning* (K. Chaudhuri and R. Salakhutdinov, eds.), vol. 97 of *Proceedings of Machine Learning Research*, pp. 6244–6253, PMLR, 09–15 Jun 2019.

[58] T. M. Moerland, J. Broekens, A. Plaat, and C. M. Jonker, "A0c: Alpha zero in continuous action space," *arXiv preprint arXiv:1805.09613*, 2018.

[59] I. Bratko, "Alphazero–what's missing?," *Informatica*, vol. 42, no. 1, 2018.

[60] L. Xu and J. Hu, "Reinforcement learning for alphago zero," in *Proceedings of the IEEE International Conference on Data Mining Workshops*, pp. 887–892, 2018.

[61] W. Huang, Y. Li, and X. Wang, "Residual monte carlo tree search in alphazero," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, pp. 7246–7253, 2019.

[62] J. Schrittwieser, I. Antonoglou, T. Hubert, K. Simonyan, L. Sifre, S. Schmitt, and et al., "Mastering atari, go, chess and shogi by planning with a learned model," *Nature*, vol. 588, no. 7839, pp. 604–609, 2019.

[63] Y. Shen and F. Wang, "Improved self-play in alphazero," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, pp. 7254–7261, 2019.

[64] Y. Bai, C. Jin, and T. Yu, "Near-optimal reinforcement learning with self-play," in *Advances in Neural Information Processing Systems* (H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, and H. Lin, eds.), vol. 33, pp. 2159–2170, Curran Associates, Inc., 2020.

[65] N. Brown, A. Bakhtin, A. Lerer, and Q. Gong, "Combining deep reinforcement learning and search for imperfect-information games," *Advances in Neural Information Processing Systems*, vol. 33, pp. 17057–17069, 2020.

[66] T.-R. Wu, T.-H. Wei, and I.-C. Wu, "Accelerating and improving alphazero using population based training," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, pp. 1046–1053, 2020.

[67] Q. Sun, S. Zhang, Y. Qi, and C. Xu, "Adversarial self-play algorithm for reinforcement learning," *International Journal of Machine Learning and Cybernetics*, vol. 11, no. 5, pp. 1061–1072, 2020.

[68] Y. Tian, Q. Yu, H. Zhu, S. Zhang, and C. Xu, "Improving reinforcement learning with self-play and opponent's model," in *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 2754–2760, 2020.

[69] M. Świechowski, "Game ai competitions: motivation for the imitation game-playing competition," in *2020 15th Conference on Computer Science and Information Systems (FedCSIS)*, pp. 155–160, IEEE, 2020.

[70] T. Pepels, M. H. Winands, and M. Lanctot, "Real-time monte carlo tree search in ms pac-man," *IEEE Transactions on Computational Intelligence and AI in games*, vol. 6, no. 3, pp. 245–257, 2014.

[71] H. Svoren, V. Thambawita, P. Halvorsen, P. Jakobsen, E. Garcia-Ceja, F. M. Noori, H. L. Hammer, M. Lux, M. A. Riegler, and S. A. Hicks, "Toadstool: A dataset for training emotional intelligent machines playing super mario bros," in *Proceedings of the 11th ACM Multimedia Systems Conference*, pp. 309–314, 2020.

[72] E. Saleh and et al., "Should models be accurate?," *arXiv preprint arXiv:2205.10736*, 2022.

[73] O. Vikbladh, D. Shohamy, and N. Daw, "Episodic contributions to model-based reinforcement learning," in *Annual conference on cognitive computational neuroscience, CCN*, 2017.

[74] D. Hafner, T. Lillicrap, M. Norouzi, and J. Ba, "Mastering atari with discrete world models," *arXiv preprint arXiv:2010.02193*, 2020.

[75] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems (NIPS)*, 2012.

[76] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen, *Classification and Regression Trees*. CRC Press, 1984.

[77] L. Breiman, "Random forests," *Machine Learning*, 2001.

[78] Y. Freund and R. E. Schapire, "A decision-theoretic generalization of on-line learning and an application to boosting," *Journal of Computer and System Sciences*, 1997.

[79] M. L. Littman, T. L. Dean, and L. P. Kaelbling, "A bayesian framework for model-based reinforcement learning in partially observable domains," *Machine Learning*, 2002.

[80] C. Cortes and V. Vapnik, "Support-vector networks," *Machine Learning*, 1995.

[81] R. Coulom, "Monte-carlo tree search and rapid action value estimation in computer go," *Artificial Intelligence*, vol. 170, no. 11, pp. 1856–1875, 2006.

[82] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proceedings of the 31st Conference on Neural Information Processing Systems*, pp. 6000–6010, 2017.

[83] Z. Zhang, "When doctors meet with alphago: potential application of machine learning to clinical medicine," *Annals of translational medicine*, vol. 4, no. 6, 2016.

[84] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous systems." `http://download.tensorflow.org/paper/whitepaper2015.pdf`, 2015.

[85] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The arcade learning environment: An evaluation platform for general agents," *Journal of Artificial Intelligence Research*, vol. 47, pp. 253–279, 2013.