Exploring Monte Carlo Tree Search

vs

Reinforcement Learning and Heuristic Algorithms in the game of Connect-4

Student Name: Rohan Arya

Student ID: 20086377

Course: BSc in Computer

Systems Supervisor: J.J. Collins

Academic Year: 2023/2024

Department of Computer Science and Information Systems

Faculty of Science and Engineering

University of Limerick

Abstract

The last few decades have witnessed an exponentially escalating trend towards the globalization of the technological world, specifically in the industry of software and computer science. (Casey, 2008) Artificial Intelligence specifically is one such field that has amassed lots of attention, time, and resources being devoted to it only in the last couple of years and has resulted in groundbreaking technologies and scientific breakthroughs which have only a promising future. One such area that AI has had a significant impact on is the use of AI in games. Modern Computer games are a result of constant evolution and improvement through the implementation of different Artificial Intelligence techniques. Some of the Artificial Intelligence algorithms and techniques that have been developed and implemented employ the use of Neural Networks, Decision Trees, and Genetic Algorithms that have given us great results, the successes of which we will discuss throughout this paper. In this paper, I will be discussing the application of Artificial Intelligence and Machine Learning techniques in board games, specifically Reinforcement Learning and Monte Carlo Tree Search in the game Connect-4.

Table of Contents

Table of Figures

Acknowledgement

I would like to thank my supervisor, Mr. J.J. Collins, for constantly supporting me, guiding me and being my pillar of strength throughout the journey of the Final Year Project. Without him, I would not have the learned important skills to deliver a project of this quality and magnitude. Mr Collins' encouragement and academic rigour have been pivotal and inspired me to overcome challenges, enabling me to achieve my goals,

I would also like to thank my parents who have always supported me and encouraged me in achieving my goals and allowing me to reach my full potential. Without them, I would not have gotten the opportunity of being a student at the University of Limerick, for which I will forever be grateful.

I would also like to thank Google's TRC program, which gave me free access to their Cloud TPU servers and allowed me to train my AI models and see promising results.

1 Introduction

Overview

Connect 4 is a game that gives us multiple good reasons to build, test and learn Artificial Intelligence and Machine Learning algorithms. We can harness these techniques by making different models and simulations capable of learning the game and improving gradually by implementing Artificial Intelligence and Machine Learning methodologies such as Reinforcement Learning while exploring different heuristic algorithms. The rules of Connect 4 are easy to understand and simple to follow along and play with, however undeterred by its simplicity, the game offers a vast and complex environment given various possibilities that can stem from within a single situation. When we look at it from a computational perspective, we see that Connect 4 is a great choice as it has a state space of 10^12 which makes it a tractable project, as compared to games like Chess and Go (verdantfox.com, n.d.). This makes it easier to make observations, track progress, and evaluate results. The branching factor of the game when explored gives us an average of 7 moves in any possible situation during the games that result in a practicable game tree size that allows an agent to efficiently explore the moves, thus maintaining a balance between complexity and computational feasibility. We will explore in greater depth the effectiveness of Reinforcement Learning agents and then look into the Monte Carlo Tree Search algorithm, and evaluate its effectiveness (www.cs.cornell.edu, n.d.).

One of the most famous decision-making and AI paradigms is Reinforcement learning. “Reinforcement learning is learning what to do—how to map situations to actions—to maximize a numerical reward signal.”(Sutton and Barto, 2018). Reinforcement Learning is a machine learning technique where the agent learns to make decisions by trial and error to maximize the cumulative reward. The goal of reinforcement learning (Sutton and Barto 1998) is to learn good policies for sequential decision problems, by optimizing a cumulative future reward signal. In the Reinforcement Learning algorithm, the state function represents the current state of the environment through which the agent interacts with, and makes decisions and takes actions accordingly. The state value function is defined as the cumulative reward that the agent achieves through multiple iterations, as it receives the rewards by following a policy. There are different types of Reinforcement Learning algorithms, techniques and methodologies, each having its applications. Let’s dive into these one by one.
Temporal Difference is another Reinforcement Learning algorithm that is model-free and performs bootstrapping. This allows for efficient sample-based learning as well as bootstrapping, without the need for a model, which makes it a promising algorithm for Reinforcement Learning Applications.

Q-learning or Q-Tabular Learning, is one of the most important main machine learning algorithms in Reinforcement Learning where the algorithm is developed in such a way that enables the agent to make decisions in a certain environment to achieve a specific goal. It achieves this by iteratively learning and improving over time by choosing the correct option (Enterprise AI, n.d.). What is specific to Q-Learning is that it is a model-free, value-based, and off-policy reinforcement learning algorithm that finds the best series of actions based on the agents' current states. (Al Awan, n.d.) DQN is another Reinforcement Learning algorithm that combines Q Learning with deep neural networks enabling it to solve problems in environments with high dimensional state space. It does this by getting the Q-values associated with the state action pairs and implementing them in a neural network. (Van Hasselt, Guez and Silver, 2016)

However, the main focus of this report and research project will be the Monte Carlo Tree Search algorithm. What is Monte Carlo Tree Search? "MCTS is a method for finding optimal decisions in a given domain by taking random samples in the search space. MCTS is based on decision and game theory (Russell and Norvig, 2020), and on Monte Carlo and bandit-based method, where sequential decision problems are modelled as a kind of search problems." (Horcas et al., 2023). There are multiple reasons why Monte Carlo Tree Search is a great AI decision-making technique to study and research its implementation in games. In the last couple of years, it has become increasingly prominent and has increased the curiosity of MCTS to be implemented in all different kinds of games with decision-making and strategy. This is a consequence of the huge success of the AlphaGo and AlphaZero engines developed at the Deepmind that resulted in the strongest computer engine in the game of Go and Chess, respectively, which was possible due to the implementation of MCTS. Specifically, it is the Upper Confidence Bound Trees or the UCT algorithm which is the foundation of all Monte Carlo Tree Search models that have been significantly rewarding. The core idea behind the development of the MCTS models which has proven to be successful, was an idea to come up with an alternative decision-making algorithm that challenged the existing AI techniques, like Alpha-Beta pruning, minimax search and so on. The reason for this was to develop models that could explore games with vast search spaces and high branching factors that the typical minimax search and alpha-beta pruning are inefficient at due to their exhaustive and deterministic nature. On the other hand, MCTS is great at handling games like these, as it doesn't require one to learn a particular set of rules or heuristics. There are other factors as well that make MCTS models superior to other AI models, due to their ability to give good results in the minimum computational time possible while also incorporating parallelization that allows the distribution of executions of the branches across multiple processors and threads. (Browne et al., 2012)

Research Objectives

The objective of this study as part of my Final Year Project (FYP) is to delve into the study of Reinforcement Learning (RL)(Sivamayil et al., 2023). Hence, the primary aim of this year-end project is to construct models of Monte Carlo Tree Search (MCTS), while also developing Minimax and Q-Learning agents that will be used as benchmarks to evaluate the MCTS algorithms' performance. I will also develop a Neural Network model and integrate that with the MCTS model to evaluate its performance based on improvement over time.

Since, the Monte Carlo Tree Search (MCTS) is a method that is most suitable for decision-based problems, wherein random sampling results in building a tree search based on its education of the stored results from the sampling. (Świechowski et all, 2023). Knowing this, the evaluation of the outcomes of this study will include assessing the performance of each model as both are employed in learning to play the game of Connect 4. This evaluation will include comparing them independently against key performance metrics, some of which will include speed of learning, efficiency in decision making, and the victory rate, to conclude progress alongside gauging the strengths and weaknesses.

I also aim to study each model's strategy as they employ their gameplays and analyze how they adapt, if at all, to multiple scenarios and change each of their decision-making processes based on the change in the environment. Moreover, an additional aspect of the evaluation will include an attempt to determine if one model performs better than the other, both in terms of where they rank in their independent performance evaluation against key metrics and in terms of playing the game against one another. This will highlight the differences in their performance and hopefully shed light on whether one model proves to be better at the game! Through all this, I hope to contribute to the discussion of future improvements and research in the field of Reinforcement Learning algorithms.

The key research questions will be, How does the MCTS algorithm perform against minimax, Q-Learning and whether Neural Network Evaluation along with MCTS contributes to any improvement or not.

Motivation

The primary reason why I decided to dedicate my time and effort to this project is the massive personal interest and amusement that I have for the games of Artificial Intelligence (AI) and the puzzle of chess. I have spent a lot of time and energy on Chess, in addition to observing the changes in AI technology and breakthroughs in the field.

I have followed with great interest the evolution of the strategies employed and the advance on technology from the first milestone in 1997, when the computer named Deep Blue defeated Gary Kasparov using only brute force computation power, to the more recent milestone which saw GM AlphaZero, a computer that autonomously learned and refined its capabilities using neural networks and Monte Carlo search tree technique, played Chess effectively (Jed, 2024)

In my background, the deep level of fascination for chess, which is a mind-challenging game, echoes my predilection for problem-solving hobbies, such as the Rubik's Cube, which are made exceptional by memory factor and pattern recognition in the approach I use that resembles pursuits like Sudoku. My mathematics and physics knowledge also works for me to study Chess as a field that I can see further (Built In, n.d.).

To start with, the issue that draws the utmost interest is the appalling complexity, what with the infinite number of possible moves that make up any potential situation in the game, which exceeds 10^120 per move, making the challenge of developing AI and machine learning models that can achieve Chess proficiency a lot harder. The complexity of the topic that I am studying drives not only my immense passion but also the fuel that is going to propel me in the process of writing my thesis (Chess.com, n.d.).

Through this project, I want to combine my love for chess with research which will help to bridge the gap between the theoretical AI concepts and the practical applications in the strategic decision-making area. Beneath the Connect-4 game, I find a way to exploit AI and ML to solve the game of the world where the strategies are complex and put us under considerable pressure, resulting in a better knowledge of intelligent behaviour in dynamic environments. The kind of research I am engaged in is a personal pursuit of wisdom and at the same time the contribution to the ongoing dialogue about the use of artificial intelligence and different range of technology innovations.

2 Literature Review

Artificial Intelligence and Machine Learning

2.1.1 Introduction:

Artificial Intelligence has always been promising in its ability to change human life forever. But in the last 10 years, it has delivered on that promise. The last 10 years have seen progress at such an unprecedented pace that some AI models are even said to mimic human intelligence, and logic and even replicate emotions. This progress can be largely credited to the availability of data and the increase in computational power for model training.

Machine Learning is a subset of AI which focuses on the machine's ability to recognize patterns and learn from data without explicitly programming it. AI and ML have revolutionized industries through advancements in natural language processing, computer vision, predictive analysis and other domains. A major focus is on scaling already existing algorithms and finding new innovative algorithms to solve current problems even better.

2.1.2 Historical Evolution of Artificial Intelligence:

Mankind has been fascinated by the idea of making inanimate objects into intelligent beings for a long time. Ideas range from myths about robots to automatons. But Artificial Intelligence as an official term did not exist before the year 1956. In a conference held in Hanover, New Hampshire, at Dartmouth College attended by Cognitive scientist Marvin Minsky at MIT and other scientists the term "Artificial Intelligence" was officially coined. All of them are very optimistic about the future of artificial intelligence. Minsk also stated that the problem of artificial intelligence will be solved at a significant scale in his book "AI: The Tumultuous Search for Artificial Intelligence".

Before even this term was coined, the British mathematician Alan Turing did pathbreaking work which proved to be the basis for early AI. Turing realised that a machine is capable of solving any problem as long as it can be represented and solved by an algorithm. Turing's Electromechanical machine managed to crack the code used by the German submarines. It is considered to be the precursor of modern computers. The first practical digital computers were built only a few decades later.

2.1.3 Foundations of Machine Learning:

Machine Learning means to make predictions about unseen data using features learned from previous data experiences. This data can be explicitly labelled or even unlabeled data obtained via interactions with the environment. In all these cases the accuracy of the predictions is hugely based upon the quantity and quality of the data the prediction model is trained on. The measure of the quality of ML algorithms is done using sample complexity along with space and time complexities.

Types of Machine Learning Algorithms:

1. *Supervised Learning:* Learning by comparing computed output and expected output finding the error and further decreasing the error to achieve the expected output.
2. *Unsupervised Learning*: Learning on its own based upon the input patterns by discovering and adopting.
3. *Reinforcement Learning:* Based on how an agent should take actions in an environment to maximise the long-term reward.
4. *Recommender Systems:* A learning technique which gives the user a personalised ranking of items based on the user's tastes.

2.1.4 Applications of AI and ML:

ML applications are all around us in our daily use without even realising them ranging from Movie recommendation systems to weather predictors. We can classify all Machine Learning application problems into the following classes to be explicit:

Classification: To assign a class to an item out of many classes. Eg. Text classification, Sentiment Analysis and Speech Recognition.

Regression: To predict a value from a range for each item. Eg. Stock value prediction and weather prediction.

Ranking: To order items based upon some criterion. Eg. Ranking web pages upon a search query, Recommendation systems.

Clustering: To partition items into homogeneous regions. Eg. Identifying communities in social media networks and document clustering.

Dimensionality Reduction/Manifold Learning: To transform a representation of items into a lower dimension representation. Eg. Preprocessing images in computer vision tasks, PCA.

2.1.5 Recent Advancements and Trends:

Recent advancements in Deep Learning have taken the world by storm. This includes Large Language Models such as Generative Pretrained Transformers(GPT) used in ChatGPT and BARD. Recent research has also combined LLMs with Computer Vision for several applications. Transfer Learning also allows developers to build upon these already trained large-scale models for very specific applications such as Programming.

Computer Vision(CNN, RNN, LSTM) and Deep Learning also form the basis for Self-driving cars along with route management and navigation. Computer Vision also enables facial recognition for identification purposes and even mask detection during the recent COVID-19 pandemic.

New AI advancements also can create Deep fakes using Generative AI of people(GANs). This has also been used by Meta recently in their new Virtual Reality projects.

This pace of development in the field of AI has been unprecedented, therefore it also becomes necessary to regulate them to ensure their ethical use and reduce bad actors and their implications.

AI Winters

“AI winters were not due to imagination traps, but due to lack of imagination. Imaginations bring order out of chaos. Deep learning with deep imagination is the road map to AI springs and AI autumns.” (Amit Ray, 2019)

AI Winter is a term that refers to a period in which funding and research in the field of artificial intelligence stagnate. The first AI winter took place in the 1970s and was caused as many of the claims of AI were not met at that time and a lot of people who were not educated in the field of AI marched into it having unrealistic expectations of a cut in funding was bound to happen, Although AI looks promising right now back then it was just a concept that was too good to be true. In the mid 1970's a prominent mathematician. James Lighthill was against AI and believed that AI had no future, this led to its bad public image and various governments stopped funding AI projects during this period growth in the field of AI was little to none, After 7 years AI started its ascend but a lack of trust and knowledge of investors led to another AI winter during the '80s. (Schuchmann, 2019)

A breakthrough happened when the world champion Gary Kasparov who is still considered to be one of the best players to have ever played the game of chess faced off against Deep Blue which was a Supercomputer created by IBM. (ResearchGate, n.d.) It was the first time humanity was witnessing a battle of man and a machine; the match was a 9-day event and was globally publicized. (Yao, 2022) Deep Blue was capable of calculating and exploring up to 200 million chess positions and even Gary

Kasparov a chess mastermind was not able to compete against the technical brilliance of Deep Blue and resigned in Game 6. The people around the world were impressed by this accomplishment by a machine as Chess is generally regarded as the game of logic and intelligence and that win not only cemented IBM's position in the tech world but also was a statement the AI would flourish shortly. (Goodrich, 2021)

Another AI program known as Alpha Go which uses advanced neural networks can be regarded as an advancement in AI. Alpha Go went up against the world champion Lee Sedol in the game "GO" which is played by over 40 million people worldwide and is very popular in China. (Cade Metz, 2016) Lee lost four out of the five rounds that were played and after defeating one of the best players of the most complicated game ever devised by humans AI proved that it can achieve anything when used at its potential. In that game Alpha Go proved that it can simulate the understanding of a human. (Demis Hassabis, 2016)

Now various AI programs such as ChatGPT, Google Bard, and Mid journey set an exemplary example in front of other research institutions that their research in the field of AI will lead to the advancement of society as a whole, and due to this the probability, that another AI winter could occur is very less soon.

Reinforcement Learning

“Reinforcement learning is learning what to do—how to map situations to actions—to maximize a numerical reward signal.” (Sutton and Barto, 2018). Reinforcement Learning revolves around the concept that the learning agent is not explicitly instructed to take specific actions but instead is programmed to discover new actions and consequently the actions that give a rewarding result. When these set of actions are followed by the agent repeatedly, the action can not only determine the immediate reward identified by exploring said action but also exploring further actions to determine further awards. These two key methodologies of discovering new actions on the basis of trial and error and of delaying rewards throughout subsequent actions are core characteristics of Reinforcement Learning. (Sutton and Barto, 2018)

2.1.6 Elements of Reinforcement Learning

When we look at Artificial Intelligence algorithms, primarily our focus is on two things. Firstly, the environment, that will be explored and possible actions that can take place within that environment yielding a certain award, and secondly the agent, that will explore the environment, discover possible actions, make decisions based on

rewards and subsequently learn and improve. Reinforcement Learning, at its heart, has 4 main sub-elements that together comprise the Reinforcement Learning system. These are the *policy*, the *reward signal*, the *value function* and optionally the *model*. (Sutton and Barto, 2018)

1. Policy: The policy of the agent is the strategy that it's going to employ to make specific decisions in order to maximize its reward. This means that the policy affects the agent's behaviour with the environment. This behaviour can be different depending on the type of strategy being employed by the agent, and the way it approaches a problem.
2. Reward Signal: The Reward Signal guides the Reinforcement Learning agent to achieve its goal. The immediate feedback that the agent receives after an action helps the agent to reach its goal. The feedback that the agent receives after one step is called the reward. The agent's goal is to maximize the long-term reward.
3. Value Function: The Value Function is simply the long,-term award that the agent accumulates over time. Once the rewards are accumulated, the value function obtained gives us an idea of the states it will explore and the rewards that will be available in those states.
4. Model: The model of the agent mimics the behaviour of the agent. The model helps predict the following states of the actions of the agent, thus giving us an inference on how the environment will behave. The main purpose of having a model is to have a plan about what course of action to take for future decisions. In Reinforcement

Learning, models are optional. Some Reinforcement Learning algorithms such as Bootstrapping have a model, whereas some algorithms like Monte Carlo methods are model-free.

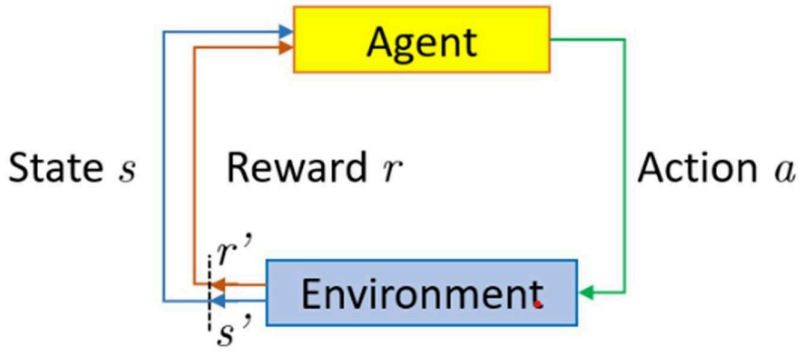

Figure 1 Reinforcement Learning adapted from Richard Sutton and Andrew Barto Reinforcement Learning, Second Edition. The MIT Press. 2018

2.1.7 Challenges in Reinforcement Learning

Reinforcement Learning is a decision-making and AI paradigm that faces significant challenges in training the agent so that it learns the optimal behaviour through its interactions with the environment. It is extremely important to address these challenges to maximize the learning of an agent.

2.1.7.1 Exploration vs Exploitation

It is important to maintain a balance in the trade-off between exploration and exploitation. The reinforcement learning agent tends to favour actions that it has previously taken and found to be successful in gaining rewards if it wants to continue receiving them. However, to find such actions, it must attempt activities that it has never chosen before. To get the reward, the agent must take advantage of its past experiences, but it must also explore to choose better actions going forward. Finding the right balance is critical as too much exploration might delay the discovery of the optimal policy, and too much exploitation might result in sub-optimal choices. (Sutton and Barto, 2018)

For example, in a game of chess, the agent would have to explore new moves so as it to improve its strategy but it would also have to exploit know good moves to increase its chances of winning.

2.1.7.2 Credit Assignment Problem

This challenge in Reinforcement Learning arises while assigning the correct amount of credit to the actions that contribute towards a long-term award. It is important to determine the right set of actions that lead to a particular outcome in Reinforcement Learning. Which is the agent is required to credit the positive outcome and learn from it regardless if the reward is delayed or sparse. (Sutton and Barto, 2018)

For example, in a game of chess, when the agent wins a game, the agent needs to learn which sequence of moves resulted in the victory, and thus assign it the credit accordingly to improve its learning.

2.1.8 Q-Value Function(Q)

“Q-learning is a model-free, value-based, off-policy algorithm that is used to find the optimal policy for an agent in a given environment.”(Trung Luu, 2023)The idea behind this algorithm is to determine the best series of actions based on the current state of the agent. The Q in Q- Learning stands for quality. This means that the key underlying concept in Q-Learning is identifying and maximizing the value of a reward. (Trung Luu, 2023)

$$Q(s, a) \leftarrow Q(s, a) + \alpha \cdot [\, r + \gamma \cdot \max_{a'} Q(s', a') - Q(s, a)]$$

Where,

- $Q(s,a)$ is the Q-value of the current state-action pair
- r is the immediate reward received for taking action a in state s
- α is the learning rate,
- γ is the discount factor (this is a value between 0 and 1) which determines the significance of future rewards
- s' is the future state following the current state, and
- a' is the action that maximizes the Q-value in the future state s' following the current state s (Sutton and Barto)

Q-Learning contributes to some of the prominent breakthroughs in Reinforcement Learning, which is an off-policy Temporal Difference algorithm (Watkins, 1989).

One of the key features of Q-learning is that it doesn’t require explicit knowledge of the transitions and reward functions. Through the process of trial and error, which is the foundational characteristic of reinforcement learning, Q-learning estimates the optimal policy, by assessing the rewards it receives from the environment. This in turn, helps guide its decisions that leads to the refining of the policy. It continues to repeat this process until it is able to maximize the cumulative reward, thus giving us the optimal policy. As we saw earlier, the value function is a significant element in reinforcement learning, as it determines the
long-term reward. In Q-Learning, the agent is trained in such a way that it chooses the more valuable actions in each state thus resulting in an optimal policy. This way it is able to update the Q-value for the given state-action pair. The ε-greedy policy is used by Q-Learning where the highest Q-value is selected by the agent. The probability of the Q-value selected is 1- ε, whereas the probability of the action it took is ε. (Trung Luu, 2023)

An interesting aspect about Q-Learning being an off-policy algorithm is that the learning of the optimal policy is facilitated by using a sub-optimal policy. As a consequence of which it is able to maximize the value of the reward for all the steps that follow. These values are stored in a table called the Q-Table acts as its value

function. At the start, the values set in the table are arbitrary or zeros. Once the learning process begins, the agent interacts with the environment and then it selects the actions by employing the exploration-vs-exploitation strategy usually is ε-greedy. This enables it to identify the rewards and see what the possible future states are. The Q-values in the Q-table are updated by using the formula above.

Algorithm 1: Q-learning Algorithm(off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$;
Initialize $Q(s, a)$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$ arbitrarily, except $Q(\text{terminal}, \cdot) = 0$;
foreach *episode* **do**
  Initialize S;
  foreach *step of episode* **do**
    Choose A from S using policy derived from Q (e.g., $\varepsilon - greedy$);
    Take action A, observe R, S';
    $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$;
    $S \leftarrow S'$;
  end
  until S is terminal
end

Figure 2 Q-Learning Algorithm adapted from Richard Sutton and Andrew Barto Reinforcement Learning, Second Edition. The MIT Press. 2018

Deep Q Network

"DQN is a neural network that takes in information about the state of the environment and outputs the estimate of the action-value function for each possible action" (Kristopher De Asis et al., 2017). Deep Q Learning was introduced to solve the problems that Q Learning was facing due to its inability to cope with the large state spaces being produced, which was a result of the increasing size of the Q-table that grows at an exponential rate. This puts a lot of computational strain on the algorithm by demanding extremely high memory to store the produced Q-Values. This would

also imply that the time to explore each state and identify the values would be extremely irrational from the developer's perspective to see any significant results.

This is why an alternative approach was implemented where Q Learning was combined with Deep Neural Networks to give us Deep Q Learning. DQN which stands for Deep Q Networks was an algorithm that was introduced by DeepMind in 2013. The main purpose of this algorithm was to play the games in the Atari environment from raw pixels.

In the game of Connect-4, a DQN is constructed by considering the board state as the input state and identifying the Q-value for every possible move in the state. When it explores and interacts with the environment of the Connect-4 game, it can develop a policy and run multiple iterations to maximize the cumulative reward and increase its chances of winning.

A key aspect of DQN is that relies on experience from replay. It does this by storing the past experiences to improve the learning over time. It also employs a target network that allows it to stabilize the training by adjusting the Q-value targets, ensuring a smooth learning process.

It has certain limitations as well that lead to hindrances in the learning process. Factors such as maximization biases and catastrophic forgetting negatively impact the training of the DQN agent, which we will look into detail in the next section. The solution to these problems is to employ algorithms such as Monte Carlo Tree Search.

An Introduction to Monte Carlo Tree Search

"Monte Carlo Tree Search (MCTS) is a method for finding optimal decisions in a given domain by taking random samples in the decision space and building a search tree according to the results. It has already had a profound impact on Artificial Intelligence (AI) approaches for domains that can be represented as trees of sequential decisions, particularly games and planning problems." (Browne et al., 2012)

Monte Carlo Tree Search has proven to be an effective decision-making paradigm, and it has been recognized, implemented, and improved exponentially over the last few years. The reason for its massive success and popularity lies in its nature which enables it to solve problems in multiple different domains using sequential decision-making in an uncertain environment. The different domains that fall in this category are game playing such as Chess, Go, Connect-4, and more, in the field of robotics, and other fields that include planning and solving optimization problems.

So how it is that it can give such positive results? Firstly, we must learn how exactly the algorithm works. MCTS consists of searching combinatorial space trees represented by trees.

About games, these trees are referred to as the game trees that represent the possibilities in a certain position. These trees consist of nodes that represent the state or the configurations of the problem while the edges represent the transition from one state to another (Springer).

Once the decision tree is laid out incrementally and asymmetrically, the process is simple yet powerful. A tree policy is used that determines the most critical nodes in the current tree, and multiple iterations are run through a process of exploration and exploitation. Then, a simulation is generated from the node that is selected, and the values of the nodes are updated, thus updating the search tree based on the result. (Browne et al., 2012)

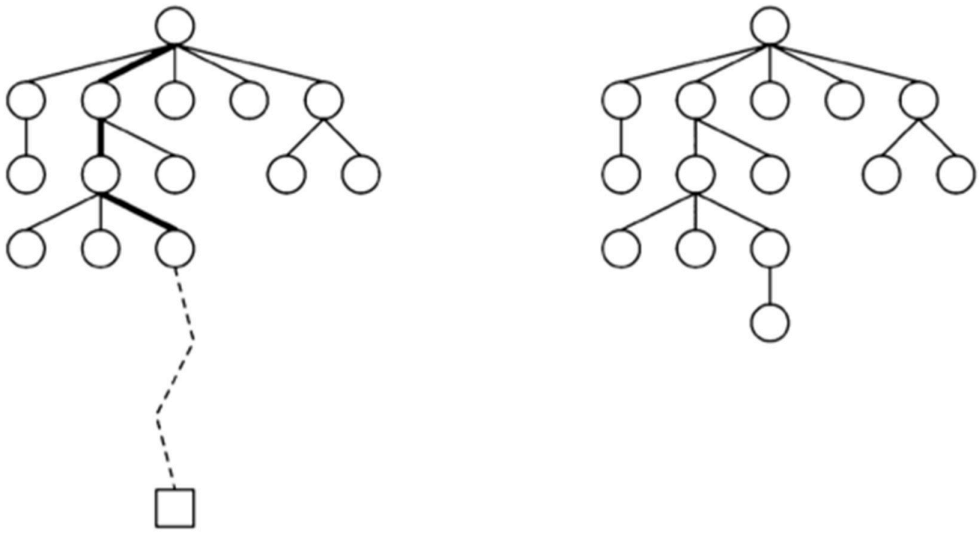

Figure 3. The basic Monte-Carlo Tree Search process adapted from Cameron Browne, Member, IEEE, Edward Powley, Member, IEEE, Daniel Whitehouse, Member, IEEE, Simon Lucas, Senior Member, IEEE, Peter I. Cowling, Member, IEEE, Philipp Rohlfshagen, Stephen Tavener, D.

Mechanics

Now that we know what the process of the MCTS simulation is, let's dive deeper into how this process happens. For that, we would need to understand the mechanics involved in developing the model of the MCTS agent. First, it consists of the process where the decision search tree is iteratively developed. "The idea is to develop the tree that consists of **n** number of nodes, in which, each node is annotated by the win count and the visit count." (Agarwal, 2020) So in the first step, the tree starts from a single root node and continues to run iterations, given the rate of exhaustion of the resources.

So, the initial setup includes a single node, which is the parent node, and a large random value is assigned, to the non-visited child nodes. This value is called the Upper Confidence Bound value or the UCB value. The UCB value generated from the formula helps us to select the child node which brings us to our first step in the MCTS simulation:

1. Selection: This phase involves navigating from the parent node to the child nodes, by selecting the most urgent nodes at a given state. It makes the selection by choosing the node with the highest UCB value, and it continues to select nodes based on this approach until the terminal state is reached. The UCB value helps to balance the exploration of the nodes that are less visited, by enabling the algorithm to explore new paths, while balancing the exploitation of the nodes that perform well, by consistently choosing these nodes to maximize chances of winning. This deals with one of the key issues in MCTS of exploration vs exploitation, which we will expand in detail in the next section. (Agarwal, 2020)

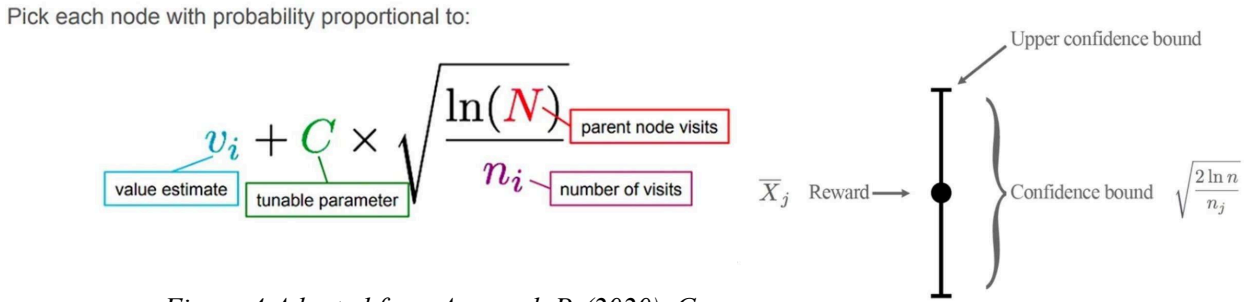

Figure 4 Adapted from Agarwal, P. (2020). Game AI: Learning to play Connect 4 using Monte Carlo Tree Search. [online] Medium. Available at: https://pranav-agarwal-2109.medium.com/game-ai-learning-to-play-connect-4-using-monte-carlo-tree-search-f083d7da451e.

2. Expansion: The tree is further expanded by adding a new child node corresponding to the possible actions in the current state, to an existing node, when the UCB value can no longer be applied to any of the existing nodes. The new nodes represent the possible future states, that will be explored and then newer nodes will be selected accordingly.

3. Simulation (Rollout): After the expansion phase, the new child node is selected and the simulation is run with the default policy to reach an outcome in the game. The aim is to achieve potential winning states and associate rewards accordingly with the states.

4. Backpropagation: When the results from the simulations are obtained with a potential winner, they are backpropagated up the tree, updating the nodes with the new win score. The purpose of this step is to refine the estimations of the nodes' values and guide future selections.

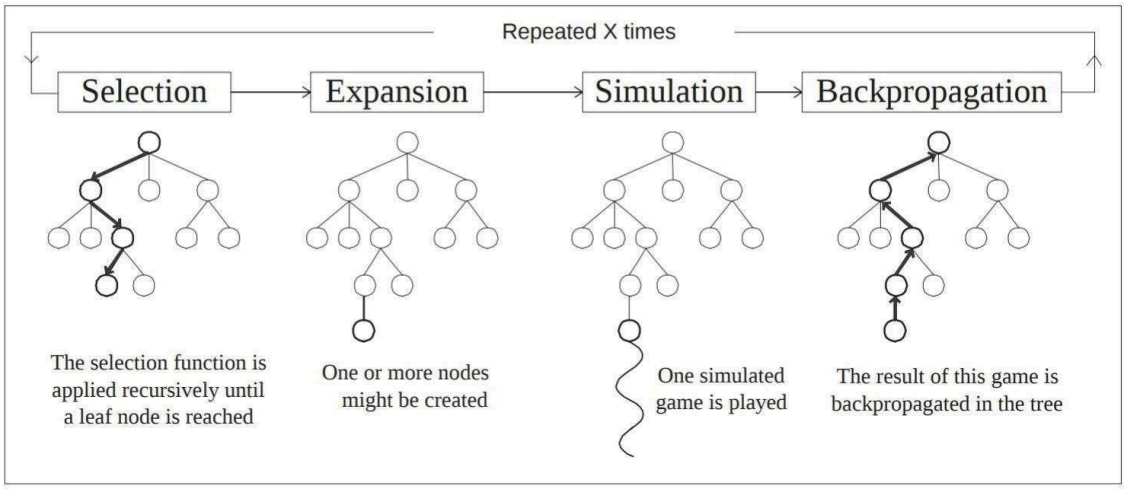

Figure 5. Monte-Carlo Tree Search simulation adapted from General Game-Playing With Monte Carlo Tree Search | by Michael Liu | Medium

UCT and Exploration vs. Exploitation

One of the key challenges in the basis of selection of the child nodes is of balancing exploration vs exploitation. It is highly crucial to maintain this balance between exploring new moves and exploiting the rewards from the existing moves as it plays an

important role in the effectiveness of the algorithm in finding new paths and choosing the right moves. If there is an imbalance, with the algorithm prioritizing the exploration, it may delay the result in delay of finding the optimal policy, on the other hand, prioritizing exploitation may result in sub-optimal choices.

The formula of the UCT which stands for Upper Confidence Bound Trees is:

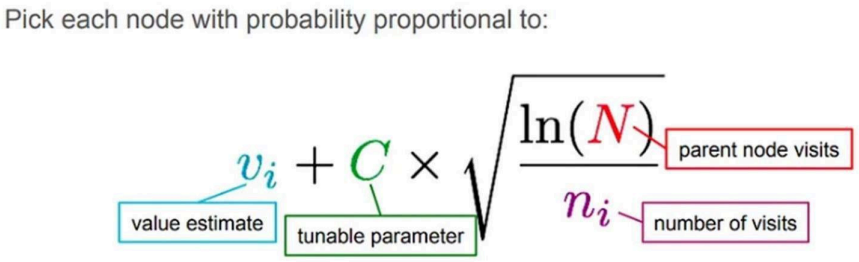

Figure 6 Adapted from Agarwal, P. (2020). Game AI: Learning to play Connect 4 using Monte Carlo Tree Search. [online] Medium. Available at: https://pranav-agarwal-2109.medium.com/game-ai-learning-to-play-connect-4-using-monte-carlo-tree-search-f083d7da451e.

Where: -

- ***Vi*** is the value estimate
- ***C*** is the tunable bias parameter
- ***N is*** the total number of times the parent node is visited
- ***Ni is*** the total number of times the child node is visited

This plays a crucial role in balancing of exploration of new states, and the exploitation of existing states to maximize rewards. The reward estimates are assigned by generating random simulations from the child nodes until an outcome is reached, and once the values are backpropagated updating the ancestor nodes, this process is repeated until the estimate values become reliable. At the very beginning, the MCTS agent will not give highly reliable value estimates but as the amount of time to run multiple iterations increases, the estimate value start becoming more reliable. (www.cs.swarthmore.edu, n.d.)

In MCTS, balancing exploration vs exploitation play a critical role that determine its success across various games and domains. This is because it allows the agent to successfully explore paths with complex state spaces and exploiting the right moves to

increase its chances of winning, that leads to enhanced performance and decision making, especially in games such as Chess, Go and Connect-4.

Game-playing applications in MCTS

MCTS (Monte Carlo Tree Search) is a powerful algorithm that consists of combinatorial spaces depicted by trees used for decision-making. In said trees, nodes show states (problem configuration) and edges donate the transitions from one state to another. The method uses intelligent tree search that balances exploration and exploitation by performing randomised sampling by running simulations and storing the statistics of actions to make educated guesses while choosing in every subsequent iteration. [Świechowski et al., 2022]

MCTS has gone through multiple enhancements and development of numerous variants after its particular success in General Game Playing (GGP) and General Video Game Playing (GVGP). A variant of MCTS, called Self-Adaptive MCTS (SA-MCTS) is used to optimise online parameters of a standard non-Self- Adaptive MCTS agent of GVGAI. The three agents of SA-MCTS use Naïve Monte-Carlo, an N-Tuple Bandit Evolutionary Algorithm and an Evolutionary algorithm to select parameter values and test on 20 single-player games of GVGAI. [Sironi et al., 1970]. We use a popular approach towards GVGP by using the framework from GVGAI, in which the decision-making time allowed is only 40 ms.

In multiplayer games, one challenge that we face is opponent modelling, there is no sure-shot way of knowing what your opponent will do. By basing our assumptions on game theory, game mechanics and psychology, we can create an MCTS algorithm in the context of these games.[Sironi et al., 1970]

For games such as Go, where it is difficult to formulate a state value function, recent MCTS algorithms successfully combine Upper Confidence Bounds with MC simulations to get refined estimates, by using domain-specific knowledge, derandomizing the MC simulations can give significant performance improvements.The goal of this is to determine whether such a simplistic approach can be used for performance improvement of MCTS [Robles et al., 2019]

The SA-MCTS agents achieve more robust results on the tested games. With the same time setting, they perform similarly to the baseline standard MCTS agent in the games for which the baseline agent performs well and significantly improve the win rate in the games for which the baseline agent performs poorly. We also test the performance of the non-self-adaptive MCTS instances that use the most sampled parameter settings during the online tuning of each of the three SA- MCTS agents for each game. Results show that these parameter settings significantly improve the win rate on the games

(Sironi et al., 1970).

How are Reinforcement Learning and Monte Carlo Tree Search related to each other?

It is imperative to look into the connection between Reinforcement Learning and Monte Carlo Tree Search and why they are strongly associated with each other. In order to study this connection in depth, we would have to look at the various factors in both algorithms that determine the similarities between them, how their mechanics affect their performance, and the rudimentary differences, that further define which algorithm is better and why.

Similarities in Learning, Decision Making, and Searching

One of the key similarities in the learning and planning methodologies between the two algorithms lies in their similar problem-solving approaches. Both these paradigms estimate the same value functions and both update these estimates incrementally. However, the main difference lies in the origin of the experience. The learning phase relies on the real interactions with the environment, whereas, the planning phase relies on the simulated environment. The inference we gather from this is that when any learning method is implemented in a simulated environment, it can be adapted for learning.

So, since MCTS is widely considered as a search and planning technique, similarities can be drawn between the search mechanics of MCTS and the sample-based methods in Reinforcement Learning, and all-in-all the framework of MCTS is aligned with Reinforcement Learning applied to planning. (Vodopivec, Samothrakis and Ster, 2017)

Despite these similarities between the two, there are varying opinions by researchers that imply that the similarities between the two might not be so evident at an earlier on stage, however, the relationship between them boils down to perspectives. Essentially, they both share similar concepts but with different perspectives and interpretations.

Often, we have these sets of conflicting opinions due to the slightly different purpose and nature of both these algorithms. MCTS was designed and developed to give us a new solution to tackle games with high branching factors, such as Chess and Go. This meant that it was designed for games that are episodic, but the knowledge is not transferred between the episodes. “On the other hand, the RL community focuses mostly on universally-applicable solutions for a wider class of problems and its philosophical foundations discourage using (too much) expert knowledge in unprincipled ways (although this can prove very effective in practice)” (Vodopivec, Samothrakis and Ster, 2017)

Though there are several other factors as well that in some cases prove why MCTS

would be the better algorithm of choice over Reinforcement Learning, such as maximization bias and catastrophic forgetting. We will understand how these factors affect the performance of these algorithms in the game of Connect-4 in the following section.

MCTS and Reinforcement Learning in Connect 4

As we discussed earlier, the game Connect 4 is an excellent choice if not ideal to study, research, and develop a learning model. Various factors make the game a desirable choice for an AI model research project, including it being easy to understand and simple to follow along and play with, however undeterred by its simplicity, the game offers a vast and complex environment given various possibilities that can stem from within a single situation. When we look at it from a computational perspective, we see that Connect 4 is a great choice as it only has a state space of 10^12 which makes it a tractable project, thus making it easier to make observations, track progress, and evaluate results. The branching factor of the game when explored gives us an average of 7 moves in any possible situation during the games that result in a practicable game tree size that enables the MCTS model to efficiently explore the moves, thus maintaining a balance between complexity and computational feasibility.

Connect 4 is a game that consists of 7 x 6 grid space. The objective of the game is to insert slots on alternative moves against a rational player until we arrive to a situation where either player has 4 slots matched in a row. The slots could be matched vertically, horizontally or diagonally and the player who achieves this first wins the game. Many different AI techniques can be used to build a model capable of learning and playing the game of Connect 4, such as Minimax search, Alpha Beta pruning, Q learning, proximal policy algorithm, Monte Carlo Tree Search, and many more. In this section, we will discuss why MCTS and Reinforcement Learning offer several more advantages giving us a convincing reason to choose them as our paradigms of choice.

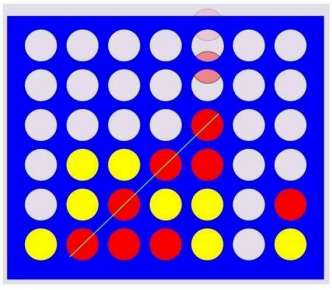

Figure 7 Diagonal Win State

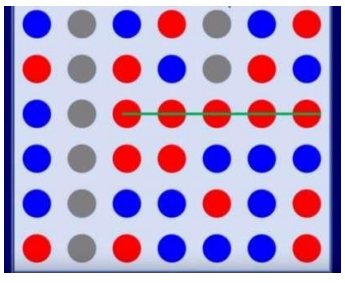

Figure 8 Horizontal Win State

2.1.9 Why MCTS and RL over other AI techniques?

One of the most important reasons why MCTS is the optimal choice is because of its flexibility in solving games with high branching factors (Agarwal, 2020)In a study conducted by Clune in a head-to-head match-up between MCTS and Alpha-Beta Pruning, it was observed that the Branching Factor, i.e., the number of moves per state, and the time allowed per move played a crucial role in determining the dominant algorithm. In the specific case where the branching factor was low, and sufficient time was provided to explore the moves with higher winning possibilities, Alpha-Beta Pruning had the advantage of winning the majority of the games. The reason for its success when the time allowed per move is high and the branching factor is low, is its higher accuracy and completeness. This is due to the exhaustive nature of Alpha-Beta pruning that enables it to explore the entire game tree, and completely evaluate all the moves in the given positions. Naturally, this means that when the branching factor is less and the time provided per move is sufficient, it can traverse the entire game tree and identify the correct move with higher accuracy, efficiently and effectively. (Avellan-Hultman and Querat, n.d.)

However, this result changes when the branching factor is just increased by 1, where MCTS starts gaining the advantage. This happens because, in cases of high branching factors, MCTS can simulate random moves which allows it to explore a much higher number of possible moves. It evaluates the moves by employing exploration and exploitation which enables it to find the optimal path and refine its strategy over time after running multiple iterations. Due to its ability to generate random simulations of the moves, and learn from its outcomes, thus improving over time, it doesn't require to learn predefined heuristics and rules. In a similar study conducted by Kavita Sheron(et al.), where a minimax agent was pitted against an MCTS agent, the result was similar. When the depth was increased from 3, the minimax agent failed to show improvement due to a lack of computational power, whereas the MCTS agent was able to win the games when the branching factor was high. (Avellan-Hultman and Querat, n.d.)

Although the game of Connect 4 in terms of action space and state space is not very complex, as a result of which algorithms such as Minimax and Alpha-Beta Pruning show great success, it is important that we study and research algorithms such as MCTS which has proven to highly effective in games with higher complexities and branching factors, such as Chess and Go.

Similarly, we also look at Reinforcement Learning and why it is great for the game of Connect 4. In the same study conducted by Kavita Sheron, a DQN agent was created. DQN is a self-play algorithm that plays with itself and improves over time by playing numerous episodes. When the same resources were given to the DQN agent and the minimax search agent, at a high branching factor, the DQN agent consistently beat the

minimax search agent. (Sheoran et al., 2022)

What's impressive about the MCTS agent is that when it was pitted against the DQN, it was victorious against it as well. However different factors are at play that result in an outcome that is in favor of MCTS. One of the reasons why MCTS has an advantage over DQN is that the DQN agent faces a maximization bias. This means that Q-values that are overestimated can eventually lead to suboptimal actions. Another reason why DQN is outperformed by MCTS is its catastrophic forgetting that results in hampering the network's ability to adjust to the new scenarios by overwriting the previous ones, which results in the deletion of data. MCTS however constructs decision trees that enable it to explore the new states and run simulations that minimize the chance of overestimating biases, thus allowing it to retain its previous experiences.

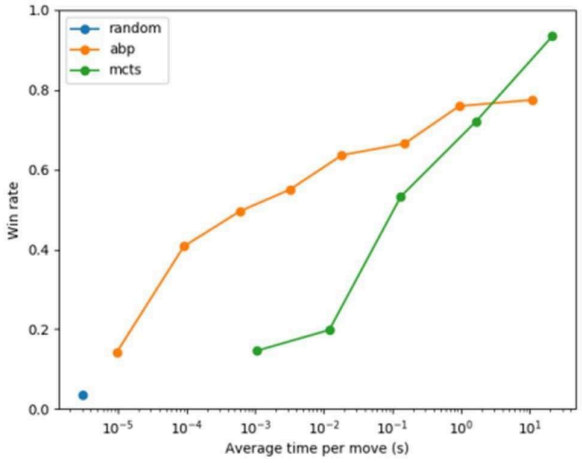

Figure 9 Adapted from Sheoran, K., Dhand, G.,Dabasz, M., Dahiya, N. and Pushparaj,P. (2022). SOLVING CONNECT 4 USING OPTIMIZED MINIMAX ANDMONTE CARLO TREE SEARCH. Advances and Applications in Mathematical Sciences, 21(6), pp.3303–3313.

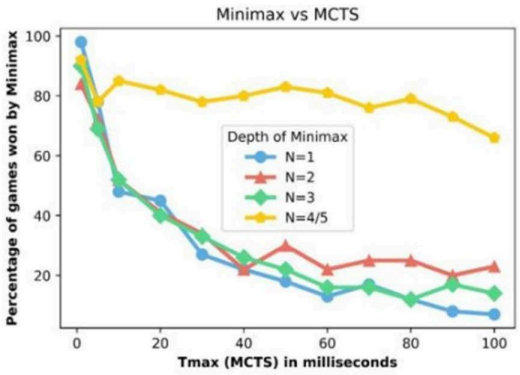

Figure 10. Winning percentage of Minimax against MCTS in Tmax milliseconds Adapted from Sheoran, K., Dhand, G., Dabasz, M., Dahiya, N. and Pushparaj, P. (2022). SOLVING CONNECT 4 USING OPTIMIZED MINIMAX AND MONTE CARLO TREE SEARCH. Advances and Applications in Mathematical Sciences, 21(6), pp.3303–3313.

2.2 Neural Networks

ResNet Neural Networks or Residual Neural Networks are Convolutional Neural Networks developed and programmed in a way capable of handling many convolutional layers. The important reason why ResNets are one of the best CNNs as they effectively solve the vanishing gradient problem due to their innovative solution of skip-connections. ResNets inherently excel at solving and identifying data with spatial relations. The reason why we mention ResNet architecture in this paper is that Neural Networks can be used in game-playing algorithms, to develop a policy over time which may lead to promising solutions. AlphaZero, developed by Google's DeepMind implemented the ResNet architecture and coupled it with MCTS algorithms to refine its search strategy over time and enhance its performance. We will discuss this further in Chapter 3 on how we can implement this (Datagen, n.d.).

3 Building the Prototypes

3.1 Developing a Vanilla Monte Carlo Tree Search Algorithm for the game of Connect-4

3.1.1 Introduction

Monte Carlo Tree Search was the core algorithm behind Google DeepMind's agents that were able to master the game of Chess and Go effectively (Google DeepMind, 2018). The efficiency of the algorithm allows it to navigate vast search spaces that enable it to develop optimal strategies that eventually lead to promising solutions. Although both Chess and Go are the ideal candidates for AI research projects, from the perspective of FYP and limited computational resources, Connect-4 is a great choice as it has a high branching factor with a large state space of 10^12, it is still tractable enough to track results and make improvements. It offers a balance of complexity and simplicity which is one of the core features of algorithmic exploration, as mentioned in Chapter 1. MCTS includes the process of running through a lot of different result scenarios, picking the best moves and their results and visualising the whole tree structure. Employing a simulation where MCTS returns a success rate for each possible move by which the same information is backpropagated leading to a search tree can effectively guide games with high branch factors and unknown outcomes. Connect Four, whose game state space is comparatively much narrower than that of Go; however, the level of complexity is sufficient enough to require formulation of plans and having foresight, provides great support for MCTS-based algorithms implementation and testing. MCTS-powered programs can develop sophisticated strategies for Connect Four by iterative refinement and learning from simulated gameplay. This gives us a new insight into decision-making in strategic board games. This indicates that MCTS is not confined to specific areas; rather, these expand the domains of application and give MCTS a chance to solve a multitude of complex decision-making problems. Let's dive into bulding the prototype in the Python Programming Language.

3.1.2 Monte Carlo Tree Search (MCTS)

Tree Search Class

The idea behind learning a game is identifying the winning moves over time that lead to a high score. In Reinforcement Learning, the agent can achieve these 2 ways: -

- Either it can learn from the reward function
- Or it can learn from knowing which states are inherently better

MCTS employs the second strategy to identify the promising solutions that lead to winning moves

Intuition of MCTS

Among the various types of tree traversal algorithms you may have encountered, the most common one is probably the Depth First Search (DFS). Take, for instance, an event where some DFS-based algorithms would be used to play the game whereby all the possible outcomes would be simulated. If we can simulate each situation, all available options and actions for all the states can be easily found. Nonetheless, it is very computing-intensive to calculate every single possible outcome. One can distinguish between 10^120 different games of chess – that's way more than the count of atoms in the universe. This very simple way of doing things gave the result: with the world's fastest supercomputer being able to carry out 10^18 floating-point operations per second this inference was established: at least 10^102 seconds which is about 1095 years. The following is just a part of a tree of the simple game tic-tac-toe (Qi Wang's Blog!, 2022).

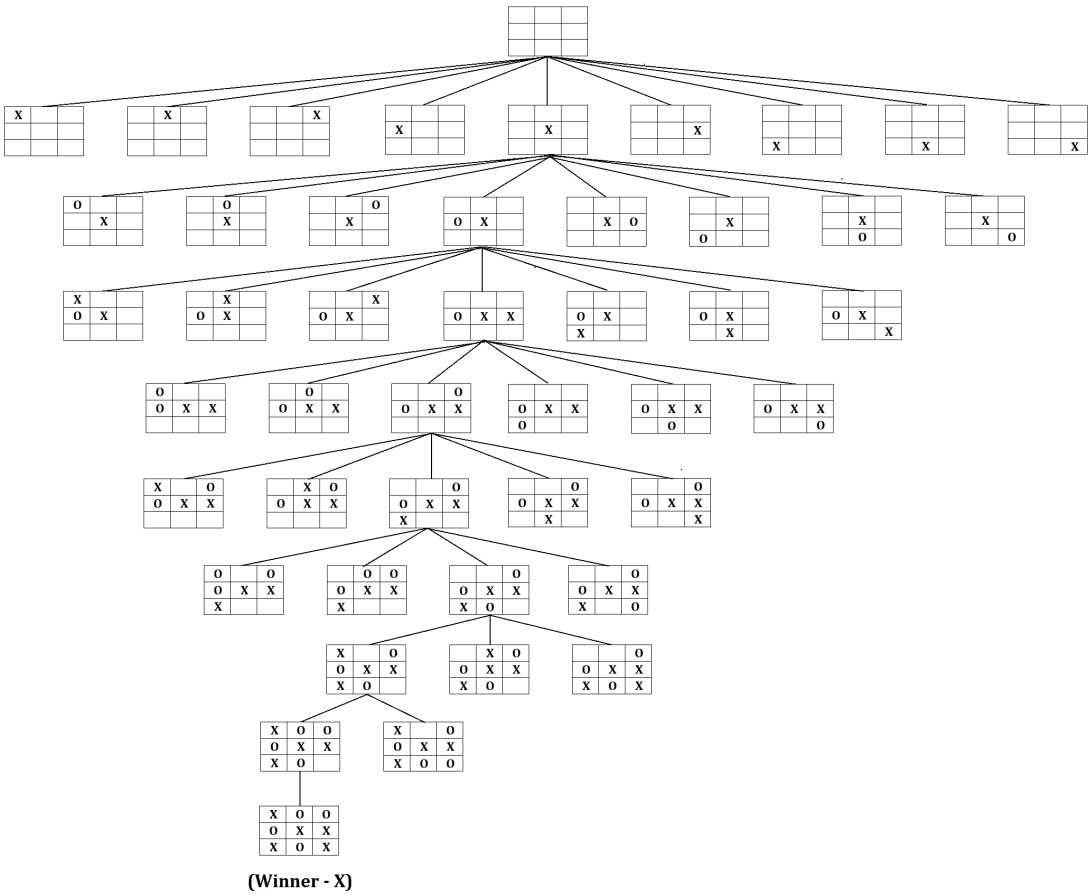

Figure 11 shows the the tree traversal in Tic-Tac-To using DFS adapted from Qi Wang's Blog! (2022). Connect 4 with Monte Carlo Tree Search. [online] Available at: https://www.harrycodes.com/blog/monte-carlo-tree-search

Of course, DFS is not going the optimal choice here. This why Monte Carlo search tree becomes the pivotal choice here. By attuning the DFS method according to the actions that tend to be winning, we can ensure that the search process is most effective. This makes it time efficient as there is no longer a need to consider all the moves moves and instead only focus on how the different options are better. MCTS in this scenario, the algorithm understands how to execute the best strategy through a series of random gameplay with itself by using simulated calculations (Qi Wang's Blog!, 2022).

Building the Tree

As mentioned in the Mechanics in Chapter 2, there are four phases of implementing MCTS:

- Selection
- Expansion

- Rollout
- Backpropagation

I will be implementing these 4 steps in the algorithm which will be performed repeatedly, within the defined ***time limit***. My aim during developing this algorithm will not be to allocate the resources towards suboptimal actions, for it to be able to fully explore good actions within lower time constraints.

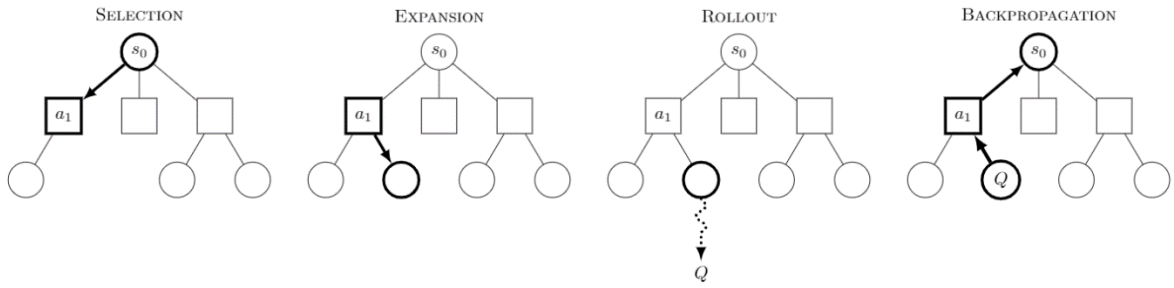

Figure 12 shows the MCTS Mechanics from Chapter 2

Selection Phase

For it to choose the node in the tree from which it will expand to the next node and carry out the game simulations, we must first define the selection function. We will choose the leaf that yields the greatest Q-values and proceed in that manner. If a node is not yet investigated, we will choose the leaf node that has not been explored yet. If however, the leaf node if it hasn't been explored yet, we add all potential actions to the tree as its children node and choose a node at random. This program of the Monte Carlo Tree search class that plays the game of Connect-4 was inspired by Qi Wang's model and reengineered by myself to conduct the experiments. (Qi Wang's Blog!, 2022).

To ensure that there is a balance of exploration vs exploitation, we will employ the Upper Confidence bound (also used by AlphaGo) applied to Trees to determine the value of the node. Below is the equation:

$$\mathrm{UCT}(n_i) = \frac{Q}{N} + c_i\sqrt{\frac{\log N_p}{N}}$$

Figure 13 UCT Formula

n_i is the current node. There are Q amount of wins the agent has out of the N simulations in

that given state. C_i is a float from 0 to 1, representing the exploration rate of the agent. N_p is the number of simulations that the parent of n_i has.

As shown in the equation above, it can be observed that the node is more likely to be explored if it yields a high probability of winning, and it has been relatively unexplored in comparison to the parent node.

```
class Node:
    def __init__(self, move, parent):
        self.move = move
        self.parent = parent
        self.N = 0
        self.Q = 0
        self.children = {}
        self.outcome = GameMeta.PLAYERS['none']

    def add_children(self, children: dict) -> None:
        for child in children:
            self.children[child.move] = child

    def value(self, explore: float = MCTSMeta.EXPLORATION):
        if self.N == 0:
            # we prioritize nodes that are not explored
            return 0 if explore == 0 else GameMeta.INF
        else:
            return self.Q / self.N + explore * math.sqrt(math.log(self.parent.N) / self.N)
```

Figure 14 Node Class

This above code defines the class called ***'Node'***, which, is the key to the whole Monte Carlo Tree Search (MCTS) algorithm. A game tree consists of **"nodes",** wherein a player can move to different perspectives or states and has several features such as the actual move itself, a connection to its parent node, the number of times **("N")** it has been visited, the aggregate reward **("Q")** associated with it, a dictionary of all its children nodes, and the game state the node refers to. The **'add_children'** function will facilitate the development of the game tree as it searches it by including children nodes in the current node. The exploration parameter and the number of visits to the node (denoted by **"N"**) are used by the **"value"** method to calculate the node's value. The given node is built around the idea of exploration versus exploitation by weighting both the exploratory term ***('explore * math.sqrt(math.log(self.parent.N) / self.N)'***. When it comes to model composition, our algorithm primarily relies on an agent **('N')**, the average reward **('Q/N')**, and handles the most complex behaviour needed for optimisation (at least when compared to other algorithms). The node will be visited only if it has not been visited **('N == 0')**, and exploration will be given priority by returning a high value. This algorithm operates based

on principles set forth here in the code, which ensure the constant search for optimal solutions in complex decision situations, particularly in the long-term game of the Connect Four (Qi Wang's Blog!, 2022).

The code for selecting a node is shown below:

```
def select_node(self) -> tuple:
    node = self.root
    state = deepcopy(self.root_state)

    while len(node.children) != 0:
        children = node.children.values()
        max_value = max(children, key=lambda n: n.value()).value()
        # select nodes with the highest UCT value
        max_nodes = [n for n in children if n.value() == max_value]

        # randomly select on to expand upon
        node = random.choice(max_nodes)
        state.move(node.move)

        if node.N == 0:
            return node, state

    if self.expand(node, state): # determines if the state is a terminal state (game over)
        node = random.choice(list(node.children.values()))
        state.move(node.move)

    return node, state
```

Figure 15 select_node function

The function ***select_node*** plays a critical role in the MCTS algorithm (Monte Carlo Tree Search) for the games' AI which uses search and reasoning for game playing. Its specific role within the game tree is to choose the next node to explore. The function starts by assigning the node variable the root node of the tree and the state variable a deep copy of the root game state. The algorithm for the next step operates is a loop that keeps on running as long as the current node has children (possible children for the next game state).
In the loop, the first thing that the function does is to get all the children of the current node and then it determines the UCT (Upper Confidence Bound for Trees) value for all the children and stores the highest value among them. The UCT value of a node is a mixture of exploration (controlled by a parameter) and exploitation (Q/N, where Q is the cumulative reward and N is the number of visits). Nodes with bigger UCT values are considered to be more precise and because of this additional exploration of such nodes is advantageous.
The loop skips a child node (node having no children) until the leaf node (node having no children) is reached. Then, the function of the function is to determine whether the node is a final point of termination or has to be divided/broken. e. On the other hand, a golf game could be intense (e.g., if the game is over). When the game is still played, the `***expand***` method is used to create and include an additional child node to the game tree indicating the likely approaches in the future (Qi Wang's Blog!, 2022).

Expansion Phase

Having chosen the node, any other actions carried out for this node should be made as direct children under it:

```python
def expand(self, parent: Node, state: ConnectState) -> bool:
    if state.game_over():
        return False

    children = [Node(move, parent) for move in state.get_legal_moves()]
    parent.add_children(children)

    return True
```

Figure 16 expansion function

This Python code's expand method creates branches for a given parent node and game state, thus expanding the game tree. Initially, the function returns false and checks whether the game is over. When it takes place, the algorithm does not only assign the level of the branch position to the child node, but it also adds it to the parent node and returns a true, which means the expansion is successful. This is because of game state calculation inside the MCTS algorithm is shortened due to the fact that this process expands the game tree dynamically.

Rollout/Simulation Phase

In the process of the rollout phase, we are mapping the initial situation provided by the game as an implication of the random walk. Through this function, each game's probability will be updated by returning a game that is generated through simulation. Here's the code in Figure 17:

```python
def roll_out(self, state: ConnectState) -> int:
    while not state.game_over():
        state.move(random.choice(state.get_legal_moves()))

    return state.get_outcome() # function in the game class shown at the bottom
```

Figure 17 rollout function

Until the game is over, the ***roll_out*** function refers to the simulation of the entire game from a certain state. It picks out legal movements of the highest priority and then starts

cycling through all the game statuses until it reaches the prescribed endpoint. The next step is the return of a result, which is an output that was calculated using the ***get_outcome*** method of the state ***object***. This function plays a very important role in the MCTS algorithm, which helps to simulate game outcomes and then use that information for decision-making.

Backpropagation Phase

This phase ensures that when the node values with the highest Q-values that is the reward is obtained, it is backpropagated up the game tree, updating all the ancestor nodes that result in a promising game move.

```
def back_propagate(self, node: Node, turn: int, outcome: int) -> None:
    # For the current player, not the next player
    reward = 0 if outcome == turn else 1

    while node is not None:
        node.N += 1
        node.Q += reward
        node = node.parent
        if outcome == GameMeta.OUTCOMES['draw']: # we count it as a loss for every state
            reward = 0
        else:
            reward = 1 - reward # alternates between 0 and 1 because each alternate depth repres
```

Figure 18 backpropagation function

The ***back_propagate*** method changes the statistics of the game tree's ***nodes*** based on the results of played simulations. It requires three parameters: ***turn***, which is a turn indicator of whose player will commence the simulation; ***outcome***, indicating the result of the simulated game; and node, which denotes the starting node of backpropagation. As a result, state the function distinctions that are currently in play for the variable. After that, it goes over the node's predecessors and updates their ***visit count (N)*** and ***cumulative reward (Q)*** accordingly. Where the backpropagation algorithm maintains the same reward despite a draw, or else the winnings will be less accurate, and thus the payout will be changed from 0 to 1 to represent a player's turn and another player's turn, respectively. This function of the Monte Carlo Tree Search algorithm is the most crucial part that helps to update the nodes' statistics in the game tree. It is from this function that the algorithm can learn from the simulated game outcomes and make informed decisions.

Combining the Four Phases

Now we just select and expand upon the node that will generate the rollout simulations, then backpropagate the results onto its ancestor nodes. This process is repeated for a set ***time_limit***.

```python
def search(self, time_limit: int):
    start_time = time.process_time()

    num_rollouts = 0
    while time.process_time() - start_time < time_limit:
        node, state = self.select_node()
        outcome = self.roll_out(state)
        self.back_propagate(node, state.to_play, outcome)
        num_rollouts += 1 # for calculating statistics

    run_time = time.process_time() - start_time
    self.run_time = run_time
    self.num_rollouts = num_rollouts
```

Figure 19 search function of the mcts algorithm

On my local machine, setting the time limit to 5 seconds generated about 150,000 rollouts.

Choosing the Best Action

The game tree considerably simplifies the process of choosing the best course of action. We end up with the state per node, which subsequently presents the highest **N** value. Significantly, we pick not the topmost **Q/N**, as this could be from an explored node to which the exploration has not been done with adequate depth. Nevertheless, we normally don't check the worst scenarios, so a node with a high **N** is almost always the best option.

```python
def best_move(self):
    if self.root_state.game_over():
        return -1

    max_value = max(self.root.children.values(), key=lambda n: n.N).N
    max_nodes = [n for n in self.root.children.values() if n.N == max_value]
    best_child = random.choice(max_nodes)

    return best_child.move
```

Figure 20 best move function

The statistics gathered during the search use the **best_move** function to identify the move with the highest reward. It is during this stage that the game implemented by the root state node is checked and verified to be still going on. If not, it will return -1 in the process. If

the root node is not the most explored, the program looks for the child node that represents the most explored move among the children of the root and has the largest visit count **(N)**. The two child nodes are picked randomly out of all the others as they have the same number of visits.

Playing Move

```
def move(self, move):
    if move in self.root.children:
        self.root_state.move(move)
        self.root = self.root.children[move]
        return

    self.root_state.move(move)
    self.root = Node(None, None)

def statistics(self) -> tuple:
    return self.num_rollouts, self.run_time
```

Figure 21 move function

The move method returns the child node associated with the root node as the move for the algorithm.
The statistics method returns a tuple that contains the number of MCTS actions simulated per run ***(num_rollouts)*** and the total time taken by the algorithm during the run ***(run_time)***.

3.2 Alpha MCTS

The aim of developing the model AlphaMCTS was to enhance the traditional Monte Carlo Tree Search (MCTS) algorithm by integrating elements from the revolutionary AlphaZero developed by Google's DeepMind. The aim was to create a robust and successful model capable of learning and effectively playing the game of Connect-4, in a style similar to the models at DeepMind that play chess and Go (Google DeepMind, 2018). My aim with this project was to optimize the playing strategies in such a way that would overcome the limitations of the conventional MCTS algorithm, which would help us achieve significant advancements in AI applications in board games like Connect-4. This model was inspired by Alex's model of the Connect-4 agent and re-engineered by myself to conduct experiments(kaggle.com, n.d.).

3.1.3 Methodology

Game Engine Development: The first stage was designing an effective game engine underneath the complex AlphaMCTS agent that would be used to play a game of Connect-Four. This engine is important for the agent as it enables it to move, evaluate the game, and play against opponents during gameplay.

Neural Network Design: The policy and value functions that are created for effective gameplay, thus the deep neural network model is developed precisely. The AlphaMCTS agent could make strategic decisions by deploying the model, which can evaluate the game, and predict the optimal moves, hence increasing its strategic decision-making capabilities.

Self-Play Generation: Thus, the self-play process was initialised with the aid of a neural network using the Monte Carlo tree search. The agent engaged with itself in the recurrent self-play sessions, playing games against itself in the game tree and optimising its strategies according to the search probability and observed results.

Training Phase: After being trained sufficiently, the neural network weights are updated based on the search probabilities and results produced by the self-play data. This part of the process was dedicated to enhancing the network's decision-making abilities and its performance to deal with various gaming scenarios.

Performance Evaluation: The trained neural network's performance was thoroughly checked, and the focus was on its ability to identify the important game states, such as instant wins or vital blocks. These assessments served as a significant criterion for how the neural network was able to claim its performance for strategic decisions made in the gaming environment. Essentially, my goal was to develop AlphaMCTS in a way that enhances the Connect-4 AI gameplay abilities, by improving the traditional MCTS algorithm (kaggle.com, n.d.).

3.1.4 Configuration and Hyperparameters

```
config_map = {
    'device': torch.device('cuda') if torch.cuda.is_available() else 'cpu',
    'num_filters': 128,              # Number of convolutional filters used in residual blocks
    'num_residual_blocks': 8,              # Number of residual blocks used in network
    'exploration_constant': 2,     # Exploration constant used in PUCT calculation
    'selection_temperature': 1.25,          # Selection temperature. A greater temperature is a more uniform distribution
    'dirichlet_alpha': 1.,         # Alpha parameter for Dirichlet noise. Larger values mean more uniform noise
    'dirichlet_epsilon': 0.25,         # Weight of dirichlet noise
    'learning_rate': 0.001,        # Adam learning rate
    'training_epochs': 1,         # How many full training epochs
    'games_per_epoch': 1,        # How many self-played games per epoch
    'minibatch_size': 128,          # Size of each minibatch used in learning update
    'num_minibatches': 4,            # How many minibatches to accumulate per learning step
    'mcts_initial_iterations': 50,  # Number of Monte Carlo tree search iterations initially
    'mcts_max_iterations': 150,   # Maximum number of MCTS iterations
    'mcts_search_increment': 1,    # After each epoch, how much should search iterations be increased by
}
```

Figure 22 Hyperparameters

The ***config_map*** dictionary also contains configuration parameters for the training of the AlphaMCTS agent in the game of Connect Four. These variables describe what kind of structure is used for the agent's architecture, how it is trained, and how the exploration strategy is utilized. Critical parameters entail the device selection between CPU or GPU; the network architecture attributes including convolutional filter number, and residual block number; and the exploitation-exploration hyperparameters during gameplay, among others, like exploration constant, and selection temperature.
Also, some other parameters like the learning rate, the number of training epochs, and the batch size are presented as they are related to the training logistics. In configuration, also there are parameters which govern the MCTS algorithm, which include an initial number of search iterations and the maximum number of searches along with the increment in search instances after every training epoch (kaggle.com, n.d.).
The following parameters together determine the nature of the AlphaMCTS player, providing them with a learning mechanism and gameplay strategy that fit with the environment of Connect Four.

3.1.5 Connect-4 Game Engine

```python
class Connect4Engine:
    "Engine for Connect 4 game with methods for game-related tasks."

    def __init__(self):
        self.num_rows = 6
        self.num_cols = 7

    def get_next_state(self, current_state, selected_action, to_play=1):
        "Update the current state based on the selected action and return the resulting board."
        # Preconditions
        assert self.evaluate(current_state) == 0
        assert np.sum(abs(current_state)) != self.num_rows * self.num_cols
        assert selected_action in self.get_valid_actions(current_state)

        # Find the next empty row in the selected column
        row_index = np.where(current_state[:, selected_action] == 0)[0][-1]

        # Apply the action
        new_state = current_state.copy()
        new_state[row_index, selected_action] = to_play
        return new_state

    def get_valid_actions(self, current_state):
        "Return an array containing the indices of valid actions."
        # If the game is over, there are no valid moves
        if self.evaluate(current_state) != 0:
            return np.array([])

        # Identify the columns where pieces can be placed
        col_sums = np.sum(np.abs(current_state), axis=0)
        return np.where((col_sums // self.num_rows) == 0)[0]
```

Figure 23 Connect-4 game engine class

The **Connect4Engine** class serves the purpose of an engine that controls Connect Four games, and the functions provide for a variety of tasks, like the ones that are critical for playing the game. It starts with the game board's dimensions being set to six rows and seven columns. The **get_next_state** function proceeds with the board of the resultant state and alters the game state in tune with the chosen action. This is to check the preconditions, like the absence of a winner, the validity of moves and whether the correct action is

selected. The **get_valid_actions** function finds and returns the indices of legal actions that apply to the current game state. Evaluating the move option evaluates if the player has a winning condition horizontally, vertically, and diagonally to check the game end. It stipulates 0 for player 1 to win, -1 for player 2 to win, and 1 otherwise. The play method in the current state conducts an action and returns the next state, the prize, and a termination flag, which indicates the game is over. The state is transformed to a 3D tensor using the function **encode_state** with the (player 1) pieces, (player 2) pieces, and empty cells. The **reset** method is called when the game board reaches the empty state of the playing board and ready for a new game. This class comprises all the features that are essential for managing the state and gaming in Connect Four (kaggle.com, n.d.).

3.1.6 Residual Neural Network

This code defines the architecture for the CNN in this case, the residual neural network (ResNet) that serves as the core of the AlphaMCTS algorithm applied to Connect 4. The great thing about ResNets is that they are known for their ability to learn from the addition of residual, or "skip" connections, mitigating the vanishing gradient problem in deep networks (He et al. 2015.). This is inspired by Google DeepMind's AlphaZero model which also utilized a ResNet as the neural network for their agent (Google DeepMind, 2018).
In the case of AlphaMCTS, the ResNet that I designed consists of two distinct heads branching from the common convolutional base:

1. **Policy head**: This outputs the probability distribution over possible actions, that guides the agent on which column to drop the disc in during gameplay.

2. **Value Head**: This produces the scalar value within the range [-1, 1] that approximates the value of the current board state, helping to evaluate the board's win-lose potential.

A three-channel presentation of the board is accepted by the network. These channels facilitate the network's ability to distinguish between and process board information by capturing three components of the game state: the relationship between the player's discs, the opponent's discs, and the empty areas, which are also known as the 'board' itself. ResNets are good at detecting the general arrangement of a game board and spatial interactions between the game pieces, which are particularly important in classifying these games, like Connect Four. The network can do this because of the tiers that it has, which allow it to transmit this spatial information more effectively

```python
class ResidualNeuralNetwork(nn.Module):
    "Complete residual neural network model."

    def __init__(self, game_engine, model_config):
        super().__init__()

        # Board dimensions
        self.board_size = (game_engine.num_rows, game_engine.num_cols)
        num_actions = game_engine.num_cols  # Number of columns represent possible actions
        num_filters = model_config.num_filters

        self.base = ConvolutionBase(model_config)  # Base layers

        # Policy head for action selection
        self.policy_head = nn.Sequential(
            nn.Conv2d(num_filters, num_filters//4, kernel_size=3, padding=1),
            nn.BatchNorm2d(num_filters//4),
            nn.ReLU(),
            nn.Flatten(),
            nn.Linear(num_filters//4 * self.board_size[0] * self.board_size[1], num_actions)
        )

        # Value head for state evaluation
        self.value_head = nn.Sequential(
            nn.Conv2d(num_filters, num_filters//32, kernel_size=3, padding=1),
            nn.BatchNorm2d(num_filters//32),
            nn.ReLU(),
            nn.Flatten(),
            nn.Linear(num_filters//32 * self.board_size[0] * self.board_size[1], 1),
            nn.Tanh()
        )
```

Figure 24 Shows the Resnet Class

The class **ResidualNeuralNetwork** is the complete residual neural network model developed for the Connect Four game. It consists of a base convolutional network and two heads: one with the help of which we solve problems (policy selection) and the other by which we evaluate states (value estimation). The basis for the network is formed by the function of the **ConvolutionBase** class, which is made up of multiple residual blocks that are responsible for generating the features from the game state. Each residual block has two convolutional layers with batch normalization, this is how the residual connection mechanism is implemented. The policy head is responsible for predicting the probability of the specific action, while the value head estimates the value of the current game's state. In the forward propagation, the input travels the base network causing the final feature map to be passed onto both the policy and the value heads which in turn produce the respective output. Such architecture is made up of modules which are used for efficient learning and decision-making in the Connect Four game situations, by using the residual connections for feature extraction and prediction.

```
    def forward(self, x):
        x = self.base(x)
        x_value = self.value_head(x)
        x_policy = self.policy_head(x)
        return x_value, x_policy

class ConvolutionBase(nn.Module):
    "Convolutional base for the network."

    def __init__(self, model_config):
        super().__init__()

        num_filters = model_config.num_filters
        num_residual_blocks = model_config.num_residual_blocks

        # Initial convolutional layer
        self.conv = nn.Sequential(
            nn.Conv2d(3, num_filters, kernel_size=3, padding=1),
            nn.BatchNorm2d(num_filters),
            nn.ReLU()
        )

        # List of residual blocks
        self.res_blocks = nn.ModuleList(
            [ResidualBlock(num_filters) for _ in range(num_residual_blocks)]
        )

    def forward(self, x):
        x = self.conv(x)
        for block in self.res_blocks:
            x = block(x)
        return x
```

Figure 25 shows the ConvolutionalBase class with the residual blocks

3.1.7 Monte Carlo Tree Search in AlphaMCTS

Vanilla MCTS calculates the value of each state by using random rollouts, which is very expensive in terms of computation and sometimes not accurate enough if the state space is very high.

Limitations of Vanilla MCTS

- **Inaccurate approximations**: In the case of the complicated game environment, the random rollouts, which are the main features of the vanilla version, are utilised as an approximation of the state value.
- **Computational expense**: The number of rollouts and simulations that the algorithm plays out to arrive at an accurate approximation can be extremely large, thus making it computationally expensive.

AlphaMCTS's Modifications

- **Neural network evaluation**: Instead of using random rollouts, AlphaMCTS approximates the policy (action probabilities) and the state value using a neural network, which makes the search more focused and effective. It does so by using the rollouts as the target of the policy head, which improves as games are played.
- **Dirichlet noise**: For increasing the exploration of rare-visit nodes, Dirichlet noise is added by AlphaMCTS placing the prior probability on the root node.

- **PUCT score**: The UCB function is called forth for tree traversal at MCTS. AlphaMCTS employs the PUCT (Polynomial Upper Confidence Trees), which is an improved version that strikes a better balance between exploration and exploitation. It does so by exploiting the highly rewarding rollouts from the game simulations and exploring the non-played nodes.

```python
class MonteCarloTreeSearch:
    def __init__(self, neural_net, game_engine, model_config):
        "Initialize Monte Carlo Tree Search with a given neural network, game instance, and configuration."
        self.neural_net = neural_net
        self.game_engine = game_engine
        self.model_config = model_config

    def search(self, initial_state, total_iterations, selection_temperature=None):
        "Performs a search for the desired number of iterations, returns an action and the tree root."
        # Create the root
        root = TreeNode(None, initial_state, 1, self.game_engine, self.model_config)

        # Expand the root, adding noise to each action
        valid_actions = self.game_engine.get_valid_actions(initial_state)
        state_tensor = torch.tensor(self.game_engine.encode_state(initial_state), dtype=torch.float).unsqueeze(0).to(self.model_config.device)
        with torch.no_grad():
            self.neural_net.eval()
            value, logits = self.neural_net(state_tensor)

        # Get action probabilities
        action_probs = F.softmax(logits.view(self.game_engine.num_cols), dim=0).cpu().numpy()

        # Calculate and add Dirichlet noise
        noise = np.random.dirichlet([self.model_config.dirichlet_alpha] * self.game_engine.num_cols)
        action_probs = ((1 - self.model_config.dirichlet_epsilon) * action_probs) + self.model_config.dirichlet_epsilon * noise

        # Mask unavailable actions
        mask = np.full(self.game_engine.num_cols, False)
        mask[valid_actions] = True
        action_probs = action_probs[mask]

        # Softmax
        action_probs /= np.sum(action_probs)
```

Figure 26 The MCTS class

```
# Begin search
for _ in range(total_iterations):
    current_node = root

    # Phase 1: Selection
    # While not currently on a leaf node, select a new node using PUCT score
    while not current_node.is_leaf():
        current_node = current_node.select_child()

    # Phase 2: Expansion
    # When a leaf node is reached and it's not terminal; expand it
    if not current_node.is_terminal():
        current_node.expand()
        # Convert node state to tensor and pass through network
        state_tensor = torch.tensor(self.game_engine.encode_state(current_node.state), dtype=torch.float).unsqueeze(0).to(self.model_config.device)
        with torch.no_grad():
            self.neural_net.eval()
            value, logits = self.neural_net(state_tensor)
            value = value.item()

        # Mask invalid actions, then calculate masked action probs
        mask = np.full(self.game_engine.num_cols, False)
        mask[valid_actions] = True
        action_probs = F.softmax(logits.view(self.game_engine.num_cols)[mask], dim=0).cpu().numpy()
        for child, prob in zip(current_node.children.values(), action_probs):
            child.prob = prob
    # If node is terminal, get the value of it from game instance
    else:
        value = self.game_engine.evaluate(current_node.state)
```

Figure 27 Search method to choose the node

During the MCTS search, firstly the tree structure is dynamically devised with each node standing for a game state and the descriptive statistics correspondingly. Node weight updating involves three selections, expansion, simulation and backpropagation, each of which is based on the working principle of exploration-exploitation and the neural network model. The **TreeNode** class is a wrapper class for the node specifications that includes expansion, selection of the child nodes, and backpropagation of the evaluation scores.

```
expand(self):
"Create child nodes for all valid actions. If state is terminal, evaluate and set the node's value."
# Get valid actions
valid_actions = self.game_engine.get_valid_actions(self.state)

# If there are no valid actions, state is terminal, so get value using game instance
if len(valid_actions) == 0:
    self.total_score = self.game_engine.evaluate(self.state)
    return

# Create a child for each possible action
for action in valid_actions:
    # Make move, then flip board to perspective of next player
    child_state = -self.game_engine.get_next_state(self.state, action)
    self.children[action] = TreeNode(self, child_state, -self.to_play, self.game_engine, self.model_config)

select_child(self):
"Select the child node with the highest PUCT score."
best_puct = -np.inf
best_child = None
for child in self.children.values():
    puct = self.calculate_puct(child)
    if puct > best_puct:
        best_puct = puct
        best_child = child
return best_child

calculate_puct(self, child):
"Calculate the PUCT score for a given child node."
# Scale Q(s,a) so it's between 0 and 1 so it's comparable to a probability
# Using 1 - Q(s,a) because it's from the perspective of the child – the opposite of the parent
exploitation_term = 1 - (child.get_value() + 1) / 2
exploration_term = child.prob * math.sqrt(self.n_visits) / (child.n_visits + 1)
return exploitation_term + self.model_config.exploration_constant * exploration_term
```

Figure 28 PUCT algorithm integration

We are able to address the limitations of vanilla MCTS by integrating a neural network approximation function. This enables AlphaMCTS to significantly improve its performance and effectiveness.

3.1.8 AlphaMCTS Algorithm

The heart of the AlphaMCTS algorithm is contained in this class, AlphaMCTS. It combines a ResNet with MCTS so that it can play and learn on its own.

- **Initialization**: pre-allocates GPU RAM for training, configures the neural network, and uses MCTS.
- **Memory Management**: effectively stores training data on the GPU by utilising pre-allocated memory.
- **Training loop**: This initiates the training of the agent over a set number of Epochs.
- **Self-play**: In self-play mode, all game states, values, and policies are stored in memory while a single game is played through.
- **Learning Step**: Updates the neural network's parameters by carrying out multiple mini-batch learning steps.

```
        # MCTS search iterations
        self.search_iterations = model_config.mcts_initial_iterations

        # Logging
        self.verbose = verbose
        self.total_games = 0

    def train(self, training_epochs):
        "Train the AlphaMCTS agent for a specified number of training epochs."
        # For each training epoch
        for _ in range(training_epochs):

            # Play specified number of games
            for _ in range(self.model_config.games_per_epoch):
                self.self_play()

            # At the end of each epoch, increase the number of MCTS search iterations
            self.search_iterations = min(self.model_config.mcts_max_iterations, self.search_iterations + self.model_config.mcts_search_increment)

    def self_play(self):
        "Perform one episode of self-play."
        state = self.game_engine.reset()
        done = False
        while not done:
            # Search for a move
            action, root = self.mcts.search(state, self.search_iterations)

            # Value target is the value of the MCTS root node
            value = root.get_value()

            # Visit counts used to compute policy target
            visits = np.zeros(self.game_engine.num_cols)
            for child_action, child in root.children.items():
                visits[child_action] = child.n_visits
            # Softmax so distribution sums to 1
            visits /= np.sum(visits)
```

Figure 29 defining self-play and training functions in the AlphaMCTS class

3.1.9 Evaluator

Evaluating the neural network's capacity to make clear-cut winning moves—like winning or stopping an opponent's winning move—is the goal of the Evaluator class. It combines

with an already-running AlphaMCTS instance, taking over its configurations, game logic, and network. Although this class isn't a good way to assess play at a high level, it can be used to show that an AlphaMCTS agent is progressing.
Components

- **Initialization**: Creates a set of sample states and activities for assessment using an AlphaMCTS instance.
- **Action selection**: Using basic game logic, choose an action that will either win the game right away or prevent the opponent from winning.
- **Example generation**: creates sets of examples for scenarios that win and those that block, then encodes them for analysis.
- **Evaluation**: Determines how accurate the neural network's predictions are based on the instances it generates.

```python
def select_action(self, state):
    "Select an action based on the given state, will choose winning or blocking moves."
    valid_actions = self.game.get_valid_actions(state)

    # Check for a winning move
    for action in valid_actions:
        next_state, reward, _ = self.game.play(state, action)
        if reward == 1:
            return action

    # Check for a blocking move
    flipped_state = -state
    for action in valid_actions:
        next_state, reward, _ = self.game.play(flipped_state, action)
        if reward == 1:
            return action

    # Default to random action if no winning or blocking move
    return random.choice(valid_actions)

def generate_examples(self):
    "Generate and prepare example states and actions for evaluation."
    winning_examples = self.generate_examples_for_condition('win')
    blocking_examples = self.generate_examples_for_condition('block')

    # Prepare states and actions for evaluation
    winning_example_states, winning_example_actions = zip(*winning_examples)
    blocking_example_states, blocking_example_actions = zip(*blocking_examples)

    target_states = np.concatenate([winning_example_states, blocking_example_states], axis=0)
    target_actions = np.concatenate([winning_example_actions, blocking_example_actions], axis=0)

    encoded_states = [self.game.encode_state(state) for state in target_states]
    self.X_target = torch.tensor(np.stack(encoded_states, axis=0), dtype=torch.float).to(self.model_config.device)
    self.y_target = torch.tensor(target_actions, dtype=torch.long).to(self.model_config.device)
```

Figure 30 displays the code for selecting action according to the current state and using that as the target

3.4.8 Training Loop

The process of training and assessment is coordinated by this code. After training for one epoch, the agent is assessed.

```
game_engine = Connect4Engine()
alpha_mcts = AlphaMCTS(game_engine, config)
evaluator = Evaluator(alpha_mcts)

# Evaluate pre-training
evaluator.evaluate()

# Main training/evaluation loop
for _ in range(config.training_epochs):
    alpha_mcts.train(1)
    evaluator.evaluate()

# Save trained weights
torch.save(alpha_mcts.network.state_dict(), 'alpha_mcts-network-weights-new.pth')
```

Figure 31 sets the model for 1 training loop for a set number of epochs specified in the hyperparameters

3.4.9 Plotting Policy Performance During Training

```
# Plot data
x_values = np.linspace(0, 101 * len(evaluator.accuracies), len(evaluator.accuracies))
y_values = [acc * 100 for acc in evaluator.accuracies]

# Create plot
plt.figure(figsize=(10, 6))
plt.plot(x_values, y_values, linewidth=2, marker='o', markersize=4, linestyle='-', color='#636EFA')

# Formatting
plt.xlabel('\nNumber of Games', fontsize=16)
plt.ylabel('Policy Evaluation Accuracy (%)', fontsize=16)
plt.title('Policy Evaluation\n', fontsize=24)
plt.grid(True, linestyle='--', linewidth=0.5, color='gray')

plt.show()
```

Figure 32 Code to generate graph for evaluation accuracy

The graph illustrates how the policy network's performance has improved over time. Remember that the policy network does not use any tree search to look ahead; it only detects moves that are presently accessible. This indicates that the performance displayed is just a simple metric to assess the tree search that is performed, which helps monitor the policy's evolution over time.

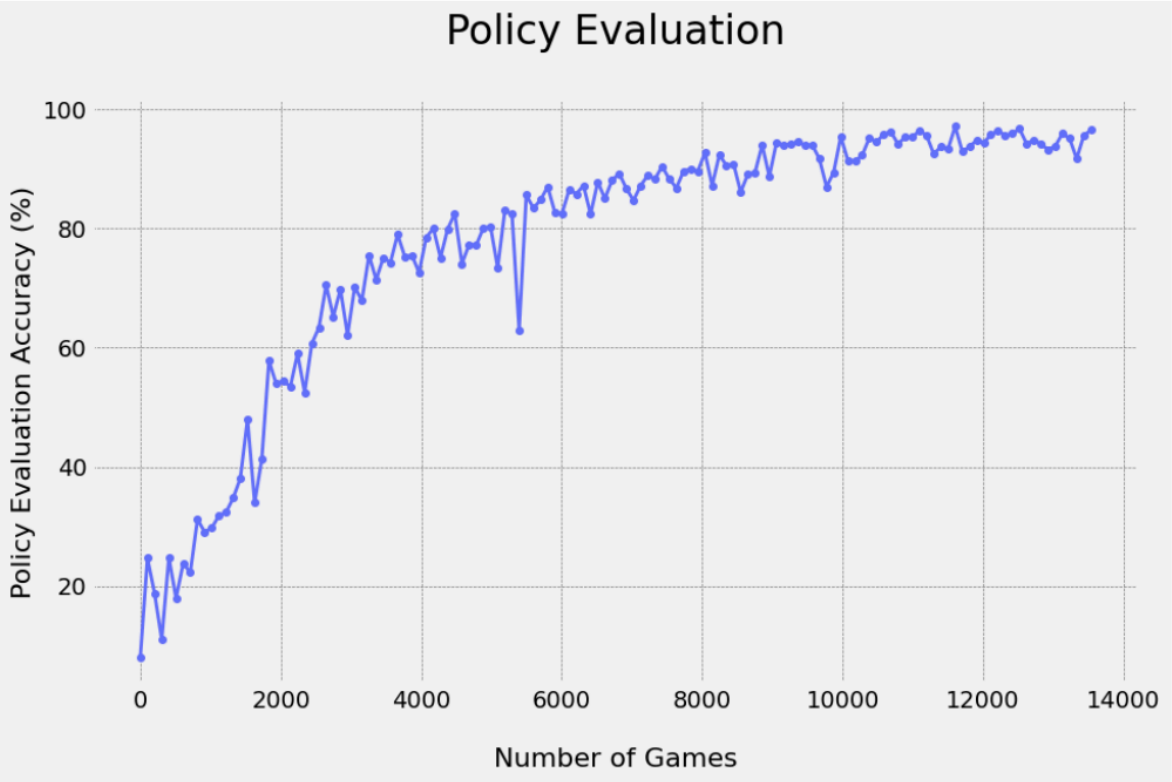

Figure 33 shows the graph's accuracy increasing over time, as it chooses the target action more frequently

3.4.10 Loading Pre-trained weights

```
# Define the game, AlphaMCTS, and evaluator
game_engine = Connect4Engine()
alpha_mcts = AlphaMCTS(game_engine, config)
evaluator = Evaluator(alpha_mcts)

# Load the pre-trained weights
file_path = "C:/Users/Rohan Arya/OneDrive/Desktop/FinalYearProject/alphamcts-network-weights-new.pth"
pre_trained_weights = torch.load(file_path, map_location=config.device)
alpha_mcts.network.load_state_dict(pre_trained_weights)

# Evaluate the pre-trained model
evaluator.evaluate()
```

Figure 34 Method to load the trained weights

This file, which I titled ***"alphamcts-network-weights-new.pth,"*** is essential for loading pre-trained weights into the AlphaMCTS model. I was able to avoid having to retrain the model each time it was used because of this. I may save time and computing resources by directly applying the knowledge I obtained during training sessions to new cases by

loading the pre-trained weights.

PyTorch is a great option for these kinds of projects because of its ease of use and versatility. I can load the pre-trained weights into our model and begin utilising them for evaluation or inference with just a few lines of code. One of PyTorch's main advantages over other deep learning frameworks is its ease of use (Pytorch.org, 2023).

Furthermore, compared to TensorFlow's static graph approach, PyTorch's eager execution architecture and dynamic computational graph provide a more natural and intuitive programming experience. Because of its versatility, we can simply debug and make changes to our models, which lowers the possibility of compatibility problems or mismatch errors, which frequently arise when using multiple versions of TensorFlow.

Using my prior knowledge from creating a DQN for the Atari Breakout game, I was able to overcome TensorFlow versioning and compatibility issues, which resulted in tedious failures and debugging work. On the other hand, PyTorch is my first option for deep learning jobs because of its smooth integration and simple syntax, which make it easier to load and use pre-trained models (Pytorch.org, 2023).

3.4.11 The AlphaMCTS agent

The class itself brings together the AlphaZero agent to play games against it

```python
class AlphaMCTSAgent:
    def __init__(self, alpha_mcts):
        self.alpha_mcts = alpha_mcts
        self.alpha_mcts.network.eval()

        # Remove noise from move calculations
        self.alpha_mcts.model_config.dirichlet_epsilon = 0

    def select_action(self, state, search_iterations=1000):
        state_tensor = torch.tensor(self.alpha_mcts.game_engine.encode_state(state), dtype=torch.float).to(self.alpha_mcts.model_config.device)

        # Get action without using search
        if search_iterations == 0:
            with torch.no_grad():
                _, logits = self.alpha_mcts.network(state_tensor.unsqueeze(0))

            # Get action probs and mask for valid actions
            action_probs = F.softmax(logits.view(-1), dim=0).cpu().numpy()
            valid_actions = self.alpha_mcts.game_engine.get_valid_actions(state)
            valid_action_probs = action_probs[valid_actions]
            best_action = valid_actions[np.argmax(valid_action_probs)]
            return best_action
        # Else use MCTS
        else:
            action, _ = self.alpha_mcts.mcts.search(state, search_iterations)
            return action
```

Figure 37 AlphaMCTS agent ready to play based on training or MCTS search iterations

3.4.12 GUI to play against the model

```python
agent = AlphaMCTSAgent(alpha_mcts)
# Constants for board dimensions
BOARD_WIDTH = 7
BOARD_HEIGHT = 6
CELL_SIZE = 60
DISK_RADIUS = CELL_SIZE // 2 - 5
WINDOW_PADDING = 20

# Function to handle human move
def human_move():
    global state, turn, done

    # Get the action entered by the player
    action = int(entry.get())

    # Check if the action is valid
    if action not in game_engine.get_valid_actions(state):
        messagebox.showerror("Invalid Move", "Please enter a valid move.")
        return

    # Perform the human move
    next_state, reward, done = game_engine.play(state, action)
    update_board(next_state)
```

Figure 36 GUI setup in Tkinter

Here, a human player can play against the AlphaMCTS model itself in a basic Connect-4 game board GUI. I have integrated 2 separate versions where the player can play against the model by inputting moves between 0-6 for move selection, or can also use cursor functionality to drag and drop pieces manually. The GUI was created using the Python library ***"tkinter".***

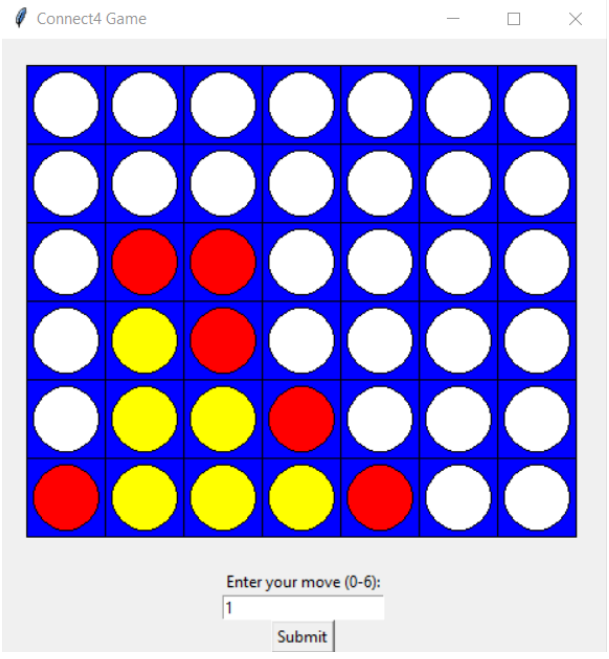

Figure 37 Game of Connect-4 against human vs AlphaMCTS with AlphaMCTS as the winner

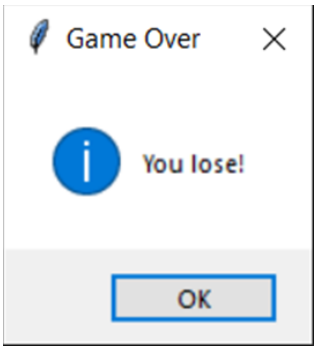

Figure 38 Game over window

Minimax

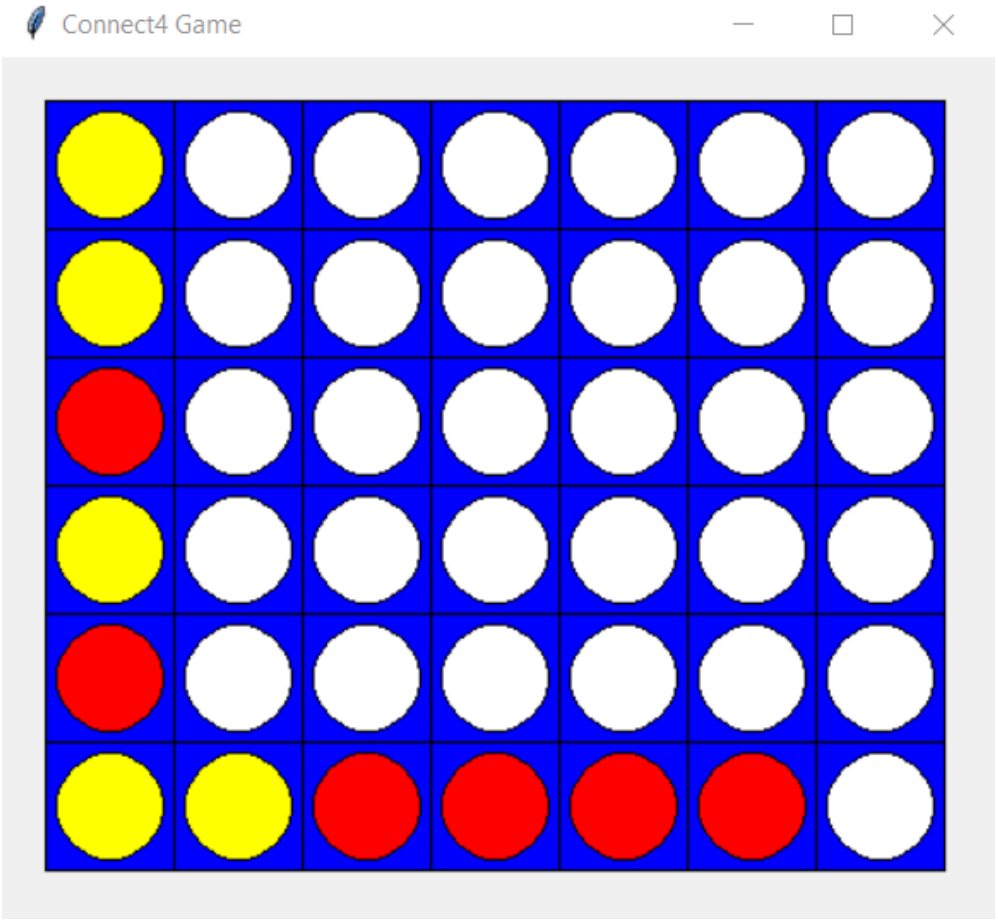

Figure 39. In the above image, we can see that the Minimax failed to block the winning move for AlphaMCTS

```python
# Minimax algorithm implementation
def minimax(state, depth, alpha, beta, maximizing_player):
    if depth == 6 or len(game_engine.get_valid_actions(state)) == 0:
        return game_engine.evaluate(state), None

    valid_actions = game_engine.get_valid_actions(state)

    if maximizing_player:
        value = float('-inf')
        best_action = None
        for action in valid_actions:
            next_state, _, _ = game_engine.play(state, action)
            eval, _ = minimax(next_state, depth - 1, alpha, beta, False)
            if eval > value:
                value = eval
                best_action = action
            alpha = max(alpha, value)
            if alpha >= beta:
                break  # Beta cut-off
        return value, best_action
    else:
        value = float('inf')
        best_action = None
        for action in valid_actions:
            next_state, _, _ = game_engine.play(state, action)
            eval, _ = minimax(next_state, depth - 1, alpha, beta, True)
            if eval < value:
                value = eval
                best_action = action
            beta = min(beta, value)
            if beta <= alpha:
                break  # Alpha cut-off
        return value, best_action
```

Figure 40 Minimax code in the GUI

This implementation of the minimax algorithm was designed to serve as an opponent for my AlphaMCTS model within the GUI gameboard environment. While the minimax algorithm is a classical approach for decision-making in adversarial games like Connect-4, this particular implementation had its limitations.

Primarily, due to the constraints imposed by the GUI environment and the simplified nature of the implementation, the minimax algorithm did not perform optimally. One notable flaw was its reliance on a fixed depth parameter for search, which limited its ability to evaluate complex game states effectively. Additionally, the absence of advanced techniques such as transposition tables or heuristic evaluation functions further constrained its performance.

Moreover, the computational demands of running both the GUI gameboard and the

minimax algorithm on my laptop, which had limited RAM, contributed to suboptimal gameplay experiences. The algorithm's performance was adversely affected by computational bottlenecks, leading to slower decision-making and potential inaccuracies in evaluating game states.

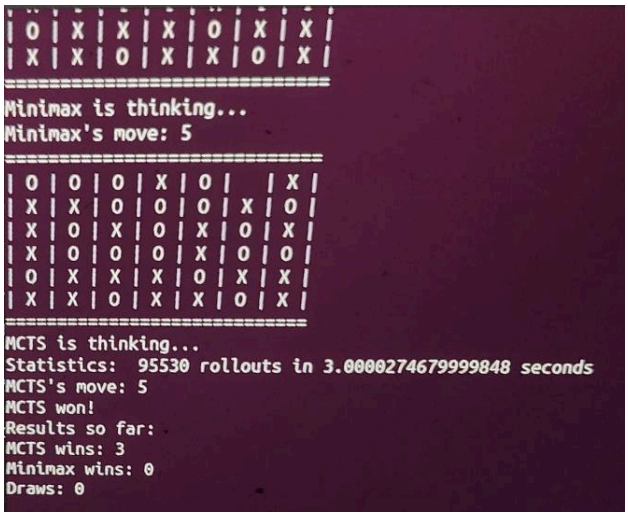

Figure 41 Minimax playing well against MCTS in text-based editor

However, I validated the Minimax algorithm by adjusting the depth parameter and properly integrating Alpha-Beta pruning, which I tested in a text-based editor that I have shown in the Figure 41 above and Chapter 4.

3.4 Google's TRC Program

Thank you to Google's TRC Program for allowing me access to their v4 and v2-8 TPUs for free, which allowed me to accelerate my deep-learning research and train my model for the extremely high number of games and training Epochs that would not have been possible on my local machine.

[TRC] You have access to free Cloud TPUs Inbox ×

TRC-Support <trc-support@google.com> Fri, 5 Apr, 04:17
to me

TPU Research Cloud

Hi Rohan,

Thanks again for your interest in using Cloud TPUs to accelerate your machine learning research. Your Google Cloud project mcts-research-project now has access to the following quota free of charge for 30 days:

- **32 on-demand Cloud TPU v4 chips in zone us-central2-b**
- **32 preemptible Cloud TPU v4 chips in zone us-central2-b**
- **50 preemptible Cloud TPU v2-8 device(s) in zone us-central1-f**
- **50 preemptible Cloud TPU v3-8 device(s) in zone europe-west4-a**

4 Empirical Studies

Metrics

When conducting these experiments where we benchmark the Monte Carlo Tree Search algorithm against the other models, two critical parameters were meticulously maintained throughout the gameplay setup. Firstly, the MCTS algorithm was deliberately given the fixed time odds of ***1 second per move***. By doing this we limit the number of rollouts that it could perform while playing a single move, generating an average of only 30,000 rollouts. This makes it comparatively weaker. Secondly, MCTS was given the role of the second player across all experiments (Readers Digest, 2023).

The reason behind this decision was that. by positioning the MCTS algorithm as the second player in all matches, the aim was to diminish any inherent advantage or edge that might bias the results. In the game of Connect-4, with perfect play, the first player always has an advantage. This is because at the time of initiating the game if they choose the middle column, they can be guaranteed a win. So, by maintaining the MCTS algorithm as the second player, any advantage favouring the first player was effectively eliminated. This ensured the evaluation of the algorithm's performance was fair and unbiased.

In the game of Connect-4 with perfect play, the result will always be a first-player win. However, this only happens if the first player can force a win by the 41st move, under the condition that they choose the middle column. This is because when choosing the middle column, the player can connect 4 in all possible directions, thus giving the player 1 the geometrical advantage (BoardGameGeek, n.d.). However, if player 1 plays another move which is not the middle column then in that case the game is a theoretical draw.

Keeping these parameters in place, the following experiments were conducted that could successfully answer our research question i.e. how well does MCTS benchmark against Q-Learning, Minimax and do Neural Networks improve the performance of the agent or not (www.datacamp.com, n.d.).

4.2 Experiment 1: Minimax vs Q-Learning

Aim: So, before I dove right into pitting my MCTS algorithm against the other agents, I wanted to ensure that both the Q-Learning agent's and the Minimax algorithm's performance strengths had been validated. To ensure that both agents were performing well, I set certain hyperparameters in place that would ensure competitive performance during the experiments.

In the case of the Q-Learning agent, I trained the agent in a series of self-play games, where the agent trained by playing 3 million games against itself. For the hyperparameters, I set the ***learning rate = 0.01, discount-factor=0.99, epsilon=0.1*** and ***decay-rate=0.9995.***

```
class QLearningAgent:
    def __init__(self, learning_rate=0.01, discount_factor=0.99, epsilon=0.1, decay_rate=0.9995):
        self.q_table = {}
        self.learning_rate = learning_rate
        self.discount_factor = discount_factor
        self.epsilon = epsilon
        self.decay_rate = decay_rate
```

Figure 42 Q-Learning agent

It took nearly an hour to train the agent over 3 million games of self-play, after which I followed it up by setting the experiment of pitting the Q-Learning agent against the minimax algorithm. In the case of the Minimax algorithm, I had set the ***depth to 4***, with which it took about 1-2 seconds per move.

```
def play_minimax_vs_qlearning(minimax_player, qlearning_player, num_games):
    results = {"Minimax Wins": 0, "Minimax Losses": 0, "Minimax Draws": 0,
               "Q-learning Wins": 0, "Q-learning Losses": 0, "Q-learning Draws": 0}
    for _ in range(num_games):
        game = ConnectFour()
```

Figure 43 Minimax vs Q-Learning agent

Observation: After experimenting with 100 games between the Minimax agent and the Q-learning agent, the Minimax agent out-performed the Q-Learning agent with a score of 73 wins, 23 losses and 4 draws. Even though the Q-Learning agent performed competitively after being trained on 3 million games, and the Minimax agent was set to a low depth of 4, the Minimax agent still proved to be the better choice between the two.

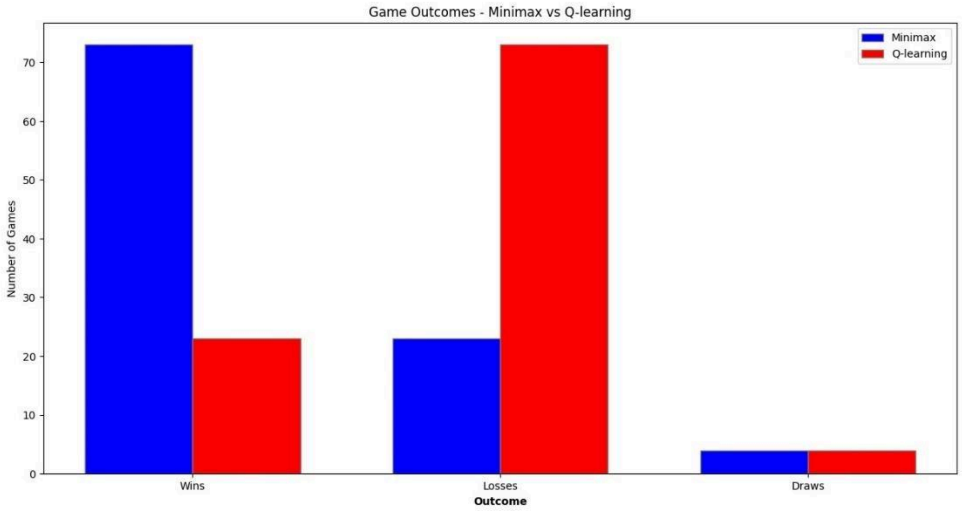

Figure 44 Minimax victorious against Q-Learning

Conclusion: There are 2 conclusions we can draw from the above experiment. Firstly, that both the Q-Learning agent performed decently and the Minimax agent performed competitively with Q-Learning scoring 23 wins and Minimax scoring 73 wins, and they had a total of 4 draws. Hence both can be used as a valid benchmark for conducting future experiments to evaluate other agents' and algorithms' performances. Secondly, through this experiment, we concluded that the Minimax agent was the stronger of the 2 agents despite Q-Learning training rigorously for 3 million games of self-play. With this kept in mind, we can proceed to pit these 2 algorithms against MCTS.

4.3 Experiment 2: MCTS v Q Learning

Aim: In Experiment 1, we saw that Q-Learning after sufficient training was able to make some progress in terms of winning. Even though it did not display strong results against Minimax, it still managed to win 23 games after being trained over 3 million games, when Minimax was set to a low depth of 4. This, however, gives us a good benchmark to check whether the MCTS algorithm itself performs well or not. To set up this experiment, the Q-Learning agent was again trained over 3 million games on the same hyperparameters as in Experiment-1, whereas the MCTS algorithm was only given 1 second per move, which generated between 10,000 and 11,000 rollouts per move, which would result in a relatively weaker performance. It is also important to note that in this experiment, the Q-Learning agent was set as player 1, whereas the MCTS algorithm was set as player 2. As mentioned in Section 4.1, this would give the advantage to Q-Learning, which takes any bias away from the MCTS algorithm. So, with these parameters set, I experimented with 100 games between the MCTS and Q-Learning and the result is as follows.

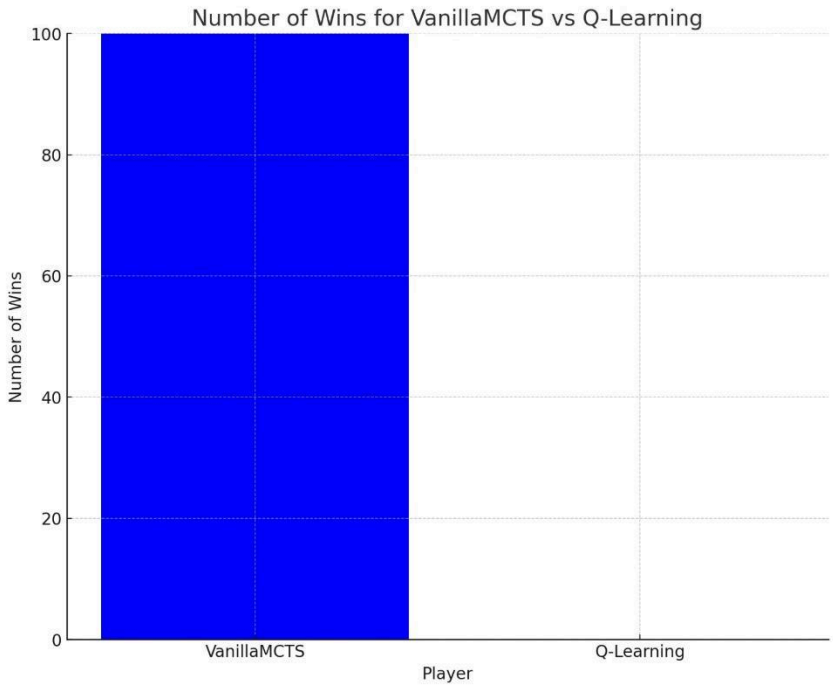

Figure 45 MCTS victorious against Q-Learning

Observation: In this match of 100 games between Q-Learning and the MCTS algorithm, despite the Q-Learning being trained over 3 million games and being the first player, Q-Learning performed poorly losing all 100 games. It is important to note that the MCTS algorithm only had 1 second think-time per move which results in a relatively weaker performance.

Result: Although Q-Learning is a promising reinforcement learning technique, that learns optimal policies over time, other factors affect the training of the agent and thus its performance such as the algorithmic design and hyperparametric tuning. In this case, Q-Learning was not so effective at capturing the intricate strategies and optimal solutions in the game of Connect-4. MCTS however, emerged as the dominant and superior algorithm, owing its credit to its extensive exploratory approach while also choosing the optimal solutions by maintaining a perfect balance between exploration and exploitation using the UCB algorithm. Given Connect-4's search space, the think-time per move allotted to MCTS was only 1 second, and it still managed to find the optimal solution in every game.

4.4 Experiment 2: Vanilla MCTS vs Minimax

Aim: In this experiment, now that we know from Experiments 1 and 2, that both the Minimax algorithm and the MCTS algorithm played well, we can further evaluate the effectiveness of the MCTS algorithm's performance by pitting it against the Minimax. Once again there were certain parameters that I set for this experiment. Firstly, the Minimax agent was set to a ***depth of 4***, like in Experiment 2. It was also set as player-1 giving it the advantage. Secondly, the MCTS agent was given ***3 seconds per move***, which on average generated about ***95,000 rollouts*** per move, shown in the figure below. After a series of 100 games between the 2 algorithms, the results were as follows.

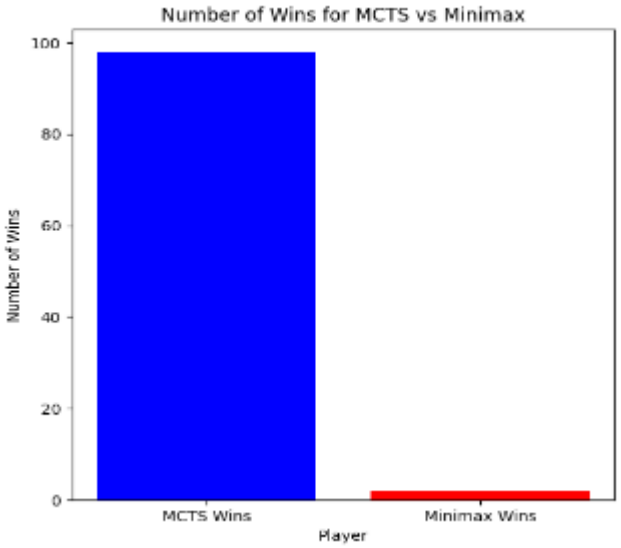

Figure 46 MCTS victorious against Minimax

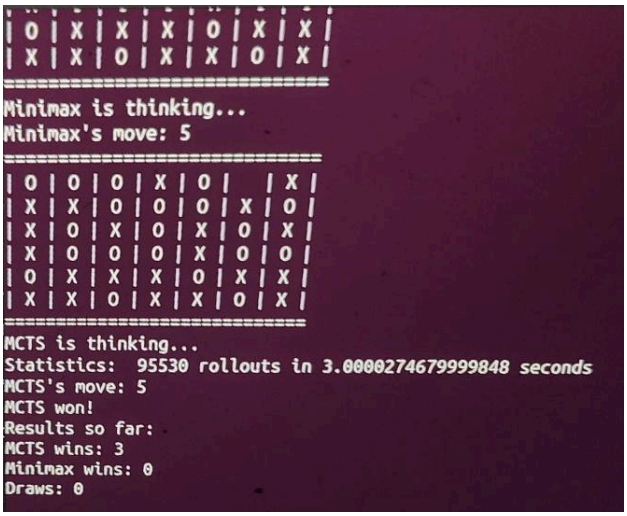

Figure 47 Game Board of Minimax vs MCTS with MCTS win

Observation: In the first match between the Minimax and MCTS, which consisted of 100 games, the match resulted in the favor of MCTS with 98 wins and 2 losses. Despite the apparent disadvantage conferred upon MCTS, it dominated with high accuracy and consistency.

Experiment Part 2: However, to further validate this experiment and to reach a conclusive result, I reset the experiment with different parameters. Once again Minimax was set as player 1, giving it the first-player advantage, however, its search depth was set to ***6***, providing approximately 12 to 15 seconds of think-time per move. The MCTS however was only given 1 second of think-time which would result in relatively poor performance as it only generated about 30,000 rollouts per move.

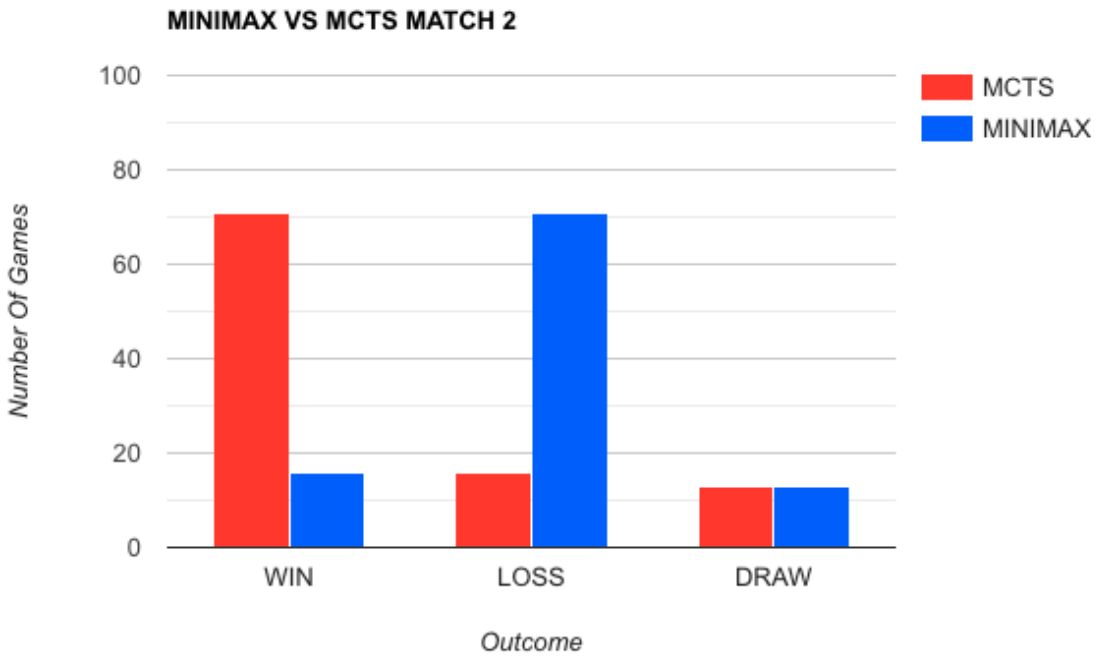

Figure 48 Rematch with Minimax at depth 6 still resulting in loss and win for MCTS

In this experiment, the MCTS agent still managed to be victorious with the result of 71 wins, 16 losses and 13 draws. Although the Minimax agent performed relatively better, MCTS still managed to dominate and win the majority of the games.

Results: Even though Minimax theoretically is one of the best algorithms due to its approach of solving the game tree entirely, by maximizing its chances in all positions, and minimizing the opponent's chances up to a certain depth, it is still however computationally very expensive as it would take extremely long, and require the use of supercomputers, to solve the entire game tree to reach the perfect solution every time. MCTS however, even with its limited time constraint can come up with strong moves, by exploring diverse moves and exploiting the moves with higher rewards effectively. Minimax with its depth-limited search performed poorly against the MCTS agent.

4.5 Experiment-4: MCTS v AlphaMCTS

Aim: The paramount experiment of this research project was to not only see the effectiveness of the MCTS algorithm in the game of connect-4 but to refine it further for improved performance. As mentioned in Chapter 3, the entire point of developing the AlphaMCTS model, which was inspired by DeepMind's AlphaZero model was, to integrate a neural network that used the MCTS algorithm to refine its search (Google DeepMind, 2018). By using the improved PUCT algorithm as opposed to the standard UCT algorithm, it was able to use the rollouts as the target, which allowed it to exploit the previously recorded winning moves in future games while using additional hyper-parameters. The importance of this experiment is not only to test the capability of the AlphaMCTS model against a vanilla MCTS algorithm but to prove that if it shows improvement over time, this approach can be highly effective in games like Chess and Go as well, where won't be any random rollouts, instead previous rewarding moves will be recorded and exploited and newer moves that haven't been played previously will be explored more. So, to set up this experiment I put in place specific hyperparameters for the training of the AlphaMCTS model.

The ***'num_filters': 128*** specifies the number of convolutional filters used in residual blocks in the ResNet architecture.

The ***'num_residual_blocks': 8*** is the number of residual blocks used.

The ***'exploration_constant'*** is set to ***2*** in the PUCT calculation.

The ***'selection_temperature': 1.25*** will ensure a more uniform distribution.

The most important hyperparameter inspired by the AlphaZero paper is ***'dirichlet_alpha': 1*** which is a parameter for Dirichlet noise that encourages exploration of more diverse moves. Larger values mean more uniform noise.

The ***'dirichlet_epsilon': 0.25*** specifies the weight of Dirichlet noise.

The ***'learning_rate'*** was set to ***0.001*** using Adam as the optimizer.

The ***'training_epochs':*** initially were only ***10*** whereas the ***'games_per_epoch':*** was set to ***100***.

The ***'minibatch_size'*** was ***128***, which is used in learning update.

'num_minibatches': 4 would set 4 minibatches to accumulate per learning step.

The ***'mcts_initial_iterations':*** were set to ***50***, which increased by 1 after every epoch.

The ***'mcts_max_iterations': 150*** sets the maximum number of MCTS iterations.

mcts_search_increment': 1, ensure that after each epoch, the search iterations should be increased by 1.

It is important to note that in this experiment, AlphaMCTS was player 2 and Vanilla MCTS was player 1 taking away the first-player advantage from the AlphaMCTS model.

Observation: In a series of 100 matches, where the AlphaMCTS model was only trained for 10 epochs, i.e., 1000 games of self-play, the Vanilla MCTS algorithm emerged victorious,

with a record of 66 wins, and 34 losses. The Vanilla MCTS algorithm was given 5 seconds per move this time which increased the average number of rollouts to 150,000 per move.

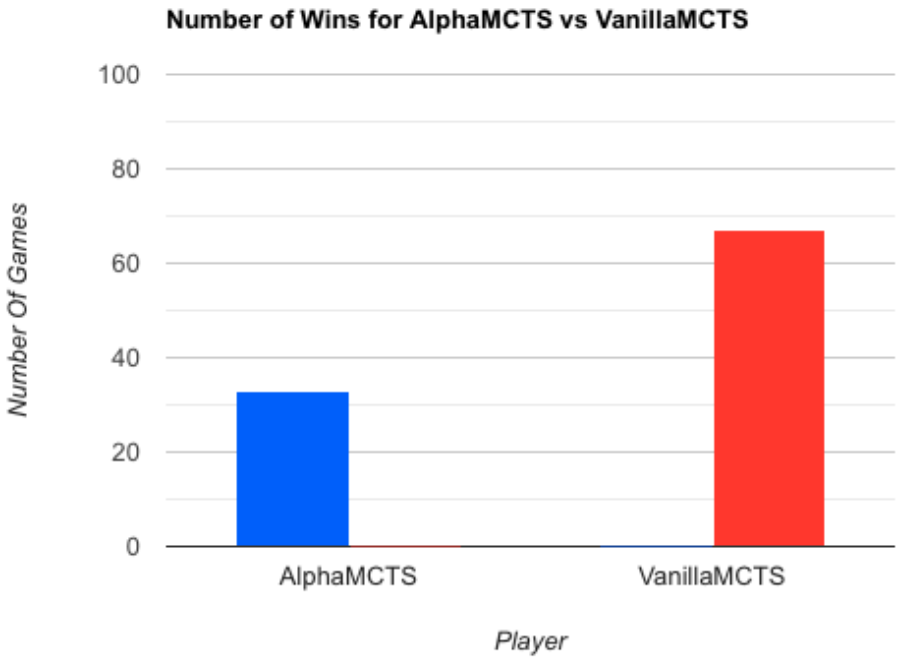

Figure 49 AlphaMCTS loses with against Vanilla MCTS with 10 Epochs training

Experiment part-2: So, I followed this up by retraining the model, only this time training it for ***50 Epochs, i.e. 5000 games.*** This not only meant more games but deeper training as well, as a result of iterative deepening after every epoch. Once the training was completed, I loaded the uploaded neural-network-weights file into the model and conducted another match of 100 games with AlphaMCTS being player 2 and Vanilla MCTS being player 1 and getting 5 seconds of think-time per move.

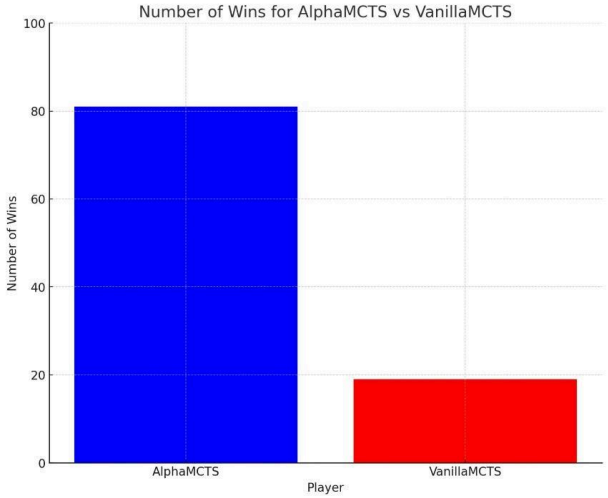

Figure 50 AlphaMCTS wins with against Vanilla MCTS with 50 Epochs training

Results:

AlphaMCTS emerged victorious with a match score of 81 wins and 19 losses. Despite there being computational constraints due to which I was only able to train my model for 5000 games (as opposed to DeepMind's AlphaZero which was trained for 44 million games), it was evident that AlphaMCTS adapted in optimizing its performance with the available training parameters. This indicates that the AlphaMCTS model with its hybrid approach of the Residual Neural Network that employs the Monte Carlo Tree Search to refine its strategy showed positive results. With access to much stronger supercomputers, and deep-learning-specific scalable machines, we can train this model at much higher depths, for over millions of games to witness its full capabilities. This result highlights the convergence of traditional search algorithms, pointing us towards better state-of-the-art machine learning methodologies with hybrid approaches to develop better strategies and methodologies in complex game-play scenarios.

4.6 Summary

The experiments I conducted in the game of Connect-4 gave us great insights into the efficiency and effectiveness of different models and algorithms under different parameters. In the case of the Minimax algorithm, even though it showed high accuracy, due to its high computational demands, and struggled in terms of scalability, due to which it was ineffective against the MCTS algorithm.

The Q-Learning agent, while theoretically an appealing choice for a research study of this nature, faced significant challenges in exploring the vast and complex game space of Connect-4 even after sufficient training which resulted in substandard moves. MCTS algorithm however was the dominant of the two. Its ability to explore diverse moves across the game tree, while choosing the better moves yielding higher rewards in the minimum amount of time proved its superiority.

MCTS with its stochastic nature, achieved this by making a trade between being computationally less demanding by leveraging leverage random simulations, which empowered it to discover novel solutions and adapt effectively. This, however, could also lead to sub-optimal choices in certain cases, as choosing moves based on random simulations may not always result in the optimal policy, specifically in games where the state-space is exponentially higher like in games like Chess and Go.

To resolve this issue, we followed a strategy akin to that of Google's DeepMind when they developed the AI agents AlphaZero and AlphaGo, where they integrated a Residual Neural Network that used the rollouts as the target for the policy and refined its gameplay over time by exploiting the winning moves and further exploring non-played moves. With this approach, I was able to develop the AlphaMCTS model that showed excellent results while only being trained for 5000 games of self-play, by leveraging the predictive nature of state space networks. This allowed AlphaMCTS to demonstrate superior performance and opened new doors to an era of hybrid algorithms in game theory and artificial intelligence.

5 Discussion and Conclusion

Undertaking this research project on Monte Carlo Tree Search(MCTS) for the game of Connect-4 was a challenge as it required me to research and study beyond our curriculum in the module of Neural Computing. However, due to my interest in and knowledge of the world of Chess, I was already aware of the MCTS algorithm and its success in board games. This was an additional motivating factor for me to take this project on, and immerse myself into the realm of advanced game-playing algorithms thus enriching my knowledge and passion of Artificial Intelligence. By taking on this topic, I also learned a great detail about Reinforcement Learning, its mechanisms, concepts and applications that contributed towards the overall success of this project.

By taking on this project, I'm extremely happy to say that I have contributed a great deal towards the constantly growing and ever-evolving world of Artificial Intelligence and Machine Learning. By conducting the experiments, and simulating game plays under specific parameters, I was able to arrive at different conclusions regarding the performances of different algorithms and why the MCTS algorithm is an excellent and effective choice to develop models and conduct research in games and problem domains with high branching factors.

By developing a model, inspired by AlphaZero, that I titled AlphaMCTS, I opened multiple new doors for myself to dive deep and conduct further research on this topic. By employing the MCTS algorithm as the core search technique and using it to refine the strategy of the Neural Network, I was successfully able to create a model capable of self-learning and improving over time by playing multiple games. This further presents me with opportunities to improve it further by incorporating the Minimax Alpha-Beta algorithm to it to improve its accuracy further and apply it to games with much higher branching factors such as Chess. This project has motivated me to develop a Chess engine by employing this ensemble algorithm and a hybrid approach that could potentially be as effective as modern state-of-the-art chess engines like Stockfish and LeelaChess0.

The road ahead is full of challenges and uncertainties, with many more AI models to develop and train, implying that new technologies and potential, waiting to be untapped. By consistently pursuing and chasing new problems, we will be able to unravel new mysteries and solutions. By doing this, we open the door for other initiatives to take a similar approach, sparking a rebirth in AI research and a new chapter in the history of human-machine cooperation.

In summary, this effort is a monument to the strength of human curiosity, inventiveness, and tenacity rather than only the result of scholastic pursuits. It reminds us of the limitless potential that is within our reach in an ever-evolving world of technological innovation, acting as a beacon of hope. Let us navigate these unknown waters with bravery, determination, and an unwavering dedication to discovering the mysteries of the technological world.

6 Appendix

6.1 MCTS for Chess

Although, due to a lack of Computational resources, which is essential for rigorous training and testing purposes, I was unable to develop a full model capable of playing Chess using MCTS and Neural Network evaluation, however, I started to work on it with a basic implementation of a Chess engine that utilizes the MCTS algorithm to play the game of chess against a human player. To set it up, I used the same MCTS class that I defined in section 3.1 for the game Connect-4, which defines the mechanics and the UCT algorithm. Additionally, I implemented the ***python-chess*** library for visualization, move validation and to check if a state is terminal (packtpub.com, n.d.).

```
import hashlib
import copy

class ChessGame:
    def find_children(self):
        if self.is_terminal():
            return set()
        return {
            self.make_move(m) for m in self.board.legal_moves
        }

    def find_random_child(self):
        if self.is_terminal():
            return None
        moves = list(self.board.legal_moves)
        m = choice(moves)
        return self.make_move(m)

    def player_win(self, turn):
        if self.board.result() == '1-0' and turn:
            return True
        if self.board.result() == '0-1' and not turn:
            return True
        return False
```

Figure 51 ChessGame class

When setting up the game board, we import the chess from the ***python-chess*** library and initiate a simple loop for facilitating the start positions prompting the user to make a move.

```python
from IPython.display import display
import chess
import chess.svg

def play_chess():
    tree = MCTS()
    game = ChessGame(chess.Board())
    display(chess.svg.board(board=game.board, size=300))
    while True:
        move_str = input('enter move: ')
        move = chess.Move.from_uci(move_str)
        if move not in list(game.board.legal_moves):
            raise RuntimeError('Invalid move')
```

Figure 51 play_chess function

Once the move is made, the gameboard appears which displays the move played relative to the position on the board.

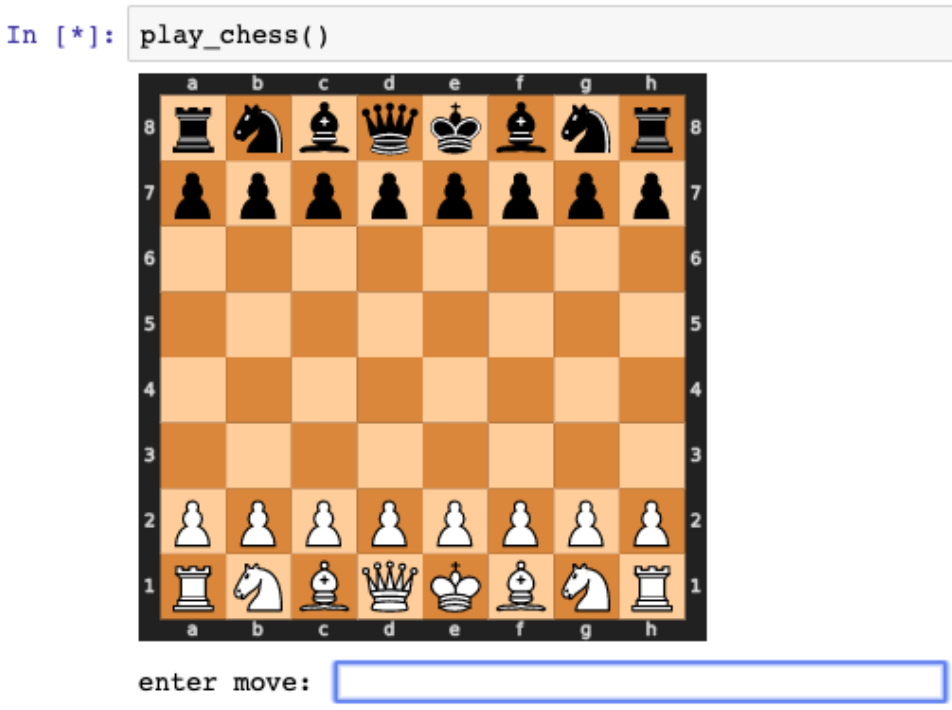

Figure 52 Chess game board with move input prompt

It is important to note that the ***python-chess*** library has UCI move notations integrated, which means it can only accept valid UCI chess moves. For example, to move the pawn in front of the King 2 squares, the user would have to enter ***'a2a4'***.

```
def __hash__(self):
    return int(
        hashlib.md5(
            self.board.fen().encode('utf-8')
        ).hexdigest()[:8],
        16
    )

def __eq__(self, other):
    return self.__hash__() == other.__hash__()

def __repr__(self):
    return '\n' + str(self.board)
```

Figure 53 hashable function to store Q-values

For implementing the nodes that would form the explored path, and would backpropagate the associated rewards I programmed the nodes to be hashable and comparable to store them in dictionaries. So, to achieve this I created ***the __hash__*** and ***__eq__*** methods. The ***__repr__()*** method was implemented for debugging purposes (packtpub.com, n.d.).

The performance of this Chess engine was extremely poor. I played multiple games against it and it resulted in an extremely poor performance with no good moves being played. The core reason that explains this poor performance is the random rollouts being generated in each move. For reference, when I measured the strength of the Chess engine on chess.com, it only reached an ELO rating of 142, which is extremely bad. I am an above average Chess player with an ELO rating of 1400, and a Chess Grandmaster has an ELO rating above 2500. This proved that the Chess engine's performance was below substandard.

Since there is no policy in place that the Chess engine can follow, the process of playing simulations was always random. Where on hand, the same algorithm performs extremely well in the game of Connect-4, which has a state-space of 10^12, in the case of chess, the state-space is 10^120, so choosing random moves will not result in any promising moves. To solve this problem, we need to develop a reinforcement learning mechanism or a neural network policy that will be capable of using the rollouts in the target head of the policy, similar to what I did in the AlphaMCTS model for Connect-4. This is similar to what DeepMind did while developing AlphaZero where they created the model with just the rules of chess and made it play games against itself. They were able to train their model over 44 million games within just 4 hours using supercomputing servers powered by TPUs and GPUs. On an average machine, like the one I used, which is an Intel I5, quadcore, I was only able to train my model over 10000 games which took more than 2 days, so training a Chess engine to show significant results would be extremely computationally expensive. Additionally, the scalar head would need to be further enhanced to store the individual mathematical values of all the Chess pieces. In the case of Connect-4, I set the values as 1 for the player and -1 for the opponent, for evaluating winning and losing positions. However, in Chess, I would have to choose the value of 1 for the Pawn, 3 for the Knight and Bishop, 5 for the Rook and 9 for the Queen and then their negative equivalents for the opponents. Moreover, one would have to create powerful static evaluating functions

that would store the complex winning principles and Checkmating patterns to enhance its strength and performance.

6 References

1. Richard Sutton and Andrew Barto Reinforcement Learning, Second Edition. The MIT Press. 2018

2. Enterprise AI. (n.d.). *What is Q-learning?* [online] Available at: https://www.techtarget.com/searchEnterpriseAI/definition/Q-learning.
3. Van Hasselt, H., Guez, A. and Silver, D. (2016). Deep Reinforcement Learning with Double Q-Learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 30(1). doi:https://doi.org/10.1609/aaai.v30i1.10295.
4. Horcas, J.-M., Galindo, J.A., Heradio, R., Fernandez-Amoros, D. and Benavides, D. (2023). A Monte Carlo tree search conceptual framework for feature model analyses. *Journal of Systems and Software*, 195, p.111551. doi:https://doi.org/10.1016/j.jss.2022.111551.
5. Browne, C.B., Powley, E., Whitehouse, D., Lucas, S.M., Cowling, P.I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S. and Colton, S. (2012). A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games*, [online] 4(1), pp.1–43. doi:https://doi.org/10.1109/tciaig.2012.2186810.
6. Kristopher De Asis, J. Fernando Hernandez-Garcia, Holland, G. and Sutton, R.S. (2017). Multi-step Reinforcement Learning: A Unifying Algorithm. *arXiv (Cornell University)*. doi:https://doi.org/10.48550/arxiv.1703.01327.
7. Agarwal, P. (2020). Game AI: Learning to play Connect 4 using Monte Carlo Tree Search. [online] Medium. Available at: https://pranav-agarwal-2109.medium.com/game-ai-learning-to-play-connect-4-using-monte-carlo-tree-search-f083d7da451e.
8. www.cs.swarthmore.edu. (n.d.). *Monte Carlo Tree Search - About*. [online] Available at: https://www.cs.swarthmore.edu/~mitchell/classes/cs63/f20/reading/mcts.html#:~:text=An%20Upper%20Confidence%20Bounds%20(UCB [Accessed 2 Jan. 2024].
9. Vodopivec, T., Samothrakis, S. and Ster, B. (2017). On Monte Carlo Tree Search and Reinforcement Learning. Journal of Artificial Intelligence Research, 60, pp.881–936. doi:https://doi.org/10.1613/jair.5507.
10. Schuchmann, S. (2019). History of the first AI Winter. [online] Medium. Available at: https://towardsdatascience.com/history-of-the-first-ai-winter-6f8c2186f80b
11. ResearchGate. (n.d.). Figure 1: Timeline of the AI winters. [online] Available at: https://www.researchgate.net/figure/Timeline-of-the-AI-winters_fig1_333039347.

12. Goodrich, J. (2021). How IBM's Deep Blue Beat World Champion Chess Player Garry Kasparov. [online] IEEE Spectrum. Available at: https://spectrum.ieee.org/how-ibms-deep-blue-beat-world-champion-chess-player-garry-kasparov.
13. Yao, D. (2022). 25 years ago today: How Deep Blue vs. Kasparov changed AI forever. [online] AI Business. Available at: https://aibusiness.com/ml/25-years-ago-today-how-deep-blue-vs-kasparov-changed-ai-forever.
14. Demis Hassabis (2016). AlphaGo: using machine learning to master the ancient game of Go. [online] Google. Available at: https://blog.google/technology/ai/alphago-machine-learning-game-go/.
15. Cade Metz (2016). What the AI Behind AlphaGo Can Teach Us About Being Human. [online] WIRED. Available at: https://www.wired.com/2016/05/google-alpha-go-ai/.
16. Świechowski, M., Godlewski, K., Sawicki, B., & Mańdziuk, J. (2022, July 19). Monte Carlo Tree Search: A review of recent modifications and Applications - Artificial Intelligence Review. SpringerLink. https://link.springer.com/article/10.1007/s10462-022-10228-y
17. Sironi, C. F., Liu, J., Perez-Liebana, D., Gaina, R. D., Bravi, I., Lucas, S. M., & Winands, M. H. M. (1970, January 1). *Self-adaptive mcts for general video game playing*. SpringerLink. https://link.springer.com/chapter/10.1007/978-3-319-77538-8_25
18. Robles, D., Rohlfshagen, P., & Lucas, S. M. (n.d.). *[PDF] learning non-random moves for playing othello: Improving Monte ...* Learning non-random moves for playing Othello: Improving Monte Carlo Tree Search. https://www.semanticscholar.org/paper/Learning-non-random-moves-for-playing-Othello%3A-Tree-Robles-Rohlfshagen/d5193e7f7fd724d296e7bffe1139f4cb49893288
19. Foundations of Machine Learning. (n.d.). Available at: https://www.hlevkin.com/hlevkin/45MachineDeepLearning/ML/Foundations_of_Machine_Learning.pdf.
20. MEGAN KU, S. D., DIETER BREHM, . 2020. *Automated Four in a Row*
21. *Explorations in Tree Searches and Game Algorithms* [Online]. Available: https://mcts.netlify.app/ [Accessed].
22. SIVAMAYIL, K., RAJASEKAR, E., ALJAFARI, B., NIKOLOVSKI, S., VAIRAVASUNDARAM, S.

& VAIRAVASUNDARAM, I. 2023. A Systematic Study on Reinforcement Learning Based Applications. *Energies,* 16**,** 1512.

23. Alderton, E., Wopat, E. and Koffman, J. (n.d.). Reinforcement Learning for Connect Four. [online] Available at: https://web.stanford.edu/class/aa228/reports/2019/final106.pdf.

24. Sheoran, K., Dhand, G., Dabasz, M., Dahiya, N. and Pushparaj, P. (2022). SOLVING CONNECT 4 USING OPTIMIZED MINIMAX AND MONTE CARLO TREE

SEARCH. Advances and Applications in Mathematical Sciences, 21(6), pp.3303– 3313.

25. Al Awan, A. ed., (n.d.). [online] DataCamp. Available at: https://www.datacamp.com/tutorial/introduction-q-learning-beginner-tutorial.

26. Trung Luu, Q. (2023). Q-Learning vs. Deep Q-Learning vs. Deep Q-Network. [online] Baeldung. Available at: https://www.baeldung.com/cs/q-learning-vs-deep-q-learning-vs-deep-q-network.

27. Casey, V. (2008). Software Testing and Global Industry. Cambridge Scholars Publishing.

28. kaggle.com. (n.d.). AlphaZero Connect 4 - RL. [online] Available at: https://www.kaggle.com/code/auxeno/alphazero-connect-4-rl/notebook

29. Qi Wang's Blog! (2022). Connect 4 with Monte Carlo Tree Search. [online] Available at: https://www.harrycodes.com/blog/monte-carlo-tree-search

30. Datagen. (n.d.). ResNet: The Basics and 3 ResNet Extensions. [online] Available at: https://datagen.tech/guides/computer-vision/resnet/#.

31. verdantfox.com. (n.d.). How I built a Connect 4 AI | Verdant Fox. [online] Available at: https://verdantfox.com/blog/how-i-built-a-connect-4-ai

32. www.cs.cornell.edu. (n.d.). Untitled Document. [online] Available at: https://www.cs.cornell.edu/boom/2001sp/Anvari/Anvari.htm.

33. Jed (2024). The Rise of Chess AI: From Deep Blue to AlphaZero. [online] Amphy Blog. Available at: https://blog.amphy.com/deep-blue-alphazero/.

34. Built In. (n.d.). Math for AI: A Guide. [online] Available at: https://builtin.com/articles/math-for-ai#:~:text=Linear%20algebra%20is%20the%20field.

35. Chess.com. (n.d.). Chess Insights - Understand and Improve Your Game. [online] Available at: https://www.chess.com/insights/hikaru

36. subscription.packtpub.com. (n.d.). Artificial Intelligence with Python Cookbook. [online] Available at: https://subscription.packtpub.com/book/data/9781789133967/5/ch05lvl1sec31/writing-a-chess-engine-with-monte-carlo-tree-search.

37. Google DeepMind. (2018). AlphaZero: Shedding new light on chess, shogi, and Go. [online] Available at: https://deepmind.google/discover/blog/alphazero-shedding-new-light-on-chess-shogi-and-go/.
38. BoardGameGeek. (n.d.). First player advantage | Earth. [online] Available at: https://boardgamegeek.com/thread/3222954/first-player-advantage
39. papergames.io. (n.d.). Connect 4 Advanced Guide | papergames.io. [online] Available at: https://papergames.io/docs/game-guides/connect4/advanced-guide/
40. Nov. 06, L.C. and 2023 (2023). How to Win Connect 4 Every Time, According to the Computer Scientist Who Solved It. [online] Reader's Digest. Available at: https://www.rd.com/article/how-to-win-connect-4/.
41. Pytorch.org. (2023). PyTorch. [online] Available at: https://pytorch.org/.
42. www.datacamp.com. (n.d.). Matplotlib Tutorial: Python Plotting. [online] Available at: https://www.datacamp.com/tutorial/matplotlib-tutorial-python.