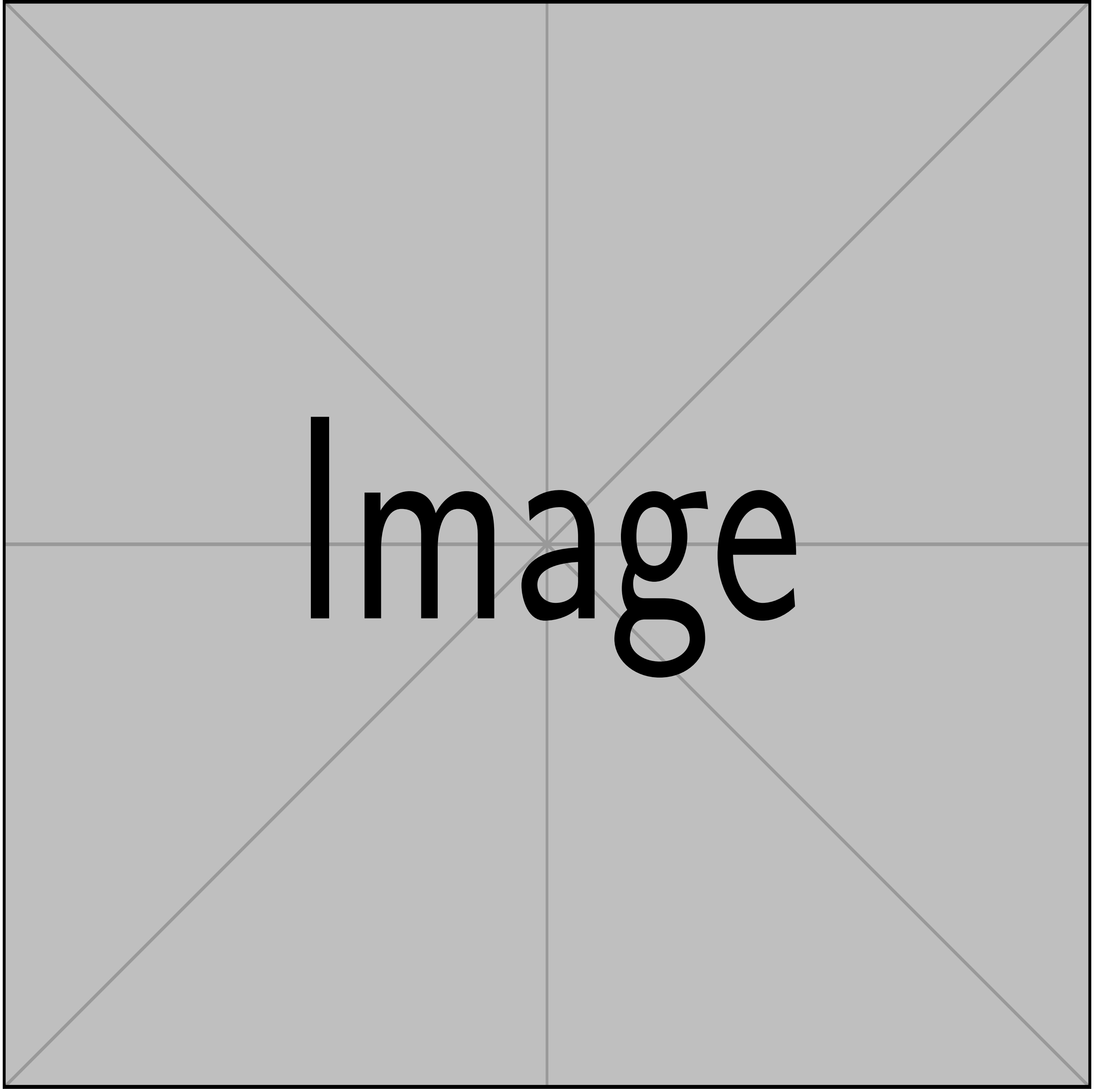

HOW TO CREATE A CHESS ENGINE USING DEEP REINFORCEMENT LEARNING

A CRITICAL LOOK AT DEEPMIND'S ALPHAZERO

INTERNAL PROMOTOR: WOUTER GEVAERT

EXTERNAL PROMOTOR: <NAME HERE>

RESEARCH CONDUCTED BY

TUUR VANHOUTTE

FOR OBTAINING A BACHELOR'S DEGREE IN

MULTIMEDIA & CREATIVE TECHNOLOGIES

HOWEST | 2021-2022

Preface

This bachelor thesis is the conclusion to the bachelor program Multimedia & Creative Technologies at Howest college West Flanders in Kortrijk, Belgium. The program teaches students a wide range of skills in the field of computer science, with a focus on creativity and Internet of Things. From the second year on, students can choose between four different modules:

1. **AI Engineer**
2. **Smart XR Developer**
3. **Next Web Developer**
4. **IoT Infrastructure Engineer**

This bachelor thesis was made under the **AI Engineer** module. The subject of the thesis is a critical look at the result of my research project in the previous semester. The goal of the project was to create a chess engine in Python with deep reinforcement learning based on DeepMind's AlphaZero algorithm.

I will explain the research I needed to create it, the technical details on how to program the chess engine and I will reflect on the results of the project. To do this, I will contact multiple people familiar with the field of reinforcement learning to get a better understanding of the impact of this research on society. Based on this, I will give advice to people and companies who wish to implement similar algorithms.

I would like to show gratitude to Wouter Gevaert for his enthusiastic support in the creation of my research project and this thesis. I also want to thank the other teachers at Howest Kortrijk, who shared their knowledge and expertise in programming and AI in very interesting classes.

Furthermore, I would like to thank my parents for giving me the chance to have a good education, and the motivation to get the best I can out of my studies.

Tuur Vanhoutte, 1st June 2022

Abstract

This bachelor thesis answers the question: "How to create a chess engine using deep reinforcement learning?". It explains the difference between normal chess engines and chess engines that use deep reinforcement learning, and specifically tries to recreate the results of AlphaZero, the chess engine by DeepMind, in Python on consumer hardware.

The technical research shows what is needed to create my implementation using Python and TensorFlow. It shows how to program the chess engine, how to build the neural network, and how to train and evaluate the network. During the creation of this chess engine, it was crucial to create a huge amount of data through self-play.

The thesis contains a reflection on the results of my research project, which proposes a solution to the problem of creating a high amount of games through self-play. It also reflects on the impact of this research on society, and the viability of this type of artificial intelligence in the future. With this comes a section on advice for companies that wish to implement similar algorithms.

Contents

List of figures

List of abbreviations

Glossary

1 Introduction

Chess is not only one of the most popular board games in the world, it is also a breeding ground for complex algorithms and more recently, machine learning. Chess is theoretically a deterministic game: no information is hidden from either player and every position has a calculable set of possible moves. Because the branching factor of chess is about 35-38 moves [1], calculating if a position is winning or losing requires an enormous amount of calculations.

Throughout the entire history of computer science, researchers have continuously tried to find better ways to calculate if a position is winning or losing. The most famous example is the StockFish engine [2], which uses the minimax algorithm with alpha-beta pruning to calculate the best move.

More recently, researchers at Google DeepMind have developed a new algorithm called AlphaZero [3]. This thesis explores the concept of AlphaZero, how to create a chess engine based on it, and the impact of the algorithm on both the world of chess and the rest of society.

Research has been conducted by investigating what is needed to recreate the results of AlphaZero, by programming a simple implementation using Python and TensorFlow. This was done as part of a research project between November 2021 and January 2022. The code was written with lots of trial and error, as DeepMind released very little information about the detailed workings of the algorithm.

2 Research

2.1 What is a chess engine?

According to Wikipedia [4], a chess engine is a computer program that analyzes chess or chess variant positions, and generates a move or list of moves that it regards as strongest. Given any chess position, the engine will estimate the winner of that position based on the strength of the possible future moves up to a certain depth. The strength of a chess engine is determined by the amount of moves, both in depth and breadth, that the engine can calculate.

2.2 How do traditional chess engines work?

Contemporary chess engines, like StockFish [2], use a variant of the minimax algorithm that employs alpha-beta pruning.

2.2.1 The minimax algorithm

The minimax algorithm [5] is a general algorithm usable in many applications, ranging from artificial intelligence to decision theory and game theory. The algorithm tries to minimize the maximum amount of loss. In chess, this means that the engine tries to minimize the possibility for the worst-case scenario: the opponent checkmating the player. For games where the player needs to maximize a score, the algorithm is called maximin: maximizing the minimum gain.

Minimax recursively creates a search tree [6], with chess positions as nodes and chess moves as edges between the nodes. Each node has a value that represents the strength of the position for the current player. At the start of the algorithm, the tree only consists of a root node that represents the current position. It then explores the tree in a depth-first manner by continuously choosing random legal moves, creating nodes and edges in the process.

This means that it will traverse the tree vertically until a certain depth is reached:

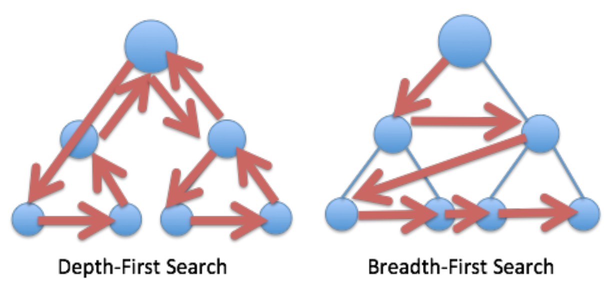

Figure 1: Depth-First search vs Breadth-First search [6]

When that happens, that leaf node's position is evaluated and its value is returned upwards to the parent node. The parent node looks at all of its children's values, and receives the maximum value when playing white, and the minimum value when playing black.

This repeats until the root node receives a value: the strength of the current position.

2.2.2 The evaluation function

The value estimation of leaf nodes is done by an evaluation function [7] written specifically for the game. This function can differ from engine to engine, and is usually written with help from chess grandmasters.

2.2.3 Pseudocode

The algorithm is recursive; it calls itself with different arguments, depending on which player's turn it is. In chess, white wants to maximize the score, and black wants to minimize it. [5]

```
function  minimax(node, depth, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := - inf
        for each child of node do
            value := max(value, minimax(child, depth - 1, FALSE))
        return value
    else (* minimizing player *)
        value := + inf
        for each child of node do
            value := min(value, minimax(child, depth - 1, TRUE))
        return value
```

Calling the function:

```
// origin = node to start
// depth = depth limit
// maximizingPlayer = TRUE if white, FALSE if black
minimax(origin, depth, TRUE)
```

2.2.4 Alpha-beta pruning

Because the necessary amount of nodes to get a good estimation of the strength of a position is so high, the algorithm needs to be optimized. Alpha-beta pruning [8] aims to reduce the amount of nodes that need to be explored by minimax. It does this by cutting off branches in the search tree that lead to worse outcomes.

Say you're playing the white pieces. You want to minimize your maximum loss, which means you want to make sure that black's score is as low as possible. Minimax always assumes that the opponent will play the best possible move. If one of white's possible moves leads to a position where black gets a big advantage, it will eliminate that branch of the search tree. As a result, the amount of nodes to explore is greatly reduced, while retaining a good estimation of the strength of the position.

2.3 Monte Carlo Tree Search

The biggest problem with minimax algorithms that use a depth limit is the dependency on the evaluation function. If the evaluation function makes incorrect or suboptimal estimations, the algorithm will suggest bad moves. Developers of contemporary chess engines like StockFish continuously try to improve this function. Since 2020, StockFish has been using a sparse and shallow neural network as its evaluation function. This neural network is still trained using supervised learning, not (deep) reinforcement learning.

Using alpha-beta pruning can also bring about some problems. Say the player can sacrifice a piece to get a huge advantage later in the game. The algorithm might cut off the branch and never explore that winning line, because it considers the sacrifice a losing position [9].

Monte Carlo Tree Search (MCTS) [10] is a search algorithm that can be used to mitigate these problems. MCTS approximates the value of a position by creating a search tree using random exploration of the most promising moves.

2.3.1 The algorithm

To create this search tree for a certain position, MCTS will run the following algorithm hundreds of times. Each of these runs is called an MCTS **simulation**. One simulation consists of four steps:

1. **Selection**:
   - Starting from the root node, select a child node based on a formula of your choice.
   - Most implementations of MCTS use some variant of Upper Confidence Bound (UCB) [11]
   - Keep selecting nodes until a node has been reached that has not been visited (=expanded) before. We call this a leaf node.
   - If the root node is a leaf node, we immediately proceed to the next step.
2. **Expansion**:
   - If the selected leaf node is a terminal node (the game ends), proceed to the backpropagation step.
   - If it doesn't, create a child node for every possible action that can be taken from the selected node.
3. **Simulation**:
   - Choose a random child node that was expanded in the previous step.
   - By only choosing random moves, simulate the rest of the game from that child node's position.
4. **Backpropagation**:
   - Return the simulation's result up the tree.
   - Every node tracks the number of times it has been visited, and the number of times it has lead to a win.

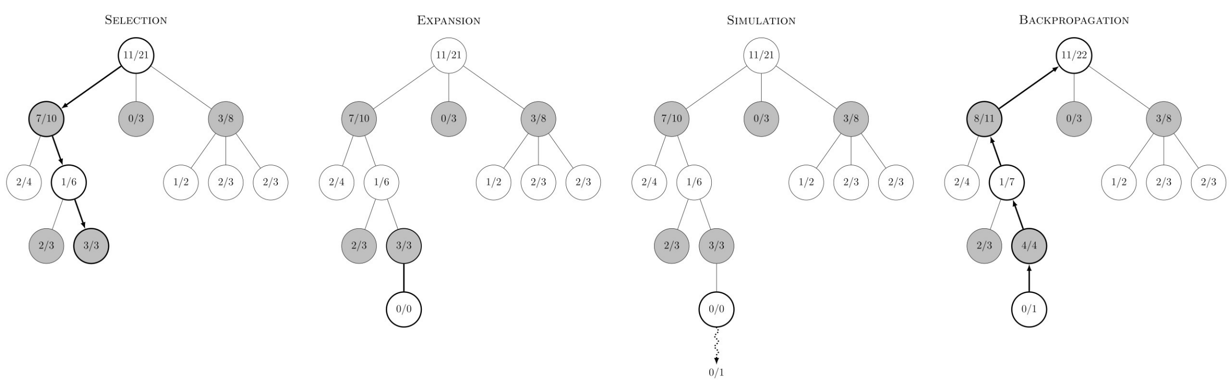

Figure 2: The 4 steps of the MCTS algorithm [10]

For chess, this algorithm is very inefficient, because of its necessity to simulate an entire game of chess in the third step of every simulation. Just to calculate the value of one position would need hundreds of these simulations to get a good estimation. This is why the selection formula needs to be chosen carefully; it's important to select nodes in a way that balances exploration and exploitation.

2.4 Go

Go is a Chinese two-player strategy board game that uses white and black stones as playing pieces [12]. It is played on a rectangular grid of (usually) 19 by 19 lines. The rules are relatively simple, but due to its extremely complex possibilities, Go has been a very popular playground for AI research similarly to chess. Go's much larger branching factor compared to chess makes it very difficult to evaluate a position using traditional methods like minimax with alpha-beta pruning.

Figure 3: Go board [12]

2.4.1 AlphaGo

In 2014, DeepMind Technologies [13], a subsidiary of Google, started developing a new algorithm called AlphaGo to play Go [14]. Previously, the strongest Go engines were still only good enough to win against amateur Go players [15]. The algorithm used a combination of the MCTS algorithm and a deep neural network to evaluate positions.

AlphaGo was built [15] [16] by first training a neural network with supervised learning by using data from human games. The weights of that network were then copied to a new reinforcement learning network. That network was used to create a training set through self-play. By playing against itself and recording the board state, the moves the network considered, and the eventual winner of the game, a training set was created. That training set was then used to train the reinforcement learning network. A separate network (the value network) was used to estimate the value of a position.

2.4.2 AlphaGo Zero

Because AlphaGo still used some amateur games to learn from, the next step was creating a version of AlphaGo that learns completely from scratch. That's why DeepMind developed AlphaGo Zero. AlphaGo Zero uses a different kind of network than AlphaGo. Instead of using two separate networks, it will combine the two outputs into a policy head and a value head. It's also using different layers: residual layers instead of convolutional layers [17].

2.5 AlphaZero

AlphaZero is a generalized version of AlphaGo Zero, created to master the games of chess, shogi ("Japanese chess"), and Go [3] [18]. For chess, AlphaZero was evaluated against StockFish version 8 by playing 1000 games with 3 hours per player, plus 15 seconds per move. It won 155 times, lost 6 times and the remaining games were drawn. AlphaZero uses a single neural network with two outputs, just like AlphaGo Zero.

2.5.1 Neural network input

The input to the network represents the state of the game. It has the following shape: $N \cdot N \cdot (M \cdot T + L)$:

- N is the board size
  - $N = 8$ in chess.
- M is the number of different pieces on the board,
  - Two players with six types of pieces each
  - Every piece is represented by its own 8x8 board of boolean values
  - For every square: 1 if the piece is on that square, 0 if it isn't
  - $M = 12$ in chess
- T is the amount of previous moves that are used as input, including the current move.
  - AlphaZero used $T = 8$ for both chess, shogi, and Go.
  - This gives the network a certain history to learn from
- L represents a set of rules specific to the game
  - $L = 7$ in chess
  - 1 plane to indicate whose turn it is
  - 1 for the total amount of moves played so far
  - 4 for castling legality (white and black can both castle kingside or queenside under certain conditions)
  - 1 to represent a repetition count (in chess, 3 repetitions results in a draw)
- $\Rightarrow 8 \cdot 8 \cdot (12 \cdot 8 + 7)$.

This means that the input to the neural network is 119 8x8 boards of boolean values.

2.5.2 Neural network layers

DeepMind tested multiple neural network architectures for AlphaGo Zero [19]. The following parts were used in these networks:

- Convolutional block
  - A convolution layer
  - Batch normalization layer
  - ReLu activation function
- Residual block
  - This consists of two convolutional blocks, and a skip-connection
  - The skip-connection will combine the input of the block to the output of the first two convolutional blocks

These networks were tested by DeepMind during development of AlphaGo Zero [17]:

1. **'dual-res':** a single tower of 20 residual blocks with combined policy and value heads. This is the architecture used in AlphaGo Zero.
2. **'sep-res':** two towers of 20 residual blocks each: one with the policy head and one with the value head.

3. **'dual-conv':** a single tower of 12 convolutional blocks with combined policy and value heads.
4. **'sep-conv':** two towers of 12 convolutional blocks each: one with the policy head and one with the value head. This is the network used in AlphaGo.

AlphaZero uses the same network architecture as AlphaGo Zero: dual-res.

2.5.3 Neural network output

The neural network has two outputs:

- A policy head, which represents a probability distribution over the possible actions.
- A value head, which represents the value of the current position.

While the value head simply outputs a single float value between -1 and 1, the policy head is quite a bit more complicated. It outputs a vector of probabilities, one for each possible action in the chosen game. For chess, 73 different types of actions are possible:

- 56 possible types of "queen-like" moves: 8 directions to move the piece a distance between 1 and 7 squares.
- 8 possible knight moves
- 9 special "underpromotion" moves:
  - If a pawn is promoted to a queen, it is counted as a queen-like move (see above)
  - If a pawn is promoted to a rook, bishop, or knight, it is seen as an underpromotion (3 pieces)
  - 3 ways to promote: pushing the pawn up to the final rank, or diagonally taking a piece and landing on the final rank
  - $\Rightarrow 3 \cdot 3 = 9$

These 73 actions are each represented by a plane of 8x8 float values. Say the first plane is a queen-like move to move a piece one square northwest, the second plane could the same type of move, but a distance of two squares, and so on. The squares on these planes represent the square from which to pick up a piece.

The result is a $73 \cdot 8 \cdot 8$ vector of probabilities, so 4672 float values.

2.6 Tensor Processing Units

To help with calculations, DeepMind used Google's newly created Tensor Processing Units (TPU) [20]. A TPU is an application-specific integrated circuit (ASIC [21]) that is specifically built for machine learning with neural networks.

2.7 Leela Chess Zero

3 Technical research

4 Reflection

5 Advice

6 Conclusion

7 Bibliography

[1] *Branching Factor - Chessprogramming wiki*. [Online]. Available: `https://www.chessprogramming.org/Branching_Factor` (visited on 03/28/2022).

[2] "Stockfish (chess)," *Wikipedia*, Mar. 2022. [Online]. Available: `https://en.wikipedia.org/w/index.php?title=Stockfish_(chess)&oldid=1079146589` (visited on 03/28/2022).

[3] "AlphaZero," *Wikipedia*, Jan. 2022. [Online]. Available: `https://en.wikipedia.org/w/index.php?title=AlphaZero&oldid=1065791194` (visited on 02/01/2022).

[4] "Chess engine," *Wikipedia*, Apr. 2022. [Online]. Available: `https://en.wikipedia.org/w/index.php?title=Chess_engine&oldid=1080874516` (visited on 04/05/2022).

[5] "Minimax," *Wikipedia*, Mar. 2022. [Online]. Available: `https://en.wikipedia.org/w/index.php?title=Minimax&oldid=1076761456` (visited on 04/05/2022).

[6] M. Eppes, *How a Computerized Chess Opponent "Thinks" — The Minimax Algorithm*, Oct. 2019. [Online]. Available: `https://towardsdatascience.com/how-a-chess-playing-computer-thinks-about-its-next-move-8f028bd0e7b1` (visited on 04/05/2022).

[7] "Evaluation function," *Wikipedia*, Mar. 2022. [Online]. Available: `https://en.wikipedia.org/w/index.php?title=Evaluation_function&oldid=1079533564` (visited on 04/06/2022).

[8] "Alpha–beta pruning," *Wikipedia*, Jan. 2022. [Online]. Available: `https://en.wikipedia.org/w/index.php?title=Alpha%E2%80%93beta_pruning&oldid=1068746141` (visited on 02/01/2022).

[9] *Minimax and Monte Carlo Tree Search - Philipp Muens*. [Online]. Available: `https://philippmuens.com/minimax-and-mcts` (visited on 04/06/2022).

[10] "Monte Carlo tree search," *Wikipedia*, Apr. 2022. [Online]. Available: `https://en.wikipedia.org/w/index.php?title=Monte_Carlo_tree_search&oldid=1081107255` (visited on 04/06/2022).

[11] *ML | Monte Carlo Tree Search (MCTS)*, Jan. 2019. [Online]. Available: `https://www.geeksforgeeks.org/ml-monte-carlo-tree-search-mcts/` (visited on 04/07/2022).

[12] "Go (game)," *Wikipedia*, Mar. 2022. [Online]. Available: `https://en.wikipedia.org/w/index.php?title=Go_(game)&oldid=1079941654` (visited on 04/06/2022).

[13] "DeepMind," *Wikipedia*, Feb. 2022. [Online]. Available: `https://en.wikipedia.org/w/index.php?title=DeepMind&oldid=1072182749` (visited on 04/07/2022).

[14] "AlphaGo," *Wikipedia*, Mar. 2022. [Online]. Available: `https://en.wikipedia.org/w/index.php?title=AlphaGo&oldid=1077595428` (visited on 04/07/2022).

[15] *AlphaGo*. [Online]. Available: `https://www.deepmind.com/research/highlighted-research/alphago` (visited on 04/07/2022).

[16] *Mastering the game of Go with Deep Neural Networks & Tree Search*. [Online]. Available: `https://www.deepmind.com/publications/mastering-the-game-of-go-with-deep-neural-networks-tree-search` (visited on 04/07/2022).

[17] *Mastering the game of Go without Human Knowledge*. [Online]. Available: `https://www.deepmind.com/publications/mastering-the-game-of-go-without-human-knowledge` (visited on 04/07/2022).

[18] D. Silver, T. Hubert, J. Schrittwieser, *et al.*, "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm," *arXiv:1712.01815 [cs]*, Dec. 2017. arXiv: `1712.01815 [cs]`. [Online]. Available: `http://arxiv.org/abs/1712.01815` (visited on 04/07/2022).

[19] *Neural Networks - Chessprogramming wiki*. [Online]. Available: `https://www.chessprogramming.org/Neural_Networks#ANNs` (visited on 04/07/2022).

[20] "Tensor Processing Unit," *Wikipedia*, Mar. 2022. [Online]. Available: `https://en.wikipedia.org/w/index.php?title=Tensor_Processing_Unit&oldid=1077688658` (visited on 04/07/2022).

[21] "Application-specific integrated circuit," *Wikipedia*, Feb. 2022. [Online]. Available: `https://en.wikipedia.org/w/index.php?title=Application-specific_integrated_circuit&oldid=1074023806` (visited on 04/07/2022).

8 Appendix