FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of bachelor's thesis

Title:	Crosphere
Student:	Daniel Ježek
Supervisor:	Ing. Magda Friedjungová, Ph.D.
Study program:	Informatics
Branch / specialization:	Artificial Intelligence 2021
Department:	Department of Applied Mathematics
Validity:	until the end of summer semester 2024/2025

Instructions

The aim of the thesis is to design and implement an abstract strategic board game called Crosphere. In this deterministic and zero-sum game, two players strategically place stones on the board to connect opposite edges. The details of the thesis are specified in the following steps:
1) Survey common and state-of-the-art approaches to two-player board games with zero-sum (e.g., Go, Chess, Quoridor). Also, focus on methods based on neural networks and reinforcement learning (e.g. [1, 2]).
2) Prepare appropriate data (e.g., using self-play as used in AlphaGo) and design and implement your own board game, including similar principles as Go, Chess, and Quoridor. The proposed game will include the following:
  a) There is one agent. This agent plays against a user.
  b) The user can set the difficulty level of the mentioned agent.
3) During the implementation, consider methods reviewed in Step 1 and implement at least three of them, including heuristic and reinforcement learning-based approaches. Experimentally compare these methods, discuss the results, and choose the best one for the final solution.
4) Design and implement a mobile app that allows the user to play the game. The app can be in the form of a prototype without emphasis on graphics and UX/UI.

[1] https://link.springer.com/chapter/10.1007/978-3-030-14174-5_16
[2] http://students.ceid.upatras.gr/~kanellop/pubs/Aig08.pdf

Electronically approved by Ing. Karel Klouda, Ph.D. on 8 January 2024 in Prague.

Bachelor’s thesis

CROSPHERE

Daniel Ježek

Faculty of Information Technology
Department of Applied Mathematics
Supervisor: Ing. Magda Friedjungová, Ph.D.
May 12, 2024

Contents

List of Figures

List of Tables

List of code listings

I would like to express my profound gratitude to Ing. Magda Friedjungová, Ph.D., for agreeing to supervise this thesis despite her busy schedule. I am thankful for the freedom she granted me in defining the assignment and choosing the methodology, for her prompt responses, and her meticulous reviews of my work. I deeply appreciate her professional perspective, willingness to help, and positive attitude.

Declaration

In Prague on May 12, 2024

Abstract

This bachelor's thesis deals with the possibilities of creating a game agent based on machine learning for a custom deterministic board game for two players called Crosphere. Three types of game agent based on Monte Carlo Tree Search algorithm combined with random simulations, heuristics and convolutional neural network were implemented. The second of the mentioned models has the best results, it is able to defeat a novice human opponent. An Android mobile application has also been created in which the user can play the game against a game agent.

Keywords artificial intelligence, machine learning, board game, Crosphere, Android, Monte Carlo Tree Search, Tensorflow Lite

Abstrakt

Tato bakalářská práce se zabývá možnostmi vytvoření herního agenta na bázi strojového učení pro vlastní deterministickou deskovou hru pro dva hráče s názvem Crosphere. Byly implementovány tři typy herního agenta založené na algoritmu Monte Carlo Tree Search v kombinaci s náhodnými simulacemi, heuristikou a konvoluční neuronovou sítí. Druhý ze zmíněných modelů má nejlepší výsledky, je schopný porazit začátečnického lidského protivníka. Byla také vytvořena mobilní aplikace pro Android, ve které uživatel může hru hrát proti hernímu agentovi.

Klíčová slova umělá inteligence, strojové učení, desková hra, Crosphere, Android, Monte Carlo Tree Search, Tensorflow Lite

List of abbreviations

AI	Artificial Intelligence
ANN	Artificial Neural Network
BFS	Breadth-First Search
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DNN	Deep Neural Network
MCTS	Monte Carlo Tree Search
ML	Machine Learning
MSE	Mean Squared Error
PPO	Proximal Policy Optimization
RL	Reinforcement Learning
SL	Supervised Learning
UCB	Upper Confidence Bound

Introduction

This thesis is driven by three main goals:

1. The invention of a new board game
2. The creation of a game agent capable of playing this game
3. The development of a mobile application that allows users to play against the game agent

The board game should be a two-player abstract strategy game that's deterministic and zero-sum, much like Go or some other games discussed in Section 1.1.

The motivation for this thesis was derived from a fondness for board games, particularly abstract strategy games, where simple rules can lead to complex strategies and heuristics. This hobby, combined with a fascination with machine learning algorithms and artificial intelligence, resulted in the choice of this topic.

State-of-the-art techniques in game-playing algorithms are investigated in Chapter 1. The rules and mechanics of Crosphere, a board game designed specifically for this thesis (goal 1), are laid out in Chapter 2. Chapter 3 explains the process of building the game agent (goal 2), while Chapter 4 describes the features and user interface of the mobile application (goal 3). Finally, Chapter 5 presents the thesis results and examines areas for improvement.

All figures, tables and code listings were created by the author.

Chapter 1

Game-Playing Algorithms

Game-playing algorithm usually consists of several distinct modules. Typical architecture is based on a tree search algorithm for exploring possible moves. To evaluate the desirability of the move, tree search is in most cases combined with an evaluation function, which can be implemented as a simple formula utilizing basic information (e.g. number of pieces each player has on the game board) or as an artificial neural network. This network can be trained by using recordings of human players, which is a form of supervised learning, or by self-play, where it plays against itself, which is a type of reinforcement learning. The whole system is often complemented by tablebase, a database used simply for memorizing the best moves in frequent situations.

1.1 Evolution of Game-Playing Algorithms: From Checkers to StarCraft II

Over the past 30 years, game-playing algorithms have managed to beat the best human players in many strategy games. In 1994, Chinook, a checkers-playing program created by Jonathan Schaeffer, beat in checkers Marion Tinsley - the best checkers player in the world. Chinook used a vast endgame tablebase (see Section 1.6) with perfect information about the outcomes from all the game positions with ten or fewer peaces on the checkerboard and combined it with forward search allowing Chinook to see 16 to 17 moves deep into the future [1]. The forward search met the endgame tablebase, thanks to which Shaeffer in 2007 solved checkers [2]. He proved that if both players play perfectly, the game will end in a draw.

Three years after Chinook's success, in 1997, an IBM supercomputer called Deep Blue beat in chess Garry Kasparov [3] – then-world chess champion, leaving the public stunned. Deep Blue utilized the Minimax algorithm (see Section 1.2.1) combined with an evaluation function designed by human chess masters, evaluating each possible position based on factors such as the number of pieces on the board, piece mobility, pawn structure, king safety, and other strategic considerations (see Section 1.5.1).

Attention turned to Go, probably the oldest board game continuously played to the present day [4]. But in this game, the game-playing algorithms did not do nearly as well as in checkers or chess, playing well below the level of human professional players. One of the reasons why the game of Go has long been an intractable problem for algorithms is the high number of possible moves a player can make at each turn, known as the branching factor. In checkers, the branching factor is around 10, in chess it is 30–35 and in Go it is 200–300 [5]. In Go, determining the leading player is also more challenging compared to chess, where you can count pieces for a rough indication. In Go, merely counting pieces doesn't provide meaningful insights into the game's dynamics or the relative strength of each player. Finally, in 2016, AlphaGo, a computer program developed by DeepMind, beat the world champion in Go Lee Sedol [6]. AlphaGo utilized a combination of Monte Carlo Tree Search (see Section 1.2.2), Deep Neural Networks (see Section 1.3) and Reinforcement Learning (see Section 1.4). The developers first trained the neural network using game recordings of human professional players and then let it play against itself and its previous versions, giving it superhuman performance.

However, the achievements of game-playing algorithms extend beyond deterministic games. In 2018, DeepMind's bot AlphaStar defeated in StarCraft II a top professional player, Grzegorz "MaNa" Komincz and became a StarCraft grandmaster [7].

As game-playing algorithms continue to evolve and conquer diverse challenges, the future holds exciting possibilities for artificial intelligence in gaming.

1.2 Game Tree Search

In game theory, the game tree is a directed graph in which nodes represent game states (e.g. distribution of pieces on the game board) and edges represent moves in the game (e.g. moving a piece from one position to another) [8]. Game-tree search is a technique for analyzing a game to determine what moves a player should make to win a game [9].

1.2.1 Minimax with Alpha-Beta Pruning

The Minimax algorithm [10] is a fundamental strategy in deterministic games. This algorithm aims to determine the optimal move for a player by exploring the game tree and evaluating potential outcomes.

The algorithm utilizes an evaluation function to assess each game state, assigning a real number to quantify its desirability. A higher numerical value indicates a more favorable state for the black player, while a lower value suggests an advantage for the white player. For instance, a score of -1 implies a slight advantage for the white player, whereas a score of 8 indicates a significantly advantageous position for the black player.

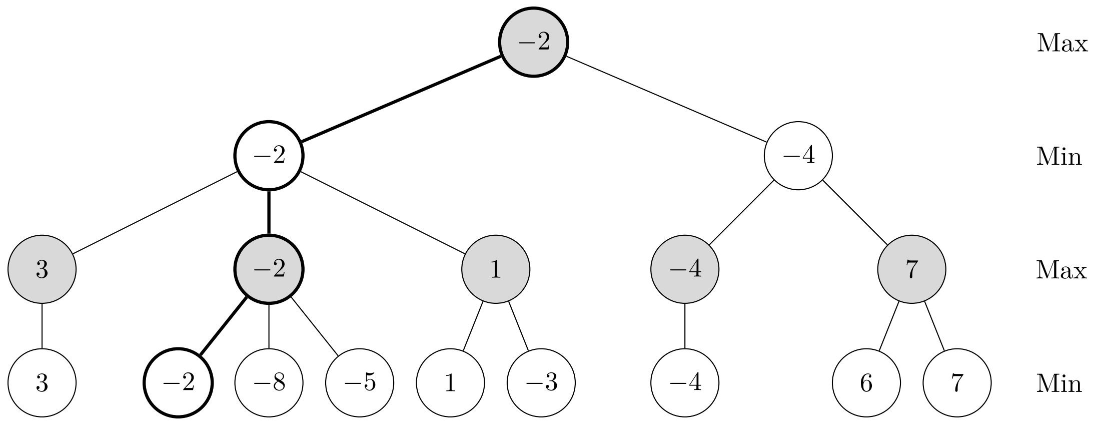


Figure 1.1 Game tree explored by Minimax algorithm

In Figure 1.1, we see a visualization of the game tree explored by the Minimax algorithm. The tree represents the possible moves and outcomes of a two-player game, where the black player is on his turn. Each node in the tree represents a game state, and the values associated with the nodes indicate a score assigned by the evaluation function. The first and the third layers represent the states where the black player is on his move, and the second and the fourth layers represent the states where the white player is on his turn.

The algorithm backtracks from the leaf nodes towards the root of the tree. At each decision node, it chooses the move that has the highest score if the black player is on his turn or the move that has the lowest score if the white player is on his turn. Once the entire tree is explored or up to a specified depth, the algorithm identifies the optimal move for the maximizing player. This move represents the path that leads to the most advantageous outcome, considering the opponent's best responses. The branch that was selected by the algorithm is highlighted in bold. The time complexity of the Minimax algorithm is $O(b^d)$ where b is the branching factor (average number of child nodes for each node) and d is the depth of the tree (the number of layers to which is the tree explored).

To enhance efficiency, the Minimax algorithm incorporates Alpha-Beta Pruning. During traversal, if the algorithm identifies a path that will not influence the final decision, it prunes that branch, reducing the number of nodes to explore. In practice, Alpha-Beta Pruning often results in a substantial performance improvement.

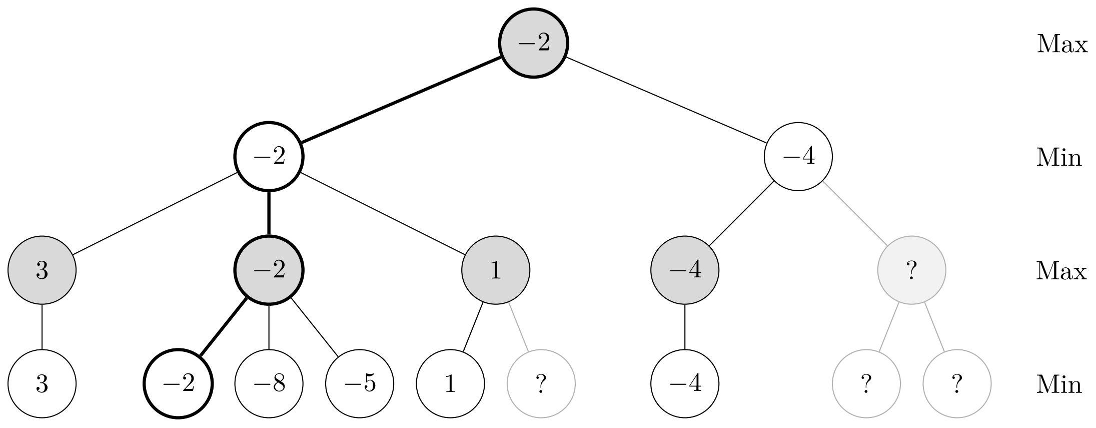


Figure 1.2 Game tree pruned by Alpha-Beta Pruning

The time complexity of the Minimax algorithm with Alpha-Beta pruning can be generally expressed as $O(b^{\frac{d}{2}})$. In the best-case scenario, where Alpha-Beta Pruning is highly effective, the algorithm may explore only a fraction of the nodes compared to the non-pruned version. However, in the worst case (no pruning benefits), the time complexity may still approach $O(b^d)$ [11].

The effectiveness of the Minimax algorithm heavily relies on the accuracy of the evaluation function used to estimate the desirability of a given game state. Minimax explores the entire game tree up to a certain depth, resulting in exponential growth in the number of nodes to be considered, which is a problem, especially in games with high branching factors. It is designed for deterministic games where the outcome solely depends on player moves. When able to explore the entire game tree, Minimax guarantees to find the optimal move that leads to the best possible outcome for the player, assuming both players play optimally [12].

1.2.2 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) [13] is a probabilistic and heuristic-driven search algorithm. It has gained prominence in the realm of decision-making processes including game-playing. Instead of exhaustively exploring all possible moves, MCTS builds a tree incrementally by sampling potential moves and simulating their outcomes, maintaining a balance between exploration and exploitation. MCTS consists of four repetitive phases:

1. **Selection:** In the selection phase, the algorithm starts at the root of the tree and traverses down the tree. At each fork, the child of the current node that has the highest score of the

following Upper Confidence Bound (UCB) formula is always selected.

$UCB_i = \frac{w_i}{n_i} + c \cdot \sqrt{\frac{\ln t}{n_i}}$ where:

- UCB_i is the UCB score for the i-th node
- w_i is the number of wins for the node considered after the i-th move
- n_i is the number of simulations for the node considered after the i-th move
- c is the constant called exploration parameter; $c \in [0, \infty)$
- t is the total number of simulations after the i-th move run by the parent node of the one considered

The traversal through the tree continues until it reaches a leaf node (a node that has not been fully expanded, meaning it still has child nodes that haven't been explored). Figure 1.3 shows the game tree generated by MCTS, with each node showing the ratio of wins to total playouts from that point in the game tree for the player that the node represents.

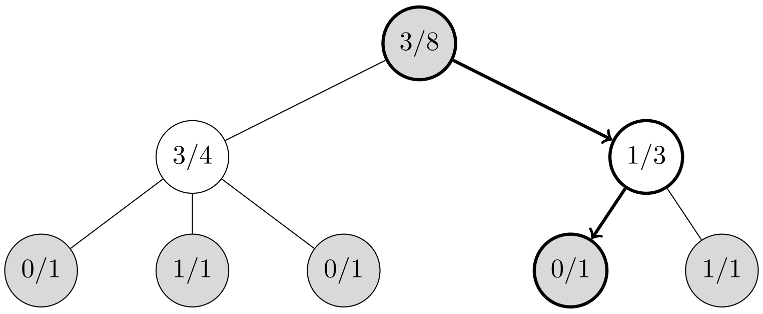


Figure 1.3 Selection phase of the MCTS

2. **Expansion:** Once a leaf node is selected, a new (unexploded) child node is added to the selected node.

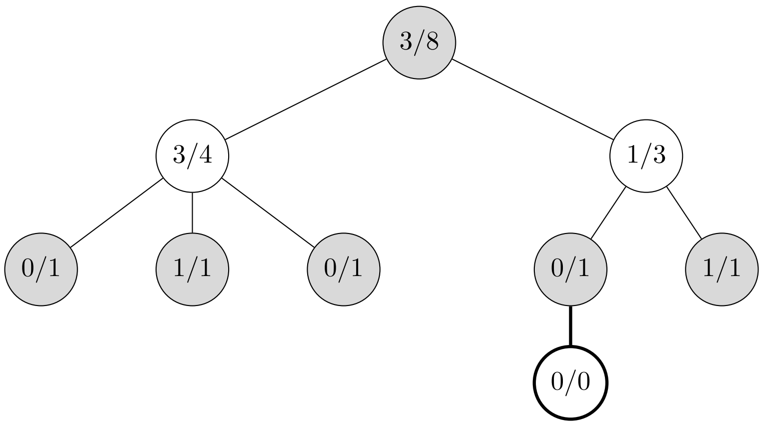


Figure 1.4 Expansion phase of the MCTS

3. **Simulation:** In the simulation phase, it plays a game of completely random decisions from the new node until it reaches either a terminal state (win or loss) or a simulation cap is reached, meaning that the predefined limit on the number of simulated steps has been exceeded.

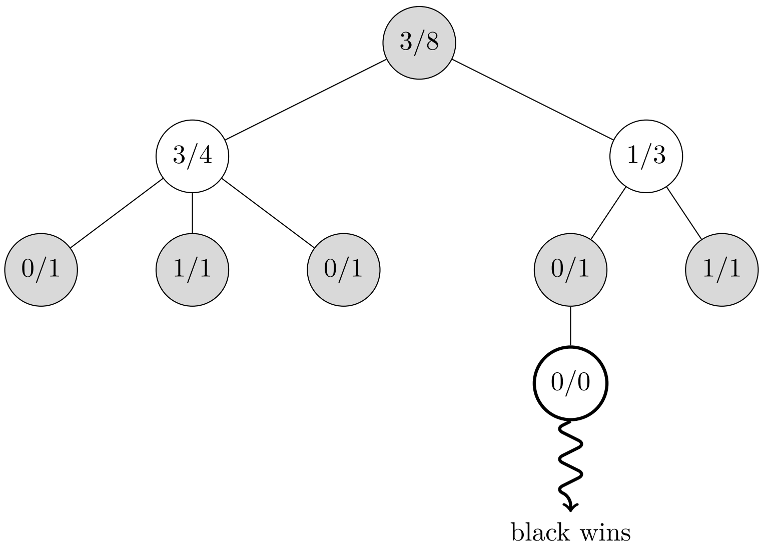


Figure 1.5 Simulation phase of the MCTS

4. **Backpropagation:** After the simulation, the results are backpropagated up the tree. The statistics of the nodes along the path from the root to the expanded node are updated based on the outcome of the simulation. In each node, the number of simulations (n) and the number of wins (w) after this move is stored. This information is updated in the backpropagation phase and then used in the UCB formula in the simulation phase when the cycle starts again.

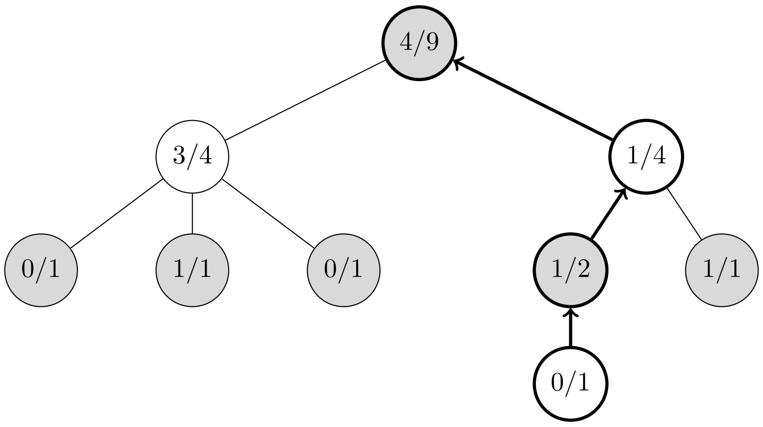


Figure 1.6 Backpropagation phase of the MCTS

MCTS excels in scenarios with large decision spaces where exhaustively exploring all possibilities is impractical. It focuses on promising branches of the search tree using the exploration-exploitation trade-off, making it well-suited for complex games with high branching factors. It can handle games with uncertain outcomes or imperfect information because it relies on sampling and statistics. This makes it more adaptable to non-deterministic games. Unlike Minimax, MCTS doesn't require an explicit evaluation function, making it applicable to games where defining a heuristic evaluation function is challenging. In certain cases, MCTS may overfit to particular patterns or biases present in the early simulations. [14, 15]

1.3 Deep Neural Networks

Artificial Neural Networks (ANNs) [16] are computational models inspired by the structure and function of biological neural networks in the human brain. They consist of interconnected nodes, often referred to as neurons or artificial neurons, organized into layers (one input layer, an optional number of hidden layers, and one output layer). Deep neural networks (DNNs) are a class of ANNs that have two or more hidden layers. These networks are capable of learning complex relationships within data and they have been successful in various machine learning tasks, including game-playing.

1.3.1 Architecture of DNNs

Layers are connected through weights representing the strength and significance of the relationships between neurons. Every neuron receives inputs from all neurons in the preceding layer and transforms them by weights and activation function (operation applied to the output of a neuron to introduce non-linearity to the network). The input signal traverses through the entire network, passing through each layer in sequence:

1. **Input Layer:** The input layer is the first layer of the network, and its primary function is to receive the raw input data. Each neuron in the input layer represents a feature or attribute of the input. In game-playing, the input is typically a current game state. For instance, one neuron in the input layer can represent one square on the game board. However, heuristic information such as (in chess) *information about the strength of the pawn structure* or *relative safety of the king* can also serve as input [17].

2. **Hidden Layers:** Hidden layers follow the input layer. These layers are called "hidden" because they are not directly observable in the input or output of the network. In game-playing, hidden layers enable the algorithm to adapt and generalize to different game scenarios. Instead of memorizing specific sequences of actions, the algorithm can learn flexible and adaptive strategies by adjusting the weights in the hidden layers based on feedback from the game environment.

3. **Output Layer:** The output layer follows the hidden layers and provides the network's decision, such as the best move to make in a board game or estimating how promising the given state is for each of the players. For zero-sum games, the output often represents a probability distribution over possible actions.

In Figure 1.7, there is an example of DNN created for game-playing. This network estimates the winning chances of players in the game state on input.

1.3.2 Training of DNNs

1. **Initialization:** Initialize the weights in the neural network. This can be done randomly.

2. **Forward Pass:** Input data (e.g. game state) is passed through the network in a forward direction. The outputs of neurons in each layer are computed based on the weighted sum of inputs and the activation function.

3. **Loss Calculation:** The predicted output (e.g. *In this state, black has a 65% chance of winning and white 35%*) is compared with the actual target output (e.g. *In the end, white won this game*) using a loss function. The loss function is used to quantify the error between the predicted and true values.

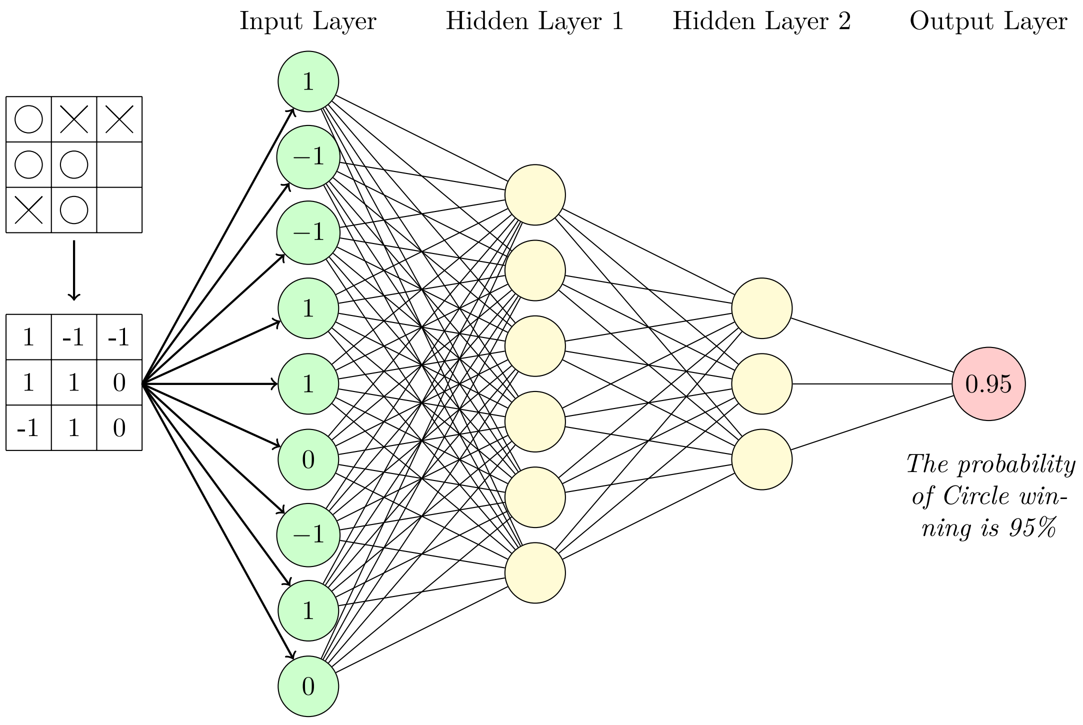


Figure 1.7 DNN estimating the winning chances in the current game state

4. **Backward Pass (Backpropagation)**: The error is propagated backward through the network to calculate how much each weight contributed to the overall error. The weights are then adjusted to minimize the loss, improving the network's predictions for similar input data (game states) in subsequent iterations.

   Steps 2 through 4 are iteratively executed for all the training data.

1.3.3 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) [18] are a specialized type of neural network designed for processing and analyzing grid data, such as images or board game states. While traditional DNNs are effective in handling structured tabular data, CNNs excel in tasks that involve spatial hierarchies and local patterns.

1.3.3.1 Convolution

As the name suggests, CNNs utilize (discrete) convolution. Discrete convolution is a mathematical operation that combines two tensors to produce a third tensor. It is defined by the following formula:

$$(f * g)[n] = \sum_m f[m] \cdot g[n-m]$$

where:

- f and g are the input tensors
- $(f * g)[n]$ represents the value at position n in the resulting convolution

- $f[m]$ is the value of the input tensor f at position m
- $g[n-m]$ is the value of the input tensor g at the relative position $n-m$
- The sum is taken over all possible positions m that overlap with the input tensors f and g

In the context of CNNs and board games:

- f is typically the input game state or feature map
- g is the convolutional filter (also known as a kernel)
- The result of the convolution operation $(f * g)$ is often used as an input to activation functions or the following layers

Convolution in CNNs can be expressed as sliding a filter over the input data and computing the dot product at each step, as shown in Figure 1.8. The filter contains learnable parameters that are adjusted during the training process, enabling the network to learn spatial information and detect features like edges, textures, and shapes. Consider a 2D convolution operation. The input is a 2D tensor, representing a game state, and the filter is a smaller tensor sliding across the input.

$$\underset{\text{Input}}{\begin{array}{|c|c|c|}\hline X_{11} & X_{12} & X_{13} \\ \hline X_{21} & X_{22} & X_{23} \\ \hline X_{31} & X_{32} & X_{33} \\ \hline\end{array}} * \underset{\text{Filter/Kernel}}{\begin{array}{|c|c|}\hline \color{red}{w_{11}} & \color{blue}{w_{12}} \\ \hline \color{green}{w_{21}} & \color{brown}{w_{22}} \\ \hline\end{array}} = \underset{\text{Output}}{\begin{array}{|c|c|}\hline O_{11} & O_{12} \\ \hline O_{21} & O_{22} \\ \hline\end{array}}$$

$$O_{11} = \color{red}{w_{11}} \cdot X_{11} + \color{blue}{w_{12}} \cdot X_{12} + \color{green}{w_{21}} \cdot X_{21} + \color{brown}{w_{22}} \cdot X_{22}$$

$$O_{11} = \color{red}{w_{11}} \cdot X_{12} + \color{blue}{w_{12}} \cdot X_{13} + \color{green}{w_{21}} \cdot X_{22} + \color{brown}{w_{22}} \cdot X_{23}$$

$$O_{11} = \color{red}{w_{11}} \cdot X_{21} + \color{blue}{w_{12}} \cdot X_{22} + \color{green}{w_{21}} \cdot X_{31} + \color{brown}{w_{22}} \cdot X_{32}$$

$$O_{11} = \color{red}{w_{11}} \cdot X_{22} + \color{blue}{w_{12}} \cdot X_{23} + \color{green}{w_{21}} \cdot X_{32} + \color{brown}{w_{22}} \cdot X_{33}$$

Figure 1.8 Convolution between Input and Filter

In discrete convolution by definition, the second input tensor is first rotated 180° and then slid over the first input tensor (discrete convolution, unlike the continuous convolution, is not commutative). In the context of CNNs, the term "convolution" is often used to refer to what is technically a "cross-correlation" operation. In this operation, the filter is not rotated during the convolution, just like in this figure.

1.3.3.2 Architecture of CNNs

In CNNs, three types of hidden layers are used:

- **Convolutional Layers:** Convolutional layers apply convolutional operations to input data, enabling the network to learn local patterns and features (refer to Figure 1.8).
- **Pooling Layers:** Pooling layers downsample the spatial dimensions of the data, reducing its resolution and computational complexity. Max pooling retains the maximum value from a local region (typically a field of the same size and shape as the filter), emphasizing the most relevant features detected by the convolutional layers. The pooling layer is often positioned immediately after the convolutional layer, and it is usually considered as a part of that convolutional layer.
- **Fully Connected Layers:** These layers are equivalent to the hidden layers found in traditional DNNs. Each neuron within this layer is connected to all the neurons in the preceding layer. These layers, located at the end of the network, transform the spatial information into a flat vector suitable for making final predictions.

  In Figure 1.9, there is an example of CNN created for game-playing.

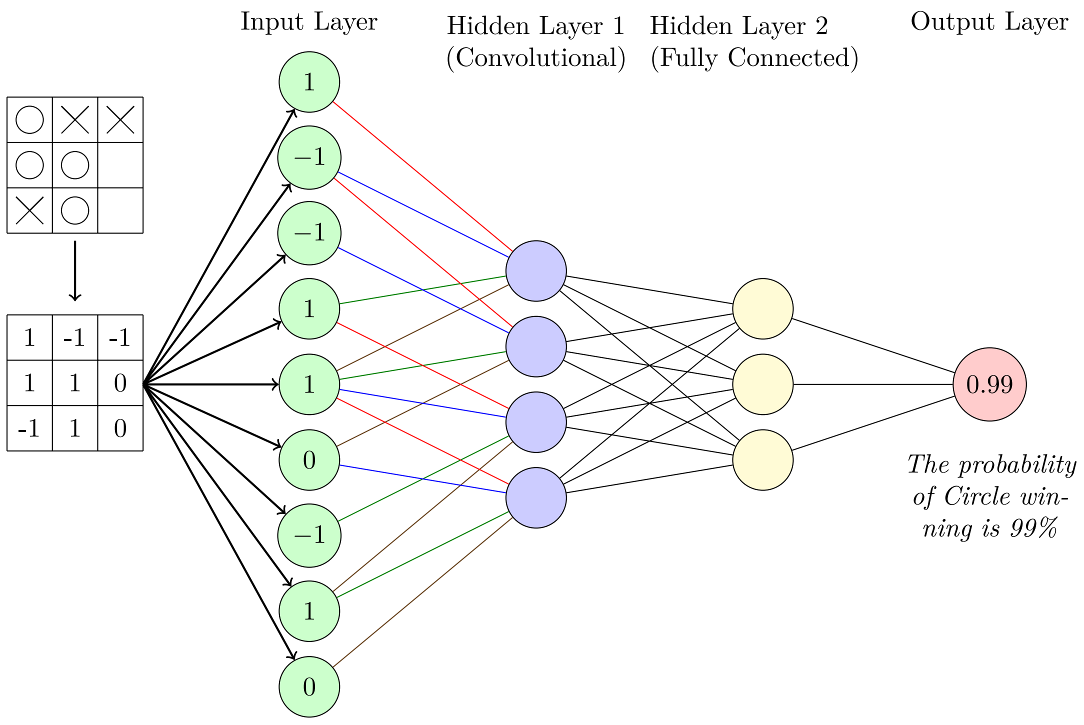

Input Layer
Hidden Layer 1 (Convolutional)
Hidden Layer 2 (Fully Connected)
Output Layer
1 -1 -1
1 1 0
-1 1 0
1
−1
−1
1
1
0
−1
1
0
0.99
The probability of Circle winning is 99%

Figure 1.9 CNN estimating the winning chances in the current game state

1.4 Reinforcement learning

Reinforcement Learning (RL) [19, 20] is a type of machine learning technique that enables an agent to learn in an interactive environment by trial and error using feedback from its actions and experiences. There are several key concepts:

- **Agent:** The entity that makes decisions within the game environment.
- **Environment:** The context or setting in which the agent operates.
- **State:** A specific situation or configuration within the environment.

- **Action:** The decision or move made by the agent.
- **Reward:** A numerical value that indicates the immediate feedback from the environment, guiding the agent towards desirable behavior.
- **Policy:** The strategy or mapping from states to actions that the agent employs.

The essence of this approach is self-play, where the agent plays games many times against itself or against some of the previous versions of itself.

1.4.1 Q-Learning

Q-Learning [21, 22, 23] is a model-free RL algorithm that has been successfully applied to game-playing scenarios. It maintains a Q-table (or Q-function) that stores the expected cumulative rewards for all possible state-action pairs in the environment. The Q-value (or quality value) represents the utility of taking a specific action in a particular state.

1.4.2 Policy Gradient methods

Policy gradient methods [24] in RL directly optimize the agent's policy, allowing it to adjust its actions based on gradients derived from performance metrics. REINFORCE [25], one of the simplest policy gradient algorithms, uses the "score function" to calculate the gradient of the expected reward, then updates the policy accordingly. However, this method can suffer from high variance, leading to unstable learning. Proximal Policy Optimization (PPO) [26] addresses this issue by implementing a "trust region" around policy updates, ensuring that changes are not too large and thus providing greater stability. PPO balances exploration and exploitation, making it a popular choice for a wide range of reinforcement learning applications.

1.5 Heuristic-Driven Approaches

Heuristics [27] are strategies that exploit specific knowledge about the game domain to evaluate positions and guide decision-making. Heuristics do not guarantee optimal solutions but aim for satisfactory outcomes in a more computationally efficient manner. In other words, heuristics are all about solving problems quicker where classic methods fail. One of the heuristic algorithms is MCTS (see Section 1.2.2).

1.5.1 Heuristics Using a Formula

In some games, it is possible to derive generally applicable rules to evaluate game states which are helpful in most of the situations. In chess, for example, common heuristics are: Chained Pawns (good), a free and advanced Pawn (very good), an isolated Pawn (bad), doubled Pawns (bad), occupying and controlling the central squares (good), a well-protected King (good) and many others [28]. This can be used to create a formula taking into account the importance of individual heuristics for evaluating the game state. This attitude was successfully used by IMB in their chess-playing expert system Deep Blue [29] in conjunction with the Minimax algorithm (see Section 1.2.1).

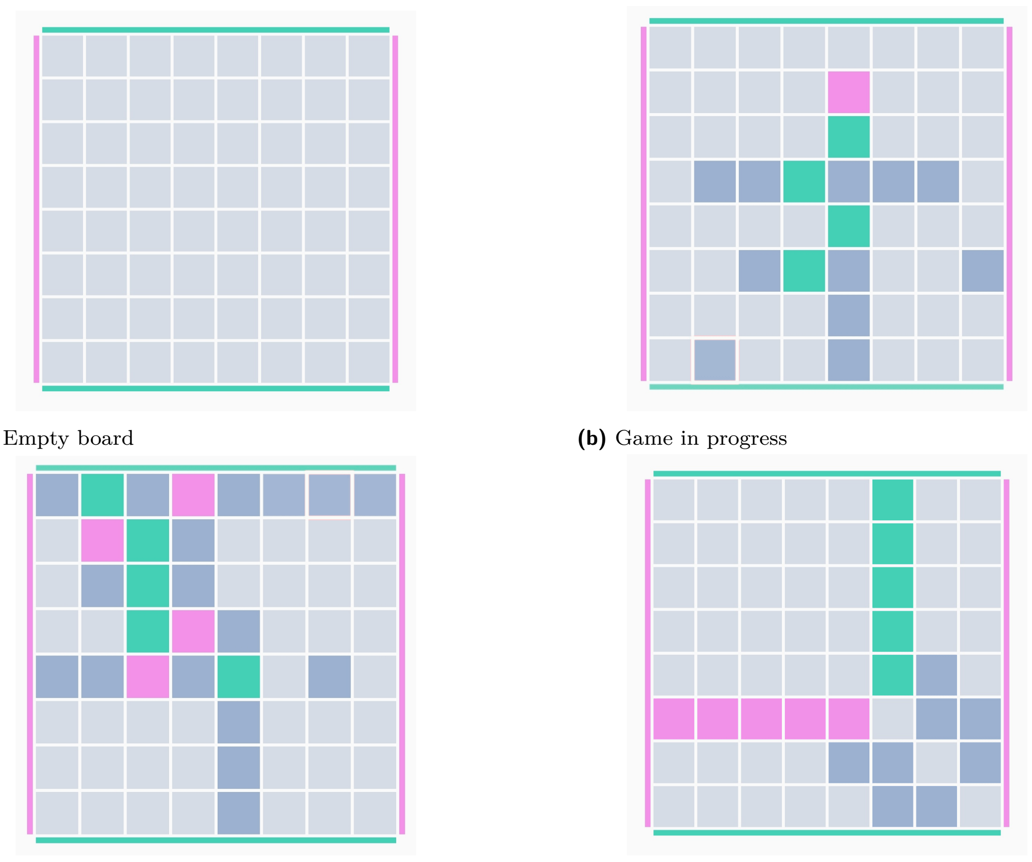
1.6 Opening and Endgame Tablebases

Tablebases in chess refer to precomputed databases that store perfect or near-perfect information about the best moves and outcomes in the specific opening (the initial phase of the game) or endgame (the final phase of the game) positions. One such endgame tablebase for chess is The Nalimov Tablebases, requiring 7.05GB of storage for all 5-piece endings, with 6 and 7-piece endings requiring 1.2TB and 140TB of storage respectively [27]. As of 2024, work is still underway to solve all 8-piece positions [30].

Chapter 2

Crosphere: The Custom Board Game

This chapter describes the setup, rules, and objectives for the grid-based strategic board game Crosphere. This abstract game, played on an 8×8 grid, is deterministic and zero-sum. The two players take turns placing stones on the board, aiming to connect opposite edges.



(a) Empty board

(b) Game in progress

(c) Pink wins

(d) Game ends in a draw

Figure 2.1 Various game scenarios

2.1 Game preparation

Each player possesses five strong stones in their respective colors (green or pink), visibly displayed next to the board. Weak stones (gray), conveniently placed within reach, are shared by both players in the game.

2.2 Legal moves

The player controlling the green stones makes the first move, choosing to place either a strong stone or a weak stone on any vacant square of the chessboard. Players alternate turns, placing stones on empty squares. Once a player deploys all their strong stones, they can only place weak stones, with no limit on the quantity (always one stone per turn). Stones cannot be relocated or removed once placed.

2.3 Aim of the game

The game concludes when a player successfully links opposite edges of the chessboard. For victory, the green player must connect the top and bottom edges, while the pink player aims to connect the left and right edges. This connection is established by forming an unbroken line of adjacent stones, either of the player's strong or weak stones, ensuring at least one stone is placed on each edge to be connected. Stones are considered adjacent if their squares share a common border, excluding those sharing only a corner without a common border.

2.4 Special situations

Weak stones serve a dual purpose in connecting edges, allowing both players to utilize them regardless of the placer. A scenario may occur where placing a weak stone simultaneously connects the top and bottom edges with green and gray stones (the goal of the green player) and the left and right edges with pink and gray stones (the goal of the pink player). However, to prevent a simultaneous win for both players, such moves are prohibited. This rule introduces the possibility of a draw, where neither player succeeds in connecting their respective edges. This situation is visible in Figure 2.1d.

2.5 Parameters of the game

In approximately 43.1% of cases, the game concludes with a victory for the first (green) player, while 56.6% of games end in a win for the second (pink) player. Draws occur in only 0.3% of cases. The game can also be played with different board sizes, such as 5×5 instead of the standard 8×8. The number of strong stones can also be varied–for example, 3 instead of the usual 5.

Chapter 3

Training of Game-Playing Agent

This chapter explores three approaches to create an optimal game-playing agent for Crosphere and describes its development: MCTS was combined with random simulations, heuristics, and CNNs.[1]

3.1 MCTS with random simulations

In accordance with the description outlined in Chapter 1.2.2, I have implemented the Monte Carlo Tree Search (MCTS) algorithm for the game Crosphere using Python.

Crosphere presents a significant branching factor, with players having 128 options for their initial move and an average of around 80 options as the game progresses. This is relatively high number compared to e.g. chess, with average branching factor around 35, but still much smaller than average branching factor in Go, which is around 250.

When I executed the MCTS code on my laptop, it achieved a computation rate of around 200 rollouts per second. Each rollout simulated a single game, influencing the evaluation of game states and aiding in the selection of optimal moves. With a 5-second time constraint per move, the exploration of the game tree reached approximately 2 moves in depth in certain sections, and up to 4 moves in others. The ideal value for exploration parameter turned out to be around 0.3.

This approach proved to be ineffective for this game. When playing against an opponent making random moves, the game agent constructed using this method won only 98 out of 100 games. Occasionally, it even lost to chance.

3.2 MCTS with heuristics

The second approach was based on the first one, the only difference was in evaluation of the game states. This time, the evaluation was not based on random simulations, but on the result of heuristic formula. The formula took into account several variables:

- d_1 – The minimum number of moves required for the first player win the game
- d_2 – The minimum number of moves required for the second player win the game
- s_1 – The count of first player's available strong stones
- s_2 – The count of second player's available strong stones

[1]Play Agent: https://gitlab.fit.cvut.cz/jezekda3/crosphere-play-agent.git

- *stoneWeight* – A constant parameter empirically determined to be 0.4

All the variables above are either directly given or easy to calculate. d_1 and d_2 is possible to compute using customized Breadth-First Search (BFS) algorithm. The resulting evaluation formula is as follows: $evaluation = d_2 - d_1 + stoneWeight \cdot (s_2 - s_1)$. The outcome is then normalized using a sigmoid function. Consequently, the algorithm selects moves that aim to minimize the number of moves required for its own victory, maximize the number of moves required for the opponent's victory, and judiciously utilizes strong stones, leveraging them only when they provide a significant advantage.

This heuristic proved surprisingly effective. In 100 games, it outperformed the previous model, winning all 100 matches.

3.3 MCTS with CNN

In this approach, the evaluation relies on outcomes derived from a CNN. To train the CNN, I utilized data generated from MCTS with heuristics. I let MCTS with heuristics play against itself in 60,000 games, with each move involving 500-2500 rollouts. Overall, this simulation required approximately 3200 computing hours and was conducted on 4 CPUs (simulating independently), taking 34 days to complete. The records of simulated games were stored in the format show in Code listing 3.1. The game record contains information regarding both models playing the game, including all their parameters, the game result (1 for the first player's victory, 0 for the second player's victory, and 0.5 for a draw), as well as details about the date and time the game was generated, and the name and operating system of the device on which it was generated. Certainly, the primary focus lies in the move records. Each record starts with a number indicating the stone type (0 for weak stone, 1 for strong stone), followed by a pair of numbers indicating the position to which the stone was placed. Moves with odd indices represent the first player's moves, while moves with even indices represent the second player's moves.

However, the raw format of the data couldn't be directly fed into the CNN for processing. Prior to input, the data required preprocessing. Since the CNN was tasked with evaluating game states rather than entire simulated games, the input consisted of specific states of the game. By providing the CNN with a game state, it could then output a value indicating how favorable the state was for the first player. A value close to 1 signified a highly favorable position for the first player, while a value close to 0 indicated the opposite. Therefore, the initial step involved extracting individual game positions from each game record.

In the preprocessed format, the board was represented using one-hot encoding, resulting in an array with a shape of 4×8×8. Additionally, to enhance the performance of the CNN, several simple features and heuristics were incorporated as input.

1. Empty fields
2. Weak stones
3. Strong stones of the first player
4. Strong stones of the second player
5. Closest distance from each field to the bottom edge (64 for unachievable)
6. Closest distance from each field to the right edge (64 for unachievable)
7. Minimum number of moves for the first player to win
8. Minimum number of moves for the second player to win
9. Whether the current player is the first player

Code listing 3.1 Raw Game Record

```
[Player_1 "type=MCTS_heuristic, exploration_weight=0.44,
 rollouts_per_move=922, stone_weight=0.39"]
[Player_2 "type=MCTS_heuristic, exploration_weight=0.5,
 rollouts_per_move=2069, stone_weight=0.42"]
[Result "0"]
[Datetime "13-03-2024 10:48:22"]
[Device "system=Linux, node=crosphere"]

1. (1, (1, 6))
2. (0, (2, 3))
3. (0, (0, 3))
4. (0, (0, 5))
5. (0, (3, 3))
6. (1, (1, 3))
7. (1, (0, 6))
8. (1, (3, 6))
9. (1, (3, 7))
10. (1, (2, 7))
11. (1, (2, 6))
12. (0, (2, 1))
13. (1, (3, 5))
14. (1, (4, 5))
15. (0, (2, 5))
16. (0, (2, 0))
17. (0, (7, 4))
18. (1, (4, 4))
19. (0, (5, 3))
20. (0, (4, 6))
21. (0, (6, 3))
22. (0, (4, 7))
23. (0, (7, 3))
24. (0, (2, 2))
25. (0, (2, 4))
26. (0, (3, 4))
```

10. Number of available strong stones for the first player
11. Number of available strong stones for the second player
12. Whether the first player won the game

Therefore, the final input array, representing a single data point in the dataset, has a shape of 11×8×8.

Code listing 3.2 Preprocessed state of the game

```
[[[ 1  1  1  0  1  0  0  1] // Empty fields
  [ 1  1  1  0  1  1  0  1]
  [ 1  0  1  0  1  1  0  0]
  [ 1  1  1  0  1  0  0  0]
  [ 1  1  1  1  1  0  1  1]
  [ 1  1  1  1  1  1  1  1]
  [ 1  1  1  1  1  1  1  1]
  [ 1  1  1  1  1  1  1  1]]

 [[ 0  0  0  1  0  1  0  0] // Weak stones
  [ 0  0  0  0  0  0  0  0]
  [ 0  1  0  1  0  0  0  0]
  [ 0  0  0  1  0  0  0  0]
  [ 0  0  0  0  0  0  0  0]
  [ 0  0  0  0  0  0  0  0]
  [ 0  0  0  0  0  0  0  0]
  [ 0  0  0  0  0  0  0  0]]

 [[ 0  0  0  0  0  0  1  0] // Strong stones of the first player
  [ 0  0  0  0  0  0  1  0]
  [ 0  0  0  0  0  0  1  0]
  [ 0  0  0  0  0  1  0  1]
  [ 0  0  0  0  0  0  0  0]
  [ 0  0  0  0  0  0  0  0]
  [ 0  0  0  0  0  0  0  0]
  [ 0  0  0  0  0  0  0  0]]

 [[ 0  0  0  0  0  0  0  0] // Strong stones of the second player
  [ 0  0  0  1  0  0  0  0]
  [ 0  0  0  0  0  0  0  1]
  [ 0  0  0  0  0  0  1  0]
  [ 0  0  0  0  0  1  0  0]
  [ 0  0  0  0  0  0  0  0]
  [ 0  0  0  0  0  0  0  0]
  [ 0  0  0  0  0  0  0  0]]

 [[ 8  7  7  7  7  6  6  7] // Closest distance from each field to the
  [ 7  6  6 64  6  7  6  7] // bottom edge (64 for unachievable)
  [ 6  5  5  4  5  6  6 64]
  [ 5  5  5  4  5  5 64  4]
  [ 4  4  4  4  4 64  4  4]
  [ 3  3  3  3  3  3  3  3]
  [ 2  2  2  2  2  2  2  2]
  [ 1  1  1  1  1  1  1  1]]

 [[ 7  6  5  4  5  5 64  1] // Closest distance from each field to the
  [ 7  6  5  4  5  6 64  1] // right edge (64 for unachievable)
  [ 6  5  5  4  5  6 64  0]
 [ 7  6  5  4  4 64  2 64]
 [ 7  6  5  4  3  2  2  1]
 [ 8  7  6  5  4  3  2  1]
 [ 8  7  6  5  4  3  2  1]
 [ 8  7  6  5  4  3  2  1]]

[[ 6  6  6  6  6  6  6  6] // Minimum number of moves for the first
 [ 6  6  6  6  6  6  6  6] // player to win
 [ 6  6  6  6  6  6  6  6]
 [ 6  6  6  6  6  6  6  6]
 [ 6  6  6  6  6  6  6  6]
 [ 6  6  6  6  6  6  6  6]
 [ 6  6  6  6  6  6  6  6]
 [ 6  6  6  6  6  6  6  6]]

[[ 6  6  6  6  6  6  6  6] // Minimum number of moves for the second
 [ 6  6  6  6  6  6  6  6] // player to win
 [ 6  6  6  6  6  6  6  6]
 [ 6  6  6  6  6  6  6  6]
 [ 6  6  6  6  6  6  6  6]
 [ 6  6  6  6  6  6  6  6]
 [ 6  6  6  6  6  6  6  6]
 [ 6  6  6  6  6  6  6  6]]

[[ 0  0  0  0  0  0  0  0] // Whether the current player is the first
 [ 0  0  0  0  0  0  0  0] // player
 [ 0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0]]

[[ 0  0  0  0  0  0  0  0] // Number of available strong stones for
 [ 0  0  0  0  0  0  0  0] // the first player
 [ 0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0]]

[[ 1  1  1  1  1  1  1  1] // Number of available strong stones for
 [ 1  1  1  1  1  1  1  1] // the second player
 [ 1  1  1  1  1  1  1  1]
 [ 1  1  1  1  1  1  1  1]
 [ 1  1  1  1  1  1  1  1]
 [ 1  1  1  1  1  1  1  1]
 [ 1  1  1  1  1  1  1  1]
 [ 1  1  1  1  1  1  1  1]]

[[ 0  0  0  0  0  0  0  0] // Whether the first player won the game
 [ 0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0]
 [ 0  0  0  0  0  0  0  0]
[ 0   0   0   0   0   0   0   0]
[ 0   0   0   0   0   0   0   0]]]
```

The last value, which denotes the result of the game, serves as the predicted variable. You can see an example of a data point in Code listing 3.2.

Initially, one game state was selected from each simulated game to avoid multicollinearity, which could occur when multiple states from the same game are used. This approach provided a dataset of 60,000 data points from 60,000 simulated games. In a later experiment, all game states from each game were included, yielding approximately 1,650,000 data points, with each game averaging 27.5 moves. The results with this larger dataset were slightly better, so this method was chosen for the final model.

3.3.1 Architecture of the CNN

The most effective CNN used in the final model consists of four layers:

1. **Convolutional Layer 1**: This layer contains 32 filters, each of size (3, 3, 3), with a stride of 1 in all dimensions. Padding is set to "same," and the activation function is ReLU.
2. **Convolutional Layer 2**: Similar to the first layer, this one contains 64 filters of size (3, 3, 3), with the same stride and padding configuration. The activation function is also ReLU.
3. **Fully Connected Layer 1**: This layer has 128 neurons, and the input is flattened before entering. The activation function is ReLU.
4. **Fully Connected Layer 2**: This is the final output layer with a single neuron, without an activation function.

Several hyperparameters were tuned during the model's development:

- **Alternative CNN Architecture**: A deeper CNN, with 4 convolutional layers and 3 fully connected layers, was also tested. However, the final model that was selected used a simpler architecture with 2 convolutional layers and 2 fully connected layers, as described above.
- **Number of Epochs**: When training the final model on a larger dataset consisting of 1,650,000 data points, the optimal number of epochs was found to be 1, as additional epochs did not lead to improved performance. However, when training an earlier model on a smaller dataset of 60,000 data points, the best results were achieved with 8 to 12 epochs.
- **Loss Function**: A variety of loss functions were initially tested, including Binary Cross-Entropy, Mean Absolute Error, and Mean Squared Error (MSE). Ultimately, MSE was chosen as the loss function. MSE is defined as follows:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2,$$

  where y_i represents the true value, and $\hat{y}_i$ is the predicted value.
- **Activation Function**: The final output of the network is a probability indicating the likelihood that the first player will win the game, which requires a value between 0 and 1. To achieve this, a sigmoid activation function was applied in the last layer. Experiments were also conducted with sigmoid replacing ReLU in earlier layers, but the architecture described above yielded the best results.
- **Learning Rate and Optimizer**: The Adam optimizer with a learning rate of 0.001 was used. Different learning rates and other optimizers were tested, but this configuration gave the best results.

- **Batch Size**: Different batch sizes were evaluated. The final configuration uses a batch size of 32 for training and 128 for validation and testing.
- **Data Splitting**: The dataset was split into three subsets: 70% for training, 20% for validation, and 10% for testing.

3.4 Comparison of model performance

Ultimately, the CNN could not outperform the heuristic-based approach. As a result, the most effective model combined MCTS with heuristics, which was capable of defeating beginner-level players. A comparison of model performance is presented in Table 3.1.

	Random moves	MCTS with random simulations	MCTS with heuristics	MCTS with CNN
Random moves	-	2:0:98	0:0:100	1:0:99
MCTS with random simulations	98:0:2	-	0:0:100	41:0:59
MCTS with heuristics	100:0:0	100:0:0	-	62:0:38
MCTS with CNN	99:0:1	59:0:41	38:0:62	-

Table 3.1 Comparative results of different MCTS techniques in format Wins:Draws:Losses

3.5 Technology used

In this project, the development of the game engine, MCTS algorithm, tests, and other components was carried out using Python. TensorFlow was employed to create and train the CNN, due to its robust framework for deep learning and its compatibility with TensorFlow Lite, making it suitable for deployment in Android mobile applications.

The simpler tasks, such as basic code implementation and unit testing, were executed on a personal laptop. However, the more resource-intensive tasks, including the simulation of 60,000 games of Crosphere played by MCTS with heuristics, required a greater amount of computational power. To accommodate these demands, a virtual machine on CloudFIT[2] was utilized, where 4 CPUs were rented for approximately 2 months to conduct these simulations.

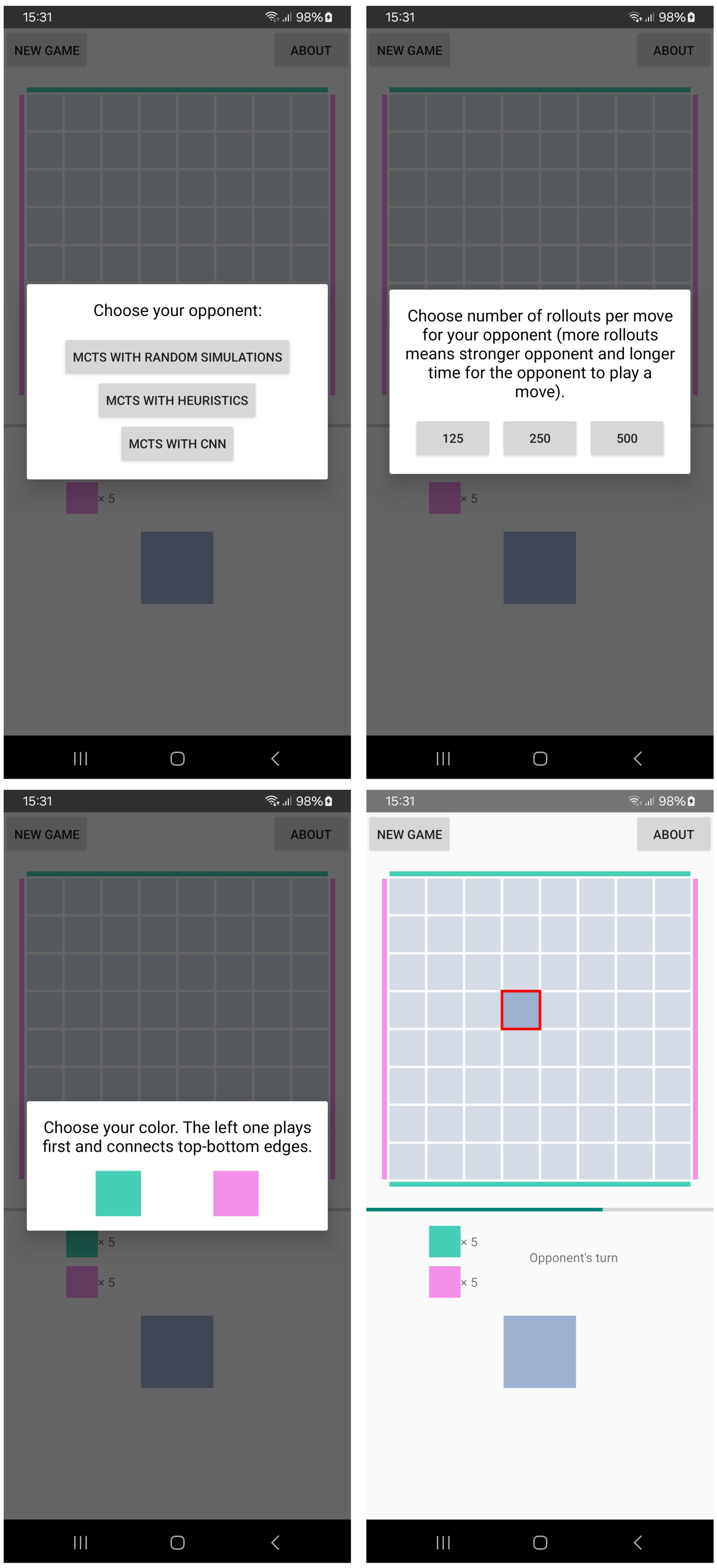
[2]CloudFIT: https://help.fit.cvut.cz/cloud-fit/index.html

Chapter 4
Mobile Application

The final objective of this thesis was to create a mobile application for Android that allows users to play Crosphere, the custom board game described in Chapter 2. In this app, developed in Android Studio, users can play against three different types of game-playing agents, as described in Chapter 3. When starting a new game, users are first prompted to select the model type: MCTS with random simulation, MCTS with heuristics, or MCTS with CNN. Next, they choose the strength of their opponent, based on the number of rollouts per move: 125 (about 1 second per move), 250 (about 3 seconds), or 500 (about 8 seconds). A higher number of rollouts means a stronger but slower opponent. Finally, users choose a color—green or pink—with the green player taking the first move.

Once the game begins, the user interface provides several features to guide players through the gameplay. A progress bar indicates which stage of decision-making the opponent is in. A textbox displays important announcements, such as whose turn it is or who has won the game. Additionally, information is shown about the number of strong stones remaining for both the player and the opponent. There's also a button to toggle between strong and weak stones during play. The user interface is visible in Figure 4.1.[1]

[1]Android App: https://gitlab.fit.cvut.cz/jezekda3/crosphere-android-app.git



Choose your opponent:

Choose number of rollouts per move for your opponent (more rollouts means stronger opponent and longer time for the opponent to play a move).

Choose your color. The left one plays first and connects top-bottom edges.

Figure 4.1 Screenshots of the app's UI

Chapter 5 Conclusion

The thesis successfully met all its primary objectives: a new board game was invented, a game agent capable of playing this game was created, and a mobile application allowing users to play against the game agent was developed. Additionally, various techniques in tree search and artificial intelligence used for game-playing were examined.

However, there is room for improvement in the game agent's performance. The strongest model in this thesis, MCTS with heuristics, can defeat a novice human player. Nonetheless, there's considerable potential for the CNN to achieve superior performance, possibly reaching or exceeding human skill levels.

Several factors might explain why the CNN model fell short. Firstly, effective training of CNNs for complex tasks like this requires significant computational resources. DeepMind simulated 30 million games of Go to train AlphaGo, whereas this thesis simulated only 60,000 games. Another possible explanation lies in the design approach: AlphaGo used two separate CNNs—one to assess the quality of a game state and another to predict the probabilities of all possible next moves—while this project used only one. Additionally, the implementation of the MCTS algorithm could be optimized for better performance.

It is important to note that the implementation used in this study–combining MCTS with heuristics or a CNN, inspired by AlphaGo and its successors—is just one of many possible approaches for constructing a game-playing agent. Various alternative methods were discussed in Chapter 1, including Q-Learning, Policy Gradient techniques, and the use of Opening and Endgame Tablebases. Additionally, numerous other strategies could be explored for improving the performance of the game-playing agent.

This thesis offers a strong foundation for further development. The game agent could be enhanced with more robust models, and the mobile application's user experience and features could be expanded. With additional resources and focused research, this work could evolve into a more sophisticated gaming platform, offering new opportunities for innovation and exploration in the field of game-playing algorithms.

Appendix A
Attachment

The Play Agent repository contains code described in Chapter 3, the Android App repository contains code described in Chapter 4.

- Play Agent: https://gitlab.fit.cvut.cz/jezekda3/crosphere-play-agent.git
- Android App: https://gitlab.fit.cvut.cz/jezekda3/crosphere-android-app.git

Bibliography

1. MADRIGAL, Alexis C. *How Checkers Was Solved* [online, cited on: 2024/01/25]. [N.d.]. Available also from: `https://www.theatlantic.com/technology/archive/2017/07/marion-tinsley-checkers/534111/`.
2. SHAEFFER, Jonathan. *Checkers Is Solved* [online, cited on: 2024/01/29]. [N.d.]. Available also from: `https://www.researchgate.net/publication/231216842_Checkers_Is_Solved`.
3. GOODRICH, Joanna. *How IBM's Deep Blue Beat World Champion Chess Player Garry Kasparov* [online, cited on: 2024/01/25]. [N.d.]. Available also from: `https://spectrum.ieee.org/how-ibms-deep-blue-beat-world-champion-chess-player-garry-kasparov`.
4. SHOTWELL, Peter. *The Game of Go: Speculations on its Origins and Symbolism in Ancient China* [online, cited on: 2024/01/25]. [N.d.]. Available also from: `https://www.usgo-archive.org/sites/default/files/bh_library/originsofgo.pdf`.
5. LEVINOVITZ, Alan. *The Mystery of Go, the Ancient Game That Computers Still Can't Win* [online, cited on: 2024/01/25]. [N.d.]. Available also from: `https://www.wired.com/2014/05/the-world-of-computer-go/`.
6. MOYER, Christopher. *How Google's AlphaGo Beat a Go World Champion* [online, cited on: 2024/01/25]. [N.d.]. Available also from: `https://www.theatlantic.com/technology/archive/2017/07/marion-tinsley-checkers/534111/`.
7. THOMPSON, Tommy. *How AlphaStar Became a StarCraft Grandmaster* [online, cited on: 2024/01/29]. [N.d.]. Available also from: `https://www.aiandgames.com/p/alphastar`.
8. LAVALLE, Steven M. *Game Trees* [online, cited on: 2024/01/29]. [N.d.]. Available also from: `https://lavalle.pl/planning/node519.html`.
9. ZUCKERMAN, Inon. *Avoiding game-tree pathology in 2-player adversarial search* [online, cited on: 2024/01/29]. [N.d.]. Available also from: `https://onlinelibrary.wiley.com/doi/full/10.1111/coin.12162`.
10. YANOVSKAYA, E.B. *Minimax principle* [online, cited on: 2024/05/06]. [N.d.]. Available also from: `https://encyclopediaofmath.org/index.php?title=Minimax_principle`.
11. MEGALOOIKONOMOU, Vasilis. *Artificial Intelligence course: CIS603 Spring 03: Lecture 7* [online, cited on: 2024/01/08]. [N.d.]. Available also from: `https://cis.temple.edu/~vasilis/Courses/CIS603/Lectures/l7.html`.
12. KALLES, Dimitris. *A Minimax Tutor for Learning to Play a Board Game* [online, cited on: 2024/01/29]. [N.d.]. Available also from: `http://students.ceid.upatras.gr/~kanellop/pubs/Aig08.pdf`.

13. BRADBERRY, Jeff. *Introduction to Monte Carlo Tree Search* [online, cited on: 2024/01/09]. [N.d.]. Available also from: `http://jeffbradberry.com/posts/2015/09/intro-to-monte-carlo-tree-search/`.
14. RAHUL, Roy. *ML — Monte Carlo Tree Search (MCTS)* [online, cited on: 2024/01/9]. [N.d.]. Available also from: `https://www.geeksforgeeks.org/ml-monte-carlo-tree-search-mcts/`.
15. THOMPSON, Tommy. *AI 101: Monte Carlo Tree Search* [online, cited on: 2024/01/09]. [N.d.]. Available also from: `https://www.youtube.com/watch?v=lhFXKNyA0QA`.
16. HARDESTY, Larry. *Explained: Neural networks* [online, cited on: 2024/05/06]. [N.d.]. Available also from: `https://news.mit.edu/2017/explained-neural-networks-deep-learning-0414`.
17. HUI, Jonathan. *AlphaGo: How it works technically?* [online, cited on: 2024/01/16]. [N.d.]. Available also from: `https://jonathan-hui.medium.com/alphago-how-it-works-technically-26ddcc085319`.
18. SAHA, Sumit. *A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way* [online, cited on: 2024/05/06]. [N.d.]. Available also from: `https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53`.
19. BHATT, Shweta. *Reinforcement Learning 101: Learn the essentials of Reinforcement Learning!* [online, cited on: 2024/01/24]. [N.d.]. Available also from: `https://towardsdatascience.com/reinforcement-learning-101-e24b50e1d292`.
20. SUTTON, Richard S. *Reinforcement Learning: An Introduction.* [N.d.]. Available also from: `http://incompleteideas.net/book/the-book-2nd.html`.
21. KERNER, Sean Michael. *Q-learning* [online, cited on: 2024/05/06]. [N.d.]. Available also from: `https://www.techtarget.com/searchenterpriseai/definition/Q-learning`.
22. XENOU, Konstantia. *Deep Reinforcement Learning in Strategic Board Game Environments* [online, cited on: 2024/01/24]. [N.d.]. Available also from: `https://link.springer.com/chapter/10.1007/978-3-030-14174-5_16`.
23. REHMAN, Ibad. *Human-Level Gaming Performance with Deep Q-Learning* [online, cited on: 2024/01/24]. [N.d.]. Available also from: `https://www.linkedin.com/pulse/human-level-gaming-performance-deep-q-learning-ibad-rehman`.
24. WENG, Lilian. *Policy Gradient Algorithms* [online, cited on: 2024/05/06]. [N.d.]. Available also from: `https://lilianweng.github.io/posts/2018-04-08-policy-gradient/`.
25. SOFEIKOV, Konstantin. *REINFORCE algorithm — Reinforcement Learning from scratch in PyTorch* [online, cited on: 2024/05/07]. [N.d.]. Available also from: `https://medium.com/@sofeikov/reinforce-algorithm-reinforcement-learning-from-scratch-in-pytorch-41fcccafa107`.
26. AL., John Schulman et. *Proximal Policy Optimization Algorithms* [online, cited on: 2024/05/07]. [N.d.]. Available also from: `https://arxiv.org/abs/1707.06347`.
27. HIRANI, Avnish. *Machine Learning and AI Case Studies - Part 2: Heuristic Decision Trees/Search and Chess* [online, cited on: 2024/01/25]. [N.d.]. Available also from: `https://www.linkedin.com/pulse/machine-learning-ai-case-studies-part-2-heuristic-chess-hirani-msc`.
28. PLOOG, David. *Abstract Game Heuristics* [online, cited on: 2024/01/25]. [N.d.]. Available also from: `https://www.abstractgames.org/heuristics.html`.
29. SALETAN, William. *The triumphant teamwork of humans and computers* [online, cited on: 2024/01/25]. [N.d.]. Available also from: `https://slate.com/technology/2007/05/the-triumphant-teamwork-of-humans-and-computers.html`.

30. ALLIS, Louis Victor. *Searching for Solutions in Games and Artificial Intelligence* [online, cited on: 2024/01/25]. [N.d.]. Available also from: `http://fragrieu.free.fr/SearchingForSolutions.pdf`.

Contents of the Attachment

Play Agent repository content:

```
│ data ............................................................ generated data
│  ├── games .................................................. records of simulated games
│  │    ├── local_games.txt ........................... games simulated on author's laptop
│  │    └── remote_games.txt ............................ games simulated using Cloud FIT
│  ├── models ............................................................ CNN models
│  │    ├── pth ................................................ models in PyTorch format
│  │    │    ├── model_10.pth
│  │    │    ├── model_5_195.pth
│  │    │    ├── model_6_202.pth
│  │    │    ├── model_7_205.pth
│  │    │    └── model_8_167.pth
│  │    └── tflite ....................................... models in TensorFlow Lite format
│  │         ├── model_1.tflite
│  │         ├── model_10.tflite
│  │         ├── model_2.tflite
│  │         ├── model_4.tflite
│  │         └── model_5.tflite
│  ├── preprocessed .......................... game states in input format for CNN models
│  │    ├── local_games.npy
│  │    └── remote_games.npy
│  └── tournaments ................................... comparison of model's performance
│       └── tournaments.txt
├── pyproject.toml ................ configuration file containing metadata and dependencies
├── README.md ........................................ a brief description of the media content
├── src ..................................................... source code of the project
│  ├── engine.py ............................................ game rules and visualization
│  ├── __init__.py
│  ├── __main__.py ................................... main file with code to be executed
│  ├── MCTS.py ............................................................ play agents
│  └── notebooks ................................................... Jupyter notebooks
│       ├── cnn_tensorflow.ipynb ............... code for training Tensorflow Lite CNN model
│       └── cnn_torch.ipynb ........................ code for training PyTorch CNN model
└── tests ........................................................... test scripts
     ├── __init__.py
     ├── test_engine.py .................................. tests for the code in engine.py file
     └── test_MCTS.py ..................................... tests for the code in MCTS.py file
```

Android App repository content:

```
app ............................................. application source code and resources
├── build.gradle.kts
├── FETCH_HEAD
├── gradle
├── gradle.properties
├── gradlew
├── gradlew.bat
├── local.properties
├── README.md ...................................... a brief description of the media content
└── settings.gradle.kts
```