TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Matvei Tarassov 213118IAIB

ANALYSIS OF MACHINE LEARNING TECHNIQUES FOR THE DEVELOPMENT AND IMPLEMENTATION OF ARTIFICIAL INTELLIGENCE IN A TURN-BASED GAME

Bachelor's Thesis

Supervisor: Gert Kanter
PhD

Tallinn 2024

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Matvei Tarassov 213118IAIB

MASINÕPPEMEETODITE ANALÜÜS TEHISINTELLEKTI ARENDAMISEKS JA RAKENDAMISEKS KÄIGUPÕHISES MÄNGUS

Bakalaureusetöö

Juhendaja: Gert Kanter
PhD

Tallinn 2024

Author's Declaration of Originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Matvei Tarassov

27.05.2024

Abstract

Turn-based games present a difficult challenge for game developers when developing AI opponents. Turn-based games demand planning, resource management, and dealing with imperfect information while presenting a large number of possible actions.

The goal of this thesis is to develop an alternative approach for creating opponents for turn-based games utilizing machine learning techniques. The work primarily focuses on adapting and implementing the AlphaZero algorithm with new modifications to the author's game. The new algorithm is referred to as PlayerZero. PlayerZero is able to automatically learn the strategies to play the game, by only utilising the rules of the game and data generated from playing against itself.

Model training was done using Python 3 with the Keras API, and the models were exported in the ONNX format. To ensure that the self-play game logic for reinforcement learning is efficient and shared with the game itself, it was developed within the Godot Game Engine, primarily using the C++ programming language. Model inference capabilities were retroactively added to the Godot Engine using the GDExtension technology with ONNX Runtime, hardware-accelerated by CUDA. The model training server trains multiple models, which are automatically upgraded to facilitate network curriculum learning.

The algorithm's performance is compared to multiple configurations of the Monte-Carlo Tree Search algorithm. We evaluate its performance based on algorithms ability to make optimal decisions, computational time, and memory usage.

The results show that PlayerZero is able to outperform the MCTS algorithm with an equal number of simulations after training for just one generation. After multiple generations of training, PlayerZero is able to outperform MCTS algorithms with a larger number of iterations while requiring less time to make decisions and using significantly less system memory.

The thesis is written in English and is 45 pages long, including 5 chapters, 10 figures and 3 tables.

Annotatsioon

Masinõppemeetodite analüüs tehisintellekti arendamiseks ja rakendamiseks käigupõhises mängus

Käigupõhised mängud pakuvad keerukat väljakutset mänguarendajatele, kes arendavad AI-vastaseid. Käigupõhised mängud nõuavad planeerimist, ressursside haldamist ja tegelemist mittetäieliku informatsiooniga, samal ajal esitades suure hulga võimalikke tegevusi.

Selle lõputöö eesmärk on välja töötada alternatiivne lähenemine AI loomiseks masinõppe tehnikate abil. Töö keskendub peamiselt modifitseeritud AlphaZero algoritmi rakendamisele autori mängus. See uus algoritm on nimetatud PlayerZero'ks. PlayerZero algoritm suudab automaatselt õppida mängu mängimise strateegiaid, kasutades ainult mängureegleid ja enda vastu mängimisel genereeritud andmeid.

Mudelite treenimist viidi läbi Python 3 abil, kasutades Keras API-d, ja mudelid eksporditi ONNX-formaati. Algoritmi enda vastu mängimise (Self-Play) loogika arendati Godot mängumootoris C++ programmeerimiskeelega, et tagada tõhusus ja ühilduvus mängumootoriga.

Mudelite ennustamise võimekus lisati tagantjärele Godot'isse, kasutades GDExtensioni tehnoloogiat ja ONNX Runtime'i, mis on kiirendatud CUDA-ga. Mudelitreeningu server treenib mitut mudelit, mida uuendatakse automaatselt, et hõlbustada võrgu õppekava õppimist (Network Curriculum Learning).

Algoritmi jõudlust võrreldakse mitme Monte-Carlo puuotsingu algoritmi konfiguratsiooniga, hinnates selle võimet teha optimaalseid otsuseid kiiresti.

Tulemused näitavad, et PlayerZero suudab pärast vaid ühe põlvkonna treenimist ületada MCTS algoritmi võrdse arvu simulatsioonidega. Mitme põlvkonna järel suudab PlayerZero ületada suuremaid MCTS algoritme, kasutades samal ajal vähem ressursse.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 45 leheküljel, 5 peatükki, 10 joonist, 3 tabelit.

List of Abbreviations and Terms

API	Application Programming Interface
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DCNN	Deep Convolutional Neural Network
DNN	Deep Neural Network
GARB	Global Attention Residual Block
GDE	GDExtension
GPU	Graphics Processing Unit
L2	Ridge Regression
MCTS	Monte Carlo Tree Search
ML	Machine Learning
NCL	Network Curriculum Learning
NN	Neural Network
ONNX	Open Neural Network Exchange
ORT	ONNX Runtime
RAM	Random Access Memory
RL	Reinforcment Learning
RPG	Role-Playing Game
ReLU	Rectified Linear Unit
TBS	Turn-Based Strategy
TBT	Turn-Based Tactics
VRAM	Video Random Access Memory
UCB	Upper Confidence Bound
UCT	Upper Confidence Bound applied for Trees

Table of Contents

List of Figures

List of Tables

1. Introduction

The topic of this bachelor's thesis is the implementation and analysis of machine learning techniques to direct actors in a new turn-based game environment. The final goal is to develop a sophisticated algorithm capable of playing the game at a level high enough to challenge human opponents.

The conventional approach to developing AIs for turn-based computer games primarily involves hand-crafted algorithms. [1, 2] This thesis explores an alternative approach where known techniques and algorithms from machine learning are used to develop a new algorithm capable of playing the game without relying on domain knowledge.

There are several significant limitations regarding the available approaches, which motivate the choices made in this thesis. Among them, the most significant is the absence of prior play data. AlphaGo [3], DeepChess [4], and other implementations [5] of AI for directing actors in turn-based games with the aid of ML leverage vast quantities of data from existing games played by human experts to train the models. However, methods utilizing existing data would not be applicable for a game that nobody has played before.

The choice of solutions is further constrained by the available computational resources during gameplay. It is crucial for the algorithm to output its decisions within a short timeframe on consumer hardware.

We employ an algorithm known to be capable of playing a variety of difficult board games, AlphaZero [6], in the author's own game. The techniques chosen are domain-agnostic and can be applied to other environments, not limited to a specific game. The thesis addresses the challenges of integrating the algorithm into the game engine, as well as the necessary modifications to the neural network architecture.

To achieve this goal, all essential game logic was rewritten in C++ with efficiency in mind. This facilitated running simulations and data generation algorithms without additional overhead from game graphics and other unnecessary systems. Modifications to the game engine were made using appropriate technologies to retroactively add support for model inference, hardware accelerated by CUDA via ORT. Both supervised learning and reinforcement learning techniques were utilized to train the final network. The core approach is largely based on AlphaZero [6], with some notable modifications and techniques from

other papers. The final algorithm is compared to a traditional MCTS algorithm on multiple characteristics through a novel tournament-based evaluation method.

The open-source solution is provided in Appendix 2.

The description of the rules of the game is written in Appendix 3.

The information about the system used for training the algorithm is provided in Appendix 4.

2. Background

2.1 Turn-Based Games

Turn-based games are a distinct type of game where players can only take actions during discrete turns. This stands in contrast to real-time games, where actions can occur at any moment, and the game progresses continuously. In turn-based games, players face no immediate penalty for taking time to consider their optimal actions, allowing them the opportunity to carefully plan and strategize. These aspects challenge the players to think ahead, allocate resources effectively, and, in some cases, make decisions with incomplete information.[7]

2.1.1 Popular Types Of Turn-Based Games

Turn-Based Strategy

Typically, strategic wargames like chess, checkers, and Go are popular examples of TBS board games. However, TBS games extend beyond traditional board games and are prevalent in the space of computer games as well. Prominent series of TBS computer games include Sid Meier's Civilization, Total War, and Heroes of Might and Magic.

Turn-Based Tactics

Turn-based tactics games, as the name implies, distinguish themselves from regular TBS games by focusing on smaller-scale combat scenarios where players control individual squads or characters to achieve specific objectives. Popular series of TBT games include XCOM, Wargroove, and Fire Emblem.

Turn-Based Elements in Other Games

Elements of turn-based combat exist in many other game series, particularly in RPGs, where the player controls a small party of characters in a series of encounters to advance through the game. These types of games are popular in the mainstream and include some of the best-selling franchises, such as Pokémon[8], and the 2023 Game of the Year: Baldur's Gate 3.[9]

2.1.2 AI And Turn-Based Games

Turn-based games only work when human players are matched against a challenging opponent. Game developers are tasked with creating algorithms that can serve as opponents for human players. "Successful turn-based strategy games depend strongly on robust artificial intelligence"[10].

Due to the way that the design of turn-based games emphasizes planning, resource management, and various other aspects, it is not trivial to develop algorithms that are able to play these games efficiently. Specifically, we are interested in decision making algorithms. The classic approach that game developers take is through algorithms that are hand-crafted for a specific domain, for example, decision trees [1].

Another example, the mechanism behind the AI in the critically acclaimed role-playing game Divinity: Original Sin 2, as outlined in Larian Studios' documentation, relies on calculating the score of an action based on how it affects the game state. The action with the highest score is selected. The system depends on many parameters, tuned by the developers specifically for the many possible scenarios that can occur in the game.[2] Appendix 5 has the descriptions for 6 out of a total of 173 multipliers used to calculate action scores [11].

Creating algorithms that are able to play turn-based strategy games has historically been a difficult task, and any time that major breakthroughs have resulted in AI being capable of beating human players in complex games such as Chess or Go, it became an important milestone for AI and AI research. [3, 12]

2.2 Monte-Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a decision-making algorithm that generates value estimates for actions from a given state. It does so by performing a large number of simulations, usually in the order of hundreds or thousands. The results from random simulations are used to iteratively build up a tree of nodes representing possible game states, where a transition from one node to another represents an in-game action.

Because the basic implementation of MCTS relies only on game simulations, it does not require any additional heuristics to work and can be implemented using only the rules of the game. However, numerous optimizations and modifications have been developed to successfully improve the performance of MCTS for specific domains. Such domains in-

clude those outside of games; for example, scheduling, vehicle routing, chemical synthesis, and security-related problems.[13, 14]

2.2.1 How Classic MCTS Works

MCTS is an algorithm that gradually builds up a tree of game states connected by actions taken by players of the game. Given enough time, the algorithm converges to the optimal action. If stopped prematurely or run for a lower number of iterations, it can still output a result with a small error probability [15]. There is a trade-off between accuracy and the resources required when running MCTS for a low number of iterations versus a higher one.

The following explination of the MCTS algorithm is a simplified summary of the Basic Theory provided in the article "Monte Carlo Tree Search: A Review of Recent Modifications and Applications" [14].

The Phases of MCTS

Each iteration of MCTS can be split into 4 main phases. The phases are described in the same order they are performed when the algorithm runs for a single iteration. The tree initially consists of just the root node, representing the state for which the next optimal action has to be determined.

1. **Selection**
   The Selection phase searches the tree for the first leaf node (a node that does not have any child nodes). The search begins from the root node and selects the child with the highest UCT score (see Section 2.2.2 for the UCT formula). This process is repeated for the children of the selected node until the first leaf node is found. On the first iteration, this will always be the root node.
2. **Expansion**
   In the Expansion phase, new children are added to the currently selected node for every possible action that can be taken from the game state associated with the node. If the selected node is terminal, meaning the game has ended, then no expansion is performed.
3. **Simulation** (Sometimes reffered to as "Rollout")
   A child is selected from the previous expansion phase, and a simulation of the game is performed starting from the game state of the node. A simulation consists of playing the game by randomly selecting actions from the state until the game ends. The game state can be evaluated using the rules of the game, and a score is calculated

based on the outcome. For example, the score for losses can be represented as -1.0, for victories as 1.0, and 0.0 for draws. For terminal nodes, the score is immediately available without the need to perform the simulation.

4. **Backpropogation**
   In the Backpropogation phase, the number of visits and the score of the selected node and all of its parents are updated by propagating it through the tree all the way back to the root.

An illustration of this process is shown in Figure 1.

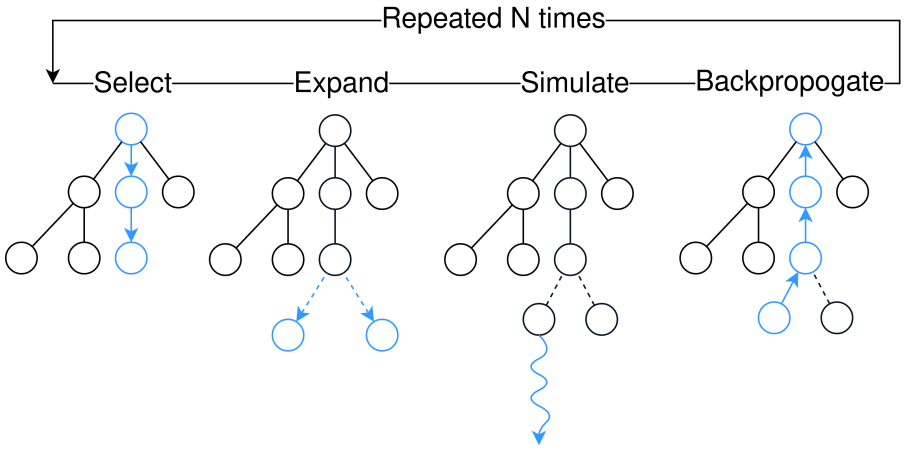

Figure 1. *An image illustrating the 4 phases of MCTS: Selection, Expansion, Simulation, and Backpropagation.*

2.2.2 Upper Confidence Bound Applied To Trees

The UCT is the policy used during the selection phase, fulfilling two important tasks. The first is selecting nodes that already have a high score associated with them, and the second is selecting nodes that are underexplored. These two terms are referred to as exploitation and exploration, respectively, and they are both essential for finding the optimal action [15]. Nodes which have never been visited have the highest priority; the rest of the nodes are evaluated with the following formula:

$$uct(n) = \frac{s}{v} + C\sqrt{\frac{\ln v_p}{v}} \tag{2.1}$$

Where:

- v is the number of times a given node has been visited.
- s is the sum of the scores of the node.

- C is a constant, usually $\sqrt{2}$, but depends on the game.
- v_p is the number of times the parent node has been visited.

When modeling the actions of the opponent for the MCTS algorithm, the exploitation term is inverted $1 - s/v$. The intuition is that while the algorithm chooses actions which maximize its own likelyhood of victoriy, the opponent chooses actions which minimize it.

Recovering The Best Action

The UCT policy exploitation term means that when the MCTS algorithm eventually converges to an optimal action, the corresponding node will have the highest score and the highest number of visits. However, when stopping the algorithm prematurely, the best possible decision for the current state can be recovered from the tree by picking either the action with the most visits or the highest score, which may not always align and produce different results [16].

2.3 Imperfect Information Games

IGG are games where there is limited observability of the game state. This can include games where certain information is unavailable, such as in card games, where it is unknown what cards are going to be drawn from the deck or the cards in the opponent's hand. Imperfect information can also include randomness; a simple example would be the outcomes of dice rolls. These types of games are more difficult to model than games with perfect information and are the primary challenge when using tree search-based algorithms such as MCTS. Many modern games are also imperfect information games, which makes addressing these challenges a high priority. [13, 14]

2.3.1 Addressing Randomness In IIG By Determinization

A rather simple approach to addressing randomness in IIG when using the MCTS algorithm is by Determinization. This technique has already been previously successfully applied to card games with inherent randomness, such as "Magic: The Gathering" [17]. It essentially works by sampling every possible outcome from a random event and adding it to the tree as new nodes. This approach has some drawbacks, as it can drastically increase the branching factor of the tree [14].

2.3.2 Other Modifications To MCTS

There are multiple articles written which review different modifications for the classic MCTS algorithm [14, 16]. The modifications in these articles cover various aspects of the algorithm, such as multiple approaches to parallelization, alternatives to the UCT selection policy, and more. However, while many of them have the potential to significantly boost the performance of the algorithm, they also introduce additional complexity into the implementation and, more importantly, bias. As outlined in [14], bias improves the algorithm's performance for one task but lowers its performance for a different task. In the scope of this thesis, we focus on the vanilla implementation of MCTS, with some exceptions, namely the introduction of determinization, as outlined in Section 2.3.1.

The introduction of Determinization into our implemenetation of MCTS is not motivated by the need to optimize it, but rather as a necessary modification to allow the algorithm to play the game. Another such unavoidable modification is the support for varying length turns. As it is not directly based on any prior work, this modification is explained in detail in the Implementation Chapter 3 of the thesis in Section 3.1.5

2.4 AlphaZero

AlphaZero is an algorithm developed by Google DeepMind that achieves superhuman performance in a variety of challenging decision-making tasks, including the games of Chess, Shogi, and Go. AlphaZero is a generalized approach created from AlphaGo Zero, an algorithm that learned to play the game of Go at a superhuman level, trained entirely through self-play. These algorithms do not receive any hints about the correct way to play the game or about the optimal strategies; however, they are able to discover them naturally through playing against themselves and learning from the data they create. AlphaZero is not restricted to just one domain like its predecessor AlphaGo Zero, and it is capable of outperforming state-of-the-art algorithms within hours of training in games like Shogi without major changes to the algorithm. [6]

AlphaZero is of particular interest to us due to its proven ability to learn to make decisions in other games, its relatively fast and stable training, and its lack of reliance on any prior knowledge of how to play a game.

We will provide a basic intuition of how the AlphaZero algorithm works; however, for a deeper understanding of the algorithm, it is recommended to read the original articles published for its predecessor AlphaGo Zero [18] and AlphaZero [6] itself, respectively.

2.4.1 How AlphaZero Works

The AlphaZero algorithm achieves its performance by combining MCTS and a DCNN. It performs a tree search similar to classic MCTS; however, instead of relying on the results of a large number of randomly played simulations, it utilizes the predictions of the DCNN to more efficiently direct the tree search.The DCNN takes as input the game state and has two outputs: the game state value and the policy. For the sake of this explanation, we will assume that the DCNN has already been trained and the predictions are close to optimal. The details on training are covered in the following Section 2.4.1.

The first output is the value, which represents the predicted outcome of the game given the current board. This is similar to the score value acquired when the game ends during the Simulation phase in classic MCTS, with the major difference being that this value is immediately available to us for any board state, rather than just a terminal one.

The second output is the policy, a probability distribution over all possible actions from the given state, where better actions have a higher probability and unfavorable actions have a close to zero probability.

A simplified AlphaZero DCNN model architecture is shown in Figure 2, with the two prediction heads illustrated.

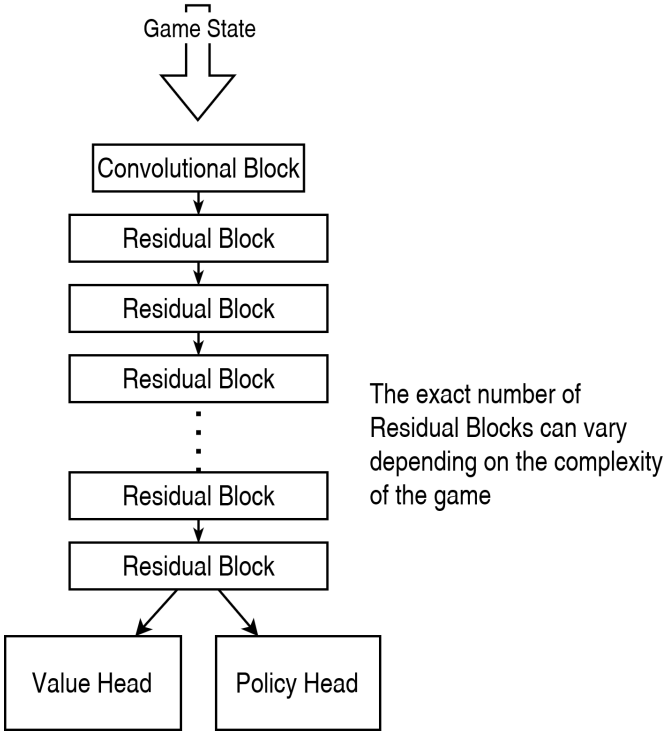

Figure 2. *A simplified diagram of the AlphaZero DCNN model architecture.*

The policy output is used during the initialization of new nodes in the expansion phase, where we are immediately able to assign which actions are more favorable given the game state. These probabilities are later used in the selection phase, where a different function is used in place of UCT, which takes advantage of these predictions to better direct its search to select actions with higher probability.

The value output is used to entirely replace the random play of the simulation phase. The value predicted by AlphaZero DCNN is a more accurate and reliable metric than a single simulation, and we use it to immediately assign the node with a score. If the node represents a terminal state, then the rules of the game are used for evaluation instead. Even a weak function for evaluating the game state is better than a long sequence of random decisions [14].

The problem with simulations in classic MCTS is that we do not know if the reason for a particular outcome of a simulation has had anything to do with our selected action or a coincidence when there are other actions which could have led to the result. The scores only start to become reliable once the MCTS has run a sufficient number of iterations and we can leverage the statistics of one action resulting in a higher average win rate than the other.

The diagram of the modified MCTS used in AlphaZero is shown in Figure 3.

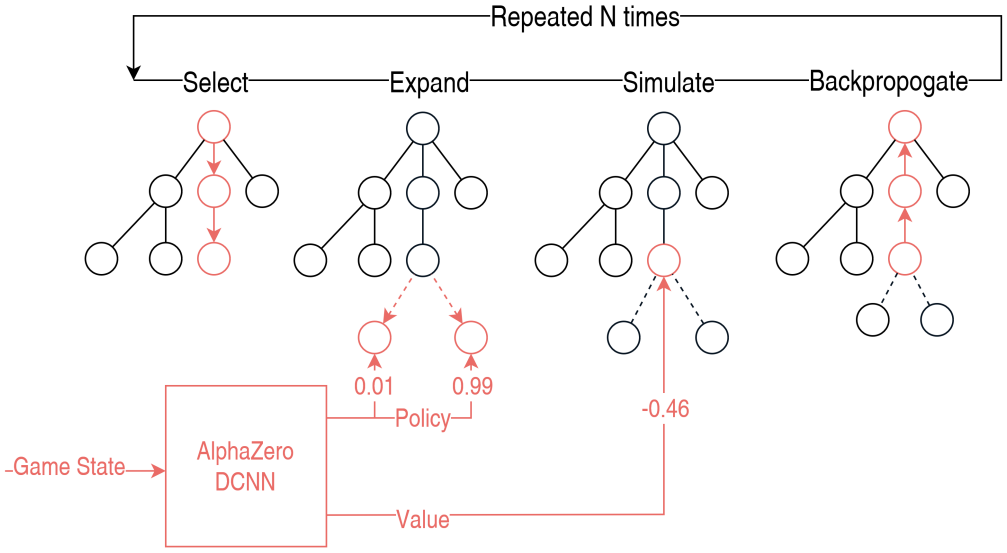

Figure 3. *The input for the DCNN is the selected node. The value head prediction is a single number in the* -1.0 *to* 1.0 *range, which represents the predicted winner given the game state. The value is relative to the current player, so* -1.0 *represents a loss, a* 0.0 *represents a draw, and* 1.0 *represents a victory. The policy head prediction is a vector of positive values which add up to* 1.0.

It is possible to use the DCNN directly without MCTS to direct an actor in a game; however,

when combined with MCTS, the algorithm is able to achieve far superior performance. The policy that is produced by MCTS is an improvement over the initial policy prediction from the DCNN. An arbitrary example of how this improvement looks is illustrated in Figure 4.

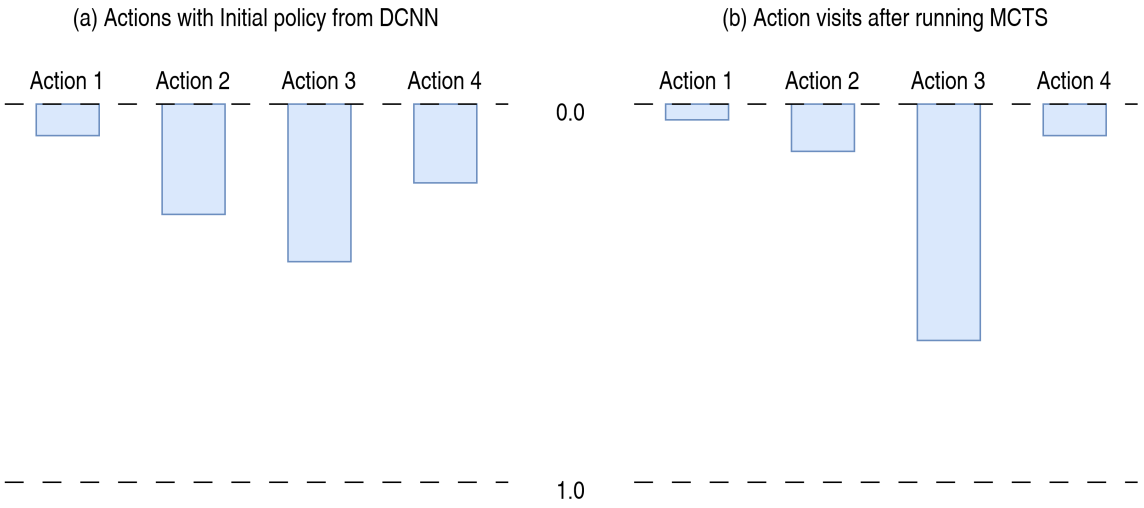

Figure 4. *(a) The initial policy prediction by the DCNN. (b) The number of visits for each of the actions after running the MCTS (values are normalized such that the sum adds up to 1).*

This description of the AlphaZero algorithm was based on the following sources: [6, 14, 18, 19].

AlphaZero Self-play And Training

The training of AlphaZero can be separated into two steps. The first step is Self-Play, where the algorithm plays a large number of games against itself and generates Self-Play data. As we established, MCTS is able to improve on the original policy predicted by the DCNN. This improved policy is the visit counts for each action, normalized to add up to 1.0. During Self-Play, the AlphaZero algorithm saves the game state, the improved policy, and the winner of the game [6].

Figure 5 illustrates this process for a single game; however, it is done many times for every generation of training to generate adequate quantities of data.

After a certain number of Self-Play games are played, the Self-Play process is stopped, and the data is used to train the DCNN using gradient descent. The policy head is trained to predict the new policy that was improved by MCTS, while the value head is trained on the final winner of the game, adjusted relative to the player.[6]

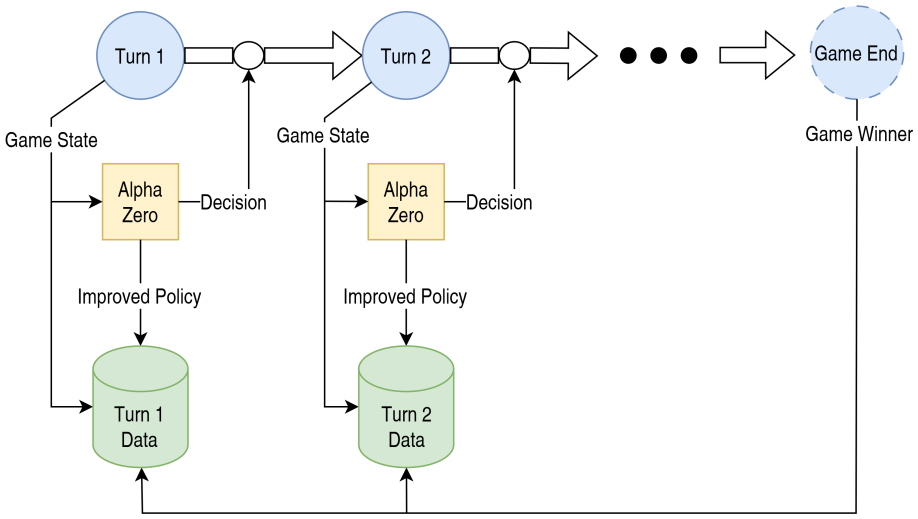

Figure 5. AlphaZero Self-Play: Data from every turn is stored. The data consists of the improved policy generated by the MCTS combined with DCNN, the game state, and the winner of the game.

2.5 The Game

The focus of this thesis is a new turn-based game that combines mechanics inspired by popular franchises with new original ideas.In short, the game is a grid-based roguelike with turn-based combat. A more detailed outline of the rules of the game is provided in Appendix 5. Development of the game began around 2022 as a hobby project using the Godot Engine. The motivation behind this thesis was the development of an algorithm capable of serving as an opponent to human players. Integrating the algorithm into the game engine was a priority, and certain technology choices were made to fulfill this need.

2.5.1 The Godot Game Engine

For the game engine, Godot Engine [20] was initially chosen due to being a free and open-source technology distributed under the MIT license, and because of its ease of use. Godot Engine is one of the most popular game engines for indie game developers, according to the total number of projects on itch.io [21]. It officially supports GDScript, C#, and C++ programming languages. GDScript is a language created specifically to be used within Godot Engine. "It is a high-level, object-oriented, imperative, and gradually typed programming language" [22].

A truncated code example from the official documentation GDScript documentation [22].

```
var a = 5
var s = "Hello"
```

```
var arr = [1, 2, 3]
var dict = {"key": "value", 2: 3}
var other_dict = {key = "value", other_key = 2}
var typed_var: int
var inferred_type := "String"

func some_function(param1, param2, param3):
        const local_const = 5

        if param1 < local_const:
                print(param1)
        elif param2 > 5:
                print(param2)
        else:
                print("Fail!")

        for i in range(20):
                print(i)

        while param2 != 0:
                param2 -= 1

        match param3:
                3:
                        print("param3 is 3!")
                _:
                        print("param3 is not 3!")

        var local_var = param1 + 3
        return local_var
```

3. Implementation

Implementation consists of two major components, reflecting the two important steps in the AlphaZero algorithm: the Self-Play client and the Training Server. The purpose of the Self-Play client is to generate Self-Play data, while the training server handles the building of new models, training them with the generated data, and deploying them to be used by the game client.

The two components are explained in detail in the following sections.

The architecure of the two components and how they integrate is shown in Figure 6.

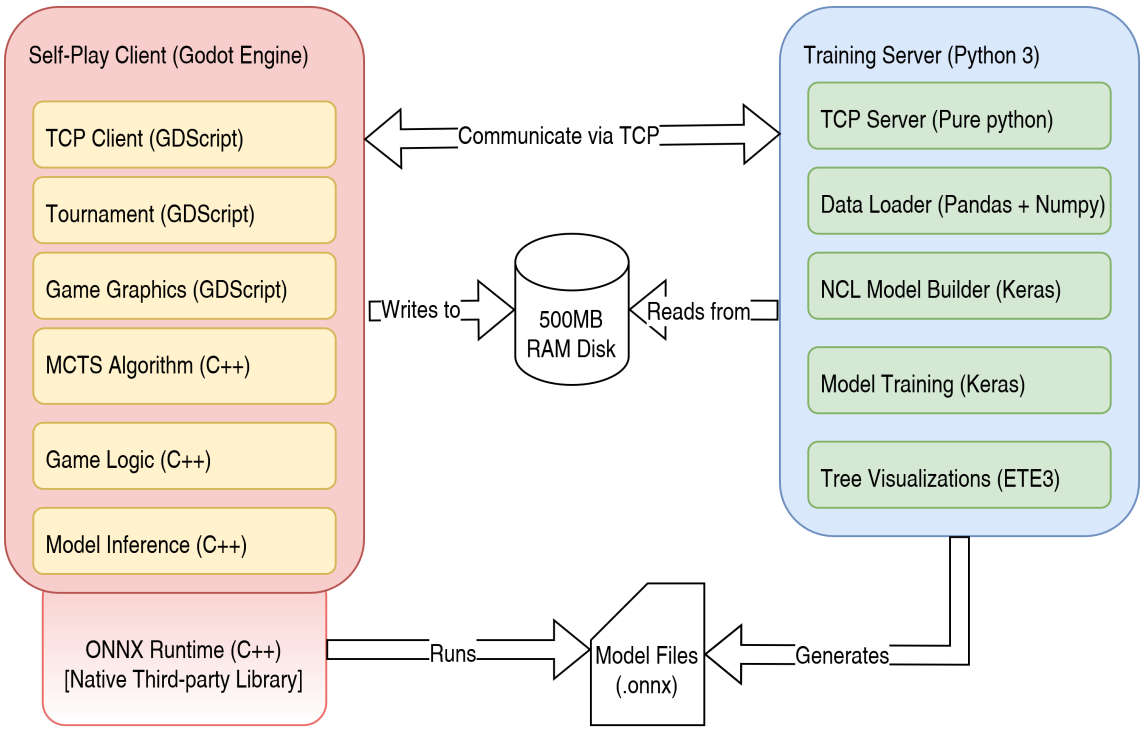

Figure 6. *Self-Play Client and Training Server architecture.*

3.1 Self-Play Client

3.1.1 Requirements For Self-Play Game Logic

The Game logic is an essential component of the game and Self-Play. It is directly responsible for implementing the rules of the game. It is paramount that the game logic fulfills specific requirements for it to be possible to use with the AlphaZero algorithm. The

requirements are outlined as follows:

1. The game logic needs to be as performant as possible.
2. The game logic needs to be integratable into the Godot Engine.
3. The game logic needs to support creating copies of game states.

The game logic needs to be as performant as possible

There are multiple reasons why this is important.

During self-play, the results of thousands of games are required to generate enough data to train the next generation of the algorithm. Every single one of those games consists of multiple turns, and during each turn, the MCTS algorithm is run. MCTS itself plays the game for thousands of turns. Needless to say, if the algorithm is not optimized, it becomes very time-consuming to develop, debug, train, and test. While the training time also depends on the hardware and the rules of the game, it is not always possible to change those aspects. We already established in Section 2.2.1 that the number of iterations is directly proportional to the performance of the algorithm; we cannot avoid these large numbers. The exact number needed for the MCTS algorithm to reach close to optimal performance can vary based on the rules of the game; however, it can easily reach thousands or tens of thousands. The more performant our code is, the more iterations we can run, and the better the results we will see.

The game logic needs to be integratable into the Godot Engine

If our final goal is to use AlphaZero to direct actors in the game, then the logic that powers it should also be integratable into the game.

The game logic needs to support creating copies of game states

Each node in the MCTS tree represents a transition from one state to another. Expansion relies on taking the current state of the node and creating a new copy of that state where the next action was taken. Many board games can be represented as a simple Bitboard [23] or a similar data structure. Copying and modifying the state for these games is fairly trivial. However, not all games are easily representable with a single bitboard or a multi-dimensional array. For our game specifically, we have several mechanics that make this a challenging requirement to fulfill. One example is that each character in the game consists of a large number of attributes and has a complex internal state.

3.1.2 Addressing Performance Limitations of GDScript

As already established, performance is incredibly important for generating large quantities of Self-Play data games. GDScript is easy to use; however, that comes at the cost of performance. According to the official engine documentation, for the fastest performance, it is advised to use GDExtension with C++ instead of GDScript [24]. All of the original game logic as presented in Appendix 5 was rewritten in C++.

3.1.3 Model Inference In Godot Engine

The Godot Engine does not natively support any form of model inference or have any ML-related functionality. Existing projects offering this functionality were considered; however, they either offer unnecessary functionality or are made for outdated versions of the engine and are no longer supported. This functionality had to be implemented.

Godot offers multiple ways of achieving this. As Godot Engine is an open-source technology, there was always an option to directly modify the source. A more reasonable approach, outlined in the official documentation for Godot 4, is GDExtension. "GDExtension is a Godot-specific technology that lets the engine interact with native shared libraries at run-time." [25] One of the requirements was to optimize the game code by rewriting it from GDScript to C++. This meant that both model inference and optimized game logic could be implemented as a single GDExtension in C++. C++ is the only officially supported language at the time of writing this thesis.

Runtime For Inference

There were never any plans to do model inference from scratch; the goal was instead to find a suitable technology that could be integrated into Godot Engine through GDExtension to perform this job. ONNX Runtime[26] fits this role perfectly. It is a cross-platform open-source high-performance inference engine for deep learning models. The ORT API is available for C++ as a thin wrapper around C, which is perfect for our needs since we can implement the GDExtension interface for ORT and optimized game logic all in one language.

ONNX Runtime supports an impressive suite of platforms, APIs, architectures, and hardware acceleration solutions. Particularly, we were interested in using CUDA [27] hardware acceleration. Using CUDA hardware acceleration can provide a significant boost to the performance of model inference during the generation of Self-Play data, as it offloads the heavy computations from the CPU to the GPU.

3.1.4 Game Logic Implementation

This section provides a short overview of the game logic implementation. The game logic was written in C++ and integrated into the Godot Game Engine using the GDExtension [25] technology.

The game state is managed using a custom *Surface* class. Various objects populate the game board, such as characters (implemented as the *Unit* class) and walls (implemented as the *DestructibleElement* class). Both of these classes inherit from the *SurfaceElement* class.

CastInfo Struct

The *CastInfo* is a C++ Struct (composite data type similar to a record). It contains all of the information about the action being used by a charather in the game.

It consists of the following:

- *Surface*: A reference to the *Surface* object where the action is used.
- *Caster*: A reference to the *Unit* object, which is the character using the action.
- *ActionIdentifier*: An enum used to identify which action is being used.
- *Target*: A *Vector2i* object representing the position where the action is used by the *Caster* (irrelevant in some cases).

Actions

The actions are implemented through static functions, each of which takes as input a *CastInfo* struct. Each action consists of the following three functions:

- **Checker Function**: Returns a boolean value representing whether a given *CastInfo* is valid with respect to the rules of the game.
- **Caster Function**: Modifies the state of the *Surface* based on the action's rules.
- **Generator Function**: Generates and returns a list of *CastInfo* structs representing all valid targets for the action. This function sometimes reuses logic from the checker function. Used by MCTS and PlayerZero during the Expansion phase.

Cloning

Every object implements a *clone* function, which returns an identical object. When the *Surface* object is cloned, it calls the clone functions of the *SurfaceElement* objects within

it.

Integration With The Rest Of The Engine

Each object inherits the Godot *RefCounted* class, and all of its methods are registered so that the game logic can be reused in the Godot Game Engine. This allows us to use the optimized logic written in C++ in conjunction with logic written in GDScript, significantly simplifying the integration into the actual game.

3.1.5 MCTS Implementation

Although the complete MCTS algorithm is not essential for the implementation of AlphaZero, it was developed for several reasons. Firstly, it offers a notably simpler alternative compared to AlphaZero. This allowed for thorough testing of the game logic, ensuring the proper functioning of its various components when running millions of iterations. Moreover, the MCTS algorithm serves as a baseline for assessing the performance of AlphaZero during benchmarking.

Varying Length Turns

A unique mechanic in the game, which is unusual for most board games like chess, is the fact that the turn of the current player does not end after performing an action. This means that a varying number of actions can be performed in a single turn. Turns are only ended when explicitly using the "End Turn" action. Figure 7 illustrates this difference when observing the game tree compared to games where the sides swap after every action.

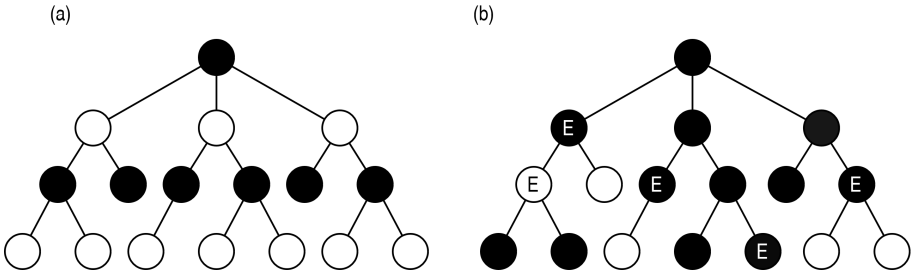

Figure 7. *A diagram demonstrating the effect of varying length turns on the game tree. Different sides (players) are represented by white and black colored nodes. (a) Game tree for games where turn duration is exactly 1 action. (b) Game tree for a game where the turn can last a varying number of actions, the nodes where the "End Turn" action is performed are marked with an E.*

Because a turn no longer consists of just one action, the logic to recover the best action is also slightly different from traditional MCTS. We perform a search on the built MCTS tree where we take a sequence of actions with the highest number of visits until we reach the

"End Turn" action. This process is illustrated in Figure 8.

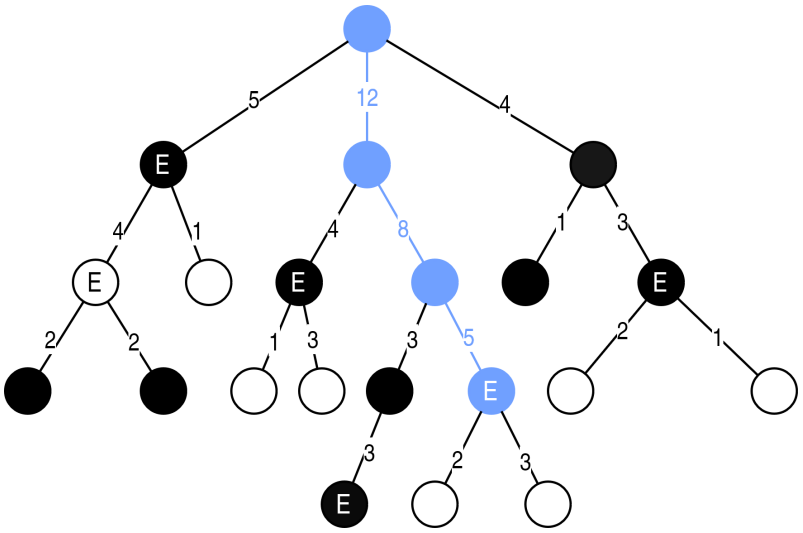

Figure 8. *A diagram demonstrating how the optimal series of actions is recovered from MCTS with varying length turns. The numbers on the connections represent the number of times the nodes have been visited. The blue highlight shows the optimal sequence of actions.*

Determinization of Random Outcomes Via Random Nodes

One of the challenges in implementing MCTS into the game was addressing random events. Imperfect information is introduced into our game through an action called "Action Refill" and at the start of each new turn. The game engine generates a random number to decide which outcome is chosen.

The solution used to address this is Determinization. Whenever a node that generates a random outcome is encountered in the expansion phase, we rely on specialized logic. In this special expansion phase, instead of adding nodes that represent the possible options of actions that can be taken, we first add nodes for each possible random outcome, as shown on the left side (a) in Figure Figure 9. We will refer to these special nodes as Random Nodes.

During the selection phase, the UCT algorithm is not used to evaluate the Random Nodes. In this case, it does not make sense to favor one random outcome over another if they are equally likely. During the MCTS selection phase, all random outcomes are sampled the same number of times. To achieve this, the selection policy picks the random node with the fewest visits.

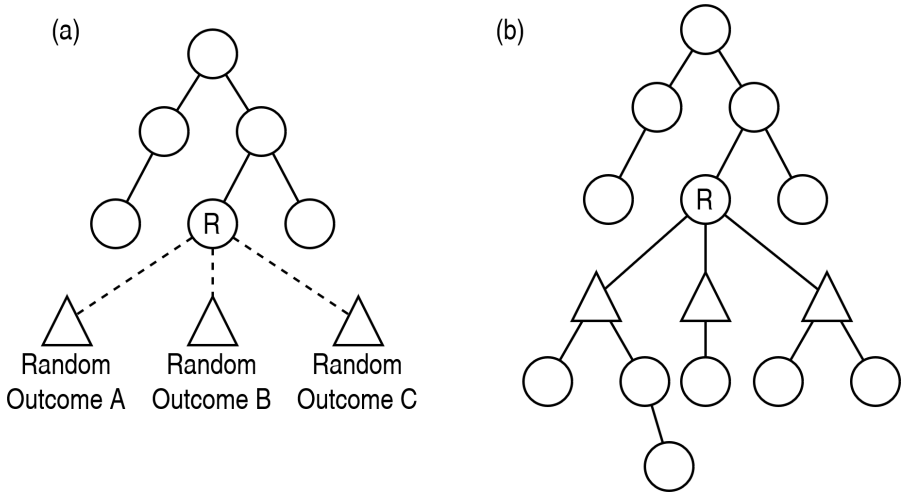

Figure 9. *Determinization with Random nodes in MCTS. (a) An action with a random outcome is marked with "R", the triangles represent random nodes corresponding to each of the possible outcomes. (b) After expansion of the random nodes, the tree structure continues as before.*

Recovering Actions After Random Outcomes

A unique case that can occur when recovering a sequence of actions is when the sequence includes a random event with multiple branches of random nodes. In this case, the recovery logic simply observes the true game state as it executes actions and determines which Random Node in the generated tree matches the outcome. After the correct branch is chosen, the action sequence recovery proceeds as usual.

3.1.6 Generating Starting Conditions For Self-Play

Chess and many other board games all start the same, with the board in a standard starting state. However, for games where there are numerous possible scenarios that can play out in different contexts, we need to simulate this variety of starting conditions. This is achieved through randomly generating the starting conditions for every self-play instance. This is done at a higher level using GDScript, as generating the initial conditions is only done once per self-play game, so there is little performance gain from doing it in GDExtension with C++.

3.1.7 Tournament

Tournaments is a new system created specifically for testing and evaluating actors within the context of a new game. Since there is no prior knowledge about the intended way to play the game and optimal strategies, evaluating the algorithms can become difficult. The tournament approach involves pitting actors against one another and saving the results of

each game. Similar to self-play, the tournament relies on the same logic to initialize the starting conditions for each game played.

However, each randomly generated starting state is played twice, with each actor playing as both sides. This approach is taken because most of the time, the starting conditions favor one of the two actors. Unfortunately, this approach is not entirely perfect due to the inherent randomness of the game. There will be some noise in the results, and there is always a possibility that the more capable actor might lose more games due to bad luck.

To mitigate some of these drawbacks, an increased number of tournaments are run.

3.2 Training Server

The training server is responsible for building the model files used for self-play and training them using gradient descent. Python 3 is the language used for training models and running the training server, benefiting from its robust suite of libraries for data science and machine learning.

The high-level API used for developing and training the models is Keras with TensorFlow. Although alternative backends for Keras, such as PyTorch, exist, the choice was made based on personal preference and familiarity with the TensorFlow and Keras APIs. These alternatives are equally valid and can be used as long as they can be converted into the format required by ONNX Runtime on the game side, which the vast majority of popular machine learning frameworks support.

The conversion from TensorFlow models to ONNX models is performed using the Python-based library tf2onnx [28].

Not all data is used for training. The data loader samples a maximum of 7 turns per game to reduce overfitting on particularly long games. The turns sampled are chosen randomly. Reducing the number of sampled games also helps with memory usage during training.

3.2.1 Data Augmentation

Data Augmentation is a common technique in computer vision that can effectively generate more data from existing data [29]. Because the game board is symmetrical both horizontally and vertically, we can exploit this to improve generalization and reduce overfitting by applying appropriate augmentations to the data prior to training.

The data loader independently applies a horizontal flip and vertical flip, each with a 50% probability. These flips are applied to the board input, mask input, and true policy output. For a single training sample, either all of these components are flipped or none of them are.

3.3 Neural Network Architecture

The NN architecture is heavily based on the AlphaZero DCNN. However, several important changes have been made to both the network and the training process. We refer to this modified model architecture as PlayerZero. A simplified diagram of the PlayerZero model architecture is shown in Figure 10.

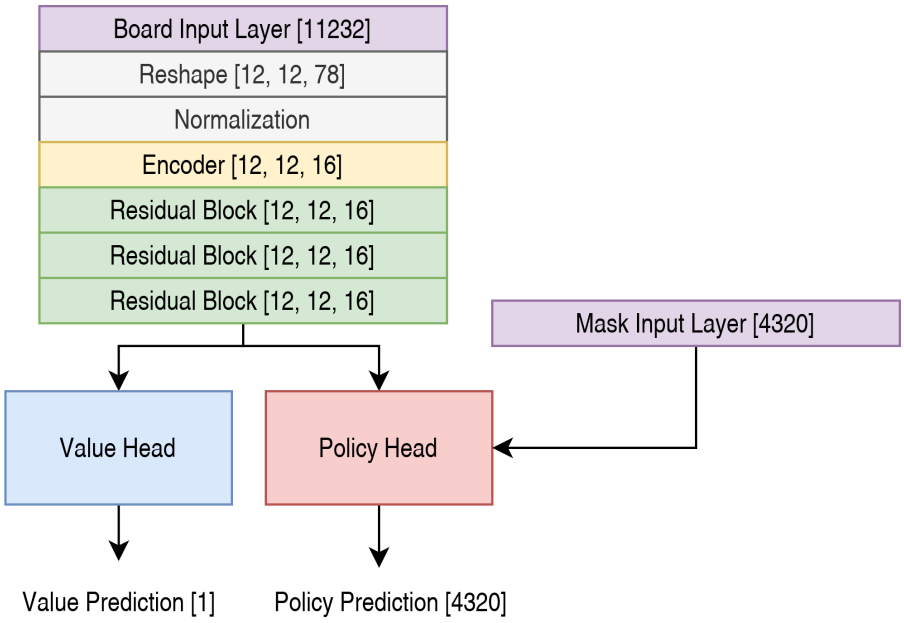

Figure 10. *The architecture of PlayerZero model. Numbers in square brackets represent the shape of data in the layers.*

3.3.1 Model Input And Output Format

The model input consists of two large vectors of values. The first vector represents the board or the game state. The second input is the policy mask, which exclusively contains only ones and zeroes, representing which actions are valid given the game state.

The output of the model is the same as the classic implementation of AlphaZero. The model determines which unit it is currently selecting an action for based on the "Is Controlled" flag on each cell. The value output ranges from -1 to 1, where -1 represents a loss, 0 represents a draw, and 1 represents a victory. The policy output is a probability distribution over all actions. Actions that are invalid given the state have a probability of 0.

In Figure 11, the format of the input and output is shown.

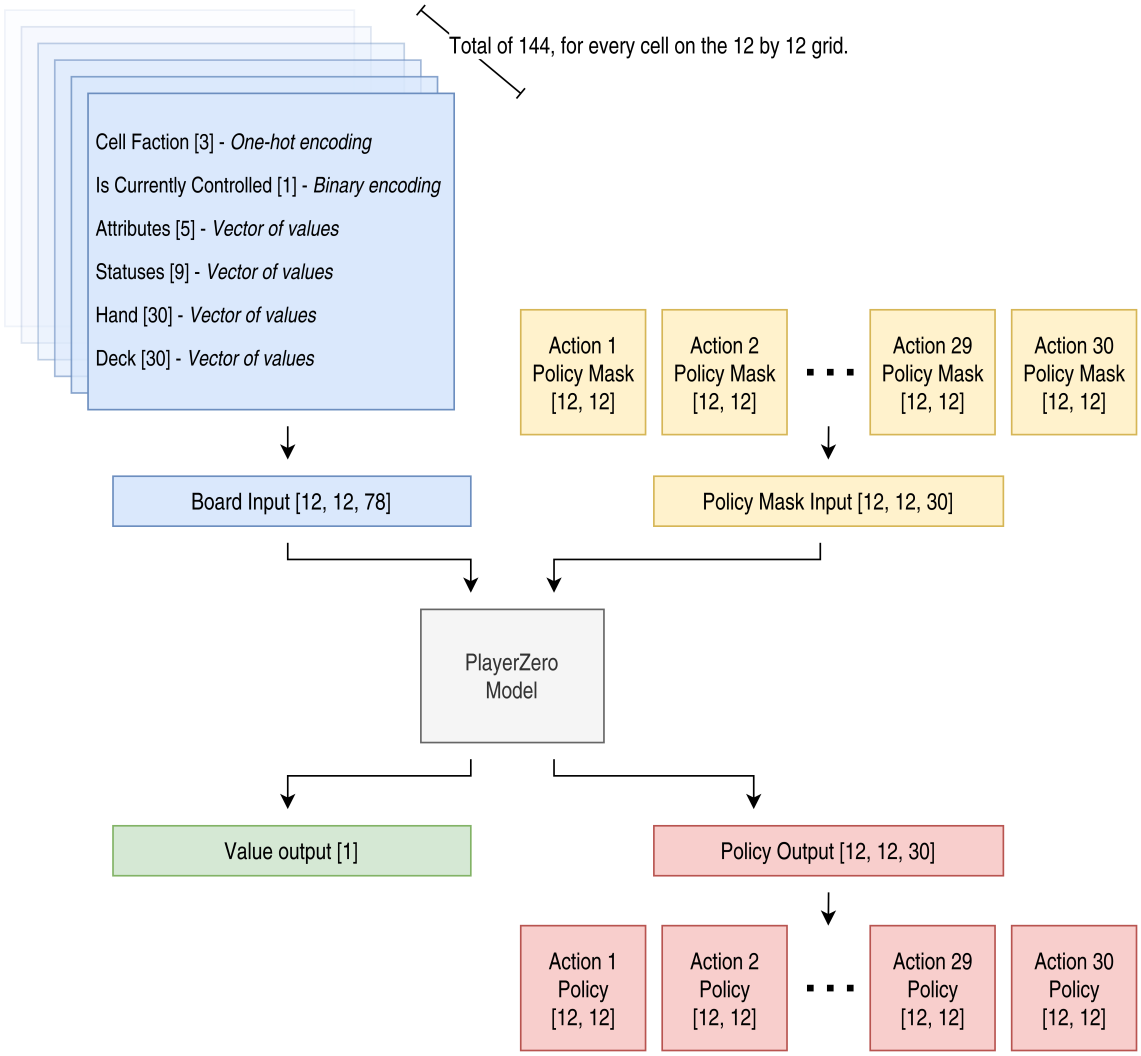

Figure 11. *The architecture of PlayerZero model.*

3.3.2 Addressing Large Number Of Features In Input With Encoder

Inspired by various types of encoder networks and word embeddings, the encoder layer transforms raw game data into densely encoded vectors. The encoder is implemented as a regular densely-connected NN layer [30]. However, instead of applying it to the game state input as a whole, it is applied to each position individually in the same manner. The weights used to encode all positions on the board are the same.

The initial feature vectors are quite large, with 78 features for each of the 144 positions on the grid, totaling 11232 features to represent the whole game state, with an additional 4320 features to represent the policy mask.

These encodings reduce the dimensionality of the initial features while capturing intricate

relationships between them.

Figure 12 illustrates the dense encoder layer of the PlayerZero DCNN.

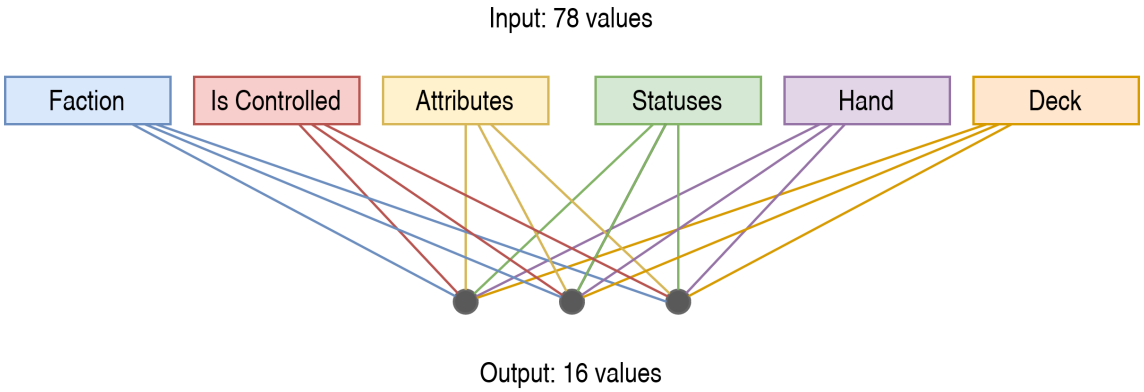

Figure 12. *The PlayerZero Input Encoder.*

3.3.3 Network Curriculum Learning

Network Curriculum Learning is a technique introduced in the training procedure for the NoGoZero+ algorithm to play NoGO. During training, two networks are trained: the smaller coach network and the larger student network. The coach network converges faster, while the student network takes longer to train. We use the smaller coach network to generate self-play data for training both networks. Once the coach network converges and can no longer be trained further, it is replaced with the student network. The student network then becomes the new coach, and a new, larger student network is created. [31]

Although not outlined in the original article that pioneered this idea, another benefit is that the coach network takes less time to perform inference, thus reducing the time required for the self-play phase compared to starting immediately with a large network. The network shown in Figure 10 is am example of the smallest initial coach network of scale 1. The table 1 displays various sizes of the PlayerZero network.

Table 1. *A Table representing how the PlayerZero network scales during NCL. A dedicated Python 3 building script is used to generate new sizes of PlayerZero directly from the 'scale' parameter.*

Scale	**Residual Blocks**	**Channels**	**Total number of parameters**
1	3	16	63,678
2	5	32	314,974

Continues...

Table 1 – *Continues...*

Scale	**Residual Blocks**	**Channels**	**Total number of parameters**
3	7	48	912,510
4	9	64	2,009,886
5	11	80	3,760,702

3.3.4 Training The First Generation From MCTS

Before we can begin Self-Play, we need to have a network. However, without generated data, the best network we can create would output either evenly distributed values or completely random values. In both cases, this network would not be able to play the game well. With enough generations, it would eventually learn the optimal strategy. However, by using an actor better than just random for Self-Play, we can skip some early struggles in the training process. Since we already have a simple MCTS implementation that can play the game, we propose to use it as our first actor for generating Self-Play data. This approach helps to speed up the initial training process for the first few generations and provides an immediate dataset with knowledge of how to play the game.

Figure 13 illustrates the data generation for training the different PlayerZero generations.

Despite the simplicity of the approach, an additional benefit of having this MCTS Self-Play was that it allowed for faster development of the PlayerZero model architecture. The same MCTS Self-Play data can be used to train multiple networks and then directly compare those networks' abilities to learn, by comparing model training metrics such as accuracy and loss.

3.3.5 Masking The Policy

The game state is not the only input the model receives; in our implementation, we also have a policy mask. The policy mask input matches the shape of the output policy; the values represent which actions are valid given the current board state. Valid actions are represented by ones, and invalid positions by zeros. This information is already available from the rules of the game, and we are simply passing it along to the network.

AlphaZero does not usually need this, as it can learn which actions are valid simply by training. However, in our game, the space of possible actions is larger than in Chess or

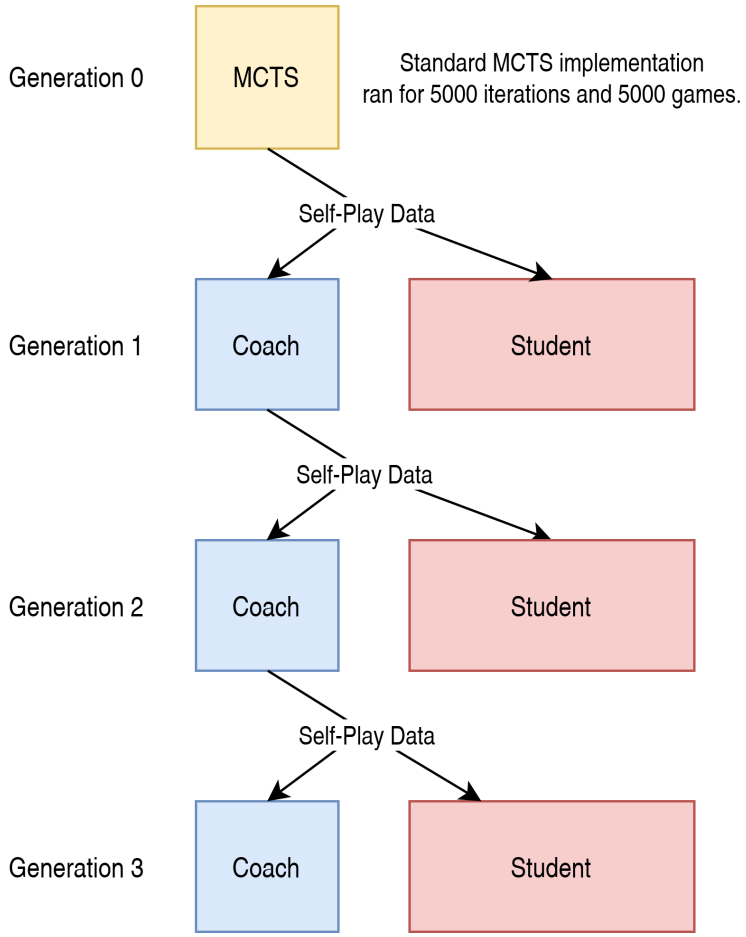

Figure 13. *First generation of PlayerZero is trained on data from MCTS Self-Play.*

Go, and the network itself is also smaller. Which actions can be cast is dependent on the hand attribute of the current unit. By providing the mask to the network, we make the task of predicting the policy significantly easier and increase its accuracy without additional training. Additionally, we can preserve a relatively low number of trainable model parameters.

Before the mask is applied it goes through a custom "Process Mask" layer that applies element-wise the following function to the values in the mask:

$$f(m_i) = -10^8 * (1.0 - m_i) \tag{3.1}$$

This function convets all values of 1.0 to 0.0, and values where there were initally 0.0 to a very large negative float. The resulting processed mask is then added element-wise to the output of the head. This process is very important, and the reason behind it is the Softmax activation function that follows it. Softmax activation function works such that it takes inputs in logits form, where their range is from negative infinity to positive infinity and

then maps them to the 0.0 to 1.0 range. Additionally, the sum over the output of a softmax activation function always adds up to 1.0, effectively creating a probability distribution. If a large negative value is added to the input, it is then interpreted by Softmax activation function as 0.0 probability, if 0.0 is added to the input, then the value remains as is.

Figure 14 illustrates the modified architecture of the policy head.

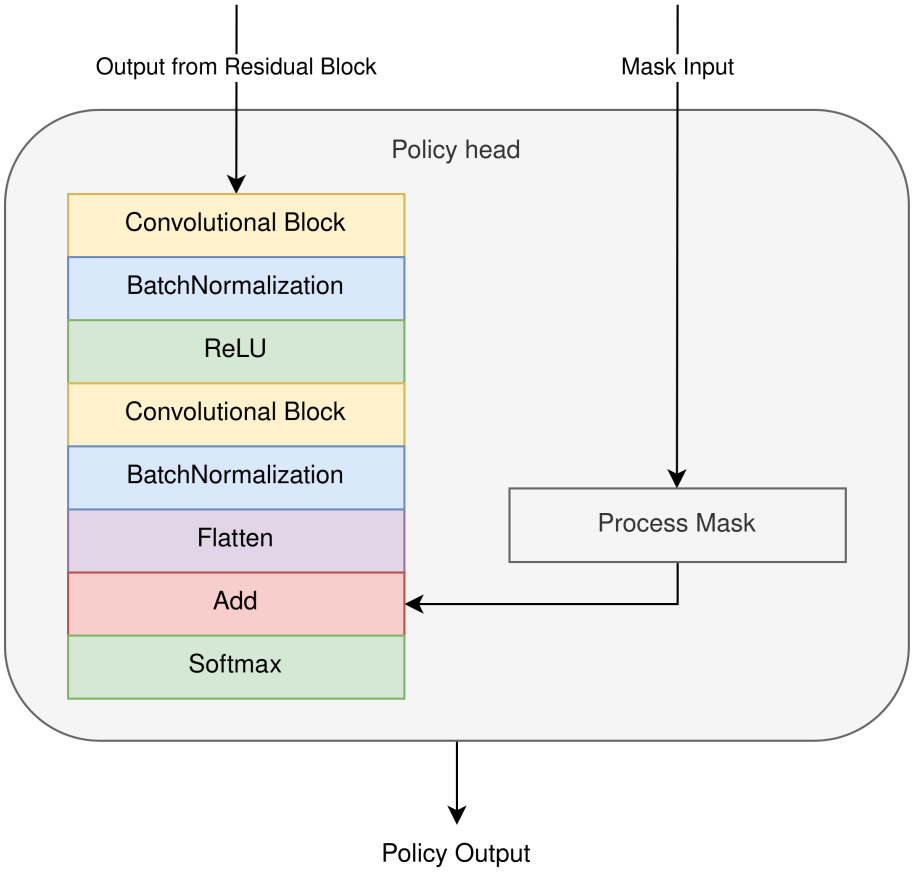

Figure 14. *Policy head architecture of PlayerZero*

3.3.6 Other PlayerZero NN Implementation Details

Other minor adjustments and techniques used in PlayerZero.

- All input features are automatically normalized by the NN.
- 32-bit floats were used instead of Pythons 3 default 64-bit to reduce VRAM usage.
- L2 Regression was used to add a penalty term to the loss function.
- Learning Rate Scheduler was used to automatically lower the learning rate during gradient decent by a factor after each step.

3.4 Communication Between Self-Play Client And Training Server

The communication between the Self-Play Client and Training Server is facilitated using a TCP connection. Both GDScript and Python 3 have built-in functionality to achieve this with minimal code. The two components communicate via packets of a set size, performing a series of handshakes.

3.4.1 Self-Play Data Format

The Self-Play Data is stored on disk in a custom compact format to represent the game state. An example of a single Self-Play data file is provided in Appendix 6, along with additional details about the meaning of the data. These files are generated by the Self-Play Client and then read from the RAM Disk by the Training Server.

3.4.2 Temporary Data Storage On RAM Disk

A RAM Disk offers a solution for storing temporary data that needs to be quickly read and written frequently. The read-write speeds of RAM Disks are orders of magnitude higher than traditional storage devices. Unlike storage devices, RAM does not wear down from repeated writes, allowing the hardware to last longer [32]. The data is stored in a serialized compact format, and only 0.5 GB of 32 GB of RAM was allocated as a RAM disk. For reference, information from 5000 Self-Play Games on average takes up 100 MB.

The RAM Disk was used to save the results of self-play for training purposes. After the data is used for training and a new self-play routine is started, all the data is removed from the drive.

4. Results

4.1 Configuration Used For Attaining The Results

The information about the hardware used for training PlayerZero model is available in Appendix 4.

PlayerZero was trained for 6 generations, with the first model trained using the MCTS self-play data. All subsequent generations are trained on data generated by the previous generation. The models are trained for 150 steps with 256 batches. Each generation took approximately 3 hours to generate and train, except for the first generation, which took 6 hours due to the configuration of MCTS used. The Self-Play Client generated the data in parallel using 10 worker threads.

4.2 Model Training Metrics

Figure 15 shows the training metrics of PlayerZero.

From these training results, it can be seen that subsequent generations more accurately match the ground truth policy of their ancestors. The loss is also rapidly decreasing, indicating convergence of the output values.

A single important metric is omitted from these results: value accuracy. This is done because the best value accuracy does not noticeably increase from one generation to another, consistently converging at around 0.66. The reason behind this is at least in part due to the random nature of the game; the outcome given the board state can never be perfectly accurate due to the inherent randomness of the game.

4.3 Tournament Results

For every pair of algorithms, 160 different initial game scenarios are played. Each scenario represents one randomly generated initial game state, played until a winner is found or the game ends in a draw. Games that last over 64 turns are also counted as draws; however, most games end with a winner, as seen in the results.

The algorithms in the tournament are configured as follows:

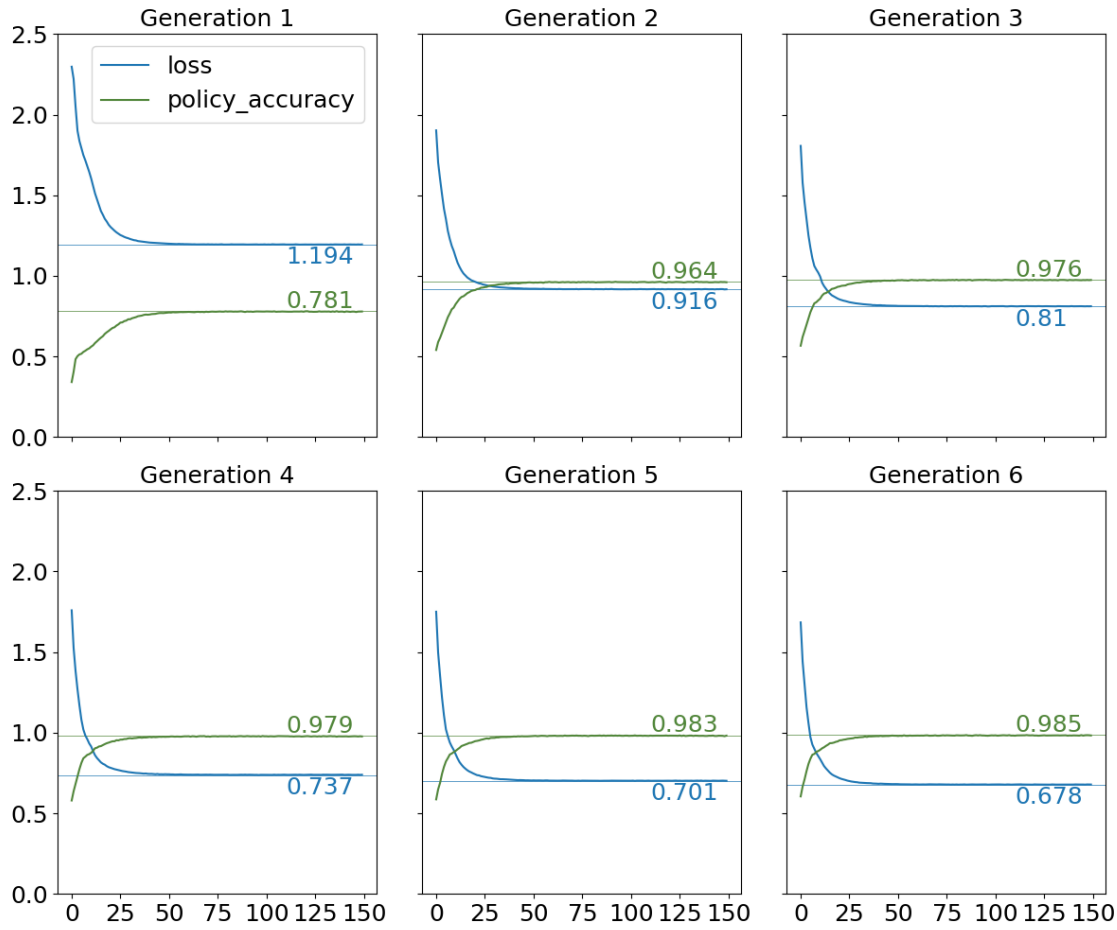

Figure 15. *Training of PlayerZero model generations. The X axis represent the training step (epoch). The blue line represents the sum of the losses of the policy and value prediction of the network (lower is better). The green line represents the accuracy of the predicted policy (higher is better). The colored text represents the best value for a given metric achieved during training the generation.*

- **MCTS 1000**: Standard MCTS implementation with 1000 iterations, and each simulation lasts a maximum of 32 turns.
- **MCTS 4000**: Similar to MCTS 1000, but with 4000 iterations.
- **PlayerZero 1**: Our algorithm after 1 generation of training using only MCTS Self-Play data.
- **PlayerZero 4**: Our algorithm after 4 generations of training, with 3 generations trained using reinforcement learning (RL).

Each scenario is played twice, so that every algorithm gets to play both sides.

Results of these tournaments are presented below in Table 2.

Table 2. *Tournament results. A1 and A2 are abbrevations for Algorithm 1 and Algorithm 2. Time taken is the average time required by the algorithm to perform 1 decision. The unit is miliseconds.*

Algorithm 1	**Algorithm 2**	**A1 Wins**	**A2 Wins**	**Draws**	**Time taken by A1 (ms)**	**Time taken by A2 (ms)**
MCTS 1000	MCTS 4000	139	159	22	15828	56806
PlayerZero 1	MCTS 1000	157	150	13	24130	19701
PlayerZero 1	MCTS 4000	150	157	14	24165	67277
PlayerZero 4	PlayerZero 1	164	140	16	28426	31147
PlayerZero 4	MCTS 1000	169	131	20	22968	20045
PlayerZero 4	MCTS 4000	163	138	19	22352	69141

The "Time Taken" is inflated due to being run in parallel, in pratices, when determining the optimal sequence of actions for a single turn, all algorithms take less than 0.7 seconds, with the exception of MCTS 4000, which takes approximately 2.5 seconds.

From the Table above 2, we are able to draw several conclusions, the first is that the PlayerZero 4 is the overall best performing algorithm, achieving higher winrate than all of the others. The second important observation is that the PlayerZero 1 algorithm is already able to outperform MCTS 1000 after just one generation of training. While MCTS 4000 is still better than PlayerZero 1, it is notable that MCTS 4000 also takes considerably more time.

5. Summary

The goal of this thesis was to explore an alternative approach utilising ML techniques for creating algorithms to control opponents in turn-based games. The approach taken utilised AlphaZero combined with multiple modifications.

The game logic was rewritten in C++, we added model inference functionality into the game enigne accelerated by CUDA, developed a Self-Play Client and Training Server components to facillitate RL. We introduced new optimizations to the algorithm, including additional model inputs in form of Policy Mask, modifications to the NN architecture in form of Input Encoder, and modified the training process by using MCTS to train the first generation.

The resulting algorithm was called PlayerZero, its performance was evaluated based on multiple metrics and compared to classic MCTS.

We show that it is indeed possible to apply these algorithms and techniques to a turn-based computer game with a more dynamic ruleset. More over, PlayerZero is able to outperform MCTS after just one generation of training while performing the same number of iterations. By the 4th generation of training, is still able to considerably outperform MCTS algorithms that are given more than twice the amount of time to make a decision.

The inital goal that was put forth in this thesis has been achieved.

References

[1] I. Millington and J. Funge. *Artificial Intelligence for Games*. CRC Press, 2016, pp. 293–306. ISBN: 9781498785815. URL: https://books.google.ee/books?id=OMb1CwAAQBAJ.

[2] *Divinity Engine Wiki: Combat AI*. [Accessed: 9-05-2024]. URL: https://docs.larian.game/Combat_AI.

[3] Google DeepMind. *AlphaGo*. [Accessed: 11-05-2024]. URL: https://deepmind.google/technologies/alphago/.

[4] Omid E. David, Nathan S. Netanyahu, and Lior Wolf. "DeepChess: End-to-End Deep Neural Network for Automatic Learning in Chess". In: *Lecture Notes in Computer Science*. Springer International Publishing, 2016, pp. 88–96. ISBN: 9783319447810. DOI: 10.1007/978-3-319-44781-0_11. URL: http://dx.doi.org/10.1007/978-3-319-44781-0_11.

[5] S. Branavan, David Silver, and Regina Barzilay. "Non-Linear Monte-Carlo Search in Civilization II". In: Jan. 2011, pp. 2404–2410. DOI: 10.5591/978-1-57735-516-8/IJCAI11-401.

[6] David Silver et al. *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*. 2017. arXiv: 1712.01815 [cs.AI].

[7] Maurice Bergsma and Pieter Spronck. "Adaptive Spatial Reasoning for Turn-based Strategy Games." In: Jan. 2008.

[8] *The Most Popular Video Game Franchises of All Time*. [Accessed: 7-05-2024]. URL: https://gamerant.com/biggest-best-selling-video-game-franchises-most-popular/.

[9] *The Game Awards 2023 Winners: The Full List*. [Accessed: 7-05-2024]. URL: https://www.ign.com/articles/the-game-awards-2023-winners-the-full-list.

[10] Ed Welch. *Designing AI Algorithms For Turn-Based Strategy Games*. [Accessed: 11-05-2024]. URL: https://www.gamedeveloper.com/design/designing-ai-algorithms-for-turn-based-strategy-games.

[11] *Divinity Engine Wiki: CombatAi Archetypes And Modifiers*. [Accessed: 14-05-2024]. URL: https://docs.larian.game/CombatAi_Archetypes_And_Modifiers.

[12] Clive Thompson. *What the history of AI tells us about its future*. [Accessed: 11-05-2024]. URL: https://www.technologyreview.com/2022/02/18/1044709/ibm-deep-blue-ai-history/.

[13] Qi-Yue Yin et al. "AI in Human-computer Gaming: Techniques, Challenges and Opportunities". In: *Machine Intelligence Research* 20.3 (Jan. 2023), pp. 299–317. ISSN: 2731-5398. DOI: 10.1007/s11633-022-1384-6. URL: http://dx.doi.org/10.1007/s11633-022-1384-6.

[14] Maciej Świechowski et al. "Monte Carlo Tree Search: a review of recent modifications and applications". In: *Artificial Intelligence Review* 56.3 (July 2022), pp. 2497–2562. ISSN: 1573-7462. DOI: 10.1007/s10462-022-10228-y. URL: http://dx.doi.org/10.1007/s10462-022-10228-y.

[15] Levente Kocsis and Csaba Szepesvari. "Bandit Based Monte-Carlo Planning". In: [Accessed: 02-05-2024]. URL: http://ggp.stanford.edu/readings/uct.pdf.

[16] Cameron B. Browne et al. "A Survey of Monte Carlo Tree Search Methods". In: *IEEE Transactions on Computational Intelligence and AI in Games* 4.1 (2012), pp. 1–43. DOI: 10.1109/TCIAIG.2012.2186810.

[17] Peter Cowling, Colin Ward, and Edward Powley. "Ensemble Determinization in Monte Carlo Tree Search for the Imperfect Information Card Game Magic: The Gathering". In: *Computational Intelligence and AI in Games, IEEE Transactions on* 4 (Dec. 2012), pp. 241–257. DOI: 10.1109/TCIAIG.2012.2204883.

[18] David Silver et al. "Mastering the game of Go without human knowledge". In: *Nature* 550 (Oct. 2017), pp. 354–359. DOI: 10.1038/nature24270.

[19] Thomas McGrath et al. "Acquisition of chess knowledge in AlphaZero". In: *Proceedings of the National Academy of Sciences* 119.47 (Nov. 2022). ISSN: 1091-6490. DOI: 10.1073/pnas.2206625119. URL: http://dx.doi.org/10.1073/pnas.2206625119.

[20] *Godot Engine - Free and open source 2D and 3D game engine*. [Accessed: 9-05-2024]. URL: https://godotengine.org/.

[21] *Most Used Engines*. [Accessed: 7-05-2024]. URL: https://itch.io/game-development/engines/most-projects.

[22] *Godot Engine 4.2 documentation: GDScript reference*. [Accessed: 9-05-2024]. URL: https://docs.godotengine.org/en/stable/tutorials/scripting/gdscript/gdscript_basics.html#doc-gdscript.

[23] *Bitboards*. [Accessed: 27-05-2024]. URL: https://www.chessprogramming.org/Bitboards.

[24] *Godot Engine 4.2 documentation: CPU optimization*. [Accessed: 9-05-2024]. URL: https://docs.godotengine.org/en/stable/tutorials/performance/cpu_optimization.html#gdscript.

[25] *Godot Engine 4.2 documentation: What is GDExtension?* [Accessed: 7-05-2024]. URL: https://docs.godotengine.org/en/stable/tutorials/scripting/gdextension/what_is_gdextension.html.

[26] *ONNX Runtime*. [Accessed: 12-05-2024]. URL: https://onnxruntime.ai/.

[27] *CUDA Toolkit*. [Accessed: 27-05-2024]. URL: https://developer.nvidia.com/cuda-toolkit.

[28] *tf2onnx - Convert TensorFlow, Keras, Tensorflow.js and Tflite models to ONNX*. [Accessed: 7-05-2024]. URL: https://github.com/onnx/tensorflow-onnx.

[29] Olga Chernytska. *Complete Guide to Data Augmentation for Computer Vision*. [Accessed: 14-05-2024]. 2021. URL: https://towardsdatascience.com/complete-guide-to-data-augmentation-for-computer-vision-1abe4063ad07.

[30] *Keras 3 API documentation: Dense layer*. [Accessed: 27-05-2024]. URL: https://keras.io/api/layers/core_layers/dense/.

[31] Yifan Gao and Lezhou Wu. "Efficiently Mastering the Game of NoGo with Deep Reinforcement Learning Supported by Domain Knowledge". In: *Electronics* 10 (June 2021), p. 1533. DOI: 10.3390/electronics10131533.

[32] Tobias Kind / FiehnLab 2009 / fiehnlab.ucdavis.edu. *RAMDISK Benchmarks WIN7*. [Accessed: 9-05-2024]. URL: https://fiehnlab.ucdavis.edu/staff/kind/collector/benchmark/ramdisk.

Appendix 1 – Non-Exclusive License for Reproduction and Publication of a Graduation Thesis[1]

I Matvei Tarassov

1. Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis "Analysis of machine learning techniques for the development and implementation of artificial intelligence in a turn-based game", supervised by Gert Kanter
   1.1. to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;
   1.2. to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright.
2. I am aware that the author also retains the rights specified in clause 1 of the non-exclusive licence.
3. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

27.05.2024

[1]The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.

Appendix 2 – Github Reposiotry

Link to the Github repostory with the code for the project `https://github.com/Klazkin/player-zero/`.

Appendix 3 – Game Rules

Introduction

The game is played on a 2-dimensional gird that is 12 by 12 cells. Each player controls one of the two factions and has access to 1 or more Units. A Unit is similar to a chess piece on a board, and factions are analagous to sides or teams. The game ends if only units of one faction remain on board. It is possible for no factions to remain on the board, in that case the game is counted as a draw. Players interact with the board through their controlled units, each unit has certain actions they can perform.

Unit Attributes

Each unit has a list of attributes that affect their combat capabilities.

- Maximum Health - Units maximum health value. If a unit is healed, the health cannot go above this value.
- Current Health - Units current health value, Unit is considered dead if it reaches 0 and is permanently removed from the game.
- Attack - Affects outgoing damage.
- Defence - Affects incoming damage.
- Speed - Used to determine the order of turns, Units with higher speed go first.
- Deck - A list of Actions that the Unit has access to during the game.
- Hand - A list of Actions immediately available to the unit.
- Status Effects - A list of temporary afflictions, effects vary based on the exact status.

Flow of the Game

The order in which units get to act depends on their Speed attribute, higher speed means that units act first. If there is a speed tie then units of Faction 1 act first.

During the units turn they are able to use any action that is in their hand, the action "End Turn" is always present. The Unit may use up all of their actions in a single turn, or none of them. Certain actions can be combined into new, more powerful actions with unique effects. The Unit gets new actions from their Deck, at the start of every turn the Unit get 1

random action from the deck. The unit may hold only 1 of each action in their hand, and if the unit already has every action then they do not recieve anything at the start of the turn.

Actions

Actions which deal damage to their targets use the following formula:

$$d = max(0, a_d + u_a - u_d) \tag{1}$$

Where:

- d is the final damage.
- a_d is the action damage.
- u_a is the attacking units "attack" attribute.
- u_d is the "defence" attribute of the unit recieving the attack.

List Of All Actions

- **Move**
  ID: 0
  Combination recipe: None
  Description: Allows the unit to move to any neighbor cell within a radius of 5 cells centered around the Unit.
  Note: This action is always restored at the start of the turn, along with 1 other action from the Units deck.
- **Sword**
  ID: 1
  Combination: None
  Description: Strikes all Units of opposing faction within a cross shaped area in-front of the Unit. Deals 4 damage points to the target.
- **Bow**
  ID: 2
  Combination: None
  Description: Shoots an arrow at a single Unit. Deals 1 damage point plus 1 extra for every 2 cells of distance between the Unit and the target. Can only be used if there is a clear line-of-sight between the two Units.
- **Spear**
  ID: 3

Combination: None

Description: Strikes all Units of opposing faction within a straight line area in-front of the Unit. Deals 4 damage points to the target.

- **Spark**

  ID: 4

  Combination: None

  Description: Deals 3 points of damage to a single selected target with 5 cell radius around the Unit. Applies a Burn status to the target which lasts 3 turns and lowers defence by -1. While the Burn status is active the target loses 2 health at the start of every single turn.

- **Ground**

  ID: 5

  Combination: None

  Description: Creates a stone wall in the selected neighbor cell. The cell becomes unoccupiable and the wall blocks line-of-sight checks. The wall can be destroyed by actions which deal damage.

- **Action Drawing**

  ID: 6

  Combination: None

  Description: Deals 3 damage points to the Unit which used this action but immediately refills the Move action and one other random action.

- **Weaker Spark**

  ID: 7

  Combination: None

  Description: Deals 3 points of damage to a single selected target with 4 cell radius around the Unit. Can only be used if there is a clear line-of-sight between the two units.

- **Dust**

  ID: 8

  Combination: None

  Description: Applies Dusted status effect to all units within 4 radius centered around the Unit for 2 turns. Dusted status effect lowers the speed of the afflicted Units by -2.

- **Slow Heal**

  ID: 9

  Combination: None

  Description: Applies Slow Healing status effect to the target Unit (can be used on your own Unit) for 4 turns. Slow healing restores 2 health points to Unit at the start of every turn while it is active.

- **Damage Chain**

ID: 11

Combination: Obtained by combining "Action Drawing" and "Slow Healing".

Description: Applies Chain status effect to the target unit for 3 turns. When your unit is attacked, the target unit also recieves the damage.

- **End Turn**

  ID: 13

  Combination: None

  Description: Ends the turn of the current Unit. Note: This action is always available to the unit.

- **Combine Actions**

  ID: 13

  Combination: None

  Description: Allows to combine two specific actions into one, the actions used for combining are lost and replaced with 1 new action. Note: This action does not disappear after use.

- **Position Swap**

  ID: 15

  Combination: Obtained by combining "Move" and "Action Drawing".

  Description: Swaps the Units position with the targets position on the board.

- **Dust Spark**

  ID: 19

  Combination: Obtained by combining "Weaker Spark" and "Dust".

  Description: Deals 10 points of damage to all units within 4 radius centered around the Unit, if those units are afflicted with "Dusted" status effect. If the units do not have the status effect, this action does nothing.

- **Spark Dive**

  ID: 20

  Combination: Obtained by combining "Move" and "Spark".

  Description: Instantly teleports the Unit to an empty cell near the target, the target is also hit with the "Spark" action.

- **Shatter**

  ID: 21

  Combination: Obtained by combining "Ground" and "Spark".

  Description: Deals 5 damage to the chosen target, if this action kills the target, then additional 7 damage points is dealt to all targets within 3 radius centered around the initial strike. The secondary effect also triggers if the target is a wall created by "Ground" action.

- **Shield**

  ID: 22

Combination: Obtained by combining "Ground" and "Slow Healing".
Description: Applies Shield status effect to the selected ally Unit for 3 turns (can be cast on self). The Shield status effect lowers speed attribute by -1 and nullifies one attack, if it disappears immediately after blocking.

- **Boost**
  ID: 23
  Combination: Obtained by combining "Spark" and "Action Drawing".
  Description: Removes 50% of current health from the selected ally Unit and applies Boost status effect to it for 4 turns. The Boost status effect increases speed by +1 and attack by +4, but lowers defence by -1. When the Boost status effect ends, it heals back the 50% of health that it removed.
- **Armor**
  ID: 26
  Combination: Obtained by combining "Move" and "Ground".
  Description: Applies Armor status effect to the selected ally Unit for 3 turns (can be cast on self). The Armor status effect increases defence by +2.
- **Fast Heal**
  ID: 27
  Combination: Obtained by combining "Move" and "Slow Healing".
  Description: Immediately restores 5 health to the Unit which used this action.

Appendix 4 – System Configuration

Software

- Operating System: Windows 10 Pro N
- Windows Subsystem for Linux (WSL) version: 2.1.5.0, Ubuntu 22.04.1 LTS
- Hardware Acceleration: CUDA 12.1

Hardware

- GPU: Nvidia GTX 1080Ti
- CPU: AMD Ryzen 7 5800X3D 8-Core Processor 3.40 GHz
- RAM: 32GB @ 3200 MHz, 0.5GB of which are dedicated to the RAM Disk

Appendix 5 – Combat AI Modifiers

Table 3. *Modifiers used to calculate the score of an action in Divinity 2 Original Sin. This a small subset of the full table.*

Modifier	**Base Value**	**Description**
MULTIPLIER_TARGET_MY_-ENEMY	1.50	Used when the target of the action is attacking the source character
MAX_HEAL_MULTIPLIER	0.50	A multiplier to determine the importance of healing someone that's close to dying. Has to be in between 0.00 and 1.00
MULTIPLIER_MOVEMENT_-COST_MULTPLIER	0.9	Used in the ActionCostModifier explained in MULTIPLIER_FREE_ACTION. MULTIPLIER_MOVEMENT_COST_-MULTIPLIER is multiplied with the AP cost of the movement needed to execute the action and added to the ActionCostModifier
MULTIPLIER_INVISIBLE_-MOVEMENT_COST_MULT-PLIER	0.30	If the player is sneaking or invisible this modifier is used on top of MULTIPLIER_-MOVEMENT_COST_MULTIPLIER
MULTIPLIER_HEAL_SELF_-POS	1.0	Healing score on self that's considered positive (this is just healing)
MULTIPLIER_ENDPOS_-HEIGHT_DIFFERENCE	0.002	Used when calculating the PositionScore. MULTIPLIER_ENDPOS_HEIGHT_-DIFFERENCE is multiplied with the height difference between the current and the new position and added to the PositionScore

Appendix 6 – Self-Play Data Example

Example of a Self-Play data file generated by the 3rd generation of the PlayerZero algorithm. In this case the game only lasted for only 3 turns. Some of the rows are shortened to be more readable.

```
3
9,10,1,0,0,0,3
11,11,0,1,0,0,8,16,0,-1,-2,2,0,0,0,0,3,0,0,0,0,0,0,0,0...
9,11,0,0,1,1,2,20,2,-2,1,2,0,0,0,0,0,2,0,0,0,1,0,0,0,0...
5
9,11,13,0.188291,0.00857259,447,-0.999907
9,11,23,0.0970233,0.000926171,229,-1
10,11,1,0.425755,0.797224,1607,-0.991381
9,10,1,0.161266,0.0858417,398,-0.999072
8,11,1,0.127665,0.107436,318,-0.998838
2
11,11,0,1,0,0,0,1,16,0,-1,-2,2,0,0,0,0,3,0,0,0,0,0,0,0,0...
9,11,0,0,1,1,2,20,2,-2,1,2,0,0,0,0,0,2,0,0,0,0,0,0,0,0...
2
9,11,13,0.683822,0.999061,1223,-0.989345
9,11,23,0.316178,0.000938957,383,-1
2
11,11,0,1,0,1,1,16,0,-1,-2,1,0,0,0,0,2,0,0,0,0,0,0,0,0...
9,11,0,0,1,0,2,20,2,-2,1,2,0,0,0,0,0,2,0,0,0,0,0,0,0,0...
2
11,11,13,0.243148,0.661426,2957,1
11,11,9,0.756852,0.338574,42,0.0237636
0
0
0
0
```

How To Read The Self-Play Data

The first row of the data represents K number of objects in the game state, then K rows follow, each row representing the either a charahter or a wall. The first two integers are the x and y coordinates of the object on the grid, the rest are the attributes. The total row

to represent characters is 80 integers long. Exactly one of the characters also has an *"Is Controlled"* flag value set to 1, the rest have it at 0. This defines which unit is currently taking action. After that follows another number N, representing the number of possible actions that the current unit can take. N rows of actions follow, each action is represented by 7 values. Some of the values are no longer used or used exclusively for debugging and testing.

The meaning of the values is described below:

1. The x coordinate of the target for the action.
2. The y coordinate of the target for the action.
3. The id used to identify the action.
4. The initial policy as predicted by the PlayerZero DCNN (not used for training, only debugging).
5. The old value used for improved policy based on score (is no longer used).
6. The visits count, used to generate improved policy.
7. The average score of the action (not used for training, only debugging).

The pattern of K rows and N rows repeats until the game ends, which is signfied by two rows of zeroes. The last row in the file, which in this case also happened to be zero, represents the winner of the game.

The possible values for the last row are:

- -1 - Game ended with a Draw.
- 0 - Game ended with Faction 1 victory.
- 1 - Game ended with Faction 2 victory.