Deep learning applied to turn-based board games

Aprendizaje profundo aplicado a juegos de tablero por turnos

Universidad Complutense de Madrid

Facultad de Informática

Trabajo Fin de Grado del Doble Grado en Ingeniería Informática-Matemáticas

Curso 2020/2021

Pablo Sanz Sanz
Juan Carlos Villanueva Quirós
Dirigido por: Antonio A. Sánchez Ruiz-Granados

Resumen

Gracias al ritmo vertiginoso al que crece la capacidad computacional, la inteligencia artificial está logrando hitos que hace tan solo unas décadas se consideraban impensables. Uno de ellos es AlphaZero, un algoritmo capaz de alcanzar un nivel de juego sobrehumano en ajedrez, shogi y Go, mediante unas pocas horas de autoaprendizaje y sin conocimiento del dominio excepto las reglas del juego.

En este trabajo, revisamos los fundamentos, explicamos cómo funciona el algoritmo y desarrollamos nuestra propia versión de este, capaz de ser ejecutada en un ordenador personal. A pesar de la escasez de recursos computacionales disponibles, hemos conseguido dominar juegos menos complejos como el Tres en Raya y el Conecta 4. Para verificar el aprendizaje, probamos nuestra implementación contra otras estrategias y analizamos los resultados obtenidos.

Palabras clave

Inteligencia artificial, AlphaZero, árboles de búsqueda de Monte Carlo, aprendizaje por refuerzo, aprendizaje profundo, juegos combinacionales.

Abstract

Due to the astonishing growth rate in computational power, artificial intelligence is achieving milestones that were considered as inconceivable just a few decades ago. One of them is AlphaZero, an algorithm capable of reaching superhuman performance in chess, shogi and Go, with just a few hours of self-play and given no domain knowledge except the game rules.

In this paper, we review the fundamentals, explain how the algorithm works, and develop our own version of it, capable of being executed on a personal computer. Despite the lack of available computational resources, we have managed to master less complex games such as Tic-Tac-Toe and Connect 4. To verify learning, we test our implementation against other strategies and analyze the results obtained.

Keywords

Artificial intelligence, AlphaZero, Monte Carlo tree search, reinforcement learning, deep learning, combinational games.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

When IBM's Deep Blue beat chess grandmaster Garry Kasparov in 1996, the intellectual supremacy of humankind was definitively called into question. Since the last decades, the rapid development in computing power has permitted Artificial Intelligence to achieve superhuman performance on many tasks. Problems that were once considered as unapproachable for computers, such as board games like chess and Go, are now being solved with simplicity. Steadily, the gap between human and machine intelligence is narrowing.

Much effort has been expended trying to create programs capable of playing at a superhuman level in such hard games. This is because it is usually necessary to search among all possible moves and the branching factor of these games makes it unfeasible in a reasonable amount of time. Thus, typically these kinds of programs use the Minimax algorithm with an AlphaBeta prune, a highly fine-tuned heuristic with handcrafted features and many other domain-specific efficiency improvements.

In the case of chess, Stockfish [1] is one of the strongest engines at the moment. In its twelfth version, which was released in 2020, it incorporated a neural network for position evaluation. But before they used handcrafted features in the heuristic. Besides, it uses a big transposition table (a board cache) or an opening book, among others, to make the search more efficient.

As for the game of Go, most of the programs were unable to compete with professional human players. The reason behind this is mainly the size of the board (19×19, whereas a game like chess has a board of 8×8) and the branching factor. For instance, a typical position in Go may have around 250 possible moves.

However, in 2016 DeepMind, Google's notorious artificial intelligence company, created AlphaGo [2]. It is an algorithm based on a neural network that was trained using both human games and self-play, as well as some other game-specific enhancements. For example, they exploited the fact that Go has a symmetric board. AlphaGo supposed a milestone in Go. It faced South Korean 9-dan player Lee Sedol, one of the strongest players, and defeated him with four wins out of five games [3]. This match was streamed live in *YouTube* and reached around 60 million viewers in China.

This group continued its research on the topic and published an improved version of the algorithm, AlphaGo Zero [4]. This time, no human games where used for training, which showed that they were not necessary. This algorithm clearly outperformed the previous

one.

Finally, in 2017 they developed AlphaZero, a general reinforcement learning algorithm which outperforms humans in these challenging games (chess, shogi and Go). But the most relevant achievement was that AlphaZero started its training from random play and no domain-knowledge was provided except the game rules. It purely learnt by self-playing and achieved superhuman level of play within 24 hours. AlphaZero was capable of thoroughly beating Stockfish (the eighth version) in chess, AlphaGo Zero in Go and Elmo, a strong shogi engine.

In this project, we will explain how algorithms applied to turn-based board games work and, in particular, how AlphaZero accomplished such astonishing results. We will also implement our own version of AlphaZero, capable of being executed in a personal computer. Unfortunately, we could not afford powerful computational resources and for this reason, we will focus in less complex games such as Tic-Tac-Toe and Connect 4. In order to verify that our implementation is learning by self-play and with no domain knowledge, we will test it against other implemented algorithms and analyze the results obtained.

1.1 Objectives

We now list the main objectives that were intended to be achieved for this project:

- Acquire a fully comprehension on how AlphaZero algorithm works. In addition, we are going to study other algorithms for games, like Minimax. This way, we can compare AlphaZero with different approaches.
- Develop and implement our own version of AlphaZero. We will also implement the rest of algorithms we are studying.
- Train and test our AlphaZero version as far as possible. For this purpose, we will use the other algorithms as rivals to check if AlphaZero can outperform them.

1.2 Planning

We divided the proposed objectives into tasks and assign deadlines to each one.

The first objective was split into three main tasks: acquire a basic foundation in reinforcement learning, study the two main components used in AlphaZero (Monte Carlo Tree Search and Convolutational Neural Networks), and put all these pieces together to fully understand the complete algorithm. We assigned two months for this research period.

Secondly, we also divided the implementation part. We distributed the work so that each one of us could focus on implementing a specific module. More time was reserved for this part, as we had to code everything from scratch and adapt many concepts to fit into the computational power of a personal computer. Altogether, three months were planned to be spent for the implementation.

The third and last objective was divided into two tasks: debugging and testing. Debugging would consist of ensuring every module worked as expected. While testing, we would obtain all the results and conclusions. We planned to start with a basic game, such as Tic-Tac-Toe and then increase the difficulty if we got positive results. We expected that

we would be able to train a perfect Tic-Tac-Toe player. If we achieved that, we would try it with Connect 4. We hoped we could reach at least a player that is better than an average human. After that, if we continued having the expected results we might try it with harder games, like checkers. In total, two months were assigned for this part.

Finally, we planned to write this document progressively as we progress in the project.

Additionally, in order to ensure proper progress, we would fix a meeting every two weeks with our supervisor, so that he could check our work and provide new ideas. During the last month, we met every week instead to finish the last details.

1.3 Memory Structure

In Chapter 2 we cover the fundamentals that are later going to be applied in the explanation and development of AlphaZero. We explain the concept of reinforcement learning and introduce the Markov Decision Process as the formal environment that we are dealing with and the multi-armed bandit problem as an example of it. Then, we present search-based algorithms used by computers to play games. In particular, we study in depth the Monte Carlo Tree Search algorithm. In addition, we provide a general overview of the concepts of deep reinforcement learning that will be used later.

Chapter 3 thoroughly describes the AlphaZero algorithm. We divide the explanation in three well distinguished parts: training loop, self-play and neural network. Furthermore, we provide an execution example in great detail to facilitate understanding.

In Chapter 4 we explain how we implemented our version of AlphaZero. We give details of every module implemented and mention the libraries that were necessary. We also state all the problems that we had to face and how we solved them.

In Chapter 5 we show the experiments carried out with AlphaZero for Tic-Tac-Toe and Connect 4. Specifically, we train multiple instances of AlphaZero with different parameters, choose the most appropriate ones and test them against other strategies.

Finally, in Chapter 6 we summarize the conclusions obtained from this project. We mention which objectives has been achieved and what further work could be done if we had enough time.

Chapter 2

Algorithms in Games

2.1 Reinforcement Learning

We often say there are three paradigms in machine learning: supervised learning, unsupervised learning and reinforcement learning [5]. On the one hand, supervised learning aims to build a model capable of learning to label input data based on already annotated examples. On the other hand, unsupervised learning tries to find some kind of structure of unlabeled data. As their names suggest, we may think that these first two paradigms cover the whole spectrum of machine learning. However, reinforcement learning cannot be classified either as supervised or unsupervised learning, even though it has some things in common with both of them.

Briefly, reinforcement learning tries to find the actions an agent should take in a certain environment. The main difference between reinforcement learning and the other two paradigms is that finding the training data is part of the problem, whereas in both supervised and unsupervised learning it is given beforehand. In order for the agent to know whether an action is beneficial or not, it is given a reward after taking it. Therefore, the goal of the agent is to maximize the acquired reward.

A reinforcement learning problem consists of two main entities: the agent and the environment. The interaction between them is as follows: the agent sees a state s_t at time t, takes an action a_t and the environment returns its new state s_{t+1} and the reward r_{t+1}. This process may be repeated indefinitely.

Formally, as David Silver does in [6], we may describe a reinforcement learning problem as a Markov Decision Process (MDP), that is, a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$ where:

- $\mathcal{S}$ is a finite set of states,
- $\mathcal{A}$ is a finite set of actions,
- $\mathcal{P}\colon \mathcal{S} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$ is a state transition probability function

$$\mathcal{P}(s, s', a) = \mathbb{P}[S_{t+1} = s' \mid S_t = s, A_t = a],$$

- $\mathcal{R}\colon \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the reward function

$$\mathcal{R}(s, a) = \mathbb{E}[R_{t+1} \mid S_t = s, A_t = a]$$

- and $\gamma \in [0, 1]$ is the discount factor.

Here, $\mathbb{P}[S_{t+1} = s' \mid S_t = s, A_t = a]$ means the probability of ending in state s' at time $t+1$ given that the actor is in state s and takes action a at time t; and $\mathbb{E}[R_{t+1} \mid S_t = s, A_t = a]$ is the expectancy of receiving reward R_{t+1} if the actor takes action a in state s at time t.

The kinds of problems we are going to cover are deterministic, so if the agent is in state s and takes action a there will only exist one state $s' \in \mathcal{S}$ such that $\mathcal{P}(s, s', a) = 1$ and therefore $\mathcal{P}(s, s'', a) = 0$ for all $s'' \in \mathcal{S} \setminus \{s'\}$. Thus, we can treat $\mathcal{P}$ as a transition function $\mathcal{P} \colon \mathcal{S} \times \mathcal{A} \to \mathcal{S}$, $(s, a) \mapsto \operatorname{argmax}_{s' \in \mathcal{S}} \mathbb{P}[S_{t+1} = s' \mid S_t = s, A_t = a]$, where argmax is the mapping that returns one argument that maximizes the given function. We will also consider a deterministic reward function.

Now a reinforcement learning agent takes actions following a given policy. The policy function $\pi \colon \mathcal{A} \times \mathcal{S} \to [0, 1]$ is a mapping from a state to the probabilities of taking each possible action. Thus, $\sum_{a \in \mathcal{A}} \pi(a \mid s) = 1$ for every $s \in \mathcal{S}$. With this policy, the agent will try to maximize its reward, given by the value function $v_\pi \colon \mathcal{S} \to \mathbb{R}$ that maps a state s to the expected return after following the policy π starting from s. More formally,

$$v_\pi(s) = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \;\middle|\; S_t = s \right], \tag{2.1}$$

where t is any time step and $\mathbb{E}_\pi$ is the expectancy operator provided that policy π has been followed. Similarly, we could define the action-value function $q_\pi \colon \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ instead:

$$q_\pi(s, a) = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \;\middle|\; S_t = s, A_t = a \right]. \tag{2.2}$$

Here is where the discount factor γ comes into play. A low value of γ will only consider important immediate rewards, whereas a high value will take also into account long-term ones. For instance, if $\gamma = 0$ the agent will be called greedy, and if $\gamma = 1$ it will try to maximize all possible rewards. But it is important that there is a finite horizon in the second case because the sum will not converge otherwise. Usually, $\gamma \in (0, 1)$, which make the series convergent if all possible rewards are bounded by a certain value $R > 0$:

$$\left| \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \right| \leq \sum_{k=0}^{\infty} \gamma^k R = \frac{1}{1-\gamma} R < \infty.$$

The agent does not know the MDP in advance but it will try to learn the policy function by experience. This process will also affect the value (or action-value) function. But we can also do it the other way around. We can first learn a value function and then modify π accordingly. For example, once an action-value function q has been learnt, it can be used to define a greedy policy function $\pi_q(s) = \operatorname{argmax}_{a \in \mathcal{A}} q(s, a)$. Note that we have omitted the dependence with the action in π_q because this type of policy is deterministic. Another typical policy function is called ε-greedy. This policy chooses a random action with probability $\varepsilon \in (0, 1)$ and the greedy action with probability $1 - \varepsilon$. Thus, it can be defined as

$$\pi_q(a \mid s) = \begin{cases} 1 - \varepsilon & a = \operatorname{argmax}_{a \in \mathcal{A}} q(s, a) \\ \frac{\varepsilon}{|\mathcal{A}| - 1} & \text{otherwise.} \end{cases} \tag{2.3}$$

There are several methods to learn the policy function but we are going to show one just as an example, the Q-learning algorithm [7]. This method uses a tabular function $Q\colon \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ that will be updated to improve a given policy π to compute either v_π or q_π in the end. The algorithm initializes the Q table randomly and iterates during a given number of episodes. Every episode consists of a succession of steps and ends when a terminal state is reached. Every step starts from the last state S and selects a valid action A following a behavior policy μ derived from Q. A typical example of μ is the ε-greedy policy with respect to Q. Then, this action is taken and new state S' and reward R are observed. Thus, we update the Q table with the new information using the Bellman equation [8]:

$$Q'(S,A) = Q(S,A) + \alpha(R + \gamma \max_{a \in \mathcal{A}} Q(S',a) - Q(S,A)),$$

where $\alpha > 0$ is the learning rate. Finally, we can compute a greedy policy function

$$\pi(s) = \operatorname{argmax}_{a \in \mathcal{A}} Q(s,a).$$

This algorithm has been proved to converge to the optimal action-value function [9], that is, Q converges to the mapping $q_* = \max_\pi q_\pi$ as the number of episodes go to infinity.

This is only one example of an algorithm but there are numerous others. In any case, we are not going to cover them as it is not the purpose of this project. But we are going to talk about a problem that all these methods encounter, which is the exploration-exploitation balance; there must be an equilibrium between exploration, which helps the agent find as many different states as possible, and exploitation, which allows the agent to collect the highest possible reward. We will cover this problem with the following example, which is one of the simplest MDPs.

2.1.1 Multi-armed Bandit

In the multi-armed [10] or k-armed bandit problem an agent can take one of the k different actions every time step and has to maximize the total reward over T time steps. Each action i yields a reward from an unknown stationary (i.e., it does not change over time) probability distribution X_i that we should be able to estimate through experience. We denote the value of a given action a with $q_*(a)$, defined as the expected reward given we selected action a at time t:

$$q_*(a) = \mathbb{E}[R_{t+1} \mid A_t = a].$$

If we knew this function in advance, we could act greedily with respect to its highest value and maximize our reward. Therefore, let $Q_t(a)$ be the estimated value for action a at time step t. This mapping is learnt through experience and can be defined as

$$Q_t(a) = \frac{\sum_{i=1}^{t-1} R_{i+1} \mathbb{1}_{A_i = a}}{\sum_{i=1}^{t-1} \mathbb{1}_{A_i = a}},$$

where $\mathbb{1}_p$ is 1 when p is true and 0 otherwise. If the denominator is 0 we fix it by defining $Q_t(a) = 0$. This way, $Q_t(a)$ estimates the mean reward yielded by a, so if we could select that action an infinite number of times, the denominator would go to infinity and $Q_t(a) \to q_*(a)$. But this is not the case because we are limited by a horizon $T < \infty$, so we need to choose actions carefully in order to have a good estimate of q_*.

If we act greedily with respect to Q_t, we will be maximizing our total reward in the short term. Thus, we are exploiting our knowledge of the environment. However, if we choose an action with a lower value, then we will be exploring the action space in order to find actions that might lead to better long-term total rewards.

Here, the exploration-exploitation problem arises; it is impossible to be both exploring and exploiting so we must find a balance between them. Therefore, we should explore enough to find the best actions so we can exploit them many times in the long run.

ε-greedy Algorithm

This is the most simple algorithm that ensures a certain balance between exploration and exploitation. This method selects actions using policy defined in (2.3), assuming that the state is always the same. This way, it is exploiting the best known action at the moment but sometimes explores some other which may be better than the current best one.

Therefore, with an infinite horizon all actions will be taken infinitely many times and, thus, their estimated values will tend to their real ones.

The main problem of this method is that all actions have the same probability of being taken even though they are already known to be suboptimal.

Upper Confidence Bound Algorithm

As exploring is really important, it is better to select useful actions rather than doing it randomly. This is what the Upper Confidence Bound (UCB) algorithm [11] was developed for. It selects the action based on its potential; higher valued actions with higher uncertainty will be preferred over lower valued ones.

The certainty of an action is defined as the number of times it has been tried. For instance, if an action has been taken few times and has yielded a low value it will be harder for it to be taken again. But every time it is selected its value will be more certain so it will have even less chances to be used in the future.

All this can be reduced to this formula:

$$A_t = \text{argmax}_{a \in \mathcal{A}} \left(Q_t(a) + C\sqrt{\frac{\log t}{N_t(a)}} \right), \tag{2.4}$$

where log denotes the natural logarithm function ($y = \log t$ if and only if $e^y = t$), $N_t(a)$ is the number of times action a has been taken until time t ($N_t(a) = \sum_{i=1}^{t-1} \mathbb{1}_{A_i=a}$) and $C > 0$ is an exploration constant.

A simple experiment comparing ε-greedy and UCB algorithms can show how the latter outperforms the former. We took 10 different normal distributions, displayed in Figure 2.1, as reward generators for each of the ten actions. Their means and standard deviations were selected randomly from the intervals $[-5, 5]$ and $[1, 2]$, respectively.

Then, we used both UCB (with $C = 2$) and ε-greedy (with $\varepsilon = 0.1$) formulas for action selection during 1,000 steps and compared the mean reward each of them obtained and the value they expected to get with each action. UCB got a final reward of 4,357.4 while ε-greedy obtained 3,974.5. We can also see in Figure 2.2a that UCB needed about 200 steps to learn which the best action was and easily beat ε-greedy (which was lucky and

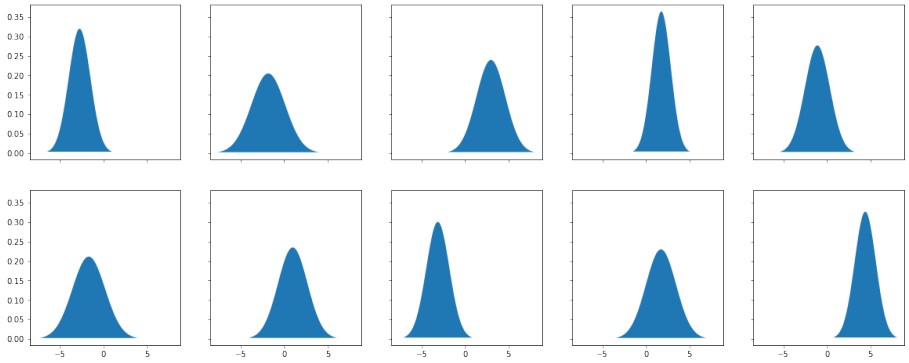
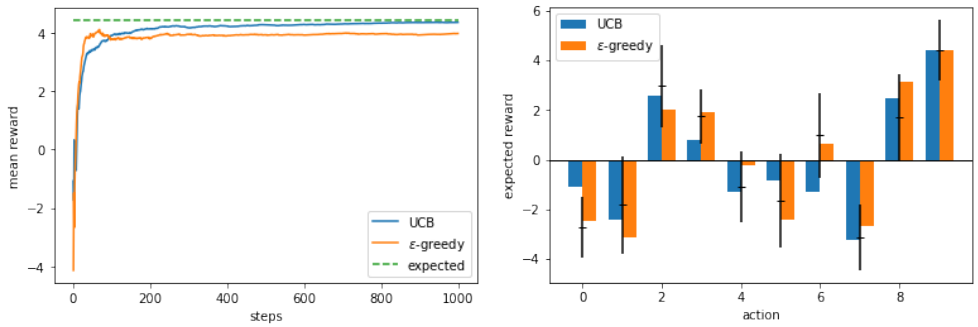


Figure 2.1: Reward distribution for each action.



(a) (b)

Figure 2.2: a) Mean reward obtained after t steps. b) Expected value of every action for each algorithm, with the real mean reward and its deviation in black.

exploited a good action at the beginning) and reached the expected mean reward (the highest mean reward).

As for the estimated value of each action, we can observe in Figure 2.2b that both algorithms estimated the mean reward of the last action really well because it was the one taken most times in both cases. They struggled more with the rest of them and that is because there were only 1,000 steps. On the one hand, ε-greedy could not explore random actions as much as needed (each action may have been selected randomly only ten times). On the other, UCB could easily find that all other actions were not worth taking. In fact, it selected the last action 976 times, while the ones that yielded a negative reward were not taken a second time. Usually, with a high number of steps UCB will have an accurate estimation of the best actions and will be more uncertain about the worst ones, but it does not need to improve its certainty.

2.2 Game Theory

In the following sections we are going to cover different algorithms that can be used to play games, but first we need to define what a game is [12].

We are going to define a game as a tuple $G = (\mathcal{N}, \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R})$, where $\mathcal{N}$ is the set of players, $\mathcal{S}$ and $\mathcal{A}$ are the sets of possible states and actions in the game, respectively, $\mathcal{T} : \mathcal{N} \times \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ is the transition function and $\mathcal{R} : \mathcal{N} \times \mathcal{S} \to \mathbb{R}$ is the reward function. We can see the similarities between this definition and the MDP defined in Section 2.1.

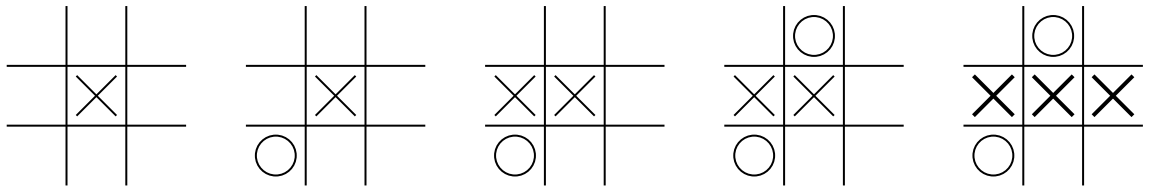


Figure 2.3: Example of a Tic-Tac-Toe game. Here, white won as it was able to place three tokens in the second row.

First, $\mathcal{N}$ represents a set of actors. It is necessary because this time there is more than one actor. The sets $\mathcal{S}$ and $\mathcal{A}$ mean exactly the same as before. Now, for each player $p \in \mathcal{N}$, $\mathcal{T}_p(s, a) = \mathcal{T}(p, s, a)$ is a simplification of $\mathcal{P}$ that maps a state-action pair into the resulting state after taking that action, and $\mathcal{R}_p(s) = \mathcal{R}(p, s)$ is another simplification of its counterpart. Finally, as there is no γ defined here, we could say $\gamma = 1$. This is not problematic since we are considering finite games, that is, games that end after a finite number of actions.

Here, we will specially focus in two player board games. Thus, we will simply say $\mathcal{N} = \{\text{white}, \text{black}\}$, similarly to chess, where white is the player that moves first. A state is defined by the board situation and some other relevant information if necessary. In addition, the reward function will only make sense for a subset $L \subset \mathcal{S}$ of terminal or leaf states where the game has already ended and we know the real outcome. Thus,

$$\mathcal{R}(p, s) = \begin{cases} -1 & s \in L \text{ and p lost} \\ 0 & s \notin L \text{ or the game ended in a draw} \\ 1 & s \in L \text{ and p won.} \end{cases} \tag{2.5}$$

Aside from that, we will also consider that the games have the following properties:

- They are zero-sum games. If we add up all the rewards that can be obtained by both players the sum will end up being zero. In other words, what is good for one player is bad for the other. The reward function defined in (2.5) matches this condition, because $\mathcal{R}(\text{white}, s) + \mathcal{R}(\text{black}, s) = 0$ for all $s \in \mathcal{S}$.
- They are perfect-information games. Every state must hold information about the entire situation of the game. It must be equal for black and white. For instance a typical card game does not have perfect information because one player only sees its hand but knows nothing about the cards of the rest of the players. Thus, a state from the same time step would be different for every player.

We will use Tic-Tac-Toe and Connect N as examples of simple games that meet these conditions.

On the one hand, Tic-Tac-Toe consists of a 3×3 board where each player alternatively places its tokens in. Typically white's tokens are represented as `X` and black's as `O`. The goal of the game is to place three tokens in the same line, either in the same row, column or diagonal. The first player that achieves this is the winner. An example of a game is shown in Figure 2.3.

On the other hand, Connect N has the same goal as Tic-Tac-Toe with the difference that in this game pieces fall by the gravity to the first empty cell in the column, starting from

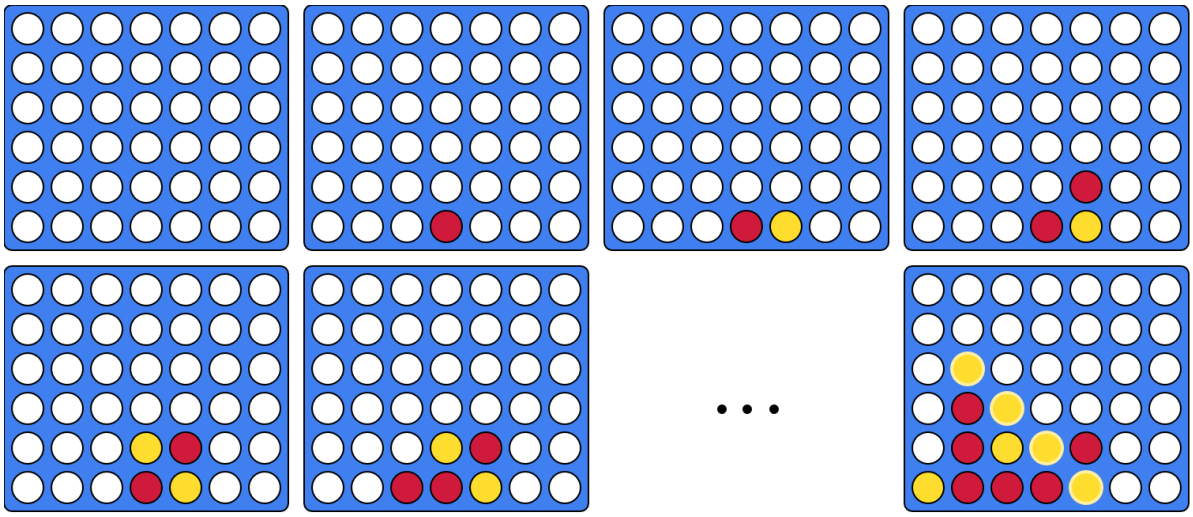


Figure 2.4: Example of a Connect 4 game. In spite of their names, white is using red pieces and black's are yellow. Some moves have been omitted to make it shorter. Black was able to win by placing four consecutive tokens in the same diagonal.

the bottom. The most usual version is Connect 4, played in a 6 rows by 7 columns board. Figure 2.4 represents a sample game of Connect 4.

Both games are solved, which means that there exists a known perfect strategy for either player. The perfect outcome for a Tic-Tac-Toe game is known to be a draw and can be computed with a brute force search, because the number of different states is only 5,478. In contrast, Connect 4's perfect strategy leads to a win by white [13]. This time, the number of different states is around 4.5×10^{12} [14], so a much more fine-tuned algorithm is needed. For example, [15] is a C++ implementation of a perfect Connect 4 player.

2.3 Search Methods

In this section we are going to cover search-based algorithms used by computers to play games. Specifically, we are going to start briefly with the Minimax algorithm and its optimization: the AlphaBeta prune. Then, we are going to study in depth the Monte Carlo Tree Search algorithm, which is the one we are really going to use.

2.3.1 Minimax and AlphaBeta

Minimax is a basic search-based algorithm used in two-player zero-sum games with perfect information. It is attributed to von Neumann, who proved the minimax theorem in 1928 [16].

Let $G = (\mathcal{N}, \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R})$ be a game, $L \subseteq \mathcal{S}$ the set of leaf states and $h \colon \mathcal{N} \times L \to \mathbb{R}$ an heuristic function. Note that L might not be the same as defined in Section 2.2. If it is the set of states where the game has already ended the heuristic function can be simply defined as $h = \mathcal{R}\big|_{\mathcal{N} \times L}$.

In the general case, it is not viable to execute the Minimax algorithm until a final state. So frequently, L is defined as the set of states at a certain depth and the heuristic function

must be an estimation of the possible outcome. The value returned by h will be a positive number below 1 when white has a higher probability to be the winner, negative but higher than -1 when black is the one with higher odds and 0 if the expected outcome is a draw.

Let us describe now the algorithm. Let $s \in \mathcal{S}$ be the current state of the game, which is called the root of the game tree, and $p \in \mathcal{N}$ the player to play in s. Minimax is a recursive algorithm that takes the root state s and returns the best move and its expected value. If $s \in L$ the algorithm cannot continue, so there is no such a move and the expected value is the result of $h(p, s)$. In the other case, this algorithm is recursively applied to all children nodes of s (i.e., the ones reachable from s after one move). Once this has been done, the best move is selected based on its value and p. If p is white, the *max* player, the move with the highest value will be chosen. Otherwise, if it is black's turn (the *min* player) the move with the minimum value will be selected. This is where the name of this algorithm comes from. White tries to maximize its value, while also trying to minimize black's, as it has to assume that the opponent is trying to make the best move as well.

This algorithm works, but usually the state space is huge and has a high branching factor. This reduces enormously the maximum depth that can be reached and, therefore, the information that can be obtained using this search. With that in mind, we must optimize the algorithm so that it does not traverse the complete state space, but rather avoids visiting branches of the tree known to lead a worse result. This is the main purpose of the AlphaBeta prune [17].

Minimax algorithm with AlphaBeta prune is similar to the original one, but it adds two additional parameters: α and β. Each of them holds, respectively, the highest and the lowest value found in the subtree that is being explored at the moment. This parameters are initialized at $\alpha = -\infty$ and $\beta = \infty$, meaning that the tree has not been visited yet, and they are passed to child nodes. Then, α is updated with the results of current node's children if it is white's turn and β if it is black's. Thus, if $\alpha \geq \beta$ at any moment it is impossible to find a better value in this subtree (from the point of view of either player) than some other already found. Therefore, it is not necessary to keep searching in this branch.

In conjunction with AlphaBeta prune, other domain-specific optimizations are often added, such as choosing carefully the order in which nodes are visited so that better moves are explored first and the tree can be pruned earlier. Also, big part of the effort is made on optimizing heuristic functions in order to anticipate as much information as possible.

2.3.2 Monte Carlo Tree Search (MCTS)

Monte Carlo Tree Search [18, 19] (MCTS) is a probabilistic algorithm that can be used in finite two-player zero-sum games with perfect information. This algorithm self-plays multiple games starting from the root of the game tree until a final state is reached. The outcome of every game is collected and backpropagated through the tree so that the expected value of every action can be estimated and the best move can be selected accordingly. As opposed to Minimax, there is no need to build a complex heuristic function, but rather it is estimated with experience.

Specifically, this method consists of four phases: selection, expansion, simulation and backpropagation. The algorithm runs N iterations and each of them executes all four

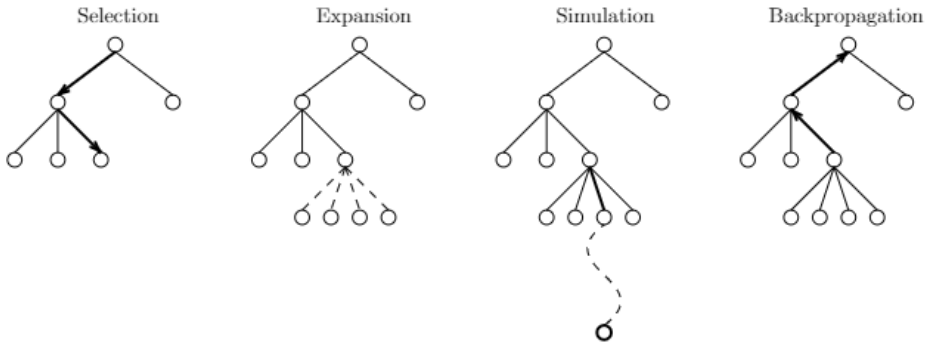


Figure 2.5: MCTS outline.

phases (except for expansion and simulation if selection phase reaches a final state). The details of every phase are listed below. They are not exactly the same as in the original MCTS algorithm, but we want to show the modification that AlphaZero uses.

1. **Selection**. Starting from the root of the tree, successive actions are taken driven by a given selection policy until a non-expanded node is reached. For instance, during the first iteration there is only one node in the tree, the root. It has not been expanded yet so the selection phase would end there.

2. **Expansion**. After the selection phase we expand the selected node. Expanding means that we add to the tree all its child nodes, that is, they are stored in memory permanently while the algorithm is running. This way any of these nodes can be selected during the next iteration.

3. **Simulation**. This is the actual self-play phase. A game is played choosing the moves at random at each node until a final state is reached. As opposed to the previous step, these nodes are stored in memory temporally until this phase ends.

4. **Backpropagation**. Once the outcome of the simulated game has been collected, it is backpropagated from the expanded node to the root of the tree. In the basic algorithm the number of visits of each node is increased, as well as its accumulated value. This information will be used during selection phase and when choosing the final move, so it is common to add some other information if required.

Figure 2.5 summarizes one iteration of the algorithm.

Once this iterative process is done, the algorithm returns the best action to take in the current state according to a given metric.

Selection policies

Selection policies are our main mechanism to ensure balance between exploration and exploitation. Selecting the correct policy for our game is crucial and can make a difference about how well the algorithm behaves. Let us introduce some examples of selection policies.

- Objective Monte Carlo (OMC). This algorithm was proposed in [20] and selects the move that has the highest probability of having a better value than the current

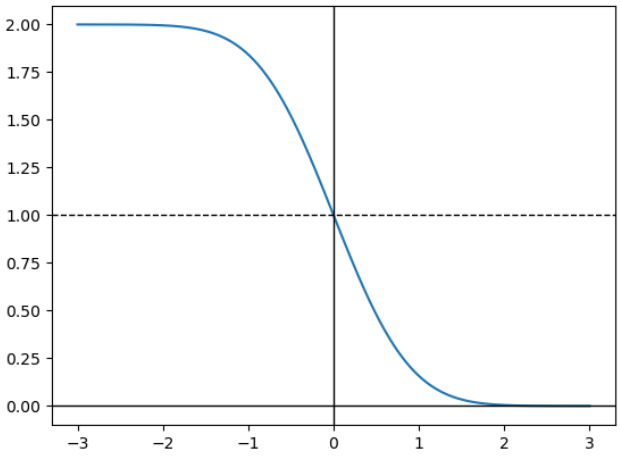


Figure 2.6: Graphical representation of erfc(x).

better node. It first defines an urgency function[1] of each node i

$$U(i) = \operatorname{erfc}\left(\frac{V - v(i)}{\sqrt{2}\sigma(i)}\right),$$

where $v(i)$ is the value of i, $V = \max_k v(k)$, $\sigma(i)$ is the standard deviation of $v(i)$ and erfc (Figure 2.6) is the complementary error function

$$\operatorname{erfc}(z) = \frac{2}{\sqrt{\pi}} \int_z^\infty e^{-t^2}\, \mathrm{d}t.$$

Then, the selected node will be the one that maximizes the fairness function

$$f(i) = \frac{NU(i)}{n(i) \sum_{j \neq i} U(j)}.$$

Here, N and $n(i)$ are the visit count of the parent node and node i, respectively. We will use $n(i)+1$ instead of just $n(i)$, as we may encounter nodes with zero visits.

- Probability to be Better than Best Move (PBBM). This algorithm is a modification of OMC used in [19]. It also maximizes the fairness function but uses a different urgency function, defined as follows:

$$U(i) = \exp\left(-2.4 \cdot \frac{V - v(i)}{\sqrt{2(\Sigma^2 + \sigma^2(i))}}\right).$$

Everything is the same as in OMC and Σ is the standard deviation of V.

- Upper Confidence bounds applied to Trees (UCT). This algorithms is one of the most widely used. It is an adaptation of the UCB algorithm (2.4). It was proposed in [21] and selects the node i that maximizes

$$v(i) + C\sqrt{\frac{\log N}{n(i)}},$$

[1] When the visit count of a node is lower than 2 we can set $\sigma(i) \approx \infty$, hence $U(i) \approx \operatorname{erfc}(0) = 1$.

where $C > 0$ is a constant (theoretically $\sqrt{2}$ works well, but it may be tuned experimentally).

In our case, we have to keep in mind that we are working with two-player games, so we must be careful when implementing these algorithms. We must take into account the turn at the node where we are selecting the move. If we are always trying to maximize the value of a node, the tree will consider that the rival's worst moves are actually the best, because those moves will lead to better states for us. However, that is not the expected behavior, as we want to model a rival able to play at least as well as we do. Thus, we can solve this problem easily if we set the value at each node from the point of view of the player to play at that node. Therefore, if a victory for black is being backpropagated, then those states where it is black's turn will be updated with a new value of 1, while in the others the new value will be -1. This way, we ensure that we always maximize the odds of winning for the player who is going to make a move during selection phase.

Final move policy

Another important thing to determine is how we select the move that is going to be returned by the algorithm. The straightforward approach is to choose the action that leads to the state with the highest value associated. This is the max child strategy, as named in [22], and it is not usually the most suitable when the maximum number of iterations is low. There are some better methods.

One of the most used methods is robust child [18, 22]. This policy just chooses the action that has been taken most times. This strategy works well because the algorithm tries to explore the most promising nodes: if a node has a high number of visits, then its expected value is more certain but it will be high also, because the branch would have been discarded otherwise. Of course, this policy's performance is closely related to the selection policy, which must ensure a correct balance between exploration and exploitation.

Previous policy does not ensure that the returned child is actually the one with the highest value. Therefore, we can join these previous methods and choose the max-robust child [19], that is, the node with the highest number of visits as well as highest value. If such a node does not exist it is recommended to keep looking until one is found, rather than returning a child with a lower value or visit count. With this method we increase the probability that the selected move is optimal.

Lastly, we might try to maximize both value and visit count together. This is the secure child policy. In this case, the action which maximizes a lower confidence bound is selected. For example, we could use the following formula [23]:

$$v(i) + \frac{A}{\sqrt{n(i)}},$$

where $A > 0$ is a constant and $v(i)$ and $n(i)$ are the value and number of visits of node i, respectively.

Anyway, all these techniques tend to behave similarly, assuming that enough number of iterations have run so that the expected value at each node is sufficiently precise. Nevertheless, the most commonly used policy is robust child as it is simpler than the others and slightly outperforms max child [22].

2.4 Deep Reinforcement Learning in Combinational Games

In this chapter, we provide a general overview of deep reinforcement learning, focusing on its applications to master combinational games. We assume from the reader some previous fundamental knowledge about artificial neural networks such as perceptron, multilayer perceptron, activation and loss functions, and backpropagation algorithm.

Deep reinforcement learning [24] (DRL), as its name may suggest, combines the advantages of deep learning (DL) with reinforcement learning (RL). It relies on deep neural networks to approximate both value and policy function (Section 2.1). Indeed, our policy and value function will be determined by the parameters and variables of our deep neural network.

2.4.1 Activation Functions

An activation function transforms the weighted sum of the input into the output of the neuron. The main purpose of activation functions is to introduce non-linearity into the output of a node. A neural network without an activation function behaves essentially as a linear regression model. With this component, our model is capable of learning more complex tasks. The most common activation functions are:

ReLU

The rectified linear unit (ReLU) replaces all negative values with 0. It is mathematically defined as

$$R(x) = \max\{0, x\} \tag{2.6}$$

and its graph is the one seen in Figure 2.7.

ReLU is the most frequent activation function, especially in Convolutional Neural Networks, as we will see later. ReLU function is able to speed up the training phase of deep neural networks compared to traditional activation functions, due to its simple derivative, which is 1 for a positive input and 0 otherwise. Because it is constant, it is not required to take additional time for computing error terms.

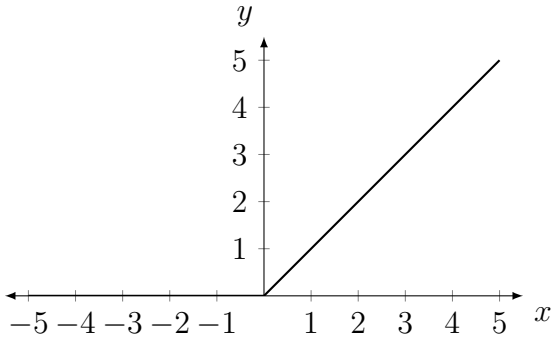


Figure 2.7: ReLU function.

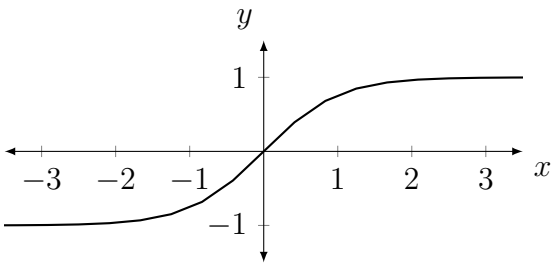


Figure 2.8: Tanh function.

Tanh

The hyperbolic tangent (Tanh) takes any real number as input and outputs a number between $(-1, 1)$. We can see its graph in Figure 2.8.

Large positive inputs will be mapped to values close to 1, whereas large negative inputs will correspond to values close to -1. The Tanh activation function is mostly used for classification between two classes.

Softmax

The Softmax activation function takes a k-dimensional vector of real values as input and outputs a vector of values between $[0, 1]$ whose components' sum is 1. The Softmax function is mathematically defined as follows:

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}. \tag{2.7}$$

The output vector may be interpreted as a categorical distribution over predicted output classes.

2.4.2 Convolutional Neural Networks

Convolutional Neural Networks [25], also known as CNN, are a type of neural networks widely used for image recognition. They are able to recognize faces, objects or signals from pictures.

As shown in Figure 2.9, we receive an image as input, and as output we get the probability that this image belongs to each of the existing categories. Obviously, the network will be working properly when it assigns the greatest probability to the corresponding category.

We should recall that an image is nothing more than a pixel matrix, and pixels are numerical values. A channel is a component of the image. For instance, a standard digital camera has three channels: red, green and blue. We can imagine a conventional image as three matrices one on top of each other (one color each), in which each pixel has a value between 0 and 255. Moreover, a black and white image will have one single channel.

There are four main operations that form the basic building blocks of all CNN: convolution, non linearity, pooling and classification. We will now present a further explanation on each of these steps.

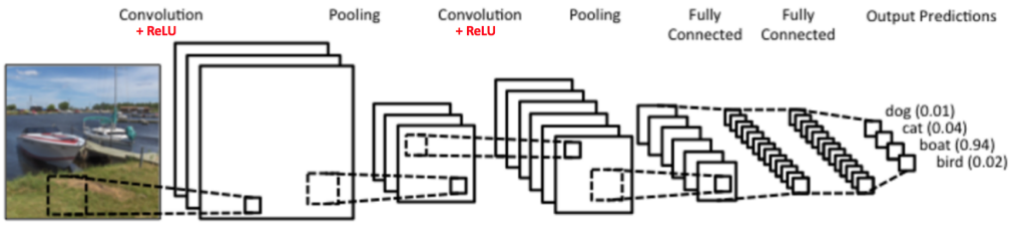


Figure 2.9: Elemental structure of a Convolutional Neural Network. Source [26].

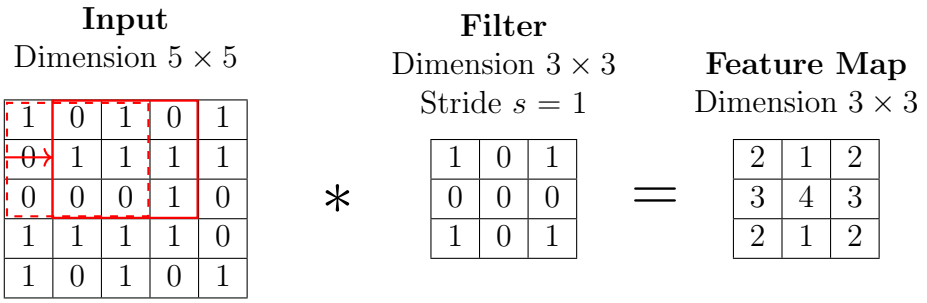


Figure 2.10: The convolution operation.

Convolution

The main goal of this step is to extract characteristics from the image received as input. Convolution preserves the spatial relationship between pixels by learning the patterns of the images.

Considering the input image as a pixel matrix $m \times m$, we denote *filter* to a $n \times n$ matrix and *feature map* to the matrix formed by moving the filter s positions (*stride*) over the image and performing the dot product (element wise multiplication) for every position. This process is illustrated in Figure 2.10

It is important to note that the filters act as detectors for features of the original input image. Different values of the filter matrix will clearly produce different feature maps for the same input image. In practice, a CNN learns the values of these filters during the learning stage. The more filters we have, the more features we can obtain from images.

Non Linearity

We need an additional operation in order to add non linearity to our system. For this purpose, we use the ReLU operation. This operation is applied to each pixel of the resulting feature map.

The feature map resulting of applying the ReLU operation is called Rectified feature map. In Figure 2.11 we see an example of applying ReLU to a feature map. Black pixels have negative values and white pixels have positive values. When ReLU is applied, rectified feature map only contains positive values.

Pooling

In the pooling step, we reduce the dimensionality of our feature map retaining the most relevant information. We divide the feature map in sub-matrices and obtain an element

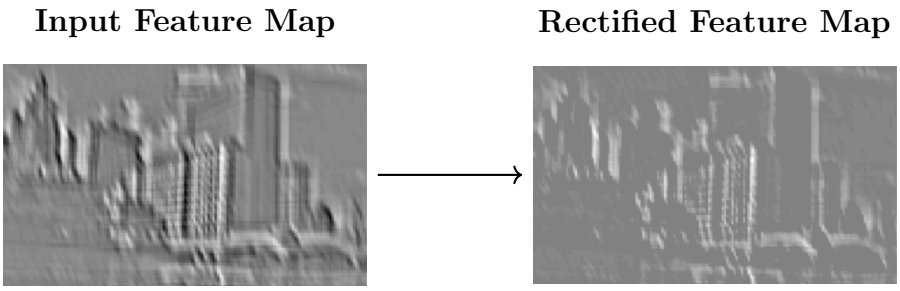


Figure 2.11: Example of a ReLU operation. Source [27].

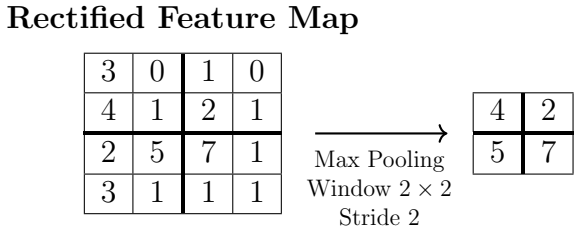


Figure 2.12: Example of Maximum Grouping.

from them. We can apply many types of pooling: Maximum, Average, Sum...

In the case of grouping by maximum, we divide the feature map in sub-matrices and take the greatest element. But we could have taken the average the sum element instead. Maximum grouping is the most used, as it normally performs better than the others.

Classification

This step is usually performed by a fully connected neural network layer, which uses Softmax as activation function on the output of the last layer.

In conclusion, the output of the previous three steps has provided us some high-level *patterns* that occur in the input image. The goal of the traditional neural network is to use these patterns to classify the input image into a specific category.

2.4.3 Residual Neural Network

There are many architectures for Convolutional Neural Networks. We have introduced the *LeNet* architecture [28], which was the very first successful architecture of CNN and the basis for the others.

Furthermore, Residual Networks [29] (*ResNets*) are state of the art in Convolutional Neural Network models. When we add many layers to neural networks some problems may arise. Very deep neural networks are difficult to train because of vanishing and exploding gradient. In order to solve this problem, Resnet uses *skip connections* ("shortcuts" to jump over some layers), and Batch Normalization to obtain a better performance.

2.4.4 Batch Normalization

Batch normalization [30] is a very common technique used to improve the training process in deep neural networks. This method allows us to stabilize and dramatically accelerate

the training stage.

A mini-batch is a subset of the training data, supplied for a given epoch. Batch normalization normalizes the inputs to a layer for each mini-batch. This normalization is done by subtracting the mini-batch's mean and dividing by the its standard deviation.

The effects of batch normalization are evident, however, the reasons of why it works so well are still under discussion.

2.4.5 Regularization

Overfitting occurs when a model tries to fit the training data so well that it fails to generalize to new observations. It is an important issue for deep learning and a solution is required. Regularization is a technique that prevents overfitting, allowing the model to generalize effectively to the underlying structure.

We have multiple types of regularization, but the one we will introduce here is the L_2 regularization. It essentially encourage weights to have values close to zero. For doing so, it adds a penalty term (the squared value of the weights) in the cost function:

$$Cost = Loss + \lambda \sum_{j=0}^{N} \theta_i^2, \tag{2.8}$$

where λ is the regularization coefficient and θ_j are the weights of the respective layer. The λ coefficient controls the amount of regularization applied to the model.

Chapter 3

AlphaZero

AlphaZero [31] is a general-purpose reinforcement learning algorithm developed by DeepMind in 2017. It can learn from *tabula rasa*[1] and achieves superhuman performance in combinational games such as Go and Chess. The most relevant characteristic is that it uses self-play, so that it starts playing randomly against itself and gradually learns further comprehension of the game.

Furthermore, AlphaZero belongs to model-based algorithms [24]. These kind of algorithms have a difference of paramount importance: future states and rewards can be accessed by the agent via the environment model. This characteristic will benefit us as the agent will be able to make a better planning. To be concrete, the rules of the combinational game that we are trying to learn are specified, so that the transition and reward functions are accessible for our agent to evaluate and improve its policy.

The algorithm is formed by three distinguished parts: a general-purpose reinforcement learning algorithm, a deep neural network, and a general-purpose tree search algorithm (Monte Carlo Tree Search). AlphaZero surpasses previous approaches made for combinational games by using a powerful combination of MCTS and neural networks.

We will then explain each of these parts and show a complete execution of a tree search iteration to provide a better understanding.

3.1 Training Loop

The overall scheme of AlphaZero's training process is a loop between self-playing and training the neural network. The algorithm plays against itself executing a MCTS for each move. At each move, the following information is stored:

- The game state: stores the current board configuration and the turn of the player who moved ($+1$ for white and -1 for black). This will be the input to the deep neural network.
- The search probabilities (π): will be obtained proportionally to the visit count in MCTS (described later).

[1] Without any human expert play or domain-specific knowledge

- The winner (z): $+1$ if the player whose turn it is won and -1 if the player lost. This data is added once the game has finished, whereas the game state and search probabilities are stored after choosing each move.

Once the algorithm has played enough games against itself, it samples a mini-batch of positions from the information stored in the last games. It will retrain the neural network on these positions comparing predictions from the neural network with the search probabilities and actual winner. We will denote the collected data as (s, π, z) and the network as $f_\theta(s) = (p, v)$, where θ are the parameters of the neural network, p are the predicted search probabilities, and v is the predicted value.

3.2 Self-play

As previously explained, we execute the self-play process with MCTS for gathering data. The Monte Carlo Tree Search is a slight different version of the one introduced in Section 2.3.2.

First of all, each node of the tree contains the following information:

- **State**: current board configuration and turn.
- **Action**: action that needs to be taken to reach the actual node.
- **N**: the node's visit count. $N = 0$ if the node has not been visited yet.
- **W**: the node's total reward. This is obtained by the output of the neural network and when reaching terminal states.
- **Q**: the node's average reward, i.e., $Q = W \,/\, N$.
- **P**: the probability of taking the action that drives to this node. This probability will be obtained by the output of the neural network.

It is important to preserve the perspective. In our case, the information in one node is always read from the perspective of its parent node. To avoid confusion, we should never forget the perspective of the node that we are dealing with.

There are also differences on the phases we take to perform the search. In AlphaZero, the phase of simulation is replaced with the evaluation step. This way, we obtain the following method:

- Selection. As we explained before, we will start from the root node and traverse through the tree selecting actions until we find a leaf node. Additionally, the selection policy is given by the formula

$$a = \text{argmax}_{a \in \mathcal{A}}(Q(s, a) + U(s, a)), \tag{3.1}$$

where $Q(s, a) = W \,/\, N$ is the average reward that is stored in the node resulting of taking the action a from the state s, and

$$U(s, a) = c_{\text{puct}} P(s, a) \frac{\sqrt{\sum_{a'} N(s, a')}}{1 + N(s, a)},$$

where c_{puct} is a parameter determining the exploration scale. In this formula, $Q(s, a)$ encourages the exploitation of higher-reward actions, and $U(s, a)$ encourages the exploration of less-visited actions.

- Expansion. The expansion phase is not different from the one described in the previous chapter.
- Evaluation. In this step, the neural network $f_\theta(s)$ outputs a vector of move probabilities p with components $p(a|s)$ for each action a, and a scalar value $v \in [-1, 1]$ estimating the expected outcome z, i.e., indicating how "good" the current state is. These values will be used later in the selection and backpropagation stages, respectively. Also, as the neural network is initialized randomly, these values may not be accurate in the early stages of the training process, but they will gradually become more accurate.
- Backpropagation. In this stage, the information from the expanded node up to the root is updated. In particular, we perform the following updates:

$$N(s,a) := N(s,a) + 1 \qquad W(s,a) := W(s,a) + v \qquad Q(s,a) := \frac{W(s,a)}{N(s,a)}$$

  changing perspectives according to the parent node's player's turn.

One last relevant detail concerning the evaluation step is the addition of Dirichlet Noise. When predicting the probabilities for the root node, we add "noise" to them, which encourages exploration and ensures that a diverse set of positions are encountered. This noise comes from the *Dirichlet distribution* [32] of parameter α: $\mathrm{Dir}(\alpha)$. The support of the Dirichlet distribution is the set of vectors whose entries are non-negative real numbers and their sum is 1. When α is close to 0, Dirichlet distribution tends to favour basis vectors. For example, $(0.95, 0.02, 0.03)$ will be more likely to be sampled than $(0.3, 0.4, 0.3)$. Instead, when α takes values greater than 1, more-balanced vectors are preferred. To add the noise to the probability vector, we sample a vector n from $\mathrm{Dir}(\alpha)$ and perform a λ-weighted sum:

$$p := (1 - \lambda) \cdot p + \lambda \cdot n,$$

where λ is the *noise fraction* parameter. Note that because p and n both have coordinates summing to one, p preserves this property.

We then repeat all four phases for N iterations. Once all iterations have been executed, we have to determine which move to finally select. In the previous chapter we have already introduced some approaches for this. However, in AlphaZero a different final move policy is used. After iterating N times, an action sampled from a categorical distribution with probabilities given by the formula

$$\pi(a|s) = \frac{N(s,a)^{1/\tau}}{\sum_{a'} N(s,a')^{1/\tau}}, \tag{3.2}$$

where τ is a temperature parameter. The temperature parameter is used to control exploration and may take two values: either $\tau = 1$ or $\tau \to 0$. Let us briefly examine (3.2).

If $\tau = 1$, we have

$$\pi(a|s) = \frac{N(s,a)}{\sum_{a'} N(s,a')}$$

and the final move is chosen according to a sample from the categorical distribution π, where probabilities are related to visit counts. For instance, if the root node performs

100 iterations and has 3 children with 20, 50 and 30 visit counts, respectively, we get the distribution π with probabilities $(0.2, 0.5, 0.3)$. Then, we would take a random sample with this probabilities and finally choose the sampled move. This way, we do not always take the most visited move and we are encouraging exploration.

On the other hand, if $\tau \to 0$, we have

$$\pi(a|s) = \lim_{\tau \to 0} \frac{N(s,a)^{1/\tau}}{\sum_{a'} N(s,a')^{1/\tau}}$$

which ultimately assigns a probability of 1 to the most visited child node and 0 to the rest. Therefore, when $\tau \to 0$ we are actually using the robust child policy. Continuing with the example from the last paragraph, in this case we would get the distribution π with probabilities $(0, 1, 0)$. Thus, we are always selecting the node with maximum visit count and encouraging exploitation.

In AlphaZero, when the self-play proccess is being executed to collect data, $\tau = 1$ for the first moves (depending on the game) and then $\tau \to 0$ for the rest of the game. Moreover, when playing a real game with an opponent, the temperature is always set to $\tau \to 0$.

3.3 Neural Network

The neural network $f_\theta(s) = (p, v)$ has the following architecture: it receives a transformed version of the game state (board, turn and other relevant information) as input and returns two outputs. The policy head outputs a vector p, with the probabilities for each of the possible actions to take in the current state, and the value head outputs the value v of the state, which tells us how good the current state is. Note that these act for the policy and value function that we have defined for our agent in Section 2.1. We will now describe all the details about the architecture of the neural network used for AlphaZero.

As we have previously specified, the neural network parameters θ, are randomly initialized and updated to minimize both the error between the predicted vector p and the search probabilities π, and the error between the predicted value v and the game outcome z. Specifically, the parameters θ are optimized by gradient descent on a loss function l that combines mean-squared error for the value head, cross-entropy loss for the policy head, and L_2 weight regularization. The formula is

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

where c is the parameter which controls the level of weight regularization.

For the input, the stored game state will be transformed into feature planes stacked one on top of each other. The format of these feature planes is meant to generalize the input for any existing combinational game. Therefore, the data is not augmented in any way in the AlphaZero algorithm, as some games are not invariant to rotation and reflection.

As the figure Figure 3.1 portrays, the input of the neural network is an $N \times N \times M$ image stack. $N \times N$ is the size of the game board and $M = (2PT + L)$. Here, there is one set of PT planes for each player, one plane for each piece type (P type of pieces) for the previous T time steps. Every feature plane is composed of binary values, indicating the presence of the player's pieces. In addition, we add L constant-valued input planes for

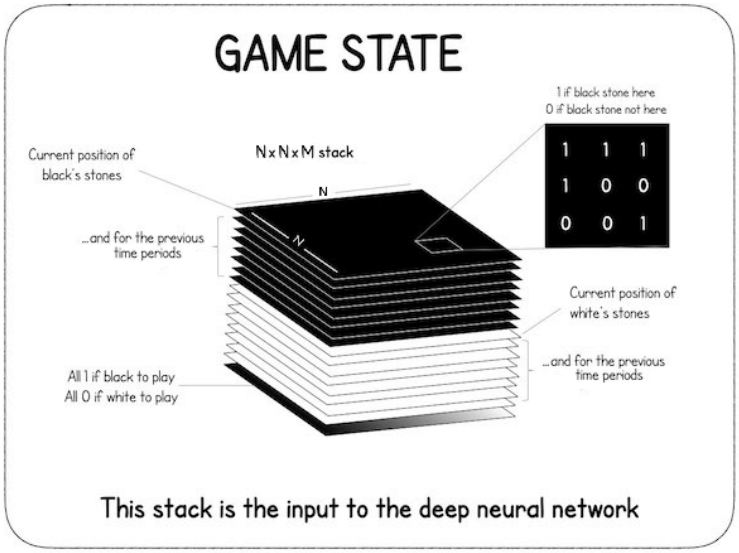


Figure 3.1: AlphaZero's game representation. Source [33]

extra information of the state as the player's turn or special rules, for example the legality of castling in chess. An explanatory example is provided in Section 3.4 to facilitate the comprehension of this conversion.

The architecture is clearly illustrated in Figure 3.2. The structure of the network consists of a convolutional layer, followed by a "body" of 19 residual layers (ResNets) and finally both policy and value "heads". The convolutional layer contains 256 convolutional filters[2] of kernel size 3×3 and stride 1, a Batch Normalization and a Rectified Linear Unit (ReLU). Each residual layer consists of two rectified batch-normalized convolutional layers with a skip connection. Furthermore, the value head is composed of one rectified batch-normalized convolution of 1 filter and 1×1 kernel size with stride 1, a rectified fully connected layer of size 256, and a last fully connected layer of size 1 with a tanh non-linearity unit. And finally, the policy head contains one rectified batch-normalized convolution of 2 filters and a final fully connected layer with a Softmax activation unit.

During training, 5000 first-generation tensor processing units (TPU) are used to generate self-play games, and 16 are used to train the neural network. With this setting, training lasts around 9 hours in chess, 12 hours in shogi and 13 days in Go, playing 44 millions, 24 million, and 140 million training games respectively [31].

[2]The number of residual layers and convolutional filters may vary depending on the game or the computational power available.

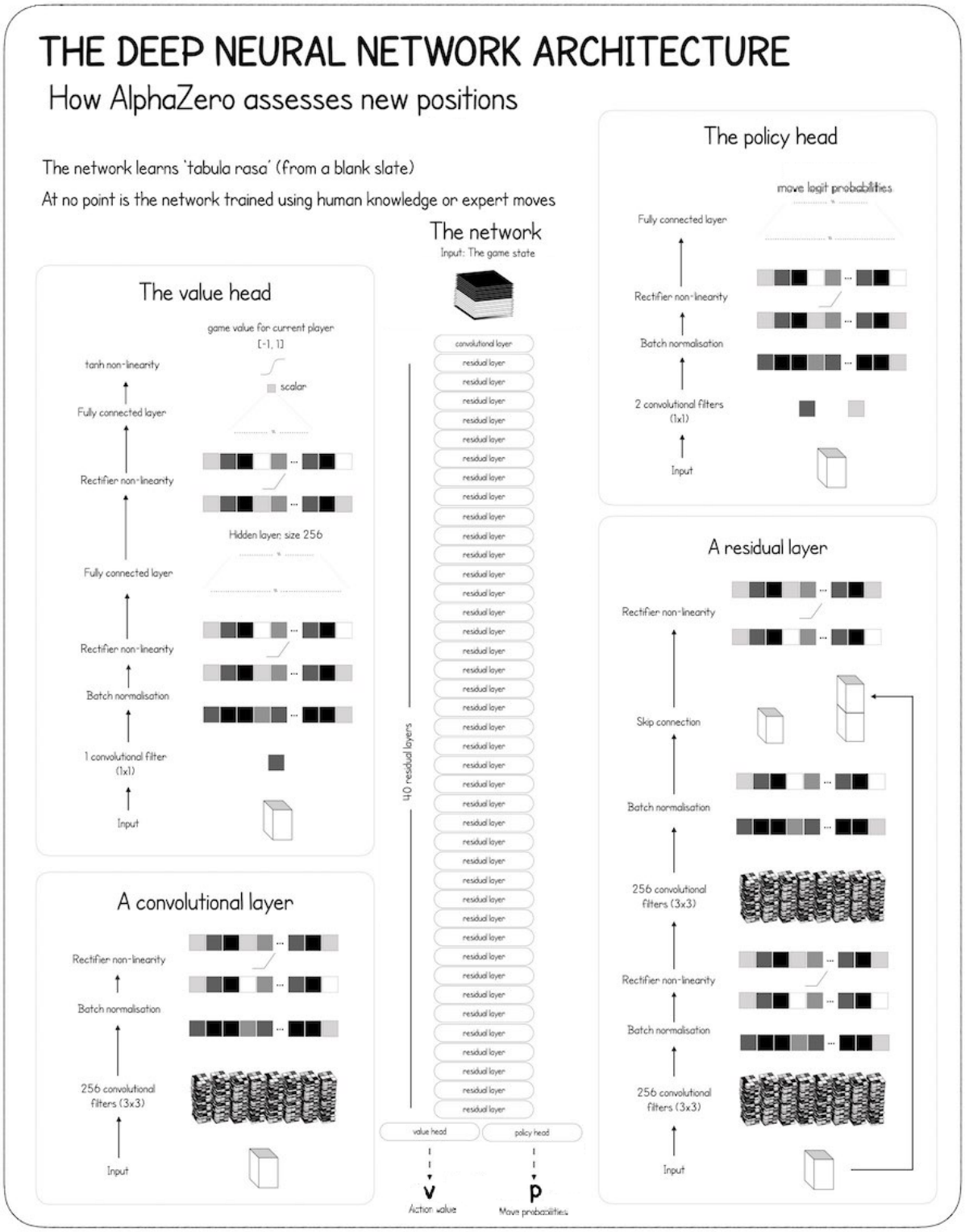


Figure 3.2: AlphaZero's neural network architecture. Source [33]

3.4 Execution Example

We are now going to show an example of the execution of MCTS in the self-play process. This example is intended to throw light on all the abstract concepts and details that we have explained so far. Once we have fully comprehended how to piece together all the concepts used, the execution is straightforward to grasp.

First, recall the objective of MCTS: choose the next move and store the corresponding information for it. Let us suppose we are playing the simplest combinational game we know: Tic-Tac-Toe. Crosses denote white player's pieces and circles denote black's. In order to represent the choice for each move, we number the possible positions on the board from 1 to 9 (left to right, top to bottom). Also, with the intention of illustrating the tree search to the end of the game, we assume the game begins from the state in Figure 3.3. It is black player's turn.

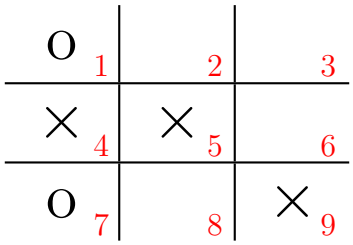


Figure 3.3: Initial board state in the example.

We start to execute MCTS from this node. We begin with the selection step. For the whole execution of MCTS, we fix the value of the exploration scale constant to $c_{puct} = 5$. As there is only one node in the tree, the root is a leaf node and we finish the step immediately. We head to the second stage: expansion. As Figure 3.4 shows, we add to the tree root all its child nodes.

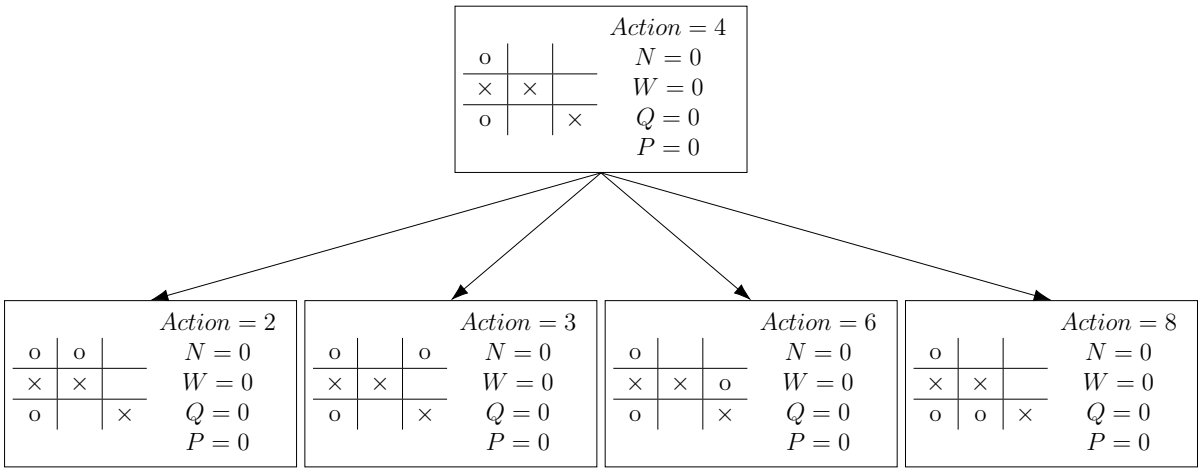


Figure 3.4: Tree after expansion step.

The following step is evaluation. To accomplish this step, we introduce the board state and the turn (black) into the neural network. As output, we obtain the state value and the probability vector. Then, we assign each probability to its corresponding node as depicted in Figure 3.5.

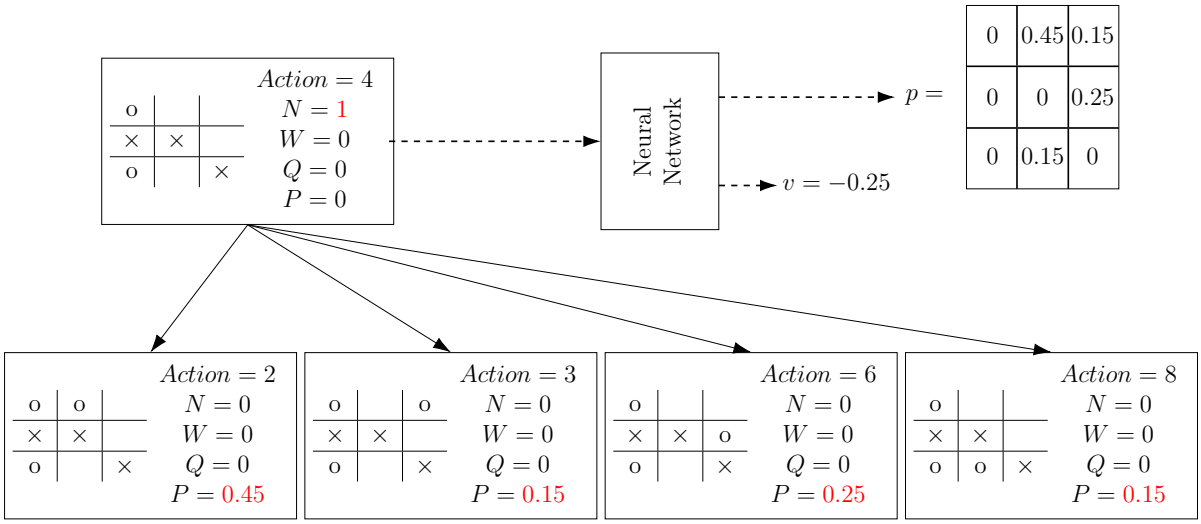


Figure 3.5: Tree after evaluation and backpropagation steps.

Taking a look at the output of the neural network, we note that our neural network is *semi-trained* at this stage of the training process. The network is giving the maximum probability to an action that may make us lose if the opponent plays correctly. Nevertheless, the second highest probability is assigned to the proper node that avoids us losing. Additionally, the backpropagation step has also been executed. It is not necessary to backup W and Q in the root node, but we update the visit count to $N = 1$. Finally, the first iteration of MCTS has finished.

We start the second iteration by the selection stage. As we defined earlier, we will begin selecting from the root node. Now, the root node is not a leaf node, so we have to decide which child node to select. According to the selection policy, we choose the action 2, as it has the highest probability and the rest of the terms involved in the formula are the same for every child node. We have reached a leaf node and the selection step is done.

Next, we expand and evaluate, that is, we create the new child nodes of the one selected and assign their corresponding probabilities resulting of the output of the neural network.

Now, we perform the backpropagation step. This time it is a little bit more complex because the player's perspective comes into play. We have remarked earlier that the information in one node is always seen from the perspective of its parent node. The output value from the network $v = -0.2$ is from the perspective of the white player. However, the information stored in the node needs to be seen from the perspective of the black player, and so it needs to be reversed. Thus, we have: $N = 1$, $W = 0.2$, $Q = 0.2$.

But we have not finished yet. We still need to return recursively from the leaf node until we reach the root node. But the parent node is already the root node, and like before, we only have to update the visit count: $N = 2$. Now, the second iteration is over and the renewed tree is the one shown in Figure 3.6

Once again, the third iteration starts with selection from the root node. Now, from the

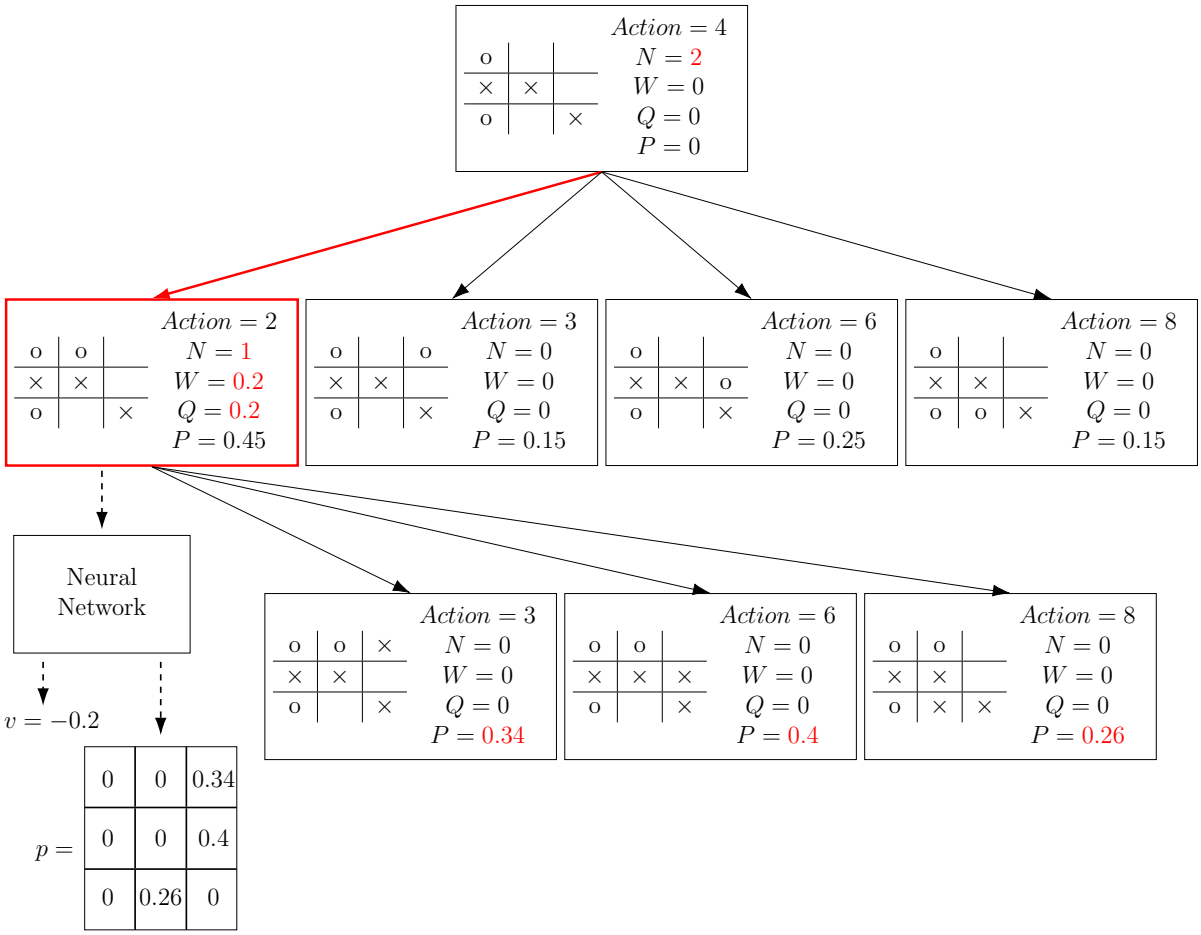


Figure 3.6: Tree after the second iteration.

formula of the selection policy we obtain:

$$Q(s,2) + U(s,2) = 0.2 + 5 \cdot 0.45 \cdot \frac{\sqrt{2}}{2} \approx 1.79$$

$$Q(s,6) + U(s,6) = 0 + 5 \cdot 0.25 \cdot \frac{\sqrt{2}}{1} \approx 1.76$$

$$Q(s,3) + U(s,3) = Q(s,8) + U(s,8) = 0 + 5 \cdot 0.15 \cdot \frac{\sqrt{2}}{1} \approx 1.06.$$

Hence, we select first action 2. It is not a leaf node, so we continue selecting from its child nodes. We choose action 6, as it has the highest probability and the rest of the terms involved in the formula are the same for every child node. Selection finishes here as we have reached a leaf node, and a special one because it is also a terminal node. The game ends in this node and there are no children to expand. Therefore, we skip the expansion step. And for the evaluation step, we do not need to use the neural network. The probabilities are not needed and the value can be obtained directly from the game reward $z = 1$.

It is time to backup recursively from the leaf node to the root node. There are three nodes in the path and the perspective of each node should be switched. Note that the reward that we obtain from a terminal state is $+1$ if white wins and -1 if black wins. We have to transform this number to the perspective of the parent node. In this situation,

the parent of the leaf node corresponds to white's turn. Consequently, the value stored in the leaf node should be +1. However, in the *Action* = 2 node, we change the perspective to the black player and we store the value -1[3]. The other values of Q and N are also updated accordingly. After the third iteration has been performed, the resulting tree is portrayed in Figure 3.7.

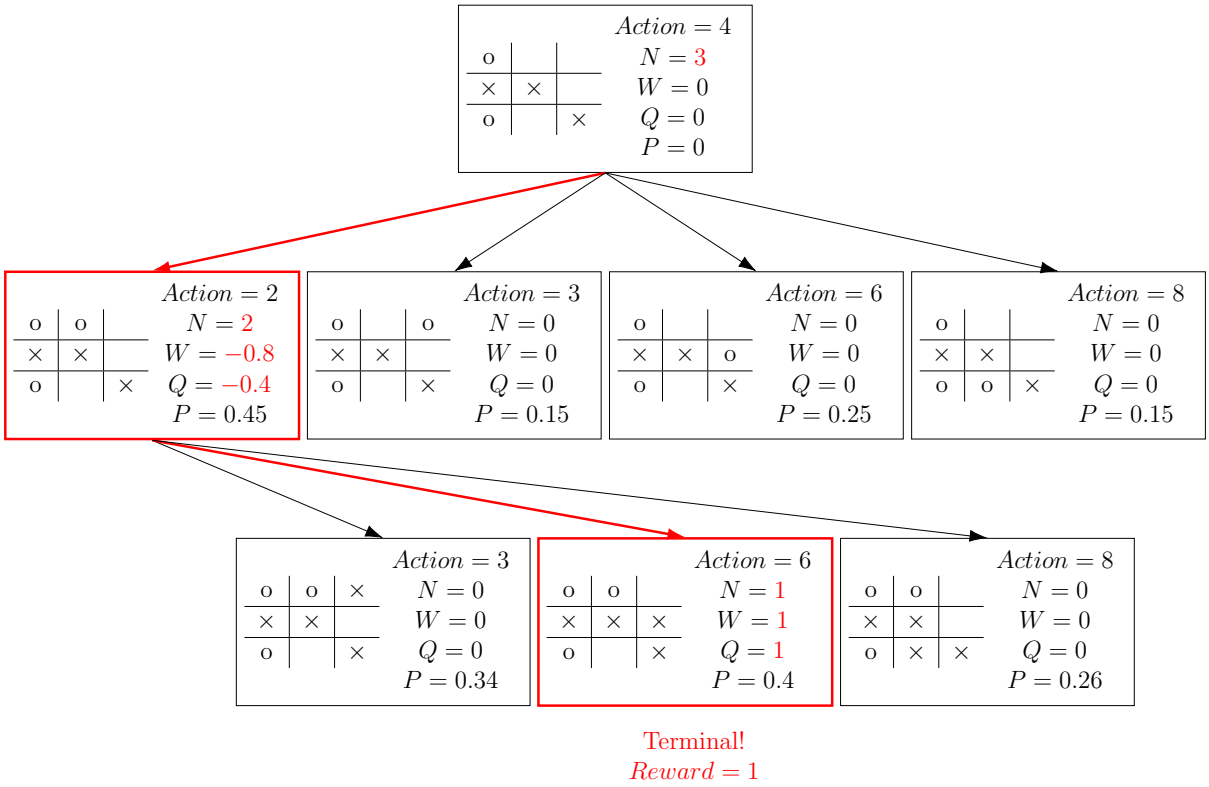


Figure 3.7: Tree after the third iteration.

We have fully executed three iterations of MCTS in order to illustrate how the tree search works. After performing 100 iterations, the tree is grown much larger and the estimated values are more accurate (shown in Figure 3.8). The tree search is over and now we have to choose our definitive move on the real board. The final move selection is accomplished according to (3.2). We are performing the self-play process, so the temperature is set to $\tau = 1$. Thus, the move will be selected according to a categorical distribution π with probabilities $(0.2, 0.05, 0.7, 0.05)$. Here, we suppose that we choose the most likely move, *Action* = 6, but we have to keep in mind that any of the other moves could have been selected.

The circle has finally been placed at position 6 on the board, so the root node in the tree will be changed to the child node and the next MCTS will go on from this new root node. Note how MCTS has allowed us to choose the proper move, even when the neural network produced an inaccurate probability vector. When the real game is over, we obtain the data and the results from each move (illustrated in Figure 3.9).

As the result of the game is a draw, the labels for the reward are 0 for every move. Now,

[3]Understanding the change of perspective might result the most confusing aspect of MCTS. If we reach a terminal state in which black wins, we obtain a reward value of -1. But the parent of the leaf node corresponds to black's turn and the value stored would be +1. In general, we will see in the following chapter that the change of perspective can be easily made multiplying by the current node's turn.

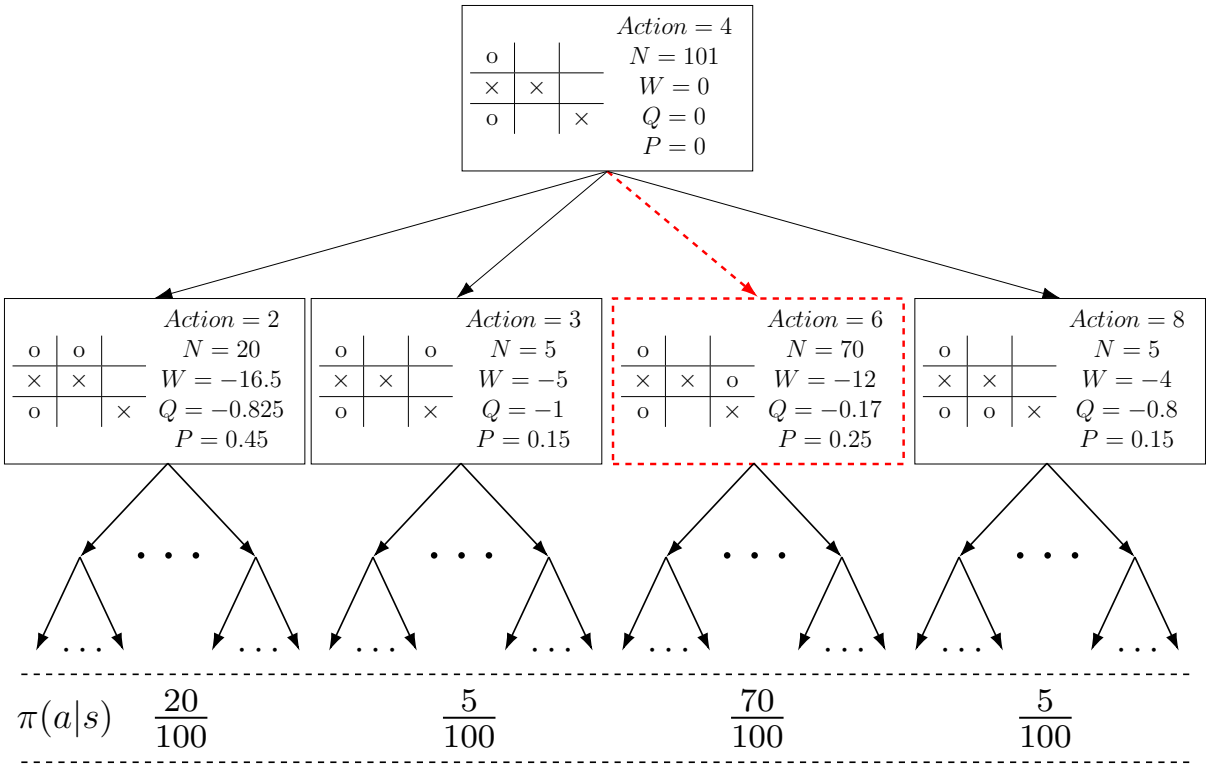


Figure 3.8: Final move selection.

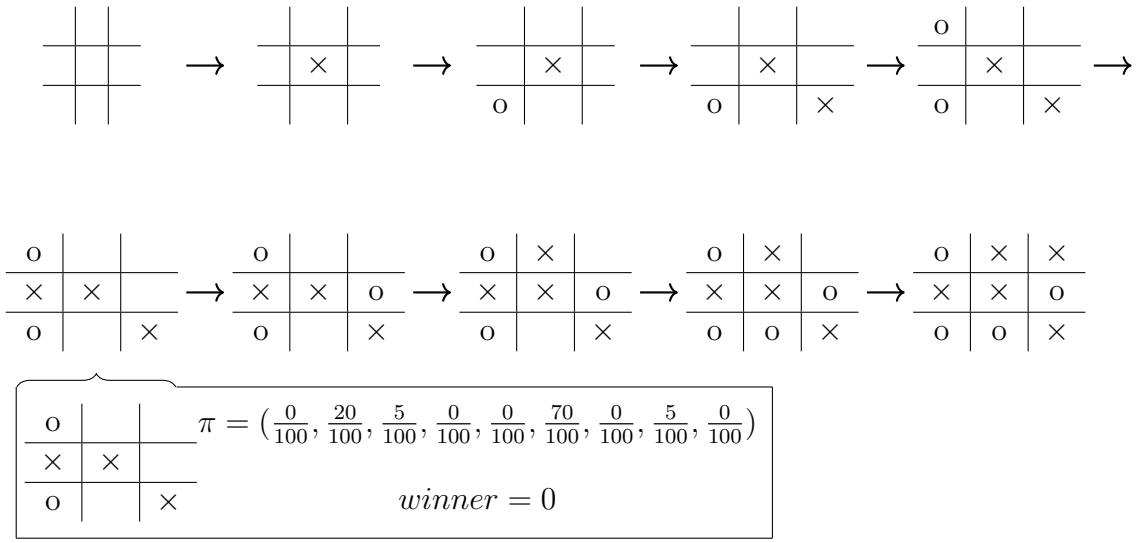


Figure 3.9: Data stored from the game.

the game has ended and the information provided by MCTS is available to train the deep neural network.

Chapter 4

AlphaZero Implementation

Our AlphaZero implementation was divided into several modules: game representation, game strategies, neural network and the actual AlphaZero, which uses the rest of them. This way, every module is independent from each other and can be reused.

Everything was implemented in *Python* using typical machine learning libraries, such as *NumPy*[1] [34], *SciPy*[2] [35] or *TensorFlow*[3] [36].

4.1 Game Representation

First of all, we needed a common interface in order to represent a game so that we could easily change from one to another without changing many lines of code. As we are in a reinforcement learning context, we are going to represent games as reinforcement learning environments. For this purpose we are using *OpenAI Gym* [37], one of the most popular reinforcement learning libraries, which includes interfaces for environments as well as several action and observation spaces.

An *OpenAI Gym* environment (`Env` class) contains at least the `action_space` and the `observation_space` attributes and must implement at least three methods: `step`, `reset` and `render`. The `reset` method resets the environment to an initial state and returns that state. The `render` method displays the current state of the environment. And the `step` method is the one where the actions are executed in. This method takes a valid action (one in `action_space`) as its argument, modifies the environment's inner state and returns the new observation (which must be in `observation_space`), the reward seen, a boolean informing whether the episode ended or not and a dictionary containing additional information. These variables are usually referred as `observation`, `reward`, `done` and `info`.

We extended the `Env` interface with a `GameEnv` class, more specific to board games but keeping the essence of *OpenAI Gym*. It is implemented in the module `tfg.games` and adds two methods and one property to the original class. One method, `legal_actions`, is used to obtain a list with all legal actions in the current state, and the other, `winner`, returns the winner of the game (1 for white, -1 for black and 0 if it is a draw) if there

[1] https://numpy.org
[2] https://www.scipy.org
[3] https://www.tensorflow.org

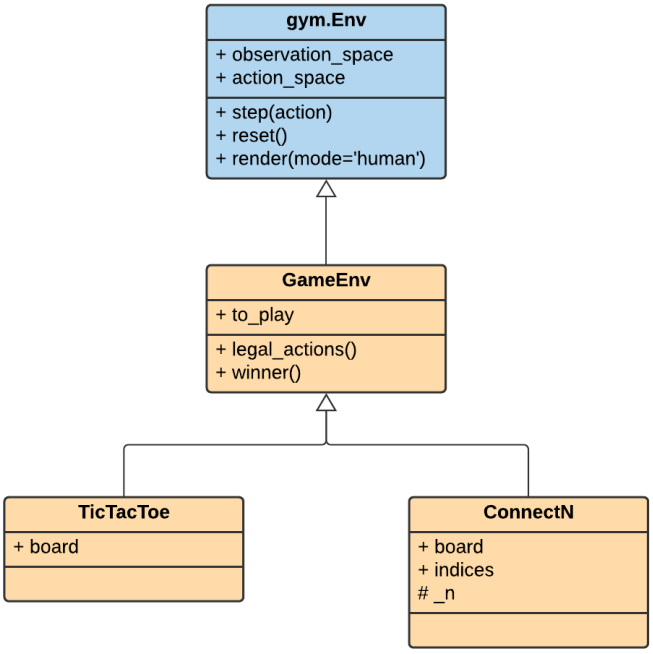


Figure 4.1: UML class diagram for `GameEnv` and its subclasses.

is one; otherwise it returns `None`. The property mentioned, `to_play`, represents which player has to move in the current state; again, 1 being white's turn and -1 black's.

Thus, we can easily play any game implemented with this interface without knowing anything about it. We just need to reset the game to start a new one and loop the `step` method until it returns `done`. We will know which player won just by observing the reward returned, which was 0 in all previous states, or by calling the `winner` method. This workflow is very similar to base *Gym*'s one except that we need to take turns into account as we have two agents, black and white.

Using this interface we created the two simple games we explained in Section 2.2: Tic-Tac-Toe and Connect N (`game.tictactoe.TicTacToe` and `game.connect_n.ConnectN`). We will use these games later in the experiments.

Figure 4.1 is a simple UML class diagram with the dependencies of all these classes (space classes have been omitted).

4.1.1 Tic-Tac-Toe

The representation of Tic-Tac-Toe is simple. We used a *Numpy* array of shape $(3, 3)$ to represent the board. If the element in (i, j) is 0 it means that the cell (i, j) is empty; otherwise, it can be either 1 if there is a white token or -1 if it is black. An action is represented as an integer $k \in \{0, \dots, 8\}$, standing for the k-th cell counting from top to bottom and from left to right. Consequently, the observation space of Tic-Tac-Toe is represented by the `gym.spaces.Box` class with shape $(3, 3)$ and the action space is a discrete space (`gym.spaces.Discrete`) with 9 elements.

Thus, a move in Tic-Tac-Toe is done by first checking that the cell is empty, that is, it is a legal move, and then writing the correct value in that position. After that, the board

will be checked to detect final positions. If there is a winner the line must pass across the last piece placed. For this reason, it is only necessary to check four lines at much: two diagonals, one vertical and one horizontal. On the other hand, if there is no winner we can check if there has been a draw checking the number of moves; after the ninth move there cannot be more, so the game is tied.

4.1.2 Connect N

We implemented Connect N's board the same way as in Tic-Tac-Toe but we needed an additional structure to store the row of the first empty cell for each column. An action is again an integer $k \in \{0, \dots, C-1\}$, where C is the number of columns, representing the column where the token is dropped. As in Tic-Tac-Toe, the action space is discrete with C elements and observations comes from the box space of shape (R, C), where R is the number of rows.

This time, to make a move we first need to access to the structure that tells us which is the first free row in the given column. If there is no free row the move is not valid; otherwise, the token is placed in that row. To check the board we can copy what we did in Tic-Tac-Toe, but now it is usual that N is smaller than R and C so we do not need to traverse the whole line. Also, it is not necessary to check the cells above the last placed token because we know they are empty.

4.2 Strategies and MCTS Implementation

If we only had games, we could only play them manually. In order to have automatic agents play we created the `Strategy` interface in `tfg.strategies`, which essentially defines the `move` method. This method takes a state as its only parameter and returns a valid action. For example, `HumanStrategy` renders the board and asks the user to input an action which will be returned by the method. This way, we can manually test other strategies. Other important one is `Minimax`, which implements the Minimax algorithm and allows the use of AlphaBeta prune. Thus, we have a perfect rival for short games, such as Tic-Tac-Toe, or, depending on heuristic, a decent rival for longer ones, like Connect N.

For this purpose, we implemented a simple heuristic for Connect N. This heuristic takes an integer n as a parameter and counts how many times each player has n non-blocked tokens in a row. This means that pieces surrounded by rival's pieces or the edges of the board are not taken into account, because it is impossible to place more in that same line. Once counted for each player, it returns the difference divided by some factor that ensures this number is between -1 and 1 (for example, the number of cells in the board). If we did not do this, the Minimax algorithm would consider that it is better to have many times n pieces aligned, rather than N just once. This is because the environment returns 1 or -1 when a player wins. For instance, Figure 4.2 is an example of the application of this heuristic to a particular board. Obviously, this heuristic is far from perfect, but will make Minimax try to align as much tokens as possible, so it will have more winning chances.

But the main objective of this module is to implement the MCTS algorithm. This is done by the `MonteCarloTree` class. We will use it for move selection in AlphaZero but we also

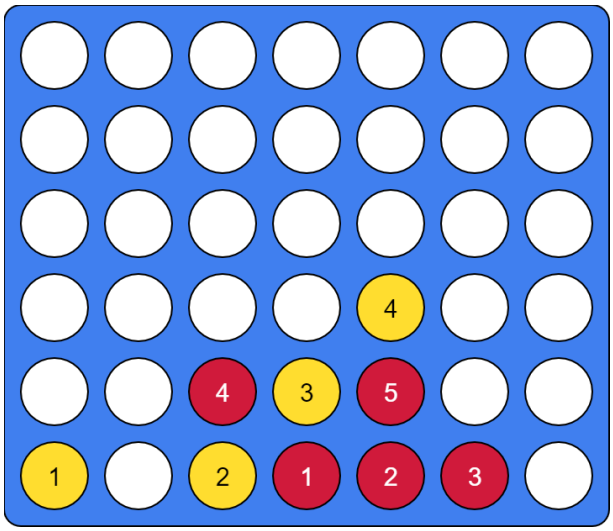


Figure 4.2: Example of the application of the heuristic for Connect 4 with a borad size of 6×7. If $n = 2$, the red player (usually referred as white) has five pairs of tokens aligned: $(1, 2)$, $(2, 3)$, $(1, 4)$, $(1, 5)$ and $(3, 5)$, whereas yellow (usually black) has only two: $(2, 3)$ and $(3, 4)$. $(2, 5)$ is not counted for white because this line is surrounded by a yellow token on the top and the edge of the board on the bottom. Thus, the result of the heuristic will be $+3$ divided by some constant. However, if $n = 3$ the result will be zero, because both players have three tokens in a row.

wanted it to work perfectly as a strategy itself. Therefore, we made it very versatile.

It takes the game environment as its first argument in the constructor, but also the maximum number of iterations or the maximum time allowed for each call to the `move` method; a selection policy, such as OMC, PBBM, UCT or any other custom callable; a value function that completely substitutes the simulation phase, as AlphaZero's neural network does; an update function that allows storing arbitrary attributes in the tree nodes; and a function that selects the returned action based on some attributes of the nodes, such as their visit count, their value, or any other. Additionally, we added the ability to recycle the tree from the previous move, so that it does not lose the information about the child nodes that have already been visited.

We also implemented some selection and final move policies, even though they are not used in AlphaZero, but enabled us to compare it to a basic MCTS. The UML diagram of this module is depicted in Figure 4.3.

The `move` method was implemented following the algorithm described in Section 2.3.2. The selection, expansion, simulation and backpropagation phases are executed in a loop while there is time remaining. That is, when neither the number of iterations nor the time elapsed has reached the maximum established. At the beginning of every iteration, the history of nodes traversed is initialized with the root node and the game environment is cloned.

Then, the selection phase starts and loops until a non-expanded node is reached. As every node has a mapping from action to child node, it is only necessary to call the selection policy function with the list of children and advance to the selected node. The selected node is then appended to the history.

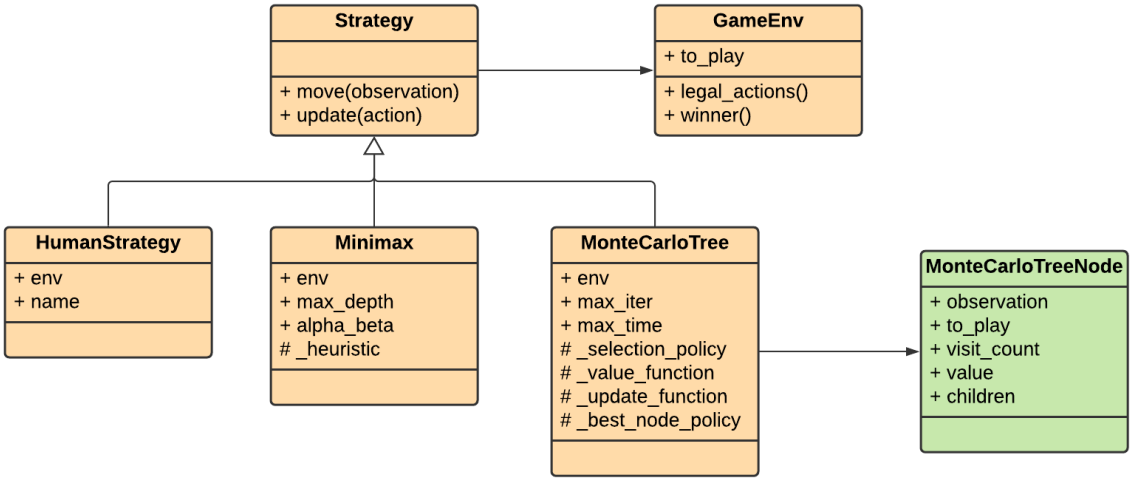


Figure 4.3: UML class diagram for `tfg.strategies`.

Next, the current node gets expanded. This implies simply calling the `step` method on the game environment once for each legal action and storing the results as node's children. Also, the update function is called on every child node in case some attributes needed to be initialized.

Once the node has been expanded the simulation or evaluation phase begins. Normally, a random game will be played from that position until a reward is found. But if there is an evaluation function it will be used to find the value of that node. Independently of where the reward came from, it will be backpropagated through the history list in accordance with the player to play in each node.

In the end, after all these iterations, the best node policy function will be called to determine which root's child will be selected and the associated action will be returned.

4.3 Neural Network Implementation

The implementation of the neural network is achieved by the `NeuralNetworkAZ` class implemented in `tfg.alphaZeroNN`. This class uses *TensorFlow* library and its *Keras* submodule to create the model for the neural network architecture specified in Section 3.3. When the class is instantiated with all the necessary parameters, it produces the model using all the components explained in Section 2.4. Also, when compiling the model, we have used either *Adam* or *SGD* optimizer.

As we have less computational power, we included the number of residual blocks and the number of filters in each convolutional layer as parameters. This way, we can create smaller networks. For instance, we can use only one residual block and 32 filters per layer, instead of 19 blocks with 256 filters, as DeepMind used. For instance, a concrete example of the former has 23,780 parameters, while the latter has 22,550,276.

The class also contains multiple methods to facilitate its use. The `fit` method is used to train the network and the `predict` to make our predictions. We have also implemented `save_model` and `load_model` which allows us to save the model and interrupt the training process when it takes too long, and load a specific model to resume training or play against

it. Additionally, we have added two extra methods: `summary_model` and `plot_model`, which provide us further information from our model, such as the number of parameters or an image of the architecture.

4.4 AlphaZero Implementation

Once we implemented both the MCTS and the neural network modules, we finally implemented an `AlphaZero` class (in `tfg.alphaZero`) which combines both of them and accomplishes the training loop process.

When instantiated, the class initializes all the parameters, MCTS and neural network of AlphaZero. As we described earlier, we implemented the `MonteCarloTree` class as versatile as possible, so that it takes a selection policy, a value function and a final move policy as parameters. Hence, we implemented the custom selection policy Equation (3.1), the final move selection policy Equation (3.2), and the value function, predicting with the neural network and adding Dirichlet exploration noise if node is root.

Moreover, two methods are used for the training loop: `self_play` and `train`. The former plays N games against itself and returns all the data collected (boards, π vectors and winners). When self-playing, each move is performed calling MCTS. An important detail is that games are played concurrently. In Section 4.5, we will explain how we achieved this parallelization. The latter loops until training time is over or error is lower than a threshold or the maximum number of played games has been reached. While looping, it calls `self_play` and stores the data in a shared buffer. Then, a mini-batch is extracted from the buffer and the data is converted to fit the network input format. Finally, we call the neural network `fit` method and start the loop again.

A sequence diagram of a training step can be found in Figure 4.4. It shows only one move of a self-played game and one iteration of MCTS.

In order to play against other opponents, this class also implements the `Strategy` interface, and its `move` method, which basically calls the MCTS's move method.

4.4.1 Adapter

When the data is collected from self-play, it needs to be converted to fit the neural network input format. This conversion corresponds to the one explained in Section 3.3. Specifically, the board and turn is converted into M binary feature planes of size $N \times N$. Nevertheless, these variables depend strongly on the game being played. Besides, the transformation of a particular action into an index of the probability vector of the neural network output is also dependent on the game considered.

The `tfg.alphaZeroAdapters.NeuralNetworkAdapter` class helps to solve these dependencies. It is an abstract class with two methods: `to_input` which transforms the board and turn to network input, and `to_indices` which converts an action into the corresponding index of the probability vector predicted. Consequently, when implementing a particular game, in order to use AlphaZero, we also have to implement its adapter for the game.

With the intention of clarifying the input conversion process, we are now going to provide a brief example of how we converted the Tic-Tac-Toe board and turn into $N \times N \times M$

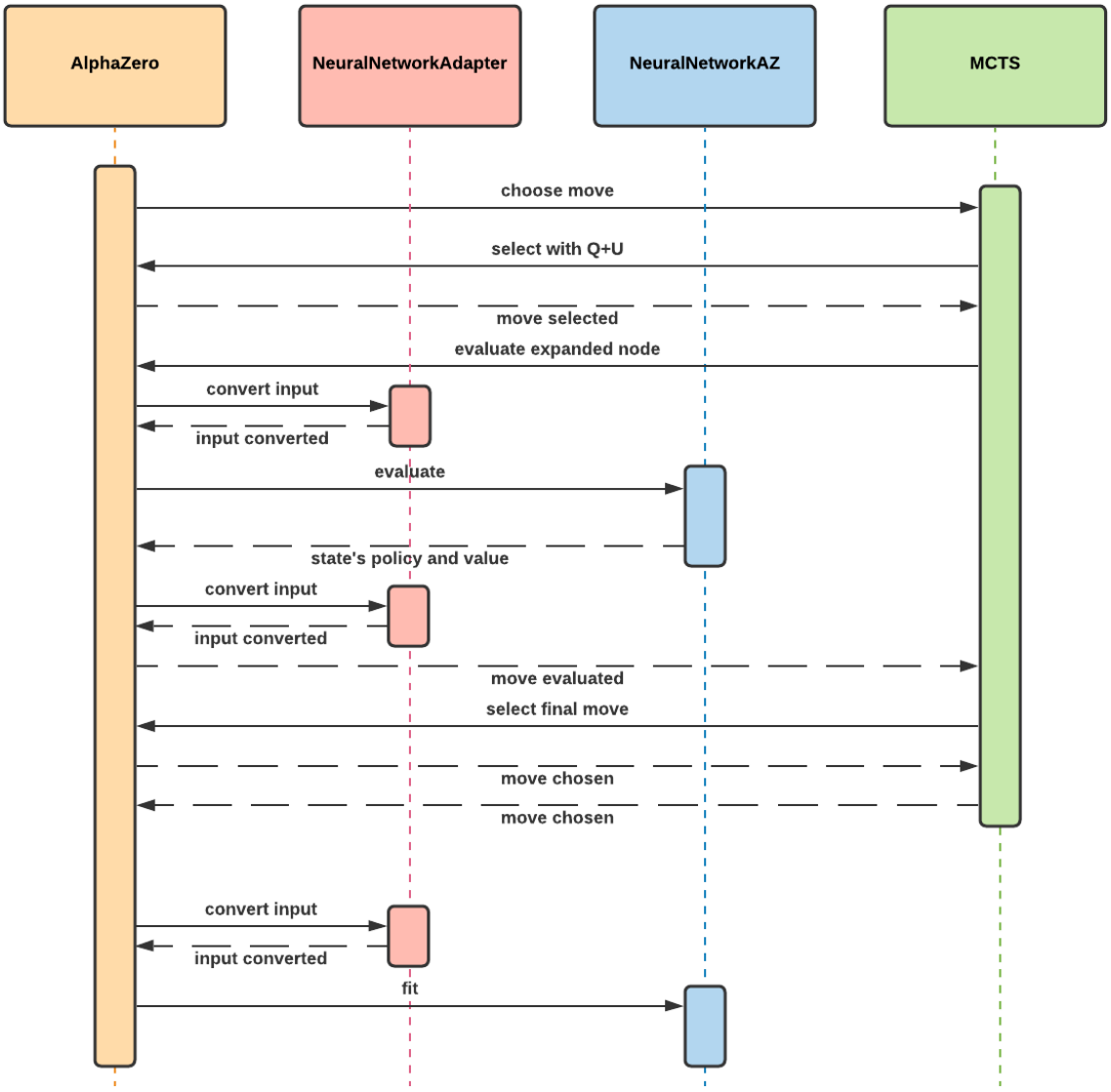


Figure 4.4: UML sequence diagram for a training step of AlphaZero. Self-play and MCTS are simplified to a single iteration of each of them and a single game. MCTS selection policies and update function return to AlphaZero because they are implemented there.

image stack. In this case, for each player there is only one piece type ($P = 1$) and we will only consider one time step ($T = 1$). Additionally, the extra information we need is the current player's turn. This way, $M = (2 \cdot 1 \cdot 1 + 1) = 3$, so that we need: 1 plane with ones in the positions of white player's pieces, 1 plane with ones in the positions of black player's pieces, and 1 last plane with all zeros if it is white's turn, or all ones if it is black's turn. The conversion is visually depicted in Figure 4.5.

Connect N's adapter is almost equal to Tic-Tac-Toe's. The $R \times C$ board and turn are converted into a $R \times C \times 3$ image stack with the same meaning as in Tic-Tac-Toe. The only difference is that here the board size is variable, so an instance of the game must be provided in advance. This is done in the adapter's constructor.

Finally, Figure 4.6 shows a complete class diagram centered in `AlphaZero`.

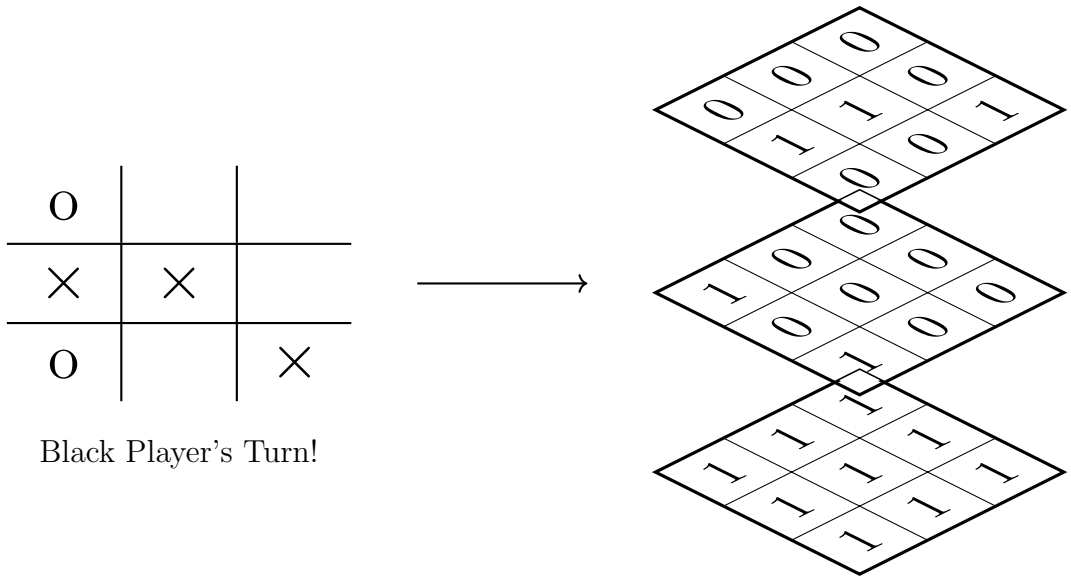


Figure 4.5: Input conversion.

4.5 Training Parallelization

We are going to dedicate a section to training parallelization because we had to make a different approach to this problem.

DeepMind researchers used 5,000 TPUs to generate self-play games and 16 to train the networks. We only had access to our personal computers with 8 to 16 logical CPUs and a single GPU. Thus, we had to find an efficient way of parallelizing training in our setups. Specifically, we started using a multiprocess parallelization and then added an additional threading parallelization.

4.5.1 Multiprocessing

First of all, we used a multiprocessing library. However, these types of libraries usually use a method called *pickling* to communicate state between different processes. This is a problem because our instances of `AlphaZero` contains an instance of the neural network class (`NeuralNetworkAZ`) that holds a *Keras* model, which cannot be pickled. For this reason, we had to find a different way of parallelizing without needing to pickle the model.

We first tried to keep the neural network in the GPU only and access it from the different CPUs. But we could not find a reasonable way of doing this. That was because we were not able to find a proper way of sending the input to a process with the neural network, while also using the barriers we added in Section 4.5.2. Therefore, we had to have a copy of the neural network in all the processors, but we could not copy it. Consequently, we had to create the processes first and then instantiate the networks. We were able to do it using a multiprocessing library called *Ray*[4] [38]. This way, we could split the computing into a cluster of machines if we had the infrastructure.

Basically, *Ray* allowed us to create `AlphaZero` instances as remote actors (in different processes), call their methods from the main program and retrieve their results there.

[4] https://ray.io

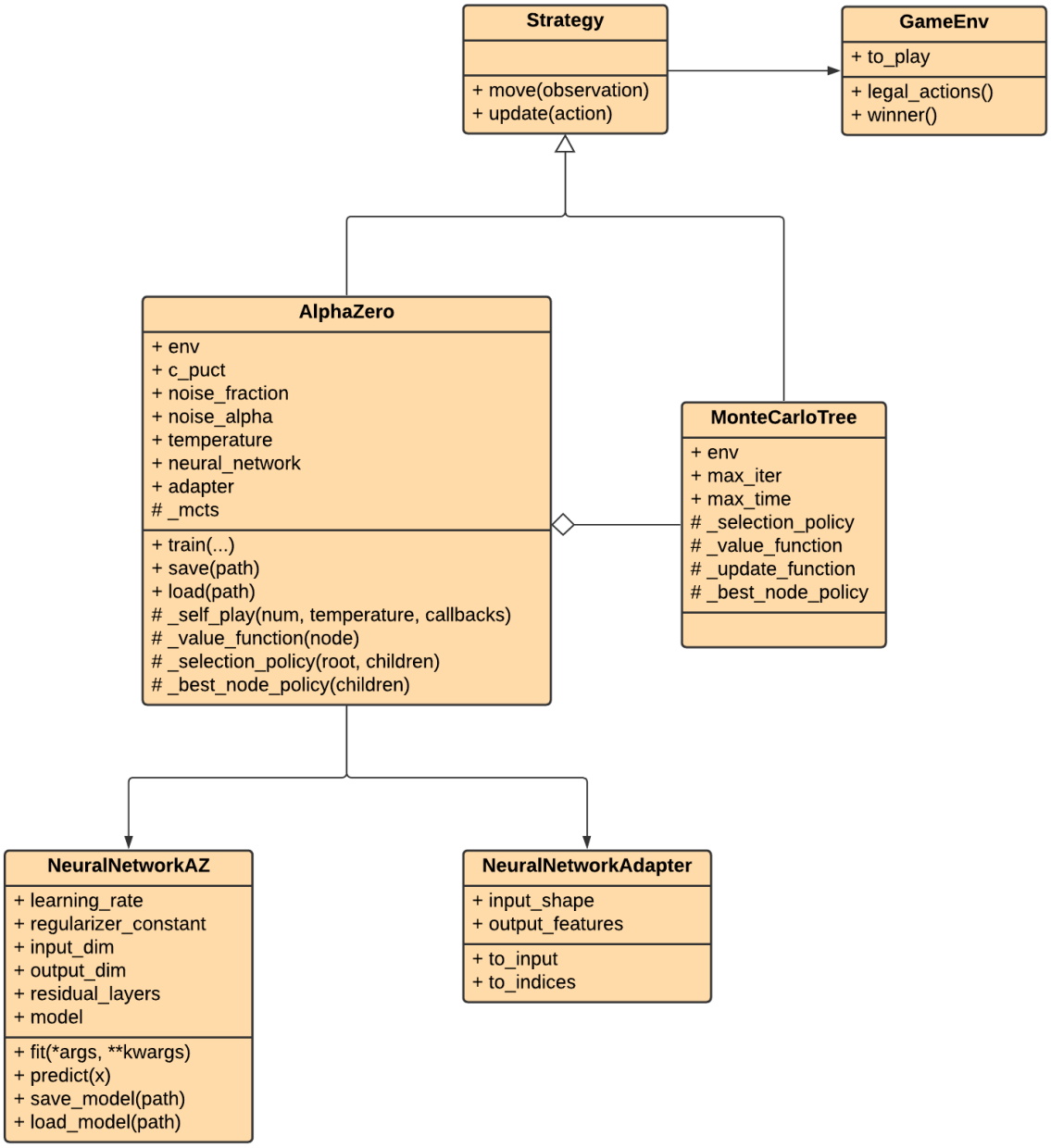


Figure 4.6: UML class diagram for AlphaZero.

For this purpose, we created an external function (`create_alphazero`) which instantiates a number of *Ray*'s remote actors. These actors are used to generate self-play games. However, their neural networks are built in their respective processes, hence they do not have access to the GPU, which makes self-play slower.

Every actor plays approximately the same number of games and returns their results to the main process, that will be waiting for all other processes to finish. This main process has another reference to an `AlphaZero` instance, but this one has its neural network built on the GPU. Then, it generates the input and trains the neural network. Finally, this process sends the updated weights to all the remote actors for the next batch of games.

With this method, we were able to reduce significantly the training time, in spite of the fact that network evaluation during self-play was made in a CPU instead of a GPU.

4.5.2 Multithreading

When we tried to train with a bigger game we found out that the biggest bottleneck was game generation. Specifically, AlphaZero is very inference heavy during self-play. MCTS requires hundreds of position evaluations for a single move. This is because neural networks are more efficient if data is fed in batches instead of one element at a time. And that was exactly what we were doing: after expansion phase in MCTS the value function was called for the expanded node. Thus, only one state was being evaluated at a time in every copy of the neural network.

Nevertheless, each process was usually playing around ten consecutive games before training the network, so we could use this situation to gather more boards before feeding them to the neural network, hence reducing the number of inferences needed.

For this reason, we used *Python*'s threading library, which allowed us to execute the MCTS algorithm at the same time for all games in a processor. For instance, if a given processor was assigned ten games, instead of finishing one and starting the next afterwards, it will make ten concurrent calls to MTCS' `move` method until all games have finished. In fact, as *Python* uses the Global Interpreter Lock (GIL) [39], games are not really played in parallel, but rather they are alternated in the single thread.

However, we needed a way of collecting all boards before calling the inference method. We did it by giving every thread an index to save their respective current boards in a common buffer. After accessing the buffer they will wait in a barrier until all other threads have done the same. Next, all threads will be awaken but the first one will evaluate the input buffer in the neural network and store the result in a common output buffer. Finally, all threads will be executing again and will be able to collect the results with the same index.

One final important remark is that some games are shorter and even some MCTS calls need less network evaluations than others. This could create some inconsistencies and cause deadlocks. To avoid this, after finishing the move selection algorithm all threads will have to pass the same barrier until all actions have been selected. This ensures that no thread is left in the barrier waiting for another that has already finished. The same applies if a thread has already finished its game.

Figure 4.7 shows a basic diagram of these two parallelization techniques together.

In addition to training parallelization, we also parallelized the playing function. We have two different playing functions. The first one (`tfg.util.play`) is intended to be used with any game and strategy. This function executes n games (maybe in parallel) and returns the results. Again, the problem was that the neural network was not *picklable* so we could only use this function for sequential playing. Thus, we had to create a specific function for AlphaZero: `tfg.alphaZero.parallel_play`. This one does the same as the previous but it imposes that one player is an `AlphaZero` instance, given by its weights file. Therefore, instances are created inside processes so they do not have to be pickled. This helped us speed up AlphaZero evaluations. Both versions of the play function use the *Joblib*[5] library, instead of *Ray* because this time all the parallel code can be executed inside the same function, so remote actors are not needed.

[5] https://github.com/joblib/joblib

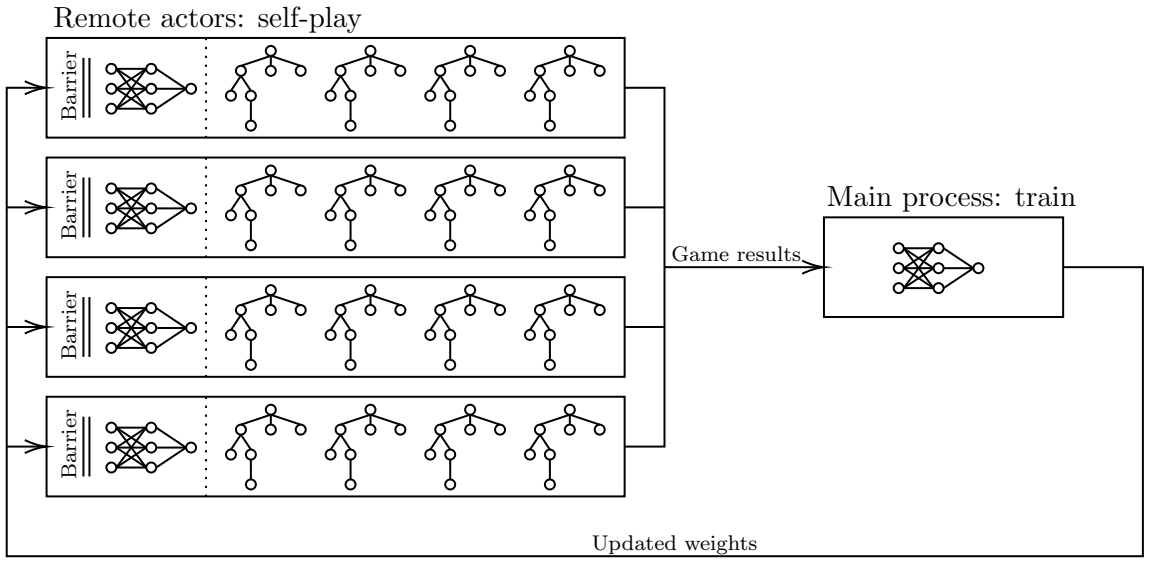


Figure 4.7: Outline of the algorithm with parallelism.

4.6 Repository

All the code reference along this chapter can be found in our *GitHub* repository:

`https://github.com/TFG-AlphaZero/Implementacion-TFG`

The main implementations can be found in modules `tfg` and `game`. The experiments we will present in Chapter 5 have been done in *Jupyter Notebooks* and are under the `experiments` folder. Finally, our trained models and checkpoints are inside the `models` folder.

Chapter 5

Experiments and Results

Now, we can start testing our implementation. For this purpose, we are going to train different instances of AlphaZero for Tic-Tac-Toe and Connect 4 and test them against other strategies. We have to use these rather simple games because we do not have access to such computational power as DeepMind's researchers did.

5.1 Tic-Tac-Toe

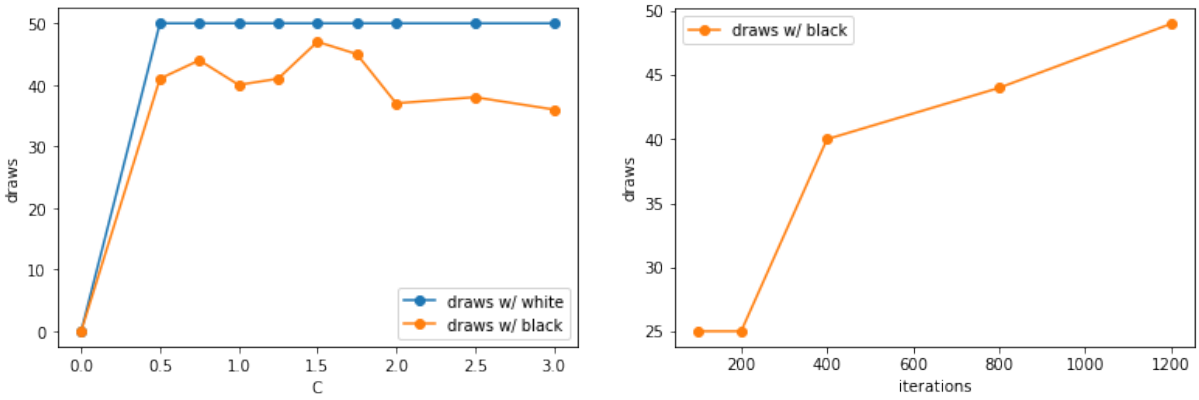
First, we are going to test our implementation with Tic-Tac-Toe. For this purpose, we are going to use three different agents: Minimax, MCTS and AlphaZero. In particular, we are going to use MCTS as AlphaZero's opponent, but we are going to compare them by making them face the same Minimax rival.

5.1.1 Tic-Tac-Toe with MCTS (no learning)

Before playing the matches, it is important to test the implementation of the Monte Carlo Tree Search algorithm as it is the one that AlphaZero uses inside. We are going to do this using Minimax with AlphaBeta prune as the rival because it can play Tic-Tac-Toe perfectly and the UCT algorithm for move selection as it is one of the most widely used. We will also compare different values for the UCT's C constant – even though in [21] they just used $C = \sqrt{2}$ – in case there were a better possible rival for AlphaZero for the next sections.

We did a selection of possible C constants between 0 (greedy selection strategy) and 3 (making uncertainty very relevant) and played 100 games with each of them, 50 with MCTS as white and 50 as black. The maximum number of iterations was set to 500, a sufficiently high number to make it possible for MCTS to draw some games, but low enough so it does not tie all of them.

Figure 5.1a shows the results of this match. As we know Tic-Tac-Toe's perfect outcome is a tie, we will be only counting draws, because Minimax will not be able to lose a single game. Thus, games that are not drawn must be MCTS' losses. Results demonstrate that a completely greedy selection is useless, because it can only select the first state the tree visits. Also, the best C value might be near 1.5, which is close to $\sqrt{2} \approx 1.41$; additionally, Tic-Tac-Toe is so simple that only few iterations are needed to obtain a draw with white,



(a) C constant vs draws.

(b) Number of iterations vs draws.

Figure 5.1: a) Number of draws achieved by MCTS against a Minimax player with respect to UCT's C constant's value, playing both black and white. b) Number of draws as black with $C = 1.5$ when the number of MCTS iterations is increased.

no matter which exploration constant has been chosen, hence we will only focus on black from now on.

With the C constant selected we later compared different number of iterations to find out how many are necessary to reach perfect play in Tic-Tac-Toe. Therefore, we selected $C = 1.5$ for the exploration constant and iterations between 100 and 1,200. We only collected draws with black, as we explained above, and obtained the results displayed in Figure 5.1b. We can check that more than 1,200 iterations are needed to achieve perfect play, because even with this number MCTS lost one game.

5.1.2 AlphaZero's Parameter Tuning

Now that we have a rival for AlphaZero we need to find out which training configuration is the best for Tic-Tac-Toe. For this purpose, we trained for 100 games several instances with different hyperparameter settings. We made variable the learning rate, the presence of L_2 regularization, the number of filters in every convolutional layer, the PUCT constant and the α value in Dirichlet noise (Dir(α)). Table 5.1 summarizes all selections of these variables.

We kept some other hyperparameters constant because we thought what it would not be necessary to change them. The number of residual layers was set to 1 because Tic-Tac-Toe is a small game. The kernel size was 3×3, the same DeepMind used. We used a temperature of 100, as we thought it would not be useful for this game. Finally, we set *Adam* as the optimizer.

Self-play games were played using MCTS with 100 iterations and 25 % exploration noise weight in a computer with 16 GB of RAM, using 10 logical processors of an *Intel Core i7 10700F* CPU to generate games and a *Nvidia GeForce RTX 3060Ti* GPU with 8 GB of VRAM to train the network. Each training lasted about 30 seconds. During this time, 100 games were played and the neural network's weights were updated twice (once after 50 games and once in the end), fitting 384 boards for 5 epochs each time. As we noted in Section 5.1.1, 100 iterations are too few for a MCTS algorithm, so it is expected that not all games will end it a draw. Thus, the combination of hyperparameters with the highest

ID	Learning rate	L_2 regularization	Filters	C	α
A	10^{-2}	0	16	1.0	1.0
B	10^{-3}	0	16	1.0	1.0
C	10^{-3}	0	32	1.0	1.0
D	10^{-2}	10^{-4}	16	1.0	1.0
E	10^{-3}	10^{-4}	32	1.0	1.0
F	10^{-2}	0	32	1.0	0.5
G	10^{-3}	0	32	1.0	0.5
H	10^{-2}	10^{-4}	32	1.0	0.5
I	10^{-3}	10^{-4}	32	1.0	0.5
J	10^{-3}	0	32	1.2	0.5
K	10^{-3}	10^{-4}	32	1.2	0.5
L	10^{-3}	0	32	0.8	0.5
M	10^{-3}	10^{-4}	32	0.8	0.5
N	10^{-3}	10^{-4}	64	1.0	0.5

Table 5.1: Variable hyperparameter settings for Tic-Tac-Toe.

ID	H	I	M	D	F	K	L	J	G	C	N	A	E	B
Draws	48	47	47	46	46	46	44	41	34	25	25	24	17	13

Table 5.2: Results with black and 100 iterations after training, ordered by number of draws.

number of draws will be used in Section 5.1.3.

Table 5.2 displays the results of each set, sorted by the number of draws. There are six combinations over 45 draws, which is a really good result, considering only 100 games were played, hence only 768 boards were used to train the network. Top three settings have in common that they all used L_2 regularization, 32 filters and $\alpha = 0.5$. Conversely, they differ in the learning rate and the C constant. I and M were exactly the same except for the constant, while H had a higher learning rate (10^{-2}) but shared the constant $C = 1$ with I. Thus it seems that these kinds of combinations are the best but it is better to have a higher learning rate.

5.1.3 Final Results

Once we know the best combination for Tic-Tac-Toe we are going to train a new instance of AlphaZero for 1,000 games to have a more fitted network. For this purpose, we used the winner combination of variable hyperparameters and the constant ones. The configuration is displayed in Table 5.3.

Training took about 3:20 minutes to complete and ended with around 0.36 loss, but the value head loss was below 0.1. Complete training loss is shown in Figure 5.2. The graph has a wavy behavior because the targets change from one set of boards to the next, after the policy function has been modified.

To check if the neural network had learnt enough, we handcrafted some board examples and evaluated them with the network. We used an empty board, some states where the

Hyperparameter	Value
Residual layers	1
Filters	32
Kernel size	3×3
Optimizer	*Adam*
Learning rate	10^{-2}
L_2 regularization	10^{-4}
Epochs	5
Batch size	384
MCTS iterations	100
PUCT C	1
Dirichlet noise	25 % with $\alpha = 0.5$
Temperature	100

Table 5.3: Hyperparameters used during Tic-Tac-Toe training.

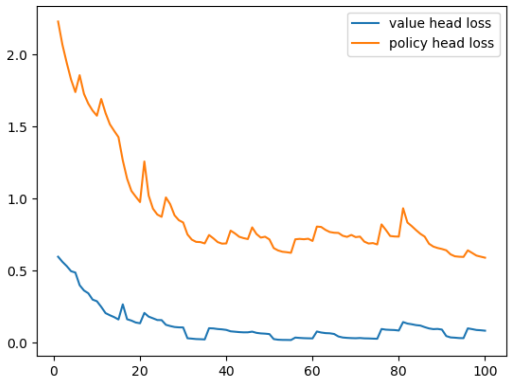


Figure 5.2: Value and policy heads losses during Tic-Tac-Toe training.

player could win in the next move, some others where the player had to avoid losing and some where the player will lose no matter what they do. Results are displayed in Figure 5.3.

The first one is an empty board, whose value should be 0 as Tic-Tac-Toe's optimal game is a draw. For the first move, the best is to place the piece in a corner, but any other move ensures a draw as well. It seems that AlphaZero has preference for the center.

Second, third and fourth boards are a succession that could have been taken from the same game, where white (`X`) is winning in all of them independently of black's moves. In the first one the network gives a slightly better value for black, but it gives higher probability to one of the winning moves (center cell). In the next one, white played a correct move but black played a wrong one. That makes white able to win in the next move. The network keeps giving a neutral value and is unable to find the correct move (the bottom-right corner). However, the selected move is also winning by force, but it takes more time. In the last board the move was already played and white won. This time the probability distribution is irrelevant but the value is important. White should be preferred. Unfortunately, the value yielded tells us again it should be a tied game.

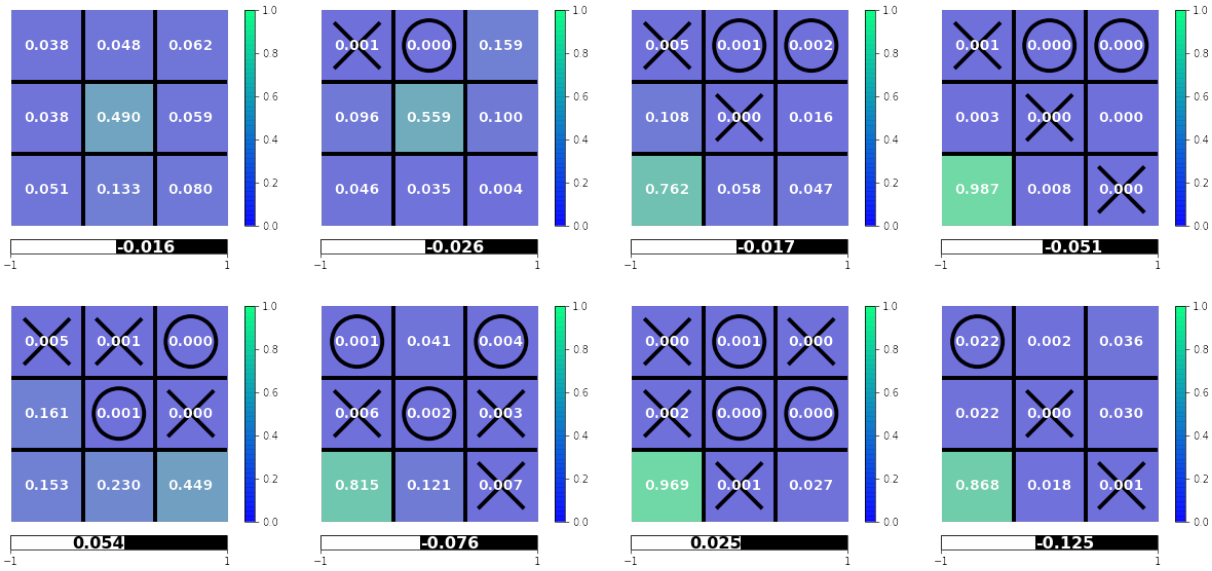


Figure 5.3: Prediction examples of AlphaZero's trained neural network for Tic-Tac-Toe. The numbers in every cell mean the probability of moving there, and the bar (and the number) below the board indicates which player the neural network prefers. The more white (more positive), the more AlphaZero thinks white will win, and vice versa.

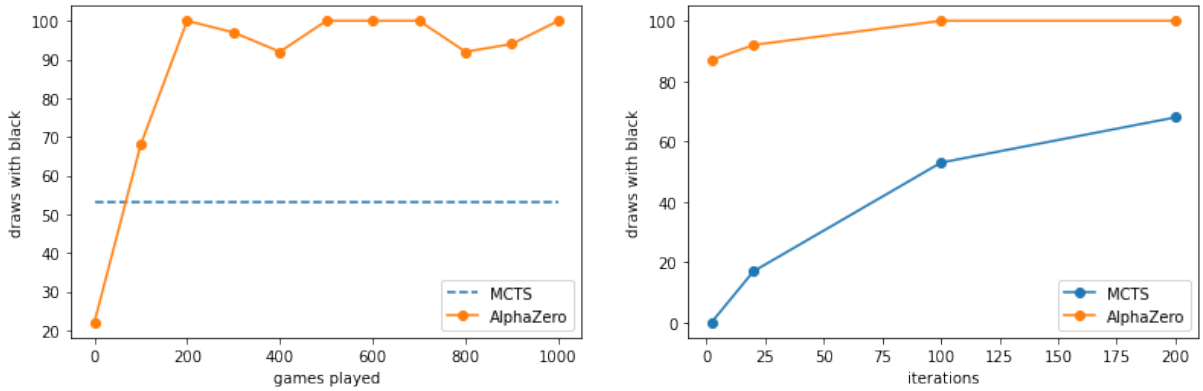
In the second row there are two examples where black is winning. However, the network does not give it a high value. It is interesting that in the first board the correct move is the one with the lowest probability of all legal ones. In the second to last board one move loses, whereas the other saves the game. The network chooses the correct one with a high probability and thinks it is a drawn game. In the last one any move but the corners loses the game for black and AlphaZero's network finds one of them.

In general, we can see that the value is not very accurate and sometimes the preferred move is not the correct one. The reason for this might be that the network needs more training, as we used very few examples in comparison with any supervised learning problem. In any case, this should not be a problem because we are also exploring with MCTS, which would fix any error the network could make.

Finally, we tested it against a simple MCTS to check how much improvement AlphaZero has introduced. MCTS will use UCT as selection policy with the default $C = \sqrt{2}$ constant because Section 5.1.1 proved that it is good enough for this game. Instead of playing against MCTS directly, we made both players play 100 games as black versus a Minimax with AlphaBeta prune player. Again, we will only take black's draws into account.

We made two different matches: the first one's goal is to compare AlphaZero's performance during training and the second one compared AlphaZero and MCTS using the same number of iterations.

We saved ten checkpoints during training and we used them to play the first match, using 100 MCTS iterations per move. We also made MCTS play to have a baseline to compare with. MCTS drew 53 games and AlphaZero beat this mark after only the first 100 games. It was capable of drawing 68 games. Figure 5.4a shows the complete result of this experiment.



(a) Draws during training.

(b) Draws after training.

Figure 5.4: a) Different instances of AlphaZero throughout training compared to MCTS. All of them are using 100 iterations. b) A trained instance of AlphaZero is compared against MCTS using different number of iterations for both of them.

In the other experiment, different instances of AlphaZero and MCTS played against Minimax four sets of 100 games using several numbers of iterations of the algorithm. First set used only one iteration, second used 20, third 100 and the last one 200. Results show the clear superiority of AlphaZero over a simple MCTS, as the former drew 87 games using only the neural network, whereas the latter was not able to reach 70 draws with the highest number of iterations. The complete plot can be seen in Figure 5.4b.

5.2 Connect 4

Now, we are going to train AlphaZero to play Connect 4, the version of Connect N with $N = 4$ and a board of size 6×7.

For this experiment we are going to introduce several changes. The first one is related to the size of the state space. Connect 4's state space is too large to be explored with a simple Minimax algorithm, even using AlphaBeta prune. Thus, we are going to use different opponents, which will not be perfect players. For this reason, we cannot expect the perfect outcome of the game (white always wins). We are going to use two types of opponents: Minimax and MCTS opponents. For Minimax we need to use a fixed depth and a heuristic. The heuristic h selected is the one explained in Section 4.2 that counts the number of times each player has n pieces in a row. Specifically, we are using $n = 2$. Thus, we are going to call Minimax(d), with $d \in \mathbb{N}$, the Minimax algorithm with depth d and heuristic h, and MCTS(k), $k \in \mathbb{N}$ the MCTS algorithm with k iterations.

Additionally, as now there will be very different results, AlphaZero will play both as white and black and we will count wins and losses, and not only draws as we did with Tic-Tac-Toe.

This time, we needed a bigger network, because it is a much bigger game. We used 3 residual layers instead of 1 and 128 filters per layer instead of 32. The kernel size stayed the same, as it is an usual value. As for the remaining hyperparameters, we reduced the learning rate to 10^{-3} and removed L_2 regularization, because it seemed to work better this way. The PUCT C constant and the Dirichlet noise were kept unchanged ($C = 1$, $\alpha = 0.5$

Hyperparameter	Value
Residual layers	3
Filters	128
Kernel size	3×3
Optimizer	*SGD*
Learning rate	10^{-3}
L_2 regularization	0
Epochs	5
Batch size	2048
MCTS iterations	400
PUCT C	1
Dirichlet noise	25 % with $\alpha = 0.5$
Temperature	30

Table 5.4: Hyperparameters used during Connect 4 training.

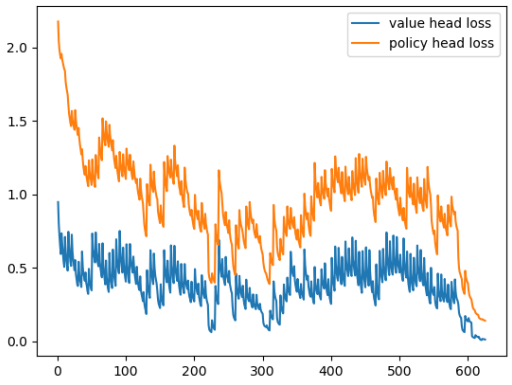


Figure 5.5: Value and policy heads losses during Connect 4 training.

weighted by a 25 %). Since Connect 4 games are longer, we incorporated temperature with a value of 30. This means that after 30 moves actions will be selected greedily with respect to the visit count of the node. For the same reason, we used a higher number of MCTS iterations, 400 instead of 100. Finally, we played 10,000 games, training the network after every 80 games, using 2,048 positions randomly sampled from the self-played games and 5 epochs per training. Table 5.4 summarizes the training configuration.

Training took 18:40 hours to complete. Training loss graph is shown in Figure 5.5.

For evaluation purposes we are going to use the final network and we are going to increase the number of MCTS iterations to 800. It will face five different rivals, namely Minimax(5), Minimax(6), MCTS(200), MCTS(400) and MCTS(600). It will play 50 games with white and 50 with black against every opponent. The results of these matches are displayed in Table 5.5.

These results show us several facts. First of them is that Minimax is a weak opponent, probably because of the heuristic selected. Anyway, we have to note that Minimax(d) can look up to 4^d leaf states in average, because the branching factor of Connect 4 is 4. So

White	Results	Black
AlphaZero	45–5	Minimax(5)
AlphaZero	46–2	Minimax(6)
AlphaZero	30–19	MCTS(400)
AlphaZero	31–19	MCTS(800)
AlphaZero	25–25	MCTS(1200)
Minimax(5)	14–35	AlphaZero
Minimax(6)	10–38	AlphaZero
MCTS(400)	23–22	AlphaZero
MCTS(800)	17–29	AlphaZero
MCTS(1200)	29–19	AlphaZero

Table 5.5: Match results for Connect 4. Every match consisted of 50 games. If results do not add up to 50, then the difference is the number of drawn games.

Minimax(5) might be looking around 1,024 leaf states and Minimax(6) 4,096. These are much higher numbers than the 800 states that AlphaZero can look in total. Therefore, the neural network introduces a better estimation of the states than any standard Minimax heuristic, hence much fewer states need to be checked.

Finding a goof Connect 4's heuristic is not the aim of this project, so we are going to focus more on the matches against MCTS. Theoretically, AlphaZero should be able to win first two MCTS rivals with white and at least draw against the second with black. This is because first MCTS uses less iterations than AlphaZero does and the second uses the same and white has advantage in this game. However, results exhibit a better performance than the expected. AlphaZero easily won both first rivals with black and white and drew against the third. Surprisingly, MCTS(800) behaved worse than MCTS(400). We blame this on the size of the game. It makes 800 iterations not enough and the randomness of the simulation phase gains more importance.

In summary, we achieved to train an agent capable of playing as well as a MCTS algorithm that uses more iterations than AlphaZero does. This is a great improvement, but we could have gotten better results if we had had more computing power. We have to note that we have a much slower hardware than DeepMind had, so we do not have the capability to play that many games in that little time. For instance, DeepMind's AlphaZero was trained to play chess in 9 hours using 44 million games in total, whereas ours spent 18 hours playing 10,000 Connect 4 games using a smaller network.

If we had a way of playing a "reasonable" number of games in a "decent" amount of time we could probably achieve better results. We could increase the number of games played between one weight update and the next, so that the network is trained with a wider variety of states. This might help it generalize better. And logically, playing one game in less time would allow us to train with many more games. For example, if we could play 1,000 games in 10 minutes, we could train AlphaZero with 100,000 games in around 17 hours, which may be enough to win all these opponents and even harder ones. But we are too far from it at the moment.

Chapter 6

Conclusions

Game theory is a field that has been studied for numerous years. Since Minimax and AlphaBeta, a lot of effort has been put into the implementation of efficient and accurate heuristics to create automatic players of games that were usually reserved to humans, such as chess or Go. However, with the appearance of Monte Carlo Tree Search, other approaches have been possible. And finally, AlphaZero has been a game changer in this field.

The aim of this project was to imitate what DeepMind's researchers did in a much smaller scale. We wanted to create a general purpose algorithm capable of playing different games, Tic-Tac-Toe and Connect 4 in our case.

But before this we had to make sure we comprehended all algorithms behind it. That is, we had to deeply understand MCTS and neural networks and gain a basic knowledge about reinforcement learning: main definitions and fundamental algorithms. Specially, we focused on understanding the exploration-exploitation balance problem and did a simple example about it. Additionally, we wanted to compare the AlphaZero algorithm to others, so we also studied Minimax and AlphaBeta.

With all this, we could start studying DeepMind's AlphaZero paper. We spent some time trying to understand how all these pieces were put together to create such an algorithm. For this purpose, it was useful for us to follow a step-by-step execution example.

Having studied all theory about these algorithms we could start our own implementation. Our main goal for this part was to make it as much extensible and reusable as possible. For this purpose, we implemented the game interface, which allow us to use them in the algorithms without knowing specific information about them. This would allow us to add more games if we wanted, without changing much code. As for the algorithms, we also tried to make them as versatile as possible. For instance, we made the selection policy in MCTS a parameter, so we could use AlphaZero's policy or any other, like UCT. Even AlphaZero's neural network was very parametric, which enabled us to make it bigger or smaller, depending on our needs. To help detach the boards from the neural network we added an intermediate adapter, which not only makes it possible to adapt more games to our AlphaZero implementation, but it can be also used to test different representations of the same game, without having to change any code inside the game class.

We did not try hard games, like chess or Go, because we knew it would not be possible

for us to train AlphaZero in our personal computers. For this reason, we implemented Tic-Tac-Toe and Connect N for testing purposes. At the beginning, we thought they would be simple enough games for the simplicity of their rules. However, we soon realized Connect 4 was not that simple, and training was too slow. Because of this, we had to pay special attention to parallelism. We used two different approaches: multiprocessing to be able to play multiple games at the same time and multithreading, to gather more states before feeding them to the network.

Finally, we did some experiments with our implementation. Due to its simplicity, Tic-Tac-Toe mainly allowed us to test if our implementation was correct. Furthermore, our trained instance for Tic-Tac-Toe achieved perfect play with few training games, but also played really well using the network only (with no search). Once we had made sure it was working properly, we were able to start testing it against a more challenging game, Connect 4. We also obtained satisfactory results in this game, but with a bigger network and much more training time. This instance of AlphaZero was not a perfect player as the other, but it was able to win widely against a weak Minimax and was superior than MCTS. It lost just one match, but it was playing with black versus a MCTS that used more iterations, so it was expected.

6.1 Review of Objectives

We had three initial objectives: learn the theory, implement our own version of AlphaZero along other algorithms and test it using two different games.

On the first goal, we can say we achieved it. First, we studied different techniques used in this field, like Minimax, MCTS and deep reinforcement learning. Once we had comprehended everything, we could start reading and understanding AlphaZero's paper. Finally, we did a simple step-by-step example following the ideas explained in the article. Having done this, we can claim we accomplished the first objective.

As for the second point, we implemented a highly versatile version of the algorithms and games. In particular, it is important for us to have a common interface for all games so they can be used interchangeably and have external adapters that convert their boards to inputs and outputs to actions. This way, games can be perfectly used without needing to know anything about AlphaZero, as it is just another part of our modules. For the same reason, MCTS and Minimax algorithms, neural network and actual AlphaZero algorithm are also very configurable. For instance, many different selection policies, heuristics, architectures, or training settings can be used.

Finally, we obtained satisfactory results for our third objective. Our main success was to be able to train two different AlphaZero instances for two different games, namely, Tic-Tac-Toe and Connect 4, without needing to change one line of code and using no human knowledge. They learnt everything by self-play and their only prior knowledge were the rules of each game. These were the main purposes of AlphaZero. It is an algorithm easy to adapt to diverse games as it does not need any domain-specific information.

We were able to master Tic-Tac-Toe and trained a strong Connect 4 player. However, with our personal resources we do not expect better results and we do not think we can address harder games. Presumably, this should be possible with greater computational power, as DeepMind showed.

6.2 Future Work

There are several ways to extend and improve our work. Mainly, we would need to speed up our implementation so that we could train Connect 4 in much less time than we do now. For this purpose, we would focus on improving our parallelism techniques or adding additional enhancements. Specifically, we should be able to achieve playing games in multiple processors that are sharing a single GPU. We could also use a cloud service that allowed us to use more machines and even more GPUs. Apart from parallelism, we could add a board cache so that we do not need to reevaluate typical states. This way, board evaluations in MCTS would be done faster, hence, training too.

Once we could play games much faster, we might try to master Connect 4 again if possible, but also try it with harder games, such as checkers. Checkers is a much more challenging game due to its state space and branching factor.

In relation with that, bigger games as checkers might need to use several time steps as input for the neural network, as AlphaZero originally did. However, we did not add this possibility because the games we are using are rather simple and we thought it would not be necessary. Thus, adapters take only one board at a time. We could easily change it if needed. It would be only necessary to add an additional attribute in the adapters meaning the number of time steps that will be used as network input and additional minor changes in some algorithms. This improvement would make our implementation even more versatile.

As this project has its focus on two-player zero-sum bard games with perfect information, all algorithms have been developed to match these constraints. However, we could explore some generalizations of the AlphaZero algorithm. In particular, we could start with multiplayer games. For instance, Connect N could be played by multiple players without changing the rules. At first glance, we would have to modify our game interface so that the reward is not a single number in $[-1, 1]$, but rather n numbers in $[0, 1]$, meaning if each player won or lost (or if they all drew). Similarly, we might need to replace the value head of the neural network with a different one that also returns n values.

A different line could be to improve our implementation to allow stochastic games, that is, games where there is a random component, such as a dice. Backgammon is an example of such a game. This fact would complicate the search, because we cannot decide which state we are going to end up in. Presumably, the value would be averaged among all possible die rolls.

Additionally, we could also allow imperfect information games, such as card games (blackjack without bets, for example), where you cannot see your rivals' hands. These types of games usually go hand in hand with stochasticity, so it would be a much harder approach than the previous. Probably, it would be necessary to sample among all possible "perfect" states in order to compute the value of a single "imperfect" one. We might even try typical casino games, like poker or blackjack, which include bets and their rewards are how much money one wins or loses.

Aside from games, we could research other fields where this kind of algorithm could be applied in. In fact, mathematical game theory has not only been developed to play games, but also has several applications in economy or biology, to name a few examples.

Finally, we could also follow the same line DeepMind did. They worked on an improvement

of this algorithm that was able to master Go, chess and shogi, but also Atari games, but this time they did not tell it the rules of these games. This work culminated in the MuZero algorithm [40], that was released in December 2020, during the elaboration of this project.

Appendices

Appendix A

Personal contributions to the project

A.1 Pablo Sanz Sanz's Contributions

The first part of this project consisted on learning the basic parts that made up AlphaZero. We both needed to learn about everything, but each of us focused more on a different part.

I was in charge of the part related to combinational games. Thus, I was more centered on game theory and also on search algorithms (Minimax, AlphaBeta and Monte Carlo Tree Search). In addition, as we had to relate game theory with reinforcement learning, I focused more on the reinforcement learning part, but we both needed to have a basic background about this topic.

Then, we had to understand the AlphaZero algorithm, so we put our findings in common and applied them to it.

Similarly, we used the same splitting for writing this report. Therefore, I wrote Sections 2.1 to 2.3.

In the next part, the implementation, we could separate our work more easily. As I studied more theory about games, I was in charge of the implementation of the game environments (`GameEnv`) and the strategies used for them, namely, `HumanStrategy`, `Minimax` (which can use AlphaBeta pruning) and `MonteCarloTreeSearch`. Specially, for the MCTS algorithm I needed to make sure that my implementation was suitable for my partner's AlphaZero implementation, but also wanted it to be used as a strategy itself. Thus, I added several parameters so that it can be used either way. This also meant that I had to code some selection and final move policies, such as `UCT`, `OMC`, `PBBM` and `SecureChild`, even though we only used `UCT` selection policy in the end. Again, I was in charge of writing the associated sections: Sections 4.1 and 4.2.

I also helped my colleague improve the training process. One of these improvements was the creation of the adapter interface (`AlphaZeroAdapter`). Initially, he had added the board conversion inside the evaluation function for simplicity and testing purposes. But we needed a way to extract this outside the `AlphaZero` class to be able to play different games without needing to change its code. Thus, I created the main interface and refactored this code as an implementation of the interface.

Additionally, I created a `Callback` (`tfg.alphaZeroCallbacks`) interface to be used for

training purposes. With some extra code inside the training loop we could do some additional useful operations. We only used the `Checkpoint` callback that allowed us to save the model in different moments during training. This way, we could evaluate the performance of AlphaZero along training, as we did in Section 5.1.3. We implemented other callbacks that we did not need in the end, such as `ParamScheduler`, to change some hyperparameters (learning rate, MCTS iterations, etc.) during training, or `GameStore`, intended to be used to save all games that were played during training.

Once we had our implementation of AlphaZero and realized it was too slow, I took some time looking for ways of parallelizing it. Initially, I tried simple approaches, like splitting training into multiple processes hoping that they all would have access to the GPU. However, I quickly discovered that *TensorFlow* initializes all its GPU variables when it was first imported. This meant that the memory filled with all these variables and the program crashed. Therefore, I had to find a way to either disable GPU usage or use a single neural network allocated in the GPU. Therefore, I tried the first approach with the same library we were using for the `play` function, *Joblib*. Nevertheless, we only wanted self-play to happen in parallel, but the weight update had to occur only once in a main process. For this reason, we could not use this library, because we would have had to create all AlphaZeros inside the parallel functions for every self-play batch.

Finally, I found *Ray*, that allowed us to have remote actors and call their methods from the main process. Hence, I implemented the new version of the training loop, imitating what my partner had done, but with parallelism. But it was still too slow for Connect 4. After that, our director suggested us how to accumulate boards before evaluating them with the neural network, instead of feeding them one by one. I tried this idea and achieved the threading parallelization. Consequently, I implemented the parallelization part, hence wrote Section 4.5.

The final part was composed of training AlphaZero and evaluating it. We both did some testings, but I was the one that did the final executions because my computer is faster than my colleague's and I have also access to a new generation GPU. For this reason, I was the one that wrote Sections 5.1 and 5.2.

After debugging our implementation and having made sure it was working we could evaluate it. As training Tic-Tac-Toe took much less time, I could try numerous hyperparameter settings to check which was the best and see the results immediately. In contrast, Connect 4 was much slower, as we have already explained. Thus, I tested a small number of hyperparameter combinations with few training games to have a quick overview of which kinds of settings seemed to work better. Once having selected a set of hyperparameters I started a new training session. It was supposed to last around 20 hours, from one afternoon to next midday. Finally, that combination seemed to work well, and we achieved the results presented in Section 5.2.

A.2 Juan Carlos Villanueva Quirós' Contributions

As we have previously explained, in the first part of the project we tried to study the fundamentals and investigate how AlphaZero worked. Personally, I thought it was an extraodinary opportunity to learn about a field that fascinated me. Although we both made sure to understand every single concept, we distributed the work and each of us would focus on a specific part. In my case, I focused on the deep learning part and training process.

I had to read and comprehend every single component involved in the deep neural network architecture that AlphaZero used. For instance, I understood how convolution operation worked, how the architecture of the neural network was organized, and the conversion of the game representation into the neural network input.

In the implementation part, we used the same splitting for the organization of the coding. We implemented the parts accordingly to what we investigated in the first part. Hence, I was in charge of the implementation of the modules `alphaZeroNN` and `alphaZero`.

In the first one, the neural network model is created using *TensorFlow* library. I was not acquainted with *TensorFlow* and therefore I had to put a lot of effort into it. Choosing which functions were needed and guessing out how they worked was a long-lasting task. First, I had many complications trying to create two outputs (policy and value head) for the neural network. After that, when testing the neural network, I noted a different behaviour at training time and predicting time. After a long time trying to figure out what was happening, I discovered that it was intentional. Indeed, the batch normalization layer behaves differently in training and predicting process. In training, calculating the mean and variance is based on mini-batch, whereas in predicting, mean and variance is calculated using the batches it has seen during training.

In the `AlphaZero` module, it was needed to piece both the `MonteCarloTreeSearch` class and the `NeuralNetworkAZ` class together. Firstly, I implemented the functions required for the MCTS, i.e., the selection policy, the best node policy and the value function. The selection function applied the $Q + U$ formula and the best node function used the $\pi(a|s)$ formula, introducing the temperature parameter τ as explained in Chapter 3.

The value function was a little bit more complex. Before predicting with the neural network, the node had to be converted to network input format. Consequently, I created a function that performed the conversion of the node board into the input format of the neural network. But this function was game dependent, it was only suitable for the Tic-Tac-Toe. After discussing with my colleague about this, we agreed that it was better to create another module `alphaZeroAdapters` in which an adapter class was implemented for each game, and the adapter would become a parameter. Once the conversion and prediction was made, I applied the Dirichlet noise for the root node, which caused some difficulties because I had to obtain only the legal probabilities and interpolate them.

Furthermore, in order to replicate the self-play process, I created the `self_play` function. Here, a fixed number of games were played executing the MCTS until the games was done. When a game ended, its data was stored in the `game_data` buffer, using the `make_policy` function to return the π vector according to temperature parameter. For the training loop, I built the `train` function, which loops until the training is over. It self plays multiple games, stores the data in a shared buffer and train the neural network with a mini-batch.

When the first version of the implementation of AlphaZero was over, I proceeded to debug it. This was without any doubt the most tedious part. I spent hours and hours trying to figure out little mistakes that were preventing AlphaZero to learn properly. For example, in the training process, it reached a point in which the same game was being played again and again. There was not exploration at all and the learning stalled. After debugging a lot, I found out that in the best node policy, we were always choosing the node with the higher visit count, even when the temperature parameter was set to $\tau = 1$. This way, we were never taking a random sample from the categorical distribution as we have explained. Removing this random factor reduced exploration a lot and caused to play the same games, leaving a great part of the game tree unexplored.

Moreover, keeping the right perspective when storing the information in the MCTS caused a lot of headaches. When self-playing, we were not changing correctly the sign of the values in the back-propagation step. This caused AlphaZero to choose wrong actions most of the times. When I fixed this bug, the algorithm started to play Tic-Tac-Toe almost perfectly and it was such a great satisfaction for me.

Additionally, I also helped to test the algorithm with Tic-Tac-Toe. Unfortunately, my computer was not powerful enough and it was much slower than the one from my colleague, therefore I could not afford to test it with Connect N.

Finally, we wrote this memory distributing the chapters according to the distribution made in the investigation part. Thus, I wrote Section 2.4 for the deep learning part and Chapter 3 for AlphaZero explanation. I also wrote Sections 4.3 and 4.4 to explain the implementation details.

Bibliography

[1] Tord Romstad, Marco Costalba, Joona Kiiski, et al. Stockfish: A strong open source chess engine. https://stockfishchess.org. [Online; Accessed: 2021-05-31].

[2] David Silver, Aja Huang, Christopher Maddison, Arthur Guez, Laurent Sifre, George Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–489, 01 2016.

[3] DeepMind. AlphaGo. https://deepmind.com/research/case-studies/alphago-the-story-so-far#our_approach. [Online; Accessed: 2021-05-31].

[4] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.

[5] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[6] David Silver. Lectures on reinforcement learning. URL: https://www.davidsilver.uk/teaching/ [Online; Accessed: 2021-05-10], 2015.

[7] Christopher John Cornish Hellaby Watkins. *Learning from delayed rewards*. PhD thesis, King's College, Oxford, 1989.

[8] Richard Bellman. A markovian decision process. *Journal of mathematics and mechanics*, 6(5):679–684, 1957.

[9] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[10] William R Thompson. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25(3/4):285–294, 1933.

[11] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2):235–256, 2002.

[12] Martin J Osborne et al. *An introduction to game theory*, volume 3. Oxford university press New York, 2004.

[13] Louis Victor Allis. A knowledge-based approach of Connect-Four. *J. Int. Comput. Games Assoc.*, 11(4):165, 1988.

[14] John Tromp. Number of legal 7 x 6 connect-four positions after n plies. URL: https://oeis.org/A212693 [Online; Accesed: 2021-05-12], 2012.

[15] Pascal Pons. Connect 4 game solver. URL: https://github.com/PascalPons/connect4 [Online; Accesed: 2021-05-05], 2019.

[16] J v Neumann. Zur theorie der gesellschaftsspiele. *Mathematische annalen*, 100(1):295–320, 1928.

[17] Daniel James Edwards and TP Hart. The alpha-beta heuristic. *M.I.T. Artificial Intelligence Project Memo*, 1961.

[18] Guillaume Chaslot et al. Monte-Carlo Tree Search: A new framework for game AI. *Artificial Intelligence and Interactive Digital Entertainment*, 2008.

[19] Rémi Coulom. Efficient selectivity and backup operators in Monte-Carlo Tree Search. *Computers and Games*, 2006.

[20] Guillaume Chaslot, Jahn-Takeshi Saito, Bruno Bouzy, JWHM Uiterwijk, and H Jaap Van Den Herik. Monte-Carlo strategies for computer Go. In *Proceedings of the 18th BeNeLux Conference on Artificial Intelligence, Namur, Belgium*, pages 83–91, 2006.

[21] Levente Kocsis and Csaba Szepesvári. Bandit based Monte-Carlo planning. In *European conference on machine learning*, pages 282–293. Springer, 2006.

[22] Frederik Christiaan Schadd. *Monte-Carlo Search Techniques in the Modern Board Game Thurn and Taxis*. PhD thesis, Maastricht University, 2009.

[23] Guillaume Chaslot. *Monte-Carlo Tree Search*. PhD thesis, Maastricht University, 2010.

[24] Hao Dong, Hao Dong, Zihan Ding, Shanghang Zhang, and Chang. *Deep Reinforcement Learning*. Springer, 2020.

[25] Saad Albawi, Tareq Abed Mohammed, and Saad Al-Zawi. Understanding of a convolutional neural network. In *2017 International Conference on Engineering and Technology (ICET)*, pages 1–6, 2017.

[26] Clarifai Technology. Elemental structure of a convolutional neural network. https://www.clarifai.com. [Online; Accessed: 2021-05-20].

[27] Machine Learning Summer School 2015 Rob Fergus. Example of a relu operation. http://mlss.tuebingen.mpg.de/2015/slides/fergus/Fergus_1.pdf. [Online; Accessed: 2021-05-20].

[28] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[29] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[30] Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, and Aleksander Madry. How does batch normalization help optimization? *arXiv preprint arXiv:1805.11604*, 2018.

[31] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362(6419):1140–1144, 2018.

[32] Jiayu Lin. On the dirichlet distribution. *Department of Mathematics and Statistics, Queens University*, 2016.

[33] Medium David Foster. Alphazero's neural network architecture. https://adspassets.blob.core.windows.net/website/content/alpha_go_zero_cheat_sheet.png. [Online; Accessed: 2021-05-31].

[34] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[35] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[36] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[37] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym, 2016.

[38] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I Jordan, et al. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 561–577, 2018.

[39] Python. Global Interpreter Lock. https://wiki.python.org/moin/GlobalInterpreterLock. [Online; Accessed: 2021-05-20].

[40] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, 2020.