Master's Thesis

Deep Reinforcement Learning with MuZero: Theoretical Foundations, Variants, and Implementation for a Collaborative Game

Contents

I: Introduction

In 2016, AlphaGo was unveiled - the first computer program capable of playing the complex strategy game of Go at a level surpassing human masters. To understand why this is a remarkable achievement, let's summarize how most traditional algorithms approach gameplay.

Traditional algorithms attempt to identify an optimal action by simulating numerous future moves and selecting the one with the best potential outcome. The underlying assumption is that examining enough moves will eventually reveal a strong move by chance. However, Go poses a significant challenge because of its expansive 19x19 grid, considerably larger than, for example, a chess board. Consequently, the branching factor is very high in the game tree: the number of possible outcomes escalates with each additional move searched ahead. To have a reasonable chance of discovering an appropriate move by chance, a prohibitively large number of options must be evaluated. As a result, traditional search approaches are computationally infeasible on the game of Go.

Even with an appropriate search approach, there remains the problem of evaluating specific board conditions. An accurate assessment of positions is necessary to guide the search towards victory. Go games can go on for hundreds of moves, making it hard to tell which positions are good or bad for either player in the long run, especially since a single move can have a significant impact on the rest of the game.

Nevertheless, in 2018, Silver, T. Hubert, et al. [2018] advanced upon this technology with AlphaZero, which was able to master three different board games: Go, Chess and Shogi. This is notable given that previous algorithms relied heavily on specific domain-specific information, such as the individual game rules. The ability of a single algorithm to excel in several distinct environments highlights its versatility - a key goal of reinforcement learning.

Like most search approaches, AlphaZero uses a simulator that must be explicitly programmed with the environment rules to determine the subsequent state following an action. This is not a limitation per se, in fact it is sometimes helpful or easier in some real-world applications. For example, Mankowitz et al. [2023] made use of the AlphaZero architecture to discover a faster sorting algorithm. In this particular case, the environment is presented as a game to modify an assembly program in incremental steps. The correctness and speed of the resulting program is measured to calculate the reward. An explicit simulator is a good fit here, since it guarantees that searched actions and their rewards are accurate, even for paths that have never been visited before. [Mankowitz et al. 2023]

In other scenarios, it is more difficult to provide a good implementation for the environment during the search. For instance, Mandhane et al. [2022] optimized video encoding to save bandwidth while preserving the visual quality. In this case, the task is framed as a game for a reinforcement learning agent, which has the ability to adjust

codec parameters. Therefore, video encoding and evaluation of the resulting visual quality must be managed in the game tree search. [Mandhane et al. 2022]

MuZero [Schrittwieser et al. 2020] is a further evolution of AlphaZero. The major difference is that it removes the need for an explicit simulator in the search phase. Instead, MuZero develops its own internal model to predict future environment states. In effect, it learns how to play games without having access to the underlying rules or dynamics. This way, even the above example of optimizing video encoding can be mastered. Perhaps surprisingly, MuZero also matches AlphaZero's performance in the aforementioned board games. Moreover, it also performs well on the Atari suite of 57 video games, demonstrating the ability to generalize across many domains. [Schrittwieser et al. 2020]

While MuZero's capabilities are impressive, the original implementation is limited in a number of ways. First, it was designed for one- or two-player games only. For two-player games, an additional requirement is that the game must be zero-sum, meaning that one player's loss is the other's gain.

The original implementation also can only handle deterministic environments. This excludes its application to games with chance events, such as dice throws.

The rest of this thesis is structured as following:
First, theoretical foundations of reinforcement learning, game theory and computers playing games are outlined. They lead up to an understanding of the MuZero algorithm and its precursors. After that, related algorithms are reviewed, highlighting their differences, strengths and limitations.

Building on this study of prior work and theoretical foundations, I present these contributions:

- a symmetrical variant of the EfficientZero[1] latent loss
- a modification to handle terminal states during the search
- an extension of MuZero to multiplayer games with chance events and arbitrary rewards

This thesis is accompanied by an implementation of MuZero that includes the proposed modifications. With this implementation, I perform three experiments in Section V: First, two ablation studies analyze the influence of each of the proposed modifications. Finally, I confirm my multiplayer extension is able to learn a collaborative multiplayer game.

[1] Another work on top of MuZero, by Ye et al. [2021]

II: Theory

II - A: Reinforcement Learning

Reinforcement learning refers to a subset of machine learning where a decision maker learns by trial and error while interacting with its environment. The decision maker takes actions that affect the state of their environment, receives rewards or penalties based on how it performs, and updates its behavior based on this feedback. The goal is to learn a behavior that leads to positive outcomes. [Sutton and Barto 2018]

In reinforcement learning, the entity responsible for learning and making decisions is called the agent. This agent exists in an environment which consists of everything outside the agent. Both interact continuously: The agent chooses actions, and the environment responds by presenting new situations to the agent. After each action, the environment provides a reward - a numerical value that the agent tries to maximize in the long term through its action choices. [Sutton and Barto 2018]

This concept can be mathematically modeled with Markov decision processes.

II - A 1: Finite Markov Decision Processes

Specifically, an agent interacts with its environment in discrete time steps $t = 0, 1, 2, ...$ At each time t, the agent receives an observation of the current state $s_t \in S$ and then chooses an action $a_t \in A$ to take. The action transitions the environment to a new state s_{t+1} and generates a numerical reward $r_{t+1} \in R \subset \mathbb{R}$ for the agent. S, A and R denote the set of all states, actions and rewards respectively. [Sutton and Barto 2018]

Note that the subscript used to represent an action that occurred between s_t and s_{t+1} varies in different literature: Some associate this action with time step t [Sutton and Barto 2018], others with $t + 1$ [Schrittwieser et al. 2020]. In this thesis, I will follow the first convention given in Sutton and Barto [2018]. This is visualized in Figure 1:

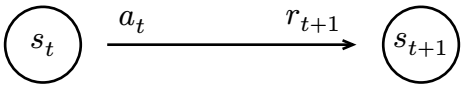

Figure 1: Transitions in an reinforcement learning environment

The concept of sequential decision making can be formalized with finite Markov decision processes. Finite means the sets of possible states S, actions A and rewards R each have a finite number of elements. Beside these sets, a Markov decision process is characterized by the dynamics function $p : S \times R \times S \times A \to [0, 1]$:

$$p(s', r|s, a) \stackrel{\text{def}}{=} \Pr\{s_{t+1} = s', r_{t+1} = r|s_t = s, a_t = a\}$$

for all $s', s \in S, r \in R, a \in A$.

It describes the probability of the environment ending up in state s' and yielding reward r when executing action a in state s. [Sutton and Barto 2018]

II - A 2: Episodes and Returns

In some scenarios the interaction between agent and environment naturally ends at some point, that is, there exists a final time step T. In games this happens when the match is over. In reinforcement learning such a series of interactions is called an episode. [Sutton and Barto 2018]

The sequence of visited states $s_0, s_1, s_2, ..., s_T$ is referred to as a trajectory. Depending on the context, a trajectory may also include the actions and rewards associated with the transitions between these states. The last state s_T is also called the terminal state. [Sutton and Barto 2018]

To translate multiple rewards earned over a period of time into a singular value that guides the agent in making optimal decisions, we use the concept of a return. The return is a function of the reward sequence. The agent's learning objective is then defined as maximizing the expected return[2]. In a simple case, the return G_t may be defined as the sum of all rewards occurring after time step t:

$$G_t \stackrel{\text{def}}{=} r_{t+1} + r_{t+2} + r_{t+3} + ... + r_T$$

Another approach is to use a discounted reward. The intuition is to value rewards far in the future less than immediate rewards. For this purpose, a hyperparameter γ is introduced, called the discount factor. The return G_t is then calculated as

$$G_t \stackrel{\text{def}}{=} r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... = \sum_{k=1}^{T-t} \gamma^{k-1} r_{t+k}$$

where $0 \leq \gamma \leq 1$.

The discount factor affects how valuable future rewards appear in the present moment: For example, a reward that arrives k time steps in the future will have its current value reduced by a factor of γ^{k-1} compared to if it had arrived immediately. [Sutton and Barto 2018]

II - A 3: Policies and Value Functions

A policy is a formal description of the agent's behavior. Given a state s_t, the policy $\pi(a|s)$ denotes the probability that the agent chooses action $a_t = a$ if $s_t = s$. [Sutton and Barto 2018]

Since the policy makes statements about the future behavior of the agent, one can now define the expected return: It describes the expected value of the return G_t in state s_t, if the policy π is followed. It is therefore also called the value $v_\pi(s)$ and defined as:

$$v_\pi(s) \stackrel{\text{def}}{=} \mathbb{E}_\pi[G_t | s_t = s] = \mathbb{E}_\pi \left[\sum_{k=1}^{T-t} \gamma^{k-1} r_{t+k} \;\middle|\; s_t = s \right]$$

[Sutton and Barto 2018]

[2] for details about how the *expected* return is defined, see Section II - A 3

The value is thus an estimate of how good it is for the agent to be in a particular state, measured by the objective function, the return. [Sutton and Barto 2018]

In terminal states, the value is always zero per definition: There are no future rewards possible. [Sutton and Barto 2018]

The definition of the value as the discounted sum of future rewards enables to calculate another type of return: The n-step return sums a truncated sequence of the next n rewards and substitutes the rest with the agent's current value function $v_\pi(s)$:

$$G_t \stackrel{\text{def}}{=} r_{t+1} + \gamma r_{t+2} + ... + \gamma^{n-1} r_{t+n} + \gamma^n v_\pi(s_{t+n}) = \gamma^n v_\pi(s_{t+n}) + \sum_{k=1}^{n} \gamma^{k-1} r_{t+k} \quad (1)$$

This use of the agent's current value function is known as bootstrapping. [Sutton and Barto 2018]

II - B: Game Theory

This section introduces basic game theoretic concepts, just enough to provide justification for the design and behavior of MuZero. It also provides foundations to discuss the limitations of the original MuZero implementation in Section IV - A.

Game theory is a broad interdisciplinary field that uses mathematical models to analyze how individuals make decisions in interactive situations. It is commonly used in economics, but has other widespread applications. [Ritzberger 2002]

In this thesis I am interested in game theoretic analysis of the behavior and interaction of multiple players in a game. Game theoretic considerations can be used to find optimal behavior in a given game under some assumptions. In this way, game theory can provide a foundation for what should or even can be learned by a reinforcement learning system. [Ritzberger 2002]

II - B 1: Basics

I begin with introducing some terminology and basic concepts:

A game in game theory is a very general concept and applies to more than what the common usage of the word "game" refers to. A game may denote any process consisting of a set of players and a number of decision points. Additionally, the information and actions available to each player at each decision point must be defined. [Ritzberger 2002]

When the game is played, each player has to make a choice at their decision points. This choice is often referred to as a move or action. A play of a game consists of the moves made by each player, so the decision points are replaced by concrete choices. Such a sequence of moves is also called a history. The history is also defined for any situation in the middle of the game. Since it contains all the moves made so far, it identifies the current state of the game. [Ritzberger 2002]

A game also needs to have a definition of its possible final outcomes. These are defined in terms of a numerical payoff, one for each player. Payoffs for multiple players are noted as a tuple, also called a payoff vector in this case. [Ritzberger 2002]

The games considered in thesis are finite, meaning that the number of possible choices is limited. Finiteness also applies to the length of the game, it should end at some point. This may sound trivial, since most games encountered in the real world are finite. However, game theory is not limited to finite games. It is therefore important to note that some of the game theory statements in this chapter only apply to finite games. [Ritzberger 2002]

Games that involve randomness, such as throwing a dice, are said to have chance events. These chance events can be seen as decision points where a special player, the chance player, has his turn. For example, a dice throw can be modelled with a decision point of the chance player with 6 possible actions, all with equal probabilities. [Ritzberger 2002]

A player's behavior is described by a strategy: It can be seen as a complete plan which describes which move to take at any given decision point of the game. [Ritzberger 2002]

Given a strategy for each player, one can calculate the probabilities of future moves and game outcomes. For any player and any state of the game, it is thus possible to derive the expected payoff, also called expected utility. [Ritzberger 2002]

Both game theory and reinforcement learning are about decision making, so naturally they employ similar concepts. However, comparable concepts have slightly different terminology in the two fields. Table 1 shows a tabular overview of related terms between game theory and reinforcement learning.

Game theory	**RL**
game	environment
player	agent
decision / move / action	action
strategy	policy
payoff	reward
outcome	sum of all experienced rewards
expected utility	value / expected return
history	trajectory

Table 1: Comparison of similar concepts in game theory and reinforcement learning

II - B 2: Properties of Games

Game theory may differentiate games according to different properties which are explained in the next chapters.

II - B 2.1: Payoff Sum

Zero-sum games are games in which the sum of all players' payoffs equals zero for every outcome of the game. It is a special case of the more general concept of constant-sum games, where all payoffs sum to a constant value. In other words, a player may benefit only at the expense of other players. Examples of two-player zero-sum games are Tic-Tac-Toe, chess and Go, since only one player may win (payoff of 1), while the other looses (payoff of -1). Also poker is considered a zero-sum game, because a player wins exactly the amount which his opponents lose.

Contrastingly, in general-sum games, no restriction is imposed on the payoff vectors. This situation arises, for example, when modeling real-world economic situations, as there may be gains from trades.

II - B 2.2: Information

An important distinction are games of perfect information and imperfect information. A perfect information game is one in which every player, at every decision point, has full knowledge of the history of the game and the previous moves of all other players. A player is thus fully and unambiguously informed about the current state of the game when he is at turn. Chess and Go are examples of games with perfect information. This is because each player can see all the pieces on the board at all times. [Ritzberger 2002]

Games with randomness can also have perfect information if the outcome of all random events is visible to the next player at his turn. An example is backgammon: when a player needs to make a decision, he has perfect information about what number the dice rolled.

Conversely, a game with imperfect information leaves players in the dark about the exact state of the game. A player at a decision point does not observe enough information to distinguish between several past game histories. He thus has to make a decision under uncertainty. The card game poker is an example of an imperfect information game because the other players' cards are concealed. [Ritzberger 2002]

II - B 2.3: Perfect Recall

The concept of perfect recall describes that a player never forgets all his past choices and information he got. This allows players to unambiguously separate past, present und future states. [Ritzberger 2002]

Perfect recall is often assumed in game theory, since it is a requirement for e.g. the rationality of players, which is explained in Section II - B 4.1.2. [Ritzberger 2002]

II - B 2.4: Simultaneous / Sequential Moves

Games can be classified according to whether the players make their moves at the same time or one after the other. In the first case, the moves are said to be simultaneous, and the game is also called a static game. [Fudenberg and Tirole 1991]

An example of a simultaneous game is Rock, Paper, Scissors: Both players choose their hand sign at the same time. Because of the static nature of the game, it is not possible to observe the action chosen by the other player and react to it. [Fudenberg and Tirole 1991]

In fact, non-observability is the defining aspect of simultaneous games: Take, for example, an election where all voters make their choices without knowing what anyone else has chosen. Even though the votes are not literally cast at the same time, the process is still an example of a simultaneous game. [Fudenberg and Tirole 1991]

In contrast, in a sequential game the players take turns in succession. These games are also called dynamic games or extensive form games. To distinguish them from simultaneous games, a player making a decision must have information about the previous decisions of other players. It is important to note that only some information is required, not necessarily perfect information. [Fudenberg and Tirole 1991]

II - B 2.5: Determinism

A game that involves no chance events is said to be deterministic. For instance, chess and Go are deterministic games since the game state only depends on the moves of the players. [Ritzberger 2002]

II - B 2.6: Cooperation and Collaboration

Traditional game theory divides games into two categories, cooperative and non-cooperative. The cooperative approach studies games where the rules are only broadly defined. In fact, the rules are kept implicit in the formal specification of the game. [Ritzberger 2002]

Since the rules are not specific enough to analyze individual decision making, cooperative game theory looks instead at coalitions of players. These coalitions assume that players can commit to work together through binding agreements or through the transfer of decision-making power. [Ritzberger 2002]

Overall, cooperative game theory provides a framework for understanding how different parties can work together toward common goals. [Ritzberger 2002]

Unlike the other branch, non-cooperative game theory requires an exact specification of the rules of the game. For this reason, it is also known as the theory of games with complete rules. It allows an analysis of the individual decisions of players without relying on commitment devices. Players are assumed to act solely out of self-interest, i.e. to choose actions that maximize their payoff. [Ritzberger 2002]

However, a third category can be identified: In a collaborative game, all players work together as a team, sharing the outcomes and thus payoffs. A team is defined by a group of players who have the same interests, albeit the individual information players have may differ. Since the rewards and penalties of their actions are shared, the challenge in a collaborative game is working together to maximize the team's payoff. In contrast, cooperation among individuals may involve different payoffs and goals of the different players. [Zagal, Rick, and Hsi 2006, Marschak and Radner 1972]

II - B 3: Extensive Form

In game theory, games can be modeled in different forms. These forms provide a formal representation of the arbitrary rules of a game. The games studied in this thesis are sequential, of perfect information and may involve multiple players. This makes the extensive form an appropriate choice. It can be seen as a kind of flowchart that shows what can happen during the course of playing the game. [Ritzberger 2002]

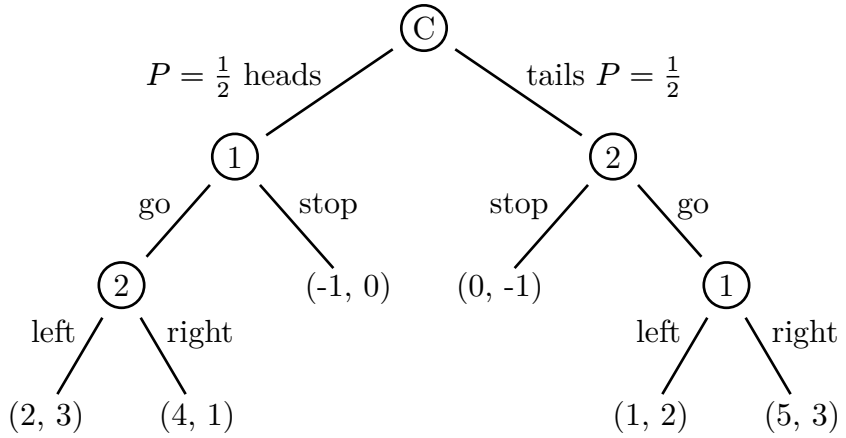

Figure 2: Extensive form example of an contrived game with two players and a chance event

The extensive form looks similar to a regular tree in computer science. It is accordingly also known as a game tree. Each node represents a decision point for a single player, with the game starting at the root node. Outgoing edges from a node are labeled with actions the player can choose. The leaf nodes are the outcomes of the game and specify the payoff vectors. In summary, it can be stated that the extensive form enumerates all possible histories of the game in a tree-like form. [Ritzberger 2002]

An example of the extensive form of an contrived game is given in Figure 2. The root node denotes the start of the game, in the example it is labeled with C. In this case the node is meant to represent a chance event, specifically a coin flip. Therefore, it has two possible outcomes with equal probabilities of $\frac{1}{2}$ each, and the edges are labeled with *heads* and *tails* respectively.

The game continues with either player 1 or 2, depending on the chance event. The coin flip therefore determines the starting player. The first player to move has two available actions, *stop* and *go*. The former ends the game immediatly with a reward of -1 for the player which chose *stop*, and zero for the other one. If the game continues, the other player is given a choice to go left or right, after which the game ends with the payoffs specified in the terminal nodes.

The extensive form also allows an visual explanation of subgames. The extensive form of a subgame is a subset of the original game's game tree. In the example of Figure 2, if a game were to start at any of the nodes labeled with 1 or 2 and shares all nodes below, this is a subgame of the original game. [Ritzberger 2002]

For example, Figure 3 shows two particular subgames from the game in Figure 2:

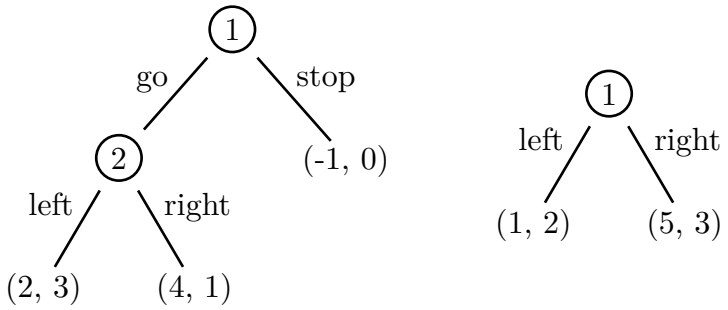

Figure 3: Some subgames of the game in Figure 2

II - B 4: Solutions, Equilibria and Optimal Strategies

A goal of game theory is to assign solutions to, or "solve" games. A solution is a set of strategies, one for each player, that leads to optimal play. The optimality is defined according to some condition or assumption. A game can be said to be solved when a solution can be obtained with reasonable resources, such as computing time and memory. [Ritzberger 2002, Allis 1994]

In non-cooperative game theory, the goal of an optimal solution is always to maximizing the player's payoff. [Ritzberger 2002]

II - B 4.1: Backwards Induction

In the case of perfect information games with sequential moves, an optimal solution can be computed with a simple algorithm, called backwards induction.

II - B 4.1.1: Single Player

Backwards induction is best introduced with a single player game, as it makes the game analysis straightforward. Consider for example this game in extensive form, as shown in Figure 4:

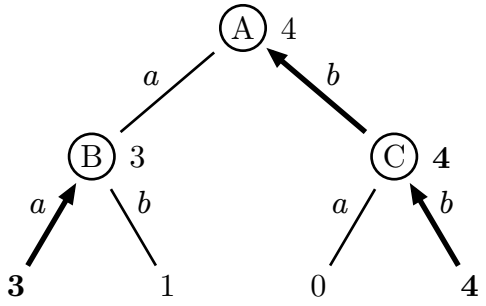

Figure 4: Backwards induction in a simple single player game

The optimal strategy can now be computed bottom-up, starting at the leaf nodes: If the player were already at decision point B, he would certainly choose action *a* since this results in the bigger payoff of 3 over all alternatives. Therefore, node B can be

assigned an utility of 3. Likewise, the utility of decision point C can be determined to be 4, as the player would always choose *b*. Now that the utilities of B and C are found, the optimal decision at A can be identified to be *b* with the same reasoning. Since A is already the root node, the optimal strategy is thus {*b b*}. [Ritzberger 2002]

II - B 4.1.2: Multiplayer

In a multiplayer setting, the strategies of other players influence the course of the game and thus the utility of states. Since it is not immediately apparent what other players will do, it raises the question of whether an optimal strategy can even be determined. However, game theory can answer this question in the affirmative. By introducing the concept of rationality, one can make robust assumptions about the behavior of the players and the strategies they will select. [Ritzberger 2002]

The concept of rationality is built on the premise that players aim to maximize their individual payoffs. Importantly, rationality also includes the understanding that each player is aware that others will act on this premise. Consequently, a player can adjust his strategy accordingly. However, all other players can also infer this change in strategy and adjust their own strategies accordingly… ad infinitum. [Ritzberger 2002]

It can be shown that this reasoning converges to a solution. The constraint that all players try to maximize their payoff has an important implication: No player can improve by choosing a different strategy, as long as he expects all other players to adhere to the solution. The solution is therefore self-enforcing and no player has incentive to deviate. Such self-enforcing strategy combinations are known as Nash equilibria. [Ritzberger 2002]

Such a solution for multiplayer games can be computed as well with backwards induction. Figure 5 visualizes the process for an example game with three players, each having one decision point. The nodes are now labeled with the number of the player at turn, so there are multiple nodes with the same number.

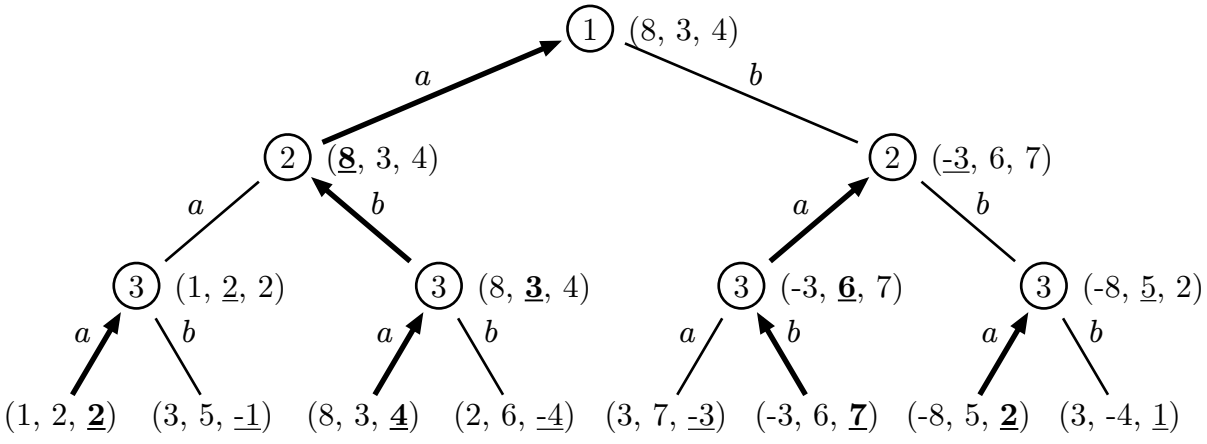

Figure 5: Backwards induction in a multiplayer game with three moves

The tree looks a bit more complicated since a multiplayer game involves payoff vectors instead of single scalar payoffs. However, the reasoning is exactly the same as in the single player scenario: Starting at the last decision points in the tree, player 3 can decide which action to take. He is only interested in maximizing his payoff, so he only looks a the third entry of the payoff vectors. In Figure 5, this is illustrated by underlining the respective entry in the tuple of payoffs. In the specific case of the leftmost decision point in the tree, player 3 has the possible outcomes 2 and -1. As the higher payoff is 2, he will always choose action a. Thus, the leftmost node can be assigned the payoff vector (1, 2, 2) since that is how the game will end from this decision point onwards.

Similarly, the other nodes of player 3 can be processed and utilities assigned that maximize player 3′s payoff. Next, we can move one step upwards in the tree, looking at player 2′s decision. He is only interested in the payoffs relevant to him, which are in the second entry of the payoff vectors. Again, this is illustrated in Figure 5 with underlining the corresponding entries. Consider for instance the left node labeled with 2: The player will choose action b, since that gives him the higher payoff of 3. In the extensive form, this means the utility (8, 3, 4) of the respective child node can be propagated upwards and assigned to player 2′s decision node.

Analogously, player 1 reasons that a is his best choice. Overall, three rational players will choose the respective actions a, b, a.

II - B 4.1.3: Chance Events

If the game involves chance events, chance nodes may be replaced by their expected outcome [Ritzberger 2002]. For example, consider the chance node depicted in Figure 6 with the two outcomes (1, -2, 4) and (-3, 0, 1) and probabilities (0.4, 0.6) respectively. The expected payoff of (-1.4, -0.8, 2.2) is calculated by weighting the possible payoffs by their probabilities.

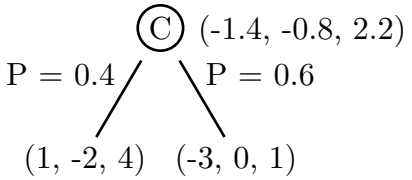

Figure 6: Expected payoff of a chance node

II - B 4.1.4: Zero-Sum Games

For two-player games with zero-sum payoffs, practical implementations of backwards induction may keep track of only a single payoff scalar in the game tree, for example the payoff of the first player. The payoff for the other player is implicitly given by the zero-sum property. Such an implementation navigates the game tree by maximizing the payoff for moves of the first player and minimizing the payoff for moves the other player. This implementation is known as minimax search. [Knuth and Moore 1975]

Instead of selecting the minimum payoff for the other player, the payoff scalar can also be negated for the moves of the other player. In this case, the algorithm can handle all moves in the same manner by maximizing the payoff. This variant is called negamax. [Knuth and Moore 1975]

II - B 4.2: Subgame Perfection

An interesting property of the solutions visualized in Figure 4 and Figure 5 is that they also contain optimal actions for game state which are not part of the overall optimal strategy. For example, in Figure 5, the right node of player 2 is not part of the optimal play. However, if player 2 would ever find himself in this game state (maybe through a mishap of player 1), he knows that his best option is a.

If an optimal solution to a game also contains optimal solutions for all its subgames, the solution is said to be subgame perfect. In the case of backwards induction, the computed solution is always subgame perfect. [Ritzberger 2002]

Subgame perfection is an desireable property of a solution, since it allows players to react to other player's deviations from the optimal strategy. Compared to all players adhering to their optimal strategies, this means:

- in a non-cooperative game: exploiting opponent's mistakes, to potentially achieve a higher payoff
- in a collaborative game: compensating for mistakes of teammates

II - C: Monte Carlo Tree Search

Monte Carlo tree search (MCTS) is a stochastic algorithm that can be applied to sequential decision problems to discover good actions. It takes random samples in the decision space and collects the outcomes in a search tree that grows iteratively. The tree thus contains statistical evidence of available action choices. [Browne et al. 2012, Świechowski et al. 2022]

Monte Carlo tree search is attractive because of its properties: First, it can handle large state spaces due to the random subsampling. This is essential for games or problems where the decision space cannot be fully searched. Second, MCTS is an anytime algorithm: It can be interrupted at any point and returns the best solution found so far. This is important for e.g. games where only a limited time budget is available for move decisions. Likewise, allowing more computing time generally leads to better solutions. Lastly, MCTS can be utilized with little or no domain knowledge. [Browne et al. 2012, Świechowski et al. 2022]

Monte Carlo tree search grows a tree[3] in an asymmetric fashion, expanding it by one node in each iteration. Each node in the tree represents a game state s and stores a visit count n_s, that indicates in how many iterations the node was visited. A node also keeps track of its mean value v_s as approximated by the Monte Carlo simulations. In the basic variant, each iteration of the algorithm consists of four phases, as illustrated in Figure 7.

[3]typically a game tree

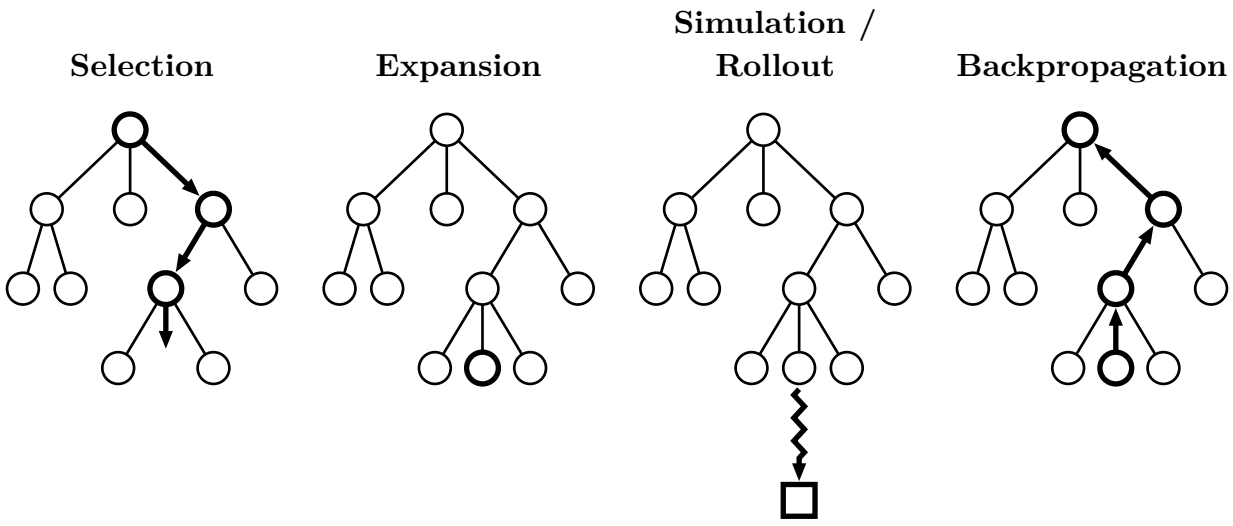

Figure 7: The four phases of one iteration of Monte Carlo tree search

The phases are executed in the following order, unless noted otherwise:

II - C 1: Selection
The algorithm descends the current tree and finds the most urgent location. Selection always begins at the root node and, at each level, selects the next node based on an action determined by a tree policy. The tree policy aims to strike a balance between exploration (focus areas that are not sampled well yet) and exploitation (focus promising areas).
This phase terminates in two conditions:

- a terminal state of the game is reached. In this case, the algorithm skips to Simulation / Rollout.
- the node corresponding to the next selected action is not contained in the tree yet.

[Browne et al. 2012, Świechowski et al. 2022]

II - C 2: Expansion
Adds a new child node to the tree at the position determined by the Selection phase. [Browne et al. 2012, Świechowski et al. 2022]

II - C 3: Simulation / Rollout
The goal of this phase is to determine a value sample of the last selected node. If the node is already a terminal state, the outcome z of the game can be used directly and the algorithm skips to Backpropagation.
In most cases however, the node is somewhere "in the middle" of the game. The algorithm then takes actions at random until a terminal state is reached. Subsequently the outcome z of this terminal state is used to proceed in the Backpropagation phase. [Browne et al. 2012, Świechowski et al. 2022]

How the game's outcome is translated into a node value may depend on the specific game and MCTS implementation. In the case of a single player game, the value may equal the single payoff scalar [Schrittwieser et al. 2020]. The MCTS implementation

may also perform a stochastic equivalent to backwards induction, so in a multiplayer game, a payoff vector with multiple entries may be required [Petosa and Balch 2019]. For two-player zero-sum games, a single payoff scalar may suffice, following the idea of minimax / negamax search [Silver, T. Hubert, et al. 2018, Knuth and Moore 1975].

Intermediate game states visited during this phase are not stored or evaluated in any way. The algorithm's "Monte Carlo" property stems from the random choice of actions during this phase. An optimisation over random actions may be to sample actions from another policy, also referred to as the rollout policy. [Browne et al. 2012, Świechowski et al. 2022, Silver, A. Huang, et al. 2016]

II - C 4: Backpropagation

Propagates the value z obtained in the Simulation / Rollout upwards in the tree, updating the statistics of all ancestor nodes. For each node on the path to the root, the visit count n_s is incremented and the value statistics of the node is updated to include z. [Browne et al. 2012, Świechowski et al. 2022]

For example, a pratical MCTS implementation may be interested in the average value v_s of all simulations that passed through the state s. It can store the sum of all simulation values u_s, and divide by the visit count n_s to calculate the average node value v_s

$$v_s = \begin{cases} \frac{u_s}{n_s} \text{ if } n_s \neq 0 \\ 0 \quad \text{else} \end{cases} \tag{2}$$

During the Backpropagation phase, the following updates would then be performed to incorporate a simulation result z in the statistics:

$$\begin{aligned} n_s &:= n_s + 1 \\ u_s &:= u_s + z \end{aligned} \tag{3}$$

[Schrittwieser et al. 2020, Silver, A. Huang, et al. 2016]

II - D: MuZero and Precursors

MuZero is a state-of-the-art deep reinforcement learning algorithm that builds on previous generations of closely related reinforcement learning algorithms. By understanding its precursors, we can gain valuable insight into MuZero's design and how it evolved from traditional game playing programs.

This section explores and explains the predecessors, they are listed in ascending order of publication date. They build on each other and eventually lead to MuZero, which is explained in detail in Section II - D 5.

Since the first two algorithms, AlphaGo and AlphaGo Zero, focused on the game of Go, I start with a overview of previous attempts at Computer Go.

II - D 1: Computer Go

Go is a two-player, zero-sum board game where players place their (black and white respectively) stones on a 19x19 grid. Go has long been known as a very difficult game for computers to play. The reason is the high complexity in the game tree: A complete game tree would be very large in both height (Go games can span hundreds of moves) and breadth (e.g. the empty board provides $19^2 = 361$ possible locations for placing a stone). This makes an exhaustive search computationally intractable. [Müller 2002, Allis 1994]

Go programs preceeding AlphaGo therefore use Monte Carlo tree search to subsample states in the game tree. A search policy is used to prioritize promising moves, reducing the effective breadth of the search tree. To estimate values of nodes of in the tree, complete rollouts are simulated until the end of the game. To obtain robust value estimations, many rollouts are needed. The policy for selecting actions during rollouts is crucial for the determined values and resulting performance. [Silver and Tesauro 2009, S.-C. Huang, Coulom, and Lin 2011, Baudiš and Gailly 2012, Enzenberger et al. 2010]

Since a high number of rollouts with many simulation steps each are needed, the policy functions are required to be fast. Prior work to AlphaGo therefore uses very simple policies in the search tree and during rollout action selection. These policy functions can be hand-crafted heuristics based on features extracted from the Go board [Gelly, Y. Wang, et al. 2006]. Another possibility is to learn a shallow function based on a linear combination of board features [Coulom 2007, Silver and Tesauro 2009, S.-C. Huang, Coulom, and Lin 2011]. All features are hand-crafted and use domain-specific knowledge of the game. [Gelly, Y. Wang, et al. 2006, Coulom 2007, Baudiš and Gailly 2012, Enzenberger et al. 2010].

However, evaluating board positions via rollouts has been shown to be often inaccurate [S.-C. Huang and Müller 2014]. There have been efforts to include a value function that directly estimates the value of a position without any rollouts [Gelly and Silver 2007]. But again, the value function used by Gelly and Silver [2007] is simple and based on a linear combination of hand-crafted features. Hence, its accuracy is limited and complete rollouts are still required to achieve good performance [Gelly and Silver 2007].

The performance of these approaches is limited, reaching only strong amateur level play [Enzenberger et al. 2010, Baudiš and Gailly 2012, Coulom 2007].

II - D 2: AlphaGo

AlphaGo by Silver, A. Huang, et al. [2016] is a novel and successful approach to the game of Go with the full 19x19 board size. Like prior work, it is based on Monte Carlo tree search, enhanced with a policy and value function. Unlike previous approaches, AlphaGo uses deep neural networks for these functions. Deep neural networks can give much more accurate approximations than previously used heuristics or shallow functions.

Specifically, all networks in AlphaGo are deep convolutional neural networks (CNN) that operate direcly on a 19x19 images of the Go board. The input to all networks is a simple representation of the current board: Several layers of images encode the positions of stones and hand-crafted features on the board in the current as well as past moves.

AlphaGo uses multiple neural networks for the Monte Carlo tree search. They are trained with a multi-stage pipeline that includes supervised and reinforcement learning. Figure 8 shows an overview of the training process and the neural networks used in AlphaGo. Some of the neural networks are only used to generate training data for other networks.

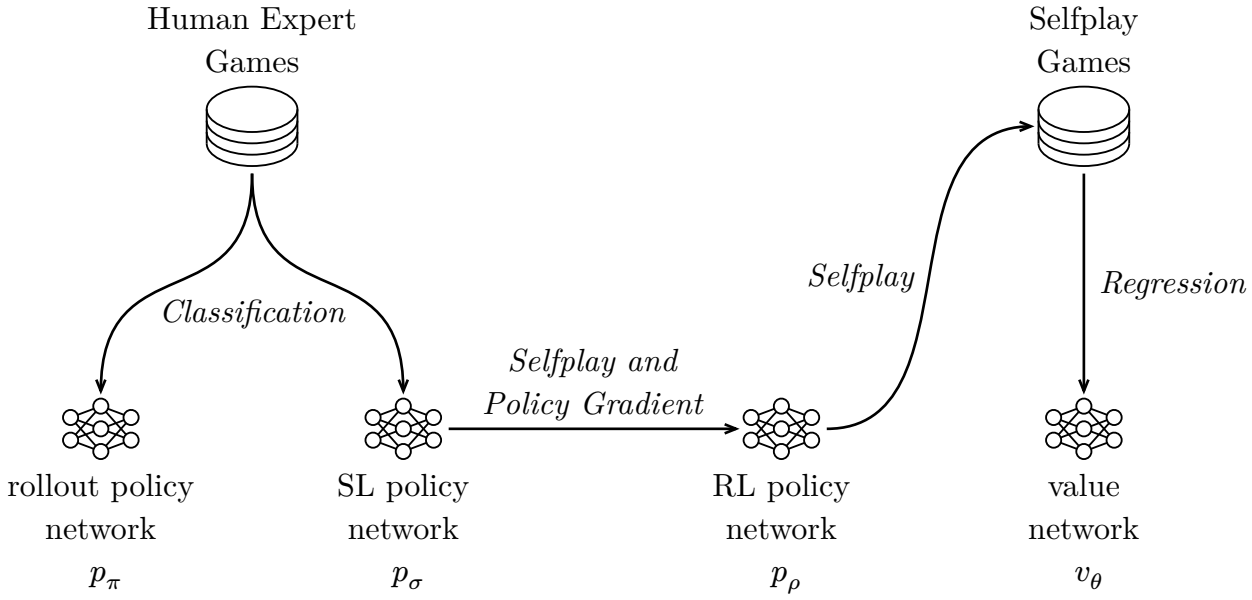

Figure 8: The AlphaGo training pipeline

The pipeline starts with supervised learning (SL) on the KGS Go dataset, which contains Go games played by human experts. A neural network trained on this data predicts which moves humans would play in a given board situation. This is actually not a novelty on its own, since previously CNNs have already been used for this task [Sutskever and Nair 2008, Chris J. Maddison et al. 2014, Christopher Clark and Amos Storkey 2014]. However, the use of a larger convolutional neural network allowed Silver, A. Huang, et al. [2016] to reach a higher accuracy than previous attempts.

Two neural networks are trained on the KGS Go dataset, a big SL policy network p_σ and a smaller one, the fast rollout policy network p_π. The rollout policy network p_π is less accurate, but an order of magnitude faster to evaluate than the SL policy network p_σ.

The next stage in the training pipeline uses reinforcement learning (RL) and selfplay to improve the SL policy network p_σ. The autors call the resulting network RL policy network p_ρ, it can be seen as a fine-tuned version of the SL policy network p_σ. The

idea is to train the network towards the relevant goal of winning games, which does not necessarily align with predicting expert moves perfectly [Silver and Tesauro 2009].

First, the RL policy network p_ρ is initialized to the same structure and weights of the SL policy network p_σ. The RL policy network p_ρ is then trained with policy gradient [Sutton, McAllester, et al. 1999, Williams 1992] to win more games against previous versions of the RL policy network p_ρ. For this, games are played between two agents which select actions sampled from predictions of the RL policy network p_ρ. One agent uses the current version of the RL policy network p_ρ, the other one a random older iteration. The authors argue that randomizing from a pool of opponents prevents overfitting and stabilizes training.

The selfplay in AlphaGo does not use any search. The final iteration of the RL policy network p_ρ already plays Go better (without search) than the strongest available open-source Go program.

The last stage trains a value network v_θ that evaluates positions directly. It is trained in supervised manner to predict the game outcome from the current board position, assuming strong players. A suitable dataset for this task requires a large number of Go games and strong play of both players. The authors used the RL policy network p_ρ and selfplay to generate a new dataset that fulfills these requirements.

Specifically, each selfplay game is carried out in three phases: First, a random number U is sampled uniformly $U \sim \text{unif}\{1, 450\}$. Then, the moves at time steps $t = 1, ..., U-1$ are sampled from predictions of the SL policy network p_σ, $a_t \sim p_\sigma(\cdot\,|s_t)$. Second, a single move a_U is sampled from the legal moves. Lastly, the RL policy network p_ρ is used to generate the remaining moves $t = U+1, ..., T$ until the game terminates, $a_t \sim p_\rho(\cdot\,|s_t)$. The game is then scored to determine its outcome z_T. From every game, only a single training example (s_{U+1}, z_T) is added to the selfplay dataset. The final dataset contains positions from 30 million distinct games.

Finally, three networks (the two prediction networks p_σ, p_π and the value network v_θ) are combined in a variant of Monte Carlo tree search (MCTS).

In the MCTS selection phase, predictions from the SL policy network p_σ are used. The SL policy network p_σ was found to perform better in this job than the RL policy network p_ρ. The predictions of the SL policy network p_σ are combined with the values already present in the tree to balance guidance from the network predictions, exploitation and exploration.

Specifically, in the search tree, each node corresponds to a game state s, and the outgoing edges represent legal actions. Each edge stores an action value $Q(s,a)$, the visit count $N(s,a)$ and a prior probability $P(s,a)$ derived from predictions from the SL policy network p_σ. In each time step t of the MCTS selection phase, the child node corresponding to the action a_t is selected from the state s_t

$$a_t = \underset{a}{\text{argmax}}(Q(s_t,a) + u(s_t,a)) \tag{4}$$

to maximize the action value plus a bonus.

The bonus term $u(s,a)$ is initially proportional to the prior probability but decays with repeated visits to encourage exploration:

$$u(s,a) = cP(s,a)\frac{\sqrt{\sum_{b\in A} N(s,b)}}{1+N(s,a)} \tag{5}$$

The influence of the bonus can be adjusted with the constant c.

The selection strategy maximizes over a probabilistic upper confidence tree [Rosin 2011]. Such formulas (and variants) are therefore also referred to as "pUCT formulas". [Silver, A. Huang, et al. 2016, Schrittwieser et al. 2020, Antonoglou et al. 2022]

The MCTS selection phase traverses the tree as usual, until time step L, where it reaches a leaf node that may be expanded (if it is not a terminal state). During expansion, the new node corresponding to state s_L is processed by the SL policy network p_σ to yield the prior probabilities $P(s_L,a)$:

$$P(s_L,a) = \begin{cases} mp_\sigma(a|s_L) & \text{if } a \in A(s_L) \\ 0 & \text{if } a \notin A(s_L) \end{cases}$$

$$\text{with } m = \frac{1}{\sum_{b\in A(s_L)} p_\sigma(b|s_L)}$$

where $A(s_L)$ denotes the set of legal actions in state s_L, as given by the game simulator. The prior for illegal actions is masked to zero, and the remaining probabilities are rescaled to sum to one: $\sum_{b\in A(s_L)} P(s_L,a) = 1$.

During the MCTS simulation phase, the new node is evaluated using a combination of Monte Carlo rollouts and the value network v_θ. Specifically, for the state s_L, a rollout until game end is performed using the rollout policy network p_π to yield a value z_L. The rollout value z_L is blended with the prediction from the value network, $v_\theta(s_L)$, to the overall value $V(s_L)$, using a mixing factor λ:

$$V(s_L) = (1-\lambda)v_\theta(s_L) + \lambda z_L$$

The algorithm performed best with a mixing factor $\lambda = 0.5$, that is, equal weighting of the rollouts and value network. However, even without any rollouts at all ($\lambda = 0$), AlphaGo performed better than previous computer Go programs.

One MCTS iteration is concluded by backpropagating $V(s_L)$ up in the tree. All edges on the search path are updated so that $Q(s,a)$ represents the average value of all simulations that passed through it, like described in Section II - C 4.

Each node s in the search tree is only processed once by the neural networks, when the node is expanded. The resulting value $v(s) = v_\theta(s)$ and policy $p(s) = p_\sigma(s)$ (represented by the edge probabilities $P(s,a)$) are stored in the tree and reused for the

next MCTS iterations. In other words, per iteration, the tree grows by one node (as usual), and network inference is performed on the new node only.

To decide on a move to play in state s_p, a search tree is initialized with the root node corresponding to s_p. A number of MCTS iterations is performed, and the action a_p with the highest visit count is played:

$$a_p = \underset{a}{\operatorname{argmax}}\big(N\big(s_p, a\big)\big)$$

In the case where the "thinking time" for each move is limited, the maximum number of iterations depends on the speed of the algorithm. To achieve more iterations in a give time budget, the authors also implemented a distributed version of AlphaGo. It utilizes multiple machines with a total of 1202 CPUs and 176 GPUs to parallelize the search. This distributed version was able to beat a professional human player in 5 out of 5 games.

II - D 3: AlphaGo Zero

AlphaGo Zero by Silver, Schrittwieser, et al. [2017] is similar to AlphaGo: It also uses MCTS with neural networks to play Go at super-human levels. However, the authors improved over AlphaGo by simplifing the architecture in several aspects, while also improving the playing strength.

The main changes over AlphaGo are:

Network architecture

AlphaGo Zero only uses a single CNN f_θ with two output heads, predicting policy p and value v together: $(p, v) = f_\theta(s)$. The input to this unified network are images consisting of the Go stone positions, no extra feature planes are added.

Contrastingly, the equivalent to f_θ in AlphaGo are the two separate neural networks p_σ and v_θ. Also, input images to AlphaGo's neural networks contain hand-crafted features. The networks p_π and p_ρ and their training procedures are dropped in AlphaGo Zero.

Monte Carlo tree search

The MCTS in AlphaGo Zero works the same way as in AlphaGo, but does not use any rollouts. AlphaGo Zero solely relies on the network value predictions to assign values in the search tree. AlphaGo Zero does not contain a rollout policy network like p_π in AlphaGo.

Training

Most importantly, AlphaGo Zero is trained solely by reinforcement learning and selfplay. AlphaGo Zero uses MCTS for all the selfplay during training.

This contrasts with AlphaGo, which uses human data and supervised learning, as well as selfplay without any search.

The AlphaGo Zero training pipeline performs these tasks in parallel:

- optimizing the neural network's parameters θ_i from recent selfplay data

- evaluating selfplay players α_{θ_i}
- using the the best performing player so far, α_{θ_*} to generate new selfplay data

Specifically, selfplay with the player α_{θ_*} is carried out in the following manner: At each decision point in the game, a Monte Carlo tree search with 1600 iterations is performed. The MCTS executes like in AlphaGo with $\lambda = 0$, that is, without any rollouts. A neural network f_{θ_*} provides policy p_s and value v_s predictions for each state s in the search tree:

$$(p_s, v_s) = f_{\theta_*}(s)$$

where p_s denotes a probability distribution over actions: $p_s(a) = \Pr(a|s)$.

Like in AlphaGo, the policy predictions p_s are masked with the set of legal actions $A(s)$ from the simulator to yield the node prior probabilities $P(s, a)$:

$$P(s,a) = \begin{cases} mp_s(a) & \text{if } a \in A(s) \\ 0 & \text{if } a \notin A(s) \end{cases}$$

$$\text{with } m = \frac{1}{\sum_{b \in A(s)} p_s(b)}$$

Like in AlphaGo, the policy predictions p serve as the tree policy and guide the search, as outlined in Equation 4 and Equation 5. The value predictions v are used to determine the value $z = v_L$ of a leaf node s_L prior to MCTS backpropagation, which is introduced in Section II - C 4. The value predictions fully replace the MCTS simulation / rollout phase.

An distinction to AlphaGo is that the image of the Go board is randomly rotated or flipped before using it as the input for the neural network. This data augmentation step exploits symmetries of the game of Go and aims to reduce prediction bias.

The search outputs probabilities π of playing each possible move, proportional to the visit counts of the root node s in the search tree:

$$\pi(a|s) = \frac{N(s,a)}{\sum\limits_{b \in A} N(s,b)} \tag{6}$$

where A is the set of actions.

To ensure diverse game openings, the first 30 moves are sampled from π, so $a_t \sim \pi_t$ for $t = 0, ..., 29$. The rest of the moves are selected according to the action a_t with the highest visit count $a_t = \underset{a}{\text{argmax}}(N(s_t, a))$. Games are played until an terminal condition is reached, and the outcome of the game is scored as z.

Dirichlet noise is blended into the prior probabilities $P'(r, a)$ of the root node r to encourage exploration:

$$P'(r,a) = (1 - \varepsilon)P(r,a) + \varepsilon\eta_a \forall a \in A(s) \tag{7}$$

where $A(s)$ is the set of legal actions in state s, with $\eta_a \sim \mathrm{Dir}(\alpha)$, $\alpha = 0.03$ and $\varepsilon = 0.25$. Adding exploration this way ensures all moves may be tried, but the search can still overrule bad actions.

The neural network f_{θ_i} is trained on samples (s, π, z) drawn from the latest selfplay games. Random rotations and reflections of the input image to the neural network are used to provide data augmentation. Mean-squared error and cross-entropy is used to align the network predictions $(p, v) = f_{\theta_i}(s)$ with the search probabilities π and game outcome z, respectively. Specifically, the network is optimized using gradient descent on the loss function

$$l = (z - v)^2 - \pi \log p + c\|\theta\|^2$$

where the last term is used for L2 weight normalisation, with the constant c determining its influence.

New versions of the neural network f_{θ_i} are evaluated in a tournament against the current selfplay player α_{θ_*} before they may be used for the selfplay data generation. If the new player α_{θ_i} wins 55% of the games against α_{θ_*}, it becomes the new α_{θ_*}. This ensures the highest quality data is generated for neural network training.

AlphaGo Zero's training process starts out with random selfplay. Over the course of 40 days and 29 million games, it learns Go capabilities that exceed those of humans and its precursor AlphaGo.

II - D 4: AlphaZero

AlphaZero, proposed by Silver, Schrittwieser, et al. [2017] is the result of minor architectural changes to make AlphaGo Zero compatible with two more board games, chess and shogi. It is a step towards a general reinforcement learning algorithm that masters more than one game.

AlphaZero uses the same algorithm and neural network architecture for all three games. The two main changes over AlphaGo Zero are:

No data augmentation

AlphaGo Zero's data augmentation step of rotating and flipping board representations is removed, because it exploits Go-specific symmetries. The data augmentation provided an eightfold increase in training data, AlphaZero therefore trains 8 times slower on the game of Go than its precursor.

Shared network for selfplay and training

AlphaGo Zero maintains separate neural networks for selfplay and training, and switches them over when the new network performs better than the current one. In contrast, AlphaZero only uses a single network. It always uses the latest network parameters for selfplay data generation.

Compared to AlphaGo Zero, there are no distinct selfplay players α_{θ_i}, and consequently no need to evaluate a best performing player α_{θ_i}

AlphaZero uses the same hyperparameters as AlphaGo Zero, except for the number of MCTS simulations and the Dirichlet exploration noise. Specifically, AlphaZero employs only 800 MCTS simulations per move during selfplay, as opposed to 1600 simulations for AlphaGo Zero. The Dirichlet noise (see Equation 7) is adjusted based on the game: AlphaZero uses $\alpha = \{0.3, 0.15, 0.03\}$ for chess, shogi and Go respectively.

The publication shows that the AlphaZero training approach is feasable of generalization by evaluating it on Go, chess and shogi. AlphaZero outperformed existing state-of-the-art programs in all of the game, including its precursor AlphaGo in the case of Go. The training success was also reported to be repeatable across multiple independent runs.

II - D 5: MuZero

MuZero by Schrittwieser et al. [2020] is yet another improvement over AlphaZero in terms of generalization. MuZero learns a dynamic model of the game, solely from interactions, and uses it in the Monte Carlo tree search to plan ahead. In contrast, AlphaZero uses an explicit simulator during the tree search to obtain a game state for a sequence of hypothetical actions. This simulator represents domain knowledge of the environment.

By learning a model instead, the MuZero architecture can generalize to different environments, including single player games and non-zero rewards at intermediate steps. The authors applied MuZero to the games Go, shogi, chess and the complete Atari suite. In the case of board games, it matched the performance of AlphaZero and in the Atari environment it outperformed existing approaches.

II - D 5.1: Dynamics Network

Just like AlphaZero, MuZero uses a network f for predictions inside the search tree. However, MuZero introduces two additional neural networks h and g. The dynamics network g learns state transitions of the environment: Given a state s^t and a hypothetical action a^t, g predicts the next state s^{t+1} and reward r^{t+1} the environment will respond with. The states s^t are encoded in a learnt latent space, so a representation network h translates observations into latent representations.

To differentiate states and actions occuring in an environment trajectory from those used by network inferences, I introduce the following notation: States and rewards occuring in a game are marked with subscript, so the state and action at time t are s_t and a_t, respectively. A superscript is added to denote states predicted by the neural networks, so for example $h(s_t) = s_t^0$. The superscript is increased for each inference step with the dynamics network g, for example $s_t^1 = g(s_t^0, a^0)$. In some cases, the subscript may be omitted to simplify notation.

The dynamics network can be applied recurrently, so any reward r^n and state s^n n time steps ahead can be predicted given an initial state s^0 and a series of hypothetical actions $a^0, a^1, ..., a^{n-1}$:

$$(s^t, r^t) = g(s^{t-1}, a^{t-1})$$

II - D 5.2: Monte Carlo Tree Search

The recurrent inference is used during Monte Carlo tree search, as visualized in Figure 9. To decide on an action in game state s, a search tree is initialized with the game observation: $s^0 = h(s)$. During the MCTS expansion phase, the dynamics network g is used to obtain the next state s^{n+1} and the reward r^{n+1} associated with the state transition for an action a^n in state s^n.

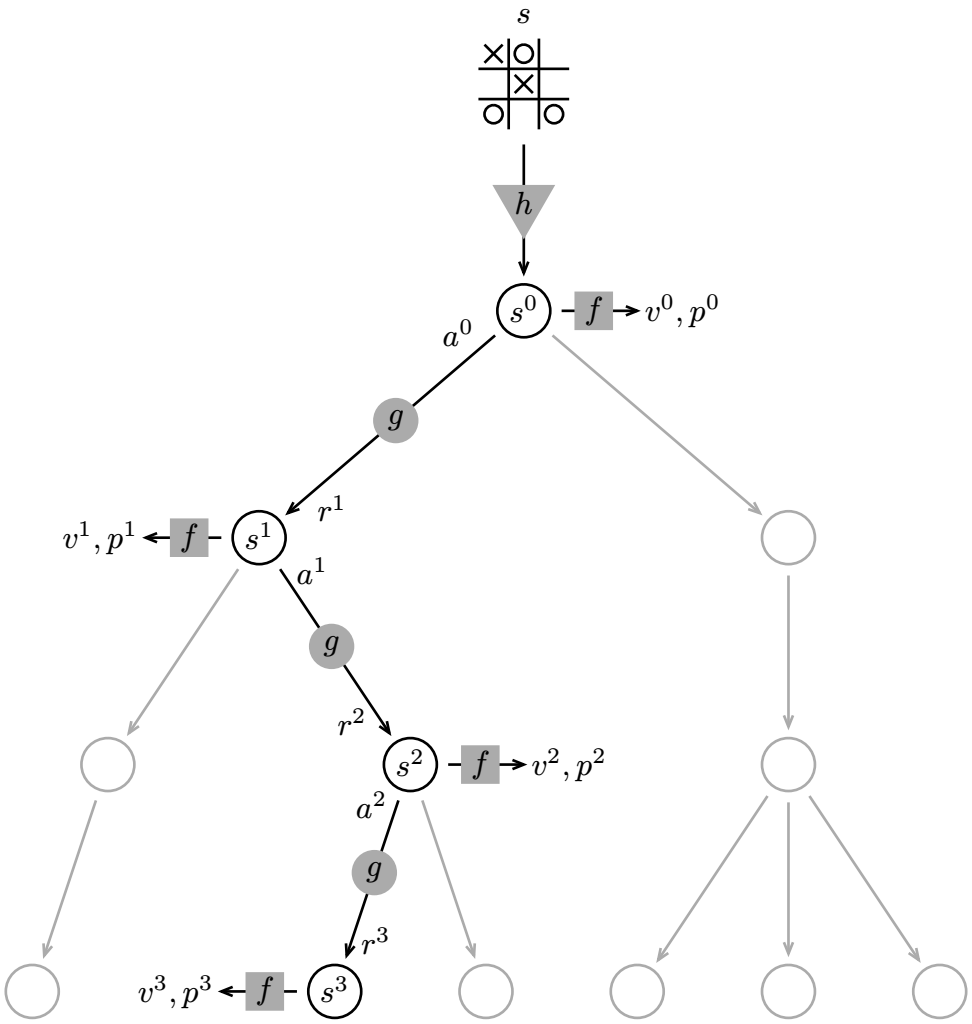

Figure 9: Monte Carlo tree search in MuZero

Nodes are selected according to a pUCT formula similar to Equation 5. Specifically, in state s^k, the child node corresponding to the action a^k is selected

$$a^k = \underset{a \in A}{\operatorname{argmax}} \left\{ Q(s^k, a) + P(s^k, a) \frac{\sqrt{\sum_{b \in A} N(s^k, b)}}{1 + N(s^k, a)} \left[c_1 + \log \left(\frac{\sum_{b \in A} N(s^k, b) + c_2 + 1}{c_2} \right) \right] \right\} \tag{8}$$

where A is the set of possible actions. $N(s,a)$ and $P(s,a)$ denote the visit count and prior probability of the child node corresponding to action a, respectively. The prior probability is a mixture of the network predictions and Dirichlet noise as shown in Equation 7.

The constants c_1 and c_2 are hyperparameters to control the influence of the policy $P(s,a)$ relative to the value $Q(s,a)$ for higher visit counts. Schrittwieser et al. [2020] used $c_1 = 1.25$ and $c_2 = 19652$.

In contrast to AlphaZero, $Q(s,a)$ consists not only of the child node value, it must also include the transition reward:

$$Q\big(s^k, a^k\big) = \gamma v^{k+1} + r^{k+1}$$

where γ is the reinforcement learning discount factor, as introduced in Section II - A 2.

The MCTS simulation phase executes like in AlphaZero, where f predicts a value estimate and search policy of a newly expanded node s^L:

$$(v^L, p^L) = f(s^L)$$

Traditional MCTS assumes no intermediate rewards and backpropagates a single simulation result through the search tree [Browne et al. 2012, Świechowski et al. 2022]. However, the backpropagation in MuZero needs to take intermediate rewards into account[4]. The value to backpropagate is updated for every parent node on the search path.

Specifically, let the search path consist of the nodes $s^0, s^1, ..., s^L$, where s^L is the newly expanded leaf node. For $k = L...0$ an n-step return is calculated, bootstrapped from the leaf node value v^L:

$$G^k = \gamma^{L-k} v^L + \sum_{i=1}^{L-k} \gamma^{i-1} r_{k+i}$$

This is equivalent to the calculation of the discounted n-step return in reinforcement learning, see Equation 1.

The statistics for each node s^k for $k = L-1...0$ are updated with the respektive G^{k+1}:

$$Q\big(s^k, a^k\big) := \frac{N\big(s^k, a^k\big) Q\big(s^k, a^k\big) + G^{k+1}}{N\big(s^k, a^k\big) + 1}$$

$$N\big(s^k, a^k\big) := N\big(s^k, a^k\big) + 1$$

This is equivalent to the formulas for a pratical implementation keeping track of average node values, as shown in Equation 2 and Equation 3.

[4] in the general case. When applied to board games without intermediate rewards, the reward predictions are ignored and MCTS backpropagation is performed as in AlphaZero.

II - D 5.3: Training

The exact training procedure differs slightly depending on the type of game. I explain the training setup for two-player board games first, as it is most similar to MuZero's precursors.

II - D 5.3.1: Board Games

Like AlphaZero, MuZero uses selfplay with MCTS to generate training data. For each selfplay step at time t the data (s_t, a_t, π_t, z) is recorded. The first two entries contain the state of the environment and executed action respectively. π_t denotes the search policy, as derived from the root node according to Equation 6. z is the final outcome of the game, indicating a win, loss or draw.

For training, the dynamics network g is unrolled for K steps and aligned with a sequence of environment states and actions from a selfplay trajectory. Specifically, a training example beginning at time step t consists of the tuple $(s_t, (a_t, a_{t+1}, ..., a_{K-1}), (\pi_t, \pi_{t+1}, ..., \pi_K), z)$, where K is the unroll length, a hyperparameter.

The process is illustrated in Figure 10 for $K = 2$:

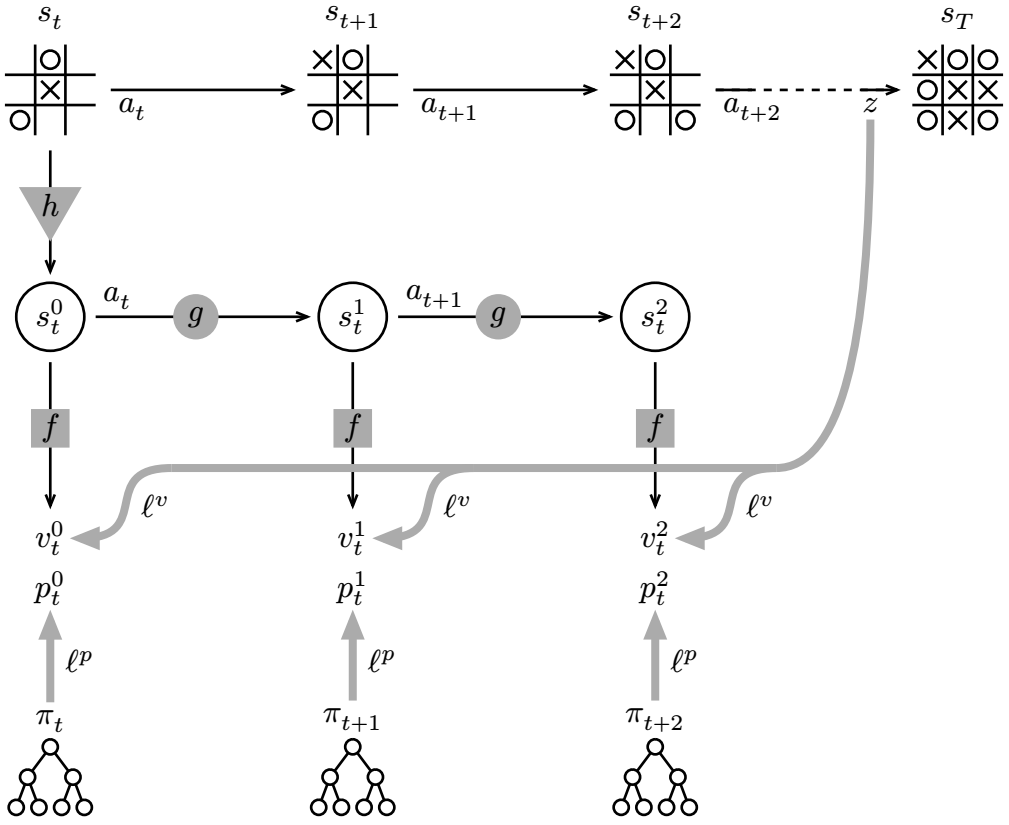

Figure 10: Training setup in MuZero in the case of board games, unrolled for 2 steps. Thick arrows indicate training losses.

First, a latent representation s_t^0 is obtained using the representation network: $s_t^0 = h(s_t)$. Then, the dynamics network g is applied recurrently $K - 1$ times with

the actions from the trajectory $a_t, a_{t+1}, ..., a_{K-1}$. This yields latent representations for the states $s_t^1, s_t^2, ..., s_t^K$. The reward predictions are ignored since board games have no intermediate rewards. At each unroll step $n = 0, 1, ..., K$, the prediction network f predicts a value and policy (v^n, p^n) from the latent representation s_t^n.

Losses for the policy and value predictions are calculated at each unroll step to align the predictions of f with the search policies π_t and game outcome z. Specifically, the total loss ℓ_t for an unroll sequence starting at s_t with length K is given by

$$\ell_t = \sum_{k=0}^{K} \ell^p\big(p_t^k, \pi_{t+k}\big) + \sum_{k=0}^{K} \ell^v\big(v_t^k, z\big)$$

with ℓ^p and ℓ^v being loss functions for policy and value, respectively.

MuZero always uses the cross-entropy loss for ℓ^p. The loss ℓ^v in the case of board games is the mean squared error.

The parameters of the three neural networks are updated jointly via backpropagation on the total loss ℓ, in an end-to-end manner. Performing backpropagation repeatedly through the same network (g) is also known as backpropagation through time. To keep the gradients stable during backpropagation of the unrolled dynamics network, Schrittwieser et al. [2020] rescale the gradients of the latent representations s^n at the input of the dynamics network by 0.5.

II - D 5.3.2: Environments with Intermediate Rewards

When applied to environments with intermediate rewards, such as the Atari suite, reward predictions are included in the training.

The data from a single time step t of selfplay is extended with the reward experienced: $(s_t, a_t, r_{t+1}, \pi_t, G_t)$. The last item of the tuple is also updated: It now contains the n-step return G_t (see Equation 1) instead of the final outcome of the game z. This n-step return is bootstrapped with the value of the MCTS root node.

Likewise, a training sample includes the sequence of sample returns and $K-1$ experienced rewards:

$$(s_t, (a_t, a_{t+1}, ..., a_{K-1}), (r_{t+1}, r_{t+2}, ..., r_K), (\pi_t, \pi_{t+1}, ..., \pi_K), (G_{t+1}, G_{t+2}, ..., G_K))$$

The dynamics network is unrolled in the same manner, and an additional reward loss ℓ^r is introduced. The total loss thus becomes

$$\ell_t = \sum_{k=0}^{K} \ell^p\big(p_t^k, \pi_{t+k}\big) + \sum_{k=0}^{K} \ell^v\big(v_t^k, G_{t+k}\big) + \sum_{k=1}^{K} \ell^r\big(r_t^k, r_{t+k}\big)$$

Figure 11 illustrated the updated training setup:

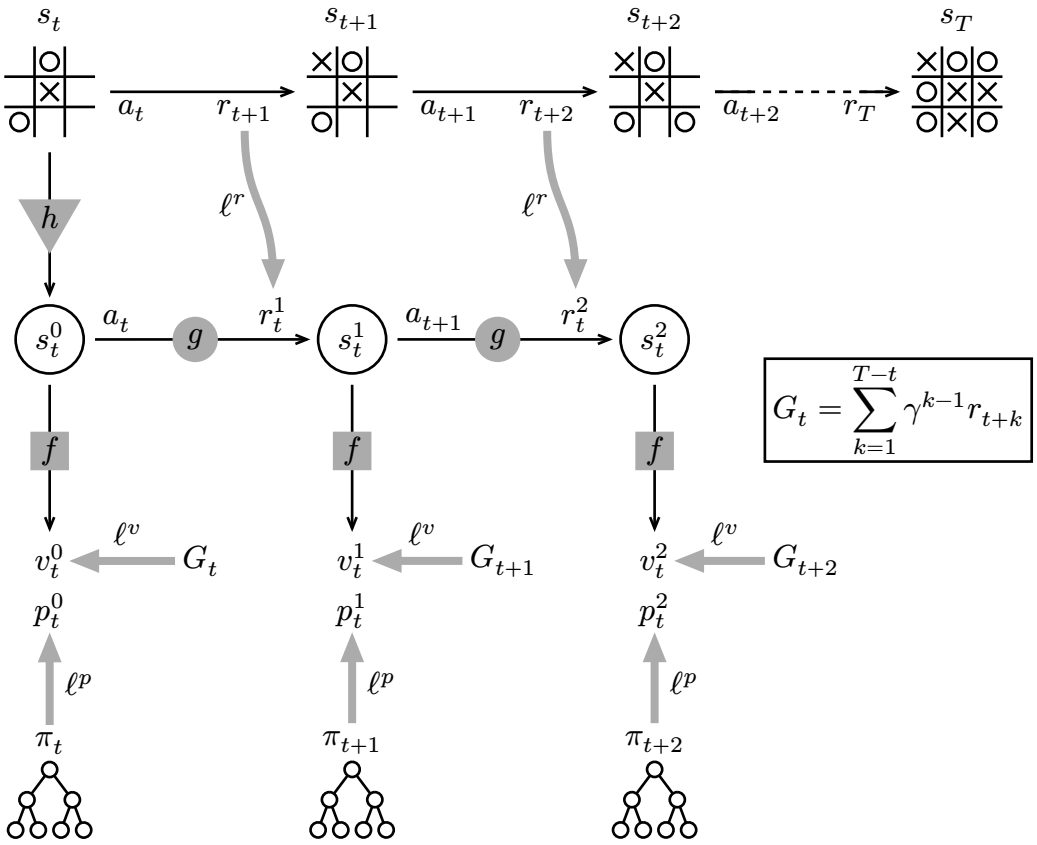

Figure 11: Training setup in MuZero for Atari games, unrolled for 2 steps. Thick arrows indicate training losses.

For value and reward predictions in Atari games, MuZero uses the following setup: First, the targets G and r are scaled using the invertible transform $e(x)$:

$$e(x) = \operatorname{sign}(x)\Big(\sqrt{|x|+1} - 1\Big) + \varepsilon x$$

where $\varepsilon = 0.001$.

Subsequently, a second transformation $\varphi(x)$ is applied to the transformed targets $e(G)$ and $e(r)$, generating equivalent categorical representations over a discrete set F of numbers, referred to as the support. Under the transformation $\varphi(x)$, each scalar is represented as the linear combination of its two adjacent supports $x_{\text{low}}, x_{\text{high}} \in F$:

$$x = p_{\text{low}} x_{\text{low}} + p_{\text{high}} x_{\text{high}}$$

where p_{low} and p_{high} denote the weights of the supports x_{low} and x_{high}, respectively. Additionally $p_{\text{low}} + p_{\text{high}} = 1$ is required.

As an example, consider the support set $F = \mathbb{Z}$: The target $x = 3.7$ would be represented by $p_{\text{low}} = 0.7, p_{\text{high}} = 0.3, x_{\text{low}} = 3, x_{\text{high}} = 4$.

The value and reward outputs of the neural networks are also modelled using a softmax output of size $|F|$. During training, a cross-entropy loss is used for ℓ^r and ℓ^v to

align the categorical predictions r_t^n and v_t^n to the transformed targets $\varphi(e(r_{t+n}))$ and $\varphi(e(G_{t+n}))$, respectively.

During inference, the actual values and rewards are obtained by inverting the two transforms: First, the expected scalar quantity $\widehat{x'}$ under the softmax distribution over the support F is computed. Subsequently, the inverse e^{-1} of the scaling transformation $e(x)$ is applied to recover the actual value x:

$$x = e^{-1}\left(\widehat{x'}\right)$$

MuZero uses a support size $|F| = 601$ with one support for every integer in the interval $[-300, 300]$.

III: Related Work

Below, I summarize some publications about MuZero and its precursors, which are similar to the work in this thesis.

III - A: Multiplayer AlphaZero

While no multiplayer version for MuZero itself exists, Petosa and Balch [2019] extended its predecessor algorithm AlphaZero to multiplayer capabilities.

The original implementation of AlphaZero heavily relies on the zero-sum property for some architectural simplifications: The Monte Carlo tree search performs a negamax search (see Section II - B 4.1.4 for details), which only uses a scalar for describing the value of nodes [Knuth and Moore 1975]. The neural network subsequently also only predicts scalar values. [Silver, T. Hubert, et al. 2018]

Multiplayer AlphaZero drops the assumption of the game being zero-sum. This makes it necessary to extend the scalar quantities used in the algorithm to vectors.

Specifically, an n player game returns a score vector $\vec{z}$:

$$\vec{z} = [z_1, z_2, ...z_n] \in \mathbb{R}^n$$

Each component z_i denotes the indvidual outcome for player i. Likewise, node values in the search tree and predictions by the neural network are extended to vectors $\vec{v} \in \mathbb{R}^n$.

MCTS backpropagation is performed with the vectors. The computations used to update node statistics are performed with elementwise vector operations. This updates each vector component independently.

Naturally, the Monte Carlo tree search rotates over the players in turn order, as given by the game simulator. When selecting a node's children, the algorithm seeks to maximize component v_i of the value vector $\vec{v}$, where i denotes the player currently at turn. This algorithm is known as max^n search [Browne et al. 2012] and similar to the multiplayer backwards induction introduced in Section II - B 4.1.2.

They evaluate their work on multiplayer versions of Connect 4 and Tic-Tac-Toe: The networks learn to encode knowledge of the game into search, indicating that the proposed multiplayer strategy works in principle. Performance-wise the algorithm places itself below human experts.

In my implementation of MuZero for multiplayer environments, I also use max^n search, as outlined in Section IV - B 3. To obtain per-player individual utilities during the search, I also extend the value and reward predictions to vectors, for details see Section IV - B 1. The player currently at turn for a tree node is predicted by the networks, as discussed in Section IV - B 2.

III - B: Stochastic MuZero

Antonoglou et al. [2022] extend MuZero to stochastic, but observable environments. The original MuZero algorithm is limited to deterministic environments, due to the deterministic predictions of the dynamics network g.

Stochastic MuZero makes use of afterstates when modeling the environment. These afterstates occur after each action of the agent and represent an hypothetical state of the environment before it has transitioned to a true state [Sutton and Barto 2018]. The idea is visualized in Figure 12. In stochastic environments, afterstates therefore can be viewed as a state of uncertainty from which a definitive outcome will emerge. From a game theoretic viewpoint, afterstates may represent decision points of chance events.

Note that in this section, the superscript in s_t^i has a special meaning in contrast to the rest of the thesis: It is used to differentiate distinct stochastic outcomes the environment may transition to, from the same state-action pair.

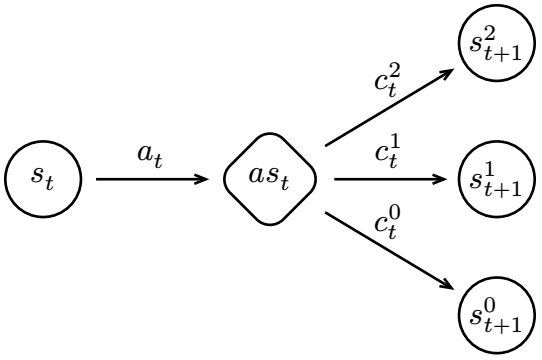

Figure 12: Afterstates in Stochastic MuZero

The transition from the afterstate as_t to the true state s_{t+1}^i is modeled using a chance outcome c_t^i from a finite set of C possible chance outcomes. This allows to use a deterministic model $\mathcal{M}$ for environment transitions: $\mathcal{M}$ receives a state s_t, the action taken a_t and a chance outcome c_t^i:

$$(s_{t+1}, r_{t+1}) = \mathcal{M}(s_t, a_t, c_t^i)$$

This way, the task of learning stochastic transitions can be reduced to learning afterstates and distributions over the chance outcomes.

In pratice, Stochastic MuZero introduces two new neural networks, the afterstate dynamics φ and afterstate prediction ψ. They are comparable to the normal dynamics and prediction network, respectively.

Specifically, the afterstate dynamics φ predicts an afterstate as_t for a given state-action pair s_t, a_t:

$$as_t = \varphi(s_t, a_t)$$

The afterstate prediction ψ outputs a value Q_t for an afterstate as_t and a probability distribution $\sigma_t = \Pr(c_t^i | as_t)$ over the chance outcomes:

$$(Q_t, \sigma_t) = \psi(as_t)$$

The regular dynamics network g takes the role of predicting the transition from an afterstate as_t to a true state s_{k+1} under chance outcome c_t^i:

$$s_{t+1}, r_{t+1} = g(as_t, c_t^i)$$

The inference of environment dynamics from a state-action pair s_t, a_t thus becomes a multi-step process: First, an afterstate as_t is predicted by $\varphi(s_t, a_t)$. Then, the distribution over chance outcomes is obtained: $\sigma_t = \psi(as_t)$. Finally, a chance outcome is sampled $c_t^i \sim \sigma_t$ and the next state s_{t+1} is obtained using $s_{t+1} = g(as_t, c_t^i)$.

Stochastic MuZero employs a variant of a Vector Quantised Variational AutoEncoder (VQ-VAE) [Oord, Oriol Vinyals, and Koray Kavukcuoglu 2017] to learn the chance transitions and their probabilities σ through interactions of the environment. VQ-VAEs have a fixed codebook size, set by a hyperparameter, which limits the number of possible outputs. In Stochastic MuZero, it is set to the maximum number of distinct chance outcomes in the game or higher.

Stochastic MuZero matches the performance of MuZero in deterministic environments. It outperforms previous approaches in stochastic domains, such as the game 2048 and backgammon.

My implementation also handles stochastic environments. Unlike Stochastic MuZero, I learn the occurence of chance events and their outcomes from ground-truth labels given by the game simulator. For more details see Section IV - B 4.

III - C: EfficientZero

EfficientZero by Ye et al. [2021] is a modification of MuZero to achieve similar performance like the original algorithm, but requiring less training data. The amount of training data is usually measured in the number of interactions with the environment, also called samples. To achieve this increase in data- and sample efficiency they propose three changes. The most effective and relevant to this work is the introduction of a latent similarity loss:

They notice that the dynamics network g should map to same latent representation as the representation network h for the same game states. By adding a similarity term to the loss function, these network outputs are encouraged to converge. This provides a richer training signal since the latent representation is usually a very wide tensor.

Specifically, consider two game states s_t and s_{t+n} with the actions $a_t, a_{t+1}, ..., a_{t+n-1}$ in between. A latent representation for s_{t+n} can be reached in two ways:
First, with the representation network h directly: $s_{t+n}^0 = h(s_{t+n})$.
Second, by s_t^n as obtained through $n-1$ inferences with the dynamics network from initially s_t and the action sequence $a_t, a_{t+1}, ..., a_{t+n-1}$:

$$s_t^x = \begin{cases} h(s_t) & \text{if } x = 0 \\ g(s_t^{x-1}, a_{t+x-1}) & \text{else} \end{cases}$$

Note that the reward predictions r_t^x of the dynamics network are omitted for brevity in the formula.

The idea of the additional similarity loss ℓ^l is to match s_t^n to s_{t+n}^0. This is illustrated in Figure 13.

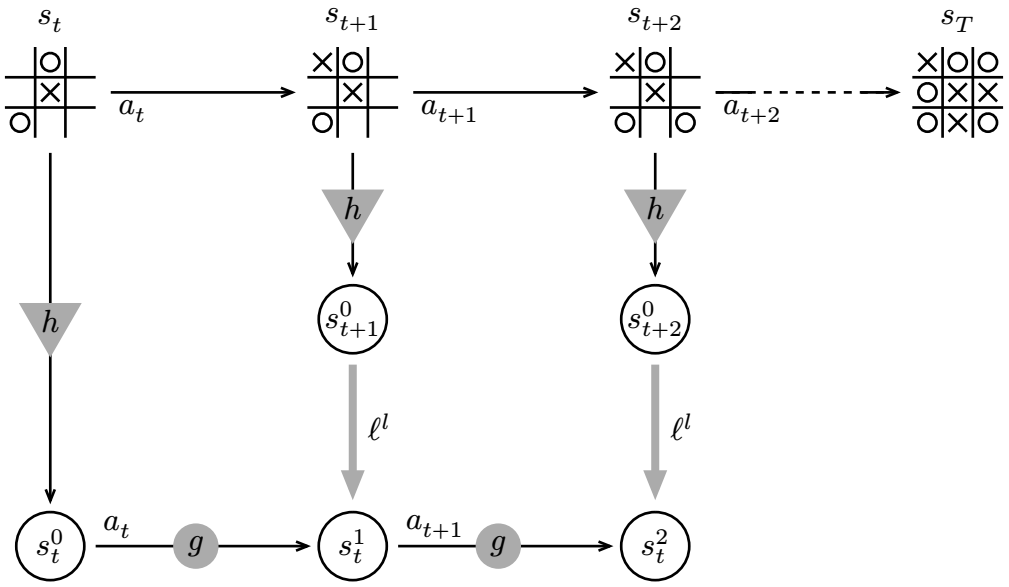

Figure 13: The latent loss ℓ^l introduced in EfficientZero, indicated by the thick arrows. The other MuZero losses are omitted for clarity.

The authors employ a stop-gradient operation on the side of s_{t+n}^0, meaning that gradients from the similarity loss are not applied to the representation network h. This is due to the fact that they closely modeled their architecture after SimSiam [Xinlei Chen and Kaiming He 2020], a self-supervised framework that learns latent representations for images. The authors further justify this decision by treating s_{t+n}^0 as the more accurate representation and therefore using it as a target for the dynamics network's predictions. In Figure 13, the stop-gradient is reflected by the unidirectionality ℓ^l arrow.

In my architecture, I also implement a latent similarity loss, but propose to remove the stop-gradient operation, as outlined in Section IV - C 1. In Section V - B, I evaluate the influence of the latent loss and the stop-gradient operation.

IV: Approach

This chapter is divided into four parts. I begin by reviewing the limitations of MuZero and the reasons behind them, and then move on to a discussion of how I propose to extend the architecture to handle more general games. The third part proposes additional modifications that aim to improve the performance of MuZero in some cases. The final part gives an overview about the MuZero implementation written as part of this thesis, detailing the processes of selfplay and training with my modifications.

IV - A: MuZero Limitations

The implementation of MuZero is designed for single-agent environments and two-player zero-sum games. Furthermore, all games are expected to be deterministic and of perfect information. The causes of these limitations and some implications are briefly discussed below.

IV - A 1: Determinism

In order to plan ahead, the future state of the environment must be predictable for an intial state s^0 and given sequence of actions. The dynamics network g in MuZero is a deterministic function and no chance states are modeled in the architecture. Perhaps unexpectedly, Antonoglou et al. [2022] show that MuZero's performance falls short in a stochastic environment compared to other methods that model stochasticity.

IV - A 2: Perfect Information

Accurately planning ahead also relies on unambiguously identifying the initial state s^0. From a game theoretic standpoint, this requires the game to be with perfect recall and of perfect information (See Section II - B 2.3 and Section II - B 2.2, respectively). In the context of reinforcement learning, it also means that an observation must uniquely capture the current state of the environment.

This aspect is best illustrated by the example of the fifty-moves rule in chess: If 50 moves pass without a capture or a pawn moved, the game may end in a draw. While a human can deduce a history of moves from successive board states, the MuZero agent starts each move afresh, given only the current observation. The current board is therefore not enough information to distinguish a regular game situation from one where the fifty-move rule applies. [Schrittwieser et al. 2020]

IV - A 3: Zero-Sum Games

In the two-player setting, MuZero assumes that the game is zero-sum. This assumption is built into the architecture itself, because it performs negamax search (see Section II - B 4.1.4) during MCTS [Schrittwieser et al. 2020].

Negamax exploits the zero-sum property by using only a single scalar for a node's value. The design of the neural networks (g and f) in MuZero follows this choice and also only predict a single scalar for state values and transition rewards.

Put differently, the original MuZero implementation can only learn to play in favor of one player, for example white in chess or Go. Generating strong moves for the other

player, black in example, is achieved by negating the value and reward predictions for moves of this player. [Schrittwieser et al. 2020]

IV - A 4: Fixed Turn Order

The limitation to games with up to two players implicitly makes assumptions about the turn order. In single player games, trivially only one player can be at turn.

In games with 2 players, alternating turns are assumed. This is not a limitation in practice, since the turn order mechanics of any game can be modeled with the set of available actions A.

As an contrived example, consider castling in chess: It may be viewed as two consecutive turns of the same player, moving king and rook separately. However, by expanding the action set A with a castling move, the assumption about alternating turn order still holds.

The original MuZero implementation uses knowledge of the turn order during Monte Carlo tree search: When MuZero is configured to operate in the two-player setting, the MCTS implementation negates node values at every odd[5] level in the search tree (also known as negamax search, see Section II - B 4.1.4) [Schrittwieser et al. 2020]. When operating in single player environments, this negation is disabled and all node value are maximized. The type of environment is specified in the configuration of the algorithm and can thus be considered as domain knowledge regarding the order of turns.

As outlined in Section IV - A 3, MuZero in fact only learns a policy for one player in zero-sum games. In other words, to generate strong play for both sides, the MCTS implementation is explicitly programmed to exploit the zero-sum property and alternating turn order of the game.

IV - B: Extension to Multiplayer, Stochastic and General Sum Games

As outlined in Section IV - A, the original MuZero implementation is not applicable to environments with more than two players, stochasticity or general-sum games. In this section, I propose modifications to the MuZero architecture which lift these three restrictions, thus improving the generality of the algorithm.

The modified algorithm retains compatibility with the original MuZero environments (board games and the Atari suite) as special cases. The games are still required to be with perfect information.

Planning ahead in a game with more than one player requires some information or assumptions about the behavior of other players. In a two-player zero-sum game, the behavior of the opponent player is easy to model: He will always try to minimize the score of the other player. This assumption does not hold for multiplayer general-sum games with arbitrary payoffs.

[5] or even, depending on who is at turn at the root node

My extension of MuZero to multiplayer games is inspired from the multiplayer backwards induction in game theory, as introduced in Section II - B 4.1.2. My reasoning is that the subgame perfection of backwards induction solutions enables the RL agent to learn a behavior that exhibits strong play, regardless of the actions of other players. Specifically, when the other players act optimally, the overall play should be close to the game theoretic optimal solution. Even if the other players do not perform optimally, the agent should still be able to play reasonably. This is especially important in collaborative games, as it allows the agent to compensate for bad teammates.

I perform a number of changes to MuZero. The updated training setup with all modifications is visualized in Figure 14.

IV - B 1: Per-Player Values

Multiplayer backwards induction requires to keep track of the individual expected utilities and rewards for each player. I follow the design of multiplayer AlphaZero [Petosa and Balch 2019] (see Section III - A) and replace all scalars describing an environment reward, state or node value, with vectors. In an environment with n agents, these reward and value vectors consist of n components:

$$\vec{r} = [r_1, r_2, ..., r_n] \in \mathbb{R}^n$$
$$\vec{v} = [v_1, v_2, ..., v_n] \in \mathbb{R}^n$$

Each component r_i and v_i denotes the individual reward and value of agent i, respectively.

For example, the reward $\vec{r} = [2, -1, 0]$ indicates that the first agent was rewarded with 2, the second agent received a reward of -1, and the third agent got no reward.

Note that in a collaborative game, all individual rewards are shared, as outlined in Section II - B 2.6:

$$r_i = r \text{ for } 1 \leq i \leq n$$

This modification is reflected in Figure 14 by adding vector arrows $\vec{x}$ to multiplayer data.

IV - B 2: Turn Order Prediction

Making informed decisions within the search tree requires not only individual rewards and values, but also an understanding who can make a decision at a particular node. As outlined in Section IV - A 4, in the original MuZero implementation, the turn order is hardcoded for single player and two-player games and therefore represents domain knowledge about the environment.

To achieve a more general algorithm, my implementation does not make any assumptions about the turn order. Instead, the next player at turn is learnt by the dynamics network g. I chose this design since in multiplayer games, the next player at turn may depend on the history of actions.

I propose to add an additional output head w to the dynamics network g:

$$(s^n, r^n, w^n) = g(s^{n-1}, a^{n-1})$$

The output w^n predicts a probability distribution over the set of possible players W, estimating how likely player $y \in W$ is at turn in state s^n:

$$w^n(y) = \Pr(y|s^n)$$

During MCTS, for each state s^n encountered, the current player y^n is assumed to be the one with the highest predicted probability:

$$y^n = \underset{y \in W}{\operatorname{argmax}}\, w^n(y)$$

The turn output w is trained like the reward $\vec{r}$, based on ground-truth labels w_t given by the game simulator during selfplay. For this purpose an additional loss term is introduced

$$\ell^w(w_{t+n}, w_t^n)$$

which aligns the network predictions w_t^n with their respective targets w_{t+n} for all $n = 1...K$, where K is the unroll length.

In my implementation, I use a categorical cross-entropy loss for ℓ^w.

Figure 14 shows this change by the added w in the environment transitions and dynamics network predictions, as well as the additional loss symbol ℓ^w.

IV - B 3: max^n Monte Carlo Tree Search

Following multiplayer backwards induction (Section II - B 4.1.2), the MCTS selection phase considers node values for the player currently at turn only. Specifically, each node value is a vector:

$$\vec{Q}(s^n, a^n) = \gamma \vec{v}^{n+1} + \vec{r}^{n+1}$$

where γ is the reinforcement learning discount factor, as introduced in Section II - A 2.

Let $Q_i(s, a)$ denote the i-th component of this vector.

In state s^k, max^n-MTCS then selects an action a^k as to maximize $Q_i(s^k, a^k)$ where $i = y^k$, the player currently at turn, as outlined in Section IV - B 2.

Specifically, I use Equation 8 to select actions in the search tree, with $Q(s^k, a) = Q_i(s^k, a)$. The constants c_1 and c_2 are detailed in Section V.

IV - B 4: Chance Events

I model stochastic environments with an explicit chance player. He is at turn whenever a chance event occurs in the game.

The occurrence of chance events is given by the dynamics network as part of the turn order prediction w. An additional special player $w_{\mathfrak{c}}$ is added to the set of players W:

$$W' = W \cup \{w_{\mathfrak{C}}\}$$

If $y^n = w_{\mathfrak{C}}$, the current decision node s^n is assumed to be a chance event. The probabilities of the different chance outcomes are predicted by the prediction network f. During MCTS, child nodes of a chance event s^n are selected solely according to the policy p^n.

Like the current player at turn, the occurrence of chance events is trained on ground-truth labels given by the game simulator. The game simulator also provides the exact chance outcomes c_t if state s_t is a chance event. These outcomes are used as targets during training for the policy p as predicted by f.

Chance events are represented by a dice in Figure 14[6]. Note that a chance event s_t differs from regular game states in the figure in two aspects:

- the policy target for training f are the chance outcomes c_t
- the target for the turn order predictions w is the constant chance player $w_{\mathfrak{C}}$

IV - B 5: Training Setup Illustration

The training setup with my proposed modifications is summarized in Figure 14:

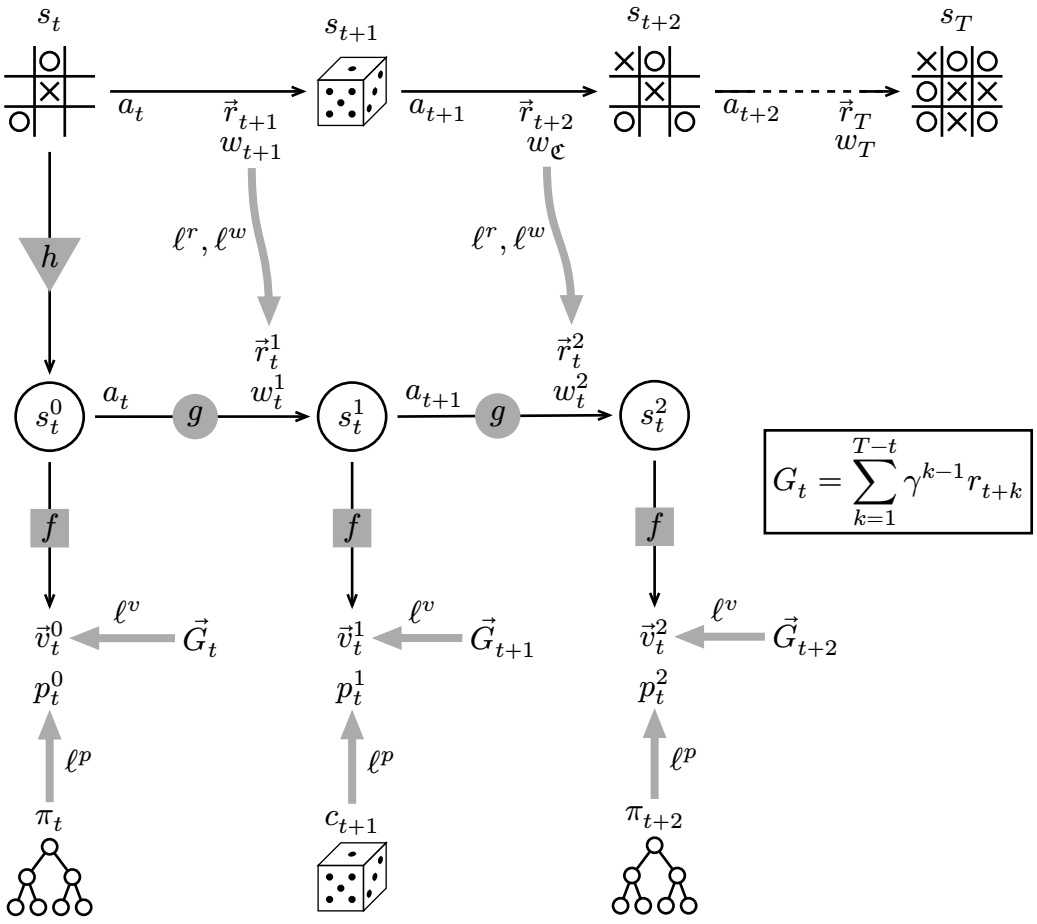

Figure 14: Training setup of my implementation of MuZero for stochastic multi-agent environments

[6]ignore the fact that the game Tic-Tac-Toe actually has no chance events

IV - C: Further Modifications

I spent a significant amount of time and effort trying to get MuZero to train reliably and accurately. At many points I was not sure if low performance is caused by a software bug or bad design choices related to the networks and their training. I therefore explored some ideas on how to improve the design of the architecture to help network convergence. I outline some of the notable changes I implemented below.

IV - C 1: Symmetric Latent Similarity Loss

As outlined in Section III - C, Ye et al. [2021] already layed the groundwork for improvements in sample efficiency by introducing a similarity loss between predictions of h and g. However, I think that their adoption of the stop-gradient operation from SimSiam [Xinlei Chen and Kaiming He 2020] may have been short-sighted:

In SimSiam, the task is to learn discriminative latent representations from input data in a self-supervised manner. Self-supervised learning may suffer from collapsed solutions, where all learned representations end up being very similar or even identical [Tianyu Hua et al. 2021]. Xinlei Chen and Kaiming He [2020] show that a stop-gradient is effective in preventing collapsed solutions in their architecture.

However, Tianyu Hua et al. [2021] show that it is possible to learn significant latent representations without the use of a stop-gradient mechanism: The only requirement is that a decorrelation mechanism of some form must be present in the architecture that penalizes collapsed solutions.

In EfficientZero, I hypothesize that a decorrelation is already achieved by the training losses ℓ^r, ℓ^p and ℓ^v.

Intuitively, if the latent representations of h and g were to collapse, the reward, policy and value predictions can not match their targets. Subsequently, the training loss would stay high. In order to accurately predict these quantities, f and g must encode useful information in the latent space from the observation and sequence of actions, respectively.

Consequently, I see no risk of latent collapse in the EfficientZero architecture, and propose to remove the stop-gradient operation.

IV - C 2: Terminal Nodes in the MCTS

In AlphaZero, the Monte Carlo tree search uses a perfect simulator to determine the next game state for hypothetical actions. This simulator also indicates when the game is over and there are no further moves to search for [Silver, T. Hubert, et al. 2018]. MuZero replaces the perfect simulator with the dynamics network g, which is a learned model of the environment [Schrittwieser et al. 2020].

However, I find it unexpected that this learned model does not include any concept of terminal nodes:

> "*MuZero* does not give special treatment to terminal nodes and always uses the value predicted by the network. Inside the tree, the search can proceed past a terminal node - in this case the network is expected to always predict the same value. This is achieved by treating terminal states as absorbing states during training."
>
> [Schrittwieser et al. 2020]

I hypothesize that this strategy may perform badly in environments where rewards only occur in terminal states: If the reward predictions beyond terminal nodes are not close to zero, these nonzero rewards get backpropagated upwards and might bias the statistics in the search tree. As the backpropagation involves summing over all future rewards[7], the error may accumulate as the search progresses deeper beyond terminal nodes.

My approach is to predict the end of the game with the dynamics network. During MCTS, nodes which are predicted to be terminal are not allowed to be expanded. In this case, when the selection phase reaches a terminal node, the backpropagation phase is triggered immediatly with the terminal node's predicted value.

To realize the prediction of terminal states, I add another special player to the set of possible turn order predictions W: The terminal player $w_{\mathfrak{T}} \in W$ is at turn when the environment reaches a terminal state.

IV - D: Overview of the Implementation

This section summarizes my MuZero implementation and describes the process of search, selfplay and training.

IV - D 1: General

At a high level, my implementation is similar to MuZero: It performs selfplay with MCTS to generate training data, and trains the neural networks on this data. However, MuZero by Schrittwieser et al. [2020] is a large-scale architecture, running selfplay and training in parallel distributed over multiple machines. In contrast, my implementation is designed to operate on a single machine, and uses a single process[8]. Specifically, my implementation performs selfplay and training in alternation.

The time base in my implementation is the total number of steps n performed in the reinforcement learning environment. A step is defined as a single state-action transition, as introduced in Section II - A 1. For example, a two-player game with 10 turns and a single chance event contributes $10 * 2 + 1 = 21$ environment steps. Selfplay and training metrics are logged with respect to this time base, as detailed in Section IV - D 2 and Section IV - D 3.

The neural networks and MCTS generally act on the set of possible actions A: MCTS selects actions $a \in A$, and the network policy predictions p are distributions over the

[7] discounted by the reinforcement learning discount factor γ

[8] parallelization is possible, but was never implemented

set of actions A. Since my implementation also handles stochastic games, chance outcomes must also be included in A The set of possible actions A is thus the union of the set A' of actions players can take and the set of possible chance outcomes C:

$$A = A' \cup C$$

For concrete values for all of the settings and hyperparameters introduced in this section, refer to Section V.

IV - D 2: Data Generation

To generate training data, games are played and their trajectory is recorded. Specifically, T steps are taken in each game, until the game terminates naturally, or the configurable setting M is reached, at which the game is truncated.

At chance events, that is when $w_t = w_{\mathfrak{C}}$, the game simulator provides the chance outcomes c_t as a distribution over actions:

$$c_t(a|s_t) = P(s, a) \tag{9}$$

where $a \in C$, the set of possible chance actions.

At each time step $t = u, u+1, ..., T-1, T$ a tuple D of training data is recorded:

$$D = \begin{cases} (s_t, a_t, r_{t+1}, w_{t+1}, \pi_t, G_t) \text{ if } w_t \neq w_{\mathfrak{C}} \\ (s_t, a_t, r_{t+1}, w_{t+1}, c_t, G_t) \text{ if } w_t = w_{\mathfrak{C}} \end{cases}$$

The recorded trajectory begins at the first time step u where the game provides an observation, that is, the current player at turn w_t is not the chance player:

$$u = \min_t t \text{ subject to } w_t \notin \{w_{\mathfrak{T}}, w_{\mathfrak{C}}\} \wedge 0 \leq t \leq T$$

The tuple of training data D contains the following data:

- s_t: game state
- a_t: action taken
- r_{t+1}: experienced reward
- w_{t+1}: player at turn in the next game state s_{t+1}
- π_t: MCTS policy according to Equation 6
- c_t: chance outcomes according to Equation 9
- G_t: n-step return according to Equation 1

In all games, if the state s_t is a chance event, the action a_t is sampled from the distribution of chance outcomes c_t, provided by the game simulator:

$$a_t \sim c_t \text{ if } w_t = w_{\mathfrak{C}}$$

When a game finishes, its metrics such as the score (cumultative reward) are logged. The metrics are associated with the total step number n (see Section IV - D 1) when the game was started. As an example, if all games take 10 steps, the score of the first game is logged at $n = 0$, the second game's score at $n = 10$, and so on.

IV - D 2.1: Warmup with random play

If the number of total environment steps n is below a configurable threshold R at the beginning of a game, player actions are selected randomly in the game. Specifically, all actions a_t at time t where $w_t \neq w_{\mathfrak{C}}$ are sampled from a uniform distribution over the set of legal actions $A(s_t)$. MCTS is not used in these games, and no neural network inferences take place.

This period of random play quickly generates training data so that the dynamics network g can learn a model of the environment in a minimal amount of wall clock time.

IV - D 2.2: Selfplay and Search

As soon as the number of total environment steps n is above the threshold R, selfplay with MCTS is used to generate data.

Dirichlet noise is blended into the root node according to Equation 7. Actions are selected as outlined in Section IV - B 3.

IV - D 3: Training

The latest N tuples of training data are stored in a buffer for training. The buffer additionally keeps track of the total number i of training data tuples added to the buffer, and the total number o of training data tuples consumed by the training process. After each played game, the generated training data is added to the buffer, and i is incremented by length of the trajectory: $i' = i + (T - u + 1)$, where u and T denote the first and last time step of the game trajectory, as outlined in Section IV - D 2.

When data is sampled from the buffer for training, the number o is incremented by $K * B$, where K is the unroll length, and B the batch size: $o' = o + (B * K)$

The ratio $\frac{o}{i}$ is roughly held constant at the setting E by training for an appropriate amount of times after each game:

$$\frac{o}{i} \approx E \tag{10}$$

A batch of training data is sampled from the buffer such as the first game state s_0 contains an observation, that is $w_0 \notin \{w_{\mathfrak{T}}, w_{\mathfrak{C}}\}$. Also, it is required the remaining length of the game starting at s_0 is greater or equals K to ensure a complete unroll is possible. Training and losses are described in in Section II - D 5.3.2 and Section IV - B 2. For the latent similarity loss ℓ^l I use negative cosine similarity.

I a discrete scalar support F for the reward and value predictions, as described in Section II - D 5.3.2, but without the transform $e(x)$.

V: Evaluation

This chapter describes the experiments I perform. The first part describes the games I use as reinforcement learning environments in my experiments. The second and third part present the setup of the ablation studies I perform to examine the effect of the proposed symmetric latent loss and the use of terminal nodes in the search tree, respectively. The final part presents the setup I use to examine if my multiplayer modifications are able to learn in a collaborative multiplayer game.

V - A: Training Environments

I apply my implementation of MuZero to two different games. In this section, I describe the game mechanics and observation tensors.

V - A 1: The Game Catch

Catch is a simple single player game which is primarily designed to test if a reinforcement learning system is capable of learning anything at all. The game consists of a two-dimensional grid with a configurable number of columns c and rows r.

At the beginning of the game, a single block spawns in the top row, in a random column. This block descends by one row in every round of the game, and the game ends when it reaches the bottom row.

The player can move another block in the bottom row and has to "catch" the falling block by moving to the same column. Movement is possible via three actions, which respectively move the bottom block one column to the left, right or let it stay in its current position. The player's block always starts in the center column when the game starts.

The game has no intermediate rewards, but a single terminal reward in the last round, which is 1 if the player caught the falling block, and -1 otherwise.

Figure 15 shows a visualisation of the game for $c = 5$ and $r = 10$ after three rounds:

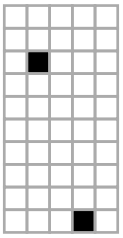

Figure 15: The game Catch in the third round

In my experiments I use a grid size of $c = 5$ and $r = 10$.

V - A 1.1: Observations

An observation in Catch consists of a 2D image with a single plane, representing the game grid. The image width and height thus equal the number of rows r and columns c,

respectively. The image has zeros everywhere except for the two blocks, where the corresponding pixel is set to one.

The observation includes a second tensor with one element, set to a constant 1.[9]

V - A 2: The Collaborative Game Carchess

- Carchess is a round-based collaborative multiplayer game on a grid-like structure. Core element of the game are a number of lanes that cross at intersections, as well as traffic lights that can be toggled to control the flow of traffic.

A screenshot of a graphical frontend to the game is shown in Figure 16. Instead of traffic lights, the controls are visualized as barriers in this version, but they work the same way.

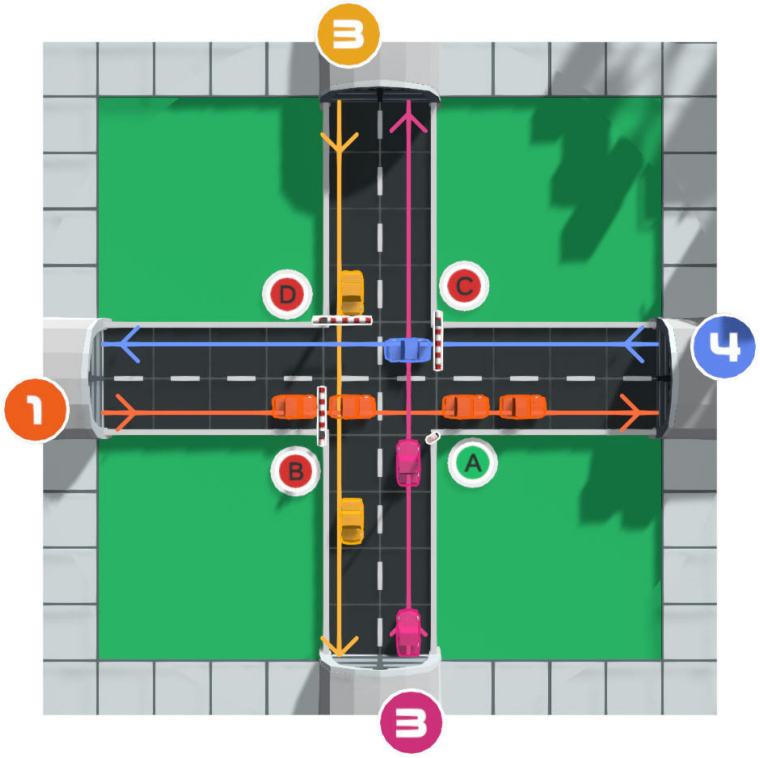

Figure 16: Screenshot of the game Carchess

The course of the lanes and the position of traffic lights is defined by the specific map the game is played on.

In each round, all players must each toggle one distinct traffic light, so that for n players, exactly n traffic lights are toggled. After all players took their turn, a number of m simulation steps are performed, in which the cars are advanced along their lanes and new cars may spawn. Finally, the spawn counts for all lanes are updated, and the next round begins.

[9]This tensor exists for technical reasons only: It is designed to encode the current player in multi-player games, but degrades to a single-element constant in single player games.

In each simulation step, all cars move forward one field, with two exceptions:

- the car is on a field with a red traffic light.
- the next field is already occupied by a waiting car, i.e. the next car cannot move because it is blocked by other cars or a traffic light.

The last rule does not always apply in intersections, because cars may collide there. Specifically, a car entering an intersection only waits for a car already in the intersection if it is on the same lane or drives in the same direction.

In all other cases, two or more cars may end up in the same field after a simulation step, which is interpreted as a car crash. The colliding cars are removed from the game and a negative reward is emitted for each crashed car.

Each lane also has a spawn counter which indicates the number of cars that may spawn during the next simulation steps. Specifically, when the first field of a lane is free after a simulation step, a new car is placed there if the spawn count is nonzero. Subsequently, the spawn counter is decremented by one.

At the end of each round, each of the spawn counters are incremented by an individual random amount x drawn from a uniform Distribution $x \sim \text{unif}\{s_{\min}, s_{\max}\}$. The maximum spawn count is bounded by the capacity of the lane times a constant $0 < c < 1$ which is a setting of the game. Note that the spawn counters may be incremented even if no cars spawned during the last simulation, so the cars waiting to be spawned can "pile up" (up to $s_{\max}$).

The objective of the players is to manage the traffic lights to maximize the number of cars which reach the end of their lane. Each car reaching the end of their lane yields a positive reward for the players.

The specific settings of Carchess I use are listed in Table 2:

Parameter	**Value**
Number of players n	2
Number of simulation steps per round m	5
Number of total rounds per game r	10
Maximum car density on a lane c	0.5
Minimum spawn count per round $s_{\min}$	0
Maximum spawn count per round $s_{\max}$	4
Reward for each car reaching the end	2
Reward for each colliding car	-1

Table 2: Settings of Carchess in my evaluation

The map I use is named "tutorial" and shown in Figure 16. The map's properties are outlined in Table 3:

Property	Value
Width w	10
Height h	10
Number of lanes l	4
Maximum lane length a	10

Table 3: Properties of the Carchess map "tutorial"

V - A 2.1: Observations

I use observations of the following form in the game Carchess, each observation consisting of three tensors:

The first observation tensor is a three-dimensional image encoding the position of lanes, cars and traffic lights on the grid. The image width and height correspond to the size of the map (w, h) (see Table 3 for specific numbers). The number of image planes depends on the map and the game settings.

Specifically, there is one image plane per lane to encode the positions of cars on that lane. I use a simple onehot encoding where 1 denotes the presence of a car belonging to the specific lane, and 0 everywhere else. Another image plane encodes the traffic lights, where -1 denotes a red light (cars cannot pass), 1 denotes a green light (cars may pass), and 0 everywhere else.

The spawn counts of the lanes are onehot-encoded over several image planes: There exists an image plane for every other possible spawn count integer $x = 0, 1, ..., \lfloor c(a-2) \rfloor$, where a is the number of fields in the longest lane. Specifically, an image plane representing a spawn count of x is first initialized with zeros everywhere. Then, for each lane which currently has a spawn count of x, a 1 is set in the plane, at the starting location of the lane (where the cars will spawn).

The total number of image planes i can thus be computed by:

$$i = 2 + l + \lfloor c(a-2) \rfloor$$

with l being the number of lanes, and a denotes the number of fields in the longest lane. The specific values of l, a and c are given in Table 3 and Table 2.

The other two observation tensors are one-dimensional onehot-encodings of the player currently at turn and the current round number in the game, respectively. Their respective lengths are defined by the game settings n and r. Refer to Table 2 for the specific settings I use.

V - B: Ablation Study: Latent Loss Variants

I perform experiments to investigate the effects of the different latent loss variants. Specifically, I compare these three variants:

- no latent loss, like the original MuZero implementation proposed by Schrittwieser et al. [2020]
- latent loss with stop-gradient, like in EfficientZero proposed by Ye et al. [2021], see Section III - C
- symmetric latent loss, which I propose in Section IV - C 1

At each training step, I choose the unroll length $K \sim \text{unif}\{1, K_{\text{max}}\}$ to ensure the unrolling starts at terminal states, too.

I use the game Catch for this evaluation, as introduced in Section V - A 1.

Apart from the latent loss weight, I fixed all hyperparameters to the values shown in

Parameter	**Value**
Batch size B	256
Number of latent features	50
Support size for value and reward predictions $\lvert F \rvert$	11
Unroll length K_{max}	3
Optimizer	Adam[10]
Weight decay	1×10^{-5}
Learning rate	1×10^{-3}
Loss weight for ℓ^v	1
Loss weight for ℓ^r	1
Loss weight for ℓ^p	1×10^{-2}
Loss weight for ℓ^w	1
Discount factor γ	0.98
pUCT formula c_1	1.5
pUCT formula c_2	1×10^3
Dirichlet exploration noise ε	0.2
Dirichlet exploration noise α	0.5
Number of MCTS iterations per move	30
Training / Selfplay ratio E	100
Random play steps R	3×10^3
Number of Steps in Buffer	1×10^4

Table 4: Hyperparameters of the neural networks and MCTS for my evaluation of the latent loss

[10] [Diederik P. Kingma and Jimmy Ba 2014]

V - B 1: Neural Network Architecture

In this section I describe the neural networks I use. If present, a residual block $r(x)$ consisting of a layer $f(x)$ denotes a skip connection around the layer:

$$r(x) = x + f(x)$$

V - B 1.1: Representation Network

All observation tensors are flattened and concatenated to yield a tensor with 51 elements. This observation tensor is processed by the following layers:

- Fully connected layer with 50 output neurons, ReLU activation

The latent representations in this architecture have therefore 50 dimensions.

V - B 1.2: Dynamics Network

The 50-dimensional latent tensor is concatenated with the 5-element action onehot to yield a tensor with 55 elements.

- Residual block consisting of: Fully connected layer with 55 output neurons, ReLU activation
- Residual block consisting of: Fully connected layer with 55 output neurons, ReLU activation
- Fully connected layer with 64 output neurons

The resulting 64-dimensional tensor is split and reshaped into three tensors, with 50 elements, shape 11×1 and 3 elements, for the latent output, reward support and turn order prediction, respectively.

V - B 1.3: Prediction Network

The 50-dimensional latent tensor is processed by the following layers:

- Fully connected layer with 16 output neurons, ReLU activation
- Residual block consisting of: Fully connected layer with 16 output neurons, ReLU activation
- Fully connected layer with 16 output neurons

The resulting 16-dimensional tensor is split and reshaped into two tensors, with shape 11×1 and 5 elements, for the value support and policy predictions, respectively.

V - C: Ablation Study: Terminal Nodes

I perform experiments to investigate the effect of predicting the end of the game and using terminal nodes in the search tree. Specifically, I compare these two variants:

- no terminal nodes, like the original MuZero implementation proposed by Schrittwieser et al. [2020]. During training, absorbing states are used for up to K steps beyond the game end.
- using terminal nodes, like proposed in Section IV - C 2

I use the game Catch for this evaluation, as introduced in Section V - A 1. I use the same setup as in Section V - B 1, with the following differences:

- the latent loss weight for ℓ^l is set to zero

V - C 1: Neural Network Architecture

I use the same neural networks as outlined in Section V - B 1.

V - D: Application to Carchess

I evaluate my multiplayer extension on the game Carchess, as outlined in Section V - A 2. I use the hyperparameters according to Table 5.

Parameter	Value
Batch size B	512
Number of latent features	150
Support size for value and reward predictions $\|F\|$	11
Unroll length K	6
Optimizer	MADGRAD[11]
Learning rate	1×10^{-3}
Loss weight for ℓ^l	0.1
Loss weight for ℓ^v	0.1
Loss weight for ℓ^r	1
Loss weight for ℓ^p	0.1
Loss weight for ℓ^w	1
Discount factor γ	0.98
pUCT formula c_1	2
pUCT formula c_2	1×10^3
Dirichlet exploration noise ε	0.2
Dirichlet exploration noise α	0.5
Number of MCTS iterations per move	30
Training / Selfplay ratio E	100
Random play steps R	1×10^5
Number of Steps in Buffer	3×10^4

Table 5: Hyperparameters of the neural networks and MCTS for my evaluation on Carchess

V - D 1: Neural Network Architecture

In this section I describe the neural networks I use. If present, a residual block $r(x)$ consisting of a layer $f(x)$ denotes a skip connection around the layer:

$$r(x) = x + f(x)$$

V - D 1.1: Representation Network

The image observation tensor is processed by a stack of these layers:

[11] [Aaron Defazio and Samy Jelassi 2021]

- Convolution with kernel size 1x1, 64 Filters
- 4x Residual Blocks, each consisting of: ReLU pre-activation and Convolution with kernel size 3x3, 64 Filters and Padding of 1 in each direction
- Per-channel max pooling along the height and width of the image, concatenation of the tensors. Specifically, an image tensor of shape $w \times h$ and with c channels yields two tensors of shape $c \times w$ and $c \times h$, which are concatenated to a $c \times (w + h)$ tensor.

The resulting tensor of shape 64x20 is flattened and concatenated with the other observation input tensors. A linear layer processes this tensor and outputs a 150-dimensional latent tensor.

V - D 1.2: Dynamics Network

The 150-dimensional latent tensor is concatenated with the 100-element action onehot to yield a tensor with 250 elements. The tensor is then processed by a stack of these layers:

- 3x Residual Blocks, each consisting of: ReLU pre-activation and Linear Layer of 250 output neurons
- Fully connected layer with 26 output neurons

The resulting 26-dimensional tensor is split and reshaped into two tensors, with shape 11×2 and 3 elements, for the reward support and turn order prediction, respectively.

The latent output s^{k+1} is produced by a single GRU[12] cell operating on the latent input s^k and action a^k. Specifically, the input hidden state of the GRU cell is the 150-dimensional tensor representing state s^k and the cell input is the action onehot representing a^k.

V - D 1.3: Prediction Network

The 150-dimensional latent tensor is processed by the following layers: The tensor is then processed by a stack of these layers:

- 4x Residual Blocks, each consisting of: ReLU pre-activation and Linear Layer of 150 output neurons
- Residual Block, consisting of: ReLU pre-activation and Linear Layer of 122 output neurons

The resulting 122-dimensional tensor is split and reshaped into two tensors, with shape 11×2 and 100 elements, for the value support and policy predictions, respectively.

[12]as proposed by [Kyunghyun Cho et al. 2014]

VI: Results

In this section I report the results of my training runs.

The x-axis of all plots shows the number of steps n taken in the environment. For plotting, the number of data points is resampled to 1000 steps.

For example, if a game takes 10 steps, a datapoint for the score is recorded every 10 steps. If the training is performed for 50000 total steps, this generates a total of 5000 data points. The resampling picks 1000 equidistant samples from these data points, so that data points are plotted in increments of 50 environment steps. The specific resampling algorithm I use is the reservoir sampling implementation in tensorboard.

The use of t in this chapter always refers to the resampled time domain, that is $t \in \mathbb{N} \wedge t < 1000$.

VI - A: Ablation Study: Latent Loss Variants

Here I report the results of the ablation study for the different latent loss variants. Specifically, Figure 17 shows a direct comparison of the score during selfplay for each variant.

The plot contains three curves, one for each latent loss variant. Each curve is the result of averaging over 4 runs with different random seeds.

Since Catch has a a binary outcome $z \in [-1, 1]$, each plotted data point x'_t at time step t is smoothed with an exponential moving average:

$$x'_t = \sum_{i=0}^{t} 0.97^i z_{t-i} \tag{11}$$

where z_t is the game outcome at time t.

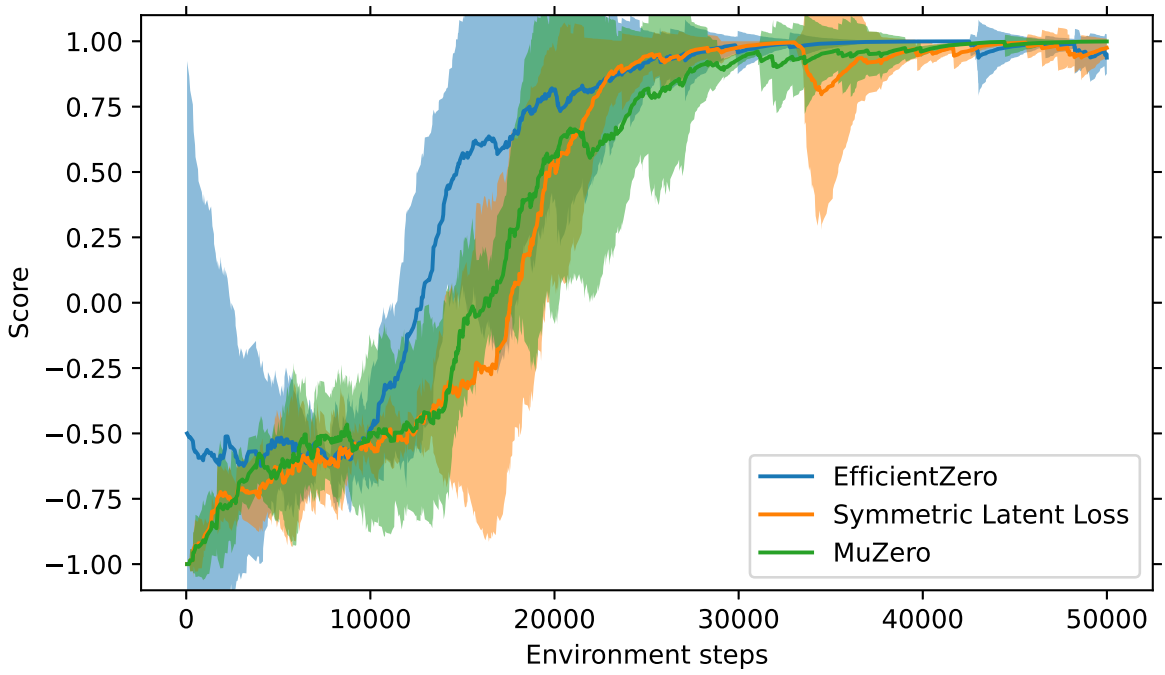

Figure 17: Comparison of the selfplay scores across variants, averaged over 4 runs each. Error bands show 90% confidence intervals.

Table 6 shows the final scores after 50k environment steps. The mean and standard deviation are calculated over the last 400 game outcomes[13] distributed over all runs. Specifically, each run contributes 100 data points z_t, where $900 \leq t < 1000$. This corresponds to the (resampled) data points at environment steps $45000 \leq n < 50000$.

Variant	**Final score $\mu \pm \sigma$**
MuZero	1.00 ± 0.0
EfficientZero	0.96 ± 0.28
Symmetric Latent Loss	0.96 ± 0.26

Table 6: Mean and standard deviation of the score during selfplay after 50k environment steps, averaged over 4 runs and last 100 data points each.

[13] in the resampled time domain

VI - B: Ablation Study: Terminal Nodes

Here I report the results of the ablation study for terminal nodes in the search tree. Specifically, Figure 18 shows a direct comparison of the score during selfplay for each variant.

The plot contains two curves, one for runs with terminal nodes, and one for runs without terminal nodes. Each curve is the result of averaging over 4 runs with different random seeds. Smoothing is applied as outlined in Equation 11.

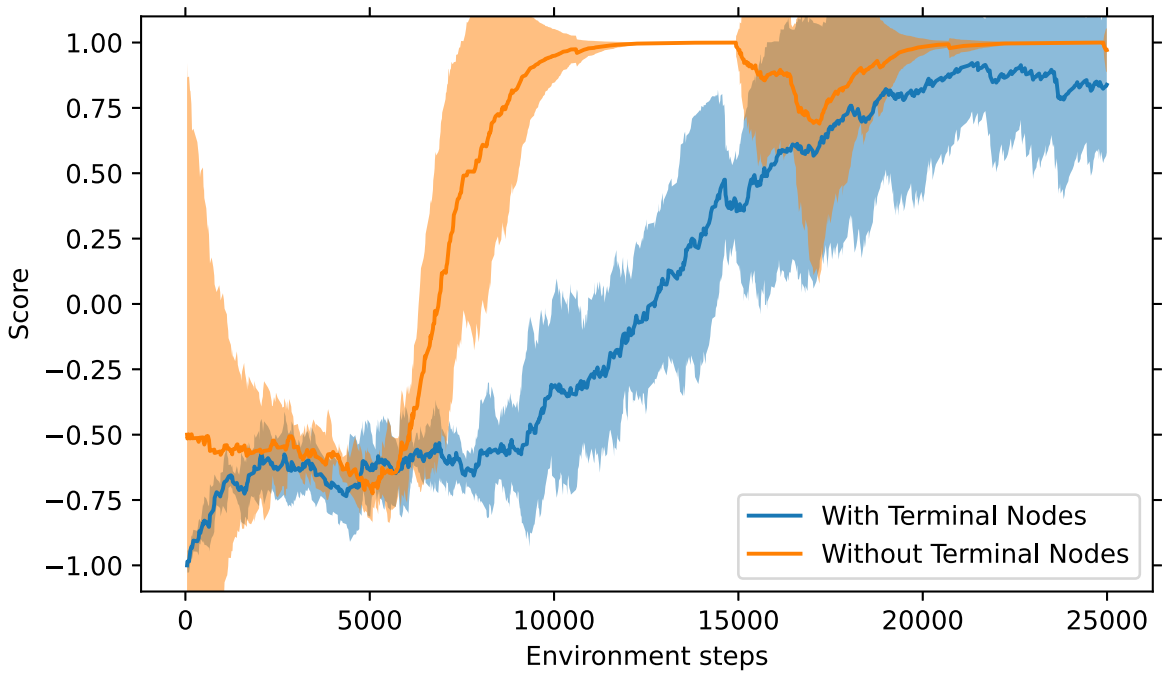

Figure 18: Comparison of the selfplay scores across variants, averaged over 4 runs each. Error bands show 90% confidence intervals.

Table 6 shows the final scores after 25k environment steps. The mean and standard deviation are calculated over the last 40 game outcomes[14] distributed over all runs. Specifically, each run contributes 10 data points z_t, where $990 \leq t < 1000$. This corresponds to the (resampled) data points at environment steps $24750 \leq n < 25000$.

Variant	Final score $\mu \pm \sigma$
Without Terminal Nodes	0.85 ± 0.53
With Terminal Nodes	0.90 ± 0.44

Table 7: Mean and standard deviation of the score during selfplay after 25k environment steps, averaged over 4 runs and last 10 data points each.

[14] in the resampled time domain

Additionally, I present an analysis of the Monte Carlo search tree of both variants near the end of training. Specifically, I pick multiple logged selfplay games at random from the last 100 selfplay games in each run. I then manually inspect the final search tree, that is, after all MCTS iterations have passed.

The key aspects observed across multiple search trees are summarized below.

For both variants:

- reward predictions for all nodes correspondig to game states are zero, except for the terminal states, where the predictions are near 1 or -1.

For the variant with terminal nodes:

- terminal nodes are present in the search tree, corresponding to terminal states of the game
- all node values are found to be in the interval $[-1, 1]$

For the variant without terminal nodes:

- node values v with magnitudes $|v| > 2$ are present in the tree
- no terminal nodes are present and the search progresses beyond nodes corresponding to terminal states of game
- nodes ≥ 3 levels beyond game end have nonzero reward predictions

VI - C: Application to Carchess

Here I report the results of the applications of my implementation of MuZero to Carchess. Figure 19 shows the score during a single run of selfplay over 824584 game steps on the map "tutorial". I define the score in Carchess as the cumultative reward as experienced by a single player.

At each time step t in the plot, the mean and 95% confidence interval is computed over the last 30 data points. The plot thus shows the average score, as computed over a sliding window of size 30.

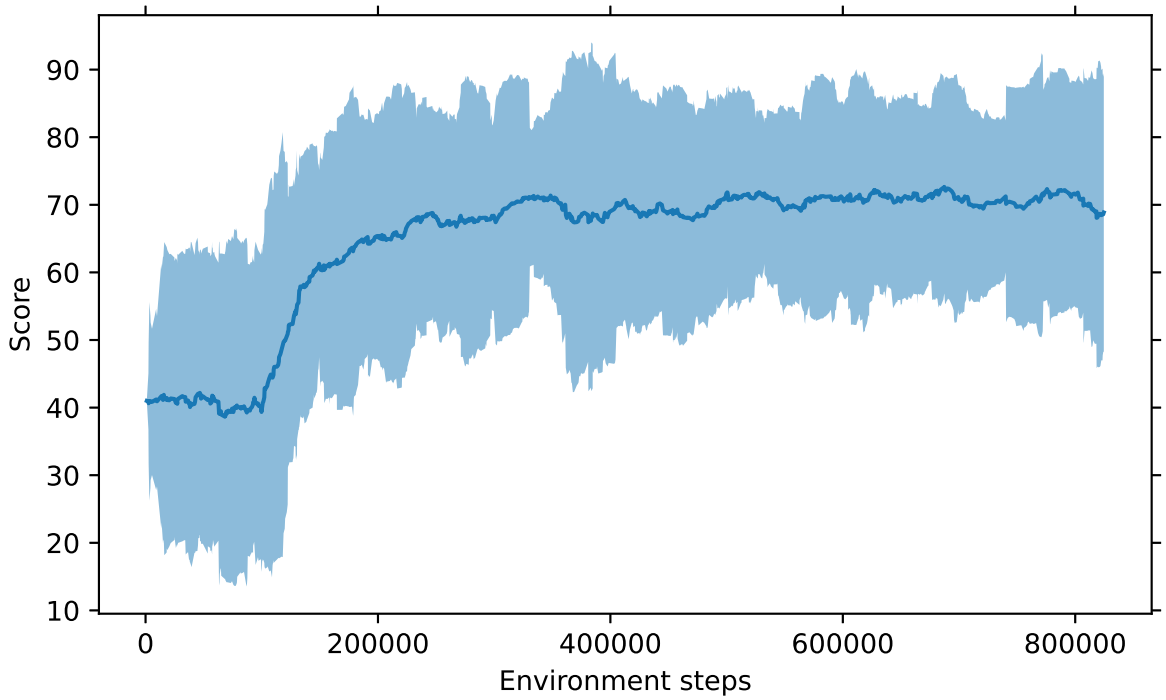

Figure 19: Mean score during a single run of selfplay. The error band shows a 95% confidence interval of the last 30 scores at each data point.

The mean and standard deviation of the selfplay score over the last 500 data points is $\mathbf{61.02 \pm 15.54}$. This corresponds to games played during environment steps $412292 \leq n < 824584$.

VII: Discussion

In this chapter I discuss the results presented in Section VI.

VII - A: Ablation Study: Latent Loss Variants

Here I provide a discussion about the results of the ablation study for the latent loss variants, as presented in Section VI - A.

At first glance, Figure 17 shows similar training dynamics among the three variants: All variants eventually learn to play the game with near-perfect accuracy, as indicated by the almost constant score of 1 by the end of the training. Accordingly, based on the final scores in Table 6, no best or worst performing variant can be identified with statistical significance.

In Figure 17, the EfficientZero variant learns the fastest, given the fast rise of the score s between $-0.5 < 0.5$ curve, beginning at $n = 12000$ environment steps. Compared to the MuZero variant with no latent loss, this is the expected result and qualitatively aligns with the findings reported by Ye et al. [2021]. Regardless, learning slows again when the score approaches 1. Overall, no practical speedup in terms of reaching the final accuracity is achieved.

My proposed symmetric latent loss does not perform any different than MuZero. On one hand, I did expect my variant to perform at least more similar to EfficientZero, because the additional latent loss provides feedback to both the representation h and dynamics network g. However, the findings suggest that the symmetric latent loss might provide detrimental training signals to the representation network h.

On the other hand, the performance is not significantly worse than MuZero, indicating that no latent collapse occurs without the stop-gradient operation, as would be evident from very poor playing performance. This is consistent with my theory that the other training losses achieve sufficient decorrelation of the latent space to prevent a collapse, as outlined in Section IV - C 1.

However, care must be taken when comparing the variants based on the data in Section VI - A: First, the differences between variants are small compared to the statistical noise, so that the observed differences can also be explained by the variance of the data. Second, the game Catch is very simple and might not exhibit enough complexity for the neural networks used.

Exploring bigger, more challenging environment with latent losses may therefore be an interesting direction for further research about latent losses in MuZero.

VII - B: Ablation Study: Terminal Nodes

Here I discuss the results of my ablation study as presented in Section VI - B.

Figure 18 shows that the two variants perform differently. Specifically, the variant without terminal nodes learns faster, it reaches the optimal game score of 1 after 10k environment steps. In contrast, the variant with terminal nodes learns slower and does not reach perfect play by the end of 25k environment steps, as shown in Table 7.

However, a direct interpretation of the final selfplay scores is difficult because the variance of the runs is very high in comparison with the difference of in scores.

My original hypothesis was that predictions for nodes beyond game end might lead to bias in the node values of the tree. I assumed that this bias slows down learning, so the observed result is surprising to me.

The results of manually inspecting the search trees reveal several interesting aspects: First, in the variant without terminal nodes, there is in fact bias in the node value estimates, as evident by deviations from the optimal node values and node values outside of the range $[-1, 1]$ of possible values for the game Catch. Second, nonzero reward predictions for nodes beyond terminal states are very likely the cause of the observed bias: I come to this conclusion, as there is no other way of creating node values v with magnitudes $|v| > 2$ in the tree, given correct reward predictions for nodes correspondig to game states. Third, the nonzero reward predictions appear at depth $K - 1$ in the tree, indicating that the training of absorbing states does not generalize well for unroll counts higher than the one used during training, K. Finally, the variant with terminal nodes correctly predicts terminal nodes and the search does not expand beyond them.

From the faster learning with terminal nodes, I conclude that the bias in the search may actually help convergence in the specific environment tested. The larger magnitudes of the node values might pronounce value differences for distinct actions, helping the pUCT selection formula focusing on more promising nodes.

An interesting direction for future research might be to further explore the effects of predicting terminal nodes in different types of environments.

VII - C: Application to Carchess

Finally, I discuss the results of applying my implementation of MuZero to the game Carchess. Note that the results are based on a single run of training, so care must be taken when interpreting the results.

Figure 19 shows three stages of training: In the beginning, random play is used to quickly generate training data up to $n < 1 \times 10^5$ environment steps. The scores encountered during this period of random play also serve as the baseline performance. Afterwards, the algorithm switches to selfplay games, and the score rises immediatly. This indicates that the neural networks learned a useful model of the environment from random play that can be used for move selection and planning ahead.

For a number $1 \times 10^5 < n < 3 \times 10^5$ of environment steps, the mean score increases slowly. This phase indicates successful learning and improvement of the policy and value predictions. Afterwards, the mean score settles at 61.02, which seems to be the best play achievable with this implementation and chosen hyperparameters.

I contribute the high variance of the scores to the of chance events in the game. These chance events come in the form of updating the car spawn counts, there are 4 chance events per round with up to 5 distinct chance outcomes.

The best score on this map with the chosen parameters is roughly 80-90 as achieved by human players. Overall, I am satisfied with the results. They serve as an proof of concept for my proposed multiplayer extension of MuZero and indicate that it is capable of learning.

Tuning of hyperparameters, as well as further experiments with the proposed multiplayer architecture on, for example other Carchess maps or entirely different environments, are left for future work.

VII - D: Limitations of my Approach

A couple of shortcomings can be identified in my multiplayer extension:

As shown by Sturtevant, n.d., MCTS with a pUCT formula in multiplayer games converges to an optimal equilibrium strategy. However, this the resulting strategy may not precisely be the optimal strategy computes by backwards induction in game theory, as introduced in Section II - B 4.1.2.

Modeling every decision point of every player in the MCTS creates a very high branching factor, which makes it difficult to reach high search depths. An alternative is to use opponent models that model the moves of other players in a deterministic fashion. This effectiveld transforms a multiplayer game into a single player game by hiding the other players's actions in the environment transitions. [Jannis Weil et al. 2023]

Exploring further limitations of my approach and how to overcome them could be an interesting direction for future research.

VIII: Conclusion

In this thesis, I analyze MuZero, a reinforcement learning algorithm that learns a model of the environment and uses it to plan ahead in a Monte Carlo tree search (MCTS).

I outline the limitations of the original implementation, in particular how it is restricted to deterministic single-agent environments and zero-sum two-player games. The main contribution is an extension of MuZero to more general environments with any number of agents, stochasticity, and general rewards.

For this, I modify the environment model such that it learns which agent can make a decision at any given state of the environment. Also, I replace the negamax MCTS of the original MuZero implementation with $\max^n$ MCTS, so that the learned behavior approaches the game theoretic optimal solution for all agents involved.

I implement and successfully evaluate my proposed architecture on the collaborative multiplayer game Carchess. My implementation is capable of learning this game, and achieves a reasonable score.

I also propose two additional modifications of MuZero. First, a modification of the MCTS to handle terminal states during search. Second, the introduction of a symmetric latent loss, building on the work of EfficientZero by Ye et al. [2021].

For each modification, I perform an ablation study to investigate its effect: In the specific environment and settings I use, the two modifications perform worse than (in the case of terminal nodes) and indistinguishable from (in the case of the symmetric latent loss) the original MuZero baseline.

Bibliography

D. Silver, T. Hubert, et al., "A general reinforcement learning algorithm that masters chess, shogi, and go through self-play," *Sci.*, vol. 362, no. 6419, pp. 1140–1144, Dec. 7, 2018, doi: 10.1126/science.aar6404.

D. J. Mankowitz, A. Michi, et al., "Faster sorting algorithms discovered using deep reinforcement learning," *Nature*, vol. 618, no. 7839, pp. 257–263, Jun. 7, 2023, doi: 10.1038/s41586-023-06004-9.

A. Mandhane, A. Zhernov, et al., "Muzero with self-competition for rate control in Vp9 video compression," Feb. 14, 2022.

J. Schrittwieser, I. Antonoglou, et al., "Mastering atari, go, chess and shogi by planning with a learned model," *Nature*, vol. 588, no. 7839, pp. 604–609, Dec. 23, 2020, doi: 10.1038/s41586-020-03051-4.

W. Ye, S. Liu, T. Kurutach, P. Abbeel, and Y. Gao, "Mastering atari games with limited data," Dec. 12, 2021.

R. S. Sutton, and A. G. Barto, *Reinforcement Learning: An Introduction*, Second, MIT Press, 2018.

K. Ritzberger, *Foundations of Non-Cooperative Game Theory*, Oxford University Press, 2002.

D. Fudenberg, and J. Tirole, *Game Theory*, MIT Press, 1991.

J. P. Zagal, J. Rick, and I. Hsi, "Collaborative games: Lessons learned from board games," *Simul. & Gaming*, vol. 37, no. 1, pp. 24–40, Mar. 2006, doi: 10.1177/1046878105282279.

J. Marschak, and R. Radner, *Economic Theory of Teams*, Yale University Press, 1972.

L. Allis, "Searching for solutions in games and artificial intelligence," Thesis, Maastricht Univ., 1994.

D. E. Knuth, and R. W. Moore, "An analysis of alpha-beta pruning," *Artif. Intell.*, vol. 6, pp. 293–326, 1975.

C. Browne, E. Powley, et al., "A survey of monte carlo tree search methods," *IEEE Trans. Comput. Intell. AI Games*, vol. 4, no. 1, pp. 1–43, Feb. 3, 2012, doi: 10.1109/TCIAIG.2012.2186810.

M. Świechowski, K. Godlewski, B. Sawicki, and J. Mańdziuk, "Monte carlo tree search: A review of recent modifications and applications," *Artif. Intell. Rev.*, vol. 56, no. 3, pp. 2497–2562, Jul. 19, 2022, doi: 10.1007/s10462-022-10228-y.

N. Petosa, and T. Balch, "Multiplayer Alphazero," Dec. 9, 2019.

D. Silver, A. Huang, et al., "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, Jan. 27, 2016, doi: 10.1038/nature16961.

M. Müller, "Computer go," *Artif. Intell.*, vol. 134, no. 1, pp. 145–179, 2002, doi: 10.1016/S0004-3702(01)00121-7.

D. Silver, and G. Tesauro, "Monte-carlo simulation balancing," in *Proc. 26th Annu. Int. Conf. Mach. Learn.*, Jun. 14, 2009, pp. 945–952, doi: 10.1145/1553374.1553495.

S.-C. Huang, R. Coulom, and S.-S. Lin, "Monte-carlo simulation balancing in practice," in *Comput. Games*, 2011, pp. 81–92, doi: 10.1007/978-3-642-17928-0_8.

P. Baudiš, and J.-l. Gailly, "PACHI: State of the art open source go program," in *Advances Comput. Games*, 2012, pp. 24–38, doi: 10.1007/978-3-642-31866-5_3.

M. Enzenberger, M. Müller, B. Arneson, and R. B. Segal, "Fuego - an open-source framework for board games and go engine based on monte carlo tree search," *IEEE Trans. Comput. Intell. AI Games*, vol. 2, no. 4, pp. 259–270, Oct. 14, 2010, doi: 10.1109/TCIAIG.2010.2083662.

S. Gelly, Y. Wang, R. Munos, and O. Teytaud, "Modification of UCT with patterns in monte-carlo go," INRIA, Dec. 20, 2006.

R. Coulom, "Computing elo ratings of move patterns in the game of go," *ICGA J.*, vol. 30, no. 4, pp. 198–208, Dec. 1, 2007, doi: 10.3233/ICG-2007-30403.

S.-C. Huang, and M. Müller, "Investigating the limits of monte-carlo tree search methods in computer go," in *Comput. Games*, Jul. 12, 2014, pp. 39–48, doi: 10.1007/978-3-319-09165-5_4.

S. Gelly, and D. Silver, "Combining online and offline knowledge in UCT," in *Proc. 24th Int. Conf. Mach. Learn.*, Jun. 20, 2007, pp. 273–280, doi: 10.1145/1273496.1273531.

I. Sutskever, and V. Nair, "Mimicking go experts with convolutional neural networks," in *Artif. Neural Networks - ICANN 2008*, 2008, pp. 101–110.

Chris J. Maddison, Aja Huang, Ilya Sutskever, and David Silver, "Move evaluation in go using deep convolutional neural networks," Dec. 20, 2014.

Christopher Clark, and Amos Storkey, "Teaching deep convolutional neural networks to play go," Dec. 10, 2014.

R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Proc. 12th Int. Conf. Neural Inf. Process. Syst.*, 1999, pp. 1057–1063.

R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," Springer US, pp. 5–32.

C. D. Rosin, "Multi-armed bandits with episode context," *Ann. Math. Artif. Intell.*, vol. 61, no. 3, pp. 203–230, Aug. 26, 2011, doi: 10.1007/s10472-011-9258-6.

I. Antonoglou, J. Schrittwieser, S. Ozair, T. K. Hubert, and D. Silver, "Planning in stochastic environments with a learned model," in *Int. Conf. Learn. Representations*, Jan. 28, 2022.

D. Silver, J. Schrittwieser, et al., "Mastering the game of go without human knowledge," *Nature*, vol. 550, no. 7676, pp. 354–359, Oct. 19, 2017, doi: 10.1038/nature24270.

Aaron van den Oord, Oriol Vinyals, and Koray Kavukcuoglu, "Neural discrete representation learning," Nov. 2, 2017.

Xinlei Chen, and Kaiming He, "Exploring simple siamese representation learning," Nov. 20, 2020.

Tianyu Hua, Wenxiao Wang, et al., "On feature decorrelation in self-supervised learning," May 2, 2021.

Diederik P. Kingma, and Jimmy Ba, "Adam: A method for stochastic optimization," Dec. 22, 2014.

Aaron Defazio, and Samy Jelassi, "Adaptivity without compromise: A momentumized, adaptive, dual averaged gradient method for stochastic optimization," Jan. 26, 2021.

Kyunghyun Cho, Bart van Merrienboer, et al., "Learning phrase representations using RNN encoder-decoder for statistical machine translation," Jun. 3, 2014.

N. R. Sturtevant, "An analysis of UCT in multi-player games," in *Comput. Games*, pp. 37–49.

Jannis Weil, Johannes Czech, Tobias Meuser, and Kristian Kersting, "Know your enemy: Investigating monte-carlo tree search with opponent models in pommerman," May 22, 2023.