The Principle of Value Equivalence for Policy Gradient Search

Das Prinzip der Wertäquivalenz für Strategie-Gradientensuche
Master thesis by Julien Raphael Brosseit
Date of submission: February 14, 2023

1. Review: João Carvalho
2. Review: Prof. Jan Peters
Darmstadt

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Erklärung zur Abschlussarbeit gemäß §22 Abs. 7 und §23 Abs. 7 APB der TU Darmstadt

Hiermit versichere ich, Julien Raphael Brosseit, die vorliegende Masterarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Fall eines Plagiats (§38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung gemäß §23 Abs. 7 APB überein.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Darmstadt, 14. Februar 2023

J. Brosseit

Erklärung zur Abschlussarbeit gemäß §22 Abs. 7 und §23 Abs. 7 APB der TU Darmstadt

Hiermit versichere ich, Julien Raphael Brosseit, die vorliegende Masterarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Fall eines Plagiats (§38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung gemäß §23 Abs. 7 APB überein.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Darmstadt, 14. Februar 2023

Julien Brosseit

J. Brosseit

Abstract

Planning algorithms have shown impressive performance in many domains such as chess and Go. In particular, Monte Carlo Tree Search (MCTS) is used, which builds a search tree toward more promising nodes, while maintaining node statistics to estimate the value of each node. However, these statistics are inaccurate for a small number of visits and the memory requirements for the tree can become unsustainable for large action spaces. A solution to this problem is presented by Policy Gradient Search (PGS). This search method is not based on node statistics but rather learns a simulation policy during the search that guides it toward higher-value nodes. No search tree is needed, which reduces memory requirements. Still, this method suffers from several problems, such as high variance in its value estimates, and is limited to tasks where a perfect simulator of the environment is available. These issues prevent the use of the method in real-world tasks e.g. in logistics or medicine. In this thesis, we address these problems by proposing a new algorithm: *Value Equivalence for Policy Gradient Search* (VE-PGS), which uses a modified search of PGS on a value equivalent (VE) model. VE models are learned models that only focus on those parts of the environment that are relevant to the search. Furthermore, we propose several changes to PGS to address its issues, which are then incorporated into VE-PGS. We test our new method in the two-player board game Hex, while we verify the effectiveness of our changes in an ablation study. Our results show that our method is not only able to plan effectively on a learned VE model but that it is also competitive with state-of-the-art approaches such as MuZero and requires much less memory. These properties make the method suitable for complex tasks with unknown dynamics, where the use of a search tree is infeasible due to high branching factors or memory requirements.

Zusammenfassung

Planungsalgorithmen haben beeindruckende Leistung in vielen Bereichen wie Schach und Go erzielt. Insbesondere wird die Monte-Carlo-Baumsuche (MCTS) verwendet, die einen Suchbaum in Richtung vielversprechenderer Knoten aufbaut und dabei Knotenstatistiken zur Schätzung des Wertes jedes Knotens verwendet. Diese Statistiken sind jedoch bei einer geringen Anzahl von Besuchen ungenau und der Speicherbedarf für den Baum kann bei großen Aktionsräumen untragbar werden. Eine Lösung für dieses Problem bietet die Strategie-Gradientensuche (PGS). Diese Suchmethode basiert nicht auf Knotenstatistiken, sondern erlernt während der Suche eine Simulationsstrategie, die sie zu höherwertigen Knoten führt. Es wird kein Suchbaum benötigt, was den Speicherbedarf reduziert. Diese Methode leidet jedoch unter mehreren Problemen, wie z. B. einer hohen Varianz bei den Wertschätzungen, und ist auf Aufgaben beschränkt, für die ein perfekter Simulator der Umgebung zur Verfügung stehen muss. Diese Probleme verhindern den Einsatz der Methode in der echten Welt, z. B. in der Logistik oder Medizin. In dieser Arbeit gehen wir diese Probleme an, indem wir einen neuen Algorithmus vorschlagen: *Wert Äquivalenz for Strategie-Gradientensuche* (VE-PGS), der eine modifizierte Suche von PGS auf einem wertäquivalenten (VE) Modell verwendet. VE-Modelle sind gelernte Modelle, die sich nur auf die Teile der Umgebung konzentrieren, die für die Suche relevant sind. Darüber hinaus schlagen wir mehrere Änderungen an PGS vor, um dessen Probleme zu lösen, die dann in VE-PGS integriert werden. Wir testen unsere neue Methode in dem Zwei-Spieler-Brettspiel Hex, während wir die Wirksamkeit unserer Änderungen in einer Ablationsstudie überprüfen. Unsere Ergebnisse zeigen, dass unsere Methode nicht nur in der Lage ist, effektiv auf einem gelernten VE-Modell zu planen, sondern dass sie auch mit modernsten Ansätzen wie MuZero konkurrenzfähig ist und viel weniger Speicherplatz benötigt. Aufgrund dieser Eigenschaften eignet sich die Methode für komplexe Aufgaben mit unbekannter Dynamik, bei denen die Verwendung eines Suchbaums aufgrund des hohen Verzweigungsgrades oder des Speicherbedarfs nicht praktikabel ist.

Contents

Figures and Tables

List of Figures

List of Tables

1. Introduction

Monte Carlo Tree Search (MCTS) [17, 48] is a powerful planning algorithm that serves as the basis for many other methods such as AlphaZero [84] and MuZero [80], which achieved great success in various domains, most notably in games such as Go [85], chess [84] and Hex [7]. In these methods, an explicit tree structure is built during the search that stores statistics about all visited states, such as the number of visits and the total sum of returns received. These statistics are then used to estimate the value of each state and guide the search to higher-value nodes. MCTS methods work quite well for problems with moderate action space and small states, such as board games. However, for real-world problems such as robotics or logistics with large branching factors and a potentially unknown and complex environment model, these methods may fail [89]. Since MCTS evaluates nodes like tabular value functions, multiple visits to the same node are required to obtain better estimates. However, for problems with large action spaces and stochastic transitions, as well as time constraints that limit the number of iterations of the algorithm, these estimates become rough and may produce incorrect results. Moreover, as the number of nodes in the search tree grows, so do the memory requirements, making these approaches difficult to use when faced with complex problems and tight memory constraints.

Because these problems are related to having a search tree, research is being conducted to investigate tree-less planning methods. One approach is *Policy Gradient Search* (PGS) [4], which builds upon Monte Carlo Search (MCS) [91]. The action-values of the root node are estimated by sampling random trajectories originating from the node using a simulation policy. The key idea of this method is to improve this policy *during* the search by training it on sampled simulations at each iteration with a policy gradient method such as REINFORCE [95] so that the information that would normally be stored in a tree structure is instead implicitly stored in the policy. Therefore, the simulation policy improves the quality of the sampled trajectories, resulting in a better estimate of the action-values. Nevertheless, PGS suffers from several problems, such as the lack of exploration when updating its policy, which leads to premature convergence. This problem

has been addressed by adding hierarchical entropy regularization [49] or by using node statistics and entropy bonuses [56]. Still, we observe other unaddressed issues, such as high variance in value estimates and policy updates resulting from the use of a vanilla version of REINFORCE with stochastic gradient descent.

One of the most severe limitations currently hindering the use of PGS in real-world problems such as discovering treatment policies in medicine [71] is the need for a model of the environment. Model-based reinforcement learning (MBRL) provides a solution to this problem by first learning the dynamics of the problem, which can then be used for planning. Typically, the model is learned to match the state transitions encountered during sampling, which makes the model very general and usable for other problems in the same environment. But the capacity of model approximators is limited, so they cannot perfectly represent all aspects of the environment. To use these resources more efficiently, the principle of *value equivalence* (VE) [31] suggests considering the future use of the model beforehand and focusing on the aspects of the environment that are important for value-based planning. In addition, the authors of [31] argue that VE serves as the basis for several methods [23, 87, 90, 61] and has contributed to their empirical success, the most popular being MuZero, which combines MCTS-based methods with VE models.

Driven by the success of MuZero and VE, we present in this thesis our method of *Value Equivalence for Policy Gradient Search* (VE-PGS). This approach combines our extended version of PGS with VE models, integrating the simulation policy into the model, so it can make use of a shared state representation. We thus demonstrate the first method based on PGS that does not require knowledge of the dynamics of the environment. Figure 1.1 shows a visualization of the algorithm. Furthermore, using *Expert Iteration* (Exit) [3], we propose a learning procedure for training a policy with VE-PGS *tabula rasa*, i.e. without prior knowledge of the problem. In addition, we propose extensions to PGS to address the high variance problems and improve performance.

We evaluate our methods using the two-player board game Hex, which has a discrete action space and compact states, making it ideal for MCTS-based methods. This fact allows us to compare our methods with existing state-of-the-art approaches such as AlphaZero and MuZero. Our results show that VE-PGS is on par with MuZero in Hex, while our extended version of PGS is competitive with AlphaZero and outperforms its predecessor. Furthermore, our methods have a significantly lower memory consumption.

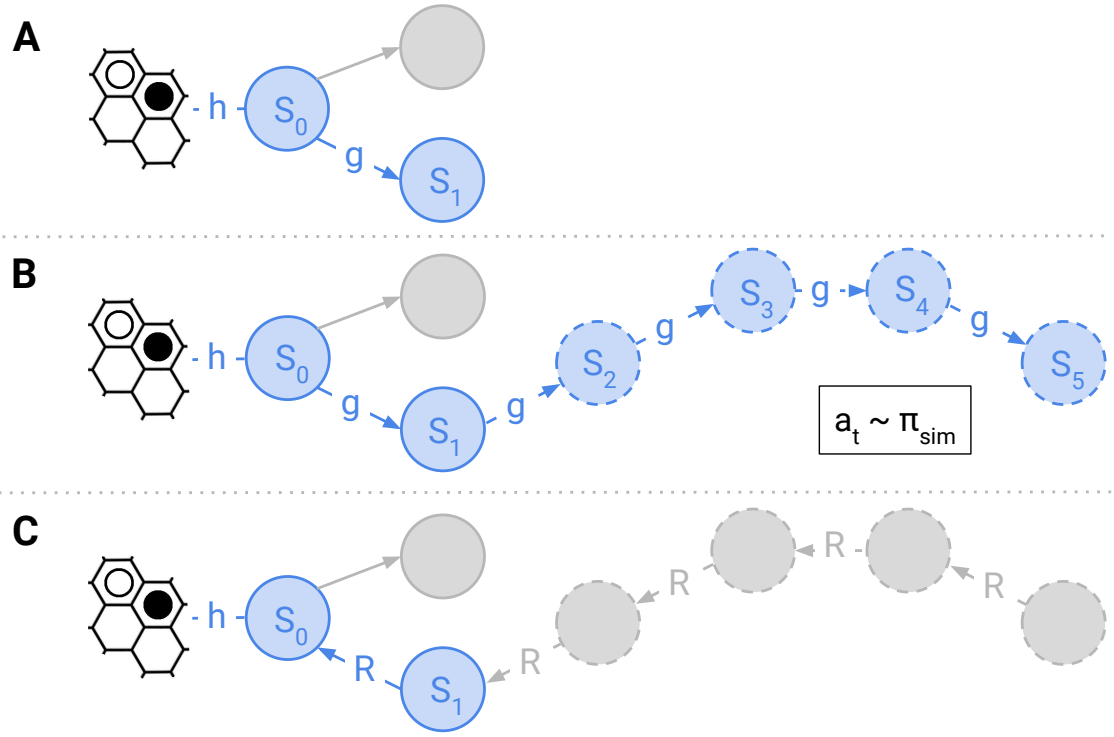

Figure 1.1.: Search with VE-PGS. Visualization based on [80]. **(A)** First, the state of the environment is transformed via a representation network h into a hidden state s_0. The representation of the hidden states is left to the model. It can choose it in a way that is convenient for planning, we only require value equivalence i.e. all rewards collected in the environment correspond to those that would be collected in the real environment. A first action is selected and the next state is computed using the learned dynamics g. These nodes are marked in blue. **(B)** Starting from s_1, a random rollout is simulated with a simulation policy π_{sim}. The learned model g is used for each state transition, which is also shown as nodes marked in blue. The dashed lines indicate that these nodes will not be added to the search tree. After a certain length, the rollout is truncated and the values of all states are determined using the model. **(C)** The result R describing the return is backpropagated up to the root node s_0. All gray dashed nodes are deleted. The collected trajectory is used to update the simulation policy, which uses the same hidden state representation as g. Afterward, a new state is selected and the whole process is repeated.

1.1. Contribution

We contribute to the field of model-based reinforcement learning by presenting the method of Value Equivalence for Policy Gradient Search (VE-PGS), the first algorithm that combines VE models with PGS, removing the need for a perfect simulator. Two aspects that we would like to highlight are: (i) That the underlying version of PGS in our method is more stable and performant than its original, because of the extensions we proposed. (ii) We propose a training scheme for VE-PGS that should be able to train policies without any prior knowledge. Our results show that VE-PGS is a competitive algorithm that can be an alternative to tree-based approaches for certain challenging real-world problems with strong memory constraints. Lastly, we made all of our implementations openly available [1]. This repository also includes our baselines of MCTS, AlphaZero, MuZero, and especially PGS, for which no online implementation exists so far.

1.2. Outline

The rest of the thesis is organized as follows. First, in Chapter 2, we explain the background knowledge necessary to understand the thesis such as basic planning algorithms like MCTS and AlphaZero, as well as policy gradient methods. Then, in Chapter 3, we introduce PGS. After doing so, we examine its weaknesses and propose extensions to address them. The principle of value equivalence is then discussed in Chapter 4, where we look at MuZero and then our main method VE-PGS. The experiments for these methods can be found in Chapter 5, where we describe the environment Hex, the architecture of our policy and VE model, and their training. This description is followed by an ablation study and an evaluation of all the methods. We will discuss similar work on PGS as well as planning methods and VE models in Chapter 6. Finally, in Chapter 7 we will conclude and show possible future work.

[1] `https://github.com/MisterJBro/MasterThesis`

2. Background

In this chapter, we explain the essential concepts needed to understand this thesis. We begin by defining Markov decision processes in Section 2.1, which serve as the basis for modeling sequential decision problems. We then continue with the planning method of MCTS in Section 2.2 and introduce a related method called AlphaZero in Section 2.3, which incorporates domain knowledge into the search. This knowledge can be stored inside neural networks, which we discuss in Section 2.4. Then, in Section 2.5, we look at policy gradient methods that are used in PGS and finally, in Section 2.6, we look at the family of Proximal Policy Optimization methods used to train policies.

2.1. Markov Decision Process

We are interested in solving sequential decision problems, which we describe as a Markov Decision Process (MDP) [70]. We define an MDP as a tuple consisting of the set of states S, the set of actions A, the state transition probability $P(S_{t+1} = s_{t+1}|S_t = s_t, A_t = a_t)$ with random variables S_t and A_t at time t and state $s_t \in S$ and action $a_t \in A$, the reward function $R(s_t, a_t) = r_t \in \mathbb{R}$ with scalar reward r_t, initial state distribution $p_0(S_0 = s_0)$ and discount factor $\gamma \in [0, 1]$. The MDP possesses the Markov property, which means that given the current state s_t the future state s_{t+1} is independent of earlier states

$$P(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, ..., s_0, a_0) = P(s_{t+1}|s_t, a_t).$$

We use a shortened notation, in the following the random variables that are obvious from the context will be omitted. Additionally, the MDP describes a sequential decision process in which an agent, formalized by the probability function $\pi(a_t|s_t)$, selects some action a_t i.e. through sampling at each time step t with the state provided by the environment. We note that the policy does not necessarily have to be represented stochastically, but can

also be expressed deterministically: $a_t = \pi(s_t)$. Still, the representation as a probability distribution is more general and also allows us to describe deterministic policies by assigning all probabilities to a single action.

The first state obtained by the agent is an initial state $s_0 \sim p_0$. After choosing an action, the agent receives the next state s_{t+1} and a reward r_{t+1}, which is a measure of how good or bad a visited state is. This process, in which the agent chooses the next action and in return receives a new state and a reward from the environment, repeats until a terminal state or a certain number of interactions is reached. We can collect all states, rewards, and actions as a sequence $\tau = (s_0, a_0, r_0, s_1, a_1, r_0, ..., s_T, a_T, r_T, s_{T+1})$, which we will call a trajectory of length T. The probability of the trajectory can be calculated via

$$P(\tau) = p_0(s_0) \prod_{t=0}^{T} P(s_{t+1}|s_t, a_t)\pi(a_t|s_t).$$

The goal in solving the MDP is to find a policy π that maximizes the reward achieved across all states. However, instead of maximizing the reward greedily, the policy should maximize the discounted sum of rewards, which we call the return $R_t = \sum_{k=0}^{\infty} \gamma^k r_{k+t}$ [88]. The discount factor γ can be used to define whether one prefers high immediate rewards or high cumulative rewards in the long run. Since we define the state transition and the policy as stochastic, the agent's goal is given as maximizing the expected return

$$\pi^* = \arg\max_{\pi} \mathbb{E}_{\pi}[R],$$

where π^* defines the optimal policy. Instead of searching over all possible functions π, we can restrict ourselves to finding parameters θ for a parameterized policy π_θ that maximizes the expected return. A helpful function to find this policy is given by the value function $V_\pi(s) = \mathbb{E}_\pi[R_t|s_t = s]$. It describes the expected discounted return if actions are selected according to a policy π. In addition to this function, there is also the action-value function $Q_\pi(s, a) = \mathbb{E}_\pi[R_t|s_t = s, a_t = a]$, which is similar but is characterized by first performing an action a and then following policy π. In this way, it can help us identify actions that are better than those proposed by our current policy so that we can update and improve the policy accordingly. An important property that value functions possess is that they can be expressed recursively, which leads to the *Bellman equation*

$$V_\pi(s) = \int_a \pi(a|s) \left(R(s, a) + \gamma \int_{s'} P(s'|s, a)V_\pi(s')ds' \right) da.$$

This formula serves as the basis for many methods for solving MDPs, some of which are the dynamic programming algorithms [70]. The two most common approaches here are policy iteration, in which we iteratively estimate the value function of our policy (policy evaluation) and act greedily with respect to this value function to improve our policy (policy improvement), and that of value iteration. Value iteration attempts to determine the optimal value function that follows the optimal policy π^* first, and extracts the policy only at the very end. Nevertheless, the application of these methods requires knowledge of the state transition probabilities, i.e. a model of the environment. If the dynamics are unknown, Monte Carlo or Temporal Difference methods can be used instead [88].

2.2. Monte Carlo Tree Search

Planning in the field of artificial intelligence (AI) describes the search for a sequence of actions to be performed by an agent for it to achieve its predefined goals. Planning under uncertainty is similar to solving an MDP [13]. However, planning requires a description of the environment, i.e. the MDP, for it to work. One of the most widely used methods for planning is Monte Carlo Tree Search (MCTS) [17, 48], which builds up a tree in the direction of promising nodes during its search to find optimal actions for problems such as sequential decisions problems [14]. It has already been successfully applied to a variety of problems, especially games [7, 16, 44].

The nodes in the search tree represent different states, while the edges connecting them describe actions leading from one state to another. For each node, statistics are kept, such as the number of visits $N(s, a)$ starting from state s with action a selected, and the sum of returns $w(s, a)$, which in games can represent the number of wins for one side. We estimate the returns using random sampling making this approach very general without requiring any domain knowledge. These statistics are then used to estimate action-values by averaging over all sampled returns $Q(s, a) = w(s, a)/N(s, a)$. At the beginning of the search, we always start with only one node in the search tree, the root node s_0.

MCTS is an iterative method consisting of four steps per iteration, see Figure 2.1 for a visualization:

(A) **Selection**: Starting from the root node, we select successive child nodes according to a selection policy until reaching a not fully expanded node, where not all child nodes are expanded.

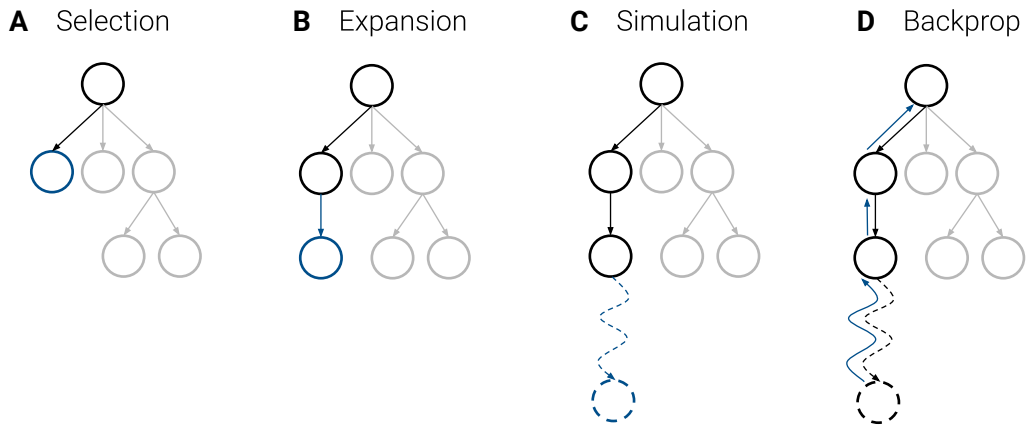

Figure 2.1.: Visualization of a single MCTS iteration. First, the leftmost node is selected. Then, a child node of his is expanded and added to the tree. Starting from the new node, a random rollout is simulated (shown as a blue dotted line), whose result is backpropagated up to the root node (represented by blue arrows).

(B) **Expansion**: Using the node from the previous step, one of its unexpanded child nodes is created and added to the search tree. Which child is expanded can be determined by a rule or randomly.

(C) **Simulation**: In order to evaluate the new node, a complete random rollout is simulated starting from the node. The actions for either side are selected according to some rollout policy. Commonly, the actions are selected uniformly. After reaching a terminal node, the return is determined and saved as a result z.

(D) **Backprop**: The result is backpropagated through the new node and all parent nodes up to the root node. In the process, the sum of returns of each node and their visitation counts are updated $w(s,a) \leftarrow w(s,a) + z; N(s,a) \leftarrow N(s,a) + 1$.

MCTS is an anytime algorithm [89], meaning that it can be stopped at any time, at which point we select the action that has the highest action-value $\arg\max_a Q(s_0, a)$.

One problem that has to be addressed in MCTS arises during the selection phase (A), in which one has to choose a successive node among all child nodes. Here, the typical problem of exploration vs exploitation [88] emerges, where one wants to visit the most promising nodes without missing out on other nodes that could lead to even higher values.

One possible solution is provided by the Upper Confidence Bounds for Trees (UCT) [48] algorithm. For a given state s, we choose the action to the respective child node that maximizes

$$\arg\max_a \frac{w(s,a)}{N(s,a)} + c\sqrt{\frac{\ln(\sum_a N(s,a))}{N(s,a)}},$$

where $c \in \mathbb{R}$ describes the exploration coefficient, which can be viewed as a trade-off between exploration with higher c values and exploitation with lower c values. Using UCT and given an infinite number of iterations and memory in the case of non-stationary reward distributions, this algorithm is guaranteed to converge to the optimal action i.e. the probability of selecting not the optimal action approaches zero [47].

2.3. AlphaZero

With the rise of deep learning methods, variations of MCTS have been developed that incorporate trained policies to improve search efficiency. One of these methods is AlphaZero [84], which is part of a family of methods that also includes AlphaGo [85], AlphaGoZero [86] and MuZero [80]. All methods have in common that they incorporate domain knowledge into their search and have achieved great empirical success, especially in the game of Go [85]. We will focus on AlphaZero, which is the most general of these methods that does not learn a model.

AlphaZero uses the same iteration scheme for its search as MCTS. But unlike MCTS, it makes use of a learned policy π_θ as well as a learned value function V_ϕ with parameters θ and ϕ. The policy is considered as a prior for all actions in the selection phase. Thus, actions are simulated earlier and more often, which are preferred by the policy. More precisely, AlphaZero uses Predictor UCB for Trees (PUCT) [75] in which the child node is selected with the respective action that maximizes

$$\arg\max_a \frac{w(s,a)}{N(s,a)} + \pi_\theta(a|s)c\frac{\sqrt{\sum_a N(s,a))}}{N(s,a)}, \tag{2.1}$$

where $c \in \mathbb{R}$ again describes the exploration coefficient that balances exploration and exploitation.

Another way AlphaZero incorporates domain knowledge into the search is in the simulation phase. Starting from the expanded node, a random rollout is performed where each action is sampled based on a rollout policy, which is often uniform in MCTS. We will briefly refer here again to AlphaGo, which instead uses a fast simulation policy π_{sim} to sample its rollouts. However, instead of simulating the rollout to the end, one can stop earlier, resulting in a truncated rollout where the last state is evaluated using a learned value function. This truncation makes the algorithm faster by requiring less network inference and environment simulation. AlphaZero even goes a step further and truncates its rollout before the first step, effectively estimating only the value of the expanded node. Trained in a self-play manner, AlphaGoZero, which is conceptually very similar to AlphaZero, was able to outperform AlphaGo (Lee) in the board game of Go defeating it 100:0 [86].

It should be noted that AlphaZero describes not only a search method but rather a training scheme associated with the search. The idea is similar to the Expert Iteration (Exit) [3] algorithm, where a student representing the current policy is trained by an expert, which in this case is the search itself. It is an iterative process, where at the beginning the student is a randomly initalized policy that is used in the AlphaZero search method. To improve the student, training data is sampled on the environment with each action being the result of a search. Moreover, to provide more exploration, Dirichlet noise is added to the action prior in the root node. After enough experience is sampled, the student policy and value function are trained using the samples, trying to match the search policy and the returns, respectively

$$\arg\min_{\theta,\phi} L(\theta,\phi) = \arg\min_{\theta,\phi}(v-R)^2 - \pi^T \log p + c_1||\theta||^2 + c_2||\phi||^2.$$

The first term describes the mean squared error (MSE) between the estimated value v and the return of the simulation R. The second term is the cross entropy between the predicted action distribution of the student π and the search distribution p of AlphaZero, while the last two terms serve as L2 regularization of the policy and value function with coefficients c_1 and c_2, respectively. Through this process, the student's predictions become better, which also improves the search and thus in turn provides better policy targets. Therefore, this procedure can be seen as a kind of policy improvement [88]. We note that when we talk about AlphaZero in the following, we refer exclusively to the search and not to the training procedure unless explicitly stated otherwise.

2.4. Neural Networks

In order to use learned policies and value functions, a representation of them is needed. The simplest way to do this representation is to use look-up tables where we store the corresponding value for each state. However, this method only works for problems with small state and action spaces, due to the otherwise high memory requirements and the time needed to compute accurate values for each state [88]. Instead, function approximators can be used, which not only require less memory but also have the ability to generalize. One of the most common approximators are linear functions

$$f(x) = Wx + b,$$

with weight matrix $W \in \mathbb{R}^{n \times m}$, input vector $x \in \mathbb{R}^n$ and bias vector $b \in \mathbb{R}^m$. Because of their convex nature, optimization of linear models can be achieved efficiently, however, their expressiveness is rather limited e.g. the XOR function cannot be learned [54]. *Feed-forward neural networks* tackle this problem by introducing non-linear activation functions. Thus, the linear output is transformed using some non-linear transformation, enabling the network to learn more complex functions. Moreover, these functions are stacked on top of each other, where the output of one function is the input to the next function. We can think of the network as a series of layers with each layer consisting of multiple artificial neurons. The first layer is called the input layer and the last layer is called the output layer, all layers in between are called hidden layers. Neural networks with at least one hidden layer have the property of being universal function approximators that is, any continuous function can be represented arbitrarily accurately with enough neurons [36].

Possible candidates for activation functions include but are not limited to the Sigmoid function $\sigma(x) = 1/(1 + e^{-x})$, Hyperbolic Tangent (Tanh) $T(x) = (e^x - e^{-x})/(e^x + e^{-x})$ and Rectified Linear Units (ReLU) [24] $R(x) = \max(0, x)$. Thus, a two-layer neural network with ReLU activations can be described by

$$f(x) = R(W_2 R(W_1 x + b_1) + b_2),$$

where W_1 and W_2 are the weight matrices and b_1 and b_2 are the bias vectors of both layers, respectively. Networks with a sufficiently high number of layers are referred to as deep, from which the term deep learning (DL) originates. DL can be viewed as a form of representational learning [29] where each layer learns its feature representation

and consecutive layers build upon these representations to build more complex features. These models are trained using backpropagation [76], where a certain criterion has to be minimized e.g. an error between the predictions and the true output values. Alternatively, an objective can also be maximized by minimizing the negative objective. For the objective, the gradient is estimated with respect to the neural network parameters and then updated using a method such as stochastic gradient descent (SGD) [43] that updates the parameters using a single sample in the direction in which the error decreases

$$\theta \leftarrow \theta - \alpha \nabla_\theta L,$$

where θ represents the parameters of the network, α the step size of the update and L describes a loss i.e. our objective. We repeat the process of updating the network until our goal is achieved, e.g. sufficiently minimizing an error on a training set. Over time, more advanced methods to update the parameters have been introduced. One of them is ADAM [45], which incorporates information about past updates in the form of momentum to improve convergence.

A special kind of neural networks that we use in this thesis are *Convolutional Neural Networks* (CNN) [25] that employ a discrete convolution operation in at least one of their layers and tend to work best with imagery data. A discrete 2D convolution can be viewed as moving a filter matrix or kernel over adjacent patches of an image and calculating the dot product of both patches. The output is stored in a 2D data grid, which results in a new image. The parameters of the kernel are learned and are shared for each region of the image in contrast to fully connected networks, where no parameters are shared. So, CNNs have much fewer parameters, resulting in a shorter inference time. This effect is amplified by the fact that the kernel is normally much smaller than the input image, only covering a small area of the image at a time, which has some similarities to the concept of receptive fields from biology [26]. So, the kernel will focus on learning local features, which in turn can be used as input for further convolutional layers to build features that span a wider area of the input.

2.5. Policy Gradient

One way of training differentiable and parametrized policies such as neural networks in an RL setting are policy gradient methods [88]. Given a policy π_θ with parameters θ, the objective is to maximize the expected return $J(\theta) = \mathbb{E}_{\pi_\theta}[R]$. If we can estimate the

gradient $\nabla_\theta J(\pi_\theta)$ with respect to our parameters, we can update the parameters with gradient ascent

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\pi_\theta),$$

where α denotes the learning rate. There is more than one way to estimate this gradient such as finite difference methods, where we slightly change the parameters of the policy, estimate the expected return, and finally estimate the gradient via the differences. However, one of the more commonly used methods is using likelihood ratio methods leading to the algorithm of REINFORCE [95]. In the following, we will derive this estimator. Given an MDP and some parametrized policy π_θ, for each trajectory τ, its probability can be calculated as

$$P(\tau|\theta) = p_0(s_0) \prod_{t=0}^{T} P(s_{t+1}|s_t, a_t)\pi_\theta(a_t|s_t).$$

The expected return of the policy can be calculated as the integral of the return of all trajectories weighted by their probability

$$\mathbb{E}_{\pi_\theta}[R] = \int P(\tau|\theta)R(\tau)d\tau.$$

We define the return of a trajectory to be $R(\tau) = \sum_{k=0}^{T} \gamma^k r_{k+t}$. In order to estimate the gradient $\nabla_\theta \mathbb{E}_{\pi_\theta}[R]$, we use the likelihood ratio trick:

$$\int \nabla_\theta P(\tau|\theta)R(\tau)d\tau = \int P(\tau|\theta)\nabla_\theta \log P(\tau|\theta)R(\tau)d\tau.$$

Thus, we can obtain the policy gradient via

$$\begin{aligned}\nabla_\theta J(\theta) = \nabla_\theta \mathbb{E}_{\pi_\theta}[R] &= \nabla_\theta \int P(\tau|\theta)R(\tau)d\tau = \int \nabla_\theta P(\tau|\theta)R(\tau)d\tau \\ &= \int P(\tau|\theta)\nabla_\theta \log P(\tau|\theta)R(\tau)d\tau = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log P(\tau|\theta)R(\tau)] \\ &= \mathbb{E}_{\pi_\theta}[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)R(\tau)].\end{aligned}$$

The expectation can be estimated using a sample mean. However, policy gradient methods suffer from high variance [52, 68], which can lead to slow convergence. One idea proposed to reduce variance is to subtract a constant baseline from the gradient:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t)(R(\tau) - b)].$$

This baseline does not introduce any bias, since $\mathbb{E}_{\pi_\theta}[\nabla_\theta \log P_\theta(\tau)] = 0$. So as the number of samples rises, the influence of the baselines diminishes going toward zero. One common candidate for the baseline is the value function approximation of the current state $V_\pi(s)$. If we view the return as an approximation of the action-value function Q_π, we obtain an estimate of the advantage function $A_\pi(s,a) = Q_\pi(s,a) - V_\pi(s)$ in our policy gradient estimate.

2.6. Proximal Policy Optimization

Proximal Policy Optimization (PPO) [82] algorithms define a family of policy gradient methods, which try to keep their updates close to the previous policies, in the style of trust-region methods [67]. Instead of maximizing the expected return directly, they optimize a surrogate objective, where data samples are reused to improve sample efficiency.

There are two main methods of PPO algorithms:

1. **PPO Penalty**: The KL divergence between the updated and old policy is included as a penalty term that is scaled dynamically during the training.

2. **PPO Clip**: The objective function is clipped, which removes the motivation for the policy to deviate too much from its original parameters without the need of using constraints.

The more common method is PPO Clip where the objective is given by

$$L^{\text{CLIP}}(\theta) = \mathbb{E}_t[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t],$$

with probability ratios $r_t(\theta) = \pi_\theta(a|s)/\pi_{\theta_{\text{old}}}(a|s)$, $\hat{A}_t$ being the advantage estimation using Generalized Advantage Estimation (GAE) [81] and clipping hyperparameter ϵ. The idea is that we have two objectives, one being the clipped objective where the probability ratios are clipped between $[1-\epsilon, 1+\epsilon]$, so a stronger deviation will not change the objective. The other one is the unclipped objective, which also Trust Region Policy Optimization (TRPO) [83] uses. Both are combined by using the minimum, which lets the probability ratios be smaller than $1-\epsilon$ for positive advantages and bigger than $1+\epsilon$ for negative ones.

From now on, we will refer to PPO-Clip as PPO, as it is usually done since it is the more common method. PPO is a competitive method and is considered state-of-the-art. One of its key benefits is the simplicity of its implementation. Pseudocode is given in Algorithm 1. It has to be noted that GAE computes advantages using a learned value function, which is learned concurrently with the policy.

Algorithm 1: PPO-Clip with GAE

Initialize π_θ and V_ϕ
for $k = 1, 2, ...$ **do**
  Sample a set of trajectories $\mathcal{D}$ using π_θ
  Estimate advantages $\hat{A}_t$ e.g. via GAE
  Optimize π_θ on L^{CLIP} via gradient ascent on minibatches $\sim \mathcal{D}$
  Fit V_ϕ via gradient descent on MSE
end

3. Policy Gradient Search

In this chapter, we look at the Policy Gradient Search (PGS) planning method. We start with a definition of the method in Section 3.1, after which we deal with its problems in Section 3.2, such as high variance in its estimates and missing exploration, proposing extensions to address them.

3.1. Definition

Policy Gradient Search (PGS) [4] describes a search algorithm that in contrast to MCTS does not build a tree structure to store search statistics, but rather trains a simulation policy π_{sim} during the search, which guides it towards higher valued nodes. The motivation behind this idea is twofold. Firstly, MCTS stores statistics about the number of visitations and the sum of returns within its nodes, which are then used to estimate action-values. However, if a node is rarely visited, which can be the case for high dimensional problems or low iteration counts, the estimations become rough and inaccurate [4]. PGS circumvents this problem by not storing any node statistics (with one exception) and instead relies on the simulation policy to learn statistics and even generalize to unseen nodes. Secondly, storing nodes can be very resource intensive based on the environment and problem. Since PGS is without a tree structure, it can be applied to problems that have environments with high memory requirements.

The algorithm builds upon Monte Carlo Search (MCS) [91], which is similar to MCTS but without expanding the search tree. MCS tries to estimate $Q_\pi(s_0, a)$ for some state s_0 given a policy π by repeated sampling starting from the initial state s_0. This sampling process is the same as the simulation phase of MCTS, so truncation of the rollouts is also a possibility. These rollouts can then be averaged to get a value estimation for each possible action. However, sampling the first action completely by chance can be wasteful, since ultimately we are more interested in higher-valued actions. Therefore, for discrete action spaces, one

can employ bandit algorithms and in the case of the availability of a trained policy, PUCT to decide on the first action to pick in each iteration. In the discrete case, MCS can be seen as MCTS but without any expansion phase and a shallow tree with only one parent node and several child nodes. Since MCTS can exploit the knowledge gained through its tree and consequently direct its search on more promising nodes, MCS is outperformed by MCTS [4].

During the search itself, we encounter new information, which can be used to improve the search further. PGS tries to incorporate this fact by training the simulation policy during the search with sampled trajectories. It should be noted that the method requires a trained policy π_ϕ with parameters ϕ and value function V, which in our case will be represented by deep neural networks. However, before each search, the policy or rather the parameters are copied from π_ϕ to π_{sim} whose parameters we denote with θ. Therefore, training of π_{sim} does not change the original parameters ϕ. After sampling a trajectory $\tau = (s_0, a_0, r_0, ..., s_T, a_T, r_T)$ in the simulation phase, we update the simulation policy according to REINFORCE

$$\theta \leftarrow \theta + \alpha V(s_T) \sum_{t=1}^{T} \nabla_\theta \log \pi_{\text{sim}}(a_t|s_t), \tag{3.1}$$

with learning rate α. Noticeably, neither returns nor rewards are used within the update and the value of the last state is bootstrapped. The reason for both lies in the original work, where the board game Hex is analyzed. It has a sparse reward setting only returning one for the winning move at the end of the game. Returns would not be of any help in this setting, especially since the rollouts are truncated. So, there is a high chance that the rollouts would not end with a terminal state, hence all rewards would be zero. As discussed in the background chapter, truncation is primarily done because of efficiency concerns, choosing the correct cut-off, however, imposes severe difficulties. Moreover, this algorithm no longer estimates the action-value function $Q(s_0, a)$ for the original policy π_ϕ, since the simulation policy deviates from it during the search. The action-value function is with respect to π_{sim}.

However, training the simulation policy during each iteration of the search creates a problem. Depending on how large the policy is in terms of parameters, the longer it will take to train it, which harms the speed of the search in general. Furthermore, since the number of samples is minimal and the trajectory length is small, it can very quickly lead to overfitting, the policy loses the ability to generalize and will give poor predictions for unseen states. A solution for both problems is to train only the last layer instead of

the whole network. So, the remaining parameters are frozen and not changed during training. The time of a forward pass of the data through the network remains the same, but for the backward pass that calculates the gradients with respect to the parameters, the time is dramatically reduced, since this estimation only has to be done for the last layer. Overfitting is also counteracted since the last layer only has a small capacity relative to the entire network. Pseudocode for PGS using PUCT in discrete action spaces is given in Algorithm 2.

Algorithm 2: Discrete-case PGS

Input: Search state s_0
Output: $\pi(a|s_0)$
Initialize π_{sim} as copy of π_ϕ, freeze all but the last layer.
Set root node n_0, get action prior $\boldsymbol{\pi}_0$ and value v_0 using π_ϕ and $V_{\phi'}$.
for $k = 1, 2, ...$ **do**
  if n_0 *is not fully expanded* **then**
    Expand unexpanded child node $n_k = \arg\max_c \boldsymbol{\pi}_0$.
  else
    Select child node using Equation (2.1).
  end
  Collect rollout τ starting from n_k by sampling from π_{sim} for T steps.
  Update π_{sim} via Equation (3.1) on $\tau = (s_0, a_0, r_0, ..., s_T, a_T, r_T)$.
  Backpropagate $V(s_T)$ up to n_0.
end
Estimate $\pi(a|s_0)$.

Since action-values can theoretically be estimated only by sampling, PGS can work in both nondeterministic and continuous environments.

3.2. Extensions

In addition to the strengths of PGS, such as the generalization capabilities of the simulation policy and the low memory consumption, there are also some disadvantages that limit its use. These problems come mainly from the vanilla REINFORCE update and the sampling process. In our tests, we found the following problems, for which we also directly propose changes:

(A) **High variance of estimate**: During the simulation phase, a random trajectory is sampled using the simulation policy and the result is backpropagated. However, due to poor sampling, the estimation itself may be rough, leading to over- or underestimation of the true value of a node, so that the PUCT algorithm selects the node incorrectly. Especially if the number of iterations is small, it can lead to poor predictions.

The variance is tightly bound to the length of the truncated trajectory, so an extension is to dynamically adjust the length of the trajectory. We start with trajectories of small length as they have low variance and scale them in length as the search progresses. How this change affects learning performance needs to be investigated as the trajectories sampled are used directly for training. In [4] a similar extension is used by sampling until a unique trajectory is created. However, this method has the disadvantage that it requires information about all trajectories sampled so far and the condition is difficult to keep when parallelizing the algorithm. Therefore, we propose to tie the length of the trajectory to the number of visits of the first action and require at maximum, under the assumption of a constant branching factor, all trajectories according to the branching factor to be sampled. For example, with a branching factor of $b = 3$, we require the first three trajectories to have a length of one and the trajectories up to the ninth to have a length of two, etc. The length T, at which the trajectory is truncated, can thus be calculated via

$$T = \left\lfloor \frac{\ln(N(s_0, a))}{\ln(b)} \right\rfloor + 1. \tag{3.2}$$

Instead of dynamically adjusting the length, we can let the length of the trajectory be fixed and adjust the return that is backpropagated. The motivation behind this idea is that trajectories of small length, e.g. length of one, do not provide enough information for updating the simulation policy properly. By sampling longer trajectories with a fixed length, we ensure enough information for our updates, but can artificially recreate the variance-reducing effects of dynamic length by using a weighting procedure that we will present in the following.

The return of the trajectory can be calculated via the rewards or in a sparse reward environment by using value estimates. In this work, we focus on value estimates. In the original PGS, the REINFORCE update only uses the value of the last observed state. We instead infer the value of each state in the trajectory, which can be often done with little extra cost, under the assumption of a shared policy and value network. For these value estimates $V(s_0), V(s_1), ..., V(s_T)$ we use a weighted sum

$$R_{\text{Backprop}} = \sum_{k=0}^{T} V(s_k) \frac{p^k}{-k \ln(1-p)}, \tag{3.3}$$

where $p \in (0, 1)$ describes a parameter to influence the weights. The idea behind this equation is that each value is weighted by a discrete logarithmic distribution that has several beneficial characteristics for our use case. The sum over the distribution is always one. Furthermore, the first value estimate has the lowest variance, and thus receives the highest weight, and this weight is lowered for subsequent values based on p. In the case of a $p \to 0$, only the first value is used for estimation. Especially at the beginning of the search, this value represents a good first-value estimate of the state. For $p \to 1$, we instead obtain a uniform distribution that gives equal weight to each value.

(B) **Missing exploration**: The problem of missing exploration in PGS due to the vanilla REINFORCE update has already been noticed by other works [49, 56], which have proposed more complex mechanisms to fix it like hierarchical entropy regularization or by using node statistics and entropy bonuses. Hierarchical entropy regularization is mainly used to counteract an early commitment problem, which the authors identified in their tasks. In this problem, earlier actions are selected with significantly lower entropy than later actions. It should be noted that this solution is task-specific as not every task suffers from early commitment. In the work of [56], entropy bonuses were assigned based on node statistics. Nevertheless, this change requires the introduction of an expanding tree structure, leading to problems such as high memory consumption. We give a simpler and more general solution to improve exploration by adding a single entropy bonus $H(\pi)$ to the update of the simulation policy. However, since the simulation policy is based on a trained policy, more entropy could lead to degraded performance. Assuming that our trained policy is already sufficiently good, we can add a Kullback-Leibler (KL) penalty term $\text{KL}(\pi_{\text{sim}} || \pi_\phi)$ instead, where π_ϕ denotes the original policy. This term keeps the simulation policy close to π_ϕ to prevent drops in performance.

(C) **Normalized visit counts**: After the search is complete, a new policy with respect to the search state is created based on the normalized number of visits

$$\pi(a|s_0) = N(s_0, a)^{1/\tau} / \sum_b N(s_0, b)^{1/\tau},$$

with the root state s_0 from which we started the search, the temperature τ, which we choose $\tau = 1$ in our experiments, and the node visit function $N(s, a)$. This formula was also used in [84]. Because it relies solely on visit statistics, large discrete action space and low visit counts can lead to rough estimates that are close to a uniform action distribution. This problem was also noted by [18] for AlphaZero, whereupon they proposed a different policy target

$$\pi(a|s_0) = \text{softmax}(\text{logits}(s_0) + \sigma(Q(s_o, a))). \tag{3.4}$$

This formula uses the logits from the previous policy summed with the action-value estimate from our search and a monotonic function σ, which we choose as a linear function in our experiments. It does not rely on visit statistics and does not have the same problems as normalized visit counts. Therefore, we consider it also for our variant of PGS.

(D) **Policy update**: We have already mentioned that the update of PGS using the last value estimate exhibits high variance. Instead of using only the last value, we can replace it with some other estimates such as the advantage function, which can be determined with GAE [81]. However, for sparse environments with possibly lots of zero rewards and truncated trajectories, GAE, which uses n-steps returns, might not work. Since we can estimate each value for each observed state $V(s_0), V(s_1), ..., V(s_T)$, we can use these individual estimates to update the policy and introduce an average baseline to further reduce the variance as discussed in Section 2.5

$$\theta \leftarrow \theta + \alpha \sum_{t=1}^{T} \nabla_\theta \log \pi_{\text{sim}}(a_t|s_t) \left(V(s_t) - \frac{1}{T} \sum_{k=1}^{T} V(s_k) \right). \tag{3.5}$$

Furthermore to optimize the policy parameters SGD is used, for which there are already better alternatives like ADAM. It introduces momentum and adaptive learning rates and has performed better in some tests than SGD [45].

This list is by no means exhaustive. The limitation of needing a simulator of the environment will be addressed in Chapter 4. Furthermore, all of the above extensions can be combined to obtain individual benefits. However, it must be mentioned that certain extensions may influence each other, such as in (A) the use of a dynamic length in the rollouts and the weighting of the backpropagated values, i.e. for shorter rollouts the weighting will be less effective.

4. Value Equivalence for PGS

In this chapter, we present our main method, a combination of VE models and PGS. The motivation for this idea comes from the fact that planning requires a model. However, a model is not always available, so we need to learn it instead. A paradigm for learning a model is proposed using the principle of value equivalence, which we will first address in Section 4.1. This principle forms the basis for our baseline MuZero, which we will introduce in Section 4.2, and finally, we present our method Value Equivalence for Policy Gradient Search (VE-PGS) in Section 4.3.

4.1. Principle of Value Equivalence

Note that the notation for formulas in this chapter relies heavily on [31]. Given a problem defined as MDP, we are mainly interested in two things when learning a model: the state transition probability function $P(s|s,a)$, also called the dynamics, and the reward function $r(s,a)$. We define the tuple $m = (P, r)$ to be the true model of our problem. Because there are tasks in which these quantities are not known [71] and a model is necessary for planning with PGS, our goal is to learn an approximate model $\tilde{m} = (\tilde{P}, \tilde{r})$ such that $\tilde{P} \approx P$ and $\tilde{r} \approx r$. Learning a model is one of the main concerns of model-based RL [58] and is also called system identification [8] in control theory. We note that model learning has another advantage as it can increase sampling efficiency, since sampling in the real world can be expensive or difficult e.g. in robotics [46].

A straightforward way to learn the model is to collect data such as rewards, actions, and states using the true model by sampling from it. Then we train the approximated model by minimizing a loss, e.g. the MSE for rewards, and for the state transition function we can use the maximum likelihood estimate

$$\mathbb{E}_{(s,a)\sim\mathcal{D}}[\mathrm{KL}(P(\cdot|s,a)||\tilde{P}(\cdot|s,a))],$$

where $\mathcal{D}$ describes the state-action distribution that can be approximated using a data set and KL describes the Kullback-Leibler divergence. There are other methods to learn an approximation $\tilde{P}$, but most commonly they learn a forward model to correctly predict the next states [58]. Although learning P has many advantages since this model is very general and can be used for other problems with the same dynamics, it is a difficult problem since inaccuracies can propagate during the search making long sampled trajectories unusable [42]. So, our approximation should be as accurate as possible to be useful for planning. However, in terms of planning, we are not interested in the predicted next states themselves, but only in the rewards and values that result from the states. The states may even model properties that are not relevant for accurate value estimation, which makes learning a model more difficult, especially since approximators often have limited capacity [58]. The idea of *value equivalence* (VE) tries to formalize this idea, by trying to learn only the parts of the environments that are relevant for value-based planning making the learning process simpler [31].

To describe VE in more detail, we first recall the value function V_π, which is defined as the expected return obtained by following a policy π. To compute the value function, we can use the Bellman operator

$$\mathcal{T}_\pi[V](s) = \mathbb{E}_{a\sim\pi(\cdot|s),s'\sim P(\cdot|s,a)}[r(s,a) + \gamma V(s')],$$

where $\gamma \in [0, 1]$ defines the discount factor and V is a function that maps states to real values and will later serve as a value function approximation. The operator estimates the expected one-step return by using the current reward and then adding it to the value estimate of the next state, which is called bootstrapping. By repeatedly applying this Bellman operator, the function V eventually converges to the real value function $\lim_{n\to\infty}(\mathcal{T}_\pi)^n V = V_\pi$. To compute this operator, we need to know a model m, and conversely, any model m will induce a Bellman operator. We can now define the principle of VE. Given a set of possible policies Π and a set of possible functions $\mathbb{V}$ that map states to real values, two models m and $\tilde{m}$ are VE if their induced Bellman operator is equal

$$\mathcal{T}_\pi = \tilde{\mathcal{T}}_\pi,$$

for all $\pi \in \Pi$ and all $V \in \mathbb{V}$. Definition from [31].

Thus, to learn a planning model, we should use a loss that minimizes the error between the Bellman operators. Still, in practice, we do not have access to the true operator $\mathcal{T}_\pi$, so instead, we can approximate the expected one-step return using random sampling.

The advantage of this principle is that we no longer need to model the state transitions correctly, but can model them arbitrarily as long as the estimated rewards and values are still correct. We can do this modeling only because the model will be used later for planning, where the values are of importance. But we also gain another advantage, since our function approximators are often limited in capacity, e.g. neural networks, modeling fewer aspects of the environment leads to better approximations for the same capacity. In [31], they show that models that were limited in capacity were able to achieve better accuracy on the values of the environment by using the principle of VE than using, for example, maximum likelihood estimation. There were even cases, where the capability to model an environment correctly in terms of its values was only given if VE was used.

4.2. MuZero

MuZero [80] is an algorithm that performs a search similar to AlphaZero, but applied to a trained model that uses the principle of VE. It achieved top performance in several benchmarks such as the Atari 2600 benchmark [11] or the board games chess, Shogi, and Go. Since the algorithm of AlphaZero has already been explained in Section 2.3, we take a closer look at the model of MuZero and why this model can be defined as VE.

The MuZero model consists of three parts: representation, dynamics, and prediction. Starting from a state s_0, we normally begin our search by using our state transition function to derive the next states s_{t+1}. However, MuZero starts by changing the representation of the state to a so-called hidden state, which we denote by $\tilde{s}_t$ for the hidden state at time step t. The key idea of these hidden states is that we do not choose their representation, but rather let the model itself decide what they should represent. So instead of the current state e.g. a game, where the state is defined as a board with the pieces on it, the hidden state can represent anything and can even be much smaller in size than the real state leading to lower memory requirements and faster computations. More specifically, the model should choose a representation for the hidden states that emphasizes only those aspects of the environment that are necessary for learning the correct values and rewards, and ignores other aspects, according to the principle of VE. This transformation from state to hidden state is performed using the representation function $\tilde{s}_0 = h_\theta(s_0)$ with learnable parameters θ. Then, the hidden state and the next action are used as input to the dynamic function $r_{t+1}, \tilde{s}_{t+1} = g_\theta(\tilde{s}_t, a_t)$ to compute the next hidden state while obtaining the current reward prediction of the model. Both functions are already sufficient for planning, starting with the calculation of the hidden state $\tilde{s}_0$ using the representation

function, after which we can simulate trajectories using the dynamics function where we can collect the corresponding rewards. However, MuZero has another component that estimates two more quantities: the prediction function $\mathbf{p}_t, v_t = f_\theta(\tilde{s}_t)$, with policy estimate $\mathbf{p}_t$ and value estimate v_t. Both properties are useful in planning. Thus, when an action is selected according to PUCT, the prior action distribution $\mathbf{p}_t$ can be used, and during the simulation phase, we use the value estimate v_t, which is then backpropagated. Otherwise, the search is the same as in MCTS.

The MuZero algorithm does not only plan actions but rather describes a complete training procedure similar to that of AlphaZero. Starting from a randomly initialized model, trajectories are sampled in each iteration in an environment by following a policy that is computed via model search. These trajectories are stored in a replay buffer. Afterward, random samples are drawn from the buffer to train the model. More precisely, the model minimizes a loss on the drawn trajectories

$$\arg\min_\theta L_t(\theta) = \arg\min_\theta \sum_{k=0}^{K} l^r(\tilde{r}_{t+k}, r_{t+k}) + l^v(\tilde{v}_{t+k}, v_{t+k}) + l^p(\mathbf{p}_{t+k}, \tilde{\mathbf{p}}_{t+k}) + c||\theta||^2, \quad (4.1)$$

with rollout length K, model parameters θ, L2 regularization coefficient c, and loss functions l^r, l^v, and l^p for reward, value, and policy, respectively. r, v, and $\mathbf{p}$ describe the observed reward, return, and action distributions during the sampling process and $\tilde{r}$, $\tilde{v}$, and $\tilde{\mathbf{p}}$ describe the predicted ones. The model is unrolled K steps into the future at each time step t. Starting from the same state, our model is also unrolled by the same number of steps into the future, and we aim to match the two sequences by minimizing the loss functions. These individual loss functions can be described by the MSE, the cross entropy, or any other appropriate function.

Next, we analyze how this model and training scheme can be viewed from a VE perspective. We define the loss for VE between our true model m and our learned model $\tilde{m}$ from MuZero as

$$L^K(m, \tilde{m}) = \sum_{\pi \in \Pi} \sum_{V \in \mathbb{V}} ||\mathcal{T}_\pi^K V - \tilde{\mathcal{T}}_\pi^K V||,$$

where $\mathbb{V}$ and Π are defined as in the previous Section 4.1 and K is a positive integer indicating how many times the Bellman operator is applied to the value approximation V. Note that this definition describes proper value equivalence [30], which can be viewed as

a generalization of VE. We will not discuss proper VE in detail, but it should be noted that models trained with it are still subject to the positive effects of VE.

We consider a particular case of MuZero's loss where l^r and l^v are defined as MSE loss and we omit the policy loss and L2 regularization since they are not important for the analysis. We call this instance of the loss $\tilde{L}_t$, for which the following holds

$$\begin{aligned} ab\mathbb{E}_{d_\pi}[\tilde{L}_t] &\geq L^K(m, \tilde{m}) \\ a &= \gamma^K(g + \gamma^n)(1 - \gamma^n)^{-1} \\ b &= a + K + 2, \end{aligned}$$

with d_π as a stationary distribution, $g > 0$ as any real number, and the discount factor γ. For the actual proof, we refer to the paper [30]. This formula states that the MuZero loss function can be seen as an upper bound on the loss for (proper) VE, and so its minimization also minimizes the error between the Bellman operators. Therefore, the MuZero model can be viewed as an application of the principle of VE making it a VE model.

4.3. Method

Combining PGS with VE models is straightforward. We use our extended PGS version as a basis, a combination of all extensions that we described in Section 3.2, and instead of planning on the real model, we use MuZero's VE model to perform our hidden state transitions and estimate the values. Furthermore, the simulation policy is not represented by an independent policy network but is described by the model's prediction component f_θ instead. It models the simulation policy, estimates the value function, and uses the model's hidden state representation as input, which encodes useful features about the state for value-based estimations. To improve the speed of the search, everything except the linear head in the prediction function responsible for computing the simulation policy is frozen, similar to what happens in PGS. We call the resulting method *Value Equivalence for Policy Gradient Search* (VE-PGS). See Figure 4.1 for a visualization of the model. We note that our method does not need knowledge about the true dynamics model nor does it need an expanding search tree leading to less memory consumption.

As there are differences between AlphaZero and MuZero, we point out some differences between PGS and VE-PGS that result from using the VE model:

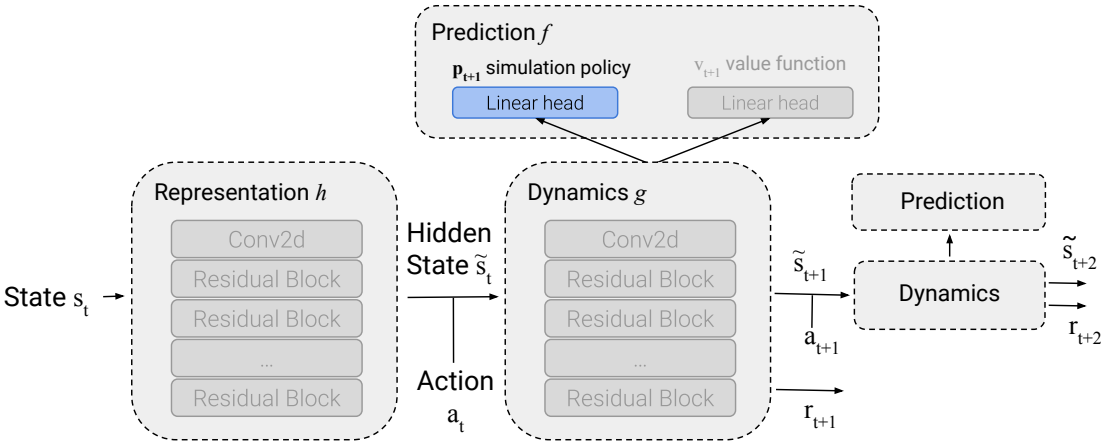

Figure 4.1.: Model for VE-PGS. Like MuZero's model, it consists of three components. First, the current state s_t is used as input to the representation function (left), which generates a hidden state with arbitrary representation. The hidden state and action are then used as input to the dynamics function, which computes the next hidden state. In addition, the hidden state can be used in the prediction function to estimate the simulation policy (blue) and value function. During the search with VE-PGS, we sample the next actions with respect to the simulation policy. All hidden states and value estimates are stored within a trajectory to update this policy. Furthermore, to speed up the search, all parts of the model except the linear head used to calculate the simulation policy are frozen in their parameters.

1. In PGS, each node in the shallow tree is represented by a state, whereas in VE-PGS, each node except the root node is instead represented by a hidden state.

2. The model does not learn which actions are legal and which are not. In environments that introduce state-dependent invalid actions, the simulation policy might sample actions during the rollouts that are not legal. In this case, the model is expected to predict a reward and a value of zero to discourage such behavior. As an exception, the root node knows all valid actions related to the search state to create a reasonable action distribution and exclude invalid first actions. We can usually get this information directly from the real environment.

3. The terminal states are also not modeled, which can lead to states being reached during rollouts that would pass over terminal states. In this case, the model is expected to return a reward of zero and estimate the same terminal value for all subsequent actions. To achieve this behavior, we add random action sequences at

the end of each sampled sequence, with zero rewards and the last value as the target value.

Like AlphaZero and MuZero, and in the spirit of Expert Iteration [3], we present a description of our algorithm for training a policy *tabula rasa*, i.e., without any prior knowledge, using our search algorithm VE-PGS. The training algorithm, which we call VE-PGS Exit, is described in Algorithm 3. The idea is simple, we sample trajectories with respect to our search method VE-PGS that are then used in a supervised learning fashion to train the VE model.

Algorithm 3: VE-PGS with Expert Iteration

Initialize model $m = (f_\theta, g_\theta, h_\theta)$.
for $i = 1, 2, ...$ **do**
  for $k = 1, 2, ...$ **do**
    Sample trajectory τ_k by sampling actions via the search of VE-PGS.
    Add τ_k to dataset $\mathcal{D}$.
  end
  Train m via Equation (4.1) on $\mathcal{D}$.
end

Note that this algorithm is more theoretical at this time, as the computational requirements for actually training a model are extremely high, and due to the environment we chose for benchmarking, we were not able to train the model using this algorithm, but resorted to another method.

5. Experiments

In this chapter, we evaluate our extended version of PGS and VE-PGS in the board game Hex, a competitive multi-agent environment with sparse rewards that we present in Section 5.1. Then, we explain self-play (SP) in Section 5.2, which serves as the basis for training agents in this environment. The architecture of the policy and the model is explained in more detail in Section 5.3. Afterward, we move on to the training of these networks in Section 5.4 and evaluate our experiments in Section 5.5. Finally, we conclude this chapter with a discussion of our results in Section 5.6.

5.1. Hex

Hex [28] is a two-player turn-based board game, invented by Piet Hein in 1942 and later separately by John Nash in 1948. It is played on a rhombus-shaped board made out of hexagonal squares on which each player can place a stone of his color unless the square is already occupied. At the beginning of the game, each player is given a color, either black or white, with the black player starting first. To compensate for the advantage of going first, one can employ the swap rule, where the second player can decide after the first move whether the players should swap colors or not. In this thesis, the swap rule is not used. A move is made by placing a stone on an unoccupied square, after which the other player makes his move. The objective of the game is to connect the opposite edges of the board with the player's respective color using a path of stones. A visualization of a Hex game can be seen in Figure 5.1.

The game is subject to deep theoretical analysis and is proven to not end in draw [27]. On symmetrical boards the first player has a winning strategy [28] that is explicitly described for 7×7 [32] and 9×9 [66] boards. However, Hex is PSPACE-complete [73], which has the consequence that for larger Hex boards polynomially more memory is needed to solve the problem, requiring also significantly more time. Thus, for larger board configurations,

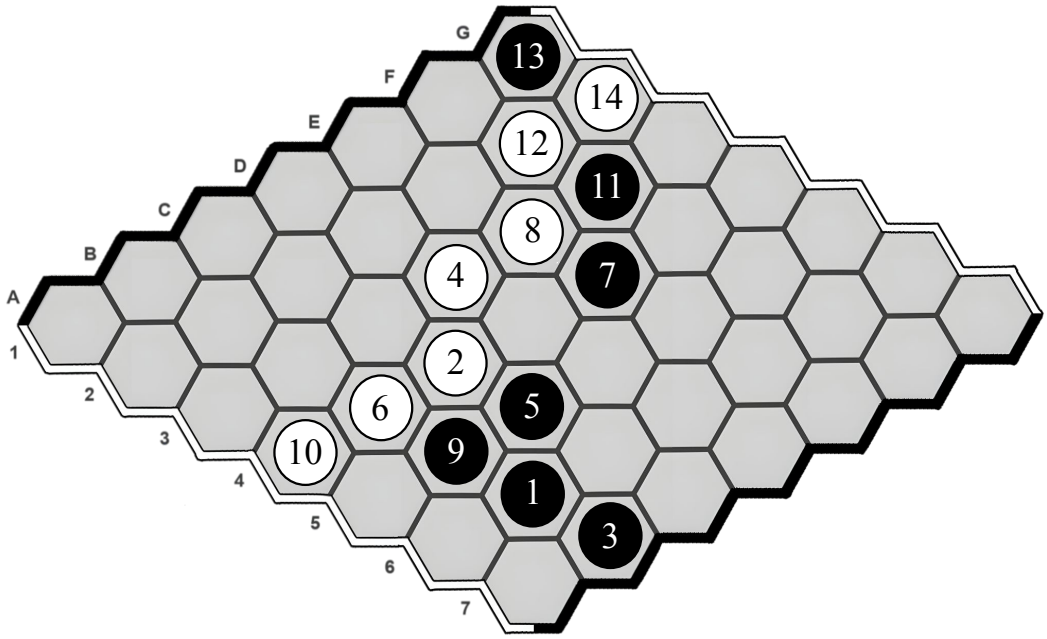

Figure 5.1.: Example of a game of Hex on a 7×7 board. The numbers on the stones correspond to the turn each move was made. After the 14th move, white won the game because a connection has been made between the white edges.

uninformed planning methods are not practical, and it makes sense to resort to methods that use learned knowledge to speed up the search.

Still, it should be noted that this environment is not particularly favorable to the strengths of PGS. To the contrary, MCTS-based approaches dominate this game and can be found in all state-of-the-art implementations since 2009 [4]. Its action space is discrete and moderate, it can be computed quickly and its state description is very compact, allowing for very large search trees. Nevertheless, we chose this setting to be able to compare PGS and VE-PGS with state-of-the-art MCTS-based methods such as AlphaZero and MuZero, which do not work directly on continuous or stochastic environments without any modifications [57, 5]. Implementing such variants are beyond the time scope of this work and is reserved for future work (see Section 7.1). In addition, the original version of PGS was tested on Hex [4], so the algorithm was designed with sparse environments in mind e.g. the update uses value estimates instead of returns. By also testing our methods on Hex, we achieve better comparability.

5.2. Self-Play

Hex is played with two players, i.e. there are two agents, each of whom chooses their action successively. This problem cannot be represented by an MDP, which describes only a single-agent setting, and we formalize it instead by an alternating two-player Markov game [50]. In this setting, each agent has its own set of actions A_i for agent $i \in [1, n]$, where n is the number of players and both players alternate with their moves, i.e. the first player performs action a_0, then the second player performs action a_1, etc. Also, there are n reward functions $R_i(s, a) = r_i \in \mathbb{R}$, where the reward for Hex is either a win $(+1)$ or a loss (-1). However, this reward is provided only at the very end of the game, while all rewards prior are zero. Thus, Hex is a sparse environment that is known to be difficult to solve [2, 93]. Moreover, it can be described as a zero-sum game, meaning that one agent's win is the other's loss, so one agent's reward can be used to derive the other agent's reward. The goal of the agents is to reach a Nash equilibrium [60], if one exists, which in a sense is an optimal strategy, where deviating from the strategy provides no benefit to the agent if the opponent keeps his strategy and all strategies are known.

The game is set in a competitive multi-agent reinforcement learning setting, so there are several ways to solve it. A naive method is to model both agents as independent and train two different policies separately. However, this approach makes the training more resource intensive and assumes that both players have nothing in common worth sharing in their strategy, which is not the case. The strategies of the two agents are similar, as the goals to connect the edges are also similar, only the direction differs. Instead of independent training, *self-play* (SP) [19] can be used, where we share the same policy between both agents, resulting in a scenario where experience is collected by letting the policy play against itself.

One advantage of this approach is that more training data is available while the number of total policies that need to be trained is reduced. Both have a beneficial effect on training time. On the other hand, the environment becomes non-stationary as both agents are constantly improving. The non-stationary nature of the environment leads to instability during training. However, there are several methods that try to solve this problem. We use the terminology of [34] to describe two of them. The simplest form is naive self-play [79], in which the policy plays only against itself. One problem that occurs with this method is *cyclic policy evolutions* [34], where the policy starts to completely forget old strategies to counter new ones, but after a while reverts to its original strategies. As a result, no real progress is made. Figure 5.2 explains this phenomenon using the rock-paper-scissors game. Another method to prevent cyclic policy evolutions is δ-Uniform self-play [10]. The

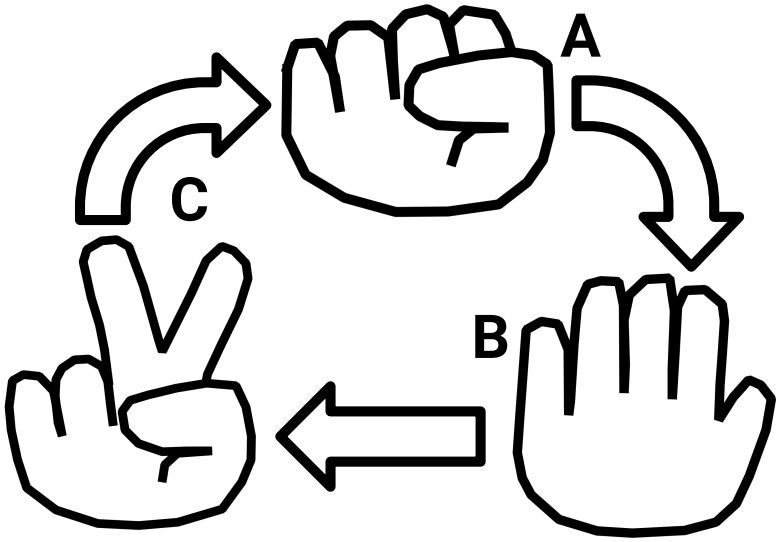

Figure 5.2.: Cyclic policy evolutions by the example of rock-paper-scissors. Assume that two agents play rock-paper-scissors and naive SP is used. **(A)** We consider the initial policy to almost exclusively play rock and only occasionally try paper or scissors. **(B)** After we have sampled experience and updated the policy accordingly, the agent should realize that playing paper significantly increases his win rate since it beats stone, which was used most often. **(C)** However, even this new policy is replaced one or two updates later by a majority use of scissors, which eventually leads us back to the old strategy of using mostly stone. The agent makes no training progress as it focuses only on beating the last strategy observed. As a result, the Nash equilibrium, which plays every action with equal probability, may never be reached.

idea is to have the policy play not only against its current version but also against previous versions of itself. These versions are collected after each training iteration and stored in a history. The value δ defines the percentage of history used, where $\delta = 1$ means that the entire history is used, while $\delta = 0.5$ defines the use of the most recent half, and $\delta = 0$ indicates that only the current version is used to play against itself, resulting in naive SP. When new games are sampled, the current policy is randomly paired against a policy from the history or against itself for the duration of one game, after which the process is repeated. This process ensures that different strategies are found in the training data so that the newest policy must learn more robust strategies to beat them.

5.3. Networks

In this section, we elaborate on the details of the architecture of our policy and VE model. To represent them, we use *deep convolutional neural networks* (CNN), which have already successfully been used in other works regarding board games [97, 84]. On the one hand, the advantages of neural networks are the variety of domains in which they can be used and their generalization capabilities, on the other hand, they may require large amounts of data to work properly and can also lead to instabilities [29]. Since neural networks are differentiable, they are also suitable for use with PPO, which we use for training. We first introduce the basic building blocks of our networks, starting with residual CNNs that use residual layers, which introduce shortcut connections. We then move on to squeeze-and-excitation blocks [37], which we incorporate into our residual blocks to improve performance. Furthermore, we explain the transformations we applied to the Hex board so that it can serve as input to our networks. Afterward, we present the architecture of our policy and model in detail. Finally, we show how we handle invalid actions in our policy.

Residual Layers

The idea behind residual layers is to introduce shortcut connections [74] that allow for better gradient flow and thus better training of deep networks. It has been successfully used in ResNet [33] training a deep network that won the ILSVRC 2015 classification task [77]. One of the main motivations for adding more layers to the network is to increase its capacity, which consequently often leads to improved performance like better accuracy in supervised learning tasks if overfitting can be avoided. Still, training these deep architectures is challenging [69] as several problems can arise such as the problems of vanishing or exploding gradients [12]. They happen during backpropagation, where the gradients are calculated with respect to the parameters, starting from the last layer and moving from layer to layer until reaching the input layer. In the process, the gradient is subject to several multiplications, which can shrink the norm of the gradient so much that the next updates have little or no effect (vanishing gradient), or increase it too much, which leads to unstable and probably performance degrading updates (exploding gradient). However, exploding gradients can be handled by clipping the norm of the gradient [63], leaving only the vanishing gradient problem. Residual networks try to address this problem as follows, let $F(x)$ be a function composed of arbitrary operations e.g. linear/convolutional operations, and let x be the input of this function, then we introduce a shortcut connection, where in the simplest case, we add the input on top of

the result of the function $F(x) + x$. Other transformations can be applied to x within the shortcut, but in this case, we only use an identity mapping. If this residual block is now updated using backpropagation, the shortcut can still propagate back sufficient gradient information, even if in $F(x)$ the problem of vanishing gradients occurs. Thus, residual blocks help to train deeper networks also fixing a degradation problem observed in these networks [33]. It should be noted that AlphaZero and PGS use residual CNNs in their experiments, which highlights the performance of these networks and ensures better comparability to our work.

Squeeze-and-Excitation

Squeeze-and-Excitation (SE) networks [37] add a so-called SE Block, which adds two operations: squeeze and excite. Suppose we have an input image with a given width and height and several channels. The squeeze operation starts by computing a global average pooling operation that averages over all pixels of each channel, effectively leaving us with one value per channel. This vector of values is fed into a two-layer feedforward network with bottleneck architecture and ReLU and then sigmoid activation functions. The output of this network defines channel weights, which are then multiplied back with each channel of the input image and can be seen as channel attention, allowing the network to focus more on certain channels than others. SE blocks are not only very lightweight to compute, but can also add to existing architecture as well such as residual CNNs. The authors showed that this change increased the accuracy of ResNet on the ImageNet classification task [37]. In the context of RL, SE blocks have been successfully applied in the state-of-the-art chess bot Leela Chess Zero [64], which is based on AlphaZero.

Input Features

To represent the game state efficiently and in a way that can also be used by our networks, we turn the board 45 degrees clockwise around its center, so that the black edges are aligned vertically, while for the player with the white pieces they are approximately horizontal. Then, the squares are mapped onto a matrix. Unoccupied squares are represented by zeros, while occupied squares are marked with a one. For each color, we represent the respective stones in a different channel. However, a problem arises using this minimal representation as the policy cannot easily determine which player's turn it is at the moment. The only way for the policy to find out is to count the number of stones and derive this information from that property. We propose a modification that does not require information about

which player's turn it is. We always represent the game state in such a way that the policy thinks it is playing with the black stones. The black stones are always represented in the first channel and the white stones are always represented in the second channel. After the first move is played, both channels are swapped so that the policy thinks it is the first player. With the omission of the color, however, a problem arises, because the player with the black stones must create a vertical connection and the other player must create a horizontal connection. The objective of the game is therefore no longer clear for the policy and training tests showed that the policy tries to occupy the main diagonal to fulfill the victory condition for both sides. We fix this problem by transposing the observation matrix after each move, thus flipping the board around the diagonal and ensuring that both players have the same goal of creating a vertical connection.

Policy and Model

Based on the architecture of AlphaZero, we build our policy as a deep residual CNN with SE blocks. The network starts with a convolutional layer that can reduce the spatial resolution of the input and increase or decrease the number of planes based on the problem. For Hex, we keep the spatial resolution, while increasing the number of planes. This operation is followed by the main structure of the network with a large number of SE residual layers that also preserve the resolution. Each block uses convolutional layers with a kernel size of 3×3, batch normalization [40] to normalize layer input, ReLU activation functions, SE layers, and shortcut connections. For 5×5 boards we use 8 residual blocks and for 7×7 boards we use 16 residual blocks. The last block transitions into two heads, each consisting of 1×1 convolutions, a ReLU activation function, and a final linear layer. The first head describes the simulation policy by computing logits for each action, so for board size n the head returns n^2 logits corresponding with each square, which are then converted to a valid categorical distribution using softmax activation. The second head estimates the value function. Because of the single reward at the end, which is bound between $r \in [-1, 1]$, the value also must be bounded in the same range. Therefore, we employ the Tanh activation function since it maps the values into this range and thus ensures valid values.

The VE model relies on a structure similar to our policy. All three components are represented by deep residual networks with SE layers. Starting with the representation function, which uses a convolutional layer first, like the policy, to significantly increase the number of planes of the input. In contrast to our policy, fewer residual blocks are used, in our experiments, we used 4, but with the same configurations as the policy. The output

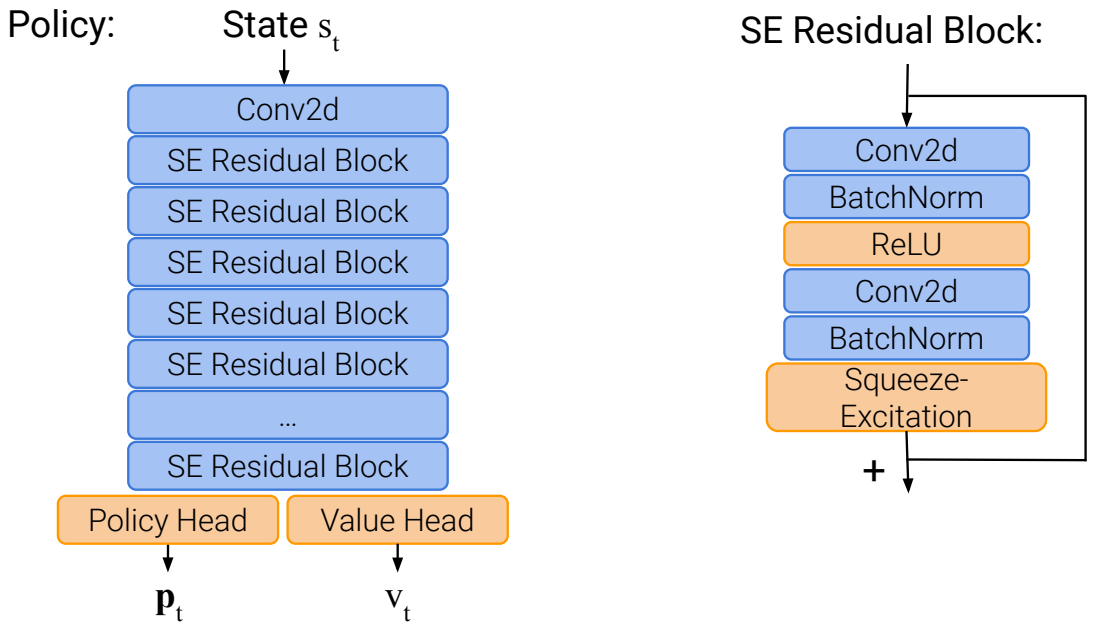

Figure 5.3.: Architecture of our policy. Our policy is shown on the left side. It consists of a convolution followed by a variable number of SE residual blocks, which are shown in more detail on the right. The blocks transition into a policy and a value head, which calculate the respective quantities. The SE residual block on the right side consists of two convolutions followed by batch normalization and a ReLU activation as well as squeeze-and-excitation and a shortcut connection that adds the block input to the residuals.

of these blocks is also the output of our representation function and defines the hidden state, which has the same spatial resolution as the Hex board but many more planes. So, theoretically, the network could use the hidden states to encode the board as it is and try to learn the true transition function if it wants to. Next comes the dynamics function, which expects a hidden state and an action. We represent the action as a one-hot encoded plane, which is then concatenated with the hidden state. This encoding serves as input to the dynamics function, which first reduces the number of planes with a 1×1 convolution and then passes it on to its residual blocks. The output defines the next hidden state but is also used within a head network to calculate the next reward. The structure of the reward head is similar to that of the policy head. Nevertheless, in Hex, a non-zero reward is not given until the very end of the episode, which makes it difficult to predict the reward and often results in noisy rewards that negatively affect our estimation of the return. To

address this issue, we omit the prediction of rewards in our environment and instead focus on estimating the returns using values only. These values are computed by the prediction function, taking the hidden states as input. It also consists of several SE residual blocks, which then transition into two heads similar to those of the policy. One calculates the simulation policy and the other estimates the values.

We implemented all networks and layers using the library PyTorch [65]. A visualization of our policy and the SE residual blocks can be seen in Figure 5.3, while a visualization of our model has already been given in Section 4.3.

Invalid Actions

The game of Hex with board size n starts with n^2 possible actions for the first player to choose from. After choosing a move, the next player has $n^2 - 1$ possible actions available, since one square is now occupied and the action corresponding with placing a stone onto this square is not valid anymore. As the game progresses, the number of legal moves shrinks and the probability of selecting an invalid move increases if our policy does not include knowledge about invalid moves to prevent selecting them. If an invalid action has been selected, we have to resample or use a different strategy to obtain a valid action. To avoid this case, there are several methods, such as penalizing invalid actions with a high negative reward. Still, this scheme allows the policy to select invalid actions. Therefore, we use a different method called invalid action masking [38]. In this method, the logits of the policies corresponding to invalid actions are masked before the softmax activation function is applied to them, which converts unnormalized logits into a valid categorical probability distribution. The masking itself is done by replacing the logits with a high negative value, which is given in the paper as approximately $M = -10^8$. We also adopt this value in our experiments. Furthermore, the authors show that the gradient with respect to the masked policy is still a valid policy gradient with respect to the original policy, so algorithms such as PPO can still be used. Finally, we would like to note that the VE model does not use any methods to handle invalid moves because it has not learned to identify them. Even if it would learn them, one must keep in mind that model errors can still lead to the selection of invalid moves.

5.4. Training

Next, we will look at the training of the networks. We start with our policies, which we update using PPO and δ-Uniform SP. Alternatively, one can use the AlphaZero training scheme, however, in our tests, it turned out much slower than SP with PPO, which is probably also due to the efficiency of the search implementation, which is more difficult to parallelize efficiently. Therefore, we only consider SP and PPO in the following. All hyperparameters used in our training can be found in Appendix A.

To make the training process as quick as possible, available hardware such as CPUs and accelerators such as GPUs must be considered. The environment we use is Hexgame [1], an efficient implementation of Hex in the low-level language Rust [53], which is CPU based. Our networks, on the other hand, can make heavy use of the parallelization capabilities of GPUs. A well-known architecture for training is IMPALA [21], where several worker processes sample trajectories on different CPU cores. Each worker performs inference and environment simulation independently from other workers. However, it is known that CPU network inference is inefficient and accelerators such as GPUs should be used instead [72]. Therefore, we use the Seed RL architecture [22], where each CPU process only deals with simulating the environment and sends the resulting observed states to a global evaluation process to do inference on the GPU. The individual inference requests of each worker are combined into batches to make use of parallelization, then evaluated using our networks, and the resulting actions and values are sent back to the respective workers so that they can continue with the simulation of the environment. Since PPO is an on-policy method and asynchronous sampling can lead to off-policy trajectories [21], we synchronize all workers between each time step ensuring on-policy data while only slightly degrading performance.

Looking at the training process itself, in each iteration, we generate games for exactly $10,000$ time steps on 240 environments distributed over 30 worker processes. We employ δ-Uniform SP with $\delta = 1$ and thus in each game, the current policy plays against an opponent sampled from the complete history of past policies. However, it quickly became apparent to us that randomly sampling opponents as described in [10] has a negative impact on training time. Specifically, if different opponents are used in different games that run in parallel, network inference must be performed for each of these opponents, which means that we lose the parallelization effect, as observations cannot be batched and processed in parallel by the same network. To optimize this process, we propose not to sample the opponents randomly, but to pick them according to a schedule. Using a schedule is valid since for the policy update the order in which our policy plays against

its opponents is not crucial. Only the distribution, i.e. how often each opponent was played, is important. Because we choose opponents uniformly, the number of matches with each opponent should be approximately the same. Combined with the fact that we can gain performance by playing the same opponent in parallel games, we propose a simple schedule where all opponents are played in succession. That is, given 30 parallel workers, all parallel running games are played against the same opponent p_0 at the beginning. Let n be the number of all opponents, then we play against this opponent for $1/n$th of the time and then switch to the next opponent p_1, which we play for the same amount of time, and so on. This process led to a decrease in training time, while we did not observe any degradation in playing strength.

The sampled games are then combined into a large batch of data, from which a dataset $\mathcal{D}$ is created. This dataset is used to sample mini-batches of random positions that are then used to update our policy. The size of these mini-batches was chosen to be as large as possible, but small enough to still fit in the GPU memory. In our experiments on an NVIDIA A100, we use a batch size of 2048 positions. The rest of the update follows the PPO method described in Algorithm 1. All other hyperparameters for the training can be found in Appendix A.

ELO

During training, the agent will be updated over several iterations by using SP. However, measuring the progress of the policy in reaching an improved strategy is difficult as there is no absolute metric to measure the playing strength. Still, for evaluation purposes it is desirable. One metric, we can easily obtain is the win rate against current or older versions of the policy. Although it can give us some insight, several win rates against different policies can be difficult to interpret. Another idea would be to test the agent against another baseline method and use the single win rate to evaluate the progress. Nevertheless, this baseline only works until the agent has reached a level of proficiency, where it always beats the baseline and another stronger baseline would be needed. The ELO rating system [20] presents a solution to this problem. It is a rating system for estimating the relative performance between several players. In our case, these players are the current history of policies. In this system, each player i is given an ELO number r_i, which describes the rating of the player and is simple to interpret. Given two players a and b, who play a game against each other, and their respective ELO ratings, we can calculate the the probability of player a defeating player b

$$P(a \text{ defeats } b) = \frac{1}{1 + 10^{(r_b - r_a)/c_{\text{elo}}}},$$

where $c_{\text{elo}} = 400$ denotes a constant, which can be chosen arbitrarily, but this assignment is the most common one and is also used in our experiments. Players with a higher ELO rating have a higher probability of winning, which seems reasonable. After both players have finished their game, a fixed score s_a is returned, which is equal to one if player a was victorious and zero otherwise. Using the score, we can update the ratings of both players

$$r_a \leftarrow r_a + k(s_a - P(a \text{ defeats } b)),$$

with k as yet another constant for controlling the strength of the update, with higher values translating into stronger updates. For our experiment, we choose $k = 30$. The update for player b can be derived by switching players a and b in the formulas.

We employ the ELO system to evaluate the training progress of our agent, which can be seen in Figure 5.4. Stable training can be achieved using δ-Uniform SP. On the other hand, tests with naive SP resulted in a divergence of policy playing strength. At the end of the training, the policy was not able to win with high probability against a baseline that uniformly selects actions. The problem of cyclic policy evolutions occurs when the policy begins to forget old strategies since earlier versions of the policy can beat the baseline, and focuses only on specific strategies. Thus, it shows the need for methods such as δ-Uniform SP.

Value Equivalent Model

The VE model can be trained in several ways. We can use either MuZero or VE-PGS Exit, but both methods need to perform a model search to compute an action distribution, so they can gather experience from the environment for training. Even searching for a single action on a GPU took several seconds (about ~ 3 seconds in our tests), and with the need to simulate several hundred thousand actions per iteration, our training runs became too slow, even using parallelization. For comparison, in [80], the authors state that they used about 1000 Google Cloud TPUs to collect trajectories with SP. Although the domains they used are more complex than Hex with a small board size, this example demonstrates the enormous computational power required to effectively train these models. Nevertheless,

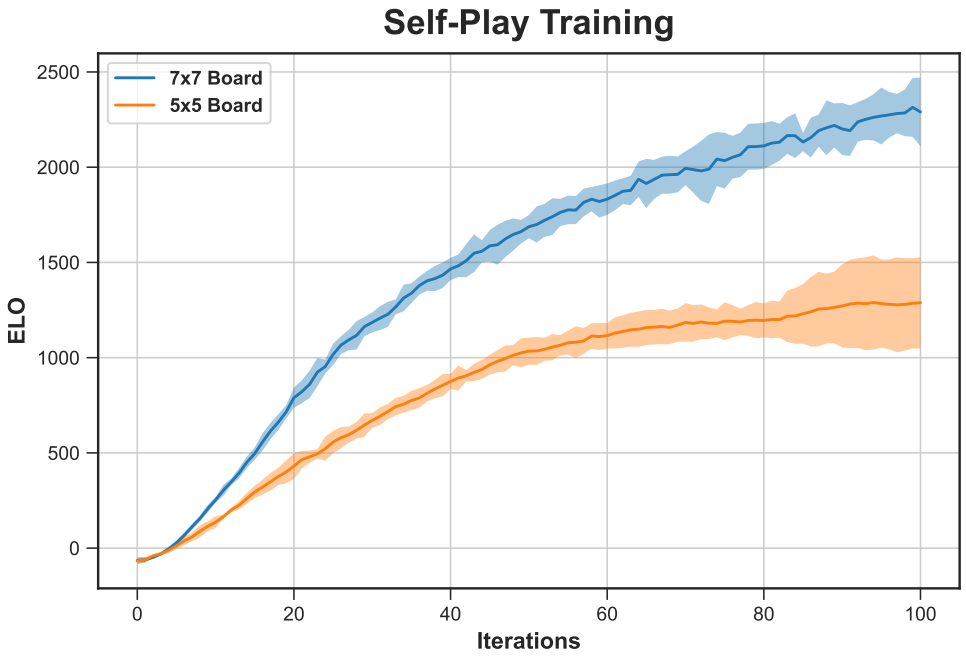

Figure 5.4.: ELO progression of policy with δ-Uniform SP. The policies were trained for 100 iterations over the course of three days on a 5×5 board and 7×7 board, respectively. The number of observed states used in the 5×5 configuration is significantly less than in the 7×7 configuration, and other policy networks were used. Therefore, ELO should not be compared between the two settings. The results show a stable training behavior where both policies converge to a local solution. The training runs were performed using five different seeds. 95% confidence intervals are plotted.

to train a VE model, we use a different training scheme, which we briefly introduce in this section.

Our method is based on supervised learning, where we have a dataset of positions with corresponding values and actions on which we fit our model. The dataset is created by letting two ϵ-greedy policies play against each other and collecting the resulting positions. In our tests, we use a high $\epsilon = 0.9$ for which a random move is played, otherwise, the best action is taken according to our trained policy. Thus, we ensure a large variety of game positions with a small preference for better moves that we assume the model will encounter during its search. All positions are evaluated using AlphaZero, which performed

the fastest of all methods in our tests. For each move, we use 1000 iterations for the search, as this number seems to be a good compromise between performance and accuracy, with position scores changing only a little at higher iteration counts. The advantage of this method is that it is highly parallelizable, and since AlphaZero's policy network is much smaller than the VE model, it can be effectively run on a single CPU. We used 1536 CPU cores to sample $256,000$ games and annotate them accordingly. Then, we trained our VE model on randomly selected 90% of the data and used the rest as a test dataset to evaluate the generalization error of our model.

During training, mini-batches are uniformly sampled from the dataset. For each observed state within the batch, the VE model is unrolled by $K = 5$ steps into the future and the predicted values as well as the simulation policy are tried to match the predicted value and policy from AlphaZero by using an MSE and cross entropy loss, respectively, similar to MuZero's loss described by Equation (4.1). Since we do not predict rewards, the reward loss is omitted. We trained for several epochs and employed early stopping [59], where we stop the training of our model if the test error starts to rise, which is a strong indicator of overfitting and loss of generalization.

5.5. Evaluation

We evaluate our extensions to PGS as well as VE-PGS on the environment of Hex, a two-player, zero-sum and deterministic board game that we introduced in Section 5.1. Thus, it enables us to test our method against state-of-the-art approaches such as AlphaZero and MuZero, without any modifications as would be needed for continuous or stochastic settings [57, 5]. Ultimately, we are interested in testing our methods in domains with resource constraints, so we limit each search to a small number of iterations per move. This constraint is especially reasonable in the game Hex since the moves are usually played under time constraints as well. We start our experiments with an ablation study of our extensions for PGS and then compare our extended version against different baselines such as AlphaZero, which all plan on a perfect simulator of the environment. Afterward, we do the same for our method VE-PGS comparing it with MuZero, which also learns a VE model to plan on but builds an expanding search tree do to so. All hyperparameters used in our experiments can be found in Appendix A.

Extensions

To verify the effectiveness of our proposed extensions, we test each extension individually using as baseline the original method of PGS [4] with a fixed trajectory length of $K = 5$. We also set the PUCT exploration coefficient $c = 5$ and the learning rate to $\alpha = 1\text{e-}3$. Both values were obtained from hyperparameter tests. All methods use a perfect simulator of the environment with a board size of 5×5. This configuration was chosen mainly due to limited computational resources. For our initial tests, we used our most performant policy network as the simulation policy. However, its win rate was already too high, so planning in general hardly led to any improvement. The win rate was about 100% with the white pieces showing that the policy came very close to a dominant strategy. To provide more clear evaluation results, we instead use a less performant policy network. We achieve this decrease by using an earlier version of the policy, in our tests we use a policy with about 150 ELO rating. Since the policy and value function are coupled, this also leads to a more inaccurate value function. We test the following extensions, which we introduced in Section 3.2:

1. **Dynamic Length**: The length of the trajectory is adjusted dynamically. We start with trajectories of length $K = 1$ and increase the length depending on the number of visits to the node. We follow Equation (3.2) and use a branching factor of $b = 6$.
2. **Value Weighting**: The trajectory length is kept fixed, however, the backpropagated return is a weighted sum of all observed values. We use Equation (3.3) with constant $p = 0.995$.
3. **Entropy Bonus**: We add an entropy bonus $c_1 H(\pi)$ to the update of the simulation policy with $c_1 = 2.0$.
4. **KL Penalty**: The update is constrained to stay close to the original policy π_ϕ using a KL divergence penalty term $c_2 \text{KL}(\pi_{\text{sim}} || \pi_\phi)$ with $c_2 = 0.1$.
5. **Policy Target**: Instead of using normalized visit counts to define the resulting action distribution, we use an alternative policy target based on the action-value estimation, see Equation (3.4). We define $\sigma(Q) = (c_3 + \max_a N(s_0, a))c_4(Q - V(s_0))$ like in [18] with $c_3 = 5$ and $c_4 = 0.01$.
6. **Update**: We replace the vanilla REINFORCE update with an improved version using an average baseline like in Equation (3.5). Furthermore, we employ ADAM instead of SGD for parameter optimization.

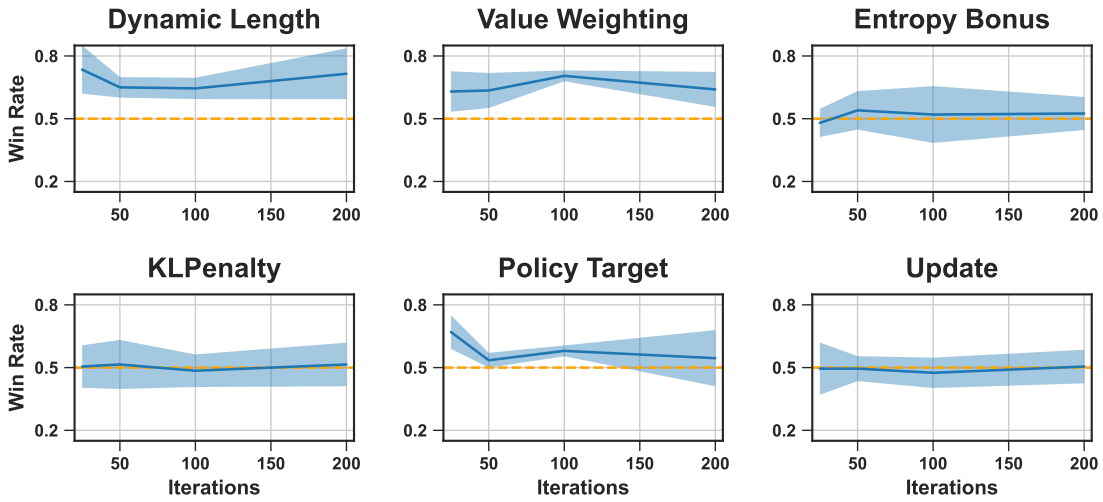

Figure 5.5.: Evaluation of individual extensions. Performance of each extension compared to the original PGS method. We plot the average win rate, giving each method a fixed search budget of iterations per move. Used policy and hyperparameters were chosen to be the same if possible. For each number of iterations per move, each method plays 20 games once with the black pieces and once with the white pieces. In general, the extensions mostly improve the win rate but some show little effect. 95% confidence intervals are shown using 5 different random seeds.

The results of the ablation study can be seen in Figure 5.5. We see that the extensions of dynamic trajectory length as well as weighting the values contribute to an increase in the win rate regardless of the number of iterations. We assume the reason for this effect to be that these extensions successfully reduce the variance of the backpropagated values, but at the cost of more bias. On the other hand, the entropy bonus and KL penalty term, seem to have minimal effect, although we can still measure a slight increase in the win rate by using the entropy bonus. We suspect this effect to be due to the sparse setting of the environment and the rather small number of iterations that there is hardly any premature convergence of the simulation policy reducing the need for these extensions. We also observe a similar effect with the modified update, which seems to have no effect. As we will see in later experiments, training the simulation policy in Hex is difficult, which may explain why update related changes have little effect. Nevertheless, for the new policy target, an improvement is observable. Especially for a small number of iterations, we can see an increase in the win rate, since normalized visit counts are inaccurate when the number of visits is small. Nevertheless, because Hex has a moderate action space, the

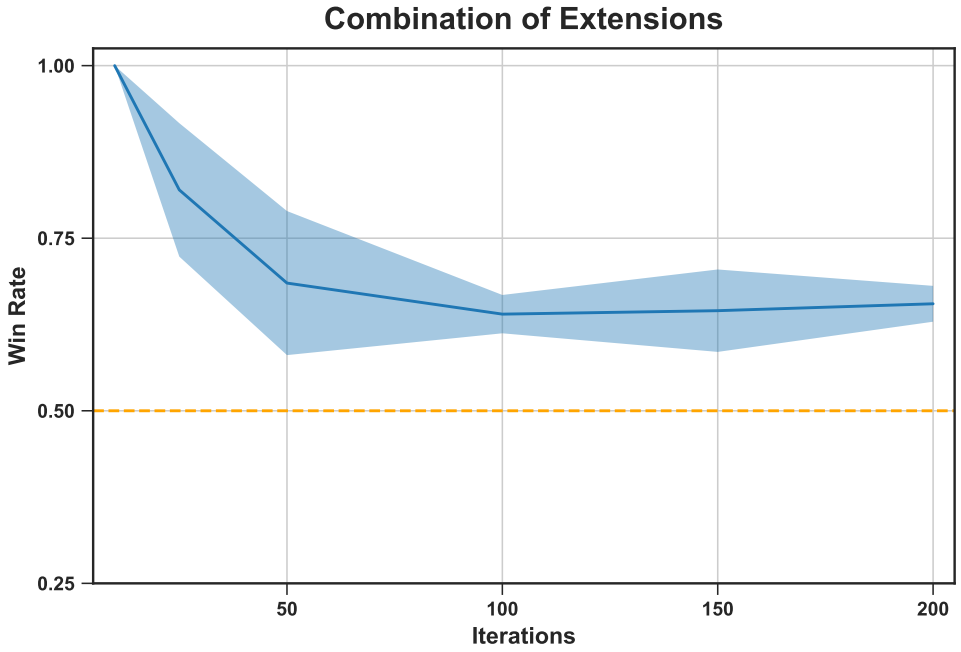

Figure 5.6.: Combinations of PGS extensions. Combinations of all extensions compared to the original version of PGS. Especially for a small number of iterations, we can observe significant improvements due to the changed policy target. With an increased number of iterations, a smaller but stable improvement can be observed. We use the same experimental setup as for the ablation study. Plotted 95% confidence intervals over 5 random seeds.

effect is only visible to a limited extent.

Next, we analyze the extent of improvement resulting from the combination of all methods. We assume that certain extensions may interfere with each other, e.g. the dynamic length of the trajectory and the weighting of the values of these trajectories, since short trajectories have fewer values that can be weighted. Still, in our experiments, using different combinations did not change our results. Therefore, we use all extensions at once. Again, we employ the original version of PGS as a baseline and apply the same hyperparameters as in our ablation study. The results can be seen in Figure 5.6. In particular, with small numbers of iterations < 25, we see a significant increase in the win rate. This iteration count is smaller than the number of possible actions for the first

move, so PGS can visit each action only a few times, resulting in a nearly uniform policy if normalized visit counts are used. The new policy target with action-value estimates is not affected by this effect, which explains the performance improvement. For later iterations, we see an improvement in the win rate that is similar to the dynamic length extension and the value weighting extension.

Finally, we compare our extended version of PGS against other methods. We use the following baselines:

1. **Policy Gradient Search** (PGS) [4]: Like in previous experiments, we test against our implementation of PGS without any extensions and fixed trajectory length. Hyperparameters are set as before.
2. **Monte Carlo Search** (MCS) [91]: Uses random rollouts without expanding its search tree similar to PGS, but does not train its simulation policy. In our experiments, we used our PGS implementation with learning rate $\alpha = 0$.
3. **Monte Carlo Tree Search** (MCTS) [17, 48]: MCTS as described in Section 2.2 with UCT, so without the use of domain knowledge. We set the exploration coefficient of UCT $c = 5$ the same as we set the PUCT coefficient.
4. **AlphaZero** (AZ) [84]: The AlphaZero search algorithm, which is based on MCTS with PUCT. We implemented the method as stated in Section 2.3. The hyperparameters are the same as in PGS if possible.
5. **Trained Policy** (TP): Instead of search, we use inference directly on the trained neural network policy employed in all other informed search methods.

We implemented all baselines using Python [92]. Additionally, we use the same trained policy network as before for all methods. For evaluation, all methods play against each other and the win rate is noted. Each method produces a categorical distribution over all actions, with invalid moves masked. Original tests in which we chose moves based on sampling resulted in high variance in win rates, which made evaluation difficult. However, increasing the number of games is impractical due to large time consumption, so for each method, we choose the best action with respect to the action distribution instead. See Table 5.1 for the results. We note that since we could not implement the same kind of parallelization scheme for all approaches, we decided to run all methods without parallelization. For example, we implemented a lock-free parallel version for MCTS with our own proposed changes, see Appendix B, but could not do the same for AlphaZero due to time constraints. As in the previous experiment, we can show that our extended PGS version outperforms the baseline version and also improves upon the trained policy

	MCS		MCTS		AZ		TP		PGS		Ours	
MCS	-		0.96 0.03	±	0.35 0.12	±	0.83 0.06	±	0.50 0.12	±	0.40 0.05	±
MCTS	0.04 0.03	±	-		0.01 0.02	±	0.15 0.15	±	0.03 0.04	±	0.07 0.08	±
AZ	**0.65** **0.12**	±	**0.99** **0.02**	±	-		0.79 0.07	±	**0.65** **0.12**	±	**0.52** **0.07**	±
TP	0.17 0.06	±	0.85 0.15	±	0.21 0.07	±	-		0.21 0.12	±	0.14 0.11	±
PGS	0.50 0.12	±	0.97 0.04	±	0.35 0.12	±	0.79 0.12	±	-		0.35 0.11	±
Ours	0.60 0.05	±	0.93 0.08	±	**0.48** **0.07**	±	**0.86** **0.11**	±	**0.65** **0.11**	±	-	

Table 5.1.: Win rate table for evaluating extended PGS. Comparing the methods of Monte Carlo Search (MCS), Monte Carlo Tree Search (MCTS), AlphaZero (AZ), the trained policy network (PT), Policy Gradient Search (PGS) and our extended PGS method on Hex with a 5×5 board. We report the average win rate of each method playing 40 games as black and 40 games as white against each other method. Search algorithms get a budget of 100 iterations per move. The best results achieved are marked in bold. Our extended version outperforms most of the other methods with competitive results compared to AlphaZero, against which it performs slightly worse. Averaged over 5 different random seeds with 95% confidence.

network. Interestingly, PGS and MCS are relatively equal in performance, which is not consistent with the results in [4], as they had better results for PGS because of its trained simulation policy. Nevertheless, it should be noted that significantly more iterations were used in their work, which could be a reason for this mismatch. It is difficult to train the simulation policy properly with a small number of iterations in this environment, which also confirms our results from the ablation study, where changes to the update led to modest results. Still, baseline PGS performs worse than AlphaZero, which is consistent with the results of [4]. Our extended version of PGS, on the other hand, is almost on par with AlphaZero and outperforms all other methods, demonstrating the strength of our extensions for PGS in Hex.

	MZ	VE-PGS	Rand	Ext-PGS
MZ	-	$\mathbf{0.5 \pm 0.00}$	0.99 ± 0.01	0.83 ± 0.04
VE-PGS	$\mathbf{0.5 \pm 0.0}$	-	$\mathbf{1.0 \pm 0.0}$	$\mathbf{0.99 \pm 0.02}$
Rand	0.01 ± 0.01	0.0 ± 0.0	-	0.0 ± 0.0
Ext-PGS	0.17 ± 0.04	0.01 ± 0.02	$\mathbf{1.0 \pm 0.0}$	-

Table 5.2.: Win rate table for evaluating VE-PGS. Comparing the methods of MuZero (MZ), Value Equivalence for Policy Gradient Search (VE-PGS), random baseline (Rand) and our Extended PGS (Ext-PGS) on a 5×5 Hex board. We report the average win rate of each method playing 40 games as black and 40 games as white against each other method. Search algorithms get a budget of 100 iterations per move. The best results achieved are marked in bold. VE-PGS is on par with MuZero, while strongly outperforming the random baseline and the extended version of PGS. It has the highest average win rate of all methods. Averaged over 5 different random seeds with 95% confidence.

Value Equivalence for PGS

In this section, we will evaluate our VE-PGS method. We test it on a 5×5 Hex board using the VE model we trained and the extensions of PGS in our method. We compare VE-PGS with the following baselines:

1. **MuZero** (MZ) [80]: This method, similar to VE-PGS, uses a learned VE model to plan on. We introduced it in detail in Section 4.2. However, it uses a search method based on AlphaZero and MCTS to build a search tree. PUCT coefficients were chosen as in the last experiment.
2. **Random Baseline** (Rand): To test the basic functionality of our method, we use a random baseline that selects actions uniformly from all legal moves.
3. **Extended PGS** (Ext-PGS): We also test VE-PGS against our previous method without a learned VE model but instead using a perfect simulator with a learned simulation policy. The hyperparameters are chosen as in the last experiment.

We note that MuZero and VE-PGS share the same model, which was chosen with the lowest error on the test set. Again, we present a table with the win rates of each method

playing against each other, which is shown in Table 5.2. All methods have a fixed budget of 100 iterations per move. Unsurprisingly, the search methods perform significantly better than the random baseline. Nevertheless, we introduce it to check the basic functionality of the VE model, since model errors could cause the methods to fail, which can be detected using this baseline. Nevertheless, all methods seem to work as they strongly outperform it. Furthermore, VE-PGS is on par with MuZero, winning 50% of the time. Both methods seem to have found dominant strategies with the black pieces, resulting in them almost always winning when they make the first move. Moreover, VE-PGS significantly outperforms our extended version of PGS. Still, we note that the accuracy of the model's value estimates compared to that of the learned value function is the main reason for this performance improvement as both methods are the same except for the model. We have introduced this baseline mainly so that we have yet another baseline to compare it with MuZero since the results with VE-PGS are otherwise completely balanced. If we look at how MuZero performs against extended PGS we see lower win rates than VE-PGS, which leads us to the assumption that MuZero's planning algorithm has problems exploiting the model as VE-PGS does. To examine this assumption, we took a closer look at the games between the two and found that VE-PGS wins against MuZero are much shorter taking only 9.50 ± 1.73 moves on average with 95% confidence, while MuZero's wins took more than twice as long, namely 19.84 ± 1.97 moves. For an example game, see Figure 5.7, where we can see that MuZero did not win in obvious winning positions, although such positions should not be a problem for MCTS-based methods. Therefore, we conclude that model errors limit the effectiveness of MuZero, while VE-PGS can handle them better, especially through extensions like return weighting, which counteracts noisy estimates. Furthermore, VE-PGS has less exploration and samples the trajectories only using its simulation policy. It seems to select the actions in a more greedy fashion, which is very effective at least in this environment. This leads to VE-PGS achieving significantly faster victories.

Finally, we would like to mention a quantity that is not immediately obvious, the memory consumption of the individual methods. Since memory usage can vary depending on the system, the implementation and the variable types used, we do not measure it but give a theoretical estimation instead. Starting with PGS and VE-PGS, which have a static and shallow tree and a simulation policy. The shallow tree has a node for the root state and all of his child actions, so at maximum $n^2 + 1$ nodes with $n = 5$ as the board size in one dimension. Each node consists of a 64 bit integer to represent the number of visits and a floating point 64 bit value to represent the number of cumulative returns. In addition, there is a 64 bit pointer to the parent node, a 64 bit integer to represent the chosen action leading to the node and two vectors with n^2 entries. The first stores the prior probabilities of each child node and the second one has pointers to all of them. Other representations

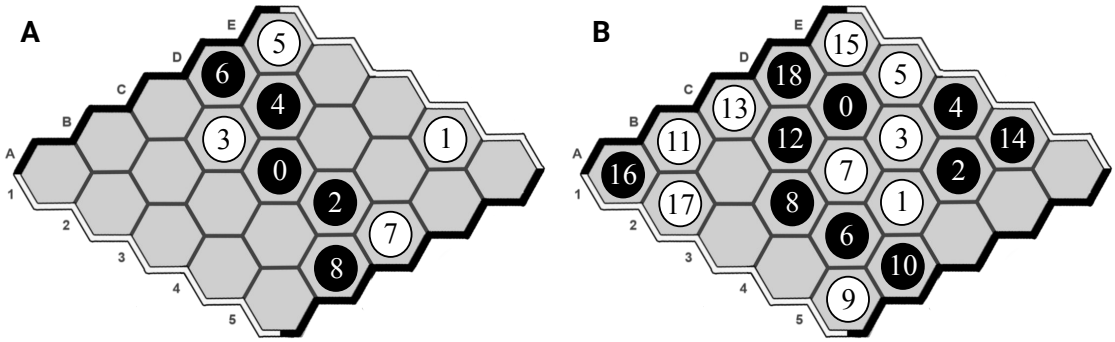

Figure 5.7.: VE-PGS vs MuZero. VE-PGS plays alternately with the black stones **(A)** and the white stones **(B)**. Both VE-PGS and MuZero have won using the black stones, however, the victory of VE-PGS is more clear. In particular, in **(B)** on move 14, MuZero has two ways to win, either D1 or E1. However, it plays E4, a move that does not contribute to winning at all. In this case, the model estimated the values of the states incorrectly, so D1 is not found until move 18. MuZero can handle this kind of mistake much less well than VE-PGS and is worse at exploiting the model properly with low iterations.

of the nodes are possible but will require more memory since these fields are all necessary. However, the number of child nodes can be smaller since invalid actions can be neglected. For simplification, we exclude this fact. Usually, also the current state is stored within the node with size m_{state} in bits. For representing a Hex board with size $n = 5$ it is only 608 bits as board states can be represented very efficiently and for our hidden state, it is $102,400$ bits. Thus, the size of a node can be estimated by $m_{\text{node}} = 64(4+2n^2)+m_{\text{state}}$ bits with n^2 child nodes. The size of a node can be estimated to $m_{\text{node}} = 64(4 + 2n^2) + m_{\text{state}}$ bits with n^2 child nodes. So, the shallow tree has a memory consumption of $(2^n + 1)m_{\text{node}}$. The memory usage of the policy consists only of the last linear head since we freeze the rest of the network. A linear layer only consists of a weight matrix and bias vector and in our experiments has the size of $m_{\text{sim}} = 32(32n^4 + n^2)$ bits. A theoretical estimate of the memory consumption of PGS and VE-PGS can thus be given by $m_{\text{PGS}} \approx 12.7$ Kilobyte (KB) and $m_{\text{VE-PGS}} \approx 330.8$ KB, which is constant with the number of iterations. We specify the memory consumption of AlphaZero and MuZero as a function with respect to the number of iterations i. Using the same nodes as PGS and VE-PGS, respectively, and adding one node each iteration, the memory consumption at the end of each iteration is $m_{\text{AZ}}(i) = (i + 1)m_{\text{node}}$. The definition for MuZero is the same but with a different size for the node since hidden states are used. So, memory consumption grows linearly for both methods and for $i = 100$ iterations, which we used in our experiments, it is estimated to

be approximately $m_{\text{AZ}}(100) \approx 51.3$ KB and $m_{\text{MZ}}(100) \approx 1336.4$ KB. Thus, memory usage is significantly higher between AlphaZero and PGS and between MuZero and VE-PGS even for a small number of iterations.

5.6. Discussion

In this section, we will briefly discuss our results as well as the choice of environment. We have shown the effectiveness of the dynamic length, the weighted return and the modified policy target extensions. The other three extensions of the entropy bonus, the KL penalty and the modified update, however, showed little improvement. We attribute this result to the difficulty of learning the simulation policy with few iterations and using only value estimates instead of returns in a sparse reward setting. Nevertheless, these ideas have already been successfully used in other domains e.g. the idea of entropy for PGS was explored in [49] and the introduction of a baseline and the use of ADAM have also shown improvements in performance [95, 45]. So there are strong indications that these extensions are also useful. By combining these extensions, we were able to show further improvements in comparison to the original version of PGS, especially for a low number of iterations that were smaller than the number of possible actions. In comparison with baseline methods, we could show that PGS can achieve good results and is competitive with AlphaZero. In further experiments, we showed the effectiveness of our method VE-PGS, which operates without knowledge of the dynamics, in comparison with MuZero. Both had equal playing strength when paired together. However, VE-PGS performed better when tested against the extended version of PGS and also won games faster and more accurately than MuZero, thus we concluded that VE-PGS can handle model inaccuracies better and chooses actions more greedily. Since the search tree is not expanded, the memory consumption of our methods is lower and constant, while it grows linearly for MCTS-based methods. Still, performance remains competitive. This feature makes the search suitable for resource-intensive problems e.g. ones with large branching factors.

Still, a limitation of PGS and our method VE-PGS is its speed. In our tests, it performed much slower than, for example, AlphaZero despite the freezing of layers. This issue makes it suitable only for static environments and not for time-critical applications such as real-time control of robotic systems [62]. We also observed a strong instability in the method. Updating the simulation policy with single trajectories can lead to poor estimation of the policy gradient and because of the sparse reward setting, the update incorporates only learned value estimates, which can be inaccurate. Additionally, the trajectory return also

exhibits high variance. Combined with a learned model, which itself may also have errors, can lead to the failure of the method. Accurate models and hyperparameter tuning thus becomes very important, but can not fix the instability either.

We also address one problem with our evaluation, which is the choice of our environment Hex. The original work of PGS and other relevant works such as Expert Iteration [3], AlphaZero, and MuZero are all tested on board games such as Hex and Go. Furthermore, in these works, they mostly limit themselves to only one task. This fact also motivated us to solely use Hex for our evaluation, which was not trivial to solve due to the sparse rewards and multi-agent setting. However, this choice of ours resulted in a difficult evaluation. To evaluate our method, we were required to compare it against other methods to determine its relative playing strength. There was no clear metric to determine the performance of our search methods independent of other methods, like in a single-agent setting, where the cumulative return is a strong indicator of absolute performance. In addition, the sparse reward setting has prevented the testing of methods such as GAE [81], since it estimates the advantages using returns and the rewards are mostly zero especially as we use truncated rollouts. Considering other problems, which was not possible due to a lack of time, would have given us more opportunities to understand the strengths and weaknesses of our method, but could also allow us to test our VE-PGS Exit training scheme. We address this issue further in our future work in Section 7.1.

6. Related Work

Our work presents VE-PGS, a combination of PGS and VE models. To our knowledge, there exists no prior work that already combines these two concepts. Therefore, we look at works that deal with either one of these topics or are related in some other way to our work. We start with a brief introduction of a paper that is similar to our training scheme and motivated it. After that, we deal with work related to PGS, move on to MCTS-based methods, which have similarities to PGS and also motivated one of our extensions, and then conclude with related work on VE models.

A work that is similar to ours in training via SP is [9]. In the paper they let agents compete against each other in two-player competitive 3D tasks such as Sumo or penalty shootouts. For training they also make use of PPO and SP, however, instead of having one policy network for both sides as we have, they have individual networks for each agent since in tasks like the soccer penalty shoutouts, the agents have different roles to fulfill i.e. one shooting the ball and the other one defending. So, the usage of naive SP does not lead to cyclic policy evolutions, but rather to one agent dominating the other in terms of skill with the other agent not being able to catch up. Therefore, they introduce the sampling of older opponents, against which the agents compete, leading to more robust agents and less imbalance between them. This method can be seen as an instance of δ-Uniform SP, which we use in our thesis.

PGS

With respect to PGS [4], there exists only little follow-up research. However, one work that also tries to improve on PGS is [49]. The authors look at the task of symbolic optimization where a sequence of symbols has to be found, which describes some objective e.g. solving a mathematical equation using expressions. In comparison to classical optimization, where parameters for a given function are searched, in symbolic regression the parameters as well as the function, which is composed of symbols, have to be found, increasing the search

space tremendously. Efficiently storing all possible sequences of symbols within a tree structure becomes infeasible because of the exponential number of possibilities, so PGS is employed. However, the authors identify an early commitment problem of the simulation policy for their task, where at the beginning of the search, tokens are chosen with near zero entropy, which leads to suboptimal solutions because of missing exploration. In order to overcome this problem, they introduce hierarchical entropy regularization, which like our work, introduces a regularization term to the update of the simulation policy. Still, they weigh the regularization terms over the search trajectories with earlier symbols getting higher weights and exponential decay for later symbols. For our experiments, we did not identify any early commitment problem, which can be explained through our use of prior knowledge within the simulation policy. The first actions taken were normally already quite good, leading us only to the use of an entropy term without any weighting. For the symbolic optimization task, however, no prior knowledge was used.

Another work that deals with PGS and its exploration problem is Exploratory Policy Gradient Search (ExPoSe) [56]. In this work, tree search and PGS are combined, in order to improve their overall performance. Like in PGS, random rollouts are performed starting from the search node, however, for every newly visited state a new node is created and added to the search tree. It keeps track of statistics like the visitation count and the values encountered during the search, which is then used to estimate values and to improve the exploration of the simulation policy, by adding an exploration bonus to logits of the policy for less visited actions according to the tree statistics. Since this adjustment changes the simulation policy, the samples are no longer collected on-policy, so basic REINFORCE is no longer applicable. In order to fix this problem, the authors introduce importance sampling to the updated formula. In the puzzle game of Sokoban and other tasks, they validate the performance of their new approach. In our work, however, we omit a search tree because of the disadvantages a search tree imposes such as its demand for memory, especially for high branching environments with resource-intensive states. Still, we also use node statistics to balance our trust in the values obtained by our simulation policy during the rollout by changing how these values are weighted before they are backpropagated.

MCTS

Next, we look at methods based on MCTS, which is conceptually very similar to PGS, however, it builds up a search tree to store node statistics, which are used during its search. The method of AlphaGo [85] can be seen as an extension to MCTS with trained neural networks for the selection phase, where the network is included using PUCT, and during

the simulation phase, where the random rollouts are done by sampling actions from a fast rollout policy, truncating and evaluating the last state using a learned value function. All networks were trained either by supervised learning from human expert matches or by reinforcement learning with SP similar to ours. The incorporated knowledge in the search helps AlphaGo to defeat not only MCTS but even the best human professional players in the board game Go. This algorithm was improved further by AlphaGoZero [86], in which a neural network was trained *tabula rasa* via the search without the need for any prior expert data. Moreover, the simulation step did not use any rollouts but rather evaluated the current expanded node directly using the value function, making this approach more different from ours in comparison to AlphaGo and our method. In the paper of AlphaZero [84], the method was generalized to be used in other domains than just two-player games. Furthermore, the game of Go is symmetric, which was exploited during the training of AlphaGoZero, however, AlphaZero does not exploit symmetry since it is not generally applicable. In our method, we also do not exploit any symmetry in the game of Hex. After AlphaZero, the paper of MuZero [80] was proposed. This work is the most similar to our work as it employs a VE model on which it plans using a variant of AlphaZero. On the other hand, we plan by using a variant of PGS. The model is also learned without any prior knowledge using only SP. Although we also present a method, which is able to achieve to learn only from SP, namely VE-PGS Exit, we train our model in a supervised learning fashion from annotated games using AlphaZero, as it was more computationally feasible for us. Lastly, we mention the method of Gumbel AlphaZero and Gumbel MuZero [18], where different extensions are proposed to improve the algorithms for low iteration planning. One of the key improvements is to replace the normalized visit counts with an action-value estimate and prior policy to present a new policy target, which we also employ in our extensions to PGS.

Value Equivalence

Finally, we consider other methods that use VE models to plan or improve a policy on them. All the methods have in common that they learn internally an abstract state transition function, in order to concentrate on learning correct values. Starting with Value Iteration Networks [90] that introduce a neural network architecture with a differentiable planning component, which is based on the value-iteration algorithm to compute the optimal value function. The algorithm can be approximated using a CNN and the model parameters can be trained using backpropagation. Similarly, the method of TreeQN [23] also proposes a network architecture with planning capabilities for learning an action-value function. In the paper, the architecture of the network resembles a tree search with state transitions,

value estimations, and tree backup operations, all of which are expressed differentiable, so the network is end-to-end trainable. Moreover, the predictron [87] learns an abstract model of the world as a Markov reward process, which it unrolled at every timestep, collecting rewards along the way and using them to estimate the value of the given state. Value Prediction Network [61] use an architecture, which is very similar to ours, transforming states into abstract states and planning on them via simulating several trajectories to a certain depth and then taking the best one. However, no estimation of a simulation policy is made.

7. Conclusion

In this work, we have presented the method of Value Equivalence for Policy Gradient Search (VE-PGS), which combines tree-less search with VE models and thus enables planning on environments for which no simulator or knowledge about the dynamic is available. Our approach is based on Policy Gradient Search (PGS), which instead of building a tree structure trains a simulation policy during the search that guides it towards higher valued nodes. The advantages of the method are the generalization capabilities of the simulation policy as well as the significantly reduced memory consumption. Still, there are several limitations, such as the high variance of the sampled trajectories, the lack of exploration and the policy targets, which do not work for small numbers of iterations. Our approach overcomes these limitations by using various extensions such as an entropy bonus, KL penalty term, dynamic lengths of trajectories, value weighting, a new policy target based on action-values and a modified update with baseline. Then, we combined our extended version of PGS with VE models, which use hidden states and leave the representation of these states completely to the model. Therefore, the model is able to focus only on those aspects of the environment that are important for value-based planning. In our conducted experiments on the two-player game of Hex on a 5×5 board, we not only demonstrated the efficacy of our extensions in an ablation study but also outperformed the base version of PGS. Moreover, our extended method achieved competitive results against the state-of-the-art method of AlphaZero, while Hex is more favorable to MCTS-based methods. In our tests with VE models, we showed that VE-PGS also achieved competitive results against MuZero, while the memory consumption was constant with respect to the number of iterations, while it grew linearly with MuZero. We have also provided a method to train a VE model without the need of a search tree using VE-PGS Exit. In conclusion, we advise a method that can plan without knowledge of the environment, and be used in highly complex problems whose high resource requirements, e.g., due to high branching factors or memory requirements, make the use of a search tree infeasible.

7.1. Future Work

As described in Section 5.6, there are still some problems with PGS that we have not been able to address, but we have ideas to do so in the future. The first problem is the instability of the method. Instead of using individual trajectories for updating, we can collect a batch of experience and use it instead to get a better estimation of the policy gradient while making the method slower as the simulation policy is not updated every iteration. The idea of a replay buffer could also be explored to reuse data, still, off-policy methods would be needed, which introduce more instability [88]. Furthermore, the method is quite slow relative to AlphaZero, mainly because of repeated network inference during sampling. An interesting idea to reduce the time of network inference is quantization [41]. Using low-precision weights on modern hardware can speed up inference significantly with minimal impact on accuracy. Additionally, the investigation of parallelization schemes for VE-PGS can increase performance even further. Based on the IMPALA and Seed RL architectures [21, 22], efficient parallelization can be devised. We have described our ideas on this topic in Appendix D, but were unable to implement and test them due to lack of time.

Nevertheless, the Hex environment we used as a benchmark was not favorable to PGS-based methods because the weaknesses and problems of search trees do not occur, such as high memory consumption, as well as stochastic transitions, and continuous action spaces. Still, Hex enabled us to test our method against AlphaZero and MuZero without any modifications, and although our algorithms were weaker, they still achieved competitive results. In the future, however, we will address problems where the drawbacks of MCTS are more obvious, such as continuous and stochastic environments, and where a model of the environment is easier to learn, such as single-agent environments without sparse rewards. Thus, not only the evaluation is simplified, but also more informative statements about the performance of our approach can be made.

Using other environments and improving the performance of VE-PGS will also enable us to test the method of VE-PGS Exit, where we train a new model using only our VE-PGS method for action calculation and as a policy target. Tests with PGS and the Exit algorithm were already successful in this regard [4]. Moreover, our current model represents the environments as deterministic. In order to test VE-PGS in stochastic environments, we need to learn a stochastic model. Some work already exists in this direction [5] with modifications to the VE model of MuZero, which can be interesting for future research in combination with PGS. Additionally, the same applies to continuous action spaces and VE models [96].

Acknowledgments

I would like to thank my parents, my church community and my friends for their support. Additionally, I would like to thank my supervisor João Carvalho for discussions on this thesis. Moreover, I gratefully acknowledge the computing time provided to me on the high-performance computer Lichtenberg at the NHR Centers NHR4CES at TU Darmstadt. It is funded by the Federal Ministry of Education and Research, and the state governments participating on the basis of the resolutions of the GWK for national high performance computing at universities (`www.nhr-verein.de/unsere-partner`).

Bibliography

[1] Altmayer, Martin. *hexgame*. Feb. 25, 2022. URL: https://github.com/MartinAltmayer/hexgame.

[2] Marcin Andrychowicz et al. "Hindsight Experience Replay". In: *CoRR* abs/1707.01495 (2017). arXiv: 1707.01495. URL: http://arxiv.org/abs/1707.01495.

[3] Thomas Anthony, Zheng Tian, and David Barber. "Thinking Fast and Slow with Deep Learning and Tree Search". In: *CoRR* abs/1705.08439 (2017). arXiv: 1705.08439. URL: http://arxiv.org/abs/1705.08439.

[4] Thomas Anthony et al. "Policy Gradient Search: Online Planning and Expert Iteration without Search Trees". In: *CoRR* abs/1904.03646 (2019). arXiv: 1904.03646. URL: http://arxiv.org/abs/1904.03646.

[5] Ioannis Antonoglou et al. "Planning in Stochastic Environments with a Learned Model". In: *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022. URL: https://openreview.net/forum?id=X6D9bAHhBQ1.

[6] Ioannis Antonoglou et al. "Planning in Stochastic Environments with a Learned Model". In: *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022. URL: https://openreview.net/forum?id=X6D9bAHhBQ1.

[7] Broderick Arneson, Ryan B. Hayward, and Philip Henderson. "Monte Carlo Tree Search in Hex". In: *IEEE Trans. Comput. Intell. AI Games* 2.4 (2010), pp. 251–258. DOI: 10.1109/TCIAIG.2010.2067212. URL: https://doi.org/10.1109/TCIAIG.2010.2067212.

[8] K. J. Åström and P. Eykhoff. "System Identification-A Survey". In: *Automatica* 7.2 (Mar. 1971), pp. 123–162. ISSN: 0005-1098. DOI: 10.1016/0005-1098(71)90059-8. URL: https://doi.org/10.1016/0005-1098(71)90059-8.

[9] Trapit Bansal et al. “Emergent Complexity via Multi-Agent Competition”. In: *CoRR* abs/1710.03748 (2017). arXiv: 1710.03748. URL: http://arxiv.org/abs/1710.03748.

[10] Trapit Bansal et al. “Emergent Complexity via Multi-Agent Competition”. In: *CoRR* abs/1710.03748 (2017). arXiv: 1710.03748. URL: http://arxiv.org/abs/1710.03748.

[11] Marc G. Bellemare et al. “The Arcade Learning Environment: An Evaluation Platform for General Agents”. In: *CoRR* abs/1207.4708 (2012). arXiv: 1207.4708. URL: http://arxiv.org/abs/1207.4708.

[12] Yoshua Bengio, Patrice Y. Simard, and Paolo Frasconi. “Learning long-term dependencies with gradient descent is difficult”. In: *IEEE Trans. Neural Networks* 5.2 (1994), pp. 157–166. DOI: 10.1109/72.279181. URL: https://doi.org/10.1109/72.279181.

[13] Jim Blythe. “Decision-Theoretic Planning”. In: *AI Mag.* 20.2 (1999), pp. 37–54. DOI: 10.1609/aimag.v20i2.1455. URL: https://doi.org/10.1609/aimag.v20i2.1455.

[14] Cameron Browne et al. “A Survey of Monte Carlo Tree Search Methods”. In: *IEEE Trans. Comput. Intell. AI Games* 4.1 (2012), pp. 1–43. DOI: 10.1109/TCIAIG.2012.2186810. URL: https://doi.org/10.1109/TCIAIG.2012.2186810.

[15] Guillaume Chaslot, Mark H. M. Winands, and H. Jaap van den Herik. “Parallel Monte-Carlo Tree Search”. In: *Computers and Games, 6th International Conference, CG 2008, Beijing, China, September 29 - October 1, 2008. Proceedings*. Ed. by H. Jaap van den Herik et al. Vol. 5131. Lecture Notes in Computer Science. Springer, 2008, pp. 60–71. DOI: 10.1007/978-3-540-87608-3_6. URL: https://doi.org/10.1007/978-3-540-87608-3%5C_6.

[16] Keh-Hsun Chen, Dawei Du, and Peigang Zhang. “Monte-Carlo Tree Search and Computer Go”. In: *Advances in Information and Intelligent Systems*. Ed. by Zbigniew W. Ras and William Ribarsky. Vol. 251. Studies in Computational Intelligence. Springer, 2009, pp. 201–225. DOI: 10.1007/978-3-642-04141-9_10. URL: https://doi.org/10.1007/978-3-642-04141-9%5C_10.

[17] Rémi Coulom. “Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search”. In: *Computers and Games, 5th International Conference, CG 2006, Turin, Italy, May 29-31, 2006. Revised Papers*. Ed. by H. Jaap van den Herik, Paolo Ciancarini, and H. H. L. M. Donkers. Vol. 4630. Lecture Notes in Computer Science.

Springer, 2006, pp. 72–83. DOI: 10.1007/978-3-540-75538-8_7. URL: https://doi.org/10.1007/978-3-540-75538-8%5C_7.

[18] Ivo Danihelka et al. "Policy improvement by planning with Gumbel". In: *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022. URL: https://openreview.net/forum?id=bERaNdoegn0.

[19] Anthony DiGiovanni and Ethan C. Zell. "Survey of Self-Play in Reinforcement Learning". In: *CoRR* abs/2107.02850 (2021). arXiv: 2107.02850. URL: https://arxiv.org/abs/2107.02850.

[20] Arpad E. Elo. *The Rating of Chessplayers, Past and Present*. New York: Arco Pub., 1978. ISBN: 0668047216 9780668047210. URL: http://www.amazon.com/Rating-Chess-Players-Past-Present/dp/0668047216.

[21] Lasse Espeholt et al. "IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures". In: *CoRR* abs/1802.01561 (2018). arXiv: 1802.01561. URL: http://arxiv.org/abs/1802.01561.

[22] Lasse Espeholt et al. "SEED RL: Scalable and Efficient Deep-RL with Accelerated Central Inference". In: *CoRR* abs/1910.06591 (2019). arXiv: 1910.06591. URL: http://arxiv.org/abs/1910.06591.

[23] Gregory Farquhar et al. "TreeQN and ATreeC: Differentiable Tree Planning for Deep Reinforcement Learning". In: *CoRR* abs/1710.11417 (2017). arXiv: 1710.11417. URL: http://arxiv.org/abs/1710.11417.

[24] Kunihiko Fukushima. "Cognitron: A Self-Organizing Multilayered Neural Network". In: *Biol. Cybern.* 20.3–4 (Sept. 1975), pp. 121–136. ISSN: 0340-1200. DOI: 10.1007/BF00342633. URL: https://doi.org/10.1007/BF00342633.

[25] Kunihiko Fukushima. "Neocognitron: A Self-Organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position". In: *Biological Cybernetics* 36 (1980), pp. 193–202.

[26] Viktor Gál et al. "Receptive Field Atlas and Related CNN Models". In: *Int. J. Bifurc. Chaos* 14.2 (2004), pp. 551–584. DOI: 10.1142/S0218127404009545. URL: https://doi.org/10.1142/S0218127404009545.

[27] David Gale. "The Game of Hex and the Brouwer Fixed-Point Theorem". In: *The American Mathematical Monthly* 86.10 (1979), pp. 818–827. DOI: 10.1080/00029890.1979.11994922. eprint: https://doi.org/10.1080/00029890.1979.11994922. URL: https://doi.org/10.1080/00029890.1979.11994922.

[28] Martin Gardner. *The 2nd Scientific American Book of Mathematical Puzzles and Diversions*. Simon Schuster, 1961.

[29] Ian J. Goodfellow, Yoshua Bengio, and Aaron C. Courville. *Deep Learning*. Adaptive computation and machine learning. MIT Press, 2016. ISBN: 978-0-262-03561-3. URL: http://www.deeplearningbook.org/.

[30] Christopher Grimm et al. "Proper Value Equivalence". In: *CoRR* abs/2106.10316 (2021). arXiv: 2106.10316. URL: https://arxiv.org/abs/2106.10316.

[31] Christopher Grimm et al. "The Value Equivalence Principle for Model-Based Reinforcement Learning". In: *CoRR* abs/2011.03506 (2020). arXiv: 2011.03506. URL: https://arxiv.org/abs/2011.03506.

[32] Ryan B. Hayward et al. "Solving 7x7 Hex: Virtual Connections and Game-State Reduction". In: *Advances in Computer Games*. 2003.

[33] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *CoRR* abs/1512.03385 (2015). arXiv: 1512.03385. URL: http://arxiv.org/abs/1512.03385.

[34] Daniel Hernández et al. "A Comparison of Self-Play Algorithms Under a Generalized Framework". In: *CoRR* abs/2006.04471 (2020). arXiv: 2006.04471. URL: https://arxiv.org/abs/2006.04471.

[35] Matteo Hessel et al. "Muesli: Combining Improvements in Policy Optimization". In: *CoRR* abs/2104.06159 (2021). arXiv: 2104.06159. URL: https://arxiv.org/abs/2104.06159.

[36] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators". In: *Neural Networks* 2.5 (1989), pp. 359–366. ISSN: 0893-6080. DOI: https://doi.org/10.1016/0893-6080(89)90020-8. URL: https://www.sciencedirect.com/science/article/pii/0893608089900208.

[37] Jie Hu, Li Shen, and Gang Sun. "Squeeze-and-Excitation Networks". In: *CoRR* abs/1709.01507 (2017). arXiv: 1709.01507. URL: http://arxiv.org/abs/1709.01507.

[38] Shengyi Huang and Santiago Ontañón. "A Closer Look at Invalid Action Masking in Policy Gradient Algorithms". In: *CoRR* abs/2006.14171 (2020). arXiv: 2006.14171. URL: https://arxiv.org/abs/2006.14171.

[39] "IEEE Standard for Floating-Point Arithmetic". In: *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), pp. 1–84. DOI: 10.1109/IEEESTD.2019.8766229.

[40] Sergey Ioffe and Christian Szegedy. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”. In: *CoRR* abs/1502.03167 (2015). arXiv: 1502.03167. URL: http://arxiv.org/abs/1502.03167.

[41] Benoit Jacob et al. “Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference”. In: *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*. Computer Vision Foundation / IEEE Computer Society, 2018, pp. 2704–2713. DOI: 10.1109/CVPR.2018.00286. URL: http://openaccess.thecvf.com/content%5C_cvpr%5C_2018/html/Jacob%5C_Quantization%5C_and%5C_Training%5C_CVPR%5C_2018%5C_paper.html.

[42] Michael Janner et al. “When to Trust Your Model: Model-Based Policy Optimization”. In: *CoRR* abs/1906.08253 (2019). arXiv: 1906.08253. URL: http://arxiv.org/abs/1906.08253.

[43] J. Kiefer and J. Wolfowitz. “Stochastic Estimation of the Maximum of a Regression Function”. In: *The Annals of Mathematical Statistics* 23.3 (1952), pp. 462–466. DOI: 10.1214/aoms/1177729392. URL: https://doi.org/10.1214/aoms/1177729392.

[44] Man-Je Kim and Kyung-Joong Kim. “Opponent modeling based on action table for MCTS-based fighting game AI”. In: *IEEE Conference on Computational Intelligence and Games, CIG 2017, New York, NY, USA, August 22-25, 2017*. IEEE, 2017, pp. 178–180. DOI: 10.1109/CIG.2017.8080432. URL: https://doi.org/10.1109/CIG.2017.8080432.

[45] Diederik P. Kingma and Jimmy Ba. “Adam: A Method for Stochastic Optimization”. In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2015. URL: http://arxiv.org/abs/1412.6980.

[46] Jens Kober and Jan Peters. “Reinforcement Learning in Robotics: A Survey”. In: *Reinforcement Learning*. Ed. by Marco A. Wiering and Martijn van Otterlo. Vol. 12. Adaptation, Learning, and Optimization. Springer, 2012, pp. 579–610. DOI: 10.1007/978-3-642-27645-3_18. URL: https://doi.org/10.1007/978-3-642-27645-3%5C_18.

[47] Levente Kocsis, Csaba Szepesvari, and Jan Willemson. “Improved Monte-Carlo Search”. In: 2006.

[48] Levente Kocsis and Csaba Szepesvári. “Bandit Based Monte-Carlo Planning”. In: *Machine Learning: ECML 2006, 17th European Conference on Machine Learning, Berlin, Germany, September 18-22, 2006, Proceedings*. Ed. by Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou. Vol. 4212. Lecture Notes in Computer Science. Springer, 2006, pp. 282–293. DOI: 10.1007/11871842_29. URL: https://doi.org/10.1007/11871842%5C_29.

[49] Mikel Landajuela et al. “Improving exploration in policy gradient search: Application to symbolic optimization”. In: *CoRR* abs/2107.09158 (2021). arXiv: 2107.09158. URL: https://arxiv.org/abs/2107.09158.

[50] Michael Lederman Littman. “Algorithms for Sequential Decision-Making”. AAI9709069. PhD thesis. USA, 1996. ISBN: 0591163500.

[51] Iou-Jen Liu, Raymond A. Yeh, and Alexander G. Schwing. “High-Throughput Synchronous Deep RL”. In: *CoRR* abs/2012.09849 (2020). arXiv: 2012.09849. URL: https://arxiv.org/abs/2012.09849.

[52] Peter Marbach and John N. Tsitsiklis. “Approximate Gradient Methods in Policy-Space Optimization of Markov Reward Processes”. In: *Discret. Event Dyn. Syst.* 13.1-2 (2003), pp. 111–148. DOI: 10.1023/A:1022145020786. URL: https://doi.org/10.1023/A:1022145020786.

[53] Nicholas D Matsakis and Felix S Klock II. “The rust language”. In: *ACM SIGAda Ada Letters*. Vol. 34. 3. ACM. 2014, pp. 103–104.

[54] Marvin Minsky and Seymour Papert. *Perceptrons: An Introduction to Computational Geometry*. Cambridge, MA, USA: MIT Press, 1969.

[55] S. Ali Mirsoleimani et al. “A Lock-free Algorithm for Parallel MCTS”. In: *Proceedings of the 10th International Conference on Agents and Artificial Intelligence, ICAART 2018, Volume 2, Funchal, Madeira, Portugal, January 16-18, 2018*. Ed. by Ana Paula Rocha and H. Jaap van den Herik. SciTePress, 2018, pp. 589–598. DOI: 10.5220/0006653505890598. URL: https://doi.org/10.5220/0006653505890598.

[56] Dixant Mittal, Siddharth Aravindan, and Wee Sun Lee. “ExPoSe: Combining State-Based Exploration with Gradient-Based Online Search”. In: *CoRR* abs/2202.01461 (2022). arXiv: 2202.01461. URL: https://arxiv.org/abs/2202.01461.

[57] Thomas M. Moerland et al. “A0C: Alpha Zero in Continuous Action Space”. In: *CoRR* abs/1805.09613 (2018). arXiv: 1805.09613. URL: http://arxiv.org/abs/1805.09613.

[58] Thomas M. Moerland et al. “Model-based Reinforcement Learning: A Survey”. In: *Found. Trends Mach. Learn.* 16.1 (2023), pp. 1–118. DOI: 10.1561/2200000086. URL: https://doi.org/10.1561/2200000086.

[59] Nelson Morgan and Herve Bourlard. *Generalization and Parameter Estimation in Feedforward Nets: Some Experiments*. Tech. rep. TR-89-017. Berkeley, CA: International Computer Science Institute, 1989.

[60] J.F. Nash. “Non-cooperative Games”. In: *Annals of Mathematics* 54.2 (1951), pp. 286–295.

[61] Junhyuk Oh, Satinder Singh, and Honglak Lee. “Value Prediction Network”. In: *CoRR* abs/1707.03497 (2017). arXiv: 1707.03497. URL: http://arxiv.org/abs/1707.03497.

[62] Djamila Ouelhadj and Sanja Petrovic. “A survey of dynamic scheduling in manufacturing systems”. In: *J. Sched.* 12.4 (2009), pp. 417–431. DOI: 10.1007/s10951-008-0090-8. URL: https://doi.org/10.1007/s10951-008-0090-8.

[63] Razvan Pascanu, Tomás Mikolov, and Yoshua Bengio. “On the difficulty of training recurrent neural networks”. In: *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013*. Vol. 28. JMLR Workshop and Conference Proceedings. JMLR.org, 2013, pp. 1310–1318. URL: http://proceedings.mlr.press/v28/pascanu13.html.

[64] Pascutto, Gian-Carlo and Linscott, Gary. *Leela Chess Zero*. Version 0.21.0. Mar. 8, 2019. URL: http://lczero.org/.

[65] Adam Paszke et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[66] Jakub Pawlewicz and Ryan B. Hayward. “Scalable Parallel DFPN Search”. In: *Computers and Games*. 2013.

[67] Jan Peters, Katharina Mülling, and Yasemin Altun. “Relative Entropy Policy Search”. In: *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*. Ed. by Maria Fox and David Poole. AAAI Press, 2010. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI10/paper/view/1851.

[68] Jan Peters and Stefan Schaal. “Policy Gradient Methods for Robotics”. In: *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2006, October 9-15, 2006, Beijing, China*. IEEE, 2006, pp. 2219–2225. DOI: 10.1109/IROS.2006.282564. URL: https://doi.org/10.1109/IROS.2006.282564.

[69] Moacir Antonelli Ponti et al. “Training Deep Networks from Zero to Hero: avoiding pitfalls and going beyond”. In: *CoRR* abs/2109.02752 (2021). arXiv: 2109.02752. URL: https://arxiv.org/abs/2109.02752.

[70] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley Series in Probability and Statistics. Wiley, 1994. ISBN: 978-0-47161977-2. DOI: 10.1002/9780470316887. URL: https://doi.org/10.1002/9780470316887.

[71] Aniruddh Raghu, Matthieu Komorowski, and Sumeetpal S. Singh. “Model-Based Reinforcement Learning for Sepsis Treatment”. In: *CoRR* abs/1811.09602 (2018). arXiv: 1811.09602. URL: http://arxiv.org/abs/1811.09602.

[72] Rajat Raina, Anand Madhavan, and Andrew Y. Ng. “Large-scale deep unsupervised learning using graphics processors”. In: *Proceedings of the 26th Annual International Conference on Machine Learning, ICML 2009, Montreal, Quebec, Canada, June 14-18, 2009*. Ed. by Andrea Pohoreckyj Danyluk, Léon Bottou, and Michael L. Littman. Vol. 382. ACM International Conference Proceeding Series. ACM, 2009, pp. 873–880. DOI: 10.1145/1553374.1553486. URL: https://doi.org/10.1145/1553374.1553486.

[73] Stefan Reisch. “Hex ist PSPACE-vollständig”. In: *Acta Informatica* 15 (1981), pp. 167–191.

[74] Brian D. Ripley. *Pattern Recognition and Neural Networks*. Cambridge University Press, 1996. ISBN: 978-0-521-46086-6. DOI: 10.1017/CBO9780511812651. URL: https://doi.org/10.1017/CBO9780511812651.

[75] Christopher D. Rosin. “Multi-armed bandits with episode context”. In: *Ann. Math. Artif. Intell.* 61.3 (2011), pp. 203–230. DOI: 10.1007/s10472-011-9258-6. URL: https://doi.org/10.1007/s10472-011-9258-6.

[76] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. “Learning Representations by Back-propagating Errors”. In: *Nature* 323.6088 (1986), pp. 533–536. DOI: 10.1038/323533a0. URL: http://www.nature.com/articles/323533a0.

[77] Olga Russakovsky et al. “ImageNet Large Scale Visual Recognition Challenge”. In: *CoRR* abs/1409.0575 (2014). arXiv: 1409.0575. URL: http://arxiv.org/abs/1409.0575.

[78] Charles Saidu, Afolayan Obiniyi, and Peter Ogedebe. “Overview of Trends Leading to Parallel Computing and Parallel Programming”. In: *British Journal of Mathematics Computer Science* 7 (Jan. 2015), pp. 40–57. DOI: 10.9734/BJMCS/2015/14743.

[79] A. L. Samuel. “Some Studies in Machine Learning Using the Game of Checkers”. In: *IBM J. Res. Dev.* 3.3 (July 1959), pp. 210–229. ISSN: 0018-8646. DOI: 10.1147/rd.33.0210. URL: https://doi.org/10.1147/rd.33.0210.

[80] Julian Schrittwieser et al. “Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model”. In: *CoRR* abs/1911.08265 (2019). arXiv: 1911.08265. URL: http://arxiv.org/abs/1911.08265.

[81] John Schulman et al. “High-Dimensional Continuous Control Using Generalized Advantage Estimation”. In: *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2016. URL: http://arxiv.org/abs/1506.02438.

[82] John Schulman et al. “Proximal Policy Optimization Algorithms”. In: *CoRR* abs/1707.06347 (2017). arXiv: 1707.06347. URL: http://arxiv.org/abs/1707.06347.

[83] John Schulman et al. “Trust Region Policy Optimization”. In: *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*. Ed. by Francis R. Bach and David M. Blei. Vol. 37. JMLR Workshop and Conference Proceedings. JMLR.org, 2015, pp. 1889–1897. URL: http://proceedings.mlr.press/v37/schulman15.html.

[84] David Silver et al. “Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm”. In: *CoRR* abs/1712.01815 (2017). arXiv: 1712.01815. URL: http://arxiv.org/abs/1712.01815.

[85] David Silver et al. “Mastering the game of Go with deep neural networks and tree search”. In: *Nat.* 529.7587 (2016), pp. 484–489. DOI: 10.1038/nature16961. URL: https://doi.org/10.1038/nature16961.

[86] David Silver et al. “Mastering the game of Go without human knowledge”. In: *Nat.* 550.7676 (2017), pp. 354–359. DOI: 10.1038/nature24270. URL: https://doi.org/10.1038/nature24270.

[87] David Silver et al. “The Predictron: End-To-End Learning and Planning”. In: *CoRR* abs/1612.08810 (2016). arXiv: 1612.08810. URL: http://arxiv.org/abs/1612.08810.

[88] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018. URL: http://incompleteideas.net/book/the-book-2nd.html.

[89] Maciej Swiechowski et al. “Monte Carlo Tree Search: A Review of Recent Modifications and Applications”. In: *CoRR* abs/2103.04931 (2021). arXiv: 2103.04931. URL: https://arxiv.org/abs/2103.04931.

[90] Aviv Tamar, Sergey Levine, and Pieter Abbeel. “Value Iteration Networks”. In: *CoRR* abs/1602.02867 (2016). arXiv: 1602.02867. URL: http://arxiv.org/abs/1602.02867.

[91] Gerald Tesauro and Gregory R. Galperin. “On-line Policy Improvement using Monte-Carlo Search”. In: *NIPS*. 1996.

[92] Guido Van Rossum and Fred L Drake Jr. *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

[93] Matej Vecerık et al. “Leveraging Demonstrations for Deep Reinforcement Learning on Robotics Problems with Sparse Rewards”. In: *CoRR* abs/1707.08817 (2017). arXiv: 1707.08817. URL: http://arxiv.org/abs/1707.08817.

[94] Jiayi Weng et al. “EnvPool: A Highly Parallel Reinforcement Learning Environment Execution Engine”. In: *CoRR* abs/2206.10558 (2022). DOI: 10.48550/arXiv.2206.10558. arXiv: 2206.10558. URL: https://doi.org/10.48550/arXiv.2206.10558.

[95] Ronald J. Williams. “Simple statistical gradient-following algorithms for connectionist reinforcement learning”. In: *Machine Learning*. 1992, pp. 229–256.

[96] Xuxi Yang, Werner Duvaud, and Peng Wei. “Continuous Control for Searching and Planning with a Learned Model”. In: *CoRR* abs/2006.07430 (2020). arXiv: 2006.07430. URL: https://arxiv.org/abs/2006.07430.

[97] Kenny Young, Ryan Hayward, and Gautham Vasan. “Neurohex: A Deep Q-learning Hex Agent”. In: *CoRR* abs/1604.07097 (2016). arXiv: 1604.07097. URL: http://arxiv.org/abs/1604.07097.

A. Hyperparameters

Hyperparameters	Value
batch size	2048
clip ratio ϵ	0.2
discount factor γ	1.0
entropy bonus	0.01
gradient clipping	100.0
GAE λ	1.0
learning rate	6e-4
number of filters	128
number of residual blocks	8 (5×5), 16 (7×7)
number of steps per iteration	0.12M (5×5), 2.4M (7×7)
PPO iterations	6
c_{elo}	30
self-play δ	1
squeeze-and-excitation	Yes
training iterations	100

Table A.1.: Hyperparameters used for policy training with δ-Uniform SP.

Hyperparameters	Value
batch size	512
epochs	100
learning rate	6e-5
number of filters	128
number of games	$256,000$
number of residual blocks	4
squeeze-and-excitation	Yes
unroll length K	5

Table A.2.: Hyperparameters used for model training in supervised learning fashion.

Hyperparameters	Value
c_{PUCT}	5.0
c_{UCT}	5.0
PGS branching factor	6
PGS entropy bonus	2.0
PGS KL penalty	0.1
PGS learning rate	1e-3
PGS policy target	$(5, 0.01)$
PGS weight constant	0.995
truncation length	5

Table A.3.: Hyperparameters used for evaluation.

B. Lock-free Implementation of MCTS

In this chapter, we describe our lock-free implementation of MCTS that is based on [55]. We quickly describe their implementation and the difference from ours. The main reason for a parallel implementation of MCTS is the gain in efficiency that comes from it. However, there are several challenges that complicate such implementations. One of them is race conditions, where two or more threads try to access the same resource at the same time and one of the threads modifies it. The result of these operations is undetermined and possibly incorrect. Therefore, synchronization operations are used, which ensure deterministic execution, but at the cost of performance [15]. For example, in parallel MCTS implementations, concurrent access to nodes is prevented by mutual excess locks. Lock-free implementations do not have this limitation, since they allow concurrent access to resources, which is achieved by using e.g. atomic operations. Therefore, these implementations are usually more efficient, but they are also more difficult to implement [55].

We now describe a high-level overview of the lock-free implementation of [15] written in C++. We address each step of MCTS and how it is modified. Suppose we have a set of threads and a shared search tree, each thread now performs the following operations:

(A) **Selection**: Starting from the root node, a selection phase is performed as in MCTS, where the node statistics are read atomically to perform e.g. UCT. In order to prevent the threads from all following the same path through the tree, a virtual loss is introduced. If a thread traverses a node, a loss is applied to this node, which makes it appear as if an iteration was performed that resulted in a loss. As a result, the node becomes less interesting to other threads and the diversity of the selection is greater.

(B) **Expansion**: If multiple threads still have selected the same node, there can be a problem with the expansion. These threads can try to expand the same child node, which leads to a race condition. The authors propose to solive this issue by allowing

only one thread to create all child nodes, then assigning one of these child nodes atomically to each thread.

(C) **Simulation**: The simulation phase can be performed as usual.

(D) **Backprop**: In backpropagation, two parameters are updated per node. First, the number of visits is incremented and then the sum of all returns is added to the result of the rollout. Both values are used to estimate the action-value for the respective node. However, interleaving can occur, i.e. while a thread in the backpropagation phase has atomically incremented the number of visits, but has not yet updated the sum of the returns, another thread in the selection phase may compute the action-value estimate of the node, which will be incorrect as only one statistic has been updated. To solve this problem, the authors propose to store both the number of visits and the sum of returns into one atomic variable of type unsigned integer 64 bit. The upper 32 bits are used to represent the returns while the lower 32 bits are used to represent the number of visits. Thus, the update of both values is performed simultaneously and the race condition can be avoided.

Our implementation is based on this approach, however, with some changes, since we noticed some problems and limitations. First, we use the programming language Rust [53] instead of C++, which offers compile-time memory safety. Furthermore, we propose the following changes:

1. **Limited expansion**: At the beginning of the expansion phase, one thread generates all child nodes and adds them to the search tree. Although these child nodes do not yet possess a simulated state, they still take some memory with empty statistics. Consequently, a large branching factor in the environment can lead to a large number of children of which several could remain unused resulting in large unused memory usage. Therefore, we propose not to generate all child nodes, but rather to assign unique actions to each thread, which are then used by the threads to expand the respective node. We accomplish this idea by generating a list of actions for each newly created node and an index variable for the list. The index is atomically incrementing for each thread trying to expand the node. So, we do not have to generate all child nodes, but only the ones that are used, which saves resources.
2. **Fixed-point arithmetic**: For each node, the statistics for the number of visits and the sum of returns are stored together in an unsigned integer field. Both are represented as 32 bit integers and are updated together via an atomic addition to prevent race conditions. However, the representation of the sum of the returns as an integer is a limitation as only integer rewards can be represented and stored like in e.g. games,

where the reward defines the winner. If one tries to represent this sum as a floating point according to IEEE-754 [39], which is the most common representation, a problem arises because the sum of returns is stored in combination with the number of visits as an integer, and the atomic addition operation of both types is different. In order to use this implementation of MCTS for more complex reward settings, we propose to use fixed-point arithmetic for the representation of the returns instead. Thus, combined updating with the number of visits is possible, since integers and fixed-point floating points can use the same addition operation and so floating-point rewards can be incorporated. To increase the accuracy of this representation, we also propose to change the equal distribution of bits from 32 bits for each variable to 40 bits for the sum of the returns and 24 bits for the number of visits. Thus, over $2^{24} = 16,777,216$ visits of a node can still be represented, which is practicable for most applications, and with the remaining 40 bits, 24 bits are used for the integer part, which covers the same range as the number of visits, 1 bit for the sign, and 13 bits for the fractional part, which provides a sufficiently good representation for floating points. We note that scaling the field for the statistics above 64 bits may cost some efficiency, especially since most system architectures support additions only up to 64 bits.

These changes not only reduce the memory requirements of the method but also increase the number of possible applications.

C. Sample Matches

In the following we present some sample games of our agents. These games present both victories and defeats of some methods and were played on 5×5 boards.

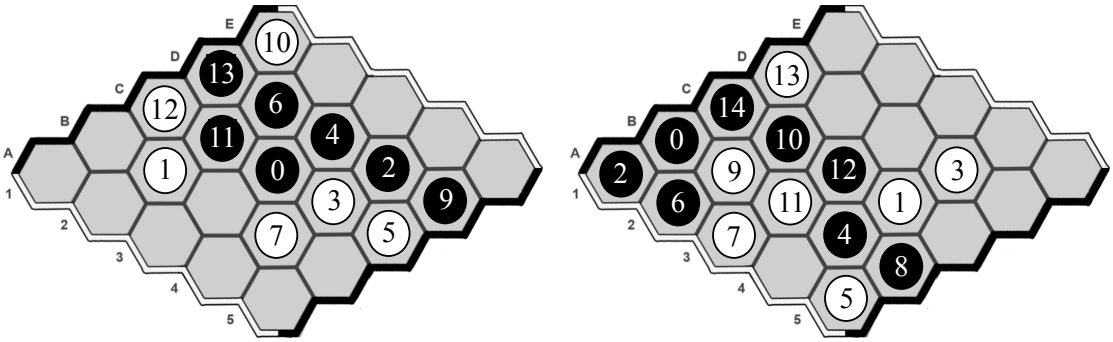

Figure C.1.: Trained Policy vs MCTS. The trained policy plays alternately with the black stones (left) and the white stones (right). In both matches, the black side won.

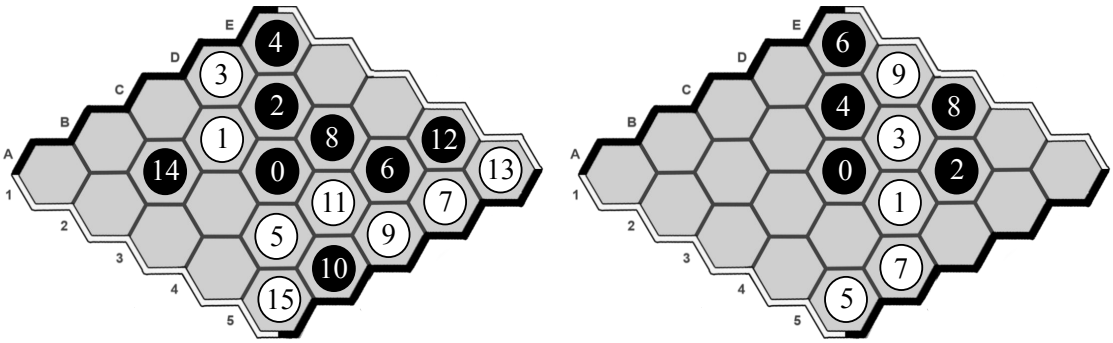

Figure C.2.: Ext-PGS vs AlphaZero. Extended PGS plays alternately with the black stones (left) and the white stones (right). In both cases, the white side won.

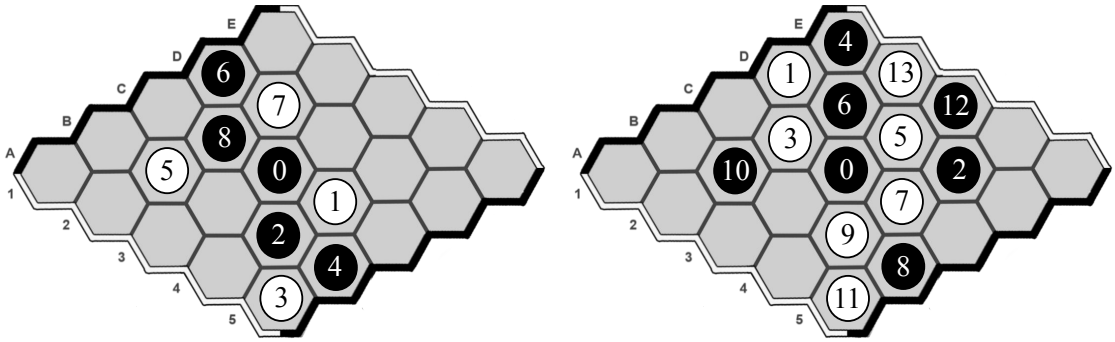

Figure C.3.: VE-PGS vs Ext-PGS. VE-PGS plays alternately with the black stones (left) and the white stones (right). VE-PGS won both times.

D. Parallelization of PGS

In this chapter, we describe our ideas to parallelize PGS. Due to a lack of time, we were not able to implement and test these ideas, but we would like to share them anyway.

Modern hardware relies heavily on parallel computing [78]. Therefore, it is important that algorithms can be parallelized to take full advantage of the hardware and increase their performance. For MCTS and AlphaZero, there are already several ways to parallelize them [15]. For example, one can parallelize the search itself by performing a separate MCTS in each process with its tree (root parallelization) and then average the results across all searches. Another option is to share a tree across all processes, with each process running the MCTS algorithm on that tree and sharing its results with the other processes (tree parallelization). Finally, one can parallelize the rollouts during the simulation phase (leaf parallelization). All processes start at the same node and run a random simulation. The result is then shared with a master process, which continues with the search.

Still, the root parallelization method, while applicable to PGS, has the problem that each search is rather shallow and also does not take advantage of the parallelization capabilities of the neural networks we use. The other two methods, which focus on only one search and attempt to parallelize it, are not directly applicable to PGS. The reason for this problem lies in the concept of the policy-lag [21]. When we simulate two or more random rollouts at the same time, the simulation policy is updated as soon as the first rollout is completed. However, this update causes all other trajectories currently being sampled to no longer be on-policy with respect to the simulation policy, as that policy has changed, making the on-policy REINFORCE update invalid to use. The policy used for sampling the trajectory lags, hence the name. To our knowledge, this issue has not yet been addressed for PGS, which circumvents the problem by synchronizing its processes.

Synchronizing the sampling processes ensures that all trajectories are completed before updating, and also to update using all trajectories at once so that no experience is invalidated. However, this solution does not scale well because all sampling processes must wait for all other processes, including the slowest one, to finish before proceeding. Therefore,

synchronous methods have lower throughput compared to asynchronous methods [51]. So, we propose to look at asynchronous sampling instead, which however requires us to deal with off-policy data. For this purpose, we use importance sampling (IS)

$$\theta \leftarrow \theta + \alpha \sum_{t=1}^{T} \nabla_\theta \log \pi_{\text{sim}}(a_t|s_t) \left(\prod_{l=1}^{t} \frac{\pi_{\text{sim}}(a_l|s_l)}{\beta(a_l|s_l)} \right) \left(V(s_t) - \frac{1}{T} \sum_{k=1}^{T} V(s_k) \right). \tag{D.1}$$

with behavior policy β that sampled the data. This update describes the off-policy gradient with which asynchronous sampling becomes possible. This formula can also be used for on-policy sampling, resulting in $\pi_{\text{sim}} = \beta$ and thus $\prod_{l=1}^{t} \pi_{\text{sim}}(a_l|s_l)/\beta(a_l|s_l) = 1$, giving us our original update again. It should be noted that off-policy methods are unstable when used with function approximators such as neural networks and bootstrapping, forming the deadly triad [88].

Another aspect we have not considered so far is network inference. Today's networks can take advantage of GPU parallelization. However, if each process performs its inference as in IMPALA [21], GPU parallelization is not possible. Therefore, we use the training architecture of Seed RL [22]. In this architecture, a process seeking to perform network inference sends the observed states to an evaluation process, which collects all requests and combines them into a batch. This batch of data is then passed to the network, which can process the data much faster than if it did so sequentially, and then the results are returned to the individual sample worker processes. This architecture has been shown to achieve much higher throughput than IMPALA (for this result see [22]), where inference is performed in each process individually.

Our idea of a parallelized PGS for discrete action spaces would thus use tree-based parallelization, in which all worker processes sampled trajectories using a shared shallow tree. During sampling, each process sends its inference requests to a global evaluation process, which uses GPU inference and caching to provide faster results. To prevent the workers from all simulating the same initial actions, we would use a virtual loss as described in Appendix B. The updates to the simulation policy would be done asynchronously whenever a process has sampled a trajectory and then updated via Equation (D.1). Our approach should allow for more throughput than synchronous methods. Finally, we note that VE-PGS can be parallelized in the same manner.