# Replacing PUCT with a Planning Model

**Ersetzung von PUCT durch ein Planungsmodell**
Master thesis by Adrian Glauben (Student ID: 2628309)
Date of submission: July 12, 2022

1. Review: Prof. Dr. Kristian Kersting
2. Review: M.Sc. Johannes Czech
3. Review: M.Sc. Jannis Blüml

Darmstadt

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Informatik

Artificial Intelligence & Machine Learning Lab

## Erklärung zur Abschlussarbeit gemäß §22 Abs. 7 APB TU Darmstadt

Hiermit versichere ich, Adrian Glauben, die vorliegende Masterarbeit gemäß §22 Abs. 7 APB der TU Darmstadt 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 Falle 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 einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

[illegible]

Darmstadt, 12. Juli 2022

A. Glauben

# Abstract

In recent years, the combination of reinforcement learning with neural networks has found success in various applications. Many of these are based on the well known *AlphaZero* architecture, which utilizes Monte Carlo tree search to train neural networks. Although, it has been seven years since its introduction by Silver et al. (2016) [16], research is still being conducted on finding potential improvements to this architecture. In particular, one of the main interests is in improving the action selection during the search, which is currently done using predictor upper confidence bounds applied to trees. Besides this, another recent and promising research direction focuses on using reinforcement learning to train models for planning.

In this thesis, we integrate these ideas by introducing a planning model to replace the use of upper confidence bounds for action selection during Monte Carlo tree search. Using the *AlphaZero* algorithm we train a base model to play the game Connect4. Afterwards, we show that it is possible to initialize a planning model using imitation learning on the actions selected by the upper confidence bound, without significantly losing playing strength. Lastly, we investigate whether our planning model can be optimized using proximal policy optimization on data generated from self play, and consequently improve the playing strength of our agent.

# Zusammenfassung

In den letzten Jahren wurden durch die Kombination von Reinforcement Learning mit Neuronalen Netzwerken in verschiedensten Anwendungen große Erfolge erzielt. Viele dieser Erfolge beruhen auf dem bekannten *AlphaZero* Algorithmus, welcher *Monte Carlo tree search* zum Trainieren Neuronaler Netzwerke verwendet. Obwohl seit dessen Einführung durch Silver et al. (2016) [16] bereits sieben Jahre vergangen sind, wird noch immer an Verbesserungen dieses Algorithmus geforscht. Ein Hauptinteresse liegt dabei in der Verbesserung der Aktionsauswahl während der Suche, welche derzeit typischerweise mittels *predictor upper confidence bounds* für Bäume erfolgt. Des Weiteren konzentriert sich eine andere aktuelle und vielversprechende Forschungsrichtung auf die Verwendung von Reinforcement Learning zum Training von Planungsmodellen.

In dieser Arbeit verbinden wir diese beiden Ideen, indem wir ein Planungsmodell einführen, welches die Verwendung von *upper confidence bounds* für die Aktionsauswahl während *Monte Carlo tree search* ersetzt. Dafür trainieren wir zunächst ein Basismodell basierend auf dem *AlphaZero* Algorithmus, für das Spiel *Vier Gewinnt*. Anschließend zeigen wir, dass es möglich ist, ohne signifikante Einbußen an Spielstärke, mittels *Imitation Learning* ein Planungsmodell zu initialisieren. Schließlich untersuchen wir, ob unser Planungsmodell mit Hilfe von *proximal policy optimization* trainiert werden kann und folglich die Spielstärke unseres Agenten verbessert.

# Contents

# List of Figures

# List of Tables

# 1. Introduction

## 1.1. Motivation

After Silver et al. (2016) [16] showed that *Monte Carlo tree search* (MCTS) could successfully be combined with deep learning to yield powerful agents for the rather complex game of Go, much interest was sparked in the research on these types of algorithms. When a later iteration of their algorithm, called *AlphaZero* [15], proved the general applicability and zero domain knowledge requirements of their technique, many researchers started to successfully apply this technique to a wide range of problems.

One of the main components of *AlphaZero* is action selection during the tree search by using *predictor upper confidence bound applied to trees* (PUCT). PUCT stems from a family of algorithms from bandit theory, called *upper confidence bound* (UCB), which serve the objective of optimally balancing exploration and exploitation during construction of the search tree. These algorithms have been studied intensively and have been shown to offer theoretical guarantees on convergence. Nevertheless, recent work has shown that different approaches can outperform PUCT action selection [2, 3, 5].

Another rising research interest is using reinforcement learning to train models for planning. A few recent works present promising results with their approaches, showing that *reinforcement learning* (RL) can successfully be utilized to do planning in various environments [7, 12].

Motivated by these findings this work proposes to replace the PUCT action selection in *AlphaZero*-like algorithms by a planning model trained with RL.

## 1.2. Problem Formulation

To our current knowledge, no approaches exist that try to replace the PUCT action selection with a neural network. However, we hypothesize that action selection during tree search could be improved by employing a neural network planning model, which is trained with RL in a self play loop. Furthermore, we hypothesize that action selection could benefit from additional features on top of the standard components of PUCT. Therefore, we propose three features as additional inputs to our planning models: the *Q-value variance*, a *4-step move history* and an *action mask*.

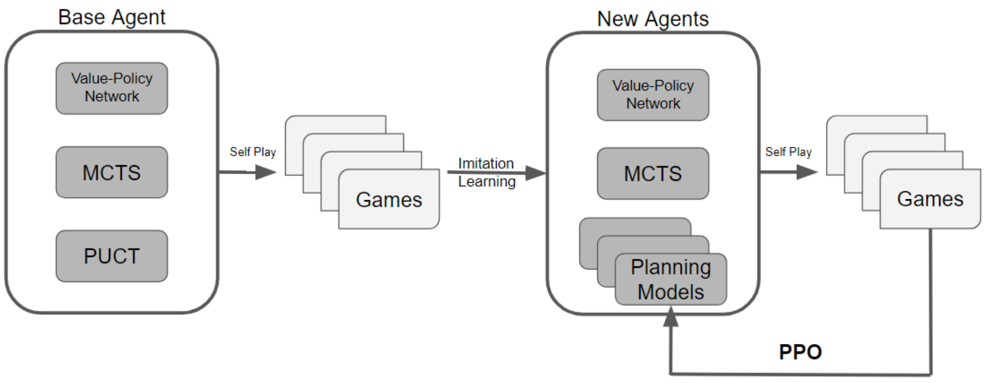


Figure 1.1.: The base agent consists of an *AlphaZero* implementation for the game of Connect4. Using this agent we generate self play games, on which we use imitation learning to initialize the planning models to behave like PUCT. Afterwards, we use the resulting agents to generate self play games, based on which we update the planning model.

Figure 1.1 illustrates our training pipeline. We use the well known game Connect4 as our environment, since it can be rapidly simulated, facilitating faster experimentation. As a first step we train an *AlphaZero* model to learn Connect4. Secondly, we use this base model to initialize our planning models to behave like PUCT. Lastly, we use *proximal policy optimization* (PPO) to update the planning model with data generated in a self play loop.

## 1.3. Outline

Chapter 2 provides the necessary background for our work. Starting with an introduction to exploration and exploitation in RL, by explaining the parts of bandit theory relevant to our work. Followed by an explanation of MCTS and its variation used in the *AlphaZero* algorithm. Closing with an introduction to PPO.

Following, Chapter 3 presents research related to our problem formulation. In particular, it contains recent work on alternatives to the standard PUCT formulation as well as research conducted towards utilizing RL for planning.

Furthermore, in Chapter 4 we provide an in-depth explanation of our implementation, ranging from an explanation of the specifics of our base model to the different planning models build on top of that.

Afterwards, Chapter 5 reports on the results of our implementation. Starting with a validation of our base model, followed by assessing the planning models initialized with imitation learning and closing with the final results of the planning models updated with PPO.

Lastly, in Chapters 6 and 7 we conclude with a discussion of our research question and possible future work on the topic.

# 2. Background

## 2.1. Exploration and Exploitation

One of the unique challenges of RL is the trade-off between exploration and exploitation. The need for exploration arises since RL agents use training information that evaluates the actions they have taken rather than being instructed with good actions. Therefore, successfully learning by experience sampled from an environment requires the agent to carefully balance between exploiting states, which are already known to yield high rewards, and efficiently exploring unknown parts of the environment [19, Chapters 1 & 2].

### 2.1.1. Multi-armed Bandits

Many of the solutions to the exploration-exploitation problem stem from the research on multi-armed bandits. To illustrate these kinds of problems consider the following setup: An agent has $k$ different options, each yielding a reward drawn from their respective stationary probability distribution. The objective is to maximize the expected total reward over a finite number of tries [19, Section 2.1].

Maximizing the expected total reward can be formulated as an action value estimation problem, where the value estimate of an action $a$ is described by the simple moving average over its rewards $R_t(a)$ at time steps $t$,

$$Q'(a) = Q(a) + \frac{1}{n}\left[R_t(a) - Q(a)\right]. \tag{2.1}$$

However, greedily selecting actions with respect to its current value estimate would result in pure exploitation of the first action taken. Therefore, we need to incorporate some form of exploration. The simplest form of exploration is the $\epsilon$-$greedy$ policy, which selects the best action according to the value estimate with a probability of $1 - \epsilon$ and otherwise selects a random action. In the limit this will ensure that the action values $Q(a)$ converge to their respective expected reward $q_\star(a) = \mathbb{E}\left[R_t | A_t = a\right]$ [19, Chapter 2].

### 2.1.2. Upper Confidence Bounds

While enforcing exploration with the $\epsilon$-$greedy$ policy will eventually converge, its reliance on pure chance and no use of additional metrics (e.g. how often an action has been selected) significantly slows down the convergence process.

An improved solution to the above problem is the UCB algorithm, which implements the principle of optimism in the face of uncertainty. In the algorithm this is expressed by the fact that each actions assigned value, it's upper confidence bound, is likely to be an overestimate of it's true expected reward [11, Section 7.1].

Initially each action is selected once, afterwards, at every time step $t > k$, where $k$ is the number of available actions, UCB selects an action according to

$$A_t = \underset{a}{argmax} \left[ Q_t(a) + c\sqrt{\frac{log\, t}{N_t(a)}} \right], \tag{2.2}$$

where $N_t(a)$ denotes the number of times action $a$ has been selected up until time step $t$. The parameter $c$ controls the amount of exploration and $Q_t(a)$ corresponds to the current value estimate of action $a$.

Intuitively the term $\sqrt{\frac{log\, t}{N_t(a)}}$ serves as an exploration bonus or a measure of uncertainty in the current value estimate $Q_t(a)$. Since $N_t(a)$ is in the denominator, the more often action $a$ is selected, the lower its exploration bonus becomes, while the exploration bonus of the other actions rises. This ensures that every action is taken sufficiently often to guarantee convergence of the value estimates towards the expected return of the respective actions [19, Section 2.7].

### 2.1.3. Upper Confidence Bounds applied to Trees

In order to apply the above idea to action selection in tree search algorithms Kocsis and Szepesári (2006) [9] developed the UCT (UCB applied to trees) algorithm. UCT treats the action selection at every internal node of a tree as a separate multi-armed bandit problem. The arms correspond to the available actions in the node and its value estimate corresponds to the empirical mean reward of all paths originating form this node. Similar to UCB (Equation 2.2), UCT chooses the action for state $s$, at time $t$ and depth $d$ by

$$A_{s,t,d} = \underset{a}{argmax} \left[ Q_t(s,a,d) + c\sqrt{\frac{log\, N_{s,d}(t)}{N_{s,d,a}(t)}} \right], \tag{2.3}$$

where $Q_t(s,a,d)$ denotes the estimated value of action $a$ in state $s$ at depth $d$. $N_{s,d}(t)$ is the number of times state $s$ has been visited at time $t$ and depth $d$ and $N_{s,a,d}(t)$ corresponds to the number of times action $a$ has been chosen in state $s$ at time $t$ and depth $d$.

Rosin (2011)[13] improved on the UCB formula by including approximate predictions of good arms, yielding the PUCB formula (Predictor + UCB). This was later adapted to MCTS by *AlphaGo* [16] and called PUCT, an in depth explanation of which follows in Section 2.2.

## 2.2. Monte Carlo Tree Search

MCTS [19, Section 8.11] is a planning algorithm that, in its original form, accumulates value estimates obtained from Monte Carlo simulations in order to direct the search towards more rewarding trajectories.

Monte Carlo simulations estimate state values by averaging over the returns of many simulated trajectories. However, more recent variations of this algorithm discard Monte Carlo simulations in favor of a neural network for performing value estimation. In particular, *AlphaZero* [15] employs a dual headed neural network $(\mathbf{p}, v) = f_\theta(s)$, which takes a representation of the state $s$ as an input and outputs a state value estimate $v$ as well as a vector of move probabilities $\mathbf{p}$. Since *AlphaZero* serves as the basis for our work, this variation of MCTS will be focused upon.

MCTS consists of the four phases depicted in Figure 2.1: selection, expansion, simulation and backpropagation. These four phases are continuously executed in succession until the specified time or computational resource is exhausted. Finally, an action for the state corresponding to the root is selected according to the visit counts of its successors. Once the environment has transitioned to the new state corresponding to the chosen action, this process is repeated until the environment reaches a terminal state.

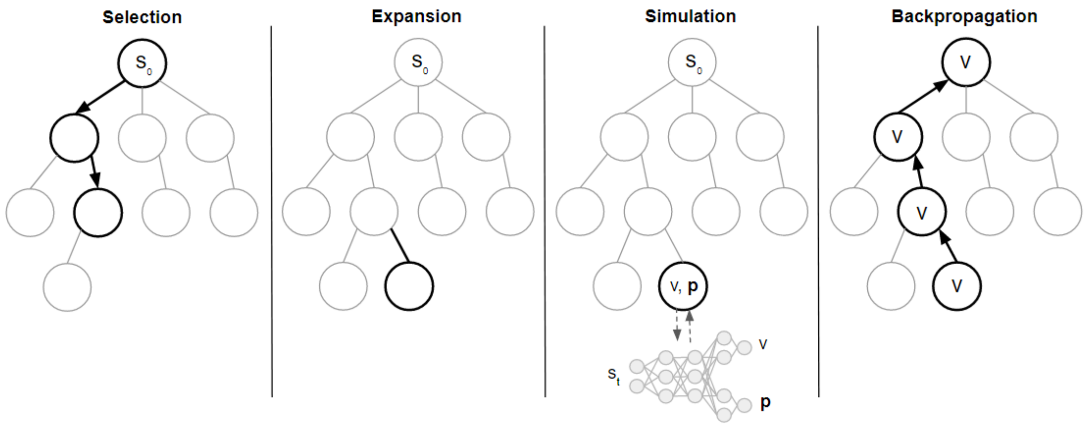


Figure 2.1.: MCTS has four phases. Firstly, during the selection phase, the current tree is traversed until a not fully expanded node is reached. Secondly, in the expansion phase, a new node is added to the tree. Thirdly, during the simulation phase, the value and policy corresponding to the node are gathered from the neural network. Lastly, the predicted value is backpropagated through the tree.

**Selection** During the selection phase the current tree is traversed, starting from the root and ending once a node with at least one unexpanded child is reached. At each node, representing a state $s_t$, the best action $A_t$ is selected according to the PUCT formula introduced by *AlphaGo* [16]:

$$A_t = \underset{a}{argmax}(Q(s_t, a) + U(s_t, a)),\ with \tag{2.4}$$

$$U(s,a) = c_{puct} P(s,a) \frac{\sqrt{\sum_b N(s,b)}}{1 + N(s,a)},$$

where $c_{puct}$ is a hyperparameter determining the amount of exploration. $\sum_b N(s,b)$ is the sum of the visit counts of the current nodes children and $N(s,a)$ is the number of times action $a$ was chosen in state $s$. $P(s,a)$ corresponds to the probability returned from the neural network for choosing action $a$ in state $s$.

**Expansion** If a non terminal leaf node is selected, a new child corresponding to an unexplored state is added to the tree. Which unexplored state is expanded depends on the implementation details. Possible methods include choosing at random or choosing the next child in the list of unexpanded children.

**Simulation** At this point, in the original version of MCTS, one or more rollouts are performed starting from the state corresponding to the newly expanded child. Actions are chosen according to a rollout policy until a terminal state is reached, after which the result is returned. This result $v$ serves as a value estimation for the newly expanded child. However, as described above, *AlphaZero* discards this simulation in favor of a single neural network evaluation, $(\mathbf{p}, v) = f_\theta(s)$, yielding an estimation of the states value $v$ and policy $\mathbf{p}$.

**Backpropagation** Finally, the value estimate obtained by the previous phase is backpropagated along the path taken in the selection phase. Each nodes statistics on this path are updated as follows:

$$N'(s,a) = N(s,a) + 1, \tag{2.5}$$

$$Q'(s,a) = Q(s,a) + \frac{1}{N(s,a)}(v - Q(s,a)). \tag{2.6}$$

In order to train the neural network *AlphaZero* uses data generated from self play. Using the search algorithm described above the agent plays games against itself, where each completed search returns the next move to play until a terminal state $s_T$ is reached. Each search yields a training data point for the neural network in form of the 3-tuple $(s_t, \pi_t, z_t)$, where $s_t$ is the root state of the search conducted at time step $t$. $\pi_t$ is generated proportionally with respect to the visit counts of the root states children, $\pi(s,a) = N(s,a)/\sum_b N(s,b)$, and corresponds to the vector containing the move probabilities for the legal moves available in state $s_t$. $z_t \in \{-1, 0, 1\}$ is the game outcome from the view of the player at time $t$. For example, if the outcome in the terminal state $s_T$ is a win for player one and at time $t$ it was player two's turn, then $z_t = -1$.

Using these data points the neural network is updated according to the loss function below, the optimization objectives of which are: minimizing the error between the predicted value $v$ and the eventual outcome $z$ as well as maximizing the similarity between the predicted policy $\mathbf{p}$ and the policy obtained from the search $\pi$. Additionally, an $L_2$ penalty is added for weight regularization.

$$l = (z - v)^2 - \pi^T log\, \mathbf{p} + c||\theta||^2. \tag{2.7}$$

## 2.3. Proximal Policy Optimization

PPO belongs to the family of policy gradient methods. Contrary to action value methods, which select actions based on their estimated action values, these methods instead learn a with $\theta$ parameterized policy function, $\pi_\theta(a|s) = Pr\{A_t = a|S_t = s, \theta_t = \theta\}$, which performs action selection without consulting a value function [19, Chapter 13].

For our work we consider the policy function to be represented by a neural network with weights $\theta$. Additionally, we consider the task to be episodic, with rewards given only at the end of an episode. Standard policy gradient methods train the neural network using the objective function

$$L^{PG}(\theta) = \hat{\mathbb{E}}_t \left[ log\, \pi_\theta(a_t|s_t)\, G \right], \tag{2.8}$$

where $\pi_\theta(a|s)$ is the output of the neural network for state $s$, representing a stochastic policy. $G$ is the reward given at the end of an episode and $\hat{\mathbb{E}}_t$ indicates the empirical average over a finite batch of samples [14, Section 2.1].

PPO improves on this objective function by introducing the clipped surrogate objective

$$\begin{aligned} L^{CLIP}(\theta) &= \hat{\mathbb{E}}_t \left[ min(r_t(\theta)G,\, clip(r_t(\theta),\, 1-\epsilon,\, 1+\epsilon)G \right],\ with \\ r_t(\theta) &= \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}, \end{aligned} \tag{2.9}$$

where $\epsilon$ is a parameter determining the clipping range. $L^{CLIP}$ takes the minimum between the clipped and unclipped objective to yield a lower bound on the unclipped objective, essentially penalizing large policy updates [14, Chapter 3]. A simple algorithm implementing PPO for the case of episodic tasks with terminal rewards, where $\pi_\theta$ is represented by a neural network with parameters $\theta$, could look as follows:

**Algorithm 1** A simple PPO algorithm

```
Initialize: $\pi_\theta$
for number of training games do:
    Using $\pi_\theta$, generate an episode $S_0, A_0, ..., S_{T-1}, A_{T-1}$
    $G \leftarrow$ Evaluate terminal state $S_T$
    $\pi_{\theta_{old}} \leftarrow \pi_\theta$
    Update $\pi_\theta$ with $L^{CLIP}$, using the generated 3-tuples $(S_t, A_t, G)$
end for
```

# 3. Related Work

The essential proposition of this work is to replace the PUCT action selection formula in *AlphaZero*-like algorithms with a learned planning model. Many works exist that propose improvements and alternatives to the PUCT action selection, most of which focus on static evaluations rather than utilizing a neural network that improves planning through RL. However, the general idea of learning a planning model to replace planning algorithms is not novel and has already been applied in other contexts. In this chapter a selection of recent approaches to replacing and improving PUCT as well as learning planning models will be presented.

## 3.1. Replacing PUCT

### 3.1.1. Planning with Gumbel

In 2022 the team at Deepmind introduced an improvement to their *AlphaZero* and *MuZero* algorithms called *Gumbel AlphaZero* and *Gumbel MuZero*, respectively [5]. This new algorithm replaces the addition of Dirichlet noise for selecting actions to search at the root node by sampling actions without replacement using the Gumbel-Top-k trick proposed by Kool et al. (2009) [10].

Furthermore, they propose to use the Sequential Halving algorithm [8] rather than the PUCT algorithm to select actions in the root node. While PUCT was designed to optimize cumulative regret, using Sequential Halving at the root node optimizes simple regret. Therefore, the latter takes better account for the fact that the performance of MCTS only depends on the final recommended action at the root and not on any actions selected during the search at depths $d > 0$.

Additionally, they discarded PUCT action selection at non-root nodes in favor of selecting actions using an improved policy $\pi'$, which is constructed using a completion of the Q-values. The action at time $t$ is selected according to,

$$A_t = \underset{a}{argmax}\left(\pi'(a) - \frac{N(a)}{1+\sum_b N(b)}\right),\ with \tag{3.10}$$

$$\pi'(a) = softmax(logits(a) + \sigma(completedQ(a))),\ with \tag{3.11}$$

$$completedQ(a) = \begin{cases} q(a), & \text{if } N(a) > 0 \\ v_\pi, & \text{otherwise,} \end{cases} \tag{3.12}$$

where $logits(a) = log\frac{\mathbf{p}}{1-\mathbf{p}}$ corresponds to the natural logarithm of the chances of the actions according to the probabilities given by the policy output **p** of the neural network. $\sigma$ is a monotonically increasing transformation and $v_\pi = \sum_a \pi(a)q(a)$ replaces the unknown Q-values of the yet unvisited actions.

While *Gumbel AlphaZero* only matches the performance of *AlphaZero* and *Gumbel MuZero* only slightly exceeds the performance of *MuZero*, the authors show that their new algorithms keep improving, even when the number of simulations during learning is set to as little as two. Therefore, the main contribution of their work lies in a potential acceleration of future research due to faster experiments facilitated by a small number of simulations.

### 3.1.2. Generalized PUCT

Besides replacing PUCT completely, research has also been conducted on improving PUCT in various ways. One such work is by Cazenave (2021) [3], who generalized PUCT in order to make it invariant of the number of expansions used in the search. Their generalized PUCT (GPUCT) formula discards the square root in the exploration term in favor of an exponential term with parameter $\tau$. The resulting action selection rule is

$$A_t = \underset{a}{argmax}(Q(s_t,a) + U(s_t,a)),\ with \tag{3.13}$$

$$U(s,a) = c_{puct}P(s,a)\frac{e^{\tau\, log \sum_b N(s,b)}}{1+N(s,a)}. \tag{3.14}$$

This is a generalization of PUCT since for $\tau = 0.5$ this becomes the standard PUCT equation. The author shows that, with appropriate values for $\tau$, GPUCT outperforms PUCT, while also being less sensitive to the expansion budget.

### 3.1.3. Novelty and MCTS

Another proposal on improving action selection in MCTS was presented by Baier and Kaisers (2021) [2]. While they don't specifically improve the PUCT formula but rather generally improve the underlying UCT algorithm, their proposal remains applicable to PUCT. The authors suggest that searching for novelty rather than directly maximizing the search objective can facilitate the search by avoiding dead ends and local minima as well as generally improve exploration.

In order to integrate a novelty measure into MCTS the authors use a technique from *Rapid Action Value Estimation* (RAVE), where the exploitation part of the UCT formula is replaced by a linear combination of the estimated value of a node and a novelty average. Similar to the node evaluation returned by e.g. a neural network, a novelty score $N(s)$ is produced by a chosen novelty measure. Just like the node evaluation, the novelty score is backpropagated along the path taken during the selection phase using

$$N_a = N_a + \frac{N(s) - N_a}{N(s,a)}, \tag{3.15}$$

where $N_a$ is the average novelty and as aforementioned $N(s)$ is the novelty score of the state $s$, both not to be confused with the visit count of action $a$ in state $s$, denoted $N(s,a)$. With this additional parameter the action is selected by

$$A_t = \underset{a}{argmax} \left( bN_a + (1-b)Q_a + c\sqrt{\frac{log \sum_b N(s,b)}{N(s,a)}} \right), \tag{3.16}$$

where $b$ is a weighting coefficient governing the trade off between novelty and exploitation, given by

$$b = \sqrt{\frac{\beta}{3N(s,a) + \beta}}, \tag{3.17}$$

where $\beta$ is hyperparameter regulating the decay of $b$.

The authors showed that, especially for a larger number of simulations, their approached statistically significant outperforms baseline MCTS. Interestingly, they showed that the performance improvement did depend less on the chosen novelty measure and more on the domain.

## 3.2. Learned Planning Models

### 3.2.1. Sub Goal Trees

RL is typically used for trajectory optimization given a single goal represented by the reward function. Naturally, it is not suited for problems with multiple goals, where a chain of sub goals lead to the final goal. In order to apply RL to such problems Jurgenson et al. (2020) [7] propose a novel RL framework, based on a dynamic programming equation for the *all pairs shortest path* problem.

The authors key idea is to construct a goal based trajectory in a divide-and-conquer fashion. Meaning, given the start and the goal, predict a sub goal partitioning the trajectory into two sub segments, then recursively predict sub goals for each sub segment until the trajectory is completed. The obtained sub goal tree can be viewed as a general parametric structure for a trajectory, where e.g. a neural network can be used to predict a sub goal given its predecessor.

In a discrete action space they formulate *sub goal tree dynamic programming* (SGTDP) for a weighted graph with $N$ nodes and non negative costs $c(s, s')$ as follows:

$$V_0(s, s') = c(s, s'), \quad \forall s, s' : s \neq s', \tag{3.18}$$

$$V_k(s, s) = 0, \quad \forall s, \tag{3.19}$$

$$V_k(s, s') = \min_{s_m}\{V_{k-1}(s, s_m) + V_{k-1}(s_m, s')\}, \quad \forall s, s' : s \neq s', \tag{3.20}$$

$$V_k(s, s') = V^\star(s, s'), \quad for \;\; k \geq log_2(N), \tag{3.21}$$

where $V_k(s, s')$ corresponds to the cost of the shortest path from $s$ to $s'$ in $2^k$ steps and $V^\star(s, s')$ denotes the cost of the shortest path from $s$ to $s'$. Given $V_0, ..., V_{log_2(N)}$ the shortest path for every start state $s_0$ and goal state $s_N$ can then be formulated as the trajectory,

$$s_{N/2} \in \underset{s_m}{argmin}\{V_{log_2(N)-1}(s_0, s_m) + V_{log_2(N)-1}(s_m, s_N)\}, \tag{3.22}$$

$$s_{N/4} \in \underset{s_m}{argmin}\{V_{log_2(N)-2}(s_0, s_m) + V_{log_2(N)-2}(s_m, s_{N/2})\}, \tag{3.23}$$

$$s_{3N/4} \in \underset{s_m}{argmin}\{V_{log_2(N)-2}(s_{N/2}, s_m) + V_{log_2(N)-2}(s_m, s_N)\}, \tag{3.24}$$

$$\ldots$$

where $s_{N/2}$ optimally partitions the trajectory between $s_0$ and $s_N$ into two segments of length $N/2$ and subsequently, $s_{N/4}$ and $s_{3N/4}$ further partition the resulting two segments into optimal smaller segments of length $N/4$. This is recursively repeated until a complete trajectory is obtained.

With the described SGTDP algorithm we can now apply an RL algorithm in order to learn a function for estimating $V_k(s, s')$. Given a data set of random state transitions and their costs in form of the 3-tuples $\{s, s', c\}$ and an estimate of the value function $\hat{V}_k(s, s')$, e.g. represented by a neural network, then,

$$\hat{V}_{k+1}(s, s') = \underset{s_m}{min}\{\hat{V}_k(s, s_m) + \hat{V}_k(s_m, s')\} \tag{3.25}$$

can be used to generate targets for fitting the function approximator.

The framework presented by the authors shows that RL can successfully be utilized to do planning in a multi goal environment. Furthermore, they show that their approach provides statistically significant performance improvements to sequentially predicting the sub goals.

### 3.2.2. Planning with Goal-conditioned Policies

Model-based RL methods in combination with planning algorithms are powerful tools for complex and temporally extended decision making problems. However, model-free methods struggle with these tasks, since they can't utilize planning algorithms to encode for the temporal compositionality of the tasks, due to their lack of accurate models. To solve this Nasiriany et al. (2019) [12] propose a model-free RL planning framework, based on using goal-conditioned policies for temporal abstraction.

Goal-conditioned policies are trained to reach a goal state, which is given as an additional input. Since these policies typically can't reach long-distance goals, which require planning, the authors complemented them with a value function for estimating the reachability of a given goal. With this actor-critic style algorithm the agent utilizes the goal-conditioned policy to determine feasible sub goals in a learned representation of the state, while the goal-conditioned value function provides information of the reachability of that state.

The goal-conditioned value function is denoted as $V(s, g, t)$ and represents the reachability of goal $g$ from state $s$ in $t$ time steps, with $V(s, g, t) = 0$ if the goal is reachable. For $K$

intermediate sub goals this generalizes to the *feasibility vector*,

$$\vec{V}(s, g_{1:K}, t_{1:K+1}, g) = \begin{bmatrix} V(s, g_1, t_1) \\ V(g_1, g_2, t_2) \\ \dots \\ V(g_K, g, t_{K+1}) \end{bmatrix}, \tag{3.26}$$

where each element describes how close the policy will reach the sub goal starting from the previous sub goal. To create a feasible plan each element of this vector should be zero, so that every sub goal is reachable from its predecessor. Therefore, the optimization objective is to minimize the norm of the feasibility vector,

$$L(g_{1:K}) = ||\vec{V}(s, g_{1:K}, t_{1:K+1}, g)||. \tag{3.27}$$

Crucially, the authors don't plan on the raw states, but rather utilize state abstractions obtained with a variational-autoencoder (VAE). Instead of learning their temporal difference models for the policy and value function networks from scratch, they use the pretrained mean-encoder of their VAE and only train additional fully connected layers on top of this with RL.

With their work the authors present a step in the direction of combining model-free RL with planning. Furthermore, they show that their proposed algorithm can solve tasks, which are difficult to solve with conventional model-free methods.

# 4. Methodology

This works builds upon the *AlphaZero* implementation for the game Connect4 by Soh (2019) [18]. The major improvements to the base implementation consist of the addition of a replay buffer and a $L_2$ penalty on the weights to stabilize training as well as making all hyperparameters adjustable in the main pipeline for easier experimentation. On top of this basis, this work implements learned planning models to replace the PUCT action selection during the MCTS with a neural network evaluation.

Section 4.1 describes the basic *AlphaZero* implementation by Soh as well as our improvements to it. This includes the chosen input and output representation for the value-policy network, the architecture of said neural network and the reinforcement learning loop. Subsequently, in Section 4.2, the different planning models are presented, including the chosen input and output representations, the various neural network architectures and the process of initializing them to fit the PUCT formula with imitation learning. Lastly, Section 4.3 describes how the planning models are integrated into the search algorithm and how PPO is applied to optimize the planning models with RL in a self play loop.

## 4.1. AlphaZero for Connect4

Connect4 is a simple game in which two players take turns placing their stones in one of seven columns. Each column consists of six rows and stones always fall to the lowest possible row. The aim of the game is to connect four stones vertically, horizontally or diagonally before the opposing player does so. Figure 4.1 depicts an example board position, where the first player (white stones) wins by diagonally connecting four stones. Although, Connect4 is a solved game where the first player to move can always win with perfect play [1], it remains valuable for experimental RL applications, since the game can be rapidly simulated and RL algorithms don't quickly converge to optimal play.

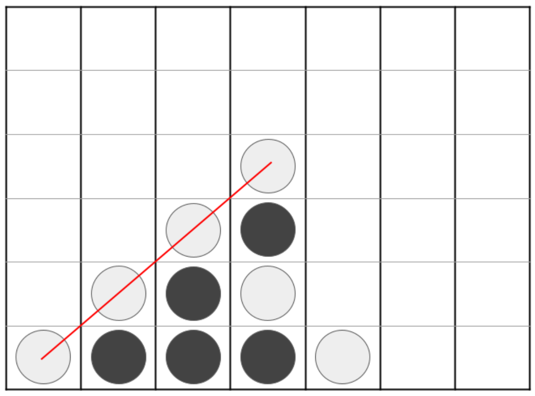


Figure 4.1.: This figure shows an example Connect4 game after eleven moves. Since the player with the white stones goes first, he has one more stone on the board than his opponent, winning the game by diagonally connecting four stones as indicated by the red line.

### 4.1.1. Replay Buffer and L2 Penalty

The aim of this work is not to train a model to converge to optimal play in Connect4, neither is the goal to evaluate general improvements to *AlphaZero* at the example of Connect4. For such efforts Allis (1988) [1] and Clausen et al. (2021) [4], respectively, can be referred to.

This work focuses on achieving an *AlphaZero* model with a descent playing strength in Connect4, in order to build and evaluate a planning model on top of this. While the basic implementation by Soh (2019) [18] could have sufficed for this task, we decided on adding a $L_2$ penalty on the weights to regularize training as it was intended by the original *AlphaZero* authors (see Section 2.2, Equation 2.7).

Furthermore, we make use of a replay buffer in order to prevent catastrophical forgetting and further regularize training. Each neural network update is conducted with a dataset consisting of two-thirds of self play data generated with the current neural network and one-third of self play data from previous iterations. The latter is sampled from a replay buffer containing the most recent 20% of data generated in previous iterations. This has the effect, that the replay buffer from which we sample grows with the number of iterations, which should help the model generalize rather than just learning to exploit its previous iteration.

### 4.1.2. Value-Policy Network

*AlphaZero* is a general RL algorithm applicable to any sequential decision making problem. The only components that need to be adjusted for use in different environments are the implementation of the environments dynamics for the search and the input and output representation for the value-policy network.

In line with the original *AlphaZero* implementation [15] our work employs a dual-headed value-policy network, $(\mathbf{p}, v) = f_\theta(s)$, within MCTS. It is trained by using the learning objective $l = (z - v)^2 - \pi^T log\, \mathbf{p} + c||\theta||^2$, with data generated in a self-play loop, as described in Section 2.2.

Considering the comparative simplicity of Connect4, our neural network consists of only 6 shared residual blocks, as opposed to the 19/39 blocks used in the original implementation. Each residual block consists of two convolutional layers with 128 filters connected by batch normalization and RELU activation. Additionally, a skip connection adds the input to the end of the block before the second RELU activation. The value head consists of one convolutional layer with three filters, followed by batch normalization and two fully connected layers with RELU activation in between and tangens hyperbolicus activation at the end. Lastly, the policy head consists of a convolutional layer with 32 filters, followed by batch normalization and one fully connected layer with RELU activation in between and softmax activation at the end.

#### Input Representation

Although, *AlphaZero* was designed with a zero knowledge approach in mind, the choice of input representation presents an opportunity to introduce domain knowledge by engineering the features.

Sticking to the zero knowledge approach we only used three planes of size $6x7$ (the standard board size) for our input representation. The first plane contains ones in all positions where player one has placed his stones and zeros for all other positions. Accordingly, the second plane contains ones in all positions where player two has placed his stones and zeros for all other positions. The last plane is filled with either ones or zeros indicating the current player to move, where zeros correspond to the first player and ones correspond to the second player.

While additional features like a move history or attention planes indicating the connected stones could have been used, the simplicity of the chosen environment justifies using an equally simple input representation. This claim is backed by our experiments showing that our model improves its playing strength (see Section 5.1).

#### Output Representation

The value head of the neural network outputs a scalar value $V_\theta(s) \in [-1, 1]$, where $V_\theta(s)$ represents the predicted value of state $s$ using a neural network with parameters $\theta$. A value of $V_\theta(s) > 0$ indicates a favorable position for player one, while a value of $V_\theta(s) < 0$ indicates a favorable position for player two.

The policy head outputs a vector of length seven, representing a probability distribution over the seven theoretically possible moves in each state, where each entry corresponds to the prior probability for the corresponding move. Illegal moves are masked with zeros before saving the priors to the node.

### 4.1.3. Reinforcement Learning

As aforementioned, the reinforcement learning of the value-policy network is done following the general concept explained in Section 2.2. However, it's worth mentioning a few details crucial for learning.

Firstly, the value-policy network is deterministic in its output given a fixed state, consequently, the games generated in self play are deterministic as well. To counteract this we add Dirichlet noise to the policy prior in the root node and use temperature sampling to select the move after each search, as suggested by Silver et al. (2017) [17].

With temperature sampling the first ten moves are sampled proportional to the exponential visit count for each move, using

$$\pi(a|s) = \frac{N(s,a)^{1/\tau}}{\sum_b N(s,b)^{1/\tau}}, \tag{4.28}$$

where $\tau$ is the temperature parameter. After ten moves $\tau$ is set to an infinitesimal number, having the effect of suppressing every action but the one with the highest visit count $N(s, a)$.

Dirichlet noise enforces additional exploration by ensuring that all moves may be tried, but the search will still overrule bad moves in the long term. Accordingly, the priors in the root node of the search are constructed as

$$\begin{aligned} P(s, a) &= (1 - \epsilon)\mathbf{p} + \epsilon\eta, \quad with \\ \eta &\sim Dir(\alpha), \end{aligned} \tag{4.29}$$

where **p** is the policy prior for the root obtained from the neural network. $Dir(\alpha)$ is the Dirichlet distribution with parameter $\alpha$ and $\epsilon$ is a hyperparameter governing the amount of noise added to the prior.

Secondly, each neural network update is performed after generating 500 self-play games. Afterwards, the updated model is evaluated against its previous version by playing 100 games. For these games we use a set of 50 predefined unique starting positions, which each model gets to play as either player. Only if the new model wins at least 55% of these evaluation games is it used for generating new self play data. Otherwise, the old model is used to generate another 500 games to increase the size of the training set. According to Silver et al. (2017) [17], this ensures that only the best quality training data is generated. Furthermore, this implicitly increases the size of each training set as soon as more data is needed to keep learning.

## 4.2. Planning Model

At the center of our work stands the idea to replace the PUCT action selection in MCTS with a learned planning model, in order to incorporate more information during the selection phase. In this section we propose six different planning models and describe the process of training them to predict promising states for the MCTS algorithm by utilizing multiple additional features compared to the PUCT formula.

### 4.2.1. I/O Representation

Our six planning models differ mainly in the choice of input features. Additionally to the standard PUCT components introduced in Equation 2.4, we propose three complementary features for our planning models, resulting in the following list of features:

| **Standard PUCT Components** | **Proposed new Features** |
|---|---|
| 1. Child Values, $Q(s,a)$ | 6. Q-Value Variance |
| 2. Child Priors, $P(s,a)$ | 7. 4-step Move History |
| 3. Child Visits, $N(s,a)$ | 8. Action Mask |
| 4. Parent Visits, $\sum_b N(s,b)$ | |

*Q-Value Variance* refers to the variance of the value predictions in the sub tree of any given node. In order to save memory and ensure numerical stability of its calculation we employ Welford's online algorithm for variance computation [21]. After initializing $M_1 = x_1$ and $\sigma_1^2 = 0$ the following update rules are used to compute the variance $\sigma_n^2$ after the $n$-$th$ sample $x_n$,

$$M_n = M_{n-1} + (x_n - \overline{x}_{n-1})(x_n - \overline{x}_n), \tag{4.30}$$

$$\sigma_n^2 = \frac{M_n}{n}, \tag{4.31}$$

where $\overline{x}_n$ denotes the mean of the value predictions in the current sub tree of the node for which the *Q-Value Variance* is computed. Accordingly, every node in our tree tracks the two additional metrics $M_n$ and $\overline{x}_n$.

The *4-step Move History* consists of the last four moves played in form of the input representation described in Section 4.1.2. Since this feature is represented in 12 planes of size 6x7 it requires the use of a convolutional neural network.

Lastly, the *Action Mask* is represented by a vector of length 7 containing zeros at the indices corresponding to illegal moves and ones for the legal moves.

With these features we construct six different planning models, $f_{\theta_k}(d_k) = \mathbf{v_a}$, where $d_k$ denotes the input representation for model $k$ and $\mathbf{v_a}$ is a vector of length 7 corresponding to the values for the 7 possible actions. Two of these models are simple *multi-layer perceptrons* (MLPs) and four are *convolutional residual neural networks* (CRNs). For all of these the standard PUCT components and the *Action Mask* are fixed inputs. By including different combinations of the remaining features we obtain the models described in Table 4.1.

Both MLPs consist of two hidden layers and the CRNs follow the same architecture described in Section 4.1.2, except that they are only comprised of two residual blocks and a single output head, which shares the same architecture as the value head of our *AlphaZero* model.

| | **PUCT Components** (22 inputs) | **Action Mask** (7 inputs) | **Q-Variance** (7 inputs) | **Move History** (12 inputs) | **Input Shape** |
|---|---|---|---|---|---|
| MLP Base | ✓ | ✓ | | | 1 x 29 |
| MLP QV | ✓ | ✓ | ✓ | | 1 x 36 |
| CRN Base | ✓ | ✓ | | | 29 x 6 x 7 |
| CRN QV | ✓ | ✓ | ✓ | | 36 x 6 x 7 |
| CRN MH | ✓ | ✓ | | ✓ | 41 x 6 x 7 |
| CRN QV/MH | ✓ | ✓ | ✓ | ✓ | 48 x 6 x 7 |

Table 4.1.: This table shows the six different planning models with their corresponding input features and the resulting input shape. CRN is short for convolutional residual neural network and MLP stands for multi layer perceptron.

### 4.2.2. Imitation Learning

Before using PPO to optimize the planning models we pretrain them to approximately fit the PUCT output by imitation learning. For this we use our *AlphaZero* base model to generate data in a self play loop in the from of the tuple $\{d, A_t\}$, where $d$ holds all necessary inputs for each of the models and, $A_t = \underset{a}{argmax}(Q(s_t, a) + U(s_t, a))$, is the action chosen during the selection phase at time $t$ according to Equation 2.4. Note that we chose the action selected by the search, rather than the one with the highest PUCT value as our target, which means that this is ensured to be a legal action.

To pretrain the planning models we sample from these data points to obtain a balanced subset of the generated data, 80% of which we use for training and 20% for validation. Finally, using the Adam optimizer with Cross Entropy Loss we fit the model to predict $A_t$ given $d$.

## 4.3. Optimizing the Planning Model

With the initialized planning models mirroring the behaviour of PUCT, this section focuses on improving their capabilities. To accomplish this the models are first integrated into the MCTS algorithm and then used to generate self play data for PPO.

### 4.3.1. Integration in MCTS

The planning models are integrated in the selection phase of MCTS. Therefore, at each time step $t$ during the selection phase we construct the input $d_k(t)$, for the currently used planning model $k$, by either concatenating the relevant features to a vector or filling the plane representation with them for the MLP or CRN, respectively. With this, the planning model, $f_{\theta_k}(d_k(t))$, outputs the predicted action value vector $\mathbf{v_a}(t)$. Afterwards, the chosen action is determined by

$$A_t = \underset{a}{argmax}(\mathcal{I}[\mathbf{v_a}(t)]), \quad with \tag{4.32}$$
$$\mathcal{I}[\mathbf{v_a}(t)] = \begin{cases} \mathbf{v_a}(t), & \text{if } a \text{ is legal,} \\ 0, & \text{if } a \text{ is illegal.} \end{cases}$$

Although, the planning models are pretrained to predict legal actions, it can't be assured that $\underset{a}{argmax}(\mathbf{v_a}(t))$ corresponds to a legal action. Hence, we apply $\mathcal{I}(x)$, which evaluates to the identity function for legal actions and to $0$ for illegal actions.

The above process is repeated until $A_t$ corresponds to an unexpanded child in the search tree, at which point the algorithm proceeds in normal MCTS fashion as described in Section 2.2.

### 4.3.2. Reinforcement Learning

With the planning models integrated in MCTS we can now use them to generate training data for PPO. Similar to the data generation for imitation learning this is accomplished by using a self play loop, resulting in data tuples of the form $\{d(t), G\}$. However, instead of using the selected action $A_t$ as a target, we now use the reward $G \in \{-1, 0, 1\}$. At the end of each game $G$ is determined for each data point by evaluating the game outcome from the perspective of the player at time $t$. In other words, if during the selection phase at time $t$ the corresponding node in the tree was one where player one had to chose an action and player two ended up winning the game, then $G$ would evaluate to $-1$.

Each iteration the planning model is updated using the data generated by 20 self play games. Although, this is a small number of games compared to the 500 games per iteration for the *AlphaZero* base model, the number of data points generated from these 20 games is significantly larger. This is the case since a data point is generated each time a node is selected during the selection phase. Per game this results in the number of data points on average being

$$\overline{N}_d = \overline{N}_m * N_e * \overline{D}_s, \tag{4.33}$$

where $\overline{N}_m$ is the average number of moves in a game, $N_e$ is the number of MCTS expansions per move and $\overline{D}_s$ denotes the average depth reached during the selection phase. Assuming $\overline{N}_m = 25$, $N_e = 200$ and $\overline{D}_s = 4$, each self play game generates 20.000 data points for the planning model, compared to just 25 data points for the base model.

Each update is conducted using a modified version of the $L^{CLIP}$ objective from Equation 2.9,

$$\begin{aligned} L^{CLIP}(\theta) &= \hat{\mathbb{E}}_t\left[min(r_t(\theta)G,\, clip(r_t(\theta),\, 1-\epsilon,\, 1+\epsilon)G\right],\ with \\ r_t(\theta) &= \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}, \end{aligned} \tag{4.34}$$

where $r_t(\theta)$ is only taken into account for computing the empirical average if the action $a_t$ is a legal action.

# 5. Empirical Evaluation

In this chapter, we evaluate all steps of our training pipeline, starting with a validation of our *AlphaZero* implementation for Connect4. Afterwards, we evaluate the planning models initialized with imitation learning. Lastly, we report the results of our final optimized planning models.

All evaluations are conducted by pitting the respective models against each other in a round robin tournament. To guarantee that only the best moves according to the models are played we disabled temperature sampling. Since this leads to deterministic games between any two models given the initial position, we employed a set of up to 50 unique starting positions to ensure variation in the games. Each starting position has up to four moves pre-applied to the board. To counteract that some of these starting positions heavily favor one of the players, each model gets to play the same position as either side against each other model.

## 5.1. Validating the AlphaZero Implementation

We trained the base *AlphaZero* model for a total of 40 iterations, which took 13 days, using a single GPU on which we ran two processes to generate games in parallel. In total, we generated 69 thousand games, which represent about 1.2 - 1.75 million training samples. As a result of the update rule described in Section 4.1.3, each iteration used between 500 and 4000 games for the eventual model update, before moving to the next iteration. Figure 5.1 shows the number of games needed for a model update to surpass 55% win ratio against the model of the previous iteration, indicating that the longer our model trained the more games were needed for further improvement of the model.

To evaluate the training process we played a round robin tournament with the models from iterations 0, 10, 20, 30 and 40, where each model started with an elo rating of 400.

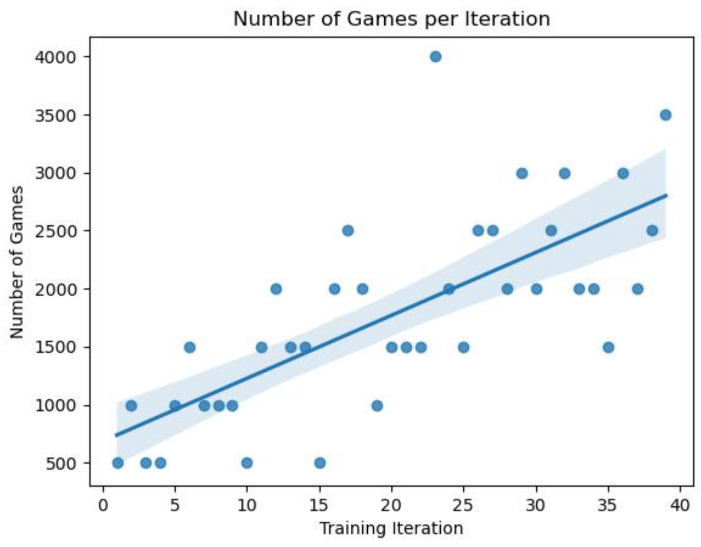


Figure 5.1.: This figure shows the number of games needed for a model update to surpass 55% win ratio against the model from the previous iteration.

Figure 5.2 shows that our model steadily improves with the exception of the model at iteration 10. We identify three possible reasons that could explain this outlier. Firstly, the number of games played in the tournament could have been insufficient to accurately reflect the real playing strength of the models. Looking at the error margin for the elo calculation indicated in Figure 5.2, we see that this might indeed be an artefact of statistical fluctuation. Secondly, the outlier could be caused by the model 10 losing some small number of positions against the model 0 while the latter had substantially lower elo, followed by the model 0 winning some small number of positions against substantially higher rated models. Thirdly, the pre-predefined starting positions could by chance contain a high number of positions where model 10 is exploitable by model 0.

In absolute numbers the model from iteration 10 won a total of 140 games and lost 259 games in this tournament, while the model from iteration 0 won 90 and lost 310 games. Hence, we assume that the elo rating for these two models doesn't reflect their actual relative playing strength as a result of the reasons mentioned above.

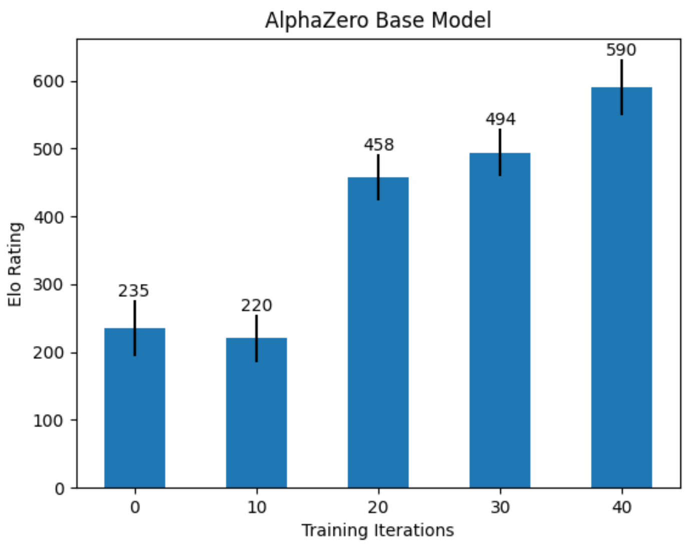


Figure 5.2.: This figure shows the elo development of our base model relative to its previous training iterations. Iteration 0 corresponds to the randomly initialized neural network. The round robin tournament was played with 50 pre-defined starting positions, where each model gets to play each starting position as either player against each other model. Every model started with an elo rating of 400.

## 5.2. Evaluating the Initialized Planning Models

To initialize the planning models, we used our best *AlphaZero* base model to generate 100 self play games, yielding 2.034.726 data points. From these we randomly sampled 1.000.000 data points for training and another distinct 250.000 for validation. Table 5.1 shows the highest validation accuracy the models reached and the corresponding epoch. Training the MLPs took approximately 1 hour each and training the CRNs took approximately 3-5 hours each, depending on the number of input features.

Although, the training process did not show clear signs of overfitting, we stopped at the indicated epochs since the increases in validation accuracy became marginal and the validation loss began to fluctuate heavily.

| | **MLP Base** | **MLP QV** | **CRN Base** | **CRN QV** | **CRN MH** | **CRN QV/MH** |
|---|---|---|---|---|---|---|
| Validation Accuracy (in %) | **84.65** | **84.95** | 79.54 | 79.52 | 78.78 | 78.38 |
| Epochs | 43 | 53 | 23 | 27 | 28 | 27 |

Table 5.1.: This table shows the highest validation accuracy reached during imitation learning for our planning models. Furthermore, it indicates the corresponding number of epochs they were trained for.

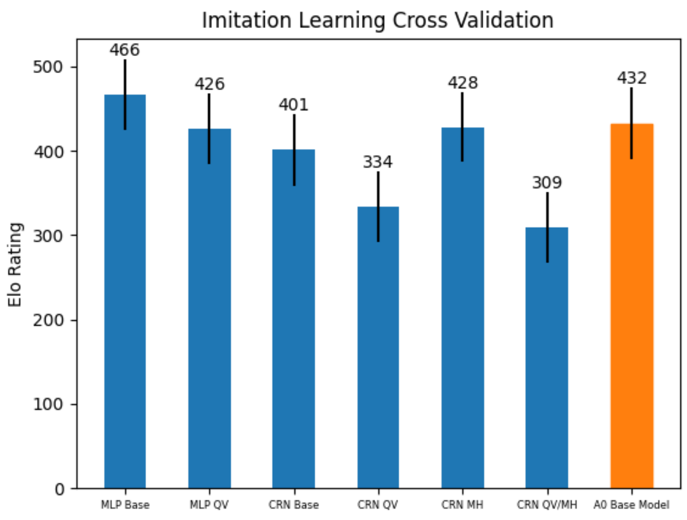


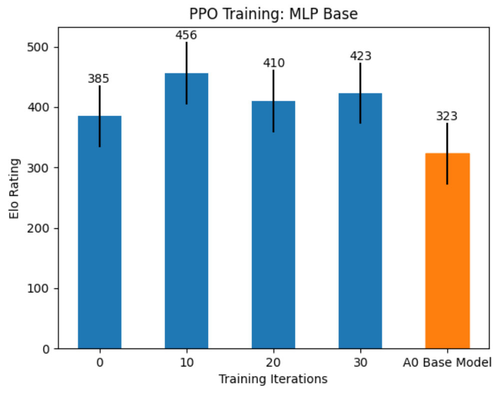
Figure 5.3.: This figure shows the relative elo of our initialized planning models and our best *AlphaZero* base model. The round robin tournament was played with 20 pre-defined starting positions, where each model gets to play each starting position as either player against each other model. Every model started with an elo rating of 400.

To evaluate the resulting models we again used a round robin tournament setup containing all initialized planning models as well as our best performing *AlphaZero* base model. Figure 5.3 indicates that most of the models are relatively equal in playing strength except for the CRN QV and CRN MH/QV. This could indicate one of two things, either the *Q-Value Variance* feature hinders learning when combined with a CRN or the bias introduced by the used pre-defined starting positions skewed these results.

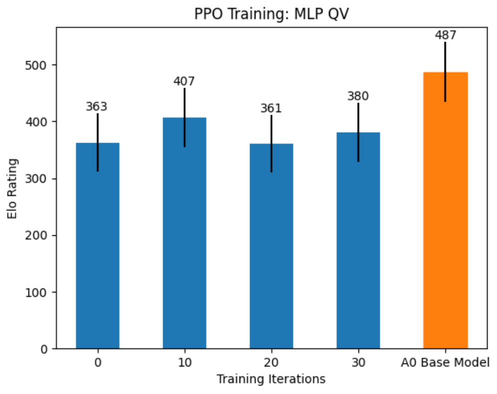
## 5.3. Evaluating the Optimized Planning Models

Starting with the initialized planning models we applied PPO with data generated in a self play loop to optimize these models. In conjunction with our best *AlphaZero* base model each of the planning models was used to generate 20 self play games per iteration. This resulted in 250 to 500 thousand data points per model update. Each model was updated for 30 iterations, taking a total of 15-30 hours, depending on the number of input features and the model architecture.

### 5.3.1. MLP Planning Models



(a) MLP Base



(b) MLP QV

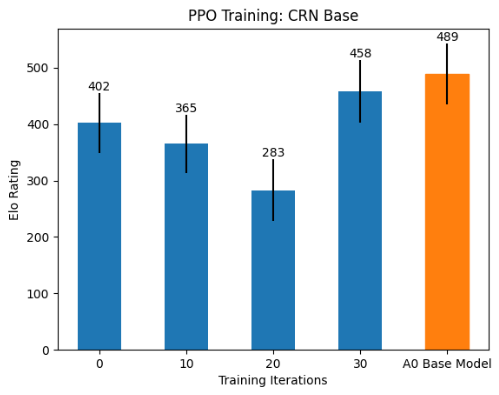
Figure 5.4.: This figure shows the relative elo development of our MLP planning models and our best *AlphaZero* base model. The round robin tournaments for (a) and (b), respectively, were played with 20 pre-defined starting positions, where each model gets to play each starting position as either player against each other model. Every model started with an elo rating of 400. Training iteration 0 corresponds to the with imitation learning initialized planning model.

Figure 5.4 shows the results from two round robin tournaments played between the MLP planning models from training iterations 0, 10, 20, 30 and our best *AlphaZero* base model. These results indicate that none of the MLP planning models show any meaningful elo progression over their 30 iterations of training. Only after the first 10 iterations a slight improvement for both MLPs can be seen. Furthermore, all MLP Base iterations have a slightly higher playing strength than our *AlphaZero* base model.

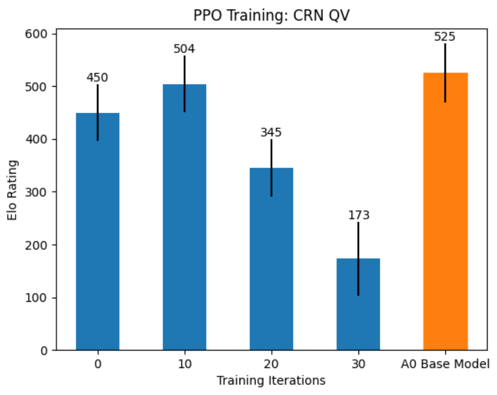
Whether or not these results reflect an actual improvement in the general capabilities of the planning models remains questionable at least, since the deviations are subtle, have high uncertainty and don't follow a clear upward trend.

### 5.3.2. CRN Planning Models

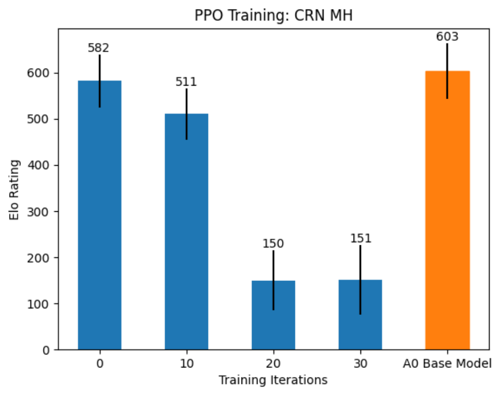
Figure 5.5 shows the results from the four round robin tournaments played between the CRN planning models from training iterations 0, 10, 20, 30 and our best *AlphaZero* base model. In these results we observe a clear downward trend in playing strength for all planning models that use our proposed additional features *Q-Value Variance* and the *4-step Move History*. Solely the CRN Base model decreases in playing strength for the first 20 iterations and then increases again in the following 10 iterations of training. However, all of the models remain below the playing strength of our best *AlphaZero* base model at all times.



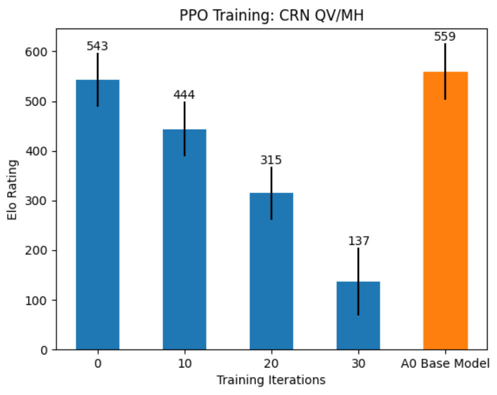
(a) CRN Base



(b) CRN QV



(c) CRN MH



(d) CRN QV/MH

Figure 5.5.: This figure shows the relative elo development of our CRN planning models and our best *AlphaZero* base model. The round robin tournaments for (a), (b), (c) and (d), were played with 20 pre-defined starting positions, where each model gets to play each starting position as either player against each other model. Every model started with an elo rating of 400. Training iteration 0 corresponds to the with imitation learning initialized planning model.

### 5.3.3. Search Speed Comparison

Integrating the planning models into MCTS essentially replaces a simple vector computation with a neural network evaluation at each step during the selection phase. Therefore, given a fixed number of expansions as our search budget, these planning models substantially increase the time each search takes. Table 5.2 shows that even the smallest planning model we used only accomplishes $\frac{1}{2}$ and the biggest model only $\frac{1}{6}$ of the speed of our *AlphaZero* base model.

| | **A0 Base Model** | **MLP Base** | **MLP QV** | **CRN Base** | **CRN QV** | **CRN MH** | **CRN QV/MH** |
|---|---|---|---|---|---|---|---|
| Nodes per Second (NPS) | **403±69** | 190±53 | 196±49 | 82±35 | 85±35 | 67±50 | 65±51 |

Table 5.2.: This table shows the search speed of our planning models compared to the search speed when using only the base model. Speeds are given in nodes per second (NPS) and correspond to the nodes expanded per second of running MCTS. Less nodes per second equal more time spent searching, when considering a fixed number of expansion as our budget.

### 5.3.4. Search Distribution in the Initial Position

Connect4 is a game that can always be won by the first player if and only if the first stone is placed in the middle row. Given this first move, the perfect game ends after 41 moves. Given any other first move, the second player can always force a draw after 42 moves with perfect play [1]. To evaluate whether or not our models tend to recognize this we evaluate the first search in the root node of the initial position for each model.

Figure 5.6 indicates that our *AlphaZero* base model shows a clear tendency of placing the first stone in the middle row (index 3). Although, our model did not learn to play the perfect game, this suggests that our model would probably converge there if it was trained further. As seen in Figure 5.7 our initialized MLPs almost perfectly mirror the distribution of the base model. However, the initialized CRNs appear more deterministic. With the exception of the CRN QV/MH, all of them show no exploration at all and always chose the middle row (Figure 5.8). The CRN QV/MH version changes the behavior completely and chooses index 4 most of the time, which is in line with our previous experiment showing that the CRN QV/MH has the lowest playing strength of all the initialized models.

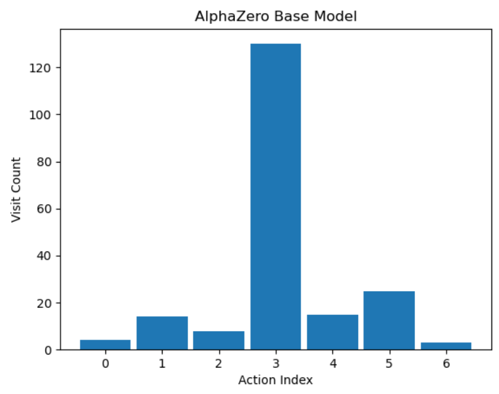
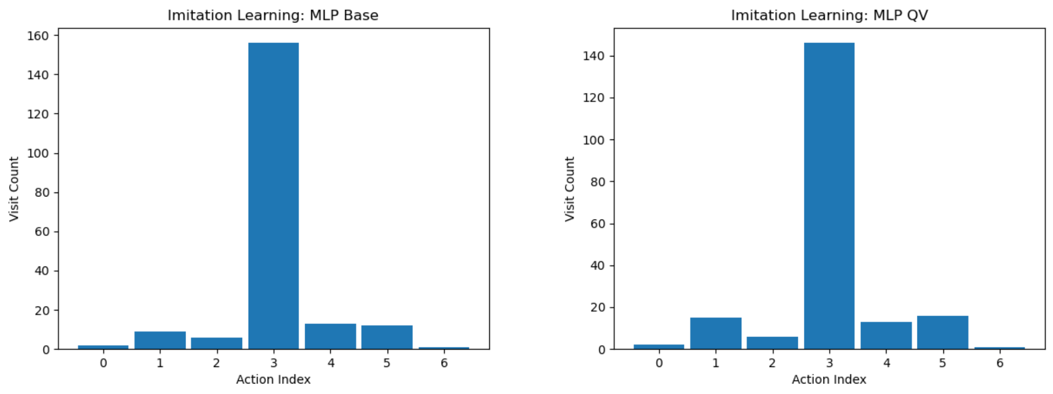


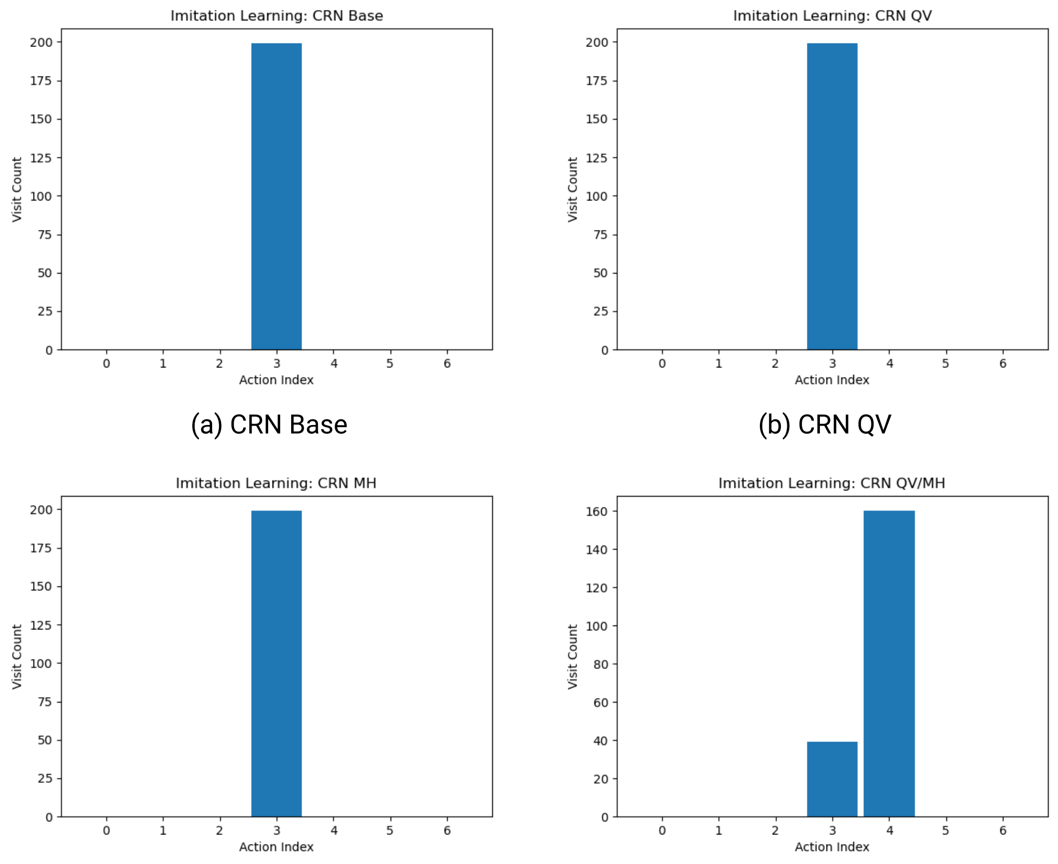
Figure 5.6.: This figure shows the search distribution of our best *AlphaZero* base model, given the initial position. The visit count indicates how often MCTS has chosen the respective action indicated on the X-axis.



(a) MLP Base

(b) MLP QV

Figure 5.7.: This figure shows the search distribution of our initialized MLPs, given the initial position. The visit count indicates how often MCTS has chosen the respective action indicated on the X-axis



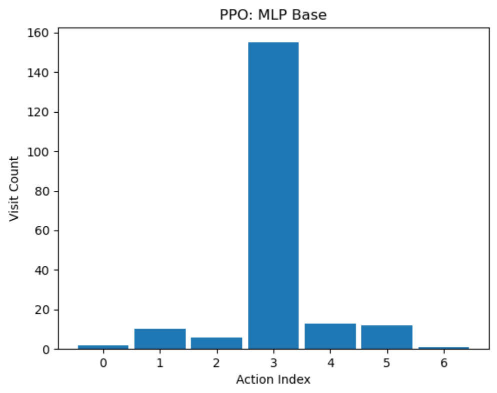
(a) CRN Base (b) CRN QV

(c) CRN MH (d) CRN QV/MH

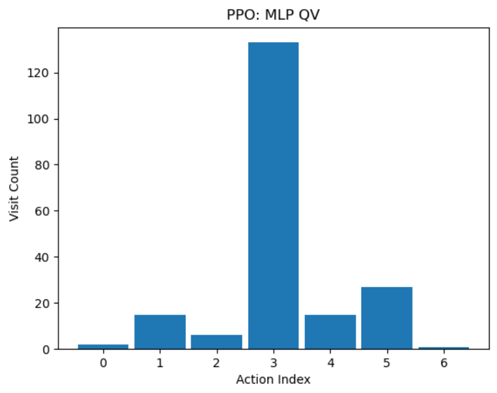
Figure 5.8.: This figure shows the search distribution of our initialized CRNs, given the initial position. The visit count indicates how often MCTS has chosen the respective action indicated on the X-axis

Conducting the same experiment with the final planning models after 30 iterations of PPO training shows that the MLPs stayed relatively constant in their evaluation of the initial position (see Figure 5.9). However, the CRNs show varying behavior. The three versions of the planning model that deterministically chose the correct row after they were initialized, now deterministically choose other actions. Interestingly, the CRN QV/MH now chooses the optimal first move (see Figure 5.10). However, we assume that this is due to random chance and not because the model has actually improved, since the evaluation in Section

5.3 does not show an improvement in playing strength for this particular model.

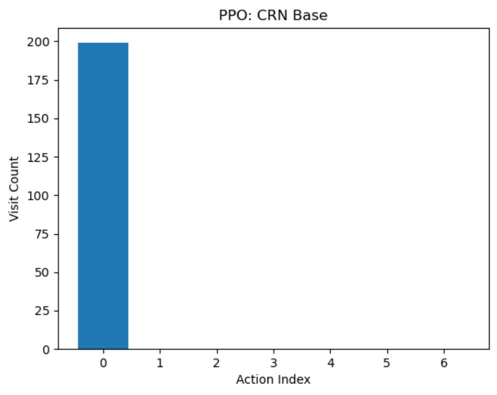


(a) MLP Base

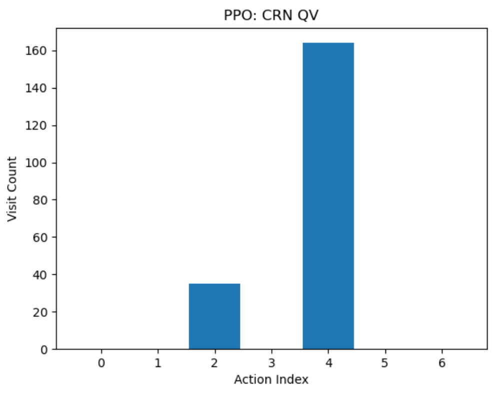


(b) MLP QV

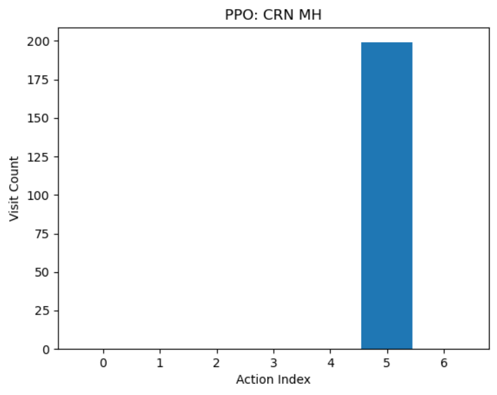
Figure 5.9.: This figure shows the search distribution of our MLPs trained with PPO for 30 iterations. The visit count indicates how often MCTS has chosen the respective action indicated on the X-axis



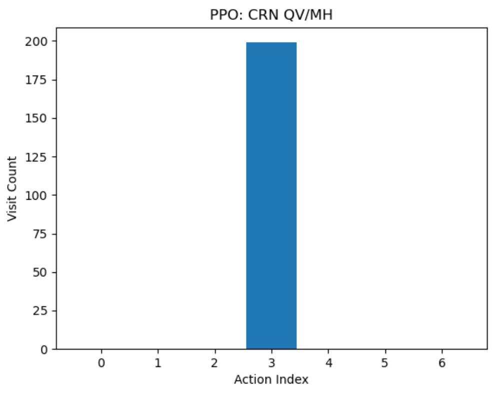
(a) CRN Base



(b) CRN QV



(c) CRN MH



(d) CRN QV/MH

Figure 5.10.: This figure shows the search distribution of our CRNs trained with PPO for 30 iterations. The visit count indicates how often MCTS has chosen the respective action indicated on the X-axis

# 6. Discussion

We start this chapter by discussing the sufficiency of our *AlphaZero* base model for the experiments we build on it. Afterwards, we discuss whether fitting the planning models to the PUCT formula yields appropriately initialized models. Lastly, we examine our approach to optimize the planning models with PPO and generally present potential improvements and concerns with our approach.

**AlphaZero Base Model** The empirical evaluation in Section 5.1 clearly shows, that our reduced *AlphaZero* implementation with only 6 residual blocks and a simple input representation, containing no domain knowledge, is sufficient for learning to play Connect4 from scratch. Nevertheless, neither did our model converge to optimal play, nor did it show signs of reaching its maximum potential. However, we argue that the benefit of replacing PUCT with a planning model would be in improving playing strength after the base model fails to improve further. Therefore, to test this hypothesis, we would have needed a base model that has converged to its maximum potential.

**Initializing with Imitation Learning** Our results from Section 5.2 indicate that using a neural network to clone the behaviour of PUCT action selection works well and results in agents of similar strength to our *AlphaZero* base model. Furthermore, we are confident that more hyperparamter tweaking and longer training times would enable the initialized planning models to behave exactly like PUCT action selection. Although it may be possible to initialize the planning model in different ways or even use randomly initialized planning models to start with, given our findings, we argue that our approach is appropriate for the intended purpose.

**Optimizing with PPO** For this work we formulated two main research questions: (1) action selection during tree search could be improved by replacing PUCT with a neural network, resulting in better agents, (2) action selection could benefit from additional features, in particular the *Q-Value Variance* and a *4-step Move History*. Due to the short time horizon of this work combined with the long run times of training and evaluating RL models, we could not run sufficient experiments. Therefore, we deem the empirical evaluation in Section 5.3 inconclusive with respect to our first research question. On the one hand, the CRNs show a clear downward trend in playing strength over their training, with the exception of the CRN Base version. On the other hand, the MLPs stay reasonably constant in playing strength over the course of their training. We identify multiple possible reason for these observations.

Firstly, although the RL loop for optimizing the planning models generates plenty of data points for each self play game played, as indicated by Equation 4.33, the reward for these data points is sparse and highly noisy. In the mean, each game generates approximately 40.000 data points, which all get combined with a single reward obtained from the end of the game to form a training sample. Since we only used 20 games per model update, it is conceivable, that this does not carry sufficient reward information for learning.

Secondly, due to the limited time we did not run a full hyperparameter search. As RL in general and PPO in particular are known to have stability issues and can be very sensitive to the hyperparameters, it remains possible that a different set of hyperparameters would have yielded different results to what our empirical analysis indicates.

Thirdly, we used the raw $L^{CLIP}$ objective to update our models. However, in practice and as suggested by Schulman et al. (2017) [14], this objective can be combined with a value function error term, when incorporated in an actor critic style algorithm, and further augmented by an entropy term.

Lastly, while we extensively tested every part of our algorithm, we can't assure that our results aren't an artefact of an implementation error.

With respect to our second research question our empirical evaluation indicates that additional features don't yield any benefit compared to the standard PUCT components. In all of our experiments the versions of the planning model using only the standard PUCT components outperform the versions that make use of our proposed additional features. However, since our planning models did not show any improvements, this might change if one accomplishes an algorithm that manages to improve the learning models.

**Search Speed** Considering that the use of planning models results in a drastic slowdown of the search, as shown in Section 5.3.3, the practicality of this approach is questionable. Furthermore, bandit theory is an already well researched field with proves of convergence for the here relevant algorithms. Additionally, in Section 3.1 we show that current research accomplishes improvements to PUCT action selection with seemingly better convergence characteristics without using neural networks.

# 7. Conclusion

## 7.1. Summary

In this work we proposed an approach to improve action selection in MCTS by introducing learned planning models to predict the next node to select. Furthermore, we proposed multiple novel features for the action selection.

As a basis for our approach we trained an *AlphaZero* agent to play the game of Connect4 and show that it improves its playing strength over the course of training. Using this base model we showed that it is possible to train different planning models with various input features to behave like PUCT action selection without significantly losing playing strength compared to the base model.

Lastly, we conducted experiments to optimize these planning models with PPO. As our experiments were inconclusive with respect to the formulated research questions we conclude that this topic requires further research to find a definitive answer. However, we layed a basis for future work by developing the pipeline needed to conduct further experiments.[1]

## 7.2. Future Work

**Hyperparameter search** Since we only tried a small set of hyperparameters we suggest that a full hyperparameter search may help to find a setup that accomplishes improvement of the planning models. Given that RL algorithms are prone to instability when using the wrong hyperparameters, this might be the most promising option to yield positive results. As discussed in the previous chapter we suggest increasing the number of games per

[1] `https://github.com/AdrianGlauben/Masterthesis`

iteration, in order to utilize more reward information for the model updates. Furthermore, to save time on the hyperparameter search we suggest to employ population based training as proposed by Jaderberg et al. (2017) [6].

**I/O representation** Our chosen I/O representation uses all information present in the current node to simultaneously predict the probabilities for all possible child nodes. This approach is appropriate for environments were the number of legal moves stays mostly constant in all states, however, many environments have a varying or even continuous action space given the current state. Therefore, the planning models would need to be adjusted for this fact to preserve the general applicability of our proposed algorithm. One possible way of accomplishing this would be to use a planning model that predicts only the probability of one of the possible actions and apply it multiple times at each step during the selection phase. First test results with this approach showed that this yields a better convergence to the PUCT action selection during imitation learning. However, this would increase the number of neural network evaluations needed during search linearly with the branching factor.

**Neural network architecture** In our work we used two types of neural network architectures, MLPs and CRNs. To speed up training our neural networks used only a few layers with a small number of parameters. We suggest to try bigger networks and different architectures to better cope for the complexity of the problem. One particularly interesting choice would be transformer models [20], as they recently found wide spread adoption and success in multiple different areas of research.

**Integrating base model and planning model** For simplicity we chose to separate learning the planning model from learning the base model. In other words, we first learn a base model and then, using the fixed base model, we learn the planning model. Finding a way to integrate learning the planning model with the learning of the base model would increase sample efficiency and reduce the over all training time. However, it is unclear if this approach is even viable since the planning model as we define it needs a trained base model for initialization.

# Bibliography

[1] Victor Allis. *A knowledge-based approach to connect-four. The game is solved: White wins*. M.Sc. Thesis, Vrije Universiteit Amsterdam, Faculty of Mathematics and Computer Science, 1998.

[2] Hendrik Baier and Michael Kaisers. "Novelty and MCTS". In: *GECCO '21: Proceedings of the Genetic and Evolutionary Computation Conference Companion* (2021), pp. 1483–1487. DOI: 10.1145/3449726.3463217.

[3] Tristan Cazenave. "Improving Model and Search for Computer Go". In: *2021 IEEE Conference on Games (CoG)* (2021), pp. 1–8. DOI: 10.1109/CoG52621.2021.9619078.

[4] Colin Clausen et al. "Improvements to Increase the Efficiency of the AlphaZero Algorithm: A Case Study in the Game 'Connect 4'". In: *13th International Conference on Agents and Artificial Intelligence* (2021), pp. 803–811. DOI: 10.5220/0010245908030811.

[5] Ivo Danihelka et al. "Policy improvement by planning with Gumbel". In: *International Conference on Learning Representations*. 2022. URL: https://openreview.net/forum?id=bERaNdoegnO.

[6] Max Jaderberg et al. "Population Based Training of Neural Networks". In: *arXiv:1711.09846* (2017).

[7] Tom Jurgenson et al. "Sub-Goal Trees a Framework for Goal-Based Reinforcement Learning". In: *Proceedings of the 37th International Conference on Machine Learning*. Ed. by Hal Daumé III and Aarti Singh. Vol. 119. Proceedings of Machine Learning Research. PMLR, 2020, pp. 5020–5030. URL: https://proceedings.mlr.press/v119/jurgenson20a.html.

[8] Zohar Karnin, Tomer Koren, and Oren Somekh. “Almost Optimal Exploration in Multi-Armed Bandits”. In: *Proceedings of the 30th International Conference on Machine Learning*. Ed. by Sanjoy Dasgupta and David McAllester. Vol. 28. Proceedings of Machine Learning Research. Atlanta, Georgia, USA: PMLR, 2013, pp. 1238–1246. URL: `https://proceedings.mlr.press/v28/karnin13.html`.

[9] Levente Kocsis and Csaba Szepesvári. “Bandit Based Monte-Carlo Planning”. In: *Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds) Machine Learning: ECML 2006. ECML 2006. Lecture Notes in Computer Science* 4212 (2006), pp. 282–293. DOI: `10.1007/11871842`.

[10] Wouter Kool, Herke van Hoof, and Max Welling. “Stochastic Beams and Where To Find Them: The Gumbel-Top-k Trick for Sampling Sequences Without Replacement”. In: *Proceedings of the 36th International Conference on Machine Learning*. Ed. by Kamalika Chaudhuri and Ruslan Salakhutdinov. Vol. 97. Proceedings of Machine Learning Research. PMLR, 2019, pp. 3499–3508. URL: `https://proceedings.mlr.press/v97/kool19a.html`.

[11] Tor Lattimore and Csaba Szepesvári. *Bandit algorithms*. Cambridge University Press, 2020. DOI: `10.1017/9781108571401`. URL: `https://tor-lattimore.com/downloads/book/book.pdf`.

[12] Soroush Nasiriany et al. “Planning with Goal-Conditioned Policies”. In: *Advances in Neural Information Processing Systems*. Ed. by H. Wallach et al. Vol. 32. Curran Associates, Inc, 2019. URL: `https://proceedings.neurips.cc/paper/2019/file/c8cc6e90ccbff44c9cee23611711cdc4-Paper.pdf`.

[13] Christopher D. Rosin. “Multi-armed bandits with episode context”. In: *Annals of Mathematics and Artificial Intelligence* 61.3 (2011), pp. 203–230. ISSN: 1012-2443. DOI: `10.1007/s10472-011-9258-6`.

[14] John Schulman et al. “Proximal Policy Optimization Algorithms”. In: *arXiv:1707.06347v2* (2017).

[15] David Silver et al. “A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play”. In: *Science (New York, N.Y.)* 362.6419 (2018), pp. 1140–1144. DOI: `10.1126/science.aar6404`.

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

[17] David Silver et al. “Mastering the game of Go without human knowledge”. In: *Nature* 550.7676 (2017), pp. 354–359. DOI: `10.1038/nature24270`.

[18] Wee Tee Soh. *From-scratch implementation of AlphaZero for Connect4*. Towardsdatascience, 2019. URL: `https://towardsdatascience.com/from-scratch-implementation-of-alphazero-for-connect4-f73d4554002a`.

[19] Richard S. Sutton and Andrew Barto. *Reinforcement learning: An introduction*. Second edition. Adaptive computation and machine learning. Cambridge, Massachusetts and London, England: The MIT Press, 2018. ISBN: 9780262039246.

[20] Ashish Vaswani et al. “Attention is All you Need”. In: *Advances in Neural Information Processing Systems*. Ed. by I. Guyon et al. Vol. 30. Curran Associates, Inc, 2017. URL: `https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf`.

[21] B. P. Welford. “Note on a Method for Calculating Corrected Sums of Squares and Products”. In: *Technometrics* 4.3 (1962), pp. 419–420. ISSN: 00401706. DOI: `10.2307/1266577`.

# Appendix

## A. Hardware

All experiments and training have been conducted on a *NVIDIA DGX-2* server instance, which features 16 *NVIDIA Tesla V100* GPUS with 32GB HBM2 memory each and 8 *Intel Xeon Platinum 8174* processors with a total of 96 CPU cores. For any training or experiment we used 1 GPU and 2 CPU threads to simultaneously generate 2 games on the same GPU.

## B. Implementation Details

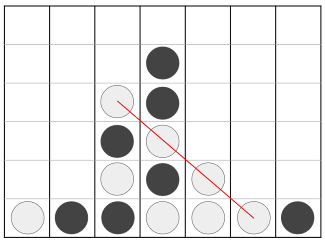
| Parameters | A0 Base Model | MLPs | CRNs |
|---|---|---|---|
| Learning Rate | 1e-3 \| 1e-4 \| 5e-5 | 1e-4 | 1e-3 |
| Epochs | 6 \| 2 | 1 | 1 |
| Batch Size | 32 | 1024 | 1024 |
| $c_{puct}$ | 3 | - | - |
| Expansions | 300 | 200 | 200 |
| Temperature $\tau$ | 1.1 | 1.1 | 1.1 |
| L2 Penalty | 1e-4 | 1e-4 | 1e-4 |
| Dirichlet $\alpha$ | 0.3 | 0.3 | 0.3 |
| Dirichlet $\epsilon$ | 0.25 | 0.25 | 0.25 |

Table B.1.: For the *AlphaZero* base model we lowered the learning rate two times during training, at iteration 20 and at iteration 30. Furthermore, we lowered the number of epochs at iteration 30. For all models we used temperature sampling with the given $\tau$ for the first ten moves and then dropped $\tau$ to an infinitesimal value.

# C. Example Games

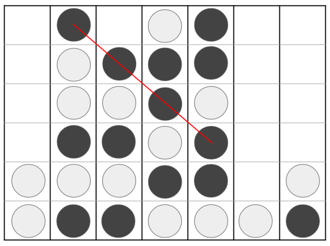
## C.1. AlphaZero Base Model

### AlphaZero Base Model Self Play

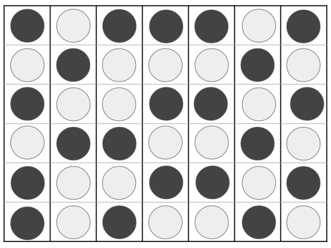
| | |
|---|---|
| Result: | Player one wins |
| Number of Moves: | 15 |
| Moves: | [3, 3, 4, 2, 3, 3, 2, 2, 2, 1, 5, 6, 0, 3, 4] |
| Ending Position: |  |

## C.2. Initialized Planning Models

### Imitation Learning MLP Base Self Play

| | |
|---|---|
| Result: | Player two wins |
| Number of Moves: | 28 |
| Moves: | [3, 3, 4, 2, 3, 3, 2, 2, 2, 1, 5, 6, 0, 3, 6, 4, 0, 4, 4, 4, 3, 4, 1, 1, 1, 2, 1, 1] |
| Ending Position: |  |

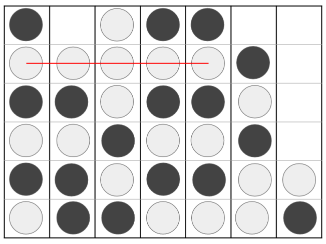
**Imitation Learning MLP QV Self Play**

| | |
|---|---|
| Result: | Draw |
| Number of Moves: | 42 |
| Moves: | [3, 3, 4, 2, 3, 3, 2, 2, 2, 5, 1, 0, 1, 1, 5, 5, 5, 5, 3, 3, 2, 2, 5, 4, 1, 1, 1, 0, 0, 0, 0, 0, 6, 6, 6, 6, 6, 6, 4, 4, 4, 4] |
| Ending Position: |  |

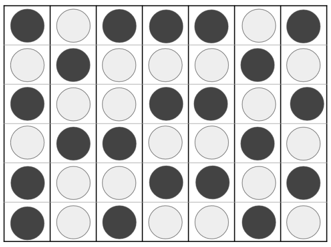
Given the initial position, all initialized CRNs play exactly the same self play game as the MLP QV model.

## C.3. Optimized Planning Models

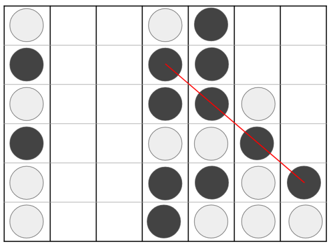
**PPO Trained MLP Base Self Play**

| | |
|---|---|
| Result: | Player one wins |
| Number of Moves: | 37 |
| Moves: | [3, 3, 4, 2, 3, 3, 2, 2, 2, 1, 5, 6, 0, 4, 3, 0, 4, 4, 4, 3, 5, 5, 5, 5, 2, 4, 0, 0, 0, 0, 2, 5, 6, 1, 1, 1, 1] |
| Ending Position: |  |

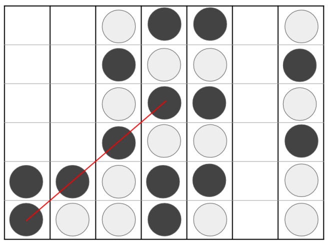
### PPO Trained MLP QV Self Play

| | |
|---|---|
| Result: | Draw |
| Number of Moves: | 42 |
| Moves: | [3, 3, 4, 2, 3, 3, 2, 2, 2, 5, 1, 0, 1, 1, 5, 5, 5, 5, 3, 3, 2, 2, 5, 4, 1, 1, 1, 0, 0, 0, 0, 0, 6, 6, 6, 6, 6, 6, 4, 4, 4, 4] |
| Ending Position: |  |

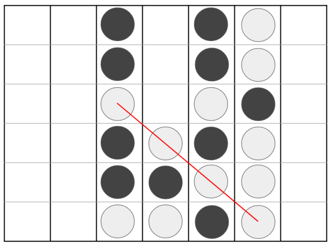
### PPO Trained CRN Base Self Play

| | |
|---|---|
| Result: | Player two wins |
| Number of Moves: | 24 |
| Moves: | [0, 3, 0, 0, 0, 0, 0, 3, 4, 4, 4, 4, 3, 3, 6, 4, 5, 3, 3, 4, 5, 5, 5, 6] |
| Ending Position: |  |

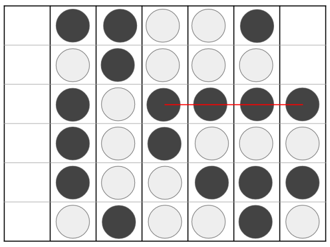
**PPO Trained CRN QV Self Play**

| | |
|---|---|
| Result: | Player two wins |
| Number of Moves: | 28 |
| Moves: | [4, 4, 4, 4, 4, 4, 2, 3, 2, 2, 2, 2, 2, 0, 6, 0, 6, 6, 6, 6, 6, 3, 3, 3, 3, 3, 1, 1] |
| Ending Position: |  |

**PPO Trained CRN MH Self Play**

| | |
|---|---|
| Result: | Player one wins |
| Number of Moves: | 21 |
| Moves: | [5, 4, 4, 4, 2, 2, 4, 4, 5, 2, 2, 2, 5, 5, 5, 2, 5, 4, 3, 3, 3] |
| Ending Position: |  |

## PPO Trained CRN QV/MH Self Play

| | |
|---|---|
| Result: | Player two wins |
| Number of Moves: | 34 |
| Moves: | [3, 2, 2, 5, 2, 5, 2, 2, 3, 3, 1, 3, 3, 2, 3, 1, 5, 5, 5, 5, 4, 4, 4, 4, 4, 1, 4, 1, 1, 1, 6, 6, 6, 6] |
| Ending Position: |  |