Master's Thesis

Investigation of possible improvements to increase the efficiency of the AlphaZero algorithm.

Christian-Albrechts-Universität zu Kiel
Institut für Informatik

written by: **Colin Clausen**
supervising university lecturer: Prof. Dr.-Ing. Sven Tomforde

Kiel, 10.8.2020

Selbstständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

.....
Colin Clausen, 10.8.2020

Contents

1 Introduction

Games have long been used as a substitute for the more complex real world in developing artificial intelligence (AI). Achieving superhuman performance in common games had been a goal of computer and AI research even before electronic computers became available.

An early example of AI beating humans at games is the Nimrod computer from 1951 [11], with work on similar machines starting in the 1940s [29], which could defeat players at the game Nim. Moving on to harder games requiring tree search algorithms with much more computational power, Connect 4 was completely solved in 1995 [1]. Chess was for a long time considered a major milestone, with Claude Shannon [30] noting how chess is an ideal ground for experimentation to investigate the possibilities of computing machines. In 1996 the chess computer Deep Blue for the first time defeated a reigning world champion in a single game, and in 1997 a full match under standard tournament rules was won [18].

In March 2016 a program called AlphaGo made history defeating a top human player in the board game Go [2]. Go had long eluded attempts at superhuman level play.

Louis Victor Allis attributed [13] this to the large game tree size of 10^{360} possible games compared to 10^{120} in chess [30], as well as to the way humans use their natural pattern recognition ability to quickly eliminate most of the 200 or more possible moves and focus on few promising ones.

This combination of the usage of ambiguous pattern recognition with an extremely large game tree has prevented computers from reaching top human strength through game tree search algorithms based on hand-designed heuristics. AlphaGo solved this issue by using the strong pattern-recognition abilities of deep learning and combining them with a tree search algorithm, allowing the computer to learn patterns, similar to a human, but also search forward in the game tree to find the best move to play.

Further development of the AlphaGo algorithm yielded the AlphaZero algorithm, which significantly simplified AlphaGo, allowing learning to start with a random network and obviating human expert input of any kind.

In the following thesis, possible improvements to reduce the computational cost of using AlphaZero to learn to play games will be investigated. Only proposals that do not lose the generality of AlphaZero will be considered.

This thesis is structured as follows:

Section 2 will discuss previous work on AlphaZero, starting with the basic tree search algorithm and ending with improvement proposals of other authors. Section 3 will describe the way new proposals are evaluated, establish various baselines, and implement a selection of improvements proposed by previous work. Finally, section 4 will discuss and evaluate proposed improvement ideas.

2 Previous work

This section will present previous work relevant to AlphaZero. This includes the introduction of the tree search algorithm used in AlphaZero, along with the description of the AlphaZero algorithm.

2.1 Monte Carlo Tree Search

Monte Carlo tree search, (MCTS) is based on the idea to search a large tree (e.g. a game tree) in a randomized fashion, gathering statistics on how good the expected return for moves at the root of the tree is.

An early suggestion of this idea came from Bernd Brügmann in 1993 [17], who proposed a tree search in a random fashion inspired by simulated annealing. He used the algorithm to play computer Go and found promising results on 9x9 boards compared to contemporary algorithms.

AlphaZero uses a variant of MCTS called UCT, which was formulated in 2006 [24] by L.Kocsis et al. and used in computer Go for the first time in the same year [20]. UCT stands for "UCB applied to trees", where UCB denotes the "Upper Confidence Bounds" algorithm of the *multi-armed bandit problem* [16]. Prior to AlphaZero, multiple other strong computer Go programs based on MCTS were released, such as Pachi [12] and CrazyStone [9].

In the *multi-armed bandit problem*, a player is faced with a machine that has a set of levers, some of which return a high reward, while others return a low reward. The player is tasked with getting a high return from a fixed number of lever pulls. This creates a dilemma, where the player has to decide to explore, i.e. try a new lever to maybe find a higher return, or exploit, i.e. pull the lever with the highest-known reward. This exploration exploitation dilemma is a key problem of reinforcement learning, and L.Kocsis et al. [24] apply it to tree searches, effectively viewing the decision of which move to play in a node of the game tree as a multi-armed bandit problem.

As described by L.Kocsis et al., UCT is a rollout-based planning algorithm that repeatedly samples possible episodes from the root of the tree. An episode is a possible sequence of moves that can be played from the root of the tree up to the end of the game or a fixed depth of the tree. The result of the episode is backpropagated upwards in the tree. When playing to a fixed depth of the tree, one way to evaluate a position is by randomly playing it to the end a number of times and using the average results of these random playouts to assess the position. UCT is thus an incremental process that improves the quality of the approximation of the move values at the root with every episode sampled and can be stopped at any time to return a result.

For every action a, state s, tree depth d, and time t, an implementation of UCT needs to track the estimated value of a in s at depth d and time t $Q_t(s, a, d)$, the number of

visits of s up to d and t $N_{s,d}(t)$, and the number of times a was selected in s at depth d and time t $N_{s,a,d}(t)$.

A bias term, shown in Equation 1, is defined where C_p is a constant.

$$C_{t,s} = 2C_p\sqrt{\frac{\ln t}{s}} \tag{1}$$

$$argmax_a(Q_t(s,a,d) + C_{N_{s,d}(t),N_{s,a,d}(t)}) \tag{2}$$

MCTS with UCT selects actions at every node of the tree according to Equation 2, updating the visit counts and estimated values of nodes as episodes are completed.

Equation 2 can be understood as weighting exploitation, in the form of the Q term, against exploration, in the form of the C term. The specific form of $C_{t,s}$ is shown to be consistent and to have finite sample bounds on the estimation error by L.Kocsis et al.

2.2 AlphaZero

The AlphaZero algorithm [32] is the application of AlphaGoZero [34] to games other than Go. AlphaGoZero is a significant simplification of AlphaGo [31], the first program to reach superhuman performance in Go.

AlphaGo used a complicated system involving initialization with example games and random rollouts during tree searches. It also used multiple networks, e.g. one to predict the value of a position and another to predict promising moves. This is shown in Figure 1. AlphaGo was first simplified into AlphaGoZero by only using a single network and not doing any rollouts anymore, instead the network evaluation for the given positions was directly used.

Further research by DeepMind produced AlphaZero. The difference between AlphaGoZero and AlphaZero is mainly that AlphaGoZero involved comparing the currently trained network with the previously known best player by letting them play a set of evaluation games against each other. Only the best player was used to generate new games. AlphaZero skips this and just uses the current network to produce new games. Surprisingly, this appears to cause no disadvantage, and learning remains stable. AlphaGoZero and AlphaZero are shown next to each other in Figure 2.

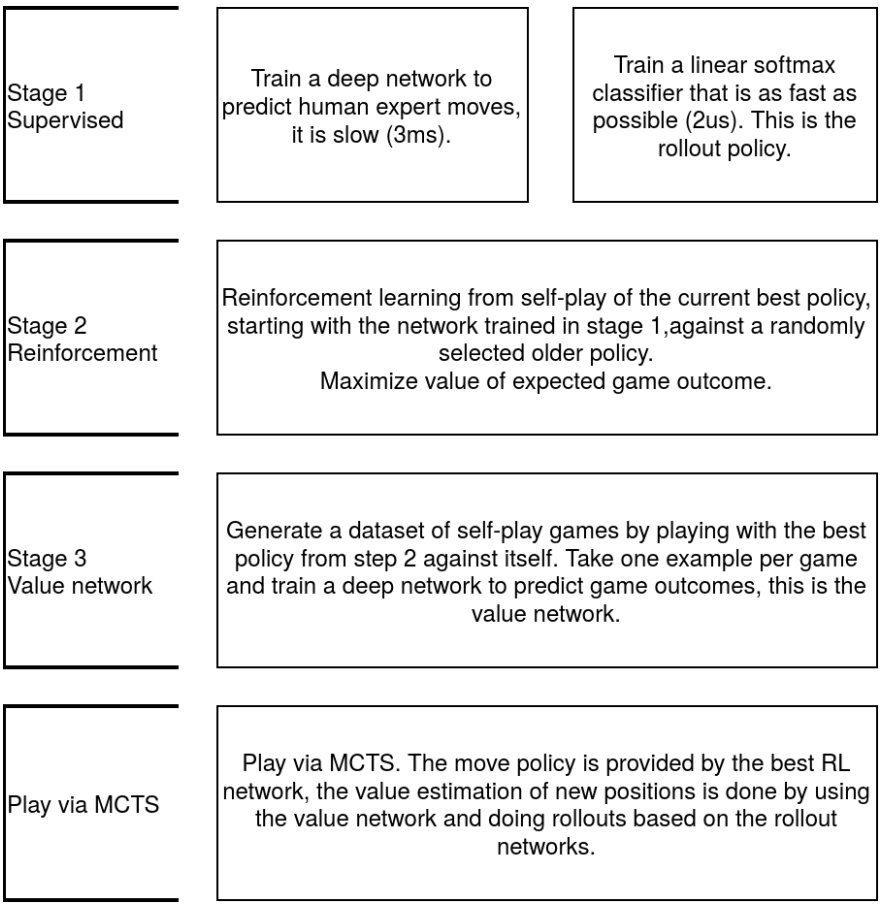

Figure 1: The original AlphaGo had multiple training stages and utilized expert human gameplay data to train multiple classifiers that were combined for MCTS.

The main advantage of the *zero versions is that they do not need any human knowledge about the game apart from the rules. The networks are trained from scratch by self-play alone, so unlike AlphaGo, AlphaGoZero does not require supervised initialization of the network with a dataset of top-level human play. This allows the algorithm to find the best way to play without human bias, which seems to slightly increase the final playing strength. Additionally, it allows using the algorithm for research of games for which no human experts exist, such as No-Castling Chess [25].

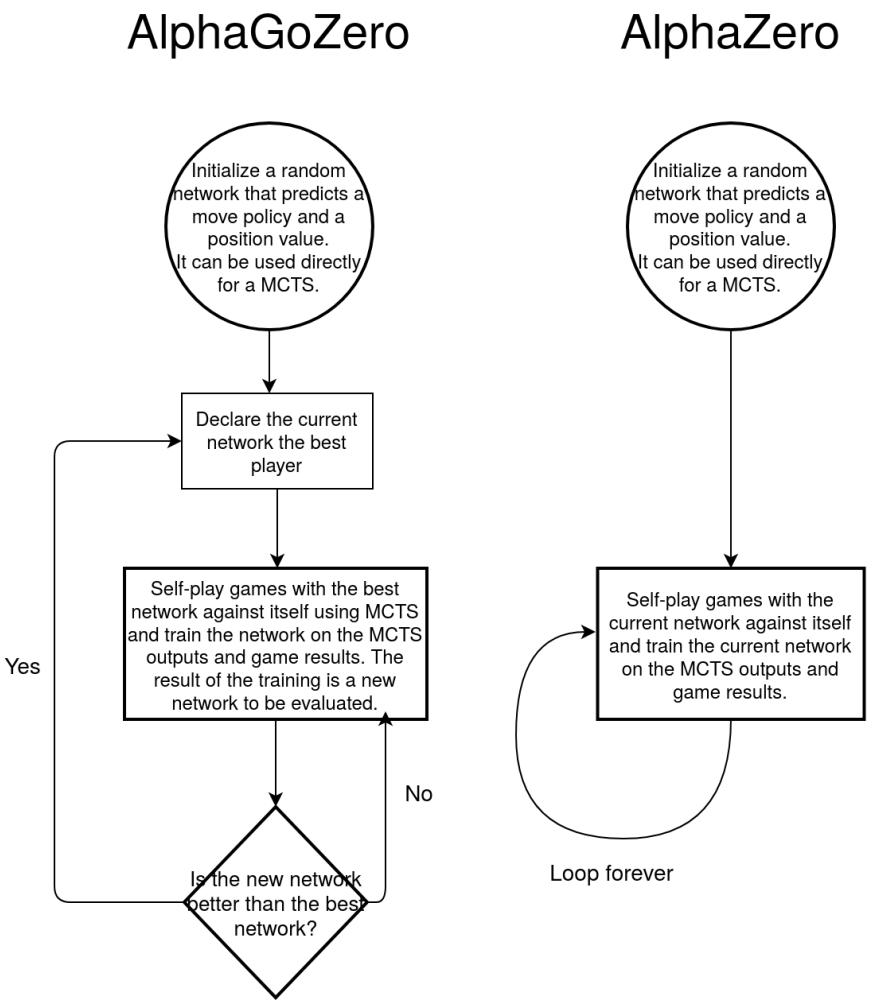

Figure 2: The *zero variants are based on a single network, which fulfills the roles of the value network and the reinforcement-learnt network from AlphaGo. There are no rollouts; only the value output is used to estimate positions. The process can be started with a randomly initialized network. AlphaZero does not check if an actual improvement has occurred and just uses every newly trained network.

2.2.1 The AlphaZero algorithm

The AlphaZero algorithm [32] uses MCTS with the UCT formulation, as described in section 2.1, and modifies it to incorporate a deep neural network that serves as a heuristic to bias the tree search and provides evaluations of unfinished games. The network is then trained to predict the final result of the MCTS search directly, as well as the final result of games played by MCTS against itself. This allows forming a closed cycle of self-improvement, which starts with a random network and has been shown to successfully learn to play such games, as chess, Shogi, and Go on the highest

level, if substantial computational resources are given.

Research done concurrently to AlphaGoZero describes a similar algorithm tested with the game hex [15]. They describe the algorithm as thinking slow, via MCTS, and fast, via the network alone, in reference to results from human behavioural science [23]. In that sense, AlphaZero learns in a similar fashion as humans. At first, a time-intensive "thought process" in the form of MCTS is used to reason about a new problem, but with practice good decisions can be made much faster, with a single forward pass through the network.

The network in AlphaZero is provided with an encoding of a game situation and has two outputs: a policy describing move likelihoods and the expected value of the situation for the players, i.e. it predicts what move should be played and who is likely to win in the given situation.

MCTS with UCT is modified to include the network outputs. This biases the search towards moves that the network believes to be promising. To this end, for every node of the search tree some statistics are stored, as shown in Table 1.

$N(s,a)$	The number of visits of action a in state s.
$W(s,a)$	The total action value.
$Q(s,a)$	The average action value, equal to $\frac{N(s,a)}{W(s,a)}$.
$P(s,a)$	The network policy output to play action a in state s.

Table 1: Statistics tracked per node in the AlphaZero MCTS

As described before in Section 2.1, UCT plays through episodes in an iterative fashion, starting at the root of the tree and playing moves until it finds a terminal game state or reaches a certain depth in the tree.

Unlike plain UCT, AlphaZero does not play out entire episodes till terminal game states and also does not use random games to evaluate positions, but rather calls the network whenever a move is made that reaches a node in the search tree that has not been reached before.

The analysis of the network is then backpropagated upwards the search tree, updating the node statistics in the process. The tree search then moves down the tree again, using the updated statistics, until it once again reaches an unknown node to be evaluated by the network.

The tree grows until some fixed number of nodes is created. The output is the distribution of visits to the actions of the root node, as a high visit count implies the tree search considers a move to be worthwhile and has analysed it thoroughly.

To use the network outputs in UCT, the formulations are changed. The action a in a given node is selected according to Equation 3, where $U(s,a)$ is a term that is inspired

by UCT but is biased by the network policy, as seen in Equation 4. C_{puct} is a constant to be chosen, the same as C_p in UCT, with values in practice ranging between 0.5 and 5, depending on the game.

$$argmax_a(Q(s,a) + U(s,a)) \tag{3}$$

$$U(s,a) = C_{puct}P(s,a)\frac{\sqrt{N(s)}}{(1+N(s,a))} \tag{4}$$

Using this tree search, games are played out; the resulting game states are stored with the MCTS policy and the final game result. The network is then trained to predict the MCTS policy for the states and the final result of the game.

To enhance exploration in self-play games Dirichlet noise is added to the root node of the tree, pushing the MCTS to randomly evaluate some moves more than others, but not disturbing it enough to prevent it from stabilizing onto a good move after enough nodes have been investigated. Specifically, for the root node, $P(s,a) = (1-\varepsilon)p_a + \varepsilon\eta_a$ with p_a as the original policy by the network and $\eta \sim \text{Dir}(\alpha)$ with $\varepsilon = 0.25$.

α has to be chosen for the respective game to be played, and it should scale with the number of legal moves in a position. For Go, 0.03 was used.

The self-play learning hinges on the assumption that the policy generated by MCTS with the network is always better than the one by the network alone. In practice, this appears to hold, as learning is quite stable even in very complex games such as Go. Intuitively, this makes sense, as the MCTS effectively averages many network predictions into one, thus forming a sort of ensemble of network evaluations on possible future positions.

A drawback of this approach is the high computational cost, because in practice for good results the search tree to play a single move has to be grown to at least hundreds of nodes, necessitating the same number of forward passes through the network to play a single move. For more complex games, it takes millions of games to be played until the highest level of play is reached with a network having millions of parameters. This is the motivation in this thesis for researching possible improvements to the algorithm that allow learning to progress faster or with fewer example games played.

2.3 Extensions to AlphaZero

There are many proposed improvements to the AlphaZero algorithm, most of which aim at reducing the extreme computational cost of learning to play a new game. In this section a collection of such proposals will be presented, especially focusing on ideas

that do not call for game-specific knowledge. Still, some improvements might work better on some games than others, depending on the hyperparameters used.

In Section 3.6 experimental results of a reimplementation of a selection of these proposals by previous work will be presented and introduced as a baseline to improve upon. Methods were selected based on the improvement claimed by previous work, compared to how much they would complicate the implementation. The goal was to produce a strong AlphaZero baseline to compare it with novel proposals.

2.3.1 Network and training modifications

The original network used in AlphaZero is a tower of 20 or 40 residual network blocks with 256 convolutional filters. There are some ways to improve this network to speed up training without any actual change to the AlphaZero algorithm, mainly by applying progress in the artificial neural network design.

- Enhancing the blocks with squeeze-and-excitation elements [21], has been proposed by the Leela Chess Zero project, which implements AlphaZero for chess as a distributed effort [3]. A comparable approach is used in [36] which also cites the similarity to squeeze-and-excitation networks.
- The original design of the network only uses 2 convolutional filters in the policy head and a single convolutional filter in the value head. Using 32 filters has been found to speed up training, as reported by [38] based on earlier work done by the Leela Chess Zero project.
- Cyclic learning rates, as developed by [35], can be used to improve the network fitting to the data. [38] shows a modest speed-up.

2.3.2 Modification of the tree search

The behaviour of the tree search can be modified to change how available time is spent to analyse given game positions.

- Instead of using a single network, [26] proposes combining multiple networks of different sizes, especially two networks, one big and one small. Specifically, in the paper, the big network has 10 residual blocks with 128 convolutional filters, whereas the small network has 5 residual block with 64 convolutional filters. This results in one forward pass of the big network taking as long as eight forward passes of the small network. Using the small network in most positions reduces computational requirements, allowing more search steps. This appears to increase playing strength. Given the same computational budget, the authors state that training is accelerated by a factor of at least 2.

- An idea called "Playout Caps" is proposed in [36]. They drastically reduce the number of MCTS playouts randomly in most of the moves played. It allows playing more games in the same time, which is somewhat similar to the concept of using a combination of a small and a big network. The authors thus argue that this gives more data to train the value target, which is starved for data since every game played is only a single data point for this target. A training acceleration of 27% is stated.

- The propagation of terminal moves through the search tree to simplify the search is proposed by the Leela Chess Zero project, with a moderate improvement in playing strength [4]. This kind of improvement falls into a family of various other ways to improve the handling of the search tree, such as detecting positional transpositions. The AlphaZero paper does not clarify to what degree DeepMind used such optimizations.

- The Leela Chess Zero project suggests analysing statistics, namely the Kullback-Leibler divergence of the policy as it evolves during the tree search, to understand how complex a position is and to apply more tree search evaluations on complex situations. They find a notable increase in playing strength [5] using the same computational budget.

2.3.3 Learning target modifications

The training targets of the neural network can be modified or extended to regularize training and reduce overfitting.

- [36] Proposes predicting the opponent's reply to regularize training. A modest improvement is shown.

- [36] also shows that major improvements can be made if some domain-specific targets are used to regularize the learning process. This hints at the possibility to search for ways to automatically determine such regularization targets. They again emphasize how the learning is highly constrained by available data on the value target and how these additional targets may help alleviate this.

- Forced Playouts and Policy Target Pruning, proposed in [36], posit that nodes, if they are ever selected, receive a minimum number of further playouts. Pruning is used to remove this from the policy distribution used to train the network for bad moves, as the forced playouts are only meant to improve exploration. Training progress is stated to be accelerated by 20%.

- The Leela Chess Zero project uses a modification of the value target which explicitly predicts a drawing probability, as this allows the network to tell the difference between an uncertain position and a position that is likely drawn [6].

- [14] suggests using a third output head which predicts the win probability for every move. This can be used to shortcut the tree search, reducing the number of network evaluations, but the win probability estimate might be of a worse quality. Small improvements are claimed.
- Instead of using the result of games as a learning target for the value output of the network, [38] proposes using the average value of the MCTS node at the end of the MCTS search for a given position. This has the advantage of providing richer data, and reducing the influence of a single bad move at the end of a game, which would taint the evaluation of all positions in that game. However, as this MCTS-based value has other accuracy issues, they specifically recommend combining the two values, which shows a noticeable improvement.

2.3.4 Training data enhancements

The way the millions of training examples for the network are handled can be modified.

- There can be multiple copies of the same identical position in the training data. [38] shows that averaging the targets for these positions and reducing them to a single example is beneficial. This is shown on the game of Connect 4, which is especially prone to identical positions showing up, so it is not clear how well it might translate to more complex games.
- As seen in [38], the playing strength quickly increases the moment the very first training examples are removed. This is likely because those examples were effectively generated with a random network and are thus very bad, holding back training. To fix this problem, a modification to the training data windowing is proposed.

3 Experimental Setup

For this thesis a framework for experiments with AlphaZero has been developed which allows easily setting up different configurations with a configuration file, switching on or off various features and improvements under investigation. An overview of the distributed set-up is shown in Figure 3.

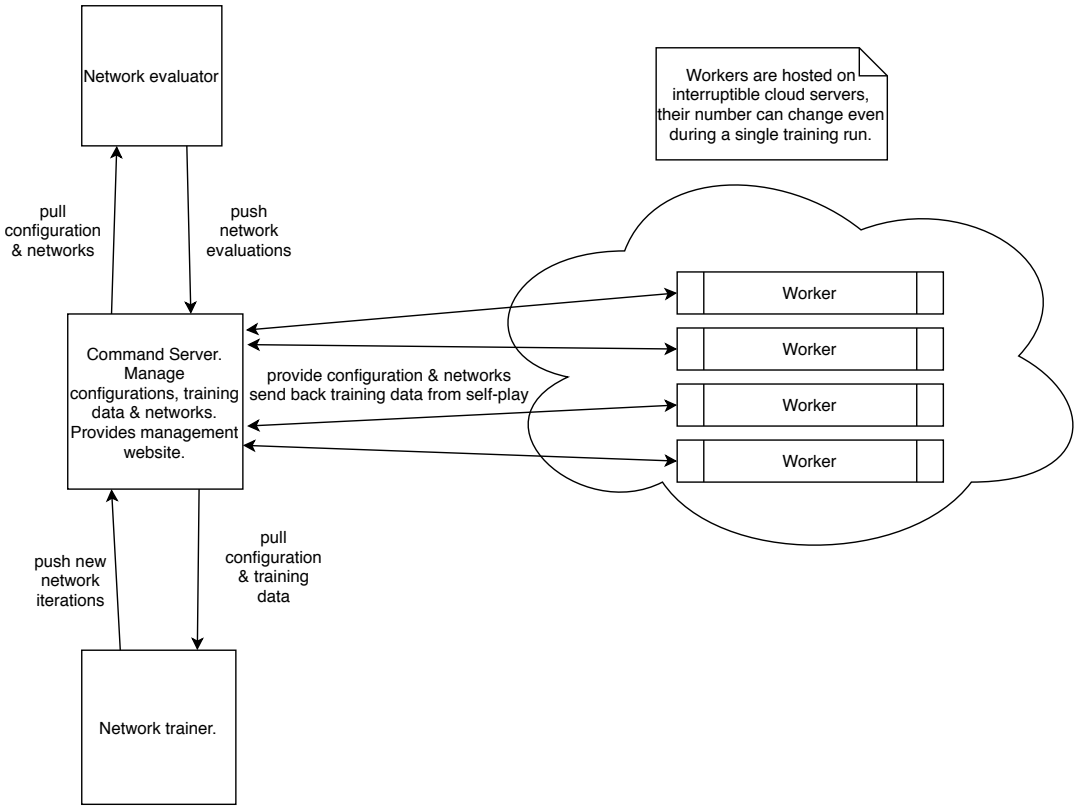

Figure 3: An overview of the distributed set-up for AlphaZero experimentation. A central command server manages all data and configurations, GPU-intensive tasks, such as network evaluation, network training, and especially self-play are handled on machines talking to this central server.

All code is available on github[1]. The repository documents all configuration files used for every experiment mentioned in this thesis. For the sake of reproducibility the commit SHA is recorded as well, allowing resetting the code to the exact form it had when a specific experiment was run.

The framework base implementation of AlphaZero uses a different network target for game results which encodes a dedicated value for a draw. This means that for a two-player game the possible outcomes are the win of either player, or a draw. This differs

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

from the original implementation which used a single output with values between zero and one. There might be a slight change in learning efficiency due to this, but the main reasons for this were of technical nature in the context of the abstraction-driven framework, primarily the goal of an easy way to implement other games than Connect 4.

The experiments are split into multiple parts:

1. Establish a baseline with a base implementation very close to raw AlphaZero
2. Implement some known extensions to AlphaZero.
3. Evaluate proposed ideas one by one.

3.1 Network architecture

The network used in all experiments is a minimal modification of the original network used in the AlphaZero work by DeepMind. Modifications are made in the output of the win probabilities, as the original work did not explicitly predict draw chances. Additionally, the input encoding was adapted for Connect 4, and the network heads use a higher number of convolutional filters, as previous work has noted that this slightly increases performance [28].

Both the input to the network and the output of the network are relative to the player currently making a turn, the network does not know if it plays red or yellow. The input encoding converts the 7×6 board into a tensor of $1 \times 7 \times 6$. Empty fields are encoded as a 3, the player currently making a move is a 1 and the other player is represented by a 2. 0 is not used, as all convolutions in the network use padding with 0, such that all convolutions keep working on a field of size 7×6. 0 thus exclusively represents the edge of the board and the network can learn this. For the output the move policy is encoded by a flat vector of 7 values, activated with SoftMax, forming a probability distribution. The win probabilities are encoded as a vector of 3 values, activated with SoftMax. The first represents the chance of a draw, the second the chance of the current player winning, and the last the chance of the other player winning. The network is shown in Table 2.

The used loss function is the sum of the negative log likelihood of the move policy and the win predictions with the win prediction loss weighted at 0.01. The idea of such a weighting stems from the work on AlphaGoZero [34].

Description	Network structure
Initial block	$\begin{pmatrix} 3 \times 3 \times 64 \\ BatchNorm \\ ReLU \end{pmatrix}$
Adapter convolution	$1 \times 1 \times 128$
Residual block, repeated n times	$\begin{pmatrix} 3 \times 3 \times 128 \\ BatchNorm \\ ReLU \\ 3 \times 3 \times 128 \\ BatchNorm \\ Addition \\ ReLU \end{pmatrix}$
Move policy output	$\begin{pmatrix} 3 \times 3 \times 32 \\ FC : 7 \\ SoftMax \end{pmatrix}$
Win prediction output	$\begin{pmatrix} 3 \times 3 \times 32 \\ FC : 3 \\ SoftMax \end{pmatrix}$

Table 2: Structure of the used network in this work, a smaller and slightly modified version of the original AlphaZero network used by DeepMind [32]. $x \times y \times z$ describes a convolution with kernel size $x \times y$ and z filters. $FC : x$ describes a fully connected layer with x neurons. $Addition$ describes the addition with the input of the residual block the addition is a part of, forming the residual structure of the block. Both the move policy output and the win prediction output are connected to the output of the last residual block. Residual blocks make up the bulk of the network. In most parts of this thesis 5 blocks are used.

3.2 Testing on Connect 4

To reduce costs of experiments, the game Connect 4 is used. Unlike Go, Connect 4 is a game with a known solution, as it is a solved game for which strong solvers are available [1, 7, 8], i.e. for any given position the solver can quickly find the optimal move to play.

This makes it possible to evaluate the playing strength of AlphaZero as a measure of accuracy against the strong solver on a dataset of test positions. It also allows running supervised training of the used network on a dataset of games played by the solver, establishing a maximum performance possible by the network.

3.2.1 Generating Connect 4 datasets

The generation of the database to be used as a testing set for the learning process is an important step that determines how comparable results are to previous work.

One previous work from OracleDevs et al. [28] published results on Connect 4 using AlphaZero. They claimed to reach 97% to 99% accuracy.

Experiments with various datasets have made clear that results can vary wildly between 90% and the claimed 97%, depending on the dataset configuration.

One important decision concerns what is considered a correct move in a given situation. In Connect 4 there are many positions where the player has multiple ways to force a win, but some may lead to a longer game before the win is forced. [28] defines here *strong* and *weak* testsets:

A *strong* testset only considers a move correct if it yields the fastest win or the slowest loss.

A *weak* testset only cares about the move producing the same result as the best move, no matter how much longer the win will take or how much faster the loss will occur.

They report 97% accuracy on strong datasets and 99% on weak datasets.

Additional important decisions made in the dataset creation which are not talked about in detail by previous work are:

1. How to play the games exactly? Only perfect moves or some mistakes are allowed?
2. Should duplicate positions be filtered out?
3. Should trivial positions, i.e. positions that have no wrong answer, be filtered out?

Experiments with various options showed that the question of how games are played can substantially influence the final accuracy values. A dataset generated by playing 50% random moves and 50% perfect moves, found by a solver [7, 8], is substantially harder than a dataset created using 90% perfect and only 10% random moves. This appears to be mainly a result of the distribution of game length, shown in Figure 4, as fewer random moves produce a dataset with more positions in the late game, while more random moves cause more positions to be in the early game. Making the right move in turn 10 is a lot harder than in turn 30, as the remaining game tree is much larger in the earlier phase of the game. Using supervised training, described in Section 3.4, the harder dataset can only reach about 92.6% accuracy, whereas the easier dataset can reach 96.97%, very close to the 97% claimed by previous work.

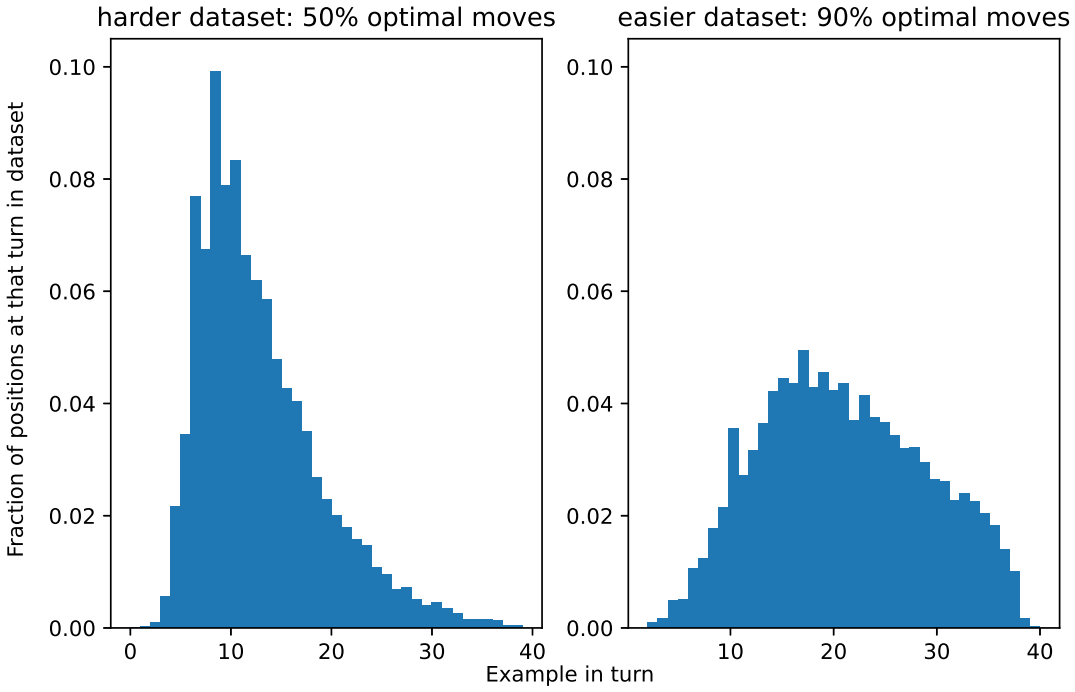

Figure 4: Different ways of generating the dataset can cause a substantially different distribution of examples.

For most of the following experiments the harder dataset, playing 50% random moves, was used. If a dataset is specified, they will henceforth be referred to as either the easy or the hard dataset. For all other options the hardest possible settings were used to create a maximally challenging dataset: No duplicates, without trivial positions and only the strongest possible moves are accepted as correct.

The average number of correct moves per position can be used as a metric to determine how challenging a dataset is. The generated maximally hard dataset, 50% random moves, has on average 1.8 correct moves per position. The generated easier dataset, playing 10% random moves, has 1.95 correct moves per position.

In contrast, OracleDevs et al. [28] report 4.07 correct moves per position in their weak dataset and 2.16 correct moves per position in their strong dataset, indicating they made choices which noticeably reduced the challenge their dataset posed, possibly throwing some light on the differences in accuracy values between this work and theirs.

3.3 Evaluation of training costs

The goal of this thesis is to identify ways to reduce the substantial costs of training using AlphaZero. The majority of GPU capacity is spent on self-playing games to be

used as training material for the neural network. In all experiments neural network training is done on a single GPU, newly produced networks are evaluated on another single GPU, and self-play workers are run on a P2P cloud service[2] using up to 20 GPUs of various types to complete full training runs for Connect 4 in 2 to 6 hours, depending on the number of self-play workers.

This huge imbalance between training hardware and self-play hardware can be seen in related work as well, e.g. Silver et. al. [33] used 5000 first-generation TPUs for self-play and 16 second-generation TPUs to learn to play chess on super-human — and potentially even super-classical-engine — level within hours. Similarly, David J. Wu [36] used up to 24 V100 GPUs to produce self-play training examples to feed a single V100 GPU training data for neural network training to play Go.

The bulk of the training cost consequently lies in the self-play workers. For all experiments in this thesis, this fact will be used to simplify the training cost measurement by ignoring the costs of the network training. Instead, only the cost of self-playing workers is measured.

Since the main source of computational resources for self-play is a P2P cloud service which provides unreliable but cheap GPU time, there is no constant number of GPU workers between experiments, or even the number of self-play workers changes during training.

After an experiment is completed, a benchmarking program is run on a reference machine using the produced networks and measures how much time is needed on average to play a single move. This value is then used to estimate the cost of the actual number of moves that were played by the self-play workers during the experiments.

This reference machine uses a single Nvidia RTX 2070S, which is saturated to 100% load by the benchmark. Thus, all self-play cost of experiments is stated as estimated self-play time on that machine and bottlenecked by the GPU capacity on it.

3.4 Supervised training

To provide guidance on how good the results of AlphaZero are, supervised training of the networks is used to establish the maximum possible performance of the network. For this a dataset of 1 million Connect 4 positions, generated using 50% random moves as outlined in Section 3.2.1, is employed. Two versions of the dataset are generated. In version 1, all positions of played games are used as training data; in version 2 of the dataset, only a single position is randomly picked from each played game, increasing the number of distinct games played to generate the dataset substantially.

For both versions of the dataset, 800000 examples were used as training data, 100000 examples were used as validation data, and the remaining 100000 examples were used

[2] `https://vast.ai`, last accessed: 5.8.2020

as test data. This dataset size was chosen with the goal of allowing multiple supervised training runs per day: With this size about three runs can be run on a single machine per day. Training started with a learning rate of 0.2 and was reduced by a factor of 10 after 8 epochs of no accuracy improvements on the validation data. The supervised training stops after 12 epochs with no improvements on the validation accuracy.

Various network sizes are evaluated. Mean results of five runs for each network are shown in Table 3. One additional set of five 5-block networks were trained using the easier 10% random moves dataset. They reached a mean supervised accuracy of 96.94% on move prediction and 84.75% on win prediction, highlighting the big difference in achieved accuracy between the harder and the easier dataset.

Network	parameters	moves % v1	wins % v1	moves % v2	wins % v2
5-blocks	1574730	91.63%	77.47%	92.44%	79.23%
10-blocks	3053130	92.37%	77.87%	93.00%	79.67%
20-blocks	6009930	92.68%	78.23%	93.49%	79.93%

Table 3: Results of supervised training. Accuracy compared to a connect 4 solver.

What is surprising about these results is the low accuracy on the prediction of the winner of a match, as AlphaZero training runs achieve values close to 90% win-prediction accuracy. This hints at some problem with the supervised training data.

To rule out overfitting on the dataset, another dataset, using the 10% random moves, with 10 million positions taken from 10 million games was generated. This dataset achieved a move prediction accuracy with the 5-block network of 97%, but the win-prediction still stayed below the accuracy reached in AlphaZero training runs, which played fewer than 10 million games overall. Given that the used 5-block network only has about 1.5 million parameters, overfitting on a dataset of 10 million examples seems unlikely.

Since overfitting is ruled out as a cause, the issue is likely to be that a dataset generated from games with a certain fraction of random-moves does not provide optimal training quality to predict the outcome of perfect-play games, which causes the low win-prediction accuracy. Games played on the best level AlphaZero can accomplish are not perfect, but mistakes made are much different from just playing random moves. This seems to be mirrored in the win prediction accuracy.

3.5 Baseline

Various baselines need to be established to compare novel proposals against. Supervised training of the used networks allows to explore the maximum possible performance, a plain AlphaZero baseline shows how the original algorithm performed, while

a baseline of AlphaZero extended with a set of previously known improvements shows the progress made since the original algorithm was published.

For all experiments, unless noted otherwise, the MCTS uses trees with 343 nodes, which was picked as it represents the number of nodes to fully search 3 moves ahead in Connect 4: 7^3. Networks will be trained in iterations of 300000 produced examples. This size of the iterations was mainly chosen as it results in iterations of a few minutes, making progress quickly visible. Every network is evaluated on the test set for accuracy compared with the Connect 4 solver. Network training happens according to parameters suggested by the original works using SGD with momentum of 0.9 and weight decay of 0.0001. The learning rate starts at 0.2; it is dropped to 0.02 in iteration 7, and to 0.002 at iteration 14. Gradient clipping at a magnitude of 1 is used to stabilize training at the maximum learning rate of 0.2. Without it or with a higher clipping threshold, training would sometimes diverge. A training window of 2 million positions is used; this is primarily limited by the speed of the machine used for network training.

Based on the results of the supervised training and under consideration of the training costs, all experiments are done with the 5-block network, which has about 1.5 million parameters. A larger network might reach slightly better results, but the goal of this thesis is to look for efficiency gains, not maximum final performance. Therefore, the smaller network is used to allow for faster experimentation.

3.5.1 Hyperparameter search

The AlphaZero algorithm requires some important hyperparameters, which can make a large difference on the learning efficiency.

To establish a baseline with sensible hyperparameters a Bayesian hyperparameter optimization with 65 search steps was used. In each search step, the AlphaZero algorithm was run on a single machine, which alternated between self-play and training. The task of the Bayesian optimization was to optimize the accuracy after two hours of real time. The number of optimization steps was limited to 65 due to real-world time constraints, this search took about a week of computation to complete all steps.

To yield meaningful results after just two hours on a single GPU, the size of iterations were chosen in such a way as to make faster progress, at the cost of the quality of the final results, i.e. fewer games were played per network trained and the MCTS trees were smaller when playing games. Thus, the accuracy values reached were rather low, around 80%, but still allowed for a relatively quick comparison between different hyperparameters.

After this initial search, three sets of hyperparameters were selected and evaluated with full experimental runs to maximum accuracy in order to pick the best hyperparameters for the following work.

The hyperparameters optimized for are listed in Table 4. A preliminary run of hyperparameter optimization was used to inform the decision to optimize these parameters in particular.

Parameter	Description
cpuct	The C_{puct} constant of AlphaZero, see Equation 4 on page 14.
fpu	First play urgency. The value of a move which has not been played yet at all during MCTS. It has some control over the balance between exploration and exploitation as well.
alphaBase	Determines the value of α, which is a constant used in AlphaZero to control explorative noise. alphaBase is an extension to adapt to the fact that the raw α value highly depends on the number of legal moves a game has on average. alphaBase is used to calculate the value of α, depending on the number of legal moves: $\alpha = \frac{alphaBase}{n}$, where n is the number of legal moves. In the context of Connect 4 with a 7x6 board, alphaBase thus needs to be divided by at most 7 to determine the corresponding α value in the context of the original implementation.
drawValue	Determines the value of a draw when calculating the value of a position in an MCTS node. This parameter does exist because of the different way chosen to implement the network value output, which produces an output indicating the likelihood of a draw. To estimate the value of a position, the formula $W + D * drawValue$ is used, where W is the network estimation of a win and D the estimation for a draw. A high value thus makes a draw as desirable as a win, a low value as undesirable as a loss.

Table 4: Hyperparameters searched

The hyperparameters chosen for further investigation in full experimental runs are shown in Table 5. Hyperopt1 and hyperopt2 were the two best sets found by the hyperparameter search. prevWork is based on results of previous work.

Name	cpuct	fpu	alphaBase	drawValue
hyperopt1	0.9267	0.91	12.5	0.4815
hyperopt2	1.545	0.8545	20.38	0.6913
prevWork	4	0	7	0.5

Table 5: Hyperparameter sets selected for further investigation.

Full training runs are done 5 times for each of the hyperparameter set under investigation. In every run, training is stopped once the MCTS accuracy has not improved for

10 iterations. The 5 runs are used to calculate a mean, which is used to compare the different runs.

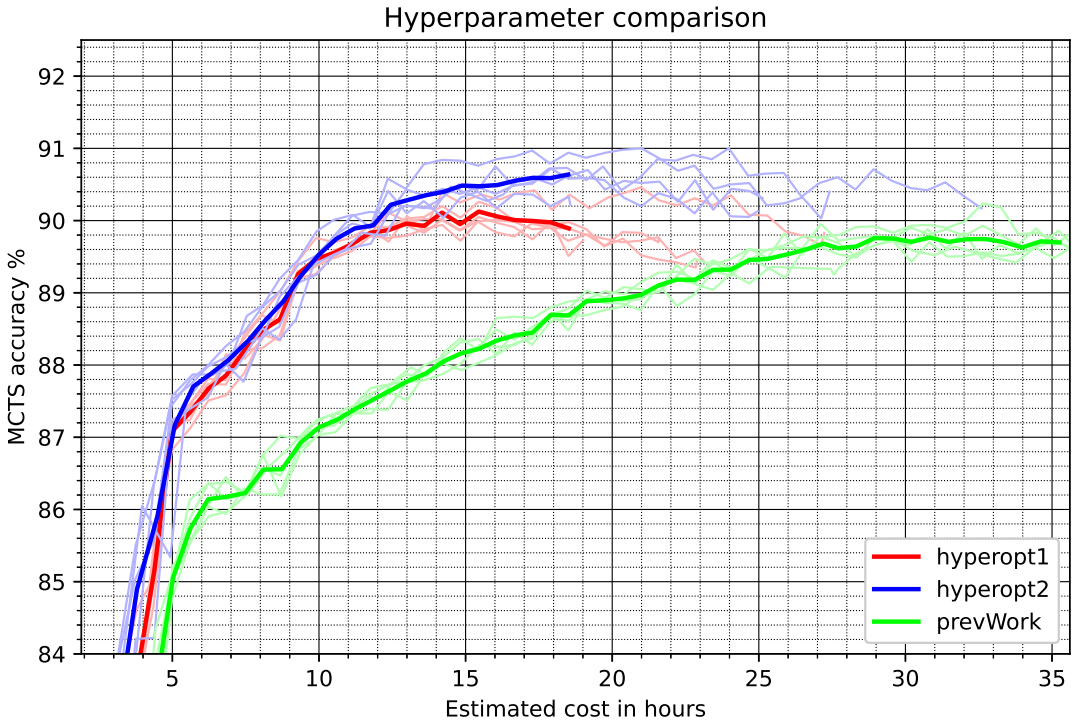

Figure 5: Results of the runs to determine a good hyperparameter set. Mean is only calculated until the first of the single runs stops showing improvements.

It can be seen that the results of the parameters taken from previous works fall substantially behind the values found via Bayesian optimization. This is likely because the previous work was on other games, using different implementation details, such as different tree sizes, iteration sizes, network sizes and/or other differences. It is not known why the set hyperopt1 shows a drop in accuracy towards the end.

The hyperparameter set hyperopt2 is used for all further experiments; its performance forms the baseline on which improvements are investigated.

3.6 Extended Baseline

As outlined in Section 2.3 on page 14, much work has been done to propose various ways of improving learning efficiency of AlphaZero.

The differentiation between the AlphaZero baseline and the extended AlphaZero baseline is important to verify that proposed improvements are cumulative with already known ways to improve AlphaZero.

3.6.1 Remove duplicate positions

Especially for games such as Connect 4 there are a lot of duplicate positions in the training data. Depending on various exploration hyperparameters and training progress, this causes between 30% and 80% duplicate positions to be reached. The neural network training is thus provided with a large set of duplicate input values, which will have conflicting target values to be learned. Additionally, positions early in the game will be overrepresented in the training data, as they make up the vast majority of duplicate positions.

In the context of Connect 4, OracleDevs et al. [38] propose merging duplicate positions to counteract this.

The present thesis implements this by acting in the training worker. Only new positions are added to the pool of training examples; previously known positions instead update the target value for that position.

For this purpose, a record of all previously known positions is kept in a hash map. Every time a duplicate is produced by a self-play worker and sent to the training worker, the training worker will merge the new target values with the old target values of the duplicate using a weight $w_{\text{duplicate}}$:

$$\text{target}_{\text{new}} = \text{target}_{\text{old}} * (1 - w_{\text{duplicate}}) + \text{target}_{\text{duplicate}} * w_{\text{duplicate}}$$

The higher $w_{\text{duplicate}}$, the less importance is given to previous targets for a position. At the maximum value of 1, targets are completely replaced whenever a new report on a position comes in.

The best value for $w_{\text{duplicate}}$ is unclear. Since it is not the primary goal of this work to establish all optimal hyperparameters for Connect 4, three possible choices were tested and the best one picked. They show some differences and it is suggested that future works targeting maximum accuracy do a more complete search for this hyperparameter. The values 0.2, 0.5, and 0.8 are tested in single runs and compared versus the baseline to evaluate how promising this approach is. Figure 6 shows the results.

A noticeable improvement can be seen. 0.8 appears to outperform the baseline the most and is chosen to be part of the extended baseline configuration.

To keep the size of iterations consistent with previous experiments, network iterations are now reduced to once every 180000 new examples. With the typical rate of duplicates during a baseline run, this comes out at about 300000 positions played per iteration, at least in the first part of the training. The more duplicates are produced during play, the longer the iterations become in real time, as fewer new examples are generated. The training window stays at 2 million positions, which however will now be distinct from each other.

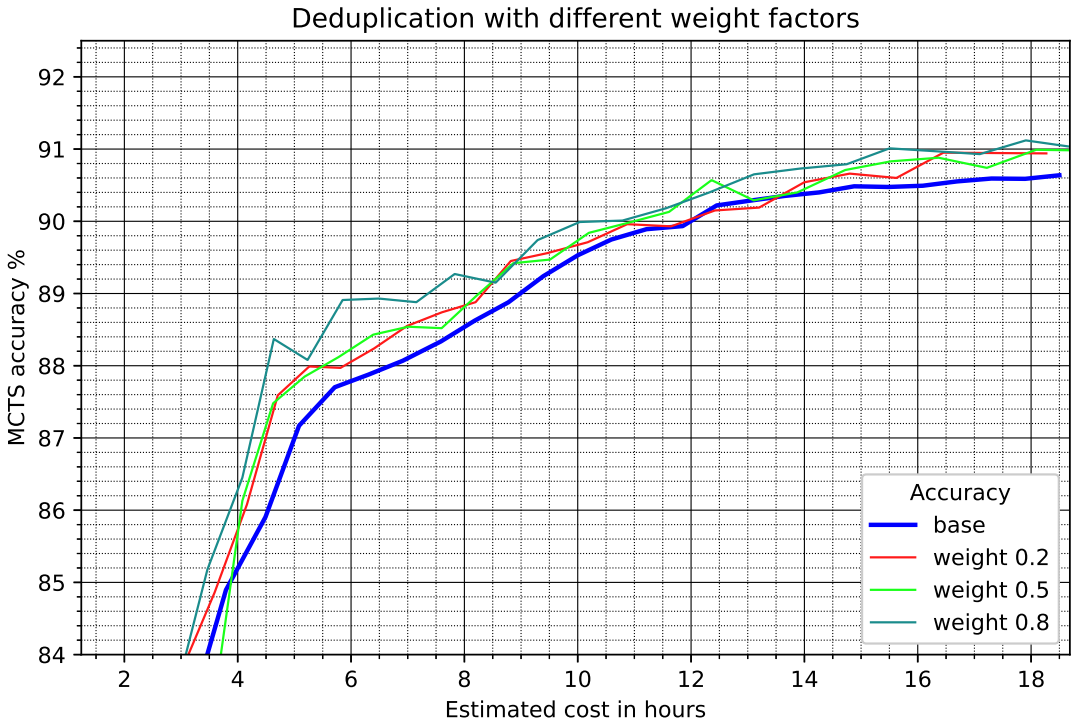

Figure 6: Comparison of different choices for $w_{\text{duplicate}}$. 0.8 is chosen for all further experiments.

3.6.2 Cyclic learning rate

Cyclical learning rates are proposed by Smith et al. [35] to speed up general neural network training. OracleDevs et al. [38] claim some improvement using them in the context of AlphaZero.

Based on these claims, cyclic learning rates and cyclic momentum have been implemented for AlphaZero as a potential addition to the extended baseline. The cycles are over single iterations of the network: 300000 examples using the baseline or 180000 new examples using deduplication.

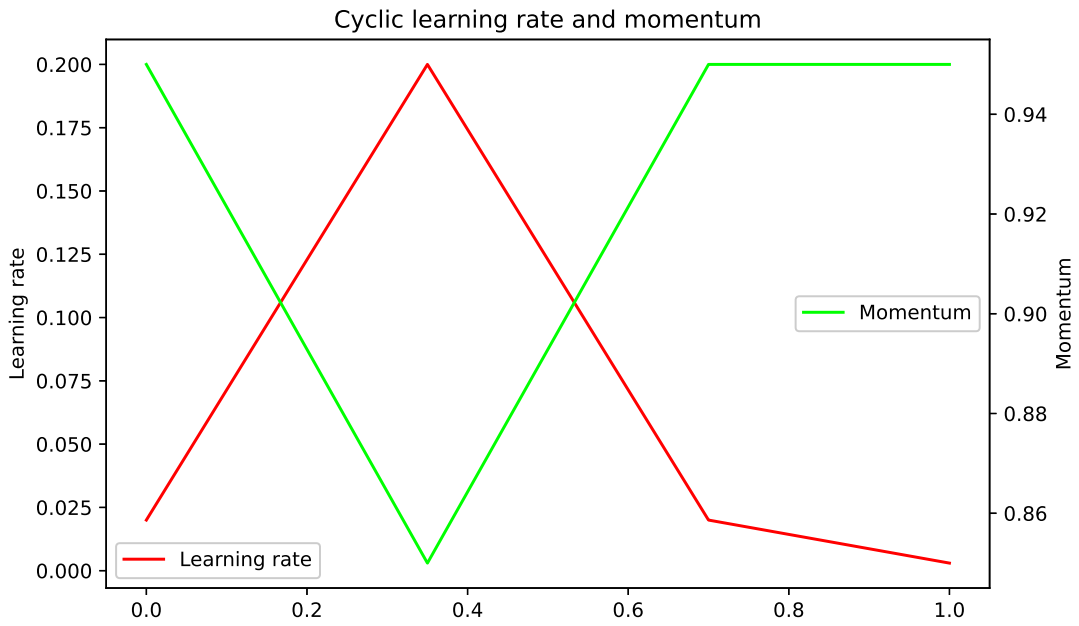

Figure 7: The change of the learning rate and momentum over the training of a new network iteration when using cyclic learning rates.

The learning rate starts out at 0.02, reaches its peak of 0.2 at 35% of the iteration, then starts to drop back to 0.02, which is reached by 70% of the iteration. Finally, the learning rate drops down to 0.003 by the end of the iteration. Using the same proportions over the iteration, the momentum starts at 0.95, drops to 0.85, and then rises back to 0.95. The values are informed by previous runs to determine the maximum and minimum useful learning rates, especially the supervised training which reduced the learning rate automatically using early stopping.

These curves can be seen in Figure 7. Additionally, though not shown in the figure, the learning rate is annealed down with a multiplicative factor that drops with the network iterations in a linear fashion. It starts at 1 in iteration 1 and reaches 0.4 in iteration 20. Beyond iteration 20, the factor stays at 0.4. Thus, at iteration 20 and later, the maximum learning rate is 0.08. This annealing is done to accommodate the fact that in later parts of training the learning rate should be smaller in general.

A single run was done with the baseline configuration, using the described cyclic learning rate and momentum. The results can be seen in Figure 8. Slight improvements were found, especially in the earlier stages of the training. Cyclic learning rates and momentum were added to the extended baseline configuration.

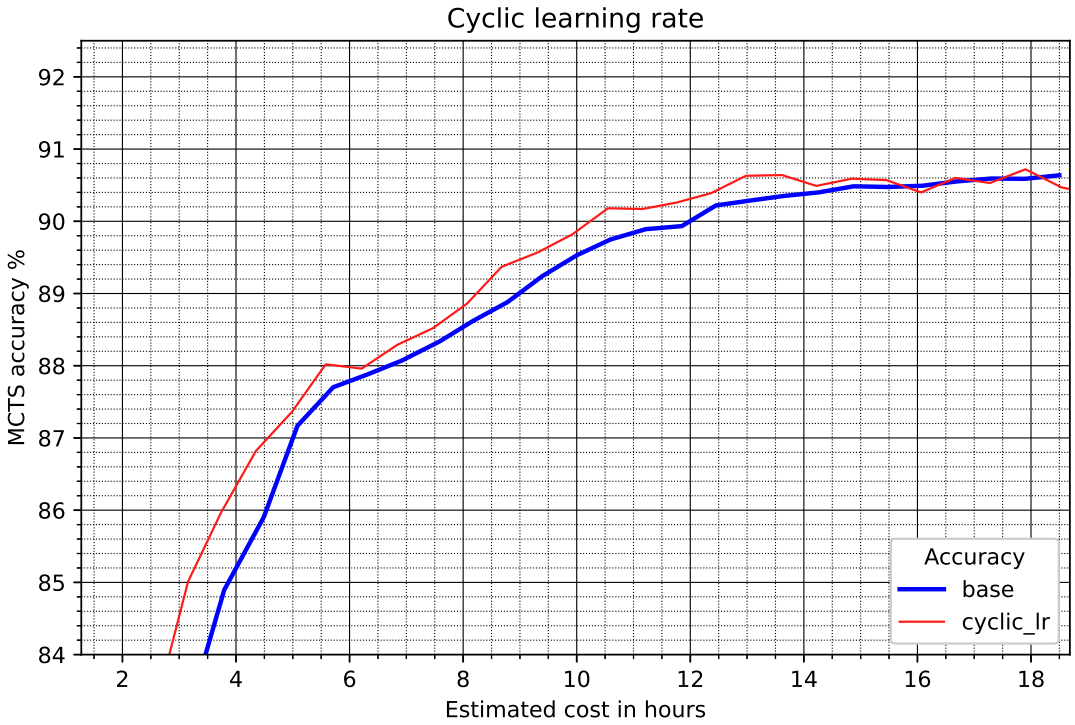

Figure 8: Comparison of the usage of cyclic learning rates and momentum with the baseline.

3.6.3 Improved training window

OracleDevs et al. [38] saw that a jump down in the training error can be observed in the first iteration which pushes the examples of the very start of the training out of the training window. It is obvious that these are especially bad training examples, as the network at that point would be very close to random play. This motives to let the training window start out small and grow over a number of iterations. Thus, the training examples of early iterations are removed from the training data more quickly. This is called a slow training window.

For this thesis, a slow training window is implemented; it starts with a size of 500000 examples, which grows between iteration 5 and 12 up to the maximum size of 2 million examples. This causes early examples to be removed by iteration 3.

A single run is done to evaluate the effects of this modification; the results are provided in Figure 9.

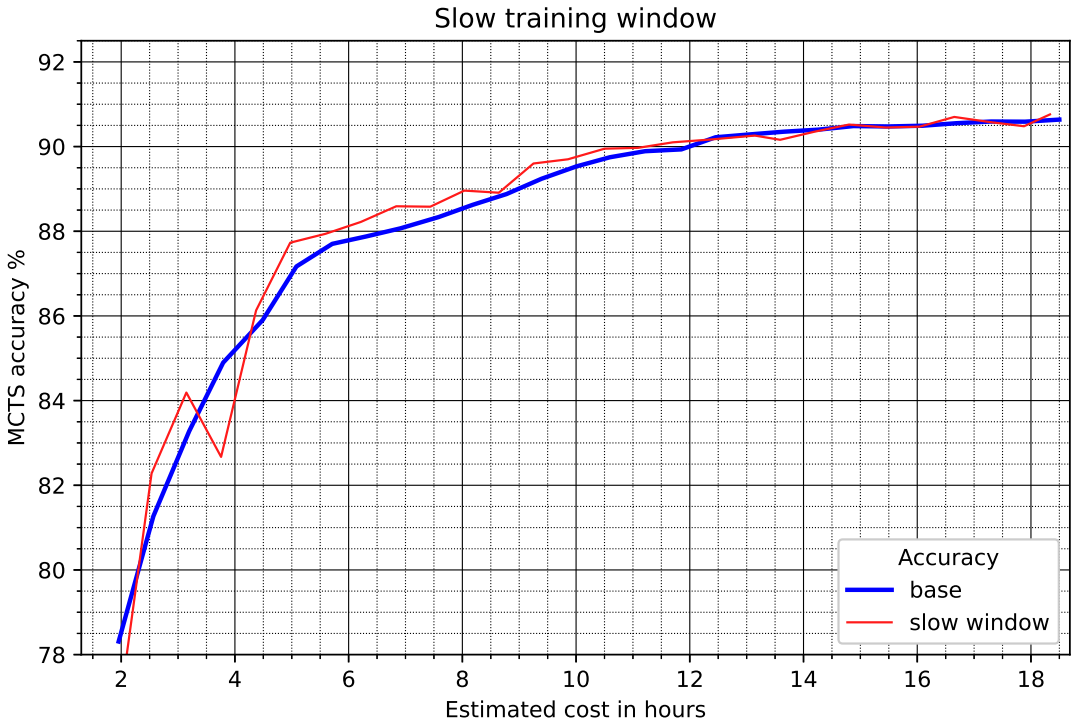

Figure 9: Comparison of the usage of a slow training window with the baseline.

The slow window is added to the extended baseline, since it does not appear to hurt and follows a sound idea. As the extended baseline uses a training window without duplicate positions, the parameters are modified to grow from iteration 5 to 15, since the number of new examples per iteration is lower.

3.6.4 Playout Caps

Playout Caps, suggested by Wu et al. [36], implement the idea to play, at random, a substantial fraction of all moves with a substantially reduced number of nodes in the MCTS and only record moves played with the full number of nodes used. This means a lot more games are played at a marginal additional cost. Since the moves played with a less deep MCTS are not recorded, the training data does not reduce in quality, but more distinct game results are collected. This can help provide better quality data for the value target of the network, which predicts the winner of games. Wu et al. claim 27% improvement in Go, which is notorious for especially long games.

Playout Caps have been implemented for this thesis and evaluated on Connect 4. For 75% of the moves played the number of MCTS nodes was reduced to 30. Only the moves played with the standard 350 nodes in the other 25% were recorded as training data. The results of a test run are provided in Figure 10.

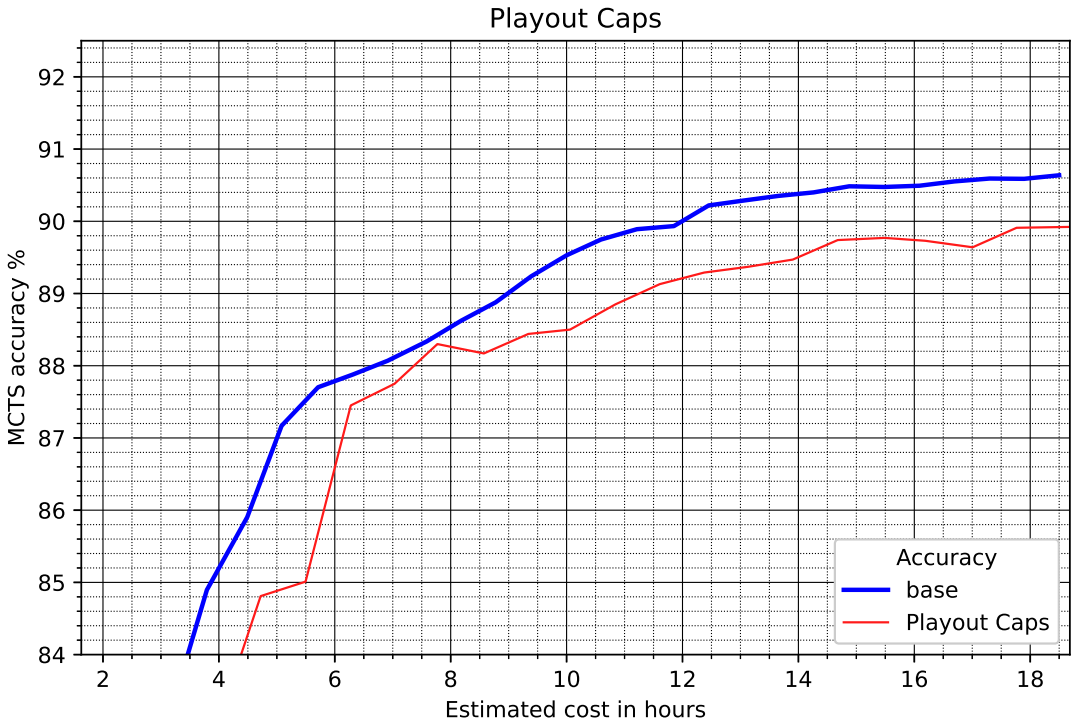

Figure 10: Results of implementing Playout Caps on Connect 4.

Clearly Playout Caps appear to hurt performance on Connect 4. This seems to result from the fact that an average Connect 4 game tends to take 30 moves. A game of Go, for which Playout Caps were developed, averages at 211 moves[3]. This means the AlphaZero learning to play Go will be a lot more pressed for more game result training data than AlphaZero learning to play Connect 4.

Playout Caps were not made part of the extended baseline.

3.6.5 Predicting the opponent's reply.

Wu et al. [36] show that predicting the opponent's reply after the current position produces a "modest but clear benefit". Specifically they propose to add a term to the loss function to regularize training:

$$-w_{\text{opp}} \sum_{m \in \text{moves}} \pi_{\text{opp}}(m) \log(\hat{\pi}_{\text{opp}}(m)) \tag{5}$$

where π_{opp} is the policy target for the turn after the current turn, $\hat{\pi}_{\text{opp}}$ is the network's prediction of π_{opp} and w_{opp} is a weight for the loss term. In the experiments with Connect 4, based on preliminary experiments, 0.35 is used.

[3] `https://homepages.cwi.nl/~aeb/go/misc/gostat.html`, last access: 5.8.2020

Figure 11 shows the results of testing this on Connect 4. There appears to be a small advantage for some parts of the run, which is why this was made part of the extended baseline.

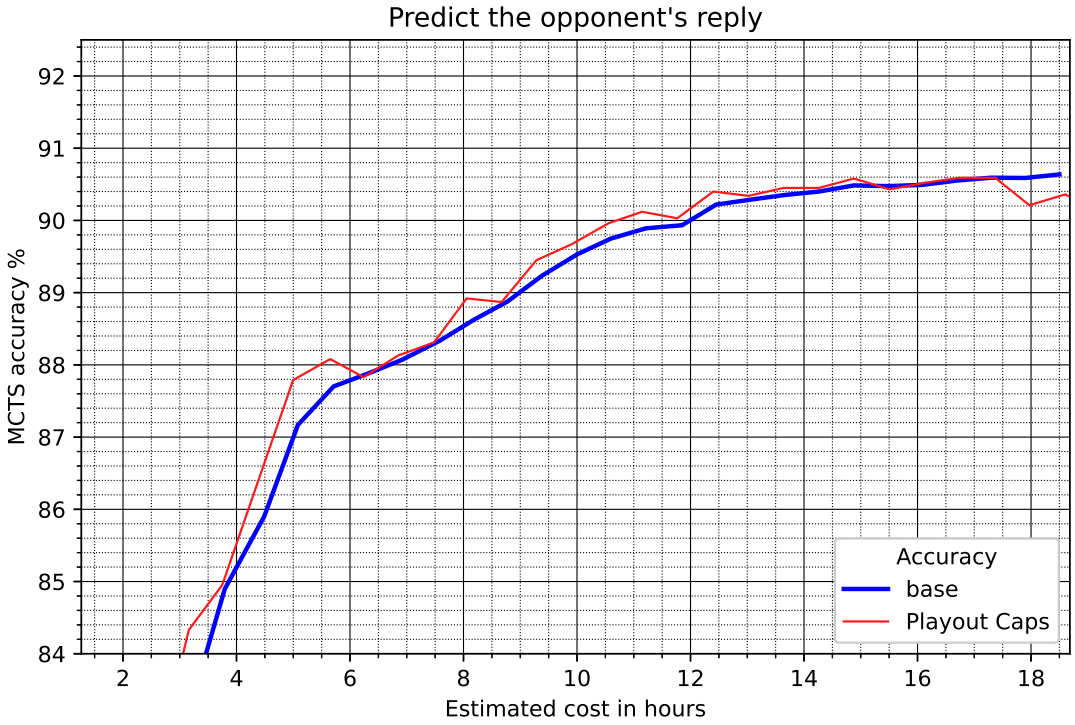

Figure 11: Results of implementing the prediction of the opponent's reply.

3.6.6 Improving the network structure

The Leela Chess Zero project [3] proposes using squeeze-and-excitation elements in the network. This is a general development of deep neural networks [21], which have been shown to improve learning efficiency at a small additional cost.

The network described in Table 2 on page 20 was modified for this. The resulting network is shown in Table 6.

Description	Network structure
Initial block	$\begin{pmatrix} 3 \times 3 \times 64 \\ BatchNorm \\ ReLU \end{pmatrix}$
Adapter convolution	$1 \times 1 \times 128$
SQ residual block, repeated n times	$\begin{pmatrix} 3 \times 3 \times 128 \\ BatchNorm \\ ReLU \\ 3 \times 3 \times 128 \\ BatchNorm \\ AVGPooling \\ FCnb : 8 \\ ReLU \\ FCnb : 128 \\ Sigmoid \\ Addition \\ ReLU \end{pmatrix}$
Move policy output	$\begin{pmatrix} 3 \times 3 \times 32 \\ FC : 7 \\ SoftMax \end{pmatrix}$
Win prediction output	$\begin{pmatrix} 3 \times 3 \times 32 \\ FC : 3 \\ SoftMax \end{pmatrix}$

Table 6: The modified network using squeeze-and-excitation residual blocks [21]. Squeeze-and-excite modifies the residual blocks to include an average pooling, which averages every feature map to a single scalar value. These scalar values are then processed by fully connected layers without bias, activated by ReLU and Sigmoid. $x \times y \times z$ describes a convolution with kernel size $x \times y$ and z filters. $FC : x$ describes a fully connected layer with x neurons. $Addition$ describes the addition with the input of the residual block the addition is a part of, forming the residual structure of the block. The move policy output and the win prediction output both are connected to the output of the last residual block. Residual blocks make up the bulk of the network. In most experiments of this thesis, 5 blocks are used.

A result of a test run can be seen in Figure 12. There might be some gains early in the training and possibly some small losses later in the training. Since this does not seem to hurt performance much and logically should help, especially given more distinct training data from deduplication, squeeze-and-excite elements were added to the extended baseline.

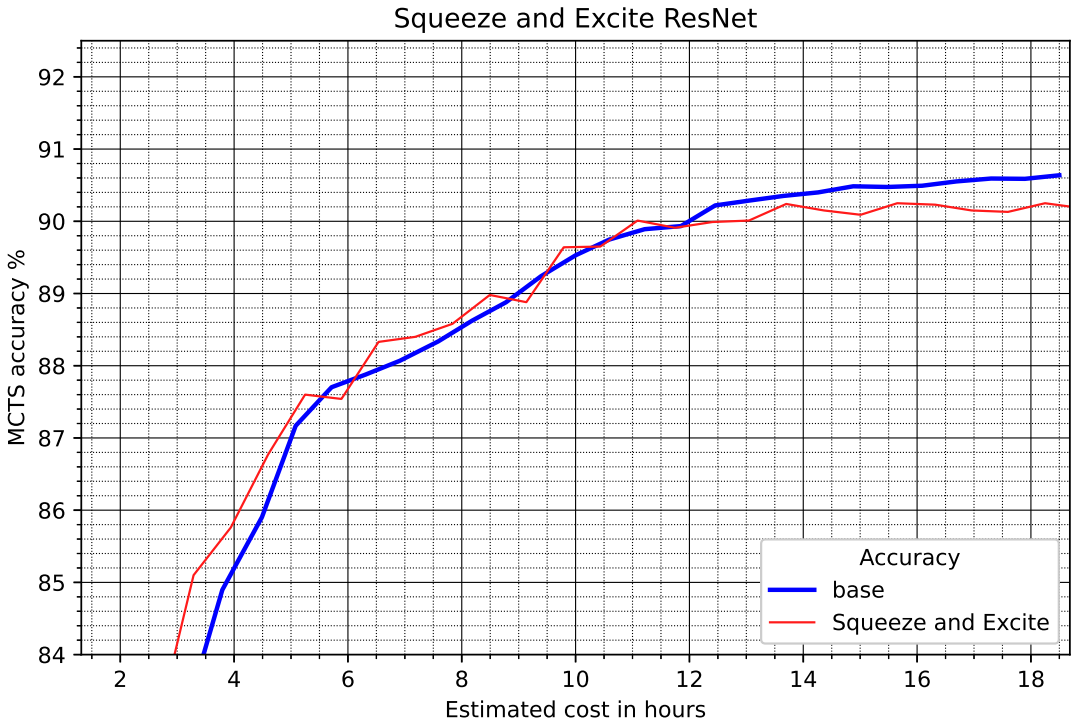

Figure 12: Results of implementing squeeze-and-excitation elements in the network.

3.7 Baseline results

Combining deduplication, a cyclic learning rate, a slow window, predicting the opponent's reply, and squeeze-and-excitation elements yields substantial improvements, yielding above 91% final performance in all runs on the hard dataset. Figure 13 shows the results. The easy dataset shows the same improved efficiency, further validating the improvements proposed by previous work. This extended baseline forms the basis of all further experiments. Unless stated otherwise, all these previous improvements are active.

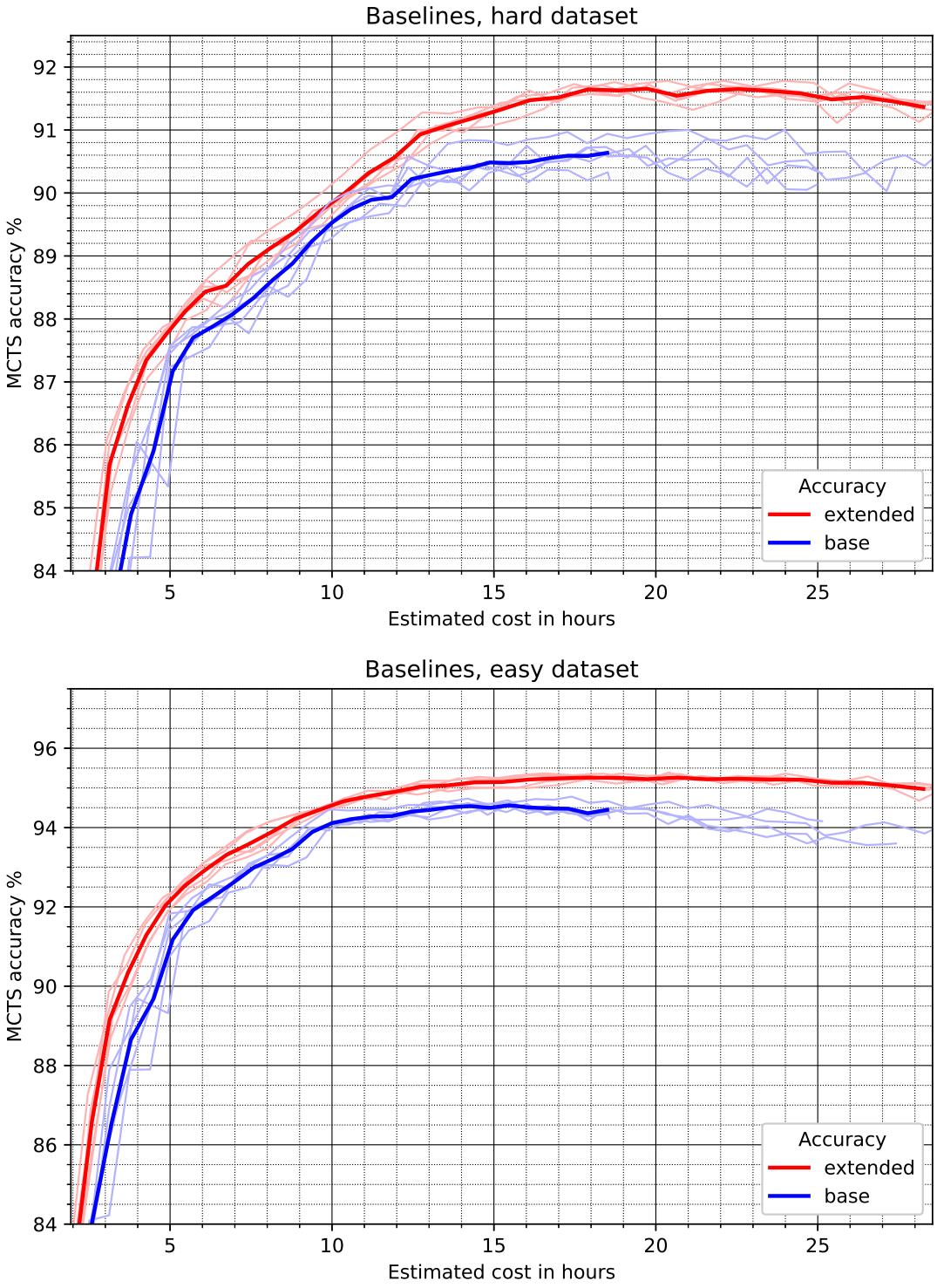

Figure 13: Comparison of the baseline and the extended baseline. Mean is only calculated until the first of the single runs stops showing improvements.

4 Investigated novel ideas

In this section, various novel ideas on possible improvements to AlphaZero are investigated experimentally. Three possible directions have been investigated: using the self-play phase as an evolutionary process, playing the self-play games in the form of a tree structure, and using internal network features as auxiliary learning targets for regularization.

4.1 Using the self-playing phase as an evolutionary process

Many hyperparameters involved in AlphaZero need tuning. An important subset of these hyperparameters controls the exact behaviour of the MCTS search, e.g. C_{puct}, as discussed in Section 2.2.1, Equation 4 on page 14. A hyperparameter search shows that these MCTS parameters have a notable impact on how much the MCTS can improve the playing strength of the network alone. Better parameters therefore could translate into faster training progression.

This thesis proposes using the self-playing phase of AlphaZero to optimize these hyperparameters by designating "players", which utilize different hyperparameters and have them play against each other in a league-system using a rating mechanism such as Elo. The games played in this league make up the self-playing phase, but the results of these games will also be used to judge which hyperparameters perform best. After some hyperparameters are found to perform best, the weaker players can be replaced with modified versions of the best players, forming an evolutionary process. The hope is to be able to search for good hyperparameters at no substantial additional cost.

4.1.1 Implementation

To implement the self-playing league, the central command server is extended to track a list of players and compute their rating based on game results reported by the self-playing workers.

For each new game to be played, the self-play workers randomly pick two players from the active population of players and let them play against each other. The result of the game is then reported back to the command server, which updates the rating of the players based on the result using Elo.

The central server tracks the population of players as a list, sorted by their Elo rating. At the start of the training, a set of initial players are randomly generated. Once the players of the most recent generation have played 1500 games, a new generation of players is created. For this purpose, the top 15 players each are mutated into two new versions of themselves. These mutated copies are then added to the list of players at

the same rating as their source players. Only the top 50 players are used as players to play new games, so old players fall out of the active population if their rating drops largely.

Games are again played, until enough games are played by the most recent generation, thus triggering another set of mutations to begin a new generation.

This cycle continues throughout the entire training run.

To be able to evaluate the accuracy of each iteration in such a training run, every time a new network is completed, the current set of players is saved. Then to evaluate a network, the best player of the players' generation that has not reported any games with a network newer than the one being evaluated is used.

4.1.2 Evolution of players

Gaussian mutation [37][page 7] is used to evolve the hyperparameters of the players. Valid parameter ranges are defined and enforced by looping values below the minimum back towards the maximum and the other way round. Each individual player is defined by a pair of two vectors (w_i, v_i). w_i represents the current value of hyperparameter i for $n \in \mathbb{N}$ hyperparameters: $i = 1, ..., n$, v_i represents the current variance vector for Gaussian mutations for hyperparameter i. Additionally, a range of valid values is defined for each hyperparameter: $(\min_i, \max_i)$.

The initial set of players selects values at random according to a uniform distribution over the allowed range of values. To create a new player (w_j, v_j), an already existing player (w_i, v_i) is used as shown in Equations 6 and 7, where $N(0, 1)$ is a value randomly picked from a Gaussian distribution with mean 0 and standard deviation 1. Once for the mutation of all hyperparameters, $N_i(0, 1)$ is the same, but a new value is randomly picked for every hyperparameter. τ' is $(\sqrt{2\sqrt{n}})^{-1}$ and τ is $(\sqrt{2n})^{-1}$.

$$v_j = v_i \exp(\tau' N(0, 1) + \tau N_i(0, 1)) \tag{6}$$

$$w_j = w_i + v_j N_i(0, 1) \tag{7}$$

4.1.3 Selection of hyperparameters

There are many hyperparameters in AlphaZero, but it is unclear which ones might be best optimized by evolutionary self-play. Only parameters that can be changed easily during a training run can be considered, so mainly parameters that control the behaviour of the MCTS search. First candidates are parameters such as cpuct, fpu, and drawValue, as described in Table 4 on page 26.

An additional candidate is to control the size of the MCTS tree by the Kullback-Leibler divergence as more nodes are added to the MCTS tree, as suggested by the Leela Chess Zero project [5].

For this purpose, the self-play workers are changed to work in steps of adding 50 nodes to the current batch of trees, then checking for each game if the kldgain is high enough to continue adding nodes or if the move in that game should be played with the current number of nodes. In particular, the Kullback-Leibler divergence is calculated between the previous output of the MCTS search and the new output of the MCTS search, which has 50 additional nodes added to it. If the binary logarithm of the Kullback-Leibler divergence is above a threshold, another 50 nodes will be added to the MCTS. The task of the player evolution is to find a good value for the threshold.

To limit the maximum number of nodes, a ”clock“ is implemented, where for each game each player starts with a budget of 4650 allowed node evaluations and gets an additional 50 each turn. For a game of average length, this will play out around 350 node evaluations per move, which is the same as without this extension. If a player is out of node evaluations, they will play with only 50 nodes per MTCS for the rest of the game.

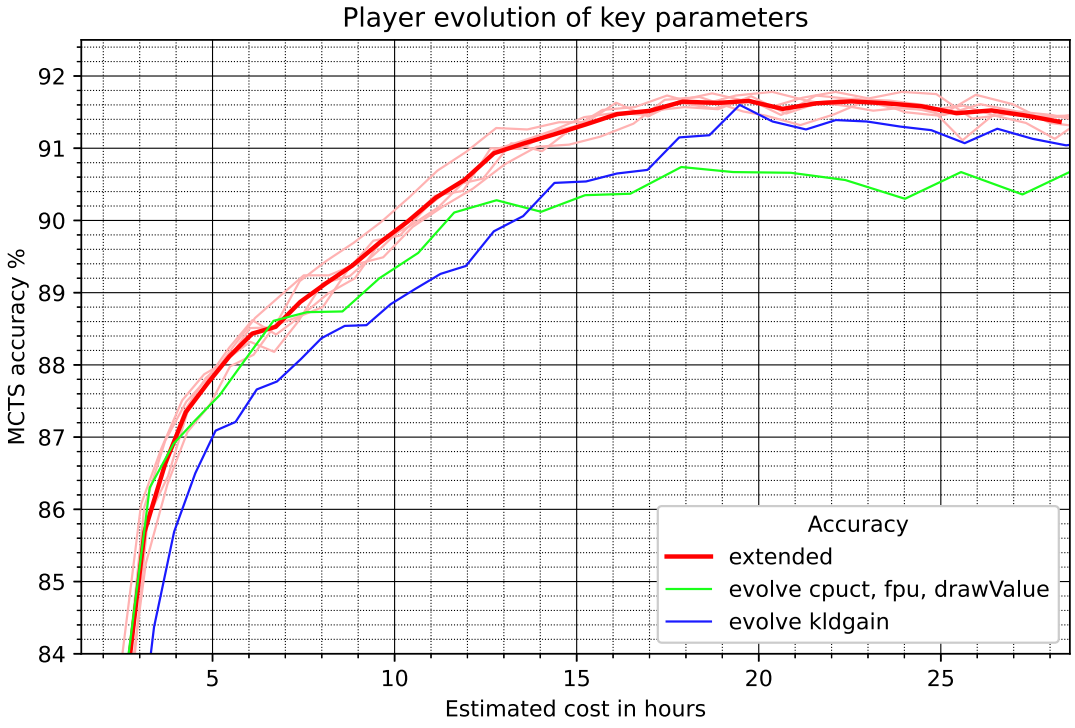

Figure 14: Initial results of evolving hyperparameters.

4.1.4 Experiments

The evolutionary approach was implemented, and initial experiments run — one evolving cpuct, fpu, and drawValue, and the other the Kullback-Leibler divergence threshold for playing with a dynamic number of MCTS nodes. Results can be seen in Figure 14. Neither approach produced promising results, under-performing the baseline by a margin much larger than the run-to-run difference within the baseline.

One potential issue can be seen for the evolution of the MCTS hyperparameters in Figure 15, which shows a substantial drop in the game play diversity when evolving those parameters.

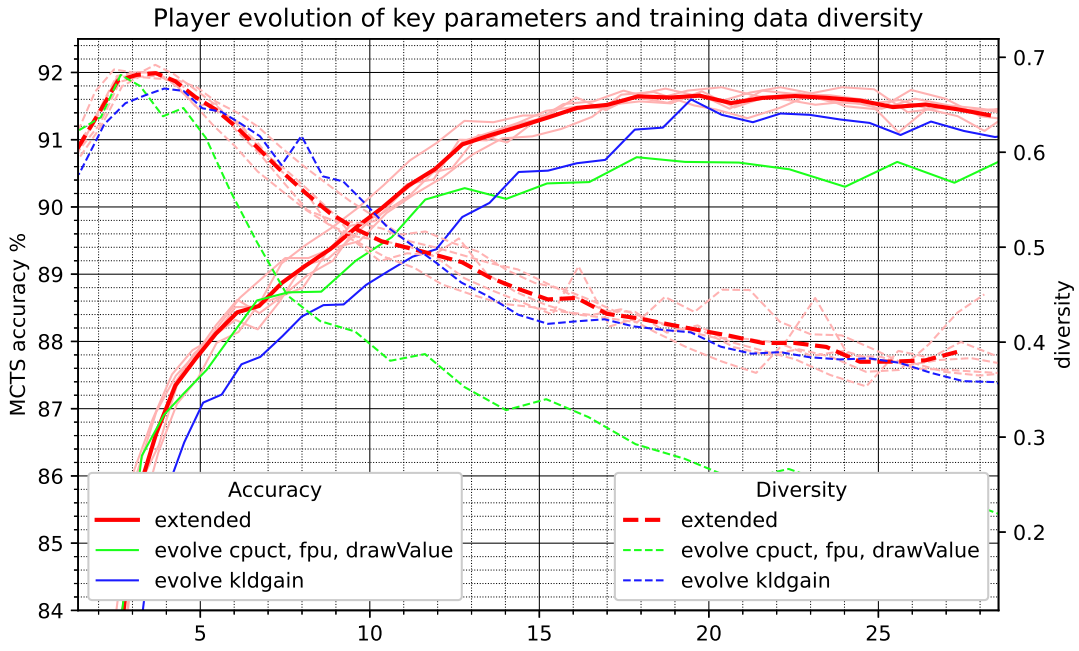

Figure 15: Evolving basic MCTS parameters results in much lower game diversity, evolving the KL divergence threshold remains similar to the baseline.

4.1.5 Requirements for evolution to succeed

First experiments with evolving players do not show good results, raising the question of what conditions have to be fulfilled to improve training efficiency with the evolutionary process. Two key points can be established:

- The player league needs to be able to pick players that actually maximize playing strength.

- Players that win a lot of games against other players should correspond to players that produce a high accuracy of play compared to the Connect 4 solver.

To investigate if the player evolution is able to find "strong" players, tests were done with another hyperparameter, designed only for this purpose, the inversion-hyperparameter $\mathbf{I}$. $\mathbf{I}$ modifies the move distribution found by MCTS in a way designed to disrupt playing strength.

If m is the vector that represents the move distribution produced by MCTS, then the new move distribution is calculated by the inversion parameter as shown in formula 8. To explain in words, the inversion parameter controls a linear interpolation between the original move distribution and a vector that is one minus the original move distribution. Therefore, when $\mathbf{I}$ is 1, the MCTS will produce results which put most weight on moves that were originally considered to be the worst moves, while the best moves will receive the lowest weight. $\mathbf{I}$ should be 0 so as not to disturb the MCTS results in any way, giving a known optimal value towards which evolution should optimize.

$$m_{\text{inverted}} = \mathbf{I}((1-m)/\sum(1-m)) + (1-\mathbf{I})m \tag{8}$$

As a first test of the players' league evolution concept, it is tested how well this parameter is optimized towards zero. To this end, runs of 10 player generations are done, evolving $\mathbf{I}$ in the range between 0 and 1. Table 7 shows the development of the average players over generation 3 to 10, as a mean of 3 runs. Earlier generations were not recorded, because only generations after network iteration 1 are stored. To gain further insight into how strong the optimization is in the face of reduced impact of the parameter, two more sets of 3 runs were made, where the inversion was only applied at random with a probability p. p was chosen to be 0.25 and 0.05, so in the most extreme case, only 1 in 20 moves would be affected.

The results show that the evolution of players works and can pick parameters that play stronger, even when there is a lower influence of the hyperparameter on game results, such as with $p = 0.25$ or even $p = 0.05$. Full training runs typically go for 30 to 50 generations, so they have even more generations to optimize parameters.

Generation	**I**, $p = 1$	**I**, $p = 0.25$	**I**, $p = 0.05$
3	0.149	0.269	0.446
4	0.084	0.217	0.380
5	0.071	0.172	0.383
6	0.052	0.150	0.307
7	0.047	0.109	0.290
8	0.038	0.106	0.244
9	0.029	0.091	0.243
10	0.027	0.085	0.248

Table 7: Results of optimizing the inversion hyperparameter. The evolution works, **I** is reduced substantially over the generations.

After establishing that the evolution works in general to optimize parameters, it is now important to also establish how well such players do in the context of accuracy compared to the Connect 4 solver.

For this purpose, three sets of MCTS hyperparameters are compared: the hyperparameters used in the extended AlphaZero baseline, the hyperparameters found by evolution, and a set of MCTS hyperparameters, which were found by running 65 steps of Bayesian optimization directly towards maximum accuracy compared to the solver. This is different from the Bayesian optimization used to find the hyperparameters for the AlphaZero baseline, as no full AlphaZero run is done. Instead the fitness function runs MCTS directly on a set of test positions and returns the accuracy.

The three hyperparameter sets are compared on a random network, a network after a single training iteration, and the best network of the training run. Table 8 shows the results. The random network has no evolved parameter set, as evolved parameter sets are only generated while also training a network.

Random network

Parameter Set	Accuracy	cpuct	drawValue	fpu
Evolved Player	-	-	-	-
Baseline	0.6227	1.545	0.6913	0.8545
Bayesian Opt.	0.6712	0.08295	0.106	0.9995

Network iteration 1

Parameter Set	Accuracy	cpuct	drawValue	fpu
Evolved Player	0.7415	0.5328	0.8753	0.9912
Baseline	0.7512	1.545	0.6913	0.8545
Bayesian Opt.	0.7613	0.8162	1.0	0.5255

Best network iteration

Parameter Set	Accuracy	cpuct	drawValue	fpu
Evolved Player	0.9068	0.2925	0.4728	0.1411
Baseline	0.906	1.545	0.6913	0.8545
Bayesian Opt.	0.9117	0.9045	0.4498	0.000977

Table 8: Different hyperparameters show different accuracies. Directly optimizing for accuracy shows that neither the baseline, nor the evolutionary hyperparameter perfectly captures correct play.

These results provide multiple interesting insights:

- Once the training run has reached the best network, the evolved player has found a parameter set that performs equally good to the baseline parameters; however, it starts weaker.
- Neither the baseline nor the evolved player reaches the performance of a hyperparameter set only optimized to reach highest accuracy.
- The optimized parameter set shows how different hyperparameters are good for different phases of the training. Especially the fpu changes drastically, from the maximum possible value of 1 to the minimum possible value of 0. It seems setting cpuct low, but fpu high is an efficient way to explore with a random network. Similarly, the drawValue also starts at a higher value and is reduced later in the training run. The evolved player captures both these developments over the run, reducing the drawValue and the fpu, once the best network is generated. Nevertheless, evolution prefers lower cpuct values.

Lastly, the three hyperparameter sets are compared in matches of 1000 games against each other in order to determine which sets actually win the most games when playing against each other using the same network. Games are played by picking a move

according to the probability distribution produced by MCTS at random. For the evolution approach to actually improve learning efficiency, a player that wins most games should be the same as a player with a high MCTS accuracy, or else the win rate of the hyperparameters would not be a good proxy for faster training progress. Table 9 shows the results of the matches between the different hyperparameter sets.

Random network

	Baseline
Bayesian Opt.	$625W, 360L, 15D$

Network iteration 1

	Bayesian Opt.	Baseline	Evolved Player
Bayesian Opt.	-	$581W, 366L, 53D$	$380W, 526L, 94D$
Baseline	$366W, 581L, 53D$	-	$291W, 665L, 44D$
Evolved Player	$526W, 380L, 94D$	$665W, 291W, 44D$	-

Best network iteration

	Bayesian Opt.	Baseline	Evolved Player
Bayesian Opt.	-	$590W, 238L, 172D$	$388W, 422L, 190D$
Baseline	$238W, 590L, 172D$	-	$155W, 557L, 288D$
Evolved Player	$442W, 388L, 190D$	$557W, 155L, 288D$	-

Table 9: Results of 1000 games between different players. Results are from the perspective of the player in the row against the player of the column. The evolved player wins every single match.

The high win rate of the evolved player shows that the hyperparameter optimization has achieved what its optimization goal requires: Win more games. It appears this is not necessarily the same as playing in such a way as to play correctly according to the Connect 4 solver and is also not generally helpful to speed up the training processes.

Most likely this is the core reason why the evolution approach does not help efficiency: The win-rate of hyperparameters in playing against each other using the same network is not a good proxy for learning progress of AlphaZero.

4.1.6 Novelty search as an optimization target

Since the win-rate of players does not appear to be a good optimization target, it might be useful to instead optimize towards a novelty search objective. Novelty is a useful optimization target to solve RL problems in Deep Learning, as well as in evolutionary approaches [27, 22].

Figure 15 established that optimizing for winning games causes MCTS hyperparameters to be selected that reduce the game diversity substantially, which is likely to hurt exploration. As a compromise, it is proposed to replace the Elo league with a point-based league where the winner of a game is given points for every new position in a won game. Therefore, a hyperparameter set has not only to win games but also to win them in novel ways. This might help to increase the game diversity. Apart from the different objective, the player-league implementation remains unchanged. Figure 16 shows that optimizing novel ways in this manner does not substantially increase game diversity; only in the later part of the training does a minimally higher game diversity seem to be created. Since players still have to win to gain points, they still mainly focus on winning games, as a 1000-game match between the novelty-evolved player and the extended baseline parameters shows, see Table 10.

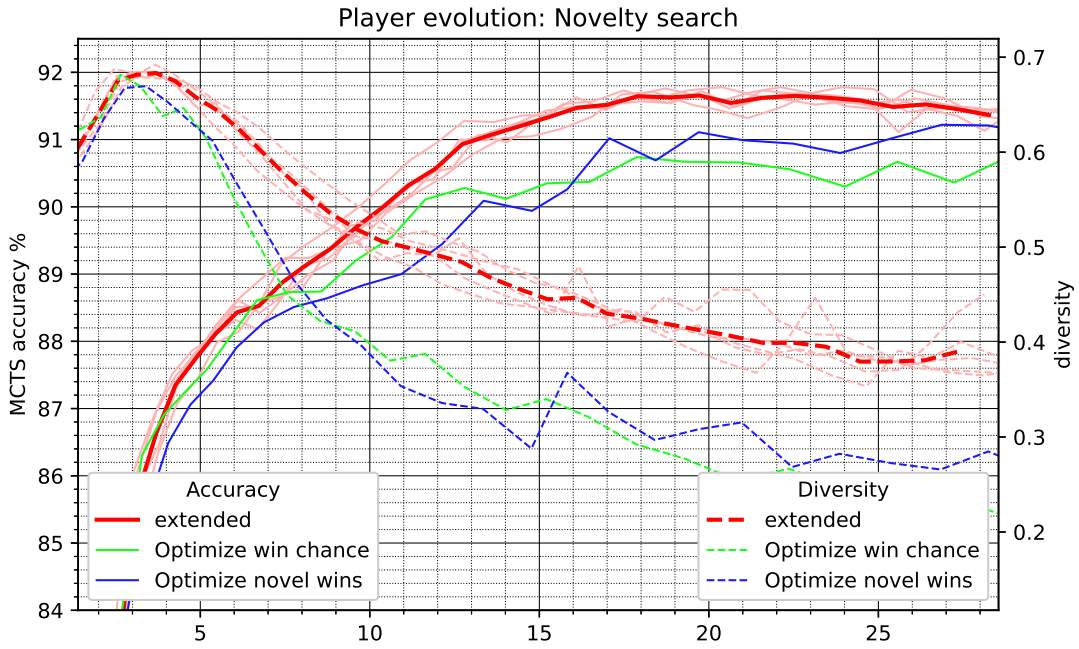

Figure 16: Novelty search for novel ways to win the game shows little difference to just optimizing for wins.

Match using fully trained network

	Baseline
Evolved	$599W, 238L, 163D$

Table 10: Rewarding novel wins instead of just wins makes no substantial difference; players mostly still get optimized towards winning games. Results are from the perspective of the player in the row against the player of the column.

Finally, it is possible to abandon the idea of rewarding winning at all and optimize for novelty alone. This means both players get points at the end of the game, one for every novel position encountered in the game. 3 runs were done using this optimization target; Figure 17 shows the results. Towards the second half of the training, game diversity is clearly better than with the baseline; however, game play performance lags substantially behind. In the first half of the training, the baseline produces more game diversity. This is likely because the evolution needs some time to find good parameters, while the baseline starts with parameters that were already optimized to produce good results. The only small advantage in game diversity, even in the latter half of training, indicates that the baseline parameter optimization already considered game diversity, and that there is not much space for improvement.

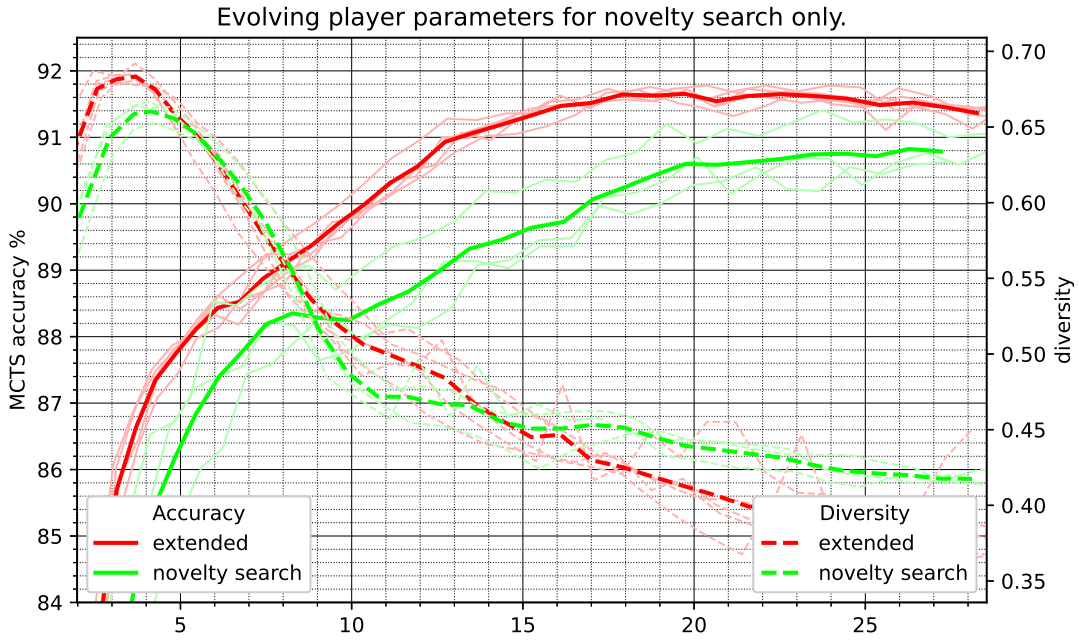

Figure 17: Even pure novelty search only produces more novel games towards the end of the training. The baseline parameters were likely to be already implicitly optimized for game diversity by the initial Bayesian hyperparameter optimization aiming to learn efficiently.

This concludes the investigation into the evolutionary optimization of hyperparameters. No meaningful improvements could be found. However, the evolution of hyperparameters using self-play has been shown to work, so future work might be able to identify better optimization targets that align better with the overarching goal of AlphaZero training.

4.2 Playing games as trees

The standard AlphaZero implementation explores new possible moves mainly by the randomization of moves, according to the policy found by MCTS. This thesis investigates an alternative scheme of exploration, by instead playing out games in a structure similar to a tree.

A first option to do this involves resetting games to a position in which the losing player probably made a mistake and try out a different move from that position.

Another option is to abandon the idea of playing independent games to explore entirely and instead focus on playing one large MCTS tree, which explores the current space of gameplay possibilities by the MCTS algorithm.

4.2.1 Implementation

To better control the games played, all games are played out on a single machine. In order to still scale to many GPUs, an MCTS evaluation service is created. The one machine playing the games queues positions to be evaluated and waits for the results to return. To fully utilize several GPUs, many thousands of games are played concurrently. The machine playing the games caches the MCTS search results for positions and only queues every position once per network iteration. This way of implementing AlphaZero increases the efficiency of GPU usage, as it prevents duplicate MCTS on common game positions. It is however only an implementation detail and cannot be considered an improvement to the algorithm. This also changes the way the training costs are measured. To simplify the measurement of the time to play a single move, the costs are measured in the MCTS evaluation service. Hence, the cost is calculated as the number of requested evaluations multiplied by the cost to do a single evaluation on the ”standard“ machine.

Since this different implementation is more efficient than to the standard baseline, normal training runs are done using the MCTS evaluation service with no further changes in the algorithm to establish a new baseline. Any algorithmic improvements which need the MCTS evaluation service are compared against this new baseline. The baseline is compared with the previously defined extended baseline in Figure 18.

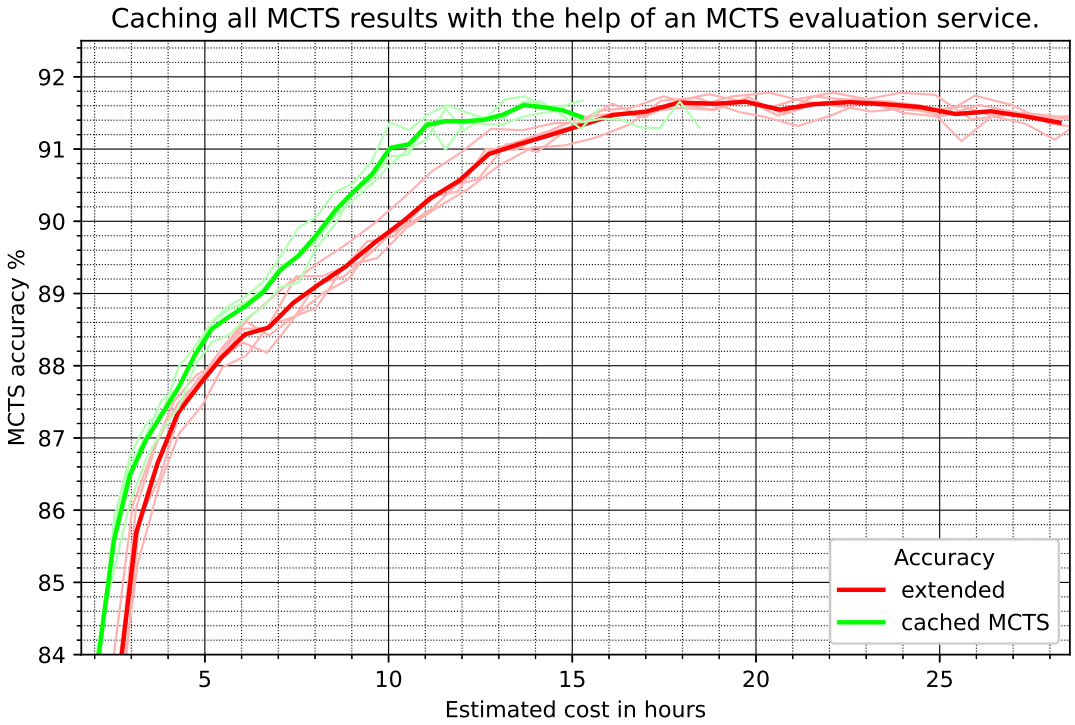

Figure 18: The MCTS evaluation service is a more efficient implementation of AlphaZero.

4.2.2 Resetting games to a position before a likely mistake

This idea aims to explore by trying out to play another move in a position that might have been the reason for the loss of a game.

Randomized play is reduced to the first 16 turns of a game, down from 30, to focus exploration on the novel way investigated here. Instead of relying on randomized moves to explore, once a game is completed, it is reset to an earlier position and replayed with a different choice of moves from there. This choice of moves is intended to be done in a way that explores new gameplay possibilities.

The suggested way to choose a new move is to first search the position in which the predicted chance of the losing player to win dropped considerably compared to before the latest move was made. For this the position value prediction of the network is used. This should indicate a position in which a likely bad move was played.

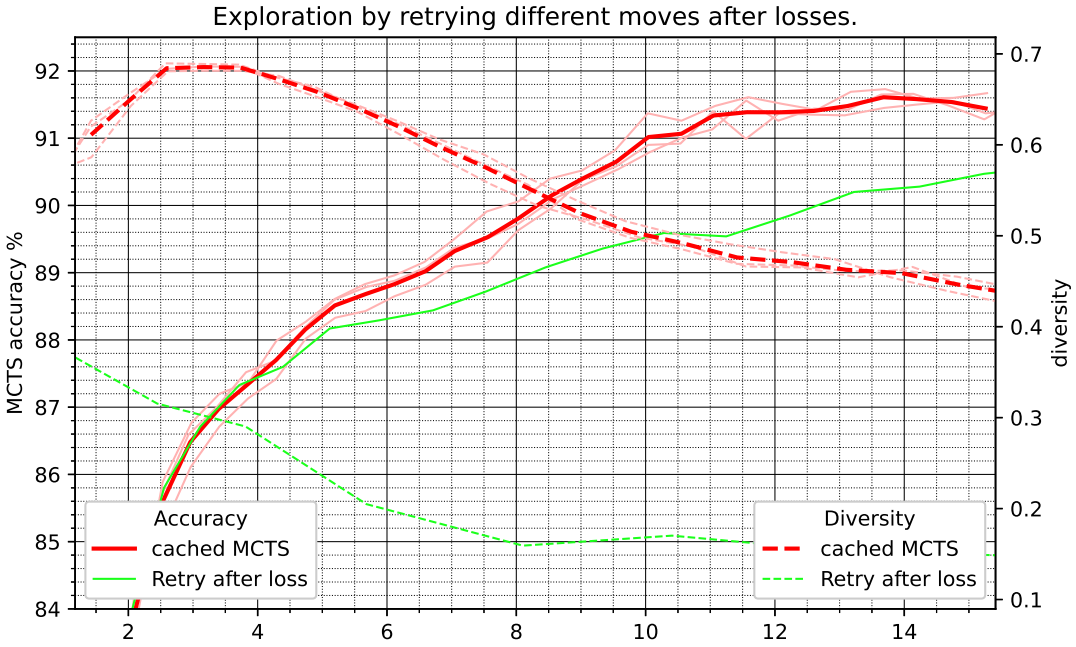

Figure 19: Retrying a different move in a critical position to explore does not appear to work.

From one position earlier, before the network value evaluation dropped, a new game is started and lets the losing player pick the best move in the position that has not been played yet. This way the player who lost a game gets another attempt to rectify a mistake that might have led to the loss, thereby exploring a new branch of the game tree that potentially represents better play.

Figure 19 shows results of this idea. The results are not good. It appears that the randomized moves are substantially better at exploration, with game diversity drastically reduced when exploring by retrying a different move in a critical position.

4.2.3 Explore the game tree in the same fashion as MCTS

The exploration exploitation dilemma faced by AlphaZero self-play is the same as that MCTS aims to solve. It might thus make sense to explore the game tree by MCTS instead of randomized self-play. Specifically, creating a tree of game position, by MCTS rules, is proposed to generate example games and game results. Since AlphaZero needs to scale to many distributed machines, virtual losses [19] are used to play out many games using the same MCTS tree. This should not be confused with the MCTS used to evaluate single-game positions. That per-position MCTS is still used to evaluate positions, effectively stacking two MCTS searches into each other. The one big MCTS that generates training data and replaces randomized self-play uses the

per-position MCTS as position evaluations, just like the per-position MCTS uses the neural network.

A virtual loss is given to every node visited, pushing following games to investigate other nodes if too many such losses push down the value of a node in the search tree. Every game played represents one thread walking down the tree, adding virtual losses to visited nodes. Unlike in AlphaZero MCTS, games have to be played out to a final result before backpropagation can occur.

To fully utilize 10 or more GPUs, thousands of games have to be played in parallel. This might prove a challenge to the virtual loss concept, as the original paper proposing virtual losses tested with only up to 16 threads, whereas in this case thousands will be necessary.

Another challenge is the exchange of the network once a new network has been trained. This makes it necessary to re-evaluate all positions in the tree. If one is not careful, this can substantially increase the training cost, as lots of positions have to be evaluated every time a new network is trained before any new game results can be reported, since all games have to start from the first turn again. A game can only be reported once it has been played through fully. Therefore, if a game is half-completed when a new network is created, there is the choice between throwing away the game, since it is based on an old network, and accepting the game reports into the training data anyway.

Throwing away the MCTS tree when a new network is completed would implement throwing away the half-finished games, which is too expensive. Even after a new network has been trained, games that finish with positions only evaluated by the previous network are still reported. Instead of throwing away the MCTS tree, nodes are only re-evaluated as they are encountered again, with currently running games untouched. This is similar to how normal AlphaZero might report some game states to the training server that use an older network due to the distributed nature of the workers.

One more difference with the MCTS implementation to discover new gameplay is that it uses a hash table between a game position and tree nodes. This way, for every game position only a single node can exist, handling game transpositions in a rudimentary way. However, for the backpropagation of game results, transpositions are ignored and only the recently played path from the root is back-propagated over.

Experiments with this self-play approach quickly revealed that MCTS tends to get stuck on a relatively low number of possible games, dropping game diversity and hurting exploration. Figure 20 shows that even setting the exploration exploitation hyperparameter cpuct of the MCTS to extremely large values up to 15 does not help prevent this. Increasing game diversity remains out of reach entirely.

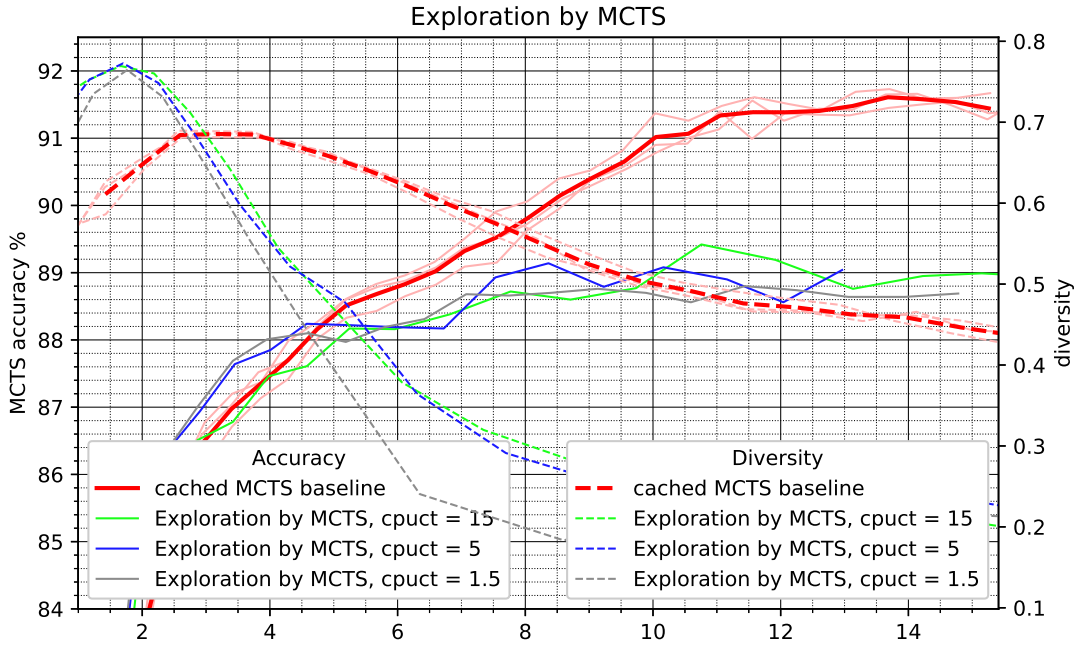

Figure 20: Replacing self-play with one large MCTS. MCTS tends to get stuck on a few paths of games and stops exploring, reducing the diversity of game positions encountered. Even increasing cpuct to very high values does not prevent this.

4.3 Using network internal features as auxiliary targets

Wu et al. [36] show that using domain-specific features as auxiliary targets to regularize learning improves learning in Go. This motivates the search for an automatic way to find such targets.

4.3.1 Implementation

This thesis proposes to use internal feature representations of a small network, shown in Figure 11, to be used as such auxiliary targets. To this end, a full AlphaZero run is done, learning to play Connect 4 using a very small network with fewer than 70000 parameters, i.e. a fraction of the 1.6 million parameters of the network used in all other experiments. This small network uses only a single filter in the last convolutional layer for both output-heads of the network. The prediction of the policy as well as the prediction of the game outcome is bottlenecked by only 42 features each, one feature for each of the 6×7 fields of Connect 4.

Training such a small network to an accuracy of about 84% takes about two hours of computational time on a single GPU. The network can afterwards be used to generate

features when training in an AlphaZero run with a larger network. There are multiple options to use these features precisely. First a distinction can be made between the features of the game outcome prediction, the win-features, and the features of the policy prediction, the move-features.

For a given training example, the small network is run on it to produce these feature sets, producing the features of the present position. Additionally, more features can be generated by considering future positions that occurred in the game after a training position. This defines the future features, which may induce the ability to predict the future of a game position in the regularized network. In this thesis future positions 2 and 4 ply in the future are considered. Including the present position this gives up to three feature layers to be used as regularization targets. They are named move0 to move2 and win0 to win2, 0 standing for the present, 2 for the 4 ply future features. move1 and win1 also include move0 and win0, respectively; move2 and win2 include all layers. Moreover, it was tried to only regularize with some of the future-feature layers and not the earlier ones; this is named only move1, only move2, only win1, and only win2.

To regularize the bigger network a mean squared error is used to add to the training loss, calculating the mean squared error between the auxiliary feature and an internal layer of the big network. No additional parameters are added to the bigger network; only the internal layer from which the features in the small network were taken is regularized to produce the same features in the bigger network. Since the bigger network has many more filters, it still has a lot more capacity to learn beyond that.

Description	Network structure
Initial block	$\begin{pmatrix} 3 \times 3 \times 128 \\ BatchNorm \\ ReLU \end{pmatrix}$
Adapter convolution	$1 \times 1 \times 32$
SQ residual block, repeated 3 times	$\begin{pmatrix} 3 \times 3 \times 32 \\ BatchNorm \\ ReLU \\ 3 \times 3 \times 32 \\ BatchNorm \\ AVGPooling \\ FCnb : 4 \\ ReLU \\ FCnb : 32 \\ Sigmoid \\ Addition \\ ReLU \end{pmatrix}$
Move policy output	$\begin{pmatrix} 3 \times 3 \times 1 \\ FC : 7 \\ SoftMax \end{pmatrix}$
Win prediction output	$\begin{pmatrix} 3 \times 3 \times 1 \\ FC : 3 \\ SoftMax \end{pmatrix}$

Table 11: The small bottleneck network used as a source of auxiliary learning targets. The network has about 70000 parameters. The network primarily has much fewer parameters, as it uses only 32 filters throughout the residual blocks, instead of 128. Both the outputs are based on a convolution with a single filter, yielding 42 features for each win prediction and move policy. Those 42 features are used as auxiliary features for game positions and are learnt by the bigger network as a regularizer.

4.3.2 Supervised

To first establish where various combinations of these options might stand, supervised training is enhanced with the auxiliary features. The results are shown in Table 12. It can be seen that regularizing with the features of the move output produces very small improvements outside the standard deviation of the runs. Using the win features worsens the results of supervised training.

configuration	accuracy moves	accuracy wins
Baseline, no auxilary features	$92.6\% \pm 0.1$	$78.94\% \pm 0.1$
move0	$92.76\% \pm 0.1$	$78.98\% \pm 0.06$
win0	$91.45\% \pm 2.23$	$\mathbf{79.98\% \pm 2.04}$
move0, win0	$92.81\% \pm 0.08$	$78.96\% \pm 0.06$
move1	$\mathbf{92.89\% \pm 0.07}$	$79.09\% \pm 0.06$
win1	$92.51\% \pm 0.05$	$78.82\% \pm 0.07$
move2	$92.80\% \pm 0.05$	$78.99\% \pm 0.06$
win2	$92.35\% \pm 0.02$	$78.75\% \pm 0.06$
only move1, only win 1	$92.57\% \pm 0.06$	$78.86\% \pm 0.05$
only move2	$92.79\% \pm 0.05$	$79.05\% \pm 0.11$
only win2	$92.41\% \pm 0.12$	$78.81\% \pm 0.11$

Table 12: Supervised results for auxiliary features. Mean and standard deviation of 5 supervised runs, using the same setup as in Section 3.4 on page 23

Since the differences are quite small, another set of supervised experiments was done using the easier dataset with only 10% random moves played during dataset generation, as described in Section 3.2.1 on page 21. The results mirror the results of the harder dataset. Once again move1 shows the most improvement, albeit a very small difference, outside the standard deviation of the runs.

configuration	accuracy moves	accuracy wins
Baseline, no auxilary features	$96.94\% \pm 0.02$	$84.75\% \pm 0.04$
move0	$97.04\% \pm 0.05$	$84.79\% \pm 0.03$
win0	$96.08\% \pm 1.5$	$83.89\% \pm 1.15$
move0, win0	$97.04\% \pm 0.071$	$84.71\% \pm 0.05$
move1	$\mathbf{97.16\% \pm 0.04}$	$\mathbf{84.81\% \pm 0.03}$
win1	$96.89\% \pm 0.04$	$84.66\% \pm 0.05$
move2	$97.10\% \pm 0.05$	$84.73\% \pm 0.02$
win2	$96.79\% \pm 0.09$	$84.58\% \pm 0.04$
only move1, only win 1	$97.00\% \pm 0.08$	$84.71\% \pm 0.04$
only move2	$97.04\% \pm 0.07$	$84.75\% \pm 0.03$
only win2	$96.91\% \pm 0.07$	$84.67\% \pm 0.05$

Table 13: Supervised results for auxiliary features on easy dataset. Mean and standard deviation of 5 supervised runs, using the same set-up as in Section 3.4 on page 23, with the easier dataset as described in Section 3.2.1 on page 21

4.3.3 AlphaZero runs

Following from these results, move1 has been picked as the best candidate for a successful AlphaZero run. Additionally, however, experiments have been run with a few other configurations to verify if supervised results correspond with AlphaZero results. Win0 was chosen because of its unstable results in supervised, combined move0 and win0 because of how close they were to move1. Finally, only move2 was chosen to include a configuration as far as possible in the future. In the following data, the cost of the runs with auxiliary features has been increased by 2 hours to include the training of the network used as a source of features. Results are shown in Figure 21.

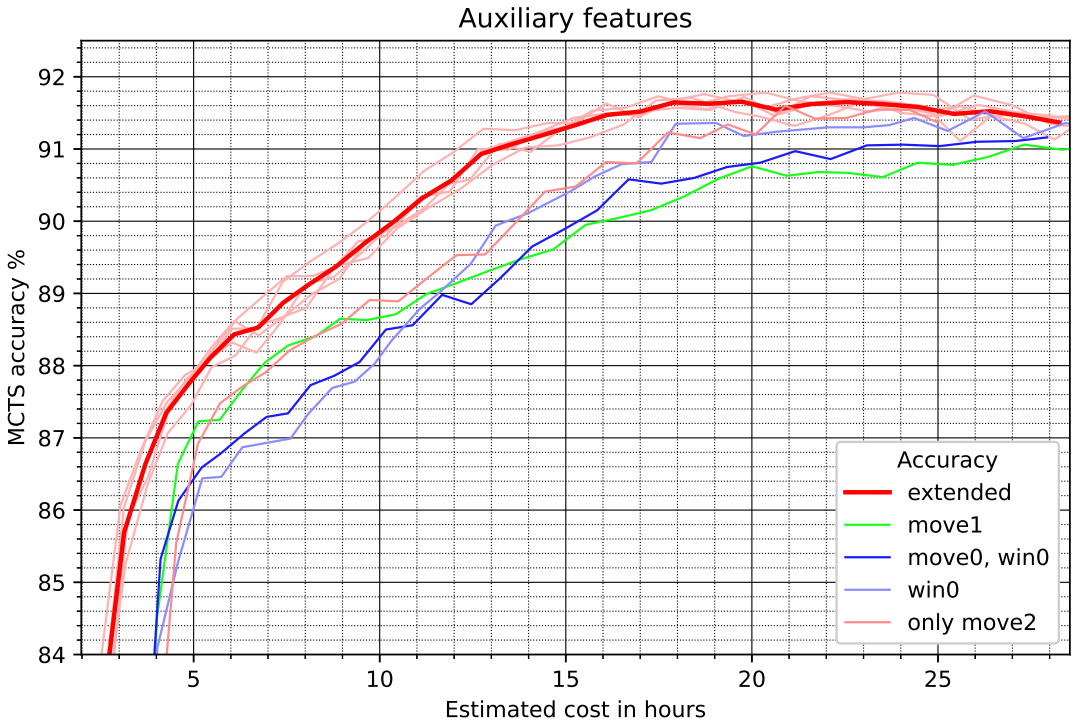

Figure 21: Using auxiliary features from a smaller network with AlphaZero.

The results are not very promising. While in the supervised setting the final accuracy was slightly higher, here it is actually lower. The extra cost of 2 hours of time to train the initial network increases the cost, thereby making the use of auxiliary features like this not worthwhile. The first few iterations, however, do show a faster rise in accuracy when the extra costs are ignored, as displayed in Figure 22. Even without the issue of the extra costs, the baseline reaches a substantially higher final result. An interesting result is found when using a random network for auxiliary features, also shown in Figure 22, instead of one trained in a previous AlphaZero run. This reduces the costs for initialization to 0. The training run with random features follows the extended

baseline very closely. It seems to fall off minimally towards the end, but that might just be a run-to-run variance.

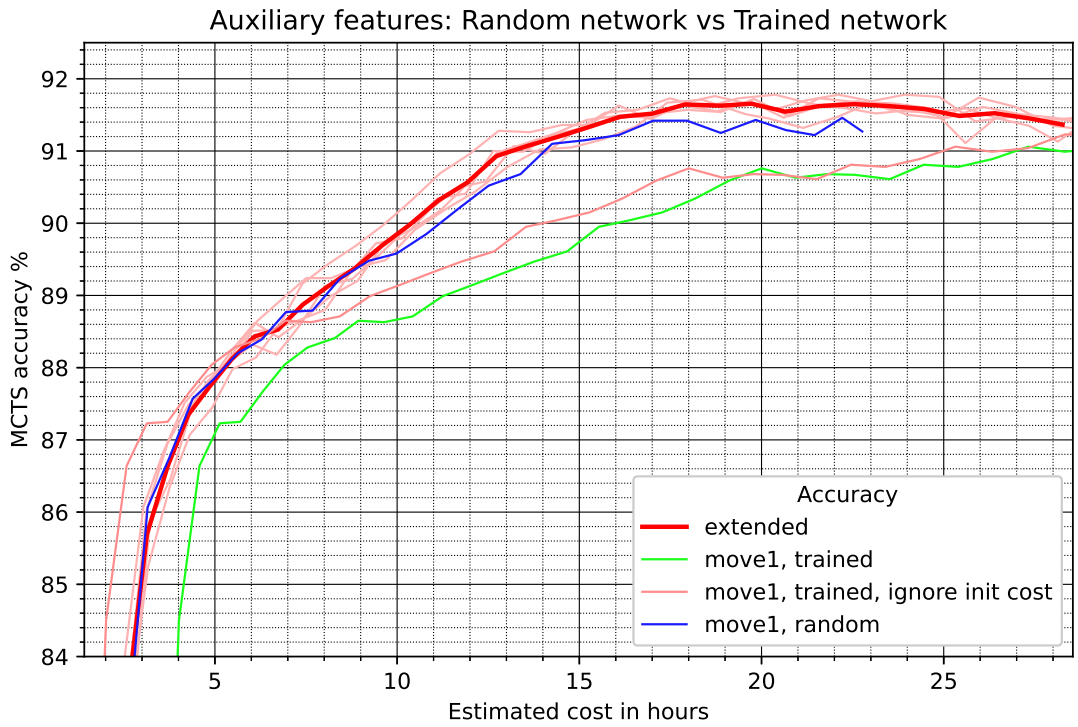

Figure 22: The training costs of the feature network push the auxiliary feature-enhanced training run behind the baseline.

In summary, using a pre-trained smaller network for features appears to help at the beginning of the training, but not enough to pay for the costs of training the network. Moreover, the performance in later parts of the training is held back, likely by the regularization that keeps enforcing the auxiliary features which are only useful for prediction at an accuracy level of about 84%.

This raises the following questions: Can the initial cost be reduced? Can the negative influence in the later parts of the training be prevented?

4.3.4 Growing the network

An important observation related to these questions is that the training of the smaller network used for the features, reached 84% in just 2 hours of training. This is a lot faster than the baseline, which requires about 3 hours for the same accuracy. Starting training with a smaller network and then switching over to a bigger network does in fact increase learning efficiency, as can be seen in Figure 23. Specifically, the number

of filters in the convolutional filters is grown in this run, starting with 16 filters and doubling the number of filters in iterations 4, 8, 16, and 32. This means that by iteration 32 the network will use twice as many filters as the other networks in this thesis. The switchover results in visible drops in accuracy, which get bigger the more the network size grows. This is not unexpected as training just a single epoch of the window of two million examples is not enough to train the bigger network back to where the previously used smaller network was. This becomes especially visible in the last step, where the biggest network causes a sharp drop in performance. The earlier parts of the training, though, look very promising in terms of efficiency gains.

This is not a novel idea. Growing the size of the network as its playing strength rises has been done by the LCZero project [10], as well as in other AlphaZero-related works such as those by Wu. et al. [36]. Nevertheless, there appears to be no specific research on how to maximize the effect.

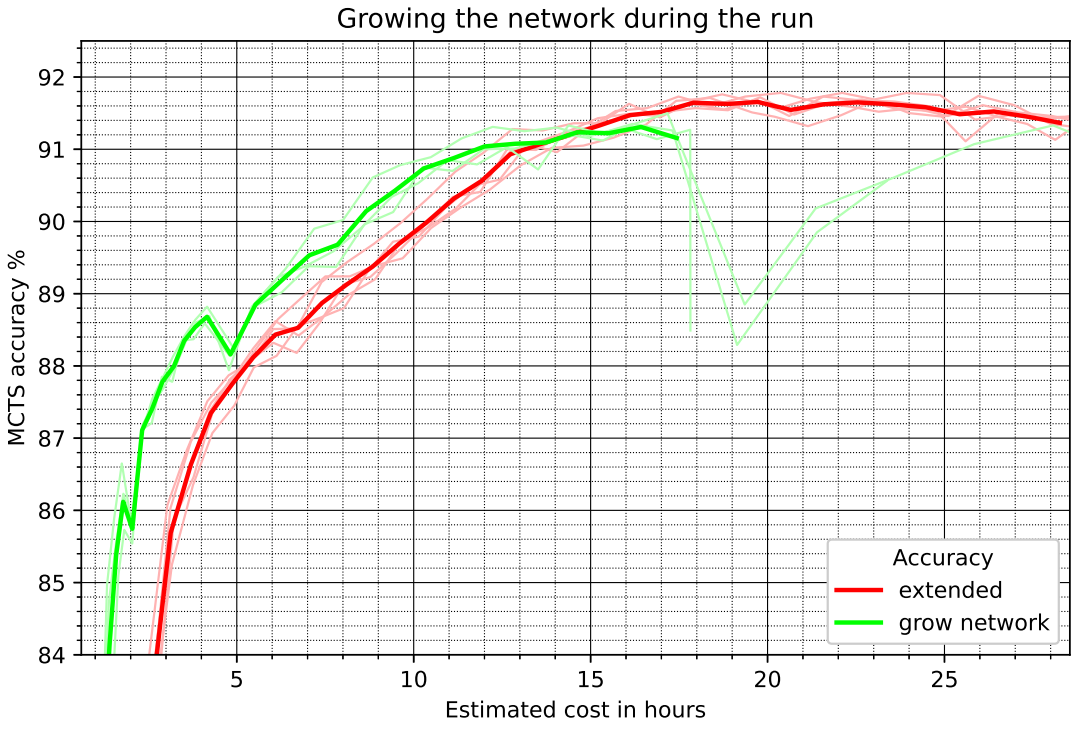

Figure 23: Growing the network as the run progresses increases efficiency, but the steps up to a bigger network cause a temporary drop in accuracy. This is most visible at the end.

Growing the network in this manner has a useful side effect for the concept of auxiliary features: The smaller networks created in the earlier iterations can be used to provide auxiliary features for the later networks, at no additional training costs. This leaves the issue of preventing the auxiliary features from holding back the bigger network,

as has been shown to happen in the experiments with auxiliary features. A possible explanation for the drop in accuracy in later parts of the training might be the layer which is regularized and directly used as in input to the output layer of the network, which might be too disruptive. To investigate, two more options were considered — an extra output convolutional head for the sole purpose of regularization and the regularization of the output of the last residual block of the network. Results of these runs are shown in Figure 24. None of the options investigated seem to be able to tackle the issue of accuracy losses later in the training; they all stay below the baseline. It follows that the use of auxiliary features from a smaller network like this does not appear to help, as the big regularized network is kept back once it starts to substantially outperform the smaller network.

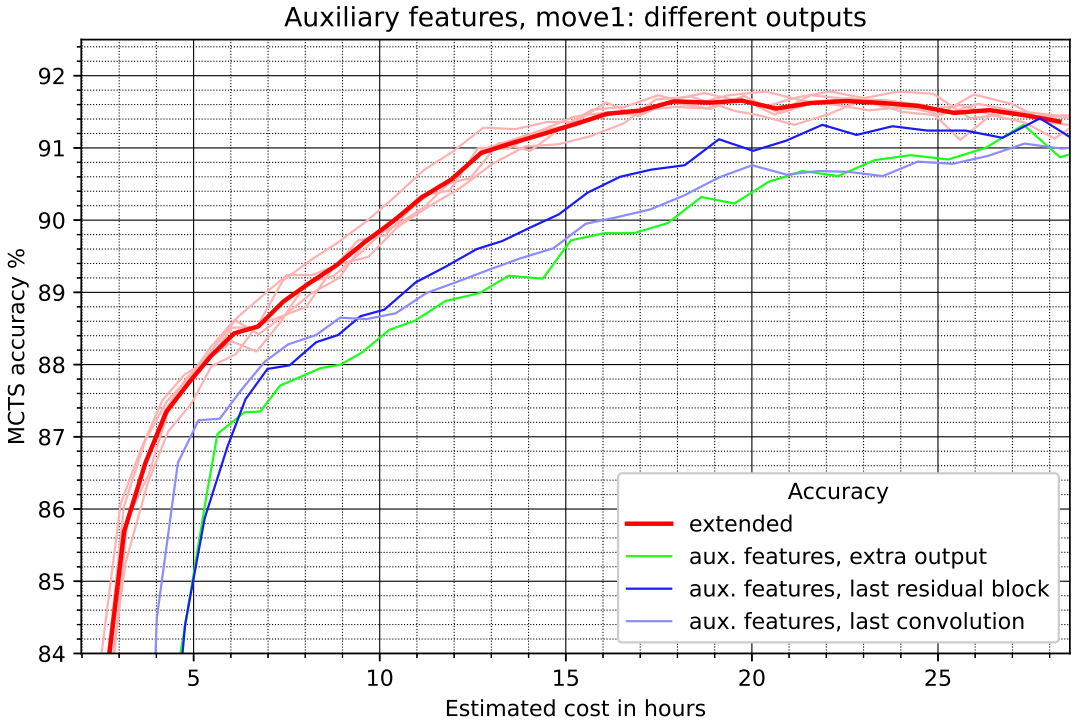

Figure 24: Various options for where to apply regularization.

5 Conclusion

In this thesis a framework has been developed for experimentation on the AlphaZero algorithm, with following experiments on previously known improvements and various novel ideas. While no substantial new improvements were found, the experiments provided multiple interesting results.

It has been shown that different baseline datasets that might look similar on the surface can produce very different results. This makes comparison of different works on AlphaZero potentially harder to compare.

Various improvements proposed by previous works, such as deduplication, cyclic learning rates, improved training windows, predicting opponent replies and an improvement to the network structure have been shown to improve the efficiency of AlphaZero on Connect 4. Another method, "Playout Caps", has been shown to not help on a simple game like Connect 4, as it was developed for Go, which poses different challenges that need to be handled differently. This also highlights that, even though AlphaZero is a very general method, different games still might profit in different ways from different small changes to the algorithm. This is also highlighted by the strong improvements found from the hyperparameter optimization for Connect 4, compared to using parameters from works using other games.

While the original plan was investigate improvements to the network that go beyond the usage of squeeze-and-excite networks, this was not implemented, as the improvements of using squeeze and excite was already minimal. The intended network improvements only produce slightly better results than squeeze-and-excite networks in image classification and thus it seemed unlikely to yield notable improvements in AlphaZero.

The usage of the self-playing phase as an evolutionary process has been shown to be effective in optimizing its fitness function: When optimizing for wins the resulting parameters outplay all other parameter combinations, and when optimizing for novelty a small increase in game novelty was measured. This alone however has proven not to be enough to improve the efficiency of AlphaZero. This in itself is an interesting result, as it shows there is an important difference between just winning games against another parameter set and playing games in such a way that training can process further. The possibility of using the self-playing phase as an evolutionary process remains interesting, and more research might reveal a better fitness function, which corresponds better with learning efficiency.

The investigation of methods to explore possible game moves in tree-like structures has primarily shown that the random sampling used in standard AlphaZero is an extremely efficient way of exploration. None of the investigated novel methods produced nearly as many novel game positions as the simple call to *random()*. On a more positive note

the work on these novel concepts also highlighted a more efficient way of implementing AlphaZero by caching MCTS results, which prevents all duplicate evaluations and increases efficiency.

Using auxiliary features provided by smaller networks has been shown to slightly improve supervised results on Connect 4, but these results do not translate into real AlphaZero runs. Instead, two issues have been identified in the usage of auxiliary features — the cost of the required smaller network and the loss of accuracy in later parts of the training. While growing the network step by step has been shown to be a good way of not only improving learning efficiency in general but also providing the required small network, no good way to use the auxiliary features without causing issues in the later parts of training has been found.

Two primary research directions for future work could be identified — growing the network in a more systematic manner and a more useful fitness function of evolutionary self-play.

References

[1] https://tromp.github.io/c4/c4.html. Accessed: 2020-05-28.

[2] http://www.straitstimes.com/asia/east-asia/googles-alphago-gets-divine-go-ranking. Accessed: 2020-05-17.

[3] https://blog.lczero.org/search?updated-max=2019-01-08T10:35:00%2B01:00&max-results=11. Accessed: 2019-11-29.

[4] https://github.com/LeelaChessZero/lc0/pull/700. Accessed: 2019-11-29.

[5] https://github.com/LeelaChessZero/lc0/pull/721. Accessed: 2019-11-29.

[6] https://github.com/LeelaChessZero/lc0/pull/635. Accessed: 2019-11-29.

[7] https://connect4.gamesolver.org/. Accessed: 2020-05-28.

[8] https://github.com/PascalPons/connect4. Accessed: 2020-05-28.

[9] Crazystone go engine. https://senseis.xmp.net/?CrazyStone. Accessed: 2020-08-05.

[10] Lczero networks. https://training.lczero.org/networks/1. Accessed: 2020-08-05.

[11] Nimrod booklet. http://www.goodeveca.net/nimrod/NIMROD_Guide.html. Accessed: 2020-08-05.

[12] Pachi go engine. https://github.com/pasky/pachi. Accessed: 2020-08-05.

[13] Louis Victor Allis et al. *Searching for solutions in games and artificial intelligence*. Ponsen & Looijen Wageningen, 1994.

[14] Anonymous. Three-head neural network architecture for alphazero learning. In *Submitted to International Conference on Learning Representations*, 2020. under review.

[15] Thomas Anthony, Zheng Tian, and David Barber. Thinking fast and slow with deep learning and tree search. In *Advances in Neural Information Processing Systems*, pages 5360–5370, 2017.

[16] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2-3):235–256, 2002.

[17] Bernd Brügmann. Monte carlo go. 1993.

[18] Murray Campbell, A Joseph Hoane Jr, and Feng-hsiung Hsu. Deep blue. *Artificial intelligence*, 134(1-2):57–83, 2002.

[19] Guillaume MJ-B Chaslot, Mark HM Winands, and H Jaap van Den Herik. Parallel monte-carlo tree search. In *International Conference on Computers and Games*, pages 60–71. Springer, 2008.

[20] Sylvain Gelly, Yizao Wang, Olivier Teytaud, Modification Uct Patterns, and Projet Tao. Modification of uct with patterns in monte-carlo go. 2006.

[21] Jie Hu, Li Shen, and Gang Sun. Squeeze-and-excitation networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7132–7141, 2018.

[22] Ethan C Jackson and Mark Daley. Novelty search for deep reinforcement learning policy network weights by action sequence edit metric distance. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, pages 173–174, 2019.

[23] Daniel Kahneman. Thinking, fast and slow. new york. 2011.

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

[25] Vladimir Kramnik. Kramnik and alphazero: How to rethink chess. https://www.chess.com/article/view/no-castling-chess-kramnik-alphazero. Accessed: 2019-11-29.

[26] Li-Cheng Lan, Wei Li, Ting-Han Wei, I Wu, et al. Multiple policy value monte carlo tree search. *arXiv preprint arXiv:1905.13521*, 2019.

[27] Joel Lehman and Kenneth O Stanley. Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary computation*, 19(2):189–223, 2011.

[28] Aditya Prasad. Lessons from implementing alphazero. https://medium.com/oracledevs/lessons-from-implementing-alphazero-7e36e9054191. Accessed: 2019-11-29.

[29] Raymond Redheffer. A machine for playing the game nim. *The American Mathematical Monthly*, 55(6):343–349, 1948.

[30] Claude E Shannon. Xxii. programming a computer for playing chess. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 41(314):256–275, 1950.

[31] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484, 2016.

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

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

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

[35] Leslie N Smith. Cyclical learning rates for training neural networks. In *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 464–472. IEEE, 2017.

[36] David J Wu. Accelerating self-play learning in go. *arXiv preprint arXiv:1902.10565*, 2019.

[37] Xin Yao. Evolving artificial neural networks. *Proceedings of the IEEE*, 87(9):1423–1447, 1999.

[38] Anthony Young. Lessons from implementing alphazero, part 6. `https://medium.com/oracledevs/lessons-from-alpha-zero-part-6-hyperparameter-tuning-b1cfcbe4ca9a`. Accessed: 2019-11-29.