Improvements to Increase the Efficiency of the AlphaZero Algorithm: A Case Study in the Game 'Connect 4'

Colin Clausen[2], Simon Reichhuber[1], Ingo Thomsen[1] and Sven Tomforde[1]
[1]*Intelligent Systems, Christian-Albrechts-Universität zu Kiel, 24118 Kiel, Germany*
[2]*PCT Digital GmbH, Fleethörn 7, 24103 Kiel, Germany*

Keywords: AlphaZero, Connect 4, Evolutionary Process, Tree Search, Network Features.

Abstract: AlphaZero is a recent approach to self-teaching gameplay without the need for human expertise. It suffers from the massive computation and hardware requirements, which are responsible for the reduced applicability of the approach. This paper focuses on possible improvements with the goal to reduce the required computation resources. We propose and investigate three modifications: We model the self-learning phase as an evolutionary process, study the game process as a tree and use network-internal features as auxiliary targets. Then behaviour and performance of these modifications are evaluated in the game Connect 4 as a test scenario.

1 INTRODUCTION

Computer games have ever been a fantastic playground for research on artificial intelligence (AI). With the increasing availability of computing resources and platforms of the last decade, groundbreaking developments in human-like and even superhuman game playing behaviour have been witnessed. Especially the presentation of AlphaGo (Silver, Huang et al., 2016) that defeated a world-class human go player in 2016 (Gibney, 2016) is a milestone towards fully sophisticated games AI.

Subsequently to the breakthrough caused by AlphaGo, the AlphaZero algorithm (Silver, Hubert et al., 2018) significantly simplified the initial AlphaGo approach. The basic idea was to allow for a learning concept that starts with a random network and obviates human expert input of any kind. Despite the success of these concepts, we face a massive challenge that inhibits ubiquitous utilisation of the approach: The dramatic requirements of computation power.

The goal of this paper is to investigate possibilities for improvements of AlphaZero's efficiency, expecting a reduction of hardware and computational cost, while simultaneously maintaining the generic concept and the success of AlphaZero. We therefore propose and investigate three different extensions, which result in a changed behaviour of the basic AlphaZero algorithm: We model the self-learning phase as an evolutionary process, study the game process as a tree and use network-internal features as auxiliary targets.

The remainder of this article is organised as follows: Section 2 describes the background of this paper by briefly summarising the state-of-the-art. Section 3 develops a baseline for AlphaZero for the game Connect 4. Afterwards, Section 4 provides the key contributions of this paper by proposing three different refinements and extensions to the current AlphaZero approach and analyses the behaviour. Finally, Section 5 concludes the article.

2 BACKGROUND

2.1 Monte Carlo Tree Search

With Monte Carlo Tree Search (MCTS) (Brügmann, 1993) a large (game) tree is randomly searched to assess the expected reward for moves starting at the tree root. The variant UTC ('UCB applied to trees') (Kocsis and Szepesvári, 2006) is used in AlphaZero: the Upper Confidence Bounds (UCB) algorithm of the *multi-armed bandit problem* (Auer et al., 2002) is used to choose the next move in tree node: Possible episodes (root to the end of game) are sampled and the results are propagated back. When playing only to a fixed tree depth, the positions can be evaluated by averaging over results from a number of random playthroughs to the end. Weighting exploitation against exploration is applied for episode sampling, which incrementally improves overall move evaluation.

Clausen, C., Reichhuber, S., Thomsen, I. and Tomforde, S.
Improvements to Increase the Efficiency of the AlphaZero Algorithm: A Case Study in the Game 'Connect 4'.
DOI: 10.5220/0010245908030811
In *Proceedings of the 13th International Conference on Agents and Artificial Intelligence (ICAART 2021) - Volume 2*, pages 803-811
ISBN: 978-989-758-484-8

2.2 From AlphaGo to AlphaZero

AlphaGo used a complex system initialisation with example games, random rollouts during tree searches and two networks (prediction of position values and of promising moves). AlphaGo Zero (Silver, Schrittwieser et al., 2017) is much simpler by using only single network without rollouts: The network evaluation for a position is directly used. AlphaZero is applied to games other than Go and offers further simplification: Instead of comparing the currently trained network with the previously known best player using evaluation games, AlphaZero always bases new games on the current network. The main advantage of the 'Zero' versions is the independence from prior human knowledge (apart from rules). This promotes strategies without human bias and even enables research on games without human expertise.

2.3 The AlphaZero Algorithm

AlphaZero combines the UTC variant with a deep neural network to bias tree search and evaluate unfinished games. The network is trained to predict final results of the MCTS and self-play games by MCTS against itself. This tends to be slower at first, but the network is trained to make "good decisions". This is subsequently faster due to using just a single forward pass. The network creates two outputs from the encoded game situation: a policy describing move likelihoods and the expected value of the situation. Unlike plain UTC, AlphaZero does not randomly select episodes, which besides are not played out until a terminal game state is reached. To steer the MCTS towards moves estimated to be promising by the network, the following statistics are gathered for each node:

$N(s,a)$ times action a was chosen in state s

$W(s,a)$ total action value

$Q(s,a)$ average action value, equal to $\frac{W(s,a)}{N(s,a)}$

$P(s,a)$ probability to play action a in state s

When the analysis of an episode is propagated upwards the tree, these statistics are updated. They are used in the next tree search until it reaches again an unknown node. When a fixed number of nodes are created, the output is a distribution of node visits: A high count implies a worthwhile and thoroughly evaluated move. The action a is selected as follows:

$$argmax_a(Q(s,a)+U(s,a)) \tag{1}$$

$$U(s,a) = C_{puct}P(s,a)\frac{\sqrt{N(s)}}{(1+N(s,a))} \tag{2}$$

Equation 2 is weighted against exploitation and is based upon plain UTC, but biased by the network policy. C_{puct} is a game-dependent constant, typically in $[0.5,5]$. Dirichlet noise is added for further exploration in self-play games, pushing the MCTS cautiously to evaluate some random moves more than others. Games are played out using this tree search: The policy, resulting game states and outcome are stored to train the network to predict the final result and policy. A policy, based upon network *and* MCTS, is expected to outperform one produced by only the network.

A drawback is the massive computational cost. In practice for playing a single move, the search tree has to grow to at least hundreds of nodes, each with a correlating forward pass through the network.

2.4 Extensions to AlphaZero

Many proposed enhancements focus on reducing the computational cost and several of those listed below originate from the Leela Chess Zero project (LCZero, 2020), a distributed effort to implement AlphaZero for Chess. Some approaches do not rely on domain-specific game knowledge, and those denoted by * were implemented to form a strong AlphaZero baseline for comparison (see Section 3.3 and 3.4).

Network and Training Modifications. Improving the neural network design to speed up training, while retaining the original tower of 20 or 40 residual network blocks with 256 convolutional filters:

1. * The network blocks themselves can be enhanced with Squeeze-and-Excitation elements (Hu et al., 2018). A similar approach is used by (Wu, 2020).
2. * Cyclic learning rates as in (Smith, 2017) can be used to improve the network fitting to the data.
3. Increasing the number of convolutional filters in the policy head and values from 3 to 32, speeding up the training (Young, 2019).

Modification of the Tree Search. Different spending of the available time for analysing game positions.

1. With 'Playout Caps' (Wu, 2020) the number of MCTS playouts is randomly and drastically reduced for most moves. An acceleration of the training by 27% is stated.
2. The handling of the search tree can be improved. One example is the propagation of terminal moves to simplify the search (Videodr0me, 2019), moderately improving the playing strength.

3. (Lan et al., 2019) use two networks instead of one. Unequal sizes (10 resp. 5 residual blocks with 128 resp. 64 convolutional filters), allow the smaller, faster network to do more search steps in most positions. This at least doubles the training speed.
4. With constant computational effort, the playing strength can be increased (Tilps, 2019a) by employing the Kullback-Leibler divergence of the policy to identify and focus on complex positions.

Learning Target Modifications. Reducing overfitting and regulating the network training.

1. * (Gao et al., 2020) add a third output head to predict the win probability for every move. This estimate might be of worse quality. Small improvements are claimed.
2. * (Wu, 2020) predict the opponent's reply to regularise training – showing modest improvement.
3. * They also argue to add some domain-specific targets to regularise the learning process, which shows major improvements.
4. Finally, they guarantee selected nodes a minimum number of further "Forced Playouts" to promote exploration. "Policy Target Pruning" then removes those from the final policy distribution. The training speed is shown to increase by 20%.
5. The LCZero project modifies the value target (Tilps, 2019b) to explicitly predict drawing probabilities: The network can discern between very likely and unlikely positions.
6. (Young, 2019) proposes to combine the game results as a learning target with the average value of the MCTS node, reducing the influence of few bad moves at the end of the game.

Training Data Enhancements. Improved handling of the numerous training examples by (Young, 2019).

1. * In some games, like Connect 4, positions can be duplicated. Averaging the respective targets and combining them into a single example is useful.
2. * A sudden increase of playing strength occurs when removing the first training examples, which were very likely generated by a random network. Modifying the training windows fixes this.

3 A BASELINE: AlphaZero FOR CONNECT 4

Connect 4 is a board game where two players, in turn, let their discs fall straight down to the lowest available space of a column in a vertical grid, typically of size 6×7. The goal is to create a horizontal, vertical or diagonal line of four of one's own discs. It is a game with known a solution for which strong solvers exists (Prasad, 2019). Therefore, we can evaluate the strength of AlphaZero by comparing the accuracy on a set of solved test positions. The network can also be trained on solved games for maximum performance.

The state space was a 6×7 matrix with 4 possible cell values: 3 (unoccupied), (player) 2 or 1. Also, 0 is used for padding during convolution. The base implementation of AlphaZero uses a different network target for game results, encoding a dedicated value for a draw. Therefore, the 3 possible outcomes are a win of either player or a draw. This differs from the output between zero and one of the original implementation. There might be a slight change in learning efficiency caused by this change. However, due to the technical requirements in the context of the abstraction-driven framework (to keep the generic capability and not refocus it to Connect 4), we changed the implementation to this additional state.

We started with a basic version of AlphaZero and adapted it for Connect 4. With a given training set of played games for a certain state s representing the state of the game, a neural network directly learns the move probabilities $m^{pred} \in [0,1]^{\#moves}$. This states how likely one of the seven possible moves leads to a win. Furthermore, it learns the outcome of the game's test set o^{test}. Since the outcome also includes draws, we extended the test set outcome to a tuple $o^{test} = (o^{test}_{win}, o^{test}_{draw}, o^{test}_{loose})$ and the predicted outcome analogously to $o^{pred} = (o^{pred}_{win}, o^{pred}_{draw}, o^{pred}_{loose})$. The vectors of the optimal move were "one-hot-encoded": Only the optimal move and the true outcome are set to 1. With these the the network $f_\Theta(s) = (m^{pred}, o^{pred})$ aims to guarantee $(m^{pred}, o^{pred}) \approx (m^{test}, o^{test})$ w.r.t. to an output weighting and regularised loss function as in (Silver, Schrittwieser et al., 2017):

$$\begin{aligned} &\mathcal{L}\left((o^{pred}, m^{pred}), (o^{test}, m^{test})\right) = \\ &- c_{output} * \sum_{i=1}^{3} o^{test}[i] * \log\left(o^{pred}[i]\right) \\ &- \sum_{j=1}^{7} m^{test}[j] * \log\left(m^{pred}[j]\right) + c_{reg} * \|\Theta\|^2 \end{aligned} \tag{3}$$

The weights were $c_{output} = 0.01$ and $c_{reg} = 0.0001$

3.1 Experimental Setup

The main goal of this paper 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 create training data for the neural network. This fact was used to simplify the training cost measurement by ignoring the costs of network training. Instead, only the cost of self-playing workers was measured. In all experiments, neural network training were done on a single GPU and newly produced networks were evaluated on another single GPU. Self-play workers are run on a P2P cloud service2 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 workers. For each experiment a benchmark was run on a reference machine using the produced networks, measuring how much time is needed on average to play a single move. This value was then used to estimate the cost of the actual number of moves played by the self-play workers during the experiments. The reference machine uses a single Nvidia RTX 2070S, which was saturated (100% load) by the benchmark. Thus, all self-play cost of experiments is stated as estimated time spent on that machine, limited by its GPU capacity.

3.2 Supervised Training

The generation of the testing set for the learning process is an important step that determines how comparable results are to previous work. An important decision is what should be considered a correct move in a given situation. In Connect 4, a player has many positions where she has multiple ways to force a win, but some lead to a longer game before the win is forced. To evaluate this (Young, 2019) defines two test sets:

- The *strong* test set only considers a move correct if it yields the fastest win resp. the slowest loss.
- The *weak* test set 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.

For training a dataset of 1 million game positions was generated by playing 50% random moves and 50% perfect moves determined by a solver (Pons, 2020). This is substantially harder than a dataset created using 90% perfect and only 10% random moves, which appears to be mainly a result of the distribution of game length as fewer random moves produce a dataset with more positions late in the game, while more random moves cause more positions to be in the early game. No duplicates, without trivial positions and only the strongest possible moves were accepted as correct. Two versions of the dataset were generated. In version 1, all positions of played games are used for training, while in version 2, only one single position is randomly picked from each played game. This substantially increased the number of distinct games played. For both dataset versions, 800 000 examples were used as training data and 100 000 examples as validation data. The remaining 100 000 examples were utilised as test data. We achieved the highest accuracy of correctly assigned outcomes by using a network with 20 ResNet blocks (see Table 1).

Table 1: Results of supervised training and accuracy compared to a Connect 4 solver. The column named N lists the number of used ResNet blocks, $Pr^{v_1,v_2}_{\text{move, win}}$ are the move and win probabilities of training set version 1 v_1 resp. 2 v_2.

N	# params	$Pr^{v_1}_{\text{move}}$	$Pr^{v_1}_{\text{win}}$	$Pr^{v_2}_{\text{move}}$	$Pr^{v_2}_{\text{win}}$
5	1.57e6	91.63%	77.47%	92.44%	79.23%
10	3.05e6	92.37%	77.87%	93.00%	79.67%
20	6.01e6	92.68%	78.23%	93.49%	79.93%

3.3 Extended AlphaZero

We analysed known AlphaZero extensions (see Section 2.4) in addition to Connect 4 specific modifications from (Prasad, 2019). In detail, successful extensions are presented in the remainder of this section.

Remove Duplicate Positions. Merging duplicate positions is proposed by (Prasad, 2019). We implemented this by using 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. A $z_{\text{duplicate}}$ produced by a self-play worker is sent to the training worker, which updates the new target values z_{new} with the duplicate's old target values z_{old}, weighted by $w_{\text{duplicate}}$:

$$z_{\text{new}} = z_{\text{old}} * (1 - w_{\text{duplicate}}) + z_{\text{duplicate}} * w_{\text{duplicate}} \quad (4)$$

Figure 1 shows tests for some $w_{\text{duplicate}} = 0.2, 0.5, 0.8$:

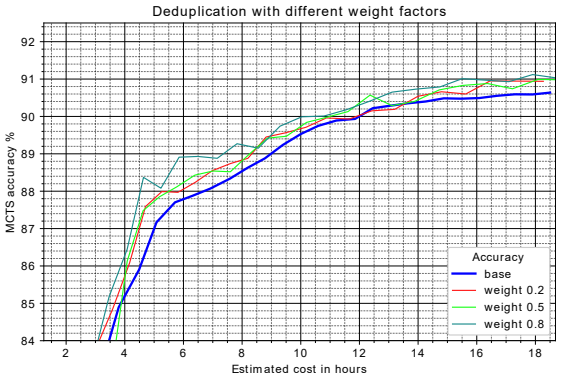

Figure 1: Comparison of different choices for $w_{\text{duplicate}}$. For all further experiments 0.8 was chosen.

Cyclic Learning Rate. To speed up general neural network training, (Smith, 2017) proposed cyclical learning rates. (Prasad, 2019) suggests some improvements in the context AlphaZero. Based on these claims, cyclic learning rates and cyclic momentum were used for AlphaZero as a potential addition to the extended baseline. The cycles are spread over a single training iteration. The values are updated by previous runs to determine the maximum and minimum useful learning rates. Their curve can be seen in Figure 2.

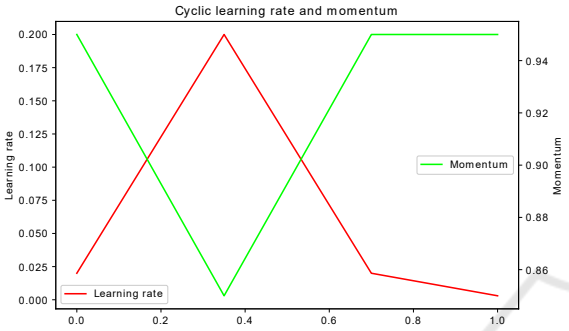

Figure 2: Change of learning rate and momentum over the training of a new network when using cyclic learning rates.

Additionally, the learning rate is annealed down with a multiplicative factor that drops linearly with the number of network iterations. It starts at 1 in iteration 1 and reaches 0.4 in iteration 20. Beyond that, the factor stays at 0.4: At iteration 20 and later, the maximum learning rate is 0.08. This annealing accommodates the fact that in later parts of training the learning rate should be smaller in general. This improvement accelerated the learning as seen in Figure 3.

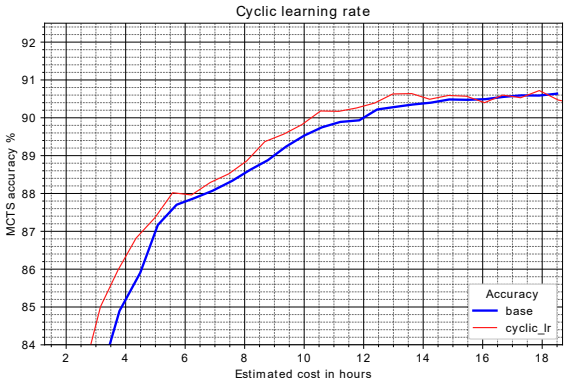

Figure 3: Comparison of the usage of cyclic learning rates and momentum with the baseline.

Improved Training Window. Similar to (Prasad, 2019) a slow training window starts with a size of 500000 examples, growing between iteration 5 and 12 up to the maximum size of 2 million examples. This causes early examples to be removed by iteration 3. This is useful as those are very close to random play. The slow window was added to the extended baseline, since it seemed to have no harmful effect (see Figure 4) and follows a sound idea. As the extended baseline uses a training window without duplicate positions, the parameters were modified to grow from iteration 5 to 15, since the number of new examples per iteration is lower.

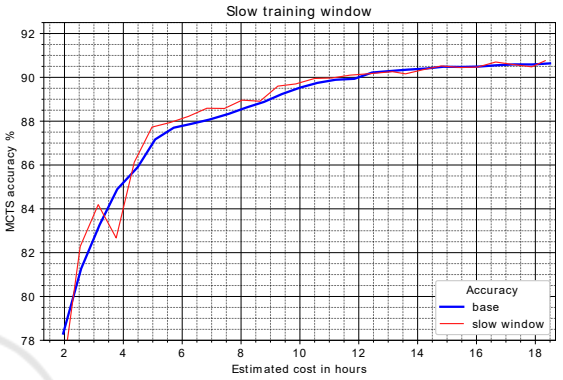

Figure 4: Comparison of the usage of a slow training window with the baseline.

Predicting the Reply of the Opponent. (Wu, 2020) also, show that predicting an opponent's reply to a position produces a *modest but clear benefit*, by adding a term to the loss function to regularise training:

$$-w_{opp} \sum_{m \in moves} \pi_{opp}(m) \log\left(\hat{\pi}_{opp}(m)\right) \qquad (5)$$

π_{opp} is the policy target for the turn after the current turn, $\hat{\pi}_{opp}$ is the network prediction of π_{opp} and w_{opp} is a weight for the loss term. Based on preliminary experiments, 0.35 was used. Figure 5 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.

Improved Network Structure. The Leela Chess Zero project proposes using squeeze-and-excitation elements in the network. This is a general development of deep neural networks (Hu et al., 2018) and shall improve the accuracy of correctly predicted game outcomes compared to the baseline. A result of a test run can be seen in Figure 6. There might be some gains early during the training and possibly some small losses later on.

Since this did not seem to reduce performance much – and intuitively should help, especially given more distinct training data from de-duplication – the

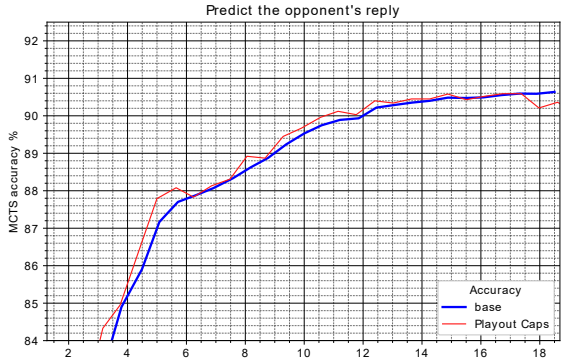

Figure 5: Results of implementing the predictions of the opponent's reply.

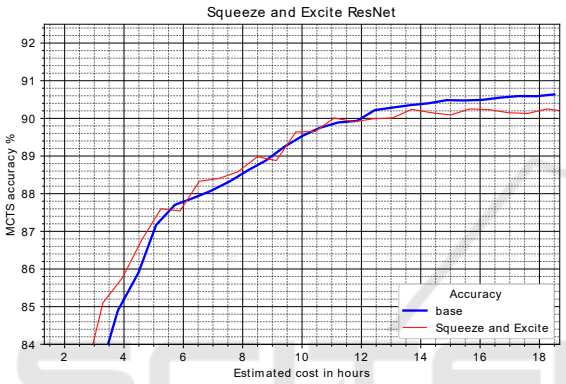

Figure 6: Results of predictions of squeeze-and-excitation.

squeeze-and-excite elements were added to the extended baseline. The network structures can be seen in Table 2.

3.4 Establishing the Baselines

Various baselines needed to be defined and implemented for the comparison. Using supervised training for the networks allowed for the exploration of the maximum possible performance: A plain AlphaZero baseline showed how the original algorithm performed and a baseline of AlphaZero, extended with a set of previously known improvements, shows the progress compared to the original algorithm.

As our goal is to find AlphaZero extensions lowering the training costs of such an algorithm, we provided a baseline measurement on how much time is needed on average to play a single move. This value was then used to estimate the cost of the actual number of moves played by the self-play workers during the experiments (see Figure 7).

Table 2: The final architecture, including the modifications of (Silver, Hubert et al., 2018). Squeeze-and-excite add pooling to the residual blocks, which averages every feature map to a single scalar value. These 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 denotes a fully connected layer with x neurons. Addition describes the addition with the input of the residual block forming the residual structure of the block. Both the win prediction and the move policy output are connected to the output of the last residual block.

Description	base	extended
Initial block	$\begin{pmatrix} 3\times3\times64 \\ \text{BatchNorm} \\ \text{ReLU} \end{pmatrix}$	—'—
Adaptor convolution	$(1\times1\times128)$	—'—
Residual block, repeated N times	$\begin{pmatrix} 3\times3\times128 \\ \text{BatchNorm} \\ \text{ReLU} \\ 3\times3\times128 \\ \text{BatchNorm} \\ \text{Addition} \\ \text{ReLU} \end{pmatrix}$	$\begin{pmatrix} 3\times3\times128 \\ \text{BatchNorm} \\ \text{ReLU} \\ 3\times3\times128 \\ \text{BatchNorm} \\ \text{AVGPooling} \\ \text{FCnb : 8} \\ \text{ReLU} \\ \text{FCnb : 128} \\ \text{Sgmoid} \\ \text{Addition} \\ \text{ReLU} \end{pmatrix}$
Move policy output	$\begin{pmatrix} 3\times3\times32 \\ \text{FC : 7} \\ \text{LogSoftMax} \end{pmatrix}$	—"—
Win prediction output	$\begin{pmatrix} 3\times3\times32 \\ \text{FC : 3} \\ \text{LogSoftMax} \end{pmatrix}$	—"—

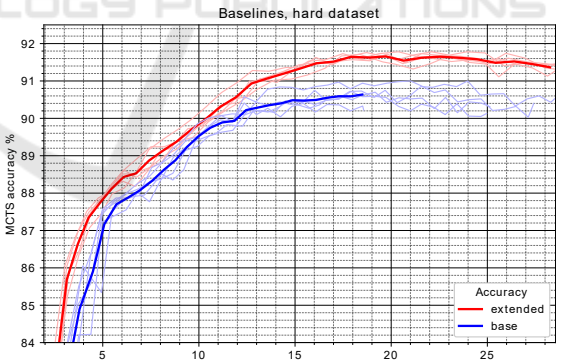

Figure 7: Comparison of the baseline and the extended baseline. *Mean* was only calculated until either run stops showing improvements.

4 THREE EXTENSIONS FOR AlphaZero

4.1 Playing Games as Trees

The first extension focuses on how AlphaZero plays games and implements a distributed approach: In-

stead of many machines playing independent games, only *one* plays all the games and requests from a service an MCTS evaluation, which is distributed across multiple machines instead. This service can easily filter out duplicate evaluations, increasing the efficiency compared to the extended baseline (see Figure 8).

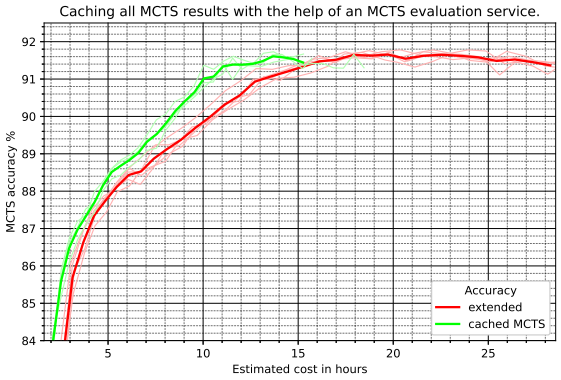

Figure 8: The MCTS evaluation service is more efficient compared to the AlphaZero baseline.

This filtering allows for further tuning: For a lost game the position can be reset, tweaking the search for alternatives: In case of a bad path (due to losing), we want to identify more promising strategies from here. However, the experiments showed that this did not result in significant improvements.

With a focus on the tree size, we grew one large MCTS tree and used its nodes as learning positions – again to explore more alternative paths for better coverage. However, the results showed no significant improvement: MCTS seems to gets stuck on certain typical games and stops exploring. It nevertheless explores quite well during the early training phase (see Figure 9). More precisely, using MCTS in this way causes a quick spike in game diversity. It then quickly drops as it settles into a certain style of play, from which it never shifts away again. Seemingly, a short boost is possible but long-term success is not visible. Section 5 will outline how to combine these two steps in future work.

4.2 Network-internal Features as Auxiliary Targets

The second approach uses network-internal features as auxiliary targets. Generally, domain-specific features can be beneficial but are not easy to obtain without human design. So, we tried to automate this and applied deep neural networks, which are known to learn good features in other tasks (e.g. image recognition). We performed a full AlphaZero training run

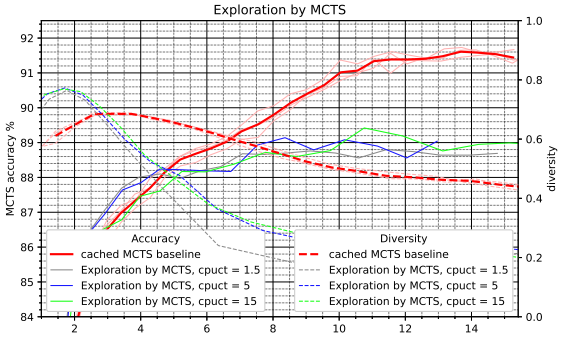

Figure 9: Replacing self-play with one large MCTS which tends to get stuck on a few game paths and stops exploring. This reduces the diversity of encountered game positions and even massively increasing C_{puct} does not prevent this.

with a tiny network on a single machine and used its internal features to regulate the training targets. For additional benefit, we analysed them using two features from the future: The playout of full games and applying the small network to all the states in the game, with which the features for each position can be found. A larger network can be trained for a limited prediction of the smaller network's feature output. The small network uses only a single filter in the last convolutional layer, that has two distinct output-heads (of the 6×7 network fields): Policy and game output prediction with 42 features each. These features, that correspond to the future (single player) moves 2 and 4, are then to regulate the bigger network: A squared error is added to the training loss for calculating the mean squared error between auxiliary feature and an internal layer of the big network, to which no additional parameters are added – only the internal layer from which the features in the small network were taken is regularised to get the same features in the bigger network. For evaluation, we performed a set of supervised runs and identified the most promising candidates – which turned out to provide a very small 0.2% absolute improvement in test accuracy, which outside the typical run-to-run variance. Within the full AlphaZero setting the improvements could not be generalised: The extended baseline mostly remained better than the novel variant as shown in Figure 10.

Since the novel approach was surprisingly ineffective we examined more closely: The 2 hours training time for the small network, to obtain the features, was rather expensive and made a huge difference. As this can explain the effect, we tried to fix it by growing the network during the run (see Figure 11). However, the effect remains. Additionally, we can observe that using the small network to regularise the large one causes consistent issues later in the training, reducing the final accuracy. This seems to be the root of the limited applicability of the approach. Again, future

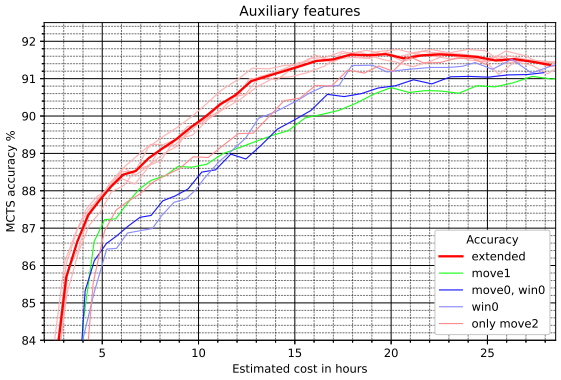

Figure 10: Using auxiliary features from a smaller network.

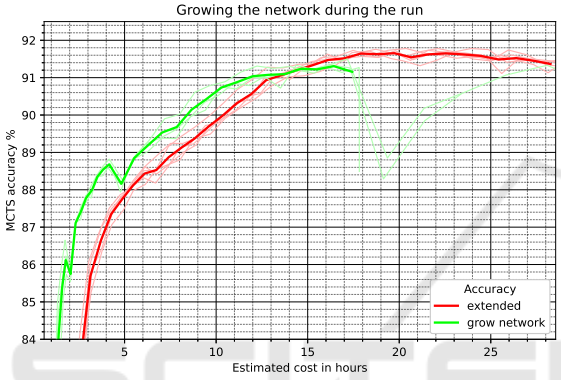

Figure 11: Growing the network as the run progresses increases efficiency, but the steps up to a bigger network cause a temporary drop in accuracy, especially towards the end.

work will focus on ways to tackle these inconsistencies to harvest the potential benefits.

4.3 Self-playing Phase as Evolutionary Process

The third extension focuses on the self-playing phase of AlphaZero: Using played games to evaluate different hyperparameter sets on the fly. This re-utilisation comes without any extra computational costs. Therefore, we modelled the hyperparameter sets as players, which play in a league with *Elo* rating: The most promising candidates ('best players') are evolved using Gaussian mutation. We use the MCTS-related hyperparameters as a basis for the investigation as they can be changed easily for every game played.

In future work, we will further investigate which other hyperparameters can be handled and show promising results. Here, we investigated how to control the reasoning period of the algorithm using a threshold on the Kullback-Leibler divergence on the development of the policy found by MCTS with the initial results outlined in Figure 12, although no significant improvements were found.

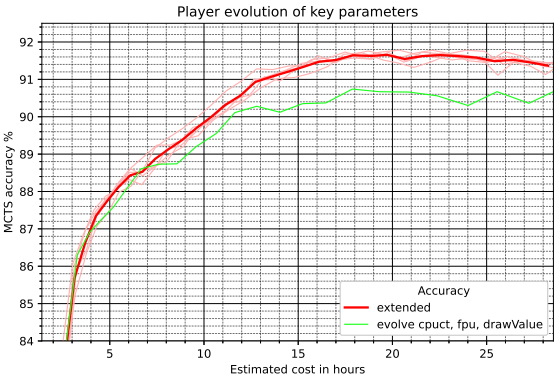

Figure 12: Initial results of evolving hyperparameters.

Consequently, we analysed if the evolution itself works. To verify the implementation of the evolution, we added an 'artificial' additional hyperparameter with a known value. This artificial parameter was perfectly optimised, meaning that the evolution itself works. Issues occurred with optimising towards winning games since this does not correlate with the metric used to determine the training progress and the accuracy against the Connect 4 solver. Therefore, we compared different hyperparameter sets found in 1000 matches: The hyperparameters found by the evolution won more games, but in turn, played worse according to the Connect 4 solver.

Motivated by this observation, we investigated using novelty search as an evolution fitness function. This novelty-driven process identifies marginally more new game-play than the parameters found by the original baseline, which was determined with Bayesian hyperparameter optimisation. From these outcomes, it can be stated that the original hyperparameters have already been optimised rather for game novelty. Figure 13 illustrates the achieved results.

5 CONCLUSIONS

This paper motivated the need for efficient and scalable modifications of the state-of-the-art technique AlphaZero with the observation of a limited application due to the massive hardware demands.

We, therefore, reviewed the most prominent algorithm extensions to develop a baseline for further experiments. Based on this, we presented three different concepts for reducing the necessary efforts: We proposed to model the self-learning phase as an evolutionary process, we studied the game process as a

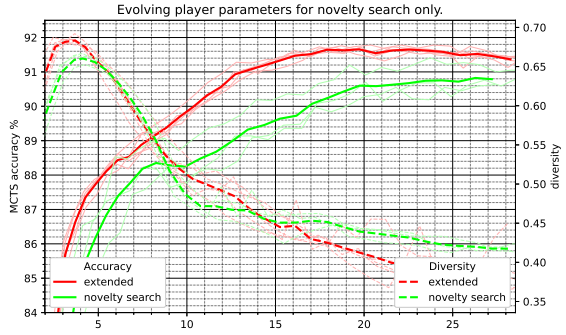

Figure 13: Even pure novelty search only produces more novel games towards the end of the training. The baseline parameters were likely to be already implicitly optimised for game diversity by the initial Bayesian hyperparameter optimisation aiming to learn efficiently.

tree and used network-internal features as auxiliary targets. We showed that, although the effort could be decreased slightly, the benefit is mostly only small.

Our future work follows two directions: We want to further analyse possible improvements of AlphaZero, e.g. based on the Connect 4 scenario, and we want to investigate the applicability to real-world control problems. For the first path, we identified an approach of growing the network more systematically as a possibly beneficial extension. Alternatively, a more sophisticated fitness function for the evolutionary self-playing phase could provide a more suitable trade-off between heterogeneity and convergence. For the second path, we will investigate if such a technique is applicable to real-world control problems (D'Angelo, Gerasimou, Gharemani et al., 2019) as given by self-learning traffic light controller (Sommer et al., 2016) or smart camera networks (Rudolph et al., 2014).

REFERENCES

D'Angelo, Gerasimou, Gharemani et al. (2019). On learning in collective self-adaptive systems: state of practice and a 3D framework. In *Proc. of SEAMS@ICSE 2019*, pages 13–24.

Auer, P., Cesa-Bianchi, N., and Fischer, P. (2002). Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2–3):235–256.

Brügmann, B. (1993). Monte Carlo Go. *www.ideanest.com /vegos/MonteCarloGo.pdf*.

Gao, C., Mueller, M., Hayward, R., Yao, H., and Jui, S. (2020). Three-Head Neural Network Architecture for AlphaZero Learning. https://openreview.net/forum?id=BJxvH1BtDS.

Gibney, E. (2016). Google AI algorithm masters ancient game of Go. *Nature News*, 529(7587):445.

Hu, J., Shen, L., and Sun, G. (2018). Squeeze-and-excitation networks. In *Proc. of IEEE CVPR*, pages 7132–7141.

Kocsis, L. and Szepesvári, C. (2006). Bandit based Monte-Carlo planning. In *European conference on machine learning*, pages 282–293. Springer.

Lan, L.-C., Li, W., Wei, T.-H., Wu, I., et al. (2019). Multiple Policy Value Monte Carlo Tree Search. *arXiv preprint arXiv:1905.13521*.

LCZero (2020). Leela Chess Zero. http://lczero.org/.

Pons, P. (2020). https://connect4.gamesolver.org and https://github.com/PascalPons/connect4. accessed: 2020-10-02.

Prasad, A. (2019). Lessons From Implementing AlphaZero. https://medium.com/oracledevs/lessons-from-implementing-alphazero-7e36e9054191. accessed: 2019-11-29.

Rudolph, S., Edenhofer, S., Tomforde, S., and Hähner, J. (2014). Reinforcement Learning for Coverage Optimization Through PTZ Camera Alignment in Highly Dynamic Environments. In *Proc. of ICDSC'14*, pages 19:1–19:6.

Silver, Huang et al. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489.

Silver, Hubert et al. (2018). A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362(6419):1140–1144.

Silver, Schrittwieser et al. (2017). Mastering the game of Go without human knowledge. *Nature*, 550(7676):354.

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

Sommer, M., Tomforde, S., and Hähner, J. (2016). An Organic Computing Approach to Resilient Traffic Management. In *Autonomic Road Transport Support Systems*, pages 113–130. Springer.

Tilps (2019a). https://github.com/LeelaChessZero/lc0/pull/721. accessed: 2019-11-29.

Tilps (2019b). https://github.com/LeelaChessZero/lc0/pull/635. accessed: 2019-11-29.

Videodr0me (2019). https://github.com/LeelaChessZero/lc0/pull/700. accessed: 2019-11-29.

Wu, D. J. (2020). Accelerating Self-Play Learning in Go.

Young, A. (2019). Lessons From Implementing AlphaZero, Part 6. https://medium.com/oracledevs/lessons-from-alpha-zero-part-6-hyperparameter-tuning-b1cfcbe4ca9a. accessed: 2019-11-29.

APPENDIX

The framework and the experimental platform for distributed job processing are available at: https://github.com/ColaColin/MasterThesis.