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, 1.8.2020

Selbstständigkeitserklärung

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

.....
Colin Clausen

Contents

1 Introduction

Games have been used for a long time as a standin of the more complex real world in developing artifical intelligence. Beating humans at various games has often been viewed as a milestone.

(TODO a few sentences here about progress on various milestone games).

In March 2016 a program called AlphaGo for the first time in history has defeated a top human player in the board game Go [1]. Go had eluded attempts at super human level play for a very long time.

Louis Victor Allis attributes [11] this to the large game tree size of 10^{360} possible games, compared to 10^{120} in chess [23], but also to the way humans use their natural pattern recognition ability to quickly eliminate most of the often 200 or more possible moves and focus on few promising ones.

This combination of the usage of hard-to-program pattern recognition with an extremely large game tree has prevented computers from reaching top human strength through game tree search algorithms based on programmed heuristics. AlphaGo solved this issue by using the strong pattern recognition abilities of deep learning and combining them with a tree search, 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 no requirements for human expert input of any kind.

In the following thesis I want to investigate further possible improvements to reduce the computational cost of using AlphaZero to learn to play games without using the generality of the algorithm.

(TODO here one could state some key results, once they exist...)

This thesis is structured as follows:

First I will look at previous work, starting with the basis of AlphaZero, Monte Carlo Tree Search, moving onto the various versions of AlphaZero with previously suggested improvements. Then a list of novel improvements will be described. Finally an extensive set of experiments will be presented, first establishing a baseline performance, then showing the results on the novel improvements.

2 Previous work

In this section previous work relevant to AlphaZero will be presented.

2.1 Monte Carlo Tree Search

Monte Carlo Tree Search, in short MCTS, is 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 was made by Bernd Brügmann in 1993 [15], who suggested 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.

AlphaZero uses a variant of MCTS called UCT, which was formulated in 2006 [19] by L.Kocsis et al. and used in computer go for the first time in the same year [16]. UCT stands for "UCB applied to trees", where UCB is the "Upper Confidence Bounds" algorithm of the *multi-armed bandit problem* [14]. Previous to AlphaZero multiple other strong computer go programs based on MCTS have been released, such as Pachi [10] or 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. [19] 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.

UCT as described by L.Kocsis et al. is a rollout-based planning algorithm, which repeatedly samples possible episodes from the root of the tree. An episode is a possible sequence of moves that can be played form 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 a way to evaluate a position is by randomly playing it to the end a number of time and using the average results of these random playouts to evaluate the position. UCT is thus an incremental process, which 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

TODO: Should I describe AlphaGo here in more detail? I don't really care about all the complications it did compared to AlphaZero, but it might still be interesting?

The AlphaZero algorithm [25] is the application of AlphaGoZero [27] to games other than Go. AlphaGoZero is a significant simplification of AlphaGo [24]. AlphaGo is the first program to reach superhuman performance in Go by combining MCTS with deep learning.

AlphaGo used a complicated system involving initialization with example games, random roleouts during tree searches and used multiple networks, one to predict the value of a position, another to predict promising moves. AlphaGoZero drastically simplified the system by only using a single network and not doing any roleouts anymore, instead the network evaluation for the given positions is directly used.

The difference between AlphaGoZero and AlphaZero is mainly that AlphaGoZero involved comparing the currently trained network against 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 always uses the current network to produce new games, surprisingly this appears to not give any disadvantage, learning remains stable.

The main advantage of the "Zero" versions is that they do not require 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 final playing strength. Additionally it allows to use the algorithm for research of games for which no human experts exist, such as No-Castling Chess [20].

2.2.1 The AlphaZero algorithm

The AlphaZero algorithm [25] uses MCTS with the UCT formulation, as described in section 2.1, and modifies it to incorporate a deep neural network which serves as

a heuristic to bias the tree search and provide 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 to form a closed cycle of self improvement, which starts with a random network and has been shown to successfully learn to play various games, such as chess, shogi or go on the highest level, if given substantial computation resources.

Work written concurrently to AlphaGoZero describes a similar algorithm tested with the game hex [13]. They describe the algorithm as thinking slow, via MCTS, and fast, via the network alone, in reference to results from human behavioral science [18]. AlphaZero in that sense learns in a similar fashion as humans: At first a time intensive "thought process" is used to reason about a new problem, but with pratice good decisions can be made much faster.

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, which 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, 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 chapter 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 which 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 reaches again 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 analyzed it thoroughly.

To use the network outputs in UCT, the formulations are changed. The action a in a given node is selected accoring 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.

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

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

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 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, forming a sort of ensemble of network evaluations on possible future positions.

A drawback of this approach is the high computational cost, as in practice for good results the search tree to play a single move has to be grown to at least hundreds of nodes, requiring 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 motivates my work in researching possible improvements to the algorithm that allow learning to progress faster or with less example games played.

2.3 Extensions to AlphaZero

Many improvements to the AlphaZero algorithm have been proposed, typically aiming at reducing the extreme computational cost of learning to play a new game. In this

section I will present a collection of such proposals, especially focusing at ideas that do not require game specific knowledge to be incorporated. Still, some improvements might work better on some games than others, depending on hyperparameters used.

In section 3.5 on page 21 I will present experimental results of my own implementation of a selection of these proposals by previous work.

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. Some ways have been found 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 [17], has been proposed by the Leela Chess Zero projects, which reimplements AlphaZero for Chess as a distributed effort [2]. A very similar approach is used in [29] which also cites the similarity to Squeeze-and-excitation networks.
- The original design of the network only uses 2 filters in the policy head and a single filter in the value head, using 32 filters has been found to speed up training, as reported by [31] based on earlier work done by the Leela Chess Zero project.
- Cyclic learning rates, as developed by [28], can be used to improve the network fitting to the data, [31] shows a modest speed up.

2.3.2 Modification of the tree search

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

- Instead of using a single network [21] proposes to combine multiple networks of different sizes, especially two networks, one big, 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, which allows 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 [29]. They drastically reduce the number of MCTS playouts randomly in most of the moves played. It allows to play more games in the same time, somewhat similar to the concept of using a combination of a small and a big network, the authors 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.

- 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 [3]. 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. From the AlphaZero paper it is not clear to what degree DeepMind used such optimizations.

- The Leela Chess Zero project suggests analyzing statistics, namly the Kullback-Leibler divergence of the policy as it evolves during the tree search, to understand how complex a position is and apply more tree search evaluations on complex situations. They find a noteable increase in playing strength [4] using the same computational budget.

2.3.3 Learning target modifications

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

- [29] also shows 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 point at 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 [29], force 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 a very uncertain position and a position that is very likely drawn [5].

- [12] 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 [31] proposes to use 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, since this MCTS-based value has other accuracy issues they specifically propose to combine the two values, which shows a noticeable improvement.

2.3.4 Training data enhancements

- There can be multiple copies of the same identical position in the training data. [31] 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 noticed in [31], 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. A modification to the training data windowing is proposed, which fixes this.

3 Experimental Setup

I have developed a framework for experiments with AlphaZero which allows to easily setup different configurations with a configuration file, switching on or off various features and improvements under investigation. TODO talk a lot more about this, publish the code in a more refined manner...

The framework base implementation of AlphaZero uses a different network target for game results, which encodes a dedicated value for a draw. This means for a two player game the possible outcomes are the win of either player, or a draw. This differs from the original implementation, which used a single output with values between 0 and 1. There might be a slight change in learing efficiency from this, but the main reasons for implementing this were of technical nature in the context of the abstraction-driven framework and no further research has been done on the difference between these two ways of implementing the value target.

My experiments are split into multiple parts:

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

3.1 Testing on Connect 4

To reduce costs of experiments I am restricting myself to experiments with the game Connect 4. Unlike Go, Connect 4 is a game with a known solution, as it is a solved game for which strong solvers are available [6, 7, 8], i.e. for any given position the solver can quickly find the most optimal move to play.

This allows me 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 to run supervised training of the used network on a dataset of games played by the solver, establishing a maximum performance possible by the network.

3.1.1 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 my results are to previous work.

Starting my work I knew of the results of [22][OracleDevs et al.], who claim to reach 97% to 99% accuracy, depending on the definition of a correct move, compared to a Connect 4 solver. However my attempts at reaching these numbers, outside of supervised training, failed. I believe this is due to various decisions made during the generation of the dataset.

A first decision is on what is 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. [22] 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 Oracle are:

1. How to play the games exactly?

2. Should duplicate positions be filtered out?

3. Should trivial positions, i.e. positions that have no wrong answer, be filtered out?

In the end I opted to make my dataset as hard as possible: Games played to generate the test dataset, as well as the dataset used for supervised training are generated without duplicates, without trivial positions and only the stronges possible moves are accepted as correct. Half the moves played in the games used to generate the dataset were played by a solver [7, 8], the other half by a call to *random()*.

The average number of correct moves per position is a good metric to determine how challenging a dataset is. In my datasets this value is 1.8.

In contrast [22][OracleDevs et al.] 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 explaining differences in accuarcy values between my work and theirs.

3.2 Evaluation of training costs

The goal of my work is to identify ways to reduce the substantial costs of training using AlphaZero. The majority of GPU capcacity is spent on self-playing games to be used as training material for the neural network. In all my experiments I train on a single GPU, evaluate newly produced networks on another single GPU and run self-play workers on a p2p cloud service[1] using up to 20 GPUs of various types.

This huge inbalance between training hardware and self-play hardware can be seen in related work as well, e.g. [26][Silver et. al.] used 5000 first-generation TPUs for self-play and 16 second-generation TPUs to learn play Chess on super-human, and potentially even super-classical-engine, level within hours. Similarly [29][David J. Wu] 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 my experiments I will use this fact to simplify the training cost measurement by ignoring the costs of the network-training and only measure the time needed to generate self-play games.

Since my main source of computational resources for self-play is a p2p cloud service which provides unreliable but cheap GPU time I am not using a constant number of GPU workers between experiments or often even change the number of workers during training.

[1] `https://vast.ai`

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 my experiments is stated as estimated self-play time on that machine and bottlenecked by GPU capcacity on it.

3.3 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 as outlined in section 3.1.1, is used. 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 as test data. 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. Results are shown in table 2.

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

What surprises of these results is the quite 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.

I suspect the issue is 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.

To rule out overfitting on the dataset, another dataset with 10 million positions taken from 10 million games was generated. This dataset achieved a move-prediction accuracy with the 5-block network of over 97%, but the win-prediction still stayed below the accuracy reached in AlphaZero training runs, that played less 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.

3.4 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, networks will be trained in iterations of 300000 produced examples. Every network is evaluated on the test set for accuracy compared with the Connect 4 solver. Network training happens using SGD with momentum of 0.9, weight decacy of 0.0001. The learning rate starts at 0.2, is dropped to 0.02 at iteration 7 and 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. A training window of 2 million positions is used.

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 my work is to look for efficiency gains, not maximum final performance. Therefore the smaller network is used to allow for faster experimentation.

3.4.1 Hyperparameter search

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

To establish a baseline with sensible hyperparameter 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 bayesian optimization was tasked with optimizing the accuracy after two hours of real time.

To yield meaningful results after just two hours on a single GPU, the size of iterations were chosen in such a way to make faster progress, at the cost of the quality of the final results, i.e. less 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 to pick the best hyperparameters for the following work.

The hyperparameters optimized for are listed in table 3. 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 12.
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 a 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 draws as desirable as wins, a low value as undesirable as losses.

Table 3: Hyperparameters searched

The hyperparameters chosen for further investigation in full experimental runs are shown in table 4. 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 4: Hyperparameter sets selected for further investigation.

Full training runs are done 5 times for every of the hyperparameter sets under investigation. In every run the training is stopped once the MCTS accuarcy has not improved for 10 iterations. The 5 runs are used to calculate a mean, which is used to compare the different runs against each other.

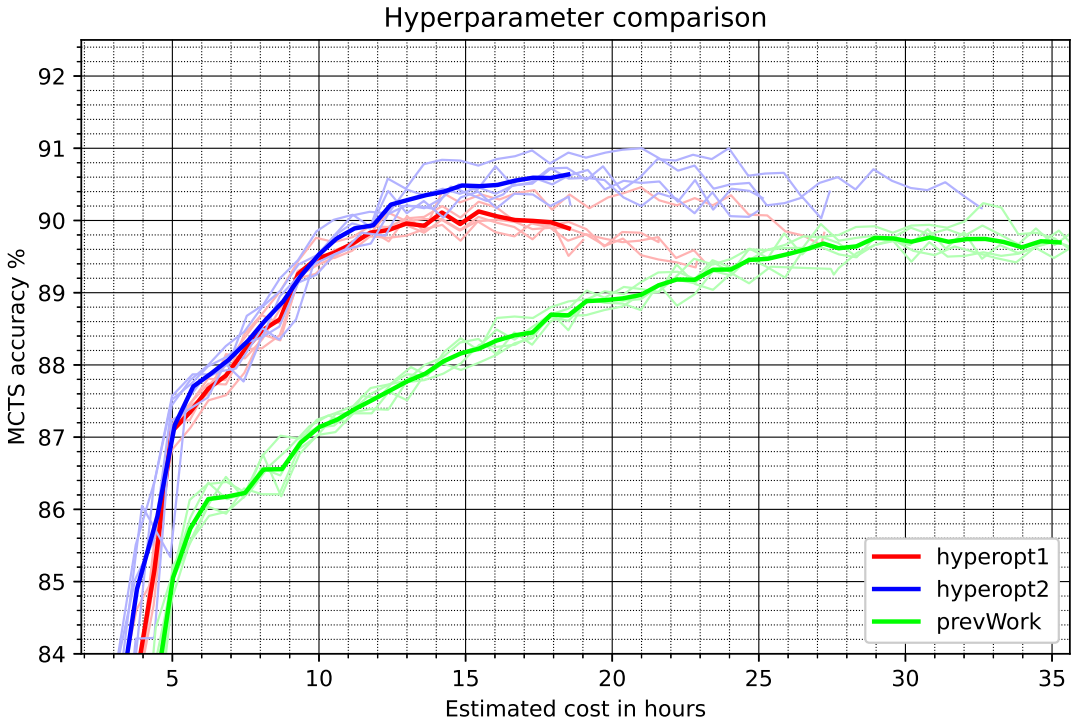

Figure 1: Results of the runs to determine a good hyperparameter set. Mean is only calculated until the first of the single runs stops showing improvements.

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 due to the fact that previous work was on other games, using different implementation details, such as different tree sizes, iteration sizes, network sizes or other differences.

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

3.5 Extended Baseline

As outlayed in chapters 2.3 on page 12, 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 proposed improvements are cumulative with already known ways to improve AlphaZero.

3.5.1 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 thus is provided with a large set of duplicate input values, which will have conflicting target values to be learnt. 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 [31][OracleDevs et al.] propose to merge duplicate positions to counteract this.

My implementation of this acts in the training worker and only adds new positions 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 hashmap. Every time a duplicate is produced by a self-play worker and send 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 values 0.2, 0.5 and 0.8 are tested in single runs and compared against the baseline to evaluate how promising this approach is. Figure 2 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 experimenets, 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 iterations become. The training window stays at 2 million positions, which all will be distinct from each other with deduplication.

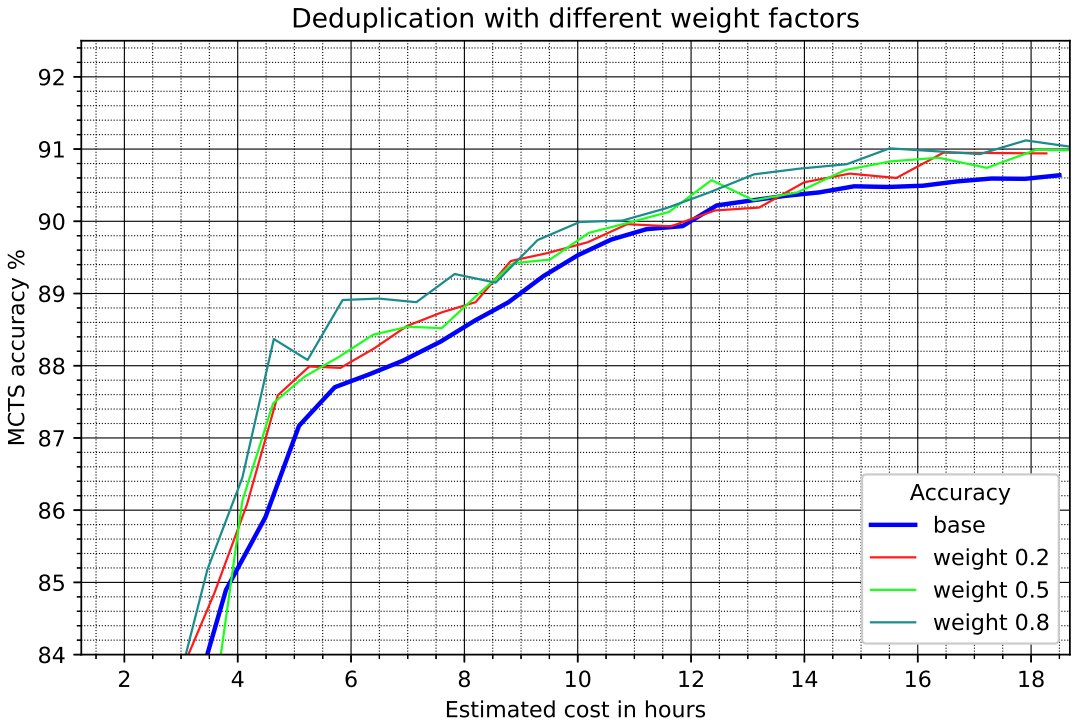

Figure 2: Comparison of different choices for $w_{\text{duplicate}}$. 0.8 is chosen for all further experiments.

3.5.2 Cyclic learning rate

Cyclical learning rates are a proposal [28] to speed up general neural network training. [31][OracleDevs et al.] claim some improvement using them in the context of AlphaZero.

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

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 is dropped 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 rate, especially the supervised training.

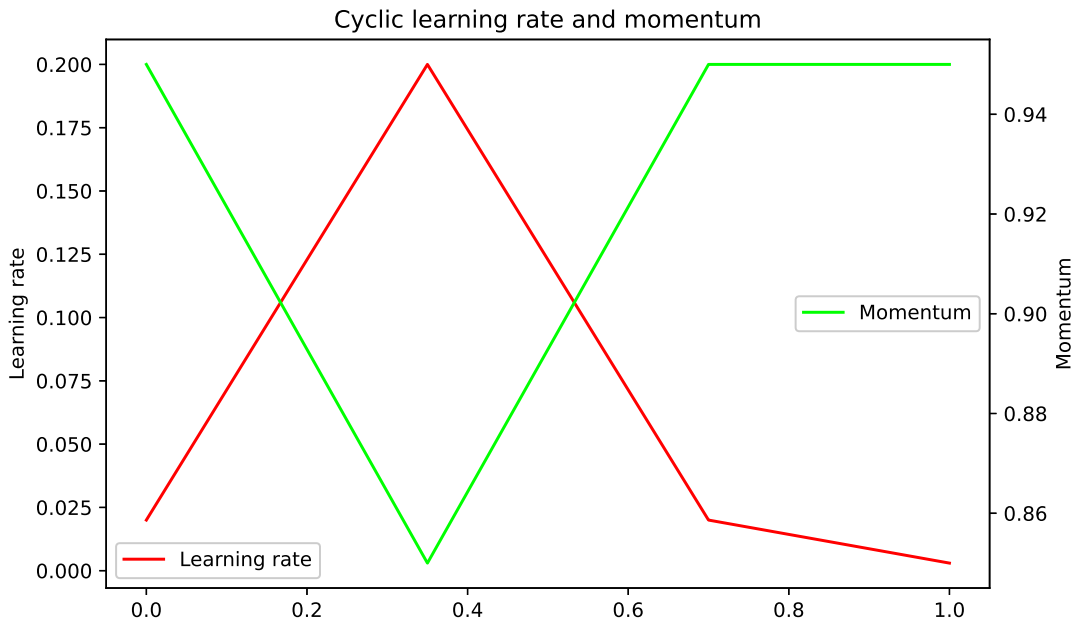

Figure 3: The change of the learning rate and momentum over the training of a new network iteration when using cyclic learning rates.

These curves can be seen in figure 3. Additionally, 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.

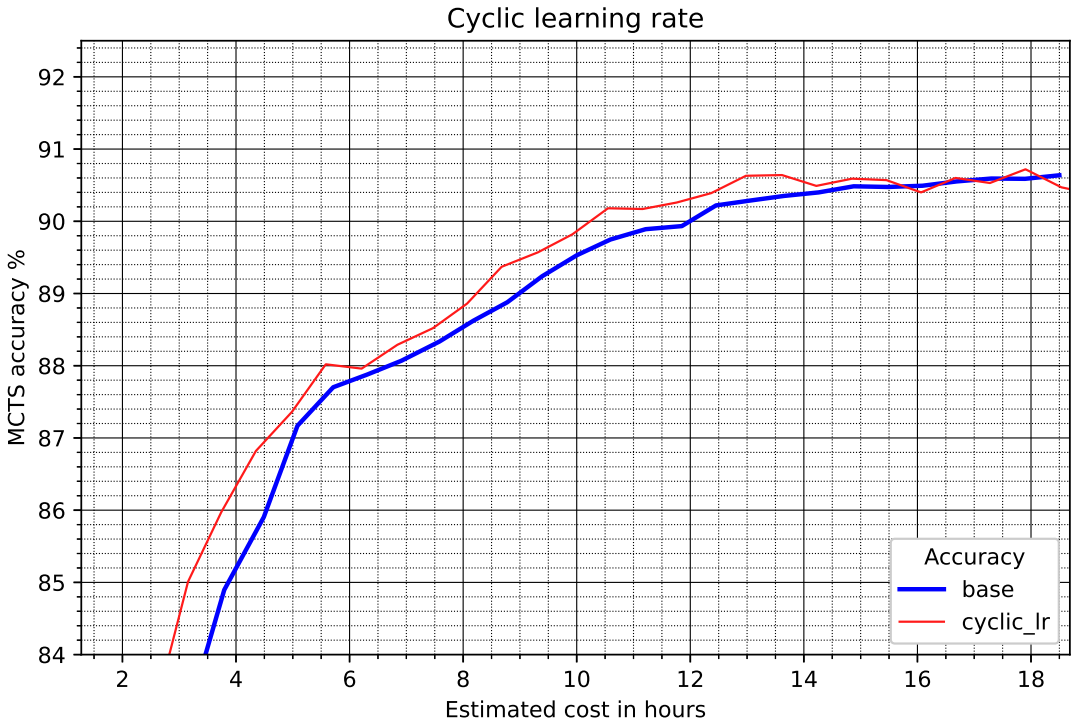

Figure 4: Comparison of the usage of cyclic learning rates and momentum with the baseline.

A single run was done with the baseline configuration, using the described cyclic learning rate and momentum. The results can be seen in figure 4. Slight improvements were found, especially in the earlier stages of the training. Cyclic learning rate and momentum was added to the extended baseline configuration.

3.5.3 Improved training window

[31][OracleDevs et al.] notice that a jump down in the training error can be observed in the first iteration that pushes the examples of the very start of the training out of the training window. It stands to reason 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. This way the training examples of early iterations are removed from the training data more quickly. This is called a slow training window.

In my implementation of the slow training window, I start with a training window 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 seen in figure 5.

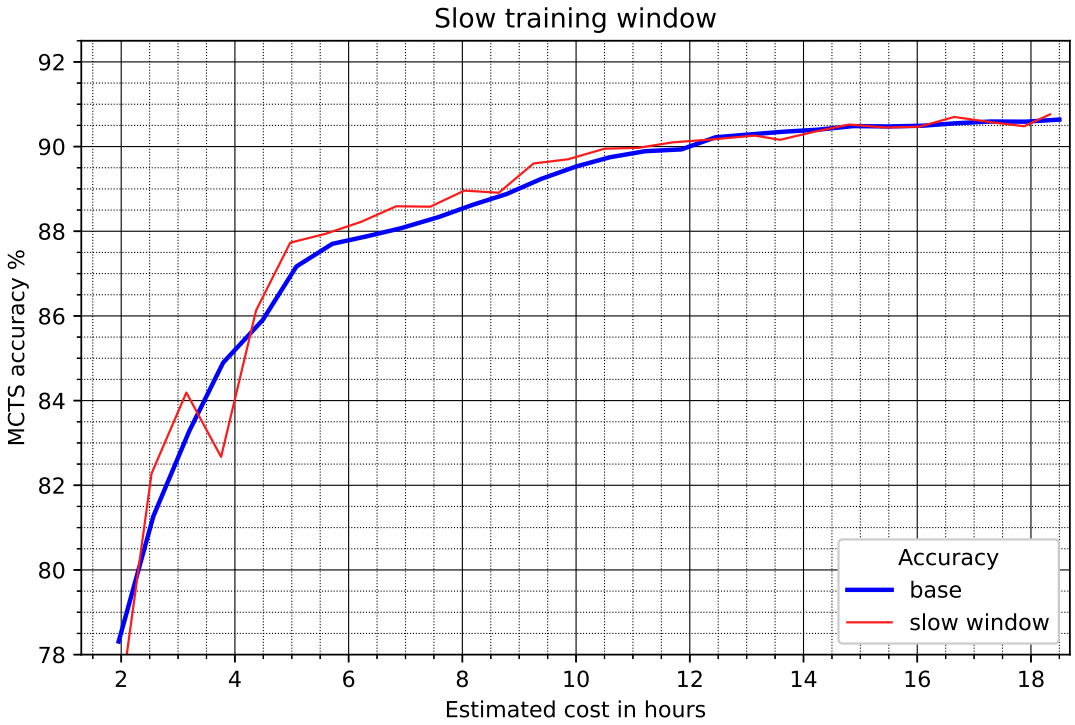

Figure 5: Comparison of the usage of a slow training window with the baseline.

The slow window is added to the extended baseline, since it does not appear to hurt and it follows a sound idea. To accomodate the fact that 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.

3.5.4 Playout Caps

Playout Caps [29][Wu et. al.] 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 marginally 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.

I have implemented Playout Caps and evaluated them 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. Results of a test run can be seen in figure 6.

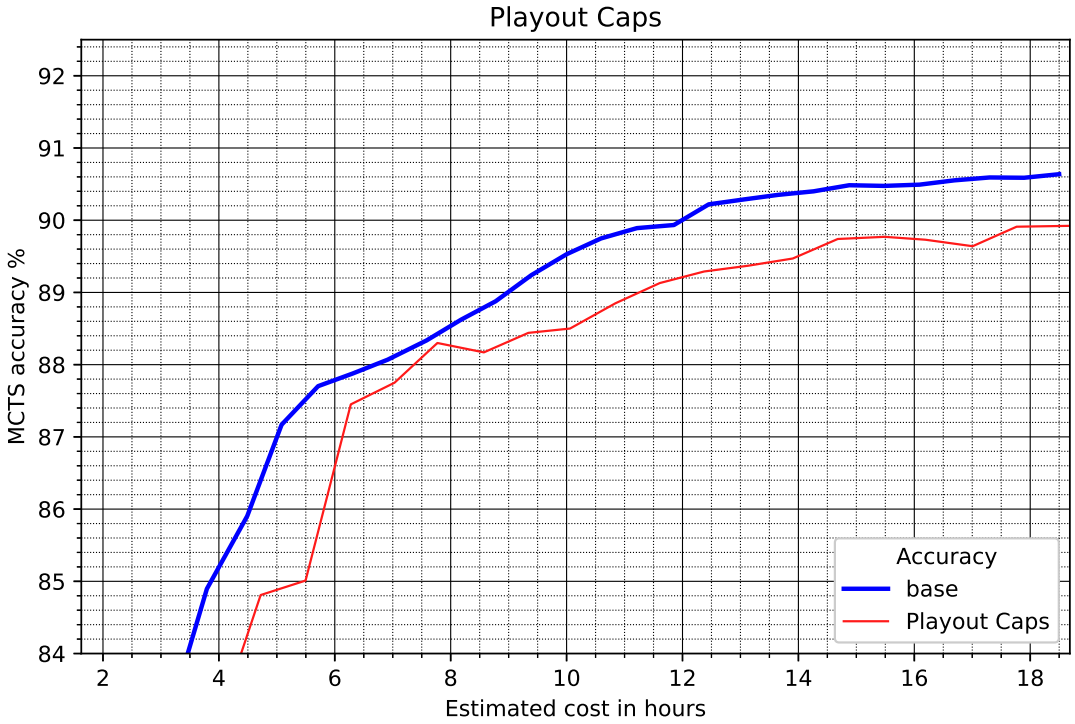

Figure 6: Results of implementing Playout Caps on Connect 4.

Clearly Playout Caps appear to hurt performance on Connect 4. This likely stems 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[2]. 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.

3.5.5 Improving the network structure

The Leela Chess Zero project [2] proposes to use Squeeze-and-excitation elements in the network [17]. These are a general development of deep neural networks, which have been shown to improve learning efficiency at a small additional cost.

I have modified the network in my implementation to use Squeeze-and-excitation residual blocks. A result of a test run can be seen in figure 7.

[2] https://homepages.cwi.nl/~aeb/go/misc/gostat.html

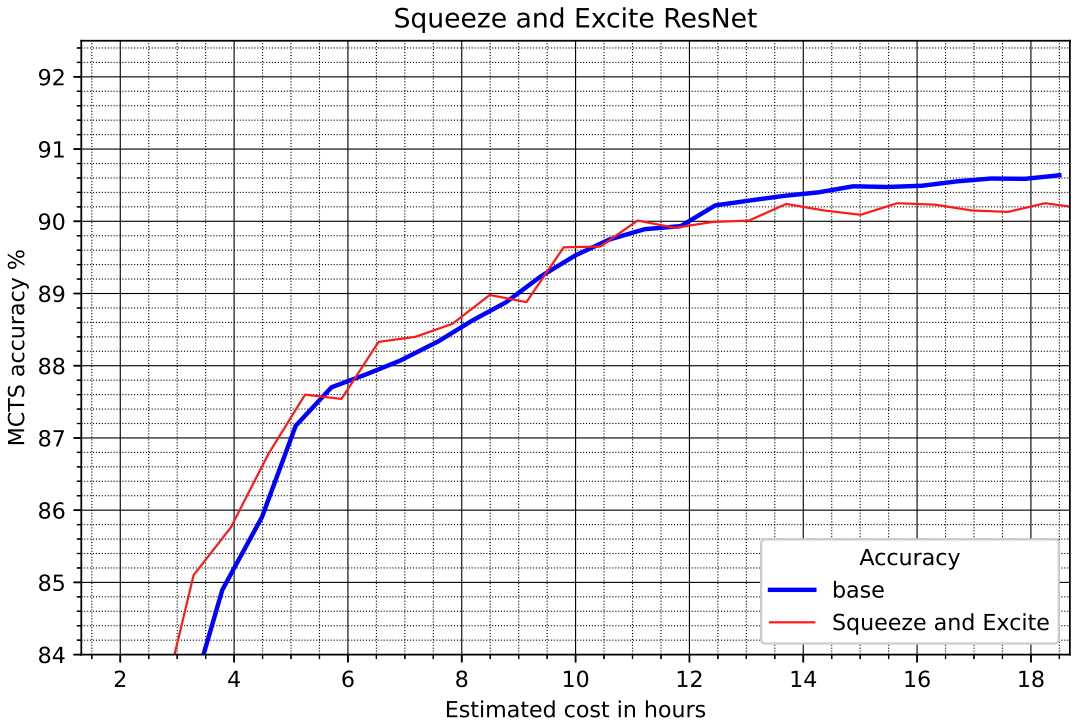

Figure 7: Results of implementing Squeeze-and-excitation elements in the network.

It appears there might be some gains early in the training and possibly some small losses later in the training. I have decided to add this modification to the extended baseline, as it does not seem to hurt performance much and logically should help, especially given more distinct training data from deduplication.

3.6 Baseline results

Combining deduplication, a cyclic learning rate, a slow window and squeeze-and-excitation elements yields substantial improvements, yielding above 91% final performance in all runs. The results can be seen in figure 8.

This extended baseline forms the basis of all further experiments, unless stated otherwise all these previous improvements are active.

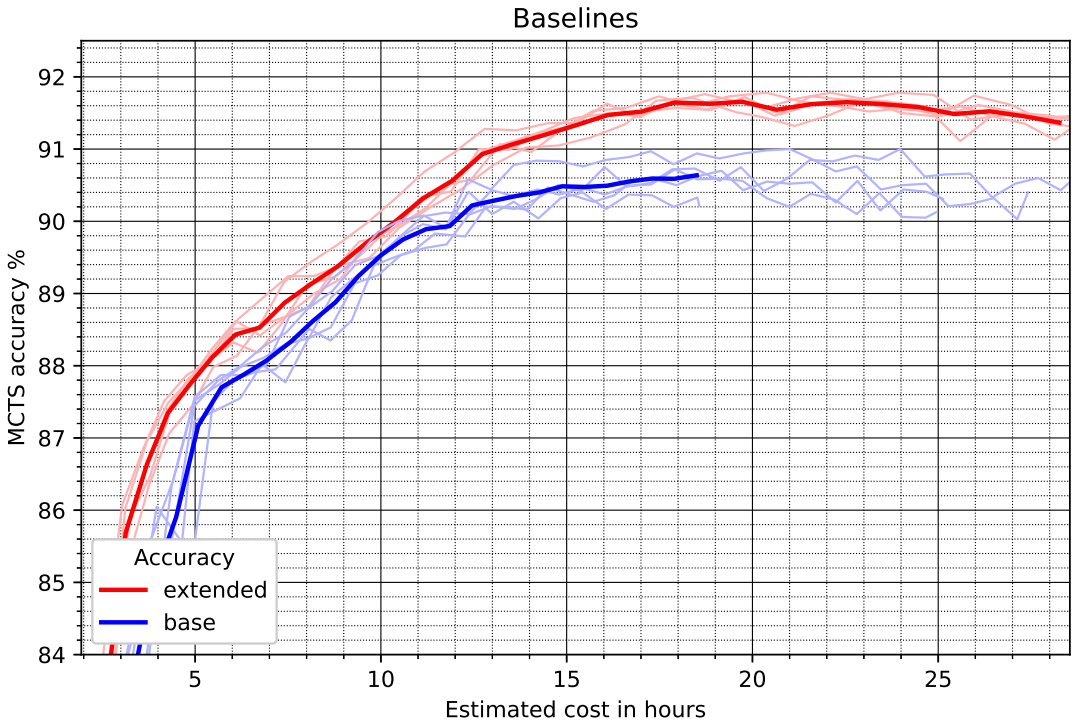

Figure 8: Comparison of the baseline and the extended baseline. Mean is only calculated until the first of the single runs stops showing improvements.

4 Evaluated novel ideas

In this section I preset my proposals on possible improvements to AlphaZero, including experimental results.

4.1 Using the self-playing phase as an evolutionary process

There are many hyperparameters involved in AlphaZero which have to be tuned. An important subset of these hyperparameters controls the exact behavior of the MCTS search, for example C_{puct}, as discussed in section 2.2.1, equation 4 on page 12. A hyperparameter search shows that these MCTS-parameters have a noteable impact on how much the MCTS can improve the playing strength of the network alone. Better parameters therefore could translate into faster training progression.

I propose to use 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 system such as Elo. The games

played in this league make up the self-playing phase, but additionally the results of these games will 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 the top 15 players are each mutated into two new versions of themself. 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 they drop in rating too much.

Then again games are played, until enough games were played by the most recent generation, which triggers 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 every iteration in such a training run, every time a new network is completed, the current set of players is saved. To then evaluate a network the best player of the players-generation which has not reported any games with a network newer than the one being evaluated is used.

4.1.2 Evolution of players

Gaussian mutation [30][page 7] is used to evolve the hyperparameters of players. Valid parameter ranges are defined and enforced by looping values below the minimum back towards the maximum and the other way around. 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 is defined: $(\min_i, \max_i)$.

The initial set of players selects values at random according to a uniform distribution over the allowed range of values. To create as new player (w_j, v_j), an already existing player (w_i, v_i) is used as shown in formulas 5 and 6 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 $(\sqrt{2n})^{-1}$.

$$v_j = v_i \exp(\tau' N(0,1) + \tau N_i(0,1)) \tag{5}$$

$$w_j = w_i + v_j N_i(0,1) \tag{6}$$

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 behavior of the MCTS search. First candidates are parameters such as cpuct, fpu and drawValue, as described in table 3 on page 20.

An additional candidate is to control the size of the MCTS tree by a the development of the Kullback-Leibler divergence as more nodes are added to the MCTS tree, as suggested by the Leela chess zero project [4].

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. Specifically, 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 player evolution is tasked with finding 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.

4.1.4 Experiments

I implemented the evolution approach and ran initial experiments, one evolving cpuct, fpu and drawValue, the other the Kullback-Leibler divergence threshold for play with a dynamic number of MCTS nodes. Results can be seen in figure 9. Neither approach produces promising results.

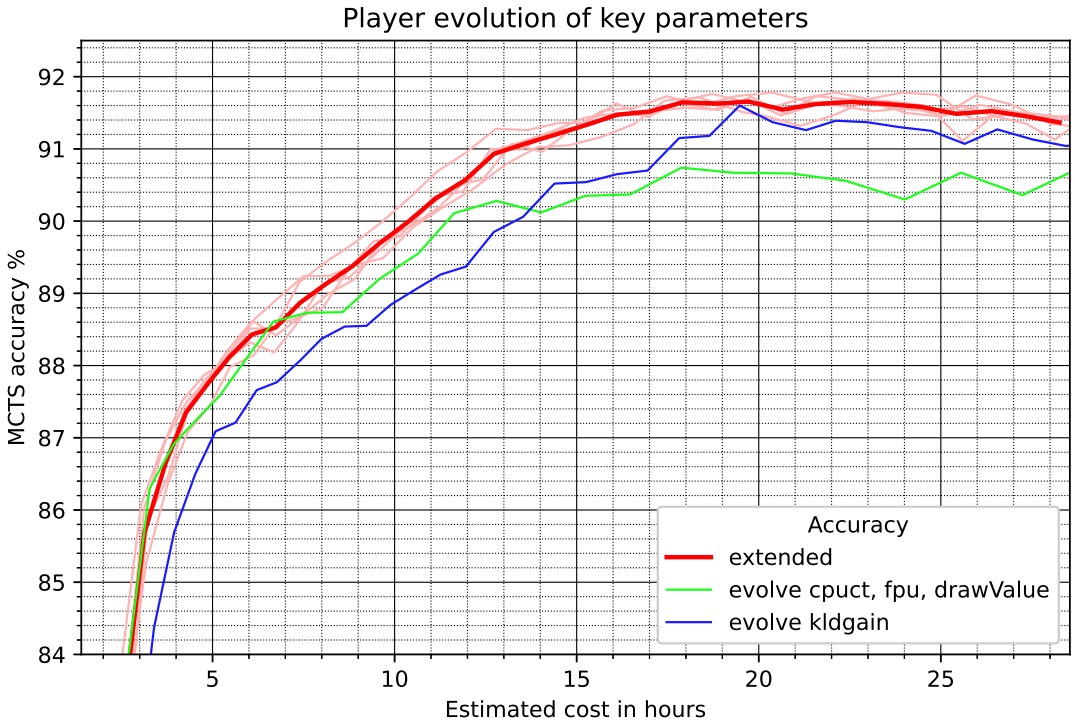

Figure 9: Initial results of evolving hyperparameters.

One potential issue can be seen for the evolution of the MCTS hyperparameters in figure 10, which shows a substantial drop in game play diversity when evolving those parameters.

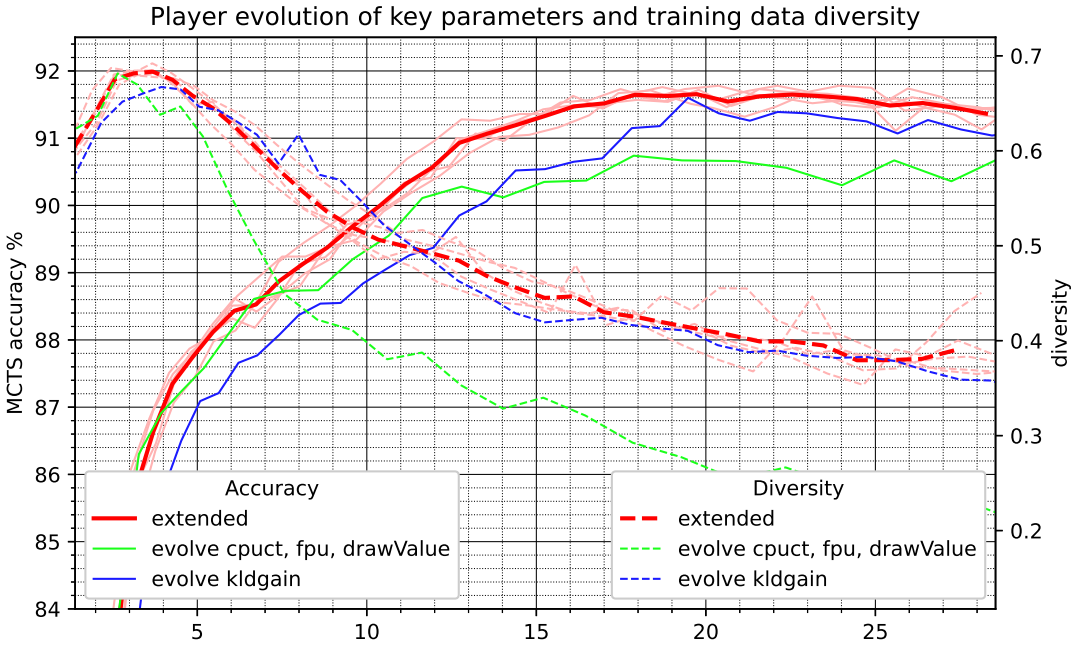

Figure 10: Evolving basic MCTS parameters results in much lower game diversity, evolving the KL divergence threshold stays similar to the baseline.

4.1.5 Requirements for evolution to succeed

The weak results of the first experiments with evolving players do not show good results. This raises the question of what conditions have to be fulfilled to improve training efficiency with the evolutionary process. A few key points can be established, that need to be verified:

- 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 I have done tests with another hyperparameter, which was designed only for this purpose, the inversion-hyperparameter $\mathbf{I}$. $\mathbf{I}$ modifies the move distribution found by MCTS.

If m is the vector that represents the move distribution produces by MCTS, then the new move distribution is calculated by the inversion parameter as shown in formula 7. 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 originally were considered to be the worst moves, while the best moves will receive the lowest weight. It is clear, that $\mathbf{I}$ should be 0, to not disturb the MCTS results at all.

$$m_{\text{inverted}} = \mathbf{I}((1-m)/\sum(1-m)) + (1-\mathbf{I})m \tag{7}$$

As a first test of the players league evolution concept, it is tested how well this parameter is optimized towards zero. For this runs of 10 player generations are done, evolving $\mathbf{I}$ in the range between 0 and 1. Table 5 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 face of reduced impact of the parameter, two more sets of 3 runs were made, were 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 clearly show, that the evolution of players works and can pick parameters that play stronger, even in face of 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 generation to optimize parameters.

Generation	$\mathbf{I}$, $p = 1$	$\mathbf{I}$, $p = 0.25$	$\mathbf{I}$, $p = 0.05$
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 5: Results of optimizing the inversion hyperparameter. Clearly the evolution works, $\mathbf{I}$ is reduced substantially over the generations.

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

For this purpose I will compare three sets of MCTS hyperparameters: 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. Results can be seen in table 6. 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 6: Various hyperparameters show different accuracy. Directly optimizing for accuracy shows that neither the baseline, nor the evolutionary hyperparameter perfectly capture 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, which performs equally good to the baseline parameters, however it starts weaker.
- Neither the baseline nor the evolved player reach 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 does capture both of these developments over the run, reducing the drawValue and the fpu once the best network is generated. Evolution however prefers lower cpuct values.

Lastly the three hyperparameter sets are compared in matches of 1000 games against each other, 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 winrate of the hyperparameters is not a good proxy for faster training progress. Table 7 shows the results of the matches between the different hyperparameter sets.

Random network

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

Network iteration 1

VS	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

VS	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 7: 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 correct 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 winrate of hyperparameters in playing against each other using the same network

is not a good proxy for learning progress of AlphaZero. Still, the evolution of hyper-parameters by using self-play clearly works, 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

TODO actually, play them as a MCTS-tree instead. Preferably a single one using a different mode of distribution. TODO or still additionally try out the idea below, playing as MCTS-trees is not working out.

Various previous work points at how the value target is starved for data, since it only receives examples by whole games that can be tarnished by random mistakes in the last few moves and suggest improvements targetting this [29], [31], [21].

I suggest to occasionally copy game states and playing on differently, for example using the 2nd best move found, in the copy. This will result in early game states having multiple possible continuations, which are played out to the end. Thus early game states played will tend to have multiple known outcomes, which can be averaged to produce a more accurate estimate of the value of those positions. The influence of a single bad move towards the end should be reduced and the value head of the network should learn faster, since the training data will be of higher quality. There should be no substantial computation cost to implement this, as it just changes the structure in which games are played.

TODO: pretty diagram showing how this forms a tree-structure

4.3 Using network internal features as auxiliary targets

[29] shows that using domain specific features as auxiliary targets to regularize learning is very helpful. This motivates the search for an automatic way to find such targets.

Since I am especially interested in ways to improve learning performance at little additional computational cost, I suggest to use internal features of the network from previous network iterations as regularization targets to learn for future iterations. Since the internal features are learnt either way, this only introduces a small additional cost in the form of the computation of the increased size of the output layer.

Specifically I propose to designate a layer $\mathbb{F}$ in the network and connect an additional head $\mathbb{H}_{future}$ to the layer before $\mathbb{F}$. The network stem $\mathbb{S}$ up to $\mathbb{F}$ is then used to produce additional learning targets by applying $\mathbb{S}$ to the 2-ply future of every game state used as training data. This uses the encoding the network found to be meaningful for game playing to encode the immediate future of the game and uses this as a learning target.

The usage of the 2-ply future is required, since otherwise the network would just be tasked with producing its own internal features. Instead the network will be required to predict the future game situation after one action by each player.

The reason for attaching the head $\mathbb{H}_{future}$ before $\mathbb{F}$ is such that $\mathbb{F}$ only contains an encoding relevant to the value and policy heads, not an encoding that also tries to encode information for $\mathbb{H}_{future}$ of previous iterations.

An extension of this could be to try n-ply futures, for example 4-ply or 6-ply to evaluate how predicting a more distance future could be helpful, if 2-ply prediction shows promise.

References

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

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

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

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

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

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

[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.

[10] Pachi go engine. https://github.com/pasky/pachi.

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

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

[13] 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.

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

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

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

[17] 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.

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

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

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

[21] 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.

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

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

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

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

[31] 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.