**Proposal Masterthesis**

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

COLIN CLAUSEN

29.11.2019

Dozent: PROF. DR.-ING. SVEN TOMFORDE

# Contents

# 1 AlphaZero basics

The AlphaZero algorithm [15] is a simplification of AlphaGoZero [16] and AlphaGo [14]. AlphaGo gained fame for beating top human players at the game of Go, a feat previously thought still a decade away. AlphaGo used a complicated system involving initialization with example games, random roleouts during tree searches and used multiple networks for different tasks. AlphaGoZero drastically simplified the system by only using a single network for all tasks, 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. This allows the algoritm to find the best way to play without human bias which seems to slightly increase final playing strength. Additionally it allows to use the algoritm for research of games for which no human experts exist, such as No-Castling Chess [11].

## 1.1 The AlphaZero algorithm

The algorithm, as described in [15], has the goal of generating training examples to be able to train a neural network to play at extremely high level. Therefore at the center of learning stands a deep neural network which, given a board position, produces a policy and a value. The policy describes move likelihoods, the value the chance of winning the game from the given board position.

The main idea is to produce examples of how to play better than the network by guiding a monte carlo tree search with the network. This averages out many position evaluation of the network over possible futures of a given position, producing a new higher quality policy for the position.

By a work written in parallel to the AlphaGoZero paper this is described as thinking fast, via the network, and slow, via the search and compared to how humans at first have to think actively about a new task to do well, but with practice may learn to do well unconsciously without active thinking [6].

The MCTS algorithm stores for every state-action pair $(s, a)$ in the tree a tuple of statistics, $\{N(s,a), W(s,a), Q(s,a), P(s,a)\}$, with $N(s,a)$ being the visit count of

that action on that state, $W(s,a)$ being the total action value, $Q(s,a)$ beeing the average action value and finally $P(s,a)$ being the prior probability of select $a$ in $s$, which is the network prediction. To decide on a better policy for a given situation a number of simulations is executed. Every simulation starts at the root node, picking actions to go down the tree until a leaf is discovered, at which point the network is invoked to find the prior probability for that leaf and backpropagate the value estimation by the network back up to the root, updating the node statistics on the way.

The action to be played on a given node $s$ is selected according to $argmax_a(Q(s,a) + U(s,a))$ where $U(s,a) = C_{puct}P(s,a)\frac{\sqrt{N(s)}}{(1+N(s,a))}$ with $N(s)$ beeing the parent visit count and $C_{puct}$ beeing a constant controlling exploration. Once a leaf is encountered the value predicted by the network is backpropagated upwards, adding to $N(s,a)$ and $W(s,a)$ of nodes and recalculating the average action values.

The final policy generated by MCTS is determined by the visitation count of the actions of the root node.

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. Since the policy generated by the MCTS is always better than the one by the network alone this improves the playing strength of the network, which in the next iteration will allow MCTS to produce even better examples of play.

# 2 Previous work

Understanding previous work is not only important to understand where to search for further improvements without reinventing the wheel, but also to speed up the learning progress to be able to experiment faster on proposed ideas.

## 2.1 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 [10], has been proposed by the Leela Chess Zero projects, which reimplements AlphaZero for Chess as a distributed effort [1]. A very similar approach is used in [17] which also cites the similarity to Squeeze-and-excitation networks.

- The original design of the network only uses 2 layers 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 [18] based on earlier work done by the Leela Chess Zero project.
- Cyclic learning rates can be used to improve the network fitting to the data, [18] shows a modest speed up.

## 2.2 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 [12] proposes to combine multiple networks of different sizes, especially two networks, one big, one small. 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 [17]. 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 termainal moves through the search tree to simplify the search is proposed by the Leela Chess Zero project, with a moderate improvement in playing strength [2]. 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 [3] using the same computational budget.

## 2.3 Learning target modifications

- [17] Proposes to predict the opponents reply to regularize training. A modest improvement is shown.

- [17] 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 [17], 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 [4].

- [5] 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 [18] 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.

## 2.4 Training data enhancements

- There can be multiple copies of the same identical position in the training data. [18] 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 [18], 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 Motivation

The AlphaZero algorithm is a way to attain super human playing strength in arbitrary board games, even very complex ones like Go. At least for more complex games like Go this is however still only possible by using very substantial amounts of processing power, thousands of machines clustered together, making it impossible to leverage the algorithm on hard problems for anyone but the largest organizations.

This motivates the search for improvements that allow to achieve results with lower requirements on computational power, preferably without losing the generality of AlphaZero. Most time during AlphaZero training is spent on generating example games, thus it follows that to improve learning speed either more learning has to happen per game or games have to be played out faster.

Previous work has established that large efficiency gains are possible and further improvements shall be systematically investigated and compared to previous improvements as well as a the baseline algorithm to understand how the learning progress is affected and if efficiency gains are accumulative with other known enhancements. Additionally it shall be attempted to find metrics by which to measure the effects of the modifications beyond simple playing strength.

# 4 Goals

First a clean implementation of the algorithm must be written, special care should be taken to be able to modularize the system as much as possible to later make enabling and disabling various modifications easy. To be able to understand the learning progress as much as possible various metrics should be tracked during training.

As a second step as many known improvements as possible should be added, focusing on simple ones, to be able to speed up experiments on newly proposed ideas and be able to check if new proposals enhance playing strength on top of existing improvements.

Finally some specific ideas to improve the algorithm shall be investigated.

## 4.1 Means of evaluation

To be able to experiment fast without massive usage of computing resources simpler games are typically used when evaluating AlphaZero. Following previous work [13] Connect 4 is suggested to be used as a primary evaluation game, because classical methods can achieve perfect play and thus evaluation of training progress can be done by judging moves made by AlphaZero in absolute terms. Depending on the final speed

of the implementation and available resources the size of the board for Connect 4 can be scaled to allow experiments to finish within acceptable time frames.

In similar fashion the game of Hex on a 9x9 board can also be used, for example this is done in [5], as there are datasets of near-perfect play available, this however seems to be a slightly inferior approach, as the network might be penalized in situations that have multiple optimal moves. Also the board is still larger than Connect 4, so experiments might take too long.

If time and computational resources permit another idea for evaluation on a more complex game would be to implement Chess and judge training progress by the average centipawn loss as measured by a strong Chess engine such as Stockfish, as can be done to judge human players [7]. Since Chess is not a solved game this is less accurate than Connect 4, but has the advantage of giving results on a much more challenging domain.

Generally the learning progress relative to spent computational resources should be measured and compared.

## 4.2 Understanding the learning process

An interesting property of the AlphaZero algorithm is the perceived stability of the training. Other learning systems often struggle to learn two players games in a pure self-play regime, because they get trapped in an endless cycle of learning and unlearning the same mistakes.

A typical way to prevent this normally would be to include earlier iterations of a learner in the learning process, to prevent forgetting previous knowledge. This was originally done in AlphaGo, however with further work it became clear that even in the AlphaZero regime, with no measures taken at all to prevent forgetting learning still works fine. This raises the question of how stable the training truly is. It might be that some amount of forgetting still occurs, just not enough to stop progress.

To research this a dataset of specific game situations could be used, checking which positions are played correctly after every learning iteration. This way it can be compared how every learning iterations adds or removes more understanding of the game to the used neural network. A better understanding of the learning progress could be valuable to suggest additional ways to speed up training, for example in regards to how the training data window is currently managed.

Further statistics could be collected on the training progress, such has the variation of positions encountered, win/loss rates between the first and second player, draw rates or various statistics on the behavior of the tree search.

## 4.3 Specific improvements to be investigated

### 4.3.1 Network modifications

Further advances in research on artificial neural networks could be applied, such as Gather-Excite networks [9] or the especially resource efficient network blocks of mobilenetv3 [8]. This does not constitute a major advance in the AlphaZero algorithm, but may still increase efficiency of the learning system and is thus worth at least a short investigation.

### 4.3.2 Playing games as trees

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 [17], [18], [12].

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.

### 4.3.3 Automatic ways to find auxilary targets

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

I propose two means of finding such targets.

Evolved targets could be created by an evolutionary algorithm tasked with evolving some form of query string that creates features which are especially useful to train a small convolutional network to predict the final value of the respective game. A likely problem with this approach however might be the possibly substantial cost of this evolutionary process.

To reduce the additional computational cost my second proposal to find regulization targets is to use the internal features the network learns. These are learnt anyway and are thus free. 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. It might be worthwhile to experiment with both options: Adding $\mathbb{H}_{future}$ before $\mathbb{F}$ or after $\mathbb{F}$. Another avenue of experiments 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.

### 4.3.4 Using the self-playing phase as an evolutionary process

There are many hyperparameters involved in AlphaZero which have to be tuned. Due to the general nature of how to algorithm is designed around the network learning to imitate the MCTS results one can argue that the goal of nearly all hyperparameters in AlphaZero should be to increase playing strength of the MCTS. Searching optimal values for the hyperparameters however is extremely expensive and it might be that depending on the learning progress of the network the best hyperparameters are not actually constant, but shift during training.

I propose to let multiple instances of players play games against each other during self play, with small differences in their hyperparameters. The generated games should still be valid as training data, but as an additional benefit, at no computational cost, the results of the games can be used to judge which hyperparameters appear to provide superior playing strength. After some number of games are played between the different players the worst players can be eliminated and new players are introduced to find even better hyperparameters. The relative results of the players could be tracked using a rating system between the players such as Elo or Glicko.

This allows the learning process to continuously pick the best hyperparameters and also adjust them if necessary after some training process has been made.

# References

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

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

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

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

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

[6] 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.

[7] Matej Guid and Ivan Bratko. Using heuristic-search based engines for estimating human skill at chess. *ICGA journal*, 34(2):71–81, 2011.

[8] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, et al. Searching for mobilenetv3. *arXiv preprint arXiv:1905.02244*, 2019.

[9] Jie Hu, Li Shen, Samuel Albanie, Gang Sun, and Andrea Vedaldi. Gather-excite: Exploiting feature context in convolutional neural networks. *CoRR*, abs/1810.12348, 2018.

[10] 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.

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

[12] 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.

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

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.