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, 22.7.2020

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 [25], 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 [21] 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. [21] 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. The result of the episode is backpropagated upwards in the tree. 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 [27] is the application of AlphaGoZero [28] to games other than Go. AlphaGoZero is a significant simplification of AlphaGo [26]. AlphaGo is the first program to reach superhuman performance in the 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 [22].

2.2.1 The AlphaZero algorithm

The AlphaZero algorithm [27] 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 [20]. 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 visists 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.

Unlike plain UCT, AlphaZero does not play out entire episodes till terminal game states, but rather calls the network whenever a move is mode 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.

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 pratice this appears to hold, as learning is quite stable even 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 4.2.3 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 [19], 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 [30] 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, [30] 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 [23] 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 [30] 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. [30] 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 [30], 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 Evaluated novel improvements

In this section I preset my proposals on possible improvements to AlphaZero, experimental results on my proposals will be presented in section 4

3.1 Network modifications

Research on artifical neural networks has advanced since the release of AlphaZero in 2018. Some of these advances might be easy to apply to the network structure of AlphaZero and yield likely small but certain improvements.

To get started novel improvements to AlphaZero I want to investigate the application of Gather-Excite elements in the network [18]. Gather-Excite elements are a development of Squeeze-and-excitation elements by the same authors, which previous work has already shown can improve performance of AlphaZero. Similar to Squeeze-and-excitation elements, Gather-Excite elements have the goal to make the global feature context more available to local decisions made in the convolutional layers of the network. Since the global situation of a game is often very important to local board decisions, it seems a reasonable to expect this to further help AlphaZero.

A more far fetched idea is to change the network structure to resemble that of the especially resource efficient mobilenetv3 [17].

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 [29], [30], [23].

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

3.3 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.

3.4 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. Experiments on hyperparameter search show that these MCTS-parameters have a noteable impact on much the MCTS can improve the playing strength of the network alone. Better parameters therefore 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 Glicko. 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. This should allow to search the hyperparameter space of the MCTS search very efficiently at no substantial additional cost.

4 Experiments

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...

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.

4.1 Testing on Connect4

To reduce costs of experiments I am restricting myself to experiments with the game Connect4. Unlike Go, Connect4 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.

4.1.1 Generating Connect4 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 [24][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 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 Connect4 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. [24] 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 contract [24][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 choises which noticeable reduced the challenge their dataset posed, possibly explaining differences in accuarcy values between my work and theirs.

4.2 Baselines

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 best performance possible without any novel proposals.

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.

Results from supervised training are used to decide what size of network to use for AlphaZero experiments.

4.2.1 Supervised training

TODO the win-prediction results of supervised training are substantially worse than AlphaZero training and show clear signs of massive overfitting. Can that be fixed somehow?

4.2.2 AlphaZero implementation

4.2.3 Extended AlphaZero

4.3 Results on novel improvements

4.3.1 Network modifications

4.3.2 Playing games as trees

4.3.3 Automatic auxilary features

4.3.4 Evolutionary hyperparameters

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] 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.

[18] 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.

[19] 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.

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

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

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

[23] 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.

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

[25] 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.

[26] 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.

[27] 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.

[28] 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.

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

[30] 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.