DEGREE PROJECT IN COMPUTER ENGINEERING,
FIRST CYCLE, 15 CREDITS
STOCKHOLM, *SWEDEN 2019*

AlphaZero to Alpha Hero

A pre-study on Additional Tree Sampling within Self-Play Reinforcement Learning

FREDRIK CARLSSON

JOEY ÖHMAN

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Bachelor in Computer Science
Date: June 5, 2019
Supervisor: Jörg Conradt
Examiner: Örjan Ekeberg
Swedish title: Från AlphaZero till alfahjälte - En förstudie om inklusion av additionella trädobservationer i straffinlärning

Abstract

In self-play reinforcement learning an agent plays games against itself and with the help of hindsight and retrospection improves its policy over time. Using this premise, AlphaZero famously managed to become the strongest known Go, Shogi, and Chess entity by training a deep neural network from data collected solely from self-play. AlphaZero couples this deep neural network with a Monte Carlo Tree Search algorithm that drastically improves the networks initial policy and state evaluation. When training AlphaZero relies on the final outcome of the game for the generation of training labels. By altering the learning target to instead make use of the improved state evaluation acquired after the tree search, the creation of training labels for states exclusively visited by tree search becomes possible. We propose the extension of Additional Tree Sampling that exploits the change of learning target and provide theoretical arguments and counter-arguments for the validity of this approach. Further, an empirical analysis is performed on the game Connect Four, which harbors results that justifies the change in learning target. The altered learning target seems to have no negative impact on the final player strength nor on the behavior of the learning algorithm over time. Based on these positive results we encourage further research of Additional Tree Sampling in order to validify or reject the usefulness of this method.

Sammanfattning

I självspelande straffinlärning spelar en agent mot sig själv. Med hjälp av sofistikerade algoritmer och tillbakablickande kan agenten lära sig en bra policy över tid. Denna metod har gjort AlphaZero till världens starkaste spelare i Go, Shogi, och Schack genom att träna ett djupt neuralt nätverk med data samlat enbart från självspel. AlphaZero kombinerar detta djupa neurala nätverk med en Monte Carlo Tree Search-algoritm som kraftigt förstärker nätverkets evaluering av ett bräde. Originalversionen av AlphaZero genererar träningsdata med det slutgiltiga resultatet av ett spel som inlärningsmål. Genom att ändra detta inlärningsmål till resultatet av trädsöket istället, möjliggörs skapandet av träningsdata från bräden som enbart blivit upptäckta genom trädsök. Vi föreslår en utökning, Additional Tree Samling, som utnyttjar denna förändring av inlärningsmål. Detta följs av teoretiska argument för och emot denna utökning av AlphaZero. Vidare utförs en empirisk analys på spelet Fyra i Rad som styrker faktumet att modifieringen av inlärningsmål är rimligt. Det förändrade inlärningsmålet visar inga tecken på att försämra den slutgiltiga spelarens skicklighet eller inlärningsalgoritmens beteende under träning. Vi uppmuntrar, baserat på dessa positiva resultat, ytterligare forskning vad gäller Additional Tree Sampling, för att se huruvida denna metod skulle förändra AlphaZero.

Acknowledgements

We both feel a strong need to personally thank all of the people that have supported, helped and guided us throughout the work of this thesis. We are truly appreciative of all the good folks that have aided us along the way. Special thanks are in place for senior AI researcher Lars Rasmusson, that showed great interest in our problem and was to great help during many discussions, accumulating into many hours spent in front of a whiteboard. Jörg Conradt, our supervisor, aided us greatly by spending a lot of his time and energy to supply us with much of the needed hardware. His devotion and constant availability were equally commendable, rarely has a humanoid been witnessed to respond so quickly to emails. Finally, a warm thank you is reserved for Sverker Jansson and RISE-SICS, giving us access to their offices and supplying us with many much-needed GPU's.

Contents

1 Introduction

Thanks to the predefined rules and limited state space, games have throughout history served as a testbed for Artificial Intelligence. The bounded environmental complexity and clear goal often give a good indication of how well a particular agent is performing and acts as a metric of success. Additionally, the artificial nature of most games has the benefit of its ease to simulate, removing the need for hardware agents to interact within the real world. Historically most research within the field of game-related AI has been devoted towards solving classical 2-player board games, such as Chess, Backgammon and Go.

One recent milestone for AI in such domains is AlphaZero[1], which achieves superhuman level strength at Go, Chess, and Shogi by only playing against itself. The predecessor AlphaGo[2] was produced and optimized for the game Go and famously managed to defeat one of the world's best Go players, Lee Sedol. Although AlphaGo, unlike AlphaZero, bootstrapped from human knowledge, AlphaZero decisively outperformed all versions of AlphaGo. Perhaps even more impressive, AlphaZero also managed to defeat the previously strongest chess AI, Stockfish[3], after only 4 hours of self-play. Where Stockfish has been developed and hand-tuned by AI researches and chess grandmasters for several years.

During self-play AlphaZero performs a tree search, exploring possible future states before deciding what move to play. Whereafter making a move, it switches sides and repeats the procedure, playing as the opponent. As a game finishes, a policy label and a value label is created for every visited state, where the value label is taken from the final outcome of the game. By doing this, AlphaZero learns to predict the final outcome of a game from any given state. Since the game outcome is only applicable to states actually visited, this limits the number of training labels that can be created from a single game, excluding states only seen during the intrinsic tree search.

Changing the evaluation to instead target information collected during tree search, would break the dependency posed on label creation and would theoretically allow for the creation of additional tree labels. This thesis proposes the extension of "Additional Tree Sampling" and provides theoretical arguments to the benefits and viability of this extension. Further, an empirical analysis is performed on the viability of using evaluation labels that do not utilize the final outcome of the game, as this is needed for the introduction of additional tree samples.

To our knowledge, the extension of additional tree sampling and its effects are yet to be proposed and properly analyzed, and as of today, there exists very little documentation regarding the effects of altering the evaluation labels.

1.1 Problem Statement

AlphaZero currently relies on the final outcome of a game for the creation of evaluation labels. As this outcome is only present for states visited during that specific game, this enforces a strong dependency on what states are possible candidates for label creation. This among other factors, creates a requisite for exploring a vast number of games, as only so much information can be gathered from a single game. We propose an extension to the AlphaZero algorithm, whereby altering the evaluation label allows for the creation of additional training samples.

Theoretical arguments and counter-arguments are provided for utilizing additional tree samples. An empirical analysis is performed, providing insights into the effects caused by alternative evaluation labels. The non-theoretical analysis is realized by implementing the original algorithm, as well as the modification, allowing for comparison in player strength and overall domain generalization. Using this information we try to validate or reject the possibility of breaking the dependency set by creating labels from the final game outcome. This task can be boiled down too answering the following question.

> *"Using the AlphaZero algorithm on the game of Connect Four, how does altering the evaluation label from the final game outcome, to post-search state evaluation, affect player strength and general domain knowledge?"*

1.2 Scope

This thesis does not provide proof nor sufficient data, to completely validate or reject the proposed alterations. Rather, it theoretically proposes modifications to AlphaZero and then provides data and conclusions that validate the first premise of these modifications. This data is gathered from several runs of AlphaZero in a single domain, using different evaluation labels, and acts as the main evidence for the final conclusions.

AlphaGo and AlphaZero were developed by Google Deepmind, that unlike this thesis, do not suffer from severe constraints in time nor hardware. This issue is addressed by limiting the research project to the game Connect Four. Thus making the assumption that similar conclusions can be transferred to similar environments. Furthermore, these constraints bounds to what extent certain hyperparameters can be tuned and limits analysis regarding the possibly complex effects our modifications might pose on these hyperparameters.

1.3 Purpose

The motivation for mastering domains via self-play using deep reinforcement learning goes beyond board games. In domains where datasets are absent or too expensive, supervised learning is not feasible. Moreover, the information contained in a dataset defines an upper limit for the behavior of the agent. By allowing an agent to learn from its own experience without prior data and human intervention, superhuman performance can be achieved, allowing for new discoveries to be made.

Although AlphaZero achieves state of the art results and decisively outperforms previous approaches, a vast amount of computational resources are needed to benefit from this algorithm. Therefore, if there is a possibility to increase the amount of information extracted per game, this could help lower the number of games needed during self-play. Hopefully, our proposed extension could thereby help make AlphaZero a more viable approach on commodity hardware.

2 Background

A wide variety of different approaches has been deployed throughout history in pursuit of creating AI agents for games. A distinction to be made is that of algorithms utilizing any sort of machine learning, and those only executing predefined sets of instructions. Throughout history, there has been a clear bias towards non-learning algorithms, as these have been able to tackle bigger domains and generally outperformed its learning counterparts. Recent events have however turned the tides, and a clear favor towards machine learning based approaches can now be noted.

Two examples of the different approaches are TD-Gammon[4] and Deep Blue[5], where the former was set to play Backgammon and Deep Blue played Chess. TD-Gammon relied on self-play reinforcement learning, in juxtaposition to Deep Blue that used a pure Minimax[6] based approach deprived of any form of machine learning. Both of these managed to achieve top human-level play in their respective domains. With Deep Blue also transcending the human capabilities by famously managing to defeated Garry Kasparov, the at the time Chess World Champion.

Chess has a game state complexity of 10^{47}[7] and comparing this to the 10^{20} possible states in Backgammon[4], it can be argued that Chess is the harder game. The validity of this argument is however questionable as Backgammon incorporates stochastic elements, unlike Chess. Nonetheless, many people took to the success of Deep Blue as a clear indication that the brute force method was the more viable approach compared to machine learning.

The complexity of Chess is however dwarfed by the game of Go, which boasts an average branching factor of 250 and a state-space complexity of 10^{170}[8]. This intricate domain complexity effectively eliminates any possibility of mapping states to actions in a tabular manner, making the Minimax approach infeasible. Using intuition and human ingenuity, top human Go players have therefore historically always decisively outperformed the best AI. Considering this vast complexity, many AI researchers argued that it would

take many years until AI could outperform humans at the game of Go.

In Mars 2016, AlphaGo made a breakthrough for not only Go programs but also deep reinforcement learning, defeating one of the world's best Go players, Lee Sedol. Instead of using a brute-force method, AlphaGo learns to play Go by training a neural network, first by supervised learning followed by further improvements through self-play. This neural network is used to guide a tree search, allowing AlphaGo to intuitively disregard moves it seems unfit. As the quality of the network predictions increase, the tree search explores states with higher intelligence, gradually overcoming the boundaries set by the initial dataset used for supervised learning.

The successor AlphaGo-Zero[9] skips the supervised stage and starts by learning from self-play, boldly stating "Mastering the game of Go without human knowledge". As AlphaGo-Zero decisively defeated AlphaGo 100-0, the question was lifted if human knowledge runs the risk of plateauing progress rather than boosting it. AlphaZero then further polished the ideas set by AlphaGo-Zero, managing to not only defeat all previous versions of AlphaGo but mastered the games Chess and Shogi as well.

Before diving into a detailed description of AlphaZero, the reader should be acquainted with crucial concepts regarding this algorithm. The following sections aim to briefly cover these concepts, as well as why and how they are utilized. Finally giving a detailed description of the original implementation and the set of hyperparameters used.

2.1 Reinforcement Learning

The machine learning subfield that is reinforcement learning can loosely be defined as having an agent act within an environment, to learn an internal decision policy to maximizes the future sum of rewards. Depending on the task, these rewards can be either external from the environment or intrinsic as shown by Deepak Pathak et al.[10]. Either way, these rewards should in some way capture the essence of the desired problem, so that an optimal policy is also a solution to the given problem. Unlike supervised and unsupervised learning there often does not exist any predefined dataset on the given problem. Rather, the agent is tasked to explore the environment whilst adapting its policy in accordance with its new discoveries.

2.1.1 Exploration versus Exploitation

Reinforcement learning algorithms are often faced with the task of finding an optimal policy whilst dealing with the problem of incomplete information. In these cases where the agent only has access to partial environmental information, the agent is continuously forced to make decisions regarding exploration and exploitation[11]. Exploitation refers to executing actions that are known to yield good results and exploring is when the agent instead prefers actions that maximize the gain of new information.

Finding a good balance between exploration-exploitation is essential, as time spent exploring non-optimal paths is by definition non-optimal. The same argument also holds for time spent exploiting suboptimal actions. It is therefore up to the algorithm to reinforce strategies yielding high rewards, but balancing this with the potential benefit of exploring new strategies.

The exploration-exploitation tradeoff is present in AlphaZero on several levels. AlphaZero tackles this when deciding what moves to play, but also internally each time it performs its guided tree search. Finding the right balance of exploration-exploitation is critical for AlphaZero's learning rate and requires the tuning of several parameters.

2.1.2 Model-based Reinforcement Learning

AlphaZero belongs to the sub-category known as Model-Based reinforcement learning algorithms[12]. In model-based reinforcement learning the agent has access to an internal world model that it can use for intrinsic foresight. This world model can either be given to the agent at the start or learned through experience and exploration of the environment.

AlphaZero is given a perfect world model in the form of hardcoded game rules. This is used to check if a state is terminal and allows for an explicit move simulation, where AlphaZero internally takes a state-action tuple (s, a) and outputs the resulting state s'. The inclusion of this world model allows AlphaZero to perform a tree search that improves the decision policy of what moves to play.

2.1.3 State Evaluation

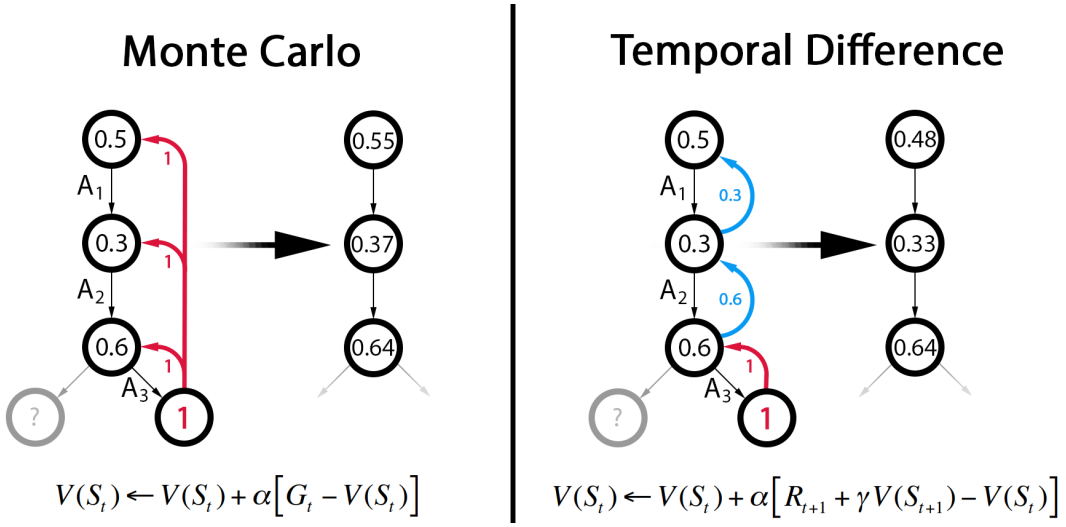

Figure 2.1: The difference between updating a tabular Value function using Temporal Difference and Monte Carlo, using learning rate $\alpha = 0.1$ and discount factor $\gamma = 1$.

Many reinforcement learning methods incorporate a value function $V(s)$. This function, when given a state and policy, outputs the expected sum of future rewards collected from s, following policy π. This is used to guide the agent as actions correlated with states having a higher sum of rewards is by definition more desirable. Further, the possession of a perfect value function allows for perfect play by greedily performing actions that maximize the value function.

There exist different ways of learning the value function, but an often used method is to sample data from the domain and gradually tune V so that it approximates the distribution of the sampled data. This can be achieved by iteratively decreasing the prediction error between the value function and the collected data. Different reinforcement learning methods perform sampling in different ways and this can impact the convergence rate. Two well-known approaches to sampling are the Monte Carlo and Temporal Difference methods[13], as displayed in figure 2.1.

The straightforward Monte Carlo method is performed by sampling the final sum of rewards acquired at the end of a session. Although the law of big numbers shows that this method converges to the true evaluation function given sufficient samples, it introduces some apparent restrictions. Since samples are only achieved at the end, we only extract one sample from every full session and are therefore bound to only update our model

after completing a full session. Additionally, the further away our collected reward is from the initial state, the lower the accuracy of the collected sample.

TD-Learning, in contrast, relies on the concept of bootstrapping and can be summarized to "estimating from an estimation". Instead of waiting until the end of a session, a sample is acquired as soon as we get to the next state and have access to a new value estimation. The difference between the value estimation acquired from $V(s_t)$ and $V(s_{t+1})$ is referred to as the TD-error. By minimizing the TD-error we essentially update our previous value estimation to approximate our newer estimation. This approach allows us to gather samples after every state transaction and update our model more frequently. However, using our new estimation as a learning target entails that the quality of our learning is directly coupled to the quality of our current estimation. Thus for learning the true value function, predictions from the final outcome are still needed and time is required for these to properly propagate into earlier states.

Since Monte Carlo and Temporal Difference can be proved to converge to the true value function[13], the choice of sampling method is domain dependent. The original implementation of AlphaZero utilizes the Monte Carlo method for updating its value function. Additional Tree Sampling requires an alternative learning target, and the target proposed is more reminiscent of the ideas concerning Temporal Difference learning.

2.2 Deep Learning

2.2.1 Artificial Neural Networks

The Artificial Neural Network is a machine learning model known for its ability to learn complex patterns and its strong generalization abilities[14]. These networks are formed by neurons connected via weighted edges which process data by scalar multiplication. Every neuron sums its incoming data and applies a nonlinear activation function e.g ReLU, Sigmoid, TanH. These nonlinear activation functions are a necessity, as otherwise no matter how many neurons, the approximated function is bound to be linear.

Most networks are formed by an input layer followed by several stacked layers referred to as hidden layers, that sequentially perform computations, propagating the result to the next layer. This is continued throughout the network until arriving at the last layer, named the output layer, where the computational result is interpreted as the model's output. Figure 2.2 demonstrates a rudimentary example of a network with two hidden

layers consisting of 9 and 4 neurons respectively. In mathematical terms, the neural network can be described as a nonlinear function, mapping inputs to outputs, processing data of arbitrary shapes.

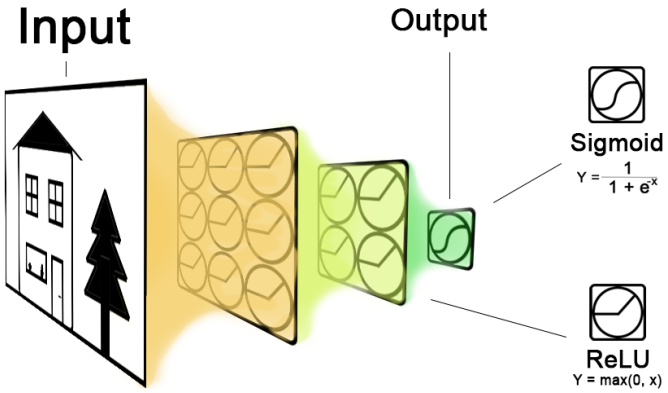

Figure 2.2: A simple neural network taking an image as input and propagating the computation through its two hidden layers, until reaching the output layer. The hidden layers use the ReLU activation function whilst the output layer uses a Sigmoid activation function.

There exists an infinite number of ways a neural network can be formed by stacking and connecting layers. Although the universal function approximation theorem[15] states that a neural network utilizing a single hidden layer can approximate any possible continuous function, it is very common to stack a multitude of layers, creating a deep neural network. Empirically this has been shown to work much better[16], and its success has been theorized to be partially caused by the hierarchical structure giving the layers the ability to share common subexpressions. Unsurprisingly, similar architectures are used in AlphaZero.

As deep neural networks are often used for complex function approximation, it is not uncommon for them to contain several million learnable parameters in the form of weights and bias values. The expressive power of a machine learning model is strongly related to its number of parameters, causing deep neural networks to inherit a significant capability for overfitting. However, Zihang et al. have proved that this vast capability by itself, does not imply that the model will overfit[17]. This contradicts conventional beliefs and they conclude we have yet to discover exactly why Deep Neural Networks tend to generalize so well. Whilst certain aspects of the neural networks remain undiscovered, its ability to approximate complex functions is evident.

AlphaZero strives to train a deep neural network that captures the intricate game intuition needed for complex games like Go, Shogi, and Chess. A game state representation s is given as input and the network outputs both the move probabilities vector $\boldsymbol{p}$ and a scalar value v representing the estimated win probability for the current player.

2.2.2 Training Neural Networks

In supervised learning there exists a dataset that contains a label output for every input sample. The most common way to train a network when having access to such a dataset, is using a gradient descent based approach[18]. This is achieved by calculating the gradient for the output error with respect to the weights, this can be performed on individual samples, batches of data or the complete dataset. Once the gradient to one or several samples has been calculated, a step in weight space is taken in the direction minimizing the error. This way, the network weights are gradually altered to increase the accuracy of the network.

Most often, gradient descent methods iteratively calculate the derivative of the average prediction error acquired from batches of the dataset, with respect to all the parameters in the model. This is known as batch gradient descent and implies that every time the parameters in the network are tuned to fit a certain target, previous adjustments are also affected. Intuitively, this can be seen as an unnecessary approximation and one might be tempted to instead include the whole dataset in every gradient calculation. Recent research, however, points towards the fact that using too large batches decreases generalization[17], as this increases the chance of converging into a sharp optimum. Thus using a small batch-size, is one of the best regularization methods currently available.

2.2.3 Skewed Datasets & Data Augmentation

As deep learning models require a vast number of training samples, data augmentation is often used to generate additional synthetic data[19]. Strong parallels can be drawn between Additional Tree Sampling and data augmentation. The reader should, therefore, be introduced to the original concept of data augmentation and its usual applications and problems.

Independent of the method used for training the network, it is important that the true goal is to find a set of parameters that not only perform well on the training data but generalizes well over samples not yet encountered. Whilst there are many hyperparameters

which affect the learning performance in different ways, the data itself is what carries the nuances of the domain and defines what the network will learn. The data thus need to provide a balanced representation of the domain in order for the network to be able to generalize.

Skewed datasets occur when there is an imbalance between samples correlated to different labels. This can be very troublesome for the gradient-based optimizers and a too big imbalance might lead to complete neglection of underrepresented labels. Skewed datasets are best solved by balancing the number of samples available for each label. In cases where collecting new data is not possible, one can sometimes make use of different forms of data augmentation. New data is then synthetically generated from previous data in the pursuit of providing a more complete and robust representation of the domain.

A simple example would be that of computer vision, where one might synthesize additional data by rotating, flipping or applying other transformations to the already collected images, as done in figure 2.3. As the network tries to learn a function from feature space to label space, increasing the number of known mappings can stabilize the learning process. These newly generated data points might be of lesser quality than the originally collect data, but nonetheless increasing the dataset by augmented samples has been shown to improve generalization[20].

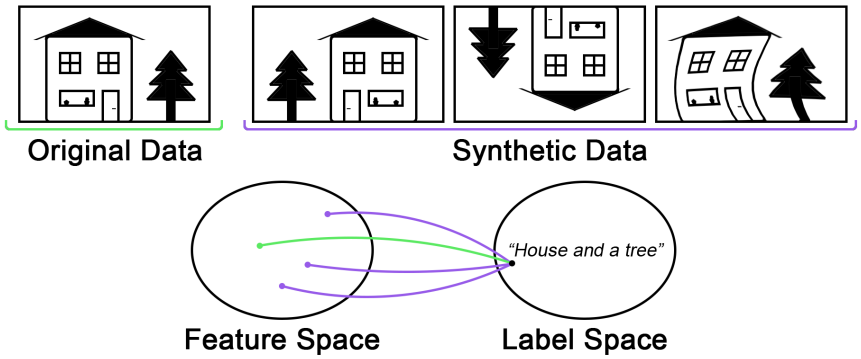

Figure 2.3: Synthesized data might be generated by applying simple transformations to originally collected data.

The creation of synthesized data requires domain knowledge since there exist no valid data transformations that are applicable in all domains. Earlier versions of AlphaGo made use of the symmetrical nature of Go by creating additional rotated and flipped training samples. AlphaZero later disregarded this domain knowledge as games like Chess and Shogi are non-symmetrical, thus abandoning the use of synthesized data.

2.2.4 Summary

In conclusion, neural networks are models capable of learning an accurate approximation of any continuous function f, given sufficient data. The neural network can be formulated as a function f_θ with parameters θ. In AlphaZero this function is a mapping from states s to policy-value tuples $(\boldsymbol{p}, v)$, as shown in equation 1 below.

$$(\boldsymbol{p}, v) = f_\theta(s) \tag{1}$$

Using conventional neural network methods, the AlphaZero network predicts the utility of states and the corresponding possible moves.

2.3 Monte Carlo Tree Search

The field of tree search has historically been tightly coupled with AI and has been used on numerous occasions to successfully compete with humans in games. Tree search in well-known deterministic environments allows for simulations of actions with full certainty, enabling an informed evaluation of each possible action. This further allows computational power to be utilized and should be seen as a powerful knowledge improvement operator.

Unlike the Minimax approach used by Deep Blue that explores every possible outcome to a certain depth, Monte Carlo Tree Search(MCTS)[21] treats the search problem as an exploration-exploitation problem. MCTS balances the allocation of resources to promising paths and exploration of paths potentially yielding needed information. By keeping track of every node's visit count and mean state value, the algorithm decides what path to visit each iteration based on previous information.

MCTS is performed in an iterative manner where every iteration starts from the root node and traverses the tree until encountering a leaf node, upon where it expands the tree if the leaf node represents a non-terminal state. Every iteration can be divided into four distinct stages, Selection, Evaluation, Expansion and Backpropagation. Following is a detailed description of these four distinct stages that together form the MCTS algorithm, as shown in algorithm 1.

Algorithm 1 Bootstrapped Monte Carlo Tree Search

```
1:  for number of iterations do                                   ┐
2:      node ← root                                               │ Selection
3:      while node.visits > 0 and ¬node.terminal do                │
4:          node ← argmaxPuct(node.children)                      ┘
5:
6:      if node.visits = 0 then                                   ┐
7:          (node.terminal, z) ← terminalEvaluation(node)         │
8:          if node.terminal then                                 │ Evaluation
9:              node.value ← z                                    │ and
10:         else                                                  │ Expansion
11:             (p, v) = f_θ(s)                                   │
12:             node.value ← v                                    │
13:             node.children ← expand(node, p)                   │
14:     value ← node.value                                        ┘
15:
16:     node.visits ← node.visits + 1                             ┐
17:     while node ≠ root do                                      │ Back-
18:         node ← node.parent                                    │ propagation
19:         node.visits ← node.visits + 1                         │
20:         node.value ← node.value + value                       ┘
21:
22: π ← normalizedPolicy(root.children)
23: v' ← root.score/root.visits
```

2.3.1 Selection

Starting from the root, the tree is iteratively traversed until a leaf node is encountered (row 3 in algorithm 1). A leaf node is defined by either containing a terminal game state or having a visit count of zero. At every stage of the traversal, the node coupled with the action a_t is selected for further traversal. Where a_t is calculated as shown in equation 2.

$$a_t = \underset{a}{\operatorname{argmax}} \left(Q(s,a) + U(s,a)\right) \tag{2}$$

$Q(s,a)$ corresponds to the mean state value v' for the node reached by performing action a in state s. $U(s,a)$ is derived from the Polynomial Upper Confidence Trees (PUCT) formula:

$$U(s,a) = c_{puct} P(s,a) \frac{\sqrt{N(s)}}{N(s')+1} \tag{3}$$

2.3.2 Evaluation

Upon encountering a leaf node its state value is acquired by either network evaluation or the predefined game rules. If the node selected holds a terminal state, the state value is determined by the game rules (row 9 in algorithm 1). If the state is non-terminal, the network is used to predict $\boldsymbol{p}$ and v (row 11), where v is propagated as the state value and $\boldsymbol{p}$ is stored to further guide selection in accordance with the PUCT formula.

2.3.3 Expansion

If the selected node is non-terminal the tree is expanded by generating new nodes for all attainable children from the selected node (row 13). This is achieved by simulating all legal moves available from the selected node where these newly generated nodes are all initialized to have the standard values and properties.

2.3.4 Backpropagation

When the state value of the selected node has been acquired, it is propagated back up the tree until it reaches the root. Thereby incrementing the mean state value of every node that was passed during the selection phase, including the root, and incrementing their visit count. This allows the root node to aggregate the information of the entire tree, resulting in $\boldsymbol{\pi}$ and v', approximating the true action policy and game outcome respectively with increased accuracy for each search iteration.

2.3.5 Monte Carlo Tree Search As A Function

$$(\boldsymbol{\pi_t}, v'_t) = \alpha_{\theta_{t-1}}(s_t) \tag{4}$$

Equation 4 shows algorithm 1 as a function. The MCTS function $\alpha_{\theta_{t-1}}(s_t)$ uses the network parameters θ_{t-1} from the previous time step. It takes the current state s_t as input and returns the policy-value tuple $(\boldsymbol{\pi_t}, v'_t)$.

Monte Carlo Tree Search can thus be seen as an improvement upon the network evaluation function shown in equation 1. This policy-value improvement operator proves extremely powerful for not only self-play learning but boosting the final performance as well. Consequently, MCTS allows AlphaZero's performance to scale with computational resources.

2.4 AlphaGo & AlphaZero Overview

Instead of using a brute-force method, AlphaGo relies on a neural network-driven MCTS for its move selection. First, the network is trained to imitate human top players by supervised learning on recorded pro games. Whereupon converging to the level attainable by the dataset, further improvements are achieved by self-play. When playing against itself AlphaGo generates training samples by using the final outcome of a game and retrospectively deduces how to improve the policy and evaluation of the network. As learning via self-play progresses and the predictions by the network improve, the MCTS explores and evaluates states with higher intelligence, causing the moves performed to gradually get stronger. This cyclic relationship between the neural network and the MCTS is what in essence drives the self-play learning of AlphaGo.

AlphaZero differs from AlphaGo mainly by skipping the supervised stage, starting from zero with self-play and learns solely from its own experience. In a versus matchup AlphaZero was shown to decisively outperform all previous versions of AlphaGo. While AlphaGo was created for the sole purpose of playing Go, AlphaZero is a more general algorithm, removing the use of any domain-specific knowledge. This was demonstrated as AlphaZero mastered Chess and Shogi, using only information about the game rules. In 2017, AlphaZero managed to defeat the previously strongest chess AI, Stockfish, after only 4 hours of training.

Although other implementational details differ between AlphaGo and AlphaZero, such as network architecture and the amount of MCTS iterations during training. The impressive results of AlphaZero seem to indicate that the incorporation of human priors can hinder AI in the long run. In 2019, Richard Sutton announced that history shows that regardless of what field of AI, utilizing human knowledge is a short term solution that plateaus long term progress[22]. This remains a controversial topic, as far from all researchers agree to this statement, and Gary Marcus argues that the innate ability to perform tree search disqualifies AlphaZero from the statement "learning from tabula rasa"[23].

2.4.1 AlphaZero Algorithm

Following is a detailed description of the AlphaZero algorithm, but unless stated otherwise the described concepts apply to all previous versions of AlphaGo. To give the reader an estimation of what range certain hyperparameters fall into, provided settings and parameters are taken from the official AlphaZero paper[1]. Slight variations in specific parameters can be noted between different versions, but are neglectable to getting an intuitive understanding of the algorithm.

AlphaZero utilizes an asynchronous approach where Self-Play, Supervised Learning and a Replay Buffer all work in parallel. At the start, the network is randomly initialized and passed to several workers that proceed to perform self-play and start collecting game data. As a game is finished, the states visited are analyzed and new training labels are created and sent to the replay buffer. The replay buffer stores data from the 500,000 most recent games and essentially works as a FIFO-queue. Simultaneously a new version of the network is created by performing gradient descent on mini-batches uniformly sampled from the replay buffer.

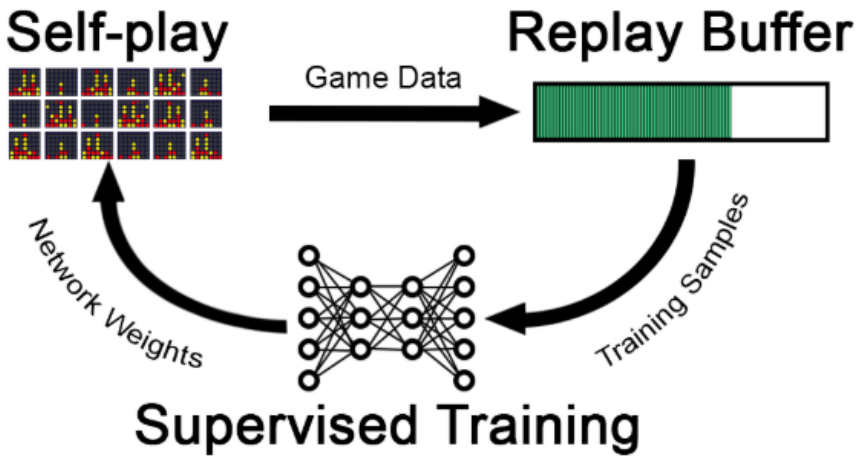

Figure 2.4: The Self-play Reinforcement Learning cycle used in AlphaZero.

After every 1,000 steps performed by gradient descent, AlphaGo and its variations perform an evaluation of the newly trained network. This is achieved by having the updated network play against the current best network and compare the win ratio. The new network is sent to the self-play workers only if it manages to achieve a 55% win ratio. AlphaZero removes this evaluation phase completely, continuously replacing the old network after every training step. Figure 2.4 depicts an overview of the learning cycle.

2.4.2 Self-Play

Using the current version of the network, AlphaZero continuously plays games against itself by guiding its move selection with its network driven MCTS. Upon encountering a new game state, 800 MCTS iterations are performed and a move is played by randomly sampling from the resulting visit vector. AlphaZero then switches sides and performs the same procedure but playing as the opponent. This is repeated until the game terminates and the game history is passed to the replay buffer, upon which a new game is started.

In order to encourage exploration and diversity of the games played, the following forms of randomness are incorporated within the self-play phase.

- Moves played are randomly selected by sampling from the resulting visit vector.
- Before a new search is started, Dirichlet noise is added to the predicted policy prior to that state.
- Previous versions of AlphaGo randomly picked one of 8 possible equivalent Go states, generated by rotating or mirroring the state, as it was passed to be evaluated by the network. However, as not all games can make use of such transformations, this was removed in AlphaZero.

These random elements allow for the exploration of new states whilst still being heavily biased towards moves favored by the network. Without this, every game played would be identical, causing self-improvement to stagnate and network predictions to rapidly deteriorate.

2.4.3 Generating Training Samples

As a game is terminated, new training samples are generated from all states visited during gameplay. For every state, an evaluation label and a policy label is created. Policy labels are given by normalizing the visit count vector yielded after the 800 search iterations, and the evaluation labels are set to the final reward of the game.

Training on the policy label is an attempt to learn how to approximate the complete result acquired by the MCTS. This approximation helps guide future tree search by acting as a prior to what states to explore, implicitly trying to build upon previous knowledge. The policy is what allows AlphaZero to take on the role as its own teacher, avoiding the convergence performance to be restricted by the quality and amount of data available from external sources.

The effect of this evaluation label is that AlphaZero converges to being able to predict the final outcome of a game, given any state. This enhances the accuracy of which AlphaZero evaluates states and helps guide future tree search to exploit truly good states. Balanced by equation 3, this allows AlphaZero to steer the MCTS towards states yielding high state values and high policy predictions.

The effect of this evaluation label is that AlphaZero converges to being able to predict the final outcome of a game, given any state. This enhances the accuracy of which AlphaZero evaluates states and helps guide future tree search to exploit truly good states.

2.4.4 Replay Buffer

A replay buffer is applied, in order to not overfit to the most recent training samples. The replay buffer stores training labels generated from the most 500,000 recent games and forces AlphaZero to gradually improve its performance as encountered discoveries will remain relevant for training until completely phased out.

The size of the replay buffer determines how gradually and robust the learning of AlphaZero will be and is, in essence, a speed vs robustness tradeoff. A replay buffer too small results in unstable training whilst a replay buffer too large slows training down as old training labels linger too long. No exact explanation has been given to the size used in the papers released by the authors.

2.4.5 Supervised Training

Mini-Batch Gradient Descent with a momentum of 0.9 is applied to the network where every batch includes 2048 labels, uniformly sampled from the replay buffer. The loss function incorporates L2 regularization and divides the loss between the evaluation and policy output equally. Learning rate annealing was applied by introducing a novel learning rate schedule that decreased the learning rate over time.

2.4.6 Network Architecture

The network architecture does not only define the complexity limit of what the network is able to learn, but it also affects its learning efficiency. With the deep neural networks commonly used today, the problem of vanishing and exploding gradients[24] have become evident. To address this issue, AlphaZero uses a residual neural network architecture[25] to form a residual tower with skip-connections allowing the gradient to bypass layers. Consequently, the original gradient can propagate from the output layers to the early layers without losing important information. This is crucial as the

AlphaGo Zero network consists of 19 residual blocks, together with accumulating 40 parameterized layers which result in 24,000,000 learnable parameters[9].

2.5 Environment

When exploring AlphaZero an appropriate environment must be considered. The games Go, Chess and Shogi are previously explored by AlphaZero and would, therefore, be appropriate environments for introducing modifications. However, these are vastly convoluted domains and thus suboptimal for projects suffering from hardware constraints.

2.5.1 Connect Four

The game of Connect Four has a suitable complexity with a branching factor of 4 and a state-space complexity of 10^{13}[8]. Despite being less complex than Go, Connect Four is still too complex to solve with a naive brute force approach if one expects a reasonable play time. Making it an interesting and feasible target for AlphaZero executed on commodity hardware.

Connect Four has however been solved thoroughly by John Tromp in 1995[26], using an enhanced Minimax approach that computed for 5 years. This implementation utilized a list of enhancements such as Alpha-Beta Pruning, Transposition Tables and Iterative Deepening. The definitive outcome of this calculation was that the starting player can force a win within 22 moves.

Although resource intensive, the possibility of solving states completely entails that a true performance metric can be created. Moves performed can be compared to the optimal moves and a models generalization ability can be estimated. The combination of a sufficiently complex domain and having a solver available therefore motivates our choice of Connect Four as a testing environment.

3 Hypothesis

Our hypothesis states that an increase in the number of samples generated per game should lower the number of games required during training. Further, we propose that states encountered exclusively through tree search still hold viable information and are therefore candidates for the generation of training samples. This is fundamentally different than acquiring more data by playing additional games and should be thought of as enhancing data utilization from independent games. We refer to this as Additional Tree Sampling.

3.1 The Learning Target

As described earlier, AlphaZero utilizes the Monte Carlo method and uses the final outcome of the game as the learning target for the value function. The final outcome is strictly representative of the states visited during the game and thus needs to be altered if one wishes to create samples from states not in the direct path of the played game. Taking inspiration from Temporal Difference learning, an alternative learning signal would be the improved node value v' obtained from the MCTS.

Every MCTS iteration can be thought of as a value and policy improvement operator, adding more information to the estimation of the outcome in accordance with policy $\boldsymbol{\pi}$. In trying to minimize the error of the initially predicted v to v', the network essentially learns to approximate its final estimation after the search. This allows for a more accurate estimation in future evaluations. The viability of v' is due to be domain dependent, as an increase in intermediate rewards would give the MCTS access to more true information.

Using this approach, both policy and evaluation labels are generated via previous estimations, putting a bigger emphasis on the initial estimation. This could be a case of

concern and could potentially destabilize the learning progress, halting convergence. Therefore, in order to validate this modification, we provide results from an empirical study, comparing z and v' as the evaluation target.

3.2 Additional Tree Sampling

By changing the target for which AlphaZero trains its value function, it becomes possible to generate additional training samples from states exclusively explored with tree search. This section provides several arguments to why the method of Additional Tree Sampling would work.

It is to be noted that information found in the nodes exclusively visited by tree search is not wasted by AlphaZero. Rather, this information is propagated to the root node of the search and guides the MCTS, which in turn shapes the policy label. However, thinking that the information found in non-root nodes is strictly a subset of the information contained within the root node is incorrect. The root node will, by the nature of MCTS have the most visits and is a less noisy sample of the true value and policy functions. But the root node only maps its information to one state, expressing nothing about the value or policy of other states. With the possible exception that the most favored move by the policy should lead to a state harboring a value similar to the value attained at the root.

Comparing the method of Additional Tree Sampling to the approach of data augmentation, it would seem that the additional data can help stabilize the learning progress. Unlike synthesized data, the additional tree samples are not only a unique mapping between input and outputs, but also carries unique domain information. The same picture of a bird, but rotated might be a new mapping between input and outputs, but no essential domain information is added. This is not the case with Additional Tree Samples as these describe a distinctly new state, albeit with less reliable information. Additional Tree Sampling does not require any domain-specific knowledge, except the ability to perform tree search.

Since every node in the expanded search tree uniquely couples its collected information to its corresponding state, a perfect learning model should be able to utilize these mappings of states to data. Considering a simple case, where a tabular learning model was applied instead of a neural network, there would exist no reason to exclude this information. New information would then always be stored in the corresponding table cell if it was collected with a higher iteration number or updated sufficiently recent. Unsurpris-

ingly, a tabular approach is not perfect since they do not generalize nor scale sufficiently for complex domains. Instead, AlphaZero uses a deep neural network that generalizes far better but requires vast amounts of data.

3.3 Potential Issues

Deep Learning is a young field and there are yet many things unexplored or unanswered, making it hard to predict any outcome with certainty. A multitude of reasons and phenomena might counter the viability of this hypothesis. Following, are some proposed counter-arguments to the method of Additional Tree Sampling and a brief discussion for each of these concerns.

3.4 Quality Of Data

As a result of changing the evaluation target, the quality of a sample is directly proportional to the number of search iterations computed from that node. The average iteration number per sample, therefore, drops when including non-root nodes. This inclusion of less qualitative samples might make the overall learning signal too noisy, aggravating the learning process.

Deep neural networks have been shown very resilient to vast amounts of noise[27], and some studies even show that the introduction of noise can enhance learning[28]. The additional tree samples are also different than noise, as the information is still valid. Considering the noise resilience of deep neural networks it seems unlikely that the introduction of less qualitative samples should halt progress.

As the iteration number associated with each sample is available to the algorithm, trade-offs can be incorporated to further utilize the indicated quality for each sample. Allowing for conscious quality-quantity decisions to be made by implementing a threshold to exclude samples of insufficient quality. The tuning of such a threshold would, therefore, increase the control of the learning procedure. Further, one could extend the algorithm by making the gradient descent impact of every training sample proportional to its iteration number.

3.5 Skewing The dataset

The inclusion of additional tree samples from a single game might steer the dataset to become homogeneous, causing the network to overfit to certain paths thus reducing generalization. This would counter the potential gain by adding more samples, as more games would have to be played for gaining full domain knowledge. The relationship between hyperparameters such as replay buffer size, domain complexity and amount of MCTS iterations are yet to be properly documented, making it hard to make any qualitative assessments on this matter.

4 Method

In order to remove the innate randomness that comes with asynchronous parallel algorithms, the original algorithm was reworked to follow a sequential pattern. The implementation still divides the learning cycle into three distinct steps: self-play, replay buffer, and training(figure 2.4), but sequentially switches between these indefinitely. This allows for reproducibility and helps segregate the results for proper analysis.

After the network has been randomly initialized, it is passed to the self-play workers and the self-play phase is started. When training data from 3000 games have been collected and added to the replay buffer, the algorithm switches to perform supervised learning. Upon finish, the updated network is broadcasted to the self-play workers and the algorithm starts 3000 new games of self-play. This back and forth between collecting new training data using the latest network and updating the network accordingly is continued until the algorithm is manually terminated.

4.1 Self-Play

During self-play, 800 iterations of MCTS is deployed with an exploration constant c_{puct} set to 4 and Dirichlet noise added to the first prior of iteration 0, with the parameter $\alpha = 1$. As a state is passed to the network for evaluation, the network is given one of the two possible equivalent mirror states.

Game data is collected by 10 self-play workers with their own dedicated GPU, each playing 300 games in parallel. In order to improve GPU utilization, each worker collects states to be predicted from all its games and sends it to the GPU for evaluation. To further improve performance each worker keeps a prediction cache, storing previous states and their predicted values.

4.2 Replay Buffer

The replay buffer holds the 2,300,000 most recently collected data points. Every data point consists of a state s, value z (or v') and policy $\boldsymbol{\pi}$. As a new training session begins these samples are pre-processed and converted into supervised training samples, before being passed to the trainer.

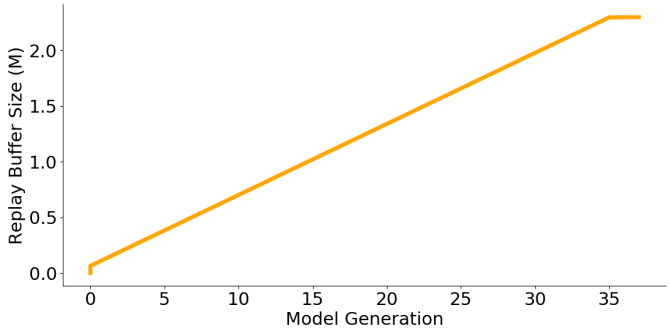

Figure 4.1: The size of the replay buffer increases over time during the first 35 generations.

Oracle Developers suggests that it is beneficial to phase out early training samples since these contain less valuable information[29]. These early samples run the risk of holding the learning procedure back, forcing the network to remember early state estimates and policies. To counter this, the buffer size is linearly increased for the first 35 model generations (see figure 4.1), until reaching its maximum size.

4.3 Pre-processing Data

To further improve the learning rate, pre-processing is performed by grouping data points with matching states. A single training sample is created for every group of states by calculating the average policy and evaluation label. This speeds up learning as the information contained in the replay buffer is condensed into fewer training samples.

4.4 Supervised Training

A new version of the network is created by applying stochastic gradient descent to the current network. Two full epochs are performed on every training sample generated

from the preprocessing phase, with the momentum parameter fixed to 0.9. The learning rate was fixed to 0.005 throughout the whole learning process.

4.5 Network Architecture

The network uses an almost identical architecture as that described in the original AlphaGo-Zero paper, although a bit smaller, see Appendix A. This decrease is applied in order to lower the computation time and scales the model size to the less complex domain. To further improve performance, float32 inference optimization was applied to the network, compressing the model into working with half of the original precision. This decrease in calculational precision entails a speedup when performing prediction but can result in less qualitative outputs. In order to counter this potential decrease in prediction accuracy, the filters in the respective policy and evaluation head were increased to 32.

The main body of the network consists of 12 residually stacked blocks, each containing 2 convolutional transformations with 128 filters. The main body then connects to a policy head and an evaluation head, resulting in a model containing 5,928,264 trainable parameters.

4.6 State Representation

The features of a board state are represented as a 7x6x5 image, where every layer of the image helps describe the existence of a particular feature (figure 4.2). The first three slices together denote the current state of every individual cell in the game grid and should be interpreted as together forming a one-hot encoding along the depth axis. The last two slices are used to indicate what player is next to make a move by setting all the values in a layer to one only if it is the corresponding players turn.

4.7 Experiment Setup

The majority of the code is written in Python 3.7, using Keras and Tensorflow. Elements of the self-play are further optimized by compiling it in C++, where the 5-slice

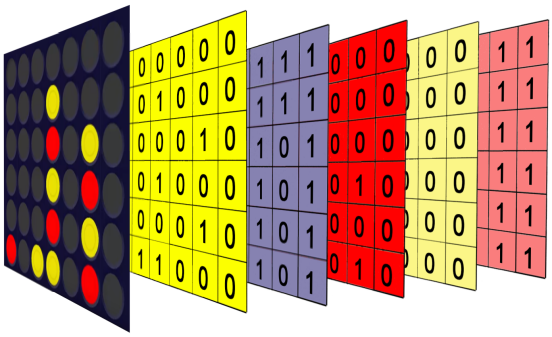

Figure 4.2: The 5 feature slices of a board where it is Red's turn to play.

state representation is replaced with a compact bitmap representation. Allowing bitwise operations to be utilized for efficient state computations, such as terminal evaluation and move simulation.

Additionally, the self-play phase scales well with additional hardware, supporting computations to be distributed across distinct hardware. A total of 48 CPU cores powered the 10 self-play workers, that each had their own dedicated GPU. (7 GTX 1080 ti, 3 GTX 1080). The supervised training phase shared 40 of these CPUs but was given its own dedicated GTX 1080 ti.

4.8 Evaluating Agents

During and after learning, it is beneficial to be able to evaluate how well the model is performing. In games not yet solved, such as Go or Chess, there lies a difficulty in providing such a metric. In fact, such metrics rely on previous works by measuring how well the trained model performs against it. On the contrary, as Connect Four is already solved, an accurate evaluation of a decision can be formed. Due to the presence of a solver, agents can be evaluated using data created by the solver.

4.8.1 Generalization Performance

Using the solver, a dataset has been generated, consisting of 10,000 unique states, each sampled from random play. For each state, the solver provides labels in the form of the theoretical winner according to optimal play, an action-policy representing the theoretical outcome corresponding to each move made in that state, and a measurement of how

many game tree nodes the solver had to visit in order to obtain the result.

If no pruning is done, the dataset will mostly contain states that are one optimal action from a terminal state. This skews the dataset towards artificially easy states, that rarely occur in natural gameplay. To achieve a more balanced set, 75% of these states are pruned. However, this set is still not optimal, as no further effort is put into making it closely resemble natural gameplay, nor are states where no action is incorrect to be excluded. In conclusion, it is used to provide a high accuracy estimation of the performance level of an agent.

The agents are evaluated by using their prediction of the outcome and optimal action of each state, then calculating the prediction accuracy against the dataset labels. These evaluations serve as an analysis of learning performance as well as a measurement of how well a final program performs on random states. However, this metric does not fully reflect player strength as a model can be capable of optimal play against an optimal opponent, whilst making mistakes when given states never reached in professional play. Consequently, this evaluation is considered a metric of domain generalization.

4.8.2 Player Strength

As the generalization strength does not necessarily reflect an agent's capability to win games, another measurement is required to complement the generalization performance. This metric is named Player Strength and is measured by having agents play each other a number of times. One severe problem that occurs when agents face each other in multiple games, is that the same game will be played repeatedly since they are deterministic by nature. To avoid this, randomness is incorporated into all agents' decision making during these matches. This way, agents are evaluated both in their general understanding of the domain and their ability to win games.

4.8.3 Agent Behavior

To further analyze the effect of modifying the learning target, the agents' behavior during self-play is measured throughout the training. The behavior measurements of interest are the average game length and the average number of terminal and non-terminal states encountered within search. This is important, as fundamental behavior differences could imply a reduced average performance. By measuring both performance and behavior, a detailed analysis of the viability of v' is possible.

5 Results

This chapter presents the results of our research, coupled with brief explanations.

5.1 General Domain Performance

First, figure 5.1 illustrates the low variance seen between independent learning sessions, spanning over the generations 0 to 30. Each generation corresponds to 3000 games of self-play, where the actions are informed by 800 MCTS iterations. To put it into perspective, Minimax accuracies of depth five and ten are added.

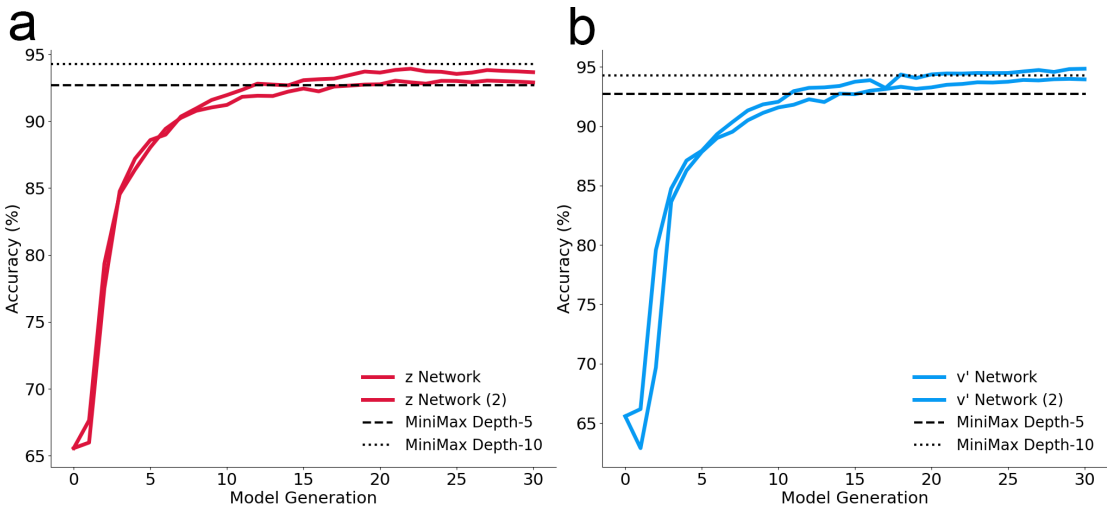

Figure 5.1: The action prediction accuracy of the deep neural networks trained using the two different state value learning targets, evaluated against the solved dataset each generation. **a,** the learning target z was used in two independent learning sessions. **b,** the learning target v' was used in two independent learning sessions.

The following figures only show results from the top performing candidates for z and v'. Figure 5.2 compares the learning performance of the deep neural networks trained using z and v'. Minimax Depth-10 reaches an accuracy of 94.28% whilst v' Network achieves an accuracy of 94.84%.

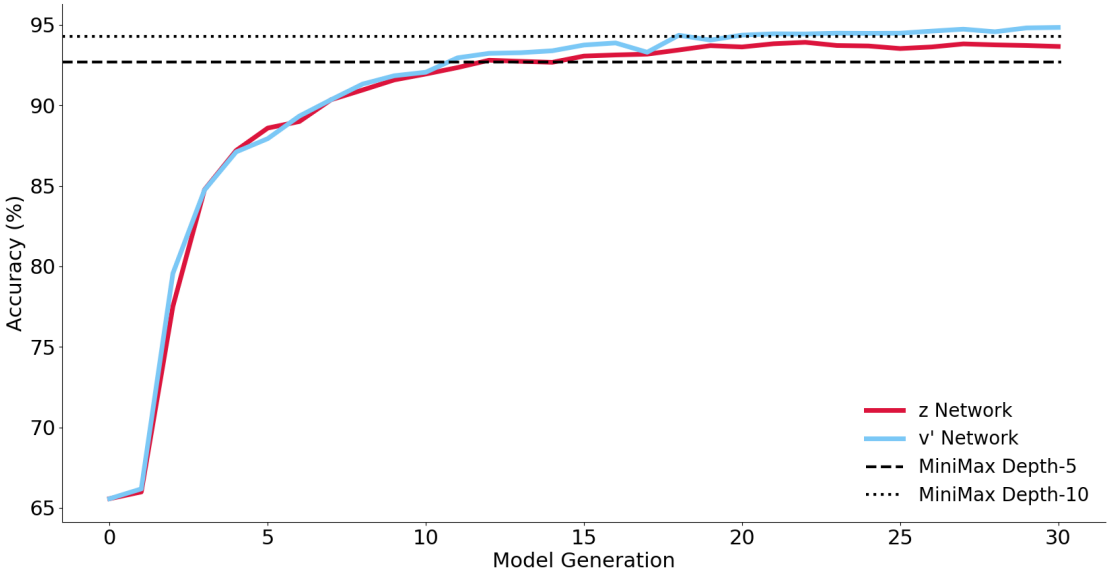

Figure 5.2: The action prediction accuracy of the deep neural networks trained using the two different state value learning targets, evaluated against the solved dataset.

Figure 5.3 depicts how MCTS steadily increases the accuracy throughout generations. Furthermore, it illustrates the strength of MCTS not only via the high accuracy numbers but also by showing that it surpasses the Minimax agents in the early stages of training. The v' variant with MCTS reaches an accuracy of 97.47%.

In contrast, figure 5.4 illustrates the error rates of the final models only, set side by side with their corresponding variant enhanced by MCTS. In addition to this, the second plot demonstrates the performance of v' as opposed to Minimax searching ten actions ahead, boosted by a heuristic. The solved dataset has been divided into subsets, representing states of increasing complexity. The first subset consists of states that are one optimal action away from a win, with rapid difficulty increments over the following subsets.

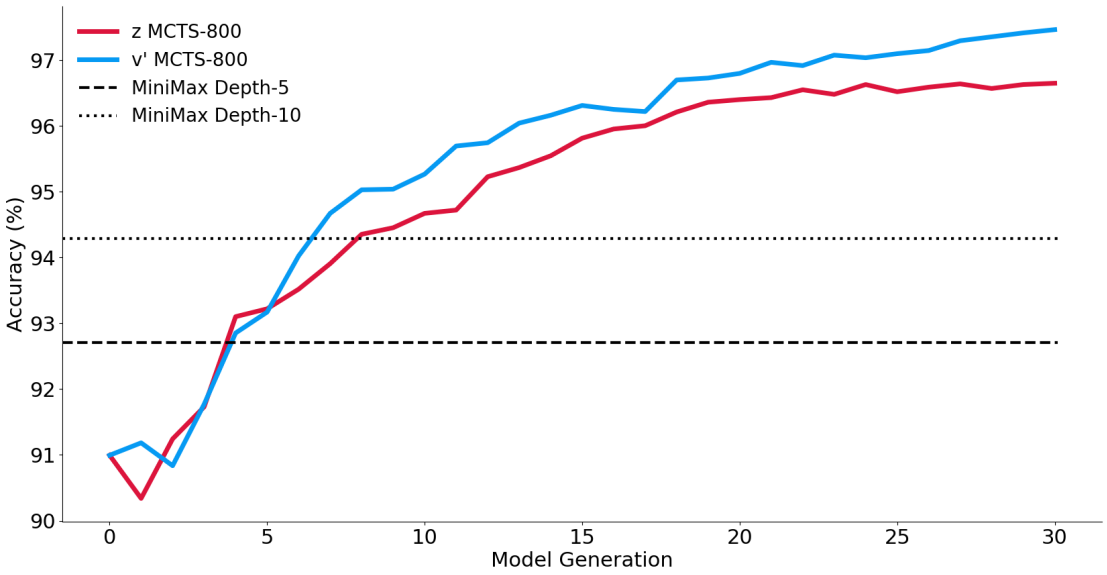

Figure 5.3: The action prediction accuracy of the deep neural networks coupled with MCTS, trained using the two different state value learning targets, evaluated against the solved dataset.

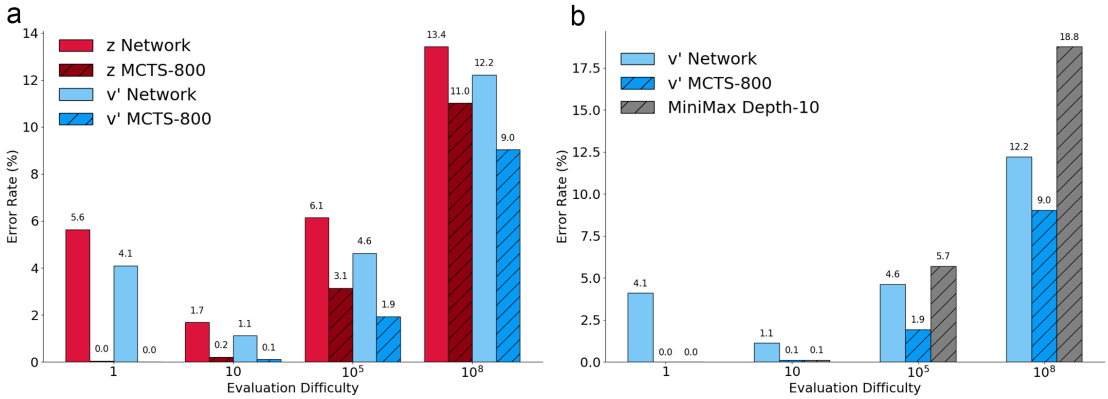

Figure 5.4: Both charts show the error rates for datasets of increasing difficulty. The x-axis labels are the average number of nodes the solver had to explore to solve the state. **a**, The error rates of both models (generation 30) along with their MCTS variants, evaluated against the solved dataset divided into four subsets of increasing difficulty. **b**, The error rates of the z model have been evicted, replaced by Minimax Depth-10.

5.2 Player Strength Comparison

Figure 5.5 shows how the agents $Minimax\ Depth$-10, z, and v' perform in different matchups. The Minimax algorithm was set to uniformly sample from moves that it found having equal action values. The network actions were sampled from a distribution over legal moves based on the policy predicted by the network. To not excessively cripple the network predictions, policy certainty was increased by performing $\hat{\boldsymbol{p}}^2$, which further decreases the probability of moves with low probability.

Player2 \ Player1	Z	V'	M-10
Z		76 - 12 - 12	7 - 8 - 85
V'	65 - 20 - 12		4 - 2 - 94
M-10	99 - 1 - 0	97 - 1 - 2	

Figure 5.5: The result of playing 100 games for each matchup, depicted in the order (P1Win - Draw - P2Win). The Minimax agent with search depth 10 uses a handcrafted heuristic for move decisions and z and v' sample moves from a distribution based on the raw network policy.

5.3 Behavior Over Generations

Figure 5.6 indicates that games get longer as training progresses. Moreover, it shows that there is no fundamental difference between z and v' regarding game length.

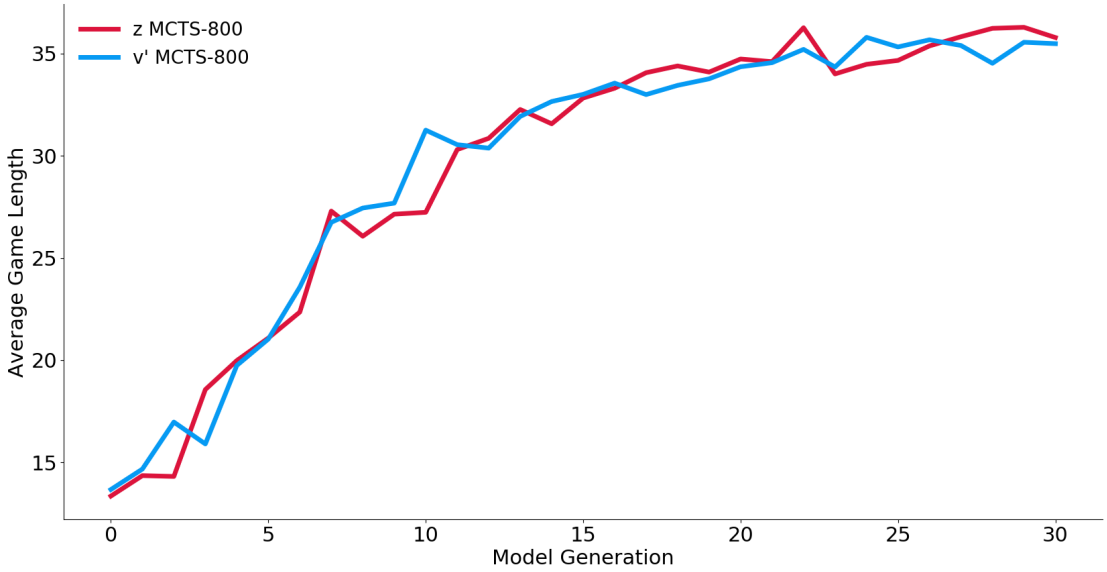

Figure 5.6: The average game length over 1000 games of self-play for each generation.

Figure 5.7 depicts the number of terminal states encountered on average in an MCTS search with 800 iterations. An increase can be seen for the first generations, then after around three generations it steadily decreases toward zero.

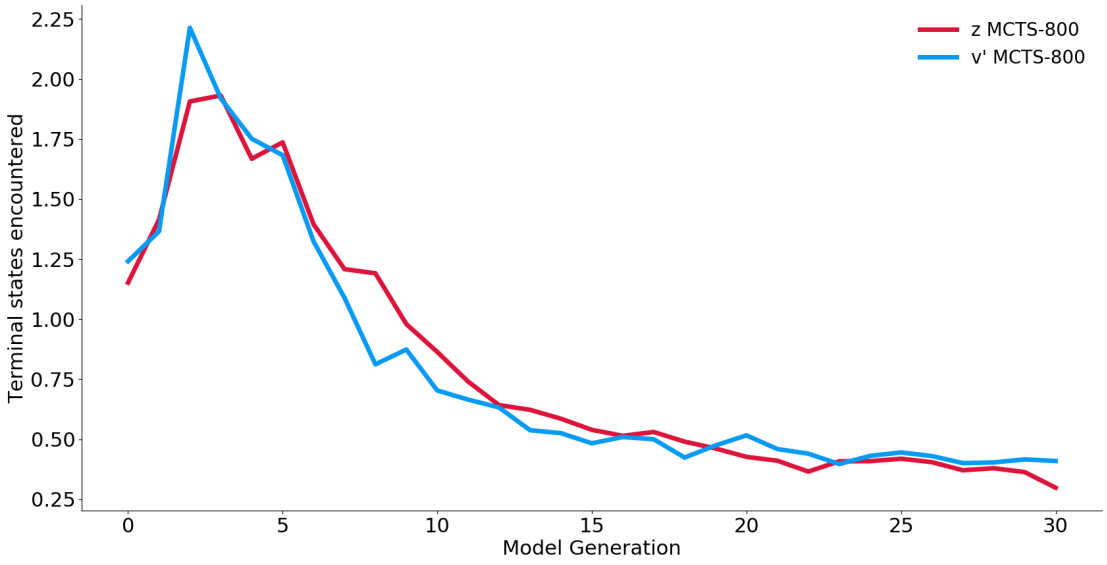

Figure 5.7: The average number of unique terminal states encountered in an MCTS search of 800 iterations, throughout the generations. For each generation, the average value is calculated over 1000 games played with one 800-iteration search per action.

Lastly, figure 5.8 shows that the number of unique states encountered in each search decreases through the generations. Diving from an average of roughly 350 to around 100 unique states. Furthermore, v' seems to encounter a slightly wider variety of states in the later stages of training.

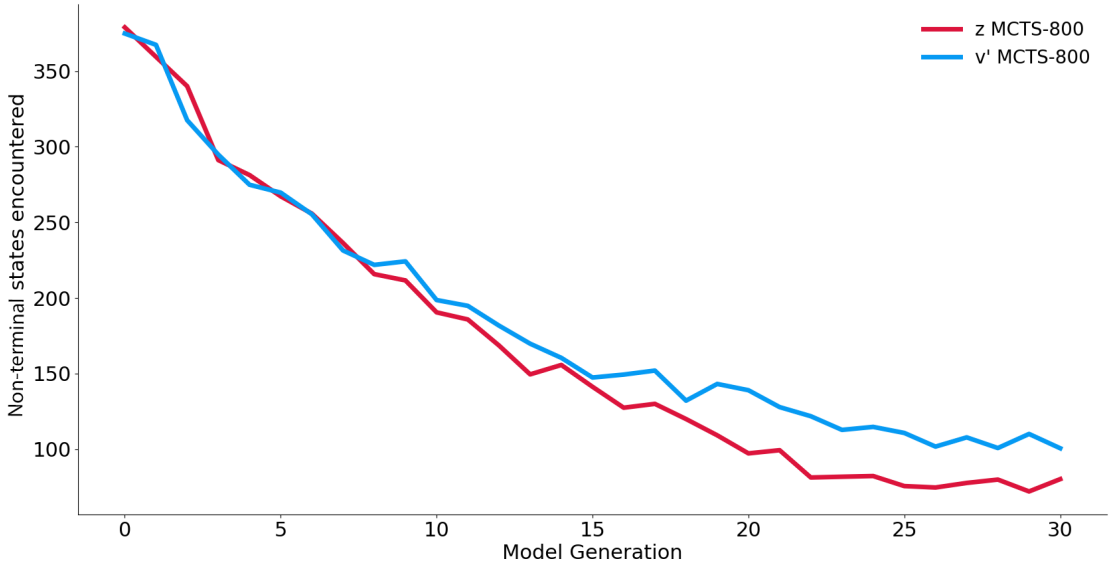

Figure 5.8: The average number of unique non-terminal states encountered in an MCTS search of 800 iterations, throughout the generations.

6 Discussion

Before diving into the discussions and conclusions, we emphasize that the elegance of the AlphaZero algorithm can give the impression that implementation and computation are equally simple. However, considering the computational burden that the algorithm entails, a parallel implementation is necessary. This introduces further complications as the program gets prone to concurrency bugs and other communication errors.

To help researchers willing to follow up on the concepts proposed here, we release all of our code and results, see Appendix C. Great efforts went into replicating the original algorithm as close as possible, with the help of not only the original papers, but also the blog post written by Oracle Developers[29]. Although rational results have been achieved, we can not be fully certain that our code does not introduce any bugs.

6.1 General Domain Performance

The variance between the different learning sessions is very small and the overall pattern is close to identical. Considering the costs and time needed for completing full 30 learning generations we, therefore, base our analyses and conclusions on the best v' and z. Further quantitative measurements are needed to fully validify this, but since no significant deviation was found during this study we still argue about the results found.

In figure 5.2 we can see that the learning performance of z and v' are similar, with v' performing ever so slightly better. This can be due to variance in training and should be tested in a number of training sessions before drawing any certain conclusions. However, what can be concluded from this is that v' is not to be rejected as a target training label.

From figure 5.4, it is evident that v' can be used as a feasible training target, showing

great resemblance to z. We see that that increasing difficulty implies increasing error rates, except for the easiest subset.

The easiest subset is speculated to contain states rarely encountered during self-play, but are included in the dataset only because we generated the dataset states from random gameplay. This would likely decrease AlphaZero's performance, as it is then tested on states which it has not been trained on. One could consider this being unfair since AlphaZero would probably not encounter these states during actual gameplay either. A possible solution, to create a more feasible dataset, could be to give the random agents generating the dataset, the ability to win the game when they are one action from a victory. However, we see that MCTS compensates for this flaw effectively. Additionally, figure 5.3 shows that the search acts as a general improvement operator throughout the generations.

6.2 Player Strength Comparison

Looking at figure 5.5, it is notable that although the Minimax algorithm was set to only select randomly from its top moves, it was decisively outperformed. It is unclear how much randomness is contributed to the network predictions. But unlike the Minimax algorithm, this randomness allows for selection of suboptimal moves.

Since Minimax achieves perfect predictions on the easier states (figure 5.4), it achieves a quite high generalization score. However, considering that v' clearly outperforms Minimax despite scoring far worse on the easier states, it is clear that overall generalization does not fully reflect player strength.

6.3 Behavior Over Generations

To gain a deeper understanding of how the agents with different learning targets learn, we analyzed how the behavior changes throughout the generations. This is particularly important to investigate before concluding whether v' is viable. Not only is it interesting to see if they vary in behavior, but also if one would prove to be more stable.

Figure 5.6 shows that the average game length is similar for both v' and z over generations. Moreover, figures 5.7 and 5.8 also seem to indicate that the behaviors of the two variants are close to equivalent. We see that the number of unique terminal states

visited are roughly the same. Additionally, it seems to be true for the number of non-terminal states as well, with v' exploring a slightly wider variety of states. This could be correlated with the performance as we saw a similar advantage for v' in the learning curves (figure 5.2 and 5.3). However, it is difficult to draw any major conclusions from the fact that v' achieved slightly higher performance since we have not made a statistically significant number of experiments. While this may be true, what can once again be concluded is that nothing points toward v' not being a viable learning target.

6.4 Limitations

The biggest limitations of this study are that it only explores a single domain and provides a small number of tests for that domain. The lack of quantitative data makes any definitive conclusion doubtful. Although the few collected tests displayed little deviation regarding the convergence rate, more tests are needed for assurance.

Limitations in hardware and time made exploration of additional domains impossible as training an agent for 30 generations took roughly 16-hours, using 10 GPU-workers. A majority of these GPUs were only attainable for a limited amount of time, forcing us to execute only a handful of qualitative experiments. The vast amount of possible hyperparameter settings forced us to devote large portions of the granted GPU time to an initial hyperparameter search.

As we are yet to run tests on different hyperparameter combinations, much is left unanswered. Although the used parameters were selected as they seemed to yield good results for the original method, there is no assurance that these are the best combination. The fact that v' outperformed z on these parameters does therefore not necessarily generalize. A possibility exists that the selected parameters happened to suit v' more and that another set of parameters would reverse the outcome.

6.5 Future Research

Since we have only explored the game Connect Four, it would be interesting to see if similar results would be achieved in other games. It would be particularly interesting to test v' in Go and Chess, to verify our thoughts on how the sparsity of rewards and using v' correlates. In addition to this, it would be interesting to see how v' performs

in a domain with a complexity similar to Connect Four, but with sparse rewards, e.g. Othello.

Moreover, as both z and v' seem to generate strong players, it could be beneficial to combine the two in different domains. For instance, taking the mean of the two or a falloff as done in the blog post by Oracle Developers. It intuitively makes sense that a combination of the two could very well be the optimal evaluation label for some domains.

We strongly encourage exploration of the proposed AlphaZero extension, Additional Tree Sampling. This could potentially greatly enhance data utilization, making the elegant AlphaZero approach more viable.

7 Conclusions

This paper has proposed an extension to the AlphaZero algorithm named Additional Tree Sampling and provided theoretical arguments for the validity of this approach. The first premise for Additional Tree Sampling is the possibility of changing the method used for generating state evaluation labels. Therefore, an empirical study was conducted to investigate the effects of altering the evaluation target. This study used the game Connect Four to analyze the effects of altering AlphaZero's evaluation target to instead use the augmented state value estimation given by MCTS. Our results showed very little variance from the original learning target, displaying similar behavior in all analyzed aspects. Although this study is solely based on a single environment and lacks a statistically significant number of experiments, the results strongly indicate that this alteration is a fully viable approach for certain domains. Posing that changing the learning target does not cripple the convergence rate, this clears the path for the proposed hypothesis Additional Tree Sampling, to which we encourage further research.

References

[1] D. Silver et al. “Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm”. In: (2017).

[2] D. Silver et al. “Mastering the game of Go with deep neural networks and tree search”. In: *Nature* 529 (2016), pp. 484–489.

[3] *Stockfish, Strong open source chess engine*. 2008. URL: `https://stockfishchess.org/` (visited on 05/11/2019).

[4] G. Tesauro. “Temporal Difference Learning and TD-Gammon”. In: *Communications of the ACM* 38(3) (1995), pp. 58–68.

[5] M. Campbell et al. “Deep Blue”. In: *Artificial Intelligence* 134(1-2) (2002), pp. 57–83.

[6] S. Russel and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1994.

[7] C. Shannon. “Programming a Computer for Playing Chess”. In: *Philosophical Magazine*. 7th ser. 41(314) (1950), pp. 57–83.

[8] L. V. Allis. “Searching for Solutions in Games and Artificial Intelligence”. PhD thesis. 1994.

[9] D. Silver et al. “Mastering the game of Go without human knowledge”. In: *Nature* 550 (2017), pp. 354–359.

[10] Deepak Pathak et al. “Curiosity-driven Exploration by Self-supervised Prediction”. In: (2017).

[11] S. Gelly and Y. Wang. “Exploration exploitation in Go: UCT for Monte-Carlo Go”. In: (2006).

[12] C. Atkeson and J. Santamaria. “A Comparison of Direct and Model-Based Reinforcement Learning”. In: *IN INTERNATIONAL CONFERENCE ON ROBOTICS AND AUTOMATION*. IEEE Press, 1997, pp. 3557–3564.

[13] R. Sutton and A. Barto. *Reinforcement Learning: An Introduction*. A Bradford Book, 1998.

[14] K. Kawaguchi. "Generalization in Deep Learning". In: (2017).

[15] Kurt Hornik. "Approximation capabilities of multilayer feedforward networks". In: *Neural Networks* 4(2) (1991), pp. 251–257.

[16] H. Mhaskar et al. "When and Why Are Deep Networks Better than Shallow Ones?" In: (2017).

[17] Chiyuan Zhang et al. "Understanding deep learning requires rethinking generalization". In: (2017).

[18] G. Hinton et al. "Learning representations by back-propagating errors". In: *Nature* 323 (1986), pp. 533–536.

[19] M. Fadaee et al. "Data Augmentation for Low-Resource Neural Machine Translation". In: (2017).

[20] L. Taylor and G. Nitschke. "Improving Deep Learning using Generic Data Augmentation". In: (2017).

[21] Rémi Coulom. "Efficient selectivity and backup operators in Monte-Carlo tree search". In: *In: Proceedings Computers and Games 2006*. Springer-Verlag, 2006.

[22] R. Sutton. *The Bitter Lesson*. 2019. URL: http://www.incompleteideas.net/IncIdeas/BitterLesson.html (visited on 05/12/2019).

[23] G. Marcus. "Innateness, AlphaZero, and Artificial Intelligence". In: (2018).

[24] D. Balduzzi et al. "The Shattered Gradients Problem: If resnets are the answer, then what is the question?" In: (2018).

[25] K. He et al. "Deep Residual Learning for Image Recognition". In: (2015).

[26] J. Tromp. *John's Connect Four Playground*. URL: https://tromp.github.io/c4/c4.html (visited on 05/12/2019).

[27] D. Rolnick et al. "Deep Learning is Robust to Massive Label Noise". In: (2017).

[28] A. Neelakantan et al. "Adding Gradient Noise Improves Learning for Very Deep Networks". In: (2015).

[29] Oracle Developers. *Lessons From Implementing AlphaZero*. 2018. URL: https://medium.com/oracledevs/lessons-from-implementing-alphazero-7e36e9054191 (visited on 05/13/2019).

Appendix A

The network used in the empirical test provided for Connect Four

The state representation is passed into a convolutional layer followed by a residual tower consisting of 12 blocks and finally splits into a policy head and an evaluation head. The following operations are performed in the first convolutional layer:

- A convolution of 128 filters, kernel size 4x4 and a stride of 1
- Batch Normalization
- Rectified Nonlinearity

Whereby every residual block performs the similar operations of:

- A convolution of 128 filters, kernel size 4x4 and a stride of 1
- Batch Normalization
- Rectified Nonlinearity
- A convolution of 128 filters, kernel size 4x4 and a stride of 1
- Batch Normalization
- Skip connection adding the input from previous blocks.
- Rectified Nonlinearity

The policy head thereafter outputs 7 normalized values, representing move probabilities to the given state after the operations of:

- A convolution of 32 filters, kernel size 4x4 and a stride of 1

- Batch Normalization
- Rectified Nonlinearity
- Fully connected layer with a size of 7 using the softmax activation function.

The evaluation head thereafter outputs a single scalar value describing the certainty of the first player winning by performing:

- A convolution of 32 filters, kernel size 4x4 and a stride of 1
- Batch Normalization
- Rectified Nonlinearity
- Fully connected layer with a size of 1 using the sigmoid activation function.

Appendix B

Played Games Against Optimal

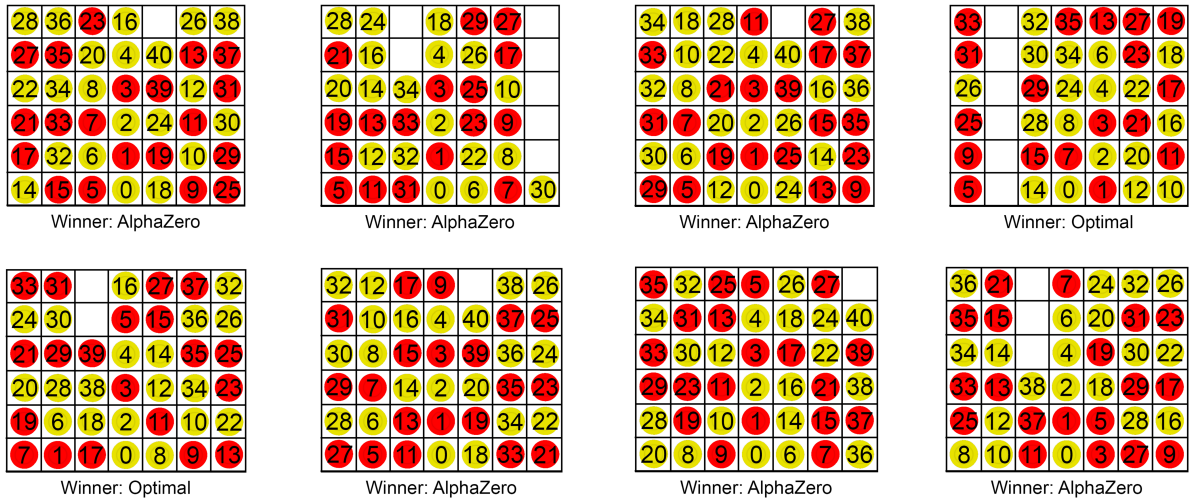

Figure B.1: Game recordings of AlphaZero playing against an optimal player as player one, using only raw network evaluations.

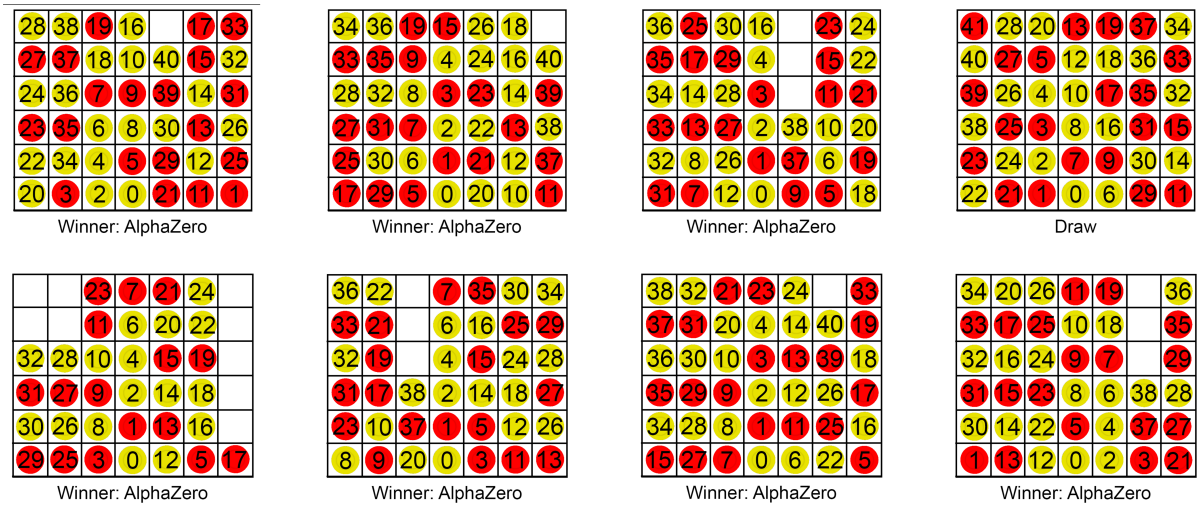

Figure B.2: Game recordings of AlphaZero playing against an optimal player as player one, using 800 MCTS iterations for each move.

Appendix C

GitHub: Implementation & Pre-trained Agents

https://github.com/FreddeFrallan/AlphaHero

TRITA-EECS-EX-2019:386