Value targets in off-policy AlphaZero: a new greedy backup

Daniel Willemsen
Centrum Wiskunde & Informatica,
Delft University of Technology
Amsterdam, The Netherlands
J.D.Willemsen@student.tudelft.nl

Hendrik Baier
Centrum Wiskunde & Informatica
Amsterdam, The Netherlands
Hendrik.Baier@cwi.nl

Michael Kaisers
Centrum Wiskunde & Informatica
Amsterdam, The Netherlands
M.Kaisers@cwi.nl

ABSTRACT

This article presents and evaluates a family of AlphaZero value targets, subsuming previous variants and introducing *AlphaZero with greedy backups (A0GB)*. Current state-of-the-art algorithms for playing board games use sample-based planning, such as Monte Carlo Tree Search (MCTS), combined with deep neural networks (NN) to approximate the value function. These algorithms, of which AlphaZero is a prominent example, are computationally extremely expensive to train, due to their reliance on many neural network evaluations. This limits their practical performance. We improve the training process of AlphaZero by using more effective training targets for the neural network. We introduce a three-dimensional space to describe a family of training targets, covering the original AlphaZero training target as well as the *soft-Z* and *A0C* variants from the literature. We demonstrate that A0GB, using a specific new value target from this family, is able to find the optimal policy in a small tabular domain, whereas the original AlphaZero target fails to do so. In addition, we show that soft-Z, A0C and A0GB achieve better performance and faster training than the original AlphaZero target on two benchmark board games (Connect-Four and Breakthrough).

KEYWORDS

Reinforcement learning; Sample-based planning; AlphaZero; MCTS

1 INTRODUCTION

Mastering complex board games has been a long studied topic in AI, as board games provide well-defined complex challenges that allow for easy measurement of performance, and thus make for ideal AI testbeds. Over the past decades, computers have been able to defeat human champions on many of these games, such as Checkers[15] (1994), Chess[5] (1997) and more recently Go [19] (2016). Algorithms for playing such games initially relied on exhaustive search methods such as alpha-beta pruning in combination with hand-crafted evaluation functions. More recently, these exhaustive search methods have been outperformed by Monte Carlo Tree Search (MCTS) and hand-crafted evaluations have given way to deep neural networks (NNs). A successful example of this is the AlphaZero approach [19–21]. In contrast to vanilla MCTS, which performs rollouts for evaluating leaf nodes, AlphaZero uses a *policy-value net*—a NN that provides value estimates as well as policy priors to MCTS. MCTS guided by this NN is used to generate data; the NN is trained on this data to provide better estimates; and using the improved NN, MCTS can in turn generate better training data, in a mutually improving cycle. One way to view this combination of NN and MCTS is as that of an expert and an apprentice, where the apprentice (NN) continually learns new knowledge from the expert agent (MCTS+NN), and is in turn able to improve the expert by providing it with better estimates [1]. Even though these algorithms are powerful, their interleaved training process has proven to be computationally extremely expensive.

We argue that the training process of AlphaZero can be improved by using more effective training targets for the neural network. The value head of the neural network provides estimates for the value of a given game position for the current player to move, and needs a value target to be trained on. AlphaZero uses game outcomes of self-play as training targets. The self-play, however, incorporates non-greedy behaviour, which is necessary to explore the state space during training. Therefore, these self-play game outcomes are only accurate value targets if the final agent, after training has ended, incorporates non-greedy exploration as well—AlphaZero behaves like an "on-policy" reinforcement learning algorithm (akin to SARSA). In practice however, moves are selected greedily when the trained solution is deployed, e.g. during tournament play. This makes the "on-policy" training undesirable in two ways:

(1) Performance of AlphaZero is limited due to the value network not converging to the true values for a greedy policy that would be followed in a tournament setting.
(2) Learned values and performance of AlphaZero are sensitive to the amount of exploration and associated hyperparameters (move selection temperature and Dirichlet noise). The more exploration is added in training, the more the value estimates deviate from the true values for a greedy policy.

We propose a family of training targets for AlphaZero that subsumes the original AlphaZero value target and others from the literature. One specific target, resulting in AlphaZero with greedy backups (A0GB), allows AlphaZero's value network to train on value estimates valid for greedy play, improving playing strength and training efficiency.

Replacing the self-play game outcomes with various other targets has been explored in previous works [6, 12]. However, these papers did not remove exploration from the value targets. Instead, they had different motives for using alternative value targets. Moerland's version of AlphaZero aims to be suitable for continuous action spaces [12], while Carlsson's adaptation is motivated by the use of additional samples from within the AlphaZero search tree [6]. The value targets used in these papers are discussed in more detail in Section 3. Even though these value targets are no longer on-policy, they still have exploration incorporated within them.

The remainder of this paper first introduces background in Section 2, and then elaborates on our proposed off-policy value target in Section 3, relating it to other value targets used in the literature by introducing a family of value targets. In Section 4, we show that

this new value target allows a tabular version of AlphaZero to converge to an optimal policy whereas other value targets fail to do so. In addition, we demonstrate our value target on two board games (Connect-Four, Breakthrough). Section 5 provides the discussion and conclusion.

2 BACKGROUND

The AlphaZero algorithm builds on two primary components: Monte Carlo Tree Search (MCTS) to perform search and Deep Neural Networks (NN) for function approximation. In this section, we first give a brief overview of MCTS. After this, in Subsection 2.2, we show how MCTS is combined with NN in the AlphaZero algorithm. Finally, in Subsection 2.3, we explain the differences between on- and off-policy learning in reinforcement learning algorithms.

2.1 Monte Carlo Tree Search

MCTS is a best-first anytime search algorithm which, in its vanilla form, uses Monte Carlo sampling to evaluate the value of a state [7, 9]. The algorithm starts from a state s for which the optimal action a is to be determined. Then, over a number of simulations, a game tree is progressively built, keeping track of the statistics of all nodes representing possible future states. A single simulation consists of four phases [4]:

(1) **Selection:** starting from the root node, a selection policy is recursively applied to descend through the tree until an unselected action is reached. This selection policy has to maintain a balance between exploiting nodes with high value estimates and exploring less frequently visited nodes. A popular selection policy is called Upper Confidence Bound Applied to Trees (UCT) [9], which selects nodes based on an upper confidence bound for the value estimates:

$$a = \arg\max_{a'} \left(Q(s, a') + c_{uct} \sqrt{\frac{\ln(N(s))}{N(s, a')}} \right) \tag{1}$$

Here, $Q(s, a')$ is the estimated value for selecting action a' from state s. c_{uct} is a coefficient determining the amount of exploration in the search. $N(s)$ and $N(s, a')$ are the visit counts of state s and the child node associated with action a' respectively.

(2) **Expansion:** Typically one new node is added to the search tree, below the newly selected action.

(3) **Simulation:** Starting from the new node, a simulation is run according to a rollout policy to produce an outcome, or return, r. This rollout policy can be as simple as selecting random actions at every step.

(4) **Backpropagation:** The outcome is backpropagated through the selected nodes to update their statistics. This backpropagation is usually done by averaging the final reward over the number of visits the node has received:

$$Q(s, a) \leftarrow \frac{(N(s, a) \cdot Q(s, a) + r)}{N(s, a) + 1}$$
$$N(s, a) \leftarrow N(s, a) + 1$$

These simulation steps are repeated until the available search time has ended. The final action is selected by selecting the action with the highest visit count.

2.2 Combined searching and learning

This section explains how MCTS and NNs can be combined to improve the performance of MCTS, as done in the AlphaZero line of algorithms. Vanilla MCTS relies on Monte Carlo evaluations to get a value estimate of a state. These value estimates can be inaccurate due to high variance. Instead, AlphaZero uses neural networks as function approximators to estimate state values. NNs can be trained in a supervised fashion, e.g. on a dataset of expert play. One core insight that AlphaZero exploits is that this expert play does not have to be human play. Instead, MCTS can be used to create stronger games than the neural network itself would have been able to play. These stronger game records can then be used to train the neural network again, resulting in a self-improving cycle.

2.2.1 The self-play training loop. AlphaZero training consists of two main steps that are performed in an iterative loop, as illustrated by **Algorithm 1**. The first step is to generate a set of training games through self-play. For every move in these games, a tree search is performed after which the next action is selected probabilistically based on the visit counts at the root. During search, the neural network is used in two ways. First, instead of doing a Monte Carlo rollout to get a value estimate when a new node is reached, the neural network provides a value estimate, $v_{NN}(s)$. Second, the policy output of the neural network, $\pi_{NN}(s)$, provides a vector of prior action probabilities. This output guides the search towards moves that the neural network predicts to be promising. To do this, the policy output is incorporated in a variant of the UCT formula, called PUCT [2], for action selection during search as can be seen in Equation 2. The states visited by all training games s, the associated game results y and the normalized MCTS visit counts of the root's child nodes (which we call $\pi_{visits}(s)$) are then stored in a replay buffer. The second step is to train the neural network on the replay buffer. The policy head is trained to approximate $\pi_{visits}(s)$ through a cross-entropy loss. Therefore, future searches will be biased towards moves that are predicted to be visited often. The value head is trained on the value targets, y, in this case the final game outcome. This is done using a mean squared error loss. An additional L2 regularization term is added to reduce overfitting of the NN parameters, θ. The complete loss function is given in Equation 3.

$$PUCT(s, a) = Q(s, a) + c_{puct} \cdot \pi_{NN}(a|s) \frac{\sqrt{N(s)}}{N(s, a) + 1} \tag{2}$$

$$l = (y - v_{NN})^2 - \pi_{visits}(s)^T \log \pi_{NN}(s) + w_{L2} ||\theta||^2 \tag{3}$$

2.2.2 Exploration in AlphaZero. To achieve diversity in the tree search as well as in the replay buffer, three types of exploration are present in the AlphaZero training. First, there is the exploration associated with the PUCT parameter c_{puct}, as explained in Subsection 2.1. Second, Dirichlet noise is added to the prior probabilities of the root node's children at the start of every search, to ensure that every node has a nonzero chance of being visited at least once during search. This modifies the PUCT equation at a root node to Equation 4, with $\pi_{NN,noisy}(a|s)$ as defined in Equation 5. $f_{dir} \in [0, 1]$ and $\alpha \in (0, \infty]$ are hyperparameters that control the

Algorithm 1: Simplified AlphaZero training loop

```
Result: A trained neural network
neural_network = NeuralNetwork();
replay_buffer = list();
while True do
    samples = list();
    // create a generation of new game samples
    for i ← 0 to n_games_per_generation do
        // play a single game
        game = new_game(neural_network) ;
        moves = list();
        while game not terminal do
            s = game.get_state() ;
            // create and search tree
            Tree = game.search() ;
            // sample action
            action = Tree.select_action();
            // store move & tree
            moves.append((s, Tree));
            // apply action in real game
            game.move(action) ;
        end
        // set policy and value targets
        value = game.outcome() ;   // set value target
        for j ← 0 to length(moves) do
            (s, Tree) = moves[j];
            // set policy target
            π = Tree.root.child_visits();
            samples.append((s, π, value));
        end
    end
    // add new samples to replay buffer
    replay_buffer.update(samples) ;
    // train neural network on replay buffer
    neural_network.train(replay_buffer);
end
```

amount and concentration of Dirichlet noise respectively.

$$UCT_{root}(s,a) = Q(s,a) + c_{puct} \cdot \pi_{NN,noisy}(s,a) \frac{\sqrt{N(s)}}{N(s,a)+1} \quad (4)$$

$$\pi_{NN,noisy}(s,\cdot) = (1 - f_{dir})\, \pi_{NN}(s,\cdot) + f_{dir} Dir(\alpha) \quad (5)$$

Finally, after the search, the action to play is selected probabilistically based on the exponentiated visit counts of each of the root's child nodes, following Equation 6. The temperature τ determines the amount of exploration in the move selection in training games.

$$\pi_{AlphaZero}(s,a) \propto N(s,a)^{1/\tau} \quad (6)$$

2.2.3 Comparing AlphaZero, AlphaGo and Expert Iteration. The AlphaZero lineage of algorithms consists of a number of sequentially developed algorithms. The first algorithm in this series is AlphaGo [19], which is initially trained in a supervised fashion on a dataset of human expert play . After this, it improves its playing strength further through self-play. This version has separate policy and value networks. The second algorithm in the series, AlphaGo Zero [21], no longer has separate policy and value networks, combining them into a single network. Also, AlphaGo Zero does not use supervised training on a human dataset, only learning from self-play. The third algorithm that was developed, AlphaZero [20], is a version of AlphaGo Zero suitable for other games than Go =. It no longer makes use of any Go-specific game features. Further, the neural network training is now done asynchronously, in parallel with the game generation. Recently, a fourth algorithm was proposed, MuZero [16], which employs a learned game model within the MCTS simulation. In addition to the AlphaZero lineage, another, very similar, algorithm has been developed in parallel: Expert Iteration (ExIt)[1]. This algorithm is used to play a game called Hex. Initially, ExIt only trained a policy network. This network was trained similarly to AlphaGo Zero. Experiments with an additional value head were also performed. The value estimates, however, were not generated from games with full searches performed in every move. Instead, the value target was retrieved from playing games purely based on the prior policy of the policy network. The output of the policy network is used in the MCTS in a slightly different manner than is done by AlphaZero. The modified UCT formula from ExIt is reproduced in Equation 7. Here, w_p is a parameter to control the relative importance of the neural network policy.

$$UCT_{ExIt}(s,a) = Q(s,a) + c_{uct}\sqrt{\frac{\log N(s)}{N(s,a)}} + w_p \frac{\pi_{NN}(s,a)}{N(s,a)+1} \quad (7)$$

2.3 On-policy and off-policy learning

The difference between on-policy and off-policy algorithms is an important distinction in reinforcement learning, which makes it also worth considering when studying AlphaZero-like algorithms. In reinforcement learning, agents are acting in an environment following a certain policy, the behavioural policy. Whilst doing this, the algorithms learn about the values for a (possibly different) target policy. If the behavioural policy and the target policy are the same, the learning algorithm is called "on-policy", if not, the algorithm is called "off-policy". On-policy learning is the simplest and avoids many difficulties associated with off-policy learning, such as the "deadly triad", where the combination of function approximation, off-policy learning and bootstrapping can result in divergent behaviour [22]. Despite this caveat, off-policy learning may improve the learning process by decoupling exploration from the value estimate, and multiple successful reinforcement learning algorithms have used it [10, 11].

3 VALUE TARGETS IN ALPHAZERO

In this section, we start with explaining the notation of AlphaZero self-play. After this, in Subsection 3.2, three value targets from the literature are described. A new greedy value target is proposed in Subsection 3.3. Finally, we show how these targets are related to each other in Subsection 3.4

3.1 Policies, value functions and notation

In AlphaZero, two types of games are used during self-play. First, there are the "real" games in which AlphaZero plays against itself.

All states encountered in these real games are then used to train the neural network. Second, there are the "simulated" games: for every move that AlphaZero makes in a real game, it performs a Monte Carlo Tree Search. This search consists of many simulated games, from which the game tree is constructed. For every node in this game tree, statistics are kept as explained in Subsection 2.1. Both the real game as well as the simulated games make use of a perfect model of the game. After finishing a real game, we can traverse through both the real game as well as through the game trees to find suitable value targets. We consider the following two policies for traversing through these games and trees:

(1) $\pi_{MCTS}(s, s_{root})$: starting at state s, in the MCTS tree which was created from root node s_{root}, this policy selects the next action based on the normalized visit counts of the children of node s. This is a policy that traverses the game tree built of simulated games. We leave out the a parameter in these policies, slightly abusing our previously used notation for policies. Leaving out this parameter indicates the function returns a probability vector over all possible actions.
(2) $\pi_{AlphaZero}(s)$: the AlphaZero move policy at a root state s. This policy traverses the real game. Every move is performed by running MCTS, and selecting actions with probabilities proportional to the exponentiated visit counts at the root node—which is influenced by the Dirichlet noise and the temperature, as explained in subsubsection 2.2.2.

Both policies have to incorporate exploration. π_{MCTS} includes exploratory moves through exploration within PUCT. In contrast, $\pi_{AlphaZero}$ needs to incorporate exploration to create diverse training data for the Neural Network, and avoid overfitting. Both these policies have greedy versions associated with them: $\pi_{MCTS,greedy}$ and $\pi_{AlphaZero,greedy}$, which select the action with the highest visit count (in simulated and real games, respectively) greedily. These are not directly used during the games of self-play training, but deployment in e.g. a competition would use $\pi_{AlphaZero,greedy}$.

Every node in a search tree has two values associated with it:

(1) $V_{NN}(s)$: The value of the represented state according to the current neural network.
(2) $V_{MCTS}(s, s_{root})$ The value of the represented state according to an MCTS search starting at root state s_{root}.

Note that for any terminal state, the values of $V_{MCTS}(s_{terminal}, s_{root})$, $V_{NN}(s_{terminal})$, and the final game outcome $V_{true}(s_{terminal})$ are identical. At any leaf node in the search tree with only a single visit, $V_{MCTS}(s_{leaf}, \cdot) = V_{NN}(s_{leaf})$. Let's denote the **multi-timestep state transition function** with $K^n_\pi : S \rightarrow S$, which uses policy π to traverse either the simulated game tree or the real game for n moves. $n = \infty$ indicates that the game tree is traversed until a leaf node or a terminal node is reached.

Now, to relate all value targets to on-policy and off-policy learning, consider the optimal value function, $v_*(s)$. The optimal value function is the maximum value function over all policies [18]:

$$v_*(s) = \max_\pi v_\pi(s)$$

Ideally, AlphaZero would learn these optimal values and their associated optimal policy. However; since this is computationally not feasible, it is instead desirable to learn about the policy that is used during final play: $\pi_{AlphaZero,greedy}$, with the associated value target:

$$y_* = v_{\pi_{AlphaZero,greedy}}(s) = \mathbb{E}\left[V_{true}\left(K^\infty_{\pi_{AlphaZero,greedy}}(s)\right)\right]$$

Finally, to describe how the different value targets approximate this value target, we introduce three kinds of policies: the ideal target policy, the actual target policy and the behavioural policy. As explained in Subsection 2.3, the behavioural policy is the policy that the agent follows during self-play: $\pi_{AlphaZero}$. The ideal target policy is the target we wish to learn about: $\pi_{AlphaZero,greedy}$. The actual target policy is the policy that is being learned about through the value targets. Getting this actual target policy to be a close approximation of the ideal target policy, with low bias[1] and variance, is the goal for designing the value targets. Actual target policies that have exploration incorporated within them result in biased value targets: sub-optimal exploratory moves bias the value of all states further on. Target policies that perform multiple steps using $\pi_{MCTS,greedy}$ contain more variance. As every step taken in $\pi_{MCTS,greedy}$ selects its action based on fewer MCTS visits than the previous one, more uncertainty and thus variance is added to the value target. A higher variance can limit the learning speed of AlphaZero as more samples are needed to get a reliable estimate.

3.2 Value targets from the literature

We can now describe the original AlphaZero and other value targets y in the literature using the notation previously described. These value targets are also illustrated in Figure 1.

AlphaZero target. In the original AlphaZero paper, the value target for every state s is set to be equal to the final game outcome of playing the game, following the non-greedy AlphaZero policy $\pi_{AlphaZero}$. This results in the following value target:

$$\begin{aligned} y_{AlphaZero}(s) &= V_{MCTS}\left(s_{terminal}, s_{terminal}\right) \\ &= V_{true}\left(s_{terminal}\right) \end{aligned}$$

where $s_{terminal} = K^\infty_{\pi_{AlphaZero}}(s)$. This value target does not bootstrap, uses single sample backups and has $\pi_{AlphaZero}$ as its target policy, making it a form of on-policy learning[2]. Due to the exploration in the target policy, this value target is biased by exploratory moves.

Soft-Z target. Instead of waiting for the final game outcome, it is also possible to use the MCTS values of the root node as a value target. We call this approach soft-Z, similar to the naming used by Carlsson [6].

$$y_{soft-Z}(s) = V_{MCTS}\left(s, s\right)$$

The MCTS value of a state converges to the true game value as the number of simulations approaches infinity. However, as AlphaZero is limited in the number of MCTS simulations it can perform, this value target is also biased by exploratory moves during the MCTS

[1] The bias that we talk about here is a different bias than the one associated with the bias-variance trade-off between bootstrapping and Monte Carlo methods. That bias is associated with the (transient) effect of value initialization. The bias we are considering here is permanent and associated with the difference between the actual and ideal target policy

[2] The replay buffer that is used for training the neural network also contains samples that were sampled from older versions of $\pi_{AlphaZero}$. One could argue that this makes AlphaZero off-policy, but we do not go into the details about the effects of replay buffers in this work.

search. This value target bootstraps in the simulated game tree, averages multiple samples for a single backup (all simulations of the MCTS search) and has the target policy π_{MCTS}, making it a form of off-policy learning.

A0C target. Moerland [12] has proposed to use a different value target in an AlphaZero variation for continuous action spaces. This target selects one child node of the root greedily, and backs up the MCTS value of this child:

$$y_{A0C}(s) = V_{MCTS}\left(K^1_{\pi_{MCTS,greedy}}(s), s\right)$$

This is again an off-policy value target. The target policy of this target takes a single step using $\pi_{MCTS,greedy}$ and afterwards uses π_{MCTS}. This target policy should be closer to $\pi_{AlphaZero,greedy}$, yet it still does not eliminate all exploration due to the usage of π_{MCTS} after the first step. This target again bootstraps in the simulated game tree and averages multiple samples for a single backup, yet over fewer samples than soft-Z does.

3.3 A greedy value target

As previously mentioned, we desire to find a value target which has a target policy that closely approximates $\pi_{AlphaZero,greedy}$, whilst following $\pi_{AlphaZero}$ as our behavioural policy. The previously described value targets all fail in that they do not have a greedy target policy and are therefore biased. The AlphaZero target incorporates exploration from $\pi_{AlphaZero}$. Soft-Z and A0C both incorporate exploration from π_{MCTS}. We propose a new greedy value target, resulting in AlphaZero with greedy backups: **A0GB**. This target no longer incorporates any exploration inside the target, allowing it to be valid for a greedy policy. This value target is found by following the MCTS policy greedily until a leaf- or terminal state is found. The value of this state is then used as the value target:

$$\begin{aligned} y_{A0GB}(s) &= V_{MCTS}\left(K^\infty_{\pi_{MCTS,greedy}}(s), s\right) \\ &= V_{NN}\left(K^\infty_{\pi_{MCTS,greedy}}(s), s\right) \end{aligned}$$

This off-policy value target once again bootstraps, although it does deeper backups than both soft-Z and A0C. It no longer is able to average multiple samples since the node from which the value is backed up only has a single MCTS visit. This value target follows a greedy policy until reaching a terminal or leaf node, and therefore has a greedy target policy: $\pi_{MCTS,greedy}$. Although this policy is greedy, it is still only an approximation for $\pi_{AlphaZero,greedy}$. Descending the tree until a leaf node results in an increasingly noisy policy as every subsequent node has fewer MCTS visits than the previous one, whereas $\pi_{AlphaZero,greedy}$ would perform a full MCTS search at every subsequent node. This difference results in value targets with higher variance.

3.4 Relationship between value targets

We now describe in more detail how the different value targets relate to one another. We can construct a value target based on three parameters, as illustrated in Figure 2: two parameters of bootstrapping and one parameter of backup width. In contrast to many other forms of reinforcement learning, where the agent moves through the environment in one "direction", AlphaZero can move through the environment in two orthogonal directions: in the direction of the real game of self-play, and in the direction of the simulated games within the MCTS trees. This also means that we can bootstrap in these two directions. From here on, we consider n-step bootstrapping in these two directions. It should be noted that in the resulting family of value targets, the action selection or the tree search itself are not modified. We only change how the training targets for the value head of the neural network are constructed once a full game of self-play has been completed.

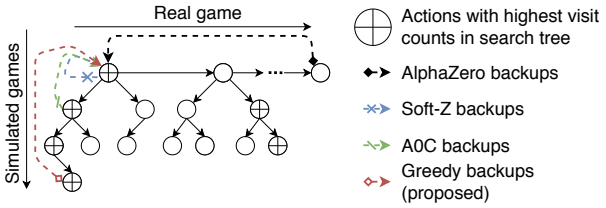


Figure 1: The relationship between the different value targets; AlphaZero uses terminated games, while greedy backups target leaves of the tree (not necessarily terminal).

When dealing with real-game bootstrapping, the only policy we can follow is $\pi_{AlphaZero}$, as the states found through following this policy are the only states that have actually been visited. This also implies that if we are bootstrapping time-wise over a greater number of steps, more exploration is incorporated in the value target.

Considering simulated-game bootstrapping, we can follow any policy as long as this policy does not take us outside of the search tree. One particularly interesting policy to follow is $\pi_{MCTS,greedy}$, as this policy is the policy that is currently estimated to be the strongest. The more steps that are taken in this direction however, the fewer MCTS visits the states have received. This results in a noisier policy and therefore could result in higher variance for the value target.

For the backup width, we can choose to backup only the neural network value estimate, V_{NN}, or backup the averaged value based on the subtree below this state in the tree, V_{MCTS}. V_{MCTS} also includes exploratory moves in the value estimate. If the simulated bootstrapping is done over more steps, these two values become more similar as the node which is used to bootstrap from has fewer visits and a smaller subtree below it. In the limit, at a leaf node, these values are equal to one another: MCTS has only visited this state once.

We can write all previously described value targets in the following manner:

$$\begin{aligned} y_{target}(s) &= V_{MCTS}\left(K^{n_{sim}}_{\pi_{MCTS,greedy}}(s_{root}), s_{root}\right) \\ s_{root} &= K^{n_{real}}_{\pi_{AlphaZero}}(s) \end{aligned} \tag{8}$$

Where n_{sim} and n_{real} are the amount of steps to bootstrap in the simulated and the real game respectively. How all four value targets outlined above are described by this unified notation can be seen in Table 1. An implementation of these value targets is shown in **Algorithm 2**.

As previously mentioned, a desirable value target has a target policy that approximates $\pi_{AlphaZero,greedy}$ closely with low bias and variance. Target policies that incorporate exploration within

	n_{sim}	n_{real}	width
AlphaZero	0	∞	single sample
soft-Z	0	0	multi sample
A0C	1	0	multi sample
A0GB	∞	0	single sample

Table 1: Relationship within the family of value targets in the unified notation as described by Equation 8.

them result in biased value targets. The amount of exploration incorporated in the value target is dependent on the backup-width and the real-game backup depth. The variance is dependent on all three dimensions: increasing the real-game backup depth results in an increased variance, as the method becomes closer to a Monte Carlo method than a TD method. Similarly, increasing the simulated-game backup depth also increases the variance. Reducing the backup-width results in larger variance as well. These effects result in the soft-Z, A0C and A0GB being viable options in the bias-variance trade-off. Whereas soft-Z has the highest bias, it also has the lowest variance. A0GB has the lowest bias, yet it also has the highest variance. A0C sits in between these two targets on both measures. The AlphaZero value target has both a high variance as well as a high bias, making this value target sub-optimal in both aspects. This value target, however, is on-policy and thus avoids the deadly triad.

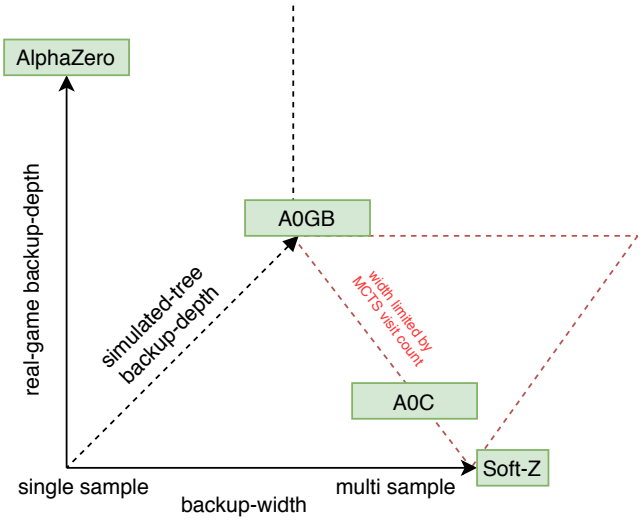


Figure 2: The three dimensions in which a value target can vary. The area beyond the red-dashed line (from Soft-Z to A0GB) cannot be reached as going deeper in the simulated-game tree results in fewer visits of backup target and therefore directly results in a narrower backup.

4 EXPERIMENTS AND RESULTS

In order to expose the differences between the various value targets, we first present results from a simple tabular domain in Subsection 4.1. In this domain the optimal strategy is trivial, and the learned values are easy to analyze. Subsequently, in Subsection 4.2, we present evaluations in the two board games Connect-Four and Breakthrough as examples of complex sequential interactions with large state spaces, which require function approximation.

Algorithm 2: Generalized sample generation of one game

Result: Training samples for neural network
`// play a single game`
game = new_game() ;
moves = list();
while *game not terminal* **do**
  s = game.get_state() ;
  Tree = game.search() ; `// create and search tree`
  action = Tree.select_action() ; `// sample` $\pi_{AlphaZero}$
  moves.append((s, Tree)); `// store move & tree`
  game.move(action) ; `// apply action in real game`
end
`// set policy and value targets`
samples = list();
for $i \leftarrow 0$ **to** *length(moves)* **do**
  (s, Tree) = moves[i];
  π = Tree.root.child_visits() ; `// set policy target`
  `// traverse real game through` $\pi_{AlphaZero}$
  (s_{real}, Tree_{target}) = moves[$i + n_{real}$];
  `// follow` $\pi_{MCTS,greedy}$ `in simulated game tree`
  s_{sim} = Tree_{target}.traverse(n_{sim});
  V_{MCTS} = Tree_{target}.value(s_{sim}); `// set value target`
  samples.append((s, π, V_{MCTS}));
end

4.1 A tabular domain

In this section, we show the behaviour of the different value targets in a small tabular domain. First, we describe the domain and tabular AlphaZero modifications, then we present and interpret the results.

Domain description. The example Markov decision process depicted in Figure 3 illustrates a problem where significant exploration, with on average poor returns, is needed to be able to reach a final positive reward. This domain has similarities to the Deep Exploration domain [13], bsuite's Deep Sea Exploration [14] and cliff-walking [22]. It is a single-player environment, where the agent has to move around in a grid. In every turn the agent can choose between moving up, right or down. The terminal states are the coloured blocks with a number in it, which is the reward the agent receives for reaching that state. It is obvious that the optimal policy would be to always move to the right in every state. We use this domain to illustrate the issues of the different backup styles.

Tabular AlphaZero modifications & experiment setup. The tabular version of AlphaZero no longer uses a Neural Network for function approximation. Instead, it uses policy and value tables, that use moving average filters to keep track of the prior policies and values. The value table is initialized with a value of 0 for every state. The policy table is initialized with equal probabilities for every action.

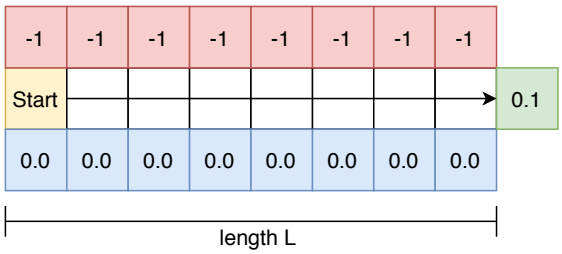


Figure 3: The gridworld domain on which a tabular version of AlphaZero is demonstrated.

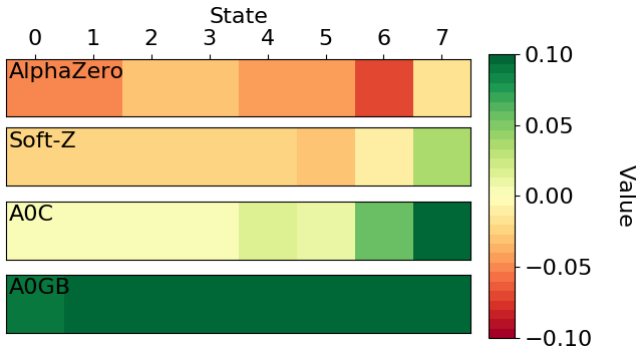


Figure 4: Value estimates in the gridworld domain, after 40 000 games of self-play with 100 MCTS simulations per move.

Exploration and search hyperparameters set to the same values as for the domains with function approximation. The gridworld domain is setup with $L = 8$ and each agent is trained for 40 000 games of self-play.

Results. The resulting value estimates and the corresponding probability of moving right for the previously described experiment can be seen in Figure 4 and Figure 5 respectively. During training, state 7 has been visited 14, 43, 106 and 540 times in total for the AlphaZero, soft-Z, A0C and A0GB value targets respectively. The off-policy value target results in the policy with the highest probability of moving to the right for every state and is the only policy that selects moving to the right when doing a greedy action selection, for all states. Also, this value target results in the highest value estimates for every state.

4.2 Domains with function-approximation

In this section, we show the behaviour of the different value targets in two domains with function approximation. The description of the domain and corresponding hyperparameters is followed by the results and their interpretation.

Domain description: Connect-Four and Breakthrough. The value targets are tested on the two board games Connect-Four and Breakthrough with a 6×6 board, illustrated in Figure 6. These domains are selected as they are both commonly used for research on MCTS [3, 17] and they require much less computational power to perform experiments on than for example Chess or Go.

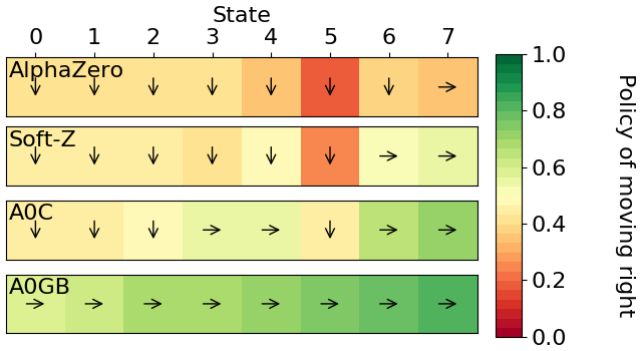
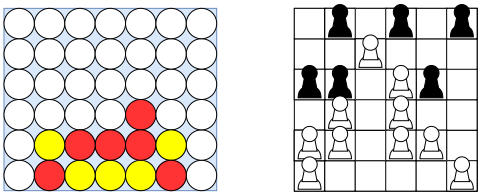


Figure 5: Color indicates policy probability of moving right after 40 000 games of self-play with 100 MCTS simulations per move. Arrows indicate the greedily selected action according to the tabular policy.



(a) Connect-Four **(b) Breakthrough**

Figure 6: Illustrations of the two board game domains on which AlphaZero value targets are compared.

Hyperparameter selection & experiment setup. The large number of hyperparameters of AlphaZero and availability of computational rescources limits the possibility of doing extensive hyperparameter optimization for all hyperparameters. As our research is strongly dependent on the amount of exploration within π_{MCTS} and $\pi_{AlphaZero}$, three hyperparameters are identified to be most influential: c_{UCT}, f_{dir} and τ. Their values are selected by individually varying them on Connect-Four for the original AlphaZero value target. c_{UCT}, f_{dir} and τ are set to 2.5, 0.25 and 1.0 respectively. They are kept constant amongst all different value targets and both games. 500 games of self-play are performed each neural network iteration, and 100 MCTS simulations were executed per move during training. Three experiments were performed. The first experiment compares the training performance of the different value targets on Connect-Four. Here, the greedy neural network policy was pitted against a pure MCTS opponent with 200 simulations for 400 games every 10 network iterations. A similar experiment is performed on Breakthrough. However, since all AlphaZero variants very quickly learn to significantly outperform our pure MCTS baselines in Breakthrough, all AlphaZero variants were tested against a baseline AlphaZero that was trained for 50 000 games instead. The final experiment is designed to measure the sensitivity of the value targets to exploration. Agents were trained for 25000 games on Connect-Four and pitted against a pure MCTS opponent with 5000 simulations for 800 games with varying temperature coefficients.

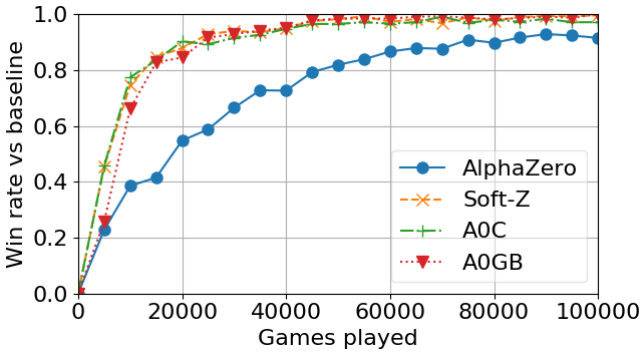


Figure 7: Performance of the neural network in Connect-Four versus a pure MCTS opponent with 200 simulations.

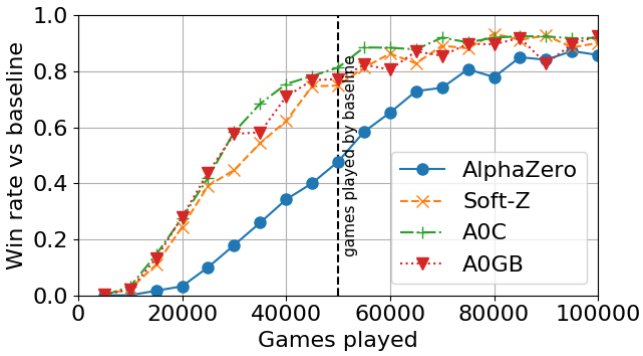


Figure 8: Performance of AlphaZero in 6x6 Breakthrough against an on-policy trained AlphaZero, trained for 50 000 games.

Results. In Figure 7 and Figure 8, the training performance of AlphaZero is shown for Connect-Four and Breakthrough respectively. As expected, the performance of all AlphaZero agents increases over its training period, for both Connect-Four as well as Breakthrough. In both games, the agents trained with the original on-policy AlphaZero value target learns significantly slower than the three alternative value targets. In addition, the final performance is worse. Interestingly, there is only a marginal difference in performance between the three alternative value targets. The results of the temperature sensitivity experiment can be seen in Figure 9. The A0GB value target is less sensitive to the temperature parameter than the original AlphaZero value target.

5 DISCUSSION AND CONCLUSIONS

In this work, we evaluated a number of different value targets to use in an AlphaZero algorithm. The value target used in the original AlphaZero algorithm incorporates exploration within the value target, which results in bias. Therefore, AlphaZero is unable to converge to an optimal policy. In addition, this value target suffers from high variance. We introduce a three-dimensional space to describe a family of training targets that subsumes the original AlphaZero training target, two other variants from the literature and a novel greedy training target.

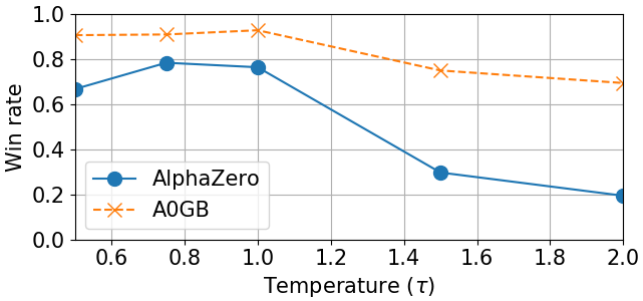


Figure 9: Performance of AlphaZero with 100 simulations after training for 25000 games on Connect-Four for different values of the temperature coefficient τ during training. In the final evaluation, the moves are selected greedily.

Our results show that within a small tabular domain, the new greedy value target is the only target resulting in the correct greedy action selection. This confirms the hypothesis that with a limited amount of MCTS simulations, all three non-greedy value targets are unable to find the optimal policy, even after playing a large number of games in a small environment. Even though this specific value target is the only tested value target that is able to find the optimal policy in the tabular domain, there is only little difference in the performance of different value targets from this family in the board games Connect-Four and Breakthrough, although they all performed significantly better than the original AlphaZero value target. All three off-policy value targets are trading off variance and bias in different ways. It is possible that the amount of variance is more important in the case of Connect-Four and Breakthrough, as the amount of computation and the limited size of the neural network prevent AlphaZero from finding a policy close to the optimal one. An alternative explanation would be that limited hyperparameter tuning is a key limiting factor in that it brings the performance of the three value targets closer together.

Future research may exploit off-policy training to improve exploration, further explore the family of value targets (e.g. such as TD(λ) backups in the two dimensions), or extend the analysis to the training targets of AlphaZero's policy head. One potential off-policy value target variant worth exploring would traverse the real game until an exploratory move is made, after which the simulated tree is followed greedily. This removes exploration from real-game bootstrapping. It is also an open question whether the winner's curse applies to A0GB (see discussion in Double Q-learning [8]). Another route worth exploring would be to train the value head on additional moves that exist in the game tree but are not played in the real game, similar to the TreeStrap algorithm that updates a heuristic function based on all search node values for a minimax search [23]. This is now possible as our proposed value target does not require a final game outcome.

ACKNOWLEDGMENTS

This work is part of the project *Flexible Assets Bid Across Markets* (FABAM, project number TEUE117015), funded within the Dutch Topsector Energie / TKI Urban Energy by Rijksdienst voor Ondernemend Nederland (RvO).

REFERENCES

[1] Thomas Anthony, Zheng Tian, and David Barber. 2017. Thinking fast and slow with deep learning and tree search. In *Advances in Neural Information Processing Systems*. 5360–5370.

[2] David Auger, Adrien Couetoux, and Olivier Teytaud. 2013. Continuous upper confidence trees with polynomial exploration–consistency. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 194–209.

[3] Hendrik Baier and Mark HM Winands. 2014. Monte-carlo tree search and minimax hybrids with heuristic evaluation functions. In *Workshop on Computer Games*. Springer, 45–63.

[4] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. 2012. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games* 4, 1 (March 2012), 1–43. https://doi.org/10.1109/TCIAIG.2012.2186810

[5] Murray Campbell, A.Joseph Hoane, and Feng hsiung Hsu. 2002. Deep Blue. *Artificial Intelligence* 134, 1 (2002), 57 – 83. https://doi.org/10.1016/S0004-3702(01)00129-1

[6] Fredrik Carlsson and Joey Öhman. 2019. *AlphaZero to Alpha Hero : A pre-study on Additional Tree Sampling within Self-Play Reinforcement Learning*. Bachelor's thesis. KTH, School of Electrical Engineering and Computer Science (EECS).

[7] Rémi Coulom. 2006. Efficient selectivity and backup operators in Monte-Carlo tree search. In *International conference on computers and games*. Springer, 72–83.

[8] Hado V Hasselt. 2010. Double Q-learning. In *Advances in neural information processing systems*. 2613–2621.

[9] Levente Kocsis and Csaba Szepesvári. 2006. Bandit Based Monte-Carlo Planning. In *Machine Learning: ECML 2006*, Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 282–293.

[10] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 2015. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971* (2015).

[11] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).

[12] Thomas M Moerland, Joost Broekens, Aske Plaat, and Catholijn M Jonker. 2018. A0C: Alpha zero in continuous action space. *arXiv preprint arXiv:1805.09613* (2018).

[13] Ian Osband, Charles Blundell, Alexander Pritzel, and Benjamin Van Roy. 2016. Deep exploration via bootstrapped DQN. In *Advances in neural information processing systems*. 4026–4034.

[14] Ian Osband, Yotam Doron, Matteo Hessel, John Aslanides, Eren Sezener, Andre Saraiva, Katrina McKinney, Tor Lattimore, Csaba Szepezvari, Satinder Singh, et al. 2019. Behaviour suite for reinforcement learning. *arXiv preprint arXiv:1908.03568* (2019).

[15] Jonathan Schaeffer, Robert Lake, Paul Lu, and Martin Bryant. 1996. CHINOOK The World Man-Machine Checkers Champion. *AI Magazine* 17, 1 (Mar. 1996), 21. https://doi.org/10.1609/aimag.v17i1.1208

[16] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. 2019. Mastering atari, go, chess and shogi by planning with a learned model. *arXiv preprint arXiv:1911.08265* (2019).

[17] Shiven Sharma, Ziad Kobti, and Scott Goodwin. 2008. Knowledge Generation for Improving Simulations in UCT for General Game Playing. In *AI 2008: Advances in Artificial Intelligence*, Wayne Wobcke and Mengjie Zhang (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 49–55.

[18] David Silver. 2015. UCL Course on RL Lecture 2: Markov Decision Processes. (2015). https://www.davidsilver.uk/wp-content/uploads/2020/03/MDP.pdf

[19] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. 2016. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature* 529, 7587 (Jan. 2016), 484–489. https://doi.org/10.1038/nature16961

[20] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. 2017. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815* (2017).

[21] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. 2017. Mastering the game of Go without human knowledge. *Nature* 550 (Oct. 2017), 354–359. http://dx.doi.org/10.1038/nature24270

[22] Richard S. Sutton and Andrew G. Barto. 2018. *Introduction to Reinforcement Learning* (2nd ed.). MIT Press, Cambridge, MA, USA.

[23] Joel Veness, David Silver, Alan Blair, and William Uther. 2009. Bootstrapping from Game Tree Search. In *Advances in Neural Information Processing Systems 22*, Y. Bengio, D. Schuurmans, J. D. Lafferty, C. K. I. Williams, and A. Culotta (Eds.). Curran Associates, Inc., 1937–1945. http://papers.nips.cc/paper/3722-bootstrapping-from-game-tree-search.pdf