Action Selection Policies for Walking Monte Carlo Tree Search

R.A.N. Starre
Action Selection Policies for Walking Monte Carlo Tree Search

by

R.A.N. Starre

to obtain the degree of Master of Science at the Delft University of Technology, to be defended publicly on Wednesday August 29, 2018 at 10:00 AM.

Student number: 4334620
Faculty: EEMCS
Master programme: Computer Science
Specialisation: Pattern Recognition and Bioinformatics
Thesis committee: Prof. dr. M. Loog, TU Delft (PRB), supervisor
Prof. dr. ir. M. J. T. Reinders, TU Delft (PRB)
Dr. M. F. Aniche, TU Delft (SERG)

An electronic version of this thesis is available at http://repository.tudelft.nl/.
During my studies in Human Movement Science, I was inspired by fellow students who were highly motivated and passionate about doing research in that area. This made me realize that I also wanted to do research, but within a different area, and Computer Science offered such an opportunity.

The courses I've enjoyed the most during my studies in Computer Science were in the areas of algorithms, artificial intelligence, and machine learning. What I like about the courses in these areas is that they are focused on solving real-world problems, using methods with a solid theoretical background and critically analyzing the performance in experimental settings.

This all contributed to my decision of doing my thesis at the TU Delft in the Pattern Recognition and Bioinformatics group under the supervision of Marco Loog.

While I was searching for a thesis topic, my attention was drawn to the Alpha(Go)Zero papers. I was hooked! Their elegant algorithm captured my imagination. It is, in essence, a combination of machine learning, artificial intelligence, and algorithms. I've found that I'm very passionate about this area of research.

This method is able to learn to play the game of Go at a very high level, it was the first program to beat a human world champion player. It learns by playing games against itself, generating data from which it learns, which leads to improved play. Since the method seems very computationally expensive, I wanted to focus on making the method more efficient. Initially, my attention was focused on the Reinforcement Learning and function approximation part of the algorithm, and on new Deep Reinforcement Learning exploration methods. Trying to simplify the algorithm so I could be sure any results I found were due to my additions, and not some strange interaction, I at some point realized that the Monte Carlo tree search provided more promising methods of exploration. After some more study in this direction, I realized that I could create a new method without function approximation, for which I could explore different exploration strategies. This method uses the self-play element and applies it to Monte Carlo tree search. Testing different exploration strategies show that there are some large differences in the stability and speed of training. The results of this method provide some insights into the exploration within the algorithms that combine function approximation and Monte Carlo tree search as well.

This thesis consists of three parts. The first and foremost is an article titled "Action Selection Policies for Walking Monte Carlo Tree Search", formatted according to the specifications of the 2019 AAAI conference. The second part contains some supplemental material as well as an extension of some of the background for readers that are less familiar with Reinforcement Learning, Bandit Problems, Upper Confidence Bound 1 applied to Trees, and Monte Carlo tree search. The final part contains some notes, experiments and other thoughts or writings that I created during my thesis work.

I would like to thank Marco Loog for his valuable advice and feedback in our meetings and email exchanges. I would also like to thank my other committee members Marcel Reinders and Mauricio Aniche. Finally, I would like to thank my family and friends for their support!

R.A.N. Starre
Delft, 15 August 2018
# Contents

1 Article

2 Appendices
   2.1 Paper supplement ........................................... 9
   2.2 Introduction .............................................. 14
   2.3 Overview .................................................. 14
   2.4 Markov Decision Processes ................................ 14
      2.4.1 Notation ............................................. 15
      2.4.2 Return ............................................... 15
      2.4.3 Policies & Value Functions ........................... 15
   2.5 Bandit Problem ............................................ 16
      2.5.1 Definition ........................................... 16
      2.5.2 Regret ............................................... 16
      2.5.3 Solving the exploration-exploitation dilemma ...... 17
      2.5.4 UCB1 .................................................. 17
   2.6 UCT .......................................................... 17
   2.7 Monte Carlo Tree Search ............................... 18

3 Notes & Log
   3.1 Reinforcement Learning .................................. 21
      3.1.1 Temporal-Difference Learning ....................... 21
   3.2 Deep Reinforcement Learning ............................ 21
      3.2.1 Problems using nonlinear function approximation 22
      3.2.2 Experience Replay .................................. 22
      3.2.3 Target Network ...................................... 23
   3.3 Thesis Proposals ......................................... 23
      3.3.1 Tuning AlphaZero .................................... 23
      3.3.2 From AlphaZero to hero .................................. 23
      3.3.3 Ideas .................................................. 23
      3.3.4 Deep Exploration .................................... 25
      3.3.5 Three ideas .......................................... 26
      3.3.6 AlphaZero & DQN .................................... 27
      3.3.7 AlphaZero training ................................... 27
      3.3.8 Tabula Rasa .......................................... 28
      3.3.9 Questions, ideas ..................................... 29
      3.3.10 More Thoughts on AlphaZero ......................... 30
   3.4 Experiments ............................................... 32
      3.4.1 Values & Priors results ............................... 32
      3.4.2 Progress Measurements ................................ 34
      3.4.3 Experiments with different settings ......... 42
   3.5 OmegaHero ............................................... 43

Bibliography ................................................. 57
**Action Selection Policies for Walking Monte Carlo Tree Search**

**R.A.N. Starre**  
Delft University of Technology

**Abstract**

Recent Reinforcement Learning methods have combined function approximation and Monte Carlo Tree Search and are able to learn by self-play up to a very high level in several games such as Go and Hex. One aspect in this combination that has not had a lot of attention is the action selection policy during self-play, which could influence the efficiency of learning in the studied games. Inspired by the recent methods we propose a sample based planning method that uses Monte Carlo tree search in a manner akin to self-play. Using this method we explore a variety of action selection policies based on the statistics from obtained with Monte Carlo Tree Search. We found that the action selection policies, combined with a parameter controlling the amount of exploration, had an effect on the speed of learning. The results suggest that methods using self-play to learn about an environment should consider the action selection policy to improve performance and learning efficiency. Since our method was able to learn faster than standard Monte Carlo Tree Search, our proposed method in itself is interesting to study further.

**Introduction**

One algorithm that has had a lot of success, particularly in Go (Browne et al. 2012), is Monte Carlo Tree Search (MCTS) (Coulom 2006) often with a Upper Confidence Bounds 1 applied to Trees (UCT) strategy (Kocsis and Szepesvári 2006). Originally created to improve play in the game of Go, it has been used in a number of other domains, such as security, physics simulations, and bus regulation (see the survey by Browne et al. (2012) for more examples). It is an algorithm that can be used without requiring a lot of domain knowledge and still perform well, even in its most basic form (Ilhan and Etaner-Uyar 2017).

Recently, a new reinforcement learning (RL) method has been used in Go that is able to learn by itself using a combination of MCTS and function approximation (Silver et al. 2017a; 2017b). This method was able to achieve a very high level of play in the game Of Go, beating their previous method that was the first playing algorithm to beat a professional Go player on a 19x19 board (Silver et al. 2016). The latest method, AlphaZero (AZ), is more generalized and was shown to be able to learn to perform at a high level not just in Go but in chess and shogi as well (Silver et al. 2017b). Concurrently a similar method, Expert Iteration (EXIT), has been developed, which was able to beat the current best public program in the game of Hex (Anthony, Tian, and Barber 2017). These methods are able to learn by itself using a general reinforcement learning and planning approach, since MCTS has been able to perform well in multiple areas (Browne et al. 2012), that makes this method an interesting one to study.

These methods learn by playing games against itself, using MCTS. They iteratively generate data through self-play and learn from this data to improve their playing strength. This learning is done through RL, it learns both the value of states and a policy, this knowledge is subsequently used within the MCTS, which results in a better search, which in turn improves the data used for learning the value and policy.

AZ and EXIT both used a different action selection policy to select actions during the self-playing phase. AZ used an action selection policy based on the number of visits, but also changed the strategy after a certain amount of moves, and some additional randomness in the form of Dirichlet noise was added. EXIT on the other hand used the learned policy. Several other studies have proposed action selection policies. But in those studies, the main focus in those studies was not on a comparison of different action selection policies (Chaslot et al. 2008; Chang et al. 2005; Nakhost and Müller 2009; Walsh, Goschin, and Littman 2010), or only in the online case where one is only interested in the direct results (Chaslot et al. 2008).

MCTS can also be seen as a self-improving learning method. With more visits, the estimates become better and thus its tree policy also becomes more accurate. Based on this idea and the self-play used in EXIT and AZ, we propose a simple MCTS based planning method for finite Markov Decision Processes (MDPs) that uses MCTS in a manner akin to self-play. In this method, we implement self-play in the sense that one iteratively performs a fixed amount of MCTS cycles in a state and then selects an action according to an action selection policy and moves to this state, until a terminal state is reached. The tree is maintained but the backup is limited to the subtree. In this way instead of starting MCTS cycles only from the root state it “walks” through the tree until it reaches a terminal state, then it restarts at the root. In essence, it is a repeated online performance where the time for MCTS in each state is limited to some fixed value. Compared to normal MCTS this method spends more time deeper in the tree. The reasoning is that by stepping deeper into
the tree the estimates deeper in the tree will converge faster, leading to more accurate estimates in future MCTS cycles earlier in the tree, which could speed up learning overall.

Additionally, we study several different action selection policies, to study the effect they have on the learning efficiency. These action selection policies are based on the statistics from the MCTS, since these could also be used in RL methods combining function approximation and MCTS.

We compare the speed of learning of our proposed method with MCTS and find it learns more efficiently. We also find that the different action selection policies, in combination with MCTS and find it learns more efficiently. We also find the different action selection policies, in combination with a parameter controlling the exploration, have a large effect on the learning efficiency and stability.

**Background**

We consider problems that can be modeled by a finite-horizon MDPs, because we are interested in environments like tic-tac-toe, Go, Hex, chess, etc.

In a MDP we have a set of states $S$, with $s_0$ the initial state of the environment, a set of actions $A$, with $A(s) \subseteq A$ a set of actions that are allowed in state $s \in S$, a transition function $T^a_{ss'}$, that determines the probability of reaching state $s'$ if action $a$ is chosen in state $s$, i.e.

$$P^a_{ss'} = Pr(s_{t+1} = s'|s_t = s, a_t = a),$$

a reward function $R(s)$, giving the expected reward for reaching a state $s$ (Russell and Norvig 2016).

An agent learning the MDP will have to choose an action at each time step based on the current state, we call this map-

3. Simulation: Starting from the leaf node a default policy were selected in step 1, updating their statistics.

4. Backpropagation: The reward $R$ that was obtained in the simulation step is backpropagated through the nodes that were selected in step 1, updating their statistics.

One of the most popular tree policies (Browne et al. 2012) is the UCT algorithm (Kocsis and Szepesvári 2006). The UCT algorithm combines tree search with the Upper Confidence Bound 1 (UCB1 (Auer, Cesa-Bianchi, and Fischer 1998). A policy is optimal if its expected return is the same or greater than that of every other policy for all states in the MDP. We define the optimal state-

$$V^*(s) = \max_{\pi} V^\pi(s),$$

for all states $s \in S$. Optimal policy also have an optimal action-value function $Q^*$, defined as follows:

$$Q^*(s, a) = \max_{\pi} Q^\pi(s, a),$$

for all states $s \in S$ and all actions $a \in A(s)$. To show the relation between $V^*$ and $Q^*$ we can write $Q^*$ as a function of $V^*$, i.e.:

$$Q^*(s, a) = \mathbb{E}[R(s_{t+1}) + V^*(s_{t+1})|s_t = s, a_t = a].$$

**Monte Carlo Tree Search**

MCTS is a tree search algorithm that uses Monte-Carlo evaluation, i.e. averaging the outcome of multiple samples, for estimating values of nodes (Coulom 2006). One attractive quality of the algorithm is that is anytime, it can be interrupted at any moment and still give a valid solution at the root (Browne et al. 2012). This is because it works iteratively and after each iteration values are backpropagated through the tree so that there always is an estimation of the best move at the root. Multiple iterations are expected to improve the estimates.

MCTS starts with an empty tree and is initially given a root state node, after which it iteratively performs the four steps shown in 1, we refer to one iteration of these four steps as a MCTS cycle. The steps work as follows:

1. Selection: Starting from the root node a child node is selected recursively, according to the tree policy, until we reach a leaf node. A leaf node is either a terminal node, i.e. a node corresponding to a terminal state, or a node that is expandable. A node is expandable if it non-terminal and has actions available that do not have a corresponding edge in the tree. If the leaf node is a terminal node step 2 and 3 are skipped.

2. Expansion: One or more child nodes, that correspond to states not yet in the tree, are attached to the leaf node.

3. Simulation: Starting from the leaf node a default policy recursively chooses an action, according to the default policy, until a terminal state $s_t$ with an outcome reward $R(s)$ is reached. If the leaf node $s_t$ itself is terminal, we use the outcome reward $R(s_t)$ from this state. The most basic form of a default policy is randomly choosing one of the available actions (Browne et al. 2012).

4. Backpropagation: The reward $R$ that was obtained in the simulation step is backpropagated through the nodes that were selected in step 1, updating their statistics.

One of the most popular tree policies (Browne et al. 2012) is the UCT algorithm (Kocsis and Szepesvári 2006). The UCT algorithm combines tree search with the Upper Confidence Bound 1 (UCB1 (Auer, Cesa-Bianchi, and Fischer 1998)) strategy, which is a strategy for selecting arms in a
multi-armed bandit problem. The main idea of the algorithm is that, by selectively choosing actions, the most promising part of the tree can be explored more, allowing for better performance (Kocsis and Szepesvári 2006). In order to do this, the suboptimal state-action pairs should be identified quickly. A state-action pair \((s, a)\) is suboptimal if for the same state there is a state-action pair \((s, b)\) \((b \in A(s), b \neq a)\) that has a higher value. To do this, UCT trades-off between greedily choosing the current best-estimated state-action pairs and exploratory choosing of suboptimal looking state-actions pairs. In this way, it attempts to obtain good estimates of the best state-action pairs while simultaneously attempting not to miss other potentially good state-actions pairs.

Concretely, in the selection phase in each state \(s\) the action that maximizes \(Q(s, a) + U(s, a)\) is selected, where

\[
U(s, a) = c \sqrt{\frac{\ln \sum_b N(s, b)}{N(s, a)}},
\]

and \(c\) an exploration constant. \(Q(s, a)\) can be seen as an exploitation term while \(U(s, a)\) is an exploration term. \(U(s, a)\) favors state-action pairs that are less visited while \(Q(s, a)\) favors state-action pairs that have a high estimated value. With more total visits the term \(U(s, a)\) becomes smaller and the method favors higher valued state-actions. \(U(s, a)\) makes sure that every state-action pair is eventually visited. Kocsis and Szepesvári (2006) have shown that for a finite horizon MDP with rewards scaled to lie within the interval \([0, 1]\), UCT converges to the optimal action in the limit, for an exploration constant of \(2 \frac{D}{\sqrt{2}}\).

Method

Here we detail our proposed method, Walking MCTS, and the different action selection policies that we investigate.

The algorithm takes the starting state of the game, \(s_0\), a parameter \(C\) that determines the amount of MCTS cycles in the root state, and an annealing rate parameter \(AR\) which reduces the number of MCTS cycles in subsequent states. Starting in \(s_0\), Walking MCTS performs \(C\) MCTS cycles, then chooses an action according to its action selection policy and observes the next state after applying that action, in the new state it performs \(C - AR\) MCTS cycles. Pseudo-code for the main algorithm is depicted in Algorithm 1.

Walking MCTS repeats episodes of the game, performing a fixed number of MCTS cycles in each state before select-
imizes the following is chosen:

\[ UCT(s, a) = Q(s, a) + c \sqrt{\frac{\ln \sum_b N(s, b)}{N(s, a)}}. \]

In the expansion of a leaf node \( s_l \) we create an edge \((s_l, a)\) for each \( a \in A(s_l) \) and in each edge we maintain the number of times the edge has been visited, \( N(s, a) \), the action value \( Q(s, a) \), and the total action value, \( W(s, a) \).

In the simulation phase, we use a random roll-out policy starting from the leaf node and use the outcome of the roll-out in the terminal state as the reward.

**Action Selection Policies**

We consider multiple action selection policies that depend on the statistics at the root node of the tree that is build using MCTS, they are mainly based on suggested methods from Chaslot et al. (2008). We are interested in these methods since the combination of MCTS with RL in combination with function approximation has been fruitful in learning games (Silver et al. 2017a; 2017b; Anthony, Tian, and Barber 2017). In those settings, action selection policies based on the statistic in the tree can also be used, although the results of each specific policy might differ from our case.

We consider four different action selection policies, based on value, visits, Upper Confidence Bound (UCB) on the value, and Lower Confidence Bound (LCB) on the value. The probabilities of selection an action for the different policies are as follows:

1. **Value policy**, \( Pr(s, a) = \frac{Q(s, a)^{1/\tau}}{\sum_b Q(s, b)^{1/\tau}} \).

2. **Visits policy**, \( Pr(s, a) = \frac{N(s, a)^{1/\tau}}{\sum_b N(s, b)^{1/\tau}} \).

3. **UCB policy**, \( Pr(s, a) = \frac{UCT(s, a)^{1/\tau}}{\sum_b UCT(s, b)^{1/\tau}} \).

4. **LCB policy**, \( Pr(s, a) = \frac{(LCT(s, a) + \min(LCT))^{1/\tau}}{\sum_b (LCT(s, b) + \min(LCT))^{1/\tau}} \).

Where \( LCT(s, a) \) is a lower confidence bound, i.e.:

\[ LCT(s, a) = Q(\text{node}, a) - c \sqrt{\frac{\ln \sum_b N(s, b)}{N(s, a)}}, \]

\( \min(LCT) \) is the minimum \( LCT(s, a) \) for \( a \in A(s) \) (or 0 if it has a value > 0, to ensure the values are ≥ 0), and \( \tau \) is a temperature that controls the amount of exploration. With \( \tau = 0 \) the actions are chosen with a probability proportionally to its value, with \( \tau > 1 \) the probabilities become more uniform, and with \( \tau < 1 \) the probabilities focus on the actions with higher value, as \( \tau \to 0 \) the action with the highest value is chosen deterministically.

Choosing actions deterministically, i.e. \( \tau \to 0 \), might result in the same actions being chosen over and over during learning which could result in the agent not learning the game efficiently. On the other hand, a \( \tau \) that is too large could result in the method not focusing enough on the most promising states, spending too much time learning part of the state-space that are non-relevant in the game when played at a high level. Therefore we experiment with several values of \( \tau \) in the range \([0, 1]\).

**Experiments**

We test our method in the game of tic-tac-toe. While there are only 5478 positions in tic-tac-toe, there are 255168 different paths (Schaefer 2002). By using values over actions instead of states this allows us to compare our methods and get some separable results.

In addition, it is a challenging game in the sense that there are many states where a player can select actions that will lead to a different outcome, meaning that the early estimates Monte-Carlo estimations will typically underestimate the value of actions leading to this state. This makes it necessary to visit these states more often in order to get an accurate estimate, but the underestimation could make the search focus on alternatives instead. Compare this to a state where all the actions lead to the same (or a similar one, when not using a game) outcome, after only a couple of visits the estimate will on average be a lot more accurate.

The goal is to learn a policy that is able to play the game of tic-tac-toe well. To measure the strength of the learned policy we let the method play against a "perfect" tic-tac-toe player. The perfect tic-tac-toe player knows the outcome (i.e. win, draw, or loss) of every state under perfect play from both players, and in each state picks an action that has the maximum value. That means the best our method can do is draw, hence we show the percentage of games that resulted in a draw.

We test the methods with the four different action selection policies and four different settings of the temperature parameter \( \tau \in [0, 0.1, 0.5, 1] \). The exploration parameter \( c \) was initially set to \( 2 \frac{D}{\sqrt{N}} \), where \( D \) is the maximum depth of the tree, based on the result from (Kocsis and Szepesvári 2006). However, after some initial experimentation, we found better results with \( c = \frac{\sqrt{2}}{\sqrt{2}} \), so we used this in all our experiments.

Every 5000 leaf evaluations a model is saved, and we let the model play 100 games against the perfect player every second model. The number of leaf evaluations was used under the assumption that the evaluation leaves is what will cost the most processing time (Kocsis, Szepesvári, and Willemsen 2006). Since there is some stochasticity in the policies and self-play we run five trials for each method and show the means as well as the variance of the results in our figures. In the games against the perfect player the model uses the value policy with \( \tau = 0 \), this is just the action \( a \in A(s) \) with maximum \( Q(s, a) \). In each state the model does 10 additional MCTS cycles, mainly to assure all actions have been visited at least once even in the earliest models, for frequently visited states this should not have a significant impact.

The initial number of MCTS cycles \( C_s \) was set to 50 with an annealing rate \( AR \) of 5. Some additional experiments, shown in supplements, showed that without annealing the method generally performed worse or similar, and there was little difference with a setting of \( C_s = 80, AR = 10 \).

We compare the effects of the temperature \( \tau \), the different action selection policies, and also compare it to a basic MCTS.
Temperature Effects

Our expectation was that in general that with temperature $\tau = 0$ all the policies would be too greedy, and lead to worse results than less greedy temperature settings.

For the value policy we indeed observe this, displayed in Figure 2. With settings of $\tau = 0.5$ and $\tau = 1$ the model is able to draw 100% of the time before the 30th iteration, while with a setting of $\tau = 0$ and $\tau = 0.1$ the models do not yet converge before the 40th iteration. All three setting with $\tau > 0$ make rapid progress at the start compared to the model with $\tau = 0$.

For the visits policy, the trends are slightly different. In Figure 3 we notice that this action selection policy is able to learn to draw 100% of the time with a setting of $\tau = 0.5$. Both the settings of $\tau = 0$ and $\tau = 0.1$ show slower initial progress, while with $\tau = 1$ the initial progress is comparable to $\tau = 0.5$.

The UCB policy has similar initial progress for all settings of $\tau$, shown in Figure 4. There is a difference in the time it takes to achieve a consistent 100% draw rate, here the greedier setting $\tau = 0$ takes a lot longer to converge.

The LCB policy shows results that are very like those of the value policy. Settings of $\tau = 0.5$ and $\tau = 1$ learn rapidly and converge to a 100% draw rate, while settings of $\tau = 0$ and $\tau = 0.1$ learn slower and only converge around the 40th iteration.

Policy Comparison

Here we compare the results of the different policies for the different settings of $\tau$.

As we expected, a very greedy temperature setting of $\tau = 0$ did overall lead to worse results than less greedy settings. In Figure 6 we can see that for $\tau = 0$ the value, visits, and LCB policies all show comparable learning speeds. They also all show a large variance, indicating somewhat unstable learning. This likely happens because the greedy strategy leads to differences in the learning experience of the models since the early estimates of value and subsequently the visits will be guided in different directions. In this case, there should be less variance in the results with less greedy temperature settings.

In Figure 6 we also see that the UCB policy learns much
more rapidly and more stable compared to the other three action selection policies. This is likely to happen because this policy interacts with the tree policy, in the sense that both select the actions with the highest UCT value, and thus influence each other. Selecting actions will reduce the exploration term in the UCT term, so this naturally leads to some exploration when using the UCB policy even with $\tau = 0$, causing this policy to work better in a greedier setting compared to the other policies.

With a temperature $\tau = 0.1$ we observe similar results, see Figure 7. As speculated there does seem to be a slight decrease in the variance of the results for the value, visits, and UCB policies.

The reduction in the variance of the results is more pronounced in Figure 8. Here all the policies show a similar learning rate and converge to a 100% draw rate around the 20th iteration. The results with a setting of $\tau = 0.5$, see Figure 9, are similar to those for $\tau = 0.5$ with the exception of the visits policy. For the visits policy the results are less stable with $\tau = 1$ compared to $\tau = 0.5$. This is likely to occur due to an interaction with the tree policy, since with low visit counts the exploration term is relatively large the actions will have visit counts that are relatively close together, this has the effect that with the visits policy with a temperature of $\tau = 1$ the chance of choosing any action is initially roughly the same, but visiting an action will increase its visit count significantly, since that action is then effectively visited $C_s - AR$ more times. This could cause relatively large differences in the learning experience between the different models.

Finally, we also show the results of running MCTS in Figure 1. Compared to our method with its best setting of $\tau = 0.5$ the MCTS converges more slowly to a 100% draw rate. Our models converge around the 20th model, while MCTS converges around the 28th model, making our method considerably faster.

Discussion

A lot of variations and improvements for MCTS have focused on improving the tree policy and the default policy (Browne et al. 2012). For the default policy often focused on a way of improving the action selection over the ran-
dom selection, and in the case of the tree policy often on making the method focus more on promising moves, to reduce the amount of time spend in likely bad moves more so than UCT does. Recent MCTS methods have been able to combine these improvements with the help of function approximation (Silver et al. 2017a; 2017b; Anthony, Tian, and Barber 2017). These methods learn by playing games against themselves. Here, the action selection policy during self-play is likely to have an impact on the efficiency of learning as well as the playing strength it obtains, since it determines the data it generates. In our proposed method we tested different action selection policies and found that in our case there indeed were differences between different selections.

A one on one comparison to the action selection policies of AZ (Silver et al. 2017b) and EXIT (Anthony, Tian, and Barber 2017) is difficult for several reasons. First of all, they use function approximation and start with an empty tree every iteration, whereas we just maintain the tree. Maintaining the tree is possible because we only consider smaller problems, this made it easier to investigate whether or not and how different action selection policies influence learning. It also avoided needing to make many adjustments to the basic MCTS algorithm, since in our case our “learned” knowledge just consisted of the added visitations and outcomes within the already built tree, rather than having it stored using function approximation. In contrast, EXIT uses an adapted method from Gelly and Silver (2007), which initially adds a term that favors moves that are thought to be good by the policy. With enough visits, this bias eventually goes to zero. AZ instead uses a method adapted from Rosin (2011), in their changed version it directly changes the exploration term of UCT by multiplying it with the probability the learned policy attaches to the action, biasing the tree policy towards moves thought to be good by the policy. Both methods also use the learned value, either to change the simulation policy (Anthony, Tian, and Barber 2017) or as a replacement for the simulation phase just using the learned value as an estimate (Silver et al. 2017a; 2017b).

Another difference is that AZ learned from the outcome of the game, in which case a less greedy approach is likely to lead to better results since it learns the value based on the actions it takes. If it takes suboptimal actions due to exploration from the action selection policy this leads to learning wrong values some of the time, making the procedure less efficient. Since they still added a lot of exploration, especially in the early phase of the game, and added additional noise for exploration it could be that an action selection policy with a temperature \(0 < \tau < 0.5\) could make the method more efficient. Or possibly learning not from the game outcome but the values accumulated during MCTS, allowing for more exploration in the action selection policy.

The annealing of the number of cycles after each action selection of our method bears some similarity to the idea in (Silver et al. 2017a) (2017a), where clearly lost games are stopped early. Closer to the leaf nodes fewer actions are required to obtain a good estimate of the value, that is likely why without the annealing rate we obtained worse results during initial experimentation, it spends too much time around the leaves after already obtaining a good estimate. It is likely that the method can be improved upon by implementing rules to determine how many cycles should be performed in a state, and whether or not it is worth the time to spend in a state. In the worst case when it just spends all the time in the root is just MCTS.

**Conclusion**

Inspired by recent RL methods that combine function approximation and MCTS and learn through self-play, we have proposed a sample based planning method based on MCTS. We investigated the effect of four different action selection policies based on statistics in the MCTS tree. We showed that the chosen action selection policy, in combination with a temperature setting that controls exploration, has an effect on the speed of learning and the final strength of the policy. This suggests that methods using function approximation in combination with MCTS to learn policies should take into account that their action selection policy can alter both the final strength as well as its learning progress.

On itself our proposed method in itself was able to learn faster than standard MCTS in our experiments, suggesting it is an interesting method to study further in its own right.

**References**


Appendices

2.1. Paper supplement

In this supplement we give a short comparison of some experimental results with different settings of the number of cycles $C_s$ and annealing rate $AR$. We compare the results of our method for three different settings, i.e.:

- Number of cycles $C_s = 50$, with annealing rate $AR = 5$.
- Number of cycles $C_s = 50$, without annealing ($AR = 0$).
- Number of cycles $C_s = 80$, with annealing rate $AR = 10$.

Looking at the value policy, we can see from comparing Figure 2.1 to Figure 2.2, that without annealing the method performs worse for all the different settings of $r$. We speculate this happens because the convergence near the leaf states happens fairly rapidly, which means that additional time spent in the subtrees yield little results, and as thus hampers the efficiency. Comparing Figures 2.1 and 2.3 we see that the different amount of cycles and annealing rate still yield similar results. There probably is some optimum setting which would depend on the problem as well. In addition, some heuristic to determine whether or not additional time in a subtree would be helpful as well, perhaps based on the rate of change in the value with additional evaluations.

In the remainder of the figures for the other action selection policies, i.e. Figures 2.4, 2.5, 2.6, 2.7, 2.8, 2.9, 2.10, 2.11, and 2.12, we observe the same trends as for the value policy.

![Figure 2.1: Value policy results, $C_s = 50, AR = 5$.](image)
Figure 2.2: Value policy results, $C_s = 50, AR = 0$.

Figure 2.3: Value policy results, $C_s = 80, AR = 10$.

Figure 2.4: Visits policy results, $C_s = 50, AR = 5$. 
2.1. Paper supplement

Figure 2.5: Visits policy results, $C_x = 50, AR = 0$.

Figure 2.6: Visits policy results, $C_x = 80, AR = 10$.

Figure 2.7: UCB policy results, $C_x = 50, AR = 5$. 
Figure 2.8: UCB policy results, $c_x = 50, AR = 0$.

Figure 2.9: UCB policy results, $c_x = 80, AR = 10$.

Figure 2.10: LCB policy results, $c_x = 50, AR = 5$. 
Figure 2.11: LCB policy results, $C_z = 50, AR = 0$.

Figure 2.12: LCB policy results, $C_z = 80, AR = 10$. 
2.2. Introduction

In artificial intelligence research, games offer interesting challenges that allow for testing of different methods. What is interesting is that games like chess or Go are too difficult to solve, in the sense that we can not be sure we know the optimal move in every possible state. This is because in a game of chess a player can on average choose between 30 actions each move, and since games often last to 80 moves, this would result in a search tree of about $30^{80}$ nodes (which is a lot) [1, 9]. Therefore, to obtain good solutions to this kind of problems we need to able to come up with good actions, even when it is not feasible to get to the optimal decision [9].

One algorithm that has had a lot of success (particularly in Go, [3]) is Monte Carlo Tree Search (MCTS) [4], often with a UCT strategy [5]. Originally created to improve play in the game of Go, it has been used in a number of other domains, such as security, physics simulations, bus regulation [3], and RNA inverse folding [15]. It is an algorithm that can be used without requiring a lot of domain knowledge and is anytime.

Recently, a new method has been used that is a combination of MCTS with reinforcement learning (RL) [10–12]. This combination has been able to achieve great results in games like chess, Go, and Shogi, even beating a world champion in the game of Go [10]. Given the success of this combination in the game of Go using a limited amount of domain knowledge, it is an interesting method to study further.

2.3. Overview

In order to use MCTS in games like chess, Go, and Shogi we want to model them as a Markov Decision Process, which we will first discuss in Section 2.4. One of the most popular [3] algorithm for MCTS is the Upper Confidence Bounds 1 applied to Trees (UCT) algorithm [5]. The UCT algorithm is based on the Upper Confidence Bounds 1 (UCB1) algorithm which is a method for optimally solving the multi-armed bandit problem. In Section 2.5 we will discuss the multi-armed bandit problem and UCB1, following this we will discuss the MCTS and UCT algorithms in Section 2.7.

2.4. Markov Decision Processes

We can view games, and other kinds of planning problems or tasks, as a sequence of decisions. Consider the process of interaction between an environment and an agent in 2.13. We define an agent as an entity that makes decisions, while the environment is that with which the agent interacts. The environment will typically be some kind of planning problem or game, while the agent can be some algorithm or a human. In this section, we will mostly use the notation and information from [13].
2.4. Markov Decision Processes

2.4.1. Notation
The environments we are interested in can be modeled as Markov Decision Processes (MDPs). We define a MDP as a tuple $<S, A, P^a_{ss'}, R^a_{ss'}>$ where:

- $S$: A set of states, with $s_0$ being the initial state of the environment.
- $A$: A set of actions, with $A(s) \subseteq A$ a set of actions that are allowed in state $s \in S$.
- $P^a_{ss'}$: A transition model that determines the probability of reaching state $s'$ if action $a$ is chosen in state $s$, i.e. $P^a_{ss'} = Pr(s_{t+1} = s'|s_t = s, a_t = a)$.
- $R^a_{ss'}$: A reward function, giving the expected reward for taking action $a$ in state $s$, with following state $s'$, i.e. $R^a_{ss'} = \mathbb{E}[r_{t+1}|s_t = s, a_t = a, s_{t+1} = s']$.

A MDP is a sequential decision problem for a fully observable (stochastic) environment that adheres to the Markov property, that is the results of actions in the present state only depend on our present state and not on past states [9], i.e. $Pr(s_{t+1}, r_{t+1}|s_t, a_t) = Pr(s_{t+1}, r_{t+1}|s_0, a_0, r_1, s_1, a_1, ..., s_{t-1}, a_{t-1}, r_{t-1}, s_t, a_t)$.

The implication of the Markov property is that when we deciding on what action to take we only have to look at the current state.

2.4.2. Return
We can consider problems to be either episodic or continuous. Continuous tasks continue ad infinitum, while episodic tasks end at some point. For instance, a game like tic-tac-toe or walking through a maze can be viewed as episodic. At some point, we reach a state in which the game ends, or we find the end of the maze. We call these final states terminal states. Each episode ends when a terminal state is reached. We are mostly interested in episodic MDPs, because we deal with games. A terminal state is a state in which an episode ends directly.

The goal of an agent in an episodic MDP is typically to maximize its expected return. We define the return as follows: $R_t = r_{t+1} + r_{t+2} + ... + r_T$, where $T$ is the final time step, when a terminal state is reached.

For continuous MDPs we define the return slightly differently: $R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$, where $\gamma$ is a parameter in $0 \leq \gamma \leq 1$ that we call a discount rate. When $\gamma = 1$ the return is the same as the episodic return. A discount rate $<1$ is useful in the continuous case since if the agent would frequently receive even small rewards the expected return could easily be infinite.

2.4.3. Policies & Value Functions
At each time step, the agent decides which action to choose based on the current state, we call this mapping from states to actions a policy. A policy, $\pi$, tells us for every state $s \in S$ which action to take with what probability, that is $\pi(s, a) = Pr(a|s)$ for $a \in A(s)$.

Given a MDP and a policy $\pi$ we can define the value function $V^\pi(s)$, the expected sum of rewards an agent following policy $\pi$ will obtain when starting in state $s$, as follows: $V^\pi(s) = \mathbb{E}[R_t|s_t = s] = \mathbb{E}_\pi \left[ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|s_t = s \right]$. 
where $\mathbb{E}_\pi$ is the expected value when the agent is following policy $\pi$. The function $V^\pi$ is the state-value function for policy $\pi$, i.e. for every state $s \in S$ we have $V^\pi(s) \in V^\pi$.

We can also define a similar notion, $Q^\pi$, the action-value function for policy $\pi$. Here we have $Q^\pi(s,a)$, the expected return when starting at state $s$ with action $a$ and following the policy $\pi$ afterwards, i.e.:

$$Q^\pi(s,a) = \mathbb{E}[R_t | s_t = s, a_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a\right].$$

For finite MDPs, i.e. MDPs that have a finite set of states and actions, there always exists an optimal policy $\pi^*$ [13]. A policy is optimal if its expected return is the same as or greater than that of every other policy for all states in the MDP. We define the optimal state-value function $V^*$ as follows:

$$V^*(s) = \max_\pi V^\pi(s),$$

for all states $s \in S$. Optimal policy also have an optimal action-value function $Q^*$, defined as follows:

$$Q^*(s,a) = \max_\pi Q^\pi(s,a),$$

for all states $s \in S$ and all actions $a \in A(s)$. To show the relation between $V^*$ and $Q^*$ we can write $Q^*$ as a function of $V^*$, i.e.:

$$Q^*(s,a) = \mathbb{E}[r_{t+1} + \gamma V^*(s_{t+1}) | s_t = s, a_t = a].$$

### 2.5. Bandit Problem

One problem that is perhaps the simplest instance of the exploration-exploitation dilemma is the multi-armed bandit problem [2]. The multi-armed bandit problem is a sequential decision-making problem, there are multiple one-armed bandits (slot machines). Each bandit has a different (unknown) reward distribution. Each decision is a choice of pulling the arm of one of the bandits, upon which we sample a reward from its reward distribution. The goal is to maximize the expected reward per draw. In other words, we want to pull the arm with the highest expected reward as often as possible. In order to maximize this, one has to play the bandit that yields the maximum expected reward. However, since the reward distributions are not known, there is a trade-off between exploring bandits that seem suboptimal, but might be optimal, and exploiting the bandit that is currently believed to be optimal. This tradeoff is known as the exploration-exploitation dilemma.

#### 2.5.1. Definition

In a $K$-armed bandit problem we have random variables $X_{i,n}$ for $1 \leq i \leq K$ and $n \geq 1$, with each $i$ corresponding to a one-armed bandit [2]. Successive plays of bandit $i$ give $X_{i,1}, X_{i,2}, \ldots$ which are independently and identically distributed (i.i.d.) according to some unknown distribution that has unknown expectation $\mu_i$. Rewards across machines are independent as well, that is: for each $1 \leq i < j \leq K$ and $s,t \geq 1$, $X_{i,s}$ and $X_{j,t}$ are independent [2].

#### 2.5.2. Regret

A measure that is often used to determine the success of a policy in this domain is the regret, the loss due to not picking the optimal bandit at all times [2]. After choosing $n$ actions, the regret is defined as:

$$R_n = \mu^* n - \mu \sum_{j=1}^{K} \mathbb{E}[n_j]$$

where $\mu^* = \max_{i \leq K} \mu_i$, $n_j$ is the number of times bandit $j$ has been chosen in the first $n$ actions, and $\mathbb{E}$ denotes expectation. We use expectation because we will look at policies that choose bandits based on the average sampled reward obtained from each bandit since those are based on samples from unknown distributions we assume they are stochastic. So, regret is
the expected loss due to not picking the bandit with the maximum expected reward every time.

2.5.3. Solving the exploration-exploitation dilemma

In order to obtain a policy that minimizes the regret, it is important that all bandits have a non-zero probability of being chosen, to make sure that the optimal bandit is not missed due to another (suboptimal) bandit temporarily providing promising rewards [3]. In [7] it was proven that, given some mild assumptions on the reward distributions [2], it is not possible for the regret to grow slower than $O(\ln n)$. These assumptions are that the rewards are within $[0, 1]$, they are i.i.d., and that the rewards across bandits are independent [2].

Thus for those reward distributions, a policy is said to solve the exploration-exploitation dilemma if its growth of regret is $O(\ln n)$ [3, 5]. We will now consider the Upper Confidence Bound 1 (UCB1) algorithm which is able to achieve logarithmic regret, not just asymptotically but uniformly over time, without any preliminary knowledge about the reward distribution except that the rewards should be within $[0, 1]$ [2]. That the regret is logarithmic uniformly over time means that the regret is upper and lower bounded as a function of $n$ in $O(\ln n)$, whereas previous algorithms were only able to achieve asymptotically logarithmic regret, i.e. regret in $O(\ln n)$ as $n \to \infty$ [2, 7].

2.5.4. UCB1

The UCB1 algorithm starts with initially choosing each bandit once. Following this initialization it plays the bandit $j$ that maximizes:

$$UCB1 = \bar{X}_j + \sqrt{\frac{2 \ln n}{n_j}},$$

where $\bar{X}_j$ is the average reward obtained from playing bandit $j$ so far, $n$ the total number of plays that have been done, and $n_j$ the number of times bandit $j$ has been played.

The $\bar{X}_j$ term encourages exploitation of the bandits we believe to have the highest rewards. The $\sqrt{\frac{2 \ln n}{n_j}}$ term encourages exploration since it is higher for less frequently played bandits and ensures that all bandits will occasionally be played. Together they provide an upper confidence bound on our estimated value since it is based on the probability that the estimated reward falls within the one-sided confidence interval of the true expected reward [2]. This exploration term is defined in a way so that if a bandits sampled rewards were i.i.d. then it follows from Hoeffding’s equality that the following two inequalities hold [5]:

$$\mathbb{P}(|\bar{X}_j - \mu_i| \leq \sqrt{\frac{2 \ln n}{n_j}}) \leq n^{-4},$$

$$\mathbb{P}(|\bar{X}_j - \mu_i| \leq \sqrt{\frac{2 \ln n}{n_j}}) \leq n^{-4}.$$

That is, as $n$ grows the probability that $\bar{X}_j$ lies within $\mu_i \pm \sqrt{\frac{2 \ln n}{n_j}}$ very quickly approaches 1.

2.6. UCT

Whereas in the Multi-armed Bandit Problem we had just one state where we wanted to find the optimal action, we will now look at finding good policies for MDPs with more than just one state. We are interested in finding (near) optimal actions for MDPs for which we have a simulator, that is we can simulate how taking actions in states would play out in our MDP. We will focus on the UCB1 applied to trees (UCT) algorithm [5, 6].

The main idea of the algorithm is that each state can be viewed as a multi-armed bandit problem [5, 6]. This allows the UCB1 policy to be used for selectively choosing actions in each
state, and in this way the most promising part of the MDP can be explored more, allowing for better performance. In order to do this, the suboptimal state-action pairs should be identified quickly. A state-action pair \((s, a)\) is suboptimal if for the same state there is a state-action pair \((s, a')\) \((a' \in A(s), a' \neq a)\) that has a higher expected reward. To do this, UCT trades-off between greedily choosing the current best-estimated state-action pairs and exploratory choosing of suboptimal looking state-actions pairs. In this way, it attempts to obtain good estimates of the best state-action pairs while simultaneously attempting not to miss other potentially good state-actions pairs.

Concretely, when selecting an action at time \(t(< L)\) UCT chooses the action \(a_t\):

\[
a_t = \text{argmax}_a (Q(s_t, a) + U(s_t, a))
\]

with

\[
U(s, a) = \epsilon \times \sqrt{\frac{\ln \sum_b N(s, b)}{N(s, a)}},
\]

and \(\epsilon\) an exploration constant. \(Q(s, a)\) can be seen as an exploitation term while \(U(s, a)\) is an exploration term. \(U(s, a)\) favors state-action pairs that are less visited while \(Q(s, a)\) favors state-action pairs that have a high estimated value. With more total visits the term \(U(s, a)\) becomes smaller and the method favors higher valued state-actions. \(U(s, a)\) makes sure that every state-action pair is eventually visited.

### 2.7. Monte Carlo Tree Search

MCTS is a tree search algorithm that uses Monte-Carlo evaluation, i.e. averaging the outcome of multiple samples, for estimating values of nodes [4]. One attractive quality of the algorithm is that it is anytime, it can be interrupted at any moment and still give a valid solution at the root [3]. This is because it works iteratively and after each iteration values are backpropagated through the tree so that there always is an estimation of the best move at the root. Multiple iterations are expected to improve the estimates.

MCTS starts with an empty tree and is initially given a root state node, after which it iteratively performs the four steps shown in ??, we refer to one iteration of these four steps as a MCTS cycle. The steps work as follows:

1. **Selection:** Starting from the root node a child node is selected recursively, according to the tree policy, until we reach a leaf node. A leaf node is either a terminal node, i.e. a node corresponding to a terminal state, or a node that is expandable. A node is expandable if it non-terminal and has actions available that do not have a corresponding edge in the tree. If the leaf node is a terminal node step 2 and 3 are skipped.

2. **Expansion:** One or more child nodes, that correspond to states not yet in the tree, are attached to the leaf node.

3. **Simulation:** Starting from the leaf node a default policy recursively chooses an action, according to the default policy, until a terminal state \(s_i\) with an outcome reward \(R(s)\) is reached. If the leaf node \(s_i\) itself is terminal, we use the outcome reward \(R(s_i)\) from this state. The most basic form of a default policy is randomly choosing one of the available actions [3].

4. **Backpropagation:** The reward \(R\) that was obtained in the simulation step is backpropagated through the nodes that were selected in step 1, updating their statistics.

One of the most popular tree policies [3] is the Upper Confidence Bounds 1 Applied to Trees (UCT) algorithm [5]. The UCT algorithm combines tree search with the Upper Confidence Bound 1 (UCB1 [2]) strategy, which is a strategy for selecting arms in a multi-armed bandit problem. The main idea of the algorithm is that, by selectively choosing actions, the most promising part of the tree can be explored more, allowing for better performance [5]. In order to do this, the suboptimal state-action pairs should be identified quickly. A state-action pair \((s, a)\) is suboptimal if for the same state there is a state-action pair \((s, b)\) \((b \in A(s), b \neq a)\) that
has a higher value. To do this, UCT trades-off between greedily choosing the current best-estimated state-action pairs and exploratory choosing of suboptimal looking state-actions pairs. In this way, it attempts to obtain good estimates of the best state-action pairs while simultaneously attempting not to miss other potentially good state-actions pairs.

Concretely, in the selection phase in each state \( s \) the action that maximizes \( Q(s, a) + U(s, a) \) is selected, where

\[
U(s, a) = c \sqrt{\frac{\ln \sum_b N(s, b)}{N(s, a)}},
\]

and \( c \) an exploration constant. \( Q(s, a) \) can be seen as an exploitation term while \( U(s, a) \) is an exploration term. \( U(s, a) \) favors state-action pairs that are less visited while \( Q(s, a) \) favors state-action pairs that have a high estimated value. With more total visits the term \( U(s, a) \) becomes smaller and the method favors higher valued state-actions. \( U(s, a) \) makes sure that every state-action pair is eventually visited. [5] have shown that for a finite horizon MDP with rewards scaled to lie within the interval \([0, 1]\), UCT converges to the optimal action in the limit, for an exploration constant of \( \frac{2D}{\sqrt{2}} \).
3.1. Reinforcement Learning

In this section we will focus mainly on Temporal-Difference (TD) learning, since this is the idea central to reinforcement learning [13]. This method learns from experience and bootstrapping by sampling its environment. Bootstrapping is the idea of updating estimates based on other estimates [13]. In general reinforcement learning methods can focus either on predicting the value function $V^\pi$ for a policy $\pi$, or on finding an optimal policy $\pi^*$. 

3.1.1. Temporal-Difference Learning

The simplest method is $TD(0)$, depicted in algorithm 1. At time step $t$ the method takes an action $a$ in state $s_t$ according to the policy $\pi$ it is evaluating and observes the reward $r_{t+1}$ and next state $s_{t+1}$ from the environment. It then uses the next state and reward to update its estimate $V(s_t)$ as follows:

$$V(s_t) = (1 - \alpha)V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1})],$$

where $\alpha$ is a step-size parameter with $0 < \alpha \leq 1$.

Algorithm 1 TD(0)

1: function TD(0)($\pi$, $\alpha$, $\gamma$)  
2: Initialize $V(s)$ for $s \in S$  
3: while Not converged or stopping criterion not reached do  
4:   $s$ = starting state  
5:   while $s$ is not terminal do  
6:     $a = \pi(s)$  
7:     $s', r = \text{do action } a, \text{ observe } r \text{ and } s'$  
8:     $V(s) = V(s) + \alpha[r + \gamma V(s') - V(s)]$  
9:   end while  
10: end while  
11: return $V(s)$  
12: end function

3.2. Deep Reinforcement Learning

The combination of using deep neural networks for Reinforcement Learning (RL), basically RL with function approximation where the function approximation is done by a deep neural network, has been coined deep RL. Since playing video games is often a frustrating and exhausting task, a lot of research into deep learning has focused on helping us out in this area, in particular a large set of Atari games is often used as a sort of benchmark.
Perhaps the first deep RL algorithm that really had success was the deep Q-learning method [8]. The input is typically represented as some kind of image which is fed to a Convolutional Neural Network (CNN), followed by some fully connected layers and then an output layer. See 3.1 for a schematic example.

### 3.2.1. Problems using nonlinear function approximation
Just throwing in a CNN was not all it took, since RL can be unstable or diverge when a nonlinear function approximator is used as a representation of the Q-function [14]. This has several causes [8]:

- The sequence of observationexperience has correlations, which means the network could just learn a specific sequence rather than just action-values. That is, instead of the value of an action at a state it would learn the value of the entire sequence of actions.
- Small updates to Q could significantly alter the policy and thereby the future data distribution.
- There is typically a strong correlation between the action-values the network has and the values it is learning (target-values), since these depend on its own prediction.

To address these issues they added two ideas to the algorithm [8]: experience replay and a separate target network.

### 3.2.2. Experience Replay
Experience replay is a biologically inspired method that is able to remove the correlations in the observation sequence by randomizing over the data, also resulting in a data distribution that changes more smoothly. Experience the agent obtains during learning is stored in a (limited size) replay memory, each element in the memory consists of the state $s$, action $a$, reward $r$, and the next state $s'$ that was reached after taking action $a$ in state $s$. Periodically the algorithm randomly samples a subset from the replay memory, computes targets values for every experience tuple, and uses this to perform gradient descent. This has several benefits over normal online learning:

- Normally every experience tuple is only used once in learning. However, because of the frequency and the size of the random samples that are used during learning with experience replay, most experience tuples are used multiple times. Since the target values are calculated at the time of actually using the samples, i.e. the target values will not be "wrong", this makes this method more data efficient.
3.2.3. Target Network
Rather than just one network they use two networks, one of which is periodically updated to equal the other. The network that is periodically updated is called the target network, and this is used to provide the continuously updated network with target Q-values. This reduces the correlations with the target.

3.3. Thesis Proposals
Mostly just ideas for things I wanted to research, some of which I have shared.

3.3.1. Tuning AlphaZero
Basically, finding out what all the settings/parameters do and how we can improve on the method to make a "stronger" agent.

Notes
What I want to have is some idea of what “goals” I want to reach, why do I do the research that I do?

My initial idea of trying to find out how to use/tune/change the Alpha Zero “algorithm”s hyperparameters to train an agent in the quickest way and in such a way that it becomes the “strongest” player was lacking a bit of that, at least in my eyes. Because, while the results would be interesting to me personally, I don’t really see the “larger” or practical reason to it, because I feel/see that it would only be useful for a Alpha Zero system/algorithm.

My current idea/"ideal" of trying to construct the “perfect” tic-tac-toe player ties more into the DQN learning that to me seems prevalent in the recent research by companies such as Google (Deepmind) and OpenAI. Here is it interesting because if you can’t even create a “perfect” tic-tac-toe player, because it fails against a relatively weak opponent due to ending up in a relatively unknown state, then I think this has implications for learning by using DQN. If you’d want to use a DQN in the “real world” you don’t want it to fail on relatively easy but unfamiliar states. This raises a question on how to prevent this, which might be related to “catastrophic forgetting” and similar known problems, which I still have to dive more into, which (I think) Deepmind claimed not to have problems with in their Alpha(Go) (Zero) algorithms.

3.3.2. From AlphaZero to hero
In the quest to creating truly intelligent systems, Google Deepmind has been able to greatly impress with its general reinforcement learning algorithm AlphaZero. It is a combination of a neural network (NN) and Monte Carlo tree search (MCTS). At the start of self-play the MCTS isn’t able to provide good action and value estimations since it relies on the predictions of the NN, which are random at the start. Currently, AlphaZero uses the binary game outcome as a signal for the value. This implicitly implies that we should be able to tell the outcome of the game from the first move onwards. While this makes sense in a solved environment like tic-tac-toe, it makes less sense in more complex games like chess.

This thesis aims to improve the AlphaZero method in two ways. 1) It aims to improve the efficiency of the algorithm by starting out training with only the NN and adding MCTS at a later stage. The goal is to let the algorithm learn when it should start using MCTS. 2) Using the probability distribution over actions to adjust value signal, and trying to combine this with advantage learning, in order to speed up training, and possibly obtain better performance. The goal is to show that with the improvements the algorithm is more efficient in Zero-sum games of perfect information, and more readily applicable to other problem domains.

3.3.3. Ideas
Alpha Zero is an algorithm that, given a lot of time and processing power, has the ability to train an agent by self-play up to a level that is "superhuman”. It does so by using a combi-
nation of a neural network (NN) and Monte Carlo tree search (MCTS).

The system starts with completely random play. Since the MCTS planning makes use of the prediction of the NN (which are basically random), this adds unnecessary overhead. So at the start, it would make more sense to do the self-play without the MCTS, since the MCTS won’t be able to improve upon random action selection and random state value estimation. By only using the NN and "regular" reinforcement learning, the system is also able to learn to play. But probably not at the level of the Alpha Zero system. But at what point should we add the MCTS to learn better policies and value estimation? Do we need to use it for every position, or can the network learn at what point it needs to use MCTS to improve its moves?

Using the binary game outcome as a value for all the states that appeared in the game would make sense if we were certain that it is already clear from the first move whether or not someone is going to win (or draw), given perfect play. And a case could be made for certain solved games that this holds, e.g. tic-tac-toe. However, for a game like chess or Go it seems unlikely that this is the case. And if a few moves before ending the game the system is not really sure yet of whether or not it is going to win and also not sure about which move it is going to play, maybe the outcome isn't certain either. So it would make sense to use some sort of “forgetting” for the value.

While in settings like Go and Chess and other two player games we know that we are always working towards a given “ending”, in other settings where we could use sequential planning this might not be the case. Consider a maze where “bumping” into the wall is a valid move. In this situation, we could potentially endlessly move in the wrong direction. And even if we eventually hit the reward, most of the actions could be very useless, and only actually moving onto the reward was useful. So here neither the action nor value that the alpha zero provides is useful in a lot of cases. It also doesn’t reward reaching the end state faster. And in a maze setting it could be that we know that a perfect player would always win, in that case, the value function should eventually just give a value of 1 in every possible state, but this then won’t be of any help in the Monte Carlo tree search, since every state will be equally valuable. And the actions aren’t necessarily the best either, since moving into the wall is not punished since it results in an equally valuable state, so as long as we still reach the end this would then be stored as a good move.

So if we could find a way to transform the training signal returned by the MCTS in such a way that it adheres to the Bellman equation, we might be able to get more reliable (and faster?) results, that are also usable in situations outside of two-player games. Another question is, how far do we have to look ahead? I.e. how many simulations do we need in MCTS before making a move? We can look at tic-tac-toe by checking out the probability of making a “losing” move as a function of the number of simulations in MCTS. If we are the second player and the opponents has made one move there are already several plays we could make that would always result in a loss for us, if the opponent plays perfectly. By looking at this we can get an idea of how far we should look ahead to all but guarantee “perfect” play.

Another point that does not really make sense, is the choice of tau and setting it to -> 0 after X moves. This conflicts with the choice of just using the binary outcome as the value even for the first moves, since by using a tau for some exploration seems to suggest that there might not be a perfect move, or you don’t know it. It was shown in thinking fast and slow that setting tau -> 0 was not ideal as a learning signal. Due to symmetries in some cases two actions could also be equally good, in which case it also doesn’t make sense to give a learning signal that says that one of them is “perfect”. Now of course it could be that if we train the system longer, it becomes better and we need less MCTS simulations to play perfectly. We can look at the change in evaluation of the positions as a function of time, and maybe give some estimate of the reduction in MCTS simulations we need to play perfectly as a function of the time. Using a “third head” to say something about the familiarity of the state, to focus attention on unfamiliar states, to use “more” mcts, to actually use mcts?
3.3.4. Deep Exploration

Within Reinforcement Learning (RL) it is important to use a form of exploration during training to get to know the environment. A simple method such as $\epsilon$-greedy is often used for this purpose, where the "best" action is chosen with probability $(1 - \epsilon)$ and a random choice is made with probability $\epsilon$. This is a method that often works fine, but recently several exploration methods have been devised for deep reinforcement learning that work a lot better (refs 1, 2, 3). Simply put, these methods lead exploration by estimating how "familiar" the network is with a state to which it can go, states that the network is less familiar with get a "bonus" when selecting the action. In a sense, this is an "active" way of exploration, by looking for states that the network is less familiar with.

Within the AlphaZero algorithm there is exploration in training, but it does not really have an "active" exploration yet. In the AlphaZero algorithm, I want to add this "active" exploration (inside the Monte Carlo Tree Search (MCTS)) to see if this can lead to faster and / or better results. What it would deliver is an algorithm that takes into account the fact that the network is still learning about the game / environment during training.


Outro

I still have to read in the "active" exploration methods, there is a lot of theory behind it. The interaction with active exploration within an RL + MCTS algorithm seems, to me at least, an interesting research direction.

The "familiarity" of a state could possibly also be used to decide how many MCTS simulations to use to choose an action, if you are already in a known state, you are probably also quite certain with less simulations about the "accuracy" of your move.

Notes / changes

After a brief exploration of the "deep" and "normal" exploration literature, I came to the conclusion that there is still too much literature in that direction to do something "quickly" with it. I still came across this statement that I liked: "Equation (1) embodies in one line the major ideas of Bayes, Ockham, Epicurus, Turing, von Neumann, Bellman, Kolmogorov, and Solomonof" (in "A Monte-Carlo") AIXI Approximation "). All the contraptions I encountered had prompted me to look again at what my algorithm actually did, and whether that could not be simpler. That ultimately resulted from this:

I realized that when you remember the visits and values in a table for all states, you might as well just remember everything in the tree. With that adaptation to the algorithm, I realized that you could actually describe it as a kind of UCT in UCT (or UCT on top of UCT or something like that). That makes UCT itself a good candidate to compare the method. (Just have to think about whether I can think of situations where one method would work better than the other and vice versa. "UCT in UCT" might be better suited for a function approximator RL method while UCT is working better in most minor problems. Could also be that "UCT in UCT" works better when it is important to learn over the entire state-space, while UCT is better when there is one clear direction / solution.)

If your problem is too big to build / store a full step and you want to use a function approximator, you can adapt both UCT on top or UCT and UCT to an "approximated" version
that learns in a reinforcement learning way. These two methods are then good to compare with AlphaZero.

So my plan is now:

• Describe UCT in UCT and prove that it preserves the properties of UCT (I think so)

• Empirical testing of the UCT in UCT and comparison with UCT (perhaps also with a comparable AlphaZero version?)

• Approximate UCT in UCT and approximated UCT description

• Empirical testing of the approximated methods and comparison with each other and AlphaZero

• Prove that, given a few assumptions on the behavior of the function approximator, they both converge towards an optimal solution

• To see to what extent neural network meets our assumptions as an approximator

3.3.5. Three ideas

AlphaZero is a reinforcement learning algorithm that in Zero-sum games of perfect information learns by self-play using MCTS guided by a neural network. The neural network learns values and a policy from the results of the MCTS during self-play.

One
They claim that the algorithm results in “rapid improvement and precise and stable learning”. Is using MCTS to learn a policy and values more rapid, stable and precise compared to “regular” policy and value learning? (and what is the effect of the number of “searches” per move when using MCTS). How well and quickly does the network learn the values and policies?

Two
There are a number of ways in which AlphaZero added exploration, however the motivation and effects are not always clear:

• The variation on the PUCT algorithm (“priors”)

• Choosing actions proportionally to their visit count

• Adding Dirichlet noise to the prior probabilities in the root node

What are the effects that these additions have on learning the values and policies when compared to a “basic” version of MCTS (+ UCT) and RL?

Three
There are a number of ways in which AlphaZero added exploration, but how important is exploration in the combination of MCTS + RL? Two simple strategies to explore are epsilon greedy (frequently used in (deep) reinforcement learning) and Boltzmann exploration (used by Alpha(Go) Zero).

Since the network is learning about the environment one “natural” way to add exploration is to add a term of “uncertainty” of the model. Some recent methods that do this include “bootstrapped dqn” and “bayesian dqn”.

Can we use the network itself to estimate it’s unfamiliarity/uncertainty with a state (and use this as an exploration term within MCTS)? Two ideas of adding (and learning about) uncertainty are:

• Adding an additional “head” to the network, which is trained on the difference between the actual reward and the value predicted by the “value” head of our network.
• Instead of using one head to estimate a value between -1 and 1, we can use three heads, each to estimate the probability of a specific outcome (win, draw, loss). In this case we can use the closeness of these three values as a measure of uncertainty, the closer they are to each other the less certain the network is of the outcome.

3.3.6. AlphaZero & DQN
Set up for game environment: First implement a DQN version with value & policy head, or more complicated?

At the start of learning, the agent plays randomly, since the network is still untrained, and the estimates for the actions and the values are therefore random (worthless). At this moment using MCTS only ensures that it takes longer to generate the training data. So it should be able to progress faster at the beginning by using only the NN to train.

In addition, it may be that you can quickly reach a certain "maximum" level by using only the NN to play / learn, and only later to hang the MCTS to learn better moves, to learn to plan.

Value of (-) 1 yield makes no sense, can not already say from the first step that you will win / lose 100%?

Works here probably "ok" because everything "goes forward", e.g. butter-cheese-and-eggs is always finished after 9 moves (or earlier). But for example, walking through a maze probably works less or just not. Imagine that the agent starts next to the end of the maze, and then first executes an x number of actions that do not move to the goal, followed by the step to the goal. Then, it is then pretended that all these actions have led to the agent having reached the goal, even if he has not moved in those x steps (e.g., by running up against a wall).

Combination of MCTS for learning actions with q-learning for learning / approaching the value of the state?

It is still difficult to see how MCTS can combine with Q-learning.

3.3.7. AlphaZero training
How long does it take to go from baseline / worse to expert?

• How much time does it cost
• How many "moves" used to learn / training steps

How important are the following options / settings?

• Step history
• Mini-batch size
• MCTS simulation per step
• Tau (exploration)
• Residual network size
• Memory self-play games
• Momentum
• Regularization constant
• Epsilon
• Dir alpha
• Resignation threshold

• Cpuct

50 games vs baseline and expert use, every few steps. Save results with "time" + training steps.

Ready when always draw against expert (will that work? E.g. could that expert is not really expert yet -> not always draws).

Start with some number of MCTS simulations per step.

Learning rate, should we use annealing?

MCTS as a policy improvement operator (since move probabilities $p_i$ are better than raw move probabilities from nn)

MCTs as policy evaluation operator (using sample from self-play with improved MCTS-based policy for selecting moves, and using game winner $z$ as sample of the value)

Does MCTS stabilize learning?

Does use of MCTS improve the agent policy and value evaluation?

How fast is it?

Value iteration, policy iteration, modified policy iteration, combined value-policy iteration?

Effect of the exploration term / Cpuct, see Figure 3.2.

3.3.8. Tabula Rasa

What exactly do they mean by learning "tabula rasa"?

Tabula rasa == without human data, guidance or domain knowledge beyond game rules?
"Here we introduce algorithm based on reinforcement learning, without human data, guidance or domain knowledge beyond game rules."
Is not there already domain knowledge decided in MCTS? Is it already "perfect" to solve these kinds of games / problems in the limit? Dirichlet noise is based on typical number of moves, also knowledge outside the game rules?

3.3.9. Questions, ideas

AlphaGo Zero uses 1600 Monte Carlo rollouts per step during training

- What is the effect of more / less monte carlo rollouts on the training speed (how fast the ai becomes "better") and on the final strength (max elo)
- Does it make sense, just like the learning rate, to also reduce this number over time (or even increase it, at the beginning, randomly?)

AlphaGo zero uses a fairly large and complex neural network

- Effect of network size / complexity on training speed and end "strength"

In actual play, the programs get x seconds / minutes per turn

- What are the effects of this on the playing strength of the different trained models?
  - E.g. Less time a bigger negative effect on more complex models?
  - Or, on simple models, a greater negative effect, are more dependent on the MCTS than on network evaluation?

Does it also work on other complete information games without a board (eg?)? And to play incomplete information?

Things to do

- View implementations alpha (go) zero, repo clones, scan through code, run some tests
- Testing "work out" + drawing up the protocol
- Reading relevant (non-neural network based) literature
- What to use to check / compare "level", standard MCTS rollouts? + play against each other (random play as baseline ), Bayes elo?

Important to read / understand all relevant literature first? "Additional" techniques (eg batch normalization, initialization, resnets, etc.) also elaborate / explain?

Or at the same time / in advance already working on experiments and the like?

Effect of resizing at 95% chance of loss?

8 moves history, effect of this?

Checking literature, has mcts been used to train in reinforcement learning?

Alpha go zero end reinforcement learning section review literature

Effect of no more "evaluation" of new agents?

Is initialization of 1st layer in cnn also random? If so, what if initialization is "random" but in such a way that the "distance" between the filters is maximized?
Using a trained model for another game, how transferable is it? Can it be made “more” transferable?

Effects of dirichlet noise + scaling with average number of possible moves in game

Effect of the number of possible moves in the position itself?

3.3.10. More Thoughts on AlphaZero

Value of game / end result
The outcome is deterministic the game is “solved”

• If not then we rely on the distribution to move the value evaluation of our network to the “right” value, and using the game outcome might not be the fastest way of converging.

We are always moving towards the “end”

• If not this is likely to not work, if we use a maze environment and change the maze every time, we won’t or only very slowly learn to move towards the endpoint, since we value every state in a successful run equally.

Possibly increases learning speed compared to regular q-step

• Because every memory will have something to learn if the game outcome is a win/loss since value evaluation is probably somewhere around 0 due to initialization.

• Seems likely to have a similar effect as Prioritized Replay Experience.

AlphaZero doesn’t use the reward from the game within the MCTS? (It does, mentioned somewhere in the article)

• At least it doesn’t say it does, it says it only uses the value evaluation by the network

• I think it is likely that using the reward from the game speeds up training and probably gives the same end results

Probability distribution over actions
Doesn’t take into account final state / win-loss

• Does this matter?

• If we made a losing move this will make it more likely to make the losing move again, although this is partially offset by the value evaluation.

• Should be ok since we are most likely to make a move moving use towards a valuable state.

Was shown somewhere that it was better to use distribution rather than a one-hot encoding

• Find out where, was it thinking fast and slow?

What if two (or more) moves are equally good?

• Do we want to move their priors both towards 1 and others down?

• “Less good” moves might be seen as better compared to when only 1 move dominates
When learning a movement like throwing a ball, the end point and angle / speed are most important. Accuracy at the end is top priority. From the start to three quarters of the movement we just need a “good trajectory”, not very detailed. Probably important that it’s smooth in a sense. Reinforcement learning will typically just learn “the best” path, but this is maybe way slower to learn. How to learn a less controlled trajectory?

- Evaluate a sequence of actions, not just one action
- So use a state, 10 actions, endstate thing?
- Control parameter to determine how many actions at once?

Deep exploring
Inside MCTS

- Learn to explore by using it “inside” the probability distribution?
- Only use it as a “later” bonus, to avoid learning it in choosing moves but to let the agent explore
- Probably better for deep exploration

Outside MCTS

- Easier?
- Maybe less deep exploration, since it only uses it when selecting an action, not inside the tree (that is looking further forward)

When is planning most / least useful
Might be difference between training and testing, in testing want to use as much time as allowed to search, not so much in training (probably). Not useful:

- When we don’t know anything about the state we are in, likely we also don’t know a lot about the next states, so might as well “randomly” pick one.
- What if we don’t know anything but are close to a good state, want to search then
- When we are already really really familiar with a state, since we already know which one we want to do.
- When decision doesn’t really matter, how to measure? When we are certain of environment and value evaluation are close together?

Useful:

- When we are vaguely familiar with a state, probably planning has some use since we at least have some information and maybe more about close states
- When close to the end of the game, can we know?
- When decision is important, how to measure?
Learn how “surprised” you are / will be

- Add a head that predicts how “surprised” it will be by a value of a state

- Learning signal would be the difference between the supplied value and it’s prediction, will go -> 0 when familiar

- Can initialize to 1 or something, to show that is unfamiliar at start

- Should hopefully learn to approximate how familiar a state is, and / or how certain the outcome is in that situation

3.4. Experiments

3.4.1. Values & Priors results

Using reinforcement learning we want to learn a value function and a policy. These are then used in the MCTS algorithm in replacement of simulation and to bias the exploration term towards promising moves (as pointed to by the policy).

Here we have created a “perfect” player for tic-tac-toe, by running MCTS starting with the end states and slowly working our way toward the earlier states. In this way MCTS policy and value estimation converged to the optimal mini-max. Each state is valued with approximately -1, 0 or 1 which was rounded toward the correct result. Similarly the policy converged to all the best states, due to starting at the end there already was a good estimation, so it didn’t bias itself towards one of the best moves (in case there were multiple best moves).

As a base we have MCTS with a random rollout policy for simulation. We test in 3 different scenarios:

1. MCTS with using the Prior as a bias.

2. MCTS with the value instead of the random rollout policy.

3. 1) and 2) combined.

We tested this against a perfect player who each turn chooses among the best valued actions, and will thus never lose. And we tested this against the base MCTS + UCT algorithm, which used 50 searches per turn.

We tested this for 10, 20, 30, 40 and 50 searches.

For prior 1.0, the prior for the best move(s) is 1 divided by the number of best moves, and 0 for the other moves. For a prior < 1.0, the prior for the best move(s) is ??, and ?? for the other moves.

From figures 3.3 and 3.4 we see that there is a clear improvement with the number of searches as well as an increase towards the correct prior in draw percentage and rating. Given that the Prior “forces” us to select better moves, which probably give, on average, better simulation results, it makes sense that a stronger prior has a stronger effect. Although with more searches this seems to become less pronounced, since if we increase this high enough we will get optimal play, this makes sense. It shows that learning and applying a correct prior should help performance. In the rating figure we see that with 40 searches all the players are clearly stronger than the standard MCTS with 50 searches. For stronger priors we note that already at 20 searches the method is a lot stronger.

Every state has a value of 1, 0 or -1. For the value we just multiply this with 0.2, 0.4, etc. So we do always correctly interpret a draw state, but there is some variation in how good we are at interpreting the win and loss states. The results here show the same story as for the Prior. At a lower number of searches the value estimation has a more pronounced
3.4. Experiments

Figure 3.3: Performance against perfect player using prior.

Figure 3.4: Performance against MCTS player using prior.

Figure 3.5: Performance against perfect player using value.
Figure 3.6: Performance against MCTS player using value.

When combining both value and prior we can see the results are much stronger, immediately it always draws, and the rating is as good as the best ratings for the other two methods. When using only the prior, the value estimation is randomized due to the random rollout, for lower amounts of searches this means there are chances of deviation in estimating the value, and thus visiting states that are incorrect more often.

When using only a (lower level of) the value, states are more likely to be visited equally often, due to the exploration weighting relatively heavy. Since when choosing where to go we only look at the visitation frequency and not the value, it is relatively more likely to move towards bad states.

When combined these weaknesses are a lot less pronounced, since the prior will reduce the exploration term in less likely candidates while the value function will also make these states less likely. Together they can quickly provide a good estimation of which states to take.

For our method this means that it is not necessary to have a perfect evaluations, we just need to have evaluations of priors and values that are in the right direction. This likely requires learning in a way that facilitates learning about all the “good” moves, since if we only focus on some of the good moves we might only learn about a part of the environment, potentially missing other important parts.

3.4.2. Progress Measurements

1st experiment: Always select with $\tau = 1$, also learn $\tau = 1$ Value outcome as value to learn Use Reward in the end state as the value of the end-leaf leaf

- Study and write AV results
- Play strengths 1st 10 versions, then per 10
- What it learns from sizes
- 10 MCTS:
  - The looks of the overall position of the agent are improving almost every iteration. With a big jump at the start, and a slower progress.

There seems to be an upward trend in the value estimations, although in some cases it goes a bit up and down. Value estimation, maybe better to look at winning, drawing and losing states. Particularly winning states should be as such, although there are many that would not happen in a real game due to anyone having been able to win before. For values it
Figure 3.7: Performance against perfect player using value and prior.

Figure 3.8: Performance against MCTS player using value and prior.

Figure 3.9: Performance against MCTS player using value and prior.
Figure 3.10: Performance against MCTS player using value and prior.

Figure 3.11: Performance against MCTS player using value and prior.

Figure 3.12: Performance against MCTS player using value and prior.
3.4. Experiments

Figure 3.13: Performance against MCTS player using value and prior.

Figure 3.14: Performance against MCTS player using value and prior.
does not matter what kind of state, since some of the block states are also fork, so that they have value of 1 while others have a value of 0.

Here we see clear learning in the defensive move types. And a slight learning in the offensive moves, which have a lot higher scores in general.

Seems not really to learn here, except for blocking a little bit. Would mean that the method learns biased priorities?

Increasing deviation about time between "equally" good moves, also seems to indicate bias. Not sure if we look at / into "quickness" or winning / losing.

ValueScore: \(|val - realVal|^2|\) Says something about whether or not we correctly estimate the value of a state. Should separate states with values of 1, 0 and -1. We want to be able to separate from bad and bad from worse.

OneActionPriorScore: \(1 - (maxQmaxP - notmaxQmaxP)^2 \) or \((notmaxQmaxP - maxQmaP)^2 + 1\) Checks if we are able to select the better moves. Should use something (or add a new function) like: 0.3 0.3 0.3 0.1 = 0.6 0.9 0.0 0.0 0.1 = 0.8 - 0.1 - 0.1 = 0.6

AllActionsPriorScore Would be 0.5 if all are equal If no block actions multiple, why is this lower? Maybe not a very good score? Or it tells that its not learning equally.

BestActionsPriorityScore: Useless for win? Not sure if other moves are useful, seems to indicate increase in bias.

OneActionPriority vs. new oneActionPriorScore should show a bit of the bias. Need to carefully select states for this?
3.4. Experiments

Figure 3.17: Performance against MCTS player using value and prior.

Figure 3.18: Performance against MCTS player using value and prior.

Figure 3.19: Performance against MCTS player using value and prior.
Figure 3.20: Performance against MCTS player using value and prior.

Figure 3.21: Performance against MCTS player using value and prior.

Figure 3.22: Performance against MCTS player using value and prior.
3.4. Experiments

Figure 3.23: Performance against MCTS player using value and prior.

Figure 3.24: Performance against MCTS player using value and prior.

Figure 3.25: Performance against MCTS player using value and prior.
Figure 3.26: Early game move scores.

Use value in first state as proxy? Can be dangerous with specific methods that target this learning.

3.4.3. Experiments with different settings

Exp 1
Considered basic setup: Always select with $\tau = 1$, also learn $\tau = 1$. Value outcome as value to learn. Use Reward in the end state as the value of the end-leaf leaf.

Exp 2
Always select with $\tau = 1$, also learn $\tau = 1$. Value outcome as value to learn. Use Reward in the end state as the value of the end-leaf leaf. Learn priors but do not use during learning. Test if using priors causes "mistakes". Not using priors make the results a lot worse, so they are important to use during learning as well. Has an effect similar to already having visited them more often.

Exp 3
Always select with $\tau = 0$, learn with $\tau = 1$. Value outcome as value to learn. Use Reward in the end state as the value of the end-leaf leaf. Learn priors but do not use during learning. By not using priors in previous experiment probably too much variation in actions that it selects by $\tau = 1$, and does not converge well to good values. Test whether $\tau = 0$ fixes this. Pretty terrible results, priors seem very important as a way to bias the selection towards good moves.
3.5. OmegaHero

Exp 4
Always select with $\tau = 0.5$, learn with $\tau = 1$. Value outcome as value to learn. Use Reward in the end state as the value of the endstate leaf. Use Priors.

Seems to learn the priors alright, similar results to basic setup.

3.5. OmegaHero

Consider algorithm 2, a tabular reinforcement learning algorithm for finite-horizon (deterministic?) MDPs with rewards scaled to lie in the $[0,1]$ interval. TabularUCT algorithm notation based on notation in [3].
For each state $s \in S$ initialize table with a value $V(s) = 0$, the set of legal actions (edges) $(s, a), a \in A(s)$, and for each edge the number of times it has been visited $N(s, a)$ (and $Q(s, a)$?)

#searches is the number of searches (i.e. number of MCTS "cycles") before playing a move. $s_0 \in S$ is the initial state of the MDP. $D$ is the horizon of the MDP.
Algorithm 2 OmegaHero

1: function OmegaHero(MDP, #searches, smdp₀, D)
2:     s = smdp₀
3:     someTable = []
4:     currentD = D
5:     tree = an empty tree with a rootnode for s
6:     while True do
7:         while s ≠ terminalState do
8:             tree = TabularUCT(tree, currentD, #searches, someTable)
9:             s = choose an edge from root of tree with probability \( \frac{N(s,a)}{\sum_b N(s,b)} \), with \( N(s,a) = \) edge.visits + edge.oldvisits
10:     end while
11:     Reinforce(tree, someTable)
12:     currentD = 1
13:     s = smdp₀
14:     currentD = D
15:     tree = empty tree with root for s
16:     end while
17: end function

18: function Reinforce(tree, someTable)
19:     for node in tree do
20:         oldValue = someTable[node.state.id].value
21:         totalValue = 0
22:         oldVisits = 0
23:         newVisits = 0
24:         for edge in node.edges do
25:             oldvisits += edge.oldvisits
26:             totalValue += edge.visits * edge.Q
27:             newVisits += edge.visits
28:         end for
29:         oldValue = oldVisits * oldValue
30:         someTable[node.state.id].value = oldValue + newValue
31:     end for
32: end function
Algorithm 3 TabularUCT part 1

1: function TabularUCT(tree, D, #searches, someTable)
2:    node₀ = root of tree
3:    for i = 0, i < #searches, i+ = 1 do
4:        nodeᵢ = Select(node₀, tree, D)
5:        Rᵢ = ExpandEvaluate(nodeᵢ)
6:        Backup(nodeᵢ, Rᵢ)
7:    end for
8:    return tree
9: end function
10:
11: function Select(node, tree, D)
12:    while s ≠ terminalState do
13:        if node is a leaf then
14:            return node
15:        else
16:            node = childnode corresponding to argmaxₐ(Q(node, a) + 2 \sqrt{\frac{D}{\sqrt{N(s, a)}}})
17:        end if
18:    end while
19:    return node
20: end function
21:
22: function ExpandEvaluate(node, someTable)
23:    if node is terminal state then
24:        return R(node.state)
25:    end if
26:    if node.state.id in someTable then
27:        node.edges = someTable(node.state.id).edges (deep copy)
28:        node.value = someTable(node.state.id).value (deep copy)
29:    else
30:        someTable(node.state.id) = node
31:        node.edges = A(node.state)
32:        node.value = randomSimulation(node) (Note: “randomSimulation” kies vanaf node.state
33:        random acties tot het einde en returned de value/reward, basically een random-rollout)
34:        for edge in node.edges do
35:            edge.oldvisits = 0
36:        end for
37:    end if
38:    if edge in node.edges do
39:        edge.visits = 0
40:        edge.W = 0
41:        edge.Q = 0
42:    end for
43:    return node.value
44: end function
45:
46: function Backup(node, R)
47:    while node is not null do
48:        edge = edge that led to this node
49:        if edge.oldvisits ≠ 0 & edge.visits == 0 then
50:            edge.W = R edge.oldvisits + R
51:        else
52:            edge.W = edge.W + R
53:        end if
54:        edge.visits = edge.visits + 1
55:        edge.Q = edge.W / edge.visits + edge.oldvisits
56:        node = edge.parent
57:    end while
58: end function
3.5. OmegaHero

Figure 3.34: Model 1 draw move position scores.

Figure 3.35: Model 2 versus perfect player.

Figure 3.36: Model 2 value scores.
Figure 3.37: Model 2 win action scores.

Figure 3.38: Model 2 block action scores.

Figure 3.39: Model 2 fork action scores.
Figure 3.40: Model 2 block fork action scores.

Figure 3.41: Model 2 winning move position scores.

Figure 3.42: Model 2 draw move position scores.
Figure 3.43: Model 3 versus perfect player.

Figure 3.44: Model 3 value scores.

Figure 3.45: Model 3 win action scores.
3.5. OmegaHero

Figure 3.46: Model 3 block action scores.

Figure 3.47: Model 3 fork action scores.

Figure 3.48: Model 3 block fork action scores.
Figure 3.49: Model 3 winning move position scores.

Figure 3.50: Model 3 draw move position scores.

Figure 3.51: Model 4 versus perfect player.
3.5. OmegaHero

Figure 3.52: Model 4 value scores.

Figure 3.53: Model 4 win action scores.

Figure 3.54: Model 4 block action scores.
Figure 3.55: Model 4 fork action scores.

Figure 3.56: Model 4 block fork action scores.

Figure 3.57: Model 4 winning move position scores.
Figure 3.58: Model 4 draw move position scores.


57