Using SARSA with Function Approximation to Create Policies for MCTS

by

Dagmar Lukka Loftsdóttir
Hera Hjaltadóttir
Valgarður Ragnheiðar Ívarsson

A thesis submitted for the degree of

Bachelor of Science (B.Sc.) in Computer Science - Research Based
and
Bachelor of Science (B.Sc.) in Computer Science

May 2019

Using SARSA with function approximation to create policies for MCTS

Dagmar Lukka Loftsdóttir[*], Hera Hjaltadóttir[†], Valgarður Ragnheiðar Ívarsson[‡]

Department of Computer Science,

Reykjavík University,

May 16, 2019

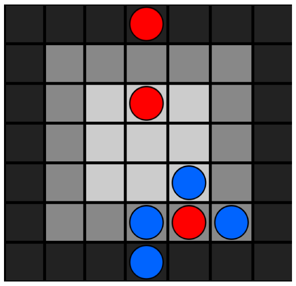

Figure 1: A graphical representation of the game Two Player Tic-Tac-Chess [1].

Abstract

This paper proposes CARL, a pair of agents that apply reinforcement learning and function approximation using regression to learn policies for games where human heuristics cannot be applied. The purpose of these policies is to do search control in Monte Carlo Tree Search (MCTS), a heuristic search algorithm to see if the learned policies can outperform upper confidence bound for trees (UCT).

Keywords: General Gameplaying, Monte Carlo Tree Search, SARSA, Reinforcement Learning

1 Introduction

Research into the field of game playing has a long history and has remained a crucial part of the field of artificial intelligence (AI) since its own naissance in the 50's. Just as games have the capacity to allow humans to develop important skills, game playing has played an important role in the development of AI methods. We use games as an abstraction of more complex systems that are difficult to model. By researching in the context of games, we can learn generalizable knowledge that we can then apply to the more complex problems. Great strides have been made recently in this field, specifically in regards to the games Chess and Go.

The field of general game playing (GGP) studies methods of playing arbitrary games based on descriptions in a formal language, without the use of human knowledge. An AI agent is passed the description of the game and without having had any heuristic based on human knowledge passed to it, it must play the game as well as it can. The goal for those interested in this field is often to create an agent that surpasses as many different agents in

[*]e-mail:margretl16@ru.is
[†]e-mail:herah16@ru.is
[‡]e-mail:valgarduri16@ru.is

as many games as possible [2]. This is not a trivial task. Games can be varied and of the different methods devised by researchers, not one is universally superior across all games.

Google Deepmind's Go-playing agent used a collection of human expert moves, in combination with self play through reinforcement learning, to become the first agent capable of beating a human professional with zero handicaps [3]. However, since general game playing agents need to be able to learn arbitrary games that may not have any human experts, using man-made heuristics is not viable to increase performance. We must therefore look into other methods to write successful general game playing agents.

That being said, simulation based methods such as Monte Carlo Tree Search [4] (MCTS) are popular choices due to repeatedly showing success in a wide range of problems ranging from playing specific games to solving complex real-life problems. Considerable research effort has been put into devising search guiding methods for MCTS to affect what part of the state space we search to maximize the value of the information we obtain in our search [5]. We propose two such methods that use reinforcement learning to learn something general about the game to be used as a heuristic for MCTS. We collectively call the agents created with those implementations Cadia Agent using Reinforcement Learning, or CARL for short.

2 History and Related Work

The history of GGP agents dates back to 1968 where Jacques Pitrat describes the idea behind general gameplay and briefly explains his own simplistic GGP program [6], likely the first of its kind. At the time only domain specific game playing programs existed but after Pitrat's paper the field of GGP would grow more traction as time passed. In 2005 the Stanford Logic Group at Stanford University launched the General Gameplaying Project in order to encourage further research into the subject. To reach this goal, they standardized the Game Description Language [7] (GDL) as we know it today and opened a server which GGP agents can use to play multiple games against different opponents. In the same year the project was initialized, the first GGP competition was held and has been annually since then [7].

In 2006 Rémi Coulom described the difficulties faced with GGP agents at the time with playing the game of Go [4]. Due to its enormous state space, traditional game search trees were too slow to allow an agent to become a competent Go player. To combat this Coulom proposed combining tree search and Monte Carlo evolution and wrote the first MCTS agent which showed potential by convincingly beating a Monte Carlo Go-playing agent which was state-of-the-art at that time. MCTS soon became a standard in GGP agents and theories sprouted on how the algorithm could further be improved. Only one year after Coulom published his paper on MCTS, the GGP agent Cadiaplayer would go on to beat the AAAI GGP 2007 competition, as well as winning the 2008 and 2012 competitions [8], each time employing a different improved version of the MCTS algorithm. Like our agents, each version of Cadiaplayer used MCTS in combination with an upper confidence bound (UCB) to provide balance in selecting between the best move and exploring new ones [9]. Cadiaplayer has used several heuristics to increase its performance. One such heuristic, named RAVE, adds bias to MCTS by using separate domain knowledge gained through play [10].

In 2015, Google Deepmind's agent AlphaGo made history by becoming the first computer program to beat a human professional in the game of Go with zero handicaps. AlphaGo starts by training a neural network using a database of expert human moves to initialize a reinforcement learning policy that continues learning by performing self play. This neural network is then used to guide MCTS [11]. This allowed AlphaGo to reach unprecedented success in the game of Go, with a 99.8% win rate against other Go programs [3].

3 MCTS

MCTS is a heuristic search algorithm which is popular in general gameplaying. It is based on performing simulations of the game to gain information. This information is used to decide which action you have available at a given state of the game is best. The most valuable action for a game play-

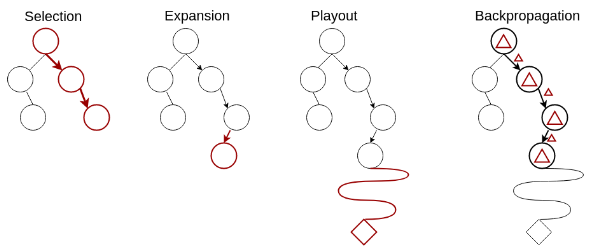

Figure 2: The four steps of MCTS.

ing agent at any game state is the action likeliest to lead the agent to win the game.

MCTS consists of four steps, as seen in Figure 2. The steps are selection, expansion, playout and backpropagation. These four steps are performed repeatedly until some stopping criteria is reached, often the time allotted to perform the search. In the first step, selection, we traverse a tree that is kept in memory. The nodes of this tree represent states of the game, and the edges actions which can be taken from that state. We use a selection policy to traverse down this tree. Frequently, the UCB formula is used for this phase [12]. MCTS that uses UCB in its selection phase is often called upper confidence for trees (UCT). In the next step, expansion, we add one more node to the tree in memory. This node is added according to an expansion policy, often a leftmost policy. The next step of MCTS is the playout, in which the game is played out to a terminal state from the newly expanded node. A terminal state is a state of the game where a player has either won, lost or reached a draw. The playout stage uses its own policy, a playout policy, to decide how the rest of the game is played out. The fourth and final step, backpropagation, takes the value of the terminal state and propagates it back up the path of selected states and actions in the tree.

Our proposition is to use the policy function learned by SARSA as the playout policy of MCTS, as well as to use the value function learned by SARSA in conjunction with UCB to create a selection policy.

4 SARSA

In reinforcement learning, we are concerned with learning some system to choose actions in different states. We call this system a policy. The goal of reinforcement learning is to create a policy that eventually converges to the optimal policy. That is, a policy that in every state, takes the best possible action. State Action Reward State Action (SARSA) is a reinforcement learning algorithm that learns the values of taking certain actions in states, a value function, through simulations of the game [13, Chapter 6.4]. This value function is then used to generate a policy. The policy selects the best perceived action in a state by comparing the value of each move from the state and selecting the action with the highest value. SARSA trains its value function by repeatedly simulating the game, updating the value function at every step of simulation. The value function is updated with two pieces of information, a reward given at favorable states and the value of a successor state-action pair. To get the value of the successor state-action pair from a state, the policy is used to choose an action in the successor state, then the value function is used on the pair consisting of the successor state and the chosen action. This is called bootstrapping. The formula for the update can be seen in Formula (1). s, a and r are the current state, action taken from it and the reward respectively. s' is the state from s with action a, and a' is the action taken from s'. α is the algorithm's learning rate, it specifies how quickly it learns a policy. If set too low, it might converge too slowly and if set too high, it might converge to the wrong value or not at all. γ is the discount factor and is used to discount the value of successor states. Discounting is used to balance between favoring immediate and distant rewards [13, Chapter 3.3]. $Q(s, a)$ is the value function.

$$Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma Q(s',a') - Q(s,a)) \tag{1}$$

As mentioned, SARSA is usually trained as it plays. This can be useful when rewards are given mid-play. In GGP however, rewards are only given at terminal states. An agent receives 0 for losing, 50 for a draw and 100 for winning. Because of this, we instead chose to collect each game into lists of state-action pairs and update SARSA after each game, from the terminal state to starting state. SARSA will converge faster under this method, as each state can immediately be updated with the given reward.

SARSA is often implemented using a tabular method, storing the value of each state-action pair. This can prove problematic when one is limited by either memory or time, as the state-space of most interesting games is prohibitively large for such an approach to be successful. Therefore, we have chosen to use function approximation instead. When using function approximation, an incremental supervised learning model is used for the value function. The supervised learning model accepts an array with an encoding of the game state and an action and returns a prediction of the value of the state-action pair.

In reinforcement learning, there is a phenomenon known as the Deadly Triad that specifies that convergence can not be guaranteed if all of three the following are chosen:

∗ Function approximation.
∗ Bootstrapping, updating values of states using values of other states.
∗ Off-policy training, updating the values of states using the perceived best action instead of the policy.

We were left with 2 options, as function approximation could not easily be given up. We opted for using bootstrapping as it increases computational efficiency and speeds up convergence. Off policy learning also speeds up convergence, but to a lesser extent [13, Chapter 11.3]. The algorithm that best fit our criteria was SARSA (as opposed to other methods such as Q-learning) and so we chose that.

5 Search control using SARSA

Our hypothesis is that by using reinforcement learning we can learn something general about the game at hand to use as a heuristic for an MCTS agent. Our implementation is based in part on the AlphaGo paper [3], but substituting human expert moves with simulations of the game.

5.1 Reinforcement Learning Before the Game Begins

Before a game starts, participating agents are sent the game description in GDL. The agents must parse this information and do any relevant preparation they need before time runs out and the game begins. Along with the description, agents also receive the time at which they must be ready to begin the game. Our agent, upon receiving the start message from the game master, begins running SARSA to learn a value function for each role in the game. It runs SARSA for the duration of the period before the game starts.

5.2 Search Guiding in MCTS

Once the game begins the agents must, for every step of the game, submit their selected move. If the game master receives no move from the agent before time runs out, it will choose a random action on behalf of the agent [2]. GDL models all games as simultaneous move games, however turn-taking games like the ones we focused on can be produced by forcing a 'noop' move from a player when it is not their turn. In other words, although the players make moves simultaneously, only one player makes an action during each turn while the other performs a non-action, essentially waiting their turn.

When an agent receives a move request from the game master, the agent has some time to decide on a move. This time is typically called the play clock. For our agents, we use this time to perform MCTS. We have implemented policies to be used in two of the four steps of MCTS, which can be seen in Figure 2. A diagram showing the structure of our agent can be seen in Figure 3.

5.2.1 Selection

As previously described, UCT is a popular version of MCTS. It is based on calculating the UCB for the value of a state-action pair, as described in Formula (2).

$$UCB = Q(s,a) + C\sqrt{\frac{\ln n(s)}{n(s,a)}} \qquad (2)$$

$Q(s,a)$ is a value function that returns some learned value for a state-action pair s and a. $n(s)$ denotes the number of visits to a state whereas $n(s,a)$ stands for the number of times an action in a state has been chosen in selection. C here is a constant which controls the balance of exploration and exploitation in MCTS. A higher value means more exploration while lower values favor

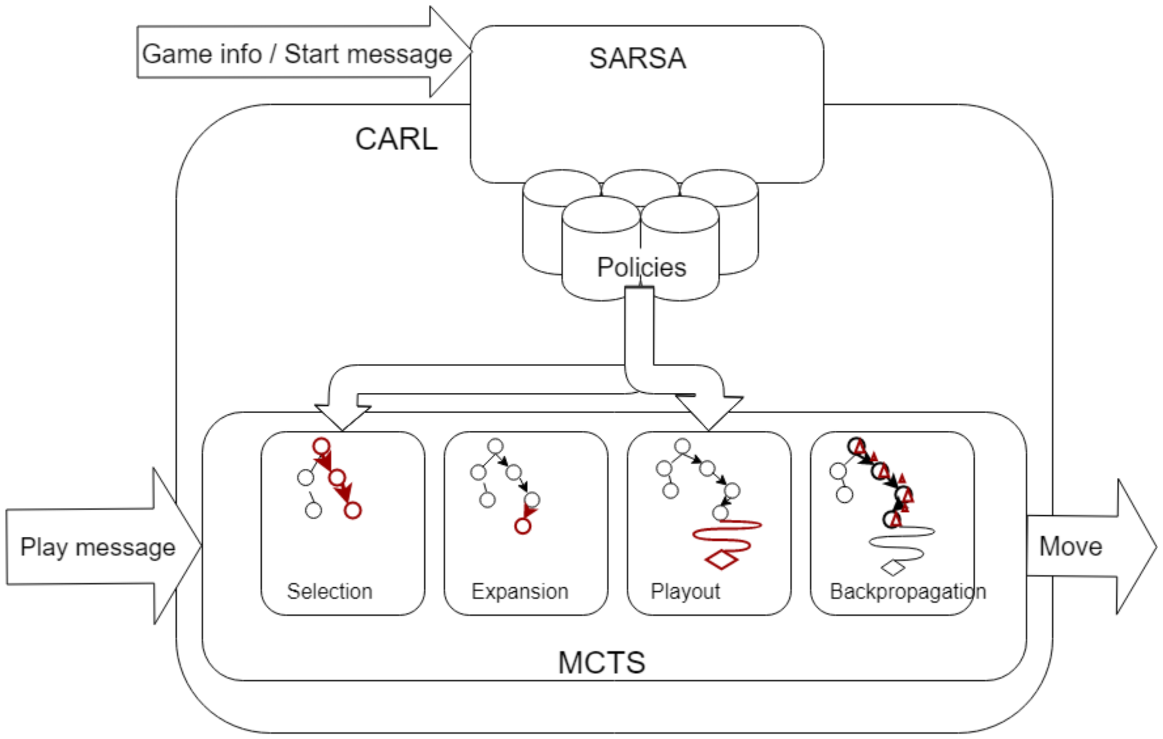

Figure 3: A diagram showing the structure of our agent as well as the flow of information between its parts.

exploitation. Since GGP has a standard rewards of 0 for loss, 100 for wins and 50 for draws in the case of two-player games, we expect $Q(s, a)$ to be in this range as well. Empirically $\sqrt{2}$ does well in cases when the reward is in the range of 0 to 1 [5]. Therefore, we chose $C = 140 \approx 100 * \sqrt{2}$. In similar situations where search control is used to guide MCTS, we may want to lean more towards exploitation [10]. However, we found ourselves leaning towards the more empirical value for our experiments.

We propose a different selection policy which uses the value function learned before the start of the game in conjunction with UCB. A similar method of search guiding for selection was used by Yngvi Björnsson and Hilmar Finnsson in their RAVE Cadiaplayer [10]. We use the same weighting scheme for phasing out our SARSA value function as MCTS explores more of the state space and its values become more representative of the state of the game. For this, we use weighting according to Formula (3), Q being the value that is passed into the UCB as Q in Formula (2). Q_{SARSA} refers to the Q-value function SARSA has learned for a given state-action pair, and Q_{MCTS} refers to the value MCTS has learned for the same pair. $\beta(s)$ is described in Formula (4). k is the equivalence parameter which controls how many visits to a given node are required before both the Q values for SARSA and MCTS are weighted equally. $n(s)$ is again the number of times that state has been visited by MCTS . Due to the number of expansions we do per move of the game, we selected a relatively low k that is still likely to give us some meaningful results, $k = 100$.

$$Q(s,a) = \beta(s) * Q_{SARSA}(s,a) + (1 - \beta(s)) * Q_{MCTS}(s,a) \tag{3}$$

$$\beta(s) = \sqrt{\frac{k}{3n(s) + k}} \tag{4}$$

5.2.2 Playout

Likewise, the value functions we learned at the start of the game can be used to guide MCTS in its playout step. In our implementation, we turn the values of all actions into a probability distribution and select an action given this distribution.

6 Empirical Analysis

To evaluate our method we first needed to tune certain hyper-parameters and perform evaluations to determine certain implementation details.

We ran a series of experiments, some on local machines with Intel Ivybridge CPUs and 4 GiB of RAM. The majority of our experiments we ran on the Google cloud computing service. On the Google cloud service we ran one game master with 18 vCPUs, 16GiB of RAM. The agents were all running on their own instances with 1 vCPU and 3GiB of RAM. The CPU architecture for all instances was Intel Skylake. All machines were running Ubuntu 16.04.

6.1 Regressors

A key aspect of our method is function approximation using a regressor, which can be done using many different machine learning models. We made the choice to implement our agent in Python, and therefore the easiest approach to regression for us was to use the library SciKit Learn [14].

A requirement for our regressors is that it can be used for online learning, as SARSA uses bootstrapping on simulations and therefore we must be able to iteratively learn new pieces of information to update its policy. The algorithms that fit our constraints were stochastic gradient descent (SGD), multi-layer perceptrons (MLP) and a passive-aggressive regressor (paggro). All these regressors are implemented by SciKit Learn and we use the default parameters, except in the case of multi-layer perceptrons, for which we chose three hidden layers with 40 nodes in the first layer, 20 in the second and 40 in the third. This structure is presumably not ideal and an aspect of our implementation that should be given more attention. However, we did not have the time or resources for further experiments.

Before we could start comparing regressors we needed to establish a baseline for testing them against. We decided on UCT with some limited number of expansions per move. To establish which number of expansions is acceptable, so that no regressor at any time range was either winning or losing consistently, we had to try a number of possible values before deciding on one. We tested all three regressors in a SARSA agent against these varying baselines, and found that 1000 expansions was sufficient for us for the game of Connect-Four. Due to the amount of time required for these experiments we only tested the regressors on this one game.

To compare regressors we are primarily interested in how good of a policy they manage to learn. To test this, we evaluate their quality against a baseline. The baseline agent runs UCT with a random playout policy. The baseline agent is capped at 1000 expansions per move. We also examined other factors such as error over time (EOT), which is described by Figure 6. α is the learning rate of the instance error, by how much we should shift it for each sampled instance, and E_i is the error for a given instance of the function. This error is described in Figure 5, where $Q(s,a)$ is the value function learned by SARSA, s and a is a state-action pair and r is the immediate reward for the action a in state s.

$$E_i = |Q(s,a) - (r + Q(s',a'))| \tag{5}$$

$$EOT = EOT + \alpha(E_i - EOT) \tag{6}$$

We also looked at a metric we have chosen to call the expected exploration factor (EEF).

$$EEF = \frac{\log_b s}{d} \tag{7}$$

EEF is described by Formula (7), where b is the average branching factor, d is the average depth for terminal states, s is the number of state expansions managed by SARSA during the training period. The size of the fully expanded search tree is estimated to be b^d, so EEF is an approximation of how much of the state space has been explored by SARSA. However, the number of state expansions done is not equal to the number of nodes in the tree SARSA has looked at as it can and does look at the same node several times. Therefore it cannot be treated as an accurate measurement of how much of the search tree has been explored, but it does give us some idea.

To measure these factors we gave the SARSA agent varying start times, 300, 600 and 1200 sec-

Game	**Branching Factor**	**Average Depth**
Breakthrough	15.5	50
Checkers	6.4	60
Connect-Four	4	36

Table 1: The average branching factor and average depth of each game used in the experiments [15]–[18].

onds. We gave the agents an abundant play clock as neither agent needs to do time-bounded search for moves. The agents were only tested by playing Connect-Four, which has an average branching factor of 4, and an average of 36 moves per game as seen in Table 1. For each combination of start times and regressors, we learned 19 policies using the same parameters and tested each policy on 50 games to obtain our results. These experiments took roughly 5.5 hours to run to completion.

6.2 Policies

The policies previously described are the focus of our research so we must do empirical analysis of their quality. To evaluate this, we look at the win rates of our suggested agents against an agent running UCT with a random playout policy.

For these games, all agents have 600 seconds of start time, and they have the same number of maximum expansions. The agents are allowed a limited number of expansions to select a move. The expansion-limited agents are tested given 1000 expansions each. The play clock for these agents is abundant so that they can always manage the number of expansions they are given.

We ran tests on the games in Table 1. For each game we learned 46 new sets of policies for our agents. For half of those policies our agent assumed the role of player 1, and the other half player 2. Each policy was tested by playing 25 games against the UCT agent. The results of these tests are described in Section 7.

7 Results

The following are the results of the experiments performed as described above.

7.1 Regressors

The win rates of the different regressors can be seen in Figure 4. Despite the number of experiments we ran, the standard error is still great enough that we cannot draw any conclusions based on this alone. However, looking at the evolution of the error over time in Figure 6 shows us that multilayer perceptron (MLP) regression has the lowest average error over time. This suggests that MLP manages to learn more valuable information about the game. To the contrary, it has the lowest expected exploration factor as seen in Figure 5 for the given time. The depth and branching factor should be the same as all regressors were tested in the same game. This therefore only reflects how many fewer state expansions MLP managed, which is unsurprising due to how many parameters it has to learn and the complexity of the algorithm. It is for this reason, that MLP has the lowest error over time despite fewer state expansions that we choose to use that regressor for our policies.

One observation one can make given the data we gathered is that none of the regressors have been trained to convergence, although it is hard to say whether this would happen in a realistic amount of time for this context or if it would happen at all. We also see that after ten minutes (600 seconds) of training, MLP seemingly begins to perform worse, although marginally (and well within a single standard error). We therefore chose to give 600 seconds of start clock in the remainder of our experiments.

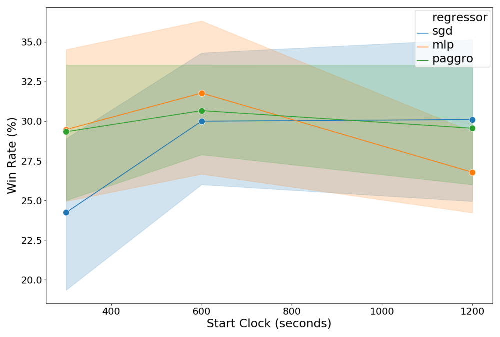

Figure 4: SARSA regressor function approximator average win rates as a function of the amount of time given to train the regressor through simulations of the game.

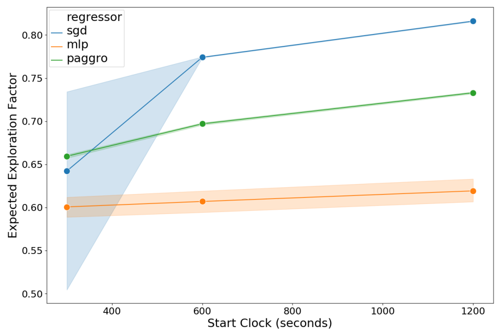

Figure 5: The expected exploration factor of different regressors as a function of the number of SARSA expansions during training. The regressors were given 1200 seconds to train. These values are averaged over each policy learned for the regressor at hand.

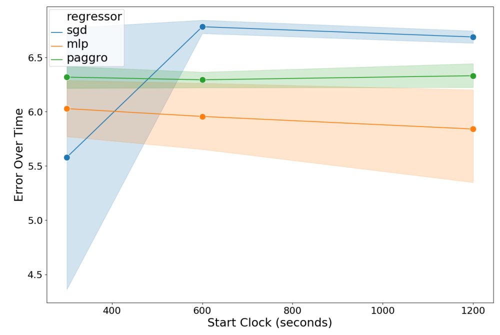

Figure 6: The error over time of different regressors as a function of the number of SARSA expansions during training. The regressors were given 1200 seconds to train. These values are averaged over each policy learned for the regressor at hand

7.2 Policies

The results of our policy tests can be seen in Tables 2, 3 and 4. P1 refers to player 1 and P2 to player 2. The confidence bound given is for one standard deviation, or a 68% certainty interval. For the game connect-four, we found that 50% win rate is within the confidence bounds given for our experiments. The numbers however, are trending towards our agents losing against UCT. Both agents appear to hold their own as player 1, but lose a majority of games as player 2.

For the game checkers, we see inconclusive results as well. CARL with guided playouts won just over half of the games played with a 54.045% winrate, though the confidence bound is ± 15.01. CARL with guided selection loses more often, with a $42.85\% \pm 19.06$ winrate. For breakthrough, the agent with guided playout has an average win rate of $78.29\% \pm 11.58$ against the UCT agent.

CARL Agent	Game: Connect-Four		
	As Player 1	**As Player 2**	**Total**
Playout	$46.17\% \pm 12.16$	$33.67\% \pm 13.11$	$39.92\% \pm 12.63$
Selection	$51.16\% \pm 11.26$	$34.5\% \pm 11.24$	$42.83\% \pm 11.25$

Table 2: The win rates of our agents against the UCT agent in the game connect-four. The confidence bound given is for one standard deviation, or a 68% certainty interval.

CARL Agent	Game: Breakthrough		
	As Player 1	**As Player 2**	**Total**
Playout	$83.65\% \pm 10.53$	$72.93\% \pm 14.85$	$78.29\% \pm 11.58$
Selection	$45.52\% \pm 18.19$	$27.52\% \pm 14.89$	$36.52\% \pm 16.54$

Table 3: The win rates of our agents against the UCT agent in the game breakthrough.

7.3 Time Measurements

We expected the addition of SARSA to slow MCTS down, especially with playouts. To get an idea of how fast our algorithms are, we decided to run a profiler on our agents. All games were played in the game BreakthroughSmall with 10 seconds between moves and using stochastic gradient descent for function approximation.

The time spent in each stage of MCTS can be seen in Tables 5, 6 and 7. Time spent in subfunctions of each MCTS stage was profiled but is not represented in the tables. As seen in Figure 6, MCTS with SARSA in selection spent around 65% of its time in the selection phase. 65% of the time spent was in using the value function learned from SARSA, meaning that 42% of total time is spent in SARSA related computation. By comparison, UCT was over twice as fast, spending 25% of its time in selection and around 55% in playout.

CARL Agent	Game: Checkers		
	As Player 1	**As Player 2**	**Total**
Playout	$48.30\% \pm 13.78$	$59.79\% \pm 16.23$	$54.045\% \pm 15.01$
Selection	$39.27\% \pm 15.53$	$46.43\% \pm 22.59$	$42.85\% \pm 19.06$

Table 4: The win rates of our agents against the UCT agent in the game checkers.

MCTS with SARSA policy guided playouts was slowest by far, with UCT being nearly 50 times faster, or spending 98.9% of its time in playouts. Around 87% of the time it uses for playouts is spent in the SARSA value function.

For both CARL agents, a significant part of the computation time is spent in the value function learned by SARSA. 53% of the time therein was in encoding states and actions into SciKit Learn friendly form and 47% in having stochastic gradient descent predict values.

Function	**callee time**
mcts_selection	23.921%
mcts_expansion	14.295%
mcts_playout	55.367%
mcts_backpropagation	5.821%

Table 5: Time spent in each stage of UCT.

Function	**callee time**
mcts_selection	64.875%
mcts_expansion	6.51%
mcts_playout	25.797%
mcts_backpropagation	2.527%

Table 6: Time spent in each stage of MCTS with SARSA and RAVE.

Function	**callee time**
mcts_selection	0.512%
mcts_expansion	0.404%
mcts_playout	98.914%
mcts_backpropagation	0.15%

Table 7: Time spent in each stage of MCTS with SARSA in playouts.

8 Conclusions

The results of our experiments imply that further research is required to draw a valuable conclusion. We see that our agents perform better than UCT in some games, and worse in others under our constraints. Our agents also seem to lose more in the role of second player, which may be due to the common dynamic in games of one role having an advantage over the other. The playout guiding seems more valuable overall, but as referenced in Section 7.3, it is significantly slower. This may rule it out as a viable option in cases such as where we are limited by time. The selection guided agent requires significant further testing to determine if it is viable. However, we can see that with the parameters we assigned, it does not work to our favor.

In particular, the equivalence parameter (k) we selected ($k = 100$) was likely too high, leading to too much exploitation and not enough exploration. If our policy was stronger, perhaps this would be acceptable, but as seen in Section 7.1 the model has probably not learned enough about the game for this. CARL with guided playouts does not seem to perform better than UCT in Connect-Four and Checkers. Both of these games have relatively low branching factors, as seen in Table 1. In games with a lower branching factor, MCTS has fewer actions to consider and therefore can explore the actions it has available more times, gathering more information. As such, MCTS does quite well in games with low branching factors, which might explain why we see the pattern of our agents not doing better than UCT in these situations

9 Future Work

It is evident from the above that there is much research to be done on the subject for definitive conclusions to be drawn. In particular there are certain shortcomings to our research that should be looked into.

The most notable shortcoming of our approach is that we did not manage to test different variations of hyperparameters sufficiently. Notably, we believe we should test different structures of MLPs as well as different values for max expansions for the playout phase and the equivalence parameter (k). One might also consider making the equivalence parameter adjust to some metric of how good the policy learned by SARSA is. There is also recent research that augments UCB using learned heuristics [19] which looks promising to apply to our context.

Additionally, there are areas beyond the scope of our project that we believe hold much promise. In particular, it would be interesting to look into the prospect of feature extraction through clustering methods or Principal Component Analysis (PCA). The MLP we use has a prohibitively large amount of parameters due to being a fully connected network. However, the weights between the input layer and the first hidden are more numerous than the rest of the weights in the network. Therefore, shrinking the input layer by reducing the dimensionality of the state would significantly lower the number of parameters in the network, and thereby lower the number of examples it needs to learn from. It might also be worth looking into adding convolutional layers, such as is done in the alphago paper [3].

A promising alternative implementation of SARSA function approximation is with a multi-label supervised learning model. Instead of having the value function take a state and an action and return a single value, make it take only a state and return the value of each action. This could speed up the policy for neural networks, as SARSA would only need to call its value function once, not for each action.

The agent was written in Python 2.7, which as of the time of writing this report is considered outdated. This caused some restrictions regarding which machine learning tools we had available. For example, a library of incremental supervised learning models called Scikit Multiflow was only available for Python 3.5 and over. Porting the code base to a newer version of Python might yield interesting results.

Acknowledgements

We would like to acknowledge that without the support and assistance of our instructor, Dr. Stephan Schiffel, this project would never have been started in the first place. He has been an incredible resource to us. Our code is built on Richard Emslie's GGPlib, who was also very accomodating to our questions. Dr. Gylfi Þór Guðmundsson also deserves great thanks for reading over our paper. We would also like to thank the Reykjavík University computer science department for providing us with a work space for our project, as well as turning a blind eye to the trolley of stacked computers we set up without asking. A special thanks to Skema as well, for providing the computers to stack.

References

[1] *Stanford game database*. [Online]. Available: `http://games.ggp.org/` (visited on 05/15/2019).

[2] M. Genesereth, *Overview of general game playing*. [Online]. Available: `http://ggp.stanford.edu/notes/overview.html` (visited on 05/14/2019).

[3] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Alphago - mastering the game of go with deep neural networks and tree search", *Nature*, vol. 529, pp. 484–503, 2016. [Online]. Available: `http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html` (visited on 05/14/2019).

[4] R. Coulom, "Efficient selectivity and backup operators in monte-carlo tree search", in *5th International Conference on Computer and Games*, P. Ciancarini and H. J. van den Herik, Eds., Turin, Italy, May 2006. [Online]. Available: `https://hal.inria.fr/inria-00116992` (visited on 05/14/2019).

[5] C. Browne, E. Powley, D. Whitehouse, S. Lucas, P. I. Cowling, S. Tavener, D. Perez, S. Samothrakis, S. Colton, and et al., "A survey of monte carlo tree search methods", *IEEE Transactions on Computational Intelligence and AI in Games*, 2012.

[6] J. Pitrat, "Realization of a general game-playing program.", *IFIP Congress*, 1968.

[7] M. Genesereth, N. Love, and B. Pell, *General game playing: Overview of the aaai competition*, Jun. 2005. [Online]. Available: `https://www.aaai.org/ojs/index.php/aimagazine/article/view/1813` (visited on 05/14/2019).

[8] H. Finnsson, G. Þ. Guðmundsson, S. F. Guðmundsson, S. Schiffel, and Y. Björnsson, *Cadiaplayer wiki*, Dec. 2012. [Online]. Available: `http://cadia.ru.is/wiki/public:cadiaplayer:main` (visited on 05/14/2019).

[9] Y. Björnsson and H. Finnsson, "Cadiaplayer: A simulation-based general game player", *Computational Intelligence and AI in Games, IEEE Transactions on*, vol. 1, pp. 4–15, Apr. 2009.

[10] H. Finnsson and Y. Björnsson, "CadiaPlayer: Search control techniques", *KI Journal*, vol. 25, no. 1, pp. 9–16, 2011.

[11] A. Karpathy, *Alphago, in context*, May 2017. [Online]. Available: https://medium.com/@karpathy/alphago-in-context-c47718cb95a5 (visited on 05/14/2019).

[12] J. Méhat and T. Cazenave, “Monte-carlo tree search for general game playing”, May 2008.

[13] R. S. Sutton and A. G. Barto, “Reinforcement learning: An introduction”, in, Second. The MIT Press, 2018. [Online]. Available: http://incompleteideas.net/book/the-book-2nd.html (visited on 05/14/2019).

[14] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, “Scikit-learn: Machine learning in Python”, *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011. [Online]. Available: http://www.jmlr.org/papers/volume12/pedregosa11a/pedregosa11a.pdf (visited on 05/14/2019).

[15] *Ggp games list*. [Online]. Available: http://www.ggp.org/view/all/games/ (visited on 05/14/2019).

[16] H. Baier and M. H. M. Winands, “Mcts-minimax hybrids with state evaluations (extended abstract)”, in *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, International Joint Conferences on Artificial Intelligence Organization, Jul. 2018, pp. 5548–5552. [Online]. Available: https://doi.org/10.24963/ijcai.2018/782 (visited on 05/14/2019).

[17] J. C. C. Guerra, “Classical checkers”, Technical University of Lisbon, Portugal, Nov. 2011. [Online]. Available: https://fenix.tecnico.ulisboa.pt/downloadFile/395143159112/dissertacao.pdf (visited on 05/14/2019).

[18] L. V. Allis, *Searching for solutions in games and artificial intelligence*, 1994. [Online]. Available: https://project.dke.maastrichtuniversity.nl/games/files/phd/SearchingForSolutions.pdf (visited on 05/14/2019).

[19] D. J. N. J. Soemers, É. Piette, and C. Browne, “Biasing MCTS with features for general games”, *CoRR*, vol. abs/1903.08942, 2019. arXiv: 1903.08942. [Online]. Available: http://arxiv.org/abs/1903.08942.