Generalized Reinforcement Learning methods to play Connect 4 on Limited Hardware

Ian Thomas

April 29, 2021

Abstract

This paper will explore two methods of learning an adversarial, zero-sum game through self play. The first is a Monte Carlo Tree Search (MCTS), which randomly draws samples from a uniformly random policy to generate a tree to solve the game, as well as a Monte Carlo method where the prior policy is given by the output of a neural network, and the rollout value is also estimated by that network (an Actor-Critic network). We first establish the baseline of MCTS against a completely random policy, and we can use this MCTS as a baseline to test the Actor-Critic augmentation.

I. Introduction

THE idea of a computer learning to play a game such as checkers or chess has been around as least as long as A.L. Samuel's paper in 1959 [1]. The basic idea involves generating a sequence of games through self-play, and using the results of the games to update the move generation algorithm. This cycle repeats until the program is able to evaluate positions extremely well and perform at super-human levels.

For the majority of more complicated games, such as chess, the dominant computer strategy for a long time was to create a fine-tuned heuristic evaluation function that could evaluate each position of a board, and then use an alpha-beta search in order to minimize the harm that your opponent can do to you. Notable examples include IBM's DeepBlue, which was the first computer chess program to beat Garry Kasparov, the highest rated chess player at the time. The best modern version implementing this method is Stockfish, which has long surpassed any human player with an estimated ELO of 3520 [1]. Even still, novel methods such as DeepMind's AlphaZero algorithm demonstrate that machines can learn to fine-tune model parameters to win even against the best heuristic methods [2], given the right hardware.

That last stipulation is likely why there haven't been many advancements in these types of games since AlphaZero, since not very many people have access to the kind of computing resources needed to fully train this sort of algorithm. A game such as Connect 4 poses an interesting problem for the algorithm to learn, given that anyone with a high end desktop computer has the resources to train an AlphaZero-like algorithm in a matter of hours.

II. Background and Related Work

We can first explore the history of the advancements that led to these modern algorithms:

A. Some Studies in Machine Learning Using the Game of Checkers

A.L. Samuel's 1959 paper, *Some Studies in Machine Learning Using the Game of Checkers*, explores two approaches to this problem: a general neural network that acts as a function approximator for the evaluation function of a board, and a specialized network with fewer parameters whose structure is specially optimized for checkers (the weights of what he calls the evaluation polynomial) [1]. Samuel determines that the first method, a generalized neural net, is impractical given the hardware available at the time and focuses on his findings from the second method, which he says requires reprogramming for each new application.

Samuel's method involved assigning weights to the kings and men in checkers, and scoring the material weight along with measuring the positional advantage in an evaluation function. The current state of the board would then be propagated by the response of the opponent to generate a training tree after each game using two actors: Alpha and Beta. All the training would happen on Alpha, and once Alpha was sufficiently stronger than Beta, Beta was set to the value of Alpha before training continued. Samuel refers to training as learning the 16 coefficients of the evaluation polynomial. This 'network' size is significantly smaller than the millions of parameters used in generalized networks enabled by modern hardware, as Samuel noted in his paper.

[1]The highest human rating ever achieved is 2872 for reference, currently held by Magnus Carlsen [2]

B. Bootstrapping from Game Tree Search

This paper heavily references Samuel's checkers algorithm in its discussion of tree search, noting a few problems with Samuel's algorithm and suggesting some improvements. The first of these flaws was that Samuel's algorithm hard coded material value in the evaluation function, for which positions which were far superior would be weighted lower since they could have had a lower material advantage. Another flaw the paper suggests is that the learning target for Samuel's method is effective only when both the player and the opponent already make good moves, which reduce the efficacy for these types of algorithms when learning from self-play without any prior domain knowledge. The paper defines *search bootstrapping* as adjusting the parameters of a heuristic evaluation function towards the value of a deep search, and it uses hypothetical games generated from the search instead of actual moves by the opponent to generate training labels.

C. Monte Carlo Sampling for Regret Minimization in Extensive Games

Marc Lanctot et al's paper, *Monte Carlo Sampling for Regret Minimization in Extensive Games*, has the same objective as the AlphaBeta algorithm described by Samuel, except the search tree is determined by sampling actions from the action space and propagating the tree based on any simulated uncertainty in the environment and the opponent strategy [5]. The paper also accounts for the strategy for any player as a function of the time history of the information present to that player, and uses it to demonstrate proficiency in the game of Poker. One of the largest takeaways from this paper is that even though sample based algorithms such as Monte Carlo Tree Search require more iterations to converge on an optimal solution, the low cost per iteration makes these methods converge "dramatically faster" than other optimized strategies.

D. Actor-Critic Algorithms

Vijay Konda's and John Tsitsiklis's 1999 paper describing Actor-Critic algorithms sets the framework for the algorithm demonstrated later on in this paper. They describe actor-only methods as working with parameterized policies where the gradients with respect to the parameters are "directly estimated by simulation" and the parameters are updated in a direction of improvement [4]. One of the major problems with actor-only methods is that the gradient estimators have a large variance and that each new gradient is independent of past estimates, so older information isn't accumulated.

On the other hand, critic-only methods (such as value iteration on a Markov Decision Process) aim to learn an approximation to the Bellman equation. These methods typically do a good job of reconstructing the value function approximately, but often fall short when prescribing a policy as a function of the value.

According to Konda and Tsitsiklis, the actor-critic methods hope to utilize the policy estimation of actor-only methods and the value estimation of the critic-only methods to update the policy parameters in the direction of value improvement. This combination reduces the variance in the policy parameter updates as well as improves convergence on the value estimates [4].

E. AlphaZero

The combination of the actor-critic policy/value estimation with Monte Carlo Tree Search is what led to AlphaZero's success in that it was able to outperform the highest rated modern computer programs in chess, Go, and Shogi, entirely from self play in a matter of hours (now these hours were on millions of dollars worth of Google Tensor Processing Units, but it is still remarkable).

AlphaZero first estimates the policy of a state via a neural network, and then uses that policy as a prior distribution when performing the Monte Carlo Tree Search [2]. Each action selected along the way is the result of the Polynomial Upper Confidence Tree algorithm (PUCT), which is a clever and deterministic way of assigning weight to exploration vs. exploitation in the multi-armed bandit problem. When the MCTS reaches a state where it hasn't explored yet, instead of doing a random rollout to the end of the game, it uses the value from the critic approximation, which is then propagated up the tree to the root node. After a certain number of iterations, the target policy of the network is then determined by the number of times MCTS took each action from the root node.

III. Problem Formulation

We define the problem of Connect 4 as follows

- $\mathcal{S}$: The state of the board is represented by a 6×7 array where each element can take on a value in $\{-1, 0, 1\}$, 1 for blue pieces, -1 for red pieces, and 0 for no piece. Terminal states include any state where a connect 4 is made in that turn, and drawn states which fill the board without connect 4.
- $\mathcal{A}$: The action space of the board is $\{1, 2, 3, 4, 5, 6, 7\}$, corresponding to the columns we can drop pieces in. If any column is full, that action is no longer available.
- $\mathcal{R}$: The reward space is simply $\{+1, 0, -1\}$. A player gets a reward of $+1$ when he makes a move that wins the game (connects 4 in a row of his color horizontally, vertically, or diagonally). Since the game is zero-sum, the opposite player gets a score of -1 if the first player wins, and vice versa. A score of 0 is assigned to both players if the game is drawn (all 42 slots are filled with no connect 4)
- $\mathcal{T}$: The transition space is deterministic. The piece played at action i will end in the lowest available space in column i.

This formulation of Connect 4 has a discount factor of $\gamma = 1$, or no discount. Since there are only 42 possible places to put a piece, the game has a maximum length of 42.

IV. Solution Approach

A. Monte Carlo Tree Search

The general MCTS algorithm involves choosing an action based on a prior distribution with some exploration vs. exploitation heuristic, reaching the next state, and repeating the process until a terminal state is reached or an unexplored state is reached. In the case of a terminal state, we can simply propagate the reward back up the search tree. In the case of an unexplored state, we compute the reward by randomly choosing actions until a terminal state is reached and propagate that reward back up the tree. In detail, the algorithm is as follows:

Main loop:

1) Initialize Q and N. $Q(s, a)$ is a dictionary that stores the propagated reward for each state-action pair in the tree. $N(s, a)$ is the number of times each state-action pair is encountered/taken.
2) Run the search algorithm for the number of Monte Carlo Iterations, passing in the root node of the game each time

Search:

1) If node is unvisited:
   a) Initialize $Q(s)$ and $N(s)$ to 0 for all actions
   b) Run random rollout function to get the reward
2) If node is visited:
   a) Run Polynomial Upper Confidence Tree (PUCT) algorithm to determine which action to pick based on Q, N, and a uniform prior distribution for each action:

$$a = \arg\max_a \left[Q(s, a) + c \frac{\sqrt{\sum_{a'} N(s, a')}}{1 + N(s, a)} \right]$$

   b) Take action that maximizes PUCT at this time step, and store the reward as v
   c) Call search on the new state, and add the returned result to the reward v
   d) Update Q and the action taken according to:

$$Q(s, a) = \frac{N(s, a)Q(s, a) + v}{N(s, a) + 1}$$

   e) Increment $N(s, a)$
   f) Return $-v$, to account for different players taking turns in a zero-sum game

To choose the action, we simply pick the action that has the highest $N(s, a)$ for the root node.

B. MCTS + Actor Critic

1) Modified MCTS: Let us first consider the modifications to MCTS to incorporate the Actor-Critic network. We must keep track of an additional dictionary $P(s,a)$ that stores the network-generated policy at each visited state. If the model is training, we add some Dirichlet noise to the policy with parameter $\alpha = 0.3$, but we skip this step in the evaluation phase. If an unvisited node is reached, we simply initialize $P(s,a)$ to the network output and instead of performing a rollout, we just return the negative of the value estimate returned by the network. If a visited node is reached, the only difference is in the PUCT criterion:

$$a = \arg\max_a \left[Q(s,a) + cP(s,a)\frac{\sqrt{\sum_{a'} N(s,a')}}{1+N(s,a)} \right]$$

We now note the addition of the prior policy distribution from the network $P(s,a)$.

2) Actor-Critic Network Architecture: The network will consist of three major components: a common layer to process the incoming state information and prepare it for the actor and critic, a policy head which takes the output of the common layer and returns a policy across every action (actor), and a value head which takes the output of the common layer and returns an estimate of the value (critic).

The common layer is built starting with a convolution layer with a 3×3 filter size, one input channel, and four output channels. The common layer also includes five residual network blocks, each made up of two batch normalized convolutional filters with a 3×3 filter size and four input/output channels. The residual blocks simply add the input of the convolutional filters to the output, and the result is fed through a rectified linear activation.

The policy head batch normalizes the output of the common layer, flattens the output, and includes a dense layer which outputs a vector of length 7, corresponding to the number of available actions. The activation on this last layer is a softmax, converting everything to probabilities.

The value head is similar to the policy layer, except with two dense layers instead of 1. The RELU activation is kept on the first dense layer, but a hyperbolic tangent activation is used to bound the value output between -1 and 1.

3) Training: Each training loop consists of three steps: generating data through self play, training the network, and evaluating the network against a baseline. During each training step, the network plays 100 games against itself with 800 Monte Carlo simulations per move. Like the AlphaZero paper, Dirichlet noise is added to the root node of every MCTS search tree [2]. This ensures an even greater degree of exploration when training. Experience is stored in the form of each state, each target policy generated from the MCTS, as well as the outcome of the game from the perspective of the player making the move (+1 for a win, -1 for a loss, and 0 for a draw).

The training step involves drawing samples from the experience buffer at random and running the training function on batches of 1000 samples. The cost function used for training is as follows [2]:

$$l = (v_\theta - z)^2 - \pi \cdot \log(\pi_\theta)$$

Where v_θ and π_θ are the value and policy estimates parameterized by θ, the network parameters. z and π are the game outcomes and the target policies, respectively.

Once training is completed, the network is evaluated against a baseline pure MCTS agent and the score is compared against the current best network. If the score is higher than the previous network's score against the MCTS agent, the new network becomes the best network from which training data is generated.

V. Results

A. Pure Monte Carlo Tree Search

The first test consists of running MCTS with progressively more iterations against a random baseline. The results were generated by playing 100 games at each iteration — 50 where the MCTS agent plays first and 50 where the MCTS agent plays second.

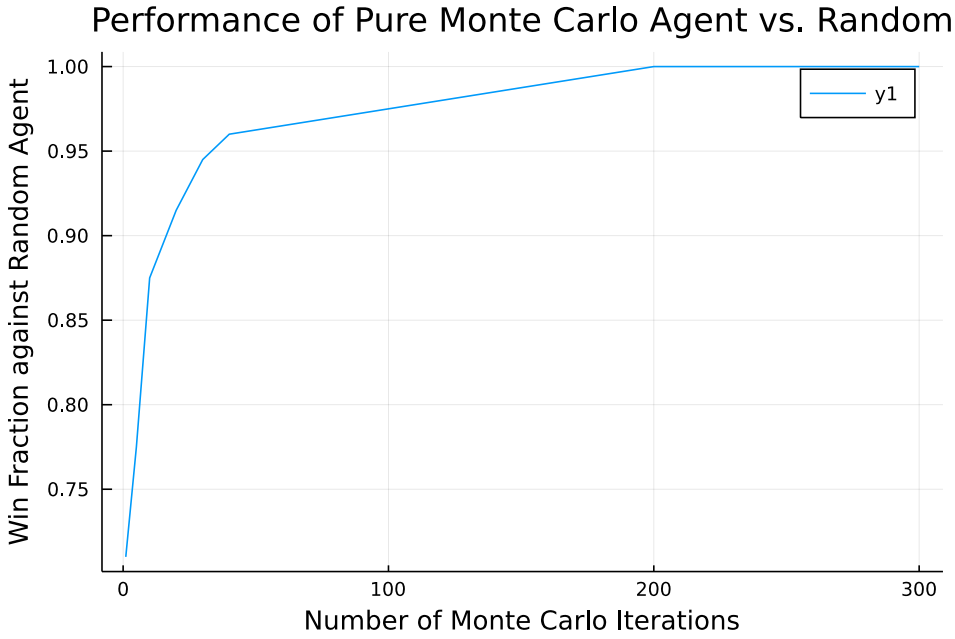

Fig. 1: Varying MCTS iterations against random agent

Even with one MCTS iteration, the win percentage against a random baseline is still around 70%. As more iterations are added, the win percentage climbs very quickly before slowing down until it finally reaches 100% at 200 iterations.

The second test is similar to the first: running MCTS with progressively more iterations against a few MCTS baselines with a fixed number of iterations. The results are for 50 total games — 25 as the first player and 25 as the second.

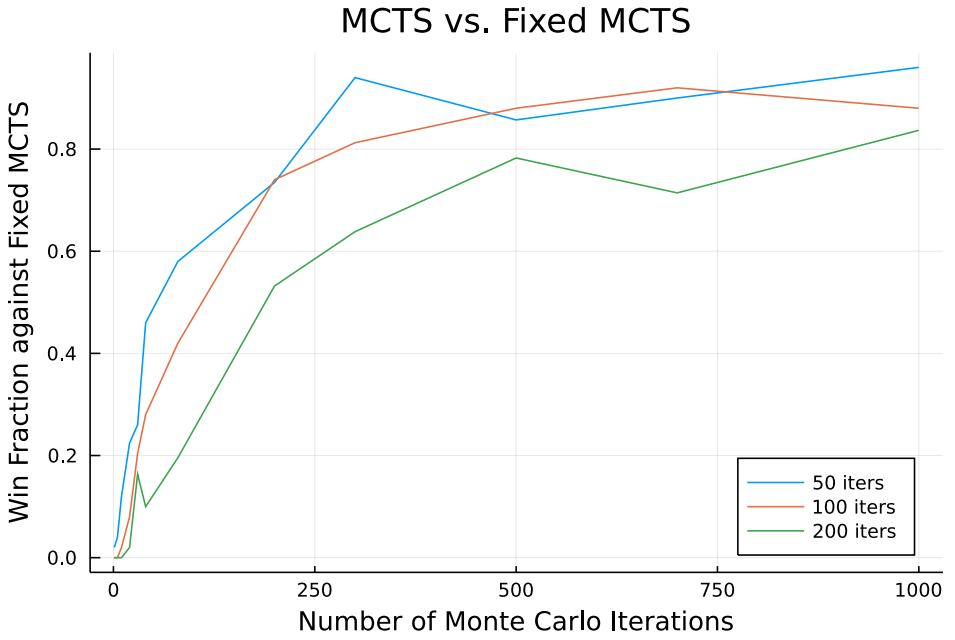

Fig. 2: Varying MCTS iterations against agents with fixed iterations as baselines

The blue line is the win fraction against an MCTS agent using 50 iterations, the orange line with 100 iterations, and finally the green line with 200 iterations. There is some noise present in the model, as the curves for 50 and 100 iterations overlap for a small stretch.

B. Actor-Critic with Monte Carlo Tree Search

Now that a baseline has been established with the MCTS, we can explore how the Actor-Critic network enables better performance than just MCTS alone. The following plots pit the network with 200 deterministic Monte-Carlo iterations against an agent using 200 pure non-deterministic Monte-Carlo iterations as a function of each training step (self play, train, evaluate):

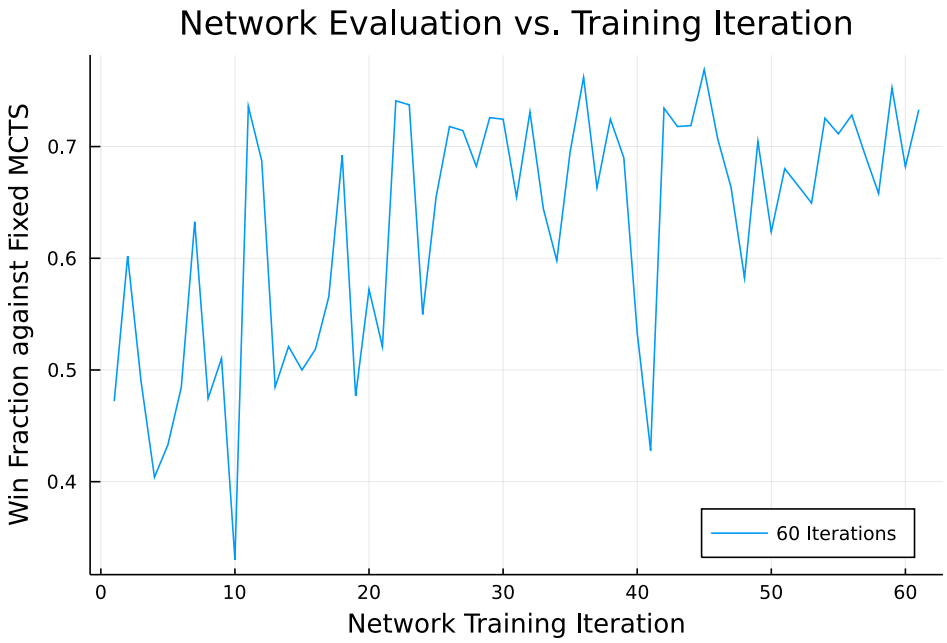

Fig. 3: Evaluation Performance of Network+MCTS implementation against pure MCTS baseline

The model was trained using an ADAM optimizer with a learning rate of $\lambda = 0.2$ for the first 20 iterations, 0.02 for the next 20, and finally 0.002 for the final 20 iterations. The noise in the training function is partially due to the number of evaluated games as well as the noise generated by training. There is a positive upward trend in the performance of the model as it trains more (with the exception of iterations 40 and 41), which is quite remarkable given how small the training set was for each iteration (100 games at an average of around 23 moves per game). Even still, each iteration took about an hour to run on an i5 9600k CPU with 6 cores (for a total of 2 days and 12 hours worth of training).

C. Heuristic Evaluation

For Connect4, it is well established that the optimal action for the first player is to play the middle column ($a = 4$) to start, since with perfect play it is possible to force a win from the very beginning [6]. We should expect to see the network output reflect something more random at first, then slowly converge to picking the optimal strategy first. We can take a look at the network policies as a function of iteration to see if this strategy is reflected:

TABLE I: Raw Network Policies for Initial State

Network Iteration	$\pi(a=1)$	$\pi(a=2)$	$\pi(a=3)$	$\pi(a=4)$	$\pi(a=5)$	$\pi(a=6)$	$\pi(a=7)$	$\arg\max_a \pi(a)$	v_θ
1	0.139	0.156	0.151	0.152	0.166	0.123	0.112	5	-0.039
2	0.102	0.151	0.141	0.142	0.171	0.142	0.15	5	-0.038
3	0.106	0.147	0.164	0.14	0.17	0.131	0.142	5	-0.04
4	0.111	0.132	0.176	0.138	0.173	0.116	0.154	3	-0.033
5	0.11	0.126	0.165	0.133	0.206	0.132	0.128	5	-0.04
6	0.122	0.151	0.15	0.135	0.178	0.129	0.134	5	-0.037
7	0.123	0.15	0.144	0.156	0.164	0.13	0.133	5	-0.041
8	0.125	0.152	0.153	0.146	0.16	0.132	0.131	5	-0.04
9	0.137	0.141	0.156	0.153	0.154	0.134	0.125	3	-0.035
10	0.145	0.152	0.163	0.153	0.14	0.132	0.114	3	-0.036
11	0.151	0.133	0.162	0.166	0.15	0.132	0.106	4	-0.029
12	0.151	0.129	0.162	0.182	0.142	0.132	0.102	4	-0.036
13	0.145	0.134	0.164	0.155	0.146	0.122	0.134	3	-0.034
14	0.136	0.129	0.169	0.158	0.153	0.125	0.131	3	-0.034
15	0.127	0.133	0.166	0.142	0.172	0.123	0.137	5	-0.038
16	0.139	0.127	0.151	0.154	0.181	0.115	0.134	5	-0.036
17	0.138	0.115	0.141	0.152	0.176	0.123	0.155	5	-0.038
18	0.139	0.129	0.137	0.161	0.164	0.125	0.144	5	-0.04
19	0.147	0.135	0.154	0.152	0.163	0.121	0.127	5	-0.035
20	0.138	0.147	0.137	0.159	0.171	0.123	0.124	5	-0.035
21	0.139	0.156	0.151	0.152	0.166	0.123	0.112	5	-0.039
22	0.135	0.14	0.144	0.165	0.168	0.134	0.113	5	0.011
23	0.141	0.148	0.149	0.159	0.165	0.131	0.107	5	0.025
24	0.136	0.143	0.151	0.154	0.157	0.15	0.11	5	0.04
25	0.129	0.142	0.156	0.163	0.159	0.139	0.112	4	0.023
26	0.141	0.143	0.154	0.154	0.152	0.141	0.115	3	0.007
27	0.132	0.138	0.152	0.162	0.153	0.143	0.121	4	0.032
28	0.133	0.136	0.149	0.17	0.147	0.14	0.125	4	0.024
29	0.133	0.136	0.149	0.163	0.146	0.147	0.126	4	-0.002
30	0.126	0.131	0.149	0.164	0.15	0.148	0.132	4	0.02
31	0.13	0.139	0.153	0.166	0.155	0.139	0.118	4	0.016
32	0.126	0.147	0.15	0.162	0.161	0.135	0.119	4	0.006
33	0.124	0.153	0.156	0.17	0.148	0.131	0.116	4	0.01
34	0.126	0.144	0.144	0.177	0.145	0.141	0.123	4	0.028
35	0.131	0.13	0.155	0.174	0.15	0.14	0.119	4	0.014
36	0.131	0.129	0.146	0.176	0.155	0.143	0.12	4	0.04
37	0.14	0.136	0.135	0.178	0.154	0.132	0.125	4	0.011
38	0.136	0.133	0.145	0.179	0.156	0.13	0.12	4	-0.003
39	0.132	0.137	0.16	0.166	0.157	0.136	0.112	4	0.012
40	0.138	0.143	0.167	0.154	0.149	0.134	0.115	3	0.022
41	0.133	0.158	0.154	0.154	0.149	0.135	0.117	2	0.01
42	0.137	0.135	0.146	0.177	0.153	0.131	0.12	4	0.012
43	0.135	0.133	0.148	0.174	0.155	0.133	0.122	4	0.015
44	0.135	0.134	0.147	0.176	0.153	0.134	0.122	4	0.02
45	0.134	0.136	0.145	0.173	0.155	0.134	0.123	4	0.017
46	0.136	0.136	0.143	0.174	0.154	0.133	0.123	4	0.021
47	0.137	0.137	0.145	0.172	0.152	0.135	0.122	4	0.02
48	0.138	0.137	0.145	0.169	0.154	0.136	0.122	4	0.024
49	0.14	0.138	0.147	0.166	0.151	0.134	0.123	4	0.021
50	0.142	0.139	0.149	0.165	0.149	0.134	0.121	4	0.022
51	0.141	0.141	0.151	0.164	0.147	0.136	0.12	4	0.021
52	0.143	0.141	0.15	0.162	0.148	0.135	0.121	4	0.016
53	0.142	0.141	0.148	0.164	0.15	0.133	0.123	4	0.023
54	0.14	0.141	0.149	0.164	0.151	0.133	0.122	4	0.02
55	0.137	0.142	0.15	0.166	0.148	0.134	0.121	4	0.026
56	0.136	0.144	0.153	0.167	0.147	0.134	0.119	4	0.022
57	0.135	0.143	0.152	0.17	0.145	0.136	0.12	4	0.022
58	0.135	0.144	0.153	0.167	0.146	0.137	0.118	4	0.022
59	0.133	0.144	0.154	0.165	0.148	0.139	0.118	4	0.018
60	0.134	0.145	0.152	0.166	0.149	0.136	0.117	4	0.02
61	0.134	0.143	0.154	0.166	0.15	0.134	0.119	4	0.019

We see clearly that the network policy initially chooses other actions (5 and 3) for about 25 iterations before essentially converging on and remaining on action 4 for the rest of the iterations. The first extremely strong positive peak (iteration 11) had a policy which chose action 4 first. The low peak at iterations 40 and 41 directly correspond to the iterations where the policy deviated from action 4. Overall, there is a general upward trend in the network-predicted value of the state, starting off as negative and finishing as positive.

VI. Conclusion

Connect4 poses an interesting starting problem for non-trivial reinforcement learning through self play in that the state space is far too big to be solved directly by methods such as value iteration but yet simple enough to be solved by a 9-rule heuristic [6]. As a result, Connect4 is an ideal testbed to test the sort of reinforcement learning algorithms that can generalize with better hardware, more time, and more complexity.

The version of the Actor-critic tree search algorithm played against a pure Monte Carlo Tree Search using the same amount of steps was able to win around 70% of the time after 60 iterations. While this isn't as stunning as some of the results produced by the original AlphaZero paper on much more complicated games, it still demonstrates that the network enhances the MCTS. For a pure MCTS agent with a random rollout, it took approximately 400 iterations to match the same 70% win performance as the Actor-critic tree search algorithm with 200 iterations, where no rollouts are necessary since the network provides a value estimate. Depending on the cost of a rollout and the cost of propagating a state through the network, the actor-critic algorithm stands to provide a massive speedup and performance game compared to brute-forcing by increasing MCTS iterations. The model complexity stays the same throughout training, so the cost of calling the network remains fixed as the benefit increases as more training iterations are run. Additionally, the model learned to chose the optimal action from the starting state even without running an MCTS, which suggests that the network improves the quality of the moves explored during the tree search.

The biggest drawbacks of the model, however, include the time and hardware required to train a model like this quickly and efficiently. The original AlphaZero network playing the game of Go had 20 residual layers with an input size of $19 \times 19 \times 17$ and was able to train from no knowledge to the highest rated computer Go program in 9 hours [2], whereas the network used for this version of Connect4 had 5 residual layers of input size $6 \times 7 \times 1$ and still took approximately 60 hours for a 20% improvement on the original model.

Future work includes implementing a version of this model on distributed hardware to speed up the training times. Tuning the hyperparameters of the model (learning rate, c for PUCT, network size and architecture) has the potential to make the model converge faster as well. Using regularization techniques such as l_2 regularization or including dropout layers can help reduce the problem of the model potentially overfitting on the states it sees in its training step. Finally, a whole new set of model architectures and algorithms have been developed to solve the problem of improving by self play. A particularly interesting one to explore is Soft Actor-Critic, which estimates the $Q-$values for each action in addition to the policy and value estimates.

References

[1] Samuel, Arthur L. "Some Studies in Machine Learning Using the Game of Checkers." Computer Games I, 1959, pp. 335–365., doi:10.1007/978-1-4613-8716-9_14.

[2] Silver, David, et al. "A General Reinforcement Learning Algorithm That Masters Chess, Shogi, and Go through Self-Play." Science, vol. 362, no. 6419, 2018, pp. 1140–1144., doi:10.1126/science.aar6404.

[3] Veness, Joel, et al. "Bootstrapping from Game Tree Search." Advances in Neural Information Processing Systems, 2009, pp. 1937–1945.

[4] Konda, Vijay R., and John N. Tsitsiklis. "Actor-Critic Algorithms." SIAM Journal on Control and Optimization, vol. 42, no. 4, 2003, pp. 1143–1166., doi:10.1137/s0363012901385691.

[5] Lanctot, Marc, et al. NIPS, 2009, Monte Carlo Sampling for Regret Minimization in Extensive Form Games, www.cs.ualberta.ca/research/techreports/2009/TR09-15.php.

[6] Allis, Victor. "A Knowledge-Based Approach for Connect 4." Informatik.uni-Trier.de, University of Amsterdam, 1988, www.informatik.uni-trier.de/ fernau/DSL0607/Masterthesis-Viergewinnt.pdf.