AI AGENTS FOR ULTIMATE TIC-TAC-TOE

PHIL CHEN, JESSE DOAN, EDWARD XU

ABSTRACT. In this paper, we report our findings of using AI methods to create agents to play Ultimate Tic-Tac-Toe. We chose to focus on Minimax, Monte Carlo Tree Search, and a Deep Q-Learning Network as our methods. We simulated batches of games in a round-robin tournament with each of these agents, as well as a hybrid agent and a random agent. Using the Elo rating system to standardize the results, we found that Minimax was generally the strongest agent while Monte Carlo Tree Search performed the best against the random agent. Our results demonstrate how important it is for the design of an algorithmic agent to match its task. Both Monte Carlo Tree Search and Deep Q-learning Network are extremely powerful methods that could not perform as well as a simple Minimax program due to the rigid structure of Ultimate Tic-Tac-Toe.

1. INTRODUCTION

Over the last few years, popular games such as Go, Chess, and Atari have been "conquered" by artificial intelligence agents that are ever-increasing in popularity, power, and publicity [3], [5]. These games, which all were previously thought to require the highest mental capacity to master, demonstrate that neural networks can achieve superhuman performance in complex games. In this project, we seek to use AI methods to create an agent for ultimate tic-tac-toe.

Ultimate tic-tac-toe is a two-player game involving a 9×9 game board that is actually composed of nine "mini-boards" of regular 3×3 tic-tac-toe boards. Like in regular tic-tac-toe, one player plays 'x' in squares he or she wishes to occupy while the other plays 'o'. The first player can occupy any of the 81 squares on the board. After this, each player is limited to moves in the "mini-board" corresponding to the same square where the opponent last moved, relative to that "mini-board" (Figure 1, left). We have decided to explore the variant of Ultimate Tic-Tac-Toe where if a mini-board is already won but not full, any player can still be forced to play in the won mini-board. If a mini-board is completely filled out and a player's turn is in that mini-board, that player has the option of going anywhere on the board. A player wins the game by winning three individual "mini-boards" that connect in a line (Figure 1, right).

Date: June 13, 2019.

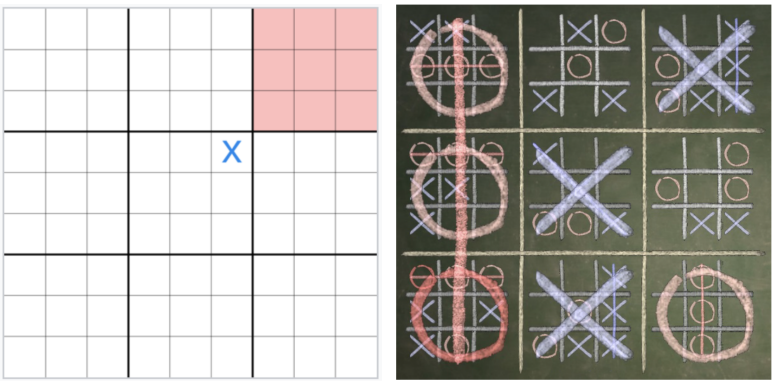

FIGURE 1. Left: Squares shaded red indicate valid moves after 'X' is placed. Right: 'O' wins the game after winning three "mini-boards" in the same column

We represent the challenge of playing ultimate tic-tac-toe as a search problem in an adversarial zero-sum game. The search space is reasonable as each state normally generates at most nine successor states, so minimax is a natural candidate for an inference algorithm.

The recent successes of AlphaGo Zero in Go, a zero-sum game suggest that Monte Carlo Tree Search and Deep Q-Learning are also likely to perform well. In Monte Carlo Tree Search, the agent simulates random traversals along the game tree in order to determine actions and states with the most promising outcome. This is effective in games with delayed reward because the final result of the game is actually propagated back to the state being evaluated.

In Deep Q-Learning, a neural network approximates the probability of winning at a given state, and the algorithm uses bootstrapped simulations to update the neural network. This is especially effective when the neural network can be activated with "winning patterns" in a game, such as AlphaGo Zero in the game of Go [3].

To evaluate the relative performances of the different algorithms, we use the standard Elo rating for zero-sum games [2]. In this system, higher rating points indicates a higher skill level, and the probability that a player wins is a logistic function of the distance between the ratings of the two players. Concretely, if player A has a rating of R_A and player B has a rating of R_B then the probability that player A wins is given by

$$E_A = \frac{1}{1 + 10^{(R_B - R_A)/400}}$$

Through this, we can examine how these different algorithms perform in ultimate tic-tac-toe and demonstrate how the structure of the problem affects the performance of algorithms.

2. Models

Since ultimate tic-tac-toe is a two-player adversarial game with at most one winner, it is natural to represent this game as a zero-sum adversarial search tree (Figure 2). The depths of the trees alternate between the two players. Each node of the tree represents a particular state and decision point of the game, where the player chooses a square on a mini-board, leading to an opponent's decision point.

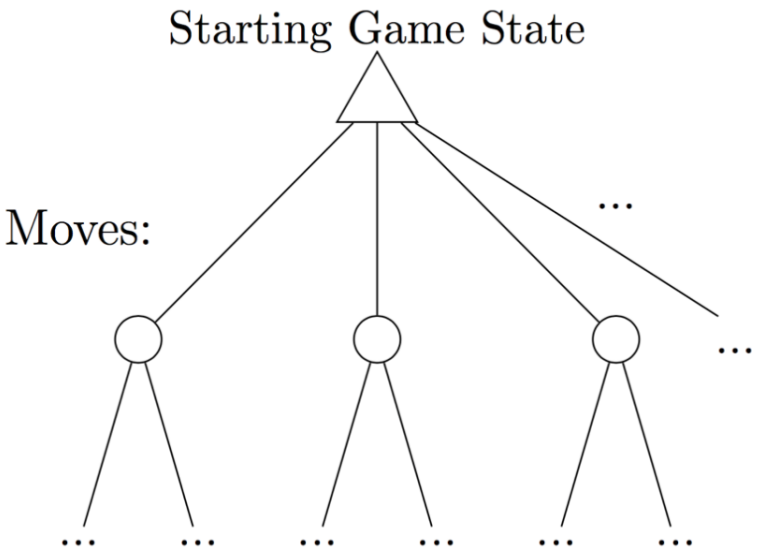

Figure 2. The game is modeled as a search tree, which our Minimax and MCTS agents will traverse to find optimal moves

The minimax agent uses alpha-beta pruning on a simple evaluation function to determine the optimal move in a relatively quick time interval. The evaluation function simply takes the number of "mini-boards" that the agent has won minus the number of "mini-boards" that the agent has lost. The motivation of this is to encourage the agent to perform moves that win additional mini-boards while avoiding moves that allow the opponent to win mini-boards.

The Monte Carlo Tree Search (MCTS) agent also traverses the game tree to determine the optimal move. Given a game state, we follow the standard MCTS algorithm using the upper confidence bound (UCB) (Figure 3): First, we sequentially select the node with the highest UCB, which balances exploration and exploitation. When we reach a node that has not yet been fully expanded, we select random moves and simulate a random game. Using the result of this randomly simulated game, we backpropogate to all the expanded nodes along the simulation to modify the UCB of each of these nodes. We note that the rollout step uses random simulation of moves until the end of the game, which may not accurately approximate the game tree against an intelligent agent.

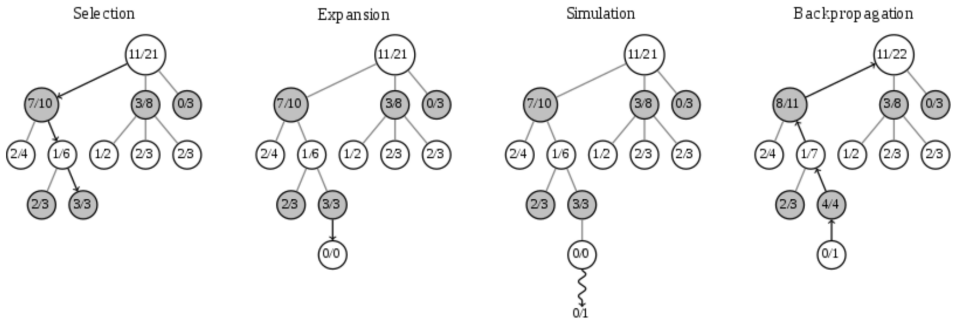

FIGURE 3. The generic Upper Confidence Bound Monte Carlo Tree Search algorithm involves a cycle of four steps. At the end of the search, the agent selects the node with the most visits

The deep Q-learning network (DQN) agent trains a neural network using reinforcement learning to evaluate the utility of a state. The DQN takes each game state in as an input and outputs a number between -1 and 1 denoting how likely the first player is to win. This number is the Q-value. To train the DQN, we use data from self-play. Initializing with random weights, we have the DQN play against itself, storing each board state, move, and final result in a memory cache. After each game, we use moves from the memory cache to create a batch of training data for the DQN. This closely follows the implementation of DeepMind's successful DQN agent for Atari [5]. The overall function of the DQN is summarized in Figure 4.

The neural network architecture involves the following: (1) The input tensor has shape $81 \times 81 \times 4$ and consists of four concatenated layers of the 81×81 board. The first layer has 1 in all the positions where x is marked and 0 in all other positions. The second layer has 1 in all positions with o and 0 elsewhere. The third layer has 1 marked across the entire board if it is player o's turn and 0 marked if it is player x's turn. The final layer has 1 marked in all valid squares for the next move and 0 elsewhere. (2) A convolution of 64 layers of size 3×3 with stride 3. (3) A fully connected layer to a hidden layer of size 16. (4) A tanh nonlinearity outputting a scalar in the range [-1, 1]

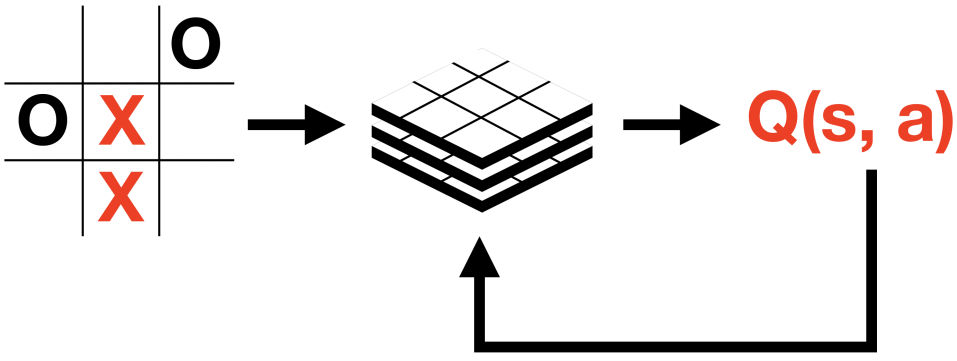

FIGURE 4. Deep Q-Learning Network architecture

We also created a hybrid agent between Minimax and MCTS. Because the board is sparse at the beginning of the game, we predicted that Minimax would be unable to distinguish between the utility different moves and that MCTS would be faster and more effective. Towards the later stages of the game, we predicted that Minimax would be more effective at finding optimal moves. Thus we created our hybrid to act as MCTS for its first 15 moves and Minimax thereafter. We also experimented with using MCTS with a Minimax-based rollout and MCTS with a DQN-based rollout, but both were less effective and slower than the original hybrid.

3. Results

To compare the agents to one another, we had 100 games between every pair of agents (i.e. random, minimax, etc.) and also accounted for which player went first (i.e. 100 games with random going first and minimax going second then 100 games with minimax going first and random going second). Figure 5 displays our win rates for 200 simulated games between the agent and the random agent (100 for going first, 100 for going second). Figure 6 displays the win rates of each against against the other agents.

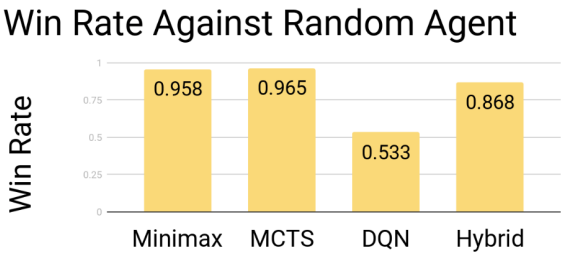

Figure 5. Performance of algorithms against random agent

Overall Win Rates

Player 1 \ Player 2	Random	Minimax	MCTS	DQN	Hybrid
Random		0.055	0.03	0.515	0.12
Minimax	0.97		0.72	0.98	0.63
MCTS	0.96	0.345		0.965	0.575
DQN	0.58	0.05	0.055		0.16
Hybrid	0.855	0.44	0.495	0.83	

Figure 6. Performance of agents against each other: The numbers in the table denote the proportion of games that player 1 won against player 2 (where a tie counts as half a win for both players).

To summarize the results of the head-to-head simulations in Figure 6, we used the Elo rating to capture the comparative skill level of each agent (Figure 7). To calculate relative Elo ratings, we initialized all agents with an Elo rating of zero and simulated round-robin tournaments between the agents until the Elo ratings converged, which required around 10,000 games. To simulate such a large volume of games, we treated the results of the head-to-head matchups in Figure 6 as win probabilities and sampled games with these probabilities.

With these results, it is clear that minimax was the superior agent by a good margin. MCTS and Hybrid agents were also competitive and far better than random and DQN, which had barely distinguishable performances.

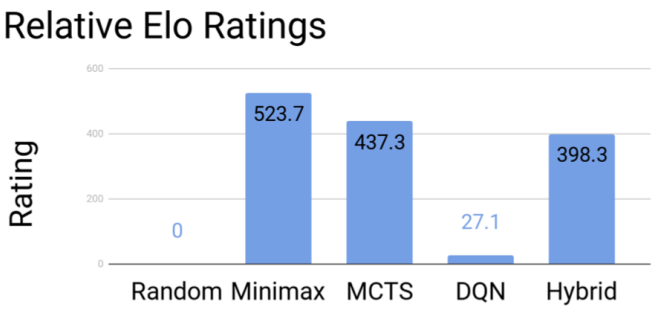

FIGURE 7. Elo ratings of the different agents based off of their performances against each other

4. Discussion

From the graphs and results above, we can see that the MCTS agent was best against random, winning around 96.5% of the time (this is calculated by averaging our results from random vs MCTS and MCTS vs random). This can be explained by the randomness that MCTS incorporates in the game tree, as it geared towards facing a random agent. Although MCTS fared the best against random, it did not do so good against minimax.

Minimax was the best overall agent, as it beat random around 95.8% of the time (this is calculated by averaging our results from random vs minimax and minimax vs random) and also consistently beat the MCTS agent around 68.7% of the time. As a result, it had the best Elo rating at 523.7. We concluded that minimax was able to beat MCTS because it expects the "worst case" situation while MCTS expects random play. Therefore, minimax is very intelligent during the endgame, while MCTS is usually only very effective during the beginning stages.

Even though the concept behind the DQN agent was extremely promising, it did not have good results, as it was almost as bad as random, winning only 53.3% of the time. It also did not fare well against either MCTS or minimax. This is because the architecture of DQN was not suited for how structured yet small the 9×9 board space was.

Our hybrid agent was a combination of minimax and MCTS, our two best agents. Since minimax is almost completely random in the beginning and more effective towards the end,

we decided to have our hybrid agent start off with a MCTS strategy. After there are at least 30 moves, the agent switches to using the minimax strategy with evaluation function. However, this hybrid was in fact less effective than both minimax and MCTS, with an Elo Rating of only 398.3. It still did well against random, winning 86.8% of the time, but it was not able to win a majority of games against its two originating agents.

We also tried implementing a different hybrid agent, which would keep a MCTS backbone strategy. Since our minimax evaluation function seemed to be working very well, we tried to incorporate minimax into the "rollout" phase of MCTS. Instead of doing a random move, we tried to move with what the minimax agent would choose. However, this ended up taking an abundance of time, even at less iterations and at a very low minimax depth.

Other attempts we found at creating AI agents to play ultimate tic-tac-toe have been in reinforcement learning [1] and better evaluation functions for minimax [4].

For the attempt with reinforcement learning [1], the agent was only able to win around 70% of the time as player 1 and win 60% of the time as player 2 through around 40,000 games. This attempt involved neural networks where the structure consisted of two layers: one input layer and one output layer each with 81 inputs (one for each square). With this, the neural network trained on the 40,000 learnt values and became ready to play random agents. Compared to our attempt with a Deep Q-learning network and less self-play data, our agent won only about 12% less games (but both this attempt and our attempt are much worse compared to our minimax and our MCTS versus random). This attempt also faced similar challenges as us regarding the unsuitability of the DQN's architecture. One method of improvement is trying out different structural forms of the neural network.

For the attempt with the better evaluation functions for minimax [4], the attempt involved multiple heuristic functions with two strong ones: one (A) that recursively returns a value pertaining to the current board state of who's in the lead, and one (B) that builds on the recursive heuristic with a more sophisticated calculation. These performed very well with a 92.5% win rate for heuristic B against a random agent and a 79% win rate for heuristic A against a random agent. The results for heuristic B are close to what we saw with our evaluation function for minimax. It's interesting to note that these heuristics and our heuristic were similar in calculating values that correlated with which player is in the lead based off the current state of the board. One interesting feature brought up for further discussion in this attempt was that there was not enough time-space complexity to be able to handle more depth in the minimax algorithm (a problem that we also saw). If we could explore more possibilities from the current state more quickly, this could lead to better moves made by our agent.

5. Conclusions

After exploring three main AI agents for Ultimate Tic-Tac-Toe, we can see that there are many complications to completely simulating a "winning strategy". Although our MCTS agent was best against a random opponent, the simplest agent, minimax, turned out to be the strongest against "intelligent agents". This demonstrates how effective minimax is in zero-sum games, especially those with a limited number of successor states.

Neural network architecture is very important for a problem with so much structure, but for Ultimate-Tic-Tac-Toe, it is very difficult to generate data, and there is not an abundant amount of data that exists for this particular game. As a result, there needs to be more data for neural network architecture to function effectively along with trying out different structures.

6. Code

Our code and data can be found in the CodaLab worksheet here:
https://worksheets.codalab.org/worksheets/0x056c7b186e834b37b4825a87e99aa6bc/

References

[1] S. Banerjee. Using reinforcement learning to play ultimate tic-tac-toe. *Medium*, 2017.
[2] A. Elo. *The Rating of Chessplayers, Past and Present*. ISHI Press International, 2008.
[3] D. S. et al. Mastering the game of go without human knowledge. *Nature*, 2017.
[4] E. A. et al. *AI agent for Ultimate Tic Tac Toe Game*. 2014.
[5] V. M. et al. Playing atari with deep reinforcement learning. *NIPS*, 2013.