Playing Ultimate Tic-Tac-Toe with TD Learning and Monte Carlo Tree Search

Karan Singhal and Varun Nambikrishnan

Abstract—We explore ultimate tic-tac-toe by building several agents and simulating games between them to compare their performance. We start with a Random Agent, which randomly chooses an action from the possible actions, and we move toward a Minimax Pruning Agent and an Expectimax Agent, which search a game tree to a maximum depth before relying on a linear evaluation function learned through TD learning updates and Monte Carlo Tree Search.

1 Introduction

Ultimate tic-tac-toe is a significantly more complex game than its more famous cousin. While tic-tac-toe is a simple enough game that a player can play a perfect game simply by following the strategy used in Newell and Simon's 1972 computer program [1], the version of Ultimate tic-tac-toe we explore in this paper has no documented solution.

1.1 Rules of the Game

The board of Ultimate tic-tac-toe is composed of nine smaller boards of regular tic-tac-toe arranged in a 3x3 grid, producing an 81-square board, as illustrated in Fig. 1. On each turn, a player places its symbol (typically an 'x' or an 'o') on a square. The goal of the game is to win three of the smaller boards in a row, either horizontally, vertically, or diagonally.

The main rule addition of Ultimate tic-tac-toe is that, after the first move, which can be anywhere, a player must play on the smaller board corresponding to the position of the last square played. For example, if player 1 plays in the top-right square of any smaller board, player 2's next move must be a square in the top-right smaller board. If at any time a player has nowhere to play after the previous move as a result of this rule, then the player can play anywhere that is open. This introduces an element of strategy to the game, as for each turn a player's action limits the potential actions of the opponent on the next turn. In the version of the game we implemented, if the board the current player would be forced to play in has already been won or tied, she can play on any of the other boards still in play, even if that board has open squares.

1.2 Prior Work

There is very little prior work on advanced agents for Ultimate Tic-tac-toe. James Irwin created a "perfect" bot to play Ultimate tic-tac-toe, and his implementation is on Khan Academy [2]. However, in the version of the game he implements, a player must play on the smaller board corresponding to the position of the last square played as long as there are squares left to play in that board. In other words, a player can be forced back to an already won board as long as there are open squares in it, thereby making their next move useless. With this rule, the first player always has a strategy to win, which involves repeatedly sending the second player back to three boards, one of which being the center board. However, in our implementation of the game, this strategy would not work, as a player cannot be forced to play in an already won board.

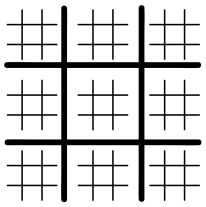

Fig. 1. Empty game board for Ultimate Tic-tac-toe

2 Methods

To explore ultimate tic-tac-toe, we started by formally specifying the game and building a simulator that can play games between two agents that each produce actions to play if given a game state.

2.1 Game Specification

Ultimate tic-tac-toe is a two-player, zero-sum game. There is a single start state, various possible actions that can be taken at a given state, a deterministic successor state given a state and an action, and several possible end states. The end states have utilities associated with them based on whether the end state is a win, loss, or draw. The specifics of our representation are as follows:

- **State** = A tuple containing the board state (a list of 9 smaller boards), the player whose turn it is, and the position of the last square played in the smaller board that contains it.
- **player(state)** = 1 or 2, representing the agent and the opponent
- **startState()** = 81 square board made up of 9 smaller boards with all squares unplayed, it is the first agent's turn, no last square played

- **actions(state)** = All possible actions from a state based on the rules of the game.
- **succ(state, action)** = The successor state given a state and action.
- **isEnd(state)** = Determines whether the state represents the end of a game.
- **utility(state)** = Determines the utility of an end state. In our case, +100 for an agent win, -100 for an opponent win, and 0 for a draw.

The game structure can be represented as a tree, where the nodes represent states and the edges represent actions. In this two-player game, each alternating level of nodes corresponds to a player turn.

2.2 Agents

We built several agents to play against each other using our simulator, which allowed a specified number of Ultimate tic-tac-toe games to be played between any two agents.

2.2.1 Random Agent

The Random Agent simply obtains the possible actions from the current state, and chooses a random action. We built this agent as a benchmark for other agents' performance.

2.2.2 Perceptron Agent

The Perceptron Agent uses global function approximation to choose an action from the current state. The agent uses a Simple Feature Extractor and an arbitrarily chosen weight vector (both detailed in Section 2.3.1) to first give a score to each of the possible actions from a state, and then chooses the action with the maximum score. If multiple actions are tied for the maximum score, then the agent randomly chooses one of them.

With the Perceptron Agent, we aim to build a more advanced benchmark for other agents, with an agent that actually does fairly well at regular tic-tac-toe (the simple feature extractor features and weight vector used by this agent were chosen to do this). To test whether Perceptron Agent actually does well in regular-tic-tac-toe, we specified the regular tic-tac-toe game, built a simulator for it, and created Simple Perceptron Agent and Simple Random Agent, which are the regular tic-tac-toe analogues of Perceptron Agent and Random Agent.

To establish a baseline, we found that after running 10,000 games between the Simple Random Agent and itself, the first agent won with probability .58, tied with probability .13, and lost with probability .29. When the Simple Perceptron Agent played against the Simple Random Agent 10,000 times, it won with probability .92, tied with probability 0, and lost with probability .08. In these games, the Simple Perceptron Agent always went first. To control against the significant advantage of going first, we also had the two agents play 10,000 games in which Simple Random Agent went first, and in those games the Simple Perceptron Agent won with probability .71, tied with probability 0, and lost with probability .29.

These preliminary results indicate that Simple Perceptron Agent is well-prepared for regular tic-tac-toe, even if it loses the advantage of going first. This indicates that Perceptron Agent will be a nontrivial benchmark for ultimate tic-tac-toe agent performance.

2.2.3 Reflex Agent

The Reflex Agent is similar to the Random Agent in that it chooses a random action from the set of possible actions, but it filters the actions before choosing them. An action may be chosen provided that there is no possible opponent action on the next turn such that the opponent will win a smaller board. If no actions that satisfy this condition are available, then the reflex agent chooses a random action.

The purpose in implementing the Reflex Agent was to have a benchmark agent with foresight, avoiding actions with clearly detrimental effects, if only in the short term. This element of foresight is key in Ultimate Tic-tac-toe, where any well-performing agent will certainly consider the potential consequences of any action it chooses.

2.2.4 Minimax Agent

The Minimax Agent uses game tree search to choose the action that will maximize the agent's utility, assuming that the opponent seeks to minimize the agent's utility (this is desirable for the opponent because this is a zero-sum game). Since its calculation assumes the worst-case behavior from the opponent, the minimax value establishes a lower bound against any opponent policy.

Computing these values exactly is too computationally expensive for a game with such a large state space. In order to approximately compute the minimax value, we use a depth-limited search of the game tree, where we compute $V_{max,min}(s, d)$, the approximate value of the state s given the maximum depth d we are allowed to search the game tree, as follows:

$$V_{max,min}(s,d) = \begin{cases} Utility(s) & isEnd(s) \\ Eval(s) & d = 0 \\ max_{a \epsilon A(s)} V_{max,min}(Succ(s,a), d) & P(s) = agent \\ min_{a \epsilon A(s)} V_{max,min}(Succ(s,a), d-1) & P(s) = opp \end{cases}$$

Eval(s) is an evaluation function we invoke at d = 0 that uses global approximation to estimate the predicted utility of the game at a given state. This is a linear evaluation function, using the Advanced Feature Extractor described in Section 2.3.2. Key to the success of the Minimax Agent is how the weights for the linear function are learned.

Our strategy was to consider two different types of execution for the Minimax Agent: training and testing.

During training, we use Monte Carlo Tree Search and TD learning to update weights at each turn. For each action requested by the simulator, the agent simulates playing until the end of the game with an opponent that seeks to minimize utility. Specifically, during this simulation, the agent chooses an action whose successor has the greatest score under the current evaluation function (a random one if multiple exist). The opponent chooses an action whose successor has the lowest score under the current evaluation function, and this continues until the end of the game.

We record the history of the game as tuples of state, action, reward, and successor, or (s, a, r, s'). Note that the reward will be 0 in every case except if the successor state is the end state. In that case, the reward is just the utility of that state.

We use the generated history in our TD learning update, which seeks to minimize the loss function below:

$$\frac{1}{2}(\underbrace{\phi(s)\cdot w}_{pred}-\underbrace{(r+\phi(s')\cdot w)}_{target})^2$$

Using stochastic gradient descent, taking the gradient with respect to weights of this loss function produced the following update based on the history:

On each (s,a,r,s'):

$$w = w - \eta[\underbrace{\phi(s)\cdot w}_{pred}-\underbrace{(r+\phi(s')\cdot w)}_{target}]\phi(s)$$

where η = .00001

We iterate through the (s, a, r, s') tuples of the episode in reverse order, performing this update. In the case where s' is an end state, we set the target in the update to be just the reward, to ensure the weights converge.

We repeat the process of Monte Carlo Tree Search and TD learning using recorded histories 25 times for every action requested of the Minimax Agent by the simulator.

Then after the training phase, the model's weights are frozen, and the Monte Carlo Tree Search and TD learning steps are skipped. Eval(s) is then just a linear function of the Advanced Feature Extractor and some static learned weights. We are then able to calculate the minimax value of any state with confidence, and to find the optimal action from any state, we simply choose the action whose successor has the greatest minimax value, or a random one if multiple actions that achieve the maximum exist.

To decrease runtime for this algorithm, at each step of the recurrence at which the current state is not an end state and d is not 0, minimax values for successors are only calculated for the top (or bottom, if P(s) = opp) three successors, as determined by the evaluation function. This is to ensure minimax values are only calculated for successors that are likely to have the highest (or lowest, if P(s) = opp) values, avoiding possibly unnecessary calculations.

2.2.5 Minimax Pruning Agent

The Minimax Pruning Agent is the same as the Minimax Agent with alpha-beta pruning, an optimization that reduces the runtime of the tree search without changing its results. At a high level, alpha-beta pruning prunes branches of the game tree that cannot possibly contain nodes that achieve the minimax value.

For any maximum node (a node in the game tree in which it is the agent's turn), alpha-beta pruning updates the alpha value, or the maximum minimax value of any of its successors so far. This alpha value represents the minimum minimax value of the node, since the node will maximize over its successors. For any minimum node (a node in the game tree in which is is the opponent's turn), alpha-beta pruning updates the beta value, or the minimum minimax value, of any of its successors so far. This beta value represents the maximum minimax value of the node, since the node will minimize over its successors.

Both types of nodes pass any updates to alpha and beta to their children, and if any of their children at any point satisfy the condition alpha is less than beta, this indicates that the minimum minimax value of a node is greater than the maximum, which is an impossible condition. Then, it is not possible that the given node leads to the bottom-most node in the tree that has the desired minimax value, and the node and its successors can be pruned.

This optimization had the effect of reducing runtime for trials involving hundreds of games from days to hours, without affecting results. As such, we used the Minimax Pruning Agent instead of the Minimax Agent to run simulations for our results.

2.2.6 Advanced Perceptron Agent

The Advanced Perceptron agent is exactly the same as the Perceptron agent except it uses the Advanced Feature Extractor and the weights learned by TD learning and Monte Carlo Tree Search (the same weights used by the Minimax Agent and Expectimax Agent after training), as described in section 2.3.2.

This agent serves as a benchmark for the more advanced agents that we implemented that do take into account the strategic differences between regular tic-tac-toe and ultimate tic-tac-toe. It is exactly the same as Minimax Agent or Expectimax Agent with depth 0.

2.2.7 Expectimax Agent

The Expectimax Agent is the same as the Minimax Agent except that it assumes the opponent will make a random action instead of the one that minimizes the agent's utility. During traversal of the game tree, if it is the opponent's turn, Expectimax Agent recursively returns the average expectimax value of its children, as follows:

$$V_{max,opp}(s,d) = \begin{cases} Utility(s) & isEnd(s) \\ max_{a\epsilon A(s)} V_{max,opp} Succ(s,a) & P(s) = agent \\ \frac{\Sigma_{a\epsilon A(s)} V_{max,opp} Succ(s,a)}{|A(s)|} & P(s) = opp \end{cases}$$

2.3 Feature Extraction

The feature extractors we created compute features given a state, which are used by some of the agents to perform function approximation.

2.3.1 Simple Feature Extractor

The simple feature extractor considers the following features:

- **Wins:** The number of smaller boards the agent has won.
- **Center Pieces:** The number of squares in the center of the smaller boards that the agent controls.
- **Corner Pieces:** The number of squares in the corner of the smaller boards that the agent controls.
- **Adjacent Pieces:** The number of squares that the agent controls that have at least one other agent-controlled square adjacent to it, either vertically, horizontally, or diagonally

TABLE 1
RESULTS

Agent 1/Agent 2	Random	Perceptron	Reflex	Minimax 1	Minimax 2	Advanced Perceptron
Random	.39, .27, .34 (1000)					
Perceptron	.64, .12, .24 (1000)	.56, .04, .40 (5000)	.54, .15, .31 (1000)			
Reflex	.42, .35, .26 (1000)		.31, .42, .27 (1000)			
Minimax 1	.55, .20, .26 (1000)	.38, .11, .51 (1000)	.43, .25, .32 (1000)	.50, .12, .37 (500)		.78, .08, .14 (1000)
Minimax 2	.51, .23, .26 (500)	.32, .10, .58 (500)	.38, .31, .31 (500)	.56, .11, .32 (500)	.48, .16, .36 (300)	.77, .09, .14 (500)
Advanced Perceptron	.71, .07, .20 (5000)	.71, .04, .25 (5000)	.57, .14, .29 (1000)			.93, .06, .01 (5000)
Expectimax 1	.98, .01, .01 (1000)	.87, .09, .04, 750	.93, .05, .02 (750)			.96, .03, .01 (500)
Expectimax 2	1.0, 0.0, 0.0 (56)*	1.0, 0.0, 0.0, (55)*				

This table contains the results of the simulations of Ultimate tic-tac-toe games between our agents. Each cell contains the results of games between the row agent (Agent 1) and the column agent (Agent 2). The numbers in the cells represent: the percentage of games Agent 1 (the row agent) won, the percentage of games tied, the percentage of games Agent 1 lost, and the total number of simulations run, respectively. Some simulations took too much time to run, and the * indicates all the simulations did not finish running by submission. Blacked out cells indicate that simulations were not run between the specified agents.

This feature extractor is used by the less sophisticated agents. While the feature extractor was effective in allowing the Perceptron Agent to win the regular Tic-tac-toe game, it fails to take into account the strategy of forcing the opponent to play in specific boards on the next move, which is a crucial aspect of Ultimate Tic-tac-toe. The Perceptron Agent used the weights:

”wins” : 5, ”centerPieces”: .2, ”cornerPieces”: .1, ”adjacentPieces”: .4,

which were chosen arbitrarily.

2.3.2 Advanced Feature Extractor

The advanced feature extractor considers the following features:

- **Wins/otherWins** The number of smaller boards the agent/opp have won
- **Relative Wins:** The number of smaller boards the opponent has won subtracted from the number of smaller boards the agent has won.
- **Center Pieces/Other Center Pieces:** The number of center squares the agent/opp control
- **Corner Pieces/Other Corner Pieces:** The number of corner squares the agent/opp control
- **Adj Pieces/Other Adj Pieces:** The number of squares the agent/opp control that have at least one other agent/opp controlled square adjacent to it
- **Adj Grids/Other Adj Grids:** The number of smaller boards the agent/opp control that have at least one other agent/opp controlled board adjacent to it.
- **gridsDifference:** The number of grids the opponent controls subtracted from the number of grids the agent controls.
- **advantageOfNextGrid:** The number of agent adjacent squares controlled in the smaller board where the current player must play next. Positive if it is the agent’s turn, and negative if it is the opponent’s turn.

This feature extractor is used by the more sophisticated agents. At a high-level, it combines the insights of the simple feature extractor with a better understanding of the dynamics of Ultimate Tic-tac-toe. For example, it considers the number of adjacent grids in addition to the number of adjacent pieces, as the way to win the whole game is to get three grids in a row. It also considers the advantage of making the next move in the grid specified by the position of the last move, since this limits the potential actions in Ultimate tic-tac-toe.

Another advantage of the Advanced Feature Extractor over the Simple Feature Extractor is that it considers the progress of the opponent toward victory as well as the agent, with features corresponding to opponent in addition to the agent for most features.

The weights we learned through Monte Carlo Tree Search and TD learning after an hour of learning were:

’numCornerPieces’: 0.811020200948285, ’numOtherCenterPieces’: 0.0809559760347321, ’relativeWins’: 1.0153244869861855, ’numAdjacentWonGrids’: 0.969296961952278, ’numOtherAdjacentWonGrids’: -0.6814807021505367, ’advantageOfNextGrid’: 0.5395673542865317, ’numOtherCornerPieces’: 0.7058793287408335, ’gridsDifference’: 1.6507776641028122, ’numAdjacentPieces’: 0.7130969880480094, ’numWins’: 0.5323490704404454, ’numCenterPieces’: 0.021778991208712777, ’otherWins’: -0.48297541654574627, ’numOtherAdjacentPieces’: -1.2963960051123138

3 Discussion

In this section, we will highlight important results and discuss their implications on our model and possible improvements.

3.1 Random Agent Baseline

In playing against itself, the random agent has a .39 probability of winning, a .27 probability of tieing, and a .34 probability of losing. This indicates the slight advantage of going first in Ultimate Tic-tac-toe if both players have no strategy, which is significant but small enough that we will generally not attribute larger disparities in probabilities of winning between agents to this effect.

3.2 Perceptron Agent

Despite winning with probability greater than .9 in the regular tic-tac-toe game against Random Agent (with Simple Perceptron Agent), Perceptron Agent does not perform as well in Ultimate Tic-tac-toe, with a win rate of .64. This indicates the complexities of Ultimate Tic-tac-toe and demonstrates that any performant agent in this game must take into account its unique rules.

3.3 Reflex Agent

As we expected, the reflex agent did not do much better than the random agent, with a win rate of .42 as compared to the baseline .39. This is likely because although the reflex agent uses a small degree of foresight in choosing its actions, when it does so, it does not often make a difference: by the time the next action may cause an opponent to win a smaller board, most of the smaller boards are close to full, meaning the reflex agent plays randomly for most of the game.

When the Perceptron Agent plays against the Reflex Agent, its win rate is .54, significantly lower than its .64 win rate against the Random Agent. This makes sense because the Perceptron Agent is more directed in its move choice, meaning the reflex agent is put into situations where it is able to filter actions earlier in the game, meaning the Reflex Agent acts less like a random agent.

3.4 Advanced Perceptron Agent

Our first agent that uses learned weights and the Advanced Feature Extractor generally performed well. It has a .71 win rate against the Random Agent, as well as the Perceptron Agent. However, it did not do as well against the Reflex Agent, with a .57 win rate. This is likely because the Advanced Perceptron Agent is very directed and opinionated in its choices for its actions, so it chooses the actions that lead to victory the fastest, thus testing the foresight of the Reflex Agent earliest in the game.

Interestingly, the Advanced Perceptron Agent has .93 win rate when played against itself. Again, this is likely because it prefers actions that lead toward quick-victory. In light of this, whichever player takes the most valued positions (according to the resulting successor state scores under the evaluation function) first has an advantage, so the first player will usually maintain an advantage if they have the same highly directed strategy.

3.5 Minimax Pruning Agent

The Minimax Agent's performance depended quite a bit on the nature of its opponent. Since it assumes the opponent is minimizing utility, it did not do as well against the Random Agent (win rate .55 with d = 1) and Reflex Agent (win rate .43 with d = 1) as it did against other agents, like the Advanced Perceptron agent (win rate .78 with d = 1).

In general, the Minimax Pruning Agent did not perform quite as well as we imagined it would. One interesting observation is that the Perceptron Agent was actually able to beat the Minimax Agent (its loss rate against the Perceptron Agent was .51 with d = 1). Both this and the subpar performance against the Random and Reflex Agents may be explained by the degree of randomness by which these agents made their actions: even for the Perceptron Agent, ties are settled by randomness. Ties are likely frequent for these simpler agents, as they do not rely on many factors of the game state to differentiate successors of actions (the Reflex Agent only relies on possible losses of smaller boards, and the Perceptron Agent relies on a limited and simple set of features).

However, since the game is strategically complex, ties should not be frequent in more sophisticated models, like the Advanced Perceptron Agent. Since the Minimax Agent assumes the opponent seeks to minimize its utility and acts accordingly, the degree of randomness by which simpler agents act reduce its effectiveness. It can be proven that while the Minimax Agent is best against an optimally minimizing agent, it is not necessarily the best agent against anything else.

3.6 Expectimax Agent

We ran simulations using this agent to test whether it was actually the minimizing assumption that was causing the subpar performance of the Minimax Agent. The argument turns out to be convincing, as the Expectimax Agent wins almost every game it plays: its win rate against the Random Agent was .98 with d = 1, against the Reflex Agent was .93 with d = 1, and even against the Advanced Perceptron Agent was .96. In general, more random agents appeared to do worse against the Expectimax Agent as expected.

One interesting result was that the Expectimax Agent performed less well against the Perceptron Agent, with a win rate of .87 and a tie rate of .09. This is likely because the Perceptron Agent places a high weight on the number of center pieces for the agent, so it prevents the Expectimax Agent from winning by placing pieces on the center wherever possible, thereby increasing the chance of draws (in regular tic-tac-toe, doing this and playing optimally guarantees at least a draw).

4 Future Work

Even with our current agents, there is much room to improve performance in Ultimate tic-tac-toe. Since there has not been much work on this game in the past, we encourage those who are interested in further work to peruse our code on Github for a starting point: https://github.com/karan1149/ultimate-tic-tac-toe

4.1 Learning

For the learned weights used by the Advanced Perceptron Agent, Minimax Agent, and Expectimax Agent, we did not attempt to learn them after our initial hour of training. It is likely that additional training would be helpful, since the weights did not converge before we stopped training.

It is also likely that the weights should have been learned independently for the agents that make use of them. Ideally, the evaluation function is a predictor for the of the expected utility of a state given the strategy of the agent. For example, the evaluation function should evaluate to the expectimax value for the Expectimax Agent, and it should evaluate to the minimax value for the Minimax Agent.

Another obvious area of improvement is constant optimization, of which we did none. We used 25 simulations for Monte Carlo Tree Search and a step size of .00001, but we did not adjust these constants for the sake of performance.

4.2 Agents

One simple addition to this work would be the testing of different constants for the filter for top (or bottom) actions by evaluation function in the Minimax and Expectimax Agents. We introduced this optimization arbitrarily and did not rigorously test the tradeoff between accuracy and performance involved in this optimization, nor did we test the optimal number of actions to filter.

Improvements in our Advanced Feature Extractor would ideally further identify differences between similar successor states, reducing the amount of randomness in picking actions for our more advanced agents. Possible improvements also may further consider the advantage the next move being constrained to a particular subgrid has for the agent, as this is a key element of the game and how it differs from regular tic-tac-toe. Our approach attempted to capture its complexities using only one linear feature, which probably was not ideal. Another possible improvement would be to better capture the complexity of subgrid relationships. For instance, for an otherwise empty game board, it is about equally advantageous to capture the top-left subgrid of the game and the bottom-left subgrid of the game as it is to capture the top-left subgrid and the middle-left subgrid, but our features, which rely on adjacencies between captured subgrids, do not capture this.

Another possible agent would be an agent that implemented a weighted version of our Monte Carlo Tree Search and TD learning, in which the agent is more likely to train on and simulate branches of the tree that have led to its victory in the past.

Another key possible improvement to our work would be to use a neural network as the function approximator instead of linear functions for the agents that involved these functions. This would allow us to easily address nonlinearities in the expected utility relative to the features, and would allow for a more inclusive class of predictor functions to choose between during training.

Acknowledgments

We would like to thank Dr. Kochenderfer and the course staff for CS238 for their work throughout the quarter.

Group Contributions

Karan worked on implementing the simulator and algorithms for the agents. Varun worked on specifying the game formally, implementing functions that access game structures, and implementing functions for the feature extractors. Both worked on the paper.

References

[1] Newell, A., Simon, H.A. (1972). *Human problem solving.*, Englewood Cliffs, NJ: Prentice-Hall.

[2] Irwin, J. (n.d.). *In-Tic-Tac-Toe-Ception (Perfect)*. Retrieved December 07, 2016, from https://www.khanacademy.org/computer-programming/in-tic-tac-toe-ception-perfect/1681243068