HW2: Tic-Tac-Toe Reinforcement Learning

Cynthia Dong, Conor Thomason

Our algorithm learns to play tic-tac-toe using model-free reinforcement learning. Two agents play each other to a win, loss, or draw in each trial; both agents contribute to the utility function. Initially, an empty tic-tac-toe board is created, as well as an empty utility board (set to 0 for every cube). A potentialMoves variable contains all of the spots still open on the tic-tac-toe board, and two variables, p1Moves and p2Moves, contain all of the moves made by the respective players (initialized to be empty). The learn function generates a utility value for each cube in the tic-tac-toe board.

Once one agent wins the game, the learn function calls a calculate function. calculate goes through all of the moves the winning agent made and, starting with a reward of 1, applies the function U(newCube) = U(currentCube) + ɑ*qMax, where ɑ is a constant factor that places more value on the final moves, as opposed to the first moves made. qMax is the reward value; here, it is 1 for a win. We do not apply a negative factor for a loss or draw. As the calculate function iterates backwards through the moves made by the winning agent, the qMax value depreciates by a factor of ɑ. ☐ was chosen through trial and error.

The learn function relies upon a random distribution of exploration vs exploitation, decided upon by using a learningRate and a random number generator. During exploitation, learn takes the best possible value from the utility board and applies it as the current move. Conversely, for exploration, if a random number is established that is less than or equal to the exploration rate (initially set to 0.25), a random cube is chosen from potentialMoves, and the explorationRate is decreased by a factor of 0.95. As a result, as the game proceeds, the explorationRate decreases, forcing the game to explore more as the utility values reach an equilibrium. Utility values stabilize towards maxima in the outer corners and the core of the board as learning progresses. This means that winning moves will always end up being in the edges, as "optimal" moves in the core and corners will be occupied first. Higher exploration rates towards the end of the game help mitigate that. Having an initial exploration value that strengthens towards the conclusion of the game allows the current utility values to be tested and potentially reevaluated, forcing a break in any patterns that may have emerged as a result of the progression of learning.

The wincheck function gets called every time a move is made. This function starts at the coordinates of the most recent move and searches the row, column, depth, and (if applicable) diagonals that run through that location. If the player that made the move gets four in a row, wincheck returns a 1, which tells the algorithm that the game is over.

The play method takes advantage of the learn method by forcing iteration by a hard-coded number of loops. This generates a utility board that can be used during play - the result will not have any impact on the utility values already generated. The play method uses two additional lists - winZones and dangerZones. These are initially checked before any move is decided as a result of the utility board, as any other move made aside from these will be either suboptimal play or strategically disadvantageous.

winZones and dangerZones are checked via the strategyCheck function, which looks for cases of 3-in-a-row for either player after the opponent has moved. If it's 3-in-a-row for the AI, the fourth coordinate in that row would be added to winZones. If it's 3-in-a-row for the opponent, the fourth coordinate would be added to dangerZones. The next time that the AI looks for a move, it checks winZones first, as winning would cut off the opponent's play in dangerZones. Next it checks dangerZones to block the opponent, and finally if dangerZones and winZones are empty, the AI will pull from the utility board.

After 25 moves, the AI will start checking for 2-in-a-row and add those to dangerZones and winZones as well; we didn't want it to check those too early and negate the advantage provided by the utility board.

To normalize the values (as you run more loops, the utility values balloon) we took a sum of the utility board values before returning it, divided that by 64, and divided the value of each cube by that value.

Individual Contributions

Cynthia Dong

- Wrote initial learn, winCheck, and calculate functions

Conor Thomason

- Set up unit testing
- Tested initial learn function
- Made modifications to exploration/exploitation

The rest was real-time collaboration, including the play function and strategyCheck.