Training an automated Tic Tac Toe player using reinforcement learning

Introduction

Tic Tac Toe is both a simple game and a simple problem for reinforcement learning, which is why I chose it as my first attempt to implement reinforcement learning techniques. The problem is simple because:

- As for chess and other similar games, both the "state space" (the set of all possible states) and the "action space" (the set of all possible moves) are discrete and finite rather than continuous and infinite. In other words, there are a limited countable number of possible states and a limited countable number of possible actions (i.e., moves) that can be made. This is different to a situation of a driverless car, say, where its speed and location are continuous variables that have an infinite number of possible values. Similarly the "action variables" of throttle position, brake pressure and steering angle are also continuous. In the case of these actions, the designer can choose to constrain the actions to a limited number (e.g., in the case of throttle by limiting the choices to full throttle, half throttle and no throttle, say). However this is a choice with consequences that must be considered, whereas in Tic Tac Toe the number of possible actions are limited by the nature of the game itself.
- There are relatively few states possible compared to more complex games like draughts and chess. Indeed the game is so simple that the perfect strategy can be derived using techniques other than machine learning.
- The game mechanics are very simple. This means the bulk of the design and implementation effort can be focused on the learning aspects rather than on modelling those game mechanics.
- The game model is deterministic, in that given a particular state and a given action (i.e., move) the new state resulting from that state is always known with 100% certainty. This is not the case with many learning problems, either because:
  - The model is stochastic (i.e., outcomes are only known with probabilities rather than with certainty, such as playing a card game like blackjack); or
  - The model is not known at all (e.g., exploring a maze).
- The learner has a "stationary" target, meaning that the goal the learner is trying to achieve is always the same. This contrasts, for example, with a tracking problem where the target is moving, either in a planned way or a random stochastic manner.

Objective

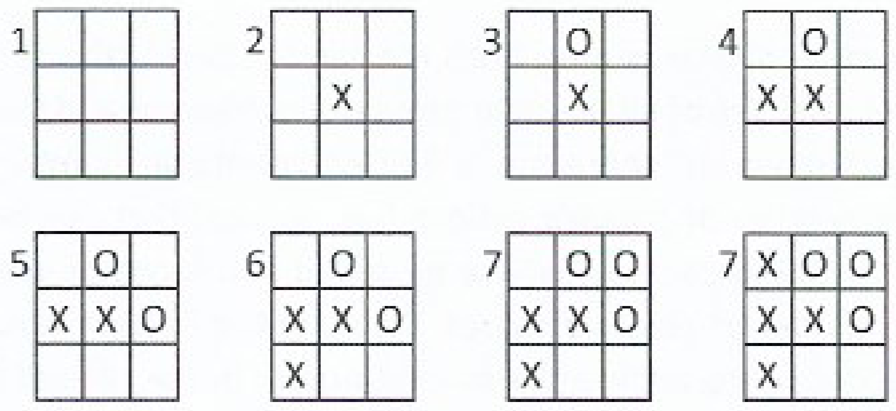
The objective I set myself was to develop and train an automated Tic Tac Toe game that learns how to play the game by playing against itself.

Approach

My approach was as follows:

- Create a class "Learning Player", and create two instances of the class, each of which has no initial knowledge of the "best" move to make.

- Have the two players play each other many times, one always playing as X and one always playing as O.
- During each game, each player records each "state" visited during the game i.e. where Xs and Os are positioned after each move.
- The players learn using an episodic Monte Carlo approach. At the end of each game, each player updates their "memory" of states and results. For example, let's assume one game follows this sequence of moves:



This game has resulted in a win for X and a loss of O. Each player's "memory" can be thought of as taking the form of a table as shown below. Each state that has been visited in one or more games is represented as a single row – by combining the three rows of the game grid together (i.e., the first three entries represent the first row, the second three entries the second row, and the third three entries the third row).

State	Visits	Wins	Draws	Losses
- - - - X O - - -	2	2	0	0
- O X - - - - - -	1	0	1	0
- - - - X O O - X	1	1	0	0
- O - - X O - X -	1	1	0	0
- O X - X - O - -	1	1	0	0
|		|		
etc.		etc.		

The extract of the memory above is for the player O as the player only needs to store those states where it has just made a move (as will become clear later). The table shows the records "memorised" assuming only three games have been played.

- When choosing which move to make next, each player uses an epsilon-greedy approach where epsilon is set to 0.7. This means that each time the player must move, the player either:
  - Chooses the "best" move based on its memory; or
  - Chooses a move at random.

Each time a player must make a move, the player "decides" to choose a random move with probability of epsilon (set to 0.7), and the rest of the time the player chooses the "best" move according to its memory. In other words, on average the player chooses the best move 70% of the time. The other 30% of time, the player chooses a move at random with uniform probability. This ensures that the player continues to explore moves that may not have been tried enough or at all.

- If the player has "decided" to use the best move, it must then decide which of the possible moves is the best. To do so, the player looks up in its memory all the states that would result from the possible moves from the current state. For example, below is an example of a current game state and the possible moves that X can then make, together with the "memory" of results for each of those "after-states". The player has calculated a score for each after-state on the basis that a win counts as +1, a draw as 0, and a loss as -1, and this is then averaged over the number of visits to that after-state. So for example, the score for the first possible move is calculated as (21 + 10 + 4) / 35. The player then chooses the move with the highest score.

Current state

- - - - X O O - X

Possible moves	Visits	Wins	Draws	Losses	Score
X - - - X O O - X	35	21	10	4	0.48571
- X - - X O O - X	15	2	9	4	-0.13333
- - X - X O O - X	12	1	1	10	-0.75000
- - - X X O O - X	27	4	20	3	0.03704
- - - - X O O X X	7	0	3	4	-0.57143

Results

By creating an interface that allowed a human player to player against the application with epsilon set to zero, I could then examine whether the perfect strategy had been learnt. An entirely rigorous test would have been to analyse the learnt memory of states and scores but this would have taken up significantly greater time.

The application seemed to have successfully learnt the perfect strategy after a training of 500,000 games. The strategy may have been learnt in a lesser number of games, but this was not assessed.