Machine Learning Engineer Nanodegree
Capstone Proposal

Robert M Salom
January 5, 2017

Domain Background:

Q-Learning as introduced in the SmartCab project, in the end, tackled a problem with a relatively small state space. There were primarily two main cases the agent had to handle. Making the correct choice when turning left on a green light, and making the correct choice when making a right turn on a red light. Additionally it would be relatively easy to just program the agent to do the right thing without appealing to a sophisticated method like Q-Learning. The present proposal describes a situation where, though not particularly important in its own right, at least presents an opportunity to test Q-learning in the much larger state space inherent in the game of ticktacktoe. It is also a problem where directly programing ideal behavior proved to be challenging and so Q-learning is a viable approach for a problem of this type, at least for me. My goal for the capstone initially was twofold; to use it as an opportunity to solidify the knowledge acquired in the course and to expand upon my programming skills by choosing something that would require a significant amount of coding. I did not intend it to be a venue for original research (its just ticktacktoe), I don't know if this is enough to meet Capstone project requirements any longer.

Problem Statement / Dataset and Inputs:

To implement Q-learning so that a player agent can refer to a Q function to decide where its next mark on an NxN ticktacktoe board will go.

The player's move, dictated by Q, should ideally approximate an actual optimal move. To facilitate my descriptions here I will define things in terms of their actual python code implementation.

- The "size" of the game is the length of a row of the game board so classic ticktacktoe is of size 3.
- A "mark" is the symbol an agent uses to mark the board for example the capital letters 'X' and 'O'
- A "move" is a tuple consisting of and index followed by a mark i.e, (4, 'X') or (18, 'O')
- An "index" is the position taken by a move on the ticktacktoe board. If the game is of size 3 then indices range from 0 to 8. In general, the indices range between 0 and size^2 – 1
- A move "sequence" is a list or tuple consisting of moves alternating in mark in the order they were made by the players. For example the first three moves of a game expressed as a sequence [(8 ,'X') , (6, 'O') , (2, 'X')] or alternatively ((8 ,'X') , (6, 'O') , (2, 'X')) . The first move is at the python list index of 0.
- The "Q function" is a python dictionary with keys in the set of all move sequences for a particular size of board and values in the real numbers. The idea is that a size of game is chosen and then a Q-function is populated with keys and values.

The ticktacktoe game is usually thought of as existing on a grid with 3 rows and 3 columns, although here we will think it as n rows and n columns. There are also two diagonals which will be referred to as the positive diagonal "D-pos", who's start point is at game index 0 and end point is at game index n^2 -1, and a negative diagonal "D-neg" with start index n-1 and end index n^2 – n. If any of these rows, columns, or diagonals has marks belonging to a single player it will be referred to as a "line".

- A "line" will be represented by a python list who's first entry is the mark of the player it belongs to, followed by the indices belonging to the given horizontal row, vertical column or diagonal.

For example in the game below, column 0 and row 0 form lines denoted as;
['X', 0 , 8] and ['X' , 0 , 2, 3] respectively.

X		X	X
	O		
X		O	
		X	

Row 1 is also a line denoted as ['O', 5]. The "length" of the line is the number of marks belonging to it. In the example above the line in row 0 has length 3, the line in column 0 has length 2 and the line in row 1 has length 1.

Solution Statement:

The overall idea for this project is to have two learning agents play ticktacktoe and accumulate experience iteratively in a global Q until their performance approximates that of a human player. In order to do this I plan to update Q at the termination of a game, in contrast to after every step (like in the smart cab project). When this occurs, a global number say, M, will be divided by the length of the of the game sequence yielding a reward unit, r. Every subsequence of length *L* (which is also a key in Q) will be updated by *L * r.* If the winner of the game was player1, r will be a positive reward, if it was player2, r will be negative, a tie 0 reward. Game sequences will be added dynamically to Q, that is Q will not start with the set of all game sequences as its keys. Player1 will choose its next move so that it yields a sequence of maximum value in Q, or, a random "uncharted" move according to a threshold. Player2 will do the same but seeking to minimize the value provided by Q. All of this most important part of the project has yet to be implemented and I do not know if it will work.

Benchmark Model:

The most intuitive and naive benchmark that I plan to use is to observe after the course of many games if the two agents running a Q policy would be ending games in a tie. Obviously this behavior is not necessarily an indicator of ideal play since there could be many reason why the agents would end games in a tie. So the next benchmark would be to explicitly write a strategy that is optimal at the very least in the 3x3 case, but applicable to the NxN game. After considerable effort I think I have succeeded in writing this section of the code. In this manner a Q policy could be pitted against the "ideal" policy and in turn obtain statistics on the number of wins, draws, and losses exhibited by the Q policy. Since I haven't really proven that the implementation of the ideal policy actually plays perfectly, one option is to provide statistical evidence that the ideal policy is "close" to perfect by running it against hundreds of randomly generated games (a feature that I already implemented). Since a perfect player will either win or draw in a 91:3 ratio for player1 and 44:3 ratio for player2 (wikipedia), I could provide a count of its performance and compare it to those ratios.

Evaluation Metrics:

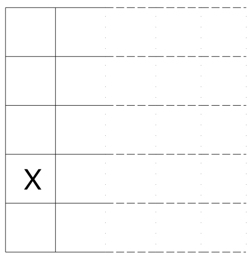
For evaluation metrics for the Q policy will mostly be centered around the win:draw:loss ratios. There will be a tally of these for Q vs Q, Q vs Random, and Q vs Ideal. However I would also have a speed measure. It would take the form of a comparison over multiple runs of the average time to game completion for, Q vs Q, Ideal vs Ideal. For larger N say 17 or so, Ideal is already proving to not be very fast on account of the calculations required to decide on a move. I would think Q should be significantly faster since it is only doing dictionary lookups. However I have yet to see how the training time will look like for Q. If Q learning is actually successful, I will also define a training time as the average time spent iteratively learning, until Q yields win:draw:loss ratios that are within a certain percentage of Ideal's win:draw:loss ratios.

Project Design:

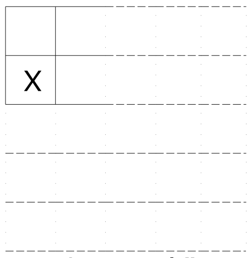
Initially I took the code for the SmartCab project as a starting point for this one, since it provided an example of how an agent class can operate inside an environment class. However I have rewritten almost all of the particulars. The code is divided into 4 python modules. A GameState class, a QMap class, and bare-bones Player class are housed in the game_state module. This takes the role of the environment module in smartcab. QMap houses the Q dictionary and methods that operate on it. At this point it just acts as a counter of visited move sequences. The play module contains the run and main methods that will run the games as well as a LearningPlayer class that extends Player, a similar layout to the smartcab. The Strateegery class (it was supposed to be funny) in the strateegery module has methods that decide player moves, as of now Random and Ideal. Finally the GameBoard class is supposed to add a graphical pygame face to the game but at this point have relegated this to lowest priority. The initial intention was to create a third HumanPlayer class so that the game could also be played interactively.
A first difficulty encountered is described in the wikipedia article, https://en.wikipedia.org/wiki/Tic-tac-toe. Here, a naive count of the number of games in 3x3 ticktacktoe is 9! or 362,880. This follows in a straight forward way from the sequence representation described in the problem statement section. So for NxN a naive count would yeild n! possible games. However, again as mentioned in that article, the symmetries of the square make many of those games equivalent. For example starting on any of the corners is strategically equivalent. In order to reduce the number of keys that will exist in Q, I implemented a coordinate transform that will use the first move made to transform the game into a "standard" position of starting moves in an upper triangular region. This is accomplished by the use of at most 3 reflections about; a vertical symmetry line, a horizontal symmetry line and a diagonal symmetry line. Take for example this starting move;

				X

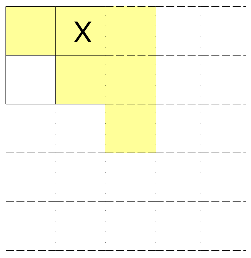
A reflection about the vertical symmetry line results in;



And about the horizontal symmetry line;



Finally about positive diagonal “D-pos”;



So that start_index = 19 is equivalent to standard_index = 1. There are sequences that are equivalent if the moves coil around the triangular region especially as N gets larger and the region grows, but still this is a big reduction in the state space and for N = 3 it gives a minimal state space.
The next big problem was the design of the ideal policy, which took considerable effort on my part, possibly because I have failed to device a smarter way of dealing with the problem. The basic idea used here, was to measure the state of the game in terms of the quantity of “lines” (as defined previously) each player owns and what their “length” is (or how filled they are). More precisely if player has made lines, *l_1, l_2, l_3 … l_k* . Each of length *m_1, m_2, m_3 … m_k.* Then the measure of the game for said player is given by;

M = *l_1 * m_1 + l_2 * m_2 + l_3 * m_3 … + l_k * m_k*

The next move for a given player is either the one that will result in the greatest M for himself, or, in the case that there is a move that gives his opponent a greater measure than what he would gain for himself with his best move, to block it by taking his opponent’s move of highest measure. After a certain point, more specifically when the game has reached n + (n-1) – 3 + n+(n-1) – 3 – 1 = 4n -9 steps, which is the first time a player can generate a fork. The ideal policy will; 1) look if there is a fork possible by *either* player in the next move, 2)If its the current player’s fork, then make it, if its the opponent’s then block it. Ideal policy will now also look for lines of size n-1 which are one move away from a win. If it they belong to the current player then win, if they are the opponent’s then block the win. After submitting this proposal I will upload my current progress to github so that the reader of this proposal can evaluate that progress: https://github.com/bb-blud/tictac . Thanks!