Machine Learning Engineer Nanodegree
Capstone Project

Robert M Salom
January 28, 2017

Project Overview:

The overall goal of this project is to explore reinforcement learning as applied to the "simple" game of tictactoe. Reinforcement learning (RL) is a big area of study which by its very nature as being somewhat domain agnostic, has been applied in many, even seemingly unrelated fields. These range from quadruped gait control and humanoid air hockey playing in robotics, to stock market trading (same link), to helicopter control (same link). More recently, in the subfield of RL known as Q-learning and on a subject closer to that of this project, we see that RL can be used to play Atari 2600 games at human expert levels (paper https://arxiv.org/pdf/1312.5602v1.pdf) A cursory look at that material however reveals the use of more advanced neural network based techniques on complex, even continuous state spaces. The intention here is to use more elementary concepts to explore and gain some understanding of the basic ideas behind RL. Additionally, there are many implementations of Q-learning applied to tictactoe found on-line, similar to the one developed here, for example, 1,2,3, and 4. The Nanodegree lectures introduced Q-learning as a sub-domain of reinforcement learning which culminated in the SmartCab project. In the end, Smartcab 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. With tictactoe, which we will consider here in a more general nxn version where n is the length of a row, there is the opportunity to apply reinforcement learning on larger more complicated state space; that is, the set of all possible game sequences. It is also a problem where directly programing ideal behavior proved to be challenging, at least for me, and so Q-learning is a viable alternative approach for a problem of this type. On this point however, tictactoe can be solved with explicitly programmed routines. As pointed out by the proposal reviewer, the general purpose *minimax with alpha beta pruning* algorithm can play tictactoe perfectly at the expense of a lot of computational power. Two methods for explicitly playing tictactoe are are also explored and used as performance references for the reinforcement learning guided play; the previously mentioned alpha-beta algorithm and a naive solution that turned out to be really fast, with some of the same minimax spirit.

Problem Statement:

The initial hope is to have two learning agents play tictactoe and accumulate experience iteratively in a global Q until their performance approximates that of a human player. That is;

To implement reinforcement learning so that a player agents can refer to a Q function to decide where their next mark on an NxN tictactoe board will go.

That being the main problem, a secondary problem that is important to me is to avoid as much as possible, even completely, the use of the explicitly programed perfect players as

reinforcement devices. My curiosity is to see how well the agents can learn purely on their own with just the game's end state as feedback. I feel this will also give me insight on how to deal with a situation where I have no clear explicit programmed solution. The player's move, dictated by Q, should ideally approximate an actual optimal move. In order to accomplish this, a global Q function, which is a python dictionary, will be updated at the conclusion of every game. This is in contrast to after every step, as in the smart cab project. The keys of the dictionary, that is the state space, are in the set of all possible game sequences. Since ideally the procedure should work for any size of board, the dictionary will start empty and keys will be populated dynamically as new sequences are explored. Now, since the intention is for Q to be used by a player agent to guide its next move, a reward, say r, is added to every subsequence of the end-game sequence that concluded a game. The sign of r depends on which player won the game. Positive for player 1 negative for player 2, and r = 0 in case of a tie. Since those subsequences (including the entire game sequence) are keys in Q, each explored sequence will have a real number value that will be used to guide subsequent player moves. Player 1 will seek actions that yield the highest value of Q in a given state of the game, and player 2 will seek actions that yield the smallest value of Q in a given state. The solution should take the form said dictionary, which when queried for max/min moves alternatingly by two player agents approximates optimal play.

Metrics:

As a means of gauging performance the two tictactoe playing agents can be set to play according to four main playing policies. *Random*, *minimax* , *ideal,* and *reinforcement*. Performance is measured within the context of dueling policies. For the 3x3 case an explicit tally of all strategically relevant final game positions (https://en.wikipedia.org/wiki/Tic-tac-toe) will also be used. In other words a count of game positions that excludes geometrically equivalent final games. There are 91 ways player 1 can win, 44 ways player 2 can win, and 3 draws. Here a perfect win:draw:loss is readily available for comparison. For games of higher dimension, in the absence of explicit game position counts, the idea is to appeal to metrics of a more statistical nature. First of all, by playing Random vs Random for a large number of games, a rough (the reason why I say rough will be discussed later) baseline of the proportions of win/draws/loss will begin to appear. Armed with that and with the fact that a necessary condition for a perfect player is that it does not lose, one can at least have a gauge of whether a policy is essentially random, near perfect, or perfect. For example in a 3x3 game for a perfect player 1, it's win/draw/loss count should be approximating 91 : 3 : 0, and 44 : 3 : 0 for player 2. So for games larger than 3 after using Random vs Random to get an idea of the games global win : draw : loss, any reduction in the proportion of loss and subsequent allocation to win : draw is a measure of improvement in performance. As it turned out a lot of this reasoning was somewhat upended, but I still have to describe the metrics in this section.

Analysis:

Input space:

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 tictactoe 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 an index followed by a mark i.e, (4, 'X') or (18, 'O')
- An "index" is the position taken by a move on the tictactoe 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.
  Ex: Indices of a 4x4 game board:

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

- Alternatively it is also convenient to refer to game positions or cells in the board by their x,y coordinates, where the uppermost left cell corresponds to 0,0. In this way for a given index, u, its coordinates can be calculated as x = u%(size) and y = u // (size). Where % is the modulus operator and // is python's floor division operator (// and / do the same thing in python2).
- A move "sequence" is a list or tuples 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 could be something like [(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. Sequences in tuple form are the actual keys to the Q function.
- 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 tictactoe 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.

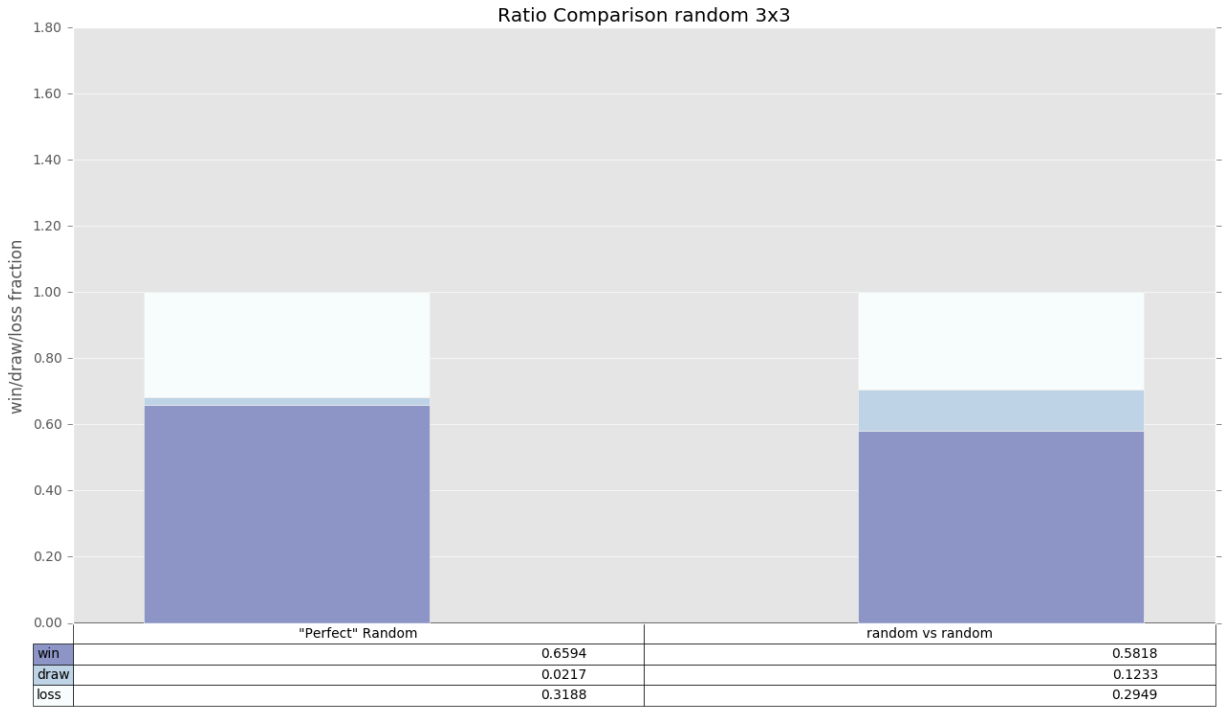
For example in the game below, column 0 and row 0 form lines denoted as;
['X', 0 , 8] and ['X' , 0 , 2, 3] respectively.

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.

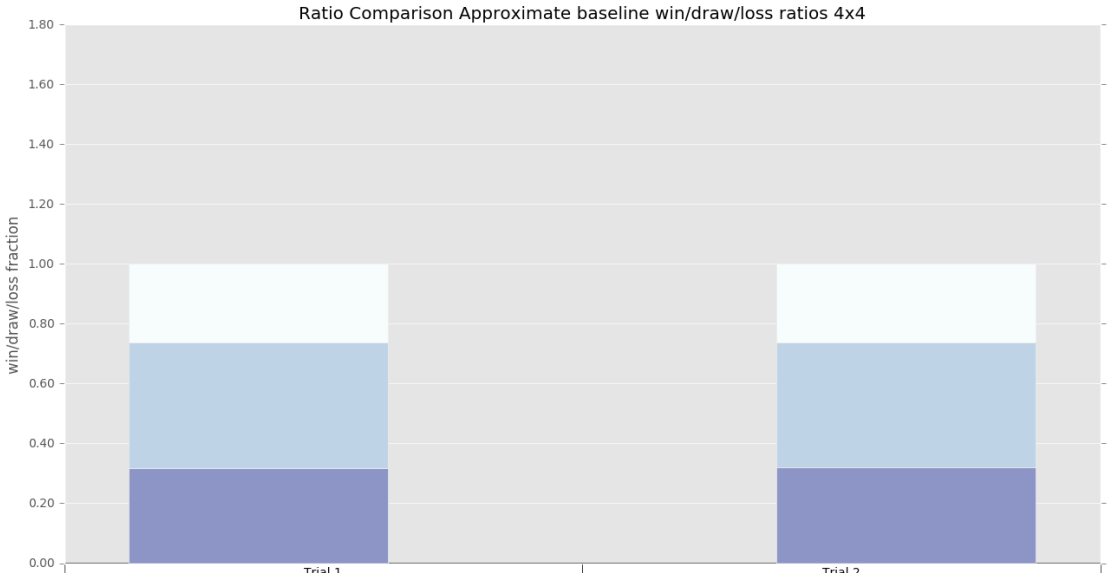
Data exploration/visualization:

To get an idea of what the space of end game sequences looks like under the above mentioned metric I started by comparing the ratios of actual game counts for a 3x3 game with the counts of 10000 games with agents choosing their next move randomly (but legally) using python's random.randint() function. That is, the proportions 91/138 : 3/138 : 44/138 were compared to (total # player 1 wins)/10000 : (total # draws)10000 : (total # player 2 wins)/10000. With the results seen on the figure below.

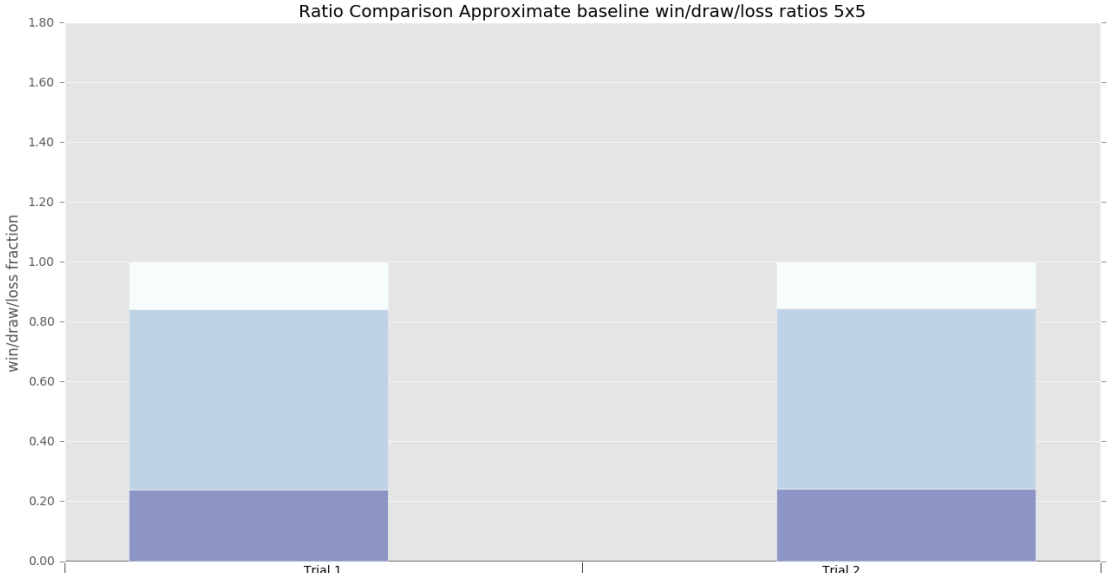


	"Perfect" Random	random vs random
win	0.6594	0.5818
draw	0.0217	0.1233
loss	0.3188	0.2949

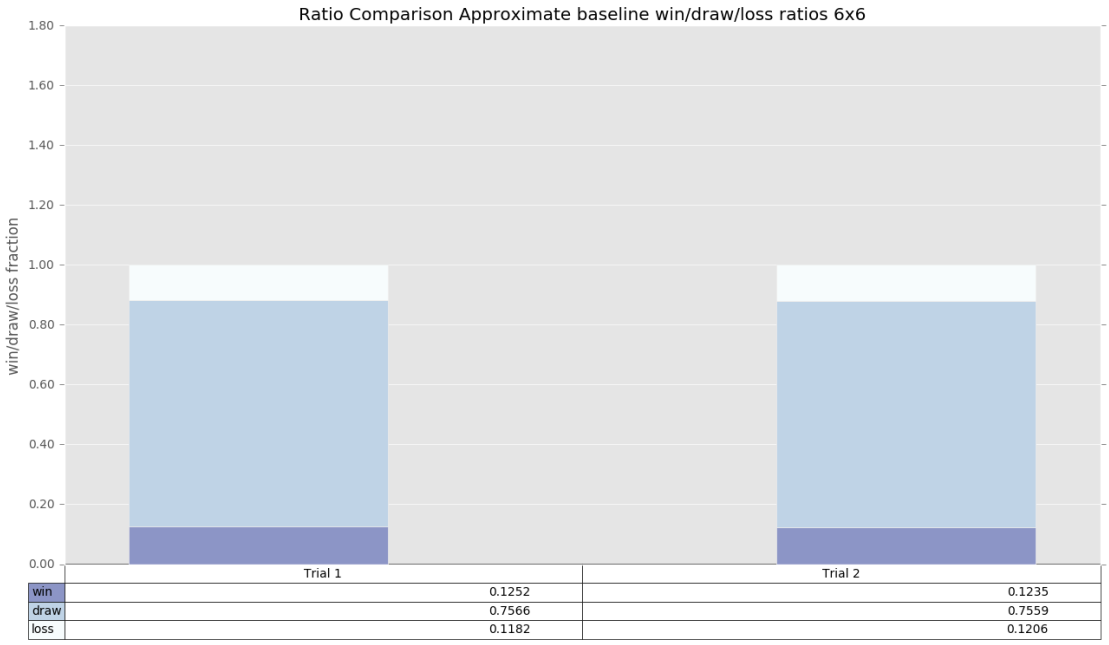
So the main thing to notice here is that the ratios acquired through simulation have an enlarged draw proportion. The simulated win proportion is ~12% smaller than that of its counted counter part and the loss proportion is ~7% smaller. The draw proportion is ~ 6 times bigger. So it jumps from being 2% of the total to around 14%. So a random policy does have a tendency towards draw. The above described metric is somewhat rough, but very practical, especially when used to measure increases in wins and reduction in losses, since two regions are not that far from the actually counted ratios. Draw's error is larger because it has been bled into by both win and loss. Confidence in this reasoning is somewhat shaken when the following trend is observed when the same process is applied to size 4, 5 and 6 games with figures below.



	Trial 1	Trial 2
win	0.3169	0.3195
draw	0.4205	0.4180
loss	0.2626	0.2625



	Trial 1	Trial 2
win	0.2380	0.2406
draw	0.6045	0.6032
loss	0.1575	0.1562



	Trial 1	Trial 2
win	0.1252	0.1235
draw	0.7566	0.7559
loss	0.1182	0.1206

In the above 3 figures two trials of 1000 random vs random games were performed for games of sizes 4,5,6. It is clear that an increase in the size of the game is leading to growth of the proportion of draws. So is this because of the simulated random gameplay's tendency towards draws or is it a real thing? That is, are these "sampled" ratios any good as rough benchmarks for nxn games? I think it is an actual trend, which shows that for larger n, a game is more likely to end in draw. Alternatively as n increases there are more ways to draw a game than to win it. The reason I think this is true, is because in order to win, one must fill a row, a column, or a diagonal. For arbitrary n, there will always be n rows, n columns, plus two diagonals. So 2n+2 "shots" at winning the game. To eliminate any one of these, you only need two opposing marks to be placed in line. For a given line, there are n places to place the first mark, and n-1 places to place the second, and then roughly times 2 because either player could have made the first mark (except for the very start move which belongs to player 1) so ~2n*(n-1) ways to "kill" a line. Though I am ignoring the count of ways you can get to a win or draw position (which I don't think would favor the wins), roughly there are 2n+2 winning lines and at least 2n*(n-1) * (2n+2) ways to kill them. So it looks like winning grows somewhat linearly and drawing grows cubicly. For all that follows however I will be restricting analysis to mostly the case of size 3 and 4, basically because of computational constraints.

Algorithms and Techniques:

Apart from the above sampled win/draw/loss ratios as benchmarks for comparison, two explicit algorithms have been implemented that are at least close to being perfect players. The first, Ideal (which was a somewhat tortuous reinvention of the wheel on my part) basically seeks to generalize roughly the common sense game play we use as humans on a 3x3 board to the nxn case. The 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). The intuition being that the more places a player agent has to fill a line the more likely it is to win. Take the following game;

X			X
	O		
X	O		O

Even though both player's are technically at minimum two moves away from being able to win, one can argue X is in a more advantageous position. X basically "owns" two lines (of length 2) at this state in the game, the first horizontal (or horizontal of y coordinate 0) and the vertical of x coordinate 0. Here I refer to a line by the common coordinate shared by all the marks belonging to it if we were to assign x,y coordinates to every cell. O also owns two lines, the vertical line of x coordinate 1 (length 2) and the horizontal of y coordinate 1, with just one mark. So for X, the measure of the game state above is 2 + 2 = 4, the length of vertical 0 plus the length of horizontal 0. For O, the measure of the game state would be 2 + 1 = 3, corresponding to vertical 1 and horizontal 1. In general then, if a player owns lines, $l\_1, l\_2, l\_3 \ldots l\_k$. Each of length $m\_1, m\_2, m\_3 \ldots m\_k.$ Then the measure of the game for said player is given by;

$M = m\_1 + m\_2 + m\_3 \ldots + m\_k$

The next move made by an Ideal agent follows the natural priority we would usually use as humans in a 3x3 game. First and most importantly;

Win/Block:

- If the player can win by making a line of length *size,* take it
- If the opponent will win on the next move by the same means, block it.

The next priority level pertains to forks. After a certain point, more specifically when the game has reached n + (n-1) – 3 + n+(n-1) – 3 – 1 = 4n -9 steps, it will be the first time a player can generate a fork. The reasoning for the above count goes as follows; two intersecting full lines have n+(n-1) marks on them (because they share a mark). To make them into a fork, take two marks (not the intersection though) away from them. Thats a fork. And in order to make them into a "pre-fork", or the state before a fork can be made, take one more away; hence n + (n-1) -3 = 2n-4.

Ex:

4x4: Two, X, "intersecting" lines (impossible as an actual game state), they contain 7 = 4 + 3 = 4 + (4 -1) marks;

X	X	X	X
X	O		
X		O	
X			O

Minus two marks we have a fork, with 4 + (4 -1) -2 marks;

X	X		X
	O		
X		O	
X			O

Minus one more mark, a pre-fork with 4 + (4 -1) – 2 – 1 = 2*4 – 4 marks;

	X		X
	O		
X		O	
X			O

Using the example illustrated above we also see that since X has made 4 marks at the pre-fork, it necessarily means that O has made 4-1 = 3 marks. In general if player 1 has made m moves, it means player 2 has made m-1 moves. So in general then, the number of steps taken at the first "pre-fork" is player 1's moves, 2n-4, plus player 2's moves 2n-4 -1, which is;

2n-4 + 2n-4 -1= 4n-9

At this point the ideal policy will;
Fork/Block:

- Look if there is a fork possible by *either* player in the next move
- If its the current player's fork, then make it
- If its the opponent's then block it.

If neither of the above scenarios are in play, then the third priority is to move by measure;

Measure State:

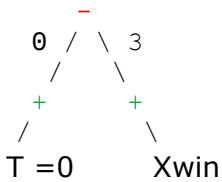
- The next move for a given player is either the one that will result in the greatest measure, M, for himself, or, in the case that there is a move that gives his opponent a greater measure than what he would gain for himself, to block it by taking his opponent's move of highest measure.
  So for example suppose it is O's turn and out of all its legal next moves there is one with largest measure of lines owned by O, say a move m that yields M(O). Now, Ideal will also consider out of all those legal moves for O, which one would yield the largest measure in terms of X lines, M(X), *even though X cannot make a move in this turn*. If M(O)>= M(X) then O will make the move that yielded M(O), but if M(X)>M(O) then O will take the position that *prevents X* from ever acquiring M(X) (it is possible they are the same position).

The whole intuition behind Ideal is that basically if the board is "filled up enough" an nxn game is pretty much the same as a 3x3 game, you try to fork and you try to fill a line entirely. But in order to have good chances to make the forks or the full lines you have to try to gain as much territory as possible and prevent your opponent from doing the same.

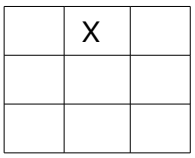
The second explicitly programmed strategy is the minimax alpha-beta pruning algorithm. The source material for the the implementation used in this project comes entirely from https://www.cs.cornell.edu/courses/cs312/2002sp/lectures/rec21.htm, which seem to be lecture notes for a Cornell CS class. As I only understand in a novice way, it is probably best to refer to those notes for a more thorough explanation. Nonetheless, minimax treats a game as a tree of possible future states. The root of the tree, or root node, is the game's current state. From there, every possible move a player could make is called a child of the root node. All the possible moves the opponent could subsequently make are represented by child nodes of the previous children. The process is repeated until the tree structure ends at points called leaves; basically in our context, corresponding to moves that lead to a win, draw, or a loss. Paths from the root to any node in the tree basically correspond in our case to what we have referring to as "subsequences". So in the case of this 3x3 game state (where it is O's turn);

	X	
X	O	X
X	O	O

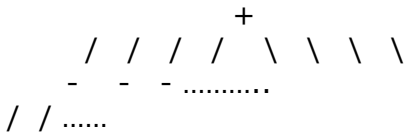
The entire remaining game can be represented by this tree;



Where the two branches represent O choices of index 0 or 3. The leaves correspond to X's subsequent only remaining choices to either tie or win respectively. Minimax with alpha beta pruning, is a recursive algorithm that explores future moves up to a specified depth and assumes that each node can be evaluated or "measured" in some way. In our context player 1, X, seeks to make a next move of value as large as possible, and player 2, 'O', seeks to make a next move of value as small as possible. In the example above it is not obvious that we need to have a value for nodes that are not leaves because we are just two moves away from finishing the game. Imagine however it is the first move of the game;

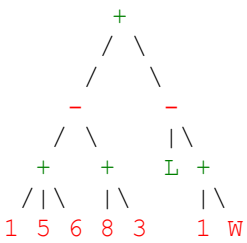


In order to generate the full tree we would have 8 possible children from this state, corresponding to O's possible moves, and for *each* of those, there would be 7 children corresponding to X's second move;

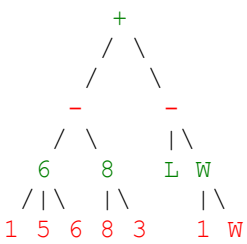


Although it will not likely grow as full as 8!, because termination states will be reached before then, it would still be completely impractical to attempt to search the tree all the way down to its leaves. Instead the tree is searched only to a certain depth, which in general will not be deep enough to see a leaf. Hence the need for what the referenced article above calls a *static evaluator*. It is basically a measure of the game at a particular node, i.e. game state. Essentially we have already encountered an example of this with M, the measure used in Ideal.

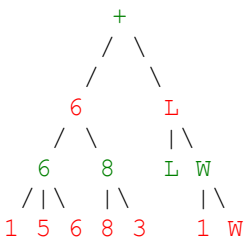
So lets walk through the mechanics of Minimax, say, with a tree searched down to a depth of 3. Here L represents a losing state for the current player (root) and W a win. They can be thought of as -infinity and infinity, respectively the most desirable states for Player2 and Player1. The numbers are the measures provided by the evaluator for those nodes.



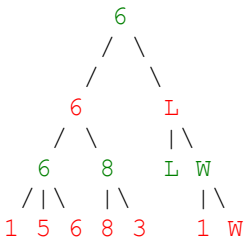
At the full depth of three Minimax is at a max layer (Player 1's possible moves) so it will take the maximums;



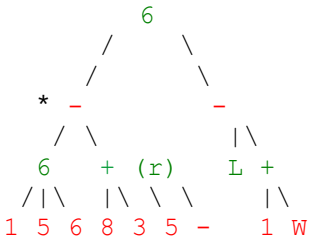
Moving now to depth 2, we are at a min layer (Player 2) so minimums are taken;



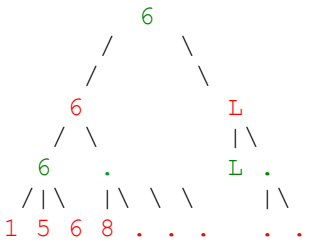
Finally the *minimax* value of the root (i.e. the games current state) is in this case the max of 6 and L; 6.



When the tree is evaluated with minimax as we have above, it automatically provides the player at the root (the current player) which move it should take. In this case the left move that leads to a value of 6 instead of -infinity or a loss L. Two see the "pruning" aspect of the algorithm, lets modify the above tree slightly by adding some child nodes;

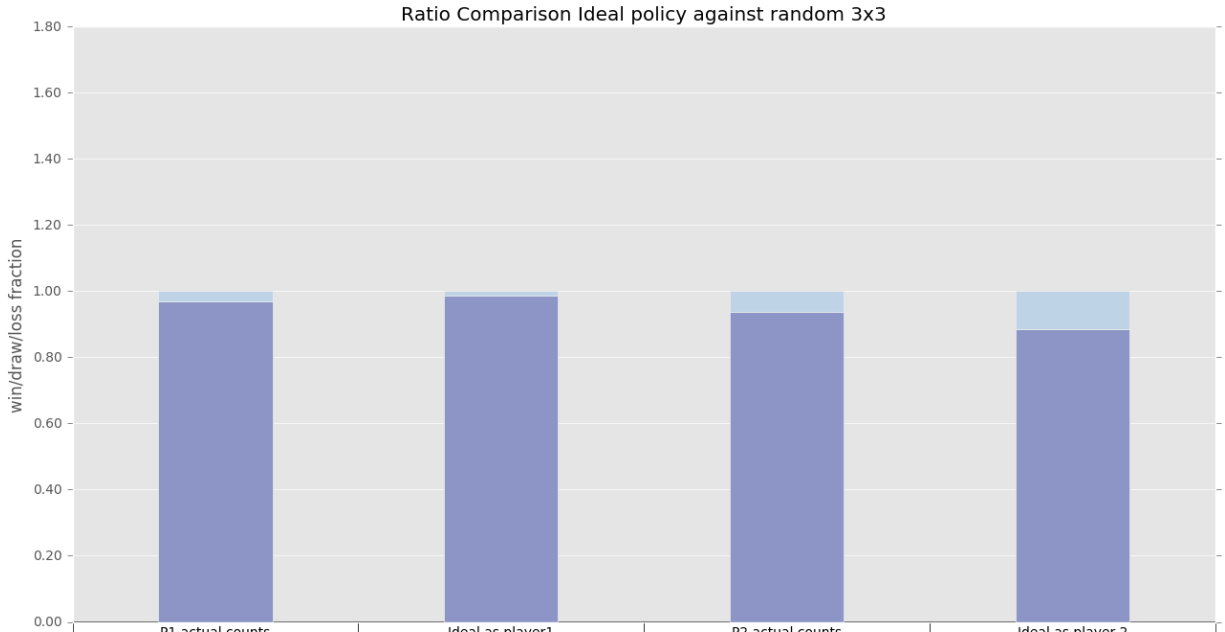


Suppose the tree is searched from left to right. Suppose we are processing the node with an *, and at this moment its left child has already been evaluated by minimax to 6 (just as before), now as we go to process *'s right child, labeled r, we are looking to find the max of [8,3,5, -] where – is a child node yet to be processed (deeper than depth 3). However as soon as we encounter the value 8, we know that max will be at least 8, already greater than *'s left child evaluated at 6. So there is no reason to process r any longer because we were going to return the smaller of 6 and, whatever r's eventual minimax value would be. So all unnecessary remaining calculations are "pruned", and the evaluation tree looks more like this, with some pruning occurring for root's left branch as well,



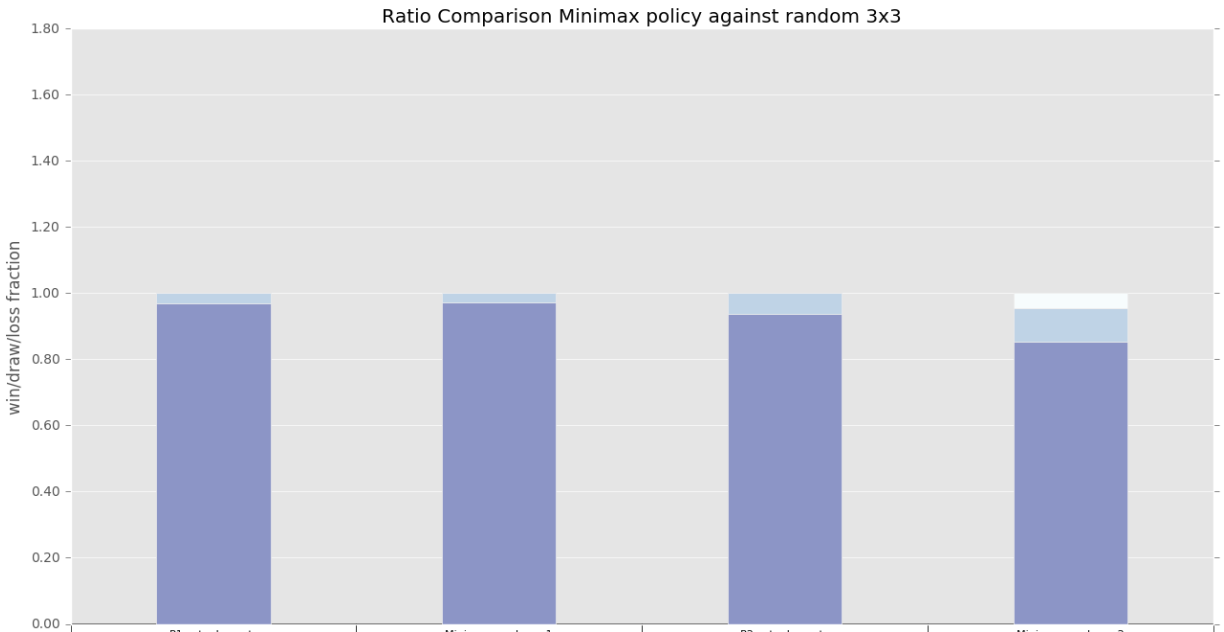
Minimax is a general purpose algorithm, so in order to use it in our context we basically have to give an evaluate function specific to our tictactoe game. Luckily it is just a matter of refurbishing the Ideal's measure function to provide minimax with its "evaluate" function. Basically, at each "node" (i.e. each game sequence) M1 (as defined for Ideal) is taken for player 1, M2 is taken for player2 and the minimaxMeasure is defined as M1 – M2. That way if a node heavily favors player 2 it will tend towards negative values, positive values for player 1, and if the state is exactly balanced then M1=M2 and minimaxMeasure = 0. If the sequence corresponds to a player 1 win a much bigger reward of (10*size)^4 is assigned, or the negative of that for a player 2 win. 0 is the measure for a tie. Interestingly that was my first attempt at the measure function for Ideal but I couldn't get it to work. It seems to work reasonably well with minimax.

The Ideal policy, because of its aggressive blocking *should* never lose and, after much debugging, the data bare that out. It possess at least that property of a perfect player. What follows is a comparison of Ideal and minimax with "perfect" player 1 and player 2 ratios of 91 : 3 : 0 and 44: 3 :0 on a 3x3 board. First;



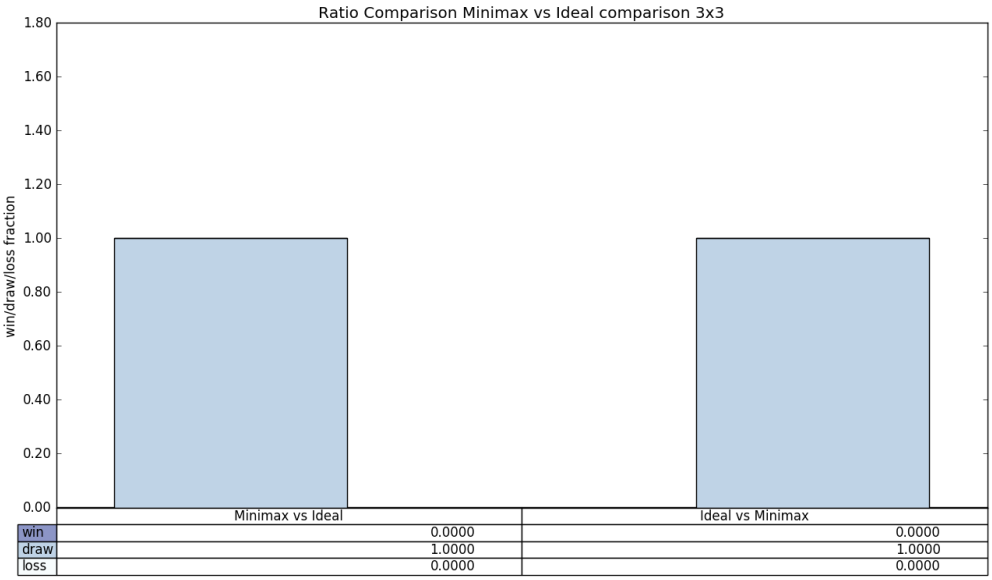
	P1 actual counts	Ideal as player1	P2 actual counts	Ideal as player 2
win	0.9681	0.9880	0.9362	0.8840
draw	0.0319	0.0120	0.0638	0.1160
loss	0.0000	0.0000	0.0000	0.0000

For the figure above we see that Ideal deviates somewhat from what we expect a perfect player to do. It tends to win more as player 1, ~2% increase and draw more as player 2 ~6% increase. Importantly, however out of 1000 trials there are never loses, up to size 13 or so (reducing the trial number) that is always the observation. Similar analysis of minimax is not as clean because due probably to inexpert implementation (i.e. not the best evaluate function) it is slow and hence most of the time it is used with a search depth of only 2. Which should be enough to spot most forks, however it leaves it open to a small number of loses when playing the disadvantaged player 2 position. As can be seen here;



	P1 actual counts	Minimax as player 1	P2 actual counts	Minimax as player 2
win	0.9681	0.9720	0.9362	0.8520
draw	0.0319	0.0280	0.0638	0.1020
loss	0.0000	0.0000	0.0000	0.0460

Remarkably minimax's ratio as player 1 is virtually indistinguishable from the "perfect" player. Additionally, after tracing back several of the specific games it lost as player 2 and setting higher search depths it would reverse the loss to a win many times, and at least draw. I feel a better implementation of minimax or more computational power would reveal ratios exactly matching the ones derived from an explicit game count. Minimax's graph above is a result of 500 trials. To further confirm Minimax's and Ideal's near perfect behavior here are three trials of Minimax vs Ideal. Minimax is operating with a depth setting of 5 to dispense any doubts of under performance. In reality only one trial is necessary as, when traced, they play the same game every time because they are pursuing optimal strategies.



As expected, they will always draw, regardless of being player 1 or 2. Finally a detailed explanation of the approach to reinforcement learning used here is overdue. I hesitate whether to call it Q-learning or not sometimes, because since the context is not that similar to what we saw in the Smartcab, the approach here differs enough that I am not sure if it still falls under that label or instead under some other type of reinforcement learning.

Methodology:

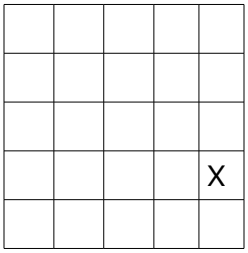
Code structure:

Before describing the details of the reinforcement learning implementation, I will briefly discuss the over all structure of the code base. It is divided into 4 python modules. The first, *game_state*, houses a GameState class, a QMap class, and bare-bones Player class. QMap houses the Q dictionary and methods that operate on it. Then there is *play* module containing the main method, along with multiple support functions to play and analyze games between player agents. It also contains the DecisionPlayer class that extends Player, a similar layout to the smartcab. Third there is *strateegery* module, housing the Strateegery class who's methods decide player moves. Three have already been discussed, *Ideal*, *minimax,* and *random*. Finally the GameBoard class is supposed to add a graphical pygame face and user interactivity but it remains unfinished. To run a game between two player agents usually a GameState instance is created of a certain size, a QMap object is created and set to the GameState, followed by two DecisionPlayer agent objects who themselves take the GameState object as a parameter in their initialization. So that in the end a player has access to the GameState, and the GameState has access to the players.

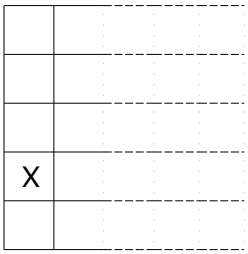
Data Preprocessing:

A first difficulty encountered when exploring possibilities for the application of reinforcement learning 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 tictactoe is 9! or 362,880. This follows in a

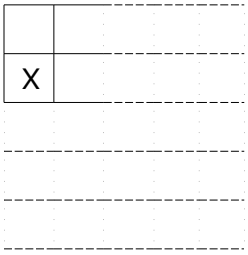
straight forward way from the sequence representation described in the problem statement section. So for a nxn game a naive count would yeild n! possible games. However, again as mentioned in the article, the symmetries of the square make many of those games equivalent. For example starting on any of the corners is strategically equivalent. In order to reduce the number of keys that will exist in Q, I implemented a coordinate transform that uses 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;



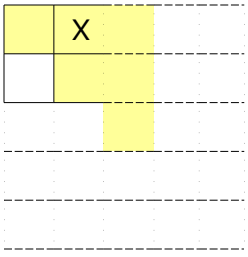
A reflection about the vertical symmetry line results in;



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.

Reinforcement learning:

It is in this context, that the global Q function, a python dictionary, will be updated at a game's conclusion. The keys of the dictionary are in the set of all possible game sequences that have started from the above described "standard" position and as mentioned before will be populated dynamically as new sequences are explored. Now, I experimented with different ways of performing the update, as well as different magnitudes for the reward, r. Regardless, if player 1 is the winner at the end of the game the sign of *r* is positive, if player 2 is the winner the sign of *r* is negative. If the game ended in a tie, essentially no update is performed because *r* is 0.
For now I have settled for r=1 for a player 1 win and r = -1 for a player 2 win. Q-learning as seen in class was, seeking to implement a version of the Bellman equation, by approximating it with a recursive function of the form;

Q[previous_action_state] = reward + λ * Q[current_action_state]

where Q[current_action_state] was actually a stand in for the actual function we were trying to approximate, that is, the *discounted* sum of Q[action, state] for *all* possible future action states after previous_action_state. We would then look for the action that would maximize that Q. Now, the Smartcab in that context, at least for me, is significantly different to the tictactoe scenario; however, I feel the update of Q in this project can be made somewhat akin to Q learning as seen in Smartcab. Since the update occurs at the end of a game, from a step by step perspective of *that particular* end game sequence, its as if we know "all future action states". So I feel like we can also approximate the theoretical "bellman" Q, end-game sequence by end-game sequence, as they are encountered. So the first iteration of that idea, looked something like this; Since a subsequence in our context is a list of moves, for example;

[(0, 'X'), (3, 'O'), (6, 'X') ….]

It kind of already a captures both the action and the state of the environment. The "action" is simply the last move on the list, and the state is the entire list. So suppose we have an end game sequence, call it EG, take a subsequence of it call it, well.. subsequence, since r is fixed by how a particular game ended, r = -1, 0, 1, then Q-learning update can take this form

Q[subsequence] += r + λ * (r + λ * (r + λ * (r + ….

where that sum is repeated however many "future states" or *moves* remain from subsequence to EG. In other words we have the simple geometric series;

1) Q[subsequence] += r + λ*r + λ^2 *r + λ^3 * r ….+ λ^n * r

Here, *n* is basically the difference between len(EG) and len(subsequence). For example for;

EG = (8, 'X') , (6, 'O') , (2, 'X'), (4, 'O'), (5, 'X')

suppose we were updating subsequence (8, 'X') , (6, 'O') , (2, 'X'), then

Q[(8, ’X’) , (6, ‘O’) , (2, ‘X’)] += r + λ*r + λ^2 *r

where n = len(EG) – len([(8, ’X’) , (6, ‘O’) , (2, ‘X’)]) = 5 – 3 = 2

So that something like the *discounted sum* of future states can be put into place. On the previous iteration of this report I had ignored the second part of the Q-learning update implementation as had been done in Smartcab. Mainly the *temporal difference* part of the update. What this adds is a gradation to the reward update of equation 1) above. Instead of updating by the full value of 1), only a fraction of it according to a parameter $0<\alpha<1$ is added, with the remaining portion coming from the current value. More concretely, if V is the current value of Q for some s,

V = Q[s]

and X is the discounted reward produced in 1),

X = r + λ *r + λ^2 *r + ….

Then the actual update of Q[s] will be

Q[s] = (1-α) * V + α * X

When agents refer to Q for there next action, player 1 would choose the next action (i.e, action_state == subsequence == key) that has maximum Q value, and player 2 would choose the one with smallest Q value. Also keys are added dynamically to Q as they are encountered, so that at any given state there could be basically “unexplored” future action_states. In that case the agent randomly chooses one according to an arbitrary threshold.
To train Q the approach taken was to basically have a combination of;

- Purely random agents play against each other. In this way populating Q with keys and values, somewhat “uniformly”, since that is the distribution that random.randInt() draws from.
- Purely random agent vs Q-learning agent, alternating between player 1 and player 2.
- After this, two Q-learning agents play against each other

Here the intention is to have Q shape itself to mimic actual game play, hopefully adding value to better moves and not adding value to bad moves (by either player). Additionally, I found that even if all subsequences at a given point had been already visited, during training having the agent still choose randomly helped dislodge Q from something like a “sink”, where, randomly, some game sequences had been over favored leading to narrow behavior.
Let see some of the issues encountered after application of the above framework. Early on after the training process described previously; more specifically, after 70 random vs random 3x3 games, followed by 1000 random vs Qlearning games, and then followed by 1000 Qlearning vs random games, a Q was produced nicknamed lucky_Q (partly because of some difficulties I have yet to discuss). To test its behavior I pit it against the perfect or near perfect players Ideal and minimax. With the following results;

	P1 win	draw	P1 loss
ideal vs lucky_Q	1	0	0
lucky_Q vs ideal	0	1	0
minimax vs lucky_Q	0.86	0.14	0
lucky_Q vs minimax	0	1	0
lucky_Q vs lucky_Q	0	1	0
random vs random	0.51	0.16	0.33

Although it is obliterated by both policies when playing as player 2, similar to how random would (although it shows some resistance to depth 2 minimax i.e, draw = 0.14); amazingly, it absolutely ties 100% of the time against the perfect players when it is in player 1 position! It also ties against itself. That's behavior, in a partial sense, only exhibited by the perfect players. How does it perform against a random agent? Here are the results;

	P1 win	draw	P1 loss
random vs lucky_Q	0.536	0.133	0.331
lucky_Q vs random	0.725	0.122	0.153
random vs random	0.591	0.12	0.289

Bleakly, we see it is virtually indistinguishable from a player simply making random moves as player 2. However as player 1 the win fraction has grown to 0.725. The draw portion is almost exactly the same as random vs random. Meaning win fraction growth is due to conversion of losses to wins. So what is going on here? The playGames function in the play module has a rudimentary convergence test, which basically keeps a buffer of the last 8 games played. If after iterative play between two agents, all previous 8 games are the same, then it tells me that their play has converged to a specific game. This is exactly what lucky_Q does to the perfect players (and itself) it just so happens to have acquired values in Q that elicit a sequence of optimal moves by both the perfect agent and lucky that converge to a tie. It is always the *same exact tie* however. This rigidity of the Q-learning strategy explains partially why it can't adapt itself to beat mostly suboptimal moves dished out by a random agent.

Refinement/New Strategy:

Initially I was having much trouble trying to reproduce Lucky_Q since I had stumbled upon it at an earlier iteration of the project that had a more bare bones version of the Q update. After exploring the matter I found that the main reason for the difficulty was having added an occasional random move to the Q-learning policy during training. After employing something like a grid search, with Q update alpha-gamma values of; [0.1, 0.26, 0.42, 0.58, 0.74, 0.9], I found that the properties of lucky_Q can be reproduced consistently with an alpha = 0.1 and gamma = 0.1 (first iteration, score!). I also used a training regiment like the one described before. See the accompanying IPython notebook for more details.
There are two main issues with this; first, seeking to improve the Q-learning by this methodology is somewhat like cheating, akin to using a test set instead of a validation set for the tweaking of parameters . In a scenario where I don't happen to have these perfectly playing policies I wouldn't be able to easily know if I have arrived at Q that is capable of

producing perfect play ties. But secondly, and more importantly, the "rigidity" of Q-learning just observed is not really overcome by tweaking alpha and gamma. Similar grid searches of these two parameters hunting specifically for non-random play against a random opponent did not improve the ratios seen in the table above. For example trying the 36 possible choices of alpha and gamma resulting from the list above resulted in the following average ratios for Q-learning as player 2;

0.54527778 : 0.12916667 : 0.32555556

with standard deviations respectively of;

0.05742077, 0.0349106, 0.04985782

The maximum fraction of wins observed in the 36 passes for Q-learning as player 2 was just 0.21. With the Q-learning implementation as is, the alpha and gamma parameters don't seem to be capable of providing much improvement in performance.
This led me to attempt a blend between Q-learning and minimax (which I think the reviewer of the proposal for this project was trying to point me too, and I had failed to understand). Minimax seems capable of perfect play at sufficient depth, but my implementation of it is slow, most likely because of the evaluate function, minimaxMeasure. Q-learning is faster, as it is based simply on calls to a dictionary, but is rigid and unadaptable. The idea was to replace minimax's bottleneck, minimaxMeasure, with calls to Q. In that way improve speed and use Q in an adaptive way. In the miniQmax strategy then, Q replaces minimaxMeasure as the evaluate function. Implementing exactly that results in the following figures;

Table 1	P1 win	draw	P1 loss
random vs Qlearning	0.61	0.11	0.28
Qlearning vs random	0.77	0.18	0.05
random vs miniQmax	0.32	0.12	0.56
miniQmax vs random	0.69	0.09	0.22
Qlearning vs miniQmax	1	0	0
miniQmax vs Qlearning	0.33	0.37	0.3
Qlearning vs Qlearning	0	1	0
random vs random	0.602	0.106	0.292
Qlearning vs minimax	0	1	0
minimax vs Qlearning	1	0	0
miniQmax vs minimax	0	0	1
minimax vs miniQmax	1	0	0

NOTE: All presented data from this point forth including Table 1 display win:draw:loss ratios generated from 100 trials or duels between two agents. Also as a note on all tables and figures provided, the duels between agents many times converge to a single game. Usually that is the case with the whole number 1's and 0's reported. Effectively it is as if 100 games were played, if they would the result would be always the *exact same game* for 100 games.

The first two rows are the now familiar ratios seen with lucky_Q. The next two rows are miniQmax as compared to random. It performs slightly better than a random player as player 1 with a 0.69 win fraction. It also demonstrates particular strength against random as player 2, increasing random's loss (miniQmax's win) to 0.56. The next two rows are Q-learning pitted against the new miniQmax policy. MiniQmax is set to a search depth of 3 without as much performance hit as minimax, however it performs badly against simple Q-learning,

completely dominated 100% of the time as player 2 and having close to a uniform distribution of win:draw:loss as player 1. MiniQmax doesn't stand a chance against minimax, losing 100% of the time as player 1 and 2. Where Q-learning is capable of at least tying minimax, miniQmax cannot manage at all. Also to keep in mind is that miniQmax is sharing the Q map with Q-learning. What the data seem to show is that miniQmax is much weaker than the other "smart" agents, Q-learning and minimax, yet it exhibits non-random play against random. A natural supposition would be that the strong Q generated for Q-learning should pass on its strength to miniQmax, is this the case? Here we have ratios of 100 duels between Qlearning using lucky_Q vs miniQmax, using lucky_Q and then using a very skimpily trained random vs random of just 70 games, *lazy Q.* Also miniQmax is still operating at a depth of 3.

Table 2	P1 win	draw	P1 loss
lucky_Qlearn v lucky_Qmax	1	0	0
lucky_Qmax v lucky_Qlearn	0.41	0.37	0.22
luckyQ_learn v lazy_Qmax	1	0	0
lazy_Qmax v lucky_Qlearn	0.75	0.05	0.2

We see that lucky Q-learning dominates miniQmax as player 1 regardless whether it uses lucky or lazy 100% of the time. For miniQmax as player 1 using luckyQ significantly reduces performance. The ratios in row three are close to what was previously observed in table 1 above where luckyQ was also used. We see that miniQmax without luckyQ, has a win ratio of 75% as opposed to 41% with lucky. Does the use of lucky_Q have an effect on miniQmax vs random play? Again as player 1 miniQmax's performance is worsened by the use of luckyQ, with 88% victories when using the lazy initialization as opposed to just 61% with luckyQ. As player 2, the patter holds, with 55% without lucky and 43% with lucky.

Table 3	P1 win	draw	P1 loss
random v lucky_Qmax	0.48	0.09	0.43
lucky_Qmax v random	0.61	0.06	0.33
random v lazy_Qmax	0.32	0.13	0.55
lazy_Qmax v random	0.88	0.01	0.11

So the lesson I took from these observations is that what is good for Q-learning is not necessarily good for miniQmax. So it seemed necessary to improve miniQmax through different means tailored for it. After some frustration tracing miniQmax behavior, it became apparent that as the minimax algorithm recursively looks into future sequences it encounters many unvisited states, or "keys". I had simply used python's Q.get method with a return value of 0 when these keys were reached. At a hunch that Q was simply too sparse and minimax would most of the time return 0's during recursion, I decided to instead return a small reward of *current depth's sign.* What that means is, if the minimax recursion is on a max pass (player 1) it returns 0.01 when a key is not there, and -0.01 for a min pass. This was accomplished through the same Q.get method, returning None instead of zero to indicate a missing key, so that miniQmax can then decide whether to return +0.01 or -0.01. In our context, this could be thought of as a reward for "discovering" a new state (temporarily by the way, because Q is not updated). This implementation leads to the the following win:draw:loss ratios;

Table 4	No Exploration reward			Rewarded for exploration		
MiniQmax and Q-learning	P1 win	draw	P1 loss	P1 win	draw	P1 loss
lucky_Qlearn v lucky_Qmax	1	0	0	0	1	0
lucky_Qmax v lucky_Qlearn	0.41	0.37	0.22	0.82	0.18	0
luckyQ_learn v miniQmax	1	0	0	0	1	0
miniQmax v lucky_Qlearn	0.75	0.05	0.2	0.93	0.05	0.02
MiniQmax and Random	P1 win	draw	P1 loss	P1 win	draw	P1 loss
random v lucky_Qmax	0.48	0.09	0.43	0.45	0.05	0.5
lucky_Qmax v random	0.61	0.06	0.33	0.93	0.06	0.01
random v miniQmax	0.32	0.13	0.55	0.39	0.13	0.48
miniQmax v random	0.88	0.01	0.11	0.93	0.06	0.01

In the table above the "No exploration reward" column holds the contents of table 2 and table 3 above. The "Rewarded for exploration" column holds the results of adding a reward during recursion as described above. A big change is that of miniQmax as player 2 against Q-learn. Both with and without luckyQ, it goes from losing 100% of the time, to tying 100% of the time. MiniQmax also sees a dramatic improvement as player 1 against Q-learn with and without luckyQ its performance increase respectively is 41%→82% and 75%→93%. miniQmax as player 2 against random doesn't see very dramatic improvement, however as player 1 with and without lucky Q performance increase respectively is 61%→ 93% and 88%→ 93%.

Now I confess that the order of events as I am reporting is somewhat artificial. When I first added the reward for undiscovered keys the numbers I encountered showed even more improvement to miniQmax on all accounts. I later discovered that I had a flaw in the code necessary to produce the above comparisons between behavior of an agent using luckyQ and not using luckyQ. Essentially, the problem was that Q-learning agents can be set to use different Q's. They can use the global Q that is tallying the game in progress, or they can be set to use an internal *fixed* Q. That is how I was trying to compare behavior of miniQmax with and without lucky; however, I had forgotten that although I had intended too, I had not yet extended that functionality of Q-learning to miniQmax! Therefore on the first iteration, all of the above comparisons were made while miniQmax used the *default empty Q map*. Here everything becomes more like black magic, because I am not fully confident with the reasons why there is such a marked improvement, but lets observe what happens when miniQmax has *no initialized Q whatsoever.* It will simply use the temporary reward for undiscovered states in the recursion.

Table 5	Rewarded for exploration			Reward and blankQ		
MiniQmax and Qlearning	P1 win	draw	P1 loss	P1 win	draw	P1 loss
lucky_Qlearn v miniQmax	0	1	0	0	1	0
miniQmax v lucky_Qlearn	0.93	0.05	0.02	0.93	0.05	0.02
MiniQmax and Random						

random v mimQmax	0.39	0.13	0.48	0.22	0.18	0.6
miniQmax v random	0.93	0.06	0.01	0.9	0.09	0.01

As a recap, table 5 is comparing miniQmax with *lazyQ* on the right column; the entries of which are also in table 4, with miniQmax using a Q with even less information, basically the empty Q. Somehow performance of miniQmax against random *improves* going from 48% wins as player 2 to 60%. This revelation was somewhat depressing because it meant that this somewhat random invisible "tree" that miniQmax was using with +0.01 on all max nodes and -0.01 on all min nodes, outperforms all the attempts at Q-learning so far. Here we see its performance against all other player agents;

Table 6	P1 win	draw	P1 loss
miniQmax v ideal	0	1	0
miniQmax v minimax	0	1	0
miniQmax v Qlearning	0.91	0.08	0.01
miniQmax v random	0.94	0.03	0.03
random v random	0.53	0.18	0.29
random v miniQmax	0.16	0.24	0.6
Qlearning v miniQmax	0	1	0
minimax v miniQmax	1	0	0
ideal v miniQmax	1	0	0

Random play references	P1 win	Draw	P2 win
random v ideal	0.0	0.1160	0.8840
random v minimax	0.0460	0.1020	.8520
random v Qlearning	0.536	0.133	0.331

Here miniQmax is configured with the adjustments discovered during the refinement section; a search depth of 3, an empty initial Q, and the small rewards during minimax recursion. In the first two rows we see that, it demonstrates the same property luckyQ imbued Q-learning with, that is it can tie the perfect players minimax and ideal as player 1. The following 4 rows provide a summary of what had gradually been observed in the previous sections, a solid dominance over Q-learning (which is still using its tweaked out luckyQ) and random players. Although not as dominant against random when playing as player2, miniQmax nevertheless performs strongly relative to Q-learning taking 66% of wins. As a player 2 against random, miniQmax lies somewhere between the "perfect" players , ideal and minimax, which take 88% and 85% of wins respectively, and the Q-learning player which only takes 33% of wins. MiniQmax ties 100% of the time as player 2 against Q-learning. Which you can see as somewhat nullifying Q-learning's first player advantage. Finally we see that miniQmax cannot go so far as to at least tie ideal and minimax during disadvantaged play. They both beat miniQmax 100% of the time when miniQmax is player 2. So this "empty" miniQmax basically has given the best performance of the non-perfect player's so far, with no Q-learning involved, terrible. I then obsessively started experimenting with better ways to train regular Q-learning using minQmax, piping trained Q's of of lower depth (Q-

learning is almost like miniQmax of depth 0 in this context) in a chain of duels of increasing depth. Here I met mostly with failure as well until I realized I had a serious flaw in the implementation of miniQmax. By simply calling on Q as *evaluate*, I ignored the fact that during recursion minimax was doing a fair amount of exploration that could be incorporated into Q, not to mention the fact that when end states, tie and win, were reached I was just returning +-0.01 or 0 ! Terminal states should have the max/min reward of +-1 OR 0 for a tie, and if they were visited during recursion it is a perfect opportunity to populate Q. A problem with Q my attempt at Q-learning in this project, is that its always somewhat sparse because we are dynamically creating it. To recap, miniQmax evaluate now does the following;

1) If it has explored a tie: return 0
2) If it is an end game: return +-1 according to whether its a max or min level
3) If its neither of those, which means its reached the depth limit its exploring at
    i) Return the value of Q at that point or 0 if its an unexplored node

These corrections really made a difference in performance. Dropping to a search depth of 2 (which should be detrimental to performance) we have the following;

Table 7	P1 win	draw	P1 loss
miniQmax v ideal	0	1	0
miniQmax v minimax	0	1	0
miniQmax v Qlearning	0.92	0.06	0.02
miniQmax v random	0.92	0.08	0
random v random	0.63	0.13	0.24
random v miniQmax	0.11	0.18	0.71
Qlearning v miniQmax	0	1	0
minimax v miniQmax	1	0	0
ideal v miniQmax	0.48	0.52	0

This is performance unseen so far. MiniQmax ties the strong players as player 1, 100%. Dominates Q-learning and random as player 1 with 92% victories. Is by no means a random player 2 against random, taking 71% of wins. But most impressively, it manages to *draw ideal* 52% of the time player 2. This would be exciting if it weren't so sad, it is still using an empty Q map, i.e. no Q-learning whatsoever. Also to keep in mind, is that Q is not being updated as it is used as that functionality is turned on during training only. So basically miniQmax just sees zeros or final game wins +-1 as the recursion hits intermediate nodes or leaves respectively. The positive aspect of the situation, is that there is little randomness involved in the above figures save for a little in Q-learning's duel (since it uses random choice to break ties in Q) and of course in the duels with random. But any performance changes observed, especially against the strong players, after changing the empty Q map to a trained one, can be attributed to said trained Q.

Results:

Model Evaluation and Validation:

What follows is a miniQmax centric comparison of all the different agent playing policies as presented in tables 6 and 7 above. The difference is that miniQmax is operating under a trained Q map using an improved "piping" training regiment. The training function pipeTrain, in the play module, takes in a "weaker" and "stronger" playing policy as input. To explain

what I mean, we see that in a sense the Q-learning policy is sort of like miniQmax if its search depth were 0. It just returns the min/max of exactly what it sees at the current state. Its not exactly so, because again, Q-learning does break ties with a random choice. But in this sense, a weaker player has search depth lower than a stronger player. I also consider random weaker than search depth 0. Here I drew some inspiration from the optional tensorflow Deep-learning course I first attempted as a capstone, basically the procedure is somewhat like stacks of the training routine used to get luckyQ. First Q is trained with the weaker model as player 1 for a set number of games. Then the roles are reversed and the Q is passed as the initial Q for a set number of duels with the stronger player as player 1. Finally the Q is "piped" to a set number of duels of strong vs strong. In summary;

```
pipeTrain(weak, strong):
        Q = duels(Q, weak vs strong, n_games1)
        Q = duels(Q, strong vs weak, n_games2)
        Q = duels(Q, strong vs strong, n_games3)
        return Q
```

To train the final model, a Q was initialized with 1000 duels of random vs random as has been done before. Then the piping process was done in steps of;
1) random vs Qlearning 400 games (weakest vs depth 0)
2) Qlearning vs minQmax at depth 1 for 100 games
3) minQmax vs miniQmax at depth 2

This yields the final Q, *pipedQ,* to be fed into miniQmax. Here is the miniQmax ratio comparison table with the corrected miniQmax algorithm and pipedQ as its reference Q.

MiniQmax with pipeQ depth 2	P1 win	draw	P1 loss
miniQmax v ideal	0	1	0
miniQmax v minimax	1	0	0
miniQmax v Qlearning	0.6	0.4	0
miniQmax v random	0.9	0.1	0
random v random	0.62	0.2	0.18
random v miniQmax	0.07	0.16	0.77
Qlearning v miniQmax	0	1	0
minimax v miniQmax	0	1	0
ideal v miniQmax	0.27	0.73	0

First of all, hurray times infinity. Second of all, it beat minimax!!! It beat a strong player! Granted minimax has been short sighted by its depth of 2 and we had observed it suffer minuscule losses as player 2 against random at the beginning of the project, so it is most likely that one of those freak game sequences was absorbed by Q during training. But miniQmax shows marked improvement in almost all other categories as well. MiniQmax still ties ideal as player 1, but amazingly it is now taking 72% of the ties when in disadvantaged second player position. With an empty Q map miniQmax would only rake in 52%. It still ties minimax 100% of the time as player 2 as well. It is *not* playing randomly taking 90% of player1 wins and 77% of player 2 wins against random. If it were a random player, the percentages would be closer to 60% and 20% respectively. Strangely it took a hit against its player 1 victories against simple Q-learning with only 60% victories (no losses though). It still ties it 100% of the time as disadvantaged player 1. After a dark period there, this seems to

indicate some measure of success. That being said, and to "curb the enthusiasm" a little bit, as a solution it is *extremely* fragile to variations in the initialization process. This is especially obvious with the IPython notebook, where in order to get the results above the kernel must be restarted so that the random function goes a state that reproduces these figures. For example just by changing the initial number of random vs random games that seed Q from 1000 to 1030, and then to 1007 results in the following drops in performance;

	Initial n_games = 1030			Initial n_games = 1007		
	P1 win	draw	P1 loss	P1 win	draw	P1 loss
miniQmax v ideal	0	1	0	0	1	0
miniQmax v minimax	0	1	0	0	1	0
miniQmax v Qlearning	1	0	0	1	0	0
miniQmax v random	0.95	0.04	0.01	0.98	0.02	0
random v random	0.61	0.2	0.19	0.58	0.16	0.26
random v miniQmax	0.09	0.15	0.76	0.09	0.2	0.71
Qlearning v miniQmax	0.8125	0	0.1875	0	1	0
minimax v miniQmax	1	0	0	0	1	0
ideal v miniQmax	0.49	0.51	0	0.28	0.72	0

MiniQmax's win against minimax is immediately absent. For n_games =1030, the figures almost exactly match those of table 7 "empty Q" , except that the third to last row has been turned to 81% losses to miniQmax (as opposed to 100% ties). n_games =1007 is not as drastically bad, where with the absence of the miniQmax vs minimax win, there is now a victory of 100% against Qlearning. Also 72% draws against ideal as player1 (last row) still hold.

Addendum:

Unfortunately I later discovered a bug in the coordinate transformation logic of the code that was used to produce the above tables. After correcting it, the above observations still hold; that is, miniQmax's final performance is extremely sensitive to variations in the initial conditions. Good training however can be reproduced now with an initial random vs random training of 1000 games, and with an alternate learning rates, gamma = 0.74 and alpha = 0.9 (also see code for random seed value). These conditions yield the slightly improved performance summarized in the following table;

	P1 win	draw	P1 loss
miniQmax v ideal	0	1	0
miniQmax v minimax	1	0	0
miniQmax v Qlearning	0.57	0.43	0
miniQmax v random	0.96	0.04	0
random v random	0.61	0.2	0.19
random v miniQmax	0.02	0.08	0.9
Qlearning v miniQmax	0	1	0
minimax v miniQmax	0	1	0
ideal v miniQmax	0	1	0

Justification:

As a solution to the original problem; that is, to use reinforcement learning to produce a Q function that will allow player agents to decide the next move on a nxn game of tictactoe, it falls short on many respects, but is close to sufficient in others. For one, even though the machinery in place is completely independent of the game size, I only managed so far to analyze the relatively simple cases of 3x3. Even here we end up with something that is by no means robust, as the performance of the final algorithm depends greatly on how "lucky" we get with Q's initial exposure to random games, as we see above. However, by first measuring miniQmax against the "test" agents, ideal, minimax, and Qlearning, with an initial empty Q map, and then by *only* changing that variable, we can see that clearly a Q-learning regiment can potentially lead to greatly improved play. On this last point, I feel that this solution completely satisfies my minor second objective of having agents learn only by the reinforcement of a win, draw or a loss. Put in another way, from pure random game play, it was possible to gradually build up a policy, guided by reinforcement learning, that could hold its own against tailored, special purpose agents, i.e. ideal and minimax. In that sense this solution is moderately adequate.

Conclusion:

Even though there was some success with the methods used in this project by now I realize that applying a reinforcement learning approach to this problem, in the manner I went about it, was maybe not most effective. As pointed out in the previous review, exposure to some basic knowledge of algorithm design could have probably avoided hours of struggle. I did not anticipate encountering such difficulties. It was important for me however to try to figure some of these things out for myself. The first hurdle was the implementation of the agents ideal and minimax, tailored for tictactoe play, for use as benchmarks to gauge effectiveness of any potential solution. The first a "common sense" generalization of the forking strategies of 3x3 play to an nxn board. The second, minimax with alpha-beta pruning, an implementation of a general recursive tree traversal algorithm to the specific problem of tictactoe play. Both basically relying on a notion of measure based on how filled, and how many, lines there are on the board. In terms of the actual solution, in the form of a reinforcment learning guided agent, a naive straightforward adaptation of the Q-learning strategy used in the previous Smart cab project, resulted in a rigid unevenly performing model. It could prove successful as the dominant first player against the "perfect" agents, but basically exhibited mostly random playing behavior in general. Its strategy of dynamically acquiring keys as states were encountered and assigning value to them at the conclusion of each game, was not enough (as I implemented it) to give rise to complex enough play. These explorations lead to blending minimax search with Q-learning. The initial idea being to replace minimax's evaluate with calls to Q. Eventually, by including minimax's depth searches as part of the training process, and by using a ladder like "pipe" training to gradually build Q with increasing depth we arrived at the final version of miniQmax. Although very fragile to variations in initial training conditions, it nonetheless showed some encouraging tallies of win:draw:loss ratios against the benchmark agents, ideal, minimax and Qlearning (with luckyQ) when a "good" initialization had taken place. One of the main benefits extracted from this project, is my gaining familiarization with a general algorithm like minimax, and understanding it could be combined with something like a Q dictionary store. I think now, it could be maybe more effective to use the minimaxMeasure, that provides such strong measures of state, and simply store its values in Q. That is, just use Q like a database to relieve it from having to perform calculations it had previously made.

In the end, miniQmax is easily outmaneuvered by a human player, and in that sense, the objective of "optimal" play was not achieved, I feel like the combination of tree searching with a Q function might prove useful as a tool in some other less performance demanding domain. Maybe I'll find some more appropriate use of it at some point. Above anything however, I feel in during the course of this project I gained more familiarity in using reinforcement learning as a strategy for solving problems, and became aware some of the issues one can face when trying to do so.