Data 640 Summer 2022

Professor Steve Knode

Tiania Chan

tiania.chan@gmail.com

Assignment 5a: TicTacToe QLearning

**Introduction**

Code Overview:

Starting right at the beginning of the Colab notebook are libraries for the TicTacToe project. Numpy for the creation and management of arrays that will be used in the project. Matplotlib used to create and save the plots used to view the results. Lastly the library Random to generate random numbers for the agent. Line 8 is used for the first set of 3 different epsilon values that will be used to test the algorithm. It’ll be done for 3x3 tictactoe game followed by 4x4 tictactoe game.

The next set of lines set the values for the learning parameters of the agent. Setting the number of episodes that it will go through which is set to 1 milling. The learning rate at .01, number of random episodes and declaring the minimum and maximum epsilon values. These are tested at the values .02 minimum and 4.0 maximum as well as .5 minimum and 1.0 maximum as well.

The next function is the epsilon decay function which gets the value of epsilon after each iteration. The function max(min\_ epsilon, min(max\_ epsilon, num\_random\_episodes/(episode + 1))) is to get the maximum value. Either the minimum value of epsilon or if the minimum between the max epsilon and the number of episodes divided by the current episode plus 1.

The next function gets the available moves by exploring where on our board is the value still set to 0 which means that the agent has not yet explored that move. The next function is called to convert the board into integers. N\_states = 3\*\* 9 gives thetotal number of possible states which is 19,683 x 9 different possible actions bringing up the total to 177,147 total moves or states. However, given that a tictactoe board does not rotate, there aren’t as many unique staes. The next line, state = 0 is to initialize the state of the cell at 0 since that is the indication that the agent has not yet explored that location in the board. Finally, the for loop is to flatten the board and return the integer value of the state.

After these functions have been declared to set up the board, the next step is the function to plot out the results on graphs. In the next cell, the value of x is the number of episodes that the agent has explored, and y is the value of epsilon. The plot size is a 6 by 6. The next few lines are the descriptors of the plot for the x and y labels, the title and then the next two lines are to close the figure and save the figure so that the next graph can be plotted. The following function will close and redisplay the values of epsilon compared to the number of episodes.

The next function, is\_terminal(board) is the most important function. If the board is terminal the function will return the result 1 or -1. It will check for a win by looking to see if columns, rows, or diagonals are all the same then it means the algorithm won. Therefore the board will return a 1 or a -1. If it is a tie and full, then it will return a 0 Otherwise, if it is not terminal, it will return none so it is still going.

The first for loop is to go through the board and check to see if any of the rows, columns, or diagonal signal a win, if not, return none. After that, declare the past results, win probability, draw probability and the sum of the q table in lists for storing the results for later.

The for loop starting with episode in range(episodes) is as follows: First get the current episode of the table, Terminal will be none as this is not the last turn so far. Create theboard, set the current state of the board, and start the turn at 1 and declare the episode memory for storage to reference later.

The while loop talks about if the agent isn’t finished exploring the board, it will explore by using np.random.random() and record each turn and record the overall reward value for this method of exploring the tictactoe board. Once this is finished, the code moves to update the qtable. Since there is a chance of the code failing if there hasn’t been at least onetie and one win, it will append nothing to the draw and win probabilities. Finally, if everything is finished, the code will plot he figures with the values of epsilon, the win probability, and draw probability.

Finally, the next cell displays the board per action to show the iterations of each turn the agent takes against itself.

3x3 Results:

A major caveat to this discussion of the results is that the code for displaying the board with each iteration was let run for more than 5 hours but finally stopped as there was not a lot of change in the actual display of the results despite the code running. Each iteration with the different minimum and epsilon values took approximately an hour using Google Colab pro using GPU and high RAM to optimize Google’s cloud capabilities for running the algorithm.

The first variation of the minimum and maximum epsilon values are .01 and 1.0 respectively. Chart

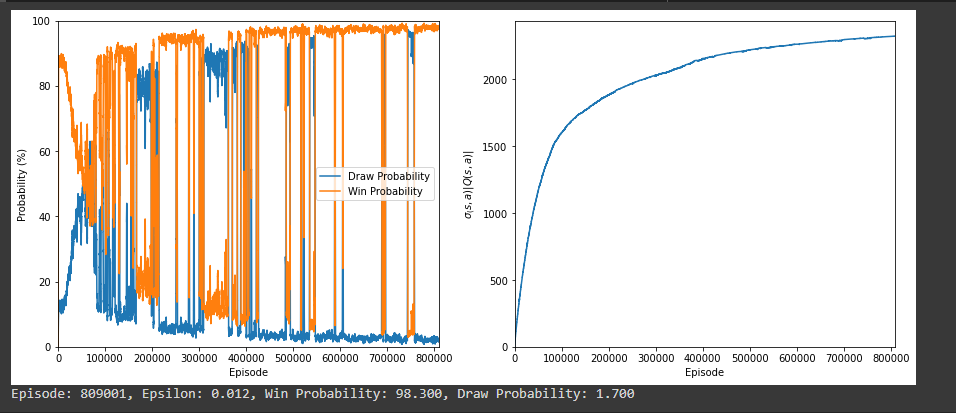
Description automatically generated with low confidenceThe beginning of the graph shows that the win probability and draw probability first began separate due to the agent still learning what moves to make to win. Then they converge as the agent learns what moves to make in the board to win. At this point, the agent has learned no new moves yet and therefore plays to a draw with itself. As the agent learns new moves, the lines diverge again to allow the agent to win against itself. The win and draw probability were pretty level for most of the runs with the two only converging when the agent was learning a new way to win and so had not yet caught up with itself. This pattern is consistent throughout the later runs using the 3x3 tictactoe board. On the right is the epsilon value for each episode or run where there is a learning curve at the very beginning because it is all new moves. Once all these moves have been completed, that is where the graph plateaus. We can see this reflected that the agent has a win probability against itself of 23.3% and a draw probability of 76.6% so it was learning the game pretty effectively to be able to learn against itself.

The second iteration of the 3x3 tictactoe board was to use a minimum epsilon value of .01 and a maximum of 4.0 to see if changing the maximum but keeping the minimum value the same would make a different in the results. From the graph below, the change is apparent. Chart, histogram

Description automatically generated

For the most part, having a higher maximum epsilon allowed the agent to win far more often than it did lose. The blue line for the draw probability versus the win probability start off far apart and don’t really converge together much. So the maximum epsilon value change the agent’s learning ability so that it wins against itself far more often than it loses or ends the game in a tie. The win probability ends at 95.1% with a draw probability of 4.9.

The final variation using the 3x3 tictactoe game for the agent was to use a minimum epsilon value of .5 and a maximum value of 1.0 to see what the difference was if the minimum value was changed to compare to the first run.

When the maximum was changed, the agent won a lot more often against itself than it lost. With the minimum changed, the overall results are like changing the maximum in that the win probability was 98.3% and the draw probability was 1.7% but the means that the agent took to get there was far different. Each graph has shown that the win and draw probability of the agent starts off separate. The difference here is that where in the first graph, the agent has an early plateau where it ends the game in a draw against itself, with this case, there are two sections in our results that show that the agent ended in draws with itself more often than it lost or won. The graph does still display convergences for each time the agent learned a new move and so was more likely to win against itself. However, even though the agent has two plateaus instead of one, with the overall win probability being higher than either of the results, this variation is the most successful and therefore the one that is preferred.

4x4 Results:

A caveat to mention with the 4x4 tictactoe development is that there were a few lines changed to account for the massive amount of possibilities that needed to be calculated using a 4x4 board for tictactoe instead of a 3x3 board. The very last variation is incomplete and the results are after 6 hours of running the agent using the exploratory method for reinforcement learning.

First, the changes that were made for the 4x4 tictactoe board as follows:

|  |  |
| --- | --- |
| **Before** | **After** |
| q\_table = np.zeros((3 \*\* 9, 9)) | q\_table = np.zeros((3 \*\* 16, 16)) |
| n\_states = 3 \*\* 9 | n\_states = 3 \*\* 16 |
| state += ((x + 1) / 3) \* n\_states | state += ((x + 1) / 4) \* n\_states |
| n\_states /= 3 | n\_states /= 4 |
| if np.any(np.abs(column\_sum) == 3): | if np.any(np.abs(column\_sum) == 4): |
| if np.any(np.abs(row\_sum) == 3): | if np.any(np.abs(row\_sum) == 4): |
| if np.any(np.abs(diagonal\_sum) == 3): | if np.any(np.abs(diagonal\_sum) == 4): |
| board = np.zeros((3, 3)) | board = np.zeros((4, 4)) |
| action = 3 \* action\_square[0] + action\_square[1] | action = 4 \* action\_square[0] + action\_square[1] |
| action\_square = [action // 3, action % 3] | action\_square = [action // 4, action % 4] |
| fig.savefig("training\_metrics.jpg") | fig.savefig("training\_metrics2.jpg") |
| np.save("q\_table.npy", q\_table) | np.save("q\_table2.npy", q\_table) |
| np.save("draw\_probs.npy", draw\_probs) | np.save("draw\_probs2.npy", draw\_probs) |
| np.save("win\_probs.npy", win\_probs) | np.save("win\_probs2.npy", win\_probs) |

First is to change the size of the q table to accommodate for a 4x4 board. Next the number of states was changed. Getting the state was changed to be divided by 4 instead of 3. Next finding the column, row, or diagonal equivalent of a tictactoe win was changed. The number of actions was multiplied by 4 instead of 3 and saving the figures was changed in order to be able to sho the different figures in a 4x4 tictactoe versus 3x3 tictactoe board.

The first variation was for a minimum of .01 and a maximum of 1.0. Th graphs of the results is vastly different from the .01 and 1.0 epsilon values. For example, where the line chart demonstrated a convergence and divergence from where the agent was able to win against itself versus draw, the line chart shows almost no difference and a low probability for either a win or a draw with the percentages coming in at .6% chance of winning and a .4 chances of drawing with itself. Therefore these values of epsilon would lead to the agent losing. We can see that on the right graph as well where the epsilon values had a linear relationship with the number of episodes. Chart, line chart

Description automatically generated

The next iteration used a minimum of .01 and a maximum of 4.0. We don’t see too much difference in the graphs as they have similar shapes.

Chart, line chart

Description automatically generated

However, the value of the epsilon values starts higher than previously. Also, the win probability slightly increased to .8% and the draw probability went up a little higher to .2%. This is the difference but still not optimal as a reinforcement learning model as the agent is more likely to lose in a game than win or tie with itself.

Finally, the last variation is for a minimum epsilon value of .5 and a maximum of 1.0. Despite the initial hypothesis that this would be the quickest algorithm since there was the least amount of change between the minimum and maximum values of epsilon, this was the version of the reinforcement algorithm that took the longest to run. The algorithm was stopped at about 6 hours run time and below is the results:

Chart, line chart

Description automatically generated

As with the other two variations, the 4x4 tictactoe reinforcement learning agent shows only a win probability of .6% and a draw probability of .4%. The conclusion that can be given is that the reinforcement agent does not work well with epsilon values tha were given. With more time, more epsilon values can be explored. Perhaps a different approach with less saving and redrawing the graphs might help to display the results without taking more than 6 hours to go through all the runs. Another suggestion is to use a greedy approach instead of an exploration approach to hopefully use less processing time and power.

References

Rudes, R. ( 2020, Sept 8). An Introductory Reinforcment Learning Project: Learning Tic-Tac-Toe via Self-Play Tabular Q Learning. Retrieved from https://towardsdatascience.com/an-introductory-reinforcement-learning-project-learning-tic-tac-toe-via-self-play-tabular-b8b845e18fe