Reinforcement Learning

Tic-Tac-Toe

Melissa Hunfalvay

Email: Melissa.Hunfalvay@gmail.com

Data 640 9040

Spring 2022

Professor Steve Knode

Date: March 22nd, 2022

Code Overview

The purpose of the exercise is to create a model that learns how to play the game Tic-Tac-Toe. There are two versions of the game, the first, is a 3-x-3 matrix and the second is a 4-x-4 matrix. Original code is credited to Ryan Rudes (2020). The Q-learning code steps are outlined in Table 1.

Table 1: Q-Learning Code Steps for Tic-Tac-Toe

Step	Purpose	Description	Sample Code (for 3x3 game)
1	Import required libraries	Import libraries for numpy, matplot, random numbers and Ipython.display	`import numpy as np`
2	Initialize the Q-Table shape	3 to the power of 9 (i.e. 9 squares) = 19,683. There are 9 actions. Therefore table size is 19,683 x 9 = 177,147 (Rudes, 2020)	`q_table = np.zeros((3 ** 9, 9))`
3	Set the learning parameters		
3.a	Episodes	All states that come in between an initial-state and a terminal-state. Each episode if a complete loop that is independent from each other (Zychlinski, 2019). Simply, the number of times the agent plays itself to obtain a result.	`episodes = 1000000`
3.b	Learning rate	The learning rate is a configurable hyperparameter used in the training of neural networks that has a small positive value, often in the range between 0.0 and 1.0. The learning rate controls how quickly the model is adapted to the problem (Brownlee, 2020)	`learning_rate = 0.01`
3.c	Number of random episodes	The number of times the episodes or independent cycles, that are run to provide learning. The random function provides a starting point (randomly) from which the episodes run.	`num_random_episodes = 10000`
3.d	Epsilon: Minimum and maximum	Epsilon refers to the probability of choosing to explore, exploits most of the time with a small chance of exploring (Shawl, 2020). Minimum and maximum thresholds are set.	`min_epsilon, max_epsilon = 0.01, 1.0`
4	Epsilon decay function	A function and hyperparameter that controls how much the agent should explore and exploit when using epislon-greedy policy (Katnoria, 2018).	`def get_epsilon(episode)`
5	Graphing	Define parameters of a graph to show exploration and learning	`X = np.arange(0, episodes)`
6	Set game restrictions for agent to follow	For example, make sure the occupied squares (i.e. those with results already determined) are no longer available. The function sets unoccupied squares to 0, X's to 1 and O's to -1.	`def get_available_moves(board):available_moves = np.argwhere(board == 0).tolist()return available_moves`
7	Convert between a 3x3 board representation and an integer state	If the cell is -1, you don't change state. If the cell is 0, you change state by one-third of the window size. If the cell is 1, you change state by two-thirds of the window size.	`def board_to_state(board):n_states = 3 ** 9 state = 0`
8	Terminal Function	Determines when the game has reached a terminal state. If terminal stat is reached then this function also returns the result.	`def is_terminal(board):`
9	Append the Q-Table	Updates the Q-table based on the latest iteration	
9.a	Past Results	Stores results of each simulated game. 0 = tie, 1 = player positive integer won, -1 = player with negative integer won.	`past_results = []`
9.b	win-probs	Stores the list of percentages updated after each episode. Each episode tells the fraction of games up to the current episode in which the player won (Rudes,	`win_probs = []`
9.c	draw-probs	Draws the percentages corresponding to the fraction of games in which the draw occurred (Rudes, 2020).	`draw_probs = []`
9.d	sum_results	Tallies the results to this point	`sum_q_table = []`
10	Save data		
10.a	Frequency	Data is saved at the end of 1000 episodes	`averaging_distance = 1000`
10.b	Type of Data being saved	Q-Table results	`np.save("q_table.npy", q_table)`
10.c	Type of Data being saved	Draw probability metrics/results	`np.save("draw_probs.npy", draw_probs)`
10.d	Type of Data being saved	Winning probability metrics/results	`np.save("win_probs.npy", win_probs)`
11	Training Script	Start with a clean board - no x's or 0's	`board = np.zeros((3, 3))`
	(See Appendix A)	Print the board	`print (board, "\n")`
		Agent takes action	`actions = q_table[current_state] * 1`
		Print the updated board	`print (board, "\n")`
		Obtain input from human	
		Repeat until a final result is either win or draw	`while not isinstance(terminal, int):`

3 x 3 Tic-Tac-Toe Results

Parameter changes and results for the 3 x 3 game options are explained in Table 2. Highlighted purple areas show the changes from one model to the next.

Table 2: 3 x 3 Tic-Tac-Toe Reinforcement Learning Models

Model Characteristics					**Results**			
Model #	*Minimum Epsilon*	*Maximum Epsilon*	*Episode #*	*Learning Rate*	*Win Probability*	*Draw Probability*	*Time to Complete (min)*	*Notes*
Baseline 1	0.01	1.00	1,000,000	0.01	99.000	1.000	91	See Figure 1. Early in training results show a win probability as high because both "players" are taking random actions. In tic-Tac-toe there is more of a chance for win states than draw states. As the agent begins to learn (i.e. play the game according to it's table policy) there is a sequence of fluctuations in the win/draw results Around the 300000 mark there are more draws as the Agent plays competitively with itself. Up to this point the results are expected. However, then, towards the end of training the agent almost always casued a win. Why? Because the Agent learned earlier to evaluate and then learn the board configurations and there were fewer yet-to-be-discovered tactics (and graph fluctuations) the longer the game is played (i.e. the more episodes) which results in a win most often. Remember, in tic-Tac-toe there is more of a chance for win states than draw states...if you know how to play you can exploit the win options more often.
2	0.50	2.00	1,000,000	0.01	8.400	81.600		Changes to the epsilon min and max values affected the model in significant ways. First, the result at the end of the 1 million episdes was oposite the first model, with probabilities of draw results occuring much more often. Why? As the epsilon min-max range is much larger than model 1 this expands the ability to explore (even though the decay rate remains the same). This led to a greater change of learning early as seen in the fluctuations in the probability plot for Model 2 (Figure 2). At about 9,000,000 episodes the win/draw probabilities switched and draw remained high (purple box, Figure 2). This is also demonstrated in Appendix A with the training dataset. Why? This indicates a state where the agent was playing optimally against itself encountering a draw as it is attempting to maximise reward. In model 2 the agent does not get to a state where it is has learned to evaluate the tactics early in the board configurations as was found in Model 1, thereby leading to more draws than wins.
3	1.00	3.00	1,000	0.01	86.700	13.300	4 min 47 sec	Significantly reducing the number of episodes did not allow the Agent to have enough attempts to learn how to play the game. Therefore, as seen in the graphs (Figure XX) the win probability remains higher as the game has more chances to win, the draw probability remains low. There are no fluctuations of learning.

Figure 1: Win/Draw Ratio for Baseline Model 1 Results

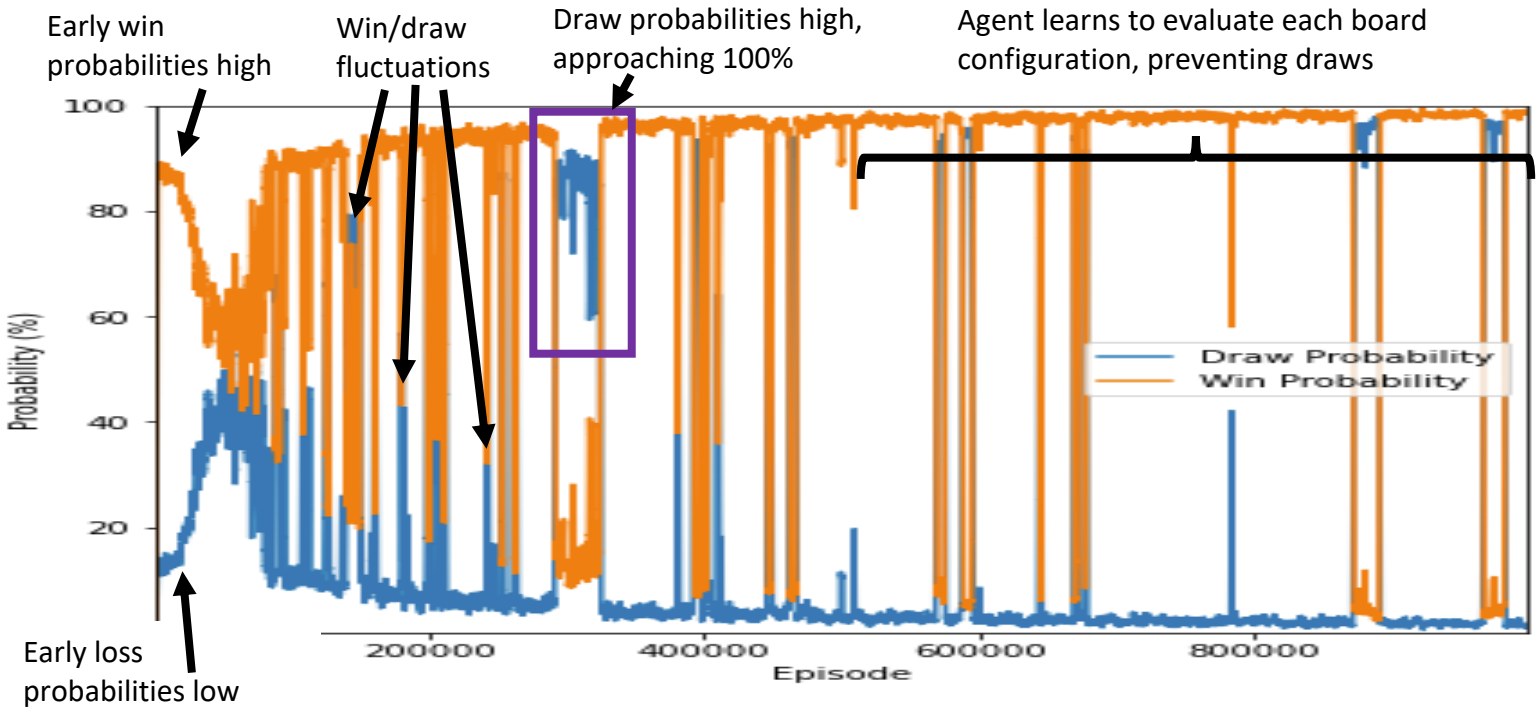


Figure 2: Win/Draw Ratio for Model 1 Versus Model 2 Training

Model 1 Model 2 Win/draw probability % switch

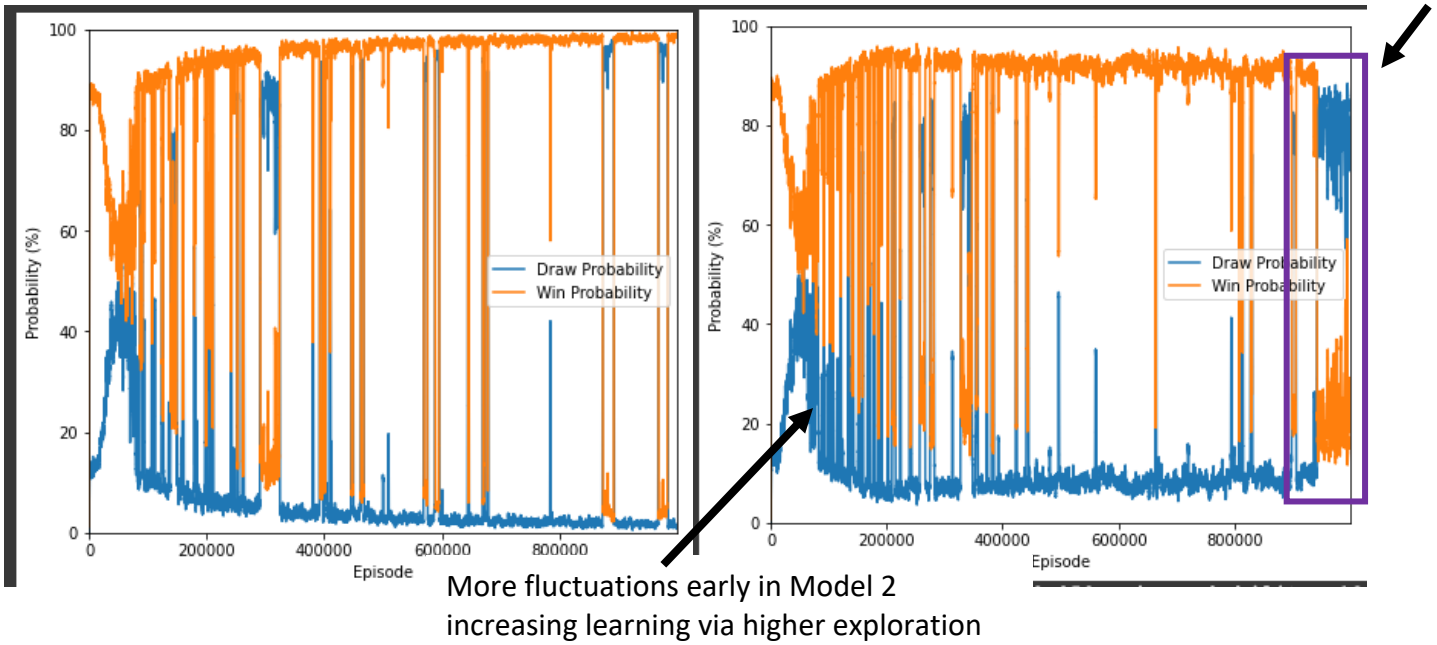
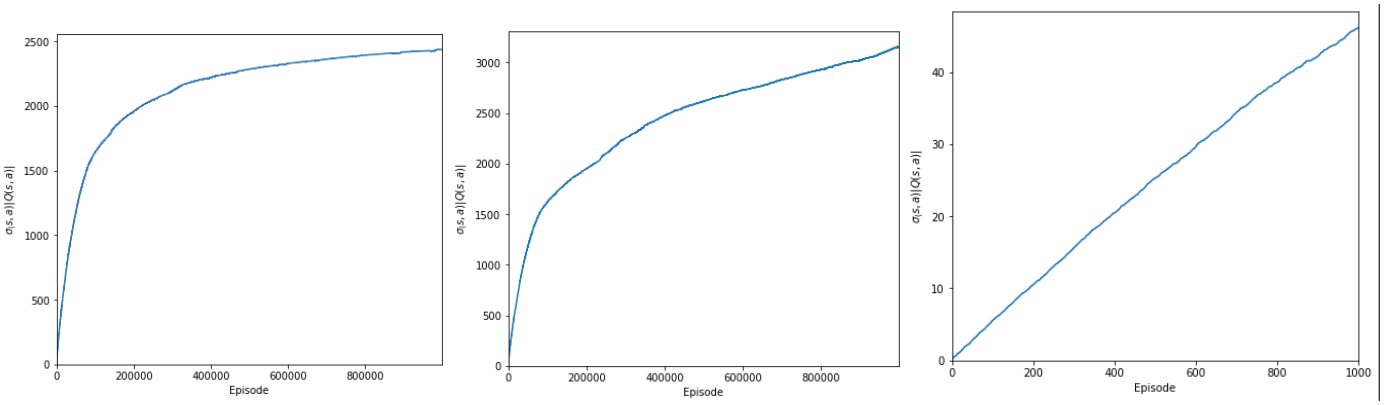


Figure 3 compares the sum of the Q-values over the number of episodes. The formula is Q,s,a)|Q(s,a)| whereby s = state, a = action = q-value. Q-Value is the output from the function for any given state-action pair (Deeplizard, 2018). Q – refers to quality. The trajectory of the line in the model graphs shows the quality of the state-action pair over time (episodes) of learning. Both model 1 and 2 show early learning, model 2 learns mores quickly early based on the graph trajectory, however, model 2 then slows around 100,000 episodes. Model 3 shows no learning as there are not enough episodes to allow for the rewards to be understood.

Figure 3: Graph displaying Q-Value by Episode



Model 1 Model 2 Model 3

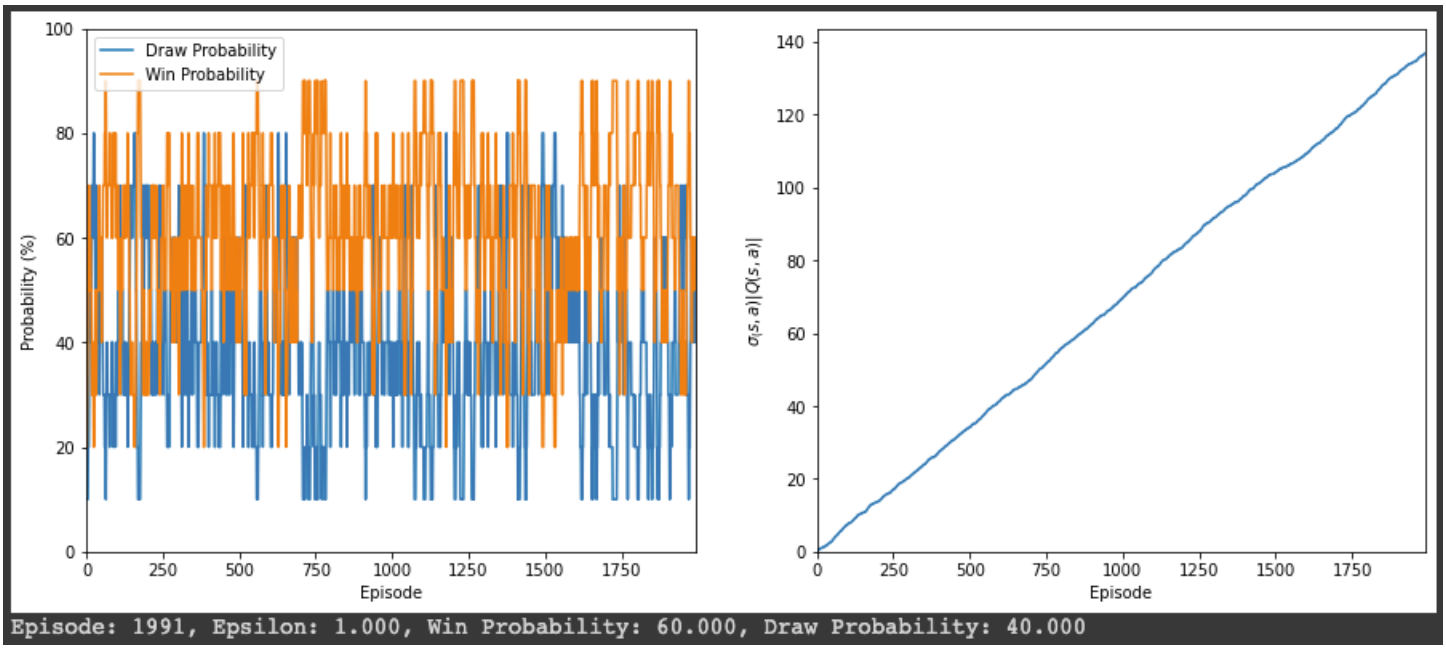
4 x 4 Tic-Tac-Toe Results

The 4 x 4 Tic-Tac-Toe board has many more states, actions, and cells then the 3 x 3 board (see Table 2). This impacts several factors including increasing calculation time and therefore requiring more processing power or alternatively reducing the number of episodes to complete a run.

Table 2: Compares States, Actions and Cells in the 3x3 and 4x4 Tic-Tac-Toe Game Boards

Board Type	Number of States	Number of Actions	Number of Cells
3 x 3	19,683	9	177,147
4 x 4	43,046,721	16	688,747,536

A larger board also increases the environment size and therefore, increases the number of possible actions the agent can take. Therefore, the longer it takes for the agent to learn the environment. This can be seen in Figure 4.a. (and Appendix B) where win and draw results consistently fluctuate as the agent has yet to learn how to effectively find the reward, i.e. the desired game outcome. This is further illustrated in 4.b. as the learning rate continues to show an upward trend and has not leveled off as seen in the 3 x 3 graphs (Figure 3: model 1 and Model 2). "Melissa" versus "Agent" in the training dataset (Appendix C) results in my ability to beat the Agent again, indicating that the agent has yet to learn how to play (I profess to be at least a competent Tic-Tac-Toe "gamer").



4.a. Win/Draw Probabilities

4.b. Learning rate

Figure 4: Results of the 4 x 4 Tic-Tac-Toe Game

References

Brownlee, J. (2020). Understanding the impact of learning rate on neural network on performance. *Machine Learning Mastery,* retrieved on March 16th, 2022 from: https://machinelearningmastery.com/understand-the-dynamics-of-learning-rate-on-deep-learning-neural-networks/

Deeplizard (2018). Reinforcement learning – Developing intelligent agents. *Deeplizard.com*, retrieved on March 17th, 2022, from: https://deeplizard.com/learn/video/eMxOGwbdqKY

Katnoria, M. (2018). Visualizing epsilon decay. Observable HQ, retrieved on March 16th, 2022 from: https://observablehq.com/@katnoria/visualising-epsilon-decay

Rudes, R. (2020). An introductory reinforcement learning project: Learning tic-tac-toe via self-play tabular Q-learning. *Towards Data Science,* retrieved on March 14th, 2022 from: https://towardsdatascience.com/an-introductory-reinforcement-learning-project-learning-tic-tac-toe-via-self-play-tabular-b8b845e18fe

Shawl, S. (2020). Epsilon-greedy algorithm in reinforcement learning. Geeks for Geeks, retrieved on March 16th, 2020 from: https://www.geeksforgeeks.org/epsilon-greedy-algorithm-in-reinforcement-learning/

Zychlinski, S. (2019). The complete reinforcement learning dictionary. *Toward Data Science.* retrieved on March 16th, 2022, from https://towardsdatascience.com/the-complete-reinforcement-learning-dictionary-e16230b7d24e#:~:text=Episode%3A%20All%20states%20that%20come,we%20consider%20an%20infinite%20episode.

Appendix

Appendix A: Training Data Explained

Agent = X. Melissa = 0

`[[0. 0. 0.]`
`[0. 0. 0.]`
`[0. 0. 0.]]`
	This first block of 0's represents an unoccupied board i.e. no values are in there to start	

`[[0. 0. 0.]`
`[0. 1. 0.]`
`[0. 0. 0.]]`
	The second block shows 1 in the center – this represents an X. This move was made by the computer Agent and is based on the Q-Value obtained from the test data	

`0 0`
`[[-1. 0. 0.]`
`[ 0. 1. 0.]`
`[ 0. 0. 0.]]`
	This third block I inputted 0 0 this resulted in a -1 which is a 0 value on the board. The first zero is the row. The second zero is the position within the row	

`[[-1. 1. 0.]`
`[ 0. 1. 0.]`
`[ 0. 0. 0.]]`
	This fourth was then inputted by the agent and they responded with a 1 (which represents an X) in the middle of the top row.	

`0 2`
`[[-1. 1. -1.]`
`[ 0. 1. 0.]`
`[ 0. 0. 0.]]`
	I then inputted 0 2 this resulted in a -1 which is a Zero value on the board in the top row. 2 represents position 3 in the row as Python starts counting at 0.	

`[[-1. 1. -1.]`
`[ 0. 1. 0.]`
`[ 0. 0. 1.]]`
	The agent and then responded with a 1 (which represents an X) in the bottom row right.	

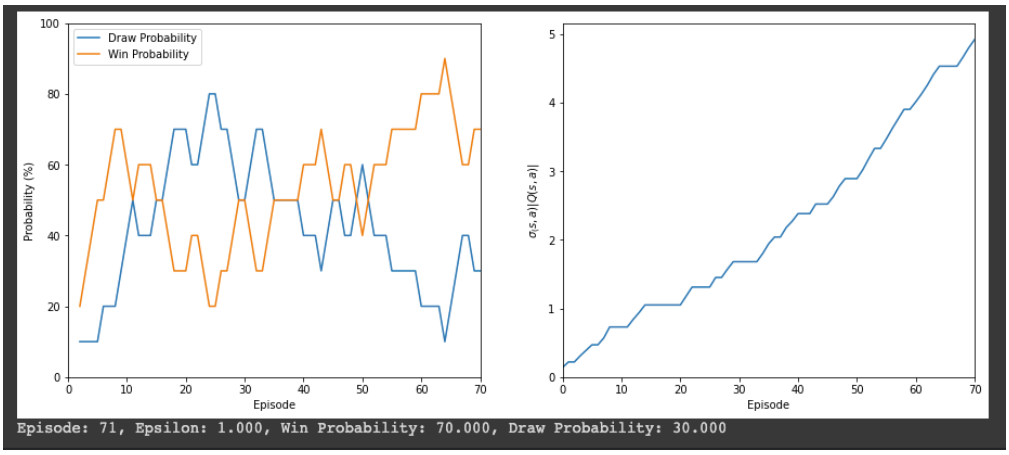
`2 1`
`[[-1. 1. -1.]`
`[ 0. 1. 0.]`
`[ 0. -1. 1.]]`
	I then inputted 2 1 this resulted in a -1 which is a Zero value on the board in the middle of the bottom row	

`[[-1. 1. -1.]`
`[ 0. 1. 0.]`
`[ 1. -1. 1.]]`
	The agent and then responded with a 1 (which represents an X) in the bottom row left.	

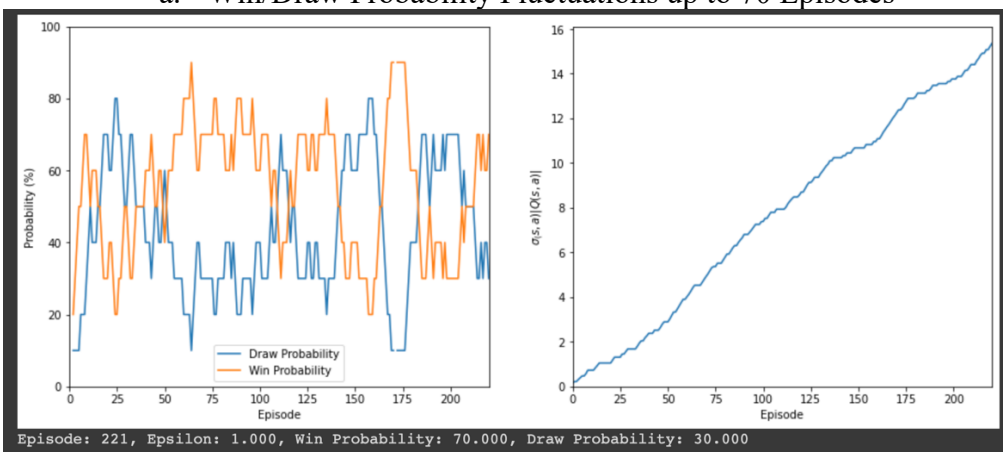
Result = **DRAW**

Appendix B

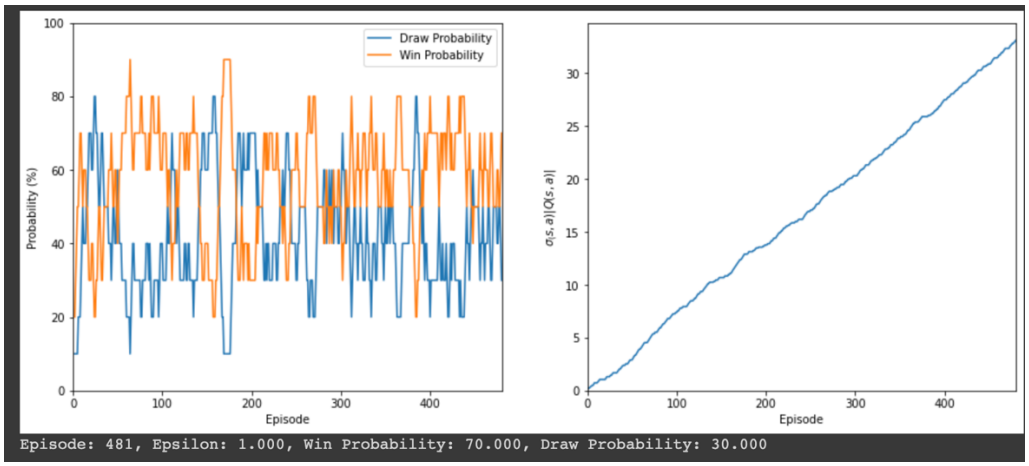
Agents Attempts at Learning the 4 x 4 Tic-Tac-Toe Board



a. Win/Draw Probability Fluctuations up to 70 Episodes



b. Win/Draw Probability Fluctuations up to 250 Episodes



c. Win/Draw Probability Fluctuations up to 500 Episodes

Appendix C

Melissa Beats the Agent after 2000 Episodes of Training

Agent = X. Melissa = 0

Result = **WIN (Melissa)**

```
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]

[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 0. 0.]]

2 2
[[ 0.  0.  0.  0.]
 [ 0.  0.  0.  0.]
 [ 0.  1. -1.  0.]
 [ 0.  0.  0.  0.]]

[[ 0.  0.  0.  1.]
 [ 0.  0.  0.  0.]
 [ 0.  1. -1.  0.]
 [ 0.  0.  0.  0.]]

1 2
[[ 0.  0.  0.  1.]
 [ 0.  0. -1.  0.]
 [ 0.  1. -1.  0.]
 [ 0.  0.  0.  0.]]

[[ 0.  0.  0.  1.]
 [ 1.  0. -1.  0.]
 [ 0.  1. -1.  0.]
 [ 0.  0.  0.  0.]]

0 2
[[ 0.  0. -1.  1.]
 [ 1.  0. -1.  0.]
 [ 0.  1. -1.  0.]
 [ 0.  0.  0.  0.]]

[[ 0.  0. -1.  1.]
 [ 1.  0. -1.  0.]
 [ 0.  1. -1.  0.]
 [ 0.  0.  0.  1.]]

3 2
[[ 0.  0. -1.  1.]
 [ 1.  0. -1.  0.]
 [ 0.  1. -1.  0.]
 [ 0.  0. -1.  1.]]
```