**University of Maryland Global Campus**

DATA 640 – Predictive Modeling

**Assignment №5a – Reinforcement Learning Practice Exercise –Tic-Tac-Toe**

**Deadline:** End of week 10, 11:59 pm Eastern Time

Submission via LEO.

*This is an* ***individual*** *assignment. Each student will complete the assignment outlined below and post his/her written results to the appropriate assignment. Please note that only 1 document is allowed to be submitted.*

Contents

[Grading criteria 1](#_Toc58551891)

[Reinforcement Learning Practice Exercise 1](#_Toc58551892)

[Submission 3](#_Toc58551893)

[Content 3](#_Toc58551894)

[Grading 5](#_Toc58551895)

[Original Article: An Introductory Reinforcement Learning Project: Learning Tic-Tac-Toe via Self-Play Tabular Q-learning by Ryan Rudes 5](#_Toc58551896)

[Original Tic-Tac-Toe Reinforcement Learning Code by Ryan Rudes 15](#_Toc58551897)

# Grading criteria

Submitted assignments will be graded for (a) content, (b) document quality (i.e. formatting, following guidelines, pleasant to read, etc.), and timeliness of submission. Assignments submitted late will be deducted 5 points for each day it is late.

# Reinforcement Learning Practice Exercise

You’ll learn to build a deep learning neural network, focusing on image recognition, sometimes called computer vision. This is one of the primary applications of deep learning neural networks. To do this you’ll do four steps.

1. Read *An Introductory Reinforcement Learning Project: Learning Tic-Tac-Toe via Self-Play Tabular Q-learning* by Ryan Rudes: [https://towardsdatascience.com/an-introductory-reinforcement-learning-project-learning-tic-tac-toe-via-self-play-tabular-b8b845e18fe](about:blank) This article is replicated in Appendix 1.
2. Follow this link which will open the 3x3 Tic-Tac-Toe code in Google Colab: [https://colab.research.google.com/drive/1w3RYXZ\_tg80qNDQZf1I2KyZcigwz8Not?usp=sharing](about:blank)
   1. You can also copy the code in **Appendix 2** of this document which is the same as in the link
3. Run the code until the end.
4. Rerun the code with different min and max epsilons. For example min=0.5 and max=2. Do the results change? Explain.
5. Look at the resulting plots with draw and win probabilities and also the $\sigma\_(s, a) |Q(s, a)|$ diagram. Explain what they are. Are the win and draw probabilities during the beginning, middle and end episodes what you would expect?
6. Expand the problem from 3x3 Tic-Tac-Toe to 4x4 Tic-Tac-Toe
   1. Make a copy of the notebook code into a new notebook. Copy the code (ctrl+a and ctrl+c) then File->New Notebook and paste the code there (ctrl+v)
   2. Change the following lines of code in the copy 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) |

The summary of changes is to change the number of states from 3 \*\* 9 to 3 \*\* 16 in q\_table and n\_states, since each board piece still has 3 states (empty, circle or x) and now we have a 4x4=14 sized board. Then the rest of the changes are to change all 3’s to 4’s.

1. Compare the diagrams from the 4x4 case to those you obtained in the 3x3 case. What are the similarities and the differences?

# Submission

Each student will submit a single document conforming to the guidelines and standards outlined below.

**Document format**:

* limited to 4 pages (**excluding** *title page*, *references*, and *appendix*),
* **Double-spaced**, 12 point Times New Roman font, 1” margins, Bottom-right page numbering.

**Note:** Submitted report must be either in MS Word or PDF format and titled: “*Practice5\_LastName*”.

Only one document will be allowed to be submitted.

# Content

(Note that the document must have at ***clearly marked sections*** at least for the items listed below)

1. **Title page** (*1 page limit*): course number and term, assignment number and project title, student name and contact information, instructor’s name. *Format it so it looks pleasant and presentable.* Follow formatting guidelines above.
2. **Tic-Tac-Toe Q-Learning Code Overview.** Using your own words, provide an overview of the code.
3. **3x3 Tic-Tac-Toe Results.** Show the resulting plots with draw and win probabilities and also the $\sigma\_(s, a) |Q(s, a)|$ diagram in the 3x3 case and explain what you see and if they are as you would expect. How does changing the min and max epsilon affect the results?
4. **4x4 Tic-Tac-Toe Results.** Show the resulting plots with draw and win probabilities and also the $\sigma\_(s, a) |Q(s, a)|$ diagram in the 4x4 case and explain what you see and if they are as you would expect. Compare to the 3x3 case.

**University of Maryland Global Campus**

DATA 640 – Predictive Modeling

**Student Name**: **Practice Exercise: Reinforcement Learning** Total points: 100

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Content** | **Explanation** | **Points** | | **Comment** |
| **Total** | **Earned** |
| **Written Report** |  |  |  |  |
| * 1. Code Overview | Is the Q-learning code well explained? | 20 |  |  |
| * 1. 3x3 Tic-Tac-Toe Results | Are the results for the 3x3 case well explained?  How does changing the min and max epsilon affect the results? Explain. | 70 |  |  |
| * 1. 4x4 Tic-Tac-Toe Results | Are the results for the 4x4 case well explained? Are they meaningfully compared to the 3x3 case? | 10 |  |  |
| **Total** |  | **100** |  |  |
|  |  |  |  |  |
| * + - 1. **Less** |  |  |  |  |
| Page limit | Is the submission written within specified page limits? 2.5 points will be deducted for submissions substantially over the page limits | - 2.5 |  |  |
| Late submission (less) | 2.5 points will be deducted for each day the assignment is late. | -2.5 each day |  |  |
| **Final Grade for Assignment 5** |  |  |  |  |

# Original Article: An Introductory Reinforcement Learning Project: Learning Tic-Tac-Toe via Self-Play Tabular Q-learning by Ryan Rudes

[Sign in](about:blank)

An Introductory Reinforcement Learning Project: Learning Tic-Tac-Toe via Self-Play Tabular Q-learning

[Ryan Rudes](about:blank)

#### **Ryan Rudes**

#### **Sep 8·8 min read**

In this project, I’ll walk through an introductory project on tabular Q-learning. We’ll train a simple RL agent to be able to evaluate tic-tac-toe positions in order to return the best move by playing against itself for many games.

First, let’s import the required libraries

Note that tabular q-learning only works for environments which can be represented by a reasonable number of actions and states. Tic-tac-toe has 9 squares, each of which can be either an X, and O, or empty. Therefore, there are approximately 3⁹ = 19683 states (and 9 actions, of course). Therefore, we have a table with 19683 x 9 = 177147 cells. This is not small, but it is certainly feasible for tabular q-learning. In fact, we could exploit the fact that the game of tic-tac-toe is unchanged by rotations of the board. Therefore, there are actually far fewer “unique states”, if you consider rotations and reflections of a particular board configuration the same. I won’t get into deep Q-learning, because this is intended to be an introductory project.

First, we initialize our q-table with the aforementioned shape:

Now, let’s set some hyperparameters for training:

Now, we need to set up an exploration strategy. Assuming you understand exploration-exploitation in RL, the exploration strategy is the way we will gradually decrease epsilon (the probability of taking random actions). We need to initially play at least semi-randomly in order to properly explore the environment (the possible tic-tac-toe board configurations). But we cannot forever take random actions, because RL is an iterative process that relies under the assumption that the evaluation of future reward gets better over time. If we simply played random games forever, we would be trying to associate a random list of actions with some final game result that has no actual dependency upon any particular action we took.

Now, let’s create a graph of epsilon vs. episodes (number of games simulated) with matplotlib, saving the figure to an image file:

When we start to simulate games, we need to set some restrictions so that the agents can’t make insensible moves. In tic-tac-toe, occupied squares are no longer available, so we need a function to return the legal moves, given a board configuration. We will be representing our board by a 3x3 NumPy array, where unoccupied squares are 0, X’s are 1, and O’s are -1. We can use NumPy’s np.argwhereto retrieve the indices of the 0 elements.

We also need a helper function to convert between a 3x3 board representation and an integer state. We’re storing the future reward estimations in a q-table, so we need to be able to index any particular board configuration with ease. My algorithm for converting between the board in the format I previously described works by partitioning the total number of possible states into a number of sections corresponding to the number of actions. For each cell in the board:

* 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.

Finally, we need one last helper function to determine when the game has reached a terminal state. This function also needs to return the result of the game if it is indeed over. My implementation checks the rows, columns, and diagonals for a series of either 3 consecutive 1's or 3 consecutive -1’s by taking the sum of the board array across each axis. This produces 3 sums, one for each axis or column. If -3 is one of these sums, this axis must have all -1’s, indicating that the player corresponding to -1 won and vice versa. The diagonals work just the same, except there are only 2 diagonals, while there are 3 rows and 3 columns. My original implementation is a bit naive, I found a much better one online. It’s much shorter and improves speed slightly.

Now, let’s initialize some lists to record training metrics.

past\_results will store the results of each simulated game, with 0 representing a tie, 1 indicating that the player corresponding to the positive integer won, and vice versa with -1.

win\_probs will store a list of percentages, updated after each episode. Each value tells the fraction of games up to the current episode in which either player has won. draw\_probs also records percentages, but corresponding to the fraction of games in which a draw occurred.

After training, if we were to graph win\_probs and draw\_probs, they should demonstrate the following behavior.

1. Early in training, the win probability will be high, while the draw probability will be low. This is because when both opponents are taking random actions in a game like tic-tac-toe, there will more often be wins than draws simply due to the existence of a larger number of win states than draw states.
2. Mid-way through training, when the agent begins to play according to its table’s policy, the win and draw probabilities will fluctuate with symmetry across the 50% line. Once the agent starts playing competitively against itself, it will encounter more draws, as both sides are playing according to the same strategic policy. Each time the agent discovers a new offensive strategy, there will be a fluctuation in the graph, for the agent is able to trick its opponent (itself) for a short period of time.
3. After fluctuating for a while, draw probabilities should approach 100%. If the agent was truly playing optimally against itself, it would always encounter a draw, for it is attempting to maximize reward according to a table of expected future rewards… the same table being used by the opponent (itself).

Let’s write the training script. For each episode, we begin at a non-terminal state: an empty 3x3 board filled with 0’s. It each move, with some probability epsilon, the agent takes a random action from the list of available squares. Otherwise, it looks up the row of the q-table corresponding to the current state and selects the action which maximizes the expected future reward. The integer representation of the new board state is computed, and we record the pair (s, a, s’). Once this game ends, we will need to correlate the state-action pair we just observed with the final game results (which are yet-to-be-determined). Once the game ends, we refer back to each recorded state-action pair, and update the corresponding cell of the q-table according to the following:

Image for post

Q-learning update rule

In the above update formula, s is the integer representation of the state, a is the integer representation of the action the agent took at state s , alpha is the learning rate, R(s, a) is the reward (in our case, the end result of the corresponding game in which this pair (s, a) was observed), Q is the q-table, and the statement involving max represents the maximum expected reward for the resulting state. Say the board configuration was:

[[0, 0, 0],  
 [0, 0, 0],  
 [0, 0, 1]]

and we took the action 3, corresponding to the cell at coordinate (1, 0), the resulting state would be:

[[0 0, 0],  
 [-1, 0, 0],  
 [0, 0, 1]]

This part of the update formula is referring to the maximum expected reward for any of the actions we could take from here, according to the policy defined by our current q-table. Therefore, s' is the second state I just described, and a' is all of the actions we could theoretically take from this state (0–8), although in reality, some are illegal (but this is irrelevant).

At the end of every 1000 episodes, I just save the list of training metrics, and a plot of these metrics. At the end, I save the q-table and the lists storing these training metrics.

**Results**

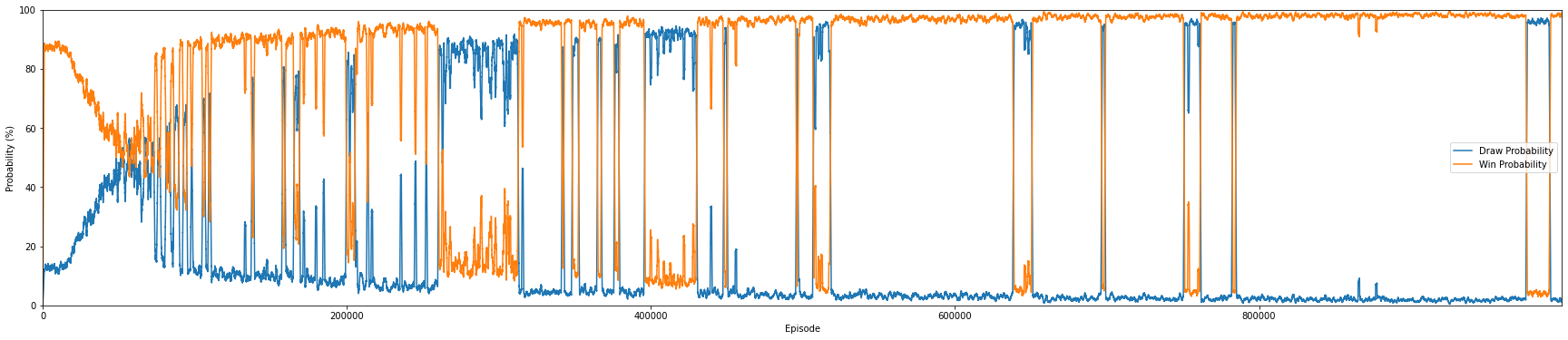
I trained mine with Google Colab’s online GPU, but you can train yours locally if you’d like; you don’t necessarily have to train all the way to convergence to see great results.

Just as I previously mentioned, the relationship between games terminating in a win/loss and those terminating in a draw should work as follows:

* Earlier in training, an unskilled, randomly-playing agent will frequently encounter win-loss scenarios.
* Each time the agent discovers a new strategy, there will be fluctuations.
* Towards the end of training, near convergence, the agent will almost always encounter a draw, as it is playing optimally against itself.

Therefore, the larger fluctuations in the graph indicate moments when the agent learned to evaluate a particular board configuration very well, and in doing so this allowed it to prevent draws.

We can see this is clearly demonstrated in the resulting graph:

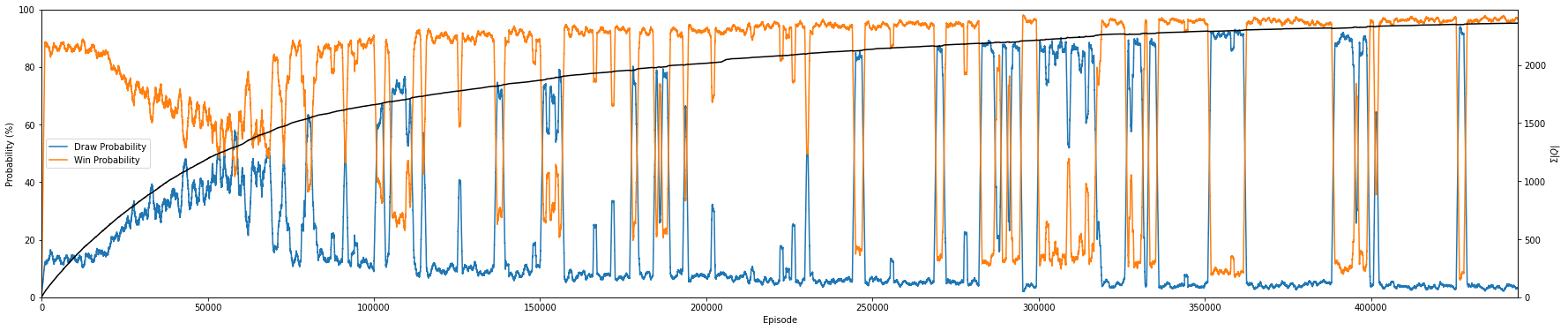


Win/Loss-Draw Ratio Throughout Training

Throughout the middle of training, it frequently appears as if the q-table will converge, only to quickly change entirely. These are the aforementioned moments when a significant strategy was exploited for the first time.

Also, as you can see, the fluctuations occur more rarely as you progress throughout training. This is due to the fact that there are less yet-to-be-discovered tactics as your progress. Theoretically, if the agent converged, there would never be any more great fluctuations like this. Draws would occur 100% of the time and after the rapid rise in the draw percentage, it would not fall back down again.

I decided it would be a good idea to visualize the change in the q-values over time, so I retrained it while recording the sum of the absolute values of the Q table at each episode. Whether a particular q-value is positive or negative recording the sum of all absolute q-values shows us when convergence is occurring (the gradient of the q-values over time decreases as we reach convergence).



Win/Loss-Draw Ratio Throughout Training + Sum of Absolute Value of Q-table throughout Training

You can visit the full code on Google Colab here:

https://colab.research.google.com/drive/1w3RYXZ\_tg80qNDQZf1I2KyZcigwz8Not?usp=sharing

Or on GitHub here:

https://gist.github.com/Ryan-Rudes/96b82cceaa8eb362650304856574a35b#file-an-introductory-reinforcement-learning-project-learning-tic-tac-toe-via-self-play-py

Experimenting with the exploration strategy will influence training. You can change parameters relating to epsilon, as well as how it is decayed in order to get different results.

One final thing to note is that Tic-Tac-Toe can be approached much more easily with simpler value iteration methods because the transition matrix is a given, as well as the reward matrix. This sort of epsilon-greedy optimization is really unnecessary for an environment like Tic-Tac-Toe.

# Original Tic-Tac-Toe Reinforcement Learning Code by Ryan Rudes

#Source:

#https://towardsdatascience.com/an-introductory-reinforcement-learning-project-learning-tic-tac-toe-via-self-play-tabular-b8b845e18fe

import numpy as np

import matplotlib.pyplot as plt

import random

# Initialize q-table of shape (states, actions), which is (3 \*\* 9, 9)

q\_table = np.zeros((3 \*\* 9, 9))

# Learning parameters

episodes = 300000

learning\_rate = 0.01

num\_random\_episodes = 1000

min\_epsilon, max\_epsilon = 0.01, 1.0

# Epsilon decay function

def get\_epsilon(episode):

  epsilon = max(min\_epsilon, min(max\_epsilon, num\_random\_episodes / (episode + 1)))

  return epsilon

# Show graph of exploration strategy

X = np.arange(0, episodes)

Y = np.vectorize(get\_epsilon)(X)

fig = plt.figure(figsize = (6, 6))

plt.plot(X, Y)

plt.xlim(0, episodes - 1)

plt.ylim(0, 1)

plt.xlabel("Episode")

plt.ylabel("Epsilon")

plt.title("Exploration Strategy")

plt.close(fig)

fig.savefig("exploration\_strategy.jpg")

# Get available moves

def get\_available\_moves(board):

  available\_moves = np.argwhere(board == 0).tolist()

  return available\_moves

# Convert board configuration to an integer state

def board\_to\_state(board):

  n\_states = 3 \*\* 9

  state = 0

  for x in board.flatten():

    state += ((x + 1) / 3) \* n\_states

    n\_states /= 3

  return int(state)

# Determine if board state is terminal

def is\_terminal(board):

  """

  If board is terminal, returns 1 or -1 depending on winner, or 0 if tie

  If board is not terminal, returns None

  """

  # Check if any columns are all the same

  column\_sum = np.sum(board, axis = 0)

  if np.any(np.abs(column\_sum) == 3):

    return int(np.sign(column\_sum[np.argmax(np.abs(column\_sum))]))

  # Check if any rows are all the same

  row\_sum = np.sum(board, axis = 1)

  if np.any(np.abs(row\_sum) == 3):

    return int(np.sign(row\_sum[np.argmax(np.abs(row\_sum))]))

  # Check if either diagonal is all the same

  diagonal\_sum = np.array([np.diag(board).sum(), np.diag(np.fliplr(board)).sum()])

  if np.any(np.abs(diagonal\_sum) == 3):

    return int(np.sign(diagonal\_sum[np.argmax(np.abs(diagonal\_sum))]))

  # If no win condition is satisfied, but the board if full, its a tie

  if not np.any(board == 0):

    return 0

  # If the board is not terminal, return None

  return None

# Credit: https://stackoverflow.com/questions/46802651/check-for-winner-in-tic-tac-toe-numpy-python

def is\_terminal(board):

  for turn in [-1, 1]:

    mask = board == turn

    out = mask.all(0).any() | mask.all(1).any()

    out |= np.diag(mask).all() | np.diag(mask[:,::-1]).all()

    if out:

      return turn

  if not np.any(board == 0):

    return 0

  return None

past\_results = []

win\_probs = []

draw\_probs = []

for episode in range(episodes):

  epsilon = get\_epsilon(episode)

  terminal = None

  board = np.zeros((3, 3))

  current\_state = board\_to\_state(board)

  turn = 1

  episode\_memory = []

  while not isinstance(terminal, int):

    available\_moves = get\_available\_moves(board)

    if np.random.random() < epsilon:

      action\_square = tuple(random.choice(available\_moves))

      action = 3 \* action\_square[0] + action\_square[1]

    else:

      actions = q\_table[current\_state] \* turn

      for action in np.argsort(actions)[::-1]:

        action\_square = [action // 3, action % 3]

        if action\_square in available\_moves:

          action\_square = tuple(action\_square)

          break

    board[action\_square] = turn

    new\_state = board\_to\_state(board)

    episode\_memory.append([current\_state, action, new\_state])

    turn \*= -1

    terminal = is\_terminal(board)

    current\_state = new\_state

  reward = terminal

  past\_results.append(reward)

  # Update q-table

  for current\_state, action, new\_state in episode\_memory:

    q\_table[current\_state, action] = q\_table[current\_state, action] + learning\_rate \* (reward + max(q\_table[new\_state]) - q\_table[current\_state, action])

  try:

    averaging\_distance = 1000

    draw\_freq, win\_freq = np.unique(np.abs(past\_results[-averaging\_distance:]), return\_counts = True)[1]

    draw\_prob = draw\_freq / averaging\_distance \* 100

    win\_prob = win\_freq / averaging\_distance \* 100

    draw\_probs.append(draw\_prob)

    win\_probs.append(win\_prob)

  except:

    # If there hasn't been at least one tie and one win, the above code block will raise an exception.

    draw\_probs.append(None)

    win\_probs.append(None)

  if episode % 1000 == 0:

    try:

      fig = plt.figure(figsize = (6, 6))

      plt.plot(draw\_probs, label = "Draw Probability")

      plt.plot(win\_probs, label = "Win Probability")

      plt.xlim(0, len(draw\_probs) - 1)

      plt.ylim(0, 100)

      plt.xlabel("Episode")

      plt.ylabel("Probability (%)")

      plt.legend()

      plt.close(fig)

      fig.savefig("training\_metrics.jpg")

      print ("Episode: {}, Epsilon: {:.3f}, Win Probability: {:.3f}, Draw Probability: {:.3f}".format(episode + 1, epsilon, win\_probs[-1], draw\_probs[-1]))

    except:

      pass

np.save("q\_table.npy", q\_table)

np.save("draw\_probs.npy", draw\_probs)

np.save("win\_probs.npy", win\_probs)