# CSC411: Project 4

Due on Monday, April 3, 2018

**Robert Bazzocchi and Sean Doughty**

April 3, 2018

# Part 1

## Environment

The Environment class provides the structure and basic functionality for the game of Tic-Tac-Toe. This includes resetting and visualizing the state board, checking the game status, and making moves.

## How is the grid represented?

The Tic-Tac-Toe grid is represented by a (9,) numpy array, originally initialized to an array of zeros. Each zero (0) represents a vacant position in the 3x3 grid. As moves are made, each spot is filled with a one (1) or two (2), depending on whether it is player one or player two making the move.

## What do the attributes *turn* and *done* represent?

- *turn* can take on the values {1,2}, representing if it is player 1 ('x') or player 2 ('o') making the current move
- *done* can take on the boolean values {True, False}, representing if the game is done or still in progress, respectively

## Playing a game of Tic-Tac-Toe

A series of calls were made to render() and step() in order to simulate a game of Tic-Tac-Toe. The output of this game is shown below:

Listing 1: Tic-Tac-Toe Example Game

```
. . .
. . .
. . .
=====
x . .
. . .
. . .
=====
x . .
. o .
. . .
=====
x x .
. o .
. . .
=====
x x o
. o .
. . .
=====
x x o
. o .
x . .
=====
x x o
. o o
x . .
=====
x x o
x o o
x . .
=====
Player 1 has won the game.
```

# Part 2

## Policy

### Part 2a: Completing the Policy Class

The Policy class was implemented as a fully-connected neural network with one hidden layer. The two fully-connected layers were used to (i) connect the input layer of size 27 to the hidden layer of size 64, and (ii) connect the hidden layer of size 64 to the output layer of size 9. The forward pass of the data was then defined using a ReLu activation on the first layer and a Softmax activation on the second layer. The completed Policy class is shown below:

Listing 2: Policy Class

```
class Policy(nn.Module):
    """
    The Tic-Tac-Toe Policy
    """
    def __init__(self, input_size=27, hidden_size=64, output_size=9):
        super(Policy, self).__init__()
        self.fc1 = torch.nn.Linear(27, 64)
        self.fc2 = torch.nn.Linear(64, 9)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = self.fc2(x)

        return F.softmax(x,dim=1)
```

### Part 2b: Interpreting the State Vector

The state vector as modified in the *select_action* function is a 27-dimensional vector. Though the tic-tac-toe grid is only 3x3, there are three states that each of these sub-squares can take on:

- 0 (".": vacant)
- 1 ("o": player 1)
- 2 ("x": player 2)

As such, a 27-dimensional vector can be viewed as three concatenated 9-dimensional vectors that represent the full grid for each of the three states. For example, the first 9-dimensional vector corresponds to the value 0 or state "." (a vacant spot). In this vector a "1" indicates an empty sub-square at that index in the grid and a "0" represents all other states.

### Part 2c: Interpreting the Output of Policy

The output of policy is a 9-dimensional vector. As the output of the neural network, it is just passed through a Softmax activation function. Each value in this vector represents the probability of winning the game if the next move is made at that index in the grid. As expected, the sum of all the values in this output vector is equal to 1 and the index corresponding to the highest probability is the best move to be played next. This policy is stochastic.

# Part 3

## Policy Gradient

### Part 3a: Computing Returns

Listing 3: Completed *compute_returns* Function

```
def compute_returns(rewards, gamma=1.0):
    """
    Compute returns for each time step, given the rewards
      @param rewards: list of floats, where rewards[t] is the reward
                      obtained at time step t
      @param gamma: the discount factor
      @returns list of floats representing the episode's returns
          G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ...
    """
    n = 0
    G = []
    while n < len(rewards):
        G.append(0)
        for i in range(n,len(rewards)):
            G[n] += (gamma**(i-n))*rewards[i]
        n+=1

    return G
```

### Part 3b: Updating Weights

In order to update the weights, it is necessary to pass the full list of *saved_rewards* and *saved_logprobs* to the function *finish_episode*. This means the full game must be played before the weights can be updated. This is due to the fact that updating the weights requires the policy loss calculation $J_{avg}(\theta)$, and the policy loss takes into account every state from the start of the game to the end of the game. Consider equation (1) below:

$$J_{\text{avg}}(\theta) = \sum_{s} d^{\pi\theta}(s) V^{\pi\theta}(s) \tag{1}$$

Here, $d^{\pi\theta}(s)$ is the probability of the game being in state $s$ if we follow the $\pi\theta$ policy for a long time. $V^{\pi\theta}(s)$ is the total return we expect at the terminating state, given that we start from state $s$. We sum over all states to yield the loss obtained over the entire game. Doing this allows us to adjust our policy and maximize $J_{avg}(\theta)$. Updating in the middle of an episode would not take into account the final result of the game (i.e., the reward of winning, losing, or tying), which would impact this value significantly. Thus, it is necessary to updating the weights at the end of each episode.

# Part 4

## Rewards

### Part 4a: Modified *get_reward* Function

```
Environment.STATUS_VALID_MOVE  :    0,
Environment.STATUS_INVALID_MOVE: -150,
Environment.STATUS_WIN         :  100,
Environment.STATUS_TIE         :    0,
Environment.STATUS_LOSE        : -200
```

### Part 4b: Updating Weights

The parameter values shown above were adjusted several times with the intent of maximizing average return and minimizing the number of invalid moves made over 1000 episodes. The values consist of zero, positive, and negative rewards.

The *invalid move* status and the *lose* status were given the largest negative awards, such that the *invalid move* reward was slightly less. This ensured that the neural network would optimize such that the number of invalid moves and losses was minimized.

The *win* status was given a large positive reward equivalent in magnitude to that of the *lose* status. This also felt appropriate due to the fact that a win and loss should have an equivalent effect on adjusting the policy gradient (just in different directions).

The *tie* status was given a value of zero, as it was empirically found that the best results were obtained when the model was indifferent about tying the game. Similarly, the *valid move* status was given a value of zero as the large negative reward from the *invalid move* status already accounts for teaching the model to make valid moves.

# Part 5

## Training

### Part 5a: Training Curve

A training curve was plotted with episodes along the x-axis and average return on the y-axis. It was found that most of the "learning" occured from the first episode to the 5000 This plot is depicted in the figure below.

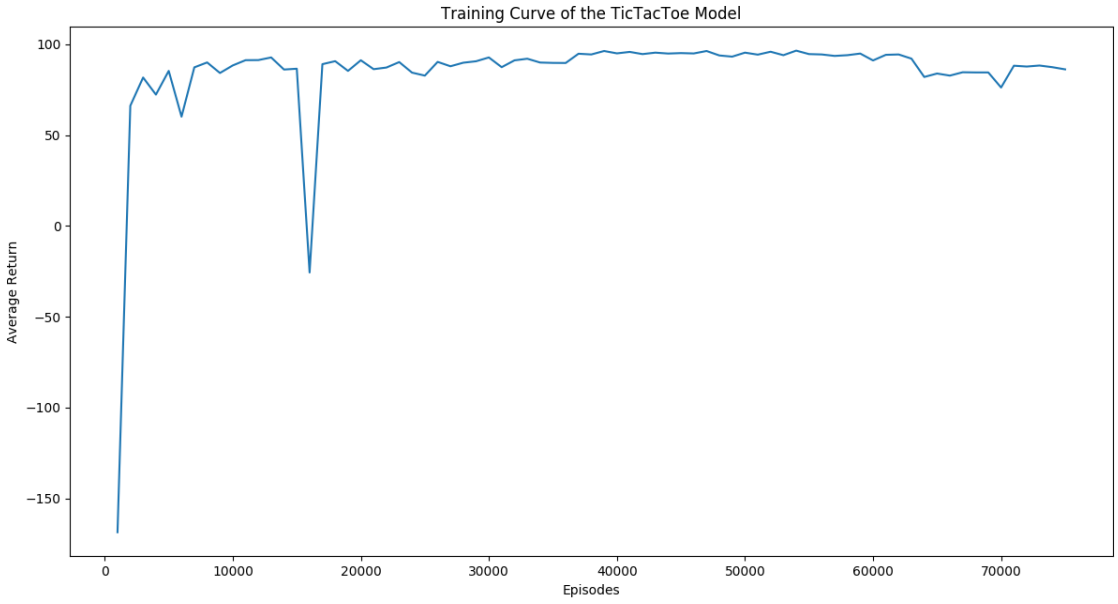


Figure 1: Training Curve with $\gamma = 0.75$

The hyper parameters were set as follows:

- Learning rate: 0.001 (Most stable for few jumps during training)
- Gamma: 0.75 (Based on the study completed in Figure 2
- Activation function: ReLu (Started with ReLu then tried other functions such as Sigmoid, Tanh, Leaky ReLu and found that they had comparable performance)

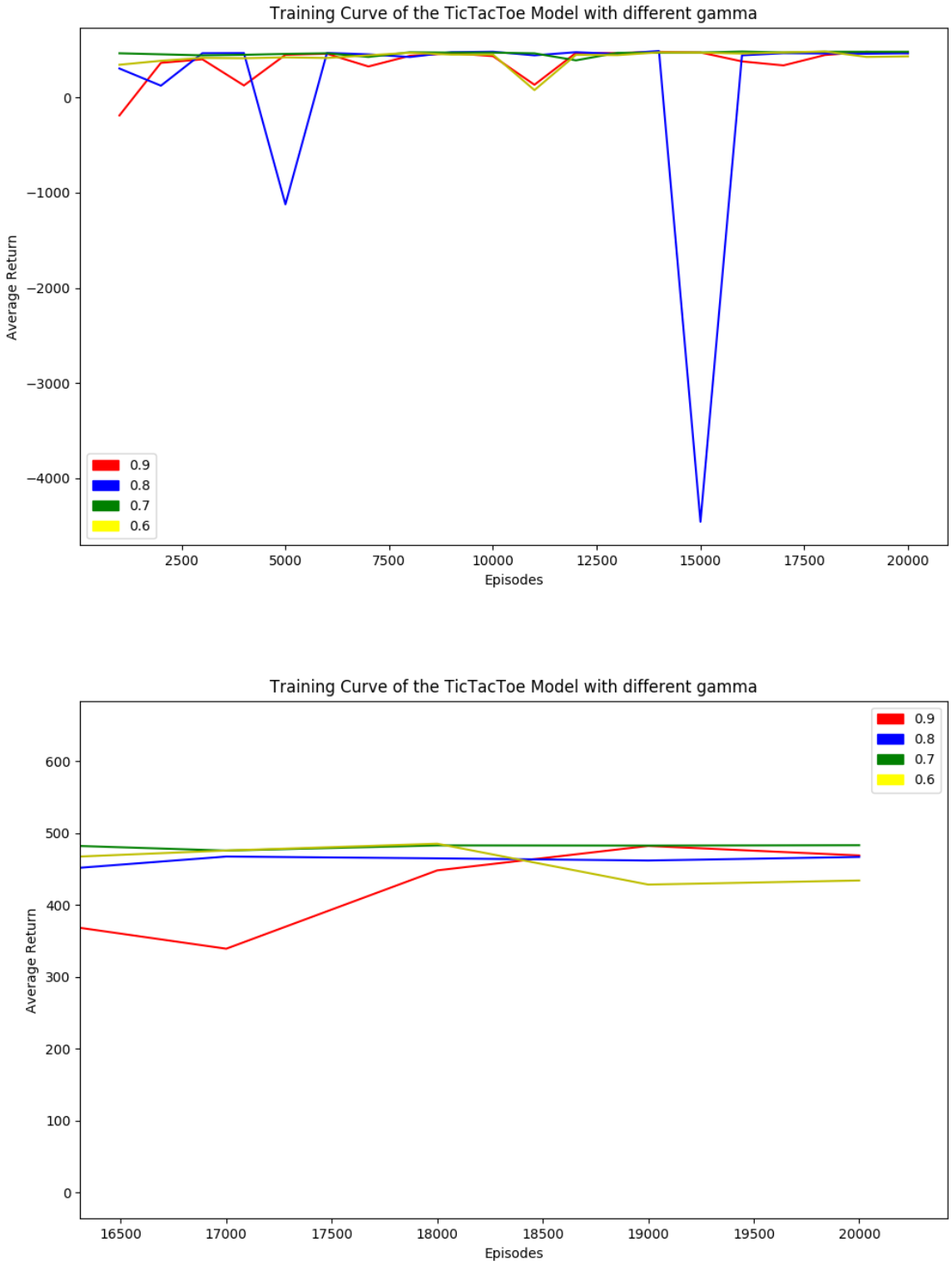


Figure 2: Gamma Sweep

## Part 5b: Optimizing the Number of Hidden Layer Units

The default number of hidden layer units given was 64. Empirical results, however, showed that this value was not ideal for optimizing average return. A sweep was performed on four *hidden_size* values: {10, 100, 500, 1000}. The graph, shown below, indicates that 500 hidden layer units results in the highest average return.


Figure 3: Average Return for Varying *hidden_size* Values

### Part 5c: Learning to Play Only Valid Moves

The rewards from Part 4 were effective in teaching the agent to only play valid moves. After around 10000 episodes the agent would rarely play an invalid move as shown in Figure 4

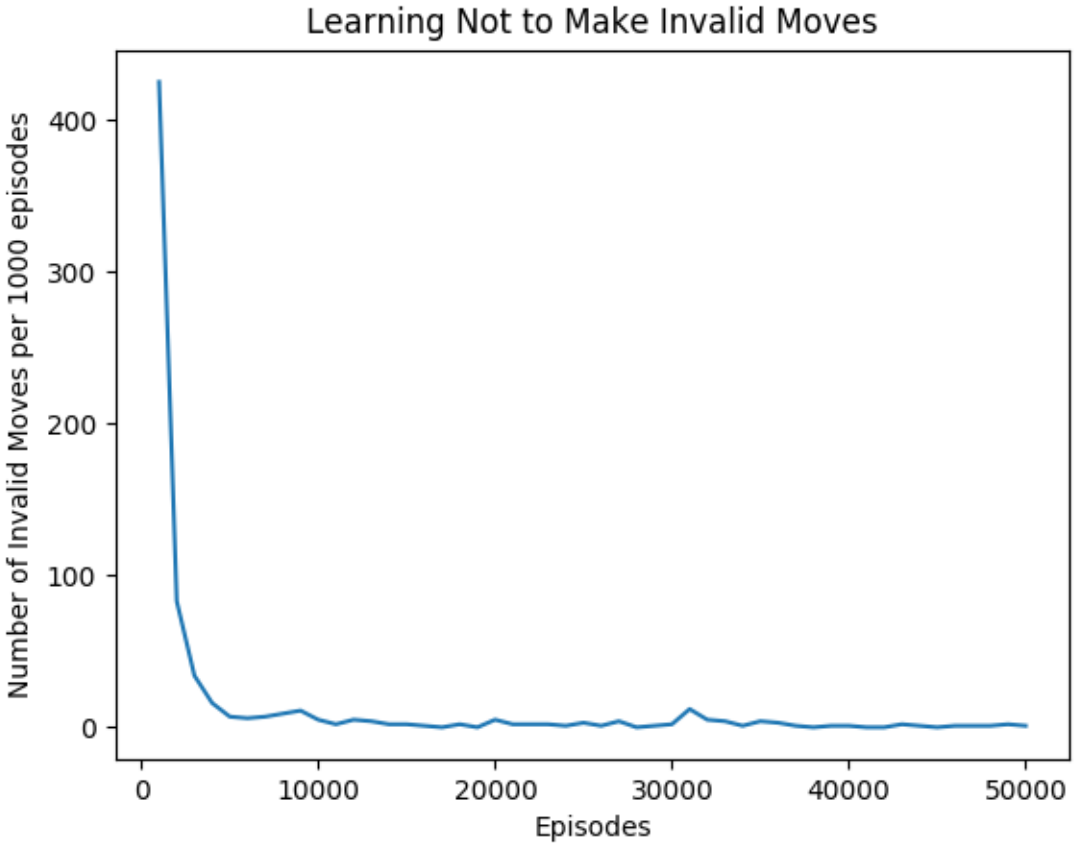


Figure 4: Average number of invalid moves over the last 1000 episodes

### Part 5d: Testing Policy on 100 Games

After all the tuning was complete the Policy was run on 100 test games the results were as follows after 50000 episodes of training.

- Wins: 97
- Losses: 2
- Ties: 1
- Invalid Moves: 0

Here is an example of several games with different results

```
WIN DISPLAYED
...
...
...
====
...
.x.
...
====
...
.x.
o..
====
x..
.x.
o..
====
x.o
.x.
o..
====
x.o
.x.
o.x
====
```

```
TIE DISPLAYED
...
...
...
====
...
x..
...
====
...
x.o
...
====
...
x.o
x..
====
o..
x.o
x..
====
o..
xxo
x..
====
o..
xxo
xo.
====
o..
xxo
xox
====
o.o
xxo
xox
====
oxo
xxo
xox
====
```

```
LOSS DISPLAYED
...
...
...
====
...
x..
...
====
..o
x..
...
====
..o
xx.
...
====
..o
xxo
...
====
..o
xxo
x..
====
o.o
xxo
x..
====
o.o
xxo
x.x
====
ooo
xxo
x.x
====
```

# Part 6

## Win Rate over Episodes

The following figure shows the wins, losses and ties over 1000 episodes over the full training period. It can be observed that while the number of wins approaches 1000 it never fully stops losing games.

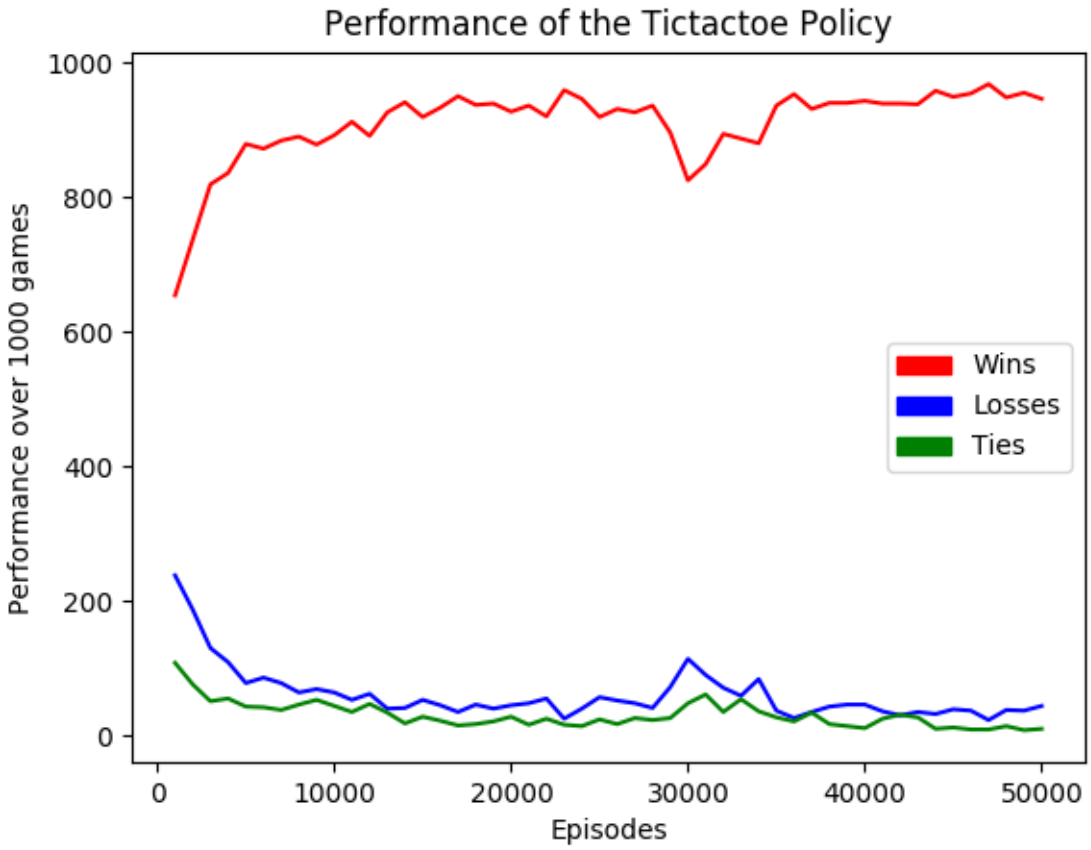


Figure 5: Win Rate over Episodes

# Part 7

## First Move Distribution

The selection of the first move chosen by the model varies throughout training. To show exactly how, nine plots were created showing the first move distributions for each potential first move. These are included below. Each graph plots the probability outputted from policy at the index of that particular move, over all episodes.

| | | |
|---|---|---|
| 0 | 1 | 2 |
| 3 | 4 | 5 |
| 6 | 7 | 8 |

Table 1: Number Position Chart

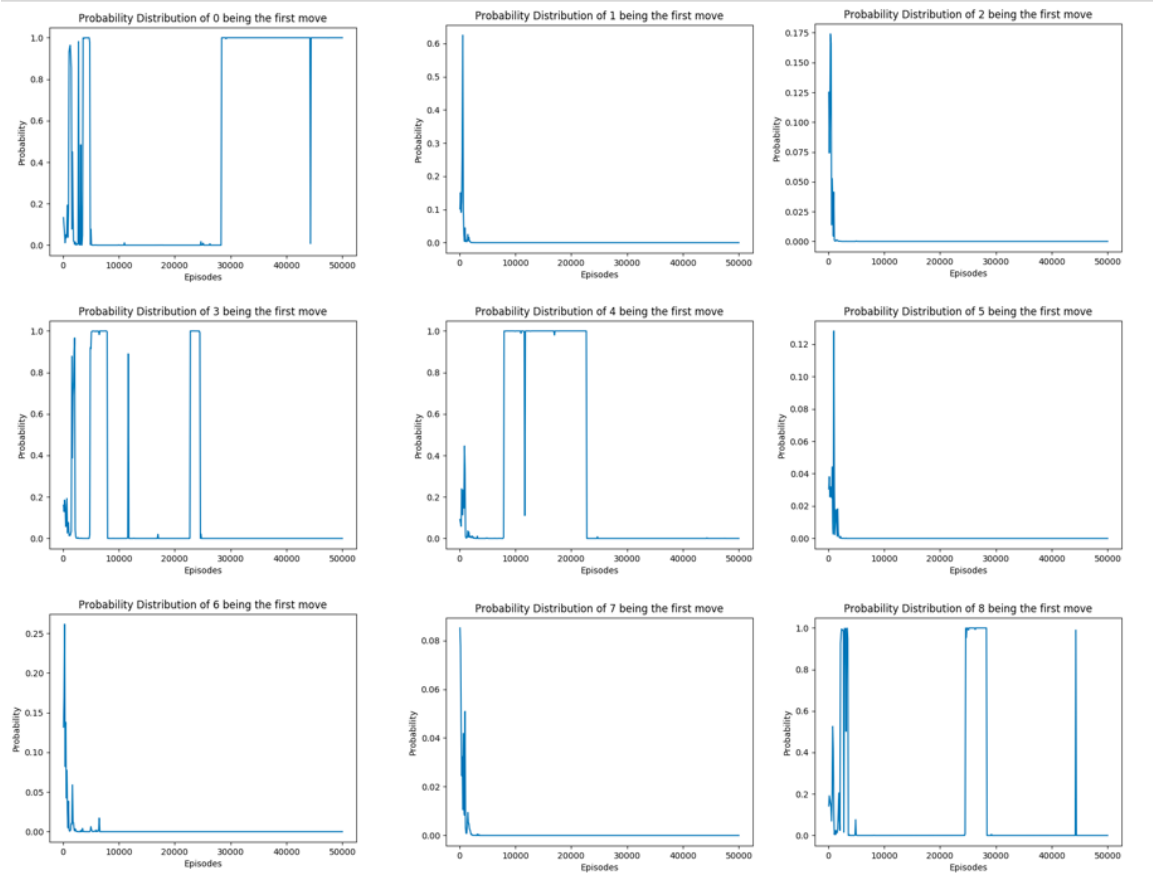


Figure 6: Probability Distributions of Each Starting Move

The final first move distribution is as follows:

Table 2: The First Move Distribution on the Trained Model

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 9.99e01 | 2.45e-17 | 9.25e-22 | 1.08e-10 | 7.47e-08 | 1.66e-17 | 8.61e-11 | 6.37e-19 | 2.70e-06 |

Thus, the trained model has learned that placing the first 'x' in the top-left-most corner (0) would provide it with the highest probability of winning. This was not always the case, however. As can be seen from the graphs above, the model selected 4 (the centre subsquare) as its first move between iterations of 8000 to 24000. To a human player, playing in the centre would seem the smartest move as it has the most number of possible winning lines. The training model may have not found this to be the case after training due to the fact that it was being trained with a random model. As such, the final first move distribution will vary each time training is performed with the same parameters. However, the trained first move is always either the centre or a corner. This makes sense because in these positions there are four and three winning lines, respectively. Whereas, a middle side subsquare has only two winning lines.

# Part 8

Some of the mistakes the agent made are as follows:

- Does not play in the center square. It is intuitive belief that playing the center square is the best first move. However, the policy never seems to learn this principle.
- Does not notice one move to win or one move to lose positions. A basic depth first search would notice these problems instantly however the policy is often unable to notice these problems.