# CSC411: Assignment #4

Due on Monday, April 2, 2018

**Zhiyan Deng, Xin Jie Lee**

April 2, 2018

## Prefix

The code for this project is written in Python 3

## Problem 1

***Part 1**: Tic-Tac-Toe environment*

The Tic-Tac-Toe grid is represented by a 9-dimensional array, each dimension representing a position on the Tic-Tac-Toe board. The attribute turn indicates which player's turn to make a move, and turn = 1 indicates that it is player 1's turn, while turn = 2 indicates it is player 2's turn. The attribute done tracks the completion of a game, and it is set to true at the end of a game and false otherwise. A demonstration of the gameplay against a random game agent is shown below. A new game can be initiated by calling *Environment()*, and our moves are represented by 'x' on the graphical representation of the board or '1' on the grid. Empty board positions are represented by '.' on the graphical board or '0' on the gaming grid. To play a move, we can call *play_against_random(pos)* and specify a position 'pos' on the board where we would like to make a move. Right after our move is made, the computer agent will make a random move that is represented by 'o' on the board or '2' on the grid. During the gameplay, the game's board can be viewed by calling *render()*.

```
>>>env = Environment()
>>>env.render()
...
...
...
====
>>>env.play_against_random(0)
(array([1, 0, 0, 0, 0, 2, 0, 0, 0]), 'valid', False)
>>>env.render()
x..
..o
...
====
>>>env.play_against_random(4)
(array([1, 0, 0, 2, 1, 2, 0, 0, 0]), 'valid', False)
>>>env.render()
x..
oxo
...
====
>>>senv.play_against_random(8)
(array([1, 0, 0, 2, 1, 2, 0, 0, 1]), 'win', True)
>>>env.render()
x..
oxo
..x
====
```

## Problem 2

***Part 2 (a)**: Implement policy*

Policy is a 3 layer neural network with a single hidden layer and its implementation is shown below.

```python
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__()
        # TODO
        self.classifier = nn.Sequential(
                nn.Linear(input_size, hidden_size),
                nn.ReLU(),
                nn.Linear(hidden_size, hidden_size),
                nn.ReLU(),
                nn.Linear(hidden_size, output_size)
        )

    def forward(self, x):
        # TODO
        out = self.classifier(x)
        out = F.sigmoid(out)
        return out
```

***Part 2 (b)****: Representing the board*

Each state that is fed as input to the policy neural network is a one-hot encoding of the game board, and each state is represented by a 27 dimensional vector. The first 9 indicies of the vector keeps track of the positions of the board that are empty, the next 9 indicies keeps trach of the positions of the moves 'x' by player 1, and the final 9 indicies keep track the moves 'o' by player 2. For example:

```
. . x
. 0 .
. . .
```

will be represented by [1,1,0,1,0,1,1,1,1,0,0,1,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0].

***Part 2 (c)****:Output of policy*

The 9-dimensional ouptut of the policy represents the distribution of the probabilities of getting a reward for each of the 9 board positions that our agent can play. The action selected by our agent is stochastic since the agent will be sampling a move based on the probability distribution from the policy output. However positions that are deemed to have higher probabilities of leading to a reward will similarly have a higher chance of being picked. If the corresponding play on a position is good, we will increase the probability at that position at the end of the training episode and vice versa.

## Problem 3

***Part 3 (a)**: Computing returns*

The total discounted rewards received at time t, also referred to as the return at time t, is computed as follows:

$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... + \gamma^{n-1} r_n$

where $r_t$ is the reward received at time t and $\gamma$ is the discount factor. Rewards that are received further away in the future are worth less, and hence they will be discounted to a greater extent. Thus is done to encourage the agent to pursue policies that generate bigger rewards earlier in the game. The function *compute_returns* calculates the total discounted reward received at every time step t after the completion of an episode. Its implementation is shown below.

```python
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} + ...

    >>> compute_returns([0,0,0,1], 1.0)
    [1.0, 1.0, 1.0, 1.0]
    >>> compute_returns([0,0,0,1], 0.9)
    [0.7290000000000001, 0.81, 0.9, 1.0]
    >>> compute_returns([0,-0.5,5,0.5,-10], 0.9)
    [-2.5965000000000003, -2.8850000000000002, -2.6500000000000004, -8.5, -10.0]
    """
    returns = []
    time_step = len(rewards)
    for i in range(time_step):
        curr_ret = 0
        for j in range(i,time_step):
            curr_ret += rewards[j] * gamma**(j-i)
        returns.append(curr_ret)
    return returns
```

***Part 3 (b)**:*

We cannot update the weights in the middle of an episode is since the gradients are not additive through each step within an episode. We must go through the entire episode to learn the total expected return and the probability of being in a certain state to compute the gradients.

## Problem 4

***Part 4 (a)**: Defining rewards*

The code below shows the implemented rewards that will be used to train our agent.

```python
def get_reward(status):
    """Returns a numeric given an environment status."""
    return {
            Environment.STATUS_VALID_MOVE  : 0,
            Environment.STATUS_INVALID_MOVE: -1,
            Environment.STATUS_WIN         : 1,
            Environment.STATUS_TIE         : -0.5,
            Environment.STATUS_LOSE        : -1
    }[status]
```

***Part 4 (b)**: Rationales behind reward*

To encourage our agent to seek policies that lead to its victory, positive rewards of +1 are assigned whenever the agent wins the game in the end. On the other hand, negative rewards are assigned to games that ended in a loss or when the agent makes an invalid move. This is to discourage our agent from pursuing policies that results in a loss or from making invalid moves. Since we want our agent to place equal importance in not playing invalid moves as well as not loosing the game, we assigned equal rewards of -1 to games that ended with an invalid move being made or in a loss. In addition, we assigned -0.5 rewards for games that ended in a tie as we wanted our agent to avoid policies that ended in a tie, but not avoid it to the same extent of avoiding policies that lead to losses.

## Problem 5

***Part 5 (a)***: *Training curve of model*

The number of hidden units used in our updated policy is 128 since it gave us a relatively good balance between high win-rates and low training times. The training curve for the model is shown below.

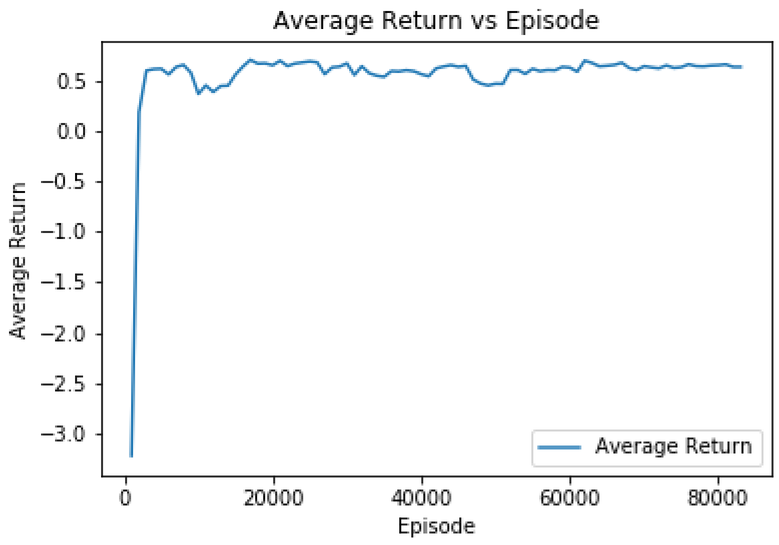

Figure 1: Training curve of model

***Part 5 (b):***

We experimented with several number of hidden units during the training process, notably 64, 128 and 256 hidden units. All models were trained with our selected rewards as shown in part 4 (a). The training curves and win/tie/lose charts of the models are shown below:

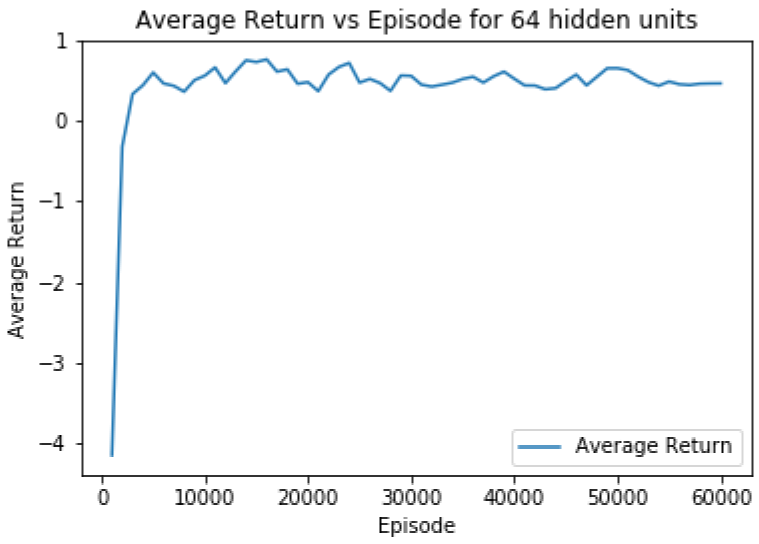

Figure 2: Training curve with 64 hidden units

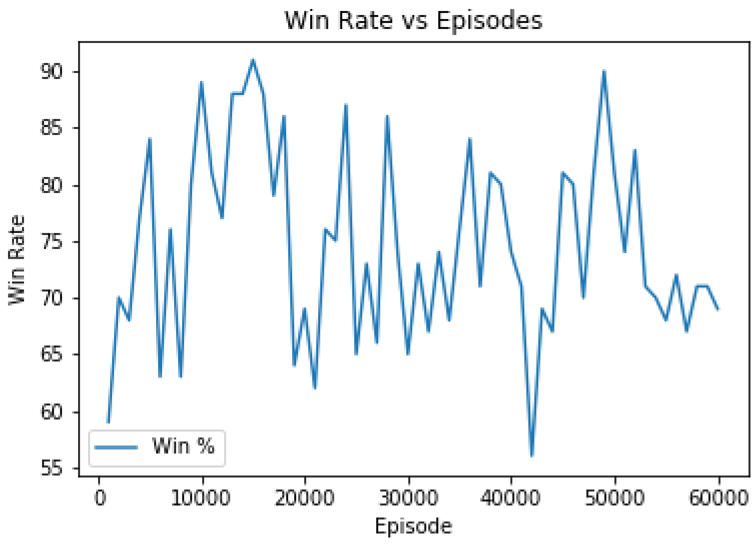

Figure 3: Win rate with 64 hidden units

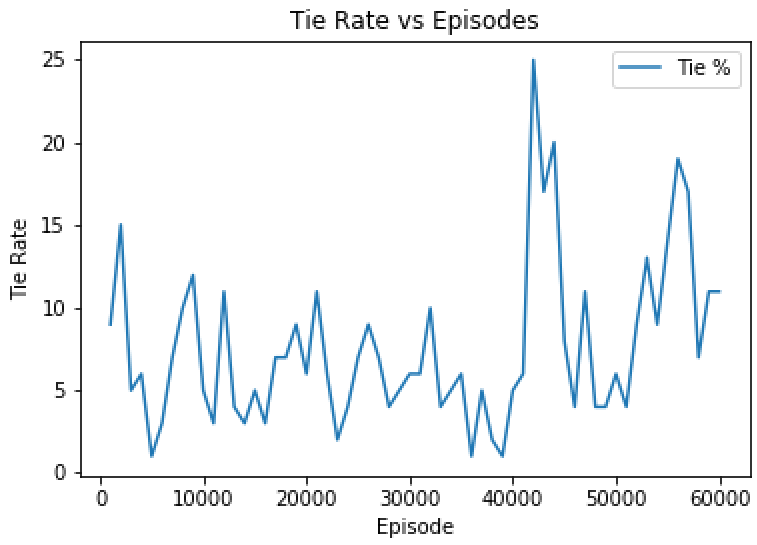

Figure 4: Tie rate with 64 hidden units

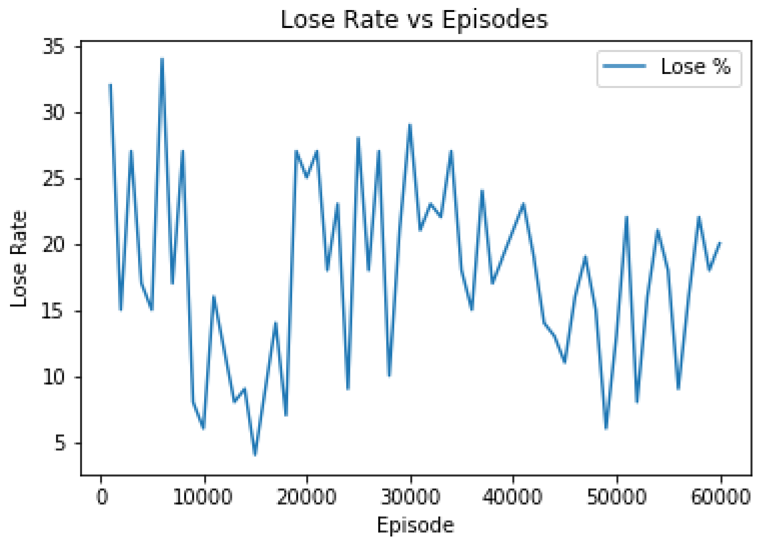

Figure 5: Lose rate with 64 hidden units

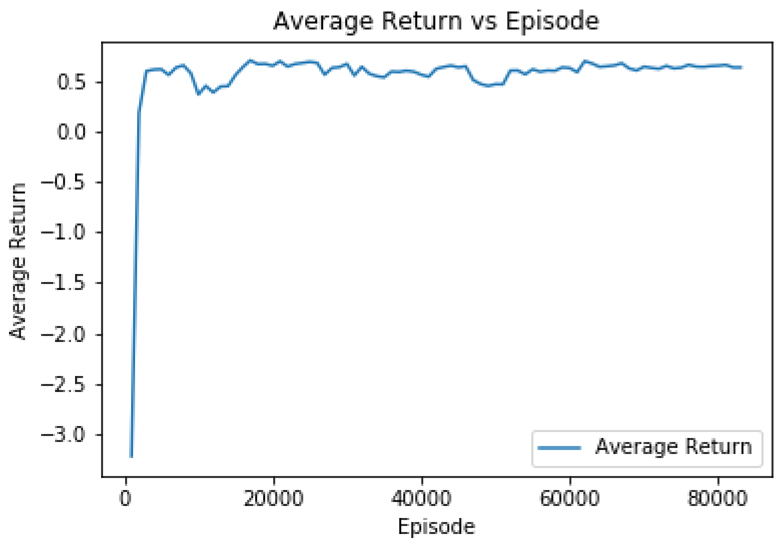

Figure 6: Training curve with 128 hidden units

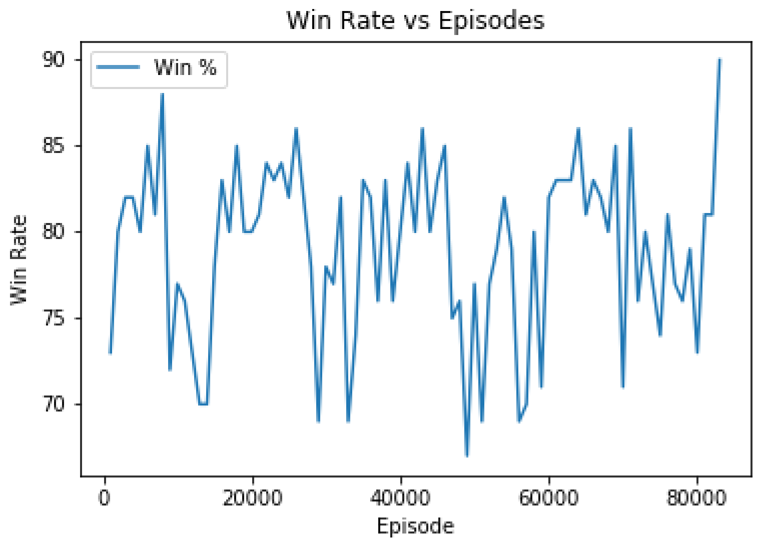

Figure 7: Win rate with 128 hidden units

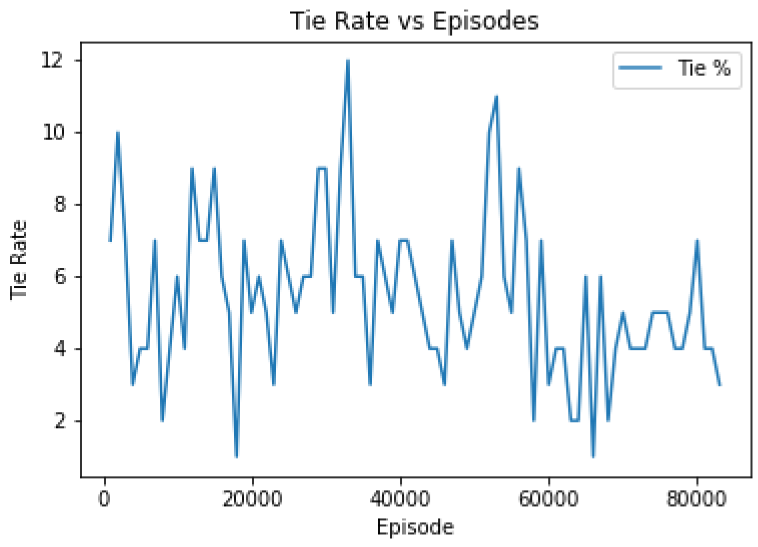

Figure 8: Tie rate with 128 hidden units

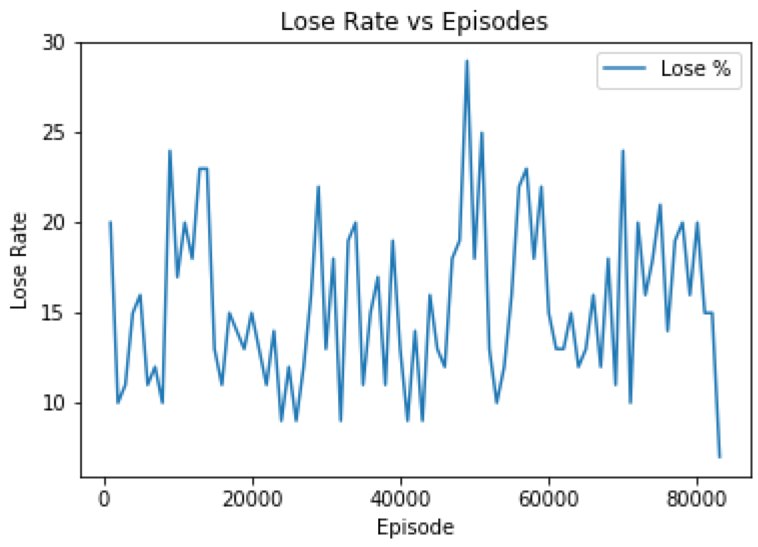

Figure 9: Lose rate with 128 hidden units

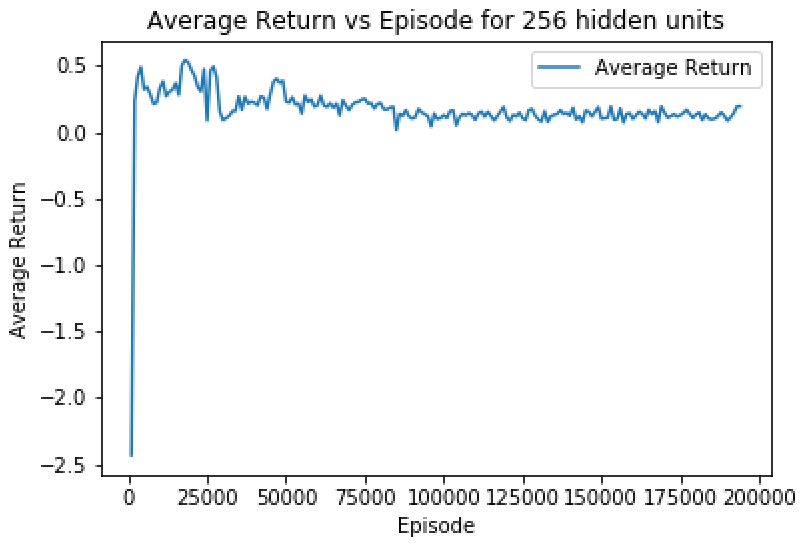

Figure 10: Training curve with 256 hidden units

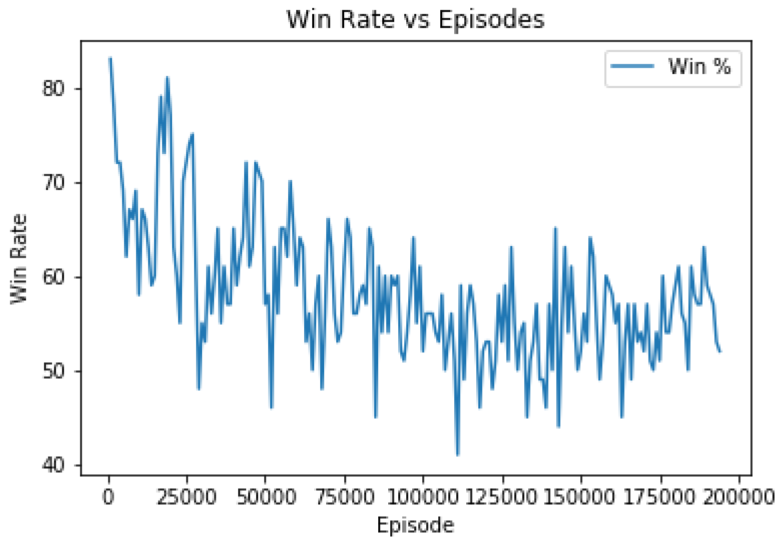

Figure 11: Win rate with 256 hidden units

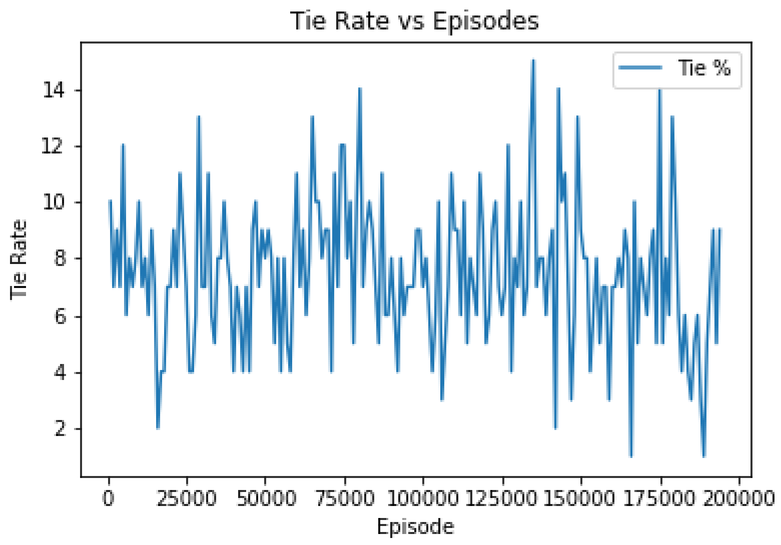

Figure 12: Tie rate with 256 hidden units

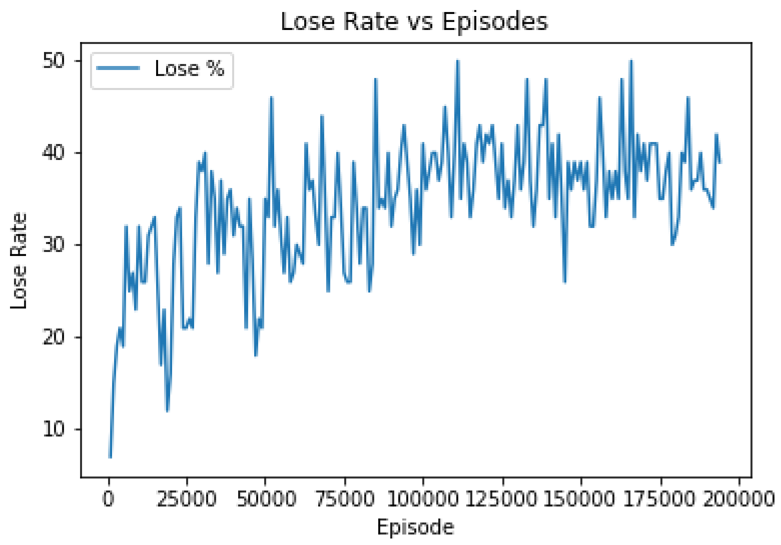

Figure 13: Lose rate with 256 hidden units

The win/tie/lose rates are fairly erratic throughout the training process regardless of the number of hidden units used, although the model with 64 and 128 hidden units seem to achieve higher win rates as compared to the model with 256 hidden units. Hence, a decision was made to go with 128 hidden units since it offered relatively higher win rates and faster training time as compared to the model with 256 hidden units, and the variance of its win rate across training episodes is narrower as compared to the model with 64 hidden units. Nonetheless, we do not expect significant differences in win rates between the models with 64 and 128 hidden units.

***Part 5 (c)**: Learning to stop playing invalid moves*

Using our chosen network that utilized 128 hidden units, our agent learned to stop playing invalid moves after approximately 70,000 episodes. The top chart depicts the number of invalid moves the agent made at every episodes, and it is evidently clear that the agent commited alot of invalid moves during the first 15,000 episodes of training. After 15,000 episodes, the number of invalid moves being made rapidly declined, with a few minor spikes at around 30,000 and 40,000 episodes. Overall, very few invalid moves are made after 60,000 episodes of training. The bottom chart depicts the episodes in which any invalid moves occur. A value of 1 is recorded if an invalid move is made during an episode and 0 otherwise. From this chart, it is apparent that the agent ceased to make invalid moves after approximately 70,000 episodes.

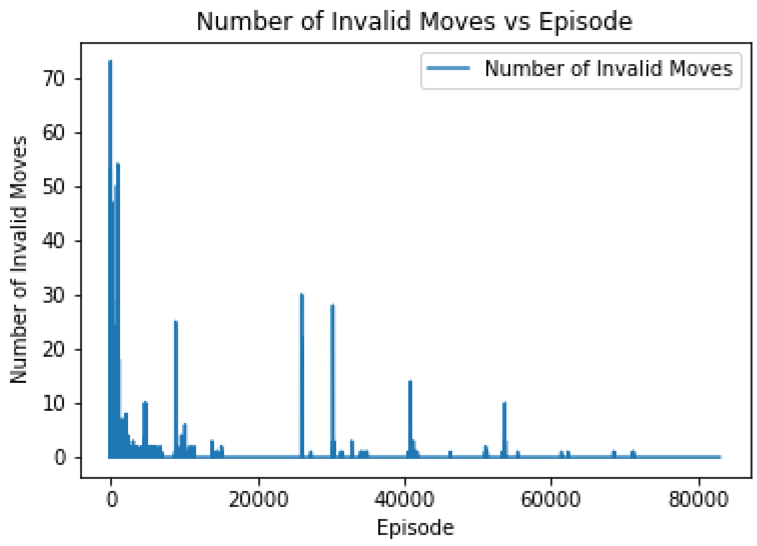

Figure 14: Number of invalid moves made during training

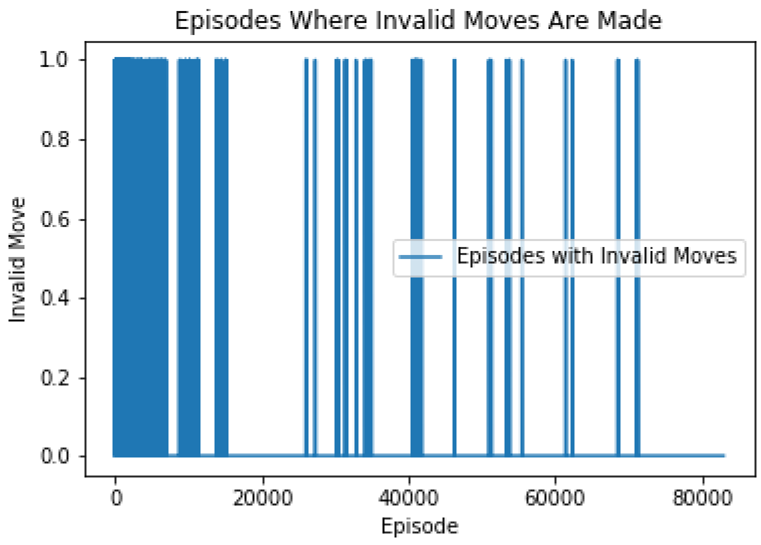

Figure 15: Episodes where invalid moves were made

***Part 5 (d)****: Performance of trained model*

The trained model is tested against a random computer agent in 100 games. Since our trained model is making the first move in all 100 games, we expect its performance to be better than 60% since a random (untrained) agent who gets to go first will tend to win 60% of the time. Undoubtedly, our trained agent performed relatively well as shown by the results below.

```
Win Rate: 83.0%
Tie Rate: 3.0%
Lose Rate: 14.000000000000002%
```

We will now show the first 5 games that our agent played. As a recap, these are the positions on the game board:

```
012
345
678
```

In the first game that our trained agent played, our agent ('x') made a corner play in its first turn at position 8. When the opponent ('o') responded with an edge play at position 5, our agent chose to make a corner play on position 0. A better move for our agent at this point will be to make a center play at position 4 which will have guaranteed its chances of winning since an "intelligient" opponent will be forced to make a play at position 0 to prevent our agent's victory, leaving our agent free to make a move at position 7 which would create a 'fork' and allow our agent to win in the next turn by making a play at positions 1 or 6. Since our agent is training against a random ("less intelligient") opponent, it might be the reason why our agent failed in learning to make these moves. Nonetheless, our trained agent demonstrated that it knew the moves needed to achieve victory and it made plays at positions 0 and 4 in its next two turns to win the game.

```
Start of Game 1
Turn: 1
...
..o
..x
====

Turn: 2
x.o
..o
..x
====

Turn: 3
x.o
.xo
..x
====

Game 1 result: win
```

In its first turn of game 2, our trained agent ('x') made a corner play at position 0 and the opponent responded by making an edge play at position 7. For our agent's second turn, a center play was made on position 4. When the opponent made a move at position 8 to block our agent's move just like an "intelligient" opponent would have, our agent responded with a move at position 6 in its third turn which seem to have guaranteed its victory since a play on either positions 2 or 3 in turn 4 would have secured victory. Perhaps, our trained agent had learned to make "big strategic" plays akin to the one mentioned previously. However, when the opponent made a move at position 1 during its third turn, our agent disappointingly made a play at position 6, rather than at position 3 that would have ended the game in a victory for our agent. Nonetheless our agent still won the game in the end, but this game clearly showed the shortfalls of our trained model.

```
Start of Game 2

Turn: 1
x..
...
.o.
====

Turn: 2
x..
.x.
.oo
====

Turn: 3
xo.
.x.
xoo
====

Turn: 4
xoo
.xx
xoo
====

Turn: 5
xoo
xxx
xoo
====

Game 2 result: win
```

In game 3, our trained agent ('x') once again made a corner play in its first turn, however it was made at position 0 in this game, as compared to position 8 previously. The opponent ('o') responded with a move at position 3, and during our agent's second turn, it made a play on position 8. Similarly to our first game, it is obvious that our agent learned a preference of making moves on directly opposite corners. However, the opponent made a move on position 4, thereby blocking our agent's path to victory. Our agent recognized this situation and immediately changed strategies, choosing to make an edge play on position 7 and going for the bottom 3 positions of the playing board. When the opponent made a move on position 2 for its third move, our agent responded by making the winning play on position 6 in turn 4. Once again, our agent showed it understood how to make winning plays.

```
Start of Game 3

Turn: 1
x.o
...
...
====

Turn: 2
x.o
.o.
..x
====

Turn: 3
xoo
.o.
.xx
====

Turn: 4
xoo
.o.
xxx
====

Game 3 result: win
```

Game 4 clearly shows the limitations of our trained agent. Once again, our agent ('x') opened with a corner play at position 0 of the board. When the opponent ('o') responded with a center play on position 4, our agent made a surprising play on position 8 in its second turn. When the opponent played position 1 on its second turn, our agent failed to block the opponent's potential path to victory with a play at position 7. instead, our agent made a move at position 6 in its third turn. On the other hand, this move will gurantee a victory in our agent's next turn should the opponent not make a move at position 7 during turn 3, since there will be two potential winning plays for our agent in turn 4: 3 and 7. Due to the fact that our opponent is a random policy, it likewise failed to make the winning play by choosing to make a play at position 2 instead. Nevertheless, our agent made the right play at position 7 in its fourth turn, winning the game.

```
Start of Game 4

Turn: 1
x..
.o.
...
```

```
====

Turn: 2
xo.
.o.
..x
====

Turn: 3
xoo
.o.
x.x
====

Turn: 4
xoo
.o.
xxx
====

Game 4 result: win
```

In the fifth game, our agent ('x') made a quick winning diagonal play at positions 8, then 4 and finally 0.

```
Start of Game 5

Turn: 1
...
...
o.x
====

Turn: 2
...
ox.
o.x
====

Turn: 3
x..
ox.
o.x
====

Game 5 result: win
```

From these 5 games, it is apparent that our trained agent has learned to make relatively good plays most of the time. However, our trained agent seems to lack the ability of making "big strategic" plays that will guarantee its victory, as seen in game 1. In addition, our agent occassionally fails to recognise winning plays, as shown in game 2 turn 4. Since our agent is trained against a random policy, it sometimes fails to spot potential winning plays by the opponent and block them, like in game 4 turn 3. Perhaps training our agent against a more "intelligient" non-random policy will have the potential of improving our model's performance and increase its awareness of identifying and blocking opponent's winning moves. This is because a random opponent policy is unlikely to maintain a coherent strategy during gameplay, hence negative rewards do not register during training situations when the opponent is just one move away from victory.

## Problem 6

***Part 6***: *Win/lose/tie rate vs training episodes*

As seen from the top graph, the win rate is fairly erratic throughout the training process, but it remained fairly consistent at the high 70% to low 80% range. The erratic progression of the win rate is perhaps reflective of the policy's attempt to balance exploration and exploitation during training. Similarly, the tie rate appears to be fairly erratic as well throughout training. However, there is a gradual decrease in the tie rate as the number of training episodes increases, since the trained policy is recognizing the negative rewards that arise when tie games occur. Lastly, the lose rate once again exhibit an erratic pattern like the others. Howevever, the lose rate seldom rose above 20%, perhaps indicating that the trained policy is actively avoiding policies that will lead to negative rewards.

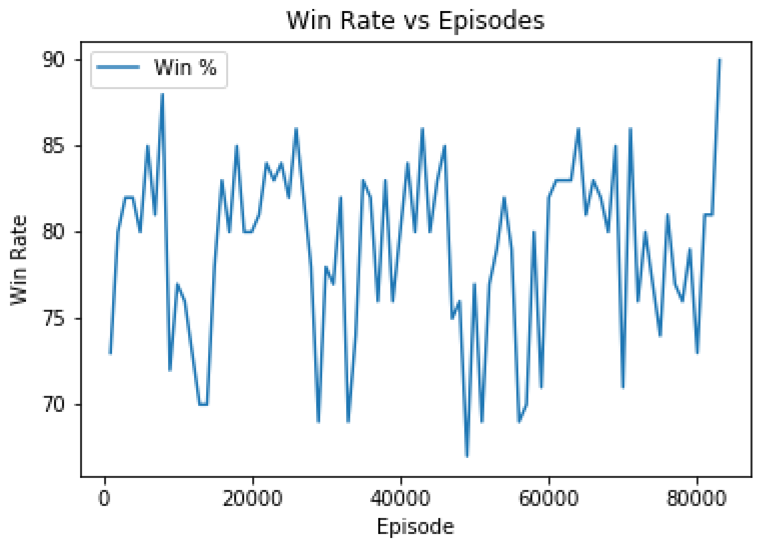

Figure 16: Win rate vs number of training episodes

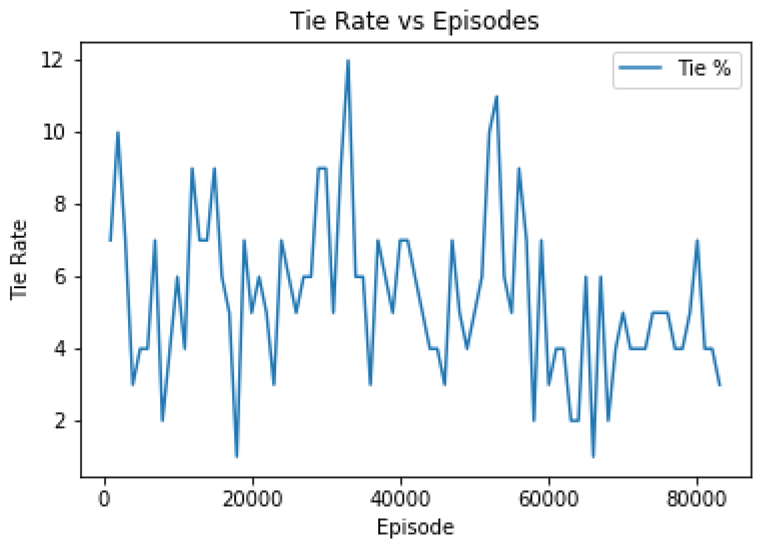

Figure 17: Tie rate vs number of training episodes

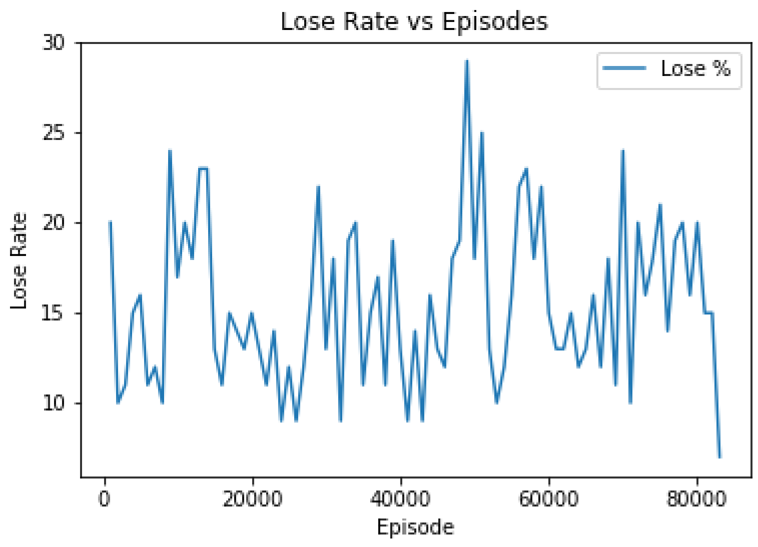

Figure 18: Lose rate vs number of training episodes

## Problem 7

***Part 7**: First move distribution over episodes*

As a recap, these are the positions on the game board:

012
345
678

The following charts depict the log probabilities of the trained policy making the first move on a certain position on the board. It is immediately apparent that as training progresses, the trained policy increasingly favours the positions 0, 4 and 8. This preference is clearly evident in the 5 games shown in problem 6, with our trained policy starting at positions 0, 4, 8 in all occasions. Perhaps this is indicative of the policy's perference of a left diagonal winning strategy, reminiscent of game 5.

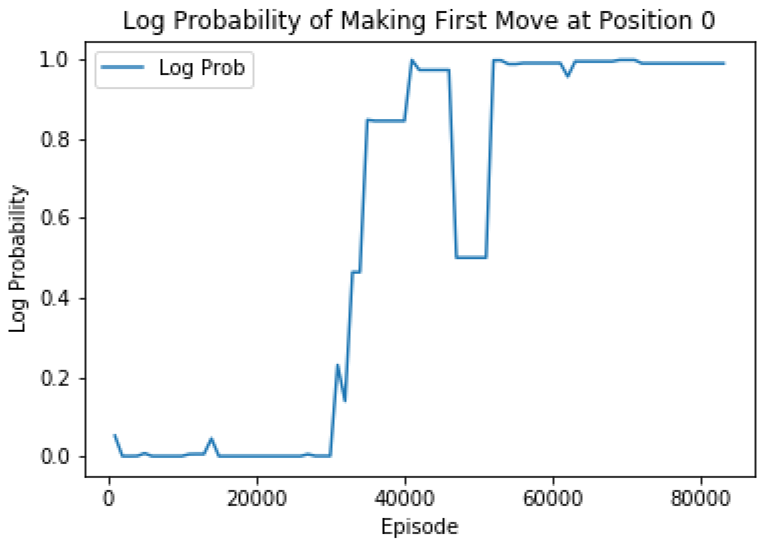

Figure 19: Log Probability of Making First Move at Position 0

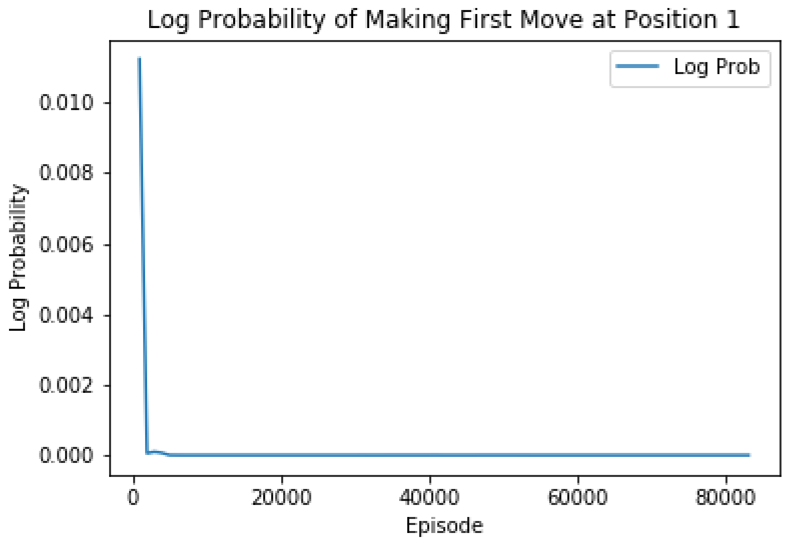

Figure 20: Log Probability of Making First Move at Position 1

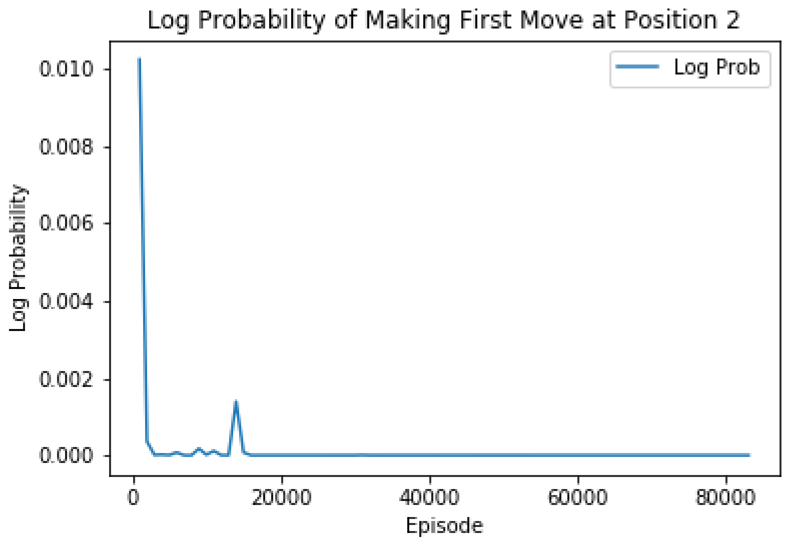

Figure 21: Log Probability of Making First Move at Position 2

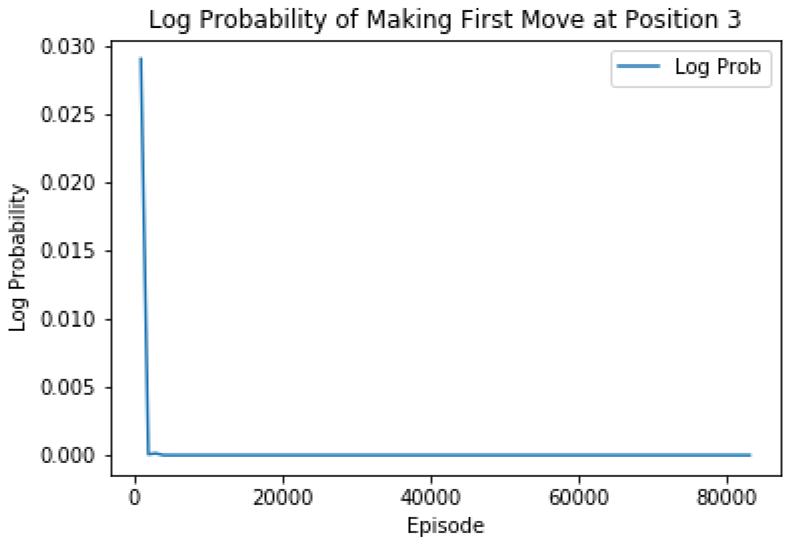

Figure 22: Log Probability of Making First Move at Position 3

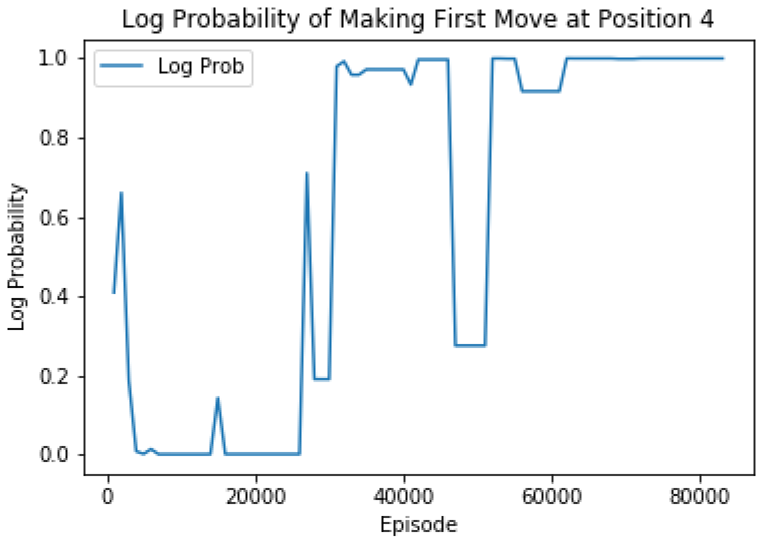

Figure 23: Log Probability of Making First Move at Position 4

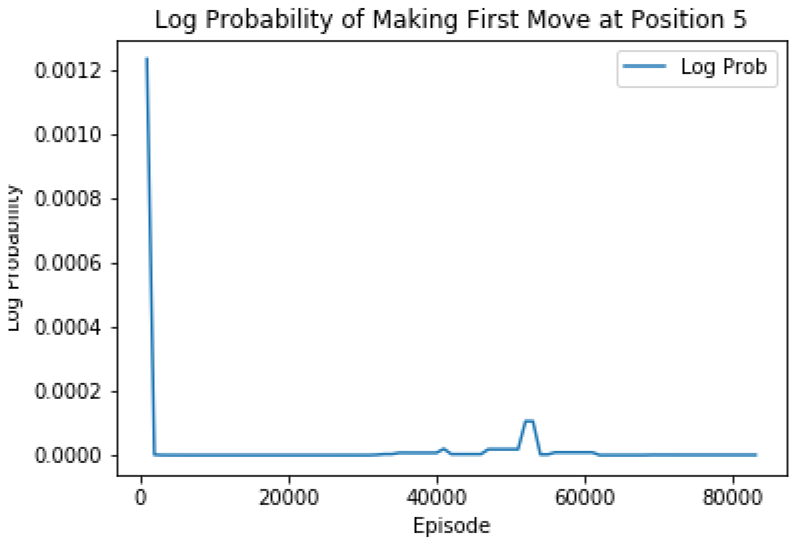

Figure 24: Log Probability of Making First Move at Position 5

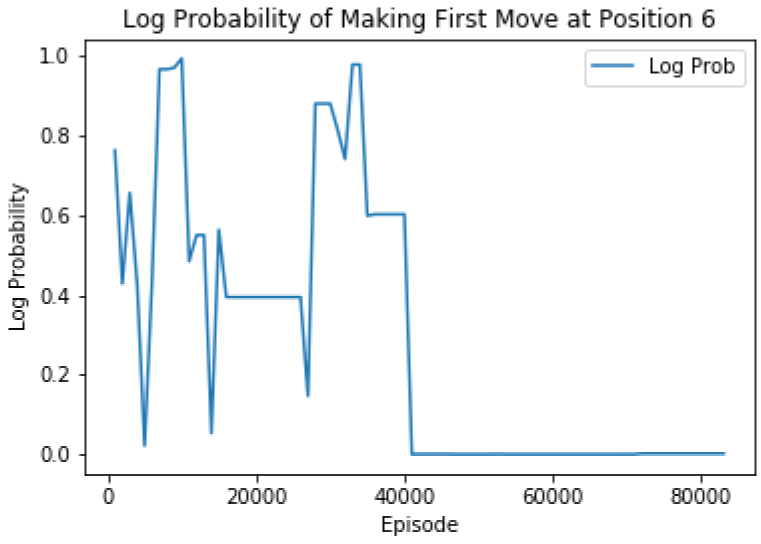

Figure 25: Log Probability of Making First Move at Position 6

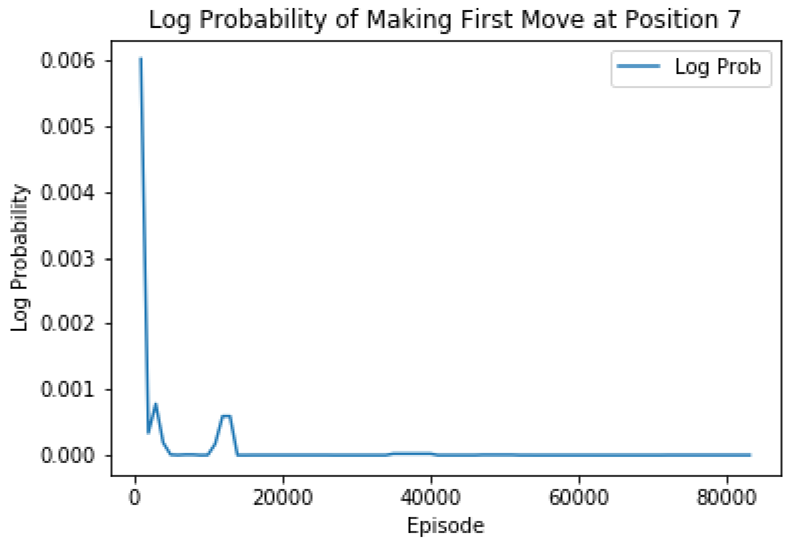

Figure 26: Log Probability of Making First Move at Position 7

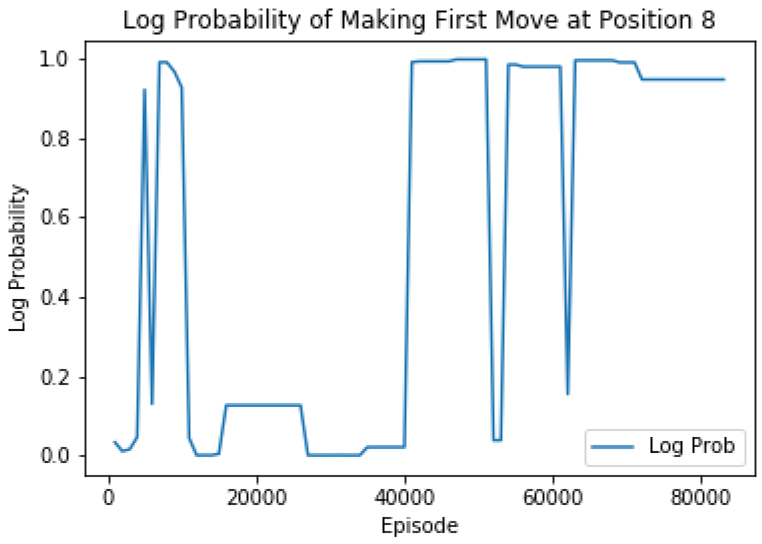

Figure 27: Log Probability of Making First Move at Position 8

## Problem 8

***Part 8****: Limitations of the trained policy*

As observed from gameplay samples of our trained policy in part 6, one glaring problem of our trained policy is that it is unable to recognize and block potential winning plays from its opponent. Since our policy is trained against a random policy, it is unlikely to encounter any coherent strategies from its opponents. Hence, our agent often fails to recognize situations such as when the opponent has 2 connected 'o' marks in a row and is just one 'o' mark away from winning. Perhaps training our model against a more intelligient opponent, such as itself, will improve our model's performance in this regard. Furthermore, there were occasions when our model failed to make the immediate winning play, such as in turn 4 of game 2. Improvements can certainly be made with regards to training our model. Training against a random opponent that starts first could be a simple solution. Alternatively, our policy can be trained against a trained version of itself, so that it will be exposed to opponents capable of maintaining some form of coherent strategies.