CSC2515: Assignment 4
Tic-Tae-Toe with Policy Gradient

Due on Friday, April 2, 2018

M.Wong, S.Pyda

March 31, 2018

Problem 0

Introductory information and readme instructions

This submissions containts the required files: *tictactoe.py, tictactoe.tex &tictactoe.pdf.* The starter code *tictactoe.py* provided in the project description is used. Code was written in Python 3.5 with the Anaconda environment. PyTorch 0.3+ is used.

Problem 1

Environment Description

The starter code *tictactoe.py* provides an *Environment* class, which is a framework for the Tic-Tac-Toe game. The tictactoe grid is initalized at each new game with *self.grid=np.array([0]*9)*; this is an numpy array of length 9 representing the flattened 3 by 3 grid. The attribute *turn* represents whose turn it is during the game (e.g. Player 1 vs. Player 2); when the grid is rendered 1 represents 'x' and 2 represents 'o'. After a valid play that does not end the game, the players change from 1 to 2 or vice versa.

The frozenset *win_set* contains 8 tuples of length 3. Each typle contains a winning combination of plays (indices on the grid). If the same label (except for 0) is contained at all three indices, that player wins the game. This is checked in *check_win(self)*. The attribute *done* determines if the current game is finished. The game is finished if a player wins or if the grid is full - if the game is finished *self.done=True*.

The *step()* function provides a method for a user/agent to input a move into the gameboard, taking in *action* as it's parameter (e.g. where to place an X/O on the board, represented by a number from 0-9). The *render()* method shows the current state of the board. The following code was written to test both functions:

```
if __name__ == '__main__':
    import sys
    policy = Policy()
    env = Environment()
    if len(sys.argv) == 1:
        # `python tictactoe.py` to train the agent
        train(policy, env)
    else:
        # `python tictactoe.py <ep>` to print the first move distribution
        # using weight checkpoint at episode int(<ep>)
        #ep = int(sys.argv[1])
        #load_weights(policy, ep)
        # print(first_move_distr(policy, env))
        print(env.step(action=4))
        print(env.step(action=1))
        print(env.render())
```

The following output was obtained from the execution of the above code.

```
(array([0, 0, 0, 0, 1, 0, 0, 0, 0]), 'valid', False)
(array([0, 2, 0, 0, 1, 0, 0, 0, 0]), 'valid', False)
.o.
.x.
...
```

Problem 2

Policy

(A) The implementation for the *TO-DO* sections in the provided Python file is listed below. The neural network is a fully connected, one hidden layer neural network.

```
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.features = nn.Sequential(
            nn.Linear(input_size, hidden_size),
            nn.ReLU(inplace=True),
            nn.Linear(hidden_size, output_size)
        )

    def forward(self, x):
        x = self.features(x)
        output = torch.nn.functional.softmax(x, dim=1)
        return output
```

(B) For a given position on the board, the select_action function will return the probabilities that the agent should take with respect to the available moves - in this case, either X, O or no action. As there are 9 total positions on the board, therefore, there are $3x3x3 = 27$ total vector dimensions (e.g. 3 moves per position $\times$ 3 positions per row $\times$ 3 rows per board).

(C) The values in each of the 9-output dimensions refer to the probabilities of the action that the agent will take - the agent will pick the action status based on the highest value obtained from the neural network. The policy is stochastic as the agent will pick a move based on the current state of the board.

Problem 3

Policy Gradient

(A) The implementation for the *compute_returns* function is located below.

```
def compute_returns(rewards, gamma=1.0):
    returns = []
    for index, reward in enumerate(rewards):
        i = index
        exponent = 1
        if i == len(rewards)-1:
            returns.append(reward)
        else:
            while i < len(rewards)-1:
                reward += (gamma**exponent)*(rewards[i+1])
                exponent += 1
                i += 1
            returns.append(reward)
    return returns
```

(B) The backwards pass to update weights is computed when the entire episode is complete. The weights cannot be updated in the middle of an episode because the final state (win/lose/tie) which determines the reward is not known until the end of the episode.

Problem 4

Rewards

(A) The *get_reward* function was modified to the following:

```
def get_reward(status):
    """Returns a numeric given an environment status."""
    return {
          Environment.STATUS_VALID_MOVE : 0, # TODO
          Environment.STATUS_INVALID_MOVE: -100,
          Environment.STATUS_WIN : 10,
          Environment.STATUS_TIE : 0,
          Environment.STATUS_LOSE : 0
    }[status]
```

(B) A large penalty (-100) was chosen to signficantly penalize invalid moves to ensure that the agent learned to stop making invalid moves earlier and consider only valid moves. Per the instructions, and in order for the agent to play the game 'correctly' we chose a penalty (by some trial and error) that ensured the agent played fewer than 3% invalid moves.

The only positive reward that was implemented was for winning a match. Each of the other statuses had a zero reward. The intuition here was to not penalize the agent for playing a move that was valid, but also, not penalize the agent for playing moves that resulted in ties or losses, only rewarding the agent for winning moves.

It was observed that by adding rewards for a tie or a negative reward for a loss, the agent was drawn to more ties and fewer wins. This reward system, by trial and error, resulted in very few invalid moves and a win rate of 90% by 100,000 episodes. It can be noted that another combination of rewards might achieve similar or better results; the reward system could also depend on what value is placed on each outcome - for example; is the objective to maximize the number of wins, or minimize the number of losses even if there are more ties.

Problem 5

Training

(A) The learning curve for the agent is shown in the following figure. The number of neurons in the hidden layer was modified to 9 instead of the default 64 from the starter code.

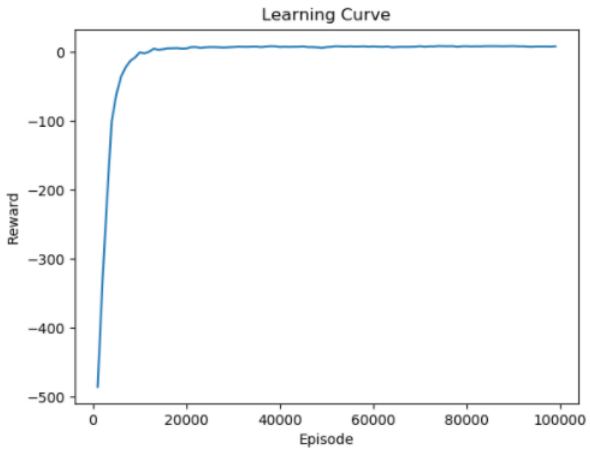
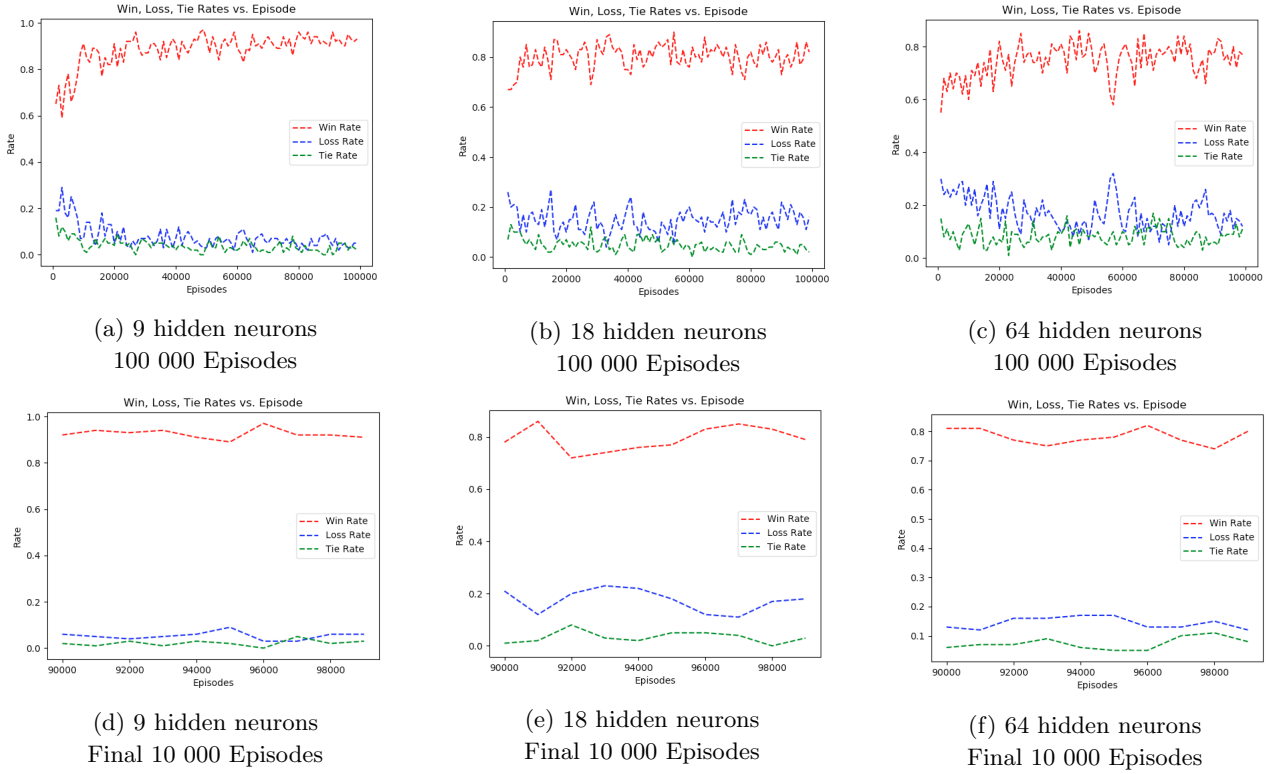


Figure 1: Learning curve - average return vs. episode number

(B) The effect of number of units in the hiddle layer vs. performance of the agent was measured by both increasing and decreasing the number of units and measuring the win rate of the trained agent at 100,000 episodes. The performance was tested with numerous numbers of hidden neurons. The performance with 9, 16, and 64 hidden neurons is shown on the following page. It can be seen that 9 hidden neurons results in both the best win rate of the trained epsiode at 100,000 episodes and the highest stability (less fluctuation).



(a) 9 hidden neurons 100 000 Episodes

(b) 18 hidden neurons 100 000 Episodes

(c) 64 hidden neurons 100 000 Episodes

(d) 9 hidden neurons Final 10 000 Episodes

(e) 18 hidden neurons Final 10 000 Episodes

(f) 64 hidden neurons Final 10 000 Episodes

Figure 2: Tic-Tac-Toe agent performance with number of hidden neurons (all episodes on top and last 10000 below

(C) Based on the choice of rewards and number of units in the hidden layer, we observed that agent stopped playing invalid moves at approximately episode 25000. To obtain this answer, we fetched the environment status every time an agent played an invalid move and incremented the total number of invalid moves by 1. The total numnber of invalid moves was then output to the console. Some examples of the output are shown below.

```
python tictactoe.py 30000
Number of games won: 90
Number of games tied: 1
Number of games lost: 9
Number of invalid moves: 1

python tictactoe.py 50000
Number of games won: 88
Number of games tied: 1
Number of games lost: 11
Number of invalid moves: 0

python tictactoe.py 99000
Number of games won: 90
Number of games tied: 1
Number of games lost: 9
Number of invalid moves: 0
```

(D) After running the agent for 100000 episodes, we were able to obtain a win rate of approximately 88-90% (the reason for a range was due to the fact that an issue with seeding any random functionality was not resolved in time for this report). The loss rate fluctuated between 15-20% and the tie rate was the remaining percentage of games played. The following output displays the first 5 games played by our agent, given a set of policy weights:

```
# Game 1
xo.
ox.
..x
====
# Game 2
x..
.x.
oox
====
# Game 3
xo.
.x.
o.x
====
# Game 4
x.x
oxo
o.x
====
# Game 5
xo.
.xo
..x
====
```

Problem 6

Win Rate over Episodes

The following graphs outlines the win, lose, and tie rates for the agent vs. episode for 9 hidden neurons (the chosen hyperparameter).

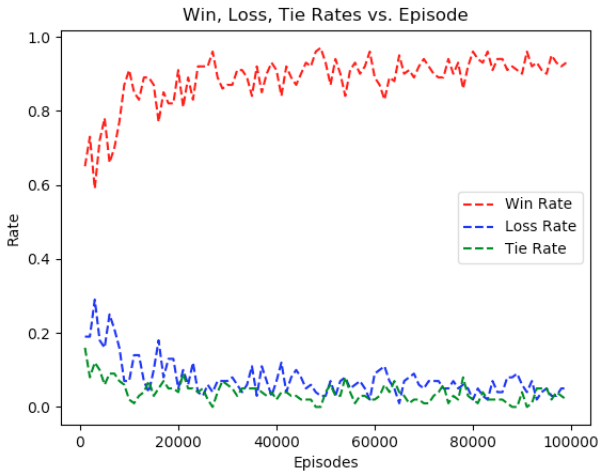


Figure 3: Win/Lose/Tie rates vs. episode - 100 000 Episodes

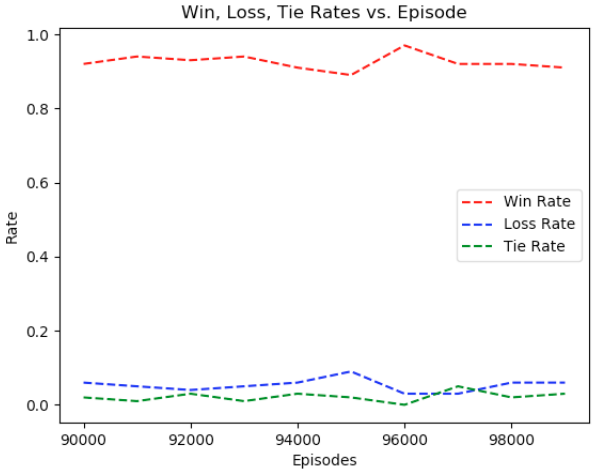


Figure 4: Win/Lose/Tie rates vs. episode - Final 10 000 Episodes

Given that a random agent playing the game results in a win rate of approximately 60%, it appears that the agent is learning effectively enough to net a win rate that is higher than if a random agent was playing the game. The three lines in the figure however, are quite noisy (e.g. they do not follow a smooth pattern), perhaps indicating that the hyperparameters could be tuned some more.

Problem 7

First Move Distribution Over Episodes

For the final trained model, execution of the first_move_distr function resulted in the following output:

```
Columns 0 to 5
 6.1930e-01 2.8002e-05 3.5696e-03 1.7899e-05 1.6235e-01 2.8408e-05

Columns 6 to 8
 1.7460e-04 4.4822e-06 2.1453e-01
[torch.FloatTensor of size 1x9]
```

This output indicates that the agent has a high likelihood of playing first in either the centre of the board or in the bottom right corner

Problem 8

Limitations

TO DO