CSC411: Assignment 4

Due on Monday, Apr. 2, 2018

Zhongtian Ouyang/Yihao Ni

Problem 1

Environment
The grid is a three by three matrix, where every spot is '.' when it is empty, 'x' for player one and 'o' for player two. The attribute turn is 1 when player one is playing, and 2 when player two is playing. The attribute done is True in two cases, either the game is in a win state, or all of the spots are filled.
Following is the output of a game of tic-tac-toe against myself.

```
>>> from tictactoe import Environment
>>> ttt = Environment()
>>> ttt.step(4)
(array([0, 0, 0, 0, 1, 0, 0, 0, 0]), 'valid', False)
>>> ttt.step(0)
(array([2, 0, 0, 0, 1, 0, 0, 0, 0]), 'valid', False)
>>> ttt.step(1)
(array([2, 1, 0, 0, 1, 0, 0, 0, 0]), 'valid', False)
>>> ttt.step(3)
(array([2, 1, 0, 2, 1, 0, 0, 0, 0]), 'valid', False)
>>> ttt.step(7)
(array([2, 1, 0, 2, 1, 0, 0, 1, 0]), 'win', True)
>>> ttt.render()
ox.
ox.
.x.
====
```

Problem 2

Policy
Part a)

```
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.net = torch.nn.Sequential(
            torch.nn.Linear(input_size, hidden_size),
            torch.nn.ReLU(),
            torch.nn.Linear(hidden_size, output_size),
            torch.nn.Softmax(dim=1)
        )

    def forward(self, x):
        return self.net(x)
```

Part b)
With a input of 9-dimensional state, the 27-dimensional state is generated with the following two lines.

```
state = torch.from_numpy(state).long().unsqueeze(0)
state = torch.zeros(3,9).scatter_(0,state,1).view(1,27)
```

Run this two lines with input

```
state = np.array([1, 0, 2, 0, 1, 1, 0, 0, 2])
```

After execute the two lines, we got

```
Columns 0 to 12
     0     1     0     1     0     0     1     1     0     1     0     0     0
Columns 13 to 25
     1     1     0     0     0     0     0     1     0     0     0     0     0
Columns 26 to 26
     1
```

So the output implies that the first 9 columns indicate whether an entry is empty, the second 9 columns indicate whether an entry is "x", the third 9 columns indicate whether an entry is "o".

Part c)
The 9-dimensional vector represents the preference/probability of playing each positions. So the first dimension is the preference/probability of making action 1, the second dimension is the preference/probability of making action 2, etc. Our policy is stochastic.

Problem 3

Compute the gradient Part a)

```
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]
    """
    max_t = len(rewards)
    result = [0] * max_t
    for i in range(max_t):
        for j in range(i, max_t):
            result[i] += rewards[j] * (gamma**(j-i))
    return result
```

Part b)
We can't update weights in the middle of an episode because the policy is for playing the whole game, not a single step. So if we change weights during a game, we can't assess how good this policy is. And the expected total reward won't be correct.

Problem 4

Choosing rewards
Part a)

```
def get_reward(status):
    """Returns a numeric given an environment status."""
    return {
            Environment.STATUS_VALID_MOVE   : 0, # TODO
            Environment.STATUS_INVALID_MOVE: -15,
            Environment.STATUS_WIN          : 10,
            Environment.STATUS_TIE          : -1,
            Environment.STATUS_LOSE         : -10
    }[status]
```

Part b)
Valid move is zero, win is positive and all the others are negative. Valid move is not encouraged or discouraged so it is zero. We do not want any invalid moves, even if we are losing, so it is negative and the magnitude is larger then win. The value of tie does not matter much as long as it is between win and lose. However, the goal is to win so we give tie a negative value. Lose is of course negative and as mentioned before, its magnitude should be smaller than invalid move.

Problem 5

Training the policy

Part a)

Following is the training curve with the rewards chosen from problem 4. The only hyperparameter to change is the value for gamma. We tried values 0.7, 0.8, 0.9 and 1.0. 0.8 gives the best win rate against random.

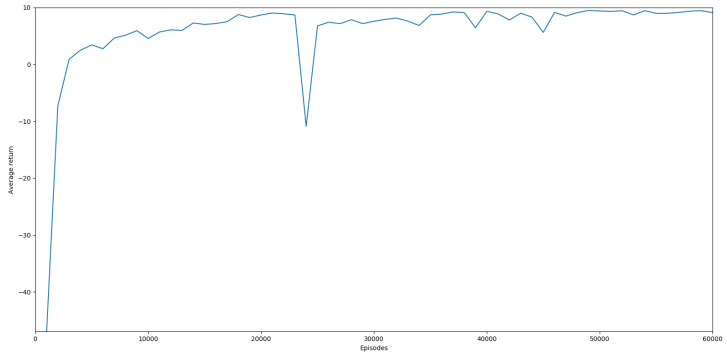

Figure 1: training curve

Part b)

To find out what is the best number of hidden units, I train the policy with 54,64,128,256 hidden units. Then I find out their winrates on playing 400 games with a random player2 using the weights at 1000, 2000, 3000, ..., 60000 episodes. The one with the best winrate what we want. In my case, it is the policy with 64 hidden units at episode 49000. The win rate is 97.75 %.

Part c)

The following graph is obtained by playing 400 games with a random player 2 using the best policy with weights from episode 1000, 2000, 3000, ..., 60000. The total number of invalid moves chose by our policy divided by 400 would get us the below values. From the graph, we can see that our policy learn to not made invalid moves at around 5000 episodes. After that, the rate is just oscillating between 0 and 0.25 invalid moves per game. As a reference, in the previous part, at 49000, our best performance episode, only 1 invalid move is made throughout the 400 games.

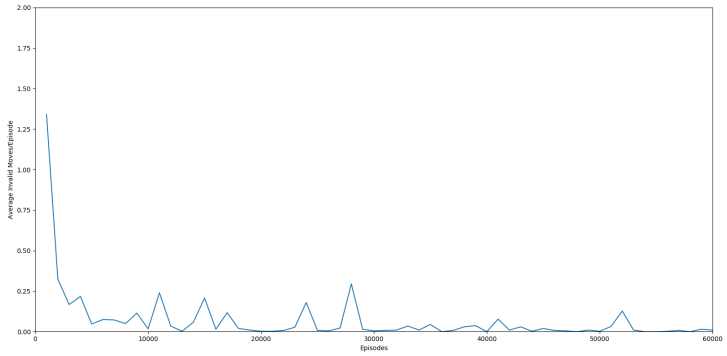

Figure 2: Average invalid moves per game

Part d)
Following is the performance of our learned policy against random (output format is modified to better display the result):

```
==================================Part 5d==================================
Use weights from episode 49000
100 games played:
Win: 92 Percentage: 0.92
Lose: 1 Percentage: 0.01
Tie: 7 Percentage: 0.07
0 invalid actions are made

====== Game1 ======
..x    ..x    ..x    ..x    ..x
...    o..    o.x    o.x    o.x
...    ...    ...    .o.    .ox
Learned policy wins against random!

====== Game2 ======
..x    .ox    .ox    .ox    .ox
...    ...    ...    .o.    .ox
...    ...    ..x    ..x    ..x
Learned policy wins against random!

====== Game3 ======
..x    .ox    .ox    .ox    .ox    .ox    xox
...    ...    ...    ..o    .xo    .xo    .xo
...    ...    ..x    ..x    ..x    .ox    .ox
Learned policy wins against random!

====== Game4 ======
..x    ..x    x.x    x.x    xxx
...    ...    ...    ...    ...
...    ..o    ..o    .oo    .oo
Learned policy wins against random!

====== Game5 ======
..x    ..x    ..x    .ox    .ox
...    ...    ..x    ..x    ..x
...    .o.    .o.    .o.    .ox
Learned policy wins against random!
```

The policy learned to place the first cross at the right-corner position, and tries to win the game by filling the right colomn. It does not make any invalid move and if there is a chance to win, it wins. However, it may not know how to not lose, ie. to block the opponent to win.

Problem 6

Win rate throughout training

From the graph below, we could see that thoughout training, even though the win rate is not stable, the overall trend is increasing, accompanied by the decreasing of both tie rate and loss rate. At around 40000 episode, the performance start to stablize.

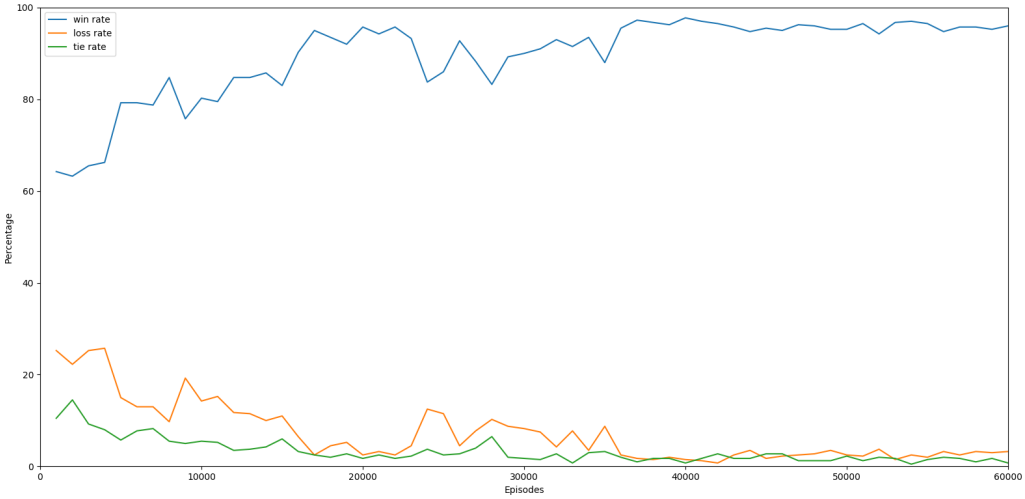

Figure 3: Win rate

Problem 7

First move distribution

```
Distribution of the first step from policy with 64 hidden units at episode 49000
Probability playing position 0: 0.00 %
Probability playing position 1: 0.00 %
Probability playing position 2: 1.00 %
Probability playing position 3: 0.00 %
Probability playing position 4: 0.00 %
Probability playing position 5: 0.00 %
Probability playing position 6: 0.00 %
Probability playing position 7: 0.00 %
Probability playing position 8: 0.00 %
```

I first thought this is unreasonable as we instinctly feels the center(position 4) is the best first move to made. However, with some research online about the tictactoe game and I found out that that placing the first move at one of the corners is at least as good as, if not better than, placing the first move at the center. Since position 2 is the top right corner, this first move is reasonable. Especially when facing an random opponent. When this first move is at corner, the only way that player 2 can tie the game, assume player one plays perfectly, is put the first move at center and second move in one of following positions: 1,3,5,7. The chance for a random player 2 for doing that is 1/8 * 2/3 = 1/12.
reference: https://www.quora.com/Is-there-a-way-to-never-lose-at-Tic-Tac-Toe

Below is a graph showing how the distribution changed throughout training. We can see that for the first 10000 episodes, the policy is not very sure what to do. After that it starts to oscillate between position 0, 2, 5. 0 and 2 are both a corner position as stated above, while 5 is a as good selection. And this is shown by the fact that postition 5 is dominant for less and less episodes as the policy evolve. And from episodes 40000 and onward, it decides position 2 is the best first move and keeps doing that.

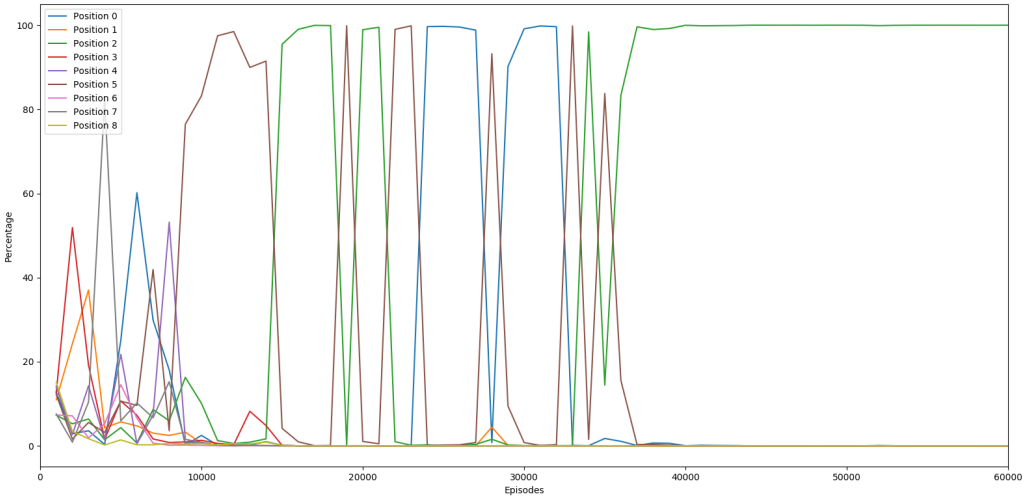

Figure 4: First move distribution

Problem 8

Limitation

The limitation of our learned policy is that it doesn't learn how to stop opponent from winning. Since out policy is trained facing an random policy, it doesn't expect the opponent to deliberately trying to win. And so it doesn't develop a stratage that would effectively and actively prevent opponent from winning