CSC411: Assignment #4

Due on Monday, April 2, 2018

ZILI XIE

March 19, 2018

Part 1

```
>>> env = Environment()

>>> env.step(4)
(array([0, 0, 0, 0, 1, 0, 0, 0, 0]), 'valid', False)

>>> env.render()
...
.x.
...
====

>>> env.step(5)
(array([0, 0, 0, 0, 1, 2, 0, 0, 0]), 'valid', False)

>>> env.render()
...
.xo
...
====

>>> env.step(0)
(array([1, 0, 0, 0, 1, 2, 0, 0, 0]), 'valid', False)

>>> env.render()
x..
.xo
...
====

>>> env.step(1)
(array([1, 2, 0, 0, 1, 2, 0, 0, 0]), 'valid', False)

>>> env.render()
xo.
.xo
...
====

>>> env.step(8)
(array([1, 2, 0, 0, 1, 2, 0, 0, 1]), 'win', True)

>>> env.render()
xo.
.xo
..x
====
```

The attribute 'turn' indicates who is the next player. If 'turn' equals '1' then the player who use 'x' is going to play next. else is 'turn' equals '2' then the next player is the one who use 'o'. The attribute 'done' represents the status of tic-tac-toe game. If 'done' equals False then the game is at a stage when the game is still on going. Otherwise, the game has finished if 'done' equals True.

Part 2

Part 2(a)

```
class Policy(nn.Module):
    """
    The Tic-Tac-Toe Policy
    """
    def __init__(self, input_size=27, hidden_size=256, output_size=9):
        super(Policy, self).__init__()
        self.fullyConnected1 = nn.Sequential(
            nn.Linear(input_size, hidden_size),
            nn.ReLU()
        )
        self.fullyConnected2 = nn.Sequential(
            nn.Linear(hidden_size, output_size),
            nn.Softmax()
        )

    def forward(self, x):
        x = self.fullyConnected1(x)
        x = self.fullyConnected2(x)
        return x
```

The class *Policy* can be implemented as above with one hidden layer.

Part 2(b)

First of all, the following code can be used for converting the 3 by 3 matrix grid to a 27 dimensional vector. The function below uses some code captured from *select_action*

```
def Part2b(state):
    convert_27 = torch.from_numpy(state).long().unsqueeze(0)
    convert_27 = torch.zeros(3,9).scatter_(0,convert_27,1).view(1,27)
    return convert_27
```

Assuming we have a state of grid like the one below.

```
>>> env.render()
x..
ox.
.xo
====
```

Running the function *Part2b* will convert whatever the state is into a 27-dimensional vector that look like the following output.

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

>>> Part2b(state)



Columns 0 to 12
    0     1     1     0     0     1     1     0     0     1     0     0     0
```

```

Columns 13 to 25
    1     0     0     1     0     0     0     0     1     0     0     0     0

Columns 26 to 26
    1
[torch.FloatTensor of size 1x27]


>>>
```

The first 9 indices of this 27 dimensions vector stands for the entries that are empty in 3 by 3 grid. 9th to 18th dimension of vector represents all the grids that are filled with 'x'. The last nine indices are the entries of 'o'.

Part 2(c)

9 policy values are the probabilities it choose from the next move for each of the 9 grids.
The select_action samples the action from the probabilities. Then the policy is stochastic.

Part 3

Part 3(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]
    """
    result = []
    G = []
    L = len(rewards)
    for j in range(L):
        G.append(gamma ** j)
    G = np.array(G)
    for i in range(L):
        R = np.array(rewards[i:])
        result.append(np.dot(R,G[0:len(R)]))
    return result
```

The implementation of *compute_returns* function which takes a list of rewards and computes the returns $G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ...$ can be shown as above.

Part 3(b)

The backward pass cannot be computed during the episode because the reward is not fully recorded and thus computing the gradient may produce a biased return.

Part 4

Part 4(a)

```
def get_reward(status):
    """Returns a numeric given an environment status."""
    return {
            Environment.STATUS_VALID_MOVE: 10,
            Environment.STATUS_INVALID_MOVE: -60,
            Environment.STATUS_WIN: 80,
            Environment.STATUS_TIE: -20,
            Environment.STATUS_LOSE: -80
    }[status]
```

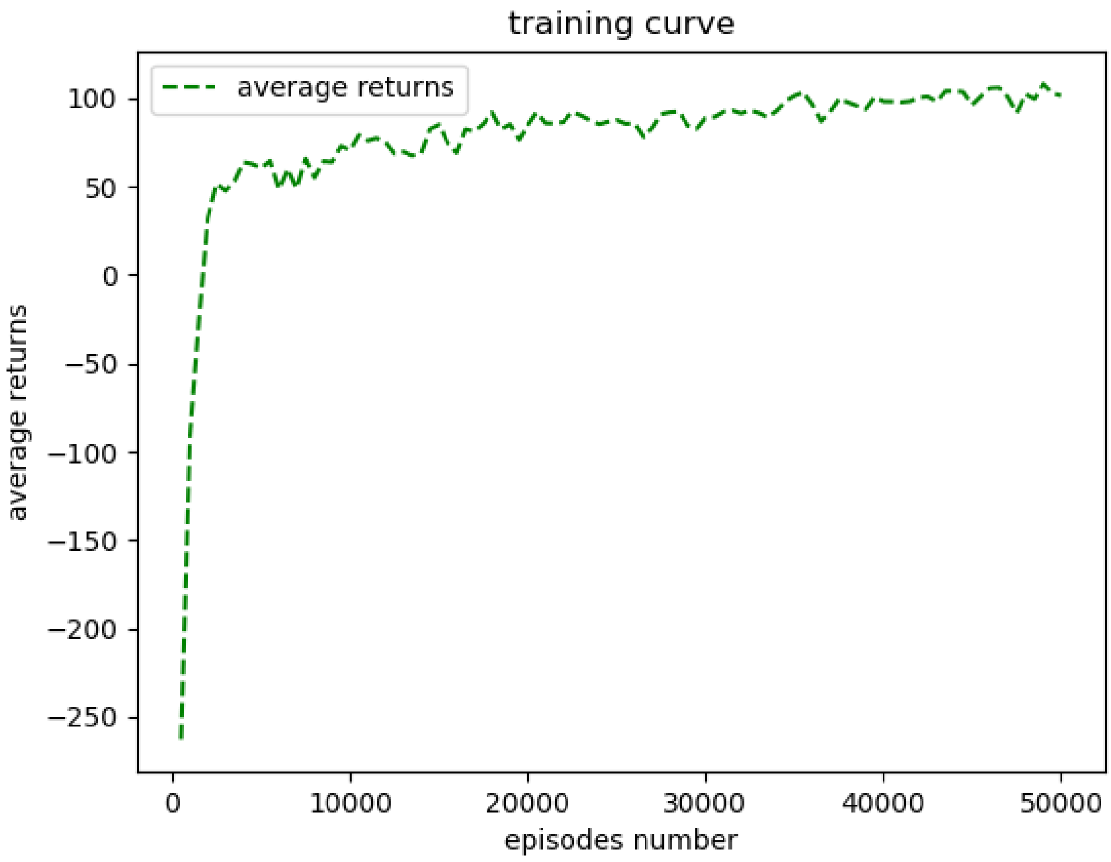
Part 4(b)

The rewards for win and lose are +80 and -80. This 2 status are 2 extreme result among given status. The reward for valid and invalid move are +10 and -60. In doing so, I try to avoid the network being satisfied with making invalid moves. The reward for tie is -20. Even a tied result is at the middle of win and lose this is still not the case that we want to have.

Part 5

Part 5(a)

The curve of training was shown above. The gamma value was set to be 0.8 to get a result in a faster speed.



learning rate value was shifted lower (0.0005) to avoid extreme values on average returns and the training being failed due to skipping minimum point. log_interval was changed to be 500 to obtain more records. Total number of iterations are 50000.

Part 5(b)

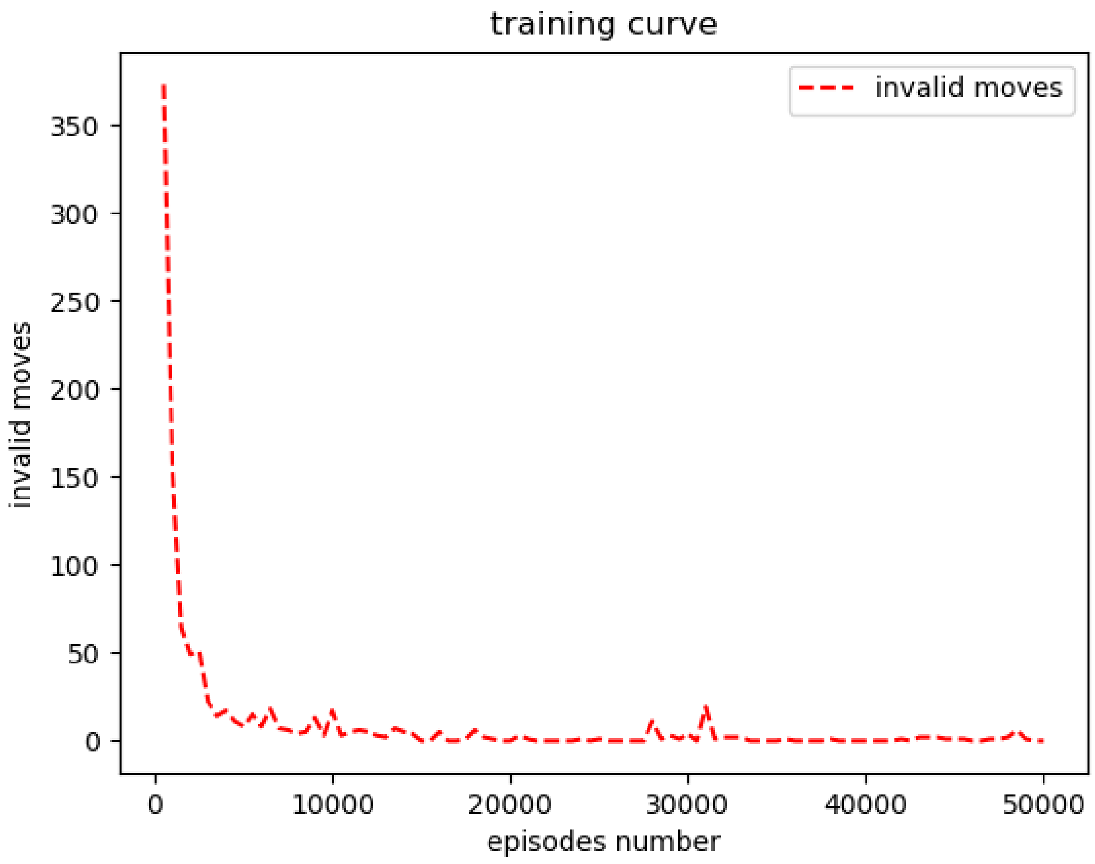
```
def p5b_diff_dim(env, policy, dim_list):
    #dim_list = [32, 64, 128, 256]
    table = list()
    for size in dim_list:
        env = Environment()
        policy = Policy(hidden_size = size)
        train(policy, env)
        counts = game_simulation(env, policy, games = 100)
        table.append(counts)
    pair = (dim_list, table)
    for i in range(len(dim_list)):
        print("size= {}; win, lose, tie, total, invalid= {}".format(dim_list[i], table[i]))
    return pair
```

Four different values of hidden units numbers are used for training policy. They are 32, 64, 128, 256. Result generated by the function can be show at below:

```
size= 32; win, lose, tie, total, invalid= (625, 159, 216, 4449, 21)
size= 64; win, lose, tie, total, invalid= (762, 118, 120, 4160, 13)
size= 128; win, lose, tie, total, invalid= (663, 85, 252, 4552, 7)
size= 256; win, lose, tie, total, invalid= (772, 60, 168, 4483, 5)
```

for the case of 256 hidden units, the statistic has the highest number of winning status outcome and smallest number of losing status observed. The result also has the lowest ratio of invalid move among 4 values. It is saying the more number of hidden units used the less likely the policy will choose a invalid move and more probability to win against random.

Part 5(c)



The number of invalid moves became stable after 4000th episodes, so that should be the time when policy learned how to stop making invalid moves.

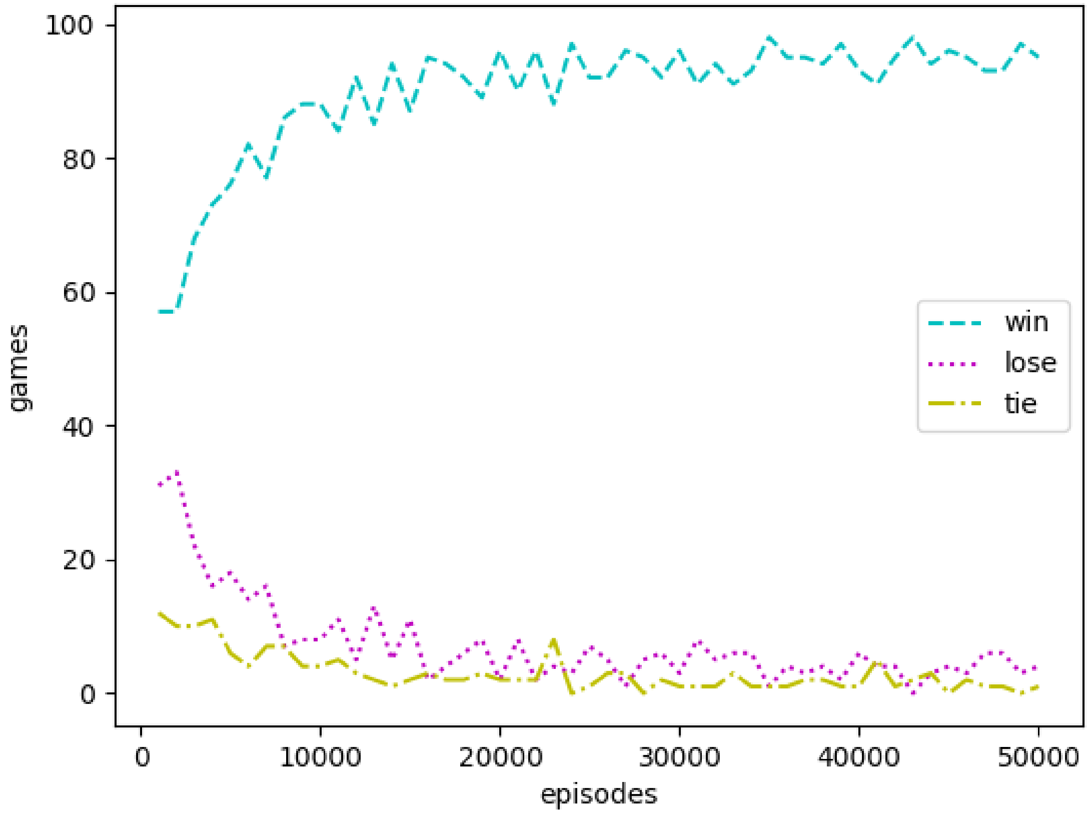
Part 5(d)

In the 100 times simulation of game play, the policy won the game 90 times, had tie result 6 times, and lose the game for 4 times. In most of the cases, my agent learn how to block the random policy. My agent also know how to play when there is just one step to win. It never make mistake in this case. Sometimes the strategy below can be observed, my trained agent choose to make the first move at the grid in very middle. This is usually the easiest way to win this game in real world if you play first.

```
game: 2

.xo
...
...
====
xxo
...
.o.
====
xxo
..o
xo.
====
xxo
x.o
xo.
====
======================
game: 3

ox.
...
...
====
oxx
..o
...
====
oxx
o.o
x..
====
oxx
oxo
x..
====
======================
game: 13

.x.
...
.o.
====
.xx
..o
.o.
```

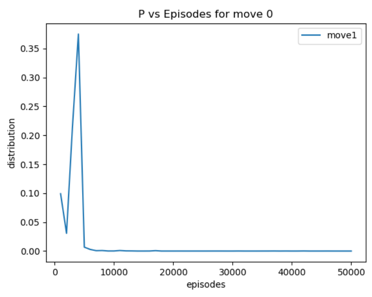
```
====
xxx
..o
.o.
====
======================
game: 16

.x.
...
..o
====
xx.
..o
..o
====
xxx
..o
..o
====
======================
game: 18

ox.
...
...
====
oxx
...
o..
====
oxx
.xo
o..
====
oxx
.xo
ox.
====

>>>
```

Part 6

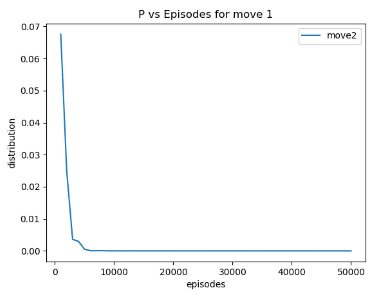


The number of game status(win, lose, tie) for different episodes are shown in plot above. From the plot we can find that the number of game winning in 100 games increases from 60 to about 90. The curves for lose and tie decrease as the episodes number increase because the network knows these status are not so good and try to avoid them.

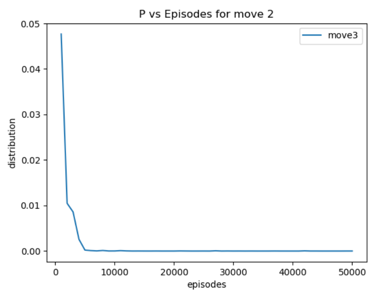
Part 7



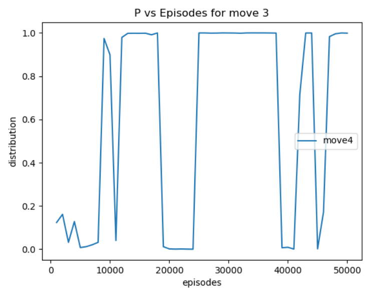
P vs Episode for move 0



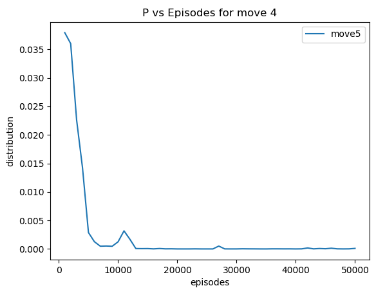
P vs Episode for move 1



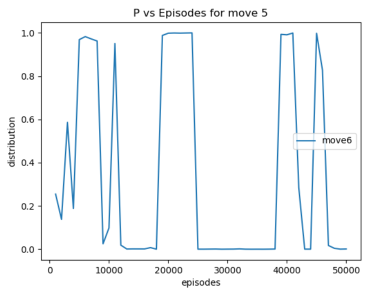
P vs Episode for move 2



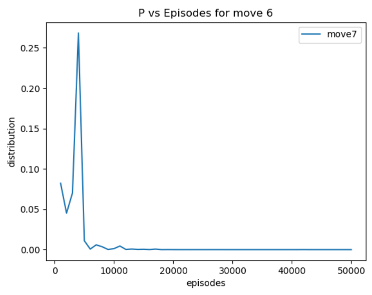
P vs Episode for move 3



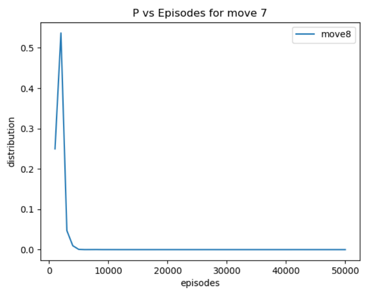
P vs Episode for move 4



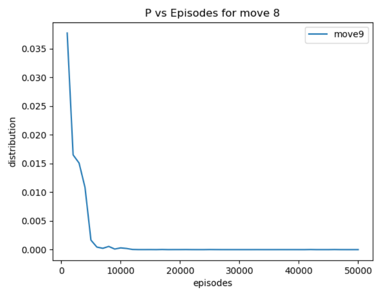
P vs Episode for move 5



P vs Episode for move 6



P vs Episode for move 7



P vs Episode for move 8

As the 9 plot above shows, the policy plays its first move on position 4 and 6 most of the time. for other positions, the probabilities are equally the same.

Part 8

This part I mainly focus on some games that the agent lose to random policy. In this simulation of 100 games played, Game 5, 18, 32, 49, 76, 81 are lose. Here are 3 samples from the list.

```
Games: 81
...
.o.
.x.
====
...
.o.
oxx
====
..x
.oo
oxx
====
.xx
ooo
oxx
====
```

```
Games: 76
..o
..x
...
====
o.o
..x
.x.
====
ooo
.xx
.x.
====
```

```
Games: 49
...
...
.xo
====
.x.
.o.
.xo
====
xxo
.o.
.xo
====
xxo
.oo
xxo
====
```

In Game 81, the random policy which was represented by 'o' won because the agent failed to block 'o' on grid 3. The agent knows how to block when in round 3 the random policy was about to move on grid 2. In this game random policy is more likely to win because it played its first move on grid 4.
In Game 76, the second move of agent should not be grid 7, because it will lead the sequence of 'x' to the edge of box.