CSC411/2525
PROJECT 4

Due on Monday, April 2, 2018

JooHyung Kim, Anas S. Alraheem

April 3, 2018

Part 1

The 3x3 grid is represented as a 1x9 dimension vector.
The attribute *done* represents the state of the game, if it is finished or not.
The attribute *turn* represents whose turn it is currently, either player 1 (x) or 2 (o).

```
env.render()
...
...
...
====

env.step(1)
Out[104]: (array([0, 1, 0, 0, 0, 0, 0, 0, 0]), 'valid', False)
env.render()
.x.
...
...
====

env.step(0)
Out[106]: (array([2, 1, 0, 0, 0, 0, 0, 0, 0]), 'valid', False)
env.render()
ox.
...
...
====

env.step(4)
Out[108]: (array([2, 1, 0, 0, 1, 0, 0, 0, 0]), 'valid', False)
env.render()
ox.
.x.
...
====

env.step(5)
Out[110]: (array([2, 1, 0, 0, 1, 2, 0, 0, 0]), 'valid', False)
env.render()
ox.
.xo
...
====

env.step(7)
Out[111]: (array([2, 1, 0, 0, 1, 2, 0, 1, 0]), 'win', True)
env.render()
ox.
.xo
.x.
====

# Game is finished.
```

Part 2

a)

Implementation of Policy class.
policy is a neural network with one hidden layer.

```
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.input = nn.Linear(input_size, hidden_size)
        self.output = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = F.relu(self.input(x))
        return F.softmax(self.output(x))
```

b)

Outputs produced to see what the state (27-dimensional vector) means.

```
state = torch.from_numpy(state).long().unsqueeze(0)
state = torch.zeros(3,9).scatter_(0,state,1).view(1,27)
state
Out[38]:
Columns 0 to 12
    1     1     1     1     1     1     1     1     1     0     0     0     0
Columns 13 to 25
    0     0     0     0     0     0     0     0     0     0     0     0     0
Columns 26 to 26
    0
[torch.FloatTensor of size 1x27]

state, status, done = env.step(1)
env.render()
.x.
...
...
====
state
Out[131]:
Columns 0 to 12
    1     0     1     1     1     1     1     1     1     0     1     0     0
Columns 13 to 25
    0     0     0     0     0     0     0     0     0     0     0     0     0
Columns 26 to 26
    0
[torch.FloatTensor of size 1x27]

state, status, done = env.step(7)
env.render()
.x.
...
.o.
```

```
====
state
Out[135]:
Columns 0 to 12
    1     0     1     1     1     1     1     0     1     0     1     0     0
Columns 13 to 25
    0     0     0     0     0     0     0     0     0     0     0     0     1
Columns 26 to 26
    0
[torch.FloatTensor of size 1x27]
```

After running few steps of the game, and taking a look at the *state* and *env.render()*, we have concluded that:
The first 9 dimensions [0:8] of the output indicates whether the cell in the grid is empty or not.
The next set of 9 dimensions [9:17] of the output indicates which cells are occupied by x's.
The last set of 9 dimensions [18:26] of the output inidicates which cells are occupied by o's

c)
Output of the policy

```
pr = policy(Variable(state))

pr
Out[142]:
Variable containing:
 0.1145  0.1305  0.1033  0.1130  0.1107  0.1141  0.0951  0.1028  0.1160
[torch.FloatTensor of size 1x9]
```

Each value in each dimension represents how good is it to choose that cell as the next move.
This policy is stochastic.

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]
    """
    G_t = []
    for i in range(len(rewards)):
        temp = [x * (gamma ** i) for i,x in enumerate(rewards[i:])]
        G_t.append(np.sum(temp))
    return G_t
```

b)

When playing a full game for each episode, the game starts with random weights and uses the result of the game to determine how to change the weights.

We can not update weights in the middle of an episode because without the result of the game, we don't have a metric on how to change the weights. Changing the values of the weights based on winning the game is different when changing the values of the weights based on losing the game. Therefore, we need to perform the updates after finishing the game.

Part 4

a)

```
def get_reward(status):
     """Returns a numeric given an environment status."""
    return {
            Environment.STATUS_VALID_MOVE  : 1,
            Environment.STATUS_INVALID_MOVE: -5,
            Environment.STATUS_WIN         : 10,
            Environment.STATUS_TIE         : -3,
            Environment.STATUS_LOSE        : -3
    }[status]
```

b)

The positive rewards are STATUS_VALID_MOVE and STATUS_WIN.
The negative rewards are STATUS_INVALID_MOVE, STATUS_TIE, STATUS_LOSE
We have no 0-value rewards.

We have chosen these values after experimenting with other numbers. However, these values produced the best results for our program. We decided to reward wins/valid moves as positives way more than the negative values of losses/ties/invalid moves. The magnitudes of the values were chosen after trial and error and the chosen values fit into our plots nicely. Moreover, when we tried to deviate from our optimal reward values even by 1 digit, it produced worse results.

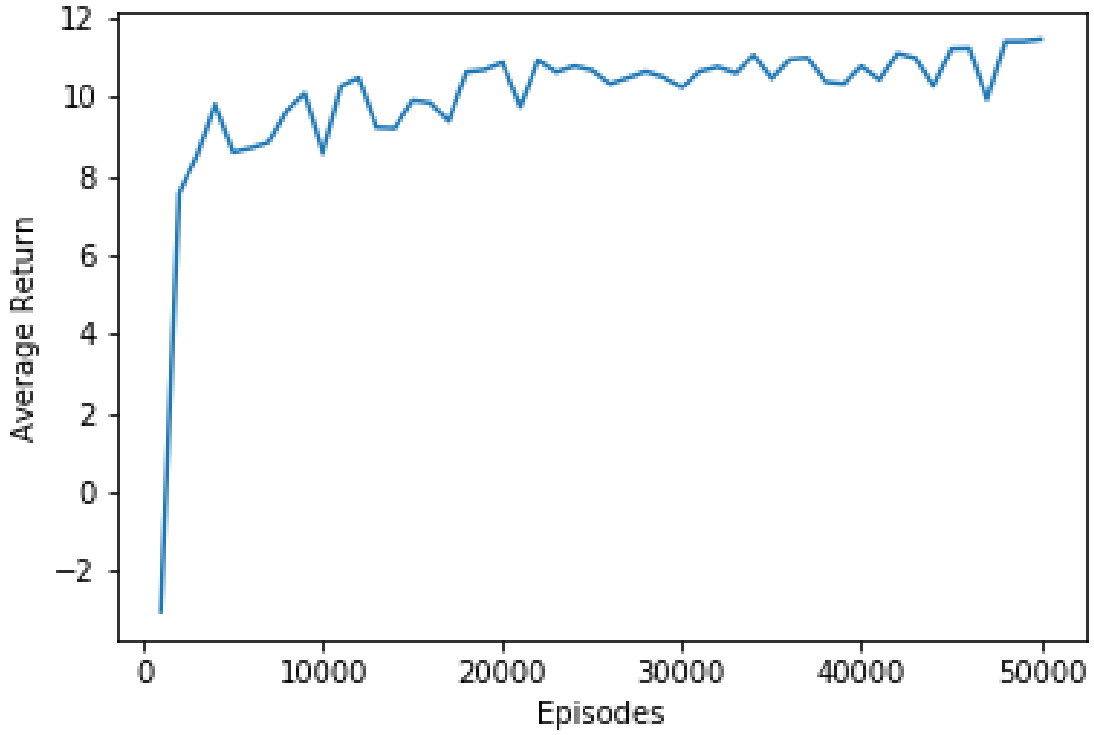
Part 5

a)

We have changed the **number of hidden units from 64 to 256** and **gamma = 0.5** because we got the best result from it, as high as 94.7% win rates towards the end of 50,000 episodes.

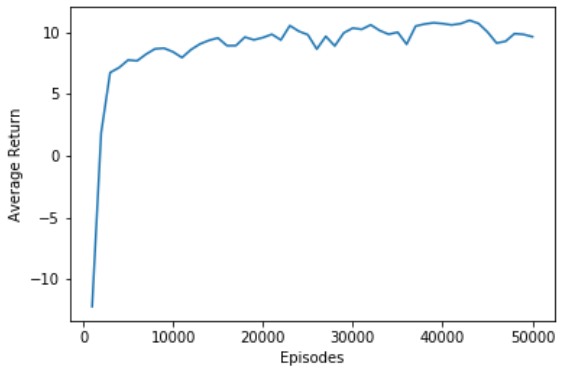
Experimented Values (average win rate over 50,000 episodes):
gamma = 0.91 with hidden units = 256 produced 77%
gamma = 0.75 with hidden units = 256 produced: 94.3% (also produced inconsistent plots)
When trying other hidden unit values with gamma=0.5, it produced worse results.

Note: After we completed our work we noticed the prof mentioned on piazza (Post @812) the first move choice our agent makes does not make much sense so we tried to experiment with different gamma values. That is why we experimented with other values (0.75 and 0.9) but we decided to keep our current results because the graphs produced were confusing and inconsistent (many low average returns in the middle of training, around episodes 30,000 and 40,000). We also thought of calculating the average of total wins throughout training and noticed it is best for gamma = 0.75.

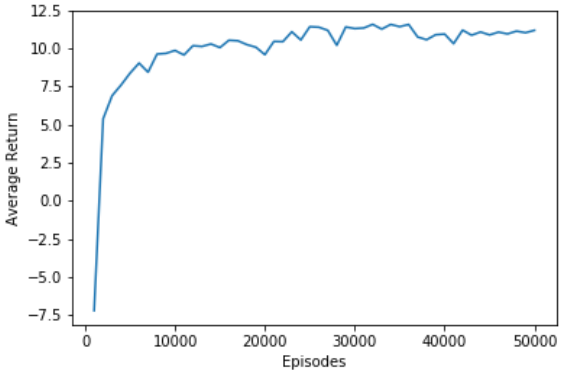


(a) Training Curve: hidden units: 256, gamma=0.5

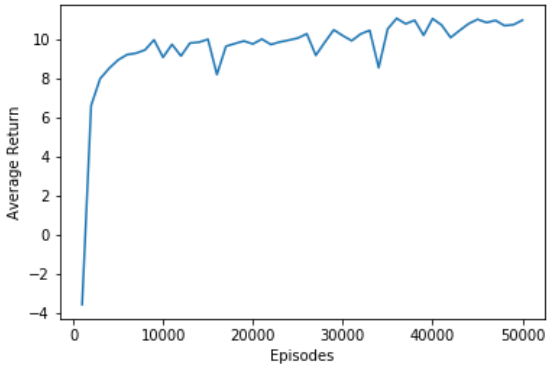
b) Experiment with different number of hidden units [32,64,128,256,400]



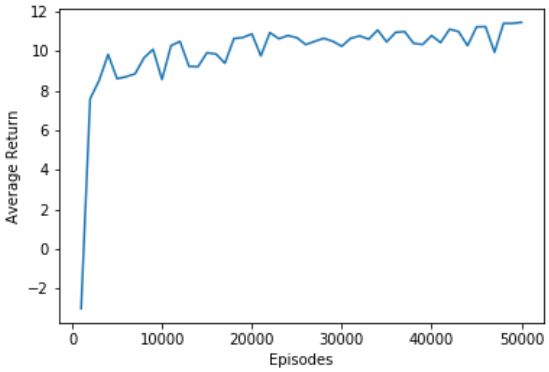
(a) Training Curve: hidden units: 32



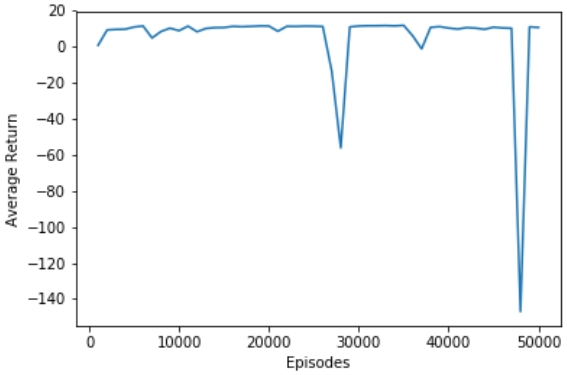
(b) Training Curve: hidden units: 64



(c) Training Curve: hidden units: 128



(d) Training Curve: hidden units: 256



(e) Training Curve: hidden units: 400

Total Number of Invalids for each experiment:

32: Total Invalid: 0.0484023555423
64: Total Invalid: 0.0398413971409
128: Total Invalid: 0.0337253535768
256: Total Invalid: 0.0328525005944
400: Total Invalid: 0.267968033307

hidden units = 256 produced the best results.

(c) At around 15000^{th} episode, the percentage of invalid moves made is around 0.4%

```
Episode 15000     Invalid: 0.00405679513185, Win: 812, tie: 64, lose:124
```

d)

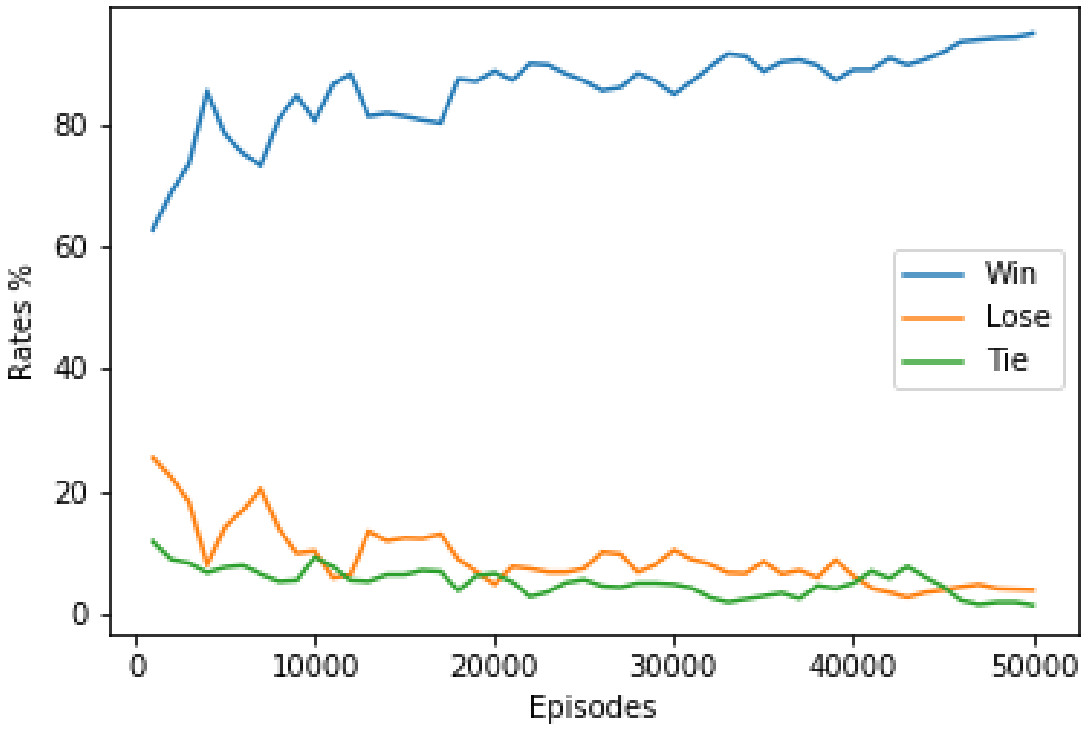
```
20th game:
.x.
...
.o.
====
.xx
...
oo.
====
xxx
...
oo.
====

40th game:
.x.
o..
...
====
.xx
o..
..o
====
xxx
o..
..o
====

60th game:
.x.
..o
...
====
.xx
.oo
...
====
xxx
.oo
...
====

80th game:
.xo
...
...
====
.xo
```

```
..x
o..
====
oxo
.xx
o..
====
oxo
.xx
ox.
====

100th game:
.x.
o..
...
====
.xx
oo.
...
====
xxx
oo.
...
====

{'win': 92, 'inv': 0, 'valid': 226, 'done': 0, 'lose': 8, 'tie': 0}
```

One of the strategies that our agent learned is to play the first move by placing the x on the top middle cell. Lastly, our agent has learned to win!

Part 6

We obtained the win / lose / tie rates while running our train function. We obtained the rates every 1000$^{\text{th}}$ episodes and created a graph with it.
We changed the hyperparameters in our train function, hidden unit=256 and gamma=0.5.



(a) win / lose / tie rates

Conclusion: For the final 1000 episodes, wins: 947, loses: 39, ties: 14
We can see that the rates fluctuated throughout the training, as our agent was exploring new moves aside from the optimal learned moves at that time.
In the end, as evident, the overall win rates improved and the lose, tie rates decreased.

Part 7

```
# 0th episode
[
 0.1243  0.0912  0.1001  0.0982  0.1174  0.1420  0.1102  0.1106  0.1061
[torch.FloatTensor of size 1x9]
,

# 10000th episode
Columns 0 to 5
 2.1241e-04  1.4382e-05  1.3317e-04  1.3351e-03  9.9746e-01  4.0962e-04

Columns 6 to 8
 1.7974e-04  4.6109e-09  2.5092e-04
[torch.FloatTensor of size 1x9]
,

# 20000th episode
Columns 0 to 5
 1.1363e-06  2.5312e-08  8.3705e-05  9.9973e-01  1.8400e-04  4.2848e-07

Columns 6 to 8
 2.5664e-09  1.0232e-11  3.6978e-08
[torch.FloatTensor of size 1x9]
,

# 30000th episode
Columns 0 to 5
 3.7690e-06  9.9970e-01  2.9216e-04  1.2763e-07  4.8369e-09  3.0131e-11

Columns 6 to 8
 4.5049e-11  2.9596e-14  9.1078e-12
[torch.FloatTensor of size 1x9]
,

# 40000th episode
Columns 0 to 5
 9.9978e-01  2.0135e-04  1.7210e-05  4.4637e-09  7.4169e-12  9.3531e-09

Columns 6 to 8
 7.0566e-13  1.8760e-14  2.5324e-13
[torch.FloatTensor of size 1x9]
,

# 50000th episode
Columns 0 to 5
 2.5837e-11  1.0000e+00  1.0941e-06  3.5961e-08  5.7096e-11  4.0071e-10

Columns 6 to 8
 1.2390e-14  2.0452e-15  6.5686e-14
[torch.FloatTensor of size 1x9]
]
```

We see that our model learned that the best first move is the top meddle cell. The distribution does not makes much sense because as logical players the best first moves are the center or the corners. However the model might have found that the top middle cell resulted in a high win rate against the random agent and thus that cell became the default first move.

We can see the value of the second cell fluctuating during the training. In the 0^{th} episode we see that all the cells have similar values, which makes sense as the training didn't start. As more episodes finish, the second cell value fluctuates between being the highest value and one of the lowest, which corresponds to the agent trying different strategies. Towards the end of the training, the $50{,}000^{\text{th}}$ episode shows that the second cell has the highest value.

Part 8

One of the mistakes our agent is making is that, the first move is always the same (placing x in the middle top cell). As a human, and through google search, we believe the best first move you can make is to place it in the middle cell. And our agent has failed to learn that.

On certain games, when blocking the other player's move is the best move. It does not do that and the distribution value on the optimal cell is lower than other empty cells.

Example game:

```
First moves:
.xo
...
...
====

Grid Cell Probability Distribution:
Columns 0 to 5
 2.6477e-03  1.0620e-03  4.8685e-06  1.9606e-02  2.0047e-02  9.5508e-01
Columns 6 to 8
 4.5125e-04  1.0973e-03  1.9568e-09
.xo
.ox       <-- The agent 'x' makes a useless move.
...
====

Grid Cell Probability Distribution:
Columns 0 to 5
 5.3413e-03  1.5633e-04  7.9803e-05  9.3969e-01  9.2121e-08  2.0375e-07
Columns 6 to 8
 5.3717e-02  2.8134e-04  7.3367e-04
.xo
xox
.o.       <-- The optimal move for 'x' would be the bottom left cell to block 'o'.
====

Grid Cell Probability Distribution:
Columns 0 to 5
 6.4252e-03  5.4179e-05  8.3312e-05  4.1068e-06  2.5790e-07  4.1198e-05
Columns 6 to 8
 4.4863e-01  9.7968e-10  5.4477e-01
[torch.FloatTensor of size 1x9]
.xo
xox
xoo
====
```