CSC411: Project #4
Tic-Tac-Toe with Policy Gradient

Due on Tuesday, April 3, 2018

Dhruv Chawla, Saila Maham Shama

April 3, 2018

Part 1

The tic-tac-toe text output is shown below.

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

Part 2

2(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.fc1 = nn.Linear(input_size, hidden_size)
  self.fc2 = nn.Linear(hidden_size, output_size)

def forward(self, x):
  x = F.relu(self.fc1(x))
  x = self.fc2(x)
  softmax = nn.Softmax()
  return softmax(x)
```

2(b)

Running the following script

```
  policy = Policy()

  # xoo
  # .x.
  # ..x
  # =====

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

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

we yield the output

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

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

Columns 26 to 26
0
[torch.FloatTensor of size 1x27]
```

The output seems to imply that the first nine dimensions indicate entries that are empty, the second nine dimensions indicate entries that are x's and the last nine dimensions indicate entires that are o's.

2(c)

The output of the select_output function is a nine-dimensional vector indicates which position of the tic-tac-toe table to place that player's mark. The policy is stochastic.

Part 3

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} + ..
"""
  G = [0] * len(rewards)

  for i in range(len(rewards)-1, -1, -1):
  if i == len(rewards)-1: G[i] = rewards[i]
  else: G[i] = rewards[i] + gamma * G[i+1]

  return G
```

3(b)

We cannot update the weights in the middle of an episode because with just one move, we cannot know if it will lead to winning/losing/having a tie. The update to weights depends on the terminating status of an episode. This is why weights must be updated after each episode, and not in between.

Part 4

4(a)

The modified `get_reward(status)` function is included below:

```
def get_reward(status):
"""Returns a numeric given an environment status."""
return {
  Environment.STATUS_VALID_MOVE  : 1,
  Environment.STATUS_INVALID_MOVE: -250,
  Environment.STATUS_WIN         : 500,
  Environment.STATUS_TIE         : -3,
  Environment.STATUS_LOSE        : -3
}[status]
```

4(b)

There are positive and negative rewards. Negative rewards are given for undesirable outcomes (invalid move, tie and lose) while positive rewards are given for winning and making a valid move. Initially the reward for tie was positive and comparable to the reward for winning but was then made negative to increase the win rate (we observed that the AI was going for ties and was effectively treating tie-ing the same as winning).

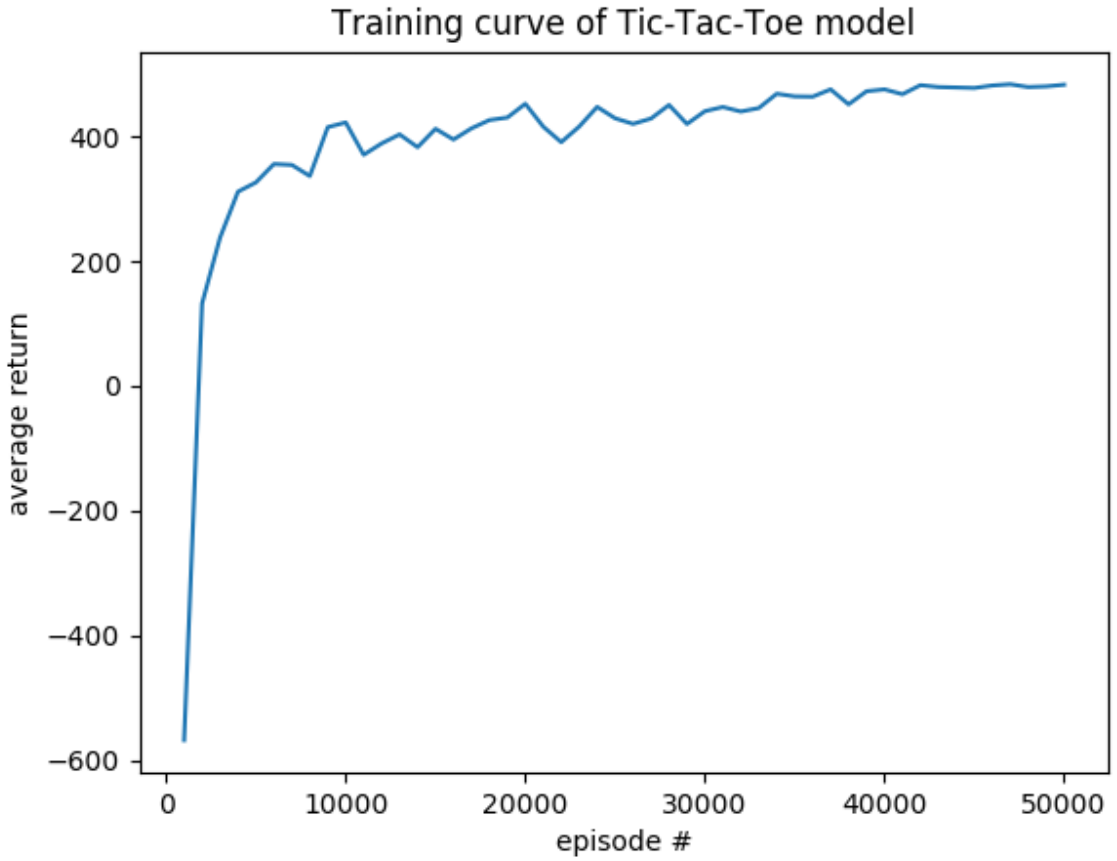
As far as the magnitude of the rewards go, the reward for winning was increased by a factor of 100 compared to the magnitudes of losing and tieing to incentivize the AI to win more. The reward for making an invalid move was initially -1 but was increased in magnitude to bring down the number of invalid moves during the game.

Part 5

5(a)

Refer to Figure 1. One hyperparameter that was changed was γ. Initially it was set to $\gamma = 1$, but changing it to $\gamma = 0.75$ incentivized the AI to finish the game in shorter number of turns, which helped penalize ties and invalid moves.

Figure 1: Training curve (# hidden units: 64)



5(b)

The number of hidden units were reset to values 32, 128 and 256 to obtain the training curves in Figure 2, 3 and 4 respectively.

Figure 2: Training curve (# hidden units: 32)

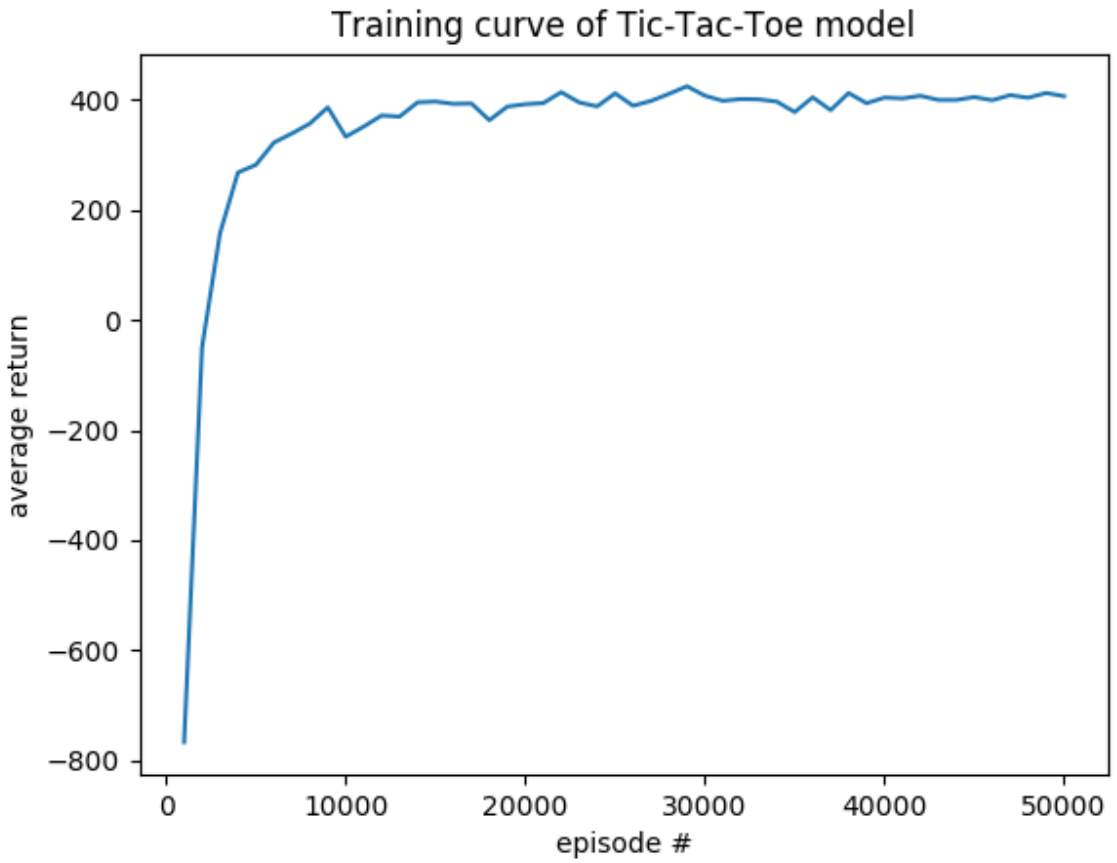


Figure 3: Training curve (# hidden units: 128)

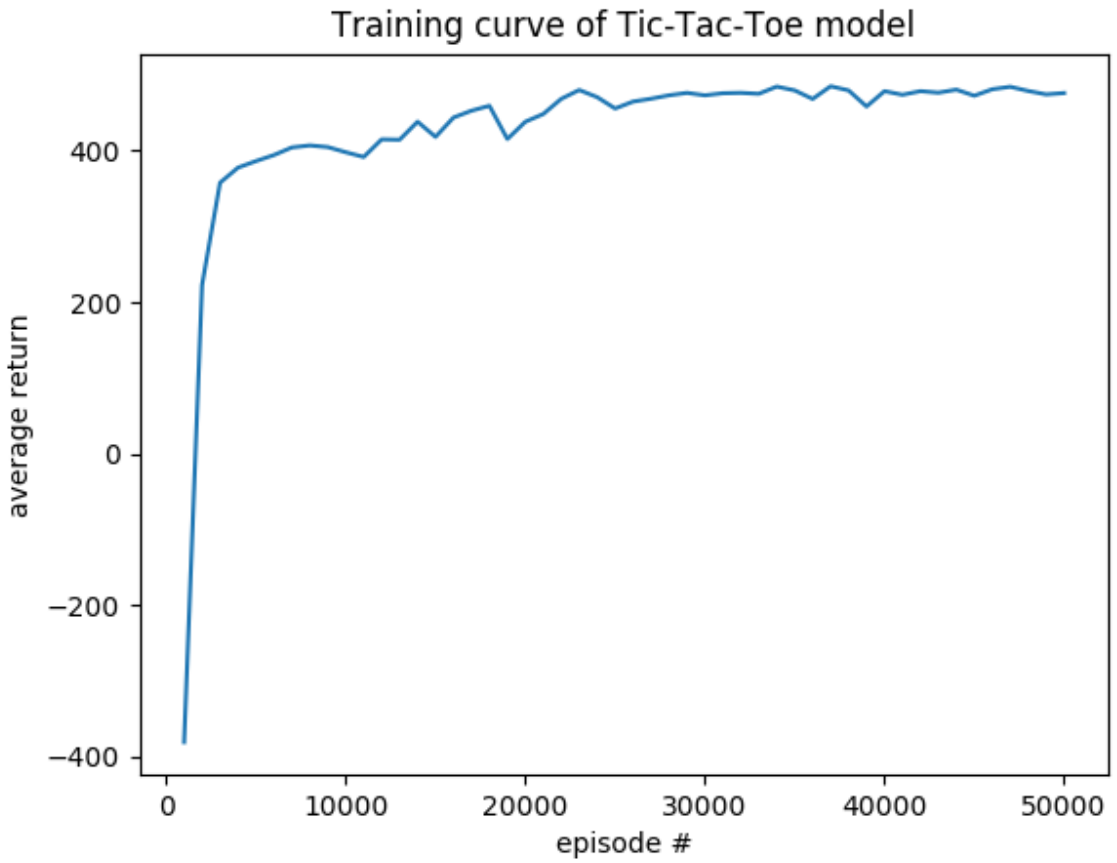
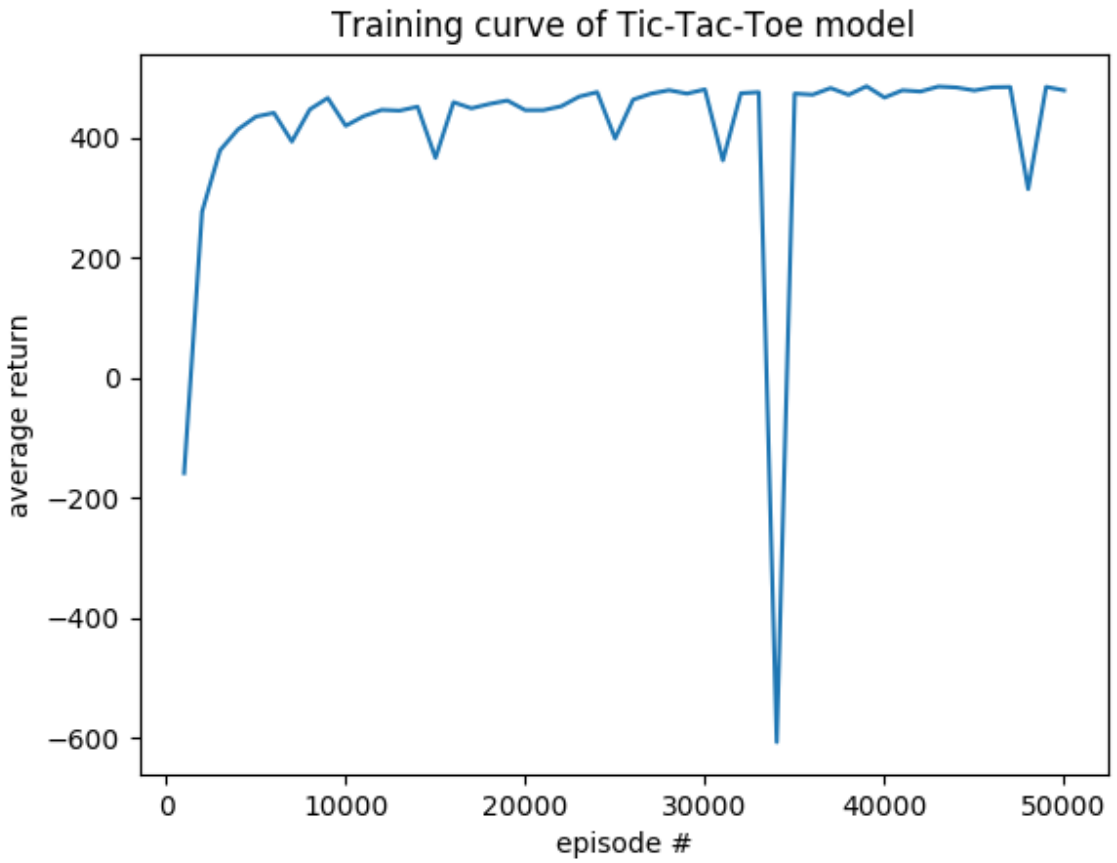


Figure 4: Training curve (# hidden units: 256)



5(c)

For training with hidden units 32, 64 and 128, Figures 1, 2 and 3 imply that the policy never plays excessive invalid moves since the average return never becomes negative. For the case of 256 hidden units however, the average dips to a significant nevative value at around the 35000 episode. So, the policy probably learnt to stop playing invalid moves after that.

5(d)

Using the final learned policy, the agent wins 98/100 games and ties 2/100. This can be seen in Figure 5. Five of the 100 sample games are displayed below. Our agent plays crosses and makes the first move. The agent clearly always makes the first cross on cell 2 (top right). It seems to make its second move on cell 8 (bottom right), if not already occupied. It seems easy for the agent to win since the random player doesn't try to stop the agent from making obvious wins.

```
Game: 0          Game: 19         Game: 57         Game: 76         Game: 95
..x              o.x              ..x              .ox              ..x
...              ...              o..              ...              ...
o..              ...              ...              ...              ..o
====             ====             ====             ====             ====
o.x              o.x              o.x              .ox              .ox
...              o..              o..              ...              .x.
o.x              ..x              ..x              .ox              ..o
====             ====             ====             ====             ====
o.x              o.x              o.x              .ox              .ox
..x              o.x              o.x              ..x              .x.
o.x              ..x              ..x              .ox              x.o
====             ====             ====             ====             ====
```

Part 6

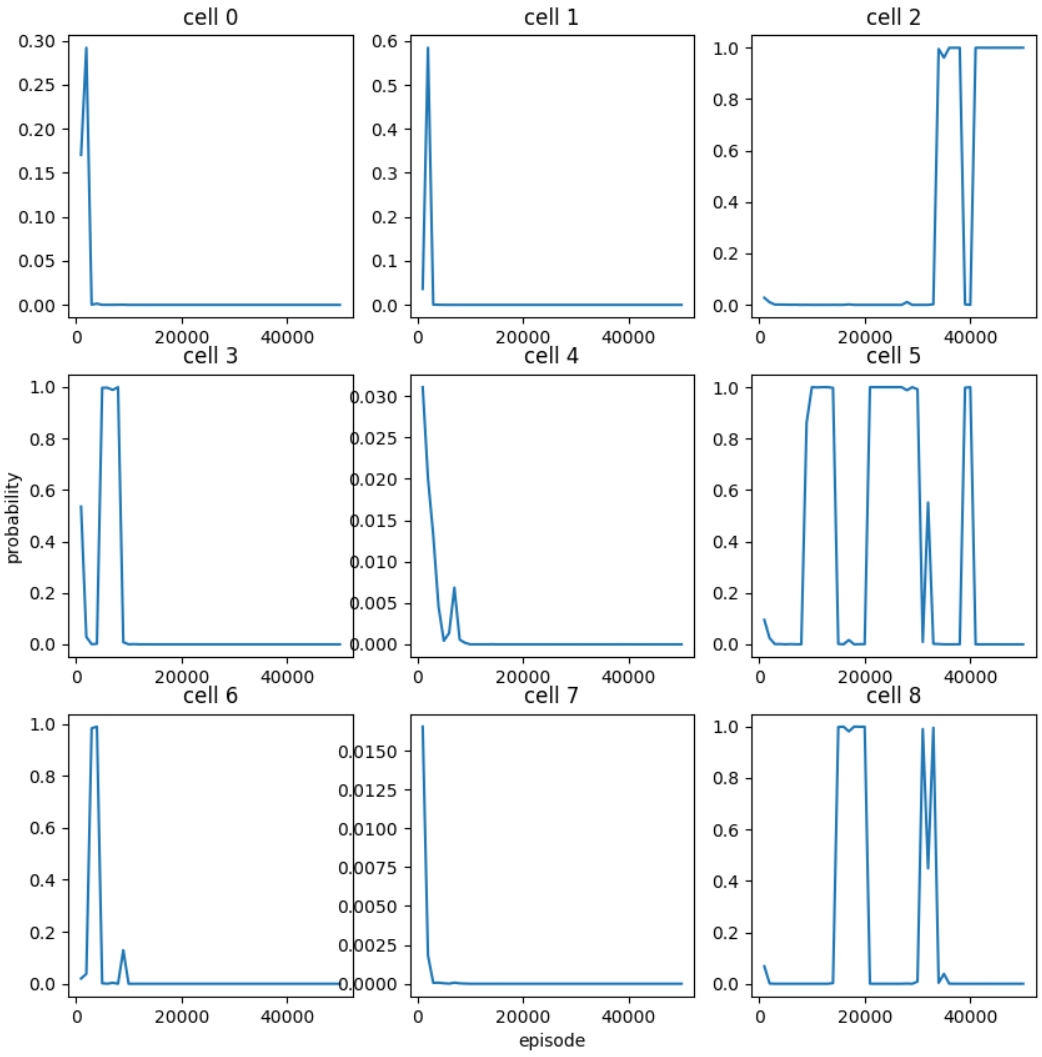
The evolution of win/loss and tie rates is shown in Figure 5. It is clear that the win rate generally increases towards the end of training while loss and tie rates decrease.

Figure 5

Part 7

Figure 6 displays how the probability that the first move will be a particular cell evolves with training. Intially, cells 1, 3 and 6 are chosen with the highest probability for first move, but their probabilities gradually go to zero with training. Cells 0, 4 and 7 always seem to have the lowest probabilities. Cells 2, 5 and 8 are preferred to be first moves as training progresses with cell 2 being the best choice at the end of training. This is consistent with the sample games displayed in part 5(d).

Figure 6: Evolution of first move probability for each of the 9 cells of the tic-tac-toe board



Part 8

The states for the two un-won games are shown below.

```
Game: 64
..x
.o.
...
====
..x
.oo
..x
====
x.x
ooo
..x
====
!!!!GAME 64 LOST!!!!
```

```
Game: 69
..x
.o.
...
====
..x
.oo
..x
====
xox
.oo
..x
====
xox
ooo
x.x
====
!!!!GAME 69 LOST!!!!
```

For game 64, the agent failed to block cell 3 (middle left) and prevent the random player from winning. Again, in game 69, the agent failed to block the same cell. The agent seems to prefer placing crosses on the corner cells.