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

Due on Monday, March 19, 2018

Dhruv Chawla, Saila Maham Shama

March 24, 2018

Part 1

Tic-Tac-Toe text output

```
>>> 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 indicat 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 s

Part 4

4(a)

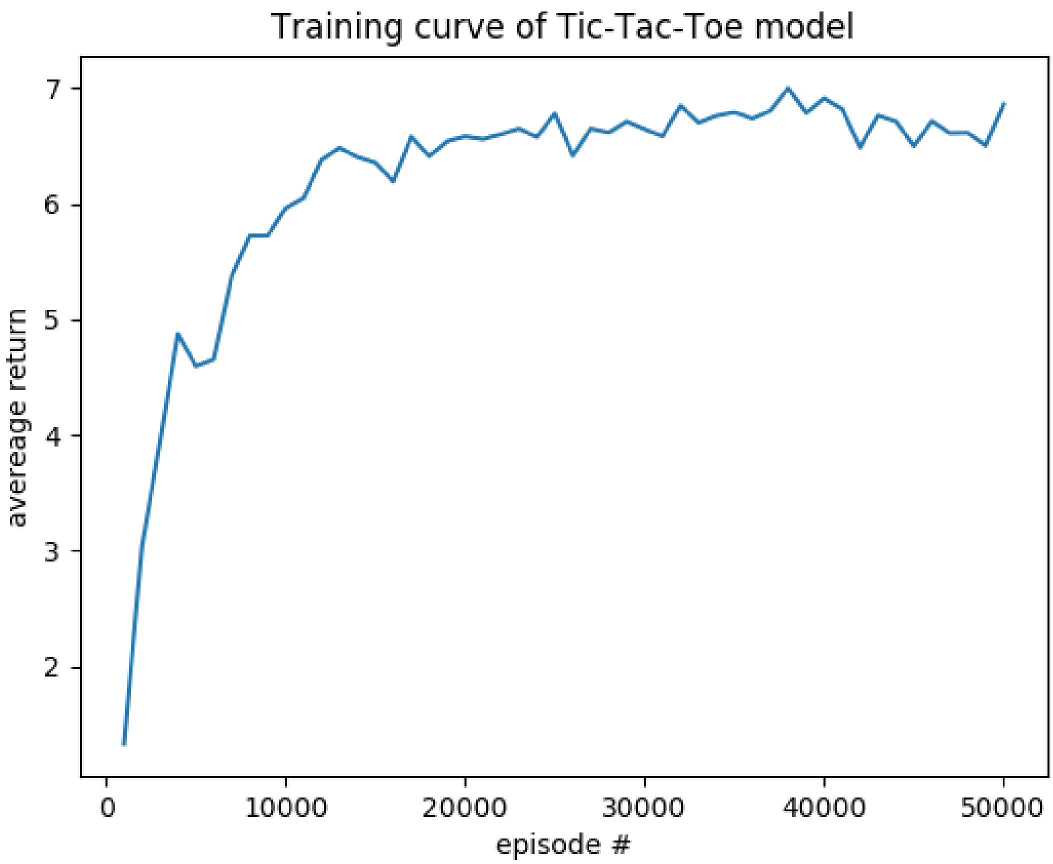
```
def get_reward(status):
"""Returns a numeric given an environment status."""
return {
    Environment.STATUS_VALID_MOVE  : 1,
    Environment.STATUS_INVALID_MOVE: -1,
    Environment.STATUS_WIN         : 5,
    Environment.STATUS_TIE         : 2,
    Environment.STATUS_LOSE        : -3
    }[status]
```

4(b)

Part 5

5(a)

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 following training curves:

5(c)

From Figure **??**, it seems as though the policy never played invalid moves.

5(d)

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

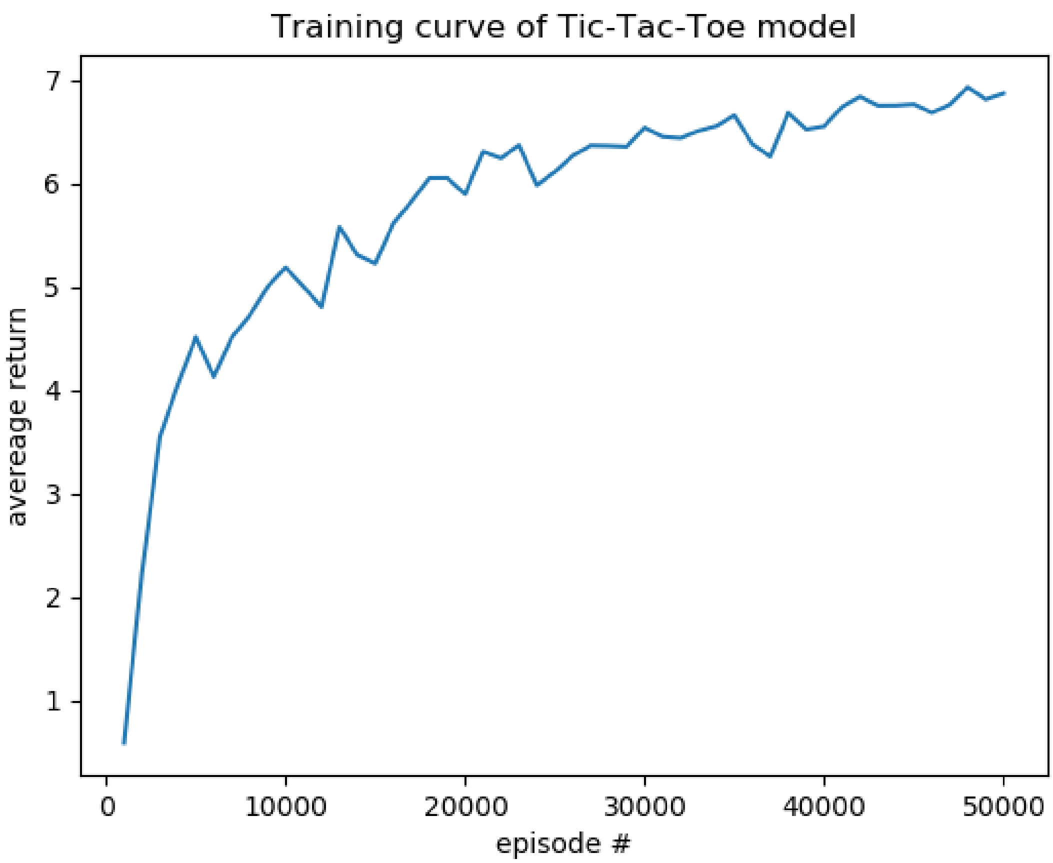


Figure 3: Training curve (# hidden units: 128)

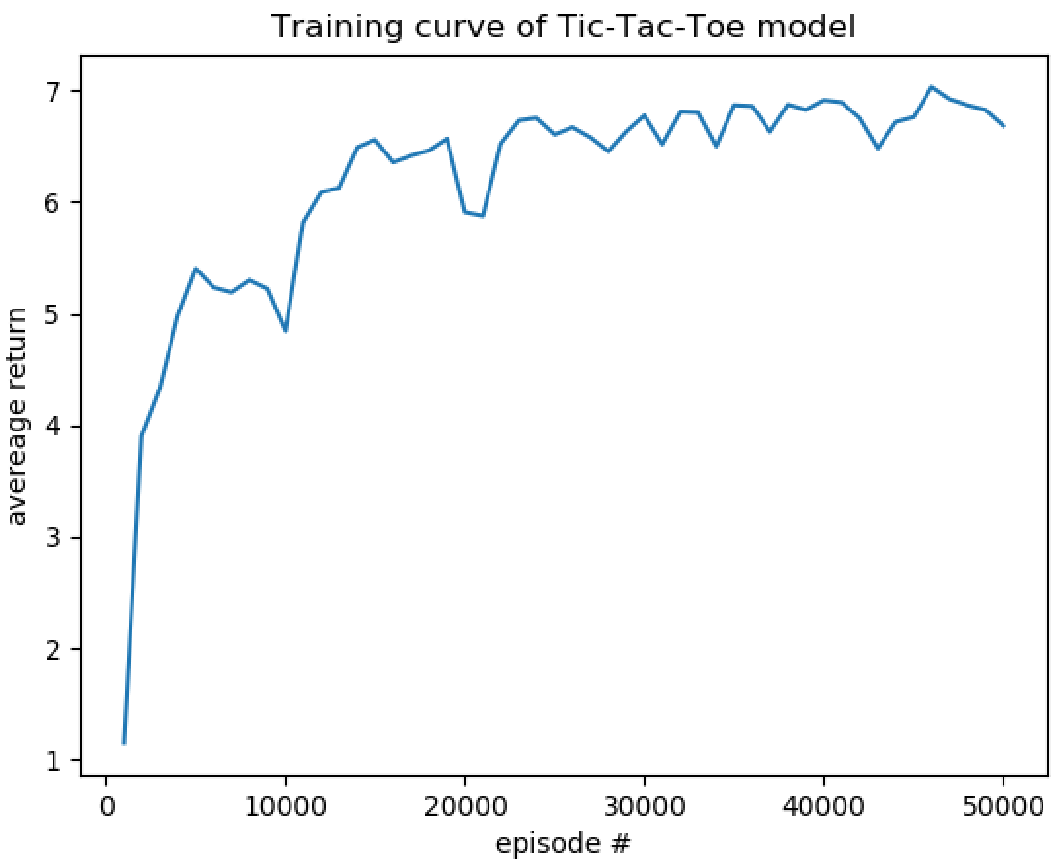


Figure 4: Training curve (# hidden units: 256)