CSC411: Project 4

Due on Sunday, April 2, 2018

Ying Yang

April 3, 2018

Part 1

Environment

The grid is represented by horizontal, vertical and diagonal indices. The coordinates are the indices from 0 - 8. The win-set is represented by a fix matrix in which the first row represents 3 sets of horizontal rows for which a player wins the game, the second row represent the vertical column, and the third row represents the diagonal of the tictactoe.
The attributes turn represents whose turn to play the game (player1 or player 2). The attribute done means whether the game is over (when a player wins, or a tie occurs).
Play a game of TicTacToe by calling the step(), and render() methods.

Listing 1: Compute network function

```
env.step(0)
(array([1, 0, 0, 0, 0, 0, 0, 0, 0]), 'valid', False)
env.render()
x..
...
...
====
env.step(2)
(array([1, 0, 2, 0, 0, 0, 0, 0, 0]), 'valid', False)
env.render()
x.o
...
...
====
env.step(4)
(array([1, 0, 2, 0, 1, 0, 0, 0, 0]), 'valid', False)
env.render()
x.o
.x.
...
====
env.step(8)
(array([1, 0, 2, 0, 1, 0, 0, 0, 2]), 'valid', False)
env.render()
x.o
.x.
..o
====
env.step(6)
(array([1, 0, 2, 0, 1, 0, 1, 0, 2]), 'valid', False)
env.render()
x.o
.x.
x.o
====
env.step(1)
(array([1, 2, 2, 0, 1, 0, 1, 0, 2]), 'valid', False)
env.render()
xoo
.x.
```

```
x.o
====
env.step(3)
(array([1, 2, 2, 1, 1, 0, 1, 0, 2]), 'win', True)
env.render()
xoo
xx.
x.o
====
```

Part 2

Complete the implementation so that policy is a neural network with one hidden layer

Part 2 (a)

Listing 2: Policy implementation

```
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__()
# TODO
self.linear_f1 = nn.Linear(input_size, hidden_size)
self.linear_f2 = nn.Linear(hidden_size, output_size)

def forward(self, x):
# TODO
h = F.relu(self.linear_f1(x))
out = F.softmax(self.linear_f2(h))
return out
```

Part 2 (b)

Listing 3: Policy

```
policy = Policy()
state = np.array([1,0,1,2,1,0,1,0,1])
state = torch.from_numpy(state).long().unsqueeze(0)
state = torch.zeros(3,9).scatter_(0, state, 1).view(1, 27)
print(state)
```

Listing 4: output

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

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

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

State what each of the 27 dimensions mean.

The 27 dimensions is formatted as a 3 X 9 matrix.
Each column index represents the position of grid.

Part 2 (c)

Explain what the value in each dimension means. The values in each dimension are the probabilities of the agent's next move to that position.
The policy is stochastic since the agent uses a random policy to make its next move.

Part 3

Part 3a

Implement the compute_returns function

Listing 5: output

```
    l = len(rewards)
    rewards = np.array(rewards)
    gammas = np.array([gamma ** (i) for i in range(l)])

    G = []
    for i in range(l):
        G.append(sum(rewards[i:] * gammas[:l - i]))
    return G
```

Part 3b

Explain why can we not update weights in the middle of an episode

When we are in the middle of the episode, we haven't yet finished computing the final result for reward. As a result, if we compute backward pass before we got the reward result, we could get inaccurate result, since if we compute the gradient based on inaccurate number. Therefore, we should update the weights after the episode.

Part 4

Part 4(a)

Listing 6: modified function

```
def get_reward(status):
"""Returns a numeric given an environment status."""
return {
Environment.STATUS_VALID_MOVE: 1,
Environment.STATUS_INVALID_MOVE: -25,
Environment.STATUS_WIN: 50,
Environment.STATUS_TIE: 0,
Environment.STATUS_LOSE: -50
}[status]
```

Part 4(b)

Explain the choices that you made in 4(a)

The environments status were given weight based on the principle of rewarding biggest on win and penalize biggest on lose. In the same way, reward on status that contribute to valid moves and penalize on status that contribute to invalid moves. As a result, valid move is considered as a good move, and was given a positive number: 1. The invalid move is bad, so it should be penalized, and it was given: -25. The win status should be rewarded big, so it was given a big positive number: 50. The tie status is given 0 since it doesn't contribute to win or lose. Finally, the lose status was given -50 to get penalized.

Part 5

Part 5a

Plot the training curve of the tictactoe model.
Hidden units = 64 units

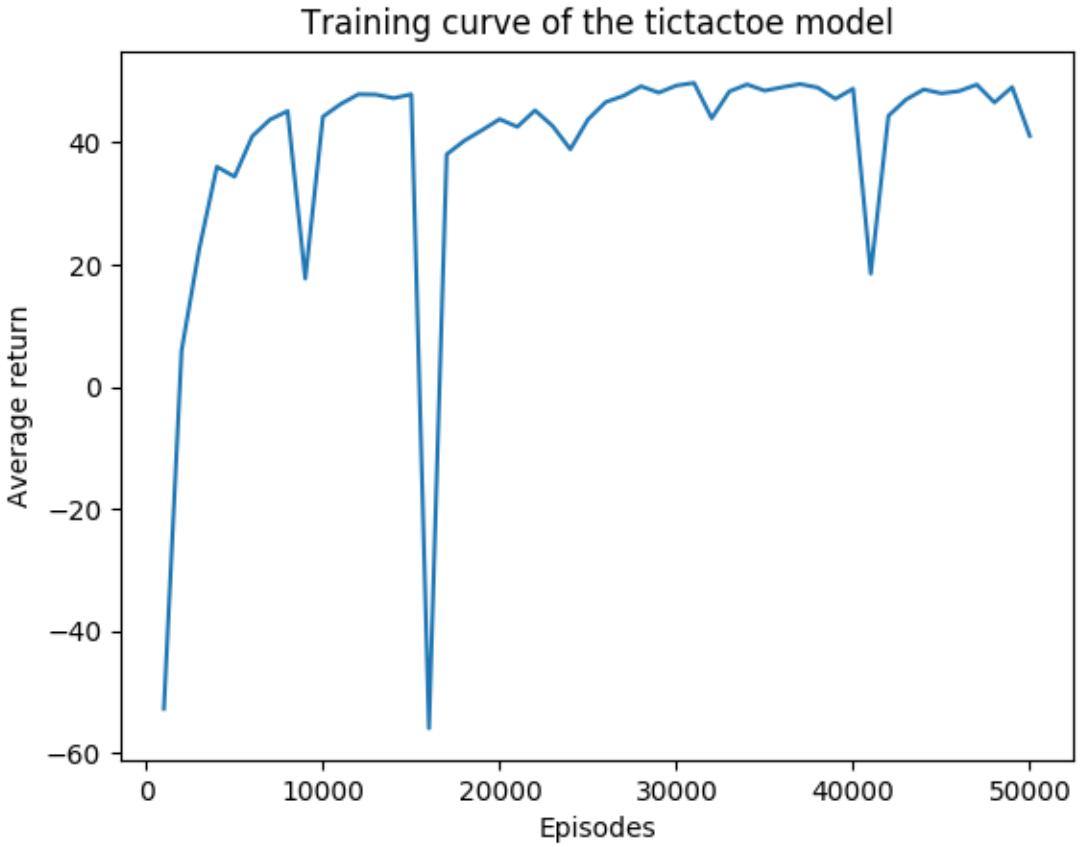


Figure 1: training curve of the tictactoe model, with 64 hidden units

Part 5b

Plot the training curve of the tictactoe model, by tuning the hyperparameter hidden units with 3 different values.

1. Hidden units = 16 units

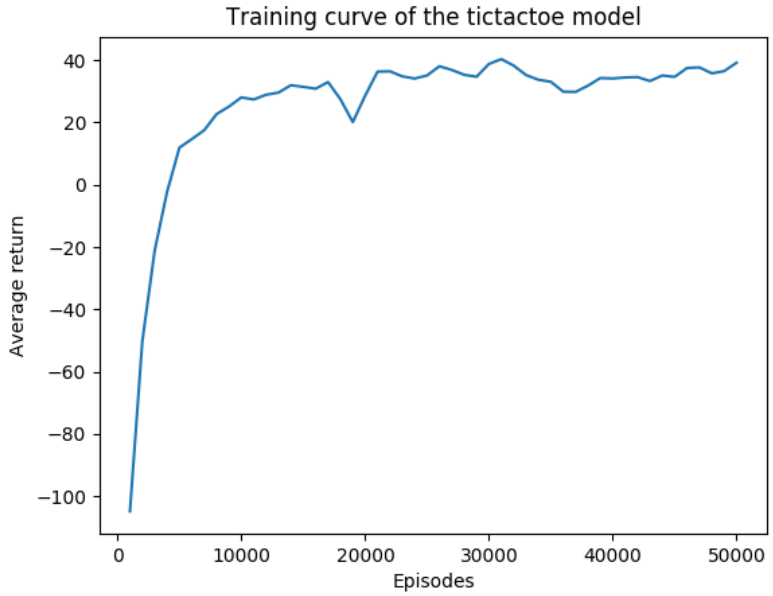


Figure 2: training curve of the tictactoe model, with 16 hidden units

2. Hidden units = 32 units

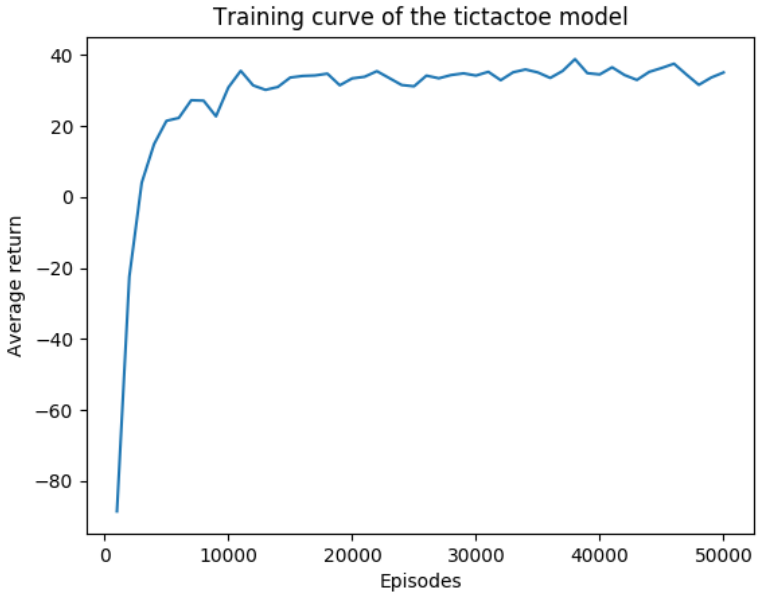


Figure 3: training curve of the tictactoe model, with 32 hidden units

3. Hidden units = 128 units

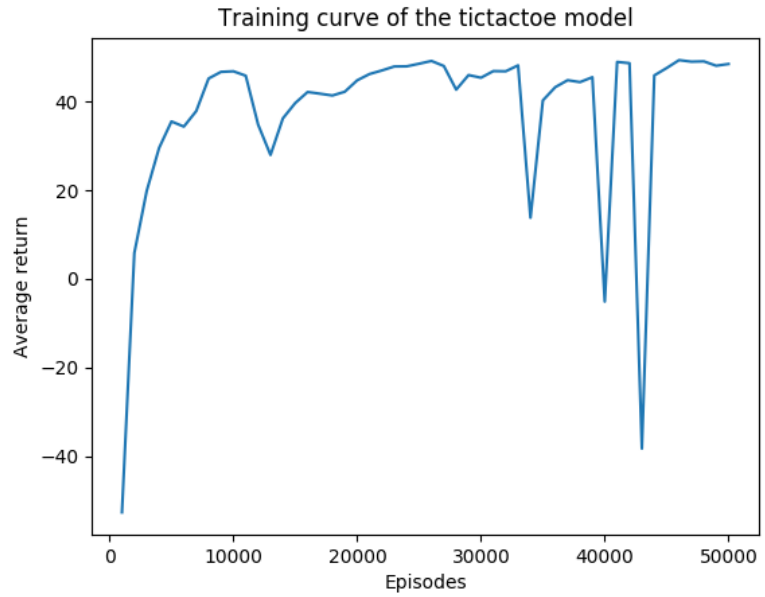


Figure 4: training curve of the tictactoe model, with 128 hidden units

4. Hidden units = 256 units

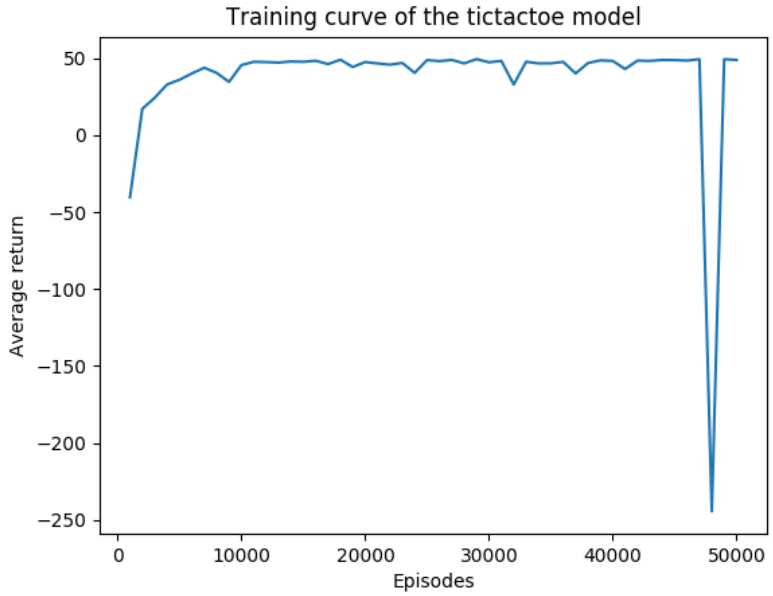


Figure 5: training curve of the tictactoe model, with 256 hidden units

After experimenting with 3 different number of hidden units : 16, 32, 128 and 256, it was possible to observe that the hyperparameter setting to 256s gave the best results. For hidden units 16 and 32, the average return was below 40, for 128, the average return was varying in the 40s and for 256, the average return almost reached 50.

Part 5c

One of the first things that your policy should learn is to stop playing invalid moves. At around what episode did your agent learn that? State how you got the answer.
The agent learned to stop playing invalid moves at around 4000th iteration. This can be observed by counting the number of invalid moves for 100 random games. By inspecting the result, starting at 3000th iteration, the number of invalid moves dramatically decreased from 126 invalid moves to only 27 invalid moves.

Part 5d

Use your learned policy to play 100 games against random.

How many did your agent win / lose / tie?
Results for playing 100 games against random
wins: 96 losses: 3 ties:1
Display five games that your trained agent plays against the random policy.

Listing 7: modified function

```
Game 1
...
.x.
o..
====
..o
.xx
o..
====
..o
xxx
o..
====
wins: 1 losses: 0       ties:0

Game 2
...
.xo
...
====
...
.xo
xo.
====
..x
.xo
xo.
====
wins: 1 losses: 0       ties:0

Game 3
..o
.x.
...
```

```
====
..o
.xx
o..
====
..o
xxx
o..
====
wins: 1 losses: 0       ties:0

Game 4
...
.x.
..o
====
.o.
.x.
x.o
====
.ox
.x.
x.o
====
wins: 1 losses: 0       ties:0

Game 5
.o.
.x.
...
====
.o.
.x.
xo.
====
.ox
.x.
xo.
====
wins: 1 losses: 0       ties:0
```

Explain any strategies that you think your agent has learned.

By examining the steps that agent made, it is possible to see that the agent might learned to prioritize to make the step that allows to win the game fastest. For example, if it makes a first move without having the opponent blocking one of its winning lines (vertical, horizontal, diagonal), it will continue making move towards that line until the opponent blocks it.
Also, it might learned to avoid making useless moves such as making a move that doesn't block the opponent, or making a move that is obviously a dead move, that is useless. (such case figure below:)
before :
o . .
x . o
. . .

After :
o . .
x x o
. . .
Finally, it always starts by picking the middle of the board, and this might be a learned strategy to more easily win the game.

Part 6

Part 6a

Use the model checkpoints saved throughout training to explore how the win / lose / tie rate changed throughout training
Graph of win/lose/tie rate changed throughout the training

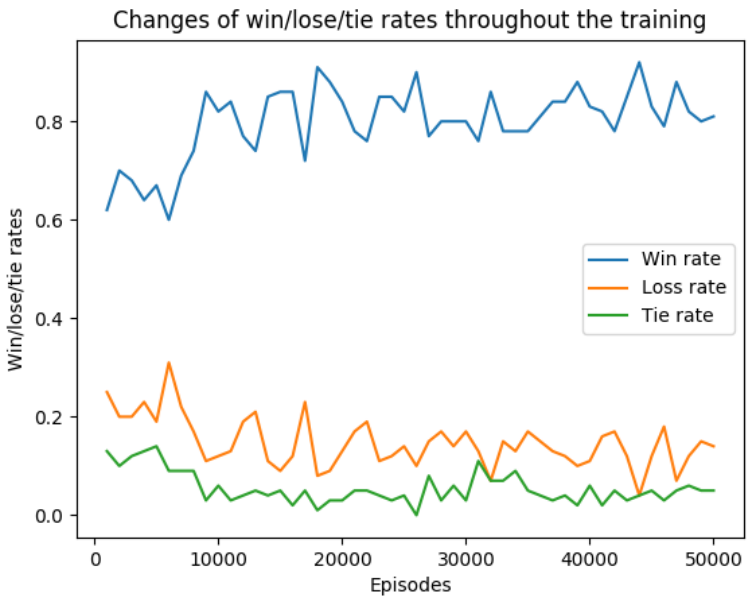
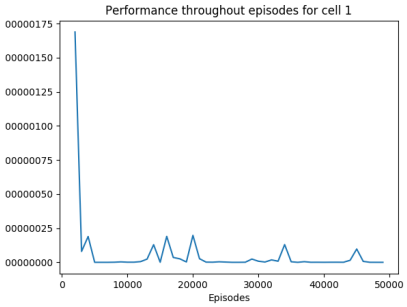


Figure 6: win/lose/tie rate changed throughout the training

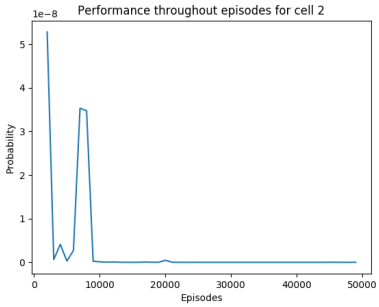
From the graph, we can see that the win rate improved from around 62% to around 81%. Meanwhile, the lose rate and tie rate decreased from around 23% to 15%., 13% to 5% respectively. This shows that the agents has learned to make moves that rewards it for winning the game.

Part 7

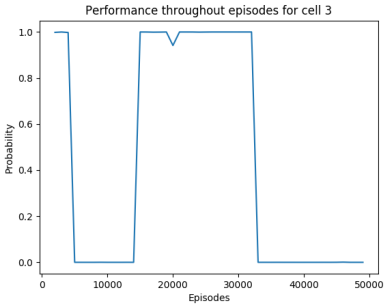
From the graphs, it is possible to observe that the probabilities for making first moves in cells 1, 2, 4, 5 were higher. As the model is trained more and more and the agent learned to win, the probabilities were higher for cells 3, 7 and 9, which makes sense. From the game play in part 6, it was observed that the agent always make the first move in the cell 5, and this is shown in the distribution where cell 5 got the highest probability compared to 1, 2 and 4. As a result, the model makes sense.



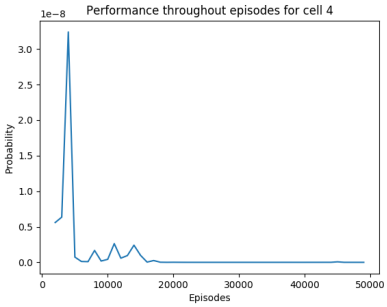
(a) Cell 1



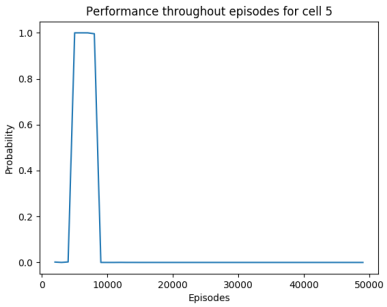
(b) Cell 2



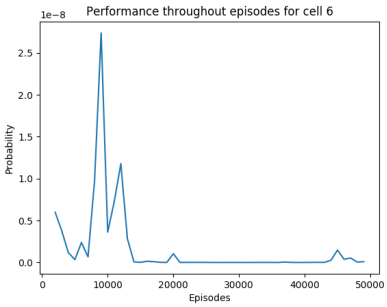
(c) Cell 3



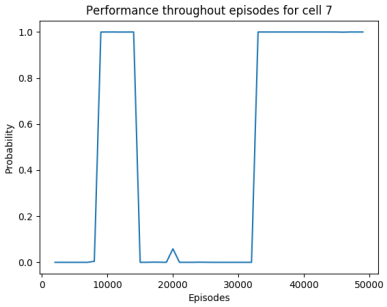
(d) Cell 4



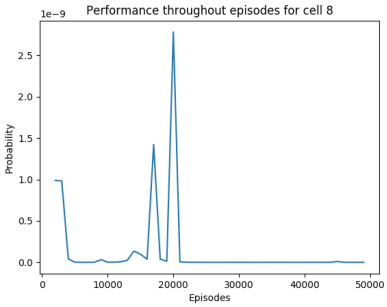
(e) Cell 5



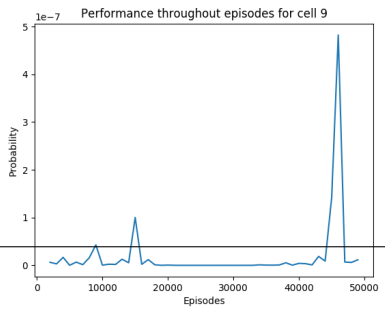
(f) Cell 6



(g) Cell 7



(h) Cell 8



(i) Cell 9

Part 8

Your learned policy should do fairly well against a random policy, but may not win consistently. What are some of the mistakes that your agent made?
Examples of mistakes that the agent made were it didn't prevent the opponent from winning by making random moves when the opponent had two same placements already.