# CSC 411: Assignment #4

**Xiangyu Kong** **Yun Lu**
**kongxi16** **luyun5**

March 30, 2018

# Problem 1

In class "Environment", the "grid" is represented by an 1-D array of 9 elements. Attribute "turn" represents whose turn it is. The valid values for "turn" is either 1 (x) or 2 (o). Attribute "done" represents whether the game is done.
The code and output of a game is shown in the listing below.

Listing 1: Code and Output of a game

```
for i in range(9):
        env.step(i)
        env.render()
return
x..
...
...
=====
xo.
...
...
=====
xox
...
...
=====
xox
o..
...
=====
xox
ox.
...
=====
xox
oxo
...
=====
xox
oxo
x..
=====
xox
oxo
x..
=====
xox
oxo
x..
=====
```

# Problem 2

1. The Policy class is implemented in the listing below. The network consist of one hidden layer and the activations used are ReLU activation.

   Listing 2: policy

```
class Policy(nn.Module):
        def __init__(self, input_size=27, hidden_size=64, output_size=9):
                super(Policy, self).__init__()
                self.network = nn.Sequential(
                        nn.Linear(input_size, hidden_size),
                        nn.ReLU(),
                        nn.Linear(hidden_size, output_size),
                        nn.ReLU())
        def forward(self, x):
                return self.network(x)
```

2. The 27-dimensional vector should be viewed as a $3 \times 9$ matrices with each column as a one-hot vector. The column number represents the position on the grid (1st column represent Environment.grid[0]), and the vector itself represents the state of the current position. The state correspond to the map in Environment.render() ({0: '.', 1: 'x', 2: 'o'}).

3. The 9 values Policy returns are the probabilities that it chooses the corresponding next move (e.g. the first element represents the probability that the next move is chosen to be placed at the first element in Environment.grid).

   The select_action samples the action according to the probabilities (stochastic) instead of choosing the maximum probability (deterministic). Then the policy is stochastic.

# Problem 3

1. the implementation for compute_returns is shown in the listing below.

Listing 3: compute return

```
def compute_returns(rewards, gamma = 1.0):
        res = []
        for i in range(len(rewards)):
                curr_return = 0
                cur_rewards = rewards[i:]
                for j in range(len(cur_rewards)):
                        curr_return += cur_rewards[j] * (gamma ** j)
                res.append(curr_return)
        return res
```

2. 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.

# Problem 4

1. See tictactoe.py for implementations.

2. The rewards are shown in the listing below. The win and lose rewards are $+100$ and $-100$ respectively. The reward for valid move is 5 and for invalid move is $-50$. This is because we do not want the network to make invalid moves and do not want the network to be satisfied with just making valid moves. Finally, we reward tie as $-10$. This is because getting a tied result is just a little bit better than losing, but still not what we wanted, so we discourage the policy to get a tied score.

Listing 4: Rewards

```
Environment.STATUS_VALID_MOVE: 5,
Environment.STATUS_INVALID_MOVE: -50,
Environment.STATUS_WIN: 100,
Environment.STATUS_TIE: -10,
Environment.STATUS_LOSE: -100
```

# Problem 5

1. The training curve is shown in Fig.1. The hyperparameters are as follows: lr = 0.0005, gamma = 0.9, max_iter = 50000. We set gamma to 0.9 to get a smoother and quicker result. We also set the learning rate to half to make sure the training does not get stuck. If the learning rate is too high, the weights will grow too fast and the policy will try to pick the same choice over and over again.

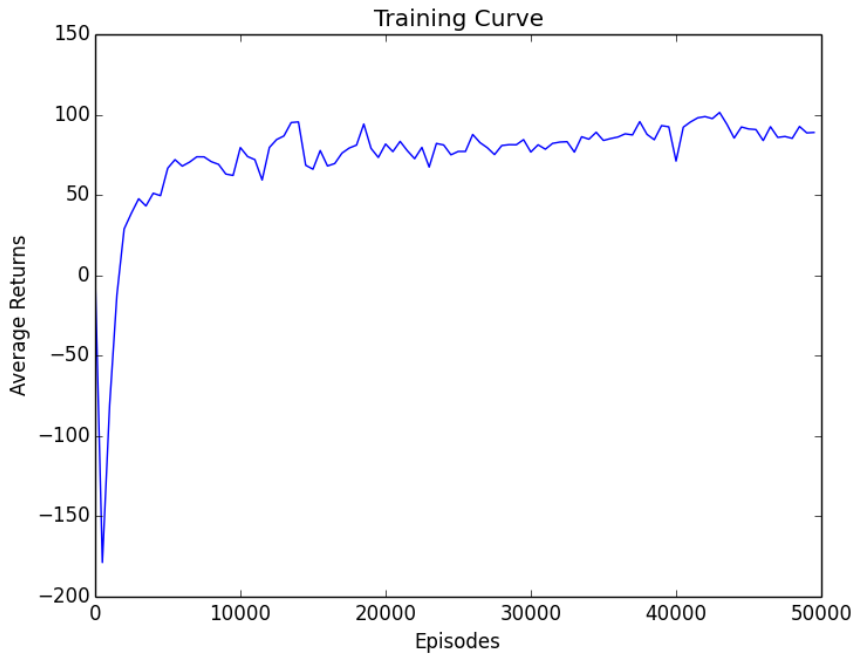



Figure 1: Training Curve

2. The hidden dimension was tested on values [32, 64, 128, 256]. Their results were calculated using function get_policy_result. The best-performing dimension was 256. It has 946 wins out of 1000 round.

3. The policy starts to understand not to make invalid moves at around the 5000th iteration. To get this result, we let the policy play with random for 100 times at each log point and count the number of invalid move and total move and thus calculate the percentage of the invalid move the policy makes. After the 5000th iteration, all iterations' invalid move rates are less than 3%, so the policy understands not to make invalid moves.

4. By looking at the game plays, we can see that the policy is capable of understanding how to block the opponent from making a winning-move. While the policy is doing that, it also tries to place the crosses at a place that is advantageous.

```
========== Game 0 ==========
. . x
. . .
. o .
=====
. ox
. . .
xo .
```

```
=====
. ox
. x .
xo .
=====
========== Game 1 ==========
o . x
. . .
. . .
=====
o . x
. . o
. x .
=====
o . x
o . o
xx .
=====
o . x
oxo
xx .
=====
========== Game 2 ==========
. . .
o . .
. x .
=====
. . x
o . o
. x .
=====
. ox
oxo
. x .
=====
. ox
oxo
xx .
=====
========== Game 3 ==========
. . x
. . .
o . .
=====
. . x
. x .
o . o
=====
. . x
```

```
xx.
ooo
=====
========== Game 4 ==========
..x
..o
...
=====
..x
.xo
o..
=====
..x
.xo
oox
=====
x.x
.xo
oox
=====
```

# Problem 6

The performance for different stages are shown in Fig.2. From the graph we can clearly see that the winning rate increases from less than 60% to around 80% and the lose and tie rates gradually decrease. This is because we want the network to maximize reward and we only set the reward for winning to be positive, so the network learns to win more.

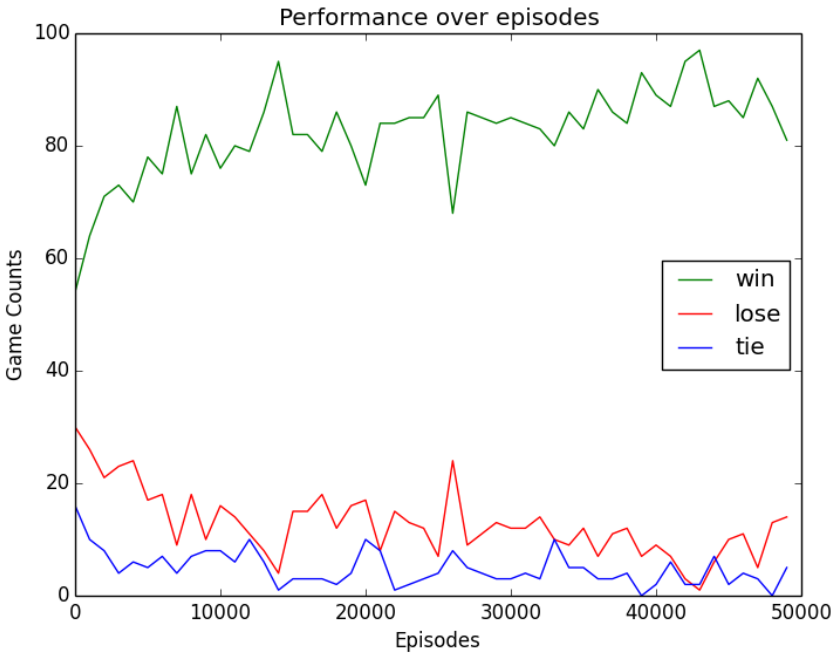
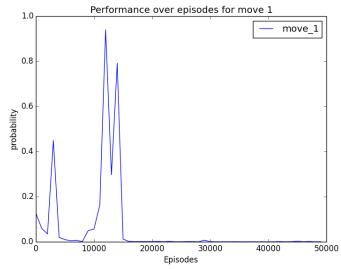


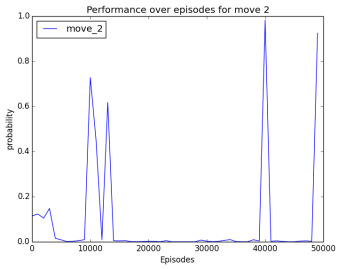

Figure 2: Performance over time

# Problem 7

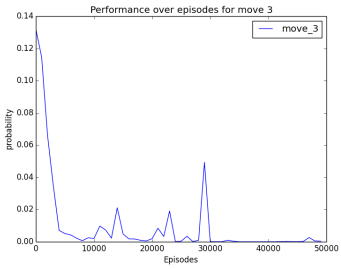
The final first move distribution is [0.002, 0.0061, 0.0003, 0.0, 0.0, 0.9859, 0.0035, 0.002, 0.0002]. We can clearly see that the network understands that it should play the first move at slot 6 (grid[2, 3]).
The distribution throughout the training process is shown in Fig.3.



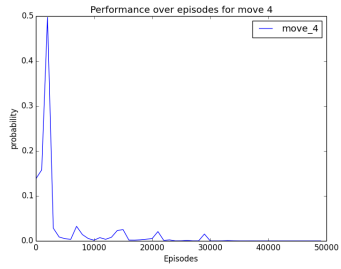

(a) Distribution for position 1



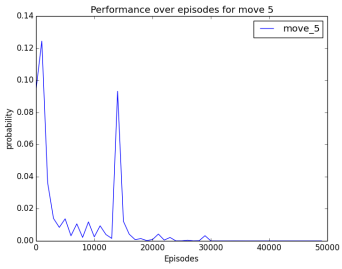

(b) Distribution for position 2



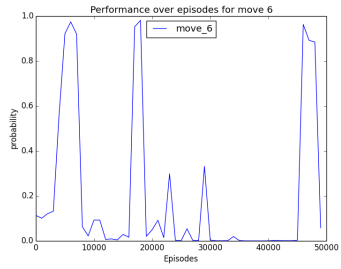

(c) Distribution for position 3



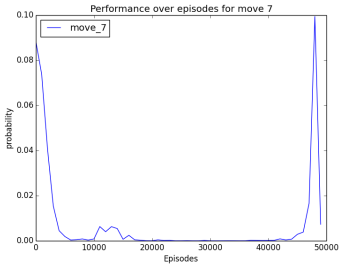

(d) Distribution for position 4



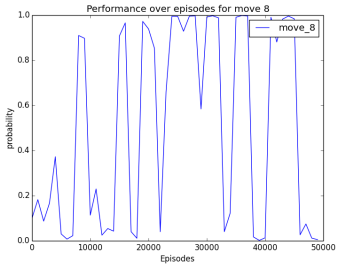

(e) Distribution for position 5



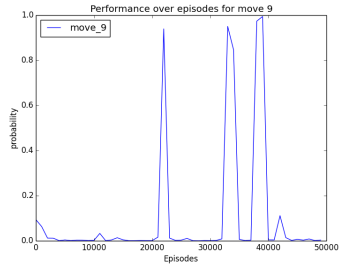

(f) Distribution for position 6




(g) Distribution for position 7




(h) Distribution for position 8




(i) Distribution for position 9

Figure 3: First move distributions

# Problem 8