CSC411 Project 4

Zikun Chen
1001117882

April 2, 2018

Part 1

The grid is represented in a numpy array of length 9 representing the 9 positions of the 3 by 3 tic-tac-toe grid. The array is stored in the self.grid attribute within the Environment class. Each element in the array is 0 (position unoccupied), 1 (occupied by 'x') or 2 (occupied by 'o').

The render function visualizes the grid by giving a text output of the grid in the console with rows, columns, x's and o's being represented.

The attribute turn (1 or 2) indicates which player's turn it is at the current game state.

The attribute done is a Boolean variable indicating whether the game is finished or not, when there is a winner or a tie (all position on the grid is occupied and there is no winner).

Text Output (Tie):

```
...
...
...
====


...
.x.
...
====


..o
.x.
```

```
...
====


..o
.x.
..x
====


o.o
.x.
..x
====


oxo
.x.
..x
====


oxo
.x.
.ox
====


oxo
xx.
.ox
====


oxo
xxo
.ox
====


oxo
xxo
xox
====
```

Part 2

(a) **Policy Code:**

```
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.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size

        self.features = nn.Sequential(
            nn.Linear(input_size , hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, output_size),
        )

    def init_weights(self):
        dtype_float = torch.FloatTensor
        dtype_long = torch.LongTensor

        np.random.seed(SEED)
        W0 = np.random.normal(0, 2/float(self.hidden_size + self.input_size),
                              size=(self.hidden_size, self.input_size))

        np.random.seed(SEED)
        b0 = np.random.normal(0, 1/float(self.hidden_size),
                              size=self.hidden_size)

        np.random.seed(SEED)
        W1 = np.random.normal(0, 2/float(self.hidden_size + self.output_size),
                              size=(self.output_size, self.hidden_size))

        np.random.seed(SEED)
        b1 = np.random.normal(0, 1/float(self.output_size),
                              size=self.output_size)

        self.features[0].weight.data = torch.from_numpy(W0).type(dtype_float)
        self.features[0].bias.data = torch.from_numpy(b0).type(dtype_float)
        self.features[2].weight.data = torch.from_numpy(W1).type(dtype_float)
        self.features[2].bias.data = torch.from_numpy(b1).type(dtype_float)
```

```
def forward(self, x):
    x = self.features(x)
    x = nn.functional.softmax(x, dim = -1)
    return x
```

(b) At each of the 9 positions on the grid, there are 3 possible state. Either no one has occupied the position yet (visualized as '.'), player 1 occupied the position (visualized as 'x') or player 2 occupied the position (visualized as 'o'). In total, 27 dimensions that are used to represent any possible state of the game.

(c) The value in each dimension means the probabilities of choosing the corresponding position in the grid to occupy. And these probabilities are generated by the policy through out training.

The policy is stochastic because the softmax output transform the outputs of the neural net to probabilities (between 0 and 1) of playing at each position on the grid. We than sample an action according to those probabilities generated from the policy.

Part 3

(a) **Implementation:**

```
def compute_returns(rewards, gamma=1.0):
    reward_list = []
    for start in range(len(rewards)):
        reward_t = 0
        power = 0
        for t in range(start, len(rewards)):
            reward_t += gamma ** power * rewards[t]
            power += 1
        reward_list.append(reward_t)
    return reward_list
```

(b) Because the policy_loss in finish_episode function sum up the product of negative log probability and rewards across all time steps. Therefore, we need to generate a full episode before we can get the value of the policy_loss, which will then be used for backpropagation for updating the weights.

Part 4

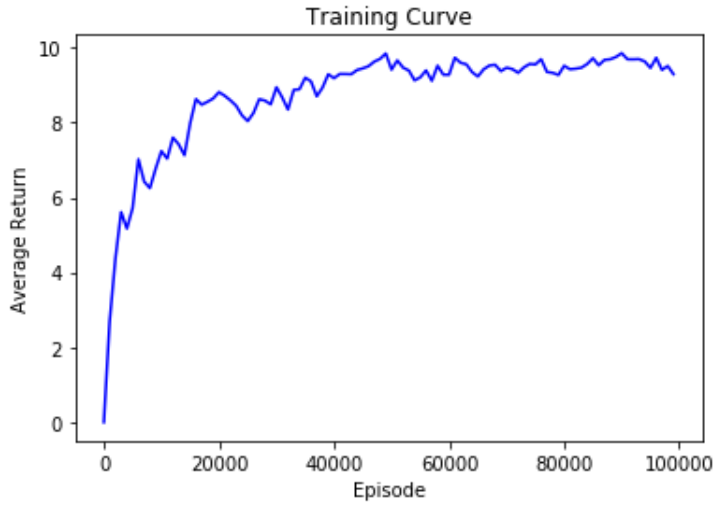
(a) **Modified Function:**

```
def get_reward(status):
    """Returns a numeric given an environment status."""
    return {
            Environment.STATUS_VALID_MOVE  : 0.1,
            Environment.STATUS_INVALID_MOVE: -0.5,
            Environment.STATUS_WIN         : 10,
            Environment.STATUS_TIE         : 0,
            Environment.STATUS_LOSE        : -5
    }[status]
```

(b) I chose 10 for winning rewards and 0.1 for performing a valid move. It makes sense to give some reward to the program when it makes a valid move so that it is going to learn to play valid moves. The winning reward is way larger than valid move reward and it is also double the magnitude of the losing reward. Because we don't want the model to just learn how to play valid moves, more importantly, we want to encourage it a lot to win the game. On the other hand, I chose losing reward to be -5 and invalid move reward -0.5 to discourage those outcomes during the training. -0.5 has a larger magnitude than 0.1, this is because it is important for the model to learn that invalid moves are unacceptable. I chose tie to have zero reward which is between winning and losing. This is because tie is not the best outcome although it is more desirable than losing.

Part 5

(a) **Training Curve:**



I have changed gamma to be 0.9 to take into the account the discounting

of the total rewards so that rewards far away in the future count for less. This is because getting a reward now is better than getting the same reward later on. If the agent chooses a longer path to win the game, there is a higher chance for the opponent to disrupt that path. Therefore, we want to discourage that with a gamma smaller than 1.

(b) I have tried 45, 55, 64, 75, and 100 as the number of hyperparameters. Note here, I did not set seed for training these models because for certain model, training gets stuck at a particular episode. I had to restart the training again with a different sampling sequences of moves produced by policy and random. To reproduce the trained models, we can load weights from the pickle files in corresponding folders. The following summary table shows the best average return for each model and at which episode the model achieved it. (Be ware if the training is ran again, the stored weights in pickle files will be replaced, therefore this part of the program is commented out.)

e.g. To retrieve weights for a certain hidden layer size, use:

```
policy = Policy(hidden_size = 55)
load_weights(policy, 90000)
```

Summary:

Hidden Layer Size	45	55	64	75	100
Best Episode	87000	90000	95000	37000	57000
Best Average Return	9.668	9.852	9.759	9.730	9.651
No Invalid Moves at	9000	2000	5000	13000	5000

Therefore, I chose the model with the best average return with hidden size of 55.

(c) At around 2000 episode, the policy (55 hidden units) stopped playing invalid moves:

```
Episode 0       Average return: -0.00   Invalid Moves: 6
Episode 1000    Average return: 2.69    Invalid Moves: 2
Episode 2000    Average return: 4.36    Invalid Moves: 0
Episode 3000    Average return: 5.61    Invalid Moves: 0
Episode 4000    Average return: 5.17    Invalid Moves: 0
Episode 5000    Average return: 5.72    Invalid Moves: 0
```

I modified the train function so that every time we play_against_random(action) where action is produced from the policy, we check the "status" output and if the action is invalid then we increase invalid_count by 1. Therefore, we aggregate the number of invalid moves and print them out for each checkpoint during training.

(d) **Out of 100 games:**

```
Win: 98 Loss: 1 Tie: 1
```

Display 5 games:

```
Game 1 - No. 12


..x
o..
...
====


.xx
o..
o..
====


xxx
o..
o..
====

Game 2 - No. 27


..x
..o
...
====


.xx
..o
o..
====


xxx
..o
o..
====

Game 3 - No. 55
```

```
..x
.o.
...
====


oxx
.o.
...
====


oxx
.oo
..x
====


oxx
ooo
x.x
====
```

Game 4 - No. 58

```
..x
...
.o.
====


oxx
...
.o.
====


oxx
.xo
.o.
====
```

```
oxx
.xo
xo.
====

Game 5 - No. 65


o.x
...
...
====


o.x
ox.
...
====


o.x
ox.
x..
====
```

Strategy:

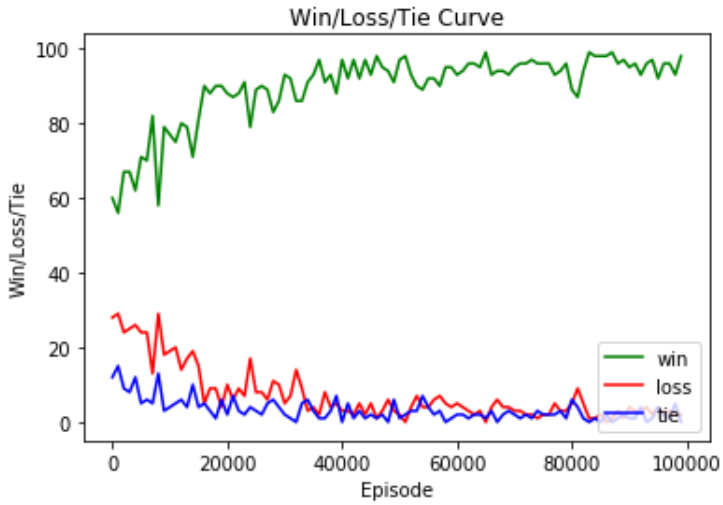
The policy always picks the top right position to start. When the random move opponent does not choose any position on the first row throughout the game, the policy will occupy the first row until it wins. (Game 1 and 2)

If the opponent chooses a position on the first row at any point during the game, the policy will try to form a diagonal win. (Game 4 and Game 5)

Game 3 is a loss, which we will discuss in part 8.

Part 6

Win/Loss/Tie Graph:



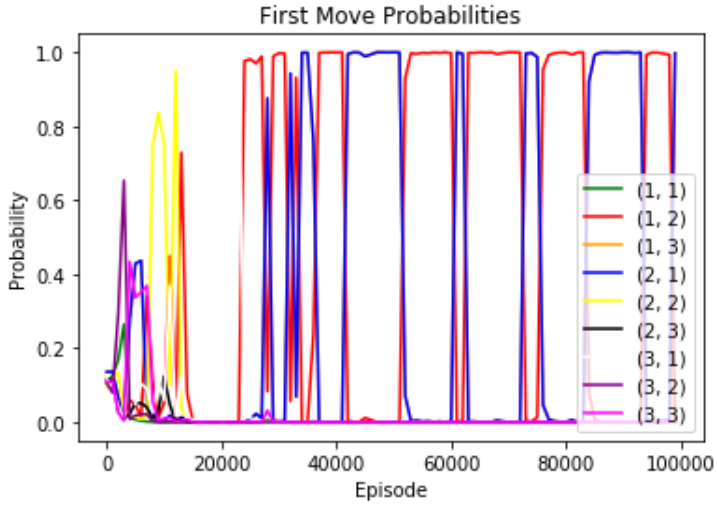
Part 7

Distribution over the First Move (90000 episode):

```
1.71291e-07
0.00049787
0.999501
1.24825e-08
1.08029e-06
3.33364e-12
7.1533e-11
1.36311e-08
```

Model has learned that to occupy the top right corner almost every time. This seems to be sub-optimal but it does make sense. Because we are training against a random strategy, so the trained first move will have a random position too. If we train against an better opponent, the policy will learn to occupy the centre position, otherwise the opponent will pick the centre position and gain an advantage. But in our case, the opponent has random strategy, so there is no guarantee that our policy will learn to play the centre position.

Distribution Throughout Training:



Part 8

Limitations:

Since the policy will pick the top right corner most of the times as its first move due to its high probability, it will have a hard time winning if the opponent picks the centre position. The centre position as we know is the key to tic-tac-toe because it secures one position along all paths that allow us to win.

In Game 3 from part 5 d), the opponent is able to get an upper hand of the game when it occupied the centre position. Consequently, the policy is not able to block both the diagonal and the horizontal path effectively.