Homework Assignment - 5

Name: Sagar Patel

NETID: sp5894

Question 1

Policy gradients. In class we derived a general form of policy gradients. Let us consider a special case here. Suppose the step size is η. We consider where past actions and states do not matter; different actions a_i give rise to different rewards R_i.

1.a

Define the mapping π such that $\pi(a_i) = \text{softmax}(\theta i) \text{ for } i = 1, \ldots, k$, where k is the total number of actions and θ_i is a scalar parameter encoding the value of each action. Show that if action a_i is sampled, then the change in the parameters in REINFORCE is given by:
$\Delta\theta_i = \eta R_i(1 - \pi(a_i))$

Solution:

$\pi(a_i) = softmax(\theta_i)$

$\Rightarrow \theta_i = \frac{e^{\theta_i}}{\sum e^{\theta_i}}$

Differentiating this, we get $\Rightarrow \theta'_i = \theta_i - \frac{\partial}{\partial\theta_i} E[R_t]$

$\Rightarrow \theta'_i = \theta_i - \eta R(\tau)\frac{\partial}{\partial\theta_i} log\pi(\theta_i)$

Therefore, $\delta\theta_i = \eta R(\tau)\frac{\partial}{\partial\theta_i} log\pi(\theta_i)$ ..... (1)

$\frac{\partial}{\partial\theta_i} log\pi(\theta_i) = \frac{\partial}{\partial\theta_i} log(\frac{e^{\theta_i}}{\sum e^{\theta_i}})$

$\Rightarrow \frac{\partial}{\partial\theta_i}(loge^{\theta_i}) = \frac{\partial}{\partial\theta_i} log \sum e^{\theta_i}$

$\Rightarrow 1 - \frac{e^{\theta_i}}{\sum e^{\theta_i}}$

$\Rightarrow 1 - \pi(a_i)$ ..... (2)

Comparing (1) and (2), we get -

Therefore, $\Delta\theta = \eta R(\tau)(1 - \pi(a_i))$

1.b

Intuitively explain the dynamics of the above gradient updates.

Solution:

The gradient updates are inversely proportional to the value of action.

When the value of $\pi(a)$ is small, we are (very) far from minima and thus $(1 - \pi(a))$ is large and we end up taking bigger steps and when $\pi(a)$ is large we are close to minima thereby making $(1 - \pi(a))$ is small thus ensuring that we take smaller steps.

Question 2

Designing rewards in Q-learning. Suppose we are trying to solve a maze with a goal and a (stationary) monster in some location, and the goal is to reach the goal in the minimum number of moves. We are tasked with designing a suitable reward function for Q-learning. There are two options:

- We declare a reward of +2 for reaching the goal, -1 for running into a monster, and 0 for every other move.
- We declare a reward of +1.5 for reaching the goal, -1.5 for running into a monster, and -0.5 for every other move.

Which of these reward functions might lead to better policies?

Solution:

Both the reward functions will give the same results.

Discounted return:

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} + \ldots$$

For some policy π and state s, the value function could be given as:

$$V^{\pi}(s) = E_{\pi}[G_t | s_t = s]$$

Using the discounted reward equation, we have -

$$V^{\pi}(s) = E_{\pi}[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | s_t = s]$$

Adding a constant C to all rewards, we then get

$$V'^{\pi}(s) = E_{\pi}[\sum_{k=0}^{\infty} \gamma^k (R_{t+k+1} + C) | s_t = s]$$

$$\Rightarrow E_{\pi}[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} + C \sum_{k=0}^{\infty} \gamma^k | s_t = s]$$

$$\Rightarrow V^{\pi} + \frac{C}{1-\gamma}$$

The above equation shows how the reward changes if a constant offset is added. We can see that after adding the constant to ALL the rewards, there is no effect to the relative values of any states under ANY policies. Thus, if we change the first policy by subtracting 0.5 from any reward, it will not have any effect in choosing the optimal policy and both the reward(s) functions will yield the same results.

Question 3

Open the (incomplete) Jupyter notebook provided as an attachment to this homework in Google Colab (or other environment of your choice) and complete the missing items. Save your finished notebook in PDF format and upload along with your answers to the above theory questions in a single PDF.

In this exercise we will train a simple Q-network in TensorFlow to solve Tic Tac Toe.

In [1]:
```
import random
import collections
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
import matplotlib.ticker
%matplotlib inline
```

Hopefully everyone has played Tic Tac Toe at some point. Here is a [reminder (https://en.wikipedia.org/wiki/Tic-tac-toe)](https://en.wikipedia.org/wiki/Tic-tac-toe). Let us set up some helper functions to define the game itself. The typical board size is 3x3 but we will be general.

In [2]:
```
def new_board(size):
    return np.zeros(shape=(size, size))
```

In [3]:
```
def available_moves(board):
    return np.argwhere(board == 0)
```

In [4]:
```
def check_game_end(board):
    best = max(list(board.sum(axis=0)) +    # columns
               list(board.sum(axis=1)) +    # rows
               [board.trace()] +            # main diagonal
               [np.fliplr(board).trace()],  # other diagonal
               key=abs)
    if abs(best) == board.shape[0]:
        return np.sign(best)  # winning player, +1 or -1
    if available_moves(board).size == 0:
        return 0  # a draw (otherwise, return None by default)
```

Now, let's define our players. We will define three types of bots. A *random* player picks a random position in the board each move.

In [5]:
```
class Player():
    def new_game(self):
        pass
    def reward(self, value):
        pass

class RandomPlayer(Player):
    def move(self, board):
        return random.choice(available_moves(board))
```

A *boring* player always picks the *first* available position on the board (measured from top-left to bottom-right).

In [6]:
```
class BoringPlayer(Player):
    def move(self, board):
        return available_moves(board)[0]
```

We can simulate games by playing one bot vs another. The starting player is labeled +1.

In [7]:
```
def play(board, player_objs):
    for player in [+1, -1]:
        player_objs[player].new_game()
    player = +1
    game_end = check_game_end(board)
    while game_end is None:
        move = player_objs[player].move(board)
        board[tuple(move)] = player
        game_end = check_game_end(board)
        player *= -1  # switch players after each move
    for player in [+1, -1]:
        # the reward for wins is +1, and -1 for draws/losses
        reward_value = +1 if player == game_end else -1
        player_objs[player].reward(reward_value)
    return game_end
```

3.1 | Random vs. Random

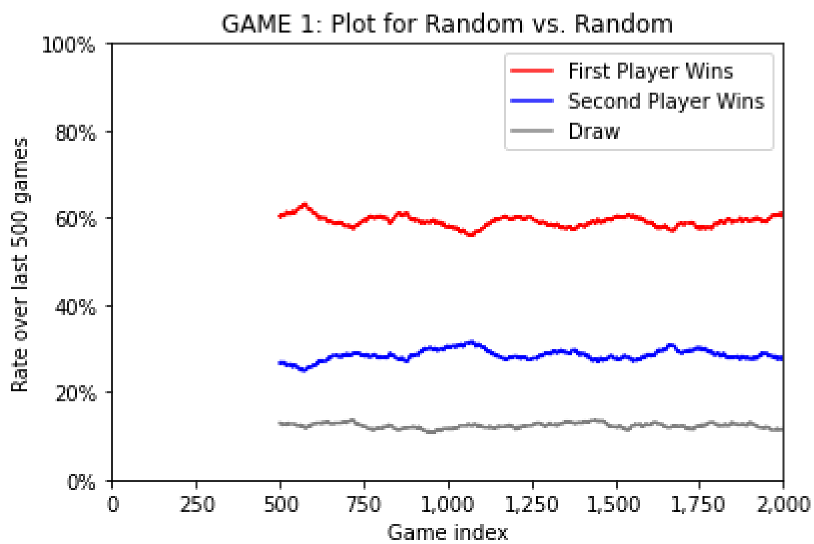
In [8]:

```
# 3x3, random vs. random
random.seed(1)

# TODO Q1. Play 2000 games between two bots, both of them random players.
# Print the number of wins by Player 1, number of wins by Player 2, and dra
# Plot (as a function of game index) the moving average of game outcomes
# over a window of size 500.
# You might find the following functions helpful for plotting.

games = 2000
results = [None]*2000

for i in range(games):
  results[i] = play(new_board(3), {+1: RandomPlayer(), -1: RandomPlayer()})

P1_win = 0
P2_win = 0
draw = 0

for result in results:
  if result == 1.0:
    P1_win += 1
  elif result == -1.0:
    P2_win +=1
  else:
    draw += 1

print("Player 1 Wins ", P1_win, " games")
print("Player 2 Wins ", P2_win, " games")
print("Number of Draw Games: ", draw)
```

```
Player 1 Wins  1194  games
Player 2 Wins  564  games
Number of Draw Games:  242
```

In [9]:
```
def moving(data, value=+1, size = 500): # calculates a moving average
    binary_data = [x == value for x in data]
    return [sum(binary_data[i-size:i])/size for i in range(size, len(data)

def show(results, size=500, title='Moving average of game outcomes',
         first_label='First Player Wins', second_label='Second Player Wins'
    x_values = range(size, len(results) + 1)
    first = moving(results, value=+1, size=size)
    second = moving(results, value=-1, size=size)
    draw = moving(results, value=0, size=size)
    first, = plt.plot(x_values, first, color='red', label=first_label)
    second, = plt.plot(x_values, second, color='blue', label=second_label)
    draw, = plt.plot(x_values, draw, color='grey', label=draw_label)
    plt.xlim([0, len(results)])
    plt.ylim([0, 1])
    plt.title(title)
    plt.legend(handles=[first, second, draw], loc='best')
    ax = plt.gca()
    ax.yaxis.set_major_formatter(matplotlib.ticker.PercentFormatter(xmax=1)
    ax.xaxis.set_major_formatter(matplotlib.ticker.StrMethodFormatter('{x:,
    plt.ylabel(f'Rate over last {size} games')
    plt.xlabel('Game index')
    plt.show()
```

In [10]:
```
show(results, title='GAME 1: Plot for Random vs. Random')
```

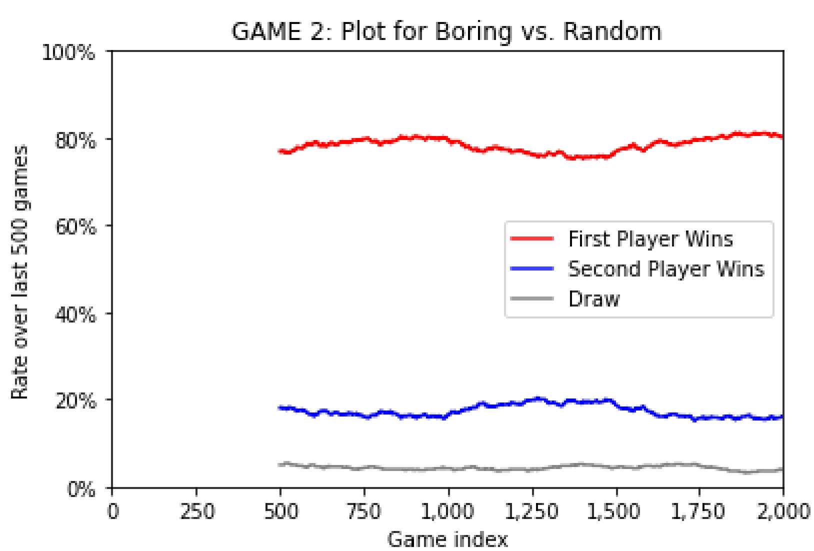


3.2 | Boring vs. Random

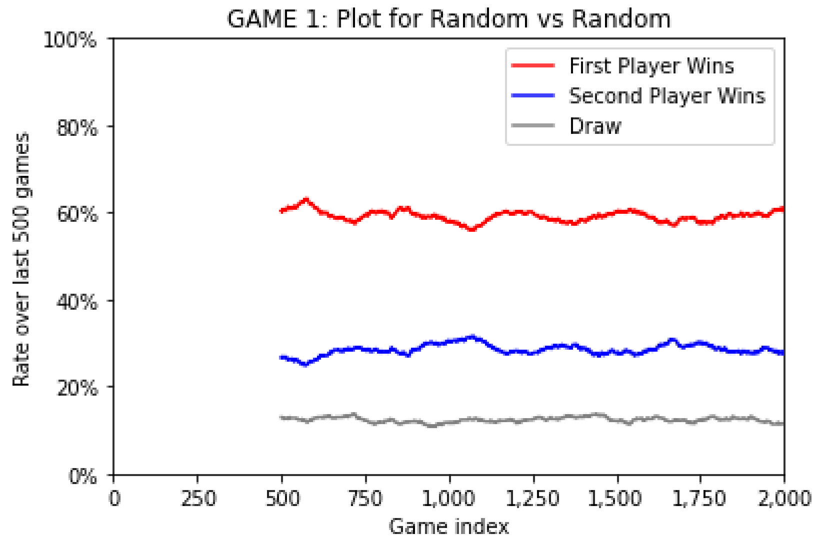
In [11]:
```
# 3x3, random vs. boring

# TODO Q2. Play 2000 games between two bots, where Player 1 is Random and P
# Print the number of wins by Player 1, number of wins by Player 2, and dra
# Plot (as a function of game index) the moving average of game outcomes.

# Comment on the results. Compare with your plot above.
#Think about why this might be happening and explain your reasons.

random.seed(1)
games = 2000
results = [None]*2000

for i in range(games):
    results[i] = play(new_board(3), {+1: BoringPlayer(), -1: RandomPlayer()

P1_win = 0
P2_win = 0
draw = 0

for result in results:
    if result == 1.0:
        P1_win += 1
    elif result == -1.0:
        P2_win += 1
    else:
        draw += 1

print("Player 1 Wins ", P1_win, " games")
print("Player 2 Wins ", P2_win, " games")
print("Number of Draw Games: ", draw)
```

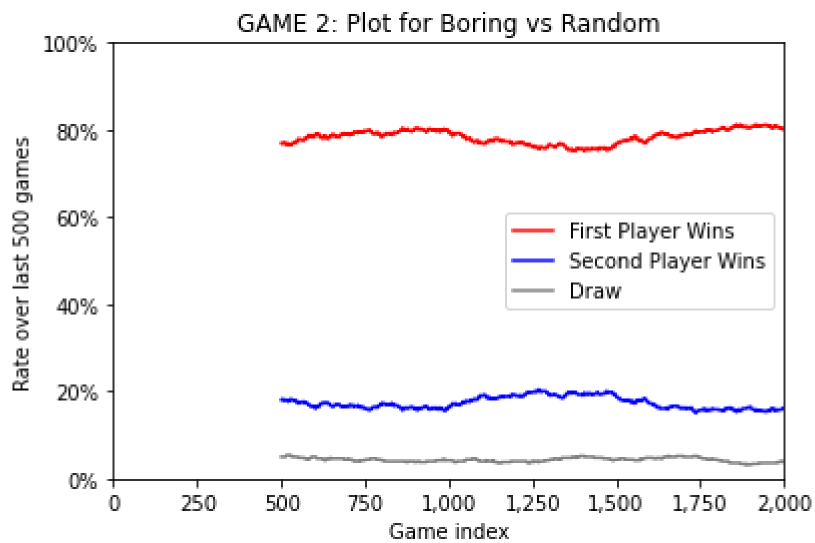
```
Player 1 Wins  1566  games
Player 2 Wins  347  games
Number of Draw Games:  87
```

In [12]:
```
show(results, title='GAME 2: Plot for Boring vs. Random')
```



Comparing the results between `GAME 1: Plot for Random vs. Random` and `GAME 2: Plot for Boring vs. Random`





Based on visual inspection, we can judge that when a player is 'boring' or when the player just picks the first available move rather than picking any at random, the player has a higher chance of winning.

The odds of Player 1 winning when the play style is shifted from Random to Boring have increased from roughly 0.6 to 0.8. Player 1's wins have most certainly impacted the winning chances of Player 2 or even ending the game with a draw (or no decision).

It is pretty clear that once a player shifts the play style from playing completely random moves to making the first available move, the odds of ending the game in a draw or even losing reduce significantly.

3.3 | Implement the "move" method

We will now use Q-learning using a neural network to train an RL agent.

The Q-function will be parametrically represented via a very simple single layer with linear activations (essentially, a linear model).

Complete the Q-learning part in the code snippet below.

In [13]:
```python
class Agent(Player):

    # Define single layer Q-network, MSE loss, and SGD optimizer
    def __init__(self, size, seed):
        self.size = size
        self.training = True
        self.model = tf.keras.Sequential()
        self.model.add(tf.keras.layers.Dense(
            size**2,
            kernel_initializer=tf.keras.initializers.glorot_uniform(seed=se
        self.model.compile(optimizer='sgd', loss='mean_squared_error')

    # Helper function to predict the Q-function
    def predict_q(self, board):
        return self.model.predict(
            np.array([board.ravel()])).reshape(self.size, self.size)

    # Helper function to train the network
    def fit_q(self, board, q_values):
        self.model.fit(
            np.array([board.ravel()]), np.array([q_values.ravel()]), verbos

    # The agent preserves history, which is reset when a new game starts.
    def new_game(self):
        self.last_move = None
        self.board_history = []
        self.q_history = []

    # TODO Q3: Implement the "move" method below.
    # The "move" method should use the output of the Q-network
    # that you defined above to pick the next best move.
    # Make sure you are only picking "legal" moves.

    def move(self, board):
        # ... COMPLETE THIS

        q_values = self.predict_q(board)
        temp = q_values.copy()
        temp[board != 0] = temp.min() - 1
        move = np.unravel_index(np.argmax(temp), board.shape)
        value = temp.max()

        if self.last_move is not None:
            self.reward(value)
        self.board_history.append(board.copy())
        self.q_history.append(q_values)
        return move

    # After picking the move, we call the reward method.
    # The reward method trains the Q-network, updating the Q-values with
    # a new estimate for the last move. This is the Bellman update.
    def reward(self, reward_value):
        if not self.training:
            return
        new_q = self.q_history[-1].copy()
        new_q[self.last_move] = reward_value
```

```
        self.fit_q(self.board_history[-1], new_q)
```

3.4 | q-Agent vs. Random

In [14]:
```
# 3x3, q-learning vs. random

# TODO Q4. Play 2000 games, where Player 1 is a Q-network and Player 2 is R
# Print the number of wins by Player 1, number of wins by Player 2, and dra
# Plot (as a function of game index) the moving average of game outcomes.

random.seed(1)
games = 2000
results=[None] * 2000

agent = Agent(3, seed=4)
for i in range(games):
    results[i] = play(new_board(3), {+1: agent, -1: RandomPlayer()})

P1_win = 0
P2_win = 0
draw = 0

for result in results:
    if result == 1.0:
        P1_win += 1
    elif result == -1.0:
        P2_win += 1
    else:
        draw += 1

print("Player 1 Wins ", P1_win, " games")
print("Player 2 Wins ", P2_win, " games")
print("Number of Draw Games: ", draw)
```

```
Player 1 Wins  1225  games
Player 2 Wins  627  games
Number of Draw Games:  148
```

In [15]:

```
show(results, title='GAME 3: Plot for Q-Agent vs. Random')
```