1.POLICY GRADIENTS

a) For Policy Gradient, our goal is to get the derivative of $V(s;\theta)$ and update θ with the derivative. The steps are:
1. Sample a trajectory/rollout $\tau = (s_0, a_0, .., s_L)$. Here, since we consider the so-called bandit setting where the trajectory does not matter, we will only consider one action, which can be regarded as $\tau = a_i$
2. Compute $R(\tau) = \sum_{t=0}^{L-1} -r(s_t, a_t)$. Since we doesn't consider the trajectory, and we know that different actions a_i give rise to different rewards R_i, we can get that $R(\tau) = R_i$
3. $\theta \leftarrow \theta - \eta R(\tau)\frac{\partial}{\partial\theta} log\pi(\tau)$, which means $\triangle\theta = \eta R(\tau)\frac{\partial}{\partial\theta} log\pi(\tau) = \eta R_i \frac{\partial}{\partial\theta} log\pi(a_i)$

As we known, we have

$$\pi(a_i) = softmax(\theta_i), i = 1, \dots, k$$

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

Thus, our goal now is to get the derivative of the log(Softmax)

We assume θ as a vector containing the action scores for a single state, that's the output of the network. Thus θ_i is an element for a certain action i in all actions k.
We can rewrite the softmax output as

$$\pi(a_i) = softmax(\theta_i) = p_i = \frac{e^{\theta_i}}{\sum_k e^{\theta_k}}$$

and the log-likelihood as

$$L_i = log(\pi(a_i)) = log(p_{y_i})$$

From chain rule we have

$$\frac{\partial L_i}{\partial\theta_i} = \frac{\partial L_i}{\partial p_i}\frac{\partial p_i}{\partial\theta_i}$$

For the first part,

$$\frac{\partial L_i}{\partial p_i} = \frac{1}{p_i}$$

For the second part,we have to recall the quotient rule for derivatives, let the derivative be represented by the operator $\mathbf{D}$,

$$\frac{f(x)}{g(x)} = \frac{g(x)\mathbf{D}f(x) - f(x)\mathbf{D}g(x)}{g(x)^2}$$

Here, set $\sum_k e^{\theta_k} = \Sigma$, we get,

$$\frac{\partial p_i}{\partial\theta_i} = \frac{\partial}{\partial\theta_i}\left(\frac{e^{\theta_i}}{\sum_k e^{\theta_k}}\right) = \frac{\Sigma\mathbf{D}e^{\theta_i} - e^{\theta_i}\mathbf{D}\Sigma}{\Sigma^2} = \frac{e^{\theta_i}(\Sigma - e^{\theta_i})}{\Sigma^2}$$

Here, $\mathbf{D}\sum = e^{\theta_i}$ because if we take the input array θ in the softmax function, we're always "looking" or we're always taking the derivative of the i-th element. In this case, the derivative with respect to the i^{th} element will always be 0 in those elements that are non-i, but $e^{\theta_i} ati$.

$$\frac{\partial p_i}{\partial f_i} = \frac{e^{\theta_i}(\Sigma - e^{\theta_i})}{\Sigma^2} = \frac{e^{\theta_i}}{\Sigma}\frac{\Sigma - e^{\theta_i}}{\Sigma} = p_i * (1 - p_i)$$

Thus, we know the derivative of the log(Softmax)

$$\frac{\partial L_i}{\partial\theta_i} = \frac{\partial L_i}{\partial p_i}\frac{\partial p_i}{\partial\theta_i} = \frac{1}{p_i}(p_i * (1 - p_i)) = (1 - p_i)$$

It means that

$$\frac{\partial}{\partial\theta} log\pi(a_i) = 1 - \pi(a_i)$$

Thus, we have

$$\triangle\theta = \eta R_i \frac{\partial}{\partial\theta} log\pi(a_i) = \eta R_i(1-\pi(a_i))$$

b) As we known from above, we can say that $\triangle\theta$ is proportional to the $E(reward - loss)$, which is the expected reward loss if action a_i doesn't happen. Here the $1-\pi(a_i)$ represents the predicted possibility of action a_i doesn't be chose, and R_i is the reward that we should received if we choose action a_i.
Also, reward R is proportional to the prediction loss. When R converges, the loss also converges, then once loss converges to a limited range, then the network will make little change on θ and then finish training.
For example, if choosing action a_i will actually result in big reward R_i, which means a_i is a good move, however the network predicts that the possibility that we should choose action a_i is small, which is $1-\pi(a_i)$ is large, then the $\triangle\theta$ would be large and parameter θ still need to be trained. And after many rounds, when agent reach the same state, the network now predicts that the possibility that we should choose action a_i is large, which is $1-\pi(a_i)$ is small, then the $\triangle\theta$ would be small and parameter θ will be slightly adjusted.

2.DESIGNING REWARDS IN Q-LEARNING

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

As we know, the formula of discounted return is

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}.$$

if a constant offset O is added to all rewards, then the discounted return becomes:

$$G_t = (R_{t+1} - O) + \gamma(R_{t+2} - O) + \gamma^2(R_{t+3} - O) + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}. - \sum_{k=0}^{\infty} \gamma^k O$$

As we can see from the second reward formula, if we add an offset to rewards, the sum of the offset to current reward would be $\sum_{k=0}^{\infty} \gamma^k O$. Once the number of total movement increases, which means the k increases, the total offset would increase, resulting in less reward. Therefore, we can know the importance of the negative rewards on every other move: it incentivizes the agent to get done as quickly as possible because it's constantly losing points when it play this game. If the agent want to maximize the rewards, it not only need to avoid monster and reach the goal, but also need to move as fewer times as possible.

Let's look into the expected discounted return $R_{(\tau)}$ of two strategy. Assume that r_1 is the reward for the agent reaching the goal, r_2 is the reward for the agent running into a monster, and r_3 is the reward for every other move. s_{t1} is the state that after agent choose action a_{t1} resulting in reaching the goal, s_{t2} is the state that after agent choose action a_{t2} resulting in running into a monster, s_{t3} is the state that after agent choose action a_{t3} resulting in an empty move.
The agent should try to minimize the negative reward:

$$minimizeR(\tau) = \sum -r(s_t, a_t) = \sum -r_1(s_{t1}, a_{t1}) + \sum -r_2(s_{t2}, a_{t2}) + \sum -r_3(s_{t3}, a_{t3})$$

For strategy a, $r_3(s_{t3}, a_{t3}) = 0$, which means that for the agent, the total number of empty move has no impact on its discounted return. Thus, agent might make more extra moves.
For strategy b, $r_3(s_{t3}, a_{t3}) = -0.5$, which means that for the agent, the total number of empty move has great impact on its discounted return and agent will try to make as less empty move as possible and reach to the goal faster, comparing to strategy a.

Thus, we know that strategy b is better than strategy a.

For strategy a), it is actually a sparse reward problem. Under this rewards strategy, agent only get positive reward when it reach the goal and negative reward when it run into a monster. And there is no feedback in other states(or ’empty’ move). Thus, there is no efficient feedback to lead the agent to move in the correct direction, so it is difficult for the agent to reach the goal only by random exploration. Therefore, in this case, the agent would take more time to explore and find a way to reach the goal. There might have the case that since there is no penalty for every extra ’empty’ move, the agent might spin in the same place and cannot figure out the path to the goal, when the agent is far away from the goal and the monster. Or in other words, the agent would be stuck in the local optimal.

For strategy b), there is negative reward for every other move, which will encourage the agent to reach the goal without running into monster as soon as possible so as to avoid accumulating penalties for every other move it makes. Thus, it will avoid the case that mentioned above in strategy a.

In conclusion, strategy b) is better as it sets negative reward for every other move.

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

In [ ]:
```
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). Let us set up some helper functions to define the game itself. The typical board size is 3x3 but we will be general.

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

In [ ]:
```
def available_moves(board):
  #  return a list of position that is aviable for next move. eg:
  #  [[1 0]
  #  [1 2]
  #  [2 0]]
    return np.argwhere(board == 0)

```

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

class RandomPlayer(Player):
    def move(self, board):
        # print(available_moves(board))
        # print(random.choice(available_moves(board)))  # the move is in [x y] format. eg. [1 2]
        return random.choice(available_moves(board))
```

A *boring* player always picks the *first* available position on the board.

In [ ]:
```python
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 [ ]:
```python
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
    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 # return the winner
```

In [ ]:

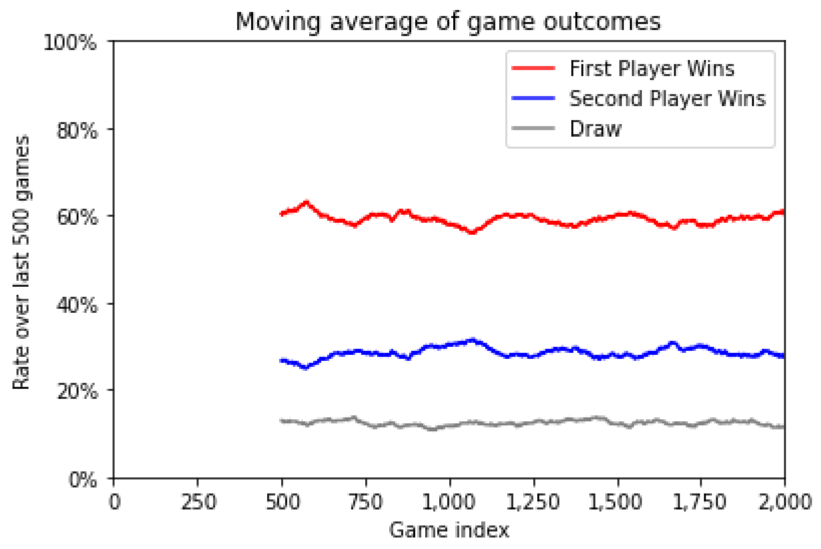
```
# 3x3, random vs. random
random.seed(1)

# TODO Q1. Play 2000 games between two bots, both of them random players.
player1 = RandomPlayer()
player2 = RandomPlayer()
player_objs = {}
player_objs[+1] = player1
player_objs[-1] = player2
# print(player_objs[+1])
player1_win_cnt = 0
player2_win_cnt = 0
draw_cnt = 0

game_res = []
for i in range(2000):
  winner = play(new_board(3),player_objs) # the board size is 3x3
  # print(winner)
  game_res.append(winner)
  if winner == 1.0:
    player1_win_cnt += 1
  elif winner == -1.0:
    player2_win_cnt += 1
  else:
    draw_cnt += 1
# Print the number of wins by Player 1, number of wins by Player 2, and draws.
print('Player1 won ' + str(player1_win_cnt) + ' times')
print('Player2 won ' + str(player2_win_cnt) + ' times')
print('Draw ' + str(draw_cnt) + ' times')

# 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.
show(game_res)
```

```
Player1 won 1194 times
Player2 won 564 times
Draw 242 times
```



In [ ]:
```python
def moving(data, size, value=+1):
    binary_data = [x == value for x in data]
    return [sum(binary_data[i-size:i])/size for i in range(size, len(data) + 1)]

def show(results, size=500, title='Moving average of game outcomes',
         first_label='First Player Wins', second_label='Second Player Wins', draw_label='Draw'):
    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:,.0f}'))
    plt.ylabel(f'Rate over last {size} games')
    plt.xlabel('Game index')
    plt.show()
```

In [ ]:

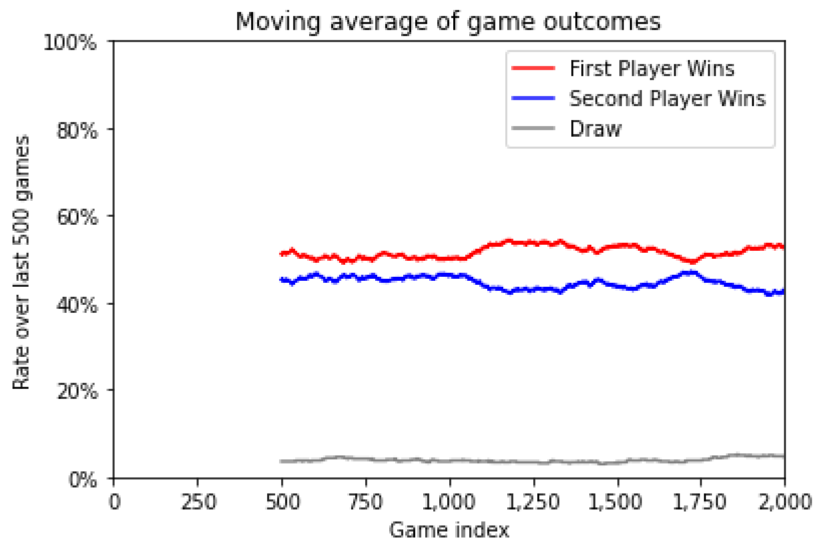
```python
# 3x3, random vs. boring

# TODO Q2. Play 2000 games between two bots, where Player 1 is Random and Player 2 is Boring.
random.seed(2)

player1 = RandomPlayer()
player2 = BoringPlayer()
player_objs = {}
player_objs[+1] = player1
player_objs[-1] = player2
# print(player_objs[+1])
player1_win_cnt = 0
player2_win_cnt = 0
draw_cnt = 0

game_res = []
for i in range(2000):
  winner = play(new_board(3),player_objs) # the board size is 3x3
  # print(winner)
  game_res.append(winner)
  if winner == 1.0:
    player1_win_cnt += 1
  elif winner == -1.0:
    player2_win_cnt += 1
  else:
    draw_cnt += 1
# Print the number of wins by Player 1, number of wins by Player 2, and draws.
print('Player1 won ' + str(player1_win_cnt) + ' times')
print('Player2 won ' + str(player2_win_cnt) + ' times')
print('Draw ' + str(draw_cnt) + ' times')
# Plot (as a function of game index) the moving average of game outcomes.
show(game_res)
# Comment on the results, and speculate on why this might be happening.
```

```
Player1 won 1033 times
Player2 won 891 times
Draw 76 times
```



COMMENT: As we can see from the above results, we knows that:

1. In both cases, the first player has higher rate than the second player. The player that goes first will win about twice as often as the player that goes second. To some extent, this makes sense. There are only nine squares on a Tic-Tac-Toe board, the first player will get five of them but the second player will only get four.
2. When change the second player from RandomPlayer to BoringPlayer, the win rate increases, and the draw time decrease, which means that the strategy of always picking the *first* available position on the board is better than picking a *random* position in the board each move, when the opponent is a RandomPlayer. The reason of this might be that the strategy of RandomPlayer, chosing the move from left to right, from top to bottom, is more likely to block opponent's winning path (three of their marks in a horizontal, vertical, or diagonal row is the winner) and more likely to fill the first row and win the game, if the RandomPlayer doesn't fill the place in the first row in the first three steps.

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 [ ]:
```
class Agent(Player):
    # Define single layer model, MSE loss, and SGD optimizer
    def __init__(self, size, seed):
        # self.decay_factor = decay_factor # I added a decay_factor for calculating the Q-values
        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=seed)))
        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()]), verbose=0)

    # 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
    # to pick the next best move.
    # Make sure you are only picking "legal" moves.

    # After picking the move, we call the reward method.

    def move(self, board):
        # ... COMPLETE THIS
        q_values = self.predict_q(board)
        # print('prediction of q table :')
        # print(q_values)

        available_moves_thistime = available_moves(board)
        move_options = []
        move = available_moves_thistime[0]
        move_options.append(move)
        # print('the available moves: ')
        # print(available_moves_thistime)

        # find the best move in available moves
        for x, y in available_moves_thistime:
            if q_values[x][y] > q_values[move[0]][move[1]]:
              move = [x, y]
              move_options = []
              move_options.append(move)
            elif ( q_values[x][y] == q_values[move[0]][move[1]] and move[0] != x and move[1] != y )
              move_options.append([x, y])
            else:
              continue


        # when there are multiple best available moves, randomly pick one
        if len(move_options) > 1:
          move = random.choice(move_options)
        # print('all the move options: ')
        # print(move_options)
        # print('the selected move: ')
        # print(move)

        # update the reward with Bellman update

        # last_q_value = 0
        # # use the q-value of our move in last round as the reward we already got
        # if self.last_move is not None:
        #   last_q_value = self.q_history[-1][self.last_move]
        # value = last_q_value + self.decay_factor * q_values[x][y]

        value = q_values[move[0]][move[1]]

        if self.last_move is not None:
            self.reward(value)

        self.board_history.append(board.copy())
        # here we just append the predited q value table without the updated Q_value
        # the reason why is that we can find the q-value of our move in last round
        # and use it as the reward we already got, which brings convience for doing Bellman equatio
        self.q_history.append(q_values)
        # update the last move as the move we just chose
        self.last_move = move
        return move

    # 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)
```

In [ ]:

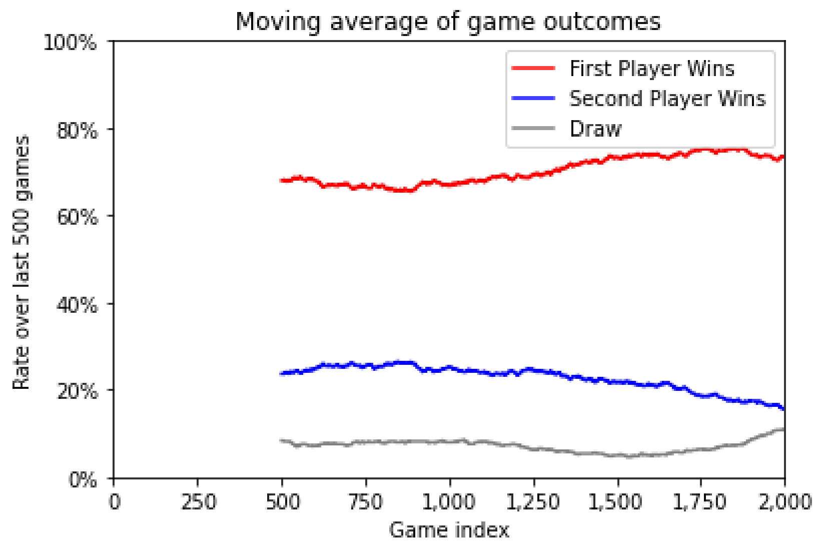
```python
# 3x3, q-learning vs. random

# TODO Q4. Play 2000 games, where Player 1 is a Q-network and Player 2 is Random.
random.seed(3)
# player1 = Agent(size=3,seed=random.seed(1), decay_factor = 0.01)
player1 = Agent(size=3,seed=3)
player2 = RandomPlayer()
player_objs = {}
player_objs[+1] = player1
player_objs[-1] = player2
# print(player_objs[+1])
player1_win_cnt = 0
player2_win_cnt = 0
draw_cnt = 0

game_res = []
for i in range(2000):
  winner = play(new_board(3),player_objs) # the board size is 3x3
  # print(winner)
  game_res.append(winner)
  if winner == 1.0:
    player1_win_cnt += 1
  elif winner == -1.0:
    player2_win_cnt += 1
  else:
    draw_cnt += 1

# Print the number of wins by Player 1, number of wins by Player 2, and draws.
print('Player1 won ' + str(player1_win_cnt) + ' times')
print('Player2 won ' + str(player2_win_cnt) + ' times')
print('Draw ' + str(draw_cnt) + ' times')
# Plot (as a function of game index) the moving average of game outcomes.
show(game_res)
```

```
Player1 won 1409 times
Player2 won 429 times
Draw 162 times
```



COMMENT: As we can see from the above result, we knows that as the Agent played more and more times, the win rate of Agent is getting higher and higher, which means that it learned the strategy of how to win the RandomPlayer and it know how to play with experience.

```
In [ ]:
```