

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

```
In [ ]: 1 import random
        2 import collections
        3 import numpy as np
        4 import tensorflow as tf
        5 import matplotlib.pyplot as plt
        6 import matplotlib.ticker
        7 %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 [ ]: 1 def new_board(size):
        2     return np.zeros(shape=(size, size))
```

```
In [ ]: 1 def available_moves(board):
        2     # return a list of position that is aviable for next move. eg:
        3     # [[1 0]
        4     # [1 2]
        5     # [2 0]]
        6     return np.argwhere(board == 0)
        7
```

```
In [ ]: 1 def check_game_end(board):
        2     best = max(list(board.sum(axis=0)) + # columns
        3               list(board.sum(axis=1)) + # rows
        4               [board.trace()] + # main diagonal
        5               [np.fliplr(board).trace()], # other diagonal
        6               key=abs)
        7     if abs(best) == board.shape[0]:
        8         return np.sign(best) # winning player, +1 or -1
        9     if available_moves(board).size == 0:
        10        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 [ ]: 1 class Player():
2         def new_game(self):
3             pass
4         def reward(self, value):
5             pass
6
7     class RandomPlayer(Player):
8         def move(self, board):
9             # print(available_moves(board))
10            # print(random.choice(available_moves(board))) # the move is in [x y] format. eg. [1 2]
11            return random.choice(available_moves(board))
```

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

```
In [ ]: 1 class BoringPlayer(Player):
2         def move(self, board):
3             return available_moves(board)[0]
```

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

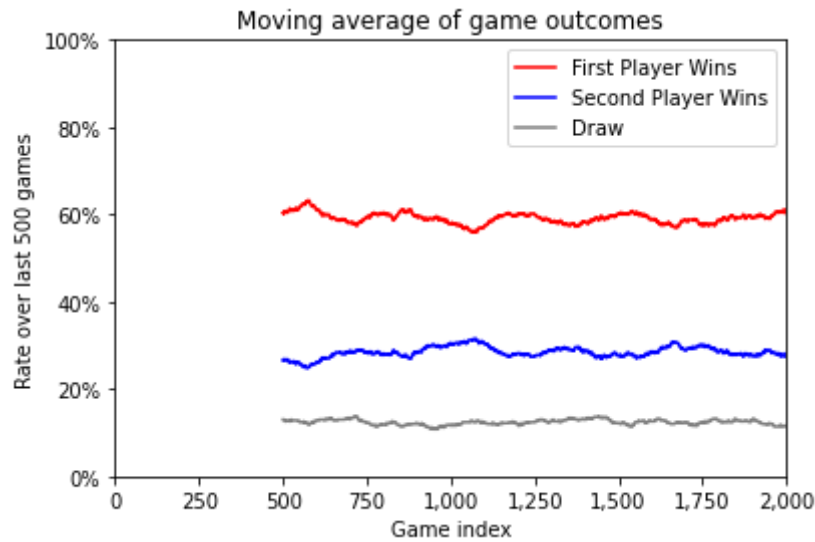
```
In [ ]: 1 def play(board, player_objs):
2         for player in [+1, -1]:
3             player_objs[player].new_game()
4         player = +1
5         game_end = check_game_end(board)
6         while game_end is None:
7             move = player_objs[player].move(board)
8             board[tuple(move)] = player
9             game_end = check_game_end(board)
10            player *= -1 # switch players
11        for player in [+1, -1]:
12            # the reward for wins is +1, and -1 for draws/losses
13            reward_value = +1 if player == game_end else -1
14            player_objs[player].reward(reward_value)
15        return game_end # return the winner
```

```
In [ ]: 1 # 3x3, random vs. random
2 random.seed(1)
3
4 # TODO Q1. Play 2000 games between two bots, both of them random players.
5 player1 = RandomPlayer()
6 player2 = RandomPlayer()
7 player_objs = {}
8 player_objs[+1] = player1
9 player_objs[-1] = player2
10 # print(player_objs[+1])
11 player1_win_cnt = 0
12 player2_win_cnt = 0
13 draw_cnt = 0
14
15 game_res = []
16 for i in range(2000):
17     winner = play(new_board(3),player_objs) # the board size is 3x3
18     # print(winner)
19     game_res.append(winner)
20     if winner == 1.0:
21         player1_win_cnt += 1
22     elif winner == -1.0:
23         player2_win_cnt += 1
24     else:
25         draw_cnt += 1
26 # Print the number of wins by Player 1, number of wins by Player 2, and draws.
27 print('Player1 won ' + str(player1_win_cnt) + ' times')
28 print('Player2 won ' + str(player2_win_cnt) + ' times')
29 print('Draw ' + str(draw_cnt) + ' times')
30
31 # Plot (as a function of game index) the moving average of game outcomes
32 # over a window of size 500.
33 # You might find the following functions helpful for plotting.
34 show(game_res)
```

Player1 won 1194 times

Player2 won 564 times

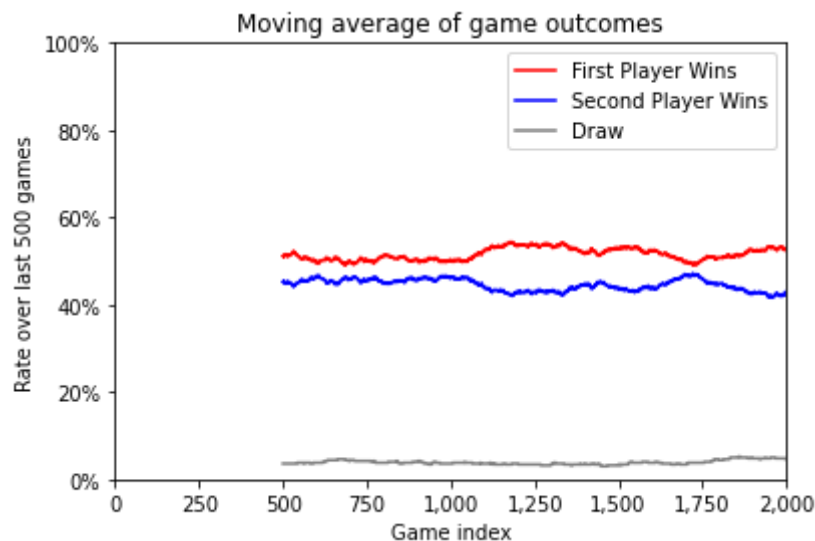
Draw 242 times



```
In [ ]: 1 def moving(data, size, value=+1):
2     binary_data = [x == value for x in data]
3     return [sum(binary_data[i-size:i])/size for i in range(size, len(data) + 1)]
4
5 def show(results, size=500, title='Moving average of game outcomes',
6         first_label='First Player Wins', second_label='Second Player Wins', draw_label='Draw'):
7     x_values = range(size, len(results) + 1)
8     first = moving(results, value=+1, size=size)
9     second = moving(results, value=-1, size=size)
10    draw = moving(results, value=0, size=size)
11    first, = plt.plot(x_values, first, color='red', label=first_label)
12    second, = plt.plot(x_values, second, color='blue', label=second_label)
13    draw, = plt.plot(x_values, draw, color='grey', label=draw_label)
14    plt.xlim([0, len(results)])
15    plt.ylim([0, 1])
16    plt.title(title)
17    plt.legend(handles=[first, second, draw], loc='best')
18    ax = plt.gca()
19    ax.yaxis.set_major_formatter(matplotlib.ticker.PercentFormatter(xmax=1))
20    ax.xaxis.set_major_formatter(matplotlib.ticker.StrMethodFormatter('{x:,.0f}'))
21    plt.ylabel(f'Rate over last {size} games')
22    plt.xlabel('Game index')
23    plt.show()
```

```
In [ ]: 1 # 3x3, random vs. boring
2
3 # TODO Q2. Play 2000 games between two bots, where Player 1 is Random and Player 2 is Boring.
4 random.seed(2)
5
6 player1 = RandomPlayer()
7 player2 = BoringPlayer()
8 player_objs = {}
9 player_objs[+1] = player1
10 player_objs[-1] = player2
11 # print(player_objs[+1])
12 player1_win_cnt = 0
13 player2_win_cnt = 0
14 draw_cnt = 0
15
16 game_res = []
17 for i in range(2000):
18     winner = play(new_board(3), player_objs) # the board size is 3x3
19     # print(winner)
20     game_res.append(winner)
21     if winner == 1.0:
22         player1_win_cnt += 1
23     elif winner == -1.0:
24         player2_win_cnt += 1
25     else:
26         draw_cnt += 1
27 # Print the number of wins by Player 1, number of wins by Player 2, and draws.
28 print('Player1 won ' + str(player1_win_cnt) + ' times')
29 print('Player2 won ' + str(player2_win_cnt) + ' times')
30 print('Draw ' + str(draw_cnt) + ' times')
31 # Plot (as a function of game index) the moving average of game outcomes.
32 show(game_res)
33 # 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 know that:

1. In both cases, the first player has a 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 changing the second player from RandomPlayer to BoringPlayer, the win rate increases, and the draw time decreases, 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 for this might be that the strategy of RandomPlayer, choosing 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 [ ]: 1 class Agent(Player):
2         # Define single layer model, MSE loss, and SGD optimizer
3         def __init__(self, size, seed):
4             # self.decay_factor = decay_factor # I added a decay_factor for calculating the Q-values
5             self.size = size
6             self.training = True
7             self.model = tf.keras.Sequential()
8             self.model.add(tf.keras.layers.Dense(
9                 size**2,
10                 kernel_initializer=tf.keras.initializers.glorot_uniform(seed=seed)))
11             self.model.compile(optimizer='sgd', loss='mean_squared_error')
12
13         # Helper function to predict the Q-function
14         def predict_q(self, board):
15             return self.model.predict(
16                 np.array([board.ravel()])).reshape(self.size, self.size)
17
18         # Helper function to train the network
19         def fit_q(self, board, q_values):
20             self.model.fit(
21                 np.array([board.ravel()]), np.array([q_values.ravel()]), verbose=0)
22
23         # The agent preserves history, which is reset when a new game starts.
24         def new_game(self):
25             self.last_move = None
26             self.board_history = []
27             self.q_history = []
28
29         # TODO Q3: Implement the "move" method below.
30         # The "move" method should use the output of the Q-network
31         # to pick the next best move.
32         # Make sure you are only picking "legal" moves.
33
34         # After picking the move, we call the reward method.
35
36         def move(self, board):
37             # ... COMPLETE THIS
38             q_values = self.predict_q(board)
39             # print('prediction of q table :')
40             # print(q_values)
41
42             available_moves_thistime = available_moves(board)
```

```

43
44 move_options = []
45 move = available_moves_thistime[0]
46 move_options.append(move)
47 # print('the available moves: ')
48 # print(available_moves_thistime)
49
50 # find the best move in available moves
51 for x, y in available_moves_thistime:
52     if q_values[x][y] > q_values[move[0]][move[1]]:
53         move = [x, y]
54         move_options = []
55         move_options.append(move)
56     elif ( q_values[x][y] == q_values[move[0]][move[1]] and move[0] != x and move[1] != y )
57         move_options.append([x, y])
58     else:
59         continue
60
61
62 # when there are multiple best available moves, randomly pick one
63 if len(move_options) > 1:
64     move = random.choice(move_options)
65 # print('all the move options: ')
66 # print(move_options)
67 # print('the selected move: ')
68 # print(move)
69
70 # update the reward with Bellman update
71
72 # last_q_value = 0
73 # # use the q-value of our move in last round as the reward we already got
74 # if self.last_move is not None:
75 #     last_q_value = self.q_history[-1][self.last_move]
76 # value = last_q_value + self.decay_factor * q_values[x][y]
77
78 value = q_values[move[0]][move[1]]
79
80 if self.last_move is not None:
81     self.reward(value)
82
83 self.board_history.append(board.copy())
84 # here we just append the predicted q value table without the updated Q_value
85 # the reason why is that we can find the q-value of our move in last round

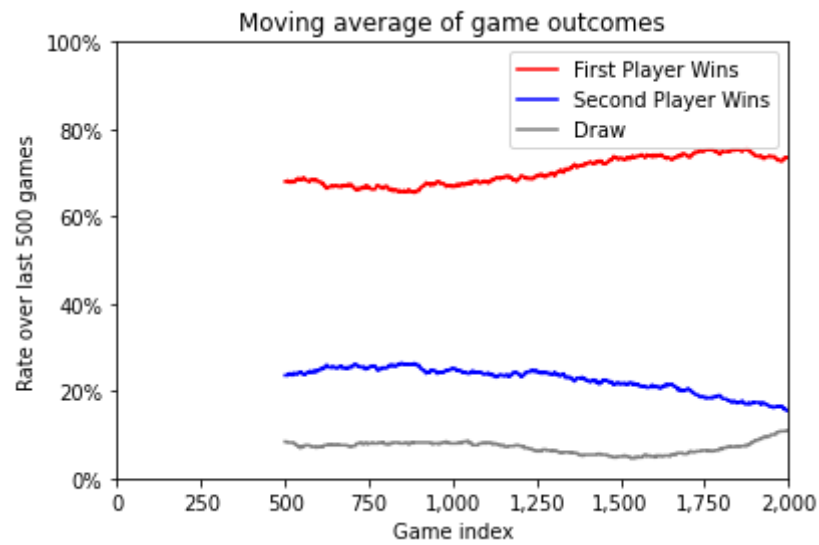
```



```
86         # and use it as the reward we already got, which brings convience for doing Bellman equatio.
87         self.q_history.append(q_values)
88         # update the last move as the move we just chose
89         self.last_move = move
90         return move
91
92     # The reward method trains the Q-network, updating the Q-values with
93     # a new estimate for the last move. This is the Bellman update.
94     def reward(self, reward_value):
95         if not self.training:
96             return
97         new_q = self.q_history[-1].copy()
98         new_q[self.last_move] = reward_value
99         self.fit_q(self.board_history[-1], new_q)
```

```
In [ ]: 1 # 3x3, q-learning vs. random
2
3 # TODO Q4. Play 2000 games, where Player 1 is a Q-network and Player 2 is Random.
4 random.seed(3)
5 # player1 = Agent(size=3,seed=random.seed(1), decay_factor = 0.01)
6 player1 = Agent(size=3,seed=3)
7 player2 = RandomPlayer()
8 player_objs = {}
9 player_objs[+1] = player1
10 player_objs[-1] = player2
11 # print(player_objs[+1])
12 player1_win_cnt = 0
13 player2_win_cnt = 0
14 draw_cnt = 0
15
16 game_res = []
17 for i in range(2000):
18     winner = play(new_board(3),player_objs) # the board size is 3x3
19     # print(winner)
20     game_res.append(winner)
21     if winner == 1.0:
22         player1_win_cnt += 1
23     elif winner == -1.0:
24         player2_win_cnt += 1
25     else:
26         draw_cnt += 1
27
28 # Print the number of wins by Player 1, number of wins by Player 2, and draws.
29 print('Player1 won ' + str(player1_win_cnt) + ' times')
30 print('Player2 won ' + str(player2_win_cnt) + ' times')
31 print('Draw ' + str(draw_cnt) + ' times')
32 # Plot (as a function of game index) the moving average of game outcomes.
33 show(game_res)
```

Player1 won 1409 times
Player2 won 429 times
Draw 162 times



COMMENT: As we can see from the above result, we know 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 knows how to play with experience.

In []:

1