project

June 4, 2022

[11]:
```
import multiprocessing
import random
import numpy as np
import matplotlib.pyplot as plt

from tic_env import TictactoeEnv, OptimalPlayer
```

[12]:
```
# In case of using parallel calculations

def worker(func, arg, return_dict):
    """worker function"""
    return_dict[arg] = func(arg)

def to_parallel(func, func_args):
    manager = multiprocessing.Manager()
    return_dict = manager.dict()
    jobs = []
    for arg in func_args:
        p = multiprocessing.Process(target=worker, args=(func, arg, return_dict))
        jobs.append(p)
        p.start()

    for proc in jobs:
        proc.join()
    return dict(return_dict)
```

TODO

- Move to real Q-table
- Generate sparse 5477x9 table and state2index dict
- Use argmax in this table
- Fix `update_qtable` and `act`
- Add `play_game` function instead of `test`
- Make *more correct* states trace list
- Plot trend line of our model
- Modify "choice of ϵ"
- Perform grid search through all decay gammas, lrs and expl rates
- Check if the "averaging the rewards" is calculated correctly

- More precise search for n*
- Modify the explaination of n* effect
- Why exploit is the same for all decreasing eps?
- Add answer to 3rd question
- Create Mrand and Mopt funcions
- Create short functions for training + params
- Fix a bit DQL Lfsp paragraph

1 Q-learning

As our 1st algorithm, we use Q-Learning combined with ϵ-greedy policy - see section 6.5 of Sutton and Barto (2018) for details. At each time t, state s_t is the board position (showing empty positions, positions taken by you, and positions taken by your opponent; c.f. tic_tac_toe.ipynb), action a_t is one of the available positions on the board (i.e. ϵ-greedy is applied only over the available actions), and reward r_t is only non-zero when the game ends where you get $r_t = 1$ if you win the game, $r_t = -1$ if you lose, and $r_t = 0$ if it is a draw.

Q-Learning has 3 hyper-parameters: learning rate α, discount factor γ, and exploration level ϵ. For convenience, we fix the learning rate at $\alpha = 0.05$ and the discount factor at $\gamma = 0.99$. We initialize all the Q-values at 0 ; if you are curious, you can explore the effect of α, γ, and initial Q-values for yourself.

[13]:
```
# Converts state [-1 1 0 0 1 -1 0 -1 0] to 021120101 and then to int from base 3
st2int = lambda st: int(((st.flatten() + 1) * (3**np.arange(9))).sum())

# https://xkcd.com/832/
class Player:
    def __init__(self, learning_rate, decay_gamma, exp_rate):
        self.states = []
        self.states_value = {}  # state -> value
        self.lr = learning_rate
        self.decay_gamma = decay_gamma
        self.exp_rate = exp_rate
        self.exp = 0
        self.env = TictactoeEnv()

    def act(self, grid, symbol, eps):
        if np.random.uniform(0, 1) <= eps:
            action = self.random_move(grid)
        else:
            positions = self.available_positions(grid)
            value_max = -999
            num_symb = 1 if symbol == 'X' else -1
            for p in positions:
                next_board = grid.copy()
                next_board[p] = num_symb
                state_key = st2int(next_board)
```

```
                value = 0 if self.states_value.get(state_key) is None else self.
↪states_value[state_key]
                if value >= value_max:
                    value_max = value
                    action = p
        return action

    def random_move(self, grid):
        """ Chose a random move from the available options. """
        return random.choice(self.available_positions(grid))

    def available_positions(self, grid): #nommé empty dans env
        '''return all empty positions'''
        return [(i // 3, i % 3) for i in range(9) if not grid[(i // 3, i % 3)]]

    def update_qtable(self, states, reward):
        for state in reversed(self.states):
            if self.states_value.get(state) is None:
                self.states_value[state] = 0
            self.states_value[state] += self.lr*(self.decay_gamma*reward - self.
↪states_value[state])
            reward = self.states_value[state]

    def add_state(self, state):
        self.states.append(st2int(state))

    def train(self, N, opponent_eps, exp_rate=None):
        exp_rate = exp_rate if exp_rate != None else self.exp_rate
        Turns = np.array(['X','O'])
        random.shuffle(Turns)
        total_reward = 0
        for i in range(1, N+1):
            self.states = []
            self.exp += 1
            self.env.reset()
            grid, _, __ = self.env.observe()
            Turns = Turns[::-1]
            player_opt = OptimalPlayer(opponent_eps, player=Turns[0])

            end = 0  # to run first iteration
            while not end:
                if self.env.current_player == player_opt.player:
                    move = player_opt.act(grid)
                    grid, end, winner = self.env.step(move, print_grid=False)
                else:
                    move = self.act(grid, Turns[1], exp_rate)
                    grid, end, winner = self.env.step(move, print_grid=False)
```

```
                    self.add_state(self.env.grid.flatten())

            reward = self.env.reward(Turns[1])
            total_reward += reward
            self.update_qtable(self.states, reward)
        return total_reward / N

    def test(self, N, opponent_eps, exp_rate=None):
        exp_rate = exp_rate if exp_rate != None else self.exp_rate
        Turns = np.array(['X','O'])
        random.shuffle(Turns)
        total_reward = 0
        for i in range(1, N+1):
            self.env.reset()
            grid, _, __ = self.env.observe()
            Turns = Turns[::-1]
            player_opt = OptimalPlayer(opponent_eps, player=Turns[0])

            end = 0  # to run first iteration
            while not end:
                if self.env.current_player == player_opt.player:
                    move = player_opt.act(grid)
                    grid, end, winner = self.env.step(move, print_grid=False)
                else:
                    move = self.act(grid, Turns[1], exp_rate)
                    grid, end, winner = self.env.step(move, print_grid=False)

            reward = self.env.reward(Turns[1])
            total_reward += reward

        return total_reward / N
```

1.1 Learning from experts

In this section, you will study whether Q-learning can learn to play Tic Tac Toe by playing against Opt (ϵ_{opt}) for some $\epsilon_{\text{opt}} \in [0, 1]$. To do so, implement the Q-learning algorithm. To check the algorithm, run a Q-learning agent, with a fixed and arbitrary $\epsilon \in [0, 1)$, against 0pt (0.5) for 20 '000 games - switch the 1st player after every game.

Question 1. Plot average reward for every 250 games during training – i.e. after the 50th game, plot the average reward of the first 250 games, after the 100th game, plot the average reward of games 51 to 100, etc. Does the agent learn to play Tic Tac Toe?
Expected answer: A figure of average reward over time (caption length < 50 words). Specify your choice of ϵ.

[14]:
```
learning_rate = 0.05
discount_factor = 0.99
my_exp_rate = 0.5
```

```
opponent_exp_rate = 0.5

my_player = Player(learning_rate, discount_factor, my_exp_rate)
```

[15]:
```
# avgs = []
# opponent_exp_rate = 1
# N = 20000
# avg_every = 250
# batch_sz = 50
# for i in range(1, N // avg_every + 1):
#     batch = [my_player.train(batch_sz, opponent_exp_rate) for _ in␣
 ↪range(avg_every // batch_sz)]
#     avgs.append(sum(batch) / len(batch))
#     print(f"Ep: {avg_every*i}, avg: {round(avgs[-1], 3)}, {batch}")
```

[16]:
```
opponent_exp_rate = 0.5
N = 20000
train_avg_every = batch_sz = 250
test_avg_every = 1000
minibatch = 50

train = []
test = []
for i in range(0, N, train_avg_every):
    batch = [
        my_player.train(minibatch, opponent_exp_rate)
        for j in range(0, batch_sz, minibatch)
    ]
    train.append(sum(batch) / len(batch))

    if i and not i % test_avg_every:
        batch = [
            my_player.test(minibatch, opponent_exp_rate, 0)
            for j in range(0, batch_sz, minibatch)
        ]
        test.append(sum(batch) / len(batch))
        print(f"Ep: {i}, avg: {round(test[-1], 3)}")

batch = [
    my_player.test(batch_sz, opponent_exp_rate, 0)
    for j in range(0, batch_sz, minibatch)
]
test.append(sum(batch) / len(batch))
print(f"Ep: {i}, avg: {round(test[-1], 3)}")
```

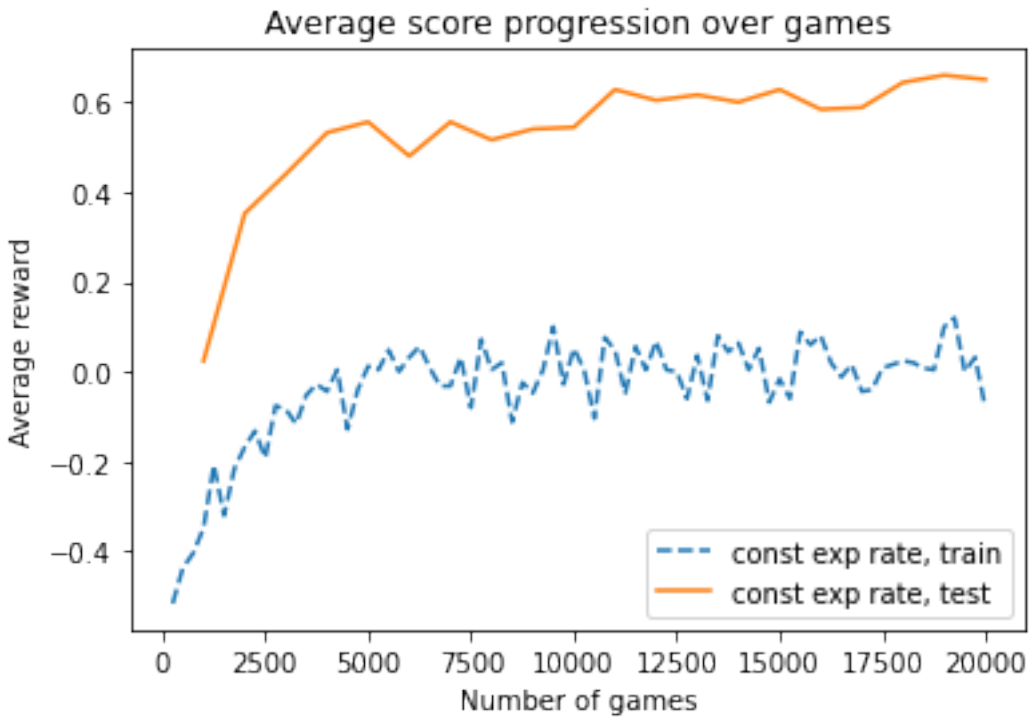
```
Ep: 1000, avg: 0.024
Ep: 2000, avg: 0.352
```

```
Ep: 3000, avg: 0.44
Ep: 4000, avg: 0.532
Ep: 5000, avg: 0.556
Ep: 6000, avg: 0.48
Ep: 7000, avg: 0.556
Ep: 8000, avg: 0.516
Ep: 9000, avg: 0.54
Ep: 10000, avg: 0.544
Ep: 11000, avg: 0.628
Ep: 12000, avg: 0.604
Ep: 13000, avg: 0.616
Ep: 14000, avg: 0.6
Ep: 15000, avg: 0.628
Ep: 16000, avg: 0.584
Ep: 17000, avg: 0.588
Ep: 18000, avg: 0.644
Ep: 19000, avg: 0.66
Ep: 19750, avg: 0.65
```

[17]:
```
plt.plot((np.arange(len(train))+1)*train_avg_every, train, '--', label='const␣
 ↪exp rate, train')
plt.plot((np.arange(len(test))+1)*test_avg_every, test, '-', label='const exp␣
 ↪rate, test')

plt.rcParams["figure.figsize"] = (16,10)

plt.legend(loc="lower right")
plt.title('Average score progression over games')
plt.xlabel("Number of games")
plt.ylabel("Average reward")
plt.show()
```



The used exploration rate is $\epsilon = 0.5$ because it should half of games explore and half of games play best strategy

1.1.1 Decreasing exploration

One way to make training more efficient is to decrease the exploration level ϵ over time. If we define $\epsilon(n)$ to be ϵ for game number n, then one feasible way to decrease exploration during training is to use

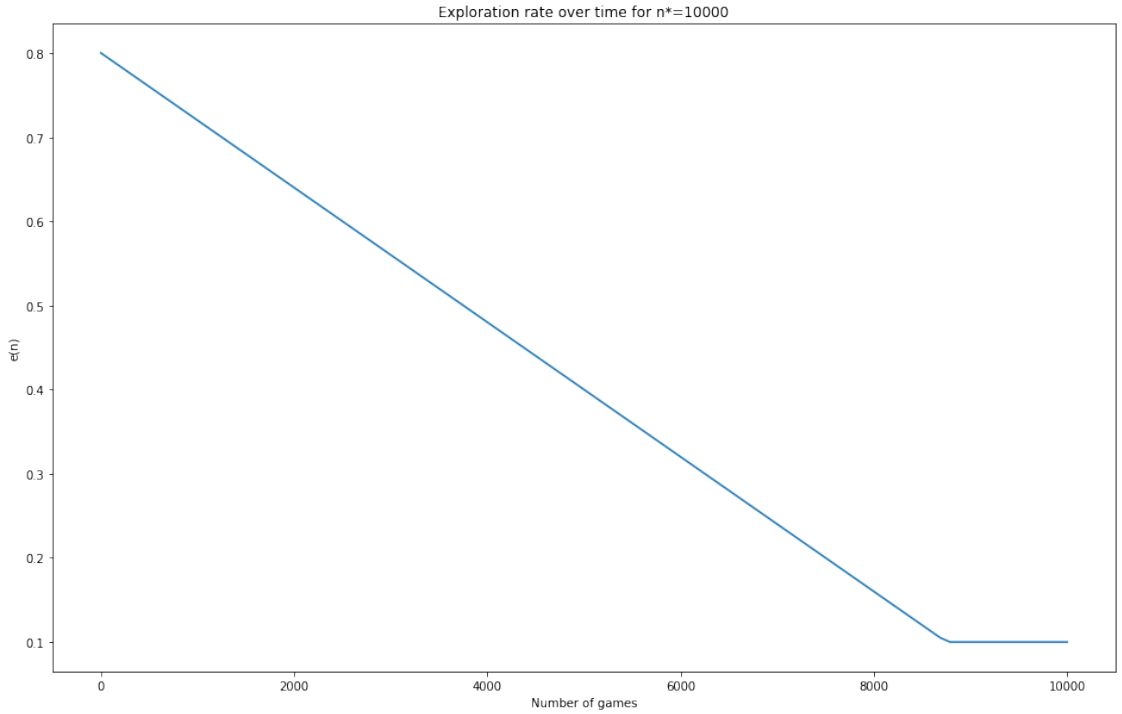
$$\epsilon(n) = \max\left\{\epsilon_{\min}, \epsilon_{\max}\left(1 - \frac{n}{n^*}\right)\right\}$$

where $\epsilon_{\min}$ and $\epsilon_{\max}$ are the minimum and maximum values for ϵ, respectively, and n^* is the number of exploratory games and shows how fast ϵ decreases. For convenience, we assume $\epsilon_{\min} = 0.1$ and $\epsilon_{\max} = 0.8$; if you are curious, you can explore their effect on performance for yourself. Use $\epsilon(n)$ as define above and run different Q-learning agents with different values of n^* against Opt (0.5) for $20'000$ games - switch the 1 st player after every game. Choose several values of n^* from a reasonably wide interval between 1 to $40'000-$ particularly, include $n^* = 1$.

[18]:
```
eps_min, eps_max = 0.1, 0.8
e = lambda n, n_star: max(eps_min, eps_max*(1 - n/n_star))

# Eps visualization
x = np.linspace(0, 10000, 100)
plt.plot(x, np.array([e(xi, 10000) for xi in x]))
```

```
plt.title('Exploration rate over time for n*=10000')
plt.xlabel("Number of games")
plt.ylabel("e(n)")
plt.show()
```



Question 2. Plot average reward for every 250 games during training. Does decreasing ϵ help training compared to having a fixed ϵ? What is the effect of n^*?
Expected answer: A figure showing average reward over time for different values of n^* (caption length $<$ 200 words).

Question 3. After every 250 games during training, compute the 'test' M_{opt} and M_{rand} for your agents - when measuring the 'test' performance, put $\epsilon = 0$ and do not update the Q-values. Plot M_{opt} and M_{rand} over time. Describe the differences and the similarities between these curves and the ones of the previous question.
Expected answer: A figure showing M_{opt} and M_{rand} over time for different values of n^* (caption length $<$ 100 words).

[19]:
```
n_stars = np.linspace(1, 40000, 5).astype(int); n_stars
```

[19]:
```
array([    1, 10000, 20000, 30000, 40000])
```

[21]:
```
learning_rate = 0.05
discount_factor = 0.99
my_exp_rate = 0.5
```

```
opponent_exp_rate = 0.5
N = 20000
train_avg_every = batch_sz = 250
test_avg_every = 1000
minibatch = 50


def get_avgs(n_star):
    player = Player(learning_rate, discount_factor, my_exp_rate)

    avgs = {'train': [], 'test': []}
    for i in range(0, N, train_avg_every):
        batch = [
            player.train(minibatch, opponent_exp_rate, e(i+j, n_star))
            for j in range(0, batch_sz, minibatch)
        ]
        avgs['train'].append(sum(batch) / len(batch))

        if i and not i % test_avg_every:
            batch = [
                player.test(minibatch, opponent_exp_rate, 0)
                for j in range(0, batch_sz, minibatch)
            ]
            avgs['test'].append(sum(batch) / len(batch))
            print(f"Ep: {i}, n: {n_star}, avg: {round(avgs['test'][-1], 3)}")

    del player
    return avgs

# Processing parallelizing
# results = to_parallel(get_avgs, n_stars)
results = {}
for n_star in n_stars:
    results[n_star] = get_avgs(n_star)
```

```
Ep: 1000, n: 1, avg: 0.224
Ep: 2000, n: 1, avg: 0.384
Ep: 3000, n: 1, avg: 0.364
Ep: 4000, n: 1, avg: 0.496
Ep: 5000, n: 1, avg: 0.484
Ep: 6000, n: 1, avg: 0.496
Ep: 7000, n: 1, avg: 0.444
Ep: 8000, n: 1, avg: 0.512
Ep: 9000, n: 1, avg: 0.52
Ep: 10000, n: 1, avg: 0.588
Ep: 11000, n: 1, avg: 0.488
Ep: 12000, n: 1, avg: 0.612
```

```
Ep: 13000, n: 1, avg: 0.532
Ep: 14000, n: 1, avg: 0.508
Ep: 15000, n: 1, avg: 0.524
Ep: 16000, n: 1, avg: 0.56
Ep: 17000, n: 1, avg: 0.588
Ep: 18000, n: 1, avg: 0.568
Ep: 19000, n: 1, avg: 0.6
Ep: 1000, n: 10000, avg: 0.08
Ep: 2000, n: 10000, avg: 0.276
Ep: 3000, n: 10000, avg: 0.304
Ep: 4000, n: 10000, avg: 0.424
Ep: 5000, n: 10000, avg: 0.516
Ep: 6000, n: 10000, avg: 0.504
Ep: 7000, n: 10000, avg: 0.616
Ep: 8000, n: 10000, avg: 0.6
Ep: 9000, n: 10000, avg: 0.56
Ep: 10000, n: 10000, avg: 0.528
Ep: 11000, n: 10000, avg: 0.596
Ep: 12000, n: 10000, avg: 0.624
Ep: 13000, n: 10000, avg: 0.64
Ep: 14000, n: 10000, avg: 0.576
Ep: 15000, n: 10000, avg: 0.544
Ep: 16000, n: 10000, avg: 0.624
Ep: 17000, n: 10000, avg: 0.556
Ep: 18000, n: 10000, avg: 0.656
Ep: 19000, n: 10000, avg: 0.632
Ep: 1000, n: 20000, avg: -0.156
Ep: 2000, n: 20000, avg: 0.1
Ep: 3000, n: 20000, avg: 0.2
Ep: 4000, n: 20000, avg: 0.464
Ep: 5000, n: 20000, avg: 0.424
Ep: 6000, n: 20000, avg: 0.536
Ep: 7000, n: 20000, avg: 0.576
Ep: 8000, n: 20000, avg: 0.576
Ep: 9000, n: 20000, avg: 0.56
Ep: 10000, n: 20000, avg: 0.548
Ep: 11000, n: 20000, avg: 0.66
Ep: 12000, n: 20000, avg: 0.612
Ep: 13000, n: 20000, avg: 0.588
Ep: 14000, n: 20000, avg: 0.6
Ep: 15000, n: 20000, avg: 0.62
Ep: 16000, n: 20000, avg: 0.612
Ep: 17000, n: 20000, avg: 0.624
Ep: 18000, n: 20000, avg: 0.596
Ep: 19000, n: 20000, avg: 0.632
Ep: 1000, n: 30000, avg: 0.072
Ep: 2000, n: 30000, avg: 0.248
Ep: 3000, n: 30000, avg: 0.272
```

```
Ep: 4000, n: 30000, avg: 0.352
Ep: 5000, n: 30000, avg: 0.396
Ep: 6000, n: 30000, avg: 0.5
Ep: 7000, n: 30000, avg: 0.572
Ep: 8000, n: 30000, avg: 0.604
Ep: 9000, n: 30000, avg: 0.54
Ep: 10000, n: 30000, avg: 0.624
Ep: 11000, n: 30000, avg: 0.6
Ep: 12000, n: 30000, avg: 0.596
Ep: 13000, n: 30000, avg: 0.572
Ep: 14000, n: 30000, avg: 0.604
Ep: 15000, n: 30000, avg: 0.604
Ep: 16000, n: 30000, avg: 0.644
Ep: 17000, n: 30000, avg: 0.624
Ep: 18000, n: 30000, avg: 0.656
Ep: 19000, n: 30000, avg: 0.648
Ep: 1000, n: 40000, avg: 0.104
Ep: 2000, n: 40000, avg: 0.104
Ep: 3000, n: 40000, avg: 0.192
Ep: 4000, n: 40000, avg: 0.428
Ep: 5000, n: 40000, avg: 0.488
Ep: 6000, n: 40000, avg: 0.56
Ep: 7000, n: 40000, avg: 0.468
Ep: 8000, n: 40000, avg: 0.568
Ep: 9000, n: 40000, avg: 0.56
Ep: 10000, n: 40000, avg: 0.616
Ep: 11000, n: 40000, avg: 0.652
Ep: 12000, n: 40000, avg: 0.612
Ep: 13000, n: 40000, avg: 0.572
Ep: 14000, n: 40000, avg: 0.592
Ep: 15000, n: 40000, avg: 0.552
Ep: 16000, n: 40000, avg: 0.54
Ep: 17000, n: 40000, avg: 0.608
Ep: 18000, n: 40000, avg: 0.724
Ep: 19000, n: 40000, avg: 0.62
```

[27]:
```
len(results[n]['test'],)
```

[27]:
```
19
```

[38]:
```
x_train = (np.arange(N // train_avg_every)+1) * train_avg_every
x_test = (np.arange(N // test_avg_every)+1)[:-1] * test_avg_every
for n in results:
    plt.plot(x_train, results[n]['train'], '--', label=f"{n} train")
    plt.plot(x_test, results[n]['test'], '-', label=f"{n} test")

plt.plot(x, train, '--', label=f"eps = const")
```

```
plt.plot(x_test, test[:-1], '-', label=f"eps = const")

plt.rcParams["figure.figsize"] = (20,20)
plt.legend(loc="lower right")
plt.title('Success measurement on train')
plt.xlabel("Number of games")
plt.ylabel("exploration Mopt")
plt.show()
```

It's fair to conclude regarding different n^* that: - eps = const — slowest learning, not the best results - n* = 1 learns fast - n* = 10k learns slower, but has results better that n* = 1 and one of the best models - n* = 20k constant growing and best results at the end - n* = 30k learns slower, results aren't the best - n* = 40k worst model so far, slowest learning

2 Good experts and bad experts

Choose the best value of n^* that you found in the previous section. Run Q-learning against Opt (ϵ_{opt}) for different values of ϵ_{opt} for 20′000 games - switch the 1st player after every game. Choose several values of ϵ_{opt} from a reasonably wide interval between 0 to 1− particularly, include $\epsilon_{\text{opt}} = 0$.

Question 4. After every 250 games during training, compute the 'test' M_{opt} and M_{rand} for your agents - for each value of ϵ_{opt}. Plot M_{opt} and M_{rand} over time. What do you observe? How can you explain it? Expected answer: A figure showing M_{opt} and M_{rand} over time for different values of ϵ_{opt} (caption length $<$ 250 words).

Question 5. What are the highest values of M_{opt} and M_{rand} that you could achieve after playing 20 '000 games?

Question 6. (Theory) Assume that Agent 1 learns by playing against Opt (0) and find the optimal Qvalues $Q_1(s, a)$. In addition, assume that Agent 2 learns by playing against Opt (1) and find the optimal Q-values $Q_2(s, a)$. Do $Q_1(s, a)$ and $Q_2(s, a)$ have the same values? Justify your answer. (answer length $<$ 150 words)

2.1 Learning by self-practice

In this section, your are supposed to ask whether Q-learning can learn to play Tic Tac Toe by only playing against itself. For different values of $\epsilon \in [0, 1)$, run a Q-learning agent against itself for 20 '000 games - i.e. both players use the same set of Q-values and update the same set of Q-values.

Question 7. After every 250 games during training, compute the 'test' M_{opt} and M_{rand} for different values of $\epsilon \in [0, 1)$. Does the agent learn to play Tic Tac Toe? What is the effect of ϵ ?
Expected answer: A figure showing M_{opt} and M_{rand} over time for different values of $\epsilon \in [0, 1)$ (caption length $<$ 100 words).

For rest of this section, use $\epsilon(n)$ in Equation 1 with different values of n^*- instead of fixing ϵ.

Question 8. After every 250 games during training, compute the 'test' M_{opt} and M_{rand} for your agents. Does decreasing ϵ help training compared to having a fixed ϵ ? What is the effect of n^* ?
Expected answer: A figure showing M_{opt} and M_{rand} over time for different values of speeds of n^* (caption length $<$ 100 words) .

Question 9. What are the highest values of M_{opt} and M_{rand} that you could achieve after playing 20 '000 games?

Question 10. For three board arrangements (i.e. states s), visualize Q-values of available actions (e.g. using heat maps). Does the result make sense? Did the agent learn the game well?
Expected answer: A figure with 3 subplots of 3 different states with Q-values shown at available actions (caption length $<$ 200 words).

3 Deep Q-Learning

As our 2nd algorithm, we use Deep Q-Learning (DQN) combined with ϵ-greedy policy. You can watch again Part 1 of Deep Reinforcement Learning Lecture 1 for an introduction to DQN and Part 1 of Deep Reinforcement Learning Lecture 2 (in particular slide 8) for more details. The idea in DQN is to approximate Q-values by a neural network instead of a look-up table as in Tabular Q-learning. For implementation, you can use ideas from the DQN tutorials of Keras and PyTorch.

```
[ ]: # import numpy as np
     # import torch
     # import torch.nn as nn
     # import torch.nn.functional as F
     # from torch.utils.data import DataLoader
     # from torchvision.datasets import MNIST
     # from torchvision.transforms import ToTensor


     # class DQLNN(nn.Module):
     #     """ DQLNN, expects input shape (3, 3, 2) """
     #     def __init__(self):
     #         super(DQLNN, self).__init__()

     #         self.fc1 = nn.Linear(3*3*2, 128)
     #         self.fc2 = nn.Linear(128, 128)
     #         self.fc3 = nn.Linear(128, 9)

     #     def forward(self, x):
     #         return self.fc3(F.relu(self.fc2(F.relu(self.fc1(x.flatten())))))

     # model_dql = DQLNN()
```

```
[ ]: # Player reinitializing
```

3.1 Learning from experts

Implement the DQN algorithm. To check the algorithm, run a DQN agent with a fixed and arbitrary $\epsilon \in [0, 1)$ against Opt (0.5) for $20'000$ games - switch the 1st player after every game.

Question 11. Plot average reward and average training loss for every 250 games during training. Does the loss decrease? Does the agent learn to play Tic Tac Toe?
Expected answer: A figure with two subplots (caption length < 50 words). Specify your choice of ϵ.

Question 12. Repeat the training but without the replay buffer and with a batch size of 1 : At every step, update the network by using only the latest transition. What do you observe?
Expected answer: A figure with two subplots showing average reward and average training loss during training (caption length < 50 words).

Instead of fixing ϵ, use $\epsilon(n)$ in Equation 1. For different values of n^*, run your DQN against Opt (0.5) for 20'000 games - switch the 1st player after every game. Choose several values of n^* from a reasonably wide interval between 1 to $40'000$ - particularly, include $n^* = 1$.

Question 13. After every 250 games during training, compute the 'test' M_{opt} and M_{rand} for your agents. Plot M_{opt} and M_{rand} over time. Does decreasing ϵ help training compared to having a fixed ϵ ? What is the effect of n^* ?
Expected answer: A figure showing M_{opt} and M_{rand} over time for different values of opt (caption length < 250 words).

Choose the best value of n^* that you found. Run DQN against Opt (ϵ_{opt}) for different values of

ϵ_{opt} for 20 '000 games - switch the 1st player after every game. Choose several values of ϵ_{opt} from a reasonably wide interval between 0 to 1− particularly, include $\epsilon_{\text{opt}} = 0$.

Question 14. After every 250 games during training, compute the 'test' M_{opt} and M_{rand} for your agents - for each value of ϵ_{opt}. Plot M_{opt} and M_{rand} over time. What do you observe? How can you explain it? Expected answer: A figure showing M_{opt} and M_{rand} over time for different values of ϵ_{opt} (caption length < 250 words).

Question 15. What are the highest values of M_{opt} and M_{rand} that you could achieve after playing 20 '000 games?

3.2 Learning by self-practice

For different values of $\epsilon \in [0, 1)$, run a DQN agent against itself for 20 ' 000 games - i.e. both players use the same neural network and share the same replay buffer. Important note: For one player, you should

add states s_t and $s_{t'}$ as `x_t` and `x_tp` to the replay buffer, but for the other player, you should first swap the opponent positions (x_t $[:, :, 1]$ and x_- tp$[:, :, 1]$) with the agent's own positions (x_t $[:, :, 0]$ and x_- tp$[:, :, 0]$) and then add them to the replay buffer.

Question 16. After every 250 games during training, compute the 'test' M_{opt} and M_{rand} for different values of $\epsilon \in [0, 1)$. Plot M_{opt} and M_{rand} over time. Does the agent learn to play Tic Tac Toe? What is the effect of ϵ ?
Expected answer: A figure showing M_{opt} and $ M_{rand}$ over time for different values of $\epsilon \in [0, 1)$ (caption length < 100 words) .

Instead of fixing ϵ, use $\epsilon(n)$ in Equation 1 with different values of n^*.

Question 17. After every 250 games during training, compute the 'test' M_{opt} and M_{rand} for your agents. Plot M_{opt} and M_{rand} over time. Does decreasing ϵ help training compared to having a fixed ϵ ? What is the effect of n^* ?
Expected answer: A figure showing M_{opt} and M_{rand} over time for different values of speeds of n^* (caption length < 100 words) .

Question 18. What are the highest values of M_{opt} and M_{rand} that you could achieve after playing 20 '000 games?

Question 19. For three board arrangements (i.e. states s), visualize Q-values of available actions (e.g. using heat maps). Does the result make sense? Did the agent learn the game well?
Expected answer: A figure with 3 subplots of 3 different states with Q-values shown at available actions (caption length < 200 words).

4 Comparing Q-Learning with Deep Q-Learning

We define the training time T_{train} as the number of games an algorithm needs to play in order to reach 80% of its final performance according to both M_{opt} and M_{rand} .

Question 20. Include a table showing the best performance (the highest M_{opt} and M_{rand}) of Q-Learning and DQN (both for learning from experts and for learning by self-practice) and their corresponding training time.
Expected answer: A table showing 12 values.

Question 21. Compare your results for DQN and Q-Learning (answer length < 300 words).

[ ]: