The "Tennis" Challenge

Introduction

The environment consists of a two-player game where agents control rackets to bounce a ball over a net. The objective is to bounce the ball between one another while not dropping or sending the ball out of bounds.

The solution I use is based on the solution of the "Reacher" project described here: https://github.com/DrSdl/RIL/tree/master/Reacher.

In the "Reacher" project I used the DDPG (deep deterministic policy gradient, see for example https://spinningup.openai.com/en/latest/algorithms/ddpg.html or https://arxiv.org/abs/1509.02971) algorithm to train the agent.

In the Tennis challenge I also used the DDPG algorithm in the form of a Multi Agent DDPG, or MADDPG for short. A thorough description of MADDPG is given here:

- Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments: https://arxiv.org/abs/1706.02275
- OpenAI's blog on MADDPG: https://openai.com/blog/learning-to-cooperate-compete-and-communicate/
- MADDPG sample code: https://github.com/openai/maddpg

In a multiagent environment there is a natural curriculum — the difficulty of the environment is determined by the skill of the other agents. This also means that usually there is no stable equilibrium: no matter how smart an agent is, the actions of the other agents exert pressure to get smarter.

Implementation

(1) The Actor and Critic network
The Actor and critic network are identical to the network and code of the "Reacher" project as described in this report: https://github.com/DrSdl/RIL/blob/master/Reacher/Report.pdf and as implemented here: https://github.com/DrSdl/RIL/blob/master/Reacher/Reacher.ipynb

(2) The MADDPG loop
The objective was to run 2000 episodes with a maximum of 1000 steps per episode. In this first attempt of implementing MADDPG there was no central critic – but two independent critics per agent. Therefore, the instantiation of the MADDPG loop was straightforward:

```
agent1 = Agent(state_size=states.shape[1], action_size=brain.vector_action_space_size, random_seed=10)
agent2 = Agent(state_size=states.shape[1], action_size=brain.vector_action_space_size, random_seed=20)
```

Before each episode the variables are initialized as follows:

```
def maddpg(n_episodes=2000, max_t=1001):
    scores_deque1 = deque(maxlen=100)
    scores_deque2 = deque(maxlen=100)
    scores_deque3 = deque(maxlen=100)
    scores1 = []
    scores2 = []
```

```
scores3 = []
max_score = -np.Inf
for i_episode in range(1, n_episodes+1):
    env_info = env.reset(train_mode=True)[brain_name]
    states = env_info.vector_observations
    #
    agent1.reset()
    agent2.reset()
    score1 = 0.0
    score2 = 0.0
    zeit=time.time()
```

Scores1 and Scores2 hold the scores of agent 1 and agent 2 respectively. Scores3 contains the score of each episode.

Each episode is run with the following for-loop:

```
   for t in range(max_t):
        #actions      = np.array([agent.act(states[0],add_noise=False)])
        actions1       = agent1.act(states[0],add_noise=False)
        actions2       = agent2.act(states[1],add_noise=False)
        #print(actions, type(actions[0]))
        actions = np.concatenate((actions1,actions2), axis=0)
        actions = np.reshape(actions,(1,4))
        #
        env_info     = env.step(actions)[brain_name]
        next_states  = env_info.vector_observations
        #print(next_states)
        rewards      = env_info.rewards
        #print(rewards)
        dones        = env_info.local_done
        #next_state, reward, done, _ = env.step(action)
        #agent.step(states[0], actions[0], rewards[0], next_states[0], dones[0])
        agent1.step(states[0], actions1, rewards[0], next_states[0], dones[0])
        agent2.step(states[1], actions2, rewards[1], next_states[1], dones[1])
        #
        states = next_states
        score1 += rewards[0]
        score2 += rewards[1]
        #print(actions)
        if t%10 == 0:
            agent1.do_learn()
            agent2.do_learn()
        if np.any(dones):
            break
```

After 10 steps each agent does a learning exercise.

Finally, the scores per episode of each agent are summed without discounting and stored in variables score1 and score2. The maximum of score1 and score2 is calculated and stored in score3:

```
scores_deque1.append(score1)
scores1.append(score1)
scores_deque2.append(score2)
scores2.append(score2)
#
scores_deque3.append(np.max([score1, score2]))
scores3.append(np.max([score1,score2]))
```

Results

If an agent hits the ball over the net, it receives a reward of +0.1. If an agent lets a ball hit the ground or hits the ball out of bounds, it receives a reward of -0.01. Thus the goal of each agent is to keep the ball in play.

The task is episodic and learning is considered completed if the agents get an average score of +0.5 (over 100 consecutive episodes). Specifically, after each episode the sum of rewards of each agent is calculated and the maximum of the two sums is taken. This is defined as the single score per episode.

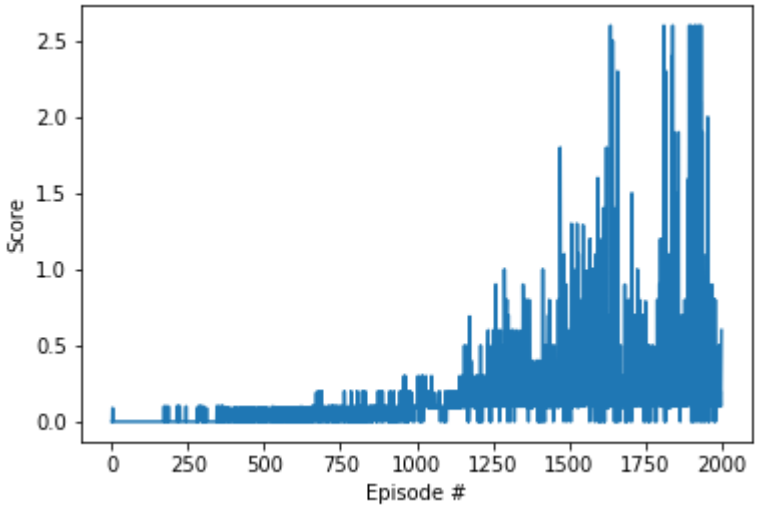
Hyperparameters used are:

- replay buffer size = 1e6
- minibatch size = 1024
- discount factor = 0.99
- tau for soft update of target parameters = 1e-3
- learning rate of the actor = 1e-3
- learning rate of the critic = 1e-3
- L2 weight decay = 0.0

The following scores were obtained:

```
Episode 100      Average Score: 0.00
Episode 200      Average Score: 0.01
Episode 300      Average Score: 0.02
Episode 400      Average Score: 0.02
Episode 500      Average Score: 0.05
Episode 600      Average Score: 0.04
Episode 700      Average Score: 0.05
Episode 800      Average Score: 0.04
Episode 900      Average Score: 0.07
Episode 1000     Average Score: 0.08
Episode 1100     Average Score: 0.11
Episode 1200     Average Score: 0.16
Episode 1300     Average Score: 0.24
Episode 1400     Average Score: 0.24
Episode 1500     Average Score: 0.26
Episode 1600     Average Score: 0.41
Episode 1700     Average Score: 0.42
Episode 1800     Average Score: 0.30
Episode 1900     Average Score: 0.61
Episode 2000     Average Score: 0.54
```

After episode 1800 the average score is 0.61 and after episode 1900 it is 0.54 until episode 2000. The following plot shows the full score history per episode.



Further improvements

Next improvement steps are:

- Recode the solution in order to have only a single critic for all agents. This means that the Critic is instantiated centrally and outside of the Actor class. This central Critic then has a single replay buffer which is feed with the experience of all agents. Expectation is that the speed of learning increases.
- In the above example each actor only received its local observations, i.e. the list of 8 variables belonging to its own state. It would be interesting to see how learning progresses if each actor would receive the observations of the other actors, too. The rationale for this is that in a real game the opponents also watch each other's movements and that at least some of each agent's local observables are accessible to all other agents, too.