Collaboration and Competition Report

Objective (Reproduced from Readme)

In this environment, two agents control rackets to bounce a ball over a net. 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 observation space consists of 8 variables corresponding to the position and velocity of the ball and racket. Each agent receives its own, local observation. Two continuous actions are available, corresponding to movement toward (or away from) the net, and jumping.
The task is episodic, and in order to solve the environment, your agents must get an average score of +0.5 (over 100 consecutive episodes, after taking the maximum over both agents). Specifically,
After each episode, we add up the rewards that each agent received (without discounting), to get a score for each agent. This yields 2 (potentially different) scores. We then take the maximum of these 2 scores. This yields a single score for each episode.
The environment is considered solved, when the average (over 100 episodes) of those score is at least +0.5.

Motivation

This project uses a DDPG similar to the Navigation project. However, here we have two players (agents) learning their own actor and shared critic networks while playing together. Note: Most of the code is very similar to what the Navigation project; except, for some changes needed for training two agents and decaying the noise process as the agents learned.

For each agent and time step, the Agent acts upon the state using a shared (at class level) replay buffer, local critic, target critic and optimizer for the critic networks with their own local actor, target actor, and optimizer for the actor networks.

One needed change from the navigation project was to really make sure Exploration happens early and then tapers off. That was accomplished by making sure the Ornstein-Uhlenbeck Noise process decays as we learn

Learning Algorithm

Below is the DDPG algorithm, the hyperparameters used, and the NN architecture for the agent and critic.

DDPG algorithm copied from: https://towardsdatascience.com/introduction-to-various-reinforcement-learning-algorithms-i-q-learning-sarsa-dqn-ddpg-72a5e0cb6287

Algorithm 1 DDPG algorithm

```
Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights $\theta^Q$ and $\theta^\mu$.
Initialize target network $Q'$ and $\mu'$ with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer $R$
**for** episode = 1, M **do**
    Initialize a random process $\mathcal{N}$ for action exploration
    Receive initial observation state $s_1$
    **for** t = 1, T **do**
        Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
        Execute action $a_t$ and observe reward $r_t$ and observe new state $s_{t+1}$
        Store transition $(s_t, a_t, r_t, s_{t+1})$ in $R$
        Sample a random minibatch of $N$ transitions $(s_i, a_i, r_i, s_{i+1})$ from $R$
        Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
        Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
        Update the actor policy using the sampled policy gradient:
            $$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s_i}$$
        Update the target networks:
            $$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
            $$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$
    **end for**
**end for**
```

Hyperparameters

```python
# parameters
BUFFER_SIZE = int(1e5)  # replay buffer size
BATCH_SIZE = 256        # minibatch size
GAMMA = 0.99            # discount factor
TAU = 0.02              # for soft update of target parameters
LR_ACTOR = 0.001        # learning rate of the actor
LR_CRITIC = 0.001       # learning rate of the critic
WEIGHT_DECAY = 0.0      # L2 weight decay
LEARN_UPDATES = 4       # number of updates
UPDATE_EVERY = 2        # how often network is updated
EPSILON = 2.0           # amplitude
EPSILON_DECAY = 0.9999  # decay rate
SIGMA = 0.2             # var for Ornstein-Uhlenbeck process
```

NN

Agent:

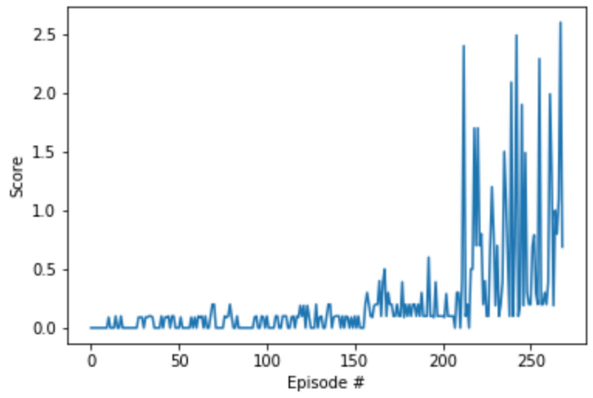
```python
nn.Linear(state_size, fc1_units),  # 128 units, fully connected layer
nn.ReLU(),
nn.BatchNorm1d(fc1_units),
nn.Linear(fc1_units, fc2_units),  # 128 units, fully connected layer
nn.ReLU(),
nn.BatchNorm1d(fc2_units),
nn.Linear(fc2_units, action_size),
nn.Tanh()  # delivers output between -1,1
```

Critic:

```python
nn.Linear(state_size, fc1_units),  # 128 units, fully connected layer
nn.ReLU(),
nn.BatchNorm1d(fc1_units)
nn.Linear((fc1_units + action_size), fc2_units),  # 128 units + 4 actions, fully
connected layer
nn.ReLU(),
nn.Linear(fc2_units, 1)
```

Plot of Rewards

```
Episode 100     Average Score Last 100 Episodes: 0.03830        Max Score (All Agents) Last E
pisode: 0.00
Episode 200     Average Score Last 100 Episodes: 0.11940        Max Score (All Agents) Last E
pisode: 0.10
Episode 269     Average Score Last 100 Episodes: 0.50100        Max Score (All Agents) Last E
pisode: 0.69
Environment solved in 169 episodes!     Average Score: 0.50100
```



Ideas for Future Work

- Implement PPO algorithm
- Implement A3C algorithm
- Implement D4PG algorithm
- Test Different Neural Net architectures for the Agent and Critic
- Hyperparameter tuning