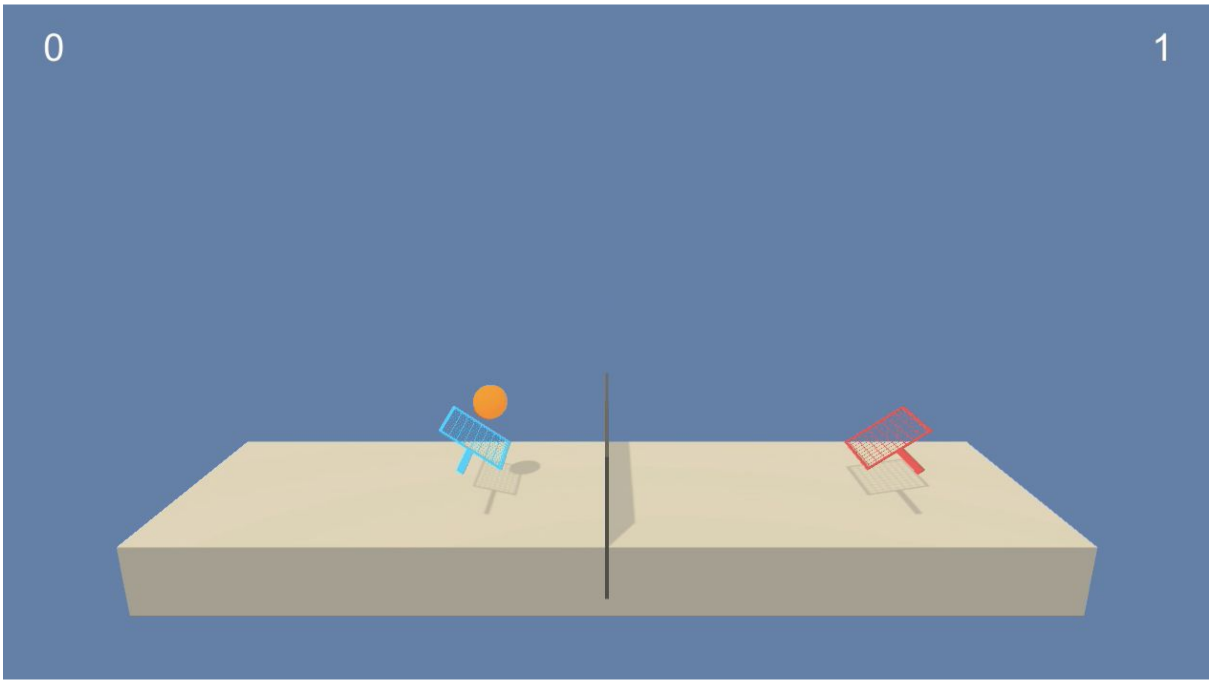
Project: Collaboration and Competition (Part of Udacity Deep Reinforcement Learning Nanodegree)

by Michael Strobl

Goal of this project:



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 **scores** is at least +0.5.

Used Neural Networks:

Actor

```
class Actor(nn.Module):
    ''' Network of the actor model (Policy) '''
    def __init__(self, state_size, action_size, seed, fc1_units=512, fc2_units=256):
        '''
        ------------------------------------------------
        Parameters

        state_size:  # of states
        action_size: # of actions
        seed:        random seed
        fc1_units:   # of nodes in first hidden layer
        fc2_units:   # of nodes in second hidden layer
        ------------------------------------------------
        '''

        super(Actor, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1 = nn.Linear(state_size, fc1_units)
        self.fc2 = nn.Linear(fc1_units, fc2_units)
        self.fc3 = nn.Linear(fc2_units, action_size)
        self.bn1 = nn.BatchNorm1d(state_size)
        self.reset_parameters()
```

- Fully connected layer - input: 24 (state size), output: 512 (fc1_units)
- Fully connected layer - input: 512 (fc1_units), output 256 (fc2_units)
- Fully connected layer - input: 256 (fc1_units), output: 2 (action size)
- Batch Normalization to speed up neural network (Source: https://towardsdatascience.com/batch-normalization-in-neural-networks-1ac91516821c)

Critic

```python
class Critic(nn.Module):
    ''' Network of critic model (Value) '''
    def __init__(self, state_size, action_size, seed, fc1_units=512, fc2_units=256):
        '''
        -----------------------------------------------
        Parameters

        state_size:  # of states
        action_size: # of actions
        seed:        random seed
        fc1_units:   # of nodes in first hidden layer
        fc2_units:   # of nodes in second hidden layer
        -----------------------------------------------
        '''

        super(Critic, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1 = nn.Linear(state_size, fc1_units)
        self.fc2 = nn.Linear(fc1_units+action_size, fc2_units)
        self.fc3 = nn.Linear(fc2_units, 1)
        self.bn1 = nn.BatchNorm1d(state_size)
        self.dropout = nn.Dropout(p=0.2)
        self.reset_parameters()
```

- Fully connected layer - input: 24 (state size), output: 512 (fc1_units)
- Fully connected layer - input: 512 (fc1_units), output 256 (fc2_units)
- Fully connected layer - input: 256 (fc1_units), output: 2 (action size)
- Batch Normalization to speed up neural network (Source: https://towardsdatascience.com/batch-normalization-in-neural-networks-1ac91516821c)

Used Learning Algorithm:

MADDPG (Multi-Agent Deep Deterministic Policy Gradient (DDPG)

Algorithm 1: Multi-Agent Deep Deterministic Policy Gradient for N agents

for episode $= 1$ to M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial state $\mathbf{x}$
  for $t = 1$ to max-episode-length **do**
    for each agent i, select action $a_i = \boldsymbol{\mu}_{\theta_i}(o_i) + \mathcal{N}_t$ w.r.t. the current policy and exploration
    Execute actions $a = (a_1, \ldots, a_N)$ and observe reward r and new state $\mathbf{x}'$
    Store $(\mathbf{x}, a, r, \mathbf{x}')$ in replay buffer $\mathcal{D}$
    $\mathbf{x} \leftarrow \mathbf{x}'$
    for agent $i = 1$ to N **do**
      Sample a random minibatch of S samples $(\mathbf{x}^j, a^j, r^j, \mathbf{x}'^j)$ from $\mathcal{D}$
      Set $y^j = r_i^j + \gamma Q_i^{\boldsymbol{\mu}'}(\mathbf{x}'^j, a'_1, \ldots, a'_N)|_{a'_k = \boldsymbol{\mu}'_k(o_k^j)}$
      Update critic by minimizing the loss $\mathcal{L}(\theta_i) = \frac{1}{S}\sum_j \left(y^j - Q_i^{\boldsymbol{\mu}}(\mathbf{x}^j, a_1^j, \ldots, a_N^j)\right)^2$
      Update actor using the sampled policy gradient:

$$\nabla_{\theta_i} J \approx \frac{1}{S}\sum_j \nabla_{\theta_i}\boldsymbol{\mu}_i(o_i^j)\nabla_{a_i} Q_i^{\boldsymbol{\mu}}(\mathbf{x}^j, a_1^j, \ldots, a_i, \ldots, a_N^j)\big|_{a_i=\boldsymbol{\mu}_i(o_i^j)}$$

    end for
    Update target network parameters for each agent i:

$$\theta'_i \leftarrow \tau\theta_i + (1-\tau)\theta'_i$$

  end for
end for

```python
def maddpg(n_episodes=2000, max_t=1000):
    '''
    -------------------------------------------
    Parameters

    n_episodes: # of episodes that the agent is training for
    max_t:      # of time steps (max) the agent is taking per episode
    -------------------------------------------
    '''
    scores_deque = deque(maxlen=100)
    scores = []
    max_score = -np.inf
    for i_episode in range(1, n_episodes+1):
        env_info = env.reset(train_mode=True)[brain_name]           # turn on train mode of the environment
        states = env_info.vector_observations                       # get the current state for each agent
        maddpg.reset()                                              # reset the OU noise parameter
        ep_scores = np.zeros(num_agents)                            # initialize the score for each agent
        for t in range(max_t):
            actions = maddpg.act(states)                            # select an action for each agent
            env_info = env.step(actions)[brain_name]                # send all actions to the environment
            next_states = env_info.vector_observations              # get next state for each agent
            rewards = env_info.rewards                              # get reward for each agent
            dones = env_info.local_done                             # check if episode finished
            maddpg.step(states, actions, rewards, next_states, dones) # agents record enviroment response in recent step
            states = next_states                                    # set the state as the next state for the following step for each agent
            ep_scores += rewards                                    # update the total score
            if np.any(dones):                                       # exit loop if episode for any agent finished
                break

        scores_deque.append(np.max(ep_scores))
        scores.append(ep_scores)

        # print average epsiode score and average 100-episode score for each episode
        print('\rEpisode {} \tMax Score: {:.2f} \tAverage Max Score: {:.2f}'.format(i_episode, np.max(ep_scores), np.mean(scores_deque)), end="")

        # print and save actor and critic weights when a score of +30 over 100 episodes has been achieved
        if np.mean(scores_deque) >= 0.5:
            for i in range(config.num_agents):
                torch.save(maddpg.maddpg_agents[i].actor_local.state_dict(), 'checkpoint_actor_{}_final.pth'.format(i))
                torch.save(maddpg.maddpg_agents[i].critic_local.state_dict(), 'checkpoint_critic_{}_final.pth'.format(i))
            print('\nEnvironment solved in {:d} episodes!\tAverage Max Score: {:.2f}'.format(i_episode, np.mean(scores_deque)))
            break
    return scores
```

Source: https://github.com/openai/maddpg
Source: https://arxiv.org/abs/1706.02275

Hyperparameters:

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 = 0.0003 # learning rate of the actor
LR_CRITIC = 0.0003 # learning rate of the critic
WEIGHT_DECAY = 0 # L2 weight decay
UPDATE_EVERY = 4 # update every 4 time steps

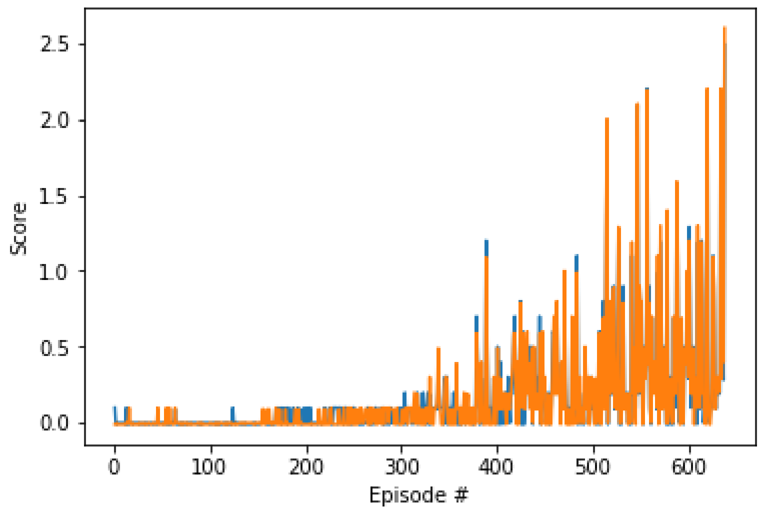
Source:
https://github.com/udacity/deep-reinforcement-learning/blob/master/ddpg-bipedal/ddpg_agent.py

Ornstein-Uhlenbeck Noise:
mu = 0 # long-running mean
theta = 0.15 # the speed of mean reversion
sigma = 0.2 # the volatility parameter

Source: https://planetmath.org/ornsteinuhlenbeckprocess

Best result:

Episode 638 Max Score: 2.60 Average Max Score: 0.52
Environment solved in **638 episodes**! Average Max Score: 0.52



Future:

- Try more different neural networks and batch norms
- Change Tau and Gamma to see if the results get better
- Try more different batch and buffer sizes

Appendix:

Best 10 results

	model.py											
	Actor		Critic		Tennis.ipynb		MADDPG_agent.py			DDPG		
Type	fc1_units	fc2_units	fc1_units	fc2_units	Episodes	Average Score	buffer_size	batch_size	gamma	tau	LR_ACTOR	LR_CRITIC
MADDPG	512	256	512	256	638	0.52	100,000	256	0.99	0.02	0.0003	0.0003
MADDPG	512	256	512	256	705	0.51	1,000,000	256	0.99	0.02	0.0005	0.0005
MADDPG	512	256	512	256	727	0.51	1,000,000	256	0.99	0.02	0.0003	0.0003
MADDPG	512	256	512	256	782	0.50	1,000,000	256	0.99	0.02	0.0002	0.0003
MADDPG	512	256	512	256	829	0.50	1,000,000	256	0.99	0.02	0.0004	0.0003
MADDPG	512	256	512	256	836	0.50	1,000,000	256	0.99	0.02	0.0001	0.0003
MADDPG	512	256	512	256	877	0.52	1,000,000	256	0.99	0.02	0.0001	0.0003
MADDPG	512	256	512	256	974	0.51	1,000,000	256	0.99	0.02	0.0002	0.0002
MADDPG	512	256	512	256	989	0.50	1,000,000	256	0.99	0.02	0.0001	0.0003
MADDPG	512	256	512	256	1055	0.5	100,000	256	0.99	0.02	0.0005	0.0005

Sources:

https://arxiv.org/abs/1706.02275
https://github.com/openai/maddpg
https://github.com/udacity/deep-reinforcement-learning/tree/master/ddpg-pendulum
https://github.com/udacity/deep-reinforcement-learning/tree/master/ddpg-bipedal
https://github.com/udacity/deep-reinforcement-learning/tree/master/p2_continuous-control
https://planetmath.org/ornsteinuhlenbeckprocess
https://towardsdatascience.com/batch-normalization-in-neural-networks-1ac91516821c
https://towardsdatascience.com/deep-deterministic-policy-gradients-explained-2d94655a9b7b