Collaboration and Competition

This document provides description of the implementation for the Collaboration and Competition project.

The Environment

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.

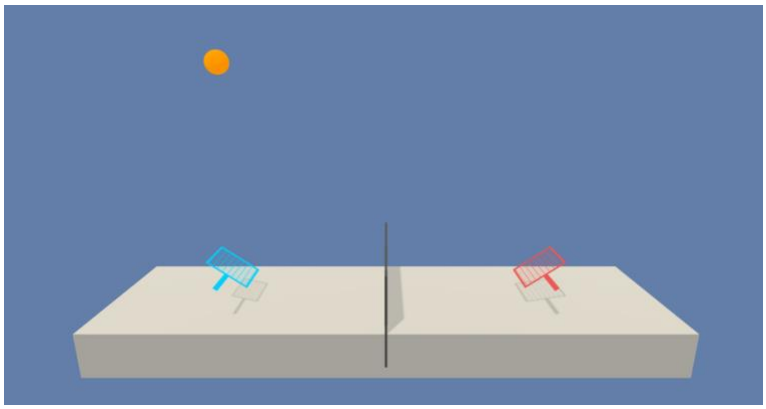

Figure 1. Unity ML-Agents 'Tennis' Environment

Solving the Environment

The task is episodic, and in order to solve the environment, agents must get an average score of +0.5 (over 100 consecutive episodes, after taking the maximum over both agents). Specifically,

- After each episode, the rewards that each agent received (without discounting) are added up, to get a score for each agent. This yields 2 (potentially different) scores. Then the maximum of these 2 scores is taken.

- 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.

The Implementation

The implementation uses **Multi-Agent DDPG (Deep Deterministic Policy Gradients)** to solve the task. Two DDPG agents interact with and learns from the Unity ML environment 'Tennis' for the task.

The agent implementation is written in PyTorch and the Python API was used to control the agent.

The solution has following files:

- ***Tennis.ipynb***: Jupyter notebook, code for training and testing the agent.

- ***maddpg.py***: The MADGPP class. The MADGPP class implements Multi-Agent DDPG behavior. It instantiates two independent DDGP agents, keeps a common replay memory and, orchestrates the act and step processes of its two DDGP agents.

- ***ddpg.py***: The DDPGAgent class. The Agent class implements DDPG agent.

- ***buffer.py***: The ReplayBuffer class. The sampled batch of experience tuples is provided thru ReplayBuffer class. ReplayBuffer class implements a fixed-size buffer to store experience tuples.

- ***OUNoise.py***: The OUNoise class. OUNoise class implements Ornstein-Uhlenbeck noise process.

- ***model.py***: The Actor and Critic classes. Defines neural network architectures for actor (policy) and critic (value) networks.

- ***checkpoint_actor0.pth***: Saved model weights for the Actor network of the first trained agent.

- ***checkpoint_critic0.pth***: Saved model weights for the Critic network of the first trained agent.

- ***checkpoint_actor1.pth***: Saved model weights for the Actor network of the second trained agent.

- ***checkpoint_critic1.pth***: Saved model weights for the Critic network of the second trained agent.

The **Actor** class has following methods:

- *__init__*: Initializes parameters and builds model
- *reset parameters*: Resets parameters
- *forward*: Builds an actor (policy) network that maps states -> actions

The **Critic** class has following methods:

- *__init__*: Initializes parameters and builds model
- *reset parameters*: Resets parameters
- *forward*: Builds a critic (value) network that maps states -> actions

The **MADDGP** class has following methods:

- *__init__*: Initializes the MADDGP object.
- *act*: Returns actions from all agents in the MADDGP object.
- *step*: Saves experience in replay memory and triggers learning of DDGP agents.
- *reset parameters*: Resets parameters

The **DDGPAgent** class has following methods:

- *__init__*: Initializes an DDGPAgent object.
- *act*: Returns actions for given state as per current policy.
- *learn*: Updates value parameters using given batch of experience tuples.
- *reset*: Resets parameters
- *soft_update*: Soft updates model parameters.

The **OUNoise** class has following methods:

- *__init__*: Initializes parameters and noise process.
- *reset*: Reset the internal state (=noise) to mean (=mu).
- *sample*: Updates internal state and returns it as a noise sample.

The **ReplayBuffer** class has following methods:

- *__init__*: Initializes a ReplayBuffer object.
- *add*: Adds a new experience to memory.
- *sample*: Randomly samples a batch of experiences from memory.
- *__len__*: Returns the current size of internal memory.

The architecture mainly consists of one MADDGP object and two independent DDGP agents. MADDGP object interacts with the environment, keeps the common replay memory and, orchestrates its two independent DDGP agents. Each agent has its own copy of Actor and Critic networks. DDGP agents learn from the experiences of two players stored in the common replay

memory. Each DDPG agent tries to predict its action with only its own state observation during execution.

The implementation could be less efficient compared to a more integrated design, but it has the simplicity of following 2 normal DDGP agent implementations.

The multi-agent code was implemented in such a way that after each step:

- MADDGP object adds all experiences to a replay buffer that is shared by all agents, and
- Triggers learning of its two DDGP agents using a sample from the replay buffer.
- The (local) actor and critic networks are updated of the DDGP agents independently.

DDPG Agents' Actor and Critic network architectures consists of fully connected layers. Input layers have 33 inputs (corresponding to 24 dimensions of the environment state). Actor network has two hidden layers (256 nodes each), Critic network has three hidden layers (256, 256 and 128 nodes). Actor network has 2 outputs (corresponding to 2 actions of the agent) and Critic network has one output.

The architecture of the DDPG Agents' Actor (local and target) network model looks like the following:

```
Actor(
  (fc1): Linear(in_features=24, out_features=256, bias=True)
  (bn): BatchNorm1d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (fc2): Linear(in_features=256, out_features=2, bias=True)
)
```

The architecture of the DDPG Agents' Critic (local and target) network model looks like the following:

```
Critic(
  (fcs1): Linear(in_features=24, out_features=256, bias=True)
  (fc2): Linear(in_features=258, out_features=256, bias=True)
  (fc3): Linear(in_features=256, out_features=128, bias=True)
  (fc4): Linear(in_features=128, out_features=1, bias=True)
)
```

Adam optimizer is used as the optimizer.

Replay buffer size was set to 1,000,000 to be sufficiently large.

Gradient clipping was used when training the critic network. The corresponding snippet of code was as follows:

```
self.critic_optimizer.zero_grad()
critic_loss.backward()
torch.nn.utils.clip_grad_norm(self.critic_local.parameters(), 1)
self.critic_optimizer.step()
```

Ornstein-Uhlenbeck noise process was used to add noise to the action space.

In this project the balance between exploration and exploitation is critical.

By decaying the amount of noise, exploration has been encouraged at the beginning, but, after a while, exploitation has been encouraged.

The agent was trained in a Jupyter notebook.

Environment was solved in **1729** episodes with an average score of **0.51**. Figure 2 and Figure 3 show how agent's score evolves throughout the training. The blue line shows the maximum score over both agents, for each episode, and the orange line shows the average score (after taking the maximum over both agents) over the next 100 episodes.

It can be observed that the agent is able to receive an average score (over 100 episodes) of at least +0.5. The maximum score over all episodes has been reached to **2.70**.

A video of trained agents playing a game can be found in the repository.

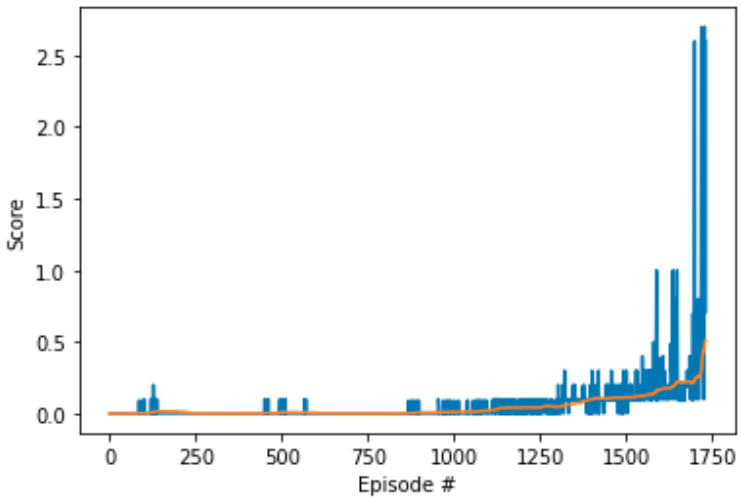

Figure 2. Plotted Scores

```
Episode 1000    Average Score: 0.01     Episode Score: 0.00     Max Score (Window): 0.10        Max Score (Overall):
0.20    Agents:0.00     -0.01
Episode 1729    Average Score: 0.51     Episode Score: 2.60     Max Score (Window): 2.70        Max Score (Overall):
2.70    Agents:0.00     0.001

Environment solved in 1729 episodes!
```

Figure 4. Training Execution

Hyperparameters and Settings

The various hyperparameters and settings used throughout the implementation are given below:

```
BUFFER_SIZE = int(1e6)      # replay buffer size
BATCH_SIZE = 128            # minibatch size
GAMMA = 0.99                # discount factor
TAU = 1e-3                  # for soft update of target parameters
LR_ACTOR = 1e-4             # learning rate of the actor
LR_CRITIC = 1e-3            # learning rate of the critic
WEIGHT_DECAY = 0            # L2 weight decay
NOISE_SCALE = 2.0           # initial noise scale for the noise added to the actions
NOISE_DECAY = 0.999         # decay for noise scale

n_episodes =  10000         # maximum number of training episodes
max_t = 1000                # maximum number of timesteps per episode

state_size = 24             # dimension of each state
action_size = 2             # dimension of each action
seed = 2                    # random seed

fc_units = 256              # number of nodes in hidden layers of Actor network

fc1_units = 256             # number of nodes in first hidden layer of Critic network
fc2_units = 256             # number of nodes in second hidden layer of Critic network
fc3_units = 128             # number of nodes in third hidden layer of Critic network

mu = 0                      # mu parameter of Ornstein-Uhlenbeck noise process
theta = 0.15                # theta parameter of Ornstein-Uhlenbeck noise process
sigma = 0.2                 # sigma parameter of Ornstein-Uhlenbeck noise process
```

Future Improvements

Although they are not tried in the implementation, hyperparameters can be further adjusted to improve the agent's performance and stability. On the other hand, by keeping the same architecture, other algorithms which have also demonstrated good performance with continuous control tasks (e.g. Proximal Policy Optimization (PPO), Distributed Distributional Deterministic Policy Gradients (D4PG)) can be implemented instead of DDPG.

As indicated in the "Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments" paper, the performance can be further improved by training agents with an ensemble of policies, an approach which is generally applicable to any multi-agent algorithm.