Andrea Pena
Udacity DRLND

Project 3: Collaboration and Competition

Description of the Project

Note: To resolve this project (P3: Collaboration and Competition), I used a similar architecture that I used in the Multi-Agent Deep Deterministic Policy Gradients (MADDPG.) coding exercise in this lesson.

Goal of the 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). 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.

Project requirements: See 'requirements.txt'.

Project Details

As in DDPG we use Actor-Critic networks, but instead of training each agent to learn from its own action, we incorporate actions taken *by all agents*. Why do this? The environment state depends on the actions taken by all agents so if we train an agent using just its own action, the policy network does not get enough information to come up with a good policy and most likely will take longer to find any good policy. In summary, MADDPG improves upon DDPG in that we now share the actions taken by all agents to train each agent.

Actor-Critic Methods

As this problem is highly unstable and tends to fall into local maxima, this implementation uses actor-critic methods to mitigate the issues. Actor model has low bias and it also helps to exit local maxima with its high variance nature whereas critic model helps the models to learn faster with its low variance nature although it has high bias.

Actor model input layer only focuses on the states of its corresponding agent. On the other hand, critic model input layer observes the states of both agents. The nodes are 24 and 48 respectively. Actor model has two fully connected hidden layers with 512 and 256 nodes as well as 2 nodes for the output layer for the actions. Critic model has three fully connected hidden layers with 516, 256 and 128 nodes. The 516 nodes come from 512 nodes of the output of input layer joined together with 2 nodes each of the actions from two agents. And the output layer of critic model is one node. Both actor and critic models utilize batch normalization and Xavier uniform weight initialization.

Agent

To reduce correlations, this implementation uses replay buffer shared by both agents. It records states and actions taken by both agents at every time step. This implementation also injects Ornstein-Uhlenbeck noise into the actions to encourage exploration around the mean value. This implementation uses Deep Deterministic Policy Gradient (DDPG) algorithm to train multiple agents with decentralized-actor and centralized-critic method. DDPG features like continuous action-space is useful to significantly reduce the number of nodes required by the neural networks. Other DDPG features like soft update is useful to perform a distinguishable targeted learning direction. Two types of clippings are also implemented. Gradient clipping is useful to avoid vanishing and exploding gradients whereas action clipping is useful to keep the actions within -1 and 1 due to noise injection.

Decentralized-actor means each actor learns from its own agent states and actions whereas centralized-critic means each critic learns from all agents states and actions. There is implementation that uses one neural networks model for all actors, but this implementation uses one neural network for each of the actors and trained separately.

Learning Algorithm

The algorithm used is Multi-Agent Deep Deterministic Policy Gradients (MADDPG).

Agent Hyper Parameters

```
BUFFER_SIZE = int(1e6)  # replay buffer size
BATCH_SIZE = 128        # minibatch size
LR_ACTOR = 1e-3         # learning rate of the actor
LR_CRITIC = 1e-3        # learning rate of the critic
WEIGHT_DECAY = 0        # L2 weight decay
LEARN_EVERY = 10        # learning timestep interval
LEARN_NUM = 5           # number of learning passes
GAMMA = 0.99            # discount factor
TAU = 8e-3              # for soft update of target parameters
OU_SIGMA = 0.2          # Ornstein-Uhlenbeck noise parameter, volatility
OU_THETA = 0.15         # Ornstein-Uhlenbeck noise parameter, speed of mean
reversion
EPS_START = 5.0         # initial value for epsilon in noise decay process in
Agent.act()
EPS_EP_END = 300        # episode to end the noise decay process
EPS_FINAL = 0           # final value for epsilon after decay
```

The model hyper-parameters used were as follows (n_episodes=2500, max_t=1000).

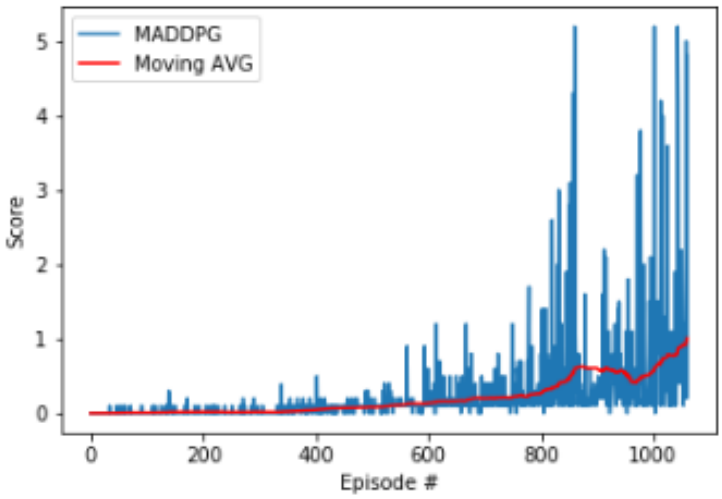
Neural Network

The neural network is implemented in the file model.py where the Actor consists of three fully connected layers. The input to fc1 is state_size, while the output of fc3 is action_size. There are 256 and 128 hidden units in fc1 and fc2, respectively, and ReLU activation is applied to fc1 and fc2, while tanh is applied to fc3**.** Also, the Critic consists of three fully connected (Linear) layers. There are also 256 and 128 hidden units in fcs1 and fc2, respectively, and ReLU activation is applied to fcs1 and fc2. In contrast, nothing is applied fc3**.**

Plot of Rewards

```
Episode 0730    Moving Average: 0.210
Episode 0740    Moving Average: 0.210
Episode 0750    Moving Average: 0.221
Episode 0760    Moving Average: 0.232
Episode 0770    Moving Average: 0.232
Episode 0780    Moving Average: 0.250
Episode 0790    Moving Average: 0.258
Episode 0800    Moving Average: 0.263
Episode 0810    Moving Average: 0.316
Episode 0820    Moving Average: 0.321
Episode 0830    Moving Average: 0.359
Episode 0840    Moving Average: 0.420
Episode 0850    Moving Average: 0.449

Environment solved in 754 episodes!
Moving Average: 0.515 (over past 100 episodes)
```



Future Work

I would recommend for future improvement a better hyperparameter implementation and try different models because I had to run this model many times in order to resolve the environment successfully. Also, add prioritized experience replay instead of the random selection of tuples. In that way, the prioritized will select experiences based on a priority value which is correlated with the magnitude error. In that manner, it will improve the learning by increasing the probability of the experience vectors that are sampled. Another good idea could be implemented batch normalization to exploding gradient issue when the computation on large input value are running and model parameters can inhibit learning. Batch normalization addresses this problem by scaling the features to be within the same range throughout the model and across different environments and units. In additional to normalizing each dimension to have unit mean and variance, the range of values is often much smaller, typically between 0 and 1.