Collaboration and Competition Project Report

Description of implementation:

The objective for this project was to find a solution to the Unity "Tennis" environment. As a rough simulation of actual tennis, the environment includes two players (referred to hereafter as the "agents"). Each of these agents controls a racket to hit the ball. The agents each receive a reward of +.1 for hitting the ball over the net, but get a negative reward (-.01) if the ball touches the ground or falls out of bounds. Since both agents benefit by keeping the ball in play as long as possible, this is actually a collaborative effort rather than a competition. The observation space consists of 8 variables corresponding to the position and velocity of the ball and racket. Two continuous actions are available, corresponding to movement toward (or away from) the net, and jumping.

For this environment, continuous control is required, so Adapted Deep Deterministic Policy Gradients (DDPG) was chosen as the basic algorithm to solve the problem. DDPG is a type of actor-critic algorithm, which was previously used with great success to solve other continuous control environments. The actor network is trained to seek out the optimal policy using a policy gradient approach. The actor is aided by the critic, which in this case tries to find the optimal Q function. Both the actor and critic each have their own neural network, which are implemented using pytorch.

To solve the tennis environment, two agents were created, so that each racket could be controlled by its own agent. In this case, creating an agent actually meant making two "instantiations" of the agent class. The trick to getting the two agents to work in the tennis environment was to notice that the environment expects to receive as input the actions from two agents simultaneously. The actions for each individual agent (a list of two elements) could therefore be combined into a larger list (containing the actions for both agents) and then sent to the environment. The environment then outputs a list of observations. But this list actually contains two smaller lists (24 elements each), which are the observations for each agent. Separately, the observations for each agent could be added to the memory (the Replay Buffer) and later used in the "learn" method (contained in the agent class) to train the actor and critic networks.

The code for this project was adapted from the DDPG code that was written to operate on OpenAI Gym's Pendulum environment. The code includes some techniques that were recently developed in the field of deep reinforcement learning. One of these techniques is known as "Experience Replay", in which data returned from the environment (experiences) is stored in a "replay buffer". A batch of these experiences is then chosen randomly from the replay buffer to train the network. This technique insures that the experiences are uncorrelated (correlated data would tend to destabilize training). Another technique used here is "Soft Update for Target Networks". This technique allows the target networks (actor and critic targets) to be updated gradually, rather than making sudden all-at-once updates. This technique also helps to stabilize learning. Exploration is facilitated by adding noise to the actions.

Hyperparameters for Continuous Control Notebook:

n_episodes = 2500	(maximum number of episodes)
print_every = 100	(how often to print episode number and average score)

Hyperparameters for ddpg_agent:

BUFFER_SIZE = int(1e5)	(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	(actor learning rate)
LR_CRITIC = 1e-3	(critic learning rate)
WEIGHT_DECAY = 0	(L2 weight decay)

Hyperparameters for model (pytorch network):

Number of nodes in first layer of actor network set to 128

Number of nodes in second layer of actor network set to 128

Number of nodes in first layer of critic network set to 128

Number of nodes in second layer of critic network set to 128

Suggestions for Improvement:

The algorithm did solve the tennis environment, achieving an average score of .508 after 6402 episodes in the final test run. However, 6402 episodes entails a rather long training time, so this seems like something that could be improved upon. One approach would be to change the number of nodes in the actor and critic networks. Some experimentation was done in the area. Initially the actor and critic networks both had two hidden layers, with 400 nodes in the first hidden layer and 300 in the second. This was changed to 128 nodes for both hidden layers for each network. The algorithm did show improvement with this change, and further optimization could lead to even better results. Another approach that might improve results is to vary the noise level during training. Noise is added to the actions in order to facilitate exploration. By having a higher level of noise at the start of training, the algorithm would tend to do more exploration, which could possibly lead to a better solution, or perhaps an equivalent solution faster. Near the end of training, the noise level could be reduced, since exploration figures to be less fruitful at this point.