Udacity Deep Reinforcement Learning Nanodegree

Project 2: Continuous Control

Goal:

In this project, I build a reinforcement learning (RL) agent that controls a robotic arm within Unity's "Reacher"(https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Learning-Environment-Examples.md#reacher) environment. The goal is to get 20 different robotic arms to maintain contact with the green spheres. A reward of +0.1 is provided for each timestep that the agent's hand is in the goal location. Thus, the goal of your agent is to maintain its position at the target location for as many time steps as possible. To solve the problem, our agent must achieve a score of +10 averaged across all 20 agents for 50 consecutive episodes.

Summary of Environment:

- Set-up: Double-jointed arm which can move to target locations.
- Goal: Each agent must move its hand to the goal location and keep it there.
- Agents: The environment contains 20 agents linked to a single Brain.
- Agent Reward Function (independent):
    - +0.1 for each timestep agent's hand is in goal location.
- Brains: One Brain with the following observation/action space.
    - Vector Observation space: 33 variables corresponding to position, rotation, velocity, and angular velocities of the two arm Rigid bodies.
    - Vector Action space: (Continuous) Each action is a vector with four numbers, corresponding to torque applicable to two joints. Every entry in the action vector should be a number between -1 and 1.
    - Visual Observations: None.
- Reset Parameters: Two, corresponding to goal size, and goal movement speed.

- Benchmark Mean Reward: 10

Approach:

Here are the high-level steps taken in building an agent that solves this environment.

1. Evaluate the state and action space.
2. Establish performance baseline using a random action policy.
3. Select an appropriate algorithm and begin implementing it.
4. Run experiments, make revisions, and retrain the agent until the performance threshold is reached.

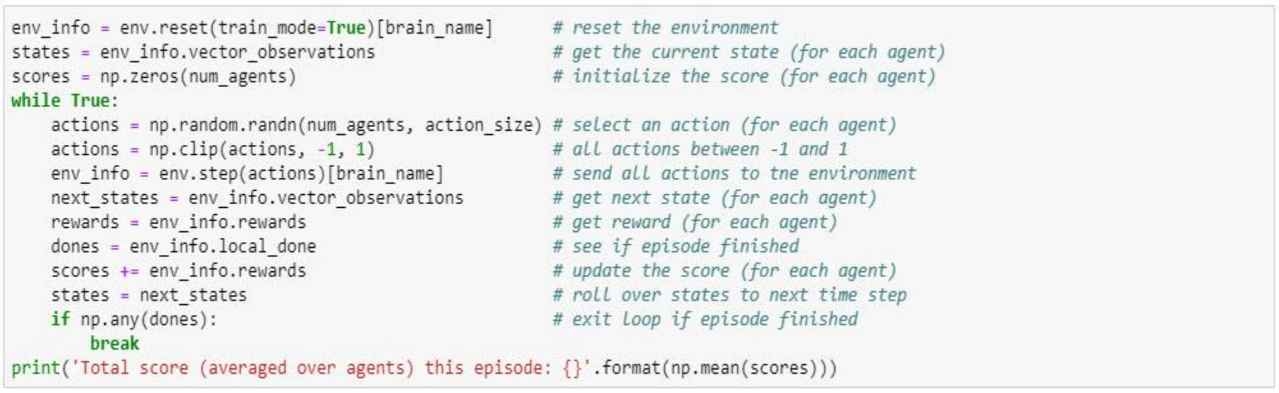
Evaluate state and action space:

The state space space has 33 dimensions corresponding to the position, rotation, velocity, and angular velocities of the robotic arm. There are two sections of the arm — analogous to those connecting the shoulder and elbow and the elbow to the wrist (i.e., the forearm) on a human body.

Each action is a vector with four numbers, corresponding to the torque applied to the two joints (shoulder and elbow). Every element in the action vector must be a number between -1 and 1, making the action space continuous.

Establish Baseline:

Before building an agent that learns, I started by testing an agent that selects actions (uniformly) at random at each time step.



```
env_info = env.reset(train_mode=True)[brain_name]      # reset the environment
states = env_info.vector_observations                  # get the current state (for each agent)
scores = np.zeros(num_agents)                          # initialize the score (for each agent)
while True:
    actions = np.random.randn(num_agents, action_size) # select an action (for each agent)
    actions = np.clip(actions, -1, 1)                  # all actions between -1 and 1
    env_info = env.step(actions)[brain_name]           # send all actions to tne 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                        # see if episode finished
    scores += env_info.rewards                         # update the score (for each agent)
    states = next_states                               # roll over states to next time step
    if np.any(dones):                                  # exit loop if episode finished
        break
print('Total score (averaged over agents) this episode: {}'.format(np.mean(scores)))
```

Running this agent a few times resulted in scores from 0.03 to 0.09. Obviously, if the agent needs to achieve an average score of +10 over 500 consecutive episodes, then choosing actions at random won't work.

Implement Learning Algorithm:

Policy-based vs Value-based Methods

There are two key differences in the Reacher environment compared to the previous 'Navigation' project:

1. **Multiple agents** — The version of the environment I'm tackling in this project has 20 different agents, whereas the Navigation project had only a single agent. To keep things simple, I decided to use a single brain to control all 20 agents, rather than training 20 individual brains. Training multiple brains seemed unnecessary since all the agents are essentially performing the same task under the same conditions. Also, training 20 brains would take a long time!
2. **Continuous action space** — The action space is now *continuous*, which allows each agent to execute more complex and precise movements. Essentially, there's an unlimited range of possible action values to control the robotic arm, whereas the agent in the Navigation project was limited to four *discrete* actions: left, right, forward, backward.

Given the additional complexity of this environment, the **value-based method** we used for the last project is not suitable — i.e., the Deep Q-Network (DQN) algorithm. Most importantly, we need an algorithm that allows the robotic arm to utilize its full range of movement. For this, we'll need to explore a different class of algorithms called **policy-based methods**.

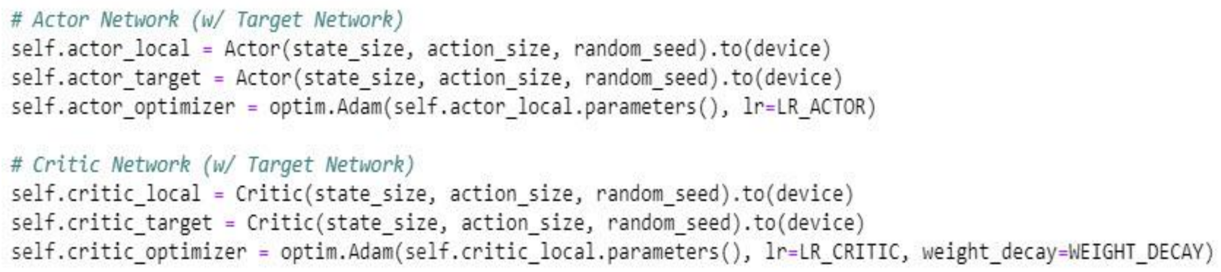
Deep Deterministic Policy Gradient (DDPG)

I used this vanilla, single-agent DDPG as a template. I further experimented with the DDPG algorithm based on other concepts covered in Udacity's classroom and lessons. My understanding and implementation of this algorithm (including various customizations) are discussed below.

Actor-Critic Method

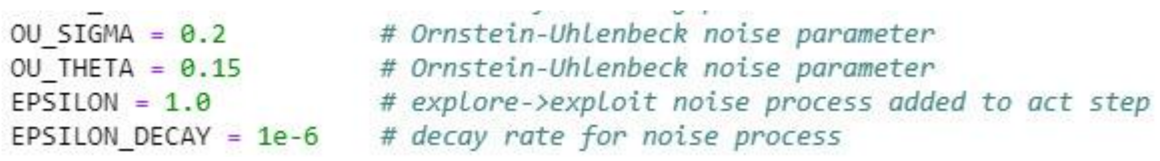
Actor-critic methods leverage the strengths of both policy-based and value-based methods.

Using a policy-based approach, the agent (actor) learns how to act by directly estimating the optimal policy and maximizing reward through gradient ascent. Meanwhile, employing a value-based approach, the agent (critic) learns how to estimate the value (i.e., the future cumulative reward) of different state-action pairs. Actor-critic methods combine these two approaches to accelerate the learning process. Actor-critic agents are also more stable than value-based agents, while requiring fewer training samples than policy-based agents.



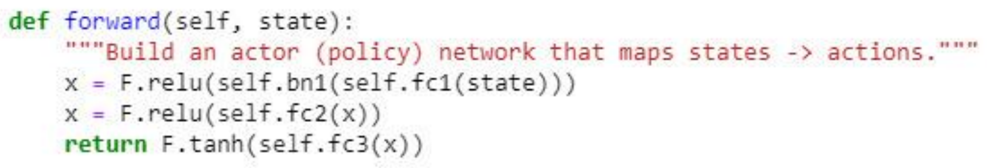
```
# Actor Network (w/ Target Network)
self.actor_local = Actor(state_size, action_size, random_seed).to(device)
self.actor_target = Actor(state_size, action_size, random_seed).to(device)
self.actor_optimizer = optim.Adam(self.actor_local.parameters(), lr=LR_ACTOR)

# Critic Network (w/ Target Network)
self.critic_local = Critic(state_size, action_size, random_seed).to(device)
self.critic_target = Critic(state_size, action_size, random_seed).to(device)
self.critic_optimizer = optim.Adam(self.critic_local.parameters(), lr=LR_CRITIC, weight_decay=WEIGHT_DECAY)
```

Noise Parameters: The final noise parameters were set as follows

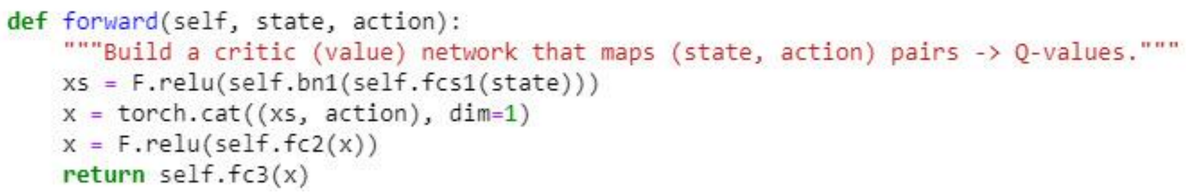


```
OU_SIGMA = 0.2          # Ornstein-Uhlenbeck noise parameter
OU_THETA = 0.15         # Ornstein-Uhlenbeck noise parameter
EPSILON = 1.0           # explore->exploit noise process added to act step
EPSILON_DECAY = 1e-6    # decay rate for noise process
```

Batch Normalization: You can find batch normalization implemented in model.py for actor and critic.



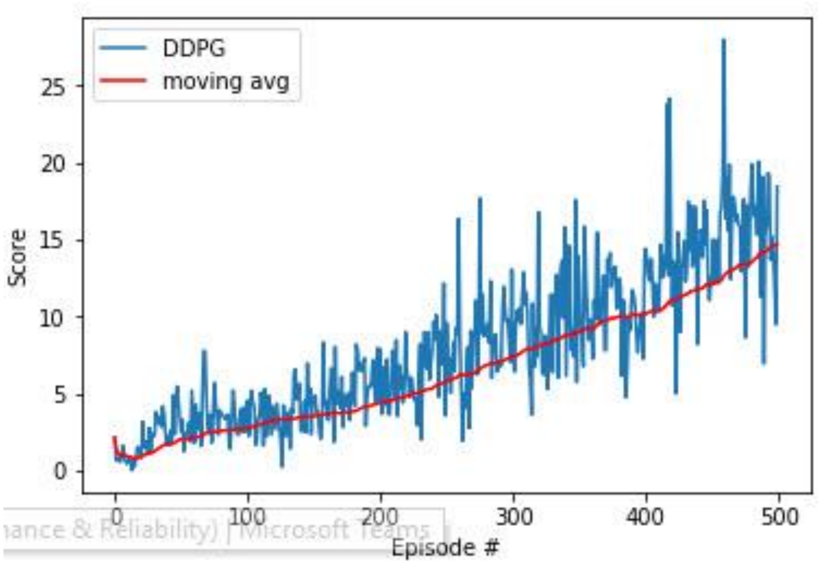
```
def forward(self, state):
    """Build an actor (policy) network that maps states -> actions."""
    x = F.relu(self.bn1(self.fc1(state)))
    x = F.relu(self.fc2(x))
    return F.tanh(self.fc3(x))
```

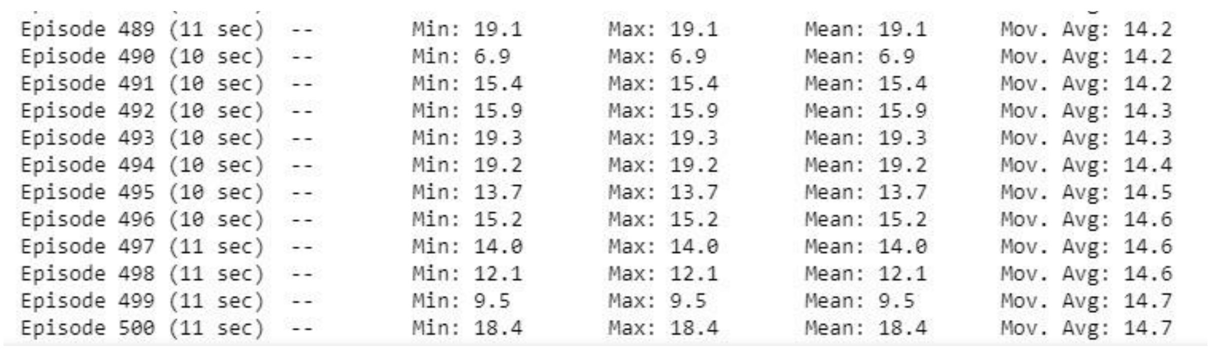


```
def forward(self, state, action):
    """Build a critic (value) network that maps (state, action) pairs -> Q-values."""
    xs = F.relu(self.bn1(self.fcs1(state)))
    x = torch.cat((xs, action), dim=1)
    x = F.relu(self.fc2(x))
    return self.fc3(x)
```

Results: Once all the various components of the algorithm were in place, my agent was able to solve the 20 agent Reacher environment. Again, the performance goal is an average reward of at least +30 over 100 episodes, and over all 20 agents.

The graph below shows the results.





```
Episode 489 (11 sec)  --          Min: 19.1        Max: 19.1        Mean: 19.1        Mov. Avg: 14.2
Episode 490 (10 sec)  --          Min: 6.9         Max: 6.9         Mean: 6.9         Mov. Avg: 14.2
Episode 491 (10 sec)  --          Min: 15.4        Max: 15.4        Mean: 15.4        Mov. Avg: 14.2
Episode 492 (10 sec)  --          Min: 15.9        Max: 15.9        Mean: 15.9        Mov. Avg: 14.3
Episode 493 (10 sec)  --          Min: 19.3        Max: 19.3        Mean: 19.3        Mov. Avg: 14.3
Episode 494 (10 sec)  --          Min: 19.2        Max: 19.2        Mean: 19.2        Mov. Avg: 14.4
Episode 495 (10 sec)  --          Min: 13.7        Max: 13.7        Mean: 13.7        Mov. Avg: 14.5
Episode 496 (10 sec)  --          Min: 15.2        Max: 15.2        Mean: 15.2        Mov. Avg: 14.6
Episode 497 (11 sec)  --          Min: 14.0        Max: 14.0        Mean: 14.0        Mov. Avg: 14.6
Episode 498 (11 sec)  --          Min: 12.1        Max: 12.1        Mean: 12.1        Mov. Avg: 14.6
Episode 499 (11 sec)  --          Min: 9.5         Max: 9.5         Mean: 9.5         Mov. Avg: 14.7
Episode 500 (11 sec)  --          Min: 18.4        Max: 18.4        Mean: 18.4        Mov. Avg: 14.7
```

Improvements:

1. Tuning the DDPG algorithm required a lot of trial and error. Perhaps another algorithm such as Trust Region Policy Optimization would be more robust.
2. Rather than selecting experience tuples randomly, prioritized replay selects experiences based on a priority value that is correlated with the magnitude of error. This can improve learning by increasing the probability that rare and important experience vectors are sampled.