Project: Continuous Control (Part of Udacity Deep Reinforcement Learning Nanodegree)

by Michael Strobl

Goal of this project:

In this environment, a double-jointed arm can move to target locations. A reward of +0.1 is provided for each step 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.

The observation space consists of 33 variables corresponding to position, rotation, velocity, and angular velocities of the arm. 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.

We use the version 1 with a single agent to get an **average score of +30 over 100** consecutive episodes.

Used Neural Networks:

Actor Network:

```
class Actor(nn.Module):
    """Actor (Policy) Model."""

    def __init__(self, state_size, action_size, seed, fc1_units=128,
                 fc2_units=128):
        """Initialize parameters and build model.
        :param state_size: int. Dimension of each state
        :param action_size: int. Dimension of each action
        :param seed: int. Random seed
        :param fc1_units: int. Number of nodes in first hidden layer
        :param fc2_units: int. Number of nodes in second hidden layer
        """
        super(Actor, self).__init__()
        self.seed = torch.manual_seed(seed)
        # source: The low-dimensional networks had 2 hidden layers
        self.fc1 = nn.Linear(state_size, fc1_units)
        # applying a Batch Normalization on the first layer output
        self.bn1 = nn.BatchNorm1d(fc1_units)

        self.fc2 = nn.Linear(fc1_units, fc2_units)
        self.fc3 = nn.Linear(fc2_units, action_size)

        self.reset_parameters()
```

- Fully connected layer - input: 37 (state size), output: 128
- Fully connected layer - input: 128 (fc1_units), output 128 (fc2_units)
- Fully connected layer - input: 128 (fc1_units), output: 4 (action size)

- Batch Normalization to speed up neural network (Source: https://towardsdatascience.com/batch-normalization-in-neural-networks-1ac91516821c)

Critic Network:

```
class Critic(nn.Module):
    """Critic (Value) Model."""

    def __init__(self, state_size, action_size, seed, fcs1_units=128,
                 fc2_units=128):
        """Initialize parameters and build model.
        :param state_size: int. Dimension of each state
        :param action_size: int. Dimension of each action
        :param seed: int. Random seed
        :param fcs1_units: int. Nb of nodes in the first hiddenlayer
        :param fc2_units: int. Nb of nodes in the second hidden layer
        """
        super(Critic, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fcs1 = nn.Linear(state_size, fcs1_units)
        self.bn1 = nn.BatchNorm1d(fcs1_units)
        self.fc2 = nn.Linear(fcs1_units+action_size, fc2_units)
        self.fc3 = nn.Linear(fc2_units, 1)


        self.reset_parameters()
```

- Fully connected layer - input: 37 (state size), output: 128
- Fully connected layer - input: 128 (fcs1_units), output 128 (fc2_units)
- Fully connected layer - input: 128 (fc1_units), output: 4 (action size)
- Batch Normalization to speed up neural network

Used Learning Algorithm:

Deep Deterministic Policy Gradient (DDPG)

Pseudo-Code:

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

Source: https://towardsdatascience.com/deep-deterministic-policy-gradients-explained-2d94655a9b7b

In [8]:
```
def ddpg(n_episodes=1000, max_t=10000, print_every=10):
    scores_deque = deque(maxlen=print_every)
    scores = []
    for i_episode in range(1, n_episodes+1):
        env_info = env.reset(train_mode=True)[brain_name]
        agent.reset()
        state = env_info.vector_observations[0]
        score = 0

        for t in range(max_t):
            action = agent.act(state)
            env_info = env.step(action)[brain_name]
            next_state = env_info.vector_observations[0]
            reward = env_info.rewards[0]
            done = env_info.local_done[0]
            agent.step(state, action, reward, next_state, done)
            score += reward
            state = next_state
            if done:
                break

        scores_deque.append(score)
        scores.append(score)
```

Code is similar to this source:
https://github.com/udacity/deep-reinforcement-learning/blob/master/ddpg-pendulum/DDPG.ipynb

Hyperparameters:

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 = 2e-4 # learning rate of the actor
LR_CRITIC = 2e-4 # learning rate of the critic
WEIGHT_DECAY = 0 # L2 weight decay

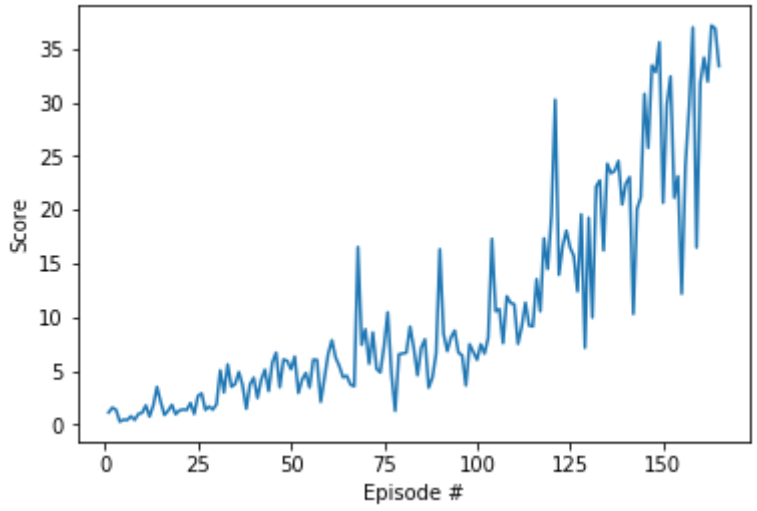
Source:
https://github.com/udacity/deep-reinforcement-learning/blob/master/ddpg-bipedal/ddpg_agent.py

Ornstein-Uhlenbeck Noise:
mu = 0. # long-running mean
theta = 0.15 # the speed of mean reversion
sigma = 0.1 # the volatility parameter

Source: https://planetmath.org/ornsteinuhlenbeckprocess

Best result:

Episode 10 Average Score: 0.84
Episode 20 Average Score: 1.61
Episode 30 Average Score: 1.75
Episode 40 Average Score: 3.90
Episode 50 Average Score: 4.78
Episode 60 Average Score: 4.67
Episode 70 Average Score: 6.85
Episode 80 Average Score: 6.11
Episode 90 Average Score: 7.33
Episode 100 Average Score: 6.91
Episode 110 Average Score: 10.27
Episode 120 Average Score: 12.17
Episode 130 Average Score: 16.93
Episode 140 Average Score: 20.98
Episode 150 Average Score: 25.36
Episode 160 Average Score: 25.76
Episode 165 Average Score: 31.25
Environment solved in 165 episodes! Average Score: 31.25



Future:

- Implementing an A2C or PPO Algorithm for the 20 agents environment
- Try more different neural networks and batch norms
- Change Tau and Gamma to see if the results get better
- Try different batch and buffer sizes

Sources:

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