Tutorial 6: DDPG

Gautham Govind A, EE19B022

```
import numpy as np
import gym
from collections import deque
import random

# Ornstein-Ulhenbeck Process
# Taken from
#https://github.com/vitchyr/rlkit/blob/master/rlkit/exploration_strate
gies/ou_strategy.py
class OUNoise(object):
    def __init__(self, action_space, mu=0.0, theta=0.15,
max_sigma=0.3, min_sigma=0.3, decay_period=100000):
        self.mu           = mu
        self.theta        = theta
        self.sigma        = max_sigma
        self.max_sigma    = max_sigma
        self.min_sigma    = min_sigma
        self.decay_period = decay_period
        self.action_dim   = action_space.shape[0]
        self.low          = action_space.low
        self.high         = action_space.high
        self.reset()

    def reset(self):
        self.state = np.ones(self.action_dim) * self.mu

    def evolve_state(self):
        x  = self.state
        dx = self.theta * (self.mu - x) + self.sigma *
np.random.randn(self.action_dim)
        self.state = x + dx
        return self.state

    def get_action(self, action, t=0):
        ou_state = self.evolve_state()
        self.sigma = self.max_sigma - (self.max_sigma -
self.min_sigma) * min(1.0, t / self.decay_period)
        return np.clip(action + ou_state, self.low, self.high)


# https://github.com/openai/gym/blob/master/gym/core.py
class NormalizedEnv(gym.ActionWrapper):
    """ Wrap action """
```

```
    def action(self, action):
        act_k = (self.action_space.high - self.action_space.low)/ 2.
        act_b = (self.action_space.high + self.action_space.low)/ 2.
        return act_k * action + act_b



class Memory:
    def __init__(self, max_size):
        self.max_size = max_size
        self.buffer = deque(maxlen=max_size)

    def push(self, state, action, reward, next_state, done):
        experience = (state, action, np.array([reward]), next_state,
done)
        self.buffer.append(experience)

    def sample(self, batch_size):
        state_batch = []
        action_batch = []
        reward_batch = []
        next_state_batch = []
        done_batch = []

        batch = random.sample(self.buffer, batch_size)

        for experience in batch:
            state, action, reward, next_state, done = experience
            state_batch.append(state)
            action_batch.append(action)
            reward_batch.append(reward)
            next_state_batch.append(next_state)
            done_batch.append(done)

        return state_batch, action_batch, reward_batch,
next_state_batch, done_batch

    def __len__(self):
        return len(self.buffer)
```

DDPG uses four neural networks: a Q network, a deterministic policy network, a target Q network, and a target policy network.

$$\text{Parameters:}$$

$$\theta^{Q} : \text{Q network}$$

$$\theta^{\mu} : \text{Deterministic policy function}$$

$$\theta^{Q'} : \text{target Q network}$$

$$\theta^{\mu'} : \text{target policy network}$$

The Q network and policy network is very much like simple Advantage Actor-Critic, but in DDPG, the Actor directly maps states to actions instead of outputting the probability distribution across a discrete action space.

The target networks are time-delayed copies of their original networks that slowly track the learned networks. Using these target value networks greatly improve stability in learning.

Let's create these networks.

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class Critic(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(Critic, self).__init__()
        self.linear1 = nn.Linear(input_size, hidden_size)
        self.linear2 = nn.Linear(hidden_size, hidden_size)
        self.linear3 = nn.Linear(hidden_size, output_size)

    def forward(self, state, action):
        """
        Params state and actions are torch tensors
        """
        x = torch.cat([state, action], 1)
        x = F.relu(self.linear1(x))
        x = F.relu(self.linear2(x))
        x = self.linear3(x)

        return x

class Actor(nn.Module):
```

```
    def __init__(self, input_size, hidden_size, output_size, learning_rate = 3e-4):
        super(Actor, self).__init__()
        self.linear1 = nn.Linear(input_size, hidden_size)
        self.linear2 = nn.Linear(hidden_size, hidden_size)
        self.linear3 = nn.Linear(hidden_size, output_size)

    def forward(self, state):
        """
        Param state is a torch tensor
        """
        x = F.relu(self.linear1(state))
        x = F.relu(self.linear2(x))
        x = torch.tanh(self.linear3(x))

        return x
```

Now, let's create the DDPG agent. The agent class has two main functions: "get_action" and "update":

- **get_action()**: This function runs a forward pass through the actor network to select a determinisitic action. In the DDPG paper, the authors use Ornstein-Uhlenbeck Process to add noise to the action output (Uhlenbeck & Ornstein, 1930), thereby resulting in exploration in the environment. Class OUNoise (in cell 1) implements this.

$$\mu'(s_t) = \mu(s_t|\theta_t^{\mu}) + \mathcal{N}$$

- **update()**: This function is used for updating the actor and critic networks, and forms the core of the DDPG algorithm. The replay buffer is first sampled to get a batch of experiences of the form **<states, actions, rewards, next_states>**.

The value network is updated using the Bellman equation, similar to Q-learning. However, in DDPG, the next-state Q values are calculated with the target value network and target policy network. Then, we minimize the mean-squared loss between the target Q value and the predicted Q value:

$$y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$$

$$Loss = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$$

For the policy function, our objective is to maximize the expected return. To calculate the policy gradient, we take the derivative of the objective function with respect to the policy parameter. For this, we use the chain rule.

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

We make a copy of the target network parameters and have them slowly track those of the learned networks via "soft updates," as illustrated below:

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

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

$$\text{where} \quad \tau \ll 1$$

```
import torch
import torch.optim as optim
import torch.nn as nn

class DDPGagent:
    def __init__(self, env, hidden_size=256, actor_learning_rate=1e-4, 
critic_learning_rate=1e-3, gamma=0.99, tau=1e-2, 
max_memory_size=50000):
        # Params
        self.num_states = env.observation_space.shape[0]
        self.num_actions = env.action_space.shape[0]
        self.gamma = gamma
        self.tau = tau

        # Networks
        self.actor = Actor(self.num_states, hidden_size, 
self.num_actions)
        self.actor_target = Actor(self.num_states, hidden_size, 
self.num_actions)
        self.critic = Critic(self.num_states + self.num_actions, 
hidden_size, self.num_actions)
        self.critic_target = Critic(self.num_states + 
self.num_actions, hidden_size, self.num_actions)
```

```
        for target_param, param in zip(self.actor_target.parameters(),
self.actor.parameters()):
            target_param.data.copy_(param.data)

        for target_param, param in
zip(self.critic_target.parameters(), self.critic.parameters()):
            target_param.data.copy_(param.data)

        # Training
        self.memory = Memory(max_memory_size)
        self.critic_criterion  = nn.MSELoss()
        self.actor_optimizer  = optim.Adam(self.actor.parameters(),
lr=actor_learning_rate)
        self.critic_optimizer = optim.Adam(self.critic.parameters(),
lr=critic_learning_rate)

    def get_action(self, state):
        state = torch.FloatTensor(state).unsqueeze(0)
        action = self.actor.forward(state)
        action = action.detach().numpy()[0,0]
        return action

    def update(self, batch_size):
        states, actions, rewards, next_states, _ =
self.memory.sample(batch_size)
        states = torch.FloatTensor(states)
        actions = torch.FloatTensor(actions)
        rewards = torch.FloatTensor(rewards)
        next_states = torch.FloatTensor(next_states)

        # Implement critic loss and update critic

        # The bootstrapped target Q value is computed using the target
actor and target critic
        # Forward passes are made on the target networks to get the
next state values
        bootstrap_Qtargets = rewards +
(self.gamma)*(self.critic_target(next_states,
self.actor_target(next_states)))

        # Current Q value is computed using the critic network
        current_Q = self.critic(states, actions)

        # Critic loss is computed
        critic_loss = self.critic_criterion(current_Q,
bootstrap_Qtargets)

        # Backward pass to compute the gradients
```

```
        self.critic_optimizer.zero_grad()
        critic_loss.backward()
        self.critic_optimizer.step()


        # Implement actor loss and update actor

        # Note that since our objective is to maximize the performance 
measure, we should include a negative sign
        # This is because, by default, pytorch minimizes whatever loss 
function we input
        actor_loss = -self.critic(states, self.actor(states)).mean()

        # Backward pass to compute the gradients
        self.actor_optimizer.zero_grad()
        actor_loss.backward()
        self.actor_optimizer.step()


        # update target networks

        # Target network parameters are updated
        for param, target_param in zip(self.critic.parameters(), 
self.critic_target.parameters()):
            target_param.data.copy_(self.tau * param.data + (1 - 
self.tau) * target_param.data)

        for param, target_param in zip(self.actor.parameters(), 
self.actor_target.parameters()):
            target_param.data.copy_(self.tau * param.data + (1 - 
self.tau) * target_param.data)
```

Putting it all together: DDPG in action.

The main function below runs 100 episodes of DDPG on the "Pendulum-v0" environment of OpenAI gym. This is the inverted pendulum swingup problem, a classic problem in the control literature. In this version of the problem, the pendulum starts in a random position, and the goal is to swing it up so it stays upright.

Each episode is for a maximum of 200 timesteps. At each step, the agent chooses an action, moves to the next state and updates its parameters according to the DDPG algorithm, repeating this process till the end of the episode.

The DDPG algorithm is as follows:

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

```
import sys
import gym
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# For more info on the Pendulum environment, check out
https://www.gymlibrary.dev/environments/classic_control/pendulum/
env = NormalizedEnv(gym.make("Pendulum-v1"))

agent = DDPGagent(env)
noise = OUNoise(env.action_space)
batch_size = 128
rewards = []
avg_rewards = []

for episode in range(100):
    state = env.reset()
    noise.reset()
    episode_reward = 0

    for step in range(200):
        action = agent.get_action(state)
        #Add noise to action
```

```
        # The get_action method of OUNoise() class is called to add OU
noise to the action
        action = noise.get_action(action)

        new_state, reward, done, _ = env.step(action)
        agent.memory.push(state, action, reward, new_state, done)

        if len(agent.memory) > batch_size:
            agent.update(batch_size)

        state = new_state
        episode_reward += reward

        if done:
            sys.stdout.write("episode: {}, reward: {}, average
_reward: {} \n".format(episode, np.round(episode_reward, decimals=2),
np.mean(rewards[-10:])))
            break

    rewards.append(episode_reward)
    avg_rewards.append(np.mean(rewards[-10:]))

plt.plot(rewards)
plt.plot(avg_rewards)
plt.plot()
plt.xlabel('Episode')
plt.ylabel('Reward')
plt.show()
```

```
/usr/local/lib/python3.9/dist-packages/gym/core.py:317:
DeprecationWarning: WARN: Initializing wrapper in old step API which
returns one bool instead of two. It is recommended to set
`new_step_api=True` to use new step API. This will be the default
behaviour in future.
  deprecation(
/usr/local/lib/python3.9/dist-packages/gym/wrappers/step_api_compatibi
lity.py:39: DeprecationWarning: WARN: Initializing environment in old
step API which returns one bool instead of two. It is recommended to
set `new_step_api=True` to use new step API. This will be the default
behaviour in future.
  deprecation(
<ipython-input-3-655225a7a624>:40: UserWarning: Creating a tensor from
a list of numpy.ndarrays is extremely slow. Please consider converting
the list to a single numpy.ndarray with numpy.array() before
converting to a tensor. (Triggered internally at
../torch/csrc/utils/tensor_new.cpp:230.)
  states = torch.FloatTensor(states)
/usr/local/lib/python3.9/dist-packages/numpy/core/fromnumeric.py:3474:
RuntimeWarning: Mean of empty slice.
  return _methods._mean(a, axis=axis, dtype=dtype,
```

```
/usr/local/lib/python3.9/dist-packages/numpy/core/_methods.py:189:
RuntimeWarning: invalid value encountered in double_scalars
  ret = ret.dtype.type(ret / rcount)

episode: 0, reward: -972.23, average _reward: nan
episode: 1, reward: -1671.1, average _reward: -972.2275283338637
episode: 2, reward: -1767.73, average _reward: -1321.664418074453
episode: 3, reward: -1759.44, average _reward: -1470.3541095821172
episode: 4, reward: -1453.49, average _reward: -1542.626388424632
episode: 5, reward: -1139.18, average _reward: -1524.799234043495
episode: 6, reward: -902.02, average _reward: -1460.52903596125
episode: 7, reward: -639.96, average _reward: -1380.7422150322147
episode: 8, reward: -797.18, average _reward: -1288.144418613789
episode: 9, reward: -780.03, average _reward: -1233.5925931077427
episode: 10, reward: -505.69, average _reward: -1188.2367156732519
episode: 11, reward: -471.91, average _reward: -1141.5834597388548
episode: 12, reward: -372.6, average _reward: -1021.664653390308
episode: 13, reward: -509.49, average _reward: -882.1511977380102
episode: 14, reward: -380.23, average _reward: -757.1553850082074
episode: 15, reward: -3.66, average _reward: -649.8289828991295
episode: 16, reward: -253.7, average _reward: -536.277487070165
episode: 17, reward: -490.02, average _reward: -471.4450867210976
episode: 18, reward: -379.95, average _reward: -456.45115498345
episode: 19, reward: -244.82, average _reward: -414.728571474835
episode: 20, reward: -497.28, average _reward: -361.20699861514447
episode: 21, reward: -256.31, average _reward: -360.3650570077962
episode: 22, reward: -243.24, average _reward: -338.80519656354227
episode: 23, reward: -372.82, average _reward: -325.8697449924837
episode: 24, reward: -362.46, average _reward: -312.20364325820617
episode: 25, reward: -487.75, average _reward: -310.42708003782866
episode: 26, reward: -452.37, average _reward: -358.83625096276427
episode: 27, reward: -609.88, average _reward: -378.70330616010887
episode: 28, reward: -493.79, average _reward: -390.68943297009247
episode: 29, reward: -252.22, average _reward: -402.07348652288226
episode: 30, reward: -329.07, average _reward: -402.8134235883037
episode: 31, reward: -373.46, average _reward: -385.992897287676
episode: 32, reward: -502.95, average _reward: -397.70712589642335
episode: 33, reward: -587.22, average _reward: -423.6775580706438
episode: 34, reward: -386.92, average _reward: -445.1172536993432
episode: 35, reward: -509.22, average _reward: -447.56357035029
episode: 36, reward: -492.14, average _reward: -449.7102752935663
episode: 37, reward: -502.44, average _reward: -453.68763557846415
episode: 38, reward: -250.88, average _reward: -442.94326203466187
episode: 39, reward: -257.83, average _reward: -418.65163936336256
episode: 40, reward: -367.72, average _reward: -419.2132665617951
episode: 41, reward: -386.5, average _reward: -423.0782240878385
episode: 42, reward: -623.67, average _reward: -424.3823863061872
episode: 43, reward: -534.26, average _reward: -436.45474130867933
episode: 44, reward: -377.92, average _reward: -431.15913173150904
episode: 45, reward: -256.88, average _reward: -430.2587939119633
```

```
episode: 46, reward: -733.19, average _reward: -405.0247033412437
episode: 47, reward: -375.95, average _reward: -429.1299009379676
episode: 48, reward: -382.56, average _reward: -416.4809438828056
episode: 49, reward: -562.89, average _reward: -429.6489105514553
episode: 50, reward: -376.38, average _reward: -460.1543993266405
episode: 51, reward: -489.94, average _reward: -461.02013348376096
episode: 52, reward: -608.88, average _reward: -471.3644294663853
episode: 53, reward: -612.49, average _reward: -469.8850338864737
episode: 54, reward: -614.73, average _reward: -477.7074316894582
episode: 55, reward: -612.9, average _reward: -501.3878766434006
episode: 56, reward: -375.13, average _reward: -536.9902318632919
episode: 57, reward: -262.72, average _reward: -501.1841851449464
episode: 58, reward: -374.11, average _reward: -489.86141773755446
episode: 59, reward: -473.59, average _reward: -489.0166853741315
episode: 60, reward: -510.15, average _reward: -480.08659611935263
episode: 61, reward: -360.3, average _reward: -493.46362761588637
episode: 62, reward: -545.68, average _reward: -480.4995280767036
episode: 63, reward: -488.18, average _reward: -474.1794758476488
episode: 64, reward: -409.44, average _reward: -461.74866132596446
episode: 65, reward: -508.97, average _reward: -441.21976651345483
episode: 66, reward: -847.07, average _reward: -430.8260077358169
episode: 67, reward: -495.47, average _reward: -478.01977814448685
episode: 68, reward: -496.97, average _reward: -501.2945826490839
episode: 69, reward: -362.09, average _reward: -513.5805401124119
episode: 70, reward: -382.33, average _reward: -502.4303958434287
episode: 72, reward: -590.32, average _reward: -491.159557580354
episode: 73, reward: -382.52, average _reward: -495.62369380007215
episode: 74, reward: -496.59, average _reward: -485.0573200888572
episode: 75, reward: -500.65, average _reward: -493.77240818653337
episode: 76, reward: -500.33, average _reward: -492.9410395873364
episode: 77, reward: -500.51, average _reward: -458.26704367861805
episode: 78, reward: -404.25, average _reward: -458.77094148376455
episode: 79, reward: -688.8, average _reward: -449.49884945367495
episode: 80, reward: -615.51, average _reward: -482.1705282930714
episode: 81, reward: -376.79, average _reward: -505.48900635723146
episode: 82, reward: -490.69, average _reward: -505.6263492671049
episode: 83, reward: -611.8, average _reward: -495.6636277174037
episode: 84, reward: -492.75, average _reward: -518.5918280966205
episode: 85, reward: -483.92, average _reward: -518.2081342570236
episode: 86, reward: -523.44, average _reward: -516.5344837867676
episode: 87, reward: -641.45, average _reward: -518.8456001908764
episode: 88, reward: -449.61, average _reward: -532.9400521589421
episode: 89, reward: -379.94, average _reward: -537.4766999473732
episode: 90, reward: -121.8, average _reward: -506.58995996444025
episode: 91, reward: -604.74, average _reward: -457.2186645603917
episode: 92, reward: -494.85, average _reward: -480.01446122557616
episode: 93, reward: -598.55, average _reward: -480.4305578255836
episode: 94, reward: -625.58, average _reward: -479.1052179606592
episode: 95, reward: -382.09, average _reward: -492.3879131363116
episode: 96, reward: -378.11, average _reward: -482.2054289231843
```

```
episode: 97, reward: -375.12, average _reward: -467.6721558276158
episode: 98, reward: -382.52, average _reward: -441.0387407906718
episode: 99, reward: -257.16, average _reward: -434.32921746307056
```