```
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

1. List item
2. List item

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 (the output of the network directly the output) 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
import torch.autograd
from torch.autograd import Variable

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
```

```
/usr/local/lib/python3.10/dist-packages/ipykernel/ipkernel.py:283:
DeprecationWarning: `should_run_async` will not call `transform_cell`
automatically in the future. Please pass the result to
`transformed_cell` argument and any exception that happen during
thetransform in `preprocessing_exc_tuple` in IPython 7.17 and above.
  and should_run_async(code)
```

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 similarly as is done in Q-learning. The updated Q value is obtained by the Bellman equation. 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 updated Q value and the original 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 loss, we take the derivative of the objective function with respect to the policy parameter. Keep in mind that the actor (policy) function is differentiable, so we have to apply the chain rule.

But since we are updating the policy in an off-policy way with batches of experience, we take the mean of the sum of gradients calculated from the mini-batch:

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

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.autograd
import torch.optim as optim
import torch.nn as nn
# from model import *
# from utils import *
```

```python
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)
            target_param.requires_grad = False

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

        # 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 = Variable(torch.from_numpy(state).float().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
        self.critic_optimizer.zero_grad()
        Qvals = self.critic.forward(states, actions)
        next_actions = self.actor_target.forward(next_states)
        next_Q = self.critic_target.forward(next_states, 
next_actions.detach())
        Qprime = rewards + self.gamma * next_Q
        critic_loss = self.critic_criterion(Qvals, Qprime)
        critic_loss.backward()
        self.critic_optimizer.step()
        # Implement actor loss and update actor
        self.actor_optimizer.zero_grad()
        policy_loss = -self.critic.forward(states, 
self.actor.forward(states)).mean()
        policy_loss.backward()
        self.actor_optimizer.step()

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

        for target_param, param in 
zip(self.critic_target.parameters(), self.critic.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 50 episodes of DDPG on the "Pendulum-v1" 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 500 timesteps. At each step, the agent chooses an action, updates its parameters according to the DDPG algorithm and moves to the next state, 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

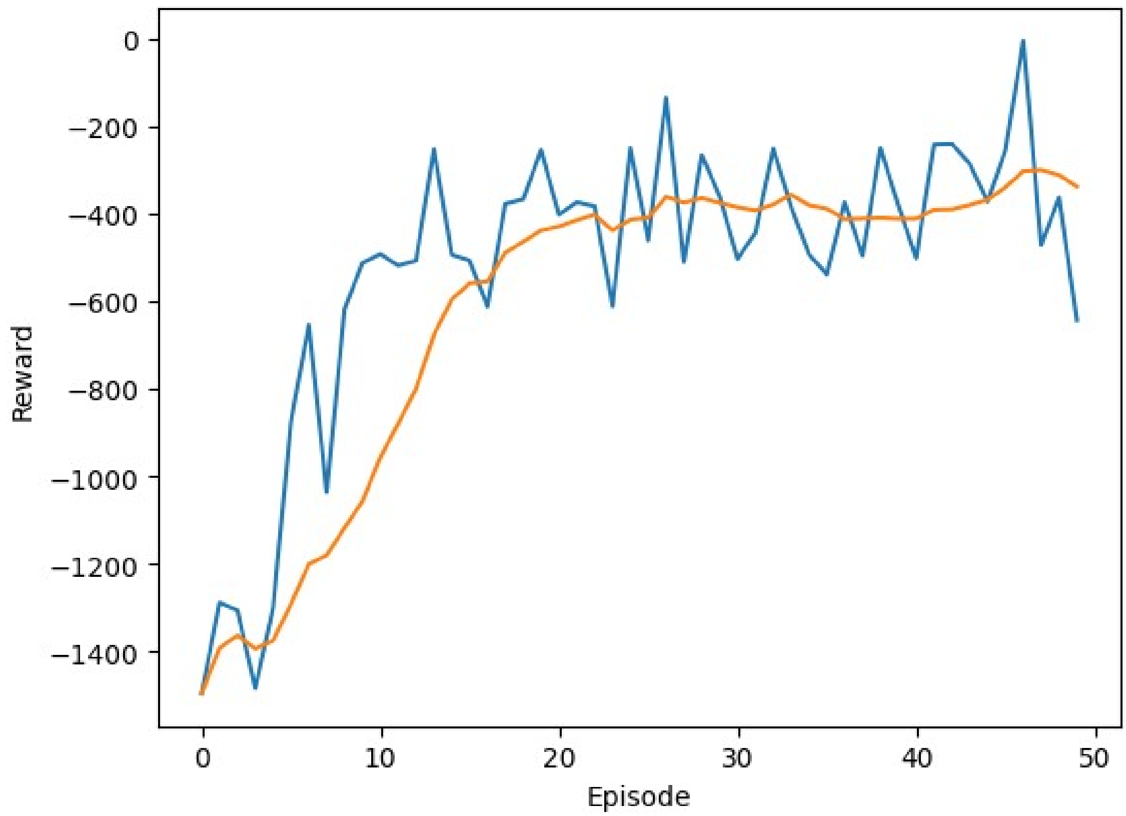
```
import sys
import gym
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

env = NormalizedEnv(gym.make("Pendulum-v1"))

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

for episode in range(50):
    state = env.reset()
    noise.reset()
    episode_reward = 0

    for step in range(500):
        action = agent.get_action(state)
        #Add noise to 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/python3.10/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/python3.10/dist-packages/gym/wrappers/step_api_compatib
ility.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(
```

```
episode: 0, reward: -1496.0, average _reward: nan
episode: 1, reward: -1288.42, average _reward: -1496.002691679898
episode: 2, reward: -1306.41, average _reward: -1392.2101768665234
episode: 3, reward: -1484.8, average _reward: -1363.6100486612788
episode: 4, reward: -1300.6, average _reward: -1393.907289083677
episode: 5, reward: -874.74, average _reward: -1375.245500277483
episode: 6, reward: -653.38, average _reward: -1291.8281511974162
episode: 7, reward: -1036.42, average _reward: -1200.621888785888
episode: 8, reward: -617.52, average _reward: -1180.0967291233142
episode: 9, reward: -512.38, average _reward: -1117.5876681459542
episode: 10, reward: -492.1, average _reward: -1057.0668120142548
```

```
episode: 11, reward: -518.26, average _reward: -956.6763988054212
episode: 12, reward: -507.05, average _reward: -879.660156320221
episode: 13, reward: -252.31, average _reward: -799.7246462318261
episode: 14, reward: -493.36, average _reward: -676.4756777078521
episode: 15, reward: -506.83, average _reward: -595.7518466844735
episode: 16, reward: -612.01, average _reward: -558.9604336692735
episode: 17, reward: -377.18, average _reward: -554.8234333269761
episode: 18, reward: -366.76, average _reward: -488.8992145564095
episode: 19, reward: -254.28, average _reward: -463.8235011419239
episode: 20, reward: -401.88, average _reward: -438.01317873262195
episode: 21, reward: -373.56, average _reward: -428.99129620536314
episode: 22, reward: -382.79, average _reward: -414.5215441615711
episode: 23, reward: -610.88, average _reward: -402.0952826445178
episode: 24, reward: -248.84, average _reward: -437.9522810048743
episode: 25, reward: -460.64, average _reward: -413.5003709298182
episode: 26, reward: -134.34, average _reward: -408.88166080003526
episode: 27, reward: -510.01, average _reward: -361.11472422085376
episode: 28, reward: -265.93, average _reward: -374.39755073981803
episode: 29, reward: -360.2, average _reward: -364.3145452201479
episode: 30, reward: -503.03, average _reward: -374.90734682183876
episode: 31, reward: -443.56, average _reward: -385.0218958734025
episode: 32, reward: -250.92, average _reward: -392.0216400101471
episode: 33, reward: -383.94, average _reward: -378.8342788999001
episode: 34, reward: -492.51, average _reward: -356.14007426616706
episode: 35, reward: -538.7, average _reward: -380.50681497120263
episode: 36, reward: -373.53, average _reward: -388.31263353216593
episode: 37, reward: -495.72, average _reward: -412.231216493857
episode: 38, reward: -249.43, average _reward: -410.8029685193019
episode: 39, reward: -380.28, average _reward: -409.1527726403632
episode: 40, reward: -501.47, average _reward: -411.16004046845944
episode: 41, reward: -241.18, average _reward: -411.00495055857607
episode: 42, reward: -239.99, average _reward: -390.7676039691165
episode: 43, reward: -285.74, average _reward: -389.67471559567764
episode: 44, reward: -372.53, average _reward: -379.85505263373705
episode: 45, reward: -254.23, average _reward: -367.857323317046
episode: 46, reward: -4.71, average _reward: -339.41097631967887
episode: 47, reward: -471.46, average _reward: -302.5290583986802
episode: 48, reward: -363.01, average _reward: -300.1029265604435
episode: 49, reward: -642.73, average _reward: -311.46133685444994
```



Your Inference

- From the output and plot, we can infer that the reward values, which started as very negative, have become less negative, indicating that the agent is learning from the environment.
- The average reward plot, smoother compared to episodic reward, shows a clear upward trend, meaning that the agent's policy is leading to better outcomes as training progresses.
- There is a significant variance in the reward from episode to episode, which suggests that the agent's experience in each episode can vary widely, but the overall trend is still towards improved performance.
- By the final episodes, the average reward seems to stabilize, which could indicate that the agent is reaching the limits of what it can learn with the current architecture and hyperparameters, or it might mean that the learning is beginning to level off.