```python
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
```

```python
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.

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):
```

```python
    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
        self.critic_optimizer.zero_grad()
        y = rewards + self.gamma * self.critic_target.forward(states,
self.actor_target.forward(states).detach())
        Q = self.critic.forward(states, actions)
        loss = self.critic_criterion(y, Q)
        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))
        policy_loss = policy_loss.mean()
        policy_loss.backward()
        self.actor_optimizer.step()

        # update target networks
```

```
        with torch.no_grad():
          for target_param, param in
zip(self.actor_target.parameters(), self.actor.parameters()):
            target_param.data.copy_(self.tau * param.data + (1.0 -
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 + (1.0 -
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
```

```python
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

        action = noise.get_action(action, step)
        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(
/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: -1402.47, average _reward: nan
episode: 1, reward: -1395.09, average _reward: -1402.4658915334019
episode: 2, reward: -1727.51, average _reward: -1398.7804329776304
episode: 3, reward: -1441.61, average _reward: -1508.3581090076798
episode: 4, reward: -1543.35, average _reward: -1491.6700328552874
episode: 5, reward: -1256.06, average _reward: -1502.005737924385
episode: 6, reward: -1557.2, average _reward: -1461.014589812629
episode: 7, reward: -1593.07, average _reward: -1474.7547231347285
episode: 8, reward: -1538.75, average _reward: -1489.5439029584786
episode: 9, reward: -1308.45, average _reward: -1495.01114188601
episode: 10, reward: -1567.27, average _reward: -1476.3549182694662
episode: 11, reward: -1410.96, average _reward: -1492.8352379433177
episode: 12, reward: -1404.76, average _reward: -1494.4217679105834
episode: 13, reward: -1475.55, average _reward: -1462.1465197359962
episode: 14, reward: -1356.35, average _reward: -1465.540560362231
episode: 15, reward: -1380.27, average _reward: -1446.8407785223942
episode: 16, reward: -1470.51, average _reward: -1459.2615009091967
episode: 17, reward: -1522.62, average _reward: -1450.5930749253298
episode: 18, reward: -1387.59, average _reward: -1443.548709983089
episode: 19, reward: -1263.18, average _reward: -1428.433049155354
episode: 20, reward: -1387.72, average _reward: -1423.906656520531
episode: 21, reward: -1305.01, average _reward: -1405.9518878826525
episode: 22, reward: -1000.29, average _reward: -1395.3567623897322
episode: 23, reward: -1159.95, average _reward: -1354.9101213649794
episode: 24, reward: -1465.45, average _reward: -1323.3506531170342
episode: 25, reward: -1155.1, average _reward: -1334.2605453256697
episode: 26, reward: -1187.53, average _reward: -1311.7439829404107
episode: 27, reward: -1362.4, average _reward: -1283.4455385137169
episode: 28, reward: -1334.57, average _reward: -1267.423073804179
episode: 29, reward: -1239.95, average _reward: -1262.121088679628
episode: 30, reward: -1237.3, average _reward: -1259.7980011625828
episode: 31, reward: -1427.09, average _reward: -1244.756040747589
episode: 32, reward: -1381.09, average _reward: -1256.964093349837
```

```
episode: 33, reward: -1387.82, average _reward: -1295.0431779729956
episode: 34, reward: -1291.11, average _reward: -1317.8302865021265
episode: 35, reward: -1188.46, average _reward: -1300.3965294941895
episode: 36, reward: -1045.18, average _reward: -1303.7319897171697
episode: 37, reward: -1189.55, average _reward: -1289.4972926913563
episode: 38, reward: -1451.08, average _reward: -1272.2121308049334
episode: 39, reward: -1213.42, average _reward: -1283.8624108792985
episode: 40, reward: -1437.29, average _reward: -1281.208642125628
episode: 41, reward: -1400.69, average _reward: -1301.2069969611844
episode: 42, reward: -1213.43, average _reward: -1298.5673840508387
episode: 43, reward: -1404.42, average _reward: -1281.8019115894078
episode: 44, reward: -1419.48, average _reward: -1283.4620612114861
episode: 45, reward: -1233.36, average _reward: -1296.2989561982718
episode: 46, reward: -1270.65, average _reward: -1300.789819911657
episode: 47, reward: -1199.6, average _reward: -1323.3369189012717
episode: 48, reward: -1159.46, average _reward: -1324.3423466949273
episode: 49, reward: -975.94, average _reward: -1295.1809773318835
episode: 50, reward: -1315.01, average _reward: -1271.432993044808
episode: 51, reward: -1273.25, average _reward: -1259.2050369292438
episode: 52, reward: -1061.19, average _reward: -1246.461049140358
episode: 53, reward: -1217.33, average _reward: -1231.2367259256089
episode: 54, reward: -1390.65, average _reward: -1212.5270508378362
episode: 55, reward: -1318.47, average _reward: -1209.6440997920377
episode: 56, reward: -1329.55, average _reward: -1218.1550078874575
episode: 57, reward: -1270.72, average _reward: -1224.0451091919049
episode: 58, reward: -1463.07, average _reward: -1231.1569382211271
episode: 59, reward: -1274.7, average _reward: -1261.5179627913833
episode: 60, reward: -1355.71, average _reward: -1291.3947416029573
episode: 61, reward: -1431.16, average _reward: -1295.4653690953005
episode: 62, reward: -928.18, average _reward: -1311.2555935486691
episode: 63, reward: -1275.4, average _reward: -1297.954784031932
episode: 64, reward: -915.49, average _reward: -1303.7618379893488
episode: 65, reward: -1413.34, average _reward: -1256.2461817151782
episode: 66, reward: -1371.74, average _reward: -1265.7327449316567
episode: 67, reward: -1395.45, average _reward: -1269.9511909172227
episode: 68, reward: -1204.49, average _reward: -1282.4236678726763
episode: 69, reward: -1413.36, average _reward: -1256.5654100033294
episode: 70, reward: -1343.45, average _reward: -1270.43093486687
episode: 71, reward: -1291.35, average _reward: -1269.2046400473473
episode: 72, reward: -1409.47, average _reward: -1255.2236614176068
episode: 73, reward: -1153.02, average _reward: -1303.3528936099221
episode: 74, reward: -1241.89, average _reward: -1291.1146402519355
episode: 75, reward: -1389.24, average _reward: -1323.75460204643
episode: 76, reward: -1307.99, average _reward: -1321.3452203841516
episode: 77, reward: -1262.11, average _reward: -1314.970273786325
episode: 78, reward: -1163.74, average _reward: -1301.6370330687398
episode: 79, reward: -1210.1, average _reward: -1297.5618793822541
episode: 80, reward: -1254.75, average _reward: -1277.2358084927241
episode: 81, reward: -1367.09, average _reward: -1268.3656534736106
episode: 82, reward: -1278.97, average _reward: -1275.93994060668
```

```
episode: 83, reward: -1431.52, average _reward: -1262.8895849595451
episode: 84, reward: -1377.3, average _reward: -1290.73963987675
episode: 85, reward: -1240.16, average _reward: -1304.2801330915845
episode: 86, reward: -1178.59, average _reward: -1289.3712245549063
episode: 87, reward: -1094.93, average _reward: -1276.4320306751958
episode: 88, reward: -1489.05, average _reward: -1259.7132660678985
episode: 89, reward: -1394.83, average _reward: -1292.2448904274777
episode: 90, reward: -1256.37, average _reward: -1310.7179932806673
episode: 91, reward: -1408.08, average _reward: -1310.8799556781396
episode: 92, reward: -1258.31, average _reward: -1314.9791657968058
episode: 93, reward: -1178.55, average _reward: -1312.9130992630771
episode: 94, reward: -1163.27, average _reward: -1287.6160523416108
episode: 95, reward: -1214.84, average _reward: -1266.2129203235613
episode: 96, reward: -1193.66, average _reward: -1263.6808822552334
episode: 97, reward: -1316.45, average _reward: -1265.1870229422207
episode: 98, reward: -1223.69, average _reward: -1287.3395048303385
episode: 99, reward: -1267.18, average _reward: -1260.803433692495
```