```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
```

```
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
```

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 *

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
        Q = self.critic(states, actions)
```

```
        actions = self.actor_target(next_states)
        Q_next = self.critic_target(next_states, actions.detach())
        Q_targets = rewards + self.gamma*Q_next

        critic_loss = self.critic_criterion(Q, Q_targets)

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

        # Implement actor loss and update actor
        actor_loss = -self.critic(states, self.actor(states)).mean()

        self.actor_optimizer.zero_grad()
        actor_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)
```

```
/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)
```

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:

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

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

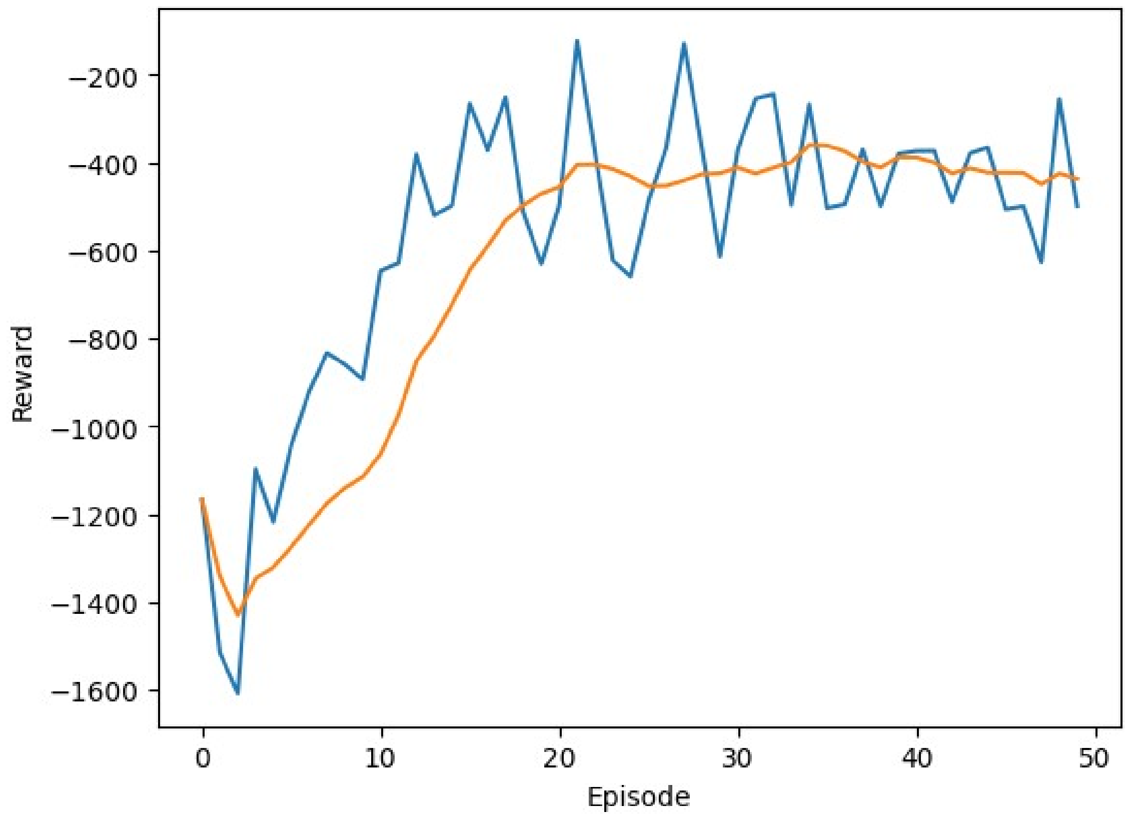
  end for
end for

```python
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()
```

```
episode: 0, reward: -1167.64, average _reward: nan
episode: 1, reward: -1514.58, average _reward: -1167.6387572283693
episode: 2, reward: -1608.08, average _reward: -1341.1111183780035
episode: 3, reward: -1097.11, average _reward: -1430.1008931041797
episode: 4, reward: -1218.12, average _reward: -1346.8538525438344
episode: 5, reward: -1043.71, average _reward: -1321.106161180275
episode: 6, reward: -921.29, average _reward: -1274.874074427158
episode: 7, reward: -833.62, average _reward: -1224.3618837421159
episode: 8, reward: -859.14, average _reward: -1175.518613093238
episode: 9, reward: -893.32, average _reward: -1140.3655637941333
episode: 10, reward: -646.28, average _reward: -1115.660613546083
episode: 11, reward: -629.19, average _reward: -1063.5250431225427
episode: 12, reward: -381.37, average _reward: -974.9859357403141
episode: 13, reward: -520.16, average _reward: -852.3149196796658
episode: 14, reward: -497.79, average _reward: -794.6199827079198
episode: 15, reward: -265.88, average _reward: -722.5878944515523
episode: 16, reward: -372.16, average _reward: -644.8043978787122
episode: 17, reward: -251.14, average _reward: -589.8917367991498
episode: 18, reward: -513.91, average _reward: -531.6440165051577
episode: 19, reward: -630.82, average _reward: -497.121206814603
episode: 20, reward: -497.57, average _reward: -470.8719562473774
episode: 21, reward: -123.09, average _reward: -456.0007592057329
episode: 22, reward: -375.53, average _reward: -405.390723916738
```

```
episode: 23, reward: -622.84, average _reward: -404.8063344520066
episode: 24, reward: -659.33, average _reward: -415.07411559565924
episode: 25, reward: -488.86, average _reward: -431.2272545571318
episode: 26, reward: -365.56, average _reward: -453.52525107764023
episode: 27, reward: -129.3, average _reward: -452.8647715790747
episode: 28, reward: -369.12, average _reward: -440.68063954698744
episode: 29, reward: -613.61, average _reward: -426.20083818385666
episode: 30, reward: -370.79, average _reward: -424.4790486428362
episode: 31, reward: -254.45, average _reward: -411.80048037289663
episode: 32, reward: -244.29, average _reward: -424.9366592007067
episode: 33, reward: -496.93, average _reward: -411.8127339247656
episode: 34, reward: -267.17, average _reward: -399.2217994522826
episode: 35, reward: -503.44, average _reward: -360.0060327068496
episode: 36, reward: -494.9, average _reward: -361.4639927172019
episode: 37, reward: -370.32, average _reward: -374.39873630306283
episode: 38, reward: -498.89, average _reward: -398.5013655036195
episode: 39, reward: -379.81, average _reward: -411.47918668058884
episode: 40, reward: -373.73, average _reward: -388.0998881510234
episode: 41, reward: -373.48, average _reward: -388.39434509501154
episode: 42, reward: -489.49, average _reward: -400.2969291214894
episode: 43, reward: -378.95, average _reward: -424.81683158250627
episode: 44, reward: -365.54, average _reward: -413.0188617293496
episode: 45, reward: -506.04, average _reward: -422.85627978043397
episode: 46, reward: -499.45, average _reward: -423.1165915505263
episode: 47, reward: -627.25, average _reward: -423.57080005284416
episode: 48, reward: -255.12, average _reward: -449.2629818975025
episode: 49, reward: -499.1, average _reward: -424.88599035574225
```



Your Inference

- From the plot we can observe that the average reward converges to the episodic reward of the agent
- Also the effect of the added noise [to the actions] is potrayed as the sporadic increase and decrease of episodic rewards, leading to a balance between exploration and exploitation for the agent.
- Similarly as the episodes increase we can see the noise decreasing this can be attributed to the agent learning a good policy
- From the increasing episodic rewards we can infer that our algorithm tries to reach and maintain balance as soon as possible.