```
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)
```

```
Warning: Gym version v0.24.1 has a number of critical issues with
`gym.make` such that environment observation and action spaces are
incorrectly evaluated, raising incorrect errors and warning . It is
recommend to downgrading to v0.23.1 or upgrading to v0.25.1
```

DDPG uses four neural networks:

1. Q Network
2. Deterministic Policy Network
3. Target Q Network
4. 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
```

```
Couldn't import dot_parser, loading of dot files will not be possible.
```

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
        self.critic_optimizer.zero_grad()
```

```
        y = rewards + self.gamma * self.critic_target(next_states, 
self.actor_target(next_states))
        y = Variable(y.data, requires_grad=False)
        q_c = self.critic(states, actions)
        critic_loss = self.critic_criterion(y, q_c)
        critic_loss.backward()
        self.critic_optimizer.step()

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

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:

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

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

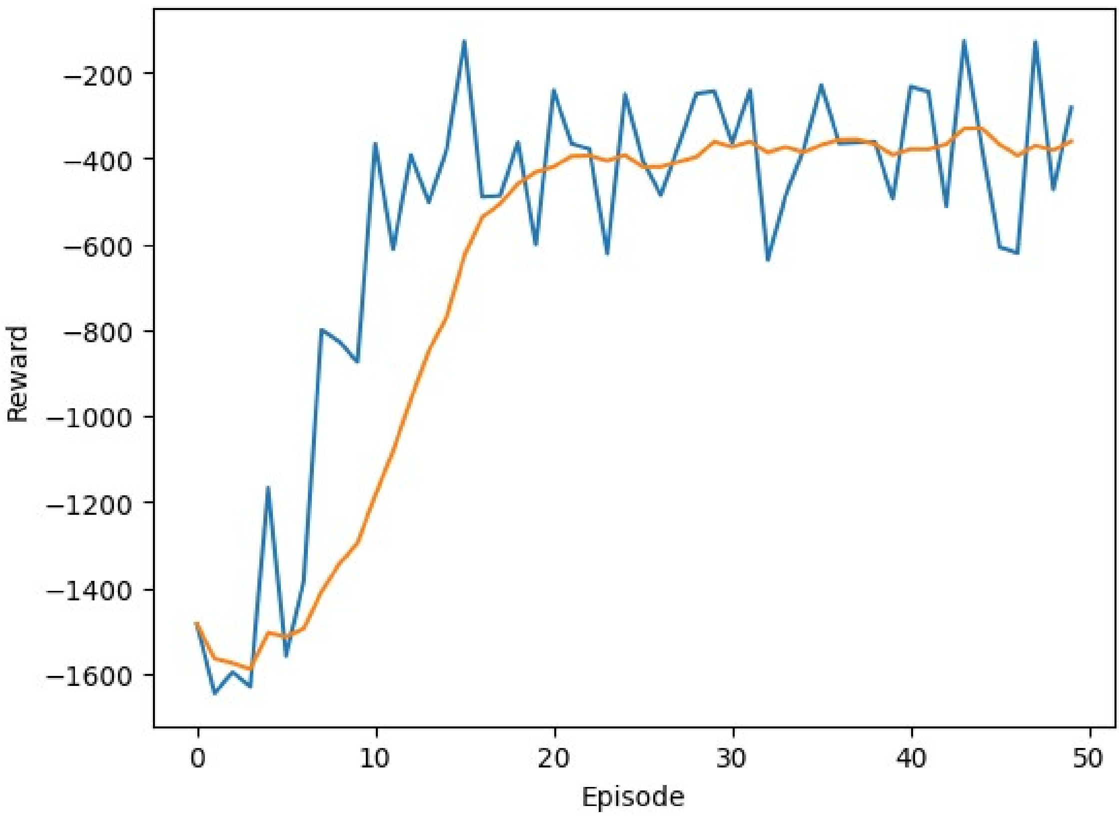
  end for
end for

```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: -1484.33, average _reward: nan
episode: 1, reward: -1646.1, average _reward: -1484.330372144136
episode: 2, reward: -1596.62, average _reward: -1565.2144323063276
episode: 3, reward: -1630.31, average _reward: -1575.6842510956067
episode: 4, reward: -1166.76, average _reward: -1589.3414136735728
episode: 5, reward: -1559.2, average _reward: -1504.8257720250292
episode: 6, reward: -1383.88, average _reward: -1513.8881080547733
episode: 7, reward: -799.97, average _reward: -1495.3152878509604
episode: 8, reward: -827.04, average _reward: -1408.3965966502135
episode: 9, reward: -874.49, average _reward: -1343.800867060284
episode: 10, reward: -366.09, average _reward: -1296.8696621680733
episode: 11, reward: -612.11, average _reward: -1185.0451565858878
episode: 12, reward: -392.37, average _reward: -1081.6462090469413
episode: 13, reward: -503.11, average _reward: -961.2205709488702
episode: 14, reward: -380.44, average _reward: -848.5005973853015
episode: 15, reward: -127.85, average _reward: -769.8682976748121
episode: 16, reward: -489.85, average _reward: -626.7337052893549
episode: 17, reward: -487.37, average _reward: -537.330817799497
episode: 18, reward: -362.64, average _reward: -506.0710951203985
episode: 19, reward: -600.65, average _reward: -459.631470322308
episode: 20, reward: -241.81, average _reward: -432.2472937921837
episode: 21, reward: -366.01, average _reward: -419.8201895243768
episode: 22, reward: -378.51, average _reward: -395.2107282466851
```

```
episode: 23, reward: -621.94, average _reward: -393.82499079564514
episode: 24, reward: -250.02, average _reward: -405.7075232533696
episode: 25, reward: -405.25, average _reward: -392.66557918505833
episode: 26, reward: -486.47, average _reward: -420.4052685464455
episode: 27, reward: -370.56, average _reward: -420.06775343211183
episode: 28, reward: -249.59, average _reward: -408.3869467480588
episode: 29, reward: -243.76, average _reward: -397.0816094067755
episode: 30, reward: -364.18, average _reward: -361.39314496301836
episode: 31, reward: -241.36, average _reward: -373.62966541627395
episode: 32, reward: -636.96, average _reward: -361.16463307766475
episode: 33, reward: -486.59, average _reward: -387.0093537846177
episode: 34, reward: -380.02, average _reward: -373.47446551028
episode: 35, reward: -229.76, average _reward: -386.47430302719914
episode: 36, reward: -364.99, average _reward: -368.92562122352115
episode: 37, reward: -362.67, average _reward: -356.77756395392436
episode: 38, reward: -361.57, average _reward: -355.988781271692
episode: 39, reward: -494.29, average _reward: -367.18727772339156
episode: 40, reward: -233.34, average _reward: -392.23981908523734
episode: 41, reward: -245.02, average _reward: -379.1560095615868
episode: 42, reward: -511.53, average _reward: -379.5212089942753
episode: 43, reward: -126.65, average _reward: -366.9783665558323
episode: 44, reward: -374.51, average _reward: -330.9844577119505
episode: 45, reward: -607.21, average _reward: -330.4339925367966
episode: 46, reward: -620.97, average _reward: -368.178167947386
episode: 47, reward: -129.02, average _reward: -393.7755295856829
episode: 48, reward: -472.31, average _reward: -370.40995808663104
episode: 49, reward: -281.95, average _reward: -381.4843994189353
```



Your Inference

- The agent's reward increases over time. This is shown by the blue line, which represents the instantaneous reward, generally trending upward.
- The agent is learning. This is because the running average, shown in orange, is also increasing over time. The running average helps smooth out the fluctuaitons in the instantaneous reward, giving a clearer picture of the agent's progress.
- We observe initially for about 5 episodes the agent's reward decreases. This might be because the agent when introduces to the environment explores the actions and eventually learn how to balance the pendulum.
- We see that the critic's learning rate is higher than that of the actor's, this is to ensure stable learning as learnt from theory. When we set the learning rate of the actor to the same as that of the critic's (10^{-3}), we observe it converges slower.
- Changing Polyak averaging parameter (τ) to 10^{-3} makes the convergence slower, inturn increasing regret. Increasing τ to 10^{-1} gives us comparable results as in the case of $\tau = 10^{-2}$ (default).