```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 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):
```

```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)
```

```python
        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
        Q_next = self.critic_target.forward(next_states,
self.actor_target.forward(next_states))
        loss_c= self.critic_criterion(self.critic.forward(states,
actions), rewards + self.gamma * Q_next)
        self.critic_optimizer.zero_grad()
        loss_c.backward()
        self.critic_optimizer.step()

        # Implement actor loss and update actor
        loss_a = -self.critic.forward(states,
self.actor.forward(states)).mean()
        self.actor_optimizer.zero_grad()
        loss_a.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_(param.data * self.tau +
target_param.data * (1.0 - self.tau))

        for target_param, param in
zip(self.critic_target.parameters(), self.critic.parameters()):
            target_param.data.copy_(param.data * self.tau +
target_param.data * (1.0 - self.tau))
```

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

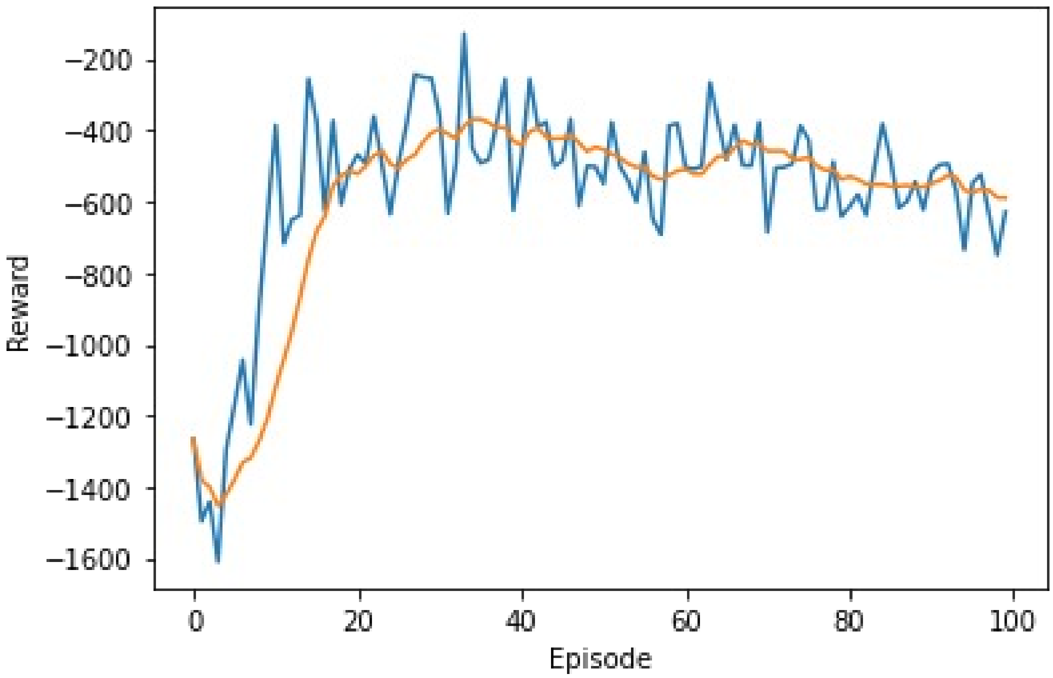
```
import sys
import gym
import numpy as np
import pandas as pd
```

```python
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()
```

```
episode: 0, reward: -1265.49, average _reward: nan
episode: 1, reward: -1495.73, average _reward: -1265.485670374393
episode: 2, reward: -1441.58, average _reward: -1380.6087886688983
```

```
episode: 3, reward: -1609.54, average _reward: -1400.9319703734218
episode: 4, reward: -1293.85, average _reward: -1453.085099584205
episode: 5, reward: -1171.9, average _reward: -1421.238578103545
episode: 6, reward: -1043.53, average _reward: -1379.681538941445
episode: 7, reward: -1224.57, average _reward: -1331.6604028934746
episode: 8, reward: -895.21, average _reward: -1318.2745896865054
episode: 9, reward: -643.25, average _reward: -1271.2676120449119
episode: 10, reward: -385.49, average _reward: -1208.4655582619184
episode: 11, reward: -717.6, average _reward: -1120.4655300240713
episode: 12, reward: -650.41, average _reward: -1042.652681427423
episode: 13, reward: -637.69, average _reward: -963.5358544729603
episode: 14, reward: -255.95, average _reward: -866.3500887210837
episode: 15, reward: -375.5, average _reward: -762.5596806501086
episode: 16, reward: -626.16, average _reward: -682.9202300610905
episode: 17, reward: -373.87, average _reward: -641.1823800775197
episode: 18, reward: -607.48, average _reward: -556.1119047564823
episode: 19, reward: -509.1, average _reward: -527.3390052139315
episode: 20, reward: -468.48, average _reward: -513.9238477497902
episode: 21, reward: -493.27, average _reward: -522.2228938403503
episode: 22, reward: -360.84, average _reward: -499.7894622662755
episode: 23, reward: -497.17, average _reward: -470.83237800546675
episode: 24, reward: -634.16, average _reward: -456.78025511964216
episode: 25, reward: -491.23, average _reward: -494.6012266406066
episode: 26, reward: -380.63, average _reward: -506.17452970478007
episode: 27, reward: -242.92, average _reward: -481.62250847073693
episode: 28, reward: -251.33, average _reward: -468.52727426189904
episode: 29, reward: -253.62, average _reward: -432.91164601269094
episode: 30, reward: -358.19, average _reward: -407.36376900461784
episode: 31, reward: -632.28, average _reward: -396.33557438948907
episode: 32, reward: -493.78, average _reward: -410.2364379734605
episode: 33, reward: -129.61, average _reward: -423.5309029285537
episode: 34, reward: -450.9, average _reward: -386.77580243675663
episode: 35, reward: -491.99, average _reward: -368.4504645090785
episode: 36, reward: -481.83, average _reward: -368.5262576167018
episode: 37, reward: -378.8, average _reward: -378.6457941290577
episode: 38, reward: -255.83, average _reward: -392.23405334264095
episode: 39, reward: -625.02, average _reward: -392.68469307060104
episode: 40, reward: -477.32, average _reward: -429.82478628069464
episode: 41, reward: -256.21, average _reward: -441.73761711560974
episode: 42, reward: -390.28, average _reward: -404.1312771930958
episode: 43, reward: -376.54, average _reward: -393.7810326650307
episode: 44, reward: -502.29, average _reward: -418.4736971321261
episode: 45, reward: -483.78, average _reward: -423.6118443910641
episode: 46, reward: -368.72, average _reward: -422.7906196143449
episode: 47, reward: -611.64, average _reward: -411.4797666276736
episode: 48, reward: -500.77, average _reward: -434.76401119998775
episode: 49, reward: -501.6, average _reward: -459.25742715659555
episode: 50, reward: -550.0, average _reward: -446.91547335185015
episode: 51, reward: -376.87, average _reward: -454.1833989347254
episode: 52, reward: -500.74, average _reward: -466.2491302288134
```

```
episode: 53, reward: -540.36, average _reward: -477.2945448031437
episode: 54, reward: -600.54, average _reward: -493.6769082253644
episode: 55, reward: -461.2, average _reward: -503.5019726303458
episode: 56, reward: -648.63, average _reward: -501.24344726406525
episode: 57, reward: -692.35, average _reward: -529.2341605899452
episode: 58, reward: -387.18, average _reward: -537.304775880867
episode: 59, reward: -380.94, average _reward: -525.9461410413492
episode: 60, reward: -504.32, average _reward: -513.8801274245656
episode: 61, reward: -508.24, average _reward: -509.31161859875283
episode: 62, reward: -501.81, average _reward: -522.4487472437074
episode: 63, reward: -265.76, average _reward: -522.5563964374105
episode: 64, reward: -378.19, average _reward: -495.0957591675757
episode: 65, reward: -481.93, average _reward: -472.8609763743082
episode: 66, reward: -384.68, average _reward: -474.93434801877504
episode: 67, reward: -498.58, average _reward: -448.53984826838735
episode: 68, reward: -499.65, average _reward: -429.16271305130374
episode: 69, reward: -376.78, average _reward: -440.40925631242374
episode: 70, reward: -685.31, average _reward: -439.99366121036473
episode: 71, reward: -505.7, average _reward: -458.0932453606376
episode: 72, reward: -503.4, average _reward: -457.8391106637958
episode: 73, reward: -494.67, average _reward: -457.9975220532212
episode: 74, reward: -387.02, average _reward: -480.88891912168594
episode: 75, reward: -426.43, average _reward: -481.77249105343645
episode: 76, reward: -622.02, average _reward: -476.2224147594511
episode: 77, reward: -619.82, average _reward: -499.95610025901885
episode: 78, reward: -487.86, average _reward: -512.0804932865531
episode: 79, reward: -641.14, average _reward: -510.90230898425153
episode: 80, reward: -613.65, average _reward: -537.3377513821182
episode: 81, reward: -580.15, average _reward: -530.1719742444709
episode: 82, reward: -636.37, average _reward: -537.6164344416534
episode: 83, reward: -507.75, average _reward: -550.9142707624621
episode: 84, reward: -381.76, average _reward: -552.2223637252683
episode: 85, reward: -481.88, average _reward: -551.6960794259206
episode: 86, reward: -616.85, average _reward: -557.2410482770158
episode: 87, reward: -599.81, average _reward: -556.7236080935903
episode: 88, reward: -544.77, average _reward: -554.7224556161397
episode: 89, reward: -622.39, average _reward: -560.4133983445361
episode: 90, reward: -517.94, average _reward: -558.5392274086203
episode: 91, reward: -496.5, average _reward: -548.9674567241224
episode: 92, reward: -493.8, average _reward: -540.6031376075605
episode: 93, reward: -570.64, average _reward: -526.3460852078977
episode: 94, reward: -735.45, average _reward: -532.6349185506544
episode: 95, reward: -547.07, average _reward: -568.003792277337
episode: 96, reward: -524.22, average _reward: -574.5227217820586
episode: 97, reward: -634.1, average _reward: -565.2599522384264
episode: 98, reward: -749.77, average _reward: -568.6889908149999
episode: 99, reward: -628.52, average _reward: -589.1882840036981
```



−200
−400
−600
−800
−1000
−1200
−1400
−1600
Reward
0
20
40
60
80
100
Episode