In [ ]:
```python
import numpy as np
import gymnasium as gym
from collections import deque
import random

# Ornstein-Ulhenbeck Process
# Taken from #https://github.com/vitchyr/rlkit/blob/master/rlkit/exploration_strategies/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

In [ ]:

```python
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):
    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:

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

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

$$\text{where} \quad \tau \ll 1$$

In [ ]:
```python
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=1.5e-3, gamma=0.99, tau=
        # 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
        target = rewards + self.gamma * self.critic_target.forward(next_states, self.actor_target.forward(next_state
        target = target.detach()

        self.critic_optimizer.zero_grad()
        critic_loss = self.critic_criterion(self.critic.forward(states, actions), target)
        critic_loss.backward()
        self.critic_optimizer.step()

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

        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)


```

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:

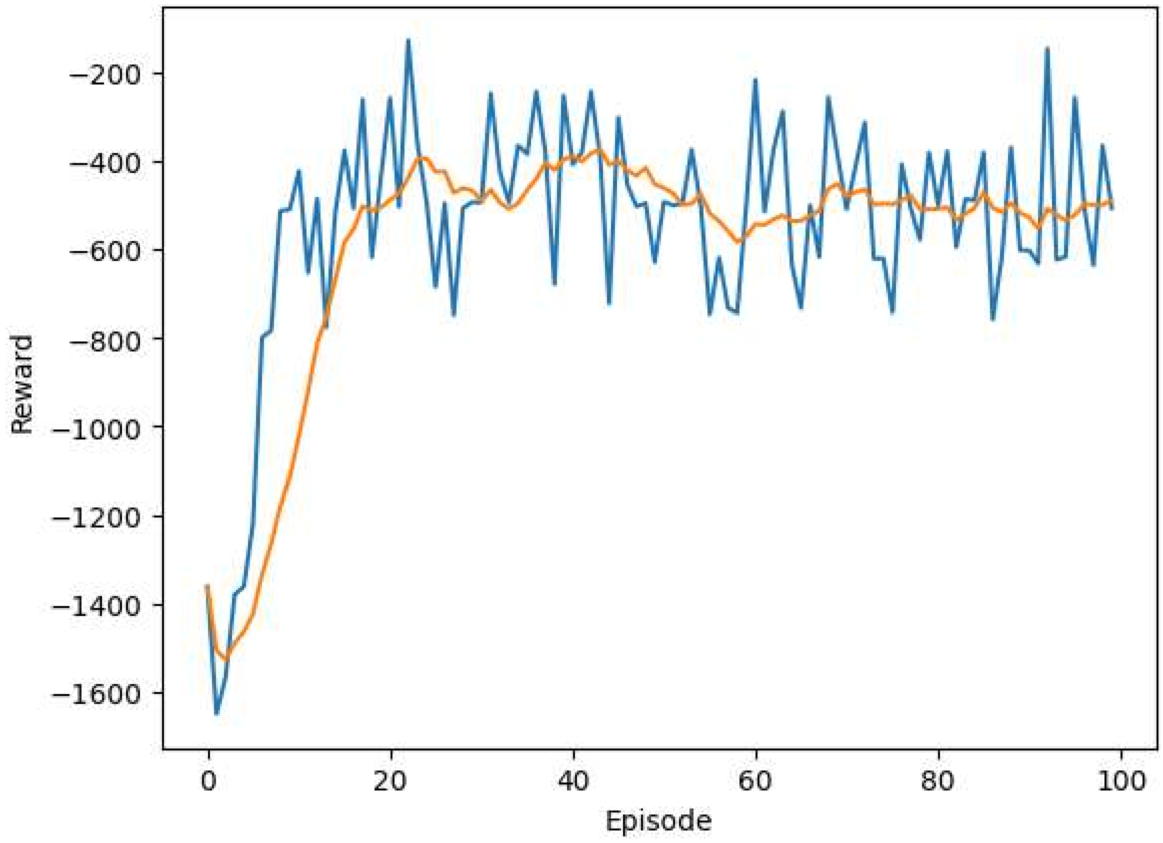
In [ ]:

```
import sys
import gymnasium as gym
import numpy as np
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/pendu
env = NormalizedEnv(gym.make("Pendulum-v1", g=9.81))

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
    done = 0

    for step in range(200):
        action = agent.get_action(state)

        #Add noise to action
        action = noise.get_action(action, step)

        new_state, reward, terminated, truncated, _ = env.step(action)
        if terminated or truncated:
            done = 1  # Ensuring backward compatibility
        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_rewa
            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/python3.11/dist-packages/numpy/_core/fromnumeric.py:3596: RuntimeWarning: Mean of empty slice.
  return _methods._mean(a, axis=axis, dtype=dtype,
/usr/local/lib/python3.11/dist-packages/numpy/_core/_methods.py:138: RuntimeWarning: invalid value encountered in scala
r divide
  ret = ret.dtype.type(ret / rcount)
```

```
episode: 0, reward: -1361.79, average _reward: nan
episode: 1, reward: -1649.11, average _reward: -1361.7929196444409
episode: 2, reward: -1566.21, average _reward: -1505.4507622115516
episode: 3, reward: -1378.96, average _reward: -1525.7037128035056
episode: 4, reward: -1360.95, average _reward: -1489.018864163167
episode: 5, reward: -1220.93, average _reward: -1463.404763496853
episode: 6, reward: -798.88, average _reward: -1422.9918552055262
episode: 7, reward: -783.03, average _reward: -1333.8332856681773
episode: 8, reward: -514.08, average _reward: -1264.9834472588589
episode: 9, reward: -509.55, average _reward: -1181.550027188068
episode: 10, reward: -423.07, average _reward: -1114.3496905883676
episode: 11, reward: -652.88, average _reward: -1020.4769979570644
episode: 12, reward: -485.88, average _reward: -920.8545808602912
episode: 13, reward: -775.25, average _reward: -812.821149638496
episode: 14, reward: -516.0, average _reward: -752.450073605027
episode: 15, reward: -376.46, average _reward: -667.9550993371506
episode: 16, reward: -507.54, average _reward: -583.5085389380858
episode: 17, reward: -262.17, average _reward: -554.3741184083075
episode: 18, reward: -618.09, average _reward: -502.28743924574457
episode: 19, reward: -436.74, average _reward: -512.6884455554484
episode: 20, reward: -258.65, average _reward: -505.40809528752277
episode: 21, reward: -504.32, average _reward: -488.96669449187175
episode: 22, reward: -128.75, average _reward: -474.1101098388766
episode: 23, reward: -359.82, average _reward: -438.397525229038
episode: 24, reward: -491.74, average _reward: -396.8540542430846
episode: 25, reward: -685.0, average _reward: -394.42863066080037
episode: 26, reward: -496.48, average _reward: -425.2820887451726
episode: 27, reward: -748.03, average _reward: -424.17593900289387
episode: 28, reward: -507.22, average _reward: -472.7617108106195
episode: 29, reward: -493.65, average _reward: -461.67419132218737
episode: 30, reward: -494.66, average _reward: -467.3650733013001
episode: 31, reward: -248.85, average _reward: -490.96629281589077
episode: 32, reward: -425.61, average _reward: -465.4193906406011
episode: 33, reward: -494.64, average _reward: -495.1052834253378
episode: 34, reward: -365.48, average _reward: -508.58714612094826
episode: 35, reward: -384.7, average _reward: -495.9608141746477
episode: 36, reward: -244.9, average _reward: -465.9315607403827
episode: 37, reward: -381.09, average _reward: -440.7744165161629
episode: 38, reward: -679.18, average _reward: -404.08106435972667
episode: 39, reward: -253.83, average _reward: -421.277403016387
episode: 40, reward: -408.41, average _reward: -397.29557599076304
episode: 41, reward: -381.16, average _reward: -388.6704804471452
episode: 42, reward: -244.77, average _reward: -401.9014736503667
episode: 43, reward: -389.05, average _reward: -383.81722998816997
episode: 44, reward: -722.68, average _reward: -373.2588616060537
episode: 45, reward: -302.24, average _reward: -408.9783163072316
episode: 46, reward: -454.66, average _reward: -400.73199429948113
episode: 47, reward: -503.27, average _reward: -421.7077255875727
episode: 48, reward: -496.07, average _reward: -433.9258305555819
episode: 49, reward: -629.1, average _reward: -415.6144833813032
episode: 50, reward: -493.23, average _reward: -453.141180726308
episode: 51, reward: -500.69, average _reward: -461.6232318489835
episode: 52, reward: -497.66, average _reward: -473.5766004777197
episode: 53, reward: -374.7, average _reward: -498.8663766551566
episode: 54, reward: -501.6, average _reward: -497.43074035867767
episode: 55, reward: -746.5, average _reward: -475.32349152004207
episode: 56, reward: -619.22, average _reward: -519.7491172154988
episode: 57, reward: -733.14, average _reward: -536.2053105287284
episode: 58, reward: -743.59, average _reward: -559.1921496262502
episode: 59, reward: -499.47, average _reward: -583.9446593428314
episode: 60, reward: -218.0, average _reward: -570.9817904411683
episode: 61, reward: -515.02, average _reward: -543.4581523978086
episode: 62, reward: -373.08, average _reward: -544.8911047110428
episode: 63, reward: -288.37, average _reward: -532.4329316129115
episode: 64, reward: -634.72, average _reward: -523.7997730246258
episode: 65, reward: -732.93, average _reward: -537.1110059252669
episode: 66, reward: -500.92, average _reward: -535.7543801985029
episode: 67, reward: -617.59, average _reward: -523.9243011154606
episode: 68, reward: -257.26, average _reward: -512.3694654560917
episode: 69, reward: -385.75, average _reward: -463.73580768071196
episode: 70, reward: -509.78, average _reward: -452.36313165838254
episode: 71, reward: -411.96, average _reward: -481.5416618035735
episode: 72, reward: -313.8, average _reward: -471.2349031832392
episode: 73, reward: -621.63, average _reward: -465.30640762172925
episode: 74, reward: -621.18, average _reward: -498.63290165386417
episode: 75, reward: -742.32, average _reward: -497.2796413387603
episode: 76, reward: -408.05, average _reward: -498.2185227267161
episode: 77, reward: -507.73, average _reward: -488.9312601105895
episode: 78, reward: -578.5, average _reward: -477.9451309229172
episode: 79, reward: -381.54, average _reward: -510.06972064121993
episode: 80, reward: -501.08, average _reward: -509.6491776069894
episode: 81, reward: -378.48, average _reward: -508.77931114179756
episode: 82, reward: -595.09, average _reward: -505.4322746527346
episode: 83, reward: -486.87, average _reward: -533.5611358224885
episode: 84, reward: -489.41, average _reward: -520.0854987775871
episode: 85, reward: -381.08, average _reward: -506.90857535313154
episode: 86, reward: -757.74, average _reward: -470.7843137601791
episode: 87, reward: -616.29, average _reward: -505.7527927723139
```

```
episode: 88, reward: -370.21, average _reward: -516.6090882670534
episode: 89, reward: -602.26, average _reward: -495.77947898360236
episode: 90, reward: -602.99, average _reward: -517.8518743740658
episode: 91, reward: -632.51, average _reward: -528.0420585921013
episode: 92, reward: -147.71, average _reward: -553.4448956442254
episode: 93, reward: -624.15, average _reward: -508.7068430080549
episode: 94, reward: -617.34, average _reward: -522.4347020453938
episode: 95, reward: -258.08, average _reward: -535.2269944871384
episode: 96, reward: -506.54, average _reward: -522.9274698822094
episode: 97, reward: -636.45, average _reward: -497.808181803762
episode: 98, reward: -365.54, average _reward: -499.8237831674202
episode: 99, reward: -507.32, average _reward: -499.35746216417147
```



In [ ]:
```
1
```

In [ ]:
```
1
```