In [1]:
```
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_s
class OUNoise(object):
    def __init__(self, action_space, mu=0.0, theta=0.15, max_sigma=0.3, min_si
        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.ac
        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(
        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)
```

```python
    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

    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.

In [2]:

```python
import torch
import torch.nn as nn
```

```python
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 = 3
        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 [3]:

```
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_
        # 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_actio
        self.critic = Critic(self.num_states + self.num_actions, hidden_size,
        self.critic_target = Critic(self.num_states + self.num_actions, hidden_


        for target_param, param in zip(self.actor_target.parameters(), self.ac
            target_param.data.copy_(param.data)

        for target_param, param in zip(self.critic_target.parameters(), self.c
            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_l
        self.critic_optimizer = optim.Adam(self.critic.parameters(), lr=critic_

    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_si
        states = torch.FloatTensor(states)
        actions = torch.FloatTensor(actions)
        rewards = torch.FloatTensor(rewards)
        next_states = torch.FloatTensor(next_states)

        # Implement critic loss and update critic
        Qvals = self.critic.forward(states, actions)
        next_actions = self.actor_target.forward(next_states)
        next_Q = self.critic_target.forward(next_states, next_actions.detach()
        Qprime = rewards + self.gamma * next_Q
        critic_loss = self.critic_criterion(Qvals, Qprime)

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

        self.actor_optimizer.zero_grad()
        policy_loss.backward()
        self.actor_optimizer.step()

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

        # update target networks
        for target_param, param in zip(self.actor_target.parameters(), self.ac
            target_param.data.copy_(param.data * self.tau + target_param.data
```

```
        for target_param, param in zip(self.critic_target.parameters(), self.c
            target_param.data.copy_(param.data * 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

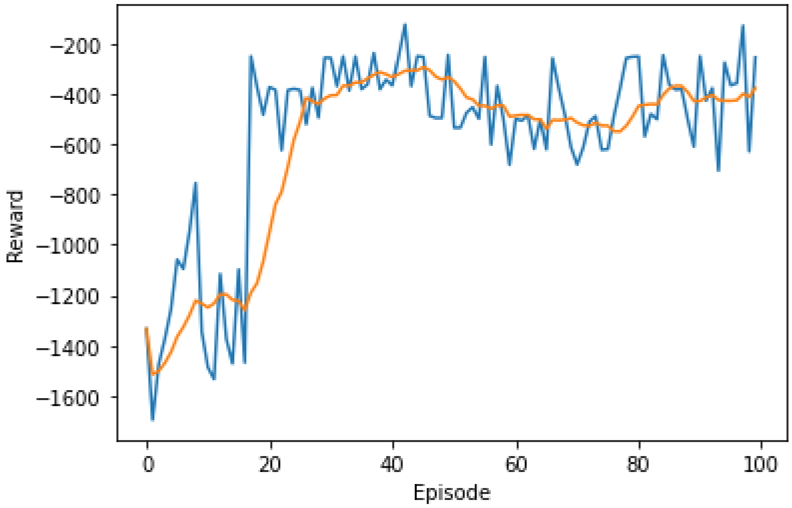
In [4]:
```
import sys
import gym
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# For more info on the Pendulum environment, check out https://www.gymlibrary.
env = NormalizedEnv(gym.make("Pendulum-v1"))

agent = DDPGagent(env)
noise = OUNoise(env.action_space)
batch_size = 128
rewards = []
avg_rewards = []
```

```python
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"
            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: WA
RN: Initializing wrapper in old step API which returns one bool instead of tw
o. 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_compatibility.py:
39: DeprecationWarning: WARN: Initializing environment in old step API which r
eturns one bool instead of two. It is recommended to set `new_step_api=True` t
o use new step API. This will be the default behaviour in future.
  deprecation(
<ipython-input-3-0dccb560461a>:40: UserWarning: Creating a tensor from a list
of numpy.ndarrays is extremely slow. Please consider converting the list to a
single numpy.ndarray with numpy.array() before converting to a tensor. (Trigge
red internally at ../torch/csrc/utils/tensor_new.cpp:230.)
  states = torch.FloatTensor(states)
/usr/local/lib/python3.9/dist-packages/numpy/core/fromnumeric.py:3474: Runtime
Warning: Mean of empty slice.
  return _methods._mean(a, axis=axis, dtype=dtype,
/usr/local/lib/python3.9/dist-packages/numpy/core/_methods.py:189: RuntimeWarn
ing: invalid value encountered in double_scalars
  ret = ret.dtype.type(ret / rcount)
```

```
episode: 0, reward: -1333.43, average _reward: nan
episode: 1, reward: -1695.38, average _reward: -1333.4348683492979
episode: 2, reward: -1473.64, average _reward: -1514.4050465825846
episode: 3, reward: -1371.55, average _reward: -1500.817354004855
episode: 4, reward: -1253.25, average _reward: -1468.5003310699265
episode: 5, reward: -1057.2, average _reward: -1425.4493343265506
episode: 6, reward: -1094.43, average _reward: -1364.0744880135464
episode: 7, reward: -948.5, average _reward: -1325.5533394238394
episode: 8, reward: -754.8, average _reward: -1278.422243627622
episode: 9, reward: -1344.02, average _reward: -1220.2414629260184
episode: 10, reward: -1486.21, average _reward: -1232.6188961942848
episode: 11, reward: -1533.24, average _reward: -1247.896460012798
episode: 12, reward: -1114.49, average _reward: -1231.6828180526895
episode: 13, reward: -1372.88, average _reward: -1195.767851064887
episode: 14, reward: -1471.2, average _reward: -1195.900589783641
episode: 15, reward: -1096.39, average _reward: -1217.6958996822189
episode: 16, reward: -1467.38, average _reward: -1221.6150517623178
episode: 17, reward: -250.0, average _reward: -1258.910505223885
episode: 18, reward: -373.19, average _reward: -1189.0595877245134
episode: 19, reward: -483.25, average _reward: -1150.89885616096
episode: 20, reward: -372.77, average _reward: -1064.82179364476
episode: 21, reward: -383.99, average _reward: -953.477634066425
episode: 22, reward: -624.09, average _reward: -838.5530957333779
episode: 23, reward: -384.95, average _reward: -789.5124354215332
episode: 24, reward: -379.54, average _reward: -690.7196021601986
episode: 25, reward: -384.79, average _reward: -581.5542446180293
episode: 26, reward: -520.49, average _reward: -510.394316847173
episode: 27, reward: -375.59, average _reward: -415.7048520328426
episode: 28, reward: -494.35, average _reward: -428.2643165463475
episode: 29, reward: -255.91, average _reward: -440.3805881511168
episode: 30, reward: -256.62, average _reward: -417.64659201622106
episode: 31, reward: -369.3, average _reward: -406.0321238292559
episode: 32, reward: -250.68, average _reward: -404.5629228243144
episode: 33, reward: -386.67, average _reward: -367.222406478004
episode: 34, reward: -250.73, average _reward: -367.39452635880036
episode: 35, reward: -381.24, average _reward: -354.51286522501744
episode: 36, reward: -361.41, average _reward: -354.1575821779678
episode: 37, reward: -237.31, average _reward: -338.24968484888745
episode: 38, reward: -381.73, average _reward: -324.4213092479647
episode: 39, reward: -343.29, average _reward: -313.1589403035373
episode: 40, reward: -366.69, average _reward: -321.8969545912129
episode: 41, reward: -246.41, average _reward: -332.903889160492
episode: 42, reward: -123.66, average _reward: -320.6146411805546
episode: 43, reward: -369.04, average _reward: -307.91265675410693
episode: 44, reward: -249.88, average _reward: -306.1495943286294
episode: 45, reward: -252.99, average _reward: -306.06517123593085
episode: 46, reward: -486.99, average _reward: -293.2402749414035
episode: 47, reward: -495.79, average _reward: -305.79810961185103
episode: 48, reward: -495.63, average _reward: -331.64676506112204
episode: 49, reward: -244.56, average _reward: -343.0371566472162
episode: 50, reward: -534.88, average _reward: -333.1647562829404
episode: 51, reward: -534.49, average _reward: -349.9830887636446
episode: 52, reward: -475.14, average _reward: -378.79167071292204
episode: 53, reward: -452.91, average _reward: -413.93978217637266
episode: 54, reward: -498.86, average _reward: -422.327104032444
episode: 55, reward: -252.64, average _reward: -447.2242355118894
episode: 56, reward: -601.27, average _reward: -447.18909264096465
```

```
episode: 57, reward: -368.1, average _reward: -458.6175847099783
episode: 58, reward: -499.38, average _reward: -445.84798688935837
episode: 59, reward: -682.23, average _reward: -446.2230339006008
episode: 60, reward: -499.21, average _reward: -489.9898455556337
episode: 61, reward: -506.39, average _reward: -486.4235581389238
episode: 62, reward: -483.91, average _reward: -483.6130431525952
episode: 63, reward: -618.98, average _reward: -484.4900602425869
episode: 64, reward: -501.7, average _reward: -501.0971498418332
episode: 65, reward: -621.08, average _reward: -501.3811605153986
episode: 66, reward: -257.51, average _reward: -538.2254507603309
episode: 67, reward: -376.16, average _reward: -503.84924794882943
episode: 68, reward: -482.63, average _reward: -504.6552150631498
episode: 69, reward: -611.72, average _reward: -502.9802489392193
episode: 70, reward: -681.12, average _reward: -495.92941649272007
episode: 71, reward: -612.7, average _reward: -514.1195467632621
episode: 72, reward: -510.39, average _reward: -524.7509075772444
episode: 73, reward: -488.4, average _reward: -527.3986799593965
episode: 74, reward: -622.22, average _reward: -514.3404197244197
episode: 75, reward: -619.79, average _reward: -526.3925173453626
episode: 76, reward: -486.39, average _reward: -526.2635974811219
episode: 77, reward: -377.36, average _reward: -549.1513538500466
episode: 78, reward: -257.28, average _reward: -549.2715553145692
episode: 79, reward: -251.99, average _reward: -526.7363609270759
episode: 80, reward: -251.02, average _reward: -490.7634506794285
episode: 81, reward: -568.95, average _reward: -447.7540230122242
episode: 82, reward: -479.88, average _reward: -443.37853839994233
episode: 83, reward: -500.68, average _reward: -440.32769031147
episode: 84, reward: -245.58, average _reward: -441.5552846216424
episode: 85, reward: -363.95, average _reward: -403.89136110196716
episode: 86, reward: -382.81, average _reward: -378.30661898357437
episode: 87, reward: -380.87, average _reward: -367.94873773733826
episode: 88, reward: -501.16, average _reward: -368.29963299997314
episode: 89, reward: -610.17, average _reward: -392.6874436423462
episode: 90, reward: -249.33, average _reward: -428.5054930033357
episode: 91, reward: -426.65, average _reward: -428.3359625141613
episode: 92, reward: -376.89, average _reward: -414.10613624203825
episode: 93, reward: -705.41, average _reward: -403.80717826702846
episode: 94, reward: -275.02, average _reward: -424.28077905950687
episode: 95, reward: -365.81, average _reward: -427.22538002307164
episode: 96, reward: -357.28, average _reward: -427.41139579909077
episode: 97, reward: -128.53, average _reward: -424.8582319774315
episode: 98, reward: -628.98, average _reward: -399.62410339034295
episode: 99, reward: -254.54, average _reward: -412.40671401820816
```



In [7]:
```
!jupyter nbconvert --to html "/content/drive/MyDrive/Colab Notebooks/Tut6_DDPG
```