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/explorat
class OUNoise(object):
    def __init__(self, action_space, mu=0.0, theta=0.15, max_sigma=0.3, m
        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(se
        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) *
        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,

    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]:
```
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_rat
        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:

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

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

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

In [9]:

```
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, cr
        # 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_
        self.critic = Critic(self.num_states + self.num_actions, hidden_s
        self.critic_target = Critic(self.num_states + self.num_actions, h


        for target_param, param in zip(self.actor_target.parameters(), se
```

```
            target_param.data.copy_(param.data)

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

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

        # Implement critic loss and update critic
        mu = self.actor_target.forward(next_states)
        q_new = self.critic_target.forward(next_states,mu)
        y = rewards + self.gamma*q_new

        q = self.critic.forward(states,actions)
        critic_loss = self.critic_criterion(y,q)

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

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

        # update target networks
        with torch.no_grad():
          for target_param, params in zip(self.actor_target.parameters(),
            target_param.copy_(self.tau*params + (1 - self.tau)*target_pa
          for target_param, params in zip(self.critic_target.parameters()
            target_param.copy_(self.tau*params + (1 - self.tau)*target_pa
```

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:

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

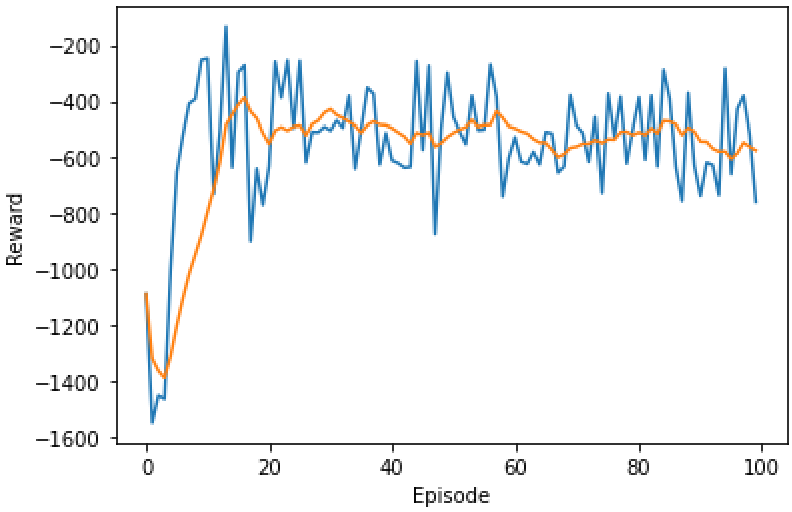
  end for
end for

In [10]:
```
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.gymlib
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: {
            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: -1088.29, average _reward: nan
episode: 1, reward: -1549.99, average _reward: -1088.2855946450463
episode: 2, reward: -1451.25, average _reward: -1319.1396624578101
episode: 3, reward: -1464.79, average _reward: -1363.1749442748107
episode: 4, reward: -997.05, average _reward: -1388.5781069779125
episode: 5, reward: -651.24, average _reward: -1310.271507006124
episode: 6, reward: -519.29, average _reward: -1200.4332524550448
episode: 7, reward: -408.53, average _reward: -1103.126913927288
episode: 8, reward: -392.69, average _reward: -1016.3016915138563
episode: 9, reward: -252.43, average _reward: -947.0109780241067
episode: 10, reward: -247.13, average _reward: -877.5530368433734
episode: 11, reward: -729.73, average _reward: -793.4379699006802
episode: 12, reward: -506.26, average _reward: -711.4111437260407
episode: 13, reward: -132.81, average _reward: -616.9123397873207
episode: 14, reward: -634.75, average _reward: -483.7145196512268
episode: 15, reward: -296.68, average _reward: -447.4850542514955
episode: 16, reward: -271.3, average _reward: -412.02895036957864
episode: 17, reward: -899.12, average _reward: -387.230254852556
episode: 18, reward: -638.73, average _reward: -436.29001587959374
episode: 19, reward: -769.13, average _reward: -460.8941746791459
episode: 20, reward: -629.99, average _reward: -512.563848649108
episode: 21, reward: -257.96, average _reward: -550.8494710403378
episode: 22, reward: -387.68, average _reward: -503.6729128006218
episode: 23, reward: -253.44, average _reward: -491.8153251374648
episode: 24, reward: -495.8, average _reward: -503.8786083991772
episode: 25, reward: -254.97, average _reward: -489.98400062820775
episode: 26, reward: -616.71, average _reward: -485.8128075337786
episode: 27, reward: -508.5, average _reward: -520.3539230300955
episode: 28, reward: -509.58, average _reward: -481.29134901147074
episode: 29, reward: -490.52, average _reward: -468.37630658369943
episode: 30, reward: -503.71, average _reward: -440.515229892945
episode: 31, reward: -468.68, average _reward: -427.8866420470392
episode: 32, reward: -493.97, average _reward: -448.95887485591464
episode: 33, reward: -380.34, average _reward: -459.5881835584763
episode: 34, reward: -639.46, average _reward: -472.27795961628397
episode: 35, reward: -498.25, average _reward: -486.64380214833454
episode: 36, reward: -351.58, average _reward: -510.97153887167343
episode: 37, reward: -373.93, average _reward: -484.4581093331015
episode: 38, reward: -624.63, average _reward: -471.0012371745418
episode: 39, reward: -511.67, average _reward: -482.50674473227383
episode: 40, reward: -609.5, average _reward: -484.62248909592654
episode: 41, reward: -619.16, average _reward: -495.20147825161837
episode: 42, reward: -635.15, average _reward: -510.24948193100664
episode: 43, reward: -632.77, average _reward: -524.3674004631196
episode: 44, reward: -256.73, average _reward: -549.6104822551804
episode: 45, reward: -572.27, average _reward: -511.3368789504608
episode: 46, reward: -272.9, average _reward: -518.7397353861703
episode: 47, reward: -873.42, average _reward: -510.8713644838789
episode: 48, reward: -496.23, average _reward: -560.8207124186853
episode: 49, reward: -298.32, average _reward: -547.980363248126
episode: 50, reward: -457.32, average _reward: -526.6450799498793
episode: 51, reward: -507.27, average _reward: -511.4274011792384
episode: 52, reward: -552.02, average _reward: -500.238275076988
episode: 53, reward: -379.7, average _reward: -491.9248318502732
episode: 54, reward: -502.37, average _reward: -466.6173975484339
episode: 55, reward: -500.24, average _reward: -491.18164291580354
episode: 56, reward: -268.32, average _reward: -483.9777983701444
episode: 57, reward: -384.86, average _reward: -483.51982343541505
episode: 58, reward: -739.16, average _reward: -434.66358808664006
episode: 59, reward: -600.19, average _reward: -458.95687452408265
```

```
episode: 60, reward: -526.67, average _reward: -489.1436255475945
episode: 61, reward: -613.7, average _reward: -496.07887481720843
episode: 62, reward: -621.73, average _reward: -506.7217292588957
episode: 63, reward: -578.81, average _reward: -513.6927326614281
episode: 64, reward: -624.14, average _reward: -533.6039904789761
episode: 65, reward: -508.72, average _reward: -545.7812660656235
episode: 66, reward: -513.49, average _reward: -546.6291504742278
episode: 67, reward: -653.36, average _reward: -571.1467860563735
episode: 68, reward: -631.47, average _reward: -597.9972419992819
episode: 69, reward: -377.94, average _reward: -587.2279679517003
episode: 70, reward: -485.34, average _reward: -565.0030651955544
episode: 71, reward: -511.45, average _reward: -560.8696092056164
episode: 72, reward: -614.58, average _reward: -550.6444644289262
episode: 73, reward: -455.91, average _reward: -549.9296063004485
episode: 74, reward: -726.22, average _reward: -537.6401229208875
episode: 75, reward: -373.0, average _reward: -547.847856067653
episode: 76, reward: -527.1, average _reward: -534.2760959795769
episode: 77, reward: -383.64, average _reward: -535.6373154557876
episode: 78, reward: -620.73, average _reward: -508.6645524176307
episode: 79, reward: -498.33, average _reward: -507.59034622143463
episode: 80, reward: -385.52, average _reward: -519.6294010980622
episode: 81, reward: -607.14, average _reward: -509.6479123896418
episode: 82, reward: -379.48, average _reward: -519.2175867710475
episode: 83, reward: -631.41, average _reward: -495.7074982957758
episode: 84, reward: -288.02, average _reward: -513.2570945343725
episode: 85, reward: -391.93, average _reward: -469.4366733834412
episode: 86, reward: -633.12, average _reward: -471.32953168602927
episode: 87, reward: -754.89, average _reward: -481.9308600113392
episode: 88, reward: -370.33, average _reward: -519.0563168538954
episode: 89, reward: -627.59, average _reward: -494.01623750759916
episode: 90, reward: -736.8, average _reward: -506.9417095270337
episode: 91, reward: -617.12, average _reward: -542.0693402268632
episode: 92, reward: -625.91, average _reward: -543.0673272979778
episode: 93, reward: -734.34, average _reward: -567.7100353576884
episode: 94, reward: -282.83, average _reward: -578.0033869582836
episode: 95, reward: -658.4, average _reward: -577.4847842260181
episode: 96, reward: -427.38, average _reward: -604.1325754590268
episode: 97, reward: -380.55, average _reward: -583.559217598897
episode: 98, reward: -516.75, average _reward: -546.1251754368129
episode: 99, reward: -758.11, average _reward: -560.7677266235804
```



In [11]:
```
!jupyter nbconvert --to html "/content/drive/MyDrive/Colab Notebooks/Tut6
```

```
Mounted at /content/drive
```

In [ ]:
```
!pip install nbconvert
!sudo apt-get install texlive-xetex texlive-fonts-recommended texlive-pla
```

```
tl-paper: setting paper size for pdftex to a4: /var/lib/texmf/tex/generi
c/config/pdftexconfig.tex
debconf: unable to initialize frontend: Dialog
debconf: (No usable dialog-like program is installed, so the dialog based
frontend cannot be used. at /usr/share/perl5/Debconf/FrontEnd/Dialog.pm l
ine 76.)
debconf: falling back to frontend: Readline
```

In [ ]:
```

```