Importing the libraries

```
import os
import time
import random
import numpy as np
import matplotlib.pyplot as plt
import pybullet_envs
import gym
import torch
import torch.nn as nn
import torch.nn.functional as F
import gym.wrappers
from torch.autograd import Variable
from collections import deque
```

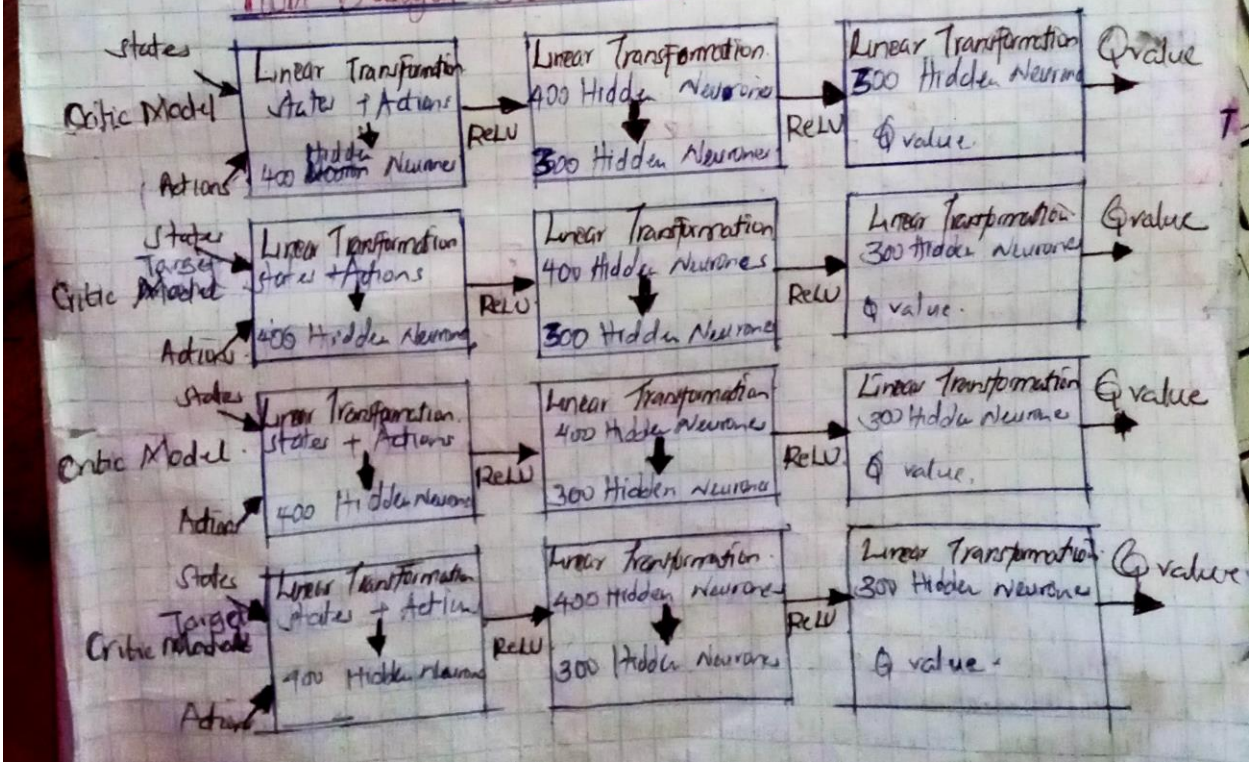
Step 1: We initialize the Experience Replay memory

```
class ReplayBuffer(object):
    def __init__(self, max_size = 1e6):
        self.storage = []
        self.max_size = max_size
        self.ptr = 0
    def add(self, transition):
        if len(self.storage) == self.max_size:
            self.storage[int(self.ptr)] = transition
            self.ptr = (self.ptr + 1) % self.max_size
        else:
            self.storage.append(transition)
    def sample(self, batch_size):
        ind = np.random.randint(0, len(self.storage), size = batch_size)
        batch_states, batch_next_states, batch_actions, batch_rewards, batch_dones = [], [], [], [], []
        for i in ind:
            state, next_state, action, reward, done = self.storage[i]
            batch_states.append(np.array(state, copy = False))
            batch_next_states.append(np.array(next_state, copy = False))
            batch_actions.append(np.array(action, copy = False))
            batch_rewards.append(np.array(reward, copy = False))
            batch_dones.append(np.array(done, copy = False))
        return np.array(batch_states), np.array(batch_next_states), np.array(batch_actions), np.array(batch_rewards).reshape(-1, 1), np.array(batch_dones).reshape(-1, 1)
```

Step 2: We build one neural network for the Actor model and one neural network for the Actor target.

```
class Actor(nn.Module):                                   # forward flow
    def __init__(self, state_dim, action_dim, max_action):   # state_dim: dimension of input; action_dim: dimension of output; max_action: clip action in certain range
        super(Actor, self).__init__()                     # activate inheritance
        self.layer_1 = nn.Linear(state_dim, 400)          # full connection
        self.layer_2 = nn.Linear(400, 300)
        self.layer_3 = nn.Linear(300, action_dim)
        self.max_action = max_action
    def forward(self, x):                                 # input state
        x = F.relu(self.layer_1(x))
        x = F.relu(self.layer_2(x))
        x = self.max_action * torch.tanh(self.layer_3(x))  # return final output signal
        return x
```

Twin Delayed DDPG (TD3)



We build two neural networks for the two Critic models and two neural networks for the two Critic targets.

```
class Critic(nn.Module):

    def __init__(self, state_dim, action_dim):
        super(Critic, self).__init__()
        #Defining the First Critic Neural Network
        self.layer_1 = nn.Linear(state_dim + action_dim, 400)
        self.layer_2 = nn.Linear(400, 300)
        self.layer_3 = nn.Linear(300, 1)   # Output layer
        # Defining the Second Critic neural network.
        self.layer_4 = nn.Linear(state_dim + action_dim, 400)
        self.layer_5 = nn.Linear(400, 300)
        self.layer_6 = nn.Linear(300, 1)   # Output layer

    def forward(self, x, u):   # Propagate signal from input layer to output layer
        xu = torch.cat([x, u], 1)   # dim=1 specifically: we are going to concatenate action-states
        # Forward-Propagation on the First Critic Neural Network
        x1 = F.relu(self.layer_1(xu))
        x1 = F.relu(self.layer_2(x1))
        x1 = self.layer_3(x1)
        # Forward-Propagation on the Second Critic neural network.
        x2 = F.relu(self.layer_4(xu))
        x2 = F.relu(self.layer_5(x2))
        x2 = self.layer_6(x2)
        return x1, x2

    def Q1(self, x, u):
        xu = torch.cat([x, u], 1)
        x1 = F.relu(self.layer_1(xu))
        x1 = F.relu(self.layer_2(x1))
        x1 = self.layer_3(x1)
        return x1
```

We want to find first output of the First Critic model

Selecting the device.

Building the whole Training Process into a class

```
class TD3(object):
    def __init__(self, state_dim, action_dim, max_action):
        self.actor = Actor(state_dim, action_dim, max_action).to(device)
        self.actor_target = Actor(state_dim, action_dim, max_action).to(device)
        self.actor_target.load_state_dict(self.actor.state_dict())
        self.actor_optimizer = torch.optim.Adam(self.actor.parameters())
        self.critic = Critic(state_dim, action_dim).to(device)
        self.critic_target = Critic(state_dim, action_dim).to(device)
        self.critic_target.load_state_dict(self.critic.state_dict())
        self.critic_optimizer = torch.optim.Adam(self.critic.parameters())
        self.max_action = max_action

    def select_action(self, state):
        state = torch.Tensor(state.reshape(1, -1)).to(device)
        return self.actor(state).cpu().data.numpy().flatten()

    def train(self, replay_buffer, iterations, batch_size=100, discount=0.99, tau=0.005,
              policy_noise=0.2, noise_clip=0.5, policy_freq=2):
        for it in range(iterations):
            # Step 4: We sample a batch of transitions (s, s', a, r) from the memory
            batch_states, batch_next_states, batch_actions, batch_rewards, batch_dones = replay_buffer.sample(batch_size)
            state = torch.Tensor(batch_states).to(device)
            next_state = torch.Tensor(batch_next_states).to(device)
            action = torch.Tensor(batch_actions).to(device)
            reward = torch.Tensor(batch_rewards).to(device)
            done = torch.Tensor(batch_dones).to(device)
            # Step 5: From the next state s', the Actor target plays the next action a'
            next_action = self.actor_target(next_state)
```

#step6: We add Gaussian noise to the next action a' and we clamp it in a range of values supported by the environment.

```
noise = torch.Tensor(batch_actions).data.normal_(0, policy_noise).to(device)
noise = noise.clamp(-noise_clip, noise_clip)
next_action = (next_action + noise).clamp(-self.max_action, self.max_action)
```

#step7: The two Critic targets take each the couple (s', a') as input and return two Q values $Q_{t1}(s', a')$ and $Q_{t2}(s', a')$ as outputs

```
target_Q1, target_Q2 = self.critic_target(next_state, next_action)
```

#step8: We keep the minimum of these two Q-values: $\min(Q_{t1}, Q_{t2})$

```
target_Q = torch.min(target_Q1, target_Q2)
```

#step9: We get the final target of the two Critic models, which is: $Q_t = r + \gamma * \min(Q_{t1}, Q_{t2})$, where γ is the discount factor.

```
target_Q = reward + ((1 - done) * discount * target_Q).detach()
```

#step10: The two Critic models take each the couple (s, a) as input and return two Q-values $Q_1(s,a)$ and $Q_2(s,a)$ as outputs

```
current_Q1, current_Q2 = self.critic(state, action)
```

#step11: We compute the loss coming from the two Critic models:

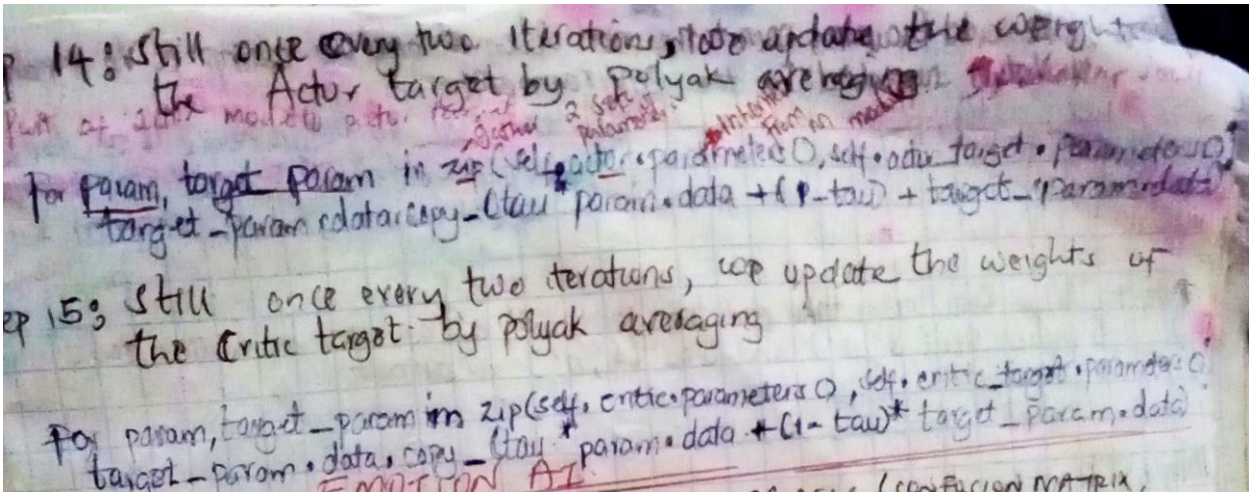
```
Critic Loss = MSE_Loss(Q1(s,a), Qt) + MSE_Loss(Q2(s,a), Qt)
critic_loss = F.mse_loss(current_Q1, target_Q) + F.mse_loss(current_Q2, target_Q)
```

#step12: We backpropagate this Critic loss and update the parameters of the two Critic models with a SGD optimizer

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

#step13: Once every two iterations, we update our Actor model by performing gradient ascent on the output of the first Critic model.

```
if it % policy_freq == 0:
    actor_loss = -self.critic.Q1(state, self.actor(state)).mean()
    self.actor_optimizer.zero_grad()
    actor_loss.backward()
    self.actor_optimizer.step()
```



p 14: still once every two iterations, we update the weights of the Actor target by polyak averaging

for param, target_param in zip(self.actor.parameters(), self.actor_target.parameters()):
    target_param.data.copy_(tau * param.data + (1-tau) + target_param.data)

ep 15: still once every two iterations, we update the weights of the Critic target by polyak averaging

for param, target_param in zip(self.critic.parameters(), self.critic_target.parameters()):
    target_param.data.copy_(tau * param.data + (1- tau)* target_param.data)