```python
import torch
import torch.optim as optim
import torch.nn.functional as F
import torch.nn
import gymnasium as gym
from replay_buffer import ReplayBufferDDPG
import wandb
import random
import numpy as np
import os
import time
import model
from utils import *
import tqdm



class OU_Noise:
    def __init__(self,action_space:int,action_range:list[np.ndarray[float]],
                mu:float = 0.0, theta:float = 0.15, sigma:float = 0.2,seed:int = 42):
        """Initialize the OU noise

        Args:
            action_space (int): The size of the action space action_range
            (list[np.ndarray[float]]): The range of the action space, the first
                element is the lower bound and the second element is the upper bound
            mu (float, optional): average of the noise. Defaults to 0.0.
            theta (float, optional): the speed of mean reversion. Defaults to 0.15.
            sigma (float, optional): the volatility of the noise. Defaults to 0.2.
            seed (int, optional): the seed for the random number generator. Defaults to 42.
        """
        self.action_space = action_space
        self.mu = mu
        self.theta = theta
        self.sigma = sigma
        self.state = np.ones(self.action_space) * self.mu
        self.action_range = action_range
        self.seed = seed
        np.random.seed(self.seed)

    def reset(self,sigma:float = 0.2):
        """Reset the noise

        Args:
            sigma (float, optional): you can change the sigma of the noise. Defaults to 0.2.
        """
        # ========== YOUR CODE HERE ==========
        # TODO:
        # hint look at line 36
        # ====================================
        self.sigma = sigma
        self.state = np.ones(self.action_space) * self.mu

        # ========== YOUR CODE ENDS ==========

    def _sample(self):
        """sample the noise per the discretized Ornstein-Uhlenbeck process detailed in the
notebook"""
        # ========== YOUR CODE HERE ==========
        dx = self.theta*(self.mu - self.state) + self.sigma*np.random.randn(self.action_space)
        self.state = self.state + dx
        return self.state

        # ========== YOUR CODE ENDS ==========

    def noise(self,action:np.ndarray[float]):
        """Add the noise to the action
```

```python
        Args:
            action (np.ndarray[float]): the action to add the noise to

        Returns:
            noised_action (np.ndarray[float]): the noised action, clipped to the action range
        """
        # ========== YOUR CODE HERE ==========
        # TODO:
        # you can use the _sample method to get the noise
        # ====================================
        noised_action = action + self._sample()

        clipped = np.clip(noised_action, self.action_range[0], self.action_range[1])
        # clipped =  np.min(np.max(noised_action, -1), 1)
        # print(action, noised_action, clipped)
        return clipped

        # ========== YOUR CODE ENDS ==========



class DDPG:
    def __init__(self, env:gym.Env,
                 actor_model:model.Actor,
                 critic_model:model.Critic,
                 actor_kwargs = {},
                 critic_kwargs = {},
                 actor_lr:float = 0.0001,
                 critic_lr:float = 0.001,
                 gamma:float = 0.99,
                 tau:float = 0.001,
                 buffer_size:int = 10**6,
                 batch_size:int = 64,
                 loss_fn:str = 'mse_loss',
                 use_wandb:bool = False,device:str = 'cpu',
                 seed:int = 42,
                 save_path:str = None):
        """Initialize the DDPG agent

        Args:
            env (_type_): The environment to train on
            actor_model (_type_): the class for the actor model
            critic_model (_type_): the class for the critic model
            actor_kwargs (dict, optional): Additional actor_kwargs . Defaults to {}.
            critic_kwargs (dict, optional): Additional critic_kwargs. Defaults to {}.
            actor_lr (float, optional): The learning rate for the optimizer. Defaults to
0.0001.
            critic_lr (float, optional): The learning rate for the critic optimizer. Defaults
to 0.001.
            gamma (float, optional): discount factor. Defaults to 0.99.
            tau (float, optional): soft update parameter for the target networks. Defaults to
0.001.
            buffer_size (int, optional): the size of the replay buffer. Defaults to 10^6
            batch_size (int, optional): the batch size for training. Defaults to 64.
            loss_fn (str, optional): name of the loss function to use. Defaults to 'mse_loss'.
            use_wandb (bool, optional): whether to use wandb. Defaults to False.
            device (str, optional): which device to use. Defaults to 'cpu'.
            seed (int, optional): seed for reproducibility. Defaults to 42.
            save_path (str, optional): path to save the model. Defaults to None.

        """
        self.env = env
        self._set_seed(seed)
        self.observation_space = self.env.observation_space.shape
        self.actor = actor_model(self.observation_space, self.env.action_space.shape[0]
                                 , **actor_kwargs).to(device)
```

```python
        self.critic = critic_model(self.observation_space, self.env.action_space.shape[0]
                                    , **critic_kwargs).to(device)

        self.target_actor = actor_model(self.observation_space, self.env.action_space.shape[0]
                                         , **actor_kwargs).to(device)
        self.target_critic = critic_model(self.observation_space,
self.env.action_space.shape[0]
                                          , **critic_kwargs).to(device)

        self.OU_noise = OU_Noise(self.env.action_space.shape[0],
                                 [self.env.action_space.low,self.env.action_space.high])

        #sync the target networks with the main networks
        self.target_actor.load_state_dict(self.actor.state_dict())
        self.target_critic.load_state_dict(self.critic.state_dict())

        self.actor_optimizer = optim.Adam(self.actor.parameters(), lr = actor_lr)
        self.critic_optimizer = optim.Adam(self.critic.parameters(), lr = critic_lr)
        self.gamma = gamma

        self.replay_buffer = ReplayBufferDDPG(buffer_size)
        self.batch_size = batch_size
        self.tau = tau
        self.device = device
        self.save_path = save_path if save_path is not None else "./"

        #set the loss function
        if loss_fn == 'smooth_l1_loss':
            self.loss_fn = F.smooth_l1_loss
        elif loss_fn == 'mse_loss':
            self.loss_fn = F.mse_loss
        else:
            raise ValueError('loss_fn must be either smooth_l1_loss or mse_loss')

        self.wandb = use_wandb
        if self.wandb:
            wandb.init(project = 'double_pendulum_ddpg')
            #log the hyperparameters
            wandb.config.update({
                'actor_lr': actor_lr,
                'critic_lr': critic_lr,
                'gamma': gamma,
                'buffer_size': buffer_size,
                'batch_size': batch_size,
                'loss_fn': loss_fn,
                'tau': tau,
                'device': device,
                'seed': seed,
                'save_path': save_path
            })

    def _set_seed(self, seed:int):
        random.seed(seed)
        np.random.seed(seed)
        self.seed = seed
        torch.manual_seed(seed)
        torch.cuda.manual_seed(seed)
        torch.backends.cudnn.deterministic = True
        gym.utils.seeding.np_random(seed)


    def play_episode(self,sigma:float = 0,return_frames:bool = False,seed:int = None,env =
None):
        """Play an episode of the environment

        Args:
            sigma (float, optional): the sigma for the OU noise. Defaults to 0.
```

```python
            return_frames (bool, optional): whether to return the frames. Defaults to False.
            seed (int, optional): the seed for the environment. Defaults to None.
        """
        if env is None:
            env = self.env
        if seed is not None:
            state,_ = env.reset(seed = seed)
        else:
            state,_ = env.reset()

        if sigma>0:
            self.OU_noise.reset(sigma)
        done = False
        total_reward = 0
        if return_frames:
            frames = []
        with torch.no_grad():
            while not done:
                action = 
self.actor(torch.tensor(state).float().to(self.device).unsqueeze(0)).cpu().numpy()[0]
                if sigma>0:
                    action = self.OU_noise.noise(action)
                next_state, reward, terminated, truncated, _ = env.step(action)
                total_reward += reward
                done = terminated or truncated
                if return_frames:
                    frames.append(env.render())
                state = next_state
        if return_frames:
            return total_reward, frames
        else:
            return total_reward



    def _train_one_batch(self,batch_size):
        """train the agent on a single batch"""
        # ========== YOUR CODE HERE ==========
        # TODO:
        # sample the batch
        # compute the target Q value
        # compute the current Q value
        # compute the critic loss
        # optimize the critic
        # compute the actor loss
        # optimize the actor
        # update the model
        # return the losses
        # ====================================
        # raise NotImplementedError
        # sample the batch

        # compute the target Q value

        # compute the current Q value

        # compute the critic loss

        # optimize the critic
        # compute the actor loss
        # optimize the actor
        # update the model
        # return the losses

        if len(self.replay_buffer) < batch_size:
            return 0.0, 0.0  # skip training if not enough samples
```

```
        # Sample a batch
        state_batch, action_batch, reward_batch, next_state_batch, done_batch =
self.replay_buffer.sample(batch_size)

        state_batch = torch.FloatTensor(state_batch).to(self.device)
        action_batch = torch.FloatTensor(action_batch).to(self.device)
        reward_batch = torch.FloatTensor(reward_batch).unsqueeze(1).to(self.device)
        next_state_batch = torch.FloatTensor(next_state_batch).to(self.device)
        # done_batch = torch.FloatTensor(done_batch).unsqueeze(1).to(self.device)
        # done_batch =
torch.FloatTensor(done_batch.astype(np.float32)).unsqueeze(1).to(self.device)
        done_batch = done_batch.to(torch.float32).unsqueeze(1).to(self.device)



        with torch.no_grad():
            next_actions = self.target_actor(next_state_batch)
            target_Q = self.target_critic(next_state_batch, next_actions)
            target_Q = reward_batch + self.gamma * target_Q * (1 - done_batch)

        current_Q = self.critic(state_batch, action_batch)
        critic_loss = self.loss_fn(current_Q, target_Q)

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

        actor_loss = -self.critic(state_batch, self.actor(state_batch)).mean()

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

        self._update_model()

        return critic_loss.item(), actor_loss.item()

        # ========== YOUR CODE ENDS ==========

    def _update_model(self):
        # ========== YOUR CODE HERE ==========
        for target_param, param in zip(self.target_actor.parameters(),
self.actor.parameters()):
            target_param.data.copy_(self.tau * param.data + (1.0 - self.tau) *
target_param.data)

        for target_param, param in zip(self.target_critic.parameters(),
self.critic.parameters()):
            target_param.data.copy_(self.tau * param.data + (1.0 - self.tau) *
target_param.data)
        # ========== YOUR CODE ENDS ==========

    def train(self, episodes:int, val_freq:int, val_episodes:int, test_episodes:int,
save_every:int,
              train_every:int = 1):
        """Train the agent

        Args:
            episodes (int): the number of episodes to train for
            val_freq (int): the frequency of validation
            val_episodes (int): the number of episodes to validate for
            test_episodes (int): the number of episodes to test for
            save_every (int): the frequency of saving the model
            train_every (int, optional): the frequency of training per enviroment interaction.
Defaults to 1.
        """
        os.makedirs(self.save_path, exist_ok=True)
```

```
        best_val_mean = -np.inf
        for i in range(episodes):
            start_time = time.time()
            # print(sigma)
            state,_ = self.env.reset()
            self.OU_noise.reset()
            done = False
            total_reward = 0
            Q_loss_total = 0
            actor_loss_total = 0
            l = 0

            while not done:
                # ========== YOUR CODE HERE ==========
                # TODO:
                # get the action
                # add the noise, hint you can do this by calling the noise method of the OU 
noise
                # get the transition
                # store the transition
                # update the state
                # if the replay buffer is large enough, and it is time to train the model
                # and update the total Q and actor loss
                # ====================================
                state_tensor = torch.tensor(state, dtype=torch.float32, 
device=self.device).unsqueeze(0)
                action = self.actor(state_tensor).detach().cpu().numpy()[0]
                action = self.OU_noise.noise(action)

                next_state, reward, terminated, truncated, _ = self.env.step(action)
                done = terminated or truncated

                self.replay_buffer.add(state, action, reward, next_state, done)
                state = next_state
                total_reward += reward

                if len(self.replay_buffer) >= self.batch_size and l % train_every == 0:
                    Q_loss, actor_loss = self._train_one_batch(self.batch_size)
                    Q_loss_total += Q_loss
                    actor_loss_total += actor_loss

                l += 1

                # ========== YOUR CODE ENDS ==========

            if self.wandb:
                wandb.log({
                    'total_reward': total_reward,
                    'Q_loss': Q_loss_total,
                    'actor_loss': actor_loss_total
                })
            print(f"Episode {i}: Time: {time.time()-start_time}, Total Reward: {total_reward}, 
Q Loss: {Q_loss_total}, Actor Loss: {actor_loss_total}")


            if i % val_freq == val_freq-1:
                val_mean, val_std = self.validate(val_episodes)

                if self.wandb:
                    wandb.log({
                        'val_mean': val_mean,
                        'val_std': val_std
                    })
                print(f"Validation Mean: {val_mean}, Validation Std: {val_std}")
                if val_mean > best_val_mean:
                    best_val_mean = val_mean
                    self.save_model('best')
```

```python
            # print("save_every",save_every,i,save_every-1)
            if i % save_every == save_every-1:
                self.save_model(i)
                print("saving model")


        self.save_model('final')
        self.load_model('best')

        test_mean, test_std = self.validate(test_episodes)
        print(f"Test Mean: {test_mean}, Test Std: {test_std}")
        if self.wandb:
            wandb.log({
                'test_mean': test_mean,
                'test_std': test_std
            })




    def validate(self, episodes:int):
        rewards = []
        for _ in range(episodes):
            rewards.append(self.play_episode())
        mean_reward = np.mean(rewards)
        std_reward = np.std(rewards)
        return mean_reward, std_reward

    def save_model(self, name:str):
        actor_path = os.path.join(self.save_path, f"actor_{name}.pt")
        actor_model_path = os.path.join(self.save_path, f"actor_target_{name}.pt")
        torch.save(self.actor.state_dict(), actor_path)
        torch.save(self.target_actor.state_dict(), actor_model_path)

        critic_path = os.path.join(self.save_path, f"critic_{name}.pt")
        critic_model_path = os.path.join(self.save_path, f"critic_target_{name}.pt")
        torch.save(self.critic.state_dict(), critic_path)
        torch.save(self.target_critic.state_dict(), critic_model_path)

    def load_model(self, name:str):
        actor_path = os.path.join(self.save_path, f"actor_{name}.pt")
        actor_model_path = os.path.join(self.save_path, f"actor_target_{name}.pt")
        self.actor.load_state_dict(torch.load(actor_path))
        self.target_actor.load_state_dict(torch.load(actor_model_path))

        critic_path = os.path.join(self.save_path, f"critic_{name}.pt")
        critic_model_path = os.path.join(self.save_path, f"critic_target_{name}.pt")
        self.critic.load_state_dict(torch.load(critic_path))
        self.target_critic.load_state_dict(torch.load(critic_model_path))
```