```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 ReplayBufferDQN
import wandb
import random
import numpy as np
import os
import time
from utils import exponential_decay
import typing

# TODO: change the logging here if you don't like wandb

class DQN:
    def __init__(self, env:typing.Union[gym.Env,gym.Wrapper],
                 #model params
                 model:torch.nn.Module,
                 model_kwargs:dict = {},
                 #overall hyperparams
                 lr:float = 0.001, gamma:float = 0.99,
                 buffer_size:int = 10000, batch_size:int = 32,
                 loss_fn:str = 'mse_loss',
                 use_wandb:bool = False,
                 device:str = 'cpu',
                 seed:int = 42,
                 epsilon_scheduler = exponential_decay(1,700,0.1),
                 save_path:str = None):
        """Initializes the DQN algorithm

        Args:
            env (gym.Env|gym.Wrapper): the environment to train on
            model (torch.nn.Module): the model to train
            model_kwargs (dict, optional): the keyword arguments to pass to the model. Defaults
to {}.
            lr (float, optional): the learning rate to use in the optimizer. Defaults to 0.001.
            gamma (float, optional): discount factor. Defaults to 0.99.
            buffer_size (int, optional): the size of the replay buffer. Defaults to 10000.
            batch_size (int, optional): the batch size. Defaults to 32.
            loss_fn (str, optional): the name of the loss function to use. Defualts to
'mse_loss'.
            use_wandb (bool, optional): _description_. Defaults to False.
            device (str, optional): _description_. Defaults to 'cpu'.
            seed (int, optional): the seed to use for reproducibility. Defaults to 42.
            epsilon_scheduler ([type], optional): the epsilon scheduler to use, must have a
__call__ method that returns a float between 0 and 1
            save_path (str, optional): _description_. Defaults to None.

        Raises:
            ValueError: _description_
        """

        self.env = env
        self._set_seed(seed)

        self.observation_space = self.env.observation_space.shape
        self.model = model(
            self.observation_space,
            self.env.action_space.n, **model_kwargs
            ).to(device)
        self.model.train()
        self.optimizer = optim.Adam(self.model.parameters(), lr = lr)
        self.gamma = gamma

        self.replay_buffer = ReplayBufferDQN(buffer_size)
```

```python
        self.batch_size = batch_size
        self.i_update = 0
        self.device = device
        self.epsilon_decay = epsilon_scheduler
        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 = 'racing-car-dqn')
            #log the hyperparameters
            wandb.config.update({
                'lr': lr,
                'gamma': gamma,
                'buffer_size': buffer_size,
                'batch_size': batch_size,
                'loss_fn': loss_fn,
                'device': device,
                'seed': seed,
                'save_path': save_path
            })

    def train(self, n_episodes:int = 1000,validate_every:int = 100, n_validation_episodes:int
= 10, n_test_episodes:int = 10,save_every:int = 100):
        os.makedirs(self.save_path, exist_ok = True)
        best_val_reward = -np.inf

        for episode in range(n_episodes):
            state,_ = self.env.reset()
            done = False
            truncated = False
            total_reward = 0
            i = 0
            loss = 0
            start_time = time.time()
            epsilon = self.epsilon_decay()
            while (not done) and (not truncated):
                action = self._sample_action(state, epsilon)
                next_state, reward, done, truncated, _ = self.env.step(action)
                self.replay_buffer.add(state, action, reward, next_state, done)
                total_reward += reward
                state = next_state

                not_warm_starting,l = self._optimize_model()
                if not_warm_starting:
                    loss += l
                    epsilon = self.epsilon_decay()
                    i += 1
            if i == 0:
                avg_loss = loss / (i+1)
            else:
                avg_loss = loss/i
            if self.wandb:
                wandb.log({'total_reward': total_reward, 'loss': avg_loss})
            print(f"Episode: {episode}: Time: {time.time() - start_time} Total Reward:
{total_reward} Avg_Loss: {avg_loss}")
            if episode % validate_every == validate_every - 1:
                mean_reward, std_reward = self.validate(n_validation_episodes)
                if self.wandb:
                    wandb.log({'mean_reward': mean_reward, 'std_reward': std_reward})
```

```
                print("Validation Mean Reward: {} Validation Std Reward:
{}".format(mean_reward, std_reward))
                if mean_reward > best_val_reward:
                    best_val_reward = mean_reward
                    self._save('best')

            if episode % save_every == save_every - 1:
                self._save(str(episode))

        self._save('final')
        self.load_model('best')
        mean_reward, std_reward = self.validate(n_test_episodes)
        if self.wandb:
            wandb.log({'mean_test_reward': mean_reward, 'std_test_reward': std_reward})
        print("Test Mean Reward: {} Test Std Reward: {}".format(mean_reward, std_reward))

    def _optimize_model(self):
        """
        Performs one optimization step on the DQN.

        Returns:
            bool: whether we have enough samples to optimize the model, which we define as
having at least 10*batch_size samples
            float: the loss, if we do not have enough samples, we return 0
        """
        # ========== YOUR CODE HERE ==========
        # TODO: some hints to get you started:
        # 1. check if the replay buffer has enough samples
        # 2. sample a minibatch
        # 3. No-op
        # 4. compute current Q: q_values = ...
        # 5. compute target Q
        # 6. compute loss between current Q and target Q
        # 7. backprop
        # ====================================

        # step 1
        if len(self.replay_buffer) < (10 * self.batch_size):
            return False, 0.0

        # step 2
        states, actions, rewards, next_states, dones =
self.replay_buffer.sample(self.batch_size, device=self.device)

        # step 3
        # No-op

        # step 4
        # Select Q-values for the taken actions
        q_values = self.model(states)  # shape: (batch_size, num_actions)

        # step 5: target_q
        with torch.no_grad():
            next_q_values = self.model(next_states)
            max_next_q_values = next_q_values.max(dim=1)[0]
            if dones.all():
                targets = rewards
            else:
                targets = rewards + self.gamma * max_next_q_values

        # step 6
        q_values_given_action = q_values.gather(1, actions.unsqueeze(1)).squeeze(1)  # shape:
(batch_size,)
        loss = self.loss_fn(q_values_given_action, targets)

        # step 7
        self.optimizer.zero_grad()
```

```
        loss.backward()
        self.optimizer.step()

        return True, loss.item()

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

    def _sample_action(self, state:np.ndarray
                       , epsilon:float = 0.1)->int:
        """
        Samples an action from the model

        Args:
            state (np.ndarray): the state, of shape [n_c,h,w]
            epsilon (float, optional): the epsilon for epsilon greedy. Defaults to 0.1.

        Returns:
            int: the index of the action to take
        """
        # ========== YOUR CODE HERE ==========
        # TODO:
        # Epsilon-greedy action selection:
        #   - if probability epsilon: random action
        #   - else: greedy action
        # ====================================

        if random.random() < epsilon:
            index = self.env.action_space.sample()
        else:
            state_tensor = torch.tensor(state, dtype=torch.float32,
device=self.device).unsqueeze(0)
            with torch.no_grad():
                q_values = self.model(state_tensor)
                index = q_values.argmax(dim=1).item()

        # ========== YOUR CODE ENDS ==========
        return index

    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 _validate_once(self):
        state,_ = self.env.reset()
        done = False
        truncated = False
        total_reward = 0
        i = 0
        # epsilon = self.epsilon_decay()
        while (not done) and (not truncated):
            action = self._sample_action(state, 0)
            # out = self.env.step(action)
            next_state, reward, done, truncated, _ = self.env.step(action)
            # next_state = np.array(state_buffer[-self.n_frames:])
            total_reward += reward
            state = next_state
        return total_reward


    def validate(self, n_episodes:int = 10):
        # self.model.eval()
        rewards_per_episode = []
```

```
        for _ in range(n_episodes):
            rewards_per_episode.append(self._validate_once())
        # self.model.train()
        return np.mean(rewards_per_episode), np.std(rewards_per_episode)

    def load_model(self, suffix:str = ''):
        self.model.load_state_dict(torch.load(os.path.join(self.save_path,
f'model_{suffix}.pt')))

    def _save(self,suffix:str = ''):
        torch.save(self.model.state_dict(), os.path.join(self.save_path,
f'model_{suffix}.pt'))

    def play_episode(self,epsilon:float  = 0, return_frames:bool = True,seed:int = None):
        """Plays an episode of the environment

        Args:
            epsilon (float, optional): the epsilon for epsilon greedy. Defaults to 0.
            return_frames (bool, optional): whether we should return frames. Defaults to True.
            seed (int, optional): the seed for the enviroment. Defaults to None.

        Returns:
            if return frames is True, returns the total reward and the frames
            if return frames is False, returns the total reward
        """
        if seed is not None:
            state,_ = self.env.reset(seed = seed)
        else:
            state,_ = self.env.reset()

        done = False
        total_reward = 0
        if return_frames:
            frames = []

        with torch.no_grad():
            while not done:
                action = self._sample_action(state, epsilon)
                next_state, reward, terminated, truncated, _ = self.env.step(action)
                total_reward += reward
                done = terminated or truncated
                if return_frames:
                    frames.append(self.env.render())
                state = next_state

        if return_frames:
            return total_reward, frames

        return total_reward



class HardUpdateDQN(DQN):

    def __init__(self,env,model,model_kwargs:dict = {},
                 update_freq:int = 5,*args,**kwargs):
        super().__init__(env,model,model_kwargs, *args,**kwargs)
        # ========== YOUR CODE HERE ==========
        # TODO:
        # fill in the initialization and synchronization of the target model weights
        # ====================================

        # Initialize target network with same architecture
        self.target_model = model(
            self.observation_space,
            self.env.action_space.n,
            **model_kwargs
```

```
        ).to(self.device)

        # Copy initial weights from model to target_model
        self.target_model.load_state_dict(self.model.state_dict())

        # Set target network to eval mode (not training)
        self.target_model.eval()

        # Store update frequency for hard target updates
        self.update_freq = update_freq

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

    def _optimize_model(self):
        """Optimizes the model

        Returns:
            bool: whether we have enough samples to optimize the model, which we define as
having at least 10*batch_size samples
            float: the loss, if we do not have enough samples, we return 0
        """
        # ========== YOUR CODE HERE ==========
        # TODO:
        # hint: you can copy over most of the code from the parent class
        # and only change one line
        # ====================================

        if len(self.replay_buffer) < (10 * self.batch_size):
            return False, 0.0

        # Sample batch
        states, actions, rewards, next_states, dones =
self.replay_buffer.sample(self.batch_size, device=self.device)

        # Compute Q(s, a) from current model
        q_values = self.model(states)
        q_values_given_action = q_values.gather(1, actions.unsqueeze(1)).squeeze(1)

        # Compute target Q using target network
        with torch.no_grad():
            next_q_values = self.target_model(next_states)   # changed from self.model
            max_next_q_values = next_q_values.max(dim=1)[0]
            targets = rewards + self.gamma * max_next_q_values * (~dones)

        # Compute loss and update
        loss = self.loss_fn(q_values_given_action, targets)
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

        # Update target model if needed
        self._update_model()

        return True, loss.item()

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

    def _update_model(self):
        self.i_update += 1
        if self.i_update % self.update_freq == 0:
            self.target_model.load_state_dict(self.model.state_dict())

    def _save(self,suffix:str = ''):
        torch.save(self.model.state_dict(), os.path.join(self.save_path,
f'model_{suffix}.pt'))
        torch.save(self.target_model.state_dict(), os.path.join(self.save_path,
f'target_model_{suffix}.pt'))
```

```python
    def load_model(self, suffix:str = ''):
        self.model.load_state_dict(torch.load(os.path.join(self.save_path,
f'model_{suffix}.pt')))
        self.target_model.load_state_dict(torch.load(os.path.join(self.save_path,
f'target_model_{suffix}.pt')))



class SoftUpdateDQN(HardUpdateDQN):
    def __init__(self,env,model,model_kwargs:dict = {},
                 tau:float = 0.01,*args,**kwargs):
        super().__init__(env,model,model_kwargs,*args,**kwargs)
        self.tau = tau

    def _update_model(self):
        """
        Soft updates the target model
        """
        # ========== YOUR CODE HERE ==========
        # TODO
        # ====================================

        for target_param, model_param in zip(self.target_model.parameters(),
self.model.parameters()):
            target_param.data.copy_(self.tau * model_param.data + (1.0 - self.tau) *
target_param.data)

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