```python
import torch
import torch.nn as nn
import torch.nn.functional as F
from ResUNet import ConditionalUnet
from utils import *

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

class ConditionalDDPM(nn.Module):
    def __init__(self, dmconfig):
        super().__init__()
        self.dmconfig = dmconfig
        self.loss_fn = nn.MSELoss()
        self.network = ConditionalUnet(1, self.dmconfig.num_feat, self.dmconfig.num_classes)

    def scheduler(self, t_s):
        beta_1, beta_T, T = self.dmconfig.beta_1, self.dmconfig.beta_T, self.dmconfig.T
        # ==================================================== #
        # YOUR CODE HERE:
        #   Inputs:
        #       t_s: the input time steps, with shape (B,1).
        #   Outputs:
        #       one dictionary containing the variance schedule
        #       $\beta_t$ along with other potentially useful constants.

        # Make sure t_s is shape (B,1)
        if t_s.dim() == 0:
            t_s = t_s.view(1,1)
        elif t_s.dim() == 1:
            t_s = t_s.unsqueeze(-1)

        B = t_s.size(0)   # number of examples in the batch

        # Build the scheduler for the DDPM process
        device = t_s.device
        betas = torch.linspace(beta_1, beta_T, T, device=device)
        alphas = 1.0 - betas
        alpha_bars = torch.cumprod(alphas, dim=0)

        # Zero-base indices
        t = t_s.squeeze(-1).long() - 1
        t = t.clamp(0, T-1)

        # Based on spec
        beta_t = betas[t].view(B,1,1,1)
        sqrt_beta_t = torch.sqrt(beta_t)
        alpha_t = alphas[t].view(B,1,1,1)
        oneover_sqrt_alpha = 1.0 / torch.sqrt(alpha_t)
        alpha_t_bar = alpha_bars[t].view(B,1,1,1)
        sqrt_alpha_bar = torch.sqrt(alpha_t_bar)
        sqrt_oneminus_alpha_bar = torch.sqrt(1.0 - alpha_t_bar)

        # ==================================================== #
        return {
            'beta_t': beta_t,
            'sqrt_beta_t': sqrt_beta_t,
            'alpha_t': alpha_t,
            'sqrt_alpha_bar': sqrt_alpha_bar,
            'oneover_sqrt_alpha': oneover_sqrt_alpha,
            'alpha_t_bar': alpha_t_bar,
            'sqrt_oneminus_alpha_bar': sqrt_oneminus_alpha_bar
        }

    def forward(self, images, conditions):
        T = self.dmconfig.T
        noise_loss = None
        # ==================================================== #
```

```python
        # YOUR CODE HERE:
        #   Complete the training forward process based on the
        #   given training algorithm.
        #   Inputs:
        #       images: real images from the dataset, with size (B,1,28,28).
        #       conditions: condition labels, with size (B). You should
        #                   convert it to one-hot encoded labels with size (B,10)
        #                   before making it as the input of the denoising network.
        #   Outputs:
        #       noise_loss: loss computed by the self.loss_fn function  .

        B, _, H, W = images.shape
        device = images.device

        # One-hot encode the labels
        c = F.one_hot(conditions, num_classes=self.dmconfig.num_classes).float()   # (B,
num_classes)

        # Classifier-free masking
        p_uncond = getattr(self.dmconfig, 'p_uncond', 0.0)
        if p_uncond > 0:
            mask = (torch.randn(B, device=device) < p_uncond)
            c[mask] = 0.0

        # Sample for each batch element
        t = torch.randint(1, T+1, (B,1), device=device)
        eps = torch.randn_like(images)

        # Compute the noisy input
        sched = self.scheduler(t)

        x_t = sched['sqrt_alpha_bar'] * images      \
            + sched['sqrt_oneminus_alpha_bar'] * eps   # broadcastâ(B,1,H,W)

        # Predict the noise
        noise_pred = self.network(x_t, t.squeeze(-1), c)

        # MSE loss
        noise_loss = self.loss_fn(noise_pred, eps)

        # ==================================================== #

        return noise_loss

    def sample(self, conditions, omega):
        T = self.dmconfig.T
        X_t = None
        # ==================================================== #
        # YOUR CODE HERE:
        #   Complete the training forward process based on the
        #   given sampling algorithm.
        #   Inputs:
        #       conditions: condition labels, with size (B). You should
        #                   convert it to one-hot encoded labels with size (B,10)
        #                   before making it as the input of the denoising network.
        #       omega: conditional guidance weight.
        #   Outputs:
        #       generated_images

        B = conditions.size(0)
        device = conditions.device

        # One-hot encode labels
        c = F.one_hot(conditions, num_classes=self.dmconfig.num_classes).float()

        # Gaussian noise x_T
        X_t = torch.randn(B, 1, 28, 28, device=device)
```

```
        # eval mode, no grad
        self.network.eval()
        with torch.no_grad():
            for t in range(T, 0, -1):
                # Build timeâstep tensors
                t_vec = torch.full((B,),   t, device=device, dtype=torch.long)
                t_batch = t_vec.unsqueeze(-1)

                # Query scheduler
                sched = self.scheduler(t_batch)

                # Predict noise
                eps_cond = self.network(X_t, t_vec, c)
                eps_uncond = self.network(X_t, t_vec, torch.zeros_like(c))

                # Classifierâfree guidance
                eps_hat = (1.0 + omega) * eps_cond - omega * eps_uncond

                # Compute the mean without noise
                coef   = (1.0 - sched['alpha_t']) / sched['sqrt_oneminus_alpha_bar']
                X_prev = sched['oneover_sqrt_alpha'] * (X_t - coef * eps_hat)

                # Add noise if t > 1
                if t > 1:
                    z = torch.randn_like(X_t)
                    X_prev = X_prev + sched['sqrt_beta_t'] * z

                # Step down
                X_t = X_prev

        # back to train mode
        self.network.train()

        # ==================================================== #
        generated_images = (X_t * 0.3081 + 0.1307).clamp(0,1) # denormalize the output images
        return generated_images


    # def sample(self, conditions, omega):
    #     T = self.dmconfig.T
    #     X_t = None
    #     # ==================================================== #
    #     # YOUR CODE HERE:
    #     #   Complete the training forward process based on the
    #     #   given sampling algorithm.
    #     #   Inputs:
    #     #       conditions: condition labels, with size (B). You should
    #     #                   convert it to one-hot encoded labels with size (B,10)
    #     #                   before making it as the input of the denoising network.
    #     #       omega: conditional guidance weight.
    #     #   Outputs:
    #     #       generated_images

    #     B = conditions.size(0)
    #     device = conditions.device

    #     # One-hot encode
    #     c = F.one_hot(conditions, num_classes=self.dmconfig.num_classes).float()

    #     # Gaussian noise
    #     X_t = torch.randn(B, 1, 28, 28, device=device)

    #     # put the network into eval & disable grads
    #     self.network.eval()
    #     with torch.no_grad():
    #         # Denoise
```

```
    #         for t in range(T, 0, -1):
    #             t_batch = torch.full((B,1), t, device=device, dtype=torch.long)
    #             sched = self.scheduler(t_batch)

    #             z = torch.randn_like(X_t) if t > 1 else torch.zeros_like(X_t)

    #             eps_cond   = self.network(X_t, t_batch.squeeze(-1),       c)
    #             eps_uncond = self.network(X_t, t_batch.squeeze(-1), torch.zeros_like(c))

    #             # Classifier-free guidance
    #             eps_hat = (1.0 + omega) * eps_cond - omega * eps_uncond

    #             coef = (1.0 - sched['alpha_t']) / sched['sqrt_oneminus_alpha_bar']
    #             X_t = (
    #                 sched['oneover_sqrt_alpha']
    #                 * (X_t - coef * eps_hat)
    #                 + sched['sqrt_beta_t'] * z
    #             )

    #     # back to train mode
    #     self.network.train()

    #     # ==================================================== #
    #     generated_images = (X_t * 0.3081 + 0.1307).clamp(0,1) # denormalize the output
images
    #     return generated_images
```