```python
from re import X
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
        denom = T-1
        previous_time_step = t_s - 1
        beta_t = beta_1 + ((beta_T-beta_1) * (previous_time_step)/denom)
        sqrt_beta_t = torch.sqrt(beta_t)
        alpha_t = 1 - beta_t
        oneover_sqrt_alpha = 1/(torch.sqrt(alpha_t))
        alpha = torch.linspace(beta_1, beta_T, steps=T)
        alpha = 1- alpha
        alpha_t_bar_list = torch.cumprod(alpha, dim=0)
        index = t_s.long()
        alpha_t_bar = alpha_t_bar_list[index-1]
        sqrt_alpha_bar = torch.sqrt(alpha_t_bar)
        sqrt_oneminus_alpha_bar = torch.sqrt(1-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
        # ==================================================== #
        # 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  .


        #device = 'cuda' if torch.cuda.is_available else 'cpu'
```

```python
        B = images.shape[0]
        one_hot_cond = F.one_hot(conditions, num_classes=10)
        X_t = torch.randn_like(images)

        t_steps = torch.randint(1, T+1, (B,1,1,1))

        schedule = self.scheduler(t_steps)
        sqrt_alpha_bar = schedule['sqrt_alpha_bar'].to('cuda')
        sqrt_oneminus_alpha_bar = schedule['sqrt_oneminus_alpha_bar'].to('cuda')

        x_t = sqrt_alpha_bar * images + sqrt_oneminus_alpha_bar * X_t
        t = t_steps/T #normalize time steps to ensure stability
        X_t_hat = self.network.forward(x_t, t, one_hot_cond)
        noise_loss = self.loss_fn(X_t_hat, X_t)
        # ==================================================== #

        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

        device = next(self.network.parameters()).device
        B = conditions.shape[0]
        h = self.dmconfig.input_dim[0]
        w = self.dmconfig.input_dim[1]
        c = self.dmconfig.num_channels
        condition_mask_value = self.dmconfig.condition_mask_value
        X_t = torch.randn(B,c,h,w, device=device)

        with torch.no_grad():
            for t in torch.arange(T,0,-1):
                schedule = self.scheduler(t)

                time_steps = torch.full((B,c,1,1),t, device=device)
                Z = torch.randn_like(X_t, device=device) if t > 1 else torch.zeros_like(X_t,
device=device)
                time_steps = (time_steps/T) # normalize time steps for stability

                beta_t = schedule['beta_t'].to(device)
                alpha_t = schedule['alpha_t'].to(device)
                oneover_sqrt_alpha = schedule['oneover_sqrt_alpha'].to(device)
                sqrt_oneminus_alpha_bar = schedule['sqrt_oneminus_alpha_bar'].to(device)
                sigma_t = torch.sqrt( beta_t )

                epsilon_theta = self.network(X_t, time_steps, conditions)
                epsilon_theta_hat = self.network(X_t, time_steps,
conditions*condition_mask_value)

                E_t = (omega+1) * epsilon_theta - omega * epsilon_theta_hat
                epsilon_t_term = (1-alpha_t)/sqrt_oneminus_alpha_bar * E_t
                X_t = oneover_sqrt_alpha * (X_t - epsilon_t_term) + sigma_t*Z
                # ==================================================== #
        generated_images = (X_t * 0.3081 + 0.1307).clamp(0,1) # denormalize the output images
        return generated_images
```

```
import torch
import torch.nn as nn
import math
class ResConvBlock(nn.Module):
    '''
    Basic residual convolutional block
    '''
    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.conv1 = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, 3, 1, 1),
            nn.BatchNorm2d(out_channels),
            nn.GELU(),
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(out_channels, out_channels, 3, 1, 1),
            nn.BatchNorm2d(out_channels),
            nn.GELU(),
        )

    def forward(self, x):
        x1 = self.conv1(x)
        x2 = self.conv2(x1)
        if self.in_channels == self.out_channels:
            out = x + x2
        else:
            out = x1 + x2
        return out / math.sqrt(2)


class UnetDown(nn.Module):
    '''
    UNet down block (encoding)
    '''
    def __init__(self, in_channels, out_channels):
        super().__init__()
        layers = [ResConvBlock(in_channels, out_channels), nn.MaxPool2d(2)]
        self.model = nn.Sequential(*layers)

    def forward(self, x):
        return self.model(x)


class UnetUp(nn.Module):
    '''
    UNet up block (decoding)
    '''
    def __init__(self, in_channels, out_channels):
        super().__init__()
        layers = [
            nn.ConvTranspose2d(in_channels, out_channels, 2, 2),
            ResConvBlock(out_channels, out_channels),
            ResConvBlock(out_channels, out_channels),
        ]
        self.model = nn.Sequential(*layers)

    def forward(self, x, skip):
        x = torch.cat((x, skip), 1)
        x = self.model(x)
        return x


class EmbedBlock(nn.Module):
    '''
    Embedding block to embed time step/condition to embedding space
```

```
    '''
    def __init__(self, input_dim, emb_dim):
        super().__init__()
        self.input_dim = input_dim
        layers = [
            nn.Linear(input_dim, emb_dim),
            nn.GELU(),
            nn.Linear(emb_dim, emb_dim),
        ]
        self.layers = nn.Sequential(*layers)

    def forward(self, x):
        # set embedblock untrainable
        for param in self.layers.parameters():
            param.requires_grad = False
        x = x.view(-1, self.input_dim)
        return self.layers(x)

class FusionBlock(nn.Module):
    '''
    Concatenation and fusion block for adding embeddings
    '''
    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.layers = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, 1),
            nn.BatchNorm2d(out_channels),
            nn.GELU(),
        )
    def forward(self, x, t, c):
        h,w = x.shape[-2:]
        return self.layers(torch.cat([x, t.repeat(1,1,h,w), c.repeat(1,1,h,w)], dim = 1))

class ConditionalUnet(nn.Module):
    def __init__(self, in_channels, n_feat = 128, n_classes = 10):
        super().__init__()

        self.in_channels = in_channels
        self.n_feat = n_feat
        self.n_classes = n_classes

        # embeddings
        self.timeembed1 = EmbedBlock(1, 2*n_feat)
        self.timeembed2 = EmbedBlock(1, 1*n_feat)
        self.conditionembed1 = EmbedBlock(n_classes, 2*n_feat)
        self.conditionembed2 = EmbedBlock(n_classes, 1*n_feat)

        # down path for encoding
        self.init_conv = ResConvBlock(in_channels, n_feat)
        self.downblock1 = UnetDown(n_feat, n_feat)
        self.downblock2 = UnetDown(n_feat, 2 * n_feat)
        self.to_vec = nn.Sequential(nn.AvgPool2d(7), nn.GELU())

        # up path for decoding
        self.upblock0 = nn.Sequential(
            nn.ConvTranspose2d(2 * n_feat, 2 * n_feat, 7, 7),
            nn.GroupNorm(8, 2 * n_feat),
            nn.ReLU(),
        )
        self.upblock1 = UnetUp(4 * n_feat, n_feat)
        self.upblock2 = UnetUp(2 * n_feat, n_feat)
        self.outblock = nn.Sequential(
            nn.Conv2d(2 * n_feat, n_feat, 3, 1, 1),
            nn.GroupNorm(8, n_feat),
            nn.ReLU(),
            nn.Conv2d(n_feat, self.in_channels, 3, 1, 1),
        )
```

```python
        # fusion blocks
        self.fusion1 = FusionBlock(3 * self.n_feat, self.n_feat)
        self.fusion2 = FusionBlock(6 * self.n_feat, 2 * self.n_feat)
        self.fusion3 = FusionBlock(3 * self.n_feat, self.n_feat)
        self.fusion4 = FusionBlock(3 * self.n_feat, self.n_feat)

    def forward(self, x, t, c):
        '''
        Inputs:
            x: input images, with size (B,1,28,28)
            t: input time steps, with size (B,1,1,1)
            c: input conditions (one-hot encoded labels), with size (B,10)
        '''
        device = 'cuda' if torch.cuda.is_available else 'cpu'
        t, c = t.float().to(device), c.float().to(device)

        # time step embedding
        temb1 = self.timeembed1(t).view(-1, self.n_feat * 2, 1, 1) # 256
        temb2 = self.timeembed2(t).view(-1, self.n_feat, 1, 1) # 128

        # condition embedding
        cemb1 = self.conditionembed1(c).view(-1, self.n_feat * 2, 1, 1) # 256
        cemb2 = self.conditionembed2(c).view(-1, self.n_feat, 1, 1) # 128

        # ==================================================== #
        # YOUR CODE HERE:
        #   Define the process of computing the output of a
        #   this network given the input x, t, and c.
        #   The input x, t, c indicate the input image, time step
        #   and the condition respectively.
        # A potential format is shown below, feel free to use your own ways to design it.
        # down0 =
        # down1 =
        # down2 =
        # up0 =
        # up1 =
        # up2 =
        # out = self.outblock(torch.cat((up2, down0), dim = 1))
        # ==================================================== #

        down0 = self.init_conv(x)
        down1 = self.downblock1(down0)
        fusion1 = self.fusion1(down1, temb2, cemb2)

        down2 = self.downblock2(fusion1)
        fusion2 = self.fusion2(down2, temb1, cemb1)

        to_vec = self.to_vec(fusion2)
        up0 = self.upblock0(to_vec)
        up1 = self.upblock1(up0, fusion2)
        fusion3 = self.fusion3(up1, temb2, cemb2)

        up2 = self.upblock2(fusion3, fusion1)
        fusion4 = self.fusion4(up2, temb2, cemb2)

        out = self.outblock(torch.cat((fusion4, down0), dim = 1))
        return out
```