Deep Learning Lab6

312554054 林辰翰

1. Introduction

In this lab, we are tasked with implementing two generative models: Generative Adversarial Networks (GANs) and Denoising Diffusion Probabilistic Models (DDPMs). Our goal is to generate synthetic images based on multi-label conditions. By observing the objects.json file, we identified 24 different classes. Given a specific set of label conditions, our DCGAN and DDPM models should be able to generate the corresponding synthetic images.

Briefly introducing these two models DCGAN and DDP:

DCGAN: Deep Convolutional Generative Adversarial Networks (DCGANs) are a type of GAN that use deep convolutional layers to generate synthetic images. In a GAN, two neural networks "the Generator" and "the Discriminator" work against each other in a competitive manner. (which is an important step to generate high quality image) The generator creates fake images from random noise, while the discriminator tries to distinguish between real and fake images. Through this adversarial process, the generator gradually improves its ability to produce realistic images by learning from the feedback provided by the discriminator. By leveraging convolutional layers, which are particularly effective for processing image data.

DDPM: Denoising Diffusion Probabilistic Models (DDPMs) are a class of generative models designed to create images by progressively denoising a sample of random noise. The method involves a forward process, where noise is systematically added to the data, and a reverse process, where the model learns to recover the original image from the noisy version. This approach is guided by a probabilistic framework that ensures each step is consistent with the data distribution.

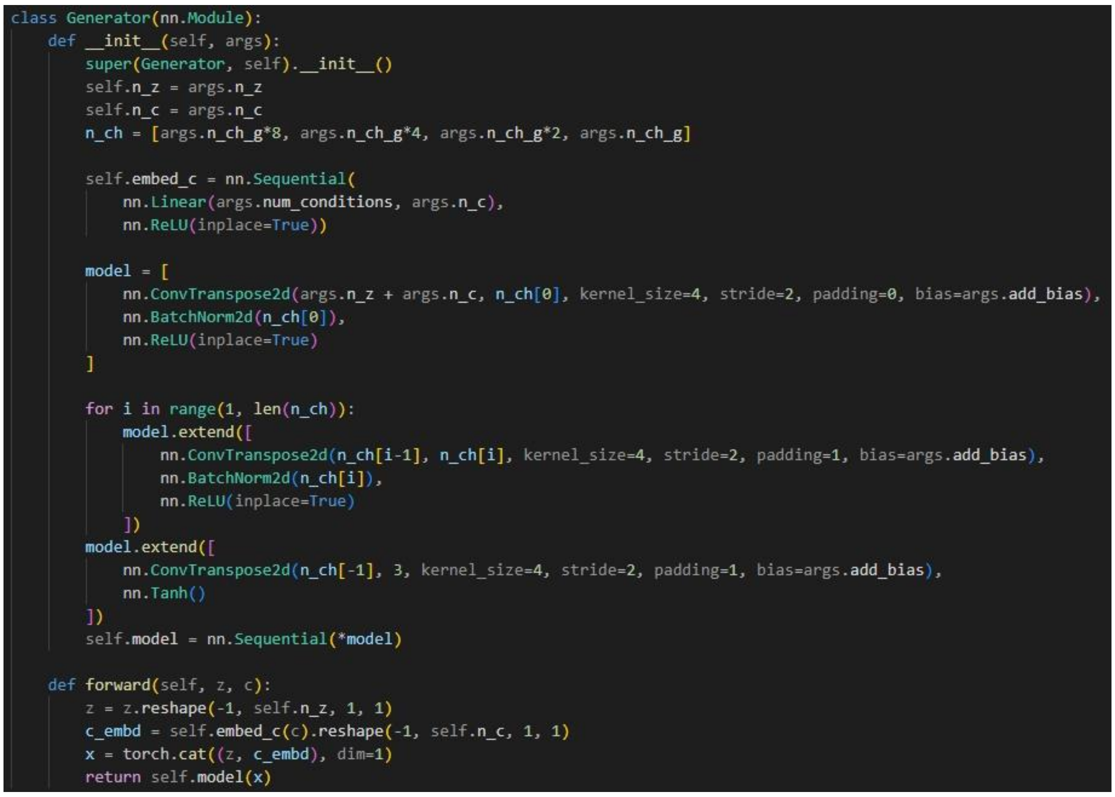
2. Implementation Details

2.1 DCGAN:

The architecture here I implement is Conditional DCGAN, which is an extension of the traditional DCGAN. It integrates additional label information to generate images conditioned on specified labels. This approach allows for more control over the generated content, ensuring that the images align with the given conditions. In this section, I will describe the design of both the generator and the discriminator in detail.

2.1.1 Generator

The generator network is designed to produce realistic images from random noise and conditional labels. The input to the generator is a concatenation of noise vector z and the embedded condition vector c.
Embedding Layer: Firstly, the condition vector ccc is passed through a fully connected layer to embed it into a higher-dimensional space, followed by a ReLU activation function to introduce non-linearity. This embedding helps the generator better understand and utilize the condition information.
The embedded condition vector is concatenated with the noise vector. This combined vector is then reshaped and fed into the initial convolutional transpose layer, which significantly increases the spatial dimensions of the input, setting the stage for further processing.
Intermediate Convolutional Transpose Layers: The core of the generator consists of a series of intermediate convolutional transpose layers, each paired with batch normalization and ReLU activation functions. These layers progressively increase the spatial dimensions while reducing the depth of the feature maps, guiding the transformation from the latent space to the desired image size.
The last layer outputs a three-channel image, with pixel values scaled between -1 and 1 using the Tanh activation function, ensuring the generated images fall within the same range as the real images, as recommended in paper.



```python
class Generator(nn.Module):
    def __init__(self, args):
        super(Generator, self).__init__()
        self.n_z = args.n_z
        self.n_c = args.n_c
        n_ch = [args.n_ch_g*8, args.n_ch_g*4, args.n_ch_g*2, args.n_ch_g]

        self.embed_c = nn.Sequential(
            nn.Linear(args.num_conditions, args.n_c),
            nn.ReLU(inplace=True))

        model = [
            nn.ConvTranspose2d(args.n_z + args.n_c, n_ch[0], kernel_size=4, stride=2, padding=0, bias=args.add_bias),
            nn.BatchNorm2d(n_ch[0]),
            nn.ReLU(inplace=True)
        ]

        for i in range(1, len(n_ch)):
            model.extend([
                nn.ConvTranspose2d(n_ch[i-1], n_ch[i], kernel_size=4, stride=2, padding=1, bias=args.add_bias),
                nn.BatchNorm2d(n_ch[i]),
                nn.ReLU(inplace=True)
            ])
        model.extend([
            nn.ConvTranspose2d(n_ch[-1], 3, kernel_size=4, stride=2, padding=1, bias=args.add_bias),
            nn.Tanh()
        ])
        self.model = nn.Sequential(*model)

    def forward(self, z, c):
        z = z.reshape(-1, self.n_z, 1, 1)
        c_embd = self.embed_c(c).reshape(-1, self.n_c, 1, 1)
        x = torch.cat((z, c_embd), dim=1)
        return self.model(x)
```

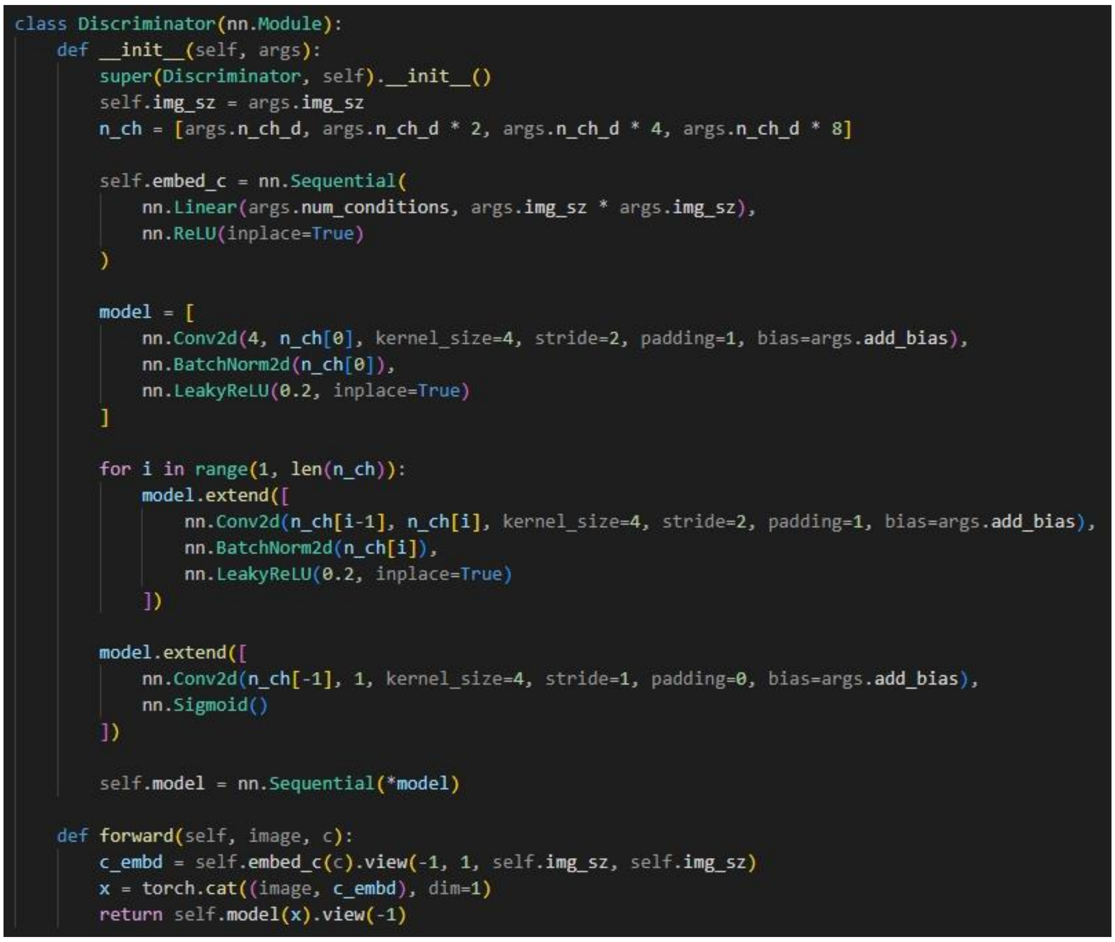
2.1.2 Discriminator

The design of discriminator ensures that itself can effectively leverage the conditional information to perform this task.

Embedding Layer: It transforms the condition vector into a higher-dimensional space, aligning its spatial dimensions with those of the input image. This embedded condition vector is then seamlessly concatenated with the input image, forming a combined input that is fed into the initial convolutional layer. This initial layer reduces the spatial dimensions while simultaneously increasing the depth of the feature maps, kickstarting the feature extraction process.

Intermediate Convolutional Layers: a series of intermediate convolutional layers, each equipped with batch normalization and LeakyReLU activation functions, further refine the extracted features. These layers progressively reduce the spatial dimensions while amplifying the depth, allowing the network to capture intricate and complex patterns within the data.

The last convolutional layer outputs a single value per image, which is then passed through a Sigmoid activation function.



```python
class Discriminator(nn.Module):
    def __init__(self, args):
        super(Discriminator, self).__init__()
        self.img_sz = args.img_sz
        n_ch = [args.n_ch_d, args.n_ch_d * 2, args.n_ch_d * 4, args.n_ch_d * 8]

        self.embed_c = nn.Sequential(
            nn.Linear(args.num_conditions, args.img_sz * args.img_sz),
            nn.ReLU(inplace=True)
        )

        model = [
            nn.Conv2d(4, n_ch[0], kernel_size=4, stride=2, padding=1, bias=args.add_bias),
            nn.BatchNorm2d(n_ch[0]),
            nn.LeakyReLU(0.2, inplace=True)
        ]

        for i in range(1, len(n_ch)):
            model.extend([
                nn.Conv2d(n_ch[i-1], n_ch[i], kernel_size=4, stride=2, padding=1, bias=args.add_bias),
                nn.BatchNorm2d(n_ch[i]),
                nn.LeakyReLU(0.2, inplace=True)
            ])

        model.extend([
            nn.Conv2d(n_ch[-1], 1, kernel_size=4, stride=1, padding=0, bias=args.add_bias),
            nn.Sigmoid()
        ])

        self.model = nn.Sequential(*model)

    def forward(self, image, c):
        c_embd = self.embed_c(c).view(-1, 1, self.img_sz, self.img_sz)
        x = torch.cat((image, c_embd), dim=1)
        return self.model(x).view(-1)
```

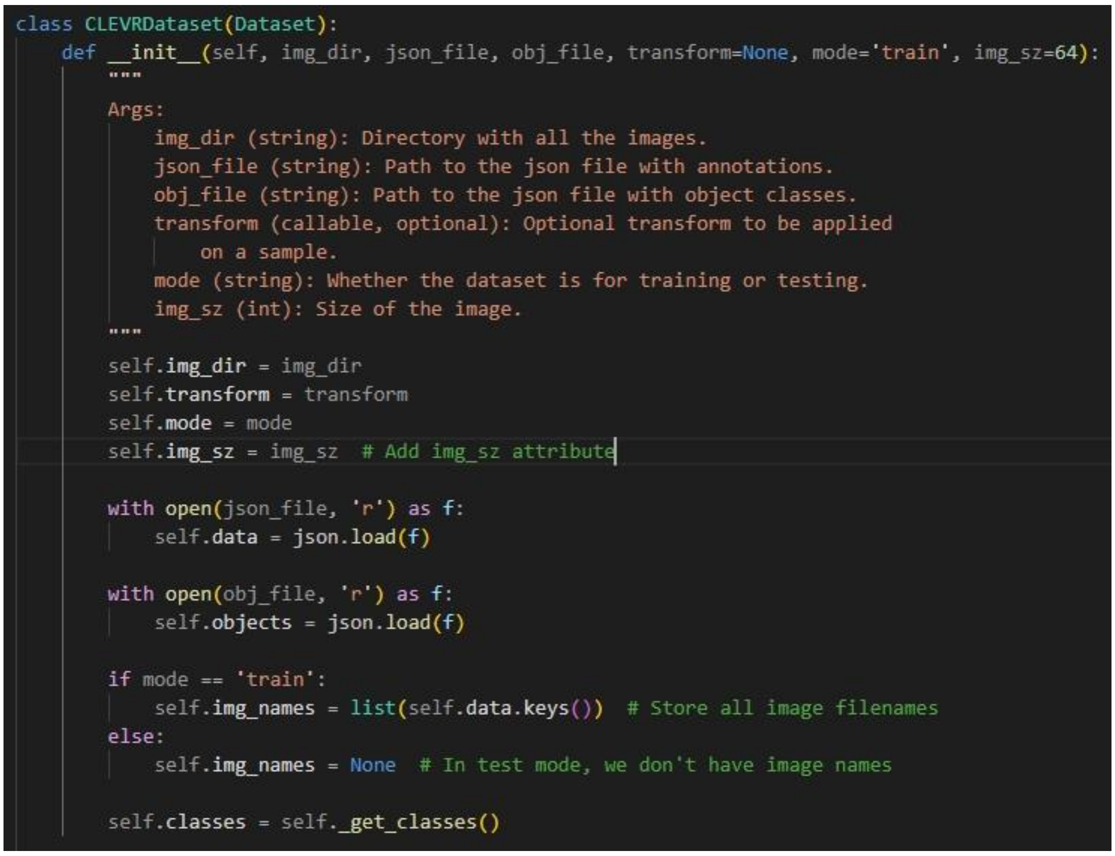
2.1.3 Dataloader

The CLEVR dataset consists of rendered images of 3D scenes with various objects, along with associated annotations describing the objects present in each image.

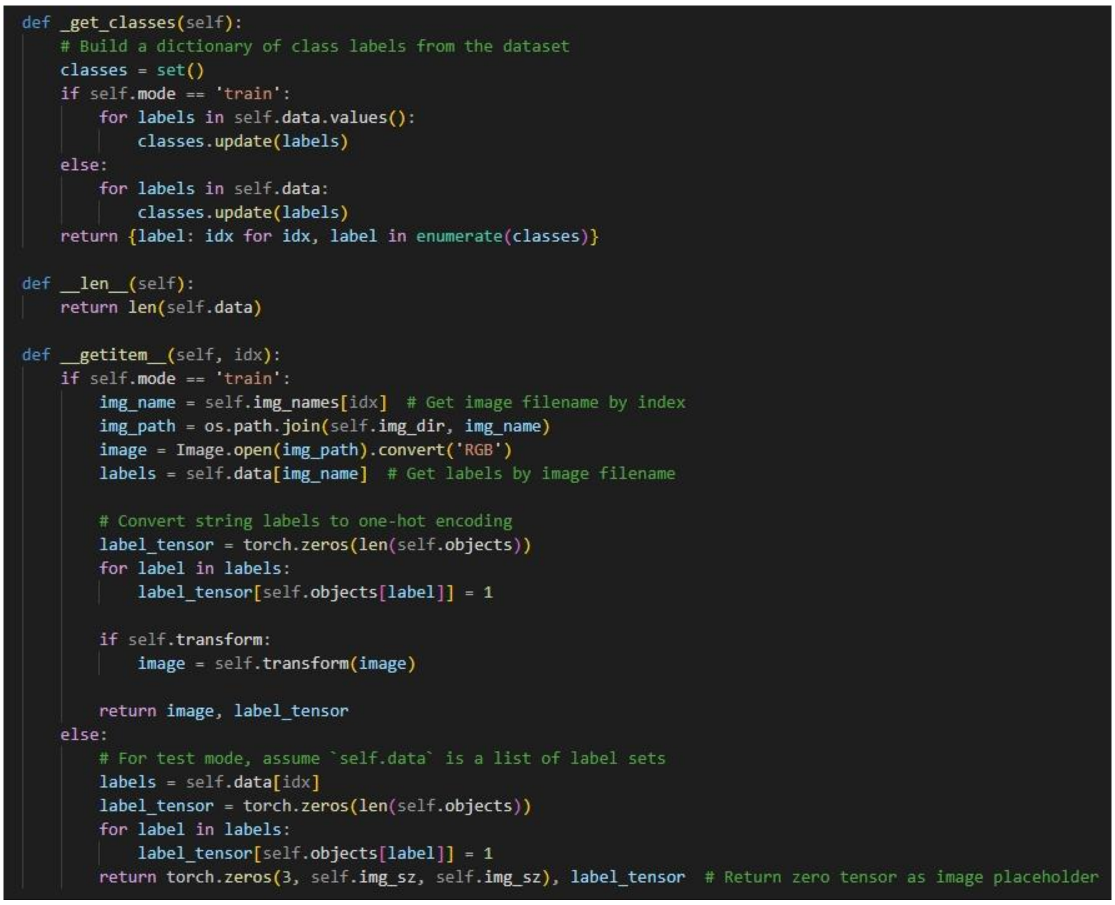
For the training mode, the annotations are converted to one-hot encoded tensors using the object classes. During the “getitem” method, the images are loaded, transformed if a transformation is provided, and returned along with the one-hot encoded label tensor.

In test mode, the JSON data contains lists of object labels without associated image filenames. The “getitem” method returns a zero tensor as a placeholder for the image, along with the one-hot encoded label tensor corresponding to the provided object labels.

”Get_classes” method builds a dictionary mapping object label to indices, which is also used for the one-hot encoding.



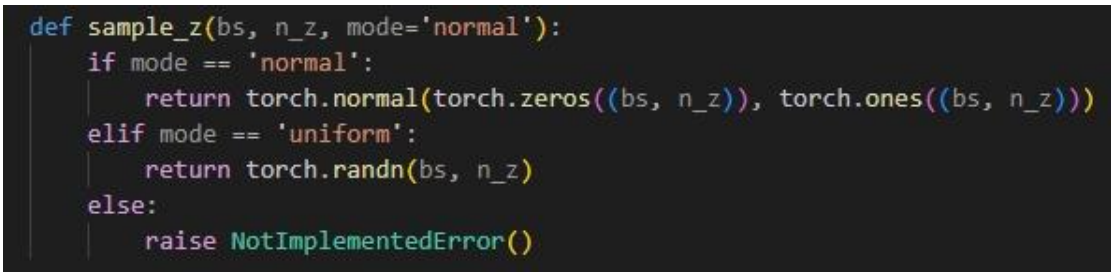
```python
class CLEVRDataset(Dataset):
    def __init__(self, img_dir, json_file, obj_file, transform=None, mode='train', img_sz=64):
        """
        Args:
            img_dir (string): Directory with all the images.
            json_file (string): Path to the json file with annotations.
            obj_file (string): Path to the json file with object classes.
            transform (callable, optional): Optional transform to be applied
                on a sample.
            mode (string): Whether the dataset is for training or testing.
            img_sz (int): Size of the image.
        """
        self.img_dir = img_dir
        self.transform = transform
        self.mode = mode
        self.img_sz = img_sz  # Add img_sz attribute

        with open(json_file, 'r') as f:
            self.data = json.load(f)

        with open(obj_file, 'r') as f:
            self.objects = json.load(f)

        if mode == 'train':
            self.img_names = list(self.data.keys())  # Store all image filenames
        else:
            self.img_names = None  # In test mode, we don't have image names

        self.classes = self._get_classes()
```



```python
def _get_classes(self):
    # Build a dictionary of class labels from the dataset
    classes = set()
    if self.mode == 'train':
        for labels in self.data.values():
            classes.update(labels)
    else:
        for labels in self.data:
            classes.update(labels)
    return {label: idx for idx, label in enumerate(classes)}

def __len__(self):
    return len(self.data)

def __getitem__(self, idx):
    if self.mode == 'train':
        img_name = self.img_names[idx]  # Get image filename by index
        img_path = os.path.join(self.img_dir, img_name)
        image = Image.open(img_path).convert('RGB')
        labels = self.data[img_name]  # Get labels by image filename

        # Convert string labels to one-hot encoding
        label_tensor = torch.zeros(len(self.objects))
        for label in labels:
            label_tensor[self.objects[label]] = 1

        if self.transform:
            image = self.transform(image)

        return image, label_tensor
    else:
        # For test mode, assume `self.data` is a list of label sets
        labels = self.data[idx]
        label_tensor = torch.zeros(len(self.objects))
        for label in labels:
            label_tensor[self.objects[label]] = 1
        return torch.zeros(3, self.img_sz, self.img_sz), label_tensor  # Return zero tensor as image placeholder
```

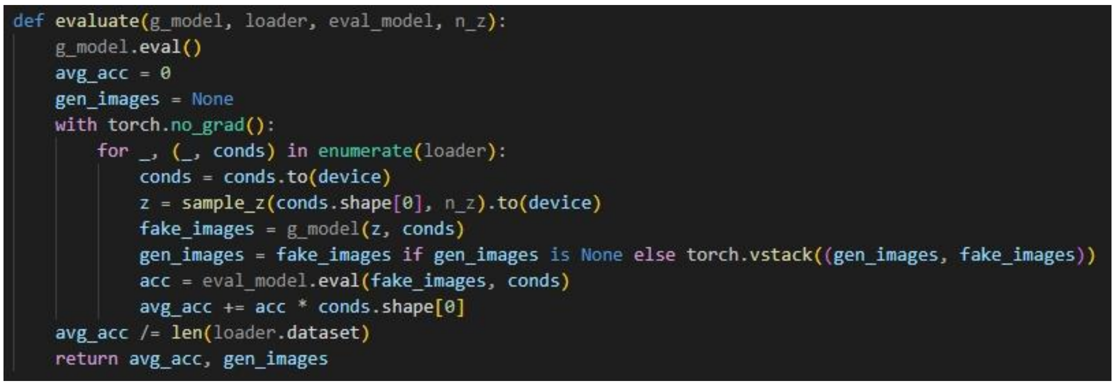
2.1.4 Training

"sample_z" function generates random noise vectors that serve as inputs for the generator model.



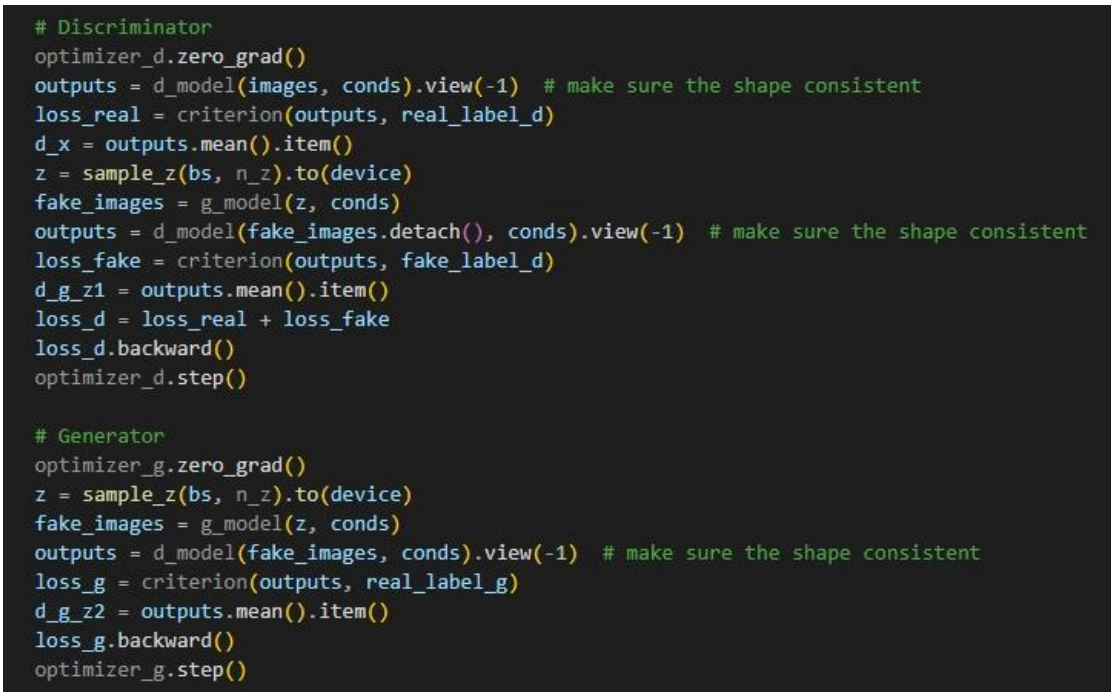
```python
def sample_z(bs, n_z, mode='normal'):
    if mode == 'normal':
        return torch.normal(torch.zeros((bs, n_z)), torch.ones((bs, n_z)))
    elif mode == 'uniform':
        return torch.randn(bs, n_z)
    else:
        raise NotImplementedError()
```

"evaluate" function which is implemented from the provided file "evaluator.py" assesses the generator's performance by calculating the average accuracy of the generated images against the conditions using a pre-trained evaluation model.

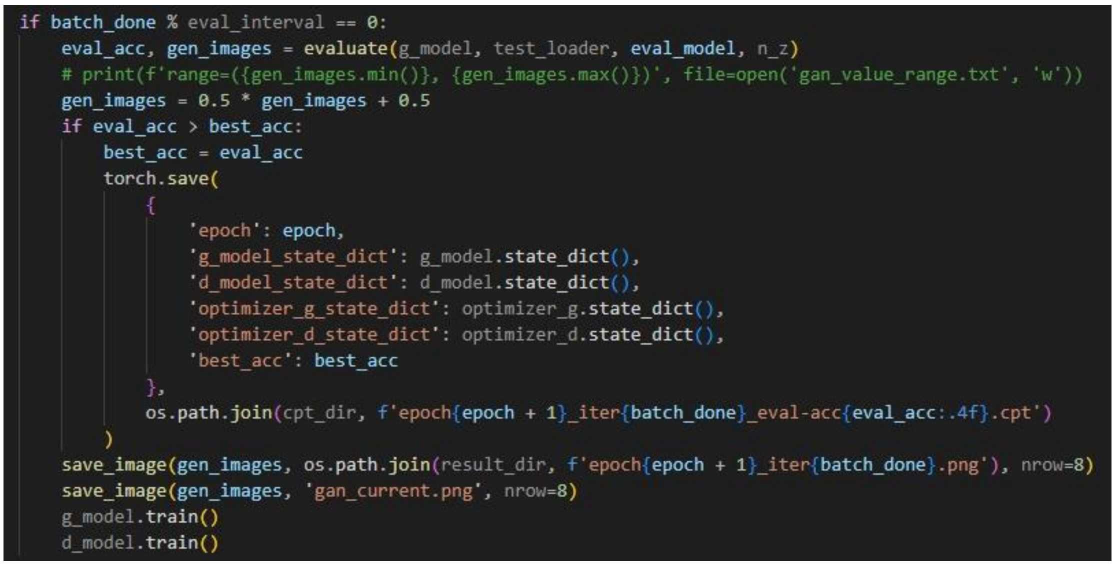


```python
def evaluate(g_model, loader, eval_model, n_z):
    g_model.eval()
    avg_acc = 0
    gen_images = None
    with torch.no_grad():
        for _, (_, conds) in enumerate(loader):
            conds = conds.to(device)
            z = sample_z(conds.shape[0], n_z).to(device)
            fake_images = g_model(z, conds)
            gen_images = fake_images if gen_images is None else torch.vstack((gen_images, fake_images))
            acc = eval_model.eval(fake_images, conds)
            avg_acc += acc * conds.shape[0]
    avg_acc /= len(loader.dataset)
    return avg_acc, gen_images
```

This function initializes the generator and discriminator models, sets up optimizers, and defines the loss criterion. The training loop iterates over epochs, training the discriminator to distinguish between real and fake images and the generator to create realistic images from noise vectors conditioned on input labels.



```
# Discriminator
optimizer_d.zero_grad()
outputs = d_model(images, conds).view(-1)  # make sure the shape consistent
loss_real = criterion(outputs, real_label_d)
d_x = outputs.mean().item()
z = sample_z(bs, n_z).to(device)
fake_images = g_model(z, conds)
outputs = d_model(fake_images.detach(), conds).view(-1)  # make sure the shape consistent
loss_fake = criterion(outputs, fake_label_d)
d_g_z1 = outputs.mean().item()
loss_d = loss_real + loss_fake
loss_d.backward()
optimizer_d.step()

# Generator
optimizer_g.zero_grad()
z = sample_z(bs, n_z).to(device)
fake_images = g_model(z, conds)
outputs = d_model(fake_images, conds).view(-1)  # make sure the shape consistent
loss_g = criterion(outputs, real_label_g)
d_g_z2 = outputs.mean().item()
loss_g.backward()
optimizer_g.step()
```



```
if batch_done % eval_interval == 0:
    eval_acc, gen_images = evaluate(g_model, test_loader, eval_model, n_z)
    # print(f'range=({gen_images.min()}, {gen_images.max()})', file=open('gan_value_range.txt', 'w'))
    gen_images = 0.5 * gen_images + 0.5
    if eval_acc > best_acc:
        best_acc = eval_acc
        torch.save(
            {
                'epoch': epoch,
                'g_model_state_dict': g_model.state_dict(),
                'd_model_state_dict': d_model.state_dict(),
                'optimizer_g_state_dict': optimizer_g.state_dict(),
                'optimizer_d_state_dict': optimizer_d.state_dict(),
                'best_acc': best_acc
            },
            os.path.join(cpt_dir, f'epoch{epoch + 1}_iter{batch_done}_eval-acc{eval_acc:.4f}.cpt')
        )
    save_image(gen_images, os.path.join(result_dir, f'epoch{epoch + 1}_iter{batch_done}.png'), nrow=8)
    save_image(gen_images, 'gan_current.png', nrow=8)
    g_model.train()
    d_model.train()
```

2.2 DDPM:

The architecture I implement is the Denoising Diffusion Probabilistic Model (DDPM), which is an extension of traditional generative models. It leverages a diffusion process that gradually adds noise to the data in a forward process and then learns to reverse this process to generate data from noise. This approach allows for more stable and high-quality image generation, ensuring that the images are refined and accurate.

2.2.1 Unet

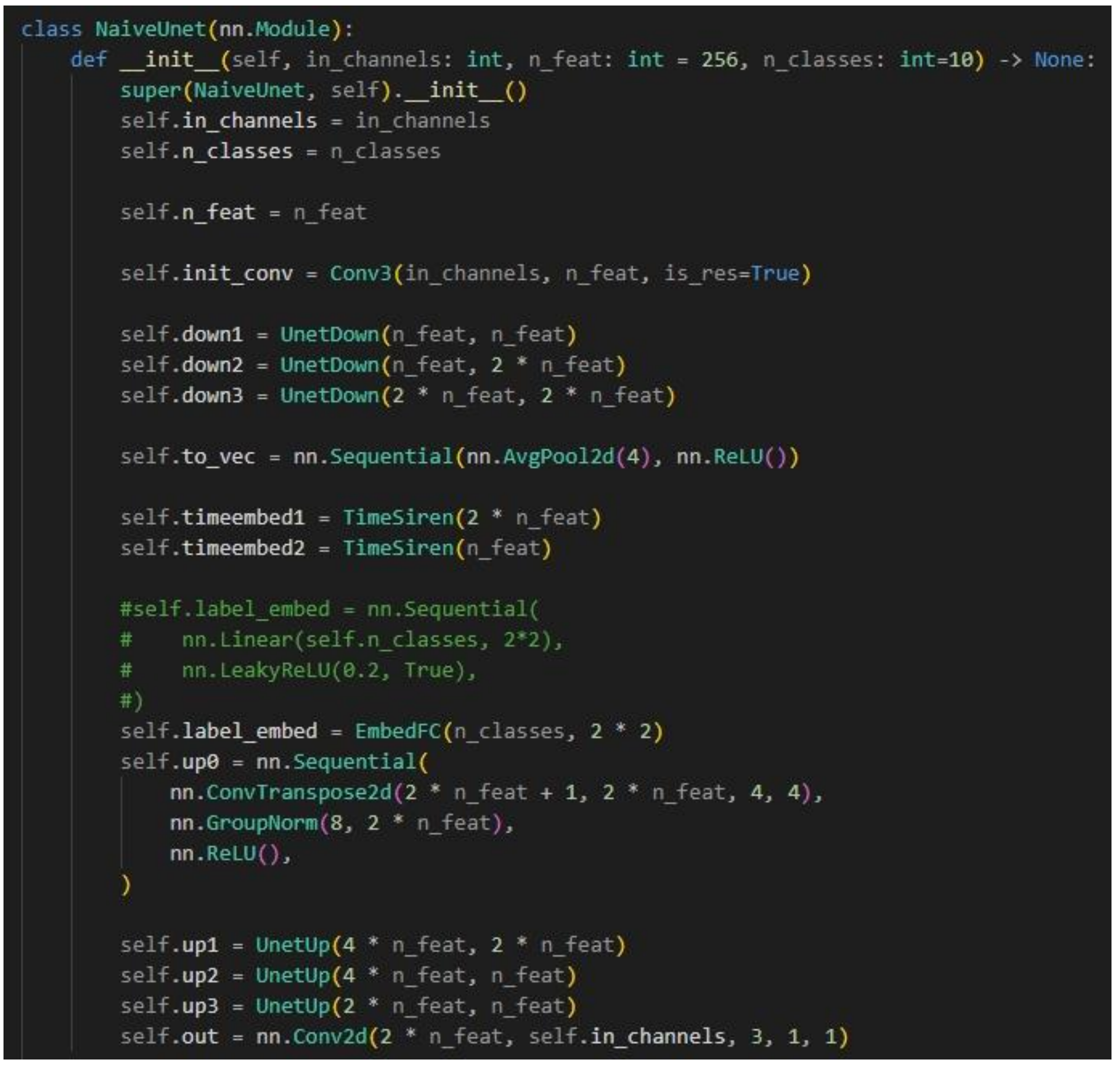
The UNet architecture I chose features a design that includes 3 downsampling layers and 3 upsampling layers. This choice balances computational efficiency and model complexity, enabling the network to effectively capture and reconstruct image features at different scales.

Downsampling Layers: The three downsampling layers progressively reduce the spatial dimensions of the input image while increasing the depth of the feature maps. Each downsampling step consists of convolutional layers, batch normalization and max pooling.

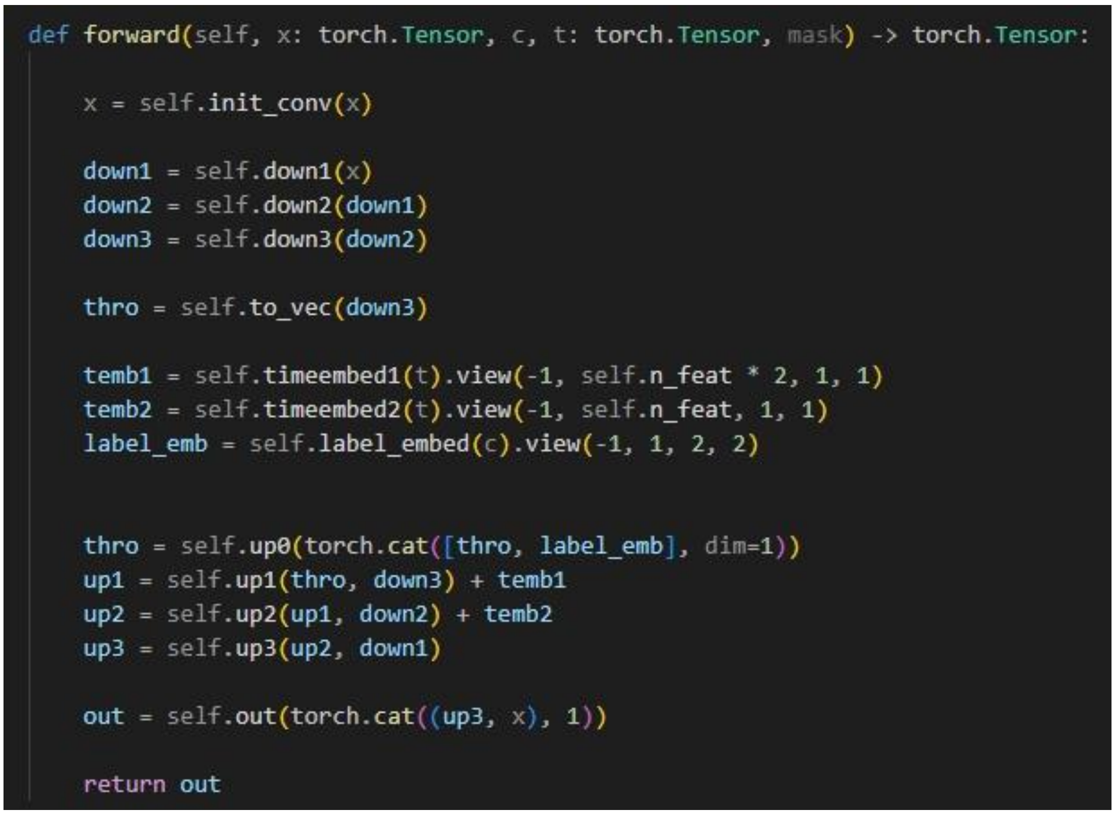
Upsampling Layers: The three upsampling layers mirror the downsampling process, progressively increasing the spatial dimensions of the feature maps to reconstruct the original image size. Each upsampling step consists of transposed convolutions, concatenation with skip connections and convolutional layers.

Embedding Layers: To effectively incorporate temporal and contextual information, the UNet architecture includes two types of embedding layers: time embedding layers and a label embedding layer.

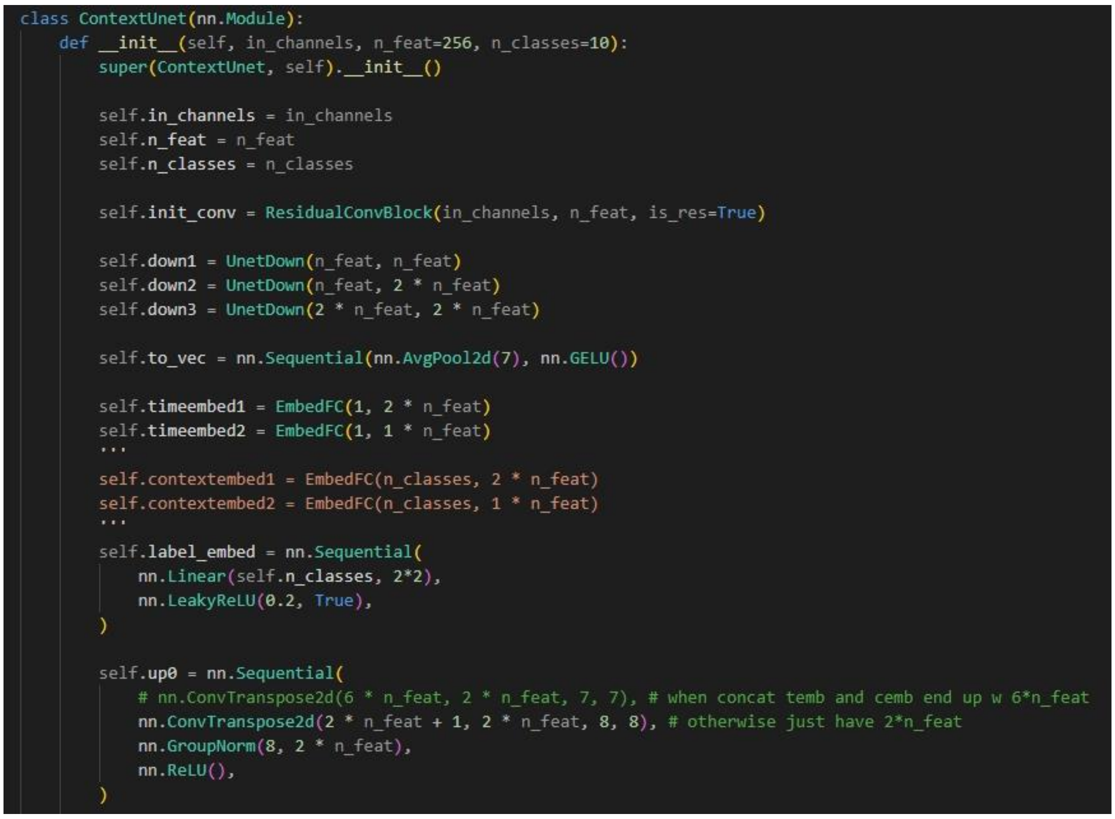
The UNet primarily serves as the generative model, responsible for denoising and reconstructing images.



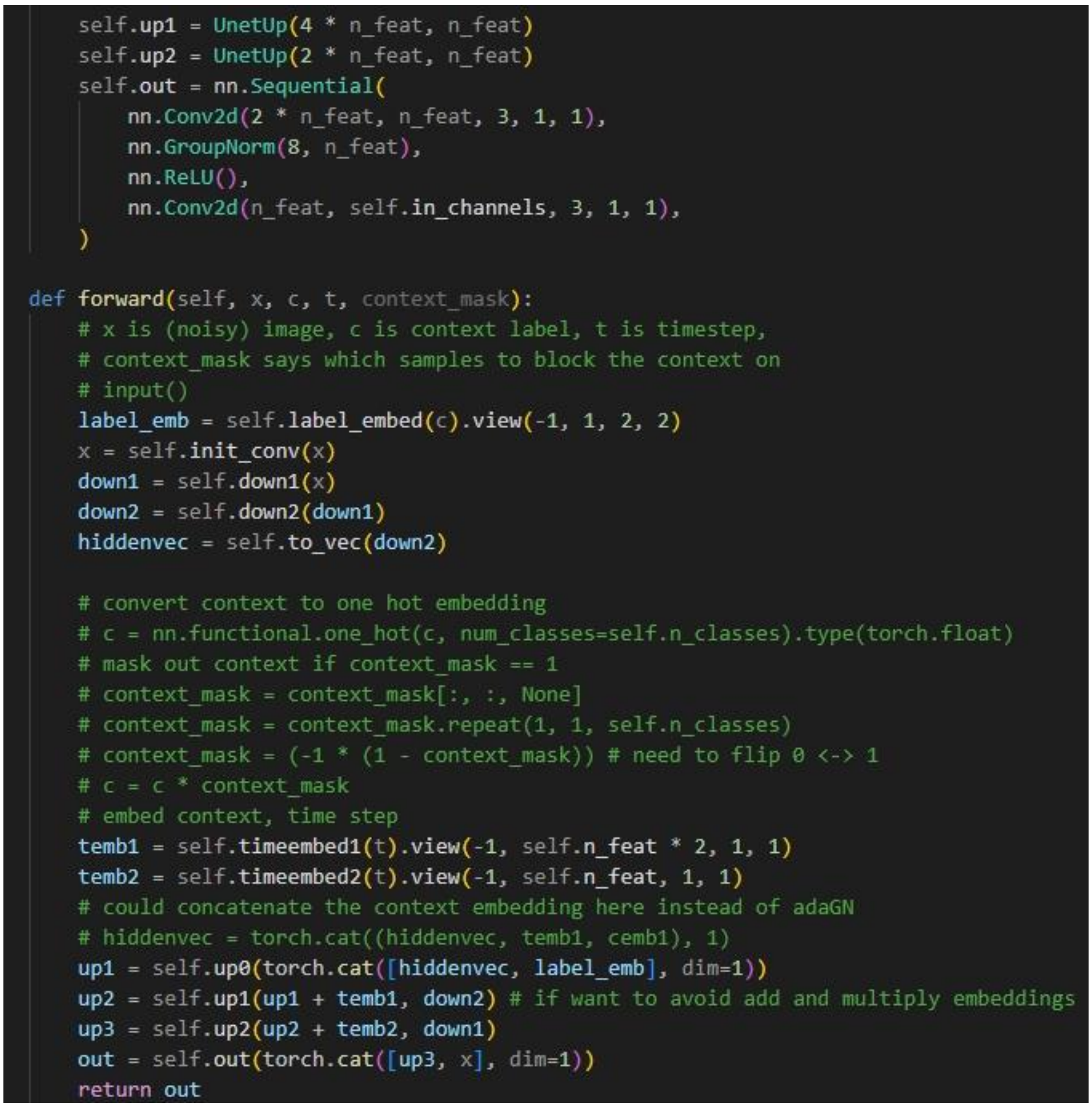
```python
class NaiveUnet(nn.Module):
    def __init__(self, in_channels: int, n_feat: int = 256, n_classes: int=10) -> None:
        super(NaiveUnet, self).__init__()
        self.in_channels = in_channels
        self.n_classes = n_classes

        self.n_feat = n_feat

        self.init_conv = Conv3(in_channels, n_feat, is_res=True)

        self.down1 = UnetDown(n_feat, n_feat)
        self.down2 = UnetDown(n_feat, 2 * n_feat)
        self.down3 = UnetDown(2 * n_feat, 2 * n_feat)

        self.to_vec = nn.Sequential(nn.AvgPool2d(4), nn.ReLU())

        self.timeembed1 = TimeSiren(2 * n_feat)
        self.timeembed2 = TimeSiren(n_feat)

        #self.label_embed = nn.Sequential(
        #    nn.Linear(self.n_classes, 2*2),
        #    nn.LeakyReLU(0.2, True),
        #)
        self.label_embed = EmbedFC(n_classes, 2 * 2)
        self.up0 = nn.Sequential(
            nn.ConvTranspose2d(2 * n_feat + 1, 2 * n_feat, 4, 4),
            nn.GroupNorm(8, 2 * n_feat),
            nn.ReLU(),
        )

        self.up1 = UnetUp(4 * n_feat, 2 * n_feat)
        self.up2 = UnetUp(4 * n_feat, n_feat)
        self.up3 = UnetUp(2 * n_feat, n_feat)
        self.out = nn.Conv2d(2 * n_feat, self.in_channels, 3, 1, 1)
```



```python
def forward(self, x: torch.Tensor, c, t: torch.Tensor, mask) -> torch.Tensor:

    x = self.init_conv(x)

    down1 = self.down1(x)
    down2 = self.down2(down1)
    down3 = self.down3(down2)

    thro = self.to_vec(down3)

    temb1 = self.timeembed1(t).view(-1, self.n_feat * 2, 1, 1)
    temb2 = self.timeembed2(t).view(-1, self.n_feat, 1, 1)
    label_emb = self.label_embed(c).view(-1, 1, 2, 2)


    thro = self.up0(torch.cat([thro, label_emb], dim=1))
    up1 = self.up1(thro, down3) + temb1
    up2 = self.up2(up1, down2) + temb2
    up3 = self.up3(up2, down1)

    out = self.out(torch.cat((up3, x), 1))

    return out
```



```python
class ContextUnet(nn.Module):
    def __init__(self, in_channels, n_feat=256, n_classes=10):
        super(ContextUnet, self).__init__()

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

        self.init_conv = ResidualConvBlock(in_channels, n_feat, is_res=True)

        self.down1 = UnetDown(n_feat, n_feat)
        self.down2 = UnetDown(n_feat, 2 * n_feat)
        self.down3 = UnetDown(2 * n_feat, 2 * n_feat)

        self.to_vec = nn.Sequential(nn.AvgPool2d(7), nn.GELU())

        self.timeembed1 = EmbedFC(1, 2 * n_feat)
        self.timeembed2 = EmbedFC(1, 1 * n_feat)
        '''
        self.contextembed1 = EmbedFC(n_classes, 2 * n_feat)
        self.contextembed2 = EmbedFC(n_classes, 1 * n_feat)
        '''
        self.label_embed = nn.Sequential(
            nn.Linear(self.n_classes, 2*2),
            nn.LeakyReLU(0.2, True),
        )

        self.up0 = nn.Sequential(
            # nn.ConvTranspose2d(6 * n_feat, 2 * n_feat, 7, 7), # when concat temb and cemb end up w 6*n_feat
            nn.ConvTranspose2d(2 * n_feat + 1, 2 * n_feat, 8, 8), # otherwise just have 2*n_feat
            nn.GroupNorm(8, 2 * n_feat),
            nn.ReLU(),
        )
```



```
        self.up1 = UnetUp(4 * n_feat, n_feat)
        self.up2 = UnetUp(2 * n_feat, n_feat)
        self.out = 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),
        )

    def forward(self, x, c, t, context_mask):
        # x is (noisy) image, c is context label, t is timestep,
        # context_mask says which samples to block the context on
        # input()
        label_emb = self.label_embed(c).view(-1, 1, 2, 2)
        x = self.init_conv(x)
        down1 = self.down1(x)
        down2 = self.down2(down1)
        hiddenvec = self.to_vec(down2)

        # convert context to one hot embedding
        # c = nn.functional.one_hot(c, num_classes=self.n_classes).type(torch.float)
        # mask out context if context_mask == 1
        # context_mask = context_mask[:, :, None]
        # context_mask = context_mask.repeat(1, 1, self.n_classes)
        # context_mask = (-1 * (1 - context_mask)) # need to flip 0 <-> 1
        # c = c * context_mask
        # embed context, time step
        temb1 = self.timeembed1(t).view(-1, self.n_feat * 2, 1, 1)
        temb2 = self.timeembed2(t).view(-1, self.n_feat, 1, 1)
        # could concatenate the context embedding here instead of adaGN
        # hiddenvec = torch.cat((hiddenvec, temb1, cemb1), 1)
        up1 = self.up0(torch.cat([hiddenvec, label_emb], dim=1))
        up2 = self.up1(up1 + temb1, down2) # if want to avoid add and multiply embeddings
        up3 = self.up2(up2 + temb2, down1)
        out = self.out(torch.cat([up3, x], dim=1))
        return out
```

2.2.2 DDPM Class

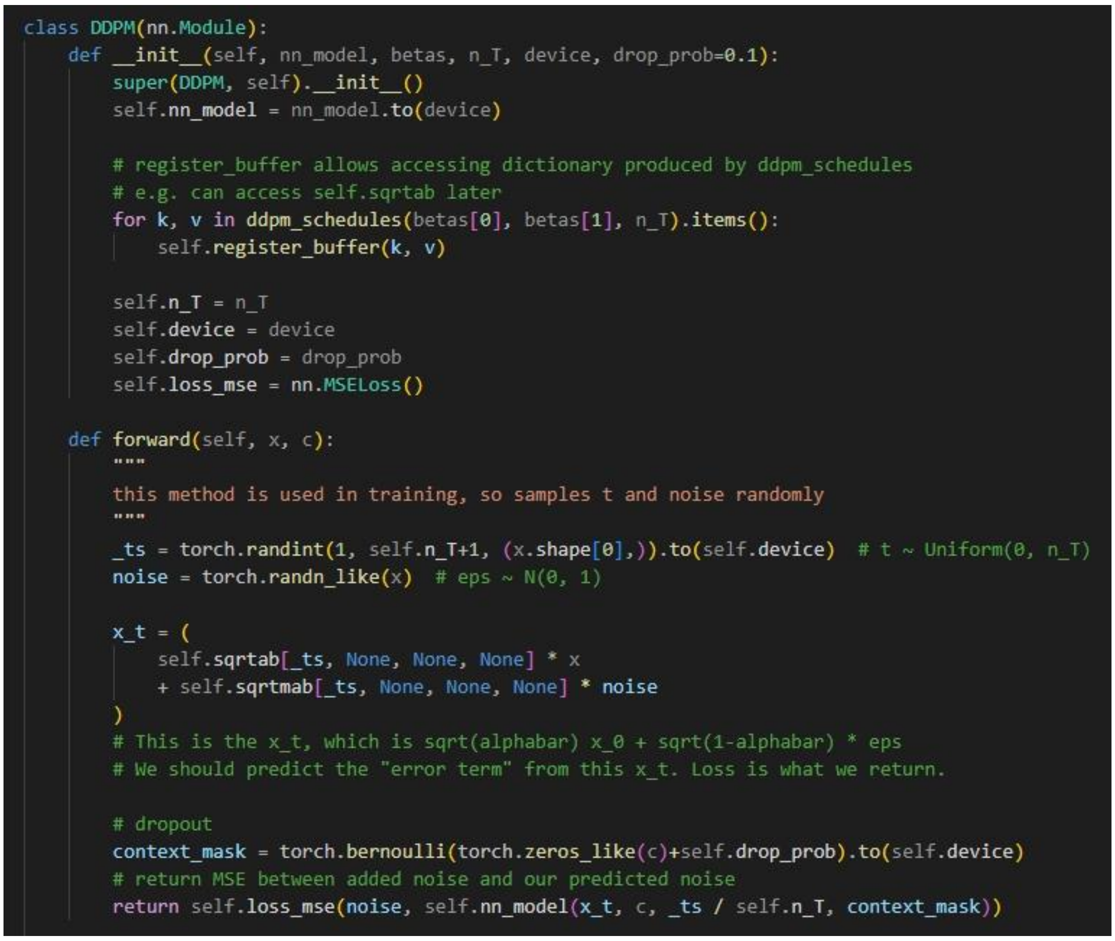
DDPM class and methods appear to be modified from a context-conditioned DDPM.

Context Variable: Both the forward and sample methods take a c (context) parameter, which is used to condition the generation process.

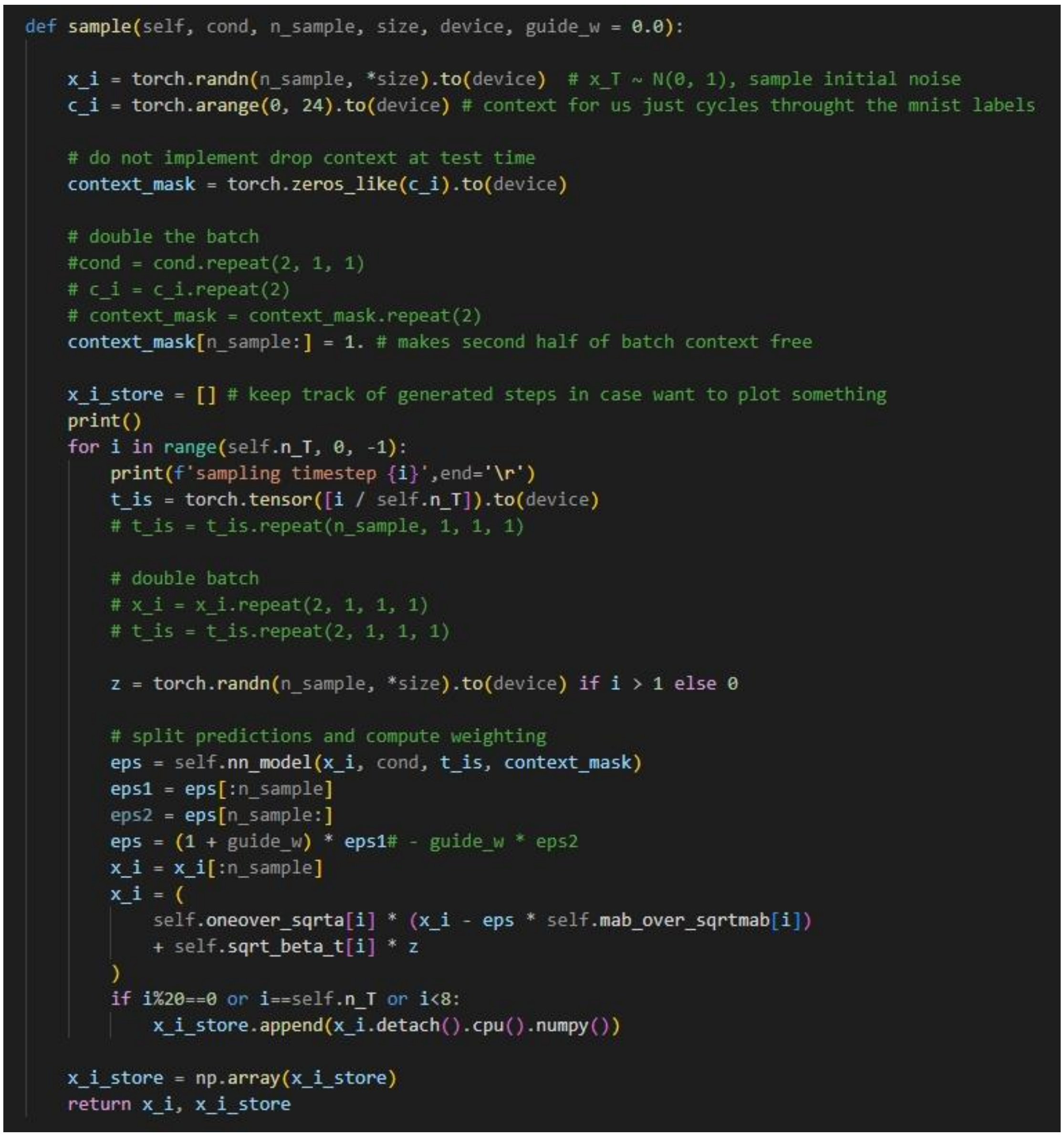
Context Masking: The forward method includes a context mask (context_mask) created using a Bernoulli dropout, which indicates that the context is sometimes masked out during training, a common technique in context-conditioned models to improve robustness.

Guidance Weight: The sample method includes a guide_w parameter, which allows for weighting the influence of the context during the sampling process.

This is used to combine unconditional and conditional predictions.



```python
class DDPM(nn.Module):
    def __init__(self, nn_model, betas, n_T, device, drop_prob=0.1):
        super(DDPM, self).__init__()
        self.nn_model = nn_model.to(device)

        # register_buffer allows accessing dictionary produced by ddpm_schedules
        # e.g. can access self.sqrtab later
        for k, v in ddpm_schedules(betas[0], betas[1], n_T).items():
            self.register_buffer(k, v)

        self.n_T = n_T
        self.device = device
        self.drop_prob = drop_prob
        self.loss_mse = nn.MSELoss()

    def forward(self, x, c):
        """
        this method is used in training, so samples t and noise randomly
        """
        _ts = torch.randint(1, self.n_T+1, (x.shape[0],)).to(self.device)  # t ~ Uniform(0, n_T)
        noise = torch.randn_like(x)  # eps ~ N(0, 1)

        x_t = (
            self.sqrtab[_ts, None, None, None] * x
            + self.sqrtmab[_ts, None, None, None] * noise
        )
        # This is the x_t, which is sqrt(alphabar) x_0 + sqrt(1-alphabar) * eps
        # We should predict the "error term" from this x_t. Loss is what we return.

        # dropout
        context_mask = torch.bernoulli(torch.zeros_like(c)+self.drop_prob).to(self.device)
        # return MSE between added noise and our predicted noise
        return self.loss_mse(noise, self.nn_model(x_t, c, _ts / self.n_T, context_mask))
```



```python
def sample(self, cond, n_sample, size, device, guide_w = 0.0):

    x_i = torch.randn(n_sample, *size).to(device)  # x_T ~ N(0, 1), sample initial noise
    c_i = torch.arange(0, 24).to(device) # context for us just cycles throught the mnist labels

    # do not implement drop context at test time
    context_mask = torch.zeros_like(c_i).to(device)

    # double the batch
    #cond = cond.repeat(2, 1, 1)
    # c_i = c_i.repeat(2)
    # context_mask = context_mask.repeat(2)
    context_mask[n_sample:] = 1. # makes second half of batch context free

    x_i_store = [] # keep track of generated steps in case want to plot something
    print()
    for i in range(self.n_T, 0, -1):
        print(f'sampling timestep {i}',end='\r')
        t_is = torch.tensor([i / self.n_T]).to(device)
        # t_is = t_is.repeat(n_sample, 1, 1, 1)

        # double batch
        # x_i = x_i.repeat(2, 1, 1, 1)
        # t_is = t_is.repeat(2, 1, 1, 1)

        z = torch.randn(n_sample, *size).to(device) if i > 1 else 0

        # split predictions and compute weighting
        eps = self.nn_model(x_i, cond, t_is, context_mask)
        eps1 = eps[:n_sample]
        eps2 = eps[n_sample:]
        eps = (1 + guide_w) * eps1# - guide_w * eps2
        x_i = x_i[:n_sample]
        x_i = (
            self.oneover_sqrta[i] * (x_i - eps * self.mab_over_sqrtmab[i])
            + self.sqrt_beta_t[i] * z
        )
        if i%20==0 or i==self.n_T or i<8:
            x_i_store.append(x_i.detach().cpu().numpy())

    x_i_store = np.array(x_i_store)
    return x_i, x_i_store
```

2.2.3 DDPM schedule

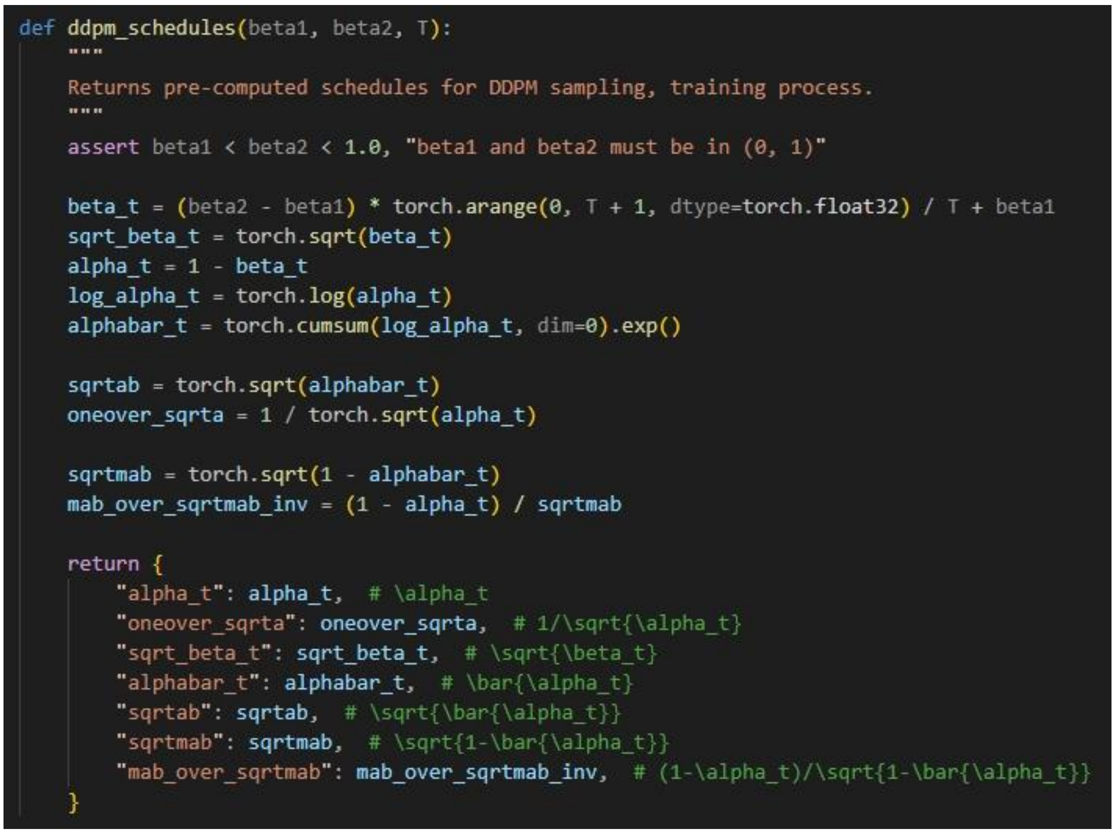
The “ddpm_schedules” function computes pre-defined schedules for DDPM sampling and training. The function takes three parameters: beta1, beta2, and T.

Beta Schedule: beta_t is computed as a linear space between beta1 and beta2 over T steps.

Alpha and Log Alpha: alpha_t is calculated as 1 - beta_t, and log_alpha_t is the natural logarithm of alpha_t.

Alpha Bar: alphabar_t is the cumulative product of log_alpha_t.

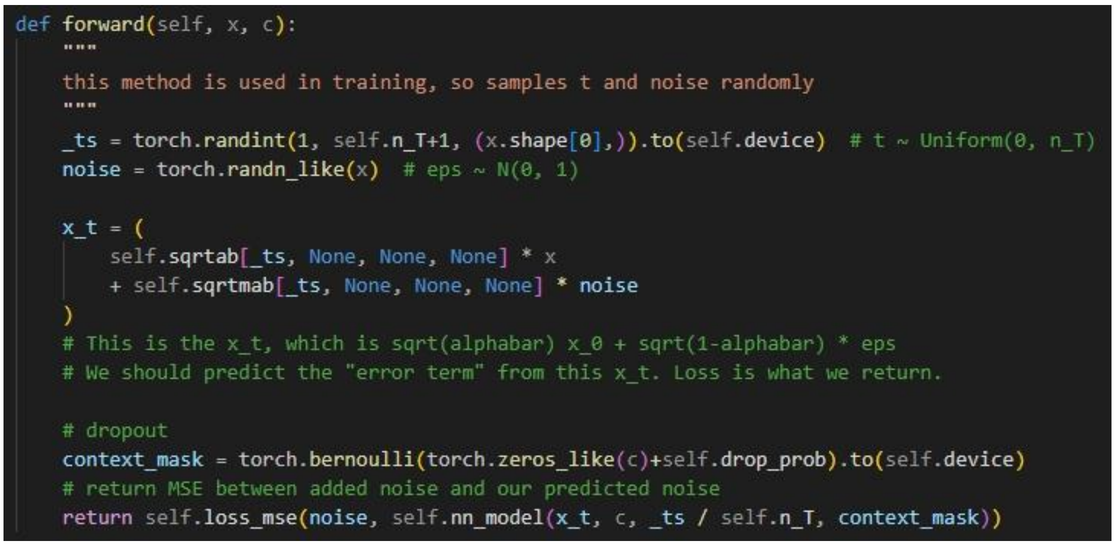
Square Roots and Inverses: The function computes several square root values and their inverses for efficient computation during sampling.



```python
def ddpm_schedules(beta1, beta2, T):
    """
    Returns pre-computed schedules for DDPM sampling, training process.
    """
    assert beta1 < beta2 < 1.0, "beta1 and beta2 must be in (0, 1)"

    beta_t = (beta2 - beta1) * torch.arange(0, T + 1, dtype=torch.float32) / T + beta1
    sqrt_beta_t = torch.sqrt(beta_t)
    alpha_t = 1 - beta_t
    log_alpha_t = torch.log(alpha_t)
    alphabar_t = torch.cumsum(log_alpha_t, dim=0).exp()

    sqrtab = torch.sqrt(alphabar_t)
    oneover_sqrta = 1 / torch.sqrt(alpha_t)

    sqrtmab = torch.sqrt(1 - alphabar_t)
    mab_over_sqrtmab_inv = (1 - alpha_t) / sqrtmab

    return {
        "alpha_t": alpha_t,  # \alpha_t
        "oneover_sqrta": oneover_sqrta,  # 1/\sqrt{\alpha_t}
        "sqrt_beta_t": sqrt_beta_t,  # \sqrt{\beta_t}
        "alphabar_t": alphabar_t,  # \bar{\alpha_t}
        "sqrtab": sqrtab,  # \sqrt{\bar{\alpha_t}}
        "sqrtmab": sqrtmab,  # \sqrt{1-\bar{\alpha_t}}
        "mab_over_sqrtmab": mab_over_sqrtmab_inv,  # (1-\alpha_t)/\sqrt{1-\bar{\alpha_t}}
    }
```

2.2.4 Loss

The forward method randomly samples a time step t and Gaussian noise for the input data x.

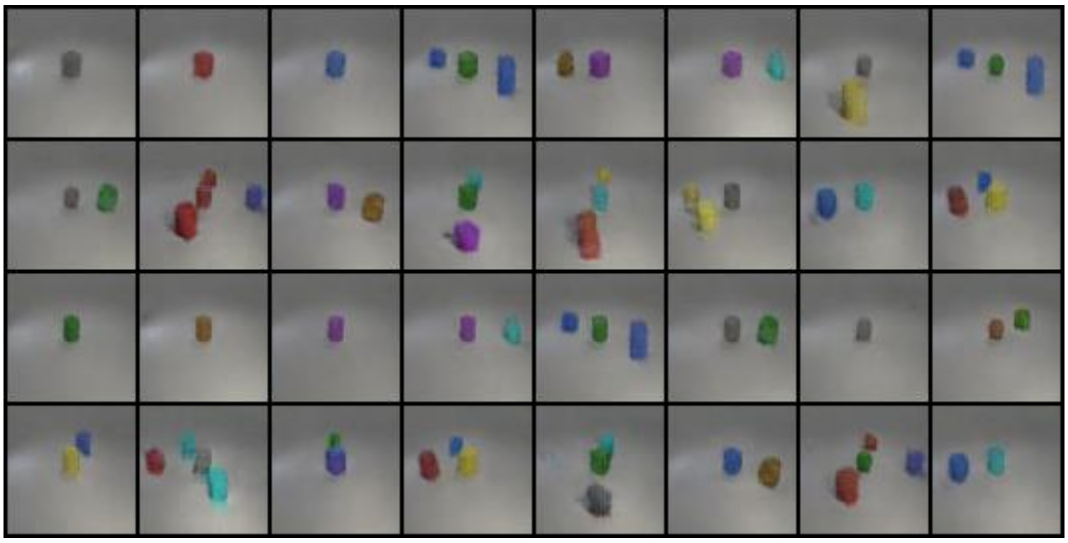


```python
def forward(self, x, c):
    """
    this method is used in training, so samples t and noise randomly
    """
    _ts = torch.randint(1, self.n_T+1, (x.shape[0],)).to(self.device)  # t ~ Uniform(0, n_T)
    noise = torch.randn_like(x)  # eps ~ N(0, 1)

    x_t = (
        self.sqrtab[_ts, None, None, None] * x
        + self.sqrtmab[_ts, None, None, None] * noise
    )
    # This is the x_t, which is sqrt(alphabar) x_0 + sqrt(1-alphabar) * eps
    # We should predict the "error term" from this x_t. Loss is what we return.

    # dropout
    context_mask = torch.bernoulli(torch.zeros_like(c)+self.drop_prob).to(self.device)
    # return MSE between added noise and our predicted noise
    return self.loss_mse(noise, self.nn_model(x_t, c, _ts / self.n_T, context_mask))
```

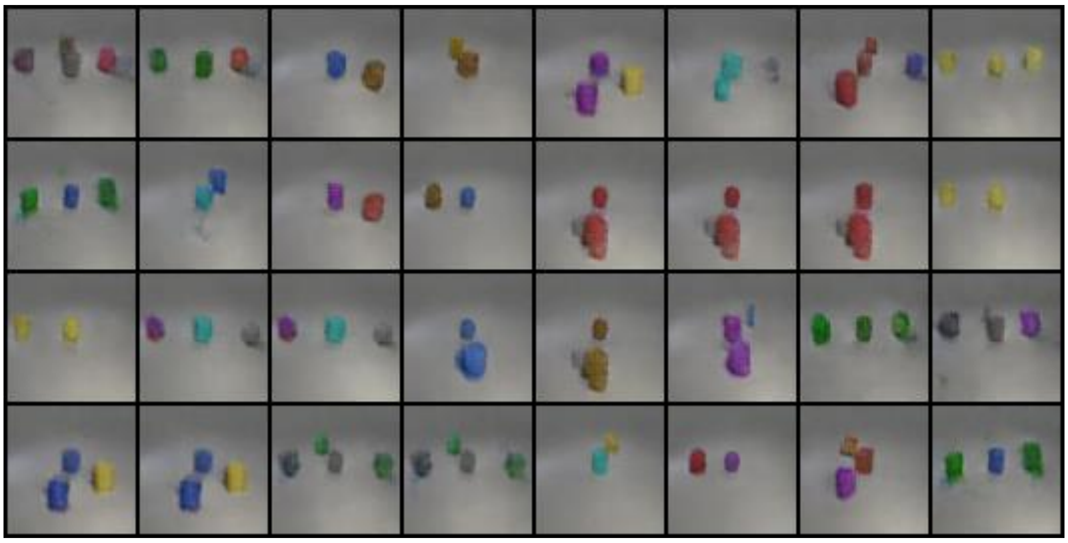
3. Results and discussion

3.1 Synthetic image grids

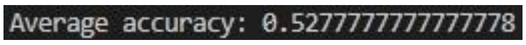
Synthetic image grids generated from test.json by DCGAN



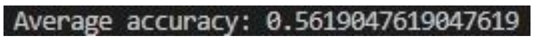
Synthetic image grids generated from new_test.json by DCGAN



Average accuracy is 0.5277777777777778 and 0.5619047619047619 respectively.

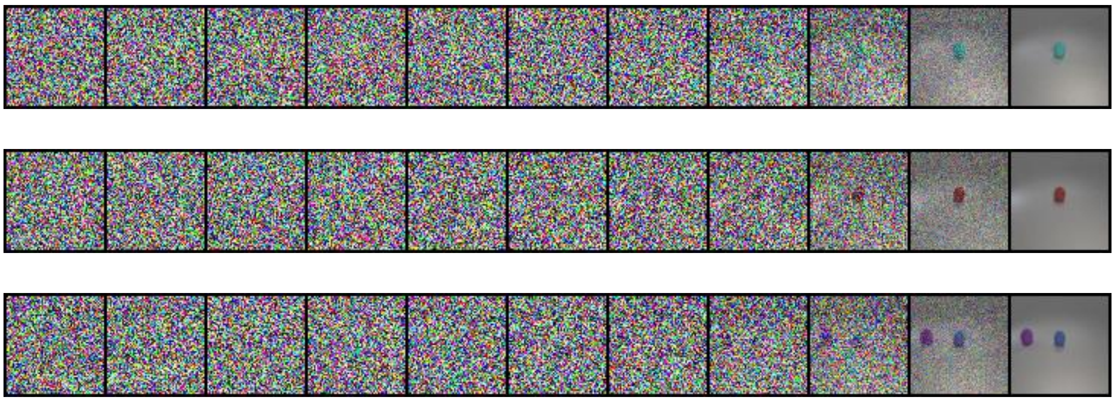


```
Average accuracy: 0.5277777777777778
```

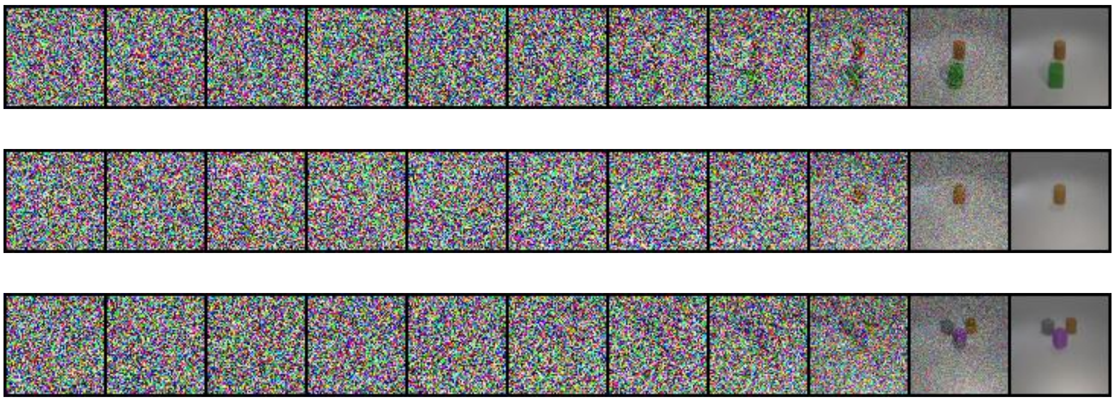


```
Average accuracy: 0.5619047619047619
```

Three randomly chosen examples of synthetic image grids generated from test.json by DDPM



Three randomly chosen examples of synthetic image grids generated from new_test.json by DDPM



3.2 Compare the advantages and disadvantages of the GAN and DDPM models

Advantages of GAN:

1. Training Speed: GANs generally train faster than DDPMs because they directly learn the mapping from noise to data distribution.

2. Quality of Generated Samples: GANs can produce high-quality, realistic samples, especially in image generation tasks.

3. Sampling Efficiency: Once trained, GANs can generate samples very quickly as it requires a single forward pass through the generator.

Disadvantages of GAN:

1. **Training Instability**: GANs are notorious for being difficult to train due to issues like mode collapse, where the generator produces limited variety in samples.

2. **Hyperparameter Sensitivity**: GAN requires careful tuning of hyperparameters and architectures of both the generator and discriminator.

3. **Evaluation Difficulty**: Quantitatively evaluating GANs is challenging, and metrics like Inception Score and FID are often used but have their own limitations.

Advantages of DDPM:

1. **Stability**: DDPMs are generally more stable to train compared to GANs because they rely on a likelihood-based framework rather than adversarial training.

2. **Theoretical Foundation**: DDPMs have a strong theoretical foundation rooted in non-equilibrium thermodynamics and variational inference.

3. **Mode Coverage**: DDPMs are better at covering the entire data distribution, reducing the risk of mode collapse seen in GANs.

4. **Flexible Sampling**: The reverse diffusion process allows for flexible sampling, potentially enabling better control over the generation process.

Disadvantages of DDPM:

1. **Training Time**: DDPMs typically require more training time due to the iterative nature of the diffusion process.
2. **Sampling Speed**: Generating samples with DDPMs is slower because it involves running multiple steps of the reverse diffusion process.
3. **Complexity**: The diffusion process involves more complex operations and

requires careful scheduling of noise addition and removal steps.

3.3 Discussion of your extra implementations or experiments

3.3.1 DCGAN

Conditional Generation: The generator model takes both noise (z) and conditional data (conds) as inputs. This allows the model to generate images conditioned on specific attributes, enhancing control over the generated output.

Resume training technique: I have implemented the resume training which enables the accuracy to achieve the lower number.

Adaptive Optimizers: It allows for choosing different optimizers for the discriminator (e.g., Adam or SGD), providing flexibility in tuning the training process.

3.3.2 DDPM

Context Conditioning: The forward method includes a context mask (context_mask) created using a Bernoulli dropout. This design introduces context conditioning during training, allowing the model to generate contextually relevant images.

Guidance Weight in Sampling: During the sampling process, a guidance weight (guide_w) is introduced to balance the influence of conditional and unconditional predictions, providing enhanced control over the generated samples.

Time Embedding and Label Embedding: The model uses time embedding and label embedding techniques to incorporate time steps and conditions into the denoising process.

Double Batch Mechanism: The sampling method uses a double batch mechanism to handle both conditional and unconditional data simultaneously. This helps in balancing the two types of data during the sampling process.