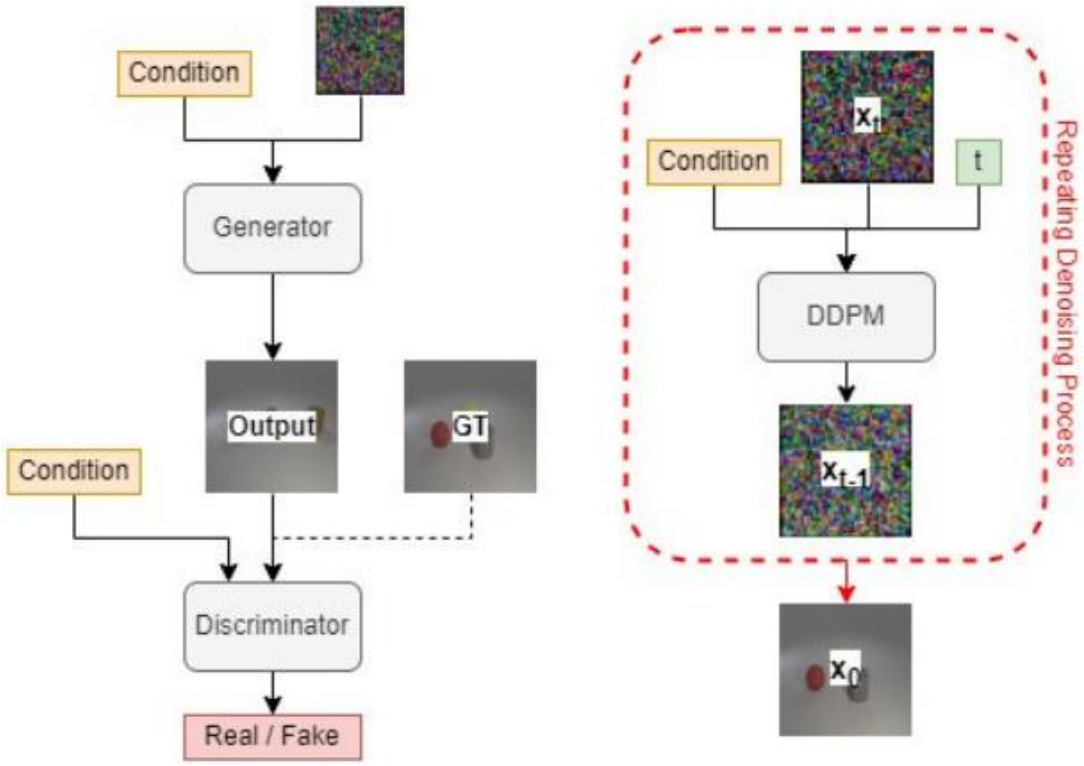
NYCU DL

Lab5 – Generative Models

312554006 羅名志

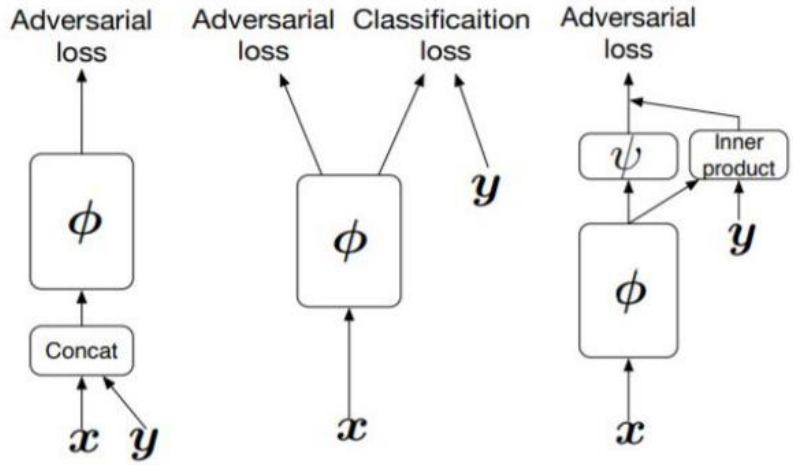
1. Introduction of Lab 6

In this lab, we need to implement two kinds of generative models: GAN (Generative Adversarial Network) and DDPM (Denoising Diffusion Probabilistic Model). The task is to use these generative models to output a clear image given specific conditions. The condition is the type of item that should appear in the image.

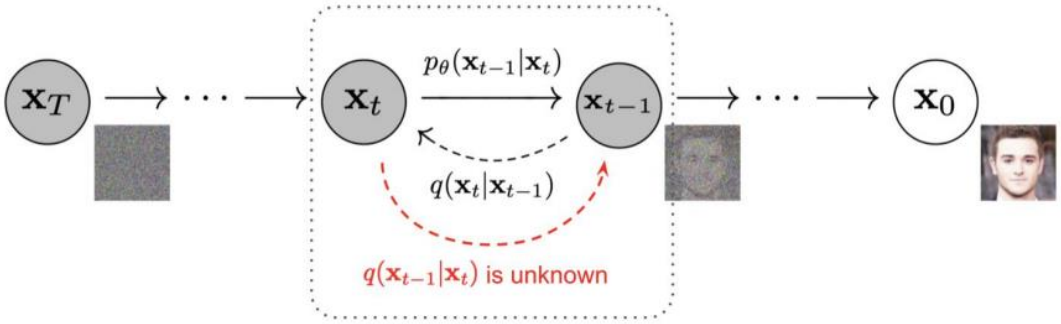


For GAN, the important part is to use the architecture of the generator and discriminator to progressively generate better images. The generator's goal is to create images that can deceive the discriminator, while the discriminator's goal is to accurately distinguish between real and fake images. Through the iterative "debate" process between these two components of the GAN, it can output a clear image.

Different kind of GAN architectures:

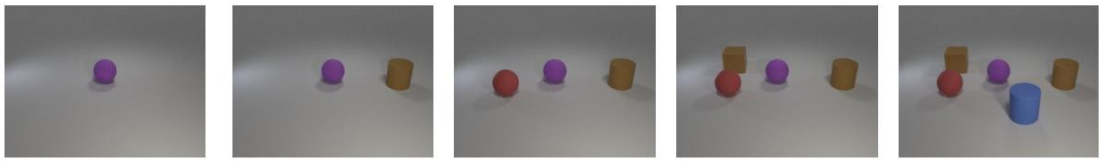


For DDPM, the model adds noise in the forward process and performs denoising in the backward process. Through this diffusion process, the model learns how to generate a good picture starting from Gaussian noise.



The training dataset consists of two files: one containing ground truth images and another containing their corresponding labels (objects that appear in the given image). For testing, there are two testing datasets, both of which only have labels.

Example of labels:



2. Implementation details

(1) Prepare dataset

For the dataloader file, the overall structure is quite similar to the dataloaders in previous labs. The details are shown below, where the variable img is the image that directly extracted by the training image directory.

```python
class Dataset_iclevr(torchData):

    def __init__(self, root, mode='train'):
        super().__init__()
        assert mode in ['train', 'test', 'new_test'], "There is no such mode !!!"
        self.mode = mode
        self.transform = transform(mode)
        self.files = get_filenames(mode)
        self.labels = get_labels(mode)
        self.root = root

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

    def __getitem__(self, index):
        # only training need to load the images
        img = torch.zeros((3, 64, 64))  # Default to a zero tensor with expected image shape
        if self.mode == 'train':
            file = self.files[index]
            img = Image.open(os.path.join(self.root, file)).convert('RGB')
            img = self.transform(img)

        label = self.labels[index]
        label = torch.Tensor(label)
        return img, label
```

However, the label is quite different, we first need to get the objects from the defined files and translate them to one hot encoding, as shown below:

```python
one_hot_list = []
for idx in range(len(labels)):
    one_hot_temp = np.zeros(24, dtype=int)
    for element in range(len(labels[idx])):
        one_hot_temp[label_mapping[labels[idx][element]]] = 1
    one_hot_list.append(one_hot_temp)  # Keep as numpy array

return np.array(one_hot_list)
```

(2) Implement a conditional GAN

a. Choose your conditional GAN architecture

The GAN architecture I have chosen here is the ACGAN (Auxiliary Classifier GAN). The key difference with ACGAN is that it includes an auxiliary loss to improve both training stability and performance. With this additional loss term, the discriminator not only considers the binary output (fake or real), but also the comparison between output embeddings and the ground truth one-hot encoding. The loss functions used are both BCELoss, and their details are shown below.

Loss function in discriminator:

```
self.adv_layer = nn.Sequential(nn.Linear(512 * 5 ** 2, 1), nn.Sigmoid())
self.aux_layer = nn.Sequential(nn.Linear(512 * 5 ** 2, self.num_classes), nn.Softmax(dim=1))
```

```
val = self.adv_layer(flat6)
label = self.aux_layer(flat6)
return val, label
```

Where the variable flat6 is the output of convolution networks and it is directly fed into the adv_layer and aux_layer to get both the adversarial loss and the auxiliary loss.

b. Design your generator and discriminator

- For the Generator, it contains a linear layer to combine the condition and the original image, along with five convolutional network blocks to extract features from the image and conditions. The details are shown below.

Forward function:

```
def forward(self, noise, labels):
    gen_input = torch.cat((self.label_embedding(labels), noise), -1)
    out = self.l1(gen_input)
    out = out.view(out.shape[0], 128, 1, 1)
    out = self.tconv2(out)
    out = self.tconv3(out)
    out = self.tconv4(out)
    out = self.tconv5(out)
    img = self.tconv6(out)
    return img
```

Model architecture:

Specifically, there are a ConvTranspose 2d layer, a BatchNorm layer and an activation function in each convolution block.

```
class Generator(nn.Module):
    def __init__(self, args):
        super(Generator, self).__init__()
        # define args
        self.num_classes = args.num_classes
        self.latent_dim = args.latent_dim
        self.image_size = args.image_size
        self.channels = 3  # Assuming RGB images

        # define forward function
        self.label_embedding = nn.Sequential(
            nn.Linear(self.num_classes, self.num_classes),
            nn.LeakyReLU(0.2, True)
        )
        self.init_size = 24
        self.l1 = nn.Sequential(nn.Linear(self.latent_dim + self.num_classes, 128))

        self.tconv2 = nn.Sequential(
            nn.ConvTranspose2d(128, 1024, 4, 1, 0, bias=False),
            nn.BatchNorm2d(1024),
            nn.ReLU(True),
        )
        self.tconv3 = nn.Sequential(
            nn.ConvTranspose2d(1024, 512, 4, 2, 1, bias=False),
            nn.BatchNorm2d(512),
            nn.ReLU(True),
        )
        self.tconv4 = nn.Sequential(
            nn.ConvTranspose2d(512, 128, 4, 2, 1, bias=False),
            nn.BatchNorm2d(128),
            nn.ReLU(True),
        )
        self.tconv5 = nn.Sequential(
            nn.ConvTranspose2d(128, 64, 4, 2, 1, bias=False),
            nn.BatchNorm2d(64),
            nn.ReLU(True),
        )
        self.tconv6 = nn.Sequential(
            nn.ConvTranspose2d(64, 3, 4, 2, 1, bias=False),
            nn.Tanh(),
        )
```

- For the discriminator, the structure is similar to that in generator. It contains 6 convolutional network blocks with two output layers: the adversarial output and the auxiliary output.

Forward function:

Where the variable flat6 is the output from those 6 convolutional blocks and resize to the input shape that can be fed into the output layers

```
def forward(self, img, labels):
    conv1 = self.conv1(img)
    conv2 = self.conv2(conv1)
    conv3 = self.conv3(conv2)
    conv4 = self.conv4(conv3)
    conv5 = self.conv5(conv4)
    conv6 = self.conv6(conv5)
    flat6 = conv6.view(-1, 5*5*512)
    val = self.adv_layer(flat6)
    label = self.aux_layer(flat6)
    return val, label
```

Model architecture:

Specifically, there are a Conv 2d layer, a BatchNorm layer, a dropout layer and an activation function in each convolution block.

```
self.conv1 = nn.Sequential(
    nn.Conv2d(3, 16, 3, 2, 1, bias=False),
    nn.LeakyReLU(0.2, inplace=True),
    nn.Dropout(0.5, inplace=False),
)
# Convolution 2
self.conv2 = nn.Sequential(
    nn.Conv2d(16, 32, 3, 1, 0, bias=False),
    nn.BatchNorm2d(32),
    nn.LeakyReLU(0.2, inplace=True),
    nn.Dropout(0.5, inplace=False),
)
# Convolution 3
self.conv3 = nn.Sequential(
    nn.Conv2d(32, 64, 3, 2, 1, bias=False),
    nn.BatchNorm2d(64),
    nn.LeakyReLU(0.2, inplace=True),
    nn.Dropout(0.5, inplace=False),
)
```

```
# Convolution 4
self.conv4 = nn.Sequential(
    nn.Conv2d(64, 128, 3, 1, 0, bias=False),
    nn.BatchNorm2d(128),
    nn.LeakyReLU(0.2, inplace=True),
    nn.Dropout(0.5, inplace=False),
)
# Convolution 5
self.conv5 = nn.Sequential(
    nn.Conv2d(128, 256, 3, 2, 1, bias=False),
    nn.BatchNorm2d(256),
    nn.LeakyReLU(0.2, inplace=True),
    nn.Dropout(0.5, inplace=False),
)
# Convolution 6
self.conv6 = nn.Sequential(
    nn.Conv2d(256, 512, 3, 1, 0, bias=False),
    nn.BatchNorm2d(512),
    nn.LeakyReLU(0.2, inplace=True),
    nn.Dropout(0.5, inplace=False),
)
```

Also, the adversarial output function is defined to output a probability through the softmax function, where the output value indicates how real the image is. The auxiliary function will output a class distribution to compare the overall loss between the ground truth one-hot encodings and its outputs.

```
self.adv_layer = nn.Sequential(nn.Linear(512 * 5 ** 2, 1), nn.Sigmoid())
self.aux_layer = nn.Sequential(nn.Linear(512 * 5 ** 2, self.num_classes), nn.Softmax(dim=1))
```

c. Choose your loss function

As describe in part a and part b, we have two loss function in the objective function and these two functions are both using BCEloss.

```
adversarial_loss = nn.BCELoss().to(args.device)
auxiliary_loss = nn.BCELoss().to(args.device)
```

The most important aspect of using auxiliary loss in our loss function is adjusting the weight for the adversarial loss and auxiliary loss. To prevent the discriminator from becoming too powerful in the early stages, which may cause "Mode Collapse," we adjust the weight of the auxiliary loss to be a hundred times that of the adversarial loss.

```
d_real_loss = (adversarial_loss(real_pred, valid) + 100*auxiliary_loss(real_aux, labels))/101
d_real_loss.backward()
```

d. Implement the training function and testing function

- Training:

We can divide the training part into two parts: training discriminator and training generator. Both loss functions follow the GAN objective functions.

Discriminator:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

Generator:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right)$$

For training discriminator, we first use the ground truth image and the image generated from the generator to train the discriminator, hoping it can identify whether the given image is true or not:

```
valid = 0.4 * torch.rand(batch_size, 1, device=args.device, requires_grad=False) + 0.6
real_pred, real_aux = discriminator(real_imgs, labels)
d_real_loss = (adversarial_loss(real_pred, valid) + 100*auxiliary_loss(real_aux, labels))/101
d_real_loss.backward()

# Fake images
fake = 0.4 * torch.rand(batch_size, 1, device=args.device, requires_grad=False)
fake_noise = torch.randn((batch_size, args.latent_dim), device=args.device)

fake_img = generator(fake_noise, labels)
fake_pred, fake_aux = discriminator(fake_img.detach(), labels)
d_fake_loss = (adversarial_loss(fake_pred, fake) + 100*auxiliary_loss(fake_aux, labels)) / 101
```

Here, we will calculate the loss function mentioned in previous parts for both fake images and true ones, and then calculate the total loss.

```
d_loss = 0.5 * (d_real_loss + d_fake_loss)
# d_loss.backward()
optimizer_D.step()

accuracy = evaluator.compute_acc(real_aux, labels)
```

For generator, its goal is to output a nearly true image. Follow by the same loss function setting, we have to decrease the loss output by the discriminator. Details are shown below:

```
# Train Generator
optimizer_G.zero_grad()
fake_noise = torch.randn((batch_size, args.latent_dim), device=args.device)
fake_img = generator(fake_noise, labels)

# Loss measures generator's ability to fool the discriminator
validity, pred_label = discriminator(fake_img, labels)
valid = torch.ones(batch_size, 1, device=args.device, requires_grad=False)
g_loss = (adversarial_loss(validity, valid) + 100*auxiliary_loss(pred_label, labels)) / 101

g_loss.backward()
optimizer_G.step()
```

- Testing:

We only need the generator part and generate an image by randomly sample a noise, then we calculate the loss by the evaluator:

```
generator.eval()
discriminator.eval()
evaluator = evaluation_model()
test_acc = 0
with torch.no_grad():
    for i, (imgs, labels) in enumerate(tqdm(test_dataloader, desc=f"Epoch {epoch}/{args.num_epochs}")):
        batch_size = imgs.size(0)
        labels = labels.to(args.device)
        noise = torch.randn((batch_size, args.latent_dim), device=args.device)
        fake_img = generator(noise, labels)
        # fake_img_normalized = normalize(fake_img)
        accuracy = evaluator.eval(fake_img, labels)
        # print(accuracy)
        test_acc = test_acc + accuracy
```

- Some tricks that I used in GAN:

  i. When calculating the adversarial loss from loss function while training discriminator, I won't directly use 1 to represent true image and 0 for fake. Instead, I random a number between 0.6 - 1 for true images and 0 - 0.4 for fake ones. By doing this, the model can add more noise and be trained more stably.

```
fake = 0.4 * torch.rand(batch_size, 1, device=args.device, requires_grad=False)
```

```
valid = 0.4 * torch.rand(batch_size, 1, device=args.device, requires_grad=False) + 0.6
```

ii. Training more times for generator than discriminator. The idea is that our main goal is to trained a good generator that can output a nearly true image. So, training more times for generator may help the overall performance. However, the result shows that it won't help too much when the number of total epochs is enough large.

```
for i in range(train_generator_epochs):
    fake_noise = torch.randn((batch_size, args.latent_dim), device=args.device)
    fake_img = generator(fake_noise, labels)

    # Loss measures generator's ability to fool the discriminator
    validity, pred_label = discriminator(fake_img, labels)
    valid = torch.ones(batch_size, 1, device=args.device, requires_grad=False)
    g_loss = (adversarial_loss(validity, valid) + 100*auxiliary_loss(pred_label, labels)) / 101

    g_loss.backward()
    optimizer_G.step()
```

(3) Implement a conditional GAN

a. Choose your conditional DDPM setting

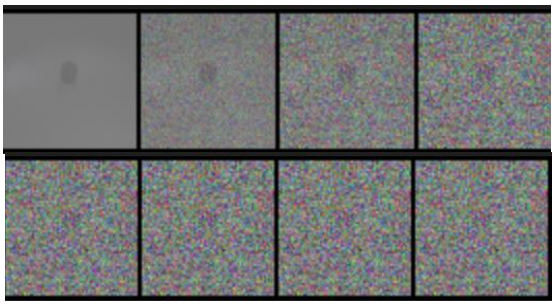
The mode I design is to consider all the given images, labels, and time index while implementing the DDPM.

Forward process:
According to the original DDPM paper, the training process involves randomly selecting a time step and adding noise to the image. In this lab, I use a U-Net architecture to predict the noise in the noisy image

The training algorithm and the noisy image from clear to noisy:

Algorithm 1 Training

1: **repeat**
2: $\mathbf{x}_0 \sim q(\mathbf{x}_0)$
3: $t \sim \text{Uniform}(\{1, \dots, T\})$
4: $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
5: Take gradient descent step on
$$\nabla_\theta \left\| \boldsymbol{\epsilon} - \boldsymbol{\epsilon}_\theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, t) \right\|^2$$
6: **until** converged



U-Net architecture I used:

```python
class ConditionalUNet(nn.Module):
    def __init__(self, num_classes=24, class_emb_size=24):
        super().__init__()

        # The embedding layer will map the class label to a
        self.class_emb1 = nn.Sequential(
            nn.Linear(num_classes, num_classes * 2),
            nn.GELU(),
            nn.Linear(num_classes * 2, num_classes * 2)
        )
        self.class_emb2 = nn.Sequential(
            nn.Linear(num_classes * 2, num_classes),
            nn.GELU(),
            nn.Linear(num_classes, num_classes)
        )
        self.time_emb1 = nn.Sequential(
            nn.Linear(1, num_classes * 2),
            nn.GELU(),
            nn.Linear(num_classes * 2, num_classes * 2)
        )
        self.time_emb2 = nn.Sequential(
            nn.Linear(num_classes * 2, num_classes),
            nn.GELU(),
            nn.Linear(num_classes, num_classes)
        )
```

```python
# Self.model is an unconditional UNet with extra input channels to accept the conditi
self.model = UNet2DModel(
    sample_size=64,  # the target image resolution
    in_channels=3 + 1* class_emb_size,  # Additional input channels for class cond.
    out_channels=3,  # the number of output channels
    layers_per_block=2,  # how many ResNet layers to use per UNet block
    block_out_channels=(32, 64, 64),
    down_block_types=(
        "DownBlock2D",  # a regular ResNet downsampling block
        "AttnDownBlock2D",  # a ResNet downsampling block with spatial self-attentio
        "AttnDownBlock2D",
    ),
    up_block_types=(
        "AttnUpBlock2D",
        "AttnUpBlock2D",  # a ResNet upsampling block with spatial self-attention
        "UpBlock2D",  # a regular ResNet upsampling block
    ),
)
```

The U-Net model here will first embed the time index and condition lables, then feed them all with the noisy images to the Upsample and Downsample block.

Lastly. The output will be a predicted noise which its size is the same as input image.

Specifically, the U-Net structure here only contains 2 layers per block where seems to be not enough.

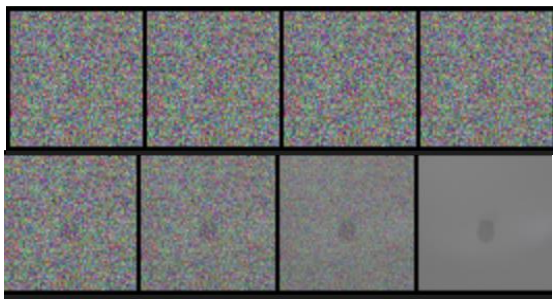
```
    layers_per_block=2,  # how many ResNet layers to use per UNet block
    block_out_channels=(32, 64, 64),
    down_block_types=(
        "DownBlock2D",  # a regular ResNet downsampling block
        "AttnDownBlock2D",  # a ResNet downsampling block with spatial self-attention
        "AttnDownBlock2D",
    ),
    up_block_types=(
        "AttnUpBlock2D",
        "AttnUpBlock2D",  # a ResNet upsampling block with spatial self-attention
        "UpBlock2D",  # a regular ResNet upsampling block
    ),
)
```

Sampling process:
According to the original DDPM paper, I sample a noise and recursively denoise the image back to the original predicted image. The details are shown below:

The Sampling algorithm and denoise process from noisy to clear:

Algorithm 2 Sampling

$\mathbf{x}_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
for $t = T, \dots, 1$ **do**
  $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ if $t > 1$, else $\mathbf{z} = \mathbf{0}$
  $\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}} \left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}} \boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t) \right) + \sigma_t \mathbf{z}$
end for
return $\mathbf{x}_0$



Note that the sampling process will not need to train any blocks in the model. It directly uses the trained U-Net to predict the noise each stage and deduct it.

b. Design your noise schedule, time embeddings

i. Noise schedule

I use two kinds of noise schedule: Linear noise and Cosine schedule

Linear:

```
self.betas = torch.linspace(0.0001, 0.02, timesteps, device=device)
self.alphas = 1.0 - self.betas
self.alpha_hat = torch.cumprod(self.alphas, dim=0)
```

The alpha values decrease linearly over time. This means that as the time index approaches the total number of time steps, alpha will be smaller. This gradual decrease ensures that more noise is added progressively at each step.

Cosine:

```
x = torch.linspace(0, timesteps, steps, device=device)
alphas_cumprod = torch.cos(((x / timesteps) + s) / (1 + s) * (np.pi / 2)) ** 2
alphas_cumprod = alphas_cumprod / alphas_cumprod[0]

self.betas = 1 - (alphas_cumprod[1:] / alphas_cumprod[:-1])
self.alphas = 1 - betas
self.alpha_hat = torch.cumprod(alphas, dim=0)
```

The alpha values decrease following a cosine function. This results in a more gradual decrease at the beginning and the end, with a steeper decrease in the middle. When the time index is close to the total number of time steps, alpha will be very small, approaching zero more smoothly compared to the linear schedule.

ii. time embeddings

As mentioned in part a, the time index will be fed into the U-Net model. And the time embedding functions are as follows:

```
self.time_emb1 = nn.Sequential(
    nn.Linear(1, num_classes * 2),
    nn.GELU(),
    nn.Linear(num_classes * 2, num_classes * 2)
)
self.time_emb2 = nn.Sequential(
    nn.Linear(num_classes * 2, num_classes),
    nn.GELU(),
    nn.Linear(num_classes, num_classes)
)
```

Where the num_calsses variable is set to be 24.

c. Choose your loss function

The loss function in the training part is MSEloss.

```
criterion = nn.MSELoss().to(args.device)
```

d. Implement the training function, testing function

- Training function

```
for epoch in range(args.num_epochs):
    tot_loss = 0
    ddpm.train()
    for i, (imgs, labels) in enumerate(tqdm(train_dataloader, desc=f"Epoch {epoch}/{args.num_epochs}")):
        batch_size = imgs.size(0)
        real_imgs = imgs.to(args.device)
        labels = labels.to(args.device)

        t = torch.randint(0, timesteps, (batch_size,), device=args.device).long()
        noise = torch.randn_like(real_imgs).to(args.device)
        xt = ddpm.forward_diffusion(real_imgs, t, noise)
        noise_pred = ddpm.unet(xt, t, labels)
        loss = criterion(noise, noise_pred)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        tot_loss += loss.item()
```

As described in the previous parts, the training goal is to minimize the loss between predicted noise and true noise. This depends on a strong U-Net architecture to output a nearly true noise.

As shown in the code snippet which follow the training code from DDPM, I first sample a time index t and noise. Then feed the image, time, and time index to my U-Net. Lastly, calculate the loss between prediction from U-Net and true noise.

```
def forward_diffusion(self, x0, t, noise=None):
    bc, c, h, w = x0.shape
    if noise == None:
        noise = torch.randn_like(x0)
    alpha_hat_t = self.alpha_hat[t]
    noisy = alpha_hat_t.sqrt().reshape(bc, 1, 1, 1) * x0 + (1 - alpha_hat_t).sqrt().reshape(bc, 1, 1, 1) * noise
    return noisy
```

Also, the noisy image is generated by the same way defined in DDPM paper.

- Testing

Following the DDPM paper, I first sample Gaussian noise and then iteratively move backward in time to recursively transform the noisy image back to a clear one.

```
with torch.no_grad():
    for i, (imgs, labels) in enumerate(tqdm(test_dataloader, desc=f"Epoch {epoch}/{args.num_epochs}")):
        batch_size = labels.size(0)
        samples = sample_images(ddpm, labels, num_samples=batch_size)
        accuracy = evaluator.eval(samples, labels)
        # print(accuracy)
        test_acc = test_acc + accuracy
```

```
def sample_images(ddpm, condition, num_samples=1):
    ddpm.eval()
    condition = condition.to(args.device)
    xt = torch.randn((num_samples, 3, 64, 64), device=args.device)

    for t in reversed(range(ddpm.timesteps)):
        t_tensor = torch.full((num_samples,), t, dtype=torch.long, device=args.device)
        n, xt = ddpm.reverse_diffusion(xt, t_tensor, condition)

    return xt
```

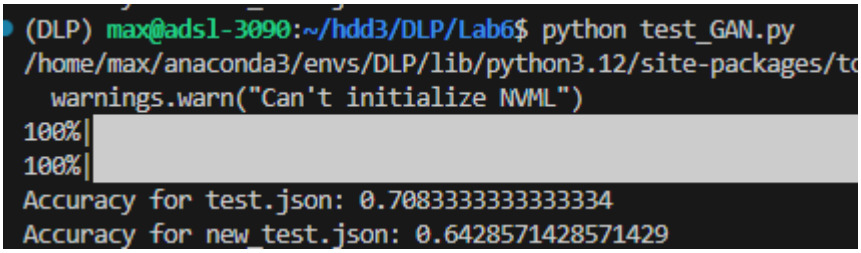
Also, the reverse function defined in sample_images function is as follow; all the setting is actually the same as DDPM algorithm:

```
denomiantor = 1/(alpha_t.sqrt().reshape(bc, 1, 1, 1))
noise_weight = (1-alpha_t.reshape(bc, 1, 1, 1))/(torch.sqrt(1-alpha_hat_t.reshape(bc, 1, 1, 1)))
random_weight = torch.sqrt(beta_t.reshape(bc, 1, 1, 1))
z = torch.randn_like(xt).to(device)
return noise_pred, (denomiantor * (xt-noise_pred*noise_weight) + (z*random_weight))
```

3. Results and discussion

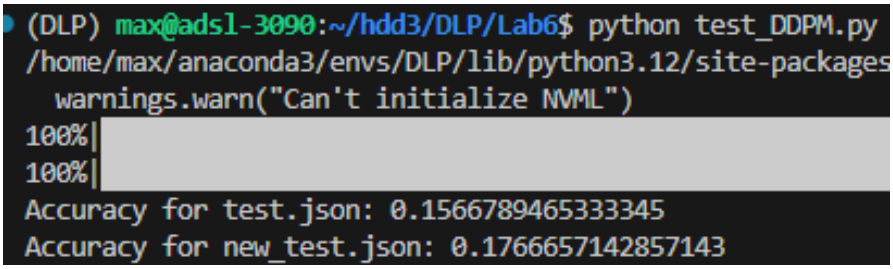
(1) Results

a. GAN



For the file test.json, the accuracy is around 0.7, while for the file new_test.json, the accuracy is around 0.65.

b. DDPM



For the file test.json, the accuracy is only around 0.15, and for the file new_test.json, the accuracy is around 0.17.

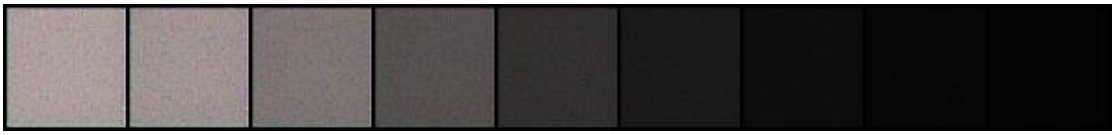
The possible reason for the poor results using DDPM here may be the inadequate design of my U-Net, where the network cannot accurately predict the noise from a noisy image.

(2) Show your synthetic image grids

a. Images from GAN, test.json

b. Images from GAN, new_test.json

c. Denoising process image, test.json



(3) Compare the advantages and disadvantages of the GAN and DDPM models

a.

The result of DDPM on testing dataset is worse than GAN, it is probably because the U-Net architecture is too simple, where the model can effectively learn the noise pattern within a noisy image. Therefore, the results here is not reasonable.

b.

Ideally, the DDPM model is more powerful, here are some reasons:

i. Training Stability

GANs consist of two neural networks—a generator and a discriminator—that compete in a zero-sum game. This adversarial setup can lead to instability in training, often causing issues like mode collapse, where the generator produces limited varieties of outputs. DDPMs, on the other hand, use a simpler and more stable training process. They progressively denoise data over a series of steps, reducing complexity and leading to more stable convergence.

ii. Quality of Generated Samples

DDPMs generate samples through a diffusion process, starting from pure noise and iteratively refining the image. This method often results in high-quality, diverse outputs, closely matching the training data distribution. GANs can struggle to capture the full diversity of the training data, especially in complex datasets, leading to artifacts and less realistic images.

c. Pros and Cons of GAN and DDPM

Although DDPMs have demonstrated superiority in generating high-quality images, there are still notable advantages to consider with GANs. Firstly, GANs boast faster training times compared to DDPMs; training a DDPM can take more than five times longer than a GAN, making GANs more efficient for rapid model development. Additionally, GANs offer flexibility in architecture, allowing for easy adaptation to various models such as ACGAN, DCGAN, and others. However, GANs are plagued by issues like mode collapse and training instability,

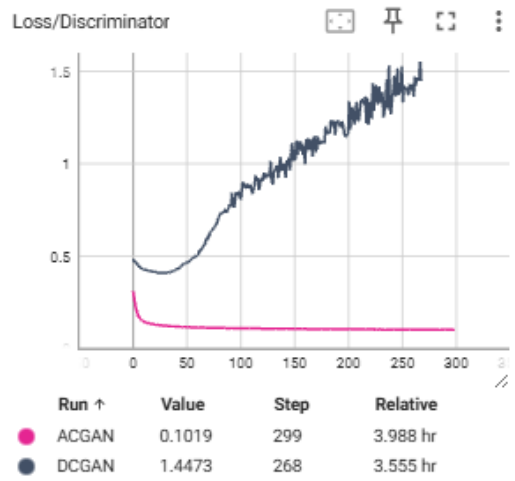
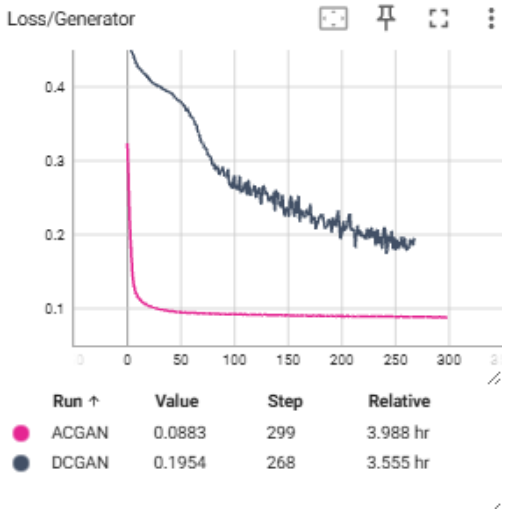
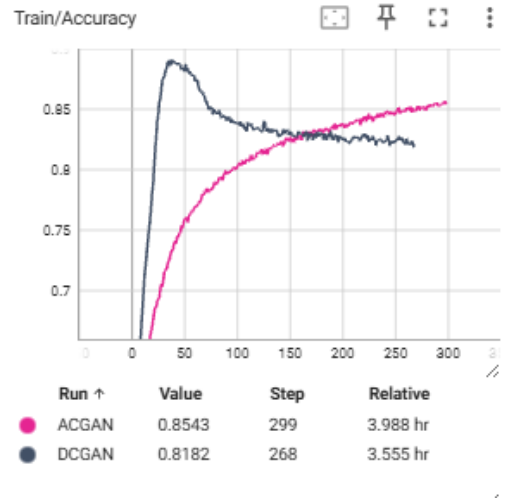
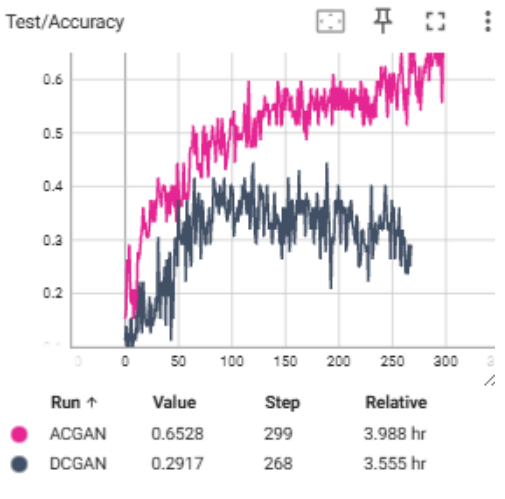
which are mitigated in DDPMs, making them a preferable choice in certain scenarios.

(4) Discussion of your extra implementations or experiments

(a) DCGAN vs. ACGAN

Unlike ACGAN, DCGAN does not incorporate class conditioning, focusing solely on generating high-resolution, realistic images from random noise vectors without consideration for specific classes or attributes.

Therefore, the result of DCGAN is little worse than using ACGAN in this scenario.

Loss / Discriminator	Loss / Generator
	
Train / Accuracy	Test / Accuracy
	

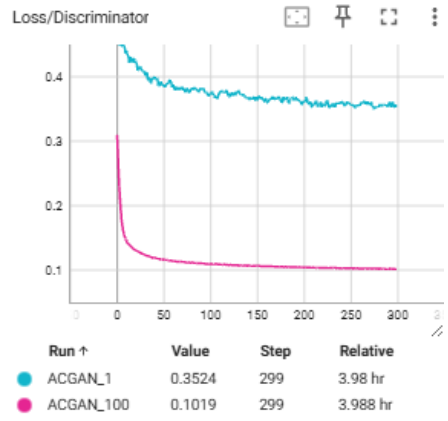
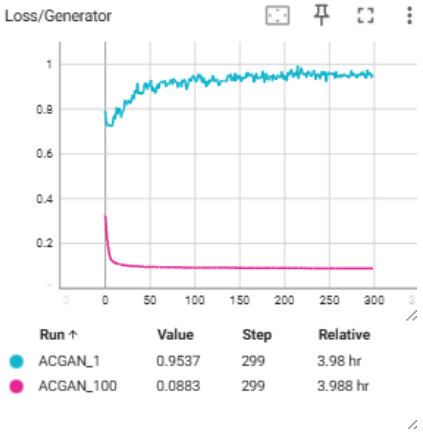
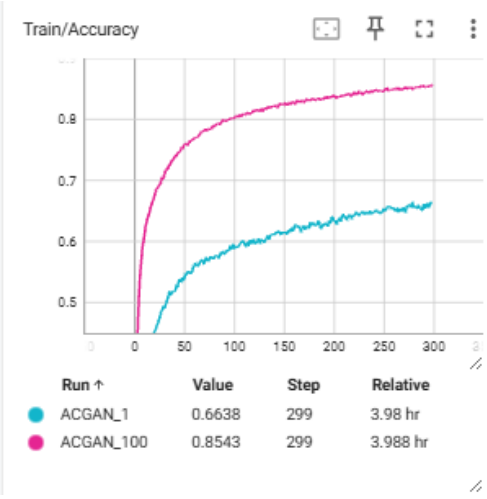
As illustrated in the comparison above, employing DCGAN without any auxiliary loss term may lead to inferior performance and a destabilized training process. This instability can manifest as an increase in the discriminator's loss during the middle of training and a subsequent decline in training accuracy.

(b) Adjust auxiliary weight vs. Not adjust

Loss = adversarial loss + weight * auxiliary loss

Eg. The one can reach best performance is weight = 100 as shown below

```
d_fake_loss = (adversarial_loss(fake_pred, fake) + 100*auxiliary_loss(fake_aux, labels)) / 101
```

Loss / Discriminator	Loss / Generator
	
Train / Accuracy	**Test / Accuracy**
	

Firstly, both results are more stable than those using DCGAN, proving that the auxiliary loss term can indeed aid in model learning.

Secondly, as depicted in the image, a weight of 100 yields better results than a weight of 1. This is likely because the model can focus more on overall performance rather than solely distinguishing between true and fake images. Additionally, this may help prevent mode collapse issues.

5. Reference

1. GAN architecture:
https://github.com/clvrai/ACGAN/PyTorch/blob/master/network.py
2. Diffusion architecture:
https://medium.com/@brianpulfer/enerating-images-with-ddpms-a-pytorch-implementation-cef5a2ba8cb1
3. U-Net architecture
https://github.com/huggingface/diffusion-models-class/blob/main/unit2/02_class_conditioned_diffusion_model_example.ipynb
4. ACGAN paper
Odena, Augustus, Christopher Olah, and Jonathon Shlens. "Conditional image synthesis with auxiliary classifier gans." International conference on machine learning. PMLR, 2017.
5. DDPM paper
Ho, Jonathan, Ajay Jain, and Pieter Abbeel. "Denoising diffusion probabilistic models." Advances in neural information processing systems 33 (2020): 6840-6851.
6. TA's spec and DL slides
7. ChatGPT: https://chat.openai.com