DLP lab6

Introduction

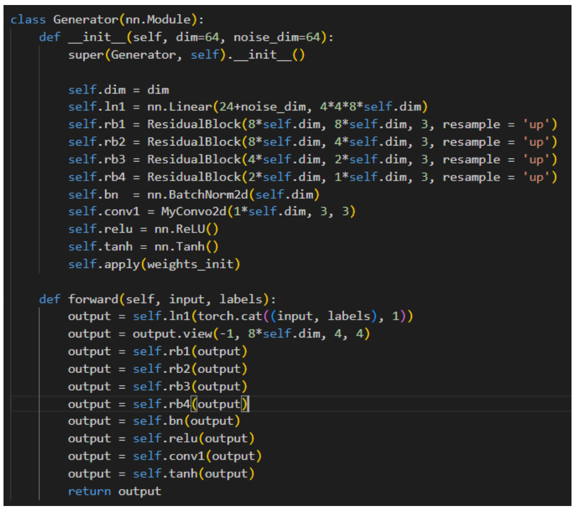
In this lab, we'll implement two powerful generative models: GAN and DDPM. Our goal is to generate synthetic images in iclevr dataset based on multi-label conditions. Additionally, we'll evaluate your generated images using a pre-trained classifier.

Implementation details

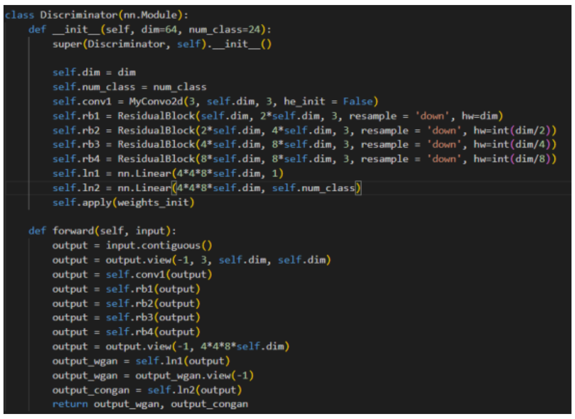
GAN

Model – deeper generator and discriminator using residual blocks

For the generator, concat the noise and conditions and send it to a linear layer to have the generator condition on the labels. For the discriminator, I add an additional linear layer before the output to let the model classify the result.



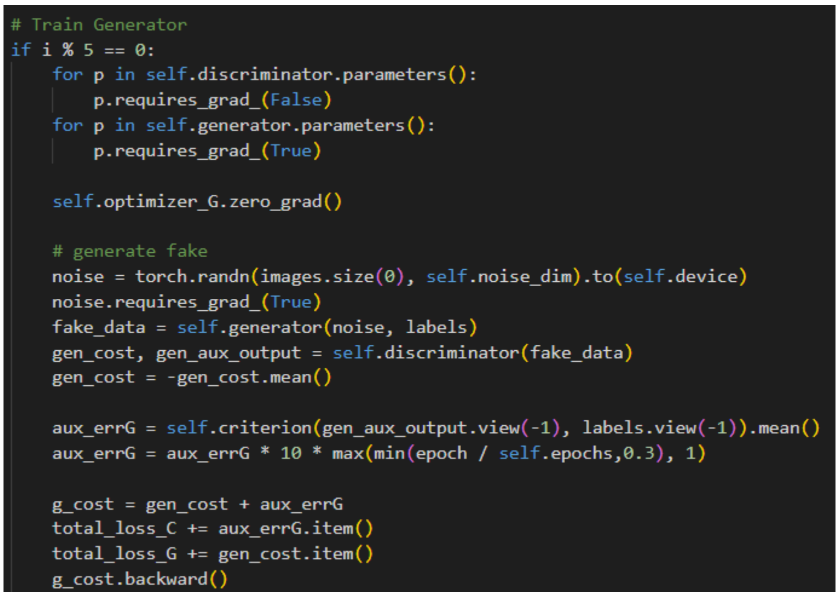
```python
class Generator(nn.Module):
    def __init__(self, dim=64, noise_dim=64):
        super(Generator, self).__init__()

        self.dim = dim
        self.ln1 = nn.Linear(24+noise_dim, 4*4*8*self.dim)
        self.rb1 = ResidualBlock(8*self.dim, 8*self.dim, 3, resample = 'up')
        self.rb2 = ResidualBlock(8*self.dim, 4*self.dim, 3, resample = 'up')
        self.rb3 = ResidualBlock(4*self.dim, 2*self.dim, 3, resample = 'up')
        self.rb4 = ResidualBlock(2*self.dim, 1*self.dim, 3, resample = 'up')
        self.bn  = nn.BatchNorm2d(self.dim)
        self.conv1 = MyConvo2d(1*self.dim, 3, 3)
        self.relu = nn.ReLU()
        self.tanh = nn.Tanh()
        self.apply(weights_init)

    def forward(self, input, labels):
        output = self.ln1(torch.cat((input, labels), 1))
        output = output.view(-1, 8*self.dim, 4, 4)
        output = self.rb1(output)
        output = self.rb2(output)
        output = self.rb3(output)
        output = self.rb4(output)
        output = self.bn(output)
        output = self.relu(output)
        output = self.conv1(output)
        output = self.tanh(output)
        return output
```



```python
class Discriminator(nn.Module):
    def __init__(self, dim=64, num_class=24):
        super(Discriminator, self).__init__()

        self.dim = dim
        self.num_class = num_class
        self.conv1 = MyConvo2d(3, self.dim, 3, he_init = False)
        self.rb1 = ResidualBlock(self.dim, 2*self.dim, 3, resample = 'down', hw=dim)
        self.rb2 = ResidualBlock(2*self.dim, 4*self.dim, 3, resample = 'down', hw=int(dim/2))
        self.rb3 = ResidualBlock(4*self.dim, 8*self.dim, 3, resample = 'down', hw=int(dim/4))
        self.rb4 = ResidualBlock(8*self.dim, 8*self.dim, 3, resample = 'down', hw=int(dim/8))
        self.ln1 = nn.Linear(4*4*8*self.dim, 1)
        self.ln2 = nn.Linear(4*4*8*self.dim, self.num_class)
        self.apply(weights_init)

    def forward(self, input):
        output = input.contiguous()
        output = output.view(-1, 3, self.dim, self.dim)
        output = self.conv1(output)
        output = self.rb1(output)
        output = self.rb2(output)
        output = self.rb3(output)
        output = self.rb4(output)
        output = output.view(-1, 4*4*8*self.dim)
        output_wgan = self.ln1(output)
        output_wgan = output_wgan.view(-1)
        output_congan = self.ln2(output)
        return output_wgan, output_congan
```

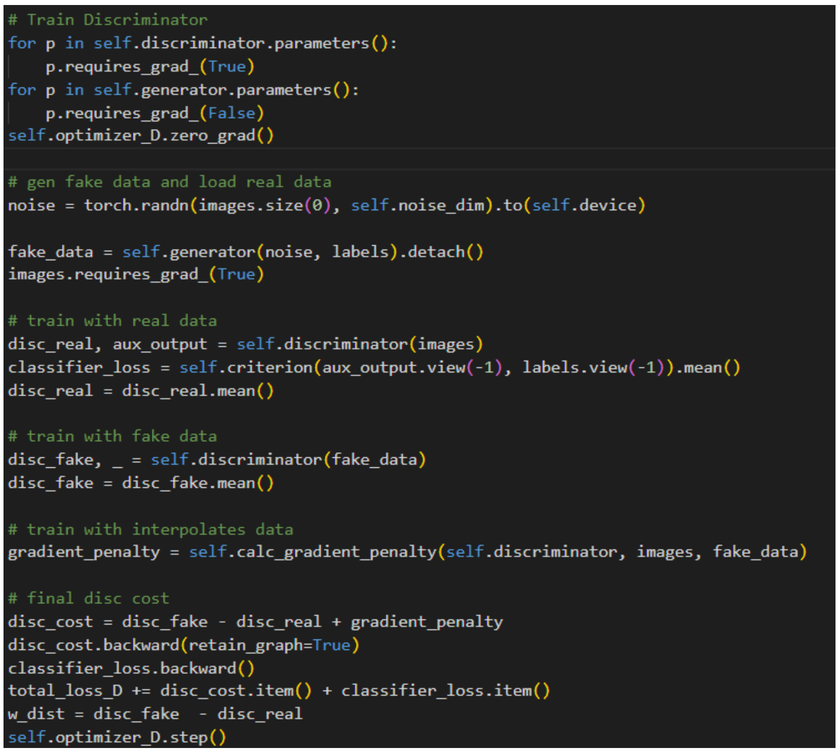
Train – WGAN-gp

Train generator every 5 batch. Loss: generator loss + classification loss (ratio grows as epoch grows and is decided by experiment) (minimum threshold 0.3).

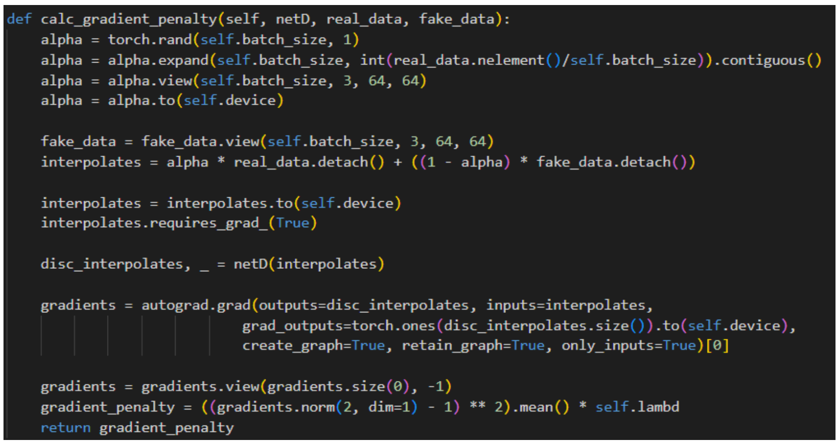


```python
# Train Generator
if i % 5 == 0:
    for p in self.discriminator.parameters():
        p.requires_grad_(False)
    for p in self.generator.parameters():
        p.requires_grad_(True)

    self.optimizer_G.zero_grad()

    # generate fake
    noise = torch.randn(images.size(0), self.noise_dim).to(self.device)
    noise.requires_grad_(True)
    fake_data = self.generator(noise, labels)
    gen_cost, gen_aux_output = self.discriminator(fake_data)
    gen_cost = -gen_cost.mean()

    aux_errG = self.criterion(gen_aux_output.view(-1), labels.view(-1)).mean()
    aux_errG = aux_errG * 10 * max(min(epoch / self.epochs,0.3), 1)

    g_cost = gen_cost + aux_errG
    total_loss_C += aux_errG.item()
    total_loss_G += gen_cost.item()
    g_cost.backward()
```

Instead of clipping in WGAN, I use the gradient penalty as implemented in WGAN



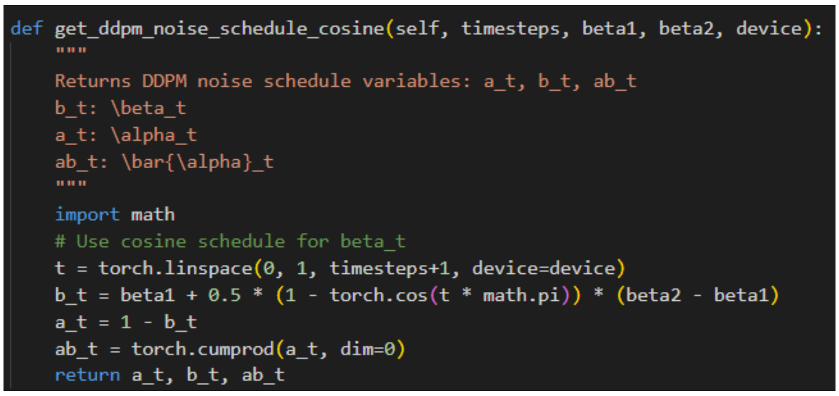
```python
# Train Discriminator
for p in self.discriminator.parameters():
    p.requires_grad_(True)
for p in self.generator.parameters():
    p.requires_grad_(False)
self.optimizer_D.zero_grad()

# gen fake data and load real data
noise = torch.randn(images.size(0), self.noise_dim).to(self.device)

fake_data = self.generator(noise, labels).detach()
images.requires_grad_(True)

# train with real data
disc_real, aux_output = self.discriminator(images)
classifier_loss = self.criterion(aux_output.view(-1), labels.view(-1)).mean()
disc_real = disc_real.mean()

# train with fake data
disc_fake, _ = self.discriminator(fake_data)
disc_fake = disc_fake.mean()

# train with interpolates data
gradient_penalty = self.calc_gradient_penalty(self.discriminator, images, fake_data)

# final disc cost
disc_cost = disc_fake - disc_real + gradient_penalty
disc_cost.backward(retain_graph=True)
classifier_loss.backward()
total_loss_D += disc_cost.item() + classifier_loss.item()
w_dist = disc_fake  - disc_real
self.optimizer_D.step()
```



```python
def calc_gradient_penalty(self, netD, real_data, fake_data):
    alpha = torch.rand(self.batch_size, 1)
    alpha = alpha.expand(self.batch_size, int(real_data.nelement()/self.batch_size)).contiguous()
    alpha = alpha.view(self.batch_size, 3, 64, 64)
    alpha = alpha.to(self.device)

    fake_data = fake_data.view(self.batch_size, 3, 64, 64)
    interpolates = alpha * real_data.detach() + ((1 - alpha) * fake_data.detach())

    interpolates = interpolates.to(self.device)
    interpolates.requires_grad_(True)

    disc_interpolates, _ = netD(interpolates)

    gradients = autograd.grad(outputs=disc_interpolates, inputs=interpolates,
                              grad_outputs=torch.ones(disc_interpolates.size()).to(self.device),
                              create_graph=True, retain_graph=True, only_inputs=True)[0]

    gradients = gradients.view(gradients.size(0), -1)
    gradient_penalty = ((gradients.norm(2, dim=1) - 1) ** 2).mean() * self.lambd
    return gradient_penalty
```

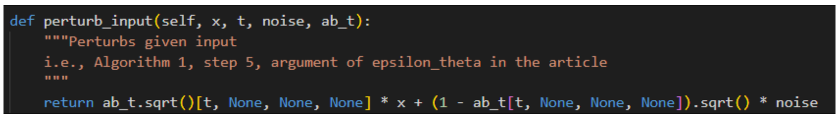
DDPM

Template from pytorch-ddpm. I set total step = 500 with cosine scheduling.

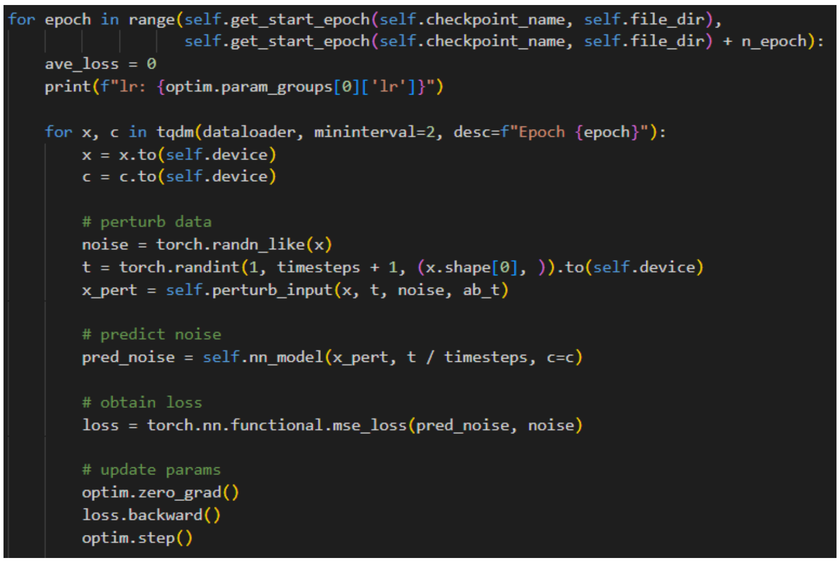


```python
def get_ddpm_noise_schedule_cosine(self, timesteps, beta1, beta2, device):
    """
    Returns DDPM noise schedule variables: a_t, b_t, ab_t
    b_t: \beta_t
    a_t: \alpha_t
    ab_t: \bar{\alpha}_t
    """
    import math
    # Use cosine schedule for beta_t
    t = torch.linspace(0, 1, timesteps+1, device=device)
    b_t = beta1 + 0.5 * (1 - torch.cos(t * math.pi)) * (beta2 - beta1)
    a_t = 1 - b_t
    ab_t = torch.cumprod(a_t, dim=0)
    return a_t, b_t, ab_t
```

Training



```python
def perturb_input(self, x, t, noise, ab_t):
    """Perturbs given input
    i.e., Algorithm 1, step 5, argument of epsilon_theta in the article
    """
    return ab_t.sqrt()[t, None, None, None] * x + (1 - ab_t[t, None, None, None]).sqrt() * noise
```



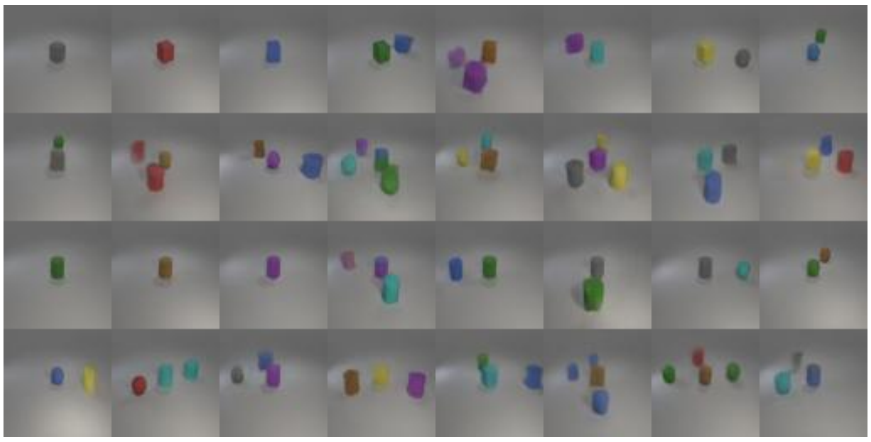
```python
for epoch in range(self.get_start_epoch(self.checkpoint_name, self.file_dir),
                   self.get_start_epoch(self.checkpoint_name, self.file_dir) + n_epoch):
    ave_loss = 0
    print(f"lr: {optim.param_groups[0]['lr']}")

    for x, c in tqdm(dataloader, mininterval=2, desc=f"Epoch {epoch}"):
        x = x.to(self.device)
        c = c.to(self.device)

        # perturb data
        noise = torch.randn_like(x)
        t = torch.randint(1, timesteps + 1, (x.shape[0], )).to(self.device)
        x_pert = self.perturb_input(x, t, noise, ab_t)

        # predict noise
        pred_noise = self.nn_model(x_pert, t / timesteps, c=c)

        # obtain loss
        loss = torch.nn.functional.mse_loss(pred_noise, noise)

        # update params
        optim.zero_grad()
        loss.backward()
        optim.step()
```

Results and discussion

Synthetic image grids

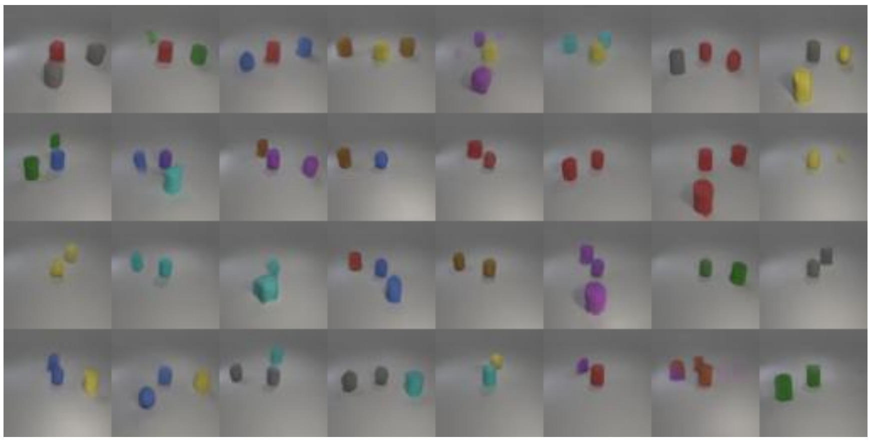
GAN

Test.json



Acc 0.9722222222222222

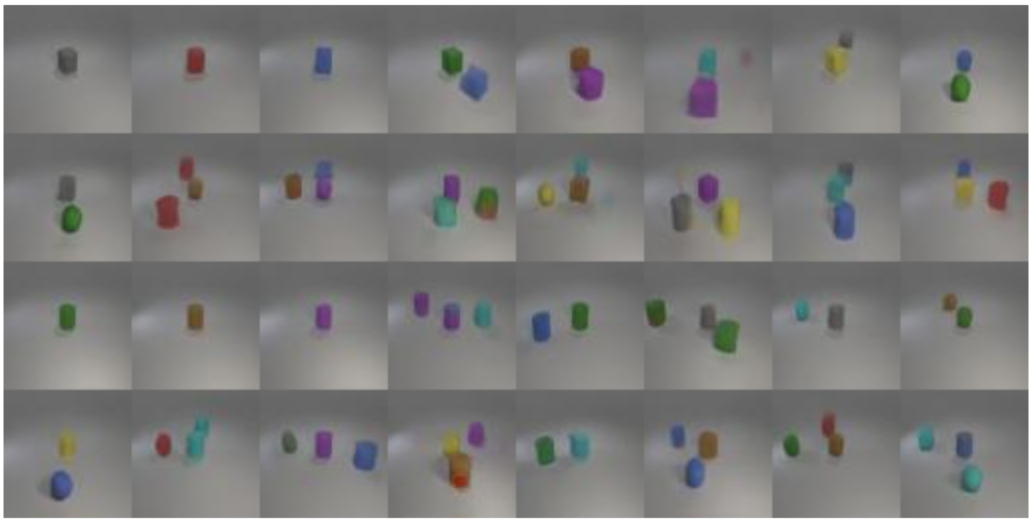
New_test.json



Acc 0.8452380952380952

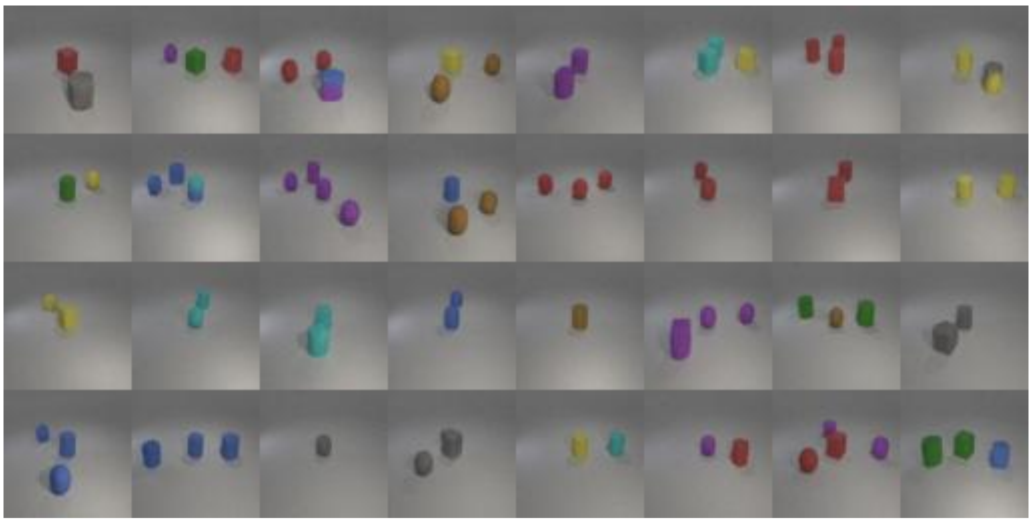
DDPM

Test.json



Acc 0.930555555555556

New_test.json



Acc 0.8095238095238095

Denoising

GAN vs. DDPM

In my own experience, GAN will be able to generate good results until about 200 epoch, but DDPM will generate normal results at about 20 epoch. In the first 200 epoch, the GAN generator can only generate noises, but DDPM can generate normal pictures (while it looks weird, but it is not pure noise).

Conditioning to GAN is easier than conditioning for DDPM. DDPM can condition well if there is only one object, but if there is more than one object, it will generate the wrong object, and sometimes 4 objects. I use mse as the loss function for DDPM, and it makes the model not able to learn well on shapes (circle ball and cylinder have lots of overlapping areas). This will not be a problem if you use a discriminator that knows exactly what the shape is instead of summing the loss of pixel values.

Extra implementations or experiments

GAN is harder to train compared to DDPM. At first, I tried the model architecture in pytorch website, but it shows that it is not strong enough to generate high quality images. Hence, I stack the residual blocks to make the model deeper, but this makes the generator hard to learn (if discriminator is too good, generator can't learn well), so I modified to originall GAN to WGAN-gp.

As for noise scheduling for DDPM, I tried linear scheduling in the beginning. I generate the denoising process and store the picture every 20 steps. It shows that the steps before 400 steps generate pure noise only, so I change it to cosine scheduing, making the model to avoid learning the useless steps.