This project explores image generation using generative models, with a focus on understanding how they work under the hood and how to implement them from scratch. The goal is to experiment with different model architectures and training strategies to perform image generation both with and without conditioning (as illustrated in the image above, which shows a generation conditioned by another image).

The work was carried out using an interactive Jupyter notebook, which combines:

- Code implementations of core generative techniques.
- Explanatory notes and reflections to demonstrate a clear understanding of key concepts.
- Hands-on exercises where models are built or completed to explore practical aspects of training generative models.

This project was an opportunity to strengthen my understanding of generative AI through practical implementation, critical thinking, and iterative experimentation—all documented in a single, cohesive notebook.

Part 1: DC-GAN

In this part, we aim to learn and understand the basic concepts of **Generative Adversarial Networks** through a DCGAN.

We want to generate handwritten digits using the MNIST dataset. It is available within the torvision package (https://pytorch.org/vision/stable/generated/torchvision.datasets.MNIST.html#torchvision.datasets.MNIST)

```
import argparse
import os
import random
import torch
import torch.nn as nn
import torch.nn.parallel
import torch.optim as optim
import torch.utils.data
import torchvision.datasets as dset
import torchvision.transforms as transforms
import torchvision.utils as vutils
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from IPython.display import HTML

# Set random seed for reproducibility
manualSeed = 999
#manualSeed = random.randint(1, 10000) # use if you want new results
print("Random Seed: ", manualSeed)
random.seed(manualSeed)
```

```
torch.manual_seed(manualSeed)

# Root directory for dataset
dataroot = "data/mnist"

# Number of workers for dataloader
workers = 2

# Batch size during training
batch_size = 128

# Spatial size of training images. All images will be resized to this
#   size using a transformer.
image_size = 64

# Number of channels in the training images. For color images this is
3
nc = 1

# Size of z latent vector (i.e. size of generator input)
nz = 100

# Size of feature maps in generator
ngf = 64

# Size of feature maps in discriminator
ndf = 64

# Number of training epochs
num_epochs = 5

# Learning rate for optimizers
lr = 0.0002

# Beta1 hyperparameter for Adam optimizers
beta1 = 0.5

# Number of GPUs available. Use 0 for CPU mode.
ngpu = 1

dataset = dset.MNIST(root=dataroot, download=True,
transform=transforms.Compose([
    transforms.Resize(image_size),
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
]))

dataloader = torch.utils.data.DataLoader(dataset,
batch_size=batch_size,
                                         shuffle=True,
num_workers=workers)
```

```
# Decide which device we want to run on
device = torch.device("cuda:0" if (torch.cuda.is_available() and ngpu
> 0) else "cpu")

# Plot some training images
real_batch = next(iter(dataloader))
plt.figure(figsize=(8,8))
plt.axis("off")
plt.title("Training Images")
plt.imshow(np.transpose(vutils.make_grid(real_batch[0].to(device)
[:64], padding=2, normalize=True).cpu(),(1,2,0)))
plt.show()

# custom weights initialization called on ``netG`` and ``netD``
def weights_init(m):
    classname = m.__class__.__name__
    if classname.find('Conv') != -1:
        nn.init.normal_(m.weight.data, 0.0, 0.02)
    elif classname.find('BatchNorm') != -1:
        nn.init.normal_(m.weight.data, 1.0, 0.02)
        nn.init.constant_(m.bias.data, 0)

# Code for the generator
class Generator(nn.Module):
    def __init__(self, ngpu):
        super(Generator, self).__init__()
        self.ngpu = ngpu
        self.main = nn.Sequential(
            # input is Z, going into a convolution
            nn.ConvTranspose2d( nz, ngf * 8, 4, 1, 0, bias=False),
            nn.BatchNorm2d(ngf * 8),
            nn.ReLU(True),
            # state size. ``(ngf*8) x 4 x 4``
            nn.ConvTranspose2d(ngf * 8, ngf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 4),
            nn.ReLU(True),
            # state size. ``(ngf*4) x 8 x 8``
            nn.ConvTranspose2d( ngf * 4, ngf * 2, 4, 2, 1,
bias=False),
            nn.BatchNorm2d(ngf * 2),
            nn.ReLU(True),
            # state size. ``(ngf*2) x 16 x 16``
            nn.ConvTranspose2d( ngf * 2, ngf, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf),
            nn.ReLU(True),
            # state size. ``(ngf) x 32 x 32``
            nn.ConvTranspose2d( ngf, nc, 4, 2, 1, bias=False),
            nn.Tanh()
            # state size. ``(nc) x 64 x 64``
```

```
        )

    def forward(self, input):
        return self.main(input)


# Create the generator
netG = Generator(ngpu).to(device)

# Handle multi-GPU if desired
if (device.type == 'cuda') and (ngpu > 1):
    netG = nn.DataParallel(netG, list(range(ngpu)))

# Apply the ``weights_init`` function to randomly initialize all
weights
#  to ``mean=0``, ``stdev=0.02``.
netG.apply(weights_init)

# Print the model
print(netG)

# Discriminator code
class Discriminator(nn.Module):
    def __init__(self, ngpu):
        super(Discriminator, self).__init__()
        self.ngpu = ngpu
        self.main = nn.Sequential(
            # input is ``(nc) x 64 x 64``
            nn.Conv2d(nc, ndf, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. ``(ndf) x 32 x 32``
            nn.Conv2d(ndf, ndf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 2),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. ``(ndf*2) x 16 x 16``
            nn.Conv2d(ndf * 2, ndf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 4),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. ``(ndf*4) x 8 x 8``
            nn.Conv2d(ndf * 4, ndf * 8, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 8),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. ``(ndf*8) x 4 x 4``
            nn.Conv2d(ndf * 8, 1, 4, 1, 0, bias=False),
            nn.Sigmoid()
        )

    def forward(self, input):
        return self.main(input)
```

```python
# Create the Discriminator
netD = Discriminator(ngpu).to(device)

# Handle multi-GPU if desired
if (device.type == 'cuda') and (ngpu > 1):
    netD = nn.DataParallel(netD, list(range(ngpu)))

# Apply the ``weights_init`` function to randomly initialize all
weights
# like this: ``to mean=0, stdev=0.2``.
netD.apply(weights_init)

# Print the model
print(netD)

# Initialize the ``BCELoss`` function
criterion = nn.BCELoss()

# Create batch of latent vectors that we will use to visualize
#  the progression of the generator
fixed_noise = torch.randn(64, nz, 1, 1, device=device)

# Establish convention for real and fake labels during training
real_label = 1.
fake_label = 0.

# Setup Adam optimizers for both G and D
optimizerD = optim.Adam(netD.parameters(), lr=lr, betas=(beta1,
0.999))
optimizerG = optim.Adam(netG.parameters(), lr=lr, betas=(beta1,
0.999))

# Training Loop

# Lists to keep track of progress
img_list = []
G_losses = []
D_losses = []
iters = 0

print("Starting Training Loop...")
# For each epoch
for epoch in range(num_epochs):
    # For each batch in the dataloader
    for i, data in enumerate(dataloader, 0):

        ############################
        # (1) Update D network: maximize log(D(x)) + log(1 - D(G(z)))
        ###########################
        ## Train with all-real batch
```

```
        netD.zero_grad()
        # Format batch
        real_cpu = data[0].to(device)
        b_size = real_cpu.size(0)
        label = torch.full((b_size,), real_label, dtype=torch.float,
device=device)
        # Forward pass real batch through D
        output = netD(real_cpu).view(-1)
        # Calculate loss on all-real batch
        errD_real = criterion(output, label)
        # Calculate gradients for D in backward pass
        errD_real.backward()
        D_x = output.mean().item()

        ## Train with all-fake batch
        # Generate batch of latent vectors
        noise = torch.randn(b_size, nz, 1, 1, device=device)
        # Generate fake image batch with G
        fake = netG(noise)
        label.fill_(fake_label)
        # Classify all fake batch with D
        output = netD(fake.detach()).view(-1)
        # Calculate D's loss on the all-fake batch
        errD_fake = criterion(output, label)
        # Calculate the gradients for this batch, accumulated (summed)
with previous gradients
        errD_fake.backward()
        D_G_z1 = output.mean().item()
        # Compute error of D as sum over the fake and the real batches
        errD = errD_real + errD_fake
        # Update D
        optimizerD.step()

        ############################
        # (2) Update G network: maximize log(D(G(z)))
        ###########################
        netG.zero_grad()
        label.fill_(real_label)  # fake labels are real for generator
cost
        # Since we just updated D, perform another forward pass of
all-fake batch through D
        output = netD(fake).view(-1)
        # Calculate G's loss based on this output
        errG = criterion(output, label)
        # Calculate gradients for G
        errG.backward()
        D_G_z2 = output.mean().item()
        # Update G
        optimizerG.step()
```

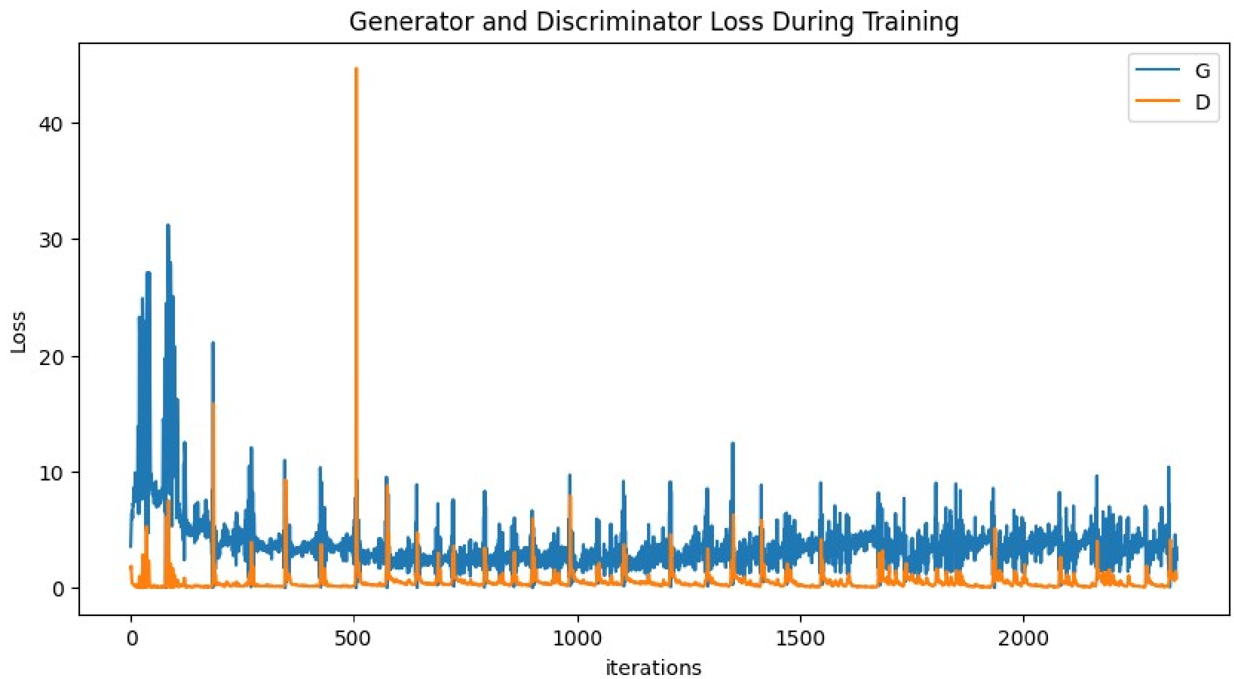
```
        # Output training stats
        if i % 50 == 0:
            print('[%d/%d][%d/%d]\tLoss_D: %.4f\tLoss_G: %.4f\tD(x): 
%.4f\tD(G(z)): %.4f / %.4f'
                  % (epoch, num_epochs, i, len(dataloader),
                     errD.item(), errG.item(), D_x, D_G_z1, D_G_z2))

        # Save Losses for plotting later
        G_losses.append(errG.item())
        D_losses.append(errD.item())

        # Check how the generator is doing by saving G's output on 
fixed_noise
        if (iters % 500 == 0) or ((epoch == num_epochs-1) and (i == 
len(dataloader)-1)):
            with torch.no_grad():
                fake = netG(fixed_noise).detach().cpu()
            img_list.append(vutils.make_grid(fake, padding=2, 
normalize=True))

        iters += 1

plt.figure(figsize=(10,5))
plt.title("Generator and Discriminator Loss During Training")
plt.plot(G_losses,label="G")
plt.plot(D_losses,label="D")
plt.xlabel("iterations")
plt.ylabel("Loss")
plt.legend()
plt.show()

fig = plt.figure(figsize=(8,8))
plt.axis("off")
ims = [[plt.imshow(np.transpose(i,(1,2,0)), animated=True)] for i in 
img_list]
ani = animation.ArtistAnimation(fig, ims, interval=1000, 
repeat_delay=1000, blit=True)

HTML(ani.to_jshtml())

Random Seed:  999

100%|██████████| 9.91M/9.91M [00:01<00:00, 5.00MB/s]
100%|██████████| 28.9k/28.9k [00:00<00:00, 133kB/s]
100%|██████████| 1.65M/1.65M [00:06<00:00, 245kB/s]
100%|██████████| 4.54k/4.54k [00:00<00:00, 5.92MB/s]
```

```
Generator(
  (main): Sequential(
    (0): ConvTranspose2d(100, 512, kernel_size=(4, 4), stride=(1, 1),
bias=False)
    (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2),
padding=(1, 1), bias=False)
    (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
```

```
    (5): ReLU(inplace=True)
    (6): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), 
padding=(1, 1), bias=False)
    (7): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, 
track_running_stats=True)
    (8): ReLU(inplace=True)
    (9): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), 
padding=(1, 1), bias=False)
    (10): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, 
track_running_stats=True)
    (11): ReLU(inplace=True)
    (12): ConvTranspose2d(64, 1, kernel_size=(4, 4), stride=(2, 2), 
padding=(1, 1), bias=False)
    (13): Tanh()
  )
)
Discriminator(
  (main): Sequential(
    (0): Conv2d(1, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 
1), bias=False)
    (1): LeakyReLU(negative_slope=0.2, inplace=True)
    (2): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), 
padding=(1, 1), bias=False)
    (3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, 
track_running_stats=True)
    (4): LeakyReLU(negative_slope=0.2, inplace=True)
    (5): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), 
padding=(1, 1), bias=False)
    (6): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, 
track_running_stats=True)
    (7): LeakyReLU(negative_slope=0.2, inplace=True)
    (8): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), 
padding=(1, 1), bias=False)
    (9): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, 
track_running_stats=True)
    (10): LeakyReLU(negative_slope=0.2, inplace=True)
    (11): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1), 
bias=False)
    (12): Sigmoid()
  )
)
Starting Training Loop...
[0/5][0/469]    Loss_D: 1.6647  Loss_G: 3.5638  D(x): 0.5276 
     D(G(z)): 0.5480 / 0.0425
[0/5][50/469]   Loss_D: 0.1198  Loss_G: 8.0849  D(x): 0.9621 
     D(G(z)): 0.0504 / 0.0005
[0/5][100/469]  Loss_D: 0.4658  Loss_G: 11.2935 D(x): 0.8092 
     D(G(z)): 0.0000 / 0.0000
[0/5][150/469]  Loss_D: 0.1800  Loss_G: 7.3761  D(x): 0.9744
```

```
      D(G(z)): 0.1290 / 0.0009
[0/5][200/469]  Loss_D: 0.3596  Loss_G: 2.8582  D(x): 0.8497
      D(G(z)): 0.1099 / 0.0699
[0/5][250/469]  Loss_D: 0.1643  Loss_G: 3.9710  D(x): 0.9223
      D(G(z)): 0.0608 / 0.0295
[0/5][300/469]  Loss_D: 0.1796  Loss_G: 3.3683  D(x): 0.9181
      D(G(z)): 0.0805 / 0.0560
[0/5][350/469]  Loss_D: 1.6565  Loss_G: 1.6952  D(x): 0.3608
      D(G(z)): 0.0197 / 0.2764
[0/5][400/469]  Loss_D: 0.1525  Loss_G: 3.3171  D(x): 0.9277
      D(G(z)): 0.0650 / 0.0519
[0/5][450/469]  Loss_D: 0.1834  Loss_G: 3.0966  D(x): 0.8968
      D(G(z)): 0.0634 / 0.0661
[1/5][0/469]    Loss_D: 0.1038  Loss_G: 3.8179  D(x): 0.9617
      D(G(z)): 0.0597 / 0.0338
[1/5][50/469]   Loss_D: 0.3714  Loss_G: 3.0747  D(x): 0.8327
      D(G(z)): 0.1453 / 0.0706
[1/5][100/469]  Loss_D: 0.2457  Loss_G: 2.3543  D(x): 0.8350
      D(G(z)): 0.0537 / 0.1342
[1/5][150/469]  Loss_D: 0.3520  Loss_G: 1.9007  D(x): 0.8245
      D(G(z)): 0.1274 / 0.1894
[1/5][200/469]  Loss_D: 0.3246  Loss_G: 2.7715  D(x): 0.9069
      D(G(z)): 0.1884 / 0.0786
[1/5][250/469]  Loss_D: 0.3692  Loss_G: 4.4221  D(x): 0.9673
      D(G(z)): 0.2690 / 0.0164
[1/5][300/469]  Loss_D: 0.2945  Loss_G: 2.2373  D(x): 0.8520
      D(G(z)): 0.1069 / 0.1350
[1/5][350/469]  Loss_D: 0.2796  Loss_G: 2.2481  D(x): 0.8587
      D(G(z)): 0.1033 / 0.1358
[1/5][400/469]  Loss_D: 0.3833  Loss_G: 2.5698  D(x): 0.7611
      D(G(z)): 0.0721 / 0.1126
[1/5][450/469]  Loss_D: 0.4571  Loss_G: 1.8122  D(x): 0.7687
      D(G(z)): 0.1403 / 0.2060
[2/5][0/469]    Loss_D: 0.2996  Loss_G: 2.7996  D(x): 0.8934
      D(G(z)): 0.1529 / 0.0853
[2/5][50/469]   Loss_D: 1.4062  Loss_G: 3.8033  D(x): 0.8706
      D(G(z)): 0.6175 / 0.0413
[2/5][100/469]  Loss_D: 0.2914  Loss_G: 3.2867  D(x): 0.9205
      D(G(z)): 0.1709 / 0.0513
[2/5][150/469]  Loss_D: 0.4913  Loss_G: 4.4762  D(x): 0.9783
      D(G(z)): 0.3419 / 0.0156
[2/5][200/469]  Loss_D: 0.3493  Loss_G: 2.1577  D(x): 0.8384
      D(G(z)): 0.1341 / 0.1554
[2/5][250/469]  Loss_D: 0.2031  Loss_G: 3.7878  D(x): 0.9422
      D(G(z)): 0.1174 / 0.0346
[2/5][300/469]  Loss_D: 0.2432  Loss_G: 2.5078  D(x): 0.8562
      D(G(z)): 0.0741 / 0.1049
[2/5][350/469]  Loss_D: 0.5573  Loss_G: 6.1547  D(x): 0.9923
      D(G(z)): 0.3870 / 0.0036
```

```
[2/5][400/469]  Loss_D: 0.1307  Loss_G: 3.7735  D(x): 0.9432
      D(G(z)): 0.0656 / 0.0355
[2/5][450/469]  Loss_D: 0.5888  Loss_G: 4.7625  D(x): 0.9700
      D(G(z)): 0.3873 / 0.0133
[3/5][0/469]    Loss_D: 0.1424  Loss_G: 3.2016  D(x): 0.9122
      D(G(z)): 0.0427 / 0.0546
[3/5][50/469]   Loss_D: 0.1532  Loss_G: 3.2051  D(x): 0.9412
      D(G(z)): 0.0841 / 0.0536
[3/5][100/469]  Loss_D: 0.2385  Loss_G: 4.1739  D(x): 0.9724
      D(G(z)): 0.1777 / 0.0215
[3/5][150/469]  Loss_D: 0.5808  Loss_G: 2.3352  D(x): 0.6584
      D(G(z)): 0.0908 / 0.1354
[3/5][200/469]  Loss_D: 0.7945  Loss_G: 6.0303  D(x): 0.9944
      D(G(z)): 0.4873 / 0.0038
[3/5][250/469]  Loss_D: 0.0685  Loss_G: 4.8411  D(x): 0.9775
      D(G(z)): 0.0434 / 0.0120
[3/5][300/469]  Loss_D: 0.4080  Loss_G: 2.2372  D(x): 0.7866
      D(G(z)): 0.1124 / 0.1404
[3/5][350/469]  Loss_D: 0.6825  Loss_G: 1.9602  D(x): 0.6148
      D(G(z)): 0.0897 / 0.1899
[3/5][400/469]  Loss_D: 1.2024  Loss_G: 5.6463  D(x): 0.9944
      D(G(z)): 0.6256 / 0.0059
[3/5][450/469]  Loss_D: 0.6225  Loss_G: 5.6355  D(x): 0.9391
      D(G(z)): 0.3755 / 0.0057
[4/5][0/469]    Loss_D: 0.1885  Loss_G: 2.9810  D(x): 0.9282
      D(G(z)): 0.0965 / 0.0737
[4/5][50/469]   Loss_D: 0.0848  Loss_G: 4.2611  D(x): 0.9693
      D(G(z)): 0.0505 / 0.0199
[4/5][100/469]  Loss_D: 0.1843  Loss_G: 3.3370  D(x): 0.9172
      D(G(z)): 0.0850 / 0.0494
[4/5][150/469]  Loss_D: 0.2098  Loss_G: 3.9255  D(x): 0.9335
      D(G(z)): 0.1174 / 0.0288
[4/5][200/469]  Loss_D: 0.0774  Loss_G: 3.9243  D(x): 0.9585
      D(G(z)): 0.0323 / 0.0279
[4/5][250/469]  Loss_D: 0.1694  Loss_G: 3.1837  D(x): 0.8885
      D(G(z)): 0.0428 / 0.0600
[4/5][300/469]  Loss_D: 0.7378  Loss_G: 3.2528  D(x): 0.9116
      D(G(z)): 0.4144 / 0.0575
[4/5][350/469]  Loss_D: 0.1903  Loss_G: 3.6803  D(x): 0.9261
      D(G(z)): 0.0994 / 0.0357
[4/5][400/469]  Loss_D: 1.1748  Loss_G: 6.8060  D(x): 0.9944
      D(G(z)): 0.6158 / 0.0019
[4/5][450/469]  Loss_D: 0.3920  Loss_G: 10.4228 D(x): 0.9928
      D(G(z)): 0.2928 / 0.0001
```




```
&lt;IPython.core.display.HTML object&gt;
```

The results for some images might be convincing, but there are some bad results as well. We will see how we can use different architectures and training objectives to get better results. More importantly, we generate images directly from noise, not knowing what number (if any) will come out on the output.

To control which number the generator should output :

We could concatenate an embedding from the label (the number that we want to generate) to the random latent state vector that is passed to the generator. Then, we have to pass also an embedding from the label to the discriminator. We can add the label information as a new channel to the image passed to the generator. Finally, we should modify the loss functions to take into account the labels.

Then, when we want to create a new image, we should pass the number that we want to generate as the label, which will be embedded and passed to the generator.

```
import argparse
import os
import random
import torch
import torch.nn as nn
import torch.nn.parallel
import torch.optim as optim
import torch.utils.data
import torchvision.datasets as dset
import torchvision.transforms as transforms
import torchvision.utils as vutils
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from IPython.display import HTML

# Set random seed for reproducibility
manualSeed = 999
#manualSeed = random.randint(1, 10000) # use if you want new results
print("Random Seed: ", manualSeed)
random.seed(manualSeed)
torch.manual_seed(manualSeed)

# Root directory for dataset
dataroot = "data/mnist"

# Number of workers for dataloader
workers = 2

# Batch size during training
batch_size = 128

# Spatial size of training images. All images will be resized to this
#   size using a transformer.
image_size = 64

# Number of channels in the training images. For color images this is
3
nc = 1

# Size of z latent vector (i.e. size of generator input)
nz = 100

# Size of feature maps in generator
ngf = 64

# Size of feature maps in discriminator
```

```
ndf = 64

# Number of training epochs
num_epochs = 5

# Learning rate for optimizers
lr = 0.0002

# Beta1 hyperparameter for Adam optimizers
beta1 = 0.5

# Number of GPUs available. Use 0 for CPU mode.
ngpu = 1

# Number of classes in the mnist dataset
n_class = 10

dataset = dset.MNIST(root=dataroot, download=True,
transform=transforms.Compose([
    transforms.Resize(image_size),
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
]))

dataloader = torch.utils.data.DataLoader(dataset,
batch_size=batch_size,
                                         shuffle=True,
num_workers=workers)

# Decide which device we want to run on
device = torch.device("cuda:0" if (torch.cuda.is_available() and ngpu
> 0) else "cpu")

# custom weights initialization called on ``netG`` and ``netD``
def weights_init(m):
    classname = m.__class__.__name__
    if classname.find('Conv') != -1:
        nn.init.normal_(m.weight.data, 0.0, 0.02)
    elif classname.find('BatchNorm') != -1:
        nn.init.normal_(m.weight.data, 1.0, 0.02)
        nn.init.constant_(m.bias.data, 0)

# Code for the generator
class Generator(nn.Module):
    def __init__(self, ngpu):
        super(Generator, self).__init__()
        self.ngpu = ngpu

        # Create the embedding layer
        self.label_emb = nn.Embedding(n_class, nz)
```

```
        self.main = nn.Sequential(
            # input is Z, going into a convolution
            nn.ConvTranspose2d( nz*2, ngf * 8, 4, 1, 0, bias=False),
            nn.BatchNorm2d(ngf * 8),
            nn.ReLU(True),
            # state size. ``(ngf*8) x 4 x 4``
            nn.ConvTranspose2d(ngf * 8, ngf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 4),
            nn.ReLU(True),
            # state size. ``(ngf*4) x 8 x 8``
            nn.ConvTranspose2d( ngf * 4, ngf * 2, 4, 2, 1,
bias=False),
            nn.BatchNorm2d(ngf * 2),
            nn.ReLU(True),
            # state size. ``(ngf*2) x 16 x 16``
            nn.ConvTranspose2d( ngf * 2, ngf, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf),
            nn.ReLU(True),
            # state size. ``(ngf) x 32 x 32``
            nn.ConvTranspose2d( ngf, nc, 4, 2, 1, bias=False),
            nn.Tanh()
            # state size. ``(nc) x 64 x 64``
        )

    def forward(self, input, labels):

        label_embedding = self.label_emb(labels)  # Shape:
(batch_size, nz)
        x = torch.cat([input.squeeze(-1).squeeze(-1),
label_embedding], dim=1)  # Concatenate noise (input) + label
embedding to add the label information
        x = x.unsqueeze(2).unsqueeze(3)  # Reshape to (batch_size,
nz*2, 1, 1)

        # Now the generator can access to the label information.
        return self.main(x)


# Create the generator
netG = Generator(ngpu).to(device)

# Handle multi-GPU if desired
if (device.type == 'cuda') and (ngpu > 1):
    netG = nn.DataParallel(netG, list(range(ngpu)))

# Apply the ``weights_init`` function to randomly initialize all
weights
#  to ``mean=0``, ``stdev=0.02``.
netG.apply(weights_init)
```

```
# Print the model
#print(netG)

# Discriminator code
class Discriminator(nn.Module):
    def __init__(self, ngpu):
        super(Discriminator, self).__init__()

        # Create the embedding layer
        self.label_emb = nn.Embedding(n_class, image_size*image_size)

        self.ngpu = ngpu
        self.main = nn.Sequential(
            # input is ``(nc) x 64 x 64``
            nn.Conv2d(nc+1, ndf, 4, 2, 1, bias=False), # Add the label
layer
            nn.LeakyReLU(0.2, inplace=True),
            # state size. ``(ndf) x 32 x 32``
            nn.Conv2d(ndf, ndf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 2),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. ``(ndf*2) x 16 x 16``
            nn.Conv2d(ndf * 2, ndf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 4),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. ``(ndf*4) x 8 x 8``
            nn.Conv2d(ndf * 4, ndf * 8, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 8),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. ``(ndf*8) x 4 x 4``
            nn.Conv2d(ndf * 8, 1, 4, 1, 0, bias=False),
            nn.Sigmoid()
        )

    def forward(self, input, labels):
        label_embedding = self.label_emb(labels).view(-1, 1,
image_size, image_size)  # Reshape to match image size
        x = torch.cat([input, label_embedding], dim=1)  # Concatenate
as additional input channel to add the label information
        # Now the discriminator has the label information
        #print("Generator input shape") [128,2,64,64]
        #print(x.shape)
        return self.main(x)

# Create the Discriminator
netD = Discriminator(ngpu).to(device)

# Handle multi-GPU if desired
if (device.type == 'cuda') and (ngpu > 1):
```

```
    netD = nn.DataParallel(netD, list(range(ngpu)))

# Apply the ``weights_init`` function to randomly initialize all
weights
# like this: ``to mean=0, stdev=0.2``.
netD.apply(weights_init)

# Print the model
#print(netD)

# Initialize the ``BCELoss`` function
criterion = nn.BCELoss()

# Create batch of latent vectors that we will use to visualize
#  the progression of the generator
fixed_noise = torch.randn(64, nz, 1, 1, device=device)

# Establish convention for real and fake labels during training
real_label = 1.
fake_label = 0.

# Setup Adam optimizers for both G and D
optimizerD = optim.Adam(netD.parameters(), lr=lr, betas=(beta1,
0.999))
optimizerG = optim.Adam(netG.parameters(), lr=lr, betas=(beta1,
0.999))

# Training Loop

# Lists to keep track of progress
img_list = []
G_losses = []
D_losses = []
iters = 0

print("Starting Training Loop...")
# For each epoch
for epoch in range(num_epochs):
    # For each batch in the dataloader
    for i, (image, label) in enumerate(dataloader, 0):

        ############################
        # (1) Update D network: maximize log(D(x)) + log(1 - D(G(z)))
        ###########################
        ## Train with all-real batch
        netD.zero_grad()
        # Format batch
        real_cpu = image.to(device)
        label_cpu = label.to(device)
```

```
        b_size = real_cpu.size(0)
        # print(f"Batch Size : {b_size}") = [128]
        label = torch.full((b_size,), real_label, dtype=torch.float,
device=device)
        # Forward pass real batch through D
        output = netD(real_cpu, label_cpu).view(-1)
        # Calculate loss on all-real batch
        errD_real = criterion(output, label)
        # Calculate gradients for D in backward pass
        errD_real.backward()
        D_x = output.mean().item()

        ## Train with all-fake batch
        # Generate batch of latent vectors
        noise = torch.randn(b_size, nz, 1, 1, device=device)
        # Generate fake image batch with G
        fake = netG(noise, label_cpu)
        label.fill_(fake_label)
        # Classify all fake batch with D
        output = netD(fake.detach(), label_cpu).view(-1)
        # Calculate D's loss on the all-fake batch
        errD_fake = criterion(output, label)
        # Calculate the gradients for this batch, accumulated (summed)
with previous gradients
        errD_fake.backward()
        D_G_z1 = output.mean().item()
        # Compute error of D as sum over the fake and the real batches
        errD = errD_real + errD_fake
        # Update D
        optimizerD.step()

        ############################
        # (2) Update G network: maximize log(D(G(z)))
        ###########################
        netG.zero_grad()
        label.fill_(real_label)  # fake labels are real for generator
cost
        # Since we just updated D, perform another forward pass of
all-fake batch through D
        output = netD(fake, label_cpu).view(-1)
        # Calculate G's loss based on this output
        errG = criterion(output, label)
        # Calculate gradients for G
        errG.backward()
        D_G_z2 = output.mean().item()
        # Update G
        optimizerG.step()

        # Output training stats
        if i % 50 == 0:
```

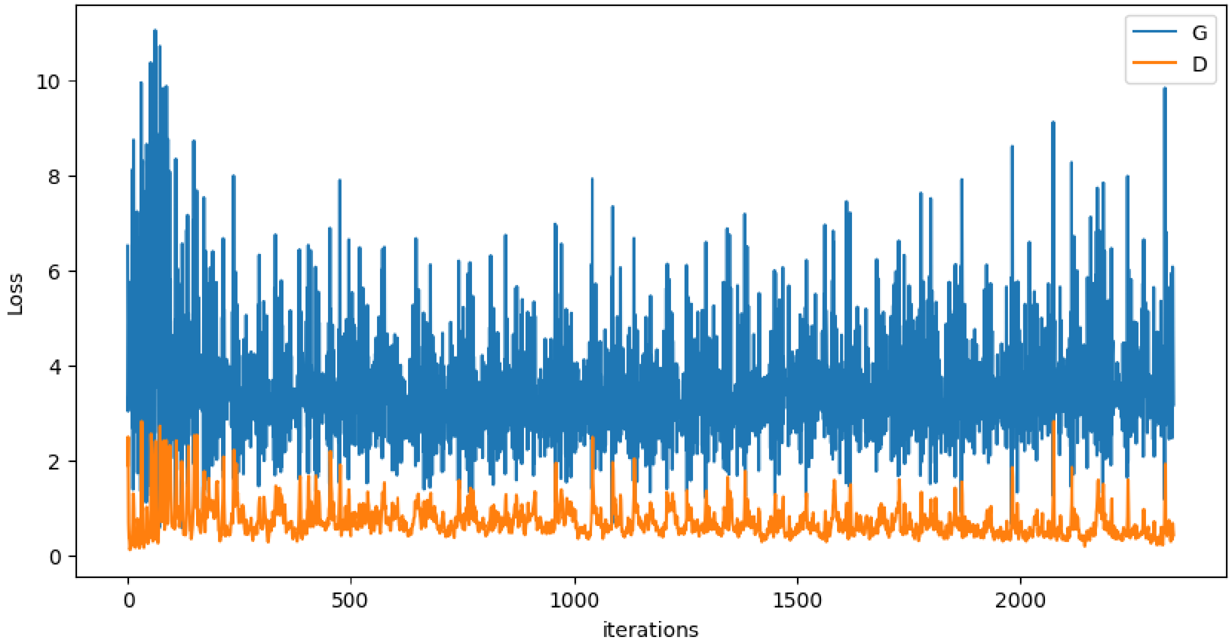
```
            print('[%d/%d][%d/%d]\tLoss_D: %.4f\tLoss_G: %.4f\tD(x):
%.4f\tD(G(z)): %.4f / %.4f'
                  % (epoch, num_epochs, i, len(dataloader),
                     errD.item(), errG.item(), D_x, D_G_z1, D_G_z2))

        # Save Losses for plotting later
        G_losses.append(errG.item())
        D_losses.append(errD.item())

        # Check how the generator is doing by saving G's output on
fixed_noise
        #if (iters % 500 == 0) or ((epoch == num_epochs-1) and (i ==
len(dataloader)-1)):
        #    with torch.no_grad():
        #        fake = netG(fixed_noise, label_cpu).detach().cpu()
        #    img_list.append(vutils.make_grid(fake, padding=2,
normalize=True))

        iters += 1

plt.figure(figsize=(10,5))
plt.title("Generator and Discriminator Loss During Training")
plt.plot(G_losses,label="G")
plt.plot(D_losses,label="D")
plt.xlabel("iterations")
plt.ylabel("Loss")
plt.legend()
plt.show()



def generate_digit(digit, nz=100):
    noise = torch.randn(1, nz, device=device)  # Generate random noise
    label = torch.tensor([digit], device=device)  # Specify label
    fake_image = netG(noise,
label).detach().cpu().squeeze(0).squeeze(0)  # Remove batch/channel
dims

    plt.imshow(fake_image, cmap="gray")
    plt.title(f"Generated {digit}")
    plt.axis("off")
    plt.show()

# Generate a fake "3"
generate_digit(3)
```

```
Random Seed:  999
Starting Training Loop...
[0/5][0/469]    Loss_D: 1.8994  Loss_G: 6.5031  D(x): 0.6861
     D(G(z)): 0.7039 / 0.0460
```

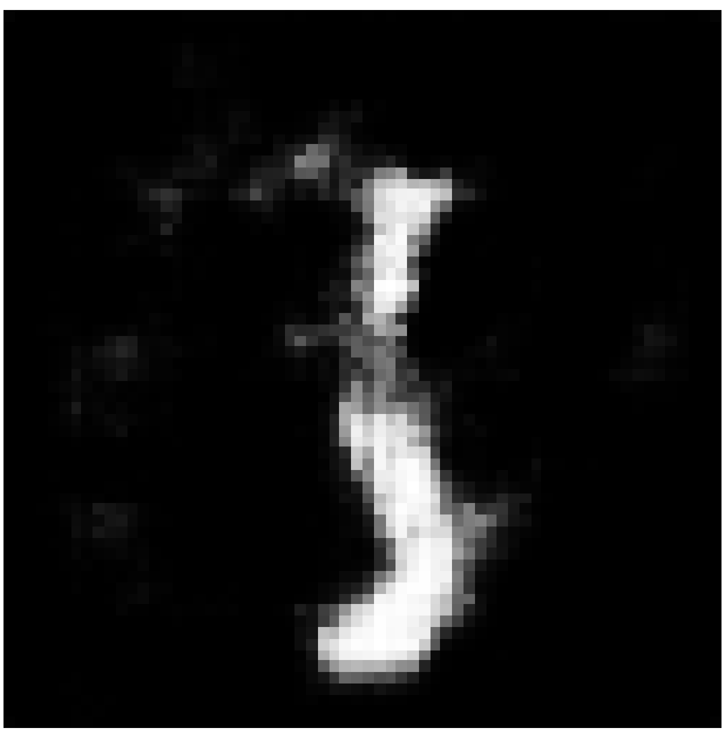
```
[0/5][50/469]	Loss_D: 0.5472	Loss_G: 1.4628	D(x): 0.6934
	D(G(z)): 0.0214 / 0.3309
[0/5][100/469]	Loss_D: 0.6805	Loss_G: 4.0875	D(x): 0.8445
	D(G(z)): 0.3351 / 0.0288
[0/5][150/469]	Loss_D: 1.3803	Loss_G: 6.3574	D(x): 0.9373
	D(G(z)): 0.6278 / 0.0046
[0/5][200/469]	Loss_D: 1.5152	Loss_G: 2.0181	D(x): 0.4625
	D(G(z)): 0.0832 / 0.1910
[0/5][250/469]	Loss_D: 0.9651	Loss_G: 4.2040	D(x): 0.8220
	D(G(z)): 0.4245 / 0.0205
[0/5][300/469]	Loss_D: 0.5713	Loss_G: 3.1283	D(x): 0.7568
	D(G(z)): 0.1579 / 0.0569
[0/5][350/469]	Loss_D: 0.7803	Loss_G: 4.2316	D(x): 0.8287
	D(G(z)): 0.3818 / 0.0229
[0/5][400/469]	Loss_D: 0.5452	Loss_G: 3.7431	D(x): 0.8313
	D(G(z)): 0.2370 / 0.0431
[0/5][450/469]	Loss_D: 0.6505	Loss_G: 2.4229	D(x): 0.7078
	D(G(z)): 0.0929 / 0.0985
[1/5][0/469]	Loss_D: 0.5669	Loss_G: 3.2375	D(x): 0.8169
	D(G(z)): 0.2506 / 0.0444
[1/5][50/469]	Loss_D: 0.7690	Loss_G: 2.1237	D(x): 0.6470
	D(G(z)): 0.0840 / 0.1496
[1/5][100/469]	Loss_D: 0.9915	Loss_G: 6.0058	D(x): 0.8431
	D(G(z)): 0.4899 / 0.0040
[1/5][150/469]	Loss_D: 0.4996	Loss_G: 3.2635	D(x): 0.8040
	D(G(z)): 0.1322 / 0.0449
[1/5][200/469]	Loss_D: 0.8887	Loss_G: 1.6377	D(x): 0.6027
	D(G(z)): 0.1068 / 0.2456
[1/5][250/469]	Loss_D: 0.4063	Loss_G: 3.1290	D(x): 0.7872
	D(G(z)): 0.0632 / 0.0497
[1/5][300/469]	Loss_D: 1.2019	Loss_G: 4.5476	D(x): 0.8573
	D(G(z)): 0.5003 / 0.0192
[1/5][350/469]	Loss_D: 0.9843	Loss_G: 1.4861	D(x): 0.5489
	D(G(z)): 0.0515 / 0.3588
[1/5][400/469]	Loss_D: 0.8113	Loss_G: 2.1335	D(x): 0.6378
	D(G(z)): 0.1324 / 0.1618
[1/5][450/469]	Loss_D: 0.5672	Loss_G: 2.8788	D(x): 0.7914
	D(G(z)): 0.1925 / 0.0589
[2/5][0/469]	Loss_D: 0.7008	Loss_G: 3.4542	D(x): 0.8377
	D(G(z)): 0.3234 / 0.0425
[2/5][50/469]	Loss_D: 0.9197	Loss_G: 2.4780	D(x): 0.6679
	D(G(z)): 0.2253 / 0.0951
[2/5][100/469]	Loss_D: 0.6190	Loss_G: 2.4651	D(x): 0.7244
	D(G(z)): 0.1679 / 0.1053
[2/5][150/469]	Loss_D: 1.7846	Loss_G: 3.1434	D(x): 0.3504
	D(G(z)): 0.0269 / 0.1948
[2/5][200/469]	Loss_D: 1.0755	Loss_G: 3.9352	D(x): 0.7819
	D(G(z)): 0.4528 / 0.0272
[2/5][250/469]	Loss_D: 0.5792	Loss_G: 3.1196	D(x): 0.8225
```

```
      D(G(z)): 0.2533 / 0.0509
[2/5][300/469]  Loss_D: 0.4712  Loss_G: 3.2003  D(x): 0.8030
      D(G(z)): 0.1725 / 0.0474
[2/5][350/469]  Loss_D: 0.6748  Loss_G: 5.1567  D(x): 0.8656
      D(G(z)): 0.3522 / 0.0076
[2/5][400/469]  Loss_D: 0.7321  Loss_G: 2.8162  D(x): 0.6607
      D(G(z)): 0.0375 / 0.0953
[2/5][450/469]  Loss_D: 1.1752  Loss_G: 6.4929  D(x): 0.9128
      D(G(z)): 0.5845 / 0.0028
[3/5][0/469]    Loss_D: 0.4821  Loss_G: 2.5098  D(x): 0.7897
      D(G(z)): 0.1355 / 0.0857
[3/5][50/469]   Loss_D: 0.5656  Loss_G: 2.4564  D(x): 0.7543
      D(G(z)): 0.1358 / 0.1044
[3/5][100/469]  Loss_D: 0.6346  Loss_G: 5.0624  D(x): 0.8674
      D(G(z)): 0.3419 / 0.0123
[3/5][150/469]  Loss_D: 0.3408  Loss_G: 3.2381  D(x): 0.8058
      D(G(z)): 0.0571 / 0.0505
[3/5][200/469]  Loss_D: 0.7516  Loss_G: 3.9021  D(x): 0.7815
      D(G(z)): 0.3079 / 0.0292
[3/5][250/469]  Loss_D: 0.6513  Loss_G: 2.8243  D(x): 0.7277
      D(G(z)): 0.1171 / 0.0837
[3/5][300/469]  Loss_D: 0.3888  Loss_G: 3.8703  D(x): 0.8678
      D(G(z)): 0.1636 / 0.0321
[3/5][350/469]  Loss_D: 0.6208  Loss_G: 2.7080  D(x): 0.6847
      D(G(z)): 0.0685 / 0.1102
[3/5][400/469]  Loss_D: 0.5391  Loss_G: 2.5718  D(x): 0.7932
      D(G(z)): 0.1626 / 0.0813
[3/5][450/469]  Loss_D: 0.6630  Loss_G: 3.5994  D(x): 0.8507
      D(G(z)): 0.2822 / 0.0374
[4/5][0/469]    Loss_D: 0.8174  Loss_G: 2.9359  D(x): 0.6537
      D(G(z)): 0.0542 / 0.0969
[4/5][50/469]   Loss_D: 0.9527  Loss_G: 2.6059  D(x): 0.5736
      D(G(z)): 0.0314 / 0.1424
[4/5][100/469]  Loss_D: 0.4598  Loss_G: 3.6535  D(x): 0.8650
      D(G(z)): 0.2126 / 0.0318
[4/5][150/469]  Loss_D: 0.5362  Loss_G: 4.6841  D(x): 0.8894
      D(G(z)): 0.2810 / 0.0136
[4/5][200/469]  Loss_D: 0.9457  Loss_G: 4.0870  D(x): 0.9119
      D(G(z)): 0.4165 / 0.0336
[4/5][250/469]  Loss_D: 0.8617  Loss_G: 5.1427  D(x): 0.8243
      D(G(z)): 0.3566 / 0.0116
[4/5][300/469]  Loss_D: 1.2585  Loss_G: 2.4399  D(x): 0.4957
      D(G(z)): 0.0596 / 0.1912
[4/5][350/469]  Loss_D: 0.3313  Loss_G: 3.6037  D(x): 0.8716
      D(G(z)): 0.1379 / 0.0289
[4/5][400/469]  Loss_D: 0.8640  Loss_G: 2.0812  D(x): 0.6261
      D(G(z)): 0.0282 / 0.2071
[4/5][450/469]  Loss_D: 0.4575  Loss_G: 2.7458  D(x): 0.8667
      D(G(z)): 0.1978 / 0.1528
```

Generator and Discriminator Loss During Training



Generated 3



Conditional GAN (cGAN)

Therefore, we do not only want to generate a picture from random noise, but rather generate a picture from another picture. For this purpose we will use a cGAN instead of a vanilla GAN, which was introduced in this paper.

A cGAN is a supervised GAN aiming at mapping a label picture to a real one or a real picture to a label one. As you can see in the diagram below, the discriminator will take as input a pair of images and try to predict if the pair was generated or not. The generator will not generate an image from noise but will intead use an image (label or real) to generate another one (real or label).

Generator

In the cGAN architecture, the generator chosen is a U-Net.

A U-Net takes as input an image, and outputs another image.

It can be divided into 2 subparts : an encoder and a decoder.

- The encoder takes the input image and reduces its dimension to encode the main features into a vector.
- The decoder takes this vector and map the features stored into an image.

A U-Net architecture is different from a vanilla convolutional encoder-decoder in that every layer of the decoder takes as input the previous decoded output as well as the output feature map from the encoder layers of the same level. This allows the decoder to map low frequencies information encoded during the descent as well as high frequencies from the original picture.

The architecture we will implement is the following:

The encoder will take as input a colored picture (3 channels: RGB), it will pass through a series of convolution layers to encode the features of the picture. It will then be decoded by the decoder using transposed convolutional layers. These layers will take as input the previous decoded vector AND the encoded features of the same level.

For this part the objective is to use a cGAN to generate facades from a template image. For this purpose, we will use the "Facade" dataset.

Let's start by creating a few classes describing the layers we will use in the U-Net.

```python
# Importing all the libraries needed
import os
import glob
import torch
import kagglehub
import argparse
import os
import random
import torch
import torch.nn as nn
```

```
import torch.nn.parallel
import torch.optim as optim
import torch.utils.data
import torchvision.datasets as dset
import torchvision.transforms as transforms
import torchvision.utils as vutils
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from IPython.display import HTML

import numpy as np
import matplotlib.pyplot as plt
import torchvision.transforms as transforms

from torch import nn
from PIL import Image
from torch.utils.data import Dataset, DataLoader

# Set random seed for reproducibility
manualSeed = 999
#manualSeed = random.randint(1, 10000) # use if you want new results
print("Random Seed: ", manualSeed)
random.seed(manualSeed)
torch.manual_seed(manualSeed)
torch.use_deterministic_algorithms(True) # Needed for reproducible
results
```

```
Random Seed:  999
```

```
# code adapted from
https://github.com/milesial/Pytorch-UNet/blob/master/unet/unet_parts.p
y

# Input layer
class inconv(nn.Module):
    def __init__(self, in_ch, out_ch):
        super(inconv, self).__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(in_ch, out_ch, kernel_size=4, padding=1,
stride=2),
            nn.LeakyReLU(negative_slope=0.2, inplace=True)
        )

    def forward(self, x):
        x = self.conv(x)
        return x

# Encoder layer
class down(nn.Module):
```

```
    def __init__(self, in_ch, out_ch):
        super(down, self).__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(in_ch, out_ch, kernel_size=4, padding=1,
stride=2),
            nn.BatchNorm2d(out_ch),
            nn.LeakyReLU(negative_slope=0.2, inplace=True)
        )

    def forward(self, x):
        x = self.conv(x)
        return x

# Decoder layer
class up(nn.Module):
    def __init__(self, in_ch, out_ch, dropout=False):
        super(up, self).__init__()
        if dropout :
            self.conv = nn.Sequential(
                nn.ConvTranspose2d(in_ch, out_ch, kernel_size=4,
padding=1, stride=2),
                nn.BatchNorm2d(out_ch),
                nn.Dropout(0.5, inplace=True),
                nn.ReLU(inplace=True)
            )
        else:
            self.conv = nn.Sequential(
                nn.ConvTranspose2d(in_ch, out_ch, kernel_size=4,
padding=1, stride=2),
                nn.BatchNorm2d(out_ch),
                nn.ReLU(inplace=True)
            )

    def forward(self, x1, x2):
        x1 = self.conv(x1)
        x = torch.cat([x1, x2], dim=1)
        return x

# Output layer
class outconv(nn.Module):
    def __init__(self, in_ch, out_ch):
        super(outconv, self).__init__()
        self.conv = nn.Sequential(
              nn.ConvTranspose2d(in_ch, out_ch, kernel_size=4,
padding=1, stride=2),
              nn.Tanh()
        )

    def forward(self, x):
```

```
        x = self.conv(x)
        return x
```

Now let's create the U-Net using the helper classes defined previously.

```
class U_Net(nn.Module):
    '''
    Ck denotes a Convolution-BatchNorm-ReLU layer with k filters.
    CDk denotes a Convolution-BatchNorm-Dropout-ReLU layer with a
dropout rate of 50%
    Encoder:
      C64   - C128   - C256   - C512  - C512  - C512 - C512 - C512
    Decoder:
      CD512 - CD1024 - CD1024 - C1024 - C1024 - C512 - C256 - C128
    '''
    def __init__(self, n_channels, n_classes):
        super(U_Net, self).__init__()
        # Encoder
        self.inc = inconv(n_channels, 64) # 64 filters
        # Create the 7 encoder layers called "down1" to "down7"
following this sequence
        # C64   - C128   - C256   - C512  - C512  - C512 - C512 - C512
        # The first one has already been implemented
        self.down1 = down(64, 128)   # C64 - C128
        self.down2 = down(128, 256)  # C128 - C256
        self.down3 = down(256, 512)  # C256 - C512
        self.down4 = down(512, 512)  # C512 - C512
        self.down5 = down(512, 512)  # C512 - C512
        self.down6 = down(512, 512)  # C512 - C512
        self.down7 = down(512, 512)  # C512 - C512

        # Decoder
        # Create the 7 decoder layers called up1 to up7 following this
sequence :
        # CD512 - CD1024 - CD1024 - C1024 - C1024 - C512 - C256 - C128
        # The last layer has already been defined
        self.up7 = up(512, 512, dropout=True)   # CD512 - CD1024
        self.up6 = up(1024, 512, dropout=True)  # CD1024 - CD1024
        self.up5 = up(1024, 512, dropout=True) # CD1024 - C1024
        self.up4 = up(1024, 512) # C1024 - C1024
        self.up3 = up(1024, 256)  # C1024 - C512
        self.up2 = up(512, 128)   # C512 - C256
        self.up1 = up(256, 64)   # C256 - C128

        self.outc = outconv(128, n_classes) # 128 filters

    def forward(self, x):
        x1 = self.inc(x)
        x2 = self.down1(x1)
```

```
        x3 = self.down2(x2)
        x4 = self.down3(x3)
        x5 = self.down4(x4)
        x6 = self.down5(x5)
        x7 = self.down6(x6)
        x8 = self.down7(x7)
        # At this stage x8 is our encoded vector, we will now decode
it
        x = self.up7(x8, x7)
        x = self.up6(x, x6)
        x = self.up5(x, x5)
        x = self.up4(x, x4)
        x = self.up3(x, x3)
        x = self.up2(x, x2)
        x = self.up1(x, x1)
        x = self.outc(x)
        return x

# We take images that have 3 channels (RGB) as input and output an
image that also have 3 channels (RGB)
generator = U_Net(3, 3)

# Check that the architecture is as expected
print(generator)

U_Net(
  (inc): inconv(
    (conv): Sequential(
      (0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2),
padding=(1, 1))
      (1): LeakyReLU(negative_slope=0.2, inplace=True)
    )
  )
  (down1): down(
    (conv): Sequential(
      (0): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2),
padding=(1, 1))
      (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): LeakyReLU(negative_slope=0.2, inplace=True)
    )
  )
  (down2): down(
    (conv): Sequential(
      (0): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2),
padding=(1, 1))
      (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): LeakyReLU(negative_slope=0.2, inplace=True)
    )
```

```
  )
  (down3): down(
    (conv): Sequential(
      (0): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2),
padding=(1, 1))
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): LeakyReLU(negative_slope=0.2, inplace=True)
    )
  )
  (down4): down(
    (conv): Sequential(
      (0): Conv2d(512, 512, kernel_size=(4, 4), stride=(2, 2),
padding=(1, 1))
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): LeakyReLU(negative_slope=0.2, inplace=True)
    )
  )
  (down5): down(
    (conv): Sequential(
      (0): Conv2d(512, 512, kernel_size=(4, 4), stride=(2, 2),
padding=(1, 1))
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): LeakyReLU(negative_slope=0.2, inplace=True)
    )
  )
  (down6): down(
    (conv): Sequential(
      (0): Conv2d(512, 512, kernel_size=(4, 4), stride=(2, 2),
padding=(1, 1))
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): LeakyReLU(negative_slope=0.2, inplace=True)
    )
  )
  (down7): down(
    (conv): Sequential(
      (0): Conv2d(512, 512, kernel_size=(4, 4), stride=(2, 2),
padding=(1, 1))
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): LeakyReLU(negative_slope=0.2, inplace=True)
    )
  )
  (up7): up(
    (conv): Sequential(
      (0): ConvTranspose2d(512, 512, kernel_size=(4, 4), stride=(2,
```

```
2), padding=(1, 1))
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): Dropout(p=0.5, inplace=True)
      (3): ReLU(inplace=True)
    )
  )
  (up6): up(
    (conv): Sequential(
      (0): ConvTranspose2d(1024, 512, kernel_size=(4, 4), stride=(2,
2), padding=(1, 1))
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): Dropout(p=0.5, inplace=True)
      (3): ReLU(inplace=True)
    )
  )
  (up5): up(
    (conv): Sequential(
      (0): ConvTranspose2d(1024, 512, kernel_size=(4, 4), stride=(2,
2), padding=(1, 1))
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): Dropout(p=0.5, inplace=True)
      (3): ReLU(inplace=True)
    )
  )
  (up4): up(
    (conv): Sequential(
      (0): ConvTranspose2d(1024, 512, kernel_size=(4, 4), stride=(2,
2), padding=(1, 1))
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): ReLU(inplace=True)
    )
  )
  (up3): up(
    (conv): Sequential(
      (0): ConvTranspose2d(1024, 256, kernel_size=(4, 4), stride=(2,
2), padding=(1, 1))
      (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): ReLU(inplace=True)
    )
  )
  (up2): up(
    (conv): Sequential(
      (0): ConvTranspose2d(512, 128, kernel_size=(4, 4), stride=(2,
2), padding=(1, 1))
```

```
      (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): ReLU(inplace=True)
    )
  )
  (up1): up(
    (conv): Sequential(
      (0): ConvTranspose2d(256, 64, kernel_size=(4, 4), stride=(2, 2),
padding=(1, 1))
      (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): ReLU(inplace=True)
    )
  )
  (outc): outconv(
    (conv): Sequential(
      (0): ConvTranspose2d(128, 3, kernel_size=(4, 4), stride=(2, 2),
padding=(1, 1))
      (1): Tanh()
    )
  )
)
```

As said in this notebook, the skip connections allow the decoder to map low frequencies information encoded during the descent as well as high frequencies from the original picture.

The deeper layers encode global features, but local details are lost. The skip connections can add this details (edges and textures) to the decoder.

It helps also with the gradient propagation, helping avoiding the vanishing gradient problem.

```
# Check the results
sample_tensor = torch.ones(5, 3, 256,256)
generator.forward(sample_tensor)
```

```
torch.Size([5, 512, 1, 1])
torch.Size([5, 1024, 2, 2])
```

```
tensor([[[[-6.3039e-01, -9.3115e-01, -1.7475e-01,  ..., -9.6688e-01,
           -1.6537e-01,  1.7066e-01],
          [ 3.1666e-01, -3.7029e-01,  9.6352e-01,  ...,  9.5525e-01,
            8.2109e-01,  4.2626e-01],
          [-9.6451e-01,  1.6698e-01, -9.9098e-01,  ..., -9.7776e-01,
           -9.5653e-01, -7.5852e-01],
          ...,
          [ 6.3846e-01,  9.4282e-01,  9.9358e-01,  ...,  9.7948e-01,
            9.9242e-01,  9.9542e-01],
          [ 2.9804e-01,  9.9996e-01, -8.8132e-01,  ..., -5.0037e-01,
           -9.5039e-01, -7.7631e-01],
          [-2.0238e-02,  1.9199e-01,  8.4132e-01,  ..., -5.0987e-01,
```

```
         -8.1710e-01,  9.2138e-01]],

        [[ 8.6059e-01, -7.9560e-01,  7.4909e-01,  ..., -9.4234e-01,
           1.9305e-01, -5.7160e-01],
         [-8.1883e-01, -2.0916e-01,  9.9444e-01,  ...,  6.4839e-01,
           9.9881e-01,  9.0411e-01],
         [ 8.9041e-01,  9.7162e-01, -9.6767e-01,  ..., -7.2810e-01,
           9.9093e-01, -7.0360e-01],
         ...,
         [-6.0750e-01, -9.9982e-01,  9.9936e-01,  ..., -9.9864e-01,
          -3.3707e-01,  9.7019e-01],
         [-7.5609e-01,  9.7917e-01, -9.3315e-01,  ..., -9.9976e-01,
          -1.0842e-01, -6.8717e-01],
         [-5.2654e-01, -5.8557e-01, -9.9610e-01,  ...,  1.9784e-01,
          -9.4711e-01, -2.6716e-01]],

        [[-8.5208e-01, -3.2094e-01, -3.1630e-02,  ...,  9.4156e-01,
           2.8174e-04, -7.2623e-01],
         [-6.7592e-01, -9.8461e-01, -9.9183e-01,  ..., -3.6514e-01,
          -8.7971e-01,  3.4298e-01],
         [-3.2404e-01, -9.6126e-01,  7.5747e-01,  ...,  9.9881e-01,
           2.8776e-01, -5.6635e-01],
         ...,
         [ 8.6432e-01,  1.8917e-01, -8.7892e-01,  ..., -9.9872e-01,
          -9.0184e-01, -8.3236e-01],
         [-2.5117e-01,  9.7536e-01, -7.9081e-01,  ...,  7.7353e-01,
          -9.7932e-01, -8.6349e-01],
         [-5.8535e-01,  6.0113e-01,  7.3289e-02,  ..., -8.4776e-01,
          -2.0718e-01, -6.5619e-01]]],


       [[[-6.2762e-01, -9.3529e-01, -1.5538e-01,  ..., -9.6705e-01,
          -1.8091e-01,  1.7697e-01],
         [ 3.2176e-01, -3.4972e-01,  9.6652e-01,  ...,  9.5035e-01,
           7.9293e-01,  4.3302e-01],
         [-9.6548e-01,  2.1823e-01, -9.9119e-01,  ..., -9.8107e-01,
          -9.6557e-01, -7.5935e-01],
         ...,
         [ 6.3531e-01,  9.4257e-01,  9.9428e-01,  ...,  9.8116e-01,
           9.9327e-01,  9.9495e-01],
         [ 2.9461e-01,  9.9996e-01, -8.9083e-01,  ..., -4.6503e-01,
          -9.5530e-01, -7.8827e-01],
         [-2.3640e-02,  2.1902e-01,  8.4361e-01,  ..., -5.0964e-01,
          -8.1953e-01,  9.2192e-01]],

        [[ 8.6164e-01, -7.9604e-01,  7.5753e-01,  ..., -9.4236e-01,
           1.7648e-01, -5.7583e-01],
         [-8.2243e-01, -1.9804e-01,  9.9540e-01,  ...,  6.7815e-01,
           9.9866e-01,  9.0475e-01],
         [ 8.8580e-01,  9.6829e-01, -9.6634e-01,  ..., -7.4320e-01,
```

```
          9.8950e-01, -7.1177e-01],
        ...,
        [-6.0413e-01, -9.9979e-01,  9.9929e-01,  ..., -9.9884e-01,
         -3.3061e-01,  9.6894e-01],
        [-7.5074e-01,  9.7850e-01, -9.3545e-01,  ..., -9.9974e-01,
         -1.3951e-01, -6.9856e-01],
        [-5.3041e-01, -5.9276e-01, -9.9615e-01,  ...,  1.8253e-01,
         -9.4795e-01, -2.7359e-01]],

       [[-8.5271e-01, -3.3668e-01, -4.7341e-02,  ...,  9.4091e-01,
          5.2158e-02, -7.2730e-01],
        [-6.6969e-01, -9.8314e-01, -9.9118e-01,  ..., -4.3515e-01,
         -8.7685e-01,  3.2005e-01],
        [-3.3576e-01, -9.6353e-01,  7.1765e-01,  ...,  9.9899e-01,
          2.4308e-01, -5.6026e-01],
        ...,
        [ 8.7147e-01,  1.3262e-01, -8.7803e-01,  ..., -9.9852e-01,
         -8.9842e-01, -8.3097e-01],
        [-2.3438e-01,  9.7466e-01, -7.9054e-01,  ...,  7.5615e-01,
         -9.7938e-01, -8.5978e-01],
        [-5.8486e-01,  5.7055e-01,  8.0757e-02,  ..., -8.4947e-01,
         -2.0559e-01, -6.5106e-01]]],


      [[[-6.2758e-01, -9.3030e-01, -1.6314e-01,  ..., -9.6782e-01,
         -1.4759e-01,  1.7695e-01],
        [ 3.2906e-01, -4.3587e-01,  9.6381e-01,  ...,  9.5246e-01,
          7.9494e-01,  4.2115e-01],
        [-9.6401e-01,  2.1372e-01, -9.9028e-01,  ..., -9.7704e-01,
         -9.5384e-01, -7.4813e-01],
        ...,
        [ 6.1850e-01,  9.4633e-01,  9.9479e-01,  ...,  9.7971e-01,
          9.9180e-01,  9.9523e-01],
        [ 2.8161e-01,  9.9995e-01, -8.7885e-01,  ..., -5.4753e-01,
         -9.5081e-01, -7.8132e-01],
        [-2.8798e-02,  1.7922e-01,  8.3951e-01,  ..., -4.9248e-01,
         -8.2223e-01,  9.2220e-01]],

       [[ 8.6287e-01, -7.9943e-01,  7.5657e-01,  ..., -9.4225e-01,
          2.1140e-01, -5.7211e-01],
        [-8.2619e-01, -1.6795e-01,  9.9538e-01,  ...,  6.0566e-01,
          9.9878e-01,  9.0617e-01],
        [ 8.8057e-01,  9.7094e-01, -9.6185e-01,  ..., -7.5338e-01,
          9.9097e-01, -6.9534e-01],
        ...,
        [-6.0437e-01, -9.9979e-01,  9.9918e-01,  ..., -9.9879e-01,
         -3.2887e-01,  9.7080e-01],
        [-7.5407e-01,  9.7947e-01, -9.2877e-01,  ..., -9.9977e-01,
         -1.1378e-01, -6.8376e-01],
        [-5.2930e-01, -5.8608e-01, -9.9610e-01,  ...,  1.8574e-01,
```

```
          -9.5017e-01, -2.7476e-01]],

         [[-8.5251e-01, -3.5366e-01, -6.1704e-02,  ...,  9.4064e-01,
            2.0848e-02, -7.3322e-01],
          [-6.7167e-01, -9.8463e-01, -9.9220e-01,  ..., -4.1852e-01,
           -8.7637e-01,  3.6231e-01],
          [-3.5552e-01, -9.6855e-01,  7.4080e-01,  ...,  9.9866e-01,
            2.8850e-01, -5.5826e-01],
          ...,
          [ 8.6610e-01,  1.4357e-01, -8.7027e-01,  ..., -9.9851e-01,
           -8.8757e-01, -8.3230e-01],
          [-2.3924e-01,  9.7540e-01, -7.9089e-01,  ...,  7.4552e-01,
           -9.8136e-01, -8.5628e-01],
          [-5.8313e-01,  5.8426e-01,  6.3842e-02,  ..., -8.4698e-01,
           -1.6617e-01, -6.5636e-01]]],


        [[[-6.2759e-01, -9.3228e-01, -1.4100e-01,  ..., -9.6833e-01,
           -1.5692e-01,  1.8332e-01],
          [ 3.2893e-01, -4.2575e-01,  9.6818e-01,  ...,  9.5633e-01,
            8.0106e-01,  4.3867e-01],
          [-9.6506e-01,  1.9157e-01, -9.8877e-01,  ..., -9.7763e-01,
           -9.5829e-01, -7.5781e-01],
          ...,
          [ 6.4271e-01,  9.3885e-01,  9.9396e-01,  ...,  9.7752e-01,
            9.9248e-01,  9.9538e-01],
          [ 3.1051e-01,  9.9996e-01, -8.9545e-01,  ..., -4.9662e-01,
           -9.5031e-01, -7.7891e-01],
          [-2.2350e-02,  2.1585e-01,  8.4391e-01,  ..., -4.9840e-01,
           -8.1714e-01,  9.2148e-01]],

         [[ 8.6013e-01, -7.9700e-01,  7.4603e-01,  ..., -9.4202e-01,
            2.0643e-01, -5.7170e-01],
          [-8.2325e-01, -2.1976e-01,  9.9482e-01,  ...,  6.5822e-01,
            9.9881e-01,  9.0394e-01],
          [ 8.8695e-01,  9.7062e-01, -9.6625e-01,  ..., -7.3338e-01,
            9.9025e-01, -7.1487e-01],
          ...,
          [-6.1226e-01, -9.9979e-01,  9.9917e-01,  ..., -9.9882e-01,
           -3.4426e-01,  9.6981e-01],
          [-7.5634e-01,  9.7743e-01, -9.3091e-01,  ..., -9.9976e-01,
           -1.0992e-01, -6.8554e-01],
          [-5.2790e-01, -5.8655e-01, -9.9610e-01,  ...,  1.7254e-01,
           -9.4916e-01, -2.7509e-01]],

         [[-8.5168e-01, -3.3885e-01, -4.9741e-02,  ...,  9.3802e-01,
            7.1702e-03, -7.3471e-01],
          [-6.8242e-01, -9.8269e-01, -9.9027e-01,  ..., -4.4612e-01,
           -8.7594e-01,  3.6058e-01],
          [-3.3645e-01, -9.6738e-01,  7.5714e-01,  ...,  9.9888e-01,
```

```
         2.1817e-01, -5.6888e-01],
        ...,
        [ 8.6720e-01,  1.3779e-01, -8.8288e-01,  ..., -9.9833e-01,
         -8.9441e-01, -8.3493e-01],
        [-2.3213e-01,  9.7367e-01, -8.0772e-01,  ...,  7.4854e-01,
         -9.8285e-01, -8.6301e-01],
        [-5.8874e-01,  5.9293e-01,  5.1827e-02,  ..., -8.4568e-01,
         -1.7308e-01, -6.5046e-01]]],


      [[[-6.2970e-01, -9.3069e-01, -1.5489e-01,  ..., -9.6691e-01,
         -1.7642e-01,  1.7805e-01],
        [ 3.2196e-01, -4.0311e-01,  9.6133e-01,  ...,  9.5865e-01,
          8.1184e-01,  4.4704e-01],
        [-9.6504e-01,  1.8542e-01, -9.8950e-01,  ..., -9.7759e-01,
         -9.6056e-01, -7.4289e-01],
        ...,
        [ 6.3288e-01,  9.4728e-01,  9.9428e-01,  ...,  9.7838e-01,
          9.9308e-01,  9.9540e-01],
        [ 3.0072e-01,  9.9996e-01, -8.7692e-01,  ..., -5.1850e-01,
         -9.5096e-01, -7.8583e-01],
        [-2.6104e-02,  2.1810e-01,  8.4234e-01,  ..., -5.1100e-01,
         -8.1703e-01,  9.2033e-01]],

       [[ 8.6000e-01, -7.9430e-01,  7.4973e-01,  ..., -9.4241e-01,
          1.9785e-01, -5.7237e-01],
        [-8.2358e-01, -1.7249e-01,  9.9477e-01,  ...,  6.8495e-01,
          9.9878e-01,  9.0250e-01],
        [ 8.8394e-01,  9.7021e-01, -9.6454e-01,  ..., -7.5397e-01,
          9.9072e-01, -6.9368e-01],
        ...,
        [-5.8993e-01, -9.9982e-01,  9.9938e-01,  ..., -9.9886e-01,
         -3.0970e-01,  9.6981e-01],
        [-7.4944e-01,  9.8052e-01, -9.3381e-01,  ..., -9.9974e-01,
         -1.3117e-01, -6.9524e-01],
        [-5.3343e-01, -5.8677e-01, -9.9606e-01,  ...,  1.8682e-01,
         -9.4607e-01, -2.7156e-01]],

       [[-8.5281e-01, -3.4805e-01, -5.7820e-02,  ...,  9.3916e-01,
          3.1729e-02, -7.2519e-01],
        [-6.6755e-01, -9.8317e-01, -9.9167e-01,  ..., -4.3321e-01,
         -8.9082e-01,  3.8138e-01],
        [-3.4762e-01, -9.6129e-01,  7.3849e-01,  ...,  9.9892e-01,
          2.0761e-01, -5.5625e-01],
        ...,
        [ 8.7045e-01,  1.3655e-01, -8.6020e-01,  ..., -9.9825e-01,
         -8.9260e-01, -8.3073e-01],
        [-2.6049e-01,  9.8066e-01, -7.8194e-01,  ...,  7.1247e-01,
         -9.7949e-01, -8.6312e-01],
```

```
          [-5.8384e-01,  5.7579e-01,  1.0061e-01,  ..., -8.5402e-01,
           -1.6924e-01, -6.5107e-01]]]], grad_fn=<TanhBackward0>)
```

Discriminator

In the cGAN architecture, the chosen discriminator is a Patch GAN. Instead of classifying if the whole image is fake or not (binary classification), this discriminator tries to classify if each N × N patch in an image is real or fake.

The size N is given by the depth of the net. According to this table :

Number of layers	N
1	16
2	34
3	70
4	142
5	286
6	574

The number of layers actually means the number of layers with `kernel=(4,4)`, `padding=(1,1)` and `stride=(2,2)`. These layers are followed by 2 layers with `kernel=(4,4)`, `padding=(1,1)` and `stride=(1,1)`. In our case we are going to create a 70x70 PatchGAN.

Let's first create a few helping classes.

```
class conv_block(nn.Module):
    def __init__(self, in_ch, out_ch, use_batchnorm=True, stride=2):
        super(conv_block, self).__init__()
        if use_batchnorm:
            self.conv = nn.Sequential(
                nn.Conv2d(in_ch, out_ch, kernel_size=4, padding=1,
stride=stride),
                nn.BatchNorm2d(out_ch),
                nn.LeakyReLU(negative_slope=0.2, inplace=True)
            )
        else:
            self.conv = nn.Sequential(
                nn.Conv2d(in_ch, out_ch, kernel_size=4, padding=1,
stride=stride),
                nn.LeakyReLU(negative_slope=0.2, inplace=True)
            )

    def forward(self, x):
        x = self.conv(x)
        return x
```

```python
class out_block(nn.Module):
    def __init__(self, in_ch, out_ch):
        super(out_block, self).__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(in_ch, 1, kernel_size=4, padding=1, stride=1),
            nn.Sigmoid()
        )

    def forward(self, x):
        x = self.conv(x)
        return x
```

Now let's create the Patch GAN discriminator. As we want a 70x70 Patch GAN, the architecture will be as follows :

```
1. C64  - K4, P1, S2
2. C128 - K4, P1, S2
3. C256 - K4, P1, S2
4. C512 - K4, P1, S1
5. C1   - K4, P1, S1 (output)
```

Where Ck denotes a convolution block with k filters, Kk a kernel of size k, Pk is the padding size and Sk the stride applied. *Note :* For the first layer, we do not use batchnorm.

These network has 2769729 learnable parameters.

From conv2D: (6x4x4+1)x64 + (64x4x4+1)x128 + (128x4x4+1)x256 + (256x4x4+1)x512 + (512x4x4+1)x1

= 2767809

Plus the norm batch parameters

64x2+128x2+256x2+512x2 = 1920

Total = 2767809 + 1920 = 2769729

```python
class PatchGAN(nn.Module):
    def __init__(self, n_channels, n_classes):
        super(PatchGAN, self).__init__()
        self.conv1 = conv_block(n_channels, 64)
        self.conv2 = conv_block(64, 128)
        self.conv3 = conv_block(128,256)
        self.conv4 = conv_block(256,512, stride=1)
        # output layer
        self.out = out_block(512, n_classes)

    def forward(self, x1, x2):
        x = torch.cat([x2, x1], dim=1)
        x = self.conv1(x)
```

```
        x = self.conv2(x)
        x = self.conv3(x)
        x = self.conv4(x)
        x = self.out(x)
        return x

# We have 6 input channels as we concatenate 2 images (with 3 channels
each)
discriminator = PatchGAN(6, 1)

total_params = sum(p.numel() for p in discriminator.parameters() if
p.requires_grad)

print(f"Total learnable parameters : {total_params}")
print(discriminator)
```

```
Total learnable parameters : 2769729
PatchGAN(
  (conv1): conv_block(
    (conv): Sequential(
      (0): Conv2d(6, 64, kernel_size=(4, 4), stride=(2, 2),
padding=(1, 1))
      (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): LeakyReLU(negative_slope=0.2, inplace=True)
    )
  )
  (conv2): conv_block(
    (conv): Sequential(
      (0): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2),
padding=(1, 1))
      (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): LeakyReLU(negative_slope=0.2, inplace=True)
    )
  )
  (conv3): conv_block(
    (conv): Sequential(
      (0): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2),
padding=(1, 1))
      (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): LeakyReLU(negative_slope=0.2, inplace=True)
    )
  )
  (conv4): conv_block(
    (conv): Sequential(
      (0): Conv2d(256, 512, kernel_size=(4, 4), stride=(1, 1),
padding=(1, 1))
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
```

```
track_running_stats=True)
      (2): LeakyReLU(negative_slope=0.2, inplace=True)
    )
  )
  (out): out_block(
    (conv): Sequential(
      (0): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1),
padding=(1, 1))
      (1): Sigmoid()
    )
  )
)
```

Loss functions

The global loss function will be made of 2 parts :

- The vanilla GAN loss, in which the discriminator tries to maximize the probability it correctly classifies reals and fakes and the generator tries to minimize the probability that the discriminator will predict its outputs are fake.
- An auxiliary image reconstruction objective, in which the generator not only has to fool the discriminator, but also generate an image that is near to the ground truth image.

Therefore, the loss can be defined as:

$$G^{\text{¿}} = arg\min_{G}\max_{D} L_{cGAN}(G, D) + \lambda L_{L1}(G)$$

In which

$$L_{cGAN}(G, D) = E_{x,y}[log D(x, y)] + E_{x,z}[log(1 - D(x, G(x, z)))]$$

$$L_{L1}(G) = E_{x,y,z}[\|y - G(x, z)\|_1]$$

```
# Loss functions
criterion_GAN = torch.nn.BCELoss()
criterion_pixelwise = torch.nn.L1Loss()

# Loss weight of L1 pixel-wise loss between translated image and real
image
lambda_pixel = 100
```

Training and evaluating models

```
# parameters
num_epochs = 200        # number of epochs of training
batch_size = 16         # size of the batches
lr = 2e-4               # learning rate
b1 = 0.5                # decay of first order momentum of gradient
```

```
b2 = 0.999                  # decay of second order momentum of gradient
img_height = 256            # size of image height
img_width = 256             # size of image width
channels = 3                # number of image channels
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

Download the Facades dataset

```
dataset_path = kagglehub.dataset_download("kokeyehya/cmp-facade-db-
base")

print("Path to dataset files:", dataset_path)
```

```
Downloading from
https://www.kaggle.com/api/v1/datasets/download/kokeyehya/cmp-facade-
db-base?dataset_version_number=1...

100%|██████████| 34.9M/34.9M [00:02<00:00, 12.5MB/s]

Extracting files...



Path to dataset files: /root/.cache/kagglehub/datasets/kokeyehya/cmp-
facade-db-base/versions/1
```

Configure the dataloader

```
class FacadeDataset(Dataset):
    def __init__(self, root, transforms_=None, mode='train'):
        self.transform = transforms.Compose(transforms_)
        self.images_path = sorted(glob.glob(root + '/base/*.jpg'))

        if mode == 'train':
            self.images_path =
self.images_path[:int(len(self.images_path) * 0.95)]
        elif mode == 'val':
            self.images_path =
self.images_path[int(len(self.images_path) * 0.95):]
        else:
            raise Exception('Invalid mode! It must be either train or
val')

        self.masks_path = [image.split(".jpg")[0] + ".png" for image
in self.images_path]
        assert len(self.images_path) == len(self.masks_path), "Number
of images and masks must be the same"


    def __getitem__(self, index):
```

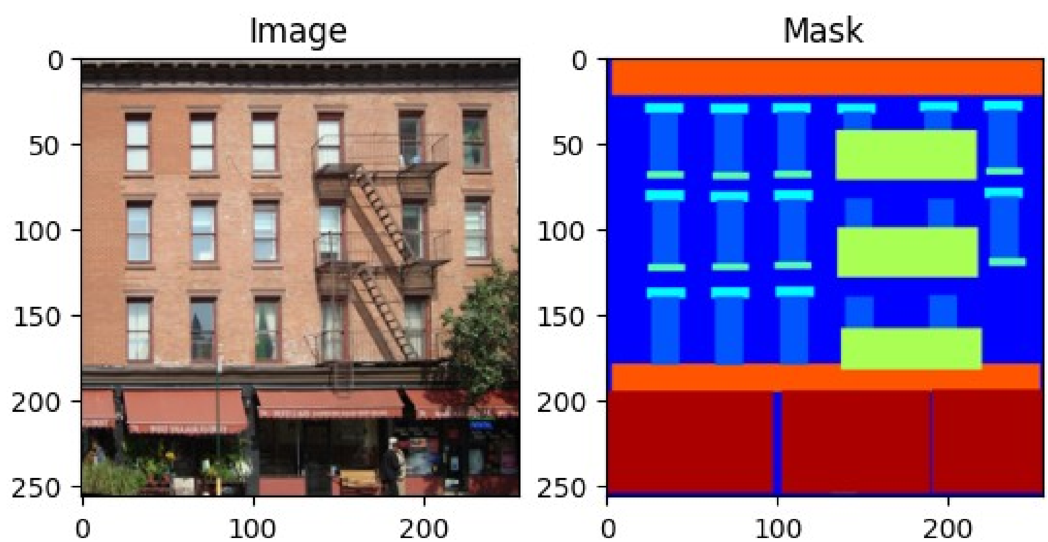
```
        img = Image.open(self.images_path[index])
        mask = Image.open(self.masks_path[index])
        mask = mask.convert('RGB')

        img = self.transform(img)
        mask = self.transform(mask)

        return img, mask

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

# Configure dataloaders
transforms_ = [transforms.Resize((img_height, img_width),
Image.BICUBIC),
               transforms.ToTensor()]

dataloader = DataLoader(FacadeDataset(dataset_path,
transforms_=transforms_),
                        batch_size=batch_size, shuffle=True,
drop_last=True, num_workers=4)

val_dataloader = DataLoader(FacadeDataset(dataset_path,
transforms_=transforms_, mode='val'),
                            batch_size=batch_size, shuffle=False)
```

```
/usr/local/lib/python3.11/dist-packages/torch/utils/data/
dataloader.py:624: UserWarning: This DataLoader will create 4 worker
processes in total. Our suggested max number of worker in current
system is 2, which is smaller than what this DataLoader is going to
create. Please be aware that excessive worker creation might get
DataLoader running slow or even freeze, lower the worker number to
avoid potential slowness/freeze if necessary.
  warnings.warn(
```

Check if the loading works and add few helper functions

```
def plot2x2Array(image, mask):
    f, axarr = plt.subplots(1, 2)
    axarr[0].imshow(image)
    axarr[1].imshow(mask)

    axarr[0].set_title('Image')
    axarr[1].set_title('Mask')


def reverse_transform(image):
    image = image.numpy().transpose((1, 2, 0))
    image = np.clip(image, 0, 1)
    image = (image * 255).astype(np.uint8)
```

```python
    return image

def plot2x3Array(image, mask,predict):
    f, axarr = plt.subplots(1,3,figsize=(15,15))
    axarr[0].imshow(image)
    axarr[1].imshow(mask)
    axarr[2].imshow(predict)
    axarr[0].set_title('input')
    axarr[1].set_title('real')
    axarr[2].set_title('fake')

images, masks = next(iter(dataloader))

for i in range(5):
    image = reverse_transform(images[i])
    mask = reverse_transform(masks[i])
    plot2x2Array(image, mask)
```



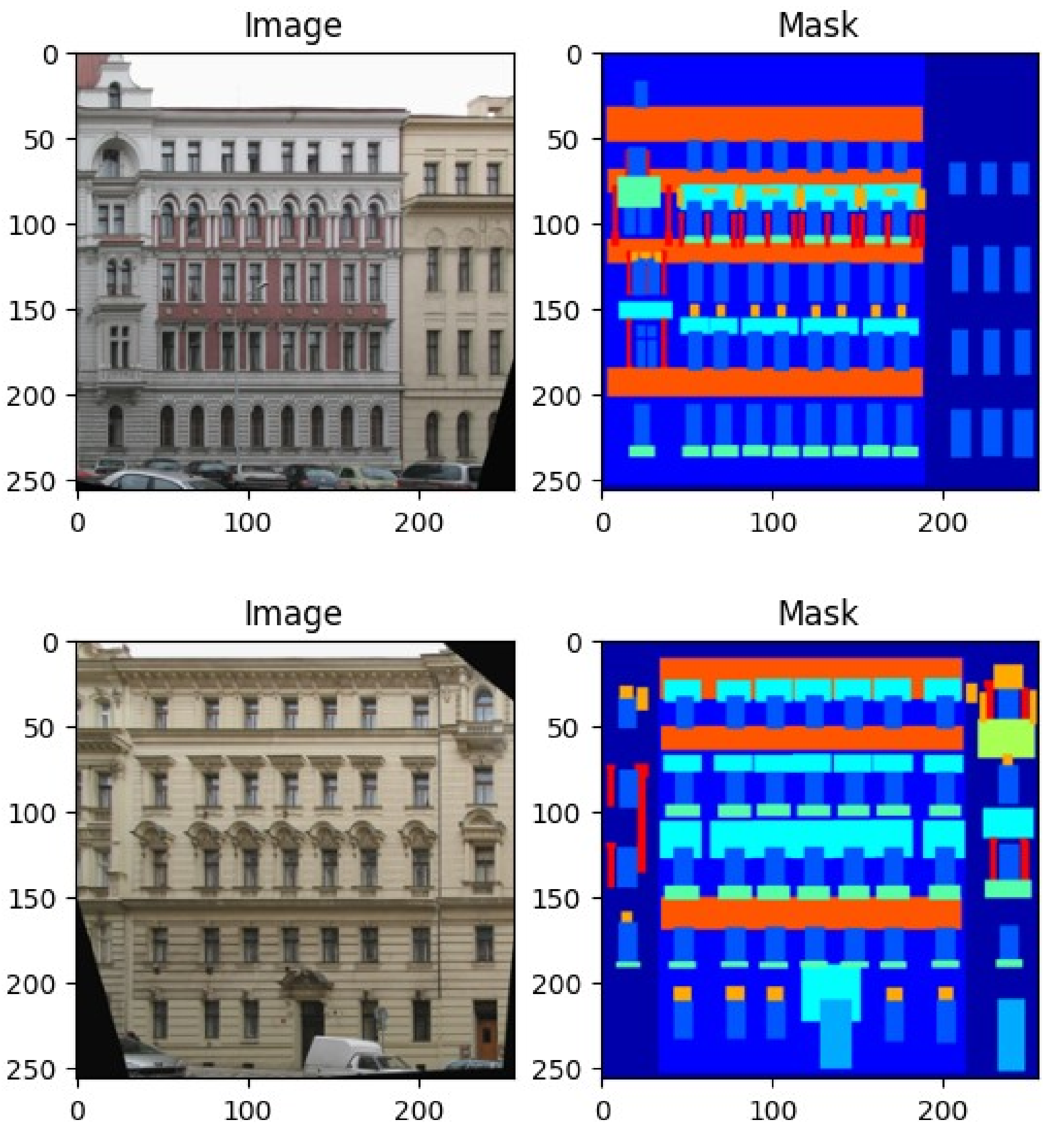
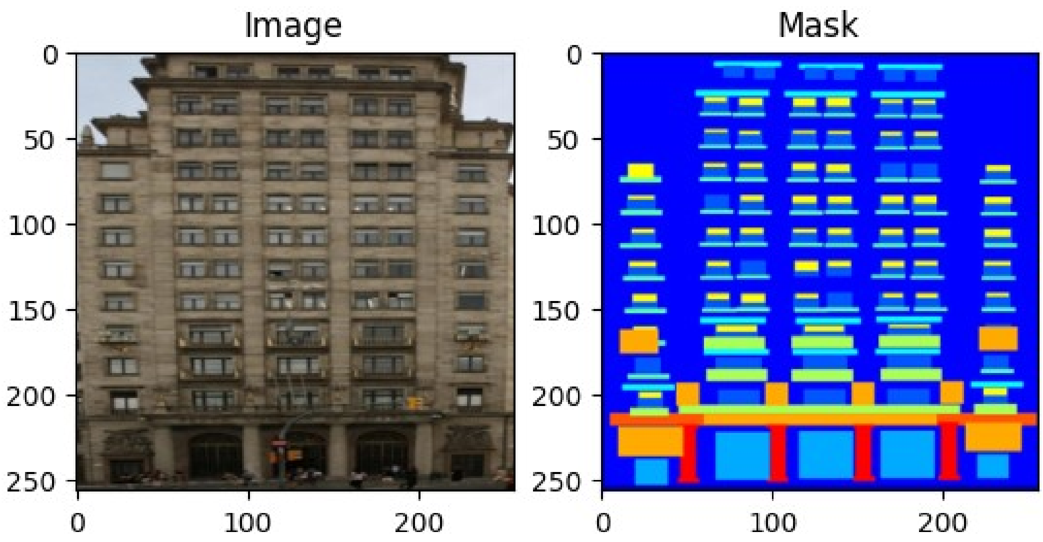
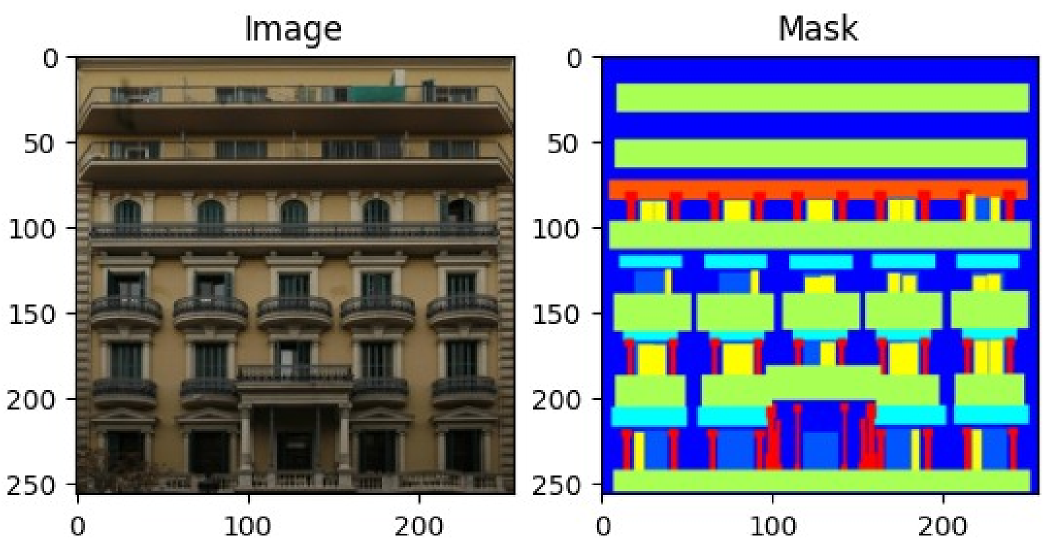


Image
Mask
0
50
100
150
200
250
0
100
200
Image
Mask
0
50
100
150
200
250
0
100
200





Initialize our GAN

```
# Calculate output of image discriminator (PatchGAN)
patch_size = (1, img_height//2**3-2, img_width//2**3-2)

generator = generator.to(device)
discriminator = discriminator.to(device)

# Optimizers
optimizer_G = torch.optim.Adam(generator.parameters(), lr=lr,
betas=(b1, b2))
optimizer_D = torch.optim.Adam(discriminator.parameters(), lr=lr,
betas=(b1, b2))
```

Additional auxiliary functions

```python
def save_model(epoch, loss_D, loss_G):
    # save your model weights
    torch.save({
                'epoch': epoch,
                'model_state_dict': generator.state_dict(),
                'optimizer_state_dict': optimizer_G.state_dict(),
                'loss': loss_G,
                }, 'generator_'+str(epoch)+'.pth')
    torch.save({
                'epoch': epoch,
                'model_state_dict': discriminator.state_dict(),
                'optimizer_state_dict': optimizer_D.state_dict(),
                'loss': loss_D,
                }, 'discriminator_'+str(epoch)+'.pth')

def weights_init_normal(m):
  classname = m.__class__.__name__
  if classname.find('Conv') != -1:
      torch.nn.init.normal_(m.weight.data, 0.0, 0.02)
  elif classname.find('BatchNorm2d') != -1:
      torch.nn.init.normal_(m.weight.data, 1.0, 0.02)
      torch.nn.init.constant_(m.bias.data, 0.0)
```

Training the model !

```python
# ----------
#  Training
# ----------

losses = []

# Initialize weights
generator.apply(weights_init_normal)
discriminator.apply(weights_init_normal)

# train the network
discriminator.train()
generator.train()
print_every = 400

for epoch in range(num_epochs):
    for i, batch in enumerate(dataloader):
        # Model inputs
        images, masks = batch
        images = images.to(device)
        masks = masks.to(device)

        # Discriminator labels
        valid = torch.ones((images.size(0), *patch_size),
requires_grad=False).to(device)
```

```
        fake = torch.zeros((images.size(0), *patch_size),
requires_grad=False).to(device)

        # ------------------
        #  Train Generators
        # ------------------

        optimizer_G.zero_grad()

        generated_images = generator(masks) # This line was missing


        # GAN loss
        gan_loss =
criterion_GAN(discriminator(generated_images.detach(), masks), valid)

        # Pixel-wise loss
        pixel_loss = criterion_pixelwise(generated_images, images)
        # Total loss
        loss_G = 0.5*(pixel_loss + gan_loss)

        loss_G.backward()
        optimizer_G.step()

        # ---------------------
        #  Train Discriminator
        # ---------------------

        optimizer_D.zero_grad()

        # Real loss
        pred_real = discriminator(images, masks)
        loss_real = criterion_GAN(pred_real, valid)

        # Fake loss
        pred_fake = discriminator(generated_images.detach(), masks)
        loss_fake = criterion_GAN(pred_fake, fake)

        # Total loss
        loss_D = 0.5 * (loss_real + loss_fake)

        loss_D.backward()
        optimizer_D.step()

        # Print some loss stats
        if i % print_every == 0:
            print(f'Epoch [{epoch+1}/{num_epochs}]
[{i}/{len(dataloader)}] | d_loss: {loss_D.item():6.4f} | g_loss:
{loss_G.item():6.4f}')

    # Keep track of discriminator loss and generator loss
```

```python
    losses.append((loss_D.item(), loss_G.item()))

    if (epoch + 1) % 100 == 0:
        print('Saving model...')
        save_model(epoch+1, loss_D, loss_G)
```

```
Epoch [1/200][0/22] | d_loss: 0.8844 | g_loss: 0.5099
Epoch [2/200][0/22] | d_loss: 0.5792 | g_loss: 0.5312
Epoch [3/200][0/22] | d_loss: 0.1054 | g_loss: 1.3309
Epoch [4/200][0/22] | d_loss: 0.0588 | g_loss: 1.4195
Epoch [5/200][0/22] | d_loss: 0.0196 | g_loss: 2.0817
Epoch [6/200][0/22] | d_loss: 0.0223 | g_loss: 2.3777
Epoch [7/200][0/22] | d_loss: 0.0075 | g_loss: 2.6470
Epoch [8/200][0/22] | d_loss: 0.0080 | g_loss: 2.6361
Epoch [9/200][0/22] | d_loss: 0.0654 | g_loss: 1.5581
Epoch [10/200][0/22] | d_loss: 0.0156 | g_loss: 2.3048
Epoch [11/200][0/22] | d_loss: 0.0103 | g_loss: 2.3807
Epoch [12/200][0/22] | d_loss: 0.0081 | g_loss: 2.5081
Epoch [13/200][0/22] | d_loss: 0.0060 | g_loss: 2.7321
Epoch [14/200][0/22] | d_loss: 0.0045 | g_loss: 2.8963
Epoch [15/200][0/22] | d_loss: 0.0044 | g_loss: 2.6959
Epoch [16/200][0/22] | d_loss: 0.0036 | g_loss: 2.8373
Epoch [17/200][0/22] | d_loss: 0.0037 | g_loss: 2.8111
Epoch [18/200][0/22] | d_loss: 0.0040 | g_loss: 2.7468
Epoch [19/200][0/22] | d_loss: 0.0025 | g_loss: 3.0127
Epoch [20/200][0/22] | d_loss: 0.0022 | g_loss: 3.1018
Epoch [21/200][0/22] | d_loss: 0.0023 | g_loss: 3.2132
Epoch [22/200][0/22] | d_loss: 0.0021 | g_loss: 3.2050
Epoch [23/200][0/22] | d_loss: 0.0015 | g_loss: 3.2380
Epoch [24/200][0/22] | d_loss: 0.0015 | g_loss: 3.2899
Epoch [25/200][0/22] | d_loss: 0.0016 | g_loss: 3.2433
Epoch [26/200][0/22] | d_loss: 0.0013 | g_loss: 3.3341
Epoch [27/200][0/22] | d_loss: 0.0012 | g_loss: 3.3894
Epoch [28/200][0/22] | d_loss: 0.0010 | g_loss: 3.5616
Epoch [29/200][0/22] | d_loss: 0.0013 | g_loss: 3.4002
Epoch [30/200][0/22] | d_loss: 0.0010 | g_loss: 3.4618
Epoch [31/200][0/22] | d_loss: 0.0009 | g_loss: 3.5188
Epoch [32/200][0/22] | d_loss: 0.0013 | g_loss: 3.5554
Epoch [33/200][0/22] | d_loss: 2.6107 | g_loss: 0.3379
Epoch [34/200][0/22] | d_loss: 0.7138 | g_loss: 0.4581
Epoch [35/200][0/22] | d_loss: 0.6702 | g_loss: 0.4354
Epoch [36/200][0/22] | d_loss: 0.7213 | g_loss: 0.5010
Epoch [37/200][0/22] | d_loss: 0.3543 | g_loss: 0.6876
Epoch [38/200][0/22] | d_loss: 0.0135 | g_loss: 2.3154
Epoch [39/200][0/22] | d_loss: 0.0092 | g_loss: 2.6732
Epoch [40/200][0/22] | d_loss: 0.0046 | g_loss: 2.8755
Epoch [41/200][0/22] | d_loss: 0.0051 | g_loss: 2.7283
Epoch [42/200][0/22] | d_loss: 0.0033 | g_loss: 2.9388
Epoch [43/200][0/22] | d_loss: 0.0042 | g_loss: 3.1576
Epoch [44/200][0/22] | d_loss: 0.0032 | g_loss: 2.9960
```

```
Epoch [45/200][0/22] | d_loss: 0.0024 | g_loss: 3.2236
Epoch [46/200][0/22] | d_loss: 0.0022 | g_loss: 3.2706
Epoch [47/200][0/22] | d_loss: 0.0026 | g_loss: 3.2863
Epoch [48/200][0/22] | d_loss: 0.0026 | g_loss: 3.1846
Epoch [49/200][0/22] | d_loss: 0.0022 | g_loss: 3.2418
Epoch [50/200][0/22] | d_loss: 0.0015 | g_loss: 3.2877
Epoch [51/200][0/22] | d_loss: 0.0012 | g_loss: 3.4576
Epoch [52/200][0/22] | d_loss: 0.0015 | g_loss: 3.2756
Epoch [53/200][0/22] | d_loss: 0.0015 | g_loss: 3.3541
Epoch [54/200][0/22] | d_loss: 0.0015 | g_loss: 3.4937
Epoch [55/200][0/22] | d_loss: 0.0014 | g_loss: 3.4107
Epoch [56/200][0/22] | d_loss: 0.0013 | g_loss: 3.6313
Epoch [57/200][0/22] | d_loss: 0.0012 | g_loss: 3.5824
Epoch [58/200][0/22] | d_loss: 0.0016 | g_loss: 3.5075
Epoch [59/200][0/22] | d_loss: 0.0009 | g_loss: 3.7088
Epoch [60/200][0/22] | d_loss: 0.0012 | g_loss: 3.4013
Epoch [61/200][0/22] | d_loss: 0.0012 | g_loss: 3.7128
Epoch [62/200][0/22] | d_loss: 0.0008 | g_loss: 3.5953
Epoch [63/200][0/22] | d_loss: 0.0009 | g_loss: 3.7340
Epoch [64/200][0/22] | d_loss: 0.0007 | g_loss: 3.7612
Epoch [65/200][0/22] | d_loss: 0.0016 | g_loss: 3.1345
Epoch [66/200][0/22] | d_loss: 0.0010 | g_loss: 3.7048
Epoch [67/200][0/22] | d_loss: 0.0011 | g_loss: 3.5210
Epoch [68/200][0/22] | d_loss: 0.0013 | g_loss: 3.2758
Epoch [69/200][0/22] | d_loss: 0.0007 | g_loss: 3.8709
Epoch [70/200][0/22] | d_loss: 0.0007 | g_loss: 3.7371
Epoch [71/200][0/22] | d_loss: 0.6431 | g_loss: 0.6092
Epoch [72/200][0/22] | d_loss: 0.7526 | g_loss: 1.5058
Epoch [73/200][0/22] | d_loss: 0.3509 | g_loss: 0.6216
Epoch [74/200][0/22] | d_loss: 0.0186 | g_loss: 2.5135
Epoch [75/200][0/22] | d_loss: 0.0076 | g_loss: 2.4891
Epoch [76/200][0/22] | d_loss: 0.0043 | g_loss: 2.7270
Epoch [77/200][0/22] | d_loss: 0.0031 | g_loss: 2.9891
Epoch [78/200][0/22] | d_loss: 0.0023 | g_loss: 3.2141
Epoch [79/200][0/22] | d_loss: 0.0030 | g_loss: 2.8961
Epoch [80/200][0/22] | d_loss: 0.0030 | g_loss: 3.1170
Epoch [81/200][0/22] | d_loss: 0.0023 | g_loss: 3.1525
Epoch [82/200][0/22] | d_loss: 0.0017 | g_loss: 3.1726
Epoch [83/200][0/22] | d_loss: 0.0016 | g_loss: 3.4916
Epoch [84/200][0/22] | d_loss: 0.0019 | g_loss: 3.4978
Epoch [85/200][0/22] | d_loss: 0.0019 | g_loss: 3.0839
Epoch [86/200][0/22] | d_loss: 0.0016 | g_loss: 3.2676
Epoch [87/200][0/22] | d_loss: 0.0027 | g_loss: 3.4256
Epoch [88/200][0/22] | d_loss: 0.0016 | g_loss: 3.5193
Epoch [89/200][0/22] | d_loss: 0.0015 | g_loss: 3.2411
Epoch [90/200][0/22] | d_loss: 0.0008 | g_loss: 3.6954
Epoch [91/200][0/22] | d_loss: 0.0012 | g_loss: 3.4074
Epoch [92/200][0/22] | d_loss: 0.0016 | g_loss: 3.1814
Epoch [93/200][0/22] | d_loss: 0.0083 | g_loss: 2.2639
```

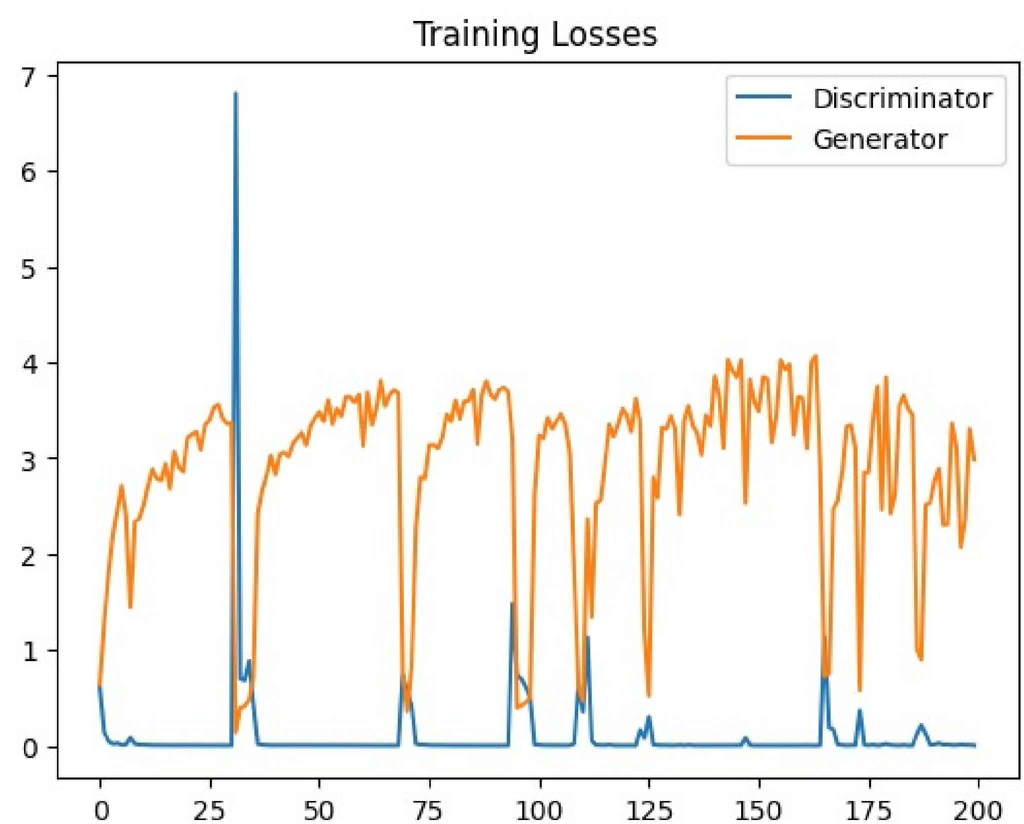
```
Epoch [94/200][0/22] | d_loss: 0.0016 | g_loss: 3.1983
Epoch [95/200][0/22] | d_loss: 0.0011 | g_loss: 3.5315
Epoch [96/200][0/22] | d_loss: 4.6471 | g_loss: 0.0402
Epoch [97/200][0/22] | d_loss: 0.7442 | g_loss: 0.3779
Epoch [98/200][0/22] | d_loss: 0.6803 | g_loss: 0.3695
Epoch [99/200][0/22] | d_loss: 0.5959 | g_loss: 0.4657
Epoch [100/200][0/22] | d_loss: 0.5079 | g_loss: 0.5649
Saving model...
Epoch [101/200][0/22] | d_loss: 0.0139 | g_loss: 2.4789
Epoch [102/200][0/22] | d_loss: 0.0036 | g_loss: 3.1997
Epoch [103/200][0/22] | d_loss: 0.0100 | g_loss: 3.3650
Epoch [104/200][0/22] | d_loss: 0.0018 | g_loss: 3.4183
Epoch [105/200][0/22] | d_loss: 0.0013 | g_loss: 3.5944
Epoch [106/200][0/22] | d_loss: 0.0023 | g_loss: 3.1933
Epoch [107/200][0/22] | d_loss: 0.0028 | g_loss: 3.0457
Epoch [108/200][0/22] | d_loss: 0.0013 | g_loss: 3.4868
Epoch [109/200][0/22] | d_loss: 0.0035 | g_loss: 3.6685
Epoch [110/200][0/22] | d_loss: 0.0748 | g_loss: 3.4746
Epoch [111/200][0/22] | d_loss: 0.6125 | g_loss: 0.3884
Epoch [112/200][0/22] | d_loss: 0.2836 | g_loss: 0.9920
Epoch [113/200][0/22] | d_loss: 0.4663 | g_loss: 0.6191
Epoch [114/200][0/22] | d_loss: 0.1250 | g_loss: 0.8579
Epoch [115/200][0/22] | d_loss: 0.0124 | g_loss: 2.0983
Epoch [116/200][0/22] | d_loss: 0.0028 | g_loss: 3.0749
Epoch [117/200][0/22] | d_loss: 0.0026 | g_loss: 3.2486
Epoch [118/200][0/22] | d_loss: 0.0024 | g_loss: 3.2320
Epoch [119/200][0/22] | d_loss: 0.0040 | g_loss: 2.7224
Epoch [120/200][0/22] | d_loss: 0.0030 | g_loss: 2.8716
Epoch [121/200][0/22] | d_loss: 0.0016 | g_loss: 3.4493
Epoch [122/200][0/22] | d_loss: 0.0013 | g_loss: 3.2977
Epoch [123/200][0/22] | d_loss: 0.0026 | g_loss: 2.9347
Epoch [124/200][0/22] | d_loss: 0.0010 | g_loss: 3.5379
Epoch [125/200][0/22] | d_loss: 0.4859 | g_loss: 0.4143
Epoch [126/200][0/22] | d_loss: 0.0523 | g_loss: 1.3639
Epoch [127/200][0/22] | d_loss: 0.1377 | g_loss: 2.6772
Epoch [128/200][0/22] | d_loss: 0.0058 | g_loss: 2.8761
Epoch [129/200][0/22] | d_loss: 0.0030 | g_loss: 2.9395
Epoch [130/200][0/22] | d_loss: 0.0031 | g_loss: 3.3003
Epoch [131/200][0/22] | d_loss: 0.0019 | g_loss: 3.2278
Epoch [132/200][0/22] | d_loss: 0.0022 | g_loss: 3.3061
Epoch [133/200][0/22] | d_loss: 0.0012 | g_loss: 3.3906
Epoch [134/200][0/22] | d_loss: 0.0020 | g_loss: 3.0453
Epoch [135/200][0/22] | d_loss: 0.0011 | g_loss: 3.6148
Epoch [136/200][0/22] | d_loss: 0.0011 | g_loss: 3.5730
Epoch [137/200][0/22] | d_loss: 0.0012 | g_loss: 3.4175
Epoch [138/200][0/22] | d_loss: 0.0027 | g_loss: 2.8867
Epoch [139/200][0/22] | d_loss: 0.0032 | g_loss: 2.7924
Epoch [140/200][0/22] | d_loss: 0.0016 | g_loss: 3.3749
Epoch [141/200][0/22] | d_loss: 0.0012 | g_loss: 3.5384
```

```
Epoch [142/200][0/22] | d_loss: 0.0009 | g_loss: 3.7157
Epoch [143/200][0/22] | d_loss: 0.0010 | g_loss: 3.4188
Epoch [144/200][0/22] | d_loss: 0.0007 | g_loss: 3.6115
Epoch [145/200][0/22] | d_loss: 0.0008 | g_loss: 3.5965
Epoch [146/200][0/22] | d_loss: 0.0011 | g_loss: 3.7474
Epoch [147/200][0/22] | d_loss: 0.0017 | g_loss: 3.6124
Epoch [148/200][0/22] | d_loss: 0.0010 | g_loss: 3.3787
Epoch [149/200][0/22] | d_loss: 0.0496 | g_loss: 2.9359
Epoch [150/200][0/22] | d_loss: 0.0018 | g_loss: 3.6618
Epoch [151/200][0/22] | d_loss: 0.0050 | g_loss: 2.5342
Epoch [152/200][0/22] | d_loss: 0.0012 | g_loss: 3.5222
Epoch [153/200][0/22] | d_loss: 0.0011 | g_loss: 3.4294
Epoch [154/200][0/22] | d_loss: 0.0073 | g_loss: 2.3113
Epoch [155/200][0/22] | d_loss: 0.0008 | g_loss: 3.6452
Epoch [156/200][0/22] | d_loss: 0.0010 | g_loss: 3.8844
Epoch [157/200][0/22] | d_loss: 0.0005 | g_loss: 3.9672
Epoch [158/200][0/22] | d_loss: 0.0009 | g_loss: 3.6777
Epoch [159/200][0/22] | d_loss: 0.0006 | g_loss: 3.7713
Epoch [160/200][0/22] | d_loss: 0.0003 | g_loss: 4.1792
Epoch [161/200][0/22] | d_loss: 0.0007 | g_loss: 3.7042
Epoch [162/200][0/22] | d_loss: 0.0049 | g_loss: 4.0408
Epoch [163/200][0/22] | d_loss: 0.0009 | g_loss: 3.5965
Epoch [164/200][0/22] | d_loss: 0.0029 | g_loss: 4.2410
Epoch [165/200][0/22] | d_loss: 0.0014 | g_loss: 3.1812
Epoch [166/200][0/22] | d_loss: 0.0008 | g_loss: 3.6014
Epoch [167/200][0/22] | d_loss: 0.9649 | g_loss: 1.1068
Epoch [168/200][0/22] | d_loss: 0.6276 | g_loss: 2.1502
Epoch [169/200][0/22] | d_loss: 0.0447 | g_loss: 1.5274
Epoch [170/200][0/22] | d_loss: 0.0097 | g_loss: 2.3951
Epoch [171/200][0/22] | d_loss: 0.0122 | g_loss: 2.0448
Epoch [172/200][0/22] | d_loss: 0.0027 | g_loss: 3.0458
Epoch [173/200][0/22] | d_loss: 0.0027 | g_loss: 2.8475
Epoch [174/200][0/22] | d_loss: 0.0025 | g_loss: 2.8933
Epoch [175/200][0/22] | d_loss: 0.1616 | g_loss: 2.3592
Epoch [176/200][0/22] | d_loss: 0.0071 | g_loss: 3.2994
Epoch [177/200][0/22] | d_loss: 0.0047 | g_loss: 2.7168
Epoch [178/200][0/22] | d_loss: 0.0017 | g_loss: 3.2367
Epoch [179/200][0/22] | d_loss: 0.0030 | g_loss: 3.0755
Epoch [180/200][0/22] | d_loss: 0.0010 | g_loss: 3.5613
Epoch [181/200][0/22] | d_loss: 0.0013 | g_loss: 3.2570
Epoch [182/200][0/22] | d_loss: 0.0008 | g_loss: 3.6740
Epoch [183/200][0/22] | d_loss: 0.0183 | g_loss: 1.8225
Epoch [184/200][0/22] | d_loss: 0.0023 | g_loss: 3.6513
Epoch [185/200][0/22] | d_loss: 0.0010 | g_loss: 3.9495
Epoch [186/200][0/22] | d_loss: 0.0014 | g_loss: 3.4392
Epoch [187/200][0/22] | d_loss: 0.0023 | g_loss: 2.9371
Epoch [188/200][0/22] | d_loss: 0.4727 | g_loss: 3.8849
Epoch [189/200][0/22] | d_loss: 0.1633 | g_loss: 1.0227
Epoch [190/200][0/22] | d_loss: 0.0404 | g_loss: 1.5395
```

```
Epoch [191/200][0/22] | d_loss: 0.0360 | g_loss: 1.5069
Epoch [192/200][0/22] | d_loss: 0.0111 | g_loss: 2.3102
Epoch [193/200][0/22] | d_loss: 0.0099 | g_loss: 2.3062
Epoch [194/200][0/22] | d_loss: 0.0735 | g_loss: 1.1676
Epoch [195/200][0/22] | d_loss: 0.0071 | g_loss: 2.6053
Epoch [196/200][0/22] | d_loss: 0.0075 | g_loss: 3.0789
Epoch [197/200][0/22] | d_loss: 0.0085 | g_loss: 2.7322
Epoch [198/200][0/22] | d_loss: 0.0025 | g_loss: 3.0146
Epoch [199/200][0/22] | d_loss: 0.0067 | g_loss: 3.0088
Epoch [200/200][0/22] | d_loss: 0.0013 | g_loss: 3.5185
Saving model...
```

Observation of the loss along the training

```
fig, ax = plt.subplots()
losses = np.array(losses)
plt.plot(losses.T[0], label='Discriminator')
plt.plot(losses.T[1], label='Generator')
plt.title("Training Losses")
plt.legend()

<matplotlib.legend.Legend at 0x7cd4ea280e10>
```



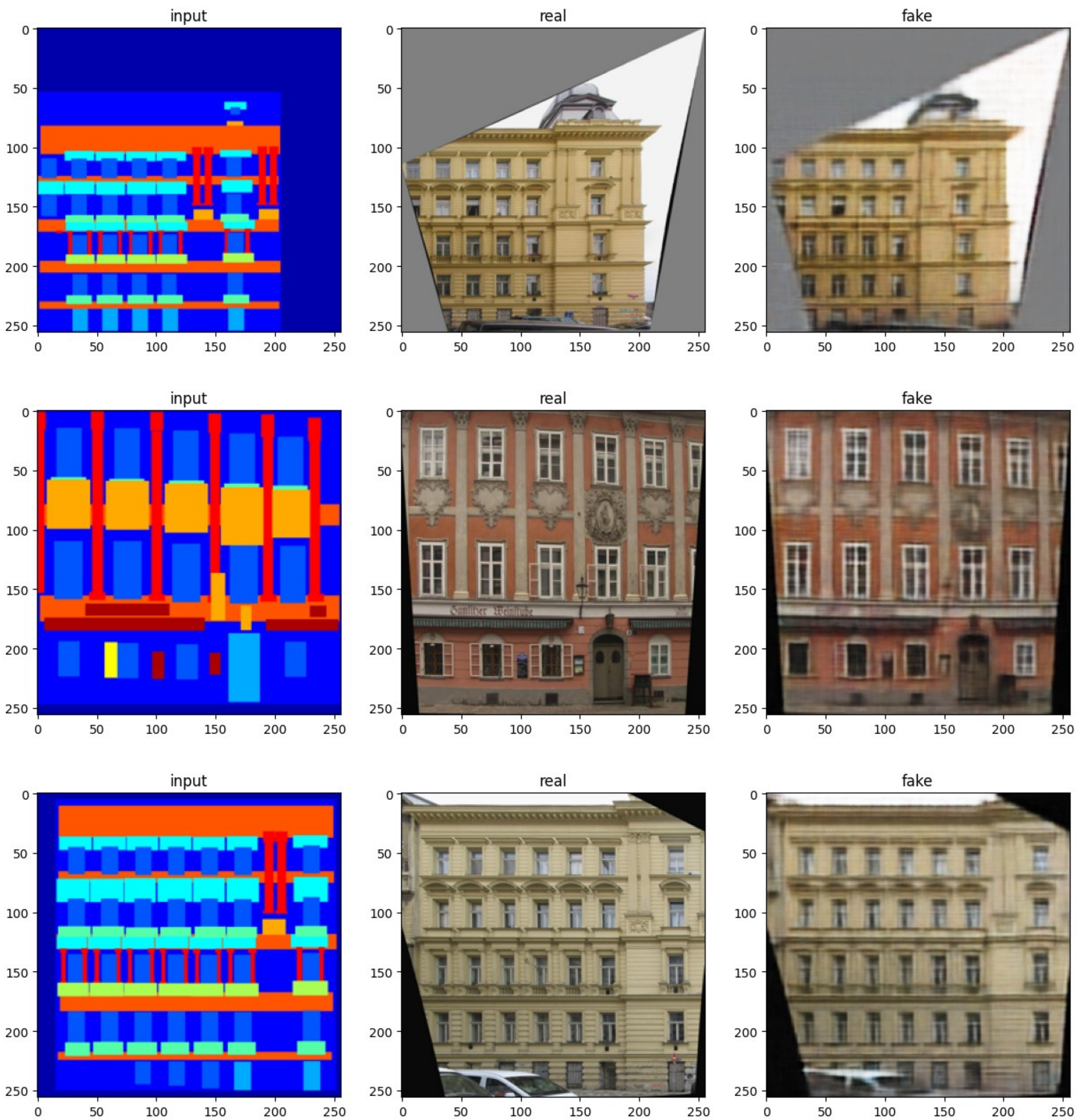
Evaluate the cGAN

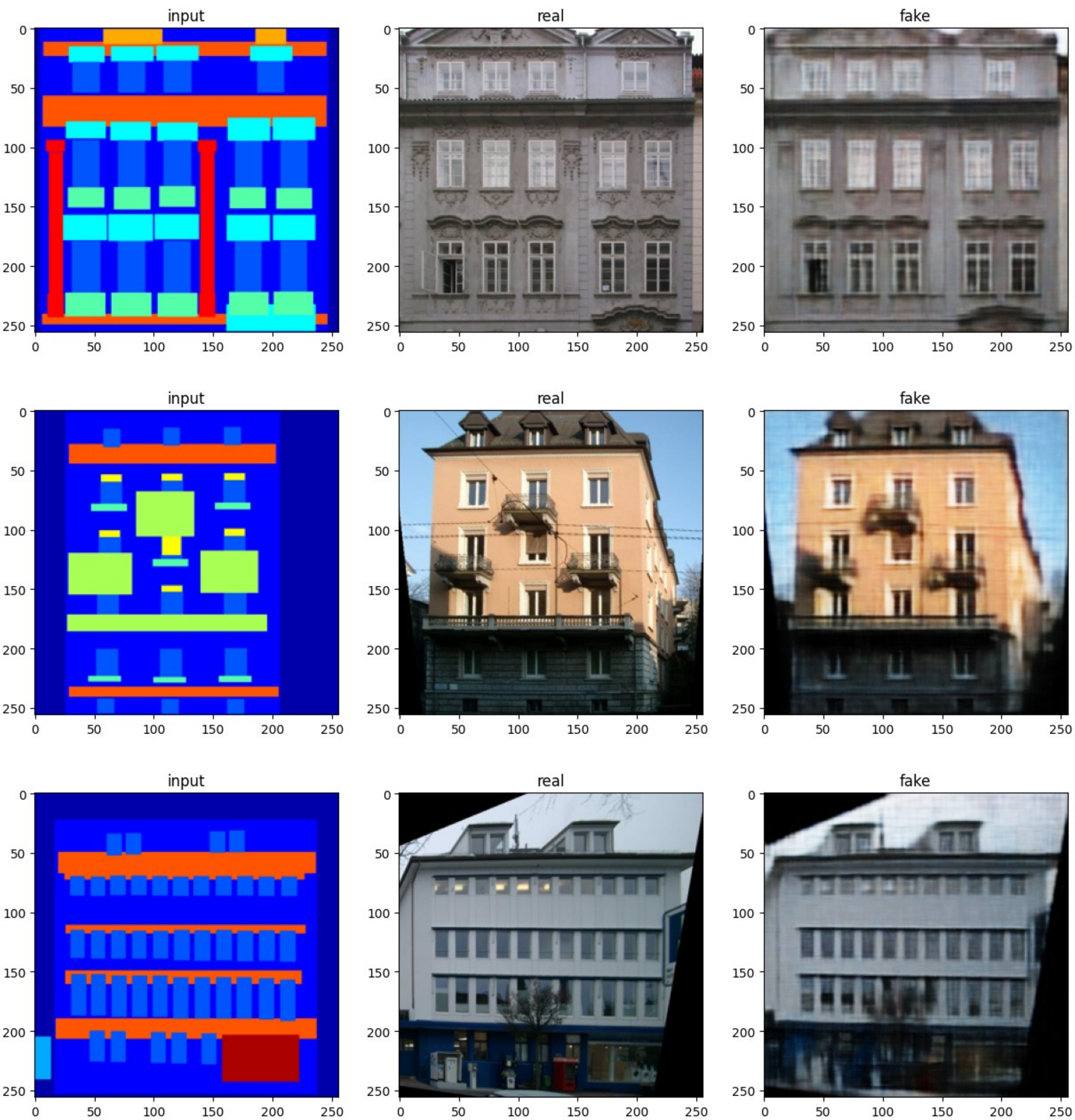
```
def load_model(epoch=200):
    if 'generator_'+str(epoch)+'.pth' in os.listdir() and
'discriminator_'+str(epoch)+'.pth' in os.listdir():
        checkpoint_generator = torch.load('generator_'+str(epoch)
+'.pth', map_location=device)

generator.load_state_dict(checkpoint_generator['model_state_dict'])

optimizer_G.load_state_dict(checkpoint_generator['optimizer_state_dict
'])
        epoch_G = checkpoint_generator['epoch']
        loss_G = checkpoint_generator['loss']

        checkpoint_discriminator =
torch.load('discriminator_'+str(epoch)+'.pth', map_location=device)

discriminator.load_state_dict(checkpoint_discriminator['model_state_di
ct'])

optimizer_D.load_state_dict(checkpoint_discriminator['optimizer_state_
dict'])
        epoch_D = checkpoint_discriminator['epoch']
        loss_D = checkpoint_discriminator['loss']

    else:
        print('There isn\' a training available with this number of
epochs')

load_model(epoch=200)

# switching mode
generator.eval()

U_Net(
  (inc): inconv(
    (conv): Sequential(
      (0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2),
padding=(1, 1))
      (1): LeakyReLU(negative_slope=0.2, inplace=True)
    )
  )
  (down1): down(
    (conv): Sequential(
      (0): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2),
padding=(1, 1))
      (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): LeakyReLU(negative_slope=0.2, inplace=True)
    )
```

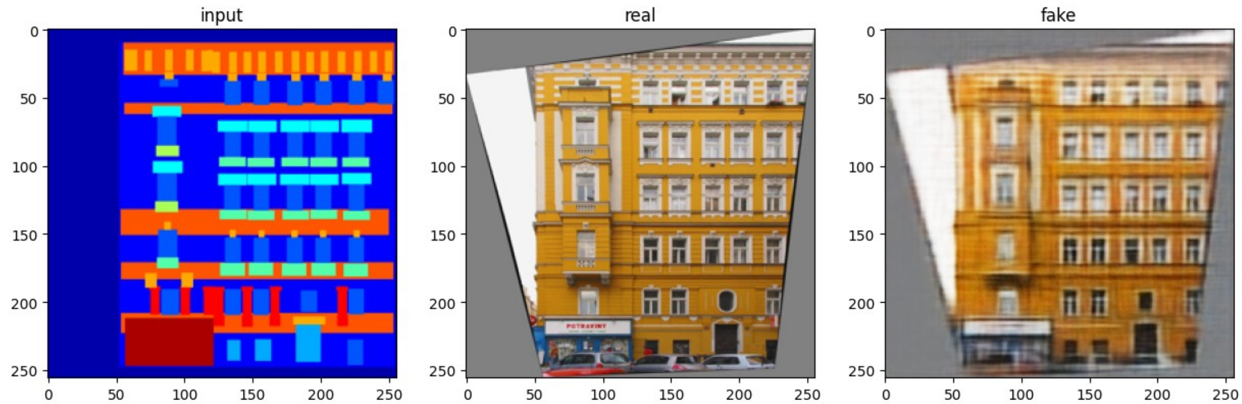
```
  )
  (down2): down(
    (conv): Sequential(
      (0): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2),
padding=(1, 1))
      (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): LeakyReLU(negative_slope=0.2, inplace=True)
    )
  )
  (down3): down(
    (conv): Sequential(
      (0): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2),
padding=(1, 1))
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): LeakyReLU(negative_slope=0.2, inplace=True)
    )
  )
  (down4): down(
    (conv): Sequential(
      (0): Conv2d(512, 512, kernel_size=(4, 4), stride=(2, 2),
padding=(1, 1))
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): LeakyReLU(negative_slope=0.2, inplace=True)
    )
  )
  (down5): down(
    (conv): Sequential(
      (0): Conv2d(512, 512, kernel_size=(4, 4), stride=(2, 2),
padding=(1, 1))
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): LeakyReLU(negative_slope=0.2, inplace=True)
    )
  )
  (down6): down(
    (conv): Sequential(
      (0): Conv2d(512, 512, kernel_size=(4, 4), stride=(2, 2),
padding=(1, 1))
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): LeakyReLU(negative_slope=0.2, inplace=True)
    )
  )
  (down7): down(
    (conv): Sequential(
      (0): Conv2d(512, 512, kernel_size=(4, 4), stride=(2, 2),
```

```
padding=(1, 1))
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): LeakyReLU(negative_slope=0.2, inplace=True)
    )
  )
  (up7): up(
    (conv): Sequential(
      (0): ConvTranspose2d(512, 512, kernel_size=(4, 4), stride=(2,
2), padding=(1, 1))
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): Dropout(p=0.5, inplace=True)
      (3): ReLU(inplace=True)
    )
  )
  (up6): up(
    (conv): Sequential(
      (0): ConvTranspose2d(1024, 512, kernel_size=(4, 4), stride=(2,
2), padding=(1, 1))
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): Dropout(p=0.5, inplace=True)
      (3): ReLU(inplace=True)
    )
  )
  (up5): up(
    (conv): Sequential(
      (0): ConvTranspose2d(1024, 512, kernel_size=(4, 4), stride=(2,
2), padding=(1, 1))
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): Dropout(p=0.5, inplace=True)
      (3): ReLU(inplace=True)
    )
  )
  (up4): up(
    (conv): Sequential(
      (0): ConvTranspose2d(1024, 512, kernel_size=(4, 4), stride=(2,
2), padding=(1, 1))
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): ReLU(inplace=True)
    )
  )
  (up3): up(
    (conv): Sequential(
      (0): ConvTranspose2d(1024, 256, kernel_size=(4, 4), stride=(2,
2), padding=(1, 1))
```

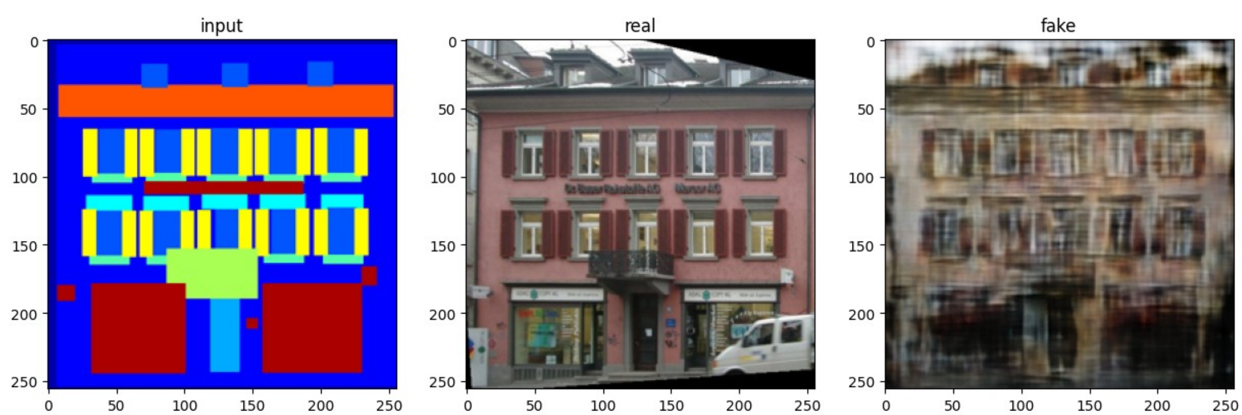
```
      (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): ReLU(inplace=True)
    )
  )
  (up2): up(
    (conv): Sequential(
      (0): ConvTranspose2d(512, 128, kernel_size=(4, 4), stride=(2,
2), padding=(1, 1))
      (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): ReLU(inplace=True)
    )
  )
  (up1): up(
    (conv): Sequential(
      (0): ConvTranspose2d(256, 64, kernel_size=(4, 4), stride=(2, 2),
padding=(1, 1))
      (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (2): ReLU(inplace=True)
    )
  )
  (outc): outconv(
    (conv): Sequential(
      (0): ConvTranspose2d(128, 3, kernel_size=(4, 4), stride=(2, 2),
padding=(1, 1))
      (1): Tanh()
    )
  )
)

# show a sample evaluation image on the training base
image, mask = next(iter(dataloader))
output = generator(mask.to(device))
output = output.cpu().detach()

for i in range(8):
    image_plot = reverse_transform(image[i])
    output_plot = reverse_transform(output[i])
    mask_plot = reverse_transform(mask[i])
    plot2x3Array(mask_plot,image_plot,output_plot)
```

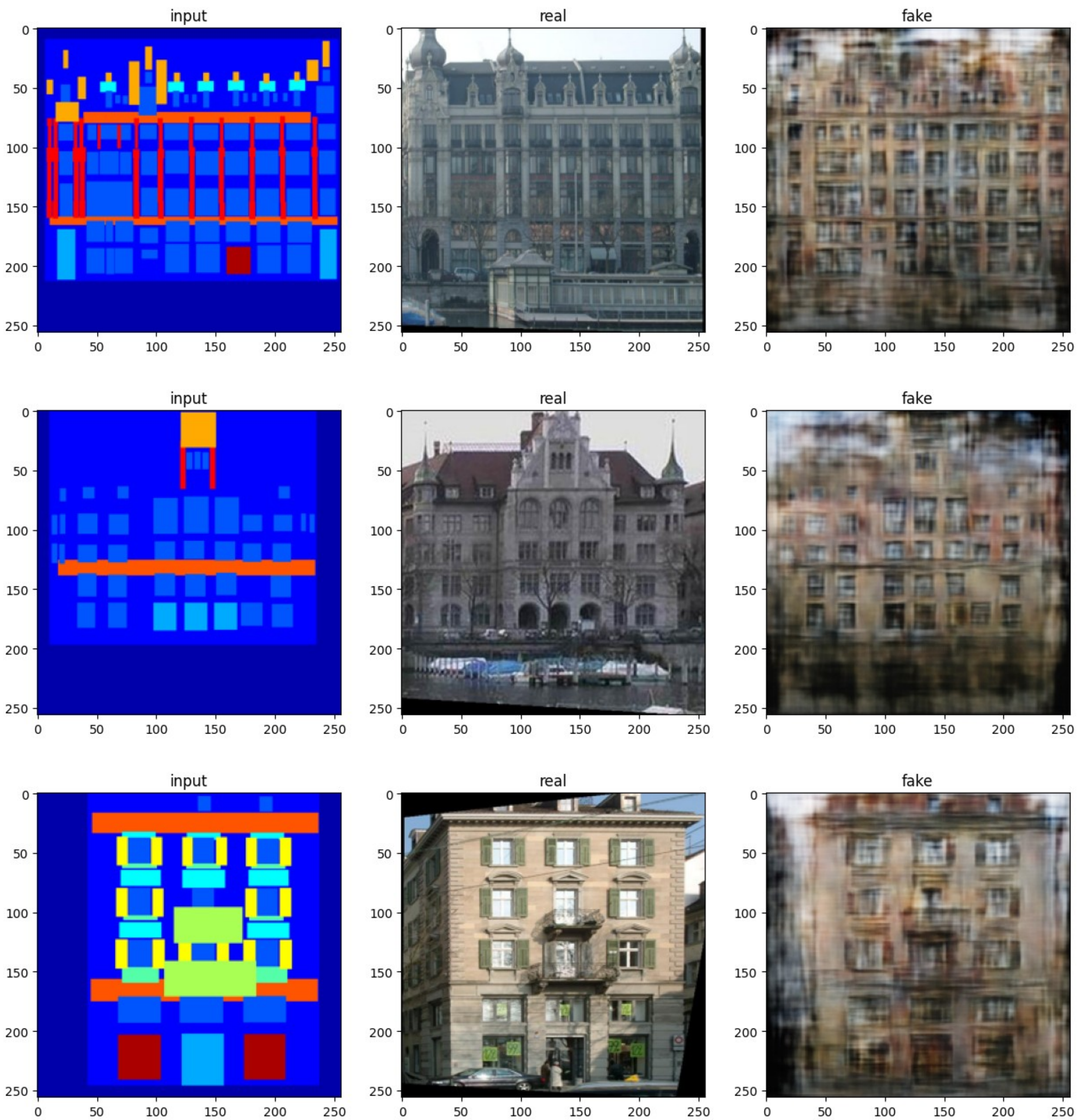

input
real
fake
input
real
fake
input
real
fake


input
real
fake
input
real
fake
input
real
fake
0
50
100
150
200
250

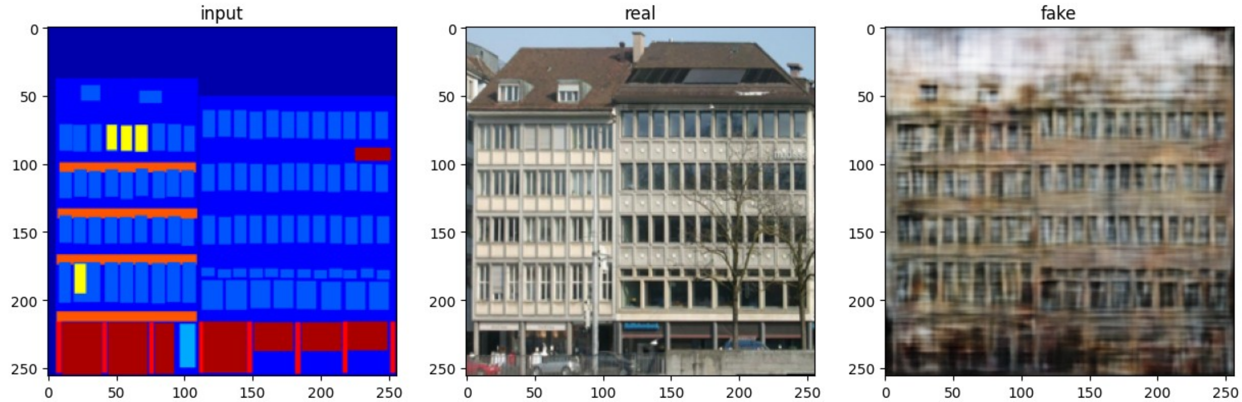



```python
# show a sample evaluation image on the validation base
image, mask = next(iter(val_dataloader))
output = generator(mask.to(device))
output = output.cpu().detach()

for i in range(8):
    image_plot = reverse_transform(image[i])
    output_plot = reverse_transform(output[i])
    mask_plot = reverse_transform(mask[i])
    plot2x3Array(mask_plot,image_plot,output_plot)
```




input
real
fake
input
real
fake
input
real
fake
0
50
100
150
200
250


input
real
fake
input
real
fake
input
real
fake
0
50
100
150
200
250



Part 3: Diffusion

Diffusion models are probabilistic generative models which learn to generate data by iteratively refining random noise through a reverse diffusion process. Given a sample of data, noise is progressvely added in small steps until it becomes pure noise. Then, a neural network is trained to reverse this process and generate realistic data from noise.

Diffusion models have gained popularity due to their ability to generate high-quality, diverse, and detailed content, surpassing GANs in the quality of the generated images.

Here we will focus on DDPMs, which were introduced in this paper and laid the foundation for generative diffusion models.

For this part, we will use the MNIST dataset, used in part 1

```
dataroot = "data/mnist"
image_size = 64
batch_size = 128
workers = 2
torch.set_deterministic_debug_mode(False)



mnist_dataset = dset.MNIST(root=dataroot, download=True,
transform=transforms.Compose([
    transforms.Resize(image_size),
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
]))

mnist_dataloader = torch.utils.data.DataLoader(mnist_dataset,
batch_size=batch_size,
                                         shuffle=True,
num_workers=workers)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-1-6588ae65246c> in <cell line: 0>()
      4 batch_size = 128
      5 workers = 2
----> 6 torch.set_deterministic_debug_mode(False)
      7
      8

NameError: name 'torch' is not defined
```

Auxiliary function for plotting images

```python
def plot1xNArray(images, labels):
    f, axarr = plt.subplots(1, len(images))

    for image, ax, label in zip(images, axarr, labels):
        ax.imshow(image, cmap='gray')
        ax.axis('off')
        ax.set_title(label)
```

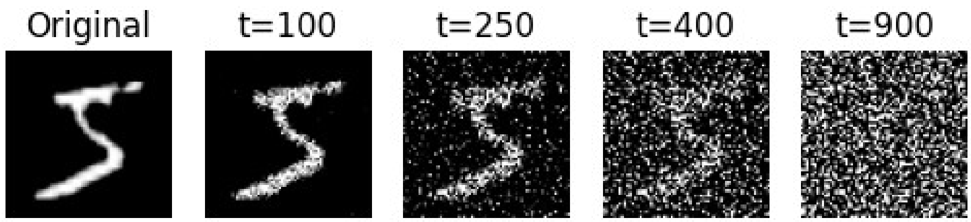
In order to train the model with the diffusion process, we will use a noise scheduler, which will be in charge of the forward diffusion process. The scheduler takes an image, a sample of random noise and a timestep, and return a noisy image for the corresponding timestep. Noise is progressvely added to the image at each timestep, therefore a noisy image at timestep 0 will have barely any noise while a noisy image at the maximum timestep will be basically just noise.

Let's create a noise scheduler with 1000 max timesteps and visualize some noise images.

We will use the diffusers library, which provides several tools for training and using diffusion models.

```python
from diffusers import DDPMScheduler

noise_scheduler = DDPMScheduler(num_train_timesteps = 1000)

image, _ = mnist_dataset[0]

noise = torch.randn_like(image)

images, labels = [reverse_transform(image)], ["Original"]

for i in [100, 250, 400, 900]:
    timestep = torch.LongTensor([i])
    noisy_image = noise_scheduler.add_noise(image, noise, timestep)
    images.append(reverse_transform(noisy_image))
    labels.append(f"t={i}")
```

```
plot1xNArray(images, labels)
```



For the reverse diffusion process we will use a neural network. Given a noisy image and the corresponding timestep, the goal of the neural network is to predict the noise, which allows for the denoising.

For the model, we will have a similar architecture as we used for the cGAN generator, a 2D UNet with a few modifications. The main difference will be that we have to indicate to the model which timestep is currently being denoised. For that purpose a timestep embedding is added, therefore the model has 2 inputs, the noisy image and the corresponding timestep.

We will use an UNet implementation from the diffusers library, which already has the timestep embedding included.

```
from diffusers import UNet2DModel

diffusion_backbone = UNet2DModel(
                        block_out_channels=(64, 128, 256, 512),
                        down_block_types=("DownBlock2D",
"DownBlock2D", "DownBlock2D", "DownBlock2D"),
                        up_block_types=("UpBlock2D", "UpBlock2D",
"UpBlock2D", "UpBlock2D"),
                        sample_size=64,
                        in_channels=1,
                        out_channels=1,
                    ).to(device)

# Optimizer
optimizer = torch.optim.AdamW(diffusion_backbone.parameters(), lr=1e-
4)

print(diffusion_backbone)
```

```
UNet2DModel(
  (conv_in): Conv2d(1, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
  (time_proj): Timesteps()
  (time_embedding): TimestepEmbedding(
    (linear_1): Linear(in_features=64, out_features=256, bias=True)
    (act): SiLU()
    (linear_2): Linear(in_features=256, out_features=256, bias=True)
```

```
  )
  (down_blocks): ModuleList(
    (0): DownBlock2D(
      (resnets): ModuleList(
        (0-1): 2 x ResnetBlock2D(
          (norm1): GroupNorm(32, 64, eps=1e-05, affine=True)
          (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (time_emb_proj): Linear(in_features=256, out_features=64,
bias=True)
          (norm2): GroupNorm(32, 64, eps=1e-05, affine=True)
          (dropout): Dropout(p=0.0, inplace=False)
          (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (nonlinearity): SiLU()
        )
      )
      (downsamplers): ModuleList(
        (0): Downsample2D(
          (conv): Conv2d(64, 64, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1))
        )
      )
    )
    (1): DownBlock2D(
      (resnets): ModuleList(
        (0): ResnetBlock2D(
          (norm1): GroupNorm(32, 64, eps=1e-05, affine=True)
          (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (time_emb_proj): Linear(in_features=256, out_features=128,
bias=True)
          (norm2): GroupNorm(32, 128, eps=1e-05, affine=True)
          (dropout): Dropout(p=0.0, inplace=False)
          (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (nonlinearity): SiLU()
          (conv_shortcut): Conv2d(64, 128, kernel_size=(1, 1),
stride=(1, 1))
        )
        (1): ResnetBlock2D(
          (norm1): GroupNorm(32, 128, eps=1e-05, affine=True)
          (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (time_emb_proj): Linear(in_features=256, out_features=128,
bias=True)
          (norm2): GroupNorm(32, 128, eps=1e-05, affine=True)
          (dropout): Dropout(p=0.0, inplace=False)
          (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
```

```
padding=(1, 1))
          (nonlinearity): SiLU()
        )
      )
      (downsamplers): ModuleList(
        (0): Downsample2D(
          (conv): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1))
        )
      )
    )
    (2): DownBlock2D(
      (resnets): ModuleList(
        (0): ResnetBlock2D(
          (norm1): GroupNorm(32, 128, eps=1e-05, affine=True)
          (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (time_emb_proj): Linear(in_features=256, out_features=256,
bias=True)
          (norm2): GroupNorm(32, 256, eps=1e-05, affine=True)
          (dropout): Dropout(p=0.0, inplace=False)
          (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (nonlinearity): SiLU()
          (conv_shortcut): Conv2d(128, 256, kernel_size=(1, 1),
stride=(1, 1))
        )
        (1): ResnetBlock2D(
          (norm1): GroupNorm(32, 256, eps=1e-05, affine=True)
          (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (time_emb_proj): Linear(in_features=256, out_features=256,
bias=True)
          (norm2): GroupNorm(32, 256, eps=1e-05, affine=True)
          (dropout): Dropout(p=0.0, inplace=False)
          (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (nonlinearity): SiLU()
        )
      )
      (downsamplers): ModuleList(
        (0): Downsample2D(
          (conv): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1))
        )
      )
    )
    (3): DownBlock2D(
      (resnets): ModuleList(
```

```
        (0): ResnetBlock2D(
          (norm1): GroupNorm(32, 256, eps=1e-05, affine=True)
          (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (time_emb_proj): Linear(in_features=256, out_features=512,
bias=True)
          (norm2): GroupNorm(32, 512, eps=1e-05, affine=True)
          (dropout): Dropout(p=0.0, inplace=False)
          (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (nonlinearity): SiLU()
          (conv_shortcut): Conv2d(256, 512, kernel_size=(1, 1),
stride=(1, 1))
        )
        (1): ResnetBlock2D(
          (norm1): GroupNorm(32, 512, eps=1e-05, affine=True)
          (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (time_emb_proj): Linear(in_features=256, out_features=512,
bias=True)
          (norm2): GroupNorm(32, 512, eps=1e-05, affine=True)
          (dropout): Dropout(p=0.0, inplace=False)
          (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (nonlinearity): SiLU()
        )
      )
    )
  )
  (up_blocks): ModuleList(
    (0): UpBlock2D(
      (resnets): ModuleList(
        (0-1): 2 x ResnetBlock2D(
          (norm1): GroupNorm(32, 1024, eps=1e-05, affine=True)
          (conv1): Conv2d(1024, 512, kernel_size=(3, 3), stride=(1,
1), padding=(1, 1))
          (time_emb_proj): Linear(in_features=256, out_features=512,
bias=True)
          (norm2): GroupNorm(32, 512, eps=1e-05, affine=True)
          (dropout): Dropout(p=0.0, inplace=False)
          (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (nonlinearity): SiLU()
          (conv_shortcut): Conv2d(1024, 512, kernel_size=(1, 1),
stride=(1, 1))
        )
        (2): ResnetBlock2D(
          (norm1): GroupNorm(32, 768, eps=1e-05, affine=True)
          (conv1): Conv2d(768, 512, kernel_size=(3, 3), stride=(1, 1),
```

```
padding=(1, 1))
          (time_emb_proj): Linear(in_features=256, out_features=512,
bias=True)
          (norm2): GroupNorm(32, 512, eps=1e-05, affine=True)
          (dropout): Dropout(p=0.0, inplace=False)
          (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (nonlinearity): SiLU()
          (conv_shortcut): Conv2d(768, 512, kernel_size=(1, 1),
stride=(1, 1))
        )
      )
      (upsamplers): ModuleList(
        (0): Upsample2D(
          (conv): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
        )
      )
    )
    (1): UpBlock2D(
      (resnets): ModuleList(
        (0): ResnetBlock2D(
          (norm1): GroupNorm(32, 768, eps=1e-05, affine=True)
          (conv1): Conv2d(768, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (time_emb_proj): Linear(in_features=256, out_features=256,
bias=True)
          (norm2): GroupNorm(32, 256, eps=1e-05, affine=True)
          (dropout): Dropout(p=0.0, inplace=False)
          (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (nonlinearity): SiLU()
          (conv_shortcut): Conv2d(768, 256, kernel_size=(1, 1),
stride=(1, 1))
        )
        (1): ResnetBlock2D(
          (norm1): GroupNorm(32, 512, eps=1e-05, affine=True)
          (conv1): Conv2d(512, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (time_emb_proj): Linear(in_features=256, out_features=256,
bias=True)
          (norm2): GroupNorm(32, 256, eps=1e-05, affine=True)
          (dropout): Dropout(p=0.0, inplace=False)
          (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (nonlinearity): SiLU()
          (conv_shortcut): Conv2d(512, 256, kernel_size=(1, 1),
stride=(1, 1))
        )
```

```
        (2): ResnetBlock2D(
          (norm1): GroupNorm(32, 384, eps=1e-05, affine=True)
          (conv1): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (time_emb_proj): Linear(in_features=256, out_features=256,
bias=True)
          (norm2): GroupNorm(32, 256, eps=1e-05, affine=True)
          (dropout): Dropout(p=0.0, inplace=False)
          (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (nonlinearity): SiLU()
          (conv_shortcut): Conv2d(384, 256, kernel_size=(1, 1),
stride=(1, 1))
        )
      )
      (upsamplers): ModuleList(
        (0): Upsample2D(
          (conv): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
        )
      )
    )
    (2): UpBlock2D(
      (resnets): ModuleList(
        (0): ResnetBlock2D(
          (norm1): GroupNorm(32, 384, eps=1e-05, affine=True)
          (conv1): Conv2d(384, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (time_emb_proj): Linear(in_features=256, out_features=128,
bias=True)
          (norm2): GroupNorm(32, 128, eps=1e-05, affine=True)
          (dropout): Dropout(p=0.0, inplace=False)
          (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (nonlinearity): SiLU()
          (conv_shortcut): Conv2d(384, 128, kernel_size=(1, 1),
stride=(1, 1))
        )
        (1): ResnetBlock2D(
          (norm1): GroupNorm(32, 256, eps=1e-05, affine=True)
          (conv1): Conv2d(256, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (time_emb_proj): Linear(in_features=256, out_features=128,
bias=True)
          (norm2): GroupNorm(32, 128, eps=1e-05, affine=True)
          (dropout): Dropout(p=0.0, inplace=False)
          (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (nonlinearity): SiLU()
```

```
          (conv_shortcut): Conv2d(256, 128, kernel_size=(1, 1),
stride=(1, 1))
        )
        (2): ResnetBlock2D(
          (norm1): GroupNorm(32, 192, eps=1e-05, affine=True)
          (conv1): Conv2d(192, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (time_emb_proj): Linear(in_features=256, out_features=128,
bias=True)
          (norm2): GroupNorm(32, 128, eps=1e-05, affine=True)
          (dropout): Dropout(p=0.0, inplace=False)
          (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (nonlinearity): SiLU()
          (conv_shortcut): Conv2d(192, 128, kernel_size=(1, 1),
stride=(1, 1))
        )
      )
      (upsamplers): ModuleList(
        (0): Upsample2D(
          (conv): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
        )
      )
    )
    (3): UpBlock2D(
      (resnets): ModuleList(
        (0): ResnetBlock2D(
          (norm1): GroupNorm(32, 192, eps=1e-05, affine=True)
          (conv1): Conv2d(192, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (time_emb_proj): Linear(in_features=256, out_features=64,
bias=True)
          (norm2): GroupNorm(32, 64, eps=1e-05, affine=True)
          (dropout): Dropout(p=0.0, inplace=False)
          (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (nonlinearity): SiLU()
          (conv_shortcut): Conv2d(192, 64, kernel_size=(1, 1),
stride=(1, 1))
        )
        (1-2): 2 x ResnetBlock2D(
          (norm1): GroupNorm(32, 128, eps=1e-05, affine=True)
          (conv1): Conv2d(128, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (time_emb_proj): Linear(in_features=256, out_features=64,
bias=True)
          (norm2): GroupNorm(32, 64, eps=1e-05, affine=True)
          (dropout): Dropout(p=0.0, inplace=False)
```

```
          (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (nonlinearity): SiLU()
          (conv_shortcut): Conv2d(128, 64, kernel_size=(1, 1),
stride=(1, 1))
        )
      )
    )
  )
  (mid_block): UNetMidBlock2D(
    (attentions): ModuleList(
      (0): Attention(
        (group_norm): GroupNorm(32, 512, eps=1e-05, affine=True)
        (to_q): Linear(in_features=512, out_features=512, bias=True)
        (to_k): Linear(in_features=512, out_features=512, bias=True)
        (to_v): Linear(in_features=512, out_features=512, bias=True)
        (to_out): ModuleList(
          (0): Linear(in_features=512, out_features=512, bias=True)
          (1): Dropout(p=0.0, inplace=False)
        )
      )
    )
    (resnets): ModuleList(
      (0-1): 2 x ResnetBlock2D(
        (norm1): GroupNorm(32, 512, eps=1e-05, affine=True)
        (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
        (time_emb_proj): Linear(in_features=256, out_features=512,
bias=True)
        (norm2): GroupNorm(32, 512, eps=1e-05, affine=True)
        (dropout): Dropout(p=0.0, inplace=False)
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
        (nonlinearity): SiLU()
      )
    )
  )
  (conv_norm_out): GroupNorm(32, 64, eps=1e-05, affine=True)
  (conv_act): SiLU()
  (conv_out): Conv2d(64, 1, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
)
```

Differences between CGAN UNet and Diffusion UNet

While cGAN Unet uses Conv2D layers with a kernel size of 4×4 and stride of 2, which aggressively reduces spatial dimensions and uses ConvTranspose2D for upsampling; the Diffusion UNet uses Conv2D layers with a 3×3 kernel and stride of 1, preserving spatial information more effectively and uses ResNet blocks inside downsampling and upsampling layers.

Now, let's train the model

```
# ----------------
#  Training Loop
# ----------------
torch.backends.cudnn.deterministic = False
torch.use_deterministic_algorithms = False


losses = []
num_epochs = 5
print_every = 100

diffusion_backbone.train()

for epoch in range(num_epochs):
    for i, batch in enumerate(mnist_dataloader):
        # Zero the gradients
        optimizer.zero_grad()

        # Send input to device
        images = batch[0].to(device)

        # Generate noisy images, different timestep for each image in
the batch
        timesteps =
torch.randint(noise_scheduler.config.num_train_timesteps,
(images.size(0),), device=device)

        noise = torch.randn_like(images).to(device)
        noisy_images = noise_scheduler.add_noise(images, noise,
timesteps)

        # Forward pass
        residual = diffusion_backbone(noisy_images, timesteps).sample

        loss = nn.functional.mse_loss(residual, noise)

        loss.backward()
        optimizer.step()

        # Print stats
        if i % print_every == 0:
            print(f'Epoch [{epoch+1}/{num_epochs}]
[{i}/{len(mnist_dataloader)}] | loss: {loss.item():6.4f}')

    losses.append(loss.item())
    torch.save(diffusion_backbone.state_dict(),
f"diffusion_{epoch+1}.pth")
```

```
Epoch [1/5][0/469] | loss: 1.1617
Epoch [1/5][100/469] | loss: 0.0300
Epoch [1/5][200/469] | loss: 0.0166
Epoch [1/5][300/469] | loss: 0.0147
Epoch [1/5][400/469] | loss: 0.0146
Epoch [2/5][0/469] | loss: 0.0132
Epoch [2/5][100/469] | loss: 0.0130
Epoch [2/5][200/469] | loss: 0.0128
Epoch [2/5][300/469] | loss: 0.0100
Epoch [2/5][400/469] | loss: 0.0094
Epoch [3/5][0/469] | loss: 0.0106
Epoch [3/5][100/469] | loss: 0.0084
Epoch [3/5][200/469] | loss: 0.0098
Epoch [3/5][300/469] | loss: 0.0074
Epoch [3/5][400/469] | loss: 0.0100
Epoch [4/5][0/469] | loss: 0.0069
Epoch [4/5][100/469] | loss: 0.0082
Epoch [4/5][200/469] | loss: 0.0065
Epoch [4/5][300/469] | loss: 0.0074
Epoch [4/5][400/469] | loss: 0.0064
Epoch [5/5][0/469] | loss: 0.0067
Epoch [5/5][100/469] | loss: 0.0069
Epoch [5/5][200/469] | loss: 0.0086
Epoch [5/5][300/469] | loss: 0.0074
Epoch [5/5][400/469] | loss: 0.0067
```

```python
diffusion_backbone.load_state_dict(torch.load("diffusion_5.pth"))
diffusion_backbone.eval()
```

```
UNet2DModel(
  (conv_in): Conv2d(1, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
  (time_proj): Timesteps()
  (time_embedding): TimestepEmbedding(
    (linear_1): Linear(in_features=64, out_features=256, bias=True)
    (act): SiLU()
    (linear_2): Linear(in_features=256, out_features=256, bias=True)
  )
  (down_blocks): ModuleList(
    (0): DownBlock2D(
      (resnets): ModuleList(
        (0-1): 2 x ResnetBlock2D(
          (norm1): GroupNorm(32, 64, eps=1e-05, affine=True)
          (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (time_emb_proj): Linear(in_features=256, out_features=64,
bias=True)
          (norm2): GroupNorm(32, 64, eps=1e-05, affine=True)
          (dropout): Dropout(p=0.0, inplace=False)
          (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
```

```
padding=(1, 1))
          (nonlinearity): SiLU()
        )
      )
      (downsamplers): ModuleList(
        (0): Downsample2D(
          (conv): Conv2d(64, 64, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1))
        )
      )
    )
    (1): DownBlock2D(
      (resnets): ModuleList(
        (0): ResnetBlock2D(
          (norm1): GroupNorm(32, 64, eps=1e-05, affine=True)
          (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (time_emb_proj): Linear(in_features=256, out_features=128,
bias=True)
          (norm2): GroupNorm(32, 128, eps=1e-05, affine=True)
          (dropout): Dropout(p=0.0, inplace=False)
          (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (nonlinearity): SiLU()
          (conv_shortcut): Conv2d(64, 128, kernel_size=(1, 1),
stride=(1, 1))
        )
        (1): ResnetBlock2D(
          (norm1): GroupNorm(32, 128, eps=1e-05, affine=True)
          (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (time_emb_proj): Linear(in_features=256, out_features=128,
bias=True)
          (norm2): GroupNorm(32, 128, eps=1e-05, affine=True)
          (dropout): Dropout(p=0.0, inplace=False)
          (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (nonlinearity): SiLU()
        )
      )
      (downsamplers): ModuleList(
        (0): Downsample2D(
          (conv): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1))
        )
      )
    )
    (2): DownBlock2D(
      (resnets): ModuleList(
```

```
        (0): ResnetBlock2D(
          (norm1): GroupNorm(32, 128, eps=1e-05, affine=True)
          (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (time_emb_proj): Linear(in_features=256, out_features=256,
bias=True)
          (norm2): GroupNorm(32, 256, eps=1e-05, affine=True)
          (dropout): Dropout(p=0.0, inplace=False)
          (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (nonlinearity): SiLU()
          (conv_shortcut): Conv2d(128, 256, kernel_size=(1, 1),
stride=(1, 1))
        )
        (1): ResnetBlock2D(
          (norm1): GroupNorm(32, 256, eps=1e-05, affine=True)
          (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (time_emb_proj): Linear(in_features=256, out_features=256,
bias=True)
          (norm2): GroupNorm(32, 256, eps=1e-05, affine=True)
          (dropout): Dropout(p=0.0, inplace=False)
          (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (nonlinearity): SiLU()
        )
      )
      (downsamplers): ModuleList(
        (0): Downsample2D(
          (conv): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1))
        )
      )
    )
    (3): DownBlock2D(
      (resnets): ModuleList(
        (0): ResnetBlock2D(
          (norm1): GroupNorm(32, 256, eps=1e-05, affine=True)
          (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (time_emb_proj): Linear(in_features=256, out_features=512,
bias=True)
          (norm2): GroupNorm(32, 512, eps=1e-05, affine=True)
          (dropout): Dropout(p=0.0, inplace=False)
          (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (nonlinearity): SiLU()
          (conv_shortcut): Conv2d(256, 512, kernel_size=(1, 1),
stride=(1, 1))
```

```
        )
        (1): ResnetBlock2D(
          (norm1): GroupNorm(32, 512, eps=1e-05, affine=True)
          (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (time_emb_proj): Linear(in_features=256, out_features=512,
bias=True)
          (norm2): GroupNorm(32, 512, eps=1e-05, affine=True)
          (dropout): Dropout(p=0.0, inplace=False)
          (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (nonlinearity): SiLU()
        )
      )
    )
  )
  (up_blocks): ModuleList(
    (0): UpBlock2D(
      (resnets): ModuleList(
        (0-1): 2 x ResnetBlock2D(
          (norm1): GroupNorm(32, 1024, eps=1e-05, affine=True)
          (conv1): Conv2d(1024, 512, kernel_size=(3, 3), stride=(1,
1), padding=(1, 1))
          (time_emb_proj): Linear(in_features=256, out_features=512,
bias=True)
          (norm2): GroupNorm(32, 512, eps=1e-05, affine=True)
          (dropout): Dropout(p=0.0, inplace=False)
          (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (nonlinearity): SiLU()
          (conv_shortcut): Conv2d(1024, 512, kernel_size=(1, 1),
stride=(1, 1))
        )
        (2): ResnetBlock2D(
          (norm1): GroupNorm(32, 768, eps=1e-05, affine=True)
          (conv1): Conv2d(768, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (time_emb_proj): Linear(in_features=256, out_features=512,
bias=True)
          (norm2): GroupNorm(32, 512, eps=1e-05, affine=True)
          (dropout): Dropout(p=0.0, inplace=False)
          (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (nonlinearity): SiLU()
          (conv_shortcut): Conv2d(768, 512, kernel_size=(1, 1),
stride=(1, 1))
        )
      )
      (upsamplers): ModuleList(
```

```
        (0): Upsample2D(
          (conv): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
        )
      )
    )
    (1): UpBlock2D(
      (resnets): ModuleList(
        (0): ResnetBlock2D(
          (norm1): GroupNorm(32, 768, eps=1e-05, affine=True)
          (conv1): Conv2d(768, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (time_emb_proj): Linear(in_features=256, out_features=256,
bias=True)
          (norm2): GroupNorm(32, 256, eps=1e-05, affine=True)
          (dropout): Dropout(p=0.0, inplace=False)
          (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (nonlinearity): SiLU()
          (conv_shortcut): Conv2d(768, 256, kernel_size=(1, 1),
stride=(1, 1))
        )
        (1): ResnetBlock2D(
          (norm1): GroupNorm(32, 512, eps=1e-05, affine=True)
          (conv1): Conv2d(512, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (time_emb_proj): Linear(in_features=256, out_features=256,
bias=True)
          (norm2): GroupNorm(32, 256, eps=1e-05, affine=True)
          (dropout): Dropout(p=0.0, inplace=False)
          (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (nonlinearity): SiLU()
          (conv_shortcut): Conv2d(512, 256, kernel_size=(1, 1),
stride=(1, 1))
        )
        (2): ResnetBlock2D(
          (norm1): GroupNorm(32, 384, eps=1e-05, affine=True)
          (conv1): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (time_emb_proj): Linear(in_features=256, out_features=256,
bias=True)
          (norm2): GroupNorm(32, 256, eps=1e-05, affine=True)
          (dropout): Dropout(p=0.0, inplace=False)
          (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (nonlinearity): SiLU()
          (conv_shortcut): Conv2d(384, 256, kernel_size=(1, 1),
stride=(1, 1))
```

```
        )
      )
      (upsamplers): ModuleList(
        (0): Upsample2D(
          (conv): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
        )
      )
    )
    (2): UpBlock2D(
      (resnets): ModuleList(
        (0): ResnetBlock2D(
          (norm1): GroupNorm(32, 384, eps=1e-05, affine=True)
          (conv1): Conv2d(384, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (time_emb_proj): Linear(in_features=256, out_features=128,
bias=True)
          (norm2): GroupNorm(32, 128, eps=1e-05, affine=True)
          (dropout): Dropout(p=0.0, inplace=False)
          (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (nonlinearity): SiLU()
          (conv_shortcut): Conv2d(384, 128, kernel_size=(1, 1),
stride=(1, 1))
        )
        (1): ResnetBlock2D(
          (norm1): GroupNorm(32, 256, eps=1e-05, affine=True)
          (conv1): Conv2d(256, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (time_emb_proj): Linear(in_features=256, out_features=128,
bias=True)
          (norm2): GroupNorm(32, 128, eps=1e-05, affine=True)
          (dropout): Dropout(p=0.0, inplace=False)
          (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (nonlinearity): SiLU()
          (conv_shortcut): Conv2d(256, 128, kernel_size=(1, 1),
stride=(1, 1))
        )
        (2): ResnetBlock2D(
          (norm1): GroupNorm(32, 192, eps=1e-05, affine=True)
          (conv1): Conv2d(192, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (time_emb_proj): Linear(in_features=256, out_features=128,
bias=True)
          (norm2): GroupNorm(32, 128, eps=1e-05, affine=True)
          (dropout): Dropout(p=0.0, inplace=False)
          (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
```

```
          (nonlinearity): SiLU()
          (conv_shortcut): Conv2d(192, 128, kernel_size=(1, 1),
stride=(1, 1))
        )
      )
      (upsamplers): ModuleList(
        (0): Upsample2D(
          (conv): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
        )
      )
    )
    (3): UpBlock2D(
      (resnets): ModuleList(
        (0): ResnetBlock2D(
          (norm1): GroupNorm(32, 192, eps=1e-05, affine=True)
          (conv1): Conv2d(192, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (time_emb_proj): Linear(in_features=256, out_features=64,
bias=True)
          (norm2): GroupNorm(32, 64, eps=1e-05, affine=True)
          (dropout): Dropout(p=0.0, inplace=False)
          (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (nonlinearity): SiLU()
          (conv_shortcut): Conv2d(192, 64, kernel_size=(1, 1),
stride=(1, 1))
        )
        (1-2): 2 x ResnetBlock2D(
          (norm1): GroupNorm(32, 128, eps=1e-05, affine=True)
          (conv1): Conv2d(128, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (time_emb_proj): Linear(in_features=256, out_features=64,
bias=True)
          (norm2): GroupNorm(32, 64, eps=1e-05, affine=True)
          (dropout): Dropout(p=0.0, inplace=False)
          (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
          (nonlinearity): SiLU()
          (conv_shortcut): Conv2d(128, 64, kernel_size=(1, 1),
stride=(1, 1))
        )
      )
    )
  )
  (mid_block): UNetMidBlock2D(
    (attentions): ModuleList(
      (0): Attention(
        (group_norm): GroupNorm(32, 512, eps=1e-05, affine=True)
```

```
        (to_q): Linear(in_features=512, out_features=512, bias=True)
        (to_k): Linear(in_features=512, out_features=512, bias=True)
        (to_v): Linear(in_features=512, out_features=512, bias=True)
        (to_out): ModuleList(
          (0): Linear(in_features=512, out_features=512, bias=True)
          (1): Dropout(p=0.0, inplace=False)
        )
      )
    )
    (resnets): ModuleList(
      (0-1): 2 x ResnetBlock2D(
        (norm1): GroupNorm(32, 512, eps=1e-05, affine=True)
        (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
        (time_emb_proj): Linear(in_features=256, out_features=512,
bias=True)
        (norm2): GroupNorm(32, 512, eps=1e-05, affine=True)
        (dropout): Dropout(p=0.0, inplace=False)
        (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
        (nonlinearity): SiLU()
      )
    )
  )
  (conv_norm_out): GroupNorm(32, 64, eps=1e-05, affine=True)
  (conv_act): SiLU()
  (conv_out): Conv2d(64, 1, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
)
```

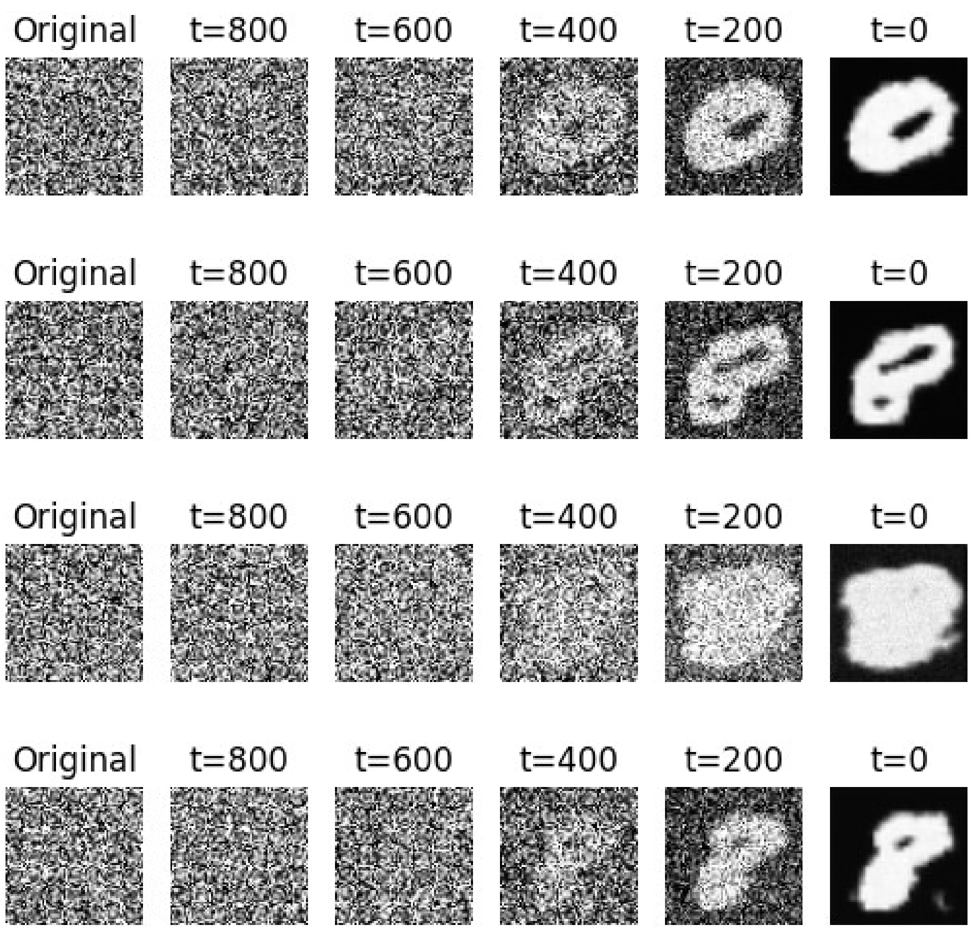
Time to generate some images.

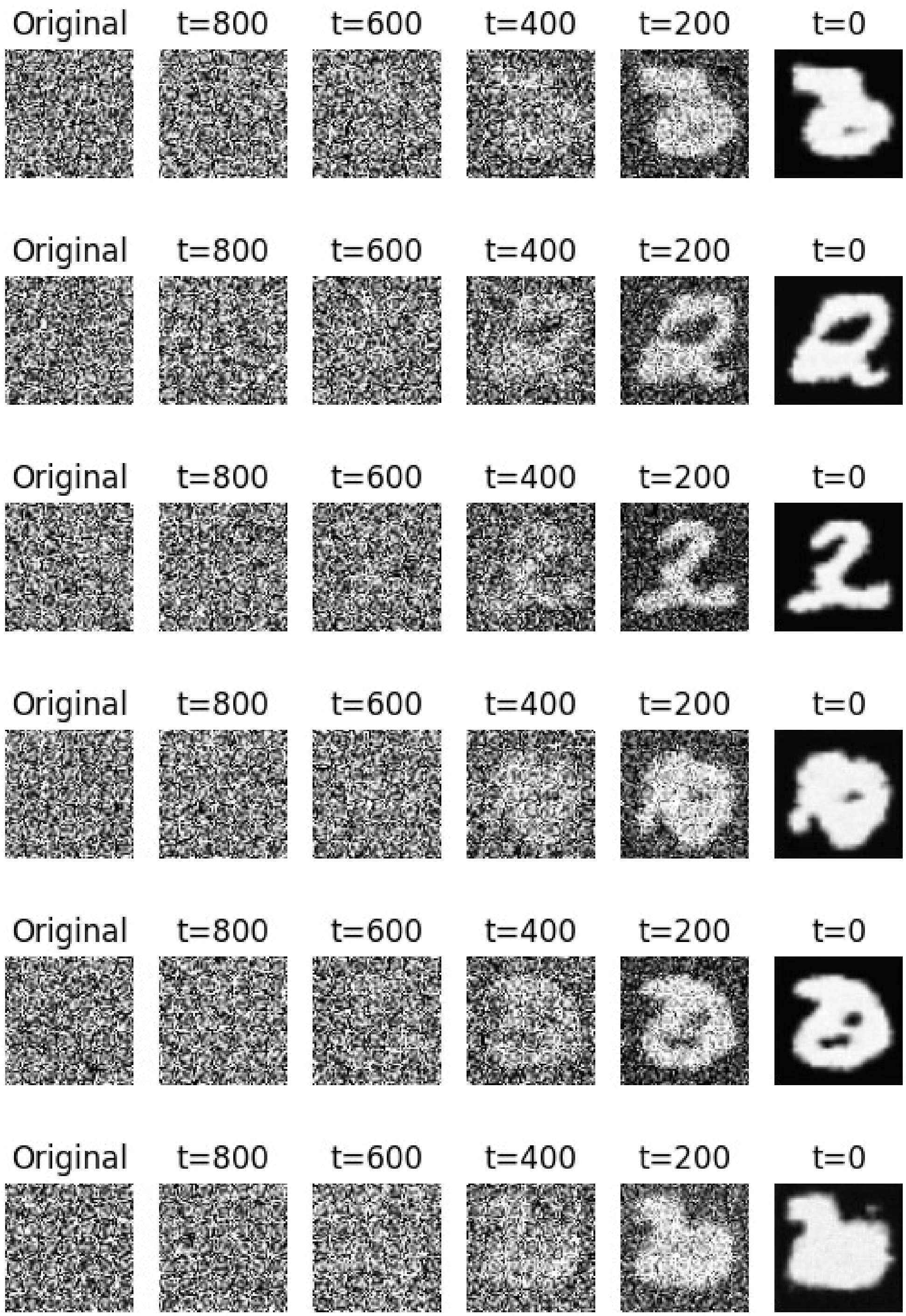
During training, for each data sample, we take a random timestep and correspondent noisy image to give it as input to our model. With suffitient training, the model should learn how to predict the noise in a noisy image for all possible timesteps.

During inference, to generate an image, we will start from pure noise and step by step predict the noise to go from one noisy image to the next, progressively denoising the image until we reach the timestep 0, in which we should have an image without any noise.

```python
from tqdm import tqdm

# Start the image as random noise
image = torch.randn((10, 1, 64, 64)).to(device)

# Create a list of images and labels for visualization
images, labels = [(image / 2 + 0.5).clamp(0, 1).cpu().permute(0, 2, 3,
1).numpy()], ["Original"]

# Use the scheduler to iterate over timesteps
```

```
noise_scheduler.set_timesteps(1000)

for timestep in tqdm(noise_scheduler.timesteps):
    with torch.no_grad():
        residual = diffusion_backbone(image, timestep).sample
        image = noise_scheduler.step(residual, timestep,
image).prev_sample

    if timestep.item() % 200 == 0:
        images.append((image / 2 + 0.5).clamp(0, 1).cpu().permute(0,
2, 3, 1).numpy())
        labels.append(f"t={timestep.item()}")

for i in range(images[0].shape[0]):
    plot1xNArray([img[i] for img in images], labels)

100%|██████████| 1000/1000 [00:57<00:00, 17.50it/s]
```





The diffusers library also provides *Pipeline* classes, which are wrappers around the model that abstracts the inference loop implemented above.

We can create a pipeline, giving it the trained model and the noise scheduler, and use it to generate images. In this case, we will only have access to the final image, generated on the last timestep, but not the intermediary images from the denoising process.

```python
from diffusers import DDPMPipeline

pipeline = DDPMPipeline(diffusion_backbone, noise_scheduler)
generated_images = pipeline(10, output_type="np")

f, axarr = plt.subplots(1, len(generated_images["images"]))

for image, ax in zip(generated_images["images"], axarr):
    ax.imshow(image, cmap='gray')
    ax.axis('off')
```

{"model_id":"ddde767d1f314be0b0406c0f40f14357","version_major":2,"version_minor":0}