

HW7_GANs_(student_version)

April 10, 2024

1 HW 7 : GANs

In this assignment, you will learn to write an advanced PyTorch implementation concept commonly used in a complex deep learning pipeline by using GAN as a learning example.

You will also start working with more complex architectures (upsampling) and writing style (complex modules such as modulelist) for pytorch.

Every TODO is weighted equally. Optional TODO is half of a regular TODO.

1.1 GPU test

```
[ ]: # !nvidia-smi  
!system_profiler SPDisplaysDataType
```

Graphics/Displays:

Apple M2:

Chipset Model: Apple M2

Type: GPU

Bus: Built-In

Total Number of Cores: 8

Vendor: Apple (0x106b)

Metal Support: Metal 3

Displays:

MSI MD272XP:

Resolution: 1920 x 1080 (1080p FHD - Full High Definition)

UI Looks like: 1920 x 1080 @ 100.00Hz

Main Display: Yes

Mirror: Off

Online: Yes

Rotation: Supported

Automatically Adjust Brightness: No

Color LCD:

Display Type: Built-in Liquid Retina Display

Resolution: 2560 x 1664 Retina

Mirror: Off

Online: Yes

Automatically Adjust Brightness: Yes

Connection Type: Internal

2 Part 1 : WGAN-GP reimplementation

In this section, you are going to reimplement WGAN-GP (<https://arxiv.org/pdf/1704.00028.pdf>) based on the pseudocode provided in the paper to generate MNIST digit characters. Some parts are intentionally modified to discourage straight copypasting from public repositories.

The pseudocode could be organized into two main parts: discriminator optimization in line 2-10, and generator optimization in line 11-12.

The discriminator part consists of four steps: - Line 4: data, and noise sampling with a batch size of m - Line 5-7: discriminator loss calculation - Line 9: discriminator update - Repeat line 4-9 for n_{critic} steps

After the discriminator is updated, the generator is then updated by performing two steps: - Line 11: noise sampling - Line 12: generator loss calculation and update

This part is divided into four subsections: network initialization, hyperparameter initialization, data preparation, and training loop. The detail for each part will be explained in the subsections.

2.1 Downloading MNIST dataset

The MNIST dataset contains 60,000 training digit character image (0-9) at 28x28 resolution that are normalized to [0, 1]. Given the training images, your task is to generate new training images using WGAN-GP by learning from the training distribution.

```
[ ]: import torchvision.datasets as datasets
import numpy as np

mnist_trainset = datasets.MNIST(root='./data', train=True, download=True, transform=None)
trainX = np.array(mnist_trainset.data[..., None]).transpose(0, 3, 1, 2) / 255
print("Dataset size : ", trainX.shape)

/Users/pupipatsingkhorn/miniconda3/envs/datascience/lib/python3.9/site-
packages/torchvision/io/image.py:13: UserWarning: Failed to load image Python
extension:
'dlopen(/Users/pupipatsingkhorn/miniconda3/envs/datascience/lib/python3.9/site-
packages/torchvision/image.so, 0x0006): Symbol not found:
__ZN3c1017RegisterOperatorsD1Ev

Referenced from: <2D1B8D5C-7891-3680-9CF9-F771AE880676>
/Users/pupipatsingkhorn/miniconda3/envs/datascience/lib/python3.9/site-
packages/torchvision/image.so

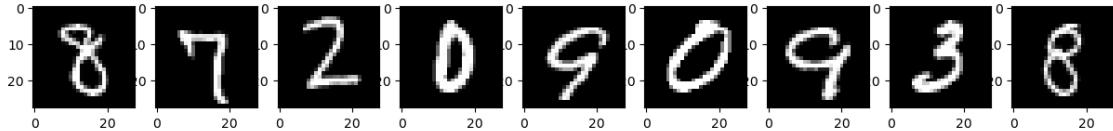
Expected in:      <657200D0-E150-3B86-A0C5-7FD4B7F7C533>
/Users/pupipatsingkhorn/miniconda3/envs/datascience/lib/python3.9/site-
packages/torch/lib/libtorch_cpu.dylib' If you don't plan on using image
functionality from `torchvision.io`, you can ignore this warning. Otherwise,
there might be something wrong with your environment. Did you have `libjpeg` or
`libpng` installed before building `torchvision` from source?

warn()

Dataset size : (60000, 1, 28, 28)
```

2.2 Dataset Visualization

```
[ ]: import matplotlib.pyplot as plt
import numpy as np
plt.figure(figsize = (15,75))
for i in range(9):
    plt.subplot( int('19{}'.format(i+1)) )
    plt.imshow( trainX[np.random.randint(len(trainX))].transpose((1, 2, 0))[...,0] , cmap = 'gray' )
plt.show()
```



2.3 Generator and Discriminator network

Before training, the deep learning networks have to be initialized first. Therefore, in this part, you are going to write a generator and discriminator network based on the description provided below.

The description of the discriminator network is shown in the Table below.

The network also has some specific requirements: - ConvBlock is a Convolution-ReLU layer - All ReLUs in the encoder are leaky, with a slope of 0.1

The description of the generator network is shown in the Table below.

The network also has some specific requirements: - ConvBlock is a ConvTranspose-BatchNorm-ReLU layer - Downsampling method is bilinear interpolation (torch.nn.Upsample or torch.nn.functional.interpolate)

TODO 1: Implement a discriminator network.

TODO 2: Implement a generator network.

```
[ ]: import torch
import torch.nn.functional as F
from torch import nn
from torchvision import transforms

class Discriminator(nn.Module):
    ##TODO1 implement the discriminator (critic)
    def __init__(self):
        super(Discriminator, self).__init__()
        # nn.Conv2d(in_channels, out_channels, kernel_size, stride, padding)
        self.conv1 = nn.Conv2d(1, 128, 5, 2, 2)
        self.conv2 = nn.Conv2d(128, 256, 5, 2, 2)
        self.conv3 = nn.Conv2d(256, 512, 5, 2, 2)
        self.linear = nn.Linear(512 * 4 * 4, 1)

    def forward(self, x):
        # x: [batch_size, 1, 28, 28]
        x = self.conv1(x) # [batch_size, 128, 14, 14]
```

```

        x = F.leaky_relu(x, 0.1)
        x = self.conv2(x) # [batch_size, 256, 7, 7]
        x = F.leaky_relu(x, 0.1)
        x = self.conv3(x) # [batch_size, 512, 4, 4]
        x = F.leaky_relu(x, 0.1)
        x = x.view(-1, 512 * 4 * 4) # [batch_size, 512 * 4 * 4]
        x = self.linear(x) # [batch_size, 1]
        return x

class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        self.linear = nn.Linear(128, 512 * 4 * 4)
        self.conv1 = nn.Sequential(
            # torch.nn.ConvTranspose2d(in_channels, out_channels, kernel_size, stride=1, padding=0, output_padding=0)
            nn.ConvTranspose2d(512, 256, kernel_size=5, stride=2, padding=2, output_padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU()
        )
        self.conv2 = nn.Sequential(
            nn.ConvTranspose2d(256, 128, kernel_size=5, stride=2, padding=2, output_padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU()
        )
        self.conv3 = nn.Sequential(
            nn.ConvTranspose2d(128, 1, kernel_size=5, stride=2, padding=2, output_padding=1),
            nn.Sigmoid()
        )

    def forward(self, x):
        x = x.view(-1, 128) # Reshape the input
        x = self.linear(x).view(-1, 512, 4, 4)
        x = self.conv1(x)
        x = self.conv2(x)
        x = self.conv3(x)
        x = F.interpolate(x, size=(28, 28), mode='bilinear', align_corners=False)
        return x

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f"Device: {device}")

```

```
discriminator = Discriminator().to(device)
generator = Generator().to(device)
```

Device: cpu

2.4 Network verification

TODO 3: What is the input and output shape of the generator and discriminator network? Verify that the implemented networks are the same as the answer you have provided.

Ans:

Discriminator

Input: (BATCH_SIZE, 1, 28, 28)
Output: (BATCH_SIZE, 1)

Generator

Input: (BATCH_SIZE, 128)
Output: (BATCH_SIZE, 1, 28, 28)

```
[ ]: from torchinfo import summary

BATCH_SIZE = 32
print(summary(Discriminator(), input_size=(BATCH_SIZE, 1, 28, 28)))
```

=====

=====

Layer (type:depth-idx)	Output Shape	Param #
Discriminator	[32, 1]	--
Conv2d: 1-1	[32, 128, 14, 14]	3,328
Conv2d: 1-2	[32, 256, 7, 7]	819,456
Conv2d: 1-3	[32, 512, 4, 4]	3,277,312
Linear: 1-4	[32, 1]	8,193

=====

=====

Total params: 4,108,289

Trainable params: 4,108,289

```
Non-trainable params: 0
```

```
Total mult-adds (G): 2.98
```

```
=====
```

```
Input size (MB): 0.10
```

```
Forward/backward pass size (MB): 11.73
```

```
Params size (MB): 16.43
```

```
Estimated Total Size (MB): 28.26
```

```
=====
```

```
[ ]: print(summary(Generator(), input_size=(BATCH_SIZE, 128)))
```

```
=====
```

Layer (type:depth-idx)	Output Shape	Param #
Generator	[32, 1, 28, 28]	--
Linear: 1-1	[32, 8192]	1,056,768
Sequential: 1-2	[32, 256, 8, 8]	--
ConvTranspose2d: 2-1	[32, 256, 8, 8]	3,277,056
BatchNorm2d: 2-2	[32, 256, 8, 8]	512
ReLU: 2-3	[32, 256, 8, 8]	--
Sequential: 1-3	[32, 128, 16, 16]	--
ConvTranspose2d: 2-4	[32, 128, 16, 16]	819,328
BatchNorm2d: 2-5	[32, 128, 16, 16]	256
ReLU: 2-6	[32, 128, 16, 16]	--

Sequential: 1-4	[32, 1, 32, 32]	--
ConvTranspose2d: 2-7	[32, 1, 32, 32]	3,201
Sigmoid: 2-8	[32, 1, 32, 32]	--

Total params: 5,157,121

Trainable params: 5,157,121

Non-trainable params: 0

Total mult-adds (G): 13.56

Input size (MB): 0.02

Forward/backward pass size (MB): 27.53

Params size (MB): 20.63

Estimated Total Size (MB): 48.17

2.5 Parameter Initialization

After the network is initialized, we then set up training hyperparameters for the training. In this part, hyperparameters have already been partially provided in the cell below, though some of them are intentionally left missing (`None`). Your task is to fill the missing parameters based on the pseudocode above.

TODO4: Initialize the missing model hyperparameters and optimizers based on the pseudocode above.

Note: To hasten the training process of our toy experiment, the training step and batch size is reduced to 3000 and 32, respectively.

```
[ ]: NUM_ITERATION = 3000
BATCH_SIZE = 32
fixed_z = torch.randn((8, 128), device=device)
def schedule(i):
    lr = 1e-4
```

```

    if(i > 2500): lr *= 0.1
    return lr
losses = {'D' : [None], 'G' : [None]}

## TODO4 initialize missing hyperparameter and optimizer
# from Algorithm 1 pseudocode
G_optimizer = torch.optim.Adam(generator.parameters(), lr=1e-4, betas=(0, 0.9))
D_optimizer = torch.optim.Adam(discriminator.parameters(), lr=1e-4, betas=(0, 0.
    ↪9))
GP_lambda = 10
n_critic = 5

```

2.6 Data preparation

TODO 5: Create a dataloader that could generate the data in line 4. The dataloader should return \mathbf{x}, \mathbf{z} , with a batch size of BATCH_SIZE

```

[ ]: from torch.utils.data import DataLoader, Dataset

# TODO5 implement dataloader
class WGAN_Dataset(Dataset):
    def __init__(self, data):
        self.data = data.astype(np.float32) #np.ndarray #default numpy: ↪
        ↪float64, pytorch: float32
        self.len = len(data)

    def __len__(self):
        return self.len

    def __getitem__(self, idx):
        x = self.data[idx]
        z = torch.randn(128) #Latent variable tensor #Latent Size = 128 #size ↪
        ↪of the random noise vector
        epsilon = torch.rand((1,1,1)) # 1 value #for (batch_size, 1, 1, 1)
        return x, z, epsilon

wgan_dataset = WGAN_Dataset(trainX) # Assuming 'data' contains your dataset
wgan_dataloader = DataLoader(wgan_dataset, batch_size=BATCH_SIZE, shuffle=True)

```

2.7 Training loop

This section is the place where the training section starts. It is highly recommended that you understand the pseudocode before performing the tasks below. To train the WGAN-GP you have to perform the following tasks:

TODO6: Update the learning rate base on the provided scheduler.

TODO7: Sample the data from the dataloader (Line 4).

TODO8 : Calcualte the discriminator loss (Line 5-7). - In the line 7 you have to implement the gradient penalty term $\lambda(\|(\hat{x} D_w(\hat{x}))\|_2 - 1)^2$, which is a custom gradient. You may read

<https://pytorch.org/docs/stable/generated/torch.autograd.grad.html> to find how custom gradient is implemented. - HINT: Gradient norm calculation is still part of the computation graph.

TODO9: Update the discriminator loss (Line 9).

TODO10: Calculate and update the generator loss (Line 11-12).

If your implementation is correct, the generated images should resemble an actual digit character after 500 iterations.

```
[ ]: from tqdm import tqdm
for i in tqdm(range(NUM_ITERATION)):
    ## TODO6 update learning rate
    D_optimizer.param_groups[0]['lr'] = schedule(i)
    G_optimizer.param_groups[0]['lr'] = schedule(i)

    for t in range(n_critic):
        ## TODO7 line 4: sample data from dataloader
        x, z, e = next(iter(wgan_dataloader))
        x, z, e = x.to(device), z.to(device), e.to(device)

        ## TODO8 line5-7 : calculate discriminator loss
        generator.zero_grad()
        discriminator.zero_grad()

        x_bar = generator(z)
        # print(e.shape, x.shape, x_bar.shape)
        x_hat = e * x + (1 - e) * x_bar

        disc_x_bar = discriminator(x_bar)
        disc_x = discriminator(x)
        disc_x_hat = discriminator(x_hat)

        div_x_hat = torch.autograd.grad(disc_x_hat, x_hat, grad_outputs=torch.
        ones_like(disc_x_hat), create_graph=True)[0]
        grad_loss = GP_lambda * (div_x_hat.norm(dim=(2,3)) - 1) ** 2
        D_loss = disc_x_bar.mean() - disc_x.mean() + grad_loss.mean()
        losses['D'].append(D_loss.item())

        ## TODO9 : line 9 update discriminator loss
        D_loss.backward()
        D_optimizer.step()

## TODO10 : line 11-12 calculate and update the generator loss
generator.zero_grad()

z = torch.randn((BATCH_SIZE, 128), device=device)
G_loss = -discriminator(generator(z)).mean()
losses['G'].append(G_loss.item())
```

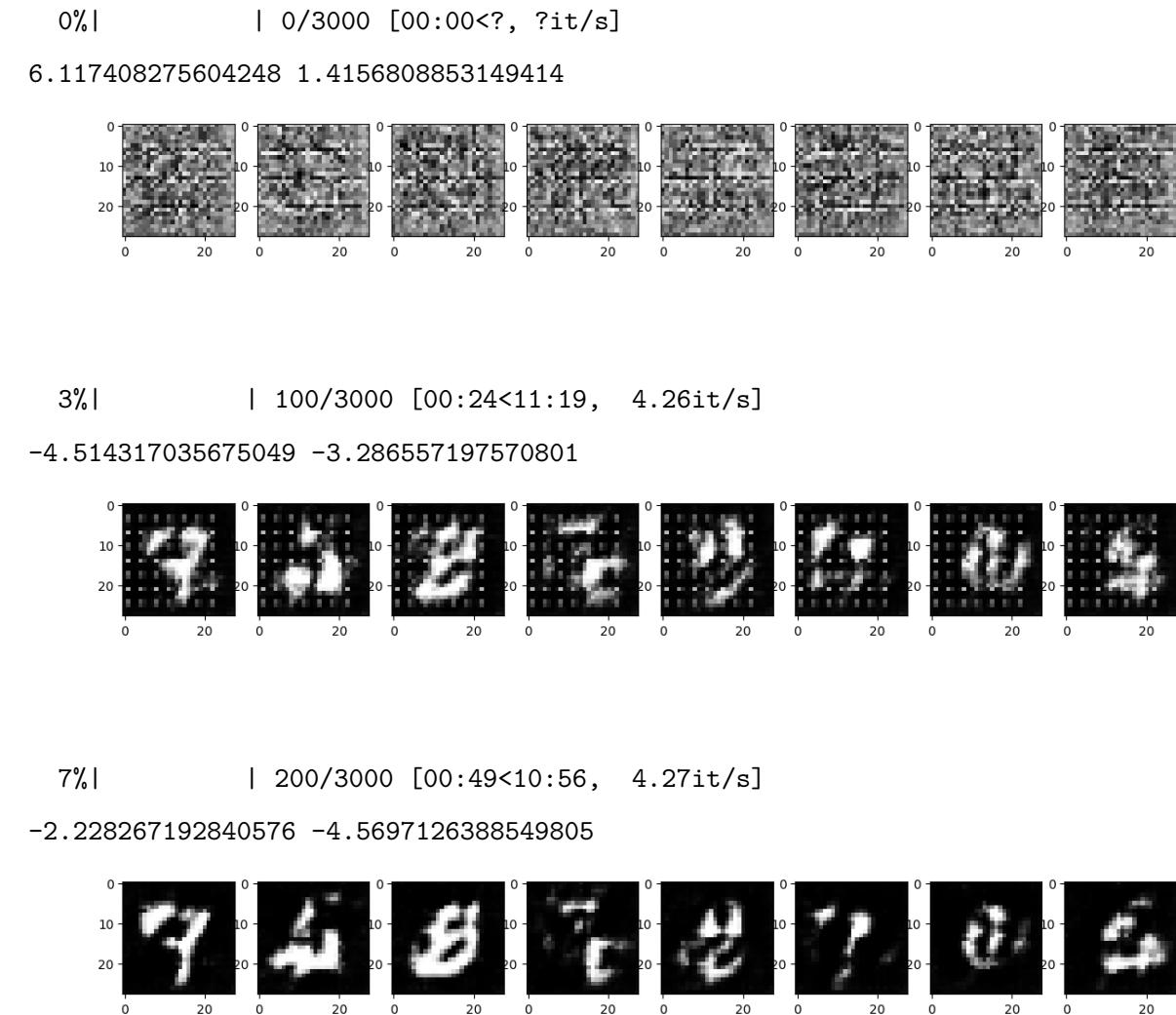
```

G_loss.backward()
G_optimizer.step()

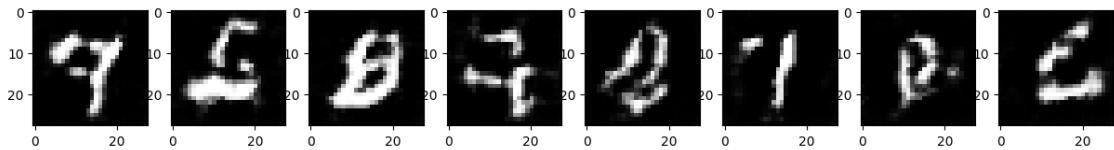
# Output visualization : If your reimplementation is correct, the generated
# images should start resembling a digit character after 500 iterations.

if(i % 100 == 0):
    plt.figure(figsize = (15,75))
    print(losses['D'][-1], losses['G'][-1])
    with torch.no_grad():
        res = generator(fixed_z).cpu().detach().numpy()
    for k in range(8):
        plt.subplot( int('18{}'.format(k+1)) )
        plt.imshow( res[k].transpose(1, 2, 0)[..., 0], cmap = 'gray' )
    plt.show()

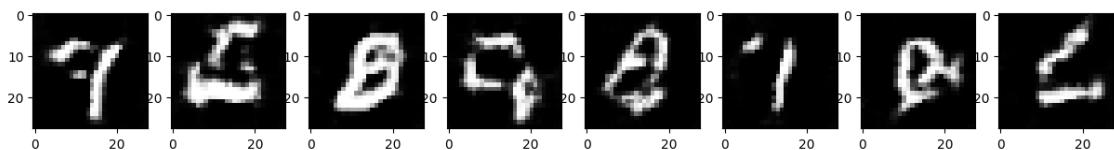
```



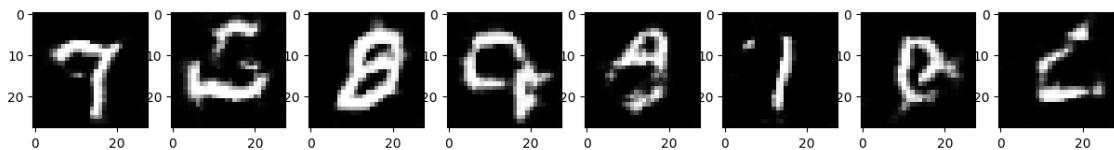
10% | 300/3000 [01:13<10:33, 4.26it/s]
-2.2778420448303223 -4.346742630004883



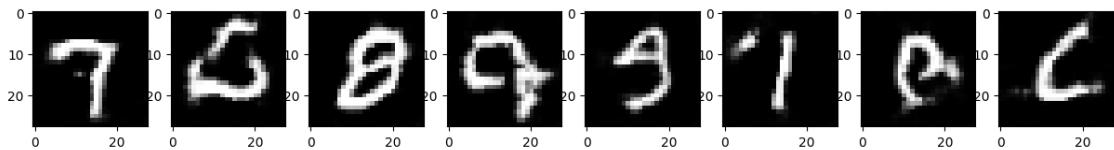
13% | 400/3000 [01:37<10:10, 4.26it/s]
-1.5141026973724365 -3.2490615844726562



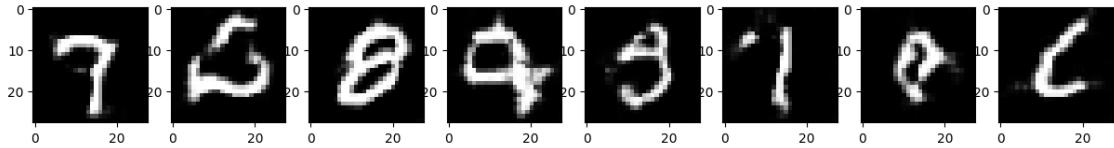
17% | 500/3000 [02:02<09:46, 4.26it/s]
-1.8121957778930664 -3.4990992546081543



20% | 600/3000 [02:26<09:23, 4.26it/s]
-1.8104069232940674 -2.4585025310516357

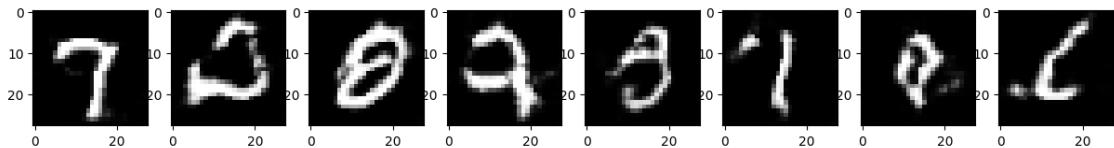


23% | 700/3000 [02:50<09:00, 4.26it/s]
-1.5247865915298462 -3.634577751159668



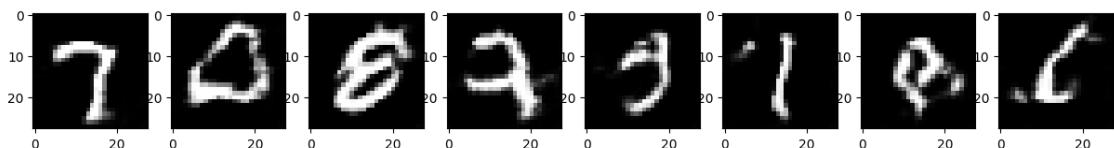
27% | 800/3000 [03:15<08:36, 4.26it/s]

-1.1932646036148071 -1.4787293672561646



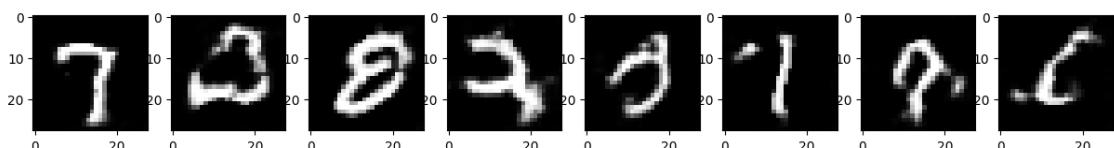
30% | 900/3000 [03:39<08:12, 4.27it/s]

-1.5655357837677002 -2.4776453971862793



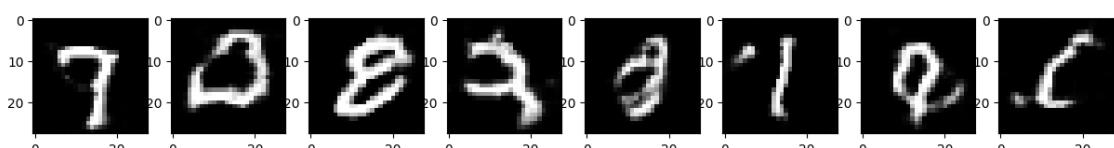
33% | 1000/3000 [04:03<07:49, 4.26it/s]

-1.7355915307998657 -1.8612182140350342



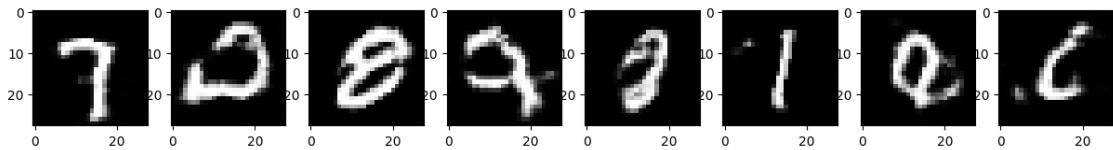
37% | 1100/3000 [04:28<07:25, 4.26it/s]

-1.544386386871338 -1.4005322456359863



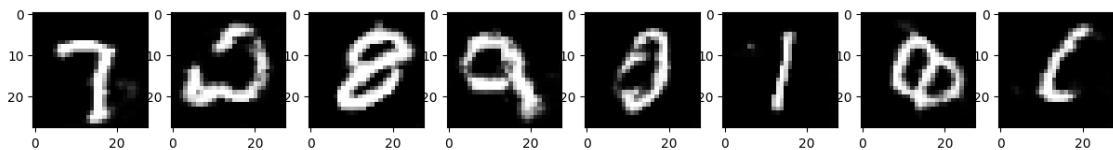
40% | 1200/3000 [04:52<07:02, 4.26it/s]

-1.0149846076965332 -2.193079948425293



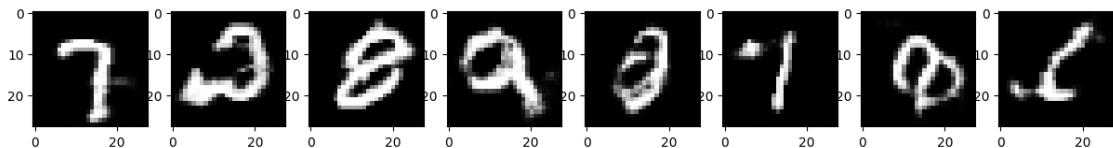
43% | 1300/3000 [05:16<06:39, 4.26it/s]

-1.5951862335205078 -1.2918610572814941



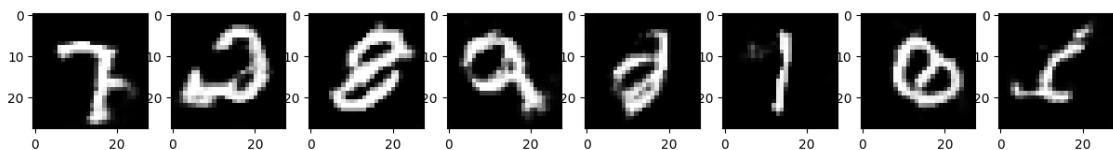
47% | 1400/3000 [05:41<06:15, 4.26it/s]

-1.21054208278656 -1.834031105041504



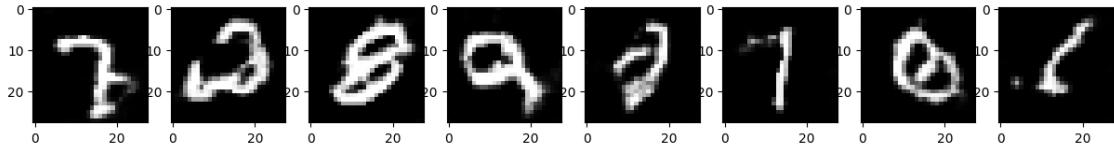
50% | 1500/3000 [06:05<05:51, 4.26it/s]

-1.2896666526794434 -0.191249817609787



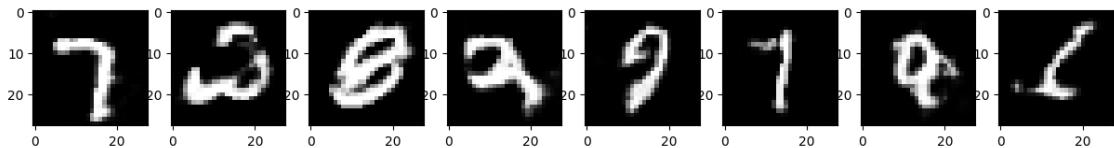
53% | 1600/3000 [06:29<05:28, 4.27it/s]

-1.4895142316818237 -0.611272931098938



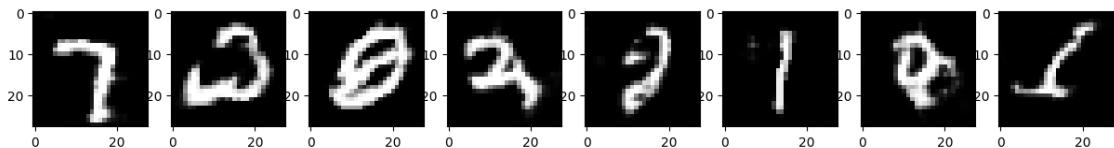
57% | 1700/3000 [06:54<05:04, 4.27it/s]

-1.4340990781784058 -0.6079342365264893



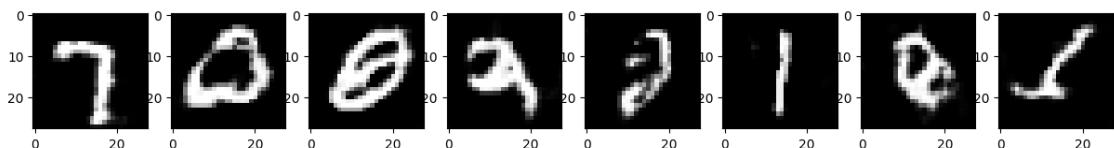
60% | 1800/3000 [07:18<04:41, 4.27it/s]

-1.9019250869750977 -1.0189197063446045



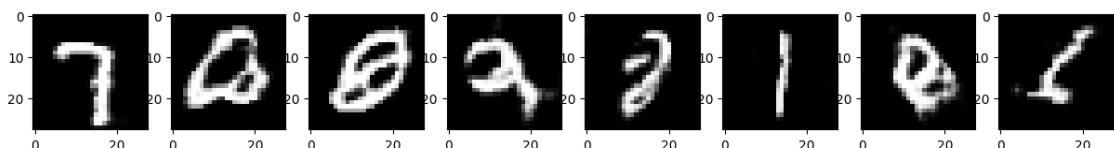
63% | 1900/3000 [07:42<04:18, 4.26it/s]

-1.81014084815979 -1.0684689283370972



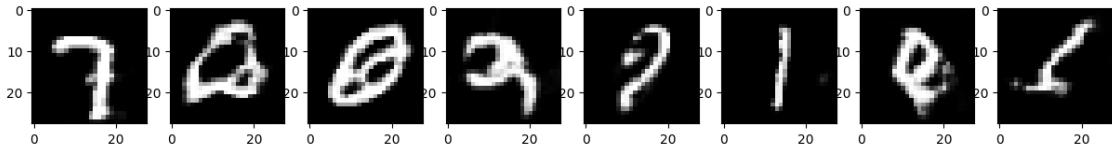
67% | 2000/3000 [08:06<03:55, 4.25it/s]

-1.3439581394195557 -0.8636952638626099



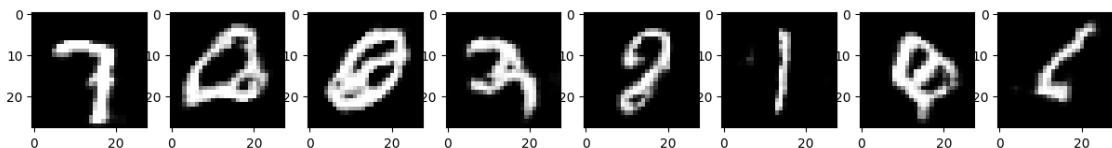
70% | 2100/3000 [08:31<03:31, 4.26it/s]

-1.5867750644683838 -0.10043462365865707



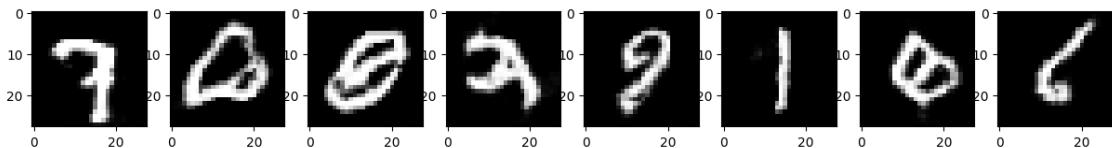
73% | 2200/3000 [08:55<03:08, 4.26it/s]

-1.4941035509109497 -0.9734705090522766



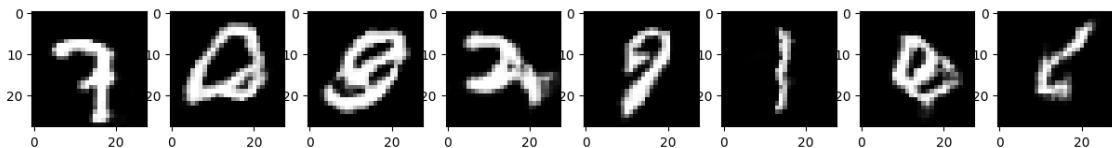
77% | 2300/3000 [09:19<02:44, 4.27it/s]

-1.2061653137207031 0.012381002306938171



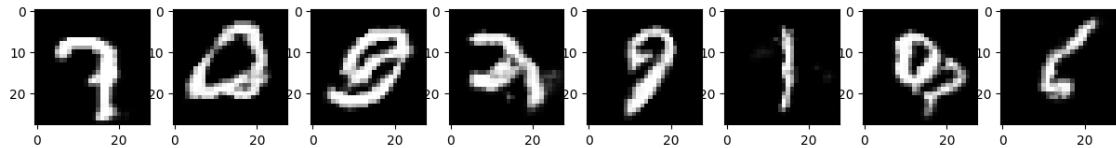
80% | 2400/3000 [09:44<02:20, 4.27it/s]

-1.6655964851379395 -1.1424744129180908



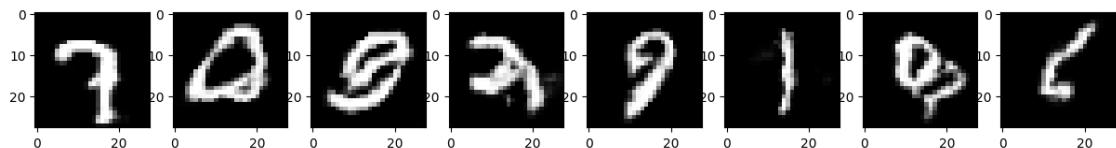
83% | 2500/3000 [10:08<01:57, 4.26it/s]

-1.3072105646133423 -0.4312283396720886



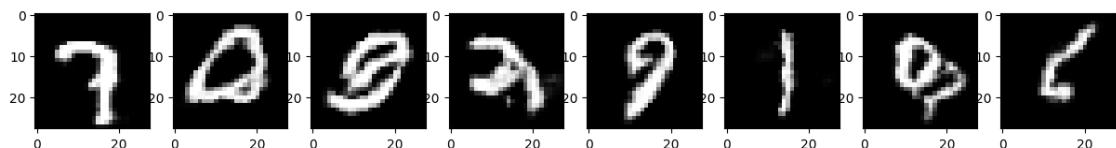
87% | 2600/3000 [10:32<01:33, 4.27it/s]

-1.3357269763946533 -0.7936104536056519



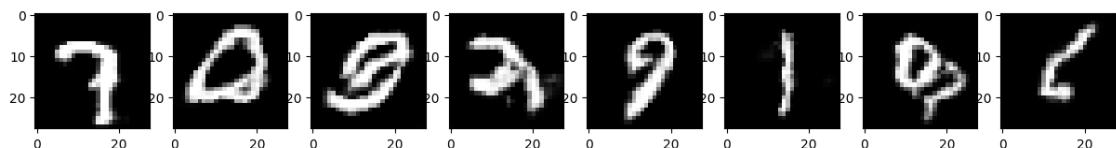
90% | 2700/3000 [10:57<01:10, 4.26it/s]

-1.3793787956237793 -0.3297176659107208



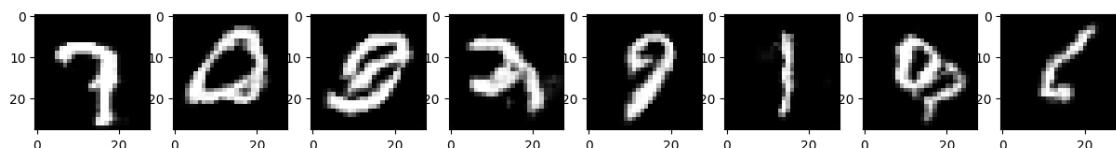
93% | 2800/3000 [11:21<00:46, 4.26it/s]

-1.4607305526733398 -0.3439843952655792



97% | 2900/3000 [11:45<00:23, 4.27it/s]

-1.6102235317230225 -0.27046599984169006



```

100%| 3000/3000 [12:10<00:00, 4.11it/s]

[ ]: plt.title('Discriminator Loss')
plt.plot(losses['D'])
plt.show()

plt.title('Generator Loss')
plt.plot(losses['G'])
plt.show()

[ ]: # Specify the path where you want to save the checkpoint
checkpoint_path = "/kaggle/working/wgan_checkpoint.pth"

# Save the model checkpoints
torch.save({
    'generator_state_dict': generator.state_dict(),
    'discriminator_state_dict': discriminator.state_dict(),
    'G_optimizer_state_dict': G_optimizer.state_dict(),
    'D_optimizer_state_dict': D_optimizer.state_dict(),
    'losses': losses,
    'num_iteration': NUM_ITERATION,
    'batch_size': BATCH_SIZE,
    'gp_lambda': GP_lambda,
    'n_critic': n_critic
}, checkpoint_path)

```

3 Part 2 : pix2pix reimplementation (cGAN on paired image translation)

In this exercise, we are reimplementing a paired image translation model, an application of a generative adversarial network (GAN). The model we are going to implement is pix2pix (<https://arxiv.org/pdf/1611.07004.pdf>), one of the earliest paired image translation models based on GAN. The pipeline of pix2pix is shown in the Figure below.

From the figure above, the pipeline consists of three main parts:

- 1. Generation phase : the generator G create the generated image $G(x)$ from the given input x .
- 2. Discrimination phase :

In step 2.1, the discriminator D receives an input image x and the generated image $G(x)$, then the discriminator has to learn to predict that the generated image $G(x)$ is fake.

In step 2.2, the discriminator D receives an input image x and the ground truth image y , then the discriminator has to learn to predict that the image y is real.

- 3. Refinement phase: Refine the quality of the generated image $G(x)$ by encouraging the generated image to be close to an actual image y by using L1 as an objective.

The objective of pix2pix is to train an optimal generator G^* based on the objective function : $G^* = \operatorname{argmin}_G \max_D L_{cGAN}(G, D) + \lambda L_1(G)$

- The term $\operatorname{argmin}_G \max_D L_{cGAN}(G, D)$ is the objective function of the first and second step, which is a standard cGAN loss : $L_{cGAN}(G, D) = E_{\{x,y\}}[\log D(x,y)] + E_{\{x,z\}}[\log(1 - D(x, G(x,z)))]$. The noise z is embedded in the generator in the form of dropout.
- The term $L_1(G)$ is the objective function of the third step where $L_1(G) = E_{x,y,z}[||y - G(x, z)||_1]$

The subsections will explain the dataset and training setup of this exercise.

3.1 Get dataset

```
[ ]: !wget http://efrosgans.eecs.berkeley.edu/pix2pix/datasets/facades.tar.gz
!tar -xzf facades.tar.gz
```

```
--2024-04-10 14:08:00--
http://efrosgans.eecs.berkeley.edu/pix2pix/datasets/facades.tar.gz
Resolving efrosgans.eecs.berkeley.edu (efrosgans.eecs.berkeley.edu)...
128.32.244.190
Connecting to efrosgans.eecs.berkeley.edu
(efrosgans.eecs.berkeley.edu)|128.32.244.190|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 30168306 (29M) [application/x-gzip]
Saving to: 'facades.tar.gz'

facades.tar.gz      100%[=====] 28.77M  4.51MB/s    in 8.9s

2024-04-10 14:08:09 (3.24 MB/s) - 'facades.tar.gz' saved [30168306/30168306]
```

3.2 Import library

```
[ ]: import cv2
import glob
import numpy as np
import torch
import torch.nn.functional as F
from torch import nn
from torchvision import transforms
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
```

3.3 Setting up facade dataset

The dataset chosen for this exercise is the CMP Facade Database which is a pair of facade images and its segmented component stored in RGB value. The objective of this exercise is to generate a facade given its simplified segmented component. Both input and output is a 256 x 256 RGB image normalized to [-1, 1].

```
[ ]: train = (np.array([cv2.imread(i) for i in glob.glob('facades/train/*')], dtype=np.float32) / 255).transpose((0, 3, 1, 2))
train = (train - 0.5) * 2 #shift from [0,1] to [-1, 1]
trainX = train[:, :, :, 256:]
trainY = train[:, :, :, :256]

val = (np.array([cv2.imread(i) for i in glob.glob('facades/val/*')], dtype = np.float32) / 255).transpose((0, 3, 1, 2))
val = (val - 0.5) * 2 #shift from [0,1] to [-1, 1]
valX = val[:, :, :, 256:]
valY = val[:, :, :, :256]

print("Input size : {}, Output size = {}".format(trainX.shape, trainY.shape))
```

Input size : (400, 3, 256, 256), Output size = (400, 3, 256, 256)

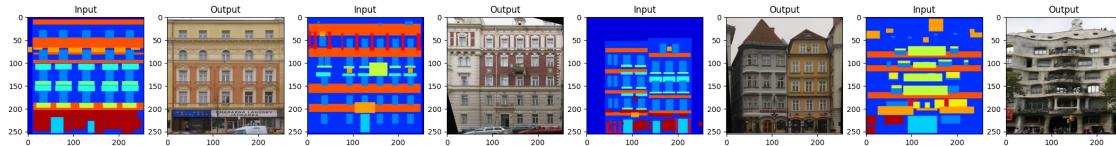
3.4 Dataset Visualization

```
[ ]: import matplotlib.pyplot as plt
import numpy as np
plt.figure(figsize = (30,90))
for i in range(4):
    idx = np.random.randint(len(trainX))

    plt.subplot( int('19{}'.format(2*i+1)) )
    plt.title('Input')
    plt.imshow( (0.5 * trainX[idx].transpose((1, 2, 0)) + 0.5)[..., :-1] , cmap='gray' )

    plt.subplot( int('19{}'.format(2*i+2)) )
    plt.title('Output')
    plt.imshow( (0.5 * trainY[idx].transpose((1, 2, 0)) + 0.5)[..., :-1] , cmap ='gray' )

plt.show()
```



3.5 Note

If you have trouble understanding the instruction provided in this homework or have any ambiguity about the instruction, you could also read the appendix section (section 6.1-6.2) in the paper for a detailed explanation.

3.6 Discriminator network

In this section, we are going to implement a discriminator network of pix2pix. The description of the discriminator network is provided in the Figure below.

The network also has the following specific requirements:

- All convolutions are 4×4 spatial filters
- ConvBlock is a Convolution-InstanceNorm-ReLU layer
- InstanceNorm is not applied to the first C64 layer
- All ReLUs are leaky, with a slope of 0.2

TODO 11: Implement the discriminator network based on the description above.

TODO 12: What should be the size of the input and output of the discriminator for this task? Verify that the input and output of the implemented network are the same as the answer you have provided.

ANS:

Input: [(batch_size, 3, 256, 256), (batch_size, 3, 256, 256)]
Output: (batch_size, 1, 30, 30)

```
[ ]: class Discriminator(nn.Module):
    #TODO11 implement the discriminator network
    def __init__(self):
        super(Discriminator, self).__init__()
        self.ConvBlock64 = nn.Sequential(
            nn.Conv2d(6, 64, 4, 2, 1),
            nn.LeakyReLU(0.2)
        )
        self.ConvBlock128 = nn.Sequential(
            nn.Conv2d(64, 128, 4, 2, 1),
            nn.InstanceNorm2d(128),
            nn.LeakyReLU(0.2)
        )
        self.ConvBlock256 = nn.Sequential(
            nn.Conv2d(128, 256, 4, 2, 1),
            nn.InstanceNorm2d(256),
            nn.LeakyReLU(0.2)
        )
        self.ConvBlock512 = nn.Sequential(
            nn.Conv2d(256, 512, 4, 1, 1),
            nn.InstanceNorm2d(512),
            nn.LeakyReLU(0.2)
```

```

        )
    self.ConvBlock1 = nn.Sequential(
        nn.Conv2d(512, 1, 4, 1, 1),
        nn.InstanceNorm2d(1),
        nn.LeakyReLU(0.2)
    )
    self.sigmoid = nn.Sigmoid()

def forward(self, x, y):
    input = torch.cat([x, y], dim=1)
    hd1 = self.ConvBlock64(input)
    hd2 = self.ConvBlock128(hd1)
    hd3 = self.ConvBlock256(hd2)
    hd4 = self.ConvBlock512(hd3)
    hd5 = self.ConvBlock1(hd4)
    out = self.sigmoid(hd5)
    return out

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f"Device: {device}")

discriminator = Discriminator().to(device)

```

Device: cuda

```
[ ]: #TODO12 verify the discriminator
from torchinfo import summary
print(summary(discriminator, input_size=[(32, 3, 256, 256), (32, 3, 256, 256)]))
```

Layer (type:depth-idx)	Output Shape	Param #
Discriminator	[32, 1, 30, 30]	--
Sequential: 1-1	[32, 64, 128, 128]	--
Conv2d: 2-1	[32, 64, 128, 128]	6,208
LeakyReLU: 2-2	[32, 64, 128, 128]	--
Sequential: 1-2	[32, 128, 64, 64]	--
Conv2d: 2-3	[32, 128, 64, 64]	131,200
InstanceNorm2d: 2-4	[32, 128, 64, 64]	--
LeakyReLU: 2-5	[32, 128, 64, 64]	--
Sequential: 1-3	[32, 256, 32, 32]	--
Conv2d: 2-6	[32, 256, 32, 32]	524,544
InstanceNorm2d: 2-7	[32, 256, 32, 32]	--
LeakyReLU: 2-8	[32, 256, 32, 32]	--
Sequential: 1-4	[32, 512, 31, 31]	--

Conv2d: 2-9	[32, 512, 31, 31]	2,097,664
InstanceNorm2d: 2-10	[32, 512, 31, 31]	--
LeakyReLU: 2-11	[32, 512, 31, 31]	--
Sequential: 1-5	[32, 1, 30, 30]	--
Conv2d: 2-12	[32, 1, 30, 30]	8,193
InstanceNorm2d: 2-13	[32, 1, 30, 30]	--
LeakyReLU: 2-14	[32, 1, 30, 30]	--
Sigmoid: 1-6	[32, 1, 30, 30]	--

Total params: 2,767,809
Trainable params: 2,767,809
Non-trainable params: 0
Total mult-adds (G): 102.38

Input size (MB): 50.33
Forward/backward pass size (MB): 595.95
Params size (MB): 11.07
Estimated Total Size (MB): 657.36

3.7 Generator network

In this section, we are going to implement a generator network of pix2pix. The generator is based on the U-NET based architecture (<https://arxiv.org/abs/1505.04597>). The Description of the generator network is provided in the Figure below.

The network also has the following specific requirements:

- All convolutions are 4×4 spatial filters
- ConvBlock is a Convolution-InstanceNorm-ReLU layer
- ConvTBlock is a ConvolutionTranspose-InstanceNorm-DropOut-ReLU layer with a dropout rate of 50%
- InstanceNorm is not applied to the first C64 layer in the encoder
- All ReLUs in the encoder are leaky, with a slope of 0.2, while ReLUs in the decoder are not leaky

TODO 13: Implement the generator network based on the description above.

TODO 14: What should be the size of the input and output of the generator for this task? Verify that the input and output of the implemented network are the same as the answer you have provided.

ANS:

Input: (32, 3, 256, 256)
Output: (32, 3, 256, 256)

[]: #HINT : you could also put multiple layers in a single list using nn.ModuleList
class Example(nn.Module):

```

def __init__(self):
    super().__init__()
    self.convs = nn.ModuleList([nn.Conv2d(25, 25, 3) for i in range(5)])
def forward(self, x):
    for i in range(len(self.convs)):
        x = self.convs[i](x)
    return x
ex = Example().to(device)
print(ex(torch.zeros((8, 25, 32, 32)).to(device)).shape)

```

torch.Size([8, 25, 22, 22])

```

[ ]: class myConv2d(nn.Module):
    def __init__(self, in_channels, out_channels):
        super().__init__()
        self.block = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, 4, 2, 1),
            nn.InstanceNorm2d(out_channels),
            nn.LeakyReLU(0.2))

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

class myConvT2d(nn.Module):
    def __init__(self, in_channels, out_channels, p_dropout=0.5):
        super().__init__()
        self.block = nn.Sequential(
            nn.ConvTranspose2d(in_channels, out_channels, 4, 2, 1),
            nn.InstanceNorm2d(out_channels),
            nn.Dropout(p_dropout),
            nn.ReLU())

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

class Generator(nn.Module):
    #TODO13 implement the generator network
    def __init__(self):
        super().__init__()

        self.down128 = nn.Sequential(
            nn.Conv2d(3, 64, 4, 2, 1),
            nn.LeakyReLU(0.2))
        self.down64 = myConv2d(64, 128)
        self.down32 = myConv2d(128, 256)
        self.down16 = myConv2d(256, 512)
        self.down8 = myConv2d(512, 512)

```

```

    self.down4 = myConv2d(512, 512)
    self.down2 = myConv2d(512, 512)
    self.down1 = nn.Sequential(
        nn.Conv2d(512, 512, 4, 2, 1),
        nn.LeakyReLU(0.2))

    self.up2 = myConvT2d(512, 512)
    self.up4 = myConvT2d(1024, 512)
    self.up8 = myConvT2d(1024, 512)
    self.up16 = myConvT2d(1024, 512)
    self.up32 = myConvT2d(1024, 256, 0.15)
    self.up64 = myConvT2d(512, 128, 0.15)
    self.up128 = myConvT2d(256, 64, 0.15)
    self.up256 = nn.Sequential(
        nn.ConvTranspose2d(128, 3, 4, 2, 1),
        nn.InstanceNorm2d(3))

def forward(self, x):
    d128 = self.down128(x)
    d64 = self.down64(d128)
    d32 = self.down32(d64)
    d16 = self.down16(d32)
    d8 = self.down8(d16)
    d4 = self.down4(d8)
    d2 = self.down2(d4)
    d1 = self.down1(d2)

    u2 = self.up2(d1)
    u4 = self.up4(torch.cat([u2, d2], 1))
    u8 = self.up8(torch.cat([u4, d4], 1))
    u16 = self.up16(torch.cat([u8, d8], 1))
    u32 = self.up32(torch.cat([u16, d16], 1))
    u64 = self.up64(torch.cat([u32, d32], 1))
    u128 = self.up128(torch.cat([u64, d64], 1))
    u256 = self.up256(torch.cat([u128, d128], 1))
    out = torch.tanh(u256)

    return out

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f"Device: {device}")

generator = Generator().to(device)

```

Device: cuda

```
[ ]: #TODO14 verify the generator
print(summary(generator, input_size=(1,3, 256, 256)))
```

Layer (type:depth-idx)	Output Shape	Param #
Generator	[1, 3, 256, 256]	--
Sequential: 1-1	[1, 64, 128, 128]	--
Conv2d: 2-1	[1, 64, 128, 128]	3,136
LeakyReLU: 2-2	[1, 64, 128, 128]	--
myConv2d: 1-2	[1, 128, 64, 64]	--
Sequential: 2-3	[1, 128, 64, 64]	--
Conv2d: 3-1	[1, 128, 64, 64]	131,200
InstanceNorm2d: 3-2	[1, 128, 64, 64]	--
LeakyReLU: 3-3	[1, 128, 64, 64]	--
myConv2d: 1-3	[1, 256, 32, 32]	--
Sequential: 2-4	[1, 256, 32, 32]	--
Conv2d: 3-4	[1, 256, 32, 32]	524,544
InstanceNorm2d: 3-5	[1, 256, 32, 32]	--
LeakyReLU: 3-6	[1, 256, 32, 32]	--
myConv2d: 1-4	[1, 512, 16, 16]	--
Sequential: 2-5	[1, 512, 16, 16]	--
Conv2d: 3-7	[1, 512, 16, 16]	2,097,664
InstanceNorm2d: 3-8	[1, 512, 16, 16]	--
LeakyReLU: 3-9	[1, 512, 16, 16]	--
myConv2d: 1-5	[1, 512, 8, 8]	--
Sequential: 2-6	[1, 512, 8, 8]	--
Conv2d: 3-10	[1, 512, 8, 8]	4,194,816
InstanceNorm2d: 3-11	[1, 512, 8, 8]	--
LeakyReLU: 3-12	[1, 512, 8, 8]	--
myConv2d: 1-6	[1, 512, 4, 4]	--
Sequential: 2-7	[1, 512, 4, 4]	--
Conv2d: 3-13	[1, 512, 4, 4]	4,194,816
InstanceNorm2d: 3-14	[1, 512, 4, 4]	--
LeakyReLU: 3-15	[1, 512, 4, 4]	--
myConv2d: 1-7	[1, 512, 2, 2]	--
Sequential: 2-8	[1, 512, 2, 2]	--
Conv2d: 3-16	[1, 512, 2, 2]	4,194,816
InstanceNorm2d: 3-17	[1, 512, 2, 2]	--
LeakyReLU: 3-18	[1, 512, 2, 2]	--
Sequential: 1-8	[1, 512, 1, 1]	--
Conv2d: 2-9	[1, 512, 1, 1]	4,194,816
LeakyReLU: 2-10	[1, 512, 1, 1]	--
myConvT2d: 1-9	[1, 512, 2, 2]	--
Sequential: 2-11	[1, 512, 2, 2]	--
ConvTranspose2d: 3-19	[1, 512, 2, 2]	4,194,816

InstanceNorm2d: 3-20	[1, 512, 2, 2]	--
Dropout: 3-21	[1, 512, 2, 2]	--
ReLU: 3-22	[1, 512, 2, 2]	--
myConvT2d: 1-10	[1, 512, 4, 4]	--
Sequential: 2-12	[1, 512, 4, 4]	--
ConvTranspose2d: 3-23	[1, 512, 4, 4]	8,389,120
InstanceNorm2d: 3-24	[1, 512, 4, 4]	--
Dropout: 3-25	[1, 512, 4, 4]	--
ReLU: 3-26	[1, 512, 4, 4]	--
myConvT2d: 1-11	[1, 512, 8, 8]	--
Sequential: 2-13	[1, 512, 8, 8]	--
ConvTranspose2d: 3-27	[1, 512, 8, 8]	8,389,120
InstanceNorm2d: 3-28	[1, 512, 8, 8]	--
Dropout: 3-29	[1, 512, 8, 8]	--
ReLU: 3-30	[1, 512, 8, 8]	--
myConvT2d: 1-12	[1, 512, 16, 16]	--
Sequential: 2-14	[1, 512, 16, 16]	--
ConvTranspose2d: 3-31	[1, 512, 16, 16]	8,389,120
InstanceNorm2d: 3-32	[1, 512, 16, 16]	--
Dropout: 3-33	[1, 512, 16, 16]	--
ReLU: 3-34	[1, 512, 16, 16]	--
myConvT2d: 1-13	[1, 256, 32, 32]	--
Sequential: 2-15	[1, 256, 32, 32]	--
ConvTranspose2d: 3-35	[1, 256, 32, 32]	4,194,560
InstanceNorm2d: 3-36	[1, 256, 32, 32]	--
Dropout: 3-37	[1, 256, 32, 32]	--
ReLU: 3-38	[1, 256, 32, 32]	--
myConvT2d: 1-14	[1, 128, 64, 64]	--
Sequential: 2-16	[1, 128, 64, 64]	--
ConvTranspose2d: 3-39	[1, 128, 64, 64]	1,048,704
InstanceNorm2d: 3-40	[1, 128, 64, 64]	--
Dropout: 3-41	[1, 128, 64, 64]	--
ReLU: 3-42	[1, 128, 64, 64]	--
myConvT2d: 1-15	[1, 64, 128, 128]	--
Sequential: 2-17	[1, 64, 128, 128]	--
ConvTranspose2d: 3-43	[1, 64, 128, 128]	262,208
InstanceNorm2d: 3-44	[1, 64, 128, 128]	--
Dropout: 3-45	[1, 64, 128, 128]	--
ReLU: 3-46	[1, 64, 128, 128]	--
Sequential: 1-16	[1, 3, 256, 256]	--
ConvTranspose2d: 2-18	[1, 3, 256, 256]	6,147
InstanceNorm2d: 2-19	[1, 3, 256, 256]	--
<hr/>		
<hr/>		

Total params: 54,409,603
Trainable params: 54,409,603
Non-trainable params: 0
Total mult-adds (G): 18.14

```
=====
=====
Input size (MB): 0.79
Forward/backward pass size (MB): 33.72
Params size (MB): 217.64
Estimated Total Size (MB): 252.15
=====
```

3.8 Data preparation

After the model is initialized, we then create a dataloader to sample the training data. In this paper, to sample the training data, you have to sequentially perform the following steps :

1. Randomly sample the data from the training set
2. Resizing both input and target to 286×286 .
3. Randomly cropping back both images to size 256×256 .
4. Random mirroring the images

TODO15: Implement a dataloader based on the description above. You are allowed to use the function in `torchvision.transforms` (<https://pytorch.org/vision/main/transforms.html>).

```
[ ]: # TODO15 implement a dataloader
from torch.utils.data import DataLoader, Dataset
from torchvision import transforms

transformer = transforms.Compose([
    transforms.Resize(286, antialias=True),
    transforms.RandomCrop(256),
    transforms.RandomHorizontalFlip(),
])

class FacadeDataset(Dataset):
    def __init__(self, x, y, transformer):
        self.x = x.astype(np.float32)
        self.y = y.astype(np.float32)
        self.transformer = transformer

    def __len__(self):
        return self.x.shape[0]

    def __getitem__(self, idx):
        x = torch.tensor(self.x[idx])
        y = torch.tensor(self.y[idx])
        inp = torch.cat([x, y], 0)
        inp = self.transformer(inp)
        x, y = torch.split(inp, 3, 0)
```

```

    return x, y

facade_train_dataloader = DataLoader(FacadeDataset(trainX, trainY, transformer), batch_size=1, shuffle=True, pin_memory=True)
facade_val_dataloader = DataLoader(FacadeDataset(valX, valY, transformer), batch_size=1, shuffle=False, pin_memory=True)

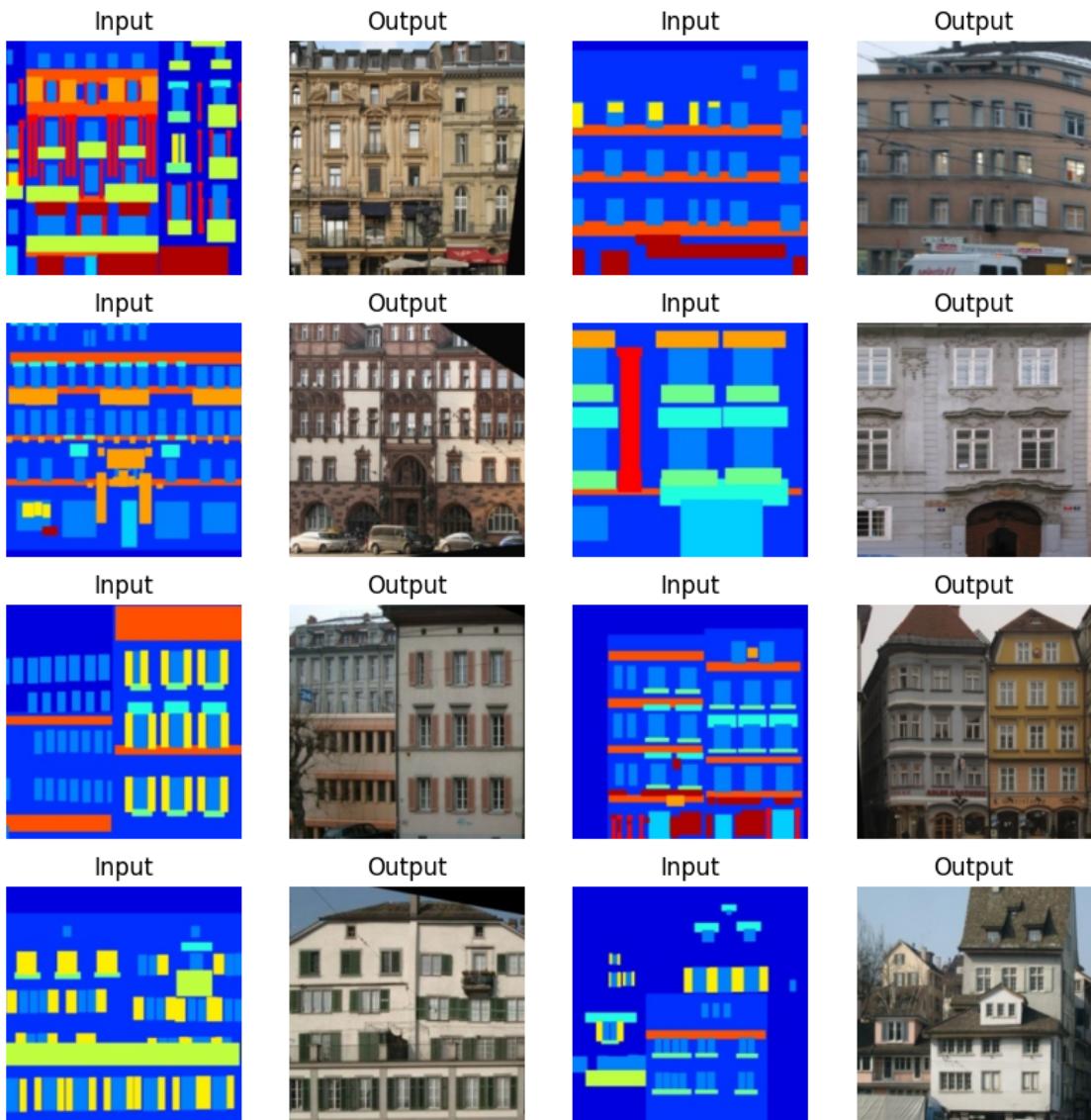
```

3.8.1 Dataloader verification

TODO16: Show that the implemented dataloader is working as intended. For instance, are both input and output are flipped and cropped correctly? To obtain a full score, you have to show at least eight data points.

```
[ ]: def to_image(x):
    x = x.numpy()
    return (0.5 * x.transpose((1, 2, 0)) + 0.5)[..., ::-1]

plt.figure(figsize=(10, 10))
for i in range(8):
    x, y = next(iter(facade_train_dataloader))
    plt.subplot(4, 4, 2*i + 1)
    plt.imshow(to_image(x[0]), cmap="gray")
    plt.title('Input')
    plt.axis('off')
    plt.subplot(4, 4, 2*i + 2)
    plt.imshow(to_image(y[0]), cmap="gray")
    plt.title('Output')
    plt.axis('off')
plt.show()
```



3.9 Parameter Initialization

Model hyperparameters and optimizers have already been prepared.

```
[ ]: import torch.optim as optim
from tqdm import tqdm
lr = 2e-4
LAMBDA = 100
BATCH_SIZE = 1
G_optimizer = optim.Adam(generator.parameters(), lr=lr, betas = (0.5, 0.999))
D_optimizer = optim.Adam(discriminator.parameters(), lr=lr, betas = (0.5, 0.
˓→999))
```

3.10 Training loop

The training process has the following specific requirements:

- The objective is divided by 2 while optimizing D , which slows down the rate at which D learns relative to G .
- This paper trains the generator G to maximize $\log D(x, G(x, z))$ instead of minimizing $\log(1 - D(x, G(x, z)))$ as the latter term does not provide sufficient gradient.

TODO17: Sample the data using the dataloader.

TODO18: Calculate the discriminator loss and update the discriminator.

- During the update, the loss term $\log(1 - D(x, G(x, z)))$ contains both generator and discriminator. However, we only want to update the discriminator. Therefore, you have to detach the input from the generator graph first. Read <https://pytorch.org/docs/stable/generated/torch.Tensor.detach.html> for additional detail.

TODO19: Calculate the generator loss and update the generator.

HINT

Hint 1: If you are struggling with this part, you could also read the PyTorch DCGAN tutorial as a guideline (https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html).

Hint 2: You could remove the L1 loss while debugging since the generator G could still generate the synthetic image even if the L1 loss is removed, though at the cost of increasing image artifacts.

Note

The training schedule in this homework is only one eighth of the original schedule. It is expected that the generated image quality is worse than the one shown in the paper. Nevertheless, the generated facade should still resemble an actual one.

```
[ ]: losses = {'D' : [None], 'G' : [None]}

L1_loss = nn.L1Loss()

for i in tqdm(range(20001)):
    #TODO17 sample the data from the dataloader
    x, y = next(iter(facade_train_dataloader))
    x, y = x.to(device), y.to(device)

    #TODO18 calculate the discriminator loss and update the discriminator
    D_optimizer.zero_grad()

    Gx = generator(x)
    D_Gx = discriminator(x, Gx.detach())
    D_y = discriminator(x, y)

    L_real = -torch.log(D_y + 1e-8).mean()
    L_fake = -torch.log(1 - D_Gx + 1e-8).mean()
    D_loss = (L_real + L_fake) * 0.5
    losses['D'].append(D_loss.item())

    D_loss.backward()
```

```

D_optimizer.step()

#TODO19 calculate the generator loss and update the generator
G_optimizer.zero_grad()

D_Gx = discriminator(x, Gx).detach()
L_gen = -torch.log(D_Gx + 1e-8).mean()

L1 = L1_loss(Gx, y)

G_loss = L_gen + LAMBDA * L1
losses['G'].append(G_loss.item())

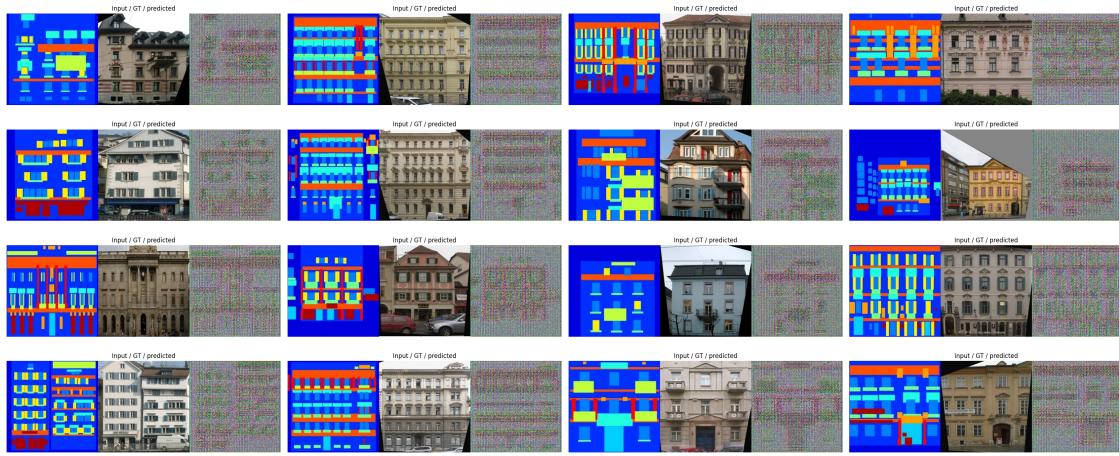
G_loss.backward()
G_optimizer.step()

# Output visualization : If your reimplementation is correct, the generated
# images should start resembling a facade after 2,500 iterations
if(i % 2500 == 0):
    with torch.no_grad():
        print(losses['D'][-1], losses['G'][-1])
        plt.figure(figsize = (40, 16))
        gs1 = gridspec.GridSpec(4, 4)
        gs1.update(wspace=0.025)

        sampleX_vis = 0.5 * valX[:16] [:, ::-1, :, :] + 0.5
        sampleY = 0.5 * valY[:16] [:, ::-1, :, :] + 0.5
        sampleX = torch.tensor(valX[:16]).to(device)
        pred_val = 0.5 * generator(sampleX).cpu().detach().numpy() [:, ::-1, :, :]
        + 0.5
        vis = np.concatenate([sampleX_vis, sampleY, pred_val], axis = 3)
        for i in range(vis.shape[0]):
            ax1 = plt.subplot(gs1[i])
            plt.title('Input / GT / predicted')
            plt.axis('off')
            plt.imshow( vis[i].transpose(1, 2, 0) )
        plt.show()

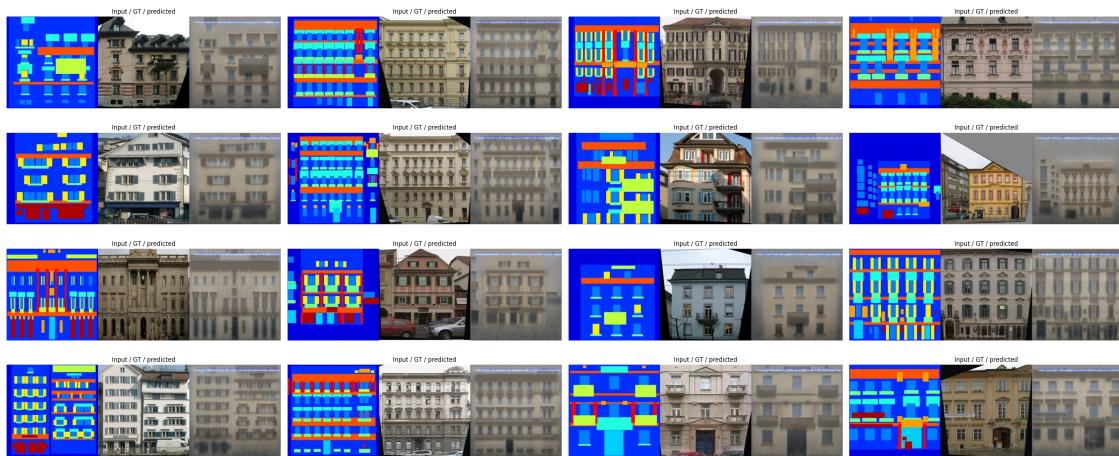
```

0% | 0/20001 [00:00<?, ?it/s]
0.7486006617546082 64.27430725097656



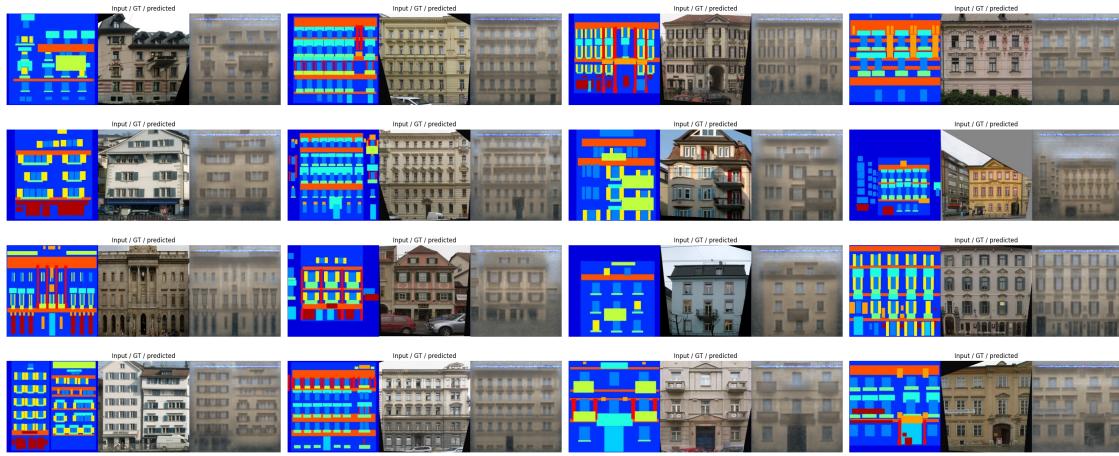
12% | 2500/20001 [01:55<13:01, 22.38it/s]

0.6393114328384399 23.22098159790039



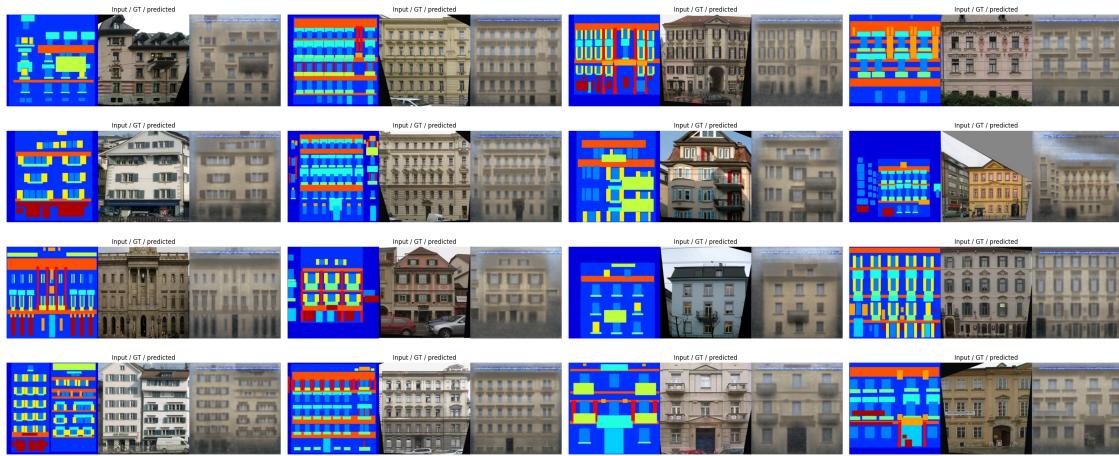
25% | 4999/20001 [03:51<11:06, 22.52it/s]

0.6354001760482788 38.60881805419922



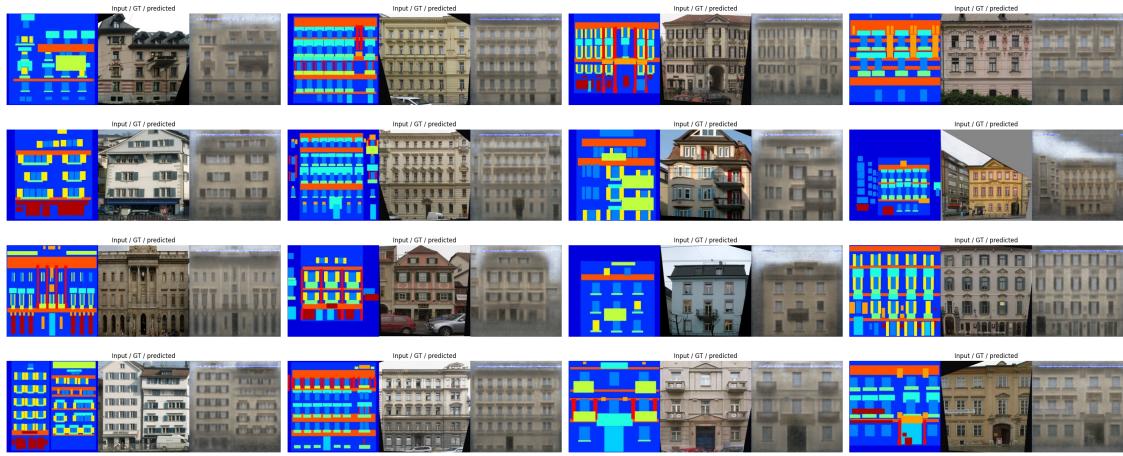
37% | 7498/20001 [05:46<09:16, 22.48it/s]

0.6362251043319702 37.79804611206055



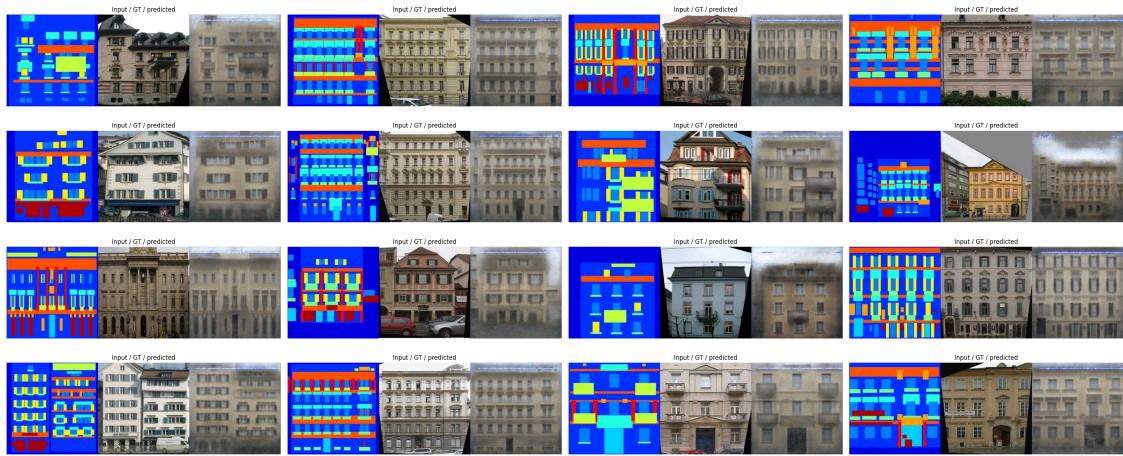
50% | 10000/20001 [07:41<07:24, 22.50it/s]

0.6313419342041016 25.40113067626953



62% | 12499/20001 [09:36<05:33, 22.50it/s]

0.6412760019302368 37.93440246582031



75% | 14998/20001 [11:30<03:41, 22.58it/s]

0.6358267068862915 24.64268684387207



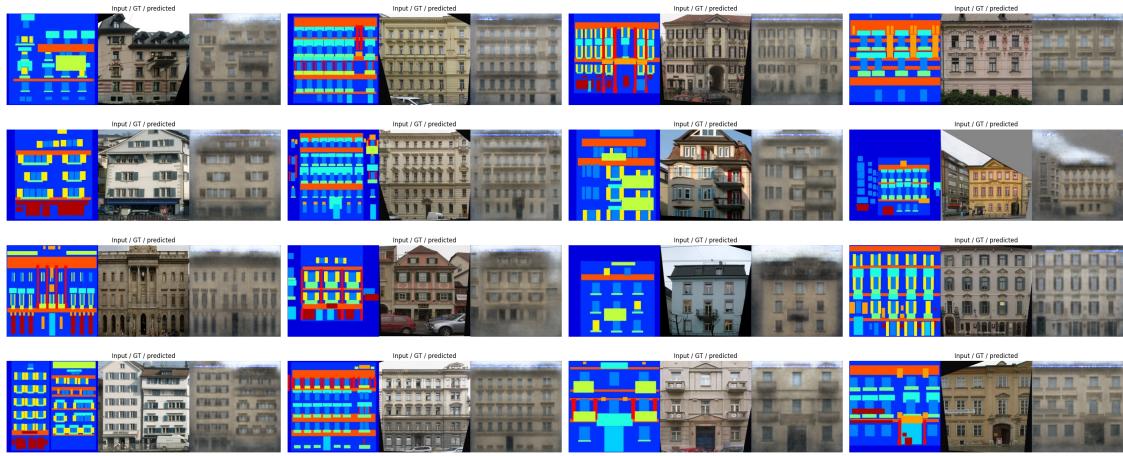
87% | 17500/20001 [13:26<01:51, 22.47it/s]

0.6358501315116882 27.690303802490234



100% | 19999/20001 [15:21<00:00, 22.61it/s]

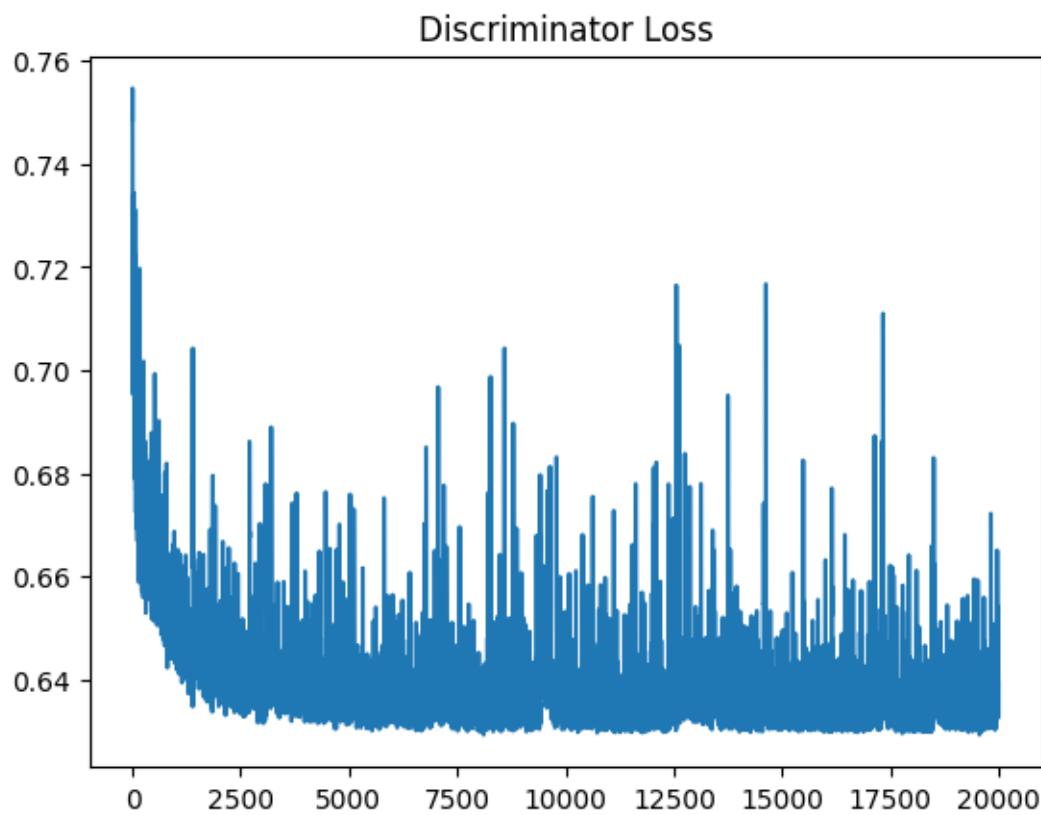
0.6393846273422241 39.5377197265625

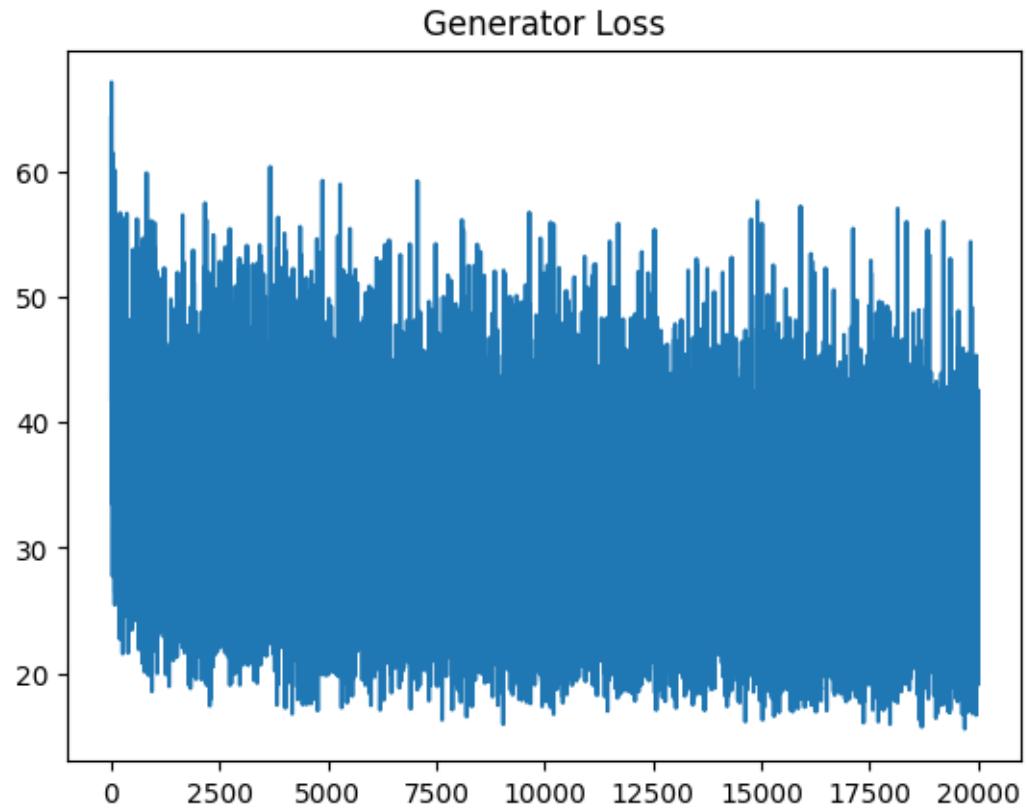


100% | 20001/20001 [15:25<00:00, 21.61it/s]

```
[ ]: plt.title('Discriminator Loss')
plt.plot(losses['D'])
plt.show()

plt.title('Generator Loss')
plt.plot(losses['G'])
plt.show()
```





3.11 (Optional)

Combine the WGAN-GP loss with the pix2pix objective.

```
[ ]: pass
```