HW 6 : 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.

Highly important -> you should avoid getting K80 GPU in your colab notebook by repeatedly factory resetting the notebook. The K80 is roughly 2x slower than Tesla T4 GPU.

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

GPU test

```
!nvidia-smi
```

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.

Algorithm 1 WGAN with gradient penalty. We use default values of $\lambda = 10$, $n_{\text{critic}} = 5$, $\alpha = 0.0001$, $\beta_1 = 0$, $\beta_2 = 0.9$.

Require: The gradient penalty coefficient λ, the number of critic iterations per generator iteration n_{critic}, the batch size m, Adam hyperparameters α, β_1, β_2.
Require: initial critic parameters w_0, initial generator parameters θ_0.

1: **while** θ has not converged **do**
2: **for** $t = 1, ..., n_{\text{critic}}$ **do**
3: **for** $i = 1, ..., m$ **do**
4: Sample real data $\boldsymbol{x} \sim \mathbb{P}_r$, latent variable $\boldsymbol{z} \sim p(\boldsymbol{z})$, a random number $\epsilon \sim U[0, 1]$.
5: $\tilde{\boldsymbol{x}} \leftarrow G_\theta(\boldsymbol{z})$
6: $\hat{\boldsymbol{x}} \leftarrow \epsilon \boldsymbol{x} + (1 - \epsilon)\tilde{\boldsymbol{x}}$
7: $L^{(i)} \leftarrow D_w(\tilde{\boldsymbol{x}}) - D_w(\boldsymbol{x}) + \lambda(\|\nabla_{\hat{\boldsymbol{x}}} D_w(\hat{\boldsymbol{x}})\|_2 - 1)^2$
8: **end for**
9: $w \leftarrow \text{Adam}(\nabla_w \frac{1}{m}\sum_{i=1}^m L^{(i)}, w, \alpha, \beta_1, \beta_2)$
10: **end for**
11: Sample a batch of latent variables $\{\boldsymbol{z}^{(i)}\}_{i=1}^m \sim p(\boldsymbol{z})$.
12: $\theta \leftarrow \text{Adam}(\nabla_\theta \frac{1}{m}\sum_{i=1}^m -D_w(G_\theta(\boldsymbol{z})), \theta, \alpha, \beta_1, \beta_2)$
13: **end while**

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.

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)
```

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()
```

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.

Discriminator (D(x))			
	Kernel size	Resample	Output shape
ConvBlock	5×5	Down	$128 \times 14 \times 14$
ConvBlock	5×5	Down	$256 \times 7 \times 7$
ConvBlock	5×5	Down	$512 \times 4 \times 4$
Linear	-	-	1

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.

Generator $(G(z))$			
	Kernel size	Resample	Output shape
z	-	-	128
Linear	-	-	$512 \times 4 \times 4$
ConvBlock	5×5	Up	$256 \times 8 \times 8$
ConvBlock	5×5	Up	$128 \times 16 \times 16$
Conv, Sigmoid	5×5	Up	$1 \times 32 \times 32$
-	-	Down	$1 \times 28 \times 28$

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().__init__()
        self.conv1 = nn.Conv2d(in_channels = 1, out_channels = 128, kernel_size=5, padding=2, stride = 2)
        self.conv2 = nn.Conv2d(in_channels = 128, out_channels = 256, kernel_size=5, padding=2, stride = 2)
        self.conv3 = nn.Conv2d(in_channels = 256, out_channels = 512, kernel_size=5, padding=2, stride = 2)
        self.linear = nn.Linear(512*4*4, 1)

    def forward(self, x):
        x = self.conv1(x)
        x = F.leaky_relu(x, negative_slope=0.1)
        x = self.conv2(x)
        x = F.leaky_relu(x, negative_slope=0.1)
        x = self.conv3(x)
        x = F.leaky_relu(x, negative_slope=0.1)
        x = x.view(-1, 512*4*4)
        x = self.linear(x)
        return x

class Generator(nn.Module):
    ##TODO2 implement the generator (actor)
    def __init__(self):
        super().__init__()
        self.linear = nn.Linear(128, 512*4*4)
        self.conv1 = nn.ConvTranspose2d(in_channels=512, out_channels=256, kernel_size=5, stride=2, padding=2, output_padding=1)
        self.conv2 = nn.ConvTranspose2d(in_channels=256, out_channels=128, kernel_size=5, stride=2, padding=2, output_padding=1)
        self.conv3 = nn.ConvTranspose2d(in_channels=128, out_channels=1, kernel_size=5, stride=2, padding=2, output_padding=1)
        self.upsample = nn.Upsample(size=(28, 28))

    def forward(self, x):
        # print(x.shape)
        x = self.linear(x)
        x = x.view(-1, 512, 4, 4)
        x = self.conv1(x)
        x = F.relu(x)
        x = self.conv2(x)
        x = F.relu(x)
        x = self.conv3(x)
        x = torch.sigmoid(x)
        x = self.upsample(x)
        return x

discriminator = Discriminator().cuda()
generator = Generator().cuda()
```

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.

```
[ ]
!pip install torchinfo
```

```
[ ]
print('The input shape of the generator is BATCH_SIZE*1')
print('The output shape of the generator is BATCH_SIZE*1*28*28')

print('The input shape of the discriminator is BATCH_SIZE*1*28*28')
print('The output shape of the discriminator is BATCH_SIZE*1')
```

```
[ ]
from torchinfo import summary
batch_size = 64
summary(discriminator, input_size=(batch_size, 1, 28, 28))
```

```
[ ]
batch_size = 64
summary(generator, input_size=(batch_size, 128))
```

Parameter Inititalization

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)).cuda()
def schedule(i):
  lr = 1e-4
  if(i > 2500): lr *= 0.1
  return lr
losses = {'D' : [None], 'G' : [None]}

## TODO4 initialize missing hyperparameter and optimser
alpha = 0.0001
beta1 = 0
beta2 = 0.9
G_optimizer = torch.optim.Adam(generator.parameters(), lr=alpha, betas=(beta1, beta2))
D_optimizer = torch.optim.Adam(discriminator.parameters(), lr=alpha, betas=(beta1, beta2))
GP_lambda = 10
n_critic = 5
```

Data preparation

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

```python
# TODO5 implement dataloader
from torch.utils.data import Dataset
from torch.utils.data import DataLoader
import numpy as np
class MinistDataset(Dataset):
    def __init__(self, x,):
        self.x = x.astype(float)

    def __getitem__(self, index):
        x = self.x[index] # Retrieve data
        # np.random.seed(seed=index)
        z = np.random.normal(loc=0.0, scale=1.0, size=128)
        e = np.random.uniform(low=0.0, high=1.0)
        return x, z, e

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

train_dataset = MinistDataset(trainX)
train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True, pin_memory=True, drop_last=True)
```

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 $\lambda(||(\nabla_{\hat{\mathbf{x}}} D_w(\hat{\mathbf{x}}))||_2 - 1)^2$, 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).

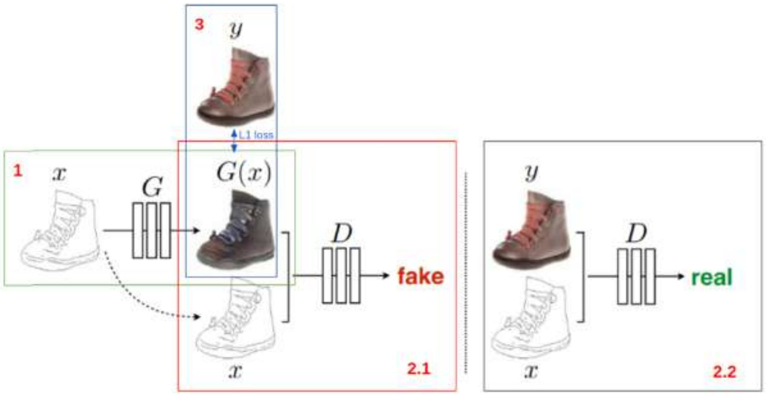
TODO10: Calculate and update the generator loss (Line 11-12).

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

```python
from tqdm import tqdm
model = torch.FloatTensor([1]).cuda()
mode2 = torch.FloatTensor([-1]).cuda()
for i in tqdm(range(NUM_ITERATION)):
  ## TODO6 update learning rate
  lr = schedule(i)
  for idx in range(len(G_optimizer.param_groups)):
    G_optimizer.param_groups[idx]['lr'] = lr
  for idx in range(len(D_optimizer.param_groups)):
    D_optimizer.param_groups[idx]['lr'] = lr

  for t in range(n_critic):
    ## TODO7 line 4: sample data from dataloader
    dataloader_iterator = iter(train_loader)
    x, z, e = next(dataloader_iterator)
    ## TODO8 line5-7 : calculate discriminator loss
    for p in discriminator.parameters():
        p.requires_grad = True
    x, z, e = x.cuda(), z.cuda(), e.cuda()
    x = x.type(torch.cuda.FloatTensor)
    x = torch.autograd.Variable(x)
    discriminator.zero_grad()
    real_out = discriminator(x).mean()


    z = z.type(torch.cuda.FloatTensor)
    z = torch.autograd.Variable(z, volatile=True)
    fake = torch.autograd.Variable(generator(z))
    fake_out = discriminator(fake).mean()


    e = e.view(BATCH_SIZE, -1).type(torch.cuda.FloatTensor)
    e = e[:, None, None].expand(x.shape)
    x_hat = e*x + (1-e)*fake
    interpolates = torch.autograd.Variable(x_hat, requires_grad=True)
    disc_interpolates = discriminator(interpolates)
    gradients = torch.autograd.grad(outputs=disc_interpolates, inputs=interpolates,
                                    grad_outputs=torch.ones(disc_interpolates.size()).cuda(),
                                    create_graph=True, retain_graph=True, only_inputs=True)[0]
    gradient_penalty = ((gradients.norm(2, dim=1) - 1) ** 2).mean() * GP_lambda
    loss = fake_out - real_out + gradient_penalty
    ## TODO9 : line 9 update discriminator loss
    loss.backward()
    losses['D'].append(loss.mean().item())
    G_optimizer.step()
  ## TODO10 : line 11-12 calculate and update the generator loss

  for p in discriminator.parameters():
    p.requires_grad = False
  generator.zero_grad()

  dataloader_iterator = iter(train_loader)
  x, z, e = next(dataloader_iterator)
  x, z, e = x.cuda(), z.cuda(), e.cuda()

  x = x.type(torch.cuda.FloatTensor)
  x = torch.autograd.Variable(x)

  z = z.type(torch.cuda.FloatTensor)
  z = torch.autograd.Variable(z)

  fake = generator(z)
  DoG = -discriminator(fake).mean()
  DoG.backward()
  D_optimizer.step()

  losses['G'].append(DoG.item())


  # 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()
```

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 genereator G^* base on the objective function : $G^* = argmin_G max_D L_{cGAN}(G, D) + \lambda L_1(G)$

- The term $argmin_G max_D L_{cGAN}(G, D)$ 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 fuction 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.

Get dataset

[1]
```
!wget http://efrosgans.eecs.berkeley.edu/pix2pix/datasets/facades.tar.gz
!tar -xzf facades.tar.gz
```

```
--2022-04-02 16:20:55--  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  2.71MB/s    in 33s

2022-04-02 16:21:28 (896 KB/s) - 'facades.tar.gz' saved [30168306/30168306]
```

Import library

[2]
```
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
```

Setting up facade dataset

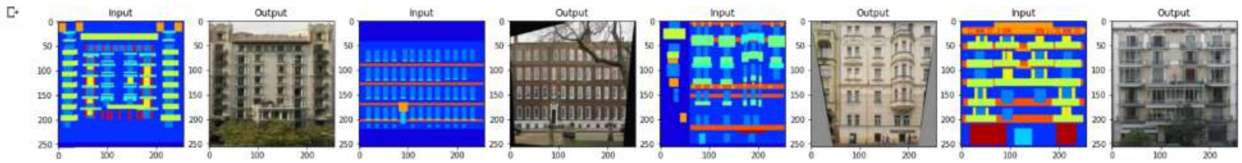
The dataset chosen for this exercise is the CMP Facade Database which is a pair of facade images and its segmented component stored in RGB value. The objective of this exercise is to generate a facade given its simplified segmented component. Both input and output is a 256 x 256 RGB image normalized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

[3]
```
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)
```

Dataset Visualization

[4]
```
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()
```

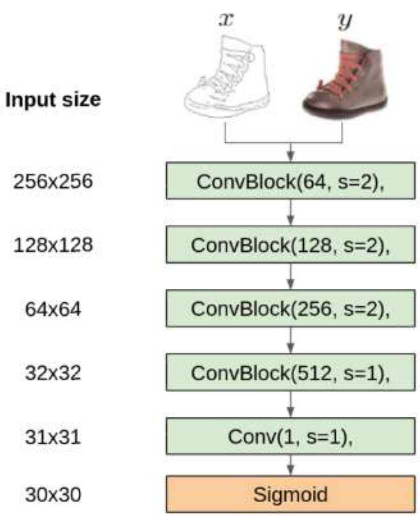


Note

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

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.

[23]
```
class Discriminator(nn.Module):
  #TODO11 implement the discriminator network
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(in_channels = 6, out_channels = 64, kernel_size=4, padding=1, stride = 2, padding_mode="reflect")
        self.conv2 = nn.Conv2d(in_channels = 64, out_channels = 128, kernel_size=4, padding=1, stride = 2, padding_mode="reflect")
        self.conv3 = nn.Conv2d(in_channels = 128, out_channels = 256, kernel_size=4, padding=1, stride = 2, padding_mode="reflect")
        self.conv4 = nn.Conv2d(in_channels = 256, out_channels = 512, kernel_size=4, padding=1, stride = 1, padding_mode="reflect")
        self.conv5 = nn.Conv2d(in_channels = 512, out_channels = 1, kernel_size=4, padding=1, stride = 1, padding_mode="reflect")

    def forward(self, x, y):

        x = torch.cat((x, y), dim=1)

        x = self.conv1(x)
        x = F.leaky_relu(x, negative_slope=0.2)

        x = self.conv2(x)
        x = F.instance_norm(x)
        x = F.leaky_relu(x, negative_slope=0.2)

        x = self.conv3(x)
        x = F.instance_norm(x)
        x = F.leaky_relu(x, negative_slope=0.2)

        x = self.conv4(x)
        x = F.instance_norm(x)
        x = F.leaky_relu(x, negative_slope=0.2)

        x = self.conv5(x)
        x = torch.sigmoid(x)

        return x

discriminator = Discriminator().cuda()
```

[24]
```
print(f'The input shape is BATCH_SIZE*6*256*256')
print(f'The output shape is BATCH_SIZE*1*30*30')
```

```
The input shape is BATCH_SIZE*6*256*256
The output shape is BATCH_SIZE*1*30*30
```

[25]
```
!pip install torchinfo
```

```
Requirement already satisfied: torchinfo in /usr/local/lib/python3.7/dist-packages (1.6.5)
```

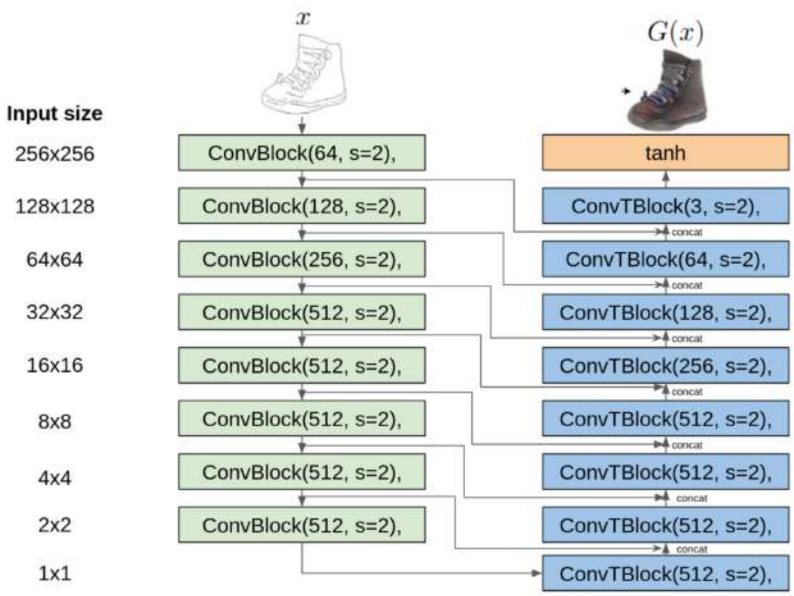
[26]
```
#TODO12 verify the discriminator
from torchinfo import summary
batch_size = 1
summary(discriminator, [(batch_size, 3, 256, 256), (batch_size, 3, 256, 256)])
```

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
Discriminator                            --                        --
```

```
├─Conv2d: 1-1                                  [1, 64, 128, 128]         6,208
├─Conv2d: 1-2                                  [1, 128, 64, 64]          131,200
├─Conv2d: 1-3                                  [1, 256, 32, 32]          524,544
├─Conv2d: 1-4                                  [1, 512, 31, 31]          2,097,664
├─Conv2d: 1-5                                  [1, 1, 30, 30]            8,193
==========================================================================================
Total params: 2,767,809
Trainable params: 2,767,809
Non-trainable params: 0
Total mult-adds (G): 3.20
==========================================================================================
Input size (MB): 1.57
Forward/backward pass size (MB): 18.62
Params size (MB): 11.07
Estimated Total Size (MB): 31.27
==========================================================================================
```

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.

```python
#HINT : you could also put multiple layers in a single list using nn.ModuleList
class Generator(nn.Module):
  #TODO13 implement the generator network
  def __init__(self):
    super().__init__()
    in_channels_list = [3, 64, 128, 256, 512, 512, 512, 512]
    out_channels_list = [64, 128, 256, 512, 512, 512, 512, 512]
    self.convs = nn.ModuleList([nn.Conv2d(in_channels = in_channels_list[i], out_channels = out_channels_list[i], kernel_size=4, padding=1, stride=2, padding_mode="reflect") for i in range(len(in_c

    in_channels_list2 = [512, 1024, 1024, 1024, 1024, 512, 256, 128]
    out_channels_list2 = [512, 512, 512, 512, 256, 128, 64, 3]
    self.convTs = nn.ModuleList([nn.ConvTranspose2d(in_channels=in_channels_list2[i], out_channels=out_channels_list2[i], kernel_size=4, stride=2, padding=1) for i in range(len(out_channels_list2))

  def forward(self, x):
    send_to_decoder = []

    # Encoder
    for i in range(len(self.convs)):
      x = self.convs[i](x)
      if i > 0 and i < len(self.convs) - 1:
        x = F.instance_norm(x)
      x = F.leaky_relu(x, negative_slope=0.2)
      if i < len(self.convs) - 1:
        send_to_decoder.append(x)

    send_to_decoder = send_to_decoder[::-1]

    # Decoder
    for i in range(len(self.convTs)):
      if i > 0:
        x = torch.cat([x, send_to_decoder[i-1]], dim=1)
      x = self.convTs[i](x)

      if i < len(self.convTs) - 1:
        x = F.instance_norm(x)
        x = F.dropout(x, p=0.5)
        x = F.relu(x)
    x = torch.tanh(x)
    return x

generator = Generator().cuda()
```

[28]
```python
print(f'The input shape is BATCH_SIZE*3*256*256')
print(f'The output shape is BATCH_SIZE*3*256*256')
```

```
The input shape is BATCH_SIZE*3*256*256
The output shape is BATCH_SIZE*3*256*256
```

[29]
```python
#TODO14 verify the generator
generator.eval()
batch_size = 1
summary(generator, input_size=(batch_size, 3, 256, 256))
```

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
Generator                                --                        --
├─ModuleList: 1-1                        --                        --
├─ModuleList: 1-2                        --                        --
├─ModuleList: 1-1                        --                        --
│    └─Conv2d: 2-1                       [1, 64, 128, 128]         3,136
│    └─Conv2d: 2-2                       [1, 128, 64, 64]          131,200
│    └─Conv2d: 2-3                       [1, 256, 32, 32]          524,544
│    └─Conv2d: 2-4                       [1, 512, 16, 16]          2,097,664
│    └─Conv2d: 2-5                       [1, 512, 8, 8]            4,194,816
│    └─Conv2d: 2-6                       [1, 512, 4, 4]            4,194,816
│    └─Conv2d: 2-7                       [1, 512, 2, 2]            4,194,816
│    └─Conv2d: 2-8                       [1, 512, 1, 1]            4,194,816
├─ModuleList: 1-2                        --                        --
│    └─ConvTranspose2d: 2-9              [1, 512, 2, 2]            4,194,816
│    └─ConvTranspose2d: 2-10             [1, 512, 4, 4]            8,389,120
│    └─ConvTranspose2d: 2-11             [1, 512, 8, 8]            8,389,120
│    └─ConvTranspose2d: 2-12             [1, 512, 16, 16]          8,389,120
│    └─ConvTranspose2d: 2-13             [1, 256, 32, 32]          4,194,560
│    └─ConvTranspose2d: 2-14             [1, 128, 64, 64]          1,048,704
│    └─ConvTranspose2d: 2-15             [1, 64, 128, 128]         262,208
│    └─ConvTranspose2d: 2-16             [1, 3, 256, 256]          6,147
==========================================================================================
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
==========================================================================================
```

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).

[30]

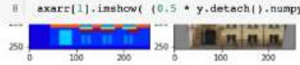
```python
# TODO15 implement a dataloader
from torch.utils.data import Dataset
from torch.utils.data import DataLoader
import torchvision.transforms as transforms
import numpy as np
from PIL import Image
import random
class FacadeDataset(Dataset):
    def __init__(self, x, y):
        self.x = x.astype(float)
        self.y = y.astype(float)
        self.transform = transforms.Compose([transforms.Resize(size=(286, 286)),\
                                             transforms.RandomCrop(size=(256, 256)),\
                                             transforms.RandomHorizontalFlip(p=0.5)])

    def __getitem__(self, index):
        x = self.x[index] # Retrieve data
        y = self.y[index]
        x = torch.from_numpy(x).type(torch.float32)
        y = torch.from_numpy(y).type(torch.float32)
        x_and_y = torch.cat((x, y), dim=0)
        x_and_y = self.transform(x_and_y)
        x = x_and_y[:3, :, :]
        y = x_and_y[3:, :, :]
        return x, y

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

train_dataset = FacadeDataset(trainX, trainY)
val_dataset = FacadeDataset(valX, valY)
train_loader = DataLoader(train_dataset, batch_size=1, shuffle=True, pin_memory=True, drop_last=True)
val_loader = DataLoader(val_dataset, batch_size=1, shuffle=True, pin_memory=True, drop_last=True)
```

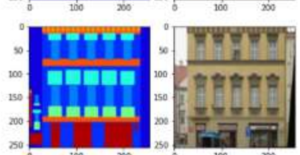
Dataloader verification

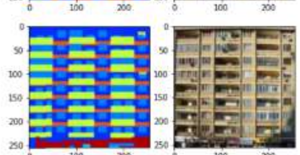
TODO16: Show that the implemented dataloader is working as intended. For instance, are both input and output are flipped and cropped correctly? To obtain a full score, you have to show at least eight data points.

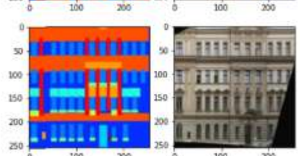
[31]

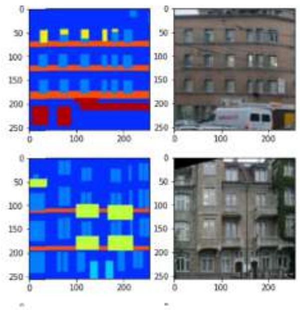
```python
#TODO16 show that the dataloader is working properly
for i in range(10):
  dataloader_iterator = iter(train_loader)
  x, y = next(dataloader_iterator)
  x, y = x[0], y[0]
  f, axarr = plt.subplots(1,2)
  axarr[0].imshow( (0.5 * x.detach().numpy().transpose((1, 2, 0)) + 0.5)[..., ::-1] , cmap = 'gray' )
  axarr[1].imshow( (0.5 * y.detach().numpy().transpose((1, 2, 0)) + 0.5)[..., ::-1], cmap = 'gray' )
```











Parameter Initialization

Model hyperparameters and optimizers have already been prepared.

[32]
```python
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))
```

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).

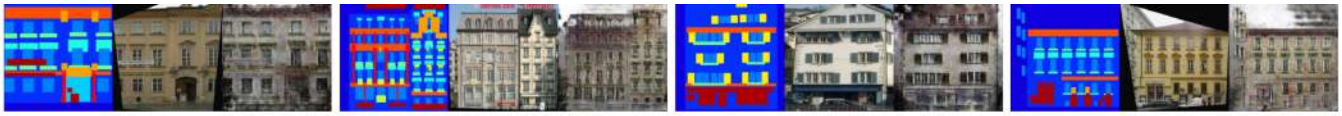
Hint 2: You could remove the L1 loss while debugging since the generator G could still generate the synthetic image even if the L1 loss is removed, though at the cost of increasing image artifacts.

Note

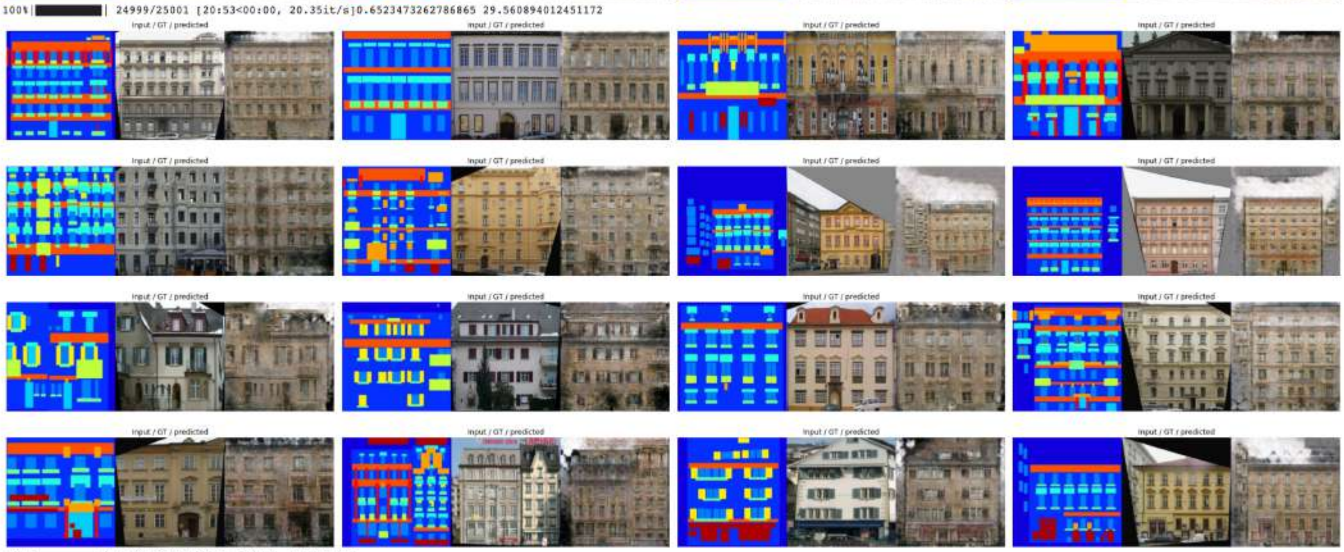
The training schedule in this homework is only one eighth of the orignal schedule. It is expected that the generated image quality is worse than the one shown in the paper. Nevertheless, the generated facade should still resemble an actual one.

[33]
```python
losses = {'D' : [None], 'G' : [None]}
l1_loss = nn.L1Loss()
adversarial_loss = nn.BCELoss()
for i in tqdm(range(25001)):

  #TODO17 sample the data from the dataloader
  dataloader_iterator = iter(train_loader)
  x, y = next(dataloader_iterator)
  x, y = x.cuda(), y.cuda()
  generator.train()
  #TODO18 calculate the discriminator loss and update the discriminator
  discriminator.zero_grad()
  y_fake = generator(x)
  D_real = discriminator(x, y)
  D_real_loss = adversarial_loss(D_real, torch.ones_like(D_real))
  D_fake = discriminator(x, y_fake.detach())
  D_fake_loss = adversarial_loss(D_fake, torch.zeros_like(D_fake))
  D_loss = ((D_real_loss + D_fake_loss) / 2)
  losses['D'].append(D_loss.item())
  D_loss.backward()
  D_optimizer.step()


  #TODO19 calculate the generator loss and update the generator
  generator.zero_grad()
  # generated_img = generator(x)
  fake_score = discriminator(x, y_fake)
  generator_loss_1 = adversarial_loss(fake_score, torch.ones_like(fake_score).cuda())
  generator_loss_2 = LAMBDA*l1_loss(y_fake, y)
  # if(i%100 == 0):
    # print(generator_loss_1, generator_loss_2)
  generator_loss = (generator_loss_1 + generator_loss_2)
  losses['G'].append(generator_loss.item())
  generator_loss.backward()
  G_optimizer.step()
  # G_optimizer.update()

  # Output visualization : If your reimplementation is correct, the generated images should start resembling a facade after 2,500 iterations
  generator.eval()
  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]).cuda()
      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()

```

Input / GT / predicted Input / GT / predicted Input / GT / predicted Input / GT / predicted

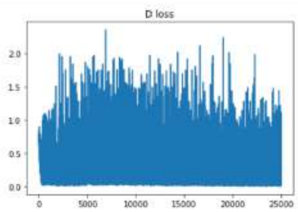


100%|██████████| 24999/25001 [20:53<00:00, 20.35it/s]0.6523473262786865 29.560894012451172

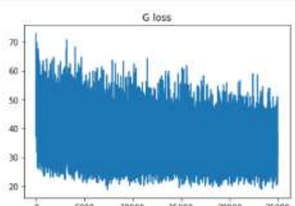


100%|██████████| 25001/25001 [20:55<00:00, 19.91it/s]

[34]
```
plt.plot(np.arange(len(losses['D'])), losses['D'])
plt.title('D loss')
plt.show()
```



[35]
```
plt.plot(np.arange(len(losses['G'])), losses['G'])
plt.title('G loss')
plt.show()
```



[36]
```
path = "/content/discriminator.pth"
torch.save(discriminator, path)
# /content/facades
path = "/content/generator.pth"
torch.save(generator, path)
```

(Optional)

Combine the WGAN-GP loss with the pix2pix objective.

[36]
```

```

```

```