Assignment 2-1
DCGAN Practice

IMVFX Autumn
November 9, 2023

Sample Code :
[IMVFX_HW2_DCGAN.ipynb - Colaboratory (google.com)]
Colab tutorial :
[IMVFX_Google_Colab_Tutorial.ipynb – Colaboratory]
DEEP LEARNING WITH PYTORCH: A 60 MINUTE BLITZ
https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html

Outline

- Introduction
- GAN
- The structure of DCGAN
- Requirements
- Reminder
- Submission
- Reference

Introduction

In this homework, you are going to use DCGAN (Deep Convolutional Generative Adversarial Networks) to generate images.

Use the dataset below to train the model:

- Anime Face dataset (21376 images): Google drive link

GAN

1. GAN is Invented by Ian Goodfellow in 2014.
2. Made of two models:

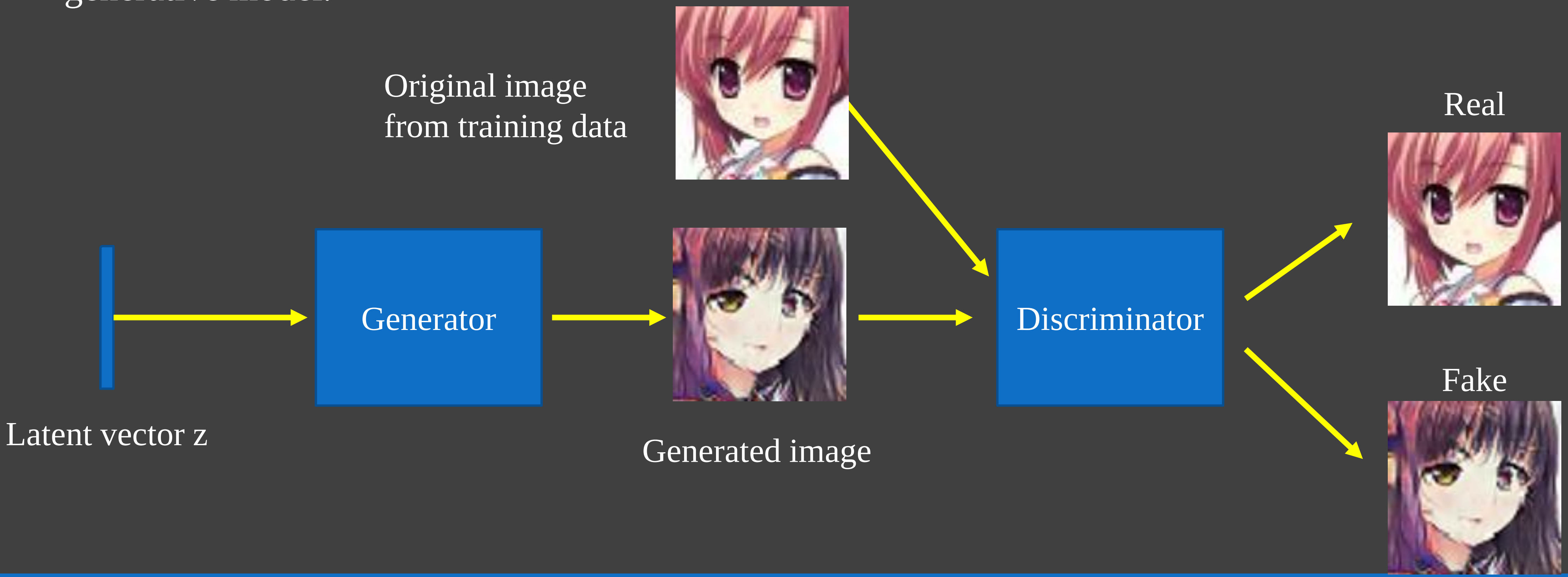
- Generator: Generate the fake data that look like training data.
- Discriminator: Look at a image and output the data is true or fake data

3. Try to achieve a equilibrium of the game by training the model.

$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z\sim p_z(z)}[\log(1 - D(G(z)))]$$

The structure of DCGAN

DCGAN introduces a CNN structure on top of the original GAN to enhance the performance of the generative model.



For implementation A.1-1

Steps for training GAN

1. Set up the parameters
2. Load the data
3. Initialize the model weight
4. Build DCGAN model (Generator, Discriminator)
5. Set up loss functions and optimizers
6. Training

For implementation A.1-1

Step 1. Set up the parameters

We need to set up the parameters for training.

You can change the parameters to improve the training result. (ex. Set the lr = 0.0005)

```
# Root directory for dataset
dataroot = "anime_face_dataset"
# 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 = 3
# 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 = 10
# Learning rate for optimizers
lr = 0.0002
# Beta1 hyperparam for Adam optimizers
beta1 = 0.5
# Number of GPUs available. Use 0 for CPU mode.
ngpu = 1
# Decide which device we want to run on
device = torch.device("cuda:0" if (torch.cuda.is_available() and ngpu > 0) else "cpu")
```

For implementation A.1-1

Step 2. Load the data

Then we can load the data by ImageFolder() and make the data loader by DataLoader().

You can use transformation functions for data augmentation.

```
def get_dataset(dataroot):
    dataset = dset.ImageFolder(root=dataroot,
                               transform=transforms.Compose([
                                   transforms.Resize(image_size),
                                   transforms.CenterCrop(image_size),
                                   transforms.ToTensor(),
                                   transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
                               ]))
    return dataset

# Create the dataset
dataset = get_dataset(os.path.join(workspace_dir, dataroot))

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

For implementation A.1-1

Step 3. Initialize the model weight

From the DCGAN paper, the authors specify that all models should be initialized by normal distribution with mean=0, stdev=0.02.
This function is applied to the models immediately after initialization.

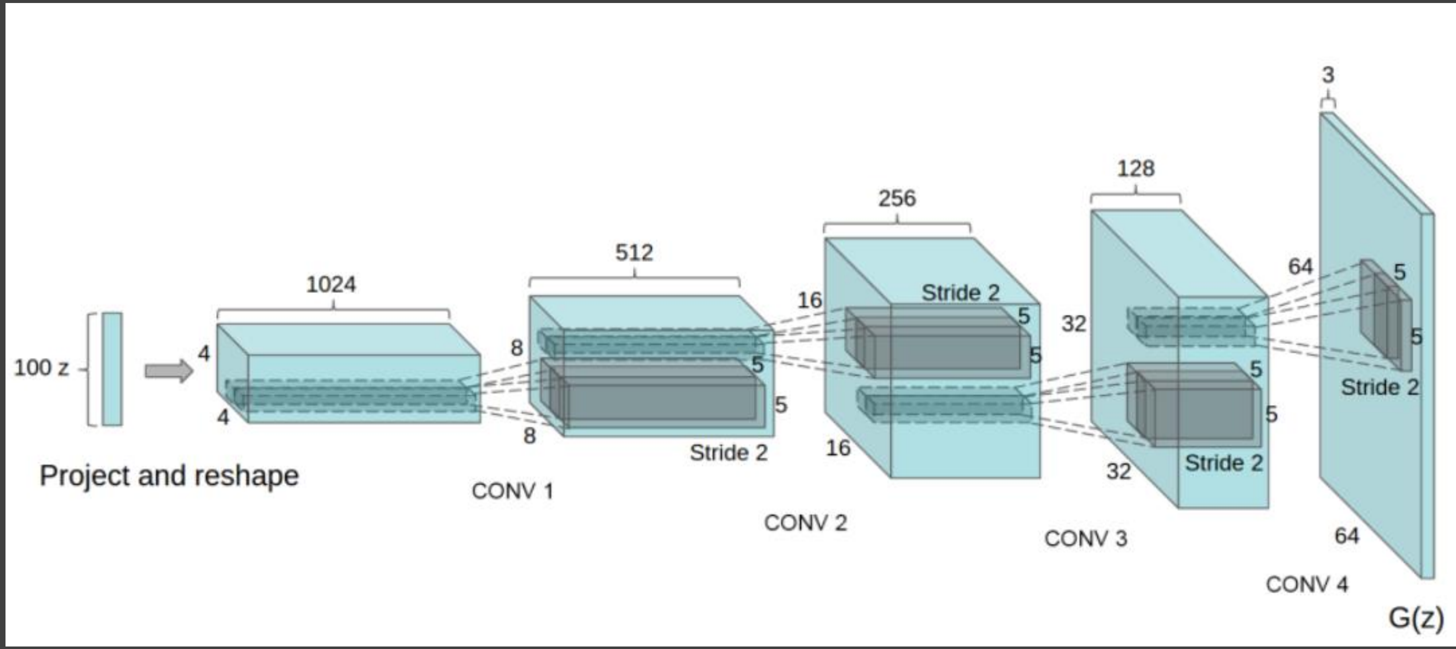
```
# custom weights initialization called on netG and netD
def weights_init(m):
    classname = m.__class__.__name__
    if classname.find('Conv') != -1:
        nn.init.normal_(m.weight.data, 0.0, 0.02)
    elif classname.find('BatchNorm') != -1:
        nn.init.normal_(m.weight.data, 1.0, 0.02)
 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

For implementation A.1-1

Step 4. Build the DCGAN model - Generator

An illustration of the generator architecture from the DCGAN paper is shown below.

The input of generator is a latent vector z and the output is a image.



For implementation A.1-1

Step 4. Build the DCGAN model - Generator

Define the Generator class that contains two functions:

- __init__():
  Initialize the layers.
  (ConvTranspose2d(), BatchNorm2d(), ReLU())
- forward():
  Forward propagate your input through the layers.

```python
# Generator Code
class Generator(nn.Module):
    """
    Input shape: (N, in_dim, 1, 1)
    Output shape: (N, nc, image_size, image_size)

    In our sample code, input/output shape are:
        Input shape: (N, 100, 1, 1)
        Output shape: (N, 3, 64, 64)
    """

    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)
```

For implementation A.1-1

Step 4. Build the DCGAN model - Generator

After define the model class, create a generator by the class.

Remember to put your generator to the device and apply the weight initialize function.

```
# 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=0.2.
netG.apply(weights_init)

# Print the model
print(netG)
```

For implementation A.1-1

Step 4. Build the DCGAN model - Discriminator

Define the Discriminator class that contain two functions:

- __init__():
  Initialize the layers.
  (Conv2d(), BatchNorm2d(), LeakyReLU())
- forward():
  Forward propagate your input through the layers.

```
class Discriminator(nn.Module):
    """
    Input shape: (N, nc, image_size, image_size)
    Output shape: (N, )

    In our sample code, input/output are:
        Input shape: (N, 3, 64, 64)
        Output shape: (N, )
    """
    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)
```

For implementation A.1-1

Step 4. Build the DCGAN model - Discriminator

Similar with generator, create a discriminator here.
Put the network to device and apply the weight initialize function.

```
# 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
#  to mean=0, stdev=0.2.
netD.apply(weights_init)

# Print the model
print(netD)
```

For implementation A.1-1

Step 5. Set up loss functions and optimizers

We use BCELoss() and Adam() here.

BCE loss function

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

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = -\left[y_n \cdot \log x_n + (1 - y_n) \cdot \log(1 - x_n)\right]$$

```
# 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))
```

You can try different optimizers here, such as SGD, Momentum, and so on.

For implementation A.1-1

Step 6. Training

Then we can start training.

GAN is hard to train, in order to get a better result, we need to apply some ganhack tricks in our training loop, include:

- Construct different mini-batches for real and fake images and adjust Generator’s objective function to maximize logD(G(z)).
- First, train discriminator: maximize log(D(x)) + log(1 - D(G(z))).
- Than train the generator: minimizing log(1−D(G(z))) -> maximize log(D(G(z))).
- Create the noise vector and the label for true and fake data.

```
# Create batch of latent vectors that we will use to visualize
# the progression of the generator
fixed_noise = torch.randn(100, nz, 1, 1, device=device)

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

For implementation A.1-1

Step 6. Training - Train the discriminator

Due to the separate mini-batch suggestion from ganhacks, we will calculate this in two steps.

- Step 1:

Construct a batch of real samples from the training set, forward pass through D, calculate the loss log(D(x)), then backward pass to compute the gradients.

```
## 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()
```

For implementation A.1-1

Step 6. Training - Train the discriminator

- Step 2:

Construct a batch of fake samples with the current generator, forward pass this batch through D, calculate the loss log(1−D(G(z))), then backward pass to accumulate the gradients. (So the gradient is computed by forwarding a batch of real and a batch of fake images)

```
## 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()
```

For implementation A.1-1

Step 6. Training - Train the discriminator

Remember to call the optimizer's step function after the backward pass to update the model's weights.

```
# Compute error of D as sum over the fake and the real batches
errD = errD_real + errD_fake
# Update D
optimizerD.step()
```

For implementation A.1-1

Step 6. Training - Train the Generator

Forward the fake data but calculate loss with real label.

And remember to call the optimizer's step function after the backward pass.

```
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()
# Update G
optimizerG.step()
```

For implementation A.1-1

Step 6. Training

Remember to save the model weight when you are training.

You can use torch.save(model, PATH), torch.save(model.state_dict(), PATH) for saving the model weight.

When you want to evaluate data or trained on pre-trained weight.

You can use torch.load(PATH), model.load_state_dict(torch.load(PATH)).

Example on pytorch.org

For implementation A.1-2, A.1-3

Plot the loss and save images

For plot loss:

Save the loss values in a list while you are training.

Then use the function in matplotlib.pyplot to plot the loss.

For save images:

Save the images by torchvision.utils.save_image() or matplotlib.pyplot.savefig().

```
torchvision.utils.save_image(f_imgs_sample,  filename,  nrow=10)
```

The fake images from your generator. Save directory The images number per row.

For implementation A.1-4

Interpolation

After finish training your model, interpolate the z vector and generate images to observe how the image changes with different z value.

First, generate latent vectors z1 and z2, then compute their interpolation by

$v = (1.0 - ratio) * z1 + ratio * z2$

Compute v at different ratio 0, (1/9), (2/9), ..., (8/9), 1, each ratio will give a different images that you need to save.

Requirements - Implementation

Implementation(30%):

A.1-1 Train a DCGAN model and generate the images. (Please use image size 64*64) (10%)

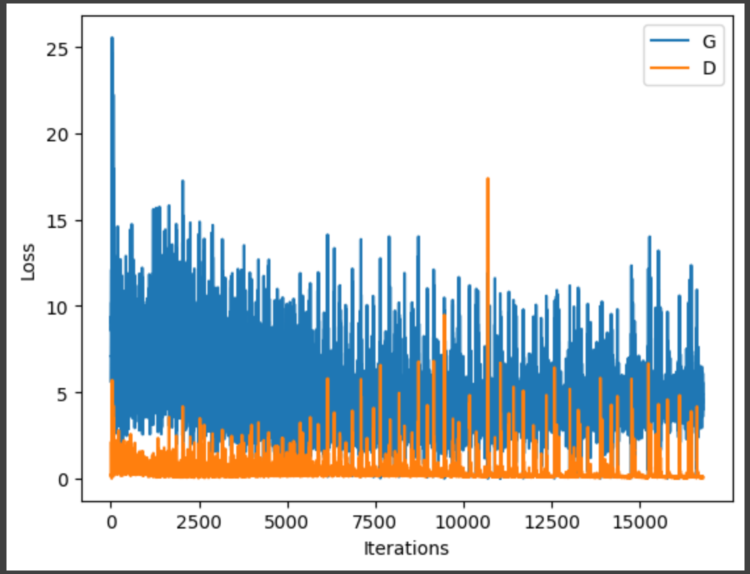
A.1-2 Plot the loss value of discriminator and generator versus training iterations. (Please upload the image to E3) (5%)

A.1-3 Store your generated images in 5*5 grid. (Please upload the image to E3) (5%)

A.1-4 Explore your latent space: Interpolate 3 pairs of z vectors and plot the generated images in 3*10 grid. (Please upload the image to E3) (10%)

Requirements - Implementation

Example for A.1-2



Example for A.1-3

Requirements - Implementation

Example for A.1-4

Requirements - Report

Report(15%): You can write in Chinese or English.

✓DCGAN

A.2-1 Please provide a brief introduction about your experiments, including details such as setting of hyperparameter, data augmentation techniques used, network structure, etc. (5%)

A.2-2 Place the generated image series from various epochs during the training process here and provide a discussion of your observations. (5%)

A.2-3 Discuss about A.1-4. Please explain how the z vector influence your images here. (5%)

Reminder

- If you refer any code from GitHub or other open source, you have to properly cite the source and comment on codes belonging to the open source. Otherwise, you will get a penalty of 20 points or more.
- You should work on all the given images.
- It takes much time to train the models, so better to start your work early.
- Feel free to modify any code provided by TA or write the code yourself.

Submission

1. Your python source code (.py or .ipynb)
2. Your report (Named as report_<your student ID>.pdf)
3. A generated 5*5 grid image (Named as result.jpg)
4. A plot of loss values (Named as loss.jpg)
5. An interpolation 3*10 grid image (Named as interpolation.jpg)

Zip all the files above to <your student ID>_hw2_1.zip and upload the zip file to E3 before the deadline.

Reference

DCGAN Implementation
https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html
https://github.com/eriklindernoren/PyTorch-GAN
https://github.com/soumith/ganhacks
https://pytorch.org/tutorials/beginner/saving_loading_models.html

Data source
https://www.kaggle.com/datasets/soumikrakshit/anime-faces

Paper of GAN & DCGAN
https://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf
https://arxiv.org/abs/1511.06434

State-of-the-art
https://paperswithcode.com/task/conditional-image-generation
https://paperswithcode.com/task/image-generation