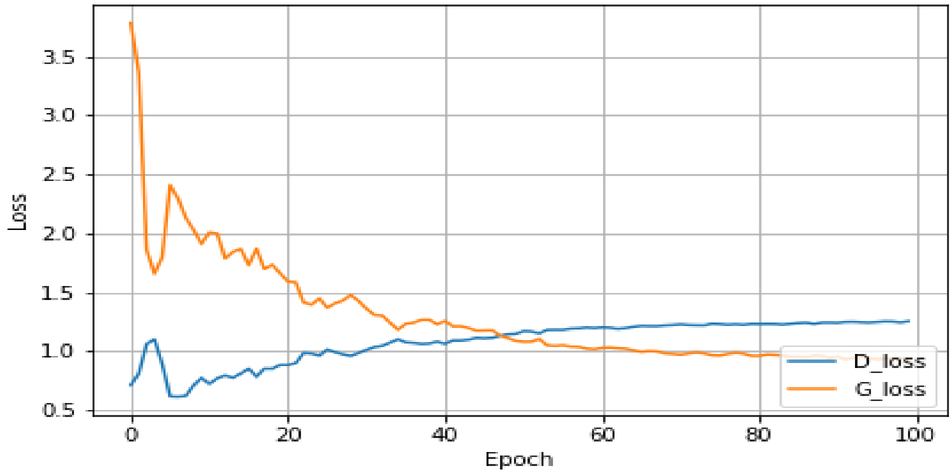
Question to be answered (Q2.4)

How do D-loss and G-loss change during training? Visualize how the D-loss and D-loss change during training and explain why.

GANs try to replicate a probability distribution. They should therefore use loss functions that reflect the distance between the distribution of the data generated by the GAN and the distribution of the real data. In the paper that introduced GANs, the generator tries to minimize the following function while the discriminator tries to maximize it during training.

As we see in the picture below the visualisation of the D-loss and the G-loss.



While the discriminator is trained, it classifies both the real data and the fake data from the generator. It penalizes itself for misclassifying a real instance as fake, or a fake instance (created by the generator) as real, by maximizing the below function.

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(x^{(i)}\right) + \log\left(1 - D\left(G\left(z^{(i)}\right)\right)\right)\right]$$

log(D(x)) refers to the probability that the generator is rightly classifying the real image,

maximizing **log(1-D(G(z)))** would help it to correctly label the fake image that comes from the generator.

While the generator is trained, it samples random noise and produces an output from that noise. The output then goes through the discriminator and gets classified as either "Real" or "Fake" based on the ability of the discriminator to tell one from the other. The generator loss is then calculated from the discriminator's classification – it gets rewarded if it successfully fools the discriminator, and gets penalized otherwise.

The following equation is minimized to training the generator:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(z^{(i)}\right)\right)\right)$$

Exercise to be conducted (E1.4)

To get and show the low-resolution image

```
import os

import pickle

import time


import matplotlib.pyplot as plt

import numpy as np

import torch

import torch.nn as nn

import torchvision.datasets as datasets

import torchvision.transforms as transforms

from torch.utils.data import DataLoader

from tqdm import tqdm




class Generator(nn.Module):
    """Image generator

    Takes a noise vector as input and syntheses a single channel image accordingly
    """

    def __init__(self, input_dims, output_dims):
        """Init function

        Declare the network structure as indicated in CW2 Guidance

        Arguments:
            input_dims {int} -- Dimension of input noise vector
            output_dims {int} -- Dimension of the output vector (flatten image)
        """
        super(Generator, self).__init__()
```

```
        ###  TODO: Change the architecture and value as CW2 Guidance required
        self.fc0 = nn.Sequential(
            nn.Linear(input_dims, 128),
            nn.LeakyReLU(0.2))

        # output hidden layer
        self.fc1 = nn.Sequential(
            nn.Linear(128, output_dims),
            nn.Tanh())

    def forward(self, x):
        """Forward function

        Arguments:
            x {Tensor} -- a batch of noise vectors in shape (<batch_size>x<input_dims>)

        Returns:
            Tensor -- a batch of flatten image in shape (<batch_size>x<output_dims>)
        """
        ###  TODO: modify to be consistent with the network structure
        x = self.fc0(x)
        x = self.fc1(x)
        return x


class Discriminator(nn.Module):
    """Image discriminator

    Takes a image as input and predict if it is real from the dataset or fake synthesised by the
generator
    """

    def __init__(self, input_dims, output_dims=1):
```

```
        """Init function

        Declare the discriminator network structure as indicated in CW2 Guidance

        Arguments:
            input_dims {int} -- Dimension of the flatten input images

        Keyword Arguments:
            output_dims {int} -- Predicted probability (default: {1})
        """
        super(Discriminator, self).__init__()

        ###  TODO: Change the architecture and value as CW2 Guidance required

        self.fc0 = nn.Sequential(
            nn.Linear(input_dims, 784),
            nn.LeakyReLU(0.2),
            nn.Dropout(0.3)
        )
        self.fc1 = nn.Sequential(
            nn.Linear(784, 1),
            nn.Sigmoid()
        )

    def forward(self, x):
        """Forward function

        Arguments:
            x {Tensor} -- a batch of 2D image in shape (<batch_size>xHxW)

        Returns:
            Tensor -- predicted probabilities (<batch_size>)
```

```
    """
    ###  TODO: modify to be consistent with the network structure

    x = self.fc0(x)
    x = self.fc1(x)
    return x

def show_result(G_net, z_, num_epoch, show=False, save=False, path='result.png'):
  """Result visualisation

  Show and save the generated figures in the grid fashion

  Arguments:
    G_net {[nn.Module]} -- The generator instant
    z_ {[Tensor]} -- Input noise vectors
    num_epoch {[int]} -- Indicate how many epoch has the generator been trained

  Keyword Arguments:
    show {bool} -- If to display the images (default: {False})
    save {bool} -- If to store the images (default: {False})
    path {str} -- path to store the images (default: {'result.png'})
  """

  ###  TODO: complete the rest of part
  # hint: use plt.subplots to construct grid
  # hint: use plt.imshow and plt.savefig to display and store the images



def show_train_hist(hist, show=False, save=False, path='Train_hist.png'):
  """Loss tracker
```

```
Plot the losses of generator and discriminator independently to see the trend

Arguments:
    hist {[dict]} -- Tracking variables

Keyword Arguments:
    show {bool} -- If to display the figure (default: {False})
    save {bool} -- If to store the figure (default: {False})
    path {str} -- path to store the figure (default: {'Train_hist.png'})
"""
x = range(len(hist['D_losses']))

y1 = hist['D_losses']
y2 = hist['G_losses']

plt.plot(x, y1, label='D_loss')
plt.plot(x, y2, label='G_loss')

plt.xlabel('Epoch')
plt.ylabel('Loss')

plt.legend(loc=4)
plt.grid(True)
plt.tight_layout()

if save:
    plt.savefig(path)

if show:
    plt.show()
else:
    plt.close()
```

```python
def create_noise(num, dim):
    """Noise constructor

    returns a tensor filled with random numbers from a standard normal distribution

    Arguments:
        num {int} -- Number of vectors
        dim {int} -- Dimension of vectors

    Returns:
        [Tensor] -- the generated noise vector batch
    """
    return torch.randn(num, dim)



if __name__ == '__main__':
    # initialise the device for training, if gpu is available, device = 'cuda', else: device = 'cpu'
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    data_dir = './MNIST_data/'
    save_dir = './MNIST_GAN_results/'
    image_save_dir = './MNIST_GAN_results/results'

    # create folder if not exist
    if not os.path.exists(save_dir):
        os.mkdir(save_dir)
    if not os.path.exists(image_save_dir):
        os.mkdir(image_save_dir)
```

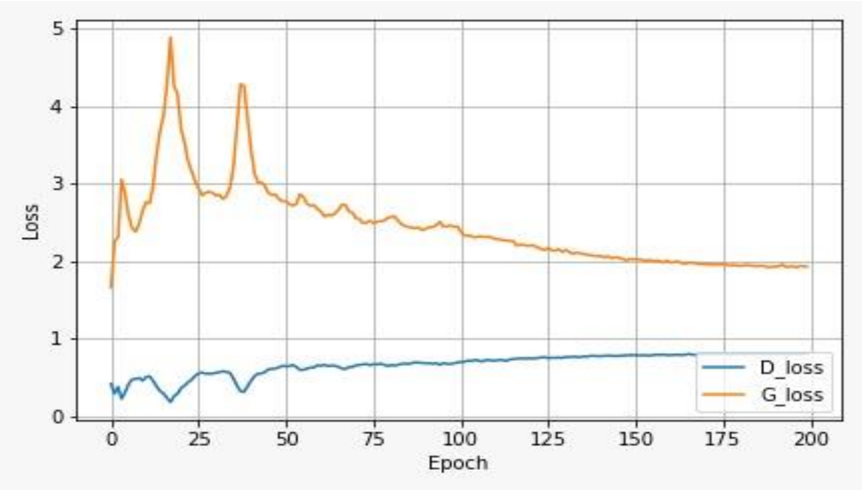
```
# training parameters : change here
batch_size = 100
learning_rate = 0.0002
epochs = 200


# parameters for Models
image_size = 28
G_input_dim = 100
G_output_dim = image_size * image_size
D_input_dim = image_size * image_size
D_output_dim = 1
hidden_size = 32


# construct the dataset and data loader
transform = transforms.Compose([transforms.ToTensor(),transforms.Normalize(mean=(0.5,),
std=(0.5,))])
train_data = datasets.MNIST(root=data_dir, train=True, transform=transform, download=True)
train_loader = DataLoader(dataset=train_data, batch_size=batch_size, shuffle=True)


# declare the generator and discriminator networks
G_net = Generator(G_input_dim, G_output_dim).to(device)
D_net = Discriminator(D_input_dim, D_output_dim).to(device)

# Binary Cross Entropy Loss function
criterion = nn.BCELoss().to(device)

# Initialise the Optimizers
G_optimizer = torch.optim.Adam(G_net.parameters(), lr=learning_rate)
D_optimizer = torch.optim.Adam(D_net.parameters(), lr=learning_rate)
```

```
# tracking variables
train_hist = {}
train_hist['D_losses'] = []
train_hist['G_losses'] = []
train_hist['per_epoch_ptimes'] = []
train_hist['total_ptime'] = []

start_time = time.time()
# training loop
for epoch in range(epochs):
    G_net.train()
    D_net.train()
    Loss_G = []
    Loss_D = []
    epoch_start_time = time.time()
    for (image, _) in tqdm(train_loader):
        image = image.to(device)
        b_size = len(image)
        # creat real and fake labels
        real_label = torch.ones(b_size, 1).to(device)
        fake_label = torch.zeros(b_size, 1).to(device)

        # generate fake images
        data_fake = G_net(create_noise(b_size, G_input_dim).to(device))
        data_real = image.view(b_size, D_input_dim)

        # --------train the discriminator network----------
        # compute the loss for real and fake images
        output_real = D_net(data_real)
        output_fake = D_net(data_fake)
        loss_real = criterion(output_real, real_label)
```

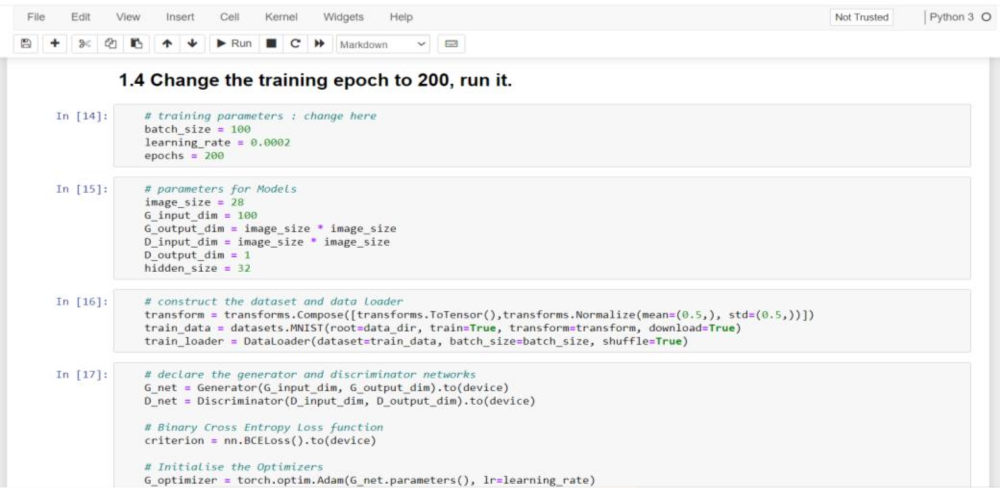
```
        loss_fake = criterion(output_fake, fake_label)
        loss_d = loss_real + loss_fake

        # back propagation
        D_optimizer.zero_grad()
        loss_d.backward()
        D_optimizer.step()

        # -------- train the generator network-----------
        data_fake = G_net(create_noise(b_size, G_input_dim).to(device))

        # compute the loss for generator network
        output_fake = D_net(data_fake)
        loss_g = criterion(output_fake, real_label)

        ## back propagation
        G_optimizer.zero_grad()
        loss_g.backward()
        G_optimizer.step()

        ## store the loss of each iter
        Loss_D.append(loss_d.item())
        Loss_G.append(loss_g.item())

    epoch_loss_g = np.mean(Loss_G)  # mean generator loss for the epoch
    epoch_loss_d = np.mean(Loss_D)  # mean discriminator loss for the epoch
    epoch_end_time = time.time()
    per_epoch_ptime = epoch_end_time - epoch_start_time

    print("Epoch %d of %d with %.2f s" % (epoch + 1, epochs, per_epoch_ptime))
    print("Generator loss: %.8f, Discriminator loss: %.8f" % (epoch_loss_g, epoch_loss_d))
```

```
        path = image_save_dir + '/MNIST_GAN_' + str(epoch + 1) + '.png'
        show_result(G_net, create_noise(25, 100).to(device), (epoch + 1), save=True, path=path)

        # record the loss for every epoch
        train_hist['G_losses'].append(epoch_loss_g)
        train_hist['D_losses'].append(epoch_loss_d)
        train_hist['per_epoch_ptimes'].append(per_epoch_ptime)

    end_time = time.time()
    total_ptime = end_time - start_time
    train_hist['total_ptime'].append(total_ptime)

    print('Avg per epoch ptime: %.2f, total %d epochs ptime: %.2f' % (
np.mean(train_hist['per_epoch_ptimes']), epochs, total_ptime))
    print("Training finish!... save training results")
    with open(save_dir + '/train_hist.pkl', 'wb') as f:
        pickle.dump(train_hist, f)

    show_train_hist(train_hist, save=True, path=save_dir +
'/MNIST_GAN_train_hist_epoch_chaged.png')
```

Screenshots:



Here we changes the epoch to 100



End of training: took too long