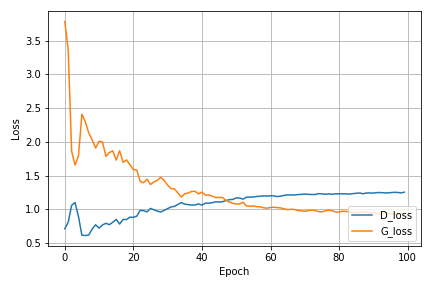
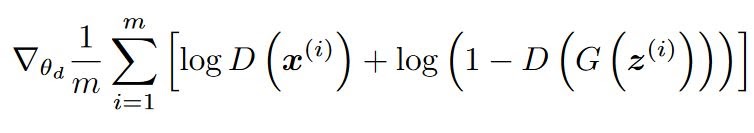
# Question to be answered (Q2.4)

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.

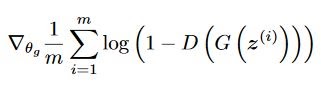


**log(D(x))**refers to the probability that the generator is rightly classifying the real image,

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:



# 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()

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))

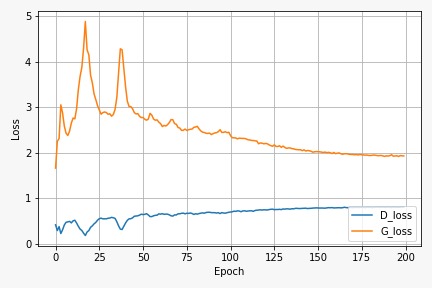
print("Training finish!... save training results")

with open(save\_dir + '/train\_hist.pkl', 'wb') as f:

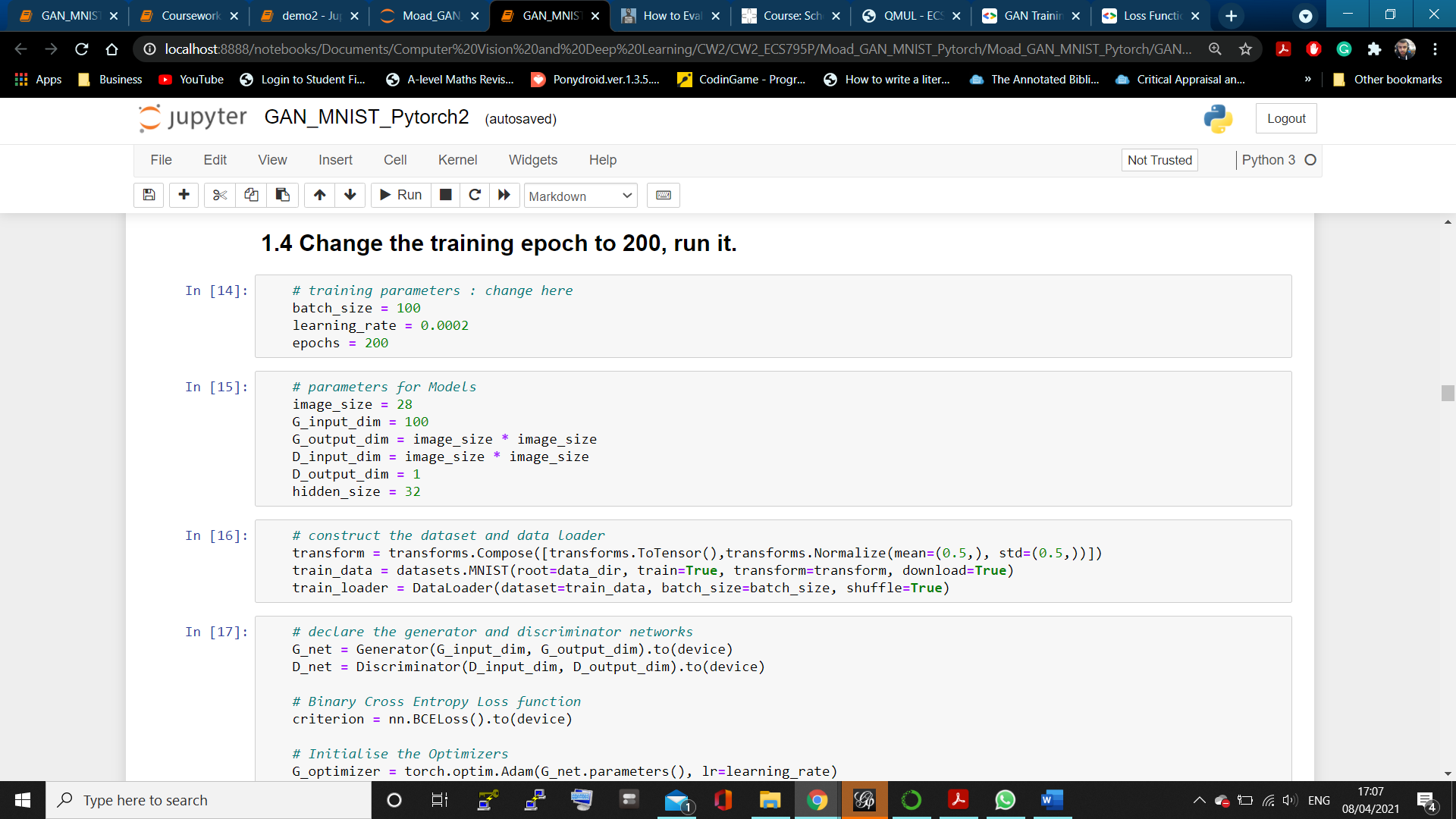
pickle.dump(train\_hist, f)

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

