PRACTICE #02

Generative Adversarial Networks

(*Keyword:* GAN)

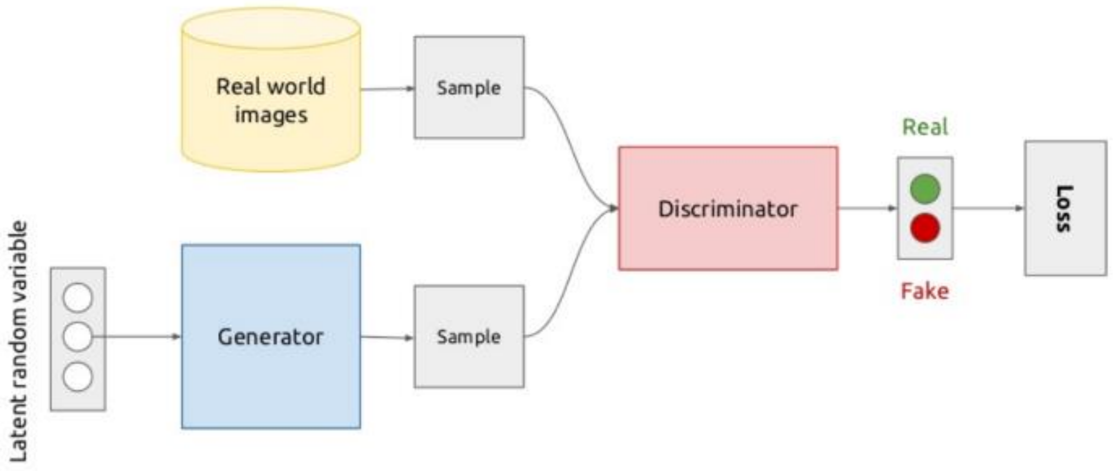
I. Goals

The objective of this assignment is to introduce the GAN model and how to train the first GAN Model using PyTorch to generate images

II. Introduction

A generative adversarial network (GAN) uses two neural networks to compete with each other like in a game, one known as a "discriminator" and the other known as the "generator".

- The Generator wants to learn to generate realistic images that are indistinguishable from the real data. The *input* of Generator is a Gaussian noise random sample, and its *output* is a generated datapoint
- The Discriminator wants to tell the real & fake images apart. The *input* of Discriminator is a datapoint or an image, and its *output* is a probability assigned to datapoint being real. It can be seen as a binary classifier



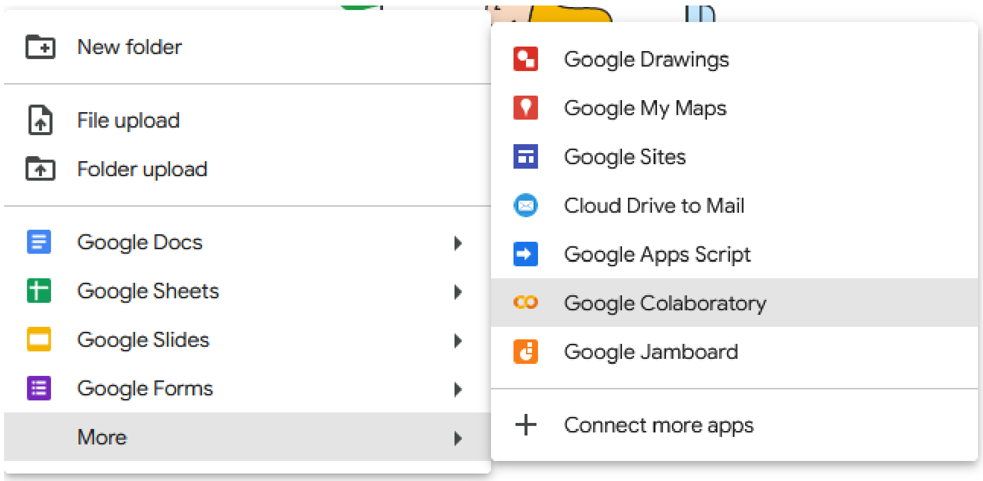
III. Content

1. Setting up Google Colab for training a deep learning model

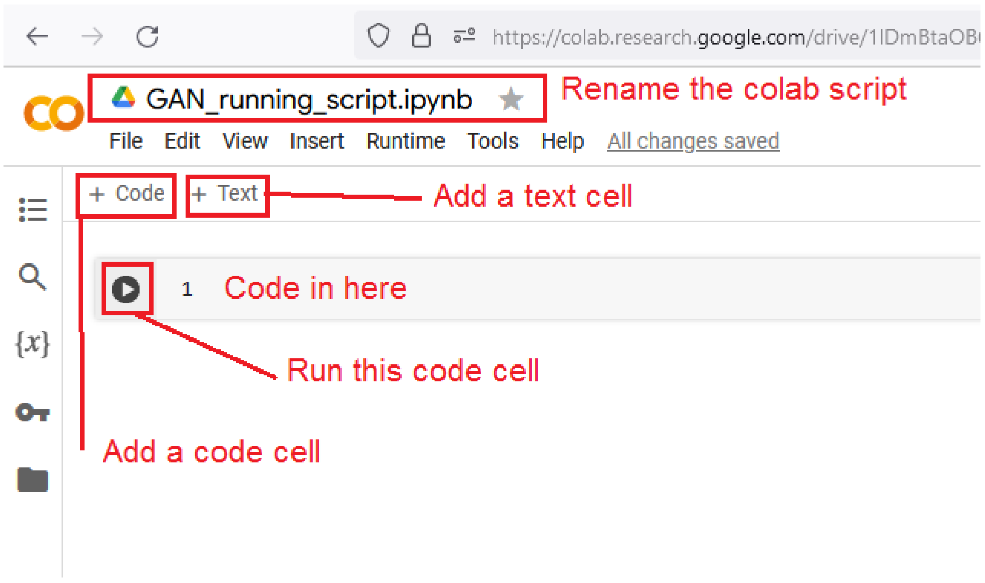
Colab is a cloud-based Jupyter notebook environment that is available for free. One of its key features is that it eliminates the need for any setup, and the notebooks you create can be collaboratively edited by your team members, similar to how you edit documents in Google Docs. Colab provides support for numerous widely-used machine learning libraries, which can be conveniently imported into your notebook.

Google Colab has the ability to connect with Google Drive for data storage. Additionally, Colab can train models on CPU, GPU, or TPU. You can choose to purchase premium packages to have more powerful configurations for training large models.

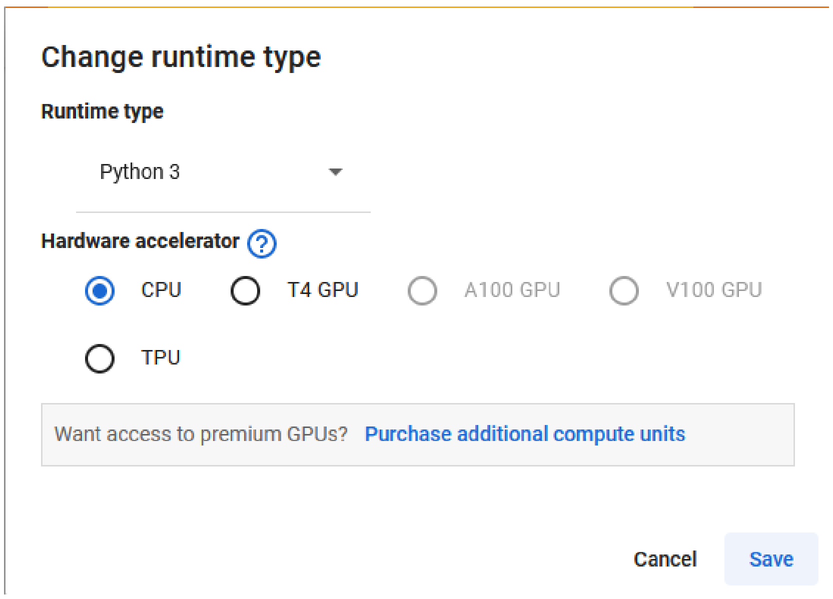
To create a Colab file in a folder on Google Drive, right-click and select "More," then choose "Google Colaboratory."



Colab opens a web page interface as a notebook where users can add and run code cells.



Users can choose the type of device to run on, such as CPU, GPU, or TPU, by selecting the "Runtime" menu and then selecting "Change runtime type."



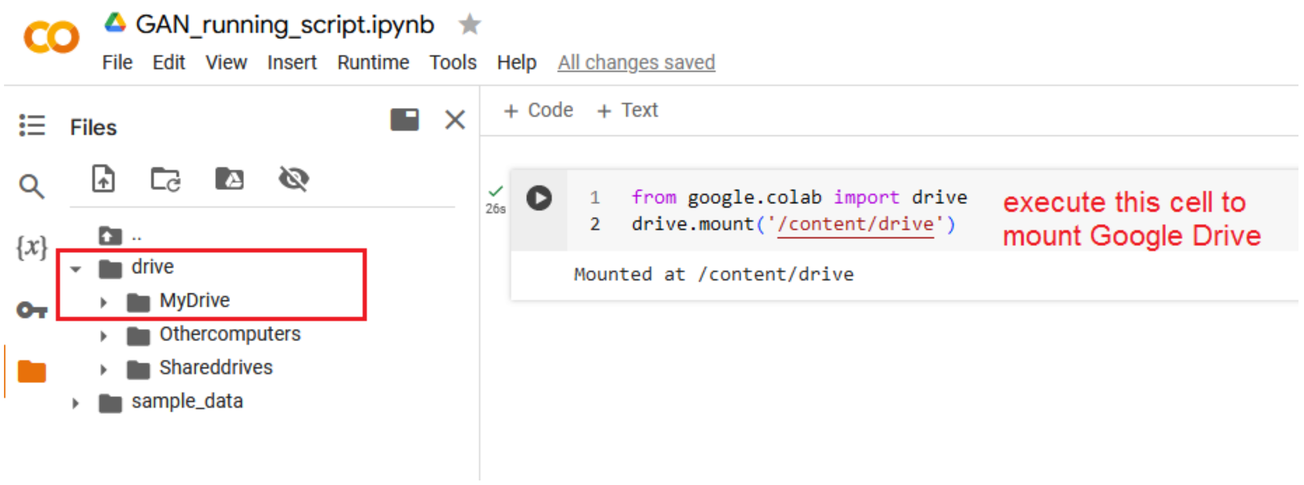
Change runtime type

Runtime type: Python 3

Hardware accelerator: [✓ CPU] T4 GPU A100 GPU V100 GPU TPU

Want access to premium GPUs? Purchase additional compute units

Users can also mount Colab to their personal Google Drive to store data.



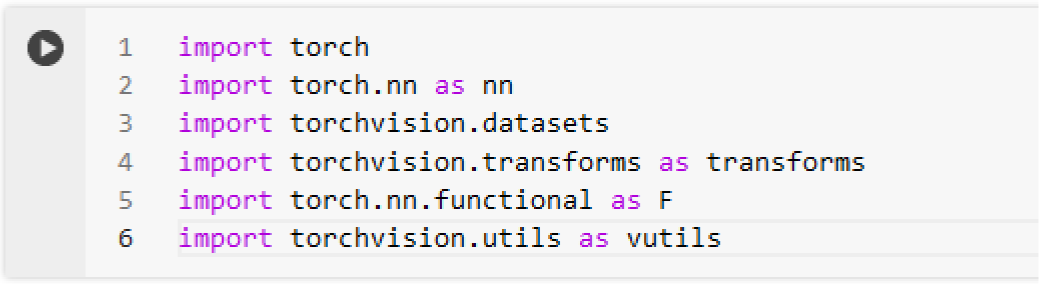
```
from google.colab import drive
drive.mount('/content/drive')
```

execute this cell to mount Google Drive

```
Mounted at /content/drive
```

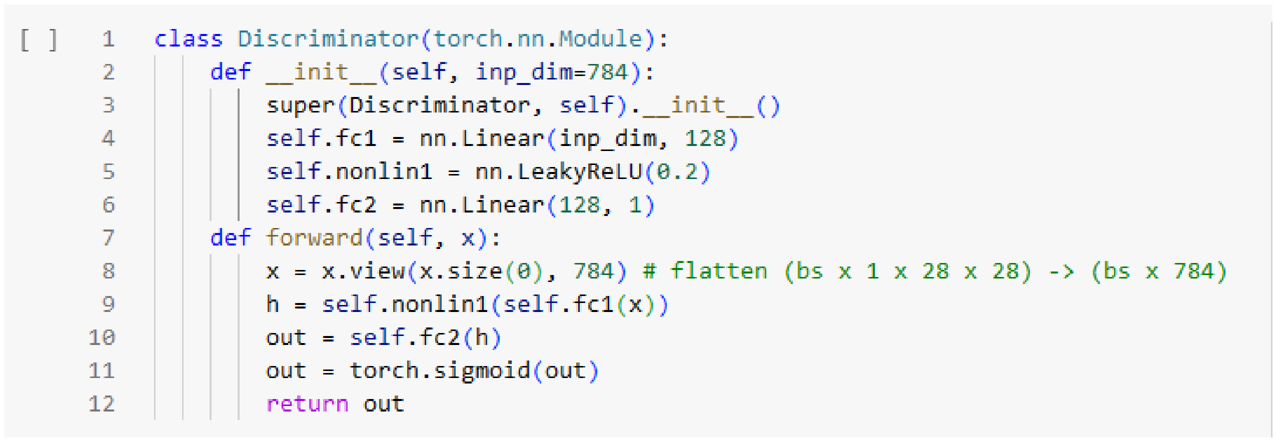
2. Define generator and discriminator model in Pytorch.

PyTorch defines a neural network by constructing a class that inherits from nn.Module. To define the operations of the model, you need to override the forward function



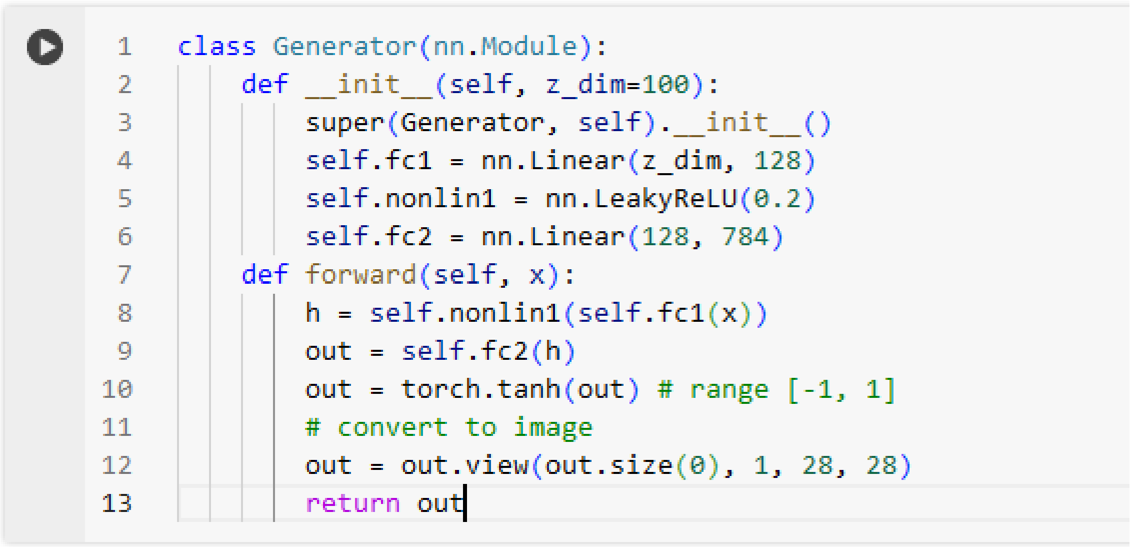
```python
import torch
import torch.nn as nn
import torchvision.datasets
import torchvision.transforms as transforms
import torch.nn.functional as F
import torchvision.utils as vutils
```

Let's define a small 2-layer fully connected neural network (so one hidden layer) for the discriminator D:



```
class Discriminator(torch.nn.Module):
    def __init__(self, inp_dim=784):
        super(Discriminator, self).__init__()
        self.fc1 = nn.Linear(inp_dim, 128)
        self.nonlin1 = nn.LeakyReLU(0.2)
        self.fc2 = nn.Linear(128, 1)
    def forward(self, x):
        x = x.view(x.size(0), 784) # flatten (bs x 1 x 28 x 28) -> (bs x 784)
        h = self.nonlin1(self.fc1(x))
        out = self.fc2(h)
        out = torch.sigmoid(out)
        return out
```

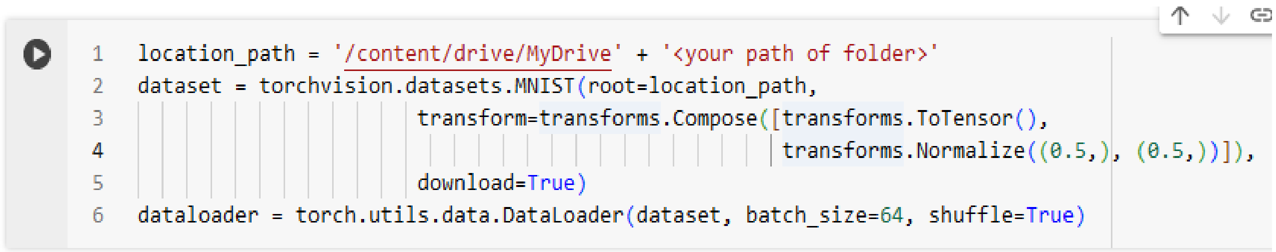
And a small 2-layer neural network for the generator G. G takes a 100-dimensional noise vector and generates an output of the size matching the data.



```
class Generator(nn.Module):
    def __init__(self, z_dim=100):
        super(Generator, self).__init__()
        self.fc1 = nn.Linear(z_dim, 128)
        self.nonlin1 = nn.LeakyReLU(0.2)
        self.fc2 = nn.Linear(128, 784)
    def forward(self, x):
        h = self.nonlin1(self.fc1(x))
        out = self.fc2(h)
        out = torch.tanh(out) # range [-1, 1]
        # convert to image
        out = out.view(out.size(0), 1, 28, 28)
        return out
```

3. Dataset processing

Let's download the MNIST data. If you have already downloaded it locally, you can modify the data paths to reference your existing files. The variable "location_path" is the path on Google Drive, indicating the directory used for saving the downloaded MNIST dataset. "Transform" is a set of transformations applied to the data samples in the dataset. "DataLoader" is a PyTorch structure used to load data in batches for training models.



```python
location_path = '/content/drive/MyDrive' + '<your path of folder>'
dataset = torchvision.datasets.MNIST(root=location_path,
                                     transform=transforms.Compose([transforms.ToTensor(),
                                                                    transforms.Normalize((0.5,), (0.5,))]),
                                     download=True)
dataloader = torch.utils.data.DataLoader(dataset, batch_size=64, shuffle=True)
```

4. Define the loss function for GAN

The training process of a GAN network involves simultaneously training the generator and discriminator models. When training the discriminator, the generator remains fixed, and when training the generator, the discriminator remains fixed. The discriminator plays a role as a binary classifier, so the core of its loss function is Binary Cross-Entropy. (BCE)

GAN loss function is commonly described as a combined function representing both the Discriminator's objective and the Generator's objective.

With ***z*** representing the noise vector used as input to the generator network, *x* is a real sample drawn from the true data distribution, *G(z)* denotes a fake sample generated by the generator network using the noise vector *z* as input. *D(x)* outputs the predicted score for a real sample *x* by the discriminator network. *D(G(z))* is the predicted score for a fake sample *G(z)* by the discriminator network, the formula for GANs objective function could be written as:

$$\underset{G}{min}\underset{D}{max}V(D,G) = E_{x\sim p_{data}(x)} log\left(D(x)\right) + E_{z\sim p_z(z)}\left(1-\left(D\left(G(z)\right)\right)\right)$$

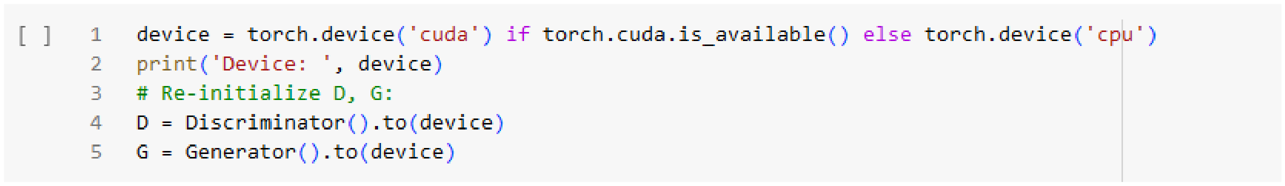
The objective function of the discriminator is to maximize V(D), which is presented as the following:

$$\underset{D}{max}V(D) = E_{x\sim p_{data}(x)} log\left(D(x)\right) + E_{z\sim p_z(z)}\left(1-\left(D\left(G(z)\right)\right)\right)$$

The goal of this is to train a Discriminator model so that its classification ability is optimized. In this function, let's temporarily treat G as constant and only focus on the $max_D V(G,D)$ term. As a result, the model wants to maximize the value of $D(x)$ so that it is close to 1. On the other hand, the objective function of the generator is to minimize $V(G)$, which is computed as:

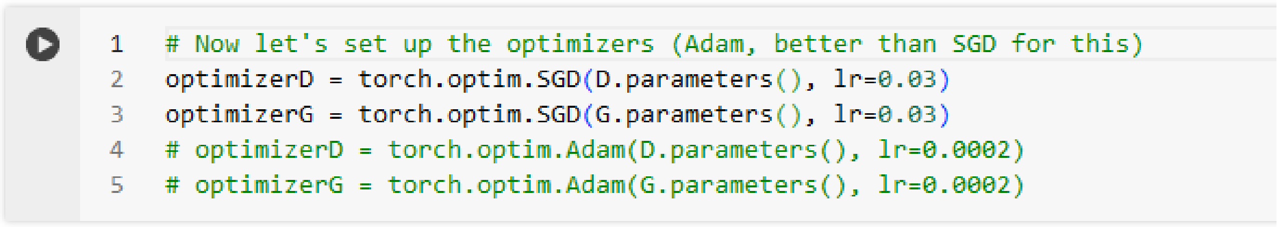
$$\underset{G}{min}V(G) = E_{z\sim p_z(z)}\left(1-\left(D\left(G(z)\right)\right)\right)$$

The first step is to get the GPU device, and transfer models to GPU



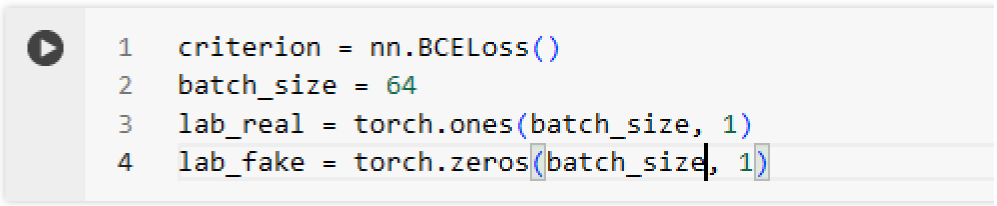
```
[ ]
device = torch.device('cuda') if torch.cuda.is_available() else torch.device('cpu')
print('Device: ', device)
# Re-initialize D, G:
D = Discriminator().to(device)
G = Generator().to(device)
```

The second step is to set up the optimizers for Discriminator and Generator



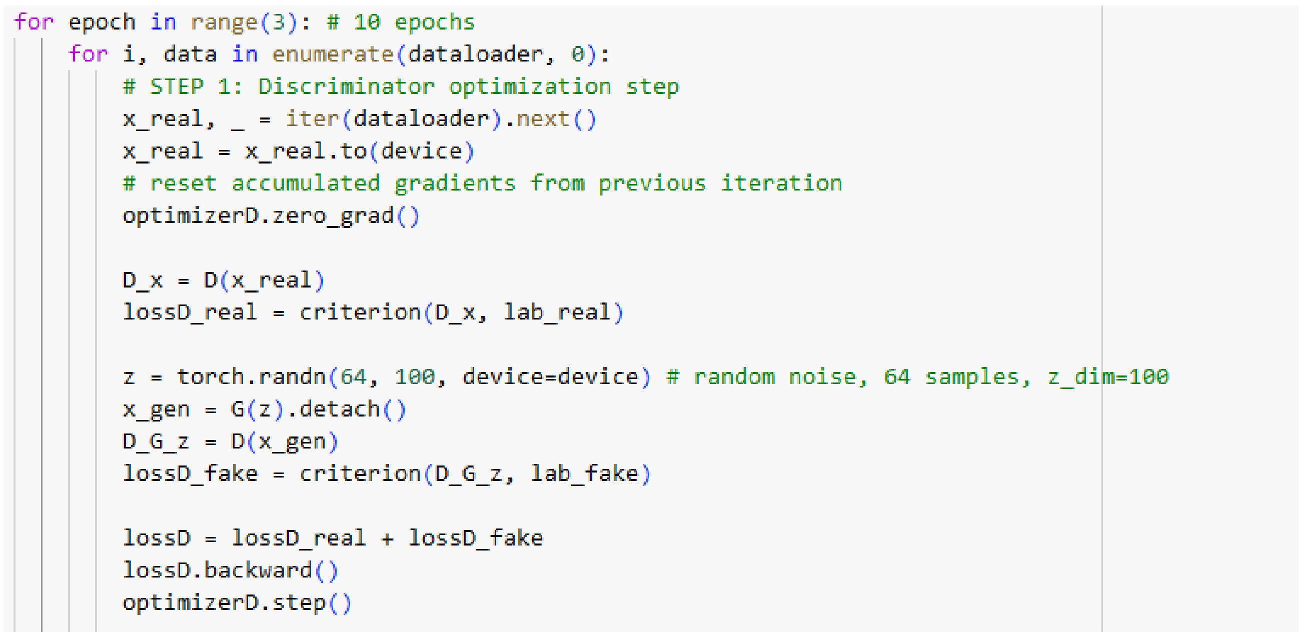
```
# Now let's set up the optimizers (Adam, better than SGD for this)
optimizerD = torch.optim.SGD(D.parameters(), lr=0.03)
optimizerG = torch.optim.SGD(G.parameters(), lr=0.03)
# optimizerD = torch.optim.Adam(D.parameters(), lr=0.0002)
# optimizerG = torch.optim.Adam(G.parameters(), lr=0.0002)
```

Next, we define the loss function BCE



```
criterion = nn.BCELoss()
batch_size = 64
lab_real = torch.ones(batch_size, 1)
lab_fake = torch.zeros(batch_size, 1)
```

Finally, we train two models G and D on dataset

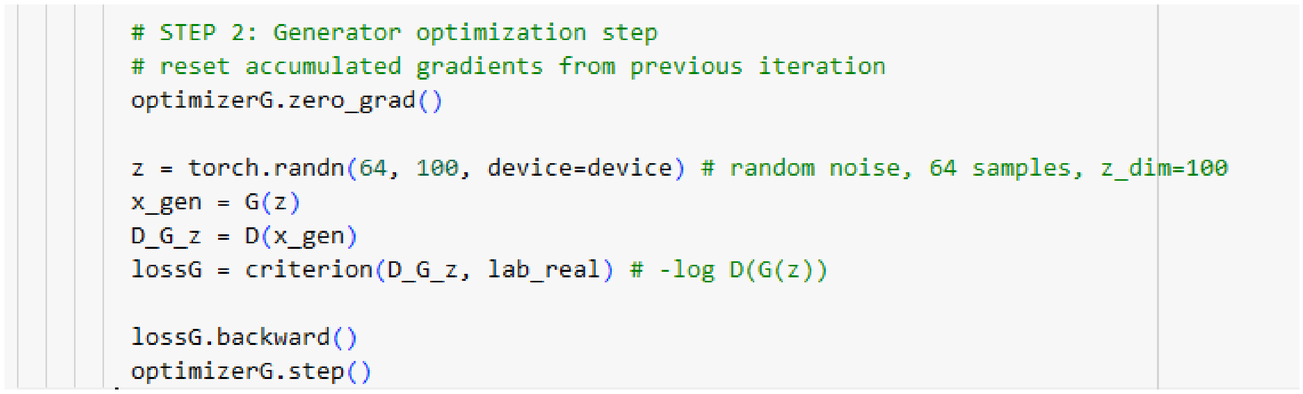


```
for epoch in range(3): # 10 epochs
    for i, data in enumerate(dataloader, 0):
        # STEP 1: Discriminator optimization step
        x_real, _ = iter(dataloader).next()
        x_real = x_real.to(device)
        # reset accumulated gradients from previous iteration
        optimizerD.zero_grad()

        D_x = D(x_real)
        lossD_real = criterion(D_x, lab_real)

        z = torch.randn(64, 100, device=device) # random noise, 64 samples, z_dim=100
        x_gen = G(z).detach()
        D_G_z = D(x_gen)
        lossD_fake = criterion(D_G_z, lab_fake)

        lossD = lossD_real + lossD_fake
        lossD.backward()
        optimizerD.step()
```



```
            # STEP 2: Generator optimization step
            # reset accumulated gradients from previous iteration
            optimizerG.zero_grad()

            z = torch.randn(64, 100, device=device) # random noise, 64 samples, z_dim=100
            x_gen = G(z)
            D_G_z = D(x_gen)
            lossG = criterion(D_G_z, lab_real) # -log D(G(z))

            lossG.backward()
            optimizerG.step()
```

IV. Requirements

1. In this assignment, you are required to implement a GAN model to generate handwritten digit images from 0 to 9. The model is trained on the MNIST dataset. You need to define the architectures of both the generator and discriminator. In the inference phase, a random noise vector is fed into the model as an input, and the output is the image generated by the generator.
2. Running inference multiple times with different random noise vectors and giving your observations about the results of the generated images by the generator.
3. The directory structure of the compressed submission
   - *doc*: a report file *StudentID_report_p02.doc/docx/pdf*. The report file includes your personal information (StudentID, name), describes the architecture of the GAN you have built, the achieved results, and the observations about those results.
   - *source*: contains entire source code
   - *bonus*: optional, for plus points, if available

   A separate report doc file is required even if students use Jupyter Notebook file (named *StduentID_p02*).
4. Other requirements
   - The report should be presented clearly and intuitively: a self-scoring table (assessment) of the results of the work compared to the corresponding requirements (0-100%), a list of the functions included in the program with proof images, summarize the usage and implementation (for example: through pseudo-code, description of methods, or how to do it, *do not copy the source code into the report*).
   - The source code needs to be commented on the corresponding lines.