Prerequisites

Installation of Anaconda: https://www.anaconda.com/products/individual

At the bottom of the site are download links, depending on the user's operating system.

Click 64-Bit Graphical Installer
Once installed, open command
type "conda create --name deeplearning"
(then enter Y to proceed)
"conda active deep-learning"

The next thing to downloaded is PyCharm which is a Python IDE. https://www.jetbrains.com/pycharm/

Choose the free-trial

Next, install Pytorch which can be installed at the website: https://pytorch.org/

Pick: Preview (Nightly), Your OS, Conda, Python, and your CUDA if applicable (otherwise, select None)
Run the command given on your terminal within deep-learning directory. Proceed with installation when it asks.
Open PyTorch. Click **Configure**, then **Settings**. Go to **Project Interpreter**
Click the wheel on the top right and press **Add**.
Then go to Conda Environment. Click **Existing environment**
Click **Make available to all projects.** Press Ok

Testing:
Create New Project. Press File, New, Python File and choose any custom name
Write the following to check if torch is installed:

```
import torch
print(torch.__version__)
```

The implementation

Generator and Discriminator

Create a file called DCGAN.py

Note: I will only cover the major parts of the code but all details will be in the folder as a .py file.

We can build our model of our DCGAN. We use the Module container within Torch.nn to create the our discriminator and generator. This will create the neural networks in our programs. So type in what is in the screen shot below.

```
import torch
import torch.nn as nn
```

We define a function discriminator that allows us to initialize features of the inputs and dimensions the of convoluted neural networks. We have 4 fractionally sided convolutions with dimensions such as kernel size, stride, and padding. The convolutional channels expand in the discriminator while getting smaller in the generator. All images were to be processed in size 64 x 64.

```
class Discriminator(nn.Module):
    def __init__(self, channels_img, features_d):
        super(Discriminator, self).__init__()
        self.disc = nn.Sequential(
            # input: N x channels_img x 64 x 64
            nn.Conv2d(
                channels_img, features_d, kernel_size=4, stride=2, padding=1
            ),
            nn.LeakyReLU(0.2),
            # _block(in_channels, out_channels, kernel_size, stride, padding)
            self._block(features_d, features_d * 2, 4, 2, 1),
            self._block(features_d * 2, features_d * 4, 4, 2, 1),
            self._block(features_d * 4, features_d * 8, 4, 2, 1),
            # After all _block img output is 4x4 (Conv2d below makes into 1x1)
            nn.Conv2d(features_d * 8, 1, kernel_size=4, stride=2, padding=0),
            nn.Sigmoid(),
        )
```

We used batch normalization, ReLU, and LeakyReLU as mentioned in the DCGAN architecture. The discriminator leads to the **sigmoid** cross entropy loss function as well.

We also need to define **block** and return the convolutions while setting bias = False and the nn.LeakyReLu = 0.2

We also define **forward**

```
    def _block(self, in_channels, out_channels, kernel_size, stride, padding):
        return nn.Sequential(
            nn.Conv2d(
                in_channels,
                out_channels,
                kernel_size,
                stride,
                padding,
                bias=False,
            ),
            #nn.BatchNorm2d(out_channels),
            nn.LeakyReLU(0.2),
        )

    def forward(self, x):
        return self.disc(x)
```

Our **generator** class follows similar code but uses nn.Tanh() in our __init__ function and we define _block once again but use nn.ReLu().
Our forward returns self.net(x).

We define functions that initialize our weight and test functions like the following
For our weight function, every convolution will have a weight set to 0 and the standard deviation was set to 0.2

We define a test function to check if our model works, and if our discriminator and generator is ready to train

```
def initialize_weights(model):
    # Initializes weights according to the DCGAN paper
    for m in model.modules():
        if isinstance(m, (nn.Conv2d, nn.ConvTranspose2d, nn.BatchNorm2d)):
            nn.init.normal_(m.weight.data, 0.0, 0.02)

def test():
    N, in_channels, H, W = 8, 3, 64, 64
    noise_dim = 100
    x = torch.randn((N, in_channels, H, W))
    disc = Discriminator(in_channels, 8)
    assert disc(x).shape == (N, 1, 1, 1), "Discriminator test failed"
    gen = Generator(noise_dim, in_channels, 8)
    z = torch.randn((N, noise_dim, 1, 1))
    assert gen(z).shape == (N, in_channels, H, W), "Generator test failed"
    print("Success")
```

We then run test()

Training Data

We create training.py. For our training data, we initialize our hyperparameters, optimizers, and loss function were optimized.

mini-batch size: 128

our learning rate: 0.0002

image size: 64

We also initialize the transforming of our images as it goes the convolutions

```
# Hyperparameters etc.
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
LEARNING_RATE = 2e-4  # could also use two lrs, one for gen and one for disc
BATCH_SIZE = 128
IMAGE_SIZE = 64
CHANNELS_IMG = 1
NOISE_DIM = 100
NUM_EPOCHS = 5
FEATURES_DISC = 64
FEATURES_GEN = 64

transforms = transforms.Compose(
    [
        transforms.Resize(IMAGE_SIZE),
        transforms.ToTensor(),
        transforms.Normalize(
            [0.5 for _ in range(CHANNELS_IMG)], [0.5 for _ in range(CHANNELS_IMG)]
        ),
    ]
)
```

We initialize our data set as the set of images we want to test and set up our data loader, and input our hyperparameter for the generator and discriminator and initialize their weights.

```
dataset = datasets.ImageFolder(root="dataset", transform=transforms)
dataloader = DataLoader(dataset, batch_size=BATCH_SIZE, shuffle=True)
gen = Generator(NOISE_DIM, CHANNELS_IMG, FEATURES_GEN).to(device)
disc = Discriminator(CHANNELS_IMG, FEATURES_DISC).to(device)
initialize_weights(gen)
initialize_weights(disc)

opt_gen = optim.Adam(gen.parameters(), lr=LEARNING_RATE, betas=(0.5, 0.999))
opt_disc = optim.Adam(disc.parameters(), lr=LEARNING_RATE, betas=(0.5, 0.999))
criterion = nn.BCELoss()
```

For the optimizer Adam, momentum term beta1: 0.5, with beta2 = 0.999

We also considered training an MNIST data set as a testing procedure.

We set the labels for discriminator outputs as “real” or “fake”.

The epoch for loop was when the actual training took place. The discriminator and generator were both trained calculation of gradients for backpropagation and constant updating. Losses were calculated and printed occasionally to the tensorboard. The tensorboard will print 32 according to the code

```
for epoch in range(NUM_EPOCHS):
    for batch_idx, (real, _) in enumerate(dataloader):
        real = real.to(device)
        noise = torch.randn(BATCH_SIZE, NOISE_DIM, 1, 1).to(device)
        fake = gen(noise)

        # Train Discriminator
        disc_real = disc(real).reshape(-1)
        loss_disc_real = criterion(disc_real, torch.ones_like(disc_real))
        disc_fake = disc(fake.detach()).reshape(-1)
        loss_disc_fake = criterion(disc_fake, torch.zeros_like(disc_fake))
        loss_disc = (loss_disc_real + loss_disc_fake) / 2
        disc.zero_grad()
        loss_disc.backward()
        opt_disc.step()

        # Train Generator
        output = disc(fake).reshape(-1)
        loss_gen = criterion(output, torch.ones_like(output))
        gen.zero_grad()
        loss_gen.backward()
        opt_gen.step()
        # Print losses occasionally and print to tensorboard
        if batch_idx % 100 == 0:
            print(
                f"Epoch [{epoch}/{NUM_EPOCHS}] Batch {batch_idx}/{len(dataloader)} \
                  Loss D: {loss_disc:.4f}, loss G: {loss_gen:.4f}"
            )
            with torch.no_grad():
                fake = gen(fixed_noise)
```

We run the code and the Tensorboard should pop up, which shows the DCGAN training the images.