DEEP LEARNING
ASSIGNMENT 2 REPORT
M22MA003

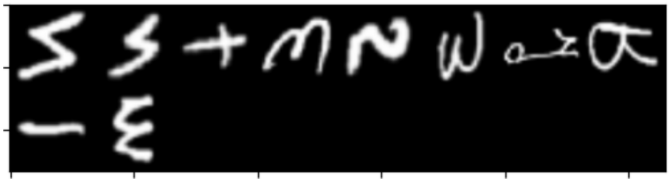
Question1: Train a DCGAN to generate images from noise. Use the EMNIST(Extended MNIST) database to learn the GAN Network.
Perform the following tasks:
a. Uniformly generate ten noise vectors that act as latent representation vectors, and generate the images for these noise vectors, and visualize them at

- **i. After the first epoch.**
- **ii. After n/2 th epoch.**
- **iii. After your last epoch. (say n epochs in total) and comment on the image interpretation at (i), (ii) and (iii) and can you identify the images?**

b. Plot generator and discriminator losses for all the iterations. Also display the best-generated images by the model.[One iteration = forward pass of a mini-batch]

Solution1:-
Roll no. % 2 == 1: Using Resnet 56 as a Discriminator.

1. Install the required libraries: Make sure you have the necessary libraries installed, such as PyTorch, torchvision, matplotlib, and numpy.
2. Import the required libraries: Add the necessary import statements at the beginning of your code to import the required libraries.
3. Set the random seed: Set the random seed value to ensure reproducibility of the results.
4. Define the image transformations: Create a transformation pipeline using the torchvision.transforms.Compose class to resize, crop, convert to tensor, and normalize the images.
5. Load the dataset: Use the torchvision.datasets module to load the dataset. In this case, the EMNIST dataset is used, but you can replace it with your desired dataset.
6. Create the data loader: Use the torch.utils.data.DataLoader class to create a data loader for your dataset, specifying the batch size and whether to shuffle the data.



7. Define the helper functions: Define the utils functions imshow and plot_loss for visualizing images and loss curves, respectively.
8. Initialize the generator and discriminator models: Define the generator and discriminator models (ResGenerator and ResDiscriminator) along with their architectures and necessary layers.

```
Generator(
main): Sequential(
 (0): ConvTranspose2d(100, 512, kernel_size=(4, 4), stride=(1, 1), bias=False)
 (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
 (2): ReLU(inplace=True)
 (3): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
 (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
 (5): ReLU(inplace=True)
 (6): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
 (7): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
 (8): ReLU(inplace=True)
 (9): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
 (10): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
 (11): ReLU(inplace=True)
 (12): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
 (13): Tanh()
```

```
ResDiscriminator(
  (resnet): ResNet56(
    (conv1): Conv2d(1, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (layer1): Sequential(
      (0): ResidualBlock(
        (conv1): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv2): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (shortcut): Sequential()
      )
      (1): ResidualBlock(
        (conv1): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv2): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (shortcut): Sequential()
      )
      (2): ResidualBlock(
        (conv1): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv2): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (shortcut): Sequential()
      )
      (3): ResidualBlock(
        (conv1): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (conv2): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (bn2): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
        (shortcut): Sequential()
      )
      (4): ResidualBlock(
```

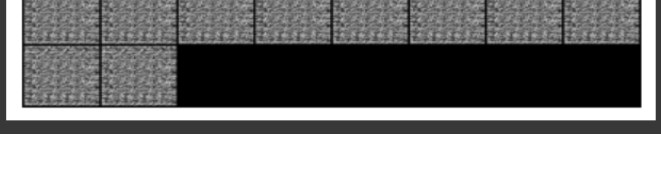
9. Define the loss function and optimizer: Define the loss function (nn.BCELoss) and the optimizers (optim.Adam) for both the generator and discriminator models.
10. Training loop: Iterate over the dataset for the specified number of epochs and perform the following steps:
    - Zero the gradients of the discriminator.
    - Forward pass real images through the discriminator and calculate the loss.
    - Backpropagate and update the discriminator's parameters.
    - Generate fake images using the generator.

- Forward pass the fake images through the discriminator and calculate the loss.
- Backpropagate and update the generator's parameters.
- Print the losses and other statistics periodically.
- Store the generator's generated images for visualization and display real and fake images at **epoch zero, five and last**:-

```
Starting Training Loop...
[0/10][0/147]   Loss_D: 1.4805  Loss_G: 0.5091  D(x): 0.6200    D(G(z)): 0.6325 / 0.6012
[0/10][50/147]  Loss_D: 0.0285  Loss_G: 4.1893  D(x): 0.9881    D(G(z)): 0.0164 / 0.0152
[0/10][100/147] Loss_D: 0.0127  Loss_G: 4.9741  D(x): 0.9945    D(G(z)): 0.0072 / 0.0069
```


```
[1/10][0/147]   Loss_D: 0.0082  Loss_G: 5.4190  D(x): 0.9964    D(G(z)): 0.0045 / 0.0044
[1/10][50/147]  Loss_D: 0.0059  Loss_G: 5.7044  D(x): 0.9975    D(G(z)): 0.0034 / 0.0033
[1/10][100/147] Loss_D: 0.0047  Loss_G: 5.9537  D(x): 0.9980    D(G(z)): 0.0026 / 0.0026
[2/10][0/147]   Loss_D: 0.0039  Loss_G: 6.1329  D(x): 0.9983    D(G(z)): 0.0022 / 0.0022
[2/10][50/147]  Loss_D: 0.0032  Loss_G: 6.3092  D(x): 0.9986    D(G(z)): 0.0018 / 0.0018
[2/10][100/147] Loss_D: 0.0027  Loss_G: 6.4824  D(x): 0.9988    D(G(z)): 0.0015 / 0.0015
[3/10][0/147]   Loss_D: 0.0023  Loss_G: 6.6250  D(x): 0.9990    D(G(z)): 0.0013 / 0.0013
[3/10][50/147]  Loss_D: 0.0021  Loss_G: 6.7376  D(x): 0.9991    D(G(z)): 0.0012 / 0.0012
[3/10][100/147] Loss_D: 0.0019  Loss_G: 6.8111  D(x): 0.9992    D(G(z)): 0.0011 / 0.0011
[4/10][0/147]   Loss_D: 0.0017  Loss_G: 6.9323  D(x): 0.9993    D(G(z)): 0.0010 / 0.0010
[4/10][50/147]  Loss_D: 0.0015  Loss_G: 7.0602  D(x): 0.9994    D(G(z)): 0.0009 / 0.0009
[4/10][100/147] Loss_D: 0.0013  Loss_G: 7.1894  D(x): 0.9994    D(G(z)): 0.0008 / 0.0008
[5/10][0/147]   Loss_D: 0.0012  Loss_G: 7.2743  D(x): 0.9995    D(G(z)): 0.0007 / 0.0007
[5/10][50/147]  Loss_D: 0.0011  Loss_G: 7.3493  D(x): 0.9995    D(G(z)): 0.0006 / 0.0006
[5/10][100/147] Loss_D: 0.0010  Loss_G: 7.4455  D(x): 0.9996    D(G(z)): 0.0006 / 0.0006
```


```
[6/10][0/147]   Loss_D: 0.0009  Loss_G: 7.5470  D(x): 0.9996    D(G(z)): 0.0005 / 0.0005
[6/10][50/147]  Loss_D: 0.0009  Loss_G: 7.6181  D(x): 0.9996    D(G(z)): 0.0005 / 0.0005
[6/10][100/147] Loss_D: 0.0008  Loss_G: 7.6983  D(x): 0.9997    D(G(z)): 0.0005 / 0.0005
[7/10][0/147]   Loss_D: 0.0007  Loss_G: 7.7816  D(x): 0.9997    D(G(z)): 0.0004 / 0.0004
[7/10][50/147]  Loss_D: 0.0007  Loss_G: 7.8455  D(x): 0.9997    D(G(z)): 0.0004 / 0.0004
[7/10][100/147] Loss_D: 0.0006  Loss_G: 7.9039  D(x): 0.9997    D(G(z)): 0.0004 / 0.0004
[8/10][0/147]   Loss_D: 0.0006  Loss_G: 7.9719  D(x): 0.9998    D(G(z)): 0.0003 / 0.0003
[8/10][50/147]  Loss_D: 0.0006  Loss_G: 8.0316  D(x): 0.9998    D(G(z)): 0.0003 / 0.0003
[8/10][100/147] Loss_D: 0.0005  Loss_G: 8.0848  D(x): 0.9998    D(G(z)): 0.0003 / 0.0003
[9/10][0/147]   Loss_D: 0.0005  Loss_G: 8.1368  D(x): 0.9998    D(G(z)): 0.0003 / 0.0003
[9/10][50/147]  Loss_D: 0.0005  Loss_G: 8.1814  D(x): 0.9998    D(G(z)): 0.0003 / 0.0003
[9/10][100/147] Loss_D: 0.0005  Loss_G: 8.2393  D(x): 0.9998    D(G(z)): 0.0003 / 0.0003
```



[Colab Link Question 1]

Question2: Download the pre-trained StyleGan(v1, v2 or v3).
1. Generate 10 realistic images using the StyleGAN.
2. Take your face image and 5 different face images of your friends (One image per friend). Perform feature disentanglement and linear interpolation between your face and your friend's face.

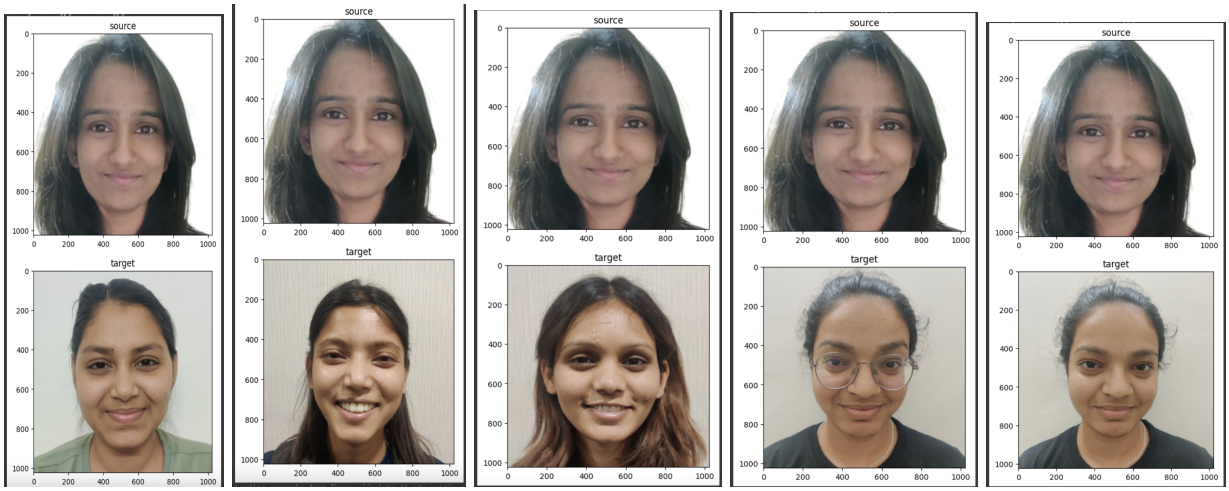
Solution2 :-

Steps followed:-

1. Download the pre-trained StyleGAN2 model file (ffhq.pkl) and save it locally.
2. Clone the StyleGAN2 repository from GitHub (stylegan2-ada-pytorch).
3. Download the shape predictor file (shape_predictor_5_face_landmarks.dat) for detecting facial landmarks.
4. Extract the shape predictor file from the compressed format.
5. Install the required packages (ninja, dnnlib, and legacy).
6. Import necessary libraries for image processing, deep learning, and video creation.
7. Generate 10 realistic images using StyleGAN. Eg:-

```
Loading networks from "https://nvlabs-fi-cdn.nvidia.com/stylegan2-ada-pytorch/pretrained/ffhq.pkl"...
Generating image for seed 90 (0/10) ...
Setting up PyTorch plugin "bias_act_plugin"... Done.
Setting up PyTorch plugin "upfirdn2d_plugin"... Done.
Generating image for seed 108 (1/10) ...
Generating image for seed 250 (2/10) ...
Generating image for seed 483 (3/10) ...
Generating image for seed 583 (4/10) ...
Generating image for seed 458 (5/10) ...
Generating image for seed 384 (6/10) ...
Generating image for seed 900 (7/10) ...
Generating image for seed 999 (8/10) ...
Generating image for seed 27 (9/10) ...
```

8. Define a function to find the positions of the eyes in an input image using the shape predictor.
9. Define a function to crop the input image to a square region centered around the face, considering the position of the eyes.
10. Set the URL of the pre-trained StyleGAN2 network and configure other parameters such as the number of optimization steps, frames per second (FPS), and freeze steps.
11. Load the pre-trained StyleGAN2 generator network.
12. Read the source image and perform cropping using the defined function.
13. Run the StyleGAN2 projector on the cropped source image to obtain the latent vector (lvec1).
14. Read the 5 target images and perform cropping.
15. Run the StyleGAN2 projector on the cropped target image to obtain the latent vector (lvec2).



16. Calculate the difference and step size between lvec1 and lvec2.
17. Generate a video by gradually morphing the source image into the target image using the StyleGAN2 generator network and the calculated step size.
18. Save the generated video as an MP4 file.
19. Optionally, allow the user to upload additional target images and repeat the process.
20. Display the source and target images, as well as the generated video.

[Video Results Drive Link]
[Colab Link Question 2]

References:-
https://towardsdatascience.com/generating-your-own-images-with-nvidia-stylegan2-ada-for-pytorch-on-ampere-a80fab52d6b5
https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html#:~:text=What%20is%20a%20DCGAN%3F,first%20described%20by%20Radford%20et.
https://www.youtube.com/watch?v=5fs9PMzrVig