NORTHEASTERN UNIVERSITY

INFORMATION SYSTEMS SPRING '21

Parallel Deep Learning Neural Networks and GAN to detect features on CelebFaces Dataset

Parallel Machine Learning & AI

CSYE7105–37380

Guided By:

Prof. Handan Liu

Created By:

Team- 5

Hemant Jain

ACKNOWLEDGEMENT

It gives me a great pleasure and immense satisfaction in presenting the project "***Parallel Deep Learning Neural Networks and GAN to detect features on CelebFaces Dataset***" in regard to Parallel Machine Learning and AI coursework at Northeastern University in Spring 2021.

I would like to take this opportunity to acknowledge the innumerable guidance and support extended to me by our **Professor Handan Liu** in the execution and preparation of the project.

I would like to thank our **TA Mr. Chaoyi Yuan** who has helped me with his valuable suggestions and guidance. Finally, I am intended to our very own Northeastern University for giving us the platform to express and exhibit our talent.

I would also like to thank Professor for giving me an opportunity this semester to attend the Nvidia GTC 2021 - GPU Technology Conference and listening and seeing the active contributions in the GPU world and high parallel processing computing technology.

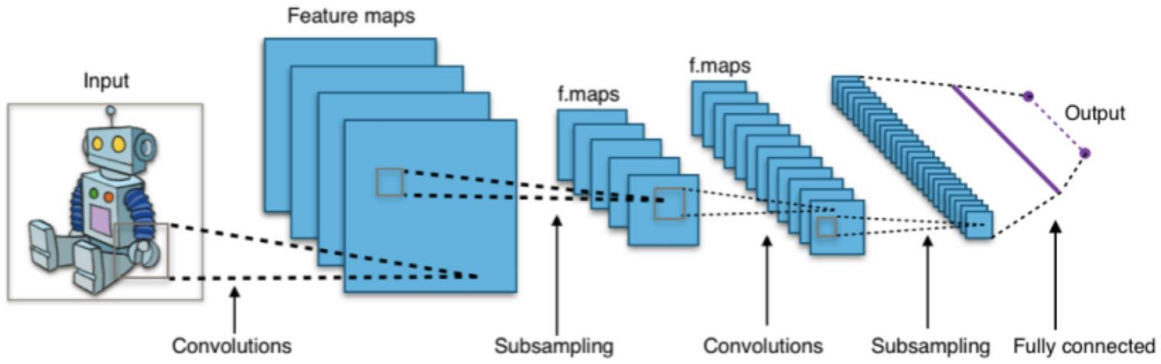
Any bouquets for the merit of this project should go to above mentioned persons

Introduction

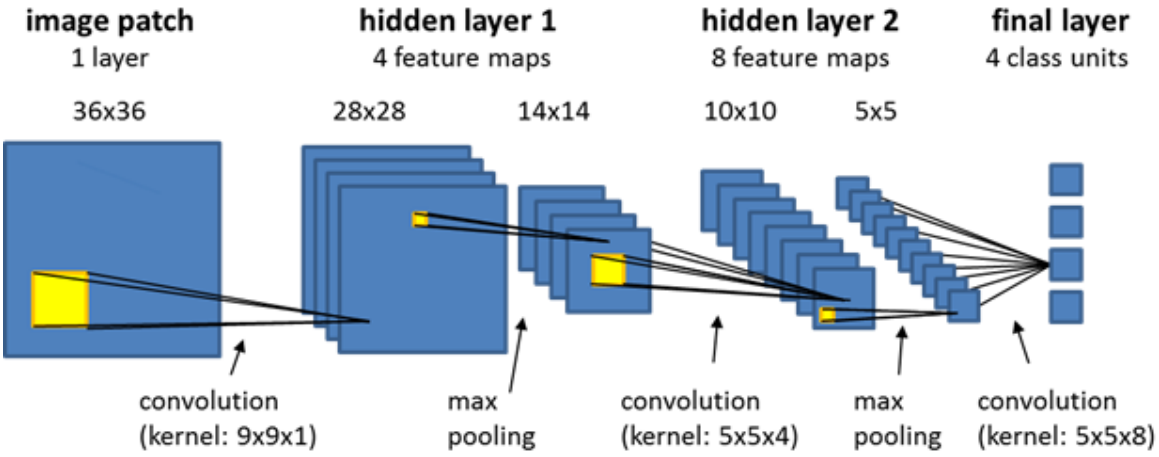
I. Background

The 2 methodologies I came across and decided to move forward with for our analysis are the following:

- Convolutional Neural Network(CNN): It is a type of neural network that allows for extraction of higher representation of image content. CNN takes images in raw pixel data, trains the model , then extracts the features from it and gives a better classification output

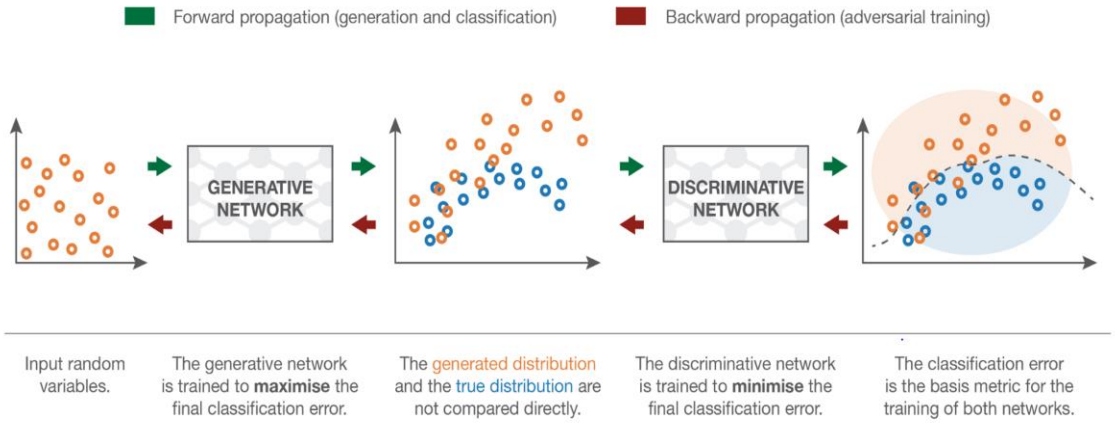


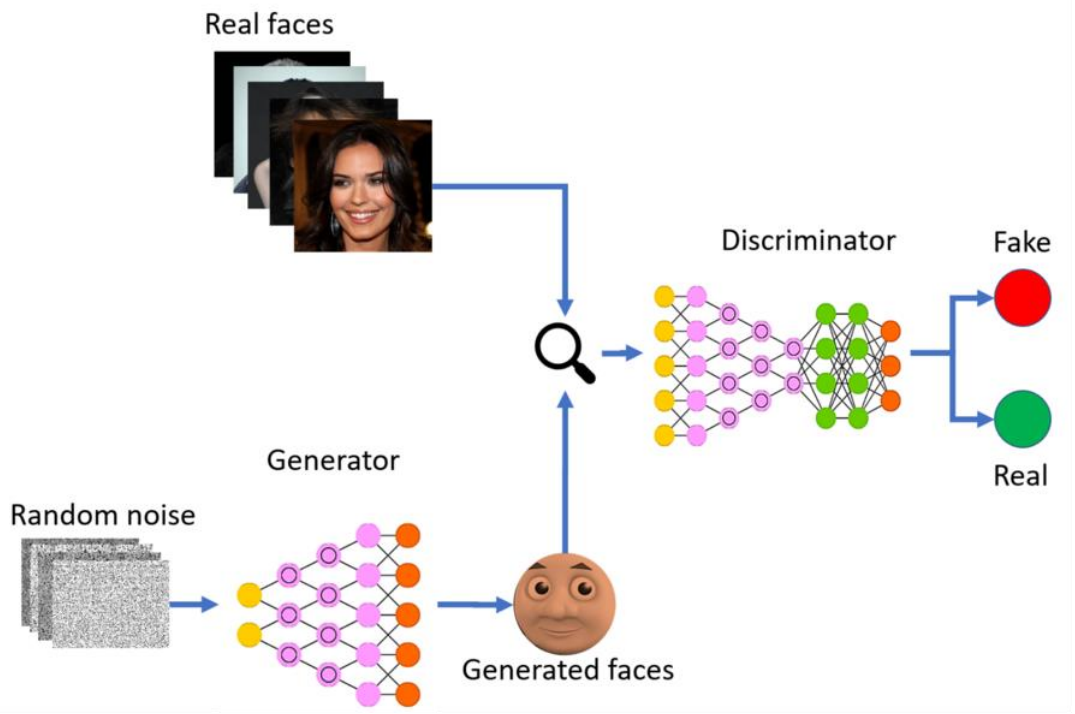
CNN architectures with convolutions, pooling (subsampling), and fully connected layers for softmax activation function



- Generative adversarial network(GAN): which is about creating images; like drawing a portrait. The main focus of GAN's is to generate data from scratch, which are mostly images. It is a neural network architecture with a game theoretic approach, the 2 model components are generator and discriminator. The role of the generator is to fool the

discriminator with fake images and the discriminator learn how not to get fooled and identify the fake images





II. Motivation

In recent years, the amount of data being used and created by companies and people has grown exponentially. This has led many avenues to come out to the forefront that make use of data to construct models and systems to further optimize a company's services and provide a better consumer experience.

Deep learning is a popular tool that has grown tremendously and has applications in many real-life scenarios. These models are used to create artificial intelligence in a system to perform a certain human task and learn through past data. In our project I focused on using two such methods: Convolutional Neural Networks, and Generative Adversarial Networks.

Convolutional Neural Networks is one such deep learning tool which is primarily used on image data and consists of having layers upon layers of nodes with multiple channels and a convolution mask to analyze the pixel values of each image and to learn and accordingly adjust the weights of the model.

GAN's is an unsupervised method of learning and is primarily used for data generation. Our project group decided to work with GAN's to get a better understanding of how to create such a network.

III. Goals:

The goal of this project is to make predictions on the celebrity face dataset using deep learning algorithms. I was set to predict the hair features for the celebrities and guess their gender using Convolutional Neural Network (CNN). I also decided to generate fake faces using Generative adversarial network (GAN) and building and deploying the machine learning model in the streamlit based web application where you can generate fake faces in real-time.

Methodology:

With the goal set for the project to make predictions for the celebrity images dataset. I decided to move forward with 2 methods

a. Convolutional Neural Network (CNN)
b. Generative adversarial network (GAN)
c. Streamlit based web app for fake celebrity face generation

Using Pytorch and parallelizing it using cuda on multiple GPU's

Step 1: Gender Prediction with CNN on Pytorch

- Exploratory Data Analysis:
  a. The attribute csv file had the column values of -1 and 1 which I change to 0 and 1 for better prediction

  b. The partition csv file had all the training, validation, and test data in one with values 0,1 and 2 for identification which I split into 3 different files.

- With using a custom dataset in Pytorch I had to construct our own class for loading this dataset.

  This required three functions:
  1) __init__: Used to initialize the csv, the image directory, and the transforms
  2) __len__: Returns the length of the images within the csv
  3) __getitem__: Joins the paths images with their respective labels and returns the values

```
class CelebTrain(Dataset):
  def __init__(self, csv_file,img_dir, transform=None):
    self.img_dir = img_dir
    self.annotations = pd.read_csv(csv_file,nrows = 5120)
    self.transform = transform

  def __len__(self):
      return len(self.annotations)

  def __getitem__(self, index):
      img_path = os.path.join(self.img_dir, self.annotations.iloc[index,0])
      img = Image.open(img_path)
      y_label = torch.tensor(int (self.annotations.iloc[index,1]))

      if self.transform:
        img = self.transform(img)

        return (img, y_label)
```

- With CNN I performed image classification of the celebrity images predicting the gender they belong to using the gender column in the attribute csv file of our dataset.
  Below is the code snippet of the CNN model for gender prediction

```
  class GenderModel(nn.Module):
    def __init__(self):
      super().__init__()
      self.conv1 = nn.Conv2d(3,32,3, stride = 1)
      self.batch1 = nn.BatchNorm2d(32)
      self.pool1 = nn.MaxPool2d(2,2)
      self.conv2 = nn.Conv2d(32,64,5,stride=1)
      self.batch2 = nn.BatchNorm2d(64)
      self.pool2 = nn.MaxPool2d(2,2)
      self.conv3 = nn.Conv2d(64,128,3,stride=1)
      self.batch3 = nn.BatchNorm2d(128)
      self.pool3 = nn.MaxPool2d(2,2)
      self.conv4 = nn.Conv2d(128, 128, 5, stride=1)
      self.batch4 = nn.BatchNorm2d(128)
      self.pool4 = nn.MaxPool2d(2,2)
      self.fc1 = nn.Linear(128 * 8 * 8 ,1024)
      self.drop1 = nn.Dropout(p=0.1)
      self.fc2 = nn.Linear(1024,128)
      self.drop2 = nn.Dropout(p=0.1)
      self.fc3 = nn.Linear(128,2)

    def forward(self,x):
      #x1  =    self.conv1(x)
      x    =    self.pool1(F.relu(self.batch1((self.conv1(x)))))
      x    =    self.pool2(F.relu(self.batch2((self.conv2(x)))))
      x    =    self.pool3(F.relu(self.batch3((self.conv3(x)))))
      x    =    self.pool4(F.relu(self.batch4((self.conv4(x)))))
      x    =    x.view(-1, 128 * 8 * 8)
      x    =    self.drop1(x)
      x    =    F.relu(self.fc1(x))
      x    =    self.drop2(x)
      x    =    F.relu(self.fc2(x))
      x    =    self.fc3(x)
      return x
  #net = GenderModel()
  print(GenderModel())
```

- Designed the CNN model for gender prediction with 4 layers which was able to learn on the training dataset and predict the correct gender when tested on the test dataset with an accuracy of 94%

The model was trained using a train function

```
def train():
    optimizer = torch.optim.SGD(model.parameters(),lr = 0.001, momentum=0.9)
    criterion = nn.CrossEntropyLoss()
    n_samples = 10240
    total_loss = 0
    total_train = 0
    correct_train = 0.0
    running_loss = 0
    for i,(img, labels) in enumerate(train_loader):
        img = img.to(device)
        labels = labels.to(device)
        optimizer.zero_grad()
        output = model(img)
        loss = criterion(output, labels)
        loss.backward()
        optimizer.step()
        total_loss = total_loss + loss.item()
        #accuracy
        _, predicted = torch.max(output.data, 1)
        predicted = predicted
        total_train += labels.size(0)
        correct_train += predicted.eq(labels.data).sum().item()
        train_accuracy = 100 * correct_train / total_train


    running_loss += loss.item() * img.size(0)

    epoch_loss = 100* running_loss/total_train
    losses.append(epoch_loss)
    return (train_accuracy,loss.item())
```

The model was validated running the epochs on the validation dataset

```
def valid(epochs):
    model.eval()
    with torch.no_grad():
        correct = 0
        total = 0
        with torch.no_grad():
            for i,(img, labels) in enumerate(valid_loader):
                # if i>=n_samp:
                #   break
                img = img.to(device)
                labels = labels.to(device)
                outputs = model(img)
                _, predicted = torch.max(outputs.data, 1)
                predicted = predicted
                total += labels.size(0)
                correct += (predicted == labels).sum().item()
                acc = 100*correct/total
    return acc
```

Tested the model through a Predict function

```
def predict(img_test_dir):
    model.eval()
    #Loading the model
    model1 = torch.load('/content/model.pt',map_location = 'cpu')
    print(model1)
    #Loadind the test image
    img = Image.open(img_test_dir)
    with torch.no_grad():
        trans1 = transforms.Compose([transforms.Resize(178),transforms.CenterCrop(178)
                                                        ,transforms.ToTensor()])
        img_tensor = trans1(img) #shape = [3, 32, 32]
        #Image Transformation
        trans = transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5])
        img_tensor = trans(img_tensor)
        print(img_tensor.shape)

        single_image_batch = img_tensor.unsqueeze(0) #shape = [1, 3, 32, 32]
        outputs = model1(single_image_batch)
        _, predicted = torch.max(outputs.data, 1)
        class_id = predicted[0].item()
        predicted_class = classes[predicted[0].item()]
        print("Predicted Class : {}".format(predicted_class))
        display(img)
```

Step 2: Hair feature Prediction Using CNN with Pytorch

- **Exploratory Data Analysis**:
  a. The attribute file had the values with -1 and 1 for the 4 hair colors which I changed to 0 and 1 for better prediction
  b. The partition csv file had all the training, validation, and test data in one with values 0,1 and 2 for identification which I split into 3 different files.
  c. The data was not accurate with incorrect values for hair colors as many had 0 value for all the colors, which was removed using the below code and saved into a new csv file.

```
import csv
updatedlist=[]
with open("train.csv",newline="") as f:
    reader=csv.reader(f)
    for row in reader:
        if '1' in row:
            updatedlist.append(row)

```

```
len(updatedlist)
```

```
100868
```

```
with open('output_train.csv','w',newline='') as f:
    writer = csv.writer(f)
    writer.writerow(['image_id', 'Black_Hair', 'Blond_Hair','Brown_Hair','Gray_Hair'])
    writer.writerows(updatedlist)
```

  d. The 0 and 1 values of different columns were difficult to make accurate predictions, to overcome this problem a new hair color class was created.

	image_id	Black_Hair	Blond_Hair	Brown_Hair	Gray_Hair	Hair_color
0	000001.jpg	0	0	1	0	0100
1	000002.jpg	0	0	1	0	0100
2	000006.jpg	0	0	1	0	0100
3	000007.jpg	1	0	0	0	1000
4	000008.jpg	1	0	0	0	1000

And the assigned proper numbers to the colors as 0,1,2 and 3

```
##Blond=0
df_train = df_train.replace(to_replace ='0010', value=0)
```

```
##Black=2
df_train = df_train.replace(to_replace ='1000', value=1)
```

```
##Brown=3
df_train = df_train.replace(to_replace ='0100', value=2)
```

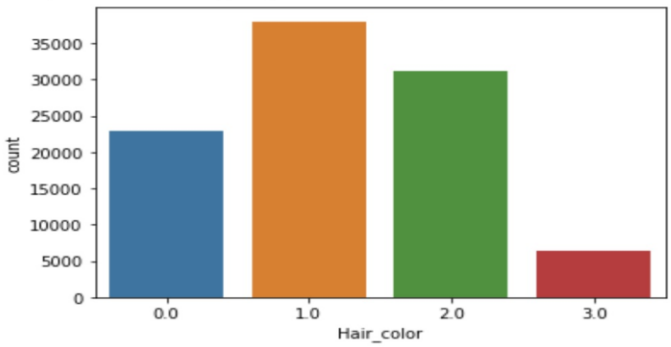
```
##Gray=4
df_train = df_train.replace(to_replace ='0001', value=3)
```

- The second part for the classification I performed hair color prediction for the celebrities with the 4 different hair colors from the data i.e. Black, Brown, Blond and Gray.
  Below is the model design for Hair feature prediction

	image_id	Black_Hair	Blond_Hair	Brown_Hair	Gray_Hair	Hair_color
0	000001.jpg	0	0	1	0	2
1	000002.jpg	0	0	1	0	2
2	000006.jpg	0	0	1	0	2
3	000007.jpg	1	0	0	0	1
4	000008.jpg	1	0	0	0	1
5	000011.jpg	1	0	0	0	1
6	000012.jpg	1	0	0	0	1

```
#training images label distribution
sns.countplot(x='Hair_color',data = train_img)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7fc61faf3588>
```



- The model learned through the train and validation data split and was able to give the correct results for the test data for the model predicting the hair color for the images with an accuracy of 93%

The train and validation images dataset were read using the below function.

```
class CelebTrain(Dataset):
  def __init__(self, csv_file,img_dir, transform=None):
    self.img_dir = img_dir
    self.annotations = csv_file
    self.transform = transform

  def __len__(self):
      #print(len(self.annotations))
      return len(self.annotations)

  def __getitem__(self, index):
      img_path = os.path.join(self.img_dir, self.annotations.iloc[index,0])
      #img_name = self.img_path[index]
      img = Image.open(img_path)
      y_label = torch.tensor(int (self.annotations.iloc[index,-1]))

      if self.transform:
        img = self.transform(img)

        return (img, y_label)

class CelebValid(Dataset):
  def __init__(self, csv_file,img_dir, transform=None):
    self.img_dir = img_dir
    self.annotations = csv_file
    self.transform = transform

  def __len__(self):
      #print(len(self.annotations))
      return len(self.annotations)

  def __getitem__(self, index):
      img_path = os.path.join(self.img_dir, self.annotations.iloc[index,0])
      #img_name = self.img_path[index]
      img = Image.open(img_path)
      y_label = torch.tensor(int (self.annotations.iloc[index,-1]))

      if self.transform:
        img = self.transform(img)

        return (img, y_label)
```

For loading the dataset, I used data loader and then transformed it to tensors with their labels

```
# Creating tensors for the images and their labels
train_data = CelebTrain(csv_file = train_img, img_dir = images_folder,
                        transform = transforms.Compose([transforms.Resize(178),transforms.CenterCrop(178)
                                                        ,transforms.ToTensor()]))

valid_data = CelebValid(csv_file = valid_img, img_dir = images_folder,
                        transform = transforms.Compose([transforms.Resize(178),transforms.CenterCrop(178),
                                                         transforms.ToTensor()]))
```

With this I resized the size of the image as it was not in proportion so to make it equal size, I transformed the image to 178 x 178

The training and the validation functions were same for both the CNN models

Functions used in the CNN models (Gender and Hair)

a. **Activation Function**: ReLu as this is more computationally efficient and has better convergence .
b. **Cost Function**: SGD as it is easier to fit into memory and the convergence is faster with larger datasets.
c. **Loss Function**: Cross Entropy as it is good to work with classification problems
d. Dimensionality Reduction using MaxPool
e. **Dropout** is good to avoid over memorization thus reducing the complexity
f. **Batch Normalization** is good to check if each node is able to learn.

Step 3: Generating fake faces using Generative adversarial network(GAN) with Pytorch

GAN is composed of two models:

- The first is the Generator model that has an aim to generate new data similar to the original input data. The Generator can be compared to a human art forger, who creates fake works of art. Designed the generator model to generate fake images for the dataset of celebrities

```
def deconv(ch_in, ch_out, k_size, stride=2, pad=1, bn=True):
    layers = []
    layers.append(nn.ConvTranspose2d(ch_in, ch_out, k_size, stride, pad))
    if bn:
        layers.append(nn.BatchNorm2d(ch_out))
    return nn.Sequential(*layers)

class Generator(nn.Module):
    def __init__(self, ngpu):
        super(Generator, self).__init__()
        self.ngpu = ngpu
        self.main = nn.Sequential(
            # input is Z, going into a convolution
            nn.ConvTranspose2d( nz, ngf * 8, 4, 1, 0, bias=False),
            nn.BatchNorm2d(ngf * 8),
            nn.ReLU(True),
            # state size. (ngf*8) x 4 x 4
            nn.ConvTranspose2d(ngf * 8, ngf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 4),
            nn.ReLU(True),
            # state size. (ngf*4) x 8 x 8
            nn.ConvTranspose2d( ngf * 4, ngf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 2),
            nn.ReLU(True),
            # state size. (ngf*2) x 16 x 16
            nn.ConvTranspose2d( ngf * 2, ngf, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf),
            nn.ReLU(True),
            # state size. (ngf) x 32 x 32
            nn.ConvTranspose2d( ngf, nc, 4, 2, 1, bias=False),
            nn.Tanh()
            # state size. (nc) x 64 x 64
        )
    def forward(self, input):
        return self.main(input)
```

- The second is the Discriminator model whose end goal is to identify that the input data entered is original that is the same as the original dataset or if it is the fake data that the generator has created. The Discriminator is the same as one of the art experts, trying to predict the original and fake. The designed discriminator model for this project was able to predict the real and fake images separately for the celebrity images of the dataset

```
def conv(ch_in, ch_out, k_size, stride=2, pad=1, bn=True):
    layers = []
    layers.append(nn.Conv2d(ch_in, ch_out, k_size, stride, pad))
    if bn:
        layers.append(nn.BatchNorm2d(ch_out))
    return nn.Sequential(*layers)


class Discriminator(nn.Module):
    def __init__(self, ngpu):
        super(Discriminator, self).__init__()
        self.ngpu = ngpu
        self.main = nn.Sequential(
            # input is (nc) x 64 x 64
            nn.Conv2d(nc, ndf, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf) x 32 x 32
            nn.Conv2d(ndf, ndf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 2),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf*2) x 16 x 16
            nn.Conv2d(ndf * 2, ndf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 4),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf*4) x 8 x 8
            nn.Conv2d(ndf * 4, ndf * 8, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 8),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf*8) x 4 x 4
            nn.Conv2d(ndf * 8, 1, 4, 1, 0, bias=False),
            nn.Sigmoid()
        )
    def forward(self, input):
        return self.main(input)
```

Functions used in the GAN Model

a. **Activation Function**: Sigmoid and ReLu as this is more computationally efficient and has better convergence.
b. **Batch Normalization** is good to check if each node can learn.
c. **Loss Function**: BCE Loss (Binary Cross Entropy) as it needs only one node to classify the data into 2 classes.

Step 4: Parallelizing the models with multiple GPU's using Cuda Module

- **Parallelizing CNN gender model and hair feature model on 1,2 and 4 GPU's**

```
if torch.cuda.is_available():
    device = torch.device("cuda:0")
    print("Running on the GPU")
else:
    device = torch.device("cpu")
    print("Running on the CPU")

if torch.cuda.device_count() >= 1:
    print("Let's use", torch.cuda.device_count(), "GPUs!")
    model = nn.DataParallel(model1)

model.to(device)
```

- **Parallelizing GAN on Multiple GPU's**

```
device = torch.device("cuda:0" if (torch.cuda.is_available() and ngpu > 0) else "cpu")
```

Generator

```
# Handle multi-gpu if desired
if (device.type == 'cuda') and (ngpu > 1):
    G = nn.DataParallel(netG, list(range(ngpu)))

# Apply the weights_init function to randomly initialize all weights
#  to mean=0, stdev=0.2.
G.apply(weights_init)

# Print the model
print(G)
```

Discriminator

```
# Handle multi-gpu if desired
if (device.type == 'cuda') and (ngpu > 1):
    D = nn.DataParallel(D, list(range(ngpu)))

# Apply the weights_init function to randomly initialize all weights
#  to mean=0, stdev=0.2.
D.apply(weights_init)

# Print the model
print(D)
```

Description of the dataset

Source Dataset:

https://www.kaggle.com/jessicali9530/celeba-dataset

Dataset Description

Context

A popular component of computer vision and deep learning revolves around identifying faces for various applications from logging into your phone with your face or searching through surveillance images for a particular suspect. This dataset is great for training and testing models for face detection, particularly for recognizing facial attributes such as finding people with brown hair, are smiling, or wearing glasses.

Images cover large pose variations, background clutter, diverse people, supported by a large quantity of images and rich annotations. This data was originally collected by researchers at MMLAB, The Chinese University of Hong Kong (specific reference in Acknowledgment section).

CelebFaces Attributes Dataset (CelebA) is a large-scale face attributes dataset with more than **200K** celebrity images, each with **40** attribute annotations. The images in this dataset cover large pose variations and background clutter. CelebA has large diversities, large quantities, and rich annotations, including

- **10,177** number of **identities**,
- **202,599** number of **face images**, and
- **5 landmark locations**, **40 binary attributes** annotations per image.

The dataset can be employed as the training and test sets for the following computer vision tasks: face attribute recognition, face detection, landmark (or facial part) localization, and face editing & synthesis.

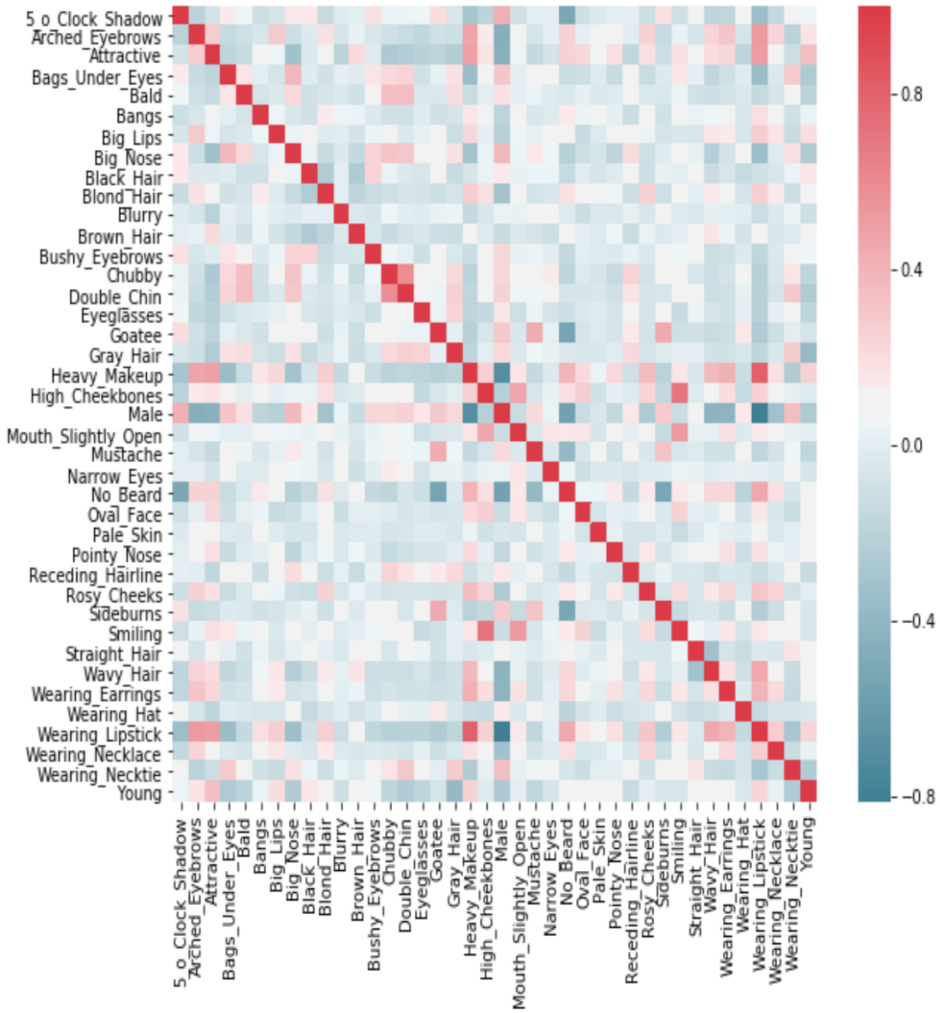
Data Files

- img*align*celeba.zip: All the face images, cropped and aligned
- list*eval*partition.csv: Recommended partitioning of images into training, validation, testing sets. Images 1-162770 are training, 162771-182637 are validation, 182638-202599 are testing
- list*bbox*celeba.csv: Bounding box information for each image. "x*1" and "y*1" represent the upper left point coordinate of bounding box. "width" and "height" represent the width and height of bounding box
- list*landmarks*align_celeba.csv: Image landmarks and their respective coordinates. There are 5 landmarks: left eye, right eye, nose, left mouth, right mouth
- list*attr*celeba.csv: Attribute labels for each image. There are 40 attributes. "1" represents positive while "-1" represents negative

Heatmap Plot for the attributes in the dataset

```
f, ax = plt.subplots(figsize=(10, 8))
corr = df1.corr()
sns.heatmap(corr, mask=np.zeros_like(corr, dtype=np.bool), cmap=sns.diverging_palette(220, 10, as_cmap=True),
            square=True, ax=ax)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x29705c89b48>
```

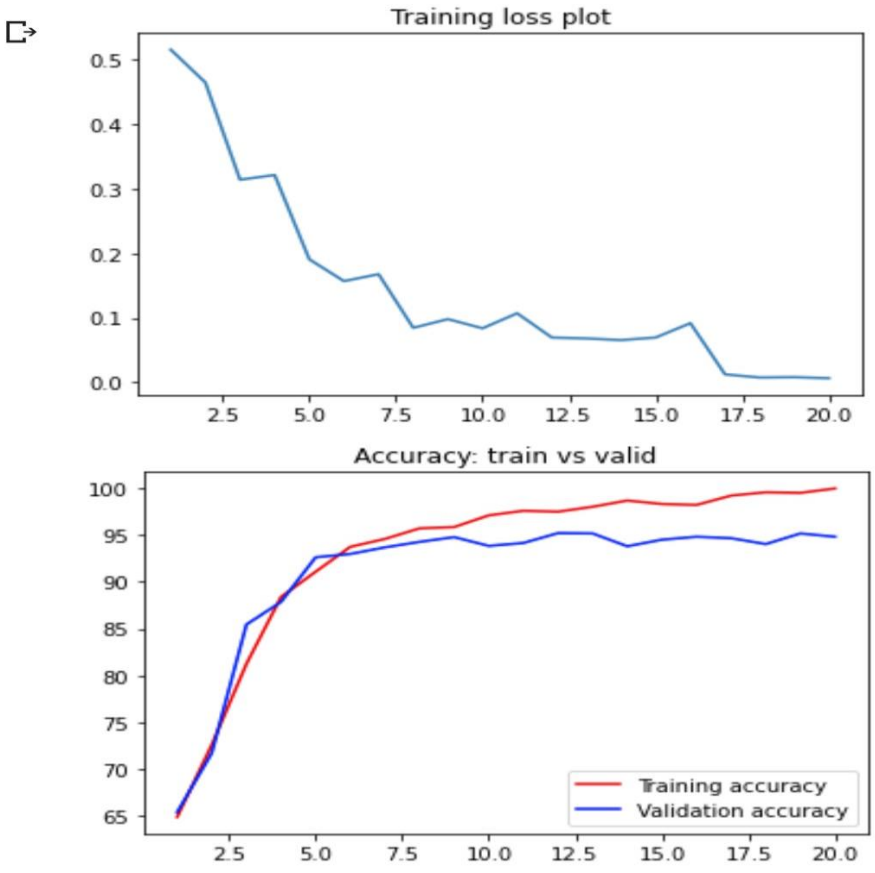


Result and Analysis

- **Gender Model Prediction with CNN Model execution:**

```
GenderModel(
  (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1))
  (batch1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): Conv2d(32, 64, kernel_size=(5, 5), stride=(1, 1))
  (batch2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1))
  (batch3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (pool3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv4): Conv2d(128, 128, kernel_size=(5, 5), stride=(1, 1))
  (batch4): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (pool4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=8192, out_features=1024, bias=True)
  (drop1): Dropout(p=0.1, inplace=False)
  (fc2): Linear(in_features=1024, out_features=128, bias=True)
  (drop2): Dropout(p=0.1, inplace=False)
  (fc3): Linear(in_features=128, out_features=2, bias=True)
`
```

The Plot for loss and Accuracy:



The final outcomes from the Gender model:

```
torch.Size([3, 178, 178])
Predicted Class : Female
```

```
)
torch.Size([3, 178, 178])
Predicted Class : Female
```

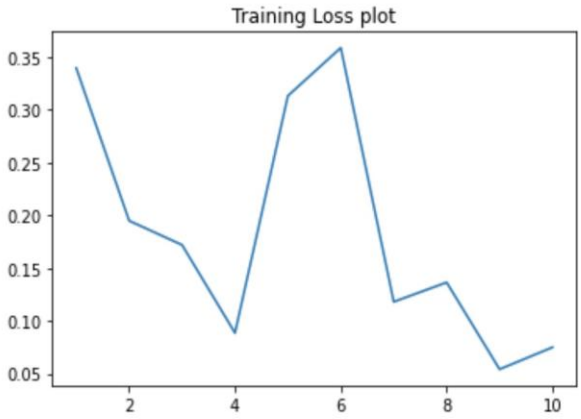
```
torch.Size([3, 178, 178])
Predicted Class : Male
```

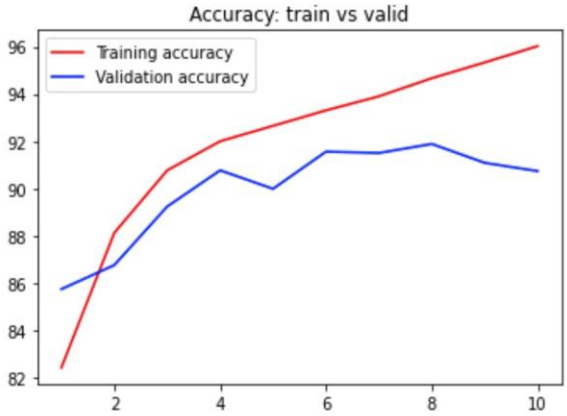
```
torch.Size([3, 178, 178])
Predicted Class : Male
```

- **Hair Feature Prediction with CNN Model Execution:**

```
Feature_Model(
  (conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1))
  (batch1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (pool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): Conv2d(32, 64, kernel_size=(5, 5), stride=(1, 1))
  (batch2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (pool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1))
  (batch3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (pool3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv4): Conv2d(128, 128, kernel_size=(5, 5), stride=(1, 1))
  (batch4): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (pool4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=8192, out_features=1024, bias=True)
  (drop1): Dropout(p=0.4, inplace=False)
  (fc2): Linear(in_features=1024, out_features=128, bias=True)
  (drop2): Dropout(p=0.25, inplace=False)
  (fc3): Linear(in_features=128, out_features=4, bias=True)
)
```

The Plot of loss and accuracy :





The final outcomes from the model:

)
torch.Size([3, 178, 178])
Predicted Class : Black Hair

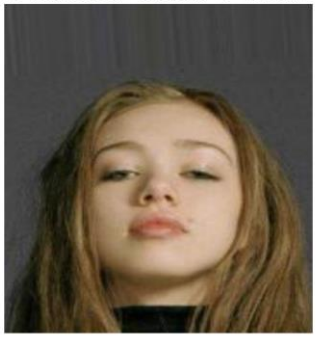
torch.Size([3, 178, 178])
Predicted Class : Brown Hair

```
torch.Size([3, 178, 178])
Predicted Class : Black Hair
```

```
torch.Size([3, 178, 178])
Predicted Class : Black Hair
```

```
torch.Size([3, 178, 178])
Predicted Class : Blond Hair
```

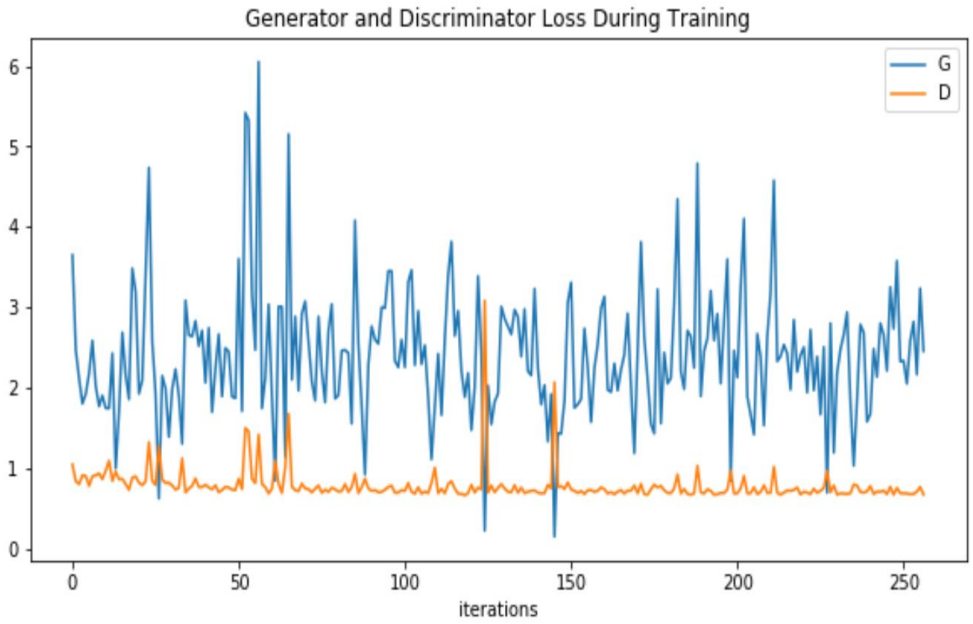
```
torch.Size([3, 178, 178])
Predicted Class : Brown Hair
```



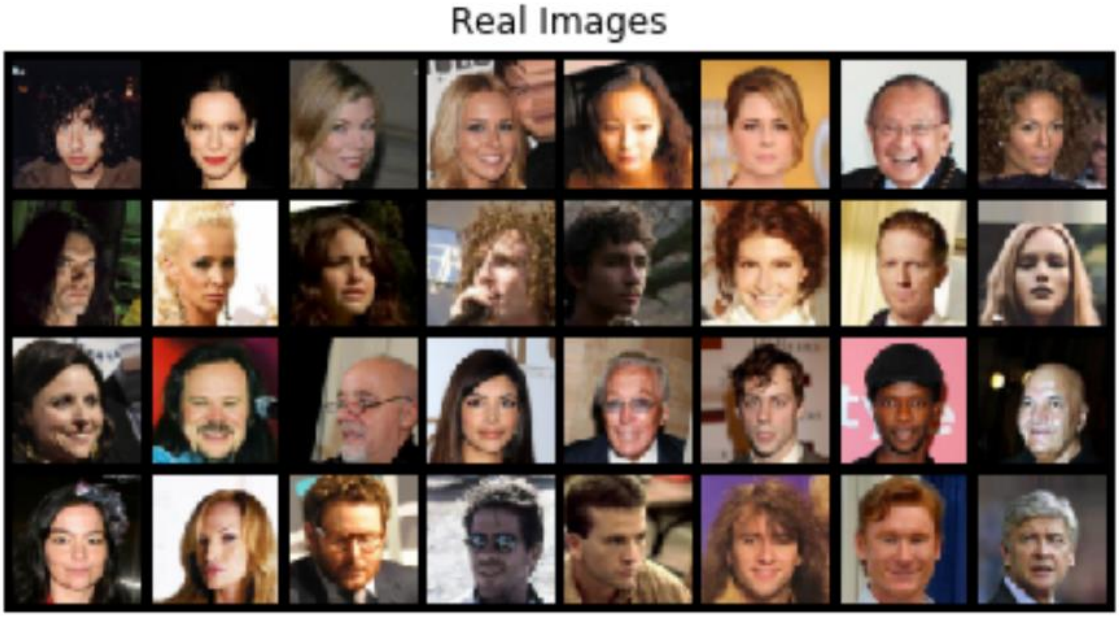
- **GAN Results Model execution:**

```
(ay20) [deval.a@d1001 Project]$ python GANS_final.py
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()
  )
)
Discriminator(
  (main): Sequential(
    (0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): LeakyReLU(negative_slope=0.2, inplace=True)
    (2): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (4): LeakyReLU(negative_slope=0.2, inplace=True)
    (5): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (6): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (7): LeakyReLU(negative_slope=0.2, inplace=True)
    (8): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (9): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (10): LeakyReLU(negative_slope=0.2, inplace=True)
    (11): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (12): Sigmoid()
  )
)
```

The plot of Losses for generator and discriminator:



Real Images:



Fake Images :

Data Parallelism:

The models were parallelized on Data parallelism on the cluster with multiple GPU's.

Starting with requesting resource on the cluster and additional memory of 4GB

```
csye/374-HemantJain  hpl-2.3.tar.gz  hw1.err  mpihello  omp_helloc  ondemand    week7
[jain.he@login-00 ~]$ srun --partition=reservation --reservation=csye7105v100 --mem=4GB --gres=gpu:4 --time=1:00:00 --export=All --pty /bin/bash
[jain.he@d1009 ~]$ module load cuda/10.2
```

- **Gender Model using CNN with pytorch and parallelizing it using Cuda device on 1,2 and 4 GPU's**
  While running on 1 GPU I obtained training accuracy of 91% and validation accuracy of 92% and a total time of 92.65 seconds

```
)
Let's use 1 GPUs!
Training for 5 epochs
Epoch 0, train Loss: 0.589 Training Accuracy 60.48828125: Valid Accuracy 64.9609375
GPU time:  19.61354997754097
Epoch 1, train Loss: 0.478 Training Accuracy 69.53125: Valid Accuracy 75.078125
GPU time:  17.895583912730217
Epoch 2, train Loss: 0.335 Training Accuracy 81.25: Valid Accuracy 86.5625
GPU time:  17.781820558011532
Epoch 3, train Loss: 0.236 Training Accuracy 86.15234375: Valid Accuracy 89.140625
GPU time:  18.08343929424882
Epoch 4, train Loss: 0.144 Training Accuracy 91.50390625: Valid Accuracy 92.4609375
GPU time:  19.1458952203393
Model successfully saved at ./model/
Time taken for 5 epochs is 92.65408503636718 seconds
```

On 2 GPU's the computing time reduced by 14 seconds and became 78.35 seconds and the training accuracy was 91% and validation accuracy came out to be 93%

```
)
Let's use 2 GPUs!
Training for 5 epochs
Epoch 0, train Loss: 0.572 Training Accuracy 63.5546875: Valid Accuracy 69.6875
GPU time:  18.169932890683413
Epoch 1, train Loss: 0.494 Training Accuracy 72.890625: Valid Accuracy 80.1171875
GPU time:  14.730295155197382
Epoch 2, train Loss: 0.311 Training Accuracy 82.3828125: Valid Accuracy 86.4453125
GPU time:  14.703572392463684
Epoch 3, train Loss: 0.215 Training Accuracy 89.70703125: Valid Accuracy 92.1484375
GPU time:  14.697187270969152
Epoch 4, train Loss: 0.151 Training Accuracy 91.171875: Valid Accuracy 93.046875
GPU time:  15.91193775832653
Model successfully saved at ./model/
Time taken for 5 epochs is 78.35718268156052 seconds
```

On 4 GPU's the reduced by 6 seconds from the 2 GPU computing time, and I noted it as 72.73 seconds. The training and validation accuracies were 92%

```
Let's use 4 GPUs!
Training for 5 epochs
Epoch 0, train Loss: 0.563 Training Accuracy 63.4375: Valid Accuracy 64.6875
GPU time:  19.318857178092003
Epoch 1, train Loss: 0.462 Training Accuracy 74.27734375: Valid Accuracy 82.6953125
GPU time:  12.9562974460423
Epoch 2, train Loss: 0.236 Training Accuracy 85.56640625: Valid Accuracy 88.828125
GPU time:  13.051421418786049
Epoch 3, train Loss: 0.209 Training Accuracy 88.8671875: Valid Accuracy 92.34375
GPU time:  12.993275303393602
Epoch 4, train Loss: 0.165 Training Accuracy 92.36328125: Valid Accuracy 92.734375
GPU time:  14.272557120770216
Model successfully saved at ./model/
Time taken for 5 epochs is 72.73714835196733 seconds
```

- **Hair Feature Model using CNN with pytorch and parallelizing it using Cuda device on 1,2 and 4 GPU's**

On 1 GPU the time taken was the highest , 2331.72 seconds with the accuracies of 92% and 90% for training and validation respectively.

```
)
Let's use 1 GPUs!
Training for 5 epochs
Epoch 0, train Loss: 0.220 Training Accuracy 83.76303577412443: Valid Accuracy 88.25229092710312
GPU time:  485.37957256659865
Epoch 1, train Loss: 0.333 Training Accuracy 87.0480000812356: Valid Accuracy 88.64030380582845
GPU time:  455.2224797680974
Epoch 2, train Loss: 0.077 Training Accuracy 90.5187908081926: Valid Accuracy 90.45653430198959
GPU time:  451.6319374963641
Epoch 3, train Loss: 0.122 Training Accuracy 91.74138648848994: Valid Accuracy 90.25840006604474
GPU time:  457.8565755710006
Epoch 4, train Loss: 0.251 Training Accuracy 92.39939479482935: Valid Accuracy 90.48955667464708
GPU time:  481.373471096158
Model successfully saved
Time taken for 5 epochs is 2331.7264383137226 seconds
```

Running on 2 GPU's I encountered the accuracies to be same as fo 1 GPU ie 92% and 90% for training and validation respectively but the total computing time reduced to 1912.40 seconds

```
Let's use 2 GPUs!
Training for 5 epochs
Epoch 0, train Loss: 0.214 Training Accuracy 82.99332852689406: Valid Accuracy 89.49888549492282
GPU time:  375.40834134072065
Epoch 1, train Loss: 0.378 Training Accuracy 89.51248489525686: Valid Accuracy 89.94468752579873
GPU time:  382.66658359766006
Epoch 2, train Loss: 0.286 Training Accuracy 91.33419307669655: Valid Accuracy 89.30900685214233
GPU time:  386.0673277080059
Epoch 3, train Loss: 0.282 Training Accuracy 92.05109718823303: Valid Accuracy 90.72896887641377
GPU time:  377.84959238767624
Epoch 4, train Loss: 0.074 Training Accuracy 92.97921384254511: Valid Accuracy 90.89408073970115
GPU time:  390.16249491274357
Model successfully saved
Time taken for 5 epochs is 1912.4015935249627 seconds
```

Running on 4 GPU's the computing time was 1711.91 seconds with training accuracy as 92% and validation accuracy as 91%

```
)
Let's use 4 GPUs!
Training for 5 epochs
Epoch 0, train Loss: 0.438 Training Accuracy 82.74962174676834: Valid Accuracy 88.26880211343185
GPU time:  344.596165638417
Epoch 1, train Loss: 0.427 Training Accuracy 88.62295514779801: Valid Accuracy 89.70527532403203
GPU time:  339.9015106037259
Epoch 2, train Loss: 0.251 Training Accuracy 90.96254023700484: Valid Accuracy 90.4813010814827
GPU time:  342.1115803979337
Epoch 3, train Loss: 0.161 Training Accuracy 92.02571106530327: Valid Accuracy 90.87756955337241
GPU time:  342.55778378248215
Epoch 4, train Loss: 0.144 Training Accuracy 92.70199738015211: Valid Accuracy 91.25732683893338
GPU time:  342.4459239318967
Model successfully saved
Time taken for 5 epochs is 1711.9142694212496 seconds
```

- **GAN model parallelism on cuda device on 1,2 and 4 GPU's**

For 1 GPU the computing time of each epoch was 9500 seconds and the resultant loss function for Generator was 3.03 and Discriminator was 0.4

```
[jain.he@login-00 Project]$ source activate py37
(py37) [jain.he@login-00 Project]$ srun --partition=reservation --reservation=csye7105v100 --mem=4GB --gres=gpu:1 --time=1:00:00 --export=All --pty /bin/bash
[jain.he@d1006 Project]$ module load python/3.7.0
[jain.he@d1006 Project]$ module list
Currently Loaded Modulefiles:
  1) discovery/2019-02-21   2) anaconda3/2020.02      3) cuda/10.2             4) python/3.7.0
[jain.he@d1006 Project]$ ls
celebFace.zip               df_test_attr.csv   GANS.py                 list_bbox_celeba.csv            model.pt
CNN_for_Gender_Prediction.py  df_train_attr.csv  img_align_celeba        list_eval_partition.csv         train_attr.csv
CNN_hair_prediction.py      df_valid_attr.csv  list_attr_celeba_gan.csv  list_landmarks_align_celeba.csv  valid_attr.csv
[jain.he@d1006 Project]$ python GANS.py
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)
    (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)
    (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)
    (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)
    (12): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (13): Tanh()
  )
)
Discriminator(
  (main): Sequential(
    (0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): LeakyReLU(negative_slope=0.2, inplace)
    (2): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (4): LeakyReLU(negative_slope=0.2, inplace)
    (5): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (6): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (7): LeakyReLU(negative_slope=0.2, inplace)
    (8): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (9): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (10): LeakyReLU(negative_slope=0.2, inplace)
    (11): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
```

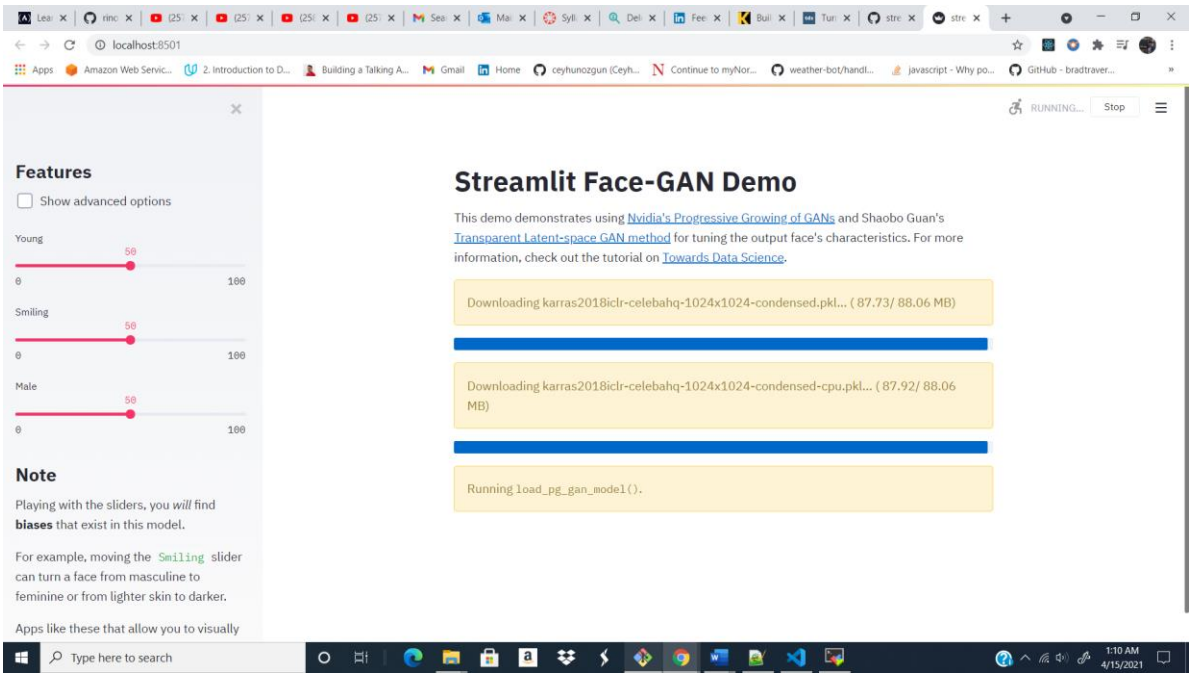
```
[3/4][0/3166]   Loss_D: 0.2941  Loss_G: 2.9544  D(x): 0.8114    D(G(z)): 0.0414 / 0.0740
[3/4][100/3166] Loss_D: 0.2985  Loss_G: 2.1006  D(x): 0.8188    D(G(z)): 0.0643 / 0.1682
[3/4][200/3166] Loss_D: 1.1090  Loss_G: 7.2072  D(x): 0.9801    D(G(z)): 0.5981 / 0.0014
[3/4][300/3166] Loss_D: 0.1801  Loss_G: 4.0457  D(x): 0.9575    D(G(z)): 0.1196 / 0.0234
[3/4][400/3166] Loss_D: 0.5885  Loss_G: 2.2194  D(x): 0.6730    D(G(z)): 0.0929 / 0.1552
[3/4][500/3166] Loss_D: 1.0843  Loss_G: 7.7787  D(x): 0.9889    D(G(z)): 0.5699 / 0.0008
[3/4][600/3166] Loss_D: 0.2924  Loss_G: 3.9692  D(x): 0.7967    D(G(z)): 0.0332 / 0.0357
[3/4][700/3166] Loss_D: 0.3704  Loss_G: 3.9877  D(x): 0.9048    D(G(z)): 0.2022 / 0.0278
[3/4][800/3166] Loss_D: 0.3373  Loss_G: 4.1174  D(x): 0.9186    D(G(z)): 0.2055 / 0.0201
[3/4][900/3166] Loss_D: 0.3750  Loss_G: 4.0370  D(x): 0.9604    D(G(z)): 0.2515 / 0.0231
[3/4][1000/3166]        Loss_D: 0.2122  Loss_G: 3.4713  D(x): 0.9087    D(G(z)): 0.0993 / 0.0414
[3/4][1100/3166]        Loss_D: 0.2527  Loss_G: 2.9662  D(x): 0.8415    D(G(z)): 0.0574 / 0.0793
[3/4][1200/3166]        Loss_D: 0.1493  Loss_G: 3.9827  D(x): 0.9429    D(G(z)): 0.0785 / 0.0251
[3/4][1300/3166]        Loss_D: 0.5325  Loss_G: 3.9614  D(x): 0.9477    D(G(z)): 0.3219 / 0.0325
[3/4][1400/3166]        Loss_D: 0.4779  Loss_G: 5.4602  D(x): 0.9845    D(G(z)): 0.3255 / 0.0067
[3/4][1500/3166]        Loss_D: 0.2695  Loss_G: 3.7219  D(x): 0.9436    D(G(z)): 0.1672 / 0.0360
[3/4][1600/3166]        Loss_D: 1.8459  Loss_G: 0.5848  D(x): 0.2780    D(G(z)): 0.0840 / 0.6597
[3/4][1700/3166]        Loss_D: 0.5022  Loss_G: 2.9573  D(x): 0.7038    D(G(z)): 0.0388 / 0.0888
[3/4][1800/3166]        Loss_D: 0.2886  Loss_G: 1.9183  D(x): 0.7940    D(G(z)): 0.0369 / 0.2000
[3/4][1900/3166]        Loss_D: 0.9606  Loss_G: 0.4805  D(x): 0.5124    D(G(z)): 0.0745 / 0.6708
[3/4][2000/3166]        Loss_D: 0.2118  Loss_G: 3.5007  D(x): 0.9164    D(G(z)): 0.0998 / 0.0441
[3/4][2100/3166]        Loss_D: 0.1511  Loss_G: 4.0647  D(x): 0.9188    D(G(z)): 0.0573 / 0.0257
[3/4][2200/3166]        Loss_D: 0.6962  Loss_G: 2.0736  D(x): 0.6283    D(G(z)): 0.0417 / 0.1901
[3/4][2300/3166]        Loss_D: 0.2170  Loss_G: 4.5004  D(x): 0.9890    D(G(z)): 0.1726 / 0.0157
[3/4][2400/3166]        Loss_D: 0.3062  Loss_G: 2.8310  D(x): 0.8031    D(G(z)): 0.0481 / 0.0971
[3/4][2500/3166]        Loss_D: 2.0125  Loss_G: 8.6106  D(x): 0.9880    D(G(z)): 0.7741 / 0.0004
[3/4][2600/3166]        Loss_D: 0.3248  Loss_G: 2.8210  D(x): 0.8039    D(G(z)): 0.0627 / 0.0984
[3/4][2700/3166]        Loss_D: 0.4955  Loss_G: 3.5225  D(x): 0.7948    D(G(z)): 0.1712 / 0.0464
[3/4][2800/3166]        Loss_D: 0.2676  Loss_G: 3.3524  D(x): 0.8429    D(G(z)): 0.0721 / 0.0559
[3/4][2900/3166]        Loss_D: 0.4529  Loss_G: 5.2670  D(x): 0.9607    D(G(z)): 0.3046 / 0.0086
[3/4][3000/3166]        Loss_D: 0.1803  Loss_G: 4.5233  D(x): 0.9635    D(G(z)): 0.1181 / 0.0183
[3/4][3100/3166]        Loss_D: 0.4313  Loss_G: 3.0371  D(x): 0.8278    D(G(z)): 0.1554 / 0.0677
GPU time:  9550.521588399999
```

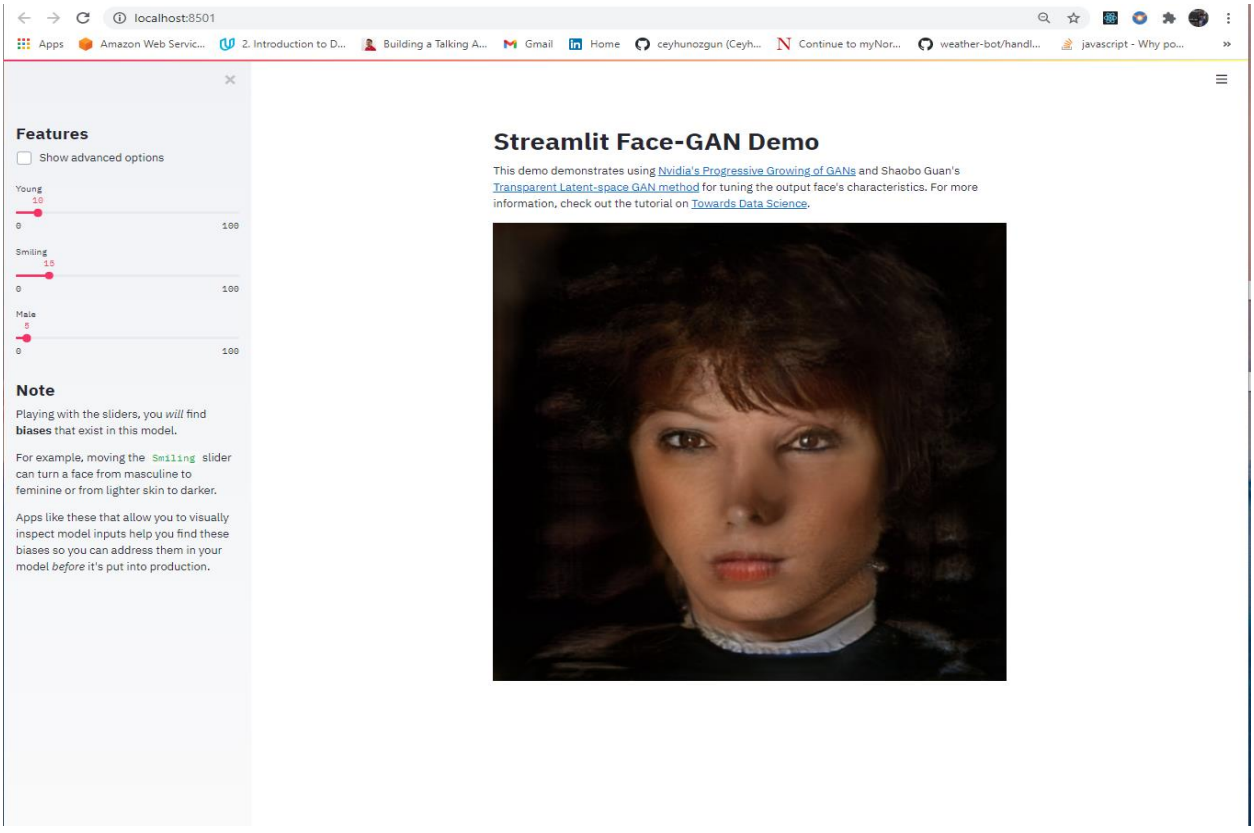
For 4 GPU's the computing time of each epoch was 4500 seconds and the resultant loss function for Generator was 0.4 and Discriminator was 1.1

```
[3/4][0/3166]   Loss_D: 0.4795  Loss_G: 2.2914  D(x): 0.7557    D(G(z)): 0.1374 / 0.1338
[3/4][100/3166] Loss_D: 1.4236  Loss_G: 3.6267  D(x): 0.5922    D(G(z)): 0.5038 / 0.0480
[3/4][200/3166] Loss_D: 3.9934  Loss_G: 5.2356  D(x): 0.9715    D(G(z)): 0.9494 / 0.0114
[3/4][300/3166] Loss_D: 0.5413  Loss_G: 2.7278  D(x): 0.7242    D(G(z)): 0.1515 / 0.0912
[3/4][400/3166] Loss_D: 0.5128  Loss_G: 3.3096  D(x): 0.8420    D(G(z)): 0.2528 / 0.0486
[3/4][500/3166] Loss_D: 0.5115  Loss_G: 3.8178  D(x): 0.8939    D(G(z)): 0.2990 / 0.0335
[3/4][600/3166] Loss_D: 0.4742  Loss_G: 3.1267  D(x): 0.8610    D(G(z)): 0.2521 / 0.0526
[3/4][700/3166] Loss_D: 2.3469  Loss_G: 0.4856  D(x): 0.2024    D(G(z)): 0.0308 / 0.7024
[3/4][800/3166] Loss_D: 1.8771  Loss_G: 7.0005  D(x): 0.9765    D(G(z)): 0.7738 / 0.0031
[3/4][900/3166] Loss_D: 1.0538  Loss_G: 1.6426  D(x): 0.4295    D(G(z)): 0.0351 / 0.2485
[3/4][1000/3166]        Loss_D: 0.4463  Loss_G: 2.4891  D(x): 0.7475    D(G(z)): 0.1193 / 0.1092
[3/4][1100/3166]        Loss_D: 0.5137  Loss_G: 3.9280  D(x): 0.8597    D(G(z)): 0.2614 / 0.0313
[3/4][1200/3166]        Loss_D: 1.4704  Loss_G: 6.8328  D(x): 0.9552    D(G(z)): 0.6734 / 0.0017
[3/4][1300/3166]        Loss_D: 0.4235  Loss_G: 2.0785  D(x): 0.7346    D(G(z)): 0.0686 / 0.1540
[3/4][1400/3166]        Loss_D: 0.6498  Loss_G: 3.3888  D(x): 0.8898    D(G(z)): 0.3539 / 0.0437
[3/4][1500/3166]        Loss_D: 0.4491  Loss_G: 3.7735  D(x): 0.9240    D(G(z)): 0.2615 / 0.0354
[3/4][1600/3166]        Loss_D: 0.4459  Loss_G: 3.2043  D(x): 0.8682    D(G(z)): 0.2399 / 0.0553
[3/4][1700/3166]        Loss_D: 0.7536  Loss_G: 4.2636  D(x): 0.9156    D(G(z)): 0.4297 / 0.0211
[3/4][1800/3166]        Loss_D: 0.5225  Loss_G: 1.8523  D(x): 0.7062    D(G(z)): 0.1086 / 0.2070
[3/4][1900/3166]        Loss_D: 0.7472  Loss_G: 1.9273  D(x): 0.5497    D(G(z)): 0.0484 / 0.1831
[3/4][2000/3166]        Loss_D: 0.6784  Loss_G: 2.4898  D(x): 0.7437    D(G(z)): 0.2474 / 0.1180
[3/4][2100/3166]        Loss_D: 0.3617  Loss_G: 3.0493  D(x): 0.8552    D(G(z)): 0.1563 / 0.0607
[3/4][2200/3166]        Loss_D: 0.3716  Loss_G: 3.5800  D(x): 0.8596    D(G(z)): 0.1750 / 0.0391
[3/4][2300/3166]        Loss_D: 0.3311  Loss_G: 2.5398  D(x): 0.8064    D(G(z)): 0.0810 / 0.1051
[3/4][2400/3166]        Loss_D: 0.7881  Loss_G: 5.3509  D(x): 0.9400    D(G(z)): 0.4635 / 0.0067
[3/4][2500/3166]        Loss_D: 0.4829  Loss_G: 2.6557  D(x): 0.7887    D(G(z)): 0.1774 / 0.0957
[3/4][2600/3166]        Loss_D: 0.4390  Loss_G: 2.8930  D(x): 0.8293    D(G(z)): 0.1955 / 0.0813
[3/4][2700/3166]        Loss_D: 0.3790  Loss_G: 2.2533  D(x): 0.8471    D(G(z)): 0.1674 / 0.1353
[3/4][2800/3166]        Loss_D: 0.6070  Loss_G: 2.5647  D(x): 0.8111    D(G(z)): 0.2721 / 0.0989
[3/4][2900/3166]        Loss_D: 0.4494  Loss_G: 3.2782  D(x): 0.7093    D(G(z)): 0.0482 / 0.0576
[3/4][3000/3166]        Loss_D: 0.4129  Loss_G: 2.5378  D(x): 0.7307    D(G(z)): 0.0628 / 0.1178
[3/4][3100/3166]        Loss_D: 1.1646  Loss_G: 0.4508  D(x): 0.4345    D(G(z)): 0.0795 / 0.6591
GPU time:  4500.85173593834
```

For 1 GPU the computing time for each epoch was 9550 seconds double of the 4 GPU result and the loss for Generator and Discriminator were 3.03 and 0.4, respectively.

Streamlit based Fake Face Generation using the Nvidia trained ML model





Conclusion

With deep learning methodologies like CNN and GANs I was able to achieve the goal I set at the starting of our project. The CNN models accurately predict the Gender and the hair colors reading through the celebrity images. With an accuracy of 92%. In the GAN model the two components i.e., the generator and the discriminator performed the task well with one generating the fake images and the other identifying between the fake and the real once. Additional Deployment of Fake Face Generative Embedded Machine Learning Model with Streamlit based App makes it quite interactive and real-time in utilizing the Pre-trained ML model from Nvidia for new face creation in no time.

References

1. **https://www.kaggle.com/jessicali9530/celeba-dataset/notebooks**
2. **https://stackoverflow.com/questions/43288550/iopub-data-rate-exceeded-in-jupyter-notebook-when-viewing-image**
3. **https://pytorch.org/tutorials/beginner/blitz/data_parallel_tutorial.html**
4. **https://analyticsindiamag.com/multi-label-image-classification-with-tensorflow-keras/**
5. **https://www.kaggle.com/waltermaffy/create-fake-face-with-pytorch**
6. **https://stackoverflow.com/questions/45303186/delete-row-from-file-using-csv-reader-and-lists-python**
7. **https://stackoverflow.com/questions/44613507/typeerror-join-argument-must-be-str-or-bytes-not-nonetype**
8. **https://www.kaggle.com/c/jigsaw-unintended-bias-in-toxicity-classification/discussion/91081**
9. **https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html**
10. **https://medium.com/ai-society/gans-from-scratch-1-a-deep-introduction-with-code-in-pytorch-and-tensorflow-cb03cdcdba0f**
11. **https://rc-docs.northeastern.edu/en/latest/software/software.html**
12. **https://rc-docs.northeastern.edu/en/latest/using-discovery/transferringdata.html**
13. **https://pytorch.org/tutorials/**
14. **https://stackoverflow.com/questions/55274469/slurm-python-multiprocessing-exceed-memory-limit**
15. **https://forum.qt.io/topic/108466/installation-error-qstandardpaths-xdg_runtime_dir-not-set-defaulting-to-tmp-runtime-root/4**
16. **https://towardsdatascience.com/how-to-train-an-image-classifier-in-pytorch-and-use-it-to-perform-basic-inference-on-single-images-99465a1e9bf5**
17. **https://towardsdatascience.com/understanding-cnn-convolutional-neural-network-69fd626ee7d4**
18. *S. Yang, P. Luo, C. C. Loy, and X. Tang, "From Facial Parts Responses to Face Detection: A Deep Learning Approach", in IEEE International Conference on Computer Vision (ICCV), 2015*
19. **https://arxiv.org/pdf/1909.11573.pdf**
20. **https://www.researchgate.net/publication/334811480_Culturally_Evolved_GANs_for_Generating_Fake_Stroke_Faces**