1 Preparing the Dataset

Our task is to have a dataset of both masked and unmasked faces. This is required to train a model to generate reasonably accurate unmasked faces given masked faces. I decided to use Celeba dataset(https://www.kaggle.com/jessicali9530/celeba-dataset), which consists of roughly 200k images of celebrity faces, for unmasked faces. For the purpose of testing, I decided to use only 100k of these images. The images look like

Figure 1: unmasked face

Now with these unmasked faces at hand, we also need their corresponding masked faces. To obtain this I referred to (https://github.com/Garvit-32/Face-recognition-in-presence-of-Mask). Here python's face recognition and open CV library is used to detect and attach masks to the faces in the images. The result looks something like this

Figure 2: unmasked face

Now we obtained the dataset as required for our project. The full dataset required for training is supposed to consist of both masked and unmasked faces in different folders. I noticed that during the application of masks, some images could not be attached with a mask by the code.

Figure 3: Difficult image to attach mask

So these corresponding masked faces(roughly 3k) are absent form our dataset. And also I find that not all these images are of same size and dimensions. So we next need to clean and process this dataset for it to be finally available to use for building and training the model.

2 Cleaning the Dataset

As mentioned before, for some faces, masks could not be appended. The unmasked versions of these images must therefore be removed from out dataset. I intend to keep masked faces and umnasked faces in separate folders, but with complementary images having the same index for ease of accessing. Moreover the individual images are quite large(178×218) in Celeba dataset. It might be a good option to try a model on low resolution images images first and then proceed on the basis of results. So I decided to work with 64x64 images(for both masked and unmasked).

(a) unmasked (b) masked

Figure 4: Dataset

3 Preparation and Training

I will use PyTorch framework to build and train deep learning models to generate an unmasked face, given a masked one. My current approach is to build a regression model(since the input and output both are numerical) and train it on the database created previously.

3.1 Splitting the dataset

The dataset consists of 97170 images. We split it into a train set(consisting of 87170 images) and test set(consisting of 10000 images). We need to set a seed so that we do not get different datasets in different iterations of the code. The split is done from the random split function from torch.utils.data.

3.2 Normalising the images

Each image can now be represented by a (3,64,64) tensor. The pixel values in different images might have different distribution, which slows down convergence while training. To avoid such issues, it is advised to normalise the images by subtracting the mean value and dividing by standard deviation of pixels. This makes the pixels lie in a Gaussian distribution centred at 0 with standard deviation 1. Now while doing this, one must keep in mind that we should only be normalising the dataset with mean and standard deviation of the pixels from the training set only. We can normalise the test set with this mean and standard deviation, but we should not include the test set while calculating them.

We therefore are also required to write a function which denormalizes the images with mean and standard deviation to view the images properly.

3.3 Model

I built a CNN model with resnet consisting of 6-layers, 28,643 trainable parameters, and total size of 4.25 MB. Reason for using resnet is that the input and output is very similar. We can think of this as

$$input + error = output \tag{3.1}$$

So instead of building a function which models the output, we can instead build one to model the error and add it to the input.

3.4 Hyperparameters and Training

PyTorch has an autograd function for computing gradients of loss, which are then back-propagated to update the weights. I have used the optimizer which implements stochastic gradient descent(torch.optim.SGD) for sake of simplicity. It requires a learning rate, which, after some experimentation, I provided to be 0.0001. The training process involves iterating over the dataset a few times(6 in my case) in batches(I took batch size to be 35).

I have considered a simple loss function(mean squared error) for my model.

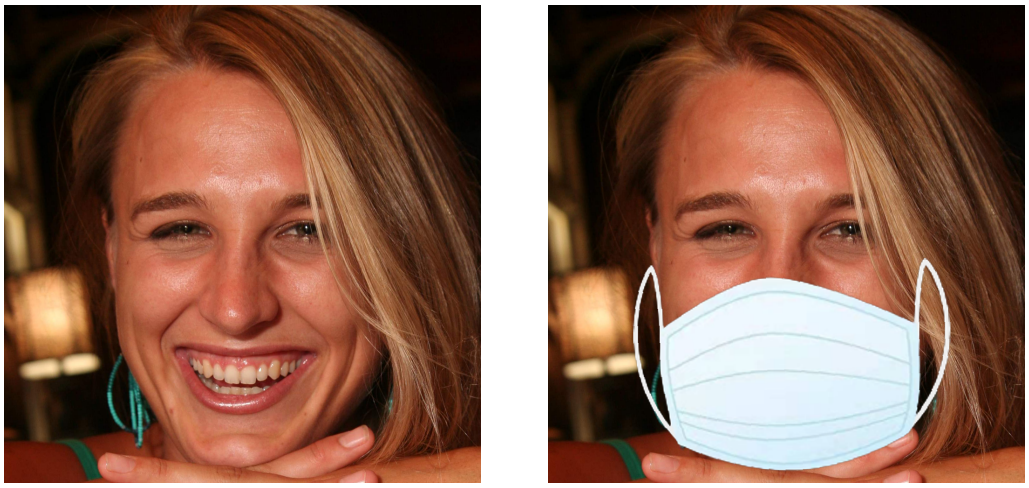
4 Literature Survey

I looked for recent papers on problems similar to our project and came across a few. I will discuss two of which I studied this week in the following subsections

- A Novel GAN-Based Network for Unmasking of Masked Face: This paper approaches the problem in two stages. In the first stage they produce a binary segmentation for the mask region. After that, in the second stage, they remove the mask and synthesize the affected region. This is very similar to out project but they have worked with Celeba dataset in which images are of considerably lower size(178×218).
- Large Scale Image Completion via co-modulated GANS: In this paper, authors discuss a new approach to image completion problem when large portions of the image are absent. This is relevant to our project as we can treat the mask as we can train the model to treat the masked region in the images as some sort of in-completion, which it has to complete.

5 Large image dataset

Both these approaches are computationally expensive especially on large images. I have started working with Flickr-Faces HQ dataset which consists of substantially larger images(1024x1024) than Celeba dataset. It consists of 63,000 images(approximately 85GB on my disk) and appending masks on these faces to generate our synthetic data for our project takes some time, for 1000 images it took 16 min, thus for the whole dataset it will take around 16-17 hours.



(a) unmasked (b) masked

Figure 5: High resolution images of FFHQ dataset with mask appended

6 Model for FFHQ

I built a simple regression model to obtain a sense of hyperparameters and time required for training. This will be useful when we deploy the full model for training. The model architecture/parameters are shown in the figure below

```
: summary(model,(3,1024,1024))
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
├─Sequential: 1-1                        [-1, 16, 1024, 1024]      --
|    └─Conv2d: 2-1                       [-1, 16, 1024, 1024]      448
|    └─ReLU: 2-2                         [-1, 16, 1024, 1024]      --
├─Sequential: 1-2                        [-1, 32, 1024, 1024]      --
|    └─Conv2d: 2-3                       [-1, 32, 1024, 1024]      4,640
|    └─ReLU: 2-4                         [-1, 32, 1024, 1024]      --
├─Sequential: 1-3                        [-1, 32, 1024, 1024]      --
|    └─Sequential: 2-5                   [-1, 32, 1024, 1024]      --
|    |    └─Conv2d: 3-1                  [-1, 32, 1024, 1024]      9,248
|    |    └─ReLU: 3-2                    [-1, 32, 1024, 1024]      --
|    └─Sequential: 2-6                   [-1, 32, 1024, 1024]      --
|    |    └─Conv2d: 3-3                  [-1, 32, 1024, 1024]      9,248
|    |    └─ReLU: 3-4                    [-1, 32, 1024, 1024]      --
├─Sequential: 1-4                        [-1, 3, 1024, 1024]       --
|    └─Sequential: 2-7                   [-1, 16, 1024, 1024]      --
|    |    └─Conv2d: 3-5                  [-1, 16, 1024, 1024]      4,624
|    |    └─ReLU: 3-6                    [-1, 16, 1024, 1024]      --
|    └─ReLU: 2-8                         [-1, 16, 1024, 1024]      --
|    └─Conv2d: 2-9                       [-1, 3, 1024, 1024]       435
|    └─Tanh: 2-10                        [-1, 3, 1024, 1024]       --
==========================================================================================
Total params: 28,643
Trainable params: 28,643
Non-trainable params: 0
Total mult-adds (G): 29.90
==========================================================================================
Input size (MB): 12.00
Forward/backward pass size (MB): 1048.00
Params size (MB): 0.11
Estimated Total Size (MB): 1060.11
==========================================================================================
```

Figure 6: Model

I trained this model with a batch size of 1(images are huge), with a learning rate of 0.0001 for 12 epochs on a fraction of FFHQ dataset containing just 1000 images. Even on this small dataset the training took approximately an hour. The training and validation losses vary as given in the figure below

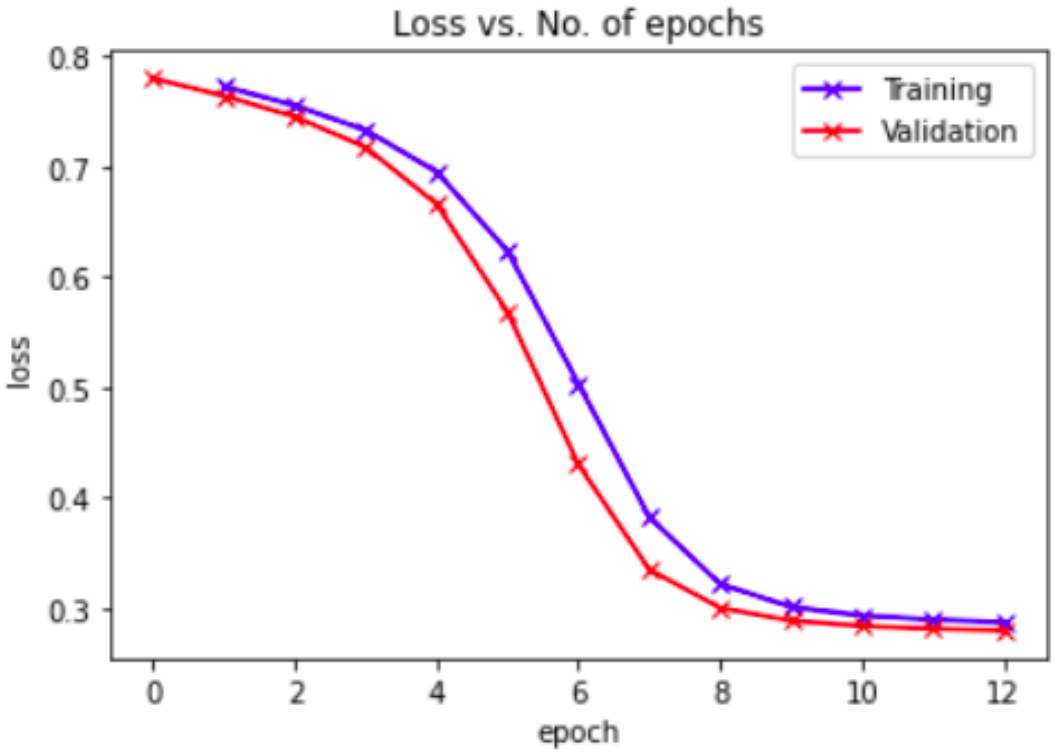


Figure 7: Losses

Based on these estimates, we can say the full model might take approximate 70-100hrs to train on a dataset of size roughly 60k. If we vary the batch size these estimates might vary. To get an idea of VRAM usage for full model, we can look at usage for this simple model

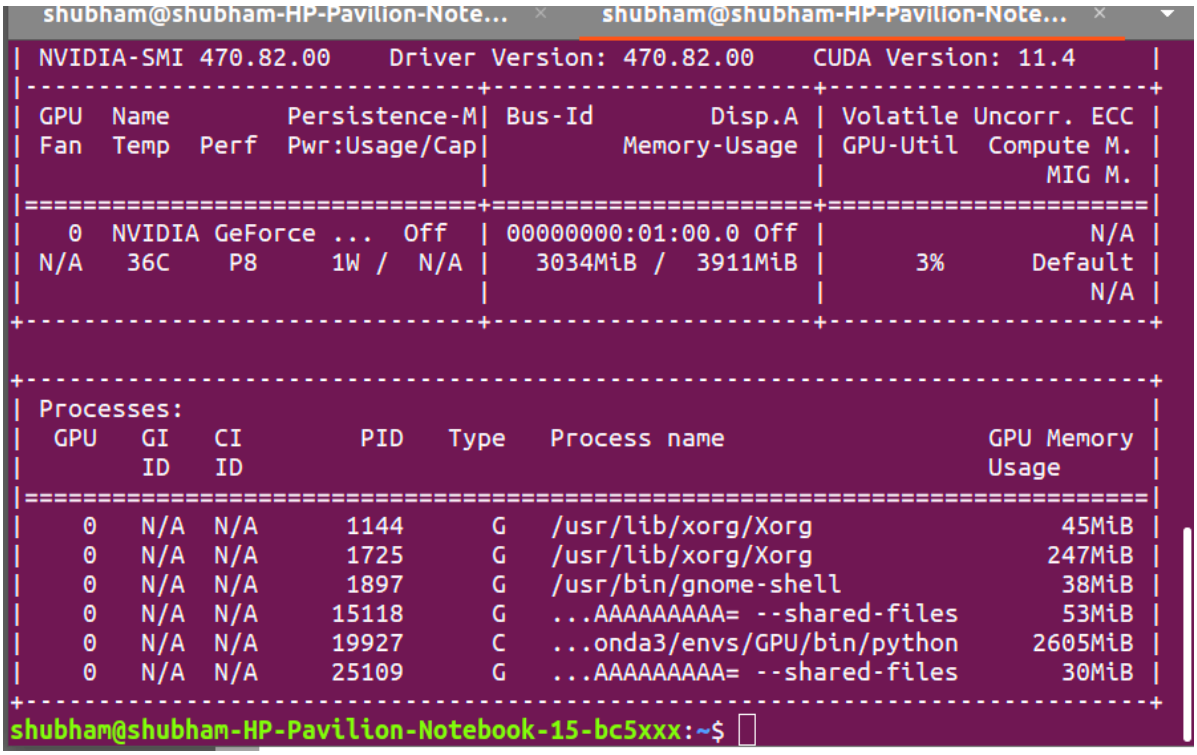


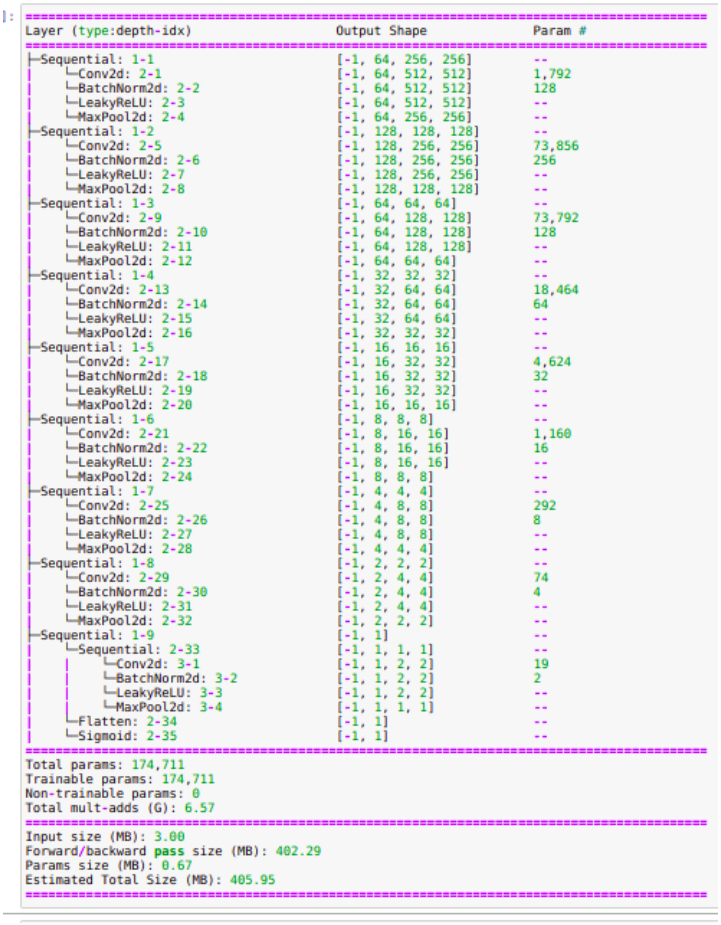
Figure 8: Losses

with model size of approximately 1GB, and batch size of just 1 image, approximately 3GB of VRAM is used. In the full model, it will increase depending mainly on model parameters and batch size.

7 GAN Model for FFHQ

7.1 Discriminator

I made a simple GAN model for our problem. It consists of a Discriminator model and a Generator model. The Discriminator, as the name suggests, trained to specialize in distinguishing between real data(our training data) and fake data(generated by the Generator). In my case, the model looks like this

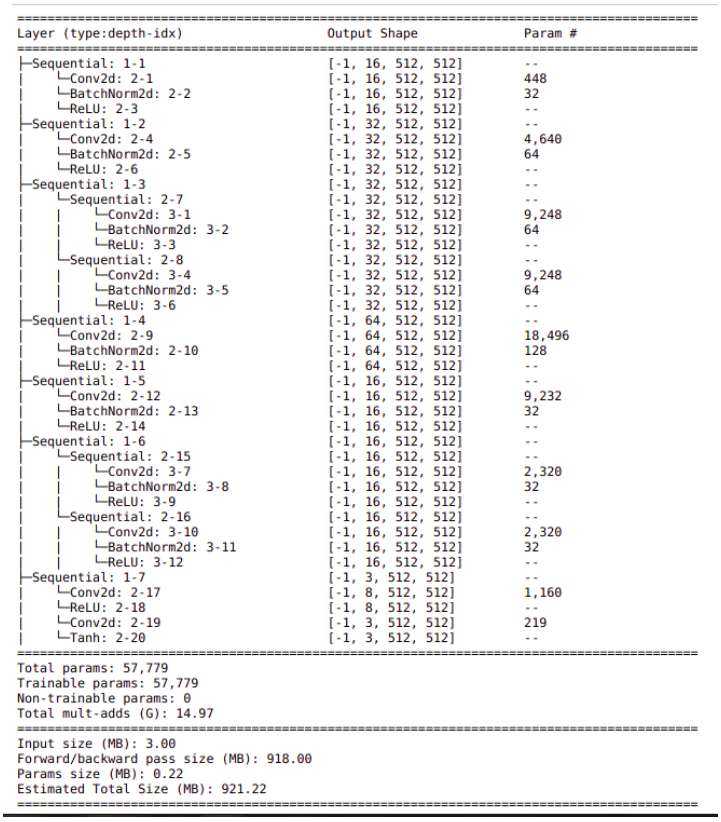


```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
├─Sequential: 1-1                        [-1, 64, 256, 256]        --
|    └─Conv2d: 2-1                       [-1, 64, 512, 512]        1,792
|    └─BatchNorm2d: 2-2                  [-1, 64, 512, 512]        128
|    └─LeakyReLU: 2-3                    [-1, 64, 512, 512]        --
|    └─MaxPool2d: 2-4                    [-1, 64, 256, 256]        --
├─Sequential: 1-2                        [-1, 128, 128, 128]       --
|    └─Conv2d: 2-5                       [-1, 128, 256, 256]       73,856
|    └─BatchNorm2d: 2-6                  [-1, 128, 256, 256]       256
|    └─LeakyReLU: 2-7                    [-1, 128, 256, 256]       --
|    └─MaxPool2d: 2-8                    [-1, 128, 128, 128]       --
├─Sequential: 1-3                        [-1, 64, 64, 64]          --
|    └─Conv2d: 2-9                       [-1, 64, 128, 128]        73,792
|    └─BatchNorm2d: 2-10                 [-1, 64, 128, 128]        128
|    └─LeakyReLU: 2-11                   [-1, 64, 128, 128]        --
|    └─MaxPool2d: 2-12                   [-1, 64, 64, 64]          --
├─Sequential: 1-4                        [-1, 32, 32, 32]          --
|    └─Conv2d: 2-13                      [-1, 32, 64, 64]          18,464
|    └─BatchNorm2d: 2-14                 [-1, 32, 64, 64]          64
|    └─LeakyReLU: 2-15                   [-1, 32, 64, 64]          --
|    └─MaxPool2d: 2-16                   [-1, 32, 32, 32]          --
├─Sequential: 1-5                        [-1, 16, 16, 16]          --
|    └─Conv2d: 2-17                      [-1, 16, 32, 32]          4,624
|    └─BatchNorm2d: 2-18                 [-1, 16, 32, 32]          32
|    └─LeakyReLU: 2-19                   [-1, 16, 32, 32]          --
|    └─MaxPool2d: 2-20                   [-1, 16, 16, 16]          --
├─Sequential: 1-6                        [-1, 8, 8, 8]             --
|    └─Conv2d: 2-21                      [-1, 8, 16, 16]           1,160
|    └─BatchNorm2d: 2-22                 [-1, 8, 16, 16]           16
|    └─LeakyReLU: 2-23                   [-1, 8, 16, 16]           --
|    └─MaxPool2d: 2-24                   [-1, 8, 8, 8]             --
├─Sequential: 1-7                        [-1, 4, 4, 4]             --
|    └─Conv2d: 2-25                      [-1, 4, 8, 8]             292
|    └─BatchNorm2d: 2-26                 [-1, 4, 8, 8]             8
|    └─LeakyReLU: 2-27                   [-1, 4, 8, 8]             --
|    └─MaxPool2d: 2-28                   [-1, 4, 4, 4]             --
├─Sequential: 1-8                        [-1, 2, 2, 2]             --
|    └─Conv2d: 2-29                      [-1, 2, 4, 4]             74
|    └─BatchNorm2d: 2-30                 [-1, 2, 4, 4]             4
|    └─LeakyReLU: 2-31                   [-1, 2, 4, 4]             --
|    └─MaxPool2d: 2-32                   [-1, 2, 2, 2]             --
├─Sequential: 1-9                        [-1, 1]                   --
|    └─Sequential: 2-33                  [-1, 1, 1, 1]             --
|    |    └─Conv2d: 3-1                  [-1, 1, 2, 2]             19
|    |    └─BatchNorm2d: 3-2             [-1, 1, 2, 2]             2
|    |    └─LeakyReLU: 3-3               [-1, 1, 2, 2]             --
|    |    └─MaxPool2d: 3-4               [-1, 1, 1, 1]             --
|    └─Flatten: 2-34                     [-1, 1]                   --
|    └─Sigmoid: 2-35                     [-1, 1]                   --
==========================================================================================
Total params: 174,711
Trainable params: 174,711
Non-trainable params: 0
Total mult-adds (G): 6.57
==========================================================================================
Input size (MB): 3.00
Forward/backward pass size (MB): 402.29
Params size (MB): 0.67
Estimated Total Size (MB): 405.95
==========================================================================================
```

Figure 9: Discriminator Model

7.2 Generator

The Generator model, on the other hand is the one trained to generates the desired output. In our case, the desired output is the mask-less image. It is the Generator which is finally deployed. The model is given by



```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
├─Sequential: 1-1                        [-1, 16, 512, 512]        --
|    └─Conv2d: 2-1                       [-1, 16, 512, 512]        448
|    └─BatchNorm2d: 2-2                  [-1, 16, 512, 512]        32
|    └─ReLU: 2-3                         [-1, 16, 512, 512]        --
├─Sequential: 1-2                        [-1, 32, 512, 512]        --
|    └─Conv2d: 2-4                       [-1, 32, 512, 512]        4,640
|    └─BatchNorm2d: 2-5                  [-1, 32, 512, 512]        64
|    └─ReLU: 2-6                         [-1, 32, 512, 512]        --
├─Sequential: 1-3                        [-1, 32, 512, 512]        --
|    └─Sequential: 2-7                   [-1, 32, 512, 512]        --
|    |    └─Conv2d: 3-1                  [-1, 32, 512, 512]        9,248
|    |    └─BatchNorm2d: 3-2             [-1, 32, 512, 512]        64
|    |    └─ReLU: 3-3                    [-1, 32, 512, 512]        --
|    └─Sequential: 2-8                   [-1, 32, 512, 512]        --
|    |    └─Conv2d: 3-4                  [-1, 32, 512, 512]        9,248
|    |    └─BatchNorm2d: 3-5             [-1, 32, 512, 512]        64
|    |    └─ReLU: 3-6                    [-1, 32, 512, 512]        --
├─Sequential: 1-4                        [-1, 64, 512, 512]        --
|    └─Conv2d: 2-9                       [-1, 64, 512, 512]        18,496
|    └─BatchNorm2d: 2-10                 [-1, 64, 512, 512]        128
|    └─ReLU: 2-11                        [-1, 64, 512, 512]        --
├─Sequential: 1-5                        [-1, 16, 512, 512]        --
|    └─Conv2d: 2-12                      [-1, 16, 512, 512]        9,232
|    └─BatchNorm2d: 2-13                 [-1, 16, 512, 512]        32
|    └─ReLU: 2-14                        [-1, 16, 512, 512]        --
├─Sequential: 1-6                        [-1, 16, 512, 512]        --
|    └─Sequential: 2-15                  [-1, 16, 512, 512]        --
|    |    └─Conv2d: 3-7                  [-1, 16, 512, 512]        2,320
|    |    └─BatchNorm2d: 3-8             [-1, 16, 512, 512]        32
|    |    └─ReLU: 3-9                    [-1, 16, 512, 512]        --
|    └─Sequential: 2-16                  [-1, 16, 512, 512]        --
|    |    └─Conv2d: 3-10                 [-1, 16, 512, 512]        2,320
|    |    └─BatchNorm2d: 3-11            [-1, 16, 512, 512]        32
|    |    └─ReLU: 3-12                   [-1, 16, 512, 512]        --
├─Sequential: 1-7                        [-1, 3, 512, 512]         --
|    └─Conv2d: 2-17                      [-1, 8, 512, 512]         1,160
|    └─ReLU: 2-18                        [-1, 8, 512, 512]         --
|    └─Conv2d: 2-19                      [-1, 3, 512, 512]         219
|    └─Tanh: 2-20                        [-1, 3, 512, 512]         --
==========================================================================================
Total params: 57,779
Trainable params: 57,779
Non-trainable params: 0
Total mult-adds (G): 14.97
==========================================================================================
Input size (MB): 3.00
Forward/backward pass size (MB): 918.00
Params size (MB): 0.22
Estimated Total Size (MB): 921.22
==========================================================================================
```

Figure 10: Generator Model

7.3 Training

As we have seen, the Discriminator is the classification model, it classifies data into real and fake. During training, the Discriminator loss is obtained by passing both real and fake data. While the Generator loss is obtained by its ability to deceive the Discriminator into thinking it is real. Both co-evolve and improve over time.

8 Training the model

I trained the model on my pc for approximately 21 hours, for 3 epochs. We might need to train it for atleast 25-30 epochs for a decent output. Below are some results from the training

Figure 11: Untrained Model output

Figure 12: Model output after epoch 3

In GANS, one expects the training of discriminator to be in sync with each other, hence their looses are supposed to be parallel(losses of one model depends on other one)

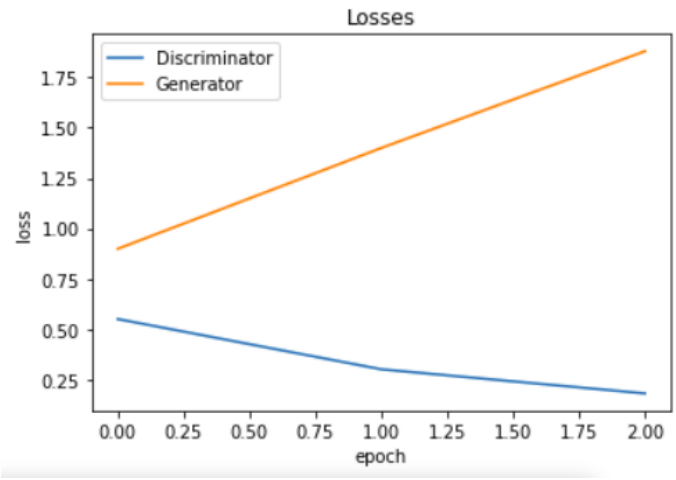


Figure 13: Losses

We can see that losses are not parallel yet, it might take a few more epochs.

9 A different approach

I trained the model on my pc for approximately 63 hours, for 9 epochs. Since each epoch is taking around 7hrs, it is very difficult to train the model in a reasonable amount of time in the absence of computational resources. Even experimenting with the the parameters is turning out to be difficult. Hence I have decided to try a different approach involving both deep leaning and Computer vision techniques. It would consist of two stages. At first, I would build a GANS model which generates random images of peoples faces. Then using face recognition library, I would try to identify the pixels in the image corresponding to the mask and remove them. Then using the GANS model I would try to predict the values of the missing pixels. This is essentially breaking down the task so that our model doesn't have to learn a lot.

In [37]: losses

Out[37]:

	Losses_d	Losses_g	Real_scores	Fake_scores
0	0.550633	0.898977	0.972442	0.407076
1	0.301438	1.399216	0.982277	0.246899
2	0.181908	1.878960	0.984060	0.152818
3	0.107125	2.358867	0.992100	0.094432
4	0.098614	2.452734	0.991652	0.086280
5	0.068096	2.802949	0.994635	0.060790
6	0.037979	3.364961	0.997158	0.034523
7	0.019865	3.997149	0.998689	0.018383
8	0.014429	4.311203	0.999089	0.013426

Figure 14: losses

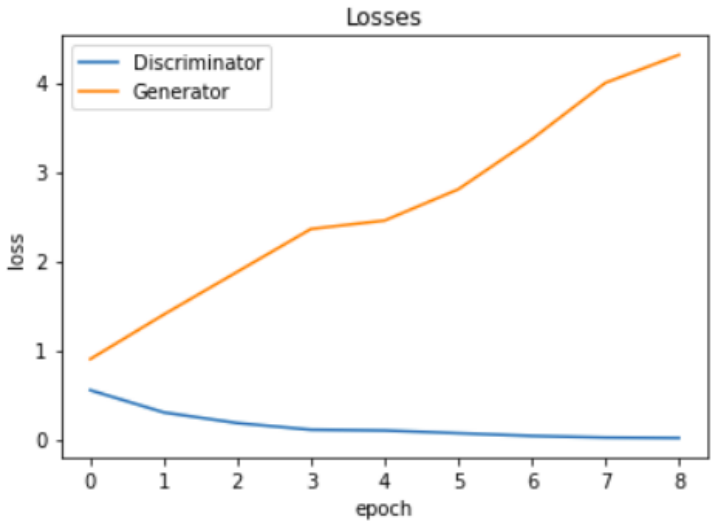


Figure 15: losses

You can observe that even after 63 hrs of training, the Generator and Discriminator are not in sync.

10 Random Face Generator

As a first part of the project discussed previously, I built a model which generates random images of people. It is trained using the FFHQ dataset I was using previously. A GANS model like this tries to generate new data by observing the training data. The training data comes from a distribution and our model learns to generate data points from this distribution.

This week, I will spend some time training the model and obtaining the optimum hyperparameters. It turns out that this model takes only 1.5 hrs to train for one epoch(instead of 7hrs for the previous approach). This makes debugging and training a lot simpler. One reason for this drastic reduction in time might be the fact that we only need to load one set of images for this model, the unmasked faces, unlike two in previous case. This allows us to pass a batch of images together(something which I couldn't do before due large size of individual data points) and hence improves training time.

I have experimented with different batch sizes and obtained the value of 5 as an optimum answer.

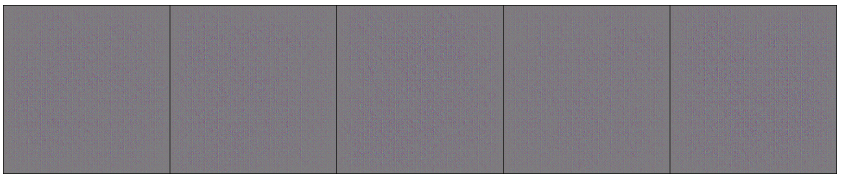


Figure 16: uninitialised model

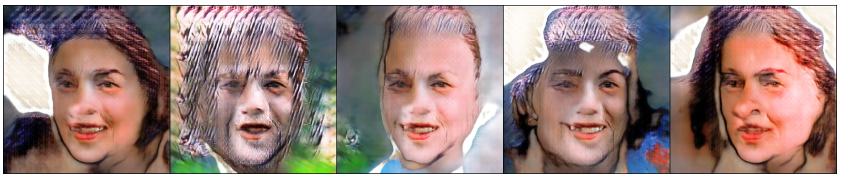


Figure 17: after 8 epochs

11 Object Detection Model

We need and object detection model to determine the pixels corresponding to mask in an image. I used a YOLOv5 model for this purpose. It is very convenient to use YOLOv5

for image segmentation task like this. Shown below is the output of the model I trained for 60 epochs on masked images.

Figure 18: Predictions on validation

12 Image Completion

For Image completion, I refereed to https://github.com/otenim/GLCIC-PyTorch. I integrated my code with theirs to obtain the desired result of successful image completion after removal of mask. The figure below is the output of the model.

Figure 19: Masked — mask removal — Image completion

13 References

- http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html
- https://github.com/otenim/GLCIC-PyTorch
- https://github.com/NVlabs/ffhq-dataset
- https://arxiv.org/pdf/2103.10428.pdf
- https://github.com/ultralytics/yolov5