FINAL PROJECT REPORT

By: Chibuikem Ezemaduka   Dept: ECSE

RIN: 661993890

Date: 5/6/2022

Human face generation using Conditional Generative Adversarial Networks (CGANS)

Synthetic data generation using a deep neural network simply involves learning from given real data samples in order to generate realistic synthetic data. This artificial data can be used for many applications such as simulations, data augmentation, graphic design applications, visualization, entertainment etc. Human face generation is the task in which the synthetic data to be generated are human faces. It involves learning from available data on real human faces in order to be able to generate artificial human faces that still look realistic and optimally cannot be differentiated by the naked human eye from real faces. This can have good application in computer vision tasks. Generated human faces can be used to augment existing data to train models for different computer vision applications. It can also be used for entertainment purposes and art. In this work, we treat the human face generation task by using a deep neural network called the Generative Adversarial Network (GAN). GANs have been used successfully to generate synthetic data that is very similar to the real data sets. For this work, we utilize a variant of GAN which is called the Conditional Generative Adversarial Network (CGAN). A cGAN is a variant of GAN which is able to utilize labels which serve as data attributes to influence the synthetic data generation process. For our particular task of human face generation, the labels (conditions) are the human face attributes. cGANs allow us to specify particular attributes that we desire to see in our generated human faces. This gives us a lot of control over the data generation process and can enable us to tailor and diversify our synthetic data to different applications. For this particular work, the attributes we select are : Black hair, Male, Oval Face, Smiling, and Young. We aim to generate synthetic human faces that satisfy our given attribute conditions. Next, we formally define and model our task of synthetic data generation using cGANs.

Conditional Generative Adversarial Network (CGAN) problem setting

Before concentrating on the task of using cGAN for the human face generation task, it is important for us to outline the principle behind synthetic data using Generative Adversarial Networks (GANs)

GANs are neural networks which approximate the underlying probability distribution of a given dataset. They achieve this by Generative Adversarial Learning. Specifically, Given training data **D** = {$\mathbf{x}$(m)}, m=1,…,M, the GAN tries to learn a generative model $\mathrm{p_g}(\mathbf{x}|\Theta)$ that can accurately capture the distribution of **x**, $\mathrm{p_D}(\mathbf{x})$ by minimizing the Jensen-Shannon divergence (JSD) between $\mathrm{p_g}(\mathbf{x}|\Theta)$ and $\mathrm{p_D}(\mathbf{x})$. This is called Adversarial Learning.

The Jensen-Shannon divergence between $\mathrm{p_g}(\mathbf{x}|\Theta)$ and $\mathrm{p_D}(\mathbf{x})$ is given as:

$$\mathrm{JSD}(p_D(\mathbf{x})\,||\,p_g(\mathbf{x}|\Theta)) = \frac{1}{2}\mathrm{KL}(p_D(\mathbf{x})\,||\,\frac{p_D(\mathrm{x})+p_g(\mathrm{x}|\Theta)}{2}) + \frac{1}{2}\mathrm{KL}(p_g(\mathbf{x})\,||\,\frac{p_D(\mathrm{x})+p_g(\mathrm{x}|\Theta)}{2})$$

$$= \log 2 + \frac{1}{2}\left[\mathrm{E}_{p_d}(\log \mathrm{f(x)}) + \mathrm{E}_{p_g}(1 - \log \mathrm{f(x)})\right]$$

where $f(x) = \frac{p_D(\mathrm{x})}{p_D(\mathrm{x})+p_g(\mathrm{x})} = y(x|\phi^*)$ is called the **discriminator**

In order to minimize the JSD, a min-max game occurs in which the first term minimizes f(x) over the data distribution, while the second term needs to maximize f(x) over the model distribution.

The aim of Adversarial Learning is to minimize the JSD in a min max optimization formulation given as:

$$\{\theta^*,\phi^*\} = \arg \min_{\theta} \max_{\phi} \; \mathrm{E}_{\mathrm{x}\sim p_D(\mathrm{x})}[\log \mathrm{y}(\mathrm{x}|\phi)] + \mathrm{E}_{\mathrm{x}\sim p_g(\mathbf{x}|\boldsymbol{\Theta})}[\log(1 - \mathrm{y}(\mathrm{x}|\phi))]$$

The aim is to first find ϕ by maximization and then Θ by minimization. The objective function achieved minimum when $p_D(\mathrm{x}) = p_g(\mathrm{x})$, $\mathrm{y}(\mathrm{x}|\phi)$ = f(x) = ½, and JSD = -2log2. $\mathrm{y}(\mathrm{x}|\phi)$ is a binary discriminator that measures the probability of being a real data. The **generator** manufactures synthetic data based on the generative model distribution, $p_g(\mathbf{x}|\Theta)$. The generator and discriminator compete during training. The objective is to train a generator that can "deceive" the best discriminator. The generator is saved after training.

Conditional GANs as mentioned are a variant of GAN in which conditions are encoded to guide the generator in generating synthetic data that satisfy the given conditions. Specifically, if we have conditional information **c** from distribution $p_c(\mathrm{c})$ with joint distribution $p_D(\mathrm{x},\mathrm{c})$, then our adversarial learning optimization formulation as described above now becomes:

$$\{\theta^*,\phi^*\} = \arg \min_{\theta} \max_{\phi} \; \mathrm{E}_{\mathrm{x,c}\sim p_D(\mathbf{x},\mathrm{c})}[\log \mathrm{y}(\mathrm{x,c}|\phi)] + \mathrm{E}_{\mathrm{x,c}\sim p_g(\mathbf{x},\mathrm{c}|\boldsymbol{\Theta})}[\log(1 - \mathrm{y}(\mathrm{x,c}|\phi))]$$

Where $\mathrm{y}(\mathrm{x},\mathrm{c}|\phi)$ is the discriminator, Θ are the generator parameters, and ϕ are the discriminator parameters.

For the human face generation task, the training data **D** = {**x**(m),**c**(m)}, m=1,…,M is a sample of M human face images with different attributes. The conditions **c**(m) are the attributes for a given face (specifically Black hair, Male, Oval Face, Smiling, and Young) . Our task is to generate synthetic face images given an encoded set of attributes based on our learned generative model $p_g(\mathbf{x},\mathrm{c}|\boldsymbol{\Theta})$ which was trained through adversarial learning by using the optimization formulation described above. In general, the cGAN is implemented as a de-convolutional neural network with a standardized probabilistic input (called a noise vector) and the conditional information. It learns its parameters by competing with a discriminative model. Its architecture consists of a generator and discriminator. The generator takes in the noise vector and conditional information as input data and then generates a synthetic image with identical dimensions as a real image. The discriminator takes in as input, the real or synthetic image with the conditional information and outputs the probability of it being real. The Generator is trained to generate the most realistic synthetic images and the discriminator is trained to best distinguish real and generated images. In the next sections, we will describe our model architecture and our specific training objective and procedure in details.

SYSTEM MODEL

We now discuss our chosen model architecture and model training objectives.

Model Architecture

We outline our architecture for the generator and discriminator:

Generator Architecture:

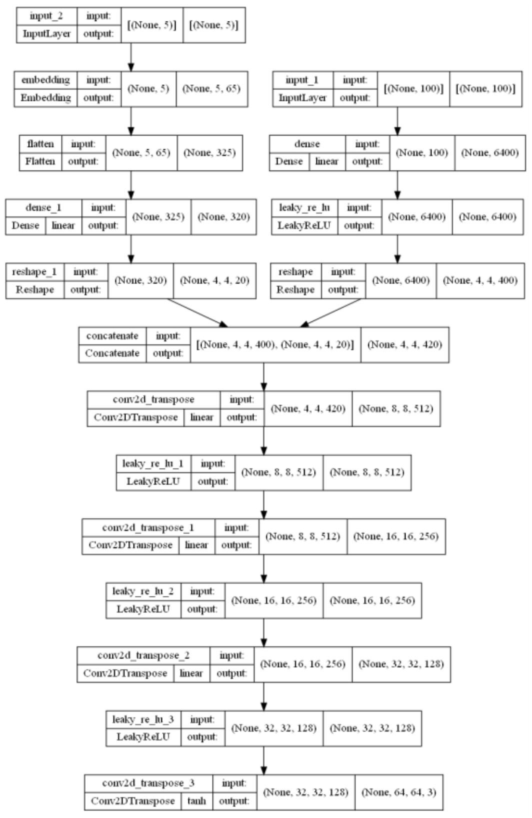

input_2 | InputLayer | input: [(None, 5)] | output: [(None, 5)]
embedding | Embedding | input: (None, 5) | output: (None, 5, 65)
flatten | Flatten | input: (None, 5, 65) | output: (None, 325)
dense_1 | Dense | linear | input: (None, 325) | output: (None, 320)
reshape_1 | Reshape | input: (None, 320) | output: (None, 4, 4, 20)
input_1 | InputLayer | input: [(None, 100)] | output: [(None, 100)]
dense | Dense | linear | input: (None, 100) | output: (None, 6400)
leaky_re_lu | LeakyReLU | input: (None, 6400) | output: (None, 6400)
reshape | Reshape | input: (None, 6400) | output: (None, 4, 4, 400)
concatenate | Concatenate | input: [(None, 4, 4, 400), (None, 4, 4, 20)] | output: (None, 4, 4, 420)
conv2d_transpose | Conv2DTranspose | linear | input: (None, 4, 4, 420) | output: (None, 8, 8, 512)
leaky_re_lu_1 | LeakyReLU | input: (None, 8, 8, 512) | output: (None, 8, 8, 512)
conv2d_transpose_1 | Conv2DTranspose | linear | input: (None, 8, 8, 512) | output: (None, 16, 16, 256)
leaky_re_lu_2 | LeakyReLU | input: (None, 16, 16, 256) | output: (None, 16, 16, 256)
conv2d_transpose_2 | Conv2DTranspose | linear | input: (None, 16, 16, 256) | output: (None, 32, 32, 128)
leaky_re_lu_3 | LeakyReLU | input: (None, 32, 32, 128) | output: (None, 32, 32, 128)
conv2d_transpose_3 | Conv2DTranspose | tanh | input: (None, 32, 32, 128) | output: (None, 64, 64, 3)

Figure 1: Generator Model Architecture

Layer	Input	Filter	Stride	Activation	Output
FullyConnected Layer for z	100 x 1	-	-	Leaky Relu	6400 x 1
Reshape for z	6400 x 1	-	-	-	4 x 4 x 400
Embedding Layer for y	5 x 1	-	-	-	5 x 65
Flatten for y	5 x 65	-	-	-	325 x 1
Fully Connected for y	325 x 1	-	-	-	320 x 1
Reshape (for y)	320 x 1	-	-	-	4 x 4 x 20
Concatenate reshaped **z** and **y**	4 x 4 x 400 4 x 4 x 20	-	-	-	4 x 4 x 420
Conv2DTranspose	4 x 4 x 420	5	2	LeakyRelu	8 x 8 x 512
Conv2DTranspose	8 x 8 x 512	5	2	LeakyRelu	16 x 16 x 256

Conv2DTranspose	16 x 16 x 256	5	2	LeakyRelu	32 x 32 x 128
Conv2DTranspose	32 x 32 x 128	5	2	Tanh	64 x 64 x 3

Table 1: Table showing Generator Architecture in Detail

Figure 1 and Table 1 show the details of the generator architecture. Figure 1 is a visual representation of how the layers interconnected and Table 1 provides more details on the components of each layer. The z noise vector of size 100 x 1 is connected to a fully connected layer and then reshaped into a 4 x 4 x 20 Tensor. The conditional vector y of size 5 x 1 is connected to an embedded layer which outputs a 5 x 65 array. This is then connected to a fully connected layer which produces a 320 x 1 vector. This is then reshaped into a 4 x 4 x 420 tensor. This reformatted y is then concatenated with the reformatted noise vector z. The concatenation is then fed through four LeakyReLU convolution transpose layers whose details are described in Table 1. The final Output Layer is also a convolution Transpose layer but instead uses Tanh for its activation function. This output is then fed into the discriminator.

Discriminator Architecture:

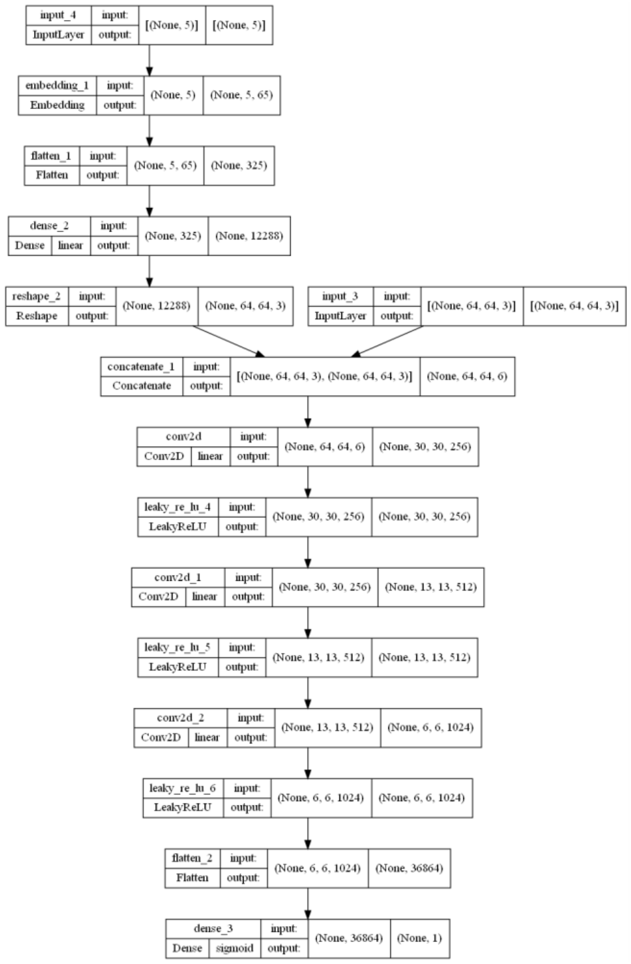

input_4 InputLayer input: output: [(None, 5)] [(None, 5)]
embedding_1 Embedding input: output: (None, 5) (None, 5, 65)
flatten_1 Flatten input: output: (None, 5, 65) (None, 325)
dense_2 Dense linear input: output: (None, 325) (None, 12288)
reshape_2 Reshape input: output: (None, 12288) (None, 64, 64, 3)
input_3 InputLayer input: output: [(None, 64, 64, 3)] [(None, 64, 64, 3)]
concatenate_1 Concatenate input: output: [(None, 64, 64, 3), (None, 64, 64, 3)] (None, 64, 64, 6)
conv2d Conv2D linear input: output: (None, 64, 64, 6) (None, 30, 30, 256)
leaky_re_lu_4 LeakyReLU input: output: (None, 30, 30, 256) (None, 30, 30, 256)
conv2d_1 Conv2D linear input: output: (None, 30, 30, 256) (None, 13, 13, 512)
leaky_re_lu_5 LeakyReLU input: output: (None, 13, 13, 512) (None, 13, 13, 512)
conv2d_2 Conv2D linear input: output: (None, 13, 13, 512) (None, 6, 6, 1024)
leaky_re_lu_6 LeakyReLU input: output: (None, 6, 6, 1024) (None, 6, 6, 1024)
flatten_2 Flatten input: output: (None, 6, 6, 1024) (None, 36864)
dense_3 Dense sigmoid input: output: (None, 36864) (None, 1)

Figure 2: Discriminator Model Architecture

Layer	Input	Filter	Stride	Activation	Output
Embedding Layer for y	5 x 1	-	-	-	5 x 65
Flatten for y	5 x 65	-	-	-	325 x 1
Fully Connected for y	325 x 1	-	-	-	12288 x 1
Reshape (for y)	12288 x 1	-	-	-	64 x 64 x 3
Concatenate x and reshaped **y**	64 x 64 x 3 64 x 64 x 3	-	-	-	64 x 64 x 6
Conv2DLayer1	64 x 64 x 6	5	2	LeakyRelu	30 x 30 x 256
Conv2DLayer2	30 x 30 x 256	5	2	LeakyRelu	13 x 13 x 512
Conv2DLayer3	13 x 13 x 512	3	2	LeakyRelu	6 x 6 x 1024
Flatten	6 x 6 x 1024	-	-	-	36864 x 1
Fully Connected Layer	36864 x 1	-	-	Sigmoid	1 x 1

Table 2: Table showing Discriminator Architecture in Detail

Figure 2 and Table 3 show the detailed schematics of the discriminator architecture. Similar to the generator, the conditional information y is reformatted through an embedded and fully connected layer in order to get a similar shape to the input image x which has dimension of 64 x64 x3. The reformatted y is then concatenated with x. The result of the concatenation is fed through 3 successive Leaky RelU activated convolution layers whose details are listed in Table 2. After the convolution layers, we then connect to a Fully connected Layer through the sigmoid activation function to get the probability of the input data.

My generator and discriminator models are a revised version of the recommended model in the Project Guidelines. I broadcast my input vectors using the embedded layer and Fully connected Layer and then reshaping. I also changed the size of some of my filters. I did not use Batch Normalization as it gave poor results and I also replaced the RelU activation with Leaky ReLU as it gave better performance. I got the idea for the reformatting using embedded and fully connected layer from several online archives and machine learning discussion forums.

Model Training Objective and procedure

We train our model based on the objective function we described in the earlier section.

We are given a real image data batch **D** = {**x**(m),**y**(m)}, m=1,…,M. **y**(m) is the 5x1 binary vector attribute for a given face image **x**(m) of dimension 64x64x3. We sample a noise vector **z** from the zero-mean gaussian distribution $N(0^{\boldsymbol{dz} \text{ x } \mathbf{1}}, \boldsymbol{I}^{\boldsymbol{dz} \text{ x } \boldsymbol{dz}})$ which together with **y**(m), will be fed as input to the generator. The dimension of the **z** vector, $\boldsymbol{dz}$ is a hyperparameter. After being fed its inputs, the generator produces a synthetic image $\widehat{\boldsymbol{x}}$(m) of same dimension as **x**(m) . The real or fake images can then be fed into the discriminator with their condition **y**(m) to produce its probability of being real.

Let us call our objective function GAN(Θ, ϕ) where Θ are the generator parameters, and ϕ are the discriminator parameters.

Then based on our earlier described problem setting we define GAN(Θ, ϕ) as:

$$\text{GAN}(\Theta, \phi) = \mathrm{E}_{\mathrm{x,y} \sim p_{Data}(\mathbf{x},\mathrm{y})}[\log D(\mathrm{x}, \mathrm{y}; \phi)] + \mathrm{E}_{\mathrm{y} \sim p_{\mathrm{y}}(\mathrm{y}), \mathrm{z} \sim p_{\mathrm{z}}(\mathrm{z})}[\log (1 - D(G(\mathrm{z}, \mathrm{y}; \theta), \mathrm{y}; \phi))]$$

Where $D(\mathrm{x}, \mathrm{y}; \phi)$ is the discriminator output and $G(\mathrm{z}, \mathrm{y}; \theta)$ is the generator output

Our training objective is to find the optimal model parameters $\{\theta^*, \phi^*\}$ such that:

$$\theta^*, \phi^* = \arg \min_{\theta} \max_{\phi} \; E_{x,y \sim p_{Data}(\mathbf{x},\mathbf{y})}[log\, D(x, y; \phi)] + E_{y \sim p_y(\mathbf{y}), z \sim p_z(\mathbf{z})}[log\, (1 - D(G(z, y; \theta), y; \phi))]$$

The above is our optimization problem.

For 1,….,M datapoints, we can define GAN(Θ, ϕ) as:

$$\text{GAN}(\Theta, \phi) = \left[\left[\frac{1}{\mathrm{M}} \sum_{\mathrm{m}=1}^{\mathrm{M}} log\, D(x(\mathrm{m}), y(\mathrm{m}); \phi)\right] + \frac{1}{\mathrm{M}} \sum_{\mathrm{m}=1}^{\mathrm{M}} log\, (1 - D(G(z(\mathrm{m}), y(\mathrm{m}); \theta), y(\mathrm{m}); \phi))\right]$$

$$= \left[\left[\frac{1}{\mathrm{M}} \sum_{\mathrm{m}=1}^{\mathrm{M}} log\, p(x(\mathrm{m}), y(\mathrm{m}); \phi)\right] + \frac{1}{\mathrm{M}} \sum_{\mathrm{m}=1}^{\mathrm{M}} log\, (1 - p(\hat{x}(\mathrm{m}), y(\mathrm{m}); \phi))\right]$$

<u>We perform the optimization in two stages</u>:

For the first stage, we solve for the parameters of the discriminator given M pairs of real and fake samples:

$$\phi^{*\mathrm{t}+1} = \arg \max_{\phi} \left[\left[\frac{1}{M} \sum_{m=1}^{M} log\, p(x(m), y(m); \phi)\right] + \frac{1}{M} \sum_{m=1}^{M} log\, (1 - p(\hat{x}(m), y(m); \phi))\right]$$

We perform the above maximization through gradient ascent. To calculate the loss, we first initialize the noise vector and then pass it into the generator together with the corresponding condition in order to generate the fake data samples. We then pass both the real datapoints and the fake data together with their corresponding condition in order to generate their probabilities. We calculate the loss using the respective probabilities. We then use backpropagation to calculate the gradient of ϕ. Then in order to maximize, we perform gradient ascent and update ϕ iteratively before moving to the second stage. The number of iterations is a hyperparameter.

For the second stage, given the updated discriminator parameters, we then solve for the parameters of the generator:

$$\theta^{*\mathrm{t}+1} = \arg \min_{\theta} \frac{1}{\mathrm{M}} \sum_{\mathrm{m}=1}^{\mathrm{M}} log\, (1 - p(\hat{x}(\mathrm{m}), y(\mathrm{m}); \phi^{*t+1})) \quad \text{Where } \hat{x}(\mathrm{m}) = G(z(\mathrm{m}), y(\mathrm{m}); \theta)$$

To perform the optimization, we generate fake samples from the generator's current weights. The fake samples are then fed into the discriminator together with their condition in order to generate their probabilities. We can calculate the loss using the generated probabilities. We also use back propagation to get the gradients and then update the generator parameters iteratively through gradient descent. The number of iterations is also a hyper parameter. We then start again from Stage 1 and iterate through the two stages until convergence.

<u>EXPERIMENT</u>

<u>Dataset</u>

We make use of the CelebFaces Attributes (CelebA) dataset. It is a dataset of different celebrity face images in different backgrounds and poses. The dataset originally has 202,559 datapoints with 40 attributes. The images in the dataset are originally large scale (218 x 218 x 3) have been cropped for our training process into smaller sizes of (64 x 64 x 3). The attribute encoding has also been changed from {-1,1} to {0,1}. We also select only five attributes to use: Black hair, Male, Oval Face, Smiling, and Young. Our attribute vector has dimension 5 x 1 in the respective order listed.

<u>Programming Language and IDE</u>

I used python 3.9.7 with Pycharm IDE through the Anaconda environment.

<u>Packages</u>

I used the following imported packages: import math, pickle, random, numpy, pathlib, matplotlib.pyplot, time, tensorflow, pandas, sklearn.utils

I made use of tensorflow 2.8.0 with the Keras API

I used the Nvidia 2070RTX 8gb GPU for the training process.

<u>Hyperparameter tuning</u>

We have Five major hyperparameters: Learning rate, batch size, Noise vector dimension, Number of iterations for updating each step of the optimization process, and Regularization

<u>Learning rate</u>:

After several trials , I discovered that the Adam Optimization algorithm produced better results than the Stochastic Gradient descent. I initially tried learning rates of 0.1,0.01,0.001 but they caused too many oscillations with bad results. I settled for a learning rate of 0.0001 with beta = 0.5. I used the same learning rate for the generator and discriminator updates.

<u>Batch Size</u>:

I settled with a batch size of 128 which gave the best performance. It performed better than other lower batch sizes. I could not test a higher batch size because my PC ran out of memory

<u>Noise Vector Dimension</u>:

I settled with the recommended noise vector dimension of 100 x 1

<u>Number of Iterations per step:</u>

I settled with only a single iteration for updating the discriminator and generator. Higher iterations did not produce better results and were also slow to train.

<u>Regularization:</u>

I did not employ Regularization in my training process

Number of Epochs:

I trained my model for 30 epochs. The training lasted for roughly 12 hours.

Generated Images

a) **Conditional Information: [1,0,1,1,0]**
FID Score: 63.15 IS score: 2.14

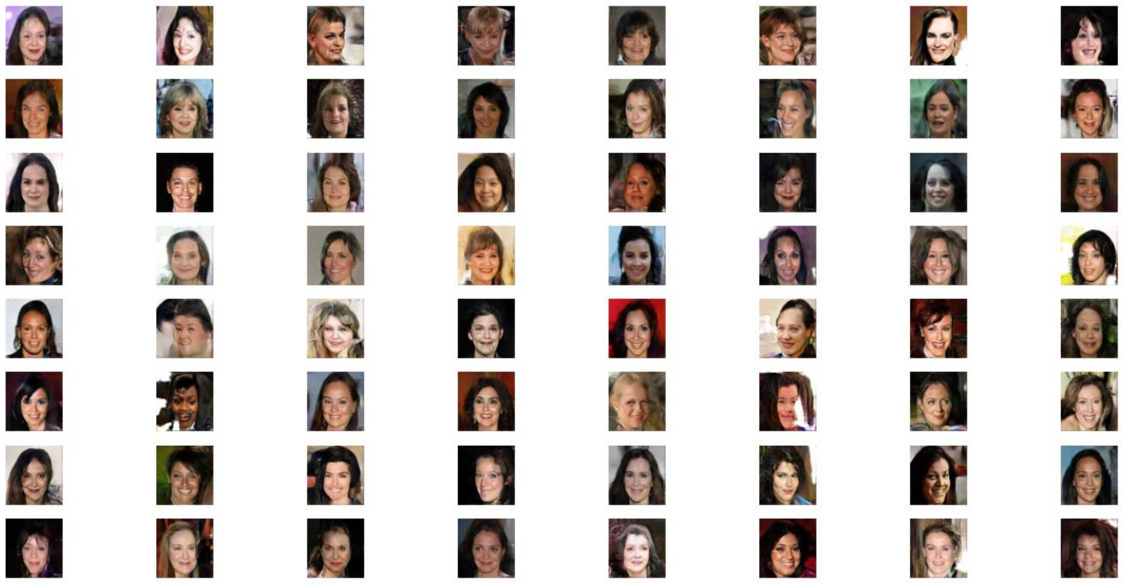


Figure 3a: 64 Generated Images with condition Black hair = 1, Male = 0, Oval Face = 1, Smiling = 1, Young = 0

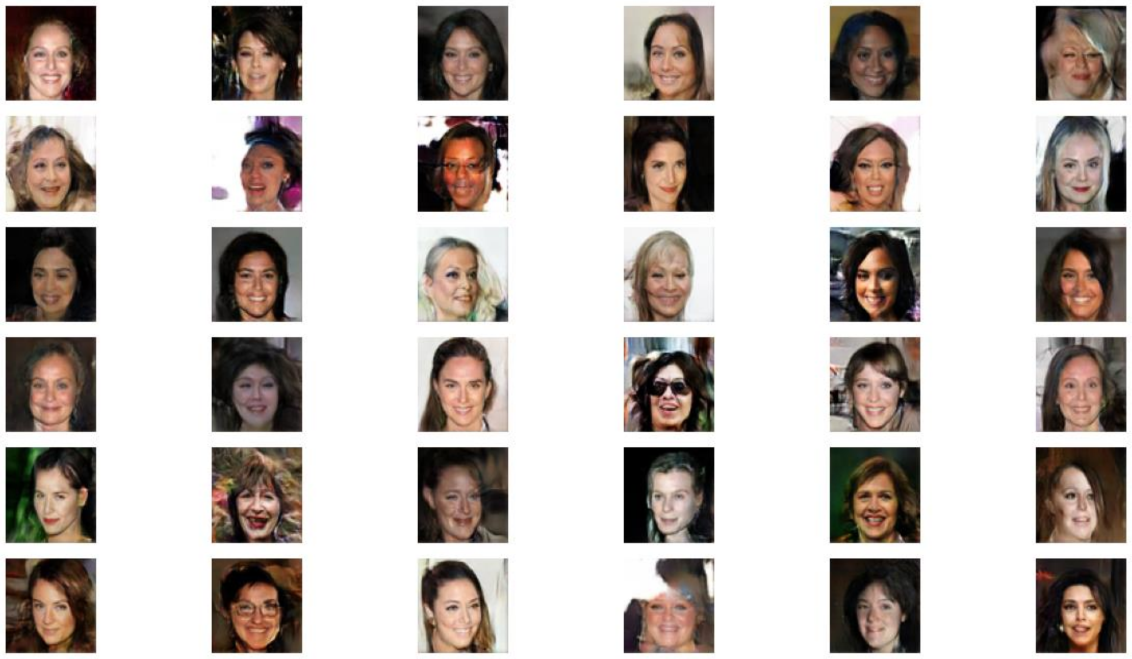


Figure 3b: 36 Generated Images with condition Black hair = 1, Male = 0, Oval Face = 1, Smiling = 1, Young = 0

b) **Conditional Information: [1,1,0,0,1]**

FID Score: 60.15 IS score: 2.284

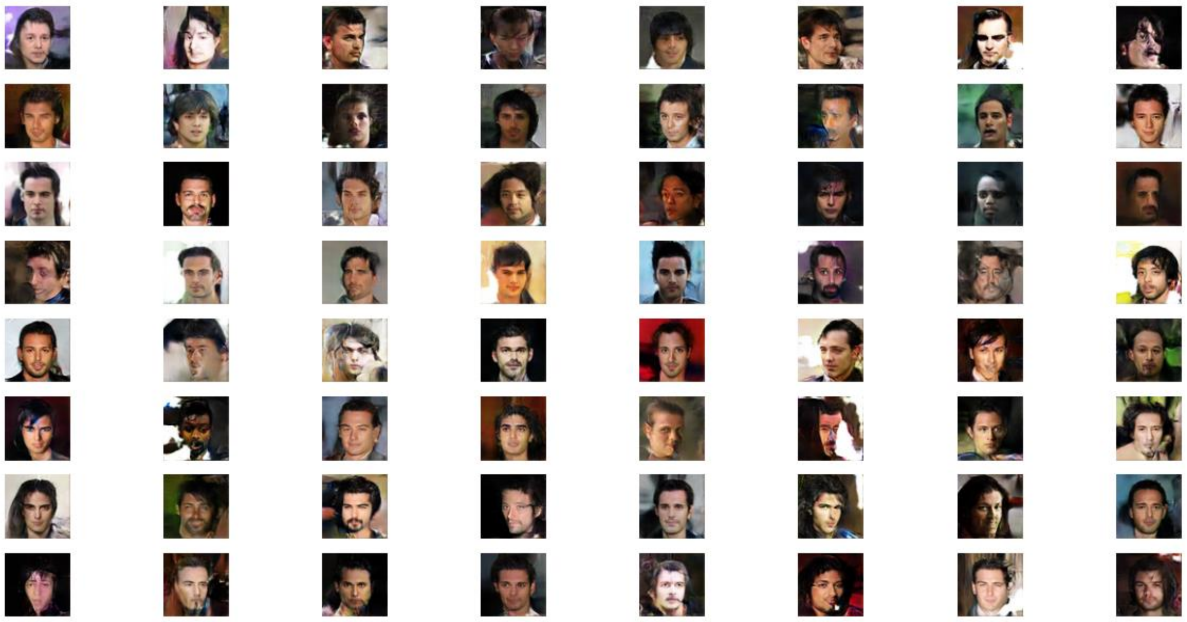


Figure 4a: 64 Generated Images with condition Black hair = 1, Male = 1, Oval Face = 0, Smiling = 0, Young = 1

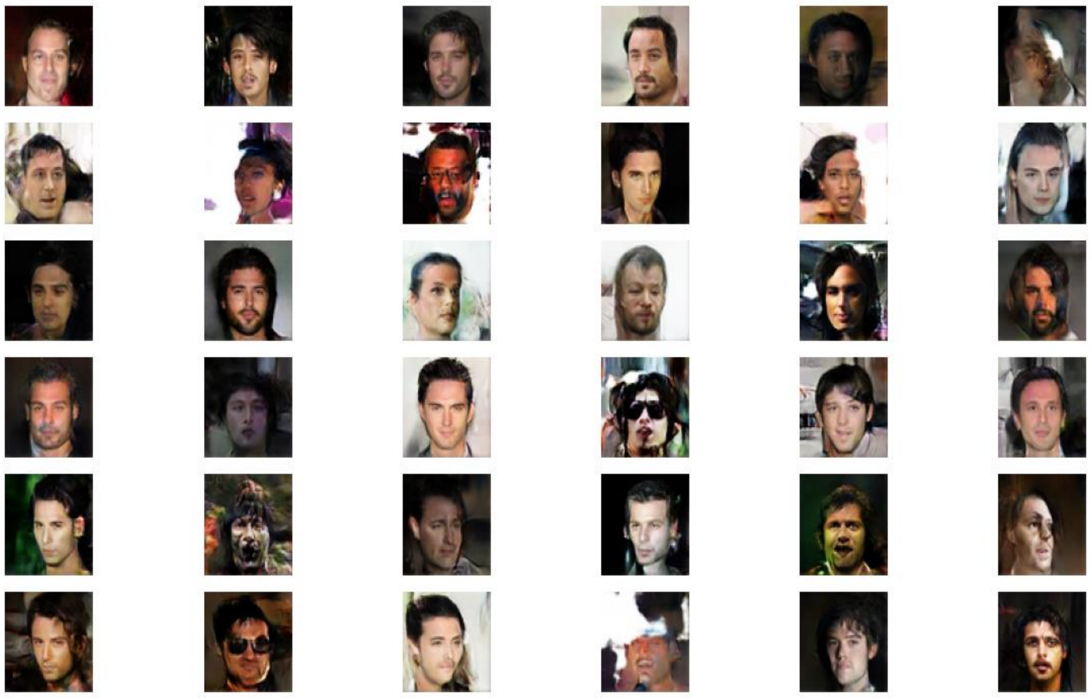
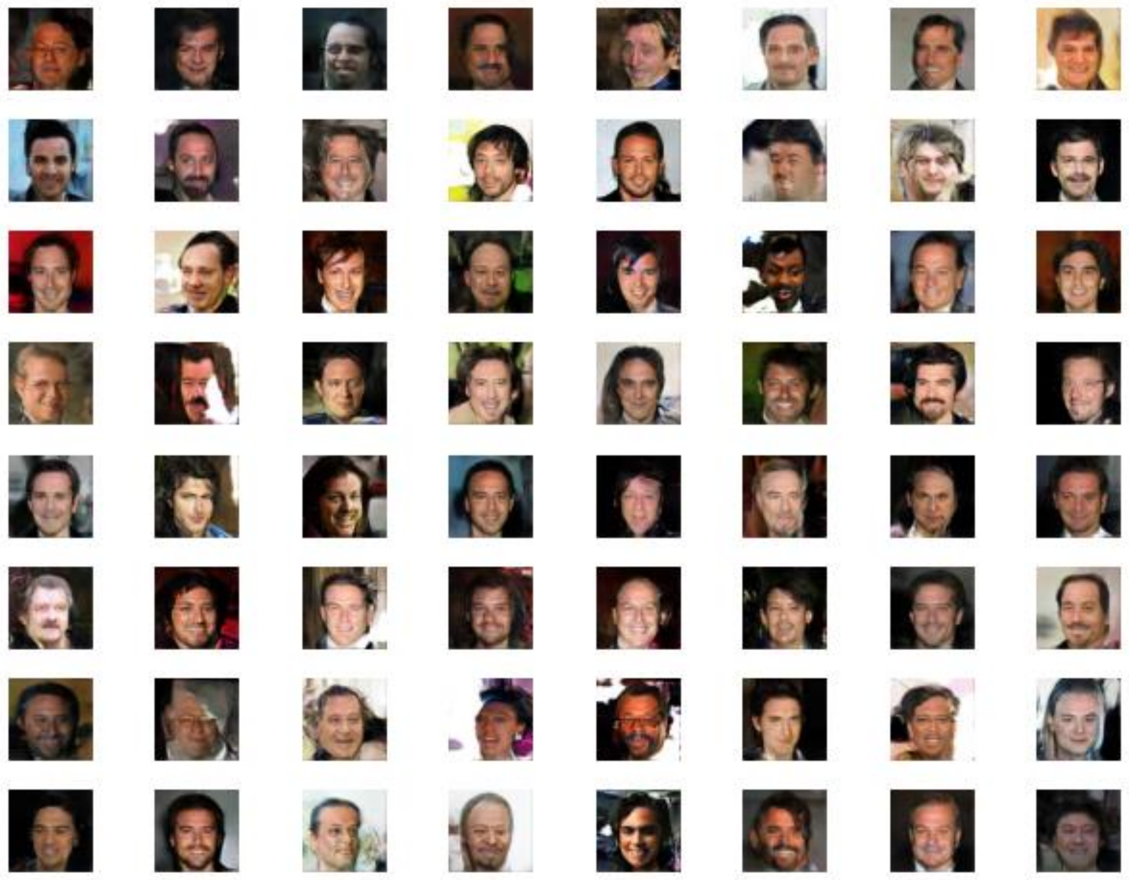


Figure 4b: 36 Generated Images with condition Black hair = 1, Male = 1, Oval Face = 0, Smiling = 0, Young = 1

c) <u>Conditional Information: [1,1,1,1,0]</u>
FID Score: 70.8 IS score: 2.01



<u>Figure 5a: 64 Generated Images with condition Black hair = 1, Male = 1, Oval Face = 1, Smiling = 1, Young = 0</u>

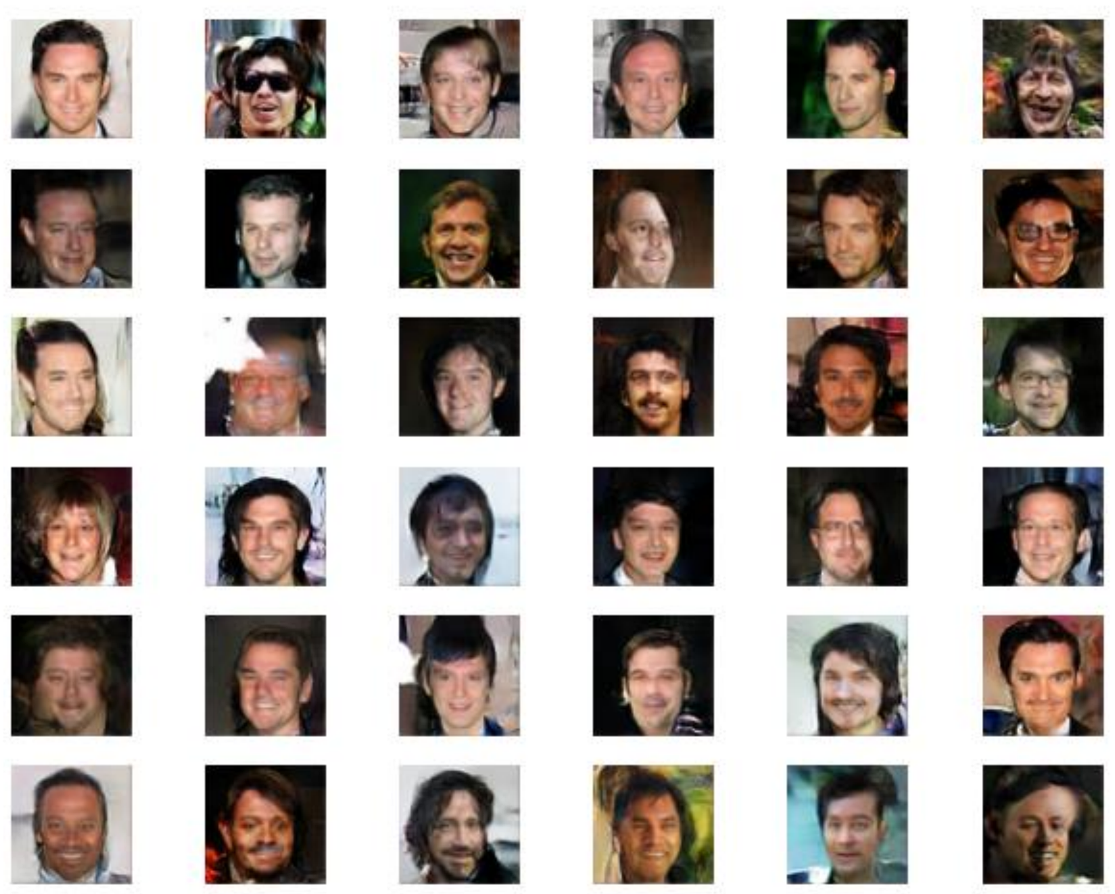


Figure 5b: 36 Generated Images with condition Black hair = 1, Male = 1, Oval Face = 1, Smiling = 1, Young = 0

d) Condition = [0,0,1,1,0]

FID Score: 40.53 IS score: 2.352

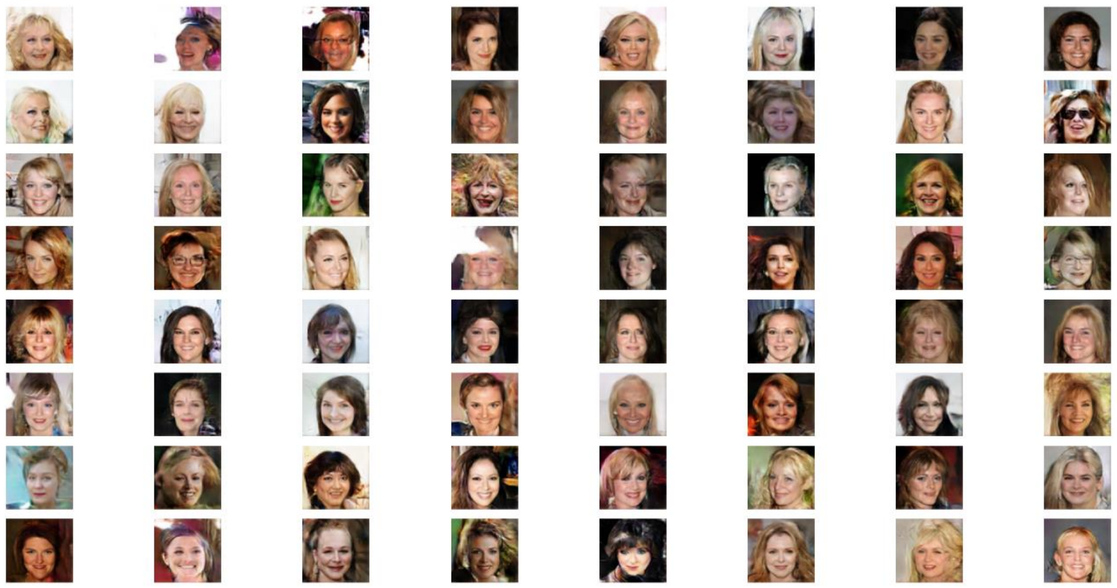


Figure 6a: 64 Generated Images with condition Black hair = 0, Male = 0, Oval Face = 1, Smiling = 1, Young = 0

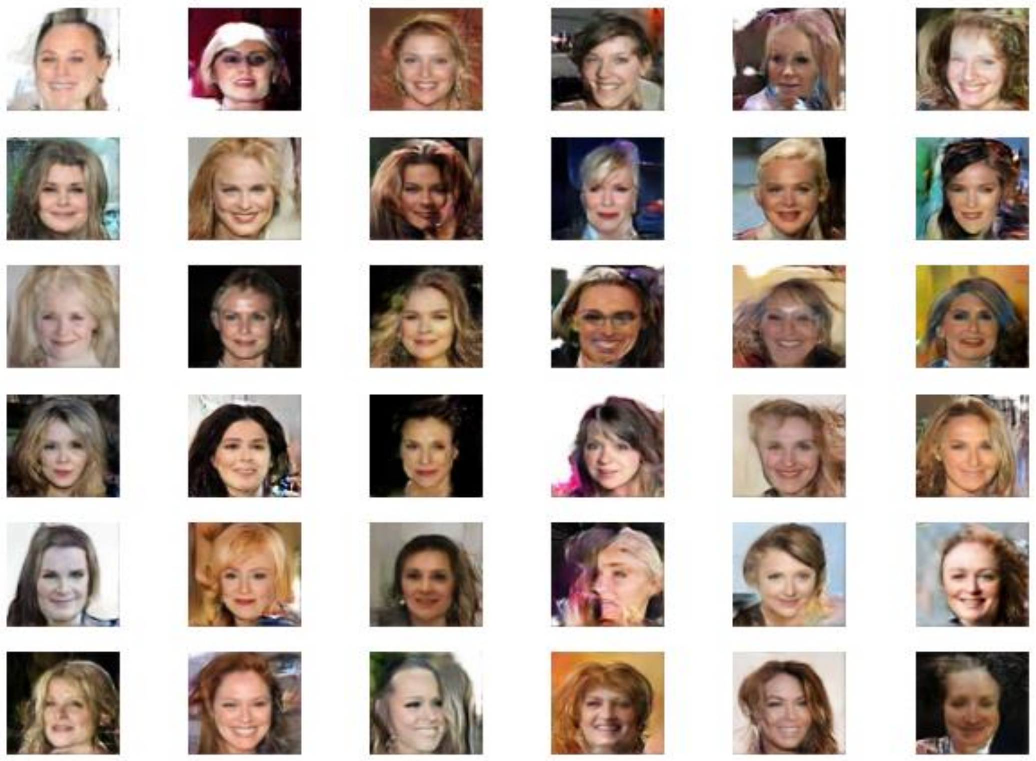


Figure 6b: 36 Generated Images with condition Black hair = 0, Male = 0, Oval Face = 1, Smiling = 1, Young = 0

ANALYSES OF EXPERIMENTAL RESULTS

The generated images show that the conditional GAN has been successful in generating synthetic Images that meet the given conditions. The images are not as highly resolute or precise as the real images but the similarities are clear to see. We vary the experiment across four different attribute vectors as indicated in the image titles. We generate 100 images for each attribute vector. The 100 images for a condition is split into two sub images of 64 and 36 in order to enable better visualization. It is clear that in most of the images, the desired images meet their respective conditions. We however note that some images have too much noise and as such are not easy to see if the conditions are met. But the majority of them satisfy their conditions. The FID and IS scores for the images are also given. The IS scores are above the desired value of 2.0. Even though the images visually correlate to that of real images with the given conditions, the FID scores are above 15.0. I am not sure why this happened. Overall, with visual inspection, the model seems to have performed relatively well on the human face generation task.

ABLATION STUDY

Here we discuss the influence of hyper-parameter tuning on the model performance.

Optimizer, learning rate and batch size: After several trials , I discovered that the Adam Optimization algorithm produced better results than the Stochastic Gradient descent. Adam had lower oscillations during my training and converged faster. SGD resulted in noisy undiscernible generated images several times. I found Adam to provide more consistent performance on average with relatively decent images. For the learning rate, I initially tried learning rates of 0.1,0.01,0.001 but they caused too many oscillations with bad results. I settled for a learning rate of 0.0001. I used the same learning rate for the generator and discriminator updates. I started with small batch sizes of 16,32,64, 128. I realized that my performance increased slightly with higher batch sizes. I settled at 128 because my training memory was insufficient for a higher size.

Epochs: After training with different hyperparameters as described above; my chosen rate of 0.0001, batch size of 128, I realized that an epoch training amount of 30 was the best for my model. On average, after 30 epochs, I did not see significant improvements in the model. Hence, I settled with epoch number of 30. Additionally, the training time was quite large and so I had to settle with a small number of epochs in other to be able to try different parameters.

Noise Vector Dimension:

I discovered that the recommended noise vector of 100 x 1 consistently gave the best performance. Any higher or lower did not improve the model

Number of Iterations per step:

The single iteration per step gave the best performance. Increasing the number of iterations increased the amount of noise in my generated images.

CONCLUSION

This project taught me the importance of good model architecture design and hyperparameter tuning in achieving good performance. My most important observation from implementing this project is that the Conditional Generative Adversarial Network is quite difficult to train. I had to try several hyperparameter variations across a week in order to get decent results. Also, the model was quite slow to train on my PC. Another issue was implementing the example architecture in the project instructions. Batch normalization had a very bad effect on my model performance. The ReLU activation also had a bad effect. I decided to go with the Leaky ReLU and no batch normalization which gave me significantly better results. Additionally, slight changes in my architecture such as at what point to concatenate the input with the conditional information had huge variations in my model performance. The suggested method of broadcasting the conditional information into my model also had very low performance for me. I found that I got much better performance by formatting the conditional information by using the embedded layer and dense layer and then reshaping into the desired dimension. I was able to gain information about the performance gain of the embedded layer by reading several online documentations. I assume that with even more data points, the model can perform better. It will be interesting to see the model performance on the original uncropped data to see how high resolution the generated images can become. Overall, I am quite impressed with the GAN and the images it generated in my experimental results.