ECSE 6850 Final Project
Conditional Generative Adversarial Network

Soha Yusuf (662011092)
Rensselaer Polytechnic Institute
`yusufs@rpi.edu` | 06 May 2022

Abstract

We train a Conditional Generative Adversarial Network (cGAN) on CelebA dataset containing 202,599 RGB images and one-hot attribute vectors (condition) associated with each image. In cGAN, generator takes condition y and noise z as input and passes them through a de-convolutional layer to create corresponding embedding y' and z', which are then passed through rest of generator model to output a 64x64x3 image. Discriminator takes concatenated real and fake image I along with the condition y which is broadcasted and passed through a convolutional layer. Image I is also passed through a convolutional layer in the discriminator to create embedding of the same dimension 32x32x64. Discriminator and generator models are trained alternatively in TensorFlow 2.6.0 with Adam optimizer and binary cross-entropy loss function. After training, the trained generator is able to generate images according to the given condition.

1 Introduction

1.1 GAN

Generative Adversarial Networks (GANs) [2] are unsupervised learning models that use two networks along with adversarial training to output new data which is similar to the input data. GANs involve a generator G that captures the data distribution, and a discriminator D that estimates the probability that the sample is from training data rather than generated data. A typical structure of GAN is shown in Figure 1 where generator takes random noise and outputs an image and discriminator takes real and fake image and outputs the probability of image being real.

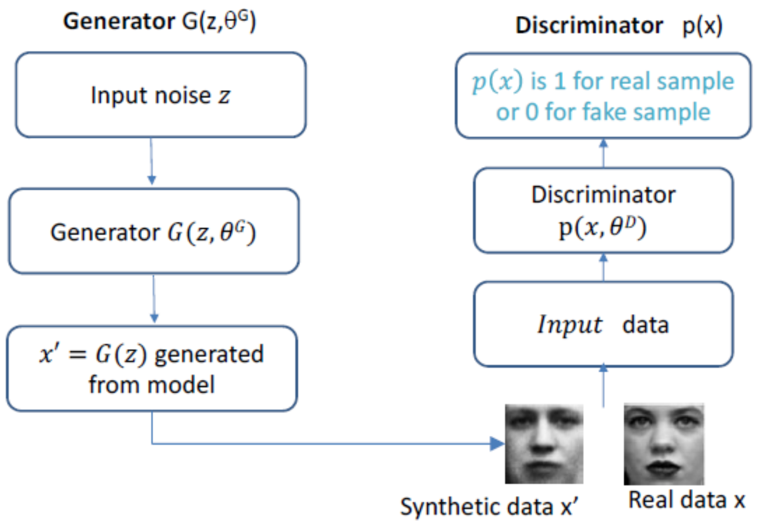


Figure 1: Typical architecture of GAN [3]

1.2 cGAN

Conditional Generative Adversarial Networks (cGANs) [5] are a modification of the original GAN where the input to discriminator is data and label y instead of only data, and the input to generator is noise z and label y instead of only noise. Condition y can be any kind of auxiliary information but in our case y represents one-hot labels for 5 classes. The addition of labels allows the generator to learn multiple representations of different training data classes, allowing for the ability to explicitly control the output of the generator. A typical cGAN architecture is shown in Figure 2:

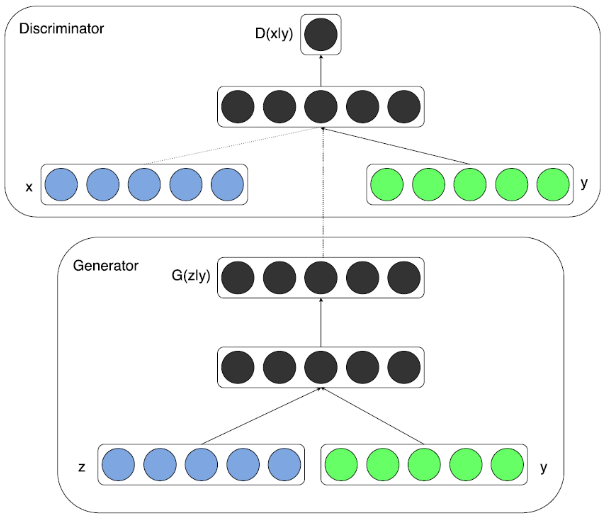


Figure 2: Typical architecture of cGAN [5]

1.3 Human face generation

GANs are able to generate deceptively real images that are very difficult to distinguish from actual data. However, such models require large datasets to realistically generate images in the corresponding domain. To overcome the challenge of obtaining large training data, conditioned data is used in conditional GANs for training. cGANs can be used to:

- generate more samples for the skewed class to balance the imbalanced datasets,
- associate the generated samples with the class labels, so that these representations can also be used for other downstream tasks, and
- perform image-to-image translation e.g Pix2Pix GAN.

In this project, we are required to train a conditional GAN on CelebA dataset which is described in more detail in section 3.1. We train generator model and discriminator model alternatively. Generator takes noise of latent dimension 100 and label of dimension 5 and outputs an image of dimension 64x64x3. Discriminator takes images of dimension 64x64x3 and outputs the probability of image being real. Model architecture, optimization method and experimental results are discussed in the following sections.

2 Method

2.1 Problem Setting

GANs consist of two 'adversarial' models: generator that captures data distribution and discriminator that estimates the probability that sample came from training data. G and D are non-linear functions i.e deep convolutional networks.

In GANs, given training data $D = \{x^{(m)}\}$, m = 1,2,...M, we have to learn a generative model $p_g(x|\Theta)$ that best captures $p_D(x)$ by minimizing the Jenson divergence between $p_g(x|\Theta)$ and $p_D(x)$ which is adversarial learning. Jensen-Shannon divergence is computed as follows:

$$\begin{aligned} JSD(P_d(x)||P_g(x|\theta)) &= \frac{1}{2}KL(P_d(x)||\frac{P_d(x)+P_g(x|\theta)}{2}) + \frac{1}{2}KL(P_g(x)||\frac{P_d(x)+P_g(x|\theta)}{2}) \\ &= log2 + \frac{1}{2}[KL(P_d(x)||P_d(x)+P_g(x|\theta)) + KL(P_g(x)||P_d(x)+P_g(x|\theta))] \\ &= log2 + \frac{1}{2}[E_{P_d}(logf(x)) + E_{P_g}(log(1-f(x)))] \end{aligned} \tag{1}$$

where,

$$f(x) = \frac{P_d(x)}{P_d(x)+P_g(x)} \tag{2}$$

The proof of equation(2) can be found in [2]. Minimizing Jensen-Shannon divergence can be formulated as min-max optimization:

$$\{\theta^*, \phi^*\} = min_\theta max_\phi E_{P_d(x)}[f(x|\phi)] + E_{P_g(x|\theta)}[log(1-f(x|\phi)] \tag{3}$$

where, $P_g(x|\theta)$ is generative model distribution, $P_d(x)$ is true data distribution, and $f(x|\phi) = \dfrac{P_d(x)}{P_d(x)+P_g(x)}$ is discriminative model output.

In this project, we will optimize a conditional GAN where the generator and discriminator will take conditioned data. We will first update the discriminator parameters Θ_D while fixing generator parameters and then update generator parameters Θ_G while fixing Θ_D. Discriminator tries to distinguish real data samples $x^{(i)}$ from fake generated samples $\hat{x}^{(i)}$ and generator tries to generate realistic samples $\hat{x}^{(i)}$. We update the discriminator using stochastic-gradient ascent:

$$\begin{aligned} \Theta_D &\leftarrow \Theta_D + \lambda^D \nabla_{\Theta_D} \mathcal{L}_d(\{x^{(i)}, y^{(i)}, z^{(i)}\}_{i=1}^n, \Theta_G, \Theta_D) \\ \Theta_D &\leftarrow \Theta_D + \lambda^D \frac{\partial[\frac{1}{n}\sum_{i=1}^n logp(x^{(i)}, y^{(i)}; \Theta_D) + log(1-p(\hat{x}^{(i)}, y^{(i)}; \Theta_D))]}{\partial\Theta_D} \end{aligned} \tag{4}$$

Generator parameters are also updated using stochastic-gradient descent:

$$\begin{aligned} \Theta_G &\leftarrow \Theta_G - \lambda^G \nabla_{\Theta_G} \mathcal{L}_g(\{x^{(i)}, y^{(i)}, z^{(i)}\}_{i=1}^n, \Theta_G, \Theta_D) \\ \Theta_G &\leftarrow \Theta_G - \lambda^G \frac{\partial[\frac{1}{n}\sum_{i=1}^n logp(G(z^{(i)}, y^{(i)}; \Theta_G); \Theta_D)}{\partial\Theta_G} \end{aligned} \tag{5}$$

2.2 Model Architecture

We use deep convolutional network for both generator and discriminator. Detailed network architecture for generator and discriminator are shown in the sections 2.2.1 and 2.2.2 respectively.

2.2.1 Generator

Generator model consist of 6 de-convolutional layers, each having ReLU activation except the last layer which uses tanh activation function. First, noise z and label y are reshaped to 1x1x100 and 1x1x5 respectively and passed through de-convolutional layers. Deconv1 and Deconv2 creates noise embedding z' of dimension 4x4x512 and label embedding y' of dimension 4x4x512. Concatenated embedding of dimension 4x4x1024 is then passed through de-convolutional layers i.e. Deconv3-6 to output a generated image of dimension 64x64x3. Details of each layer and output shapes are shown in the figure below:

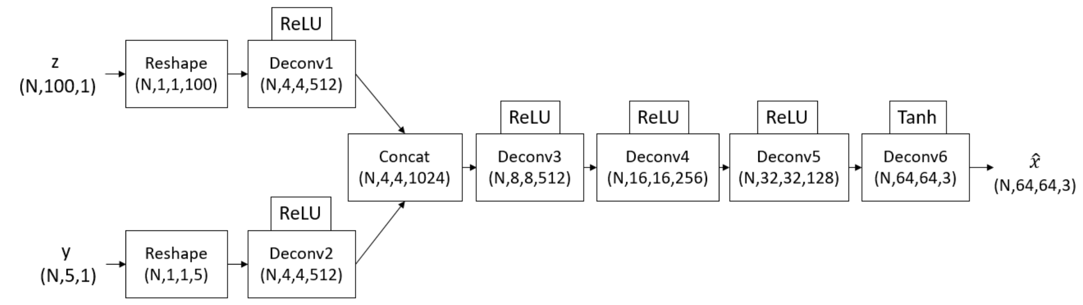
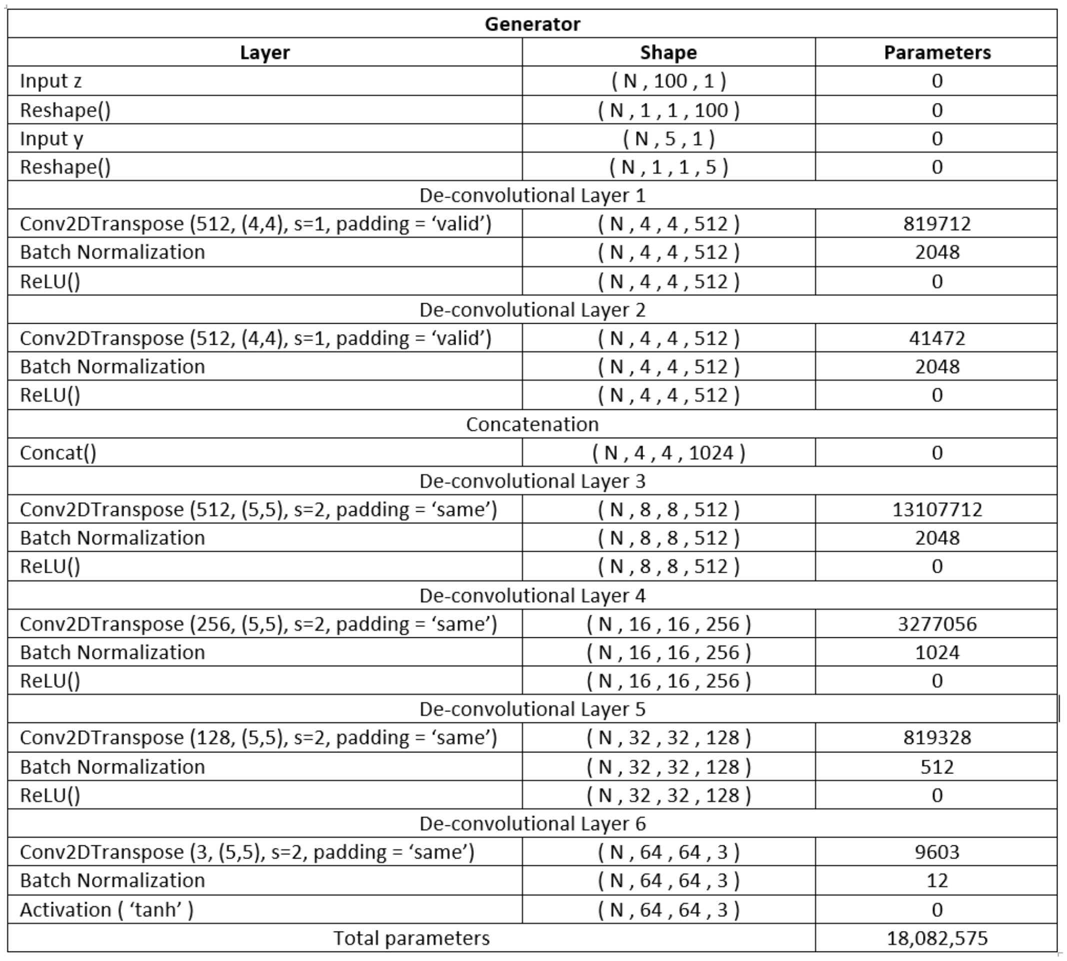


Figure 3: Network architecture of generator for cGAN, where N = batch size.



Generator

Layer	Shape	Parameters
Input z	(N , 100 , 1)	0
Reshape()	(N , 1 , 1 , 100)	0
Input y	(N , 5 , 1)	0
Reshape()	(N , 1 , 1 , 5)	0
De-convolutional Layer 1		
Conv2DTranspose (512, (4,4), s=1, padding = 'valid')	(N , 4 , 4 , 512)	819712
Batch Normalization	(N , 4 , 4 , 512)	2048
ReLU()	(N , 4 , 4 , 512)	0
De-convolutional Layer 2		
Conv2DTranspose (512, (4,4), s=1, padding = 'valid')	(N , 4 , 4 , 512)	41472
Batch Normalization	(N , 4 , 4 , 512)	2048
ReLU()	(N , 4 , 4 , 512)	0
Concatenation		
Concat()	(N , 4 , 4 , 1024)	0
De-convolutional Layer 3		
Conv2DTranspose (512, (5,5), s=2, padding = 'same')	(N , 8 , 8 , 512)	13107712
Batch Normalization	(N , 8 , 8 , 512)	2048
ReLU()	(N , 8 , 8 , 512)	0
De-convolutional Layer 4		
Conv2DTranspose (256, (5,5), s=2, padding = 'same')	(N , 16 , 16 , 256)	3277056
Batch Normalization	(N , 16 , 16 , 256)	1024
ReLU()	(N , 16 , 16 , 256)	0
De-convolutional Layer 5		
Conv2DTranspose (128, (5,5), s=2, padding = 'same')	(N , 32 , 32 , 128)	819328
Batch Normalization	(N , 32 , 32 , 128)	512
ReLU()	(N , 32 , 32 , 128)	0
De-convolutional Layer 6		
Conv2DTranspose (3, (5,5), s=2, padding = 'same')	(N , 64 , 64 , 3)	9603
Batch Normalization	(N , 64 , 64 , 3)	12
Activation ('tanh')	(N , 64 , 64 , 3)	0
Total parameters		18,082,575

Figure 4: Network architecture of generator for cGAN, where N = batch size.

2.2.2 Discriminator

Discriminator model consist of 5 convolutional layers and one fully-connected layer, each having Leaky ReLU activation except the last layer which uses sigmoid activation function. Model takes concatenated real and generated image as I along with the label y as input and outputs the probability from 0 to 1. Label y is broadcasted to y' of dimension 64x64x5 and passed through a convolutional layer with 64 (2,2) filters. Image I is also passed through a convolutional layer before concatenation with y'. Concatenated embedding of dimension 32x32x128 is then passed three convolutional layers i.e. Conv3-5. The output of Conv5 is connected to a fully-connected layer with one node and sigmoid activation function. Details of each layer and output shapes are shown in the figure below:

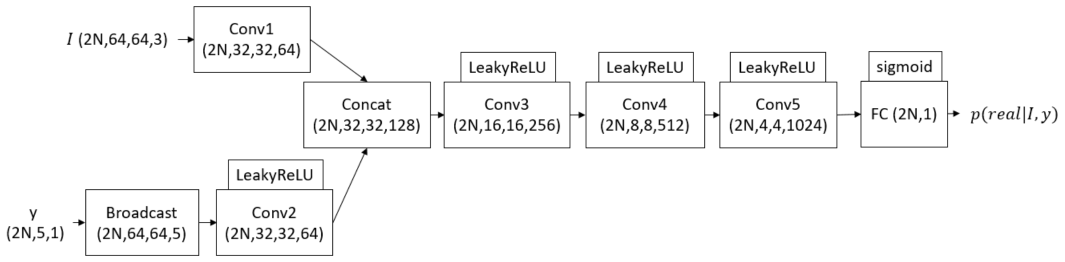
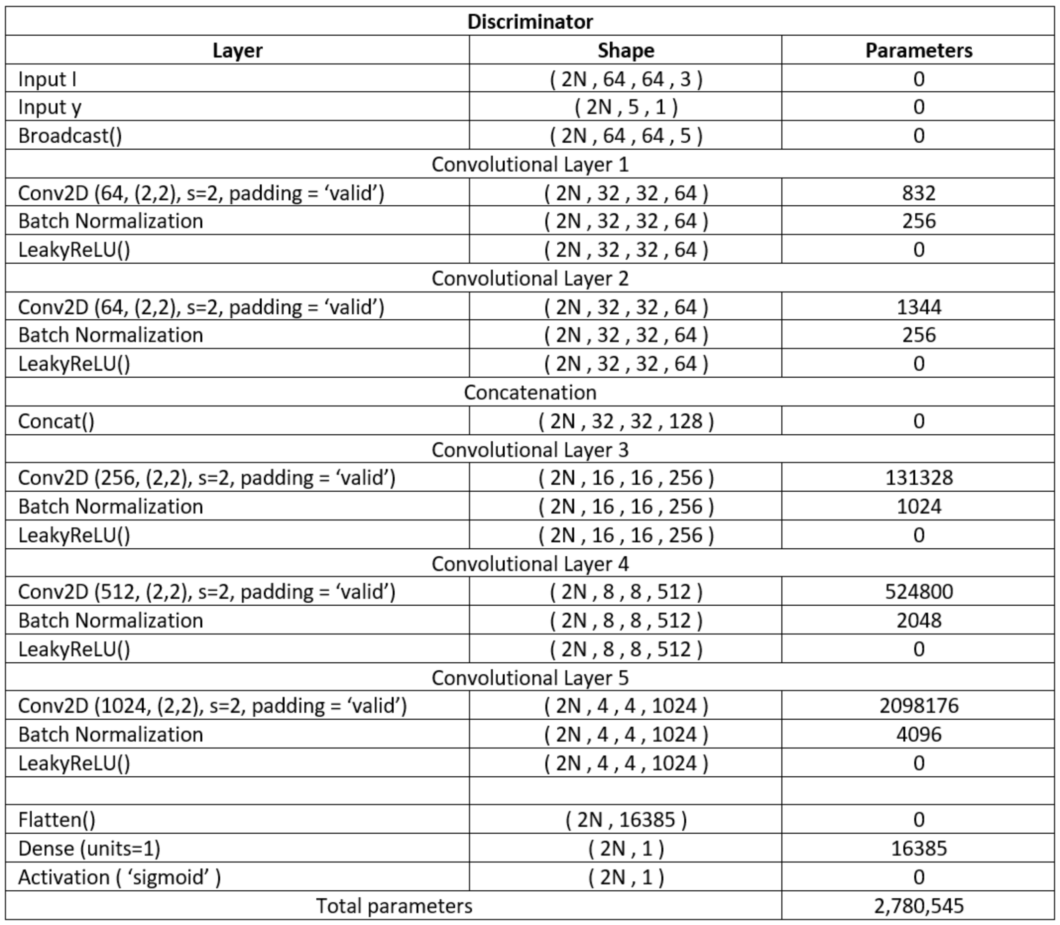


Figure 5: Network architecture of discriminator for cGAN, where N = batch size.



Discriminator		
Layer	**Shape**	**Parameters**
Input I	(2N , 64 , 64 , 3)	0
Input y	(2N , 5 , 1)	0
Broadcast()	(2N , 64 , 64 , 5)	0
Convolutional Layer 1		
Conv2D (64, (2,2), s=2, padding = 'valid')	(2N , 32 , 32 , 64)	832
Batch Normalization	(2N , 32 , 32 , 64)	256
LeakyReLU()	(2N , 32 , 32 , 64)	0
Convolutional Layer 2		
Conv2D (64, (2,2), s=2, padding = 'valid')	(2N , 32 , 32 , 64)	1344
Batch Normalization	(2N , 32 , 32 , 64)	256
LeakyReLU()	(2N , 32 , 32 , 64)	0
Concatenation		
Concat()	(2N , 32 , 32 , 128)	0
Convolutional Layer 3		
Conv2D (256, (2,2), s=2, padding = 'valid')	(2N , 16 , 16 , 256)	131328
Batch Normalization	(2N , 16 , 16 , 256)	1024
LeakyReLU()	(2N , 16 , 16 , 256)	0
Convolutional Layer 4		
Conv2D (512, (2,2), s=2, padding = 'valid')	(2N , 8 , 8 , 512)	524800
Batch Normalization	(2N , 8 , 8 , 512)	2048
LeakyReLU()	(2N , 8 , 8 , 512)	0
Convolutional Layer 5		
Conv2D (1024, (2,2), s=2, padding = 'valid')	(2N , 4 , 4 , 1024)	2098176
Batch Normalization	(2N , 4 , 4 , 1024)	4096
LeakyReLU()	(2N , 4 , 4 , 1024)	0
Flatten()	(2N , 16385)	0
Dense (units=1)	(2N , 1)	16385
Activation ('sigmoid')	(2N , 1)	0
Total parameters		2,780,545

Figure 6: Network architecture of discriminator for cGAN, where N = batch size.

2.3 Model training objective

Generator $G(z;\Theta_G)$ learns a distribution p_g that maps the joint representation of $p_z(z)$ and y to data space. Discriminator $D(x;\Theta_D)$ learns a mapping to output a probability p_g. Generator and discriminator play min-max game to optimize the training objective V(D,G):

For GAN:

$$V(D,G) = \mathbb{E}_{x\sim p_{data(x)}}[logD(x;\Theta_D)] + \mathbb{E}_{z\sim p_{z(z)}}[log(1-D(G(z;\Theta_G);\Theta_D))] \tag{6}$$

For cGAN:

$$V(D,G) = \mathbb{E}_{x,y\sim p_{data(x,y)}}[logD(x,y;\Theta_D)] + \mathbb{E}_{y\sim p_{y(y)},z\sim p_{z(z)}}[log(1-D(G(z,y;\Theta_G),y;\Theta_D))] \tag{7}$$

Training objective V(D,G) is optimized as:

$$\Theta_G^*,\Theta_D^* = argmin_{\Theta_G}max_{\Theta_D}V(\Theta_G,\Theta_D) \tag{8}$$

Loss function for cGAN becomes:

$$\begin{aligned} &V(\{(x^{(i)},y^{(i)})\}_{i=1}^n,\Theta_G,\Theta_D) \\ &= \frac{1}{2n}[\sum_{i=1}^n log(D(x^{(i)},y^{(i)};\Theta_D)) + \sum_{i=1}^n log(1-D(G(z^{(i)},y^{(i)};\Theta_G),y^{(i)};\Theta_D))] \end{aligned} \tag{9}$$

where, we sample $z^{(i)}\sim p_z(z)$ independently for each image and n is the total number of samples.

cGAN is trained by simultaneously training generator and discriminator. First, discriminator parameters are updated using the loss function:

$$\begin{aligned} \Theta_D^{*,r+1} &= argmax_{\Theta_D}V(\{(x^{(i)},y^{(i)},z^{(i)})\}_{i=1}^n,\Theta_G^{*,r},\Theta_D) \\ &= argmax_{\Theta_D}\frac{1}{2n}[\sum_{i=1}^n log(D(x^{(i)},y^{(i)};\Theta_D)) + \sum_{i=1}^n log(1-D(G(z^{(i)},y^{(i)};\Theta_G),y^{(i)};\Theta_D))] \end{aligned} \tag{10}$$

Then, generator parameters are updated using the loss function:

$$\begin{aligned} \Theta_G^{*,r+1} &= argmin_{\Theta_G}V(\{(x^{(i)},y^{(i)},z^{(i)})\}_{i=1}^n,\Theta_G,\Theta_D^{*,r}) \\ &= argmin_{\Theta_G}\frac{1}{n}[\sum_{i=1}^n log(1-D(G(z^{(i)},y^{(i)};\Theta_G),y^{(i)};\Theta_D))] \end{aligned} \tag{11}$$

We train G and D until convergence, as shown in the pseudo-code in section 2.3.1.

2.3.1 Pseudo-Code

Algorithm 1 Stochastic gradient descent for conditional GAN

```
Initialize training images $X$, training labels $Y$ and epochs T
Construct generator G and discriminator D
Initialize generator parameters $\Theta_G$ and learning rate $\lambda^G$
Initialize discriminator parameters $\Theta_D$ and learning rate $\lambda^D$
for $t = 1, 2, \ldots, \mathbf{T}$ do
    Take batch of n images $\{x^{(1)}, x^{(2)}, x^{(3)}, ..., x^{(n)}\}$ and corresponding labels $\{y^{(1)}, y^{(2)}, y^{(3)}, ..., y^{(n)}\}$
    Sample batch of n noise samples $\{z^{(1)}, z^{(2)}, z^{(3)}, ..., z^{(n)}\}$ from Gaussian distribution $p_z(z)$

    {Train the discriminator} Update $\Theta_D$ while fixing $\Theta_G$ :
    for $t_d = 1, 2, \ldots, \mathbf{T_d}$ do
        Compute predictions through forward propagation
        Compute discriminator loss: $\mathcal{L}_d(\{x^{(i)}, y^{(i)}, z^{(i)}\}_{i=1}^n, \Theta_G, \Theta_D) =$
        $\frac{1}{2n}[\sum_{i=1}^n log(D(x^{(i)}, y^{(i)}; \Theta_D)) + \sum_{i=1}^n log(1 - D(G(z^{(i)}, y^{(i)}; \Theta_G), y^{(i)}; \Theta_D))]$
        Compute gradients through back propagation $\nabla_{\Theta_D}\mathcal{L}_d(\{x^{(i)}, y^{(i)}, z^{(i)}\}_{i=1}^n, \Theta_G, \Theta_D)$
        Update $\Theta_D$ by gradient ascent: $\Theta_D \leftarrow \Theta_D + \lambda^D \nabla_{\Theta_D}\mathcal{L}_d(\{x^{(i)}, y^{(i)}, z^{(i)}\}_{i=1}^n, \Theta_G, \Theta_D)$
    end for

    {Train the generator} Update $\Theta_G$ while fixing $\Theta_D$ :
    for $t_g = 1, 2, \ldots, \mathbf{T_g}$ do
        Generate images through forward propagation
        Compute generator loss: $\mathcal{L}_g(\{z^{(i)}, y^{(i)}\}_{i=1}^n, \Theta_G, \Theta_D) =$
        $\frac{1}{n}[\sum_{i=1}^n log(1 - D(G(z^{(i)}, y^{(i)}; \Theta_G), y^{(i)}; \Theta_D))]$
        Compute gradients through back propagation $\nabla_{\Theta_G}\mathcal{L}_g(\{z^{(i)}, y^{(i)}\}_{i=1}^n, \Theta_G, \Theta_D)$
        Update $\Theta_G$ by gradient descent: $\Theta_G \leftarrow \Theta_G - \lambda^G \nabla_{\Theta_G}\mathcal{L}_g(\{x^{(i)}, y^{(i)}\}_{i=1}^n, \Theta_G, \Theta_D$
    end for
end for
```

3 Experiment

3.1 Dataset

The dataset contains 202,599 RGB images and attribute vectors associated with each image. Each 64x64x3 image has a corresponding attribute vector that describes the condition of the image. All images are normalized to 0-1 by dividing each image by 255 and all attributes are converted to 5-dimensional one-hot labels. For example, label [1,1,0,0,0] will represent a male with black hair who is not smiling, not young and does not have an oval face. Labels are in the order: Black hair, Male, Oval Face, Smiling and Young. Some examples from the dataset are shown in figure 7.

After loading all the data using $np.load()$, it has dimensions:

- Training images: $202599 \times 64 \times 64 \times 3$
- Training labels: $202599 \times 5 \times 1$

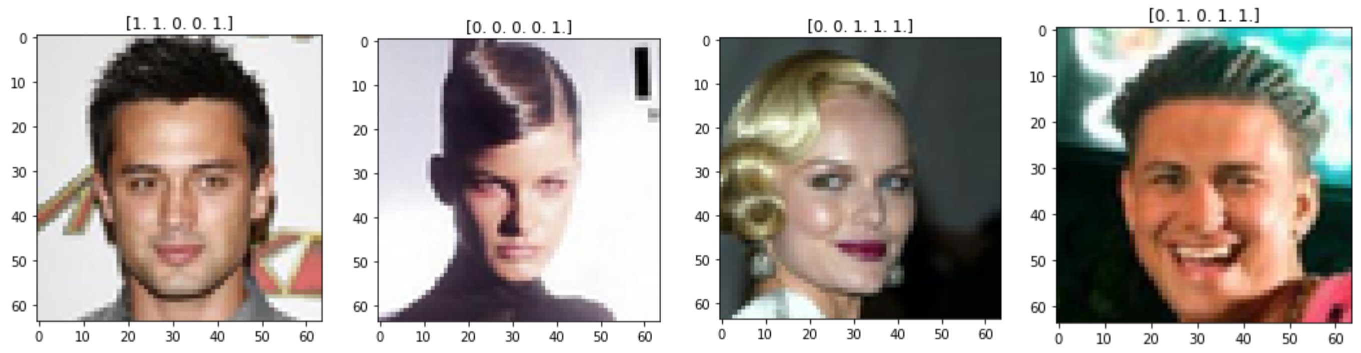


Figure 7: Training images and labels from CelebA dataset. Each image has a one-hot label describing the image in the order: Black hair, Male, Oval Face, Smiling, Young. For example, [1 1 0 0 1] shows a young male with black hair who is not smiling and does not have an oval face.

3.2 Experimental settings

Conditional GAN is implemented in TensorFlow 2.6.0, CUDA 10.2, Python 3 and trained on GPU Tesla V100-SXM2-32GB for 84.7 minutes. Implementation is inspired by [6]. The model is trained on CelebA dataset using Adam optimizer (momentum = 0.5), learning rate of 0.0001 and batch size of 128 for 20 epochs. Generator and discriminator models are trained by optimizing binary cross-entropy loss function.

Before training, all training images are normalized to [0,1] by dividing each image by 255.0 and all training labels are converted to one-hot labels of type float32. Custom class of conditional GAN is initialized under 'Model' class in keras and images are generated during training using callbacks method. To optimize the training process, multiple GPUs are used for training and evaluation using tf.device().

3.3 Model evaluation

Three evaluation metrics are used to evaluate the trained model:

- Generated images - match the given labels
- Fréchet Inception Distance (FID) - is lower than 15 i.e. 0.2
- Inception score (IS) - is higher than 2 i.e. 2.6

3.3.1 Generated images

Images are generated by loading the saved weights and predicting using trained generator. Figure 8 shows generated images according to specific labels y and randomly generated noise vector z. Generated images have pixel values in range [-1,1] because tanh activation function is applied at the output layer of generator. All generated images are normalized to range [0,1] before computing FID and IS using the formula $\hat{x}_{normalized} = (\hat{x} - min)/(max - min)$.

We can observe in Figure 8 that generator learns the attributes and adjusts the pixel values of image according to the specific label. For example, label 1 for smile will always generate images which has smiling humans. Also, generated images are visually clear and show all 5 attributes according to the given label.

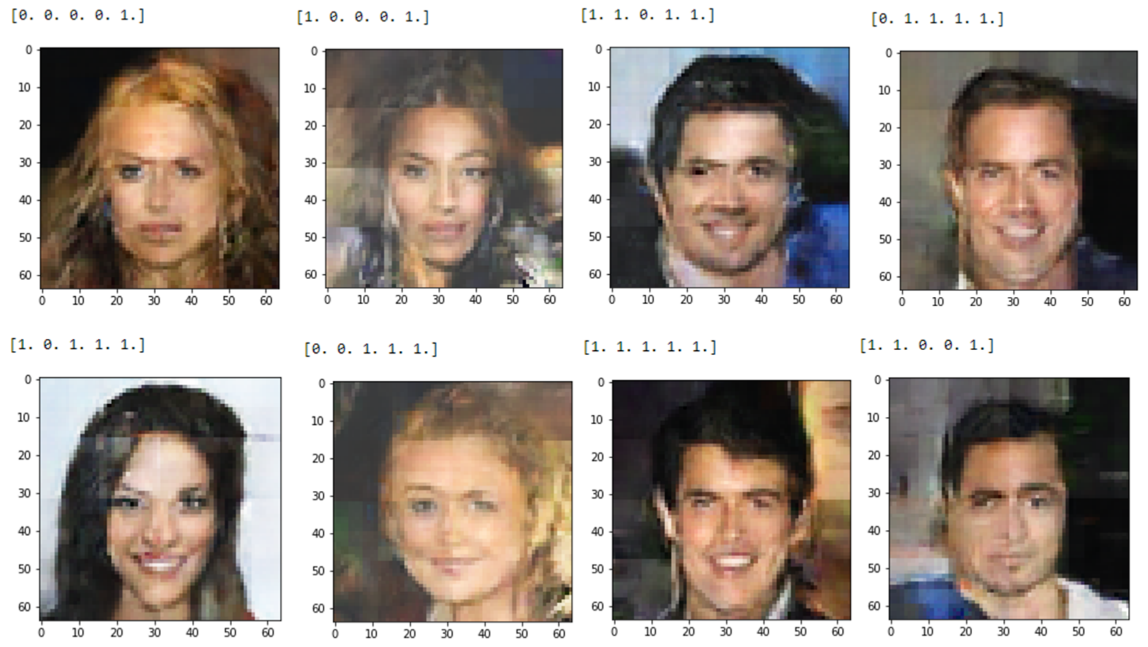


Figure 8: Generated images from conditional GAN with the corresponding labels. We can observe that each image is generated according to the label. For example, image with label [0,0,0,0,1] is a young female who is not smiling and does not have black nor oval face.

3.3.2 FID and IS score

Fréchet Inception Distance (FID) [4] is used to measure the diversity between generated and real images and is computed as follows:

$$FID = |\mu_1 - \mu_2| + Tr(\sigma_1 + \sigma_2 - 2\sqrt{\sigma_1 * \sigma_2}) \tag{12}$$

where, μ_1 is the mean of embeddings of real images, μ_2 is the mean of embeddings of generated images, σ_1 is the standard deviation of embeddings of real images and σ_2 is the standard deviation of embeddings of generated images.

Inception score (IS) [7] is used to measure the quality of generated images and is computed as follows:

$$IS(\hat{x}) = exp(\frac{1}{N}\sum_{i=1}^{N} D_{KL}(p(y|x^{(i)}||\hat{p}(y)))) \tag{13}$$

where, $\hat{x}$ is the generated image, N is the batch size, $p(y|x)$ is the conditional probability of image, $p(y)$ is the marginal probability that the generated image is real and D_{KL} is the KL-divergence of above probabilities. Note: all embeddings are created using pre-trained InceptionV3.

Metric	Score
FID	0.212
IS	2.621

Table 1: FID and IS for generated images

3.3.3 Pseudo-code for computing FID and IS

Algorithm 2 Inception Score (IS) and Fréchet Inception Distance (FID)

```
Initialize training images $X$, training labels $Y$ and batch size N
Construct generator G, discriminator D and class cGAN
Load weights for trained generator G
Take batch of n images $\{x^{(1)}, x^{(2)}, x^{(3)}, ..., x^{(n)}\}$ and corresponding labels $\{y^{(1)}, y^{(2)}, y^{(3)}, ..., y^{(n)}\}$
Sample batch of n noise samples $\{z^{(1)}, z^{(2)}, z^{(3)}, ..., z^{(n)}\}$ from Gaussian distribution $p_z(z)$
Generate n images $\{\hat{x}^{(1)}, \hat{x}^{(2)}, \hat{x}^{(3)}, ...\hat{x}^{(n)}\}$
Resize generated and real images to shape $(N, 299, 299, 3)$
Load pre-trained InceptionV3 and compute embeddings
compute FID and IS
```

3.3.4 Generated images for varying condition

We will vary the label for all 5 attributes three times and generate images using trained generator.

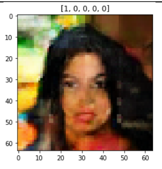
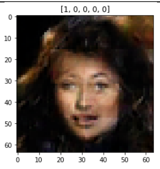
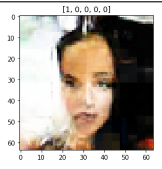
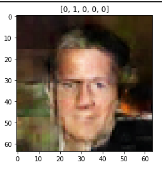
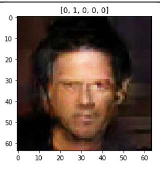
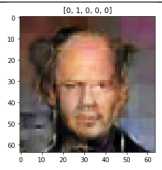
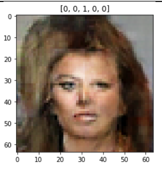
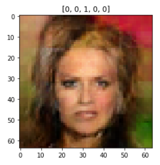
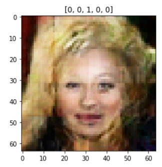
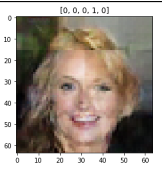
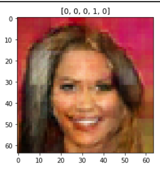
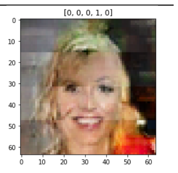
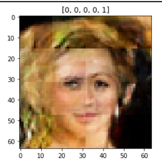
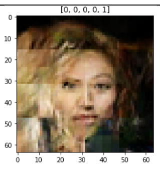
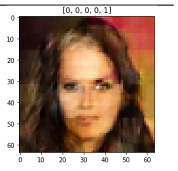
Black Hair (IS = 2.5 , FID = 0.156)			
Male (IS = 2.7 , FID = 0.194)			
Oval Face (IS = 2.8 , FID = 0.209)			
Smiling (IS = 2.8 , FID = 0.366)			
Young (IS = 2.6 , FID = 0.187)			

Table 2: Generated images by varying the condition three times for all 5 attributes

3.3.5 Loss versus epochs

Generator loss decreases and converges are around 20 epochs from loss value of 3.8 to 0.9. Discriminator loss increases in the start and then starts to decrease, converging at around 20 epochs. We observe that loss:

- is unstable when ReLU activation is used for both discriminator and generator
- is more stochastic when lower batch sizes are used i.e 9,32
- diverges when higher learning rate is used i.e. 0.002
- continues to decrease with learning rate 0.0001
- is stable when Leaky ReLU is used for discriminator and ReLU is used for generator
- is smoother at batch size of 128
- is stable when batch normalization is used after each layer in both generator and discriminator

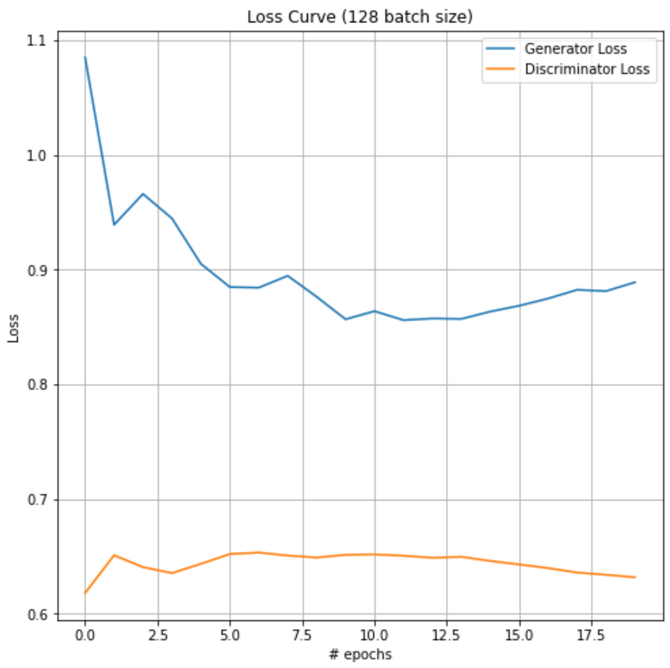


Figure 9: Loss curves for generator and discriminator for conditional GAN

3.4 Ablation study

In this section, we will discuss the effect of the following on the evaluation metrics (generated images, FID and IS):

- Training a simple GAN without any labels
- Concatenating the condition y with the noise z in generator and with images in discriminator
- Creating embedding of labels y and noise z of same dimension and then concatenating the embeddings

3.4.1 Training a simple GAN without any labels

Simple GAN is able to generate images given random noise. We can observe that images are good quality but simple GAN can not output the image based on a specific label. Simple GAN is implemented in tensorflow

2.6.0 using the implementation by [1] and trained on CelebA dataset using Tesla V100-SXM2-32GB to generate the following images:

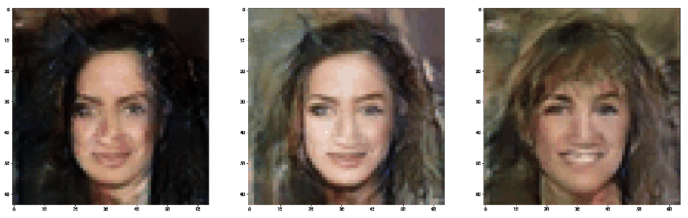


Figure 10: Generated images from a simple GAN without any labels

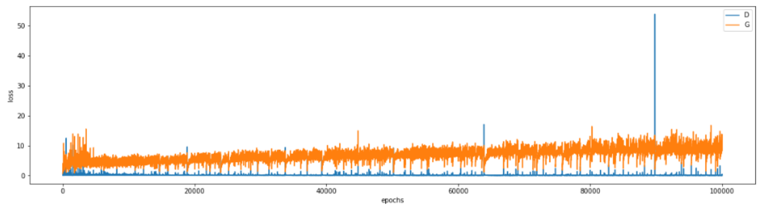


Figure 11: Loss curves from a simple GAN without any labels

3.4.2 Concatenating the condition y

The simplest way of integrating the condition into the network is to concatenate it to the first layers, i.e., to concatenate it to the z vector before passing to the generator and to the images forwarded to the discriminators. This method was used in the first paper on cGANs [5] as shown in Figure 2 in section 1.2. If we concatenate 5 dimensional label with 100 dimensional noise vector, generator may completely ignore the label, generating data with high similarity in each class. We can observe in Figure 12 that generator is not able to distinguish the labels for example, it generates black hair for images with label 0 for black hair. Loss curves for generator and discriminator are shown in Figure 14 (a).

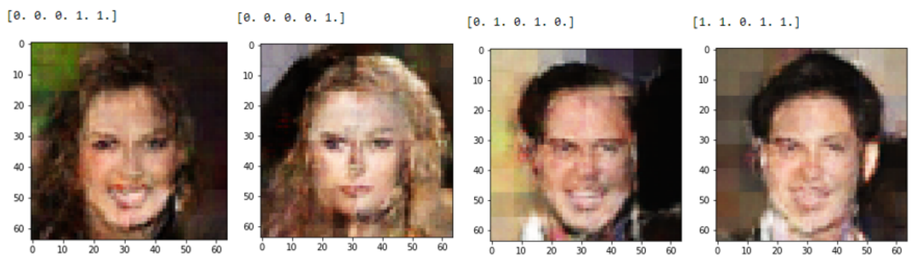


Figure 12: Generated images for conditional GAN by concatenating labels y with noise z in G and with images in D

3.4.3 Creating embedding of labels y and noise z in conditional GAN

In generator, we pass the labels y and noise z through de-convolutional layers to create embedding of same dimension and then concatenate them. In discriminator, we pass the labels y and images I through convolutional layers to create embedding of same dimension and then concatenate the embeddings. We can observe in Figure 13 that creating embedding of conditioned data allows the generator to use labels and create images according to given labels. For example, images with label 1 for 'smiling' shows smiling faces. Loss curves for generator and discriminator are shown in Figure 14 (b).

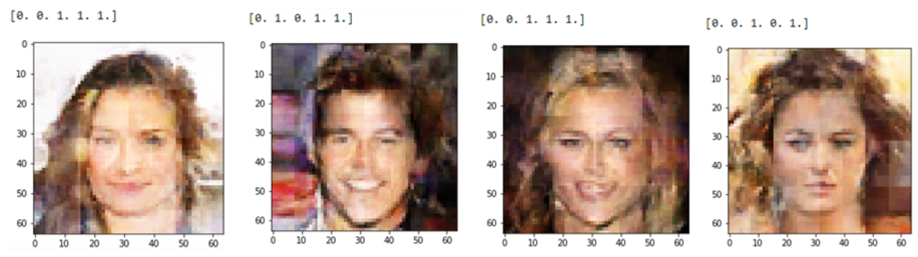
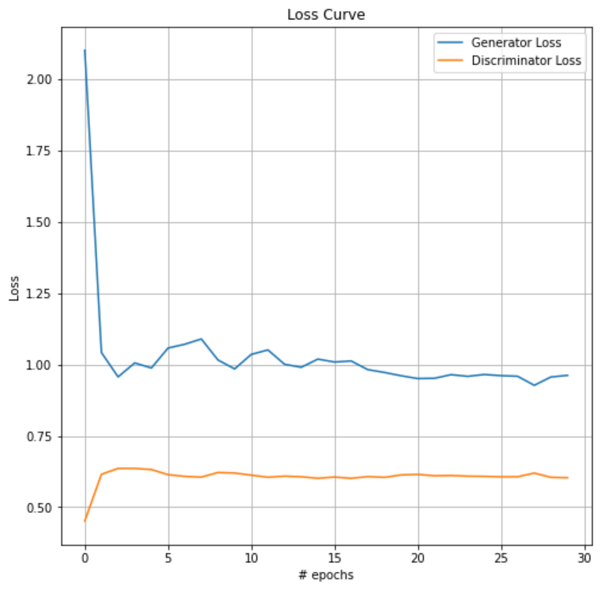
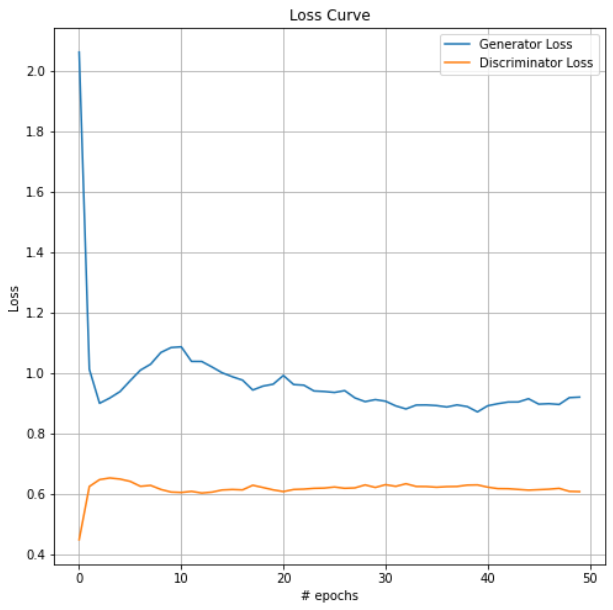


Figure 13: Generated images for conditional GAN by creating embedding of label y and noise z



(a) Concatenate condition y



(b) Create embedding of condition y

Figure 14: Loss curves for conditional GAN

3.5 Experimental analysis

We observe that generator and discriminator loss becomes unstable when hyper-parameters are changed. Effect of hyper-parameters on model performance is discussed below:

- Loss curves are smoother at batch size of 128
- Loss continues to decrease with Adam optimizer and 0.5 momentum
- Loss does not diverge at learning rate of 0.0001
- Generated images have lower FID when leaky ReLU is used for discriminator and ReLU is used for generator
- FID is lower and IS is higher when generated images are re-sized to 299x299x3
- Creating embedding of condition allows the generator to use the labels
- Loss is more stable when batch normalization is used in generator and discriminator layers
- Generated images are visually better when model is trained for 20 epochs
- Saving model weights allows faster evaluation and computation of FID and IS
- Total run time is reduced when model is trained on multiple GPUs using tf.device()

4 Conclusion

We trained a deep convolutional conditional GAN by alternatively training generator and discriminator models. We observe that generator and discriminator loss is unstable at the start but start to converge at 20 epochs. Training the model at higher batch size is computationally expensive but increase in batch size reduces the stochasticity in the loss curves. Also, using Adam optimizer with learning rate 0.0001 and 0.5 momentum gives optimum results. FID is lower than 15 and IS is higher than 2 when batch size of 128 is used along with leaky ReLU activation function for discriminator. Finally, conditional GAN is able to generate images for specific conditions. cGANs can be used in many applications for example, for data augmentation in medical fields.

References

[1] Yumi's Blog. My first gan using celeba data. 2018.

[2] Mehdi Mirza Bing Xu David Warde-Farley Sherjil Ozair Aaron Courville Yoshua Bengio Ian J. Goodfellow, Jean Pouget-Abadie. Generative adversarial networks. 2014.

[3] Qiang Ji. Gan framework. *ECSE 6850 Chapter 5 Lecture Slides*, page 14, 2022.

[4] Thomas Unterthiner Bernhard Nessler Sepp Hochreiter Martin Heusel, Hubert Ramsauer. Gans trained by a two time-scale update rule converge to a local nash equilibrium. 2017.

[5] Simon Osindero Mehdi Mirza. Conditional generative adversarial nets. 2014.

[6] Sayak Paul. Conditional gan. 2021.

[7] Rishi Sharma Shane Barratt. A note on the inception score. 2018.