**Practical Assignment**

**Objective: - Generate Human Faces with DCGAN**

Powerful techniques to generate images, audio, text, or videos that are indistinguishable from real-world data. The idea behind this project is to start with random noise and apply DCGAN to generate real-like human faces that don’t even exist.

**Dataset Link: -**

**Use any dataset of your choice**

**Task: -** Create a detailed Google Colab Notebook Result with proper documentation.

**Assignment Submission: -** Only submit the hosted app link. OR GitHub Link

**Necessary Imports**

**import** numpy **as** np

**import** tensorflow **as** tf

**import** matplotlib.pyplot **as** plt

**import** os

**from** scipy.misc **import** pilutil **as** sp

**from** glob **import** glob

**from** datetime **import** datetime

**%matplotlib** inline

## Util Functions

**Standard Scaler**

**def** standard\_scaler(img):

**return** (img**/**255.0) **\*** 2 **-** 1

**Preprocess**

**def** preprocess(batch):

**return** [standard\_scaler(sp**.**imread(b)) **for** b **in** batch]

**Leaky ReLU**

**def** leaky\_relu(x, alpha**=**0.2):

**return** tf**.**maximum(alpha**\***x, x)

## Building Neural Network layers

### Dense Layer

**class** Dense(object):

**def** \_\_init\_\_(self, name, X1, X2, apply\_batch\_norm, fun**=**tf**.**nn**.**relu):

*# Weight parameters*

self**.**W **=** tf**.**get\_variable("W\_%s" **%** name, shape**=**(X1, X2), initializer**=**tf**.**random\_normal\_initializer(stddev**=**0.02),)

self**.**b **=** tf**.**get\_variable("b\_%s" **%** name, shape**=**(X2,), initializer**=**tf**.**zeros\_initializer(),)

*# layer attributes*

self**.**fun **=** fun

self**.**name **=** name

self**.**apply\_batch\_norm **=** apply\_batch\_norm

*# params list for updating weights*

self**.**params **=** [self**.**W, self**.**b]

**def** forward(self, X, reuse, is\_training):

out **=** tf**.**matmul(X, self**.**W) **+** self**.**b

**if** self**.**apply\_batch\_norm:

out **=** tf**.**contrib**.**layers**.**batch\_norm(out, decay**=**0.9, updates\_collections**=None**, epsilon**=**1e-5, scale**=True**, is\_training**=**is\_training, reuse**=**reuse, scope**=**self**.**name,)

**return** self**.**fun(out)

### Convolutional Layer (2-Dimensional)

**class** Conv:

**def** \_\_init\_\_(self, name, feat\_in, feat\_out, apply\_batch\_norm, filters**=**5, stride**=**2, fun**=**tf**.**nn**.**relu):

*# Weight parameters*

self**.**W **=** tf**.**get\_variable("W\_%s" **%** name, shape**=**(filters, filters, feat\_in, feat\_out), initializer**=**tf**.**truncated\_normal\_initializer(stddev**=**0.02),)

self**.**b **=** tf**.**get\_variable("b\_%s" **%** name, shape**=**(feat\_out,), initializer**=**tf**.**zeros\_initializer(),)

*# layer attributes*

self**.**name **=** name

self**.**fun **=** fun

self**.**stride **=** stride

self**.**apply\_batch\_norm **=** apply\_batch\_norm

*# params list for updating weights*

self**.**params **=** [self**.**W, self**.**b]

**def** forward(self, X, reuse, is\_training):

conv\_out **=** tf**.**nn**.**conv2d(X, self**.**W, strides**=**[1, self**.**stride, self**.**stride, 1], padding**=**'SAME')

conv\_out **=** tf**.**nn**.**bias\_add(conv\_out, self**.**b)

**if** self**.**apply\_batch\_norm:

conv\_out **=** tf**.**contrib**.**layers**.**batch\_norm(conv\_out, decay**=**0.9, updates\_collections**=None**, epsilon**=**1e-5, scale**=True**, is\_training**=**is\_training, reuse**=**reuse, scope**=**self**.**name,)

**return** self**.**fun(conv\_out)

### Fractionally Strided Convolutional Layer

**class** FractionalStrideConv:

**def** \_\_init\_\_(self, name, feat\_in, feat\_out, output\_shape, apply\_batch\_norm, filters**=**5, stride**=**2, fun**=**tf**.**nn**.**relu):

*# Weight parameters*

self**.**W **=** tf**.**get\_variable("W\_%s" **%** name, shape**=**(filters, filters, feat\_out, feat\_in), initializer**=**tf**.**random\_normal\_initializer(stddev**=**0.02),)

self**.**b **=** tf**.**get\_variable("b\_%s" **%** name, shape**=**(feat\_out,), initializer**=**tf**.**zeros\_initializer(),)

*# layer attributes*

self**.**fun **=** fun

self**.**stride **=** stride

self**.**name **=** name

self**.**output\_shape **=** output\_shape

self**.**apply\_batch\_norm **=** apply\_batch\_norm

*# params list for updating weights*

self**.**params **=** [self**.**W, self**.**b]

**def** forward(self, X, reuse, is\_training):

conv\_out **=** tf**.**nn**.**conv2d\_transpose(value**=**X, filter**=**self**.**W, output\_shape**=**self**.**output\_shape, strides**=**[1, self**.**stride, self**.**stride, 1],)

conv\_out **=** tf**.**nn**.**bias\_add(conv\_out, self**.**b)

**if** self**.**apply\_batch\_norm:

conv\_out **=** tf**.**contrib**.**layers**.**batch\_norm(conv\_out, decay**=**0.9, updates\_collections**=None**, epsilon**=**1e-5, scale**=True**, is\_training**=**is\_training, reuse**=**reuse, scope**=**self**.**name,)

**return** self**.**fun(conv\_out)

## Building the GAN Model

**class** GAN:

**def** \_\_init\_\_(self, img\_size, num\_channels, disc\_size, gen\_size):

*# GAN attributes*

self**.**img\_size **=** img\_size

self**.**num\_channels **=** num\_channels

self**.**latent\_dim **=** gen\_size['z']

*# Input data*

self**.**X **=** tf**.**placeholder(tf**.**float32, shape**=**(**None**, img\_size, img\_size, num\_channels), name**=**'X')

*# Input noise*

self**.**Z **=** tf**.**placeholder(tf**.**float32, shape**=**(**None**, self**.**latent\_dim), name**=**'Z')

*# Batch size*

self**.**batch\_size **=** tf**.**placeholder(tf**.**int32, shape**=**(), name**=**'batch\_size')

*# our discriminator*

logits **=** self**.**init\_discriminator(self**.**X, disc\_size)

*# our generator*

self**.**sample\_images **=** self**.**init\_generator(self**.**Z, gen\_size)

*# get sample logits from discriminator*

**with** tf**.**variable\_scope("discriminator") **as** scope:

scope**.**reuse\_variables()

sample\_logits **=** self**.**disc\_forward(self**.**sample\_images, **True**)

*# get sample images for test from generator*

**with** tf**.**variable\_scope("generator") **as** scope:

scope**.**reuse\_variables()

self**.**test\_sample **=** self**.**gen\_forward(self**.**Z, reuse**=True**, is\_training**=False**)

*# loss functions*

*# seperate losses for discriminator fake and real operations*

self**.**d\_loss\_real **=** tf**.**nn**.**sigmoid\_cross\_entropy\_with\_logits(logits**=**logits, labels**=**tf**.**ones\_like(logits))

self**.**d\_loss\_fake **=** tf**.**nn**.**sigmoid\_cross\_entropy\_with\_logits(logits**=**sample\_logits, labels**=**tf**.**zeros\_like(sample\_logits))

*# loss function of discriminator*

self**.**d\_loss **=** tf**.**reduce\_mean(self**.**d\_loss\_real) **+** tf**.**reduce\_mean(self**.**d\_loss\_fake)

*# loss function of generator*

self**.**g\_loss **=** tf**.**reduce\_mean(tf**.**nn**.**sigmoid\_cross\_entropy\_with\_logits(logits**=**sample\_logits, labels**=**tf**.**ones\_like(sample\_logits)))

real\_predictions **=** tf**.**cast(logits **>** 0, tf**.**float32)

fake\_predictions **=** tf**.**cast(sample\_logits **<** 0, tf**.**float32)

num\_predictions **=** 2.0**\***BATCH\_SIZE

*# accuracy operation*

num\_correct **=** tf**.**reduce\_sum(real\_predictions) **+** tf**.**reduce\_sum(fake\_predictions)

self**.**d\_accuracy **=** num\_correct **/** num\_predictions

*# optimizers*

*# discriminator params for updating weights by the optimizer*

self**.**d\_params **=** [t **for** t **in** tf**.**trainable\_variables() **if** t**.**name**.**startswith('d')]

*# generator params for updating weights by the optimizer*

self**.**g\_params **=** [t **for** t **in** tf**.**trainable\_variables() **if** t**.**name**.**startswith('g')]

*# Adam optimizer for generator and discriminator, reduce losses respectively*

self**.**d\_train\_operation **=** tf**.**train**.**AdamOptimizer(LEARNING\_RATE, beta1**=**BETA1)**.**minimize(self**.**d\_loss, var\_list**=**self**.**d\_params)

self**.**g\_train\_operation **=** tf**.**train**.**AdamOptimizer(LEARNING\_RATE, beta1**=**BETA1)**.**minimize(self**.**g\_loss, var\_list**=**self**.**g\_params)

*# session and variables initialization*

self**.**init\_operation **=** tf**.**global\_variables\_initializer()

self**.**sess **=** tf**.**InteractiveSession()

self**.**sess**.**run(self**.**init\_operation)

*# model saver object*

self**.**saver **=** tf**.**compat**.**v1**.**train**.**Saver()

**def** init\_discriminator(self, X, disc\_size):

**with** tf**.**variable\_scope("discriminator") **as** scope:

*# build convolutional layers*

self**.**d\_conv\_layers **=** []

feat\_in **=** self**.**num\_channels

dim **=** self**.**img\_size

count **=** 0

**for** feat\_out, filters, stride, apply\_batch\_norm **in** disc\_size['conv\_layers']:

name **=** "d\_conv\_layer\_%s" **%** count

count **+=** 1

layer **=** Conv(name, feat\_in, feat\_out, apply\_batch\_norm, filters, stride, leaky\_relu)

self**.**d\_conv\_layers**.**append(layer)

feat\_in **=** feat\_out

print("Discriminator Dimensions:", dim)

dim **=** int(np**.**ceil(float(dim) **/** stride))

feat\_in **=** feat\_in **\*** dim **\*** dim

*# build dense layers*

self**.**d\_dense\_layers **=** []

**for** feat\_out, apply\_batch\_norm **in** disc\_size['dense\_layers']:

name **=** "d\_dense\_layer\_%s" **%** count

count **+=** 1

layer **=** Dense(name, feat\_in, feat\_out, apply\_batch\_norm, leaky\_relu)

feat\_in **=** feat\_out

self**.**d\_dense\_layers**.**append(layer)

*# output layer*

name **=** "d\_final\_dense\_layer\_%s" **%** count

self**.**d\_final\_layer **=** Dense(name, feat\_in, 1, **False**, **lambda** x: x)

*# get sample logits*

logits **=** self**.**disc\_forward(X)

*# return the logits*

**return** logits

**def** disc\_forward(self, X, reuse**=None**, is\_training**=True**):

output **=** X

**for** layer **in** self**.**d\_conv\_layers:

output **=** layer**.**forward(output, reuse, is\_training)

output **=** tf**.**contrib**.**layers**.**flatten(output)

**for** layer **in** self**.**d\_dense\_layers:

output **=** layer**.**forward(output, reuse, is\_training)

logits **=** self**.**d\_final\_layer**.**forward(output, reuse, is\_training)

**return** logits

**def** init\_generator(self, Z, gen\_size):

**with** tf**.**variable\_scope("generator") **as** scope:

*# size of data*

dims **=** [self**.**img\_size]

dim **=** self**.**img\_size

**for** \_, \_, stride, \_ **in** reversed(gen\_size['conv\_layers']):

dim **=** int(np**.**ceil(float(dim) **/** stride))

dims**.**append(dim)

*# dimensions are backwards*

dims **=** list(reversed(dims))

print("Generator Dimensions:", dims)

self**.**g\_dims **=** dims

*# build dense layers*

feat\_in **=** self**.**latent\_dim

self**.**g\_dense\_layers **=** []

count **=** 0

**for** feat\_out, apply\_batch\_norm **in** gen\_size['dense\_layers']:

name **=** "g\_dense\_layer\_%s" **%** count

count **+=** 1

layer **=** Dense(name, feat\_in, feat\_out, apply\_batch\_norm)

self**.**g\_dense\_layers**.**append(layer)

feat\_in **=** feat\_out

*# output dense layer*

feat\_out **=** gen\_size['projection'] **\*** dims[0] **\*** dims[0]

name **=** "g\_dense\_layer\_%s" **%** count

layer **=** Dense(name, feat\_in, feat\_out, **not** gen\_size['bn\_after\_project'])

self**.**g\_dense\_layers**.**append(layer)

*# fractionally strided convolutional layer*

feat\_in **=** gen\_size['projection']

self**.**g\_conv\_layers **=** []

*# output activation either tanh or sigmoid*

num\_relus **=** len(gen\_size['conv\_layers']) **-** 1

activation\_functions **=** [tf**.**nn**.**relu]**\***num\_relus **+** [gen\_size['output\_activation']]

*# build "deconvolutional" layer*

**for** i **in** range(len(gen\_size['conv\_layers'])):

name **=** "g\_fs\_conv\_layer\_%s" **%** i

feat\_out, filters, stride, apply\_batch\_norm **=** gen\_size['conv\_layers'][i]

fun **=** activation\_functions[i]

output\_shape **=** [self**.**batch\_size, dims[i**+**1], dims[i**+**1], feat\_out]

print("Input Features:", feat\_in, "Output Features:", feat\_out, "Output Shape:", output\_shape)

layer **=** FractionalStrideConv(name, feat\_in, feat\_out, output\_shape, apply\_batch\_norm, filters, stride, fun)

self**.**g\_conv\_layers**.**append(layer)

feat\_in **=** feat\_out

*# output*

self**.**gen\_size **=** gen\_size

**return** self**.**gen\_forward(Z)

**def** gen\_forward(self, Z, reuse**=None**, is\_training**=True**):

*# output from dense*

output **=** Z

**for** layer **in** self**.**g\_dense\_layers:

output **=** layer**.**forward(output, reuse, is\_training)

*# project and reshape*

output **=** tf**.**reshape(output, [**-**1, self**.**g\_dims[0], self**.**g\_dims[0], self**.**gen\_size['projection']],)

*# apply batch normalization*

**if** self**.**gen\_size['bn\_after\_project']:

output **=** tf**.**contrib**.**layers**.**batch\_norm(output, decay**=**0.9, updates\_collections**=None**, epsilon**=**1e-5, scale**=True**, is\_training**=**is\_training, reuse**=**reuse, scope**=**'bn\_after\_project')

*# output via fractionally strided convolutional layers*

**for** layer **in** self**.**g\_conv\_layers:

output **=** layer**.**forward(output, reuse, is\_training)

**return** output

**def** fit(self, X):

d\_losses **=** []

g\_losses **=** []

d\_accs **=** []

offset **=** 0

N **=** len(X)

num\_batches **=** N **//** BATCH\_SIZE

print("Total batches per epoch is {}\n"**.**format(num\_batches))

total\_iters **=** 0

**for** i **in** range(EPOCHS):

print("Epoch", i**+**1)

np**.**random**.**shuffle(X)

**for** offset **in** range(num\_batches):

batch **=** preprocess(X[offset**\***BATCH\_SIZE:(offset**+**1)**\***BATCH\_SIZE])

Z **=** np**.**random**.**uniform(**-**1, 1, size**=**(BATCH\_SIZE, self**.**latent\_dim))

*# train the discriminator*

\_, d\_loss, d\_acc **=** self**.**sess**.**run((self**.**d\_train\_operation, self**.**d\_loss, self**.**d\_accuracy), feed\_dict**=**{self**.**X: batch, self**.**Z: Z, self**.**batch\_size: BATCH\_SIZE},)

d\_losses**.**append(d\_loss)

*# train the generator*

\_, g\_loss1 **=** self**.**sess**.**run((self**.**g\_train\_operation, self**.**g\_loss), feed\_dict**=**{self**.**Z: Z, self**.**batch\_size: BATCH\_SIZE},)

*# do it again*

\_, g\_loss2 **=** self**.**sess**.**run((self**.**g\_train\_operation, self**.**g\_loss), feed\_dict**=**{self**.**Z: Z, self**.**batch\_size: BATCH\_SIZE},)

*# store the loss*

g\_losses**.**append((g\_loss1 **+** g\_loss2)**/**2)

*# store the accuracy*

d\_accs**.**append(d\_acc)

*# print("Discriminator Accuracy: %.2f | Discriminator Loss: %.2f | Generator Loss: %.2f" % (d\_acc, d\_loss, g\_losses[offset]))*

*# save samples periodically*

total\_iters **+=** 1

*# save trained model*

self**.**saver**.**save(self**.**sess, "models\\GAN\_face")

**if** total\_iters **%** SAVE\_PERIOD **==** 0:

print("Saving sample {}"**.**format(total\_iters))

**if** **not** os**.**path**.**exists('new\_samples'):

os**.**mkdir('new\_samples')

samples **=** self**.**sample(64)

d **=** self**.**img\_size

**if** samples**.**shape[**-**1] **==** 1:

samples **=** samples**.**reshape(64, d, d)

flat\_image **=** np**.**empty((8**\***d, 8**\***d))

k **=** 0

**for** i **in** range(8):

**for** j **in** range(8):

flat\_image[i**\***d:(i**+**1)**\***d, j**\***d:(j**+**1)**\***d] **=** samples[k]**.**reshape(d, d)

k **+=** 1

**else**:

flat\_image **=** np**.**empty((8**\***d, 8**\***d, 3))

k **=** 0

**for** i **in** range(8):

**for** j **in** range(8):

flat\_image[i**\***d:(i**+**1)**\***d, j**\***d:(j**+**1)**\***d] **=** samples[k]

k **+=** 1

sp**.**imsave('new\_samples\\sample%d.png' **%** total\_iters, flat\_image,)

print("Discriminator Accuracy: %.2f | Discriminator Loss: %.2f | Generator Loss: %.2f" **%** (d\_accs[offset], d\_losses[offset], g\_losses[offset]))

*# plot the losses and save them*

plt**.**clf()

plt**.**plot(g\_losses, label**=**'Generator Loss')

plt**.**plot(d\_losses, label**=**'Discriminator Loss')

plt**.**title('GAN Loss')

plt**.**legend()

plt**.**savefig('loss\_metrics.png')

**def** sample(self, n):

*# generate a sample from noise*

Z **=** np**.**random**.**uniform(**-**1, 1, size**=**(n, self**.**latent\_dim))

samples **=** self**.**sess**.**run(self**.**test\_sample, feed\_dict**=**{self**.**Z: Z, self**.**batch\_size: n})

**return** samples

**def** save\_weights(self, path):

*# save model weights*

self**.**saver**.**save(self**.**sess, path)

print("Saved successfully")

## Load the Dataset

X **=** glob("faces\\\*.jpg")

## Training Configuration

dimensions **=** 64

channels **=** 3

LEARNING\_RATE **=** 0.0002

BETA1 **=** 0.5

BATCH\_SIZE **=** 64

EPOCHS **=** 10

SAVE\_PERIOD **=** 50

disc\_sizes **=** {

'conv\_layers': [

(64, 5, 2, **False**),

(128, 5, 2, **True**),

(256, 5, 2, **True**),

(512, 5, 2, **True**)

],

'dense\_layers': [],

}

gen\_sizes **=** {

'z': 100,

'projection': 512,

'bn\_after\_project': **True**,

'conv\_layers': [

(256, 5, 2, **True**),

(128, 5, 2, **True**),

(64, 5, 2, **True**),

(channels, 5, 2, **False**)

],

'dense\_layers': [],

'output\_activation': tf**.**tanh,

}

## Train the GAN

gan\_model **=** GAN(dimensions, channels, disc\_sizes, gen\_sizes)

gan\_model**.**fit(X)

## Save the GAN model

gan\_model**.**save\_weights("models\\face\_GAN")

## Evaluation

gi **=** sp**.**imread('new\_samples\\sample100.png')

plt**.**imshow(gi)

gi **=** sp**.**imread('new\_samples\\sample500.png')

plt**.**imshow(gi)

gi **=** sp**.**imread('new\_samples\\sample1000.png')

plt**.**imshow(gi)

gi **=** sp**.**imread('new\_samples\\sample2000.png')

plt**.**imshow(gi)

gi **=** sp**.**imread('new\_samples\\sample5000.png')

plt**.**imshow(gi)

gi **=** sp**.**imread('new\_samples\\sample15000.png')

plt**.**imshow(gi)

gi **=** sp**.**imread('new\_samples\\sample25000.png')

plt**.**imshow(gi)

gi **=** sp**.**imread('new\_samples\\sample31600.png')

plt**.**imshow(gi)