- Reference

```
import numpy as np
import PIL
import os
from glob import glob
import matplotlib.pyplot as plt
import seaborn as sns
import tensorflow as tf
from tensorflow.keras import layers
import cv2
import random
from keras.utils.vis_utils import plot_model

MAIN_PATH  =  "../input/celeba-dataset/img_align_celeba/img_align_celeba"


image_paths = glob(MAIN_PATH+"/*")
```

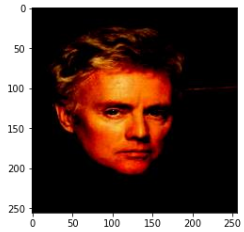
```
len(image_paths)
```

```
202599
```

```
def  readImage(path,image_size=(256,256)):
    img = np.asarray(PIL.Image.open(path).resize(image_size))
    img = ((img - 127.5) / 127.5).astype("float32")
    return img
```

```
test_img = readImage("../input/celeba-dataset/img_align_celeba/img_align_celeba/000030.jpg")
print(test_img.shape)

plt.imshow(test_img)
plt.show()
```

```
(256, 256, 3)
```



- I said we'll read images as batches so now we'll define a generator that yield one batch data.

```
BATCH_SIZE = 128
STEPS_PER_EPOCH = 500
print("Steps per epochh are",STEPS_PER_EPOCH)
def dataGenerator(batch_size):
    while True:
        paths = random.choices(image_paths,k=batch_size)
        batch = []
        for p in paths:
            batch.append(readImage(p))

        yield np.asarray(batch)

dataGen = dataGenerator(BATCH_SIZE)
print(next(dataGen).shape)
```

```
Steps per epochh are 500
(128, 256, 256, 3)
```

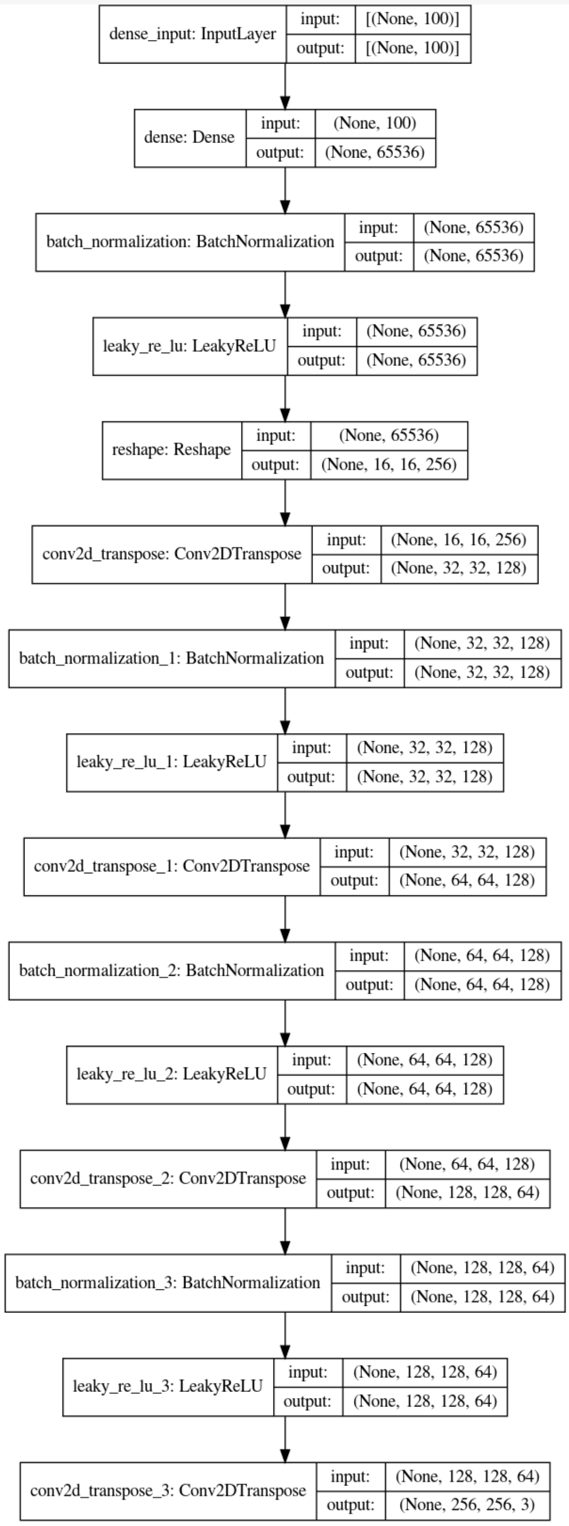
```
WEIGHT_INIT = tf.keras.initializers.RandomNormal(mean=0.0,stddev=0.2)
def make_generator():
    model = tf.keras.Sequential()

    model.add(layers.Dense(16*16*256,use_bias=False,input_shape=(100,)))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())
    model.add(layers.Reshape((16,16,256)))

    assert model.output_shape == (None,16,16,256)

    model.add(layers.Conv2DTranspose(128,(5,5),strides=(2,2),use_bias=False,padding="same",kernel_initializer=WEIGHT_INIT))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    assert model.output_shape == (None,32,32,128)

    model.add(layers.Conv2DTranspose(128,(5,5),strides=(2,2),use_bias=False,padding="same",kernel_initializer=WEIGHT_INIT))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    assert model.output_shape == (None,64,64,128)

    model.add(layers.Conv2DTranspose(64,(5,5),strides=(2,2),use_bias=False,padding="same",kernel_initializer=WEIGHT_INIT))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    assert model.output_shape == (None,128,128,64)

    model.add(layers.Conv2DTranspose(3,(5,5),strides=(2,2),use_bias=False,padding="same",kernel_initializer=WEIGHT_INIT,
                                     activation="tanh"
                                    ))
    assert model.output_shape == (None,256,256,3)
    return model
```

```
generator = make_generator()
```

```
generator.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 65536)             6553600
_________________________________________________________________
batch_normalization (BatchNo (None, 65536)             262144
leaky_re_lu (LeakyReLU)      (None, 65536)             0
_________________________________________________________________
reshape (Reshape)            (None, 16, 16, 256)       0
_________________________________________________________________
conv2d_transpose (Conv2DTran (None, 32, 32, 128)       819200
_________________________________________________________________
batch_normalization_1 (Batch (None, 32, 32, 128)       512
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 32, 32, 128)       0
_________________________________________________________________
conv2d_transpose_1 (Conv2DTr (None, 64, 64, 128)       409600
_________________________________________________________________
batch_normalization_2 (Batch (None, 64, 64, 128)       512
_________________________________________________________________
leaky_re_lu_2 (LeakyReLU)    (None, 64, 64, 128)       0
_________________________________________________________________
conv2d_transpose_2 (Conv2DTr (None, 128, 128, 64)      204800
_________________________________________________________________
batch_normalization_3 (Batch (None, 128, 128, 64)      256
_________________________________________________________________
leaky_re_lu_3 (LeakyReLU)    (None, 128, 128, 64)      0
_________________________________________________________________
conv2d_transpose_3 (Conv2DTr (None, 256, 256, 3)       4800
=================================================================
Total params: 8,255,424
Trainable params: 8,123,712
Non-trainable params: 131,712
_________________________________________________________________
```

```
plot_model(generator,show_shapes = True, show_layer_names = True, to_file='Generator_Model.png')
```


dense_input: InputLayer
input: [(None, 100)]
output: [(None, 100)]
dense: Dense
input: (None, 100)
output: (None, 65536)
batch_normalization: BatchNormalization
input: (None, 65536)
output: (None, 65536)
leaky_re_lu: LeakyReLU
input: (None, 65536)
output: (None, 65536)
reshape: Reshape
input: (None, 65536)
output: (None, 16, 16, 256)
conv2d_transpose: Conv2DTranspose
input: (None, 16, 16, 256)
output: (None, 32, 32, 128)
batch_normalization_1: BatchNormalization
input: (None, 32, 32, 128)
output: (None, 32, 32, 128)
leaky_re_lu_1: LeakyReLU
input: (None, 32, 32, 128)
output: (None, 32, 32, 128)
conv2d_transpose_1: Conv2DTranspose
input: (None, 32, 32, 128)
output: (None, 64, 64, 128)
batch_normalization_2: BatchNormalization
input: (None, 64, 64, 128)
output: (None, 64, 64, 128)
leaky_re_lu_2: LeakyReLU
input: (None, 64, 64, 128)
output: (None, 64, 64, 128)
conv2d_transpose_2: Conv2DTranspose
input: (None, 64, 64, 128)
output: (None, 128, 128, 64)
batch_normalization_3: BatchNormalization
input: (None, 128, 128, 64)
output: (None, 128, 128, 64)
leaky_re_lu_3: LeakyReLU
input: (None, 128, 128, 64)
output: (None, 128, 128, 64)
conv2d_transpose_3: Conv2DTranspose
input: (None, 128, 128, 64)
output: (None, 256, 256, 3)

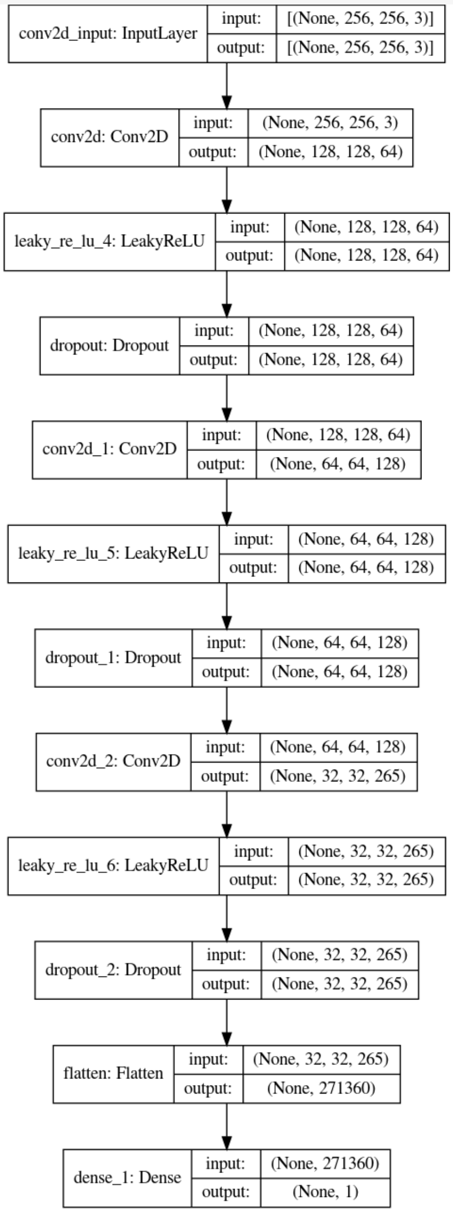
```
cross_entropy  =  tf.keras.losses.BinaryCrossentropy(from_logits=True)

def generator_loss(fake_output):
    return cross_entropy(tf.ones_like(fake_output),fake_output)

gen_optimizer = tf.keras.optimizers.Adam(lr=1e-4)
```

```
def make_discriminator():
    model = tf.keras.Sequential()

    model.add(layers.Conv2D(64,(5,5),strides=(2,2),padding="same",input_shape=(256,256,3)))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))

    model.add(layers.Conv2D(128,(5,5),strides=(2,2),padding="same"))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))

    model.add(layers.Conv2D(265,(5,5),strides=(2,2),padding="same"))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))

    model.add(layers.Flatten())
    model.add(layers.Dense(1))

    return model
```

```
discriminator = make_discriminator()

discriminator.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 128, 128, 64)      4864
_________________________________________________________________
leaky_re_lu_4 (LeakyReLU)    (None, 128, 128, 64)      0
_________________________________________________________________
dropout (Dropout)            (None, 128, 128, 64)      0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 64, 64, 128)       204928
_________________________________________________________________
leaky_re_lu_5 (LeakyReLU)    (None, 64, 64, 128)       0
_________________________________________________________________
dropout_1 (Dropout)          (None, 64, 64, 128)       0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 32, 32, 265)       848265
_________________________________________________________________
leaky_re_lu_6 (LeakyReLU)    (None, 32, 32, 265)       0
_________________________________________________________________
dropout_2 (Dropout)          (None, 32, 32, 265)       0
_________________________________________________________________
flatten (Flatten)            (None, 271360)            0
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 271361
=================================================================
Total params: 1,329,418
Trainable params: 1,329,418
Non-trainable params: 0
_________________________________________________________________
```

```
plot_model(discriminator,show_shapes = True, show_layer_names = True, to_file='Generator_Model.png')
```



```
def   discriminator_loss(real_images,fake_images):
    real_loss = cross_entropy(tf.ones_like(real_images),real_images)
    fake_loss = cross_entropy(tf.zeros_like(fake_images),fake_images)
    total_loss = real_loss + fake_loss
    return total_loss
```

```
discriminator_optimizer = tf.keras.optimizers.Adam(lr=1e-4)

EPOCHS = 5
NOISE_DIM = 100

@tf.function
def train_step(images):
    noise = tf.random.normal([BATCH_SIZE,NOISE_DIM])

    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
        generated_images = generator(noise,training=True)

        real_output = discriminator(images,training=True)
        fake_output = discriminator(generated_images,training=True)
        L_G = tf.reduce_mean(tf.abs(real_output - fake_output))
        L_D = tf.reduce_mean(tf.abs(real_output - fake_output)) - k * tf.reduce_mean(tf.abs(real_output - fake_output))

        gen_loss = generator_loss(fake_output)
        disc_loss = discriminator_loss(real_output,fake_output)


        gradients_of_generator = gen_tape.gradient(L_G, generator.trainable_variables)
        gradients_of_discriminator = disc_tape.gradient(L_D, discriminator.trainable_variables)

        gen_optimizer.apply_gradients(zip(gradients_of_generator, generator.trainable_variables))
        discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator, discriminator.trainable_variables))
```

```
import time
import sys
def train(epochs):
    for epoch in range(epochs):
        start = time.time()
        for step in range(STEPS_PER_EPOCH):
            train_step(next(dataGen))

            sys.stdout.write(f"\rSTEP: {step}/{STEPS_PER_EPOCH}")
            sys.stdout.flush()

        finish_time = round(time.time() - start,2)
        print(f"Epoch {epoch}/{epochs} Process Time : {finish_time}")
        print("-"*15)
```

```
train(EPOCHS)
```

```
STEP: 499/500Epoch 0/5 Process Time : 640.75
---------------
STEP: 499/500Epoch 1/5 Process Time : 501.48
---------------
STEP: 499/500Epoch 2/5 Process Time : 485.45
---------------
STEP: 499/500Epoch 3/5 Process Time : 444.74
---------------
STEP: 499/500Epoch 4/5 Process Time : 423.32
---------------
```

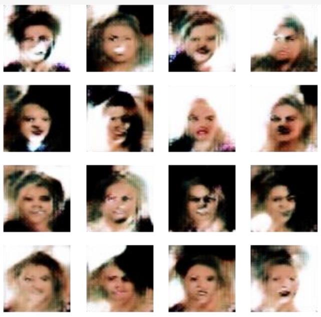
```
train(3)
```

```
STEP: 499/500Epoch 0/3 Process Time : 417.55
---------------
STEP: 499/500Epoch 1/3 Process Time : 415.64
---------------
STEP: 499/500Epoch 2/3 Process Time : 414.8
---------------
```

```
train(2)
```

```
STEP: 499/500Epoch 0/2 Process Time : 414.82
---------------
STEP: 499/500Epoch 1/2 Process Time : 414.91
---------------
```

```
noise = tf.random.normal([16,100])
generated_images = np.asarray(generator(noise,training=False))

fig = plt.figure(figsize=(6,6))
for i in range(16):
    plt.subplot(4,4,i+1)
    plt.imshow((generated_images[i,:,:,:]*127.5+127.5).astype("int"))
    plt.axis("off")

plt.show()
```



Compared to the results presented in the BEGAN paper, the images generated by my model are clearly lacking in quality andcoherence. While the BEGAN model was able to produce high-resolution (up to 128x128 pixels), diverse, and natural-lookingface images, my model is struggling to generate anything beyond blurred and disoriented faces.