GAN to generate faces.

Diana Laura Reyes Youshimatz

UDLAP ID:173391

https://github.com/rydianary/MLProjects

Introduction: What is GAN?

Generative Adversarial networks also known as GAN's create data instances based on a set of data used for training. This algorithmic architecture uses two neural networks that are one against the other, to create and determine if the results are good enough. The first one is the Generator, that crates the new data, and the other one is the Discriminator, that check whether the data created could be authentic.

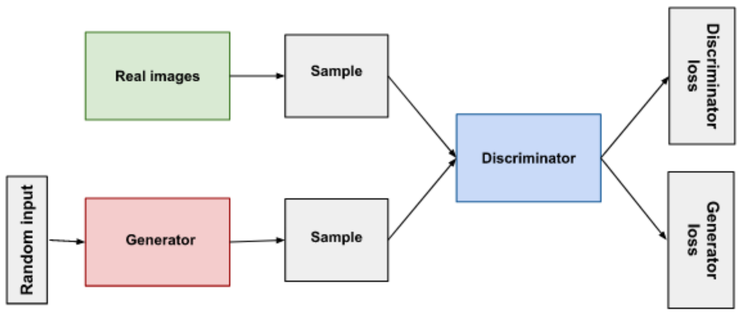
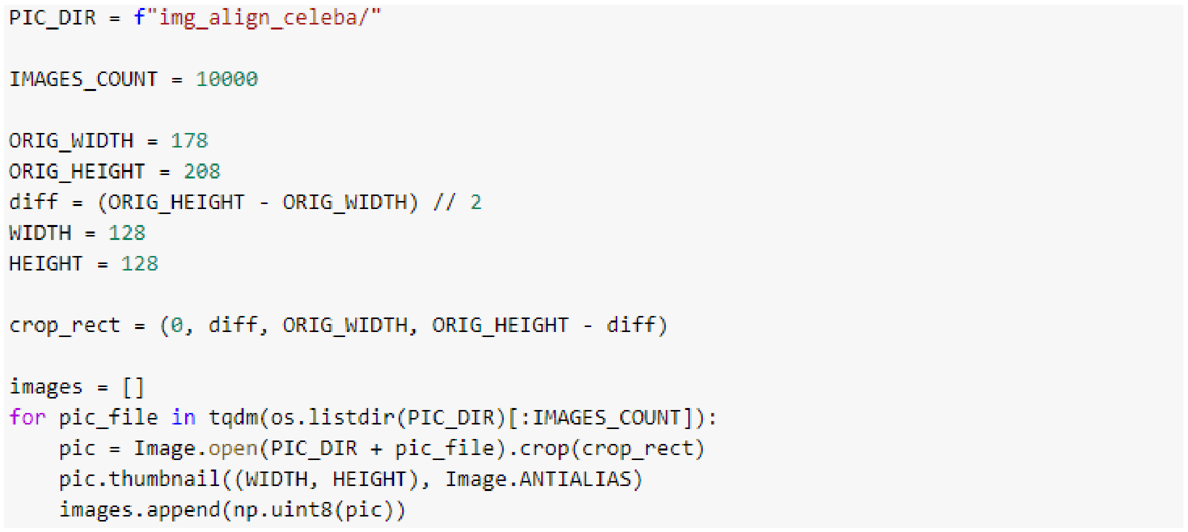


Figure 1. GAN's system diagram

Here is a diagram of how the system of a GAN works. Given the previously mentioned neural network the generator output is connected to the discriminator input that updates its weights using backpropagation. On this report we will learn about how to generate faces using GAN's.

Training data preparation

After syncing our drive to our notebook, unzipping our dataset, and importing all the libraries that we will use (these simple steps can be followed in the ipynb file), we can start standardizing our images for a proper training.



```python
PIC_DIR = f"img_align_celeba/"

IMAGES_COUNT = 10000

ORIG_WIDTH = 178
ORIG_HEIGHT = 208
diff = (ORIG_HEIGHT - ORIG_WIDTH) // 2
WIDTH = 128
HEIGHT = 128

crop_rect = (0, diff, ORIG_WIDTH, ORIG_HEIGHT - diff)

images = []
for pic_file in tqdm(os.listdir(PIC_DIR)[:IMAGES_COUNT]):
    pic = Image.open(PIC_DIR + pic_file).crop(crop_rect)
    pic.thumbnail((WIDTH, HEIGHT), Image.ANTIALIAS)
    images.append(np.uint8(pic))
```

Figure 2. Image resizing code

Most of our images are little rectangles, but for the sake of simplicity we crop our images, so there is no difference between width and height. We save all these new images and crop them as thumbnails, we use anti-aliasing for a smoother look and pixel blending. At this point we cropped 10,000 images into equal squares, of length and width of 128 and 3 channels (RGB)

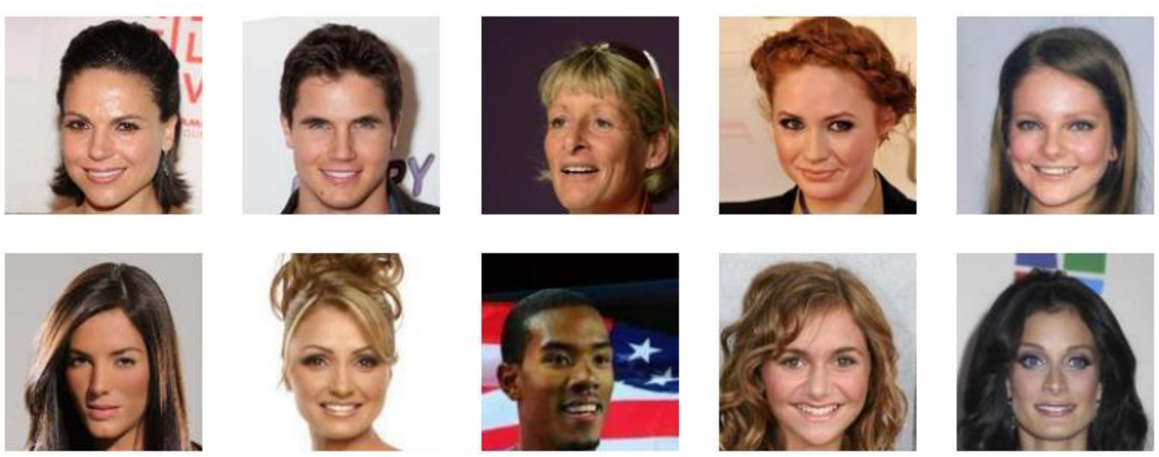


Figure 3. Display of resized images

Mathematics of neural networks.

Training GAN's serve for two main objectives. The discriminator intends to maximize the probability of assigning the correct label to both training faces and faces generated by the generator. The generator tries to minimize the probability that the discriminator can tell that what it generates can be noticed as fake, so with the proper training, the generator becomes better at creating fake faces.

To achieve these purposes we use minimax, it is a decision rule used to minimize the worst-case potential loss and optimize the function V(D,G).

$$\min_{G} \max_{D} V(D, G) = \mathbb{E}_{\boldsymbol{x} \sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z} \sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))].$$

Equation 1. Min Max function

The generator is defined by G(z), which transforms noise z we input into the fake images.

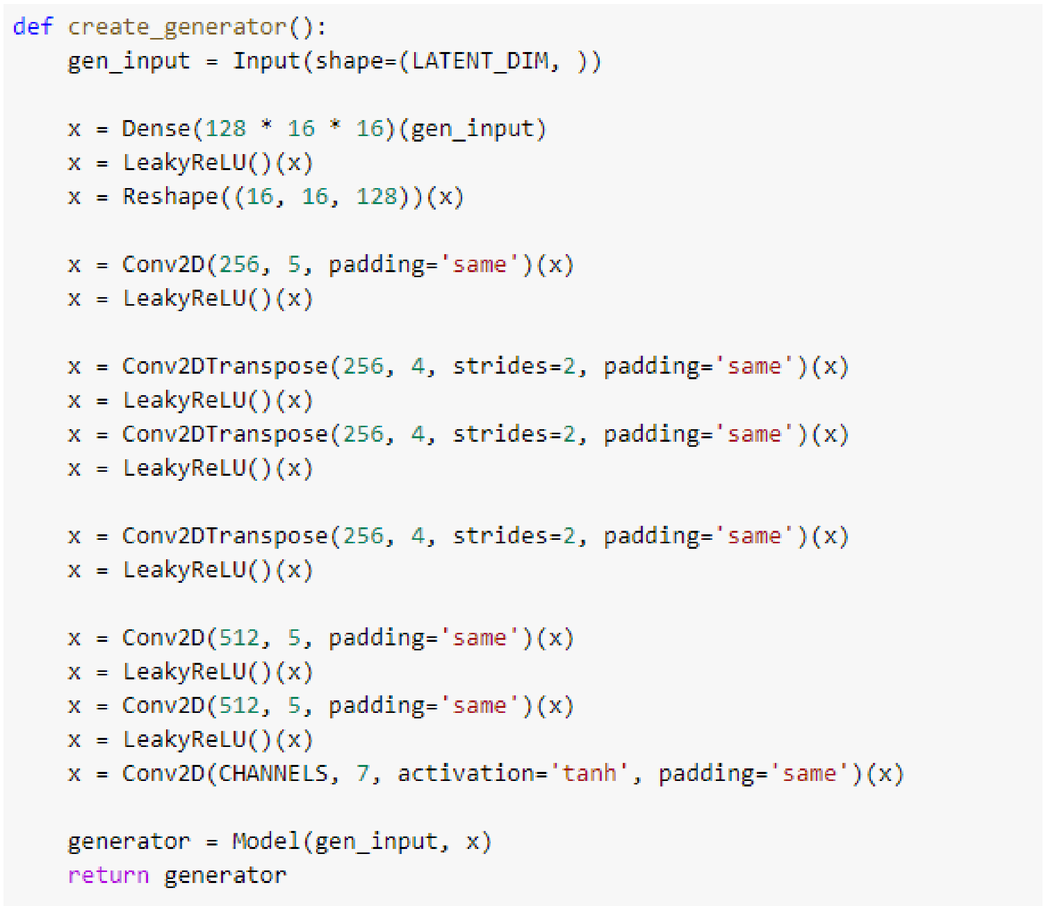
The discriminator is defined by D(x), which calculates the probability that the input given came from the training dataset.

We certainly want the predictions on the dataset by the discriminator to be as close to 1 as possible, and on the generator to be as close to 0 as possible. To achieve this, we use the log-likelihood of D(x) and 1-D(z).

Generator Creation

We take our input, the variable gen_input, and feed it into our first convolutional layer. Each convolutional layer performs a convolution and then performs batch normalization and leaky ReLu. This network has 8 convolutional layers and finally return the thanh activation function (decides whether a neuron should be activated or not). We briefly define the functions and classes used in this section.

- Dense() layer is the regular deeply connected neural network layer. It is most common and frequently used layer. Dense layer does the below operation on the input and return the output.
  output = activation(dot(input, kernel) + bias)
  - input represent the input data.
  - kernel represent the weight data.
  - dot represent the dot product of all input and its corresponding weights.
  - bias represent a biased value used in machine learning to optimize the model.
  - activation represent the activation function.
- LeakyReLu() is a type of activation function that is linear in the positive dimension and has a small slope for negative values. This activation function es very commonly used for training on GAN's
- Conv2D() convolutional layer applies sliding convolutional filters to a 2D input. The layer combines the input by moving the filters along the input vertically and horizontally, computing the dot product of the weights and the input, and then adding a bias term.



```
def create_generator():
    gen_input = Input(shape=(LATENT_DIM, ))

    x = Dense(128 * 16 * 16)(gen_input)
    x = LeakyReLU()(x)
    x = Reshape((16, 16, 128))(x)

    x = Conv2D(256, 5, padding='same')(x)
    x = LeakyReLU()(x)

    x = Conv2DTranspose(256, 4, strides=2, padding='same')(x)
    x = LeakyReLU()(x)
    x = Conv2DTranspose(256, 4, strides=2, padding='same')(x)
    x = LeakyReLU()(x)

    x = Conv2DTranspose(256, 4, strides=2, padding='same')(x)
    x = LeakyReLU()(x)

    x = Conv2D(512, 5, padding='same')(x)
    x = LeakyReLU()(x)
    x = Conv2D(512, 5, padding='same')(x)
    x = LeakyReLU()(x)
    x = Conv2D(CHANNELS, 7, activation='tanh', padding='same')(x)

    generator = Model(gen_input, x)
    return generator
```

Figure 4. Generator Creation Code

Discriminator creation.

Just like the generator, we have convolutional layers, where we perform convolution, batch normalization Leaky ReLu, and return the sigmoid activation function. In this case the additional functions and classes we used are:

Flatten() returns a copy of a multidimensional input into one dimension

Dropout() randomly sets input units to 0 with a frequency of rate 0.4 in this case at each step during training time, which helps prevent overfitting.

Model() is a class that models layers into an object with training/inference features.

RMS()prop is an optimizer that implements the Root Mean Squared Propagation algorithm, which is a type of gradient descent algorithm. The lr argument specifies the learning rate, clipvalue clips the gradients to prevent exploding gradients, and decay reduces the learning rate over time.

Discriminator's main purpose is to detect between the faces given are either real or fake so we can state its loss function as Binary Cross Entropy, which is commonly used for binary classification problems. What the generator does is a density estimation, from the noise to real data, and feed it to Discriminator to mislead it.

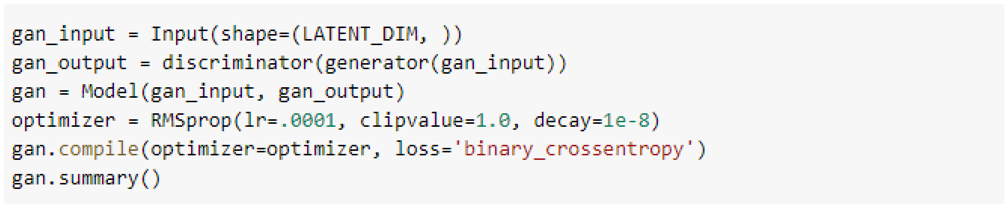
```
def create_discriminator():
    disc_input = Input(shape=(HEIGHT, WIDTH, CHANNELS))

    x = Conv2D(256, 3)(disc_input)
    x = LeakyReLU()(x)

    x = Conv2D(256, 4, strides=2)(x)
    x = LeakyReLU()(x)

    x = Conv2D(256, 4, strides=2)(x)
    x = LeakyReLU()(x)

    x = Conv2D(256, 4, strides=2)(x)
    x = LeakyReLU()(x)

    x = Conv2D(256, 4, strides=2)(x)
    x = LeakyReLU()(x)

    x = Flatten()(x)
    x = Dropout(0.4)(x)

    x = Dense(1, activation='sigmoid')(x)
    discriminator = Model(disc_input, x)

    optimizer = RMSprop(
        lr=.0001,
        clipvalue=1.0,
        decay=1e-8
    )
    discriminator.compile(
        optimizer=optimizer,
        loss='binary_crossentropy'
    )

    return discriminator
```

Defining the GAN model

Here we combine the generator and the discriminator we have just created

```
generator = create_generator()
generator.summary()
discriminator = create_discriminator()
discriminator.trainable = False
discriminator.summary()
```

Figure 5. Generator and discriminator declaration



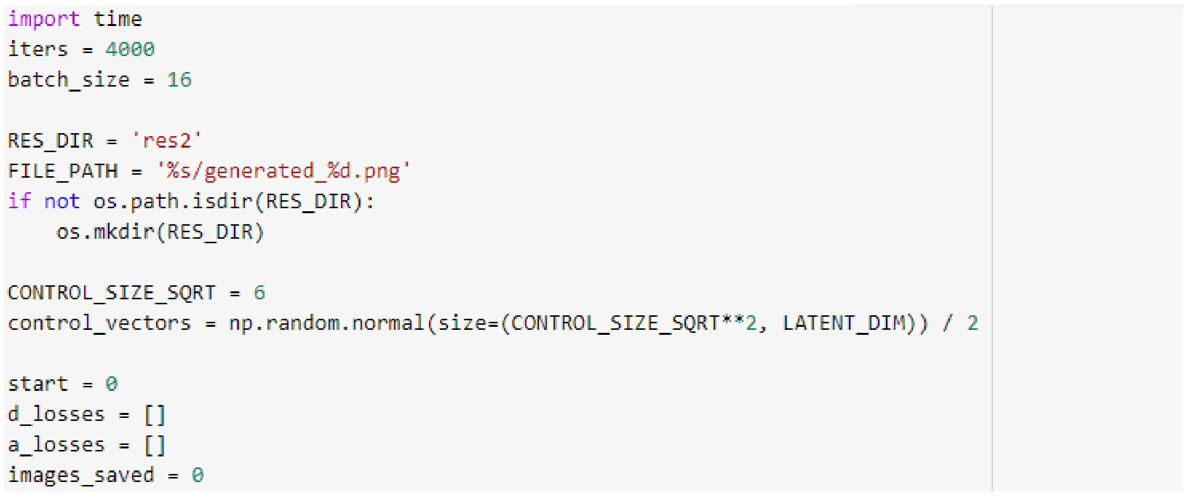
```
gan_input = Input(shape=(LATENT_DIM, ))
gan_output = discriminator(generator(gan_input))
gan = Model(gan_input, gan_output)
optimizer = RMSprop(lr=.0001, clipvalue=1.0, decay=1e-8)
gan.compile(optimizer=optimizer, loss='binary_crossentropy')
gan.summary()
```

Figure 6. GAN model definition

How our model definition works:

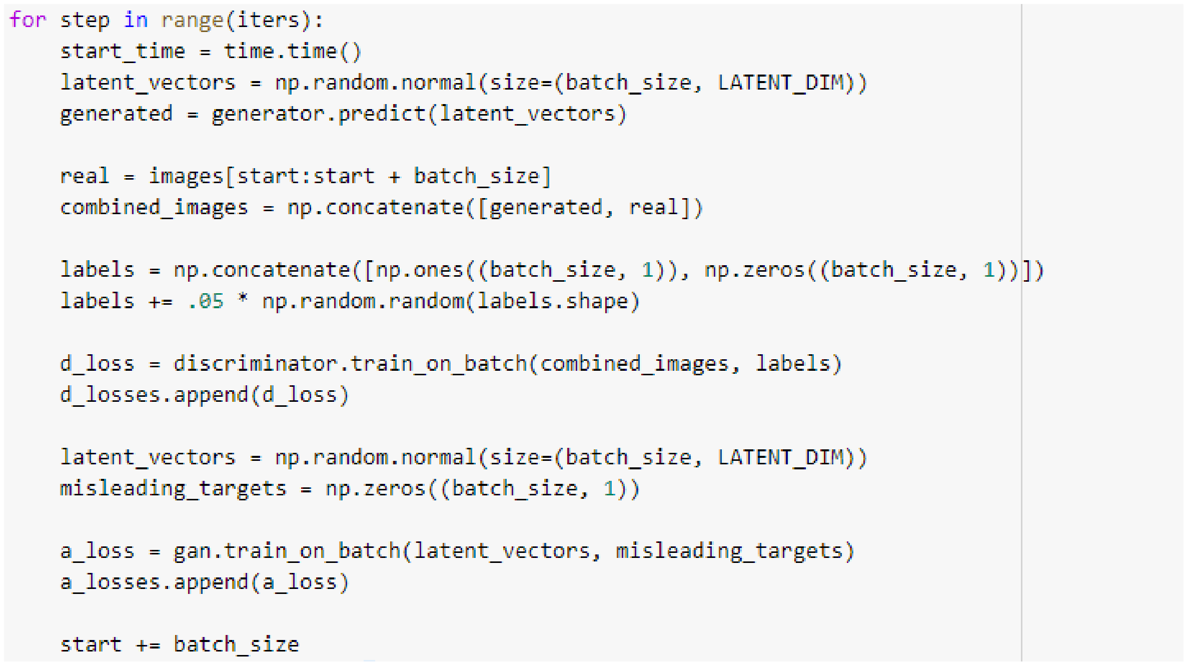
- We set the discriminator to be non-trainable for the model, this means that when we train the adversarial model, only the generator will be updated,
- we define the size of the input, where shape is the size of the input tensor
- we define the output as de discriminator's prediction
- our adversarial model works with an input, and output as parameters
- we create an optimizer for the adversarial model. The optimizer used here is RMSprop.
- Finally, we compile the adversarial model, specifying the optimizer and loss function. The loss function used here is binary cross-entropy.

Training the GAN model.



```
import time
iters = 4000
batch_size = 16

RES_DIR = 'res2'
FILE_PATH = '%s/generated_%d.png'
if not os.path.isdir(RES_DIR):
    os.mkdir(RES_DIR)

CONTROL_SIZE_SQRT = 6
control_vectors = np.random.normal(size=(CONTROL_SIZE_SQRT**2, LATENT_DIM)) / 2

start = 0
d_losses = []
a_losses = []
images_saved = 0
```

These lines define some parameters for the GAN model, including the number of iterations to run (iters), the batch size (batch_size), the name of the directory to save the generated images (RES_DIR) as 'res2' and creates this directory if it does not exist yet, and the file path to save each generated image (FILE_PATH). To keep track of during training, including the index of the first image in the current batch (start), the losses of the discriminator and generator networks (d_losses and a_losses), and the number of images saved so far (images_saved).



```python
for step in range(iters):
    start_time = time.time()
    latent_vectors = np.random.normal(size=(batch_size, LATENT_DIM))
    generated = generator.predict(latent_vectors)

    real = images[start:start + batch_size]
    combined_images = np.concatenate([generated, real])

    labels = np.concatenate([np.ones((batch_size, 1)), np.zeros((batch_size, 1))])
    labels += .05 * np.random.random(labels.shape)

    d_loss = discriminator.train_on_batch(combined_images, labels)
    d_losses.append(d_loss)

    latent_vectors = np.random.normal(size=(batch_size, LATENT_DIM))
    misleading_targets = np.zeros((batch_size, 1))

    a_loss = gan.train_on_batch(latent_vectors, misleading_targets)
    a_losses.append(a_loss)

    start += batch_size
```

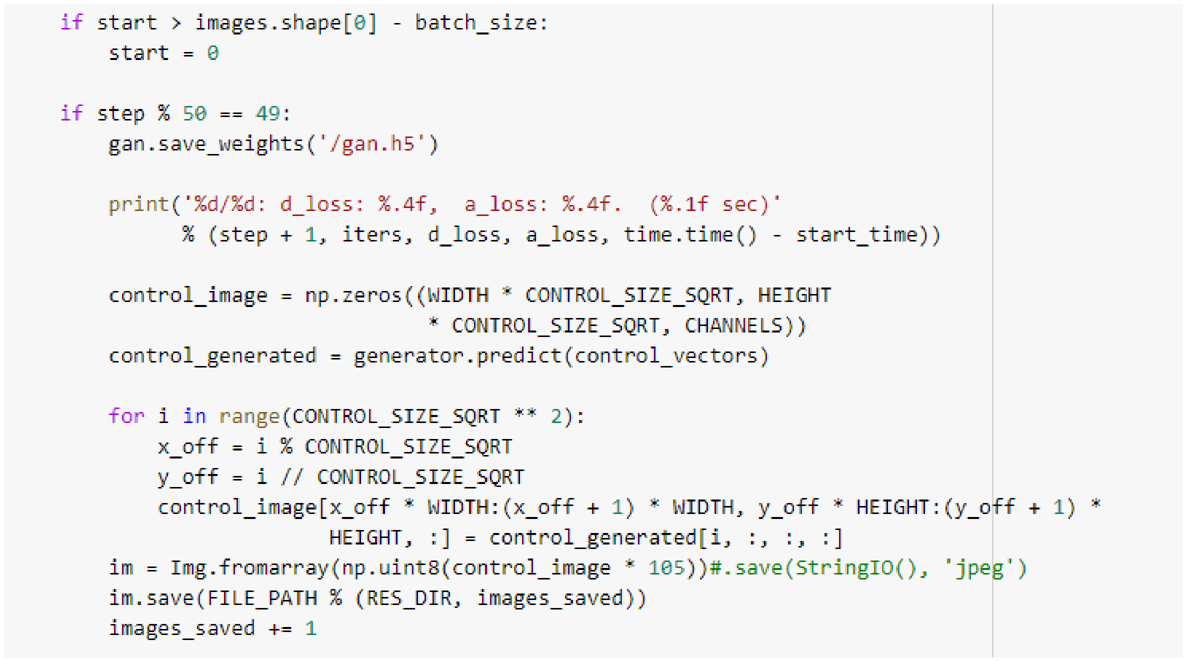
In each iteration of the loop, a batch of random noise vectors (latent_vectors) is generated from a normal distribution of shape (batch_size, LATENT_DIM) using the np.random.normal() function.

The generator network is then used to generate a batch of fake images by passing in the random noise vector using generator.predict()

From a batch of real images of the data sate we combine fake and real images in a single array, then the first batch_size labels are set to 1 to indicate that they are real images, and the next batch_size labels are set to 0 to indicate that they are fake images. A small amount of noise is added to the labels to make the training more robust.

The discriminator network is trained on a combination of the generated fake images and real images from the dataset. The discriminator is trained to classify which faces are real and which are fake. The combined images and their corresponding labels are passed to the discriminator.train_on_batch() function, and its losses are recorded in the d_losses list

Another batch of latent vectors is generated using the random.normal() function. This time, the generator is trained using the combined images and a target of 0 (indicating that the images are fake). This is done to update the weights of the generator model while keeping the discriminator model frozen. The loss for the generator is then recorded in the a_losses list.



```
    if start > images.shape[0] - batch_size:
        start = 0

    if step % 50 == 49:
        gan.save_weights('/gan.h5')

        print('%d/%d: d_loss: %.4f,  a_loss: %.4f.  (%.1f sec)'
              % (step + 1, iters, d_loss, a_loss, time.time() - start_time))

        control_image = np.zeros((WIDTH * CONTROL_SIZE_SQRT, HEIGHT
                                  * CONTROL_SIZE_SQRT, CHANNELS))
        control_generated = generator.predict(control_vectors)

        for i in range(CONTROL_SIZE_SQRT ** 2):
            x_off = i % CONTROL_SIZE_SQRT
            y_off = i // CONTROL_SIZE_SQRT
            control_image[x_off * WIDTH:(x_off + 1) * WIDTH, y_off * HEIGHT:(y_off + 1) *
                          HEIGHT, :] = control_generated[i, :, :, :]
        im = Img.fromarray(np.uint8(control_image * 105))#.save(StringIO(), 'jpeg')
        im.save(FILE_PATH % (RES_DIR, images_saved))
        images_saved += 1
```

The start variable is incremented by the batch size to move to the next batch of real images in the images dataset. If the end of the dataset is reached, the start variable is reset to 0.

Every 50 iterations, the current state of the GAN model is saved using the predefined gan.save_weights() function. The control_image variable is created as an empty array with dimensions (WIDTH * CONTROL_SIZE_SQRT, HEIGHT * CONTROL_SIZE_SQRT, CHANNELS). This array will be used to store a grid of images generated by the generator using a set of fixed input vectors (called control_vectors).

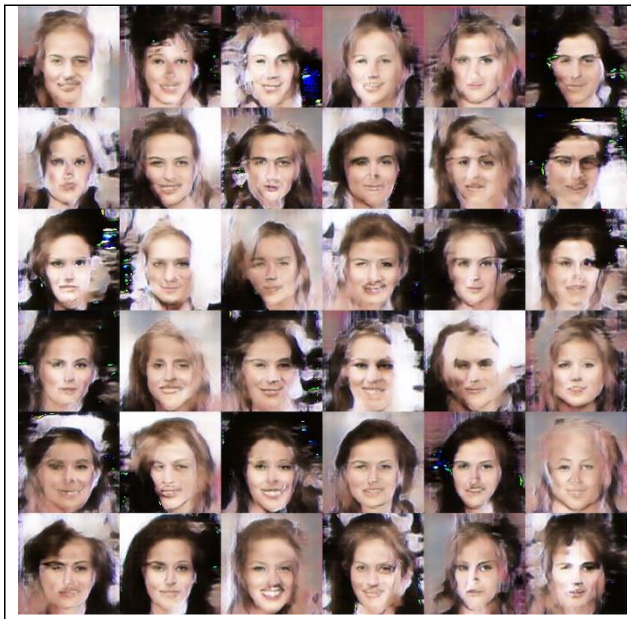
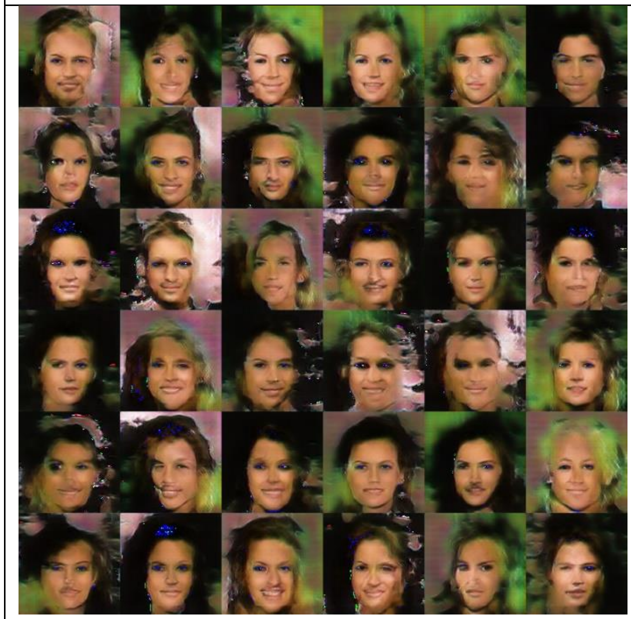
The generator is used to generate a batch of images using the control_vectors input, and the resulting images are stored in the control_generated variable. The control_image array is filled with the generated images from the control_generated variable.

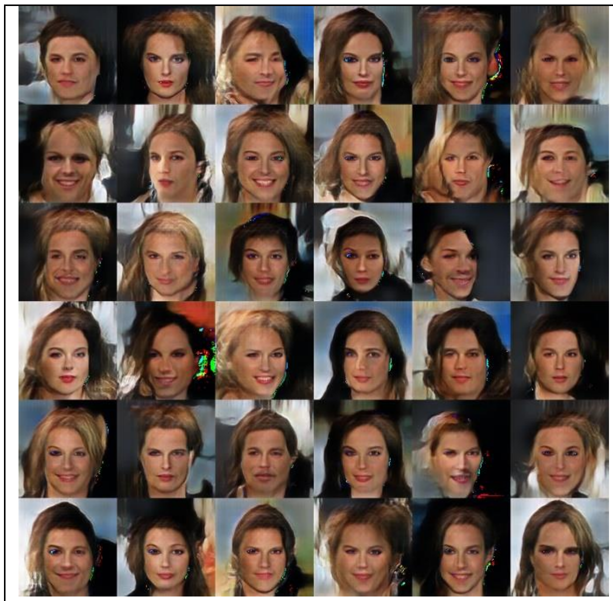
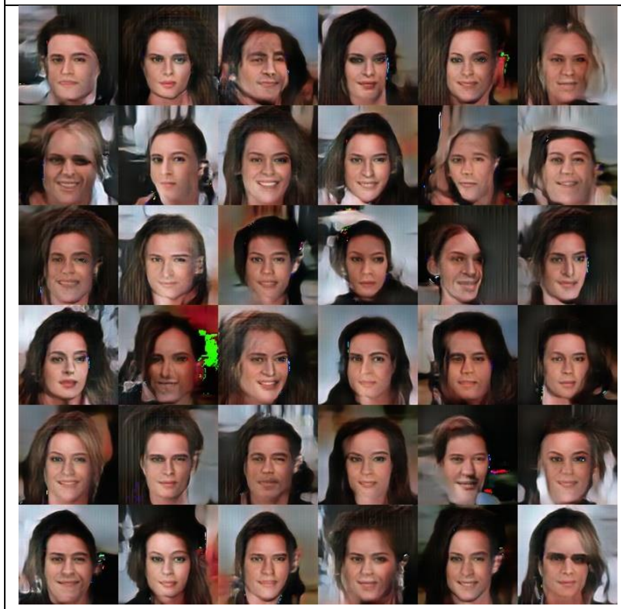
The control_image array is then saved using the Image.fromarray() predefined function. The resulting image is saved using the im.save() function, with the file path determined by the RES_DIR and images_saved variables.

Finally, the images_saved variable is incremented by 1 to keep track of the number of images that have been saved so far. This loop repeats for a total of iters iterations, which is set to by us.

Hands on training.

Here we can see the improvement of some of the obtained results, comparing the results of start and finish extremes of the training.

 Implementation	800 epochs, we can see already see that we are trying to generate human faces, but there is quite a lot of distortion.
 Implementation	1000 epochs, the faces are a bit more defined, but all the faces still have distorted features.

 Implementation	6000 epochs, we have already some good results. The faces are a lot more defined, some faces have unnatural proportions.
 **Implementation**	6500 epochs, we have great results, the main problem we can point is the hair, and some faces that tres to resemble any other position than frontal positioning, but most of them have very defined features.

Finally, we show the performance of this GAN comparing the losses of the two main neural networks.

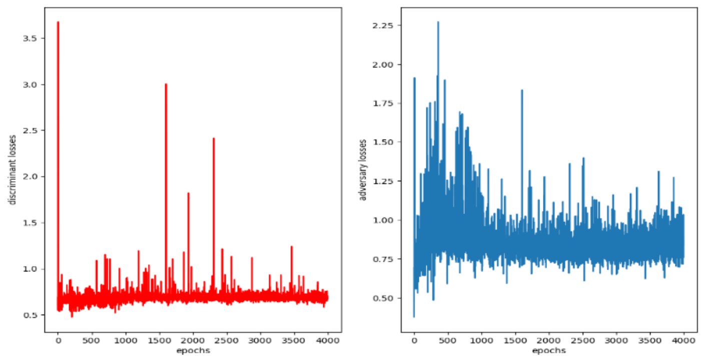


Figure 7. Performance plots

Here we can clearly see that the frequency of looses of the generator network clearly decreases as the training time increases. So, the more training given, the better the results are.

What could I have improved from this GAN model?

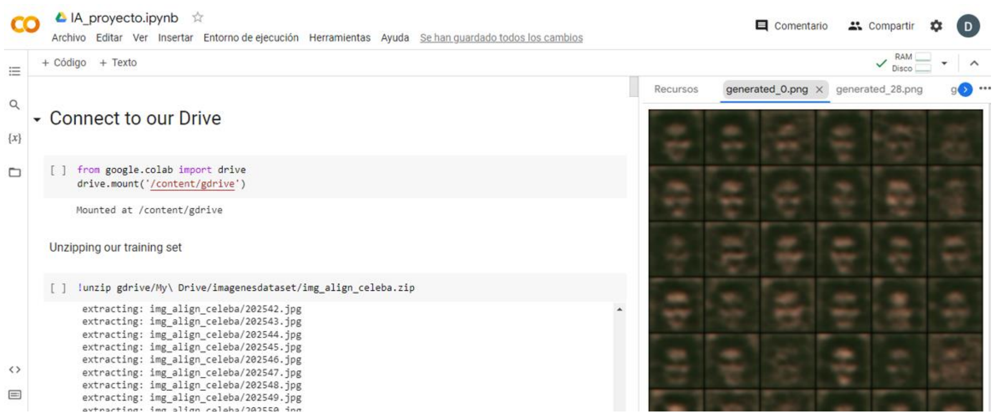
During the compilation of this model, I noticed that I could still see where some generated faces came from, I address this problem due to the very little number of images used (10,000) when the original dataset had 200k images, so the model memorized some faces, that's the reason why we can still see Brad Pitt or Carlisle Cullen in some of the generated faces. Unfortunately, if I added more images, it would take quite longer to get defined faces in the given number of epochs, and the resources wouldn't (GPU and running time) be enough to have decent results using Collaborate free version. At the very end of this process, I realized I could have added checkpoints, and save some weights to run it little by little or in different sessions, so in the future there might be a version of this model with a function that saves the weights that can be used in different sessions.

Conclusions.

Throughout the development of this problem, we confirmed that it is possible to create human faces using a GAN model, where around 6000 epochs we could see some decent results. To get better results, more epochs can be implemented.

This technology has quite a lot of potential to create new information and to develop or improve tools that we already use, such as avatars in social networks, it facilitate some tests in various researches, and it can be used in fashion and marketing industries.

Proof of work.


IA_proyecto.ipynb
Archivo Editar Ver Insertar Entorno de ejecución Herramientas Ayuda Se han guardado todos los cambios
Comentario
Compartir
+ Código + Texto
Recursos
generated_0.png
generated_28.png

Connect to our Drive

```
from google.colab import drive
drive.mount('/content/gdrive')
```

```
Mounted at /content/gdrive
```

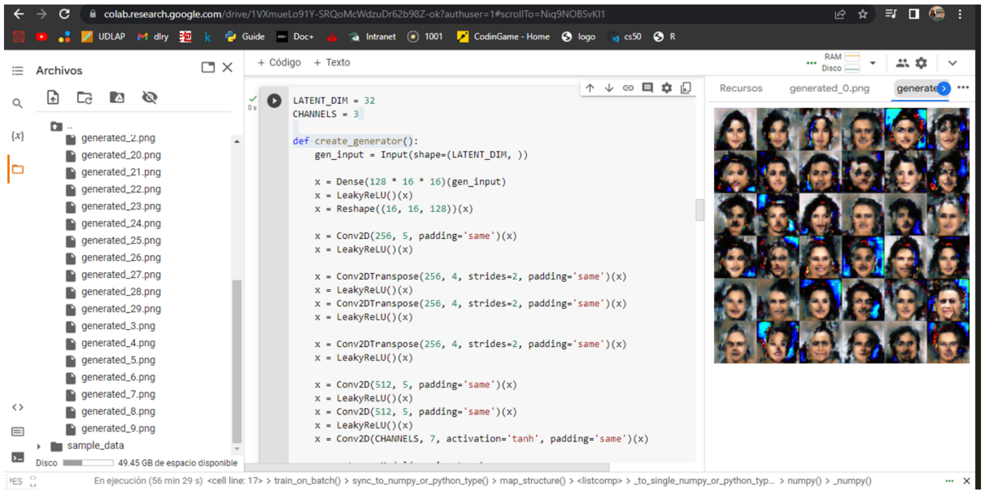
Unzipping our training set

```
!unzip gdrive/My\ Drive/imagenesdataset/img_align_celeba.zip
```

```
 extracting: img_align_celeba/202542.jpg
 extracting: img_align_celeba/202543.jpg
 extracting: img_align_celeba/202544.jpg
 extracting: img_align_celeba/202545.jpg
 extracting: img_align_celeba/202546.jpg
 extracting: img_align_celeba/202547.jpg
 extracting: img_align_celeba/202548.jpg
 extracting: img_align_celeba/202549.jpg
```

Figure 8. Proof of work 1

Here we have generated the first instance of faces.



```
LATENT_DIM = 32
CHANNELS = 3

def create_generator():
    gen_input = Input(shape=(LATENT_DIM, ))

    x = Dense(128 * 16 * 16)(gen_input)
    x = LeakyReLU()(x)
    x = Reshape((16, 16, 128))(x)

    x = Conv2D(256, 5, padding='same')(x)
    x = LeakyReLU()(x)

    x = Conv2DTranspose(256, 4, strides=2, padding='same')(x)
    x = LeakyReLU()(x)
    x = Conv2DTranspose(256, 4, strides=2, padding='same')(x)
    x = LeakyReLU()(x)

    x = Conv2DTranspose(256, 4, strides=2, padding='same')(x)
    x = LeakyReLU()(x)

    x = Conv2D(512, 5, padding='same')(x)
    x = LeakyReLU()(x)
    x = Conv2D(512, 5, padding='same')(x)
    x = LeakyReLU()(x)
    x = Conv2D(CHANNELS, 7, activation='tanh', padding='same')(x)
```

```
En ejecución (56 min 29 s) <cell line: 17> > train_on_batch() > sync_to_numpy_or_python_type() > map_structure() > <listcomp> > _to_single_numpy_or_python_typ... > numpy() > _numpy()
```

Figure 9. Proof of work 2

The program during executions, where the 57 image is generated.

References:

Chauhan, N.S. (2020) *Generate realistic human face using gan*, *The AI dream*. Available at: https://www.theaidream.com/post/generate-realistic-human-face-using-gan-1 (Accessed: April 2, 2023).

Kulpati, S. (2019) A brief introduction to gans (and how to code them), Medium. Sigmoid. Available at: https://medium.com/sigmoid/a-brief-introduction-to-gans-and-how-to-code-them-2620ee465c30 (Accessed: April 2, 2023).

Overview of gan structure | machine learning | google developers (no date) *Google*. Google. Available at: https://developers.google.com/machine-learning/gan/gan_structure (Accessed: April 2, 2023).

Shreyash SinhaShreyash Sinha 2122 bronze badges et al. (1967) What is anti-aliasing in rendering an image in python?, Stack Overflow. Available at: https://stackoverflow.com/questions/66274137/what-is-anti-aliasing-in-rendering-an-image-in-python (Accessed: April 2, 2023).

Team, K. (no date) Keras documentation: Dense layer, Keras. Available at: https://keras.io/api/layers/core_layers/dense/ (Accessed: April 2, 2023).