UNIVERSIDAD DE LAS AMERICAS PUEBLA

PROJECT 2

GAN TO GENERATE FACES

DR. INGRID KIRSCHNING ALBERS

ANGEL DAVID MORENO LOZANO

INTELIGENCIA ARTIFICIAL

02/ABRIL/2023

Generative Adversarial Networks (GANs) are important because they provide a powerful framework for generating synthetic data, particularly images, that are difficult to distinguish from real data. This has important applications in many fields, including computer vision, graphics, and data augmentation for machine learning.

GANs can be used to generate new, realistic images of faces, landscapes, animals, and other objects. By training a generator network on a large dataset of real images, the generator can learn to produce new images that are similar in style and content to the training data. This has important applications in the fields of computer graphics and visual effects, where GANs can be used to generate realistic images of objects and scenes that are difficult or expensive to capture with traditional methods.

GANs also have important applications in the field of data augmentation for machine learning. By generating synthetic images that are similar to real images, GANs can be used to augment training datasets, which can improve the performance of machine learning models. For example, in object recognition tasks, GANs can be used to generate new images of objects from different angles or in different lighting conditions, which can improve the accuracy of object recognition models.

Furthermore, GANs have the potential to revolutionize many areas of computer vision and machine learning. They provide a powerful framework for generating synthetic data that can be used to train and test machine learning models, and they have already been shown to improve the performance of state-of-the-art models in many tasks. Additionally, GANs have the potential to enable new applications in fields such as robotics, autonomous vehicles, and virtual reality, where realistic and diverse visual data is critical for success.

Overall, GANs are important because they provide a powerful framework for generating synthetic data that is difficult to distinguish from real data. They have important applications in many fields, including computer vision, graphics, and machine learning, and have the potential to revolutionize many areas of technology and society. As research into GANs continues, we can expect to see many exciting new developments and applications of this powerful technology.

GANs have gained significant attention from the machine learning community since their introduction in 2014 by Ian Goodfellow and his colleagues. GANs are a type of deep neural network architecture that can generate synthetic data, such as realistic images of faces, by training two neural networks in a game-theoretic setting. The generator network takes random noise as input and produces fake images, while the discriminator network tries to distinguish between fake and real images from a training dataset.

The generator network tries to improve its performance based on the discriminator's feedback, while the discriminator tries to improve its performance based on the generator's output. This process is repeated until the generator generates realistic images that are difficult for the discriminator to distinguish from real images. In the

context of generating faces, the generator takes random noise as input and generates an image that looks like a human face, while the discriminator is trained to distinguish between real and synthetic faces.

To train GANs to generate realistic faces, large amounts of training data are required. The CelebA dataset, which contains over 200,000 celebrity face images, has been widely used in GAN research. Various improvements to the basic GAN architecture have been proposed, such as DCGAN, WGAN, and StyleGAN, which have shown to produce high-quality and diverse images of faces. A good representation of GNAs is "GANs represent a promising approach for unsupervised and semi-supervised learning, as well as generating large amounts of complex data for use in computer vision, robotics, and other applications." (Goodfellow et al., 2014)

GANs are still an active area of research, and there are many challenges and open questions in the field. For instance, improving stability and convergence during training, controlling the generation process, and understanding the underlying principles that govern GANs. However, GANs have already shown great promise in generating realistic and diverse images, and they have the potential to revolutionize many areas of computer vision and image synthesis.

Python Code:

After create our Notebook in Google Colab, we need to put the next lines of code. It is important to mention that the next code was supported by the code in a repository: https://github.com/nageshsinghc4/Face-generation-GAN/blob/master/face_GAN.ipynb

And the personal repository where the next code is located is the next one:

https://github.com/Engel02/GANFORFACES/blob/master/GAN_generate_r_faces.ipynb

So the code was adapted in order to our project.

First, we need to import our drive environment:

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

Then, because our dataset obtained from CELEBA is stored in our drive we need to unzip the folder and select the location. The origin and the destiny.

```
zip_path = '/content/drive/MyDrive/CELEB_ALL/img_align_celeba.zip'

dest_path = '/content/drive/MyDrive/CELEB_ALL/img_align_celeba/'

!unzip -q "{zip_path}" -d "{dest_path}"
```

After that, we import the libraries that we are going to use. This code imports NumPy, Pandas, OS, Tqdm, and Pillow libraries that are required for image processing, data manipulation, and progress bar.

Then we select where the images are located.

With IMAGES_COUNT we can select how many images we are going to use from our dataset. Provides the original width and heigh of the images and the final dimensions that we can, it is recommended select 64x64 or 128x128.

Finally, in image[] and next lines opens each image file in the directory, crops the image according to the specified dimensions, and resizes the image to the desired size. Then, the image is converted to NumPy array and normalized to have pixel values between 0 and 1.

```
import numpy as np
import pandas as pd
import os
#from google.colab import drive
#drive.mount('/content/drive')

PIC_DIR = f'drive/MyDrive/CELEB_A_ALL/img_align_celeba/img_align_celeb
a/'

from tqdm import tqdm
from PIL import Image

IMAGES_COUNT = 200000

ORIG_WIDTH = 178
ORIG_HEIGHT = 218
diff = (ORIG_HEIGHT - ORIG_WIDTH) // 2
```

```
WIDTH = 64
HEIGHT = 64

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))
```

The code snippet above loads a set of images from a directory using the PIL library, crops them, and resizes them to a specified width and height. The resulting images are then normalized by dividing all pixel values by 255, and saved as a NumPy array.

The images variable is a NumPy array of shape (n, height, width, channels), where n is the number of images, height and width are the dimensions of each image, and channels is the number of color channels (in this case, 3 for RGB).

The matplotlib library is then used to plot the first 25 images from the images array in a 5x5 grid.

```
images = np.array(images) / 255
print(images.shape)


from matplotlib import pyplot as plt
```

Result of the previous lines: `(9171, 128, 128, 3)`

The next code block uses Matplotlib to display 25 images from the "images" array.

First, it creates a new figure with a size of 10 by 10 inches using plt.figure(1, figsize=(10, 10)).

Then, a loop is used to display 25 images in a 5 by 5 grid using plt.subplot(5, 5, i+1) to position the images, plt.imshow(images[i]) to display the image data, and plt.axis('off') to remove the axis labels.

Finally, plt.show() is called to display the figure. The resulting output is a grid of 25 images from the "images" array, each with a size of 64 by 64 pixels and normalized between 0 and 1.

```
plt.figure(1, figsize=(10, 10))
```

```
for i in range(25):
    plt.subplot(5, 5, i+1)
    plt.imshow(images[i])
    plt.axis('off')
plt.show()
```

This code imports necessary modules from the Keras deep learning library. Keras is a high-level neural networks API written in Python that can run on top of TensorFlow, CNTK, or Theano. Input module is used to create a Keras tensor with a specific shape. Dense is a fully connected layer, which means that all the neurons in a layer are connected to those in the previous and the next layer. Reshape layer is used to reshape the tensor into the desired shape. LeakyReLU is an activation function that is used to introduce non-linearity into the neural network. Conv2D is a 2-dimensional convolutional layer, which is used to extract features from images.

Conv2DTranspose is a transposed convolutional layer, also known as a deconvolutional layer, which is used to generate images from the features. Flatten is used to convert a multi-dimensional tensor into a 1-dimensional tensor. Dropout is a regularization technique used to prevent overfitting by randomly dropping out some of the neurons in a layer during training. Model is used to create a Keras model, which is a graph of layers that can be trained on data. RMSprop is an optimizer that is used to minimize the loss function during training.

```
from keras import Input
from keras.layers import Dense, Reshape, LeakyReLU, Conv2D, Conv2DTran
spose, Flatten, Dropout
from keras.models import Model
from keras.optimizers import RMSprop

LATENT_DIM = 32
CHANNELS = 3
```

This code defines two functions, create_generator() and create_discriminator(), which create a generator and discriminator respectively using the Keras library in Python.

The create_generator() function takes as input a latent vector of shape (LATENT_DIM,) and returns a Keras Model object representing the generator

network. The generator network takes the latent vector as input, passes it through a series of dense and convolutional layers with LeakyReLU activation functions, and outputs an image tensor of shape (HEIGHT, WIDTH, CHANNELS).

The create_discriminator() function takes as input an image tensor of shape (HEIGHT, WIDTH, CHANNELS) and returns a Keras Model object representing the discriminator network. The discriminator network takes the image tensor as input, passes it through a series of convolutional and dense layers with LeakyReLU activation functions, and outputs a scalar value representing the likelihood that the input image is real, as opposed to generated by the generator network. After defining the generator and discriminator networks, the code sets the discriminator to be non-trainable so that it is not updated during the training process of the combined generator and discriminator model.

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

    x = Dense(32 * 8 * 8)(gen_input)
    x = LeakyReLU()(x)
    x = Reshape((8, 8, 32))(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

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(
        learning_rate=.0001,
        clipvalue=1.0,
        decay=1e-8
    )

    discriminator.compile(
        optimizer=optimizer,
        loss='binary_crossentropy'
    )

    return discriminator



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

The next code first defines a generator function create_generator() that takes the random input vector and generates an image using a series of convolutional and transposed convolutional layers, with leaky ReLU activation functions. The generator uses a tanh activation function in the output layer to ensure the pixel values are between -1 and 1.

Then, it defines a discriminator function create_discriminator() that takes an image as input and predicts whether it's real or fake. The discriminator is a convolutional neural network that has several convolutional layers followed by a few dense layers. The output of the discriminator is a single sigmoid activation function that produces a value between 0 and 1 indicating the probability of the input image being real.

Finally, it creates the full GAN model by combining the generator and discriminator. The generator is trained to generate realistic images that can fool the discriminator, and the discriminator is trained to correctly classify real and fake images. The GAN model is compiled with a binary cross-entropy loss function and an RMSprop optimizer with a learning rate of 0.0001 and clip value of 1.0.

```
gan_input = Input(shape=(LATENT_DIM, ))
gan_output = discriminator(generator(gan_input))
gan = Model(gan_input, gan_output)

optimizer = RMSprop(learning_rate=.0001, clipvalue=1.0, decay=1e-8)
gan.compile(optimizer=optimizer, loss='binary_crossentropy')
```

Finally, The generator takes a random input vector of size LATENT_DIM and produces an image that should look similar to the dataset. The discriminator takes an image and tries to predict whether it is a real image from the dataset or a fake image generated by the generator.

During training, the discriminator and the generator are trained alternately. First, the discriminator is trained on a batch of real and fake images with their corresponding labels. Then, the generator is trained to fool the discriminator by generating images that are predicted as real. This process is repeated for several iterations.

Every 50 iterations, the weights of the GAN are saved and a grid of images is generated to visualize the progress of the generator. Finally, the code generates a GIF animation of the saved images.

Actually, in this part we can select the number of epochs that we want, in the variable called iteres, so in this case we selected 500 and selected the location where the GIF file will be stored and the name of it.

```
import time
iters = 500
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
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
    if start > images.shape[0] - batch_size:
        start = 0
```

```
    if step % 50 == 49:
        gan.save_weights('./drive/MyDrive/gan12.h5')

        print('%d/%d: d_loss: %.4f,  a_loss: %.4f.  (%.1f sec)' % (ste
p + 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 * H
EIGHT:(y_off + 1) * HEIGHT, :] = control_generated[i, :, :, :]
        im = Image.fromarray(np.uint8(control_image * 255))
        im.save(FILE_PATH % (RES_DIR, images_saved))
        images_saved += 1


plt.figure(1, figsize=(12, 8))
plt.subplot(121)
plt.plot(d_losses)
plt.xlabel('epochs')
plt.ylabel('discriminant losses')
plt.subplot(122)
plt.plot(a_losses)
plt.xlabel('epochs')
plt.ylabel('adversary losses')
plt.show()

import imageio
import shutil

images_to_gif = []
for filename in os.listdir(RES_DIR):
    images_to_gif.append(imageio.imread(RES_DIR + '/' + filename))
imageio.mimsave('drive/MyDrive/visual12.gif', images_to_gif)
shutil.rmtree(RES_DIR)
```

Results

Now, the best part, the running of the previous code with some modifications.

In order to make a better organization, the structure of the result is the next:

First if the description of the experiment, then the size of the final images, number of images used and the epochs. Then a screenshot about the final operation where the code finalize. Another screenshot about the graph and finally a screenshot of the images of faces.

Experiment 1

SIZE	#IMAGES	#EPOCHS
128x128	3,171	50

Because this was the first experiment, well the panic was in the air, we take the general recommendations about epochs, images, batch size and learning rate.

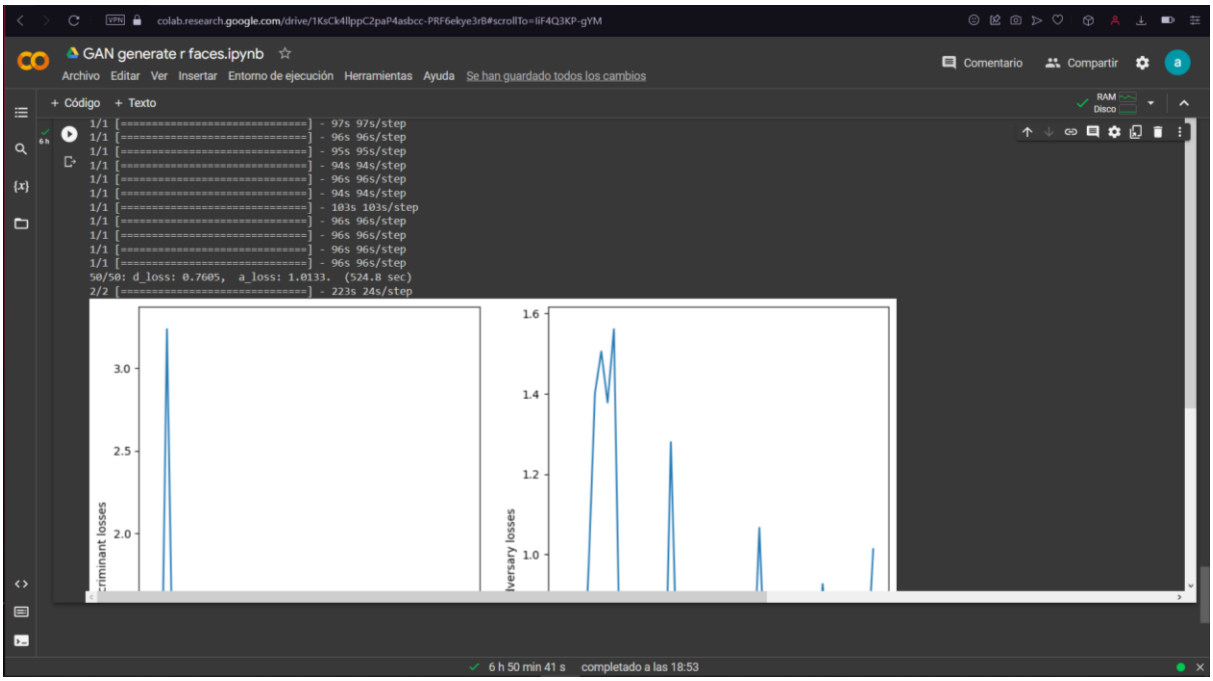
The result is the output of the GAN training loop. The loop is designed to train the discriminator and the generator in an adversarial fashion for a specified number of iterations.

In this case, the loop has been run for 50 iterations. During each iteration, the generator produces fake images from random noise vectors, which are then fed into the discriminator along with real images from the dataset. The discriminator is then trained to distinguish between the real and fake images. The generator is trained to generate fake images that fool the discriminator.

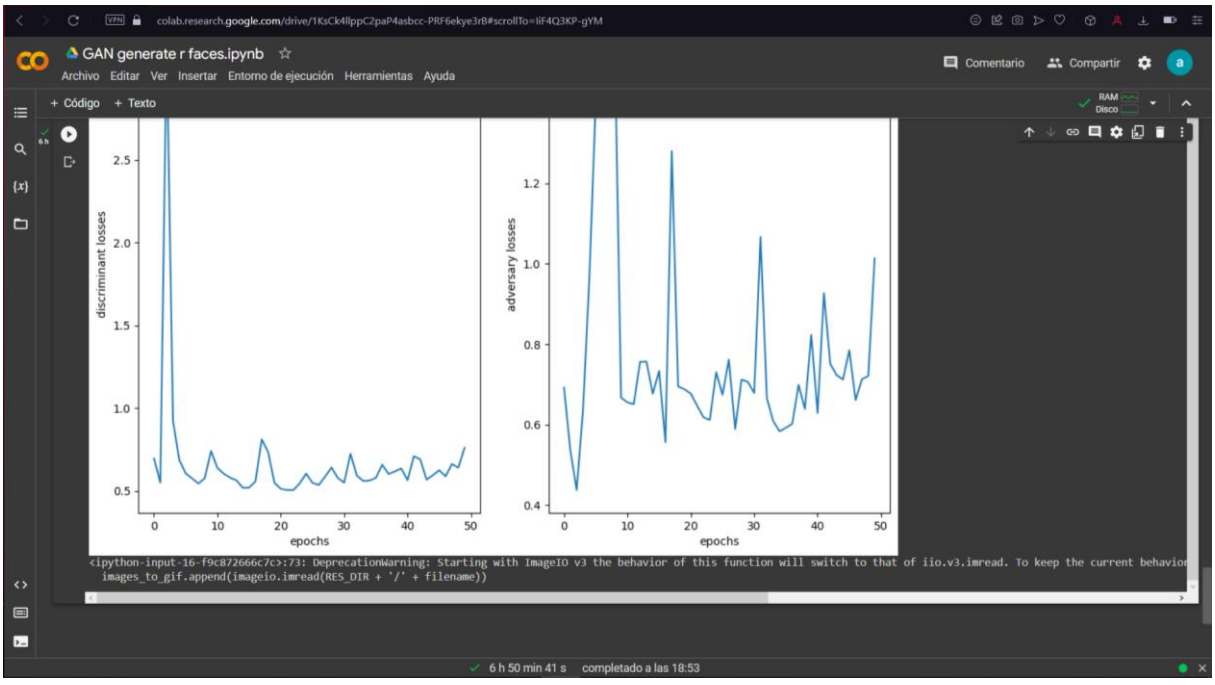
The reported result shows the current iteration number (50/50), the current discriminator loss (d_loss) which is the binary cross-entropy loss of the discriminator on the combined (real and generated) images, and the current adversarial loss (a_loss), which is the binary cross-entropy loss of the generator when the discriminator is fooled by the generated images. The reported time (524.8 sec) is the time taken to complete the current iteration.

The reported d_loss of 0.7605 indicates that the discriminator is a little bit able to distinguish between real and fake images with a certain degree of accuracy. A high

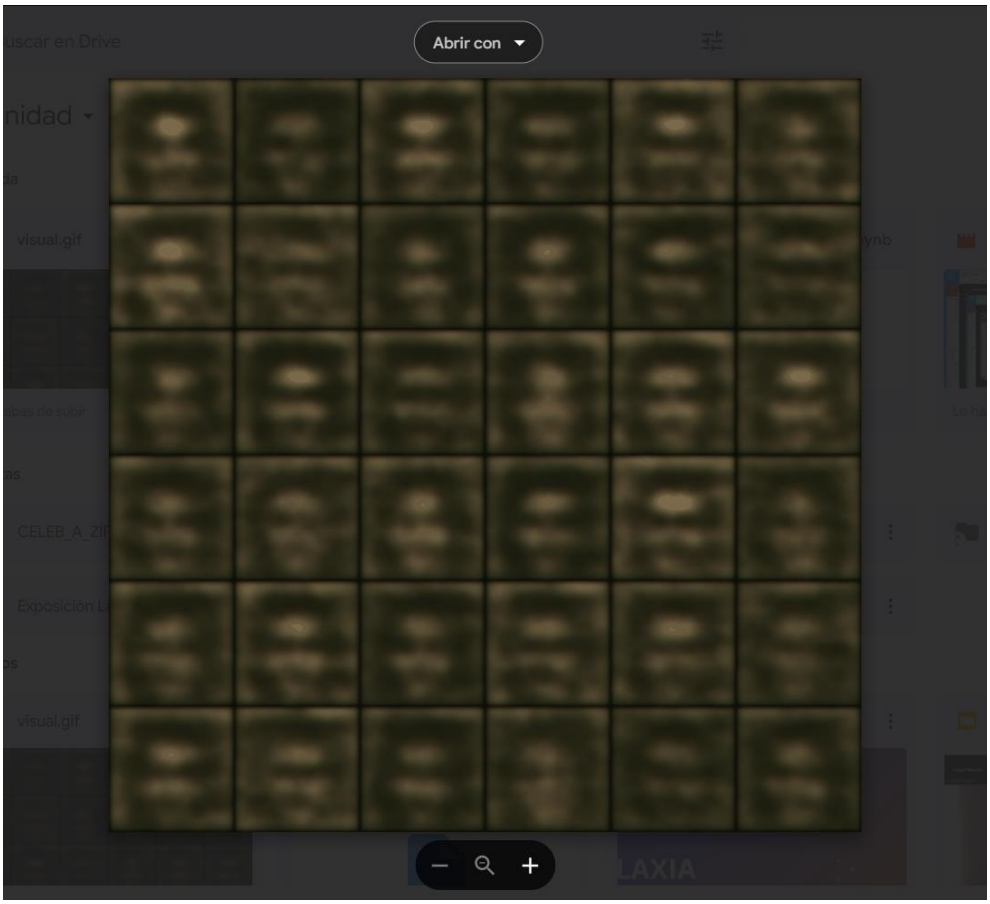
d_loss value indicates that the discriminator is performing well and is able to distinguish between real and fake images. The reported a_loss of 1.0133 indicates that the generator is not at all able to generate images that can fool the discriminator to some extent. A low a_loss value indicates that the generator is performing well and is able to generate images that closely resemble the real images.



Experient 1. Image 1. Epochs done.



Experient 1. Image 2. Graphics.

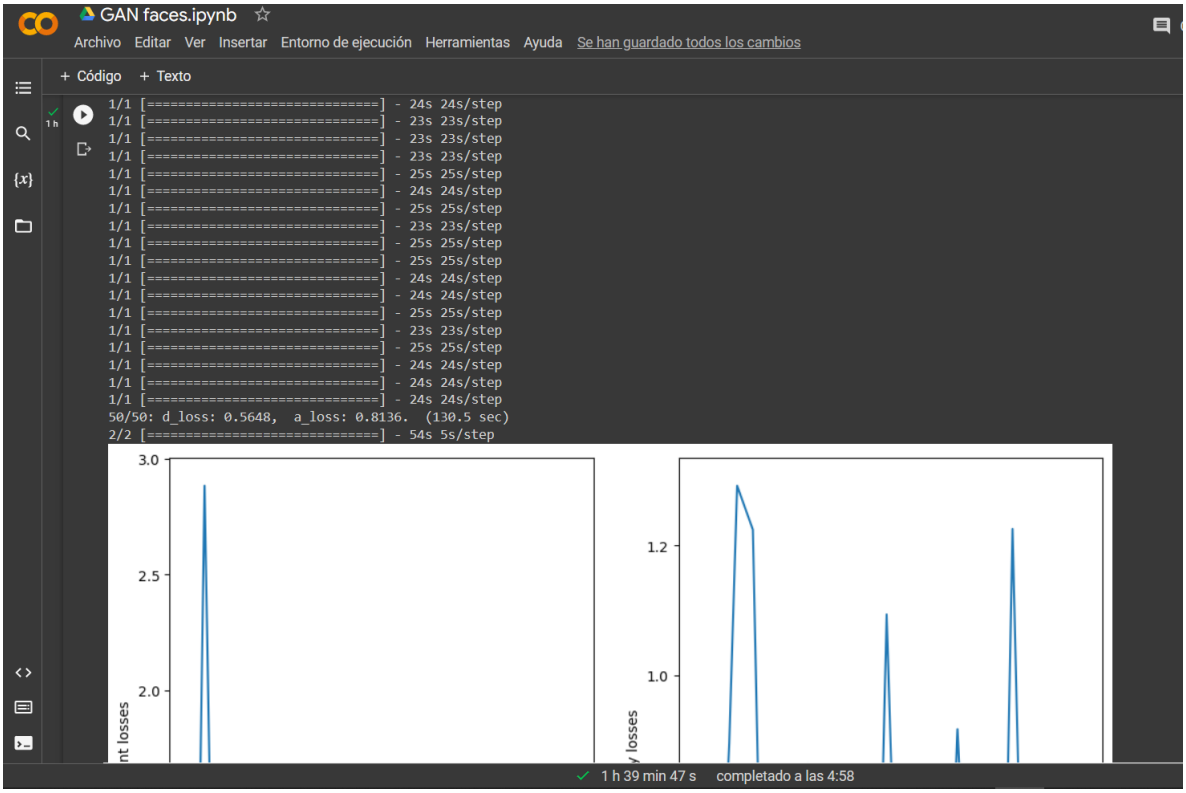


Experient 1. Image 3. Images Results.

In the image 3 of the experiment 1 maybe is not a perfect face, actually anyone that see that may tell that is not a face of a person. Like a first look at it, we can tell that is because is the first running of the code, maybe we need to train it much more, maybe is the quantity of the images, so many thins to check. A good factor are the graphs, because we can think about what can be wrong or change some parameters, but is a beginning, in the next experiments we can see the progress of the face generator.

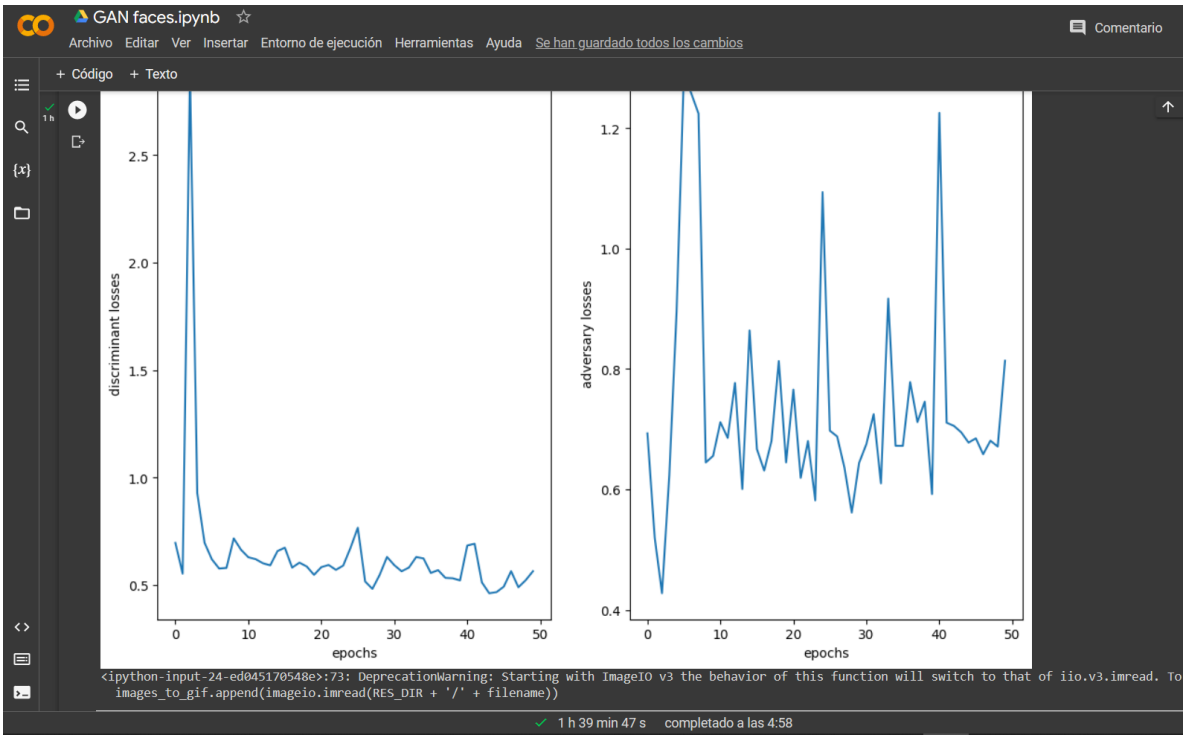
Experiment 2

SIZE	#IMAGES	#EPOCHS
64x64	20,000	50

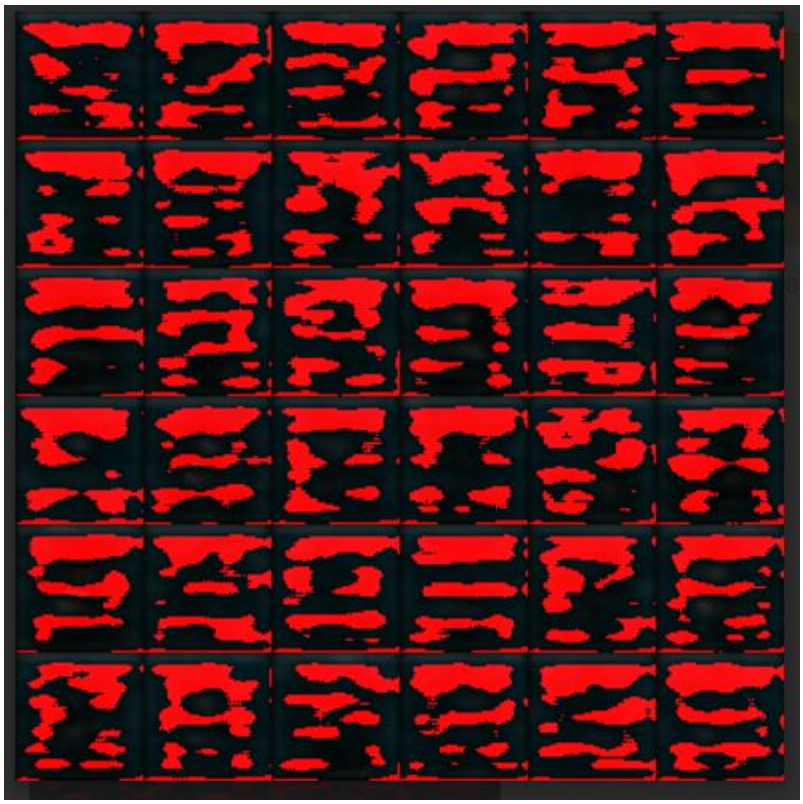


Experient 2. Image 1. Epochs done.

In the image 1 of the experiment 2, the reported d_loss of 0.5648 indicates that the discriminator is able to distinguish between real and fake images with a certain degree of accuracy. The reported a_loss of 0.8136 indicates that the generator is not at all able to generate images that can fool the discriminator to some extent.



Experient 2. Image 2. Graphics.

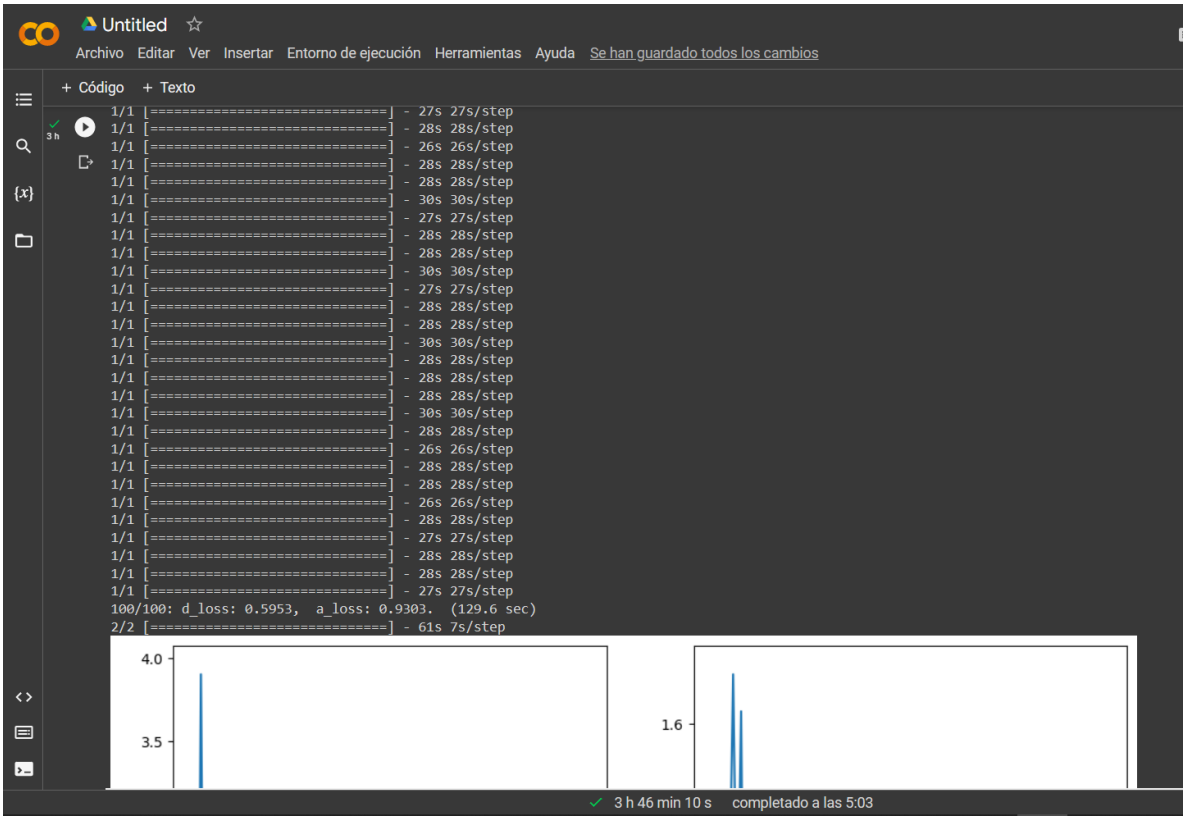


Experient 2. Image 3. Images Results.

In the results of the previous image 3, something terrible happened, but what? Maybe because in the second experiment we change the size of the final images (64x64, first exp 128x128) or maybe the number of images because we increase too many images for the second experiment. It is a good option think about that maybe changing the number of images and increase the epochs could be a positive difference.

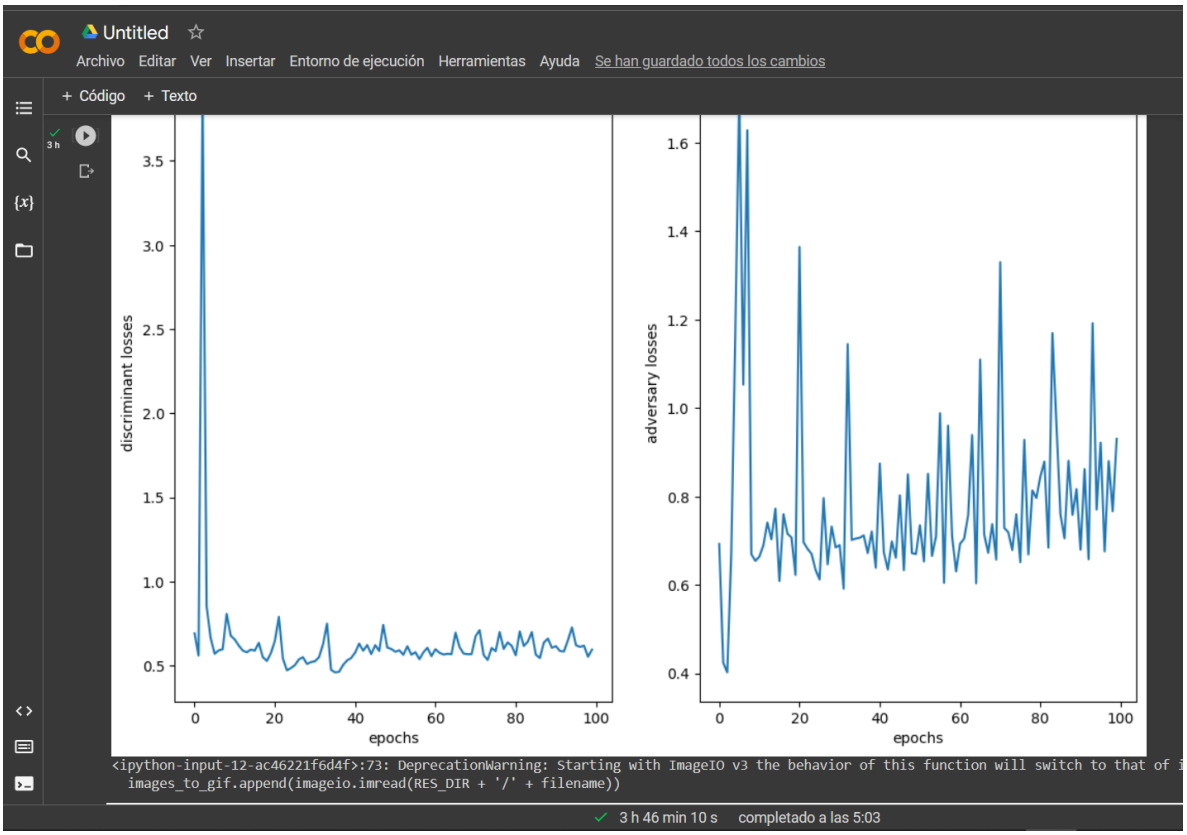
Experiment 3

SIZE	#IMAGES	#EPOCHS
64x64	10,232	100



Experient 3. Image 1. Epochs done.

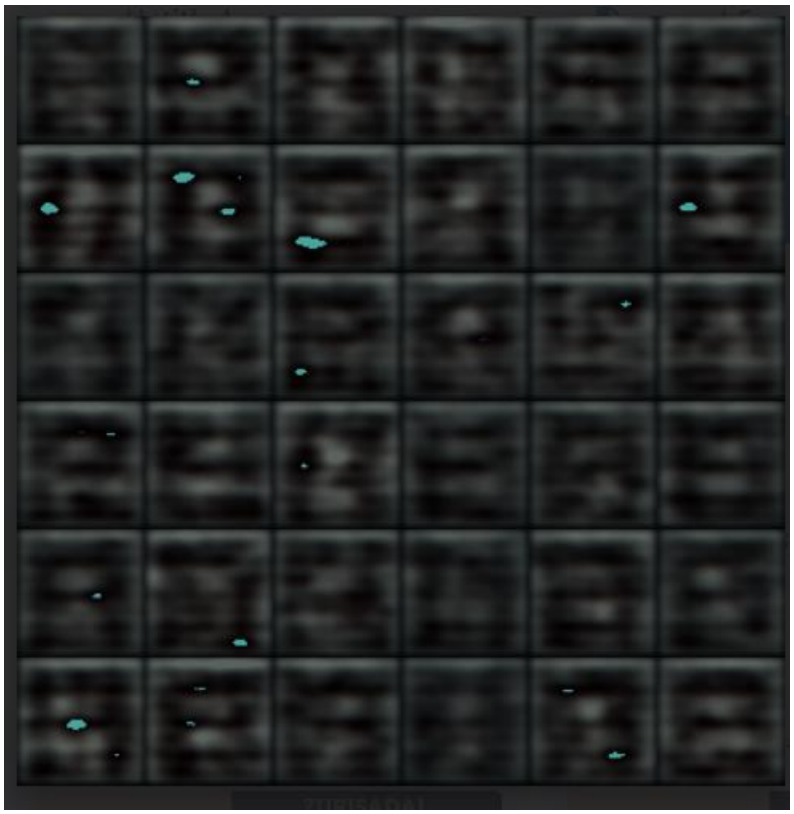
Now applying the changes lets see what happened.



Experient 3. Image 2. Graphics.

We can see in the images 2 about the graphics that the losses of the discriminator are lesser than the adversary losses along the 100 epochs.

Like an observation, something interesting is that "Recent research has demonstrated the vulnerability of GANs to adversarial attacks, which can cause them to produce misleading or incorrect results." (Balaji & Chakraborty, 2021, p. 2). So the error are common, just is important investigate what can we do in order to minimize them.



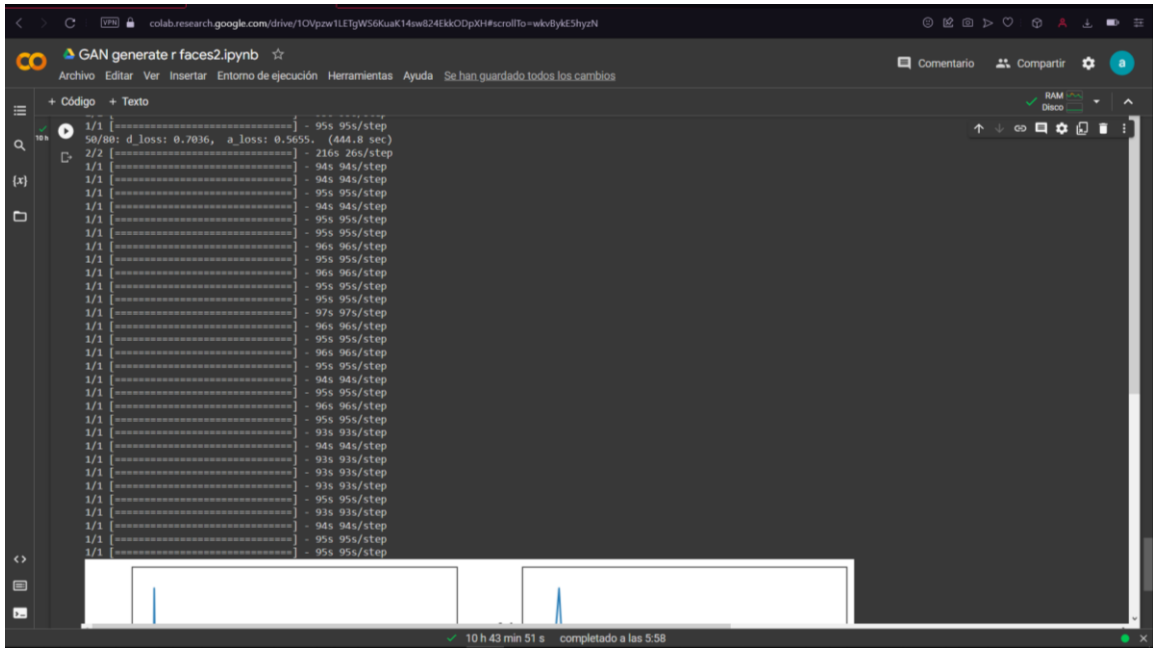
Experient 3. Image 3. Images Results.

Here the results in the image 3 are not exactly what we want, maybe we can try change the size and reduce the number of images.

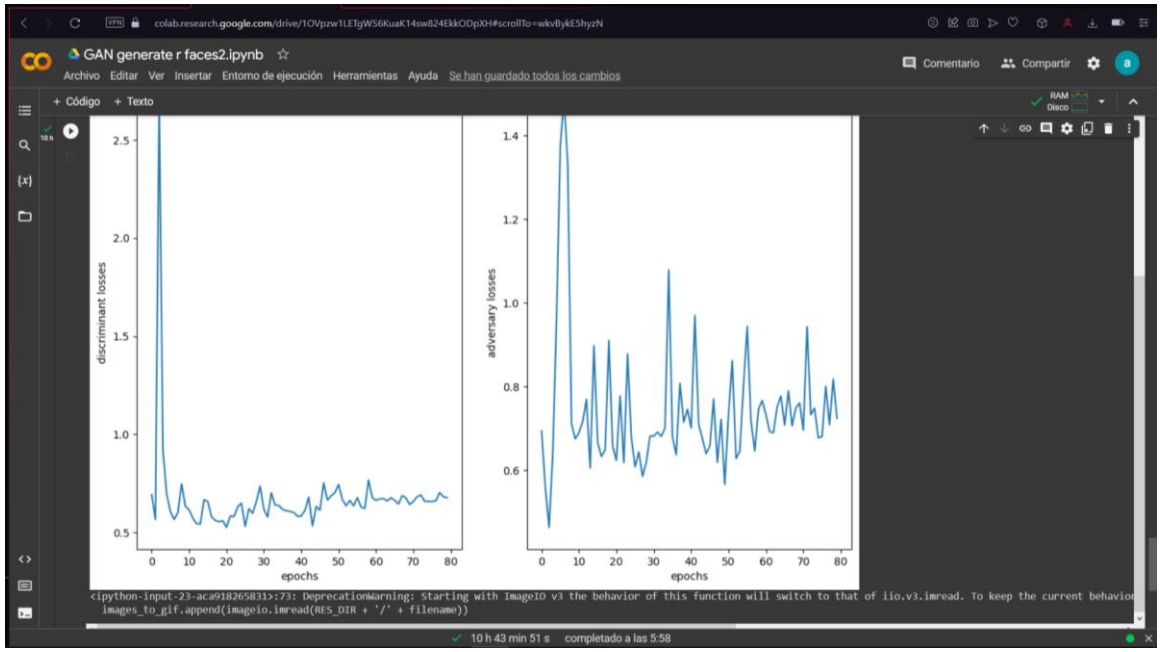
A phrase for the situations like this, is interesting say that "GANs have been applied successfully to a range of tasks, from generating realistic images and videos to improving image-to-image translation and data augmentation. However, training GANs remains a challenging task due to issues such as mode collapse, vanishing gradients, and instability." (Gulrajani et al., 2017). If errors appears, just be focus and fix them.

Experiment 4

SIZE	#IMAGES	#EPOCHS
128x128	3171	80



Experient 4. Image 1. Epochs done.



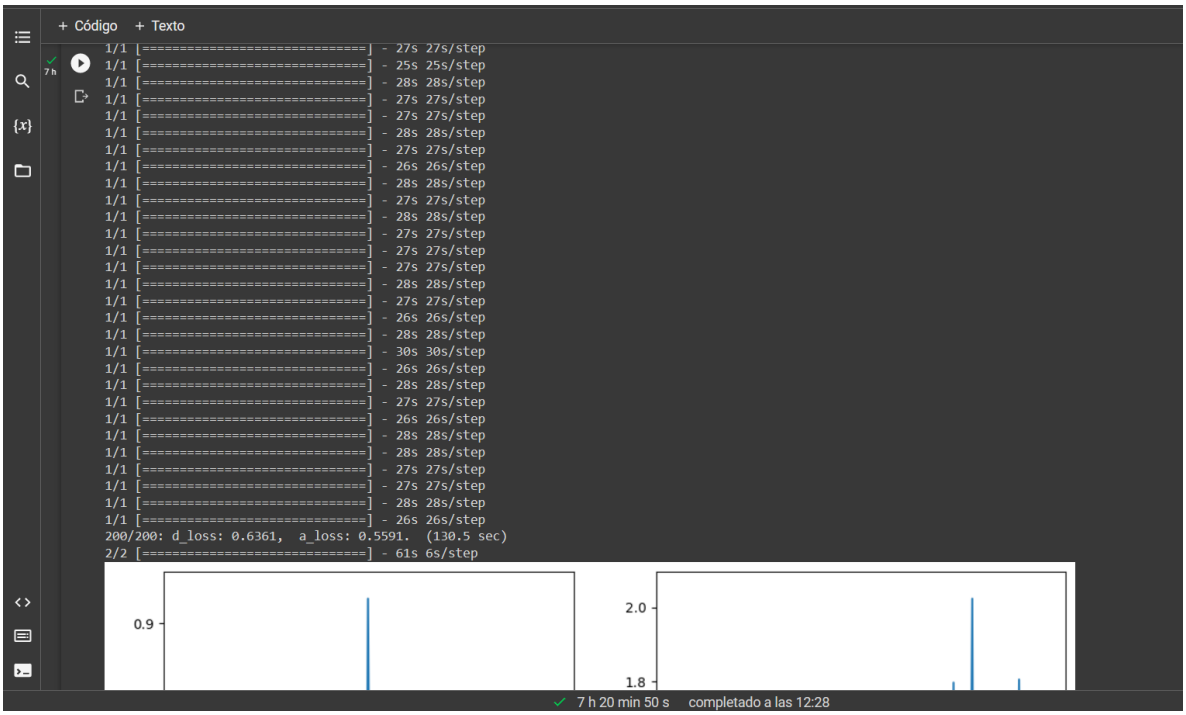
Experient 4. Image 2. Graphics.

Experient 4. Image 3. Images Results.

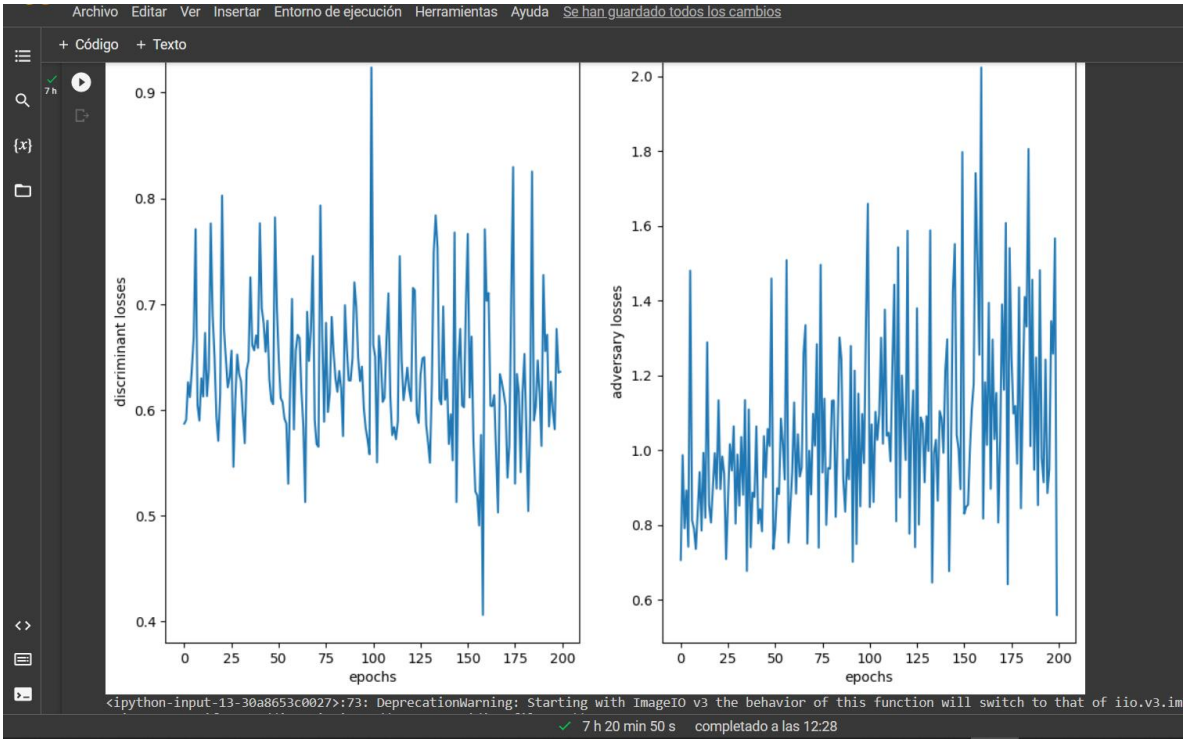
In the experiment 4 we have an important advance, because we can see that the model is trying to make a human face, the color is not the correct but we have good results.

Experiment 5

SIZE	#IMAGES	#EPOCHS
64x64	10,232	200



Experient 5. Image 1. Epochs done.



Experient 5. Image 2. Graphics.

Here in the image 2 of the graphs we can see a similar behavior between losses along the 200 epochs.

Experient 5. Image 3. Images Results.

Experiment 5 is the most successful so far, although it seems like something terrifying and creepy is a very good advance, we can see human deformed faces in the results.

In the next episode we have something interesting and something embarrassing, a confession, you can see in the previous images that the time of execution of the epochs is for hours, 3, 7, 10 and more, but to be honest I did not realize that google colab was not configurated in the way that it suppose to be, I did not configure the environment execution and I did not select the GPU. That was a big mistake that I supposed that was configurated, but now we can work faster and more efficiently.

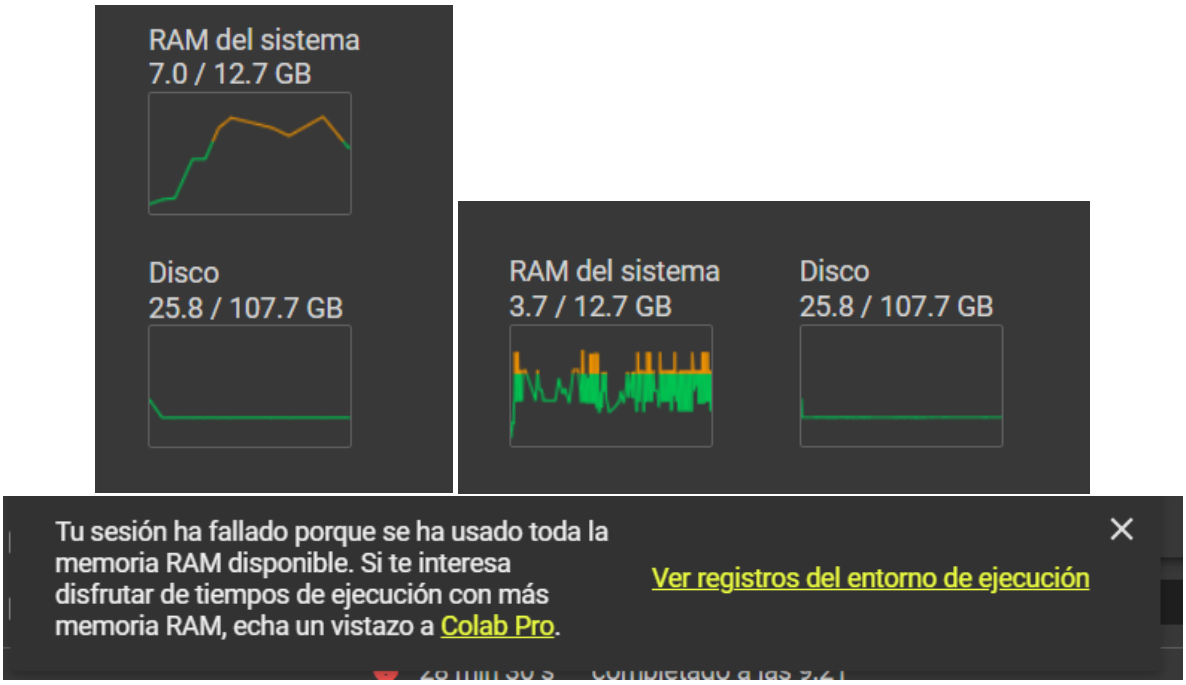
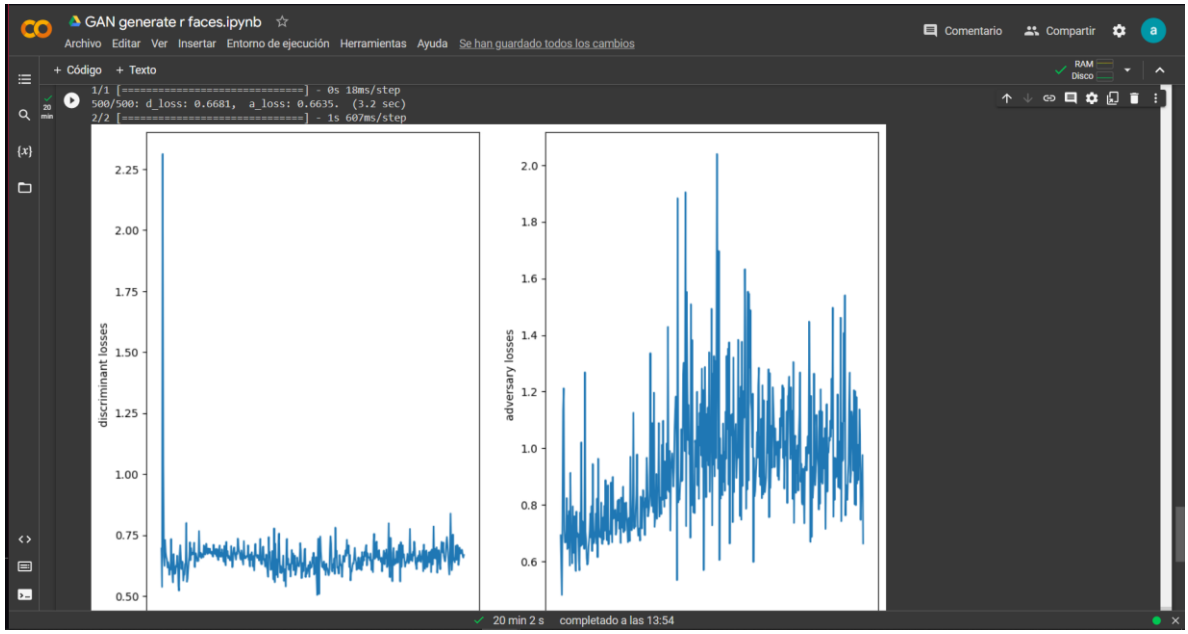


Image 1. Errors.

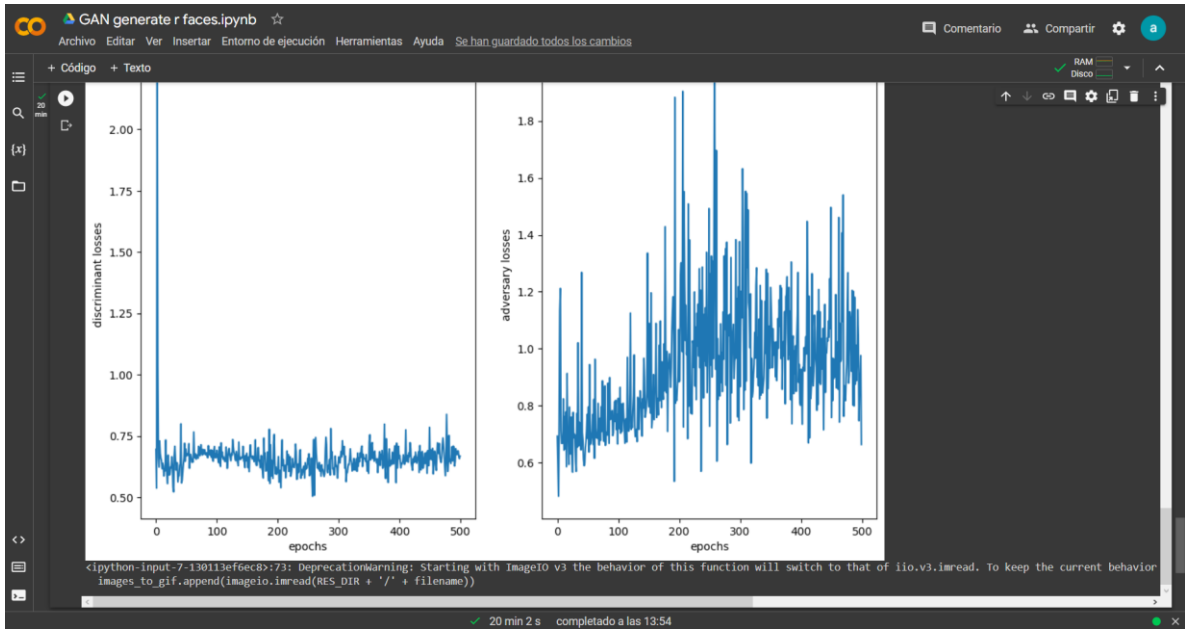
The errors along the experiments were frustrating, working for hours and once you are in the last epoch the memory of google colab or session are not able anymore, so it is important make some checkpoint just to be sure and take in mind other factors that can affect.

Experiment 6

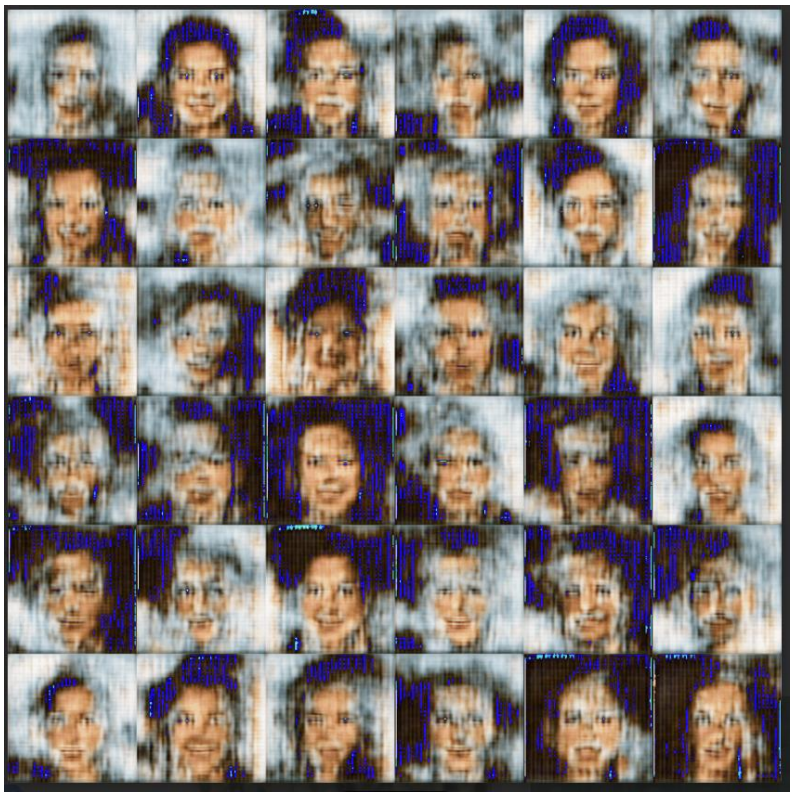
SIZE	#IMAGES	#EPOCHS
128x128	8,200	500



Experient 6. Image 1. Epochs done.



Experient 6. Image 2. Graphics.

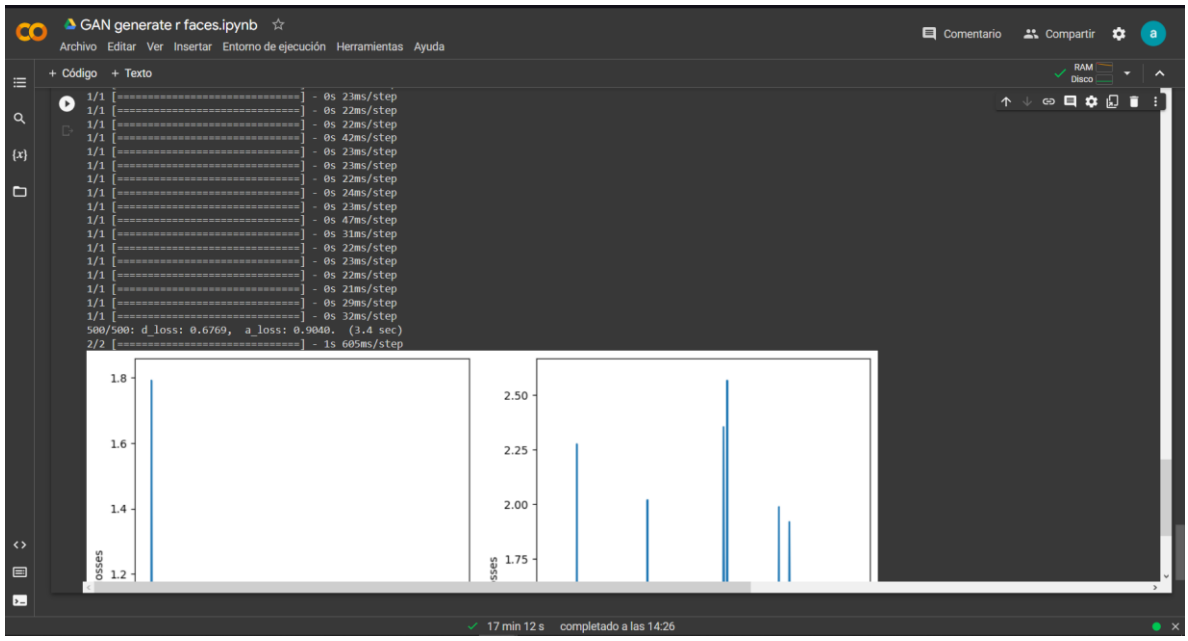


Experient 6. Image 3. Images Results.

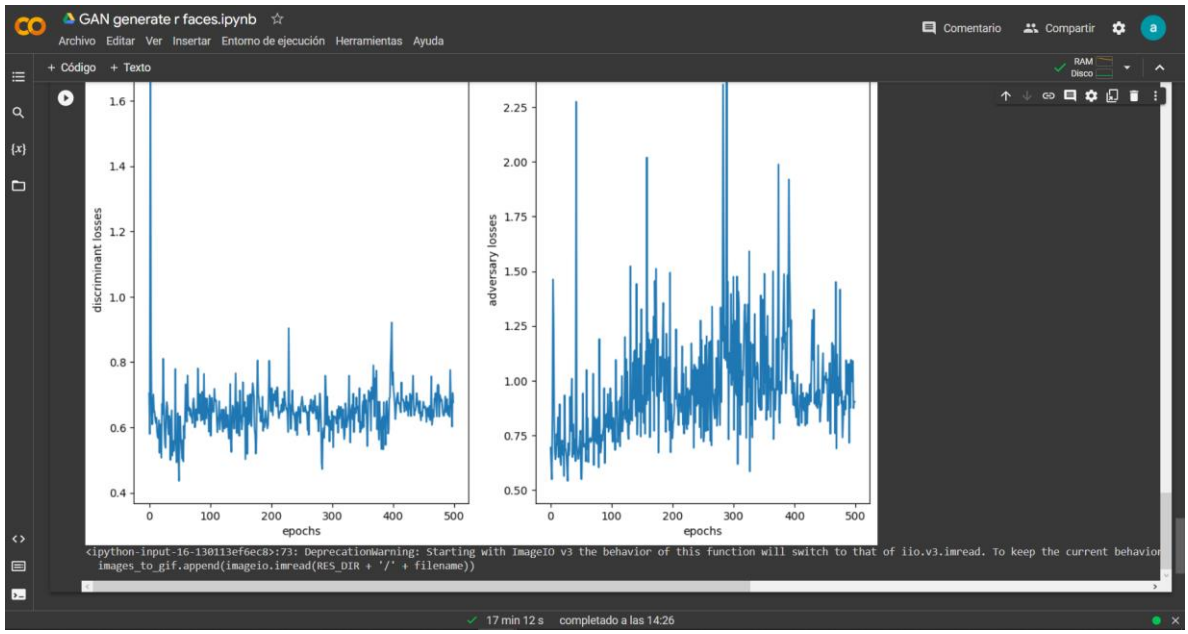
Experiment 6 looks promising, now we can distinguish human faces, we are going to the right way.

Experiment 7

SIZE	#IMAGES	#EPOCHS
128x128	12,000	500



Experient 7. Image 1. Epochs done.



Experient 7. Image 2. Graphics.

Experient 7. Image 3. Images Results.

After some experiments we can see that when we try to make the final size 128x128 the code running slower, so what if we prefer 64x64 instead if 128x128.

Experiment 8

SIZE	#IMAGES	#EPOCHS
64x64	7,000	350

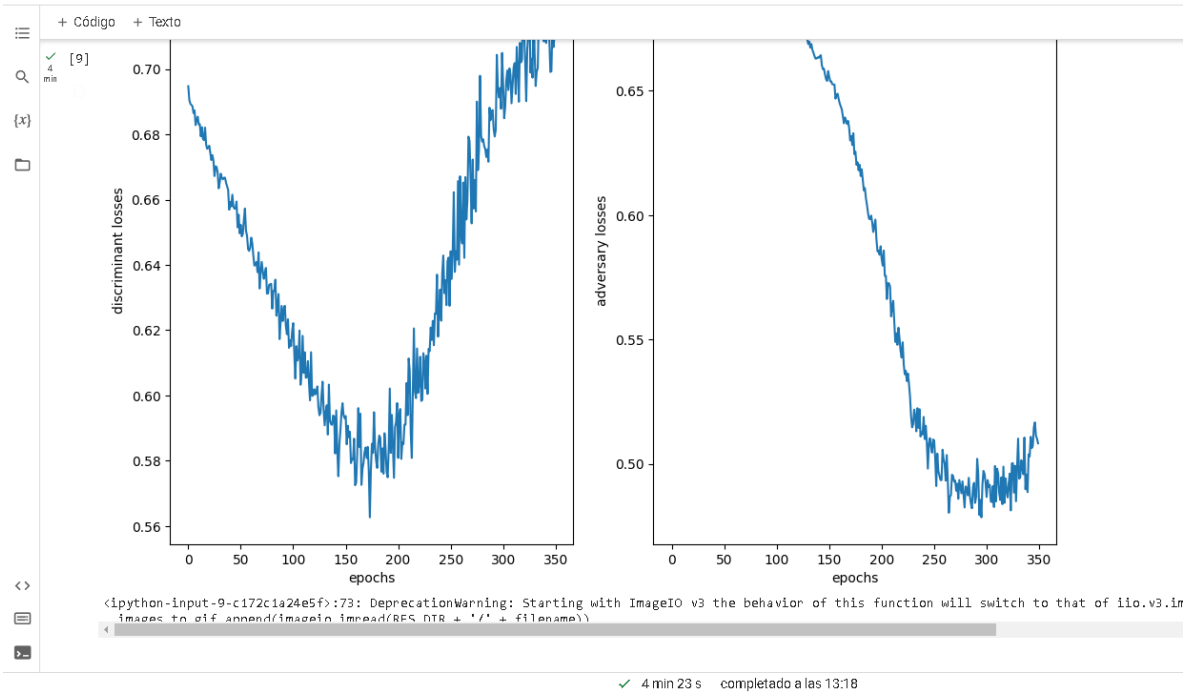
Experient 8. Image 1. Images Results.

Just another creepy image, but the color and appearance looks more for a human face.

Experiment 9

SIZE	#IMAGES	#EPOCHS
64x64	7,000	350

In this case we modify the learning rate to 0.000001 (Originally 0.0001) because in some lectures they say that we can try to modify this in order to see what happened and maybe the problem is related to that.



Experient 9. Image 1. Graphics.

Since we see the graphs, we can see a particular behavior on the losses, so you know something will be wrong.

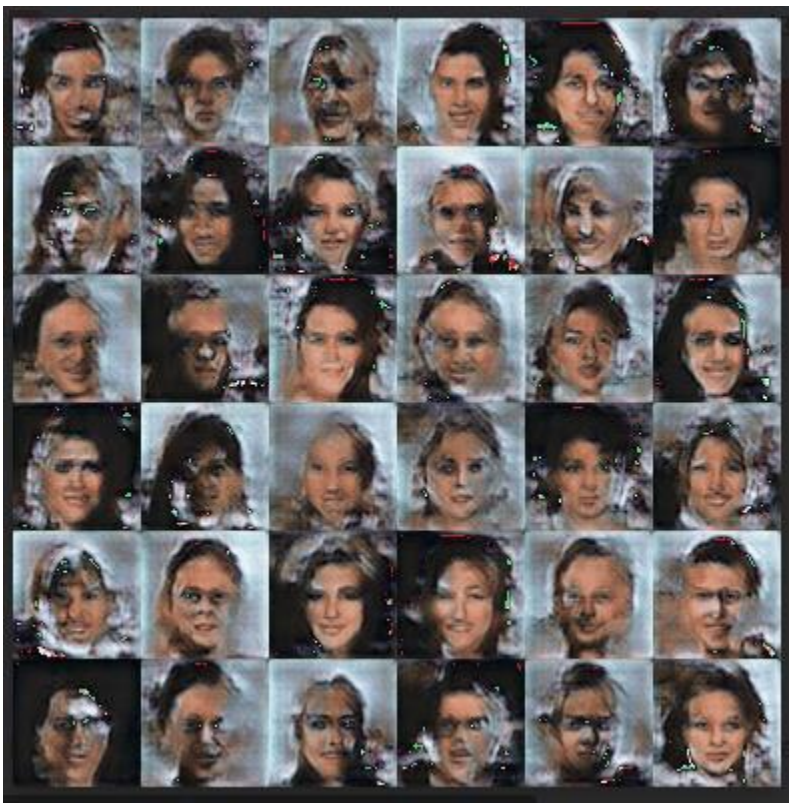
Experient 9. Image 2. Images Results.

In fact the error is worst that we though, basically is nothing, lets change the learning rate to the previous value.

Experiment 10

SIZE	#IMAGES	#EPOCHS
64x64	7,000	1,350

After the fatal error in the previous experiment, the learning_rate returns to 0.001 in the next codes.

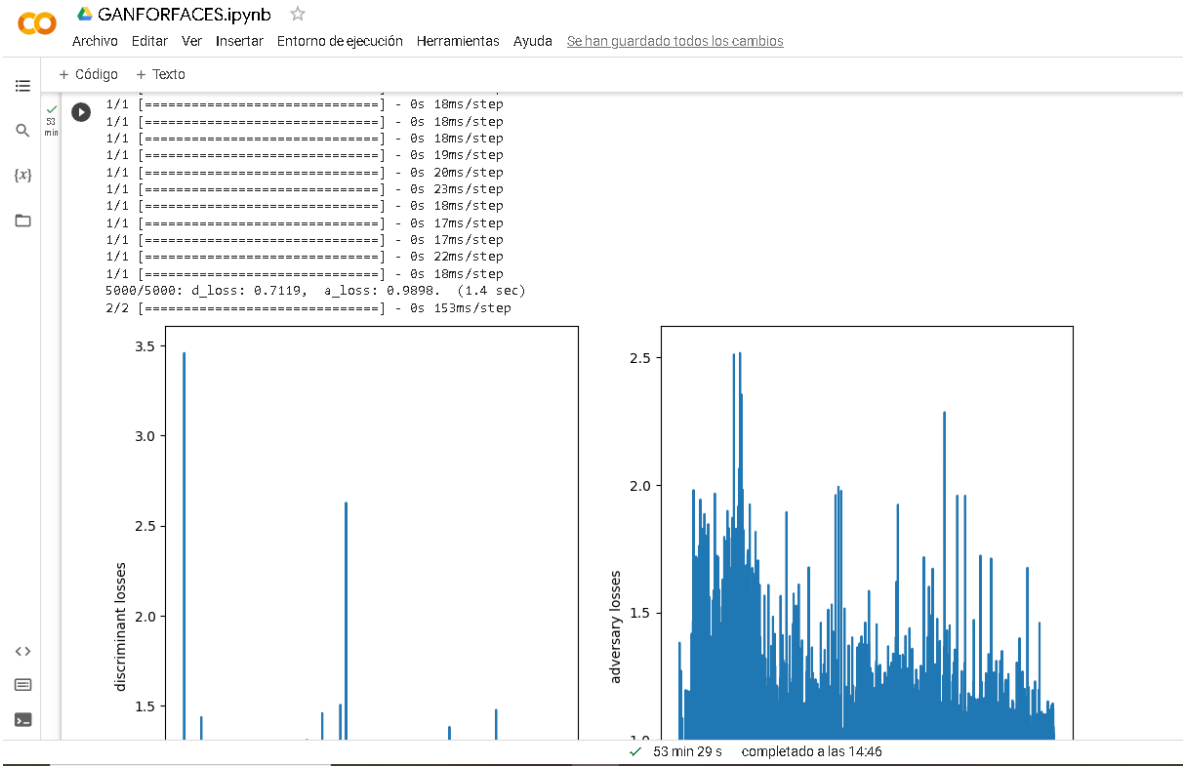


Experient 10. Image 1. Images Results.

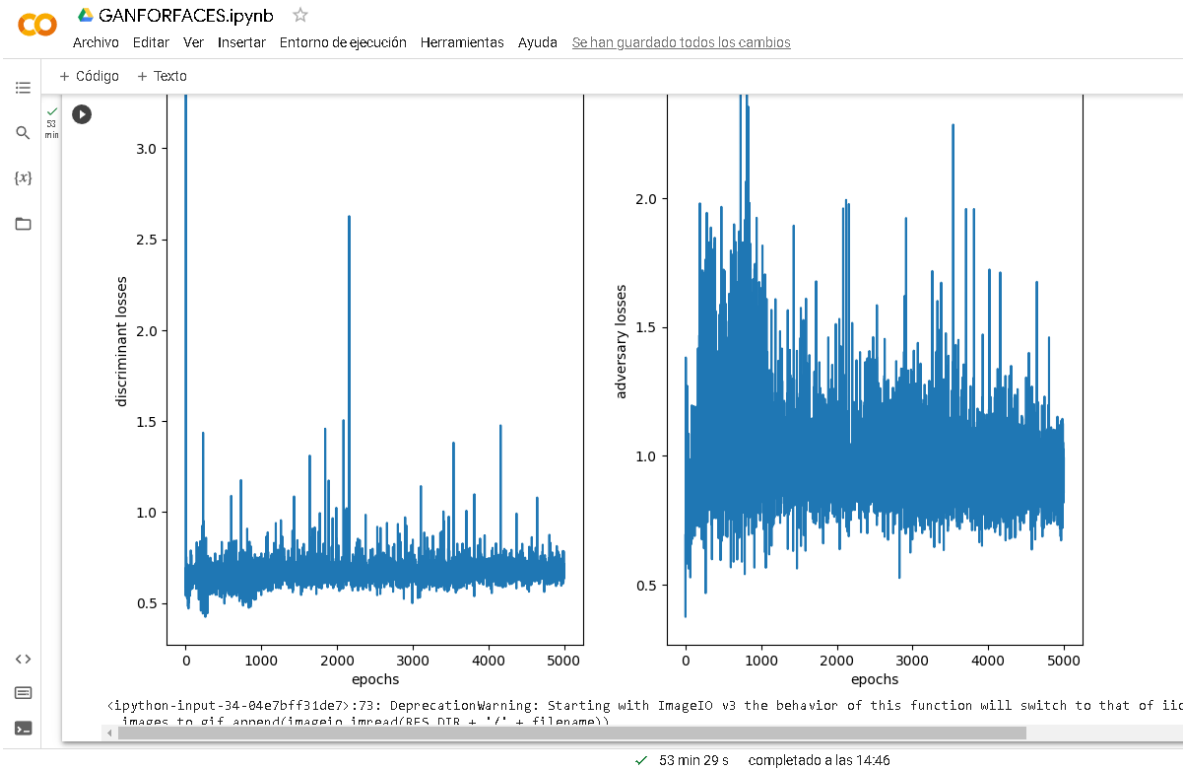
Better! We can see human faces, it is important to mention that we arrive here because of the epochs help so much, we can see human faces and it is a great step.

Experiment 11

SIZE	#IMAGES	#EPOCHS
64x64	15,000	500



Experient 11. Image 1. Epochs done.



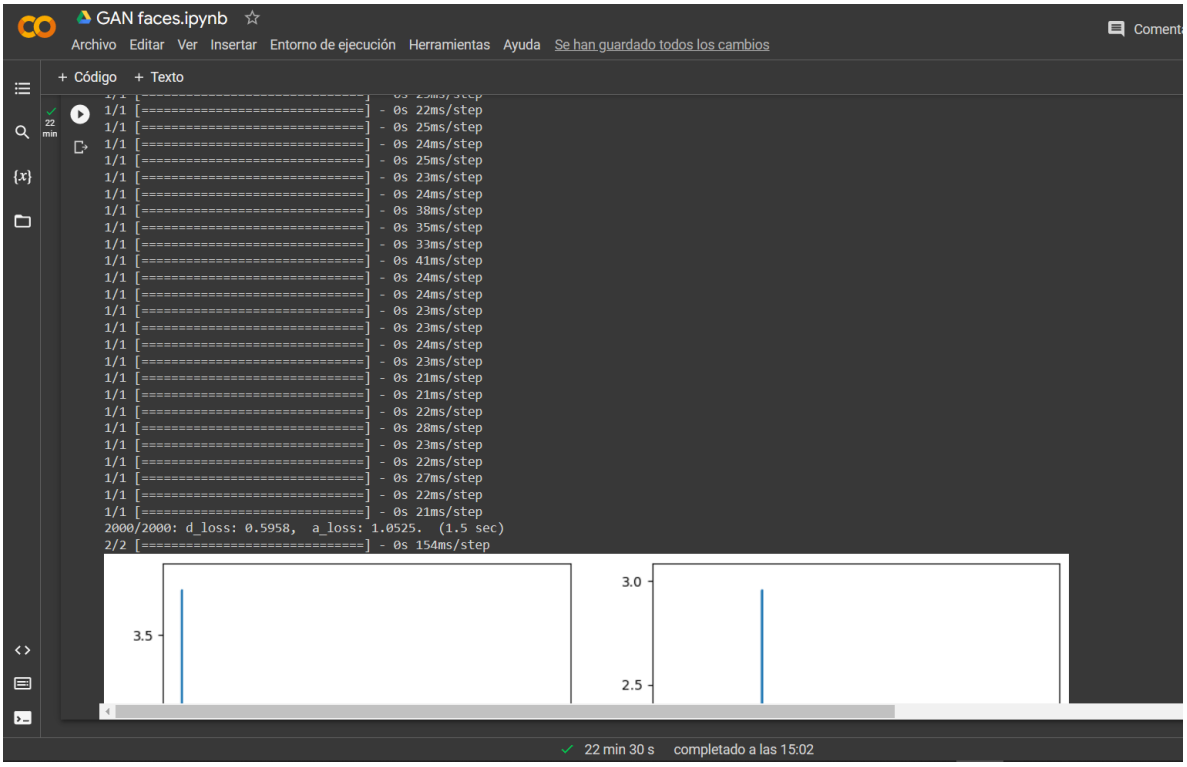
Experient 11. Image 2. Graphics.

Experient 11. Images 3-4. Epochs done.

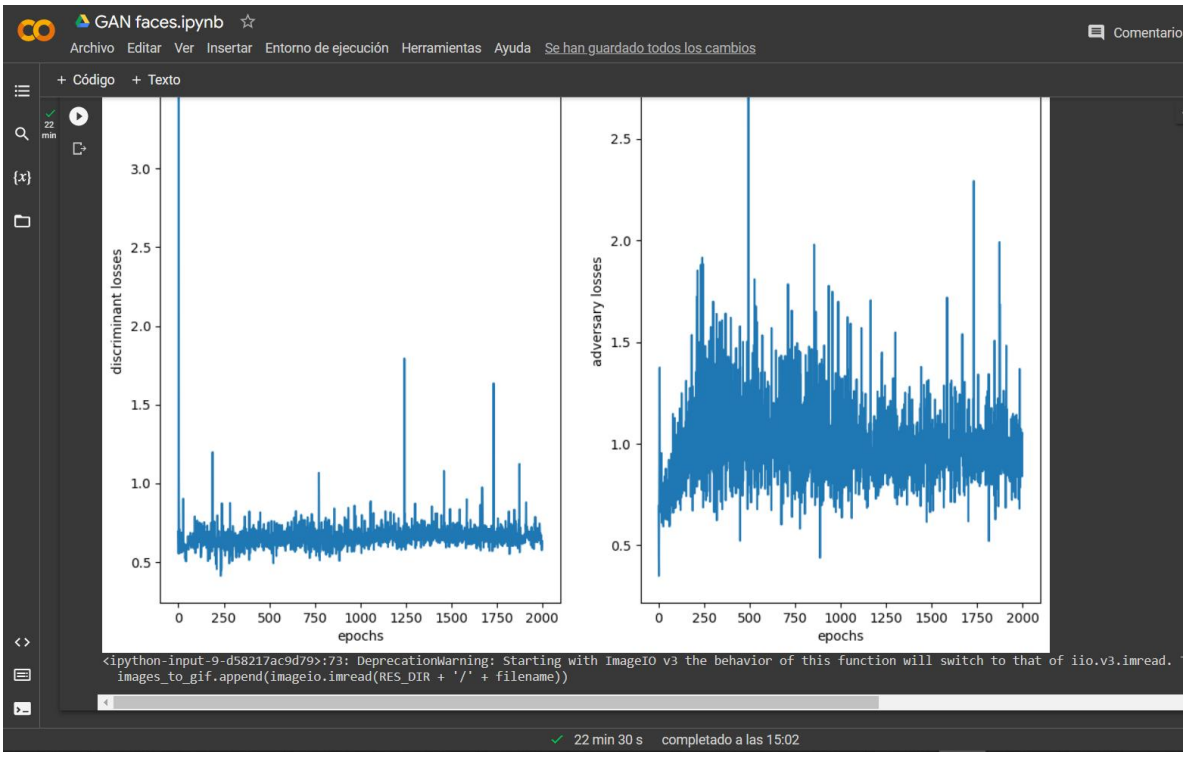
Like a gif, I put some screenshots of the gif file so you can see the faces in a great way, in the image 4 of the experiment 11 we can see amazing fake human faces.

Experiment 12

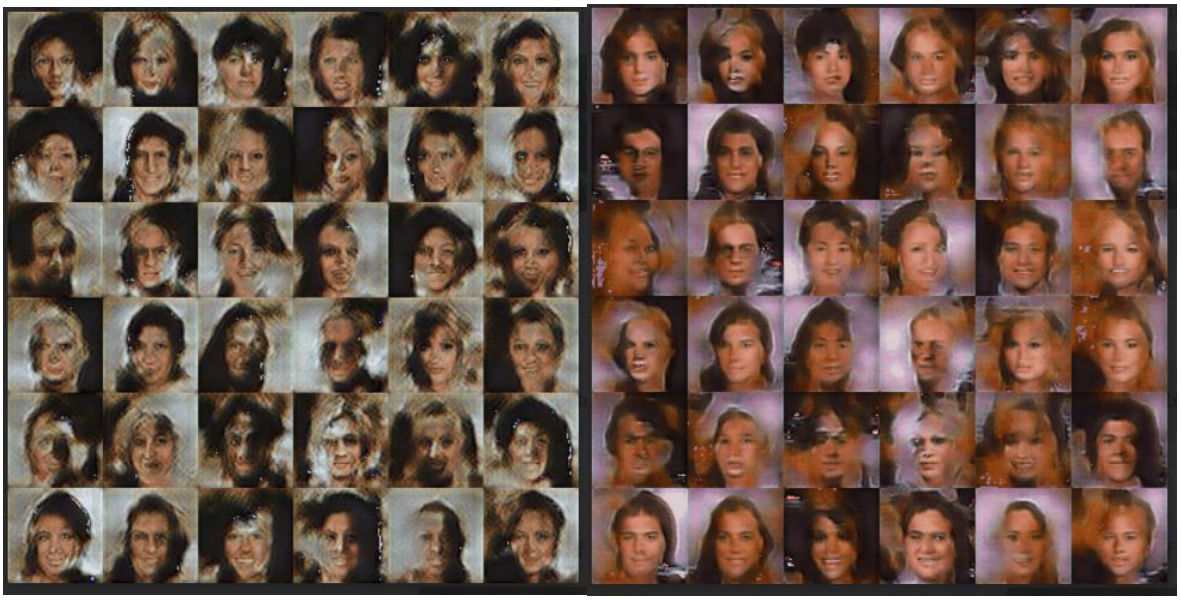
SIZE	#IMAGES	#EPOCHS
64x64	20,000	2,000



Experient 12. Image 1. Epochs done.



Experient 12. Image 2. Graphics.

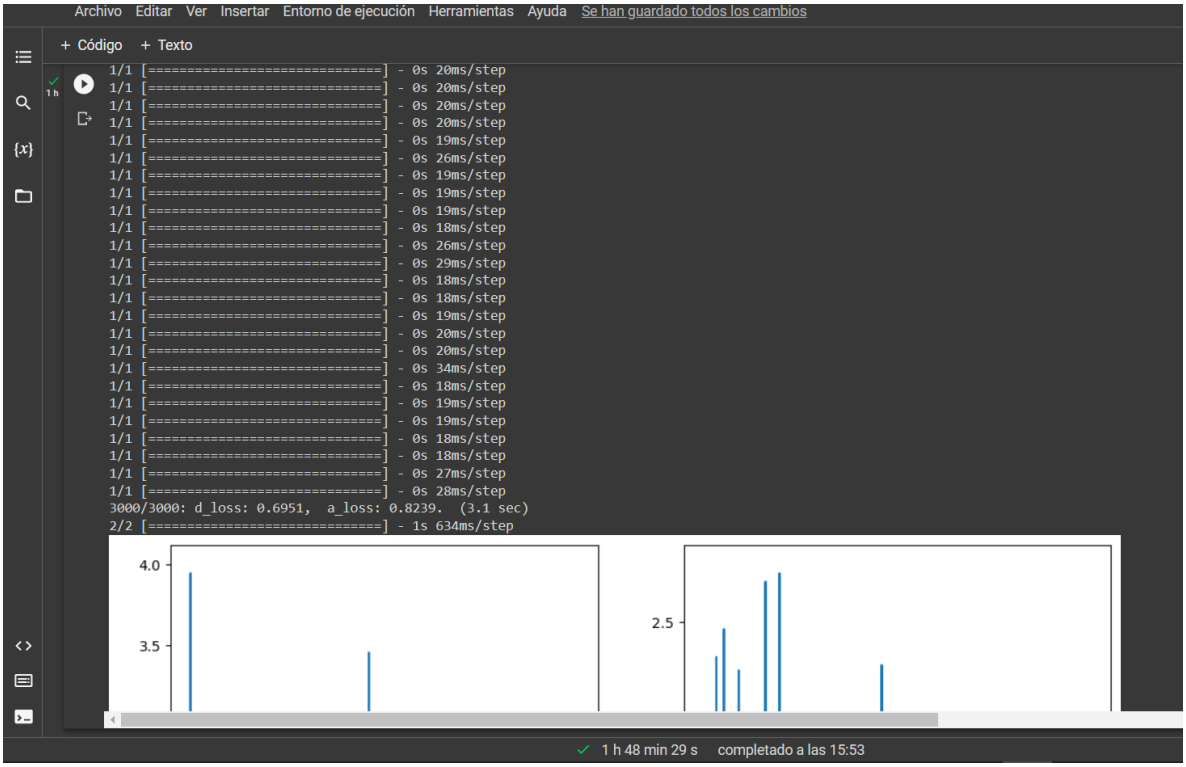


Experient 12. Image 3-4. Images Results.

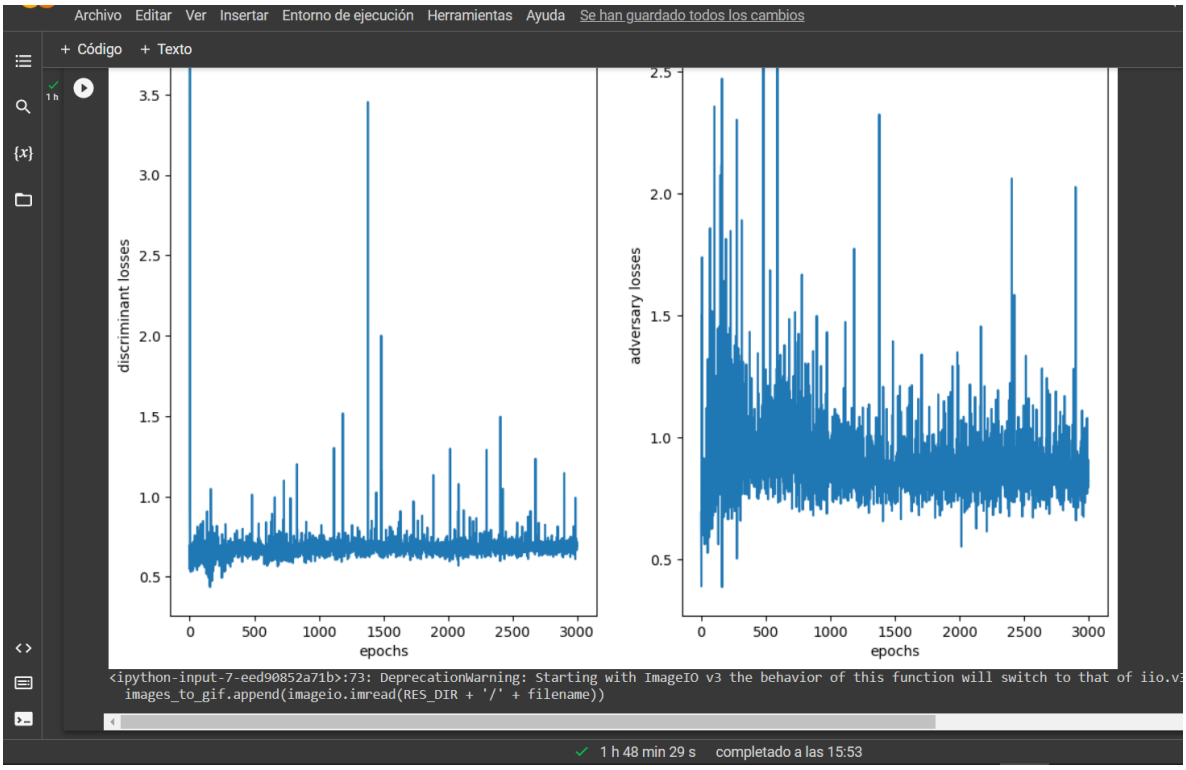
The results looks more real, is not enough but is a good grow.

Experiment 13

SIZE	#IMAGES	#EPOCHS
128x128	10,232	3,000

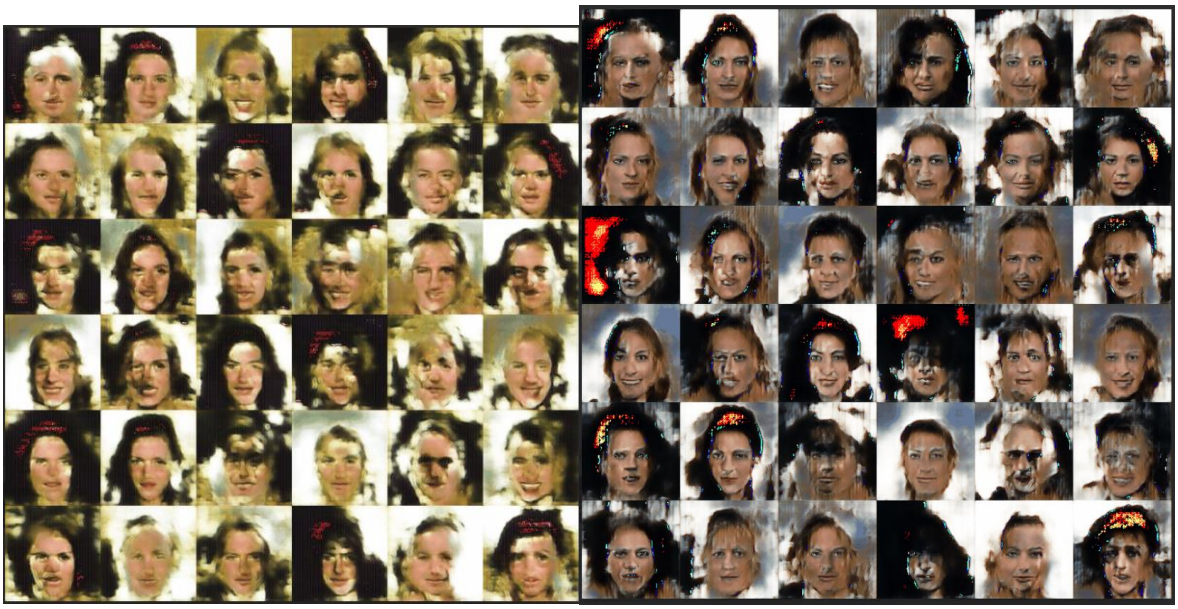


Experient 13. Image 1. Epochs done.



Experient 13. Image 2. Graphics.

Experient 13. Image 3-4. Images Results.



Experient 13. Image 5-6. Images Results.

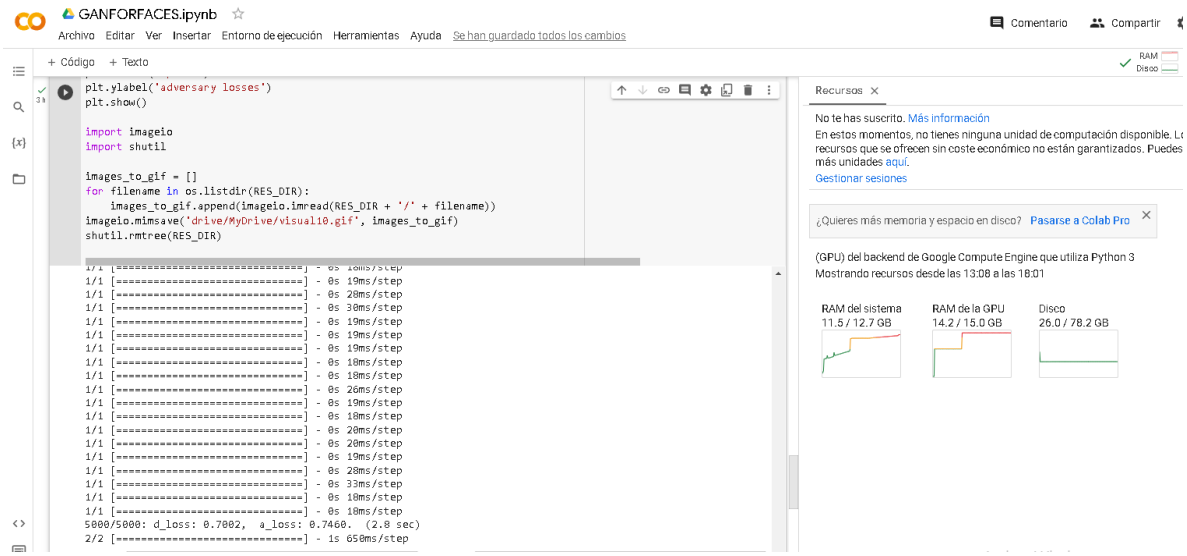
Experient 13. Image 7. Images Results.

Our goal is almost done, we can see incredible fake human faces and some of them looks realistic, the color, division, size and other characteristics like human factors looks great.

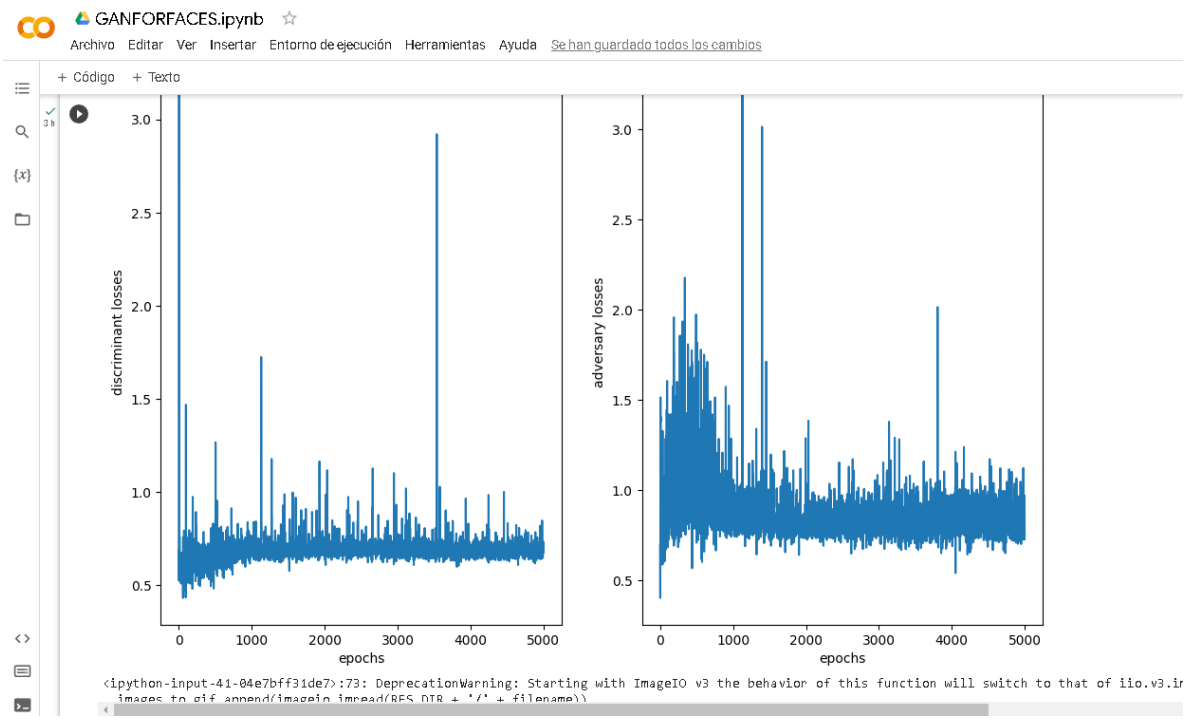
Finally our last experiment, in this part we can see the results of train, make modifications and more.

Experiment 14

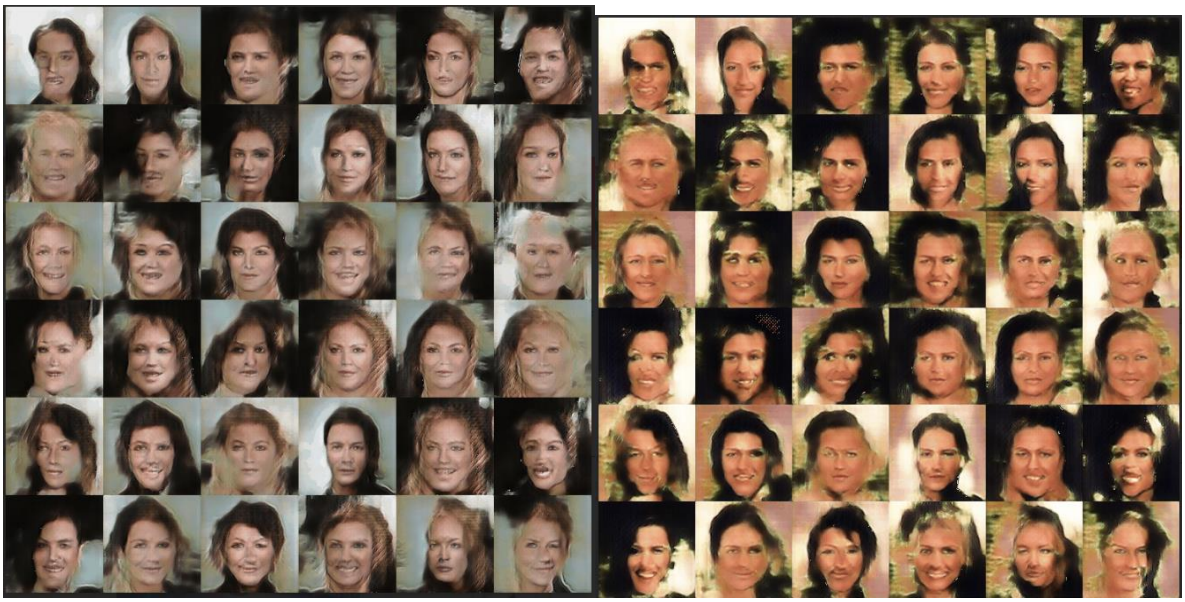
SIZE	#IMAGES	#EPOCHS
128x128	15,000	5,000



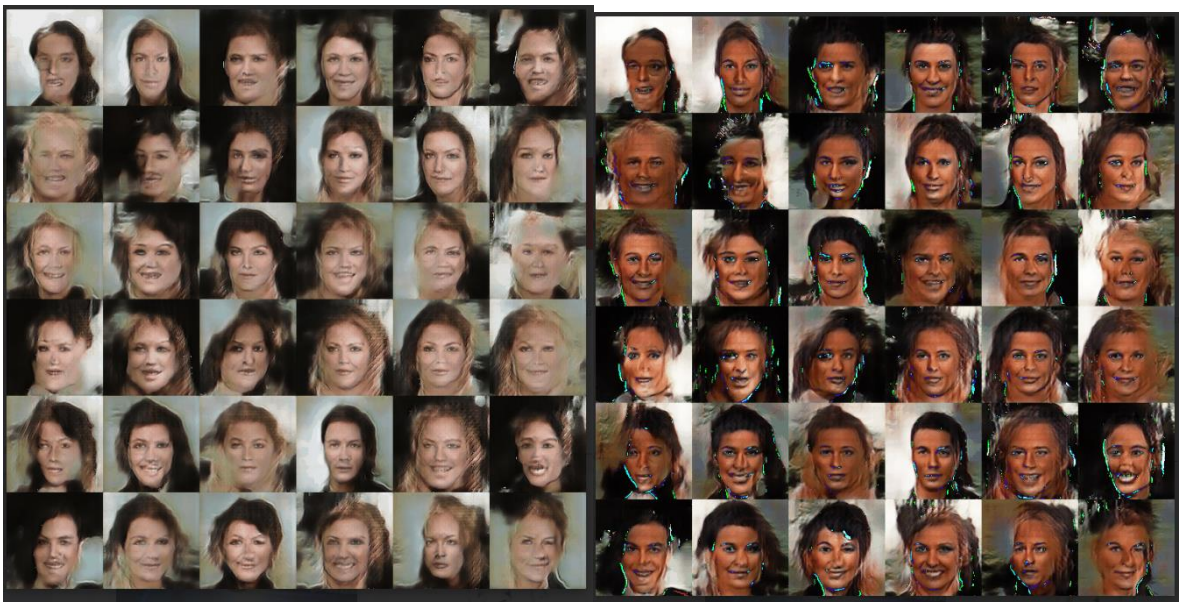
Experient 14. Image 1. Epochs done.



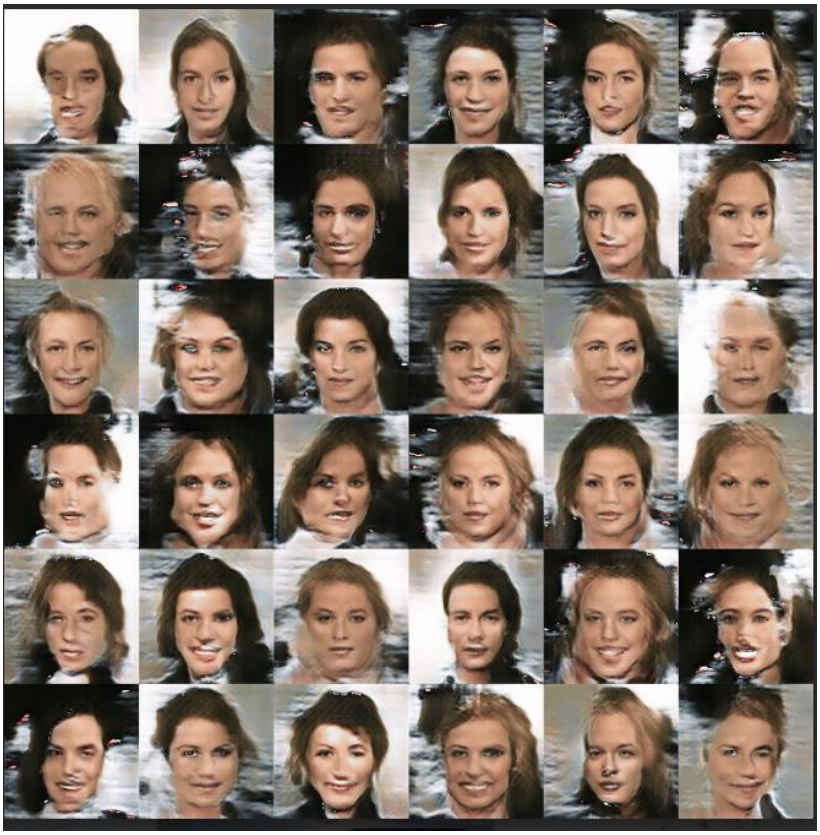
Experient 14. Image 2. Graphics.



Experient 14. Image 3-4. Images Results.



Experient 14. Image 5-6. Images Results.



Experient 14. Image 7. Images Results.

Nothing to say but satisfying, it is just satisfying see the results, after hours, days and modifications to make a fake human face. Let's see a comparative between the first experiment and the last one.

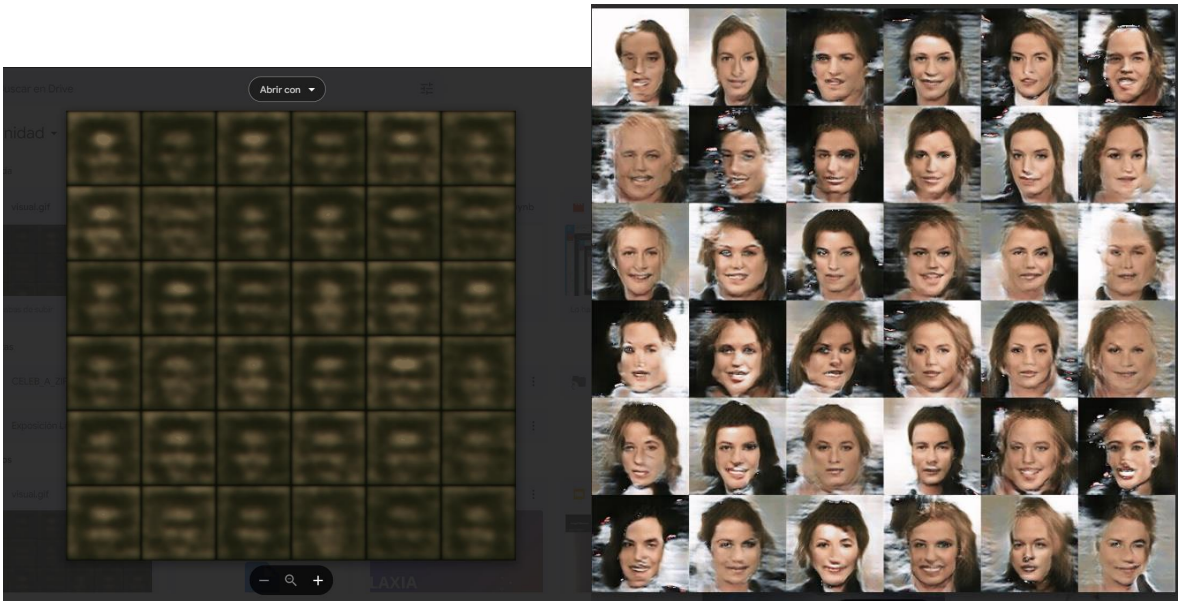


Image 1-2. Comparative first experiment and last

In conclusion, GANs have shown impressive results in generating realistic faces and other types of images. GANs have also been used for a wide range of applications beyond image generation. With ongoing research, GANs have the potential to advance the field of machine learning and open up new possibilities for generating synthetic data. After making all the experiment I enjoyed it, I feel happy to see something good at the end because at the beginning it looks like a human face never will happened, but actually yes, like a project and more than that was something fun work on it. I learned so much about GANs and the accuracy of the things that we are working on, so let's continuous working and make more amazing things.

References:

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., ... Bengio, Y. (2014). Generative adversarial networks. In Advances in neural information processing systems (pp. 2672-2680). Retrieved from https://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf

Radford, A., Metz, L., & Chintala, S. (2016). Unsupervised representation learning with deep convolutional generative adversarial networks. arXiv preprint arXiv:1511.06434. Retrieved from https://arxiv.org/abs/1511.06434

Zhang, Han, et al. "Self-attention generative adversarial networks." International Conference on Machine Learning. PMLR, 2019.

Karras, T., Aila, T., Laine, S., & Lehtinen, J. (2019). A style-based generator architecture for generative adversarial networks. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (pp. 4401-4410). Retrieved from https://openaccess.thecvf.com/content_CVPR_2019/papers/Karras_A_Style-Based_Generator_Architecture_for_Generative_Adversarial_Networks_CVPR_2019_paper.pdf

Arjovsky, M., Chintala, S., & Bottou, L. (2017). Wasserstein GAN. arXiv preprint arXiv:1701.07875. Retrieved from https://arxiv.org/abs/1701.07875v3

Balaji, A. V., & Chakraborty, S. (2021). Adversarial attacks on deep learning models: A comprehensive survey. Information Fusion, 75, 61-96. URL: https://www.sciencedirect.com/science/article/pii/S1566253520307724