Generating New Celebrities

In the present project we will build a Generative Adversarial Network (GAN) to obtain new faces from images of celebrities faces. The new generated faces have to seem real, eventhough they don't belong to any real person. Some questions will appear:

- Do the "new celebrities" look real?
- If so, might them resemble to a specific person?
- Will the generated faces present more trend to a specific kind of feature (for instance, asian-like eyes or blond hair)?

Let's check it out!

First, we will import all the modules and libraries that we need. Then, we will download the training data from the available Kaggle datasets (in folder 'celebrities-100k'). Then we have to buid a 'Discriminator' model and a 'Generator' model and train them.

```
!pip install jovian --upgrade --quiet
```

```
import os
import torch
import torchvision
import tarfile
import torch.nn as nn
import numpy as np
import torch.nn.functional as F
from torchvision.datasets.utils import download_url
from torchvision.datasets import ImageFolder
from torch.utils.data import DataLoader
import torchvision.transforms as tt
from torch.utils.data import random_split
from torchvision.utils import make_grid
import matplotlib
import matplotlib.pyplot as plt
%matplotlib inline

matplotlib.rcParams['figure.facecolor'] = '#ffffff'
```

```
project_name='new-celebrities'
```

```
!pip install opendatasets --upgrade --quiet
```

```
import opendatasets as od
```

```
od.download('https://www.kaggle.com/greg115/celebrities-100k')
```

```
  1%|          | 10.0M/828M [00:00<00:08, 103MB/s]
```

```
Downloading celebrities-100k.zip to ./celebrities-100k
100%|██████████| 828M/828M [00:04<00:00, 188MB/s]
```

```
import os
```

The images have a size of 128 x 128 px and I'll resize them to a size of 80 x 80 px and then crop the images to a size of 64 x 64 px, thus avoiding most of the background and obtaining a smaller size that is better for calculations (it's easier to have power of 2 sizes). Then I'll normalize the pixel values (between -1 and 1) with a standard deviation of 0.5 and a mean value of 0.5, as well.

```
from torch.utils.data import DataLoader
from torchvision.datasets import ImageFolder
import torchvision.transforms as T
```

```
image_size = 64
batch_size = 128
stats = (0.5, 0.5, 0.5), (0.5, 0.5, 0.5)
```

```
DATA_DIR = './celebrities-100k/100k'
train_ds = ImageFolder(DATA_DIR, transform=T.Compose([
    T.Resize(80),
    T.CenterCrop(image_size),
    T.ToTensor(),
    T.Normalize(*stats)]))
```

Now I will create the date loader.

```
train_dl = DataLoader(train_ds, batch_size, shuffle=True, num_workers=3, pin_memory=Tru
```

I will define some functions to denormalize the image tensors, in order to observe the images during the training process.

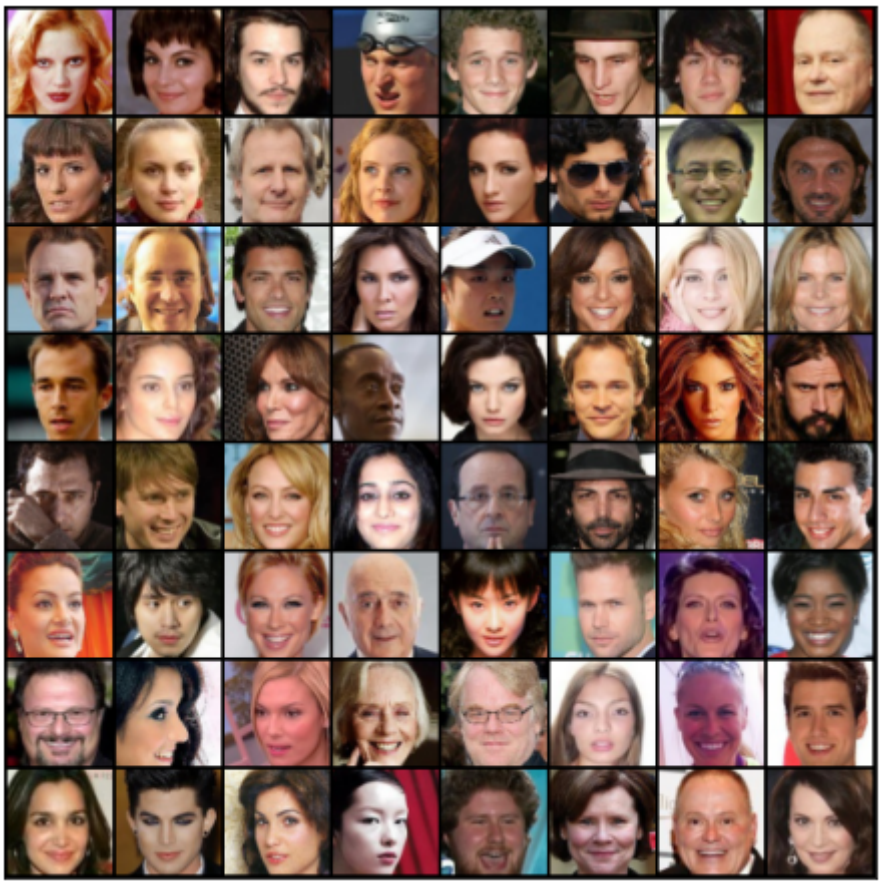
```
import torch
from torchvision.utils import make_grid
import matplotlib.pyplot as plt
%matplotlib inline
```

```
def denorm(img_tensors):
    return img_tensors * stats[1][0] + stats[0][0]
```

```
def show_images(images, nmax=64):
    fig, ax = plt.subplots(figsize=(8, 8))
    ax.set_xticks([]); ax.set_yticks([])
    ax.imshow(make_grid(denorm(images.detach()[:nmax]), nrow=8).permute(1, 2, 0))
```

```
def show_batch(dl, nmax=64):
    for images, _ in dl:
        show_images(images, nmax)
        break
```

```
show_batch(train_dl)
```



```
jovian.commit(project=project_name, environment=None)
```

Preparing the GPU

We define some helper functions to take the data to the GPU.

```
def get_default_device():
    """Pick GPU if available, else CPU"""
    if torch.cuda.is_available():
        return torch.device('cuda')
    else:
        return torch.device('cpu')

def to_device(data, device):
    """Move tensor(s) to chosen device"""
    if isinstance(data, (list,tuple)):
        return [to_device(x, device) for x in data]
    return data.to(device, non_blocking=True)
```

```
class DeviceDataLoader():
    """Wrap a dataloader to move data to a device"""
    def __init__(self, dl, device):
        self.dl = dl
        self.device = device

    def __iter__(self):
        """Yield a batch of data after moving it to device"""
        for b in self.dl:
            yield to_device(b, self.device)

    def __len__(self):
        """Number of batches"""
        return len(self.dl)
```

Let's check if we have a working GPU.

```
device = get_default_device()
device
```

```
device(type='cuda')
```

As we have a 'cuda' type device, this means that our GPU is available, so we can move our data to the GPU.

```
train_dl = DeviceDataLoader(train_dl, device)
```

Building the discriminator

Building the discriminator is not so difficult, it's made by a sequence of convolutional layers plus the corresponding normalization and activation functions. In this step, we could try different types of activation functions and we could try linear layers instead of convolutional ones. In first place I will start with a sequence as that used in Lesson 6 of the course, but I will try changing the activation functions or the layers if I'm not convinced with the obtained result.

```
import torch.nn as nn
```

```
discriminator = nn.Sequential(
    # in: 3 x 64 x 64

    nn.Conv2d(3, 64, kernel_size=4, stride=2, padding=1, bias=False),
    nn.BatchNorm2d(64),
    nn.LeakyReLU(0.2, inplace=True),
    # out: 64 x 32 x 32

    nn.Conv2d(64, 128, kernel_size=4, stride=2, padding=1, bias=False),
    nn.BatchNorm2d(128),
    nn.LeakyReLU(0.2, inplace=True),
    # out: 128 x 16 x 16

    nn.Conv2d(128, 256, kernel_size=4, stride=2, padding=1, bias=False),
```

```
    nn.BatchNorm2d(256),
    nn.LeakyReLU(0.2, inplace=True),
    # out: 256 x 8 x 8

    nn.Conv2d(256, 512, kernel_size=4, stride=2, padding=1, bias=False),
    nn.BatchNorm2d(512),
    nn.LeakyReLU(0.2, inplace=True),
    # out: 512 x 4 x 4

    nn.Conv2d(512, 1, kernel_size=4, stride=1, padding=0, bias=False),
    # out: 1 x 1 x 1

    nn.Flatten(),
    nn.Sigmoid())
```

As the discriminator is a binary classification model, the result will be between 0 and 1, showing the probability of having a real image (result close or equal to 1) or a fake image (close or equal to 0). Let's move the discriminator model to the GPU.

```
discriminator = to_device(discriminator, device)
```

Creating the Generator

This step is more difficult to perform than the previous one beacause the Generator model needs to use the Discriminator to work properly. The Generator creates images from the so called 'latent tensors'. To convert the latent tensors to images we can make use of a sequence of 'transposed convolution' layers (or 'deconvolutions'). Finally we can convert the obtained values to values between -1 and 1 applying the TanH function, so the values of the obtained tensor are in the same range as the values of the training image tensors.

The generator is fed with 'latent tensors' of the size: 128 x 1 x 1, filled with random values. Then it has to convert the tensor to a 3 x 64 x 64 image tensor.

```
latent_size = 128
```

```
generator = nn.Sequential(
    # in: latent_size x 1 x 1

    nn.ConvTranspose2d(latent_size, 512, kernel_size=4, stride=1, padding=0, bias=False
    nn.BatchNorm2d(512),
    nn.ReLU(True),
    # out: 512 x 4 x 4

    nn.ConvTranspose2d(512, 256, kernel_size=4, stride=2, padding=1, bias=False),
    nn.BatchNorm2d(256),
    nn.ReLU(True),
    # out: 256 x 8 x 8

    nn.ConvTranspose2d(256, 128, kernel_size=4, stride=2, padding=1, bias=False),
    nn.BatchNorm2d(128),
    nn.ReLU(True),
```

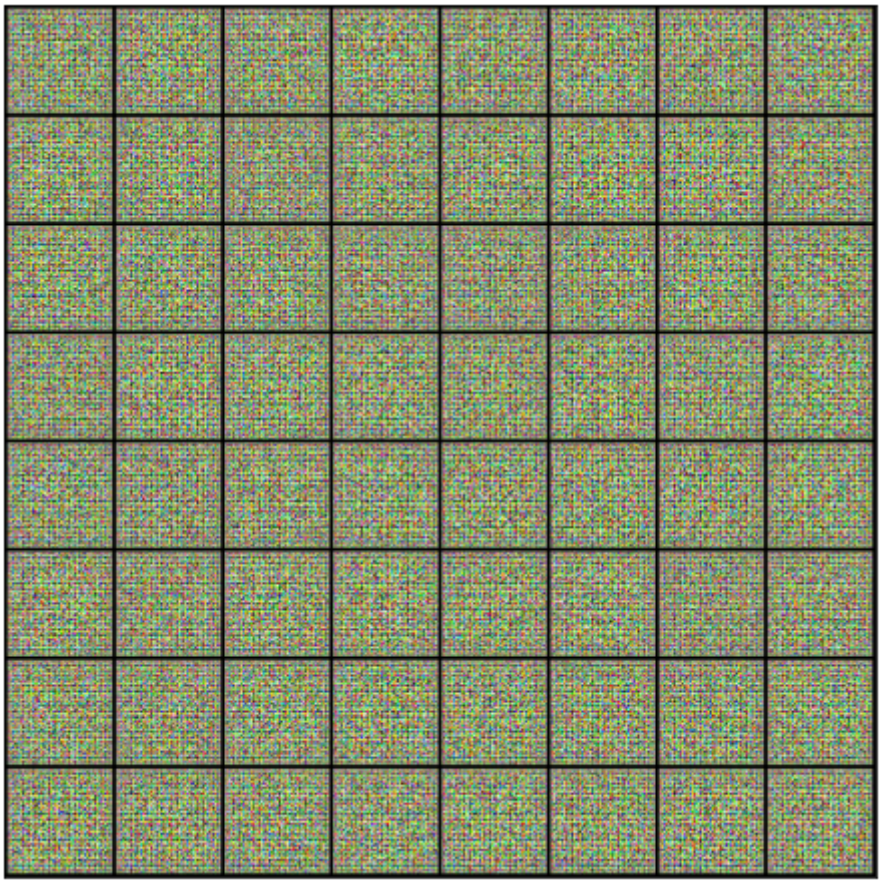
```
    # out: 128 x 16 x 16

    nn.ConvTranspose2d(128, 64, kernel_size=4, stride=2, padding=1, bias=False),
    nn.BatchNorm2d(64),
    nn.ReLU(True),
    # out: 64 x 32 x 32

    nn.ConvTranspose2d(64, 3, kernel_size=4, stride=2, padding=1, bias=False),
    nn.Tanh()
    # out: 3 x 64 x 64
)
```

```
xb = torch.randn(batch_size, latent_size, 1, 1) # random latent tensors
fake_images = generator(xb)
print(fake_images.shape)
show_images(fake_images)
```

```
torch.Size([128, 3, 64, 64])
```



```
generator = to_device(generator, device)
```

Training the Discriminator Model

The discriminator model has to be prepared to distinguish between a real image and an artificially generated image. For this purpose, we define a function that trains the discriminator. As the discriminator is a classification model, one of the best choices for the losss function is the cross entropy, but in this case as we have a 'binary' classification model we should apply the 'binary cross entropy' function as our loss function instead of the regular

cross entropy function. Also, we have to set to 1 the target when we are feeding the discriminator with real images and 0 when the input images are not real, that's why we use the torch.ones and torch.zeros functions for the respective targets. The 'real_loss' will be computed as the binary cross entropy between results obtained by the discriminator when applied to real images (real_preds) and the set target for this kind of images (that is 1). The same procedure is applied to generated or 'fake' images. The fake images for training are generated randomly from latent tensors. Thus the real_loss must be small for real images and the fake_loss will be small for fake images. Then we compute the gradient descent to the overall loss (the addition of both losses). The optimizer for the discriminator (opt_d) only affects the weights in the discriminator model and doesn't have any effect in the generator model.

```
def train_discriminator(real_images, opt_d):
    # Clear discriminator gradients
    opt_d.zero_grad()

    # Pass real images through discriminator
    real_preds = discriminator(real_images)
    real_targets = torch.ones(real_images.size(0), 1, device=device)
    real_loss = F.binary_cross_entropy(real_preds, real_targets)
    real_score = torch.mean(real_preds).item()

    # Generate fake images
    latent = torch.randn(batch_size, latent_size, 1, 1, device=device)
    fake_images = generator(latent)

    # Pass fake images through discriminator
    fake_targets = torch.zeros(fake_images.size(0), 1, device=device)
    fake_preds = discriminator(fake_images)
    fake_loss = F.binary_cross_entropy(fake_preds, fake_targets)
    fake_score = torch.mean(fake_preds).item()

    # Update discriminator weights
    loss = real_loss + fake_loss
    loss.backward()
    opt_d.step()
    return loss.item(), real_score, fake_score
```

We have added some parameters to keep track of the evolution of the training: real_score, fake_score which are, respectively, the mean values of the real and fake predictions over each image batch and the loss.

```
jovian.commit(project=project_name, environment=None)
```

```
[jovian] Detected Colab notebook...
[jovian] Uploading colab notebook to Jovian...
[jovian] Attaching records (metrics, hyperparameters, dataset etc.)
[jovian] Committed successfully! https://jovian.ai/viquiriglos/new-celebrities
```

```
'https://jovian.ai/viquiriglos/new-celebrities'
```

Training the Generator model

This is a harder task than training the discriminator because it envolves the presence of the Discriminator. So, only after the discriminator is trained we can train the Generator. The objective of this training is to make the discriminator 'believe' that the output from the Generator is a real image, so the generator must generate each time images that resemble to those from real datasets. The goal is to make the discriminator produce a value close to 1 for the fake images, so this will be the target value for the training.

```
def train_generator(opt_g):
    # Clear generator gradients
    opt_g.zero_grad()

    # Generate fake images
    latent = torch.randn(batch_size, latent_size, 1, 1, device=device)
    fake_images = generator(latent)

    # Try to fool the discriminator
    preds = discriminator(fake_images)
    targets = torch.ones(batch_size, 1, device=device)
    loss = F.binary_cross_entropy(preds, targets)

    # Update generator weights
    loss.backward()
    opt_g.step()

    return loss.item()
```

```
jovian.commit(project=project_name, environment=None)
```

```
[jovian] Detected Colab notebook...
[jovian] Please enter your API key ( from https://jovian.ai/ ):
API KEY: ··········
[jovian] Uploading colab notebook to Jovian...
[jovian] Committed successfully! https://jovian.ai/viquiriglos/new-celebrities
```

```
'https://jovian.ai/viquiriglos/new-celebrities'
```

Configuring the Output images and tracking the evolution of the model

We will create a directory where we can save intermediate outputs from the generator to visually inspect the progress of the model. We'll also create a helper function to export the generated images.

```
from torchvision.utils import save_image
```

```
sample_dir = 'generated'
os.makedirs(sample_dir, exist_ok=True)
```

```
def save_samples(index, latent_tensors, show=True):
    fake_images = generator(latent_tensors)
```

```
    fake_fname = 'generated-images-{0:0=4d}.png'.format(index)
    save_image(denorm(fake_images), os.path.join(sample_dir, fake_fname), nrow=8)
    print('Saving', fake_fname)
    if show:
        fig, ax = plt.subplots(figsize=(8, 8))
        ax.set_xticks([]); ax.set_yticks([])
        ax.imshow(make_grid(fake_images.cpu().detach(), nrow=8).permute(1, 2, 0))
```

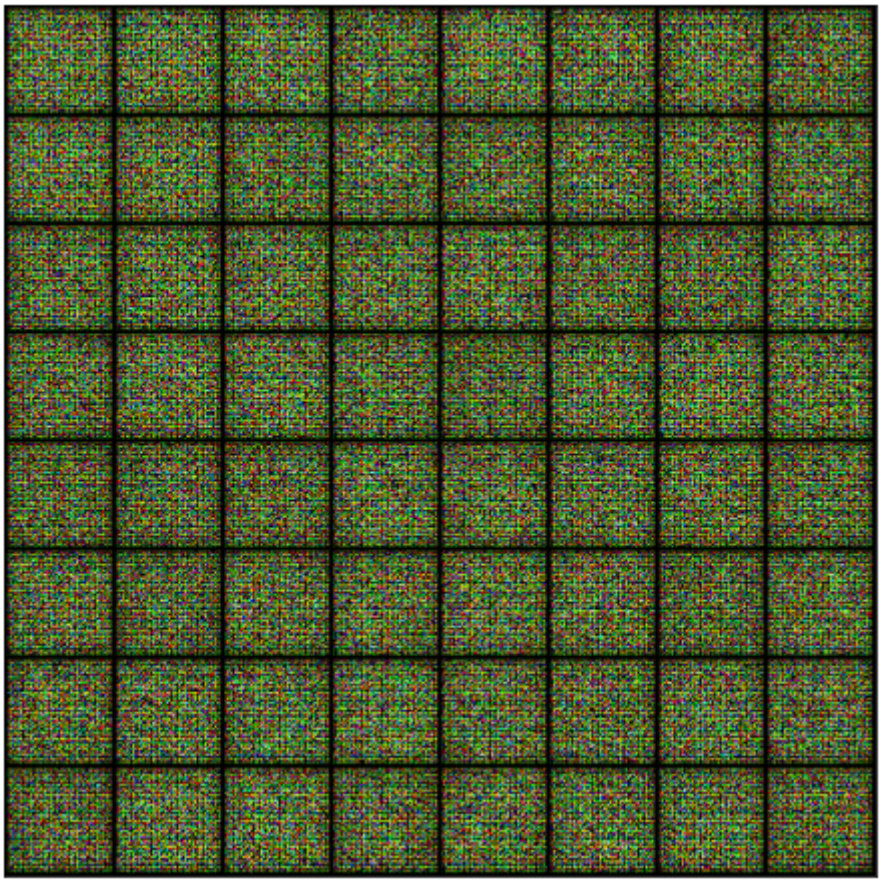
To check is these functions work properly, we will prepare a set of 64 latent vectors, transform them into fake images and save them.

```
fixed_latent = torch.randn(64, latent_size, 1, 1, device=device)
```

```
save_samples(0, fixed_latent)
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
```

```
Saving generated-images-0000.png
```



```
jovian.commit(project=project_name, environment=None)
```

Training Loop

We will generate a training function which includes both the discriminator training and the generator training for each batch. We'll also save the data in each training step. The chosen optimizer for both models is the Adam. We can play around with the beta parameters to improve the performance.

```
from tqdm.notebook import tqdm
import torch.nn.functional as F
```

```
def fit(epochs, lr, start_idx=1):
    torch.cuda.empty_cache()

    # Losses & scores
    losses_g = []
    losses_d = []
    real_scores = []
    fake_scores = []

    # Create optimizers
    opt_d = torch.optim.Adam(discriminator.parameters(), lr=lr, betas=(0.5, 0.999))
    opt_g = torch.optim.Adam(generator.parameters(), lr=lr, betas=(0.5, 0.999))

    for epoch in range(epochs):
        for real_images, _ in tqdm(train_dl):
            # Train discriminator
            loss_d, real_score, fake_score = train_discriminator(real_images, opt_d)
            # Train generator
            loss_g = train_generator(opt_g)

        # Record losses & scores
        losses_g.append(loss_g)
        losses_d.append(loss_d)
        real_scores.append(real_score)
        fake_scores.append(fake_score)

        # Log losses & scores (last batch)
        print("Epoch [{}/{}], loss_g: {:.4f}, loss_d: {:.4f}, real_score: {:.4f}, fake_
            epoch+1, epochs, loss_g, loss_d, real_score, fake_score))

        # Save generated images
        save_samples(epoch+start_idx, fixed_latent, show=False)

    return losses_g, losses_d, real_scores, fake_scores
```

We will now train our GAN trying different learning rates and epochs and save the corresponding hyperparameters to jovian.

```
lr = 0.001
epochs = 25
```

```
jovian.reset()
jovian.log_hyperparams(lr=lr, epochs=epochs)
```

```
[jovian] Hyperparams logged.
```

```
history = fit(epochs, lr)
```

```
HBox(children=(FloatProgress(value=0.0, max=782.0), HTML(value='')))

 Epoch [1/25], loss_g: 4.7989, loss_d: 0.1946, real_score: 0.8829, fake_score: 0.0499
 Saving generated-images-0001.png
HBox(children=(FloatProgress(value=0.0, max=782.0), HTML(value='')))

 Epoch [2/25], loss_g: 1.7554, loss_d: 2.2669, real_score: 0.2113, fake_score: 0.0004
 Saving generated-images-0002.png
HBox(children=(FloatProgress(value=0.0, max=782.0), HTML(value='')))

 Epoch [3/25], loss_g: 6.2196, loss_d: 0.8448, real_score: 0.5567, fake_score: 0.0005
 Saving generated-images-0003.png
HBox(children=(FloatProgress(value=0.0, max=782.0), HTML(value='')))

 Epoch [4/25], loss_g: 6.1711, loss_d: 0.5238, real_score: 0.9210, fake_score: 0.2931
 Saving generated-images-0004.png
HBox(children=(FloatProgress(value=0.0, max=782.0), HTML(value='')))

 Epoch [5/25], loss_g: 3.2823, loss_d: 0.5063, real_score: 0.7820, fake_score: 0.0760
 Saving generated-images-0005.png
HBox(children=(FloatProgress(value=0.0, max=782.0), HTML(value='')))

 Epoch [6/25], loss_g: 5.3980, loss_d: 0.1232, real_score: 0.9497, fake_score: 0.0616
 Saving generated-images-0006.png
HBox(children=(FloatProgress(value=0.0, max=782.0), HTML(value='')))

 Epoch [7/25], loss_g: 2.7621, loss_d: 0.4695, real_score: 0.6916, fake_score: 0.0277
 Saving generated-images-0007.png
HBox(children=(FloatProgress(value=0.0, max=782.0), HTML(value='')))

 Epoch [8/25], loss_g: 3.1133, loss_d: 0.2309, real_score: 0.8633, fake_score: 0.0494
 Saving generated-images-0008.png
HBox(children=(FloatProgress(value=0.0, max=782.0), HTML(value='')))

 Epoch [9/25], loss_g: 6.0768, loss_d: 0.5601, real_score: 0.9427, fake_score: 0.3215
 Saving generated-images-0009.png
HBox(children=(FloatProgress(value=0.0, max=782.0), HTML(value='')))

 Epoch [10/25], loss_g: 2.9225, loss_d: 0.4375, real_score: 0.7434, fake_score: 0.0613
```

```
 Saving generated-images-0010.png
HBox(children=(FloatProgress(value=0.0, max=782.0), HTML(value='')))

 Epoch [11/25], loss_g: 3.7090, loss_d: 0.3812, real_score: 0.8080, fake_score: 0.0680
 Saving generated-images-0011.png
HBox(children=(FloatProgress(value=0.0, max=782.0), HTML(value='')))

 Epoch [12/25], loss_g: 5.0462, loss_d: 0.4550, real_score: 0.9705, fake_score: 0.2819
 Saving generated-images-0012.png
HBox(children=(FloatProgress(value=0.0, max=782.0), HTML(value='')))

 Epoch [13/25], loss_g: 7.9731, loss_d: 2.1820, real_score: 0.9892, fake_score: 0.7191
 Saving generated-images-0013.png
HBox(children=(FloatProgress(value=0.0, max=782.0), HTML(value='')))

 Epoch [14/25], loss_g: 6.5359, loss_d: 0.5085, real_score: 0.9703, fake_score: 0.3049
 Saving generated-images-0014.png
HBox(children=(FloatProgress(value=0.0, max=782.0), HTML(value='')))

 Epoch [15/25], loss_g: 5.9964, loss_d: 0.2750, real_score: 0.9197, fake_score: 0.1279
 Saving generated-images-0015.png
HBox(children=(FloatProgress(value=0.0, max=782.0), HTML(value='')))

 Epoch [16/25], loss_g: 4.1773, loss_d: 0.3378, real_score: 0.9104, fake_score: 0.1702
 Saving generated-images-0016.png
HBox(children=(FloatProgress(value=0.0, max=782.0), HTML(value='')))

 Epoch [17/25], loss_g: 5.6675, loss_d: 0.3646, real_score: 0.9837, fake_score: 0.2347
 Saving generated-images-0017.png
HBox(children=(FloatProgress(value=0.0, max=782.0), HTML(value='')))

 Epoch [18/25], loss_g: 2.9681, loss_d: 0.2587, real_score: 0.8618, fake_score: 0.0636
 Saving generated-images-0018.png
HBox(children=(FloatProgress(value=0.0, max=782.0), HTML(value='')))

 Epoch [19/25], loss_g: 4.6578, loss_d: 0.1939, real_score: 0.8911, fake_score: 0.0443
 Saving generated-images-0019.png
HBox(children=(FloatProgress(value=0.0, max=782.0), HTML(value='')))

 Epoch [20/25], loss_g: 4.4747, loss_d: 0.7615, real_score: 0.7673, fake_score: 0.1871
 Saving generated-images-0020.png
```

```
HBox(children=(FloatProgress(value=0.0, max=782.0), HTML(value='')))

 Epoch [21/25], loss_g: 4.5599, loss_d: 0.2344, real_score: 0.9487, fake_score: 0.1427
 Saving generated-images-0021.png
HBox(children=(FloatProgress(value=0.0, max=782.0), HTML(value='')))

 Epoch [22/25], loss_g: 2.6194, loss_d: 0.2409, real_score: 0.8587, fake_score: 0.0595
 Saving generated-images-0022.png
HBox(children=(FloatProgress(value=0.0, max=782.0), HTML(value='')))

 Epoch [23/25], loss_g: 4.7554, loss_d: 0.2202, real_score: 0.8771, fake_score: 0.0554
 Saving generated-images-0023.png
HBox(children=(FloatProgress(value=0.0, max=782.0), HTML(value='')))

 Epoch [24/25], loss_g: 2.2241, loss_d: 0.2652, real_score: 0.8554, fake_score: 0.0385
 Saving generated-images-0024.png
HBox(children=(FloatProgress(value=0.0, max=782.0), HTML(value='')))

 Epoch [25/25], loss_g: 3.8751, loss_d: 0.6666, real_score: 0.7686, fake_score: 0.1364
 Saving generated-images-0025.png
```

```
losses_g, losses_d, real_scores, fake_scores = history
```

```
jovian.log_metrics(loss_g=losses_g[-1],
                   loss_d=losses_d[-1],
                   real_score=real_scores[-1],
                   fake_score=fake_scores[-1])
```

```
[jovian] Metrics logged.
```

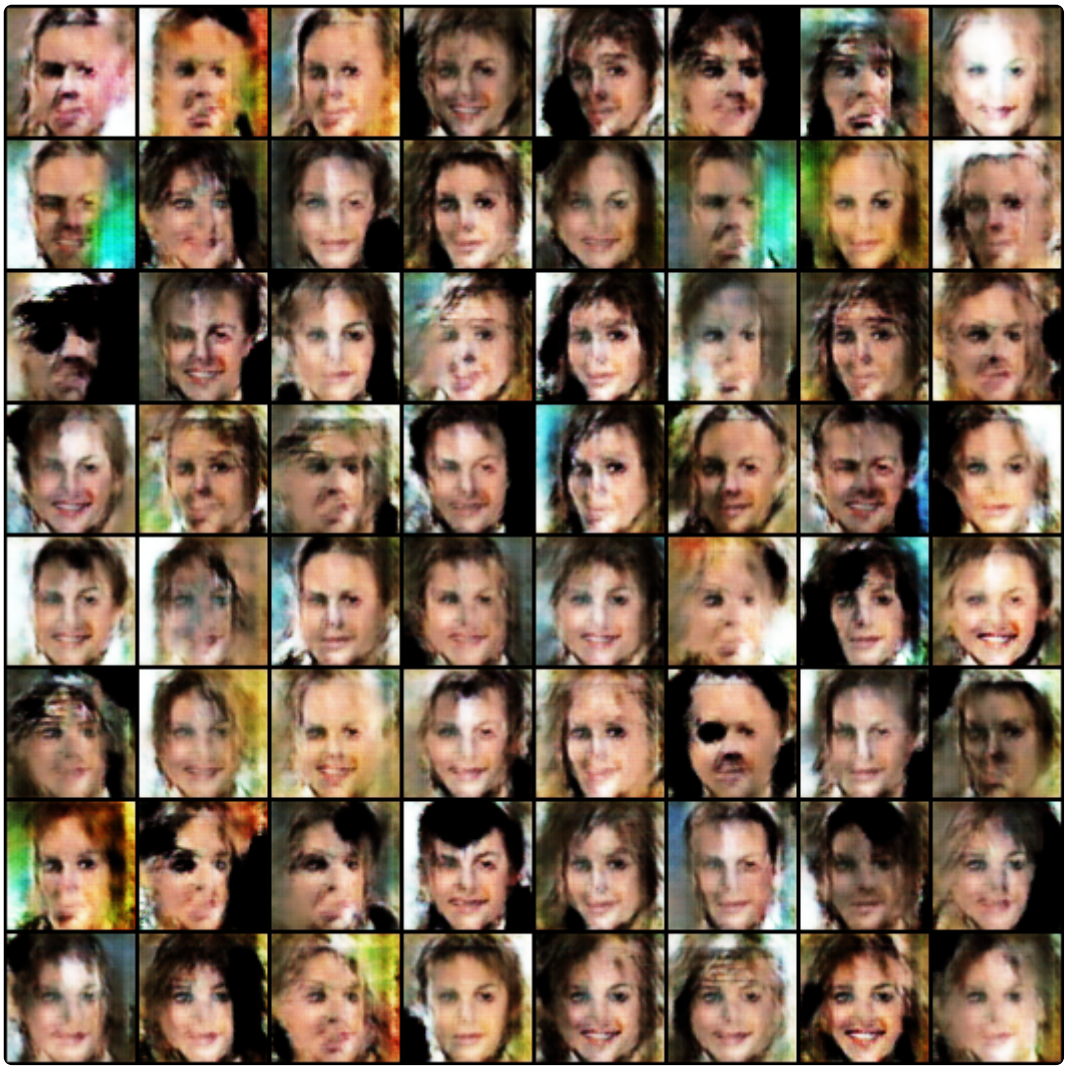
We must also save the checkpoints:

```
# Save the model checkpoints
torch.save(generator.state_dict(), 'G.pth')
torch.save(discriminator.state_dict(), 'D.pth')
```

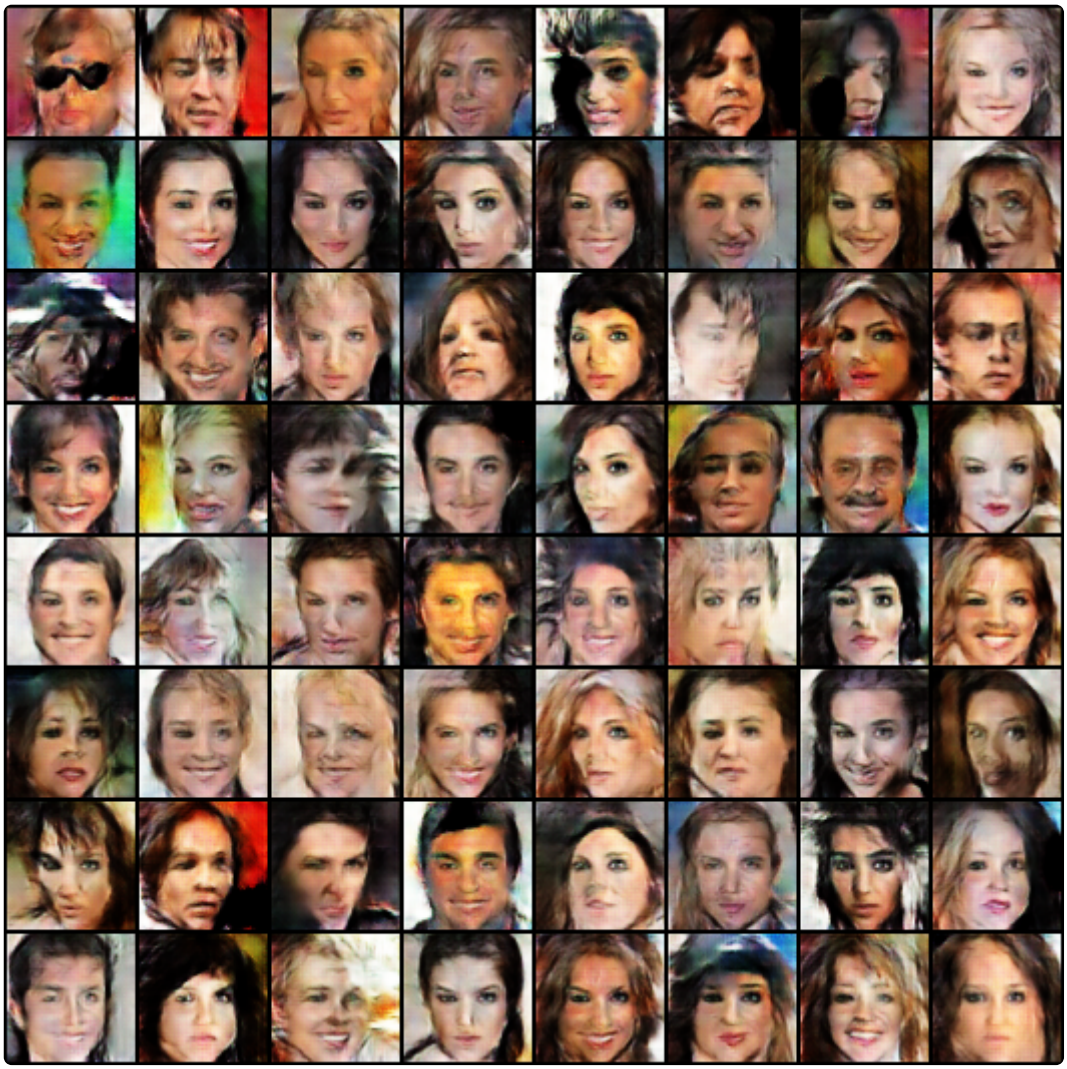
To display some the intermediate generated images we'll do as follows:

```
from IPython.display import Image
```

```
Image('./generated/generated-images-0001.png')
```



```
Image('./generated/generated-images-0005.png')
```



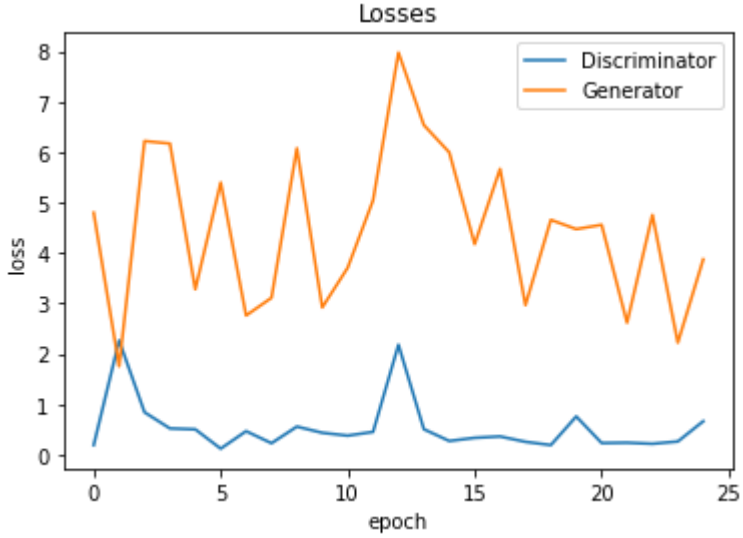
```
Image('./generated/generated-images-0010.png')
```

```
Image('./generated/generated-images-0020.png')
```

```
Image('./generated/generated-images-0025.png')
```

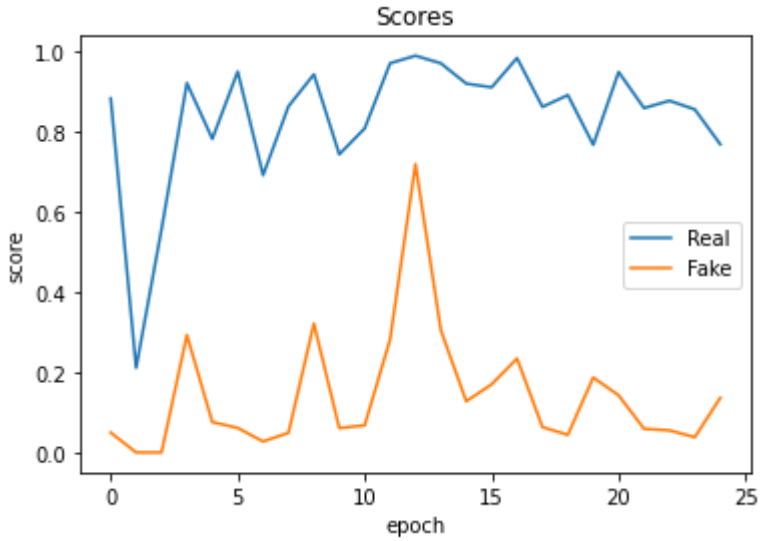
To analyze in a more accurate way if our model is improving or not, we will plot the losses of the discriminator and the generator as a function of the epochs. A good model would keep the discriminator loss close to zero while the generator loss should be decreasing as the number of epochs increase.

```
plt.plot(losses_d, '-')
plt.plot(losses_g, '-')
plt.xlabel('epoch')
plt.ylabel('loss')
plt.legend(['Discriminator', 'Generator'])
plt.title('Losses');
```



In the same way as we study the losses, we can analyze the 'scores'. The score should be close to 1 if the input is a real image and close to 0 if the image is fake, beacuse we've set this values when training the discriminator.

```
plt.plot(real_scores, '-')
plt.plot(fake_scores, '-')
plt.xlabel('epoch')
plt.ylabel('score')
plt.legend(['Real', 'Fake'])
plt.title('Scores');
```



We can also create a video using the generated images:

```
import cv2
import os

vid_fname = 'gans_training.avi'

files = [os.path.join(sample_dir, f) for f in os.listdir(sample_dir) if 'generated' in
files.sort()

out = cv2.VideoWriter(vid_fname,cv2.VideoWriter_fourcc(*'MP4V'), 1, (530,530))
[out.write(cv2.imread(fname)) for fname in files]
out.release()
```

Our model presented a decent behavior in the first try (lr=0.0001 , epochs = 10) but it can be further improved. We will try in first place, different values for epoch and training rates. Then, if the model is still not performing properly, we might change the size of the latent tensors, or the amount and dimensions of hidden layers in the discriminator and generator models. We could also try different activation functions. I will not try to use linear layers, I'll keep on using convolution (and deconvolution) layers because the images in this dataset are quite complex and convolution layers should perform better.

Conclusions

We've analyzed how GANs work to generate new faces from a very large data set (100k images) with celebrities' faces. The idea was to create new faces that can look like real people, eventhough they do not belong to real persons. A good model that creates new faces could be usefull for publicity or other purposes, when we don't want

to use faces of real people to avoid problems like ilegal use of image or others. The use of GANs is a very efficient and inexpensive method to achieve this goal. GANs are built using two models inside of them: the discriminator and the generator. The generator will create images (starting from random noise) trying to output images that each time we run the code resemble more and more to real images. The discriminator model should distinguish between real and fake images, thus it is just a binary classificator. Both interplay during the training process and this dialogue between generator and discriminator should gradually improve the performance of the whole model. In our present model we've created a binary classifier using a convolutional neural network for the discrminator, with 5 layers and a 4x4 kernel. The activation function for each layer was a the LeakyReLu. On the other hand, the generator model received a batch of vectors (latent vectors) of size 128x1x1 as its input and then generated a set of image tensors (3x64x64) as its output, by means of 5 deconvolution layers and plain ReLU as the activation function. To improve the performance of the model, we've tried changing the learning rate (lr) and the number of epochs. We've found out that an lr around 0.001 behaves better than higher values like 0.005. In the case of lr=0.005, even when the final loss in the generator was smaller than for lr=0.001, the behavior as the epochs increased was not very stable and the performance in the scores was very poor (real_score was around 0.6 or less). Increasing the number of epochs improves the performance when the lr is the adequate, if lr is very high the scores start oscilating much as epochs number grows. After this, changing the dimension of the latent vectors was the next step to try improving the model. Increasing the size (latent_size=512) of the latent vectors was not favourable to the performance of the model: the generator loss was increased and the scores were oscilating far from the desired value. If we use a latent_size=64 (lower than the initially chosen) after 10 epochs, the model doesn't generate images that resemble to the pictures of the real dataset. Other changes that can be made to the model is changing the number of layers in the generator or in the discriminator models(in this procedure we have to be carefull with the dimensions in each layer of the generator) or the activation function in each layer. Although we've tried many things (lr=0.0001, 0.0002, 0.0005, 0,00075, 0.001, 0.005; epochs=10, 20, 25; latent_size=128, 64, 256, 512, 1024), sometimes the faces look a little bit like Frankenstein or as a person that has suffered a terrible accident, but in many cases the obtained faces does look like obtained directly from the camera. This was the purpose of creating the model and, despite some exceptions, we've reached our goal. The best result obtained, in my opinion, was for the following parameters: latent_size=128, lr=0.001, epochs=25. I'm displaying it in this notebook.

```
jovian.commit(project=project_name,
              outputs=['G.pth', 'D.pth', 'gans_training.avi'],
              environment=None)
```

```
[jovian] Detected Colab notebook...
[jovian] Uploading colab notebook to Jovian...
[jovian] Uploading additional outputs...
[jovian] Attaching records (metrics, hyperparameters, dataset etc.)
[jovian] Committed successfully! https://jovian.ai/viquiriglos/new-celebrities
```

```
'https://jovian.ai/viquiriglos/new-celebrities'
```