Introducción

Etapa I - Reconocimiento Facial

[ ] *↳ 46 celdas ocultas*

Etapa II - Carga de datos

Instalar dependencias

```
import os
import gc
from tqdm.notebook import tqdm
import multiprocessing as mp

import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.models as models
import torch.nn.functional as F

import matplotlib.pyplot as plt

from torch.utils.data import Dataset, DataLoader
from torchvision import transforms, datasets
from torchvision.utils import save_image, make_grid
from torchsummary import summary
```

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

```
Mounted at /content/drive
```

Carga de Datasets y Dataloaders

Se crean dos conjuntos de datos y dataloaders para entrenar el modelo:

- **Dataset Original**: Contiene imágenes sin recortar de celebridades. Este conjunto de datos es útil para evaluar el rendimiento del modelo en su forma original.
- **Dataset Preprocesado**: Contiene imágenes donde se ha recortado la cara de los actores. Este conjunto es importante para concentrarse en las características faciales únicamente.

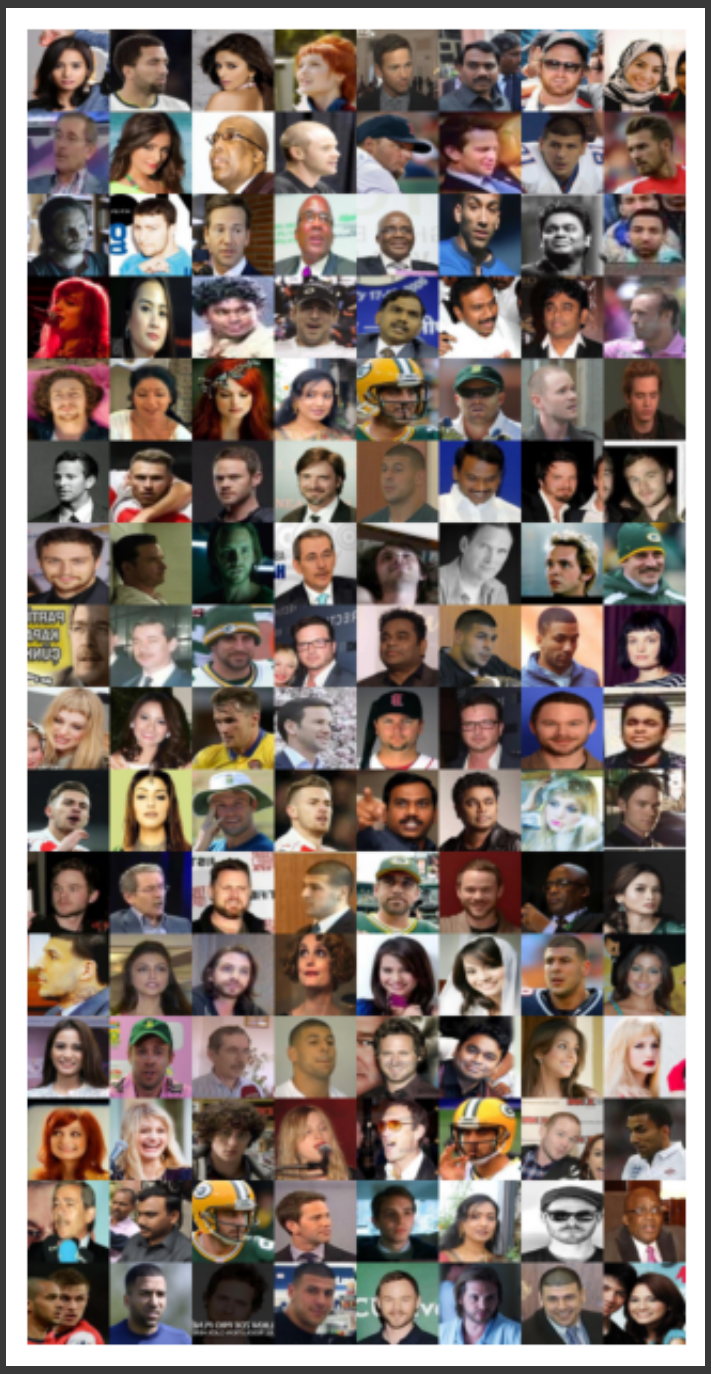
```
image_size = 64
batch_size = 128
device = torch.device("cuda") if torch.cuda.is_available() else torch.de
print(device)

transform = transforms.Compose([
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.Resize((image_size, image_size)),
    transforms.ToTensor(),
    transforms.ConvertImageDtype(torch.float),
])

# Cargamos el dataset original y el dataset preprocesado con las caras c
original_dataset = datasets.ImageFolder(root='/content/drive/MyDrive/Tra
dataloader_original = DataLoader(original_dataset, batch_size=batch_size

preprocessed_dataset = datasets.ImageFolder(root='/content/drive/MyDrive
dataloader_preprocessed = DataLoader(preprocessed_dataset, batch_size=ba
```

```
cuda
/usr/local/lib/python3.10/dist-packages/torch/utils/data/dataloader.p
  warnings.warn(
```

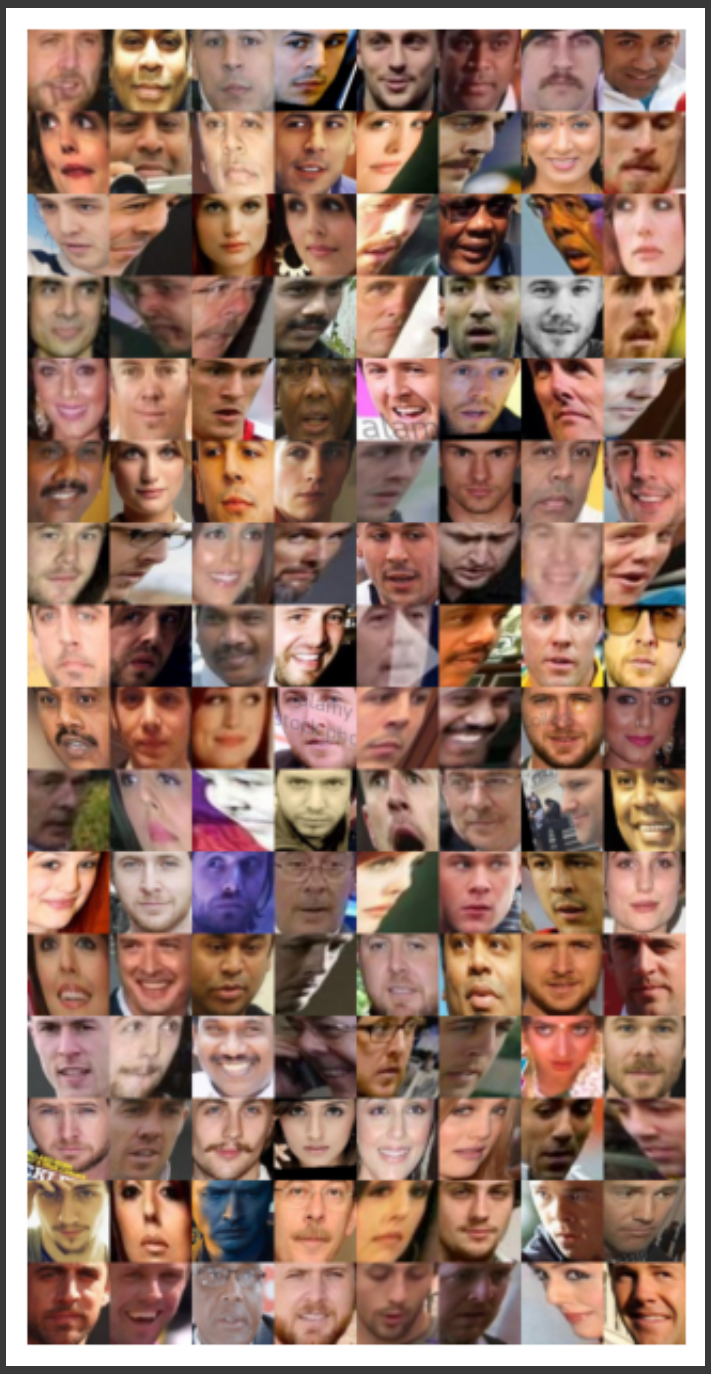
Verificación de dataloaders

A continuación, verificamos que los dataloaders funcionan correctamente mostrando las clases y las imágenes en ambos conjuntos de datos.

```
faces, _ = next(iter(dataloader_original))

grid = torchvision.utils.make_grid(faces, nrow=8, padding=0, scale_each=

fig = plt.figure(figsize=(16,8))
plt.imshow(grid.cpu().permute(1, 2, 0))
plt.axis('off')
plt.show()
```



```
cropped_faces, _ = next(iter(dataloader_preprocessed))

grid = torchvision.utils.make_grid(cropped_faces, nrow=8, padding=0, sca

fig = plt.figure(figsize=(16,8))
plt.imshow(grid.cpu().permute(1, 2, 0))
plt.axis('off')
plt.show()
```



ETAPA III - Entrenamiento de modelos

`Decoder`

Esta clase `Decoder` es un de decodificador que toma un vector de características latente `z` y lo transforma en una imagen de salida de dimensiones específicas.

Estructura del Código

1. Inicialización de parámetros

- `z_dim`: Dimensión del vector latente (512 por defecto), que representa la información comprimida del decodificador.
- `gf_dim`: Dimensión base para los filtros de convolución, que se multiplica para determinar el número de filtros en cada capa.
- `output_size`: Tamaño de la imagen de salida (por defecto 64x64).
- `c_dim`: Número de canales de la imagen de salida (3 por defecto, para imágenes RGB).

2. Capa completamente conectada (Fully connected layer)

- `self.fc`: Es una capa lineal que convierte el vector latente `z` en una matriz de dimensiones adecuadas para las capas de convolución transpuesta posteriores.
- El resultado se redimensiona a `(-1, gf_dim * 8, s8, s8)`, donde `s8` es `output_size // 8`.

3. Capas de convolución transpuesta (Transposed Convolutional Layers)

- Estas capas (`deconv1` a `deconv4`) expanden gradualmente el tamaño espacial de los mapas de características hasta alcanzar el tamaño de la imagen de salida.
- Cada capa tiene activación *ReLU* y, en las primeras tres capas, una normalización de lotes (Batch Normalization) para mejorar la estabilidad del entrenamiento.

4. Activación final

- La última capa (`deconv4`) utiliza activación `tanh`, lo que limita los valores de salida entre -1 y 1, útil para imágenes normalizadas en ese rango.

Método `forward`

En el método `forward`, el flujo de datos a través de la red es el siguiente:

1. El vector latente `z` pasa por la capa `fc`, se transforma con `ReLU` y luego se redimensiona para ajustarse al primer mapa de características de convolución transpuesta.
2. Cada capa de convolución transpuesta expande gradualmente el tamaño espacial y reduce la cantidad de canales.
3. La última capa aplica `tanh` para obtener la imagen de salida, que tiene las dimensiones especificadas (`output_size` x `output_size` x `c_dim`).

```
class Decoder(nn.Module):
    def __init__(self, z_dim=512, gf_dim=64, output_size=64, c_dim=3):
        super(Decoder, self).__init__()
        self.z_dim = z_dim
        self.gf_dim = gf_dim
        s8 = output_size // 8

        # Capas densas
        self.fc = nn.Linear(z_dim, gf_dim * 8 * s8 * s8)

        # Capas convolucionales transpuestas
        self.deconv1 = nn.ConvTranspose2d(gf_dim * 8, gf_dim * 4, kernel
        self.bn1 = nn.BatchNorm2d(gf_dim * 4)

        self.deconv2 = nn.ConvTranspose2d(gf_dim * 4, gf_dim * 2, kernel
        self.bn2 = nn.BatchNorm2d(gf_dim * 2)

        self.deconv3 = nn.ConvTranspose2d(gf_dim * 2, gf_dim // 2, kerne
        self.bn3 = nn.BatchNorm2d(gf_dim // 2)

        self.deconv4 = nn.ConvTranspose2d(gf_dim // 2, c_dim, kernel_siz

    def forward(self, z):
        x = F.relu(self.fc(z)).view(-1, self.gf_dim * 8, 8, 8)

        x = F.relu(self.bn1(self.deconv1(x)))
        x = F.relu(self.bn2(self.deconv2(x)))
        x = F.relu(self.bn3(self.deconv3(x)))

        x = torch.tanh(self.deconv4(x))
        return x
```

Encoder

La clase `Encoder` es un codificador que toma una imagen de entrada y la convierte en un vector latente de características `z`.

Estructura del Código

1. Inicialización de parámetros

- `z_dim`: Dimensión del vector latente (512 por defecto).
- `ef_dim`: Dimensión base para los filtros de las capas convolucionales, que se multiplica en cada capa para aumentar el número de filtros.

2. Capas convolucionales

- Las capas `conv1` a `conv4` son convoluciones 2D que extraen características espaciales de la imagen, reduciendo progresivamente el tamaño espacial de los mapas de características mediante el uso de `stride=2`.
- Cada capa convolucional tiene una activación *ReLU* y una normalización de lotes (*Batch Normalization*) para mejorar la estabilidad del entrenamiento.

3. Capas densas

- `fc_mean` y `fc_logvar`: Estas capas lineales son responsables de producir la media (`z_mean`) y la varianza logarítmica (`z_logvar`) del vector latente. Estas variables permiten que el codificador aprenda una distribución latente en lugar de un único valor, útil en los *Variational Autoencoders*.
- `fc_logvar` utiliza una activación `softplus` para garantizar que la varianza sea positiva.

Método `forward`

En el método `forward`, el flujo de datos es el siguiente:

1. La imagen de entrada `x` pasa a través de las capas convolucionales, donde se reduce el tamaño espacial y se aumentan los filtros.
2. El resultado se aplana y se pasa a través de las capas densas `fc_mean` y `fc_logvar`, que producen la media y varianza logarítmica del vector latente.
3. La salida es una tupla `(z_mean, z_logvar)`, que representan la distribución latente de la imagen.

```
class Encoder(nn.Module):
    def __init__(self, z_dim=512, ef_dim=64):
        super(Encoder, self).__init__()
        self.z_dim = z_dim
        self.ef_dim = ef_dim

        # Capas convolucionales
        self.conv1 = nn.Conv2d(3, ef_dim, kernel_size=5, stride=2, paddi
        self.bn1 = nn.BatchNorm2d(ef_dim)

        self.conv2 = nn.Conv2d(ef_dim, ef_dim * 2, kernel_size=5, stride
        self.bn2 = nn.BatchNorm2d(ef_dim * 2)

        self.conv3 = nn.Conv2d(ef_dim * 2, ef_dim * 4, kernel_size=5, st
        self.bn3 = nn.BatchNorm2d(ef_dim * 4)

        self.conv4 = nn.Conv2d(ef_dim * 4, ef_dim * 8, kernel_size=5, st
        self.bn4 = nn.BatchNorm2d(ef_dim * 8)

        # Capas densas
        self.fc_mean = nn.Linear(ef_dim * 8 * 4 * 4, z_dim)
        self.fc_logvar = nn.Linear(ef_dim * 8 * 4 * 4, z_dim)

    def forward(self, x):
        x = F.relu(self.bn1(self.conv1(x)))
        x = F.relu(self.bn2(self.conv2(x)))
        x = F.relu(self.bn3(self.conv3(x)))
        x = F.relu(self.bn4(self.conv4(x)))
        x = x.view(x.size(0), -1)

        # Media y desviación estandar
        z_mean = self.fc_mean(x)
        z_logvar = F.softplus(self.fc_logvar(x)) + 1e-6
        return z_mean, z_logvar
```

VAE

La clase `VAE` implementa un *Variational Autoencoder*, una arquitectura de red neuronal que mapea datos de entrada (como imágenes) a una representación latente y luego genera datos similares al decodificar desde este espacio latente.

1. Inicialización (`__init__`)

- **`z_dim`**: Dimensión del espacio latente.
- **`ef_dim` y `gf_dim`**: Dimensiones base de los filtros para el *Encoder* y el *Decoder*, respectivamente.
- **`output_size`**: Tamaño de la salida reconstruida (usualmente el tamaño de la imagen original).
- **`c_dim`**: Número de canales de la imagen (3 para RGB).

El `VAE` se compone de dos subcomponentes:

- **Encoder**: Convierte la entrada en un vector latente con una media y varianza logarítmica.
- **Decoder**: Reconstruye los datos a partir del vector latente.

2. Método `reparameterize`

El método `reparameterize` permite la generación del vector latente `z` usando la técnica de reparametrización:

- Calcula `std` (desviación estándar) aplicando `torch.exp(0.5 * logvar)`.
- Genera una variable aleatoria `eps` del mismo tamaño que `std`.
- Devuelve el valor `z` usando la fórmula: `mean + eps * std`, lo que permite el flujo de gradientes durante el entrenamiento.

3. Método `forward`

En el método `forward`, el flujo es el siguiente:

1. La entrada `x` pasa al `encoder`, produciendo la media (`mean`) y varianza logarítmica (`logvar`) de la distribución latente.
2. Usa `reparameterize` para obtener el vector `z` en el espacio latente.
3. El vector `z` pasa al `decoder` para reconstruir la entrada, generando `x_reconstructed`.
4. El método devuelve una tupla `(x_reconstructed, mean, logvar)`.

```
class VAE(nn.Module):
    def __init__(self, z_dim=512, ef_dim=64, gf_dim=64, output_size=64,
        super(VAE, self).__init__()
        self.encoder = Encoder(z_dim, ef_dim)
        self.decoder = Decoder(z_dim, gf_dim, output_size, c_dim)

    def reparameterize(self, mean, logvar):
        std = torch.exp(0.5 * logvar)
        eps = torch.randn_like(std)
        return mean + eps * std

    def forward(self, x):
        mean, logvar = self.encoder(x)
        z = self.reparameterize(mean, logvar)
        x_reconstructed = self.decoder(z)
        return x_reconstructed, mean, logvar
```

```
model = VAE().to(device)
```

```
model(
    torch.randn(2, 3, image_size, image_size).to(device)
)[0].shape
```

```
torch.Size([2, 3, 64, 64])
```

```
summary(model, (3, image_size, image_size), device="cuda")
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1           [-1, 64, 32, 32]           4,864
       BatchNorm2d-2           [-1, 64, 32, 32]             128
            Conv2d-3          [-1, 128, 16, 16]         204,928
       BatchNorm2d-4          [-1, 128, 16, 16]             256
            Conv2d-5            [-1, 256, 8, 8]         819,456
       BatchNorm2d-6            [-1, 256, 8, 8]             512
            Conv2d-7            [-1, 512, 4, 4]       3,277,312
       BatchNorm2d-8            [-1, 512, 4, 4]           1,024
            Linear-9                  [-1, 512]       4,194,816
           Linear-10                  [-1, 512]       4,194,816
          Encoder-11     [[-1, 512], [-1, 512]]               0
           Linear-12                [-1, 32768]      16,809,984
  ConvTranspose2d-13          [-1, 256, 16, 16]       3,277,056
      BatchNorm2d-14          [-1, 256, 16, 16]             512
  ConvTranspose2d-15          [-1, 128, 32, 32]         819,328
      BatchNorm2d-16          [-1, 128, 32, 32]             256
  ConvTranspose2d-17           [-1, 32, 64, 64]         102,432
      BatchNorm2d-18           [-1, 32, 64, 64]              64
  ConvTranspose2d-19            [-1, 3, 64, 64]           2,403
          Decoder-20            [-1, 3, 64, 64]               0
================================================================
Total params: 33,710,147
Trainable params: 33,710,147
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.05
Forward/backward pass size (MB): 5.32
Params size (MB): 128.59
Estimated Total Size (MB): 133.96
----------------------------------------------------------------
```

Entrenamiento

La funcion `train_vae` entrena un modelo *Variational Autoencoder* (VAE) utilizando PyTorch. Incluye la configuración de la función de pérdida, la optimización, y el guardado del modelo.

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

```
import torch.optim as optim
from torch.utils.data import DataLoader
import time

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Método de entrenamiento
def train_vae(model, data_loader, epochs=30, learning_rate=0.001, beta=0
    # Configuración del optimizador
    optimizer = optim.Adam(model.parameters(), lr=learning_rate)

    # Definición de las funciones de pérdida
    mse_loss = nn.MSELoss(reduction='sum')

    # Entrenamiento del modelo
    model.train()
    training_start_time = time.time()

    for epoch in range(epochs):
        epoch_start_time = time.time()
        total_epoch_loss = 0

        # Iteramos sobre el dataset
        for idx, (batch_images, _) in enumerate(data_loader):
            # Pasa los datos al dispositivo
            batch_images = batch_images.to(device)

            # Forward: reconstrucción y cálculo de mu y log_var
            recon_batch, mu, log_var = model(batch_images)

            # Pérdidas: reconstrucción (MSE) y KL divergence
            sse_loss = mse_loss(recon_batch, batch_images)
            kl_loss = -0.5 * torch.sum(1 + log_var - mu.pow(2) - log_var

            # Pérdida total con beta-weighting para KL divergence
            vae_loss = sse_loss + beta * kl_loss

            # Acumula la pérdida total para esta epoch
            total_epoch_loss += vae_loss.item()

            # Backpropagation y optimización
            optimizer.zero_grad()
            vae_loss.backward()
            optimizer.step()

        average_epoch_loss = total_epoch_loss / len(data_loader)
```

```
        print(f"Epoch {epoch+1}/{epochs} - Loss: {average_epoch_loss:.4f

    print(f"Entrenamiento completo en {(time.time() - training_start_tim

    # Guardar el modelo entrenado
    torch.save(model.state_dict(), checkpoint_path)
    print(f"Modelo guardado en {checkpoint_path}")
```

Dataloader original

```
train_vae(model, dataloader_original, epochs=200, learning_rate=0.001, b
```

```
Epoch 1/200 - Loss: 87917151080370.5156 - Tiempo: 10.40 mins
Epoch 2/200 - Loss: 138734.0994 - Tiempo: 0.48 mins
Epoch 3/200 - Loss: 119040.5692 - Tiempo: 0.48 mins
Epoch 4/200 - Loss: 102362.9402 - Tiempo: 0.47 mins
Epoch 5/200 - Loss: 88996.5536 - Tiempo: 0.46 mins
Epoch 6/200 - Loss: 77790.6640 - Tiempo: 0.47 mins
Epoch 7/200 - Loss: 73446.8354 - Tiempo: 0.48 mins
Epoch 8/200 - Loss: 66231.3424 - Tiempo: 0.47 mins
Epoch 9/200 - Loss: 63713.1630 - Tiempo: 0.46 mins
Epoch 10/200 - Loss: 62677.4297 - Tiempo: 0.50 mins
Epoch 11/200 - Loss: 59478.0126 - Tiempo: 0.47 mins
Epoch 12/200 - Loss: 57335.9014 - Tiempo: 0.48 mins
Epoch 13/200 - Loss: 53716.4549 - Tiempo: 0.47 mins
Epoch 14/200 - Loss: 51768.7782 - Tiempo: 0.48 mins
Epoch 15/200 - Loss: 51574.3697 - Tiempo: 0.48 mins
Epoch 16/200 - Loss: 49896.4872 - Tiempo: 0.48 mins
Epoch 17/200 - Loss: 47771.0855 - Tiempo: 0.49 mins
Epoch 18/200 - Loss: 46572.2065 - Tiempo: 0.48 mins
Epoch 19/200 - Loss: 46110.3909 - Tiempo: 0.49 mins
Epoch 20/200 - Loss: 45160.2509 - Tiempo: 0.47 mins
Epoch 21/200 - Loss: 44376.6069 - Tiempo: 0.51 mins
Epoch 22/200 - Loss: 43495.9556 - Tiempo: 0.48 mins
Epoch 23/200 - Loss: 43079.8952 - Tiempo: 0.48 mins
Epoch 24/200 - Loss: 43134.6929 - Tiempo: 0.47 mins
Epoch 25/200 - Loss: 42359.4373 - Tiempo: 0.47 mins
Epoch 26/200 - Loss: 41539.0286 - Tiempo: 0.47 mins
Epoch 27/200 - Loss: 40664.7058 - Tiempo: 0.48 mins
Epoch 28/200 - Loss: 40233.3268 - Tiempo: 0.47 mins
Epoch 29/200 - Loss: 40060.7718 - Tiempo: 0.48 mins
Epoch 30/200 - Loss: 39401.8414 - Tiempo: 0.49 mins
Epoch 31/200 - Loss: 38338.0609 - Tiempo: 0.48 mins
Epoch 32/200 - Loss: 38270.7930 - Tiempo: 0.50 mins
Epoch 33/200 - Loss: 37920.7184 - Tiempo: 0.47 mins
Epoch 34/200 - Loss: 37207.2425 - Tiempo: 0.46 mins
Epoch 35/200 - Loss: 37506.4786 - Tiempo: 0.48 mins
Epoch 36/200 - Loss: 36554.4539 - Tiempo: 0.53 mins
```

```
Epoch 37/200 - Loss: 37175.8211 - Tiempo: 0.48 mins
Epoch 38/200 - Loss: 35953.9844 - Tiempo: 0.48 mins
Epoch 39/200 - Loss: 35701.3977 - Tiempo: 0.49 mins
Epoch 40/200 - Loss: 35235.8887 - Tiempo: 0.47 mins
Epoch 41/200 - Loss: 34515.4349 - Tiempo: 0.49 mins
Epoch 42/200 - Loss: 33727.1992 - Tiempo: 0.49 mins
Epoch 43/200 - Loss: 34113.6336 - Tiempo: 0.52 mins
Epoch 44/200 - Loss: 33411.8895 - Tiempo: 0.47 mins
Epoch 45/200 - Loss: 32974.1371 - Tiempo: 0.49 mins
Epoch 46/200 - Loss: 32638.8785 - Tiempo: 0.48 mins
Epoch 47/200 - Loss: 32458.5829 - Tiempo: 0.48 mins
Epoch 48/200 - Loss: 32366.8262 - Tiempo: 0.49 mins
Epoch 49/200 - Loss: 32225.7589 - Tiempo: 0.48 mins
Epoch 50/200 - Loss: 31570.3665 - Tiempo: 0.47 mins
Epoch 51/200 - Loss: 31458.9638 - Tiempo: 0.48 mins
Epoch 52/200 - Loss: 31120.6598 - Tiempo: 0.47 mins
Epoch 53/200 - Loss: 30683.3450 - Tiempo: 0.48 mins
Epoch 54/200 - Loss: 30141.2933 - Tiempo: 0.51 mins
Epoch 55/200 - Loss: 30090.7425 - Tiempo: 0.47 mins
Epoch 56/200 - Loss: 30551.6028 - Tiempo: 0.47 mins
Epoch 57/200 - Loss: 29604.3766 - Tiempo: 0.49 mins
Epoch 58/200 - Loss: 28895.9539 - Tiempo: 0.49 mins
Epoch 59/200 - Loss: 28586.7661 - Tiempo: 0.48 mins
```

Dataloader preprocesado

```
train_vae(model, dataloader_preprocessed, epochs=200, learning_rate=0.00
```

```
Epoch 1/200 - Loss: 42797.2415 - Tiempo: 8.15 mins
Epoch 2/200 - Loss: 21197.4232 - Tiempo: 0.30 mins
Epoch 3/200 - Loss: 17705.2296 - Tiempo: 0.32 mins
Epoch 4/200 - Loss: 16034.9654 - Tiempo: 0.30 mins
Epoch 5/200 - Loss: 15125.6311 - Tiempo: 0.31 mins
Epoch 6/200 - Loss: 14095.2015 - Tiempo: 0.31 mins
Epoch 7/200 - Loss: 13632.1558 - Tiempo: 0.30 mins
Epoch 8/200 - Loss: 12914.4673 - Tiempo: 0.31 mins
Epoch 9/200 - Loss: 12602.7408 - Tiempo: 0.31 mins
Epoch 10/200 - Loss: 12076.8940 - Tiempo: 0.30 mins
Epoch 11/200 - Loss: 12071.1881 - Tiempo: 0.30 mins
Epoch 12/200 - Loss: 11530.3029 - Tiempo: 0.31 mins
Epoch 13/200 - Loss: 10833.8528 - Tiempo: 0.31 mins
Epoch 14/200 - Loss: 10775.2195 - Tiempo: 0.30 mins
Epoch 15/200 - Loss: 10570.0967 - Tiempo: 0.31 mins
Epoch 16/200 - Loss: 10112.6063 - Tiempo: 0.30 mins
Epoch 17/200 - Loss: 9767.5270 - Tiempo: 0.29 mins
Epoch 18/200 - Loss: 9476.0635 - Tiempo: 0.31 mins
Epoch 19/200 - Loss: 9290.8661 - Tiempo: 0.31 mins
Epoch 20/200 - Loss: 9241.2979 - Tiempo: 0.31 mins
Epoch 21/200 - Loss: 8970.8963 - Tiempo: 0.30 mins
Epoch 22/200 - Loss: 8688.8613 - Tiempo: 0.32 mins
```

```
Epoch 23/200 - Loss: 8662.9566 - Tiempo: 0.30 mins
Epoch 24/200 - Loss: 8588.5126 - Tiempo: 0.31 mins
Epoch 25/200 - Loss: 8270.8286 - Tiempo: 0.32 mins
Epoch 26/200 - Loss: 8127.1739 - Tiempo: 0.31 mins
Epoch 27/200 - Loss: 7882.1252 - Tiempo: 0.31 mins
Epoch 28/200 - Loss: 7796.3300 - Tiempo: 0.31 mins
Epoch 29/200 - Loss: 7756.7199 - Tiempo: 0.30 mins
Epoch 30/200 - Loss: 7559.7728 - Tiempo: 0.30 mins
Epoch 31/200 - Loss: 7441.1974 - Tiempo: 0.33 mins
Epoch 32/200 - Loss: 7780.1513 - Tiempo: 0.30 mins
Epoch 33/200 - Loss: 7447.3408 - Tiempo: 0.31 mins
Epoch 34/200 - Loss: 7109.0341 - Tiempo: 0.31 mins
Epoch 35/200 - Loss: 7173.3149 - Tiempo: 0.30 mins
Epoch 36/200 - Loss: 6956.5645 - Tiempo: 0.30 mins
Epoch 37/200 - Loss: 6941.2025 - Tiempo: 0.32 mins
Epoch 38/200 - Loss: 6825.3992 - Tiempo: 0.31 mins
Epoch 39/200 - Loss: 6820.4381 - Tiempo: 0.31 mins
Epoch 40/200 - Loss: 6689.0508 - Tiempo: 0.31 mins
Epoch 41/200 - Loss: 6547.3057 - Tiempo: 0.32 mins
Epoch 42/200 - Loss: 6527.3742 - Tiempo: 0.30 mins
Epoch 43/200 - Loss: 6385.4085 - Tiempo: 0.32 mins
Epoch 44/200 - Loss: 6712.1401 - Tiempo: 0.31 mins
Epoch 45/200 - Loss: 6633.3906 - Tiempo: 0.32 mins
Epoch 46/200 - Loss: 6189.6827 - Tiempo: 0.32 mins
Epoch 47/200 - Loss: 6216.3693 - Tiempo: 0.30 mins
Epoch 48/200 - Loss: 6388.0728 - Tiempo: 0.31 mins
Epoch 49/200 - Loss: 6287.6729 - Tiempo: 0.32 mins
Epoch 50/200 - Loss: 6082.7708 - Tiempo: 0.30 mins
Epoch 51/200 - Loss: 6099.9227 - Tiempo: 0.30 mins
Epoch 52/200 - Loss: 5933.2827 - Tiempo: 0.32 mins
Epoch 53/200 - Loss: 6056.7877 - Tiempo: 0.31 mins
Epoch 54/200 - Loss: 5925.0375 - Tiempo: 0.31 mins
Epoch 55/200 - Loss: 5928.1537 - Tiempo: 0.31 mins
Epoch 56/200 - Loss: 5795.8991 - Tiempo: 0.30 mins
Epoch 57/200 - Loss: 5900.5931 - Tiempo: 0.30 mins
Epoch 58/200 - Loss: 5813.0122 - Tiempo: 0.32 mins
Epoch 59/200 - Loss: 5880.8697 - Tiempo: 0.31 mins
```

Cargamos el modelo

```
def load_model(model, checkpoint_path='/content/drive/MyDrive/Trabajo In
    model.load_state_dict(torch.load(checkpoint_path, map_location=devic
    model.to(device)
    model.eval()
    return model
```

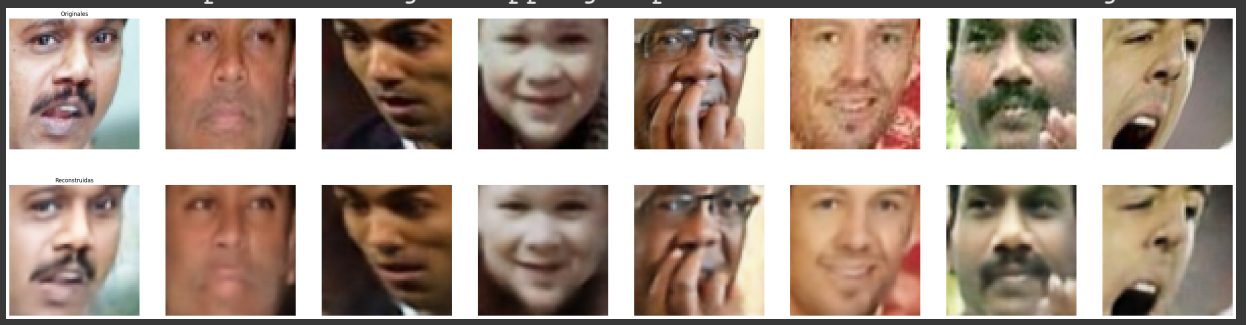
```
original_model = load_model(model, checkpoint_path='/content/drive/MyDri
preprocessed_model = load_model(model, checkpoint_path='/content/drive/M
```

```
<ipython-input-10-d24bc378bb3b>:2: FutureWarning: You are using `tor
  model.load_state_dict(torch.load(checkpoint_path, map_location=dev
```

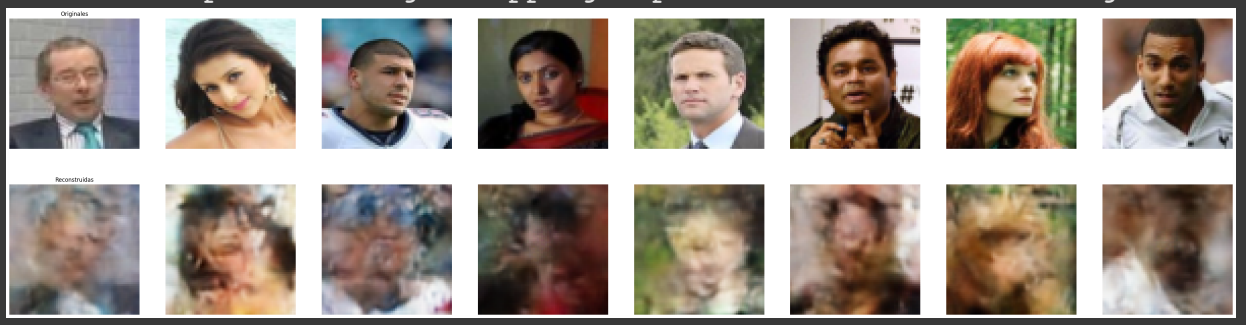
Inferencias

```
def infer_and_show_images(dataloader, model, device, num_images=8, image
    model.eval()

    with torch.no_grad():
        images, _ = next(iter(dataloader))
        images = images.to(device)

        # Generar inferencia
        reconstructed_images, _, _ = model(images)

        # Mover los tensores a CPU para visualización
        images = images.cpu()
        reconstructed_images = reconstructed_images.cpu()

        # Visualizar las imágenes originales y reconstruidas con mayor t
        fig, axes = plt.subplots(2, num_images, figsize=(num_images * im

        # Mostrar imágenes originales
        for i in range(num_images):
            axes[0, i].imshow(images[i].permute(1, 2, 0))
            axes[0, i].axis('off')
        axes[0, 0].set_title("Originales")

        # Mostrar imágenes reconstruidas
        for i in range(num_images):
            axes[1, i].imshow(reconstructed_images[i].permute(1, 2, 0))
            axes[1, i].axis('off')
        axes[1, 0].set_title("Reconstruidas")

        plt.show()

# Ejecutar inferencias y mostrar resultados con mayor tamaño de imagen
infer_and_show_images(dataloader_preprocessed, preprocessed_model, devic
infer_and_show_images(dataloader_original, original_model, device=device
```

```
/usr/local/lib/python3.10/dist-packages/torch/utils/data/dataloader.p
  warnings.warn(
WARNING:matplotlib.image:Clipping input data to the valid range for i
WARNING:matplotlib.image:Clipping input data to the valid range for i
```




```
WARNING:matplotlib.image:Clipping input data to the valid range for i
WARNING:matplotlib.image:Clipping input data to the valid range for i
```



Podemos ver que claramente el modelo entrenado con las imagenes preprocesadas es mucho mejor por lo que seleccionamos este modelo para hacer las inferencias y graficar el espacio latente

Interpolación

```
import numpy as np

def get_interp(v1, v2, n):
  if not v1.shape == v2.shape:
    raise Exception('Diferent vector size')

  v1 = v1.to("cpu")
  v2 = v2.to("cpu")

  return np.array([np.linspace(v1[i], v2[i], n+2) for i in range(v1.shap

def model_interp(model, index1, index2, size = 10):
  img1 = preprocessed_dataset[index1][0].to(device).unsqueeze(0)
  img2 = preprocessed_dataset[index2][0].to(device).unsqueeze(0)

  with torch.no_grad():
    img1_compressed = model.encoder(img1)[0]
    img2_compressed = model.encoder(img2)[0]
    interps = get_interp(img1_compressed, img2_compressed, size)

    interps = torch.tensor(interps).to(device).squeeze()
    interps = interps.permute(1, 0)

    decoded_interps = model.decoder(interps)

  return decoded_interps

def show_interp(imgs, index1, index2, titles=None, scale=1.5):
  figsize = (12 * scale, 1 * scale)
  _, axes = plt.subplots(1, 12, figsize=figsize)
  axes = axes.flatten()
  for i, (ax, img) in enumerate(zip(axes, imgs)):
    try:
      img = img.detach().numpy()
    except:
      pass
    if i==0:
      ax.set_title(titles[0])
      ax.imshow(preprocessed_dataset[index1][0].permute(1,2,0).cpu().det
    elif i==11:
      ax.set_title(titles[1])
      ax.imshow(preprocessed_dataset[index2][0].permute(1,2,0).cpu().det
    else:
      ax.imshow(img)
      ax.axes.get_xaxis().set_visible(False)
      ax.axes.get_yaxis().set_visible(False)
```

```
  return axes
```

```
index1 = 300
index2 = 560
interp_result = model_interp(model=preprocessed_model, index1=index1, in
imgs = [img.permute(1, 2, 0).cpu() for img in interp_result]
show_interp(imgs, index1, index2, scale=2,  titles=['1', '2'])
```

```
WARNING:matplotlib.image:Clipping input data to the valid range for i
WARNING:matplotlib.image:Clipping input data to the valid range for i
array([<Axes: title={'center': '1'}>, <Axes: >, <Axes: >, <Axes: >,
       <Axes: >, <Axes: >, <Axes: >, <Axes: >, <Axes: >, <Axes: >,
       <Axes: >, <Axes: title={'center': '2'}>], dtype=object)
```


```
import torch
from sklearn.manifold import TSNE
import matplotlib.pyplot as plt
import psutil
import numpy as np

# Tamaño del batch reducido para evitar sobrecargar la memoria
batch_size = 500  # Puedes ajustar este valor según tu memoria disponibl
data_list = []
labels_list = []

# Listas para almacenar los resultados de TSNE y las etiquetas
z_2D_list = []
labels_data_list = []

# Itera sobre el DataLoader para obtener los datos
for inputs, labels in dataloader_preprocessed:
    data_list.append(inputs)
    labels_list.append(labels)

    # Si hemos acumulado suficientes datos, procesamos un lote
    if len(data_list) * batch_size >= batch_size:
        # Concatenamos los lotes actuales
        input_data = torch.cat(data_list, dim=0)  # Concatenar todos los
        labels_data = torch.cat(labels_list, dim=0)
```

```
        # Pasa los datos por el encoder para obtener la salida
        output = preprocessed_model.encoder(input_data.to(device))

        # Si la salida es una tupla, asignamos el primer valor (que norm
        z = output[0]

        # Si el espacio latente tiene 2 dimensiones, lo usamos directame
        if z.shape[1] == 2:
            z_2D = z
        else:
            tsne = TSNE()
            z_2D = tsne.fit_transform(z.detach().cpu().numpy())
            z_2D = (z_2D - z_2D.min()) / (z_2D.max() - z_2D.min())  # No

        # Almacena los resultados de TSNE y las etiquetas para graficar
        z_2D_list.append(z_2D)
        labels_data_list.append(labels_data.numpy())

        # Limpiamos las listas para liberar memoria
        data_list.clear()
        labels_list.clear()

        # Liberar memoria después de cada lote
        del input_data
        del labels_data
        torch.cuda.empty_cache()  # Si estás usando CUDA

# Si hay datos restantes que no se procesaron, procesarlos también
if len(data_list) > 0:
    input_data = torch.cat(data_list, dim=0)
    labels_data = torch.cat(labels_list, dim=0)

    output = preprocessed_model.encoder(input_data.to(device))
    z = output[0]

    if z.shape[1] == 2:
        z_2D = z
    else:
        tsne = TSNE()
        z_2D = tsne.fit_transform(z.detach().cpu().numpy())
        z_2D = (z_2D - z_2D.min()) / (z_2D.max() - z_2D.min())  # Normal

    labels_data = labels_data.numpy()

    z_2D_list.append(z_2D)
```

```
    labels_data_list.append(labels_data)

    del input_data
    del labels_data
    torch.cuda.empty_cache()  # Si estás usando CUDA

# Al final, concatena todas las partes de z_2D y etiquetas
z_2D_all = np.concatenate(z_2D_list, axis=0)
labels_data_all = np.concatenate(labels_data_list, axis=0)

# Muestra el gráfico con todos los datos
plt.scatter(z_2D_all[:, 0], z_2D_all[:, 1], c=labels_data_all, s=10, cma
plt.show()
```

```
/usr/local/lib/python3.10/dist-packages/torch/utils/data/dataloader.p
  warnings.warn(
```