Máster en Inteligencia Artificial Aplicada

Unidad: Deep Learning - Caso Práctico 2 / Fasion Mnist

Implementación de una cGAN para Generacion Imágenes

Nombre: Patricio Galván

Fecha: 07 de Octubre de 2024

Contenido:

Resumen
En este apartado, crearemos una CGAN, similar a la vista en el notebook *conditional-gan.ipynb*, pero para generar imágenes sintéticas a partir del dataset Fashion MNIST.

Descripcion del Dataset Fasion Mnist

Fashion MNIST es un conjunto de datos diseñado para servir como una alternativa más desafiante y moderna al clásico dataset MNIST de dígitos manuscritos. Contiene imágenes de prendas de ropa en escala de grises, organizadas en 10 categorías de productos de moda.

Características del dataset:

- Cantidad de imágenes: 70,000 imágenes en total.
  - 60,000 imágenes para entrenamiento.
  - 10,000 imágenes para prueba.
- Dimensiones de las imágenes:
  - Cada imagen es de 28x28 píxeles en escala de grises.
- *Categorías:** Las imágenes están clasificadas en 10 categorías, que representan diferentes tipos de prendas de vestir:
  - T-shirt/top
  - Trouser
  - Pullover
  - Dress
  - Coat
  - Sandal
  - Shirt
  - Sneaker
  - Bag
  - Ankle boot

Respuestas a Preguntas

1. Desde el punto de vista técnico, sin fijarnos aún en la calidad de las imágenes sintetizadas, ¿habría que hacer algún cambio obligatorio con respecto a la red utilizada para sintetizar datos de MNIST? Justifique su respuesta.
   `Resp: En realidad no hay ningun parametro obligatorio a cambiar, ya que ambos dataset tienen 10 clases y ambos tienen solamente un canal de color.`
2. A priori, ¿piensas que la red generadora y la discriminadora deberán ser más complejas, igual de complejas, o menos complejas que en el caso de la red utilizada con MNIST para sintetizar datos de calidad suficiente? Justifique su respuesta.
   `Resp: A priori parece logico pensar que al ser imagenes mas complejas, al menos al generadosgenerador debieramos potenciarle la arquitectura. Es probable que el discriminador que funcione bien en MNIST tambien funciones bien con estas imagenes, pero eso hay que probarlo.`

1. Cargamos las Bibliotecas y Funciones

```
!pip install -q git+https://github.com/tensorflow/docs
```

```
  Preparing metadata (setup.py) ... done
```

```
#!pip install --upgrade keras
```

```
import keras
from keras import layers
from keras import ops
from tensorflow_docs.vis import embed
import tensorflow as tf
import numpy as np
import imageio
import matplotlib.pyplot as plt
```

2. Importamos los datos del Dataset Fashion MNIST y preparamos los datos

```
# Hiperparámetros
batch_size = 64
num_channels = 1
num_classes = 10
image_size = 28
latent_dim = 128
```

```
# Importamos el dataset MNIST, que ya viene separado por X, y y train y test
(x_train, y_train), (x_test, y_test) = keras.datasets.fashion_mnist.load_data()
all_clothes = np.concatenate([x_train, x_test])
all_labels = np.concatenate([y_train, y_test])

# Al igual que hemos hecho otras veces, escalamos las imágenes entre 0 y 1
all_clothes = all_clothes.astype("float32") / 255.0
```

Guardamos los nombres de las clases para poder graficarlas

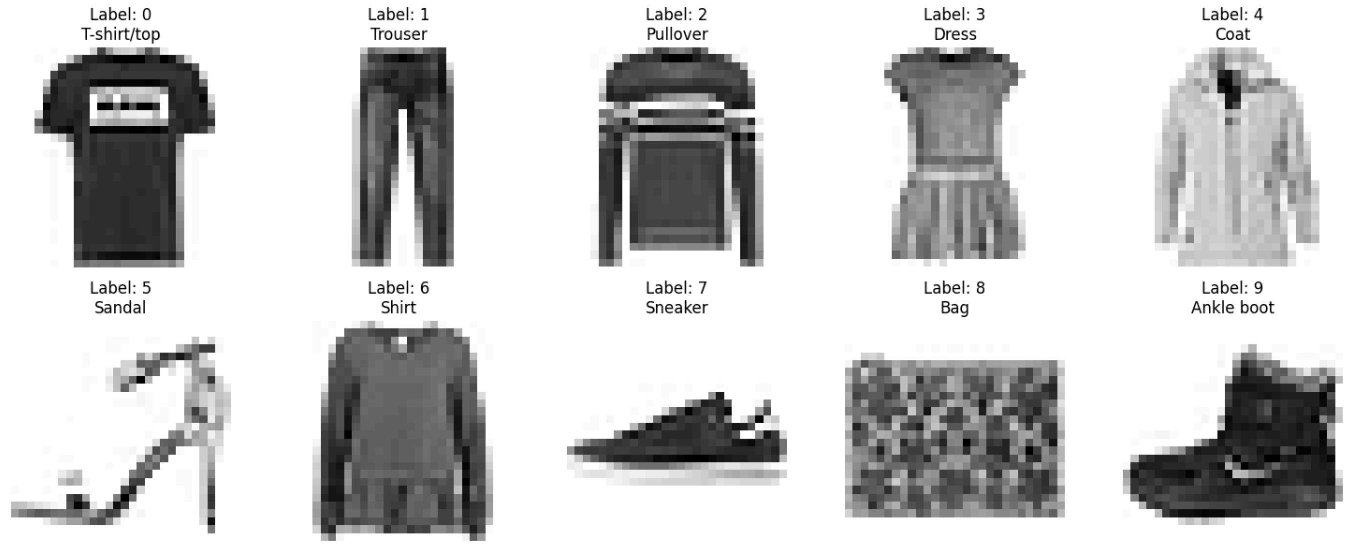
```
class_names = [ "T-shirt/top", "Trouser","Pullover", "Dress", "Coat","Sandal","Shirt","Sneaker","Bag","Ankle boot"
]
```

Las imágenes se redimensionan para tener una forma de 28x28x1, donde el 1 representa que son imágenes en escala de grises (solo un canal de color). Esto es necesario para que la red CNN las procese correctamente, ya que espera una entrada con tres dimensiones.
Posteriormente convetimos las etiquetas a one-hot encoding.

```
all_clothes = np.reshape(all_clothes, (-1, 28, 28, 1))
all_labels = keras.utils.to_categorical(all_labels, 10)
```

Ploteamos ejemplos del Dataset Fasion Mnist

```
# Crear una figura con subplots
fig, axes = plt.subplots(2, 5, figsize=(15, 6))
fig.tight_layout()  # Ajustar el espaciado entre subplots

# Iterar sobre las clases y graficar un ejemplo
for i in range(10):
    # Encontrar el índice de la primera imagen de la clase i
    index = np.where(np.argmax(all_labels, axis=1) == i)[0][0]

    # Obtener la imagen y la etiqueta
    image = all_clothes[index]
    label = all_labels[index]

    # Get the class index from the one-hot encoded label
    class_index = np.argmax(label)

    # Graficar la imagen en el subplot correspondiente
    row = i // 5  # Calcular la fila del subplot
    col = i % 5   # Calcular la columna del subplot
    axes[row, col].imshow(image, cmap='gray_r')

    # Use class_index to index into class_names
    axes[row, col].set_title(f"Label: {class_index}\n{class_names[class_index]}")
    axes[row, col].axis("off")  # Ocultar los ejes

# Mostrar la figura
plt.show()
```



Creamos un objeto tf.data.Dataset, que es la forma eficiente de manipular datos en TensorFlow.
Los datos (all_clothes y all_labels) se combinan para formar el conjunto de datos.
Reordenamos el conjunto de datos con un buffer de 1024 muestras y luego dividimos en lotes (batches) batch_size = 64. El reordenamiento ayuda a romper el orden secuencial de los datos y evitamos patrones no deseados durante el entrenamiento.

```
# Creamos el dataset y separamos en batches
dataset = tf.data.Dataset.from_tensor_slices((all_clothes, all_labels))
dataset = dataset.shuffle(buffer_size=1024).batch(batch_size)

print(f"Shape of training images: {all_clothes.shape}")
print(f"Shape of training labels: {all_labels.shape}")
```

```
Shape of training images: (70000, 28, 28, 1)
Shape of training labels: (70000, 10)
```

Definimos el numero de canales para el Generador y Discriminador.

```
generator_in_channels = latent_dim + num_classes
discriminator_in_channels = num_channels + num_classes

print("generator_in_channels: ", generator_in_channels)
print("discriminator_in_channels: ", discriminator_in_channels)
```

```
generator_in_channels:  138
discriminator_in_channels:  11
```

3. Modelo Inicial

Creamos el Generados y Discriminador. `Probaremos la misma arquitectura empleada en la cGAN del problema de **MNIST** para clasificacion de numeros. En ese problema esta arquitectura funcionó de forma aceptable.`

```
# Creando el discriminador.
discriminator = keras.Sequential(
    [
        # El número de entradas es el calculado anteriormente, discriminator_in_channels
        keras.layers.InputLayer((28, 28, discriminator_in_channels)),
        # Capas ocultas
        layers.Conv2D(64, (3, 3), strides=(2, 2), padding="same"),
        layers.LeakyReLU(negative_slope=0.2),
        layers.Conv2D(128, (3, 3), strides=(2, 2), padding="same"),
        layers.LeakyReLU(negative_slope=0.2),
        layers.GlobalMaxPooling2D(),
        # La salida es binaria, para decidir si la entrada es una imagen real o sintética
        layers.Dense(1),
```

```
    ],
    name="discriminator",
)

discriminator.summary()
```

Model: "discriminator"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 14, 14, 64)	6,400
leaky_re_lu (LeakyReLU)	(None, 14, 14, 64)	0
conv2d_1 (Conv2D)	(None, 7, 7, 128)	73,856
leaky_re_lu_1 (LeakyReLU)	(None, 7, 7, 128)	0
global_max_pooling2d (GlobalMaxPooling2D)	(None, 128)	0
dense (Dense)	(None, 1)	129

Total params: 80,385 (314.00 KB)
Trainable params: 80,385 (314.00 KB)
Non-trainable params: 0 (0.00 B)

```
# Creando el generador.
generator = keras.Sequential(
    [
        # El número de entradas es el calculado anteriormente, generator_in_channels
        keras.layers.InputLayer((generator_in_channels,)),
        # Capas ocultas
        layers.Dense(7 * 7 * generator_in_channels),
        layers.LeakyReLU(negative_slope=0.2),
        layers.Reshape((7, 7, generator_in_channels)),
        layers.Conv2DTranspose(128, (4, 4), strides=(2, 2), padding="same"),
        layers.LeakyReLU(negative_slope=0.2),
        layers.Conv2DTranspose(128, (4, 4), strides=(2, 2), padding="same"),
        layers.LeakyReLU(negative_slope=0.2),
        # La imagen de salida va comprimida en un único canal
        layers.Conv2D(num_channels, (7, 7), padding="same", activation="sigmoid"),
    ],
    name="generator",
)
generator.summary()
```

Model: "generator"

Layer (type)	Output Shape	Param #
dense_1 (Dense)	(None, 6762)	939,918
leaky_re_lu_2 (LeakyReLU)	(None, 6762)	0
reshape (Reshape)	(None, 7, 7, 138)	0
conv2d_transpose (Conv2DTranspose)	(None, 14, 14, 128)	282,752
leaky_re_lu_3 (LeakyReLU)	(None, 14, 14, 128)	0
conv2d_transpose_1 (Conv2DTranspose)	(None, 28, 28, 128)	262,272
leaky_re_lu_4 (LeakyReLU)	(None, 28, 28, 128)	0
conv2d_2 (Conv2D)	(None, 28, 28, 1)	6,273

Total params: 1,491,215 (5.69 MB)
Trainable params: 1,491,215 (5.69 MB)
Non-trainable params: 0 (0.00 B)

```
class ConditionalGAN(keras.Model):
    def __init__(self, discriminator, generator, latent_dim):
        super().__init__()
        self.discriminator = discriminator
        self.generator = generator
        self.latent_dim = latent_dim
        self.seed_generator = keras.random.SeedGenerator(1337)
        self.gen_loss_tracker = keras.metrics.Mean(name="generator_loss")
        self.disc_loss_tracker = keras.metrics.Mean(name="discriminator_loss")

    @property
    def metrics(self):
        return [self.gen_loss_tracker, self.disc_loss_tracker]

    def compile(self, d_optimizer, g_optimizer, loss_fn):
        super().compile()
        self.d_optimizer = d_optimizer
        self.g_optimizer = g_optimizer
```

```
        self.loss_fn = loss_fn

    def train_step(self, data):
        # Unpack the data.
        real_images, one_hot_labels = data

        # Add dummy dimensions to the labels so that they can be concatenated with
        # the images. This is for the discriminator.
        image_one_hot_labels = one_hot_labels[:, :, None, None]
        image_one_hot_labels = ops.repeat(
            image_one_hot_labels, repeats=[image_size * image_size]
        )
        image_one_hot_labels = ops.reshape(
            image_one_hot_labels, (-1, image_size, image_size, num_classes)
        )

        # Sample random points in the latent space and concatenate the labels.
        # This is for the generator.
        batch_size = ops.shape(real_images)[0]
        random_latent_vectors = keras.random.normal(
            shape=(batch_size, self.latent_dim), seed=self.seed_generator
        )
        random_vector_labels = ops.concatenate(
            [random_latent_vectors, one_hot_labels], axis=1
        )

        # Decode the noise (guided by labels) to fake images.
        generated_images = self.generator(random_vector_labels)

        # Combine them with real images. Note that we are concatenating the labels
        # with these images here.
        fake_image_and_labels = ops.concatenate(
            [generated_images, image_one_hot_labels], -1
        )
        real_image_and_labels = ops.concatenate([real_images, image_one_hot_labels], -1)
        combined_images = ops.concatenate(
            [fake_image_and_labels, real_image_and_labels], axis=0
        )

        # Assemble labels discriminating real from fake images.
        labels = ops.concatenate(
            [ops.ones((batch_size, 1)), ops.zeros((batch_size, 1))], axis=0
        )

        # Train the discriminator.
        with tf.GradientTape() as tape:
            predictions = self.discriminator(combined_images)
            d_loss = self.loss_fn(labels, predictions)
        grads = tape.gradient(d_loss, self.discriminator.trainable_weights)
        self.d_optimizer.apply_gradients(
            zip(grads, self.discriminator.trainable_weights)
        )

        # Sample random points in the latent space.
        random_latent_vectors = keras.random.normal(
            shape=(batch_size, self.latent_dim), seed=self.seed_generator
        )
        random_vector_labels = ops.concatenate(
            [random_latent_vectors, one_hot_labels], axis=1
        )

        # Assemble labels that say "all real images".
        misleading_labels = ops.zeros((batch_size, 1))

        # Train the generator (note that we should *not* update the weights
        # of the discriminator)!
        with tf.GradientTape() as tape:
            fake_images = self.generator(random_vector_labels)
            fake_image_and_labels = ops.concatenate(
                [fake_images, image_one_hot_labels], -1
            )
            predictions = self.discriminator(fake_image_and_labels)
            g_loss = self.loss_fn(misleading_labels, predictions)
        grads = tape.gradient(g_loss, self.generator.trainable_weights)
        self.g_optimizer.apply_gradients(zip(grads, self.generator.trainable_weights))

        # Monitor loss.
        self.gen_loss_tracker.update_state(g_loss)
        self.disc_loss_tracker.update_state(d_loss)
        return {
            "g_loss": self.gen_loss_tracker.result(),
            "d_loss": self.disc_loss_tracker.result(),
        }
```

Creamos la clase con_gan, compilamos y corremos el entrenamisnto.

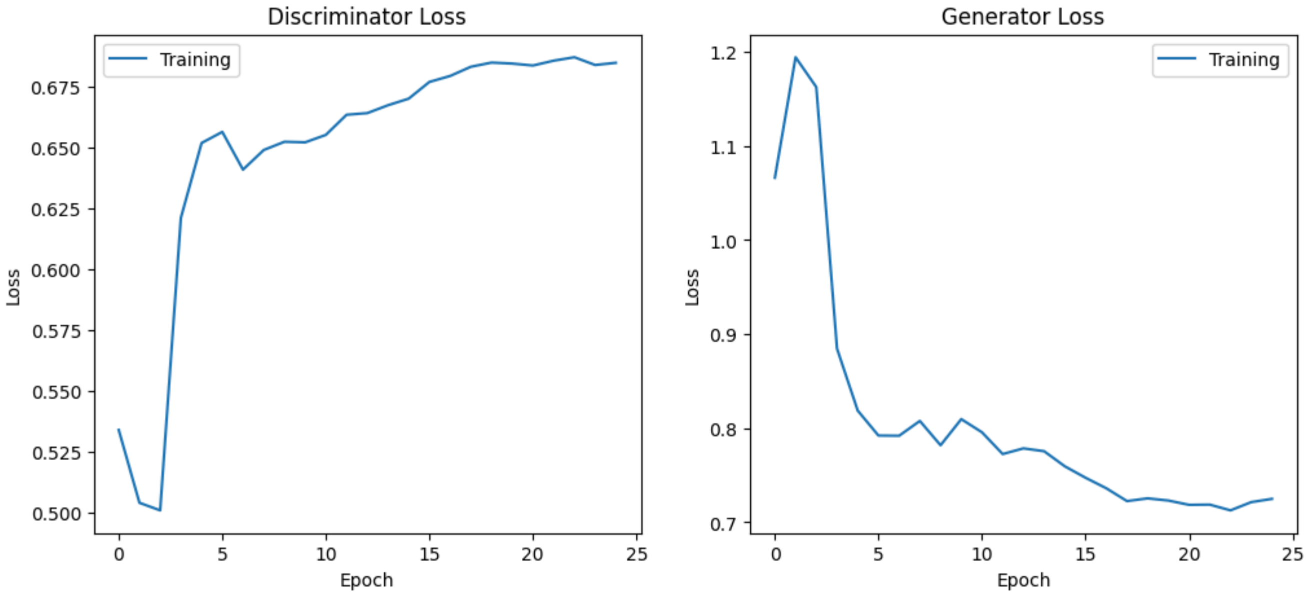
```
d_loss = []
g_loss = []

cond_gan = ConditionalGAN(
    discriminator=discriminator, generator=generator, latent_dim=latent_dim
)

cond_gan.compile(
    d_optimizer=keras.optimizers.Adam(learning_rate=0.0001),
    g_optimizer=keras.optimizers.Adam(learning_rate=0.0001),
    loss_fn=keras.losses.BinaryCrossentropy(from_logits=True),
)

# Guarda las pérdidas en cada época
for epoch in range(25):
    print("Epoch: ", epoch)
    history = cond_gan.fit(dataset, epochs=1, verbose=1)
    d_loss.append(history.history['d_loss'][0])
    g_loss.append(history.history['g_loss'][0])
```

```
Epoch:  0
1094/1094 ━━━━━━━━━━━━━━━━━━━━ 16s 8ms/step - d_loss: 0.5539 - g_loss: 0.9416
Epoch:  1
1094/1094 ━━━━━━━━━━━━━━━━━━━━ 5s 5ms/step - d_loss: 0.4863 - g_loss: 1.2191
Epoch:  2
1094/1094 ━━━━━━━━━━━━━━━━━━━━ 5s 5ms/step - d_loss: 0.4707 - g_loss: 1.2244
Epoch:  3
1094/1094 ━━━━━━━━━━━━━━━━━━━━ 5s 5ms/step - d_loss: 0.5689 - g_loss: 1.0212
Epoch:  4
1094/1094 ━━━━━━━━━━━━━━━━━━━━ 5s 5ms/step - d_loss: 0.6574 - g_loss: 0.7725
Epoch:  5
1094/1094 ━━━━━━━━━━━━━━━━━━━━ 5s 5ms/step - d_loss: 0.6713 - g_loss: 0.7683
Epoch:  6
1094/1094 ━━━━━━━━━━━━━━━━━━━━ 5s 5ms/step - d_loss: 0.6428 - g_loss: 0.8047
Epoch:  7
1094/1094 ━━━━━━━━━━━━━━━━━━━━ 5s 5ms/step - d_loss: 0.6491 - g_loss: 0.8273
Epoch:  8
1094/1094 ━━━━━━━━━━━━━━━━━━━━ 5s 5ms/step - d_loss: 0.6512 - g_loss: 0.7721
Epoch:  9
1094/1094 ━━━━━━━━━━━━━━━━━━━━ 5s 5ms/step - d_loss: 0.6491 - g_loss: 0.8165
Epoch:  10
1094/1094 ━━━━━━━━━━━━━━━━━━━━ 5s 5ms/step - d_loss: 0.6518 - g_loss: 0.8019
Epoch:  11
1094/1094 ━━━━━━━━━━━━━━━━━━━━ 5s 5ms/step - d_loss: 0.6598 - g_loss: 0.7627
Epoch:  12
1094/1094 ━━━━━━━━━━━━━━━━━━━━ 5s 5ms/step - d_loss: 0.6619 - g_loss: 0.8002
Epoch:  13
1094/1094 ━━━━━━━━━━━━━━━━━━━━ 5s 5ms/step - d_loss: 0.6686 - g_loss: 0.7573
Epoch:  14
1094/1094 ━━━━━━━━━━━━━━━━━━━━ 5s 5ms/step - d_loss: 0.6728 - g_loss: 0.7922
Epoch:  15
1094/1094 ━━━━━━━━━━━━━━━━━━━━ 5s 5ms/step - d_loss: 0.6764 - g_loss: 0.7489
Epoch:  16
1094/1094 ━━━━━━━━━━━━━━━━━━━━ 5s 5ms/step - d_loss: 0.6792 - g_loss: 0.7299
Epoch:  17
1094/1094 ━━━━━━━━━━━━━━━━━━━━ 5s 5ms/step - d_loss: 0.6840 - g_loss: 0.7214
Epoch:  18
1094/1094 ━━━━━━━━━━━━━━━━━━━━ 5s 5ms/step - d_loss: 0.6850 - g_loss: 0.7329
Epoch:  19
1094/1094 ━━━━━━━━━━━━━━━━━━━━ 5s 5ms/step - d_loss: 0.6848 - g_loss: 0.7170
Epoch:  20
1094/1094 ━━━━━━━━━━━━━━━━━━━━ 5s 5ms/step - d_loss: 0.6826 - g_loss: 0.7228
Epoch:  21
1094/1094 ━━━━━━━━━━━━━━━━━━━━ 5s 5ms/step - d_loss: 0.6862 - g_loss: 0.7155
Epoch:  22
1094/1094 ━━━━━━━━━━━━━━━━━━━━ 5s 5ms/step - d_loss: 0.6873 - g_loss: 0.7119
Epoch:  23
1094/1094 ━━━━━━━━━━━━━━━━━━━━ 5s 5ms/step - d_loss: 0.6832 - g_loss: 0.7253
Epoch:  24
1094/1094 ━━━━━━━━━━━━━━━━━━━━ 5s 5ms/step - d_loss: 0.6840 - g_loss: 0.7134
```

Graficamos las funciones de perdida del Discriminador y Generador.

```
#Crear la figura y los subplots
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

# Gráfico para d_loss
ax1.plot(d_loss, label='Training')  # Línea para datos de entrenamiento
ax1.set_title('Discriminator Loss')
ax1.set_xlabel('Epoch')
ax1.set_ylabel('Loss')
ax1.legend()

# Gráfico para g_loss
```

```
ax2.plot(g_loss, label='Training')  # Línea para datos de entrenamiento
ax2.set_title('Generator Loss')
ax2.set_xlabel('Epoch')
ax2.set_ylabel('Loss')
ax2.legend()

# Mostrar los gráficos
plt.show()
```



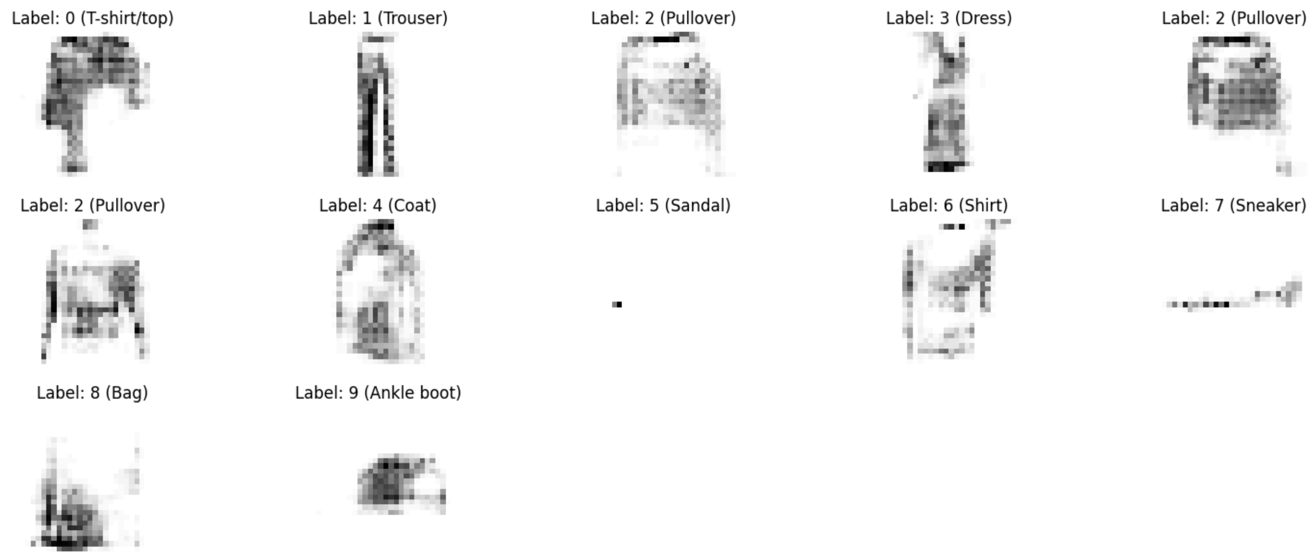

```
# Etiquetas a generar. Modificar la lista labels como queramos
labels = [0,1,2,3,2,2,4,5,6,7,8,9]
n_samples = len(labels)

# Extraemos el generador de la CGAN
trained_gen = cond_gan.generator

# Convertimos las etiquetas en labels a categóricas
labels = keras.utils.to_categorical(labels, num_classes)

# Generamos el ruido para las diferentes imágenes a generar.
noise = keras.random.normal(shape=(n_samples, latent_dim))
noise = ops.reshape(noise, (n_samples, latent_dim))

# Concatenamos el ruido y las etiquetas para tener el input entero del generador
noise_and_labels = ops.concatenate([noise, labels], 1)

# Generamos las imágenes con el input
fake_images = trained_gen.predict(noise_and_labels)

# Convertimos a las dimensiones originales (28x28, aunque podríamos modificar los valores) y los valores de los píxeles yendo de 0 a 255 en
fake_images *= 255.0
converted_images = fake_images.astype(np.uint8)
converted_images = ops.image.resize(converted_images, (28, 28)).numpy().astype(np.uint8)
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 409ms/step
```

Graficamos las imagenes de ejemplo que creo el Generador.

```
# Assuming converted_images contains your generated images
num_images = len(converted_images)
num_cols = 5  # Adjust as needed

# Calculate the actual number of rows and columns needed
num_rows = (num_images + num_cols - 1) // num_cols

fig, axes = plt.subplots(num_rows, num_cols, figsize=(15, 6))
fig.tight_layout()

# Flatten axes if necessary
if num_rows == 1 or num_cols == 1:
    axes = axes.flatten()  # Makes it work for single row or column
else:
    axes = axes.ravel()  # Makes it work for multiple rows and columns

for i, image in enumerate(converted_images):
    # Plot image on the current subplot
    if i < len(axes):
```

```
        ax = axes[i]
        ax.imshow(image[:, :, 0], cmap='gray_r')
        ax.axis('off')  # Hide axes

        # Get label name using ll_labels and class_names
        label_number = labels[i]
        label_index = np.argmax(label_number)

        # Access the label name using list indexing
        label_name = class_names[label_index]

        # Add label as title
        ax.set_title(f"Label: {label_index} ({label_name})")

# Hide any unused subplots
for i in range(num_images, num_rows * num_cols):
    if i < len(axes):
        axes[i].axis('off')

plt.show()
```



Comentario

- Lo que primero podemos observar son las curvas de las funciones de perdida del discriminador y generador.
- Mientras la curva del discriminador aumenta (con ciertos saltos) la curva del generador disminuye (con ciertos saltos). Esto quiere decir que el discriminador no esta captando la informacion de si las imagenes son verdaderas o no. Pero considerando lo malo de las imagenes, es de esperarse.
- No obstante, las imagenes creadas por el algoritmo no son buenas, tienden a generar una forma semejante de las clases, pero les falta mucha definicion.

==

4. Modelo Final

Probamos una nueva Arquitectura para el Discriminados y el Generador agregando mas capas, dropout y batch_normalization.

```
discriminator = keras.Sequential(
    [
        # Capa de entrada
        keras.layers.InputLayer((28, 28, discriminator_in_channels)),

        # 1ª capa convolucional
        layers.Conv2D(64, (3, 3), strides=(2, 2), padding="same"),
        layers.LeakyReLU(negative_slope=0.2),
        layers.BatchNormalization(),
        layers.Dropout(0.2),
```

```
        # 2ª capa convolucional
        layers.Conv2D(128, (3, 3), strides=(2, 2), padding="same"),
        layers.LeakyReLU(negative_slope=0.2),
        layers.BatchNormalization(),
        layers.Dropout(0.2),

        # 3ª capa convolucional
        layers.Conv2D(256, (3, 3), strides=(2, 2), padding="same"),
        layers.LeakyReLU(negative_slope=0.2),
        layers.BatchNormalization(),
        layers.Dropout(0.2),

        # Global Max Pooling
        layers.GlobalMaxPooling2D(),

        # Capa densa adicional
        layers.Dense(128),
        layers.LeakyReLU(negative_slope=0.2),
        layers.Dropout(0.2),

        # Capa de salida binaria
        layers.Dense(1, activation='sigmoid'),
    ],
    name="discriminator",
)
discriminator.summary()
```

Model: "discriminator"

Layer (type)	Output Shape	Param #
conv2d_3 (Conv2D)	(None, 14, 14, 64)	6,400
leaky_re_lu_5 (LeakyReLU)	(None, 14, 14, 64)	0
batch_normalization (BatchNormalization)	(None, 14, 14, 64)	256
dropout (Dropout)	(None, 14, 14, 64)	0
conv2d_4 (Conv2D)	(None, 7, 7, 128)	73,856
leaky_re_lu_6 (LeakyReLU)	(None, 7, 7, 128)	0
batch_normalization_1 (BatchNormalization)	(None, 7, 7, 128)	512
dropout_1 (Dropout)	(None, 7, 7, 128)	0
conv2d_5 (Conv2D)	(None, 4, 4, 256)	295,168
leaky_re_lu_7 (LeakyReLU)	(None, 4, 4, 256)	0
batch_normalization_2 (BatchNormalization)	(None, 4, 4, 256)	1,024
dropout_2 (Dropout)	(None, 4, 4, 256)	0
global_max_pooling2d_1 (GlobalMaxPooling2D)	(None, 256)	0
dense_2 (Dense)	(None, 128)	32,896
leaky_re_lu_8 (LeakyReLU)	(None, 128)	0
dropout_3 (Dropout)	(None, 128)	0
dense_3 (Dense)	(None, 1)	129

Total params: 410,241 (1.56 MB)
Trainable params: 409,345 (1.56 MB)

```
# Creando el generador.
generator = keras.Sequential(
    [
        # Capa de entrada con el número de entradas calculado anteriormente
        keras.layers.InputLayer((generator_in_channels,)),

        # Primera capa densa para expandir el espacio latente
        layers.Dense(7 * 7 * generator_in_channels),
        layers.LeakyReLU(negative_slope=0.2),
        layers.Reshape((7, 7, generator_in_channels)),

        # Primera capa Conv2DTranspose para expandir la imagen a 14x14
        layers.Conv2DTranspose(256, (5, 5), strides=(2, 2), padding="same"),
        layers.BatchNormalization(),  # Normalización para mejorar estabilidad
        layers.LeakyReLU(negative_slope=0.2),
```

```
        # Segunda capa Conv2DTranspose para expandir la imagen a 28x28
        layers.Conv2DTranspose(128, (4, 4), strides=(2, 2), padding="same"),
        layers.BatchNormalization(),  # Normalización
        layers.LeakyReLU(negative_slope=0.2),

        # Añadimos una tercera capa transpuesta para generar más detalle
        layers.Conv2DTranspose(64, (3, 3), padding="same"),
        layers.LeakyReLU(negative_slope=0.2),

        # Capa de salida para generar la imagen con un solo canal (grises)
        layers.Conv2D(num_channels, (7, 7), padding="same", activation="sigmoid")
    ],
    name="generator",
)

generator.summary()
```

Model: "generator"

Layer (type)	Output Shape	Param #
dense_6 (Dense)	(None, 6762)	939,918
leaky_re_lu_17 (LeakyReLU)	(None, 6762)	0
reshape_3 (Reshape)	(None, 7, 7, 138)	0
conv2d_transpose_8 (Conv2DTranspose)	(None, 14, 14, 256)	883,456
batch_normalization_7 (BatchNormalization)	(None, 14, 14, 256)	1,024
leaky_re_lu_18 (LeakyReLU)	(None, 14, 14, 256)	0
conv2d_transpose_9 (Conv2DTranspose)	(None, 28, 28, 128)	524,416
batch_normalization_8 (BatchNormalization)	(None, 28, 28, 128)	512
leaky_re_lu_19 (LeakyReLU)	(None, 28, 28, 128)	0
conv2d_transpose_10 (Conv2DTranspose)	(None, 28, 28, 64)	73,792
leaky_re_lu_20 (LeakyReLU)	(None, 28, 28, 64)	0
conv2d_8 (Conv2D)	(None, 28, 28, 1)	3,137

Total params: 2,426,255 (9.26 MB)
Trainable params: 2,425,487 (9.25 MB)

```
class ConditionalGAN(keras.Model):
    def __init__(self, discriminator, generator, latent_dim):
        super().__init__()
        self.discriminator = discriminator
        self.generator = generator
        self.latent_dim = latent_dim
        self.seed_generator = keras.random.SeedGenerator(1337)
        self.gen_loss_tracker = keras.metrics.Mean(name="generator_loss")
        self.disc_loss_tracker = keras.metrics.Mean(name="discriminator_loss")

    @property
    def metrics(self):
        return [self.gen_loss_tracker, self.disc_loss_tracker]

    def compile(self, d_optimizer, g_optimizer, loss_fn):
        super().compile()
        self.d_optimizer = d_optimizer
        self.g_optimizer = g_optimizer
        self.loss_fn = loss_fn

    def train_step(self, data):
        # Unpack the data.
        real_images, one_hot_labels = data

        # Add dummy dimensions to the labels so that they can be concatenated with
        # the images. This is for the discriminator.
        image_one_hot_labels = one_hot_labels[:, :, None, None]
        image_one_hot_labels = ops.repeat(
            image_one_hot_labels, repeats=[image_size * image_size]
        )
        image_one_hot_labels = ops.reshape(
            image_one_hot_labels, (-1, image_size, image_size, num_classes)
        )

        # Sample random points in the latent space and concatenate the labels.
        # This is for the generator.
```

```
        batch_size = ops.shape(real_images)[0]
        random_latent_vectors = keras.random.normal(
            shape=(batch_size, self.latent_dim), seed=self.seed_generator
        )
        random_vector_labels = ops.concatenate(
            [random_latent_vectors, one_hot_labels], axis=1
        )

        # Decode the noise (guided by labels) to fake images.
        generated_images = self.generator(random_vector_labels)

        # Combine them with real images. Note that we are concatenating the labels
        # with these images here.
        fake_image_and_labels = ops.concatenate(
            [generated_images, image_one_hot_labels], -1
        )
        real_image_and_labels = ops.concatenate([real_images, image_one_hot_labels], -1)
        combined_images = ops.concatenate(
            [fake_image_and_labels, real_image_and_labels], axis=0
        )

        # Assemble labels discriminating real from fake images.
        labels = ops.concatenate(
            [ops.ones((batch_size, 1)), ops.zeros((batch_size, 1))], axis=0
        )

        # Train the discriminator.
        with tf.GradientTape() as tape:
            predictions = self.discriminator(combined_images)
            d_loss = self.loss_fn(labels, predictions)
        grads = tape.gradient(d_loss, self.discriminator.trainable_weights)
        self.d_optimizer.apply_gradients(
            zip(grads, self.discriminator.trainable_weights)
        )

        # Sample random points in the latent space.
        random_latent_vectors = keras.random.normal(
            shape=(batch_size, self.latent_dim), seed=self.seed_generator
        )
        random_vector_labels = ops.concatenate(
            [random_latent_vectors, one_hot_labels], axis=1
        )

        # Assemble labels that say "all real images".
        misleading_labels = ops.zeros((batch_size, 1))

        # Train the generator (note that we should *not* update the weights
        # of the discriminator)!
        with tf.GradientTape() as tape:
            fake_images = self.generator(random_vector_labels)
            fake_image_and_labels = ops.concatenate(
                [fake_images, image_one_hot_labels], -1
            )
            predictions = self.discriminator(fake_image_and_labels)
            g_loss = self.loss_fn(misleading_labels, predictions)
        grads = tape.gradient(g_loss, self.generator.trainable_weights)
        self.g_optimizer.apply_gradients(zip(grads, self.generator.trainable_weights))

        # Monitor loss.
        self.gen_loss_tracker.update_state(g_loss)
        self.disc_loss_tracker.update_state(d_loss)
        return {
            "g_loss": self.gen_loss_tracker.result(),
            "d_loss": self.disc_loss_tracker.result(),
        }
```

Se crea la clase cond_gan, se compila y se entrena el modelo

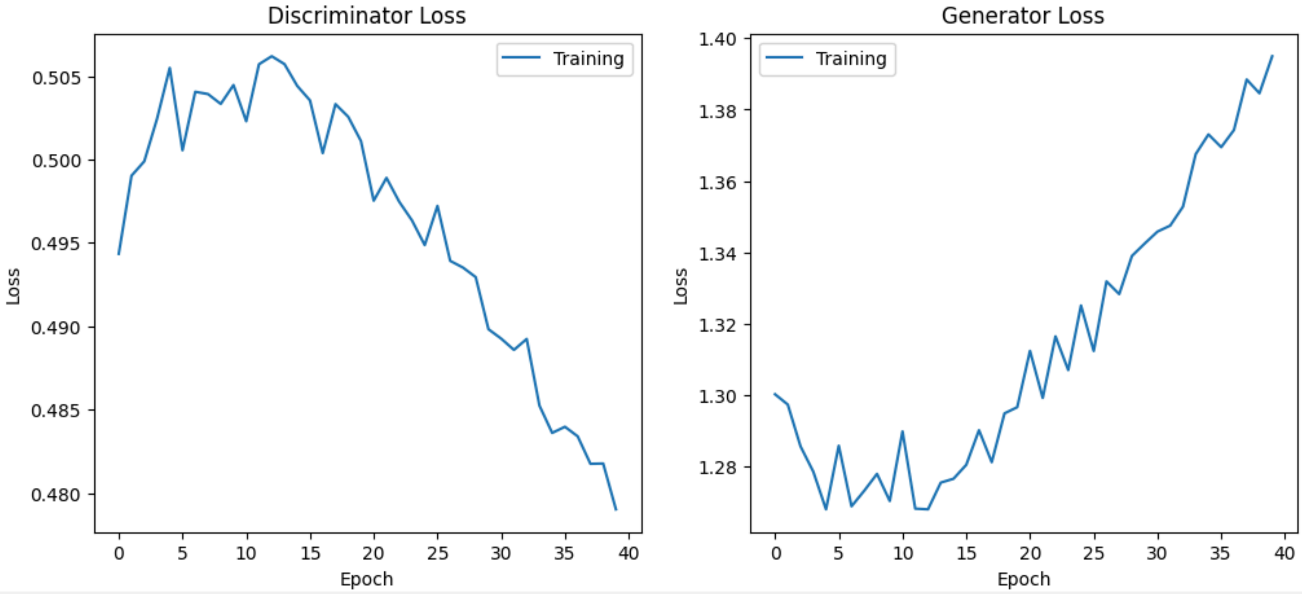
```
d_loss = []
g_loss = []

cond_gan = ConditionalGAN(
    discriminator=discriminator, generator=generator, latent_dim=latent_dim
)

cond_gan.compile(
    d_optimizer=keras.optimizers.Adam(learning_rate=0.0002, decay=1e-6),
    g_optimizer=keras.optimizers.Adam(learning_rate=0.0002, decay=1e-6),
    #d_optimizer=keras.optimizers.RMSprop(learning_rate=0.0002, decay=1e-6),
    #g_optimizer=keras.optimizers.RMSprop(learning_rate=0.0002, decay=1e-6),
    loss_fn=keras.losses.BinaryCrossentropy(from_logits=True),
)
```

```
# Guarda las pérdidas en cada época
for epoch in range(40):
    print("Epoch: ", epoch+1)
    history = cond_gan.fit(dataset, epochs=1, verbose=1)
    d_loss.append(history.history['d_loss'][0])
    g_loss.append(history.history['g_loss'][0])
```

```
Epoch:  11
1094/1094 ━━━━━━━━━━━━━━━━━━━━ 7s 6ms/step - d_loss: 0.5018 - g_loss: 1.2891
Epoch:  12
1094/1094 ━━━━━━━━━━━━━━━━━━━━ 7s 6ms/step - d_loss: 0.5054 - g_loss: 1.2611
Epoch:  13
1094/1094 ━━━━━━━━━━━━━━━━━━━━ 7s 6ms/step - d_loss: 0.5080 - g_loss: 1.2541
Epoch:  14
1094/1094 ━━━━━━━━━━━━━━━━━━━━ 7s 6ms/step - d_loss: 0.5065 - g_loss: 1.2692
Epoch:  15
1094/1094 ━━━━━━━━━━━━━━━━━━━━ 7s 6ms/step - d_loss: 0.5056 - g_loss: 1.2781
Epoch:  16
1094/1094 ━━━━━━━━━━━━━━━━━━━━ 7s 6ms/step - d_loss: 0.5035 - g_loss: 1.2777
Epoch:  17
1094/1094 ━━━━━━━━━━━━━━━━━━━━ 7s 6ms/step - d_loss: 0.5011 - g_loss: 1.2853
Epoch:  18
1094/1094 ━━━━━━━━━━━━━━━━━━━━ 7s 6ms/step - d_loss: 0.5053 - g_loss: 1.2797
Epoch:  19
1094/1094 ━━━━━━━━━━━━━━━━━━━━ 7s 6ms/step - d_loss: 0.5053 - g_loss: 1.2894
Epoch:  20
1094/1094 ━━━━━━━━━━━━━━━━━━━━ 7s 6ms/step - d_loss: 0.5044 - g_loss: 1.2900
Epoch:  21
1094/1094 ━━━━━━━━━━━━━━━━━━━━ 7s 7ms/step - d_loss: 0.4971 - g_loss: 1.3069
Epoch:  22
1094/1094 ━━━━━━━━━━━━━━━━━━━━ 7s 6ms/step - d_loss: 0.5008 - g_loss: 1.2948
Epoch:  23
1094/1094 ━━━━━━━━━━━━━━━━━━━━ 7s 6ms/step - d_loss: 0.4989 - g_loss: 1.3169
Epoch:  24
1094/1094 ━━━━━━━━━━━━━━━━━━━━ 7s 6ms/step - d_loss: 0.4962 - g_loss: 1.3029
Epoch:  25
1094/1094 ━━━━━━━━━━━━━━━━━━━━ 7s 6ms/step - d_loss: 0.4967 - g_loss: 1.3181
Epoch:  26
1094/1094 ━━━━━━━━━━━━━━━━━━━━ 7s 6ms/step - d_loss: 0.4982 - g_loss: 1.3133
Epoch:  27
1094/1094 ━━━━━━━━━━━━━━━━━━━━ 7s 6ms/step - d_loss: 0.4973 - g_loss: 1.3142
Epoch:  28
1094/1094 ━━━━━━━━━━━━━━━━━━━━ 7s 6ms/step - d_loss: 0.4946 - g_loss: 1.3284
Epoch:  29
1094/1094 ━━━━━━━━━━━━━━━━━━━━ 7s 6ms/step - d_loss: 0.4933 - g_loss: 1.3364
Epoch:  30
1094/1094 ━━━━━━━━━━━━━━━━━━━━ 7s 6ms/step - d_loss: 0.4909 - g_loss: 1.3325
Epoch:  31
1094/1094 ━━━━━━━━━━━━━━━━━━━━ 7s 6ms/step - d_loss: 0.4896 - g_loss: 1.3546
Epoch:  32
1094/1094 ━━━━━━━━━━━━━━━━━━━━ 7s 6ms/step - d_loss: 0.4893 - g_loss: 1.3473
Epoch:  33
1094/1094 ━━━━━━━━━━━━━━━━━━━━ 7s 7ms/step - d_loss: 0.4899 - g_loss: 1.3501
Epoch:  34
1094/1094 ━━━━━━━━━━━━━━━━━━━━ 7s 6ms/step - d_loss: 0.4862 - g_loss: 1.3622
Epoch:  35
1094/1094 ━━━━━━━━━━━━━━━━━━━━ 7s 6ms/step - d_loss: 0.4857 - g_loss: 1.3617
Epoch:  36
1094/1094 ━━━━━━━━━━━━━━━━━━━━ 7s 6ms/step - d_loss: 0.4866 - g_loss: 1.3657
Epoch:  37
1094/1094 ━━━━━━━━━━━━━━━━━━━━ 7s 6ms/step - d_loss: 0.4837 - g_loss: 1.3702
Epoch:  38
1094/1094 ━━━━━━━━━━━━━━━━━━━━ 7s 6ms/step - d_loss: 0.4821 - g_loss: 1.3893
Epoch:  39
1094/1094 ━━━━━━━━━━━━━━━━━━━━ 7s 6ms/step - d_loss: 0.4849 - g_loss: 1.3786
```

Graficamos las funciones de perdida del Generados y Discriminador.

```
# Crear la figura y los subplots
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 5))

# Gráfico para d_loss
ax1.plot(d_loss, label='Training')  # Línea para datos de entrenamiento
ax1.set_title('Discriminator Loss')
ax1.set_xlabel('Epoch')
ax1.set_ylabel('Loss')
ax1.legend()

# Gráfico para g_loss
ax2.plot(g_loss, label='Training')  # Línea para datos de entrenamiento
ax2.set_title('Generator Loss')
ax2.set_xlabel('Epoch')
ax2.set_ylabel('Loss')
ax2.legend()

# Mostrar los gráficos
plt.show()
```



Comentario El grafico muestra que al comienzo la funcion de perdida del discriminador aumenta, para luego comenzar a decrecer, al opuesto de lo que hace la funcion del Generador.
La funcion de perdida generador, aunque tiende a aumentar, no lo hace en un rango muy grande. Pareceido a lo que hace el discriminador. De la grafica podemos inferir que el discriminador cada vez logra hacerse mejor para discriminar imagenes que no son verdaderas.

```
# Etiquetas a generar. Modificar la lista labels como queramos
labels = [0,1,2,3,2,2,4,5,6,7,8,9]
n_samples = len(labels)

# Extraemos el generador de la CGAN
trained_gen = cond_gan.generator

# Convertimos las etiquetas en labels a categóricas
labels = keras.utils.to_categorical(labels, num_classes)

# Generamos el ruido para las diferentes imágenes a generar.
noise = keras.random.normal(shape=(n_samples, latent_dim))
noise = ops.reshape(noise, (n_samples, latent_dim))

# Concatenamos el ruido y las etiquetas para tener el input entero del generador
noise_and_labels = ops.concatenate([noise, labels], 1)

# Generamos las imágenes con el input
fake_images = trained_gen.predict(noise_and_labels)

# Convertimos a las dimensiones originales (28x28, aunque podríamos modificar los valores) y los valores de los píxeles yendo de 0 a 255 en
fake_images *= 255.0
converted_images = fake_images.astype(np.uint8)
converted_images = ops.image.resize(converted_images, (28, 28)).numpy().astype(np.uint8)
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 22ms/step
```

Graficamos las imagenes de ejemplo creadas por el Generador.

```
# Assuming converted_images contains your generated images
num_images = len(converted_images)
num_cols = 5  # Adjust as needed
# Calculate the actual number of rows and columns needed
num_rows = (num_images + num_cols - 1) // num_cols

fig, axes = plt.subplots(num_rows, num_cols, figsize=(15, 6))
fig.tight_layout()

# Flatten axes if necessary
if num_rows == 1 or num_cols == 1:
    axes = axes.flatten()  # Makes it work for single row or column
else:
    axes = axes.ravel()  # Makes it work for multiple rows and columns

for i, image in enumerate(converted_images):
    # Plot image on the current subplot
    if i < len(axes):
        ax = axes[i]
        ax.imshow(image[:, :, 0], cmap='gray_r')
        ax.axis('off')  # Hide axes
```

```
        # Get label name using ll_labels and class_names
        label_number = labels[i]
        label_index = np.argmax(label_number)

        # Access the label name using list indexing
        label_name = class_names[label_index]

        # Add label as title
        ax.set_title(f"Label: {label_index} ({label_name})")

# Hide any unused subplots
for i in range(num_images, num_rows * num_cols):
    if i < len(axes):
        axes[i].axis('off')

plt.show()
```

Comentarios

Ya hemos comentado el grafico de las funciones de perdida, corresponde comentar la calidad de las imagenes, la cual se observan bastante bien, incluso la de la "Sandal" que es relativamente mas dificil de reproducir.