Proyecto 2

Andre Yahir Gonzalez Cuevas

Link: https://github.com/Andr3Glez/Proyecto_GANs

Objetivos:

Comprender los principios fundamentales de las redes generativas antagónicas (GANs) y su aplicación en el aprendizaje profundo generativo. Implementar una GAN básica y explorar variantes como las GANs condicionales o las CycleGANs para tareas específicas, como la generación de imágenes, traducción de imágenes o generación de texto. Analizar el rendimiento y las características de los modelos generados por las GANs.

Descripción:

Deberán seleccionar un conjunto de datos adecuado para su proyecto, que puede ser de imágenes, texto o cualquier otro tipo que permita la aplicación de GANs. Implementar una GAN, como una GAN básica, una GAN condicional o una CycleGAN, dependiendo de la naturaleza del conjunto de datos y el objetivo del proyecto. El proyecto incluirá una fase de experimentación donde los estudiantes deberán entrenar, ajustar y evaluar sus modelos. Presentar sus resultados a través de un informe escrito y una presentación, discutiendo la implementación, los desafíos encontrados, el rendimiento de sus modelos y las aplicaciones potenciales de su trabajo.

En este proyecto se usará una base de datos con imagenes de mujeres en la vida real y de anime, el objetivo es que se puedan pasar las imagens originales a formato o estilo anime y que mantenga la eacencia de la imagen original.

```
import os

import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms
from PIL import Image

import numpy as np
import pickle as pkl
import matplotlib.pyplot as plt
from mpl_toolkits.axes_grid1 import ImageGrid

from sklearn.model_selection import train_test_split


gpu_avail = torch.cuda.is_available()
print(f"Is the GPU available? {gpu_avail}")
```

```
Is the GPU available? True
```

```
device = torch.device("cuda") if torch.cuda.is_available() else torch.device("cpu")
print("Device", device)
```

```
Device cuda
```

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

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

Datasets

En este código se define una clase llamada Dataset que será utilizada para cargar y transformar un conjunto de datos de imágenes para su uso en el entrenamiento y prueba en PyTorch.

```
class Dataset(torch.utils.data.Dataset):
    def __init__(self, img_dir):
        base_path = BASE_DATASET_PATH
        img_dir = os.path.join(base_path, img_dir)

        path_list = os.listdir(img_dir)
        abspath = os.path.abspath(img_dir)

        # Divide las rutas de imágenes en conjuntos de entrenamiento y prueba
        self.train_paths, self.test_paths = train_test_split([os.path.join(abspath, path) for path in path_list], test_size=0.2, random_

        self.transform = transforms.Compose([
            transforms.Resize(IMG_SIZE),
            transforms.ToTensor(),
            transforms.Normalize([0.5, 0.5, 0.5], [0.5, 0.5, 0.5]), # Normalizar la imagen entre -1 y 1
        ])

    def __len__(self):
        return len(self.train_paths)

    def __getitem__(self, idx):
        path = self.train_paths[idx]
        img = Image.open(path).convert('RGB')

        img_tensor = self.transform(img)
        return img_tensor
```

Discriminador

En la inicialización (**init**), se define la estructura del discriminador. super(Discriminator, self).**init**() llama al constructor de la clase base nn.Module para inicializar el objeto Discriminator.

El parámetro conv_dim se utiliza para definir la dimensionalidad de los canales de características en las capas convolucionales.

Se definen una serie de capas convolucionales, cada una seguida de una función de activación LeakyReLU y, en la mayoría de las capas, una capa de normalización de instancia (nn.InstanceNorm2d). Estas capas convolucionales reducen progresivamente la dimensión espacial de las características mientras aumentan la profundidad de los canales.

La última capa convolucional produce una salida con un solo canal, que se utiliza para clasificar si la entrada es real o falsa.

El método forward define cómo se propaga la entrada a través del modelo. Toma un tensor de entrada x que representa una imagen. Este tensor se pasa a través de las capas definidas en self.main.

Luego de pasar por las capas convolucionales, se aplica una operación de promediado de la agrupación (F.avg_pool2d) sobre la salida para reducir las dimensiones espaciales a un solo valor por canal.

El tensor resultante se aplana (torch.flatten) para convertirlo en un tensor unidimensional.

Finalmente, este tensor se devuelve como salida del discriminador.

```
class Discriminator(nn.Module):
    def __init__(self,conv_dim=32):
        super(Discriminator, self).__init__()

        self.main = nn.Sequential(
            nn.Conv2d(3, conv_dim, 4, stride=2, padding=1),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(conv_dim, conv_dim*2, 4, stride=2, padding=1),
            nn.InstanceNorm2d(conv_dim*2),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(conv_dim*2, conv_dim*4, 4, stride=2, padding=1),
            nn.InstanceNorm2d(conv_dim*4),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(conv_dim*4, conv_dim*8, 4, padding=1),
            nn.InstanceNorm2d(conv_dim*8),
            nn.LeakyReLU(0.2, inplace=True),

            nn.Conv2d(conv_dim*8, 1, 4, padding=1),
        )

    def forward(self, x):
        x = self.main(x)
        x = F.avg_pool2d(x, x.size()[2:])
        x = torch.flatten(x, 1)
        return x
```

ResidualBlock

El ResidualBlock ayuda a abordar el problema del desvanecimiento del gradiente y permite entrenar modelos más profundos de manera más efectiva.

El bloque residual consiste en dos convoluciones de 3x3 (ambas con un relleno de 1 píxel para mantener el tamaño) aplicadas a la entrada in_channels, seguidas de funciones de normalización de instancia (nn.InstanceNorm2d) y activaciones ReLU.

Esto permite que el bloque aprenda las diferencias entre la entrada y la salida, en lugar de aprender directamente la representación de la salida.

```
class ResidualBlock(nn.Module):
    def __init__(self, in_channels):
        super(ResidualBlock, self).__init__()

        self.main = nn.Sequential(
            nn.ReflectionPad2d(1),
            nn.Conv2d(in_channels, in_channels, 3),
            nn.InstanceNorm2d(in_channels),
            nn.ReLU(inplace=True),
            nn.ReflectionPad2d(1),
            nn.Conv2d(in_channels, in_channels, 3),
            nn.InstanceNorm2d(in_channels)
        )

    def forward(self, x):
        return x + self.main(x)
```

Generador

El generador toma un tensor de ruido como entrada y produce una imagen generada como salida. La estructura del generador consiste en capas convolucionales, bloques residuales y capas de convolución transpuesta para transformar el ruido de entrada en una imagen generada de alta calidad.

El generador comienza con una capa de reflexión (nn.ReflectionPad2d) seguida de una convolución de 7x7 que aumenta la dimensionalidad de las características de entrada a conv_dim. Después de esto, se aplica una normalización de instancia y una activación ReLU.

Se siguen dos capas convolucionales, que reducen las dimensiones espaciales de las características mientras aumentan su profundidad.

Se agregan 9 bloques residuales (ResidualBlock) para aprender las características residuales de la imagen.

Termina con una capa de reflexión, una convolución de 7x7 y una función de activación Tanh, que escala los valores a un rango de [-1, 1]

```
class Generator(nn.Module):
    def __init__(self, conv_dim=64, n_res_block=9):
        super(Generator, self).__init__()
        self.main = nn.Sequential(
            nn.ReflectionPad2d(3),
            nn.Conv2d(3, conv_dim, 7),
            nn.InstanceNorm2d(64),
            nn.ReLU(inplace=True),

            nn.Conv2d(conv_dim, conv_dim*2, 3, stride=2, padding=1),
            nn.InstanceNorm2d(conv_dim*2),
            nn.ReLU(inplace=True),
            nn.Conv2d(conv_dim*2, conv_dim*4, 3, stride=2, padding=1),
            nn.InstanceNorm2d(conv_dim*4),
            nn.ReLU(inplace=True),

            ResidualBlock(conv_dim*4),
            ResidualBlock(conv_dim*4),
            ResidualBlock(conv_dim*4),
            ResidualBlock(conv_dim*4),
            ResidualBlock(conv_dim*4),
            ResidualBlock(conv_dim*4),
            ResidualBlock(conv_dim*4),
            ResidualBlock(conv_dim*4),
            ResidualBlock(conv_dim*4),

            nn.ConvTranspose2d(conv_dim*4, conv_dim*2, 3, stride=2, padding=1, output_padding=1),
            nn.InstanceNorm2d(conv_dim*2),
            nn.ReLU(inplace=True),
            nn.ConvTranspose2d(conv_dim*2, conv_dim, 3, stride=2, padding=1, output_padding=1),
            nn.InstanceNorm2d(conv_dim),
            nn.ReLU(inplace=True),

            nn.ReflectionPad2d(3),
            nn.Conv2d(conv_dim, 3, 7),
            nn.Tanh()
        )

    def forward(self, x):
        return self.main(x)
```

CycleGAN

Esta clase implementa el ciclo de entrenamiento completo para el modelo CycleGAN, incluyendo la definición de arquitecturas de red, el cálculo de las pérdidas y el ciclo de entrenamiento.

Para las funciones de perdida se usaran varias para poder tener una mejor idea de la perdida, esto se hizo basado en el AnimeGans.

real_mse_loss calcula la pérdida cuadrática media entre las salidas del discriminador y el valor de referencia "real" (1 para imágenes reales).

fake_mse_loss calcula la pérdida cuadrática media entre las salidas del discriminador y el valor de referencia "falso" (0 para imágenes falsas).

cycle_consistency_loss calcula la pérdida de consistencia cíclica entre la imagen original y la imagen reconstruida.

```
class CycleGAN:

    def __init__(self, g_conv_dim=64, d_conv_dim=64, n_res_block=6):
        self.device = torch.device('cuda') if torch.cuda.is_available() else torch.device("cpu")

        self.G_XtoY = Generator(conv_dim=g_conv_dim, n_res_block=n_res_block).to(self.device)
        self.G_YtoX = Generator(conv_dim=g_conv_dim, n_res_block=n_res_block).to(self.device)

        self.D_X = Discriminator(conv_dim=d_conv_dim).to(self.device)
        self.D_Y = Discriminator(conv_dim=d_conv_dim).to(self.device)

        print(f"Models running of {self.device}")

    def load_model(self, filename):
        save_filename = os.path.splitext(os.path.basename(filename))[0] + '.pt'
        return torch.load(save_filename)

    def real_mse_loss(self, D_out):
        return torch.mean((D_out-1)**2)


    def fake_mse_loss(self, D_out):
        return torch.mean(D_out**2)


    def cycle_consistency_loss(self, real_img, reconstructed_img, lambda_weight):
        reconstr_loss = torch.mean(torch.abs(real_img - reconstructed_img))
        return lambda_weight*reconstr_loss


    def train_generator(self, optimizers, images_x, images_y):
        # Generator YtoX
        optimizers["g_optim"].zero_grad()

        fake_images_x = self.G_YtoX(images_y)

        d_real_x = self.D_X(fake_images_x)
        g_YtoX_loss = self.real_mse_loss(d_real_x)

        recon_y = self.G_XtoY(fake_images_x)
        recon_y_loss = self.cycle_consistency_loss(images_y, recon_y, lambda_weight=10)


        # Generator XtoY
        fake_images_y = self.G_XtoY(images_x)

        d_real_y = self.D_Y(fake_images_y)
        g_XtoY_loss = self.real_mse_loss(d_real_y)

        recon_x = self.G_YtoX(fake_images_y)
        recon_x_loss = self.cycle_consistency_loss(images_x, recon_x, lambda_weight=10)

        g_total_loss = g_YtoX_loss + g_XtoY_loss + recon_y_loss + recon_x_loss
        g_total_loss.backward()
        optimizers["g_optim"].step()

        return g_total_loss.item()


    def train_discriminator(self, optimizers, images_x, images_y):
        # Discriminator x
        optimizers["d_x_optim"].zero_grad()

        d_real_x = self.D_X(images_x)
        d_real_loss_x = self.real_mse_loss(d_real_x)

        fake_images_x = self.G_YtoX(images_y)

        d_fake_x = self.D_X(fake_images_x)
        d_fake_loss_x = self.fake_mse_loss(d_fake_x)

        d_x_loss = d_real_loss_x + d_fake_loss_x
        d_x_loss.backward()
        optimizers["d_x_optim"].step()


        # Discriminator y
        optimizers["d_y_optim"].zero_grad()

        d_real_y = self.D_Y(images_y)
        d_real_loss_x = self.real_mse_loss(d_real_y)
```

```
        fake_images_y = self.G_XtoY(images_x)

        d_fake_y = self.D_Y(fake_images_y)
        d_fake_loss_y = self.fake_mse_loss(d_fake_y)

        d_y_loss = d_real_loss_x + d_fake_loss_y
        d_y_loss.backward()
        optimizers["d_y_optim"].step()

        return d_x_loss.item(), d_y_loss.item()


    def train(self, optimizers, data_loader_x, data_loader_y, print_every=10, sample_every=100):
        losses = []
        g_total_loss_min = np.Inf

        fixed_x = next(iter(data_loader_x))[1].to(self.device)
        fixed_y = next(iter(data_loader_y))[1].to(self.device)

        print(f'Running on {self.device}')
        for epoch in range(EPOCHS):
            for (images_x, images_y) in zip(data_loader_x, data_loader_y):
                images_x, images_y = images_x.to(self.device), images_y.to(self.device)

                g_total_loss = self.train_generator(optimizers, images_x, images_y)
                d_x_loss, d_y_loss = self.train_discriminator(optimizers, images_x, images_y)


            if epoch % print_every == 0:
                losses.append((d_x_loss, d_y_loss, g_total_loss))
                print('Epoch [{:5d}/{:5d}] | d_X_loss: {:6.4f} | d_Y_loss: {:6.4f} | g_total_loss: {:6.4f}'
                .format(
                    epoch,
                    EPOCHS,
                    d_x_loss,
                    d_y_loss,
                    g_total_loss
                ))

            if g_total_loss < g_total_loss_min:
                g_total_loss_min = g_total_loss

                torch.save(self.G_XtoY.state_dict(), "G_X2Y")
                torch.save(self.G_YtoX.state_dict(), "G_Y2X")

                torch.save(self.D_X.state_dict(), "D_X")
                torch.save(self.D_Y.state_dict(), "D_Y")

                print("Models Saved")



        return losses
```

Configuracion de las clases

N_WORKERS: Es el número de subprocesos que se utilizarán para cargar los datos. Aquí se ha establecido en 0, lo que significa que no se usarán subprocesos adicionales.

IMG_SIZE: Es el tamaño al que se redimensionarán las imágenes. Aquí se ha establecido en 64x64 píxeles.

LR (tasa de aprendizaje), BETA1 y BETA2: Son los parámetros de optimización utilizados en el optimizador Adam para el entrenamiento de la red. La tasa de aprendizaje (LR) se ha establecido en 0.0002, y BETA1 y BETA2 son los parámetros de decaimiento de momento y la tasa de decaimiento de momento cuadrático respectivamente, ambos establecidos en 0.5 y 0.999.

```
BASE_DATASET_PATH = "/content/drive/MyDrive/Deep_Learning/Proyecto_2"
X_DATASET = "selected_faces"
Y_DATASET = "selected_anime"

BATCH_SIZE = 32
N_WORKERS = 0

IMG_SIZE = 64
LR = 0.0002
BETA1 = 0.5
BETA2 = 0.999

EPOCHS = 50
```

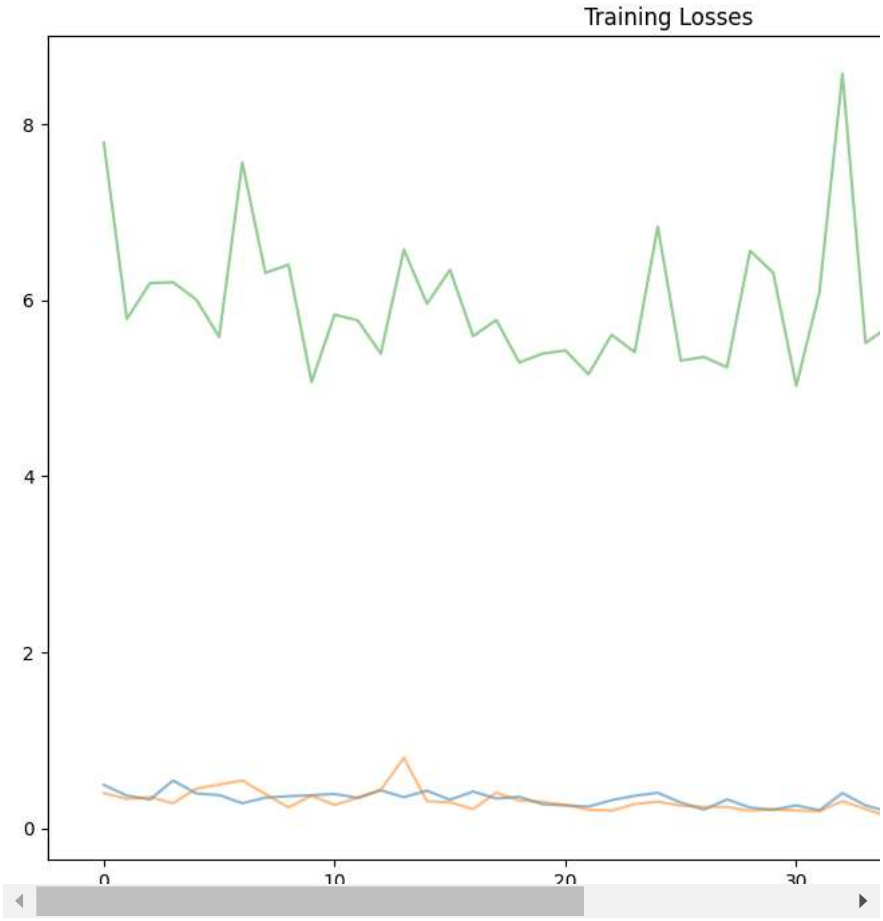
Train

```
# Train
x_train_dataset = Dataset(X_DATASET)
y_train_dataset = Dataset(Y_DATASET)

data_loader_x_train = DataLoader(x_train_dataset, BATCH_SIZE, shuffle=True, num_workers=N_WORKERS)
data_loader_y_train = DataLoader(y_train_dataset, BATCH_SIZE, shuffle=True, num_workers=N_WORKERS)


# Model
cycleGan = CycleGAN()

# Oprimizer
g_params = list(cycleGan.G_XtoY.parameters()) + list(cycleGan.G_YtoX.parameters())

optimizers = {
    "g_optim": optim.Adam(g_params, LR, [BETA1, BETA2]),
    "d_x_optim": optim.Adam(cycleGan.D_X.parameters(), LR, [BETA1, BETA2]),
    "d_y_optim": optim.Adam(cycleGan.D_Y.parameters(), LR, [BETA1, BETA2])
}

# Train
losses = cycleGan.train(optimizers, data_loader_x_train, data_loader_y_train, print_every=1)
```

```
Models running of cuda
Running on cuda
Epoch [    0/   50] | d_X_loss: 0.4953 | d_Y_loss: 0.4028 | g_total_loss: 7.7897
Models Saved
Epoch [    1/   50] | d_X_loss: 0.3744 | d_Y_loss: 0.3370 | g_total_loss: 5.7881
Models Saved
Epoch [    2/   50] | d_X_loss: 0.3296 | d_Y_loss: 0.3561 | g_total_loss: 6.1942
Epoch [    3/   50] | d_X_loss: 0.5455 | d_Y_loss: 0.2882 | g_total_loss: 6.2037
Epoch [    4/   50] | d_X_loss: 0.3995 | d_Y_loss: 0.4517 | g_total_loss: 6.0089
Epoch [    5/   50] | d_X_loss: 0.3799 | d_Y_loss: 0.4995 | g_total_loss: 5.5826
Models Saved
Epoch [    6/   50] | d_X_loss: 0.2875 | d_Y_loss: 0.5457 | g_total_loss: 7.5640
Epoch [    7/   50] | d_X_loss: 0.3513 | d_Y_loss: 0.3914 | g_total_loss: 6.3097
Epoch [    8/   50] | d_X_loss: 0.3675 | d_Y_loss: 0.2400 | g_total_loss: 6.4039
Epoch [    9/   50] | d_X_loss: 0.3779 | d_Y_loss: 0.3740 | g_total_loss: 5.0727
Models Saved
Epoch [   10/   50] | d_X_loss: 0.3935 | d_Y_loss: 0.2675 | g_total_loss: 5.8366
Epoch [   11/   50] | d_X_loss: 0.3480 | d_Y_loss: 0.3518 | g_total_loss: 5.7711
Epoch [   12/   50] | d_X_loss: 0.4344 | d_Y_loss: 0.4411 | g_total_loss: 5.3944
Epoch [   13/   50] | d_X_loss: 0.3562 | d_Y_loss: 0.8032 | g_total_loss: 6.5757
Epoch [   14/   50] | d_X_loss: 0.4300 | d_Y_loss: 0.3091 | g_total_loss: 5.9599
Epoch [   15/   50] | d_X_loss: 0.3270 | d_Y_loss: 0.2982 | g_total_loss: 6.3457
Epoch [   16/   50] | d_X_loss: 0.4204 | d_Y_loss: 0.2219 | g_total_loss: 5.5932
Epoch [   17/   50] | d_X_loss: 0.3404 | d_Y_loss: 0.4075 | g_total_loss: 5.7752
Epoch [   18/   50] | d_X_loss: 0.3607 | d_Y_loss: 0.3188 | g_total_loss: 5.2931
Epoch [   19/   50] | d_X_loss: 0.2768 | d_Y_loss: 0.3037 | g_total_loss: 5.3938
Epoch [   20/   50] | d_X_loss: 0.2623 | d_Y_loss: 0.2706 | g_total_loss: 5.4294
Epoch [   21/   50] | d_X_loss: 0.2497 | d_Y_loss: 0.2124 | g_total_loss: 5.1597
Epoch [   22/   50] | d_X_loss: 0.3227 | d_Y_loss: 0.2032 | g_total_loss: 5.6077
Epoch [   23/   50] | d_X_loss: 0.3722 | d_Y_loss: 0.2771 | g_total_loss: 5.4109
Epoch [   24/   50] | d_X_loss: 0.4066 | d_Y_loss: 0.3055 | g_total_loss: 6.8348
Epoch [   25/   50] | d_X_loss: 0.2942 | d_Y_loss: 0.2604 | g_total_loss: 5.3136
Epoch [   26/   50] | d_X_loss: 0.2151 | d_Y_loss: 0.2459 | g_total_loss: 5.3562
Epoch [   27/   50] | d_X_loss: 0.3308 | d_Y_loss: 0.2453 | g_total_loss: 5.2394
Epoch [   28/   50] | d_X_loss: 0.2377 | d_Y_loss: 0.2023 | g_total_loss: 6.5622
Epoch [   29/   50] | d_X_loss: 0.2145 | d_Y_loss: 0.2155 | g_total_loss: 6.3109
Epoch [   30/   50] | d_X_loss: 0.2631 | d_Y_loss: 0.2048 | g_total_loss: 5.0278
Models Saved
Epoch [   31/   50] | d_X_loss: 0.2079 | d_Y_loss: 0.1963 | g_total_loss: 6.0883
Epoch [   32/   50] | d_X_loss: 0.4033 | d_Y_loss: 0.3117 | g_total_loss: 8.5739
Epoch [   33/   50] | d_X_loss: 0.2629 | d_Y_loss: 0.2208 | g_total_loss: 5.5151
Epoch [   34/   50] | d_X_loss: 0.1914 | d_Y_loss: 0.1346 | g_total_loss: 5.7003
Epoch [   35/   50] | d_X_loss: 0.1909 | d_Y_loss: 0.1844 | g_total_loss: 5.6493
Epoch [   36/   50] | d_X_loss: 0.1750 | d_Y_loss: 0.1966 | g_total_loss: 5.4919
Epoch [   37/   50] | d_X_loss: 0.2537 | d_Y_loss: 0.1763 | g_total_loss: 5.5090
Epoch [   38/   50] | d_X_loss: 0.3285 | d_Y_loss: 0.1500 | g_total_loss: 6.5379
```

```
Epoch [   39/   50] | d_X_loss: 0.2345 | d_Y_loss: 0.2542 | g_total_loss: 6.2636
Epoch [   40/   50] | d_X_loss: 0.1597 | d_Y_loss: 0.5447 | g_total_loss: 4.7719
Models Saved
Epoch [   41/   50] | d_X_loss: 0.3529 | d_Y_loss: 0.2940 | g_total_loss: 5.9907
Epoch [   42/   50] | d_X_loss: 0.2107 | d_Y_loss: 0.1516 | g_total_loss: 5.4665
Epoch [   43/   50] | d_X_loss: 0.1171 | d_Y_loss: 0.1104 | g_total_loss: 5.7140
Epoch [   44/   50] | d_X_loss: 0.2438 | d_Y_loss: 0.0692 | g_total_loss: 5.2335
Epoch [   45/   50] | d_X_loss: 0.1692 | d_Y_loss: 0.2139 | g_total_loss: 6.3099
Epoch [   46/   50] | d_X_loss: 0.2375 | d_Y_loss: 0.1010 | g_total_loss: 6.4381
Epoch [   47/   50] | d_X_loss: 0.1536 | d_Y_loss: 0.3687 | g_total_loss: 5.4456
Epoch [   48/   50] | d_X_loss: 0.3807 | d_Y_loss: 0.1674 | g_total_loss: 6.5404
Epoch [   49/   50] | d_X_loss: 0.0974 | d_Y_loss: 0.2257 | g_total_loss: 6.2288
```

```
# Visualizacion del comportamiento del Discriminador
fig, ax = plt.subplots(figsize=(12,8))
losses = np.array(losses)
plt.plot(losses.T[0], label='Discriminator, X', alpha=0.5)
plt.plot(losses.T[1], label='Discriminator, Y', alpha=0.5)
plt.plot(losses.T[2], label='Generators', alpha=0.5)
plt.title("Training Losses")
plt.legend()
plt.show()
```




```
# Test
x_test_dataset = Dataset(X_DATASET)
y_test_dataset = Dataset(Y_DATASET)

data_loader_x_test = DataLoader(x_test_dataset, BATCH_SIZE, shuffle=False, num_workers=N_WORKERS)
data_loader_y_test = DataLoader(y_test_dataset, BATCH_SIZE, shuffle=False, num_workers=N_WORKERS)
samples = []

for i in range(12):
    fixed_x = next(iter(data_loader_x_test))[i].to(cycleGan.device)
    fake_y = cycleGan.G_XtoY(torch.unsqueeze(fixed_x, dim=0))
    samples.extend([fixed_x, torch.squeeze(fake_y, 0)])
```

Resultados

```
fig = plt.figure(figsize=(18, 14))
grid = ImageGrid(fig, 111, nrows_ncols=(5, 4), axes_pad=0.5)


for i, (ax, im) in enumerate(zip(grid, samples)):
    _, w, h = im.size()
    im = im.detach().cpu().numpy()
    im = np.transpose(im, (1, 2, 0))

    im = ((im +1)*255 / (2)).astype(np.uint8)
    ax.imshow(im.reshape((w,h,3)))

    ax.xaxis.set_visible(False)
    ax.yaxis.set_visible(False)

    if i%2 == 0: title = "original"
    else: title = "anime"

    ax.set_title(title)

plt.show()
```