Laboratorio 6 Parte 1

En este laboratorio, estaremos repasando los conceptos de Generative Adversarial Networks En la segunda parte nos acercaremos a esta arquitectura a través de buscar generar numeros que parecieran ser generados a mano. Esta vez ya no usaremos versiones deprecadas de la librería de PyTorch, por ende, creen un nuevo virtual env con las librerías más recientes que puedan por favor.

Al igual que en laboratorios anteriores, para este laboratorio estaremos usando una herramienta para Jupyter Notebooks que facilitará la calificación, no solo asegurándo que ustedes tengan una nota pronto sino también mostrandoles su nota final al terminar el laboratorio.

De nuevo me discupo si algo no sale bien, seguiremos mejorando conforme vayamos iterando. Siempre pido su comprensión y colaboración si algo no funciona como debería.

Al igual que en el laboratorio pasado, estaremos usando la librería de Dr John Williamson et al de la University of Glasgow, además de ciertas piezas de código de Dr Bjorn Jensen de su curso de Introduction to Data Science and System de la University of Glasgow para la visualización de sus calificaciones.

NOTA: Ahora tambien hay una tercera dependecia que se necesita instalar. Ver la celda de abajo por favor

```
# Una vez instalada la librería por favor, recuerden volverla a
        comentar.
!pip install -U --force-reinstall --no-cache
        https://github.com/johnhw/jhwutils/zipball/master
!pip install scikit-image
!pip install -U --force-reinstall --no-cache
        https://github.com/AlbertS789/lautils/zipball/master
```

```
Collecting https://github.com/johnhw/jhwutils/zipball/master
  Downloading https://github.com/johnhw/jhwutils/zipball/master
\ 119.1 kB 891.5 kB/s 0:00:00
etadata (setup.py) ... e=jhwutils-1.3-py3-none-any.whl size=41854
sha256=5ff146e0eb8d770a1eb15bd154e86d5a5e112bcebf18c2cbdf69b3635f0c70f0
  Stored in directory: /tmp/pip-ephem-wheel-cache-
9yq0bzsf/wheels/ea/a5/c1/a5a791fc4679b6b51da9afbde7aaf0a44719030ae5715
Successfully built jhwutils
Installing collected packages: jhwutils
```

```
  Attempting uninstall: jhwutils
    Found existing installation: jhwutils 1.3
    Uninstalling jhwutils-1.3:
      Successfully uninstalled jhwutils-1.3
Successfully installed jhwutils-1.3
Requirement already satisfied: scikit-image in
/usr/local/lib/python3.12/dist-packages (0.25.2)
Requirement already satisfied: numpy>=1.24 in
/usr/local/lib/python3.12/dist-packages (from scikit-image) (2.0.2)
Requirement already satisfied: scipy>=1.11.4 in
/usr/local/lib/python3.12/dist-packages (from scikit-image) (1.16.1)
Requirement already satisfied: networkx>=3.0 in
/usr/local/lib/python3.12/dist-packages (from scikit-image) (3.5)
Requirement already satisfied: pillow>=10.1 in
/usr/local/lib/python3.12/dist-packages (from scikit-image) (11.3.0)
Requirement already satisfied: imageio!=2.35.0,>=2.33 in
/usr/local/lib/python3.12/dist-packages (from scikit-image) (2.37.0)
Requirement already satisfied: tifffile>=2022.8.12 in
/usr/local/lib/python3.12/dist-packages (from scikit-image)
(2025.6.11)
Requirement already satisfied: packaging>=21 in
/usr/local/lib/python3.12/dist-packages (from scikit-image) (25.0)
Requirement already satisfied: lazy-loader>=0.4 in
/usr/local/lib/python3.12/dist-packages (from scikit-image) (0.4)
Collecting https://github.com/AlbertS789/lautils/zipball/master
  Downloading https://github.com/AlbertS789/lautils/zipball/master
- 4.2 kB ? 0:00:00
etadata (setup.py) ... e=lautils-1.0-py3-none-any.whl size=2826
sha256=372f9532092146d4710830775d9394c8af1af000c370c5d8cf71dd18d8d7e078
  Stored in directory: /tmp/pip-ephem-wheel-cache-
s_ngait9/wheels/bf/c9/fa/c5e1b05da8bca40206e1a779b037d0fd93eea00e7fcfa5
Successfully built lautils
Installing collected packages: lautils
  Attempting uninstall: lautils
    Found existing installation: lautils 1.0
    Uninstalling lautils-1.0:
      Successfully uninstalled lautils-1.0
Successfully installed lautils-1.0
```

```
!pip install torch torchvision
```

```
Requirement already satisfied: torch in
/usr/local/lib/python3.12/dist-packages (2.8.0+cu126)
```

```
Requirement already satisfied: torchvision in
/usr/local/lib/python3.12/dist-packages (0.23.0+cu126)
Requirement already satisfied: filelock in
/usr/local/lib/python3.12/dist-packages (from torch) (3.19.1)
Requirement already satisfied: typing-extensions>=4.10.0 in
/usr/local/lib/python3.12/dist-packages (from torch) (4.14.1)
Requirement already satisfied: setuptools in
/usr/local/lib/python3.12/dist-packages (from torch) (75.2.0)
Requirement already satisfied: sympy>=1.13.3 in
/usr/local/lib/python3.12/dist-packages (from torch) (1.13.3)
Requirement already satisfied: networkx in
/usr/local/lib/python3.12/dist-packages (from torch) (3.5)
Requirement already satisfied: jinja2 in
/usr/local/lib/python3.12/dist-packages (from torch) (3.1.6)
Requirement already satisfied: fsspec in
/usr/local/lib/python3.12/dist-packages (from torch) (2025.3.0)
Requirement already satisfied: nvidia-cuda-nvrtc-cu12==12.6.77 in
/usr/local/lib/python3.12/dist-packages (from torch) (12.6.77)
Requirement already satisfied: nvidia-cuda-runtime-cu12==12.6.77 in
/usr/local/lib/python3.12/dist-packages (from torch) (12.6.77)
Requirement already satisfied: nvidia-cuda-cupti-cu12==12.6.80 in
/usr/local/lib/python3.12/dist-packages (from torch) (12.6.80)
Requirement already satisfied: nvidia-cudnn-cu12==9.10.2.21 in
/usr/local/lib/python3.12/dist-packages (from torch) (9.10.2.21)
Requirement already satisfied: nvidia-cublas-cu12==12.6.4.1 in
/usr/local/lib/python3.12/dist-packages (from torch) (12.6.4.1)
Requirement already satisfied: nvidia-cufft-cu12==11.3.0.4 in
/usr/local/lib/python3.12/dist-packages (from torch) (11.3.0.4)
Requirement already satisfied: nvidia-curand-cu12==10.3.7.77 in
/usr/local/lib/python3.12/dist-packages (from torch) (10.3.7.77)
Requirement already satisfied: nvidia-cusolver-cu12==11.7.1.2 in
/usr/local/lib/python3.12/dist-packages (from torch) (11.7.1.2)
Requirement already satisfied: nvidia-cusparse-cu12==12.5.4.2 in
/usr/local/lib/python3.12/dist-packages (from torch) (12.5.4.2)
Requirement already satisfied: nvidia-cusparselt-cu12==0.7.1 in
/usr/local/lib/python3.12/dist-packages (from torch) (0.7.1)
Requirement already satisfied: nvidia-nccl-cu12==2.27.3 in
/usr/local/lib/python3.12/dist-packages (from torch) (2.27.3)
Requirement already satisfied: nvidia-nvtx-cu12==12.6.77 in
/usr/local/lib/python3.12/dist-packages (from torch) (12.6.77)
Requirement already satisfied: nvidia-nvjitlink-cu12==12.6.85 in
/usr/local/lib/python3.12/dist-packages (from torch) (12.6.85)
```

```
Requirement already satisfied: nvidia-cufile-cu12==1.11.1.6 in
/usr/local/lib/python3.12/dist-packages (from torch) (1.11.1.6)
Requirement already satisfied: triton==3.4.0 in
/usr/local/lib/python3.12/dist-packages (from torch) (3.4.0)
Requirement already satisfied: numpy in
/usr/local/lib/python3.12/dist-packages (from torchvision) (2.0.2)
Requirement already satisfied: pillow!=8.3.*,>=5.3.0 in
/usr/local/lib/python3.12/dist-packages (from torchvision) (11.3.0)
Requirement already satisfied: mpmath<1.4,>=1.1.0 in
/usr/local/lib/python3.12/dist-packages (from sympy>=1.13.3->torch)
(1.3.0)
Requirement already satisfied: MarkupSafe>=2.0 in
/usr/local/lib/python3.12/dist-packages (from jinja2->torch) (3.0.2)
```

```
import numpy as np
import copy
import matplotlib.pyplot as plt
import scipy
from PIL import Image
import os
from collections import defaultdict

#from IPython import display
#from base64 import b64decode



# Other imports
from unittest.mock import patch
from uuid import getnode as get_mac

from jhwutils.checkarr import array_hash, check_hash, check_scalar,
        check_string, array_hash, _check_scalar
import jhwutils.image_audio as ia
import jhwutils.tick as tick
from lautils.gradeutils import new_representation, hex_to_float,
        compare_numbers, compare_lists_by_percentage,
        calculate_coincidences_percentage


###
tick.reset_marks()

%matplotlib inline

# Celda escondida para utlidades necesarias, por favor NO edite esta
        celda
```

Información del estudiante en dos variables

- carne_1 : un string con su carne (e.g. "12281"), debe ser de al menos 5 caracteres.
- firma_mecanografiada_1: un string con su nombre (e.g. "Albero Suriano") que se usará para la declaracion que este trabajo es propio (es decir, no hay plagio)
- carne_2 : un string con su carne (e.g. "12281"), debe ser de al menos 5 caracteres.
- firma_mecanografiada_2: un string con su nombre (e.g. "Albero Suriano") que se usará para la declaracion que este trabajo es propio (es decir, no hay plagio)

```
carne_1 = '22243'
firma_mecanografiada_1 = 'Francis Aguilar'
carne_2 = '22535'
firma_mecanografiada_2 = 'Cesar Lopez'
# YOUR CODE HERE
# raise NotImplementedError()

# Deberia poder ver dos checkmarks verdes [0 marks], que indican que
        su información básica está OK


with tick.marks(0):
    assert(len(carne_1)>=5 and len(carne_2)>=5)


with tick.marks(0):
    assert(len(firma_mecanografiada_1)>0 and
        len(firma_mecanografiada_2)>0)
```

✓ [0 marks]

✓ [0 marks]

Introducción

Créditos: Esta parte de este laboratorio está tomado y basado en uno de los blogs de Renato Candido, así como las imagenes presentadas en este laboratorio a menos que se indique lo contrario.

Las redes generativas adversarias también pueden generar muestras de alta dimensionalidad, como imágenes. En este ejemplo, se va a utilizar una GAN para generar imágenes de dígitos escritos a mano. Para ello, se entrenarán los modelos utilizando el conjunto de datos MNIST de dígitos escritos a mano, que está incluido en el paquete torchvision.

Dado que este ejemplo utiliza imágenes en el conjunto de datos de entrenamiento, los modelos necesitan ser más complejos, con un mayor número de parámetros. Esto hace que el proceso de entrenamiento sea más lento, llevando alrededor de dos minutos por época (aproximadamente) al ejecutarse en la CPU. Se necesitarán alrededor de cincuenta épocas para obtener un resultado relevante, por lo que el tiempo total de entrenamiento al usar una CPU es de alrededor de cien minutos.

Para reducir el tiempo de entrenamiento, se puede utilizar una GPU si está disponible. Sin embargo, será necesario mover manualmente tensores y modelos a la GPU para usarlos en el proceso de entrenamiento.

Se puede asegurar que el código se ejecutará en cualquier configuración creando un objeto de dispositivo que apunte a la CPU o, si está disponible, a la GPU. Más adelante, se utilizará este dispositivo para definir dónde deben crearse los tensores y los modelos, utilizando la GPU si está disponible.

```
import torch
from torch import nn

import math
import matplotlib.pyplot as plt
import torchvision
import torchvision.transforms as transforms

import random
import numpy as np

seed_ = 111

def seed_all(seed_):
    random.seed(seed_)
    np.random.seed(seed_)
    torch.manual_seed(seed_)
    torch.cuda.manual_seed(seed_)
    torch.backends.cudnn.deterministic = True

seed_all(seed_)
```

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

```
cuda
```

Preparando la Data

El conjunto de datos MNIST consta de imágenes en escala de grises de 28 × 28 píxeles de dígitos escritos a mano del 0 al 9. Para usarlos con PyTorch, será necesario realizar algunas conversiones. Para ello, se define transform, una función que se utilizará al cargar los datos:

La función tiene dos partes:

- transforms.ToTensor() convierte los datos en un tensor de PyTorch.
- transforms.Normalize() convierte el rango de los coeficientes del tensor.

Los coeficientes originales proporcionados por transforms.ToTensor() varían de 0 a 1, y dado que los fondos de las imágenes son negros, la mayoría de los coeficientes son iguales a 0 cuando se representan utilizando este rango.

transforms.Normalize() cambia el rango de los coeficientes a -1 a 1 restando 0.5 de los coeficientes originales y dividiendo el resultado por 0.5. Con esta transformación, el número de elementos iguales a 0 en las muestras de entrada se reduce drásticamente, lo que ayuda en el entrenamiento de los modelos.

Los argumentos de transforms.Normalize() son dos tuplas, $(M_1, ..., M_n)$ y $(S_1, ..., S_n)$, donde n representa el número de canales de las imágenes. Las imágenes en escala de grises como las del conjunto de datos MNIST tienen solo un canal, por lo que las tuplas tienen solo un valor. Luego, para cada canal i de la imagen, transforms.Normalize() resta M_i de los coeficientes y divide el resultado por S_i.

Luego se pueden cargar los datos de entrenamiento utilizando torchvision.datasets.MNIST y realizar las conversiones utilizando transform

El argumento download=True garantiza que la primera vez que se ejecute el código, el conjunto de datos MNIST se descargará y almacenará en el directorio actual, como se indica en el argumento root.

Después que se ha creado train_set, se puede crear el cargador de datos como se hizo antes en la parte 1.

Cabe decir que se puede utilizar Matplotlib para trazar algunas muestras de los datos de entrenamiento. Para mejorar la visualización, se puede usar cmap=gray_r para invertir el mapa de colores y representar los dígitos en negro sobre un fondo blanco:

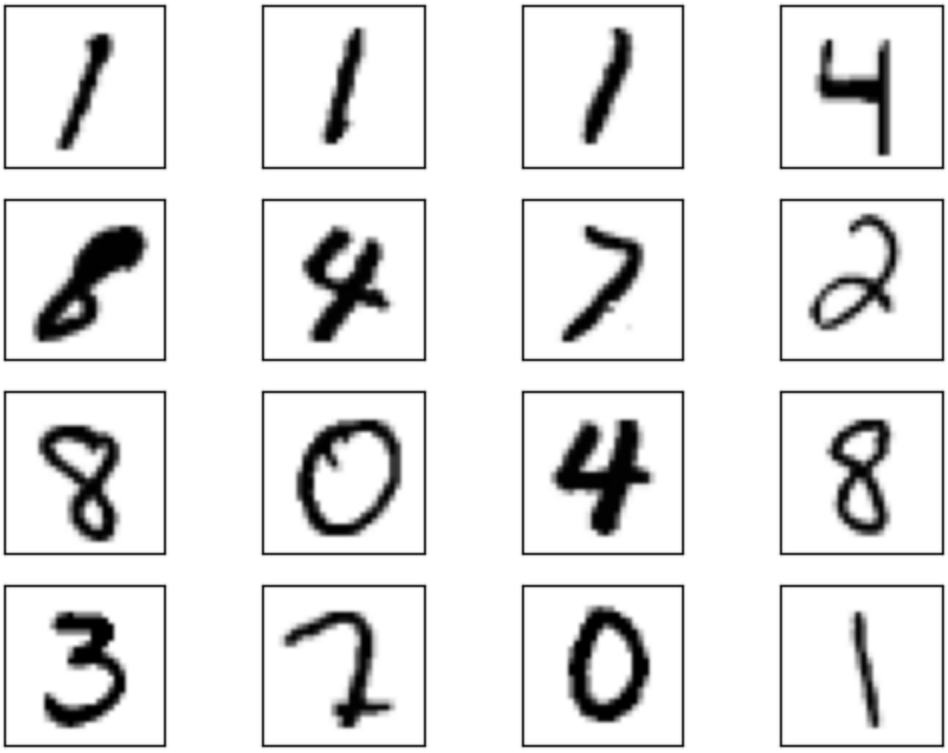
Como se puede ver más adelante, hay dígitos con diferentes estilos de escritura. A medida que la GAN aprende la distribución de los datos, también generará dígitos con diferentes estilos de escritura.

```
transform = transforms.Compose(
    [transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))]
)

train_set = torchvision.datasets.MNIST(
    root=".", train=True, download=True, transform=transform
)

batch_size = 32
train_loader = torch.utils.data.DataLoader(
    train_set, batch_size=batch_size, shuffle=True
)

real_samples, mnist_labels = next(iter(train_loader))
for i in range(16):
    ax = plt.subplot(4, 4, i + 1)
    plt.imshow(real_samples[i].reshape(28, 28), cmap="gray_r")
    plt.xticks([])
    plt.yticks([])
```



Implementando el Discriminador y el Generador

En este caso, el discriminador es una red neuronal MLP (multi-layer perceptron) que recibe una imagen de 28 × 28 píxeles y proporciona la probabilidad de que la imagen pertenezca a los datos reales de entrenamiento.

Para introducir los coeficientes de la imagen en la red neuronal MLP, se vectorizan para que la red neuronal reciba vectores con 784 coeficientes.

La vectorización ocurre cuando se ejecuta .forward(), ya que la llamada a x.view() convierte la forma del tensor de entrada. En este caso, la forma original de la entrada "x" es 32 × 1 × 28 × 28, donde 32 es el tamaño del batch que se ha configurado. Después de la conversión, la forma de "x" se convierte en 32 × 784, con cada línea representando los coeficientes de una imagen del conjunto de entrenamiento.

Para ejecutar el modelo de discriminador usando la GPU, hay que instanciarlo y enviarlo a la GPU con .to(). Para usar una GPU cuando haya una disponible, se puede enviar el modelo al objeto de dispositivo creado anteriormente.

Dado que el generador va a generar datos más complejos, es necesario aumentar las dimensiones de la entrada desde el espacio latente. En este caso, el generador va a recibir una entrada de 100 dimensiones y proporcionará una salida con 784 coeficientes, que se organizarán en un tensor de 28 × 28 que representa una imagen.

Luego, se utiliza la función tangente hiperbólica Tanh() como activación de la capa de salida, ya que los coeficientes de salida deben estar en el intervalo de -1 a 1 (por la normalización que se hizo anteriormente). Después, se instancia el generador y se envía a device para usar la GPU si está disponible.

```
class Discriminator(nn.Module):
    def __init__(self):
        super().__init__()
        self.model = nn.Sequential(
            # Aprox 11 lineas
            # lineal de la entrada dicha y salida 1024
            # ReLU
            # Dropout de 30%
            # Lineal de la entrada correspondiente y salida 512
            # ReLU
            # Dropout de 30%
            # Lienal de la entrada correspondiente y salida 256
            # ReLU
            # Dropout de 30%
            # Lineal de la entrada correspondiente y salida 1
            # Sigmoide
            # YOUR CODE HERE
            nn.Linear(784, 1024),
            nn.ReLU(),
            nn.Dropout(0.3),

            nn.Linear(1024, 512),
            nn.ReLU(),
            nn.Dropout(0.3),

            nn.Linear(512, 256),
            nn.ReLU(),
            nn.Dropout(0.3),

            nn.Linear(256, 1),
            nn.Sigmoid()
        )

    def forward(self, x):
        x = x.view(x.size(0), 784)
        output = self.model(x)
        return output
```

```
class Generator(nn.Module):
    def __init__(self):
        super().__init__()
        self.model = nn.Sequential(
            # Aprox 8 lienas para
            # Lineal input = 100, output = 256
            # ReLU
            # Lineal output = 512
            # ReLU
            # Lineal output = 1024
            # ReLU
            # Lineal output = 784
            # Tanh
            # YOUR CODE HERE
            nn.Linear(100, 256),
            nn.ReLU(),

            nn.Linear(256, 512),
            nn.ReLU(),

            nn.Linear(512, 1024),
            nn.ReLU(),

            nn.Linear(1024, 784),
            nn.Tanh()
        )

    def forward(self, x):
        output = self.model(x)
        output = output.view(x.size(0), 1, 28, 28)
        return output
```

Entrenando los Modelos

Para entrenar los modelos, es necesario definir los parámetros de entrenamiento y los optimizadores como se hizo en la parte anterior.

Para obtener un mejor resultado, se disminuye la tasa de aprendizaje de la primera parte. También se establece el número de épocas en 10 para reducir el tiempo de entrenamiento.

El ciclo de entrenamiento es muy similar al que se usó en la parte previa. Note como se envían los datos de entrenamiento a device para usar la GPU si está disponible

Algunos de los tensores no necesitan ser enviados explícitamente a la GPU con device. Este es el caso de generated_samples, que ya se envió a una GPU disponible, ya que latent_space_samples y generator se enviaron a la GPU previamente.

Dado que esta parte presenta modelos más complejos, el entrenamiento puede llevar un poco más de tiempo. Después de que termine, se pueden verificar los resultados generando algunas muestras de dígitos escritos a mano.

```
list_images = []

# Aprox 1 linea para que decidan donde guardar un set de imagen que
        vamos a generar de las graficas
# path_imgs =
# YOUR CODE HERE
path_imgs = "/MNIST"

#seed_all(seed_)

discriminator = Discriminator().to(device=device)
generator = Generator().to(device=device)

lr = 0.0001
num_epochs = 50
loss_function = nn.BCELoss()

optimizer_discriminator = torch.optim.Adam(discriminator.parameters(),
        lr=lr)
optimizer_generator = torch.optim.Adam(generator.parameters(), lr=lr)

for epoch in range(num_epochs):
    for n, (real_samples, mnist_labels) in enumerate(train_loader):
        # Data for training the discriminator
        real_samples = real_samples.to(device=device)
        real_samples_labels = torch.ones((batch_size, 1)).to(
            device=device
        )
        latent_space_samples = torch.randn((batch_size, 100)).to(
            device=device
        )
```

```
generated_samples = generator(latent_space_samples)
generated_samples_labels = torch.zeros((batch_size, 1)).to(
    device=device
)
all_samples = torch.cat((real_samples, generated_samples))
all_samples_labels = torch.cat(
    (real_samples_labels, generated_samples_labels)
)

# Training the discriminator
# Aprox 2 lineas para
# setear el discriminador en zero_grad
# output_discriminator =
# YOUR CODE HERE
optimizer_discriminator.zero_grad()
output_discriminator = discriminator(all_samples)
loss_discriminator = loss_function(
    output_discriminator, all_samples_labels
)
# Aprox dos lineas para
# llamar al paso backward sobre el loss_discriminator
# llamar al optimizador sobre optimizer_discriminator
# YOUR CODE HERE
loss_discriminator.backward()
optimizer_discriminator.step()

# Data for training the generator
latent_space_samples = torch.randn((batch_size, 100)).to(
    device=device
)

# Training the generator
# Training the generator
# Aprox 2 lineas para
# setear el generador en zero_grad
# output_discriminator =
# YOUR CODE HERE
optimizer_generator.zero_grad()
generated_samples = generator(latent_space_samples)
output_discriminator_generated =
discriminator(generated_samples)
loss_generator = loss_function(
    output_discriminator_generated, real_samples_labels
```

```
        )

        # Aprox dos lineas para
        # llamar al paso backward sobre el loss_generator
        # llamar al optimizador sobre optimizer_generator
        # YOUR CODE HERE
        loss_generator.backward()
        optimizer_generator.step()

        # Guardamos las imagenes
        if epoch % 2 == 0 and n ==  batch_size - 1:
            generated_samples_detached =
generated_samples.cpu().detach()
            for i in range(16):
                ax = plt.subplot(4, 4, i + 1)
                plt.imshow(generated_samples_detached[i].reshape(28,
28), cmap="gray_r")
                plt.xticks([])
                plt.yticks([])
                plt.title("Epoch "+str(epoch))
            name = path_imgs + "epoch_mnist"+str(epoch)+".jpg"
            plt.savefig(name, format="jpg")
            plt.close()
            list_images.append(name)

        # Show loss
        if n == batch_size - 1:
            print(f"Epoch: {epoch} Loss D.: {loss_discriminator}")
            print(f"Epoch: {epoch} Loss G.: {loss_generator}")
```

```
Epoch: 0 Loss D.: 0.533566951751709
Epoch: 0 Loss G.: 0.5314837694168091
Epoch: 1 Loss D.: 0.019780190661549568
Epoch: 1 Loss G.: 4.661293029785156
Epoch: 2 Loss D.: 0.006425430998206139
Epoch: 2 Loss G.: 5.366535663604736
Epoch: 3 Loss D.: 0.06317351013422012
Epoch: 3 Loss G.: 4.940392971038818
Epoch: 4 Loss D.: 0.06464124470949173
Epoch: 4 Loss G.: 4.966769218444824
Epoch: 5 Loss D.: 0.13390204310417175
Epoch: 5 Loss G.: 4.263368606567383
Epoch: 6 Loss D.: 0.16198483109474182
Epoch: 6 Loss G.: 3.174546241760254
```

```
Epoch: 7 Loss D.: 0.16018427908420563
Epoch: 7 Loss G.: 3.449517250061035
Epoch: 8 Loss D.: 0.3173588514328003
Epoch: 8 Loss G.: 2.5413260459899902
Epoch: 9 Loss D.: 0.3811987638473511
Epoch: 9 Loss G.: 2.123145341873169
Epoch: 10 Loss D.: 0.3346656560897827
Epoch: 10 Loss G.: 1.9928719997406006
Epoch: 11 Loss D.: 0.4741729497909546
Epoch: 11 Loss G.: 1.6833062171936035
Epoch: 12 Loss D.: 0.30002495646476746
Epoch: 12 Loss G.: 1.5235552787780762
Epoch: 13 Loss D.: 0.5187399983406067
Epoch: 13 Loss G.: 1.5060712099075317
Epoch: 14 Loss D.: 0.3872467279434204
Epoch: 14 Loss G.: 1.3475391864776611
Epoch: 15 Loss D.: 0.3825750946998596
Epoch: 15 Loss G.: 1.2087275981903076
Epoch: 16 Loss D.: 0.3753535747528076
Epoch: 16 Loss G.: 1.613954782485962
Epoch: 17 Loss D.: 0.508043646812439
Epoch: 17 Loss G.: 1.2816338539123535
Epoch: 18 Loss D.: 0.527686595916748
Epoch: 18 Loss G.: 1.0642900466918945
Epoch: 19 Loss D.: 0.409196138381958
Epoch: 19 Loss G.: 1.2939021587371826
Epoch: 20 Loss D.: 0.5026688575744629
Epoch: 20 Loss G.: 1.3210129737854004
Epoch: 21 Loss D.: 0.51596599817276
Epoch: 21 Loss G.: 1.2165497541427612
Epoch: 22 Loss D.: 0.5356346964836121
Epoch: 22 Loss G.: 1.0918538570404053
Epoch: 23 Loss D.: 0.5763896703720093
Epoch: 23 Loss G.: 0.9177920818328857
Epoch: 24 Loss D.: 0.5305367708206177
Epoch: 24 Loss G.: 1.1039694547653198
Epoch: 25 Loss D.: 0.5765623450279236
Epoch: 25 Loss G.: 0.9773527383804321
Epoch: 26 Loss D.: 0.4999510943889618
Epoch: 26 Loss G.: 1.3337275981903076
Epoch: 27 Loss D.: 0.6191483736038208
Epoch: 27 Loss G.: 1.0444190502166748
```

```
Epoch: 28 Loss D.: 0.5308555960655212
Epoch: 28 Loss G.: 1.151049256324768
Epoch: 29 Loss D.: 0.5877175331115723
Epoch: 29 Loss G.: 1.0916951894760132
Epoch: 30 Loss D.: 0.5847316980361938
Epoch: 30 Loss G.: 1.2050634622573853
Epoch: 31 Loss D.: 0.5613664388656616
Epoch: 31 Loss G.: 1.029157042503357
Epoch: 32 Loss D.: 0.5265491008758545
Epoch: 32 Loss G.: 1.0787984132766724
Epoch: 33 Loss D.: 0.5835425853729248
Epoch: 33 Loss G.: 1.0844082832336426
Epoch: 34 Loss D.: 0.5341429710388184
Epoch: 34 Loss G.: 1.0565402507781982
Epoch: 35 Loss D.: 0.5583127737045288
Epoch: 35 Loss G.: 1.1494441032409668
Epoch: 36 Loss D.: 0.542460560798645
Epoch: 36 Loss G.: 0.8773486614227295
Epoch: 37 Loss D.: 0.584244430065155
Epoch: 37 Loss G.: 0.9438858032226562
Epoch: 38 Loss D.: 0.5876556038856506
Epoch: 38 Loss G.: 1.0425779819488525
Epoch: 39 Loss D.: 0.6092830300331116
Epoch: 39 Loss G.: 1.0023518800735474
Epoch: 40 Loss D.: 0.6027634143829346
Epoch: 40 Loss G.: 0.9922776222229004
Epoch: 41 Loss D.: 0.5704353451728821
Epoch: 41 Loss G.: 0.8712483644485474
Epoch: 42 Loss D.: 0.6792483329772949
Epoch: 42 Loss G.: 0.9543055295944214
Epoch: 43 Loss D.: 0.5483911037445068
Epoch: 43 Loss G.: 1.036293387413025
Epoch: 44 Loss D.: 0.618618905544281
Epoch: 44 Loss G.: 1.1315854787826538
Epoch: 45 Loss D.: 0.590842604637146
Epoch: 45 Loss G.: 0.9995961785316467
Epoch: 46 Loss D.: 0.6064241528511047
Epoch: 46 Loss G.: 0.8814513683319092
Epoch: 47 Loss D.: 0.6064466238021851
Epoch: 47 Loss G.: 1.0360443592071533
Epoch: 48 Loss D.: 0.5010616183280945
Epoch: 48 Loss G.: 1.0362693071365356
```

```
Epoch: 49 Loss D.: 0.6005738973617554
Epoch: 49 Loss G.: 0.9671766757965088
```

```
with tick.marks(35):
    assert compare_numbers(new_representation(loss_discriminator),
        "3c3d", '0x1.3333333333333p-1')
```

```
with tick.marks(35):
    assert compare_numbers(new_representation(loss_generator), "3c3d",
        '0x1.8000000000000p+0')
```

✓ [35 marks]

✓ [35 marks]

Validación del Resultado

Para generar dígitos escritos a mano, es necesario tomar algunas muestras aleatorias del espacio latente y alimentarlas al generador.

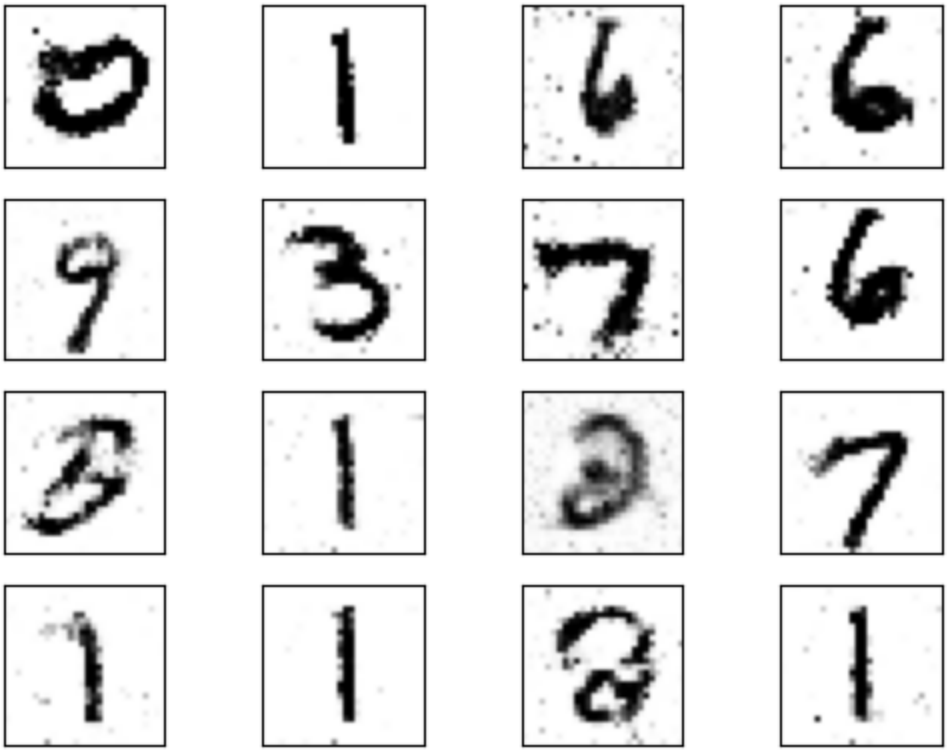
Para trazar generated_samples, es necesario mover los datos de vuelta a la CPU en caso de que estén en la GPU. Para ello, simplemente se puede llamar a .cpu(). Como se hizo anteriormente, también es necesario llamar a .detach() antes de usar Matplotlib para trazar los datos.

La salida debería ser dígitos que se asemejen a los datos de entrenamiento. Después de cincuenta épocas de entrenamiento, hay varios dígitos generados que se asemejan a los reales. Se pueden mejorar los resultados considerando más épocas de entrenamiento. Al igual que en la parte anterior, al utilizar un tensor de muestras de espacio latente fijo y alimentarlo al generador al final de cada época durante el proceso de entrenamiento, se puede visualizar la evolución del entrenamiento.

Se puede observar que al comienzo del proceso de entrenamiento, las imágenes generadas son completamente aleatorias. A medida que avanza el entrenamiento, el generador aprende la distribución de los datos reales y, a algunas épocas, algunos dígitos generados ya se asemejan a los datos reales.

```
latent_space_samples = torch.randn(batch_size, 100).to(device=device)
generated_samples = generator(latent_space_samples)
```

```
generated_samples = generated_samples.cpu().detach()
for i in range(16):
    ax = plt.subplot(4, 4, i + 1)
    plt.imshow(generated_samples[i].reshape(28, 28), cmap="gray_r")
    plt.xticks([])
    plt.yticks([])
```

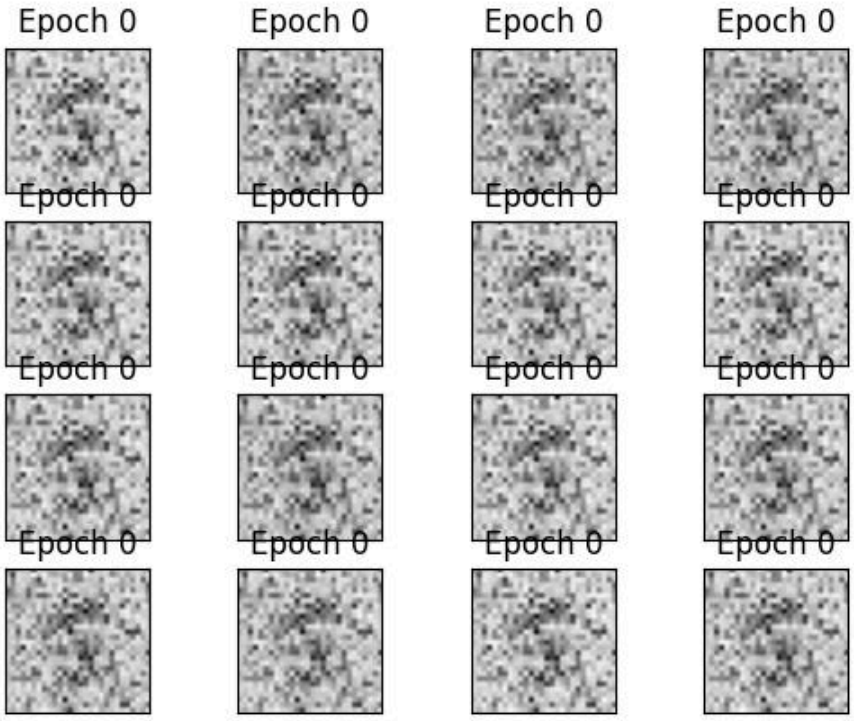


```
# Visualización del progreso de entrenamiento
# Para que esto se ve bien, por favor reinicien el kernel y corran
        todo el notebook


from PIL import Image
from IPython.display import display, Image as IPImage


images = [Image.open(path) for path in list_images]


# Save the images as an animated GIF
gif_path = "animation.gif"  # Specify the path for the GIF file
images[0].save(gif_path, save_all=True, append_images=images[1:],
        loop=0, duration=1000)
display(IPImage(filename=gif_path))
```



Las respuestas de estas preguntas representan el 30% de este notebook

PREGUNTAS:

- ¿Qué diferencias hay entre los modelos usados en la primera parte y los usados en esta parte?

En la primera parte los modelos se enfocaban en generar puntos dentro de un espacio bidimensional R^2, mientras que en esta segunda fase el objetivo es producir imágenes de 28x28 píxeles. Además, a nivel de implementación, en la parte dos se integran funciones de activación como tanh, que permiten normalizar los datos del conjunto MNIST de manera más adecuada.

- ¿Qué tan bien se han creado las imagenes esperadas?

Las imágenes muestran una buena calidad a partir de la época 28, ya que en ese punto comienzan a percibirse formas claras de los dígitos. Aun así, algunas cifras permanecen algo difusas o incompletas, pero en general ya se distinguen los números de forma aceptable.

- ¿Cómo mejoraría los modelos?

Probaría otros optimizadores como Adam o incluso una tasa de aprendizaje variable para afinar los resultados. También sería útil incrementar la cantidad de datos de entrenamiento.

- Observe el GIF creado, y describa la evolución que va viendo al pasar de las epocas

En las primeras épocas lo único visible es ruido sin forma definida. Alrededor de la época 14 empiezan a aparecer contornos y líneas que sugieren números. Para la época 30 los dígitos se distinguen con bastante claridad, aunque aún con un poco de distorsión. En las últimas épocas los números resultan bastante legibles, aunque se observa cierta repetición en la generación, destacando la aparición frecuente del número 1.

```
print()
print("La fraccion de abajo muestra su rendimiento basado en las
        partes visibles de este laboratorio")
tick.summarise_marks()
```

```
La fraccion de abajo muestra su rendimiento basado en las partes
visibles de este laboratorio
```

140 / 140 marks (100.0%)