Laboratorio 6 Parte 2

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
     - 0 bytes ? 0:00:00
     - 12.8 kB ? 0:00:00
     - 38.1 kB 800.1 kB/s 0:00:00
  Preparing metadata (setup.py): started
  Preparing metadata (setup.py): finished with status 'done'
Building wheels for collected packages: jhwutils
```

```
  Building wheel for jhwutils (setup.py): started
  Building wheel for jhwutils (setup.py): finished with status 'done'
  Created wheel for jhwutils: filename=jhwutils-1.0-py3-none-any.whl
size=33803
sha256=196dbdf5344908ecc4b758a965ae086aa8ce030c8307e63e0280b122b18a295
  Stored in directory: C:\Users\charl\AppData\Local\Temp\pip-ephem-
wheel-cache-
x7lq0c77\wheels\27\3c\cb\eb7b3c6ea36b5b54e5746751443be9bb0d733529190335
Successfully built jhwutils
Installing collected packages: jhwutils
  Attempting uninstall: jhwutils
    Found existing installation: jhwutils 1.0
    Uninstalling jhwutils-1.0:
      Successfully uninstalled jhwutils-1.0
Successfully installed jhwutils-1.0
Requirement already satisfied: scikit-image in
c:\users\charl\appdata\local\packages\pythonsoftwarefoundation.python.3
packages\python310\site-packages (0.21.0)
Requirement already satisfied: numpy>=1.21.1 in
c:\users\charl\appdata\local\packages\pythonsoftwarefoundation.python.3
packages\python310\site-packages (from scikit-image) (1.24.3)
Requirement already satisfied: scipy>=1.8 in
c:\users\charl\appdata\local\packages\pythonsoftwarefoundation.python.3
packages\python310\site-packages (from scikit-image) (1.11.2)
Requirement already satisfied: networkx>=2.8 in
c:\users\charl\appdata\local\packages\pythonsoftwarefoundation.python.3
packages\python310\site-packages (from scikit-image) (3.1)
Requirement already satisfied: pillow>=9.0.1 in
c:\users\charl\appdata\local\packages\pythonsoftwarefoundation.python.3
packages\python310\site-packages (from scikit-image) (10.0.0)
Requirement already satisfied: imageio>=2.27 in
c:\users\charl\appdata\local\packages\pythonsoftwarefoundation.python.3
packages\python310\site-packages (from scikit-image) (2.31.1)
Requirement already satisfied: tifffile>=2022.8.12 in
c:\users\charl\appdata\local\packages\pythonsoftwarefoundation.python.3
packages\python310\site-packages (from scikit-image) (2023.8.12)
Requirement already satisfied: PyWavelets>=1.1.1 in
c:\users\charl\appdata\local\packages\pythonsoftwarefoundation.python.3
packages\python310\site-packages (from scikit-image) (1.4.1)
Requirement already satisfied: packaging>=21 in
```

```
c:\users\charl\appdata\local\packages\pythonsoftwarefoundation.python.3
packages\python310\site-packages (from scikit-image) (23.1)
Requirement already satisfied: lazy_loader>=0.2 in
c:\users\charl\appdata\local\packages\pythonsoftwarefoundation.python.3
packages\python310\site-packages (from scikit-image) (0.3)
Collecting https://github.com/AlbertS789/lautils/zipball/master
  Downloading https://github.com/AlbertS789/lautils/zipball/master
     - 0 bytes ? 0:00:00
     - 4.2 kB ? 0:00:00
  Preparing metadata (setup.py): started
  Preparing metadata (setup.py): finished with status 'done'
Building wheels for collected packages: lautils
  Building wheel for lautils (setup.py): started
  Building wheel for lautils (setup.py): finished with status 'done'
  Created wheel for lautils: filename=lautils-1.0-py3-none-any.whl
size=2833
sha256=a7d36b6142f18fadaf67599750f6b779975b4d5b0d2c694ffb98360fb379cb23
  Stored in directory: C:\Users\charl\AppData\Local\Temp\pip-ephem-
wheel-cache-
1ptqw5sb\wheels\16\3a\a0\5fbae86e17ef6bb8ed057aa04b591584005d1212c72d69
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
```

```
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 = "20347"
firma_mecanografiada_1 = "Alejandro Gómez"
carne_2 =  "20498"
firma_mecanografiada_2 = "Gabriel Vicente"
# 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)
```

```
cpu
```

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.

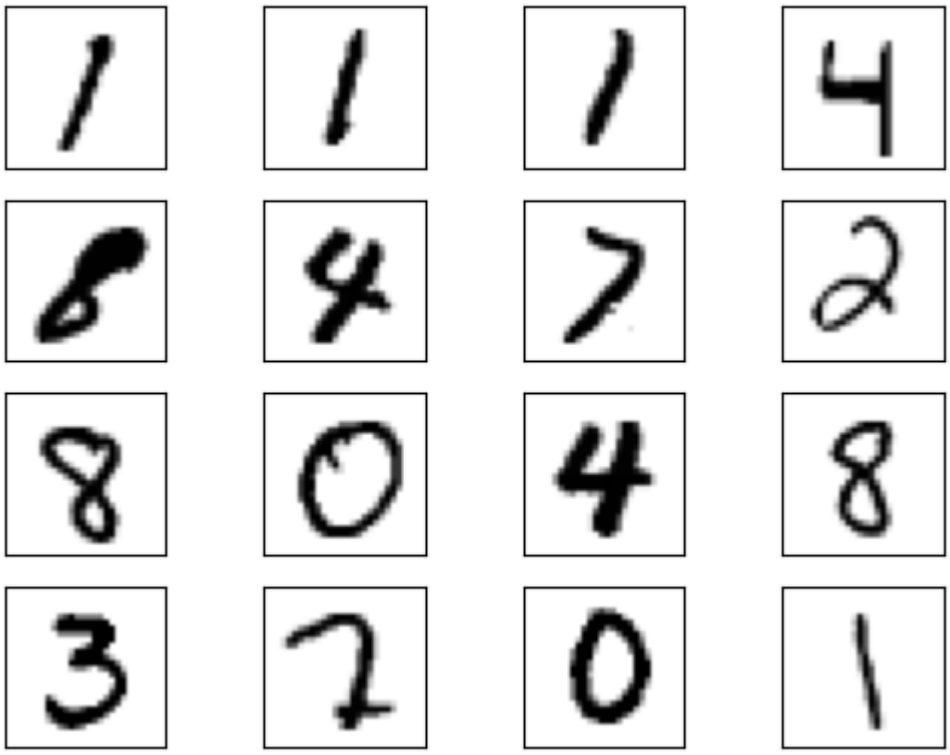
```
transform = transforms.Compose(
    [transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))]
)

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

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-
ubyte.gz
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-
ubyte.gz to .\MNIST\raw\train-images-idx3-ubyte.gz

100.0%

Extracting .\MNIST\raw\train-images-idx3-ubyte.gz to .\MNIST\raw

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-
ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-
ubyte.gz to .\MNIST\raw\train-labels-idx1-ubyte.gz

100.0%

Extracting .\MNIST\raw\train-labels-idx1-ubyte.gz to .\MNIST\raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz

6.0%

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
to .\MNIST\raw\t10k-images-idx3-ubyte.gz

100.0%

Extracting .\MNIST\raw\t10k-images-idx3-ubyte.gz to .\MNIST\raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
to .\MNIST\raw\t10k-labels-idx1-ubyte.gz

100.0%

Extracting .\MNIST\raw\t10k-labels-idx1-ubyte.gz to .\MNIST\raw
```

```
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
            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()
            # 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
                # raise NotImplementedError()
            )

        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(
                nn.Linear(100, 256),
                nn.ReLU(),
                nn.Linear(256, 512),
                nn.ReLU(),
                nn.Linear(512, 1024),
                nn.ReLU(),
                nn.Linear(1024, 784),
                nn.Tanh()
                # 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
                # raise NotImplementedError()
            )

        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 = r'C:\Users\charl\Desktop\S22023\DeepLearning\Laboratorio
        6\MNIST\raw'
# YOUR CODE HERE
# raise NotImplementedError()


#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
        discriminator.zero_grad()
        output_discriminator = discriminator(all_samples)
        # output_discriminator =
        # YOUR CODE HERE
        # raise NotImplementedError()
        loss_discriminator = loss_function(
            output_discriminator, all_samples_labels
        )
        # Aprox dos lineas para
        loss_discriminator.backward()
        optimizer_discriminator.step()
        # llamar al paso backward sobre el loss_discriminator
        # llamar al optimizador sobre optimizer_discriminator
        # YOUR CODE HERE
        # raise NotImplementedError()

        # 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 =
generator.zero_grad()
generated_samples = generator(latent_space_samples)
output_discriminator_generated =
discriminator(generated_samples)
# YOUR CODE HERE
# raise NotImplementedError()
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



loss_generator.backward()
optimizer_generator.step()


# YOUR CODE HERE
# raise NotImplementedError()


# 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.: 0.582188069820404
Epoch: 0 Loss G.: 0.4767847955226898
Epoch: 1 Loss D.: 0.020387854427099228
Epoch: 1 Loss G.: 4.5718159675598145
Epoch: 2 Loss D.: 0.09870754927396774
Epoch: 2 Loss G.: 6.217487335205078
Epoch: 3 Loss D.: 0.003388999029994011
Epoch: 3 Loss G.: 6.0547637939453125
Epoch: 4 Loss D.: 0.054444871842861176
Epoch: 4 Loss G.: 4.556949138641357
Epoch: 5 Loss D.: 0.010440755635499954
Epoch: 5 Loss G.: 5.5177178382873535
Epoch: 6 Loss D.: 0.11780352890491486
Epoch: 6 Loss G.: 3.5024919509887695
Epoch: 7 Loss D.: 0.08827097713947296
Epoch: 7 Loss G.: 3.1080594062805176
Epoch: 8 Loss D.: 0.13687069714069366
Epoch: 8 Loss G.: 2.828256845474243
Epoch: 9 Loss D.: 0.24902907013893127
Epoch: 9 Loss G.: 1.7165559530258179
Epoch: 10 Loss D.: 0.3276643753051758
Epoch: 10 Loss G.: 2.143460750579834
Epoch: 11 Loss D.: 0.34233343601226807
Epoch: 11 Loss G.: 2.0585784912109375
Epoch: 12 Loss D.: 0.32676413655281067
Epoch: 12 Loss G.: 1.541398286819458
Epoch: 13 Loss D.: 0.3864485025405884
Epoch: 13 Loss G.: 1.409684181213379
Epoch: 14 Loss D.: 0.48910266160964966
Epoch: 14 Loss G.: 1.7369248867034912
Epoch: 15 Loss D.: 0.39884841442108154
```

```
Epoch: 15 Loss G.: 1.5961031913757324
Epoch: 16 Loss D.: 0.4512018859386444
Epoch: 16 Loss G.: 1.5333669185638428
Epoch: 17 Loss D.: 0.4437119960784912
Epoch: 17 Loss G.: 1.3884249925613403
Epoch: 18 Loss D.: 0.4805162847042084
Epoch: 18 Loss G.: 1.2687146663665771
Epoch: 19 Loss D.: 0.5600918531417847
Epoch: 19 Loss G.: 1.388888955116272
Epoch: 20 Loss D.: 0.4301069378852844
Epoch: 20 Loss G.: 1.2361881732940674
Epoch: 21 Loss D.: 0.4010413885116577
Epoch: 21 Loss G.: 1.083971619606018
Epoch: 22 Loss D.: 0.5017399787902832
Epoch: 22 Loss G.: 1.3337831497192383
Epoch: 23 Loss D.: 0.533522367477417
Epoch: 23 Loss G.: 1.2215850353240967
Epoch: 24 Loss D.: 0.5897166728973389
Epoch: 24 Loss G.: 1.1219390630722046
Epoch: 25 Loss D.: 0.5004821419715881
Epoch: 25 Loss G.: 1.2980324029922485
Epoch: 26 Loss D.: 0.46156948804855347
Epoch: 26 Loss G.: 1.4588916301727295
Epoch: 27 Loss D.: 0.5458274483680725
Epoch: 27 Loss G.: 0.9553470611572266
Epoch: 28 Loss D.: 0.5294545888900757
Epoch: 28 Loss G.: 1.2799947261810303
Epoch: 29 Loss D.: 0.5302296876907349
Epoch: 29 Loss G.: 1.0439183712005615
Epoch: 30 Loss D.: 0.46474507451057434
Epoch: 30 Loss G.: 1.175995111465454
Epoch: 31 Loss D.: 0.5740188360214233
Epoch: 31 Loss G.: 1.1883225440979004
Epoch: 32 Loss D.: 0.5106662511825562
Epoch: 32 Loss G.: 0.9743540287017822
Epoch: 33 Loss D.: 0.50377357006073
Epoch: 33 Loss G.: 1.0549322366714478
Epoch: 34 Loss D.: 0.5126389861106873
Epoch: 34 Loss G.: 1.0631219148635864
Epoch: 35 Loss D.: 0.5444536209106445
```

```
Epoch: 35 Loss G.: 1.1541411876678467
Epoch: 36 Loss D.: 0.5781678557395935
Epoch: 36 Loss G.: 0.9662452936172485
Epoch: 37 Loss D.: 0.5835438966751099
Epoch: 37 Loss G.: 0.9496454000473022
Epoch: 38 Loss D.: 0.5879670977592468
Epoch: 38 Loss G.: 1.0505330562591553
Epoch: 39 Loss D.: 0.5950089693069458
Epoch: 39 Loss G.: 1.1801791191101074
Epoch: 40 Loss D.: 0.5534310340881348
Epoch: 40 Loss G.: 1.265071988105774
Epoch: 41 Loss D.: 0.5562151670455933
Epoch: 41 Loss G.: 0.9334210157394409
Epoch: 42 Loss D.: 0.5207641124725342
Epoch: 42 Loss G.: 0.8960757851600647
Epoch: 43 Loss D.: 0.5479260087013245
Epoch: 43 Loss G.: 0.97198086977005
Epoch: 44 Loss D.: 0.6090037822723389
Epoch: 44 Loss G.: 0.9844634532928467
Epoch: 45 Loss D.: 0.6015199422836304
Epoch: 45 Loss G.: 1.0163099765777588
Epoch: 46 Loss D.: 0.59343421459198
Epoch: 46 Loss G.: 1.1771951913833618
Epoch: 47 Loss D.: 0.5294280052185059
Epoch: 47 Loss G.: 1.0949243307113647
Epoch: 48 Loss D.: 0.576279878616333
Epoch: 48 Loss G.: 1.0430094003677368
Epoch: 49 Loss D.: 0.5563277006149292
Epoch: 49 Loss G.: 1.2212655544281006
```

```
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

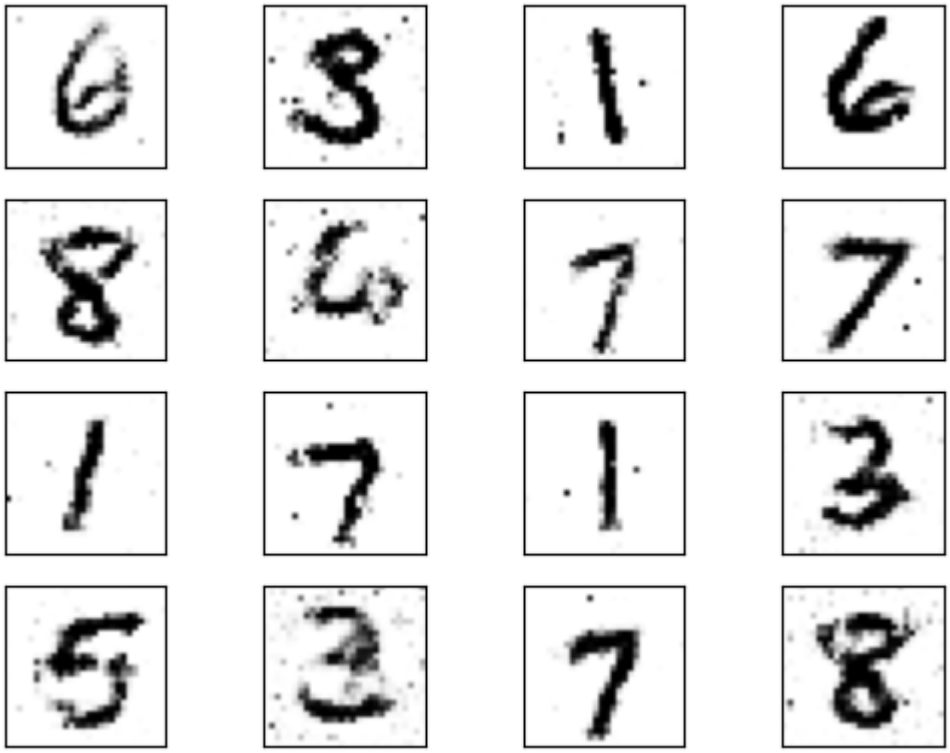
Para generar dígitos escritos a mano, es necesario tomar algunas muestras aleatorias del espacio latente y alimentarlas al generador.

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.

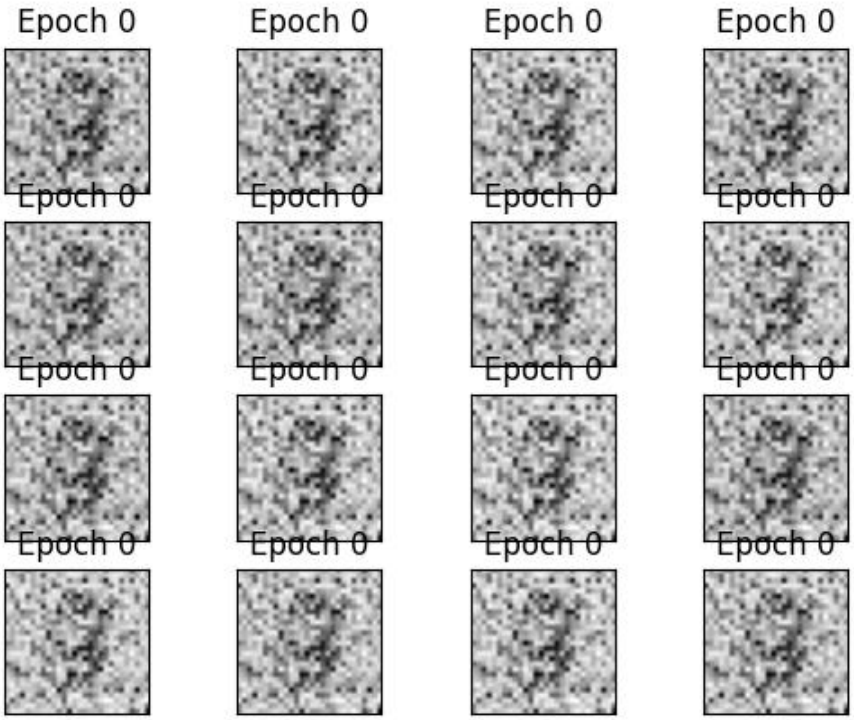
Se puede observar que al comienzo del proceso de entrenamiento, las imágenes generadas son completamente aleatorias. A medida que avanza el entrenamiento, el generador aprende la distribución de los datos reales y, a algunas épocas, algunos dígitos generados ya se asemejan a los datos reales.

```
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?

La diferencia es que entre la primer y segunda parte se utilizan distintas delimitantes en el generador, además de que en esta parte (además de ser más tardada) analiza toda la data (sin muestreo) y en el generetor tenia más capas, además de que su forma de clasificacion se ve reflejado en el scatter.

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

Los bocetos de los trazos permiten distinguir de manera aceptable los números de tal manera que una persistencia del mismo hace factible su distinción de manera aceptable.

- ¿Cómo mejoraría los modelos?

Utilizando más epocas, poniendo más filtros de activación además de intentar distintos parametros dentro de las funciones de activación y una posible forma de mejora es aumentar la cantidad de pruebas que esta analizando

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

Entre las primeras 10 epocas se puede apreciar que las imagenes no permiten con claridad observar el número correspondiente, además de que existen muchos grises pixeles dentro del mismo, a partir de estas 10 epocas las imagenes tienen una clasificación más clara respecto a los coloes (hay más blancos y oscuros) además de que el uso de grises es mucho menos recurrente. Los trazos realizados son más claros al final de las 5 ulimas epocas y el uso de espacios en blanco es más delimitado, de tal manera que el trazo es más fino

```
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
```

70 / 70 marks (100.0%)