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
- 38.1 kB 525.4 kB/s 0:00:00
etadata (setup.py) ... e=jhwutils-1.0-py3-none-any.whl size=33801
sha256=95da95c2e25a0a57ef01ddae5509e04a39aa301ded7dc521c1301802a791371
c
  Stored in directory:
/private/var/folders/h4/rlgjjv6s50z2sflx2_9bt49h0000gn/T/pip-ephem-
wheel-cache-qrtmdzj_/wheels/27/3c/cb/
eb7b3c6ea36b5b54e5746751443be9bb0d73352919033558a2
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
/Users/fredyvelasquez/anaconda3/lib/python3.10/site-packages (0.19.3)
Requirement already satisfied: tifffile>=2019.7.26 in
/Users/fredyvelasquez/anaconda3/lib/python3.10/site-packages (from
scikit-image) (2021.7.2)
Requirement already satisfied: numpy>=1.17.0 in
/Users/fredyvelasquez/anaconda3/lib/python3.10/site-packages (from
scikit-image) (1.23.5)
Requirement already satisfied: pillow!=7.1.0,!=7.1.1,!=8.3.0,>=6.1.0
in /Users/fredyvelasquez/anaconda3/lib/python3.10/site-packages (from
scikit-image) (9.4.0)
Requirement already satisfied: imageio>=2.4.1 in
/Users/fredyvelasquez/anaconda3/lib/python3.10/site-packages (from
scikit-image) (2.26.0)
Requirement already satisfied: PyWavelets>=1.1.1 in
/Users/fredyvelasquez/anaconda3/lib/python3.10/site-packages (from
scikit-image) (1.4.1)
Requirement already satisfied: scipy>=1.4.1 in
/Users/fredyvelasquez/anaconda3/lib/python3.10/site-packages (from
scikit-image) (1.10.0)
Requirement already satisfied: packaging>=20.0 in
/Users/fredyvelasquez/anaconda3/lib/python3.10/site-packages (from
scikit-image) (22.0)
Requirement already satisfied: networkx>=2.2 in
/Users/fredyvelasquez/anaconda3/lib/python3.10/site-packages (from
scikit-image) (2.8.4)
Collecting https://github.com/AlbertS789/lautils/zipball/master
  Downloading https://github.com/AlbertS789/lautils/zipball/master
- 4.2 kB 2.6 MB/s 0:00:00
etadata (setup.py) ... e=lautils-1.0-py3-none-any.whl size=2825
sha256=0cc7ba2c97c86902cc10fe7e68023d654a9ac434b10e452254d7b8fa3e74ae4
f
  Stored in directory:
/private/var/folders/h4/rlgjjv6s50z2sflx2_9bt49h0000gn/T/pip-ephem-
wheel-cache-pro0gizq/wheels/16/3a/
a0/5fbae86e17ef6bb8ed057aa04b591584005d1212c72d69fc70
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 = "201011"
firma_mecanografiada_1 = "Fredy Velasquez"
carne_2 =  "20460"
firma_mecanografiada_2 = "Angel Higueros"
# 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)
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

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.

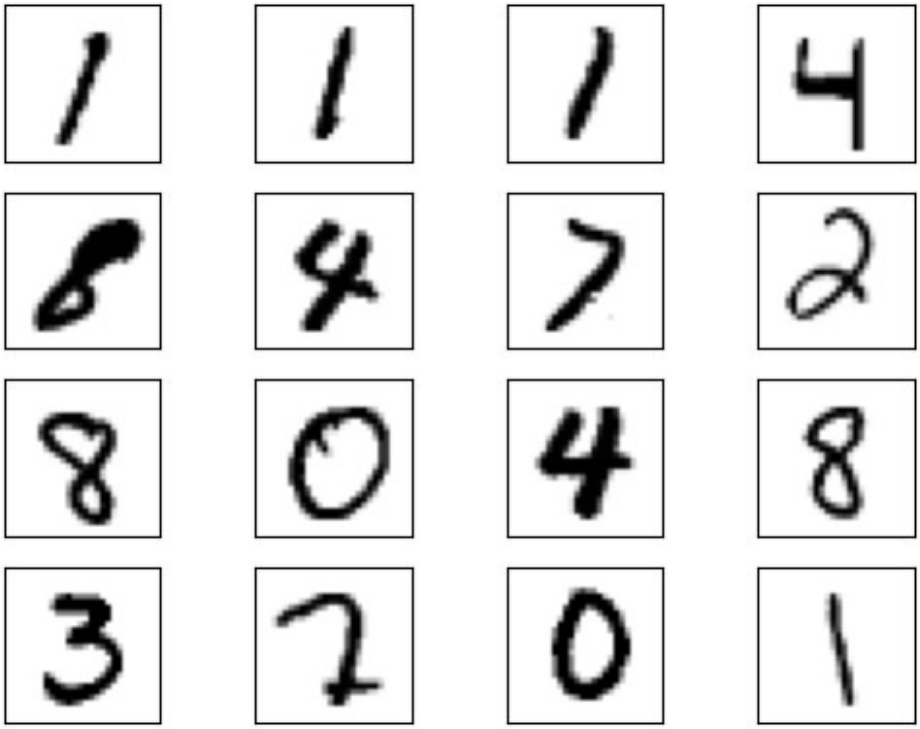
Cabe decir que se puede utilizar Matplotlib para trazar algunas muestras de los datos de entrenamiento. Para mejorar la visualización, se puede usar cmap=gray_r para invertir el mapa de colores y representar los dígitos en negro sobre un fondo blanco:

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%|██████████| 9912422/9912422 [00:00<00:00, 28030181.90it/s]

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%|██████████| 28881/28881 [00:00<00:00, 30466723.80it/s]

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

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



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

100%|██████████| 1648877/1648877 [00:00<00:00, 8420243.72it/s]

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%|██████████| 4542/4542 [00:00<00:00, 18604032.00it/s]

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.

```
import torch.nn as nn

class Discriminator(nn.Module):
    def __init__(self):
        super().__init__()
        self.model = nn.Sequential(
            nn.Linear(784, 1024),
            nn.ReLU(),
            nn.Dropout(0.4),
            nn.Linear(1024, 512),
            nn.ReLU(),
            nn.Dropout(0.4),
            nn.Linear(512, 256),
            nn.ReLU(),
            nn.Dropout(0.4),
            nn.Linear(256, 1),
            nn.Sigmoid()
        )

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

import torch.nn as nn

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()
        )

    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.

```
import os

list_images = []

# Directorio donde se guardarán las imágenes generadas
path_imgs = 'generated_images/'

if not os.path.exists(path_imgs):
    os.makedirs(path_imgs)

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
        discriminator.zero_grad()
        output_discriminator = discriminator(all_samples)
        loss_discriminator = loss_function(
            output_discriminator, all_samples_labels
        )
        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
        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
        )

        loss_generator.backward()
        optimizer_generator.step()

        # Guardamos las imágenes
        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.: 0.6018939018249512
Epoch: 0 Loss G.: 0.41723933815956116
Epoch: 1 Loss D.: 0.09307156503200531
Epoch: 1 Loss G.: 6.188798427581787
Epoch: 2 Loss D.: 0.10932619869709015
Epoch: 2 Loss G.: 5.1541337966918945
Epoch: 3 Loss D.: 0.05704091116786003
Epoch: 3 Loss G.: 5.1883392333984375
Epoch: 4 Loss D.: 0.22556592524051666
Epoch: 4 Loss G.: 3.8153076171875
Epoch: 5 Loss D.: 0.11532564461231232
Epoch: 5 Loss G.: 4.056291103363037
Epoch: 6 Loss D.: 0.25395312905311584
Epoch: 6 Loss G.: 2.5699188709259033
Epoch: 7 Loss D.: 0.3223820626735687
Epoch: 7 Loss G.: 2.7495899200439453
Epoch: 8 Loss D.: 0.21418136358261108
Epoch: 8 Loss G.: 2.762181520462036
Epoch: 9 Loss D.: 0.36796581745147705
Epoch: 9 Loss G.: 1.4140710830688477
Epoch: 10 Loss D.: 0.3905206620693207
Epoch: 10 Loss G.: 1.7615777254104614
Epoch: 11 Loss D.: 0.510434091091156
Epoch: 11 Loss G.: 1.5997064113616943
Epoch: 12 Loss D.: 0.489119291305542
Epoch: 12 Loss G.: 1.5284250974655151
Epoch: 13 Loss D.: 0.2772318124771118
Epoch: 13 Loss G.: 1.6911265850067139
Epoch: 14 Loss D.: 0.3843252956867218
Epoch: 14 Loss G.: 1.1549787521362305
Epoch: 15 Loss D.: 0.4387287199497223
Epoch: 15 Loss G.: 1.2881702184677124
Epoch: 16 Loss D.: 0.4993837773799896
Epoch: 16 Loss G.: 1.293308138847351
Epoch: 17 Loss D.: 0.5667741298675537
```

```
Epoch: 17 Loss G.: 1.2442429065704346
Epoch: 18 Loss D.: 0.514097273349762
Epoch: 18 Loss G.: 1.208249568939209
Epoch: 19 Loss D.: 0.5383874773979187
Epoch: 19 Loss G.: 1.281370759010315
Epoch: 20 Loss D.: 0.5915012359619141
Epoch: 20 Loss G.: 0.9429638385772705
Epoch: 21 Loss D.: 0.6724088788032532
Epoch: 21 Loss G.: 1.125020146369934
Epoch: 22 Loss D.: 0.5082643628120422
Epoch: 22 Loss G.: 1.051515817642212
Epoch: 23 Loss D.: 0.5535485148429871
Epoch: 23 Loss G.: 1.2780512571334839
Epoch: 24 Loss D.: 0.5286291837692261
Epoch: 24 Loss G.: 0.9265242218971252
Epoch: 25 Loss D.: 0.5359048247337341
Epoch: 25 Loss G.: 1.2618166208267212
Epoch: 26 Loss D.: 0.5385575294494629
Epoch: 26 Loss G.: 1.2400779724121094
Epoch: 27 Loss D.: 0.5702499151229858
Epoch: 27 Loss G.: 0.9368977546691895
Epoch: 28 Loss D.: 0.5742008090019226
Epoch: 28 Loss G.: 1.220198154449463
Epoch: 29 Loss D.: 0.568840742111206
Epoch: 29 Loss G.: 0.9578932523727417
Epoch: 30 Loss D.: 0.5663488507270813
Epoch: 30 Loss G.: 0.8187006711959839
Epoch: 31 Loss D.: 0.7054715156555176
Epoch: 31 Loss G.: 0.8499274253845215
Epoch: 32 Loss D.: 0.6388115882873535
Epoch: 32 Loss G.: 1.060251235961914
Epoch: 33 Loss D.: 0.6618168354034424
Epoch: 33 Loss G.: 1.0224159955978394
Epoch: 34 Loss D.: 0.5894342660903931
Epoch: 34 Loss G.: 1.0231651067733765
Epoch: 35 Loss D.: 0.6360452175140381
Epoch: 35 Loss G.: 1.0648269653320312
Epoch: 36 Loss D.: 0.6239414215087891
Epoch: 36 Loss G.: 0.8446050882339478
Epoch: 37 Loss D.: 0.5665749311447144
Epoch: 37 Loss G.: 0.8580578565597534
Epoch: 38 Loss D.: 0.5548936724662781
Epoch: 38 Loss G.: 0.7437372803688049
Epoch: 39 Loss D.: 0.6147865056991577
Epoch: 39 Loss G.: 0.7675015330314636
Epoch: 40 Loss D.: 0.6427566409111023
Epoch: 40 Loss G.: 0.8911498785018921
Epoch: 41 Loss D.: 0.6452158689498901
Epoch: 41 Loss G.: 0.7302830219268799
```

```
Epoch: 42 Loss D.: 0.7090854048728943
Epoch: 42 Loss G.: 0.8436764478683472
Epoch: 43 Loss D.: 0.6384574770927429
Epoch: 43 Loss G.: 0.8331651091575623
Epoch: 44 Loss D.: 0.7309616804122925
Epoch: 44 Loss G.: 0.7723614573478699
Epoch: 45 Loss D.: 0.6386353373527527
Epoch: 45 Loss G.: 0.8594781756401062
Epoch: 46 Loss D.: 0.5956626534461975
Epoch: 46 Loss G.: 0.8964163661003113
Epoch: 47 Loss D.: 0.6395959854125977
Epoch: 47 Loss G.: 0.9810835719108582
Epoch: 48 Loss D.: 0.6311392784118652
Epoch: 48 Loss G.: 0.8573312759399414
Epoch: 49 Loss D.: 0.6110059022903442
Epoch: 49 Loss G.: 0.9243584871292114
```

```
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')
```

```
<IPython.core.display.HTML object>

<IPython.core.display.HTML object>
```

Validación del Resultado

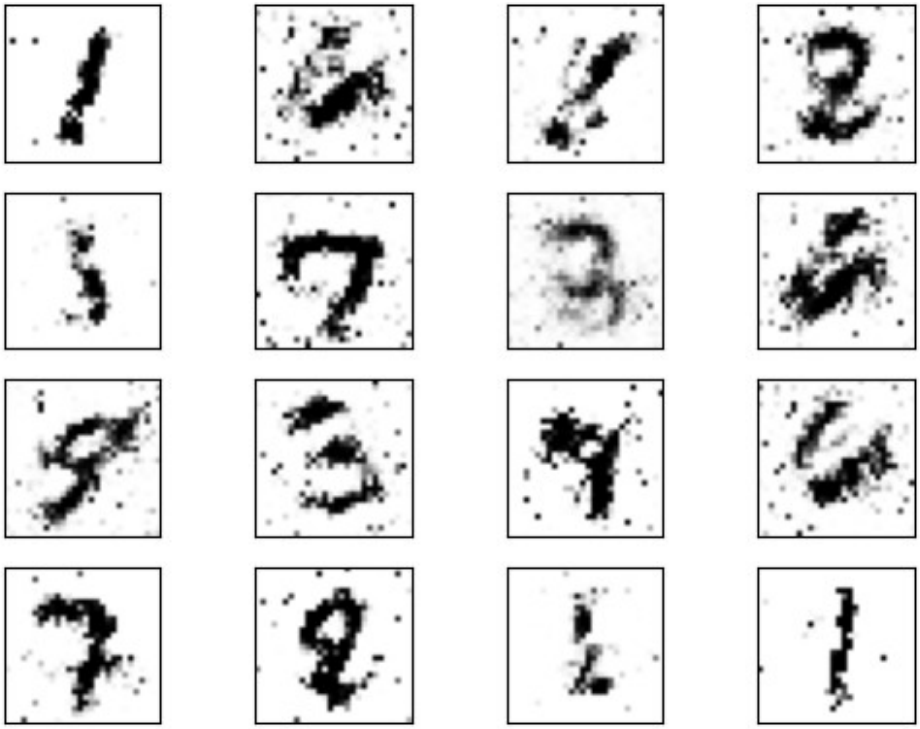
Para generar dígitos escritos a mano, es necesario tomar algunas muestras aleatorias del espacio latente y alimentarlas al generador.

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([])
```



RESPUESTAS:

- Las características que distinguen los modelos empleados en la primera parte de los utilizados en esta sección son notables. En la primera parte, los modelos se orientaban hacia la generación a través de patrones, en contraste con la segunda parte que involucra un análisis exhaustivo de los datos completos sin ningún muestreo. El generador en la primera parte presentaba una estructura con capas más abundantes y diferentes límites, mientras que en la segunda parte se ajusta de forma visual en un scatter plot para representar su capacidad de clasificación.
- La calidad de las imágenes generadas es razonablemente buena. Los trazos bosquejados permiten identificar de manera adecuada los dígitos numéricos, y su duración en la persistencia contribuye significativamente a su reconocibilidad.
- Para optimizar los modelos, se podrían considerar diversas estrategias. Aumentar la cantidad de épocas podría influir positivamente en la convergencia hacia soluciones

más precisas. También sería útil experimentar con variaciones en las funciones de activación, junto con la adaptación de parámetros para alcanzar un equilibrio óptimo en la generación de imágenes.

- Al examinar el GIF generado, se puede notar una evolución gradual al avanzar las épocas. En las etapas iniciales (primeras 10 eps), las imágenes resultantes carecen de nitidez y distintividad, con una presencia considerable de tonos grises. A medida que transcurren las épocas, las imágenes muestran una clasificación más nítida en términos de colores, evidenciando un marcado contraste entre tonos oscuros y claros. En particular, el uso de tonos grises disminuye significativamente. Hacia las últimas 5 eps, los trazos generados adquieren mayor definición, y la utilización de espacios en blanco se vuelve más selectiva, resultando en trazos más delicados y precisos.

```
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
```

```
<IPython.core.display.HTML object>
```