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/maste
```

```
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, _ch
import jhwutils.image_audio as ia
import jhwutils.tick as tick
from lautils.gradeutils import new_representation, hex_to_float, compare_numbers, compare_lists_b
```

```
###
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 =
# firma_mecanografiada_1 =
# carne_2 =
# firma_mecanografiada_2 =
# YOUR CODE HERE
carne_1 = "20880"
firma_mecanografiada_1 = "Sebastian Aristondo"
carne_2 = "20293"
firma_mecanografiada_2 = "Daniel Gonzalez"
# raise NotImplementedError()
```

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

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, $(M_1, \ldots, M_n)$ y $(S_1, \ldots, S_n)$, 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:

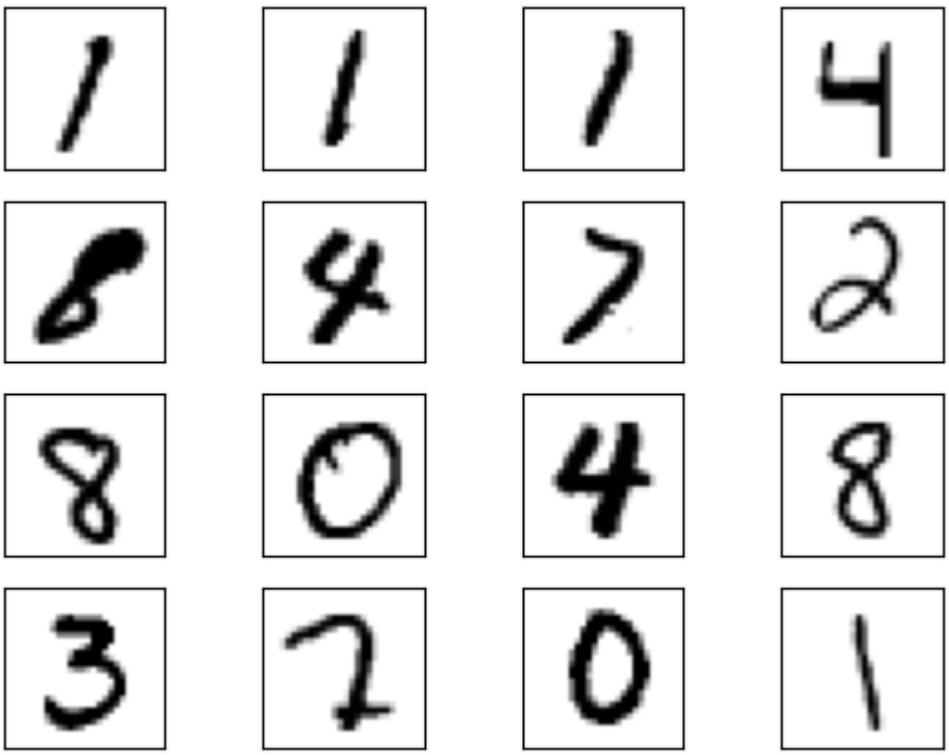
Como se puede ver más adelante, hay dígitos con diferentes estilos de escritura. A medida que la GAN aprende la distribución de los datos, también generará dígitos con diferentes estilos de escritura.

```
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()
            # 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(
            # 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 grafic
# path_imgs =
# YOUR CODE HERE
path_imgs = "./imgs/"

#seed_all(seed_)

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

lr = 0.0001
num_epochs = 100
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
        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
        generator.zero_grad()
        generated_samples = generator(latent_space_samples)
        output_discriminator = discriminator(generated_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.: 0.5794613361358643
Epoch: 0 Loss G.: 0.48083001375198364
Epoch: 1 Loss D.: 0.0539088174700737
Epoch: 1 Loss G.: 5.812363624572754
Epoch: 2 Loss D.: 0.0220984797924757
Epoch: 2 Loss G.: 5.781692981719971
Epoch: 3 Loss D.: 0.036894652992486954
Epoch: 3 Loss G.: 5.553284645080566
Epoch: 4 Loss D.: 0.15485869348049164
Epoch: 4 Loss G.: 6.769937515258789
Epoch: 5 Loss D.: 0.022755704820156097
Epoch: 5 Loss G.: 3.9356765747070312
Epoch: 6 Loss D.: 0.17136110365390778
Epoch: 6 Loss G.: 3.5466156005859375
Epoch: 7 Loss D.: 0.37119582295417786
Epoch: 7 Loss G.: 2.654788017272949
Epoch: 8 Loss D.: 0.3179759979248047
Epoch: 8 Loss G.: 2.285111904144287
Epoch: 9 Loss D.: 0.25608009099960327
Epoch: 9 Loss G.: 2.109407663345337
Epoch: 10 Loss D.: 0.34116506576538086
Epoch: 10 Loss G.: 2.3068079948425293
Epoch: 11 Loss D.: 0.25385382771492004
Epoch: 11 Loss G.: 1.7565871477127075
Epoch: 12 Loss D.: 0.3878878951072693
Epoch: 12 Loss G.: 1.442704677581787
Epoch: 13 Loss D.: 0.42220890522003174
Epoch: 13 Loss G.: 1.408586025238037
Epoch: 14 Loss D.: 0.29440051317214966
Epoch: 14 Loss G.: 1.803763747215271
Epoch: 15 Loss D.: 0.38363978266716003
Epoch: 15 Loss G.: 1.2464557886123657
Epoch: 16 Loss D.: 0.47423315048217773
Epoch: 16 Loss G.: 1.352437973022461
Epoch: 17 Loss D.: 0.5367080569267273
Epoch: 17 Loss G.: 1.204041600227356
Epoch: 18 Loss D.: 0.5197944641113281
Epoch: 18 Loss G.: 1.2552108764648438
```

```
Epoch: 19 Loss D.: 0.45158255100250244
Epoch: 19 Loss G.: 1.2009464502334595
Epoch: 20 Loss D.: 0.4217942953109741
Epoch: 20 Loss G.: 1.3807621002197266
Epoch: 21 Loss D.: 0.5580126643180847
Epoch: 21 Loss G.: 1.2244584560394287
Epoch: 22 Loss D.: 0.48951831459999084
Epoch: 22 Loss G.: 1.0421112775802612
Epoch: 23 Loss D.: 0.4797348380088806
Epoch: 23 Loss G.: 1.0483325719833374
Epoch: 24 Loss D.: 0.37687018513679504
Epoch: 24 Loss G.: 1.2524746656417847
Epoch: 25 Loss D.: 0.5797836780548096
Epoch: 25 Loss G.: 1.2007484436035156
Epoch: 26 Loss D.: 0.46257179975509644
Epoch: 26 Loss G.: 1.3704063892364502
Epoch: 27 Loss D.: 0.5865782499313354
Epoch: 27 Loss G.: 1.1765080690383911
Epoch: 28 Loss D.: 0.5387046933174133
Epoch: 28 Loss G.: 1.068549633026123
Epoch: 29 Loss D.: 0.5799427032470703
Epoch: 29 Loss G.: 1.1467679738998413
Epoch: 30 Loss D.: 0.5502262711524963
Epoch: 30 Loss G.: 0.9462218284606934
Epoch: 31 Loss D.: 0.6127558946609497
Epoch: 31 Loss G.: 1.1022870540618896
Epoch: 32 Loss D.: 0.5034487247467041
Epoch: 32 Loss G.: 1.1657713651657104
Epoch: 33 Loss D.: 0.6262367367744446
Epoch: 33 Loss G.: 0.9859303832054138
Epoch: 34 Loss D.: 0.5052598118782043
Epoch: 34 Loss G.: 1.0968968868255615
Epoch: 35 Loss D.: 0.6204651594161987
Epoch: 35 Loss G.: 1.129453420639038
Epoch: 36 Loss D.: 0.539236843585968
Epoch: 36 Loss G.: 1.1194326877593994
Epoch: 37 Loss D.: 0.5530719757080078
Epoch: 37 Loss G.: 0.9322466254234314
Epoch: 38 Loss D.: 0.5158356428146362
Epoch: 38 Loss G.: 0.9954675436019897
Epoch: 39 Loss D.: 0.5395253896713257
Epoch: 39 Loss G.: 0.8925639390945435
Epoch: 40 Loss D.: 0.6283810138702393
Epoch: 40 Loss G.: 0.9760441184043884
Epoch: 41 Loss D.: 0.5091524720191956
Epoch: 41 Loss G.: 0.9726918935775757
Epoch: 42 Loss D.: 0.7071595788002014
Epoch: 42 Loss G.: 1.1121106147766113
Epoch: 43 Loss D.: 0.5962830781936646
Epoch: 43 Loss G.: 1.0384501218795776
```

```
Epoch: 44 Loss D.: 0.7343319654464722
Epoch: 44 Loss G.: 0.9763522148132324
Epoch: 45 Loss D.: 0.5165814161300659
Epoch: 45 Loss G.: 1.1424474716186523
Epoch: 46 Loss D.: 0.6568767428398132
Epoch: 46 Loss G.: 0.9742985963821411
Epoch: 47 Loss D.: 0.5985961556434631
Epoch: 47 Loss G.: 1.0522743463516235
Epoch: 48 Loss D.: 0.5785568356513977
Epoch: 48 Loss G.: 0.9655101895332336
Epoch: 49 Loss D.: 0.6207848787307739
Epoch: 49 Loss G.: 1.0303587913513184
Epoch: 50 Loss D.: 0.5457786321640015
Epoch: 50 Loss G.: 1.1791088581085205
Epoch: 51 Loss D.: 0.5826497077941895
Epoch: 51 Loss G.: 0.9461702108383179
Epoch: 52 Loss D.: 0.5915989279747009
Epoch: 52 Loss G.: 0.9751262664794922
Epoch: 53 Loss D.: 0.5017040371894836
Epoch: 53 Loss G.: 0.9090300798416138
Epoch: 54 Loss D.: 0.5894002914428711
Epoch: 54 Loss G.: 1.0716569423675537
Epoch: 55 Loss D.: 0.49750590324401855
Epoch: 55 Loss G.: 1.0234401226043701
Epoch: 56 Loss D.: 0.5694664716720581
Epoch: 56 Loss G.: 0.938962459564209
Epoch: 57 Loss D.: 0.5584607720375061
Epoch: 57 Loss G.: 1.1258388757705688
Epoch: 58 Loss D.: 0.5659523010253906
Epoch: 58 Loss G.: 0.9243825674057007
Epoch: 59 Loss D.: 0.5099338293075562
Epoch: 59 Loss G.: 1.097508192062378
Epoch: 60 Loss D.: 0.6299842596054077
Epoch: 60 Loss G.: 0.9109359979629517
Epoch: 61 Loss D.: 0.621730387210846
Epoch: 61 Loss G.: 0.9456491470336914
Epoch: 62 Loss D.: 0.5626068115234375
Epoch: 62 Loss G.: 0.8615549802780151
Epoch: 63 Loss D.: 0.614894449710846
Epoch: 63 Loss G.: 1.046862244606018
Epoch: 64 Loss D.: 0.5817046165466309
Epoch: 64 Loss G.: 1.0380115509033203
Epoch: 65 Loss D.: 0.5336825847625732
Epoch: 65 Loss G.: 0.9501345753669739
Epoch: 66 Loss D.: 0.5683041214942932
Epoch: 66 Loss G.: 0.9639052152633667
Epoch: 67 Loss D.: 0.5710508823394775
Epoch: 67 Loss G.: 1.0858683586120605
Epoch: 68 Loss D.: 0.5390716791152954
Epoch: 68 Loss G.: 1.0155216455459595
```

```
Epoch: 69 Loss D.: 0.5887241363525391
Epoch: 69 Loss G.: 0.9371286630630493
Epoch: 70 Loss D.: 0.6185636520385742
Epoch: 70 Loss G.: 0.9176428914070129
Epoch: 71 Loss D.: 0.6318212747573853
Epoch: 71 Loss G.: 1.0062944889068604
Epoch: 72 Loss D.: 0.5692639350891113
Epoch: 72 Loss G.: 1.0835679769515991
Epoch: 73 Loss D.: 0.5492775440216064
Epoch: 73 Loss G.: 0.9415536522865295
Epoch: 74 Loss D.: 0.516574501991272
Epoch: 74 Loss G.: 1.0112316608428955
Epoch: 75 Loss D.: 0.6207066774368286
Epoch: 75 Loss G.: 1.0754507780075073
Epoch: 76 Loss D.: 0.5636848211288452
Epoch: 76 Loss G.: 1.0512584447860718
Epoch: 77 Loss D.: 0.5395544767379761
Epoch: 77 Loss G.: 1.0795409679412842
Epoch: 78 Loss D.: 0.5783387422561646
Epoch: 78 Loss G.: 1.1168142557144165
Epoch: 79 Loss D.: 0.615585446357727
Epoch: 79 Loss G.: 0.7968440055847168
Epoch: 80 Loss D.: 0.5324131846427917
Epoch: 80 Loss G.: 1.0754544734954834
Epoch: 81 Loss D.: 0.5510247945785522
Epoch: 81 Loss G.: 0.8500566482543945
Epoch: 82 Loss D.: 0.6582499742507935
Epoch: 82 Loss G.: 1.0931975841522217
Epoch: 83 Loss D.: 0.6272894144058228
Epoch: 83 Loss G.: 1.0638673305511475
Epoch: 84 Loss D.: 0.5568708777427673
Epoch: 84 Loss G.: 1.0246894359588623
Epoch: 85 Loss D.: 0.5678132772445679
Epoch: 85 Loss G.: 0.9472911357879639
Epoch: 86 Loss D.: 0.5999842882156372
Epoch: 86 Loss G.: 1.095841884613037
Epoch: 87 Loss D.: 0.594698965549469
Epoch: 87 Loss G.: 0.9961576461791992
Epoch: 88 Loss D.: 0.5666934251785278
Epoch: 88 Loss G.: 1.1287649869918823
Epoch: 89 Loss D.: 0.5732617378234863
Epoch: 89 Loss G.: 0.9224717617034912
Epoch: 90 Loss D.: 0.5661096572875977
Epoch: 90 Loss G.: 1.0052828788757324
Epoch: 91 Loss D.: 0.5994973182678223
Epoch: 91 Loss G.: 1.198476791381836
Epoch: 92 Loss D.: 0.6475637555122375
Epoch: 92 Loss G.: 0.9221539497375488
Epoch: 93 Loss D.: 0.5353103280067444
Epoch: 93 Loss G.: 0.9971413612365723
```

```
Epoch: 94 Loss D.: 0.5787596106529236
Epoch: 94 Loss G.: 0.9588674306869507
Epoch: 95 Loss D.: 0.5410958528518677
Epoch: 95 Loss G.: 0.8472305536270142
Epoch: 96 Loss D.: 0.4926695227622986
Epoch: 96 Loss G.: 0.9608584642410278
Epoch: 97 Loss D.: 0.5611334443092346
Epoch: 97 Loss G.: 1.080500841140747
Epoch: 98 Loss D.: 0.5886278748512268
Epoch: 98 Loss G.: 1.2082140445709229
Epoch: 99 Loss D.: 0.5574293732643127
Epoch: 99 Loss G.: 1.257434368133545
```

```
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.

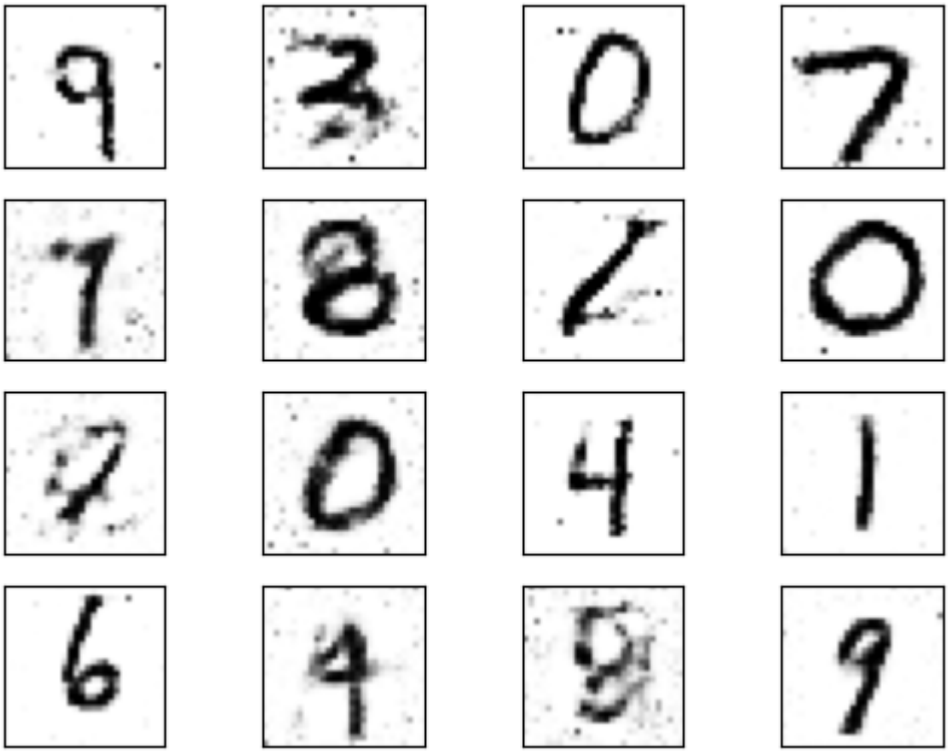
Para trazar generated_samples, es necesario mover los datos de vuelta a la CPU en caso de que estén en la GPU. Para ello, simplemente se puede llamar a .cpu(). Como se hizo anteriormente, también es necesario llamar a .detach() antes de usar Matplotlib para trazar los datos.

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))
```

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

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? * Se pueden observar algunas diferencias entre modelos. Por un lado, la preparación de los datos es más compleja, debido a las transformaciones que se deben realizar, de forma que el input no esté plagado de 0s y esto afecte el entrenamiento. También, la arquitectura del discriminador se modifica por la necesidad de usar las 784 neuronas debido a cada imágen que es una matriz de 28x28. También la arquitectura del generador se modifica, teniendo que tomar en consideración la generación de los 784 pixeles a partir de las 100 dimensiones del espacio latente. Finalmente, en el generador es necesario poner una Tanh, a diferencia del primero modelo donde no hubo que poner nada al final, esto para que los valores estén entre -1 y 1 y se adecuen a la normalización que se hizo en la preparación de los datos. * ¿Qué tan bien se han creado las imagenes esperadas? * Se puede observar que las imágenes creadas son regulares. En ocasiones no se puede distinguir bien entre 9 y 4. Hay dos que no se puede distinguir que número son. Las que sí se pueden distinguir parece que se hubieran escrito con carbón. En general no están mal, pero se pueden mejorar. * ¿Cómo mejoraría los modelos? * Tuvimos varios problemas con el kernel ya que nos corrio a la primera obtuvimos todas las marcas, pero se reinicio el kernel. Lo que nos obligo a correr nuevamente el notebook teniendo un gran problema. Ahora la perdida del generador en la epoca 4 se disparaba a 90.54. Tras reinicar el kernel y volverlo a correr no obteniamos el loss esperado, asi que decidimos agregarle un 0 al learning rate y agregar 10 epcoas más. Lo que resulto en disminuir drasticamente la perida en ambos modelos. Esta experiencia nos permite concluir que entrenar con más epocas el modelo y con un learning rate menor podemos hacer que mejore el modelo. Cabe destacar que mejorar la arquitectura agregando alguna cnn también podria potencial al modelo/ * Observe el GIF creado, y describa la evolución que va viendo al pasar de las epocas * Es muy interesante ver la evolución que tuvo el modelo, lo más interesante es la época 2. Si vemos el gif nos podemos dar cuenta que en la epoca 2 todas las imagenes son iguales. Muy seguramente se debe a que el 4,9 y 7 tienen una forma muy similar un tallo y luego uno o más líneas que salen de la copa. El modelo todavía estaba ajustando sus pesos pero al tener 3 dígitos similares ya comenzaban a tomar forma. Luego en las siguiente épocas muchos números ya comenzaron a tener forma unicamente se tenian alguno que otro pixel negro “perdido”. Los números de las épocas intermedias se ven como difuminados. A diferencia de los finales que ya son más claros.

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

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

70 / 70 marks (100.0%)