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

In [1]:
```
# Una vez instalada la librería por favor, recuerden volverla a comentar.
# !pip install -U --force-reinstall --no-cache https://github.com/johnhw/jhwutils/z
# !pip install scikit-image
# !pip install -U --force-reinstall --no-cache https://github.com/AlbertS789/lautil
```

In [2]:
```
!pip install -U --force-reinstall --no-cache https://github.com/johnhw/jhwutils/zip
!pip install scikit-image
!pip install -U --force-reinstall --no-cache https://github.com/AlbertS789/lautils/
```

```
Collecting https://github.com/johnhw/jhwutils/zipball/master
  Downloading https://github.com/johnhw/jhwutils/zipball/master
     - 119.1 kB 41.6 MB/s 0:00:00
  Preparing metadata (setup.py) ... done
Building wheels for collected packages: jhwutils
  Building wheel for jhwutils (setup.py) ... done
  Created wheel for jhwutils: filename=jhwutils-1.3-py3-none-any.whl size=41854 sha2
56=e9e89b308e2ab39bfaed0ccd03a89cdca75653a8b35a49676272d9fd99d3a9d6
  Stored in directory: /tmp/pip-ephem-wheel-cache-1mosp8lf/wheels/a8/e7/e3/9542f8e41
59ba644c6acd9f78babbe8489bb72667fb02ac54d
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/python3.11/dist-packag
es (0.25.2)
Requirement already satisfied: numpy>=1.24 in /usr/local/lib/python3.11/dist-package
s (from scikit-image) (2.0.2)
Requirement already satisfied: scipy>=1.11.4 in /usr/local/lib/python3.11/dist-packa
ges (from scikit-image) (1.16.1)
Requirement already satisfied: networkx>=3.0 in /usr/local/lib/python3.11/dist-packa
ges (from scikit-image) (3.5)
Requirement already satisfied: pillow>=10.1 in /usr/local/lib/python3.11/dist-packag
es (from scikit-image) (11.3.0)
Requirement already satisfied: imageio!=2.35.0,>=2.33 in /usr/local/lib/python3.11/d
ist-packages (from scikit-image) (2.37.0)
Requirement already satisfied: tifffile>=2022.8.12 in /usr/local/lib/python3.11/dist
-packages (from scikit-image) (2025.6.11)
Requirement already satisfied: packaging>=21 in /usr/local/lib/python3.11/dist-packa
ges (from scikit-image) (25.0)
Requirement already satisfied: lazy-loader>=0.4 in /usr/local/lib/python3.11/dist-pa
ckages (from scikit-image) (0.4)
Collecting https://github.com/AlbertS789/lautils/zipball/master
  Downloading https://github.com/AlbertS789/lautils/zipball/master (4.2 kB)
  Preparing metadata (setup.py) ... done
Building wheels for collected packages: lautils
  Building wheel for lautils (setup.py) ... done
  Created wheel for lautils: filename=lautils-1.0-py3-none-any.whl size=2826 sha256=
b39e7d3f8e57b23a11ac84e9a4927ed23c4121affaa0ed160ed1d8268eebedcb
  Stored in directory: /tmp/pip-ephem-wheel-cache-jjpge2ov/wheels/1a/50/ba/b3ceb9379
49f5894a896b68af5b5fdb598e50244141063e4db
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
```

In [3]:
```
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, a
import jhwutils.image_audio as ia
import jhwutils.tick as tick
from lautils.gradeutils import new_representation, hex_to_float, compare_numbers, c

###
tick.reset_marks()

%matplotlib inline
```

In [4]:
```
# 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)

In [5]:
```
# carne_1 =
# firma_mecanografiada_1 =
# carne_2 =
# firma_mecanografiada_2 =
# YOUR CODE HERE
carne_1 = "22397"
firma_mecanografiada_1 = "Josue Marroquin"
carne_2 = "22295"
firma_mecanografiada_2 = "Sebastian Huertas"
```

In [6]:
```
# Deberia poder ver dos checkmarks verdes [0 marks], que indican que su información

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.

In [7]:
```
%pip install torch torchvision
```

```
Requirement already satisfied: torch in /usr/local/lib/python3.11/dist-packages (2.
6.0+cu124)
Requirement already satisfied: torchvision in /usr/local/lib/python3.11/dist-package
s (0.21.0+cu124)
Requirement already satisfied: filelock in /usr/local/lib/python3.11/dist-packages
(from torch) (3.18.0)
Requirement already satisfied: typing-extensions>=4.10.0 in /usr/local/lib/python3.1
1/dist-packages (from torch) (4.14.1)
Requirement already satisfied: networkx in /usr/local/lib/python3.11/dist-packages
(from torch) (3.5)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.11/dist-packages (fr
om torch) (3.1.6)
Requirement already satisfied: fsspec in /usr/local/lib/python3.11/dist-packages (fr
om torch) (2025.3.0)
Requirement already satisfied: nvidia-cuda-nvrtc-cu12==12.4.127 in /usr/local/lib/py
thon3.11/dist-packages (from torch) (12.4.127)
Requirement already satisfied: nvidia-cuda-runtime-cu12==12.4.127 in /usr/local/lib/
python3.11/dist-packages (from torch) (12.4.127)
Requirement already satisfied: nvidia-cuda-cupti-cu12==12.4.127 in /usr/local/lib/py
thon3.11/dist-packages (from torch) (12.4.127)
Requirement already satisfied: nvidia-cudnn-cu12==9.1.0.70 in /usr/local/lib/python
3.11/dist-packages (from torch) (9.1.0.70)
Requirement already satisfied: nvidia-cublas-cu12==12.4.5.8 in /usr/local/lib/python
3.11/dist-packages (from torch) (12.4.5.8)
Requirement already satisfied: nvidia-cufft-cu12==11.2.1.3 in /usr/local/lib/python
3.11/dist-packages (from torch) (11.2.1.3)
Requirement already satisfied: nvidia-curand-cu12==10.3.5.147 in /usr/local/lib/pyth
on3.11/dist-packages (from torch) (10.3.5.147)
Requirement already satisfied: nvidia-cusolver-cu12==11.6.1.9 in /usr/local/lib/pyth
on3.11/dist-packages (from torch) (11.6.1.9)
Requirement already satisfied: nvidia-cusparse-cu12==12.3.1.170 in /usr/local/lib/py
thon3.11/dist-packages (from torch) (12.3.1.170)
Requirement already satisfied: nvidia-cusparselt-cu12==0.6.2 in /usr/local/lib/pytho
n3.11/dist-packages (from torch) (0.6.2)
Requirement already satisfied: nvidia-nccl-cu12==2.21.5 in /usr/local/lib/python3.1
1/dist-packages (from torch) (2.21.5)
Requirement already satisfied: nvidia-nvtx-cu12==12.4.127 in /usr/local/lib/python3.
11/dist-packages (from torch) (12.4.127)
Requirement already satisfied: nvidia-nvjitlink-cu12==12.4.127 in /usr/local/lib/pyt
hon3.11/dist-packages (from torch) (12.4.127)
Requirement already satisfied: triton==3.2.0 in /usr/local/lib/python3.11/dist-packa
ges (from torch) (3.2.0)
Requirement already satisfied: sympy==1.13.1 in /usr/local/lib/python3.11/dist-packa
ges (from torch) (1.13.1)
Requirement already satisfied: mpmath<1.4,>=1.1.0 in /usr/local/lib/python3.11/dist-
packages (from sympy==1.13.1->torch) (1.3.0)
Requirement already satisfied: numpy in /usr/local/lib/python3.11/dist-packages (fro
m torchvision) (2.0.2)
Requirement already satisfied: pillow!=8.3.*,>=5.3.0 in /usr/local/lib/python3.11/di
st-packages (from torchvision) (11.3.0)
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.11/dist-pac
kages (from jinja2->torch) (3.0.2)
```

In [8]:
```
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
```

In [9]:
```
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_)
```

In [10]:
```
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:

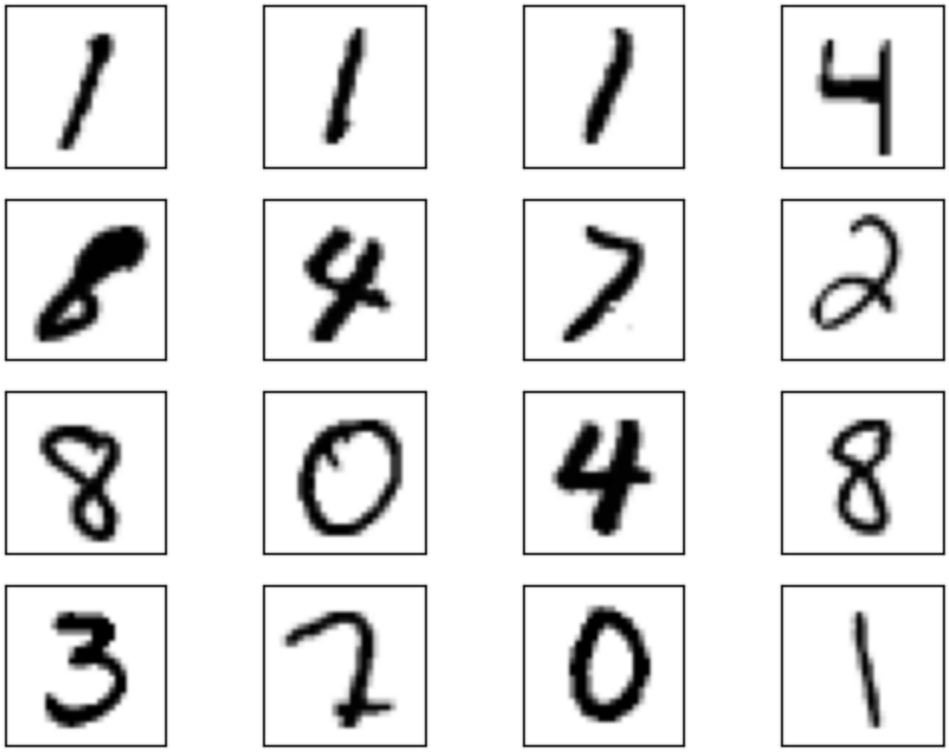
Como se puede ver más adelante, hay dígitos con diferentes estilos de escritura. A medida que la GAN aprende la distribución de los datos, también generará dígitos con diferentes estilos de escritura.

In [11]:
```
transform = transforms.Compose(
    [transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))]
)
```

In [12]:
```
train_set = torchvision.datasets.MNIST(
    root=".", train=True, download=True, transform=transform
)
```

In [13]:
```
batch_size = 32
train_loader = torch.utils.data.DataLoader(
    train_set, batch_size=batch_size, shuffle=True
)
```

In [14]:
```
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.

In [15]:
```
class Discriminator(nn.Module):
    def __init__(self):
        super().__init__()
        self.model = nn.Sequential(
            # 784 -> 1024
            nn.Linear(784, 1024),
            nn.ReLU(inplace=True),
            nn.Dropout(p=0.3),
            # 1024 -> 512
            nn.Linear(1024, 512),
            nn.ReLU(inplace=True),
            nn.Dropout(p=0.3),
            # 512 -> 256
            nn.Linear(512, 256),
            nn.ReLU(inplace=True),
            nn.Dropout(p=0.3),
            # 256 -> 1
            nn.Linear(256, 1),
            nn.Sigmoid(),
        )

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

In [16]:
```
class Generator(nn.Module):
    def __init__(self):
        super().__init__()
        self.model = nn.Sequential(
            # 100 -> 256
            nn.Linear(100, 256),
            nn.ReLU(inplace=True),
            # 256 -> 512
            nn.Linear(256, 512),
            nn.ReLU(inplace=True),
            # 512 -> 1024
            nn.Linear(512, 1024),
            nn.ReLU(inplace=True),
            # 1024 -> 784
            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.

In [17]:
```
list_images = []

path_imgs = "/generated_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
        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
        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 =
        optimizer_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
        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="gra
                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.: 0.579461395740509
Epoch: 0 Loss G.: 0.48083001375198364
Epoch: 1 Loss D.: 0.05051320046186447
Epoch: 1 Loss G.: 6.068345069885254
Epoch: 2 Loss D.: 0.05511898919939995
Epoch: 2 Loss G.: 5.448096752166748
Epoch: 3 Loss D.: 2.1532989194383845e-05
Epoch: 3 Loss G.: 16.962926864624023
Epoch: 4 Loss D.: 0.08928969502449036
Epoch: 4 Loss G.: 6.14481258392334
Epoch: 5 Loss D.: 0.10884732007980347
Epoch: 5 Loss G.: 4.215405464172363
Epoch: 6 Loss D.: 0.09483630955219269
Epoch: 6 Loss G.: 3.21101713180542
Epoch: 7 Loss D.: 0.22285659611225128
Epoch: 7 Loss G.: 2.713930606842041
Epoch: 8 Loss D.: 0.2541957199573517
Epoch: 8 Loss G.: 2.358832836151123
Epoch: 9 Loss D.: 0.380463182926178
Epoch: 9 Loss G.: 2.297487735748291
Epoch: 10 Loss D.: 0.2581935524940491
Epoch: 10 Loss G.: 2.0025949478149414
Epoch: 11 Loss D.: 0.3344593942165375
Epoch: 11 Loss G.: 1.5453968048095703
Epoch: 12 Loss D.: 0.6096579432487488
Epoch: 12 Loss G.: 1.367133617401123
Epoch: 13 Loss D.: 0.4511878490447998
Epoch: 13 Loss G.: 1.7415800094604492
Epoch: 14 Loss D.: 0.4271559715270996
Epoch: 14 Loss G.: 1.3395737409591675
Epoch: 15 Loss D.: 0.5366544723510742
Epoch: 15 Loss G.: 1.3298306465148926
Epoch: 16 Loss D.: 0.47841161489486694
Epoch: 16 Loss G.: 1.2421855926513672
Epoch: 17 Loss D.: 0.6765965223312378
Epoch: 17 Loss G.: 1.3059967756271362
Epoch: 18 Loss D.: 0.46334680914878845
Epoch: 18 Loss G.: 1.3036528825759888
Epoch: 19 Loss D.: 0.4785153567790985
Epoch: 19 Loss G.: 1.006347894668579
Epoch: 20 Loss D.: 0.4795367419719696
Epoch: 20 Loss G.: 1.1087570190429688
Epoch: 21 Loss D.: 0.5375280380249023
Epoch: 21 Loss G.: 1.0615729093551636
Epoch: 22 Loss D.: 0.612403154373169
Epoch: 22 Loss G.: 1.072886347770691
Epoch: 23 Loss D.: 0.5136703252792358
Epoch: 23 Loss G.: 1.1114912033081055
Epoch: 24 Loss D.: 0.5698415637016296
Epoch: 24 Loss G.: 0.8994057178497314
Epoch: 25 Loss D.: 0.5302388072013855
Epoch: 25 Loss G.: 1.0963795185089111
Epoch: 26 Loss D.: 0.5748724341392517
Epoch: 26 Loss G.: 1.123506784439087
Epoch: 27 Loss D.: 0.5208436250686646
Epoch: 27 Loss G.: 1.167156457901001
```

```
Epoch: 28 Loss D.: 0.5705808401107788
Epoch: 28 Loss G.: 0.8487507104873657
Epoch: 29 Loss D.: 0.5819945931434631
Epoch: 29 Loss G.: 1.1690728664398193
Epoch: 30 Loss D.: 0.5531786680221558
Epoch: 30 Loss G.: 0.961190402507782
Epoch: 31 Loss D.: 0.5740288496017456
Epoch: 31 Loss G.: 1.202712893486023
Epoch: 32 Loss D.: 0.6051527857780457
Epoch: 32 Loss G.: 1.0441689491271973
Epoch: 33 Loss D.: 0.5219682455062866
Epoch: 33 Loss G.: 0.9191558361053467
Epoch: 34 Loss D.: 0.5279998183250427
Epoch: 34 Loss G.: 0.8871105909347534
Epoch: 35 Loss D.: 0.5409789681434631
Epoch: 35 Loss G.: 1.0849337577819824
Epoch: 36 Loss D.: 0.5889805555343628
Epoch: 36 Loss G.: 0.9740126132965088
Epoch: 37 Loss D.: 0.6099923849105835
Epoch: 37 Loss G.: 1.041672706604004
Epoch: 38 Loss D.: 0.5416132807731628
Epoch: 38 Loss G.: 0.7644791603088379
Epoch: 39 Loss D.: 0.6080719232559204
Epoch: 39 Loss G.: 0.8506370782852173
Epoch: 40 Loss D.: 0.5814114809036255
Epoch: 40 Loss G.: 0.8692820072174072
Epoch: 41 Loss D.: 0.5883400440216064
Epoch: 41 Loss G.: 1.0908710956573486
Epoch: 42 Loss D.: 0.6001614332199097
Epoch: 42 Loss G.: 0.9915401339530945
Epoch: 43 Loss D.: 0.5511277318000793
Epoch: 43 Loss G.: 1.0228344202041626
Epoch: 44 Loss D.: 0.6676848530769348
Epoch: 44 Loss G.: 0.9827540516853333
Epoch: 45 Loss D.: 0.5821056365966797
Epoch: 45 Loss G.: 0.9951506853103638
Epoch: 46 Loss D.: 0.6078661680221558
Epoch: 46 Loss G.: 1.210347056388855
Epoch: 47 Loss D.: 0.5205300450325012
Epoch: 47 Loss G.: 0.9726237654685974
Epoch: 48 Loss D.: 0.5471475720405579
Epoch: 48 Loss G.: 0.9798892736434937
Epoch: 49 Loss D.: 0.6437340974807739
Epoch: 49 Loss G.: 0.9566174149513245
```

In [18]:
```
print(new_representation(loss_generator))
```

```
0x1.032f7e0000000p+0
```

In [19]:
```
with tick.marks(35):
    assert compare_numbers(new_representation(loss_discriminator), "3c3d", '0x1.333

with tick.marks(35):
    assert compare_numbers(new_representation(loss_generator), "3c3d", '0x1.8000000
```

✓ [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.

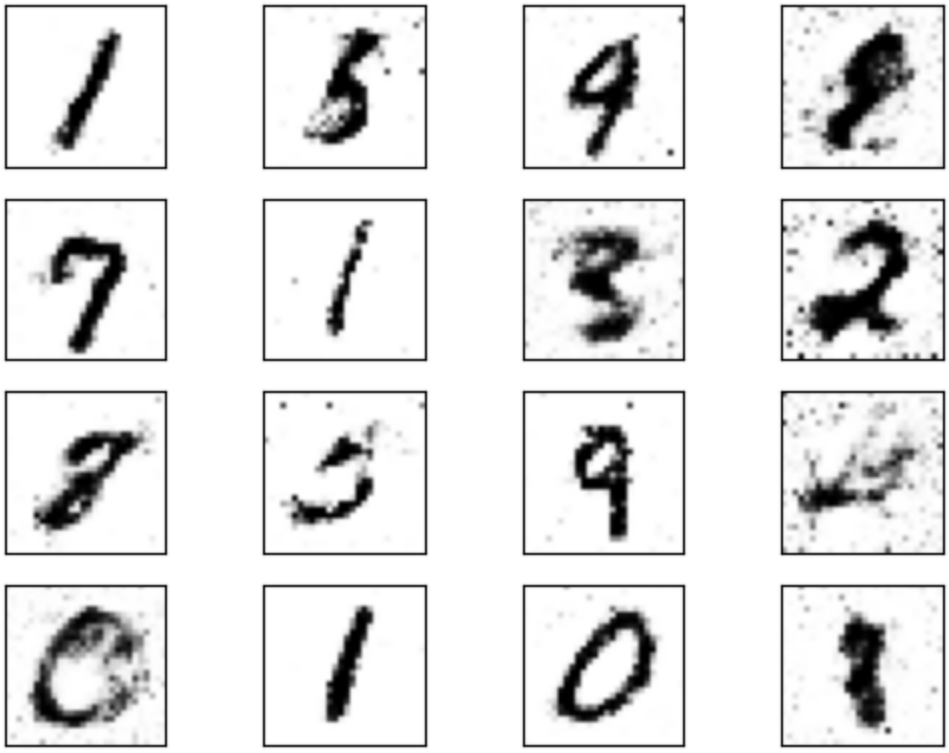
Para trazar generated_samples, es necesario mover los datos de vuelta a la CPU en caso de que estén en la GPU. Para ello, simplemente se puede llamar a .cpu(). Como se hizo anteriormente, también es necesario llamar a .detach() antes de usar Matplotlib para trazar los datos.

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.

In [20]:
```
latent_space_samples = torch.randn(batch_size, 100).to(device=device)
generated_samples = generator(latent_space_samples)
```

In [21]:
```
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([])
```



In [22]:
```
# 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=
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? en la primera parte se usó un MLP pequeño para datos bidimensionales; en esta parte se usa un MLP más grande que genera imágenes de 28x28 a partir de un vector aleatorio. Ambos entrenan con la misma pérdida y optimizador, pero aquí se normalizan las imágenes y hay más capacidad.

- ¿Qué tan bien se han creado las imagenes esperadas? los dígitos son en general reconocibles, pero se ven borrosos y con poca variedad; a veces se repiten estilos o clases, típico de usar solo capas densas sin componentes específicos para imágenes.

- ¿Cómo mejoraría los modelos? migrar a una arquitectura tipo DCGAN con capas convolucionales y normalización por lotes, usar una pérdida más estable como BCE con logits o variantes como WGAN-GP o Hinge, aplicar label smoothing y ajustar hiperparámetros clásicos de GAN. Para evitar la falta de diversidad, aplicar técnicas como ruido en las entradas del discriminador o minibatch discrimination.

- Observe el GIF creado, y describa la evolución que va viendo al pasar de las epocas al inicio se observa ruido sin forma; a mitad del entrenamiento surgen contornos y fragmentos de dígitos; en épocas tardías aparecen dígitos legibles, aunque suaves y algo repetitivos. Se nota una mejora progresiva en estructura, pero limitada en nitidez y diversidad por la arquitectura actual.

In [23]:
```
print()
print("La fraccion de abajo muestra su rendimiento basado en las partes visibles de
tick.summarise_marks() #
```

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

70 / 70 marks (100.0%)