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

In [2]:
```
import numpy as np
import copy
import matplotlib.pyplot as plt
import scipy
from PIL import Image
import os
from collections import defaultdict

#from IPython import display
#from base64 import b64decode


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

from jhwutils.checkarr import array_hash, check_hash, check_scalar, check_string, 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 [ ]:
```
# 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 [3]:
```
carne_1 = "22537"
firma_mecanografiada_1 = "Derek Arreaga"
carne_2 = "22249"
firma_mecanografiada_2 = "Mónica Salvatierra"
```

In [4]:
```
# 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 [5]:
```
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 [6]:
```
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 [7]:
```
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.

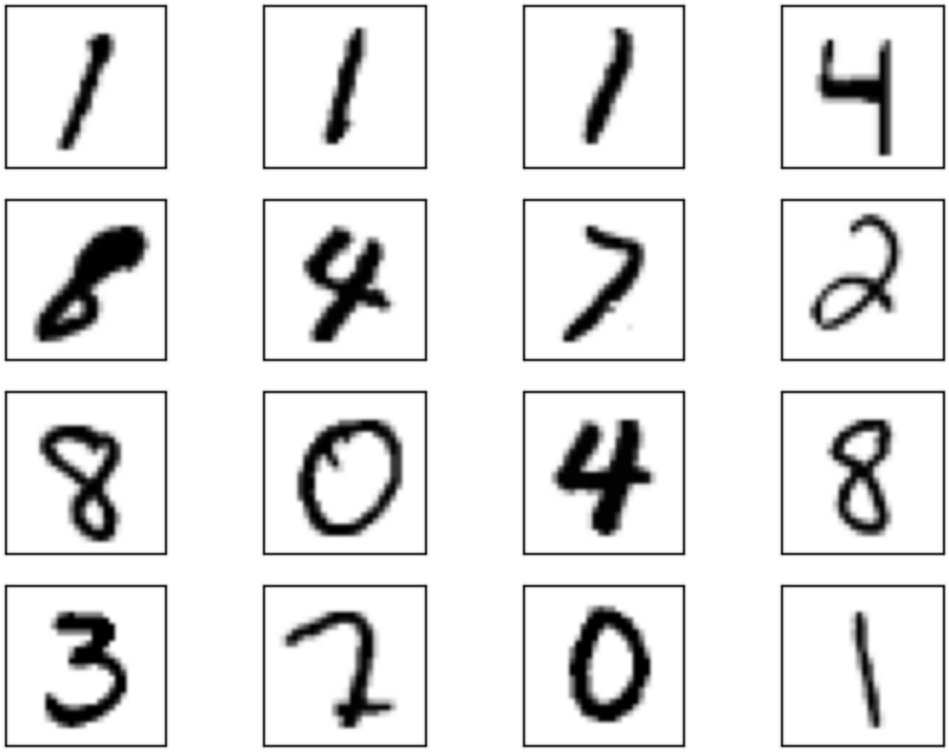
In [8]:
```
transform = transforms.Compose(
    [transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))]
```

```
)
```

In [9]:
```
train_set = torchvision.datasets.MNIST(
    root=".", train=True, download=True, transform=transform
)
```

In [10]:
```
batch_size = 32
train_loader = torch.utils.data.DataLoader(
    train_set, batch_size=batch_size, shuffle=True
)
```

In [11]:
```
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 [12]:
```
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
            nn.Linear(784, 1024),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(1024, 512),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(512, 256),
            nn.ReLU(),
            nn.Dropout(0.3),
            nn.Linear(256, 1),
            nn.Sigmoid()
        )

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

In [13]:
```
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
            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.

In [19]:
```
list_images = []


path_imgs = 'mnistdata/'
```

```
os.makedirs(path_imgs, exist_ok=True)

#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)

        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
        # Aprox 2 lineas para
        # setear el generador en zero_grad
```

```
        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.: 0.5011066794395447
Epoch: 0 Loss G.: 0.6000990867614746
Epoch: 1 Loss D.: 0.03474899008870125
Epoch: 1 Loss G.: 6.868279933929443
Epoch: 2 Loss D.: 0.010708819143474102
Epoch: 2 Loss G.: 5.091273784637451
Epoch: 3 Loss D.: 0.07455518841743469
Epoch: 3 Loss G.: 7.1937689781188965
Epoch: 4 Loss D.: 0.02771400474011898
Epoch: 4 Loss G.: 4.889484405517578
Epoch: 5 Loss D.: 0.043227534741163254
Epoch: 5 Loss G.: 4.765044689178467
Epoch: 6 Loss D.: 0.1514996886253357
Epoch: 6 Loss G.: 2.945019006729126
Epoch: 7 Loss D.: 0.10292315483093262
Epoch: 7 Loss G.: 3.674926280975342
Epoch: 8 Loss D.: 0.17289377748966217
Epoch: 8 Loss G.: 3.0300133228302
Epoch: 9 Loss D.: 0.3646804988384247
Epoch: 9 Loss G.: 2.1674885749816895
Epoch: 10 Loss D.: 0.3072262704372406
Epoch: 10 Loss G.: 1.8481247425079346
Epoch: 11 Loss D.: 0.371298611164093
Epoch: 11 Loss G.: 1.4094512462615967
Epoch: 12 Loss D.: 0.3721049427986145
Epoch: 12 Loss G.: 1.5555992126464844
Epoch: 13 Loss D.: 0.5165724158287048
Epoch: 13 Loss G.: 1.3560930490493774
Epoch: 14 Loss D.: 0.38807016611099243
Epoch: 14 Loss G.: 1.5396807193756104
Epoch: 15 Loss D.: 0.3775888979434967
Epoch: 15 Loss G.: 1.1439380645751953
Epoch: 16 Loss D.: 0.3630494177341461
Epoch: 16 Loss G.: 1.3999695777893066
Epoch: 17 Loss D.: 0.5038143396377563
Epoch: 17 Loss G.: 1.5870482921600342
Epoch: 18 Loss D.: 0.4951270818710327
Epoch: 18 Loss G.: 1.1820967197418213
Epoch: 19 Loss D.: 0.4327355623245239
Epoch: 19 Loss G.: 1.3127540349960327
Epoch: 20 Loss D.: 0.5607686042785645
Epoch: 20 Loss G.: 1.178614616394043
Epoch: 21 Loss D.: 0.41448256373405457
Epoch: 21 Loss G.: 1.2603973150253296
Epoch: 22 Loss D.: 0.5533496737480164
Epoch: 22 Loss G.: 1.2645509243011475
Epoch: 23 Loss D.: 0.44523948431015015
Epoch: 23 Loss G.: 1.142289400100708
Epoch: 24 Loss D.: 0.4895051121711731
Epoch: 24 Loss G.: 1.19744873046875
Epoch: 25 Loss D.: 0.5057028532028198
Epoch: 25 Loss G.: 1.234075903892517
Epoch: 26 Loss D.: 0.57773357629776
Epoch: 26 Loss G.: 1.1435458660125732
Epoch: 27 Loss D.: 0.5524777770042419
Epoch: 27 Loss G.: 1.4228715896606445
```

```
Epoch: 28 Loss D.: 0.6085057854652405
Epoch: 28 Loss G.: 1.0301165580749512
Epoch: 29 Loss D.: 0.527237057685852
Epoch: 29 Loss G.: 1.0369949340820312
Epoch: 30 Loss D.: 0.5522568821907043
Epoch: 30 Loss G.: 1.0356512069702148
Epoch: 31 Loss D.: 0.5054879784584045
Epoch: 31 Loss G.: 1.1488969326019287
Epoch: 32 Loss D.: 0.5519049167633057
Epoch: 32 Loss G.: 1.0906935930252075
Epoch: 33 Loss D.: 0.5127324461936951
Epoch: 33 Loss G.: 0.9982081651687622
Epoch: 34 Loss D.: 0.6495967507362366
Epoch: 34 Loss G.: 1.060173749923706
Epoch: 35 Loss D.: 0.5519865155220032
Epoch: 35 Loss G.: 1.100609302520752
Epoch: 36 Loss D.: 0.6513745188713074
Epoch: 36 Loss G.: 0.9884518384933472
Epoch: 37 Loss D.: 0.5234681367874146
Epoch: 37 Loss G.: 1.1271170377731323
Epoch: 38 Loss D.: 0.46481087803840637
Epoch: 38 Loss G.: 1.1310518980026245
Epoch: 39 Loss D.: 0.5877102613449097
Epoch: 39 Loss G.: 0.9647234678268433
Epoch: 40 Loss D.: 0.558350145816803
Epoch: 40 Loss G.: 1.0401034355163574
Epoch: 41 Loss D.: 0.5052496790885925
Epoch: 41 Loss G.: 1.1129372119903564
Epoch: 42 Loss D.: 0.6328728199005127
Epoch: 42 Loss G.: 1.062151312828064
Epoch: 43 Loss D.: 0.5644454956054688
Epoch: 43 Loss G.: 0.9613176584243774
Epoch: 44 Loss D.: 0.5519870519638062
Epoch: 44 Loss G.: 1.0500354766845703
Epoch: 45 Loss D.: 0.6094589829444885
Epoch: 45 Loss G.: 1.2121914625167847
Epoch: 46 Loss D.: 0.6016254425048828
Epoch: 46 Loss G.: 1.0429612398147583
Epoch: 47 Loss D.: 0.6139931678771973
Epoch: 47 Loss G.: 0.9204051494598389
Epoch: 48 Loss D.: 0.604719877243042
Epoch: 48 Loss G.: 1.0359512567520142
Epoch: 49 Loss D.: 0.5589917898178101
Epoch: 49 Loss G.: 1.0846112966537476
```

In [20]:
```
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.

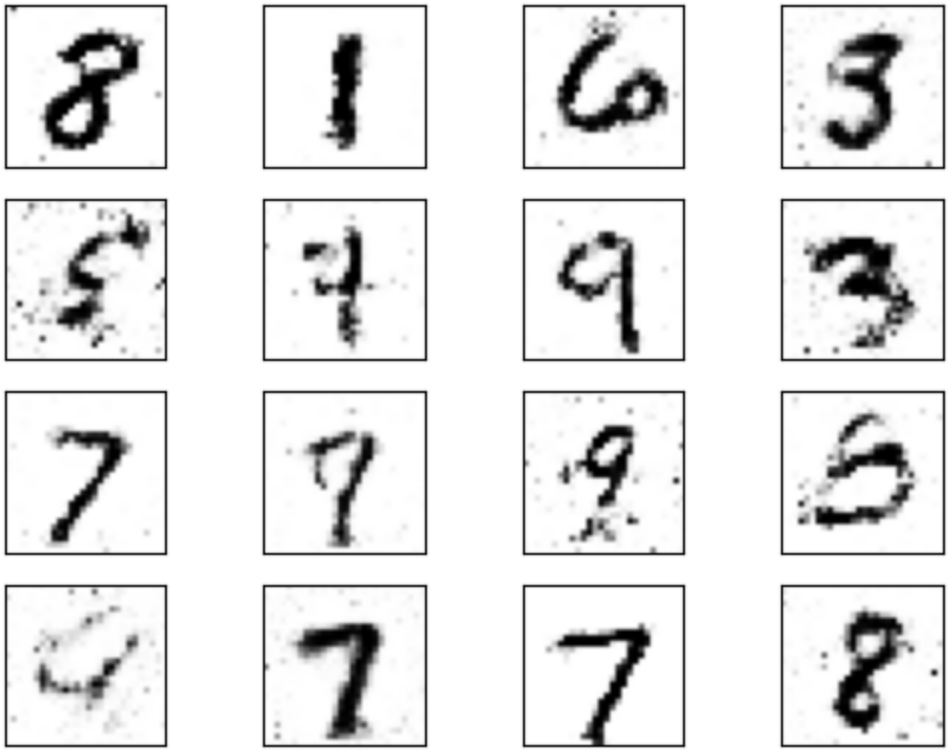
Para trazar generated_samples, es necesario mover los datos de vuelta a la CPU en caso de que estén en la GPU. Para ello, simplemente se puede llamar a .cpu(). Como se hizo anteriormente, también es necesario llamar a .detach() antes de usar Matplotlib para trazar los datos.

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 [21]:
```
latent_space_samples = torch.randn(batch_size, 100).to(device=device)
generated_samples = generator(latent_space_samples)
```

In [22]:
```
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 [23]:
```
# 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:

1. **¿Qué diferencias hay entre los modelos usados en la primera parte y los usados en esta parte?**

- En la primera parte trabajamos con un problema mucho más simple, puesto que generaba puntos que siguieran una curva senoidal. Como los datos eran solo pares de números, bastaba con usar redes pequeñas y capas densas.En la segunda parte ya se trata de generar imágenes de 28×28 píxeles, lo cual es mucho más complejo porque el modelo tiene que aprender trazos, formas y estilos de los dígitos.

2. **¿Qué tan bien se han creado las imagenes esperadas?**

- Después de 50 épocas ya se ven dígitos que se pueden reconocer como 7, 0, 8, 5 o 1. Sin embargo, todavía tienen ruido, los bordes no son del todo claros y algunos números están incompletos o deformados. O sea, el modelo aprendió la idea general de los dígitos, pero aún no logra hacerlos con la nitidez y calidad de los reales.

3. **¿Cómo mejoraría los modelos?**

- Se mejoraría el generador y el discriminador usando redes convolucionales en lugar de solo capas densas. También, consideraríaos normalizar bien los datos, entrenar más épocas y aplicar técnicas que ayudan a estabilizar los GAN, como suavizar etiquetas, por ejemplo.

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

- Al inicio solo aparecen manchas y ruido sin forma. Después de unas cuantas épocas empiezan a verse trazos que recuerdan a números, aunque todavía borrosos. Conforme avanza el entrenamiento, los dígitos se van haciendo cada vez más claros y reconocibles, aunque con algo de ruido en los bordes. Ya hacia las últimas épocas se distinguen bien varios números, aunque todavía no alcanzan la calidad de las imágenes reales.

In [24]:
```
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%)