Laboratorio 6 Parte 1

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 --upgrade pip setuptools wheel
# %pip install -U --force-reinstall --no-cache https://github.com/johnhw/jhwutil
# %pip install scikit-image
# %pip install -U --force-reinstall --no-cache https://github.com/AlbertS789/lau
```

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

```
###
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 [ ]:
```
carne_1 = "21764"
firma_mecanografiada_1 = "Sebastián Silva"
carne_2 = "21781"
firma_mecanografiada_2 = "Erick Guerra"
# YOUR CODE HERE
# raise NotImplementedError()
```

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

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 [ ]:
```
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 [ ]:
```
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 [ ]:
```
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:

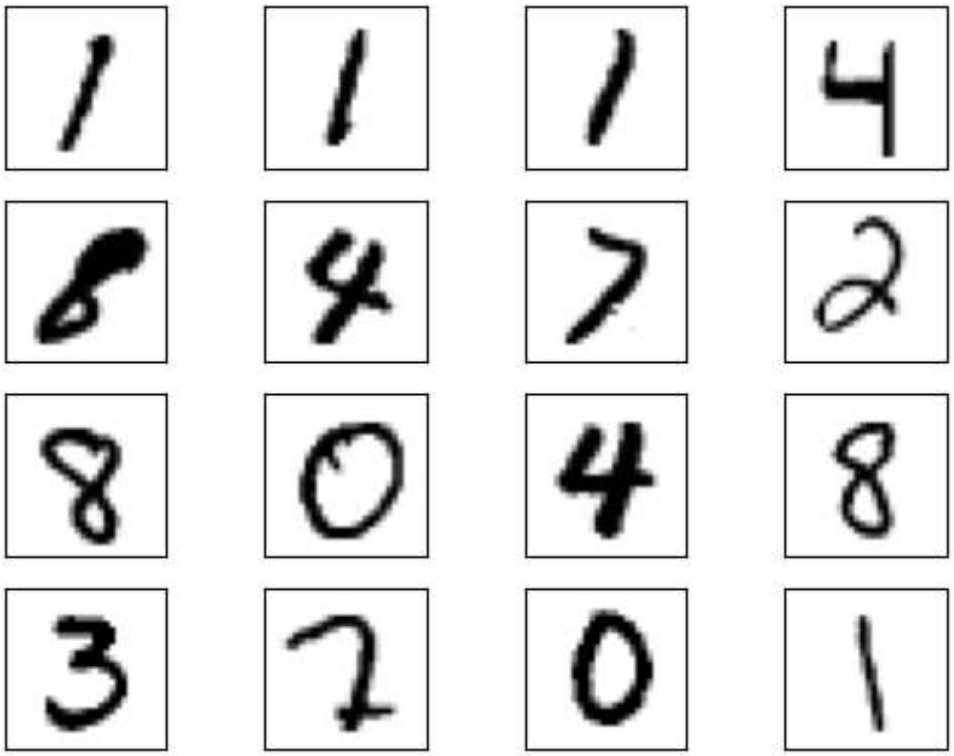
Como se puede ver más adelante, hay dígitos con diferentes estilos de escritura. A medida que la GAN aprende la distribución de los datos, también generará dígitos con diferentes estilos de escritura.

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

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

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

In [ ]:
```
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 [ ]:
```
class Discriminator(nn.Module):
    def __init__(self):
        super().__init__()
        self.model = nn.Sequential(
            # Aprox 11 lineas
            # lineal de la entrada dicha y salida 1024
            nn.Linear(784, 1024),
            # ReLU
            nn.ReLU(),
            # Dropout de 30%
            nn.Dropout(0.3),
            # Lineal de la entrada correspondiente y salida 512
            nn.Linear(1024, 512),
            # ReLU
            nn.ReLU(),
            # Dropout de 30%
            nn.Dropout(0.3),
            # Lienal de la entrada correspondiente y salida 256
            nn.Linear(512, 256),
            # ReLU
            nn.ReLU(),
            # Dropout de 30%
            nn.Dropout(0.3),
            # Lineal de la entrada correspondiente y salida 1
            nn.Linear(256, 1),
            # Sigmoide
            nn.Sigmoid()
            # YOUR CODE HERE
            # raise NotImplementedError()
        )

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

In [ ]:
```
class Generator(nn.Module):
    def __init__(self):
        super().__init__()
        self.model = nn.Sequential(
            # Aprox 8 lienas para
            # Lineal input = 100, output = 256
            nn.Linear(100, 256),
            # ReLU
            nn.ReLU(),
            # Lineal output = 512
            nn.Linear(256, 512),
            # ReLU
            nn.ReLU(),
            # Lineal output = 1024
            nn.Linear(512, 1024),
            # ReLU
            nn.ReLU(),
```

```
            # Lineal output = 784
            nn.Linear(1024, 784),
            # Tanh
            nn.Tanh()
            # YOUR CODE HERE
            # raise NotImplementedError()
        )

    def forward(self, x):
        output = self.model(x)
        output = output.view(x.size(0), 1, 28, 28)
        return output
```

Entrenando los Modelos

Para entrenar los modelos, es necesario definir los parámetros de entrenamiento y los optimizadores como se hizo en la parte anterior.

Para obtener un mejor resultado, se disminuye la tasa de aprendizaje de la primera parte. También se establece el número de épocas en 10 para reducir el tiempo de entrenamiento.

El ciclo de entrenamiento es muy similar al que se usó en la parte previa. Note como se envían los datos de entrenamiento a device para usar la GPU si está disponible

Algunos de los tensores no necesitan ser enviados explícitamente a la GPU con device. Este es el caso de generated_samples, que ya se envió a una GPU disponible, ya que latent_space_samples y generator se enviaron a la GPU previamente.

Dado que esta parte presenta modelos más complejos, el entrenamiento puede llevar un poco más de tiempo. Después de que termine, se pueden verificar los resultados generando algunas muestras de dígitos escritos a mano.

In [ ]:
```
import os
import glob

list_images = []

# Aprox 1 linea para que decidan donde guardar un set de imagen que vamos a gene
# path_imgs =
files = glob.glob('imgs/*.jpg')
for f in files:
    os.remove(f)
path_imgs = "imgs/"
# YOUR CODE HERE
# raise NotImplementedError()

#seed_all(seed_)

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

lr = 0.0001
num_epochs = 50
loss_function = nn.BCELoss()
```

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

        # Data for training the generator
        latent_space_samples = torch.randn((batch_size, 100)).to(
            device=device
        )
        generated_samples = generator(latent_space_samples)

        # Training the generator
        # Training the generator
        # Aprox 2 lineas para
        # setear el generador en zero_grad
        generator.zero_grad()
        # output_discriminator =
        # YOUR CODE HERE
        # raise NotImplementedError()
        output_discriminator_generated = discriminator(generated_samples)
        loss_generator = loss_function(
            output_discriminator_generated, real_samples_labels
        )
```

```
        # Aprox dos lineas para
        # llamar al paso backward sobre el loss_generator
        loss_generator.backward(retain_graph=True)
        # llamar al optimizador sobre optimizer_generator
        optimizer_discriminator.step()
        # YOUR CODE HERE
        # raise NotImplementedError()

        # Guardamos las imagenes
        if epoch % 2 == 0 and n ==  batch_size - 1:
            generated_samples_detached = generated_samples.cpu().detach()
            for i in range(16):
                ax = plt.subplot(4, 4, i + 1)
                plt.imshow(generated_samples_detached[i].reshape(28, 28), cmap="
                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.: 0.3657624423503876
Epoch: 0 Loss G.: 0.6562583446502686
Epoch: 1 Loss D.: 0.3461167812347412
Epoch: 1 Loss G.: 0.6963969469070435
Epoch: 2 Loss D.: 0.346778929233551
Epoch: 2 Loss G.: 0.6936269998550415
Epoch: 3 Loss D.: 0.34515371918678284
Epoch: 3 Loss G.: 0.6949262619018555
Epoch: 4 Loss D.: 0.3480178117752075
Epoch: 4 Loss G.: 0.690433144569397
Epoch: 5 Loss D.: 0.3462327718734741
Epoch: 5 Loss G.: 0.6934514045715332
Epoch: 6 Loss D.: 0.3464050590991974
Epoch: 6 Loss G.: 0.6940963268280029
Epoch: 7 Loss D.: 0.3467215895652771
Epoch: 7 Loss G.: 0.692907989025116
Epoch: 8 Loss D.: 0.34621661901474
Epoch: 8 Loss G.: 0.6931207776069641
Epoch: 9 Loss D.: 0.3468884825706482
Epoch: 9 Loss G.: 0.69339919090271
Epoch: 10 Loss D.: 0.3462490439414978
Epoch: 10 Loss G.: 0.6937119960784912
Epoch: 11 Loss D.: 0.345819890499115
Epoch: 11 Loss G.: 0.6947296857833862
Epoch: 12 Loss D.: 0.3462247848510742
Epoch: 12 Loss G.: 0.6940178275108337
Epoch: 13 Loss D.: 0.34694361686706543
Epoch: 13 Loss G.: 0.6933536529541016
Epoch: 14 Loss D.: 0.3467235565185547
Epoch: 14 Loss G.: 0.692925214767456
Epoch: 15 Loss D.: 0.34699881076812744
Epoch: 15 Loss G.: 0.6932973861694336
Epoch: 16 Loss D.: 0.3469310402870178
Epoch: 16 Loss G.: 0.6929868459701538
Epoch: 17 Loss D.: 0.3466535210609436
Epoch: 17 Loss G.: 0.6930985450744629
Epoch: 18 Loss D.: 0.346589595079422
Epoch: 18 Loss G.: 0.693063497543335
Epoch: 19 Loss D.: 0.34706002473831177
Epoch: 19 Loss G.: 0.6914981603622437
Epoch: 20 Loss D.: 0.3470665216445923
Epoch: 20 Loss G.: 0.6925935745239258
Epoch: 21 Loss D.: 0.34699901938438416
Epoch: 21 Loss G.: 0.6929854154586792
Epoch: 22 Loss D.: 0.3468185067176819
Epoch: 22 Loss G.: 0.6925135850906372
Epoch: 23 Loss D.: 0.3469780683517456
Epoch: 23 Loss G.: 0.6923430562019348
Epoch: 24 Loss D.: 0.346052885055542
Epoch: 24 Loss G.: 0.6942394375801086
Epoch: 25 Loss D.: 0.34681612253189087
Epoch: 25 Loss G.: 0.6928931474685669
Epoch: 26 Loss D.: 0.34589022397994995
Epoch: 26 Loss G.: 0.6944361925125122
Epoch: 27 Loss D.: 0.34673723578453064
Epoch: 27 Loss G.: 0.69324791431427
Epoch: 28 Loss D.: 0.34646084904670715
Epoch: 28 Loss G.: 0.6935505867004395
Epoch: 29 Loss D.: 0.3467782139778137
Epoch: 29 Loss G.: 0.6925615072250366
```

```
Epoch: 30 Loss D.: 0.34664827585220337
Epoch: 30 Loss G.: 0.6918661594390869
Epoch: 31 Loss D.: 0.3465401530265808
Epoch: 31 Loss G.: 0.6928743720054626
Epoch: 32 Loss D.: 0.34688401222229004
Epoch: 32 Loss G.: 0.6924732327461243
Epoch: 33 Loss D.: 0.34641122817993164
Epoch: 33 Loss G.: 0.6929231882095337
Epoch: 34 Loss D.: 0.3462632894515991
Epoch: 34 Loss G.: 0.6940061450004578
Epoch: 35 Loss D.: 0.3464047908782959
Epoch: 35 Loss G.: 0.6932170391082764
Epoch: 36 Loss D.: 0.3467024862766266
Epoch: 36 Loss G.: 0.6932193636894226
Epoch: 37 Loss D.: 0.34609392285346985
Epoch: 37 Loss G.: 0.6942047476768494
Epoch: 38 Loss D.: 0.3466096520423889
Epoch: 38 Loss G.: 0.6929594278335571
Epoch: 39 Loss D.: 0.34643274545669556
Epoch: 39 Loss G.: 0.6931163668632507
Epoch: 40 Loss D.: 0.3468930721282959
Epoch: 40 Loss G.: 0.6927411556243896
Epoch: 41 Loss D.: 0.3467029333114624
Epoch: 41 Loss G.: 0.6924269199371338
Epoch: 42 Loss D.: 0.3465946912765503
Epoch: 42 Loss G.: 0.692749559879303
Epoch: 43 Loss D.: 0.3465031385421753
Epoch: 43 Loss G.: 0.6927957534790039
Epoch: 44 Loss D.: 0.34695929288864136
Epoch: 44 Loss G.: 0.6926320791244507
Epoch: 45 Loss D.: 0.34672674536705017
Epoch: 45 Loss G.: 0.6926779747009277
Epoch: 46 Loss D.: 0.3464621603488922
Epoch: 46 Loss G.: 0.6933028697967529
Epoch: 47 Loss D.: 0.3464423418045044
Epoch: 47 Loss G.: 0.6932103037834167
Epoch: 48 Loss D.: 0.3469226658344269
Epoch: 48 Loss G.: 0.6928709745407104
Epoch: 49 Loss D.: 0.34678515791893005
Epoch: 49 Loss G.: 0.6931144595146179
```

In [ ]:

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

with tick.marks(35):
    assert compare_numbers(new_representation(loss_generator), "3c3d", '0x1.8000
```

✓ [35 marks]

✓ [35 marks]

In [ ]:

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call dri
ve.mount("/content/drive", force_remount=True).
```

Validación del Resultado

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

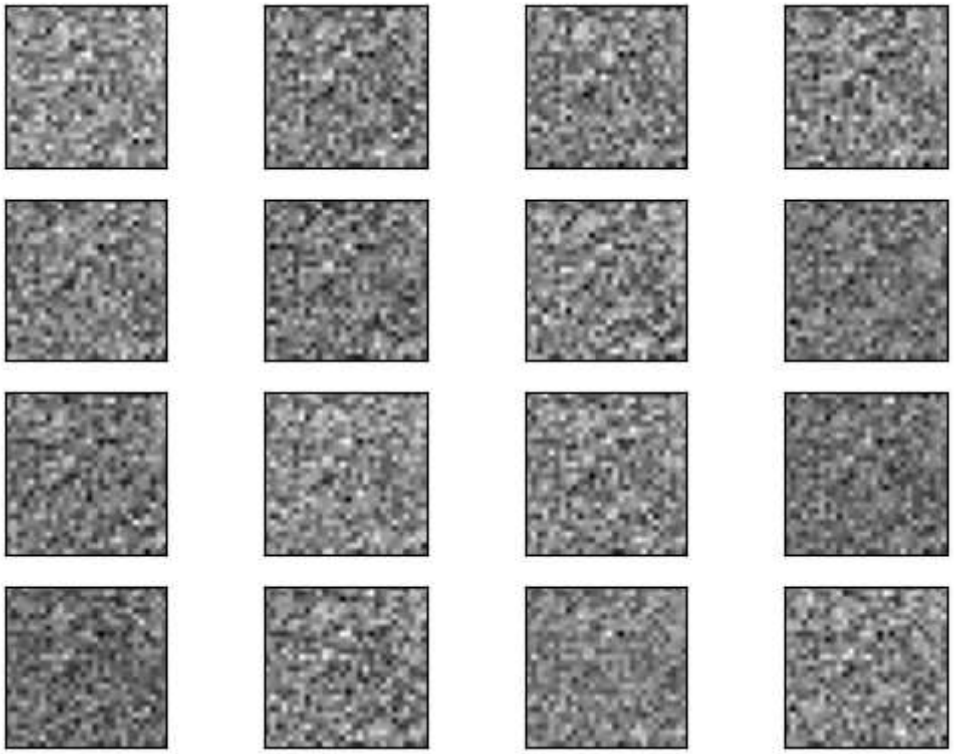
Para trazar generated_samples, es necesario mover los datos de vuelta a la CPU en caso de que estén en la GPU. Para ello, simplemente se puede llamar a .cpu(). Como se hizo anteriormente, también es necesario llamar a .detach() antes de usar Matplotlib para trazar los datos.

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

In [ ]:
```
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 [ ]:
```
# Visualización del progreso de entrenamiento
# Para que esto se ve bien, por favor reinicien el kernel y corran todo el noteb

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, durati
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 del laboratorio, se utilizan modelos discriminativos y generativos, mientras que la segunda parte se enfoca en la implementación de redes generativas adversarias (GANs) más complejas. Los modelos en la primera parte son más generales y teóricos, mientras que en la segunda parte se aplican específicamente a la generación de imágenes de dígitos, con un enfoque en mejorar la calidad de las imágenes generadas a través de entrenamiento adversarial.
- **¿Qué tan bien se han creado las imagenes esperadas?**
  - Al final del entrenamiento, como se ve en el GIF generado al final, realmente no se ha logrado generar dígitos distinguibles, sino que solamente se ve que el ruido se dispersa de forma distinta en cada iteración. Esto podría deberse a que

las épocas no fueron suficientes (que podría no ser la respuesta más apropiada puesto que no se ve mejora alguna a lo largo de cada época) o que sencillamente el discriminante no está logrando identificar de forma correcta que las imágenes no están satisfaciendo las caracterísitcas esperadas.

- **¿Cómo mejoraría los modelos?**
  - Se podría aumentar el número de épocas de entrenamiento para permitir que el generador aprenda mejor la distribución de los datos reales. Además, sería beneficioso experimentar ajustando ciertos hiperparámetros como la tasa de aprendizaje y el optimizador. Implementar técnicas de regularización, como dropout o batch normalization, podría ayudar a prevenir el sobreajuste y mejorar la estabilidad del entrenamiento.
- **Observe el GIF creado, y describa la evolución que va viendo al pasar de las epocas**
  - Como se mencionó anteriormente, la evolución entre cada época parece no existente o es no percibible a simple vista dentro de las imágenes. Podría ser que las épocas no fueron suficientes, o bien, que el modelo utilizado no es el apropiado o existe algún error en el entrenamiento aplicado.

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

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

70 / 70 marks (100.0%)