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 [ ]:
```
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 [ ]:
```
# 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 [ ]:
```
# 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 = "21700"
firma_mecanografiada_1 = "Angel Castellanos"
carne_2 = "21146"
firma_mecanografiada_2 = "Diego Morales"
```

In [ ]:
```
# 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 [ ]:
```
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.

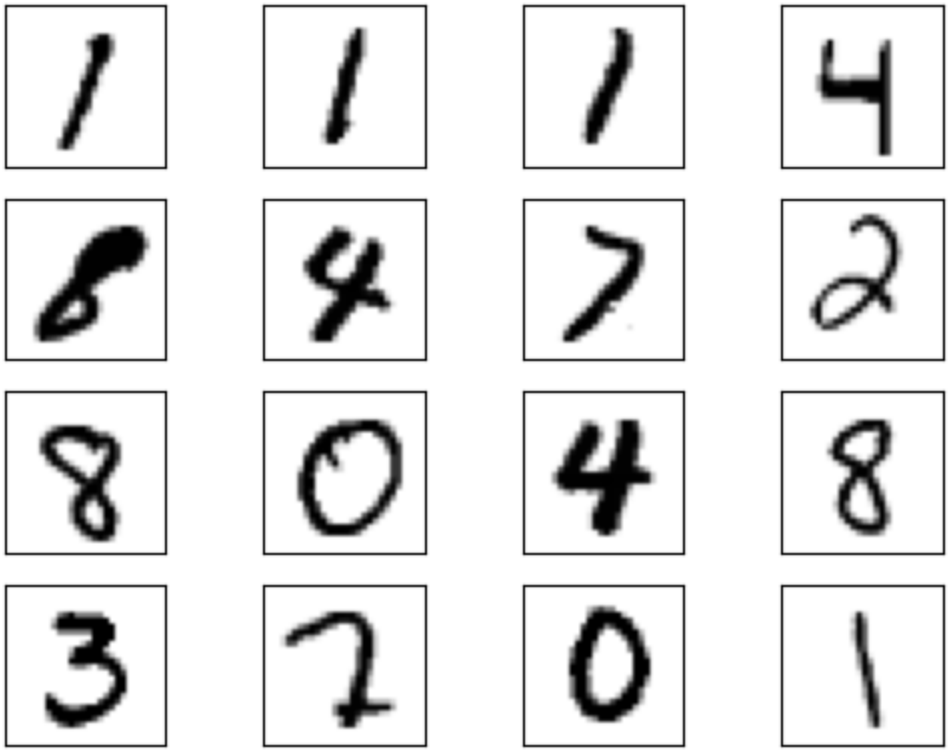
In [ ]:
```
transform = transforms.Compose(
    [transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))]
```

```
)
```

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
            # 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
            #raise NotImplementedError()
            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 [ ]:
```
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
            # #raise NotImplementedError()
            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 [ ]:
```
list_images = []

# Aprox 1 linea para que decidan donde guardar un set de imagen que vamos a generar
# path_imgs =
# YOUR CODE HERE
#raise NotImplementedError()
path_imgs = "D:/UVG Tareas/8vo Semestre/Deep learning/LAB-06-DL/images/"

#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
        # output_discriminator =
        # YOUR CODE HERE
        discriminator.zero_grad()
        output_discriminator = discriminator(all_samples)
        #raise NotImplementedError()
        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
        #raise NotImplementedError()
```

```
        loss_discriminator.backward()
        optimizer_discriminator.step()

        # 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
        # output_discriminator =
        # YOUR CODE HERE
        #raise NotImplementedError()
        generator.zero_grad()
        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
        #raise NotImplementedError()
        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.: 0.5889772176742554
Epoch: 0 Loss G.: 0.4736277461051941
Epoch: 1 Loss D.: 0.01701052114367485
Epoch: 1 Loss G.: 5.873589038848877
Epoch: 2 Loss D.: 0.02129947394132614
Epoch: 2 Loss G.: 4.940909385681152
Epoch: 3 Loss D.: 0.19378040730953217
Epoch: 3 Loss G.: 4.2737932205200195
Epoch: 4 Loss D.: 0.08771371096372604
Epoch: 4 Loss G.: 4.430030345916748
Epoch: 5 Loss D.: 0.05200854316353798
Epoch: 5 Loss G.: 4.885374546051025
Epoch: 6 Loss D.: 0.049161817878484726
Epoch: 6 Loss G.: 5.422560691833496
Epoch: 7 Loss D.: 0.006538892164826393
Epoch: 7 Loss G.: 11.574234008789062
Epoch: 8 Loss D.: 0.0005126949981786311
Epoch: 8 Loss G.: 25.479747772216797
Epoch: 9 Loss D.: 1.1175904290894323e-07
Epoch: 9 Loss G.: 55.17871856689453
Epoch: 10 Loss D.: 3.71949413855135e-18
Epoch: 10 Loss G.: 55.92223358154297
Epoch: 11 Loss D.: 4.029400782219056e-14
Epoch: 11 Loss G.: 48.148189544677734
Epoch: 12 Loss D.: 1.591247307051342e-31
Epoch: 12 Loss G.: 97.53176879882812
Epoch: 13 Loss D.: 6.219328551019835e-31
Epoch: 13 Loss G.: 97.98086547851562
Epoch: 14 Loss D.: 3.871443808330163e-34
Epoch: 14 Loss G.: 98.522705078125
Epoch: 15 Loss D.: 3.931378647283304e-28
Epoch: 15 Loss G.: 99.57902526855469
Epoch: 16 Loss D.: 1.9084067200949038e-36
Epoch: 16 Loss G.: 96.1341781616211
Epoch: 17 Loss D.: 2.0937925598143945e-38
Epoch: 17 Loss G.: 99.42227172851562
Epoch: 18 Loss D.: 9.91645113341643e-38
Epoch: 18 Loss G.: 97.70267486572266
Epoch: 19 Loss D.: 8.886579449957472e-35
Epoch: 19 Loss G.: 98.80015563964844
Epoch: 20 Loss D.: 1.7287763257409095e-32
Epoch: 20 Loss G.: 98.80509948730469
Epoch: 21 Loss D.: 3.4048646390078045e-38
Epoch: 21 Loss G.: 98.06031799316406
Epoch: 22 Loss D.: 1.1640081875699098e-39
Epoch: 22 Loss G.: 97.23662567138672
Epoch: 23 Loss D.: 8.008210954841414e-36
Epoch: 23 Loss G.: 97.63447570800781
Epoch: 24 Loss D.: 1.004839599551879e-38
Epoch: 24 Loss G.: 98.99722290039062
Epoch: 25 Loss D.: 2.5434173149532796e-30
Epoch: 25 Loss G.: 95.30679321289062
Epoch: 26 Loss D.: 1.7037428217216454e-31
Epoch: 26 Loss G.: 93.82962036132812
Epoch: 27 Loss D.: 3.144655686048585e-31
Epoch: 27 Loss G.: 91.17491149902344
```

```
Epoch: 28 Loss D.: 3.1962847515387693e-06
Epoch: 28 Loss G.: 28.886327743530273
Epoch: 29 Loss D.: 3.430985240899147e-24
Epoch: 29 Loss G.: 73.29696655273438
Epoch: 30 Loss D.: 0.0024521811865270138
Epoch: 30 Loss G.: 20.580419540405273
Epoch: 31 Loss D.: 2.3056166351120844e-13
Epoch: 31 Loss G.: 46.33039093017578
Epoch: 32 Loss D.: 1.0613338680132145e-10
Epoch: 32 Loss G.: 47.423831939697266
Epoch: 33 Loss D.: 7.650855250061689e-25
Epoch: 33 Loss G.: 78.16291046142578
Epoch: 34 Loss D.: 3.0598775325960943e-28
Epoch: 34 Loss G.: 79.37368774414062
Epoch: 35 Loss D.: 2.7244884153237427e-25
Epoch: 35 Loss G.: 81.31855010986328
Epoch: 36 Loss D.: 1.261750477388668e-24
Epoch: 36 Loss G.: 70.79839324951172
Epoch: 37 Loss D.: 1.5209400821116456e-21
Epoch: 37 Loss G.: 69.91432189941406
Epoch: 38 Loss D.: 4.868925237670986e-24
Epoch: 38 Loss G.: 67.02011108398438
Epoch: 39 Loss D.: 3.3951168202224923e-26
Epoch: 39 Loss G.: 65.86579895019531
Epoch: 40 Loss D.: 5.934965653007896e-30
Epoch: 40 Loss G.: 95.27542114257812
Epoch: 41 Loss D.: 3.107364284455189e-29
Epoch: 41 Loss G.: 95.13655090332031
Epoch: 42 Loss D.: 6.826115678628988e-32
Epoch: 42 Loss G.: 92.83621215820312
Epoch: 43 Loss D.: 9.706940901723085e-29
Epoch: 43 Loss G.: 93.0690689086914
Epoch: 44 Loss D.: 1.97603854893906e-28
Epoch: 44 Loss G.: 91.92302703857422
Epoch: 45 Loss D.: 9.454437089687584e-27
Epoch: 45 Loss G.: 88.254638671875
Epoch: 46 Loss D.: 5.3857758207783e-29
Epoch: 46 Loss G.: 90.17214965820312
Epoch: 47 Loss D.: 4.814568180756328e-34
Epoch: 47 Loss G.: 90.63739013671875
Epoch: 48 Loss D.: 1.6734737981069334e-33
Epoch: 48 Loss G.: 92.93150329589844
Epoch: 49 Loss D.: 1.016966124725478e-31
Epoch: 49 Loss G.: 94.01290893554688
```

In [ ]:
```
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.

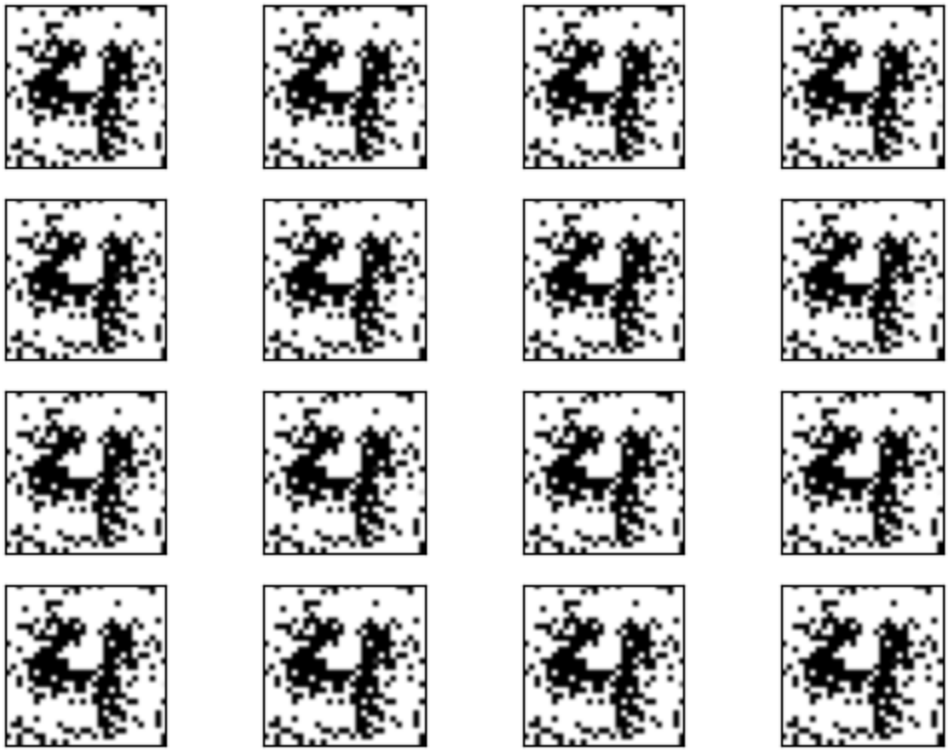
Para trazar generated_samples, es necesario mover los datos de vuelta a la CPU en caso de que estén en la GPU. Para ello, simplemente se puede llamar a .cpu(). Como se hizo anteriormente, también es necesario llamar a .detach() antes de usar Matplotlib para trazar los datos.

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 [ ]:
```
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 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?

Primera Parte: El generador y el discriminador pueden ser redes neuronales simples como Multi-Layer Perceptrons (MLP). Segunda Parte: El discriminador es una red neuronal MLP que recibe una imagen de 28 × 28 píxeles y proporciona la probabilidad de que la imagen pertenezca a los datos reales de entrenamiento. El generador recibe una entrada de 100 dimensiones y proporciona una salida con 784 coeficientes, que se organizan en un tensor de 28 × 28 que representa una imagen.

- ¿Qué tan bien se han creado las imagenes esperadas?

Aunque las imágenes evolucionan y muestran patrones más claros a medida que avanza el entrenamiento, puede que aún no alcancen la calidad o definición deseada. Las imágenes muestran cierta estructura, pero pueden seguir siendo un poco borrosas o indistintas en comparación con el objetivo esperado.

- ¿Cómo mejoraría los modelos?

- Ajuste de hiperparámetros: Experimentar con la tasa de aprendizaje, el tamaño del lote (batch size), y otros parámetros de entrenamiento podría mejorar la calidad de las imágenes generadas.
- Arquitectura del modelo: Considerar el uso de capas adicionales, diferentes funciones de activación o arquitecturas avanzadas como DCGAN (Deep Convolutional GAN) podría ayudar a generar imágenes más detalladas.
- Aumento de datos: Si es posible, aumentar la cantidad de datos de entrenamiento o aplicar técnicas de data augmentation podría proporcionar al generador una mayor diversidad de ejemplos, lo que podría mejorar la calidad de las imágenes producidas.
- Más épocas de entrenamiento: Entrenar el modelo durante más épocas podría ayudar al generador a producir imágenes más realistas, siempre y cuando el modelo no se sobreajuste.

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

A medida que las épocas avanzan, las imágenes empiezan a mostrar patrones más definidos, y el ruido aleatorio comienza a transformarse en estructuras que se asemejan más a las características esperadas del conjunto de datos de entrenamiento. Los cambios son graduales y consisten en una mayor claridad y definición de las imágenes.

In [ ]:
```
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%)