Proyecto 3 *GAN*

Grupo 6

In [1]:
```
import random

# Alphabetic order
authors = [
    "Lluís Bernat Ladaria",
    "David Carretero García",
    "Ramón Rotaeche Fernández de la Riva"]

random.shuffle(authors)

print("Autores:")
print("========")
for a in authors:
  print(f"- %s" % a, sep="\n")
```

```
Autores:
========
- Lluís Bernat Ladaria
- Ramón Rotaeche Fernández de la Riva
- David Carretero García
```

Resumen

Presentamos nuestra propuesta de red *GAN* para la generación de dígitos manuscritos, basada en la librería *Pytorch*. Para elaborarla hemos considerado un subconjunto de elementos de esta librería.

Elementos de la solución

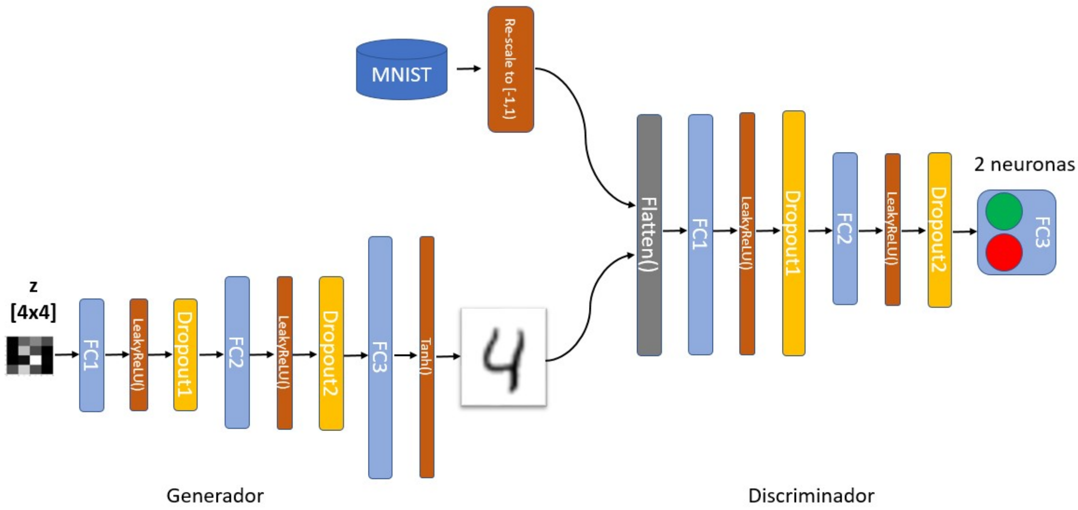
Nuestra propuesta de red *GAN* se basa en los siguientes elementos:

- Capas *Full Connected* (no usamos las convolucionales): pensamos que la dimensión de las imágenes es suficientemente pequeña como para poder tratarlas de forma eficiente con capas *FC*
- Capas *Dropout*: para evitar que la red memorize en lugar de aprender características usamos capas de olvido entre cada una de las *FC*
- Función de activación LeakyReLU(): esta función al ser no nula en todo su dominio excepto en x = 0, permite que la *backpropagation* sea *end-to-end*
- Función de activación Tanh(): la función de tangente hiperbólica funciona mejor que la sigmoide en el tratamiento de imágenes.

Hemos combinado estos elementos en una arquitectura que describimos sucintamente acontinuación.

Arquitectura de la propuesta

La arquitectura de capas es la de la ilustración siguiente. Cabe destacar que para clasificar si las imágenes son verdaderas o falsas hemos considerado una clase para cada opción, es decir a la salida del discriminador obtenemos dos valores (hay dos neuronas), uno para indicar cuan de verdadera parece y el otro para indicar cuan de falsa parece. Pensamos que usar este planteamiento en lugar de uno con una sola neurona, va a promover un funcionamiento más eficiente en la red clasificadora del discriminador.



También debemos señalar que a la salida del discriminador no se aplica una función de activación al uso, sinó que a fin de conseguir una mayor estabilidad numérica la función de activación (que sería una sigmoide) y la de pérdida (que sería el cálculo de la entropía cruzada binaria) se combinan en una sola, llamada *BCEWithLogitsLoss* que optimiza el cálculo al aplicar las funciones logarítmicas del cálculo de la entropía a la sigmoide (que contiene una exponencial).

Resultados

Después de múltiples pruebas combinando:

- la dimensión de z a 25, 16 y 9
- valores de Dropout de 0.01, 0.1, 0.2, 0.3, 0.4 y 0.5
- número de epochs entre 40 y 100

Hemos concluido que los resultados de mayor *credibilidad* que hemos podido conseguir con el generador son los que se consiguen con los parámetros e hiperparámetros que se presentan en los comentarios y el código de esta práctica.

Aquí puede consultar los parámetros de las capas

Índice de comentarios

Además de este resumen, hay un total de **8 comentarios** señalizados de esta forma **>>> Comentario # <<<**:

Los cuatro últimos sobre el código de *Training*

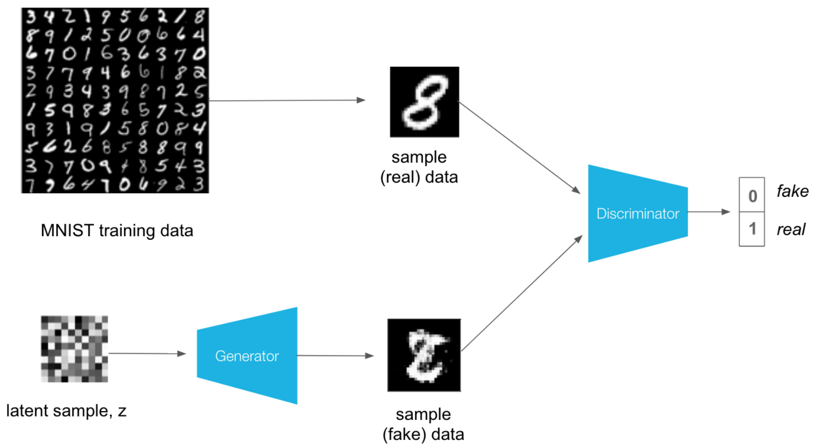
Generative Adversarial Network

In this exercise, you'll be building a generative adversarial network (GAN) trained on the MNIST dataset. From this, new handwritten digits will be generated!

The idea behind GANs is that you have two networks, a generator G and a discriminator D, competing against each other. The generator makes "fake" data to pass to the discriminator. The discriminator also sees real training data and predicts if the data it's received is real or fake.

> - The generator is trained to fool the discriminator, it wants to output data that looks *as close as possible* to real, training data.
> - The discriminator is a classifier that is trained to figure out which data is real and which is fake.

What ends up happening is that the generator learns to make data that is indistinguishable from real data to the discriminator.



The general structure of a GAN is shown in the diagram above, using MNIST images as data. The latent sample is a random vector that the generator uses to construct its fake images. This is often called a **latent vector** and that vector space is called **latent space**. As the generator trains, it figures out how to map latent vectors to recognizable images that can fool the discriminator.

If you're interested in generating only new images, you can throw out the discriminator after training. In this exercise, you will define and train these adversarial networks in PyTorch and generate new images!

In [2]:
```
%matplotlib inline
```

```
import numpy as np
import torch
import matplotlib.pyplot as plt
```

In [3]:

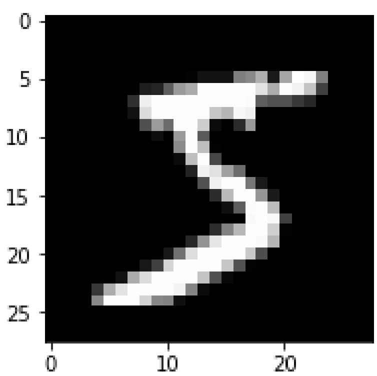
```
from torchvision import datasets
import torchvision.transforms as transforms

# number of subprocesses to use for data loading
num_workers = 0
# how many samples per batch to load
batch_size = 64

# convert data to torch.FloatTensor
transform = transforms.ToTensor()

# get the training datasets
train_data = datasets.MNIST(root='data', train=True, download=True, transform

# prepare data loader
train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size,
```

Visualize the data

In [4]:

```
# obtain one batch of training images
dataiter = iter(train_loader)
images, labels = dataiter.next()
images = images.numpy()

# get one image from the batch
img = np.squeeze(images[0])

fig = plt.figure(figsize = (3,3))
ax = fig.add_subplot(111)
ax.imshow(img, cmap='gray')
```

Out[4]: <matplotlib.image.AxesImage at 0x7fda9bc34090>



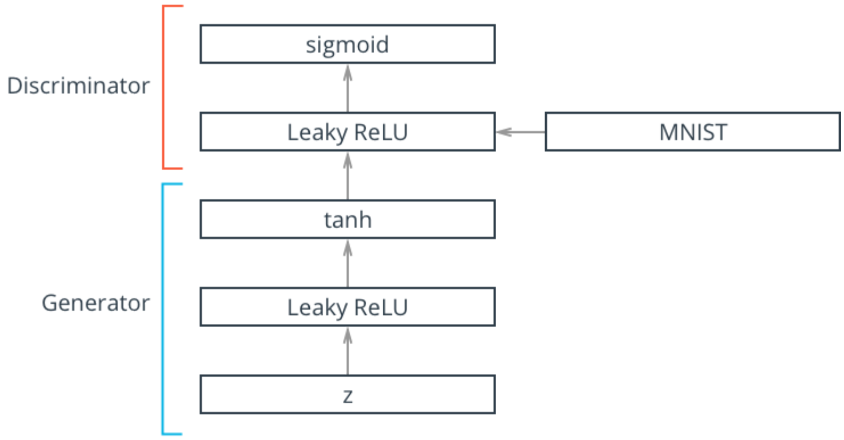
Define the Model

A GAN is comprised of two adversarial networks, a discriminator and a generator.

Discriminator

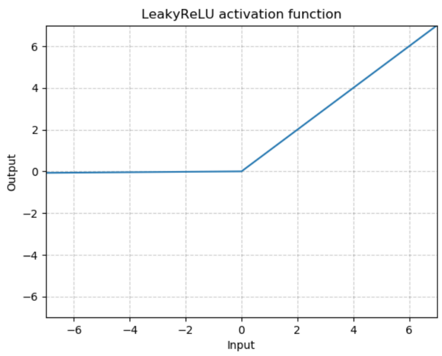
The discriminator network is going to be a pretty typical linear classifier. To make this network a universal function approximator, we'll need at least one hidden layer, and these hidden layers should have one key attribute:

> All hidden layers will have a Leaky ReLu activation function applied to their outputs.



Leaky ReLu

We should use a leaky ReLU to allow gradients to flow backwards through the layer unimpeded. A leaky ReLU is like a normal ReLU, except that there is a small non-zero output for negative input values. It has a small slope for negative values, instead of altogether zero. For example, leaky ReLU may have y = 0.01x when x < 0.



Sigmoid Output

We'll also take the approach of using a more numerically stable loss function on the outputs. Recall that we want the discriminator to output a value 0-1 indicating whether an image is *real or fake*.

> We will ultimately use BCEWithLogitsLoss, which combines a `sigmoid` activation function **and** binary cross entropy loss in one function.

So, our final output layer should not have any activation function applied to it.

>>> Comentario 1 <<<

Estructura de las capas del discrimidador

Orden a aplicación	Comentario
FC1	capa de entrada de la imatgen (gris) 28x28
LeakyReLU	activación no lineal, recta con pendiente 0.01 para x<0 y pendiente 1 para x>=0
Dropout	para mitigar *overfitting*
FC2	oculta, le asignamos la mitad del número de neuronas de salida + entrada
LeakyReLU	este tipo de función (no nula excepto en x=0) permite *backpropagation* plena (desde el discrimidador hasta la primera capa del generador)
Dropout	
FC3	capa de salida, nos dirá si la imagen es verdadera o falsa

In [5]:

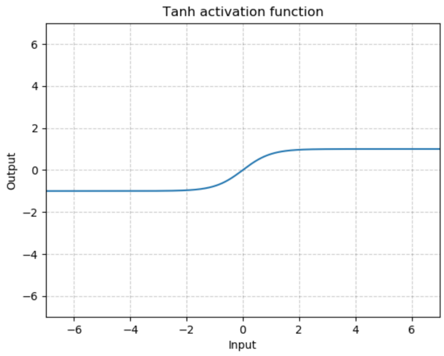
```
import torch.nn as nn
import torch.nn.functional as F


class Discriminator(nn.Module):


    def __init__(self, input_size, hidden_dim, output_size):
        super(Discriminator, self).__init__()

        # define all layers
        #TODO
        #DONE
        self.fc1 = nn.Linear(input_size, hidden_dim)
        self.do1 = nn.Dropout(0.1)
        self.fc2 = nn.Linear(hidden_dim, hidden_dim)
        self.do2 = nn.Dropout(0.1)
        self.fc3 = nn.Linear(hidden_dim, output_size)


    def forward(self, x):
        # flatten image
        #TODO
        #DONE
        x = torch.flatten(x, 1)

        # pass x through all layers
        # apply leaky relu activation to all hidden layers
        #TODO
        #DONE
        x = self.fc1(x)
        x = F.leaky_relu(x)
        x = self.do1(x)
        x = self.fc2(x)
        x = F.leaky_relu(x)
        x = self.do2(x)
        x = self.fc3(x)

        return x
```

Generator

The generator network will be almost exactly the same as the discriminator network, except that we're applying a tanh activation function to our output layer.

tanh Output

The generator has been found to perform the best with $tanh$ for the generator output, which scales the output to be between -1 and 1, instead of 0 and 1.



Recall that we also want these outputs to be comparable to the *real* input pixel values, which are read in as normalized values between 0 and 1.

> So, we'll also have to **scale our real input images to have pixel values between -1 and 1** when we train the discriminator.

This is done in the training loop, later on.

>>> Comentario 2 <<<

Estructura de las capas del generador

Orden de aplicación	Comentario
FC1	capa de entrada del ruido (entropia), suponemos imagen 4x4
LeakyReLU	activación no lineal, recta con pendiente 0.01 para x<0 y pendiente 1 para x>=0
Dropout	mitigar *overfitting*
FC2	oculta, le asignamos la mitad del número de neuronas de salida + entrada
LeakyReLU	este tipo de función (no nula excepto en x=0) permite *backpropagation* plena (desde el discrimidador hasta la primera capa del generador)
Dropout	
FC3	capa de salida, tendrá la misma dimensión de las imágenes a comparar
Tanh	la creación de imágenes funciona mejor usando valores del intervalo (-1,1)

In [6]:

```
class Generator(nn.Module):


    def __init__(self, input_size, hidden_dim, output_size):
        super(Generator, self).__init__()

        # define all layers
        #TODO
        #DONE
        self.fc1 = nn.Linear(input_size, hidden_dim)
        self.do1 = nn.Dropout(0.2)
        self.fc2 = nn.Linear(hidden_dim, hidden_dim)
        self.do2 = nn.Dropout(0.2)
        self.fc3 = nn.Linear(hidden_dim, output_size)

    def forward(self, x):
        # pass x through all layers
```

```
        # final layer should have tanh applied
        #TODO
        #DONE
        x = self.fc1(x)
        x = F.leaky_relu(x)
        x = self.do1(x)
        x = self.fc2(x)
        x = F.leaky_relu(x)
        x = self.do2(x)
        x = self.fc3(x)
        x = torch.tanh(x)


        return x
```

Model hyperparameters

>>> Comentario 3 <<<

Caso discriminador

Imagen (entrada) -> clasificación: verdadera o falsa (salida)

- Entra imagen monocromo de 28x28 píxeles: 1 *28* 28
- Sale clasificación verdadera/falsa: en lugar de una sola neurona, usamos 2, una para cada clase (c1: imagen es verdadera, c2: imagen es falsa). Pensamos que usando dos neuronas para *mapear* dos clases, en lugar de una sola neurona, nos va a dar mayor robustez.
- Capa oculta: a falta de hacer pruebas, de momento la mitad de las neuronas de entrada + salida

Caso generador

Ruido (entrada) -> imagen (salida)

- Entra entropía en forma de ruido, supondremos *imagen* monocroma de 4x4 píxeles: 4 *4* 1
- Sale imagen monocromo del mismo tamaño que las verdaderas: 28 *28* 1
- Capa oculta: a falta de hacer pruebas, de momento la mitad de las neuronas de entrada + salida

In [7]:

```
# Discriminator hyperparams
#TODO
#DONE
# Size of input image to discriminator (28*28)
input_size = 28 * 28
# Size of discriminator output (real or fake)
d_output_size = 2
# Size of *last* hidden layer in the discriminator
d_hidden_size = (input_size + d_output_size) // 2

# Generator hyperparams
#TODO
#DONE
# Size of latent vector to give to generator
z_size = 4 * 4
```

```
# Size of discriminator output (generated image)
g_output_size = 28 * 28
# Size of *first* hidden layer in the generator
g_hidden_size = (z_size + g_output_size) // 2
```

Build complete network

Now we're instantiating the discriminator and generator from the classes defined above. Make sure you've passed in the correct input arguments.

In [8]:

```
# instantiate discriminator and generator
D = Discriminator(input_size, d_hidden_size, d_output_size)
G = Generator(z_size, g_hidden_size, g_output_size)

# check that they are as you expect
print(D)
print()
print(G)
```

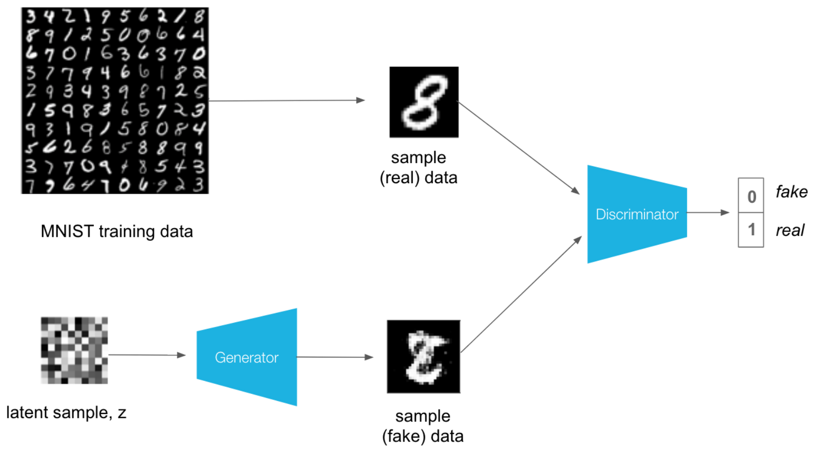
```
Discriminator(
  (fc1): Linear(in_features=784, out_features=393, bias=True)
  (do1): Dropout(p=0.1, inplace=False)
  (fc2): Linear(in_features=393, out_features=393, bias=True)
  (do2): Dropout(p=0.1, inplace=False)
  (fc3): Linear(in_features=393, out_features=2, bias=True)
)

Generator(
  (fc1): Linear(in_features=16, out_features=400, bias=True)
  (do1): Dropout(p=0.2, inplace=False)
  (fc2): Linear(in_features=400, out_features=400, bias=True)
  (do2): Dropout(p=0.2, inplace=False)
  (fc3): Linear(in_features=400, out_features=784, bias=True)
)
```

Discriminator and Generator Losses

Now we need to calculate the losses.

Discriminator Losses

> - For the discriminator, the total loss is the sum of the losses for real and fake images, `d_loss = d_real_loss + d_fake_loss` .
> - Remember that we want the discriminator to output 1 for real images and 0 for fake images, so we need to set up the losses to reflect that.



The losses will by binary cross entropy loss with logits, which we can get with BCEWithLogitsLoss. This combines a `sigmoid` activation function **and** binary cross entropy loss in one function.

For the real images, we want `D(real_images) = 1`. That is, we want the discriminator to classify the real images with a label = 1, indicating that these are real. To help the discriminator generalize better, the labels are **reduced a bit from 1.0 to 0.9**. For this, we'll use the parameter `smooth`; if True, then we should smooth our labels. In PyTorch, this looks like `labels = torch.ones(size) * 0.9`

The discriminator loss for the fake data is similar. We want `D(fake_images) = 0`, where the fake images are the *generator output*, `fake_images = G(z)`.

Generator Loss

The generator loss will look similar only with flipped labels. The generator's goal is to get `D(fake_images) = 1`. In this case, the labels are **flipped** to represent that the generator is trying to fool the discriminator into thinking that the images it generates (fakes) are real!

>>> Comentario 4 <<<

En la capa de salida del discriminador tenemos dos neuronas que representan dos canales. El primero, para señalar las imágenes reales y el segundo para las falsas.

real_loss

Por tanto el tensor de etiquetas para el caso de querer calcular la función de pérdida de un lote tendrá unos (1) en la primera columna (canal para marcar las imágenes reales) y ceros (0) en la segunda columna (canal imágenes falsas) y tantas filas como imágenes en el lote.

Ejemplo de confección de etiquetas para un lote de 5 imágenes reales:

```
labels = torch.zeros([5,2]) # lotes de 5 imágenes, dos canales
labels[:,0] = 1
print(labels)
tensor([[1., 0.],
        [1., 0.],
        [1., 0.],
```

```
        [1., 0.],
        [1., 0.]])
```

fake_loss

Aquí deberemos etiquetar como correcta la imagen falsa. Por tanto el tensor de etiquetas tendrá su segunda columna a unos (1) y la primera a ceros (0).

In [9]:

```python
# Calculate losses

def real_loss(D_out, smooth=False):
    # compare logits to real labels
    # smooth labels if smooth=True
    #TODO
    #DONE

    labels = torch.zeros(D_out.shape)

    if smooth:
        labels[:,0] = 0.9 # Primera columna a 0.9
    else:
        labels[:,0] = 1.0 # Primera columna a uno (1)

    criterion = torch.nn.BCEWithLogitsLoss()

    loss = criterion(D_out, labels)

    return loss

def fake_loss(D_out):
    # compare logits to fake labels
    #TODO
    #DONE

    labels = torch.zeros(D_out.shape)

    labels[:,1] = 1.0 # Sólo segunda columna a uno (1)

    criterion = torch.nn.BCEWithLogitsLoss()

    loss = criterion(D_out, labels)

    return loss
```

Optimizers

We want to update the generator and discriminator variables separately. So, we'll define two separate Adam optimizers.

In [10]:

```python
import torch.optim as optim

# learning rate for optimizers
lr = 0.002


# Create optimizers for the discriminator and generator
#TODO
#DONE
```

```
d_optimizer = optim.Adam(D.parameters(), lr)

g_optimizer = optim.Adam(G.parameters(), lr)
```

Training

Training will involve alternating between training the discriminator and the generator. We'll use our functions `real_loss` and `fake_loss` to help us calculate the discriminator losses in all of the following cases.

Discriminator training

1. Compute the discriminator loss on real, training images
2. Generate fake images
3. Compute the discriminator loss on fake, generated images
4. Add up real and fake loss
5. Perform backpropagation + an optimization step to update the discriminator's weights

Generator training

1. Generate fake images
2. Compute the discriminator loss on fake images, using **flipped** labels!
3. Perform backpropagation + an optimization step to update the generator's weights

Saving Samples

As we train, we'll also print out some loss statistics and save some generated "fake" samples.

In [11]:
```
import pickle as pkl

# training hyperparams
num_epochs = 100  #TODO (it could be changed) #DONE

# keep track of loss and generated, "fake" samples
samples = []
losses = []

print_every = 400

# Get some fixed data for sampling. These are images that are held
# constant throughout training, and allow us to inspect the model's performar
sample_size=16
fixed_z = np.random.uniform(-1, 1, size=(sample_size, z_size))
fixed_z = torch.from_numpy(fixed_z).float()

# train the network
D.train()
G.train()
for epoch in range(num_epochs):

    for batch_i, (real_images, _) in enumerate(train_loader):

        batch_size = real_images.size(0)
```

```
        ## Important rescaling step ##
        real_images = real_images * (-2) + 1  # rescale input images from [0,


        # ==============================================
        #            TRAIN THE DISCRIMINATOR
        # ==============================================
        #TODO
        #DONE

        # >>> Comentario 5 <<<
        #
        # el re-escalado a [-1, 1) de las imágenes (línea anterior a este com
        # ha sido invertido para obtenerlas en trazo negro sobre fondo blanco
        # porque nos gusta más así :-)
        #
        # empezamos otro lote, así que hay que
        # poner a cero los gradientes acumulados en el ciclo anterior
        #
        D.zero_grad()

        # 1. Train with real images
        # Compute the discriminator losses on real images
        # use smoothed labels

        d_real_decision = D(real_images)

        d_real_loss = real_loss(d_real_decision, smooth = True)

        #TODO
        #DONE

        # 2. Train with fake images

        # Generate fake images
        z = np.random.uniform(-1, 1, size=(batch_size, z_size))
        z = torch.from_numpy(z).float()
        #fake_images = G(z)
        #
        # >>> Comentario 6 <<<
        #
        # detach() para evitar el cálculo innecesario de los gradientes
        # en el generador
        #
        fake_images = G(z).detach()

        # Compute the discriminator losses on fake images
        #TODO
        #DONE

        # >>> Comentario 7 <<<
        #
        # pasamos el lote de imágenes falsas por el discriminador
        #
        d_fake_decision = D(fake_images)

        d_fake_loss = fake_loss(d_fake_decision)

        # add up real and fake losses and perform backprop
        #TODO
        #DONE

        d_loss = d_real_loss + d_fake_loss

        d_loss.backward()
```

```
        d_optimizer.step()

        # ==========================================
        #            TRAIN THE GENERATOR
        # ==========================================
        #TODO
        #DONE

        G.zero_grad()

        # 1. Train with fake images and flipped labels

        # Generate fake images
        #TODO
        #DONE

        z = np.random.uniform(-1, 1, size=(batch_size, z_size))
        z = torch.from_numpy(z).float()
        fake_images = G(z)

        # Compute the discriminator losses on fake images
        # using flipped labels!
        #TODO
        #DONE

        # >>> Comentario 8 <<<
        #
        # pasamos el lote de imágenes falsas por el discriminador
        #
        g_fake_decision = D(fake_images)

        # perform backprop
        #TODO
        #DONE

        g_loss = real_loss(g_fake_decision)

        g_loss.backward()

        g_optimizer.step()

        # Print some loss stats
        if batch_i % print_every == 0:
            # print discriminator and generator loss
            print('Epoch [{:5d}/{:5d}] | d_loss: {:6.4f} | g_loss: {:6.4f}'.f
                    epoch+1, num_epochs, d_loss.item(), g_loss.item()))


    ## AFTER EACH EPOCH##
    # append discriminator loss and generator loss
    losses.append((d_loss.item(), g_loss.item()))

    # generate and save sample, fake images
    G.eval() # eval mode for generating samples
    samples_z = G(fixed_z)
    samples.append(samples_z)
    G.train() # back to train mode


# Save training generator samples
with open('train_samples.pkl', 'wb') as f:
    pkl.dump(samples, f)
```

```
Epoch [    1/  100] | d_loss: 1.3629 | g_loss: 0.6955
```

```
Epoch [    1/  100] | d_loss: 0.6467 | g_loss: 5.8848
Epoch [    1/  100] | d_loss: 0.3686 | g_loss: 4.9208
Epoch [    2/  100] | d_loss: 0.9999 | g_loss: 1.8344
Epoch [    2/  100] | d_loss: 0.4877 | g_loss: 4.2266
Epoch [    2/  100] | d_loss: 0.8777 | g_loss: 2.3573
Epoch [    3/  100] | d_loss: 0.9087 | g_loss: 2.2708
Epoch [    3/  100] | d_loss: 0.8059 | g_loss: 1.8934
Epoch [    3/  100] | d_loss: 0.8141 | g_loss: 1.8604
Epoch [    4/  100] | d_loss: 1.1674 | g_loss: 1.6101
Epoch [    4/  100] | d_loss: 0.6469 | g_loss: 1.7969
Epoch [    4/  100] | d_loss: 0.6966 | g_loss: 2.8498
Epoch [    5/  100] | d_loss: 0.9479 | g_loss: 1.8198
Epoch [    5/  100] | d_loss: 0.7920 | g_loss: 2.1537
Epoch [    5/  100] | d_loss: 0.9547 | g_loss: 1.5441
Epoch [    6/  100] | d_loss: 0.8249 | g_loss: 1.7607
Epoch [    6/  100] | d_loss: 1.0490 | g_loss: 1.7918
Epoch [    6/  100] | d_loss: 0.9895 | g_loss: 1.2462
Epoch [    7/  100] | d_loss: 1.1373 | g_loss: 1.3041
Epoch [    7/  100] | d_loss: 1.0538 | g_loss: 1.3995
Epoch [    7/  100] | d_loss: 1.1821 | g_loss: 1.2854
Epoch [    8/  100] | d_loss: 1.3065 | g_loss: 1.0700
Epoch [    8/  100] | d_loss: 1.2248 | g_loss: 1.1293
Epoch [    8/  100] | d_loss: 1.1202 | g_loss: 0.8936
Epoch [    9/  100] | d_loss: 1.1963 | g_loss: 1.2716
Epoch [    9/  100] | d_loss: 1.4345 | g_loss: 1.2749
Epoch [    9/  100] | d_loss: 1.1864 | g_loss: 0.8857
Epoch [   10/  100] | d_loss: 1.2065 | g_loss: 0.9332
Epoch [   10/  100] | d_loss: 1.0704 | g_loss: 1.1559
Epoch [   10/  100] | d_loss: 1.3335 | g_loss: 0.9683
Epoch [   11/  100] | d_loss: 1.2161 | g_loss: 1.0844
Epoch [   11/  100] | d_loss: 1.0642 | g_loss: 1.0944
Epoch [   11/  100] | d_loss: 1.3290 | g_loss: 0.9437
Epoch [   12/  100] | d_loss: 1.2621 | g_loss: 1.3032
Epoch [   12/  100] | d_loss: 1.2850 | g_loss: 1.0808
Epoch [   12/  100] | d_loss: 1.2489 | g_loss: 1.0242
Epoch [   13/  100] | d_loss: 1.3282 | g_loss: 0.8804
Epoch [   13/  100] | d_loss: 1.2600 | g_loss: 0.9934
Epoch [   13/  100] | d_loss: 1.3724 | g_loss: 1.0831
Epoch [   14/  100] | d_loss: 1.3044 | g_loss: 1.1286
Epoch [   14/  100] | d_loss: 1.2286 | g_loss: 1.1243
Epoch [   14/  100] | d_loss: 1.1928 | g_loss: 0.8236
Epoch [   15/  100] | d_loss: 1.2975 | g_loss: 1.3106
Epoch [   15/  100] | d_loss: 1.1163 | g_loss: 1.1549
Epoch [   15/  100] | d_loss: 1.3531 | g_loss: 1.0640
Epoch [   16/  100] | d_loss: 1.2201 | g_loss: 1.1937
Epoch [   16/  100] | d_loss: 1.0934 | g_loss: 1.3115
Epoch [   16/  100] | d_loss: 1.5908 | g_loss: 1.0800
Epoch [   17/  100] | d_loss: 1.2238 | g_loss: 0.9095
Epoch [   17/  100] | d_loss: 1.1395 | g_loss: 1.0425
Epoch [   17/  100] | d_loss: 1.3073 | g_loss: 0.9726
Epoch [   18/  100] | d_loss: 1.4426 | g_loss: 0.8934
Epoch [   18/  100] | d_loss: 1.1988 | g_loss: 1.0070
Epoch [   18/  100] | d_loss: 1.2948 | g_loss: 0.9945
Epoch [   19/  100] | d_loss: 1.2441 | g_loss: 1.0923
Epoch [   19/  100] | d_loss: 1.3148 | g_loss: 1.0592
Epoch [   19/  100] | d_loss: 1.2779 | g_loss: 1.0209
Epoch [   20/  100] | d_loss: 1.1760 | g_loss: 1.1902
Epoch [   20/  100] | d_loss: 1.1737 | g_loss: 1.0303
Epoch [   20/  100] | d_loss: 1.2780 | g_loss: 1.0424
Epoch [   21/  100] | d_loss: 1.1423 | g_loss: 1.1006
Epoch [   21/  100] | d_loss: 1.0758 | g_loss: 1.0419
Epoch [   21/  100] | d_loss: 1.0872 | g_loss: 1.1972
Epoch [   22/  100] | d_loss: 1.1304 | g_loss: 1.1766
Epoch [   22/  100] | d_loss: 1.1446 | g_loss: 1.1971
Epoch [   22/  100] | d_loss: 1.0996 | g_loss: 1.5876
Epoch [   23/  100] | d_loss: 1.0525 | g_loss: 1.3436
Epoch [   23/  100] | d_loss: 0.9737 | g_loss: 1.1700
Epoch [   23/  100] | d_loss: 0.9634 | g_loss: 1.3370
Epoch [   24/  100] | d_loss: 1.0663 | g_loss: 1.6866
```

```
Epoch [   24/  100] | d_loss: 1.1503 | g_loss: 1.2314
Epoch [   24/  100] | d_loss: 1.0397 | g_loss: 1.4092
Epoch [   25/  100] | d_loss: 1.3307 | g_loss: 1.5804
Epoch [   25/  100] | d_loss: 1.1102 | g_loss: 1.1823
Epoch [   25/  100] | d_loss: 1.2034 | g_loss: 1.1884
Epoch [   26/  100] | d_loss: 0.9700 | g_loss: 1.5021
Epoch [   26/  100] | d_loss: 1.0555 | g_loss: 1.1423
Epoch [   26/  100] | d_loss: 1.1395 | g_loss: 1.5137
Epoch [   27/  100] | d_loss: 1.0198 | g_loss: 1.3431
Epoch [   27/  100] | d_loss: 1.0984 | g_loss: 1.2087
Epoch [   27/  100] | d_loss: 1.0384 | g_loss: 1.3401
Epoch [   28/  100] | d_loss: 0.9406 | g_loss: 1.8121
Epoch [   28/  100] | d_loss: 1.0930 | g_loss: 1.1140
Epoch [   28/  100] | d_loss: 1.2613 | g_loss: 1.3635
Epoch [   29/  100] | d_loss: 1.1655 | g_loss: 1.3701
Epoch [   29/  100] | d_loss: 1.1122 | g_loss: 1.3975
Epoch [   29/  100] | d_loss: 1.1453 | g_loss: 1.1876
Epoch [   30/  100] | d_loss: 1.1007 | g_loss: 1.1794
Epoch [   30/  100] | d_loss: 0.9934 | g_loss: 1.1776
Epoch [   30/  100] | d_loss: 1.0452 | g_loss: 1.6861
Epoch [   31/  100] | d_loss: 1.2922 | g_loss: 1.4154
Epoch [   31/  100] | d_loss: 1.0097 | g_loss: 1.2674
Epoch [   31/  100] | d_loss: 1.1272 | g_loss: 1.0421
Epoch [   32/  100] | d_loss: 1.2632 | g_loss: 1.2567
Epoch [   32/  100] | d_loss: 1.0534 | g_loss: 1.6410
Epoch [   32/  100] | d_loss: 1.1043 | g_loss: 1.6980
Epoch [   33/  100] | d_loss: 1.1540 | g_loss: 1.5908
Epoch [   33/  100] | d_loss: 1.0792 | g_loss: 1.3799
Epoch [   33/  100] | d_loss: 1.1077 | g_loss: 1.3843
Epoch [   34/  100] | d_loss: 1.1487 | g_loss: 1.4400
Epoch [   34/  100] | d_loss: 1.0219 | g_loss: 1.0474
Epoch [   34/  100] | d_loss: 1.2872 | g_loss: 1.2073
Epoch [   35/  100] | d_loss: 1.0764 | g_loss: 1.4818
Epoch [   35/  100] | d_loss: 1.1665 | g_loss: 1.3610
Epoch [   35/  100] | d_loss: 1.1763 | g_loss: 1.1575
Epoch [   36/  100] | d_loss: 1.2415 | g_loss: 1.3708
Epoch [   36/  100] | d_loss: 1.0224 | g_loss: 1.4120
Epoch [   36/  100] | d_loss: 1.1511 | g_loss: 1.0408
Epoch [   37/  100] | d_loss: 1.1454 | g_loss: 1.3545
Epoch [   37/  100] | d_loss: 1.2238 | g_loss: 1.1536
Epoch [   37/  100] | d_loss: 1.2462 | g_loss: 1.4248
Epoch [   38/  100] | d_loss: 1.2235 | g_loss: 1.3085
Epoch [   38/  100] | d_loss: 1.1373 | g_loss: 1.2078
Epoch [   38/  100] | d_loss: 1.4170 | g_loss: 1.1861
Epoch [   39/  100] | d_loss: 1.1876 | g_loss: 1.2737
Epoch [   39/  100] | d_loss: 1.1337 | g_loss: 1.2933
Epoch [   39/  100] | d_loss: 1.2148 | g_loss: 1.2277
Epoch [   40/  100] | d_loss: 1.1959 | g_loss: 1.3256
Epoch [   40/  100] | d_loss: 1.1656 | g_loss: 1.0804
Epoch [   40/  100] | d_loss: 1.2055 | g_loss: 1.3667
Epoch [   41/  100] | d_loss: 1.1599 | g_loss: 1.3311
Epoch [   41/  100] | d_loss: 0.9644 | g_loss: 1.2173
Epoch [   41/  100] | d_loss: 1.2414 | g_loss: 1.3292
Epoch [   42/  100] | d_loss: 1.3309 | g_loss: 1.1658
Epoch [   42/  100] | d_loss: 1.0905 | g_loss: 1.1134
Epoch [   42/  100] | d_loss: 1.1524 | g_loss: 1.2718
Epoch [   43/  100] | d_loss: 1.1699 | g_loss: 1.3542
Epoch [   43/  100] | d_loss: 1.0600 | g_loss: 1.2217
Epoch [   43/  100] | d_loss: 1.2546 | g_loss: 1.1723
Epoch [   44/  100] | d_loss: 1.1488 | g_loss: 1.6364
Epoch [   44/  100] | d_loss: 1.1167 | g_loss: 1.9689
Epoch [   44/  100] | d_loss: 1.3412 | g_loss: 1.3673
Epoch [   45/  100] | d_loss: 1.1954 | g_loss: 1.5489
Epoch [   45/  100] | d_loss: 1.1688 | g_loss: 1.1780
Epoch [   45/  100] | d_loss: 1.2277 | g_loss: 1.3372
Epoch [   46/  100] | d_loss: 1.2190 | g_loss: 1.3593
Epoch [   46/  100] | d_loss: 1.1314 | g_loss: 1.1889
Epoch [   46/  100] | d_loss: 1.2720 | g_loss: 1.4676
Epoch [   47/  100] | d_loss: 1.0660 | g_loss: 1.5052
```

```
Epoch [   47/  100] | d_loss: 1.0738 | g_loss: 1.3059
Epoch [   47/  100] | d_loss: 1.3907 | g_loss: 1.0089
Epoch [   48/  100] | d_loss: 1.1970 | g_loss: 1.9163
Epoch [   48/  100] | d_loss: 1.0644 | g_loss: 1.1114
Epoch [   48/  100] | d_loss: 1.2311 | g_loss: 1.1873
Epoch [   49/  100] | d_loss: 1.2138 | g_loss: 1.1408
Epoch [   49/  100] | d_loss: 1.1029 | g_loss: 1.2937
Epoch [   49/  100] | d_loss: 1.2012 | g_loss: 1.3235
Epoch [   50/  100] | d_loss: 1.0184 | g_loss: 1.5261
Epoch [   50/  100] | d_loss: 1.0036 | g_loss: 1.4858
Epoch [   50/  100] | d_loss: 1.1464 | g_loss: 1.3859
Epoch [   51/  100] | d_loss: 1.1575 | g_loss: 1.5850
Epoch [   51/  100] | d_loss: 1.0514 | g_loss: 1.2687
Epoch [   51/  100] | d_loss: 1.1045 | g_loss: 1.4116
Epoch [   52/  100] | d_loss: 1.0527 | g_loss: 1.5588
Epoch [   52/  100] | d_loss: 1.1332 | g_loss: 1.1816
Epoch [   52/  100] | d_loss: 1.1618 | g_loss: 1.1627
Epoch [   53/  100] | d_loss: 1.0604 | g_loss: 1.2802
Epoch [   53/  100] | d_loss: 1.0399 | g_loss: 1.2407
Epoch [   53/  100] | d_loss: 1.0710 | g_loss: 1.2809
Epoch [   54/  100] | d_loss: 1.1023 | g_loss: 1.7144
Epoch [   54/  100] | d_loss: 1.2234 | g_loss: 1.5108
Epoch [   54/  100] | d_loss: 1.0958 | g_loss: 1.1212
Epoch [   55/  100] | d_loss: 0.9168 | g_loss: 1.5582
Epoch [   55/  100] | d_loss: 0.9347 | g_loss: 1.2542
Epoch [   55/  100] | d_loss: 1.0115 | g_loss: 1.3014
Epoch [   56/  100] | d_loss: 1.0655 | g_loss: 1.7834
Epoch [   56/  100] | d_loss: 1.1492 | g_loss: 1.5283
Epoch [   56/  100] | d_loss: 1.1039 | g_loss: 1.1989
Epoch [   57/  100] | d_loss: 1.1619 | g_loss: 1.6426
Epoch [   57/  100] | d_loss: 0.9768 | g_loss: 1.4038
Epoch [   57/  100] | d_loss: 1.1033 | g_loss: 1.2672
Epoch [   58/  100] | d_loss: 1.0495 | g_loss: 1.4199
Epoch [   58/  100] | d_loss: 1.0033 | g_loss: 1.3305
Epoch [   58/  100] | d_loss: 1.0320 | g_loss: 1.6801
Epoch [   59/  100] | d_loss: 1.0443 | g_loss: 2.3375
Epoch [   59/  100] | d_loss: 1.0290 | g_loss: 1.9096
Epoch [   59/  100] | d_loss: 0.9079 | g_loss: 1.4617
Epoch [   60/  100] | d_loss: 1.1933 | g_loss: 1.7568
Epoch [   60/  100] | d_loss: 1.1443 | g_loss: 1.1867
Epoch [   60/  100] | d_loss: 1.2768 | g_loss: 1.3279
Epoch [   61/  100] | d_loss: 1.0172 | g_loss: 1.7639
Epoch [   61/  100] | d_loss: 0.9013 | g_loss: 1.2537
Epoch [   61/  100] | d_loss: 1.0456 | g_loss: 1.2955
Epoch [   62/  100] | d_loss: 1.0707 | g_loss: 1.8388
Epoch [   62/  100] | d_loss: 1.2148 | g_loss: 1.0362
Epoch [   62/  100] | d_loss: 1.2212 | g_loss: 1.3968
Epoch [   63/  100] | d_loss: 1.0890 | g_loss: 1.4725
Epoch [   63/  100] | d_loss: 0.9737 | g_loss: 1.5733
Epoch [   63/  100] | d_loss: 1.0945 | g_loss: 1.1866
Epoch [   64/  100] | d_loss: 1.1530 | g_loss: 1.4461
Epoch [   64/  100] | d_loss: 1.0907 | g_loss: 1.4492
Epoch [   64/  100] | d_loss: 1.1350 | g_loss: 1.3316
Epoch [   65/  100] | d_loss: 1.1984 | g_loss: 1.5681
Epoch [   65/  100] | d_loss: 1.0649 | g_loss: 1.4102
Epoch [   65/  100] | d_loss: 1.2636 | g_loss: 1.4952
Epoch [   66/  100] | d_loss: 1.0727 | g_loss: 1.4714
Epoch [   66/  100] | d_loss: 1.0799 | g_loss: 1.2308
Epoch [   66/  100] | d_loss: 1.1552 | g_loss: 1.2444
Epoch [   67/  100] | d_loss: 1.1517 | g_loss: 1.4801
Epoch [   67/  100] | d_loss: 1.0285 | g_loss: 1.3249
Epoch [   67/  100] | d_loss: 1.1673 | g_loss: 1.3889
Epoch [   68/  100] | d_loss: 1.0188 | g_loss: 1.6486
Epoch [   68/  100] | d_loss: 0.9628 | g_loss: 1.3122
Epoch [   68/  100] | d_loss: 1.2620 | g_loss: 1.1648
Epoch [   69/  100] | d_loss: 1.2009 | g_loss: 1.2307
Epoch [   69/  100] | d_loss: 0.9537 | g_loss: 2.0804
Epoch [   69/  100] | d_loss: 1.4151 | g_loss: 1.6207
Epoch [   70/  100] | d_loss: 1.1854 | g_loss: 1.4176
```

```
Epoch [   70/  100] | d_loss: 1.2417 | g_loss: 1.4960
Epoch [   70/  100] | d_loss: 1.1385 | g_loss: 1.4049
Epoch [   71/  100] | d_loss: 1.0904 | g_loss: 1.4364
Epoch [   71/  100] | d_loss: 1.1284 | g_loss: 1.3581
Epoch [   71/  100] | d_loss: 1.0879 | g_loss: 1.1342
Epoch [   72/  100] | d_loss: 1.0504 | g_loss: 1.4355
Epoch [   72/  100] | d_loss: 1.0343 | g_loss: 1.7089
Epoch [   72/  100] | d_loss: 1.2033 | g_loss: 1.3609
Epoch [   73/  100] | d_loss: 1.0767 | g_loss: 1.2830
Epoch [   73/  100] | d_loss: 0.8596 | g_loss: 1.4319
Epoch [   73/  100] | d_loss: 1.1528 | g_loss: 1.2245
Epoch [   74/  100] | d_loss: 1.1933 | g_loss: 1.5872
Epoch [   74/  100] | d_loss: 0.9714 | g_loss: 1.5555
Epoch [   74/  100] | d_loss: 1.1913 | g_loss: 1.4545
Epoch [   75/  100] | d_loss: 1.0838 | g_loss: 1.4534
Epoch [   75/  100] | d_loss: 0.9985 | g_loss: 1.4350
Epoch [   75/  100] | d_loss: 1.1729 | g_loss: 1.5004
Epoch [   76/  100] | d_loss: 0.9905 | g_loss: 1.9510
Epoch [   76/  100] | d_loss: 1.0857 | g_loss: 1.3225
Epoch [   76/  100] | d_loss: 1.0481 | g_loss: 1.3420
Epoch [   77/  100] | d_loss: 1.1436 | g_loss: 1.5492
Epoch [   77/  100] | d_loss: 0.9948 | g_loss: 1.3673
Epoch [   77/  100] | d_loss: 1.1805 | g_loss: 1.2948
Epoch [   78/  100] | d_loss: 1.1727 | g_loss: 1.5550
Epoch [   78/  100] | d_loss: 0.9178 | g_loss: 1.5042
Epoch [   78/  100] | d_loss: 1.1109 | g_loss: 1.5470
Epoch [   79/  100] | d_loss: 1.0771 | g_loss: 1.8611
Epoch [   79/  100] | d_loss: 1.0550 | g_loss: 1.3915
Epoch [   79/  100] | d_loss: 1.0048 | g_loss: 1.4555
Epoch [   80/  100] | d_loss: 1.1371 | g_loss: 1.6715
Epoch [   80/  100] | d_loss: 1.0539 | g_loss: 1.4241
Epoch [   80/  100] | d_loss: 0.8708 | g_loss: 1.1971
Epoch [   81/  100] | d_loss: 1.0515 | g_loss: 1.6131
Epoch [   81/  100] | d_loss: 1.0312 | g_loss: 1.3886
Epoch [   81/  100] | d_loss: 0.9212 | g_loss: 1.3302
Epoch [   82/  100] | d_loss: 0.9914 | g_loss: 1.5833
Epoch [   82/  100] | d_loss: 0.9324 | g_loss: 1.5259
Epoch [   82/  100] | d_loss: 1.1520 | g_loss: 1.0740
Epoch [   83/  100] | d_loss: 1.1309 | g_loss: 1.6534
Epoch [   83/  100] | d_loss: 1.0009 | g_loss: 1.7361
Epoch [   83/  100] | d_loss: 1.0990 | g_loss: 1.5864
Epoch [   84/  100] | d_loss: 1.1277 | g_loss: 1.9122
Epoch [   84/  100] | d_loss: 0.8992 | g_loss: 1.5544
Epoch [   84/  100] | d_loss: 1.0471 | g_loss: 1.4275
Epoch [   85/  100] | d_loss: 1.0434 | g_loss: 1.7387
Epoch [   85/  100] | d_loss: 1.0973 | g_loss: 1.5721
Epoch [   85/  100] | d_loss: 0.9227 | g_loss: 1.7647
Epoch [   86/  100] | d_loss: 1.1445 | g_loss: 1.5704
Epoch [   86/  100] | d_loss: 0.9174 | g_loss: 1.2598
Epoch [   86/  100] | d_loss: 1.0285 | g_loss: 1.6627
Epoch [   87/  100] | d_loss: 1.0879 | g_loss: 1.7334
Epoch [   87/  100] | d_loss: 0.9875 | g_loss: 1.1223
Epoch [   87/  100] | d_loss: 1.1087 | g_loss: 1.6336
Epoch [   88/  100] | d_loss: 1.0850 | g_loss: 1.8996
Epoch [   88/  100] | d_loss: 1.0664 | g_loss: 1.5522
Epoch [   88/  100] | d_loss: 1.1495 | g_loss: 1.8332
Epoch [   89/  100] | d_loss: 1.0751 | g_loss: 1.6307
Epoch [   89/  100] | d_loss: 0.9789 | g_loss: 1.2070
Epoch [   89/  100] | d_loss: 1.1157 | g_loss: 1.8021
Epoch [   90/  100] | d_loss: 1.0267 | g_loss: 1.7396
Epoch [   90/  100] | d_loss: 0.9668 | g_loss: 1.4893
Epoch [   90/  100] | d_loss: 1.1008 | g_loss: 1.5880
Epoch [   91/  100] | d_loss: 1.2427 | g_loss: 1.7785
Epoch [   91/  100] | d_loss: 1.0793 | g_loss: 1.4255
Epoch [   91/  100] | d_loss: 0.9223 | g_loss: 1.8432
Epoch [   92/  100] | d_loss: 1.1394 | g_loss: 1.8739
Epoch [   92/  100] | d_loss: 1.0412 | g_loss: 1.4466
Epoch [   92/  100] | d_loss: 1.0406 | g_loss: 1.4127
Epoch [   93/  100] | d_loss: 1.0850 | g_loss: 1.9401
```

```
Epoch [   93/  100] | d_loss: 0.9586 | g_loss: 1.4103
Epoch [   93/  100] | d_loss: 1.2251 | g_loss: 1.3022
Epoch [   94/  100] | d_loss: 1.0675 | g_loss: 1.4363
Epoch [   94/  100] | d_loss: 0.9553 | g_loss: 1.3485
Epoch [   94/  100] | d_loss: 1.0923 | g_loss: 1.5421
Epoch [   95/  100] | d_loss: 1.0160 | g_loss: 1.9057
Epoch [   95/  100] | d_loss: 0.9935 | g_loss: 1.2164
Epoch [   95/  100] | d_loss: 1.0341 | g_loss: 1.5211
Epoch [   96/  100] | d_loss: 1.0109 | g_loss: 1.8614
Epoch [   96/  100] | d_loss: 0.9375 | g_loss: 1.5423
Epoch [   96/  100] | d_loss: 0.8618 | g_loss: 1.7875
Epoch [   97/  100] | d_loss: 1.1394 | g_loss: 1.4920
Epoch [   97/  100] | d_loss: 1.0284 | g_loss: 1.4685
Epoch [   97/  100] | d_loss: 1.0712 | g_loss: 1.7202
Epoch [   98/  100] | d_loss: 0.9911 | g_loss: 2.0666
Epoch [   98/  100] | d_loss: 1.0426 | g_loss: 1.6476
Epoch [   98/  100] | d_loss: 1.0311 | g_loss: 1.9197
Epoch [   99/  100] | d_loss: 1.0343 | g_loss: 1.7156
Epoch [   99/  100] | d_loss: 1.0471 | g_loss: 1.5421
Epoch [   99/  100] | d_loss: 1.1334 | g_loss: 1.6256
Epoch [  100/  100] | d_loss: 1.0419 | g_loss: 2.0526
Epoch [  100/  100] | d_loss: 1.0428 | g_loss: 1.6276
Epoch [  100/  100] | d_loss: 1.0954 | g_loss: 1.4285
```

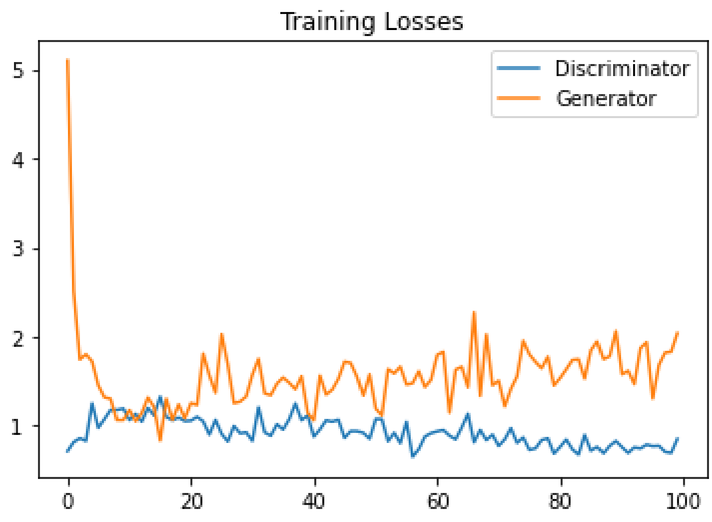
Training loss

Here we'll plot the training losses for the generator and discriminator, recorded after each epoch.

In [12]:

```python
fig, ax = plt.subplots()
losses = np.array(losses)
plt.plot(losses.T[0], label='Discriminator')
plt.plot(losses.T[1], label='Generator')
plt.title("Training Losses")
plt.legend()
```

Out[12]: <matplotlib.legend.Legend at 0x7fda9813fa50>



Generator samples from training

Here we can view samples of images from the generator. First we'll look at the images we saved during training.

In [13]:

```python
# helper function for viewing a list of passed in sample images
```

```
def view_samples(epoch, samples):
    fig, axes = plt.subplots(figsize=(7,7), nrows=4, ncols=4, sharey=True, sh
    for ax, img in zip(axes.flatten(), samples[epoch]):
        img = img.detach()
        ax.xaxis.set_visible(False)
        ax.yaxis.set_visible(False)
        im = ax.imshow(img.reshape((28,28)), cmap='Greys_r')
```

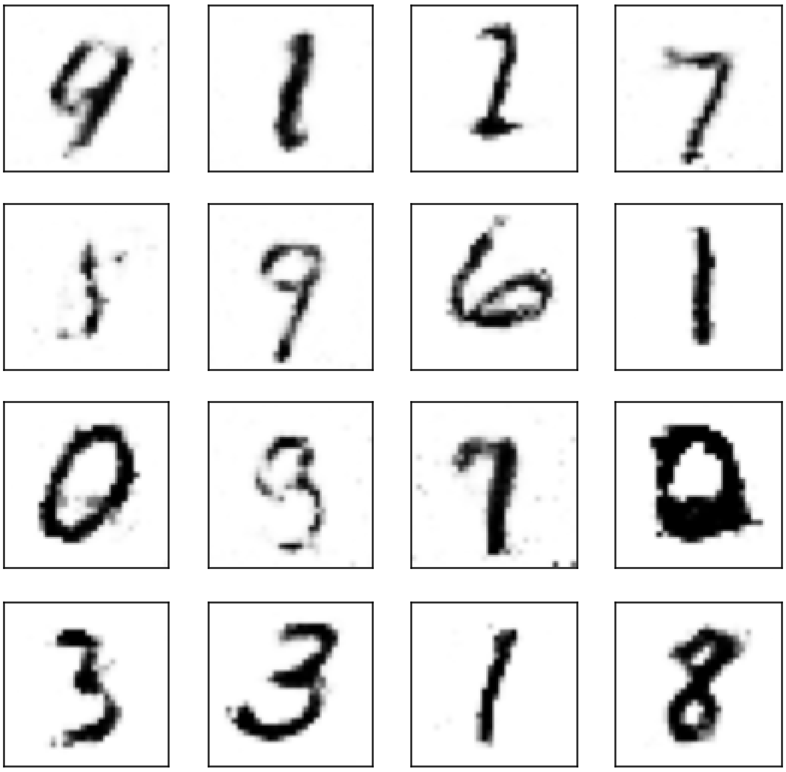
In [14]:

```
# Load samples from generator, taken while training
with open('train_samples.pkl', 'rb') as f:
    samples = pkl.load(f)
```

These are samples from the final training epoch. You can see the generator is able to reproduce numbers like 1, 7, 3, 2. Since this is just a sample, it isn't representative of the full range of images this generator can make.

In [15]:

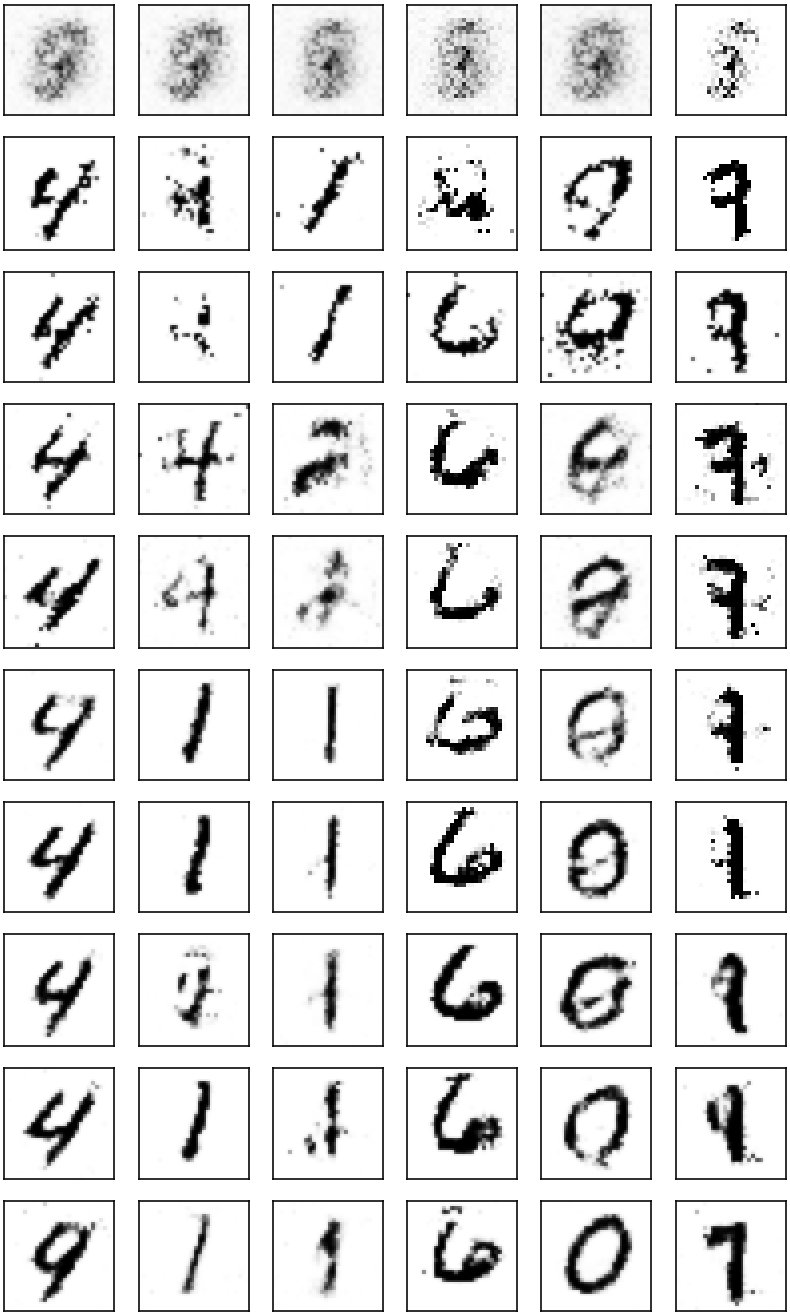
```
# -1 indicates final epoch's samples (the last in the list)
view_samples(-1, samples)
```



Below the generated images are shown, as the network was training, every 10 epochs.

In [16]:

```
rows = 10 # split epochs into 10, so 100/10 = every 10 epochs
cols = 6
fig, axes = plt.subplots(figsize=(7,12), nrows=rows, ncols=cols, sharex=True,

for sample, ax_row in zip(samples[::int(len(samples)/rows)], axes):
    for img, ax in zip(sample[::int(len(sample)/cols)], ax_row):
        img = img.detach()
        ax.imshow(img.reshape((28,28)), cmap='Greys_r')
        ax.xaxis.set_visible(False)
        ax.yaxis.set_visible(False)
```



It starts out as all noise. Then it learns to make only the center white and the rest black. You can start to see some number like structures appear out of the noise like 1s and 9s.

Sampling from the generator

We can also get completely new images from the generator by using the checkpoint we saved after training. **We just need to pass in a new latent vector z and we'll get new samples**!

In [17]:

```
# randomly generated, new latent vectors
sample_size=16
rand_z = np.random.uniform(-1, 1, size=(sample_size, z_size))
rand_z = torch.from_numpy(rand_z).float()

G.eval() # eval mode
# generated samples
rand_images = G(rand_z)

# 0 indicates the first set of samples in the passed in list
```

```
# and we only have one batch of samples, here
view_samples(0, [rand_images])
```