Lecture 8 - Generative Adversarial Networks for Synthesising New Data (Part 1)

Contents

[1]:
```
from IPython.display import Image
%matplotlib inline
```

Introducing generative adversarial networks

The overall objective of a GAN is to synthesize new data that has the same distribution as its training dataset.

Therefore, GANs, in their original form, are considered to be in the unsupervised learning category of machine learning tasks, since no labeled data is required.

The initial GAN architecture proposed was based on fully connected layers, similar to multilayer perceptron architectures, and trained to generate low-resolution MNIST-like handwritten digits.

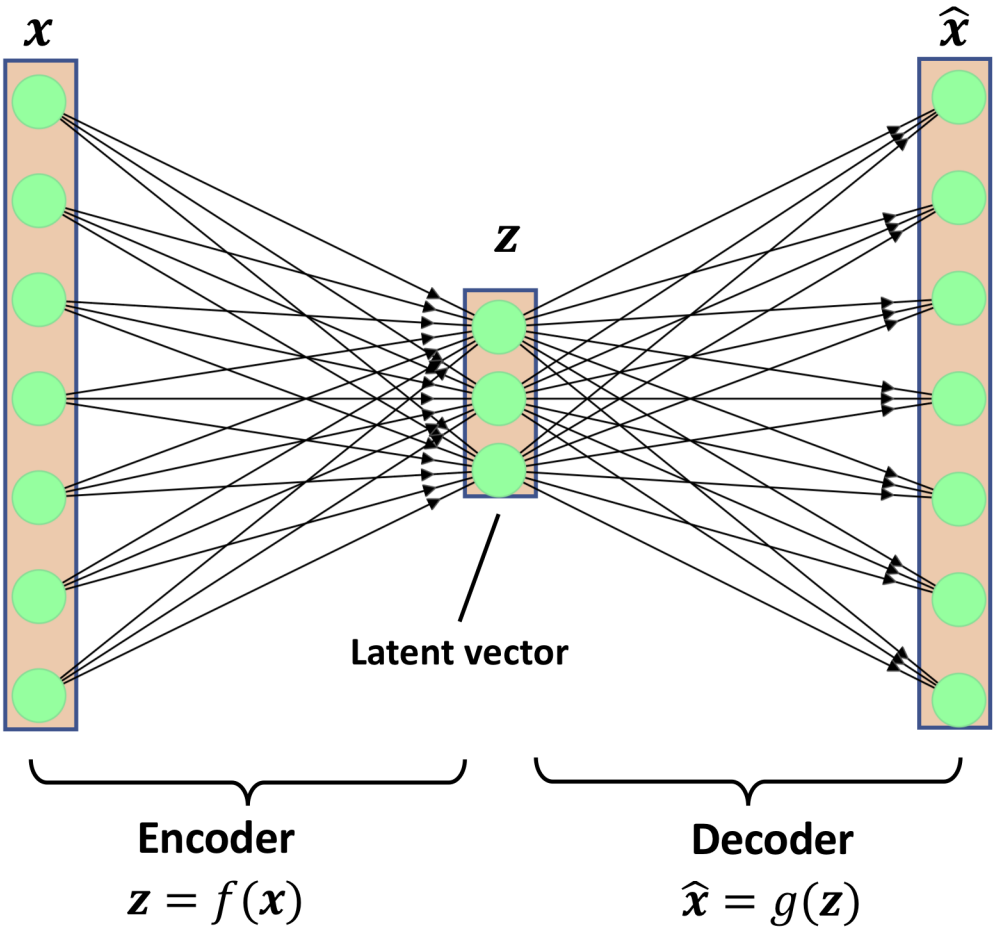
Since its introduction, numerous improvements have been proposed and various applications in different fields of engineering and science; for example, in computer vision, GANs are used for image-to-image translation (learning how to map an input image to an output image), image super-resolution (making a high-resolution image from a low-resolution version), image inpainting (learning how to reconstruct the missing parts of an image), and many more applications. For instance, we will look at using GANs for generating financial data during the lab.

Starting with autoencoders

Autoencoders can compress and decompress training data.

[2]:
```
Image(filename='figures/17_01.png', width=500)
```

[2]:



The encoder network receives a d-dimensional input feature vector associated with an example $\mathbf{x}$ ($\mathbf{x} \in \mathbb{R}^d$) and encodes it into a p-dimensional vector,$\mathbf{z}$ ($\mathbf{z} \in \mathbb{R}^d$).

The role of the encoder is to learn how to model the function $\mathbf{z} = f(\mathbf{x})$. The encoded vector, $\mathbf{z}$, is also called the **latent vector**, or the latent feature representation.

Typically, $p < d$. Hence, the encoder acts as a data compression function.

Then, the decoder decompresses $\hat{\mathbf{x}}$ from the lower- dimensional latent vector, $\mathbf{z}$, where we can think of the decoder as a function, $\hat{\mathbf{x}} = g(\mathbf{z})$.

The figure above depicts an autoencoder without hidden layers within the **encoder** and **decoder**, but we can add multiple hidden layers with nonlinearities (as in a multilayer NN) to construct a deep autoencoder that can learn more effective data compression and reconstruction functions.

Also, note that the autoencoder mentioned in this section uses fully connected layers. When we work with images, however, we can replace the fully connected layers with **convolutional layers**.

Generative models for synthesising new data

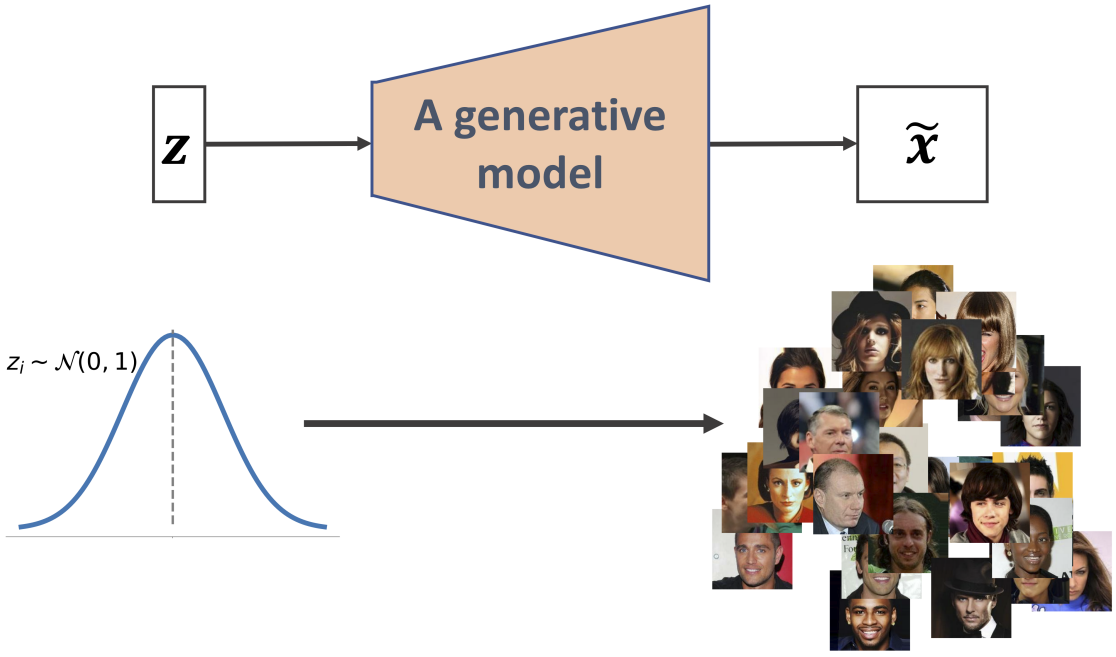
Autoencoders are deterministic models, which means that after an autoencoder is trained, given an input, $\mathbf{x}$, it will be able to reconstruct the input from its compressed version in a lower-dimensional space.

A generative model, on the other hand, can generate a new example, $\widetilde{\mathbf{x}}$, from a random vector, $\mathbf{z}$ (corresponding to the latent representation).

The random vector $\mathbf{z}$ comes from a distribution with fully known characteristics, so we can easily sample from such a distribution. For example, each element of z may come from the uniform distribution in the range $[-1, 1]$ (for which we write $z_i \sim Uniform(-1, 1)$) or from a standard normal distribution (in which case, we write $z_i \sim Normal(mean = 0, variance = 1)$).

[3]:
```
Image(filename='figures/17_02.png', width=700)
```

[3]:



The decoder component of an autoencoder has some similarities with a generative model. For the autoencoder, $\widehat{\mathbf{x}}$ is the reconstruction of an input $\mathbf{x}$, and for the generative model, $\widetilde{\mathbf{x}}$ is a synthesized sample.

The major difference between the two is that we do not know the distribution of $\mathbf{z}$ in the autoencoder, while in a generative model, the distribution of $\mathbf{z}$ is fully characterizable.

Generating new samples with GANs

Let us first assume we have a network that receives a random vector, $\mathbf{z}$, sampled from a known distribution, and generates an output image, $\mathbf{x}$. We will call this network **generator** (G) and use the notation $\widetilde{\mathbf{x}} = G(\mathbf{z})$ to refer to the generated output.

Assume our goal is to generate some images, for example, handwritten digits such as MNIST.

As always, we will initialize this network with random weights. Therefore, the first output images, before these weights are adjusted, will look like white noise.

Now, imagine there is a function that can assess the quality of images (let us call it an *assessor function*).

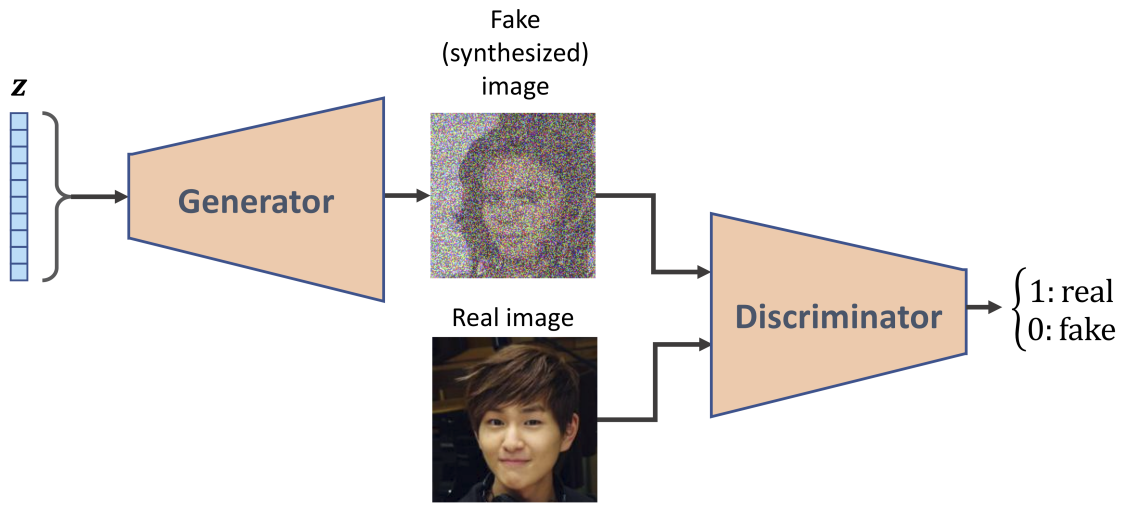
If such a function exists, we can use the feedback from that function to tell our generator network how to adjust its weights to improve the quality of the generated images. This way, we can train the generator based on the feedback from that assessor function, such that the generator learns to improve its output toward producing realistic-looking images.

How to get such an assesor function? We design a NN to assess the quality of synthesised images - that is the general idea of a GAN.

A GAN model consists of an additional NN called **discriminator** (D), which is a classifier that learns to detect a synthesized image, $\widetilde{\mathbf{x}}$, from a real image, $\mathbf{x}$.

[4]:
```
Image(filename='figures/17_03.png', width=700)
```

[4]:



Train together: In a GAN model, the two networks, generator and discriminator, are trained together.

At first, after initializing the model weights, the generator creates images that do not look realistic. Similarly, the discriminator does a poor job of distinguishing between real images and images synthesized by the generator. But over time (that is, through training), both networks become better as they interact with each other.

Adversarial game: To improve both generator and discriminator, we design an adversarial game where the generator learns to improve its output to be able to fool the discriminator. At the same time, the discriminator becomes better at detecting the synthesized images.

Understanding the loss functions for the generator and discriminator networks in a GAN model

The objective function of GANs is as follows:

$$V(\theta^{(D)}, \theta^{(G)}) = \mathbb{E}_{\mathbf{x}\sim p_{data}(\mathbf{x})}[\log D(\mathbf{x})] + \mathbb{E}_{\mathbf{z}\sim p_{(\mathbf{z})}(\mathbf{z})}\left[\log\left(1 - D\big(G(\mathbf{z})\big)\right)\right].$$

Here $V(\theta^{(D)}, \theta^{(G)})$ is called the **value function**. We want to maximize its value with respect to the discriminator (D), while minimizing its value with respect to the generator (G).

$$\min_G \max_D V(\theta^{(D)}, \theta^{(G)}).$$

- $D(\mathbf{x})$ is the probability that indicates whether the input example, $\mathbf{x}$, is real or fake (that is, generated).
- $\mathbb{E}_{\mathbf{x}\sim p_{data}(\mathbf{x})}[\log D(\mathbf{x})]$ refers to the expected value of the quantity in brackets with respect to the examples from the data distribution (distribution of the real examples).
- $\mathbb{E}_{\mathbf{z}\sim p_{(\mathbf{z})}(\mathbf{z})}\left[\log\left(1 - D\big(G(\mathbf{z})\big)\right)\right]$ refers to the expected value of the quantity with respect to the distribution of the input, $\mathbf{z}$, vectors.

One training step of a GAN model with such a value function equals to 1. maximizing the payoff for the discriminator 2. minimizing the payoff for the generator

1. fix (freeze) the parameters of one network and optimize the weights of the other one, and
2. fix the second network and optimize the first one.

Saturation: $\log\left(1 - D\big(G(\mathbf{z})\big)\right)$ suffers from vanishing gradients in the early training stages because $G(\mathbf{z})$ looks nothing like real examples early in the learning process and therefore $D\big(G(\mathbf{z})\big)$ will be close to zero with high confidence.

To resolve this, we reformulate the minimisation objective $\min_G \mathbb{E}_{\mathbf{z}\sim p_{(\mathbf{z})}(\mathbf{z})}\left[\log\left(1 - D\big(G(\mathbf{z})\big)\right)\right]$ as $\max_G \mathbb{E}_{\mathbf{z}\sim p_{(\mathbf{z})}(\mathbf{z})}\left[\log D\big(G(\mathbf{z})\big)\right]$.

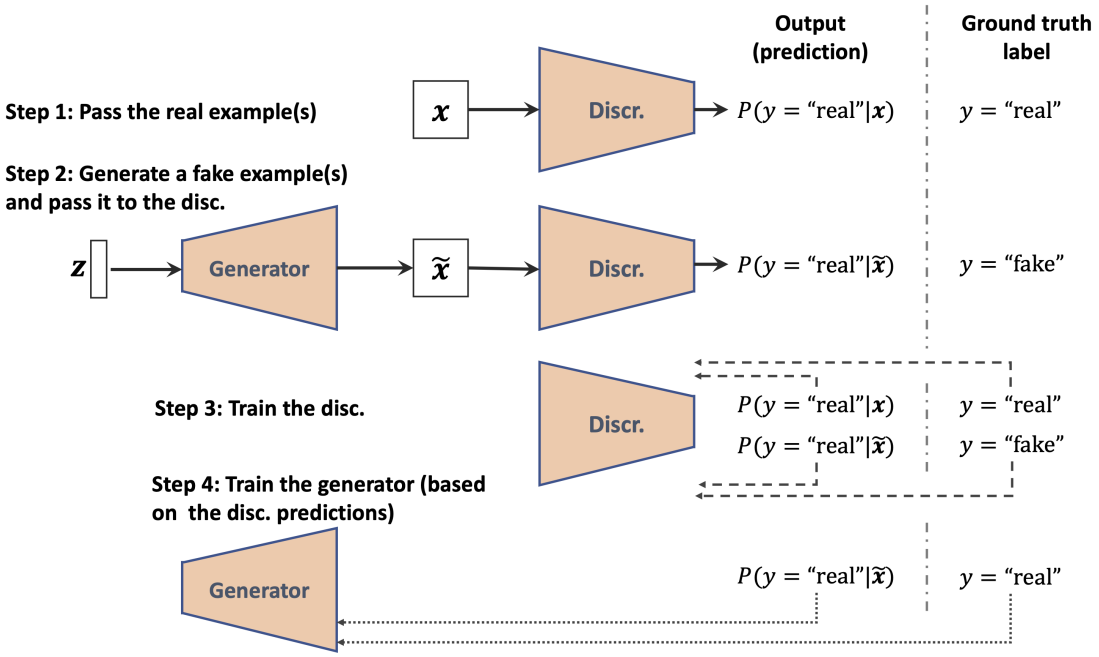
However, we can swap the labels of real and fake examples and carry out a regular function minimisation. Even though the examples synthesised by the generator are fake and are therefore labeled 0, we can flip the labels by assigning label 1 to these examples and minimise the binary cross-entropy loss with these new labels instead of $\max_G \mathbb{E}_{\mathbf{z}\sim p_{(\mathbf{z})}(\mathbf{z})}\left[\log D\big(G(\mathbf{z})\big)\right]$.

Given that the discriminator is a binary classifier - the class labels are 0 and 1 for fake and real images, respectively - we can use the binary cross-entropy loss function. Thus, we have the *ground truth for the discriminator loss* as

$$\text{Ground truth labels for the discriminator} = \begin{cases} 1 & : \text{for real images } \mathbf{x} \\ 0 & : \text{for outputs of } G, G(\mathbf{z}) \end{cases}$$

For training the generator: As we want the generator to synthesise realistic images, we want to penalise the generator when its outputs are not classified as real by the discriminator. This means that we will assume the ground truth labels for the outputs of the generator to be 1 when computing the loss function for the generator.

[5]:
```
Image(filename='figures/17_04.png', width=800)
```
[5]:



Implementing a GAN from scratch

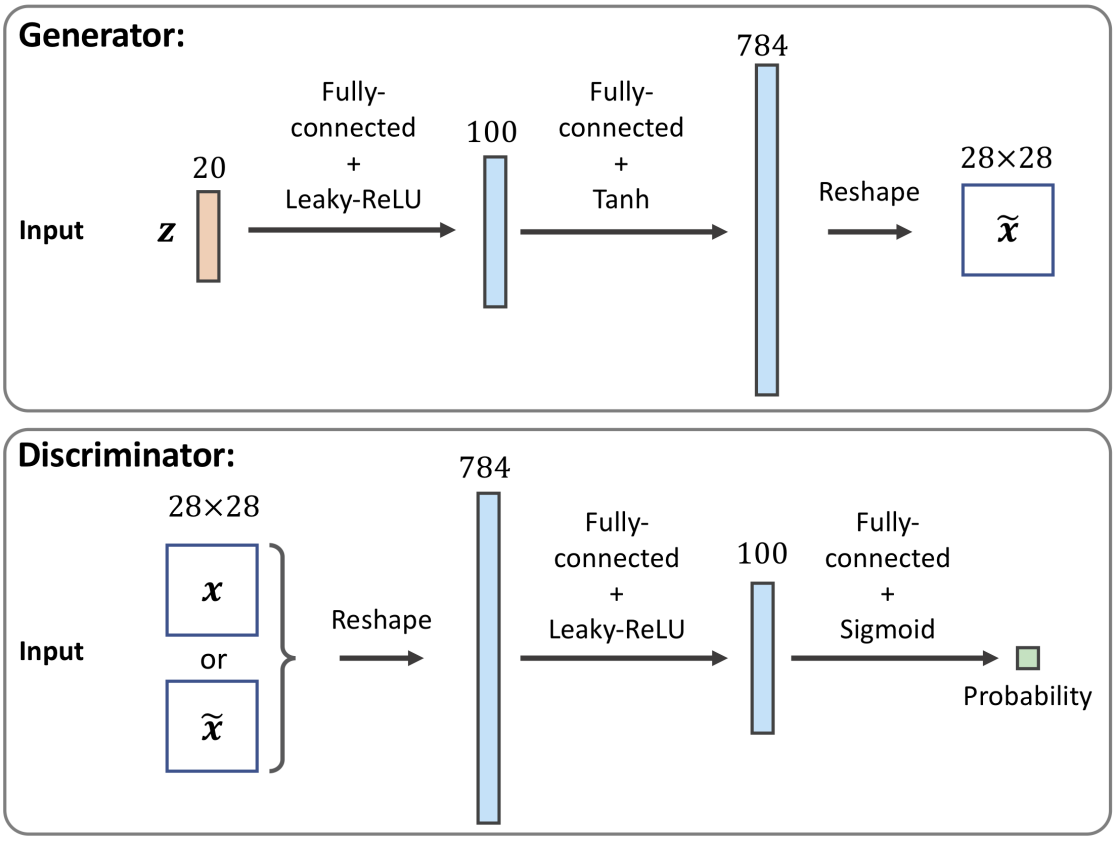
[6]:
```
import torch


print(torch.__version__)
print("GPU Available:", torch.cuda.is_available())

if torch.cuda.is_available():
    device = torch.device("cuda:0")
else:
    device = "cpu"
```

```
2.2.0+cu118
GPU Available: True
```

Implementing the generator and the discriminator networks

Architecture of the *vanilla GAN* being used here - both generator and discriminator are two fully connected networks with one or more hiddne layers.

[7]:
```
Image(filename='figures/17_08.png', width=600)
```
[7]:

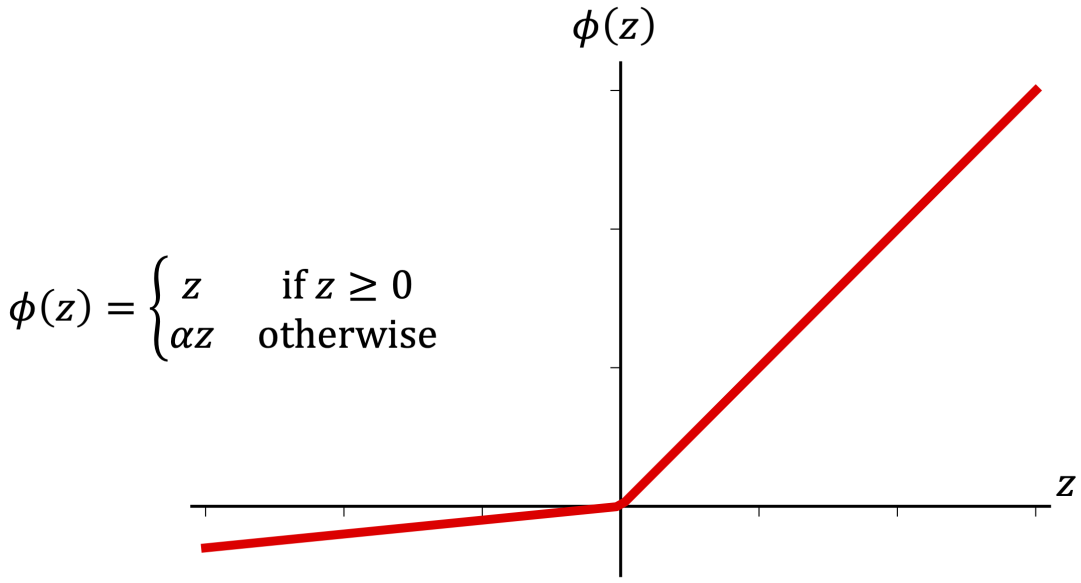


For each hidden layer, we apply the **leaky ReLU** activation function. The leaky ReLU activation function permits non-zero gradients for negative inputs as well, and as a result, it makes the networks more expressive overall.

In the discriminator network, each hidden layer is also followed by a dropout layer. Furthermore, the output layer in the generator uses the hyperbolic tangent (tanh) activation function.

[8]:
```
Image(filename='figures/17_17.png', width=600)
```

[8]:



[9]:
```
import torch.nn as nn
import numpy as np
import os

import matplotlib.pyplot as plt
%matplotlib inline
os.environ["KMP_DUPLICATE_LIB_OK"]="TRUE"
```

[29]:
```
## define a function for the generator:
def make_generator_network(
        input_size=20,
        num_hidden_layers=1,
        num_hidden_units=100,
        num_output_units=784):
    model = nn.Sequential()
    for i in range(num_hidden_layers):
        model.add_module(f'fc_g{i}',
                         nn.Linear(input_size,
                                   num_hidden_units))
        model.add_module(f'relu_g{i}',
                         nn.LeakyReLU())
        input_size = num_hidden_units
    model.add_module(f'fc_g{num_hidden_layers}',
                     nn.Linear(input_size, num_output_units))
    model.add_module('tanh_g', nn.Tanh())
    return model
```

```
## define a function for the discriminator:
def make_discriminator_network(
        input_size,
        num_hidden_layers=1,
        num_hidden_units=100,
        num_output_units=1):
    model = nn.Sequential()
    for i in range(num_hidden_layers):
        model.add_module(f'fc_d{i}',
                 nn.Linear(input_size,
                           num_hidden_units, bias=False))
        model.add_module(f'relu_d{i}',
                         nn.LeakyReLU())
        model.add_module('dropout', nn.Dropout(p=0.5))
        input_size = num_hidden_units
    model.add_module(f'fc_d{num_hidden_layers}',
                     nn.Linear(input_size, num_output_units))
    model.add_module('sigmoid', nn.Sigmoid())
    return model
```

The image size in the MNIST dataset is 28×28 pixels (grayscale images).

The size of the input vector $\mathbf{z}$ is set to be 20.

[11]:
```
image_size = (28, 28)
z_size = 20

gen_hidden_layers = 1 # vanialla GAN
gen_hidden_size = 100
disc_hidden_layers = 1 # vanilla GAN
disc_hidden_size = 100

torch.manual_seed(1)

gen_model = make_generator_network(
    input_size=z_size,
    num_hidden_layers=gen_hidden_layers,
    num_hidden_units=gen_hidden_size,
    num_output_units=np.prod(image_size))

print(gen_model)
```

```
Sequential(
  (fc_g0): Linear(in_features=20, out_features=100, bias=True)
  (relu_g0): LeakyReLU(negative_slope=0.01)
  (fc_g1): Linear(in_features=100, out_features=784, bias=True)
  (tanh_g): Tanh()
)
```

[12]:
```
disc_model = make_discriminator_network(
    input_size=np.prod(image_size),
    num_hidden_layers=disc_hidden_layers,
    num_hidden_units=disc_hidden_size)

print(disc_model)
```

```
Sequential(
  (fc_d0): Linear(in_features=784, out_features=100, bias=False)
  (relu_d0): LeakyReLU(negative_slope=0.01)
  (dropout): Dropout(p=0.5, inplace=False)
  (fc_d1): Linear(in_features=100, out_features=1, bias=True)
  (sigmoid): Sigmoid()
)
```

Defining the training dataset

`torchvision.transforms.ToTensor` converts the input image tensor to a float tensor.

Besides changing the data type, calling this function will also change the range of input pixel intensities to $[0, 1]$. Then, we can shift them by $\breve{}0.5$ and scale them by a factor of 0.5 such that the pixel intensities will be rescaled to be in the range $[\breve{}1, 1]$.

Note that the output layer of the generator uses `tanh` activation function, which would give the range as $[-1, 1]$.

[13]:
```
import torchvision
from torchvision import transforms


image_path = './'
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize(mean=(0.5), std=(0.5)),
])
mnist_dataset = torchvision.datasets.MNIST(root=image_path,
                                           train=True,
                                           transform=transform,
                                           download=False)

example, label = next(iter(mnist_dataset))
print(f'Min: {example.min()} Max: {example.max()}')
print(example.shape)
```

```
Min: -1.0 Max: 1.0
torch.Size([1, 28, 28])
```

Next create a random vector **z** based on the desired random distribution.

[14]:
```
def create_noise(batch_size, z_size, mode_z):
    if mode_z == 'uniform':
        input_z = torch.rand(batch_size, z_size)*2 - 1
    elif mode_z == 'normal':
        input_z = torch.randn(batch_size, z_size)
    return input_z
```

Let us print array shapes of one batch of examples (input vectors and images). Also, let us process a forward pass of our generator and discriminator.

[15]:
```
from torch.utils.data import DataLoader


batch_size = 32
dataloader = DataLoader(mnist_dataset, batch_size, shuffle=False)
input_real, label = next(iter(dataloader))
input_real = input_real.view(batch_size, -1)

torch.manual_seed(1)
mode_z = 'uniform'  # 'uniform' vs. 'normal'
input_z = create_noise(batch_size, z_size, mode_z)

print('input-z -- shape:', input_z.shape)
print('input-real -- shape:', input_real.shape)

g_output = gen_model(input_z)
print('Output of G -- shape:', g_output.shape)

d_proba_real = disc_model(input_real)
d_proba_fake = disc_model(g_output)
print('Disc. (real) -- shape:', d_proba_real.shape)
print('Disc. (fake) -- shape:', d_proba_fake.shape)
```

```
input-z -- shape: torch.Size([32, 20])
input-real -- shape: torch.Size([32, 784])
Output of G -- shape: torch.Size([32, 784])
Disc. (real) -- shape: torch.Size([32, 1])
Disc. (fake) -- shape: torch.Size([32, 1])
```

The two probabilities, `d_proba_fake` and `d_proba_real`, are used to compute the loss functions for training the model.

Training the GAN model

To compute the loss, we need the ground truth labels for each output.

For the generator: we create a vector of 1s with the same shape as the vector containing the predicted probabilities for the generated images, `d_proba_fake`.

For the discriminator loss: we have two terms: the loss for detecting the fake examples involving

`d_proba_fake` and the loss for detecting the real examples based on `d_proba_real`.

The ground truth labels for the fake images is a vector of 0s.

The ground truth values for the real images is a vector of 1s.

[16]:
```
loss_fn = nn.BCELoss()

## Loss for the Generator
g_labels_real = torch.ones_like(d_proba_fake)
g_loss = loss_fn(d_proba_fake, g_labels_real)
print(f'Generator Loss: {g_loss:.4f}')

## Loss for the Discriminator
d_labels_real = torch.ones_like(d_proba_real)
d_labels_fake = torch.zeros_like(d_proba_fake)

d_loss_real = loss_fn(d_proba_real, d_labels_real)
d_loss_fake = loss_fn(d_proba_fake, d_labels_fake)
print(f'Discriminator Losses: Real {d_loss_real:.4f} Fake {d_loss_fake:.4f}')
```

```
Generator Loss: 0.6983
Discriminator Losses: Real 0.7479 Fake 0.6885
```

The following code sets up the GAN model and implement the training loop. We have a separate Adam optimizer for each of the two models - generator and discriminator.

[17]:
```
batch_size = 64

torch.manual_seed(1)
np.random.seed(1)

## Set up the dataset
mnist_dl = DataLoader(mnist_dataset, batch_size=batch_size,
                      shuffle=True, drop_last=True)

## Set up the models
gen_model = make_generator_network(
    input_size=z_size,
    num_hidden_layers=gen_hidden_layers,
    num_hidden_units=gen_hidden_size,
    num_output_units=np.prod(image_size)).to(device)

disc_model = make_discriminator_network(
    input_size=np.prod(image_size),
    num_hidden_layers=disc_hidden_layers,
    num_hidden_units=disc_hidden_size).to(device)

## Loss function and optimizers:
```

```
loss_fn = nn.BCELoss()
g_optimizer = torch.optim.Adam(gen_model.parameters())
d_optimizer = torch.optim.Adam(disc_model.parameters())
```

[18]:
```
## Train the discriminator
def d_train(x):
    disc_model.zero_grad()

    # Train discriminator with a real batch
    batch_size = x.size(0)
    x = x.view(batch_size, -1).to(device)
    d_labels_real = torch.ones(batch_size, 1, device=device)

    d_proba_real = disc_model(x)
    d_loss_real = loss_fn(d_proba_real, d_labels_real)

    # Train discriminator on a fake batch
    input_z = create_noise(batch_size, z_size, mode_z).to(device)
    g_output = gen_model(input_z)

    d_proba_fake = disc_model(g_output)
    d_labels_fake = torch.zeros(batch_size, 1, device=device)
    d_loss_fake = loss_fn(d_proba_fake, d_labels_fake)

    # gradient backprop & optimize ONLY D's parameters
    d_loss = d_loss_real + d_loss_fake
    d_loss.backward()
    d_optimizer.step()

    return d_loss.data.item(), d_proba_real.detach(), d_proba_fake.detach()
```

[19]:
```
## Train the generator
def g_train(x):
    gen_model.zero_grad()

    batch_size = x.size(0)
    input_z = create_noise(batch_size, z_size, mode_z).to(device)
    g_labels_real = torch.ones(batch_size, 1, device=device)

    g_output = gen_model(input_z)
    d_proba_fake = disc_model(g_output)
    g_loss = loss_fn(d_proba_fake, g_labels_real)

    # gradient backprop & optimize ONLY G's parameters
    g_loss.backward()
    g_optimizer.step()
```

```
    return g_loss.data.item()
```

[20]:
```
fixed_z = create_noise(batch_size, z_size, mode_z).to(device)

def create_samples(g_model, input_z):
    g_output = g_model(input_z)
    images = torch.reshape(g_output, (batch_size, *image_size))
    return (images+1)/2.0

epoch_samples = []

all_d_losses = []
all_g_losses = []

all_d_real = []
all_d_fake = []

num_epochs = 100
torch.manual_seed(1)
for epoch in range(1, num_epochs+1):
    d_losses, g_losses = [], []
    d_vals_real, d_vals_fake = [], []
    for i, (x, _) in enumerate(mnist_dl):
        d_loss, d_proba_real, d_proba_fake = d_train(x)
        d_losses.append(d_loss)
        g_losses.append(g_train(x))

        d_vals_real.append(d_proba_real.mean().cpu())
        d_vals_fake.append(d_proba_fake.mean().cpu())

    all_d_losses.append(torch.tensor(d_losses).mean())
    all_g_losses.append(torch.tensor(g_losses).mean())
    all_d_real.append(torch.tensor(d_vals_real).mean())
    all_d_fake.append(torch.tensor(d_vals_fake).mean())
    print(f'Epoch {epoch:03d} | Avg Losses >>'
          f' G/D {all_g_losses[-1]:.4f}/{all_d_losses[-1]:.4f}'
          f' [D-Real: {all_d_real[-1]:.4f} D-Fake: {all_d_fake[-1]:.4f}]')
    epoch_samples.append(create_samples(gen_model, fixed_z).detach().cpu().
 ↪numpy())
```

```
Epoch 001 | Avg Losses >> G/D 0.9110/0.8838 [D-Real: 0.8127 D-Fake: 0.4652]
Epoch 002 | Avg Losses >> G/D 1.0143/1.0904 [D-Real: 0.6329 D-Fake: 0.4148]
Epoch 003 | Avg Losses >> G/D 0.9330/1.2136 [D-Real: 0.5761 D-Fake: 0.4327]
Epoch 004 | Avg Losses >> G/D 0.9672/1.2009 [D-Real: 0.5798 D-Fake: 0.4236]
Epoch 005 | Avg Losses >> G/D 0.8637/1.2685 [D-Real: 0.5532 D-Fake: 0.4453]
Epoch 006 | Avg Losses >> G/D 0.9244/1.2555 [D-Real: 0.5559 D-Fake: 0.4349]
Epoch 007 | Avg Losses >> G/D 1.0276/1.1821 [D-Real: 0.5883 D-Fake: 0.4101]
```

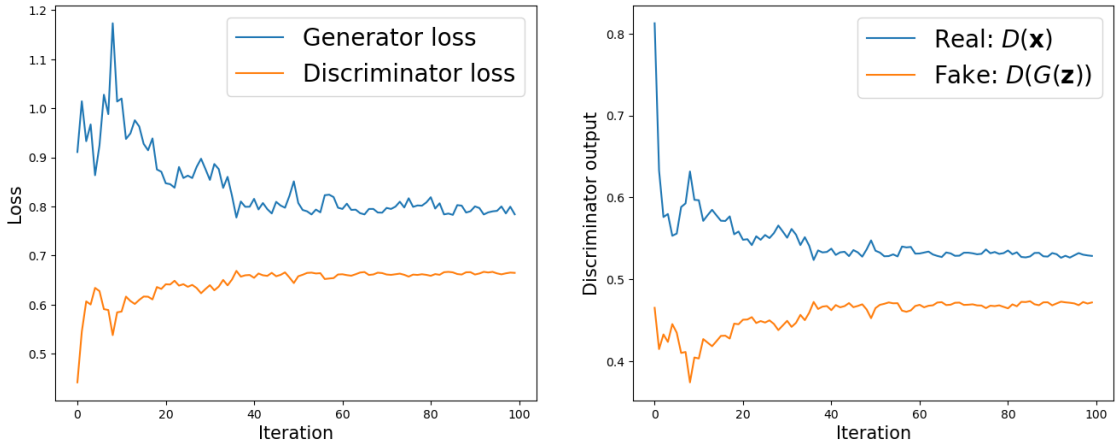
```
Epoch 008 | Avg Losses >> G/D 0.9881/1.1784 [D-Real: 0.5927 D-Fake: 0.4113]
Epoch 009 | Avg Losses >> G/D 1.1731/1.0762 [D-Real: 0.6317 D-Fake: 0.3741]
Epoch 010 | Avg Losses >> G/D 1.0138/1.1690 [D-Real: 0.5970 D-Fake: 0.4045]
Epoch 011 | Avg Losses >> G/D 1.0198/1.1722 [D-Real: 0.5966 D-Fake: 0.4033]
Epoch 012 | Avg Losses >> G/D 0.9376/1.2331 [D-Real: 0.5713 D-Fake: 0.4270]
Epoch 013 | Avg Losses >> G/D 0.9488/1.2151 [D-Real: 0.5784 D-Fake: 0.4226]
Epoch 014 | Avg Losses >> G/D 0.9757/1.2031 [D-Real: 0.5849 D-Fake: 0.4183]
Epoch 015 | Avg Losses >> G/D 0.9630/1.2196 [D-Real: 0.5781 D-Fake: 0.4244]
Epoch 016 | Avg Losses >> G/D 0.9276/1.2333 [D-Real: 0.5715 D-Fake: 0.4309]
Epoch 017 | Avg Losses >> G/D 0.9145/1.2327 [D-Real: 0.5713 D-Fake: 0.4311]
Epoch 018 | Avg Losses >> G/D 0.9386/1.2216 [D-Real: 0.5769 D-Fake: 0.4276]
Epoch 019 | Avg Losses >> G/D 0.8755/1.2721 [D-Real: 0.5550 D-Fake: 0.4459]
Epoch 020 | Avg Losses >> G/D 0.8708/1.2644 [D-Real: 0.5584 D-Fake: 0.4451]
Epoch 021 | Avg Losses >> G/D 0.8473/1.2833 [D-Real: 0.5483 D-Fake: 0.4507]
Epoch 022 | Avg Losses >> G/D 0.8455/1.2824 [D-Real: 0.5490 D-Fake: 0.4509]
Epoch 023 | Avg Losses >> G/D 0.8383/1.2976 [D-Real: 0.5418 D-Fake: 0.4538]
Epoch 024 | Avg Losses >> G/D 0.8804/1.2779 [D-Real: 0.5525 D-Fake: 0.4466]
Epoch 025 | Avg Losses >> G/D 0.8581/1.2837 [D-Real: 0.5483 D-Fake: 0.4490]
Epoch 026 | Avg Losses >> G/D 0.8629/1.2731 [D-Real: 0.5541 D-Fake: 0.4472]
Epoch 027 | Avg Losses >> G/D 0.8580/1.2807 [D-Real: 0.5504 D-Fake: 0.4499]
Epoch 028 | Avg Losses >> G/D 0.8807/1.2680 [D-Real: 0.5564 D-Fake: 0.4455]
Epoch 029 | Avg Losses >> G/D 0.8973/1.2468 [D-Real: 0.5657 D-Fake: 0.4380]
Epoch 030 | Avg Losses >> G/D 0.8761/1.2641 [D-Real: 0.5584 D-Fake: 0.4436]
Epoch 031 | Avg Losses >> G/D 0.8542/1.2796 [D-Real: 0.5508 D-Fake: 0.4493]
Epoch 032 | Avg Losses >> G/D 0.8865/1.2588 [D-Real: 0.5615 D-Fake: 0.4419]
Epoch 033 | Avg Losses >> G/D 0.8762/1.2738 [D-Real: 0.5546 D-Fake: 0.4468]
Epoch 034 | Avg Losses >> G/D 0.8379/1.3010 [D-Real: 0.5419 D-Fake: 0.4566]
Epoch 035 | Avg Losses >> G/D 0.8604/1.2788 [D-Real: 0.5514 D-Fake: 0.4501]
Epoch 036 | Avg Losses >> G/D 0.8219/1.3026 [D-Real: 0.5409 D-Fake: 0.4593]
Epoch 037 | Avg Losses >> G/D 0.7776/1.3378 [D-Real: 0.5236 D-Fake: 0.4724]
Epoch 038 | Avg Losses >> G/D 0.8102/1.3145 [D-Real: 0.5353 D-Fake: 0.4638]
Epoch 039 | Avg Losses >> G/D 0.7994/1.3198 [D-Real: 0.5329 D-Fake: 0.4667]
Epoch 040 | Avg Losses >> G/D 0.7996/1.3211 [D-Real: 0.5334 D-Fake: 0.4673]
Epoch 041 | Avg Losses >> G/D 0.8156/1.3096 [D-Real: 0.5374 D-Fake: 0.4623]
Epoch 042 | Avg Losses >> G/D 0.7940/1.3272 [D-Real: 0.5298 D-Fake: 0.4685]
Epoch 043 | Avg Losses >> G/D 0.8071/1.3200 [D-Real: 0.5329 D-Fake: 0.4658]
Epoch 044 | Avg Losses >> G/D 0.7946/1.3175 [D-Real: 0.5336 D-Fake: 0.4673]
Epoch 045 | Avg Losses >> G/D 0.7859/1.3299 [D-Real: 0.5283 D-Fake: 0.4710]
Epoch 046 | Avg Losses >> G/D 0.8095/1.3160 [D-Real: 0.5357 D-Fake: 0.4660]
Epoch 047 | Avg Losses >> G/D 0.8020/1.3216 [D-Real: 0.5329 D-Fake: 0.4675]
Epoch 048 | Avg Losses >> G/D 0.7976/1.3318 [D-Real: 0.5277 D-Fake: 0.4695]
Epoch 049 | Avg Losses >> G/D 0.8215/1.3111 [D-Real: 0.5366 D-Fake: 0.4632]
Epoch 050 | Avg Losses >> G/D 0.8512/1.2884 [D-Real: 0.5475 D-Fake: 0.4525]
Epoch 051 | Avg Losses >> G/D 0.8072/1.3158 [D-Real: 0.5349 D-Fake: 0.4648]
Epoch 052 | Avg Losses >> G/D 0.7929/1.3220 [D-Real: 0.5326 D-Fake: 0.4688]
Epoch 053 | Avg Losses >> G/D 0.7902/1.3295 [D-Real: 0.5281 D-Fake: 0.4701]
Epoch 054 | Avg Losses >> G/D 0.7839/1.3307 [D-Real: 0.5284 D-Fake: 0.4718]
Epoch 055 | Avg Losses >> G/D 0.7938/1.3272 [D-Real: 0.5305 D-Fake: 0.4707]
```

```
Epoch 056 | Avg Losses >> G/D 0.7881/1.3285 [D-Real: 0.5280 D-Fake: 0.4708]
Epoch 057 | Avg Losses >> G/D 0.8230/1.3048 [D-Real: 0.5400 D-Fake: 0.4618]
Epoch 058 | Avg Losses >> G/D 0.8241/1.3068 [D-Real: 0.5390 D-Fake: 0.4603]
Epoch 059 | Avg Losses >> G/D 0.8192/1.3085 [D-Real: 0.5396 D-Fake: 0.4621]
Epoch 060 | Avg Losses >> G/D 0.7976/1.3232 [D-Real: 0.5314 D-Fake: 0.4674]
Epoch 061 | Avg Losses >> G/D 0.7950/1.3241 [D-Real: 0.5315 D-Fake: 0.4688]
Epoch 062 | Avg Losses >> G/D 0.8058/1.3206 [D-Real: 0.5327 D-Fake: 0.4658]
Epoch 063 | Avg Losses >> G/D 0.7931/1.3181 [D-Real: 0.5339 D-Fake: 0.4677]
Epoch 064 | Avg Losses >> G/D 0.7936/1.3243 [D-Real: 0.5303 D-Fake: 0.4682]
Epoch 065 | Avg Losses >> G/D 0.7865/1.3311 [D-Real: 0.5286 D-Fake: 0.4716]
Epoch 066 | Avg Losses >> G/D 0.7839/1.3331 [D-Real: 0.5272 D-Fake: 0.4721]
Epoch 067 | Avg Losses >> G/D 0.7950/1.3213 [D-Real: 0.5328 D-Fake: 0.4685]
Epoch 068 | Avg Losses >> G/D 0.7951/1.3233 [D-Real: 0.5317 D-Fake: 0.4690]
Epoch 069 | Avg Losses >> G/D 0.7878/1.3305 [D-Real: 0.5287 D-Fake: 0.4713]
Epoch 070 | Avg Losses >> G/D 0.7875/1.3289 [D-Real: 0.5289 D-Fake: 0.4716]
Epoch 071 | Avg Losses >> G/D 0.7971/1.3231 [D-Real: 0.5325 D-Fake: 0.4687]
Epoch 072 | Avg Losses >> G/D 0.7950/1.3216 [D-Real: 0.5325 D-Fake: 0.4690]
Epoch 073 | Avg Losses >> G/D 0.7997/1.3241 [D-Real: 0.5318 D-Fake: 0.4695]
Epoch 074 | Avg Losses >> G/D 0.8098/1.3266 [D-Real: 0.5306 D-Fake: 0.4681]
Epoch 075 | Avg Losses >> G/D 0.7980/1.3219 [D-Real: 0.5312 D-Fake: 0.4680]
Epoch 076 | Avg Losses >> G/D 0.8168/1.3146 [D-Real: 0.5365 D-Fake: 0.4649]
Epoch 077 | Avg Losses >> G/D 0.7993/1.3227 [D-Real: 0.5319 D-Fake: 0.4678]
Epoch 078 | Avg Losses >> G/D 0.8022/1.3215 [D-Real: 0.5333 D-Fake: 0.4673]
Epoch 079 | Avg Losses >> G/D 0.8017/1.3242 [D-Real: 0.5311 D-Fake: 0.4682]
Epoch 080 | Avg Losses >> G/D 0.8083/1.3217 [D-Real: 0.5320 D-Fake: 0.4663]
Epoch 081 | Avg Losses >> G/D 0.8189/1.3177 [D-Real: 0.5352 D-Fake: 0.4646]
Epoch 082 | Avg Losses >> G/D 0.7958/1.3248 [D-Real: 0.5306 D-Fake: 0.4694]
Epoch 083 | Avg Losses >> G/D 0.8065/1.3219 [D-Real: 0.5330 D-Fake: 0.4671]
Epoch 084 | Avg Losses >> G/D 0.7839/1.3325 [D-Real: 0.5273 D-Fake: 0.4725]
Epoch 085 | Avg Losses >> G/D 0.7857/1.3339 [D-Real: 0.5268 D-Fake: 0.4722]
Epoch 086 | Avg Losses >> G/D 0.7830/1.3318 [D-Real: 0.5279 D-Fake: 0.4733]
Epoch 087 | Avg Losses >> G/D 0.8028/1.3250 [D-Real: 0.5322 D-Fake: 0.4695]
Epoch 088 | Avg Losses >> G/D 0.8018/1.3231 [D-Real: 0.5323 D-Fake: 0.4684]
Epoch 089 | Avg Losses >> G/D 0.7877/1.3322 [D-Real: 0.5281 D-Fake: 0.4719]
Epoch 090 | Avg Losses >> G/D 0.7904/1.3326 [D-Real: 0.5278 D-Fake: 0.4720]
Epoch 091 | Avg Losses >> G/D 0.8002/1.3231 [D-Real: 0.5321 D-Fake: 0.4682]
Epoch 092 | Avg Losses >> G/D 0.7969/1.3271 [D-Real: 0.5307 D-Fake: 0.4705]
Epoch 093 | Avg Losses >> G/D 0.7838/1.3340 [D-Real: 0.5263 D-Fake: 0.4727]
Epoch 094 | Avg Losses >> G/D 0.7880/1.3313 [D-Real: 0.5287 D-Fake: 0.4719]
Epoch 095 | Avg Losses >> G/D 0.7901/1.3338 [D-Real: 0.5265 D-Fake: 0.4713]
Epoch 096 | Avg Losses >> G/D 0.7911/1.3276 [D-Real: 0.5293 D-Fake: 0.4706]
Epoch 097 | Avg Losses >> G/D 0.8001/1.3238 [D-Real: 0.5320 D-Fake: 0.4685]
Epoch 098 | Avg Losses >> G/D 0.7860/1.3278 [D-Real: 0.5301 D-Fake: 0.4723]
Epoch 099 | Avg Losses >> G/D 0.7996/1.3309 [D-Real: 0.5292 D-Fake: 0.4704]
Epoch 100 | Avg Losses >> G/D 0.7841/1.3298 [D-Real: 0.5286 D-Fake: 0.4717]
```

We print the average probabilities of the batches of real and fake examples as computed by the discriminator in each iteration. We expect these probabilities to be around 0.5, which means that

the discriminator is not able to confidently distinguish between real and fake images.

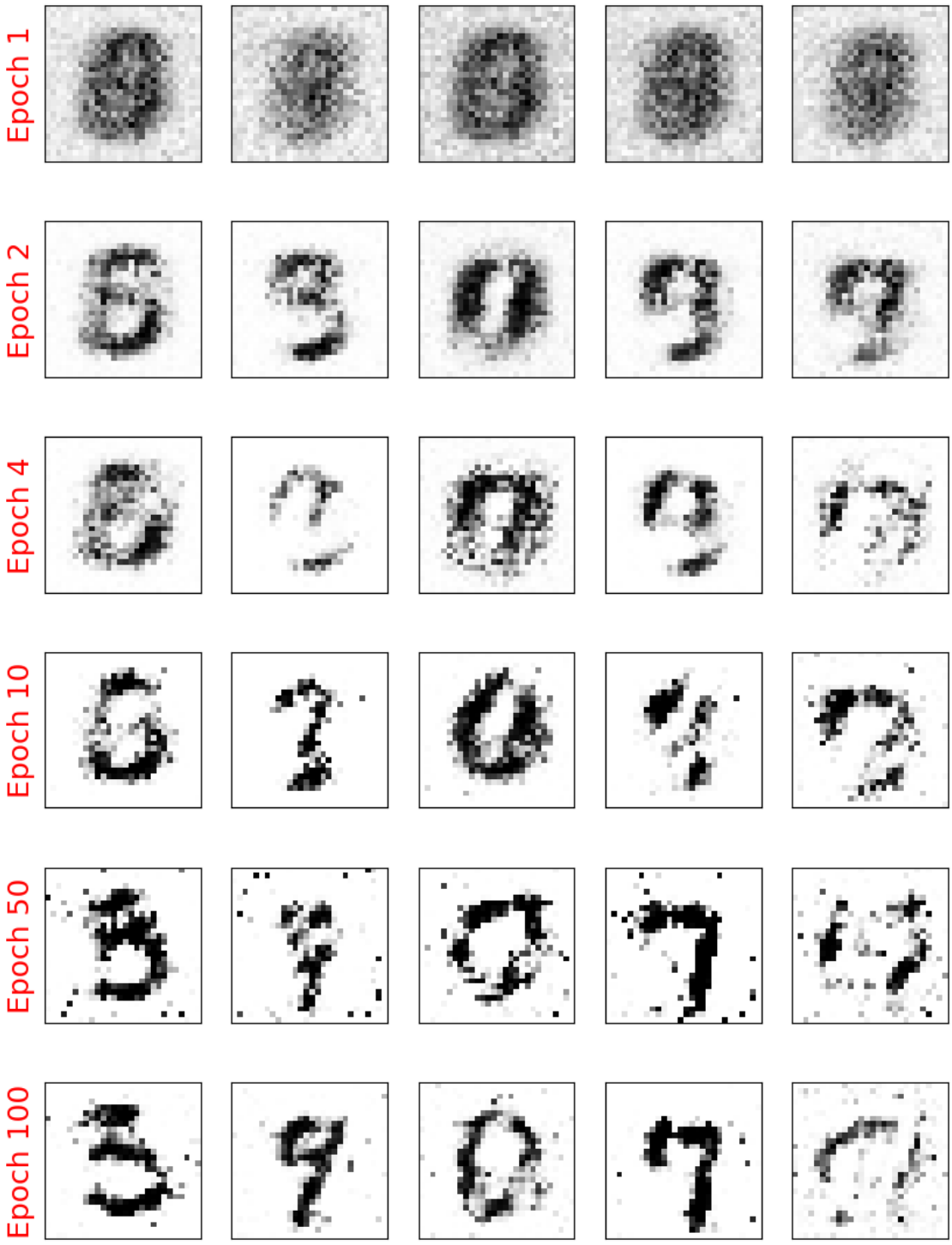
[21]:
```
import itertools


fig = plt.figure(figsize=(16, 6))

## Plotting the losses
ax = fig.add_subplot(1, 2, 1)

plt.plot(all_g_losses, label='Generator loss')
half_d_losses = [all_d_loss/2 for all_d_loss in all_d_losses]
plt.plot(half_d_losses, label='Discriminator loss')
plt.legend(fontsize=20)
ax.set_xlabel('Iteration', size=15)
ax.set_ylabel('Loss', size=15)

## Plotting the outputs of the discriminator
ax = fig.add_subplot(1, 2, 2)
plt.plot(all_d_real, label=r'Real: $D(\mathbf{x})$')
plt.plot(all_d_fake, label=r'Fake: $D(G(\mathbf{z}))$')
plt.legend(fontsize=20)
ax.set_xlabel('Iteration', size=15)
ax.set_ylabel('Discriminator output', size=15)

#plt.savefig('figures/ch17-gan-learning-curve.pdf')
plt.show()
```



During the early stages of the training, the discriminator was able to quickly learn to distinguish quite accurately between the real and fake examples.

As the training proceeds further, the generator will become better at synthesising realistic images, which will result in probabilities of both real and fake examples that are close to 0.5.

[22]:
```
selected_epochs = [1, 2, 4, 10, 50, 100]
fig = plt.figure(figsize=(10, 14))
for i,e in enumerate(selected_epochs):
    for j in range(5):
        ax = fig.add_subplot(6, 5, i*5+j+1)
        ax.set_xticks([])
        ax.set_yticks([])
        if j == 0:
            ax.text(
                -0.06, 0.5, f'Epoch {e}',
                rotation=90, size=18, color='red',
                horizontalalignment='right',
                verticalalignment='center',
                transform=ax.transAxes)

        image = epoch_samples[e-1][j]
        ax.imshow(image, cmap='gray_r')

#plt.savefig('figures/ch17-vanila-gan-samples.pdf')
plt.show()
```



Even after 100 epochs, the produced images look very different from the handwritten digits contained in the MNIST dataset.

After training the GAN model on the MNIST dataset, we were able to achieve promising, although

not yet satisfactory, results with the new handwritten digits.

In the second part, we implement a **deep convolutional GAN (DCGAN)**, which uses convolutional layers for both the generator and the discriminator networks. *Convolutional layers* have several advantages over fully connected layers when it comes to image classification. We have not covered *convolutional neural networks (CNNs)* in this course so we will use them as a black-box for creating DCGAN. There is an additional technique **Wasserstein GAN (WGAN)** which gives fantastic results.