17

Generative Adversarial Networks for Synthesizing New Data

In the previous chapter, we focused on **recurrent neural networks** for modeling sequences. In this chapter, we will explore **generative adversarial networks** (**GANs**) and see their application in synthesizing new data samples. GANs are considered to be one of the most important breakthroughs in deep learning, allowing computers to generate new data (such as new images).

In this chapter, we will cover the following topics:

- Introducing generative models for synthesizing new data
- Autoencoders, variational autoencoders, and their relationship to GANs
- Understanding the building blocks of GANs
- Implementing a simple GAN model to generate handwritten digits
- Understanding transposed convolution and batch normalization
- Improving GANs: deep convolutional GANs and GANs using the Wasserstein distance

Introducing generative adversarial networks

Let's first look at the foundations of GAN models. 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. It is worth noting, however, that extensions made to the original GAN can lie in both the semi-supervised and supervised domains.

The general GAN concept was first proposed in 2014 by Ian Goodfellow and his colleagues as a method for synthesizing new images using deep **neural networks** (**NNs**) (*Generative Adversarial Nets*, in *Advances in Neural Information Processing Systems* by *I. Goodfellow*, *J. Pouget-Abadie*, *M. Mirza*, *B. Xu*, *D. Warde-Farley*, *S. Ozair*, *A. Courville*, and *Y. Bengio*, pp. 2672-2680, 2014). While the initial GAN architecture proposed in this paper was based on fully connected layers, similar to multilayer perceptron architectures, and trained to generate low-resolution MNIST-like handwritten digits, it served more as a proof of concept to demonstrate the feasibility of this new approach.

However, since its introduction, the original authors, as well as many other researchers, have proposed numerous improvements 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, recent advances in GAN research have led to models that are able to generate new, high-resolution face images. Examples of such high-resolution images can be found on `https://www.thispersondoesnotexist.com/`, which showcases synthetic face images generated by a GAN.

Starting with autoencoders

Before we discuss how GANs work, we will first start with autoencoders, which can compress and decompress training data. While standard autoencoders cannot generate new data, understanding their function will help you to navigate GANs in the next section.

Autoencoders are composed of two networks concatenated together: an **encoder** network and a **decoder** network. The encoder network receives a d-dimensional input feature vector associated with example $\boldsymbol{x}$ (that is, $\boldsymbol{x} \in R^d$) and encodes it into a p-dimensional vector, $\boldsymbol{z}$ (that is, $\boldsymbol{z} \in R^p$). In other words, the role of the encoder is to learn how to model the function $\boldsymbol{z} = f(\boldsymbol{x})$. The encoded vector, $\boldsymbol{z}$, is also called the **latent vector**, or the latent feature representation. Typically, the dimensionality of the latent vector is less than that of the input examples; in other words, $p < d$. Hence, we can say that the encoder acts as a data compression function. Then, the decoder decompresses $\hat{\boldsymbol{x}}$ from the lower-dimensional latent vector, $\boldsymbol{z}$, where we can think of the decoder as a function, $\hat{\boldsymbol{x}} = g(\boldsymbol{z})$. A simple autoencoder architecture is shown in *Figure 17.1*, where the encoder and decoder parts consist of only one fully connected layer each:

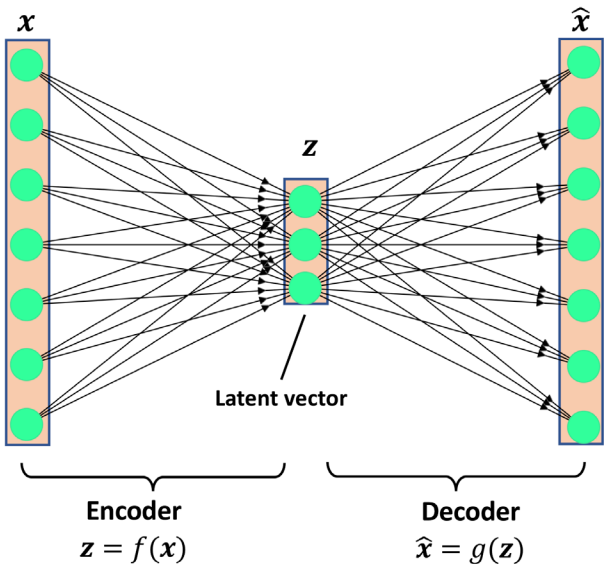

Figure 17.1: The architecture of an autoencoder

The connection between autoencoders and dimensionality reduction

In *Chapter 5*, *Compressing Data via Dimensionality Reduction*, you learned about dimensionality reduction techniques, such as **principal component analysis** (**PCA**) and **linear discriminant analysis** (**LDA**). Autoencoders can be used as a dimensionality reduction technique as well. In fact, when there is no nonlinearity in either of the two subnetworks (encoder and decoder), then the autoencoder approach is *almost identical* to PCA.

In this case, if we assume the weights of a single-layer encoder (no hidden layer and no nonlinear activation function) are denoted by the matrix U, then the encoder models $\boldsymbol{z} = \boldsymbol{U}^T\boldsymbol{x}$. Similarly, a single-layer linear decoder models $\hat{\boldsymbol{x}} = \boldsymbol{U}\boldsymbol{z}$. Putting these two components together, we have $\hat{\boldsymbol{x}} = \boldsymbol{U}\boldsymbol{U}^T\boldsymbol{x}$. This is exactly what PCA does, with the exception that PCA has an additional orthonormal constraint: $\boldsymbol{U}\boldsymbol{U}^T = \boldsymbol{I}_{n\times n}$.

While *Figure 17.1* depicts an autoencoder without hidden layers within the encoder and decoder, we can, of course, 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, as you learned in *Chapter 14*, *Classifying Images with Deep Convolutional Neural Networks*.

Other types of autoencoders based on the size of latent space

As previously mentioned, the dimensionality of an autoencoder's latent space is typically lower than the dimensionality of the inputs ($p < d$), which makes autoencoders suitable for dimensionality reduction. For this reason, the latent vector is also often referred to as the "bottleneck," and this particular configuration of an autoencoder is also called **undercomplete**. However, there is a different category of autoencoders, called **overcomplete**, where the dimensionality of the latent vector, z, is, in fact, greater than the dimensionality of the input examples ($p > d$).

When training an overcomplete autoencoder, there is a trivial solution where the encoder and the decoder can simply learn to copy (memorize) the input features to their output layer. Obviously, this solution is not very useful. However, with some modifications to the training procedure, overcomplete autoencoders can be used for *noise reduction*.

In this case, during training, random noise, $\boldsymbol{\epsilon}$, is added to the input examples and the network learns to reconstruct the clean example, x, from the noisy signal, $\boldsymbol{x} + \boldsymbol{\epsilon}$. Then, at evaluation time, we provide the new examples that are naturally noisy (that is, noise is already present such that no additional artificial noise, $\boldsymbol{\epsilon}$, is added) in order to remove the existing noise from these examples. This particular autoencoder architecture and training method is referred to as a *denoising autoencoder*.

If you are interested, you can learn more about it in the research article *Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion* by *Pascal Vincent* and colleagues, 2010 (`http://www.jmlr.org/papers/v11/vincent10a.html`).

Generative models for synthesizing new data

Autoencoders are deterministic models, which means that after an autoencoder is trained, given an input, $\boldsymbol{x}$, it will be able to reconstruct the input from its compressed version in a lower-dimensional space. Therefore, it cannot generate new data beyond reconstructing its input through the transformation of the compressed representation.

A generative model, on the other hand, can generate a new example, $\tilde{\boldsymbol{x}}$, from a random vector, $\boldsymbol{z}$ (corresponding to the latent representation). A schematic representation of a generative model is shown in the following figure. The random vector, $\boldsymbol{z}$, comes from a distribution with fully known characteristics, so we can easily sample from such a distribution. For example, each element of $\boldsymbol{z}$ 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 $z_i \sim Normal(\mu = 0, \sigma^2 = 1)$):

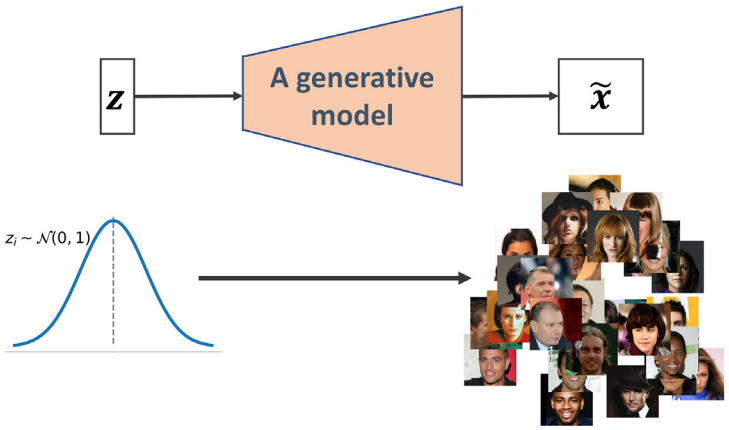

Figure 17.2: A generative model

As we have shifted our attention from autoencoders to generative models, you may have noticed that the decoder component of an autoencoder has some similarities with a generative model. In particular, they both receive a latent vector, $\boldsymbol{z}$, as input and return an output in the same space as $\boldsymbol{x}$. (For the autoencoder, $\hat{\boldsymbol{x}}$ is the reconstruction of an input, $\boldsymbol{x}$, and for the generative model, $\tilde{\boldsymbol{x}}$ is a synthesized sample.)

However, the major difference between the two is that we do not know the distribution of $\boldsymbol{z}$ in the autoencoder, while in a generative model, the distribution of $\boldsymbol{z}$ is fully characterizable. It is possible to generalize an autoencoder into a generative model, though. One approach is the **variational autoencoder** (**VAE**).

In a VAE receiving an input example, $\boldsymbol{x}$, the encoder network is modified in such a way that it computes two moments of the distribution of the latent vector: the mean, $\boldsymbol{\mu}$, and variance, $\boldsymbol{\sigma}^2$. During the training of a VAE, the network is forced to match these moments with those of a standard normal distribution (that is, zero mean and unit variance). Then, after the VAE model is trained, the encoder is discarded, and we can use the decoder network to generate new examples, $\tilde{\boldsymbol{x}}$, by feeding random $\boldsymbol{z}$ vectors from the "learned" Gaussian distribution.

Besides VAEs, there are other types of generative models, for example, *autoregressive models* and *normalizing flow models*. However, in this chapter, we are only going to focus on GAN models, which are among the most recent and most popular types of generative models in deep learning.

What is a generative model?

Note that generative models are traditionally defined as algorithms that model data input distributions, $p(x)$, or the joint distributions of the input data and associated targets, $p(x, y)$. By definition, these models are also capable of sampling from some feature, $\boldsymbol{x}_i$, conditioned on another feature, $\boldsymbol{x}_j$, which is known as **conditional inference**. In the context of deep learning, however, the term **generative model** is typically used to refer to models that generate realistic-looking data. This means that we can sample from input distributions, $p(x)$, but we are not necessarily able to perform conditional inference.

Generating new samples with GANs

To understand what GANs do in a nutshell, let's first assume we have a network that receives a random vector, $\boldsymbol{z}$, sampled from a known distribution, and generates an output image, $\boldsymbol{x}$. We will call this network **generator** (G) and use the notation $\widetilde{\boldsymbol{x}} = G(\boldsymbol{z})$ to refer to the generated output. Assume our goal is to generate some images, for example, face images, images of buildings, images of animals, or even 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's 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.

While an assessor function, as described in the previous paragraph, would make the image generation task very easy, the question is whether such a universal function to assess the quality of images exists and, if so, how it is defined. Obviously, as humans, we can easily assess the quality of output images when we observe the outputs of the network; although, we cannot (yet) backpropagate the result from our brain to the network. Now, if our brain can assess the quality of synthesized images, can we design an NN model to do the same thing? In fact, that's the general idea of a GAN.

As shown in *Figure 17.3*, a GAN model consists of an additional NN called **discriminator** (*D*), which is a classifier that learns to detect a synthesized image, $\tilde{\boldsymbol{x}}$, from a real image, $\boldsymbol{x}$:

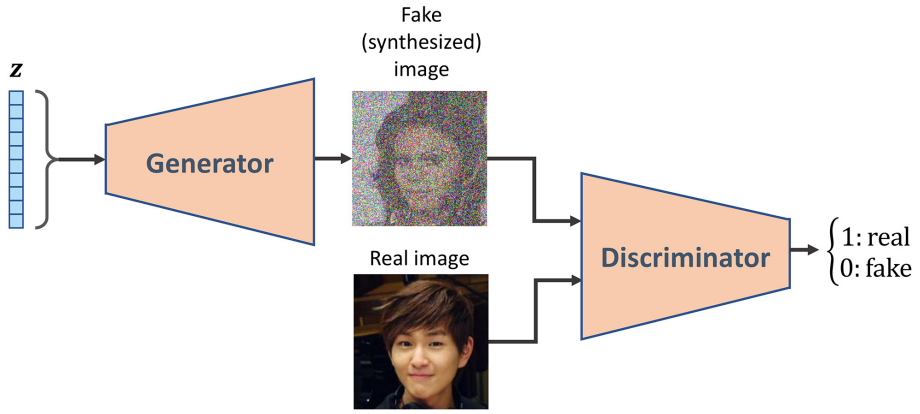

Figure 17.3: The discriminator distinguishes between the real image and the one created by the generator

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. In fact, the two networks play 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 of the generator and discriminator networks in a GAN model

The objective function of GANs, as described in the original paper *Generative Adversarial Nets* by *I. Goodfellow* and colleagues (`https://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf`), is as follows:

$$V\left(\theta^{(D)}, \theta^{(G)}\right) = E_{\boldsymbol{x}\sim p_{data}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + E_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}\left[\log\left(1 - D\left(G(\boldsymbol{z})\right)\right)\right]$$

Here, $V\left(\theta^{(D)}, \theta^{(G)}\right)$ is called the **value function**, which can be interpreted as a payoff: we want to maximize its value with respect to the discriminator (*D*), while minimizing its value with respect to the generator (*G*), that is, $\min_G \max_D V\left(\theta^{(D)}, \theta^{(G)}\right)$. $D(\boldsymbol{x})$ is the probability that indicates whether the input example, $\boldsymbol{x}$, is real or fake (that is, generated). The expression $E_{\boldsymbol{x}\sim p_{data}(\boldsymbol{x})}[\log D(\boldsymbol{x})]$ refers to the expected value of the quantity in brackets with respect to the examples from the data distribution (distribution of the real examples); $E_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}\left[\log\left(1 - D\left(G(\boldsymbol{z})\right)\right)\right]$ refers to the expected value of the quantity with respect to the distribution of the input, $\boldsymbol{z}$, vectors.

One training step of a GAN model with such a value function requires two optimization steps: (1) maximizing the payoff for the discriminator and (2) minimizing the payoff for the generator. A practical way of training GANs is to alternate between these two optimization steps: (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. This process should be repeated at each training iteration. Let's assume that the generator network is fixed, and we want to optimize the discriminator. Both terms in the value function $V\left(\theta^{(D)}, \theta^{(G)}\right)$ contribute to optimizing the discriminator, where the first term corresponds to the loss associated with the real examples, and the second term is the loss for the fake examples. Therefore, when G is fixed, our objective is to *maximize* $V\left(\theta^{(D)}, \theta^{(G)}\right)$, which means making the discriminator better at distinguishing between real and generated images.

After optimizing the discriminator using the loss terms for real and fake samples, we then fix the discriminator and optimize the generator. In this case, only the second term in $V\left(\theta^{(D)}, \theta^{(G)}\right)$ contributes to the gradients of the generator. As a result, when D is fixed, our objective is to *minimize* $V\left(\theta^{(D)}, \theta^{(G)}\right)$, which can be written as $\min_G E_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}\left[\log\left(1 - D\left(G(\boldsymbol{z})\right)\right)\right]$. As was mentioned in the original GAN paper by Goodfellow and colleagues, this function, $\log\left(1 - D\left(G(\boldsymbol{z})\right)\right)$, suffers from vanishing gradients in the early training stages. The reason for this is that the outputs, $G(\boldsymbol{z})$, early in the learning process, look nothing like real examples, and therefore $D(G(\boldsymbol{z}))$ will be close to zero with high confidence. This phenomenon is called **saturation**. To resolve this issue, we can reformulate the minimization objective, $\min_G E_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}\left[\log\left(1 - D\left(G(\boldsymbol{z})\right)\right)\right]$, by rewriting it as $\max_G E_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}\left[\log\left(D\left(G(\boldsymbol{z})\right)\right)\right]$.

This replacement means that for training the generator, we can swap the labels of real and fake examples and carry out a regular function minimization. In other words, even though the examples synthesized by the generator are fake and are therefore labeled 0, we can flip the labels by assigning label 1 to these examples and *minimize* the binary cross-entropy loss with these new labels instead of maximizing $\max_G E_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}\left[\log\left(D\left(G(\boldsymbol{z})\right)\right)\right]$.

Now that we have covered the general optimization procedure for training GAN models, let's explore the various data labels that we can use when training GANs. 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. Therefore, we can determine the ground truth labels for the discriminator loss as follows:

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

What about the labels to train the generator? As we want the generator to synthesize realistic images, we want to penalize 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.

Putting all of this together, the following figure displays the individual steps in a simple GAN model:

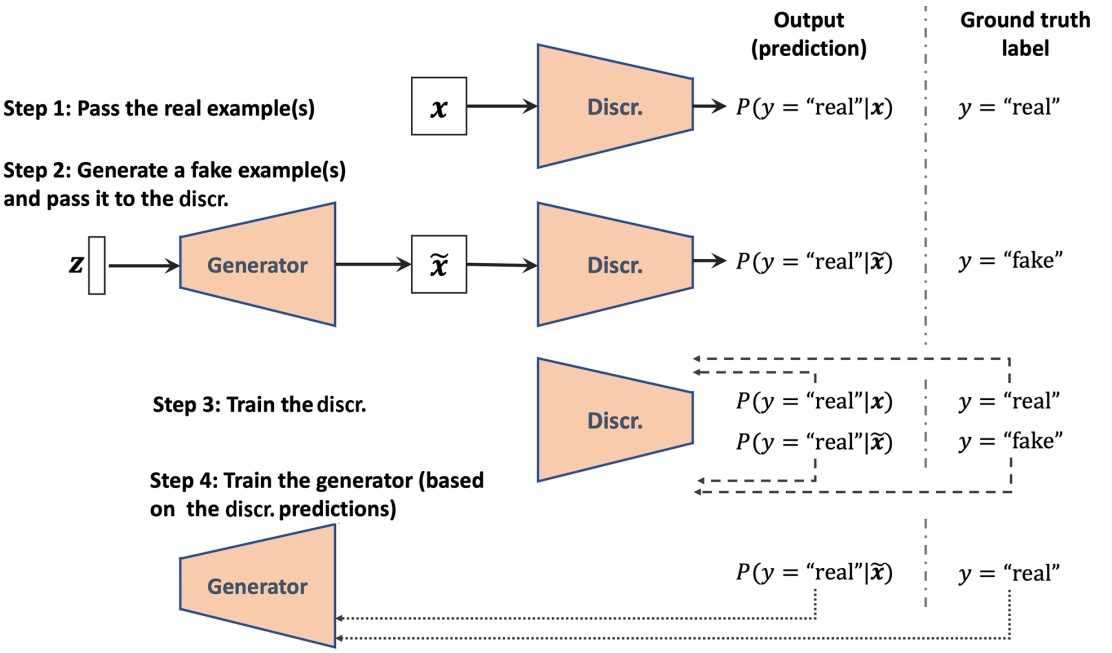

Figure 17.4: The steps in building a GAN model

In the following section, we will implement a GAN from scratch to generate new handwritten digits.

Implementing a GAN from scratch

In this section, we will cover how to implement and train a GAN model to generate new images such as MNIST digits. Since the training on a normal **central processing unit** (**CPU**) may take a long time, in the following subsection, we will cover how to set up the Google Colab environment, which will allow us to run the computations on **graphics processing units** (**GPUs**).

Training GAN models on Google Colab

Some of the code examples in this chapter may require extensive computational resources that go beyond a conventional laptop or a workstation without a GPU. If you already have an NVIDIA GPU-enabled computing machine available, with CUDA and cuDNN libraries installed, you can use that to speed up the computations.

However, since many of us do not have access to high-performance computing resources, we will use the Google Colaboratory environment (often referred to as Google Colab), which is a free cloud computing service (available in most countries).

Google Colab provides Jupyter Notebook instances that run on the cloud; the notebooks can be saved on Google Drive or GitHub. While the platform provides various different computing resources, such as CPUs, GPUs, and even **tensor processing units** (**TPUs**), it is important to highlight that the execution time is currently limited to 12 hours. Therefore, any notebook running longer than 12 hours will be interrupted.

The code blocks in this chapter will need a maximum computing time of two to three hours, so this will not be an issue. However, if you decide to use Google Colab for other projects that take longer than 12 hours, be sure to use checkpointing and save intermediate checkpoints.

Jupyter Notebook

Jupyter Notebook is a graphical user interface (GUI) for running code interactively and interleaving it with text documentation and figures. Due to its versatility and ease of use, it has become one of the most popular tools in data science.

For more information about the general Jupyter Notebook GUI, please view the official documentation at `https://jupyter-notebook.readthedocs.io/en/stable/`. All the code in this book is also available in the form of Jupyter notebooks, and a short introduction can be found in the code directory of the first chapter.

Lastly, we highly recommend *Adam Rule* et al.'s article *Ten simple rules for writing and sharing computational analyses in Jupyter Notebooks* on using Jupyter Notebook effectively in scientific research projects, which is freely available at `https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1007007`.

Accessing Google Colab is very straightforward. You can visit `https://colab.research.google.com`, which automatically takes you to a prompt window where you can see your existing Jupyter notebooks. From this prompt window, click the **Google Drive** tab, as shown in *Figure 17.5*. This is where you will save the notebook on your Google Drive.

Then, to create a new notebook, click on the **New notebook** link at the bottom of the prompt window:

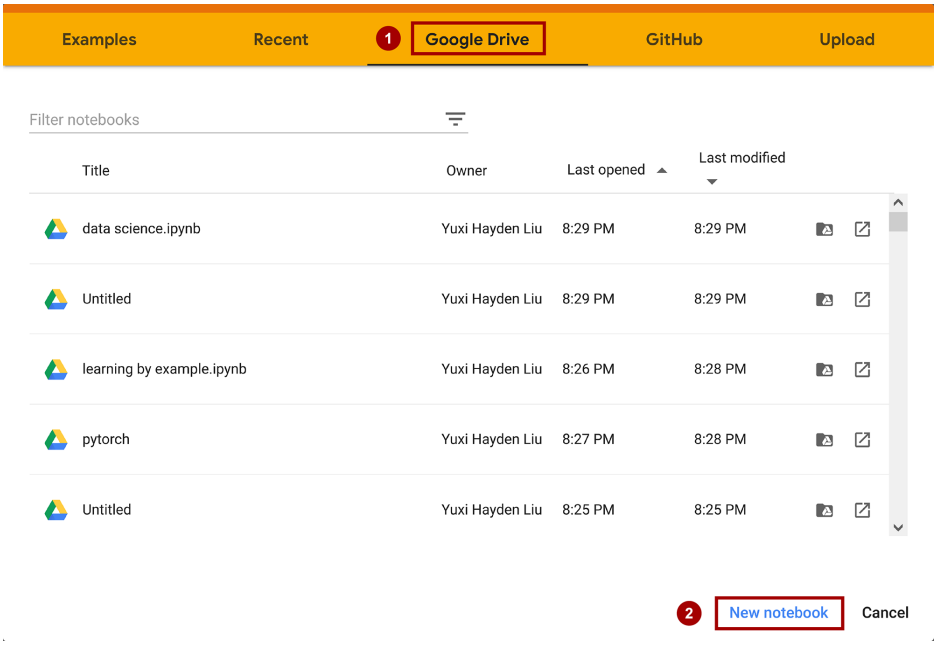

Figure 17.5: Creating a new Python notebook in Google Colab

This will create and open a new notebook for you. All the code examples you write in this notebook will be automatically saved, and you can later access the notebook from your Google Drive in a directory called **Colab Notebooks**.

In the next step, we want to utilize GPUs to run the code examples in this notebook. To do this, from the **Runtime** option in the menu bar of this notebook, click on **Change runtime type** and select **GPU**, as shown in *Figure 17.6*:

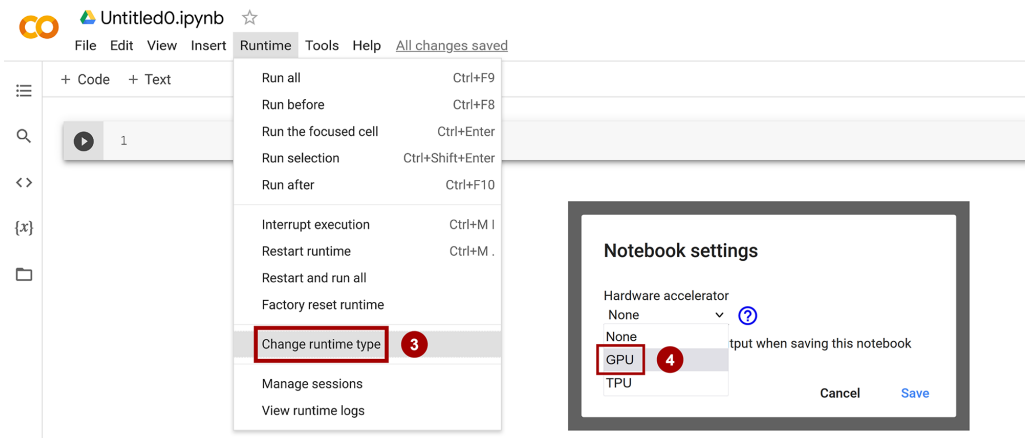

Figure 17.6: Utilizing GPUs in Google Colab

In the last step, we just need to install the Python packages that we will need for this chapter. The Colab Notebooks environment already comes with certain packages, such as NumPy, SciPy, and the latest stable version of PyTorch. At the time of writing, the latest stable version on Google Colab is PyTorch 1.9.

Now, we can test the installation and verify that the GPU is available using the following code:

```
>>> import torch
>>> print(torch.__version__)
1.9.0+cu111
>>> print("GPU Available:", torch.cuda.is_available())
GPU Available: True
>>> if torch.cuda.is_available():
...     device = torch.device("cuda:0")
... else:
...     device = "cpu"
>>> print(device)
cuda:0
```

Furthermore, if you want to save the model to your personal Google Drive, or transfer or upload other files, you need to mount Google Drive. To do this, execute the following in a new cell of the notebook:

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

This will provide a link to authenticate the Colab Notebook accessing your Google Drive. After following the instructions for authentication, it will provide an authentication code that you need to copy and paste into the designated input field below the cell you have just executed. Then, your Google Drive will be mounted and available at `/content/drive/My Drive`. Alternatively, you can mount it via the GUI interface as highlighted in *Figure 17.7*:

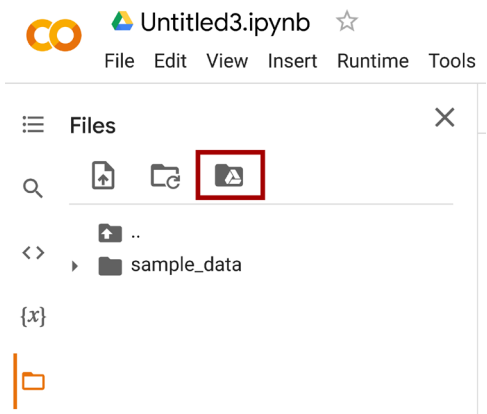

Figure 17.7: Mounting your Google Drive

Implementing the generator and the discriminator networks

We will start the implementation of our first GAN model with a generator and a discriminator as two fully connected networks with one or more hidden layers, as shown in *Figure 17.8*:

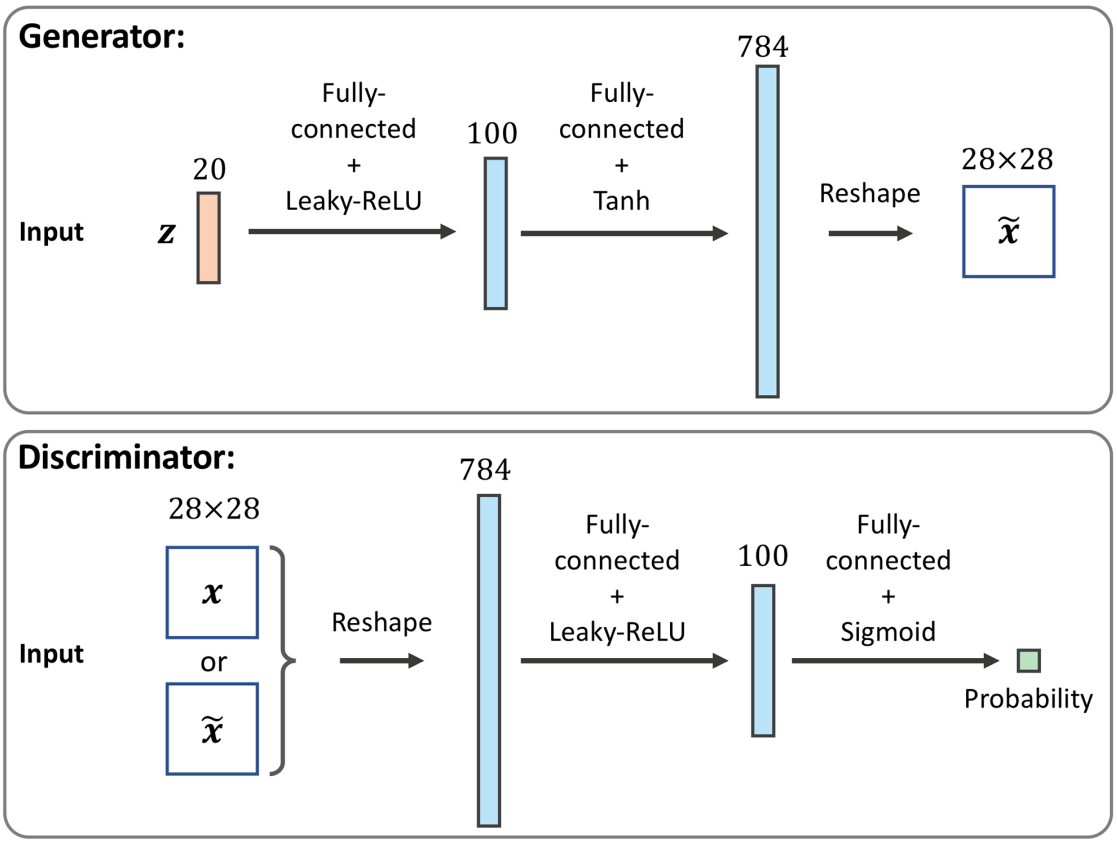

Figure 17.8: A GAN model with a generator and discriminator as two fully connected networks

Figure 17.8 depicts the original GAN based on fully connected layers, which we will refer to as a *vanilla GAN*.

In this model, for each hidden layer, we will apply the leaky ReLU activation function. The use of ReLU results in sparse gradients, which may not be suitable when we want to have the gradients for the full range of input values. 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. (Using tanh activation is recommended for the generator network since it helps with the learning.)

The output layer in the discriminator has no activation function (that is, linear activation) to get the logits. Alternatively, we can use the sigmoid activation function to get probabilities as output.

Leaky rectified linear unit (ReLU) activation function

In *Chapter 12*, *Parallelizing Neural Network Training with PyTorch*, we covered different nonlinear activation functions that can be used in an NN model. If you recall, the ReLU activation function was defined as $\sigma(z) = \max(0, z)$, which suppresses the negative (pre-activation) inputs; that is, negative inputs are set to zero. Consequently, using the ReLU activation function may result in sparse gradients during backpropagation. Sparse gradients are not always detrimental and can even benefit models for classification. However, in certain applications, such as GANs, it can be beneficial to obtain the gradients for the full range of input values, which we can achieve by making a slight modification to the ReLU function such that it outputs small values for negative inputs. This modified version of the ReLU function is also known as **leaky ReLU**. In short, 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.

The leaky ReLU activation function is defined as follows:

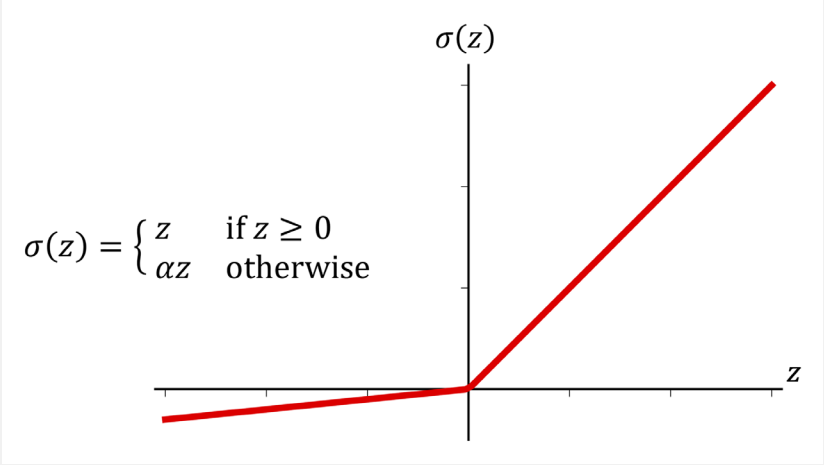

Figure 17.9: The leaky ReLU activation function

Here, α determines the slope for the negative (preactivation) inputs.

We will define two helper functions for each of the two networks, instantiate a model from the PyTorch `nn.Sequential` class, and add the layers as described. The code is as follows:

```
>>> import torch.nn as nn
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> ## 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
```

Next, we will specify the training settings for the model. As you will remember from previous chapters, the image size in the MNIST dataset is 28×28 pixels. (That is only one color channel because MNIST contains only grayscale images.) We will further specify the size of the input vector, z, to be 20. Since we are implementing a very simple GAN model for illustration purposes only and using fully connected layers, we will only use a single hidden layer with 100 units in each network. In the following code, we will specify and initialize the two networks, and print their summary information:

```
>>> image_size = (28, 28)
>>> z_size = 20
>>> gen_hidden_layers = 1
>>> gen_hidden_size = 100
>>> disc_hidden_layers = 1
>>> 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=False)
  (relu_g0): LeakyReLU(negative_slope=0.01)
  (fc_g1): Linear(in_features=100, out_features=784, bias=True)
  (tanh_g): Tanh()
)

>>> 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

In the next step, we will load the MNIST dataset from PyTorch and apply the necessary preprocessing steps. Since the output layer of the generator is using the tanh activation function, the pixel values of the synthesized images will be in the range (–1, 1). However, the input pixels of the MNIST images are within the range [0, 255] (with a data type `PIL.Image.Image`). Thus, in the preprocessing steps, we will use the `torchvision.transforms.ToTensor` function to convert the input image tensors to a tensor. As a result, 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 –0.5 and scale them by a factor of 0.5 such that the pixel intensities will be rescaled to be in the range [–1, 1], which can improve gradient descent-based learning:

```
>>> 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])
```

Furthermore, we will also create a random vector, z, based on the desired random distribution (in this code example, uniform or normal, which are the most common choices):

```
>>> 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's inspect the dataset object that we created. In the following code, we will take one batch of examples and print the array shapes of this sample of input vectors and images. Furthermore, in order to understand the overall data flow of our GAN model, in the following code, we will process a forward pass for our generator and discriminator.

First, we will feed the batch of input, z, vectors to the generator and get its output, `g_output`. This will be a batch of fake examples, which will be fed to the discriminator model to get the probabilities for the batch of fake examples, `d_proba_fake`. Furthermore, the processed images that we get from the dataset object will be fed to the discriminator model, which will result in the probabilities for the real examples, `d_proba_real`. The code is as follows:

```
>>> 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)
input-z -- shape: torch.Size([32, 20])
input-real -- shape: torch.Size([32, 784])

>>> g_output = gen_model(input_z)
>>> print('Output of G -- shape:', g_output.shape)
Output of G -- shape: torch.Size([32, 784])

>>> 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)
Disc. (real) -- shape: torch.Size([32, 1])
Disc. (fake) -- shape: torch.Size([32, 1])
```

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

Training the GAN model

As the next step, we will create an instance of `nn.BCELoss` as our loss function and use that to calculate the binary cross-entropy loss for the generator and discriminator associated with the batches that we just processed. To do this, we also need the ground truth labels for each output. For the generator, we will 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 term will be a vector of 0s that we can generate via the `torch.zeros()` (or `torch.zeros_like()`) function. Similarly, we can generate the ground truth values for the real images via the `torch.ones()` (or `torch.ones_like()`) function, which creates a vector of 1s:

```
>>> 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}')
Generator Loss: 0.6863

>>> ## 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}')
Discriminator Losses: Real 0.6226 Fake 0.7007
```

The previous code example shows the step-by-step calculation of the different loss terms for the purpose of understanding the overall concept behind training a GAN model. The following code will set up the GAN model and implement the training loop, where we will include these calculations in a `for` loop.

We will start with setting up the data loader for the real dataset, the generator and discriminator model, as well as a separate Adam optimizer for each of the two models:

```
>>> batch_size = 64
>>> torch.manual_seed(1)
>>> np.random.seed(1)
>>> mnist_dl = DataLoader(mnist_dataset, batch_size=batch_size,
...                       shuffle=True, drop_last=True)

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

In addition, we will compute the loss gradients with respect to the model weights and optimize the parameters of the generator and discriminator using two separate Adam optimizers. We will write two utility functions for training the discriminator and the generator as follows:

```
>>> ## 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()
>>>
>>> ## 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()
```

Next, we will alternate between the training of the generator and the discriminator for 100 epochs. For each epoch, we will record the loss for the generator, the loss for the discriminator, and the loss for the real data and fake data respectively. Furthermore, after each epoch, we will generate some examples from a fixed noise input using the current generator model by calling the `create_samples()` function. We will store the synthesized images in a Python list. The code is as follows:

```
>>> 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
>>>
>>> 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}'
...           f' 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 0.9546/0.8957 [D-Real: 0.8074 D-Fake: 0.4687]
Epoch 002 | Avg Losses >> G/D 0.9571/1.0841 [D-Real: 0.6346 D-Fake: 0.4155]
Epoch ...
Epoch 100 | Avg Losses >> G/D 0.8622/1.2878 [D-Real: 0.5488 D-Fake: 0.4518]
```

Using a GPU on Google Colab, the training process that we implemented in the previous code block should be completed in less than an hour. (It may even be faster on your personal computer if you have a recent and capable CPU and a GPU.) After the model training has completed, it is often helpful to plot the discriminator and generator losses to analyze the behavior of both subnetworks and assess whether they converged.

It is also helpful to plot 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:

```
>>> 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.show()
```

Figure 17.10 shows the results:

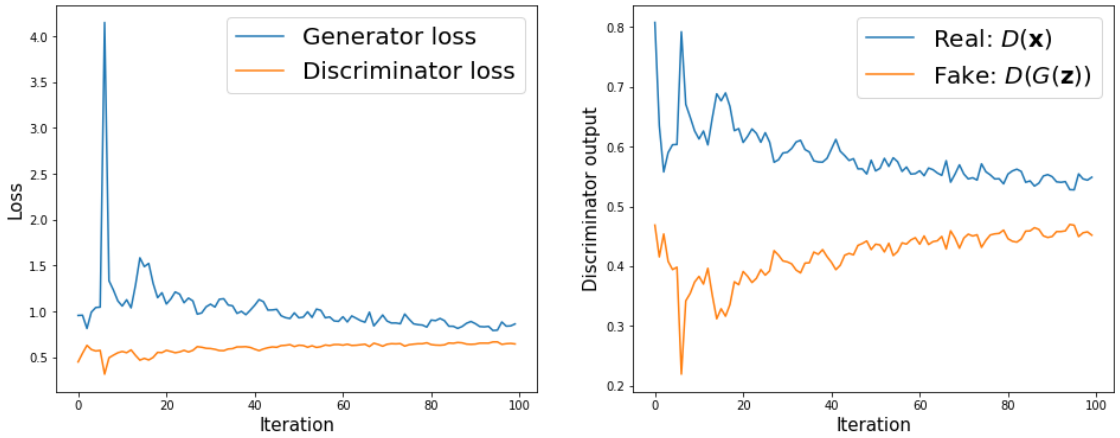

Figure 17.10: The discriminator performance

As you can see from the discriminator outputs in the previous figure, during the early stages of the training, the discriminator was able to quickly learn to distinguish quite accurately between the real and fake examples; that is, the fake examples had probabilities close to 0, and the real examples had probabilities close to 1. The reason for that was that the fake examples were nothing like the real ones; therefore, distinguishing between real and fake was rather easy. As the training proceeds further, the generator will become better at synthesizing realistic images, which will result in probabilities of both real and fake examples that are close to 0.5.

Furthermore, we can also see how the outputs of the generator, that is, the synthesized images, change during training. In the following code, we will visualize some of the images produced by the generator for a selection of epochs:

```
>>> 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.show()
```

Figure 17.11 shows the produced images:

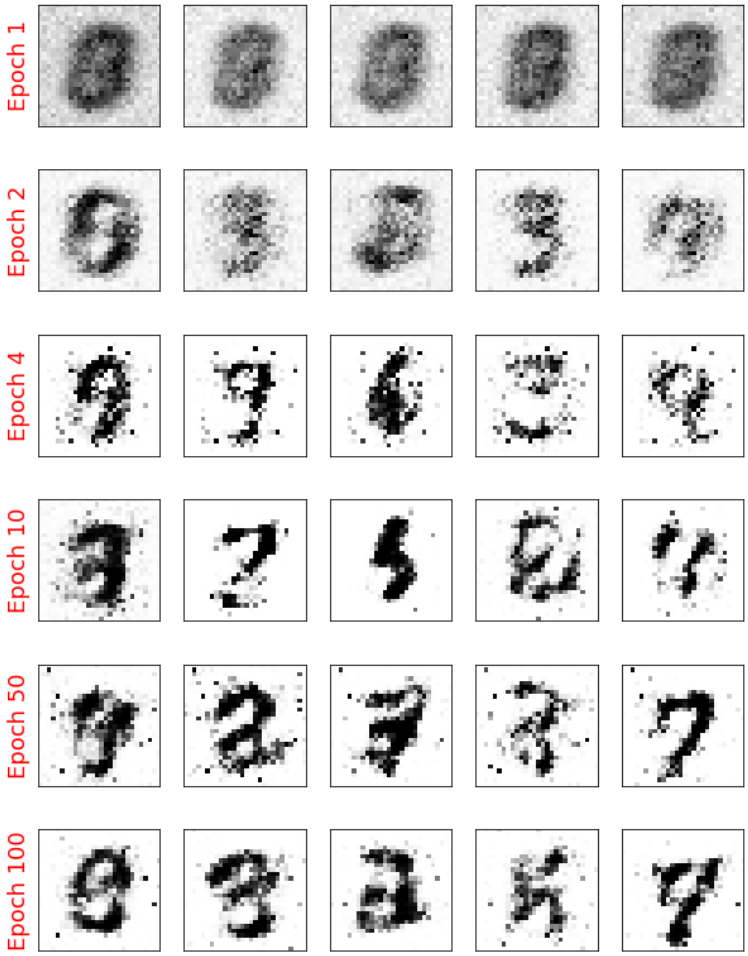

Figure 17.11: Images produced by the generator

As you can see from *Figure 17.11*, the generator network produced more and more realistic images as the training progressed. However, even after 100 epochs, the produced images still look very different from the handwritten digits contained in the MNIST dataset.

In this section, we designed a very simple GAN model with only a single fully connected hidden layer for both the generator and discriminator. 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.

As we learned in *Chapter 14*, *Classifying Images with Deep Convolutional Neural Networks*, NN architectures with convolutional layers have several advantages over fully connected layers when it comes to image classification. In a similar sense, adding convolutional layers to our GAN model to work with image data might improve the outcome. In the next section, we will implement a **deep convolutional GAN** (**DCGAN**), which uses convolutional layers for both the generator and the discriminator networks.

Improving the quality of synthesized images using a convolutional and Wasserstein GAN

In this section, we will implement a DCGAN, which will enable us to improve the performance we saw in the previous GAN example. Additionally, we will briefly talk about an extra key technique, **Wasserstein GAN** (**WGAN**).

The techniques that we will cover in this section will include the following:

- Transposed convolution
- Batch normalization (BatchNorm)
- WGAN

The DCGAN was proposed in 2016 by *A. Radford*, *L. Metz*, and *S. Chintala* in their article *Unsupervised representation learning with deep convolutional generative adversarial networks*, which is freely available at `https://arxiv.org/pdf/1511.06434.pdf`. In this article, the researchers proposed using convolutional layers for both the generator and discriminator networks. Starting from a random vector, $\boldsymbol{z}$, the DCGAN first uses a fully connected layer to project $\boldsymbol{z}$ into a new vector with a proper size so that it can be reshaped into a spatial convolution representation ($h \times w \times c$), which is smaller than the output image size. Then, a series of convolutional layers, known as **transposed convolution**, are used to upsample the feature maps to the desired output image size.

Transposed convolution

In *Chapter 14*, you learned about the convolution operation in one- and two-dimensional spaces. In particular, we looked at how the choices for the padding and strides change the output feature maps. While a convolution operation is usually used to downsample the feature space (for example, by setting the stride to 2, or by adding a pooling layer after a convolutional layer), a *transposed convolution* operation is usually used for *upsampling* the feature space.

To understand the transposed convolution operation, let's go through a simple thought experiment. Assume that we have an input feature map of size $n \times n$. Then, we apply a 2D convolution operation with certain padding and stride parameters to this $n \times n$ input, resulting in an output feature map of size $m \times m$. Now, the question is, how we can apply another convolution operation to obtain a feature map with the initial dimension $n \times n$ from this $m \times m$ output feature map while maintaining the connectivity patterns between the input and output? Note that only the shape of the $n \times n$ input matrix is recovered and not the actual matrix values.

This is what transposed convolution does, as shown in *Figure 17.12*:

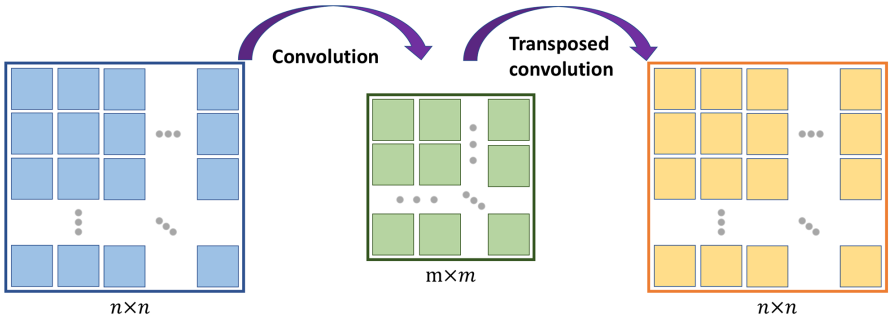

Figure 17.12: Transposed convolution

Transposed convolution versus deconvolution

Transposed convolution is also called **fractionally strided convolution**. In deep learning literature, another common term that is used to refer to transposed convolution is **deconvolution**. However, note that deconvolution was originally defined as the inverse of a convolution operation, f, on a feature map, $\boldsymbol{x}$, with weight parameters, $\boldsymbol{w}$, producing feature map $\boldsymbol{x}'$, $f_w(\boldsymbol{x}) = \boldsymbol{x}'$. A deconvolution function, f^{-1}, can then be defined as $f_w^{-1}\big(f(\boldsymbol{x})\big) = \boldsymbol{x}$. However, note that the transposed convolution is merely focused on recovering the dimensionality of the feature space and not the actual values.

Upsampling feature maps using transposed convolution works by inserting 0s between the elements of the input feature maps. *Figure 17.13* shows an example of applying transposed convolution to an input of size 4×4, with a stride of 2×2 and kernel size of 2×2. The matrix of size 9×9 in the center shows the results after inserting such 0s into the input feature map. Then, performing a normal convolution using the 2×2 kernel with a stride of 1 results in an output of size 8×8. We can verify the backward direction by performing a regular convolution on the output with a stride of 2, which results in an output feature map of size 4×4, which is the same as the original input size:

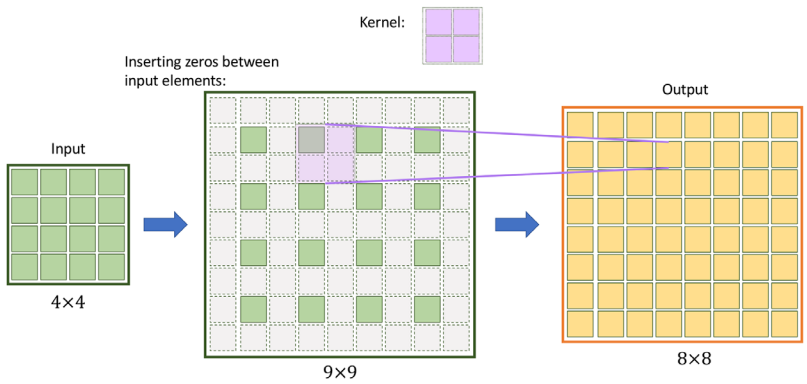

Figure 17.13: Applying transposed convolution to a 4×4 input

Figure 17.13 shows how transposed convolution works in general. There are various cases in which input size, kernel size, strides, and padding variations can change the output. If you want to learn more about all these different cases, refer to the tutorial *A Guide to Convolution Arithmetic for Deep Learning* by *Vincent Dumoulin* and *Francesco Visin*, 2018 (`https://arxiv.org/pdf/1603.07285.pdf`.)

Batch normalization

BatchNorm was introduced in 2015 by Sergey Ioffe and Christian Szegedy in the article *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*, which you can access via arXiv at `https://arxiv.org/pdf/1502.03167.pdf`. One of the main ideas behind BatchNorm is normalizing the layer inputs and preventing changes in their distribution during training, which enables faster and better convergence.

BatchNorm transforms a mini-batch of features based on its computed statistics. Assume that we have the net preactivation feature maps obtained after a convolutional layer in a four-dimensional tensor, $\boldsymbol{Z}$, with the shape $[m \times c \times h \times w]$, where m is the number of examples in the batch (i.e., batch size), $h \times w$ is the spatial dimension of the feature maps, and c is the number of channels. BatchNorm can be summarized in three steps, as follows:

1. Compute the mean and standard deviation of the net inputs for each mini-batch:

$$\boldsymbol{\mu}_B = \frac{1}{m \times h \times w} \sum_{i,j,k} \boldsymbol{Z}^{[i,j,k]}$$

$$\boldsymbol{\sigma}_B^2 = \frac{1}{m \times h \times w} \sum_{i,j,k} \left(\boldsymbol{Z}^{[i,j,k]} - \boldsymbol{\mu}_B\right)^2$$

   where $\boldsymbol{\mu}_B$ and $\boldsymbol{\sigma}_B^2$ both have size c.

2. Standardize the net inputs for all examples in the batch:

$$\boldsymbol{Z}_{\text{std}}^{[i]} = \frac{\boldsymbol{Z}^{[i]} - \boldsymbol{\mu}_B}{\boldsymbol{\sigma}_B + \epsilon}$$

   where ϵ is a small number for numerical stability (that is, to avoid division by zero).

3. Scale and shift the normalized net inputs using two learnable parameter vectors, $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$, of size c (number of channels):

$$\boldsymbol{A}_{\text{pre}}^{[i]} = \boldsymbol{\gamma} \boldsymbol{Z}_{\text{std}}^{[i]} + \boldsymbol{\beta}$$

Figure 17.14 illustrates the process:

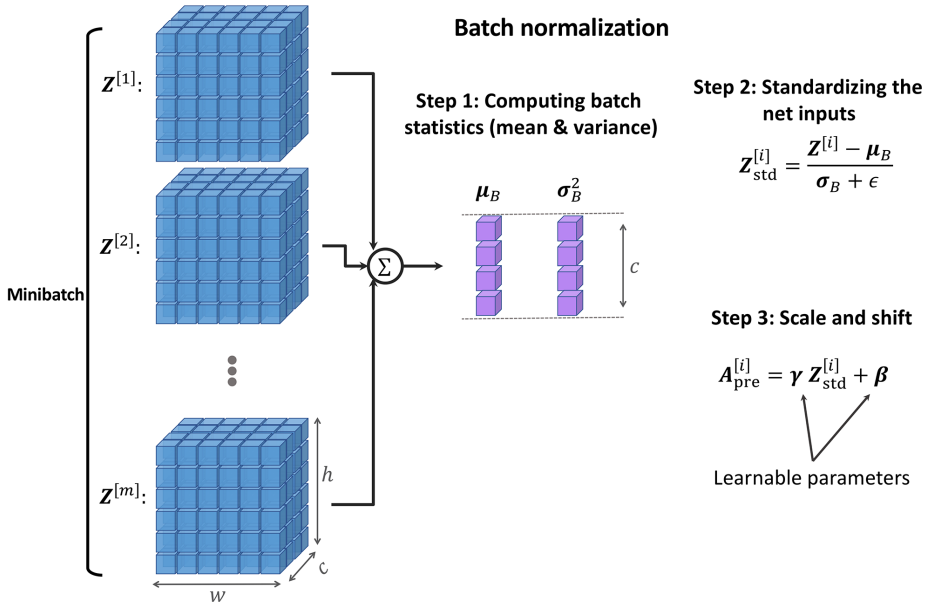

Figure 17.14: The process of batch normalization

In the first step of BatchNorm, the mean, $\boldsymbol{\mu}_B$, and standard deviation, $\boldsymbol{\sigma}_B$, of the mini-batch are computed. Both $\boldsymbol{\mu}_B$ and $\boldsymbol{\sigma}_B$ are vectors of size c (where c is the number of channels). Then, these statistics are used in *step 2* to scale the examples in each mini-batch via z-score normalization (standardization), resulting in standardized net inputs, $\boldsymbol{Z}_{\text{std}}^{[i]}$. As a consequence, these net inputs are mean-centered and have *unit variance*, which is generally a useful property for gradient descent-based optimization. On the other hand, always normalizing the net inputs such that they have the same properties across the different mini-batches, which can be diverse, can severely impact the representational capacity of NNs. This can be understood by considering a feature, $x \sim N(0,1)$, which, after sigmoid activation to $\sigma(x)$, results in a linear region for values close to 0. Therefore, in step 3, the learnable parameters, $\boldsymbol{\beta}$ and $\boldsymbol{\gamma}$, which are vectors of size c (number of channels), allow BatchNorm to control the shift and spread of the normalized features.

During training, the running averages, $\boldsymbol{\mu}_B$, and running variance, $\boldsymbol{\sigma}_B^2$, are computed, which are used along with the tuned parameters, $\boldsymbol{\beta}$ and $\boldsymbol{\gamma}$, to normalize the test example(s) at evaluation.

Why does BatchNorm help optimization?

Initially, BatchNorm was developed to reduce the so-called **internal covariance shift**, which is defined as the changes that occur in the distribution of a layer's activations due to the updated network parameters during training.

To explain this with a simple example, consider a fixed batch that passes through the network at epoch 1. We record the activations of each layer for this batch. After iterating through the whole training dataset and updating the model parameters, we start the second epoch, where the previously fixed batch passes through the network. Then, we compare the layer activations from the first and second epochs. Since the network parameters have changed, we observe that the activations have also changed. This phenomenon is called the **internal covariance shift**, which was believed to decelerate NN training.

However, in 2018, S. Santurkar, D. Tsipras, A. Ilyas, and A. Madry further investigated what makes BatchNorm so effective. In their study, the researchers observed that the effect of BatchNorm on the internal covariance shift is marginal. Based on the outcome of their experiments, they hypothesized that the effectiveness of BatchNorm is, instead, based on a smoother surface of the loss function, which makes the non-convex optimization more robust.

If you are interested in learning more about these results, read through the original paper, *How Does Batch Normalization Help Optimization?*, which is freely available at `http://papers.nips.cc/paper/7515-how-does-batch-normalization-help-optimization.pdf`.

The PyTorch API provides a class, `nn.BatchNorm2d()` (`nn.BatchNorm1d()` for 1D input), that we can use as a layer when defining our models; it will perform all of the steps that we described for BatchNorm. Note that the behavior for updating the learnable parameters, $\boldsymbol{\gamma}$ and $\boldsymbol{\beta}$, depends on whether the model is a training model not. These parameters are learned only during training and are then used for normalization during evaluation.

Implementing the generator and discriminator

At this point, we have covered the main components of a DCGAN model, which we will now implement. The architectures of the generator and discriminator networks are summarized in the following two figures.

The generator takes a vector, $\boldsymbol{z}$, of size 100 as input. Then, a series of transposed convolutions using `nn.ConvTranspose2d()` upsamples the feature maps until the spatial dimension of the resulting feature maps reaches 28×28. The number of channels is reduced by half after each transposed convolutional layer, except the last one, which uses only one output filter to generate a grayscale image. Each transposed convolutional layer is followed by BatchNorm and leaky ReLU activation functions, except the last one, which uses tanh activation (without BatchNorm).

The architecture for the generator (the feature maps after each layer) is shown in *Figure 17.15*:

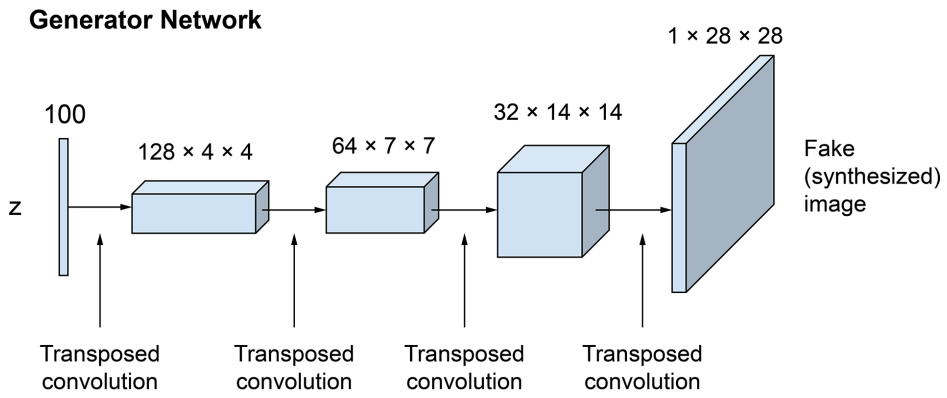

Figure 17.15: The generator network

The discriminator receives images of size 1×28×28, which are passed through four convolutional layers. The first three convolutional layers reduce the spatial dimensionality by 4 while increasing the number of channels of the feature maps. Each convolutional layer is also followed by BatchNorm and leaky ReLU activation. The last convolutional layer uses kernels of size 4×4 and a single filter to reduce the spatial dimensionality of the output to 1×1×1. Finally, the convolutional output is followed by a sigmoid function and squeezed to one dimension:

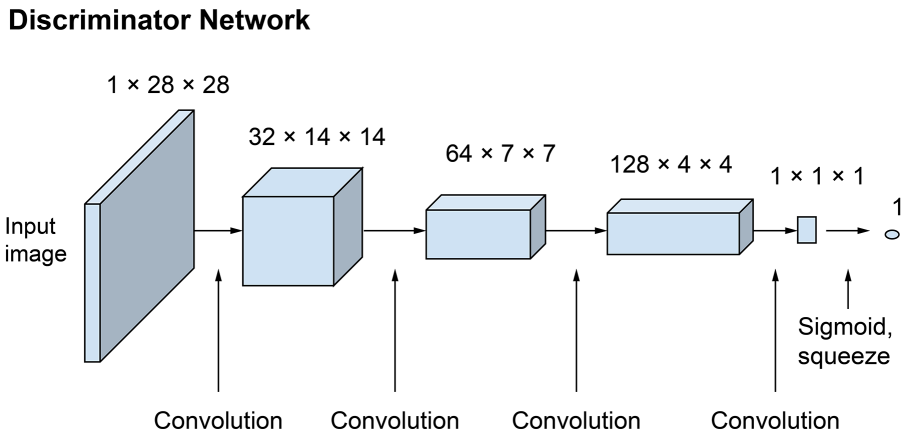

Figure 17.16: The discriminator network

Architecture design considerations for convolutional GANs

Notice that the number of feature maps follows different trends between the generator and the discriminator. In the generator, we start with a large number of feature maps and decrease them as we progress toward the last layer. On the other hand, in the discriminator, we start with a small number of channels and increase it toward the last layer. This is an important point for designing CNNs with the number of feature maps and the spatial size of the feature maps in reverse order. When the spatial size of the feature maps increases, the number of feature maps decreases and vice versa.

In addition, note that it's usually not recommended to use bias units in the layer that follows a BatchNorm layer. Using bias units would be redundant in this case, since BatchNorm already has a shift parameter, $\boldsymbol{\beta}$. You can omit the bias units for a given layer by setting `bias=False` in `nn.ConvTranspose2d` or `nn.Conv2d`.

The code for the helper function to make the generator and the discriminator network class is as follows:

```
>>> def make_generator_network(input_size, n_filters):
...     model = nn.Sequential(
...         nn.ConvTranspose2d(input_size, n_filters*4, 4,
...                            1, 0, bias=False),
...         nn.BatchNorm2d(n_filters*4),
...         nn.LeakyReLU(0.2),
...         nn.ConvTranspose2d(n_filters*4, n_filters*2,
...                            3, 2, 1, bias=False),
...         nn.BatchNorm2d(n_filters*2),
...         nn.LeakyReLU(0.2),
...         nn.ConvTranspose2d(n_filters*2, n_filters,
...                            4, 2, 1, bias=False),
...         nn.BatchNorm2d(n_filters),
...         nn.LeakyReLU(0.2),
...         nn.ConvTranspose2d(n_filters, 1, 4, 2, 1,
...                            bias=False),
...         nn.Tanh()
...     )
...     return model
>>>
>>> class Discriminator(nn.Module):
...     def __init__(self, n_filters):
...         super().__init__()
...         self.network = nn.Sequential(
...             nn.Conv2d(1, n_filters, 4, 2, 1, bias=False),
```

```
...             nn.LeakyReLU(0.2),
...             nn.Conv2d(n_filters, n_filters*2,
...                       4, 2, 1, bias=False),
...             nn.BatchNorm2d(n_filters * 2),
...             nn.LeakyReLU(0.2),
...             nn.Conv2d(n_filters*2, n_filters*4,
...                       3, 2, 1, bias=False),
...             nn.BatchNorm2d(n_filters*4),
...             nn.LeakyReLU(0.2),
...             nn.Conv2d(n_filters*4, 1, 4, 1, 0, bias=False),
...             nn.Sigmoid()
...         )
...
...     def forward(self, input):
...         output = self.network(input)
...         return output.view(-1, 1).squeeze(0)
```

With the helper function and class, you can build a DCGAN model and train it by using the same MNIST dataset object we initialized in the previous section when we implemented the simple, fully connected GAN. We can create the generator networks using the helper function and print its architecture as follows:

```
>>> z_size = 100
>>> image_size = (28, 28)
>>> n_filters = 32
>>> gen_model = make_generator_network(z_size, n_filters).to(device)
>>> print(gen_model)
Sequential(
  (0): ConvTranspose2d(100, 128, kernel_size=(4, 4), stride=(1, 1), bias=False)
  (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
  (2): LeakyReLU(negative_slope=0.2)
  (3): ConvTranspose2d(128, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1,
1), bias=False)
  (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
  (5): LeakyReLU(negative_slope=0.2)
  (6): ConvTranspose2d(64, 32, kernel_size=(4, 4), stride=(2, 2), padding=(1,
1), bias=False)
  (7): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
```

```
  (8): LeakyReLU(negative_slope=0.2)
  (9): ConvTranspose2d(32, 1, kernel_size=(4, 4), stride=(2, 2), padding=(1,
1), bias=False)
  (10): Tanh()
)
```

Similarly, we can generate the discriminator network and see its architecture:

```
>>> disc_model = Discriminator(n_filters).to(device)
>>> print(disc_model)
Discriminator(
  (network): Sequential(
    (0): Conv2d(1, 32, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1),
bias=False)
    (1): LeakyReLU(negative_slope=0.2)
    (2): Conv2d(32, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1),
bias=False)
    (3): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
    (4): LeakyReLU(negative_slope=0.2)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1),
bias=False)
    (6): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
    (7): LeakyReLU(negative_slope=0.2)
    (8): Conv2d(128, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (9): Sigmoid()
  )
)
```

Also, we can use the same loss functions and optimizers as we did in the *Training the GAN model* subsection:

```
>>> loss_fn = nn.BCELoss()
>>> g_optimizer = torch.optim.Adam(gen_model.parameters(), 0.0003)
>>> d_optimizer = torch.optim.Adam(disc_model.parameters(), 0.0002)
```

We will be making a few small modifications to the training procedure. The `create_noise()` function for generating random input must change to output a tensor of four dimensions instead of a vector:

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

The `d_train()` function for training the discriminator doesn't need to reshape the input image:

```
>>> def d_train(x):
...     disc_model.zero_grad()
...     # Train discriminator with a real batch
...     batch_size = x.size(0)
...     x = x.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()
```

Next, we will alternate between the training of the generator and the discriminator for 100 epochs. After each epoch, we will generate some examples from a fixed noise input using the current generator model by calling the `create_samples()` function. The code is as follows:

```
>>> fixed_z = create_noise(batch_size, z_size, mode_z).to(device)
>>> epoch_samples = []
>>> torch.manual_seed(1)
>>> for epoch in range(1, num_epochs+1):
...     gen_model.train()
...     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))
...     print(f'Epoch {epoch:03d} | Avg Losses >>'
...           f' G/D {torch.FloatTensor(g_losses).mean():.4f}'
...           f'/{torch.FloatTensor(d_losses).mean():.4f}')
...     gen_model.eval()
...     epoch_samples.append(
...         create_samples(
...             gen_model, fixed_z
...         ).detach().cpu().numpy()
...     )
Epoch 001 | Avg Losses >> G/D 4.7016/0.1035
Epoch 002 | Avg Losses >> G/D 5.9341/0.0438
...
Epoch 099 | Avg Losses >> G/D 4.3753/0.1360
Epoch 100 | Avg Losses >> G/D 4.4914/0.1120
```

Finally, let's visualize the saved examples at some epochs to see how the model is learning and how the quality of synthesized examples changes over the course of learning:

```
>>> 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.show()
```

Figure 17.17 shows the results:

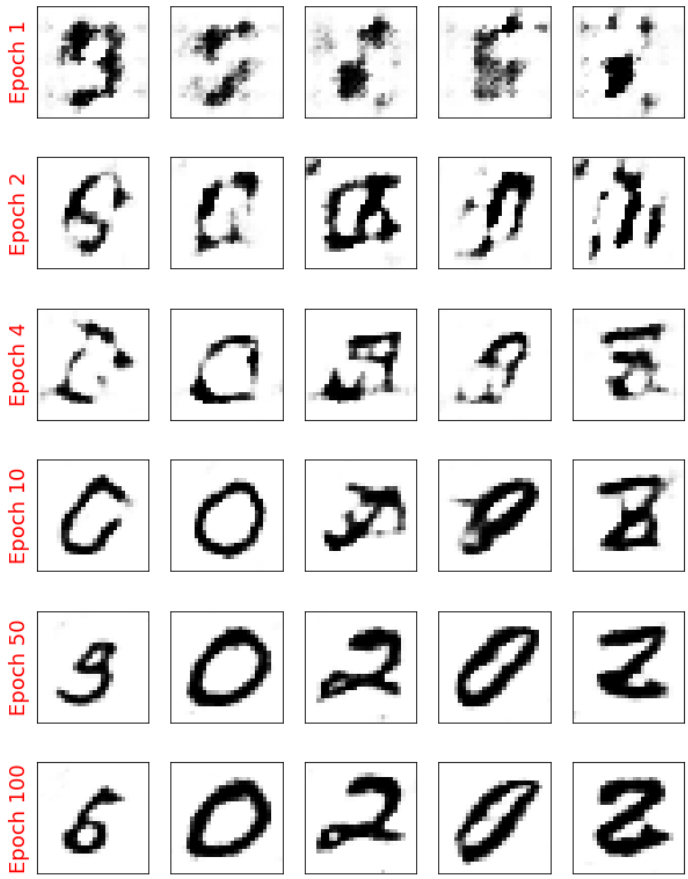

Figure 17.17: Generated images from the DCGAN

We used the same code to visualize the results as in the section on vanilla GAN. Comparing the new examples shows that DCGAN can generate images of a much higher quality.

You may wonder how we can evaluate the results of GAN generators. The simplest approach is visual assessment, which involves evaluating the quality of the synthesized images in the context of the target domain and the project objective. Furthermore, there have been several more sophisticated evaluation methods proposed that are less subjective and less limited by domain knowledge. For a detailed survey, see *Pros and Cons of GAN Evaluation Measures: New Developments* (`https://arxiv.org/abs/2103.09396`). The paper summarizes generator evaluation into qualitative and quantitative measures.

There is a theoretical argument that training the generator should seek to minimize the dissimilarity between the distribution observed in the real data and the distribution observed in synthesized examples. Hence our current architecture would not perform very well when using cross-entropy as a loss function.

In the next subsection, we will cover WGAN, which uses a modified loss function based on the so-called Wasserstein-1 (or earth mover's) distance between the distributions of real and fake images for improving the training performance.

Dissimilarity measures between two distributions

We will first see different measures for computing the divergence between two distributions. Then, we will see which one of these measures is already embedded in the original GAN model. Finally, switching this measure in GANs will lead us to the implementation of a WGAN.

As mentioned at the beginning of this chapter, the goal of a generative model is to learn how to synthesize new samples that have the same distribution as the distribution of the training dataset. Let $P(x)$ and $Q(x)$ represent the distribution of a random variable, x, as shown in the following figure.

First, let's look at some ways, shown in *Figure 17.18*, that we can use to measure the dissimilarity between two distributions, P and Q:

Measures	Formulation
Total variation (TV)	$TV(P,Q) = \sup_x \lvert P(x) - Q(x) \rvert$
Kullback-Leibler (KL) divergence	$KL(P\lvert\lvert Q) = \int P(x) \log \frac{P(x)}{Q(x)} dx$
Jensen-Shannon (JS) divergence	$JS(P,Q) = \frac{1}{2}\left(KL\left(P\lvert\lvert \frac{P+Q}{2}\right) + KL\left(Q\lvert\lvert \frac{P+Q}{2}\right)\right)$
Earth mover's (EM) distance	$EM(P,Q) = \inf_{\gamma \in \Pi(P,Q)} E_{(u,v)\in\gamma}(\lVert u - v \rVert)$

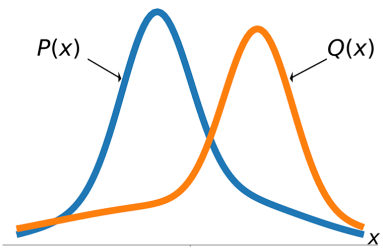

Figure 17.18: Methods to measure the dissimilarity between distributions P and Q

The function supremum, *sup*(*S*), used in the **total variation** (**TV**) measure, refers to the smallest value that is greater than all elements of *S*. In other words, *sup*(*S*) is the least upper bound for *S*. Vice versa, the infimum function, *inf*(*S*), which is used in EM distance, refers to the largest value that is smaller than all elements of *S* (the greatest lower bound).

Let's gain an understanding of these measures by briefly stating what they are trying to accomplish in simple words:

- The first one, TV distance, measures the largest difference between the two distributions at each point.
- The EM distance can be interpreted as the minimal amount of work needed to transform one distribution into the other. The infimum function in the EM distance is taken over $\Pi(P, Q)$, which is the collection of all joint distributions whose marginals are P or Q. Then, $\gamma(u, v)$ is a transfer plan, which indicates how we redistribute the earth from location u to v, subject to some constraints for maintaining valid distributions after such transfers. Computing EM distance is an optimization problem by itself, which is to find the optimal transfer plan, $\gamma(u, v)$.
- The **Kullback-Leibler** (**KL**) and **Jensen-Shannon** (**JS**) divergence measures come from the field of information theory. Note that KL divergence is not symmetric, that is, $KL(P\|Q) \neq KL(Q\|P)$ in contrast to JS divergence.

The dissimilarity equations provided in *Figure 17.18* correspond to continuous distributions but can be extended for discrete cases. An example of calculating these different dissimilarity measures with two simple discrete distributions is illustrated in *Figure 17.19*:

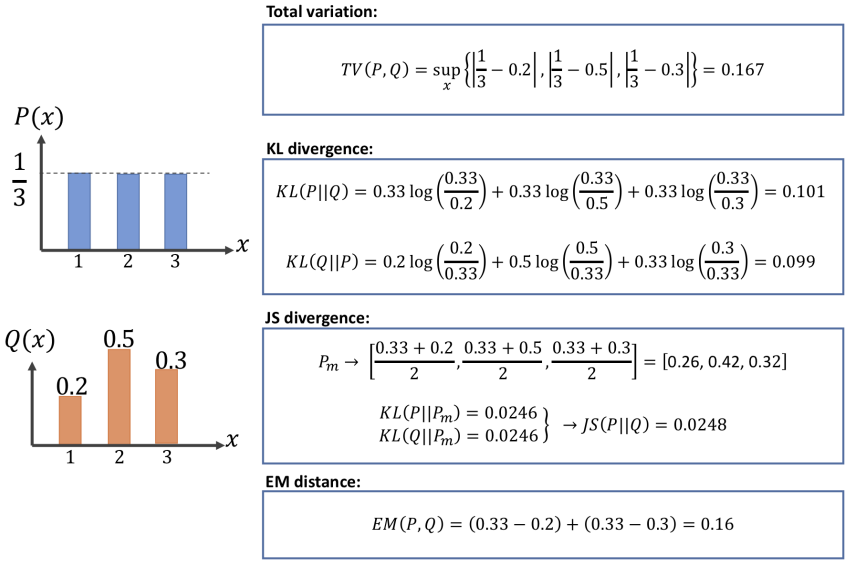

Figure 17.19: An example of calculating the different dissimilarity measures

Note that, in the case of the EM distance, for this simple example, we can see that $Q(x)$ at $x = 2$ has the excess value of $0.5 - \frac{1}{3} = 0.166$, while the value of Q at the other two x's is below 1/3. Therefore, the minimal amount of work is when we transfer the extra value at $x = 2$ to $x = 1$ and $x = 3$, as shown in *Figure 17.19*. For this simple example, it's easy to see that these transfers will result in the minimal amount of work out of all possible transfers. However, this may be infeasible to do for more complex cases.

The relationship between KL divergence and cross-entropy

KL divergence, $KL(P\|Q)$, measures the relative entropy of the distribution, P, with respect to a reference distribution, Q. The formulation for KL divergence can be extended as:

$$KL(P\|Q) = -\int P(x)\log\big(Q(x)\big)\,dx - \left(-\int P(x)\log\big(P(x)\big)\right)$$

Moreover, for discrete distributions, KL divergence can be written as:

$$KL(P\|Q) = -\sum_i P(x_i)\log\left(\frac{P(x_i)}{Q(x_i)}\right)$$

which can be similarly extended as:

$$KL(P\|Q) = -\sum_i P(x_i)\log\big(Q(x_i)\big) - \left(-\sum_i P(x_i)\log\big(P(x_i)\big)\right)$$

Based on the extended formulation (either discrete or continuous), KL divergence is viewed as the cross-entropy between P and Q (the first term in the preceding equation) subtracted by the (self-) entropy of P (second term), that is, $KL(P\|Q) = H(P,Q) - H(P)$.

Now, going back to our discussion of GANs, let's see how these different distance measures are related to the loss function for GANs. It can be mathematically shown that the loss function in the original GAN indeed *minimizes the JS divergence between the distribution of real and fake examples*. But, as discussed in an article by *Martin Arjovsky* and colleagues (*Wasserstein Generative Adversarial Networks*, `http://proceedings.mlr.press/v70/arjovsky17a/arjovsky17a.pdf`), JS divergence has problems training a GAN model, and therefore, in order to improve the training, the researchers proposed using the EM distance as a measure of dissimilarity between the distribution of real and fake examples.

What is the advantage of using EM distance?

To answer this question, we can consider an example that was given in the previously mentioned article by Martin Arjovsky and colleagues. To put it in simple words, assume we have two distributions, P and Q, which are two parallel lines. One line is fixed at $x = 0$ and the other line can move across the x-axis but is initially located at $x = \theta$, where $\theta > 0$.

It can be shown that the KL, TV, and JS dissimilarity measures are $KL(P\|Q) = +\infty$, $TV(P,Q) = 1$, and $JS(P,Q) = \frac{1}{2}\log 2$. None of these dissimilarity measures are a function of the parameter θ, and therefore, they cannot be differentiated with respect to θ toward making the distributions, P and Q, become similar to each other. On the other hand, the EM distance is $EM(P,Q) = |\theta|$, whose gradient with respect to θ exists and can push Q toward P.

Now, let's focus our attention on how EM distance can be used to train a GAN model. Let's assume P_r is the distribution of the real examples and P_g denotes the distributions of fake (generated) examples. P_r and P_g replace P and Q in the EM distance equation. As was mentioned earlier, computing the EM distance is an optimization problem by itself; therefore, this becomes computationally intractable, especially if we want to repeat this computation in each iteration of the GAN training loop. Fortunately, though, the computation of the EM distance can be simplified using a theorem called **Kantorovich-Rubinstein duality**, as follows:

$$W(P_r, P_g) = \sup_{\|f\|_L \leq 1} E_{u \in P_r}[f(u)] - E_{v \in P_g}[f(v)]$$

Here, the supremum is taken over all the *1-Lipschitz* continuous functions denoted by $\|f\|_L \leq 1$.

Lipschitz continuity

Based on 1-Lipschitz continuity, the function, f, must satisfy the following property:

$$|f(x_1) - f(x_2)| \leq |x_1 - x_2|$$

Furthermore, a real function, f:$R \to R$, that satisfies the property

$$|f(x_1) - f(x_2)| \leq K|x_1 - x_2|$$

is called **K-Lipschitz continuous**.

Using EM distance in practice for GANs

Now, the question is, how do we find such a 1-Lipschitz continuous function to compute the Wasserstein distance between the distribution of real (P_r) and fake (P_g) outputs for a GAN? While the theoretical concepts behind the WGAN approach may seem complicated at first, the answer to this question is simpler than it may appear. Recall that we consider deep NNs to be universal function approximators. This means that we can simply train an NN model to approximate the Wasserstein distance function. As you saw in the previous section, the simple GAN uses a discriminator in the form of a classifier. For WGAN, the discriminator can be changed to behave as a *critic*, which returns a scalar score instead of a probability value. We can interpret this score as how realistic the input images are (like an art critic giving scores to artworks in a gallery).

To train a GAN using the Wasserstein distance, the losses for the discriminator, D, and generator, G, are defined as follows. The critic (that is, the discriminator network) returns its outputs for the batch of real image examples and the batch of synthesized examples. We use the notations $D(\boldsymbol{x})$ and $D(G(\boldsymbol{z}))$, respectively.

Then, the following loss terms can be defined:

- The real component of the discriminator's loss:

$$L^D_{\text{real}} = -\frac{1}{N}\sum_i D(\boldsymbol{x}_i)$$

- The fake component of the discriminator's loss:

$$L^D_{\text{fake}} = \frac{1}{N}\sum_i D\big(G(\boldsymbol{z}_i)\big)$$

- The loss for the generator:

$$L^G = -\frac{1}{N}\sum_i D\big(G(\boldsymbol{z}_i)\big)$$

That will be all for the WGAN, except that we need to ensure that the 1-Lipschitz property of the critic function is preserved during training. For this purpose, the WGAN paper proposes clamping the weights to a small region, for example, [–0.01, 0.01].

Gradient penalty

In the paper by Arjovsky and colleagues, weight clipping is suggested for the 1-Lipschitz property of the discriminator (or critic). However, in another paper titled *Improved Training of Wasserstein GANs* by *Ishaan Gulrajani* and colleagues, 2017, which is freely available at `https://arxiv.org/pdf/1704.00028.pdf`, Ishaan Gulrajani and colleagues showed that clipping the weights can lead to exploding and vanishing gradients. Furthermore, weight clipping can also lead to capacity underuse, which means that the critic network is limited to learning only some simple functions, as opposed to more complex functions. Therefore, rather than clipping the weights, Ishaan Gulrajani and colleagues proposed **gradient penalty** (**GP**) as an alternative solution. The result is the **WGAN with gradient penalty** (**WGAN-GP**).

The procedure for the GP that is added in each iteration can be summarized by the following sequence of steps:

1. For each pair of real and fake examples $\left(\boldsymbol{x}^{[i]}, \tilde{\boldsymbol{x}}^{[i]}\right)$ in a given batch, choose a random number, $\alpha^{[i]}$, sampled from a uniform distribution, that is, $\alpha^{[i]} \in U(0,1)$.
2. Calculate an interpolation between the real and fake examples: $\breve{\boldsymbol{x}}^{[i]} = \alpha\boldsymbol{x}^{[i]} + (1-\alpha)\tilde{\boldsymbol{x}}^{[i]}$, resulting in a batch of interpolated examples.
3. Compute the discriminator (critic) output for all the interpolated examples, $D\left(\breve{\boldsymbol{x}}^{[i]}\right)$.
4. Calculate the gradients of the critic's output with respect to each interpolated example, that is, $\nabla_{\breve{\boldsymbol{x}}^{[i]}} D\left(\breve{\boldsymbol{x}}^{[i]}\right)$.
5. Compute the GP as:

$$L^D_{gp} = \frac{1}{N}\sum_i \left(\left\|\nabla_{\breve{\boldsymbol{x}}^{[i]}} D\left(\breve{\boldsymbol{x}}^{[i]}\right)\right\|_2 - 1\right)^2$$

The total loss for the discriminator is then as follows:

$$L^D_{\text{total}} = L^D_{\text{real}} + L^D_{\text{fake}} + \lambda L^D_{gp}$$

Here, λ is a tunable hyperparameter.

Implementing WGAN-GP to train the DCGAN model

We have already defined the helper function and class that create the generator and discriminator networks for DCGAN (`make_generator_network()` and `Discriminator()`). It is recommended to use layer normalization in WGAN instead of batch normalization. Layer normalization normalizes the inputs across features instead of across the batch dimension in batch normalization. The code to build the WGAN model is as follows:

```
>>> def make_generator_network_wgan(input_size, n_filters):
...     model = nn.Sequential(
...         nn.ConvTranspose2d(input_size, n_filters*4, 4,
...                            1, 0, bias=False),
...         nn.InstanceNorm2d(n_filters*4),
...         nn.LeakyReLU(0.2),
...
...         nn.ConvTranspose2d(n_filters*4, n_filters*2,
...                            3, 2, 1, bias=False),
...         nn.InstanceNorm2d(n_filters*2),
...         nn.LeakyReLU(0.2),
...
...         nn.ConvTranspose2d(n_filters*2, n_filters, 4,
...                            2, 1, bias=False),
...         nn.InstanceNorm2d(n_filters),
...         nn.LeakyReLU(0.2),
...
...         nn.ConvTranspose2d(n_filters, 1, 4, 2, 1, bias=False),
...         nn.Tanh()
...     )
...     return model
>>>
>>> class DiscriminatorWGAN(nn.Module):
...     def __init__(self, n_filters):
...         super().__init__()
...         self.network = nn.Sequential(
...             nn.Conv2d(1, n_filters, 4, 2, 1, bias=False),
...             nn.LeakyReLU(0.2),
...
```

```
...             nn.Conv2d(n_filters, n_filters*2, 4, 2, 1,
...                       bias=False),
...             nn.InstanceNorm2d(n_filters * 2),
...             nn.LeakyReLU(0.2),
...
...             nn.Conv2d(n_filters*2, n_filters*4, 3, 2, 1,
...                       bias=False),
...             nn.InstanceNorm2d(n_filters*4),
...             nn.LeakyReLU(0.2),
...
...             nn.Conv2d(n_filters*4, 1, 4, 1, 0, bias=False),
...             nn.Sigmoid()
...     )
...
...     def forward(self, input):
...         output = self.network(input)
...         return output.view(-1, 1).squeeze(0)
```

Now we can initiate the networks and their optimizers as follows:

```
>>> gen_model = make_generator_network_wgan(
...     z_size, n_filters
... ).to(device)
>>> disc_model = DiscriminatorWGAN(n_filters).to(device)
>>> g_optimizer = torch.optim.Adam(gen_model.parameters(), 0.0002)
>>> d_optimizer = torch.optim.Adam(disc_model.parameters(), 0.0002)
```

Next, we will define the function to compute the GP component as follows:

```
>>> from torch.autograd import grad as torch_grad
>>> def gradient_penalty(real_data, generated_data):
...     batch_size = real_data.size(0)
...
...     # Calculate interpolation
...     alpha = torch.rand(real_data.shape[0], 1, 1, 1,
...                        requires_grad=True, device=device)
...     interpolated = alpha * real_data + \
...                    (1 - alpha) * generated_data
...
...     # Calculate probability of interpolated examples
...     proba_interpolated = disc_model(interpolated)
...
```

```
...     # Calculate gradients of probabilities
...     gradients = torch_grad(
...         outputs=proba_interpolated, inputs=interpolated,
...         grad_outputs=torch.ones(proba_interpolated.size(),
...                                 device=device),
...         create_graph=True, retain_graph=True
...     )[0]
...
...     gradients = gradients.view(batch_size, -1)
...     gradients_norm = gradients.norm(2, dim=1)
...     return lambda_gp * ((gradients_norm - 1)**2).mean()
```

The WGAN version of discriminator and generator training functions are as follows:

```
>>> def d_train_wgan(x):
...     disc_model.zero_grad()
...
...     batch_size = x.size(0)
...     x = x.to(device)
...
...     # Calculate probabilities on real and generated data
...     d_real = disc_model(x)
...     input_z = create_noise(batch_size, z_size, mode_z).to(device)
...     g_output = gen_model(input_z)
...     d_generated = disc_model(g_output)
...     d_loss = d_generated.mean() - d_real.mean() + \
...              gradient_penalty(x.data, g_output.data)
...     d_loss.backward()
...     d_optimizer.step()
...     return d_loss.data.item()
>>>
>>> def g_train_wgan(x):
...     gen_model.zero_grad()
...
...     batch_size = x.size(0)
...     input_z = create_noise(batch_size, z_size, mode_z).to(device)
...     g_output = gen_model(input_z)
...
...     d_generated = disc_model(g_output)
...     g_loss = -d_generated.mean()
...
```

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

Then we will train the model for 100 epochs and record the generator output of a fixed noise input:

```
>>> epoch_samples_wgan = []
>>> lambda_gp = 10.0
>>> num_epochs = 100
>>> torch.manual_seed(1)
>>> critic_iterations = 5
>>> for epoch in range(1, num_epochs+1):
...     gen_model.train()
...     d_losses, g_losses = [], []
...     for i, (x, _) in enumerate(mnist_dl):
...         for _ in range(critic_iterations):
...             d_loss = d_train_wgan(x)
...         d_losses.append(d_loss)
...         g_losses.append(g_train_wgan(x))
...
...     print(f'Epoch {epoch:03d} | D Loss >>'
...           f' {torch.FloatTensor(d_losses).mean():.4f}')
...     gen_model.eval()
...     epoch_samples_wgan.append(
...         create_samples(
...             gen_model, fixed_z
...         ).detach().cpu().numpy()
...     )
```

Finally, let's visualize the saved examples at some epochs to see how the WGAN model is learning and how the quality of synthesized examples changes over the course of learning. The following figure shows the results, which demonstrate slightly better image quality than what the DCGAN model generated:

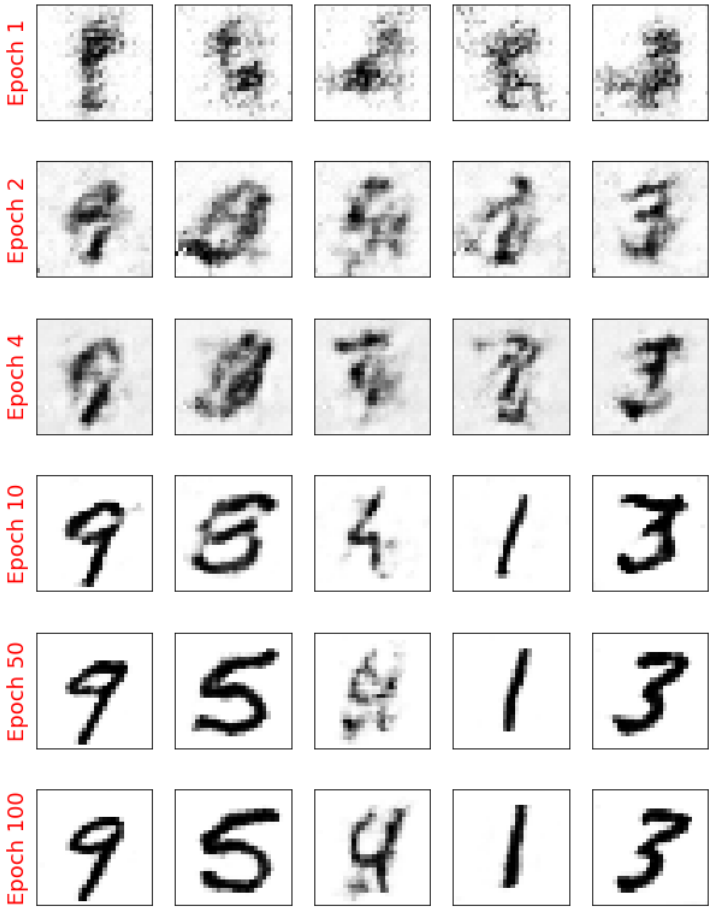

Figure 17.20: Generated images using WGAN

Mode collapse

Due to the adversarial nature of GAN models, it is notoriously hard to train them. One common cause of failure in training GANs is when the generator gets stuck in a small subspace and learns to generate similar samples. This is called **mode collapse**, and an example is shown in *Figure 17.21*.

The synthesized examples in this figure are not cherry-picked. This shows that the generator has failed to learn the entire data distribution, and instead, has taken a lazy approach focusing on a subspace:

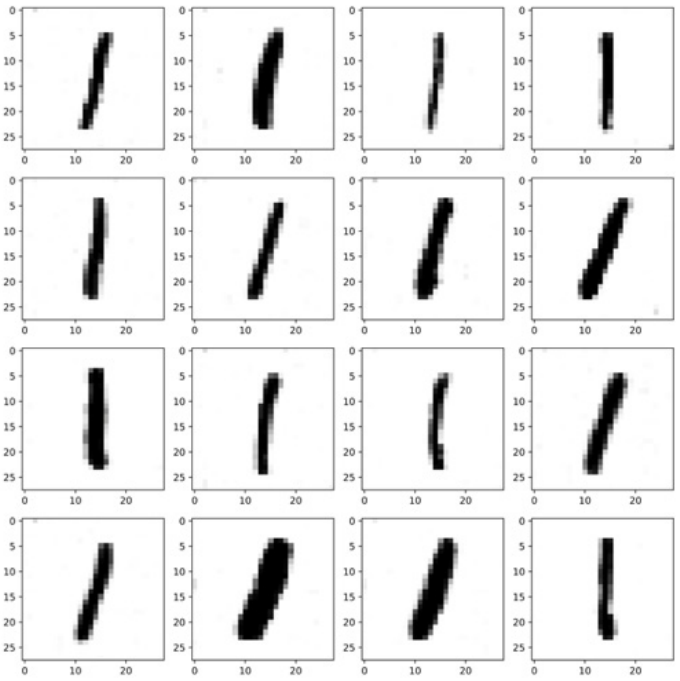

Figure 17.21: An example of mode collapse

Besides the vanishing and exploding gradient problems that we saw previously, there are some further aspects that can also make training GAN models difficult (indeed, it is an art). Here are a few suggested tricks from GAN artists.

One approach is called **mini-batch discrimination**, which is based on the fact that batches consisting of only real or fake examples are fed separately to the discriminator. In mini-batch discrimination, we let the discriminator compare examples across these batches to see whether a batch is real or fake. The diversity of a batch consisting of only real examples is most likely higher than the diversity of a fake batch if a model suffers from mode collapse.

Another technique that is commonly used for stabilizing GAN training is *feature matching*. In feature matching, we make a slight modification to the objective function of the generator by adding an extra term that minimizes the difference between the original and synthesized images based on intermediate representations (feature maps) of the discriminator. We encourage you to read more about this technique in the original article by *Ting-Chun Wang* and colleagues, titled *High Resolution Image Synthesis and Semantic Manipulation with Conditional GANs,* which is freely available at `https://arxiv.org/pdf/1711.11585.pdf`.

During the training, a GAN model can also get stuck in several modes and just hop between them. To avoid this behavior, you can store some old examples and feed them to the discriminator to prevent the generator from revisiting previous modes. This technique is referred to as *experience replay*. Furthermore, you can train multiple GANs with different random seeds so that the combination of all of them covers a larger part of the data distribution than any single one of them.

Other GAN applications

In this chapter, we mainly focused on generating examples using GANs and looked at a few tricks and techniques to improve the quality of synthesized outputs. The applications of GANs are expanding rapidly, including in computer vision, machine learning, and even other domains of science and engineering. A nice list of different GAN models and application areas can be found at `https://github.com/hindupuravinash/the-gan-zoo`.

It is worth mentioning that we covered GANs in an unsupervised fashion; that is, no class label information was used in the models that were covered in this chapter. However, the GAN approach can be generalized to semi-supervised and supervised tasks, as well. For example, the **conditional GAN** (**cGAN**) proposed by *Mehdi Mirza* and *Simon Osindero* in the paper *Conditional Generative Adversarial Nets*, 2014 (`https://arxiv.org/pdf/1411.1784.pdf`) uses the class label information and learns to synthesize new images conditioned on the provided label, that is, $\tilde{\boldsymbol{x}} = G(\boldsymbol{z}|y)$—applied to MNIST. This allows us to generate different digits in the range 0-9 selectively. Furthermore, conditional GANs allow us to do image-to-image translation, which is to learn how to convert a given image from a specific domain to another. In this context, one interesting work is the Pix2Pix algorithm, published in the paper *Image-to-Image Translation with Conditional Adversarial Networks* by *Philip Isola* and colleagues, 2018 (`https://arxiv.org/pdf/1611.07004.pdf`). It is worth mentioning that in the Pix2Pix algorithm, the discriminator provides the real/fake predictions for multiple patches across the image as opposed to a single prediction for an entire image.

CycleGAN is another interesting GAN model built on top of the cGAN, also for image-to-image translation. However, note that in CycleGAN, the training examples from the two domains are unpaired, meaning that there is no one-to-one correspondence between inputs and outputs. For example, using a CycleGAN, we could change the season of a picture taken in summer to winter. In the paper *Unpaired Image-to-Image Translation Using Cycle-Consistent Adversarial Networks* by *Jun-Yan Zhu* and colleagues, 2020 (`https://arxiv.org/pdf/1703.10593.pdf`), an impressive example shows horses converted into zebras.

Summary

In this chapter, you first learned about generative models in deep learning and their overall objective: synthesizing new data. We then covered how GAN models use a generator network and a discriminator network, which compete with each other in an adversarial training setting to improve each other. Next, we implemented a simple GAN model using only fully connected layers for both the generator and the discriminator.

We also covered how GAN models can be improved. First, you saw a DCGAN, which uses deep convolutional networks for both the generator and the discriminator. Along the way, you also learned about two new concepts: transposed convolution (for upsampling the spatial dimensionality of feature maps) and BatchNorm (for improving convergence during training).

We then looked at a WGAN, which uses the EM distance to measure the distance between the distributions of real and fake samples. Finally, we talked about the WGAN with GP to maintain the 1-Lipschitz property instead of clipping the weights.

In the next chapter, we will look at graph neural networks. Previously, we have been focused on tabular and image datasets. In contrast, graph neural networks are designed for graph-structured data, which allows us to work with datasets that are ubiquitous in social sciences, engineering, and biology. Popular examples of graph-structure data include social network graphs and molecules consisting of atoms connected by covalent bonds.

Join our book's Discord space

Join our Discord community to meet like-minded people and learn alongside more than 2000 members at:
`https://packt.link/MLwPyTorch`