9 CycleGAN

This chapter covers

- Expanding on the idea of Conditional GANs by conditioning on an entire image
- Exploring one of the most powerful and complex GAN architectures: CycleGAN
- Presenting an object-oriented design of GANs and the architecture of its four main components
- Implementing a CycleGAN to run a conversion of apples to oranges

Finally, a technological breakthrough of almost universal appeal, seeing as everyone seems to love comparing apples to oranges. In this chapter, you will learn how! But this is no small feat, so we will need at least *two* sets of Discriminators and *two* Generators to achieve this. That obviously complicates the architecture, so we will have to spend more time discussing it, but at the very least, it is a great point to start thinking in a fully object-oriented programming (OOP) way.

9.1 Image-to-image translation

One fascinating area of GANs' application that we touched on at the end of the previous chapter is *image-to-image translation*. In this use, GANs have been massively successful—in video, static images, or even style transfer. Indeed, GANs have been at the forefront of many of these applications as they enable almost a new class of uses. Because of their visual nature, the more successful GAN variants typically make their rounds on YouTube and Twitter, so if you have not seen these videos, we encourage you to check them out by searching for *pix2pix*, *CycleGAN*, or *vid2vid*.

This type of translation in practice means that our input to the Generator is a picture, because we need our Generator (translator) to start from this image. In other words, we are mapping an image from one domain to another. Previously, the latent vector seeding the generation was typically a somewhat uninterpretable vector. Now we are swapping that for an input image.

A good way to think of image-to-image translation is as a special case of the Conditional GAN. However, in this case, we are conditioning on a complete image (rather than just a class)—typically of the same dimensionality as the output image—that is then provided to the network as a kind of a label (presented in chapter 8). One of the first famous examples in this space was an image-translation work coming out of the University of California, Berkeley, as shown in figure 9.1.

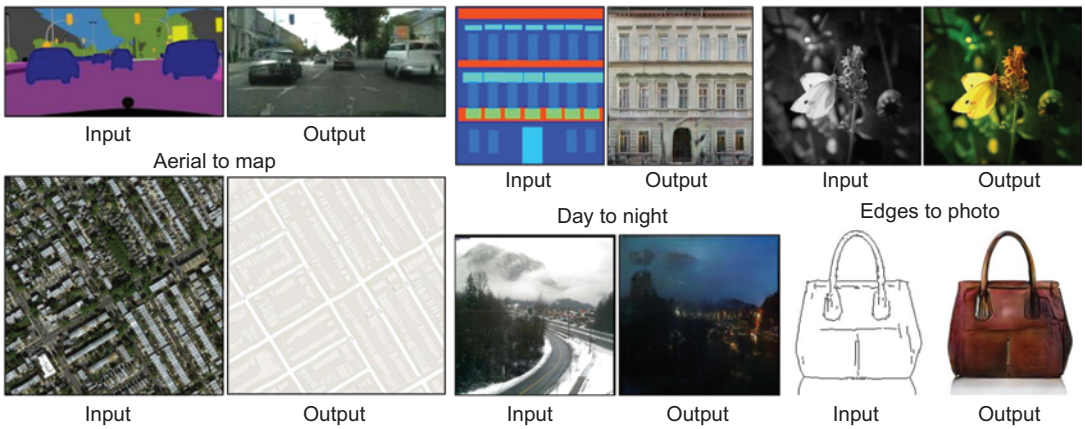

Figure 9.1 Conditional GANs provide a powerful framework for image translation that performs well across many domains.
(Source: "Image-to-Image Translation with Conditional Adversarial Networks," by Phillip Isola, https://github.com/phillipi/pix2pix.)

As you can see, we can map from any of the following:

- From semantic labels (for example, drawing blue where a car should be and purple where a road should be) to photorealistic images of streets
- From satellite images to a view like the one in Google Maps
- From day images to night images

- From black-and-white to color
- From outlines to synthesized fashion items

The idea is clearly powerful and versatile; however, the issue lies with the need for paired data. From chapter 8, you understand that we need labels for the Conditional GAN. Because in this case we are using another image as a label, the mapping does not make sense unless we're mapping to the corresponding image—the exact same image, except in the other domain.

So, the night image needs to be taken from exactly the same place as the day image. The fashion item's outline needs to have the exact match of a fully colored/synthesized item in the training set in the other domain. In other words, during training, the GAN needs to have access to corresponding labels of the items in the original domain.

This is typically done—for example, in the case of black-and-white images—by first taking loads of colored pictures, applying the B&W filter on all of them, and then using the unmodified image as one domain and the B&W-filtered images as the other. This ensures that we have the corresponding images in both domains. Then we can apply the trained GAN anywhere, but if we do not have an easy way of generating these "perfect" pairs, we are out of luck!

9.2 Cycle-consistency loss: There and back aGAN

The genius insight of this UC Berkeley group was that we do not, in fact, need perfect pairs.[1] Instead, we simply complete the cycle: we translate from one domain to another and then back again. For example, we go from summer picture (domain A) of a park to a winter one (domain B) and then back again to summer (domain A). Now we have essentially created a cycle, and, ideally, the original picture (a) and the reconstructed picture ($\hat{a}$) are the same. If they are not, we can measure their loss on a pixel level, thereby getting the first loss of our CycleGAN: *cycle-consistency loss*, which is depicted in figure 9.2.

A common analogy is thinking about the process of *back-translation*—a sentence in Chinese that is translated to English and then back again to Chinese should give back the same sentence. If not, we can measure the cycle-consistency loss by how much the first and the third sentences differ.

To be able to use the cycle-consistency loss, we need to have two Generators: one translating from A to B, called G_{AB}, sometimes referred to as simply G, and then another one translating from B to A, called G_{BA}, referred to as F for brevity. There are technically two losses—forward cycle-consistency loss and backward cycle-consistency loss—but because all they mean is that $\hat{a} = F(G(a)) \approx a$ as well as $\hat{b} = G(F(b)) \approx b$, you may think of these as essentially the same, but off by one.

[1] See "Unpaired Image-to-Image Translation Using Cycle-Consistent Adversarial Networks," by Jun-Yan Zhu et al., 2017, https://arxiv.org/pdf/1703.10593.pdf.

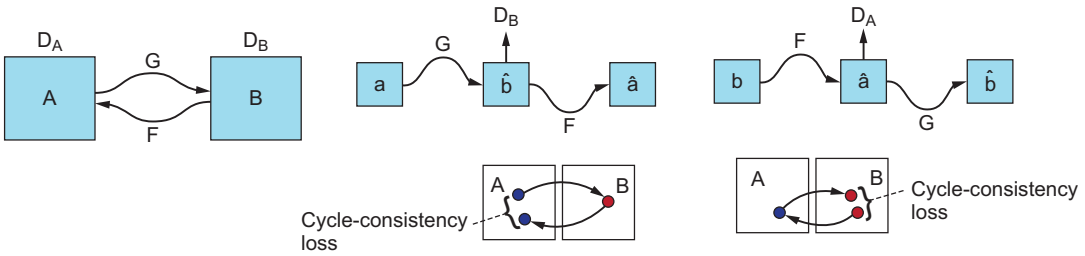

Figure 9.2 Because the loss works both ways, we can now reproduce not just images from summer to winter, but also from winter to summer. If *G* is our Generator from A to B, and *F* is our Generator from B to A, then $\hat{a} = F(G(a)) \approx a$.

(Source: Jun-Yan Zhu et al., 2017, https://arxiv.org/pdf/1703.10593.pdf.

9.3 Adversarial loss

In addition to the cycle-consistency loss, we still have the *adversarial loss*. Every translation with a Generator G_{AB} has a corresponding Discriminator D_B, and G_{BA} has Discriminator D_A. The way to think about it is that we are always testing, when translating *to* domain A, whether the picture looks real; hence we use D_A and vice versa.

This is the same idea as with simpler architectures, but now, because of the two losses, we have two Discriminators. We need to make sure that not only the translation from apple to orange looks real, but also that the translation from our estimated orange back to reconstructed apple looks real. Recall that the adversarial loss ensures that the images look real, and as a result, it is still key for the CycleGAN to work. Hence adversarial loss is presented as second. The first Discriminator in the cycle is especially important—otherwise, we'd simply get noise that would help the GAN memorize what it should reconstruct.[2]

9.4 Identity loss

The idea of *identity loss* is simple: we want to enforce that CycleGAN preserves the overall color structure (or *temperature*) of the picture. So we introduce a regularization term that helps us keep the tint of the picture consistent with the original image. Imagine this as a way of ensuring that even after applying many filters onto your image, you still can recover the original image.

This is done by feeding the images already in domain A to the Generator from B to A (G_{BA}), because the CycleGAN should understand that they are already in the correct domain. In other words, we penalize unnecessary changes to the image: if we feed in a zebra and are trying to "zebrafy" an image, we get the same zebra back, as there is nothing to do.[3] Figure 9.3 illustrates the effects of identity loss.

---

[2] In practice, this is a little bit more complicated and would depend on, for example, whether you include both forward and backward cyclical loss. But you may use this as a mental model for how to think of the importance of the adversarial loss—remembering that we have both mappings A-B-A and B-A-B, so both Discriminators get to be the first one at some point.

[3] Jun Yan Zhu et al., 2017, https://arxiv.org/pdf/1703.10593.pdf. More at http://mng.bz/loE8.

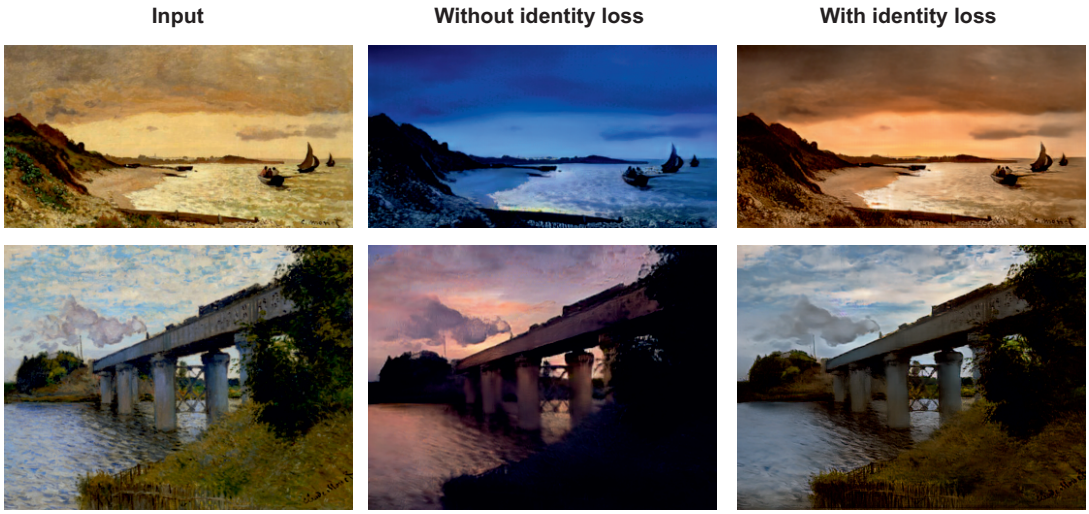

Figure 9.3 A picture is worth a thousand words to clarify the effects of identity loss: there is a clear tint in the cases without identity loss, and since there seems to be no reason for it, so we try to penalize this behavior. Even in black and white, you should be able to see the difference. However, to see the full extent of it, check out the full-color version online.

Even though identity loss is not, strictly speaking, required for the CycleGAN to work, we include it for completeness. Both our implementation and the CycleGAN authors' latest implementation contain it, because frequently this adjustment leads to empirically better results and enforces a constraint that seems reasonable. But even the CycleGAN paper itself mentions it only briefly as a seeming ex-post justification, so we do not cover it extensively.

Table 9.1 summarizes the losses you've learned about in this chapter.

Table 9.1 Losses

	Calculation	Measures	Ensures
Adversarial loss	$\mathcal{L}_{GAN}(G,D_B,B,A)$ $= E_{b\sim p(b)}[logD_B(b)]$ $+ E_{a\sim p(a)}[log(1-D_B(G_{AB}(a))]$ (This is just the good old NS-GAN presented in chapter 5.)	As in previous cases, the loss measures two terms: first is the likelihood of a given image being the real one rather than the translated image. Second is the part where the Generator may get to fool the Discriminator. Note that this formulation is only for D_B, with equivalent D_A that comes into the final loss.	That the translated images look real, sharp, and indistinguishable from the real ones.

Table 9.1 Losses *(continued)*

	Calculation	Measures	Ensures
Cycle-consistency loss: forward pass	Difference between a and $\hat{a}$ (denoted by $\|\hat{a} - a\|_1$[a])	The difference between the images from the original domain a and the twice-translated images $\hat{a}$.	That the original image and the twice-translated image are the same. If this fails, we may not have a coherent mapping A-B-A.
Cycle-consistency loss: backward pass	$\|\hat{b} - b\|_1$	The difference between the images from the original domain b and the twice-translated images $\hat{b}$.	That the original image and the twice-translated image are the same. If this fails, we may not have a coherent mapping B-A-B.
Overall loss	$\mathcal{L} = \mathcal{L}_{GAN}(G,D_B,A,B)$ $+ \mathcal{L}_{GAN}(F,D_A,B,A)$ $+ \lambda\mathcal{L}_{cyc}(G,F)$	All of the four losses combined (2× adversarial because of two Generators) plus cyclical loss: forward and backward in one term.	That the overall translation is photorealistic and makes sense (provides matching pictures).
Identity loss (outside the overall loss, for consistency with the CycleGAN paper notation)	$\mathcal{L}_{identity} =$ $= E_{a\sim p(a)}[\|\| G_{BA}(a) - a \|\|]$ $+ E_{b\sim p(b)}[\|\| G_{AB}(b) - b \|\|]$	The difference between the image in B and $G_{AB}(b)$ and vice versa.	That the CycleGAN changes parts of the image only when it needs to.

a. This notation may be unfamiliar to some, but it represents the L1 norm between the two items. For simplicity, you may think of this as for each pixel, an absolute difference between it and the corresponding pixel on the reconstructed image.

9.5 Architecture

The CycleGAN setup builds directly on the CGAN architecture and is, in essence, two CGANs joined together—or, as the CycleGAN authors themselves point out, an autoencoder. Recall from chapter 2 that we had an input image x and the reconstructed image x^*, which was the result of reconstruction after being fed through the latent space z; see figure 9.4.

To translate this diagram into the CycleGAN's world, a is an image in the A domain, b is an image in B, and $\hat{a}$ is reconstructed A. In CycleGAN's case, however, we are dealing with a latent space—step 2—of equal dimensionality. It just happens to be another meaningful domain (B) that the CycleGAN has to find. Even with the autoencoder, the latent space was just another domain, though it was not as easily interpretable.

Compared to what we know from chapter 2, the main new concept is the introduction of the adversarial losses. These and many other mixtures of autoencoders and GANs are an active area of research in themselves! So that is also a good area for interested researchers. But for now, think of the two mappings as two autoencoders: $F(G(a))$

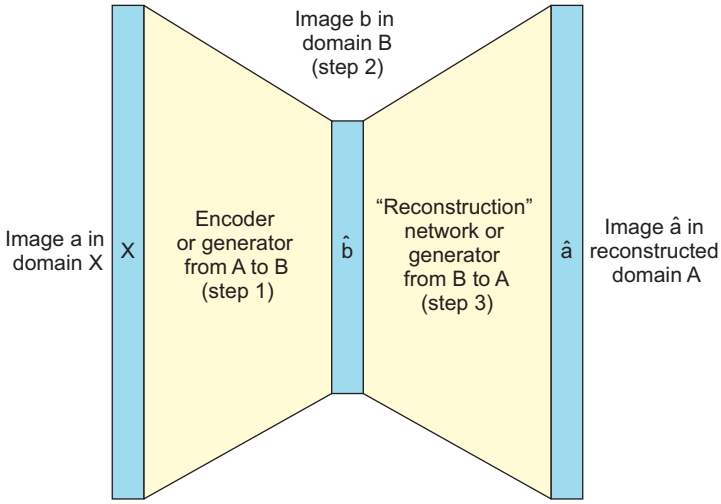

Figure 9.4 In this image of an autoencoder from chapter 2, we used the analogy of compressing (step 1) a human concept into a more compact written form in a letter (step 2) and then expanding this concept out to the (imperfect) idea of the same notion in someone else's head (step 3).

and $G(F(b))$. We take the basic idea of the autoencoder—including a kind of *explicit* loss function as substituted by the cycle-consistency loss—and add Discriminators to it. The two Discriminators, one at each step, ensure that both translations (including into the kind of *latent space*) look like real images in their respective domains.

9.5.1 CycleGAN architecture: building the network

Before we jump into the actual implementation of the CycleGAN, let's briefly look at the overall simplified implementation depicted in figure 9.5. There are two flows: in the top diagram, the flow A-B-A starts from an image in domain A, and in the bottom diagram, the flow B-A-B starts with an image in domain B.

The image then follows two paths: it is (1) fed to the Discriminator to get our decision as to whether it is real or not, and (2) (i) fed to the Generator to translate it to B, then (ii) evaluated by the Discriminator B to see if it looks real in domain B, and eventually (iii) translated back to A to allow us to measure the cyclic loss.

The bottom image is basically an *off-by-one* cycle of the top image and follows all the same fundamental steps. We'll use the apple2orange dataset, but many other datasets are available, including the famous horse2zebra dataset, which you can easily use by making a slight modification to the code and downloading the data by using the bash script provided.

To summarize figure 9.5 in another representation for further clarity, table 9.2 reviews all four major networks.

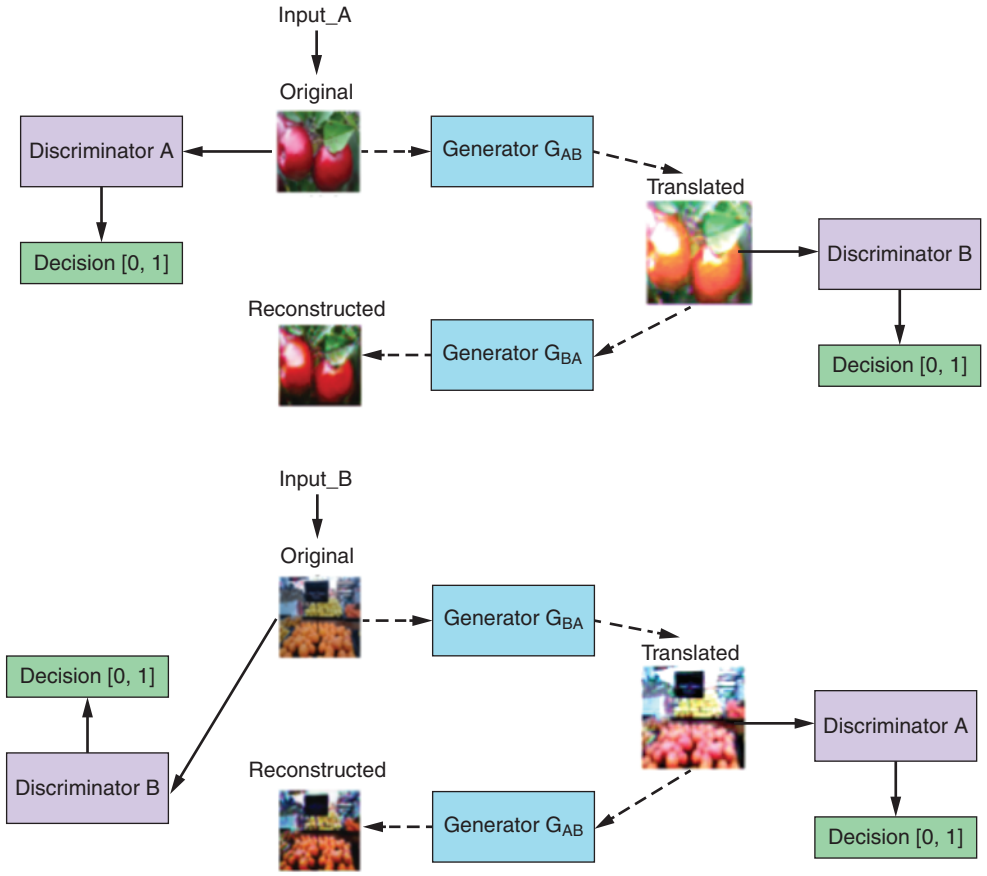

Figure 9.5 In this simplified architecture of the CycleGAN, we start with the input image, which either (1) goes to the Discriminator for evaluation or (2) is translated to one domain, evaluated by the other Discriminator, and then translated back.
(Source: "Understanding and Implementing CycleGAN in TensorFlow," by Hardik Bansal and Archit Rathore, 2017, https://hardikbansal.github.io/CycleGANBlog/.)

Table 9.2 Networks

	Input	Output	Goal
Generator: from A to B	We load either a real picture from A or a translation from B to A.	We translate it to domain B.	Try to create realistic-looking images in domain B.
Generator: from B to A	We load either a real picture from B or a translation from A to B.	We translate it to domain A.	Try to create realistic-looking images in domain A.
Discriminator A	We provide a picture in the A domain—either translated or real.	The probability that the picture is real.	Try to not get fooled by the Generator from B to A.
Discriminator B	We provide a picture in the B domain—either translated or real.	The probability that the picture is real.	Try to not get fooled by the Generator from A to B.

9.5.2 Generator architecture

Figure 9.6 shows the architecture of the Generator. We have re-created the diagram by using the variable names from our code and included the shapes for your benefit. This is an example of a *U-Net* architecture, because when you draw it in a way that each resolution gets its own level, the network looks like a U.

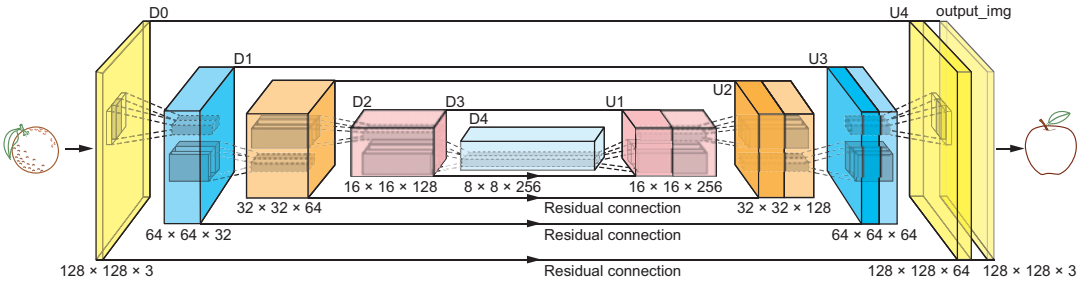

Figure 9.6 Architecture of the Generator. The generator itself has a *contraction path* (d0 to d3) and *expanding path* (u1 to u4). The contraction and expanding paths are sometimes referred to as *encoder* and *decoder*, respectively.

A couple of things to note here:

- We are using standard convolutional layers in the encoder.
- From those, we create *skip connections* so that the information has an easier time propagating through the network. In the figure, this is denoted by the outlines and color-coding between the d0 to d3 and u1 to u4, respectively. You can see that half of the blocks in the decoder are coming from those skip connections (notice double the number of feature maps!).[4]
- The decoder uses deconvolutional layers with one final convolutional layer to upscale the image into the equivalent size of the original image.

The autoencoder is a useful teaching tool for the architecture of the Generator alone as well, because the Generator has an encoder-decoder architecture:

- *Encoder*—Step 1 from figure 9.4: these are the convolutional layers that reduce the resolution of each feature map (*layer* or *slice*). This is the contraction path (d0 to d3).
- *Decoder*—Step 3 from figure 9.4: these are the *deconvolutional* layers (transposed convolutions) that upscale the image back to 128 × 128. This is the expansion path (u1 to u4).

[4] As you will see, this just means we concatenate the entire block/tensor to the equivalently colored tensor in the decoder part of the Generator.

To clarify, the autoencoder model here is useful in two ways. First, the overall CycleGAN architecture can be viewed as training two autoencoders.[5] Second, the U-Net itself has parts referred to as *encoder* and *decoder*.

You may also be a bit puzzled by the downscaling and the subsequent upscaling, but this is just so that we compress the image to the most meaningful representation, but at the same time are able to add back all the detail. It's the same reasoning as with the autoencoder, except now we also have a path to remember the nuances. This architecture—the *U-Net architecture*—has just been empirically shown in several domains as better performing on various segmentation tasks. The key idea is that although during downsampling we can focus on classification and understanding of large regions, including higher-resolution skip connections preserves the detail that can then be accurately segmented.

In our implementation of CycleGAN, we'll use the U-Net architecture with skip connections as shown in figure 9.6, which is more readable. However, many CycleGAN implementations use the ResNet architecture, which you can implement yourself with a bit more work.

> **NOTE** The main advantage of ResNet is that it uses fewer parameters and introduces a step in the middle called *transformer*, which has residual connections in lieu of our encoder-decoder skip connections.

Based on our testing, at least on the dataset used, the apple2orange results remain the same. Instead of explicitly defining the transformer, we provide skip connections (as used in the diagram) from the convolutional to the deconvolutional layers. We will mention these similarities again in code. For now, just remember that.

9.5.3 Discriminator architecture

The CycleGAN's Discriminator is based on the PatchGAN architecture—we will dive into the technical details in the code section. One thing that may be confusing is that we do not get a single float as an output of this Discriminator, but rather a set of single-channel values that may be thought of as a set of mini-discriminators that we then average together.

Ultimately, this allows the design of the CycleGAN to be fully convolutional, meaning that it can scale relatively easily to higher resolutions. Indeed, in the examples of translating video games to reality or vice versa, the CycleGAN authors have used an upscaled version of the CycleGAN, with only minor modifications thanks to the fully convolutional design. Other than that, the Discriminator should be a relatively straightforward implementation of the Discriminators you have seen before, except there are now two of them.

[5] See Jun-Yan Zhu et al., 2017, https://arxiv.org/pdf/1703.10593.pdf.

9.6 Object-oriented design of GANs

We have always used objects in TensorFlow and object-oriented programming (OOP) in our code, but we have usually treated the architectures more functionally, because they were generally simple. In the CycleGAN's case, the architecture is complex, and as a result, we need a structure that allows us to keep accessing the original attributes and methods that we have defined. As a result, we will write out the CycleGAN as a Python class of its own with methods to build the Generator and Discriminator, and run the training.

9.7 Tutorial: CycleGAN

In this tutorial, we'll use the Keras-GAN implementation and use Keras with a TensorFlow backend.[6] Tested as late as Keras 2.2.4 and TensorFlow 1.12.0, `Keras_contrib` was installed from the hash 46fcdb9384b3bc9399c651b2b43640aa54098e64. This time, we have to use a different dataset (also to show you that despite our joke from chapter 2, we *do know* other datasets). But for educational purposes, we will keep using one of the simpler datasets—apple2orange. Let's jump right into it by doing all our usual imports, as shown in the following listing.

Listing 9.1 Import all the things

```
from __future__ import print_function, division
import scipy
from keras.datasets import mnist
from keras_contrib.layers.normalization import InstanceNormalization
from keras.layers import Input, Dense, Reshape, Flatten, Dropout, Concatenate
from keras.layers import BatchNormalization, Activation, ZeroPadding2D
from keras.layers.advanced_activations import LeakyReLU
from keras.layers.convolutional import UpSampling2D, Conv2D
from keras.models import Sequential, Model
from keras.optimizers import Adam
import datetime
import matplotlib.pyplot as plt
import sys
from data_loader import DataLoader
import numpy as np
import os
```

As promised, we'll use the object-oriented style of programming. In the following listing, we create a CycleGAN class with all the initializing parameters, including the data loader. The data loader is defined in the GitHub repository for our book. It simply loads the preprocessed data.

[6] See the Keras-GAN GitHub repository by Erik Linder-Norén, 2017, https://github.com/eriklindernoren/Keras-GAN.

Listing 9.2 Starting the CycleGAN class

```
class CycleGAN():
    def __init__(self):
        self.img_rows = 128
        self.img_cols = 128
        self.channels = 3
        self.img_shape = (self.img_rows, self.img_cols, self.channels)

        self.dataset_name = 'apple2orange'
        self.data_loader = DataLoader(dataset_name=self.dataset_name,
                                      img_res=(self.img_rows, self.img_cols))

        patch = int(self.img_rows / 2**4)
        self.disc_patch = (patch, patch, 1)

        self.gf = 32
        self.df = 64

        self.lambda_cycle = 10.0
        self.lambda_id = 0.9 * self.lambda_cycle

        optimizer = Adam(0.0002, 0.5)
```

Input shape

Configures data loader

Uses the DataLoader object to import a preprocessed dataset

Calculates output shape of D (PatchGAN)

Number of filters in the first layer of G

Number of filters in the first layer of D

Cycle-consistency loss weight

Identity loss weight

Two new terms are `lambda_cycle` and `lambda_id`. The second hyperparameter influences identity loss. The CycleGAN authors themselves note that this value influences how dramatic the changes are—especially early in the training process.[7] Setting a lower value leads to unnecessary changes: for example, completely inverting the colors early on. We have selected this value, based on rerunning the training process for apple2orange several times. Frequently, the process is theory-driven alchemy.

The first hyperparameter—`lambda_cycle`—controls how strictly the cycle-consistency loss is enforced. Setting this value higher will ensure that your original and reconstructed images are as close together as possible.

9.7.1 Building the network

So now that we have our basic parameters out of the way, we will build the basic network, as shown in listing 9.3. We will start from the high-level view and move down. This entails the following:

1. Creating the two Discriminators D_A and D_B and compiling them
2. Creating the two Generators:
   a. Instantiating G_{AB} and G_{BA}
   b. Creating placeholders for the image input for both directions
   c. Linking them both to an image in the other domain

[7] See "pytorch-CycleGAN-and-pix2pix Frequently Asked Questions," by Jun-Yan Zhu, April 2019, http://mng.bz/BY58.

d Creating placeholders for the reconstructed images back in the original domain
e Creating the identity loss constraint for both directions
f Not making the parameters of the Discriminators trainable for now
g Compiling the two Generators

Listing 9.3 Building the networks

```
self.d_A = self.build_discriminator()
self.d_B = self.build_discriminator()
self.d_A.compile(loss='mse',
                 optimizer=optimizer,
                 metrics=['accuracy'])
self.d_B.compile(loss='mse',
                 optimizer=optimizer,
                 metrics=['accuracy'])

self.g_AB = self.build_generator()
self.g_BA = self.build_generator()

img_A = Input(shape=self.img_shape)
img_B = Input(shape=self.img_shape)

fake_B = self.g_AB(img_A)
fake_A = self.g_BA(img_B)
reconstr_A = self.g_BA(fake_B)
reconstr_B = self.g_AB(fake_A)
img_A_id = self.g_BA(img_A)
img_B_id = self.g_AB(img_B)

self.d_A.trainable = False
self.d_B.trainable = False

valid_A = self.d_A(fake_A)
valid_B = self.d_B(fake_B)

self.combined = Model(inputs=[img_A, img_B],
                      outputs=[valid_A, valid_B,
                               reconstr_A, reconstr_B,
                               img_A_id, img_B_id])
self.combined.compile(loss=['mse', 'mse',
                            'mae', 'mae',
                            'mae', 'mae'],
                      loss_weights=[1, 1,
                                    self.lambda_cycle, self.lambda_cycle,
                                    self.lambda_id, self.lambda_id],
                      optimizer=optimizer)
```

Builds and compiles the Discriminators

Beginning here, we construct the computational graph of the Generators. These first two lines build the Generators.

Inputs images from both domains

Translates images to the other domain

Translates images back to original domain

Identity mapping of images

For the combined model, we will train only the Generators.

Discriminators determine validity of translated images

Combined model trains Generators to fool Discriminators

One last thing to clarify from the preceding code: the outputs from the `combined` model come in lists of six. This is because we always get validities (from the Discriminator),

reconstruction, and identity losses—one for A-B-A and one for the B-A-B cycle—hence six. The first two are squared errors, and the rest are mean absolute errors. The relative weights are influenced by the `lambda` factors described earlier.

9.7.2 Building the Generator

Next, we build the Generator code in listing 9.4, which uses the skip connections as we described in section 9.5.2. This is the U-Net architecture. This architecture is simpler to write than the ResNet architecture, which some implementations use. Within our Generator function we first define the helper functions:

1. Define the `conv2d()` function as follows:
   a. Standard 2D convolutional layer
   b. Leaky ReLU activation
   c. Instance normalization[8]
2. Define the `deconv2d()` function as a transposed[9] convolution (aka *deconvolution*) layer that does the following:
   a. Upsamples the `input_layer`
   b. Possibly applies dropout if we set the dropout rate
   c. Always applies `InstanceNormalization`
   d. More importantly, creates a skip connection between its output layer and the layer of corresponding dimensionality from the downsampling part from figure 9.4

NOTE In step 2d, we're using a simple `UpSampling2D`, which is not a learned parameter, but rather uses the nearest neighbors interpolation.

Then we create the actual Generator:

3. Take the input (128 × 128 × 3) and assign that to `d0`.
4. Run that through a convolutional layer `d1`, arriving at a 64 × 64 × 32 layer.
5. Take `d1` (64 × 64 × 32) and apply `conv2d` to get 32 ×32 × 64 (`d2`).
6. Take `d2` (32 × 32 × 64) and apply `conv2d` to get 16 × 16 × 128 (`d3`).
7. Take `d3` (16 × 16 × 128) and apply `conv2d` to get 8 × 8 × 256 (`d4`).
8. `u1`: Upsample `d4` and create a skip connection between `d3` and `u1`.
9. `u2`: Upsample `u1` and create a skip connection between `d2` and `u2`.
10. `u3`: Upsample `u2` and create a skip connection between `d1` and `u3`.
11. `u4`: Use regular upsampling to arrive at a 128 × 128 × 64 image.

[8] Instance normalization is similar to the batch normalization in chapter 4, except that instead of normalizing based on information from the entire batch, we normalize each feature map within each channel separately. Instance normalization often results in better-quality images for tasks such as style transfer or image-to-image translation—just what we need for the CycleGAN!

[9] Here, *transposed convolution* is—some argue—a more correct term. However, just think of it as the opposite of convolution, or deconvolution.

12 Use a regular 2D convolution to get rid of the extra feature maps and get only 128 × 128 × 3 (height × width × color_channels)

Listing 9.4 Building the generator

```
def build_generator(self):
    """U-Net Generator"""

    def conv2d(layer_input, filters, f_size=4):
        """Layers used during downsampling"""
        d = Conv2D(filters, kernel_size=f_size,
                   strides=2, padding='same')(layer_input)
        d = LeakyReLU(alpha=0.2)(d)
        d = InstanceNormalization()(d)
        return d

    def deconv2d(layer_input, skip_input, filters, f_size=4,
        dropout_rate=0):
        """Layers used during upsampling"""
        u = UpSampling2D(size=2)(layer_input)
        u = Conv2D(filters, kernel_size=f_size, strides=1,
                   padding='same', activation='relu')(u)
        if dropout_rate:
            u = Dropout(dropout_rate)(u)
        u = InstanceNormalization()(u)
        u = Concatenate()([u, skip_input])
        return u

    d0 = Input(shape=self.img_shape)  <-- Image input

    d1 = conv2d(d0, self.gf)
    d2 = conv2d(d1, self.gf * 2)
    d3 = conv2d(d2, self.gf * 4)
    d4 = conv2d(d3, self.gf * 8)      <-- Downsampling

    u1 = deconv2d(d4, d3, self.gf * 4)
    u2 = deconv2d(u1, d2, self.gf * 2)
    u3 = deconv2d(u2, d1, self.gf)    <-- Upsampling

    u4 = UpSampling2D(size=2)(u3)
    output_img = Conv2D(self.channels, kernel_size=4,
                        strides=1, padding='same', activation='tanh')(u4)

    return Model(d0, output_img)
```

9.7.3 Building the Discriminator

Now for the Discriminator method, which uses a helper function that creates layers formed of 2D convolutions, `LeakyReLU`, and optionally, `InstanceNormalization`.

We apply these layers the following way, as shown in listing 9.5:

1 We take the input image (128 × 128 × 3) and assign that to `d1` (64 × 64 × 64).
2 We take `d1` (64 × 64 × 64) and assign that to `d2` (32 × 32 × 128).

3. We take `d2` ($32 \times 32 \times 128$) and assign that to `d3` ($16 \times 16 \times 256$).
4. We take `d3` ($16 \times 16 \times 256$) and assign that to `d4` ($8 \times 8 \times 512$).
5. We take `d4` ($8 \times 8 \times 512$) and flatten by `conv2d` to $8 \times 8 \times 1$.

Listing 9.5 Building the Discriminator

```
def build_discriminator(self):

        def d_layer(layer_input, filters, f_size=4, normalization=True):
            """Discriminator layer"""
            d = Conv2D(filters, kernel_size=f_size,
                       strides=2, padding='same')(layer_input)
            d = LeakyReLU(alpha=0.2)(d)
            if normalization:
                d = InstanceNormalization()(d)
            return d

        img = Input(shape=self.img_shape)

        d1 = d_layer(img, self.df, normalization=False)
        d2 = d_layer(d1, self.df * 2)
        d3 = d_layer(d2, self.df * 4)
        d4 = d_layer(d3, self.df * 8)

        validity = Conv2D(1, kernel_size=4, strides=1, padding='same')(d4)

        return Model(img, validity)
```

9.7.4 Training the CycleGAN

With all networks written, now we will implement the method that creates our training loop. For the CycleGAN training algorithm, the details of each training iteration are as follows.

CycleGAN training algorithm

For each training iteration ***do***

1. Train the Discriminator:
   a. Take a mini-batch of random images from each domain ($imgs_A$ and $imgs_B$).
   b. Use the Generator G_{AB} to translate $imgs_A$ to domain B and vice versa with G_{BA}.
   c. Compute $D_A(imgs_A, 1)$ and $D_A(G_{BA}(imgs_B), 0)$ to get the losses for real images in A and translated images from B, respectively. Then add these two losses together. The 1 and 0 in D_A serve as labels.
   d. Compute $D_B(imgs_B, 1)$ and $D_B(G_{AB}(imgs_A), 0)$ to get the losses for real images in B and translated images from A, respectively. Then add these two losses together. The 1 and 0 in D_B serve as labels.
   e. Add the losses from steps c and d together to get a total Discriminator loss.

2 Train the Generator:
   a We use the combined model to
      – Input the images from domain A ($imgs_A$) and B ($imgs_B$)
      – The outputs are
         1 Validity of A: $D_A(G_{BA}(imgs_B))$
         2 Validity of B: $D_B(G_{AB}(imgs_A))$
         3 Reconstructed A: $G_{BA}(G_{AB}(imgs_A))$
         4 Reconstructed B: $G_{AB}(G_{BA}(imgs_B))$
         5 Identity mapping of A: $G_{BA}(imgs_A)$)
         6 Identity mapping of B: $G_{AB}(imgs_B)$)
   b We then update the parameters of both Generators inline with the cycle-consistency loss, identity loss, and adversarial loss with
      – Mean squared error (MSE) for the scalars (discriminator probabilities)
      – Mean absolute error (MAE) for images (either reconstructed or identity-mapped)

End for

The following listing implements this CycleGAN training algorithm.

Listing 9.6 Training CycleGAN

```
def train(self, epochs, batch_size=1, sample_interval=50):

    start_time = datetime.datetime.now()

    valid = np.ones((batch_size,) + self.disc_patch)
    fake = np.zeros((batch_size,) + self.disc_patch)


    for epoch in range(epochs):
        for batch_i, (imgs_A, imgs_B) in enumerate(
            self.data_loader.load_batch(batch_size)):

            fake_B = self.g_AB.predict(imgs_A)
            fake_A = self.g_BA.predict(imgs_B)

            dA_loss_real = self.d_A.train_on_batch(imgs_A, valid)
            dA_loss_fake = self.d_A.train_on_batch(fake_A, fake)
            dA_loss = 0.5 * np.add(dA_loss_real, dA_loss_fake)

            dB_loss_real = self.d_B.train_on_batch(imgs_B, valid)
            dB_loss_fake = self.d_B.train_on_batch(fake_B, fake)
            dB_loss = 0.5 * np.add(dB_loss_real, dB_loss_fake)

            d_loss = 0.5 * np.add(dA_loss, dB_loss)
```

Adversarial loss ground truths

Now we begin to train the Discriminators. These lines translate images to the opposite domain.

Trains the Discriminators (original images = real / translated = Fake)

Total Discriminator loss

Trains the Generators

```
g_loss = self.combined.train_on_batch([imgs_A, imgs_B],
                                      [valid, valid,
                                       imgs_A, imgs_B,
                                       imgs_A, imgs_B])
```

If at save interval => save generated image samples

```
if batch_i % sample_interval == 0:
    self.sample_images(epoch, batch_i)
```

This function is similar to what you have encountered and is made explicit in the GitHub repository.

9.7.5 Running CycleGAN

We have written all of this complicated code and are now ready to instantiate a CycleGAN object and look at some results, from the sampled images:

```
gan = CycleGAN()
gan.train(epochs=100, batch_size=64, sample_interval=10)
```

Figure 9.7 shows some results of our hard work.

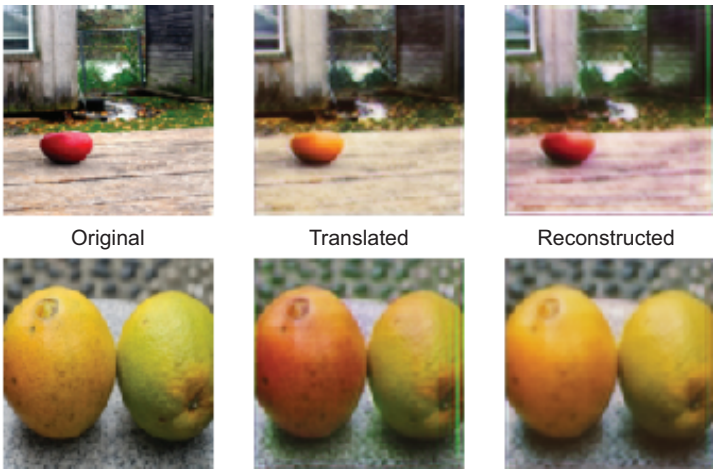

Figure 9.7 Apples translated into oranges, and oranges into apples. These are results as they appear verbatim in our Jupyter notebook. (Results may vary slightly based on random seeds, implementation of TensorFlow and Keras, and hyperparameters.)

9.8 Expansions, augmentations, and applications

When you run these results, we hope you will be as impressed as we were. Because of the absolutely astonishing results, lots of researchers flocked to improve on the technique. This section details a CycleGAN extension and then discusses some CycleGAN applications.

9.8.1 Augmented CycleGAN

"Augmented CycleGAN: Learning Many-to-Many Mappings from Unpaired Data" is a really neat extension to standard CycleGAN that injects latent space information during both translations. Presented at ICML 2018 in Stockholm, Augmented CycleGAN gives us extra variables that drive the generative process.[10] In the same way that we have used latent space in Conditional GANs' case, we can use it in the CycleGAN setting over and above what CycleGAN already does.

For example, if we have an outline of a shoe in the A domain, we can generate a sample in the B domain, where the same type of shoe is blue. In traditional CycleGAN's case, it would always be blue. But now, with the latent variables at our disposal, it can be orange, yellow, or whatever we choose.

This is also a useful framework to think about the limitations of the original CycleGAN: because we are not given any extra seeding parameters (such as an extra latent vector z), we cannot control or alter what comes out the other end. If from a particular handbag outline we get an image that is orange, it will always be orange. Augmented CycleGAN gives us more control over the outcomes, as shown in figure 9.8.

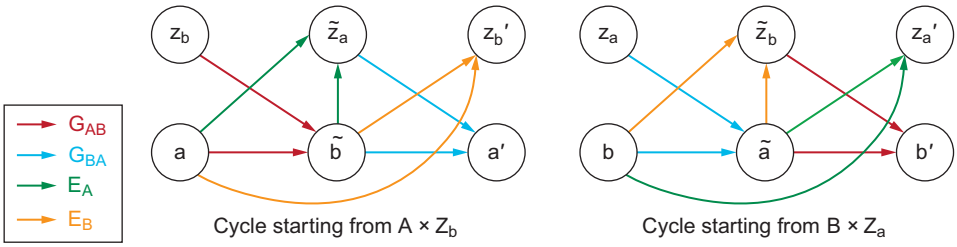

Figure 9.8 In this information flow of the augmented CycleGAN, we have latent vectors Z_a and Z_b that seed the Generator along with the image input, effectively reducing the problem to two CGANs joined together. This allows us to control the generation.
(Source: "Augmented CycleGAN: Learning Many-to-Many Mappings from Unpaired Data," by Amjad Almahairi et al., 2018, http://arxiv.org/abs/1802.10151.)

9.8.2 Applications

Many CycleGAN (or CycleGAN-inspired) applications have been proposed in the short time it has been around. They usually revolve around creating simulated virtual environments and subsequently making them photorealistic. For example, imagine you need more training data for a self-driving car company: just simulate it in Unity or a GTA 5 graphics engine and then use CycleGAN to translate the data.

This works especially well if you need to have particular risk situations that are expensive or time-consuming to re-create (for example, car crashes, or fire trucks speeding to reach a destination), but you need them in your dataset. For a self-driving

[10] See "Augmented Cyclic Adversarial Learning for Low Resource Domain Adaptation," by Ehsan Hosseini-Asl, 2019, https://arxiv.org/pdf/1807.00374.pdf.

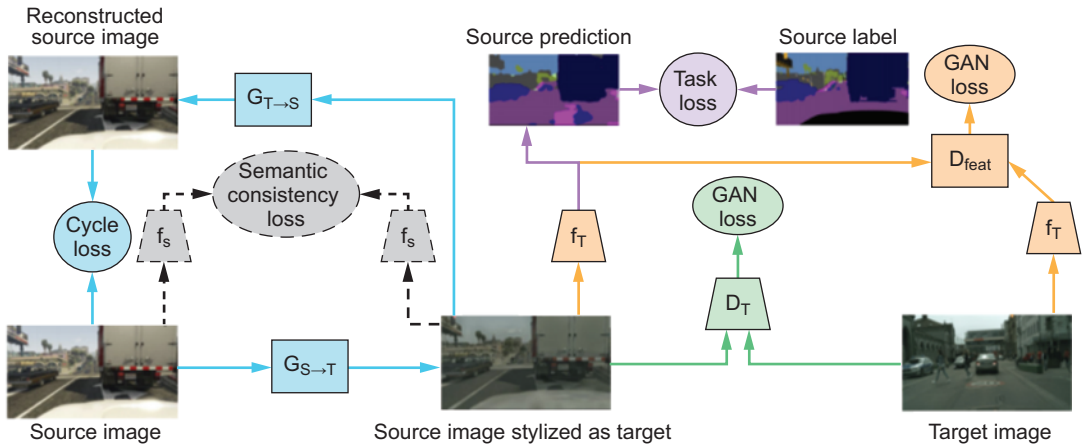

Figure 9.9 This structure should be somewhat familiar from earlier, so hopefully this chapter has at least given you a head start. One extra thing to point out: we now have an extra step with labels and semantic understanding that gives us the so-called *task loss*. This allows us to also check the produced image for semantic meaning.

car company, this could be extremely useful to balance the dataset with at-risk situations, which are rare, but correct behavior is all the more important.

One example of this kind of framework is Cycle Consistent Adversarial Domain Adaptation (CyCADA).[11] Unfortunately, a full explanation of the way it works is beyond the scope of this chapter. This is because there are many more such frameworks: some even experiment with CycleGAN in language, music, or other forms of domain adaptation. To give you a sense of the complexity, figure 9.9 shows the architecture and design of CyCADA.

Summary

- Image-to-image translation frameworks are frequently difficult to train because of the need for perfect pairs; the CycleGAN solves this by making this an unpaired domain translation.
- The CycleGAN has three losses:
  - Cycle-consistent, which measures the difference between the original image and an image translated into a different domain and back again
  - Adversarial, which ensures realistic images
  - Identity, which preserves the color space of the image
- The two Generators use the U-Net architecture, and the two Discriminators use the PatchGAN-based architecture.
- We implemented an object-oriented design of the CycleGAN and used it to convert apples to oranges.
- Practical applications of the CycleGAN include self-driving car training and extensions that allow us to create different styles of images during the translation process.

[11] See "CyCADA: Cycle-Consistent Adversarial Domain Adaptation," by Judy Hoffman et al., 2017, https://arxiv.org/pdf/1711.03213.pdf.