Chapter 4. Generative Adversarial Networks

On Monday, December 5, 2016, at 2:30 p.m., Ian Goodfellow of Google Brain presented a tutorial entitled "Generative Adversarial Networks" to the delegates of the Neural Information Processing Systems (NIPS) conference in Barcelona.[1] The ideas presented in the tutorial are now regarded as one of the key turning points for generative modeling and have spawned a wide variety of variations on his core idea that have pushed the field to even greater heights.

This chapter will first lay out the theoretical underpinning of generative adversarial networks (GANs). You will then learn how to use the Python library Keras to start building your own GANs.

First though, we shall take a trip into the wilderness to meet Gene…

Ganimals

One afternoon, while walking through the local jungle, Gene sees a woman thumbing through a set of black and white photographs, looking worried. He goes over to ask if he can help.

The woman introduces herself as Di, a keen explorer, and explains that she is hunting for the elusive *ganimal*, a mythical creature that is said to roam around the jungle. Since the creature is nocturnal, she only has a collection of nighttime photos of the beast that she once found lying on the floor of the jungle, dropped by another ganimal enthusiast. Some of these photos are shown in Figure 4-1. Di makes money by selling the images to collectors but is starting to worry, as she hasn't actually ever seen the creatures and is concerned that her business will falter if she can't produce more original photographs soon.

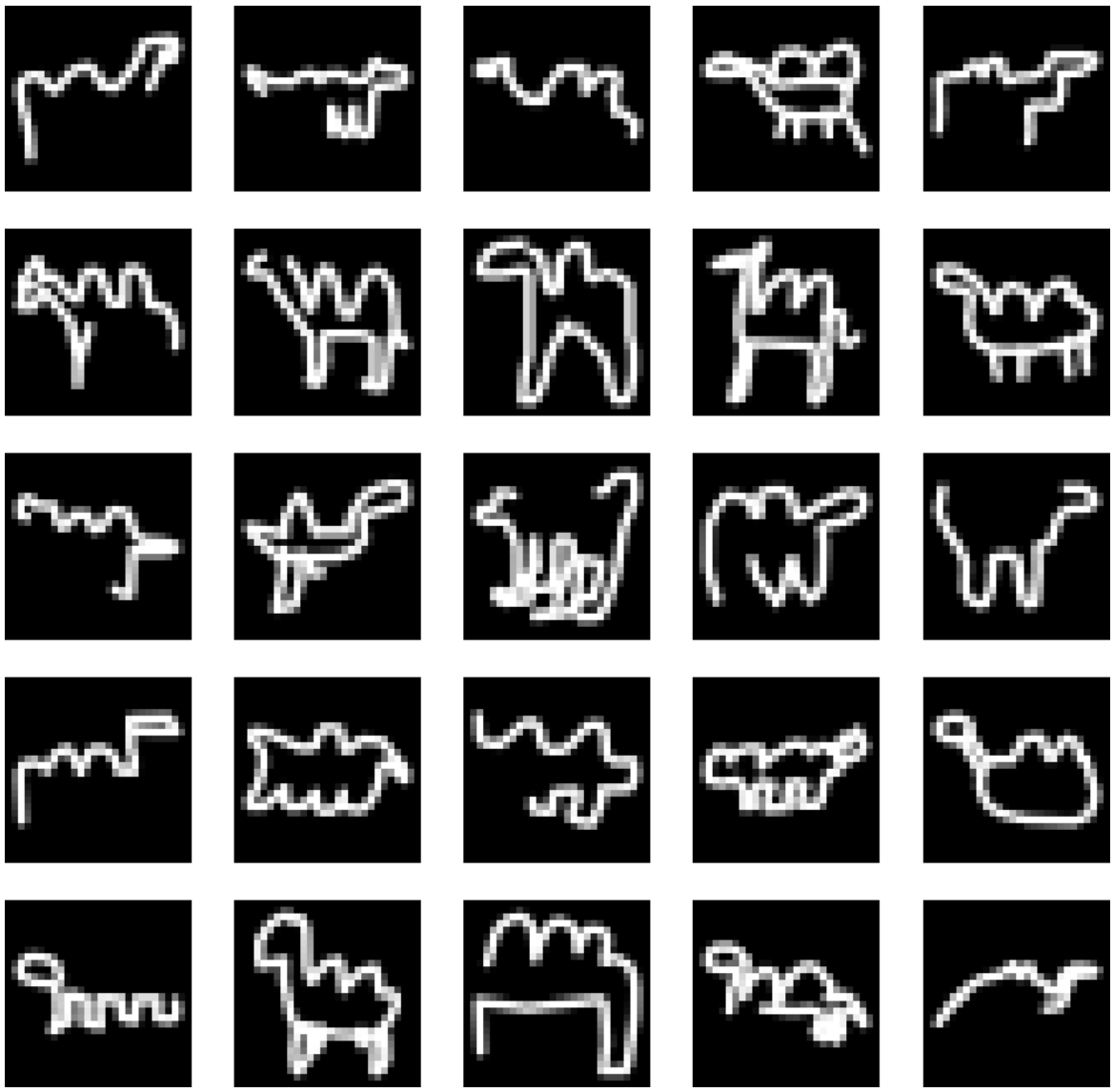

Figure 4-1. Original ganimal photographs

Being a keen photographer, Gene decides to help Di. He agrees to search for the ganimal himself and give her any photographs of the nocturnal beast that he manages to take.

However, there is a problem. Since Gene has never seen a ganimal, he doesn't know how to produce good photos of the creature, and also, since Di has only ever sold the photos she found, she cannot even tell the difference between a good photo of a ganimal and a photo of nothing at all.

Starting from this state of ignorance, how can they work together to ensure Gene is eventually able to produce impressive ganimal photographs?

They come up with following process. Each night, Gene takes 64 photographs, each in a different location with different random moonlight readings, and mixes them with 64 ganimal photos from the original collection. Di then looks at this set of photos and tries to guess which were taken by Gene and which are originals. Based on her mistakes, she updates her own understanding of how to discriminate between Gene's attempts and the original photos. Afterwards, Gene takes another 64 photos and shows them to Di. Di gives each photo a score between 0 and 1, indicating how realistic she thinks each photo is. Based on this feedback, Gene updates the settings on his camera to ensure that next time, he takes photos that Di is more likely to rate highly.

This process continues for many days. Initially, Gene doesn't get any useful feedback from Di, since she is randomly guessing which photos are genuine. However, after a few weeks of her training ritual, she gradually gets better at this, which means that she can provide better feedback to Gene so that he can adjust his camera accordingly in his training session. This makes Di's task harder, since now Gene's photos aren't quite as easy to distinguish from the real photos, so she must again learn how to improve. This back-and-forth process continues, over many days and weeks.

Over time, Gene gets better and better at producing ganimal photos, until eventually, Di is once again resigned to the fact that she cannot tell the difference between Gene's photos and the originals. They take Gene's generated photos to the auction and the experts cannot believe the quality of the new sightings—they are just as convincing as the originals. Some examples of Gene's work are shown in Figure 4-2.

Figure 4-2. Samples of Gene's ganimal photography

Introduction to GANs

The adventures of Gene and Di hunting elusive nocturnal ganimals are a metaphor for one of the most important deep learning advancements of recent years: generative adversarial networks.

Simply put, a GAN is a battle between two adversaries, the generator and the discriminator. The generator tries to convert random noise into observations that look as if they have been sampled from the original dataset and the discriminator tries to predict whether an observation comes from the original dataset or is one of the generator's forgeries. Examples of the inputs and outputs to the two networks are shown in Figure 4-3.

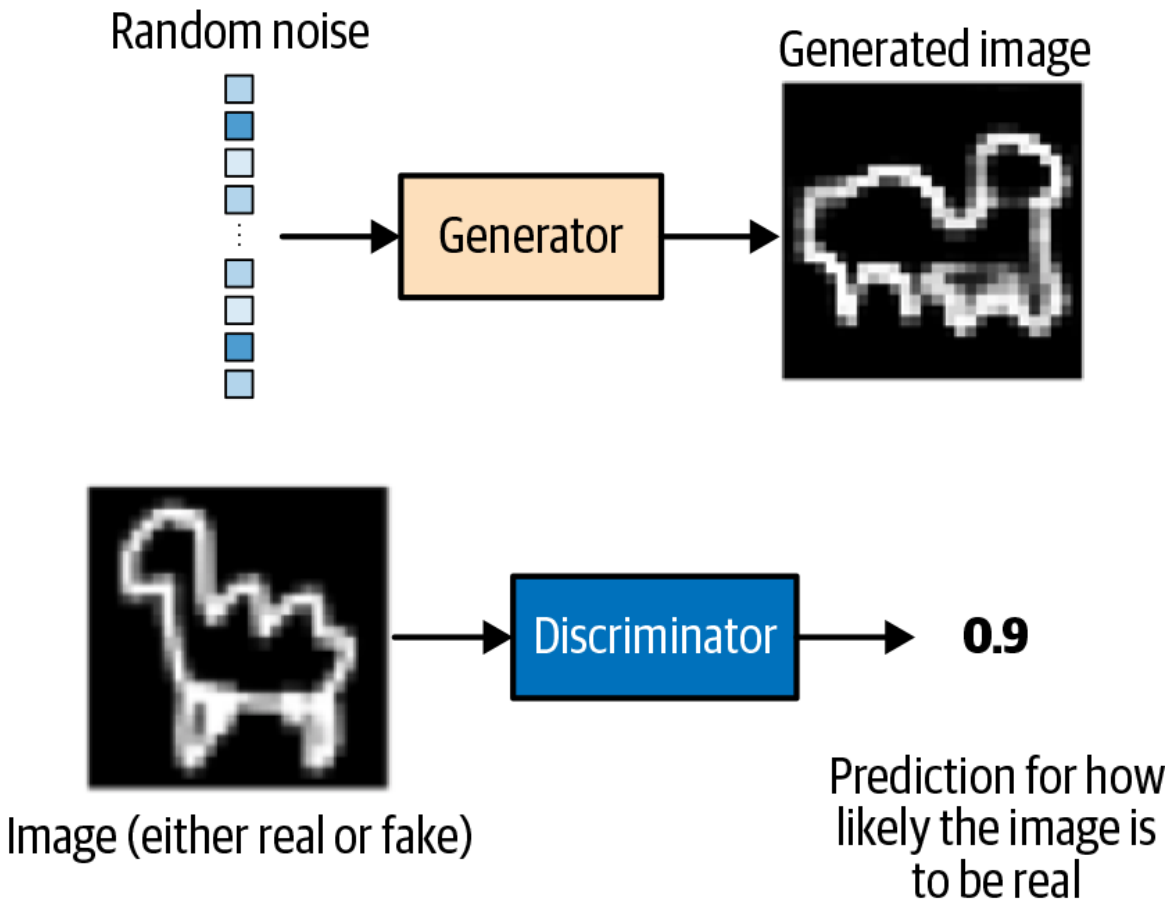

Figure 4-3. Inputs and outputs of the two networks in a GAN

At the start of the process, the generator outputs noisy images and the discriminator predicts randomly. The key to GANs lies in how we alternate the training of the two networks, so that as the generator becomes more adept at fooling the discriminator, the discriminator must adapt in order to maintain its ability to correctly identify which observations are fake. This drives the generator to find new ways to fool the discriminator, and so the cycle continues.

To see this in action, let's start building our first GAN in Keras, to generate pictures of nocturnal ganimals.

Your First GAN

First, you'll need to download the training data. We'll be using the Quick, Draw! dataset from Google. This is a crowdsourced collection of 28 × 28–

pixel grayscale doodles, labeled by subject. The dataset was collected as part of an online game that challenged players to draw a picture of an object or concept, while a neural network tries to guess the subject of the doodle. It's a really useful and fun dataset for learning the fundamentals of deep learning. For this task you'll need to download the *camel* `numpy` file and save it into the *./data/camel/* folder in the book repository.[2] The original data is scaled in the range [0, 255] to denote the pixel intensity. For this GAN we rescale the data to the range [–1, 1].

Running the notebook *04_01_gan_camel_train.ipynb* in the book repository will start training the GAN. As in the previous chapter on VAEs, you can instantiate a GAN object in the notebook, as shown in Example 4-1, and play around with the parameters to see how it affects the model.

Example 4-1. Defining the GAN

```
gan = GAN(input_dim = (28,28,1)
          , discriminator_conv_filters = [64,64,128,128]
          , discriminator_conv_kernel_size = [5,5,5,5]
          , discriminator_conv_strides = [2,2,2,1]
          , discriminator_batch_norm_momentum = None
          , discriminator_activation = 'relu'
          , discriminator_dropout_rate = 0.4
          , discriminator_learning_rate = 0.0008
          , generator_initial_dense_layer_size = (7, 7, 64)
          , generator_upsample = [2,2, 1, 1]
          , generator_conv_filters = [128,64, 64,1]
          , generator_conv_kernel_size = [5,5,5,5]
          , generator_conv_strides = [1,1, 1, 1]
          , generator_batch_norm_momentum = 0.9
          , generator_activation = 'relu'
          , generator_dropout_rate = None
          , generator_learning_rate = 0.0004
          , optimiser = 'rmsprop'
          , z_dim = 100
          )
```

Let's first take a look at how we build the discriminator.

The Discriminator

The goal of the discriminator is to predict if an image is real or fake. This is a supervised image classification problem, so we can use the same network architecture as in Chapter 2: stacked convolutional layers, followed by a dense output layer.

In the original GAN paper, dense layers were used in place of the convolutional layers. However, since then, it has been shown that convolutional layers give greater predictive power to the discriminator. You may see this type of GAN called a DCGAN (deep convolutional generative adversarial network) in the literature, but now essentially all GAN architectures contain convolutional layers, so the "DC" is implied when we talk about GANs. It is also common to see batch normalization layers in the discriminator for vanilla GANs, though we choose not to use them here for simplicity.

The full architecture of the discriminator we will be building is shown in Figure 4-4.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
discriminator_input (InputLa (None, 28, 28, 1)         0
_________________________________________________________________
discriminator_conv_0 (Conv2D (None, 14, 14, 64)        1664
_________________________________________________________________
activation_1 (Activation)    (None, 14, 14, 64)        0
_________________________________________________________________
dropout_1 (Dropout)          (None, 14, 14, 64)        0
_________________________________________________________________
discriminator_conv_1 (Conv2D (None, 7, 7, 64)          102464
_________________________________________________________________
activation_2 (Activation)    (None, 7, 7, 64)          0
_________________________________________________________________
dropout_2 (Dropout)          (None, 7, 7, 64)          0
_________________________________________________________________
discriminator_conv_2 (Conv2D (None, 4, 4, 128)         204928
_________________________________________________________________
activation_3 (Activation)    (None, 4, 4, 128)         0
_________________________________________________________________
dropout_3 (Dropout)          (None, 4, 4, 128)         0
_________________________________________________________________
discriminator_conv_3 (Conv2D (None, 4, 4, 128)         409728
_________________________________________________________________
activation_4 (Activation)    (None, 4, 4, 128)         0
_________________________________________________________________
dropout_4 (Dropout)          (None, 4, 4, 128)         0
_________________________________________________________________
flatten_1 (Flatten)          (None, 2048)              0
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 2049
=================================================================
Total params: 720,833
Trainable params: 720,833
Non-trainable params: 0
_________________________________________________________________
```

Figure 4-4. The discriminator of the GAN

The Keras code to build the discriminator is provided in Example 4-2.

Example 4-2. The discriminator

```
discriminator_input = Input(shape=self.input_dim,
name='discriminator_input') ❶
x = discriminator_input

for i in range(self.n_layers_discriminator): ❷
```

```
    x = Conv2D(
        filters = self.discriminator_conv_filters[i]
        , kernel_size = self.discriminator_conv_kernel_size[i]
        , strides = self.discriminator_conv_strides[i]
        , padding = 'same'
        , name = 'discriminator_conv_' + str(i)
        )(x)

    if self.discriminator_batch_norm_momentum and i > 0:
        x = BatchNormalization(momentum =
self.discriminator_batch_norm_momentum)(x)

    x = Activation(self.discriminator_activation)(x)

    if self.discriminator_dropout_rate:
        x = Dropout(rate = self.discriminator_dropout_rate)(x)

x = Flatten()(x) ❸
discriminator_output= Dense(1, activation='sigmoid'
    , kernel_initializer = self.weight_init)(x) ❹

discriminator = Model(discriminator_input, discriminator_output) ❺
```

❶ Define the input to the discriminator (the image).

❷ Stack convolutional layers on top of each other.

❸ Flatten the last convolutional layer to a vector.

❹ `Dense` layer of one unit, with a sigmoid activation function that transforms the output from the dense layer to the range [0, 1].

❺ The Keras model that defines the discriminator—a model that takes an input image and outputs a single number between 0 and 1.

Notice how we use a stride of 2 in some of the convolutional layers to reduce the size of the tensor as it passes through the network, but increase the number of channels (1 in the grayscale input image, then 64, then 128).

The sigmoid activation in the final layer ensures that the output is scaled between 0 and 1. This will be the predicted probability that the image is real.

The Generator

Now let's build the generator. The input to the generator is a vector, usually drawn from a multivariate standard normal distribution. The output is an

image of the same size as an image in the original training data.

This description may remind you of the decoder in a variational autoencoder. In fact, the generator of a GAN fulfills exactly the same purpose as the decoder of a VAE: converting a vector in the latent space to an image. The concept of mapping from a latent space back to the original domain is very common in generative modeling as it gives us the ability to manipulate vectors in the latent space to change high-level features of images in the original domain.

The architecture of the generator we will be building is shown in Figure 4-5.

Layer (type)	Output Shape	Param #
generator_input (InputLayer)	(None, 100)	0
dense_9 (Dense)	(None, 3136)	316736
batch_normalization_10 (Batc	(None, 3136)	12544
activation_36 (Activation)	(None, 3136)	0
reshape_4 (Reshape)	(None, 7, 7, 64)	0
up_sampling2d_10 (UpSampling	(None, 14, 14, 64)	0
generator_conv_0 (Conv2D)	(None, 14, 14, 128)	204928
batch_normalization_11 (Batc	(None, 14, 14, 128)	512
activation_37 (Activation)	(None, 14, 14, 128)	0
up_sampling2d_11 (UpSampling	(None, 28, 28, 128)	0
generator_conv_1 (Conv2D)	(None, 28, 28, 64)	204864
batch_normalization_12 (Batc	(None, 28, 28, 64)	256
activation_38 (Activation)	(None, 28, 28, 64)	0
generator_conv_2 (Conv2D)	(None, 28, 28, 64)	102464
batch_normalization_13 (Batc	(None, 28, 28, 64)	256
activation_39 (Activation)	(None, 28, 28, 64)	0
generator_conv_3 (Conv2D)	(None, 28, 28, 1)	1601
activation_40 (Activation)	(None, 28, 28, 1)	0

Total params: 844,161
Trainable params: 837,377
Non-trainable params: 6,784

Figure 4-5. The generator

First though, we need to introduce a new layer type: the *upsampling* layer.

UPSAMPLING

In the decoder of the variational autoencoder that we built in the previous chapter, we doubled the width and height of the tensor at each layer using `Conv2DTranspose` layers with stride 2. This inserted zero values in between pixels before performing the convolution operations.

In this GAN, we instead use the Keras `Upsampling2D` layer to double the width and height of the input tensor. This simply repeats each row and column of its input in order to double the size. We then follow this with a normal convolutional layer with stride 1 to perform the convolution operation. It is a similar idea to convolutional transpose, but instead of filling the gaps between pixels with zeros, upsampling just repeats the existing pixel values.

Both of these methods—`Upsampling` + `Conv2D` and `Conv2DTranspose`—are acceptable ways to transform back to the original image domain. It really is a case of testing both methods in your own problem setting and seeing which produces better results. It has been shown that the `Conv2DTranspose` method can lead to *artifacts*, or small checkerboard patterns in the output image (see Figure 4-6) that spoil the quality of the output. However, they are still used in many of the most impressive GANs in the literature and have proven to be a powerful tool in the deep learning practitioner's toolbox—again, I suggest you experiment with both methods and see which works best for you.

Figure 4-6. Artifacts when using convolutional transpose layers[3]

The code for building the generator is given in Example 4-3.

Example 4-3. The generator

```
generator_input = Input(shape=(self.z_dim,), name='generator_input') ❶
x = generator_input

x = Dense(np.prod(self.generator_initial_dense_layer_size))(x) ❷

if self.generator_batch_norm_momentum:
    x = BatchNormalization(momentum = self.generator_batch_norm_momentum)(x)
x = Activation(self.generator_activation)(x)

x = Reshape(self.generator_initial_dense_layer_size)(x) ❸

if self.generator_dropout_rate:
    x = Dropout(rate = self.generator_dropout_rate)(x)

for i in range(self.n_layers_generator): ❹

    x = UpSampling2D()(x)
    x = Conv2D(
        filters = self.generator_conv_filters[i]
        , kernel_size = self.generator_conv_kernel_size[i]
        , padding = 'same'
        , name = 'generator_conv_' + str(i)
        )(x)

    if i < n_layers_generator - 1: ❺
        if self.generator_batch_norm_momentum:
```

```
            x = BatchNormalization(momentum =
self.generator_batch_norm_momentum))(x)
        x = Activation('relu')(x)
    else:
        x = Activation('tanh')(x)


generator_output = x
generator = Model(generator_input, generator_output) ❻
```

❶ Define the input to the generator—a vector of length 100.

❷ We follow this with a `Dense` layer consisting of 3,136 units…

❸ …which, after applying batch normalization and a ReLU activation function, is reshaped to a 7 × 7 × 64 tensor.

❹ We pass this through four `Conv2D` layers, the first two preceded by `Upsampling2D` layers, to reshape the tensor to 14 × 14, then 28 × 28 (the original image size). In all but the last layer, we use batch normalization and ReLU activation (LeakyReLU could also be used).

❺ After the final `Conv2D` layer, we use a tanh activation to transform the output to the range [–1, 1], to match the original image domain.

❻ The Keras model that defines the generator—a model that accepts a vector of length 100 and outputs a tensor of shape `[28, 28, 1]`.

Training the GAN

As we have seen, the architecture of the generator and discriminator in a GAN is very simple and not so different from the models that we looked at earlier. The key to understanding GANs is in understanding the training process.

We can train the discriminator by creating a training set where some of the images are randomly selected *real* observations from the training set and some are outputs from the generator. The response would be 1 for the true images and 0 for the generated images. If we treat this as a supervised learning problem, we can train the discriminator to learn how to tell the difference between the original and generated images, outputting values near 1 for the true images and values near 0 for the fake images.

Figure 5-3. Examples of the oranges and apples in the greengrocers' store

CycleGAN

The preceding story is an allegory for a key development in generative modeling and, in particular, style transfer: the cycle-consistent adversarial network, or *CycleGAN*. The original paper represented a significant step forward in the field of style transfer as it showed how it was possible to train a model that could copy the style from a reference set of images onto a different image, without a training set of paired examples.[2]

Previous style transfer models, such as *pix2pix*,[3] required each image in the training set to exist in both the source and target domain. While it is possible to manufacture this kind of dataset for some style problem settings (e.g., black and white to color photos, maps to satellite images), for others it is impossible. For example, we do not have original photographs of the pond where Monet painted his *Water Lilies* series, nor do we have a Picasso painting of the Empire State Building. It would also take enormous effort to arrange photos of horses and zebras standing in identical positions.

The CycleGAN paper was released only a few months after the pix2pix paper and shows how it is possible to train a model to tackle problems where we do not have pairs of images in the source and target domains. Figure 5-4 shows the difference between the paired and unpaired datasets of pix2pix and CycleGAN, respectively.

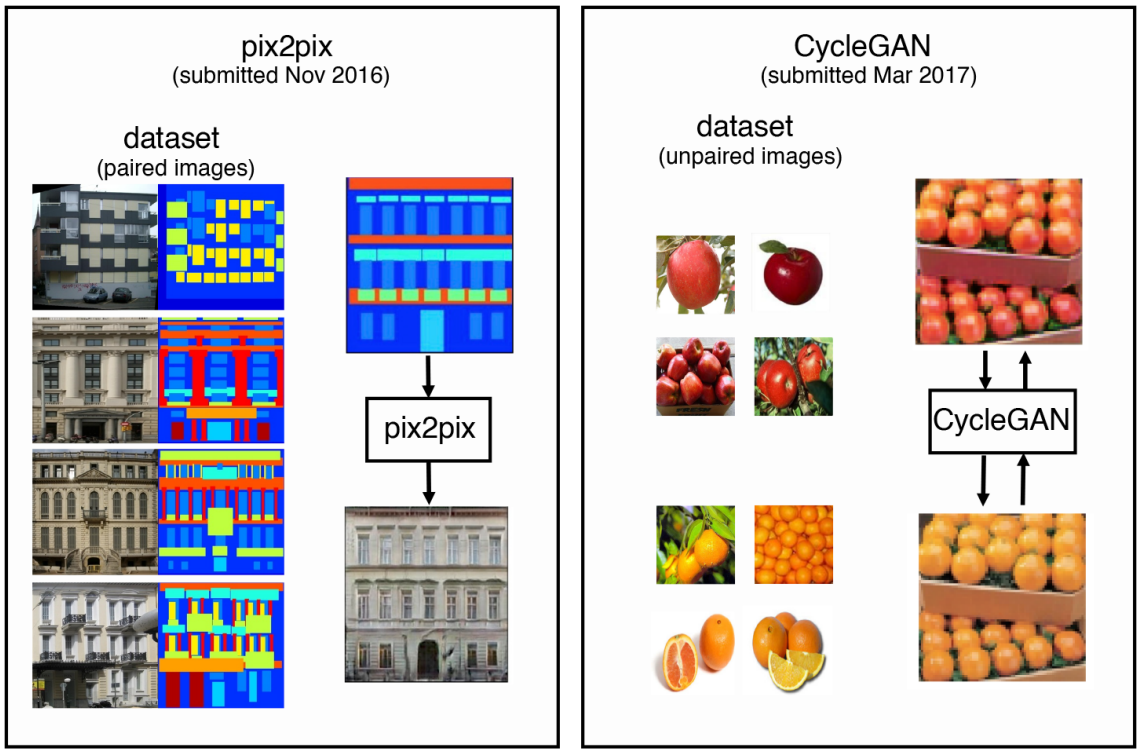

Figure 5-4. pix2pix dataset and domain mapping example

While pix2pix only works in one direction (from source to target), CycleGAN trains the model in both directions simultaneously, so that the model learns to translate images from target to source as well as source to target. This is a consequence of the model architecture, so you get the reverse direction for free.

Let's now see how we can build a CycleGAN model in Keras. To begin with, we shall be using the apples and oranges example from earlier to walk through each part of the CycleGAN and experiment with the architecture. We'll then apply the same technique to build a model that can apply a given artist's style to a photo of your choice.

Your First CycleGAN

Much of the following code has been inspired by and adapted from the amazing Keras-GAN repository maintained by Erik Linder-Norén. This is an excellent resource for many Keras examples of important GANs from the literature.

To begin, you'll first need to download the data that we'll be using to train the CycleGAN. From inside the folder where you cloned the book's repository, run the following command:

```
bash ./scripts/download_cyclegan_data.sh apple2orange
```

This will download the dataset of images of apples and oranges that we will be using. The data is split into four folders: *trainA* and *testA* contain images of apples and *trainB* and *testB* contain images of oranges. Thus domain A is the space of apple images and domain B is the space of orange images. Our goal is to train a model using the *train* datasets to convert images from domain A into domain B and vice versa. We will test our model using the *test* datasets.

Overview

A CycleGAN is actually composed of four models, two generators and two discriminators. The first generator, `G_AB`, converts images from domain A into domain B. The second generator, `G_BA`, converts images from domain B into domain A.

As we do not have paired images on which to train our generators, we also need to train two discriminators that will determine if the images produced by the generators are convincing. The first discriminator, `d_A`, is trained to be able to identify the difference between real images from domain A and fake images that have been produced by generator `G_BA`. Conversely, discriminator `d_B` is trained to be able to identify the difference between real

images from domain B and fake images that have been produced by generator `G_AB`. The relationship between the four models is shown in Figure 5-5.

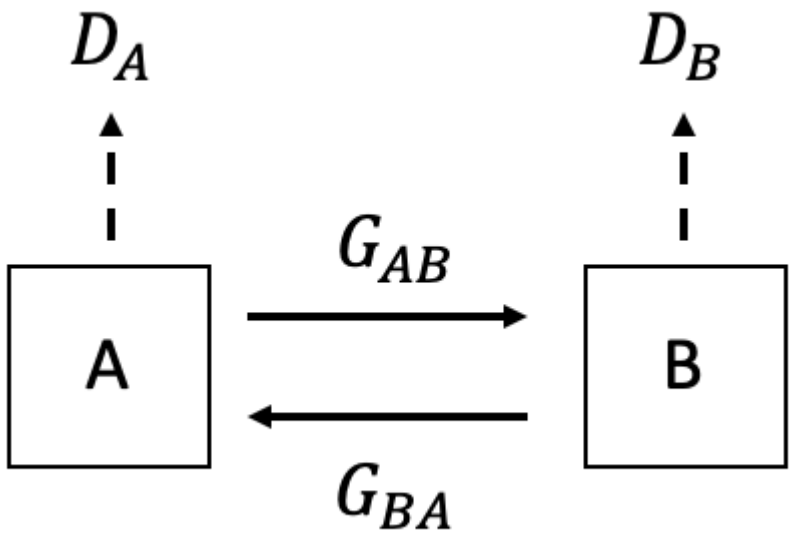

Figure 5-5. Diagram of the four CycleGAN models[4]

Running the notebook *05_01_cyclegan_train.ipynb* in the book repository will start training the CycleGAN. As in previous chapters, you can instantiate a `CycleGAN` object in the notebook, as shown in Example 5-1, and play around with the parameters to see how it affects the model.

Example 5-1. Defining the CycleGAN

```
gan = CycleGAN(
        input_dim = (128,128,3)
        , learning_rate = 0.0002
        , lambda_validation = 1
        , lambda_reconstr = 10
        , lambda_id = 2
        , generator_type = 'u-net'
        , gen_n_filters = 32
        , disc_n_filters = 32
        )
```

Let's first take a look at the architecture of the generators. Typically, CycleGAN generators take one of two forms: *U-Net* or *ResNet* (residual network). In their earlier pix2pix paper,[5] the authors used a U-Net

architecture, but they switched to a ResNet architecture for CycleGAN. We'll be building both architectures in this chapter, starting with U-Net.[6]

The Generators (U-Net)

Figure 5-6 shows the architecture of the U-Net we will be using—no prizes for guessing why it's called a U-Net![7]

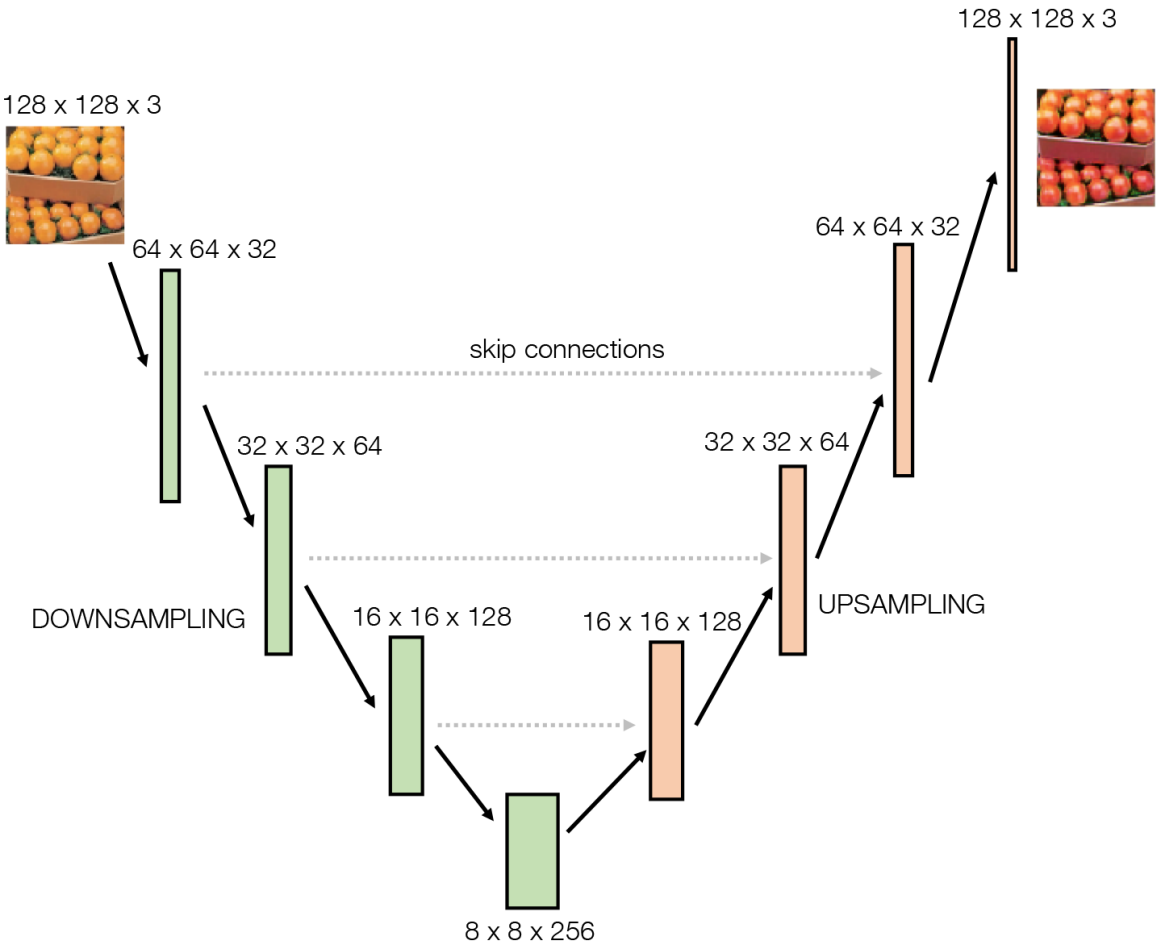

Figure 5-6. The U-Net architecture diagram

In a similar manner to a variational autoencoder, a U-Net consists of two halves: the downsampling half, where input images are compressed spatially but expanded channel-wise, and an upsampling half, where representations are expanded spatially while the number of channels is reduced.

However, unlike in a VAE, there are also *skip connections* between equivalently shaped layers in the upsampling and downsampling parts of the

network. A VAE is linear; data flows through the network from input to the output, one layer after another. A U-Net is different, because it contains skip connections that allow information to shortcut parts of the network and flow through to later layers.

The intuition here is that with each subsequent layer in the downsampling part of the network, the model increasingly captures the *what* of the images and loses information on the *where*. At the apex of the U, the feature maps will have learned a contextual understanding of what is in the image, with little understanding of where it is located. For predictive classification models, this is all we require, so we could connect this to a final `Dense` layer to output the probability of a particular class being present in the image. However, for the original U-Net application (image segmentation) and also for style transfer, it is critical that when we upsample back to the original image size, we pass back into each layer the spatial information that was lost during downsampling. This is exactly why we need the skip connections. They allow the network to blend high-level abstract information captured during the downsampling process (i.e., the image *style*) with the specific spatial information that is being fed back in from previous layers in the network (i.e., the image *content*).

To build in the skip connections, we will need to introduce a new type of layer: $Concatenate$.

CONCATENATE LAYER

The `Concatenate` layer simply joins a set of layers together along a particular axis (by default, the last axis). For example, in Keras, we can join two previous layers, `x` and `y` together as follows:

```
Concatenate()([x,y])
```

In the U-Net, we use `Concatenate` layers to connect upsampling layers to the equivalently sized layer in the downsampling part of the network. The layers are joined together along the channels dimension so the number of channels increases from *k* to *2k*, while the number of spatial dimensions remains the same.

Note that there are no weights to be learned in a `Concatenate` layer; they are just used to "glue" previous layers together.

The generator also contains another new layer type, `InstanceNormalization`.

INSTANCE NORMALIZATION LAYER

The generator of this CycleGAN uses `InstanceNormalization` layers rather than `BatchNormalization` layers, which in style transfer problems can lead to more satisfying results.[8]

An `InstanceNormalization` layer normalizes every single observation individually, rather than as a batch. Unlike a `BatchNormalization` layer, it doesn't require `mu` and `sigma` parameters to be calculated as a running average during training, since at test time the layer can normalize per instance in the same way as it does at train time. The means and standard deviations used to normalize each layer are calculated per channel and per observation.

Also, for the `InstanceNormalization` layers in this network, there are no weights to learn since we do not use scaling (`gamma`) or shift (`beta`) parameters.

Figure 5-7 shows the difference between batch normalization and instance normalization, as well as two other normalization methods (layer and group normalization).

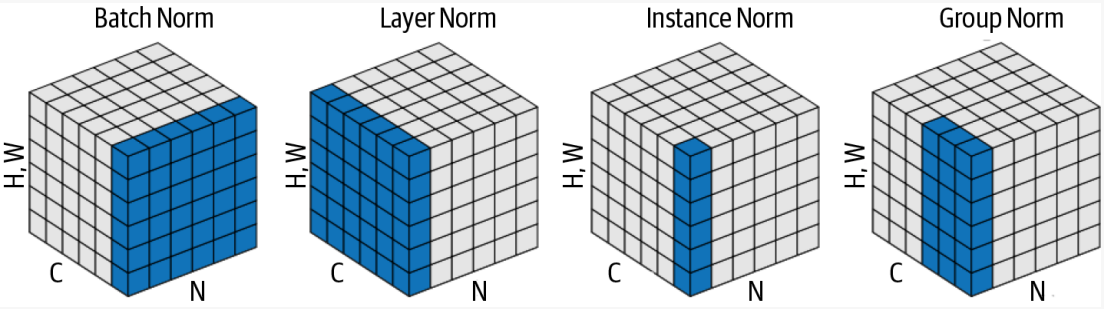

Figure 5-7. Four different normalization methods.[9]

Here, N is the batch axis, C is the channel axis, and (H, W) represent the spatial axes. The cube therefore represents the input tensor to the normalization layer. Pixels colored blue are normalized by the same mean and variance (calculated over the values of these pixels).

We now have everything we need to build a U-Net generator in Keras, as shown in Example 5-2.

Example 5-2. Building the U-Net generator

```
def build_generator_unet(self):

    def downsample(layer_input, filters, f_size=4):
        d = Conv2D(filters, kernel_size=f_size
            , strides=2, padding='same')(layer_input)
        d = InstanceNormalization(axis = -1, center = False, scale = False)
(d)
        d = Activation('relu')(d)

        return d

    def upsample(layer_input, skip_input, filters, f_size=4,
dropout_rate=0):
        u = UpSampling2D(size=2)(layer_input)
        u = Conv2D(filters, kernel_size=f_size, strides=1, padding='same')
(u)
        u = InstanceNormalization(axis = -1, center = False, scale = False)
(u)
        u = Activation('relu')(u)
        if dropout_rate:
            u = Dropout(dropout_rate)(u)

        u = Concatenate()([u, skip_input])
        return u

    # Image input
    img = Input(shape=self.img_shape)

    # Downsampling ❶
    d1 = downsample(img, self.gen_n_filters)
    d2 = downsample(d1, self.gen_n_filters*2)
    d3 = downsample(d2, self.gen_n_filters*4)
    d4 = downsample(d3, self.gen_n_filters*8)

    # Upsampling ❷
    u1 = upsample(d4, d3, self.gen_n_filters*4)
    u2 = upsample(u1, d2, self.gen_n_filters*2)
    u3 = upsample(u2, d1, self.gen_n_filters)

    u4 = UpSampling2D(size=2)(u3)

    output = Conv2D(self.channels, kernel_size=4, strides=1
```

```
        , padding='same', activation='tanh')(u4)

    return Model(img, output)
```

❶ The generator consists of two halves. First, we downsample the image, using `Conv2D` layers with stride 2.

❷ Then we upsample, to return the tensor to the same size as the original image. The upsampling blocks contain `Concatenate` layers, which give the network the U-Net architecture.

The Discriminators

The discriminators that we have seen so far have output a single number: the predicted probability that the input image is "real." The discriminators in the CycleGAN that we will be building output an 8×8 single-channel tensor rather than a single number.

The reason for this is that the CycleGAN inherits its discriminator architecture from a model known as a *PatchGAN*, where the discriminator divides the image into square overlapping "patches" and guesses if each patch is real or fake, rather than predicting for the image as a whole. Therefore the output of the discriminator is a tensor that contains the predicted probability for each patch, rather than just a single number.

Note that the patches are predicted simultaneously as we pass an image through the network—we do not divide up the image manually and pass each patch through the network one by one. The division of the image into patches arises naturally as a result of the discriminator's convolutional architecture.

The benefit of using a PatchGAN discriminator is that the loss function can then measure how good the discriminator is at distinguishing images based on their *style* rather than their *content*. Since each individual element of the discriminator prediction is based only on a small square of the image, it must use the style of the patch, rather than its content, to make its decision. This is exactly what we require; we would rather our discriminator is good at identifying when two images differ in style than content.

The Keras code to build the discriminators is provided in Example 5-3.

Example 5-3. Building the discriminators

```
def build_discriminator(self):

    def conv4(layer_input,filters, stride = 2, norm=True):
        y = Conv2D(filters, kernel_size=4, strides=stride
               , padding='same')(layer_input)

        if norm:
            y = InstanceNormalization(axis = -1, center = False, scale =
False)(y)

        y = LeakyReLU(0.2)(y)

        return y

    img = Input(shape=self.img_shape)

    y = conv4(img, self.disc_n_filters, stride = 2, norm = False) ❶
    y = conv4(y, self.disc_n_filters*2, stride = 2)
    y = conv4(y, self.disc_n_filters*4, stride = 2)
    y = conv4(y, self.disc_n_filters*8, stride = 1)

    output = Conv2D(1, kernel_size=4, strides=1, padding='same')(y) ❷

    return Model(img, output)
```

❶ A CycleGAN discriminator is a series of convolutional layers, all with instance normalization (except the first layer).

❷ The final layer is a convolutional layer with only one filter and no activation.

Compiling the CycleGAN

To recap, we aim to build a set of models that can convert images that are in domain A (e.g., images of apples) to domain B (e.g., images of oranges) and vice versa. We therefore need to compile four distinct models, two generators and two discriminators, as follows:

`g_AB`

Learns to convert an image from domain A to domain B.

`g_BA`

Learns to convert an image from domain B to domain A.

`d_A`

Learns the difference between real images from domain A and fake images generated by `g_BA`.

`d_B`

Learns the difference between real images from domain B and fake images generated by `g_AB`.

We can compile the two discriminators directly, as we have the inputs (images from each domain) and outputs (binary responses: 1 if the image was from the domain or 0 if it was a generated fake). This is shown in Example 5-4.

Example 5-4. Compiling the discriminator

```
self.d_A = self.build_discriminator()
self.d_B = self.build_discriminator()
self.d_A.compile(loss='mse',
    optimizer=Adam(self.learning_rate, 0.5),
    metrics=['accuracy'])
self.d_B.compile(loss='mse',
    optimizer=Adam(self.learning_rate, 0.5),
    metrics=['accuracy'])
```

However, we cannot compile the generators directly, as we do not have paired images in our dataset. Instead, we judge the generators simultaneously on three criteria:

1. *Validity*. Do the images produced by each generator fool the relevant discriminator? (For example, does output from `g_BA` fool `d_A` and does output from `g_AB` fool `d_B`?)
2. *Reconstruction*. If we apply the two generators one after the other (in both directions), do we return to the original image? The CycleGAN gets its name from this *cyclic* reconstruction criterion.
3. *Identity*. If we apply each generator to images from its own target domain, does the image remain unchanged?

Example 5-5 shows how we can compile a model to enforce these three criteria (the numeric markers in the code correspond to the preceding list).

Example 5-5. Building the combined model to train the generators

```
self.g_AB = self.build_generator_unet()
self.g_BA = self.build_generator_unet()

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

img_A = Input(shape=self.img_shape)
img_B = Input(shape=self.img_shape)
fake_A = self.g_BA(img_B)
fake_B = self.g_AB(img_A)

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

reconstr_A = self.g_BA(fake_B)
reconstr_B = self.g_AB(fake_A) ❷

img_A_id = self.g_BA(img_A)
img_B_id = self.g_AB(img_B) ❸

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=[
                          self.lambda_validation
                          , self.lambda_validation
                          , self.lambda_reconstr
                          , self.lambda_reconstr
                          , self.lambda_id
                          , self.lambda_id
                      ],
                      optimizer=optimizer)
```

The combined model accepts a batch of images from each domain as input and provides three outputs (to match the three criteria) for each domain—so, six outputs in total. Notice how we freeze the weights in the discriminator, as

is typical with GANs, so that the combined model only trains the generator weights, even though the discriminator is involved in the model.

The overall loss is the weighted sum of the loss for each criterion. Mean squared error is used for the validity criterion—checking the output from the discriminator against the real (1) or fake (0) response—and mean absolute error is used for the image-to-image-based criteria (reconstruction and identity).

Training the CycleGAN

With our discriminators and combined model compiled, we can now train our models. This follows the standard GAN practice of alternating the training of the discriminators with the training of the generators (through the combined model).

In Keras, the code in Example 5-6 describes the training loop.

Example 5-6. Training the CycleGAN

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

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

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

        fake_B = self.g_AB.predict(imgs_A) ❷
        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)
```

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

❶ We use a response of 1 for real images and 0 for generated images. Notice how there is one response per patch, as we are using a PatchGAN discriminator.

❷ To train the discriminators, we first use the respective generator to create a batch of fake images, then we train each discriminator on this fake set and a batch of real images. Typically, for a CycleGAN the batch size is 1 (a single image).

❸ The generators are trained together in one step, through the combined model compiled earlier. See how the six outputs match to the six loss functions defined earlier during compilation.

Analysis of the CycleGAN

Let's see how the CycleGAN performs on our simple dataset of apples and oranges and observe how changing the weighting parameters in the loss function can have dramatic effects on the results.

We have already seen an example of the output from the CycleGAN model in Figure 5-3. Now that you are familiar with the CycleGAN architecture, you might recognize that this image represents the three criteria through which the combined model is judged: validity, reconstruction, and identity.

Let's relabel this image with the appropriate functions from the codebase, so that we can see this more explicitly (Figure 5-8).

We can see that the training of the network has been successful, because each generator is visibly altering the input picture to look more like a valid image from the opposite domain. Moreover, when the generators are applied one after the other, the difference between the input image and the reconstructed image is minimal. Finally, when each generator is applied to an image from its own input domain, the image doesn't change significantly.

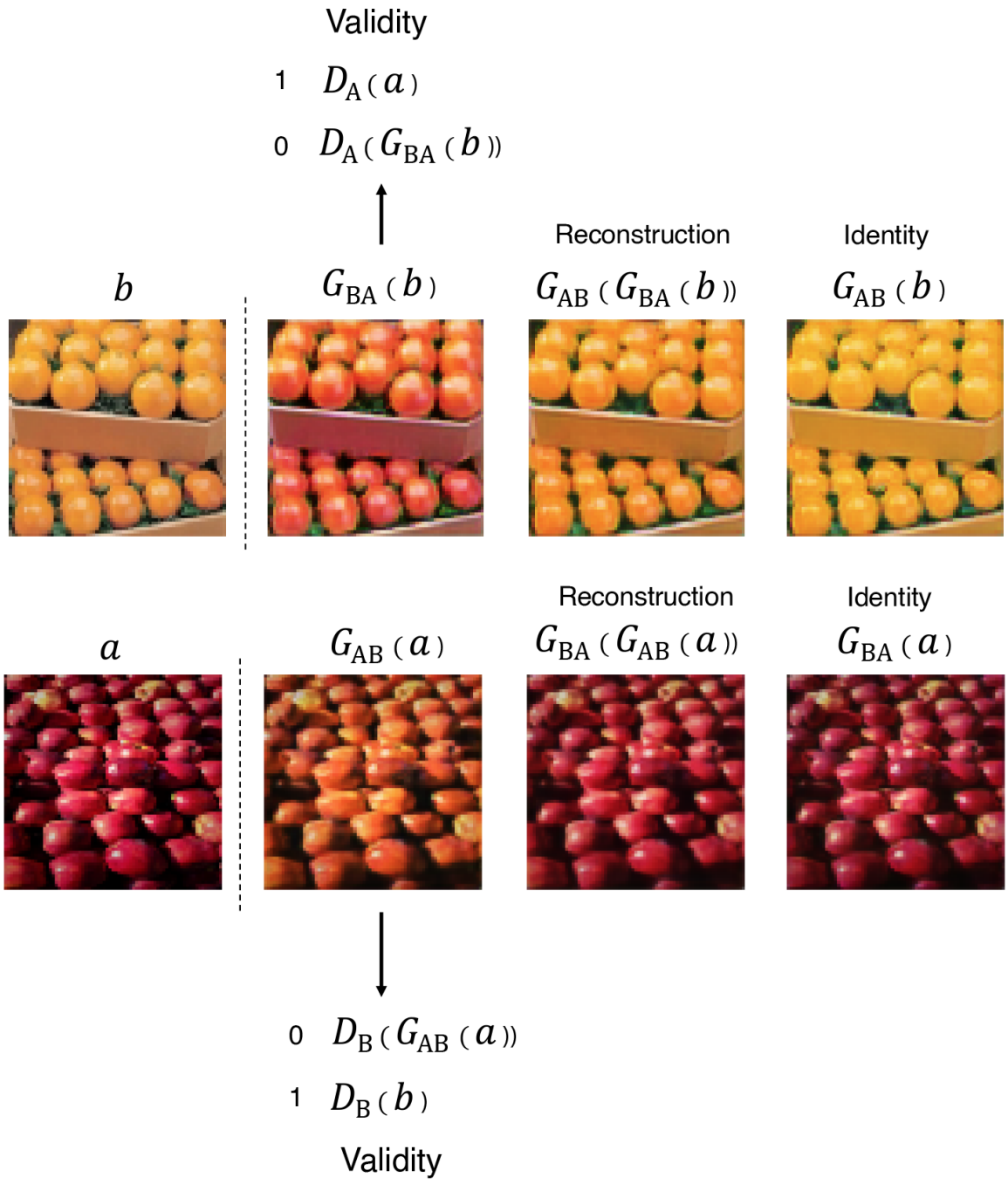

Figure 5-8. Outputs from the combined model used to calculated the overall loss function

In the original CycleGAN paper, the identity loss was included as an optional addition to the necessary reconstruction loss and validity loss. To demonstrate the importance of the identity term in the loss function, let's see what happens if we remove it, by setting the identity loss weighting parameter to zero in the loss function (Figure 5-9).

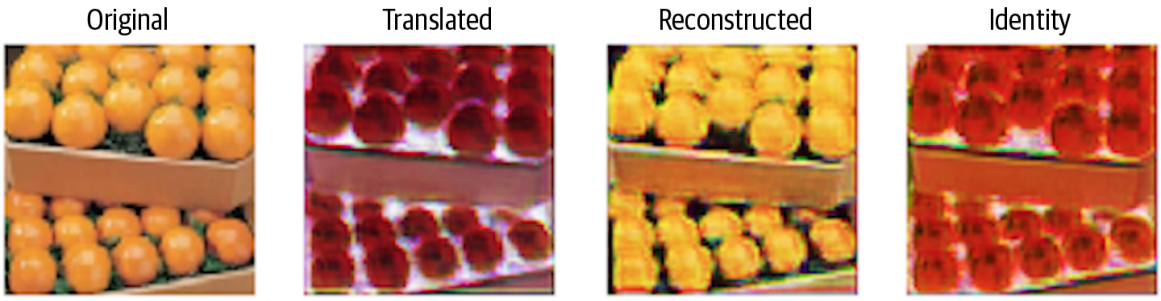

Figure 5-9. Output from the CycleGAN when the identity loss weighting is set to zero

The CycleGAN has still managed to translate the oranges into apples but the color of the tray holding the oranges has flipped from black to white, as there is now no identity loss term to prevent this shift in background colors. The identity term helps regulate the generator to ensure that it only adjust parts of the image that are necessary to complete the transformation and no more.

This highlights the importance of ensuring the weightings of the three loss functions are well balanced—too little identity loss and the color shift problem appears; too much identity loss and the CycleGAN isn't sufficiently incentivized to change the input to look like an image from the opposite domain.

Creating a CycleGAN to Paint Like Monet

Now that we have explored the fundamental structure of a CycleGAN, we can turn our attention to more interesting and impressive applications of the technique.

In the original CycleGAN paper, one of the standout achievements was the ability for the model to learn how to convert a given photo into a painting in the style of a particular artist. As this is a CycleGAN, the model is also able to translate the other way, converting an artist's paintings into realistic-looking photographs.

To download the Monet-to-photo dataset, run the following command from inside the book repository:

```
bash ./scripts/download_cyclegan_data.sh monet2photo
```

This time we will use the parameter set shown in Example 5-7 to build the model:

Example 5-7. Defining the Monet CycleGAN

```
gan = CycleGAN(
        input_dim = (256,256,3)
        , learning_rate = 0.0002
        , lambda_validation = 1
        , lambda_reconstr = 10
        , lambda_id = 5
        , generator_type = 'resnet'
        , gen_n_filters = 32
        , disc_n_filters = 64
        )
```

The Generators (ResNet)

In this example, we shall introduce a new type of generator architecture: a residual network, or ResNet.[10] The ResNet architecture is similar to a U-Net in that it allows information from previous layers in the network to skip ahead one or more layers. However, rather than creating a U shape by connecting layers from the downsampling part of the network to corresponding upsampling layers, a ResNet is built of residual blocks stacked on top of each other, where each block contains a skip connection that sums the input and output of the block, before passing this on to the next layer. A single residual block is shown in Figure 5-10.

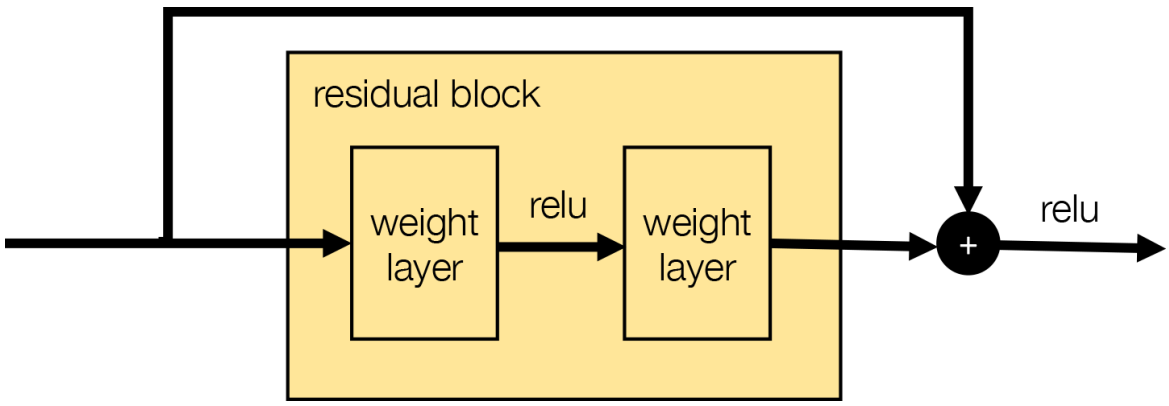

Figure 5-10. A single residual block

In our CycleGAN, the "weight layers" in the diagram are convolutional layers with instance normalization. In Keras, a residual block can be coded as shown in Example 5-8.

Example 5-8. A residual block in Keras

```
from keras.layers.merge import add

def residual(layer_input, filters):
    shortcut = layer_input
    y = Conv2D(filters, kernel_size=(3, 3), strides=1, padding='same')
(layer_input)
    y = InstanceNormalization(axis = -1, center = False, scale = False)(y)
    y = Activation('relu')(y)

    y = Conv2D(filters, kernel_size=(3, 3), strides=1, padding='same')(y)
    y = InstanceNormalization(axis = -1, center = False, scale = False)(y)

    return add([shortcut, y])
```

On either side of the residual blocks, our ResNet generator also contains downsampling and upsampling layers. The overall architecture of the ResNet is shown in Figure 5-11.

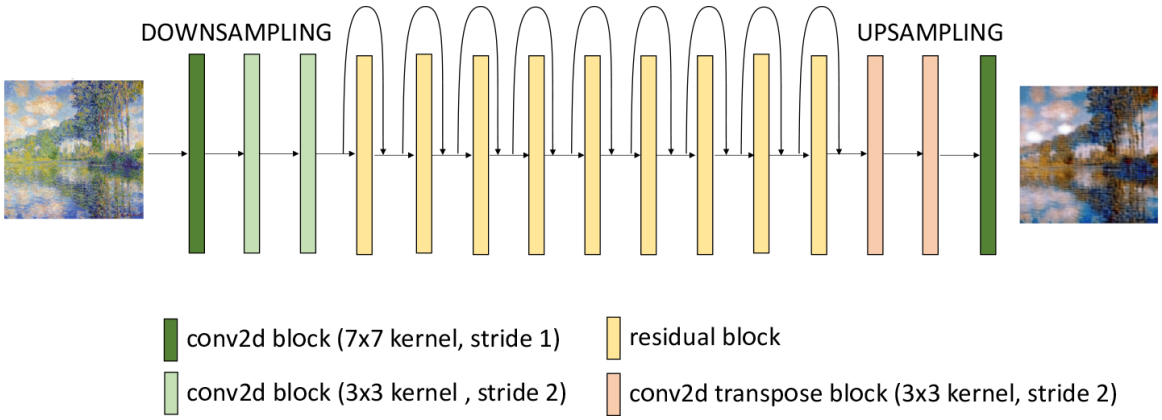

Figure 5-11. A ResNet generator

It has been shown that ResNet architectures can be trained to hundreds and even thousands of layers deep and not suffer from the *vanishing gradient* problem, where the gradients at early layers are tiny and therefore train very slowly. This is due to the fact that the error gradients can backpropagate freely through the network through the skip connections that are part of the residual blocks. Furthermore, it is believed that adding additional layers

never results in a drop in model accuracy, as the skip connections ensure that it is always possible to pass through the identity mapping from the previous layer, if no further informative features can be extracted.

Analysis of the CycleGAN

In the original CycleGAN paper, the model was trained for 200 epochs to achieve state-of-the-art results for artist-to-photograph style transfer. In Figure 5-12 we show the output from each generator at various stages of the early training process, to show the progression as the model begins to learn how to convert Monet paintings into photographs and vice versa.

In the top row, we can see that gradually the distinctive colors and brushstrokes used by Monet are transformed into the more natural colors and smooth edges that would be expected in a photograph. Similarly, the reverse is happening in the bottom row, as the generator learns how to convert a photograph into a scene that Monet might have painted himself.

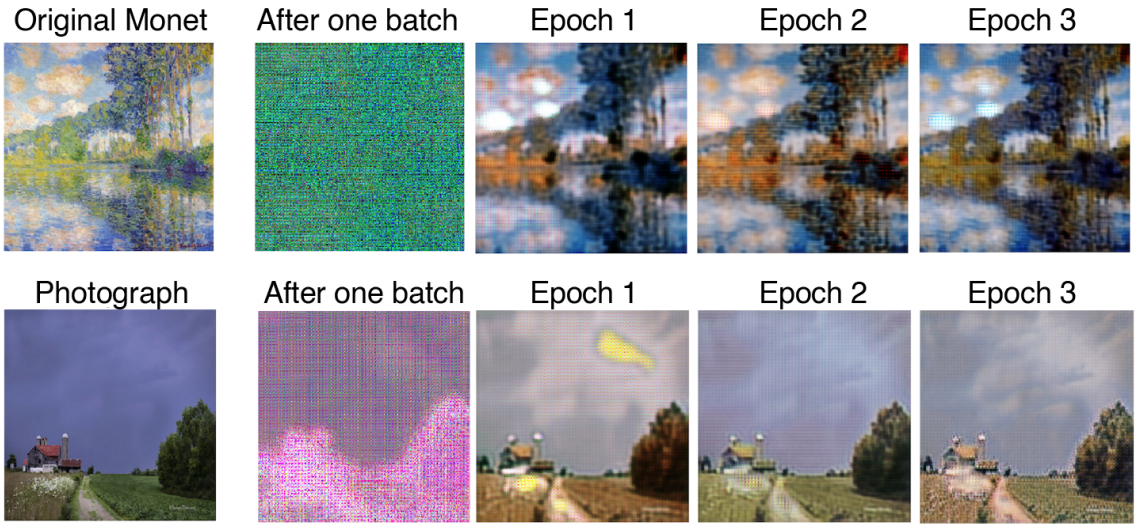

Figure 5-12. Output at various stages of the training process

Figure 5-13 shows some of the results from the original paper achieved by the model after it was trained for 200 epochs.

Figure 5-13. Output after 200 epochs of training[11]

Neural Style Transfer

So far, we have seen how a CycleGAN can transpose images between two domains, where the images in the training set are not necessarily paired. Now we shall look at a different application of style transfer, where we do not have a training set at all, but instead wish to transfer the style of one single image onto another, as shown in Figure 5-14. This is known as *neural style transfer*.[12]

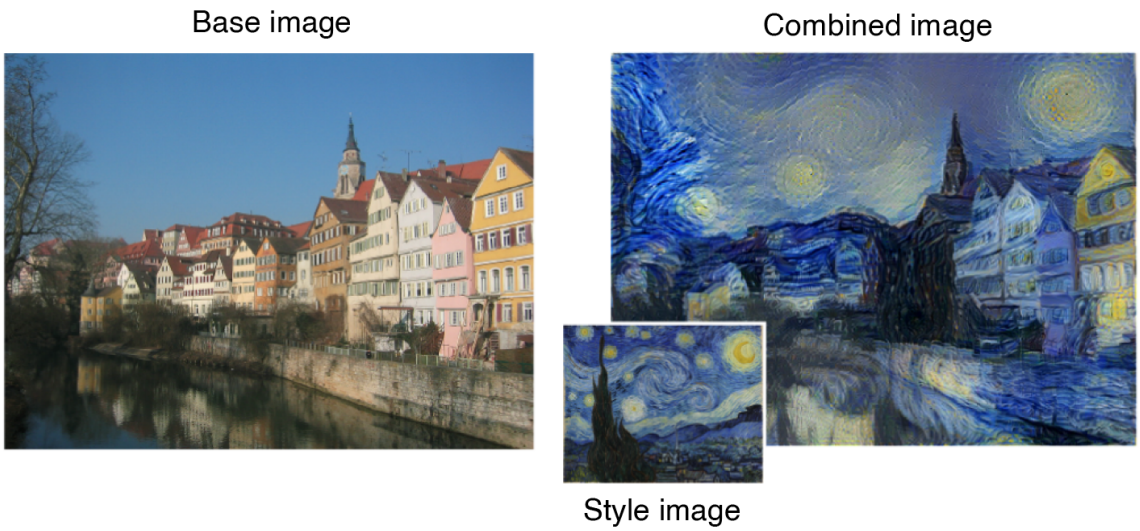

Figure 5-14. An example of neural style transfer[13]

The idea works on the premise that we want to minimize a loss function that is a weighted sum of three distinct parts:

Content loss

> We would like the combined image to contain the same content as the base image.

Style loss

> We would like the combined image to have the same general style as the style image.

Total variance loss

> We would like the combined image to appear smooth rather than pixelated.

We minimize this loss via gradient descent—that is, we update each pixel value by an amount proportional to the negative gradient of the loss function, over many iterations. This way, the loss gradually decreases with each iteration and we end up with an image that merges the content of one image with the style of another.

Optimizing the generated output via gradient descent is different to how we have tackled generative modeling problems thus far. Previously we have

trained a deep neural network such as a VAE or GAN by backpropagating the error through the entire network to learn from a training set of data and generalize the information learned to generate new images. Here, we cannot take this approach as we only have two images to work with, the base image and the style image. However, as we shall see, we can still make use of a pretrained deep neural network to provide vital information about each image inside the loss functions.

We'll start by defining the three individual loss functions, as they are the core of the neural style transfer engine.

Content Loss

The content loss measures how different two images are in terms of the subject matter and overall placement of their content. Two images that contain similar-looking scenes (e.g., a photo of a row of buildings and another photo of the same buildings taken in different light from a different angle) should have a smaller loss than two images that contain completely different scenes. Simply comparing the pixel values of the two images won't do, because even in two distinct images of the same scene, we wouldn't expect individual pixel values to be similar. We don't really want the content loss to care about the values of individual pixels; we'd rather that it scores images based on the presence and approximate position of higher-level features such as *buildings*, *sky*, or *river*.

We've seen this concept before. It's the whole premise behind deep learning —a neural network trained to recognize the content of an image naturally learns higher-level features at deeper layers of the network by combining simpler features from previous layers. Therefore, what we need is a deep neural network that has already been successfully trained to identify the content of an image, so that we can tap into a deep layer of the network to extract the high-level features of a given input image. If we measure the mean squared error between this output for the base image and the current combined image, we have our content loss function!

The pretrained network that we shall be using is called VGG19. This is a 19-layer convolutional neural network that has been trained to classify images into one thousand object categories on more than one million images from the ImageNet dataset. A diagram of the network is shown in Figure 5-15.

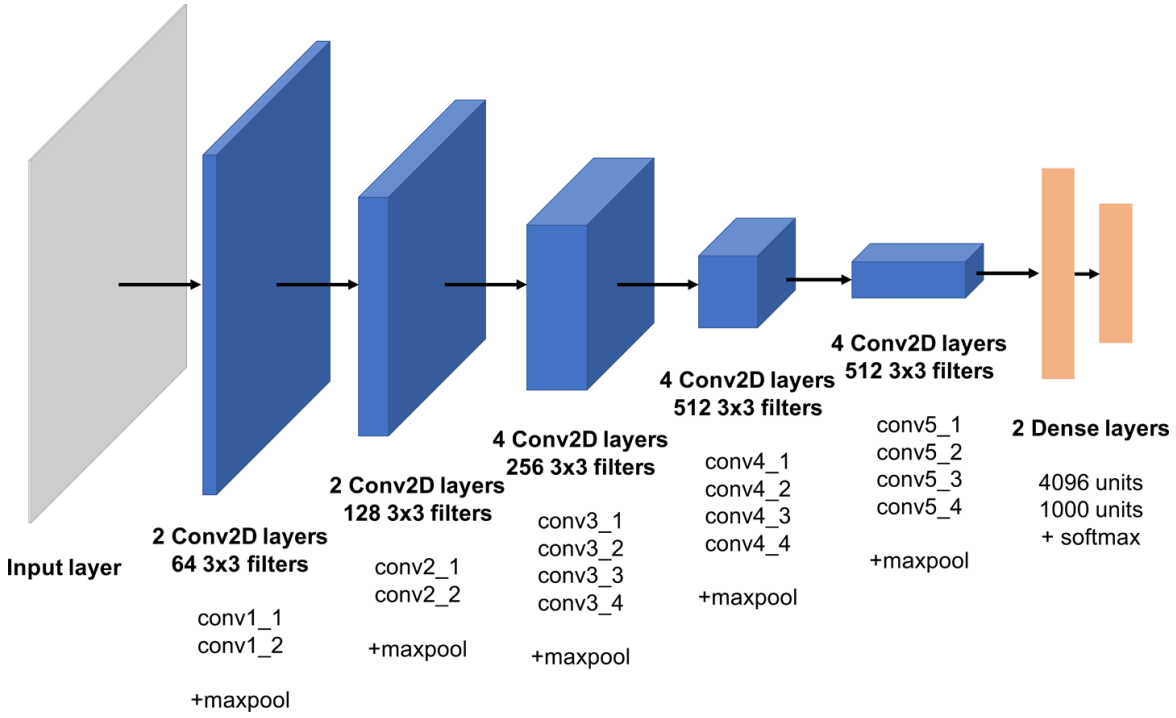

Figure 5-15. The VGG19 model

Example 5-9 is a code snippet that calculates the content loss between two images, adapted from the neural style transfer example in the official Keras repository. If you want to re-create this technique and experiment with the parameters, I suggest working from this repository as a starting point.

Example 5-9. The content loss function

```
from keras.applications import vgg19 ❶
from keras import backend as K

base_image_path = '/path_to_images/base_image.jpg'
style_reference_image_path = '/path_to_images/styled_image.jpg'

content_weight = 0.01

base_image = K.variable(preprocess_image(base_image_path)) ❷
style_reference_image =
K.variable(preprocess_image(style_reference_image_path))
```

```
combination_image = K.placeholder((1, img_nrows, img_ncols, 3))

input_tensor = K.concatenate([base_image,
                              style_reference_image,
                              combination_image], axis=0) ❸

model = vgg19.VGG19(input_tensor=input_tensor,
                    weights='imagenet', include_top=False) ❹

outputs_dict = dict([(layer.name, layer.output) for layer in model.layers])
layer_features = outputs_dict['block5_conv2'] ❺

base_image_features = layer_features[0, :, :, :]
combination_features = layer_features[2, :, :, :] ❻

def content_loss(content, gen):
    return K.sum(K.square(gen - content))

content_loss = content_weight * content_loss(base_image_features
                                             , combination_features) ❼
```

❶ The Keras library contains a pretrained VGG19 model that can be imported.

❷ We define two Keras variables to hold the base image and style image and a placeholder that will contain the generated combined image.

❸ The input tensor to the VGG19 model is a concatenation of the three images.

❹ Here we create an instance of the VGG19 model, specifying the input tensor and the weights that we would like to preload. The `include_top = False` parameter specifies that we do not need to load the weights for the final dense layers of the networks that result in the classification of the image. This is because we are only interested in the preceding convolutional layers, which capture the high-level features of an input image, not the actual probabilities that the original model was trained to output.

❺ The layer that we use to calculate the content loss is the second convolutional layer of the fifth block. Choosing a layer at a shallower or deeper point in the network affects how the loss function defines "content" and therefore alters the properties of the generated combined image.

❻ Here we extract the base image features and combined image features from the input tensor that has been fed through the VGG19 network.

❼ The content loss is the sum of squares distance between the outputs of the chosen layer for both images, multiplied by a weighting parameter.

Style Loss

Style loss is slightly more difficult to quantify—how can we measure the similarity in style between two images?

The solution given in the neural style transfer paper is based on the idea that images that are similar in style typically have the same pattern of correlation between feature maps in a given layer. We can see this more clearly with an example.

Suppose in the VGG19 network we have some layer where one channel has learned to identify parts of the image that are colored green, another channel has learned to identify spikiness, and another has learned to identify parts of the image that are brown.

The output from these channels (feature maps) for three inputs is shown in Figure 5-16.

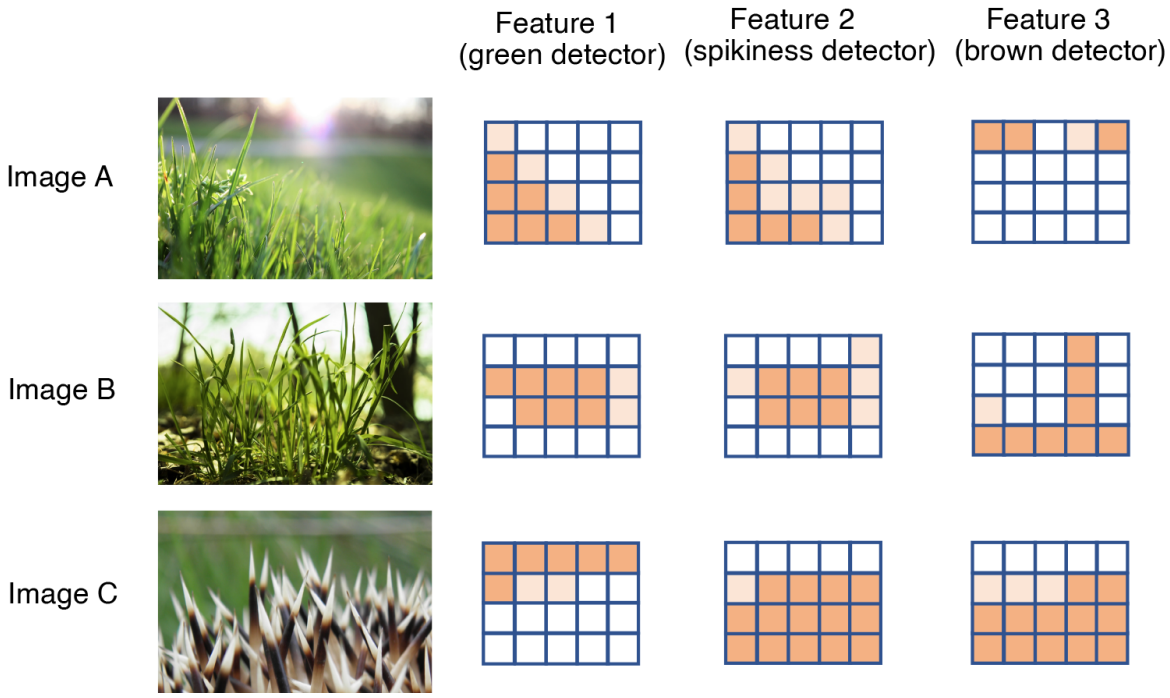

Figure 5-16. The output from three channels (feature maps) for three given input images—the darker orange colors represent larger values

We can see that A and B are similar in style—both are *grassy*. Image C is slightly different in style to images A and B. If we look at the feature maps, we can see that the *green* and *spikiness* channels often fire strongly together at the same spatial point in images A and B, but not in image C. Conversely, the *brown* and *spikiness* channels are often activated together at the same point in image C, but not in images A and B. To numerically measure how much two feature maps are jointly activated together, we can flatten them and calculate the dot product.[14] If the resulting value is high, the feature maps are highly correlated; if the value is low, the feature maps are not correlated.

We can define a matrix that contains the dot product between all possible pairs of features in the layer. This is called a *Gram matrix*. Figure 5-17 shows the Gram matrices for the three features, for each of the images.

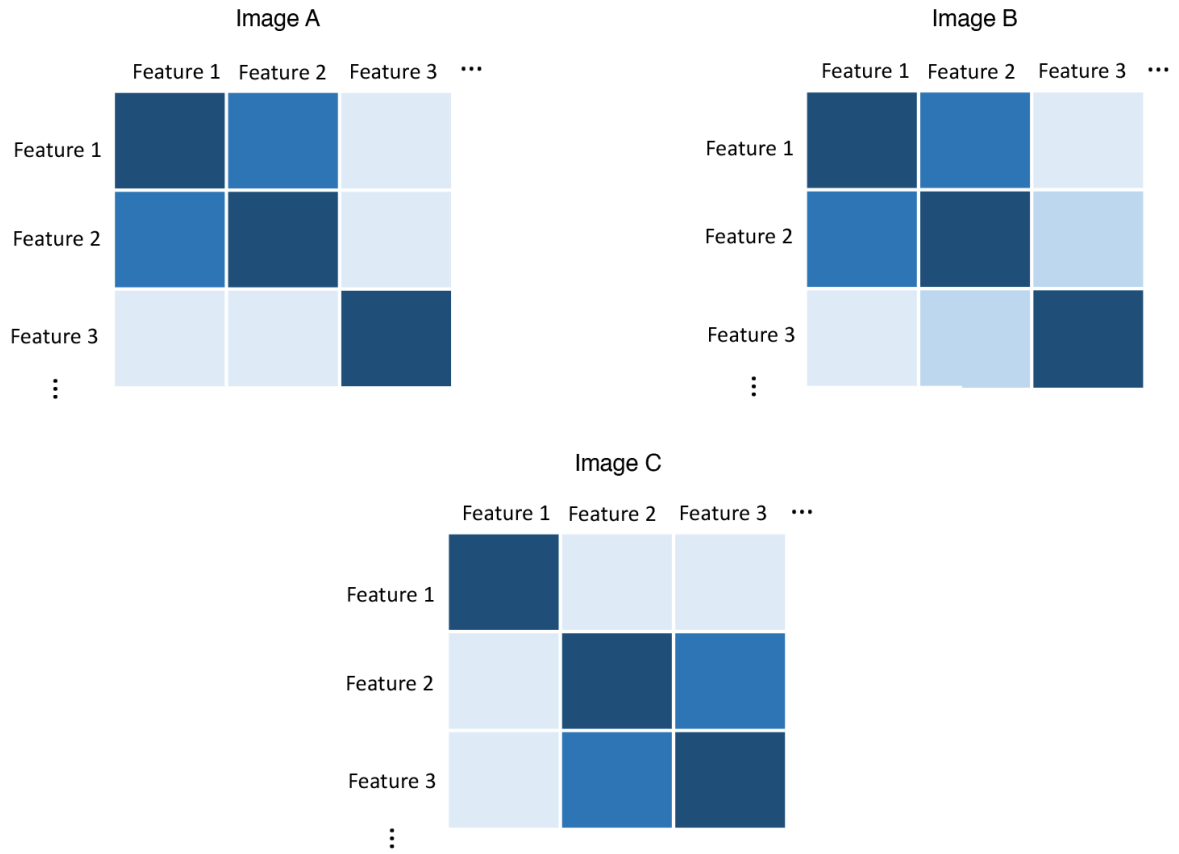

Figure 5-17. Parts of the Gram matrices for the three images—the darker blue colors represent larger values

It is clear that images A and B, which are similar in style, have similar Gram matrices for this layer. Even though their content may be very different, the Gram matrix—a measure of correlation between all pairs of features in the layer—is similar.

Therefore to calculate the style loss, all we need to do is calculate the Gram matrix (GM) for a set of layers throughout the network for both the base image and the combined image and compare their similarity using sum of squared errors. Algebraically, the style loss between the base image (S) and the generated image (G) for a given layer (l) of size M_l (height x width) with N_l channels can be written as follows:

$$L_{GM}(S,G,l) = \frac{1}{4N_l^2 M_l^2} \sum_{ij} \left(GM[l](S)_{ij} - GM[l](G)_{ij} \right)^2$$

Notice how this is scaled to account for the number of channels (N_l) and size of the layer (M_l). This is because we calculate the overall style loss as a weighted sum across several layers, all of which have different sizes. The total style loss is then calculated as follows:

$$L_{style}(S, G) = \sum_{l=0}^{L} w_l L_{GM}(S, G, l)$$

In Keras, the style loss calculations can be coded as shown in Example 5-10.[15]

Example 5-10. The style loss function

```
style_loss = 0.0

def gram_matrix(x):
    features = K.batch_flatten(K.permute_dimensions(x, (2, 0, 1)))
    gram = K.dot(features, K.transpose(features))
    return gram

def style_loss(style, combination):
    S = gram_matrix(style)
    C = gram_matrix(combination)
    channels = 3
    size = img_nrows * img_ncols
    return K.sum(K.square(S - C)) / (4.0 * (channels ** 2) * (size ** 2))


feature_layers = ['block1_conv1', 'block2_conv1',
                  'block3_conv1', 'block4_conv1',
                  'block5_conv1'] ❶

for layer_name in feature_layers:
    layer_features = outputs_dict[layer_name]
    style_reference_features = layer_features[1, :, :, :] ❷
    combination_features = layer_features[2, :, :, :]
    sl = style_loss(style_reference_features, combination_features)
    style_loss += (style_weight / len(feature_layers)) * sl ❸
```

❶ The style loss is calculated over five layers—the first convolutional layer in each of the five blocks of the VGG19 model.

❷ Here we extract the style image features and combined image features from the input tensor that has been fed through the VGG19 network.

❸ The style loss is scaled by a weighting parameter and the number of layers that it is calculated over.

Total Variance Loss

The total variance loss is simply a measure of noise in the combined image. To judge how noisy an image is, we can shift it one pixel to the right and calculate the sum of the squared difference between the translated and original images. For balance, we can also do the same procedure but shift the image one pixel down. The sum of these two terms is the total variance loss.

Example 5-11 shows how we can code this in Keras.[16]

Example 5-11. The variance loss function

```
def total_variation_loss(x):
    a = K.square(
        x[:, :img_nrows - 1, :img_ncols - 1, :] - x[:, 1:, :img_ncols - 1,
:]) ❶
    b = K.square(
        x[:, :img_nrows - 1, :img_ncols - 1, :] - x[:, :img_nrows - 1, 1:,
:]) ❷
    return K.sum(K.pow(a + b, 1.25))

tv_loss = total_variation_weight * total_variation_loss(combination_image)
❸

loss = content_loss + style_loss + tv_loss ❹
```

❶ The squared difference between the image and the same image shifted one pixel down.

❷ The squared difference between the image and the same image shifted one pixel to the right.

❸ The total variance loss is scaled by a weighting parameter.

❹ The overall loss is the sum of the content, style, and total variance losses.

Running the Neural Style Transfer

The learning process involves running gradient descent to minimize this loss function, with respect to the pixels in the combined image. The full code for this is included in the, *neural_style_transfer.py* script included as part of the

official Keras repository. Example 5-12 is a code snippet showing the training loop.

Example 5-12. The training loop for the neural style transfer model

```
from scipy.optimize import fmin_l_bfgs_b

iterations = 1000
x = preprocess_image(base_image_path) ❶

for i in range(iterations):
    x, min_val, info = fmin_l_bfgs_b( ❷
        evaluator.loss ❸
        , x.flatten()
        , fprime=evaluator.grads ❸
        , maxfun=20
        )
```

❶ The process is initialized with the base image as the starting *combined* image.

❷ At each iteration we pass the current combined image (flattened) into a optimization function, `fmin_l_bfgs_b` from the `scipy.optimize` package, that performs one gradient descent step according to the L-BFGS-B algorithm.

❸ Here, `evaluator` is an object that contains methods that calculate the overall loss, as described previously, and gradients of the loss with respect to the input image.

Analysis of the Neural Style Transfer Model

Figure 5-18 shows the output of the neural style transfer process at three different stages in the learning process, with the following parameters:

- `content_weight`: `1`
- `style_weight`: `100`
- `total_variation_weight`: `20`

Figure 5-18. Output from the neural style transfer process at 1, 200, and 400 iterations

We can see that with each training step, the algorithm becomes stylistically closer to the style image and loses the detail of the base image, while retaining the overall content structure.

There are many ways to experiment with this architecture. You can try changing the weighting parameters in the loss function or the layer that is used to determine the content similarity, to see how this affects the combined output image and training speed. You can also try decaying the weight given to each layer in the style loss function, to bias the model toward transferring finer or coarser style features.

Summary

In this chapter, we have explored two different ways to generate novel artwork: CycleGAN and neural style transfer.