8 Telling things apart: Image segmentation

This chapter covers

- Understanding segmentation data and working with it in Python
- Implementing a fully fledged segmentation data pipeline
- Implementing an advanced segmentation model (DeepLab v3)
- Compiling models with custom-built image segmentation loss functions/metrics
- Training the image segmentation model on the clean and processed image data
- Evaluating the trained segmentation model

In the last chapter, we learned about various advanced computer vision models and techniques to push the performance of an image classifier. We learned about the architecture of Inception net v1 as well as its successors (e.g., Inception net v2, v3, and v4). Our objective was to lift the performance of the model on an image classification data set with 64 × 64–sized RGB images of objects belonging to 200

different classes. While trying to train a model on this data set, we learned many important concepts:

- *Inception blocks*—A way to group convolutional layers having different-sized windows (or kernels) to encourage learning features at different scales while making the model parameter efficient due to the smaller-sized kernels.
- *Auxiliary outputs*—Inception net uses a classification layer (i.e., a fully connected layer with softmax activation) not only at the end of the network, but also in the middle of the network. This enables the gradients from the final layer to flow strongly all the way to the first layer.
- *Augmenting data*—Using various image transformation techniques (adjusting brightness/contrast, rotating, translating, etc.) to increase the amount of labeled data using the `tf.keras.preprocessing.image.ImageDataGenerator`.
- *Dropout*—Switching on and off nodes in the layers randomly. This forces the neural networks to learn more robust features as the network does not always have all the nodes activated.
- *Early stopping*—Using the performance on the validation data set as a way to control when the training stops. If the validation performance has not increased in a certain number of epochs, training is halted.
- *Transfer learning*—Downloading and using a pretrained model (e.g., Inception-ResNet v2) trained on a larger, similar data set as the initialization and fine-tuning it to perform well on the task at hand.

In this chapter, we will learn about another important task in computer vision: image segmentation. In image classification, we only care if an object exists in a given image. Image segmentation, on the other hand, recognizes multiple objects in the same image as well as where they are in the image. It is a very important topic of computer vision, and applications like self-driving cars live and breathe image segmentation models. Self-driving cars need to precisely locate objects in their surroundings, which is where image segmentation comes into play. As you might have guessed already, they also have their roots in many other applications:

- Image retrieval
- Identifying galaxies (http://mng.bz/gwVx)
- Medical image analysis

If you are a computer vision/deep learning engineer/researcher working on image-related problems, there is a high chance that your path will cross with image segmentation. Image segmentation models classify each pixel in the image to one of a predefined set of object categories. Image segmentation has ties to the image classification task we saw earlier. Both solve a classification task. Additionally, pretrained image classification models are used as the backbone of segmentation models, as they can provide crucial image features at different granularities to solve the segmentation task better and faster. A key difference is that image classifiers are solving a sparse prediction task,

where each image has a single class label associated, as opposed to segmentation models that solve a dense prediction task that has a class label associated with every pixel in the image.

Any image segmentation algorithm can be classified as one of the following:

- *Semantic segmentation*—The algorithm is only interested in identifying different categories of objects present in the image. For example, if there are multiple persons in the image, the pixels corresponding to all of them will be tagged with the same class.
- *Instance segmentation*—The algorithm is interested in identifying different objects separately. For example, if there are multiple persons in the image, pixels belonging to each person are represented by a unique class. Instance-based segmentation is considered more difficult than semantic segmentation.

Figure 8.1 depicts the difference between the data found in a semantic segmentation task and an instance-based segmentation task. In this chapter, we will focus on semantic segmentation (http://mng.bz/5QAZ).

Figure 8.1 Semantic segmentation versus instance segmentation

In the next section, we will look at the data we are dealing with more closely.

8.1 Understanding the data

You are experimenting with a startup idea. The idea is to develop a navigation algorithm for small remote-control (RC) toys. Users can choose between how safe or adventurous the navigation needs to be. As the first step, you plan to develop an image segmentation model. The output of the image segmentation model will later feed to a different model that will predict the navigation path depending on what the user requests.

For this task, you feel the Pascal VOC 2012 data set will be a good fit as it mostly comprises indoor and outdoor images that are found in urban/domestic environments. It contains pairs of images: an input image containing some objects and an annotated image. In the annotated image, each pixel has an assigned color, depending on which object that pixel belongs to. Here, you plan to download the data set and load the data successfully into Python.

After having a good understanding/framing of the problem you want to solve, your next focus point should be understanding and exploring the data. Segmentation data is different from the image classification data sets we've seen thus far. One major difference is that both the input and target are images. The input image is a standard image, similar to what you'd find in an image classification task. Unlike in image classification, the target is not a label, but an image, where each pixel has a color from a predefined palette of colors. In other words, each object we're interested in segmenting is assigned a color. Then a pixel corresponding to that object in the input image is colored with that color. The number of available colors is the same as the number of different objects (plus background) that you're interested in identifying (figure 8.2).

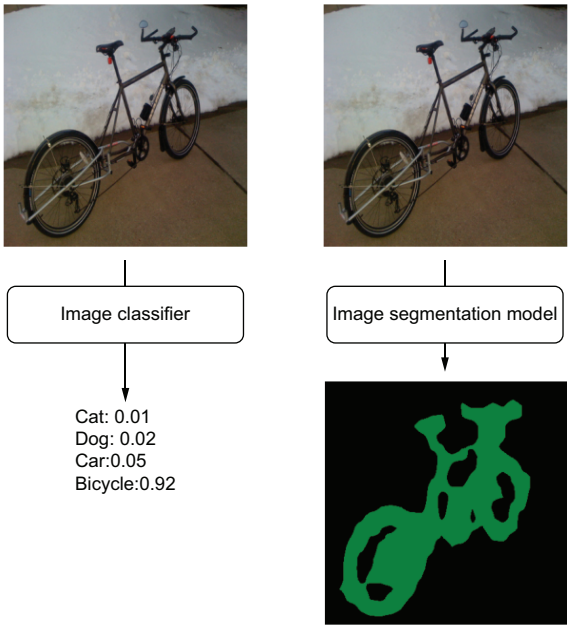

Figure 8.2 Inputs and outputs of an image classifier versus an image segmentation model

For this task, we will be using the PASCAL VOC 2012 data set, which is popular and consists of real-world scenes. The data set has labels for 22 different classes, as outlined in table 8.1.

Table 8.1 Different classes and their respective labels in the PASCAL VOC 2012 data set

Class	Assigned Label	Class	Assigned Label
Background	0	Dining table	11
Aeroplane	1	Dog	12
Bicycle	2	Horse	13

Table 8.1 Different classes and their respective labels in the PASCAL VOC 2012 data set *(continued)*

Class	Assigned Label	Class	Assigned Label
Bird	3	Motorbike	14
Boat	4	Person	15
Bottle	5	Potted plant	16
Bus	6	Sheep	17
Car	7	Sofa	18
Cat	8	Train	19
Chair	9	TV/monitor	20
Cow	10	Boundaries/unknown object	255

The white pixels represent object boundaries or unknown objects. Figure 8.3 illustrates the data set by showing a sample for every single object class present.

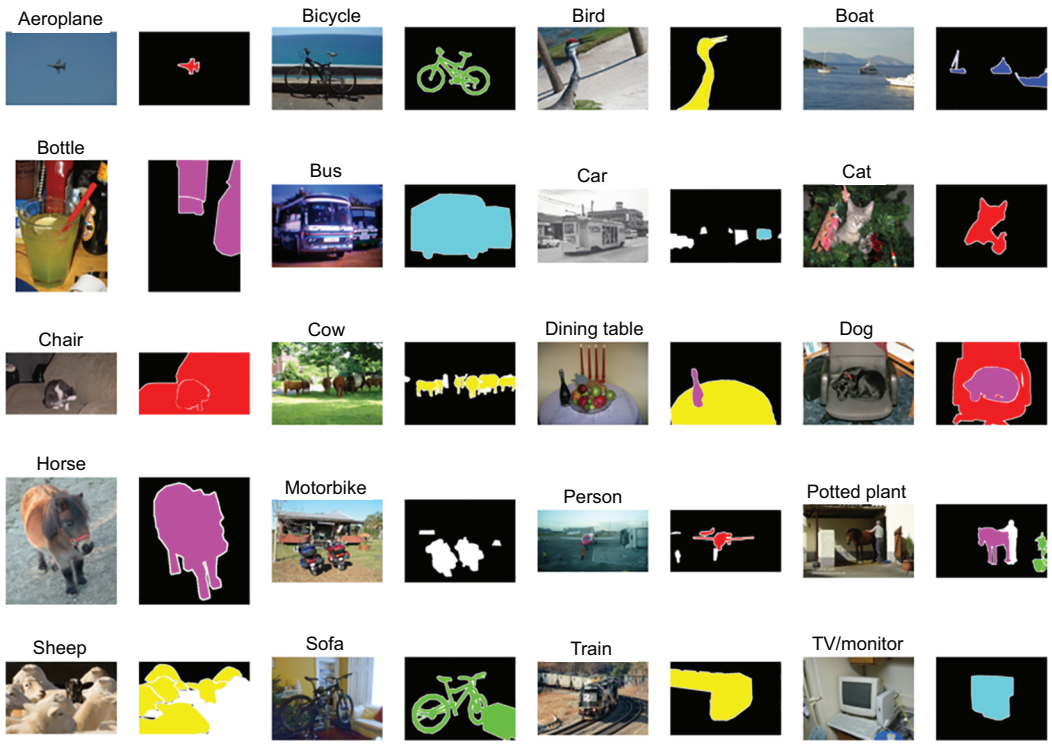

Figure 8.3 Samples from the PASCAL VOC 2012 data set. The data set shows a single example image, along with the annotated segmentation of it for the 20 different object classes.

In figure 8.4, diving a bit deeper, you can see a single sample datapoint (best viewed in color) up close. It has two objects: a chair and a dog. As it is shown, different colors are assigned to different object categories. While the figure is best viewed in color, you still can distinguish different objects by paying attention to the white border that outlines the objects in the figure.

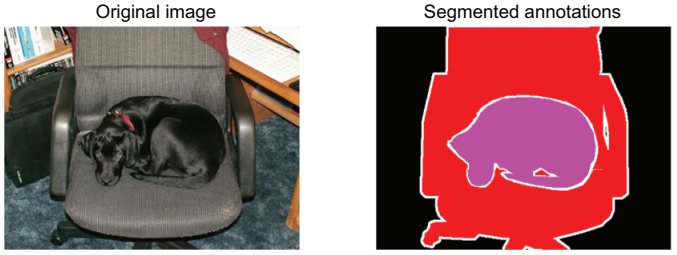

Figure 8.4 An original input image in image segmentation and the corresponding target annotated/segmented image

First, we'll download the data set, if it does not exist, from http://mng.bz/6XwZ (see the next listing).

Listing 8.1 Downloading data

```
import os
import requests
import tarfile

# Retrieve the data
if not os.path.exists(os.path.join('data','VOCtrainval_11-May-2012.tar')):
    url = "http://host.robots.ox.ac.uk/pascal/VOC/voc2012/VOCtrainval_11-
➥ May-2012.tar"
    # Get the file from web
    r = requests.get(url)

    if not os.path.exists('data'):
        os.mkdir('data')

    # Write to a file
    with open(os.path.join('data','VOCtrainval_11-May-2012.tar'), 'wb') as f:
        f.write(r.content)
else:
    print("The tar file already exists.")

if not os.path.exists(os.path.join('data', 'VOCtrainval_11-May-2012')):
    with tarfile.open(os.path.join('data','VOCtrainval_11-May-2012.tar'),
     'r') as tar:
        tar.extractall('data')
else:
    print("The extracted data already exists")
```

Check if the file is already downloaded. If so, don't download again.

Get the content from the URL.

Save the file to disk.

If the file exists but is not extracted, extract the file.

The data set download is quite similar to our past experience. The data exists as a tarfile. We download the file if it doesn't exist and extract it. Next, we will discuss how to use the image library Pillow and NumPy to load the images into memory. Here, the

target images will need special treatment, as you will see that they are not stored using the conventional approach. There are no surprises involved with loading input images to memory. Using the `PIL` (i.e., Pillow) library, they can be loaded with a single line of code:

```
from PIL import Image

orig_image_path = os.path.join('data', 'VOCtrainval_11-May-2012',
➥ 'VOCdevkit', 'VOC2012', 'JPEGImages', '2007_000661.jpg')

orig_image = Image.open(orig_image_path)
```

Next, you can inspect the image's attributes:

```
print("The format of the data {}".format(orig_image.format))
>>> The format of the data JPEG

print("This image is of size: {}".format(orig_image.shape))
>>> This image is of size: (375, 500, 3)
```

It's time to load the corresponding annotated/segmented target images. As mentioned earlier, target images require special attention. The target images are not stored as standard images but as *palettized* images. Palettization is a technique to reduce memory footprint while storing images with a fixed number of colors in the image. The crux of the method is to maintain a palette of colors. The palette is stored as a sequence of integers, which has a length of the number of colors or the number of channels. (E.g., in the case of RGB, where a pixel is made of three values corresponding to red, green, and blue, the number of channels is three. A grayscale image has a single channel, where each pixel is made of a single value). The image itself then stores an array of indices (size = height × width), where each index maps to a color in the palette. Finally, by mapping the palette indices from the image to palette colors, you can compute the original image. Figure 8.5 provides a visual exposition of this discussion.

The next listing shows the code for reconstructing the original image pixels from the palettized image.

Listing 8.2 Reconstructing the original image from a palettized image

The palette is stored as a vector. We reshape it to an array, where each row represents a single RGB color.

```
def rgb_image_from_palette(image):

    """ This function restores the RGB values form a palletted PNG image """
    palette = image.get_palette()

    palette = np.array(pallette).reshape(-1,3)
    if isinstance(image, PngImageFile):
        h, w = image.height, image.width
        # Squash height and width dimensions (makes slicing easier)
        image = np.array(image).reshape(-1)
```

Get the color palette from the image.

Convert the palettized image stored as an array to a vector (helps with our next steps).

Get the image's height and width.

```
elif isinstance(image, np.ndarray):
    h, w = image.shape[0], image.shape[1]
    image = image.reshape(-1)
```

Get the image as a vector if the image is provided as an array instead of a Pillow image.

```
rgb_image = np.zeros(shape=(image.shape[0],3))
rgb_image[(image != 0),:] = pallette[image[(image != 0)], :]
rgb_image = rgb_image.reshape(h, w, 3)

return rgb_image
```

Restore the original shape.

We first define a vector of zeros that has the same length as our image. Then, for all the indices found in the image, we gather corresponding colors from the palette and assign them to the same position in the rgb_image.

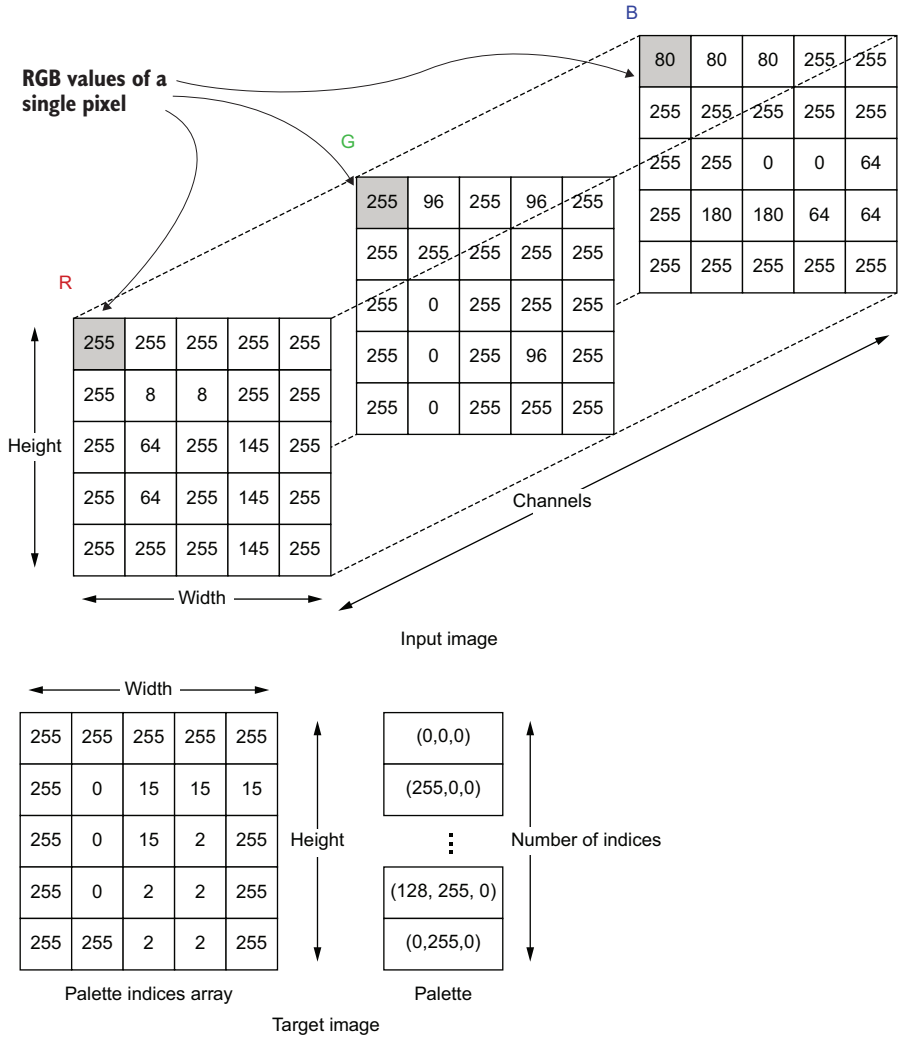

Figure 8.5 The numerical representation of input images and target images in the PASCAL VOC 2012 data set

Here, we first obtain the palette of the image using the `get_palette()` function. This will be present as a one-dimensional array (of length number of classes × number of channels). Next, we need to reshape the array to a (number of classes, number of channels)–sized array. In our case, this will be converted to a (22,3)–sized array. As we define the first dimension of the reshape as –1, it will be automatically inferred from the original size of the data and the other dimensions of the reshape operation. Finally, we define an array of zeros, which will ultimately store the actual colors the indices found in the image. To do that, we index the `rgb_image` vector using the `image` (which contains indices) and assign matching colors from the palette to those indices.

With the data we have looked at thus far, let's define a TensorFlow data pipeline that can transform and convert the data to a format acceptable by the model.

EXERCISE 1

You have been provided with an `rgb_image` in RGB format, where each pixel belongs to one of n distinctive colors and has been given a palette called palette, which is a [n,3]–sized array. How would you convert the `rgb_image` to a palettized image?

> **HINT** You can create the naïve solution by using three `for` loops: two loops to get a single pixel of `rgb_image` and then a final loop to traverse each color in the palette.

8.2 Getting serious: Defining a TensorFlow data pipeline

So far, we have discussed the data that will help us build a navigation algorithm for the RC toy. Before building a model, an important task to complete is having a scalable data ingestion method from disk to the model. Doing this upfront will save us a lot of time when we're ready to scale or productionize. You think the best way is to implement a `tf.data` pipeline to retrieve images from the disk, preprocess them, transform them, and have them ready for the model to grab them. This pipeline should read images in, reshape them to a fixed size (in the case of variable-sized images), augment them (during the training stage), batch them, and repeat this process for a desired number of epochs. Finally, we will define three pipelines: a training data pipeline, a validation data pipeline, and a testing data pipeline.

Our goal at the end of the data exploration stage should be to build a reliable data pipeline from the disk to the model. This is what we will be looking at here. At a high level, we will build a TensorFlow data pipeline that will perform the following tasks:

- Get the filenames belonging to a certain subset (e.g., training, validation, or testing).
- Read the specified images from the disk.
- Preprocess the images (this involves normalizing/resizing/cropping images).
- Perform augmentation on the images to increase the volume of data.
- Batch the data in small batches.
- Optimize data retrieval using several built-in optimization techniques.

As the first step, we will write a function that returns a generator that will generate filenames of the data that we want to be fetched. We will also provide the ability to specify which subset the user wants to be fetched (e.g., training, validation, or testing). Returning data through a generator will make writing a `tf.data` pipeline easier (see the following listing).

Listing 8.3 Retrieving the filenames for a given subset of data

```
def get_subset_filenames(orig_dir, seg_dir, subset_dir, subset):
    """ Get the filenames for a given subset (train/valid/test)"""

    if subset.startswith('train'):
        ser = pd.read_csv(
            os.path.join(subset_dir, "train.txt"),
            index_col=None, header=None, squeeze=True
        ).tolist()
        elif subset.startswith('val') or subset.startswith('test'):

        random.seed(random_seed)

        ser = pd.read_csv(
            os.path.join(subset_dir, "val.txt"),
            index_col=None, header=None, squeeze=True
        ).tolist()

        random.shuffle(ser)

        if subset.startswith('val'):
            ser = ser[:len(ser)//2]
        else:
            ser = ser[len(ser)//2:]
    else:
        raise NotImplementedError("Subset={} is not
     recognized".format(subset))

    orig_filenames = [os.path.join(orig_dir,f+'.jpg') for f in ser]
    seg_filenames = [os.path.join(seg_dir, f+'.png') for f in ser]

    for o, s in zip(orig_filenames, seg_filenames):
        yield o, s
```

For validation/test subsets, perform a one-time shuffle to make sure we get a good mix with a fixed seed.

Read the CSV file that contains the training instance filenames.

Read the CSV file that contains validation/test filenames.

Shuffle the data after fixing the seed.

Get the first half as the validation set.

Get the second half as the test set.

Form absolute paths to the input image files we captured (depending on the subset argument).

Form absolute paths to the segmented image files.

Return the filename pairs (input and annotations) as a generator.

You can see that we're passing a few arguments when reading the CSV files. These arguments characterize the file we're reading. These files are extremely simple and contain just a single image filename on a single line. `index_col=None` means that the file does not have an index column, `header=None` means there is no header in the file, and `squeeze=True` means that the output will be presented as a pandas `Series`, not a

pandas Dataframe. With that, we can define a TensorFlow data set (tf.data.Dataset) as follows:

```
filename_ds = tf.data.Dataset.from_generator(
        subset_filename_gen_func, output_types=(tf.string, tf.string)
    )
```

TensorFlow has several different functions for generating data sets using different sources. As we have defined the function get_subset_filenames() to return a generator, we will use the tf.data.Dataset.from_generator() function. Note that we need to provide the format as well as the datatypes of the returned data, by the generator, using the output_types argument. The function subset_filename_gen_func returns two strings; therefore, we define output types as a tuple of two tf.string elements.

One other important aspect is the different txt files we read from depending on the subset. There are three different files in the relative path: the data\VOCtrainval_11-May-2012\VOCdevkit\VOC2012\ImageSets\Segmentation folder; train.txt, val.txt, and trainval.txt. Here, train.txt contains the filenames of the training images, whereas val.txt contains the filenames of the validation/testing images. We will use these files to create different pipelines that produce different data.

Where does tf.data come from?

TensorFlow's tf.data pipeline can consume data from various sources. Here are some of the commonly used methods to retrieve data:

tf.data.Dataset.from_generator(gen_fn)—You have already seen this function in action. If you have a generator (i.e., gen_fn) that produces data, you want it to be processed through a tf.data pipeline. This is the easiest method to use.

tf.data.Dataset.from_tensor_slices(t)—This is a very useful function if you have data already loaded as a big matrix. t can be an N-dimensional matrix, and this function will extract element by element on the first dimension. For example, assume that you have loaded a tensor t of size 3 × 4 to memory:

```
t = [ [1,2,3,4],
      [2,3,4,5],
      [6,7,8,9] ]
```

Then you can easily set up a tf.data pipeline as follows. tf.data.Dataset.from_tensor_slices(t) will return [1,2,3,4], then [2,3,4,5], and finally [6,7,8,9] when you iterate this data pipeline. In other words, you are seeing one row (i.e., a slice from the batch dimension, hence the name from_tensor_slices) at a time. You can now incorporate functions like tf.data.Dataset.batch() to get a batch of rows.

Now it's time to read in the images found in the file paths we obtained in the previous step. TensorFlow has support to easily load an image, where the path to a filename is

`img_filename`, using the functions `tf.io.read_file` and `tf.image.decode_image`. Here, `img_filename` is a `tf.string` (i.e., a string in TensorFlow):

```
tf.image.decode_jpeg(tf.io.read_file(image_filename))
```

We will use this pattern to load input images. However, we need to implement a custom image load function to load the target image. If you use the previous approach, it will automatically convert the image to an array with pixel values (instead of palette indices). But if we don't perform that conversion, we will have a target array that is in the exact format we need because the palette indices that are in the target image are the actual class labels for each corresponding pixel in the input image. We will use `PIL.Image` within our TensorFlow data pipeline to load the image as a palettized image and avoid converting it to RGB:

```
from PIL import Image

def load_image_func(image):
    """ Load the image given a filename """

    img =  np.array(Image.open(image))
    return img
```

However, you can't yet use custom functions as part of the `tf.data` pipeline. They need to be streamlined with the data-flow graph of the data pipeline by wrapping it as a TensorFlow operation. This can be easily achieved by using the `tf.numpy_function` operation, which allows you to wrap a custom function that returns a NumPy array as a TensorFlow operation. If we have the target image's file path represented by `y`, you can use the following code to load the image into TensorFlow with a custom image-loading function:

```
tf.numpy_function(load_image_func, inp=[y], Tout=[tf.uint8])
```

> **The dark side of tf.numpy_function**
>
> NumPy has larger coverage for various scientific computations than TensorFlow, so you might think that `tf.numpy_funtion` makes things very convenient. This is not quite true, as you can infest your TensorFlow code with terrible performance degradations. When TensorFlow executes NumPy code, it can create very inefficient data flow graphs and introduce overheads. Therefore, always try to stick to TensorFlow operations and use custom NumPy code only if you have to. In our case, since there is no alternative way for us to load a palletized image without mapping palletized values to actual RGB, we used a custom function.

Notice how we're passing both the input (i.e., `inp=[y]`) and its data type (i.e., `Tout=[tf.uint8]`) to this function. They both need to be in the form of a Python list. Finally, let's collate everything we discussed in one place:

```
def load_image_func(image):
    """ Load the image given a filename """

    img =  np.array(Image.open(image))
    return img

# Load the images from the filenames returned by the above step
    image_ds = filename_ds.map(lambda x,y: (
        tf.image.decode_jpeg(tf.io.read_file(x)),
        tf.numpy_function(load_image_func, [y], [tf.uint8])
    ))
```

The `tf.data.Dataset.map()` function will be used quite heavily throughout this discussion. You can find a lengthy explanation of the `map()` function in the sidebar.

A Refresher: tf.data.Dataset.map() function

This `tf.data` pipeline will make extensive use of the `tf.data.Dataset.map()` function. Therefore, it is extremely helpful for us to remind ourselves what this function accomplishes.

The `td.data.Dataset.map()` function applies a given function or functions across all the records in a data set. In other words, it transforms the data points in the data set using a specified transformation. For example, assume the `tf.data.Dataset`

```
dataset = tf.data.Dataset.from_tensor_slices([1, 2, 3, 4])
```

to get the square of each element, you can use the map function as

```
dataset = dataset.map(lambda x: x**2)
```

If you have multiple elements in a single record, leveraging the flexibility of `map()`, you can transform them individually:

```
dataset = tf.data.Dataset.from_tensor_slices([[1,3], [2,4], [3,5], [4,6]])
dataset = dataset.map(lambda x, y: (x**2, y+x))
which will return,
[[1, 4], [4, 6], [9, 8], [16, 10]]
```

As a normalization step we will bring the pixel values to [0,1] range by using

```
image_ds = image_ds.map(lambda x, y: (tf.cast(x, 'float32')/255.0, y))
```

Note that we are keeping our target image (y) as it is. Before I continue with any more steps in our pipeline, I want to direct your attention to an important matter. This is a caveat that is quite common, and it is thus worthwhile to be aware of it. After the step we just completed, you might feel like, if you want, you can batch the data and feed it to the model. For example

```
image_ds = image_ds.batch(10)
```

If you do that for this data set, you will get an error like the following:

```
InvalidArgumentError: Cannot batch tensors with different shapes in
➥ component 0. First element had shape [375,500,3] and element 1 had
➥ shape [333,500,3]. [Op:IteratorGetNext]
```

This is because you ignored a crucial characteristic and a sanity check of the data set. Unless you're using a curated data set, you are unlikely to find images with the same dimensions. If you look at images in the data set, you will notice that they are not of the same size; they have different heights and widths. In TensorFlow, unless you use a special data structure like `tf.RaggedTensor`, you cannot batch unequally sized images together. That is exactly what TensorFlow is complaining about in the error.

To alleviate the problem, we need to bring all the images to a standard size (see listing 8.4). To do that, we will define the following function. It will either

- Resize the image to a larger size (`resize_to_before_crop`) and then crop the image to the desired size (`input_size`) or
- Resize the image to the desired size (`input_size`)

Listing 8.4 Bringing images to a fixed size using random cropping or resizing

```
def randomly_crop_or_resize(x,y):
    """ Randomly crops or resizes the images """

    def rand_crop(x, y):
        """ Randomly crop images after enlarging them """
        x = tf.image.resize(x, resize_to_before_crop, method='bilinear')
        y = tf.cast(
                tf.image.resize(
                    tf.transpose(y,[1,2,0]),
                    resize_to_before_crop, method='nearest'
                ),
                'float32'
            )

        offset_h = tf.random.uniform(
            [], 0, x.shape[0]-input_size[0], dtype='int32'
        )
        offset_w = tf.random.uniform(
            [], 0, x.shape[1]-input_size[1], dtype='int32'
        )
        x = tf.image.crop_to_bounding_box(
            image=x,
            offset_height=offset_h, offset_width=offset_w,
            target_height=input_size[0], target_width=input_size[1]
        )
        y = tf.image.crop_to_bounding_box(
            image=y,
            offset_height=offset_h, offset_width=offset_w,
            target_height=input_size[0], target_width=input_size[1]
        )
```

Define a function to randomly crop images after resizing.

Resize the input image using bilinear interpolation to a larger size.

Resize the target image using the nearest interpolation to a larger size.

To resize, we first swap the axis of y as it has the shape [1, height, width]. We convert this back to [height, width, 1] (i.e., a single channel image) using the tf.transpose() function.

Define a random variable to offset images on height during cropping.

Define a random variable to offset images on width during cropping.

Crop the input image and the target image using the same cropping parameters.

```
    return x, y

def resize(x, y):
    """ Resize images to a desired size """
    x = tf.image.resize(x, input_size, method='bilinear')
    y = tf.cast(
            tf.image.resize(
              tf.transpose(y,[1,2,0]),
                 input_size, method='nearest'
            ),
            'float32'
        )

    return x, y

rand = tf.random.uniform([], 0.0,1.0)

if augmentation and \
    (input_size[0] < resize_to_before_crop[0] or \
     input_size[1] < resize_to_before_crop[1]):
    x, y = tf.cond(
            rand < 0.5,
            lambda: rand_crop(x, y),
             lambda: resize(x, y)
            )
    else:
        x, y = resize(x, y)

    return x, y
```

Resize both the input image and the target image to a desired size (no cropping).

Define a random variable (used to perform augmentations).

If augmentation is enabled and the resized image is larger than the input size we requested, perform augmentation.

During augmentation, the rand_crop or resize function is executed randomly.

If augmentation is disabled, only resize images.

Here, we define a function called `randomly_crop_or_resize`, which has two nested functions, `rand_crop` and `resize`. The `rand_crop` first resizes the image to the size specified in `resize_to_before_crop` and creates a random crop. It is imperative to check that you applied the exact same crop to both the input and the target. For example, same-crop parameters should be used to crop both the input and the target. In order to crop images, we use

```
x = tf.image.crop_to_bounding_box(
    image=x,
    offset_height=offset_h, offset_width=offset_w,
    target_height=input_size[0], target_width=input_size[1]
)
y = tf.image.crop_to_bounding_box(
    image=y,
    offset_height=offset_h, offset_width=offset_w,
    target_height=input_size[0], target_width=input_size[1]
)
```

The arguments are self-explanatory: `image` takes an image to be cropped, `offset_height` and `offset_width` decide the starting point for the crop, and `target_height` and `target_width` specify the final size after the crop. The resize function will simply resize the input and the target to a specified size using the `tf.image.resize` operation.

When resizing, we use *bilinear interpolation* for the input images and *nearest interpolation* for targets. Bilinear interpolation resizes the images by computing the resulting pixels, as an average of neighboring pixels, whereas nearest interpolation computes the output pixel as the nearest most common pixel from the neighbors. Bilinear interpolation leads to a smoother result after resizing. However, you must use nearest interpolation for the target image, as bilinear interpolation will lead to fractional outputs, corrupting the integer-based annotations. The interpolation techniques described are visualized in figure 8.6.

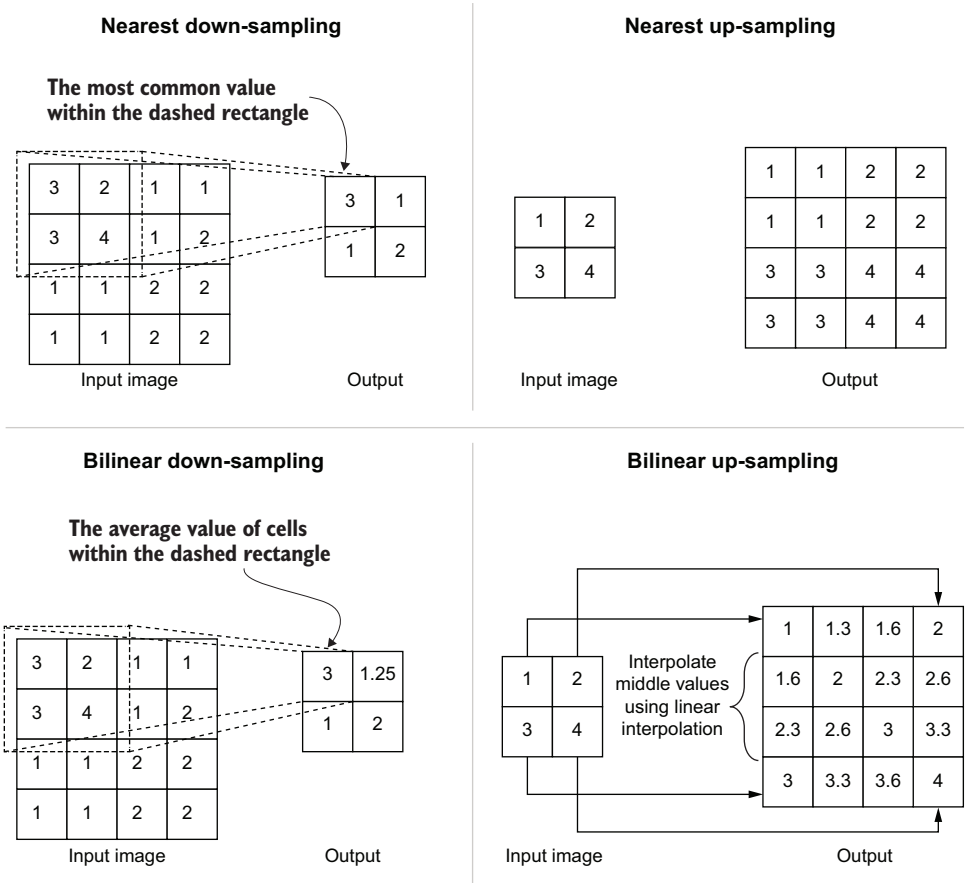

Figure 8.6 Nearest interpolation and bilinear interpolation for both up-sampling and down-sampling tasks

Next, we will introduce an additional step to the way we're going to use these two nested functions. If augmentation is enabled, we want the cropping or resizing to take place randomly within the pipeline. We will define a random variable (drawn from a uniform distribution between 0 and 1) and perform crop or resize depending on the

value of the random variable at a given time. This conditioning can be achieved using the `tf.cond` function, which takes three arguments and returns output according to these arguments:

- `Condition`—This is a computation that results in a Boolean value (i.e., is the random variable `rand` greater than 0.5).
- `true_fn`—If the condition is true, then this function will be executed (i.e., perform `rand_crop` on both `x` and `y`)
- `false_fn`—If the condition is false, then this function will be executed (i.e., perform a resize on both `x` and `y`)

If augmentation is disabled (i.e., by setting the augmentation variable to `False`), only resizing is performed. With the details fleshed out, we can use the `randomly_crop_or_resize` function in our data pipeline as follows:

```
    image_ds = image_ds.map(lambda x,y: randomly_crop_or_resize(x,y))
```

At this point, we have a globally fixed-sized image coming out of our pipeline. The next thing we address is very important. Factors such as variable size of images and custom NumPy functions used to load images make it impossible for TensorFlow to infer the shape of its final tensor (though it's a fixed-sized tensor) after a few steps. If you check the shapes of the tensors produced at this point, you will probably perceive them as

```
(None, None, None)
```

This means that TensorFlow was unable to infer the shape of the tensors. To avoid any ambiguities or problems moving forward, we will set the shape of the output we have in the pipeline. For a tensor `t`, if the shape is ambiguous but you know the shape, you can set the shape manually using

```
t.set_shape([<shape of the tensor>])
```

In our data pipeline, we can set the shape as

```
def fix_shape(x, y, size):
    """ Set the shape of the input/target tensors """

    x.set_shape((size[0], size[1], 3))
    y.set_shape((size[0], size[1], 1))

    return x, y

image_ds = image_ds.map(lambda x,y: fix_shape(x,y, target_size=input_size))
```

We know that the outputs following the resize or crop are going to be

- *Input image*—An RGB image with `input_size` height and width
- *Target image*—A single-channel image with `input_size` height and width

We will set the shape accordingly using the `tf.data.Dataset.map()` function. We cannot underestimate the power of data augmentation, so we will introduce several data augmentation steps to our data pipeline (see the next listing).

Listing 8.5 Functions used for random augmentation of images

```
def randomly_flip_horizontal(x, y):
    """ Randomly flip images horizontally. """

    rand = tf.random.uniform([], 0.0,1.0)

    def flip(x, y):
        return tf.image.flip_left_right(x), tf.image.flip_left_right(y)

    x, y = tf.cond(rand < 0.5, lambda: flip(x, y), lambda: (x, y))

    return x, y

if augmentation:
    image_ds = image_ds.map(lambda x, y: randomly_flip_horizontal(x,y))

    image_ds = image_ds.map(lambda x, y: (tf.image.random_hue(x, 0.1), y))

    image_ds = image_ds.map(lambda x, y: (tf.image.random_brightness(x, 0.1), y))

    image_ds = image_ds.map(lambda x, y: (tf.image.random_contrast(x, 0.8, 1.2), y))
```

Define a random variable.

Define a function to flip images deterministically.

Randomly flip images in the data set.

Using the same pattern as before, we use tf.cond to randomly perform horizontal flipping.

Randomly adjust the contrast of the input image (target stays the same).

Randomly adjust the brightness of the input image (target stays the same).

Randomly adjust the hue (i.e., color) of the input image (target stays the same).

In listing 8.5, we perform the following translations:

- Randomly flipping images horizontally
- Randomly changing the hue of the images (up to 10%)
- Randomly changing the brightness of the images (up to 10%)
- Randomly changing the contrast of the images (up to 20%)

By using the `tf.data.Dataset.map()` function, we can easily perform the specified random augmentation steps, should the user enable augmentation in the pipeline (i.e., by setting the augmentation variable to `True`). Note that we're performing some augmentations (e.g., random hue, brightness, and contrast adjustments) on the input image only. We will also give the user the option to have different-sized inputs and targets (i.e., outputs). This is achieved by resizing the output to a desired size, defined by the `output_size` argument. The model we use for this task has different-sized input and output dimensions:

```
if output_size:
    image_ds = image_ds.map(
                    lambda x, y: (
```

```
                    x,
                    tf.image.resize(y, output_size, method='nearest')
                )
    )
```

Again, here we use the nearest interpolation to resize the target. Next, we will shuffle the data (if the user set the shuffle argument to `True`):

```
if shuffle:
    image_ds = image_ds.shuffle(buffer_size=batch_size*5)
```

The shuffle function takes an important argument called `buffer_size`, which determines how many samples are loaded to memory in order to select a sample randomly. The higher the `buffer_size`, the more randomness you are introducing. On the other hand, a higher `buffer_size` implies higher memory consumption. It's now time to batch the data, so instead of a single data point, we get a batch of data when we iterate:

```
image_ds = image_ds.batch(batch_size).repeat(epochs)
```

This is done using the `tf.data.Dataset.batch()` function and passing the desired batch size as the argument. When using the `tf.data` pipeline, if you are running it for multiple epochs, you also need to use the `tf.data.Dataset.repeat()` function to repeat the pipeline for a given number of epochs.

Why do we need tf.data.Dataset.repeat()?

`tf.data.Dataset` is a generator. A unique characteristic of a generator is that you only can iterate it once. After the generator reaches the end of the sequence it's iterating, it will exit by throwing an exception. Therefore, if you need to iterate through a generator multiple times, you need to redefine the generator as many times as needed. By adding `tf.data.Dataset.repeat(epochs)`, the generate is redefined as many times as we would like (`epochs` times in this example).

One more step is needed before our `tf.data` pipeline is done and dusted. If you look at the shape of the target (`y`) output, you will see that it has a channel dimension of 1. However, for the loss function we will be using, we need to get rid of that dimension:

```
image_ds = image_ds.map(lambda x, y: (x, tf.squeeze(y)))
```

For this, we will use the `tf.squeeze()` operation, which removes any dimensions that are of size 1 and returns a tensor. For example, if you squeeze a tensor of size [1,3,2,1,5], you will get a [3,2,5] sized tensor. The final code is provided in listing 8.6. You might notice two steps that are highlighted. These are two popular optimization steps available: caching and prefetching.

Listing 8.6 The final tf.data pipeline

```
def get_subset_tf_dataset(
    subset_filename_gen_func, batch_size, epochs,
    input_size=(256, 256), output_size=None, resize_to_before_crop=None,
    augmentation=False, shuffle=False
):

    if augmentation and not resize_to_before_crop:
        raise RuntimeError(
            "You must define resize_to_before_crop when augmentation is enabled."
        )

    filename_ds = tf.data.Dataset.from_generator(
        subset_filename_gen_func, output_types=(tf.string, tf.string)
    )

    image_ds = filename_ds.map(lambda x,y: (
        tf.image.decode_jpeg(tf.io.read_file(x)),
        tf.numpy_function(load_image_func, [y], [tf.uint8])
    )).cache()

    image_ds = image_ds.map(lambda x, y: (tf.cast(x, 'float32')/255.0, y))


    def randomly_crop_or_resize(x,y):
        """ Randomly crops or resizes the images """
        …

        def rand_crop(x, y):
            """ Randomly crop images after enlarging them """
            …

        def resize(x, y):
            """ Resize images to a desired size """
            …

    image_ds = image_ds.map(lambda x,y: randomly_crop_or_resize(x,y))
    image_ds = image_ds.map(lambda x,y: fix_shape(x,y, target_size=input_size))

    if augmentation:
        image_ds = image_ds.map(lambda x, y: randomly_flip_horizontal(x,y))
        image_ds = image_ds.map(lambda x, y: (tf.image.random_hue(x, 0.1), y))
        image_ds = image_ds.map(lambda x, y: (tf.image.random_brightness(x, 0.1), y))
        image_ds = image_ds.map(
            lambda x, y: (tf.image.random_contrast(x, 0.8, 1.2), y)
        )

    if output_size:
        image_ds = image_ds.map(
            lambda x, y: (x, tf.image.resize(y, output_size,  method='nearest'))
        )

    if shuffle:
        image_ds = image_ds.shuffle(buffer_size=batch_size*5)
```

Return a list of filenames depending on the subset of data requested.

If augmentation is enabled, resize_to_before_crop needs to be defined.

Load the images into memory. cache() is an optimization step and will be discussed in the text.

Normalize the input images.

The function that randomly crops or resizes images

Perform random crop or resize on the images.

Set the shape of the resulting images.

Randomly perform various augmentations on the data.

Resize the output image if needed.

Shuffle the data using a buffer.

```
image_ds = image_ds.batch(batch_size).repeat(epochs)

image_ds = image_ds.prefetch(tf.data.experimental.AUTOTUNE)

image_ds = image_ds.map(lambda x, y: (x, tf.squeeze(y)))

return image_ds
```

Batch the data and repeat the process for a desired number of epochs.

This is an optimization step discussed in detail in the text.

Remove the unnecessary dimension from target images.

Get the final tf.data pipeline.

It wasn't an easy journey, but it was a rewarding one. We have learned some important skills in defining the data pipeline:

- Defining a generator that returns the filenames of the data to be fetched
- Loading images within a `tf.data` pipeline
- Manipulating images (resizing, cropping, brightness adjustment, etc.)
- Batching and repeating data
- Defining multiple pipelines for different data sets with different requirements

Next, we will look at some optimization techniques to turn our mediocre data pipeline into an impressive data highway.

8.2.1 Optimizing tf.data pipelines

TensorFlow is a framework meant for consuming large data sets, where consuming data in an efficient manner is a key priority. One thing still missing from our conversation is what kind of optimization steps are available for `tf.data` pipelines, so let us nudge this discussion in that direction. Two steps were set in bold in listing 8.6: caching and prefetching. If you are interested in other optimization techniques, you can read more at https://www.tensorflow.org/guide/data_performance.

Caching will store the data in memory as it flows through the pipeline. This means that, when cached, that step (e.g., loading the data from the disk) happens only in the first epoch. The subsequent epochs will read from the cached data that's held in memory. Here, you can see that we're caching the images after we load them to memory. This way, TensorFlow loads the images in the first epoch only:

```
image_ds = filename_ds.map(lambda x,y: (
        tf.image.decode_jpeg(tf.io.read_file(x)),
        tf.numpy_function(load_image_func, [y], [tf.uint8])
)).cache()
```

Prefetching is another powerful weapon you have at your disposal, and it allows you to leverage the multiprocessing power of your device:

```
image_ds = image_ds.prefetch(tf.data.experimental.AUTOTUNE)
```

The argument provided to the function decides how much data is prefetched. By setting it to `AUTOTUNE`, TensorFlow will decide the best amount of data to be fetched depending on the resources available. Assume a simple data pipeline that loads images

from the disk and trains a model. Then, the data read and model training will happen in interleaved steps. This leads to significant idling time, as the model idles while the data is loading, and vice versa.

However, thanks to prefetching, this doesn't need to be the case. Prefetching employs background threads and an internal buffer to load the data in advance while the model is training. When the next iteration comes, the model can seamlessly continue the training as data is already fetched into the memory. The differences between sequential execution and prefetching are shown in figure 8.7.

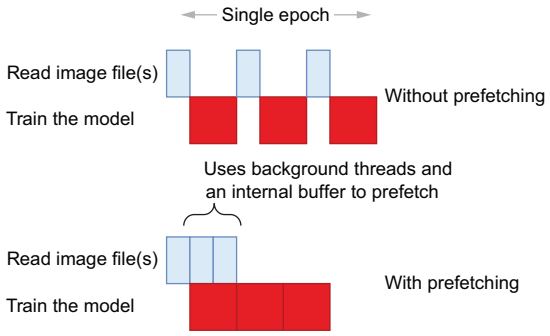

Figure 8.7 Sequential execution versus pre-fetching-based execution in model training

Next, we will look at the finished `tf.data` pipeline for the image segmentation problem.

8.2.2 The final tf.data pipeline

Finally, you can define the data pipeline(s) using the functions we have defined so far. Here, we define three different data pipelines for three different purposes: training, validation, and testing (see the following listing).

Listing 8.7 Creating the train/validation/test data pipelines instances

```
orig_dir = os.path.join(
    'data', 'VOCtrainval_11-May-2012', 'VOCdevkit', 'VOC2012', 'JPEGImages'
)
seg_dir = os.path.join(
    'data', 'VOCtrainval_11-May-2012', 'VOCdevkit', 'VOC2012',
     'SegmentationClass'
)
subset_dir = os.path.join(
    'data', 'VOCtrainval_11-May-2012', 'VOCdevkit', 'VOC2012', 'ImageSets',
    'Segmentation'
)

partial_subset_fn = partial(
    get_subset_filenames, orig_dir=orig_dir, seg_dir=seg_dir,
     subset_dir=subset_dir
)
```

Directory where the input images are

Directory where the annotated images (targets) are

Directory where the text files containing train/validation/test filenames are

Define a reusable partial function from get_subset_filenames.

```
train_subset_fn = partial(partial_subset_fn, subset='train')
val_subset_fn = partial(partial_subset_fn, subset='val')
test_subset_fn = partial(partial_subset_fn, subset='test')

input_size = (384, 384)

tr_image_ds = get_subset_tf_dataset(
    train_subset_fn, batch_size, epochs,
    input_size=input_size, resize_to_before_crop=(444,444),
    augmentation=True, shuffle=True
)
val_image_ds = get_subset_tf_dataset(
    val_subset_fn, batch_size, epochs,
    input_size=input_size,
    shuffle=False
)
test_image_ds = get_subset_tf_dataset(
    test_subset_fn, batch_size, 1,
    input_size=input_size,
    shuffle=False
)
```

Define input image size.

Define three generators for train/validation/test data.

Define a train data pipeline that uses data augmentation and shuffling.

Define a validation data pipeline that doesn't use data augmentation or shuffling.

Define a test data pipeline.

First, we define several important paths:

- `orig_dir`—Directory containing input images
- `seg_dir`—Directory containing the target images
- `subset_dir`—Directory containing text files (train.txt, val.txt) that enlist training and validation instances, respectively

Then we will define a partial function from the `get_subset_filenames()` function we defined earlier so that we can get a generator just by setting the subset argument of the function. Using this technique, we will define three generators: `train_subset_fn`, `val_subset_fn`, and `test_subset_fn`. Finally, we will define three `tf.data.Datasets` using the `get_subset_tf_dataset()` function. Our pipelines will have the following characteristics:

- *Training pipeline*—Performs data augmentation and data shuffling on every epoch
- *Validation pipeline and test pipeline*—No augmentation or shuffling

The model we will define expects a 384 × 384–sized input and an output. In the training data pipeline, we will resize images to 444 × 444 and then randomly crop a 384 × 384–sized image. Following this, we will look at the core part of the solution: defining the image segmentation model.

EXERCISE 2

You have been given a small set of data that contains two tensors: tensor a contains 100 64 × 64 × 3–sized images (i.e., 100 × 64 × 64 × 3 shaped), and tensor b contains 100 32 × 32 × 1–sized segmentation masks (i.e., 100 × 32 × 32 × 1 shaped). You have been asked to define a `tf.data.Dataset` using the functions discussed that will

- Resize the segmentation masks to match the input image size (using nearest interpolation)
- Normalize the input images using the transformation (x – 128)/255 where a single image is x
- Batch the data to batches of 32 and repeat for five epochs
- Prefetch the data with an auto-tuning feature

8.3 DeepLabv3: Using pretrained networks to segment images

It's now time to create the brains of the pipeline: the deep learning model. Based on feedback from a colleague at a self-driving car company working on similar problems, you will implement a DeepLab v3 model. This is a model built on the back of a pretrained ResNet 50 model (trained on image classification) but with the last several layers changed to perform *atrous convolution* instead of standard convolution. It uses a pyramidal aggregation module that uses atrous convolution at different scales to generate image features at different scales to produce the final output. Finally, it uses a bilinear interpolation layer to resize the final output to a desired size. You are confident that DeepLab v3 can deliver good initial results.

Deep neural network–based segmentation models can be broadly categorized into two types:

- Encoder decoder models (e.g., U-Net model)
- Fully convolutional network (FCN) followed by a pyramidal aggregation module (e.g., DeepLab v3 model)

A well-known example of the encoder-decoder model is the U-Net model. In other words, U-Net has an encoder that gradually creates smaller, coarser representations of the input. This is followed by a decoder that takes the representations the encoder built and gradually up-samples (i.e., increases the size of) the output until it reaches the size of the input image. The up-sampling is achieved through an operation known as *transpose convolution*. Finally, you train the whole structure end to end, where an input is the input image and the target is the segmentation mask for the corresponding image. We will not discuss this type of model in this chapter. However, I have included a detailed walkthrough in appendix B (along with an implementation of the model).

The other type of segmentation models introduces a special model that replaces the decoder. We call this module a *pyramidal aggregation module*. Its purpose is to garner spatial information at different scales (e.g., different-sized outputs from various interim convolution layers) that provides fine-grained contextual information about the objects present in the image. DeepLab v3 is a prime example of this approach. We will put the DeepLab v3 model under the microscope and use it to excel at the segmentation task.

Researchers and engineers gravitate toward methods that use pyramidal aggregation modules more. There could be many reasons for this. One lucrative reason is that there are less parameters in networks that use pyramidal aggregation than an encoder-decoder based counterpart. Another reason may be that, typically, introducing a novel module offers more flexibility (compared to an encoder-decoder) to engineer efficient and accurate feature extraction methods at multiple scales.

How important is the pyramidal aggregation module? To know that, we have to first understand what the fully convolutional part of the network looks like. Figure 8.8 illustrates the generic structure of such a segmentation model.

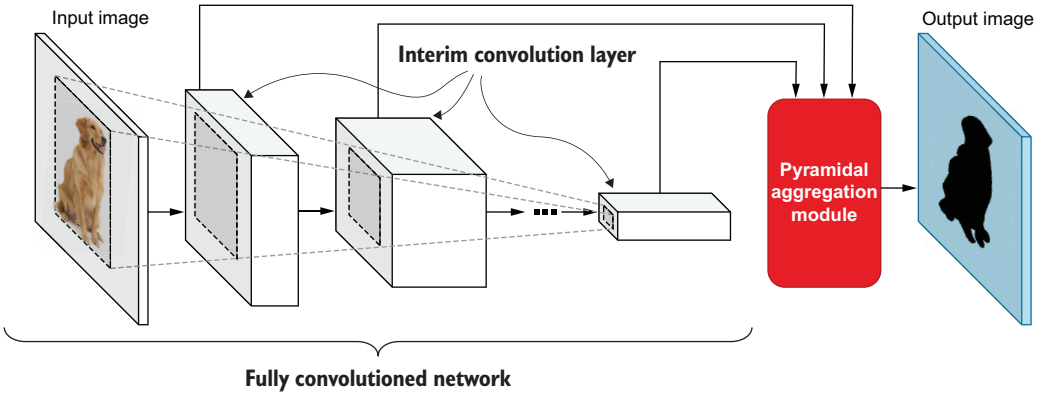

Figure 8.8 General structure and organization of a fully convolutional network that uses a pyramidal aggregation module

The best way to understand the importance of the pyramidal aggregation module is to see what happens if we don't have it. If that is the case, then the last convolutional layer will have the enormous and unrealistic responsibility of building the final segmentation mask (which is typically 16–32x times larger than the layer output). It is no surprise that there is a massive representational bottleneck between the final convolution layer and the final segmentation mask, leading to poor performance. The pyramidal structure typically enforced in CNNs results in a very small output width and height in the final layer.

The pyramidal aggregation module bridges this gap. It does so by combining several different interim outputs. This way, the network has ample fine-grained (from earlier layers) and coarser (from deeper layers) details to construct the desired segmentation mask. Fine-grained representations provide spatial/contextual information about the image, whereas the coarser representations provide high-level information about the image (e.g., what objects are present). By fusing both types of these representations, the task of generating the final output becomes more achievable.

Why not a skyscraper instead of a pyramid?

You might be tempted to ponder, if making the outputs smaller as you go causes loss of information, "Why not keep it the same size?" (hence the term *skyscraper*). This is an impractical solution for two main reasons.

First, decreasing the size of the outputs through pooling or striding is an important regularization method that forces the network to learn translation-invariant features (as we discussed in chapter 6). By taking this away, we can hinder the generalizability of the network.

Second, not decreasing the output size will increase the memory footprint of the model significantly. This will, in turn, restrict the depth of the network dramatically, making it more difficult to create deeper networks.

DeepLab v3 is the golden child of a lineage of models that emerged from and was introduced in the paper "Rethinking Atrous Convolution for Semantic Image Segmentation" (https://arxiv.org/pdf/1706.05587.pdf) by several researchers from Google.

Most segmentation models face an adverse side effect caused by a common and beneficial design principle. Vision models incorporate stride/pooling to make network translation invariant. But an ill-favored outcome of that is the compounding reduction of the size of the outputs produced. This typically leads to a final output that is 16–32 times smaller than the input. Being a dense prediction task, image segmentation tasks suffer heavily from this design idea. Therefore, most of the groundbreaking networks that have surfaced have been about solving this. The DeepLab model came into the world for exactly that purpose. Let's now see how DeepLab v3 solves this problem.

DeepLab v3 uses a ResNet-50 (https://arxiv.org/pdf/1512.03385.pdf) pretrained on an ImageNet image classification data set as its backbone for extracting features of an image. It is one of the pioneering residual networks that made waves in the computer vision community a few years ago. DeepLab v3 introduces several architectural changes to the model to alleviate this issue. Furthermore, DeepLab v3 introduces a shiny new component called *atrous spatial pyramid pooling* (ASPP). We will discuss each of these in more detail in the coming sections.

8.3.1 A quick overview of the ResNet-50 model

The ResNet-50 model consists of several convolution blocks, followed by a global average pooling layer and a fully connected final prediction layer with softmax activation. The convolution block is the innovative part of the model. The original model has 16 convolution blocks organized into five groups. A single block consists of three convolution layers (1×1 convolution layer with stride 2, 3×3 convolution layer, and 1×1 convolution layer), batch normalization, and residual connections. We discussed residual connections in depth in chapter 7. Next, we will discuss a core computation used throughout the model known as atrous convolution.

8.3.2 Atrous convolution: Increasing the receptive field of convolution layers with holes

Compared to the standard ResNet-50, a major change that DeepLab v3 boasts is the use of atrous convolutions. Atrous (meaning "holes" in French) convolution, also known as dilated convolution, is a variant of the standard convolution. Atrous convolution works by inserting "holes" in between the convolution parameters. The increase in the receptive field is controlled by a parameter called *dilation rate*. A higher dilation rate means more holes between actual parameters in the convolution. A major benefit of atrous convolution is the ability to increase the size of the receptive field without compromising the parameter efficiency of a convolution layer.

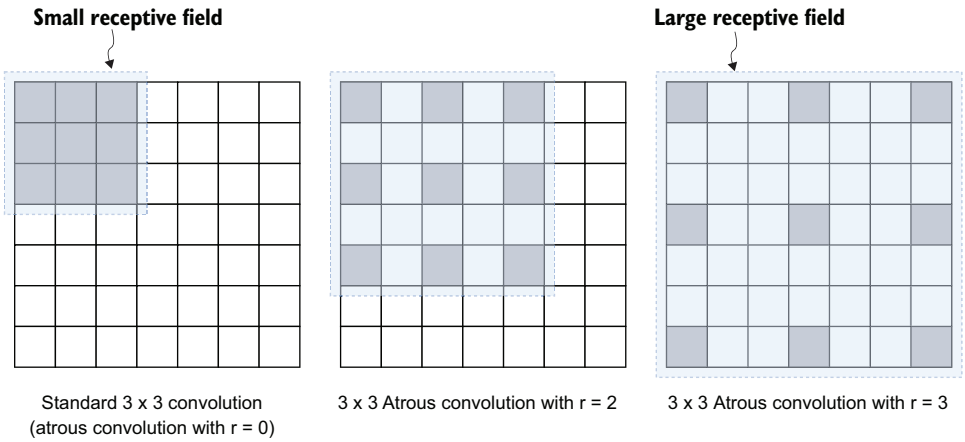

Figure 8.9 Atrous convolution compared to standard convolution. Standard convolution is a special case of atrous convolution, where the rate is 1. As you increase the dilation rate, the receptive field of the layer increases.

Figure 8.9 shows how a large dilation rate leads to a larger receptive field. The number of shaded gray boxes represents the number of parameters, whereas the dashed, lightly shaded box represents the size of the receptive field. As you can see, the number of parameters stays constant, while the receptive field increases. Computationally, it is quite straightforward to extend standard convolution to atrous convolution. All you need to do is insert zeros for the holes in the atrous convolution operation.

> **Wait! How does atrous convolution help segmentation models?**
> As we discussed, the main issue presented by the pyramidal structure of CNNs is that the output gets gradually smaller. The easiest solution, leaving the learned parameters untouched, is to reduce the stride of the layers. Though technically that will increase output size, conceptually there is a problem.

(continued)

To understand it, assume the i^{th} layer of a CNN has a stride of 2 and gets a h × w–sized input. Then the $i+1^{th}$ layer gets a h/2 × w/2–sized input. By removing the stride of the i^{th} layer, it gets a h × w–sized output. However, the kernel of the $i+1^{th}$ layer has been trained to see a smaller output, so by increasing the size of the input, we are disrupting (or reducing) the receptive field of the layer. By introducing atrous convolution, we compensate for that reduction of the receptive field.

Let's now see how the ResNet-50 is repurposed for image segmentation. First, we download it from the `tf.keras.applications` module. The architecture of the ResNet-50 model has the following format. To start, it has a stride 2 convolution layer and a stride 2 pooling layer. After that, it has sequence of convolution blocks and finally an average pooling layer and fully connected output layer. These convolution blocks have a hierarchical organization of convolution layers. Each convolution block consists of several subblocks, which consist of three convolution layers (i.e., a 1 × 1 convolution, a 3 × 3 convolution, and a 1 × 1 convolution) along with batch normalization.

8.3.3 Implementing DeepLab v3 using the Keras functional API

The network starting from the input up to the `conv4` block remains unchanged. Following the notation from the original ResNet paper, these blocks are identified as `conv2`, `conv3`, and `conv4` block groups. Our first task is to create a model containing the input layer up to the `conv4` block of the original ResNet-50 model. After that, we will focus on recreating the final convolution block (i.e., `conv5`) as per the DeepLab v3 paper:

```
# Pretrained model and the input
inp = layers.Input(shape=target_size+(3,))
resnet50 = tf.keras.applications.ResNet50(
    include_top=False, input_tensor=inp,pooling=None
)

for layer in resnet50.layers:
    if layer.name == "conv5_block1_1_conv":
        break
    out = layer.output

resnet50_upto_conv4 = models.Model(resnet50.input, out)
```

As shown here, we find the last layer in the ResNet-50 model just before the `"conv5_block1_1_conv"`, which would be the last layer of the `conv4` block group. With that, we can define a makeshift model that contains layers from the input to the final output of the `conv4` block group. Later, we will focus on augmenting this model by introducing modifications and novel components from the paper. We will redefine the `conv5` block with dilated convolutions. To do this, we need to understand the composition of a ResNet block (figure 8.10). We can assume it has three different levels.

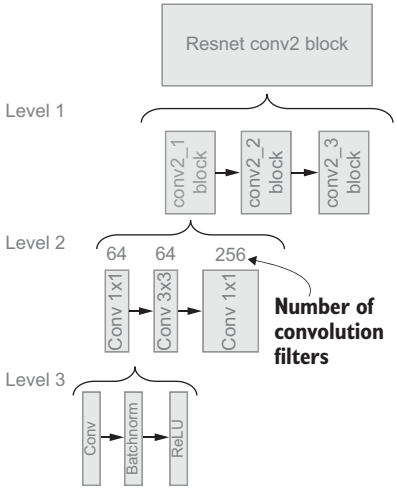

Figure 8.10 Anatomy of a convolution block in ResNet-50. For this example, we show the very first convolution block of ResNet-50. The organization of a convolution block group consists of three different levels.

Let's now implement a function to represent each level while using dilated convolution. In order to convert a standard convolution layer to a dilated convolution, we just have to pass in the desired rate to the `dilation_rate` parameter in the `tf.keras.layers.Conv2D` layer. First, we will implement a function that represents a level 3 block, as shown in the following listing.

Listing 8.8 A level 3 convolution block in ResNet-50

Here, inp takes a 4D input having shape [batch size, height, width, channels].

```
def block_level3(
    inp, filters, kernel_size, rate, block_id, convlayer_id, activation=True
):
    """ A single convolution layer with atrous convolution and batch
     normalization
    inp: 4-D tensor having shape [batch_size, height, width, channels]
    filters: number of output filters
    kernel_size: The size of the convolution kernel
    rate: dilation rate for atrous convolution
    block_id, convlayer_id - IDs to distinguish different convolution blocks
     and layers
    activation: If true ReLU is applied, if False no activation is applied
    """

    conv5_block_conv_out = layers.Conv2D(
        filters, kernel_size, dilation_rate=rate, padding='same',
        name='conv5_block{}_{}_conv'.format(block_id, convlayer_id)
    )(inp)

    conv5_block_bn_out = layers.BatchNormalization(
        name='conv5_block{}_{}_bn'.format(block_id, convlayer_id)
    )(conv5_block_conv_out)
```

Perform 2D convolution on the input with a given number of filters, kernel_size, and dilation rate.

Perform batch normalization on the output of the convolution layer.

```
    if activation:
        conv5_block_relu_out = layers.Activation(
            'relu', name='conv5_block{}_{}_relu'.format(block_id, convlayer_id)
        )(conv5_block_bn_out)

        return conv5_block_relu_out
    else:
        return conv5_block_bn_out
```

Apply ReLU activation if activation is set to True.

Return the output without an activation if activation is set to False.

A level 3 block has a single convolution layer with a desired dilation rate and a batch normalization layer followed by a nonlinear ReLU activation layer. Next, we will write a function for the level 2 block (see the next listing).

Listing 8.9 A level 2 convolution block in ResNet-50

```
def block_level2(inp, rate, block_id):
    """ A level 2 resnet block that consists of three level 3 blocks """

    block_1_out = block_level3(inp, 512, (1,1), rate, block_id, 1)
    block_2_out = block_level3(block_1_out, 512, (3,3), rate, block_id, 2)
    block_3_out = block_level3(
        block_2_out, 2048, (1,1), rate, block_id, 3, activation=False
    )

    return block_3_out
```

A level 2 block consists of three level 3 blocks with a given dilation rate that have convolution layers with the following specifications:

- 1×1 convolution layer having 512 filters and a desired dilation rate
- 3×3 convolution layer having 512 filters and a desired dilation rate
- 1×1 convolution layer having 2048 filters and a desired dilation rate

Apart from using atrous convolution, this is identical to a level 2 block of the original `conv5` block in the ResNet-50 model. With all the building blocks ready, we can implement the fully fledged `conv5` block with atrous convolution (see the next listing).

Listing 8.10 Implementing the final ResNet-50 convolution block group (level 1)

```
def resnet_block(inp, rate):
    """ Redefining a resnet block with atrous convolution """

    block0_out = block_level3(
        inp, 2048, (1,1), 1, block_id=1, convlayer_id=0, activation=False
    )
    block1_out = block_level2(inp, 2, block_id=1)
    block1_add = layers.Add(
        name='conv5_block{}_add'.format(1))([block0_out, block1_out]
    )
    block1_relu = layers.Activation(
        'relu', name='conv5_block{}_relu'.format(1)
    )(block1_add)
```

Create a level 3 block (block0) to create residual connections for the first block.

Define the first level 2 block, which has a dilation rate of 2 (block1).

Create a residual connection from block0 to block1.

Apply ReLU activation to the result.

The second level 2 block with a dilation rate of 2 (block2)

```
block2_out = block_level2 (block1_relu, 2, block_id=2) # no relu
block2_add = layers.Add(
    name='conv5_block{}_add'.format(2)
)([block1_add, block2_out])
block2_relu = layers.Activation(
    'relu', name='conv5_block{}_relu'.format(2)
)(block2_add)

block3_out = block_level2 (block2_relu, 2, block_id=3)
block3_add = layers.Add(
    name='conv5_block{}_add'.format(3)
)([block2_add, block3_out])
block3_relu = layers.Activation(
    'relu', name='conv5_block{}_relu'.format(3)
)(block3_add)

 return block3_relu
```

Create a residual connection from block1 to block2.

Apply ReLU activation.

Apply a similar procedure to block1 and block2 to create block3.

There's no black magic here. The function `resnet_block` lays the outputs of the functions we already discussed to assemble the final convolution block. Particularly, it has three level 2 blocks with residual connections going from the previous block to the next. Finally, we can get the final output of the `conv5` block with a dilation rate of 2 by calling the `resnet_block` function with the output of the interim model (`resnet50_upto_conv4`) we defined as the input and a dilation rate of 2:

```
resnet_block4_out = resnet_block(resnet50_upto_conv4.output, 2)
```

8.3.4 Implementing the atrous spatial pyramid pooling module

Here, we will discuss the most exciting innovation of the DeepLab v3 model. The atrous spatial pyramid pooling (ASPP) module serves two purposes:

- Aggregates multiscale information about an image, obtained through outputs produced using different dilation rates
- Combines highly summarized information obtained through global average pooling

The ASPP module gathers multiscale information by performing different convolutions on the last ResNet-50 output. Specifically, the ASPP module performs 1 × 1 convolution, 3 × 3 convolution (r = 6), 3 × 3 convolution (r = 12), and 3 × 3 convolution (r = 18), where r is the dilation rate. All of these convolutions have 256 output channels and are implemented as level 3 blocks (provided by the function `block_level3()`).

ASRP captures high-level information by performing global average pooling, followed by a 1 × 1 convolution with 256 output channels to match the output size of multiscale outputs, and finally a bilinear up-sampling layer to up-sample the height and width dimensions shrunk by the global average pooling. Remember that bilinear interpolation up-samples the images by computing the resulting pixels as an average of neighboring pixels. Figure 8.11 illustrates the ASPP module.

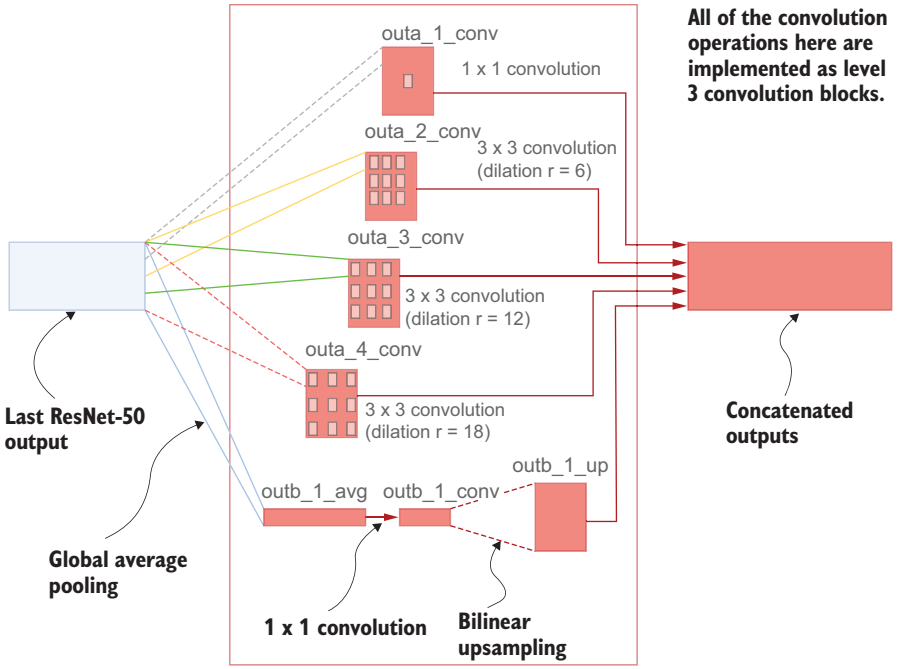

Figure 8.11 The ASPP module used in the DeepLab v3 model

The job of the ASPP module can be summarized as a concise function. We have all the tools we need to implement this function from the previous work we have done (see the following listing).

Listing 8.11 Implementing ASPP

```
def atrous_spatial_pyramid_pooling(inp):
    """ Defining the ASPP (Atrous spatial pyramid pooling) module """

    # Part A: 1x1 and atrous convolutions
    outa_1_conv = block_level3(
        inp, 256, (1,1), 1, '_aspp_a', 1, activation='relu'
    )
    outa_2_conv = block_level3(
        inp, 256, (3,3), 6, '_aspp_a', 2, activation='relu'
    )
    outa_3_conv = block_level3(
        inp, 256, (3,3), 12, '_aspp_a', 3, activation='relu'
    )
    outa_4_conv = block_level3(
        inp, 256, (3,3), 18, '_aspp_a', 4, activation='relu'
    )

    # Part B: global pooling
    outb_1_avg = layers.Lambda(
```

Define a 1 × 1 convolution.

Define a 3 x 3 convolution with 256 filters and a dilation rate of 6.

Define a 3 x 3 convolution with 256 filters and a dilation rate of 12.

Define a 3 x 3 convolution with 256 filters and a dilation rate of 18.

```
        lambda x: K.mean(x, axis=[1,2], keepdims=True)
    )(inp)
    outb_1_conv = block_level3(
        outb_1_avg, 256, (1,1), 1, '_aspp_b', 1, activation='relu'
    )
    outb_1_up = layers.UpSampling2D((24,24),
     interpolation='bilinear')(outb_1_avg)
    out_aspp = layers.Concatenate()(
        [outa_1_conv, outa_2_conv, outa_3_conv, outa_4_conv, outb_1_up]
    )

    return out_aspp

out_aspp = atrous_spatial_pyramid_pooling(resnet_block4_out)
```

Define a global average pooling layer.

Define a 1 × 1 convolution with 256 filters.

Up-sample the output using bilinear interpolation.

Concatenate all the outputs.

Create an instance of ASPP.

The ASPP module consists of four level 3 blocks, as outlined in the code. The first block comprises a 1 × 1 convolution with 256 filters without dilation (this produces `outa_1_conv`). The latter three blocks consist of 3 × 3 convolutions with 256 filters but with varying dilation rates (i.e., 6, 12, 18; they produce `outa_2_conv`, `outa_3_conv`, and `outa_4_conv`, respectively). This covers aggregating features from the image at multiple scales. However, we also need to preserve the global information about the image, similar to a global average pooling layer (`outb_1_avg`). This is achieved through a lambda layer that averages the input over the height and width dimensions:

```
outb_1_avg = layers.Lambda(lambda x: K.mean(x, axis=[1,2], keepdims=True))(inp)
```

The output of the averaging is then followed by a 1 × 1 convolution filter with 256 filters. Then, to bring the output to the same size as previous outputs, an up-sampling layer that uses bilinear interpolation is used (this produces `outb_1_up`):

```
outb_1_up = layers.UpSampling2D((24,24), interpolation='bilinear')(outb_1_avg)
```

Finally, all these outputs are concatenated to a single output using a `Concatenate` layer to produce the final output `out_aspp`.

8.3.5 Putting everything together

Now it's time to collate all the different components to create one majestic segmentation model. The next listing outlines the steps required to build the final model.

Listing 8.12 The final DeepLab v3 model

```
inp = layers.Input(shape=target_size+(3,))

resnet50= tf.keras.applications.ResNet50(
    include_top=False, input_tensor=inp,pooling=None
)

for layer in resnet50.layers:
    if layer.name == "conv5_block1_1_conv":
```

Define the RGB input layer.

Download and define the resnet50.

```
        break
    out = layer.output

resnet50_upto_conv4 = models.Model(resnet50.input, out)

resnet_block4_out = resnet_block(resnet50_upto_conv4.output, 2)

out_aspp = atrous_spatial_pyramid_pooling(resnet_block4_out)

out = layers.Conv2D(21, (1,1), padding='same')(out_aspp)
final_out = layers.UpSampling2D((16,16), interpolation='bilinear')(out)

deeplabv3 = models.Model(resnet50_upto_conv4.input, final_out)
```

Get the output of the last layer we're interested in.

Define the removed conv5 resnet block.

Define the ASPP module.

Define the final output.

Define the final model.

Define an interim model from the input up to the last layer of the conv4 block.

Note how the model has a linear layer that does not have any activation present (e.g., sigmoid or softmax). This is because we are planning to use a special loss function that uses logits (unnormalized scores obtained from the last layer before applying softmax) instead of normalized probability scores. Due to that, we will keep the last layer a linear output with no activation.

We have one final housekeeping step to perform: copying the weights from the original `conv5` block to the newly created `conv5` block in our model. To do that, first we need to store the weights from the original model as follows:

```
w_dict = {}
for l in ["conv5_block1_0_conv", "conv5_block1_0_bn",
          "conv5_block1_1_conv", "conv5_block1_1_bn",
          "conv5_block1_2_conv", "conv5_block1_2_bn",
          "conv5_block1_3_conv", "conv5_block1_3_bn"]:
    w_dict[l] = resnet50.get_layer(l).get_weights()
```

We cannot copy the weights to the new model until we compile the model, as weights are not initialized until the model is compiled. Before we do that, we need to learn loss functions and evaluation metrics that are used in segmentation tasks. To do that, we will need to implement custom loss functions and metrics and use them to compile the model. This will be discussed in the next section.

EXERCISE 3

You want to create a new pyramidal aggregation module called aug-ASPP. The idea is similar to the ASPP module we implemented earlier, but with a few differences. Let's say you have been given two interim outputs from the model: `out_1` and `out_2` (same size). You have to write a function, `aug_aspp`, that will take these two outputs and do the following:

- Perform atrous convolution with r = 16, 128 filters, 3 × 3 convolution, stride 1, and ReLU activation on `out_1` (output will be called `atrous_out_1`)
- Perform atrous convolution with r = 8, 128 filters, 3 × 3 convolution, stride 1, and ReLU activation on both `out_1` and `out_2` (output will be called `atrous_out_2_1` and `atrous_out_2_2`)

- Concatenate `atrous_out_2_1` and `atrous_out_2_2` (output will be called `atrous_out_2`)
- Apply 1 × 1 convolution with 64 filters to both `atrous_out_1` and `atrous_out_2` and concatenate (output will be called `conv_out`)
- Use bilinear up-sampling to double the size of `conv_out` (on height and width dimensions) and apply sigmoid activation

8.4 Compiling the model: Loss functions and evaluation metrics in image segmentation

In order to finalize the DeepLab v3 model (built using mostly the ResNet-50 structure and the ASPP module), we have to define a suitable loss function and metrics to measure the performance of the model. Image segmentation is quite different from image classification tasks, so the loss function and metrics don't necessarily translate to the segmentation problem. One key difference is that there is typically a large class imbalance in segmentation data, as a "background" class typically dominates an image compared to other object-related pixels. To get started, you read a few blog posts and research papers and identify weighted categorical cross-entropy loss and dice loss as good candidates. You focus on three different metrics: pixel accuracy, mean (class-weighted) accuracy, and mean IoU.

Loss functions and evaluation metrics used in image segmentation models are different from what is used in image classifiers. To start, image classifiers take in a single class label for a single image, whereas a segmentation model predicts a class for every single pixel in the image. This highlights the necessity of not only reimagining existing loss functions and metrics, but also inventing new losses and evaluation metrics that are more appropriate for the output produced by segmentation models. We will first discuss loss functions and then metrics.

8.4.1 Loss functions

A *loss function* is what is used to optimize the model whose purpose is to find the parameters that minimize a defined loss. A loss function used in a deep network must be *differentiable*, as the minimization of the loss happens with the help of gradients. The loss functions we'll use comprise two loss functions:

- Cross-entropy loss
- Dice loss

Cross-entropy loss

Cross-entropy loss is one of the most common losses used in segmentation tasks and can be implemented with just one line in Keras. We already used cross-entropy loss quite a few times but didn't analyze it in detail. However, it is worthwhile to review the underpinning mechanics that govern cross-entropy loss.

Cross-entropy loss takes in a predicted target and a true target. Both these tensors are of shape [batch size, height, width, object classes]. The object class dimension is a

one-hot encoded representation of which object class a given pixel belongs to. The cross-entropy loss is then computed for every pixel independently using

$$CE(i,j) = -\left(\sum_{k=1}^{c} y_k \log(\hat{y}_k) + (1-y_k)\log(1-\hat{y}_k)\right)$$

where $CE(i, j)$ represents the cross-entropy loss for pixel at position (i, j) on the image, c is the number of classes, and y_k and $\hat{y}_k$ represent the elements in the one-hot encoded vector and the predicted probability distribution over classes of that pixel. This is then summed across all the pixels to get the final loss.

Beneath the simplicity of the method, a critical issue lurks. Class imbalance is almost certain to rear its ugly head in image segmentation problems. You will find hardly any real-world images where each object occupies an equal area in the image. The good news is it is not very difficult to deal with this issue. This can be mitigated by assigning a weight for each pixel in the image, depending on the dominance of the class it represents. Pixels belonging to large objects will have smaller weights, whereas pixels belonging to smaller objects will have larger weights, providing an equal say despite the size in the final loss. The next listing shows how to do this in TensorFlow.

Listing 8.13 Computing the label weights for a given batch of data

```
def get_label_weights(y_true, y_pred):

    weights = tf.reduce_sum(tf.one_hot(y_true, num_classes), axis=[1,2])

    tot = tf.reduce_sum(weights, axis=-1, keepdims=True)

    weights = (tot - weights) / tot  # [b, classes]

    y_true = tf.reshape(y_true, [-1, y_pred.shape[1]*y_pred.shape[2]])

    y_weights = tf.gather(params=weights, indices=y_true, batch_dims=1)
    y_weights = tf.reshape(y_weights, [-1])

    return y_weights
```

Get the total pixels per-class in y_true.

Get the total pixels in y_true.

Compute the weights per-class. Rarer classes get more weight.

Reshape y_true to a [batch size, height*width]–sized tensor.

Make y_weights a vector.

Create a weight vector by gathering the weights corresponding to indices in y_true.

Here, for a given batch, we compute the weights as a sequence/vector that has a number of elements equal to `y_true`. First, we get the total number of pixels for each class by computing the sum over the width and height of the one-hot encoded `y_true` (i.e., has dimensions batch, height, width, and class). Here, a class that has a value larger than `num_classes` will be ignored. Next, we compute the total number of pixels per sample by taking the sum over the class dimension resulting in *tot* (a `[batch size, 1]`–sized tensor). Now the weights can be computed per sample and per class using

$$w_i = \frac{n - n_i}{n_i}$$

where n is the total number of pixels and n^i is the total number of pixels belonging to the i^{th} class. After that, we reshape `y_true` to shape [batch size, -1] as preparation for an important step in weight computation. As the final output, we want to create a tensor out of weights, where we gather elements from the `y_weights` that correspond to elements in `y_true`. In other words, we fetch the value from `y_weights`, where the index to fetch is given by the values in `y_true`. At the end, the result will be of the same shape and size as `y_true`. This is all we need to weigh the samples: multiply weights element-wise with the loss value for each pixel. To achieve this, we will use the function `tf.gather()`, which gathers the elements from a given tensor (`params`) while taking a tensor that represents indices (`indices`) and returns a tensor that is of the same shape as the indices:

```
y_weights = tf.gather(params=weights, indices=y_true, batch_dims=1)
```

Here, to ignore the batch dimension during performing the gather, we pass the argument `batch_dims` indicating how many batch dimensions we have. With that, we will define a function that outputs the weighted cross-entropy loss given a batch of predicted and true targets.

With the weights ready, we can now implement our first segmentation loss function. We will implement weighted cross-entropy loss. At a glance, the function masks irrelevant pixels (e.g., pixels belonging to unknown objects) and unwraps the predicted and true labels to get rid of the height and width dimensions. Finally, we can compute the cross-entropy loss using the built-in function in TensorFlow (see the next listing).

Listing 8.14 Implementing the weighted cross-entropy loss

```
def ce_weighted_from_logits(num_classes):

    def loss_fn(y_true, y_pred):
        """ Defining cross entropy weighted loss """

        valid_mask = tf.cast(
            tf.reshape((y_true <= num_classes - 1), [-1,1]), 'int32'
        )

        y_true = tf.cast(y_true, 'int32')
        y_true.set_shape([None, y_pred.shape[1], y_pred.shape[2]])

        y_weights = get_label_weights(y_true, y_pred)
        y_pred_unwrap = tf.reshape(y_pred, [-1, num_classes])
        y_true_unwrap = tf.reshape(y_true, [-1])

        return tf.reduce_mean(
            y_weights * tf.nn.sparse_softmax_cross_entropy_with_logits(
                y_true_unwrap * tf.squeeze(valid_mask),
                y_pred_unwrap * tf.cast(valid_mask, 'float32'))
        )

    return loss_fn
```

Define the valid mask, masking unnecessary pixels.

Some initial setup that casts y_true to int and sets the shape

Get the label weights.

Unwrap y_pred and y_true so that batch, height, and width dimensions are squashed.

Compute the cross-entropy loss with y_true, y_pred, and the mask.

Return the function that computes the loss.

You might be thinking, "Why is the loss defined as a nested function?" This is a standard pattern we have to follow if we need to include extra arguments to our loss function (i.e., `num_classes`). All we are doing is capturing the computations of the loss function in the `loss_fn` function and then creating an outer function `ce_weighted_from_logits()` that will return the function that encapsulates the loss computations (i.e., `loss_fn`).

Specifically, a valid mask is created to indicate whether the labels in `y_true` are less than the number of classes. Any label that has a value larger than the number of classes is ignored (e.g., unknown objects). Next, we get the weight vector and indicate a weight for each pixel using the `get_label_weights()` function. We will unwrap `y_pred` to a `[-1, num_classes]`–sized tensor, as `y_pred` contains *logits* (i.e., unnormalized probability scores output by the model) across all classes in the data set. `y_true` will be unwrapped to a vector (i.e., a single dimension), as `y_true` only contains the class label. Finally, we use `tf.nn.sparse_softmax_cross_entropy_with_logits()` to compute the loss over masked predicted and true targets. The function takes two arguments, `labels` and `logits`, which are self-explanatory. We can make two salient observations:

- We are computing sparse cross-entropy loss (i.e., not standard cross-entropy loss).
- We are computing cross-entropy loss from logits.

When using sparse cross entropy, we don't have to one-hot encode the labels, so we can skip this, which leads to a more memory-efficient data pipeline. This is because one-hot encoding is handled internally by the model. By using a sparse loss, we have less to worry about.

Computing the loss from logits (i.e., unnormalized scores) instead of from normalized probabilities leads to better and more stable gradients. Therefore, whenever possible, make sure to use logits instead of normalized probabilities.

DICE LOSS

The second loss we will discuss is called the *dice loss*, which is computed as

$$DiceLoss = 1 - \frac{2\ \text{Intersection}}{\text{Intersection} \mid \text{Union}}$$

Here, the intersection between the prediction and target tensors can be computed with element-wise multiplication, whereas the union can be computed using element-wise addition between the prediction and the target tensors. You might be thinking that using element-wise operations is a strange way to compute intersection and union. To understand the reason behind this, I want to refer to a statement made earlier: a loss function used in a deep network must be *differentiable*.

This means that we cannot use the standard conventions we use to compute intersection and union from a given set of values. Rather, we need to resort to a differentiable computation, leading to intersection and union between two tensors. Intersection can be computed by taking element-wise multiplication between the predicted and true targets. Union can be computed by taking the element-wise addition between the predicted and true targets. Figure 8.12 clarifies how these operations lead to intersection and union between two tensors.

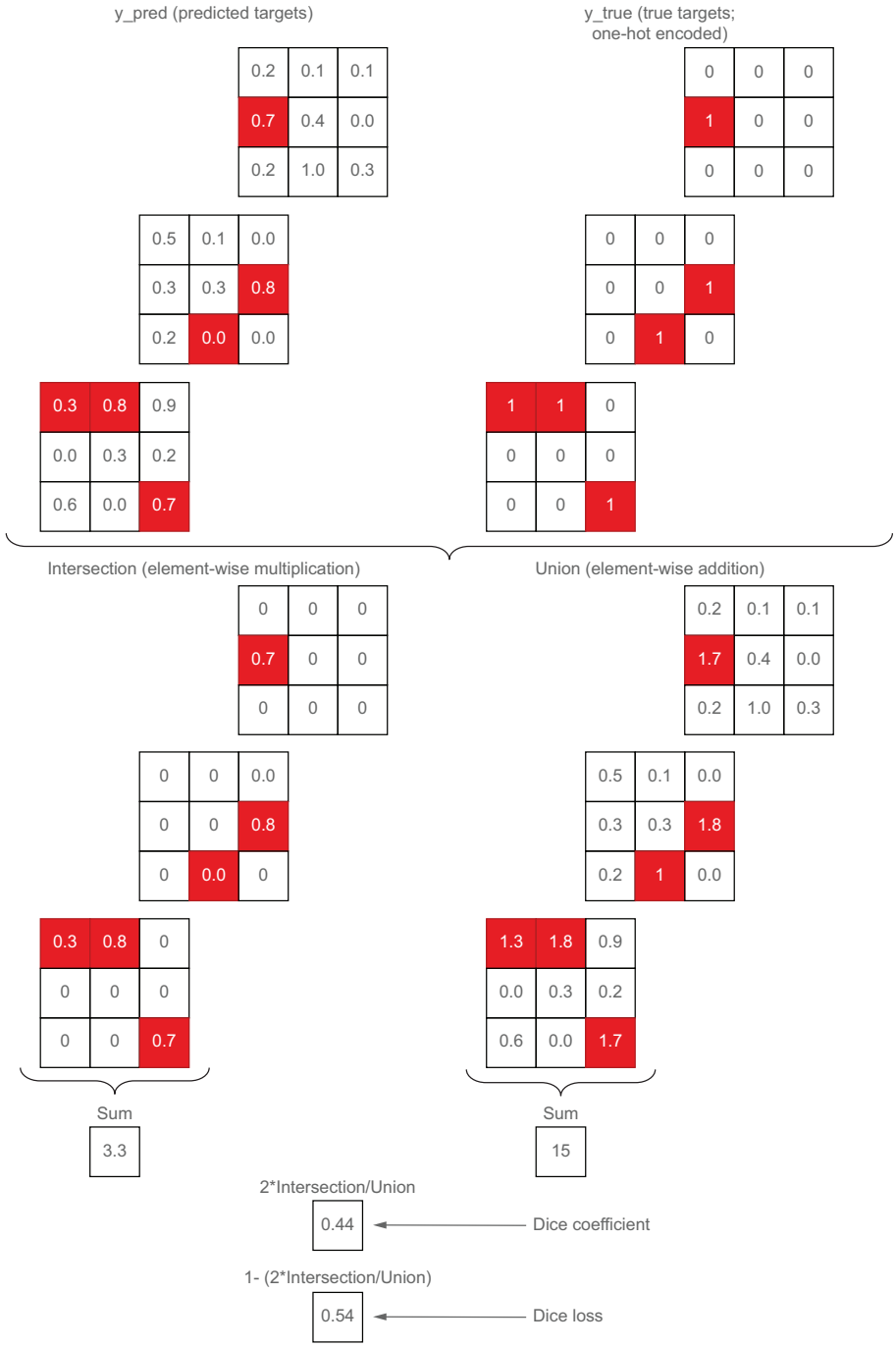

Figure 8.12 Computations involved in dice loss. The intersection can be computed as a differentiable function by taking element-wise multiplication, whereas union can be computed as the element-wise sum.

This loss is predominantly focused on maximizing the intersection between the predicted and true targets. The multiplier of 2 is used to balance out the duplication of values that comes from the overlap between the intersection and the union, found in the denominator (see the following listing).

Listing 8.15 Implementing the dice loss

```
def dice_loss_from_logits(num_classes):
    """ Defining the dice loss 1 - [(2* i + 1)/(u + i)]"""

    def loss_fn(y_true, y_pred):

        smooth = 1.

        # Convert y_true to int and set shape
        y_true = tf.cast(y_true, 'int32')
        y_true.set_shape([None, y_pred.shape[1], y_pred.shape[2]])

        # Get pixel weights
        y_weights = tf.reshape(get_label_weights(y_true, y_pred), [-1, 1])

        # Apply softmax to logits
        y_pred = tf.nn.softmax(y_pred)

        y_true_unwrap = tf.reshape(y_true, [-1])
        y_true_unwrap = tf.cast(
            tf.one_hot(tf.cast(y_true_unwrap, 'int32'), num_classes),
➥ 'float32'
        )
        y_pred_unwrap = tf.reshape(y_pred, [-1, num_classes])

        intersection = tf.reduce_sum(y_true_unwrap * y_pred_unwrap * y_weights)

        union = tf.reduce_sum((y_true_unwrap + y_pred_unwrap) * y_weights)

        score = (2. * intersection + smooth) / ( union + smooth)

        loss = 1 - score

        return loss

    return loss_fn
```

Initial setup for y_true

Get the label weights and reshape it to a [-1, 1] shape.

Apply softmax on y_pred to get normalized probabilities.

Unwrap y_pred and one-hot–encoded y_true to the [-1, num_classes] shape.

Compute intersection using element-wise multiplication.

Compute union using element-wise addition.

Compute the dice coefficient.

Compute the dice loss.

Here, `smooth` is a smoothing parameter that we'll use to avoid potential NaN values resulting in division by zero. After that we do the following:

- Obtain weights for each `y_true` label
- Apply a softmax activation to `y_pred`
- Unwrap `y_pred` to the `[-1, num_classes]` tensor and `y_true` to a `[-1]`–sized vector

Then intersection and union are computed for `y_pred` and `y_true`. Specifically, intersection is computed as the result of element-wise multiplication of `y_pred` and `y_true` and the union as the result of the element-wise addition of `y_pred` and `y_true`.

Focal loss

Focal loss is a relatively novel loss introduced in the paper "Focal Loss for Dense Object Prediction" (https://arxiv.org/pdf/1708.02002.pdf). Focal loss was introduced to combat the severe class imbalance found in segmentation tasks. Specifically, it solves a problem in many easy examples (e.g., samples from common classes with smaller loss), over-powering small numbers of hard examples (e.g., samples from rare classes with larger loss). Focal loss solves this problem by introducing a modulating factor that will down-weight easy examples, so, naturally, the loss function focuses more on learning hard examples.

The loss function we will use to optimize the segmentation model will be the loss resulting from addition of sparse cross-entropy loss and dice loss (see the next listing).

Listing 8.16 Final combined loss function

```
def ce_dice_loss_from_logits(num_classes):

    def loss_fn(y_true, y_pred):
        # Sum of cross entropy and dice losses
        loss = ce_weighted_from_logits(num_classes)(
            tf.cast(y_true, 'int32'), y_pred
        ) + dice_loss_from_logits(num_classes)(
            y_true, y_pred
        )

        return loss

    return loss_fn
```

Next, we will discuss evaluation metrics.

8.4.2 Evaluation metrics

Evaluation metrics play a vital role in model training as a health check for the model. This means low performance/issues can be quickly identified by making sure evaluation metrics behave in a reasonable way. Here we will discuss three different metrics:

- Pixel
- Mean accuracy
- Mean IoU

We will implement these as custom metrics by leveraging some of the existing metrics in TensorFlow, where you have to subclass from the `tf.keras.metrics.Metric` class or one of the existing metrics. This means that you create a new Python class, which

inherits from the base `tf.keras.metrics.Metric` base class of one of the existing concrete metrics classes:

```
class MyMetric(tf.keras.metrics.Metric):

  def __init__(self, name='binary_true_positives', **kwargs):
    super(MyMetric, self).__init__(name=name, **kwargs)

    # Create state related variables

  def update_state(self, y_true, y_pred, sample_weight=None):

    # update state in this function

  def result(self):

    # We return the result computed from the state

  def reset_states():
    # Do what's required to reset the maintained states
    # This function is called between epochs
```

The first thing you need to understand about a metric is that it is a stateful object, meaning it maintains a state. For example, a single epoch has multiple iterations and assumes you're interested in computing the accuracy. The metric needs to accumulate the values required to compute the accuracy over all the iterations so that at the end, it can compute the average accuracy for that epoch. When defining a metric, there are three functions you need to be mindful of: `__init__`, `update_state`, `result`, and `reset_states`.

Let's get concrete by assuming that we are implementing an accuracy metric (i.e., the number of elements in `y_pred` that matched `y_true` as a percentage). It needs to maintain a total: the sum of all the accuracy values we passed and the count (number of accuracy values we passed). With these two state elements, we can compute the mean accuracy at any time. When implementing the accuracy metric, you implement these functions:

- `__init__`—Defines two states; total and count
- `update_state`—Updates total and count based on `y_true` and `y_pred`
- `result`—Computes the mean accuracy as total/count
- `reset_states`—Resets both the total and count (this needs to happen at the beginning of an epoch)

Let's see how this knowledge translates to the evaluation metrics we're interested in solving.

PIXEL AND MEAN ACCURACIES

Pixel accuracy is the simplest metric you can think of. It measures the pixel-wise accuracy between the prediction and the true target (see the next listing).

Listing 8.17 Implementing the pixel accuracy metric

```
class PixelAccuracyMetric(tf.keras.metrics.Accuracy):

  def __init__(self, num_classes, name='pixel_accuracy', **kwargs):
    super(PixelAccuracyMetric, self).__init__(name=name, **kwargs)

  def update_state(self, y_true, y_pred, sample_weight=None):

    y_true.set_shape([None, y_pred.shape[1], y_pred.shape[2]])
    y_true = tf.reshape(y_true, [-1])

    y_pred = tf.reshape(tf.argmax(y_pred, axis=-1),[-1])

    valid_mask = tf.reshape((y_true <= num_classes - 1), [-1])

    y_true = tf.boolean_mask(y_true, valid_mask)
    y_pred = tf.boolean_mask(y_pred, valid_mask)

    super(PixelAccuracyMetric, self).update_state(y_true, y_pred)
```

Set the shape of y_true (in case it is undefined).

Reshape y_true to a vector.

Reshape y_pred after taking argmax to a vector.

Define a valid mask (mask out unnecessary pixels).

Gather pixels/labels that satisfy the valid_mask condition.

With the processed y_true and y_pred, compute the accuracy using the update_state() function.

Pixel accuracy computes the one-to-one match between predicted pixels and true pixels. To compute this, we subclass from `tf.keras.metrics.Accuracy` as it has all the computations we need. To do this, we override the `update_state` function as shown. There are a few things we need to take care of:

- We need to set the shape of `y_true` as a precaution. This is because when working with `tf.data.Dataset`, sometimes the shape is lost.
- Reshape `y_true` to a vector.
- Get the class labels of `y_pred` by performing `tf.argmax()` and reshape it to a vector.
- Define a valid mask that ignores unwanted classes (e.g., unknown objects).
- Get the pixels that satisfy only the `valid_mask` filter.

Once we complete these tasks, we simply call the parent object's (i.e., `tf.keras.metrics.Accuracy`) `update_state` method with the corresponding arguments. We don't have to override `result()` and `reset_states()` functions, as they already contain the correct computations.

We said that class imbalance is prevalent in image segmentation problems. Typically, background pixels will spread in a large region of the image, potentially leading to misguided conclusions. Therefore, a slightly better approach might be to compute the accuracy individually per class and then average it. Enter mean accuracy, which prevents the undesired characteristics of pixel accuracy (see the next listing).

Listing 8.18 Implementing the mean accuracy metric

```
class MeanAccuracyMetric(tf.keras.metrics.Mean):

  def __init__(self, num_classes, name='mean_accuracy', **kwargs):
    super(MeanAccuracyMetric, self).__init__(name=name, **kwargs)

  def update_state(self, y_true, y_pred, sample_weight=None):

    smooth = 1

    y_true.set_shape([None, y_pred.shape[1], y_pred.shape[2]])

    y_true = tf.reshape(y_true, [-1])
    y_pred = tf.reshape(tf.argmax(y_pred, axis=-1),[-1])

    valid_mask = tf.reshape((y_true <= num_classes - 1), [-1])

    y_true = tf.boolean_mask(y_true, valid_mask)
    y_pred = tf.boolean_mask(y_pred, valid_mask)

    conf_matrix = tf.cast(
        tf.math.confusion_matrix(y_true, y_pred, num_classes=num_classes),
'float32'
    )
    true_pos = tf.linalg.diag_part(conf_matrix)

    mean_accuracy = tf.reduce_mean(
        (true_pos + smooth)/(tf.reduce_sum(conf_matrix, axis=1) + smooth)
    )

    super(MeanAccuracyMetric, self).update_state(mean_accuracy)
```

Initial setup

Compute the confusion matrix using y_true and y_pred.

Get the true positives (elements on the diagonal).

Compute the mean accuracy using true positives and true class counts for each class.

Compute the average of mean_accuracy using the update_state() function.

The `MeanAccuracyMetric` will branch out from `tf.keras.metrics.Mean`, which computes the average over a given sequence of values. The plan is to compute the `mean_accuracy` within the `update_state()` function and then pass the value to the parent's `update_state()` function so that we get the average value of mean accuracy. First, we perform the initial setup and clean-up of `y_true` and `y_pred` we discussed earlier.

Afterward, we compute the confusion matrix (figure 8.13) from predicted and true targets. A confusion matrix for an n-way classification problem (i.e., a classification problem with n possible classes) is defined as a $n \times n$ matrix. Here, the element at the (i, j) position indicates how many instances were predicted as belonging to the i^{th} class but actually belong to the j^{th} class. Figure 8.13 portrays this type of confusion matrix. We can get the true positives by extracting the diagonal (i.e., (i, i) elements in the matrix for all $1 <= i <= n$). We can now compute the mean accuracy in two steps:

1. Perform element-wise division on the true positive count by actual counts for all the classes. This produces a vector whose elements represents per-class accuracy.
2. Compute the vector mean that resulted from step 1.

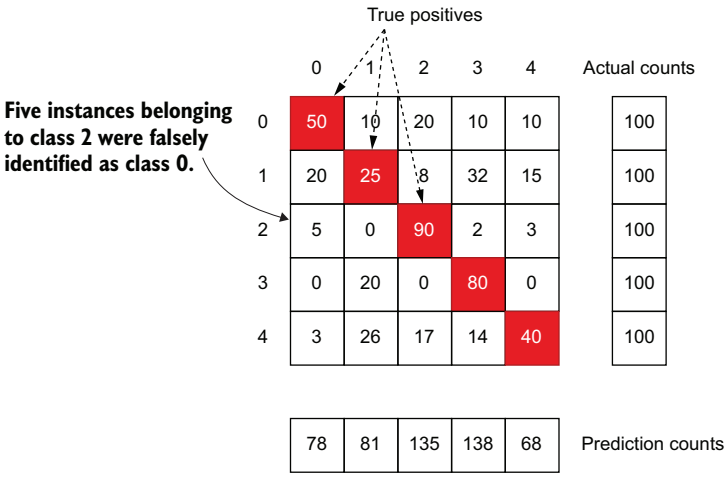

Figure 8.13 Illustration of a confusion matrix for a five-class classification problem. The shaded boxes represent true positives.

Finally, we pass the mean accuracy to its parent's `update_state()` function.

MEAN IOU

Mean IoU (mean intersection over union) is a popular evaluation metric pick for segmentation tasks and has close ties to the dice loss we discussed earlier, as they both use the concept of intersection and union to compute the final result (see the next listing).

Listing 8.19 Implementing the mean IoU metric

```
class MeanIoUMetric(tf.keras.metrics.MeanIoU):

  def __init__(self, num_classes, name='mean_iou', **kwargs):
    super(MeanIoUMetric, self).__init__(num_classes=num_classes, name=name,
     **kwargs)

  def update_state(self, y_true, y_pred, sample_weight=None):

    y_true.set_shape([None, y_pred.shape[1], y_pred.shape[2]])
    y_true = tf.reshape(y_true, [-1])

    y_pred = tf.reshape(tf.argmax(y_pred, axis=-1),[-1])

    valid_mask = tf.reshape((y_true <= num_classes - 1), [-1])
```

```
    # Get pixels corresponding to valid mask
    y_true = tf.boolean_mask(y_true, valid_mask)
    y_pred = tf.boolean_mask(y_pred, valid_mask)

    super(MeanIoUMetric, self).update_state(y_true, y_pred)
```

After the initial setup of y_true and y_pred, all we need to do is call the parent's update_state() function.

The mean IoU computations are already found in `tf.keras.metrics.MeanIoU`. Therefore, we will use that as our parent class. All we need to do is perform the aforementioned setup for `y_true` and `y_pred` and then call the parent's `update_state()` function. Mean IoU is computed as

$$IoU = \frac{\text{True Positives}}{(\text{True Positives} + \text{False Positives} + \text{False Negatives})}$$

Various elements used in this computation are depicted in figure 8.14.

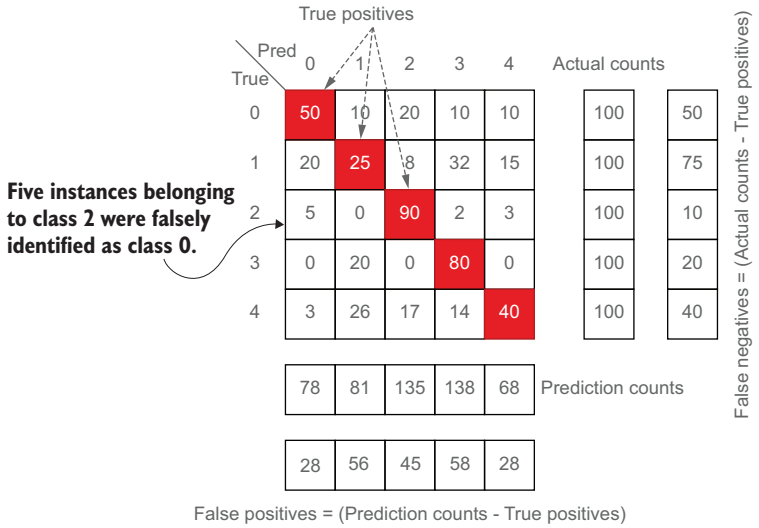

Figure 8.14 Confusion matrix and how it can be used to compute false positives, false negatives, and true positives

We now understand the loss functions and evaluation metrics that are available to us and have already implemented them. We can incorporate these losses to compile the model:

```
deeplabv3.compile(
    loss=ce_dice_loss_from_logits(num_classes),
    optimizer=optimizer,
    metrics=[
        MeanIoUMetric(num_classes),
        MeanAccuracyMetric(num_classes),
        PixelAccuracyMetric(num_classes)
    ])
```

Remember that we stored the weights from a convolution block we removed earlier. Now that we have compiled the model, we can copy the weights to the new model using the following syntax:

```
# Setting weights for newly added layers
for k, w in w_dict.items():
    deeplabv3.get_layer(k).set_weights(w)
```

We now move on to training the model with the data pipeline and the model we defined.

EXERCISE 4

You are coming up with a new loss function that computes the disjunctive union between `y_true` and `y_pred`. The disjunctive union between two sets A and B is the set of elements that are in either A or B but not in the intersection. You know you can compute the intersection with element-wise multiplication and union with element-wise addition of `y_true` and `y_pred`. Write the equation to compute the disjunctive union as a function of `y_true` and `y_pred`.

8.5 Training the model

You're coming to the final stages of the first iteration of your product. Now it's time to put the data and knowledge you garnered to good use (i.e., train the model). We will train the model for 25 epochs and monitor the pixel accuracy, mean accuracy, and mean IoU metrics. During the training, we will measure the performance on validation data set.

Training the model is the easiest part, as we have done the hard work that leads up to training. It is now just a matter of calling `fit()` with the correct parameters on the DeepLab v3 we just defined, as the following listing shows.

Listing 8.20 Training the model

```
if not os.path.exists('eval'):
    os.mkdir('eval')

csv_logger = tf.keras.callbacks.CSVLogger(
    os.path.join('eval','1_pretrained_deeplabv3.log')

monitor_metric = 'val_loss'
mode = 'min' if 'loss' in monitor_metric else 'max'
print("Using metric={} and mode={} for EarlyStopping".format(monitor_metric,
     mode))
lr_callback = tf.keras.callbacks.ReduceLROnPlateau(
    monitor=monitor_metric, factor=0.1, patience=3, mode=mode, min_lr=1e-8
)
es_callback = tf.keras.callbacks.EarlyStopping(
    monitor=monitor_metric, patience=6, mode=mode
)
```

Train logger

Set the mode for the following callbacks automatically by looking at the metric name.

Learning rate scheduler

Early stopping callback

```
# Train the model
deeplabv3.fit(                                    <-- Train the model while using the validation set for learning rate adaptation and early stopping.
    x=tr_image_ds, steps_per_epoch=n_train,
    validation_data=val_image_ds, validation_steps=n_valid,
    epochs=epochs, callbacks=[lr_callback, csv_logger, es_callback])
```

First, we will define a directory called eval if it does not exist. The training logs will be saved in this directory. Next, we define three different callbacks to be used during the training:

- `csv_logger`—Logs the training loss/metrics and validation loss/metrics
- `lr_callback`—Reduces the learning rate by a factor of 10, if the validation loss does not decrease within three epochs
- `es_callback`—Performs early stopping if the validation loss does not decrease within six epochs

NOTE On an Intel Core i5 machine with an NVIDIA GeForce RTX 2070 8GB, the training took approximately 45 minutes to run 25 epochs.

With that, we call `deeplabv3.fit()` with the following parameters:

- `x`—The tf.data pipeline producing training instances (set to `tr_image_ds`).
- `steps_per_epoch`—Number of steps per epoch. This is obtained by computing the number of training instances and dividing it by the batch size (set to `n_train`).
- `validation_data`—The `tf.data` pipeline producing validation instances. This is obtained by computing the number of validation instances and dividing it by the batch size (set to `val_image_ds`).
- `epochs`—Number of epochs (set to epochs).
- `callbacks`—The callbacks we set up earlier (set to `[lr_callback, csv_logger, es_callback]`).

After the model is trained, we will evaluate it on the test set. We will also visualize segmentations generated by the model.

EXERCISE 5

You have a data set of 10,000 samples and have split it into 90% training data and 10% validation data. You use a batch size of 10 for training and a batch size of 20 for validation. How many training and validation steps will be there in a single epoch?

8.6 Evaluating the model

Let's take a moment to reflect on what we have done so far. We defined a data pipeline to read images and prepare them as inputs and targets for the model. Then we defined a model known as DeepLab v3 that uses a pretrained ResNet-50 as its backbone and a special module called atrous spatial pyramid pooling to predict the final segmentation mask. Then we defined task-specific losses and metrics to make sure we

could evaluate the model with a variety of metrics. Afterward, we trained the model. Now it's time for the ultimate reveal. We will measure the performance on an unseen test data set to see how well the model does. We will also visualize the model outputs and compare them against the real targets by plotting them side by side.

We can run the model over the unseen test images and gauge how well it is performing. To do that, we execute the following:

```
deeplabv3.evaluate(test_image_ds, steps=n_valid)
```

The size of the test set is the same as the validation set, as we split the images listed in val.txt into two equal validation and test sets. This will return around

- 62% mean IoU
- 87% mean accuracy
- 91% pixel accuracy

These are very respectable scores given our circumstances. Our training data set consists of less than 1,500 segmented images. Using this data, we were able to train a model that achieves around 62% mean IoU on a test data set of approximately size 725.

What does state of the art look like?

The state-of-the-art performance on Pascal VOC 2012 reports around 90% mean IoU (http://mng.bz/o2m2). However, these are models that are much larger and complex than what we used here. Furthermore, they are typically trained with significantly more data by using an auxiliary data set known as the semantic boundary data set (SBD) (introduced in the paper http://mng.bz/nNve). This will push the training datapoint count to over 10,000 (close to seven times the size of our current training set).

You can further investigate the model by visually inspecting some of the results our module produces. After all, it is a vision model that we are developing. Therefore, we should not rely solely on numbers to make decisions and conclusions. We should also visually analyze the results before settling on a conclusion.

What would the results from a U-Net based network look like?

Under similar conditions provided for the DeepLab v3 model, the U-Net model built with a pretrained ResNet-50 model as the encoder was only able to achieve approximately 32.5% mean IoU, 78.5% mean accuracy, and 81.5% pixel accuracy. The implementation is provided in the Jupyter notebook in the ch08 folder.

On an Intel Core i5 machine with an NVIDIA GeForce RTX 2070 8GB, the training took approximately 55 minutes to run 25 epochs.

There is a detailed explanation of the U-Net model in appendix B.

To complete this investigation, we will get a random sample from the test set and ask the model to predict the segmentation map for each of those images. Then we will plot the results side by side to ensure that our model is doing a good job (figure 8.15).

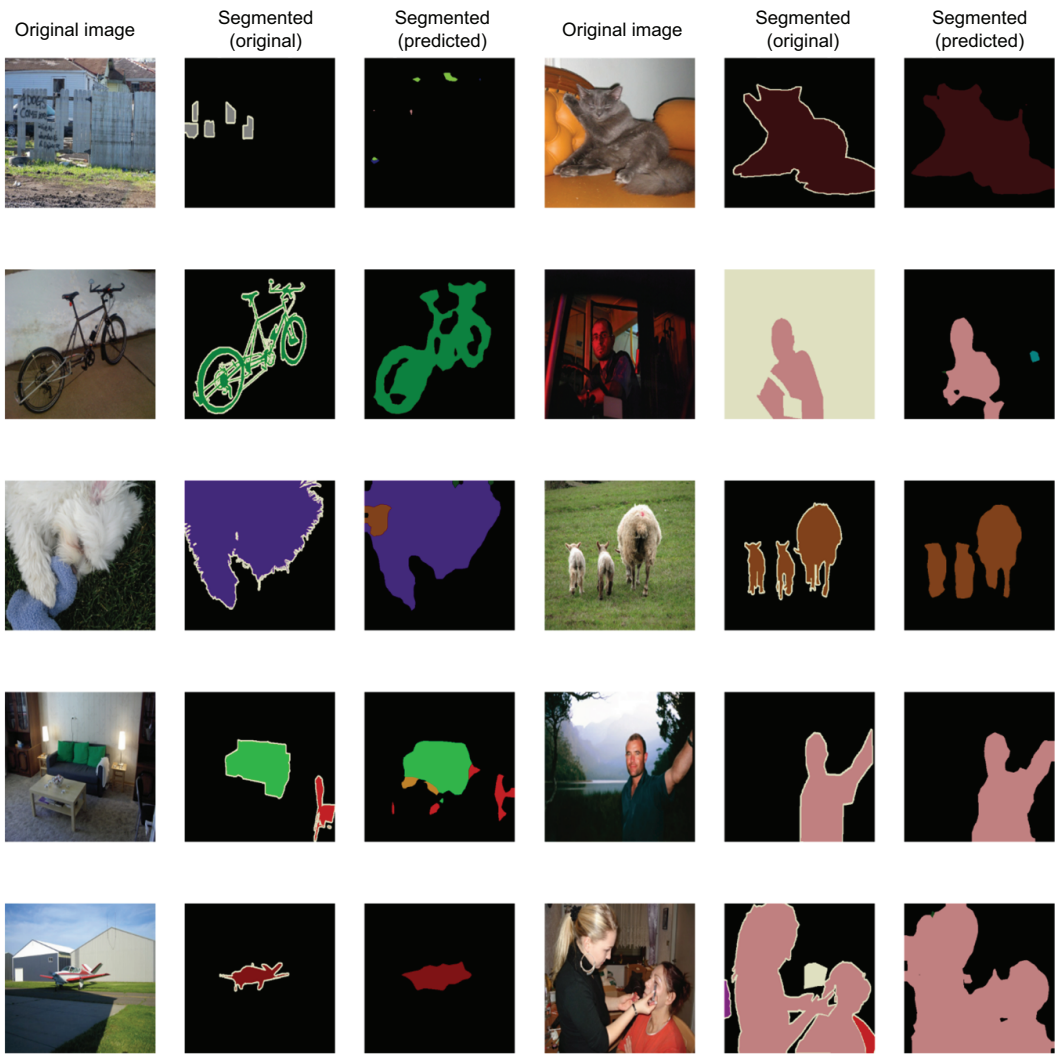

Figure 8.15 Comparing the true annotated targets to model predictions. You can see that the model is quite good at separating objects from different backgrounds.

We can see that unless it is an extremely difficult image (e.g., the top-left image, where there's a car obscured by a gate), our model does a very good job. It can identify almost all the images found in the sample we analyzed with high accuracy. The code for visualizing the images is provided in the notebook.

This concludes our discussion of image segmentation. In the next few chapters, we will discuss several natural language processing problems.

EXERCISE 6

You are given

- A model (called model)
- A batch of images called `batch_image` (already preprocessed and ready to be fed to a model)
- A corresponding batch of targets, `batch_targets` (the true segmentation mask in one-hot encoded format)

Write a function called `get_top_bad_examples(model, batch_images, batch_targets, n)` that will return the top `n` indices of the hardest (highest loss) images in `batch_images`. Given a predicted mask and a target mask, you can use the sum over element-wise multiplication as the loss of a given image.

You can use the `model.predict()` function to make a prediction on `batch_images`, and it will return a predicted mask as the same size as `batch_targets`. Once you compute the losses for the batch (`batch_loss`), you can use the `tf.math.top_k(batch_loss, n)` function to get the indices of elements with the highest value. `tf.math.top_k()` returns a tuple containing the top values and indices of a given vector, in that order.

Summary

- Segmentation models fall into two broad categories: semantic segmentation and instance segmentation.
- The `tf.data` API provides various functionality to implement complex data pipelines, such as using custom NumPy functions, performing quick transformations using `tf.data.Dataset.map()`, and I/O optimization techniques like prefetch and cache.
- DeepLab v3 is a popular segmentation model that uses a pretrained ResNet-50 model as its backbone and atrous convolutions to increase the receptive field by inserting holes (i.e., zeros) between the kernel weights.
- The DeepLab v3 model uses a module known as atrous spatial pyramid pooling to aggregate information at multiple scales, which helps to create a fine-grained segmented output.
- In segmentation tasks, cross entropy and dice loss are two popular losses, whereas pixel accuracy, mean accuracy, and mean IoU are popular evaluation metrics.
- In TensorFlow, loss functions can be implemented as stateless functions. But metrics must be implemented as stateful objects by subclassing from the `tf.keras.metrics.Metric` base class or a suitable class.
- The DeepLab v3 model achieved a very good accuracy of 62% mean IoU on the Pascal VOC 2010 data set.

Answers to exercises

Exercise 1

```
palettized_image = np.zeros(shape=rgb_image.shape[:-1])
for i in range(rgb_image.shape[0]):
    for j in range(rgb_image.shape[1]):
        for k in range(palette.shape[0]):
            if (palette[k] == rgb_image[i,j]).all():
                palettized_image[i,j] = k
                break
```

Exercise 2

```
dataset_a = tf.data.Dataset.from_tensor_slices(a)
dataset_b = tf.data.Dataset.from_tensor_slices(b)

image_ds = tf.data.Dataset.zip((dataset_a, dataset_b))

image_ds = image_ds.map(
            lambda x, y: (x, tf.image.resize(y, (64,64),  method='nearest'))
        )

image_ds = image_ds.map(
            lambda x, y: ((x-128.0)/255.0, tf.image.resize(y, (64,64),
     method='nearest'))
        )

image_ds = image_ds.batch(32).repeat(5).prefetch(tf.data.experimental.AUTOTUNE)
```

Exercise 3

```
import tensorflow.keras.layers as layers

def aug_aspp(out_1, out_2):

    atrous_out_1 = layers.Conv2D(128, (3,3), dilation_rate=16,
➥ padding='same', activation='relu')(out_1)

    atrous_out_2_1 = layers.Conv2D(128, (3,3), dilation_rate=8,
➥ padding='same', activation='relu')(out_1)
    atrous_out_2_2 = layers.Conv2D(128, (3,3), dilation_rate=8,
➥ padding='same', activation='relu')(out_2)
    atrous_out_2 = layers.Concatenate()([atrous_out_2_1, atrous_out_2_2])

    tmp1 = layers.Conv2D(64, (1,1), padding='same',
     activation='relu')(atrous_out_1)
    tmp2 = layers.Conv2D(64, (1,1), padding='same',
     activation='relu')(atrous_out_2)
    conv_out = layers.Concatenate()([tmp1,tmp2])

    out = layers.UpSampling2D((2,2), interpolation='bilinear')(conv_out)
    out = layers.Activation('sigmoid')(out)

    return out
```

Exercise 4

```
out = (y_pred - (y_pred * y_true)) + (y_true - (y_pred * y_true))
```

Exercise 5

```
9000 and 500
```

Exercise 6

```
def get_top_n_bad_examples(model, batch_images, batch_targets, n):

    batch_pred = model.predict(batch_images)

    batch_loss = tf.reduce_sum(batch_pred*batch_targets, axis=[1,2,3])

    _, hard_inds = tf.math.top_k(batch_loss, n)

    return hard_inds
```