CHAPTER 8

Telling Things Apart: Image Segmentation

Chapter Overview

In the previous chapter, advanced computer vision architectures such as Inception networks, data augmentation, dropout, early stopping, and transfer learning were used to improve image classification performance. Image classification focuses on predicting a single label per image.

This chapter introduces **image segmentation**, a more advanced computer vision task where **each pixel** in an image is classified. Image segmentation enables models to identify **multiple objects and their precise locations** within a single image. This task is fundamental in applications such as self-driving cars, medical image analysis, image retrieval, and astronomical image processing.

Unlike image classification (a **sparse prediction problem**), image segmentation is a **dense prediction problem**, assigning a class label to every pixel. Segmentation models commonly reuse pretrained image classification networks as backbones to extract multiscale image features efficiently.

Types of Image Segmentation

Image segmentation methods can be categorized into two main types:

- **Semantic segmentation**
  All pixels belonging to the same object category are assigned the same class label. Multiple instances of the same object type (e.g., several people) share the same label.
- **Instance segmentation**
  Each individual object instance is assigned a unique label, even if multiple objects belong to the same category. This approach is more complex than semantic segmentation.

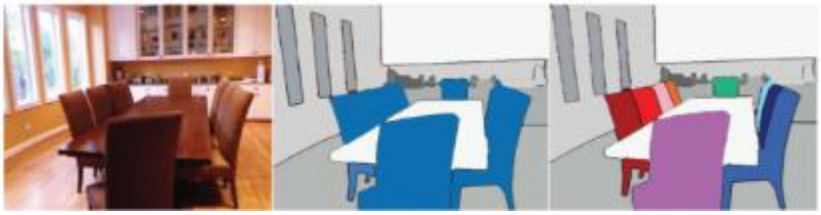


Figure 8.1
Semantic segmentation versus instance segmentation

This chapter focuses exclusively on **semantic segmentation**.

8.1 Understanding the Data

Problem Context

The task is framed around developing a navigation system for small remote-control (RC) vehicles. An image segmentation model is used as a first step to understand the environment. The segmentation output will later feed into a navigation-planning model that adapts behavior based on user preferences (safe vs. adventurous navigation).

For this purpose, the **PASCAL VOC 2012 dataset** is selected, as it contains realistic indoor and outdoor scenes commonly found in urban and domestic environments.

Characteristics of Segmentation Data

Segmentation datasets differ from classification datasets in several key ways:

- **Input**: A standard RGB image.
- **Target**: An image where each pixel represents a class label using a predefined color palette.
- Each object category (including background) is assigned a unique color.
- The number of colors equals the number of object classes.

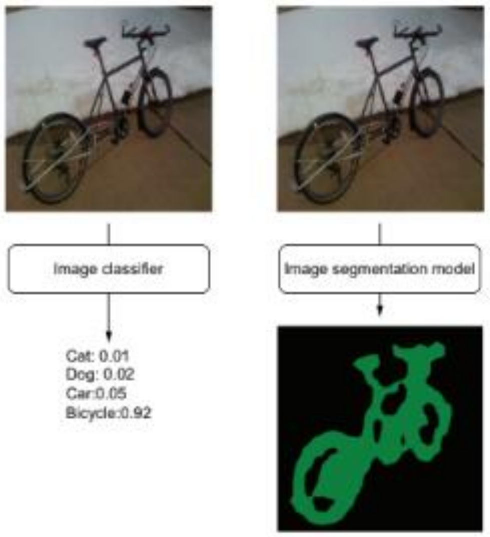
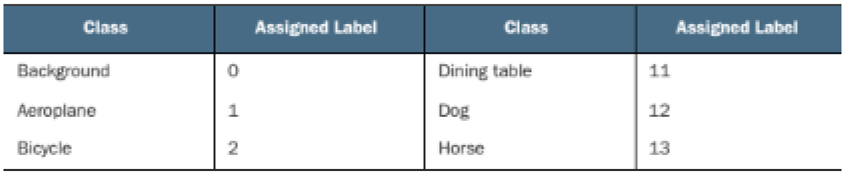


Figure 8.2 Inputs and outputs of an image classifier versus an image segmentation model

PASCAL VOC 2012 Dataset

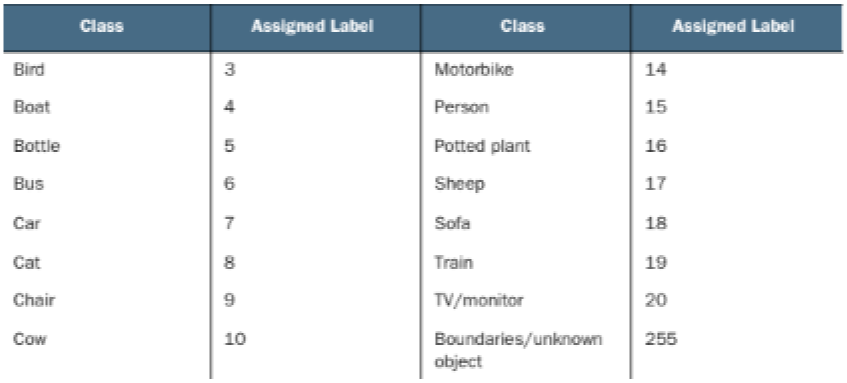
- Contains **22 classes** (20 object classes, background, and boundary/unknown).
- Target images use specific colors to represent object classes.
- White pixels represent object boundaries or unknown regions.

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



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

Table 8.1 (continued)
Continuation of class–label mappings



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

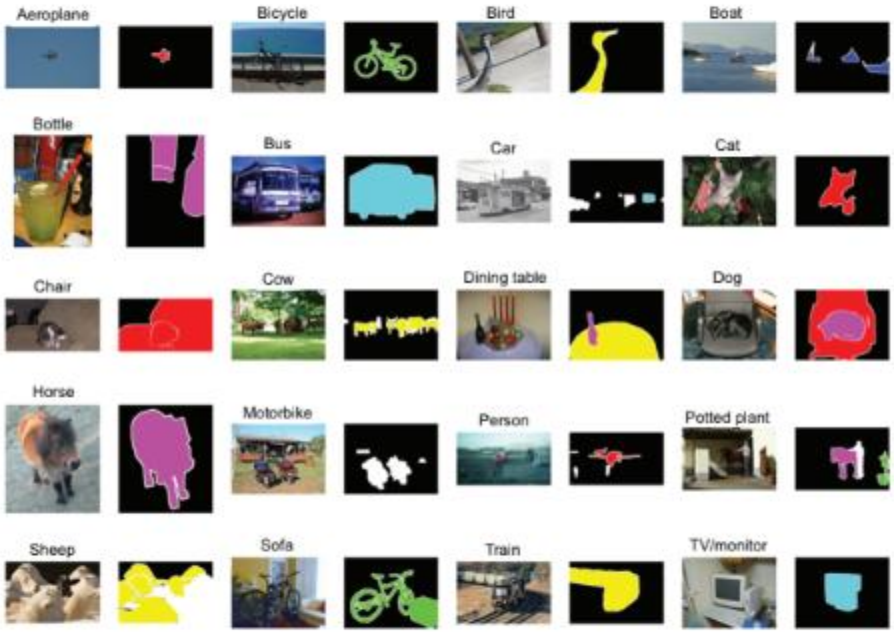


Figure 8.3 Samples from the PASCAL VOC 2012 data set. The data set shows a single example image, along with the annotated segmentation of it for the 20 different object classes.

Example of Segmentation Data

A single sample may contain multiple objects, each represented by a different color in the annotated image. Object boundaries can be identified visually, often highlighted by white edges.

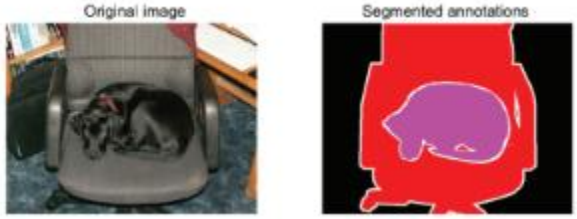


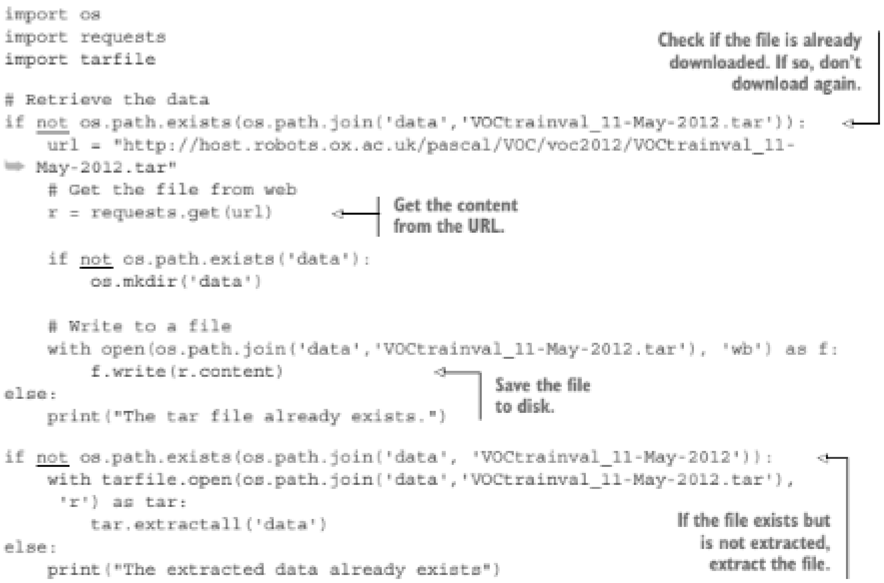
Figure 8.4
An original input image and its corresponding annotated/segmented image

Downloading the Dataset

The dataset is distributed as a compressed tar file. The process includes:

1. Checking whether the dataset has already been downloaded.
2. Downloading the dataset if it does not exist.
3. Saving the file to disk.
4. Extracting the archive if it has not yet been unpacked.

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.

Loading Input Images

Input images are standard JPEG images and can be loaded using the Pillow (PIL) library. Image properties such as format and shape can be inspected after loading.

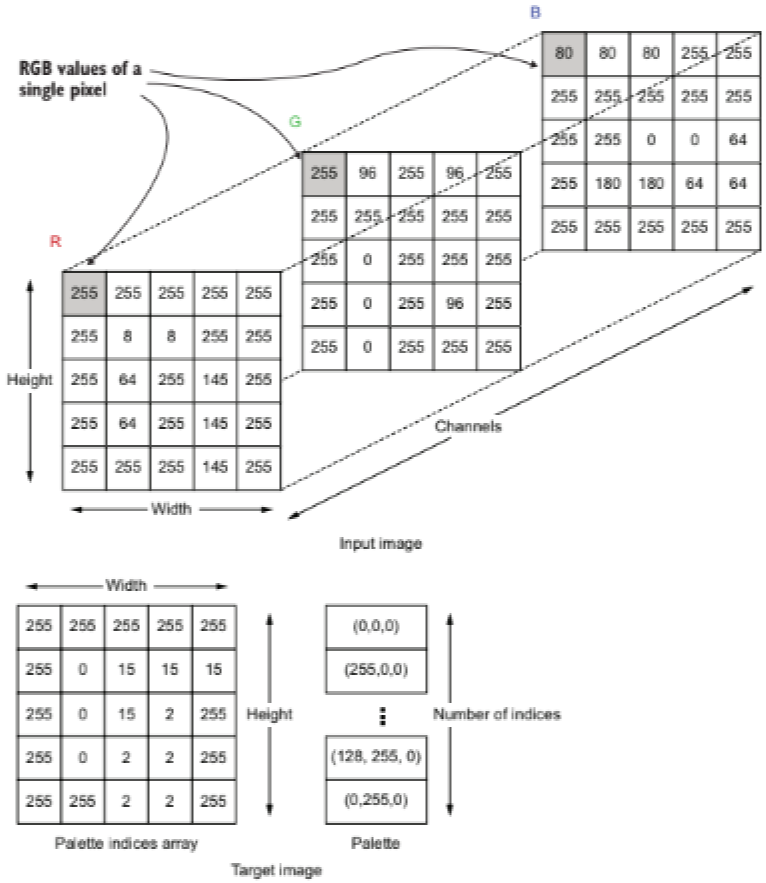
Loading Target (Annotated) Images

Target images require special handling because they are stored as **palettized images** rather than standard RGB images.

Palettization is a memory-efficient image storage technique:

- A **palette** stores all possible colors as a one-dimensional array.
- The image itself stores indices referencing entries in the palette.
- Each pixel value corresponds to a color in the palette.

This approach is suitable for segmentation tasks, where the number of colors (classes) is fixed.



255	255	255	255	255
255	8	8	255	255
255	64	255	145	255
255	64	255	145	255
255	255	255	145	255

255	96	255	96	255
255	255	255	255	255
255	0	255	255	255
255	0	255	96	255
255	0	255	255	255

80	80	80	255	255
255	255	255	255	255
255	255	0	0	64
255	180	180	64	64
255	255	255	255	255

255	255	255	255	255
255	0	15	15	15
255	0	15	2	255
255	0	2	2	255
255	255	2	2	255

(0,0,0)
(255,0,0)
⋮
(128, 255, 0)
(0,255,0)

Figure 8.5

Numerical representation of input images and palettized target images in the PASCAL VOC 2012 dataset

---

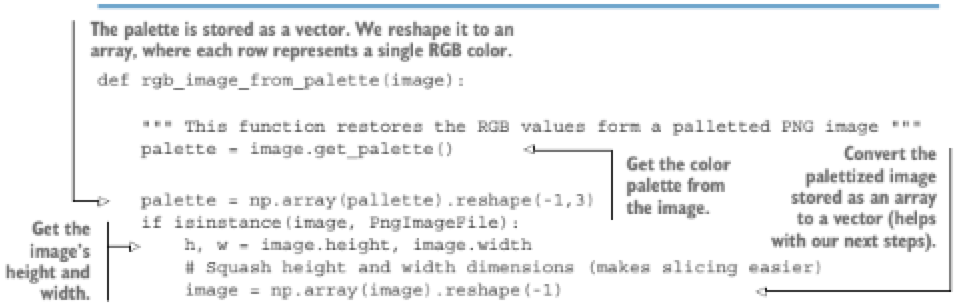
Reconstructing RGB Images from Palettized Images

To reconstruct the original RGB representation:

1. Extract the palette using get_palette().
2. Reshape the palette into a (number_of_classes, number_of_channels) array (e.g., (22, 3)).
3. Use the index values in the image to map each pixel to its corresponding RGB color.
4. Store the reconstructed colors in a new array.

Listing 8.2

Reconstructing the original image from a palettized image



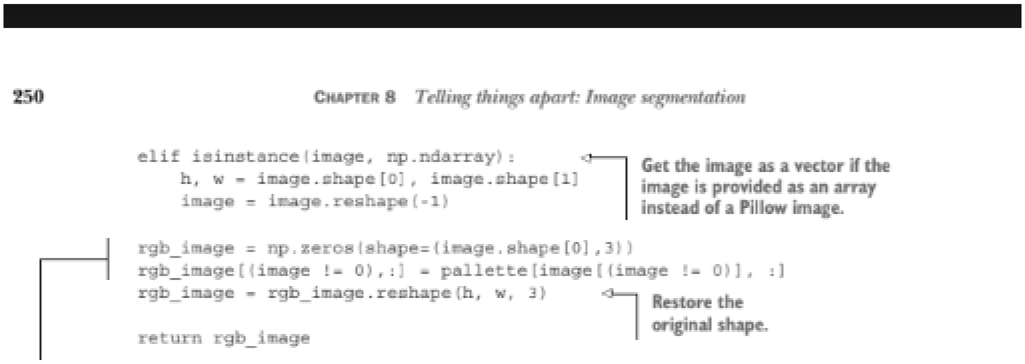
The palette is stored as a vector. We reshape it to an array, where each row represents a single RGB color.

```
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.



250 CHAPTER 8 *Telling things apart: Image segmentation*

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

    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
```

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

Restore the original shape.

Preparing for a TensorFlow Data Pipeline

After understanding how segmentation data is structured and loaded, the next step is to build a TensorFlow data pipeline that:

- Transforms images into a model-compatible format
- Efficiently handles both input images and palettized target images

This pipeline will serve as the foundation for training a segmentation model.

Exercise 1

You are given:

- rgb_image: an RGB image where each pixel belongs to one of *n* distinct colors
- palette: a (n, 3) array containing RGB color values

Task
Convert the rgb_image into a palettized image representation.

Hint
A naïve solution can be implemented using:

- Two nested loops to iterate over image pixels
- A third loop to match pixel values against the palette

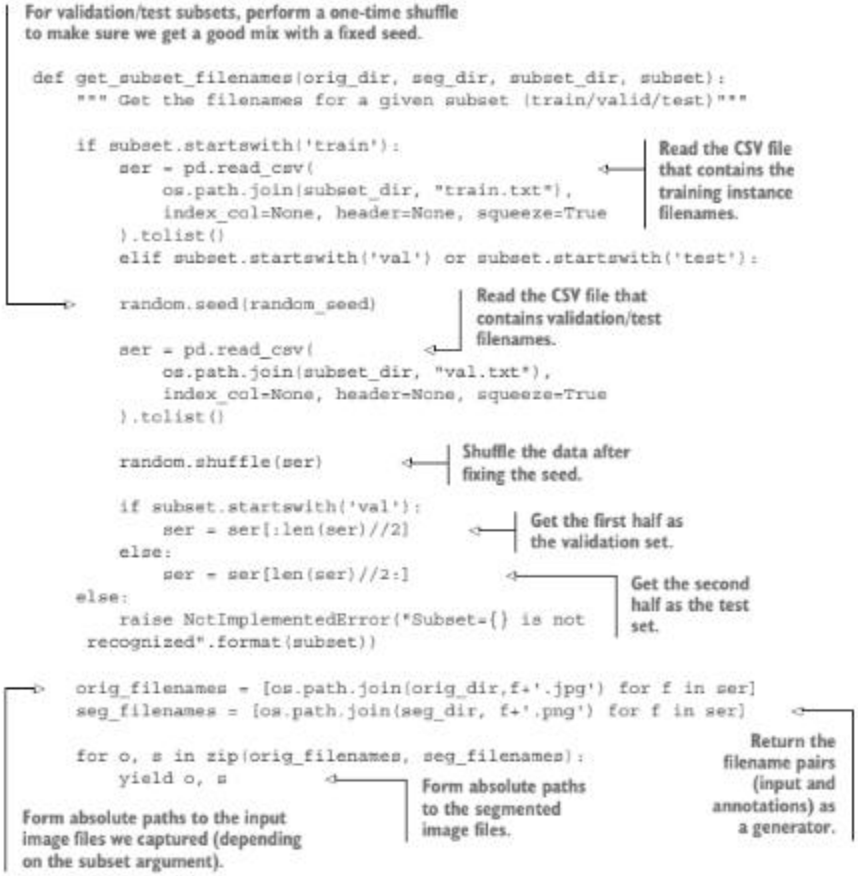
8.2 Getting Serious: Defining a TensorFlow Data Pipeline

A scalable **tf.data pipeline** is introduced to handle large datasets efficiently. The pipeline is designed to retrieve filenames, load images from disk, preprocess and augment data, batch samples, and optimize data throughput.

The pipeline architecture ensures flexibility for training, validation, and testing while enabling efficient scaling for production.

Filename Generation and Dataset Creation

A generator-based approach is used to retrieve filenames corresponding to specific dataset subsets (training, validation, testing). This design simplifies pipeline construction and allows seamless integration with TensorFlow.



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

```
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
```

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.

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

Form absolute paths to the segmented image files.

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

Listing 8.3 Retrieving the filenames for a given subset of data Insert after describing generator-based filename retrieval.

The generator output is consumed using tf.data.Dataset.from_generator, which requires explicit specification of output data types.

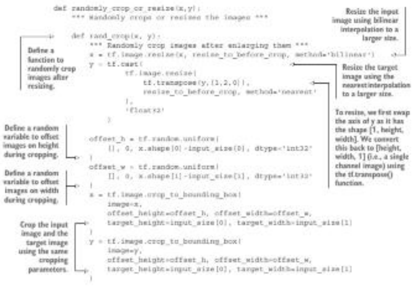
Image Loading and Custom Operations

Input images are loaded using TensorFlow’s native image decoding utilities. However, target segmentation images require a custom NumPy-based loading function to preserve palette indices. This function is wrapped using tf.numpy_function to integrate it into the TensorFlow graph.

A caution is emphasized regarding performance degradation when overusing NumPy operations inside TensorFlow pipelines.

Normalization and Shape Consistency

Input images are normalized to the [0, 1] range, while target images remain unchanged. Attempting to batch variable-sized images leads to runtime errors, highlighting the necessity of resizing or cropping images to a fixed size.



```
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.

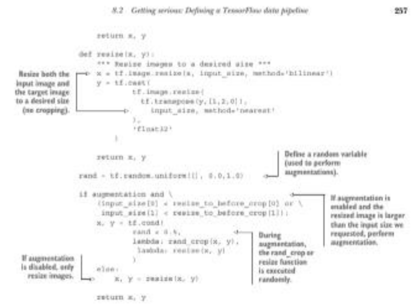
Resize the target image using the nearest interpolation to a larger size.

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.



8.2 Getting serious: Defining a TensorFlow data pipeline **257**

```
        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.

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

Place after explaining why variable image sizes cause batching errors.When resizing, **bilinear interpolation** is applied to input images, and **nearest-neighbor interpolation** is applied to target masks to avoid corrupting class labels.

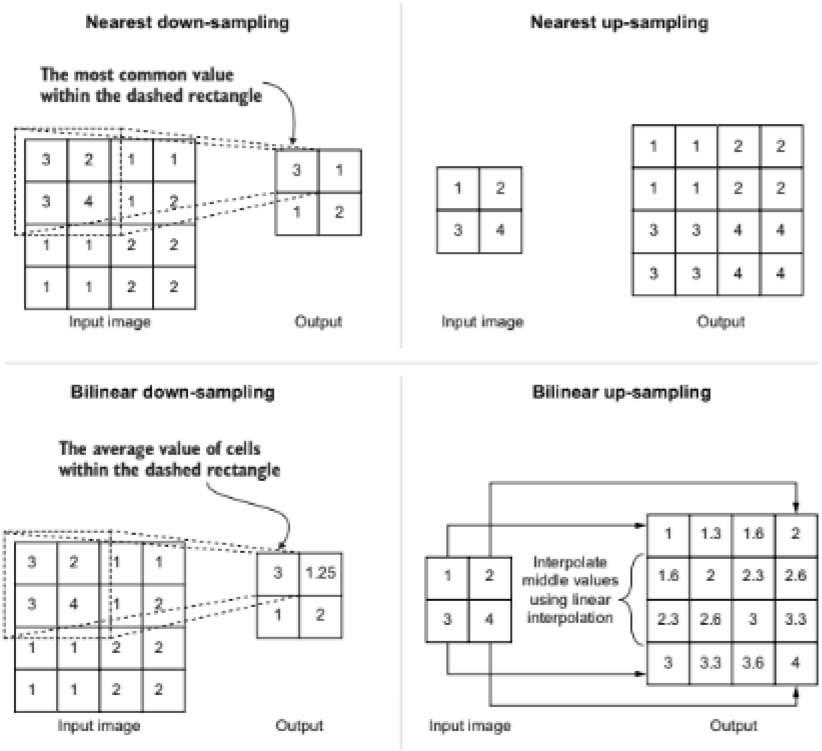
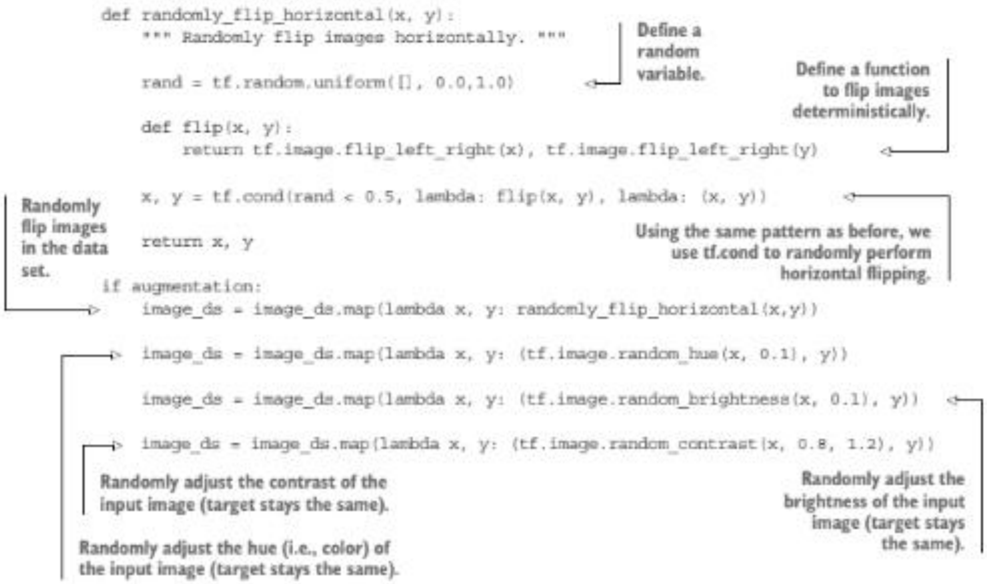


Figure 8.6 Nearest interpolation and bilinear interpolation
Insert immediately after discussing interpolation strategies.

Data Augmentation and Shape Fixing

Random cropping, flipping, brightness adjustment, contrast adjustment, and hue changes are applied to improve generalization. Augmentations are applied conditionally using tf.cond.

Because TensorFlow may lose shape information during preprocessing, tensor shapes are explicitly set to avoid ambiguity.



```
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.

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

Randomly flip images in the data set.

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

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

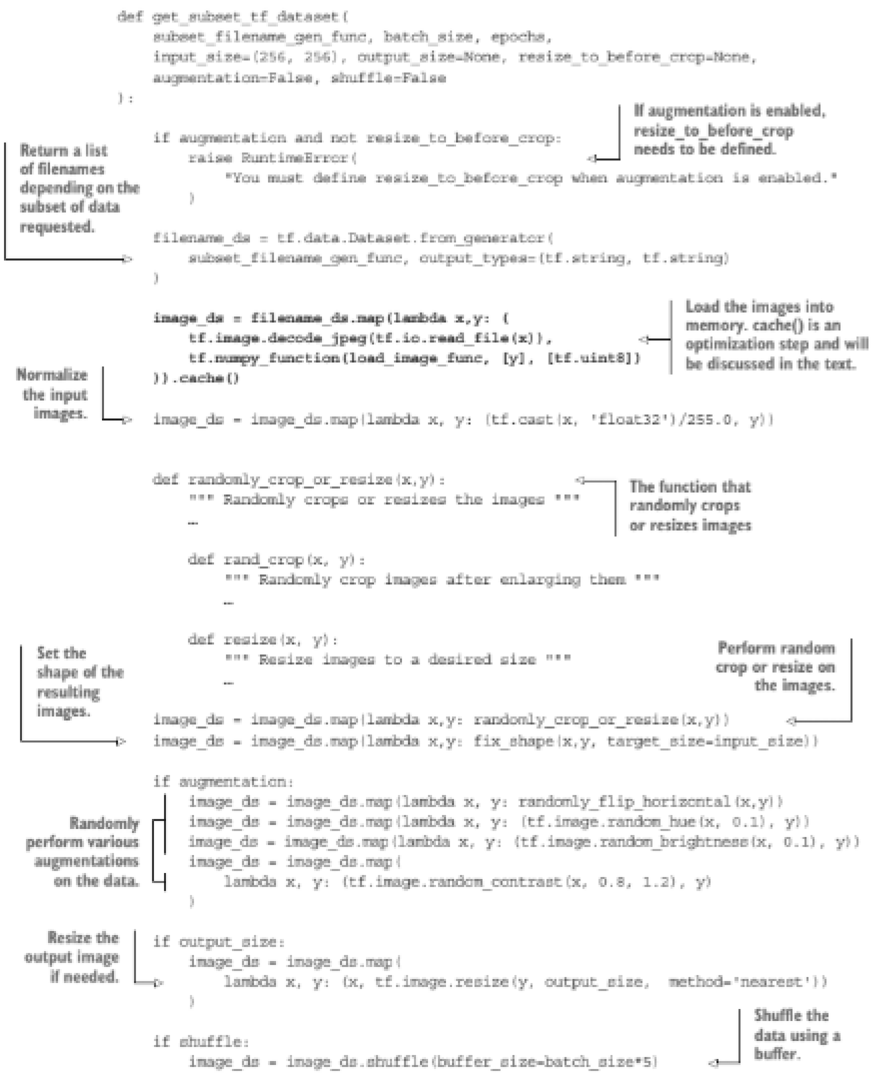
Randomly adjust the hue (i.e., color) of the input image (target stays the same).

Listing 8.5 Functions used for random augmentation of images
Place after introducing augmentation strategies.

Final Pipeline Assembly and Optimization

The pipeline concludes by shuffling, batching, repeating for multiple epochs, and removing unnecessary singleton dimensions from targets. Two key optimizations are introduced:

- **Caching**, which prevents repeated disk reads
- **Prefetching**, which overlaps data loading with model execution



```
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)
```

If augmentation is enabled, resize_to_before_crop needs to be defined.

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

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

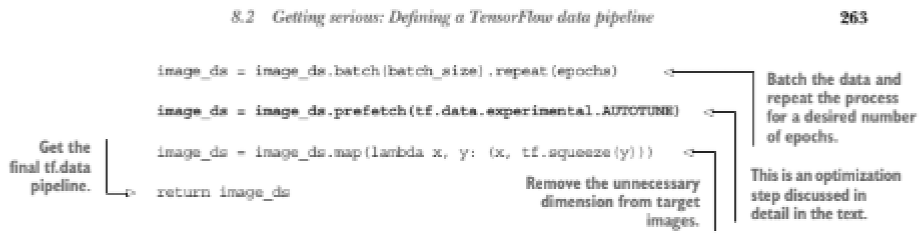
Perform random crop or resize on the images.

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.

Get the final tf.data pipeline.

Remove the unnecessary dimension from target images.

// Listing 8.6 The final tf.data pipeline //

Insert after describing the full pipeline assembly.

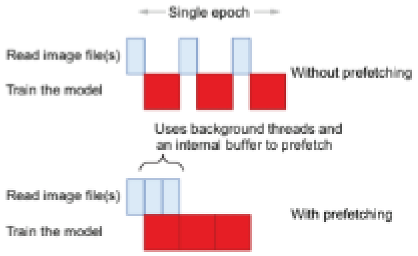
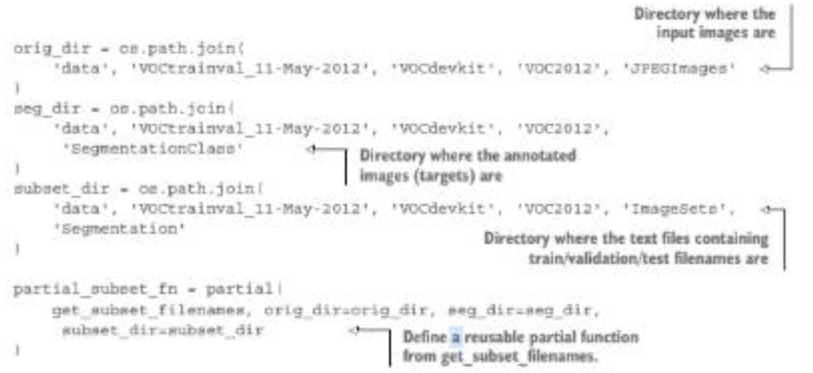


Figure 8.7 Sequential execution versus pre-fetching-based execution in model training

---

8.2.2 Final Pipeline Definitions

Three separate pipelines are constructed for training, validation, and testing. Training pipelines include augmentation and shuffling, while validation and test pipelines do not.



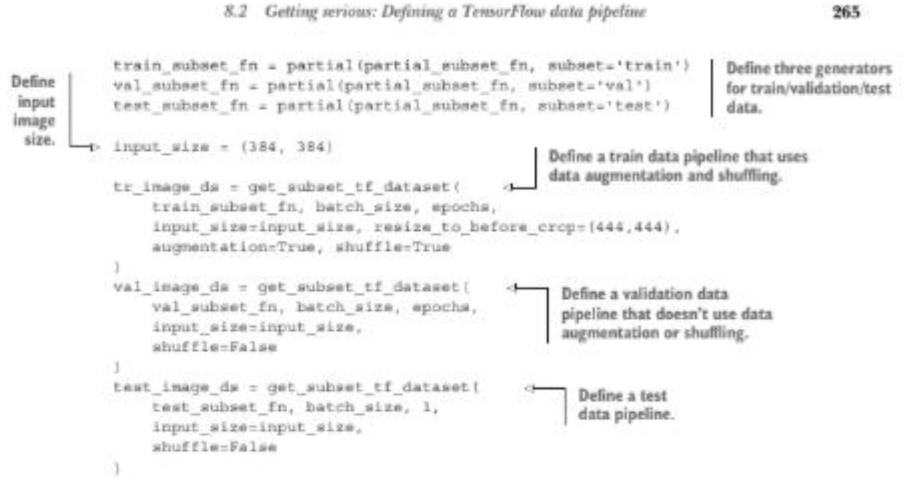
```
orig_dir = os.path.join(
    'data', 'VOCtrainval_11-May-2012', 'VOCdevkit', 'VOC2012', 'JPEGImages'
)
seg_dir = os.path.join(
    'data', 'VOCtrainval_11-May-2012', 'VOCdevkit', 'VOC2012',
     'SegmentationClass'
)
subset_dir = os.path.join(
    'data', 'VOCtrainval_11-May-2012', 'VOCdevkit', 'VOC2012',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



8.2 Getting serious: Defining a TensorFlow data pipeline **265**

```
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 three generators for train/validation/test data.

Define input image size.

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.

Listing 8.7 Creating the train/validation/test data pipelines instances
Insert after describing the separation of pipelines.

8.3 DeepLab v3: Using Pretrained Networks for Segmentation

DeepLab v3 is introduced as the segmentation model of choice, leveraging a pretrained **ResNet-50 backbone** combined with **atrous convolutions** and an **Atrous Spatial Pyramid Pooling (ASPP)** module. This design mitigates the resolution loss caused by pooling and striding while maintaining a large receptive field.

Atrous Convolution and Pyramidal Aggregation

Atrous convolution increases the receptive field without increasing the number of parameters by inserting holes between kernel elements. This enables dense predictions while preserving spatial resolution.

Figure placement:

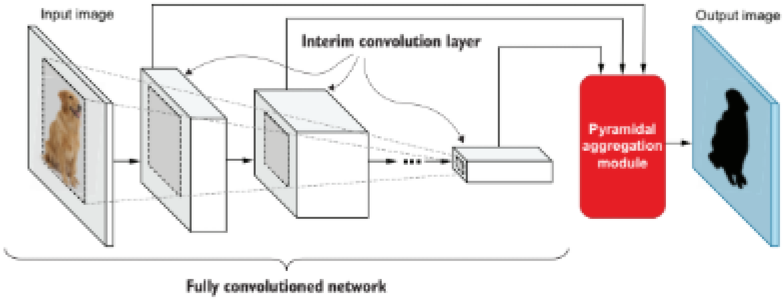

Input image
Interim convolution layer
Output image
Pyramidal aggregation module
Fully convolutioned network

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

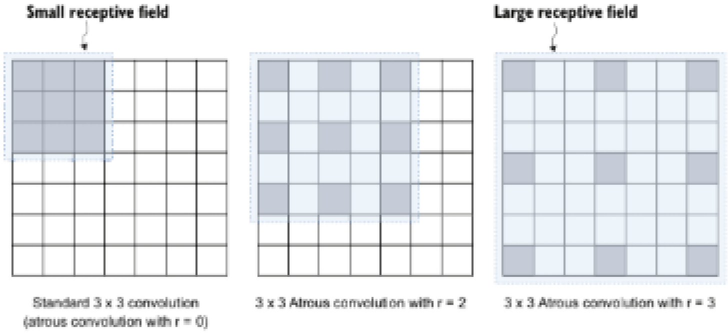

Small receptive field
Large receptive field
Standard 3 x 3 convolution
(atrous convolution with r = 0)
3 x 3 Atrous convolution with r = 2
3 x 3 Atrous convolution with r = 3

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.Both figures should appear immediately after explaining pyramidal aggregation and atrous convolution, respectively.

Implementing DeepLab v3

The ResNet-50 model is truncated at the conv4 block. The final conv5 block is reimplemented using atrous convolutions. The architecture is rebuilt using the **Keras Functional API**.

Figure placement:

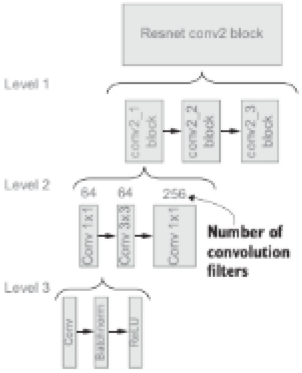
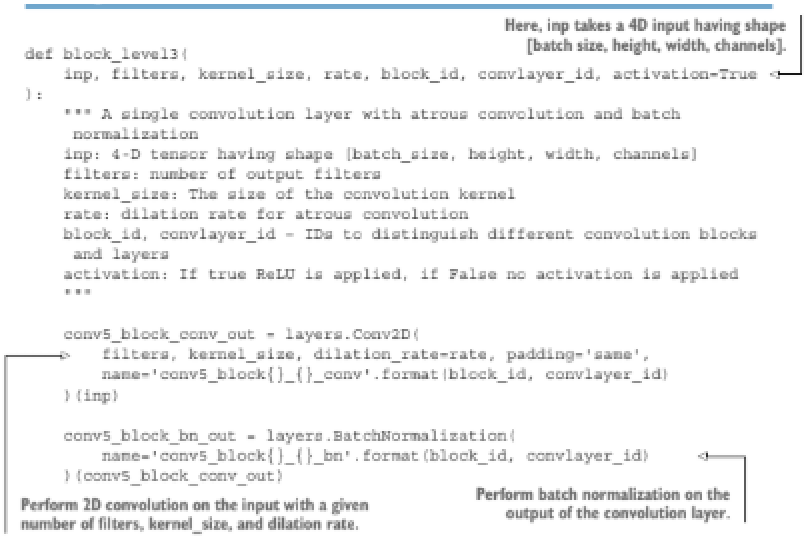


Figure 8.10 Anatomy of a convolution block in ResNet-50. For this example, we show the very first convolution block of ResNet-50. The organization of a convolution block group consists of three different levels.

Listing placements:

- Listing 8.8 A level 3 convolution block in ResNet-50

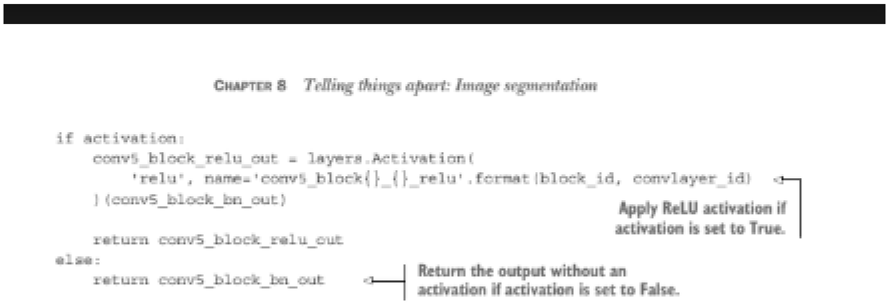


```
def block_level3(
    inp, filters, kernel_size, rate, block_id, convlayer_id, activation=True
):
    """ A single convolution layer with atrous convolution and batch
     normalization
    inp: 4-D tensor having shape [batch_size, height, width, channels]
    filters: number of output filters
    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)
```

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

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.

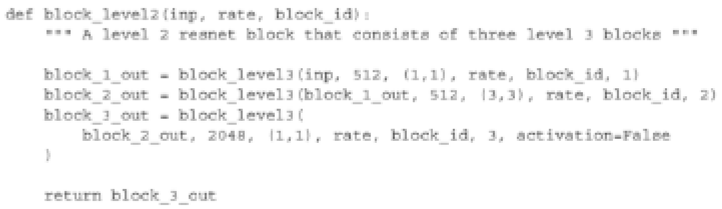


```
if activation:
    conv5_block_relu_out = layers.Activation(
        'relu', name='conv5_block{}_{}_relu'.format(block_id, convlayer_id)
    )(conv5_block_bn_out)

    return conv5_block_relu_out
else:
    return conv5_block_bn_out
```

Apply ReLU activation if activation is set to True.

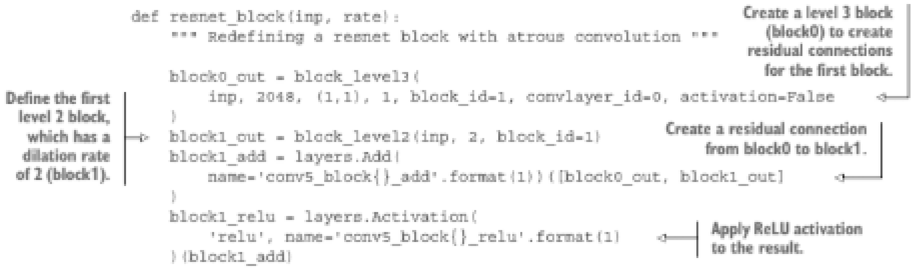
Return the output without an activation if activation is set to False.

- 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
```

- Listing 8.10 Implementing the final ResNet-50 convolution block group



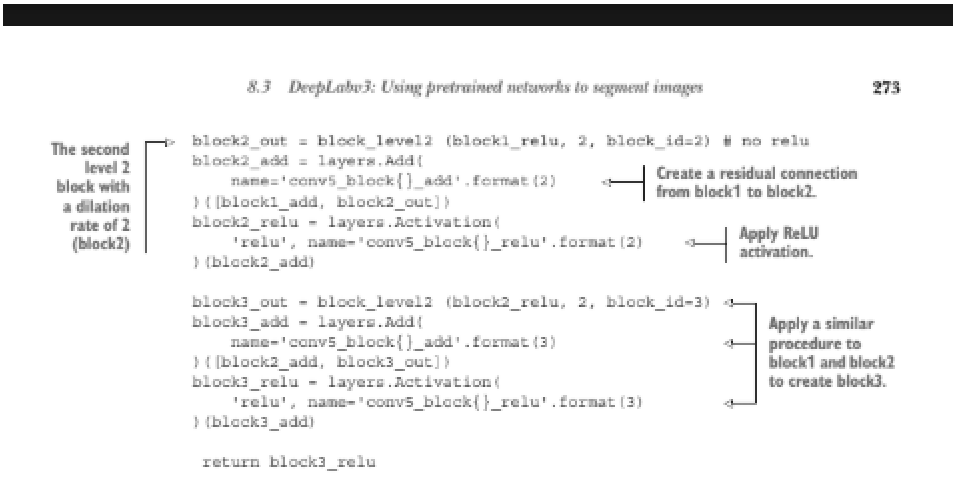
```
def resnet_block(inp, rate):
    """ Redefining a resnet block with atrous convolution """

    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.



8.3 DeepLabv3: Using pretrained networks to segment images 273

```
    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
```

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

Create a residual connection from block1 to block2.

Apply ReLU activation.

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

ASPP Module

The ASPP module aggregates multiscale contextual information using parallel atrous convolutions with different dilation rates, combined with global average pooling and bilinear upsampling.

Figure placement:

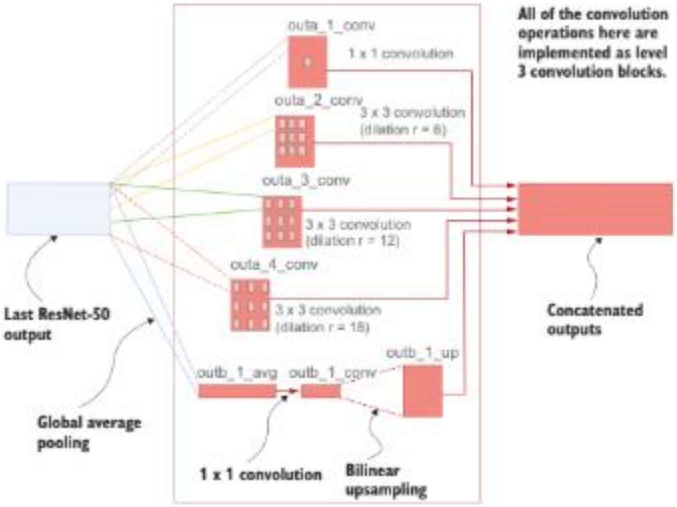
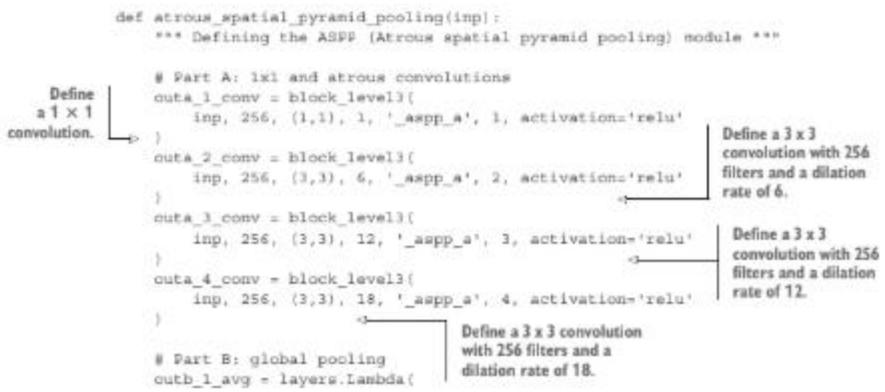


Figure 8.11 The ASPP module used in the DeepLab v3 model



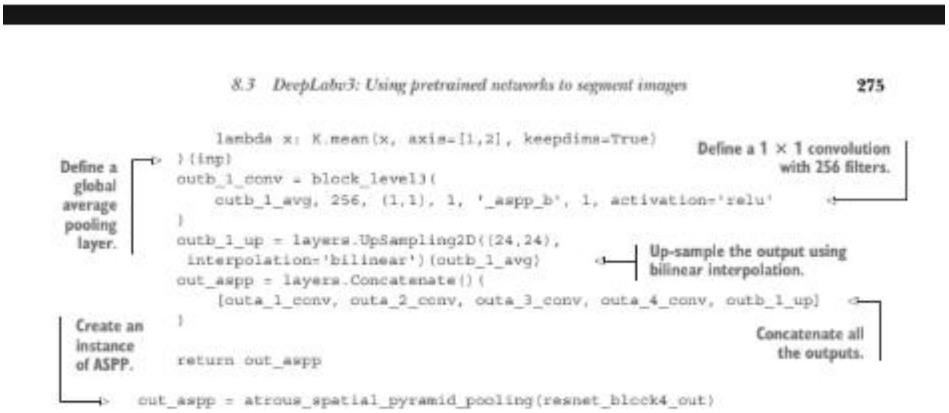
```
def atrous_spatial_pyramid_pooling(inp):
    """ Defining the ASPP (Atrous spatial pyramid pooling) module """

    # 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.



8.3 DeepLabv3: Using pretrained networks to segment images 275

```
        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 1 × 1 convolution with 256 filters.

Define a global average pooling layer.

Up-sample the output using bilinear interpolation.

Concatenate all the outputs.

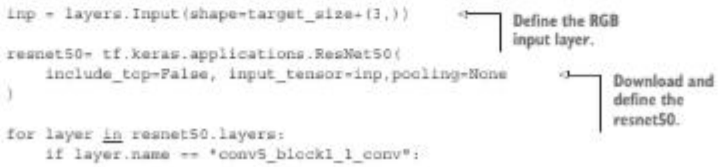
Create an instance of ASPP.

Listing 8.11 Implementing ASPP

Final Model Assembly

All components are integrated into the final DeepLab v3 model. The output layer produces logits without activation, as specialized loss functions operating on logits are used.

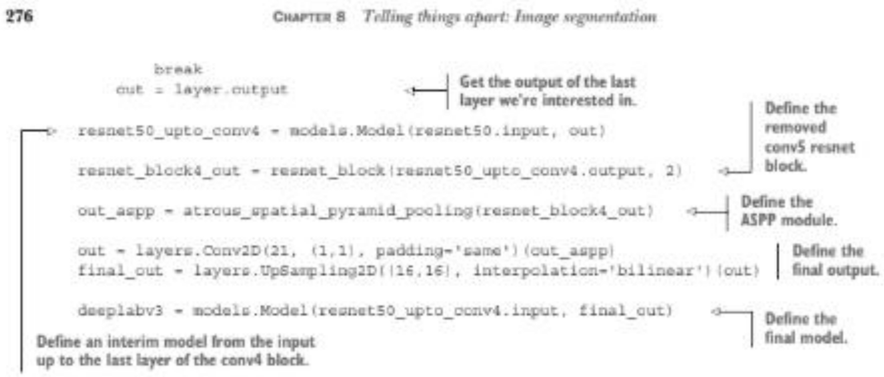
Listing placement:



```
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.



276 CHAPTER 8 *Telling things apart: Image segmentation*

```
        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.

Listing 8.12 The final DeepLab v3 model

8.4 Compiling the Model

Segmentation requires specialized loss functions and metrics due to pixel-wise prediction and severe class imbalance. Weighted categorical cross-entropy and Dice loss are identified as suitable loss functions, while pixel accuracy, mean accuracy, and mean IoU are introduced as evaluation metrics.

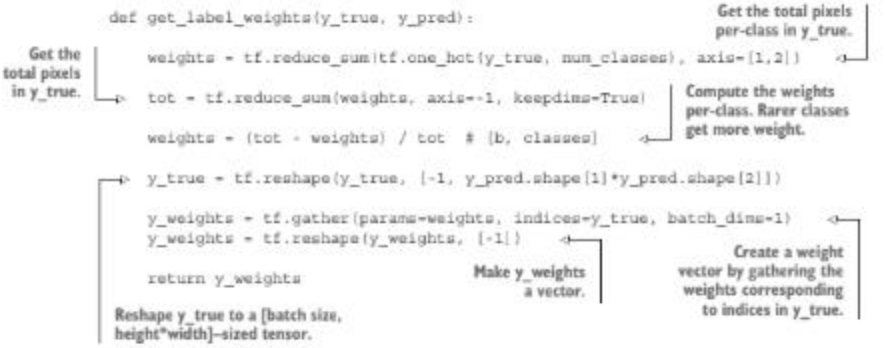
8.4.1 Loss Functions

Loss functions guide the optimization of deep learning models by defining an objective that must be minimized. In image segmentation, loss functions must be **differentiable**, as optimization relies on gradient-based methods. This chapter combines two complementary loss functions—**cross-entropy loss** and **dice loss**—to address both pixel-wise classification accuracy and spatial overlap quality.

Cross-Entropy Loss

Cross-entropy loss is one of the most widely used loss functions in segmentation tasks. It operates on predicted outputs and ground-truth labels shaped as [batch size, height, width, number of classes], where the class dimension represents a one-hot encoded vector for each pixel. The loss is computed independently for every pixel and then summed across the entire image.

Although effective, cross-entropy loss suffers from a major drawback in segmentation problems: **class imbalance**. Objects rarely occupy equal areas in an image, which causes dominant classes (such as background) to overwhelm rare classes. To mitigate this, **class-dependent pixel weights** are introduced so that pixels belonging to smaller objects contribute more to the loss.



```
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.

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

Make y_weights a vector.

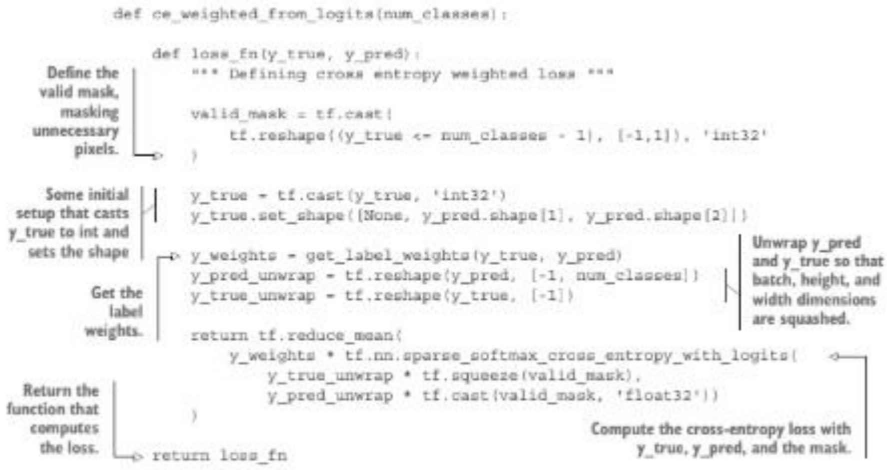
Listing 8.13 Computing the label weights for a given batch of data

This listing should be placed immediately after introducing class imbalance and weighted loss.

The weighting strategy computes the total number of pixels per class and assigns higher weights to rarer classes. The final weight tensor is constructed by gathering class-specific weights using the label indices in y_true, ensuring that each pixel contributes appropriately during loss computation.

Weighted Cross-Entropy Loss Implementation

With class weights defined, the weighted cross-entropy loss is implemented by masking irrelevant pixels (such as unknown object classes), flattening spatial dimensions, and computing sparse cross-entropy directly from logits.



```
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.

Return the function that computes the loss.

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

Listing 8.14 Implementing the weighted cross-entropy loss

The loss function is implemented as a **nested function** to allow passing additional parameters such as num_classes. Sparse cross-entropy is used instead of standard cross-entropy to avoid explicit one-hot encoding, making the data pipeline more memory efficient. Additionally, computing loss directly from logits improves numerical stability and gradient quality.

Dice Loss

Dice loss complements cross-entropy by directly optimizing for spatial overlap between predicted and ground-truth segmentation masks. It is derived from the Dice coefficient, which measures similarity using **intersection over union–like behavior**, but implemented in a fully differentiable manner.

Intersection is computed using element-wise multiplication, while union is computed using element-wise addition. This formulation satisfies differentiability requirements while preserving the geometric intuition behind overlap-based evaluation.

Figure placement:

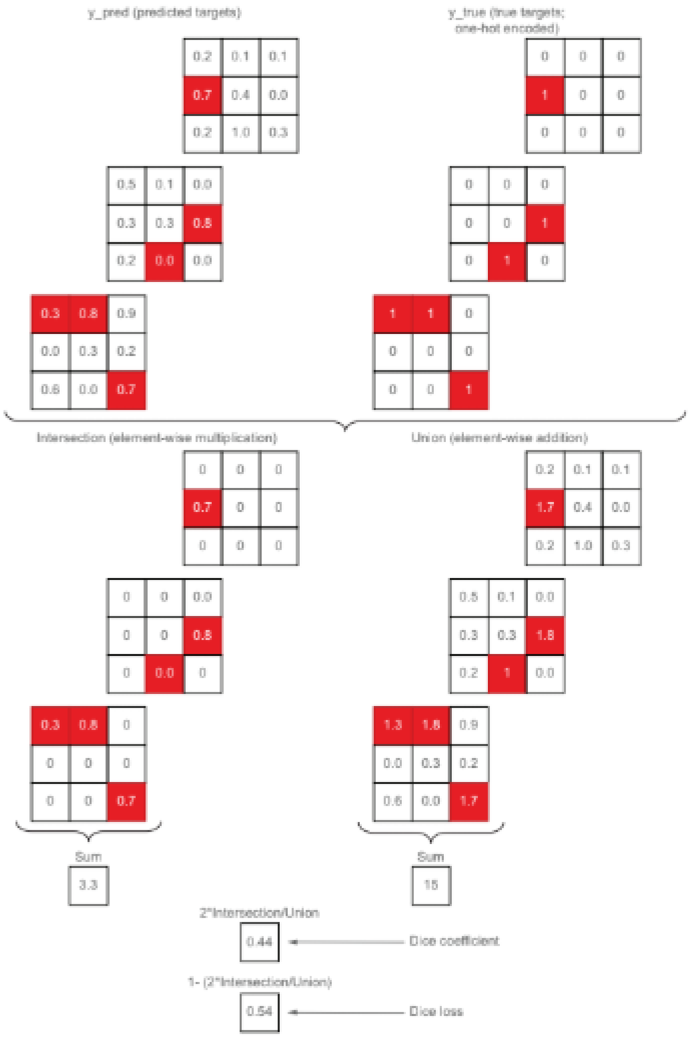
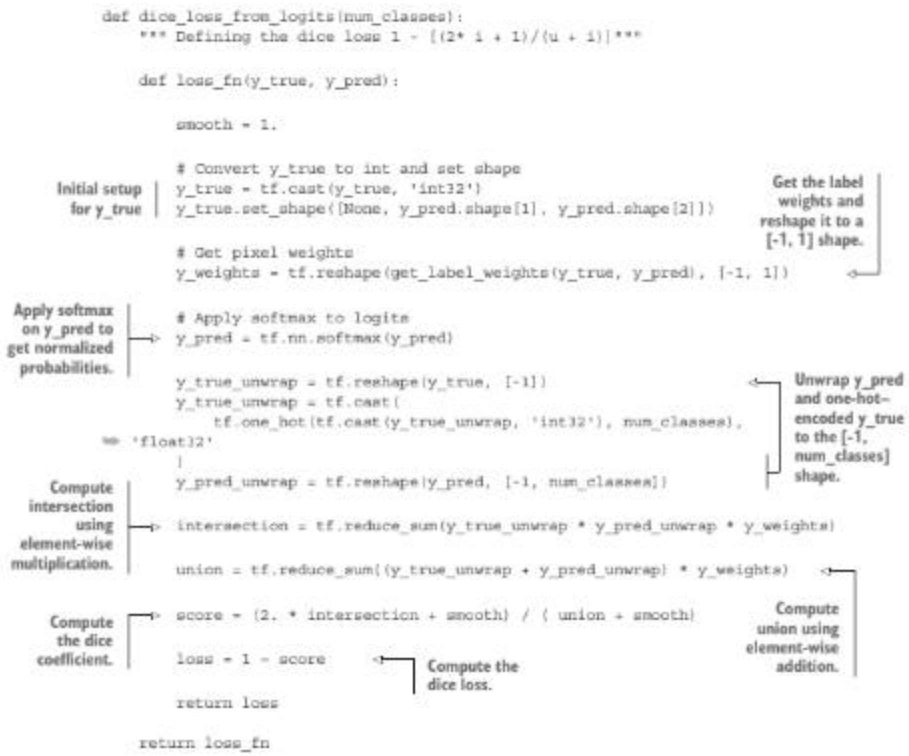


Figure 8.12 Computations involved in dice loss. The intersection can be computed as a differentiable function by taking element-wise multiplication, whereas union can be computed as the element-wise sum.



```
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.

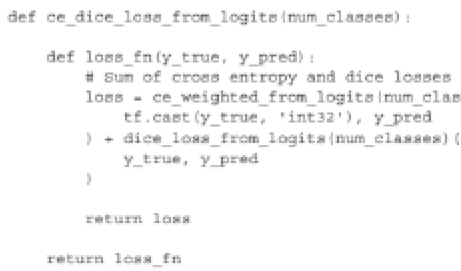
Listing 8.15 Implementing the dice loss

The implementation includes a smoothing term to avoid division by zero and applies softmax to predictions before computing overlap. Class weights are reused to ensure consistency with weighted cross-entropy.

Final Combined Loss Function

The segmentation model is optimized using a **combined loss**, obtained by summing weighted sparse cross-entropy loss and dice loss. This hybrid approach balances pixel-wise classification accuracy with spatial consistency.

Listing placement:



```
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
```

Listing 8.16 Final combined loss function

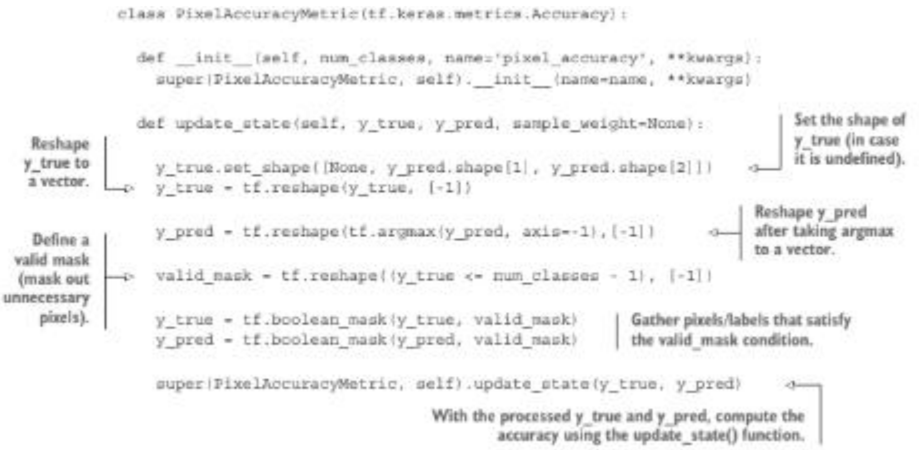
8.4.2 Evaluation Metrics

Evaluation metrics serve as a diagnostic tool during training, ensuring that learning progresses meaningfully. Unlike loss functions, metrics in TensorFlow are **stateful**, meaning they accumulate statistics across batches within an epoch.

Custom metrics are implemented by subclassing tf.keras.metrics.Metric or existing metric classes and defining four core methods: __init__, update_state, result, and reset_states.

Pixel Accuracy

Pixel accuracy measures the proportion of correctly classified pixels across the entire image. While simple, it can be misleading in the presence of class imbalance.



```
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.

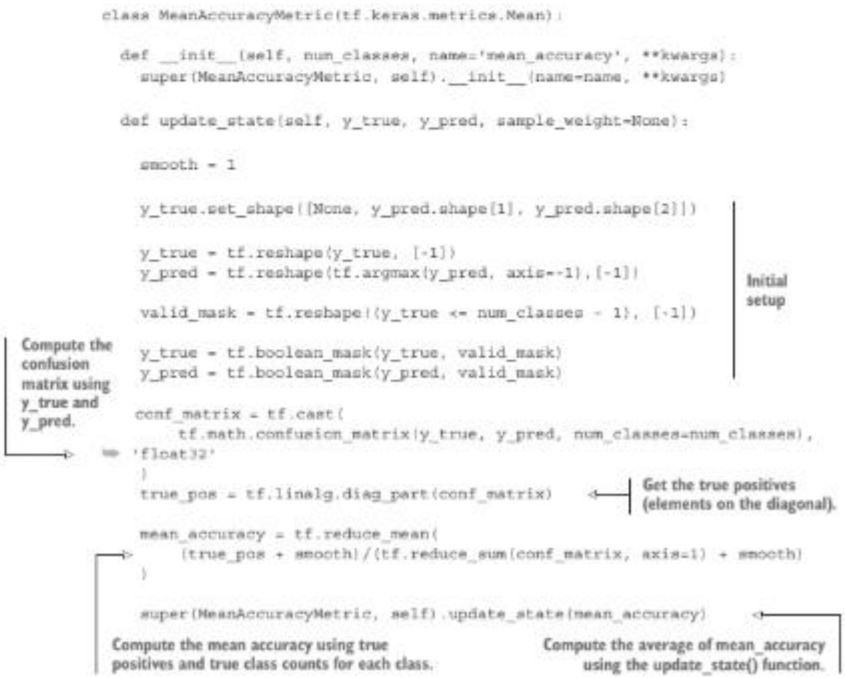
With the processed y_true and y_pred, compute the accuracy using the update_state() function.

Listing 8.17 Implementing the pixel accuracy metric

The metric reshapes predictions and labels into vectors, applies a validity mask to ignore unknown classes, and then delegates computation to TensorFlow’s built-in accuracy logic.

Mean Accuracy

To counteract class imbalance, **mean accuracy** computes accuracy separately for each class and then averages the results. This ensures that rare classes contribute equally to the final score.



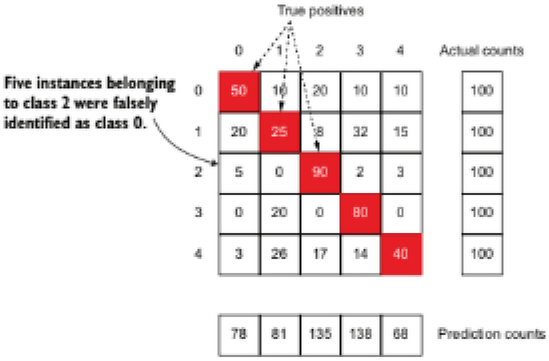
```
class MeanAccuracyMetric(tf.keras.metrics.Mean):

  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.

Listing 8.18 Implementing the mean accuracy metric

This metric relies on computing a **confusion matrix**, from which true positives and per-class counts are derived.

Figure placement:



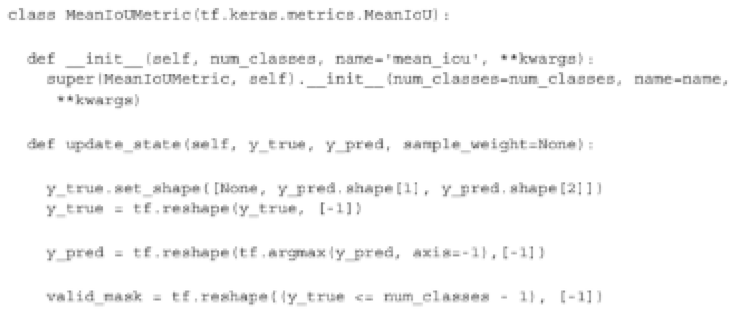
True positives
Five instances belonging to class 2 were falsely identified as class 0.

	0	1	2	3	4	Actual counts
0	50	10	20	10	10	100
1	20	25	8	32	15	100
2	5	0	90	2	3	100
3	0	20	0	80	0	100
4	3	26	17	14	40	100
Prediction counts	78	81	135	138	68	

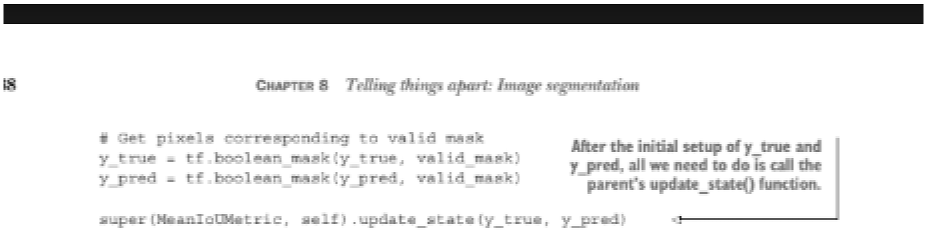
Figure 8.13 Illustration of a confusion matrix for a five-class classification problem. The shaded boxes represent true positives.

Mean Intersection over Union (Mean IoU)

Mean IoU is one of the most widely used segmentation metrics. It measures overlap quality by comparing true positives against the sum of true positives, false positives, and false negatives.



```
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])
```



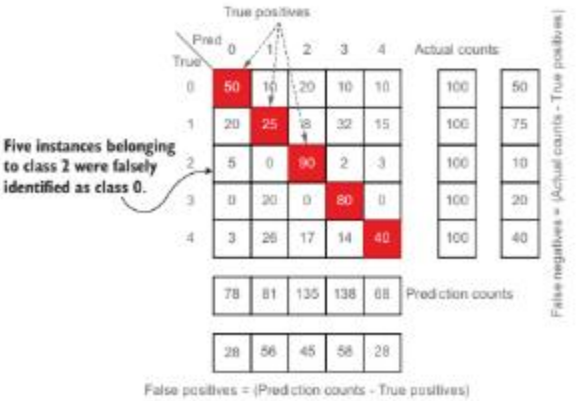
```
    # Get pixels corresponding to valid mask
    y_true = tf.boolean_mask(y_true, valid_mask)
    y_pred = tf.boolean_mask(y_pred, valid_mask)

    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.

Listing 8.19 Implementing the mean IoU metric

The metric leverages TensorFlow's built-in MeanIoU class and requires minimal customization beyond preprocessing predictions and labels.

Figure placement:



True positives
Pred
True
Actual counts
Five instances belonging to class 2 were falsely identified as class 0.
Prediction counts
False negatives = (Actual counts - True positives)
False positives = (Prediction counts - True positives)

True \ Pred	0	1	2	3	4	Actual counts	
0	50	10	20	10	10	100	50
1	20	25	8	32	15	100	75
2	5	0	90	2	3	100	10
3	0	20	0	80	0	100	20
4	3	26	17	14	40	100	40
Prediction counts	78	81	135	138	68		
	28	56	45	58	28		

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

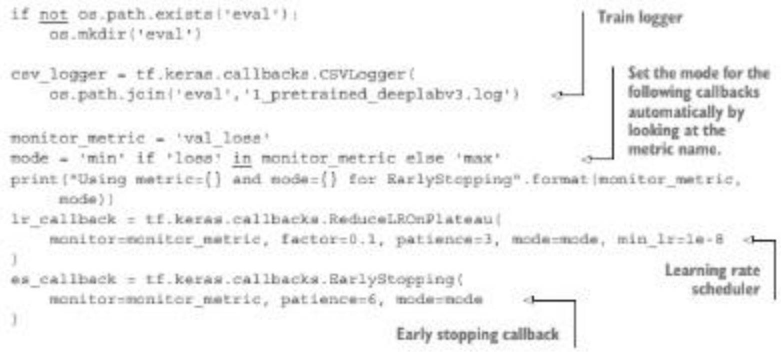
Model Compilation

With losses and metrics defined, the DeepLab v3 model is compiled using the combined loss function and all three evaluation metrics.

The pretrained weights removed earlier are restored into the new model layers before training, ensuring continuity in learned representations.

8.5 Training the Model

Training the model involves calling fit() with the prepared data pipelines. The model is trained for 25 epochs while monitoring pixel accuracy, mean accuracy, and mean IoU on the validation set.



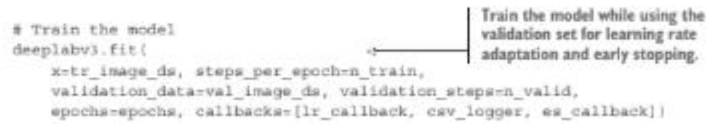
```
if not os.path.exists('eval'):
    os.mkdir('eval')

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(
    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])
```

Train the model while using the validation set for learning rate adaptation and early stopping.

/Listing 8.20 Training the model

Three callbacks are used: CSV logging, learning-rate reduction on plateau, and early stopping. On the referenced hardware, training completes in approximately 45 minutes.

8.6 Evaluating the Model

After training, the model is evaluated on an unseen test set using the same metrics. The reported results demonstrate strong performance given the limited dataset size, achieving approximately **62% mean IoU**, **87% mean accuracy**, and **91% pixel accuracy**.

Beyond numerical metrics, qualitative evaluation is emphasized by visualizing predicted segmentation maps alongside ground-truth annotations.

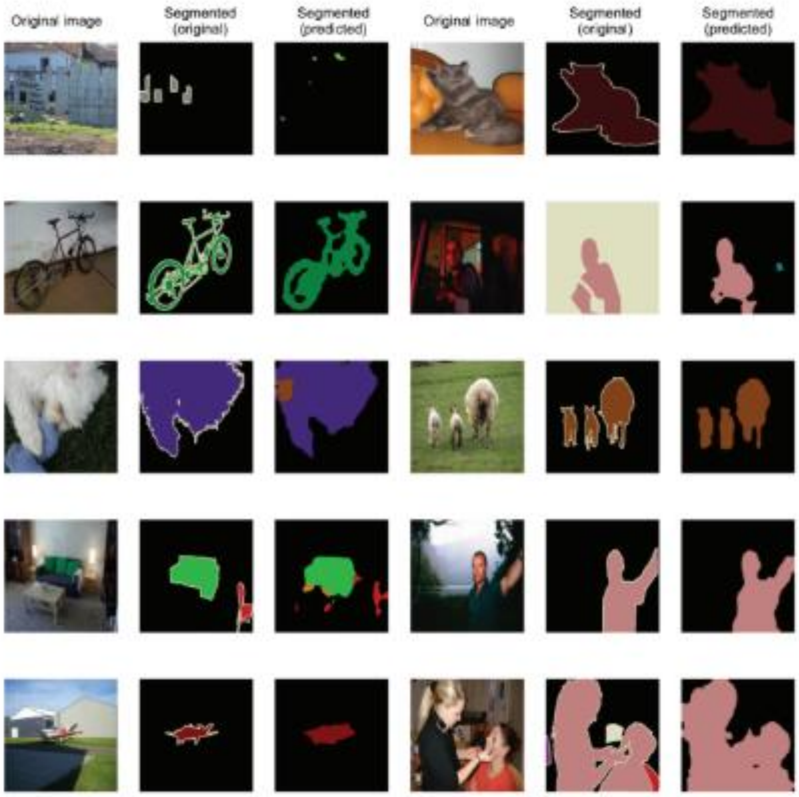


Figure 8.15 Comparing the true annotated targets to model predictions. You can see that the model is quite

good at separating objects from different backgrounds.The visual results confirm that the model effectively separates objects from backgrounds, even under challenging conditions.

Chapter Summary

This chapter presented a complete workflow for semantic image segmentation, from data handling and pipeline construction to model definition, loss design, training, and evaluation. DeepLab v3 demonstrated strong performance using a pretrained backbone, atrous convolutions, and task-specific losses and metrics.

The chapter concludes by positioning segmentation as a foundational technique for advanced computer vision applications and transitions toward natural language processing topics in subsequent chapters.