```
            skip_Z = layer(skip_Z)
        return self.activation(Z + skip_Z)
```

As you can see, this code matches Figure 14-18 pretty closely. In the constructor, we create all the layers we will need: the main layers are the ones on the right side of the diagram, and the skip layers are the ones on the left (only needed if the stride is greater than 1). Then in the `call()` method, we simply make the inputs go through the main layers, and the skip layers (if any), then we add both outputs and we apply the activation function.

Next, we can build the ResNet-34 simply using a `Sequential` model, since it is really just a long sequence of layers (we can treat each residual unit as a single layer now that we have the `ResidualUnit` class):

```
model = keras.models.Sequential()
model.add(DefaultConv2D(64, kernel_size=7, strides=2,
                        input_shape=[224, 224, 3]))
model.add(keras.layers.BatchNormalization())
model.add(keras.layers.Activation("relu"))
model.add(keras.layers.MaxPool2D(pool_size=3, strides=2, padding="SAME"))
prev_filters = 64
for filters in [64] * 3 + [128] * 4 + [256] * 6 + [512] * 3:
    strides = 1 if filters == prev_filters else 2
    model.add(ResidualUnit(filters, strides=strides))
    prev_filters = filters
model.add(keras.layers.GlobalAvgPool2D())
model.add(keras.layers.Flatten())
model.add(keras.layers.Dense(10, activation="softmax"))
```

The only slightly tricky part in this code is the loop that adds the `ResidualUnit` layers to the model: as explained earlier, the first 3 RUs have 64 filters, then the next 4 RUs have 128 filters, and so on. We then set the strides to 1 when the number of filters is the same as in the previous RU, or else we set it to 2. Then we add the `ResidualUnit`, and finally we update `prev_filters`.

It is quite amazing that in less than 40 lines of code, we can build the model that won the ILSVRC 2015 challenge! It demonstrates both the elegance of the ResNet model, and the expressiveness of the Keras API. Implementing the other CNN architectures is not much harder. However, Keras comes with several of these architectures built in, so why not use them instead?

Using Pretrained Models From Keras

In general, you won't have to implement standard models like GoogLeNet or ResNet manually, since pretrained networks are readily available with a single line of code, in the `keras.applications` package. For example:

```
model = keras.applications.resnet50.ResNet50(weights="imagenet")
```

That's all! This will create a ResNet-50 model and download weights pretrained on the ImageNet dataset. To use it, you first need to ensure that the images have the right size. A ResNet-50 model expects 224 × 224 images (other models may expect other sizes, such as 299 × 299), so let's use TensorFlow's `tf.image.resize()` function to resize the images we loaded earlier:

```
images_resized = tf.image.resize(images, [224, 224])
```

The `tf.image.resize()` will not preserve the aspect ratio. If this is a problem, you can try cropping the images to the appropriate aspect ratio before resizing. Both operations can be done in one shot with `tf.image.crop_and_resize()`.

The pretrained models assume that the images are preprocessed in a specific way. In some cases they may expect the inputs to be scaled from 0 to 1, or -1 to 1, and so on. Each model provides a `preprocess_input()` function that you can use to preprocess your images. These functions assume that the pixel values range from 0 to 255, so we must multiply them by 255 (since earlier we scaled them to the 0–1 range):

```
inputs = keras.applications.resnet50.preprocess_input(images_resized * 255)
```

Now we can use the pretrained model to make predictions:

```
Y_proba = model.predict(inputs)
```

As usual, the output `Y_proba` is a matrix with one row per image and one column per class (in this case, there are 1,000 classes). If you want to display the top K predictions, including the class name and the estimated probability of each predicted class, you can use the `decode_predictions()` function. For each image, it returns an array containing the top K predictions, where each prediction is represented as an array containing the class identifier[21], its name and the corresponding confidence score:

```
top_K = keras.applications.resnet50.decode_predictions(Y_proba, top=3)
for image_index in range(len(images)):
    print("Image #{}".format(image_index))
    for class_id, name, y_proba in top_K[image_index]:
        print("  {} - {:12s} {:.2f}%".format(class_id, name, y_proba * 100))
    print()
```

The output looks like this:

```
Image #0
  n03877845 - palace       42.87%
  n02825657 - bell_cote    40.57%
  n03781244 - monastery    14.56%
```

21 In the ImageNet dataset, each image is associated to a word in the WordNet dataset: the class ID is just a WordNet ID.

```
Image #1
  n04522168 - vase          46.83%
  n07930864 - cup           7.78%
  n11939491 - daisy         4.87%
```

The correct classes (monastery and daisy) appear in the top 3 results for both images. That's pretty good considering that the model had to choose among 1,000 classes.

As you can see, it is very easy to create a pretty good image classifier using a pretrained model. Other vision models are available in `keras.applications`, including several ResNet variants, GoogLeNet variants like InceptionV3 and Xception, VGGNet variants, MobileNet and MobileNetV2 (lightweight models for use in mobile applications), and more.

But what if you want to use an image classifier for classes of images that are not part of ImageNet? In that case, you may still benefit from the pretrained models to perform transfer learning.

Pretrained Models for Transfer Learning

If you want to build an image classifier, but you do not have enough training data, then it is often a good idea to reuse the lower layers of a pretrained model, as we discussed in Chapter 11. For example, let's train a model to classify pictures of flowers, reusing a pretrained Xception model. First, let's load the dataset using TensorFlow Datasets (see Chapter 13):

```
import tensorflow_datasets as tfds

dataset, info = tfds.load("tf_flowers", as_supervised=True, with_info=True)
dataset_size = info.splits["train"].num_examples # 3670
class_names = info.features["label"].names # ["dandelion", "daisy", ...]
n_classes = info.features["label"].num_classes # 5
```

Note that you can get information about the dataset by setting `with_info=True`. Here, we get the dataset size and the names of the classes. Unfortunately, there is only a `"train"` dataset, no test set or validation set, so we need to split the training set. The TF Datasets project provides an API for this. For example, let's take the first 10% of the dataset for testing, the next 15% for validation, and the remaining 75% for training:

```
test_split, valid_split, train_split = tfds.Split.TRAIN.subsplit([10, 15, 75])

test_set = tfds.load("tf_flowers", split=test_split, as_supervised=True)
valid_set = tfds.load("tf_flowers", split=valid_split, as_supervised=True)
train_set = tfds.load("tf_flowers", split=train_split, as_supervised=True)
```

Next we must preprocess the images. The CNN expects 224 × 224 images, so we need to resize them. We also need to run the image through Xception's `preprocess_input()` function:

```
def preprocess(image, label):
    resized_image = tf.image.resize(image, [224, 224])
    final_image = keras.applications.xception.preprocess_input(resized_image)
    return final_image, label
```

Let's apply this preprocessing function to all 3 datasets, and let's also shuffle & repeat the training set, and add batching & prefetching to all datasets:

```
batch_size = 32
train_set = train_set.shuffle(1000).repeat()
train_set = train_set.map(preprocess).batch(batch_size).prefetch(1)
valid_set = valid_set.map(preprocess).batch(batch_size).prefetch(1)
test_set = test_set.map(preprocess).batch(batch_size).prefetch(1)
```

If you want to perform some data augmentation, you can just change the preprocessing function for the training set, adding some random transformations to the training images. For example, use `tf.image.random_crop()` to randomly crop the images, use `tf.image.random_flip_left_right()` to randomly flip the images horizontally, and so on (see the notebook for an example).

Next let's load an Xception model, pretrained on ImageNet. We exclude the top of the network (by setting `include_top=False`): this excludes the global average pooling layer and the dense output layer. We then add our own global average pooling layer, based on the output of the base model, followed by a dense output layer with 1 unit per class, using the softmax activation function. Finally, we create the Keras `Model`:

```
base_model = keras.applications.xception.Xception(weights="imagenet",
                                                  include_top=False)
avg = keras.layers.GlobalAveragePooling2D()(base_model.output)
output = keras.layers.Dense(n_classes, activation="softmax")(avg)
model = keras.models.Model(inputs=base_model.input, outputs=output)
```

As explained in Chapter 11, it's usually a good idea to freeze the weights of the pretrained layers, at least at the beginning of training:

```
for layer in base_model.layers:
    layer.trainable = False
```

Since our model uses the base model's layers directly, rather than the `base_model` object itself, setting `base_model.trainable=False` would have no effect.

Finally, we can compile the model and start training:

```
optimizer = keras.optimizers.SGD(lr=0.2, momentum=0.9, decay=0.01)
model.compile(loss="sparse_categorical_crossentropy", optimizer=optimizer,
              metrics=["accuracy"])
history = model.fit(train_set,
                    steps_per_epoch=int(0.75 * dataset_size / batch_size),
                    validation_data=valid_set,
                    validation_steps=int(0.15 * dataset_size / batch_size),
                    epochs=5)
```

This will be very slow, unless you have a GPU. If you do not, then you should run this chapter's notebook in Colab, using a GPU runtime (it's free!). See the instructions at *https://github.com/ageron/handson-ml2*.

After training the model for a few epochs, its validation accuracy should reach about 75-80%, and stop making much progress. This means that the top layers are now pretty well trained, so we are ready to unfreeze all layers (or you could try unfreezing just the top ones), and continue training (don't forget to compile the model when you freeze or unfreeze layers). This time we use a much lower learning rate to avoid damaging the pretrained weights:

```
for layer in base_model.layers:
    layer.trainable = True

optimizer = keras.optimizers.SGD(lr=0.01, momentum=0.9, decay=0.001)
model.compile(...)
history = model.fit(...)
```

It will take a while, but this model should reach around 95% accuracy on the test set. With that, you can start training amazing image classifiers! But there's more to computer vision than just classification. For example, what if you also want to know *where* the flower is in the picture? Let's look at this now.

Classification and Localization

Localizing an object in a picture can be expressed as a regression task, as discussed in Chapter 10: to predict a bounding box around the object, a common approach is to predict the horizontal and vertical coordinates of the object's center, as well as its height and width. This means we have 4 numbers to predict. It does not require much change to the model, we just need to add a second dense output layer with 4 units (typically on top of the global average pooling layer), and it can be trained using the MSE loss:

```
base_model = keras.applications.xception.Xception(weights="imagenet",
                                                  include_top=False)
avg = keras.layers.GlobalAveragePooling2D()(base_model.output)
class_output = keras.layers.Dense(n_classes, activation="softmax")(avg)
```

```
loc_output = keras.layers.Dense(4)(avg)
model = keras.models.Model(inputs=base_model.input,
                           outputs=[class_output, loc_output])
model.compile(loss=["sparse_categorical_crossentropy", "mse"],
              loss_weights=[0.8, 0.2], # depends on what you care most about
              optimizer=optimizer, metrics=["accuracy"])
```

But now we have a problem: the flowers dataset does not have bounding boxes around the flowers. So we need to add them ourselves. This is often one of the hardest and most costly part of a Machine Learning project: getting the labels. It's a good idea to spend time looking for the right tools. To annotate images with bounding boxes, you may want to use an open source image labeling tool like VGG Image Annotator, LabelImg, OpenLabeler or ImgLab, or perhaps a commercial tool like LabelBox or Supervisely. You may also want to consider crowdsourcing platforms such as Amazon Mechanical Turk or CrowdFlower if you have a very large number of images to annotate. However, it is quite a lot of work to setup a crowdsourcing platform, prepare the form to be sent to the workers, to supervise them and ensure the quality of the bounding boxes they produce is good, so make sure it is worth the effort: if there are just a few thousand images to label, and you don't plan to do this frequently, it may be preferable to do it yourself. Adriana Kovashka et al. wrote a very practical paper[22] about crowdsourcing in Computer Vision, I recommend you check it out, even if you do not plan to use crowdsourcing.

So let's suppose you obtained the bounding boxes for every image in the flowers dataset (for now we will assume there is a single bounding box per image), you then need to create a dataset whose items will be batches of preprocessed images along with their class labels and their bounding boxes. Each item should be a tuple of the form: `(images, (class_labels, bounding_boxes))`. Then you are ready to train your model!

The bounding boxes should be normalized so that the horizontal and vertical coordinates, as well as the height and width all range from 0 to 1. Also, it is common to predict the square root of the height and width rather than the height and width directly: this way, a 10 pixel error for a large bounding box will not be penalized as much as a 10 pixel error for a small bounding box.

The MSE often works fairly well as a cost function to train the model, but it is not a great metric to evaluate how well the model can predict bounding boxes. The most common metric for this is the Intersection over Union (IoU): it is the area of overlap between the predicted bounding box and the target bounding box, divided by the

22 "Crowdsourcing in Computer Vision," A. Kovashka et al. (2016).

area of their union (see Figure 14-23). In tf.keras, it is implemented by the `tf.keras.metrics.MeanIoU` class.

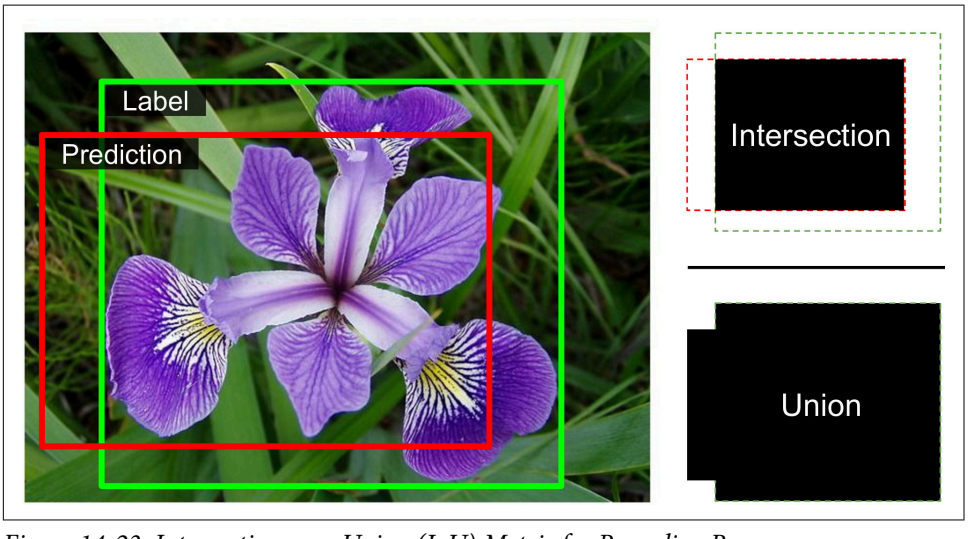

Figure 14-23. Intersection over Union (IoU) Metric for Bounding Boxes

Classifying and localizing a single object is nice, but what if the images contain multiple objects (as is often the case in the flowers dataset)?

Object Detection

The task of classifying and localizing multiple objects in an image is called *object detection*. Until a few years ago, a common approach was to take a CNN that was trained to classify and locate a single object, then slide it across the image, as shown in Figure 14-24. In this example, the image was chopped into a 6 × 8 grid, and we show a CNN (the thick black rectangle) sliding across all 3 × 3 regions. When the CNN was looking at the top left of the image, it detected part of the left-most rose, and then it detected that same rose again when it was first shifted one step to the right. At the next step, it started detecting part of the top-most rose, and then it detected it again once it was shifted one more step to the right. You would then continue to slide the CNN through the whole image, looking at all 3 × 3 regions. Moreover, since objects can have varying sizes, you would also slide the CNN across regions of different sizes. For example, once you are done with the 3 × 3 regions, you might want to slide the CNN across all 4 × 4 regions as well.

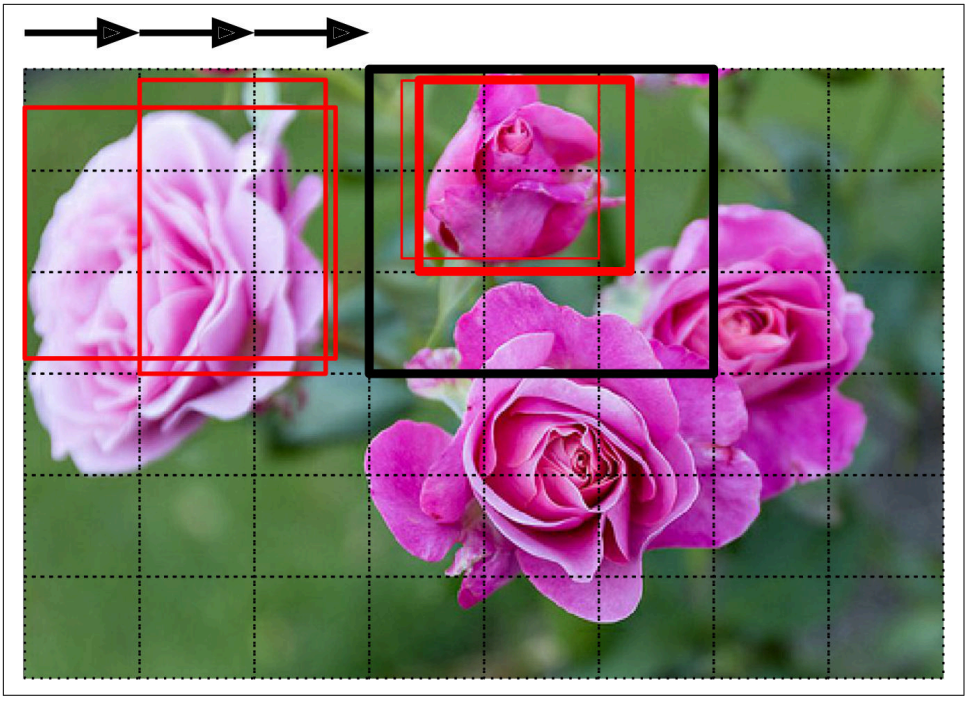

Figure 14-24. Detecting Multiple Objects by Sliding a CNN Across the Image

This technique is fairly straightforward, but as you can see it will detect the same object multiple times, at slightly different positions. Some post-processing will then be needed to get rid of all the unnecessary bounding boxes. A common approach for this is called *non-max suppression*:

- First, you need to add an extra *objectness* output to your CNN, to estimate the probability that a flower is indeed present in the image (alternatively, you could add a "no-flower" class, but this usually does not work as well). It must use the sigmoid activation function and you can train it using the `"binary_crossentropy"` loss. Then just get rid of all the bounding boxes for which the objectness score is below some threshold: this will drop all the bounding boxes that don't actually contain a flower.
- Second, find the bounding box with the highest objectness score, and get rid of all the other bounding boxes that overlap a lot with it (e.g., with an IoU greater than 60%). For example, in Figure 14-24, the bounding box with the max objectness score is the thick bounding box over the top-most rose (the objectness score is represented by the thickness of the bounding boxes). The other bounding box over that same rose overlaps a lot with the max bounding box, so we will get rid of it.

- Third, repeat step two until there are no more bounding boxes to get rid of.

This simple approach to object detection works pretty well, but it requires running the CNN many times, so it is quite slow. Fortunately, there is a much faster way to slide a CNN across an image: using a *Fully Convolutional Network*.

Fully Convolutional Networks (FCNs)

The idea of FCNs was first introduced in a 2015 paper[23] by Jonathan Long et al., for semantic segmentation (the task of classifying every pixel in an image according to the class of the object it belongs to). They pointed out that you could replace the dense layers at the top of a CNN by convolutional layers. To understand this, let's look at an example: suppose a dense layer with 200 neurons sits on top of a convolutional layer that outputs 100 feature maps, each of size 7×7 (this is the feature map size, not the kernel size). Each neuron will compute a weighted sum of all $100 \times 7 \times 7$ activations from the convolutional layer (plus a bias term). Now let's see what happens if we replace the dense layer with a convolution layer using 200 filters, each 7×7, and with VALID padding. This layer will output 200 feature maps, each 1×1 (since the kernel is exactly the size of the input feature maps and we are using VALID padding). In other words, it will output 200 numbers, just like the dense layer did, and if you look closely at the computations performed by a convolutional layer, you will notice that these numbers will be precisely the same as the dense layer produced. The only difference is that the dense layer's output was a tensor of shape [batch size, 200] while the convolutional layer will output a tensor of shape [batch size, 1, 1, 200].

To convert a dense layer to a convolutional layer, the number of filters in the convolutional layer must be equal to the number of units in the dense layer, the filter size must be equal to the size of the input feature maps, and you must use VALID padding. The stride may be set to 1 or more, as we will see shortly.

Why is this important? Well, while a dense layer expects a specific input size (since it has one weight per input feature), a convolutional layer will happily process images of any size[24] (however, it does expect its inputs to have a specific number of channels, since each kernel contains a different set of weights for each input channel). Since an FCN contains only convolutional layers (and pooling layers, which have the same property), it can be trained and executed on images of any size!

23 "Fully Convolutional Networks for Semantic Segmentation," J. Long, E. Shelhamer, T. Darrell (2015).

24 There is one small exception: a convolutional layer using VALID padding will complain if the input size is smaller than the kernel size.

For example, suppose we already trained a CNN for flower classification and localization. It was trained on 224 × 224 images and it outputs 10 numbers: outputs 0 to 4 are sent through the softmax activation function, and this gives the class probabilities (one per class); output 5 is sent through the logistic activation function, and this gives the objectness score; outputs 6 to 9 do not use any activation function, and they represent the bounding box's center coordinates, and its height and width. We can now convert its dense layers to convolutional layers. In fact, we don't even need to retrain it, we can just copy the weights from the dense layers to the convolutional layers! Alternatively, we could have converted the CNN into an FCN before training.

Now suppose the last convolutional layer before the output layer (also called the bottleneck layer) outputs 7 × 7 feature maps when the network is fed a 224 × 224 image (see the left side of Figure 14-25). If we feed the FCN a 448 × 448 image (see the right side of Figure 14-25), the bottleneck layer will now output 14 × 14 feature maps.[25] Since the dense output layer was replaced by a convolutional layer using 10 filters of size 7 × 7, VALID padding and stride 1, the output will be composed of 10 features maps, each of size 8 × 8 (since 14 - 7 + 1 = 8). In other words, the FCN will process the whole image only once and it will output an 8 × 8 grid where each cell contains 10 numbers (5 class probabilities, 1 objectness score and 4 bounding box coordinates). It's exactly like taking the original CNN and sliding it across the image using 8 steps per row and 8 steps per column: to visualize this, imagine chopping the original image into a 14 × 14 grid, then sliding a 7 × 7 window across this grid: there will be 8 × 8 = 64 possible locations for the window, hence 8 × 8 predictions. However, the FCN approach is *much* more efficient, since the network only looks at the image once. In fact, *You Only Look Once* (YOLO) is the name of a very popular object detection architecture!

25 This assumes we used only SAME padding in the network: indeed, VALID padding would reduce the size of the feature maps. Moreover, 448 can be neatly divided by 2 several times until we reach 7, without any rounding error. If any layer uses a different stride than 1 or 2, then there may be some rounding error, so again the feature maps may end up being smaller.