Deep Learning
CS-Elective

Instructor : Dr Muhammad Ismail Mangrio

Slides prepared by Dr. M Asif Khan

ismail@iba-suk.edu.pk

Unit 02 CNN Week 2

Contents

Training a convnet from scratch on a small dataset

- Having to train an **image-classification** model using, you'll likely encounter situation to have **small dataset**
- A "**few**" samples means from **a few hundred to a few tens** of thousands of images.
- As a practical example, we'll focus on **classifying images as dogs or cats**, in a dataset containing **4,000** pictures of cats and dogs (**2,000 cats, 2,000 dogs**).
- We'll use **2,000 pictures for training—1,000 for validation**, and **1,000 for testing**.
- We'll use training a **new model from scratch** using what little data you have.
- You'll start by **naively training** a small convnet on the 2,000 training samples, **without any regularization**, to **set a baseline** for what can be achieved.
- This will get you to a classification **accuracy of 71%**.
- At that point, the main issue will be **overfitting**.

Training a convnet from scratch on a small dataset

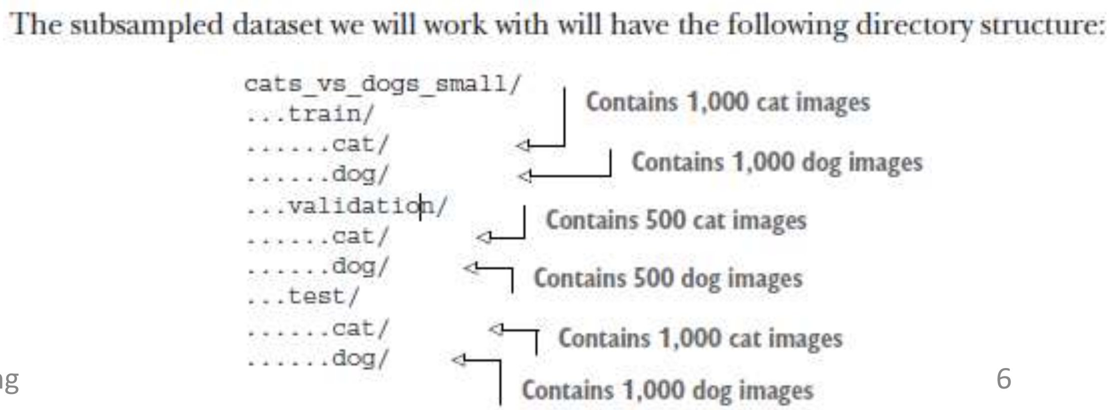
- Then we'll introduce **data augmentation**, a powerful technique for **mitigating overfitting** in computer vision.
- By using data augmentation, you'll improve the network to reach an accuracy of **82%.**
- In the next section, we'll review two more essential techniques for applying deep learning to small datasets: **feature extraction with a pretrained network** (which will get you to an **accuracy of 90% to 96%**).
- Then **fine-tuning a pretrained network** (this will get you to a final **accuracy of 97%**).

Downloading the data

- The **Dogs vs. Cats** dataset that you'll use **isn't packaged with Ke**ras.
- It was made available by **Kaggle** as part of a computer-vision competition in late 2013,
- You can download the original dataset from **www.kaggle.com/c/dogs-vs-cats/data** (you'll need to create a Kaggle account if you don't already have one—don't worry, the process is painless).

Downloading the data

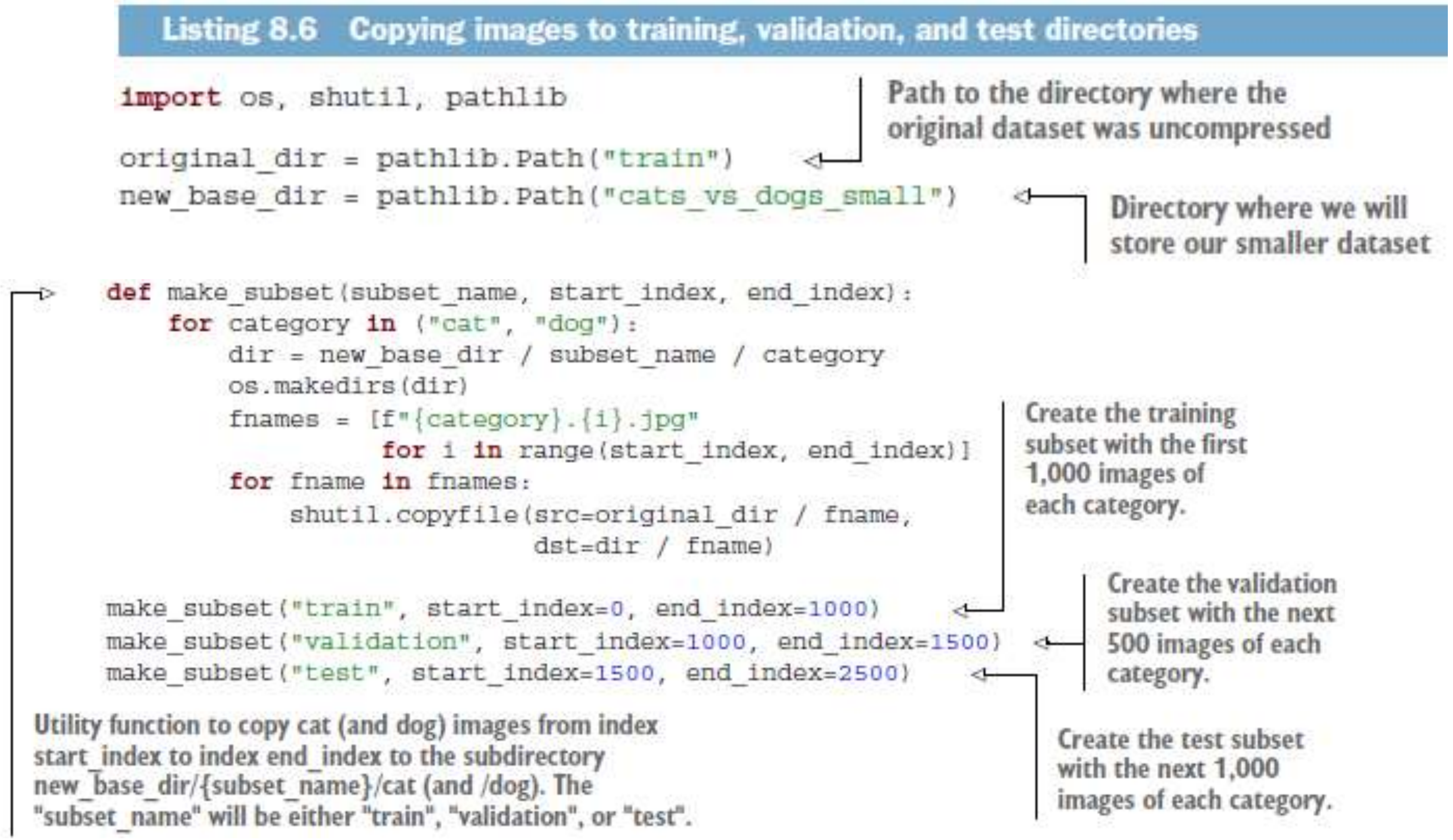
- This dataset contains **25,000 images** of dogs and cats (**12,500** from each class) and is 543 MB (compressed).
- After downloading and uncompressing it, you'll **create a new dataset** containing three subsets: a **training set with 1,000 samples** of each class,
- a **validation set with 500** samples of each class,
- and a **test set with 500** samples of each class.



The subsampled dataset we will work with will have the following directory structure:

```
cats_vs_dogs_small/
...train/
......cat/      Contains 1,000 cat images
......dog/      Contains 1,000 dog images
...validation/
......cat/      Contains 500 cat images
......dog/      Contains 500 dog images
...test/
......cat/      Contains 1,000 cat images
......dog/      Contains 1,000 dog images
```

Copying images to training, validation, and test directories



Listing 8.6 Copying images to training, validation, and test directories

```python
import os, shutil, pathlib

original_dir = pathlib.Path("train")
new_base_dir = pathlib.Path("cats_vs_dogs_small")

def make_subset(subset_name, start_index, end_index):
    for category in ("cat", "dog"):
        dir = new_base_dir / subset_name / category
        os.makedirs(dir)
        fnames = [f"{category}.{i}.jpg"
                  for i in range(start_index, end_index)]
        for fname in fnames:
            shutil.copyfile(src=original_dir / fname,
                            dst=dir / fname)

make_subset("train", start_index=0, end_index=1000)
make_subset("validation", start_index=1000, end_index=1500)
make_subset("test", start_index=1500, end_index=2500)
```

Path to the directory where the original dataset was uncompressed

Directory where we will store our smaller dataset

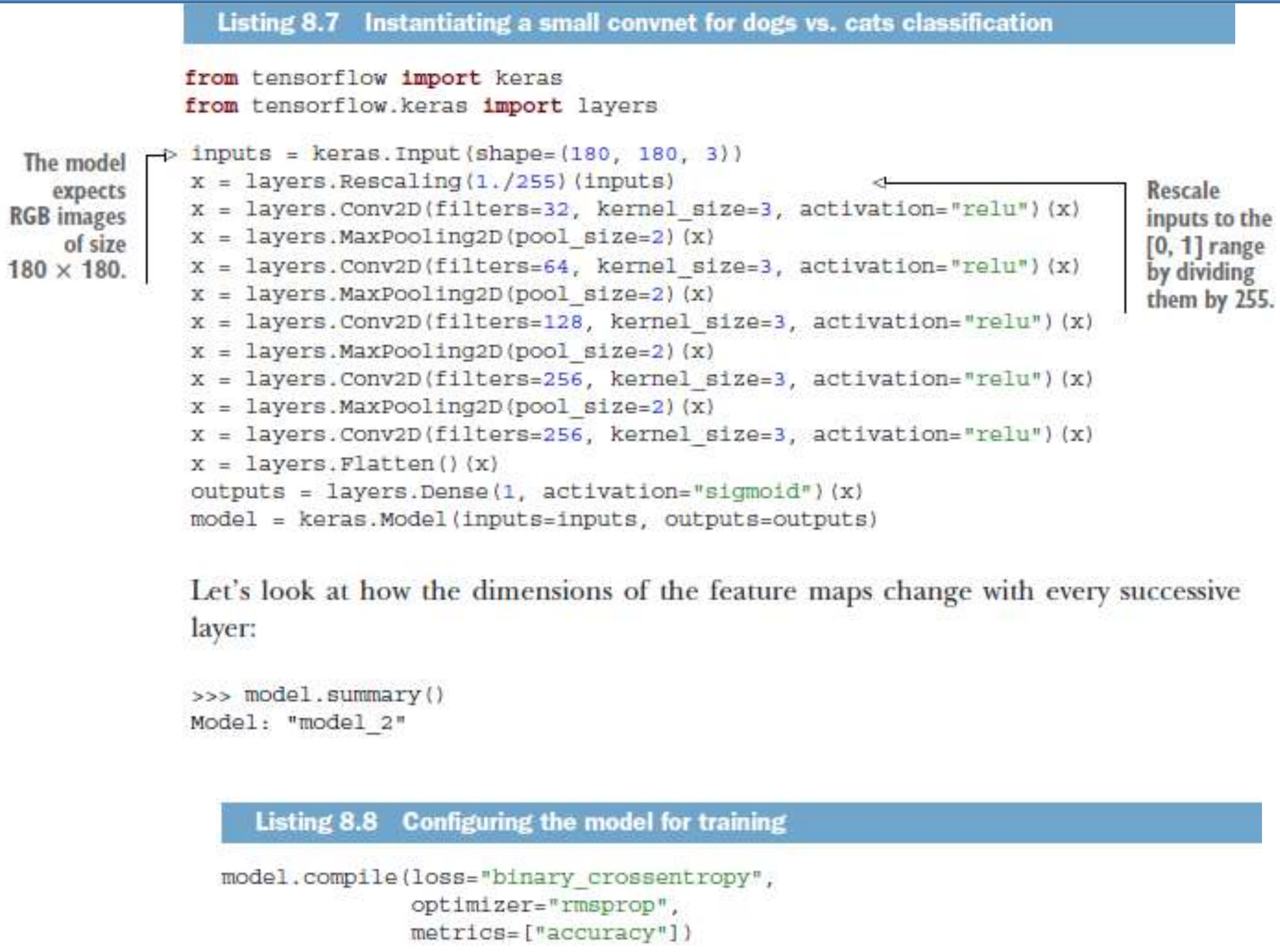
Utility function to copy cat (and dog) images from index start_index to index end_index to the subdirectory new_base_dir/{subset_name}/cat (and /dog). The "subset_name" will be either "train", "validation", or "test".

Create the training subset with the first 1,000 images of each category.

Create the validation subset with the next 500 images of each category.

Create the test subset with the next 1,000 images of each category.

Building your network



Listing 8.7 Instantiating a small convnet for dogs vs. cats classification

```
from tensorflow import keras
from tensorflow.keras import layers

inputs = keras.Input(shape=(180, 180, 3))
x = layers.Rescaling(1./255)(inputs)
x = layers.Conv2D(filters=32, kernel_size=3, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=64, kernel_size=3, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=128, kernel_size=3, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=256, kernel_size=3, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=256, kernel_size=3, activation="relu")(x)
x = layers.Flatten()(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs=inputs, outputs=outputs)
```

The model expects RGB images of size 180 × 180.

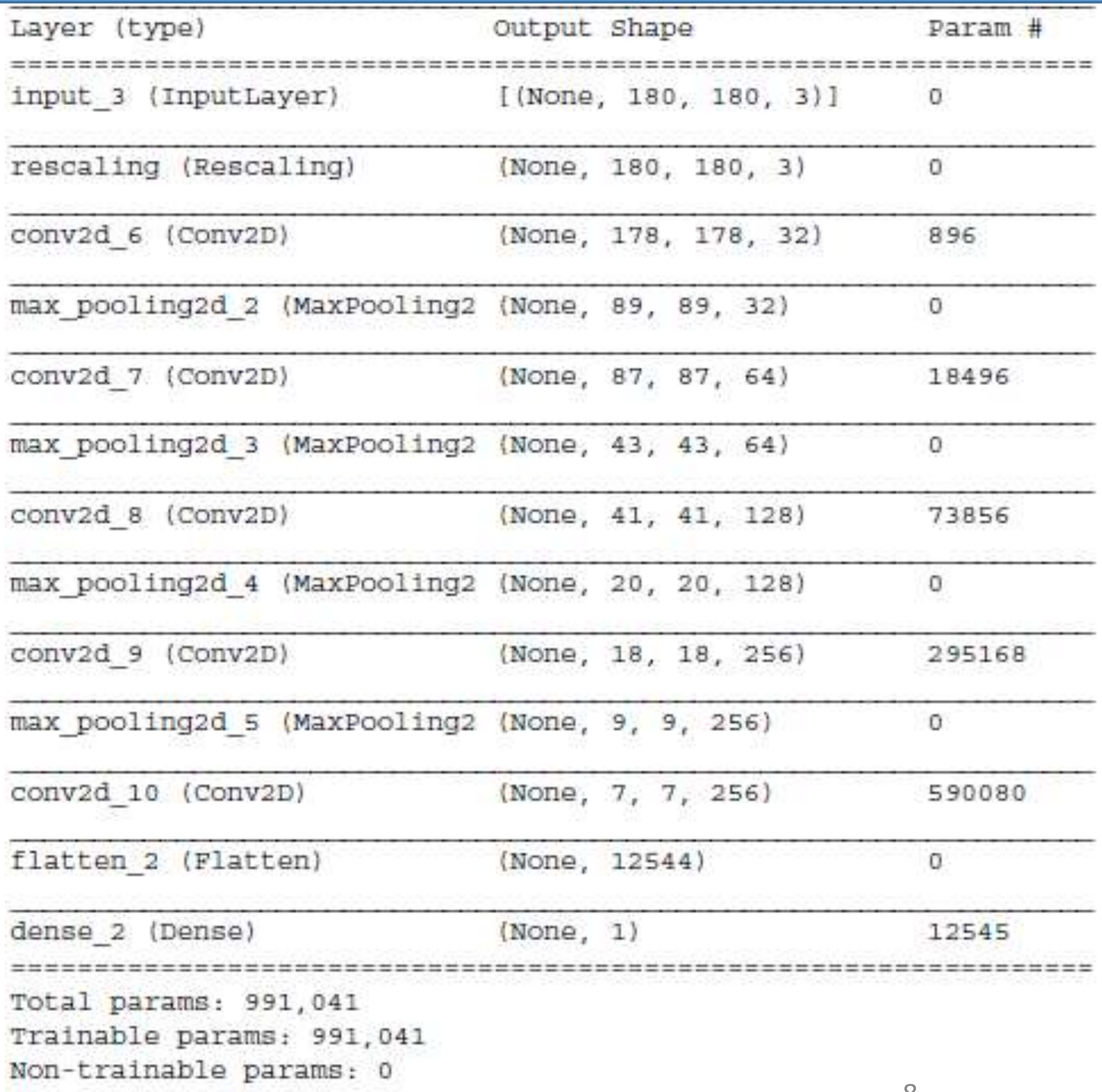
Rescale inputs to the [0, 1] range by dividing them by 255.

Let's look at how the dimensions of the feature maps change with every successive layer:

```
>>> model.summary()
Model: "model_2"
```

Listing 8.8 Configuring the model for training

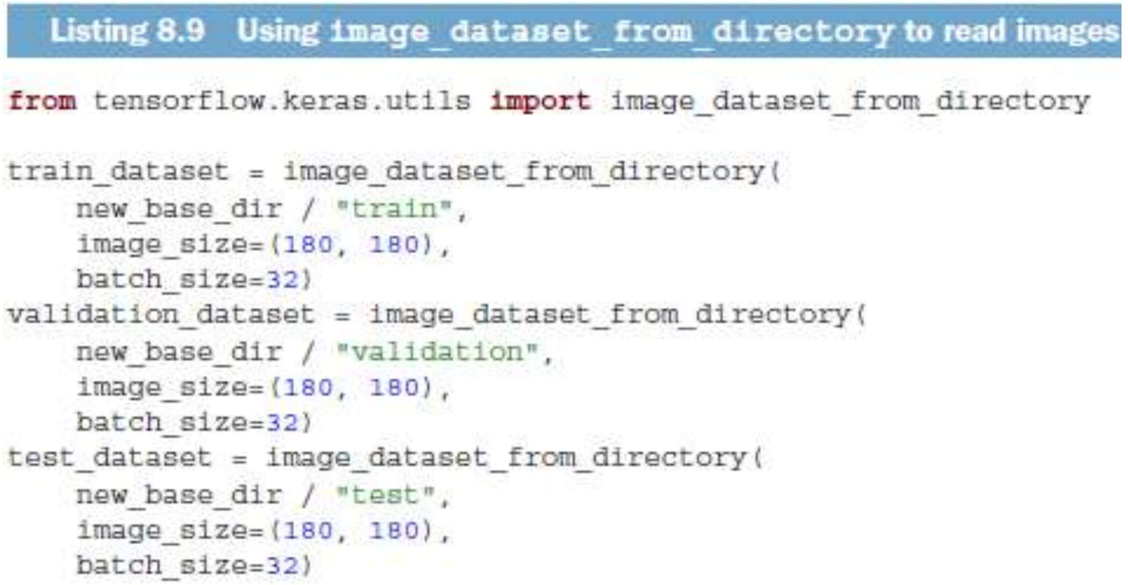
```
model.compile(loss="binary_crossentropy",
              optimizer="rmsprop",
              metrics=["accuracy"])
```



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_3 (InputLayer)         [(None, 180, 180, 3)]     0
_________________________________________________________________
rescaling (Rescaling)        (None, 180, 180, 3)       0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 178, 178, 32)      896
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 89, 89, 32)        0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 87, 87, 64)        18496
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 43, 43, 64)        0
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 41, 41, 128)       73856
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 20, 20, 128)       0
_________________________________________________________________
conv2d_9 (Conv2D)            (None, 18, 18, 256)       295168
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 9, 9, 256)         0
_________________________________________________________________
conv2d_10 (Conv2D)           (None, 7, 7, 256)         590080
_________________________________________________________________
flatten_2 (Flatten)          (None, 12544)             0
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 12545
=================================================================
Total params: 991,041
Trainable params: 991,041
Non-trainable params: 0
```

Data preprocessing

- Currently, the data sits on a drive as JPEG files, so the steps for getting it into the network are roughly as follows:
  1. Read the picture files.
  2. Decode the JPEG content to RGB grids of pixels.
  3. Convert these into floating-point tensors.
  4. Rescale the pixel values (between 0 and 255) to the [0, 1] interval (as you know, neural networks prefer to deal with small input values).

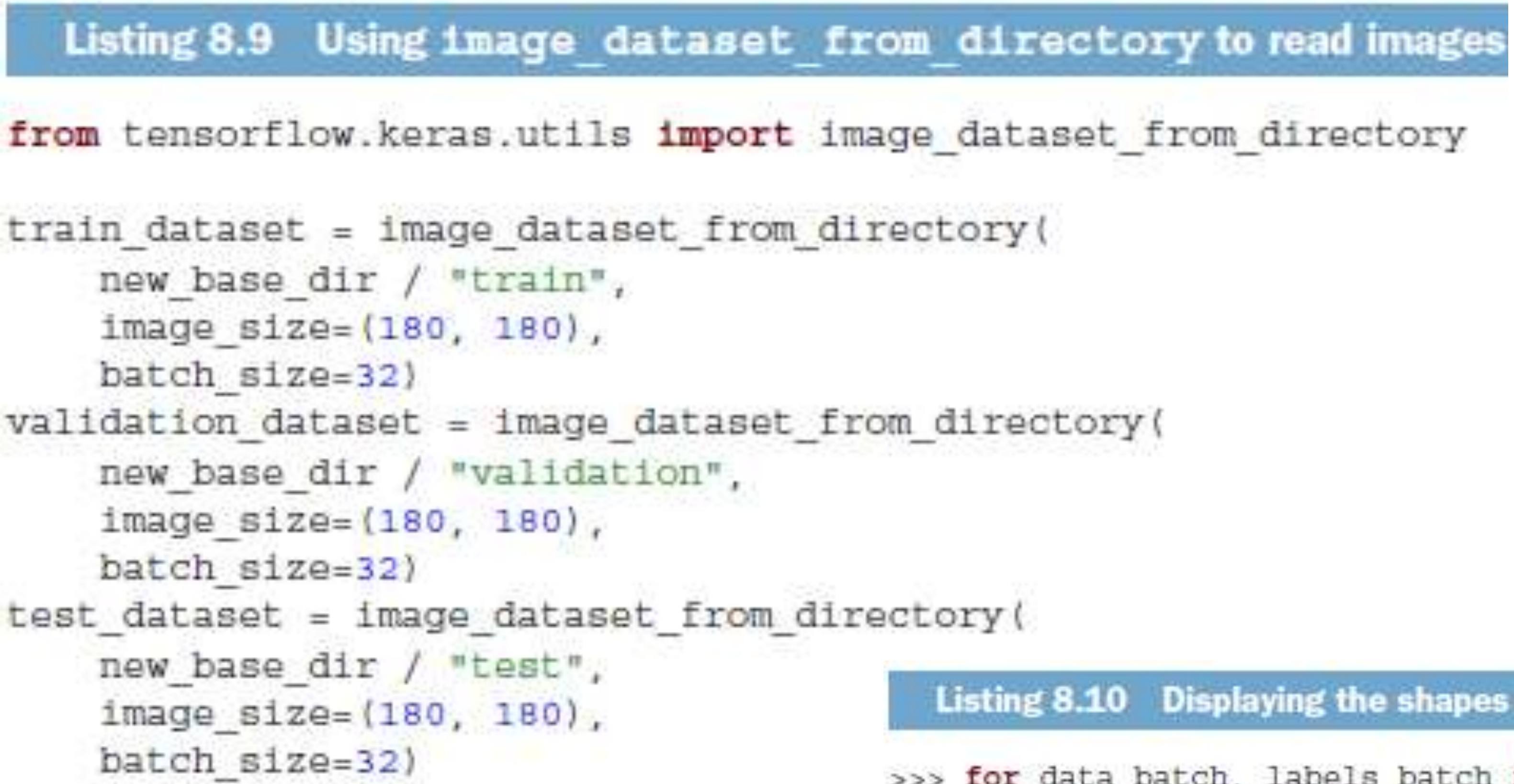


Listing 8.9 Using `image_dataset_from_directory` to read images

```
from tensorflow.keras.utils import image_dataset_from_directory

train_dataset = image_dataset_from_directory(
    new_base_dir / "train",
    image_size=(180, 180),
    batch_size=32)
validation_dataset = image_dataset_from_directory(
    new_base_dir / "validation",
    image_size=(180, 180),
    batch_size=32)
test_dataset = image_dataset_from_directory(
    new_base_dir / "test",
    image_size=(180, 180),
    batch_size=32)
```

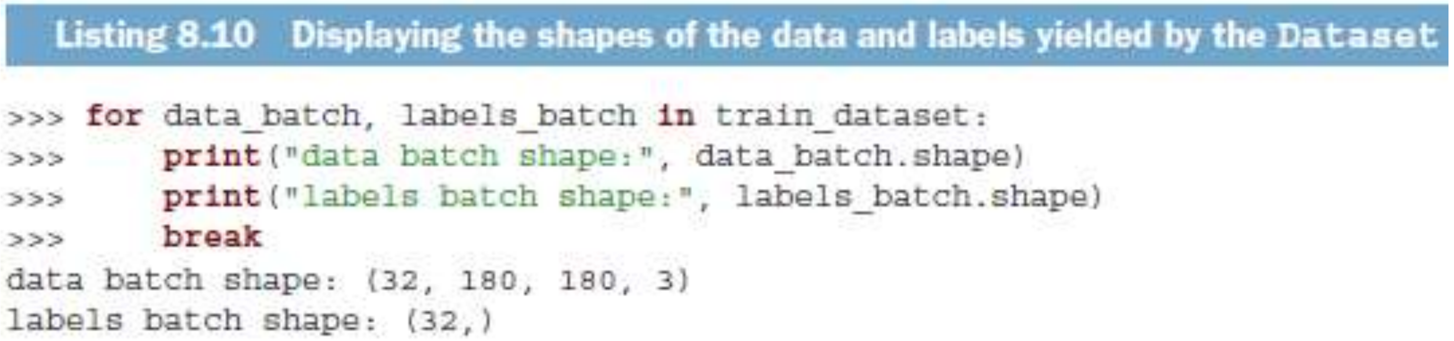
Keras has a module with image-processing helper tools, located at ***keras.preprocessing.image***.
In particular, it contains the class ImageDataGenerator, which lets you quickly set up Python generators that can automatically **turn image files on disk into batches of preprocessed tensors**.

Data preprocessing



Listing 8.9 Using `image_dataset_from_directory` to read images

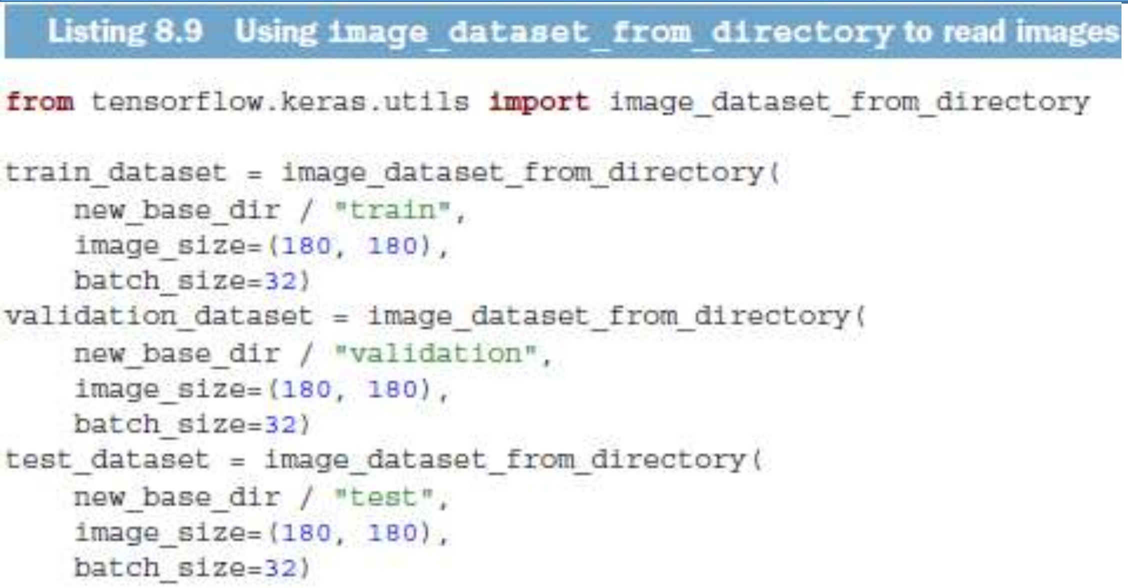
```
from tensorflow.keras.utils import image_dataset_from_directory

train_dataset = image_dataset_from_directory(
    new_base_dir / "train",
    image_size=(180, 180),
    batch_size=32)
validation_dataset = image_dataset_from_directory(
    new_base_dir / "validation",
    image_size=(180, 180),
    batch_size=32)
test_dataset = image_dataset_from_directory(
    new_base_dir / "test",
    image_size=(180, 180),
    batch_size=32)
```



Listing 8.10 Displaying the shapes of the data and labels yielded by the `Dataset`

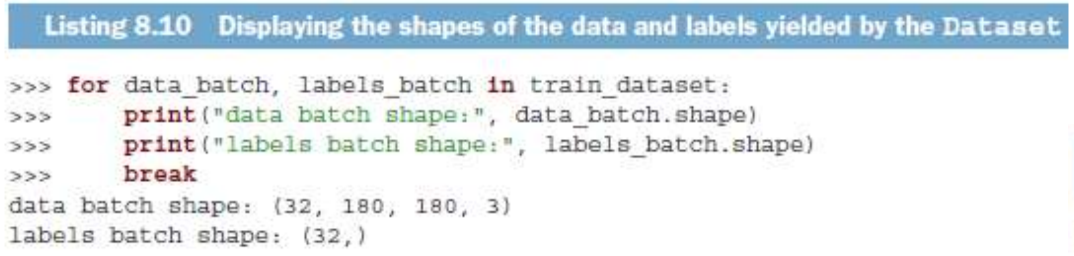
```
>>> for data_batch, labels_batch in train_dataset:
>>>     print("data batch shape:", data_batch.shape)
>>>     print("labels batch shape:", labels_batch.shape)
>>>     break
data batch shape: (32, 180, 180, 3)
labels batch shape: (32,)
```

Data preprocessing



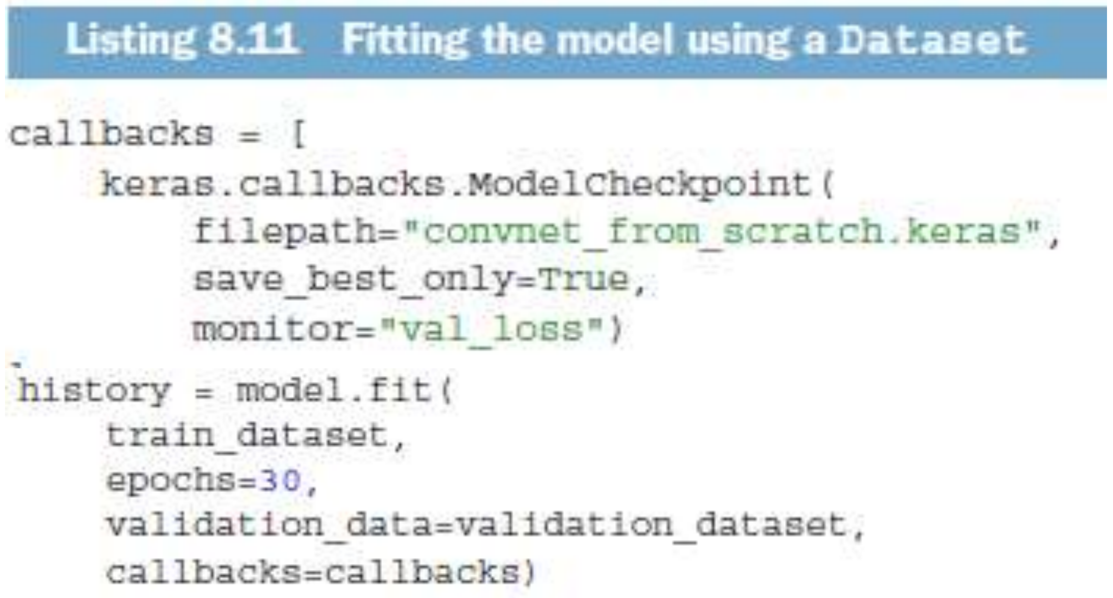
Listing 8.9 Using `image_dataset_from_directory` to read images

```
from tensorflow.keras.utils import image_dataset_from_directory

train_dataset = image_dataset_from_directory(
    new_base_dir / "train",
    image_size=(180, 180),
    batch_size=32)
validation_dataset = image_dataset_from_directory(
    new_base_dir / "validation",
    image_size=(180, 180),
    batch_size=32)
test_dataset = image_dataset_from_directory(
    new_base_dir / "test",
    image_size=(180, 180),
    batch_size=32)
```



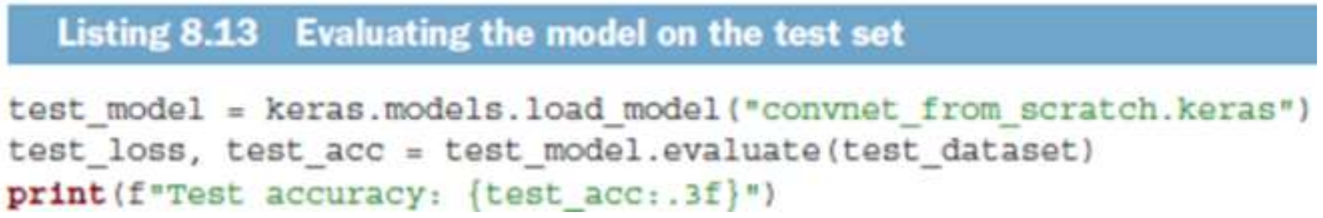
Listing 8.10 Displaying the shapes of the data and labels yielded by the `Dataset`

```
>>> for data_batch, labels_batch in train_dataset:
>>>     print("data batch shape:", data_batch.shape)
>>>     print("labels batch shape:", labels_batch.shape)
>>>     break
data batch shape: (32, 180, 180, 3)
labels batch shape: (32,)
```



Listing 8.11 Fitting the model using a `Dataset`

```
callbacks = [
    keras.callbacks.ModelCheckpoint(
        filepath="convnet_from_scratch.keras",
        save_best_only=True,
        monitor="val_loss")
]
history = model.fit(
    train_dataset,
    epochs=30,
    validation_data=validation_dataset,
    callbacks=callbacks)
```



Listing 8.13 Evaluating the model on the test set

```
test_model = keras.models.load_model("convnet_from_scratch.keras")
test_loss, test_acc = test_model.evaluate(test_dataset)
print(f"Test accuracy: {test_acc:.3f}")
```

We get a test **accuracy of 69.5%.** (Due to the randomness of neural network initializations, you may get numbers within one percentage point of that.)

First milestone

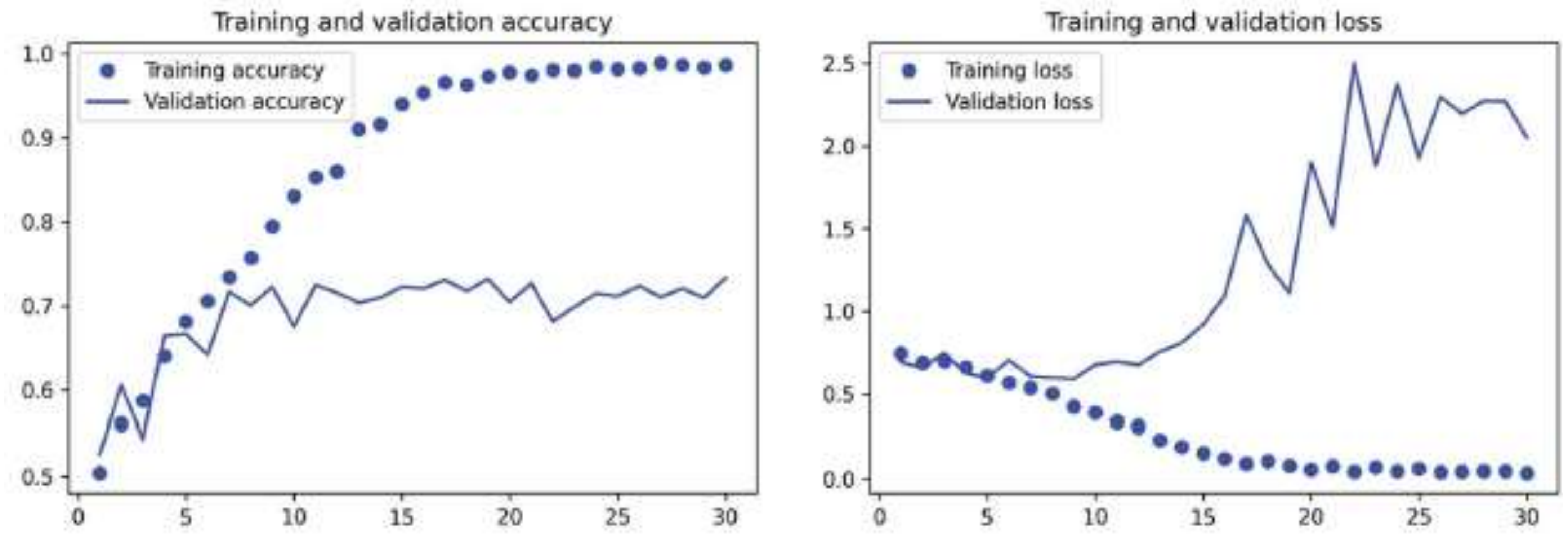


Figure 8.9 Training and validation metrics for a simple convnet

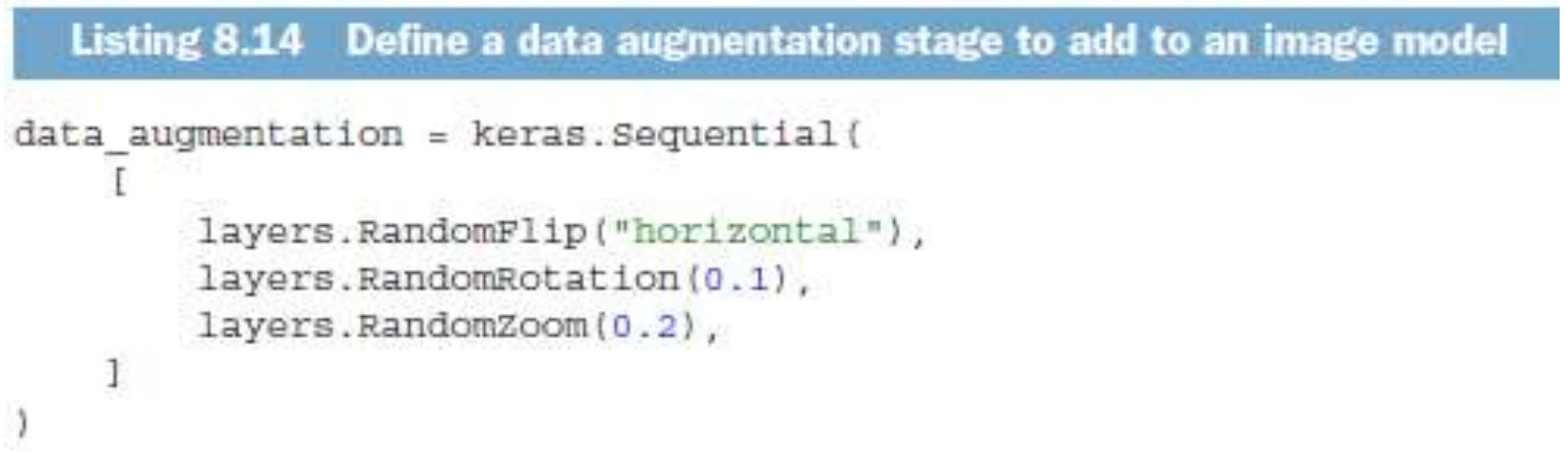
- The training accuracy increases linearly over time, until it reaches nearly 100%, whereas the validation accuracy peaks at 75%.
- The validation loss reaches its minimum after only ten epochs and then stalls, whereas the training loss keeps decreasing linearly as training proceeds. **Accuracy of 69.5%!**

First milestone

- These plots are characteristic of **overfitting**.
- The training accuracy increases linearly over time, until it reaches nearly 100%,
- whereas the **validation accuracy stalls** at 69**–72%**.
- The validation loss reaches its **minimum after only five epochs** and then stalls,
- Whereas the training loss keeps decreasing linearly until it reaches nearly 0.
- Because you have relatively **few training samples** (2,000), overfitting will be your number-one concern.
- You already know about a number of **techniques that can help mitigate overfitting**, such as **dropou**t and **weight decay** (L2 regularization).
- We're now going to work with a new one, specific to computer vision and used almost universally when processing images with deep-learning models: **data augmentation**.

Using data augmentation

- **Data augmentation** takes the approach of **generating more training data** from existing training samples, by augmenting (increasing) the samples via a number of random transformations.
- These random transformations yield believable-looking images.
- The goal is that at training time, your model will **never see the exact same picture twice**.
- This helps expose the model to **more aspects** of the data and **generalize better**.
- To further fight overfitting, you'll also **add a Dropout layer** to your model.
- The dropout will be added right **before the densely connected classifier**.
- In **Keras**, this can be done by adding a number of **data augmentation layers at the start** of your model.



Listing 8.14 Define a data augmentation stage to add to an image model

```
data_augmentation = keras.Sequential(
    [
        layers.RandomFlip("horizontal"),
        layers.RandomRotation(0.1),
        layers.RandomZoom(0.2),
    ]
)
```

Using data augmentation

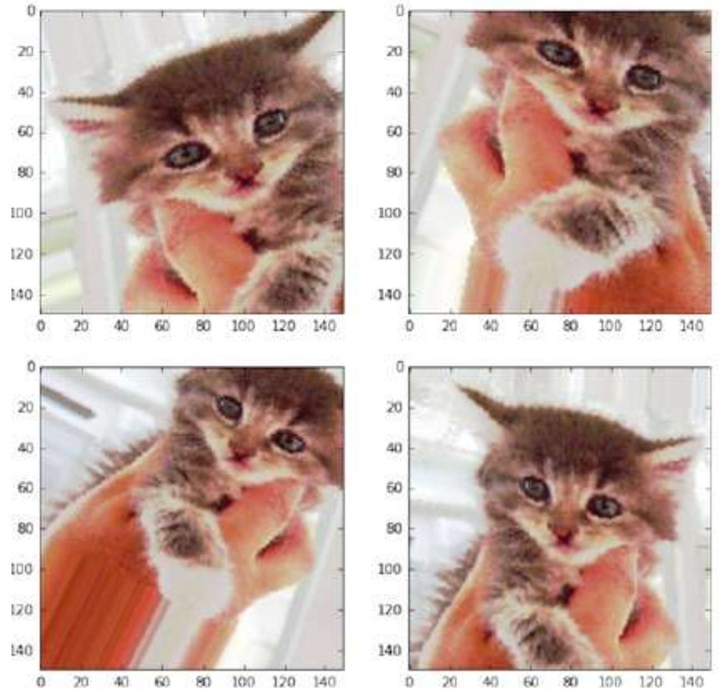
Listing 8.15 Displaying some randomly augmented training images

```
plt.figure(figsize=(10, 10))
for images, _ in train_dataset.take(1):
    for i in range(9):
        augmented_images = data_augmentation(images)
        ax = plt.subplot(3, 3, i + 1)
        plt.imshow(augmented_images[0].numpy().astype("uint8"))
        plt.axis("off")
```

We can use take(N) to only sample *N* batches from the dataset. This is equivalent to inserting a break in the loop after the *N*th batch.

Apply the augmentation stage to the batch of images.

Display the first image in the output batch. For each of the nine iterations, this is a different augmentation of the same image.



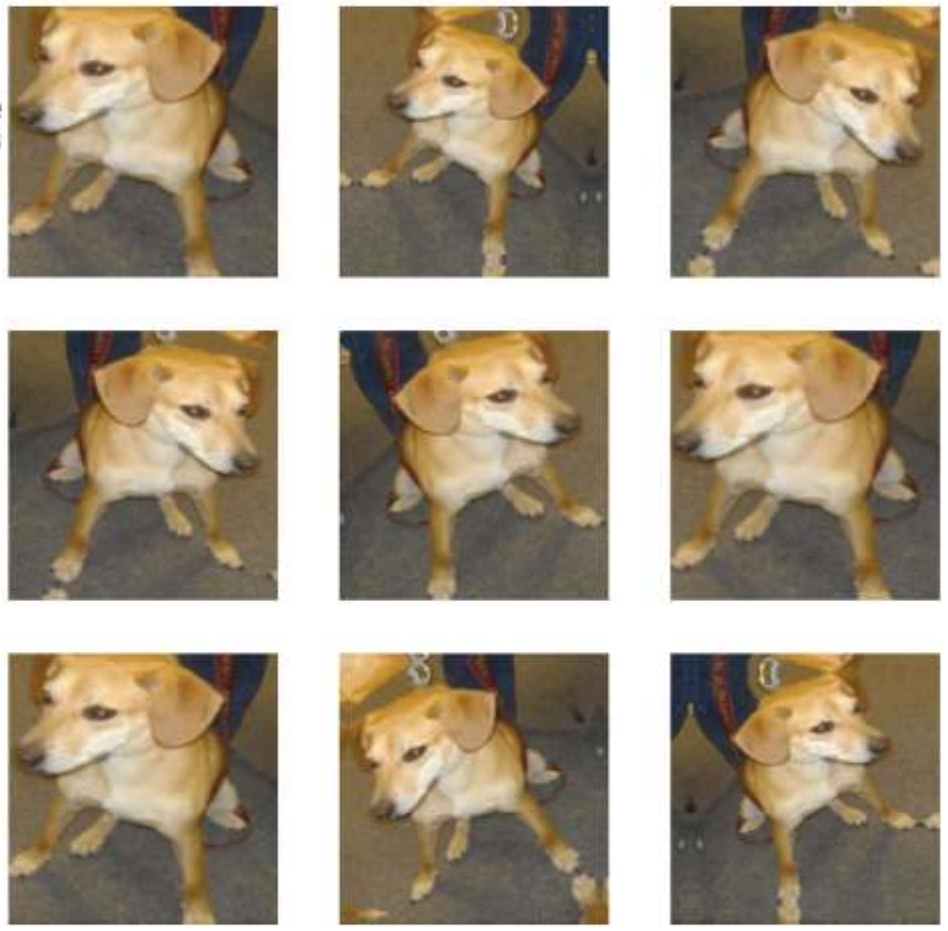
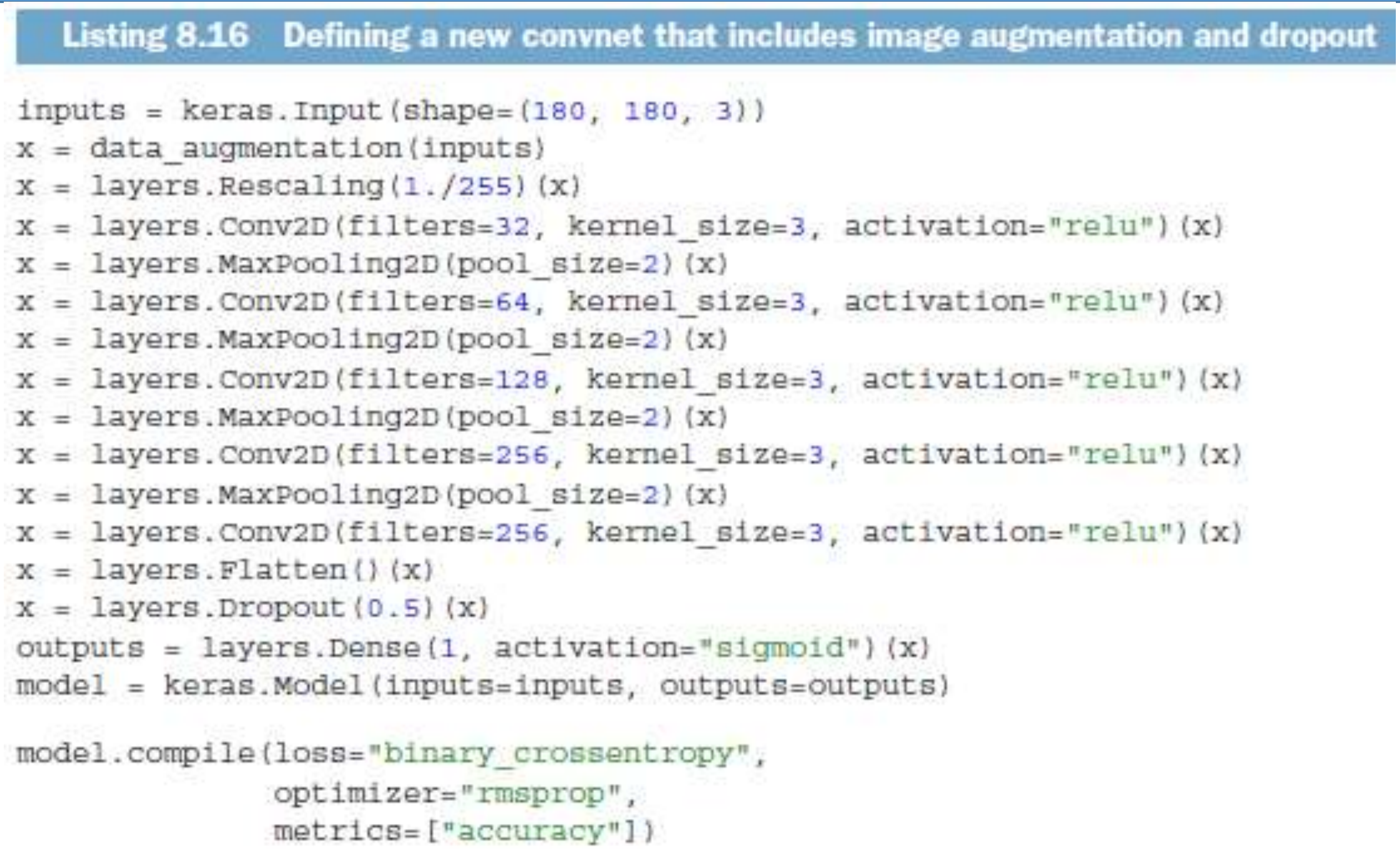


Figure 8.10 Generating variations of a very good boy via random data augmentation

Using data augmentation

- .

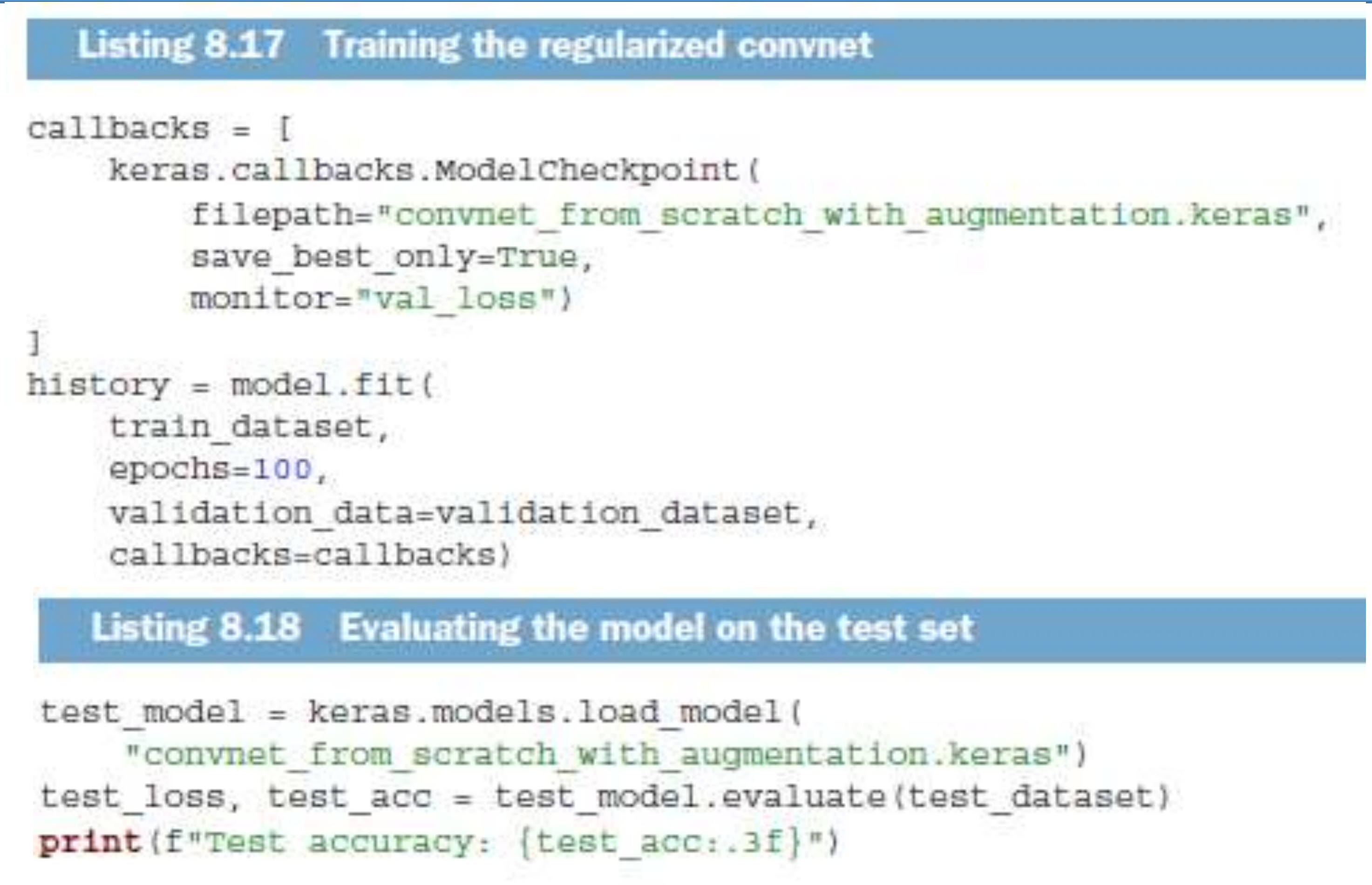


Listing 8.16 Defining a new convnet that includes image augmentation and dropout

```
inputs = keras.Input(shape=(180, 180, 3))
x = data_augmentation(inputs)
x = layers.Rescaling(1./255)(x)
x = layers.Conv2D(filters=32, kernel_size=3, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=64, kernel_size=3, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=128, kernel_size=3, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=256, kernel_size=3, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=256, kernel_size=3, activation="relu")(x)
x = layers.Flatten()(x)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs=inputs, outputs=outputs)

model.compile(loss="binary_crossentropy",
              optimizer="rmsprop",
              metrics=["accuracy"])
```

Using data augmentation

- .



Listing 8.17 Training the regularized convnet

```
callbacks = [
    keras.callbacks.ModelCheckpoint(
        filepath="convnet_from_scratch_with_augmentation.keras",
        save_best_only=True,
        monitor="val_loss")
]
history = model.fit(
    train_dataset,
    epochs=100,
    validation_data=validation_dataset,
    callbacks=callbacks)
```

Listing 8.18 Evaluating the model on the test set

```
test_model = keras.models.load_model(
    "convnet_from_scratch_with_augmentation.keras")
test_loss, test_acc = test_model.evaluate(test_dataset)
print(f"Test accuracy: {test_acc:.3f}")
```

Using data augmentation (2nd milestone)

- You now reach an **accuracy of 82%,** a **15% relative improvement** over the non-regularized model.

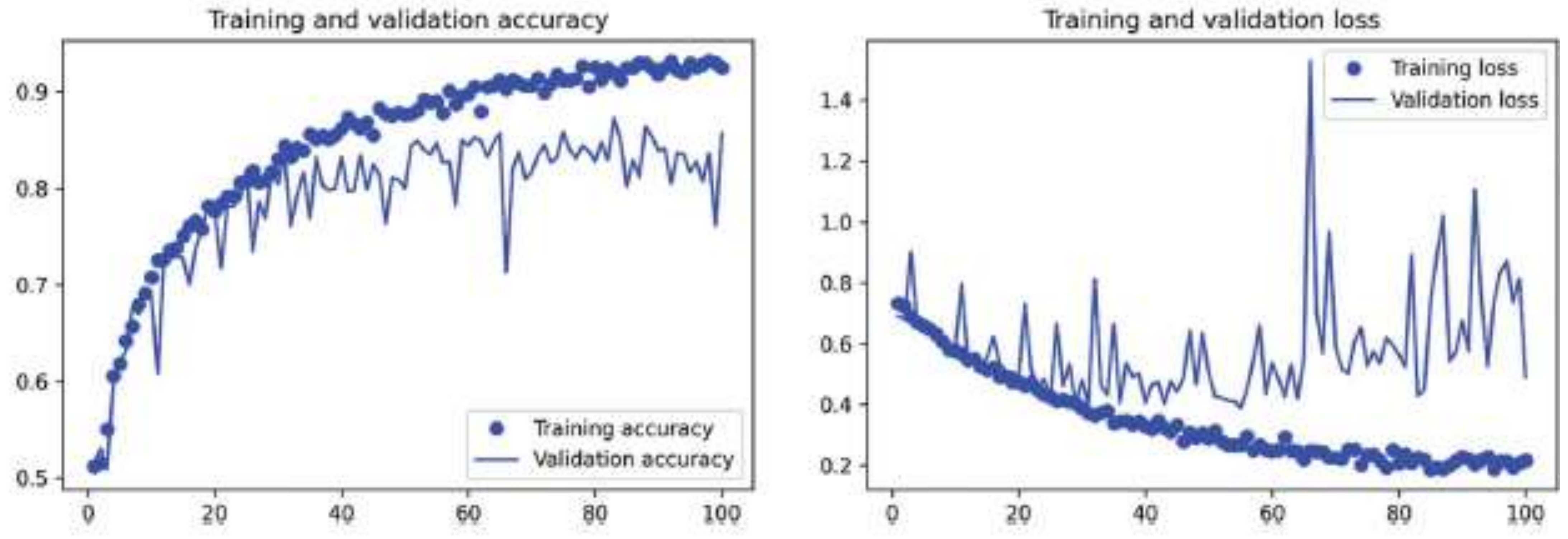


Figure 8.11 Training and validation metrics with data augmentation

Leveraging a pretrained convnet

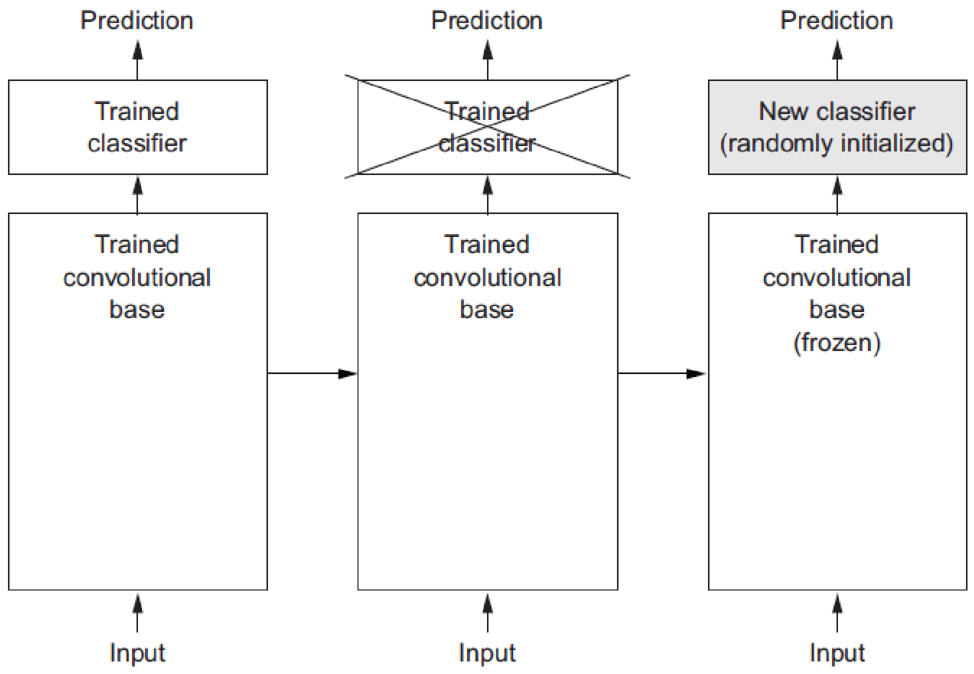
- A common and highly effective approach to deep learning on small image datasets is to **use a pretrained network**.
- A pretrained network is **a saved network** that was **previously trained on a large dataset**, typically on a large-scale image-classification task.
- If this original dataset **is large enough** and **general enough**, then the spatial hierarchy of features learned by the pretrained network can **effectively act as a generic model of the visual world**,
- and hence its features can prove useful for many different computer vision
- problems,
- even though these new problems **may involve completely different classes** than those of the original task.

Using a pretrained convnet

- For instance, you might train a network on **ImageNet** (where classes are mostly **animals** and everyday objects) and then repurpose this trained network for something as remote as **identifying furniture** items in images.
- In this case, let’s consider a large convnet trained on the ImageNet dataset
- (1.4 million labeled images and 1,000 different classes).
- ImageNet contains many animal classes, including different species of cats and dogs, and you can thus expect to perform well on the dogs-versus-cats classification problem.
- You’ll use the **VGG16 architecture.**

Using a pretrained convent (Feature extraction)

- **Convnets** used for image classification comprise two parts:
  - Start with a series of **pooling and convolution layers**,
  - and they end with **a densely connected** classifier.
- The first part is called the **convolutional base** of the model.
- In the case of convnets, feature extraction consists of **taking the convolutional base** of a previously trained network, running the new data through it, and training a new classifier on top of the output (see figure 5.14).



Prediction
Trained classifier
Trained convolutional base
Input
Prediction
~~Trained classifier~~
Trained convolutional base
Input
Prediction
New classifier (randomly initialized)
Trained convolutional base (frozen)
Input

Figure 5.14 Swapping classifiers while keeping the same convolutional base

Using a pretrained convent (Feature extraction)

- The VGG16 model, among others, comes **prepackaged with Keras**.
- You can import it from the keras.applications module.
- Many other image-classification models (all pretrained on the ImageNet dataset) are available as part of keras.applications:
  - Xception
  - ResNet
  - MobileNet
  - EfficientNet
  - DenseNet, etc.

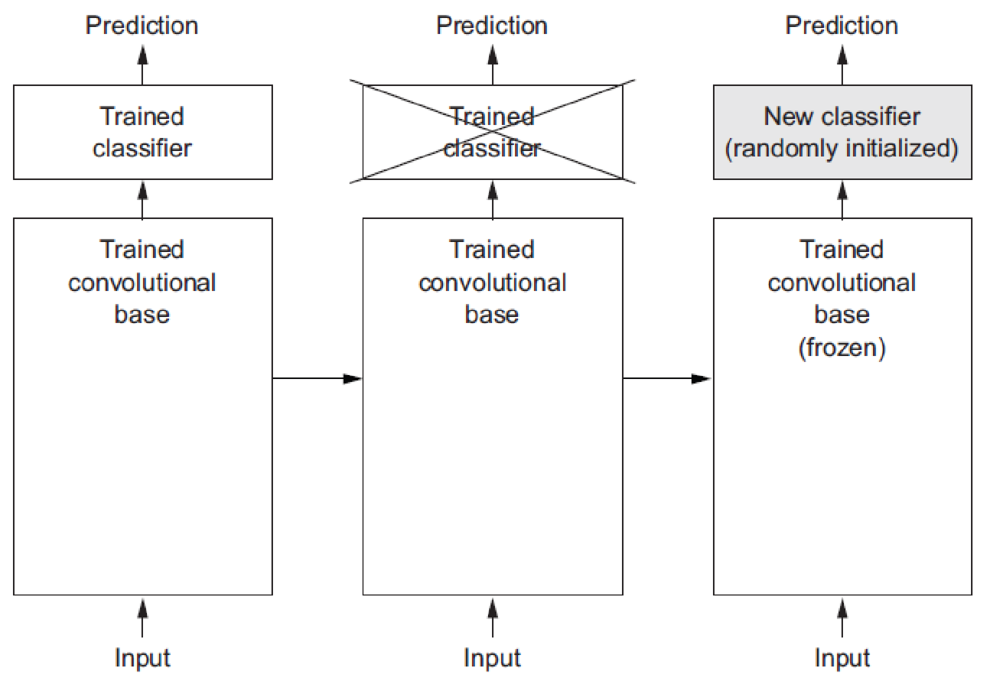
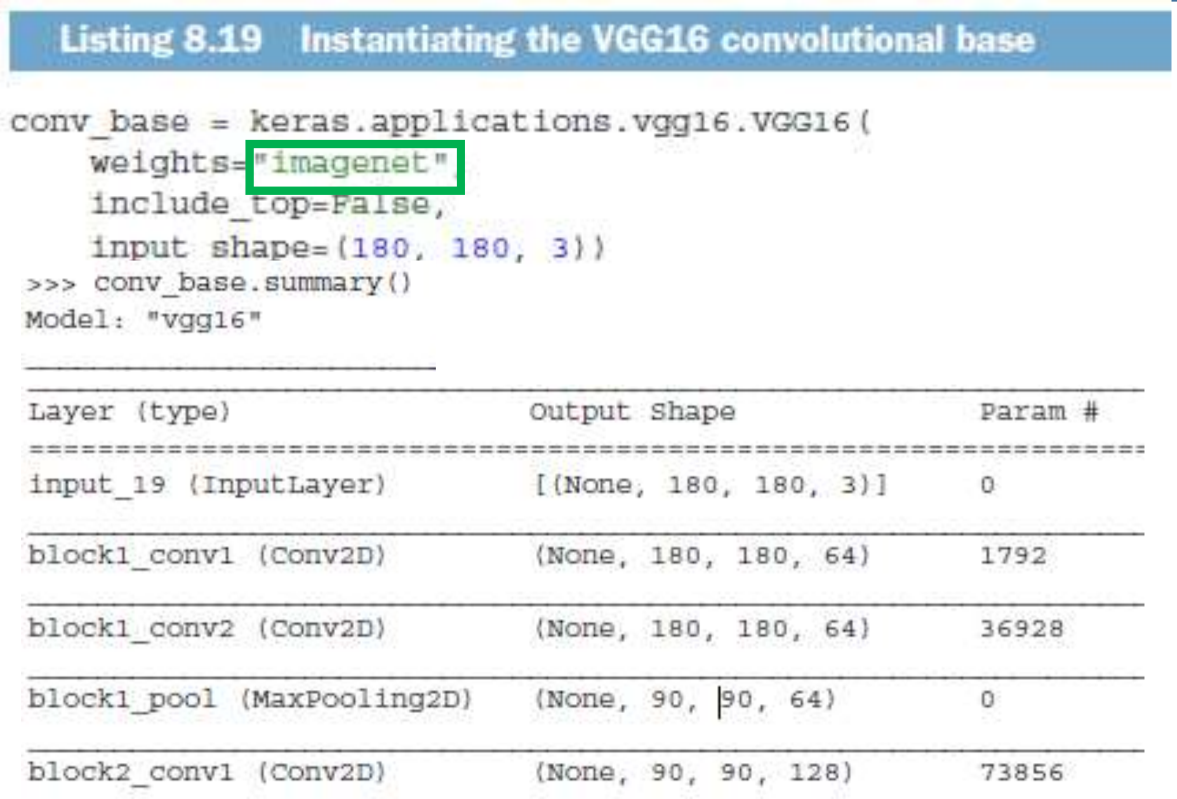

Prediction
Prediction
Prediction
Trained classifier
~~Trained classifier~~
New classifier (randomly initialized)
Trained convolutional base
Trained convolutional base
Trained convolutional base (frozen)
Input
Input
Input

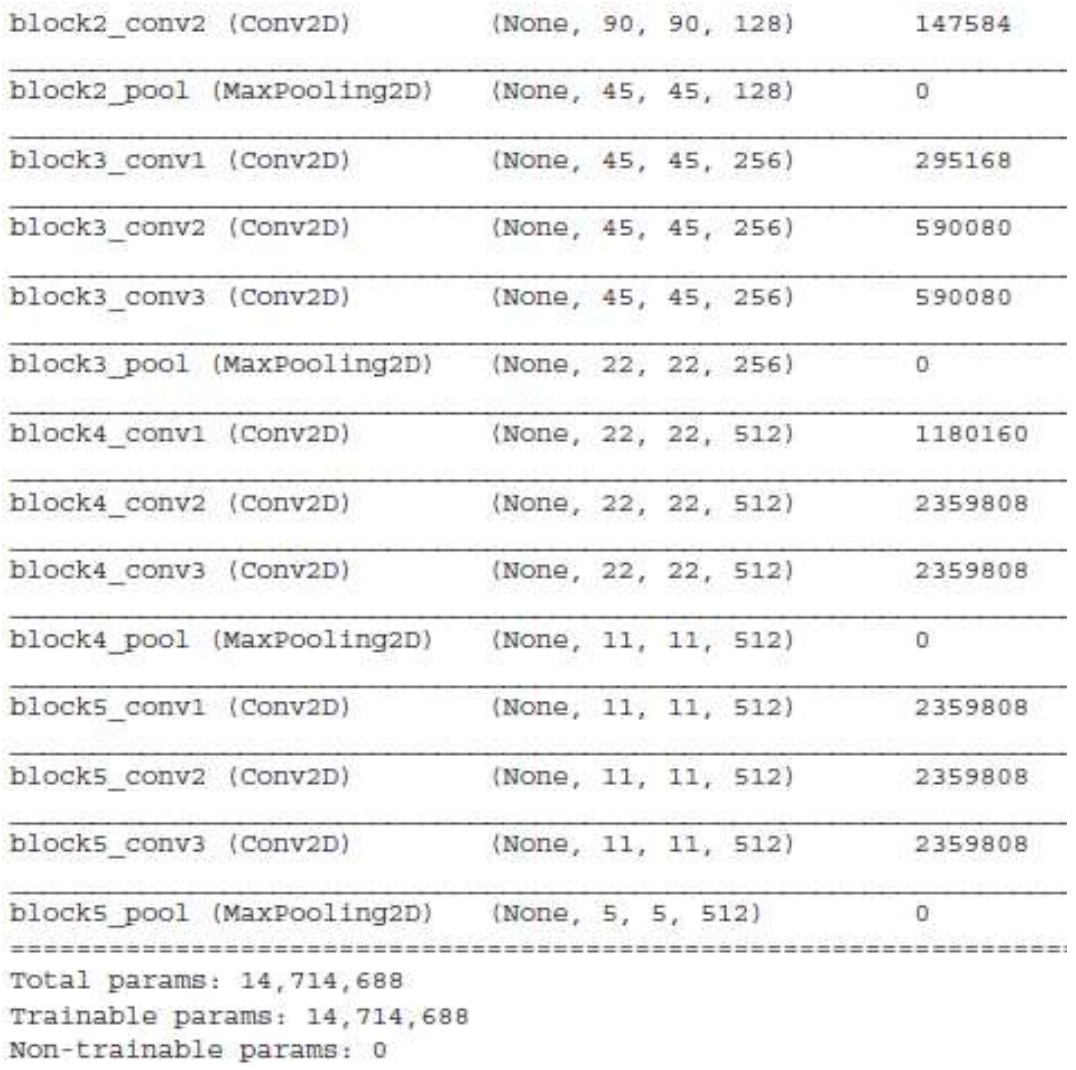
Figure 5.14 **Swapping classifiers while keeping the same convolutional base**

Using a pretrained convent (Feature extraction)



Listing 8.19 Instantiating the VGG16 convolutional base

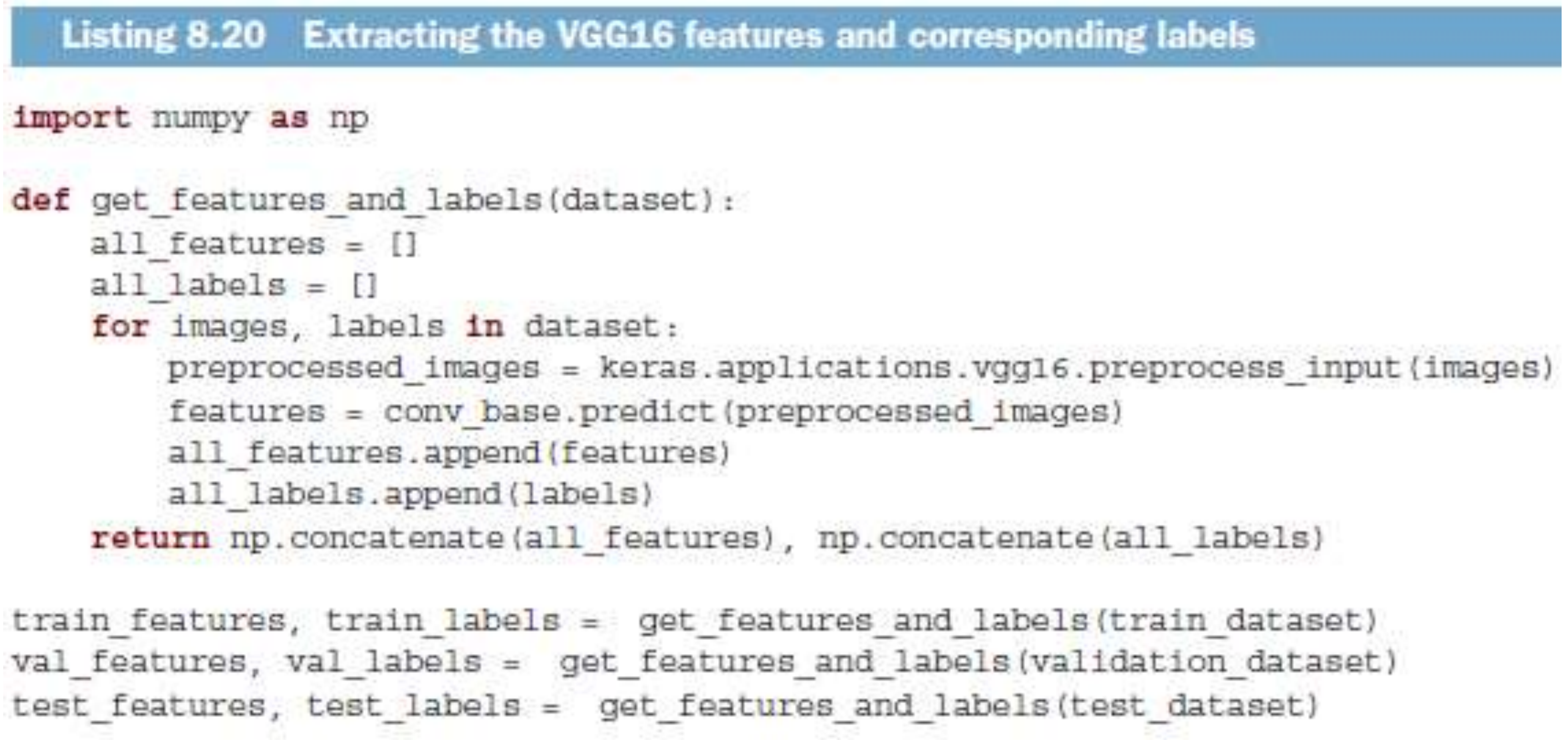
```
conv_base = keras.applications.vgg16.VGG16(
    weights="imagenet",
    include_top=False,
    input_shape=(180, 180, 3))
>>> conv_base.summary()
Model: "vgg16"
________________________
_______________________________________________________________
Layer (type)                 Output Shape              Param #
===============================================================
input_19 (InputLayer)        [(None, 180, 180, 3)]     0
_______________________________________________________________
block1_conv1 (Conv2D)        (None, 180, 180, 64)      1792
_______________________________________________________________
block1_conv2 (Conv2D)        (None, 180, 180, 64)      36928
_______________________________________________________________
block1_pool (MaxPooling2D)   (None, 90, 90, 64)        0
_______________________________________________________________
block2_conv1 (Conv2D)        (None, 90, 90, 128)       73856
```



```
block2_conv2 (Conv2D)        (None, 90, 90, 128)       147584
_______________________________________________________________
block2_pool (MaxPooling2D)   (None, 45, 45, 128)       0
_______________________________________________________________
block3_conv1 (Conv2D)        (None, 45, 45, 256)       295168
_______________________________________________________________
block3_conv2 (Conv2D)        (None, 45, 45, 256)       590080
_______________________________________________________________
block3_conv3 (Conv2D)        (None, 45, 45, 256)       590080
_______________________________________________________________
block3_pool (MaxPooling2D)   (None, 22, 22, 256)       0
_______________________________________________________________
block4_conv1 (Conv2D)        (None, 22, 22, 512)       1180160
_______________________________________________________________
block4_conv2 (Conv2D)        (None, 22, 22, 512)       2359808
_______________________________________________________________
block4_conv3 (Conv2D)        (None, 22, 22, 512)       2359808
_______________________________________________________________
block4_pool (MaxPooling2D)   (None, 11, 11, 512)       0
_______________________________________________________________
block5_conv1 (Conv2D)        (None, 11, 11, 512)       2359808
_______________________________________________________________
block5_conv2 (Conv2D)        (None, 11, 11, 512)       2359808
_______________________________________________________________
block5_conv3 (Conv2D)        (None, 11, 11, 512)       2359808
_______________________________________________________________
block5_pool (MaxPooling2D)   (None, 5, 5, 512)         0
===============================================================
Total params: 14,714,688
Trainable params: 14,714,688
Non-trainable params: 0
```

Using a pretrained convent (Feature extraction)

FAST FEATURE EXTRACTION WITHOUT DATA AUGMENTATION

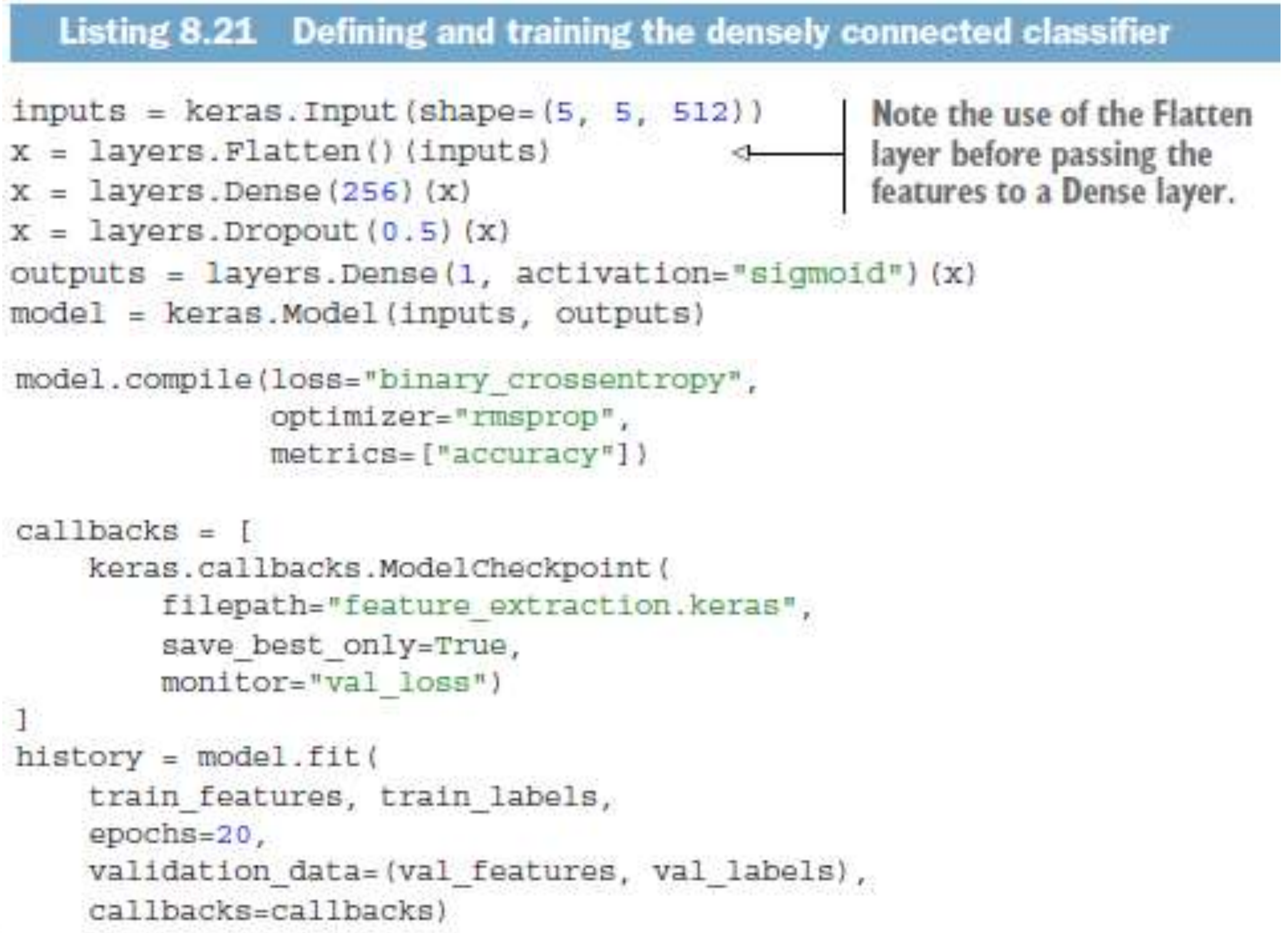


Listing 8.20 Extracting the VGG16 features and corresponding labels

```
import numpy as np

def get_features_and_labels(dataset):
    all_features = []
    all_labels = []
    for images, labels in dataset:
        preprocessed_images = keras.applications.vgg16.preprocess_input(images)
        features = conv_base.predict(preprocessed_images)
        all_features.append(features)
        all_labels.append(labels)
    return np.concatenate(all_features), np.concatenate(all_labels)

train_features, train_labels =  get_features_and_labels(train_dataset)
val_features, val_labels =  get_features_and_labels(validation_dataset)
test_features, test_labels =  get_features_and_labels(test_dataset)
```

Using a pretrained convent (Feature extraction)

FAST FEATURE EXTRACTION WITHOUT DATA AUGMENTATION



Listing 8.21 Defining and training the densely connected classifier

```
inputs = keras.Input(shape=(5, 5, 512))
x = layers.Flatten()(inputs)
x = layers.Dense(256)(x)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)

model.compile(loss="binary_crossentropy",
              optimizer="rmsprop",
              metrics=["accuracy"])

callbacks = [
    keras.callbacks.ModelCheckpoint(
        filepath="feature_extraction.keras",
        save_best_only=True,
        monitor="val_loss")
]
history = model.fit(
    train_features, train_labels,
    epochs=20,
    validation_data=(val_features, val_labels),
    callbacks=callbacks)
```

Note the use of the Flatten layer before passing the features to a Dense layer.

Using a pretrained convent (Feature extraction)

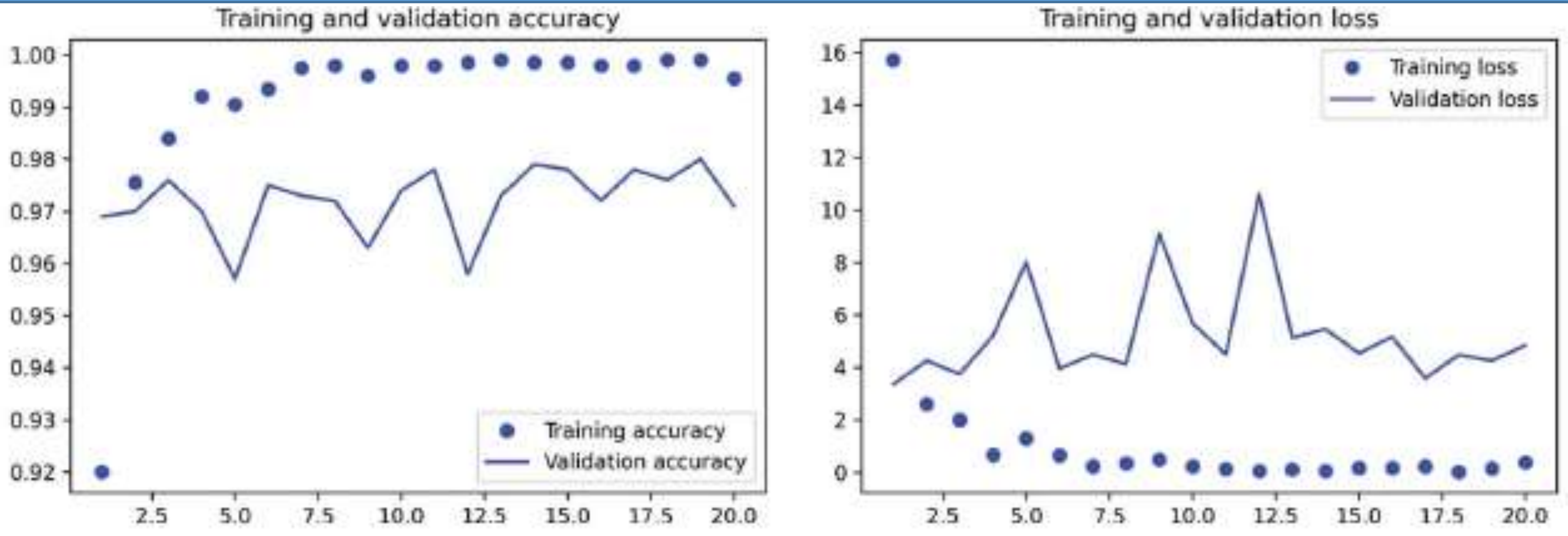


Figure 8.13 Training and validation metrics for plain feature extraction

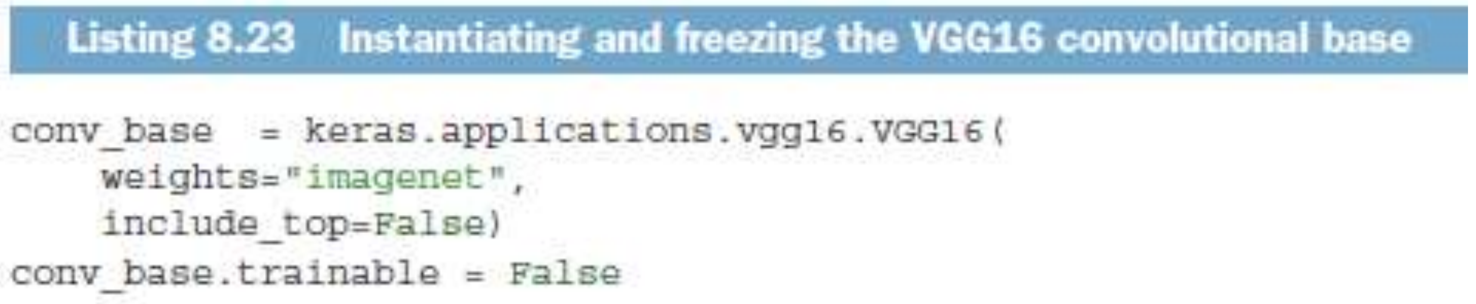
- You reach **a validation accuracy of about 90%—**much better than you achieved in the previous section with the small model trained from scratch.
- But the plots also indicate that **you're overfitting** almost **from the start**—despite using **dropout with a fairly large rate**.
- That's because this technique **doesn't use data augmentation**, which is essential for preventing overfitting with small image datasets.

Feature Extraction with Data Augmentation

- For instance, you might train a network on **ImageNet** (where classes are mostly **animals** and everyday objects) and then repurpose this trained network for something as remote as **identifying furniture** items in images.
- In this case, let’s consider a large convnet trained on the ImageNet dataset
- (1.4 million labeled images and 1,000 different classes).
- ImageNet contains many animal classes, including different species of cats and dogs, and you can thus expect to perform well on the dogs-versus-cats classification problem.
- You’ll use the **VGG16 architecture.**

Feature Extraction with Data Augmentation

- Now let’s review the second technique for doing feature extraction, which is much slower and more expensive.
- But which allows us to use data augmentation during training: creating a model that chains the conv_base with a new dense classifier, and training it end to end on the inputs.
- In order to do this, we will first freeze the convolutional base.
- Freezing a layer or set of layers means preventing their weights from being updated during training.
- In Keras, we freeze a layer or model by setting its trainable attribute to False.

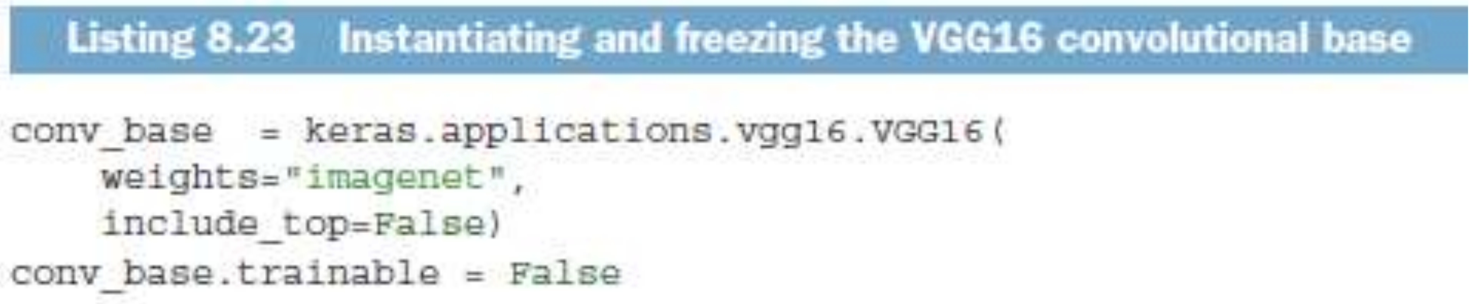


Listing 8.23 Instantiating and freezing the VGG16 convolutional base

```
conv_base  = keras.applications.vgg16.VGG16(
    weights="imagenet",
    include_top=False)
conv_base.trainable = False
```

Feature Extraction with Data Augmentation

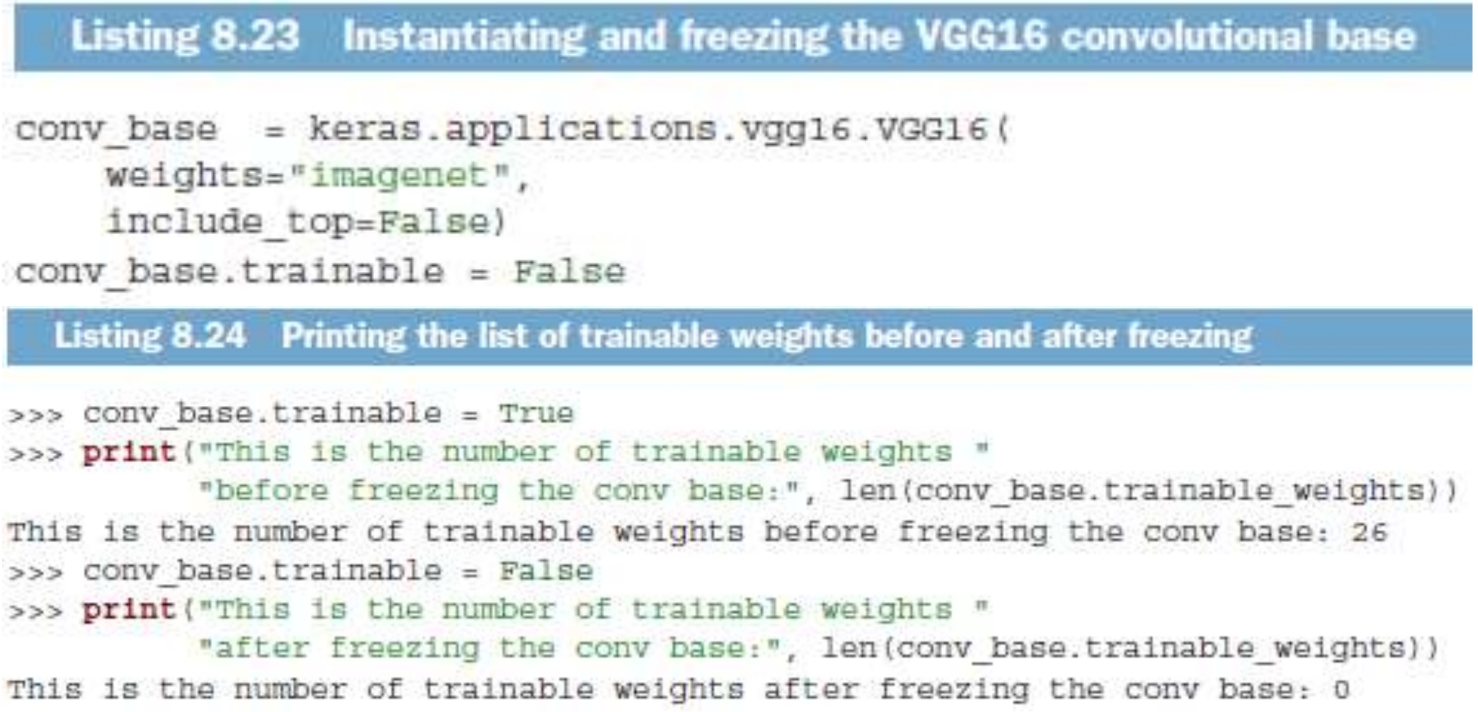
- Now let’s review the second technique for doing feature extraction, which is much slower and more expensive.
- But which allows us to use data augmentation during training: creating a model that chains the conv_base with a new dense classifier, and training it end to end on the inputs.
- In order to do this, we will first freeze the convolutional base.
- Freezing a layer or set of layers means preventing their weights from being updated during training.
- In Keras, we freeze a layer or model by setting its trainable attribute to False.



Listing 8.23 Instantiating and freezing the VGG16 convolutional base

```
conv_base  = keras.applications.vgg16.VGG16(
    weights="imagenet",
    include_top=False)
conv_base.trainable = False
```

Feature Extraction with Data Augmentation



Listing 8.23 Instantiating and freezing the VGG16 convolutional base

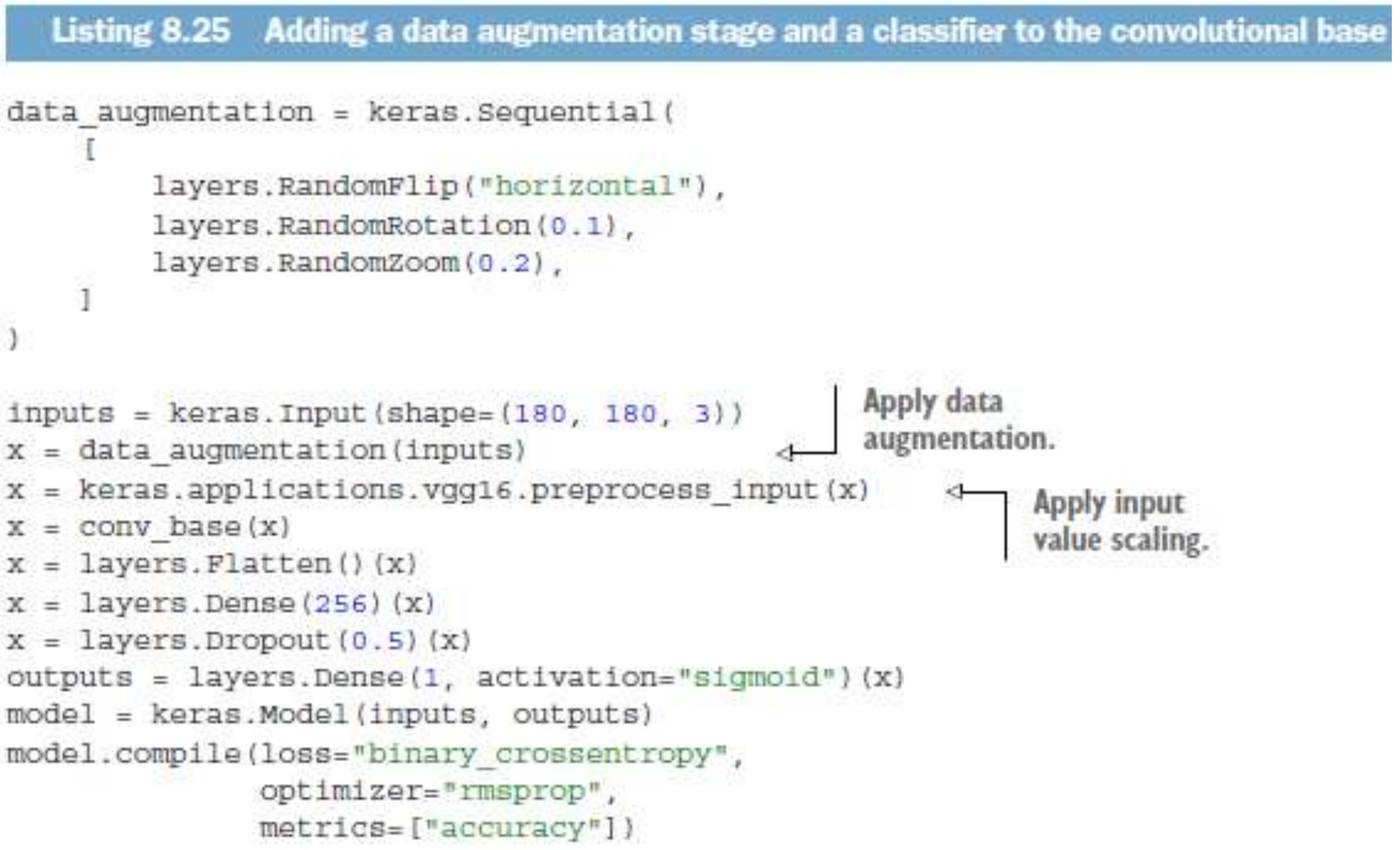
```
conv_base  = keras.applications.vgg16.VGG16(
    weights="imagenet",
    include_top=False)
conv_base.trainable = False
```

Listing 8.24 Printing the list of trainable weights before and after freezing

```
>>> conv_base.trainable = True
>>> print("This is the number of trainable weights "
          "before freezing the conv base:", len(conv_base.trainable_weights))
This is the number of trainable weights before freezing the conv base: 26
>>> conv_base.trainable = False
>>> print("This is the number of trainable weights "
          "after freezing the conv base:", len(conv_base.trainable_weights))
This is the number of trainable weights after freezing the conv base: 0
```

- Now we can create a new model that chains together
  - A data augmentation stage
  - Our frozen convolutional base
  - A dense classifier

Feature Extraction with Data Augmentation



Listing 8.25 Adding a data augmentation stage and a classifier to the convolutional base

```
data_augmentation = keras.Sequential(
    [
        layers.RandomFlip("horizontal"),
        layers.RandomRotation(0.1),
        layers.RandomZoom(0.2),
    ]
)

inputs = keras.Input(shape=(180, 180, 3))
x = data_augmentation(inputs)
x = keras.applications.vgg16.preprocess_input(x)
x = conv_base(x)
x = layers.Flatten()(x)
x = layers.Dense(256)(x)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(1, activation="sigmoid")(x)
model = keras.Model(inputs, outputs)
model.compile(loss="binary_crossentropy",
              optimizer="rmsprop",
              metrics=["accuracy"])
```

Apply data augmentation.

Apply input value scaling.

Feature Extraction with Data Augmentation

```
callbacks = [
    keras.callbacks.ModelCheckpoint(
        filepath="feature_extraction_with_data_augmentation.keras",
        save_best_only=True,
        monitor="val_loss")
]
```

```
history = model.fit(
    train_dataset,
    epochs=50,
    validation_data=validation_dataset,
    callbacks=callbacks)
```

Listing 8.26 Evaluating the model on the test set

```
test_model = keras.models.load_model(
    "feature_extraction_with_data_augmentation.keras")
test_loss, test_acc = test_model.evaluate(test_dataset)
print(f"Test accuracy: {test_acc:.3f}")
```

Feature Extraction with Data Augmentation

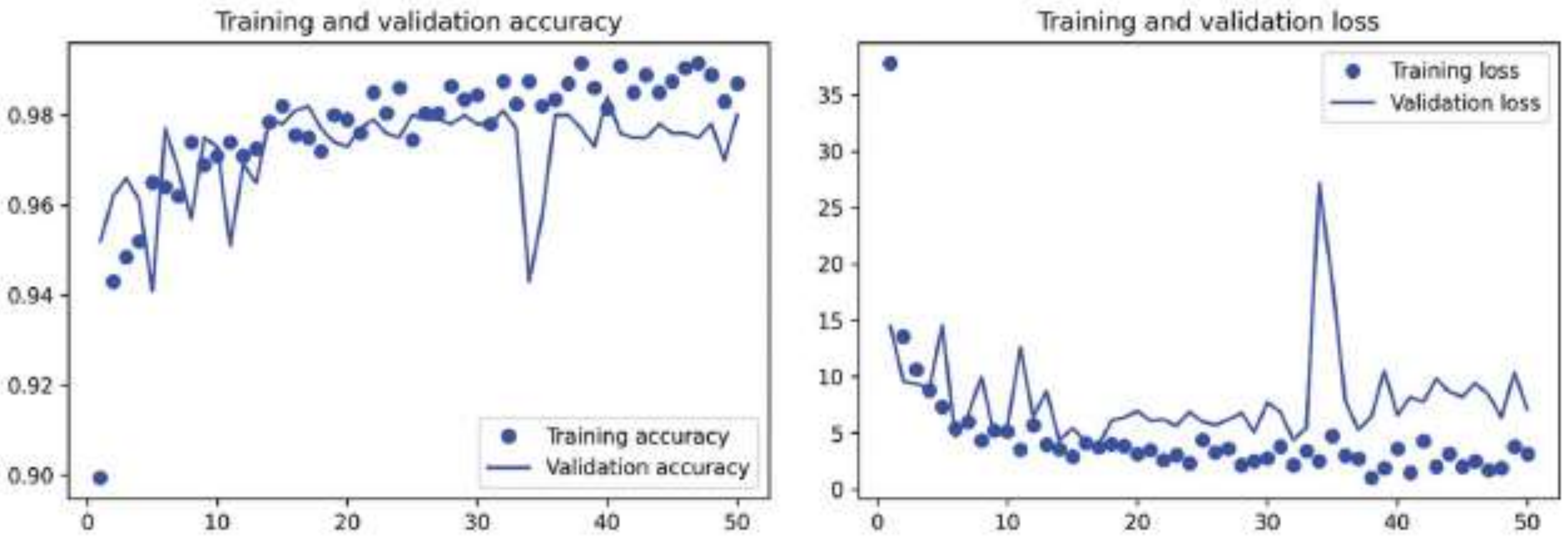


Figure 8.14 Training and validation metrics for feature extraction with data augmentation

Feature Extraction with Data Augmentation (3rd milestone)

- We get a **test accuracy of 97.5%**.
- This is only a modest improvement compared to the previous test accuracy, which is a **bit disappointing** given the strong results on the validation data.
- A model's accuracy always depends on the set of samples you evaluate it on!
- Some sample sets may be more difficult than others, and strong results on one set won't necessarily fully translate to all other sets.

Fine-tuning a pretrained model

- Another widely used technique for model reuse, complementary to feature extraction, is **fine-tuning** (see figure 8.15).
- Fine-tuning consists of **unfreezing a few of the top layers** of a frozen model base used for feature extraction, and jointly training both the newly added part of the model (in this case, the fully connected classifier) and these top layers.
- This is called fine-tuning because it slightly adjusts the more **abstract representations** of the model being reused in order to make them **more relevant for the problem at hand**.

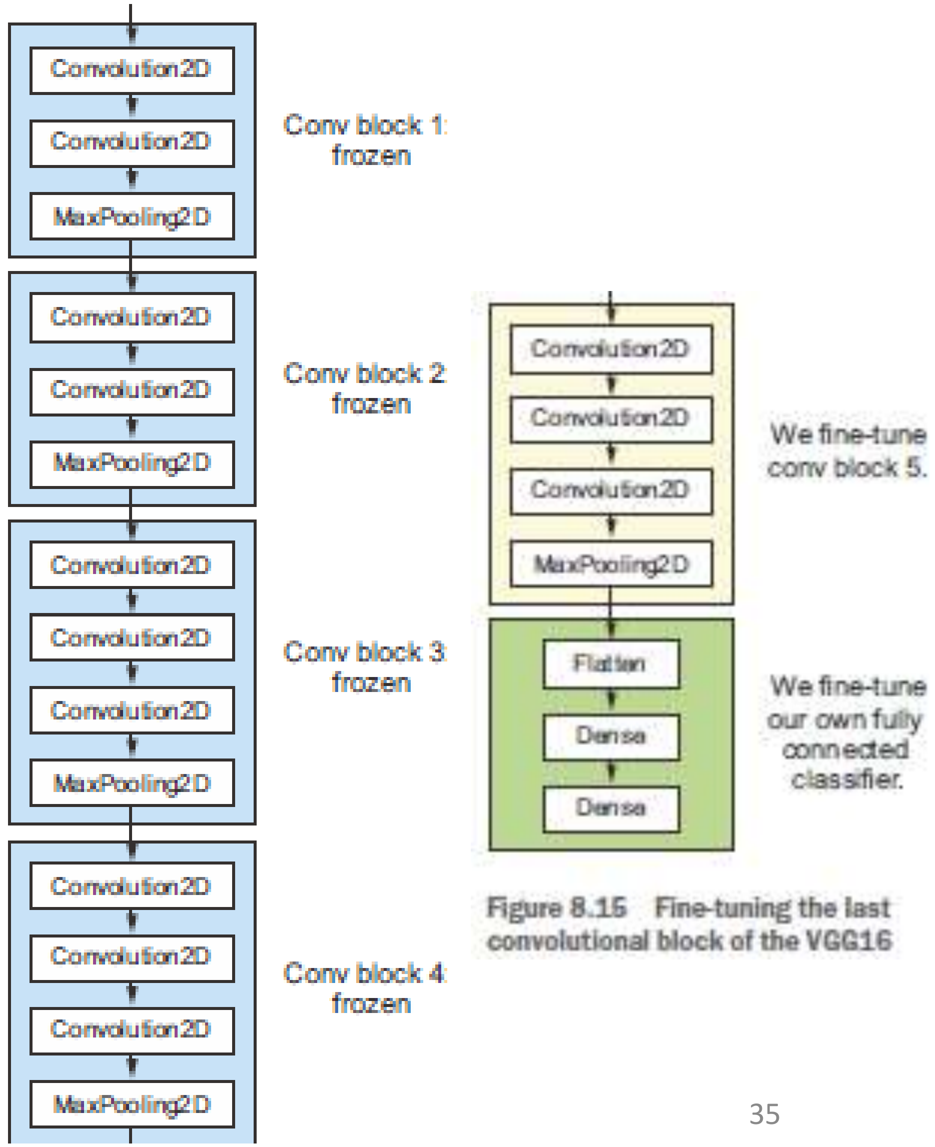


Figure 8.15 Fine-tuning the last convolutional block of the VGG16

Fine-tuning a pretrained model

- The steps for fine-tuning a network are as follows:
  1. Add our custom network on top of an already-trained base network.
  2. Freeze the base network.
  3. Train the part we added.
  4. Unfreeze some layers in the base network. (Note that you should not unfreeze “batch normalization” layers, which are not relevant here since there are no such layers in VGG16. Batch normalization and its impact on finetuning is explained in the next chapter.)
  5. Jointly train these layers and the part we added.
- You already **completed the first three steps** when doing feature extraction.
- Let’s proceed with step 4: **we’ll unfreeze our conv_base** and then freeze individual layers inside it.

Fine-tuning a pretrained model

- The steps for fine-tuning a network are as follows:
  1. Add our custom network on top of an already-trained base network.
  2. Freeze the base network.
  3. Train the part we added.
  4. Unfreeze some layers in the base network. (Note that you should not unfreeze “batch normalization” layers, which are not relevant here since there are no such layers in VGG16. Batch normalization and its impact on finetuning is explained in the next chapter.)
  5. Jointly train these layers and the part we added.
- You already **completed the first three steps** when doing feature extraction.
- Let’s proceed with step 4: **we’ll unfreeze our conv_base** and then freeze individual layers inside it.
- We’ll fine-tune the last three convolutional layers, which means all layers up to block4_pool should be frozen, and the layers block5_conv1, block5_conv2, and block5_conv3 should be trainable.

Fine-tuning a pretrained model

- **Why not fine-tune more layers**? Why not fine-tune the entire convolutional base? You could. But you need to consider the following:
  - **Earlier layers** in the convolutional base **encode more generic**, **reusable features**, whereas layers higher up encode more specialized features.
  - It’s more useful to **fine-tune the more specialized features**, because these are the ones that need to be repurposed on your new problem. There would be fast-decreasing returns in fine-tuning lower layers.
  - The **more parameters** you’re training, the more you’re at **risk of overfitting**.
  - The **convolutional base has 15 million** parameters, so it would be risky to attempt to train it on your small dataset.
- Thus, in this situation, **it’s a good strategy** to **fine-tune** only the **top two or three layers** in the convolutional base.
- Let’s set this up, starting from where we left off in the previous example.

Fine-tuning a pretrained model

Listing 8.27 Freezing all layers until the fourth from the last

```
conv_base.trainable = True
for layer in conv_base.layers[:-4]:
    layer.trainable = False
```

Listing 8.28 Fine-tuning the model

```
model.compile(loss="binary_crossentropy",
              optimizer=keras.optimizers.RMSprop(learning_rate=1e-5),
              metrics=["accuracy"])

callbacks = [
    keras.callbacks.ModelCheckpoint(
        filepath="fine_tuning.keras",
        save_best_only=True,
        monitor="val_loss")
]
history = model.fit(
    train_dataset,
    epochs=30,
    validation_data=validation_dataset,
    callbacks=callbacks)
```

We can finally evaluate this model on the test data:

```
model = keras.models.load_model("fine_tuning.keras")
test_loss, test_acc = model.evaluate(test_dataset)
print(f"Test accuracy: {test_acc:.3f}")
```

Fine-tuning a pretrained model (final accuracy)

- Here, we get a **test accuracy of 98.5%**
- In the original Kaggle competition around this dataset, **this would have been one of the top results**. It's **not quite a fair comparison**, however, since **we used pretrained features** that already contained prior knowledge about cats and dogs, **which competitors couldn't use** at the time.
- On the positive side, by **leveraging modern deep learning techniques**, we managed to reach this result using **only a small fraction of the training data** that was available for the competition (**about 10%**).
- There is a **huge difference between** being able to train on **20,000 samples** compared to **2,000 samples**!
- Now you have a solid set of tools for dealing with image-classification problems—in particular, with small datasets.

More CNN Example codes

- Run and understand CIFAR classification using CNN
- Run and understand MNIST-fashion classification using CNN

Summary

- A step by step solution to train for classification models
  - With small datasets
  - Also most techniques can also be used for large datasets
- Data augmentation is used to overcome overfitting with small datasets.
- Regularization can be useful for both small and large datasets.
- Pre-trained models play a vital role for deep learning.
- You can fine tune pre-trained models.