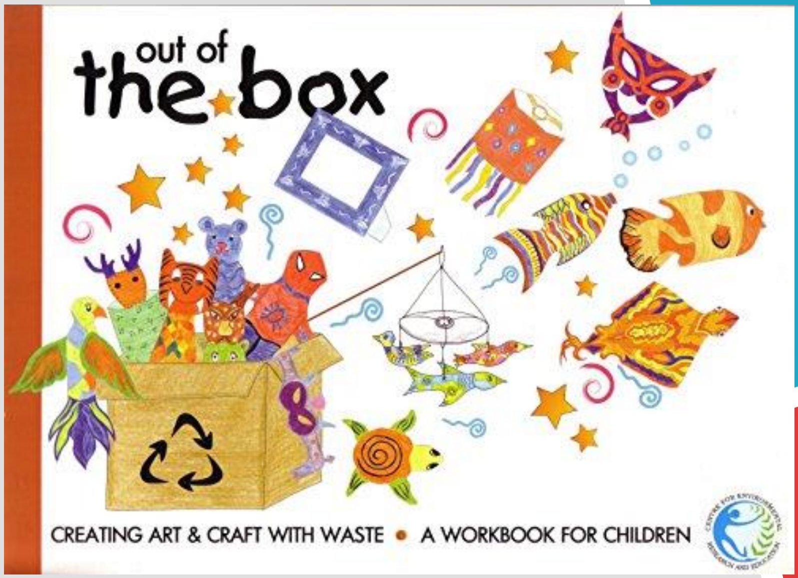
5장 *Deep learning for computer vision*

"Out of the Box"



This chapter covers

- Understanding convolutional neural networks (convnets)
- Using data augmentation to mitigate overfitting
- Using a pretrained convnet to do feature extraction
- Fine-tuning a pretrained convnet
- Visualizing what convnets learn and how they make classification decisions

5.1 Introduction to convnets

- a convnet to classify MNIST digits - its accuracy will blow out of the water that of the densely connected model
- a basic convnet - a stack of Conv2D and MaxPooling2D layers

Listing 5.1 Instantiating a small convent

```
from keras import layers
from keras import models
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu',    # The number of channels
                        input_shape=(28, 28, 1))) # (image_height, image_width, image_channels)
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
# padding = 'valid', stride = 1
>>> model.summary()
Layer (type)                     Output Shape          Param #
================================================================
conv2d_1 (Conv2D)                (None, 26, 26, 32)    320
maxpooling2d_1 (MaxPooling2D)    (None, 13, 13, 32)    0
conv2d_2 (Conv2D)                (None, 11, 11, 64)    18496
maxpooling2d_2 (MaxPooling2D)    (None, 5, 5, 64)      0
conv2d_3 (Conv2D)                (None, 3, 3, 64)      36928
================================================================
Total params: 55,744
Trainable params: 55,744
Non-trainable params: 0
```

5.1 Introduction to convnets

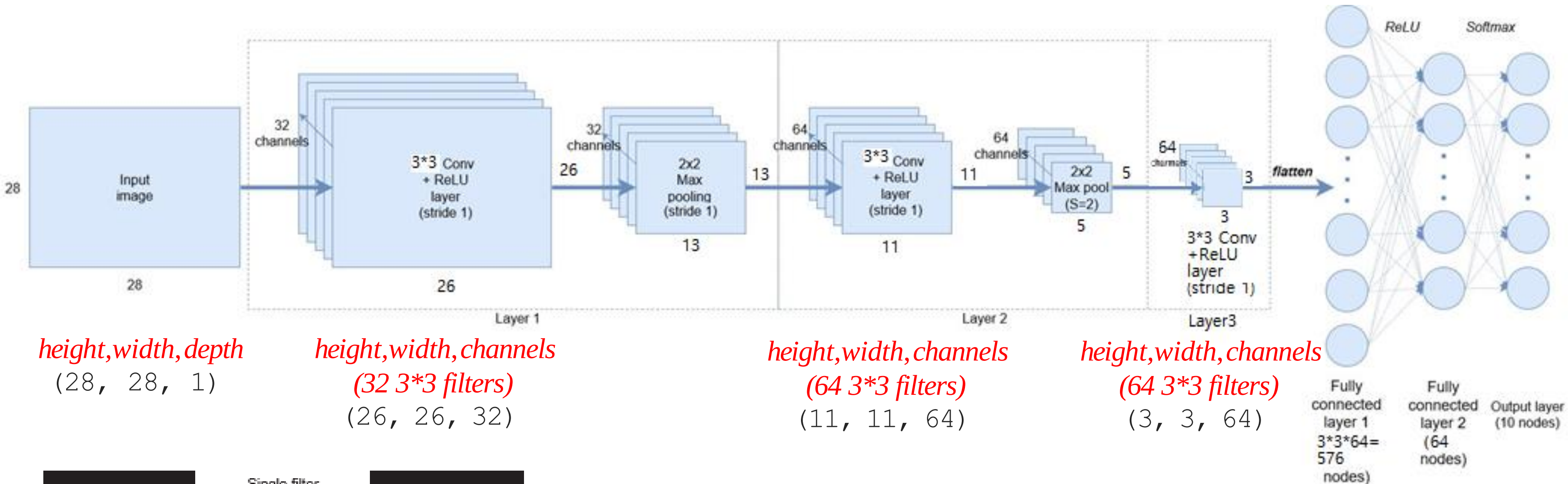
- flatten the 3D outputs to 1D, and then add a few `Dense` layers on top.
- feed the last output tensor (of shape `(3, 3, 64)`) into a densely connected classifier network: a stack of `Dense` layers.

Listing 5.2 Adding a classifier on top of the convnet

```
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
```

```
>>> model.summary()
Layer (type)                    Output Shape        Param #
================================================================
conv2d_1 (Conv2D)               (None, 26, 26, 32) 320
maxpooling2d_1 (MaxPooling2D) (None, 13, 13, 32) 0
conv2d_2 (Conv2D)               (None, 11, 11, 64) 18496
maxpooling2d_2 (MaxPooling2D) (None, 5, 5, 64)   0
conv2d_3 (Conv2D)               (None, 3, 3, 64)   36928
flatten_1 (Flatten)             (None, 576)         0
dense_1 (Dense)                 (None, 64) 36928
dense_2 (Dense)                 (None, 10) 650
================================================================
```

5.1Introduction to convnets



height,width,depth
(28, 28, 1)

height,width,channels
*(32 3*3 filters)*
(26, 26, 32)

height,width,channels
*(64 3*3 filters)*
(11, 11, 64)

height,width,channels
*(64 3*3 filters)*
(3, 3, 64)

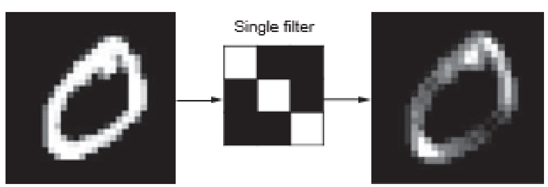


Figure 5.3 The concept of a response map

Architecture of the Convolutional neural network

5.1 Introduction to convnets

Listing 5.3 Training the convnet on MNIST images

```
from keras.datasets import mnist
from keras.utils import to_categorical
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
train_images = train_images.reshape((60000, 28, 28, 1))
train_images = train_images.astype('float32') / 255
test_images = test_images.reshape((10000, 28, 28, 1))
test_images = test_images.astype('float32') / 255
train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)

model.compile(optimizer='rmsprop',
     loss='categorical_crossentropy', metrics=['accuracy'])
model.fit(train_images, train_labels, epochs=5, batch_size=64)
```

- Let's evaluate the model on the test data:

```
>>> test_loss, test_acc = model.evaluate(test_images, test_labels)
>>> test_acc
0.9908
```

- Whereas the densely connected network had a test accuracy of 97.8%, the basic convnet has a test accuracy of 99.3%: we decreased the error rate by 68% (relative). Not bad!
- Why? Let's dive into what the `Conv2D` and `MaxPooling2D` layers do.

5.1 Introduction to convnets

5.1.1 The convolution operation

- `Dense` layers - learn global patterns in their input feature space (for example, for a MNIST digit, patterns involving all pixels),
- convolution layers - learn local patterns (see figure 5.1): in the case of images, patterns found in small 2D windows (3×3, etc.) of the inputs.
- This key characteristic gives convnets two interesting properties:
  - *translation invariant* - After learning a certain pattern in the lower-right corner of a picture, a convnet can recognize it anywhere.
  - A densely connected network would have to learn the pattern a new if it appeared at anew location ***vs.*** Convnets need fewer training samples to learn representations that have generalization power.

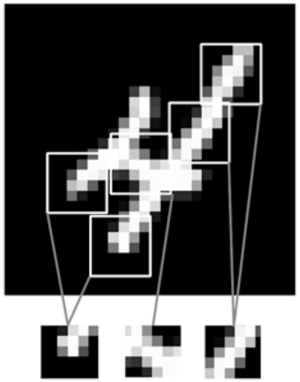
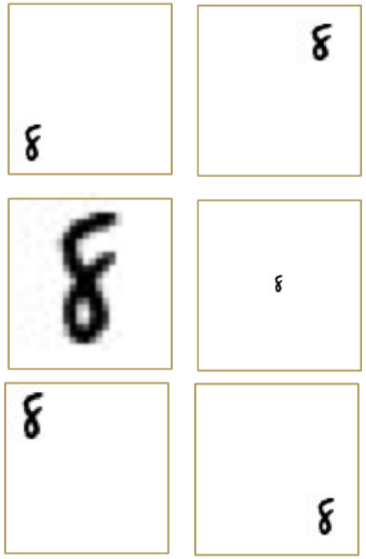


Figure 5.1 Images can be broken into local patterns such as edges, textures, and so on.



5.1 Introduction to convnets

5.1.1 The convolution operation

- *They can learn spatial hierarchies of patterns (see figure 5.2).*
- A first convolution layer - learn small local patterns such as edges,
- A second convolution layer - learn larger patterns made of the features of the first layers. This allows convnets to efficiently learn increasingly complex and abstract visual concepts (because *the visual world is fundamentally spatially hierarchical*).

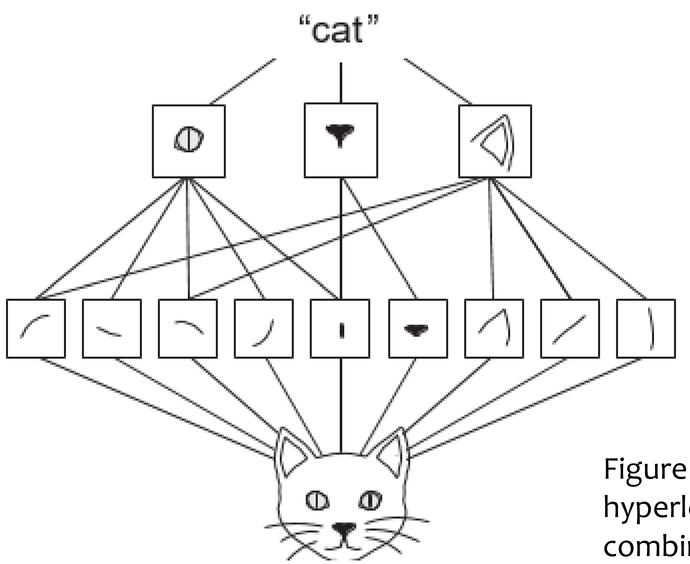


Figure 5.2 The visual world forms a spatial hierarchy of visual modules: hyperlocal edges combine into local objects such as eyes or ears, which combine into high-level concepts such as "cat."

5.1 Introduction to convnets

5.1.1 The convolution operation

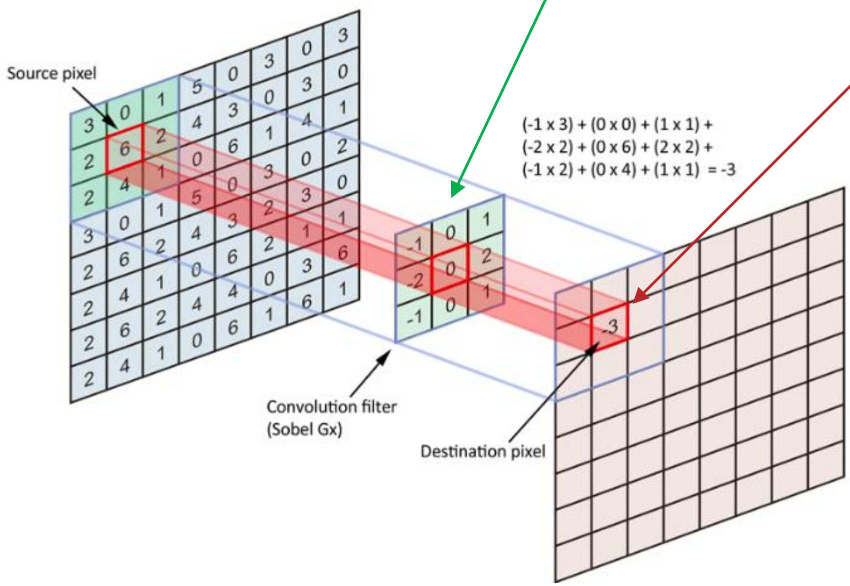
- The convolution operation: input feature map → *output feature* with 3D tensor (width, height, channels)
- The first convolution layer in the MNIST : a feature map of size `(28, 28, 1)` → outputs a feature map of size `(26, 26, 32)`
- Convolutions are defined by two key parameters:
  - *Size of the filters* — 3 × 3 or 5 × 5
  - *number of filters* — 32 or 64
- In Keras `Conv2D` layers, these parameters are the first arguments passed to the layer:

```
model.add(layers.Conv2D(32, (3, 3), activation='relu'))
```

5.1 Introduction to convnets

5.1.1 The convolution operation

- 3D output map of shape `(height, width, output_depth)` → 1D vector of shape `(output_depth,)`
- For instance, with 3 × 3 windows, the vector `output[i, j, :]` comes from the 3D patch `input[i-1:i+1, j-1:j+1, :]`
  - Border effects, which can be countered by padding the input feature map
  - The use of *strides*, which I'll define in a second



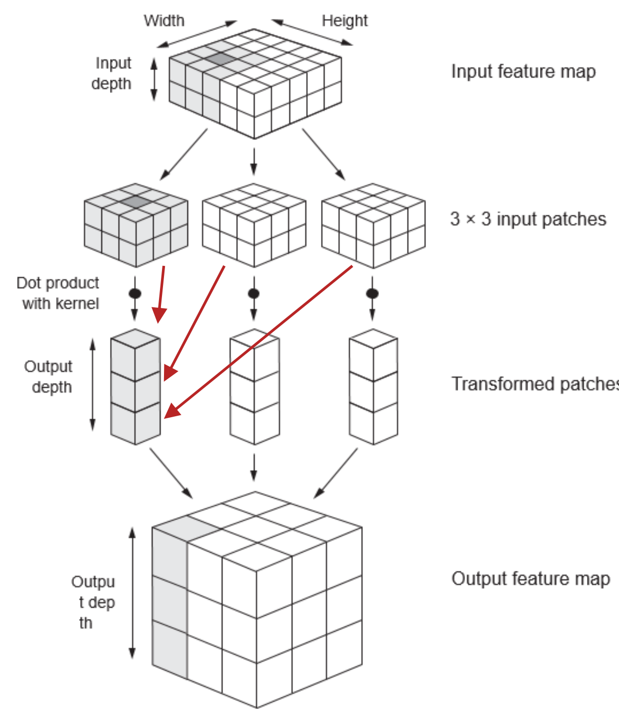


Figure 5.4 How convolution works

5.1.1 The convolution operation

UNDERSTANDING BORDER EFFECTS AND PADDING

- border effect - 5 × 5 feature map (25 tiles total) → output feature map 3 × 3 by 3 × 3 filter
- 28 × 28 inputs → 26 × 26 after the first convolution
- `padding` argument: "valid", which means no padding (only valid window locations will be used); and "same", which means "pad in such a way as to have an output with the same width and height as the input." The `padding` argument defaults to "valid".

```
model.add(layers.Conv2D(32, (3, 3), padding='valid',
                input_shape=(28, 28, 1), activation='relu'))
```

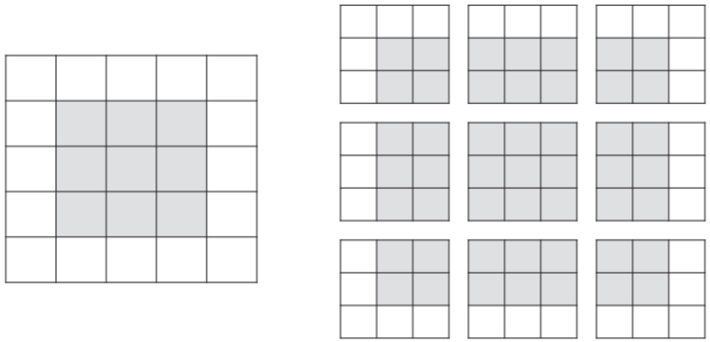


Figure 5.5 Valid locations of 3 × 3 patches in a 5 × 5 input feature map

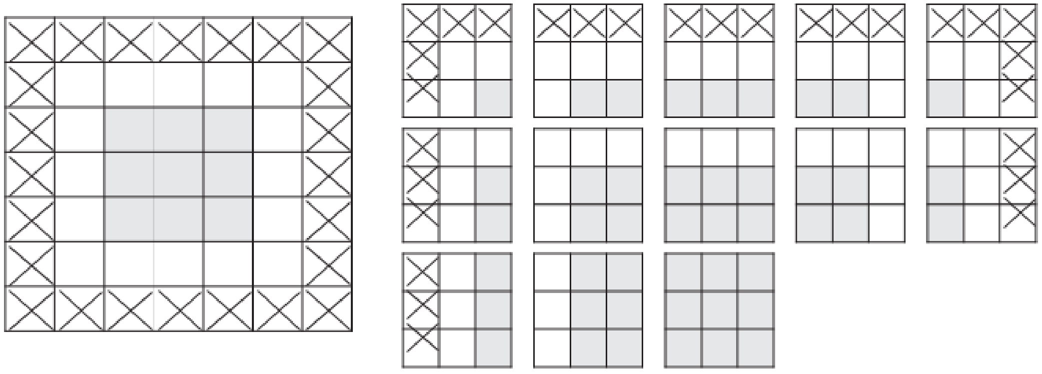


Figure 5.6 Padding a 5 × 5 input in order to be able to extract 25 3 × 3 patches

5.1 Introduction to convnets

5.1.1 The convolution operation

UNDERSTANDING CONVOLUTION STRIDES

- *strides* - The distance between two successive windows is a parameter of the convolution, called its *stride*, which defaults to 1.
- It's possible to have *strided convolutions* : 3 × 3 convolution with stride 2 over a 5 × 5 input (without padding).

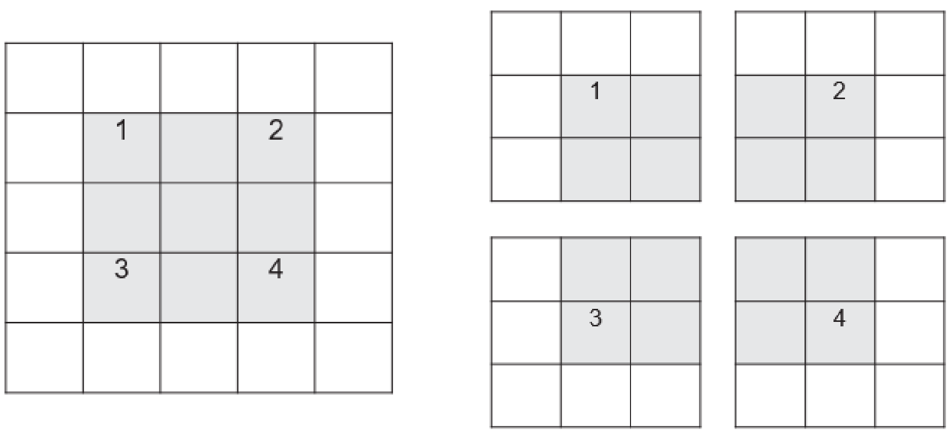


Figure 5.7 3 × 3 convolution patches with 2 × 2 strides

5.1 Introduction to convnets

5.1.2 The max-pooling operation

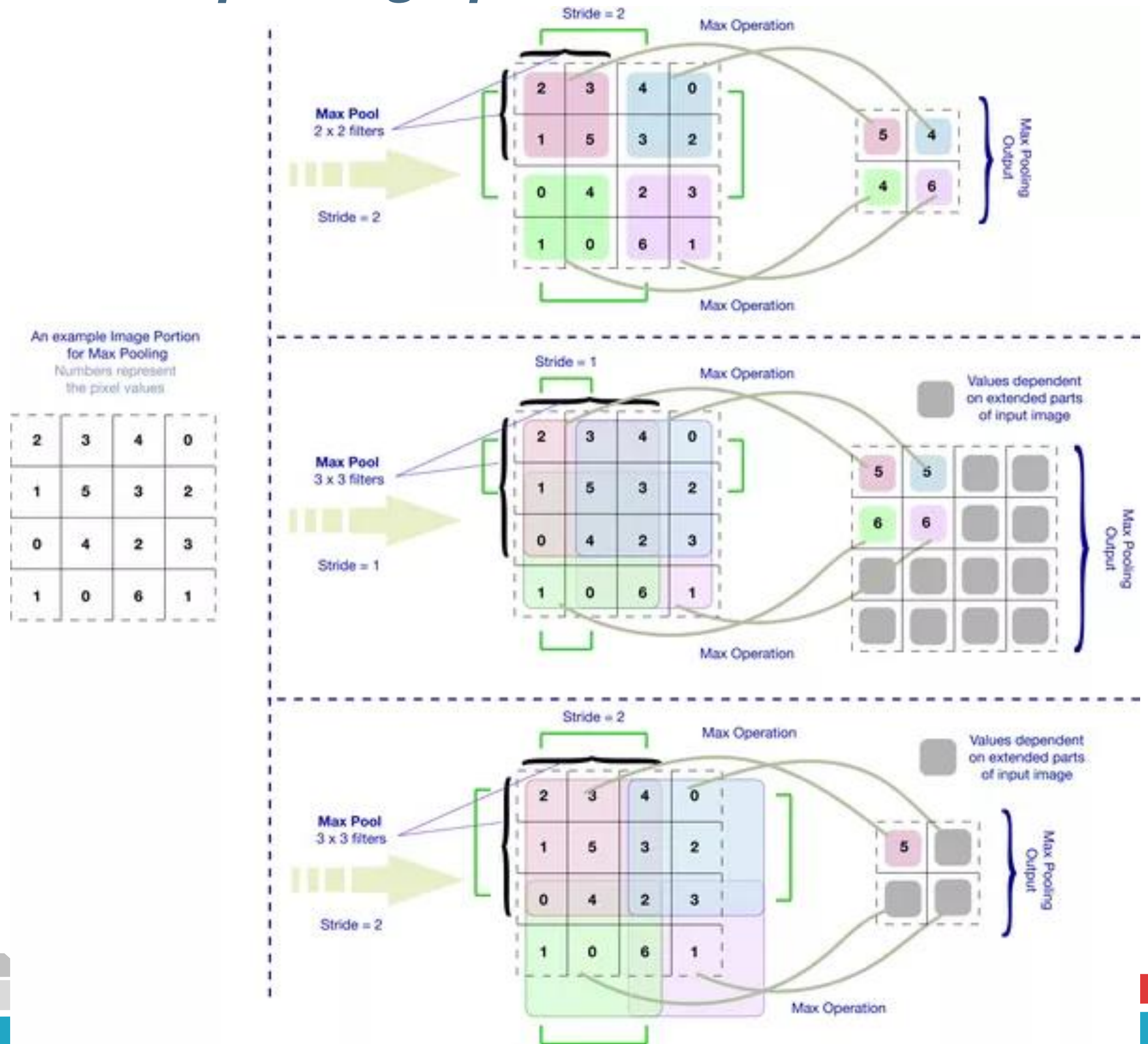
- max-pooling operation - feature map 26 × 26 → 13 × 13
- role of max pooling: aggressively downsample feature maps, much like strided convolutions.
- max pooling is usually done with 2 × 2 windows and stride 2.

```
model.add(MaxPooling2D(pool_size=(2, 2),
                       strides=(2, 2)))
```

- contain information about the totality of the input

5.1 Introduction to convnets

5.1.2 The max-pooling operation



5.2 Training a convnet from scratch on a small dataset

- classifying images as dogs or cats - 4,000 pictures of cats and dogs (2,000 cats, 2,000 dogs)
- 2,000 pictures for training—1,000 for validation, and 1,000 for testing.
- 2,000 training samples - classification accuracy of 71%
- data augmentation - mitigating overfitting, 82%.
- feature extraction with a pretrained network - accuracy of 90% to 96%
- fine-tuning a pretrained network - final accuracy of 97%

5.2 Training a convnet from scratch on a small dataset

5.2.1 The relevance of deep learning for small-data problems

- convnets learn local, translation-invariant without the need for any custom feature engineering
- deep-learning models are by nature highly repurposable - an image-classification or speech-to-text model trained on a large-scale dataset and reuse it on a significantly different problem with only minor changes.
- many pretrained models (usually trained on the Image-Net dataset) are now publicly available for download and can be used to bootstrap powerful vision models out of very little data.

5.2 Training a convnet from scratch on a small dataset

5.2.2 Downloading the data

- Dogs vs. Cats dataset - Kaggle as part of a computer-vision competition in late 2013, won by entrants who used convnets (95% accuracy)
- download the original dataset from

  www.kaggle.com/c/dogs-vs-cats/data
- The pictures are medium-resolution color JPEGs. Figure 5.8 shows some examples.

Figure 5.8 Samples from the Dogs vs. Cats dataset. Sizes weren't modified: the samples are heterogeneous in size, appearance, and so on.

5.2 Training a convnet from scratch on a small dataset

5.2.2 Downloading the data

- In this example, you'll train your models on less than 10% of the data that was available to the competitors.
- This dataset contains 25,000 images of dogs and cats (12,500 from each class) and is 543 MB (compressed).
- training set - 1,000 samples * 2 class
- validation set with 500 samples * 2 class
- test set with 500 samples * 2 class

5.2 Training a convnet from scratch on a small dataset

5.2.2 Downloading the data

Listing 5.4 Training the convnet on MNIST images

```
import os, shutil
original_dataset_dir = './datasets/cats_and_dogs/train' # 원본 데이터셋
base_dir = './datasets/cats_and_dogs_small' # 소규모 데이터셋
os.mkdir(base_dir)
train_dir = os.path.join(base_dir, 'train') # 훈련, 검증, 테스트 분할
os.mkdir(train_dir)
validation_dir = os.path.join(base_dir, 'validation')
os.mkdir(validation_dir)
test_dir = os.path.join(base_dir, 'test')
os.mkdir(test_dir)
train_cats_dir = os.path.join(train_dir, 'cats') # 훈련용 고양이
os.mkdir(train_cats_dir)
train_dogs_dir = os.path.join(train_dir, 'dogs') # 훈련용 강아지
os.mkdir(train_dogs_dir)
validation_cats_dir = os.path.join(validation_dir, 'cats') # 검증용 고양이
os.mkdir(validation_cats_dir)
validation_dogs_dir = os.path.join(validation_dir, 'dogs') # 검증용 강아지
os.mkdir(validation_dogs_dir)
test_cats_dir = os.path.join(test_dir, 'cats') # 테스트용 고양이
os.mkdir(test_cats_dir)
test_dogs_dir = os.path.join(test_dir, 'dogs') # 테스트용 강아지
os.mkdir(test_dogs_dir)
```

5.2 Training a convnet from scratch on a small dataset

5.2.2 Downloading the data

Listing 5.4 Training the convnet on MNIST images

```
fnames=['cat.{}.jpg'.format(i) for i in range(1000)] # 처음 1,000개의 고양이 이미지
for fname in fnames:
    src = os.path.join(original_dataset_dir, fname)
    dst = os.path.join(train_cats_dir, fname)
    shutil.copyfile(src, dst) # train_cats_dir에 복사
fnames=['cat.{}.jpg'.format(i) for i in range(1000, 1500)]#다음 500개 고양이 이미지
for fname in fnames:
    src = os.path.join(original_dataset_dir, fname)
    dst = os.path.join(validation_cats_dir, fname)
    shutil.copyfile(src, dst) # validation_cats_dir에 복사
fnames=['cat.{}.jpg'.format(i) for i in range(1500, 2000)] #다음 500개 고양이 이미지
for fname in fnames:
    src = os.path.join(original_dataset_dir, fname)
    dst = os.path.join(test_cats_dir, fname)
    shutil.copyfile(src, dst) # test_cats_dir에 복사
```

5.2 Training a convnet from scratch on a small dataset

5.2.2 Downloading the data

Listing 5.4 Training the convnet on MNIST images

```
fnames=['dog.{}.jpg'.format(i) for i in range(1000)] # 처음 1,000개의 강아지 이미지
for fname in fnames:
    src = os.path.join(original_dataset_dir, fname)
    dst = os.path.join(train_dogs_dir, fname)
    shutil.copyfile(src, dst) # train_dogs_dir에 복사
fnames=['dog.{}.jpg'.format(i) for i in range(1000, 1500)]#다음 500개 강아지 이미지
for fname in fnames:
    src = os.path.join(original_dataset_dir, fname)
    dst = os.path.join(validation_dogs_dir, fname)
    shutil.copyfile(src, dst) # validation_dogs_dir에 복사
fnames=['dog.{}.jpg'.format(i) for i in range(1500, 2000)]#다음 500개 강아지 이미지
for fname in fnames:
    src = os.path.join(original_dataset_dir, fname)
    dst = os.path.join(test_dogs_dir, fname)
    shutil.copyfile(src, dst) # test_dogs_dir에 복사
```

5.2 Training a convnet from scratch on a small dataset

5.2.2 Downloading the data

- As a sanity check, let's count how many pictures are in each training split (train/validation/test):

```
>>> print('total training cat images:', len(os.listdir(train_cats_dir)))
total training cat images: 1000
>>> print('total training dog images:', len(os.listdir(train_dogs_dir)))
total training dog images: 1000
>>> print('total validation cat images:', len(os.listdir(validation_cats_dir)))
total validation cat images: 500
>>> print('total validation dog images:', len(os.listdir(validation_dogs_dir)))
total validation dog images: 500
>>> print('total test cat images:', len(os.listdir(test_cats_dir)))
total test cat images: 500
>>> print('total test dog images:', len(os.listdir(test_dogs_dir)))
total test dog images: 500
```

- 2,000 training images
- 1,000 validation images
- 1,000 test images

5.2 Training a convnet from scratch on a small dataset

5.2.3 Building your network

- one more `Conv2D` + `MaxPooling2D` stage
- inputs of size 150×150 → feature maps of size 7×7 just before the `Flatten` layer.
- binary-classification problem - `Dense` layer of size 1 with a `sigmoid` activation.

5.2 Training a convnet from scratch on a small dataset

5.2.3 Building your network

Listing 5.5 Instantiating a small convnet for dogs vs. cats classification

```
from keras import layers from keras import models
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3),
      activation='relu', input_shape=(150, 150, 3))) # 148×148
model.add(layers.MaxPooling2D((2, 2)))              # 74×74
model.add(layers.Conv2D(64, (3, 3), activation='relu'))# 72×72
model.add(layers.MaxPooling2D((2, 2)))              # 36×36
model.add(layers.Conv2D(128, (3, 3), activation='relu'))# 34×34
model.add(layers.MaxPooling2D((2, 2)))              # 17×17
model.add(layers.Conv2D(128, (3, 3), activation='relu'))# 15×15
model.add(layers.MaxPooling2D((2, 2)))              # 7×7
model.add(layers.Flatten())                         # 6272
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

5.2 Training a convnet from scratch on a small dataset

5.2.3 Building your network

```
>>> model.summary()
________________________________________________________________
Layer (type)                 Output Shape              Param #
================================================================
conv2d_1 (Conv2D)            (None, 148, 148, 32)      896
________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 74, 74, 32)        0
________________________________________________________________
conv2d_2 (Conv2D)            (None, 72, 72, 64)        18496
________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 36, 36, 64)        0
________________________________________________________________
conv2d_3 (Conv2D)            (None, 34, 34, 128)       73856
________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 17, 17, 128)       0
________________________________________________________________
conv2d_4 (Conv2D)            (None, 15, 15, 128)       147584
________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 7, 7, 128)         0
________________________________________________________________
flatten_1 (Flatten)          (None, 6272)              0
________________________________________________________________
dense_1 (Dense)              (None, 512)               3211776
________________________________________________________________
dense_2 (Dense)              (None, 1)                 513
================================================================
Total params: 3,453,121
Trainable params: 3,453,121
Non-trainable params: 0
________________________________________________________________
```

5.2 Training a convnet from scratch on a small dataset

5.2.3 Building your network

- compilation step - `RMSprop` optimizer
- ended with one sigmoid unit - binary crossentropy as the loss

Listing 5.6 Configuring the model for training

```
from keras import optimizers
model.compile(loss='binary_crossentropy',
       optimizer=optimizers.RMSprop(lr=1e-4),
       metrics=['acc'])
```

5.2 Training a convnet from scratch on a small dataset

5.2.4 Data preprocessing

- 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].
- Keras has a module with image-processing helper tools, located at `keras.preprocessing.image`.
- class `ImageDataGenerator` - automatically turn image files on disk into batches of preprocessed tensors.

Listing 5.7 Using `ImageDataGenerator` to read images from directories

```
from keras.preprocessing.image import ImageDataGenerator
train_datagen = ImageDataGenerator(rescale=1./255) # Rescale
test_datagen = ImageDataGenerator(rescale=1./255)
train_generator = train_datagen.flow_from_directory(
        train_dir,    # 타깃 디렉터리
        target_size=(150, 150), # JPEG content to 150×150 RGB
        batch_size=20,
        class_mode='binary') # 이진 레이블, 2개 폴더-cats, dogs
validation_generator = test_datagen.flow_from_directory(
        validation_dir, target_size=(150, 150),
        batch_size=20, class_mode='binary')
```

5.2 Training a convnet from scratch on a small dataset

5.2.4 Data preprocessing

- generator yields these batches indefinitely: break the iteration loop at some point:

```
for data_batch, labels_batch in train_generator:
    print('data batch shape:', data_batch.shape)
    print('labels batch shape:', labels_batch.shape)
    break
data batch shape: (20, 150, 150, 3)
labels batch shape: (20,)
```

- fit_generator = fit – yield batches of inputs and targets indefinitely
- **steps_per_epoch**: 20 batches from the generator, 100 steps until you see target of 2,000 samples.
- **validation_steps**: 20 batches from the generator, 50 steps until you see validation of 1,000 samples.

Listing 5.8 Fitting the model using a batch generator

```
history = model.fit_generator(train_generator, # 20
    steps_per_epoch=100, # 20 batches*100 steps=2000
    epochs=30,
    validation_data=validation_generator,
    validation_steps=50) # 20 batches * 50 steps=1000
```

5.2 Training a convnet from scratch on a small dataset

5.2.4 Data preprocessing

Listing 5.9 Saving the model

```
model.save('cats_and_dogs_small_1.h5')
```

Listing 5.10 Displaying curves of loss and accuracy during training

```
acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(len(acc))
plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.legend()
plt.figure()
plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()
plt.show()
```

5.2 Training a convnet from scratch on a small dataset

5.2.4 Data preprocessing

- overfitting - training accuracy reaches nearly 100%, whereas the validation accuracy stalls at 70–72%.
- 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.
- relatively few training samples (2,000) - dropout and weight decay (L2 regularization), specific to computer vision: *data augmentation*

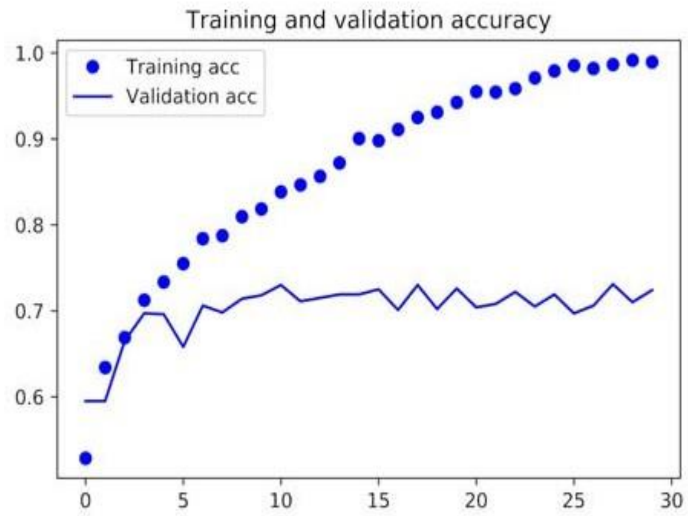


Figure 5.9 Training and validation accuracy

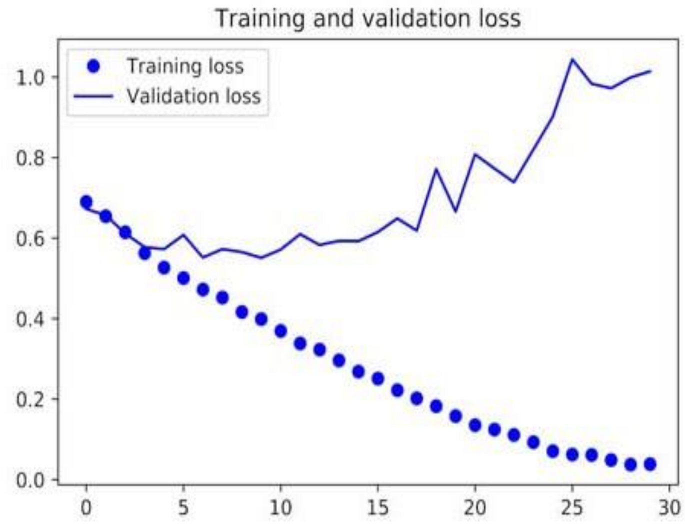


Figure 5.10 Training and validation loss

5.2 Training a convnet from scratch on a small dataset

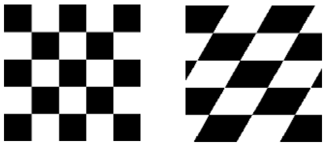
5.2.5 Using data augmentation

- Data augmentation - generating more training data via a number of random transformations
- expose the model to more aspects of the data and generalize better
- `ImageDataGenerator` instance - number of random transformations

training할 때 실시간으로 바꾸는 것
이미지를 바꿔놓고 학습하는게 아니고 (?)

Listing 5.11 Setting up a data augmentation configuration via ImageDataGenerator

```
datagen=ImageDataGenerator(rotation_range=40 #degrees
          width_shift_range=0.2,height_shift_range=0.2,
          shear_range=0.2, zoom_range=0.2,
          horizontal_flip=True, fill_mode='nearest')
```



shear

바꾸는 것과 학습하는 것의 순서를 명확하게
학습할 필요가 있는 것 같음(교수님 뉘앙스가..)

- These are just a few of the options available (for more, see the Keras documentation):
  - **rotation_range** - degrees (0–180), a range within which to randomly rotate pictures.
  - **width_shift** and **height_shift** - ranges (as a fraction of total width or height) within which to randomly translate pictures vertically or horizontally.
  - **shear_range** - randomly applying shearing transformations.
  - **zoom_range** - randomly zooming inside pictures.
  - **horizontal_flip** - randomly flipping half the images horizontally—relevant when there are no assumptions of horizontal asymmetry (for example, real-world pictures).
  - **fill_mode** is the strategy used for filling in newly created pixels, which can appear after a rotation or a width/height shift. {"constant", "nearest", "reflect" or "wrap"}.

5.2 Training a convnet from scratch on a small dataset

5.2.5 Using data augmentation

cat.100.jpg

Listing 5.12 Displaying some randomly augmented training images

```
from keras.preprocessing import image # 이미지 전처리 유틸리티 모듈
fnames = sorted([os.path.join(train_cats_dir, fname) for
        fname in os.listdir(train_cats_dir)])
img_path = fnames[3] # 증식할 이미지 선택
# 이미지를 읽고 크기 변경
img = image.load_img(img_path, target_size=(150, 150))
# (150, 150, 3) 크기의 넘파이 배열로 변환, [:,:,0:3] 반환
x = image.img_to_array(img)
x = x.reshape((1,)+x.shape) # (1,150,150,3) 크기로 변환
# flow() 메서드는 랜덤하게 변환된 이미지의 배치를 생성
# 무한 반복되기 때문에 어느 지점에서 중지해야 합니다!
i = 0
# flow-이미지를 배치 단위로(save_to_dir='폴더'로) 가져옴
for batch in datagen.flow(x, batch_size=1):
    plt.figure(i)
    imgplot = plt.imshow(image.array_to_img(batch[0]))
    i += 1
    if i % 4 == 0:
        break
plt.show()  # flow(data), flow_from_directory(directory)
```

이름 별로 sort하는게 아니고 나름대로 sort한다?..

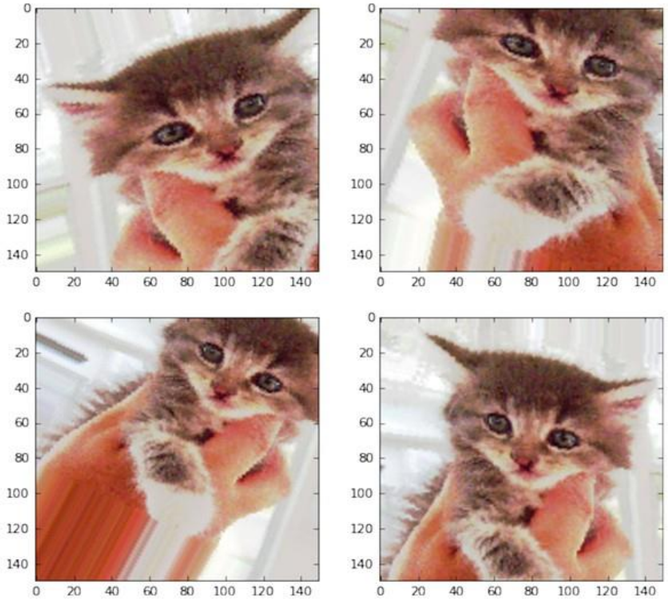


Figure 5.11 Generation of cat pictures via random data augmentation

5.2 Training a convnet from scratch on a small dataset

5.2.5 Using data augmentation

- data-augmentation - never prduce the same input twice.
- overfitting - remix existing inputs are still heavily intercorrelated
- add a `Dropout` layer to your model, right before the densely connected classifier

Listing 5.13 Defining a new convnet that includes dropout

```
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu',
           input_shape=(150, 150, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dropout(0.5)) # Flatten 다음, FCN 전
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy',
     optimizer=optimizers.RMSprop(lr=1e-4), metrics=['acc'])
```

5.2 Training a convnet from scratch on a small dataset

Listing 5.14 Training the convnet using data-augmentation

```
train_datagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=40,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True,)
test_datagen = ImageDataGenerator(rescale=1./255) # 검증 데이터는 증식하지 않음
train_generator = train_datagen.flow_from_directory(
        train_dir, # 타깃 디렉터리
        target_size=(150, 150), # 150 × 150 크기로 바꿉니다
        batch_size=32,
        class_mode='binary') # 이진 레이블
validation_generator = test_datagen.flow_from_directory(
        validation_dir,
        target_size=(150, 150),
        batch_size=32,
        class_mode='binary')
history = model.fit_generator(
      train_generator,
      steps_per_epoch=100,
      epochs=100,
      validation_data=validation_generator,
      validation_steps=50)
```

5.2 Training a convnet from scratch on a small dataset

Listing 5.15 Saving the model generators

```
model.save('cats_and_dogs_small_2.h5')
```

- data augmentation and dropout - no longer overfitting: the training curves are closely tracking the validation curves.
- accuracy of 82%, a 15% relative improvement over the non-regularized model.
- use a pretrained model

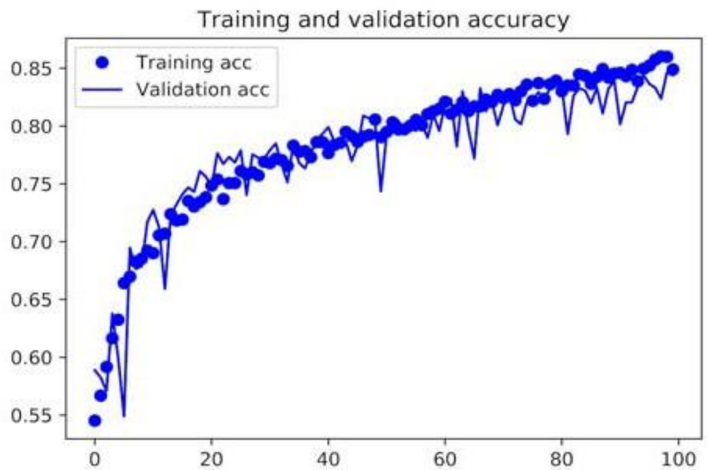


Figure 5.12 Training and validation accuracy with data augmentation

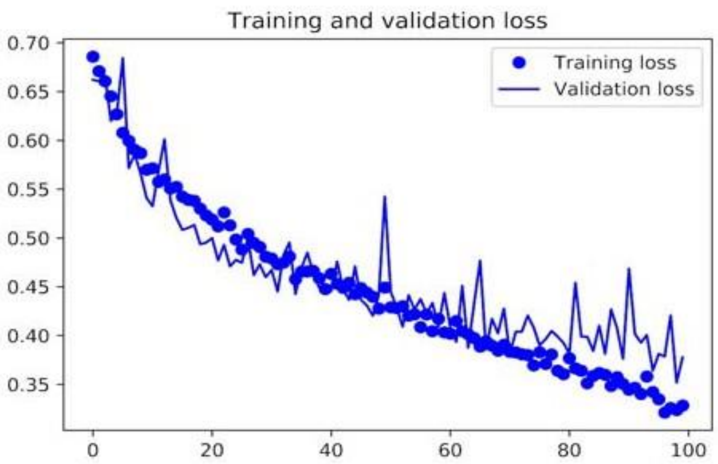


Figure 5.13 Training and validation loss with data augmentation

5.3 Using a pretrained convnet

이미 훈련된 convnet

- pretrained network - previously trained on a large dataset, typically on a large-scale image-classification task
- large and general enough dataset - generic model, useful for many different computer-vision problems
- train a network on ImageNet (1000 of classes, 1.4 million of images) - identifying furniture items in images
- a key advantage of deep learning - portability of learned features across different problems, very effective for small-data problems.

5.3Using a pretrained convnet

- ImageNet contains many animal classes, including different species of cats and dogs – perform well on the dogs-versus-cats classification problem.
- VGG16 convnet architecture for ImageNet - by Karen Simonyan and Andrew Zisserman in 2014, easy to understand without introducing any new concepts
- Previous works - VGG, ResNet, Inception, Inception-ResNet, Xception

하나 하나가 논문 두어개씩 나올 정도로 훌륭한 모델

- There are two ways to use a pretrained network: feature extraction and fine-tuning.

5.3 Using a pretrained convnet

5.3.1 Feature extraction

- Feature extraction - learned by a previous network to extract interesting features from new samples. These features are then run through a new classifier, which is trained from scratch.
- convnets used for image classification comprise two parts:
  - *convolutional base* - series of pooling and convolution layers
  - *densely connected classifier*
- Running the new data through it, and training a new classifier on top of the output (figure 5.14).

이거는 1000가지를 분류하는건데
우리는 2가지만 분류하면 되니깐 loss가 많음!
이거를 우리가 쓰기위해 잘 바꿔야함

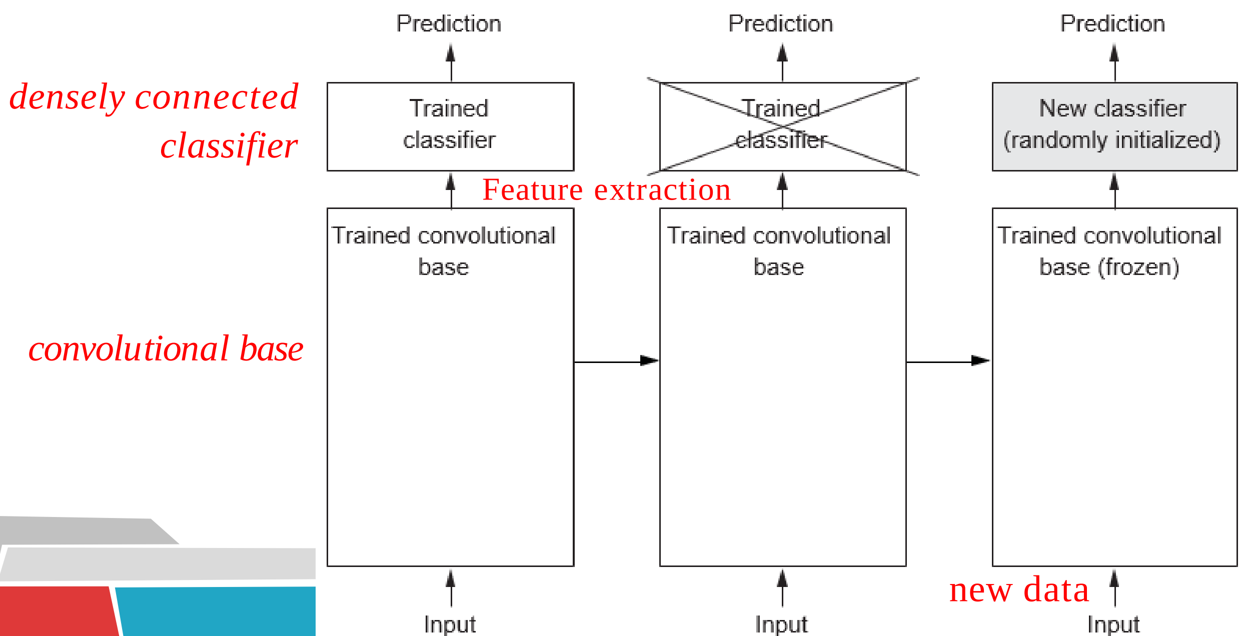


Figure 5.14 Swapping classifiers while keeping the same convolutional base

5.3 Using a pretrained convnet

5.3.1 Feature extraction

- Reuse the densely connected classifier? should be avoided.
- The feature maps of a convnet are presence maps of generic concepts over a picture.
- For problems where object location matters, densely connected features are largely useless.
- level of generality - depth of the layer in the model.
  - low layers - highly generic feature maps (such as visual edges, colors, and textures)

    lower레벨(점, 선, 곡선, 원 등…)
    -> higher레벨(귀, 눈, 코 등…)
    higher레벨의 모든 정보가 필요하지는 않다.
  - high layers - more-abstract concepts (such as “cat ear” or “dog eye”)
- New dataset differs a lot from the dataset on which the original model was trained – use the first few layers of the model to do feature extraction

5.3 Using a pretrained convnet

5.3.1 Feature extraction

- ImageNet class set contains multiple dog and cat classes - reuse the information contained in the densely connected layers of the original model.
- VGG16 network trained on ImageNet - train a dogs-versus-cats classifier on top of these features for general cases
- Import it from the `keras.applications` module.
- Here's the list of image-classification models (all pretrained on the ImageNet dataset) that are available as part of `keras.applications`:
  - Xception
  - Inception V3
  - ResNet50
  - VGG16
  - VGG19
  - MobileNet

5.3.1 Feature extraction

Listing 5.16 Instantiating the VGG16 convolutional base

```
from keras.applications import VGG16

                          이미 학습된 weight값을 사용?
conv_base = VGG16(weights='imagenet',
                  include_top=False,  # densely connected classifier
                  input_shape=(150,150,3))  # optional
```

- Three arguments to the constructor:
  - `weights` - weights initialization
  - `include_top` – densely connected classifier on top of the network.
    By default, 1,000 classes from ImageNet.
    For the two classes of `cat` and `dog`, don't include it.
  - `input_shape` – shape of the image tensors

5.3 Using a pretrained convnet

>>> conv_base.summary()

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 150, 150, 3)	0
block1_conv1 (Convolution2D)	(None, 150, 150, 64)	1792
block1_conv2 (Convolution2D)	(None, 150, 150, 64)	36928
block1_pool (MaxPooling2D)	(None, 75, 75, 64)	0
block2_conv1 (Convolution2D)	(None, 75, 75, 128)	73856
block2_conv2 (Convolution2D)	(None, 75, 75, 128)	147584
block2_pool (MaxPooling2D)	(None, 37, 37, 128)	0
block3_conv1 (Convolution2D)	(None, 37, 37, 256)	295168
block3_conv2 (Convolution2D)	(None, 37, 37, 256)	590080
block3_conv3 (Convolution2D)	(None, 37, 37, 256)	590080
block3_pool (MaxPooling2D)	(None, 18, 18, 256)	0
block4_conv1 (Convolution2D)	(None, 18, 18, 512)	1180160
block4_conv2 (Convolution2D)	(None, 18, 18, 512)	2359808
block4_conv3 (Convolution2D)	(None, 18, 18, 512)	2359808
block4_pool (MaxPooling2D)	(None, 9, 9, 512)	0
block5_conv1 (Convolution2D)	(None, 9, 9, 512)	2359808
block5_conv2 (Convolution2D)	(None, 9, 9, 512)	2359808
block5_conv3 (Convolution2D)	(None, 9, 9, 512)	2359808
block5_pool (MaxPooling2D)	**(None, 4, 4, 512)**	0

4by4 크기의 이미지가 512개로 output
4*4*512 만큼의 픽셀들이 feature의 개수가 되는 것

5.3 Using a pretrained convnet

5.3.1 Feature extraction

- The final feature map has shape `(4, 4, 512)`. That's the feature on top of densely connected classifier.
- two ways to proceed:
  - Running the convolutional base over your dataset → recording its output to a Numpy array on disk → input to densely connected classifier - running the convolutional base once for every input image without data augmentation.
  - Running the whole thing end to end on the input data with data augmentation

5.3 Using a pretrained convnet

FAST FEATURE EXTRACTION WITHOUT DATA AUGMENTATION

Listing 5.17 Extracting features using the pretrained convolutional base

```
import os
import numpy as np
from keras.preprocessing.image import ImageDataGenerator
base_dir = './datasets/cats_and_dogs_small'
train_dir = os.path.join(base_dir, 'train')
validation_dir = os.path.join(base_dir, 'validation')
test_dir = os.path.join(base_dir, 'test')
datagen = ImageDataGenerator(rescale=1./255)
def extract_features(directory, sample_count):
    features = np.zeros(shape=(sample_count, 4, 4, 512))
    labels = np.zeros(shape=(sample_count))
    generator = datagen.flow_from_directory(
        directory,        target_size=(150, 150),
        batch_size=20,        class_mode='binary')
    i = 0
    for inputs_batch, labels_batch in generator:
        features_batch = conv_base.predict(inputs_batch) # conv_base = VGG16
        features[i * 20 : (i + 1) * 20] = features_batch
        labels[i * 20 : (i + 1) * 20] = i += 1labels_batch
        if i * 20 >= sample_count:
            break
    return features, labels
train_features, train_labels = extract_features(train_dir, 2000)
validation_features, validation_labels = extract_features(validation_dir, 1000)
test_features, test_labels = extract_features(test_dir, 1000)
```

5.3 Using a pretrained convnet

- The extracted features are currently of shape `(samples, 4, 4, 512)` → densely connected classifier `(samples, 8192)`:

```
train_features = np.reshape(train_features, (2000, 4 * 4 * 512))
validation_features = np.reshape(validation_features, (1000, 4 * 4 * 512))
test_features = np.reshape(test_features, (1000, 4 * 4 * 512))
```

Listing 5.18 Defining and training the densely connected classifier

```
from keras import models
from keras import layers
from keras import optimizers
model = models.Sequential()
model.add(layers.Dense(256, activation='relu', input_dim=4 * 4 * 512))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(1, activation='sigmoid'))
model.compile(optimizer=optimizers.RMSprop(lr=2e-5),
              loss='binary_crossentropy',
              metrics=['acc'])
history = model.fit(train_features, train_labels,
                    epochs=30,
                    batch_size=20,
                    validation_data=(validation_features, validation_labels))
```

작은 값

- Training is very fast, because you only have to deal with two `Dense` layers—an epoch takes less than one second even on CPU.

5.3 Using a pretrained convnet

Listing 5.19 Plotting the results

```
import matplotlib.pyplot as plt
acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']
epochs = range(len(acc))
plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.legend()
plt.figure()
plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()
plt.show()
```

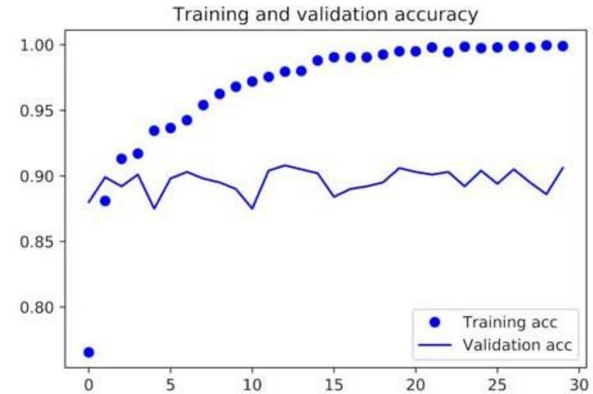


Figure 5.15 Training and validation accuracy for simple feature extraction

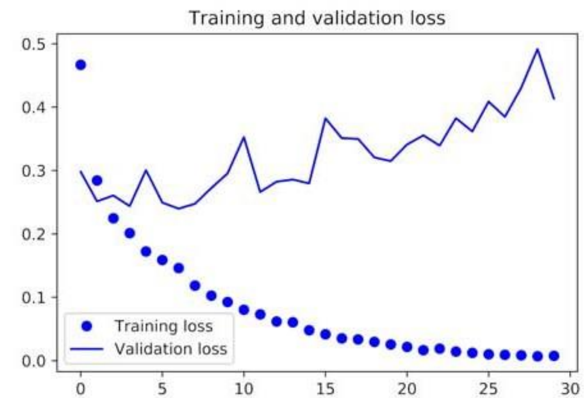


Figure 5.16 Training and validation loss for simple feature extraction

- You reach a validation accuracy of about 90%. 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.

5.3 Using a pretrained convnet

FEATURE EXTRACTION WITH DATA AUGMENTATION

- data augmentation during training - much slower and more expensive
- extending the conv_base model and running it end to end on the inputs

NOTE This technique is so expensive that you should only attempt it if you have access to a GPU—it's absolutely intractable on CPU. If you can't run your code on GPU, then the previous technique is the way to go

Listing 5.20 Adding a densely connected classifier on top of the convolutional base

```
from keras import models
from keras import layers
model = models.Sequential()
model.add(conv_base)
model.add(layers.Flatten())
model.add(layers.Dense(256, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

```
>>> model.summary()
Layer (type)                     Output Shape          Param #
================================================================
vgg16 (Model)                    (None, 4, 4, 512)     14714688
________________________________________________________________
flatten_1 (Flatten)              (None, 8192)          0
________________________________________________________________
dense_1 (Dense)                  (None, 256)           2097408
________________________________________________________________
dense_2 (Dense)                  (None, 1)             257
================================================================
Total params: 16,812,353
Trainable params: 16,812,353
Non-trainable params: 0
```

5.3 Using a pretrained convnet

FEATURE EXTRACTION WITH DATA AUGMENTATION

- *Freezing* - freeze the convolutional base, preventing weights from being updated during training
- If you don't do this, then the representations that were previously learned by the convolutional base will be modified during training.

- In Keras, you freeze a network by setting its `trainable` attribute to `False`:

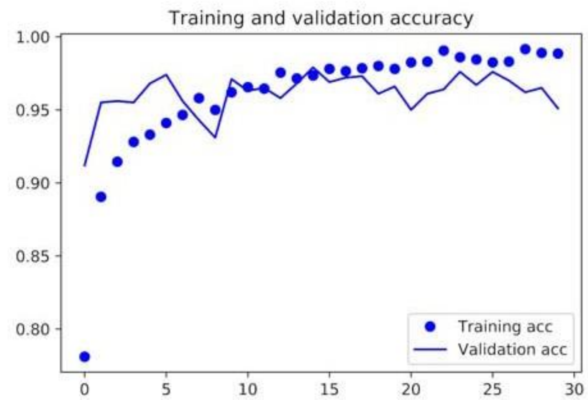
```
>>> print('This is the number of trainable weights `
          'before freezing the conv base:', len(model.trainable_weights))
This is the number of trainable weights before freezing the conv base: 30
>>> conv_base.trainable = False
>>> print('This is the number of trainable weights '
          'after freezing the conv base:', len(model.trainable_weights))
This is the number of trainable weights after freezing the conv base: 4
```

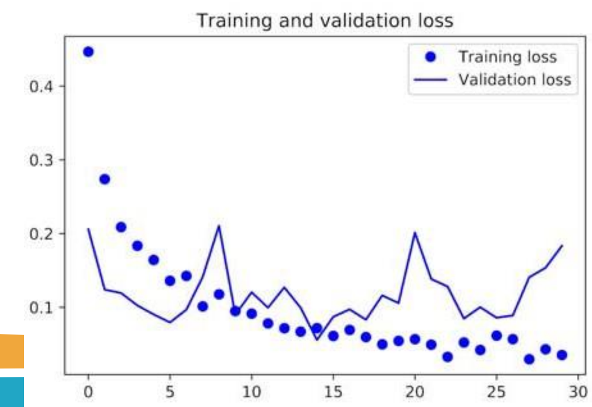
- *2 `Dense` layers that you added will be trained - That's a total of 4 weight tensors: two per layer (the main weight matrix and the bias vector). (13+2)*2 =30 weight tensors for whole model.*
- *Now you can start training your model, with the same data-augmentation configuration that you used in the previous example.*

5.3 Using a pretrained convnet

Listing 5.21 Training the model end to end with a frozen convolutional base

```
from keras.preprocessing.image import ImageDataGenerator
train_datagen = ImageDataGenerator(
      rescale=1./255,      rotation_range=20,
      width_shift_range=0.1,      height_shift_range=0.1,
      shear_range=0.1,      zoom_range=0.1,
      horizontal_flip=True,      fill_mode='nearest')
test_datagen=ImageDataGenerator(rescale=1./255) # No augmented!
train_generator = train_datagen.flow_from_directory(
        train_dir, # 타깃 디렉터리
        target_size=(150, 150), # 150 × 150로 변경
        batch_size=20,
        class_mode='binary') # 이진 레이블
validation_generator = test_datagen.flow_from_directory(
        validation_dir,
        target_size=(150, 150),
        batch_size=20,
        class_mode='binary')
model.compile(loss='binary_crossentropy',
        optimizer=optimizers.RMSprop(lr=2e-5),
        metrics=['acc']) augmented!
history = model.fit_generator(
        train_generator,  # 2000/100=20 data augmentations
        steps_per_epoch=100,      epochs=30,
        validation_data=validation_generator,
        validation_steps=50,
        verbose=2) # 진행 막대(progress bar)가 나오지 않도록 설정
# - 13s - loss: 0.5570 - acc: 0.7270 - val_loss: 0.4234 - val_acc: 0.8400
```





5.3.2 Fine-tuning

- *fine-tuning* - slightly adjusts the more abstract representations of the model being reused, in order to make them more relevant for the problem at hand.
- fine-tune the top layers of the convolutional base
- The steps for fine-tuning a network are as follow:
  1. Add your custom network on top of an already-trained base network.
  2. Freeze the base network.
  3. Train the part you added.
  4. Unfreeze some layers in the base network.
  5. Jointly train both these layers and the part you added.
- You already completed the first three steps when doing feature extraction.
- Let's proceed with step 4: you'll unfreeze your `conv_base` and then freeze individual layers inside it.

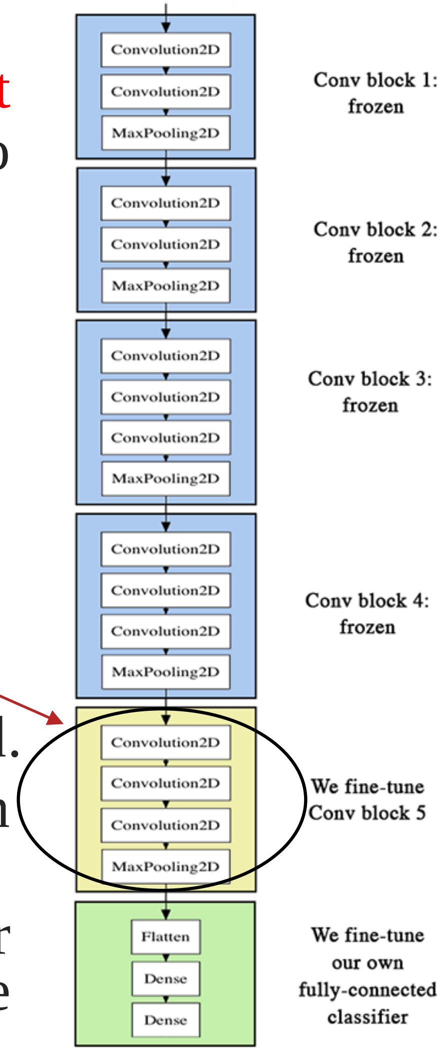


Figure 5.19 Fine-tuning the last convolutional block of the VGG16 network

5.3 Using a pretrained convnet

- You'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.
- Why not fine-tune more layers? Why not fine-tune the entire convolutional base? consider the following:
  - Earlier layers - more-generic, reusable features
  - Higher layers - more-specialized features.
  - 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.
- fine-tune only the top two or three layers in the convolutional base. Let's set this up, starting from where you left off in the previous example.

Listing 5.22 Freezing all layers up to a specific one

```
conv_base.trainable = True
set_trainable = False
for layer in conv_base.layers:
    if layer.name == 'block5_conv1':
        set_trainable = True
    if set_trainable:
        layer.trainable = True
    else:  # freeze before block5_conv1
        layer.trainable = False
```

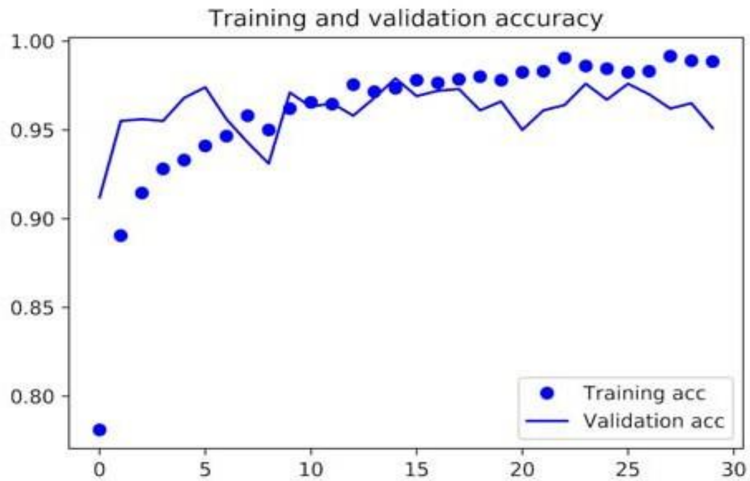
Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 150, 150, 3)	0
block1_conv1 (Conv2D)	(None, 150, 150, 64)	1792
block1_conv2 (Conv2D)	(None, 150, 150, 64)	36928
block1_pool (MaxPooling2D)	(None, 75, 75, 64)	0
block2_conv1 (Conv2D)	(None, 75, 75, 128)	73856
block2_conv2 (Conv2D)	(None, 75, 75, 128)	147584
block2_pool (MaxPooling2D)	(None, 37, 37, 128)	0
block3_conv1 (Conv2D)	(None, 37, 37, 256)	295168
block3_conv2 (Conv2D)	(None, 37, 37, 256)	590080
block3_conv3 (Conv2D)	(None, 37, 37, 256)	590080
block3_pool (MaxPooling2D)	(None, 18, 18, 256)	0
block4_conv1 (Conv2D)	(None, 18, 18, 512)	1180160
block4_conv2 (Conv2D)	(None, 18, 18, 512)	2359808
block4_conv3 (Conv2D)	(None, 18, 18, 512)	2359808
block4_pool (MaxPooling2D)	(None, 9, 9, 512)	0
block5_conv1 (Conv2D)	(None, 9, 9, 512)	2359808
block5_conv2 (Conv2D)	(None, 9, 9, 512)	2359808
block5_conv3 (Conv2D)	(None, 9, 9, 512)	2359808
block5_pool (MaxPooling2D)	(None, 4, 4, 512)	0

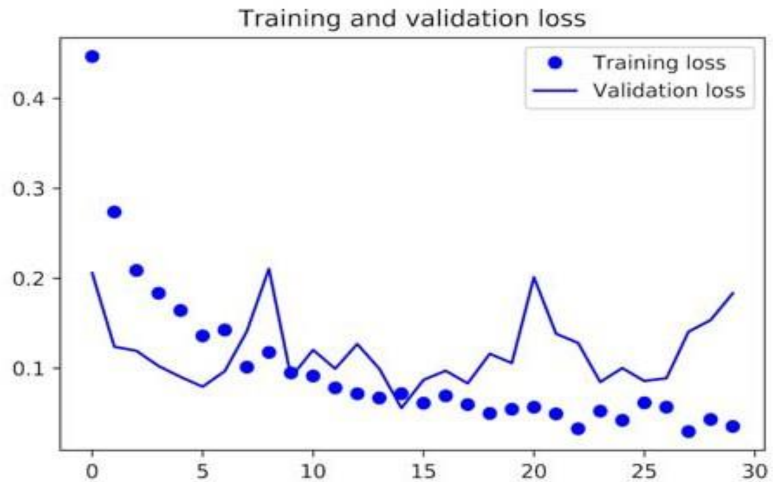
Total params: 14,714,688
Trainable params: 0
Non-trainable params: 14,714,688

5.3 Using a pretrained convnet

Listing 5.23 Fine-tuning the model

```
model.compile(loss='binary_crossentropy',
      optimizer=optimizers.RMSprop(lr=1e-5),
      metrics=['acc'])
history = model.fit_generator(
      train_generator,
      steps_per_epoch=100,
      epochs=100,
      validation_data=validation_generator,
      validation_steps=50)
```





5.3 Using a pretrained convnet

Listing 5.24 Smoothing the plots

```
def smooth_curve(points, factor=0.8):
  smoothed_points = []
  for point in points:
    if smoothed_points:
      previous = smoothed_points[-1]
      smoothed_points.append(previous * factor + point * (1 - factor))
    else:
      smoothed_points.append(point)
  return smoothed_points

plt.plot(epochs,
         smooth_curve(acc), 'bo', label='Smoothed training acc')
plt.plot(epochs,
         smooth_curve(val_acc), 'b', label='Smoothed validation acc')
plt.title('Training and validation accuracy')
plt.legend()
plt.figure()
plt.plot(epochs,
         smooth_curve(loss), 'bo', label='Smoothed training loss')
plt.plot(epochs,
         smooth_curve(val_loss), 'b', label='Smoothed validation loss')
plt.title('Training and validation loss')
plt.legend()
plt.show()
```

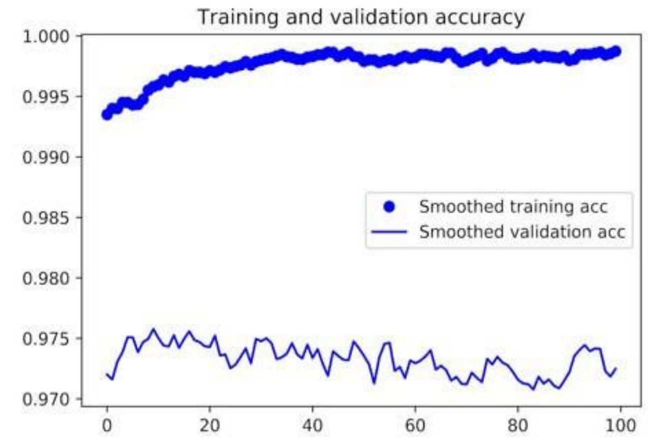


Figure 5.22 Smoothed curves for training and validation accuracy for fine-tuning

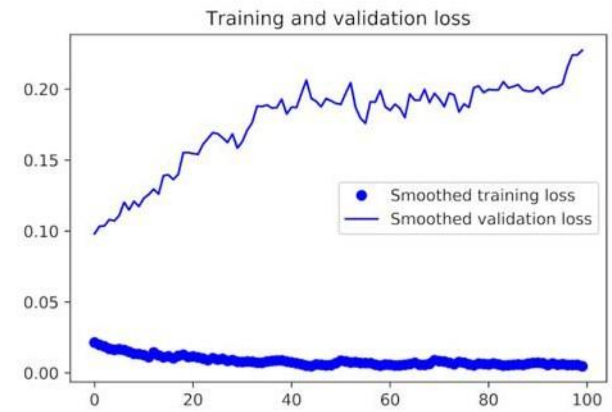


Figure 5.23 Smoothed curves for training and validation loss for fine-tuning

- The validation accuracy curve from about 96% to above 97%
- Note that the loss curve doesn't show any real improvement (in fact, it's deteriorating).
  What you display is an average of pointwise loss values;
  but what matters for accuracy is the distribution of the loss values, not their average.

5.3 Using a pretrained convnet

- You can now finally evaluate this model on the test data:

```
test_generator =test_datagen.flow_from_directory(
        test_dir,
        target_size=(150, 150),
        batch_size=20,
        class_mode='binary')
```

- test accuracy of 97% - In the original Kaggle competition around this dataset, this would have been one of the top results using only a small fraction of the training data available (about 10%). There is a huge difference between being able to train on 20,000 samples compared to 2,000 samples!