Lecture_6_DL_for_computer_vision

November 9, 2022

1 Introduction to deep learning for computer vision

Computer vision is the earliest and biggest success story of deep learning. Every day, we're interacting with deep vision models - via Google Photos, Google image search, YouTube, video filters in camera apps, OCR software, and many more. These models are also at the heart of cutting-edge research in autonomous driving, robotics, AI-assisted medical diagnosis, and even autonomous farming.

This lecture introduces convolutional neural networks, also known as **convnets**, the type of DL model that is now used almost universally in computer vision applications. We will learn how to apply convnets to image-classification problems.

[1]:
```
import matplotlib.pyplot as plt

def plot_helper(x_data, y_data, plot_styles, labels, title, x_label, y_label):
    assert len(x_data) == len(y_data)
    assert len(x_data) == len(plot_styles)
    assert len(x_data) == len(labels)

    fig = plt.figure(figsize=(9 / 2.54, 5 / 2.54), dpi=150)
    plt.rc('axes', titlesize=7)  # fontsize of the axes title
    plt.rc('axes', labelsize=6)  # fontsize of the x and y labels
    plt.rc('xtick', labelsize=6)  # fontsize of the tick labels
    plt.rc('ytick', labelsize=6)  # fontsize of the tick labels
    plt.autoscale(enable=True, axis='x', tight=True)

    for i in range(len(x_data)):
        plt.plot(x_data[i], y_data[i], plot_styles[i], label=labels[i],␣
 ↪linewidth=1, markersize=3)
    plt.title(title)
    plt.xlabel(x_label)
    plt.ylabel(y_label)
    plt.legend(fontsize=5.5)
```

1.1 The Functional API

Before we dive into the convnets, let's introduce the basics of the Keras's Functional API. We'll consider a simple example: the stack of two layers (one hidden and output). A Functional API version of such a model would look like:

[2]:
```
from tensorflow import keras
from tensorflow.keras import layers

inputs = keras.Input(shape=(3,), name="my_input")
features = layers.Dense(64, activation="relu")(inputs)
outputs = layers.Dense(10, activation="softmax")(features)
model = keras.Model(inputs=inputs, outputs=outputs, name="functional")
```

We started by declaring an `Input` (we can also give names to input objects, like everything else):

```
inputs = keras.Input(shape=(3,), name="my_input")
```

The `inputs` object holds information about the shape and dtype of the data that our model will process:

[3]:
```
print(inputs.shape)
print(inputs.dtype)
```

```
(None, 3)
<dtype: 'float32'>
```

We call such an object a **symbolic tensor**. It doesn't contain any actual data, but it encodes the specifications of the actual tensors of data that the model will see when we use it. It stands for future tensors of data.

Next, we created a layer and called it on the input, as a **function**:

```
features = layers.Dense(64, activation="relu")(inputs)
```

All Keras layers can be called both on real tensors of data and on symbolic tensors. In the latter case, they return a new symbolic tensor, with updated shape and dtype information:

[4]:
```
features.shape
```

[4]:
```
TensorShape([None, 64])
```

After obtaining the final outputs, we instantiated the model by specifying its inputs and outputs in the `Model` constructor:

```
outputs = layers.Dense(10, activation="softmax")(features)
```

```
model = keras.Model(inputs=inputs, outputs=outputs)
```

Here's the summary of our model:

[5]:
```
model.summary()
```

```
Model: "functional"
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 my_input (InputLayer)        [(None, 3)]               0

 dense (Dense)                (None, 64)                256
```

```
 dense_1 (Dense)             (None, 10)                650

=================================================================
Total params: 906
Trainable params: 906
Non-trainable params: 0
_________________________________________________________________
```

1.2 Introduction to convnets

First, let's take a look at a simple convnet example that classifies MNIST digits, a task we've already carried out using a densely connected network (we achieved the test accuracy of 97.8%). Even though the convnet will be basic, it will outperform significantly our densely connected model in terms of accuracy.

The following listing presents a basic convnet. It's a stack of **`Conv2D`** and **`MaxPooling2D`** layers. We'll build the model using the Functional API.

Instantiating a small convnet

[6]:
```
inputs = keras.Input(shape=(28, 28, 1))
x = layers.Conv2D(filters=32, kernel_size=3, activation="relu")(inputs)
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.Flatten()(x)
outputs = layers.Dense(10, activation="softmax")(x)
model = keras.Model(inputs=inputs, outputs=outputs, name='small_convnet')
```

As input, a convnet takes tensors of shape `(image_height, image_width, image_channels)`, not including the batch dimension. In this case, we'll configure the convnet to process inputs of size `(28, 28, 1)`, which is the format of MNIST grayscale images (one channel).

[7]:
```
model.summary()
```

```
Model: "small_convnet"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 28, 28, 1)]       0

 conv2d (Conv2D)             (None, 26, 26, 32)        320

 max_pooling2d (MaxPooling2D  (None, 13, 13, 32)       0
 )

 conv2d_1 (Conv2D)           (None, 11, 11, 64)        18496
```

```
 max_pooling2d_1 (MaxPooling  (None, 5, 5, 64)         0
 2D)

 conv2d_2 (Conv2D)           (None, 3, 3, 128)         73856

 flatten (Flatten)           (None, 1152)              0

 dense_2 (Dense)             (None, 10)                11530

=================================================================
Total params: 104,202
Trainable params: 104,202
Non-trainable params: 0
_________________________________________________________________
```

We can see that the output of every `Conv2D` and `MaxPooling2D` layer is a rank-3 tensor of shape `(height, width, channels)`. The width and height dimensions tend to shrink as we go deeper in the model. The number of channels is controlled by the first argument passed to the `Conv2D` layers (32, 64, or 128).

After the last `Conv2D` layer, we end up with an output of shape `(3, 3, 128)`, a 3×3 feature map of 128 channels. The next step is to feed this output into a densely connected classifier. These classifiers process vectors, which are 1D, whereas the current output is a rank-3 tensor. To bridge the gap, we **flatten** the 3D outputs to 1D with a `Flatten` layer before adding the `Dense` layers.

Finally, we do a 10-class classification, so our last layer has 10 outputs and a softmax activation.

Training the convnet on MNIST images

Now, let's train the convnet on the MNIST digits. We'll reuse a lot of the code from our first MNIST classifier. Because we're doing 10-way classification with a softmax output, we'll use the categorical crossentropy loss, and because our labels are integers, we'll use the sparse version, `sparse_categorical_crossentropy`.

[8]:
```
from tensorflow.keras.datasets import mnist

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

```
Epoch 1/5
938/938 [==============================] - 9s 3ms/step - loss: 0.1510 -
accuracy: 0.9523
Epoch 2/5
938/938 [==============================] - 3s 3ms/step - loss: 0.0438 -
```

```
accuracy: 0.9863
Epoch 3/5
938/938 [==============================] - 3s 3ms/step - loss: 0.0306 -
accuracy: 0.9907
Epoch 4/5
938/938 [==============================] - 2s 3ms/step - loss: 0.0225 -
accuracy: 0.9932
Epoch 5/5
938/938 [==============================] - 2s 3ms/step - loss: 0.0174 -
accuracy: 0.9944
```

[8]:
```
<keras.callbacks.History at 0x1b6a193f0d0>
```

Evaluating the convnet

[9]:
```
test_loss, test_acc = model.evaluate(test_images, test_labels)
print(f"Test accuracy: {test_acc:.3f}")
```

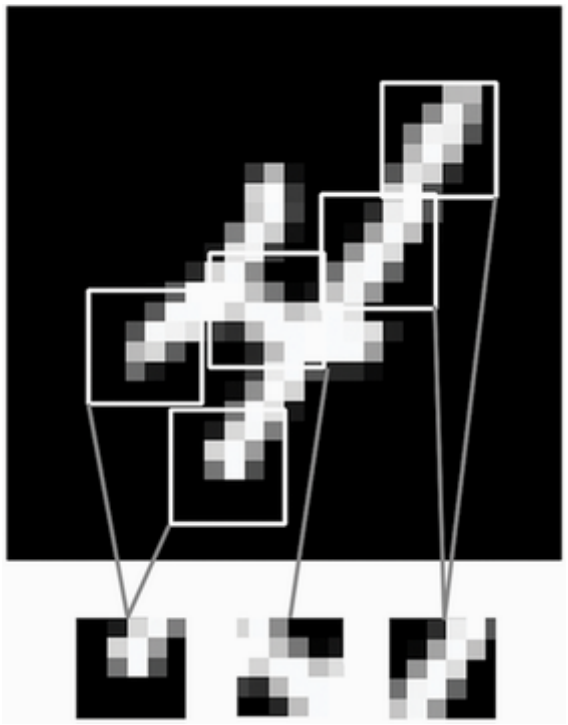
```
313/313 [==============================] - 1s 1ms/step - loss: 0.0397 -
accuracy: 0.9889
Test accuracy: 0.989
```

Our first densely connected model had a test accuracy of 97.8%, whereas the basic convnet has a test accuracy of 99.3%. The error rate is reduced from 2.2% to 0.7%, which is a pretty notable improvement!

Why does this simple convnet work so well, compared to a densely connected model? To answer this, let's see what the `Conv2D` and `MaxPooling2D` layers do.

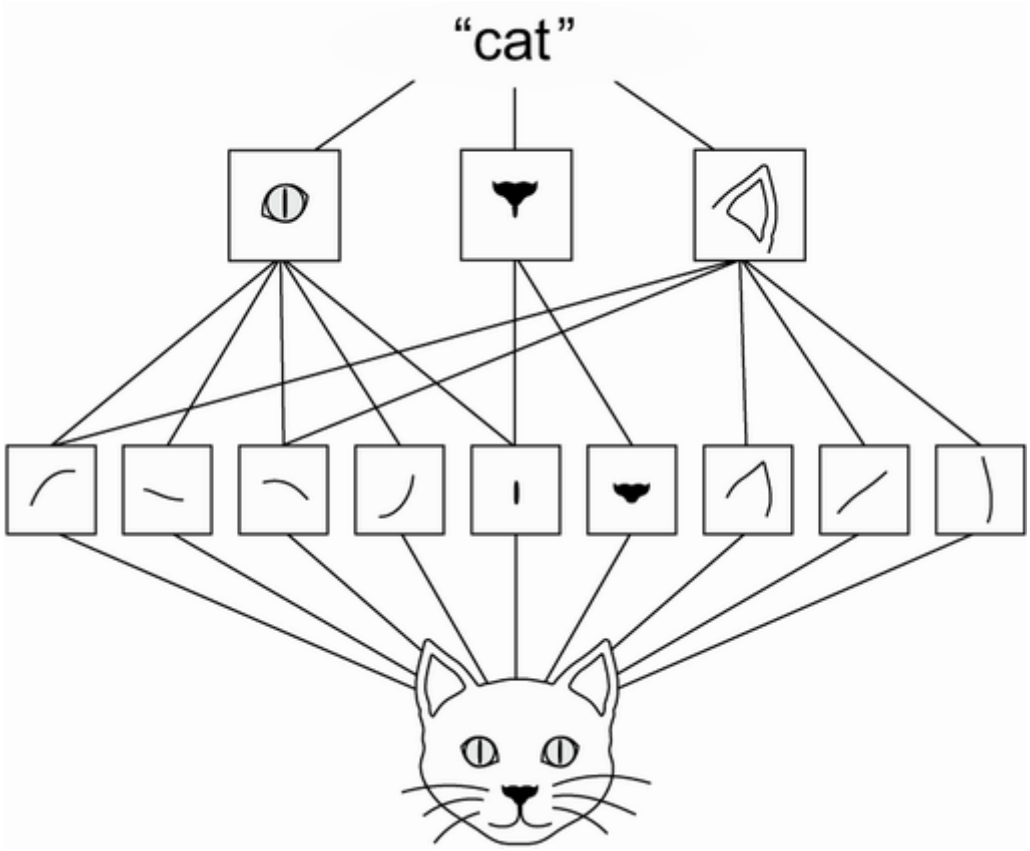
1.2.1 The convolution operation

The fundamental difference between a densely connected layer and a convolution layer is this: **Dense layers learn global patterns in their input feature space** (for example, for a MNIST digit, patterns involving all pixels), whereas **convolution layers learn local patterns** — in the case of images, patterns found in small 2D windows of the inputs (see figure below). In our example, the windows size is 3×3.



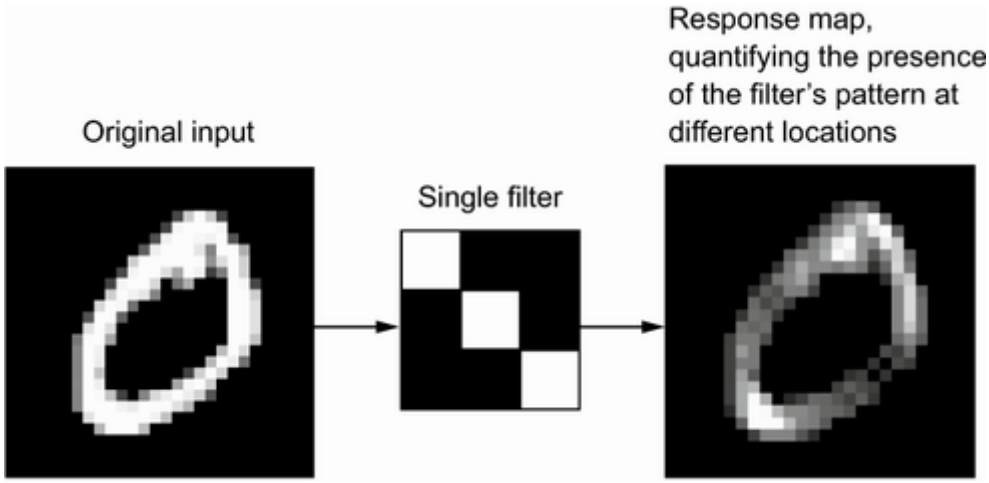
This key characteristic gives convnets two interesting properties: - **The patterns they learn are translation-invariant**. After learning a certain pattern in the lower-right corner of a picture, a convnet can recognize it anywhere in a picture. A densely connected model would have to learn the pattern anew if it appeared at a new location. This makes convnets data-efficient when processing images (because the visual world is fundamentally translation-invariant): they need fewer training samples to learn representations that have generalization power. - **They can learn spatial hierarchies of patterns**. The first convolution layer will learn small local patterns such as edges, the second will learn larger patterns made of the features of the first layers, and so on (see figure below). This allows convnets to efficiently learn increasingly complex and abstract visual concepts, because the visual world is fundamentally spatially hierarchical.

The visual world forms a spatial hierarchy of visual modules: elementary lines or textures combine into simple objects such as eyes or ears, which combine into high-level concepts such as “cat”.



Convolutions operate over rank-3 tensors called **feature maps**, with two spatial axes (*height* and *width*) and a *depth* axis (also called the *channels* axis). For an RGB image, the dimension of the depth axis is 3, because the image has three color channels: red, green, and blue. For a black-and-white picture, like the MNIST digits, the depth is 1 (levels of gray). The convolution operation extracts patches from its input feature map and applies the same transformation to all of these patches, producing an **output feature map**. This output feature map is still a rank-3 tensor: it has a width and a height. Its depth can be arbitrary, because the output depth is a parameter of the layer, and the different channels in that depth axis no longer stand for specific colors as in RGB input; rather, they stand for **filters**. Filters encode specific aspects of the input data: at a high level, a single filter could encode the concept “presence of a face in the input,” for instance.

In the MNIST example, the first convolution layer takes a feature map of size `(28, 28, 1)` and outputs a feature map of size `(26, 26, 32)`: it computes 32 filters over its input. Each of these 32 output channels contains a 26×26 grid of values, which is a **response map** of the filter over the input, indicating the response of that filter pattern at different locations in the input (see figure below).

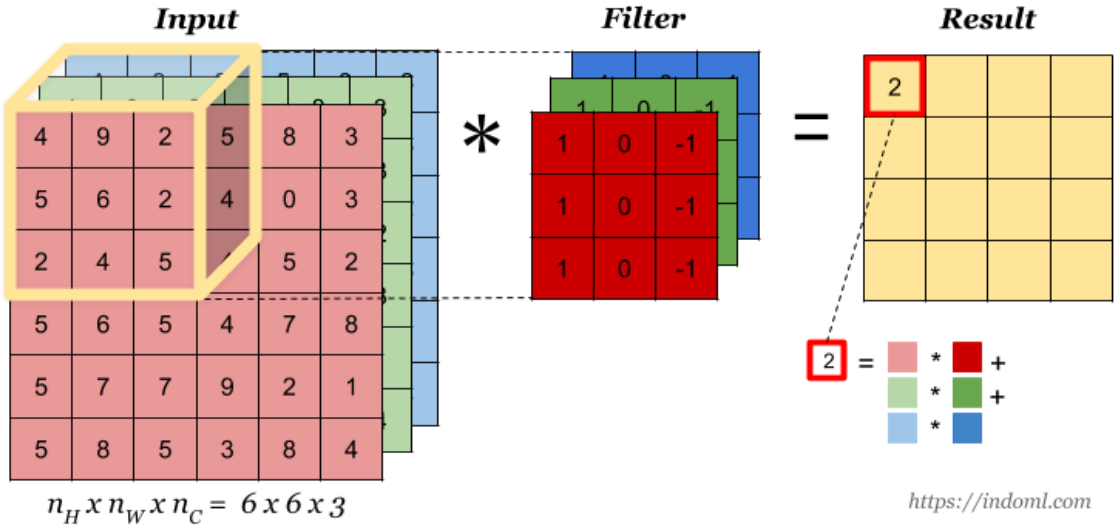


That is what the term **feature map** means: every dimension in the depth axis is a feature (or filter), and the rank-2 tensor `output[:, :, n]` is the 2D spatial map of the response of this filter over the input.

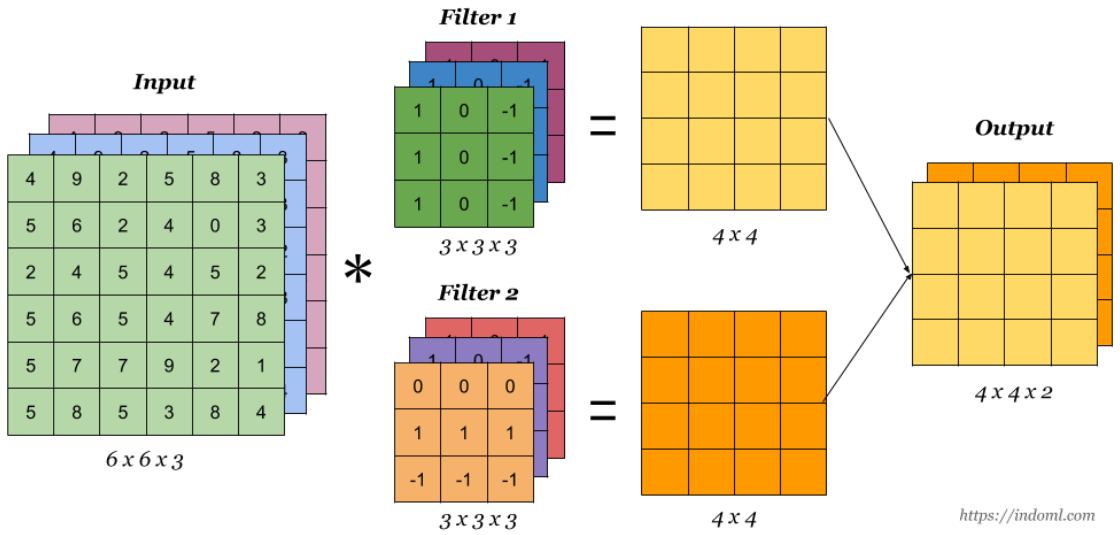
Convolutions are defined by two key parameters: - **Size of the patches extracted from the inputs** — These are typically 3×3 or 5×5. In the example, they were 3×3, which is a common choice. - **Depth of the output feature map** — This is the number of filters computed by the convolution. The example started with a depth of 32 and ended with a depth of 128.

In Keras `Conv2D` layers, these parameters are the first arguments passed to the layer: `Conv2D(output_depth, (window_height, window_width))`.

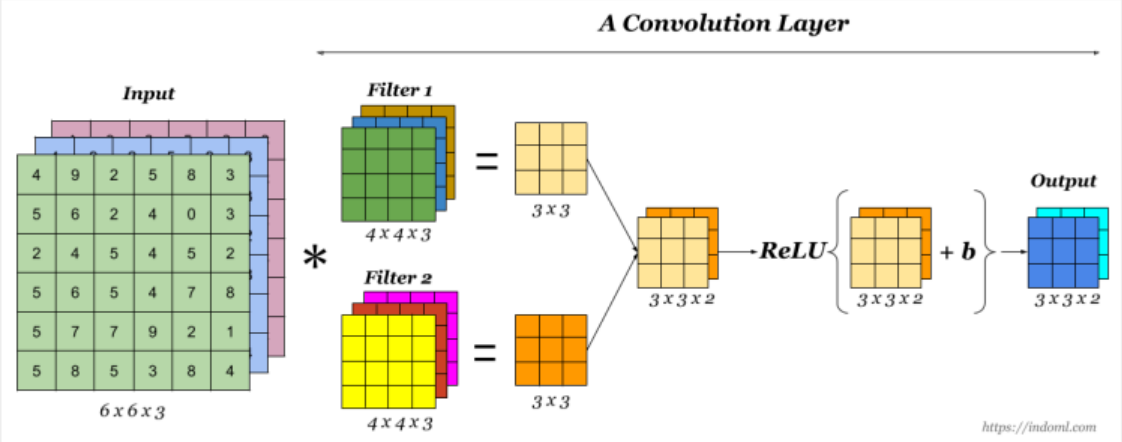
Convolution works by sliding the filters of size `(window_height, window_width, input_depth)` over the 3D input feature map, stopping at every possible location, and calculating convolution between the filter and the 3D patch of input feature at a current location (see the figure below). This process results in an output feature map of size `(output_height, output_width, output_depth)`. Every spatial location in the output feature map corresponds to the same location in the input feature map (for example, the lower-right corner of the output contains information about the lower-right corner of the input).



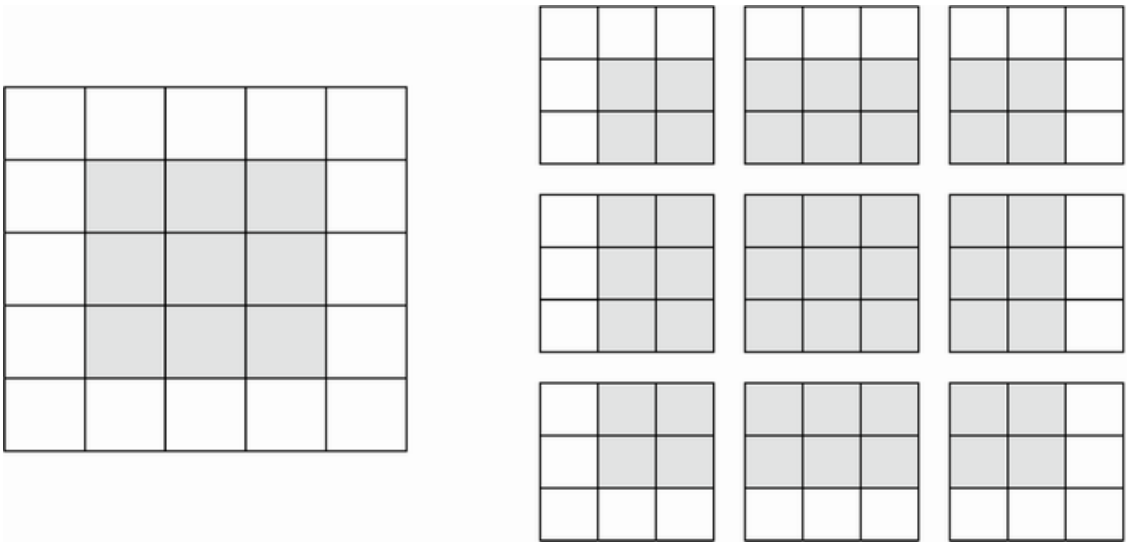
An example of a `(6, 6, 3)` input feature map and `(4, 4, 2)` output feature map, with 3×3 windows, in the figure below.



The overall convolution process is described in the figure below.



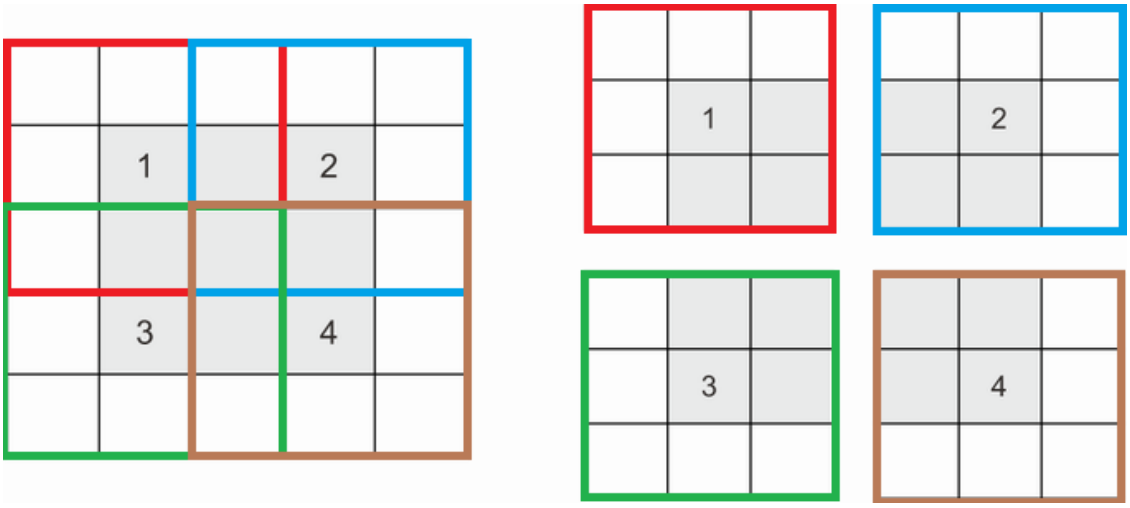
Understanding border effects and padding Consider a 5×5 feature map (25 tiles total). There are only 9 tiles around which you can center a 3×3 window, forming a 3×3 grid (figure below). Hence, the output feature map will be 3×3. It shrinks a little: by exactly two tiles alongside each dimension, in this case. We can see this border effect in action in the earlier example: we start with 28×28 inputs, which become 26×26 after the first convolution layer.



In order to get an output feature map with the same spatial dimensions as the input, we can use **padding**. Padding consists of adding an appropriate number of rows and columns on each side of the input feature map so as to make it possible to fit center convolution windows around every input tile. For a 3 × 3 window, we add one column on the right and left, one row at the top and bottom. For a 5 × 5 window, we add two rows.

In `Conv2D` layers, padding is configurable via the `padding` argument, which takes two values: `"valid"`, which means no padding (only valid window locations will be used), and `"same"`, which means to pad in such a way that an output has the same width and height as the input. The padding argument defaults to `"valid"`.

Understanding convolution strides The other factor that affects the output size is the notion of **strides**. Our description of convolution so far has assumed that the center tiles of the convolution windows are all contiguous. But 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: convolutions with a stride higher than 1. In the figure below, we see the patches extracted by a 3 × 3 convolution with stride 2 over a 5 × 5 input (without padding).



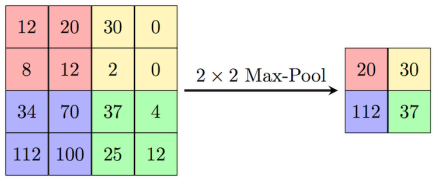
Using stride 2 means the width and height of the feature map are downsampled by a factor of 2 (in addition to any changes induced by border effects). **Strided convolutions are rarely used in classification models, but they are useful in image segmentation tasks**.

In classification models, instead of strides, we use **max-pooling** to downsample feature maps, which we've seen in action in our first convnet example. Let's look at it in more depth.

1.2.2 The max-pooling operation

In the convnet example, notice that the size of the feature maps is halved after every `MaxPooling2D` layer. For instance, before the first `MaxPooling2D layers`, the feature map is 26 × 26, but the max-pooling operation made it 13 × 13. That's the role of max pooling: to downsample feature maps, similar to strided convolutions.

Max pooling consists of extracting windows from the input feature maps and outputting the max value of each channel. It's conceptually similar to convolution, except that instead of transforming local patches via a learned linear transformation (the convolution kernel), they're transformed via a hardcoded `max` tensor operation. A big difference from convolution is that max pooling is usually done with 2 × 2 windows and stride 2, in order to downsample the feature maps by a factor of 2. On the other hand, convolution is typically done with 3 × 3 windows and no stride (stride 1).



Why downsample feature maps this way? Why not remove the max-pooling layers and keep fairly large feature maps all the way up? Let's look at this option. Our model would then look like the following listing.

An incorrectly structured convnet missing its max-pooling layers

[10]:
```
inputs = keras.Input(shape=(28, 28, 1))
x = layers.Conv2D(filters=32, kernel_size=3, activation="relu")(inputs)
x = layers.Conv2D(filters=64, kernel_size=3, activation="relu")(x)
x = layers.Conv2D(filters=128, kernel_size=3, activation="relu")(x)
x = layers.Flatten()(x)
outputs = layers.Dense(10, activation="softmax")(x)
model_no_max_pool = keras.Model(inputs=inputs, outputs=outputs)
```

[11]:
```
model_no_max_pool.summary()
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, 28, 28, 1)]       0

 conv2d_3 (Conv2D)           (None, 26, 26, 32)        320
```

```
 conv2d_4 (Conv2D)            (None, 24, 24, 64)        18496

 conv2d_5 (Conv2D)            (None, 22, 22, 128)       73856

 flatten_1 (Flatten)          (None, 61952)             0

 dense_3 (Dense)              (None, 10)                619530

=================================================================
Total params: 712,202
Trainable params: 712,202
Non-trainable params: 0
_________________________________________________________________
```

What's wrong with this setup? Two things: - It isn't useful for learning a spatial hierarchy of features. The 3×3 windows in the third layer will only contain information coming from 7×7 windows in the initial input. The high-level patterns learned by the convnet will still be very small with regard to the initial input, which may not be enough to learn to classify digits (try recognizing a digit by only looking at it through windows that are 7×7 pixels). **We need the features from the last convolution layer to contain information about the totality of the input!**. - The final feature map has $22 \times 22 \times 128 = 61{,}952$ total coefficients per sample. This is huge! When we flatten it to stick a `Dense` layer of size 10 on top, that layer would have over half a million parameters. This is far too large for such a small model and would result in intense overfitting.

In short, the reason to use downsampling is to reduce the number of feature-map coefficients to process, as well as to induce spatial-filter hierarchies by making successive convolution layers look at increasingly large windows (in terms of the fraction of the original input they cover).

Note that max pooling isn't the only way you can achieve such downsampling. We can also use strides in the prior convolution layer. And we can use **average pooling** instead of max pooling, where each local input patch is transformed by taking the average value of each channel over the patch, rather than the max. But max pooling tends to work better than these alternative solutions. The reason is that features tend to encode the spatial presence of some pattern or concept over the different tiles of the feature map (hence the term **feature map**), and it's more informative to look at the maximal presence of different features than at their average presence.

The most reasonable subsampling strategy is to first produce dense maps of features (via unstrided convolutions) and then look at the maximal activation of the features over small patches, rather than looking at sparser windows of the inputs (via strided convolutions) or averaging input patches, which could cause us to miss or dilute feature-presence information.

Now let's move on to more useful, practical applications of convnets.

1.3 Training a convnet from scratch on a small dataset

Having to train an image-classification model using very little data is a common situation, which is likely to be encountered in practice. A "few" samples can mean anywhere 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 5,000 pictures of cats and dogs (2,500 cats, 2,500 dogs). We'll

use 2,000 pictures for training, 1,000 for validation, and 2,000 for testing.

In this lecture, we'll review one basic strategy to tackle this problem: training a new model from scratch using what little data you have. We'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 us to a classification accuracy of about 70%. At that point, the main issue will be overfitting. Then we'll introduce data augmentation, a powerful technique for mitigating overfitting in computer vision. By using data augmentation, we'll improve the model to reach an accuracy of 80–85%.

1.3.1 The relevance of deep learning for small-data problems

What qualifies as "enough samples" to train a model is relative — relative to the size and depth of the model we're trying to train, for starters. It isn't possible to train a convnet to solve a complex problem with just a few tens of samples, but a few hundred can potentially suffice if the model is small and well regularized and the task is simple. Because convnets learn local, translation-invariant features, they're highly data-efficient on perceptual problems. Training a convnet from scratch on a very small image dataset will yield reasonable results despite a relative lack of data, without the need for any custom feature engineering. We'll see this in action in this lecture.

What's more, DL models are by nature highly repurposable: we can take, say, 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. Specifically, in the case of computer vision, many pre-trained models (usually trained on the ImageNet dataset) are now publicly available for download and can be used to bootstrap powerful vision models out of very little data. This is one of the greatest strengths of DL: feature reuse, which will be explored in the following lectures.

1.3.2 Downloading the data

The **Dogs vs. Cats** dataset that we will use isn't packaged with Keras. It was made available by Kaggle as part of a computer vision competition in late 2013, back when convnets weren't mainstream. You can download the dataset from https://www.microsoft.com/en-us/download/details.aspx?id=54765.

The pictures in our dataset are medium-resolution color JPEGs. Some examples are given below.

Unsurprisingly, the original dogs-versus-cats Kaggle competition, all the way back in 2013, was won by entrants who used convnets. The best entries achieved up to 95% accuracy. With the first network, we will get fairly close to this accuracy, even though we will train our 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). After downloading and uncompressing the data, we'll create a new dataset, **cats_vs_dogs_small**, which will contain three subsets: training, validation and test set, with 1,000, 500 and 1,000 samples of each class, respectively. Why do this? Because many of the image datasets you'll encounter in your career only contain a few thousand samples, not tens of thousands. Having more data available would make the problem easier, so it's good practice to learn with a small dataset.

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

```
cats_vs_dogs_small/
   train/
      cat/
      dog/
   validation/
      cat/
      dog/
   test/
      cat/
      dog/
```

It can be downloaded from https://drive.google.com/drive/folders/1E0R8ngo3IcSthRQmdH5TPOYrq5XTLyck?us

1.3.3 Building the model

We will reuse the same general model structure you saw in the first example: the convnet will be a stack of alternated `Conv2D` (with `relu` activation) and `MaxPooling2D` layers.

But because we're dealing with bigger images and a more complex problem, we'll make our model larger, accordingly: it will have two more `Conv2D` and `MaxPooling2D` stages. This serves both to augment the capacity of the model and to further reduce the size of the feature maps so they aren't overly large when we reach the `Flatten` layer. Here, because we start from inputs of size 180 pixels × 180 pixels (a somewhat arbitrary choice), we end up with feature maps of size 7 × 7 just before the `Flatten` layer.

Note

The depth of the feature maps progressively increases in the model (from 32 to 256), whereas the size of the feature maps decreases (from 180 × 180 to 7 × 7). This is a pattern we'll see in almost all convnets.

Because we're looking at a binary-classification problem, we'll end the model with a single unit (a `Dense` layer of size 1) and a `sigmoid` activation. This unit will encode the probability that the model is looking at one class or the other.

One last small difference: we will start the model with a `Rescaling` layer, which will rescale image inputs (whose values are originally in the [0, 255] range) to the [0, 1] range.

Instantiating a small convnet for dogs vs cats classification

[12]:
```
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow import get_logger

get_logger().setLevel('ERROR')    # Suppress info and warnings in logger

def cat_vs_dogs_model():
    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, name="cat_vs_dogs")
    model.compile(loss="binary_crossentropy",
                  optimizer="rmsprop",
                  metrics=["accuracy"])

    return model
```

[13]:
```
model = cat_vs_dogs_model()
model.summary()
```

```
Model: "cat_vs_dogs"
_________________________________________________________________
 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 (MaxPooling  (None, 89, 89, 32)       0
 2D)

 conv2d_7 (Conv2D)           (None, 87, 87, 64)        18496

 max_pooling2d_3 (MaxPooling  (None, 43, 43, 64)       0
 2D)

 conv2d_8 (Conv2D)           (None, 41, 41, 128)       73856

 max_pooling2d_4 (MaxPooling  (None, 20, 20, 128)      0
 2D)

 conv2d_9 (Conv2D)           (None, 18, 18, 256)       295168

 max_pooling2d_5 (MaxPooling  (None, 9, 9, 256)        0
 2D)

 conv2d_10 (Conv2D)          (None, 7, 7, 256)         590080

 flatten_2 (Flatten)         (None, 12544)             0

 dense_4 (Dense)             (None, 1)                 12545

=================================================================
Total params: 991,041
Trainable params: 991,041
Non-trainable params: 0
_________________________________________________________________
```

1.3.4 Data preprocessing

Data should be formatted into appropriately preprocessed floating-point tensors before being fed into the model. Currently, the data sits on a drive as JPEG files, so the steps for getting it into the model 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. Resize them to a shared size (we'll use 180 × 180). 5. Pack them into batches (we'll use batches of 32 images).

It may seem a bit daunting, but fortunately Keras has utilities to take care of these steps automatically. In particular, Keras features the utility function **`image_dataset_from_ directory()`**, which lets you quickly set up a data pipeline that can automatically turn image files on disk into batches of preprocessed tensors. This is what we'll use here.

Calling `image_dataset_from_directory(directory)` will first list the subdirectories of directory and assume each one contains images from one of our classes. It will then index the image files in each subdirectory. Finally, it will create and return a **`tf.data.Dataset`** object configured to read these files, shuffle them, decode them to tensors, resize them to a shared size, and pack them into batches.

Using `image_dataset_from_directory` to read images

[14]:
```
import pathlib
from tensorflow.keras.utils import image_dataset_from_directory

base_dir = pathlib.Path("cats_vs_dogs_small")
train_dataset = image_dataset_from_directory(
    base_dir / "train",
    image_size=(180, 180),
    batch_size=32)
validation_dataset = image_dataset_from_directory(
    base_dir / "validation",
    image_size=(180, 180),
    batch_size=32)
test_dataset = image_dataset_from_directory(
    base_dir / "test",
    image_size=(180, 180),
    batch_size=32)
```

```
Found 2000 files belonging to 2 classes.
Found 1000 files belonging to 2 classes.
Found 2000 files belonging to 2 classes.
```

Basics of TensorFlow Dataset objects

TensorFlow makes available the **`tf.data`** API to create efficient input pipelines for machine learning models. Its core class is `tf.data.Dataset`.

A `Dataset` object is an iterator: we can use it in a `for` loop. It will typically return batches of input data and labels. We can pass a `Dataset` object directly to the `fit()` method of a Keras model.

The `Dataset` class handles many key features that would otherwise be cumbersome to implement yourself - in particular, asynchronous data prefetching (preprocessing the next batch of data while the previous one is being handled by the model, which keeps execution flowing without interruptions).

The `Dataset` class also exposes a functional-style API for modifying datasets. Here's a quick

example: let's create a `Dataset` instance from a NumPy array of random numbers. We'll consider 1,000 samples, where each sample is a vector of size 8:

[15]:
```
import numpy as np
import tensorflow as tf
random_numbers = np.random.normal(size=(1000, 8))
dataset = tf.data.Dataset.from_tensor_slices(random_numbers)
```

[16]:
```
for i, element in enumerate(dataset):
    print(element.shape)
    if i >= 2:
        break
```

```
(8,)
(8,)
(8,)
```

We can use the `batch()` method to batch the data:

[17]:
```
batched_dataset = dataset.batch(32)
for i, element in enumerate(batched_dataset):
    print(element.shape)
    if i >= 2:
        break
```

```
(32, 8)
(32, 8)
(32, 8)
```

The `.map()` method is used to reshape the elements of a dataset. In our case, we can use it to reshape the dataset from shape `(8,)` to shape `(2, 4)`:

[18]:
```
reshaped_dataset = dataset.map(lambda x: tf.reshape(x, (2, 4)))
for i, element in enumerate(reshaped_dataset):
    print(element.shape)
    if i >= 2:
        break
```

```
(2, 4)
(2, 4)
(2, 4)
```

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

[19]:
```
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,)
```

Fitting the model using a `Dataset`

Let's fit the model on our dataset. We'll use the `validation_data` argument in `fit()` to monitor validation metrics on a separate `Dataset` object.

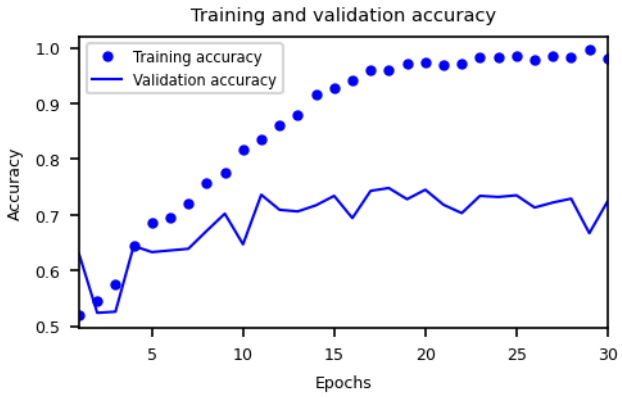
Note that we'll also use a `ModelCheckpoint` callback to save the model after each epoch. We'll configure it with the path specifying where to save the file, as well as the arguments `save_best_only=True` and `monitor="val_loss"`: they tell the callback to only save a new file (overwriting any previous one) when the current value of the `val_loss` metric is lower than at any previous time during training. This guarantees that our saved file will always contain the state of the model corresponding to its best-performing training epoch, in terms of its performance on the validation data. As a result, we won't have to retrain a new model for a lower number of epochs if we start overfitting: we can just reload our saved file.

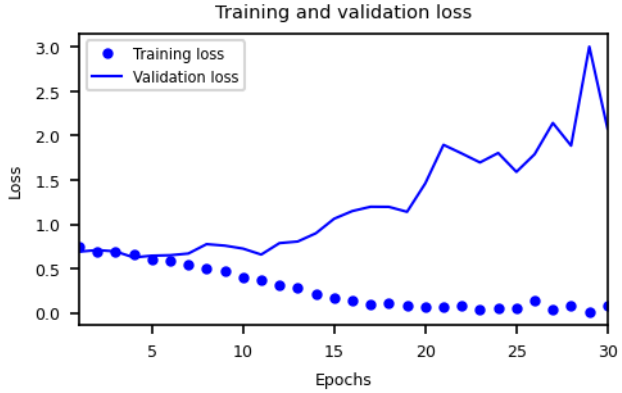
[20]:
```
callbacks = [
    keras.callbacks.ModelCheckpoint(
        filepath="models/convnet_from_scratch.keras",
        save_best_only=True,
        monitor="val_loss")
]
history = model.fit(
    train_dataset,
    epochs=30,
    validation_data=validation_dataset,
    callbacks=callbacks,
    verbose=0)
```

Displaying curves of loss and accuracy during training

[21]:
```
accuracy = history.history["accuracy"]
val_accuracy = history.history["val_accuracy"]
loss = history.history["loss"]
val_loss = history.history["val_loss"]
epochs = range(1, len(accuracy) + 1)

plot_helper([epochs, epochs],
            [accuracy, val_accuracy],
            ["bo", "b"],
            ["Training accuracy", "Validation accuracy"],
            "Training and validation accuracy",
            "Epochs",
            "Accuracy")
plot_helper([epochs, epochs],
            [loss, val_loss],
            ["bo", "b"],
            ["Training loss", "Validation loss"],
            "Training and validation loss",
            "Epochs",
            "Loss")
```





These plots are characteristic of overfitting. The training accuracy increases linearly over time, until it reaches nearly 100%, whereas the validation accuracy peaks at around 75%. The validation loss reaches its minimum after only ten epochs and then stalls and starts increasing, whereas the training loss keeps decreasing as training proceeds.

Evaluating the model on the test set

[22]:
```
test_model = keras.models.load_model("models/convnet_from_scratch.keras")
test_loss, test_acc = test_model.evaluate(test_dataset)
print(f"Test accuracy: {test_acc:.3f}")
```

```
63/63 [==============================] - 1s 14ms/step - loss: 0.6239 - accuracy:
0.6545
Test accuracy: 0.655
```

Because we have relatively few training samples (2,000), overfitting will be our number one concern. We already know about a number of techniques that can help mitigate overfitting, such as dropout 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 DL models: **data augmentation**.

Before diving into data augmentation, let's make a short digression on the concept of **callbacks** used in the previous example.

1.3.5 Using callbacks

Starting training on a large dataset for tens of epochs using `model.fit()` can be a bit like launching a paper airplane: after the throw, we don't have any control over its trajectory or its landing spot. If we want to avoid bad outcomes, it's smarter to use, not a paper plane, but a drone that can sense its environment, send data back to its operator, and automatically make steering decisions based on its current state. The Keras **callbacks** API helps us transform our call to `model.fit()` from a paper airplane into a smart, autonomous drone that can self-introspect and dynamically take action.

A callback is an object that is passed to the model in the call to `fit()` and that is called by the model at various points during training. It has access to all the available data about the state of the model and its performance, and it can take action: interrupt training, save a model, load a different weight set, or otherwise alter the state of the model.

Here are some examples of ways we can use callbacks: - *Model checkpointing* - Saving the current state of the model at different points during training. - *Early stopping* - Interrupting training when the validation loss is no longer improving (and of course, saving the best model obtained during training). - *Dynamically adjusting the value of certain parameters during training* - Such as the learning rate of the optimizer. - *Logging training and validation metrics during training, or visualizing the representations learned by the model as they're updated* - The `fit()` progress bar that we're familiar with is in fact a callback!

The `keras.callbacks` module comprises a number of built-in callbacks, including: - `keras.callbacks.ModelCheckpoint` - `keras.callbacks.EarlyStopping` - `keras.callbacks.LearningRateScheduler` - `keras.callbacks.ReduceLROnPlateau` - `keras.callbacks.CSVLogger`

Let's review two of them: `EarlyStopping` and `ModelCheckpoint`.

When we're training a model, there are many things you can't predict from the start. In particular, we can't tell how many epochs will be needed to get to an optimal validation loss. Our examples so far have adopted the strategy of training for enough epochs that we begin overfitting, using the first run to figure out the proper number of epochs to train for, and then finally launching a new training run from scratch using this optimal number. Of course, this approach is wasteful. A better way to handle this is to stop training when the validation loss is no longer improving. This can be achieved using the `EarlyStopping` callback, which interrupts training once a target metric being monitored has stopped improving for a fixed number of epochs (the `patience` argument in the callback).

The `EarlyStopping` callback is typically used in combination with `ModelCheckpoint`, which lets us continually save the model during training. A few options that `ModelCheckpoint` provides: - Whether to only keep the model that has achieved the "best performance" so far, or whether to save the model at the end of every epoch regardless of performance. - Definition of 'best'; which quantity to monitor and whether it should be maximized or minimized. - The frequency it should save at. Currently, the callback supports saving at the end of every epoch, or after a fixed number of training batches. - Whether only weights are saved, or the whole model is saved.

Using the `callbacks` argument in `fit()`

[23]:
```
callbacks_list = [
    keras.callbacks.EarlyStopping(
        monitor="val_accuracy",
        patience=3,
    ),
    keras.callbacks.ModelCheckpoint(
        filepath="models/convnet_from_scratch.keras",
        save_best_only=True,
        monitor="val_loss")
]
model = cat_vs_dogs_model()
model.fit(
    train_dataset,
    epochs=30,
    validation_data=validation_dataset,
    callbacks=callbacks_list)
```

We can always save models manually after training: `model.save("my_checkpoint_path")`.

To reload the model we've saved: `model = keras.models.load_model("models/convnet_from_scratch.keras"`

1.3.6 Using data augmentation

Overfitting is caused by having too few samples to learn from, rendering us unable to train a model that can generalize to new data. Given infinite data, our model would be exposed to every possible aspect of the considered data distribution: we would never overfit. Data augmentation generates more training data from the existing training samples by augmenting the samples via a number of **random transformations** that produce believable-looking images. The goal is that, at training time, our model will never see the exact same picture twice. This helps expose the model to more aspects of the data so it can generalize better.

In Keras, this can be done by adding a number of **data augmentation layers** at the start of our model. Let's get started with an example: the following Sequential model chains several random image transformations. In our model, we'd include it right before the `Rescaling` layer.

Define a data augmentation stage to add to an image model

[24]:
```
data_augmentation = keras.Sequential(
    [
        layers.RandomFlip("horizontal"),
        layers.RandomRotation(0.1),
        layers.RandomZoom(0.2),
    ]
)
```

These are just a few of the layers available (for more, see the Keras documentation). Let's quickly go over this code: - `RandomFlip("horizontal")` - Applies horizontal flipping to a random 50% of the images that go through it - `RandomRotation(0.1)` - Rotates the input images by a random value in the range [–10%, +10%] (these are fractions of a full circle — in degrees, the range would

be [–36 degrees, +36 degrees]) - `RandomZoom(0.2)` - Zooms in or out of the image by a random factor in the range [-20%, +20%].

Displaying some randomly augmented training images

[25]:
```
plt.figure(figsize=(10, 10))
for images, _ in train_dataset.take(1):    # Method `take` creates a Dataset␣
 ↪with at most `count` elements from this dataset
    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")
```

If we train a new model using this data-augmentation configuration, the model will never see the same input twice. But the inputs it sees are still heavily intercorrelated because they come from a small number of original images - we can't produce new information; we can only remix existing information. As such, this may not be enough to completely get rid of overfitting. To further fight overfitting, we'll also add a `Dropout` layer to our model right before the densely connected classifier.

One thing we should know about random image augmentation layers: just like `Dropout`, they're inactive during inference (when we call `predict()` or `evaluate()`). During evaluation, our model will behave just the same as when it did not include data augmentation and dropout.

Defining a new convnet that includes image augmentation and dropout

[26]:
```
def cats_vs_dogs_model_augm_drop():
    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,␣
 ↪name="cats_vs_dogs_augm_drop")
    model.compile(loss="binary_crossentropy",
                  optimizer="rmsprop",
                  metrics=["accuracy"])

    return model
```

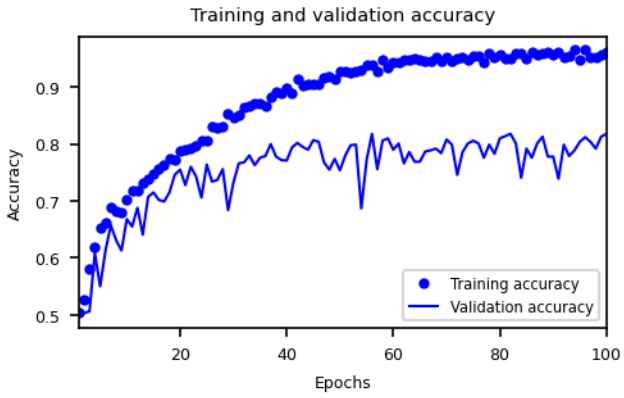
Training the regularized convnet

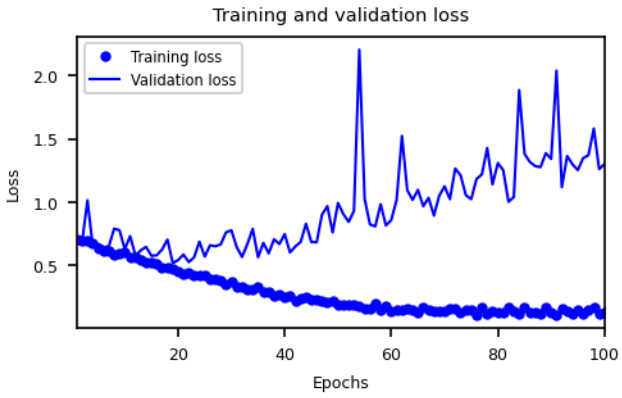
[27]:
```
model = cats_vs_dogs_model_augm_drop()
callbacks = [
    keras.callbacks.ModelCheckpoint(
        filepath="models/convnet_with_augmentation.keras",
        save_best_only=True,
        monitor="val_loss")
]
history = model.fit(
    train_dataset,
    epochs=100,
    validation_data=validation_dataset,
```

```
    batch_size=64,
    callbacks=callbacks,
    verbose=0)
```

[28]:
```
accuracy = history.history["accuracy"]
val_accuracy = history.history["val_accuracy"]
loss = history.history["loss"]
val_loss = history.history["val_loss"]
epochs = range(1, len(accuracy) + 1)

plot_helper([epochs, epochs],
            [accuracy, val_accuracy],
            ["bo", "b"],
            ["Training accuracy", "Validation accuracy"],
            "Training and validation accuracy",
            "Epochs",
            "Accuracy")
plot_helper([epochs, epochs],
            [loss, val_loss],
            ["bo", "b"],
            ["Training loss", "Validation loss"],
            "Training and validation loss",
            "Epochs",
            "Loss")
```





Evaluating the model on the test set

[29]:
```
test_model = keras.models.load_model("models/convnet_with_augmentation.keras")
test_loss, test_acc = test_model.evaluate(test_dataset)
print(f"Test accuracy: {test_acc:.3f}")
```

```
63/63 [==============================] - 1s 14ms/step - loss: 0.5193 - accuracy:
0.7625
Test accuracy: 0.762
```

We got a test accuracy of around 76%, which is a significant improvement over the 65% accuracy!

By further tuning the model's configuration (such as the number of filters per convolution layer, or the number of layers in the model), we might be able to get an even better accuracy, likely up to 90%. But it would prove difficult to go any higher just by training our own convnet from scratch, because we have little data to work with.