3 Convolutional neural networks

This chapter covers

- Classifying images using MLP
- Working with the CNN architecture to classify images
- Understanding convolution on color images

Previously, we talked about artificial neural networks (ANNs), also known as multilayer perceptrons (MLPs), which are basically layers of neurons stacked on top of each other that have learnable weights and biases. Each neuron receives some inputs, which are multiplied by their weights, with nonlinearity applied via activation functions. In this chapter, we will talk about convolutional neural networks (CNNs), which are considered an evolution of the MLP architecture that performs a lot better with images.

The high-level layout of this chapter is as follows:

1. *Image classification with MLP*—We will start with a mini project to classify images using MLP topology and examine how a regular neural network architecture processes images. You will learn about the MLP architecture's drawbacks when processing images and why we need a new, creative neural network architecture for this task.

2. *Understanding CNNs*—We will explore convolutional networks to see how they extract features from images and classify objects. You will learn about the three main components of CNNs: the convolutional layer, the pooling layer, and the fully connected layer. Then we will apply this knowledge in another mini project to classify images with CNNs.
3. *Color images*—We will compare how computers see color images versus grayscale images, and how convolution is implemented over color images.
4. *Image classification project*—We will apply all that you learn in this chapter in an end-to-end image classification project to classify color images with CNNs.

The basic concepts of how the network learns and optimizes parameters are the same with both MLPs and CNNs:

- *Architecture*—MLPs and CNNs are composed of layers of neurons that are stacked on top of each other. CNNs have different structures (convolutional versus fully connected layers), as we are going to see in the coming sections.
- *Weights and biases*—In convolutional and fully connected layers, inference works the same way. Both have weights and biases that are initially randomly generated, and their values are learned by the network. The main difference between them is that the weights in MLPs are in a vector form, whereas in convolutional layers, weights take the form of convolutional filters or kernels.
- *Hyperparameters*—As with MLPs, when we design CNNs we will always specify the error function, activation function, and optimizer. All hyperparameters explained in the previous chapters remain the same; we will add some new ones that are specific to CNNs.
- *Training*—Both networks learn the same way. First they perform a forward pass to get predictions; second, they compare the prediction with the true label to get the loss function ($y - \hat{y}$); and finally, they optimize parameters using gradient descent, backpropagate the error to all the weights, and update their values to minimize the loss function.

Ready? Let's get started!

3.1 Image classification using MLP

Let's recall the MLP architecture from chapter 2. Neurons are stacked in layers on top of each other, with weight connections. The MLP architecture consists of an input layer, one or more hidden layers, and an output layer (figure 3.1).

This section uses what you know about MLPs from chapter 2 to solve an image classification problem using the MNIST dataset. The goal of this classifier will be to classify images of digits from 0 to 9 (10 classes). To begin, let's look at the three main components of our MLP architecture (input layer, hidden layers, and output layer).

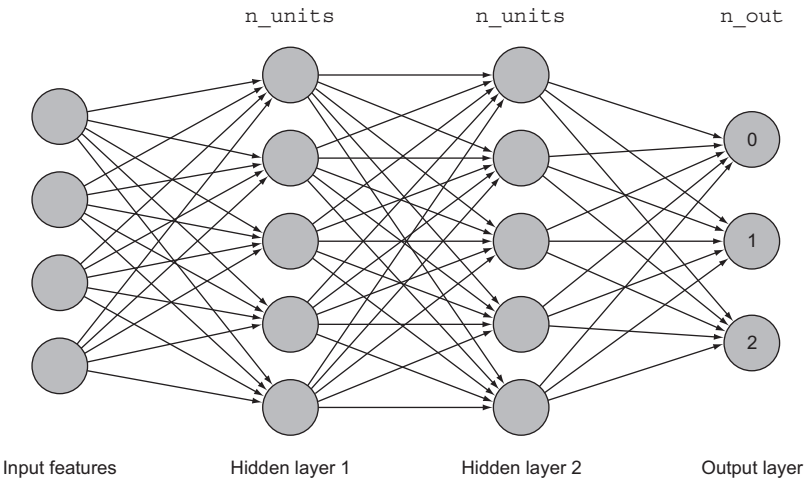


Figure 3.1 The MLP architecture consists of layers of neurons connected by weight connections.

3.1.1 Input layer

When we work with 2D images, we need to preprocess them into something the network can understand before feeding them to the network. First, let's see how computers perceive images. In figure 3.2, we have an image 28 pixels wide × 28 pixels high. This image is seen by the computer as a 28 × 28 matrix, with pixel values ranging from 0 to 255 (0 for black, 255 for white, and the range in between for grayscale).

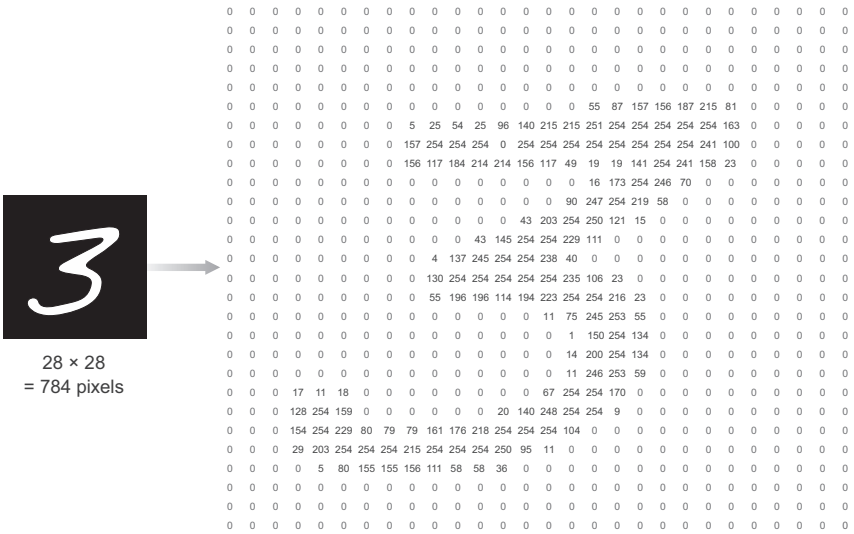


Figure 3.2 The computer sees this image as a 28 × 28 matrix of pixel values ranging from 0 to 255.

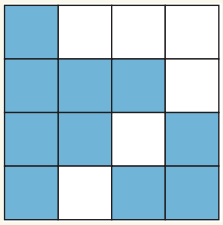
Since MLPs only take as input 1D vectors with dimensions (1, n), they cannot take a raw 2D image matrix with dimensions (x, y). To fit the image in the input layer, we first need to transform our image into one large vector with the dimensions (1, n) that contains all the pixel values in the image. This process is called *image flattening*. In this example, the total number (n) of pixels in this image is 28 × 28 = 784. Then, in order to feed this image to our network, we need to flatten the (28 × 28) matrix into one long vector with dimensions (1, 784). The input vector looks like this:

$$x = [row1, row2, row3, ..., row28]$$

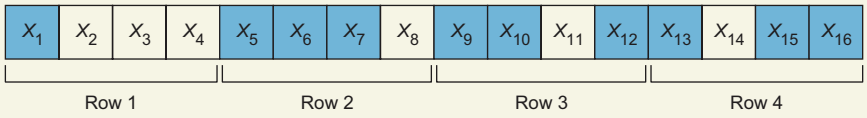
That said, the input layer in this example will have a total of 784 nodes: x_1, x_2, …, x_{784}.

Visualizing input vectors

To help visualize the flattened input vector, let's look at a much smaller matrix (4, 4):



The input (x) is a flattened vector with the dimensions (1, 16):



So, if we have pixel values of 0 for black and 255 for white, the input vector will be as follows:

Input = [0, 255, 255, 255, 0, 0, 0, 255, 0, 0, 255, 0, 0, 255, 0, 0]

Here is how we flatten an input image in Keras:

```
from keras.models import Sequential
from keras.layers import Flatten

model = Sequential()
model.add( Flatten(input_shape = (28,28) ))
```

Defines the model

As before, imports the Keras library

Imports a layer called Flatten to convert the image matrix into a vector

Adds the Flatten layer, also known as the input layer

The `Flatten` layer in Keras handles this process for us. It takes the 2D image matrix input and converts it into a 1D vector. Note that the `Flatten` layer must be supplied a parameter value of the shape of the input image. Now the image is ready to be fed to the neural network.

What's next? Hidden layers.

3.1.2 Hidden layers

As discussed in the previous chapter, the neural network can have one or more hidden layers (technically, as many as you want). Each layer has one or more neurons (again, as many as you want). Your main job as a neural network engineer is to design these layers. For the sake of this example, let's say you decided to arbitrarily design the network to have two hidden layers, each having 512 nodes—and don't forget to add the ReLU activation function for each hidden layer.

> **Choosing an activation function**
>
> In chapter 2, we discussed the different types of activation functions in detail. As a DL engineer, you will often have a lot of different choices when you are building your network. Choosing the activation function that is the most suitable for the problem you are solving is one of these choices. While there is no single best answer that fits all problems, in most cases, the ReLU function performs best in the hidden layers; and for most classification problems where classes are mutually exclusive, softmax is generally a good choice in the output layer. The softmax function gives us the probability that the input image depicts one of the (*n*) classes.

As in the previous chapter, let's add two fully connected (also known as *dense*) layers, using Keras:

```
from keras.layers import Dense
```

Imports the Dense layer

```
model.add(Dense(512, activation = 'relu'))
model.add(Dense(512, activation = 'relu'))
```

Adds two Dense layers with 512 nodes each

3.1.3 Output layer

The output layer is pretty straightforward. In classification problems, the number of nodes in the output layer should be equal to the number of classes that you are trying to detect. In this problem, we are classifying 10 digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9). Then we need to add one last `Dense` layer that contains 10 nodes:

```
model.add(Dense(10, activation = 'softmax'))
```

3.1.4 Putting it all together

When we put all these layers together, we get a neural network like the one in figure 3.3.

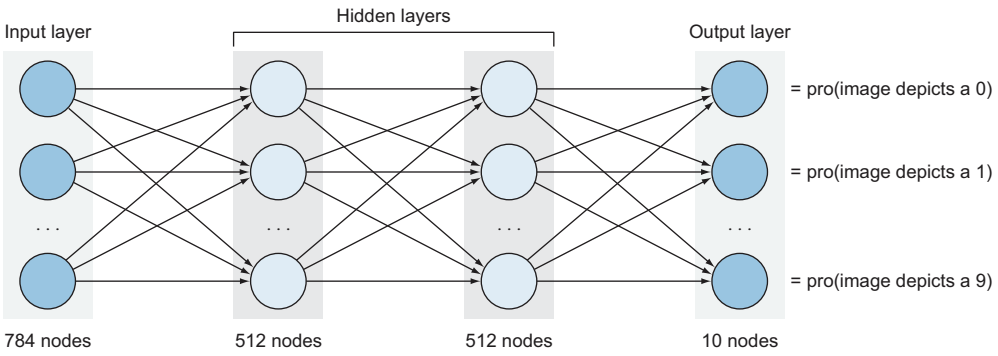


Figure 3.3 The neural network we create by combining the input, hidden, and output layers

Here is how it looks in Keras:

```
from keras.models import Sequential
from keras.layers import Flatten, Dense

model = Sequential()

model.add( Flatten(input_shape = (28,28) ))

model.add(Dense(512, activation = 'relu'))
model.add(Dense(512, activation = 'relu'))

model.add(Dense(10, activation = 'softmax'))
model.summary()
```

Imports the Keras library

Imports a Flatten layer to convert the image matrix into a vector

Defines the neural network architecture

Adds the Flatten layer

Adds 2 hidden layers with 512 nodes each. Using the ReLU activation function is recommended in hidden layers.

Adds 1 output Dense layer with 10 nodes. Using the softmax activation function is recommended in the output layer for multiclass classification problems.

Prints a summary of the model architecture

When you run this code, you will see the model summary printed as shown in figure 3.4.

You can see that the output of the flatten layer is a vector with 784 nodes, as discussed before, since we have 784 pixels in each 28 × 28 images. As designed, the hidden layers produce 512 nodes each; and, finally, the output layer (`dense_3`) produces a layer with 10 nodes.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
Flatten_1 (Flatten)          (None, 784)               0
_________________________________________________________________
dense_1 (Dense)              (None, 512)               401920
_________________________________________________________________
dense_2 (Dense)              (None, 512)               262656
_________________________________________________________________
dense_3 (Dense)              (None, 10)                5130
=================================================================
Total params: 669,706
Trainable params: 669,706
Non-trainable params: 0
_________________________________________________________________
```

Figure 3.4 The model summary

The Param # field represents the number of parameters (weights) produced at each layer. These are the weights that will be adjusted and learned during the training process. They are calculated as follows:

1. Params after the flatten layer = 0, because this layer only flattens the image to a vector for feeding into the input layer. The weights haven't been added yet.
2. Params after layer 1 = (784 nodes in input layer) × (512 in hidden layer 1) + (512 connections to biases) = 401,920.
3. Params after layer 2 = (512 nodes in hidden layer 1) × (512 in hidden layer 2) + (512 connections to biases) = 262,656.
4. Params after layer 3= (512 nodes in hidden layer 2) × (10 in output layer) + (10 connections to biases) = 5,130.
5. Total params in the network = 401,920 + 262,656 + 5,130 = 669,706.

This means that in this tiny network, we have a total of 669,706 parameters (weights and biases) that the network needs to learn and whose values it needs to tune to optimize the error function. This is a huge number for such a small network. You can see how this number would grow out of control if we added more nodes and layers or used bigger images. This is one of the two major drawbacks of MLPs that we will discuss next.

MLPs vs. CNNs

If you train the example MLP on the MNIST dataset, you will get pretty good results (close to 96% accuracy compared to 99% with CNNs). But MLPs and CNNs do not usually yield comparable results. The MNIST dataset is special because it is very clean and perfectly preprocessed. For example, all images have the same size and are centered in a 28 × 28 pixel grid. Also, the MNIST dataset contains only grayscale images. It would be a much harder task if the images had color or the digits were skewed or not centered.

If you try the example MLP architecture with a slightly more complex dataset like CIFAR-10, as we will do in the project at the end of this chapter, the network will perform very poorly (around 30–40% accuracy). It performs even worse with more complex datasets. In messy real-world image data, CNNs truly outshine MLPs.

3.1.5 Drawbacks of MLPs for processing images

We are nearly ready to talk about the topic of this chapter: CNNs. But first, let's discuss the two major problems in MLPs that convolutional networks are designed to fix.

SPATIAL FEATURE LOSS

Flattening a 2D image to a 1D vector input results in losing the spatial features of the image. As we saw in the mini project earlier, before feeding an image to the hidden layers of an MLP, we must flatten the image matrix to a 1D vector. This means throwing away all the 2D information contained in the image. Treating an input as a simple vector of numbers with no special structure might work well for 1D signals; but in 2D images, it will lead to information loss because the network doesn't relate the pixel values to each other when trying to find patterns. MLPs have no knowledge of the fact that these pixel numbers were originally spatially arranged in a grid and that they are connected to each other. CNNs, on the other hand, do not require a flattened image. We can feed the raw image matrix of pixels to a CNN network, and the CNN will understand that pixels that are close to each other are more heavily related than pixels that are far apart.

Let's oversimplify things to learn more about the importance of spatial features in an image. Suppose we are trying to teach a neural network to identify the shape of a square, and suppose the pixel value 1 is white and 0 is black. When we draw a white square on a black background, the matrix will look like figure 3.5.

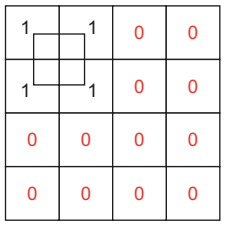


Figure 3.5 If the pixel value 1 is white and 0 is black, this is what our matrix looks like for identifying a square.

Since MLPs take a 1D vector as an input, we have to flatten the 2D image to a 1D vector. The input vector of figure 3.5 looks like this:

Input vector = [1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

When the training is complete, the network will learn to identify a square *only* when the input nodes x_1, x_2, x_5, and x_6 are fired. But what happens when we have new

images with square shapes located in different areas in the image, as shown in figure 3.6?

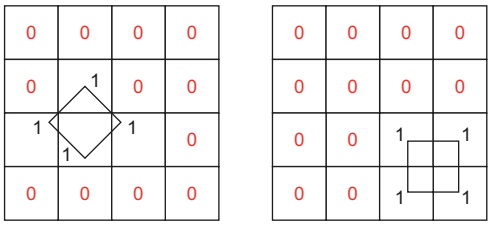


Figure 3.6 Square shapes in different areas of the image

The MLP will have no idea that these are the shapes of squares because the network didn't learn the square shape as a feature. Instead, it learned the input nodes that, when fired, might lead to a square shape. If we want our network to learn squares, we need a lot of square shapes located everywhere in the image. You can see how this solution won't scale for complex problems.

Another example of feature learning is this: if we want to teach a neural network to recognize cats, then ideally, we want the network to learn all the shapes of cat features regardless of where they appear on the image (ears, nose, eyes, and so on). This only happens when the network looks at the image as a set of pixels that, when close to each other, are heavily related.

The mechanism of how CNNs learn will be explained in detail in this chapter. But figure 3.7 shows how the network learns features throughout its layers.

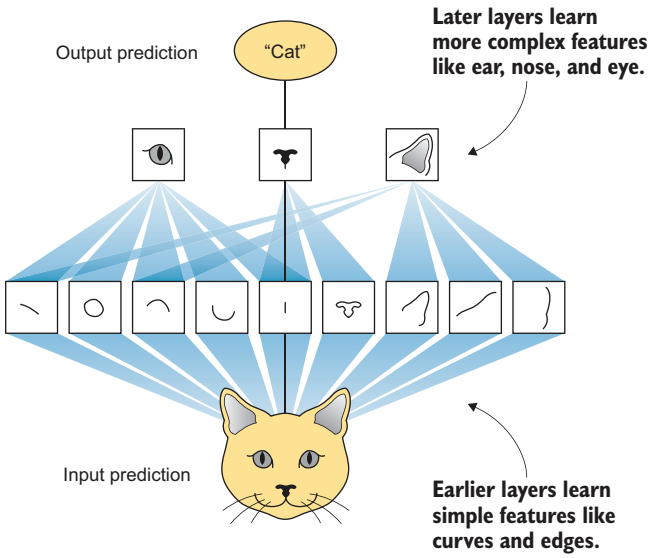


Figure 3.7 CNNs learn the image features through its layers.

Fully connected (dense) layers

MLPs are composed of dense layers that are fully connected to each other. *Fully connected* means every node in one layer is connected to *all* nodes of the previous layer and *all* nodes in the next layer. In this scenario, each neuron has parameters (weights) to train per neuron from the previous layer. While this is not a big problem for the MNIST dataset because the images are really small in size (28 × 28), what happens when we try to process larger images? For example, if we have an image with dimensions 1,000 × 1,000, it will yield 1 million parameters for each node in the first hidden layer. So if the first hidden layer has 1,000 neurons, this will yield 1 billion parameters even in such a small network. You can imagine the computational complexity of optimizing 1 billion parameters after only the first layer. This number will increase drastically when we have tens or hundreds of layers. This can get out of control pretty fast and will not scale.

CNNs, on the other hand, are *locally connected* layers, as figure 3.8 shows: nodes are connected to only a small subset of the previous layers' nodes. Locally connected layers use far fewer parameters than densely connected layers, as you will see.

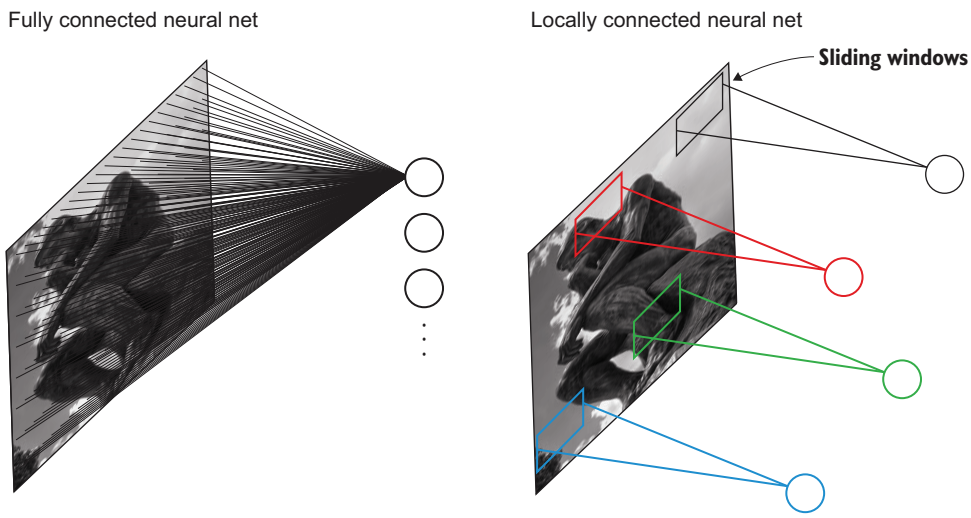


Figure 3.8 (Left) Fully connected neural network where all neurons are connected to all pixels of the image. (Right) Locally connected network where only a subset of pixels is connected to each neuron. These subsets are called *sliding windows*.

What does it all mean?

The loss of information caused by flattening a 2D image matrix to a 1D vector and the computational complexity of fully connected layers with larger images suggest that we need an entirely new way of processing image input, one where 2D information is not entirely lost. This is where convolutional networks come in. CNNs accept the full

image matrix as input, which significantly helps the network understand the patterns contained in the pixel values.

3.2 CNN architecture

Regular neural networks contain multiple layers that allow each layer to find successively complex features, and this is the way CNNs work. The first layer of convolutions learns some basic features (edges and lines), the next layer learns features that are a little more complex (circles, squares, and so on), the following layer finds even more complex features (like parts of the face, a car wheel, dog whiskers, and the like), and so on. You will see this demonstrated shortly. For now, know that the CNN architecture follows the same pattern as neural networks: we stack neurons in hidden layers on top of each other; weights are randomly initiated and learned during network training; and we apply activation functions, calculate the error $(y - \hat{y})$, and backpropagate the error to update the weights. This process is the same. The difference is that we use convolutional layers instead of regular fully connected layers for the feature-learning part.

3.2.1 The big picture

Before we look in detail at the CNN architecture, let's back up for a moment to see the big picture (figure 3.9). Remember the image classification pipeline we discussed in chapter 1?

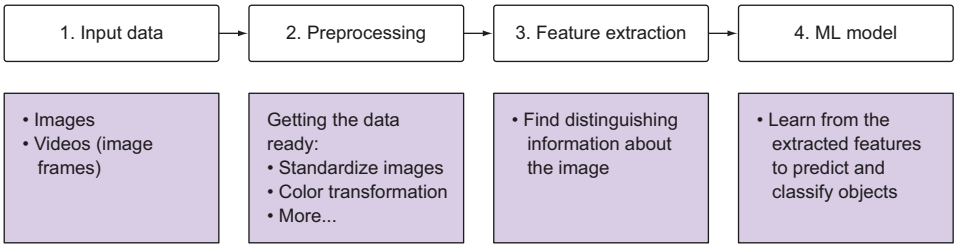


Figure 3.9 The image classification pipeline consists of four components: data input, data preprocessing, feature extraction, and the ML algorithm.

Before deep learning (DL), we used to manually extract features from images and then feed the resulting feature vector to a classifier (a regular ML algorithm like SVM). With the magic that neural networks provide, we can replace the manual work of step 3 in figure 3.9 with a neural network (MLP or CNN) that does both feature learning and classification (steps 3 and 4).

We saw earlier, in the digit-classification project, how to use MLP to learn features and classify an image (steps 3 and 4 together). It turned out that our issue with fully connected layers was not the classification part—fully connected layers do that very

well. Our issue was in the way fully connected layers process the image to learn features. Let's get a little creative: we'll keep what's working and make modifications to what's not working. The fully connected layers aren't doing a good job of feature extraction (step 3), so let's replace that with locally connected layers (convolutional layers). On the other hand, fully connected layers do a great job of classifying the extracted features (step 4), so let's keep them for the classification part.

The high-level architecture of CNNs looks like figure 3.10:

- Input layer
- Convolutional layers for feature extraction
- Fully connected layers for classification
- Output prediction

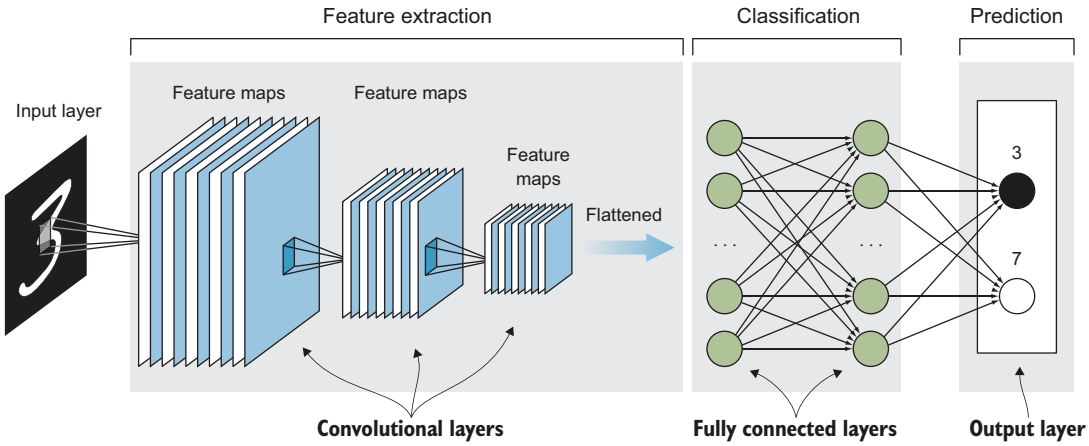


Figure 3.10 The CNN architecture consists of the following: input layer, convolutional layers, fully connected layers, and output prediction.

Remember, we are still talking about the big picture. We will dive into each of these components soon. In figure 3.10, suppose we are building a CNN to classify images into two classes: the numbers 3 and 7. Look at the figure, and follow along with these steps:

1. Feed the raw image to the convolutional layers.
2. The image passes through the CNN layers to detect patterns and extract features called *feature maps*. The output of this step is then flattened to a vector of the learned features of the image. Notice that the image dimensions shrink after each layer, and the number of feature maps (the layer depth) increases until we have a long array of small features in the last layer of the feature-extraction part. Conceptually, you can think of this step as the neural network learning to represent more abstract features of the original image.

3 The flattened feature vector is fed to the fully connected layers to classify the extracted features of the image.
4 The neural network fires the node that represents the correct prediction of the image. Note that in this example, we are classifying two classes (3 and 7). Thus the output layer will have two nodes: one to represent the digit 3, and one for the digit 7.

DEFINITION The basic idea of neural networks is that neurons learn features from the input. In CNNs, a *feature map* is the output of one filter applied to the previous layer. It is called a feature map because it is a mapping of where a certain kind of feature is found in the image. CNNs look for features such as straight lines, edges, or even objects. Whenever they spot these features, they report them to the feature map. Each feature map is looking for something specific: one could be looking for straight lines and another for curves.

3.2.2 A closer look at feature extraction

You can think of the feature-extraction step as breaking large images into smaller pieces of features and stacking them into a vector. For example, an image of the digit 3 is one image (depth = 1) and is broken into smaller images that contain specific features of the digit 3 (figure 3.11). If it is broken into four features, then the depth equals 4. As the image passes through the CNN layers, it shrinks in dimensions, and the layer gets deeper because it contains more images of smaller features.

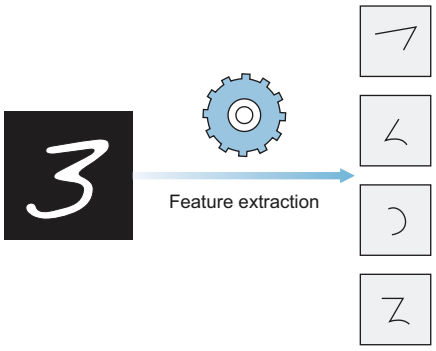


Figure 3.11 An image is broken into smaller images that contain distinctive features.

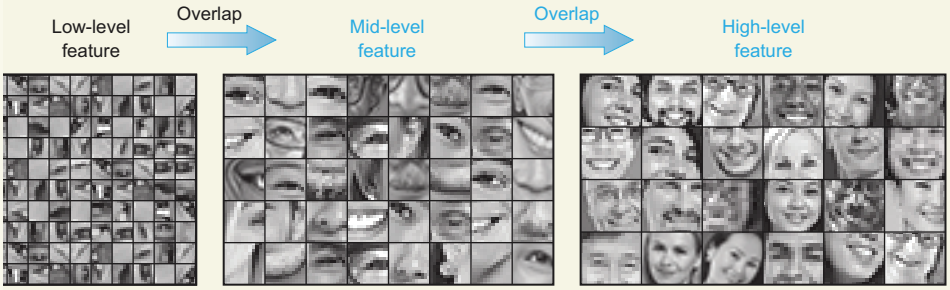
Note that this is just a metaphor to help visualize the feature-extraction process. CNNs don't literally break an image into pieces. Instead, they *extract meaningful features* that separate this object from other images in the training set, and stack them in an array of features.

3.2.3 A closer look at classification

After feature extraction is complete, we add fully connected layers (a regular MLP) to look at the features vector and say, "The first feature (top) has what looks like an edge: this could be 3, or 7, or maybe an ugly 2. I'm not sure; let's look at the second feature. Hmm, this is definitely not a 7 because it has a curve," and so on until the MLP is confident that the image is the digit 3.

How CNNs learn patterns

It is important to note that a CNN doesn't go from the image input to the features vector directly in one layer. This usually happens in tens or hundreds of layers, as you will see later in this chapter. The feature-learning process happens step by step after each hidden layer. So the first layer usually learns very basic features like lines and edges, and the second assembles those lines into recognizable shapes, corners, and circles. Then, in the deeper layers, the network learns more complex shapes such as human hands, eyes, ears, and so on. For example, here is a simplified version of how CNNs learn faces.



A simplified version of how CNNs learn faces

You can see that the early layers detect patterns in the image to learn low-level features like edges, and the later layers detect *patterns within patterns* to learn more complex features like parts of the face, then *patterns within patterns within patterns*, and so on:

Input image

+ Layer 1 ⇒ patterns

+ Layer 2 ⇒ patterns within patterns

+ Layer 3 ⇒ patterns within patterns within patterns

... and so on

This concept will come in handy when we discuss more advanced CNN architectures in later chapters. For now, know that in neural networks, we stack hidden layers to learn patterns from each other until we have an array of meaningful features to identify the image.

3.3 Basic components of a CNN

Without further ado, let's discuss the main components of a CNN architecture. There are three main types of layers that you will see in almost every convolutional network (figure 3.12):

1. Convolutional layer (CONV)
2. Pooling layer (POOL)
3. Fully connected layer (FC)

CNN text representation

The text representation of the architecture in figure 3.12 goes like this:

CNN architecture: INPUT ⇒ CONV ⇒ RELU ⇒ POOL ⇒ CONV ⇒ RELU ⇒ POOL ⇒ FC ⇒ SOFTMAX

Note that the ReLU and softmax activation functions are not really standalone layers—they are the activation functions used in the previous layer. The reason they are shown this way in the text representation is to call out that the CNN designer is using the ReLU activation function in the convolutional layers and softmax activation in the fully connected layer. So this represents a CNN architecture that contains two convolutional layers plus one fully connected layer. You can add as many convolutional and fully connected layers as you see fit. The convolutional layers are for feature learning or extraction, and the fully connected layers are for classification.

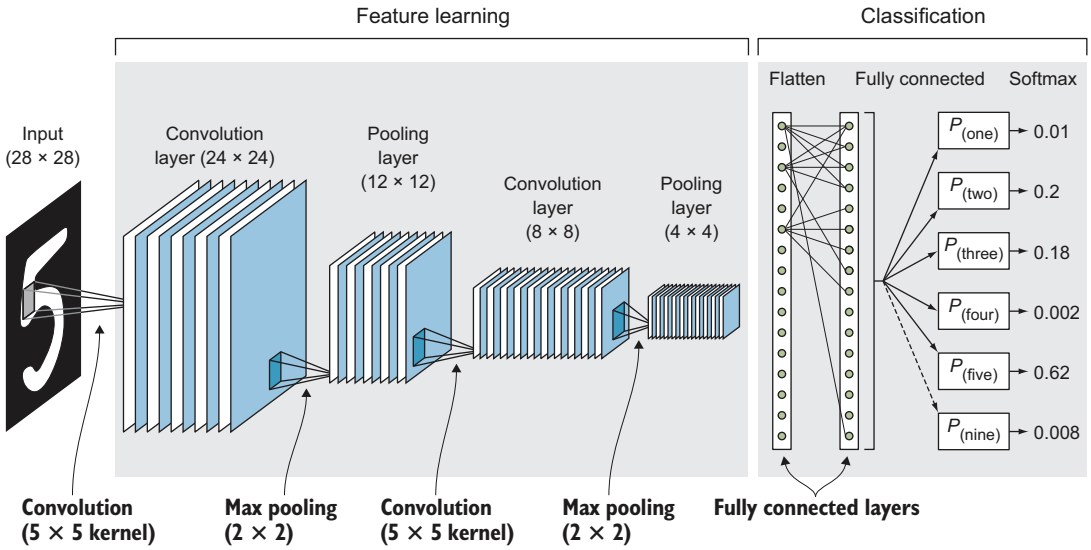


Figure 3.12 The basic components of convolutional networks are convolutional layers and pooling layers to perform feature extraction, and fully connected layers for classification.

Now that we've seen the full architecture of a convolutional network, let's dive deeper into each of the layer types to get a deeper understanding of how they work. Then at the end of this section, we will put them all back together.

3.3.1 Convolutional layers

A convolutional layer is the core building block of a convolutional neural network. Convolutional layers act like a feature finder window that slides over the image pixel by pixel to extract meaningful features that identify the objects in the image.

WHAT IS CONVOLUTION?

In mathematics, convolution is the operation of two functions to produce a third modified function. In the context of CNNs, the first function is the input image, and the second function is the convolutional filter. We will perform some mathematical operations to produce a modified image with new pixel values.

Let's zoom in on the first convolutional layer to see how it processes an image (figure 3.13). By sliding the convolutional filter over the input image, the network breaks the image into little chunks and processes those chunks individually to assemble the modified image, a feature map.

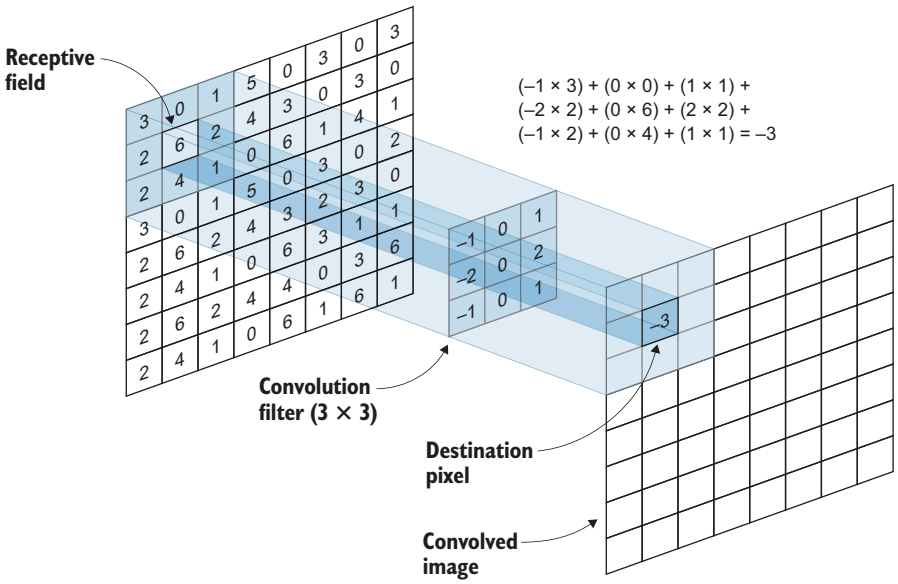


Figure 3.13 A 3 × 3 convolutional filter is sliding over the input image.

Keeping this diagram in mind, here are some facts about convolution filters:

- The small 3 × 3 matrix in the middle *is* the convolution filter, also called a *kernel*.
- The kernel slides over the original image pixel by pixel and does some math calculations to get the values of the new "convolved" image on the next layer.

The area of the image that the filter convolves is called the *receptive field* (see figure 3.14).

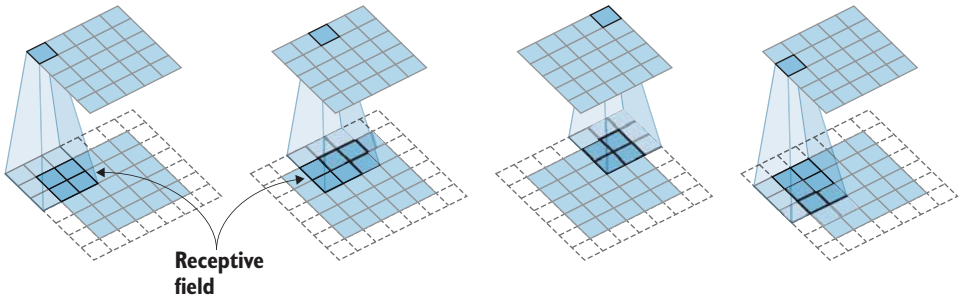


Figure 3.14 The kernel slides over the original image pixel by pixel and calculates the convolved image on the next layer. The convolved area is called the *receptive field*.

What are the kernel values? In CNNs, the convolution matrix *is* the weights. This means they are also *randomly initialized* and the values are *learned* by the network (so you will not have to worry about assigning its values).

CONVOLUTIONAL OPERATIONS

The math should look familiar from our discussion of MLPs. Remember how we multiplied the input by the weights and summed them all together to get the weighted sum?

$$\text{weighted sum} = x_1 \cdot w_1 + x_2 \cdot w_2 + x_3 \cdot w_3 + \ldots + x_n \cdot w_n + b$$

We do the same thing here, except that in CNNs, the neurons and weights are structured in a matrix shape. So we multiply each pixel in the receptive field by the corresponding pixel in the convolution filter and sum them all together to get the value of the center pixel in the new image (figure 3.15). This is the same matrix dot product we saw in chapter 2:

$$(93 \times -1) + (139 \times 0) + (101 \times 1) + (26 \times -2) + (252 \times 0) + (196 \times 2) + (135 \times -1) + (240 \times 0) + (48 \times 1) = 243$$

The filter (or kernel) slides over the whole image. Each time, we multiply every corresponding pixel element-wise and then add them all together to create a new image with new pixel values. This convolved image is called a *feature map* or *activation map*.

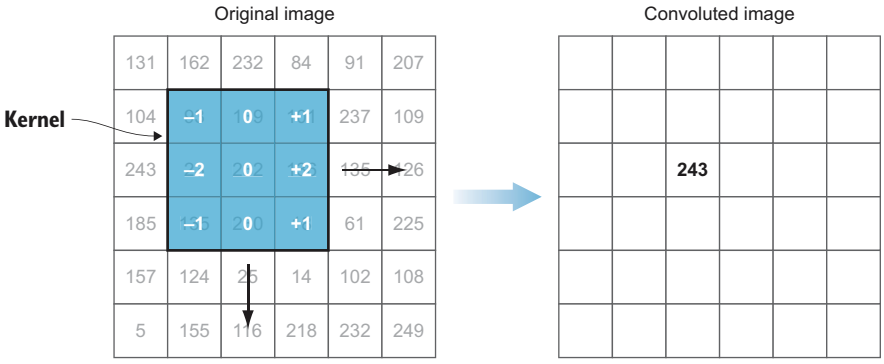


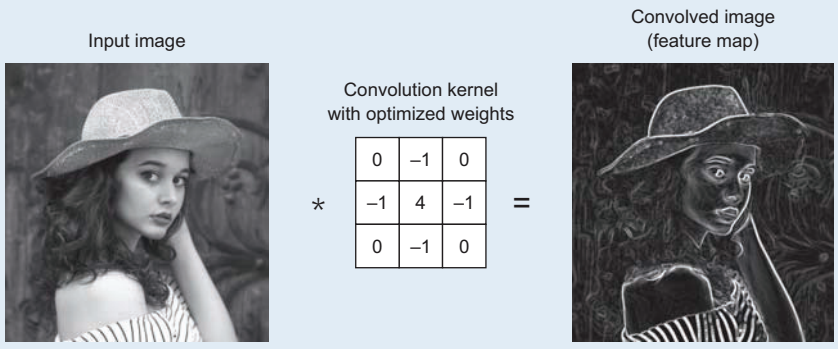
Figure 3.15 Multiplying each pixel in the receptive field by the corresponding pixel in the convolution filter and summing them gives the value of the center pixel in the new image.

Applying filters to learn features

Let's not lose focus of the initial goal. We are doing all this so the network extracts features from the image. How does applying filters lead toward this goal? In image processing, filters are used to filter out unwanted information or amplify features in an image. These filters are matrices of numbers that convolve with the input image to modify it. Look at this edge-detection filter:

0	–1	0
–1	4	–1
0	–1	0

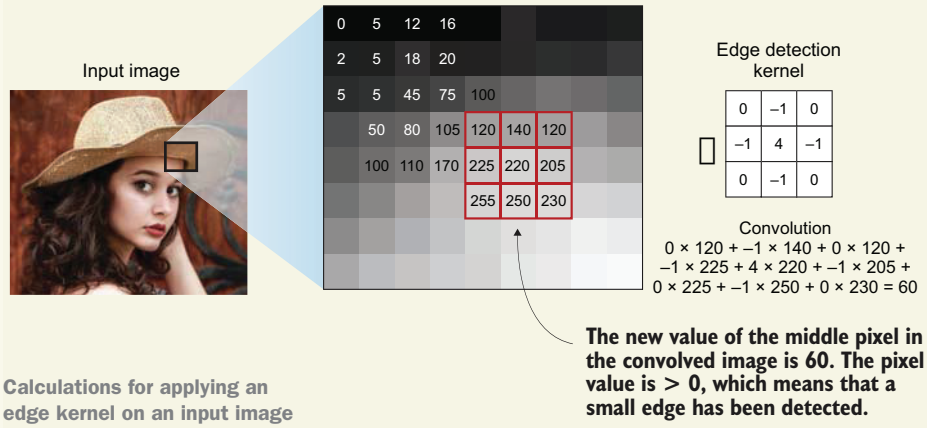
When this kernel (K) is convolved with the input image $F(x,y)$, it creates a new convolved image (a feature map) that amplifies the edges.



Applying an edge detection kernel on an image

(continued)

To understand how the convolution happens, let's zoom in on a small piece of the image.



Calculations for applying an edge kernel on an input image

This image shows the convolution calculations in one area of the image to compute the value of one pixel. We compute the values of all the pixels by sliding the kernel over the input image pixel by pixel and applying the same convolution process.

These kernels are often called *weights* because they determine how important a pixel is in forming a new output image. Similar to what we discussed about MLP and weights, these weights represent the importance of the feature on the output. In images, the input features are the pixel values.

Other filters can be applied to detect different types of features. For example, some filters detect horizontal edges, others detect vertical edges, still others detect more complex shapes like corners, and so on. The point is that these filters, when applied in the convolutional layers, yield the feature-learning behavior we discussed earlier: first they learn simple features like edges and straight lines, and later layers learn more complex features.

We are basically done with the concept of filter. That is all there is to it!

Now, let's take a look at the convolutional layer as a whole: Each convolutional layer contains one or more convolutional filters. The number of filters in each convolutional layer determines the depth of the next layer, because each filter produces its own feature map (convolved image). Let's look at the convolutional layers in Keras to see how they work:

```
from keras.layers import Conv2D

model.add(Conv2D(filters=16, kernel_size=2, strides='1', padding='same',
          activation='relu'))
```

And there you have it. One line of code creates the convolutional layer. We will see where this line fits in the full code later in this chapter. Let's stay focused on the convolutional layer. As you can see from the code, the convolutional layer takes five main arguments. As mentioned in chapter 2, it is recommended that we use the ReLU activation function in the neural networks' hidden layers. That's one argument out of the way. Now, let's explain the remaining four hyperparameters that control the size and depth of the output volume:

- Filters: the number of convolutional filters in each layer. This represents the depth of its output.
- Kernel size: the size of the convolutional filter matrix. Sizes vary: 2×2, 3×3, 5×5.
- Stride.
- Padding.

We will discuss strides and padding in the next section. But now, let's look at each of these four hyperparameters.

> **NOTE** As you learned in chapter 2 on deep learning, hyperparameters are the knobs you tune (increase and decrease) when configuring your neural network to improve performance.

NUMBER OF FILTERS IN THE CONVOLUTIONAL LAYER

Each convolutional layer has one or more filters. To understand this, let's review MLPs from chapter 2. Remember how we stacked neurons in hidden layers, and each hidden layer has *n* number of neurons (also called *hidden units*)? Figure 3.16 shows the MLP diagram from chapter 2.

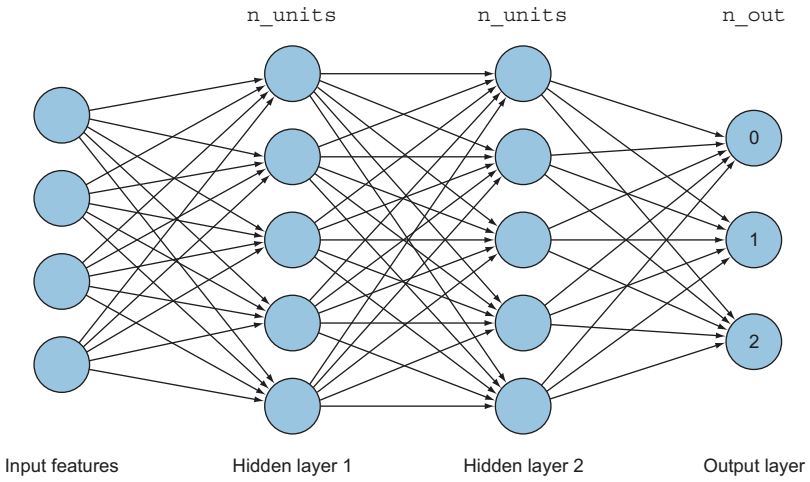


Figure 3.16 Neurons are stacked in hidden layers, and each hidden layer has *n* neurons (hidden units).

Similarly, with CNNs, the convolutional layers are the hidden layers. And to increase the number of neurons in hidden layers, we increase the number of kernels in convolutional layers. Each kernel unit is considered a neuron. For example, if we have a 3×3 kernel in the convolutional layer, this means we have 9 hidden units in this layer. When we add another 3×3 kernel, we have 18 hidden units. Add another one, and we have 27, and so on. So, by increasing the number of kernels in a convolutional layer, we increase the number of hidden units, which makes our network more complex and able to detect more complex patterns. The same was true when we added more neurons (hidden units) to the hidden layers in the MLP. Figure 3.17 provides a representation of the CNN layers that shows the number-of-kernels idea.

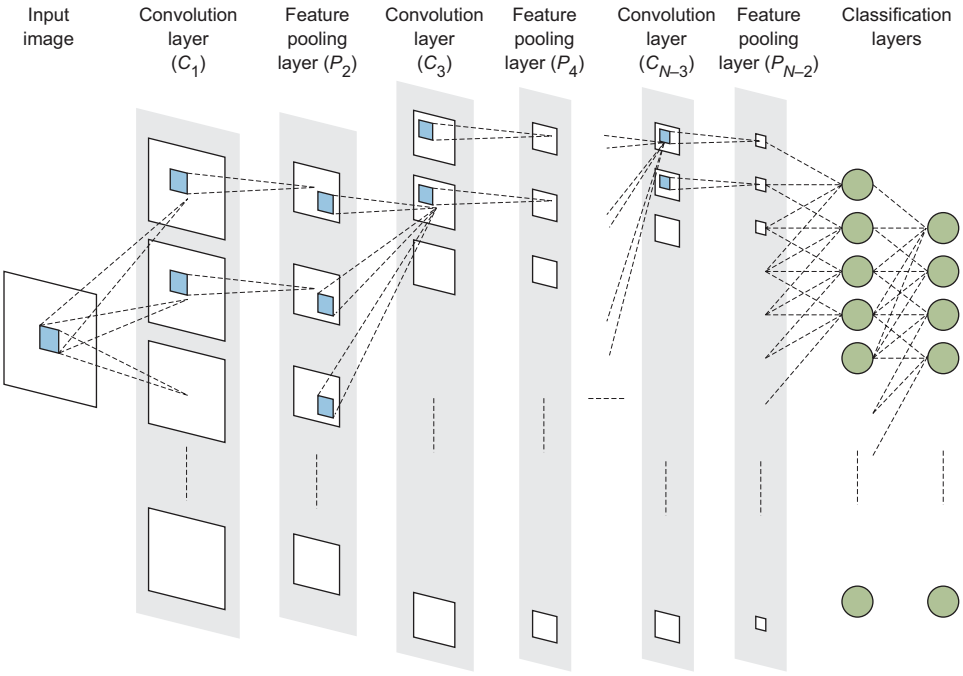


Figure 3.17 Representation of the CNN layers that shows the number-of-kernels idea

KERNEL SIZE

Remember that a convolution filter is also known as a *kernel*. It is a matrix of weights that slides over the image to extract features. The kernel size refers to the dimensions of the convolution filter (width times height; figure 3.18).

`kernel_size` is one of the hyperparameters that you will be setting when building a convolutional layer. Like most neural network hyperparameters, no single best answer fits all problems. The intuition is that smaller filters will capture very fine details of the image, and bigger filters will miss minute details in the image.

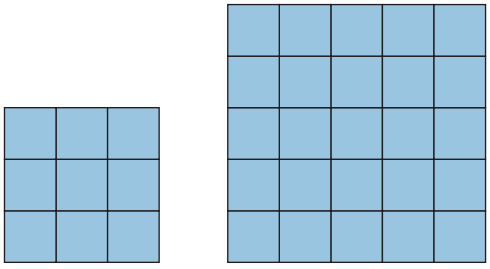


Figure 3.18 The kernel size refers to the dimensions of the convolution filter.

Remember that filters contain the weights that will be learned by the network. So, theoretically, the bigger the `kernel_size`, the deeper the network, which means the better it learns. However, this comes with higher computational complexity and might lead to overfitting.

Kernel filters are almost always square and range from the smallest at 2 × 2 to the largest at 5 × 5. Theoretically, you can use bigger filters, but this is not preferred because it results in losing important image details.

Tuning

I don't want you to get overwhelmed with all the hyperparameter tuning. Deep learning is really an art as well as a science. I can't emphasize this enough: most of your work as a DL engineer will be spent not building the actual algorithms, but rather building your network architecture and setting, experimenting, and tuning your hyperparameters. A great deal of research today is focused on trying to find the optimal topologies and parameters for a CNN, given a type of problem. Fortunately, the problem of tuning hyperparameters doesn't have to be as hard as it might seem. Throughout the book, I will indicate good starting points for using hyperparameters and help you develop an instinct for evaluating your model and analyzing its results to know which knob (hyperparameter) you need to tune (increase or decrease).

Strides and padding

You will usually think of these two hyperparameters together, because they both control the shape of the output of a convolutional layer. Let's see how:

- *Strides*—The amount by which the filter slides over the image. For example, to slide the convolution filter one pixel at a time, the strides value is 1. If we want to jump two pixels at a time, the strides value is 2. Strides of 3 or more are uncommon and rare in practice. Jumping pixels produces smaller output volumes spatially.

  Strides of 1 will make the output image roughly the same height and width of the input image, while strides of 2 will make the output image roughly half of the input image size. I say "roughly" because it depends on what you set the padding parameter to do with the edge of the image.

- *Padding*—Often called *zero-padding* because we add zeros around the border of an image (figure 3.19). Padding is most commonly used to allow us to preserve the spatial size of the input volume so the input and output width and height are the same. This way, we can use convolutional layers without necessarily shrinking the height and width of the volumes. This is important for building deeper networks, since, otherwise, the height/width would shrink as we went to deeper layers.

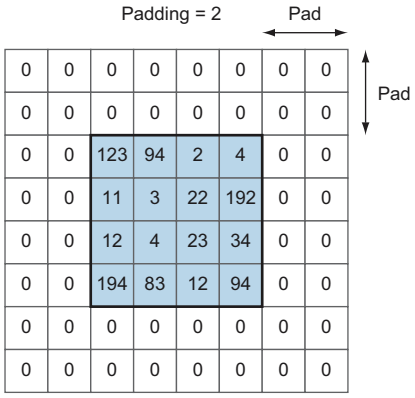


Figure 3.19 Zero-padding adds zeros around the border of the image. Padding = 2 adds two layers of zeros around the border.

NOTE The goal when using strides and padding hyperparameters is one of two things: keep all the important details of the image and transfer them to the next layer (when the `strides` value is `1` and the `padding` value is `same`); or ignore some of the spatial information of the image to make the processing computationally more affordable. Note that we will be adding the pooling layer (discussed next) to reduce the size of the image to focus on the extracted features. For now, know that strides and padding hyperparameters are meant to control the behavior of the convolutional layer and the size of its output: whether to pass on all image details or ignore some of them.

3.3.2 Pooling layers or subsampling

Adding more convolutional layers increases the depth of the output layer, which leads to increasing the number of parameters that the network needs to optimize (learn). You can see that adding several convolutional layers (usually tens or even hundreds) will produce a huge number of parameters (weights). This increase in network dimensionality increases the time and space complexity of the mathematical operations that take place in the learning process. This is when pooling layers come in handy. *Subsampling* or *pooling* helps reduce the size of the network by reducing the number of parameters passed to the next layer. The pooling operation resizes its input by applying a summary statistical function, such as a maximum or average, to reduce the overall number of parameters passed on to the next layer.

The goal of the pooling layer is to downsample the feature maps produced by the convolutional layer into a smaller number of parameters, thus reducing computational complexity. It is a common practice to add pooling layers after every one or two convolutional layers in the CNN architecture (figure 3.20).

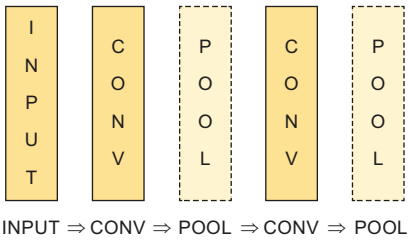


Figure 3.20 Pooling layers are commonly added after every one or two convolutional layers.

MAX POOLING VS. AVERAGE POOLING

There are two main types of pooling layers: max pooling and average pooling. We will discuss max pooling first.

Similar to convolutional kernels, max pooling kernels are windows of a certain size and strides value that slide over the image. The difference with max pooling is that the windows don't have weights or any values. All they do is slide over the feature map created by the previous convolutional layer and select the max pixel value to pass along to the next layer, ignoring the remaining values. In figure 3.21, you see a pooling filter with a size of 2 × 2 and strides of 2 (the filter jumps 2 pixels when sliding over the image). This pooling layer reduces the feature map size from 4 × 4 down to 2 × 2.

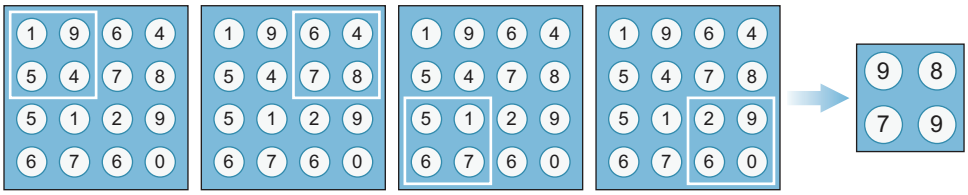


Figure 3.21 A 2 × 2 pooling filter and strides of 2, reducing the feature map from 4 × 4 to 2 × 2

When we do that to all the feature maps in the convolutional layer, we get maps of smaller dimensions (width times height), but the depth of the layer is kept the same because we apply the pooling filter to each of the feature maps from the previous filter. So if the convolutional layer has three feature maps, the output of the pooling layer will also have three feature maps, but of smaller size (figure 3.22).

Global average pooling is a more extreme type of dimensionality reduction. Instead of setting a window size and strides, global average pooling calculates the average

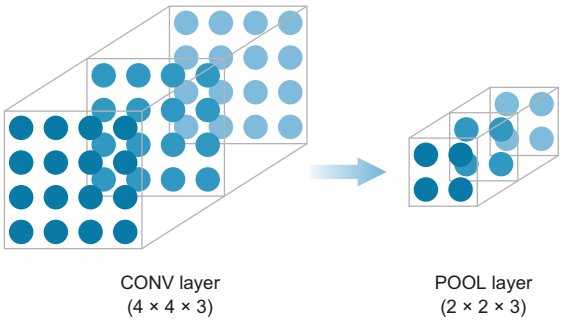


Figure 3.22 If the convolutional layer has three feature maps, the pooling layer's output will have three smaller feature maps.

values of all pixels in the feature map (figure 3.23). You can see in figure 3.24 that the global average pooling layer takes a 3D array and turns it into a vector.

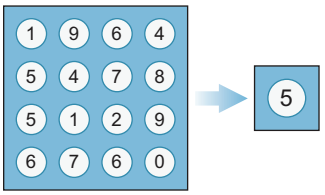


Figure 3.23 Global average pooling calculates the average values of all the pixels in a feature map.

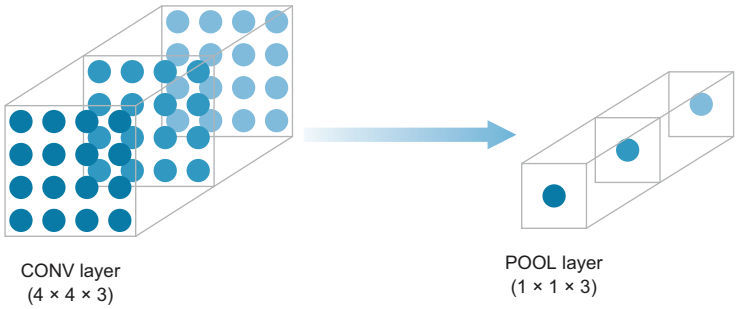


Figure 3.24 The global average pooling layer turns a 3D array into a vector.

WHY USE A POOLING LAYER?

As you can see from the examples we have discussed, pooling layers reduce the dimensionality of our convolutional layers. The reason it is important to reduce dimensionality is that in complex projects, CNNs contain many convolutional layers, and each has tens or hundreds of convolutional filters (kernels). Since the kernel contains the parameters (weights) that the network learns, this can get out of control very quickly, and the dimensionality of our convolutional layers can get very large. So adding pooling layers helps keep the important features and pass them along to the next layer, while shrinking image

dimensionality. Think of pooling layers as image-compressing programs. They reduce the image resolution while keeping its important features (figure 3.25).

Figure 3.25 Pooling layers reduce image resolution and keep the image's important features.

Pooling vs. strides and padding

The main purpose of pooling and strides is to reduce the number of parameters in the neural network. The more parameters we have, the more computationally expensive the training process will be. Many people dislike the pooling operation and think that we can get away without it in favor of tuning strides and padding the convolutional layer. For example, "Striving for Simplicity: The All Convolutional Net"[a] proposes discarding the pooling layer in favor of architecture that only consists of repeated convolutional layers. To reduce the size of the representation, the authors suggest occasionally using larger strides in the convolutional layer. Discarding pooling layers has also been found helpful in training good generative models, such as generative adversarial networks (GANs), which we will discuss in chapter 10. It seems likely that future architectures will feature very few to no pooling layers. But for now, pooling layers are still widely used to downsample images from one layer to the next.

[a] Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, and Martin Riedmiller, "Striving for Simplicity: The All Convolutional Net," https://arxiv.org/abs/1412.6806.

CONVOLUTIONAL AND POOLING LAYERS RECAP

Let's review what we have done so far. Up until this point, we used a series of convolutional and pooling layers to process an image and extract meaningful features that are specific to the images in the training dataset. To summarize how we got here:

1. The raw image is fed to the convolutional layer, which is a set of kernel filters that slide over the image to extract features.
2. The convolutional layer has the following attributes that we need to configure:

```
from keras.layers import Conv2D

model.add(Conv2D(filters=16, kernel_size=2, strides='1',
   padding='same', activation='relu'))
```

   - `filters` is the number of kernel filters in each layer (the depth of the hidden layer).
   - `kernel_size` is the size of the filter (aka kernel). Usually 2, or 3, or 5.
   - `strides` is the amount by which the filter slides over the image. A `strides` value of 1 or 2 is usually recommended as a good start.

- `padding` adds columns and rows of zero values around the border of the image to reserve the image size in the next layer.
- `activation` of `relu` is strongly recommended in the hidden layers.

3 The pooling layer has the following attributes that we need to configure:

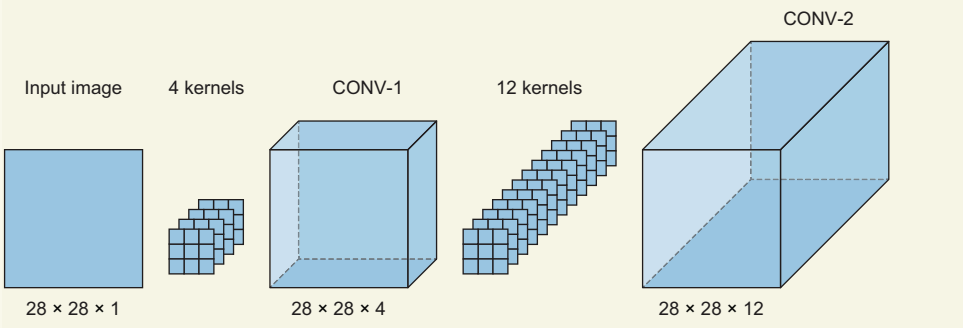
```
from keras.layers import MaxPooling2D

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

And we keep adding pairs of convolutional and pooling layers to achieve the required depth for our "deep" neural network.

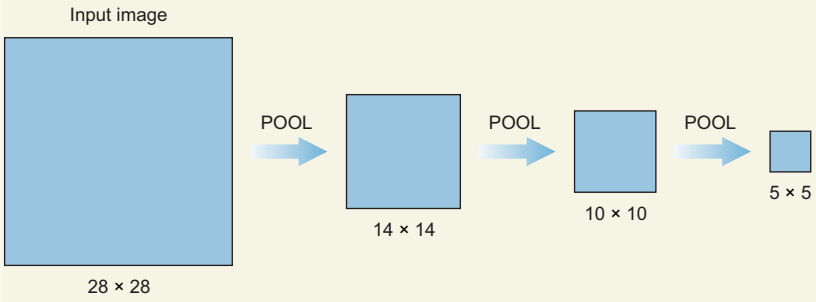
Visualize what happens after each layer

After the convolutional layers, the image keeps its width and height dimensions (usually), but it gets deeper and deeper after each layer. Why? Remember the cutting-the-image-into-pieces-of-features analogy we mentioned earlier? That is what's happening after the convolutional layer.

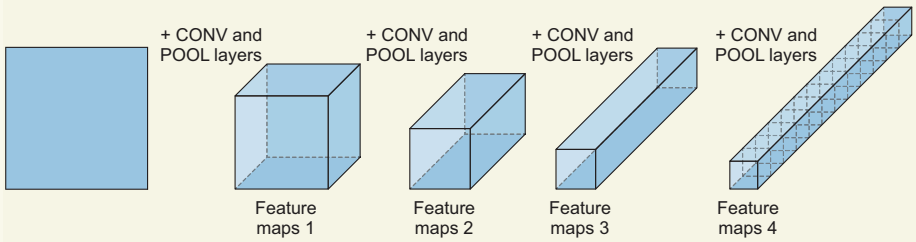
For example, suppose the input image is 28 × 28 (like in the MNIST dataset). When we add a CONV_1 layer (with `filters` of 4, `strides` of 1, and `padding` of `same`), the output will be the same width and height dimensions but with `depth` of 4 (28 × 28 × 4). Now we add a CONV_2 layer with the same hyperparameters but more filters (12), and we get deeper output: 28 × 28 × 12.



After the pooling layers, the image keeps its depth but shrinks in width and height:



Putting the convolutional and pooling together, we get something like this:



This keeps happening until we have, at the end, a long tube of small shaped images that contain all the features in the original image.

The output of the convolutional and pooling layers produces a feature tube ($5 \times 5 \times 40$) that is *almost* ready to be classified. We use 40 here as an example for the depth of the feature tube, as in 40 feature maps. The last step is to flatten this tube before feeding it to the fully connected layer for classification. As discussed earlier, the flattened layer will have the dimensions of $(1, m)$ where $m = 5 \times 5 \times 40 = 1{,}000$ neurons.

3.3.3 Fully connected layers

After passing the image through the feature-learning process using convolutional and pooling layers, we have extracted all the features and put them in a long tube. Now it is time to use these extracted features to classify images. We will use the regular neural network architecture, MLP, that we discussed in chapter 2.

WHY USE FULLY CONNECTED LAYERS?

MLPs work great in classification problems. The reason we used convolutional layers in this chapter is that MLPs lose a lot of valuable information when extracting features from an image—we have to flatten the image before feeding it to the network—whereas convolutional layers can process raw images. Now we have the features extracted, and after we flatten them, we can use regular MLPs to classify them.

We discussed the MLP architecture thoroughly in chapter 2: nothing new here. To reiterate, here are the fully connected layers (figure 3.26):

- *Input flattened vector*—As illustrated in figure 3.26, to feed the features tube to the MLP for classification, we flatten it to a vector with the dimensions $(1, n)$. For example, if the features tube has the dimensions of $5 \times 5 \times 40$, the flattened vector will be $(1, 1000)$.
- *Hidden layer*—We add one or more fully connected layers, and each layer has one or more neurons (similar to what we did when we built regular MLPs).
- *Output layer*—Chapter 2 recommended using the softmax activation function for classification problems involving more than two classes. In this example, we are classifying digits from 0 to 9: 10 classes. The number of neurons in the output layer is equal to the number of classes; thus, the output layer will have 10 nodes.

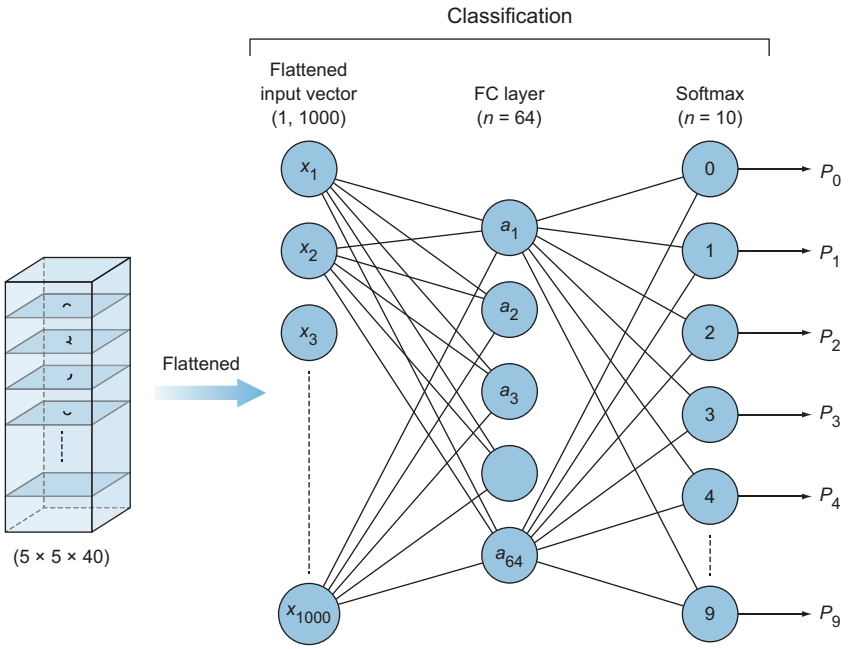
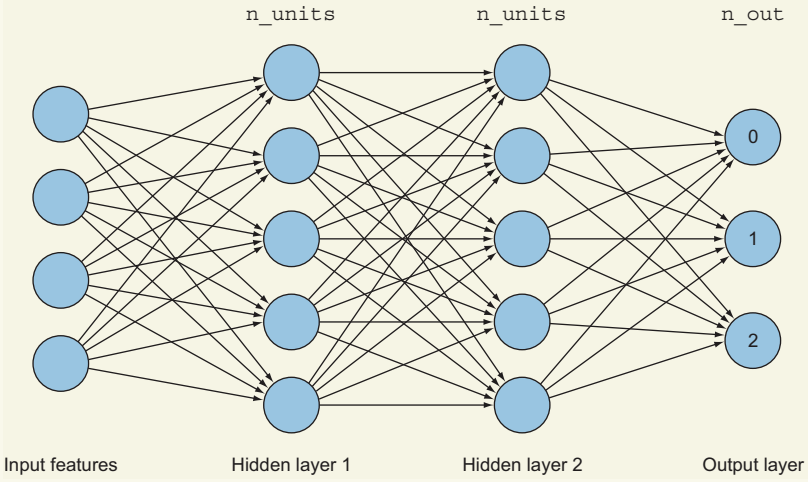


Figure 3.26 Fully connected layers for an MLP

MLPs and fully connected layers

Remember from chapter 2 that multilayer perceptrons (MLPs) are also called fully connected layers, because all the nodes from one layer are connected to all the nodes in the previous and next layers. They are also called *dense layers*. The terms *MLP*, *fully connected*, *dense*, and sometimes *feedforward* are used interchangeably to refer to the regular neural network architecture.



3.4 Image classification using CNNs

Okay, you are now fully equipped to build your own CNN model to classify images. For this mini project, which is a simple problem but which will help build the foundation to more complex problems in the following chapters, we will use the MNIST dataset. (The MNIST dataset is like "Hello World" for deep learning.)

> **NOTE** Regardless of which DL library you decide to use, the concepts are pretty much the same. You start with designing the CNN architecture in your mind or on a piece of paper, and then you begin stacking layers on top of each other and setting their parameters. Both Keras and MXNet (along with TensorFlow, PyTorch, and other DL libraries) have pros and cons that we will discuss later, but the concepts are similar. So for the rest of this book, we will be working mostly with Keras with a little overview of other libraries here and there.

3.4.1 Building the model architecture

This is the part in your project where you define and build the CNN model architecture. To look at the full code of the project that includes image preprocessing, training, and evaluating the model, go to the book's GitHub repo at https://github.com/moelgendy/deep_learning_for_vision_systems and open the mnist_cnn notebook or go to the book's website: www.manning.com/books/deep-learning-for-vision-systems or www.computerVisionBook.com. At this point, we are concerned with the code that builds the model architecture. At the end of this chapter, we will build an end-to-end image classifier and dive deeper into the other pieces:

```
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout

model = Sequential()

model.add(Conv2D(32, kernel_size=(3, 3), strides=1, padding='same',
          activation='relu', input_shape=(28,28,1)))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(64, (3, 3), strides=1, padding='same', activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Flatten())

model.add(Dense(64, activation='relu'))

model.add(Dense(10, activation='softmax'))

model.summary()
```

Builds the model object

CONV_1: adds a convolutional layer with ReLU activation and depth = 32 kernels

CONV_2: increases the depth to 64

POOL_1: downsamples the image to choose the best features

POOL_2: more downsampling

Flatten, since there are too many dimensions; we only want a classification output

FC_1: Fully connected to get all relevant data

FC_2: Outputs a softmax to squash the matrix into output probabilities for the 10 classes

Prints the model architecture summary

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 28, 28, 32)        320
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 14, 14, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 14, 14, 64)        18496
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 7, 7, 64)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 3136)              0
_________________________________________________________________
dense_1 (Dense)              (None, 64)                200768
_________________________________________________________________
dense_2 (Dense)              (None, 10)                650
=================================================================
Total params: 220,234
Trainable params: 220,234
Non-trainable params: 0
_________________________________________________________________
```

Figure 3.27 The printed model summary

When you run this code, you will see the model summary printed as in figure 3.27.

Following are some general observations before we look at the model summary:

- We need to pass the `input_shape` argument to the first convolutional layer only. Then we don't need to declare the input shape to the model, since the output of the previous layer is the input of the current layer—it is already known to the model.
- You can see that the output of every convolutional and pooling layer is a 3D tensor of shape (`None`, `height`, `width`, `channels`). The `height` and `width` values are pretty straightforward: they are the dimensions of the image at this layer. The `channels` value represents the depth of the layer. This is the number of feature maps in each layer. The first value in this tuple, set to `None`, is the number of images that are processed in this layer. Keras sets this to `None`, which means this dimension is variable and accepts any number of `batch_size`.
- As you can see in the Output Shape columns, as you go deeper through the network, the image dimensions shrink and the depth increases, as we discussed earlier in this chapter.
- Notice the number of total params (weights) that the network needs to optimize: 220,234, compared to the number of params from the MLP network we created earlier in this chapter (669,706). We were able to cut it down to almost a third.

Let's take a look at the model summary line by line:

- `CONV_1`—We know the input shape: (28 × 28 × 1). Look at the output shape of `conv2d`: (28 × 28 × 32). Since we set the `strides` parameter to `1` and `padding` to `same`, the dimensions of the input image did not change. But `depth` increased

to 32. Why? Because we added 32 filters in this layer. Each filter produces one feature map.

- `POOL_1`—The input of this layer is the output of its previous layer: (28 × 28 × 32). After the pooling layer, the image dimensions shrink, and `depth` stays the same. Since we used a 2 × 2 pool, the output shape is (14 × 14 × 32).
- `CONV_2`— Same as before, convolutional layers increase depth and keep dimensions. The input from the previous layer is (14 × 14 × 32). Since the filters in this layer are set to 64, the output is (14 × 14 × 64).
- `POOL_2`—Same 2 × 2 pool, keeping the depth and shrinking the dimensions. The output is (7 × 7 × 64).
- `Flatten`—Flattening a features tube that has dimensions of (7 × 7 × 64) converts it into a flat vector of dimensions (1, 3136).
- `Dense_1`—We set this fully connected layer to have 64 neurons, so the output is 64.
- `Dense_2`—This is the output layer that we set to 10 neurons, since we have 10 classes.

3.4.2 Number of parameters (weights)

Okay, now we know how to build the model and read the summary line by line to see how the image shape changes as it passes through the network layers. One important thing remains: the Param # column on the right in the model summary.

WHAT ARE THE PARAMETERS?

Parameters is just another name for weights. These are the things that your network learns. As we discussed in chapter 2, the network's goal is to update the weight values during the gradient descent and backpropagation processes until it finds the optimal parameter values that minimize the error function.

HOW ARE THESE PARAMETERS CALCULATED?

In MLP, we know that the layers are fully connected to each other, so the weight connections or edges are simply calculated by multiplying the number of neurons in each layer. In CNNs, weight calculations are not as straightforward. Fortunately, there is an equation for this:

number of params = filters × kernel size × depth of the previous layer + number of filters (for biases)

Let's apply this equation in an example. Suppose we want to calculate the parameters at the second layer of the previous mini project. Here is the code for `CONV_2` again:

```
model.add(Conv2D(64, (3, 3), strides=1, padding='same', activation='relu'))
```

Since we know that the depth of the previous layer is 32, then

⇒ Params = 64 × 3 × 3 × 32 + 64 = 18,496

Note that the pooling layers *do not* add any parameters. Hence, you will see the Param # value is 0 after the pooling layers in the model summary. The same is true for the flatten layer: no extra weights are added (figure 3.28).

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================

max_pooling2d_1  (MaxPooling2  (None, 14, 14, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 14, 14, 64)        18496
_________________________________________________________________
max_pooling2d_2  (MaxPooling2  (None, 7, 7, 64)         0
_________________________________________________________________
flatten_1 (Flatten)          (None, 3136)              0
```

Figure 3.28 Pooling and flatten layers don't add parameters, so Param # is 0 after pooling and flattening layers in the model summary.

When we add all the parameters in the Param # column, we get the total number of parameters that this network needs to optimize: 220,234.

TRAINABLE AND NON-TRAINABLE PARAMS

In the model summary, you will see the total number of params and, below it, the number of trainable and non-trainable params. The trainable params are the weights that this neural network needs to optimize during the training process. In this example, all our params are trainable (figure 3.29).

```
=================================================================
Total params: 220,234
Trainable params: 220,234
Non-trainable params: 0
_________________________________________________________________
```

Figure 3.29 All of our params are trainable and need to be optimized during training.

In later chapters, we will talk about using a pretrained network and combining it with your own network for faster and more accurate results: in such a case, you may decide to freeze some layers because they are pretrained. So, not all of the network params will be trained. This is useful for understanding the memory and space complexity of your model before starting the training process; but more on that later. As far as we know now, all our params are trainable.

3.5 Adding dropout layers to avoid overfitting

So far, you have been introduced to the three main layers of CNNs: convolution, pooling, and fully connected. You will find these three layer types in almost every CNN architecture. But that's not all of them—there are additional layers that you can add to avoid overfitting.

3.5.1 What is overfitting?

The main cause of poor performance in machine learning is either overfitting or underfitting the data. *Underfitting* is as the name implies: the model fails to fit the training data. This happens when the model is too simple to fit the data: for example, using one perceptron to classify a nonlinear dataset.

Overfitting, on the other hand, means fitting the data too much: memorizing the training data and not really learning the features. This happens when we build a super network that fits the training dataset perfectly (very low error while training) but fails to generalize to other data samples that it hasn't seen before. You will see that, in overfitting, the network performs very well in the training dataset but performs poorly in the test dataset (figure 3.30).

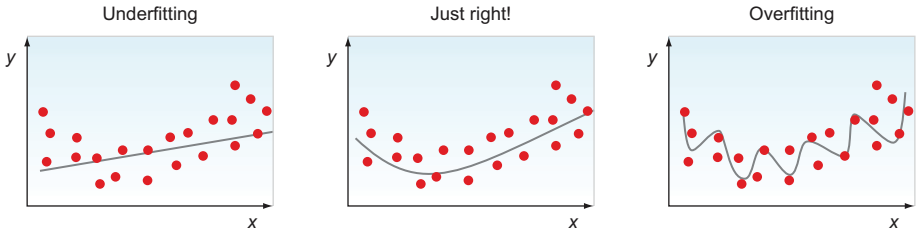


Figure 3.30 Underfitting (left): the model doesn't represent the data very well. Just right (middle): the model fits the data very well. Overfitting (right): the model fits the data too much, so it won't be able to generalize for unseen examples.

In machine learning, we don't want to build models that are too simple and so underfit the data or are too complex and overfit it. We want to use other techniques to build a neural network that is just right for our problem. To address that, we will discuss dropout layers next.

3.5.2 What is a dropout layer?

A dropout layer is one of the most commonly used layers to prevent overfitting. Dropout turns off a percentage of neurons (nodes) that make up a layer of your network (figure 3.31). This percentage is identified as a hyperparameter that you tune when you build your network. By "turns off," I mean these neurons are not included in a particular forward or backward pass. It may seem counterintuitive to throw away a connection in your network, but as a network trains, some nodes can dominate others or end up making large mistakes. Dropout gives you a way to balance your network so that every node works equally toward the same goal, and if one makes a mistake, it won't dominate the behavior of your model. You can think of dropout as a technique that makes a network resilient; it makes all the nodes work well as a team by making sure no node is too weak or too strong.

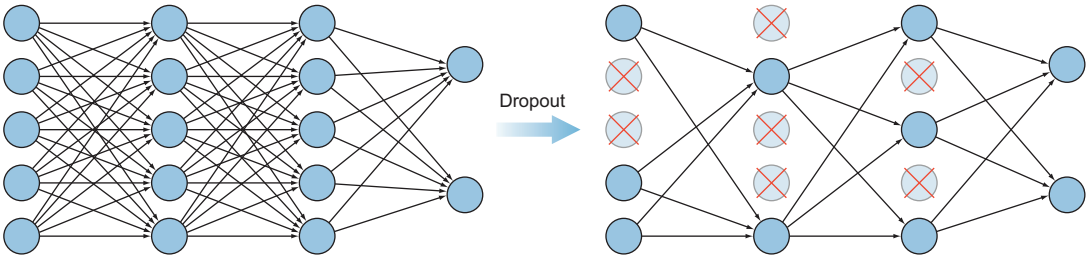


Figure 3.31 Dropout turns off a percentage of the neurons that make up a network layer.

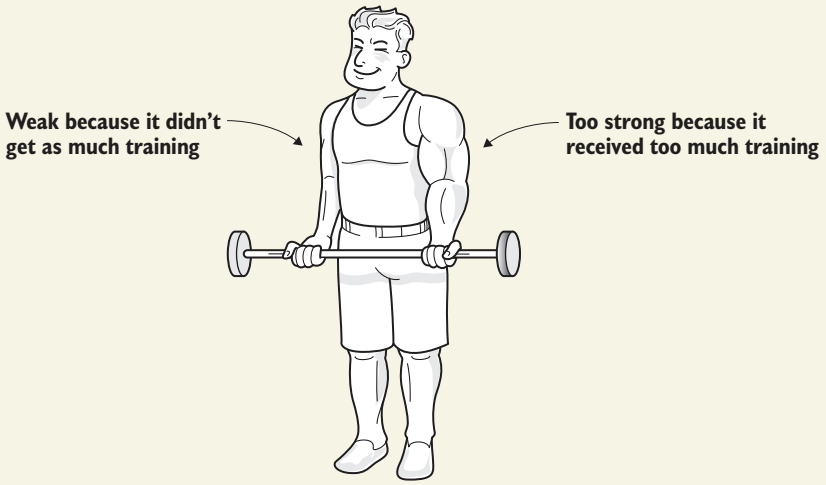
3.5.3 Why do we need dropout layers?

Neurons develop codependency among each other during training, which controls the individual power of each neuron, leading to overfitting of training data. To really understand why dropouts are effective, let's take a closer look at the MLP in figure 3.31 and think about what the nodes in each layer really represent. The first layer (far left) is the input layer that contains the input features. The second layer contains the features learned from the patterns of the previous layer when multiplied by the weights. Then the following layer is patterns learned within patterns, and so on. Each neuron represents a certain feature that, when multiplied by a weight, is transformed into another feature. When we randomly turn off some of these nodes, we force the other nodes to learn patterns without relying on only one or two features, because any feature can be randomly dropped out at any point. This results in spreading out the weights among all the features, leading to more trained neurons.

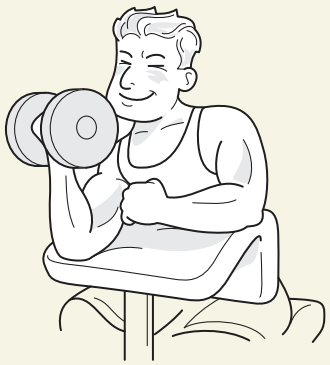
Dropout helps reduce interdependent learning among the neurons. In that sense, it helps to view dropout as a form of ensemble learning. In ensemble learning, we train a number of weaker classifiers separately, and then we use them at test time by averaging the responses of all ensemble members. Since each classifier has been trained separately, it has learned different aspects of the data, and their mistakes (errors) are different. Combining them helps to produce a stronger classifier, which is less prone to overfitting.

Intuition

An analogy that helps me understand dropout is training your biceps with a bar. When lifting a bar with both arms, we tend to rely on our stronger arm to lift a little more weight than our weaker arm. Our stronger arm will end up getting more training than the other and develop a larger muscle:



Dropout means mixing up our workout (training) a little. We tie our right arm and train our left arm only. Then we tie the left arm and train the right arm only. Then we mix it up and go back to the bar with both arms, and so on. After some time, you will see that you have developed both of your biceps:



This is exactly what happens when we train neural networks. Sometimes part of the network has very large weights and dominates all the training, while another part of the network doesn't get much training. What dropout does is turn off some neurons and let the rest of the neurons train. Then, in the next epoch, it turns off other neurons, and the process continues.

3.5.4 Where does the dropout layer go in the CNN architecture?

As you have learned in this chapter, a standard CNN consists of alternating convolutional and pooling layers, ending with fully connected layers. To prevent overfitting, it's become standard practice after you flatten the image to inject a few dropout layers

between the fully connected layers at the end of the architecture. Why? Because dropout is known to work well in the fully connected layers of convolutional neural nets. Its effect in convolutional and pooling layers is, however, not well studied yet:

CNN architecture: ... CONV ⇒ POOL ⇒ Flatten ⇒ DO ⇒ FC ⇒ DO ⇒ FC

Let's see how we use Keras to add a dropout layer to our previous model:

```
# CNN and POOL layers
# ...
# ...
model.add(Flatten())

model.add(Dropout(rate=0.3))

model.add(Dense(64, activation='relu'))

model.add(Dropout(rate=0.5))

model.add(Dense(10, activation='softmax'))

model.summary()
```

Flatten layer

Dropout layer with 30% probability

FC_1: fully connected to get all relevant data

Dropout layer with 50% probability

FC_2: outputs a softmax to squash the matrix into output probabilities for the 10 classes

Prints the model architecture summary

As you can see, the dropout layer takes `rate` as an argument. The rate represents the fraction of the input units to drop. For example, if we set `rate` to 0.3, it means 30% of the neurons in this layer will be randomly dropped in each epoch. So if we have 10 nodes in a layer, 3 of these neurons will be turned off, and 7 will be trained. The three neurons are randomly selected, and in the next epoch other randomly selected neurons are turned off, and so on. Since we do this randomly, some neurons may be turned off more than others, and some may never be turned off. This is okay, because we do this many times so that, on average, each neuron will get almost the same treatment. Note that this rate is another hyperparameter that we tune when building our CNN.

3.6 Convolution over color images (3D images)

Remember from chapter 1 that computers see grayscale images as 2D matrices of pixels (figure 3.32). To a computer, the image looks like a 2D matrix of the pixels' values, which represent intensities across the color spectrum. There is no context here, just a massive pile of data.

Color images, on the other hand, are interpreted by the computer as 3D matrices with height, width, and depth. In the case of RGB images (red, green, and blue) the depth is three: one channel for each color. For example, a color 28 × 28 image will be seen by the computer as a 28 × 28 × 3 matrix. Think of this as a stack of three 2D matrices—one each for the red, green, and blue channels of the image. Each of the

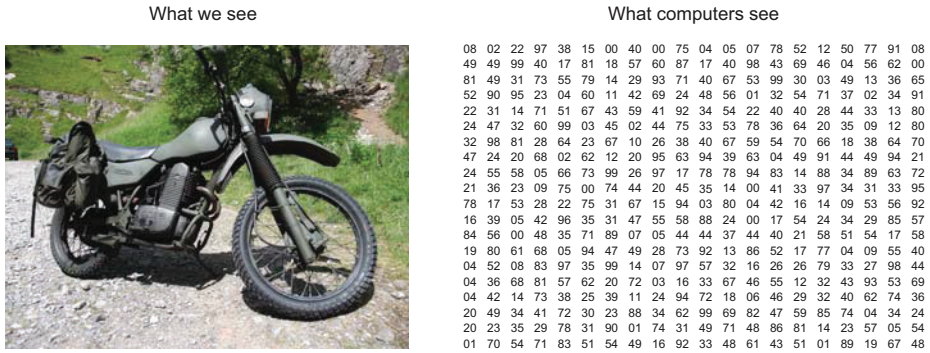


Figure 3.32 To a computer, an image looks like a 2D matrix of pixel values.

three matrices represents the value of intensity of its color. When they are stacked, they create a complete color image (figure 3.33).

NOTE For generalization, we represent images as a 3D array: height × width × depth. For grayscale images, depth is 1; and for color images, depth is 3.

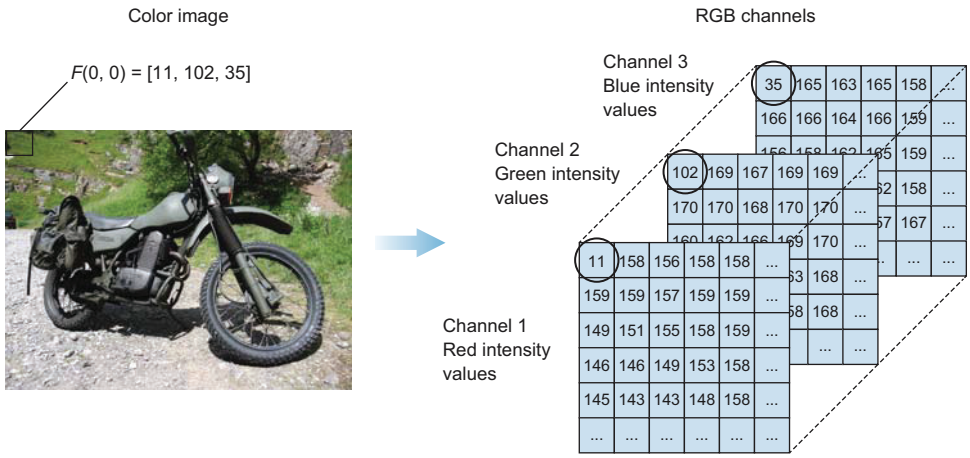


Figure 3.33 Color images are represented by three matrices. Each matrix represents the value of its color's intensity. Stacking them creates a complete color image.

3.6.1 How do we perform a convolution on a color image?

Similar to what we did with grayscale images, we slide the convolutional kernel over the image and compute the feature maps. Now the kernel is itself three-dimensional: one dimension for each color channel (figure 3.34).

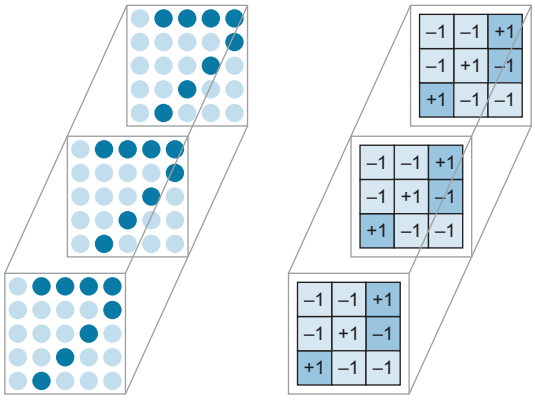


Figure 3.34 We slide the convolutional kernel over the image and compute the feature maps, resulting in a 3D kernel.

To perform convolution, we will do the same thing we did before, except that now, our sum is three times as many terms. Let's see how (figure 3.35):

- Each of the color channels has its own corresponding filter.
- Each filter will slide over its image, multiply every corresponding pixel element-wise, and then add them all together to compute the convolved pixel value of each filter. This is similar to what we did previously.
- We then add the three values to get the value of a single node in the convolved image or feature map. And don't forget to add the bias value of 1. Then we slide the filters over by one or more pixels (based on the strides value) and do the same thing. We continue this process until we compute the pixel values of all nodes in the feature map.

3.6.2 What happens to the computational complexity?

Note that if we pass a 3 × 3 filter over a grayscale image, we will have a total of 9 parameters (weights) for each filter (as already demonstrated). In color images, every filter is itself a 3D filter. This means every filter has a number of parameters: (height × width × depth) = (3 × 3 × 3) = 27. You can see how the network complexity increases when processing color images because it has to optimize more parameters; color images also take up more memory space.

Color images contain more information than grayscale images. This can add unnecessary computational complexity and take up memory space. However, color images are also really useful for certain classification tasks. That's why in some use cases, you, as a computer vision engineer, will use your judgement as to whether to convert your color images to grayscale where color doesn't really matter. This is because for many objects, color is not needed to recognize and interpret an image: grayscale could be enough to recognize objects.

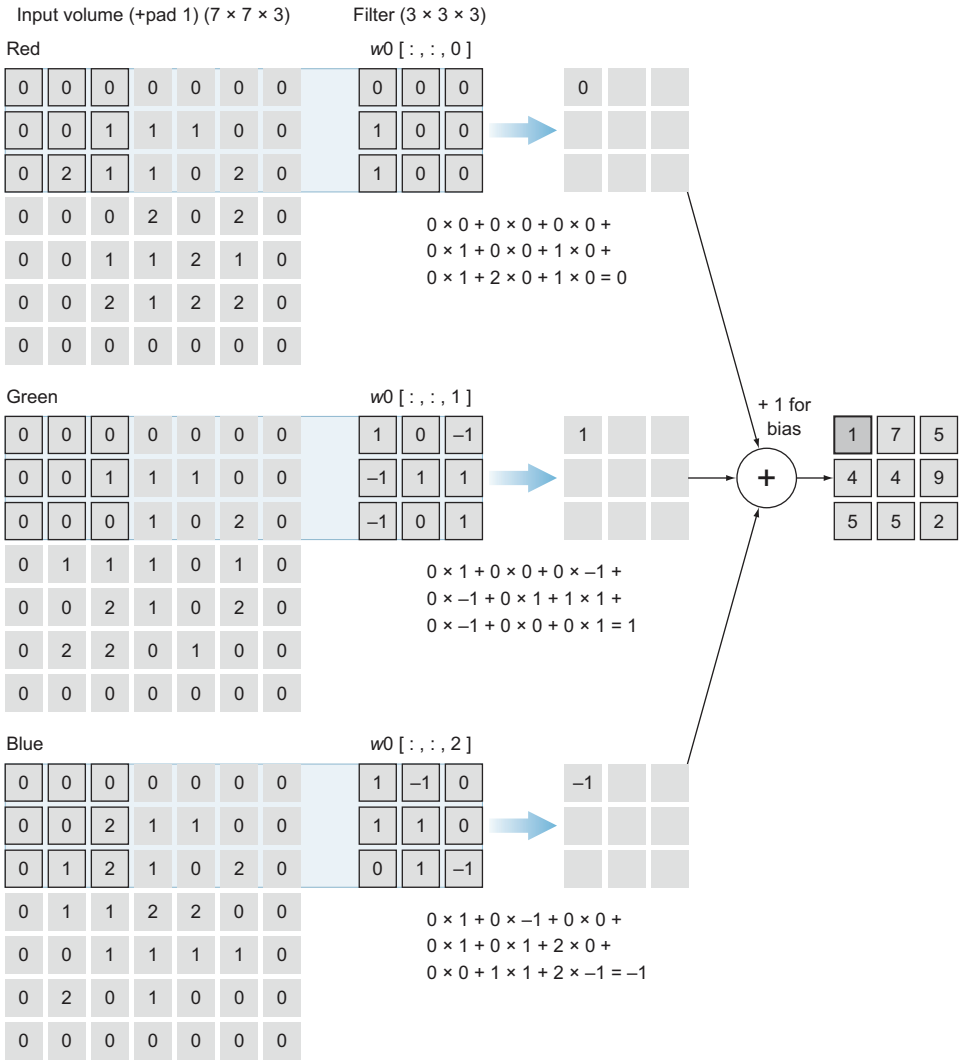


Figure 3.35 Performing convolution

In figure 3.36, you can see how patterns of light and dark in an object (intensity) can be used to define its shape and characteristics. However, in other applications, color is important to define certain objects: for example, skin cancer detection relies heavily on skin color (red rashes). In general, when it comes to CV applications like identifying cars, people, or skin cancer, you can decide whether color information is important or not by thinking about your own vision. If the identification problem is easier in color for us humans, it's likely easier for an algorithm to see color images, too.

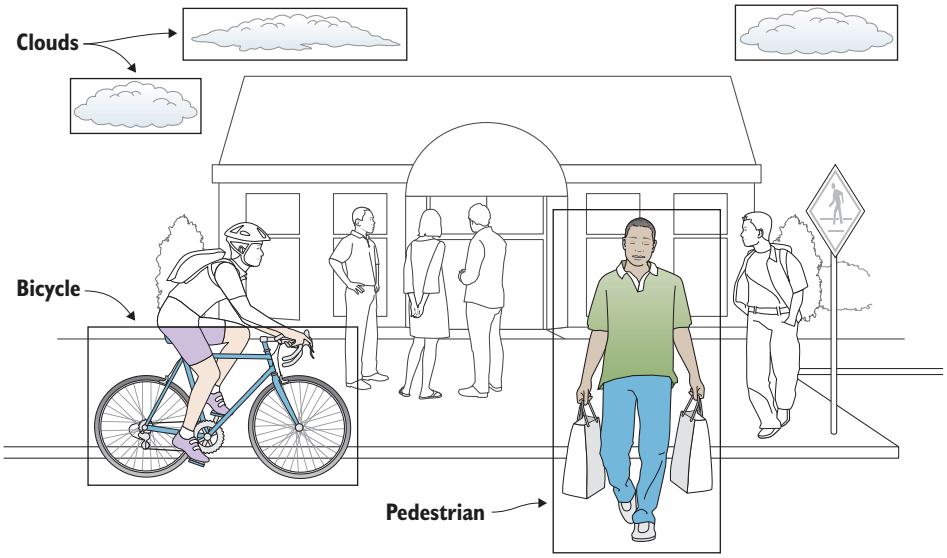


Figure 3.36 Patterns of light and dark in an object (intensity) can be used to define its shape and characteristics in a grayscale image.

Note that in figure 3.36, we added only one filter (that contains 3 channels), which produced one feature map. Similarly to grayscale images, each filter we add will produce its own feature map. In the CNN in figure 3.37, we have an input image of dimensions ($7 \times 7 \times 3$). We add two convolution filters of dimensions (3×3). The output feature map has a depth of 2, since we added two filters, similar to what we did with grayscale images.

An important closing note on CNN architecture

I strongly recommend looking at existing architectures, since many people have already done the work of throwing things together and seeing what works. Practically speaking, unless you are working on research problems, you should start with a CNN architecture that has already been built by other people to solve problems similar to yours. Then tune it further to fit your data.

In chapter 4, we will explain how to diagnose your network's performance and discuss tuning strategies to improve it. In chapter 5, we will discuss the most popular CNN architectures and examine how other researchers built them. What I want you to take from this section is, first, a conceptual understanding of how a CNN is built; and, second, that more layers lead to more neurons, which lead to more learning behavior. But this comes with computational cost. So you should always consider the size and complexity of your training data (many layers may not be necessary for a simple task).

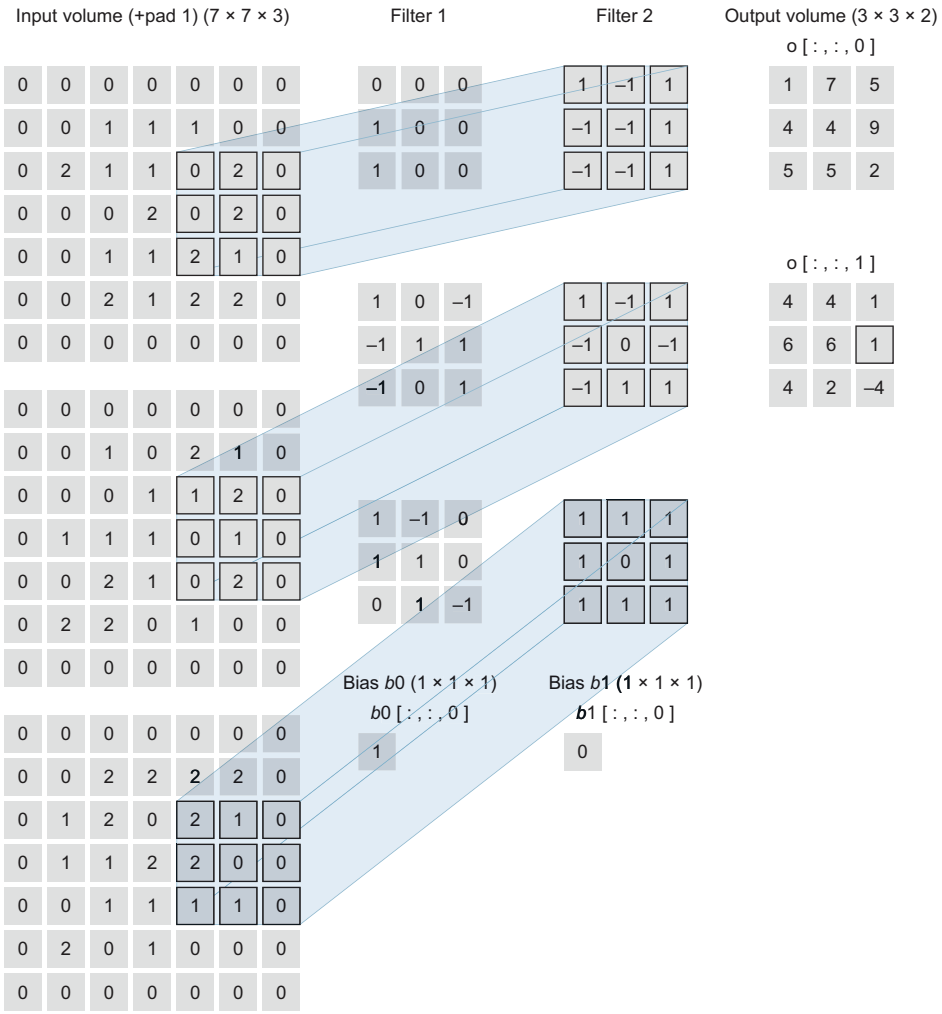


Figure 3.37 Our input image has dimensions (7 × 7 × 3), and we add two convolution filters of dimensions (3 × 3). The output feature map has a depth of 2.

3.7 Project: Image classification for color images

Let's take a look at an end-to-end image classification project. In this project, we will train a CNN to classify images from the CIFAR-10 dataset (www.cs.toronto.edu/~kriz/cifar.html). CIFAR-10 is an established CV dataset used for object recognition. It is a subset of the 80 Million Tiny Images dataset[1] and consists of 60,000 (32 × 32) color

[1] Antonio Torralba, Rob Fergus, and William T. Freeman, "80 Million Tiny Images: A Large Data Set for Non-parametric Object and Scene Recognition," *IEEE Transactions on Pattern Analysis and Machine Intelligence* (November 2008), https://doi.org/10.1109/TPAMI.2008.128.

images containing 1 of 10 object classes, with 6,000 images per class. Now, fire up your notebook and let's get started.

STEP 1: LOAD THE DATASET

The first step is to load the dataset into our train and test objects. Luckily, Keras provides the CIFAR dataset for us to load using the `load_data()` method. All we have to do is import `keras.datasets` and then load the data:

```
import keras
from keras.datasets import cifar10
(x_train, y_train), (x_test, y_test) = cifar10.load_data()

import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

fig = plt.figure(figsize=(20,5))
for i in range(36):
    ax = fig.add_subplot(3, 12, i + 1, xticks=[], yticks=[])
    ax.imshow(np.squeeze(x_train[i]))
```

Loads the preshuffled train and tests the data

STEP 2: IMAGE PREPROCESSING

Based on your dataset and the problem you are solving, you will need to do some data cleanup and preprocessing to get it ready for your learning model. A cost function has the shape of a bowl, but it can be an elongated bowl if the features have very different scales. Figure 3.38 shows gradient descent on a training set where features 1 and 2 have the same scale (on the left), and on a training set where feature 1 has much smaller values than feature 2 (on the right).

TIP When using gradient descent, you should ensure that all features have a similar scale; otherwise, it will take much longer to converge.

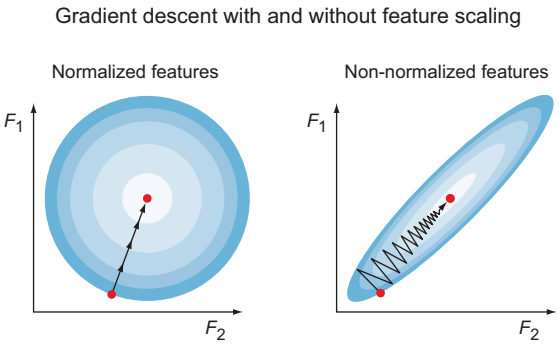


Figure 3.38 Normalized features are on the same scale represented by a uniform bowl (left). Non-normalized features are not on the same scale and are represented by an elongated bowl (right). Gradient descent on a training set with features that have the same scale (left) and on a training set where feature 1's values are much smaller than feature 2's (right).

Rescale the images

Rescale the input images as follows:

```
x_train = x_train.astype('float32')/255
x_test = x_test.astype('float32')/255
```

Rescales the images by dividing the pixel values by 255: [0,255] ⇒ [0,1]

Prepare the labels (one-hot encoding)

In this chapter and throughout the book, we will discuss how computers process input data (images) by converting it into numeric values in the form of matrices of pixel intensities. But what about the labels? How are the labels understood by computers? Every image in our dataset has a specific label that explains (in text) how this image is categorized. In this particular dataset, for example, the labels are categorized by the following 10 classes: `['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']`. We need to convert these text labels into a form that can be processed by computers. Computers are good with numbers, so we will do something called *one-hot encoding*. One-hot encoding is a process by which categorical variables are converted into a numeric form.

Suppose the dataset looks like the following:

Image	Label
image_1	dog
image_2	automobile
image_3	airplane
image_4	truck
image_5	bird

After one-hot encoding, we have the following:

	airplane	bird	cat	deer	dog	frog	horse	ship	truck	automobile
image_1	0	0	0	0	1	0	0	0	0	0
image_2	0	0	0	0	0	0	0	0	0	1
image_3	1	0	0	0	0	0	0	0	0	0
image_4	0	0	0	0	0	0	0	0	1	0
image_5	0	1	0	0	0	0	0	0	0	0

Luckily, Keras has a method that does just that for us:

```
from keras.utils import np_utils

num_classes = len(np.unique(y_train))
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)
```

One-hot encodes the labels

Split the dataset for training and validation

In addition to splitting our data into train and test datasets, it is a standard practice to further split the training data into training and validation datasets (figure 3.39). Why? Because each split is used for a different purpose:

- *Training dataset*—The sample of data used to train the model.
- *Validation dataset*—The sample of data used to provide an unbiased evaluation of model fit on the training dataset while tuning model hyperparameters. The evaluation becomes more biased as skill on the validation dataset is incorporated into the model configuration.
- *Test dataset*—The sample of data used to provide an unbiased evaluation of final model fit on the training dataset.

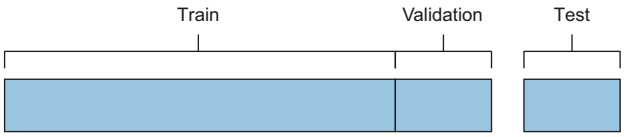


Figure 3.39 Splitting the data into training, validation, and test subsets

Here is the Keras code:

```
(x_train, x_valid) = x_train[5000:], x_train[:5000]
(y_train, y_valid) = y_train[5000:], y_train[:5000]
```

Breaks the training set into training and validation sets

```
print('x_train shape:', x_train.shape)
```

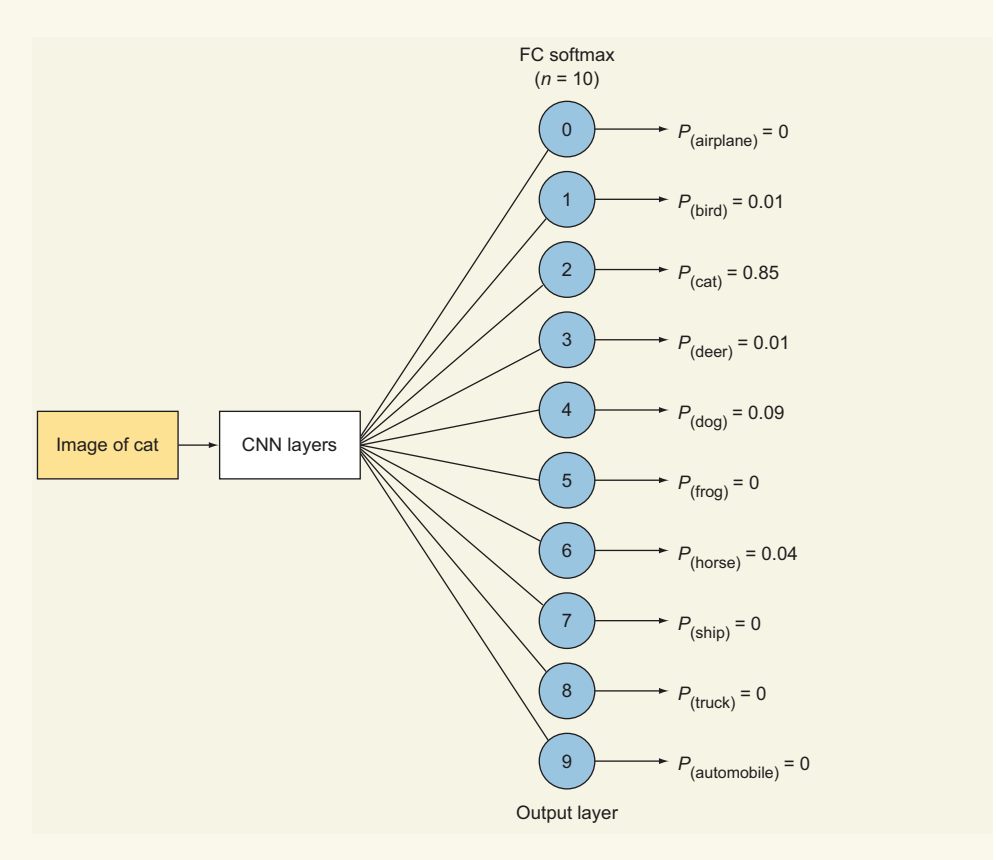
Prints the shape of the training set

```
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')
print(x_valid.shape[0], 'validation samples')
```

Prints the number of training, validation, and test images

The label matrix

One-hot encoding converts the (1 × *n*) label vector to a label matrix of dimensions (10 × *n*), where *n* is the number of sample images. So, if we have 1,000 images in our dataset, the label vector will have the dimensions (1 × 1000). After one-hot encoding, the label matrix dimensions will be (1000 × 10). That's why, when we define our network architecture in the next step, we will make the output softmax layer contain 10 nodes, where each node represents the probability of each class we have.



STEP 3: DEFINE THE MODEL ARCHITECTURE

You learned that the core building block of CNNs (and neural networks in general) is the layer. Most DL projects consist of stacking together simple layers that implement a form of *data distillation*. As you learned in this chapter, the main CNN layers are convolution, pooling, fully connected, and activation functions.

How do you decide on the network architecture?

How many convolutional layers should you create? How many pooling layers? In my opinion, it is very helpful to read about some of the most popular architectures (AlexNet, ResNet, Inception) and extract the key ideas leading to the design decisions. Looking at how these state-of-the-art architectures are built and playing with your own projects will help you build an intuition about the CNN architecture that most suits the problem you are solving. We will discuss the most popular CNN architectures in chapter 5. Until then, here is what you need to know:

- The more layers you add, the better (at least theoretically) your network will learn; but this will come at the cost of increasing the computational and memory

space complexity, because it increases the number of parameters to optimize. You will also face the risk of the network overfitting your training set.

- As the input image goes through the network layers, its depth increases, and the dimensions (width, height) shrink, layer by layer.
- In general, two or three layers of 3×3 convolutional layers followed by a 2×2 pooling can be a good start for smaller datasets. Add more convolutional and pooling layers until your image is a reasonable size (say, 4×4 or 5×5), and then add a couple of fully connected layers for classification.
- You need to set up several hyperparameters (like `filter`, `kernel_size`, and `padding`). Remember that you do not need to reinvent the wheel: instead, look in the literature to see what hyperparameters usually work for others. Choose an architecture that worked well for someone else as a starting point, and then tune these hyperparameters to fit your situation. The next chapter is dedicated to looking at what has worked well for others.

Learning to work with layers and hyperparameters

I don't want you to get hung up on setting hyperparameters when building your first CNNs. One of the best ways to gain an instinct for how to put layers and hyperparameters together is to actually see concrete examples of how others have done it. Most of your work as a DL engineer will involve building your architecture and tuning the parameters. The main takeaways from this chapter are these:

- Understand how the main CNN layers work (convolution, pooling, fully connected, dropout) and why they exist.
- Understand what each hyperparameter does (number of filters in the convolutional layer, kernel size, strides, and padding).
- Understand, in the end, how to implement any given architecture in Keras. If you are able to replicate this project on your own dataset, you are good to go.

In chapter 5, we will review several state-of-the-art architectures and see what worked for them.

The architecture shown in figure 3.40 is called AlexNet: it's a popular CNN architecture that won the ImageNet challenges in 2011 (more details on AlexNet in chapter 5). The AlexNet CNN architecture is composed of five convolutional and pooling layers, and three fully connected layers.

Let's try a smaller version of AlexNet and see how it performs with our dataset (figure 3.41). Based on the results, we might add more layers. Our architecture will stack three convolutional layers and two fully connected (dense) layers as follows:

CNN: INPUT $\Rightarrow$ CONV_1 $\Rightarrow$ POOL_1 $\Rightarrow$ CONV_2 $\Rightarrow$ POOL_2 $\Rightarrow$ CONV_3 $\Rightarrow$ POOL_3 $\Rightarrow$ DO $\Rightarrow$ FC $\Rightarrow$ DO $\Rightarrow$ FC (softmax)

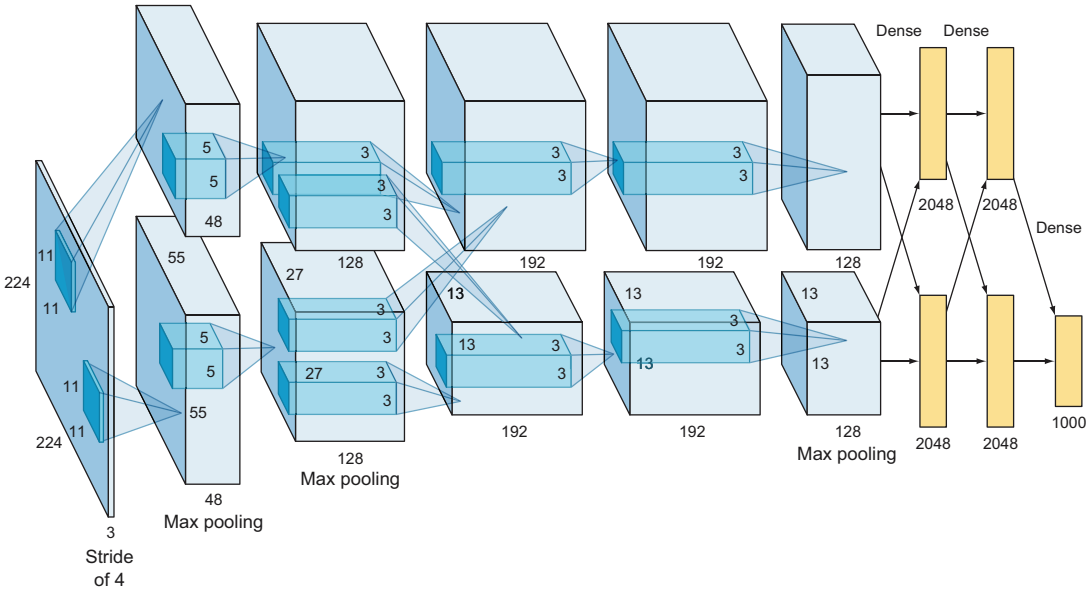


Figure 3.40 AlexNet architecture

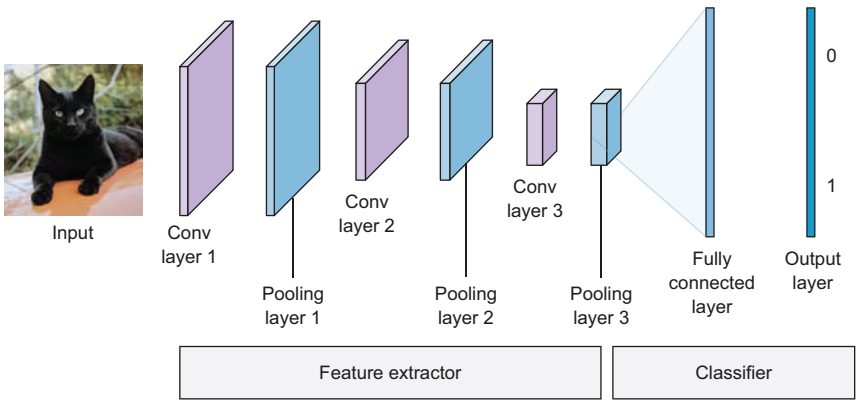


Figure 3.41 We will build a small CNN consisting of three convolutional layers and two dense layers.

Note that we will use the ReLU activation function for all the hidden layers. In the last dense layer, we will use a softmax activation function with 10 nodes to return an array of 10 probability scores (summing to 1). Each score will be the probability that the current image belongs to our 10 image classes:

```
from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout

model = Sequential()
model.add(Conv2D(filters=16, kernel_size=2, padding='same',
    activation='relu', input_shape=(32, 32, 3)))
model.add(MaxPooling2D(pool_size=2))

model.add(Conv2D(filters=32, kernel_size=2, padding='same',
    activation='relu'))
model.add(MaxPooling2D(pool_size=2))

model.add(Conv2D(filters=64, kernel_size=2, padding='same',
    activation='relu'))
model.add(MaxPooling2D(pool_size=2))

model.add(Dropout(0.3))

model.add(Flatten())

model.add(Dense(500, activation='relu'))
model.add(Dropout(0.4))

model.add(Dense(10, activation='softmax'))

model.summary()
```

First convolutional and pooling layers. Note that we need to define input_shape in the first convolutional layer only.

Second convolutional and pooling layers with a ReLU activation function

Third convolutional and pooling layers

Dropout layer to avoid overfitting with a 30% rate

Flattens the last feature map into a vector of features

Adds the first fully connected layer

Another dropout layer with a 40% rate

The output layer is a fully connected layer with 10 nodes and softmax activation to give probabilities to the 10 classes.

Prints a summary of the model architecture

When you run this cell, you will see the model architecture and how the dimensions of the feature maps change with every successive layer, as illustrated in figure 3.42.

We discussed previously how to understand this summary. As you can see, our model has 528,054 parameters (weights and biases) to train. We also discussed previously how this number was calculated.

STEP 4: COMPILE THE MODEL

The last step before training our model is to define three more hyperparameters—a loss function, an optimizer, and metrics to monitor during training and testing:

- *Loss function*—How the network will be able to measure its performance on the training data.
- *Optimizer*—The mechanism that the network will use to optimize its parameters (weights and biases) to yield the minimum loss value. It is usually one of the variants of stochastic gradient descent, explained in chapter 2.
- *Metrics*—List of metrics to be evaluated by the model during training and testing. Typically we use `metrics=['accuracy']`.

Feel free to revisit chapter 2 for more details on the exact purpose and different types of loss functions and optimizers.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 32, 32, 16)        208
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 16, 16, 16)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 16, 16, 32)        2080
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 8, 8, 32)          0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 8, 8, 64)          8256
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 4, 4, 64)          0
_________________________________________________________________
dropout_1 (Dropout)          (None, 4, 4, 64)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 1024)              0
_________________________________________________________________
dense_1 (Dense)              (None, 500)               512500
_________________________________________________________________
dropout_2 (Dropout)          (None, 500)               0
_________________________________________________________________
dense_2 (Dense)              (None, 10)                5010
=================================================================
Total params: 528,054
Trainable params: 528,054
Non-trainable params: 0
_________________________________________________________________
```

Figure 3.42 Model summary

Here is the code to compile the model:

```
model.compile(loss='categorical_crossentropy', optimizer='rmsprop',
    metrics=['accuracy'])
```

STEP 5: TRAIN THE MODEL

We are now ready to train the network. In Keras, this is done via a call to the network's .fit() method (as in fitting the model to the training data):

```
from keras.callbacks import ModelCheckpoint

checkpointer = ModelCheckpoint(filepath='model.weights.best.hdf5', verbose=1,
    save_best_only=True)

hist = model.fit(x_train, y_train, batch_size=32, epochs=100,
    validation_data=(x_valid, y_valid), callbacks=[checkpointer],
    verbose=2, shuffle=True)
```

When you run this cell, the training will start, and the verbose output shown in figure 3.43 will show one epoch at a time. Since 100 epochs of display do not fit on one page, the screenshot shows the first 13 epochs. But when you run this on your notebook, the display will keep going for 100 epochs.

```
Train on 45000 amples, validation 5000 samples
Epoch 1/100
Epoch 00000: val_loss improved from inf to 1.35820, saving model to model.weights.best.hdf5
46s - loss: 1.6192 - acc: 0.4140 - val_loss: 1.3582 - val_acc: 0.5166
Epoch 2/100
Epoch 00001: val_loss improved from 1.35820 to 1.22245, saving model to model.weights.best.hdf5
53s - loss: 1.2881 - acc: 0.5402 - val_loss: 1.2224 - val_acc: 0.5644
Epoch 3/100
Epoch 00002: val_loss improved from 1.22245 to 1.12096, saving model to model.weights.best.hdf5
49s - loss: 1.1630 - acc: 0.5879 - val_loss: 1.1210 - val_acc: 0.6046
Epoch 4/100
Epoch 00003: val_loss improved from 1.12096 to 1.10724, saving model to model.weights.best.hdf5
56s - loss: 1.0928 - acc: 0.6160 - val_loss: 1.1072 - val_acc: 0.6134
Epoch 5/100
Epoch 00004: val_loss improved from 1.10724 to 0.97377, saving model to model.weights.best.hdf5
52s - loss: 1.0413 - acc: 0.6382 - val_loss: 0.9738 - val_acc: 0.6596
Epoch 6/100
Epoch 00005: val_loss improved from 0.97377 to 0.95501, saving model to model.weights.best.hdf5
50s - loss: 1.0090 - acc: 0.6484 - val_loss: 0.9550 - val_acc: 0.6768
Epoch 7/100
Epoch 00006: val_loss improved from 0.95501 to 0.94448, saving model to model.weights.best.hdf5
49s - loss: 0.9967 - acc: 0.6561 - val_loss: 0.9445 - val_acc: 0.6828
Epoch 8/100
Epoch 00007: val_loss did not improve
61s - loss: 0.9934 - acc: 0.6604 - val_loss: 1.1300 - val_acc: 0.6376
Epoch 9/100
Epoch 00008: val_loss improved from 0.94448 to 0.91779, saving model to model.weights.best.hdf5
49s - loss: 0.9858 - acc: 0.6672 - val_loss: 0.9178 - val_acc: 0.6882
Epoch 10/100
Epoch 00009: val_loss did not improve
50s - loss: 0.9839 - acc: 0.6658 - val_loss: 0.9669 - val_acc: 0.6748
Epoch 11/100
Epoch 00010: val_loss improved from 0.91779 to 0.91570, saving model to model.weights.best.hdf5
49s - loss: 1.0002 - acc: 0.6624 - val_loss: 0.9157 - val_acc: 0.6936
Epoch 12/100
Epoch 00011: val_loss did not improve
54s - loss: 1.0001 - acc: 0.6659 - val_loss: 1.1442 - val_acc: 0.6646
Epoch 13/100
Epoch 00012: val_loss did not improve
56s - loss: 1.0161 - acc: 0.6633 - val_loss: 0.9702 - val_acc: 0.6788
```

Figure 3.43 The first 13 epochs of training

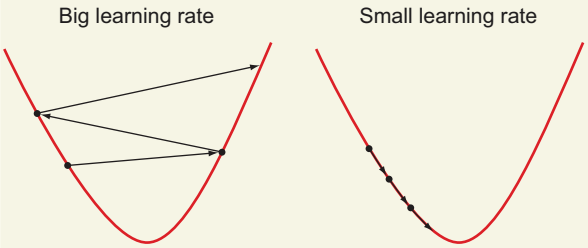
Looking at the verbose output in figure 3.43 will help you analyze how your network is performing and suggest which knobs (hyperparameter) to tune. We will discuss this in detail in chapter 4. For now, let's look at the most important takeaways:

- `loss` and `acc` are the error and accuracy values for the training data. `val_loss` and `val_acc` are the error and accuracy values for the validation data.
- Look at the `val_loss` and `val_acc` values after each epoch. Ideally, we want `val_loss` to be decreasing and `val_acc` to be increasing, indicating that the network is actually learning after each epoch.
- From epochs 1 through 6, you can see that the model is saving the weights after each epoch, because the validation loss value is improving. So at the end of each epoch, we save the weights that are considered the best weights so far.

- At epoch 7, `val_loss` went up to 1.1300 from 0.9445, which means that it did not improve. So the network did not save the weights at this epoch. If you stop the training now and load the weights from epoch 6, you will get the best results that you achieved during the training.
- The same is true for epoch 8: `val_loss` decreases, so the network saves the weights as best values. And at epoch 9, there is no improvement, and so forth.
- If you stop your training after 12 epochs and load the best weights, the network will load the weights saved after epoch 10 at (`val_loss` = 0.9157) and (`val_acc` = 0.6936). This means you can expect to get accuracy on the test data close to 69%.

Keep your eye on these common phenomena

- `val_loss` *is oscillating.* If `val_loss` is oscillating up and down, you might want to decrease the learning-rate hyperparameter. For example, if you see `val_loss` going from 0.8 to 0.9, to 0.7, to 1.0, and so on, this might mean that your learning rate is too high to descend the error mountain. Try decreasing the learning rate and letting the network train for a longer time.



If `val_loss` oscillates, the learning rate may be too high.

- `val_loss` *is not improving (underfitting).* If `val_loss` is not decreasing, this might mean your model is too simple to fit the data (underfitting). Then you may want to build a more complex model by adding more hidden layers to help the network fit the data.
- `loss` *is decreasing and* `val_loss` *stopped improving*. This means your network started to overfit the training data and failed to decrease the error for the validation data. In this case, consider using a technique to prevent overfitting, like dropout layers. There are other techniques to avoid overfitting, as we will discuss in the next chapter.

STEP 6: LOAD THE MODEL WITH THE BEST VAL_ACC

Now that the training is complete, we use the Keras method `load_weights()` to load into our model the weights that yielded the best validation accuracy score:

```
model.load_weights('model.weights.best.hdf5')
```

STEP 7: EVALUATE THE MODEL

The last step is to evaluate our model and calculate the accuracy value as a percentage indicating how often our model correctly predicts the image classification:

```
score = model.evaluate(x_test, y_test, verbose=0)
print('\n', 'Test accuracy:', score[1])
```

When you run this cell, you will get an accuracy of about 70%. That is not bad. But we can do a lot better. Try playing with the CNN architecture by adding more convolutional and pooling layers, and see if you can improve your model.

In the next chapter, we will discuss strategies to set up your DL project and hyperparameter tuning to improve the model's performance. At the end of chapter 4, we will revisit this project to apply these strategies and improve the accuracy to above 90%.

Summary

- MLPs, ANNs, dense, and feedforward all refer to the regular fully connected neural network architecture that we discussed in chapter 2.
- MLPs usually work well for 1D inputs, but they perform poorly with images for two main reasons. First, they only accept feature inputs in a vector form with dimensions $(1 \times n)$. This requires flattening the image, which will lead to losing its spatial information. Second, MLPs are composed of fully connected layers that will yield millions and billions of parameters when processing bigger images. This will increase the computational complexity and will not scale for many image problems.
- CNNs really shine in image processing because they take the raw image matrix as an input without having to flatten the image. They are composed of locally connected layers called convolution filters, as opposed to the MLPs' dense layers.
- CNNs are composed of three main layers: the convolutional layer for feature extraction, the pooling layer to reduce network dimensionality, and the fully connected layer for classification.
- The main cause of poor prediction performance in machine learning is either overfitting or underfitting the data. Underfitting means that the model is too simple and fails to fit (learn) the training data. Overfitting means that the model is so complex that it memorizes the training data and fails to generalize for test data that it hasn't seen before.
- A dropout layer is added to prevent overfitting. Dropout turns off a percentage of neurons (nodes) that make up a layer of our network.