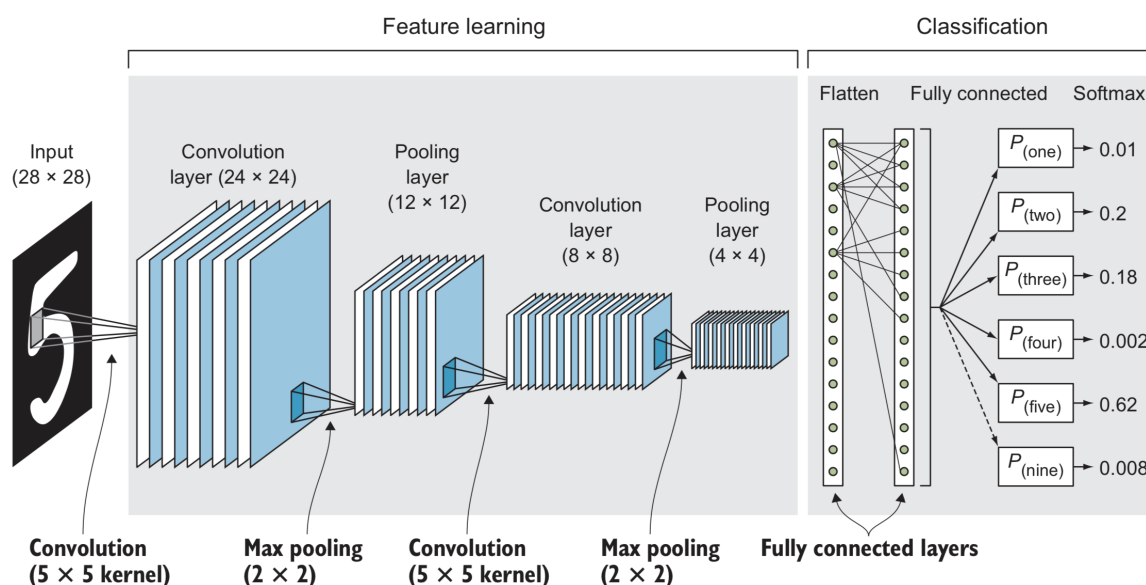


## Basic Components of Convolutional Networks

There are **three** main types of layers that you will see in almost every Convolutional Network (CNN):

1. Convolutional layer
2. Pooling layer
3. Fully connected layer

As we have seen, convolutional layers and pooling layers are to perform feature extraction, and fully connected layers for classification. These three basic components are illustrated below:



We will go through the details of these components in sequel.

### 1. Convolutional Layer

A convolutional layer is the **core** building block of a CNN. 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?

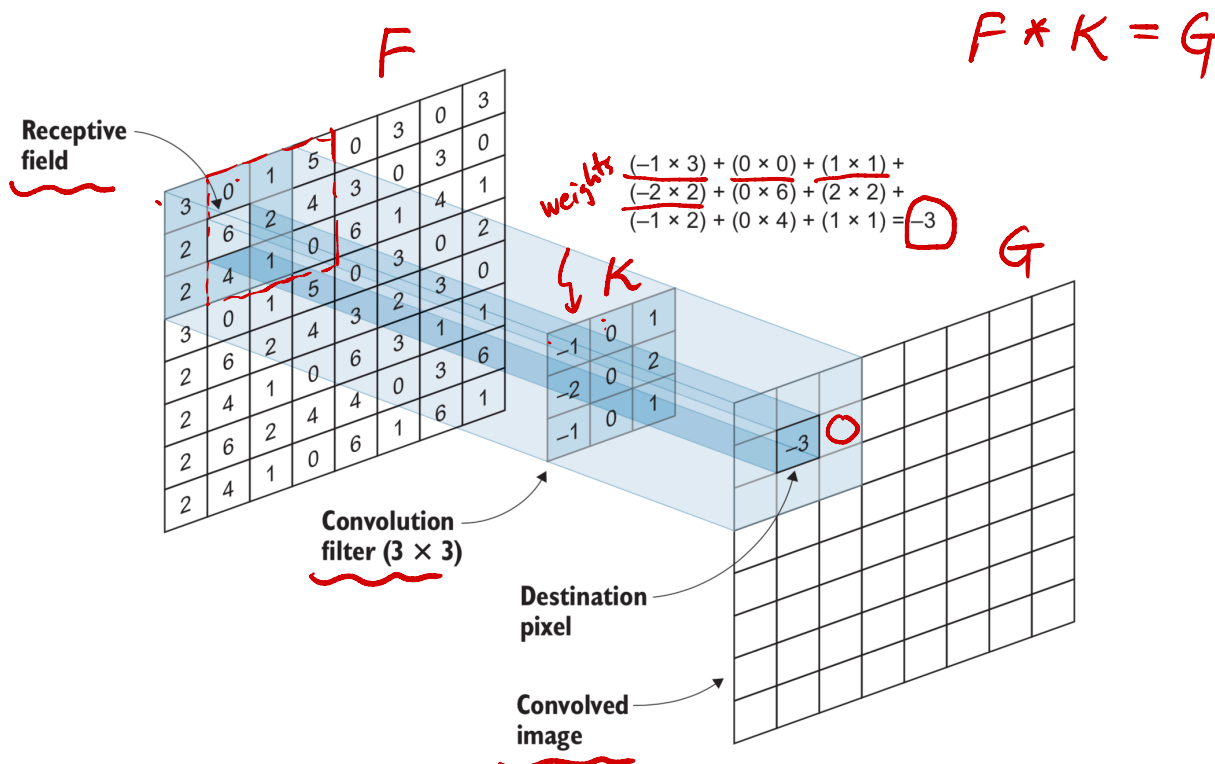
image  $\sim$  kernel  $\sim$  output image

$$F * K = G$$

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 a convolutional layer to see how it processes an image. 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**.

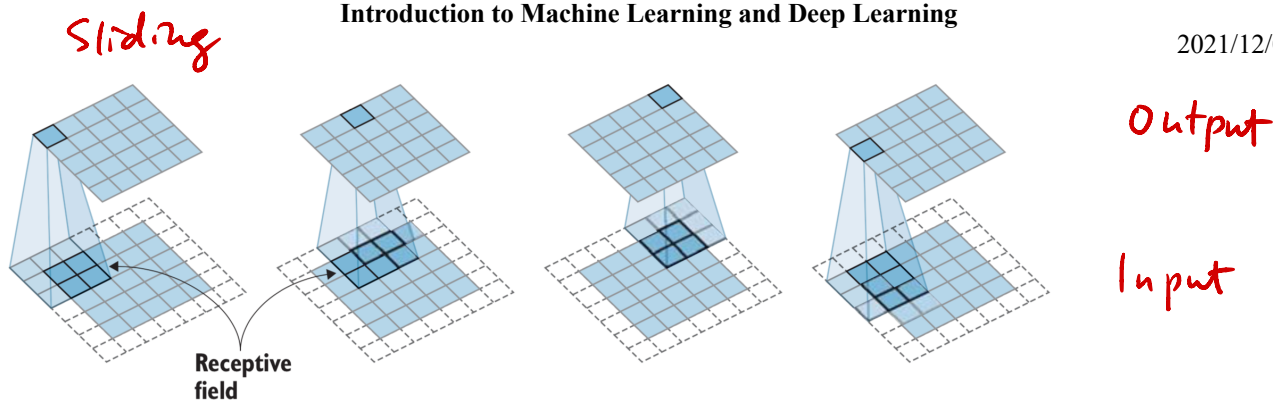


**Remark on definition of feature map:** In CNNs, a feature map is the output of one filter applied to the previous layer. It is called a feature map because it is a mapping of where a certain kind of feature is found in the image. CNNs look for features such as straight lines, edges, or even objects. 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.

Keeping the above figure in mind, here are some facts about convolution filters:

- The small  $3 \times 3$  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 below).



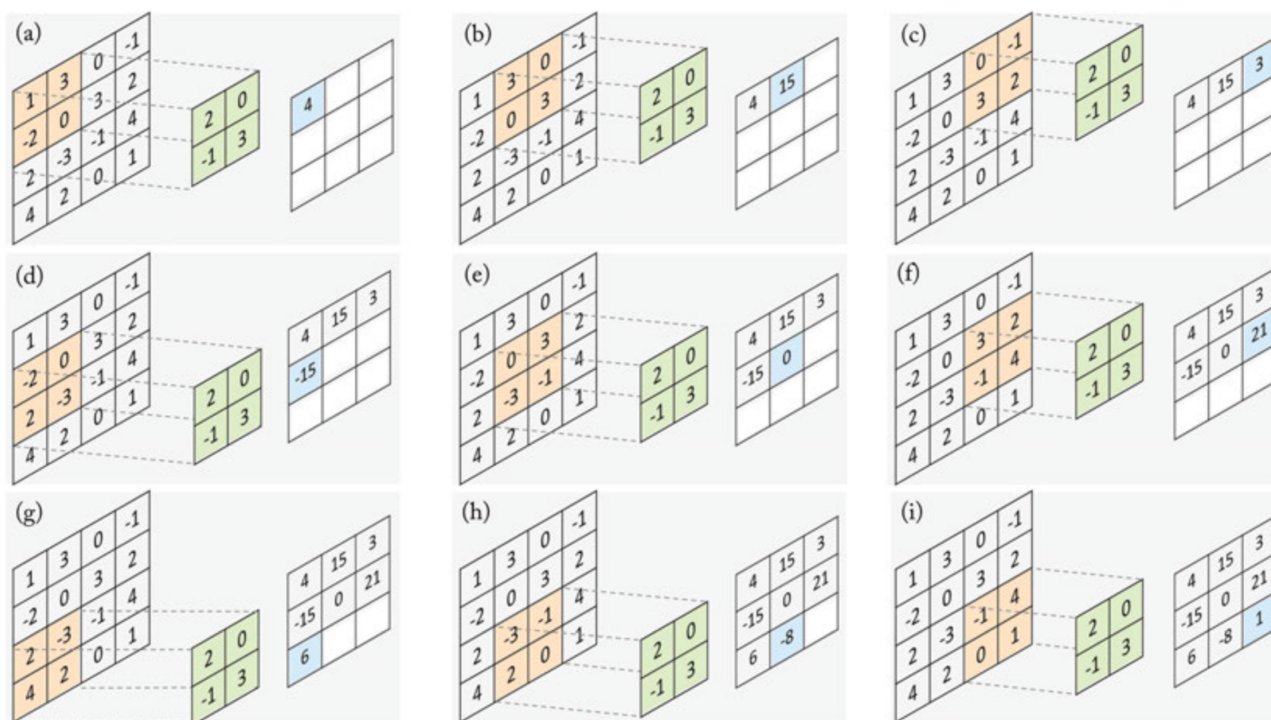
**Remark:** What are the values in the convolutional filter? In CNNs, they are the **weights**. This means they are also randomly initialized and the values are learned by the network (so you will not have to worry about assigning its values).

### Convolutional Operations

*Kernel  
filter  
convolution filter*

In CNNs, the neurons and weights are structured in a matrix shape. 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.

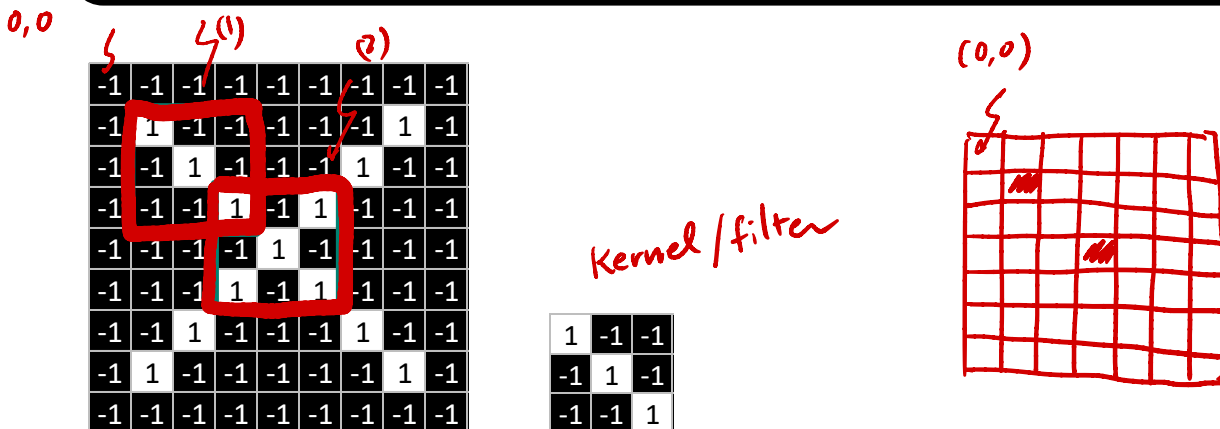
The filter (or kernel) slides over the whole image. Each time, we multiply every corresponding pixel element-wise and then add them all together to create a new image with new pixel values as illustrated below. Again, this convolved image is called a **feature map**.



#073374

**Fun Time:** consider the following 9x9 input image and 3x3 filter, what are the **values** in the 7x7 output feature map for the two highlighted patches? (1) ~~1 and 1~~ (2) ~~1 and 50~~ (3) ~~55 and 1~~ (4) ~~1 and 55~~ **9 and 5**

**Fun Time:** consider the following 9x9 input image and 3x3 filter, what are the **positions** in the 7x7 output feature map for the two highlighted patches? (1) (0, 0) and (3, 3) (2) (1, 1) and (4, 4) (3) (1, 1) and (3, 3).



### Applying Filters to Learn Features

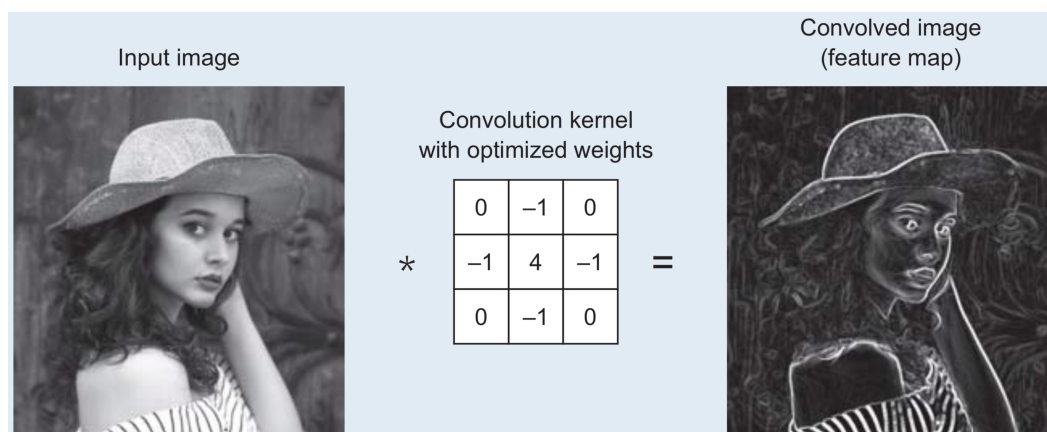
We are doing all convolution operation so the network extracts features from the image. **How does applying filters lead toward this goal?**

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. Let us 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  $G(x, y)$  (a feature map) that **amplifies the edges**.





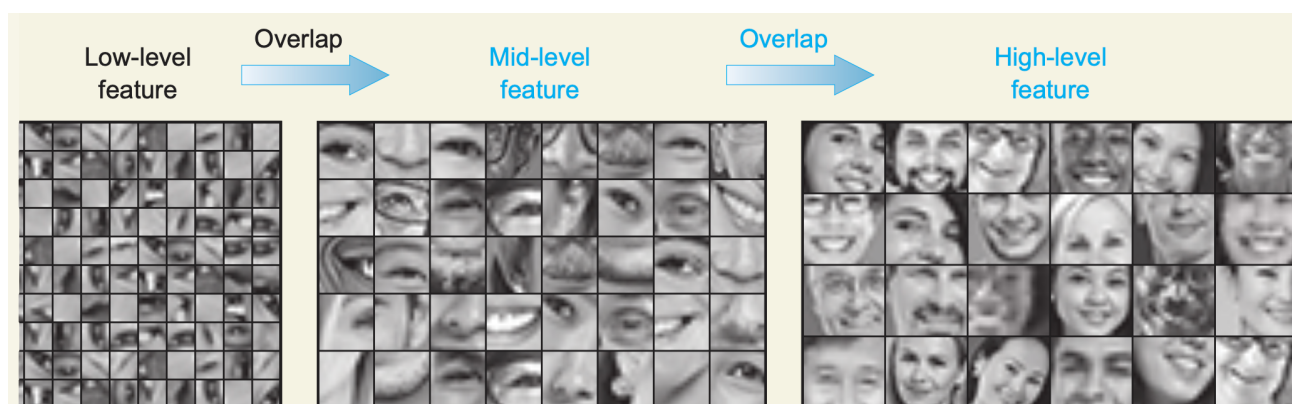
Mathematically, it read:

$$F(x, y) \star K = G(x, y)$$

### Remarks:

1. 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.
2. 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. For example, below is a simplified version of how CNNs learn faces.

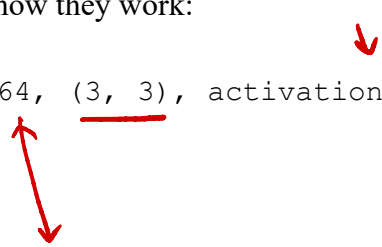
You can see that the early layers detect patterns in the image to learn low-level features like edges, and the later layers detect patterns within patterns to learn more complex features like parts of the face, then patterns within patterns within patterns, and so on:



### Conv2D in Keras

Let's take a look at the convolutional layer as a whole: each convolutional layer contains **some 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 a typical convolutional layer in Keras to see how they work:

```
| model.add(layers.Conv2D(64, (3, 3), activation='relu'))
```



Or equivalently,

```
| model.add(layers.Conv2D(filters=64, kernel_size=3, strides=1,  
padding= 'valid', activation='relu'))
```

<b>filters</b>	Integer, the dimensionality of the output space (i.e. the number of output filters in the convolution).
<b>kernel_size</b>	An integer or tuple/list of 2 integers, specifying the height and width of the 2D convolution window. Can be a single integer to specify the same value for all spatial dimensions.
<b>strides</b>	An integer or tuple/list of 2 integers, specifying the strides of the convolution along the height and width. Can be a single integer to specify the same value for all spatial dimensions. Specifying any stride value != 1 is incompatible with specifying any <b>dilation_rate</b> value != 1.
<b>padding</b>	one of "valid" or "same" (case-insensitive).

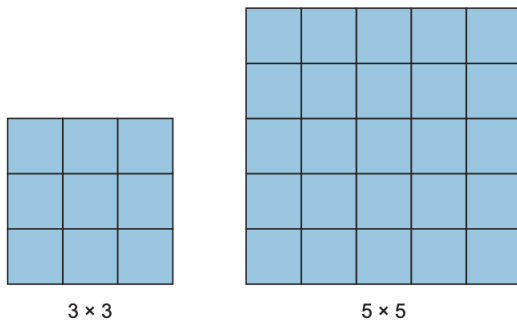
Let us take a look of each hyperparameter.

### Number of Filters in the Convolutional Layer

Each convolutional layer has some filters. Similar to MLPs, the convolutional layers in CNNs are the hidden layers. By increasing the number of kernels in a convolutional layer, we increase the number of neurons, which makes our network more complex and able to detect more complex patterns.

### 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).



**Remark:** `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.

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

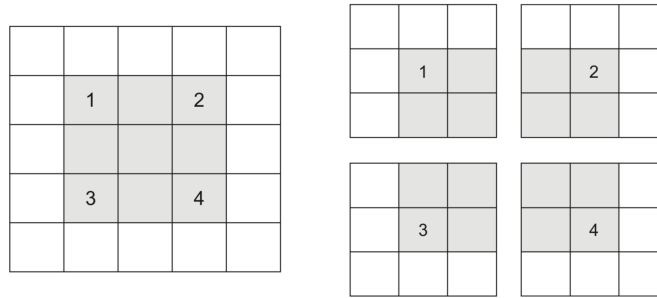
### Strides and Padding

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 (default). 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.

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. We say “roughly” because it depends on what you set the padding parameter to do with the edge of the image.

In figure below, you can see the patches extracted by a  $3 \times 3$  convolution filter with stride 2 over a  $5 \times 5$  input (without padding).



**Fun Time:** what is the shape of the output of a convolutional layer for this case? (1) 5x5x1 (2) 4x4x1 (3) 3x3x1 (4) 2x2x1.

**Padding**—Often called zero-padding because we add zeros around the border of an image (see figure below). 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.

Padding = 2      Pad

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	123	94	2	4	0	0
0	0	11	3	22	192	0	0
0	0	12	4	23	34	0	0
0	0	194	83	12	94	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Pad

**padding**      one of "valid" or "same" (case-insensitive).

In `Keras`, you can apply 'valid' or 'same' options, 'valid' means no padding and 'same' means `Keras` will maintain the output shape as its input if you use a stride of 1.

The output width and height ( $w' \times h'$ ) are determined by the input width and height ( $w \times h$ ), the filter width and height ( $f \times f$ ), the padding ( $p$ ) and the stride ( $s$ ).

For the 'valid' padding:

$$w' = \text{ceil}\left(\frac{w - f + 1}{s}\right)$$

$$h' = \text{ceil}\left(\frac{h - f + 1}{s}\right)$$

For the 'same' padding:

$$w' = \text{ceil}\left(\frac{w}{s}\right)$$

$$h' = \text{ceil}\left(\frac{h}{s}\right)$$

with the default values  $p = \text{'valid'}$  (no padding) and  $s = 1$  (continuous).

**Example:** what is the shape of the output of a convolutional layer after we apply the following command?

```
from tensorflow.keras import models
from tensorflow.keras.layers import Conv2D
model = models.Sequential()
model.add(Conv2D(64, 5, activation='relu', input_shape=(28, 28, 3)))
model.summary()
```

**A:**

$$w' = \text{ceil}\left(\frac{w-f+1}{s}\right) = \text{ceil}\left(\frac{28-5+1}{1}\right) = 24$$

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 24, 24, 64)	4864
Total params: 4,864		
Trainable params: 4,864		
Non-trainable params: 0		

Note that first dimension of output shape refers to `batch_size`.

**Example:** What is the shape of the output of a convolutional layer after we apply the following command?

```
| from tensorflow.keras import models
```

```
from tensorflow.keras.layers import Conv2D
model = models.Sequential()
model.add(Conv2D(64, 5, padding='same', activation='relu',
input_shape=(28, 28, 3)))
model.summary()
```

A:

$$w' = \text{ceil}\left(\frac{w}{s}\right) = 28$$

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 28, 28, 64)	4864
Total params: 4,864		
Trainable params: 4,864		
Non-trainable params: 0		

**Remark:** we rarely apply stride other than 1. But for fun, let us test our formula for computing output shape. What is the shape of the output of a convolutional layer after we apply the following command?

```
from tensorflow.keras import models
from tensorflow.keras.layers import Conv2D
model = models.Sequential()
model.add(Conv2D(64, 5, padding='same', strides=3, activation='relu',
input_shape=(28, 28, 3)))
model.summary()
```

A:

$$w' = \text{ceil}\left(\frac{28}{3}\right) = 10$$

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 10, 10, 64)	4864
Total params: 4,864		
Trainable params: 4,864		
Non-trainable params: 0		

## 2. Pooling Layers or Subsampling

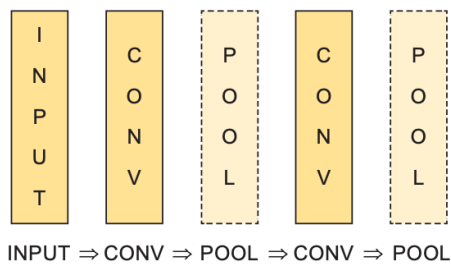
Adding more convolutional layers increases the depth of the output layer, which leads to increasing the number of parameters that the network needs to optimize (learn). You can see that adding several convolutional layers (usually tens or even hundreds) will produce a huge number of parameters (weights).

This 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



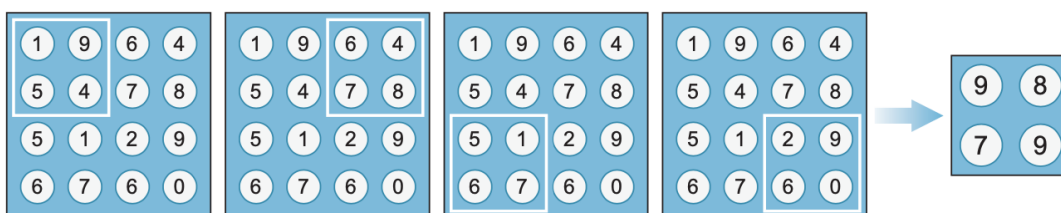
its input by applying a summary statistical function, such as a **maximum** or **average**, to reduce the overall number of parameters passed on to the next layer.

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 (see figure below).

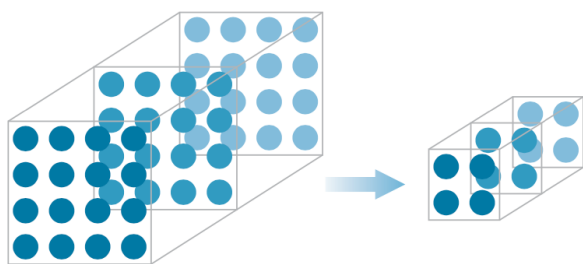


### Max Pooling

The main type of pooling layers is max pooling. 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 below, you see a pooling filter with a size of  $2 \times 2$  and strides of 2 (the filter jumps 2 pixels when sliding over the image). This pooling layer reduces the feature map size from  $4 \times 4$  down to  $2 \times 2$ .



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 (see figure below).



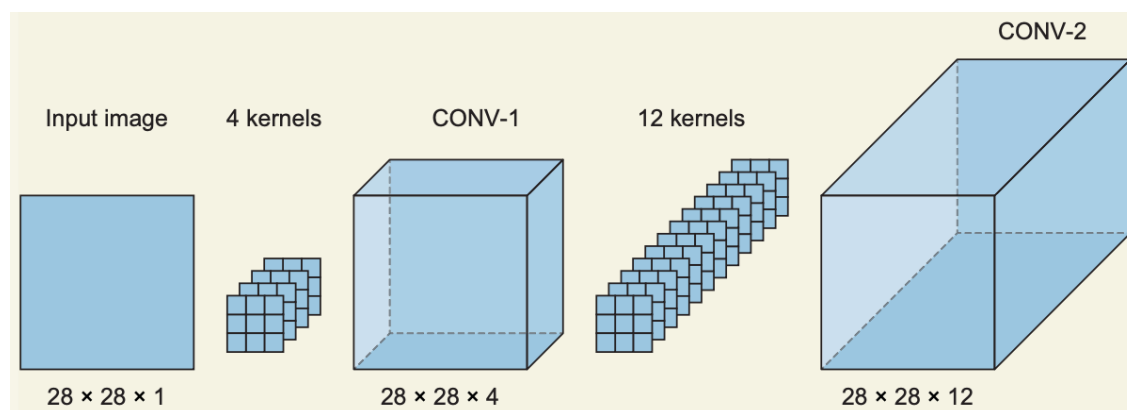
### Why Use a Pooling Layer

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:

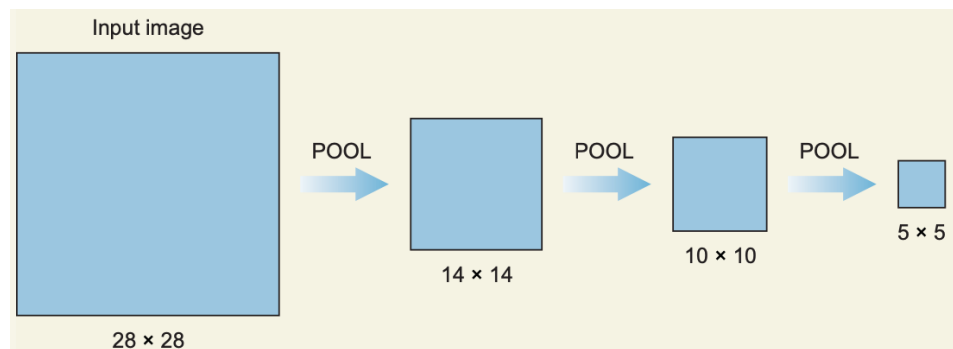


### 3. Visualize What Happens after Each Layer

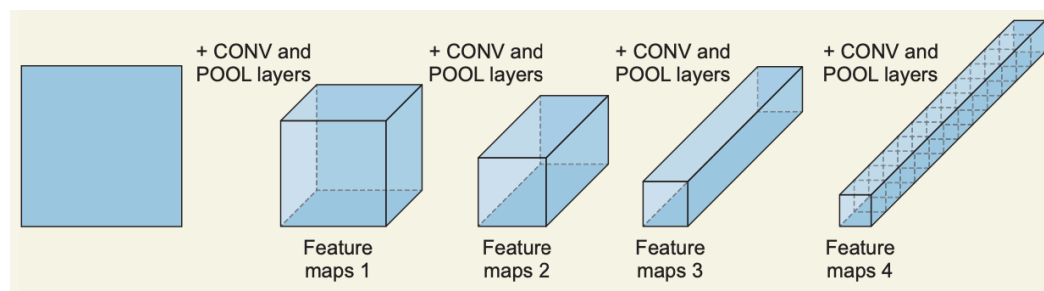
After the convolutional layers, the image keeps its width and height dimensions (usually), but it gets deeper and deeper after each layer.



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

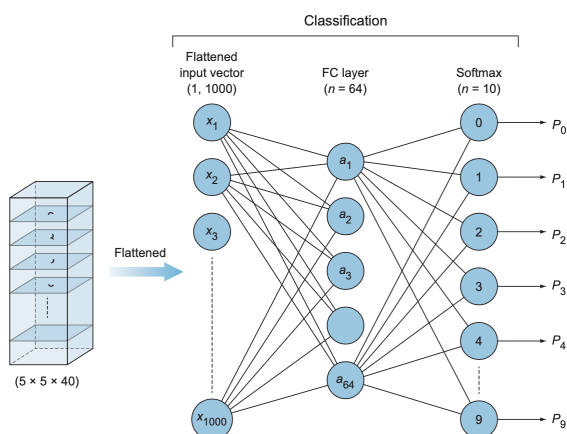


Putting the convolutional and pooling together, we get something like this:



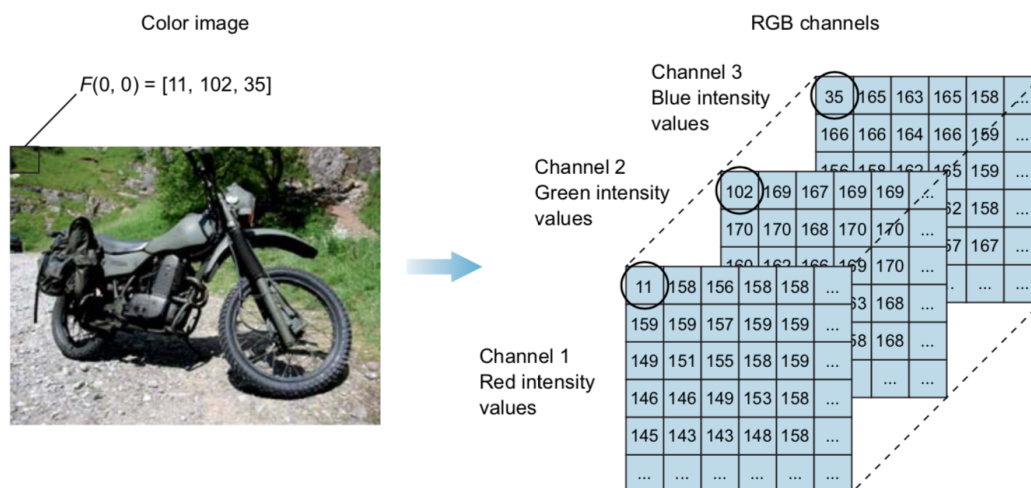
This keeps happening until we have, at the end, a long tube of small shaped images that contain all the features in the original image.

For example, the output of the convolutional and pooling layers can produce a feature tube ( $5 \times 5 \times 40$ ) that is almost ready to be classified. The last step is to flatten this tube before feeding it to the fully connected layer for classification. The flattened layer will have the dimensions of  $(1, m)$  where  $m = 5 \times 5 \times 40 = 1,000$  neurons.



## 4. Convolution Over Color Images

Color images 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.

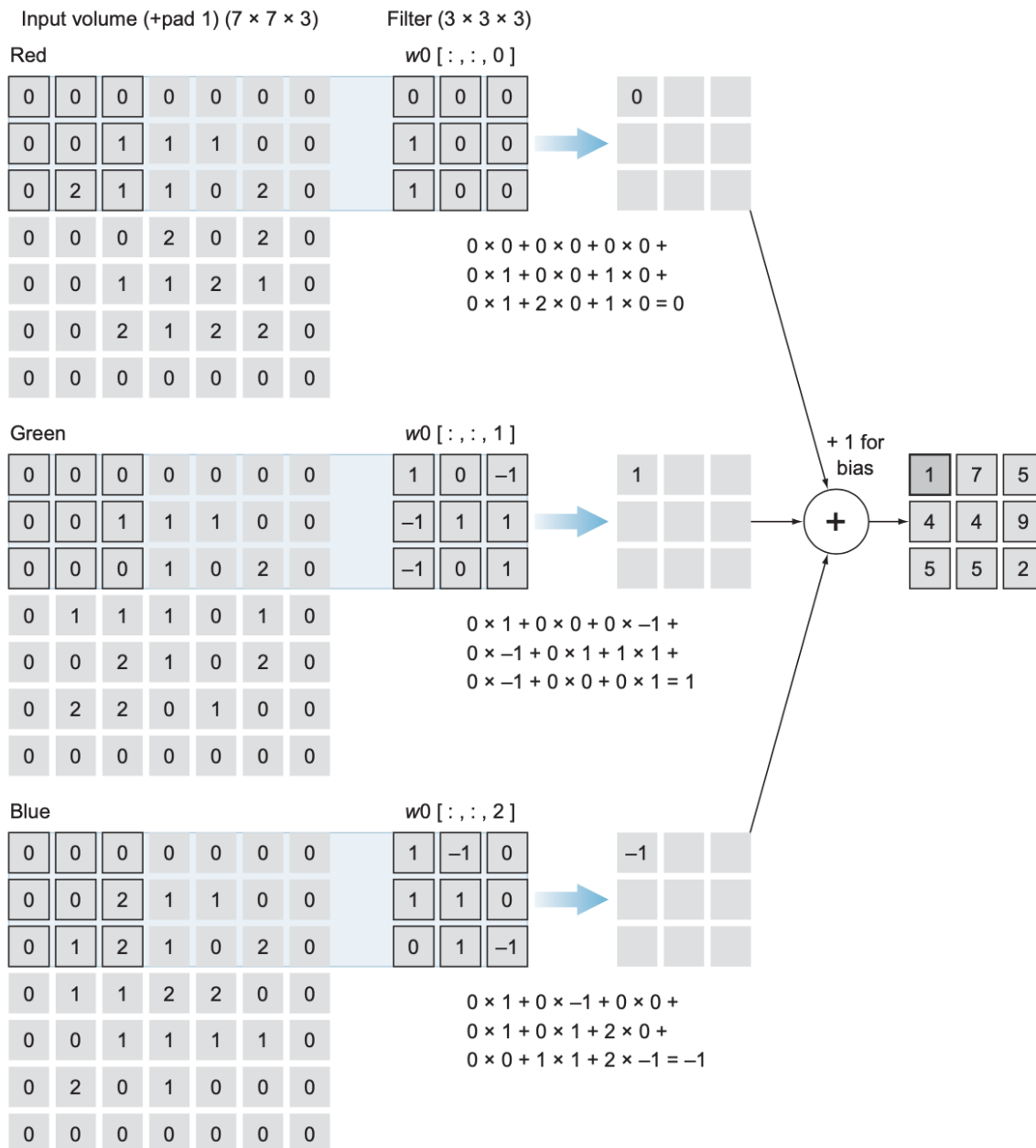


Similar to what we did with grayscale images, we slide the convolutional kernel over the image and compute the feature maps. **Now the kernel is itself three-dimensional:** one dimension for each color channel.

**Remark:** our kernel will adjust to the channels accordingly and even though we define a 3x3 kernel, the true dimensions of the kernel when initialized will be 3x3xchannels. For a RGB input, since we have 3 input channels, we will have the kernel of 3x3x3.

To perform convolution, we will do the same thing we did before. Let's see how (see figure below):

- Each of the color channels has its own corresponding filter.
- Each filter will slide over its image, multiply every corresponding pixel elementwise, 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. We continue this process until we compute the pixel values of all nodes in the feature map.



## 5. Number of Trainable Parameters

We finally show how to compute the number of trainable parameters in CNNs. During the training process, we basically learn the weights and bias from data. For example, for a  $3 \times 3$  filter with channel = 3 (RGB), the total number of the trainable parameters for a single filter is  $3 \times 3 \times 3 + 1$ .

The general equation to compute the number of training parameters is:

number of training params

=  $\text{kernel\_width} \times \text{kernel\_height} \times (\text{depth of the previous layer}) \times (\text{\#filters}) + \text{\#filters (for biases)}$

$$= (\text{kernel\_width} \times \text{kernel\_height} \times \text{depth of the previous layer} + 1) \times \text{\#filters}$$

**Example:** compute the number of training parameters for the MNIST example.

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d_1 (MaxPooling2D)	(None, 13, 13, 32)	0
conv2d_2 (Conv2D)	(None, 11, 11, 64)	18496
max_pooling2d_2 (MaxPooling2D)	(None, 5, 5, 64)	0
conv2d_3 (Conv2D)	(None, 3, 3, 64)	36928
Total params: 55,744		
Trainable params: 55,744		
Non-trainable params: 0		

The number of trainable parameters for each layer is:

$$(3 * 3 * 1 + 1) * 32 = 320$$

$$(3 * 3 * 32 + 1) * 64 = 18496$$

$$(3 * 3 * 64 + 1) * 64 = 36928$$

These numbers look daunting but if we consider the alternative to use fully connected (FC) dense layers from conv2d\_1 to conv2d\_2 (ignore the max pooling layer for now), we will need:

$$(26 * 26 * 32) * (11 * 11 * 64) + (11 * 11 * 64) = 167,525,952 !$$

## 6. Summary

At this point, you should understand the basic concepts and operations of CNNs—**filters, feature maps, convolution, and max pooling**—and you know how to build a small CNN in `Keras` to solve a toy problem such as MNIST digits classification. We are now ready to look at some more realistic examples and tricks to improve the performance of CNNs.